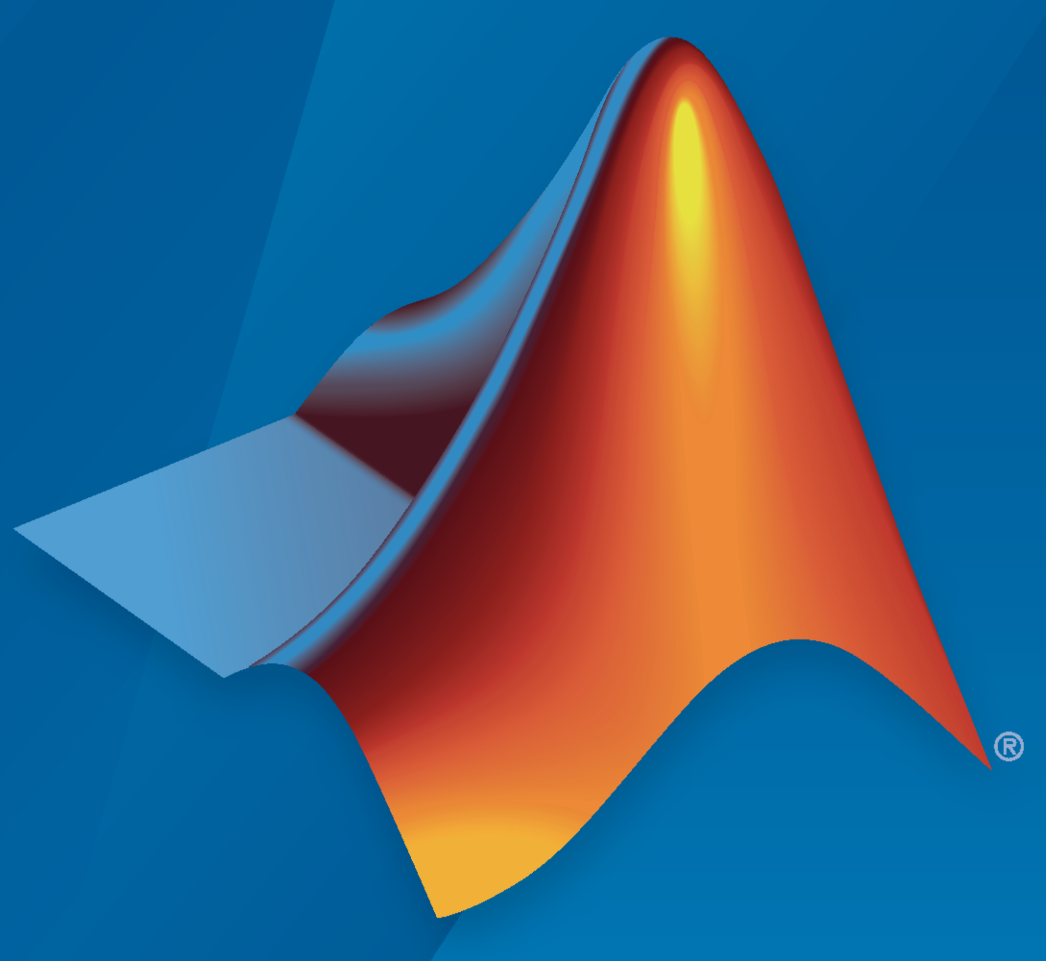
HDL Coder™

User's Guide



MATLAB® & SIMULINK®

R2024a

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

HDL Coder™ User's Guide

© COPYRIGHT 2012–2024 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

March 2012	Online only	New for Version 3.0 (R2012a)
September 2012	Online only	Revised for Version 3.1 (R2012b)
March 2013	Online only	Revised for Version 3.2 (R2013a)
September 2013	Online only	Revised for Version 3.3 (R2013b)
March 2014	Online only	Revised for Version 3.4 (R2014a)
October 2014	Online only	Revised for Version 3.5 (R2014b)
March 2015	Online only	Revised for Version 3.6 (R2015a)
September 2015	Online only	Revised for Version 3.7 (R2015b)
October 2015	Online only	Rereleased for Version 3.6.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.8 (R2016a)
September 2016	Online only	Revised for Version 3.9 (R2016b)
March 2017	Online only	Revised for Version 3.10 (Release 2017a)
September 2017	Online only	Revised for Version 3.11 (R2017b)
March 2018	Online only	Revised for Version 3.12 (Release 2018a)
September 2018	Online only	Revised for Version 3.13 (Release 2018b)
March 2019	Online only	Revised for Version 3.14 (Release 2019a)
September 2019	Online only	Revised for Version 3.15 (Release 2019b)
March 2020	Online only	Revised for Version 3.16 (Release 2020a)
September 2020	Online only	Revised for Version 3.17 (Release 2020b)
March 2021	Online only	Revised for Version 3.18 (Release 2021a)
September 2021	Online only	Revised for Version 3.19 (Release 2021b)
March 2022	Online only	Revised for Version 3.20 (Release 2022a)
September 2022	Online only	Revised for Version 4.0 (Release 2022b)
March 2023	Online only	Revised for Version 4.1 (Release R2023a)
September 2023	Online only	Revised for Version 23.2 (R2023b)
March 2024	Online only	Revised for Version 24.1 (R2024a)

Contents

HDL Code Generation from MATLAB

1 MATLAB Algorithm Design

Functions Supported for HDL and HLS Code Generation **1-2**
Supported MATLAB and Fixed Point Runtime Library Functions **1-2**
Fixed-Point Function Limitations **1-2**

Supported MATLAB Data Types, Operators, and Control Flow Statements **1-4**
Supported Data Types **1-4**
Supported Operators **1-5**
Control Flow Statements **1-7**

Persistent Variables and Persistent Array Variables **1-9**
Persistent Variables **1-9**
Persistent Array Variables **1-9**

Complex Data Type Support **1-11**
Declaring Complex Signals **1-11**
Conversion Between Complex and Real Signals **1-12**
Support for Vectors of Complex Numbers **1-12**

Complex Data Type Support for High-Level Synthesis Code Generation **1-14**
Declaring Complex Signals **1-14**
Conversion Between Complex and Real Signals **1-14**
Support for Vectors of Complex Numbers **1-15**

HDL Code Generation for System Objects **1-16**
Why Use System Objects? **1-16**
Predefined System Objects **1-16**
User-Defined System Objects **1-16**
Limitations of HDL Code Generation for System Objects **1-16**
System object Examples for HDL Code Generation **1-17**

HDL Code Generation from System Objects **1-18**

HDL Code Generation for Streaming Matrix Inverse System Object **1-22**

HDL Code Generation for Streaming Matrix Multiply System Object **1-31**

HDL Code Generation from hdl.RAM System Object **1-39**

HDL Code Generation from a Non-Restoring Square Root System Object **1-42**

HDL Code Generation from Viterbi Decoder System Object **1-46**

Predefined System Objects Supported for HDL Code Generation . . **1-50**
Predefined System Objects in MATLAB Code **1-50**
Predefined System Objects in the MATLAB System Block **1-51**

Load constants from a MAT-File **1-52**

Generate Code for User-Defined System Objects **1-53**
How To Create A User-Defined System object **1-53**
User-Defined System object Example **1-53**

Map Matrices to ROM **1-55**

Model State with Persistent Variables and System Objects **1-56**

Bitwise Operations in MATLAB for HDL and HLS Code Generation **1-58**
Bit Shifting and Rotation **1-58**
Bit Slicing and Bit Concatenation **1-60**

Mapping of Different Rounding and Overflow Methods from MATLAB to HLS **1-61**

Handling Constants in HDL and HLS Code Generation **1-63**
Specify Constants in Generated Code **1-63**
Load Compile-Time Constants from MAT-file **1-63**
Generate HDL or HLS Code **1-64**

Structure Definition for HLS Code Generation **1-65**
Limitations **1-65**

Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code **1-66**
MATLAB Design Requirements for HDL and HLS Code Generation **1-66**
Guidelines for Writing MATLAB Code **1-66**

Indexing Best Practices for HDL Code Generation **1-68**
Minimize Automatic Index Conversions **1-68**
Example of Area Consumption Optimization **1-68**

For-Loop Best Practices for HDL Code Generation **1-73**
Monotonically Increasing Loop Counters **1-73**
Find Indices Using Loops **1-73**
Persistent Variables in Loops **1-74**
Persistent Arrays in Loops **1-74**

MATLAB Test Bench Requirements and Best Practices for Code Generation . . . 1-76
What Is a MATLAB Test Bench? . . . 1-76
MATLAB Test Bench Requirements . . . 1-76
MATLAB Test Bench Best Practices . . . 1-76

2 MATLAB to HDL Examples for Communications and Signal Processing Applications

HDL Code Generation for LMS Filter . . . 2-2
Bisection Algorithm to Calculate Square Root of an Unsigned Fixed-Point Number . . . 2-8
Timing Offset Estimation . . . 2-12
Data Packetization . . . 2-16
Transmit and Receive FIFO Registers . . . 2-22
HDL Code Generation for Harris Corner Detection Algorithm . . . 2-29
HDL Code Generation for Adaptive Median Filter . . . 2-36
Contrast Adjustment . . . 2-43
Image Enhancement by Histogram Equalization . . . 2-51
HDL Code Generation for Image Format Conversion from RGB to YUV . . . 2-56
High Dynamic Range Imaging . . . 2-60
Accelerate Pixel-Streaming Designs Using MATLAB Coder . . . 2-65
Enhanced Edge Detection from Noisy Color Video . . . 2-68
Verify Sobel Edge Detection Algorithm in MATLAB-to-HDL Workflow . . . 2-71

3 MATLAB Best Practices and Design Patterns for HDL Code Generation

Model a Counter for HDL and High-Level Synthesis Code Generation . . . 3-2
MATLAB Code for the Counter . . . 3-2

Model a State Machine for HDL and High-Level Synthesis Code Generation . . . 3-4
MATLAB Code for the Mealy State Machine . . . 3-4
MATLAB Code for the Moore State Machine . . . 3-6

Generate Hardware Instances For Local Functions . . . 3-8
MATLAB Local Functions . . . 3-8
MATLAB Code for mlhdlc_two_counters.m . . . 3-8

Implement RAM Using MATLAB Code . . . 3-10
Implement RAM Using a Persistent Array or System object Properties . . . 3-10
Implement RAM Using hdl.RAM . . . 3-11

4 Fixed-Point Conversion

Specify Type Proposal Options . . . 4-2

Log Data for Histogram . . . 4-5

View and Modify Variable Information . . . 4-7
View Variable Information . . . 4-7
Modify Variable Information . . . 4-7
Revert Changes . . . 4-8
Promote Sim Min and Sim Max Values . . . 4-8

Automated Fixed-Point Conversion . . . 4-9
License Requirements . . . 4-9
Automated Fixed-Point Conversion Capabilities . . . 4-9
Code Coverage . . . 4-10
Proposing Data Types . . . 4-12
Locking Proposed Data Types . . . 4-14
Viewing Functions . . . 4-14
Viewing Variables . . . 4-14
Histogram . . . 4-19
Function Replacements . . . 4-20
Validating Types . . . 4-21
Testing Numerics . . . 4-21
Detecting Overflows . . . 4-21

Custom Plot Functions . . . 4-23

Edit Configuration Parameters for Fixed-Point Code Generation . . . 4-24
Create and Modify Configuration Objects . . . 4-24
Additional Functionalities . . . 4-24

Visualize Differences Between Floating-Point and Fixed-Point Results . . . 4-26

Inspecting Data Using the Simulation Data Inspector . . . 4-31
What Is the Simulation Data Inspector? . . . 4-31

Import Logged Data 4-31
Export Logged Data 4-31
Group Signals 4-31
Run Options 4-31
Create Report 4-32
Comparison Options 4-32
Enabling Plotting Using the Simulation Data Inspector 4-32
Save and Load Simulation Data Inspector Sessions 4-32

Enable Plotting Using the Simulation Data Inspector 4-33
From the UI 4-33
From the Command Line 4-33

Replacing Functions Using Lookup Table Approximations 4-34

Replace a Custom Function with a Lookup Table 4-35
Using the HDL Coder App 4-35
From the Command Line 4-38

Replace the exp Function with a Lookup Table 4-41
From the UI 4-41
From the Command Line 4-44

Data Type Issues in Generated Code 4-47
Enable the Highlight Option in a Project 4-47
Enable the Highlight Option at the Command Line 4-47
Stowaway Doubles 4-47
Stowaway Singles 4-47
Expensive Fixed-Point Operations 4-47

Working with Fixed-Point Code 4-49

Floating-Point to Fixed-Point Conversion 4-51

Fixed-Point Type Conversion and Refinement 4-61

Working with Generated Fixed-Point Files 4-68

Fixed-Point Type Conversion and Derived Ranges 4-73

Generate HDL-Compatible Lookup Table Function Replacements Using coder.approximate 4-78

5 Code Generation

Create and Set Up Your Project 5-2
Create a New Project 5-2
Open an Existing Project 5-3
Add Files to the Project 5-3

Specify Properties of Entry-Point Function Inputs **5-4**
When to Specify Input Properties **5-4**
Why You Must Specify Input Properties **5-4**
Methods for Defining Properties of Primary Inputs **5-4**
Properties to Specify **5-5**
Rules for Specifying Properties of Primary Inputs **5-6**

Code Generation Reports **5-7**
Report Generation **5-7**
Report Location **5-8**
Errors and Warnings **5-8**
Files and Functions **5-8**
MATLAB Source **5-8**
MATLAB Variables **5-9**
Additional Reports **5-10**
Report Limitations **5-10**

Generate Instantiable Code for Functions **5-12**
How to Generate Instantiable Code for Functions **5-12**
Generate Code Inline for Specific Functions **5-12**
Limitations for Instantiable Code Generation for Functions **5-12**

Edit Configuration Parameters for HDL Coder **5-13**
Create and Modify Configuration Objects **5-13**
Additional Functionalities **5-13**

Integrate Custom HDL Code Into MATLAB Design **5-15**
Define the hdl.BlackBox System object **5-15**
Use System object In MATLAB Design Function **5-16**
Generate HDL Code **5-17**
Limitations for hdl.BlackBox **5-19**

Enable MATLAB Function Block Generation **5-20**
Requirements for MATLAB Function Block Generation **5-20**
Enable MATLAB Function Block Generation **5-20**
Restrictions for MATLAB Function Block Generation **5-20**
Results of MATLAB Function Block Generation **5-20**

System Design with HDL Code Generation from MATLAB and Simulink **5-21**

Specify the Clock Enable Rate **5-24**
Why Specify the Clock Enable Rate? **5-24**
How to Specify the Clock Enable Rate **5-24**

Specify Test Bench Clock Enable Toggle Rate **5-26**
When to Specify Test Bench Clock Enable Toggle Rate **5-26**
How to Specify Test Bench Clock Enable Toggle Rate **5-26**

Generate an HDL Coding Standard Report from MATLAB **5-28**
Using the HDL Workflow Advisor **5-28**
Using the Command Line **5-30**

Generate an HDL Lint Tool Script **5-31**
How To Generate an HDL Lint Tool Script **5-31**

Generate HDL Code from MATLAB Functions That Use Automated Lookup Table Generation **5-33**

Generate Board-Independent IP Core from MATLAB Algorithm . . . **5-38**
Requirements and Limitations for IP Core Generation **5-38**
Generate Board-Independent IP Core **5-38**

Minimize Clock Enables **5-40**
Using the GUI **5-40**
Using the Command Line **5-40**
Limitations **5-41**

6 Verification

Verify Code with HDL Test Bench **6-2**

Test Bench Generation **6-5**
How Test Bench Generation Works **6-5**
Test Bench Data Files **6-5**
Test Bench Data Type Limitations **6-5**
Use Constants Instead of File I/O **6-5**

Resolve Index Errors During Simulation **6-6**
Issue **6-6**
Possible Solutions **6-6**

7 Deployment

Generate Synthesis Scripts **7-2**

8 Optimization

Map Matrices to Block RAMs to Reduce Area **8-2**

Map Persistent Arrays and dsp.Delay Objects to RAM **8-6**
Enable RAM Mapping **8-6**
RAM Mapping Requirements for Persistent Arrays and System object Properties **8-7**
RAM Mapping Requirements for dsp.Delay System Objects **8-9**
RAM Mapping Comparison **8-9**

Pipelining MATLAB Code **8-11**
Port Registers **8-11**

Input and Output Pipeline Registers 8-11
Operation Pipelining 8-11

Pipeline MATLAB Expressions **8-12**
How To Pipeline a MATLAB Expression 8-12
Limitations of Pipelining for MATLAB Expressions 8-12

Distributed Pipelining **8-14**
What is Distributed Pipelining? 8-14
Benefits and Costs of Distributed Pipelining 8-14
Selected Bibliography 8-14

Distributed Pipelining for Clock Speed Optimization **8-15**

Optimize Clock Speed for MATLAB Code by Using Adaptive Pipelining **8-19**

Optimize Feedback Loop Design and Maintain High Data Precision for HDL Code Generation **8-26**

Optimize MATLAB Loops **8-29**
Loop Streaming 8-29
Loop Unrolling 8-29
How to Optimize MATLAB Loops 8-29
Limitations for MATLAB Loop Optimization 8-30

Constant Multiplier Optimization **8-31**
What is Constant Multiplier Optimization? 8-31
Specify Constant Multiplier Optimization 8-31

Resource Sharing of Multipliers to Reduce Area **8-33**

Loop Streaming to Reduce Area **8-39**

Constant Multiplier Optimization to Reduce Area **8-44**

9 HDL Workflow Advisor Reference

HDL Workflow Advisor **9-2**
Overview 9-2

MATLAB to HDL Code and Synthesis **9-6**
MATLAB to HDL Code Conversion 9-6
Code Generation: Target Tab 9-6
Code Generation: Coding Style Tab 9-7
Code Generation: Clocks and Ports Tab 9-8
Code Generation: Test Bench Tab 9-10
Code Generation: Optimizations Tab 9-11
Simulation and Verification 9-12
Synthesis and Analysis 9-13

10 MATLAB to HLS Examples for Communications and Signal Processing Applications

High-Level Synthesis Code Generation for LMS Filter 10-2

High-Level Synthesis Code Generation for Bisection Algorithm . . . 10-8

High-Level Synthesis Code Generation for Data Packetization . . . 10-12

High-Level Synthesis Code Generation for DF2T Filter 10-18

High-Level Synthesis Code Generation for Transmit and Receive FIFO Registers 10-21

High-Level Synthesis Code Generation for Contrast Adjustment 10-28

High-Level Synthesis Code Generation for Image Format Conversion from RGB to YUV 10-37

High-Level Synthesis Code Generation for Advanced Encryption Standard 10-41

11 High-Level Synthesis Code Generation

Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App 11-5

Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface 11-14

Replace Arithmetic Operation to Generate Efficient HDL and High-Level Synthesis Code 11-18

HLS Code Generation Report 11-29
- Report Generation 11-29
- Files and Functions 11-29
- MATLAB Source 11-30
- MATLAB Variables 11-31
- Tracing Code 11-32
- Code Insights 11-33
- Additional Reports 11-33
- Report Limitations 11-35

12 Verification

Verify HLS Code That Has an HDL Test Bench **12-2**

MATLAB to HLS Code Generation Options **12-4**
MATLAB to HLS Code Generation **12-4**
HLS Code Generation: Target Tab **12-4**
HLS Code Generation: Coding Style Tab **12-4**
HLS Code Generation: Clocks & Ports Tab **12-5**
HLS Code Generation: Optimizations Tab **12-6**
HLS Code Generation: Advance Tab **12-6**

Floating-Point Tolerance Parameters **12-8**
Tolerance Strategy **12-8**
Tolerance Value **12-8**

Verify Generated HLS Code Using MATLAB Desktop Host **12-9**

13 HLS Optimizations

Map Persistent Arrays to RAM **13-2**
Enable RAM Mapping **13-2**
RAM Mapping Requirements for Persistent Arrays **13-2**

Pipelining of for-Loops **13-4**
Issues with Pipelined for-Loops **13-5**

Map Persistent Variables to RAM for Histogram Equalization **13-9**

Create a Line Buffer Interface for SystemC Code Generation **13-16**

HDL Code Generation from Simulink

14 Model Design for HDL Code Generation

Signal and Data Type Support **14-3**
Buses **14-3**
Enumerations **14-4**
Matrices **14-4**
Unsupported Signal and Data Types **14-7**

Use Simulink Templates for HDL Code Generation **14-8**
Create Model Using HDL Coder Model Template **14-8**
HDL Coder Model Templates **14-8**

Generate DUT Ports for Tunable Parameters **14-18**
Prerequisites **14-18**
Create and Add Tunable Parameter That Maps to DUT Ports **14-18**
Generated Code **14-19**
Limitations **14-20**
Use Tunable Parameter in Other Blocks **14-20**

Generate Parameterized Code for Referenced Models **14-21**
Parameterize Referenced Model for HDL Code Generation **14-21**
Restrictions **14-21**

Generating HDL Code for Subsystems with Array of Buses **14-22**
How HDL Coder Generates Code for Array of Buses **14-22**
Array of Buses Limitations **14-24**

Generate HDL Code with Record or Structure Types for Bus Signals
. . . . **14-25**
Requirements and Considerations **14-25**
Use Record or Structure Types for Bus in HDL Code Generation . **14-26**
Generate Record Types for Bus Signals at Subsystem Interface . . **14-26**
Generate Record Types for Array of Bus **14-28**
Limitations **14-30**

Implement Control Signal-Based Mathematical Functions by Using HDL Coder **14-31**

Implement Sqrt Block with Control Signals **14-33**

Implement Reciprocal Block with Control Signals **14-37**

Implement rSqrt Block with Control Signals **14-41**

Implement Divide Block with Control Signals **14-45**

Implement Sine and Cosine Block with Control Signals **14-49**

Implement Atan2 Block with Control Signals **14-53**

Using ForEach Subsystems in HDL Coder **14-57**

Generate HDL Code for Blocks Inside For Each Subsystem **14-61**

Field-Oriented Control of a Permanent Magnet Synchronous Machine **14-66**

Model and Debug Test Point Signals with HDL Coder **14-71**

Optimize Generated HDL Code for Multirate Designs with Large Rate Differentials **14-80**
Issue **14-80**
Description **14-80**

Recommendations 14-82

Getting Started with HDL Coder Native Floating-Point Support . 14-88
Key Features 14-88
Numeric Considerations and IEEE-754 Standard Compliance 14-88
Floating Point Types 14-89
Data Type Considerations 14-90

Numeric Considerations for Native Floating-Point 14-92
Nearest Even Digit Rounding 14-92
Denormal Numbers 14-92
Exception Handling 14-93
Relative Accuracy and ULP Considerations 14-93

ULP Considerations of Native Floating-Point Operators 14-95
Adherence of Native Floating Point Operators to IEEE-754 Standard 14-95
ULP Values of Floating Point Operators 14-95
Considerations 14-97

Latency Values of Floating-Point Operators 14-99
Math Operations 14-99
Trigonometric Operations 14-101
Comparisons and Conversions 14-102

Latency Considerations with Native Floating Point 14-104

Generate Target-Independent HDL Code with Native Floating-Point 14-111
How HDL Coder Generates Target-Independent HDL Code 14-111
Enable Native Floating Point and Generate Code 14-112
View Code Generation Report 14-113
Analyze Results 14-114
Limitation 14-116

Floating Point Support: Field-Oriented Control Algorithm 14-118

Design Model by Using HDL Coder Native Floating Point and Intel Hard Floating Point 14-125

Verify the Generated Code from Native Floating-Point 14-133
Specify the Tolerance Strategy 14-133
Verify the Generated Code with HDL Test Bench 14-134
Verify the Generated Code with Cosimulation 14-134
Limitation 14-136

Simulink Blocks Supported by Using Native Floating Point 14-137
HDL Floating Point Operations Library 14-137
Supported Simulink Blocks in Math Operations Library 14-138
Supported Functions in Math Function Block 14-139
Supported Simulink Blocks in Other Libraries 14-139
Simulink Block Restrictions 14-141

Synthesis Benchmark of Common Native Floating Point Operators 14-143

Supported Data Types and Scope 14-155
Supported Data Types 14-155
Unsupported Data Types 14-156
Scope for Variables 14-156

Supported Synthesizable RTL Constructs and keywords in HDL Coder 14-157
Supported VHDL Constructs 14-157
Supported Verilog Constructs 14-160
Supported VHDL Keywords 14-163
Supported Verilog Keywords 14-164

Import Verilog Code and Generate Simulink Model 14-165
HDL Import 14-165
HDL Import Requirements 14-165
Import HDL Code 14-165
Model Location 14-166
Errors and Warnings 14-166
Limitations of Verilog HDL Import 14-167

Supported Verilog Constructs for HDL Import 14-169
Module Definition and Instantiations 14-169
Data Types and Vectors 14-170
Identifiers and Comments 14-170
Assignments 14-171
Operators 14-171
Conditional and Looping Statements 14-172
Procedural Blocks and Events 14-172
Other Constructs 14-172

Verilog Dataflow Modeling with HDL Import 14-174
Supported Verilog Dataflow Patterns 14-174
Unsupported Verilog Dataflow Patterns 14-176

Simulate and Generate HDL Code for the Float Typecast Block 14-185

Generate Simulink Model from CORDIC Atan2 Verilog Code . . . 14-187

15 Simulink to HDL Examples for Communication and Signal Processing Applications

Programmable FIR Filter for FPGA 15-2

Multichannel FIR Filter for FPGA 15-7

High-Throughput Channelizer for FPGA 15-10

Implement Digital Downconverter for FPGA 15-20

Implement Digital Upconverter for FPGA 15-38

HDL QAM Transmitter and Receiver **15-58**

Airplane Tracking with ADS-B Captured Data **15-78**

Generate HDL Code for Viterbi Decoder **15-85**

Design Video Processing Algorithms for HDL in Simulink **15-95**

Edge Detection and Image Overlay **15-102**

Lane Detection **15-107**

HDL QPSK Transmitter and Receiver **15-125**

16 Code Generation Options in the HDL Coder Dialog Boxes

Set HDL Code Generation Options **16-2**
HDL Code Generation Options in the Configuration Parameters Dialog Box **16-2**
HDL Code Tab in Simulink Toolstrip **16-3**
HDL Code Options in the Block Context Menu **16-4**
The HDL Block Properties Dialog Box **16-4**

Generate HDL Code from Simulink Model Using Configuration Parameters **16-6**
FIR Filter Model **16-6**
Create a Folder and Copy Relevant Files **16-7**
Open HDL Code Generation Pane of Configuration Parameters Dialog Box **16-8**
Generate HDL Code **16-8**

Generate HDL Code from Simulink Model from Command Line . **16-10**
FIR Filter Model **16-10**
Create a Folder and Copy Relevant Files **16-11**
Generate HDL Code **16-12**

17 HDL Code Generation Pane: General

18 Modeling Guidelines

HDL Modeling Guidelines Severity Levels **18-3**

Model Design and Compatibility Guidelines - By Numbered List 18-4
Guidelines 1.1: Basic Settings 18-4
Guidelines 1.2: DUT Subsystem and Hierarchical Modeling 18-5
Guidelines 1.3: Vectors, Matrices, and Buses 18-5
Guidelines 1.4: Clock Bundle Signals 18-6
Guidelines 1.5: Native Floating Point 18-6

Guidelines for Supported Blocks and Data Types - By Numbered List 18-7
Guidelines 2.1: HDL RAMs and HDL Operations Library 18-7
Guidelines 2.2: Logic and Bit Operations Library 18-7
Guidelines 2.3: Lookup Table and Signal Routing Blocks 18-7
Guidelines 2.4: Ports and Subsystems 18-8
Guideline 2.5: Rate Change and Constant Blocks 18-8
Guideline 2.6: Delay Blocks 18-9
Guideline 2.7: Multiplication and Accumulation Operations 18-9
Guideline 2.8: MATLAB Function Blocks 18-10
Guideline 2.9: Stateflow Charts 18-10
Guidelines 2.10: Data Types 18-11
Guidelines 2.11: Square Root Operations 18-11

Guidelines for Speed and Area Optimizations - By Numbered List 18-12
Guidelines 3.1: Resource Sharing and Streaming 18-12
Guidelines 3.2: Clock Rate Pipelining and Distributed Pipelining . . 18-13

Basic Guidelines for Modeling HDL Algorithm in Simulink 18-14
Use HDL-Supported Blocks 18-14
Partition Model into DUT and Test Bench 18-15
Avoid Using Double-Byte Characters 18-17
Document Model Features and Attributes 18-17

Guidelines for Model Setup and Checking Model Compatibility . . 18-20
Customize hdlsetup Function Based on Target Application 18-20
Check Subsystem for HDL Compatibility 18-21
Run Model Checks for HDL Coder 18-21

Modeling with Simulink, Stateflow, and MATLAB Function Blocks 18-24
Guideline ID 18-24
Severity 18-24
Description 18-24

Terminate Unconnected Block Outputs and Usage of Commenting Blocks 18-27
Terminate Unconnected Block Outputs 18-27
Using Comment Out and Comment Through of Blocks 18-28

Identify and Programmatically Change and Display HDL Block Parameters 18-31
Adjust Sizes of Constant and Gain Blocks for Identifying Parameters 18-31
Display Parameters That Affect HDL Code Generation 18-31
Change Block Parameters by Using find_system and set_param . . 18-36

DUT Subsystem Guidelines **18-37**
DUT Subsystem Considerations **18-37**
Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks **18-37**
Insert Handwritten Code into Simulink Modeling Environment . . . **18-39**

Hierarchical Modeling Guidelines **18-41**
Avoid Constant Block Connections to Subsystem Port Boundaries **18-41**
Generate Parameterized HDL Code for Constant and Gain Blocks **18-42**
Place Physical Signal Lines Inside a Subsystem **18-44**

Design Considerations for Matrices and Vectors **18-47**
Modeling Requirements for Matrices **18-47**
Avoid Generating Ascending Bit Order in HDL Code From Vector Signals **18-48**

Use Bus Signals to Improve Readability of Model and Generate HDL Code **18-52**

Guidelines for Clock and Reset Signals **18-58**
Use Global Oversampling to Create Frequency-Divided Clock **18-58**
Create Multirate Model with Integer Clock Multiples by Clock Division **18-58**
Use Dual Rate Dual Port RAM for Noninteger Multiple Sample Times **18-61**
Asynchronous Clock Modeling in HDL Coder **18-62**
Use Global Reset Type Setting Based on Target Hardware **18-64**

Modeling with Native Floating Point **18-65**
Guideline ID **18-65**
Severity **18-65**
Description **18-65**

Design Considerations for RAM Blocks and Blocks in HDL Operations Library **18-68**
RAM Block Access Considerations **18-68**
Serial to Parallel Conversion **18-70**

Usage of Blocks in Logic and Bit Operations Library **18-72**
Logical and Arithmetic Bit Shift Operations **18-72**
Usage of Logical Operator, Bitwise Operator, and Bit Reduce Blocks **18-74**
Use Boolean Output for Compare to Constant and Relational Operator Blocks **18-76**

Generate FPGA Block RAM from Lookup Tables **18-78**

Recommended Block Parameter Settings of Multiport Switch Block for Numeric and Enumerated Types **18-83**

Guidelines for Using Selector Blocks to Extract Input Elements from Vector or Matrix Signals **18-87**
Guideline ID **18-87**

Severity . . . 18-87
Description . . . 18-87

Guidelines for Using Assignment Blocks to Write Elements in Vectors, Matrices, and 3-D Arrays . . . **18-91**
Guideline ID . . . 18-91
Severity . . . 18-91
Description . . . 18-91

Usage of Different Subsystem Types . . . **18-93**
Virtual Subsystem: Use as DUT . . . 18-93
Atomic and Virtual Subsystems: Generate Reusable HDL Files . . . 18-93
Variant Subsystem: Using Variant Subsystems for HDL Code Generation . . . 18-94
Model References: Build Model Design Using Smaller Partitions . . . 18-96
Block Settings of Enabled and Triggered Subsystems . . . 18-98

Usage of Rate Change and Constant Blocks . . . **18-100**
Usage of Rate Conversion Blocks . . . 18-100
Use Discrete and Finite Sample Time for Constant Block . . . 18-101
Dynamically Change Sample Offset for Downsample Block . . . 18-103

Guidelines for Using Delays and Goto and From Blocks for HDL Code Generation . . . **18-105**
Appropriate Usage of Delay Blocks as Registers . . . 18-105
Absorb Delays to Avoid Timing Difference . . . 18-105
Map Large Delays to Block RAM . . . 18-106
Required HDL Settings for Goto and From Blocks . . . 18-107

Modeling Efficient Multiplication and Division Operations for FPGA Targeting . . . **18-109**
Designing Multipliers and Adders for Efficient Mapping to DSP Blocks on FPGA . . . 18-109
Set ConstMultiplierOptimization HDL Block Property to auto for Gain Block . . . 18-113
Use ShiftAdd Architecture of Divide Block for Fixed-Point Types 18-115
Use Gain Block for Fixed-Point Constant Operations . . . 18-115

Using Persistent Variables and fi Objects Inside MATLAB Function Blocks for HDL Code Generation . . . **18-117**
Update Persistent Variables at End of MATLAB Function . . . 18-117
Avoid Algebraic Loop Errors from Persistent Variables inside MATLAB Function Blocks . . . 18-118
Use hdlfimath Setting and Specify fi Objects inside MATLAB Function Block . . . 18-120

Guidelines for HDL Code Generation Using Stateflow Charts . . . **18-123**
Choose State Machine Type based on HDL Implementation Requirements . . . 18-123
Specify Block Configuration Settings of Stateflow Chart . . . 18-123
Insert Unconditional Transition State for Else Statement in HDL Code . . . 18-124
Data Type Settings and Casting in Stateflow Chart for HDL Code Generation . . . 18-127
Using Absolute Time Temporal Logic in Stateflow Charts . . . 18-129

Modeling Error (default) State in Stateflow Charts **18-129**
Enable Clock-Driven Outputs of Stateflow Charts (Moore Charts Only) **18-130**
Enumeration type for active state monitoring in a Stateflow chart with no default value **18-131**

Simulink Data Type Considerations **18-134**
Use Boolean for Logical Data and Ufix1 for Numerical Data **18-134**
Specify Data Type of Gain Blocks **18-134**
Enumerated Data Type Restrictions **18-135**
Choose Optimal Simulink Block to Compute Sine and Cosine Functions with Fixed-Point Data Types **18-135**
Choose Optimal Simulink Block to Compute Sine and Cosine Functions with Floating-Point Data Types **18-140**

Guidelines for Using Rounding and Saturation Settings for Fixed-Point Data Types **18-144**
Guideline ID **18-144**
Severity **18-144**
Description **18-144**

Guideline for Using Sqrt Block for HDL Code Generation **18-146**
Use SqrtFunction Architecture for Square Root Block **18-146**

Resource Sharing Settings for Various Blocks **18-148**
Resource Sharing of Add Blocks **18-148**
Resource Sharing of Gain Blocks **18-149**
Resource Sharing of Product Blocks **18-150**
Resource Sharing of Multiply-Add Blocks **18-150**

Resource Sharing of Subsystems and Floating-Point IPs **18-152**
General Considerations for Sharing of Subsystems **18-152**
Use MATLAB Datapath Architecture for Sharing with MATLAB Function Blocks **18-153**
Sharing of Subsystems **18-153**
Resource Sharing of Floating-Point IPs **18-154**

Resource Sharing Guidelines for Vector Processing and Matrix Multiplication **18-156**
Use StreamingFactor for Resource Sharing of Vector Signals . . . **18-156**
Use SharingFactor and HDL Block Properties for Sharing Matrix Multiplication Operations **18-159**

Distributed Pipelining and Clock-Rate Pipelining Guidelines . . . **18-161**
Clock-Rate Pipelining Guidelines **18-161**
Recommended Distributed Pipelining Settings **18-161**

Insert Distributed Pipeline Registers for Blocks with Vector Data Type Inputs **18-164**
Guideline ID **18-164**
Severity **18-164**
Description **18-164**

19 Supported Blocks Library and Block Properties

View HDL-Supported Blocks and HDL-Specific Block Documentation **19-2**
View HDL-Supported Blocks and Documentation **19-2**
View HDL-Specific Block Documentation **19-2**

HDL Block Properties: General **19-3**
Overview **19-3**
AdaptivePipelining **19-4**
AllowDelayDistribution **19-5**
BalanceDelays **19-5**
ClockRatePipelining **19-6**
CodingStyle **19-7**
ConstMultiplierOptimization **19-8**
ConstrainedOutputPipeline **19-8**
DistributedPipelining **19-9**
DotProductStrategy **19-10**
DSPStyle **19-11**
FlattenHierarchy **19-13**
GuardIndexVariables **19-14**
InputPipeline **19-14**
InstantiateFunctions **19-15**
InstantiateStages **19-16**
LoopOptimization **19-16**
LUTRegisterResetType **19-17**
MapPersistentVarsToRAM **19-17**
MapToRAM **19-19**
OutputPipeline **19-19**
PreserveUpstreamLogic **19-20**
RAMDirective **19-20**
ResetType **19-21**
SerialPartition **19-23**
SharingFactor **19-23**
SoftReset **19-23**
StreamingFactor **19-25**
UseRAM **19-25**
VariablesToPipeline **19-28**

HDL Block Properties: Native Floating Point **19-29**
Overview **19-29**
CheckResetToZero **19-30**
DivisionAlgorithm **19-30**
HandleDenormals **19-31**
InputRangeReduction **19-32**
LatencyStrategy **19-33**
CustomLatency **19-34**
NFPCustomLatency **19-35**
MantissaMultiplyStrategy **19-36**
MaxIterations **19-37**

HDL Filter Block Properties **19-39**
AdderTreePipeline **19-39**
AddPipelineRegisters **19-39**

ChannelSharing 19-40
CoeffMultipliers 19-40
DALUTPartition 19-40
DARadix 19-41
FoldingFactor 19-42
MultiplierInputPipeline 19-42
MultiplierOutputPipeline 19-42
NumMultipliers 19-43
ReuseAccum 19-43
SerialPartition 19-43

HDL Filter Architectures **19-45**
Fully Parallel Architecture 19-45
Serial Architectures 19-46
Frame-Based Architecture 19-47

Distributed Arithmetic for HDL Filters **19-50**
Requirements and Considerations for Generating Distributed Arithmetic Code 19-50
Further References 19-51

Set and View HDL Model and Block Parameters **19-52**
Set HDL Block Parameters 19-52
Set HDL Block Parameters for Multiple Blocks Programmatically 19-52
View All HDL Block Parameters 19-54
View Non-Default HDL Block Parameters 19-54
View HDL Model Parameters 19-54

Pass through and No HDL Implementations **19-56**
Pass-through and No HDL Implementations 19-56

Build a ROM Block with Simulink Blocks **19-57**

Getting Started with RAM and ROM in Simulink **19-58**

Wireless Communications Design for ASICs, FPGAs, and SoCs . . **19-61**
From Mathematical Algorithm to Hardware Implementation 19-61
HDL-Optimized Blocks 19-63
Reference Applications 19-63
Generate HDL Code and Prototype on FPGA 19-63

20 Generating HDL Code for Multirate Models

Code Generation from Multirate Models **20-2**
Clock Enable Generation for a Multirate DUT 20-2

Timing Controller for Multirate Models **20-4**
Timing Controller Naming 20-4

Generate Reset for Timing Controller **20-6**
Requirements for Timing Controller Reset Port Generation **20-6**
How To Generate Reset for Timing Controller **20-6**
Limitations for Timing Controller Reset Port Generation **20-6**

Multirate Model Requirements for HDL Code Generation **20-7**
Model Configuration Parameters **20-7**
Sample Rate **20-7**
Blocks To Use For Rate Transitions **20-7**

Generate a Global Oversampling Clock **20-9**
Specifying the Oversampling Value **20-9**
Requirements for the Oversampling Factor **20-10**
Resolving Oversampling Rate Conflicts **20-10**

Using Multiple Clocks in HDL Coder **20-14**

Using Triggered Subsystems for HDL Code Generation **20-19**
Best Practices **20-19**
Using the Signal Editor Block **20-19**
Using Trigger As Clock **20-19**
Specify Trigger As Clock **20-20**
Trigger As Clock Without Synchronous Registers **20-20**
Model Trigger Signal As Clock in Triggered Subsystem **20-20**
Use Triggered and Resettable Subsystem to Model Clock and Reset Signals **20-21**
Model Single Clock and Reset Signal Using Triggered and Resettable Subsystems **20-22**
Limitations **20-23**

Use Triggered Subsystem for Asynchronous Clock Domain **20-25**
Generate Multiple Clocks Using Trigger As Clock **20-25**
Generate Multiple Clocks and Resets Using Triggered and Resettable Subsystems **20-28**
Model Asynchronous and Synchronous Reset for a Simulink Model **20-30**

Meet Timing Requirements Using Enable-Based Multicycle Path Constraints **20-32**
How Enable-Based Multicycle Path Constraints Work **20-32**
Specify Enable-Based Constraints **20-33**
Benefits of Using Enable-Based Constraints **20-34**
Modeling Guidelines **20-34**
Multicycle Path Constraints for Various Synthesis Tools **20-35**
Preserve Enable Signals in Timing Control Logic **20-36**
Limitations **20-37**

Use Multicycle Path Constraints to Meet Timing for Slow Paths . **20-38**

21 Optimization

Speed and Area Optimizations in HDL Coder **21-3**
Optimizations in MATLAB HDL Code Generation **21-3**
Optimizations in Simulink HDL Code Generation **21-3**
General Optimizations **21-4**
Speed Optimizations **21-5**
Area Optimizations **21-5**

Automatic Iterative Optimization **21-7**
How Automatic Iterative Optimization Works **21-7**
Automatic Iterative Optimization Output **21-8**
Automatic Iterative Optimization Report **21-8**
Automatic Iterative Optimization Synthesis Tool and Hardware . . . **21-9**
Limitations of Automatic Iterative Optimization **21-9**

Generated Model and Validation Model **21-10**
Generated Model **21-10**
Validation Model **21-11**

Locate Numeric Differences After Speed Optimization **21-13**

Simplify Constant Operations and Reduce Design Complexity in HDL Coder **21-18**

Optimization with Constrained Overclocking **21-23**
Optimizations that Overclock Resources **21-23**
How to Use Constrained Overclocking **21-23**
Constrained Overclocking Limitations **21-24**

Resolve Numeric Mismatch with Delay Balancing **21-25**

Resolve Simulation Mismatch When Pipelining with a Feedback Loop Outside the DUT **21-31**

Streaming **21-42**
What Is Streaming? **21-42**
Specify Streaming **21-42**
How to Determine Streaming Factor and Sample Time **21-43**
Determine Blocks That Support Streaming **21-43**
Requirements for Streaming Subsystems **21-43**
Streaming Report **21-44**

Resource Sharing **21-45**
How Resource Sharing Works **21-45**
Benefits and Costs of Resource Sharing **21-46**
Shareable Resources in Different Blocks **21-46**
Specify Resource Sharing **21-46**
Block Requirements for Resource Sharing **21-47**
Resource Sharing Report **21-47**
Limitations for Resource Sharing **21-47**

Streaming: Area Optimization **21-49**

Resource Sharing for Area Optimization **21-54**

Single-Rate Resource Sharing Architecture **21-65**

Improve Resource Sharing with Design Modifications **21-69**

Improve Resource Sharing with Clone Detection and Replacement **21-75**

Delay Balancing **21-81**
Specify Delay Balancing **21-81**
Delay Balancing Considerations **21-83**
Delay Absorption During Delay Balancing **21-83**
Delay Balancing Report **21-84**
Delay Balancing Limitations **21-84**

Use Delay Absorption While Modeling with Latency **21-86**

Delay Balancing and Validation Model Workflow in HDL Coder . . **21-94**

Control the Scope of Delay Balancing **21-102**

Delay Balancing on Multirate Designs **21-107**

Find Feedback Loops **21-115**
Specify Highlighting of Feedback Loops **21-115**
Remove Highlighting **21-115**
Limitations **21-115**

Hierarchy Flattening **21-117**
What Is Hierarchy Flattening? **21-117**
When to Flatten Hierarchy **21-117**
Considerations **21-117**
How to Flatten Hierarchy **21-117**
Hierarchy Flattening Report **21-118**
Limitations for Hierarchy Flattening **21-118**

Apply RAM Mapping to Optimize Area **21-120**
RAM Mapping for a Simulink Model **21-120**
RAM Mapping for a MATLAB Design **21-121**
Use the RAM Mapping Threshold **21-121**
Exclude Inefficient RAM Mapping **21-122**

RAM Mapping with the MATLAB Function Block **21-125**

Distributed Pipelining **21-130**
What Is Distributed Pipelining? **21-130**
Benefits and Costs of Distributed Pipelining **21-131**
How Distributed Pipelining Works **21-131**
Requirements for Distributed Pipelining **21-132**
Specify Distributed Pipelining **21-132**
Distributed Pipelining Report **21-133**
Limitations of Distributed Pipelining **21-133**
Selected Bibliography **21-135**

Distributed Pipelining Using Synthesis Timing Estimates **21-136**
Synthesis Timing Estimates in Distributed Pipelining **21-136**
How Distributed Pipelining Works By Using Synthesis Timing Estimates **21-136**
Requirements for Synthesis Timing Estimates for Distributed Pipelining **21-137**
Specify Distributed Pipelining to Use Synthesis Timing Estimates **21-137**
Limitations **21-137**

Distributed Pipelining: Speed Optimization **21-139**

Constrained Output Pipelining **21-146**
What Is Constrained Output Pipelining? **21-146**
When to Use Constrained Output Pipelining **21-146**
Requirements for Constrained Output Pipelining **21-146**
Specify Constrained Output Pipelining **21-146**
Limitations of Constrained Output Pipelining **21-147**

Clock-Rate Pipelining **21-148**
Rationale for Clock-Rate Pipelining **21-148**
How Clock-Rate Pipelining Works in a Simulink Model **21-148**
How Clock-Rate Pipelining Works in a MATLAB Function **21-149**
Clock-Rate Pipelining and Hierarchy Flattening **21-149**
Clock-Rate Pipelining for DUT Output Ports **21-150**
Specify Clock-Rate Pipelining **21-150**
Clock-Rate Pipelining Report **21-151**
Limitations for Clock-Rate Pipelining **21-151**

Increase Clock Frequency Using Clock-Rate Pipelining **21-153**

Iteratively Maximize Clock Frequency by Using Speed Optimizations **21-167**

Adaptive Pipelining **21-181**
Requirements **21-181**
Specify Adaptive Pipelining **21-182**
Supported Blocks **21-182**
Pipeline Insertion for Rate Transition and Downsample Blocks . . **21-182**
Pipeline Insertion for Product and Gain Blocks **21-183**
Pipeline Insertion for Multiply-Add and Multiply-Accumulate Blocks **21-184**
Pipeline Insertion for MATLAB Function Blocks **21-186**
Adaptive Pipelining Report **21-186**

Design Patterns That Require Adaptive Pipelining **21-188**
Audio System That Uses Low Pass, Band Pass, and High Pass Filters **21-188**
Discrete FIR Filter That Uses Resource Sharing **21-188**

Critical Path Estimation Without Running Synthesis **21-192**
Critical Path Estimation Process **21-192**
Use Critical Path Estimation **21-194**
Characterized Blocks **21-195**
Considerations **21-201**

Generate Custom Timing Database for Custom Tools and Devices **21-202**

HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture 21-205

Subsystem Optimizations for Filters 21-214
Sharing **21-214**
Streaming **21-214**
Pipelining **21-214**
Area Reduction of Multichannel Filter Subsystem **21-215**
Area Reduction of Filter Subsystem **21-220**

Remove Redundant Logic and Unused Blocks in Generated HDL Code 21-224

Optimize Unconnected Ports in HDL Code for Simulink Models 21-246

22 I/O Optimizations

HDL Code Generation from Frame-Based Algorithms 22-2
Generating HDL Code from a Frame-based Algorithm **22-2**
Specify the Frame-to-Sample Conversion Optimization **22-3**
Generate HDL Code from a Frame-Based Model Example **22-4**
Hardware Considerations **22-7**
Supported Blocks and Operations **22-8**
Limitations **22-8**

Use Neighborhood, Reduction, and Iterator Patterns with a Frame-Based Model or Function for HDL Code Generation 22-10

Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods 22-17

Use Sample-Based Inputs and Frame-Based Inputs in an Algorithm 22-22

Synthesize Code for Frame-Based Model 22-26

Offload Large Delays from Frame-Based Models to External Memory 22-32

Optimize Area Usage for Frame-Based Algorithms with Tall Array Inputs 22-40

Compute Image Characteristics with a Frame-Based Model for HDL Code Generation 22-46

Generate HDL Code from Frame-Based Model by Using Neighborhood Processing with States 22-52

23 Code Generation Reports, HDL Compatibility Checker, Block Support Library, and Code Annotation

Create and Use Code Generation Reports **23-2**
Generate Reports **23-2**
Navigate the Code Generation Report **23-8**
Use Code Generation Reports to Evaluate Code Before Synthesis **23-10**

Navigate Between Simulink Model and HDL Code by Using Traceability **23-12**
How Traceability Works **23-12**
Generate Traceability Report **23-13**
Report Location **23-13**
View the Traceability Report **23-14**
Traceability by Using Code View **23-14**
Code-to-Model Navigation **23-15**
Model-to-Code Navigation **23-18**
Traceability Report Limitations **23-20**

Generate Web View of Model in Code Generation Report **23-21**
Generate HTML Code Generation Report with Model Web View . . **23-21**
Model Web View Limitations **23-22**

Generate HDL Code with Annotations or Comments **23-24**
Simulink Annotations **23-24**
Signal Descriptions **23-25**
Text Comments **23-25**
Requirement Comments and Hyperlinks **23-26**

Check Your Model for HDL Compatibility **23-29**

Display Blocks for HDL Code Generation in Library Browser **23-31**
Generate Table of Supported Blocks **23-32**

Trace Code Using the Mapping File **23-34**

Add or Remove the HDL Configuration Component **23-37**
HDL Configuration Component **23-37**
Add the HDL Configuration Component to a Model **23-37**
Remove the HDL Configuration Component from a Model **23-37**

24 HDL Coding Standards

HDL Coding Standard Report **24-2**
Rule Summary **24-2**
Rule Hierarchy **24-2**
Rule and Report Customization **24-3**
How to Fix Warnings and Errors **24-3**

HDL Coding Standards 24-4

Generate HDL Coding Standard Report from Simulink 24-5
Using the Configuration Parameters Dialog Box 24-5
Using the Command Line 24-5

Basic Coding Practices 24-7
1.A General Naming Conventions 24-8
1.B General Guidelines for Clocks and Resets 24-14
1.C Guidelines for Initial Reset 24-15
1.D Guidelines for Clocks 24-16
1.F Guidelines for Hierarchical Design 24-17

RTL Description Rules and Checks 24-18
2.A Guidelines for Combinational Logic 24-18
2.B Guidelines for "Always" Constructs of Combinational Logic . . . 24-23
2.C Guidelines for Flip-Flop Inference 24-25
2.D Guidelines for Latch Description 24-29
2.E Guidelines for Tristate Buffer 24-30
2.F Guidelines for Always/Process Construct with Circuit Structure into Account 24-32
2.G Guidelines for "IF" Statement Description 24-33
2.H Guidelines for "CASE" Statement Description 24-35
2.I Guidelines for "FOR" Statement Description 24-38
2.J Guidelines for Operator Description 24-39
2.K Guidelines for Finite State Machine Description 24-43

RTL Design Methodology Guidelines 24-44
3.A Guidelines for Creating Function Libraries 24-44
3.B Guidelines for Using Function Libraries 24-45
3.C Guidelines for Test Facilitation Design 24-47

Generate HDL Lint Tool Script 24-49
How to Generate an HDL Lint Tool Script 24-49

Cascaded Conditional Region Variable Assignments 24-51
Example Patterns that Fail the Check for Guideline 2.F.B.1.a 24-51
Example Patterns that Pass the Check for Guideline 2.F.B.1.a 24-52
Simulink Blocks and Modeling Patterns that Fail the Check for Guideline 2.F.B.1.a 24-53

25 Interfacing Subsystems and Models to HDL Code

Model Referencing for HDL Code Generation 25-2
How To Generate Code for a Referenced Model 25-2
Generate Code for Model Arguments 25-3
Generate Comments 25-3
Limitations 25-3

Generate Black Box Interface for Subsystem 25-4
What Is a Black Box Interface? 25-4

Requirements . . . 25-4
Generate a Black Box Interface for a Subsystem . . . 25-4
Generate Code for a Black Box Subsystem Implementation . . . 25-6

Generate Black Box Interface for Referenced Model . . . **25-8**
When to Generate a Black Box Interface . . . 25-8
How to Generate a Black Box Interface . . . 25-8
Caveats and Limitations . . . 25-8

Integrate Custom HDL Code by Using DocBlock . . . **25-10**
When to Use DocBlock for Integrating Custom Code . . . 25-10
Use DocBlock to Integrate Custom Code . . . 25-10
Restrictions . . . 25-10
Include Custom HDL Code Using Doc Block . . . 25-11

Customize Black Box or HDL Cosimulation Interface . . . **25-12**
Interface Parameters . . . 25-12

Specify Bidirectional Ports . . . **25-17**
Requirements . . . 25-17
How To Specify a Bidirectional Port . . . 25-17
Limitations . . . 25-17

Generate Reusable Code for Subsystems . . . **25-18**
Requirements for Generating Reusable Code for Atomic Subsystems . . . 25-18
Requirements for Generating Reusable Code for Virtual Subsystems . . . 25-18
Generate Reusable Code for Atomic Subsystems . . . 25-19
Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters . . . 25-21

Scalarization of Vector Ports in Generated VHDL Code . . . **25-25**

Create a Xilinx System Generator Subsystem . . . **25-29**
Why Use Xilinx System Generator Subsystems . . . 25-29
Requirements for Xilinx System Generator Subsystems . . . 25-29
Create a Xilinx System Generator Subsystem . . . 25-29
Limitations for Code Generation from Xilinx System Generator Subsystems . . . 25-30

Create an Altera DSP Builder Subsystem . . . **25-31**
Why Use Altera DSP Builder Subsystems? . . . 25-31
Requirements for Altera DSP Builder Subsystems . . . 25-31
How to Create an Altera DSP Builder Subsystem . . . 25-31
Determine Clocking Requirements for Altera DSP Builder Subsystems . . . 25-32
Limitations for Code Generation from Altera DSP Builder Subsystems . . . 25-32

Using Altera DSP Builder Advanced Blockset with HDL Coder . . . **25-33**

Using Xilinx System Generator for DSP with HDL Coder . . . **25-38**

Choose a Test Bench for Generated HDL Code . . . **25-41**

Generate a Cosimulation Model **25-43**
Requirements **25-43**
What Is A Cosimulation Model? **25-43**
Generating a Cosimulation Model using the Model Configuration Parameters **25-44**
Structure of the Generated Model **25-46**
Launching a Cosimulation **25-50**
The Cosimulation Script File **25-52**
Complex and Vector Signals in the Generated Cosimulation Model **25-54**
Generating a Cosimulation Model from the Command Line **25-55**
Naming Conventions for Generated Cosimulation Models and Scripts **25-55**
Limitations for Cosimulation Model Generation **25-56**

HDL Verifier Cosimulation Model Generation in HDL Coder **25-57**

Verify HDL Design Using SystemVerilog DPI Test Bench **25-68**

Pass-Through and No-Op Implementations **25-74**

Synchronous Subsystem Behavior with the State Control Block . **25-75**
What Is a State Control Block? **25-75**
State Control Block Modes **25-75**
Synchronous Badge for Subsystems by Using Synchronous Mode **25-76**
Generate HDL Code with the State Control Block **25-77**

Using the State Control Block to Generate More Efficient Code with HDL Coder **25-80**

Resettable Subsystem Support in HDL Coder **25-87**

26 Stateflow HDL Code Generation Support

Introduction to Stateflow HDL Code Generation **26-2**
Example **26-2**
Chart Initialization **26-2**
Tunable Parameters **26-3**
Comments in Stateflow Charts **26-3**
Restrictions **26-3**

Hardware Realization of Stateflow Semantics **26-6**

Generate HDL for Mealy and Moore Finite State Machines **26-7**
Generate HDL Code for Moore Finite State Machine **26-7**
Generate HDL for Mealy Finite State Machine **26-8**
Initialize Outputs Every Time Chart Wakes Up **26-10**

Design Patterns Using Advanced Chart Features **26-14**
Temporal Logic **26-14**

Graphical Function 26-15
Hierarchy and Parallelism 26-16
Stateless Charts 26-17
Truth Tables 26-18

Initialize Persistent Variables in MATLAB Functions **26-22**
MATLAB Function Block with No Direct Feedthrough 26-23
State Control Block in Synchronous Mode 26-24
Stateflow Chart Implementing Moore Semantics 26-26

27 Generating HDL Code with the MATLAB Function Block

HDL Applications for the MATLAB Function Block **27-2**
Structure of Generated HDL Code 27-2
HDL Applications 27-2

Generate HDL Code from a MATLAB Function Block **27-4**

Generate Instantiable Code for Functions **27-12**
How To Generate Instantiable Code for Functions 27-12
Generate Code Inline for Specific Functions 27-12
Limitations for Instantiable Code Generation for Functions 27-12

MATLAB Function Block Design Patterns for HDL **27-14**
HDL Design Pattern Blocks 27-16
Using Blocks in this Library for HDL Code Generation 27-17
Fixed-Point Algorithm Support 27-17

Design Guidelines for the MATLAB Function Block **27-19**
Use Compiled External Functions With MATLAB Function Blocks 27-19
Build the MATLAB Function Block Code First 27-19
Use the hdlfimath Utility for Optimized FIMATH Settings 27-19
Use Optimal Fixed-Point Option Settings 27-20
Set the Output Data Type of MATLAB Function Blocks Explicitly . 27-20
Using Tunable Parameters 27-20
Run HDL Model Check for MATLAB Function Blocks 27-20
Use MATLAB Datapath Architecture for Enhanced HDL Optimizations 27-21

CORDIC Algorithm Using the MATLAB Function Block **27-22**

Create Hardware Design Patterns Using the MATLAB Function Block For HDL Code Generation **27-23**

Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks **27-28**

28 Generating Scripts for HDL Simulators and Synthesis Tools

Generate Scripts for Compilation, Simulation, and Synthesis 28-2

Structure of Generated Script Files 28-3

Properties for Controlling Script Generation 28-4
Enabling and Disabling Script Generation 28-4
Customizing Script Names 28-4
Customizing Script Code 28-4
Examples 28-6

Configure Compilation, Simulation, Synthesis, and Lint Scripts . . 28-7
Compilation Script Options 28-8
Simulation Script Options 28-9
Synthesis Script Options 28-11
Lint Script Options 28-14

Add Synthesis Attributes 28-17

Configure Synthesis Project Using Tcl Script 28-18

29 Using the HDL Workflow Advisor

Workflows in HDL Workflow Advisor 29-2
Set Up HDL Workflow Advisor in MATLAB 29-2
Set Up HDL Workflow Advisor in Simulink 29-2
Generic ASIC/FPGA 29-2
IP Core Generation 29-3
Simulink Real-Time FPGA I/O 29-4
FPGA-in-the-Loop 29-4

Getting Started with the HDL Workflow Advisor 29-5
Open the HDL Workflow Advisor 29-5
Run Tasks in the HDL Workflow Advisor 29-6
Fix HDL Workflow Advisor Warnings or Failures 29-7
Save and Restore the HDL Workflow Advisor State 29-7
View and Save HDL Workflow Advisor Reports 29-8

Generate Code and Synthesize on FPGA Using HDL Workflow Advisor 29-11
FIR Filter Model 29-11
Create a Folder and Copy Relevant Files 29-12
Set Up Tool Path 29-13
Open the HDL Workflow Advisor 29-13
Generate HDL Code and Synthesize on FPGA 29-14
Run Workflow at Command Line with a Script 29-14

Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor 29-16

Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries 29-19
Set Up Design for Mixed-Mode Mapping 29-19
Map to Native Floating-Point and FPGA Floating-Point Libraries . . 29-20
View Code Generation Reports of Floating-Point Library Mapping 29-21
Analyze Results of Floating-Point Library Mapping 29-22

FPGA Floating-Point Library IP Mapping 29-25

Customize Floating-Point IP Configuration 29-36
Customize the IP Latency with Target Frequency 29-36
Customize the IP Latency with Latency Strategy 29-39

HDL Coder Support for FPGA Floating-Point Library Mapping . . . 29-41
Supported Blocks That Map to FPGA Floating-Point Target IP 29-41
Supported Blocks That Do Not Need to Map to FPGA Floating-Point Target IP 29-43
Limitations for FPGA Floating-Point Library Mapping 29-44

Synthesis Objective to Tcl Command Mapping 29-45
Altera Quartus II 29-45
Xilinx Vivado 2014.4 29-45
Xilinx ISE 14.7 with PlanAhead 29-46

Run HDL Workflow with a Script 29-47
Export an HDL Workflow Script 29-48
Specify Verbosity of Workflow Script 29-48
Enable or Disable Tasks in HDL Workflow Script 29-48
Run a Single Workflow Task 29-48
Import an HDL Workflow Script 29-49
Generic ASIC/FPGA Workflow Script Example 29-49
FPGA-in-the-Loop Script Example 29-50
IP Core Generation Workflow Script Example 29-51
Simulink Real-Time FPGA I/O Workflow Example 29-53

Get Started with HDL Workflow Command-Line Interface 29-56

30 Simscape to HDL Workflow

Get Started with Simscape Hardware-in-the-Loop Workflow 30-2
Modeling Physical Systems in Simscape for HDL Compatibility 30-2
Simscape Example Models for HDL Code Generation 30-3
Generate HDL Implementation Model by Using the Simscape HDL Workflow Advisor 30-4
HDL Code Generation and Deployment 30-7
Restrictions for HDL Code Generation from Simscape Models 30-8

Modeling Guidelines for Simscape Subsystem Replacement **30-9**
Enclose Simscape Blocks Inside a Subsystem **30-9**
Multiple Simscape Network Considerations **30-10**
Avoid Using Certain Blocks in Simscape Utilities Library **30-11**

Generate HDL Code for Simscape Models **30-13**

Generate Optimized HDL Implementation Model from Simscape **30-20**

Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model **30-28**

Deploy Simscape Buck Converter Model to Speedgoat IO Module Using HDL Workflow Script **30-36**

Deploy Simscape Grid Tied Converter Model to Speedgoat IO Module Using HDL Workflow Script **30-47**

Partition Simscape Models Containing a Large Network into Multiple Smaller Networks **30-61**

Generate HDL Code for Simscape Models with Multiple Networks **30-67**

Replace Piecewise-Constant Resistor with Switched Linear Components **30-74**

Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules **30-82**

Validate HDL Implementation Model to Simscape Algorithm **30-89**
Bridge Rectifier Model **30-89**
Increase Validation Logic Tolerance **30-91**
Increase Number of Solver Iterations **30-92**
Use Larger Floating-Point Precision **30-93**

Improve FPGA Sampling Frequency of HDL Implementation Model Generated from Simscape Algorithm **30-96**
FPGA Sampling Frequency **30-96**
Boost Converter Model **30-97**
Reducing Number of Solver Iterations **30-98**
Using Oversampling Factor and Latency Strategy **30-99**

Generate HDL Code for Nonlinear Simscape Models by Using Partitioning Solver **30-100**

Deploy Simscape DC Motor Model to Speedgoat FPGA IO Module **30-106**

Generate HDL Code for Simscape Three-Phase PMSM Drive Containing Averaged Switch **30-118**

Generate HDL Code for Two-Speed Transmission Model Containing Mode Charts **30-125**

Simscape Language Support **30-131**
Domain and Component Declarations **30-131**
Equations **30-132**
Discrete Events and Mode Charts **30-133**
Composite Components **30-134**

Generate HDL Code for Simscape Models by Using Trapezoidal Rule Solver **30-135**

Generate HDL Code for Simscape Models by Using Linearized Switch Approximation **30-146**

Estimate Achievable Target Frequency Without Running Synthesis **30-155**
Example Models with Estimated Sample Time **30-155**

Generate FPGA Bitstream for Two-Phase DC-DC Converter with Tunable Run-Time Parameters **30-156**

31 Simscape HDL Workflow Advisor Tasks

Simscape HDL Workflow Advisor Tasks **31-2**
Simscape HDL Workflow Advisor **31-2**
Code Generation Compatibility **31-3**
Check Solver Configuration **31-3**
Check Model Compatibility **31-3**
State-Space Conversion **31-4**
Extract Discrete Equations **31-4**
Implementation Model Generation **31-4**
Set Target **31-4**
Generate Implementation Model **31-5**

Simscape HDL Workflow Advisor Tips and Guidelines **31-7**
Estimating Resource Consumption Using Algebraic and Differential Variables **31-7**
Setting Simulation Stop Time for Extract Discrete Equations **31-8**
Changing Sample Time for Extract Discrete Equations **31-9**
Using Number of Solver Iterations **31-9**
Data Type Precision and Numerical Accuracy **31-10**
Map State Space Parameters to RAMs **31-11**
Optimal Sharing Factor Supported FPGA Device Families **31-11**

32 Troubleshooting

Troubleshooting Real-Time Hardware Deployment Issues in Simscape Hardware-in-the-Loop Workflow **32-2**
Parameter Settings **32-2**

Critical Path Estimation . . . 32-3
How Sample Rate Affects the Timing on the Hardware . . . 32-4
How to Reduce the Sample Rate Variation . . . 32-6
Limitations . . . 32-8

Troubleshoot Validation Errors in Simscape Hardware-in-the-Loop Workflow . . . **32-9**
Causes of Validation Errors . . . 32-10

33 Model Protection in HDL Coder

Create Protected Models to Conceal Contents and Generate HDL Code . . . **33-2**
How Model Protection Works . . . 33-2
How to Create a Protected Model . . . 33-2
General Protected Model Requirements and Limitations . . . 33-3
Protected Model Restrictions for HDL Code Generation . . . 33-3
Prepare the Parent Model . . . 33-4
Protect the Referenced Model . . . 33-5
Protected Model Report . . . 33-7
Generate HDL Code for Models Referencing Protected Model . . . 33-9

Test Protected Models . . . **33-10**

Package and Share Protected Models . . . **33-12**
Harness Model . . . 33-12
MAT-File with Base Workspace Definitions . . . 33-12
Simulink Data Dictionary . . . 33-13
Protected Model File Contents . . . 33-13

Obfuscate Generated HDL Code from Simulink Models . . . **33-15**
How to Generate Obfuscated HDL Code . . . 33-15
Generated HDL Code with Obfuscation . . . 33-15
Code Obfuscation Report . . . 33-16
HDL Model Parameters Incompatible with Code Obfuscation . . . 33-16
Code Obfuscation Considerations and Restrictions . . . 33-17

34 HDL Test Bench

Verify Generated Code Using HDL Test Bench from Configuration Parameters . . . **34-2**
FIR Filter Model . . . 34-2
Create a Folder and Copy Relevant Files . . . 34-4
What is a HDL Test Bench? . . . 34-5
How to Verify the Generated Code . . . 34-5
Generate HDL Test Bench . . . 34-5
View HDL Test Bench Files . . . 34-6

Run Simulation and Verify Generated HDL Code **34-7**

Verify Generated Code Using HDL Test Bench at Command Line 34-9
FIR Filter Model **34-9**
Create a Folder and Copy Relevant Files **34-11**
What is a HDL Test Bench? **34-12**
How to Verify the Generated Code **34-12**
Generate HDL Test Bench **34-12**
View HDL Test Bench Files **34-13**
Run Simulation and Verify Generated HDL Code **34-13**

Test Bench Generation 34-15
How Test Bench Generation Works **34-15**
Test Bench Data Files **34-15**
Test Bench Data Type Limitations **34-15**
Use Constants Instead of File I/O **34-15**

Test Bench Block Restrictions 34-17

35 FPGA Board Customization

FPGA Board Customization 35-2
Feature Description **35-2**
Custom Board Management **35-2**
FPGA Board Requirements **35-2**

Create Custom FPGA Board Definition 35-6

Create Xilinx KC705 Evaluation Board Definition File 35-7
Overview **35-7**
What You Need to Know Before Starting **35-7**
Start New FPGA Board Wizard **35-7**
Provide Basic Board Information **35-8**
Specify FPGA Interface Information **35-9**
Enter FPGA Pin Numbers **35-10**
Run Optional Validation Tests **35-12**
Save Board Definition File **35-13**
Use New FPGA Board **35-14**

FPGA Board Manager 35-18
Introduction **35-18**
Filter **35-19**
Search **35-19**
FIL Enabled/Turnkey Enabled **35-20**
Create Custom Board **35-20**
Add Board from File **35-20**
Get More Boards **35-20**
View/Edit **35-20**
Remove **35-20**
Clone **35-20**

Validate 35-20

New FPGA Board Wizard **35-22**
Basic Information 35-23
Interfaces 35-23
FIL I/O 35-26
Turnkey I/O 35-28
Validation 35-31
Finish 35-32

FPGA Board Editor **35-33**
General Tab 35-33
Interface Tab 35-35

36 HDL Workflow Advisor Tasks

HDL Workflow Advisor Tasks **36-2**
HDL Workflow Advisor Tasks Overview 36-3
Set Target Overview 36-3
Set Target Device and Synthesis Tool 36-4
Set Target Reference Design 36-5
Set Target Interface 36-6
Set Target Interface 36-7
Set Target Frequency 36-7
Prepare Model for HDL Code Generation Overview 36-8
Check Model Settings 36-9
Check FPGA-in-the-Loop Compatibility 36-9
HDL Code Generation Overview 36-9
Set HDL Options 36-10
Generate RTL Code and Testbench 36-10
Verify with HDL Cosimulation 36-11
Generate RTL Code and IP Core 36-11
FPGA Synthesis and Analysis Overview 36-13
Create Project 36-14
Perform Synthesis and P/R Overview 36-15
Perform Logic Synthesis 36-15
Perform Mapping 36-15
Perform Place and Route 36-16
Run Synthesis 36-16
Run Implementation 36-17
Annotate Model with Synthesis Result 36-17
Download to Target Overview 36-18
Generate Programming File 36-18
Program Target Device 36-18
Generate Simulink Real-Time Interface 36-19
Save and Restore HDL Workflow Advisor State 36-19
FPGA-in-the-Loop (FIL) Implementation 36-19
Set FPGA-in-the-Loop Options 36-19
Build FPGA-in-the-Loop 36-20
Embedded System Integration 36-20
Create Project 36-20

Generate Software Interface **36-21**
Build FPGA Bitstream **36-22**
Program Target Device **36-22**

37 HDL Code Advisor

HDL Coder Checks in Model Advisor / HDL Code Advisor Overview **37-3**

Model configuration checks overview **37-4**

Check for model parameters suited for HDL code generation **37-5**
Description **37-5**
Results and Recommended Actions **37-6**
See Also **37-6**

Check for global reset setting for Xilinx and Altera devices **37-7**
Description **37-7**
Results and Recommended Actions **37-7**
See Also **37-7**

Check inline configurations setting **37-8**
Description **37-8**
Results and Recommended Actions **37-8**

Check algebraic loops **37-9**
Description **37-9**
Results and Recommended Actions **37-9**
See Also **37-9**

Check for visualization settings **37-10**
Description **37-10**
Results and Recommended Actions **37-10**
See Also **37-10**

Check delay balancing setting **37-11**
Description **37-11**
Results and Recommended Actions **37-11**
See Also **37-11**

Check for ports and subsystems overview **37-12**

Check for invalid top level subsystem **37-13**
Description **37-13**
Results and Recommended Actions **37-13**

Check for blocks and block settings overview **37-14**

Check for infinite and continuous sample time sources **37-15**
Description **37-15**
Results and Recommended Actions **37-15**

See Also 37-15

Check for unsupported blocks 37-16
Description 37-16
Results and Recommended Actions 37-16

Check for large matrix operations 37-17
Description 37-17
Results and Recommended Actions 37-17
See Also 37-17

Check for MATLAB Function block settings 37-18
Description 37-18
Results and Recommended Actions 37-18
See Also 37-18

Check for Stateflow chart settings 37-19
Description 37-19
Results and Recommended Actions 37-19
See Also 37-19

Check for obsolete Unit Delay Enabled/Resettable Blocks 37-20
Description 37-20
Results and Recommended Actions 37-20

Check for blocks that have nonzero output latency 37-21
Description 37-21
Results and Recommended Actions 37-21
See Also 37-21

Check for unsupported storage class for signal objects 37-22
Description 37-22
Results and Recommended Actions 37-22

Check for HDL Reciprocal block usage 37-23
Description 37-23
Results and Recommended Actions 37-23
See Also 37-23

Check for Trigonometric Function block for LUT-based approximation method 37-24
Description 37-24
Results and Recommended Actions 37-24
Capabilities and Limitations 37-24
See Also 37-24

Native Floating Point Checks Overview 37-25

Check for single datatypes in the model 37-26
Description 37-26
Results and Recommended Actions 37-26
See Also 37-26

Check for double data types in the model 37-27
Description 37-27

Results and Recommended Actions 37-27
See Also 37-27

Check for Data Type Conversion blocks with incompatible settings 37-28
Description 37-28
Results and Recommended Actions 37-28
See Also 37-28

Check for HDL Reciprocal block usage 37-29
Description 37-29
Results and Recommended Actions 37-29
See Also 37-29

Check for Relational Operator block usage 37-30
Description 37-30
Results and Recommended Actions 37-30
See Also 37-30

Check for unsupported blocks with Native Floating Point 37-31
Description 37-31
Results and Recommended Actions 37-31
See Also 37-31

Check blocks with nonzero ULP error 37-32
Description 37-32
Results and Recommended Actions 37-32
See Also 37-32

Industry standard checks overview 37-33

Check file extension 37-34
Description 37-34
Results and Recommended Actions 37-34
See Also 37-34

Check naming conventions 37-35
Description 37-35
Results and Recommended Actions 37-35
See Also 37-35

Check top-level subsystem/port names 37-36
Description 37-36
Results and Recommended Actions 37-36
See Also 37-36

Check module/entity names 37-37
Description 37-37
Results and Recommended Actions 37-37
See Also 37-37

Check signal and port names 37-38
Description 37-38
Results and Recommended Actions 37-38
See Also 37-38

Check package file names **37-39**
Description **37-39**
Results and Recommended Actions **37-39**
See Also **37-39**

Check generics **37-40**
Description **37-40**
Results and Recommended Actions **37-40**
See Also **37-40**

Check clock, reset, and enable signals **37-41**
Description **37-41**
Results and Recommended Actions **37-41**
See Also **37-41**

Check architecture name **37-42**
Description **37-42**
Results and Recommended Actions **37-42**
See Also **37-42**

Check entity and architecture **37-43**
Description **37-43**
Results and Recommended Actions **37-43**
See Also **37-43**

Check clock settings **37-44**
Description **37-44**
Results and Recommended Actions **37-44**
See Also **37-44**

38 Using the HDL Code Advisor

Check HDL Compatibility of Simulink Model Using HDL Code Advisor **38-2**
Open the HDL Code Advisor **38-2**
Run Checks In the HDL Code Advisor **38-3**
Fix HDL Code Advisor Warnings or Failures **38-3**
View and Save HDL Code Advisor Reports **38-4**

Run Model Advisor Checks for HDL Coder **38-6**
Open the Model Advisor Checks **38-6**
Run Checks in the Model Advisor **38-6**
Display Check Results in the Model Advisor Report **38-7**
Fix Warnings or Failures **38-8**

HDL Code Advisor Checks **38-9**
Model configuration checks **38-10**
Checks for ports and subsystems **38-10**
Checks for blocks and block settings **38-10**
Native Floating Point checks **38-11**
Industry standard checks **38-12**

Hardware-Software Codesign

39 Hardware-Software Co-Design Basics

Targeting FPGA & SoC Hardware Overview **39-3**
Hardware-Software Co-Design **39-4**
Prototype and Deploy HDL Algorithm for an FPGA **39-4**
Deploy C Algorithm for Processor **39-6**
Create Custom Hardware Platform **39-7**

Hardware-Software Co-Design Workflow for SoC Platforms **39-9**

Speedgoat FPGA Support with HDL Workflow Advisor **39-15**
Speedgoat Simulink-Programmable I/O Module Support **39-15**
Prepare for FPGA Workflow **39-15**

Custom IP Core Generation **39-17**
Custom IP Core Architectures **39-17**
Target Platform Interfaces **39-17**
Processor and FPGA Synchronization **39-18**
Custom IP Core Generated Files **39-18**
Restrictions **39-19**

Custom IP Core Report **39-20**
Summary **39-20**
Target Interface Configuration **39-20**
Register Address Mapping **39-21**
Bit Packing Order **39-22**
IP Core User Guide **39-23**
IP Core File List **39-26**

Comparison of IP Core Generation Techniques **39-27**
IP Core Process Comparison **39-27**
Limitations **39-29**

Comparison of IP Core Deployment and Verification Techniques
. . . . **39-30**
Embedded System Integration Process Comparison **39-30**
Limitations **39-31**

Generate Board-Independent HDL IP Core from Simulink Model
. . . . **39-33**
Generate Board-Independent IP Core **39-33**
Generate Tool-Independent IP Core **39-35**
IP Core Without AXI4 Slave Interfaces **39-36**
Requirements and Limitations for IP Core Generation **39-36**

Processor and FPGA Synchronization **39-38**
Free Running Mode **39-38**
Coprocessing – Blocking Mode **39-38**

Synchronization of Global Reset Signal to IP Core Clock Domain **39-40**

IP Caching for Faster Reference Design Synthesis **39-44**
Requirements for Using IP Caching **39-44**
What Is an IP Cache? **39-44**
How IP Caching Works **39-45**
Enable IP Caching **39-45**
IP Caching in HDL Coder Reference Designs **39-46**
IP Caching in Custom Reference Designs **39-47**

Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows **39-49**
Step 1: Identify the Timing Failure **39-49**
Step 2: Find the Critical Path **39-52**
Step 3: Resolve Timing Failures **39-56**

Define Multiple AXI Master Interfaces in Reference Designs to Access DUT AXI4 Slave Interface **39-60**
Vivado-Based Reference Designs **39-61**
Qsys-Based Reference Designs **39-62**

Program Target FPGA Boards or SoC Devices **39-64**
How to Program Target Device **39-64**
Programming Methods **39-65**

Generate and Manage FPGA I/O Host Interface Scripts **39-68**
Prerequisites **39-68**
Generate Host Interface Scripts **39-68**
Host Interface Script Files **39-69**
Use the Host Interface Script with Hardware **39-71**
Manage Host Interface Scripts **39-73**

Choose a Method to Interact with IP Cores on Target Hardware . **39-74**
Overview of Interface Methods **39-74**
Prerequisites **39-75**
FPGA I/O **39-75**
Simulink Host Interface Model **39-77**
FPGA Data Capture **39-78**
Simulink Software Interface Model in External Mode **39-79**
Simulink Software Interface Model for Standalone Applications . . **39-80**
Generic Software Interface **39-81**

Generate Software Interface Model to Probe and Rapidly Prototype HDL IP Core **39-84**
Prerequisites **39-84**
Generate Software Interface **39-84**
Software Interface Model **39-86**

Use FPGA I/O to Rapidly Prototype HDL IP Core **39-90**
Prerequisites **39-90**
Ethernet-Based Interface **39-90**
JTAG-Based Interface **39-95**

Getting Started with Targeting Xilinx Zynq Platform **39-98**

Getting Started with Targeting Zynq UltraScale+ MPSoC Platform **39-112**

Getting Started with Targeting Intel SoC Devices **39-132**

Getting Started with Targeting Intel Quartus Pro Based Devices **39-147**

Integrate HDL IP Core with Microchip PolarFire SoC Icicle Kit Reference Design **39-161**

Save Target Hardware Settings in Model **39-177**

Generate IP Core from MATLAB for Blinking LEDs on FPGA Board **39-180**

Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow **39-190**

Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow **39-201**

IP Core Generation Workflow with a MicroBlaze processor: Xilinx Kintex-7 KC705 . . . . **39-211**

Map Bus Data Types to AXI4 Slave Interfaces **39-234**

Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands **39-241**

Author a Xilinx Zynq Linux Image for a Custom Zynq Board by Using MathWorks Buildroot **39-253**

Generate Board-Independent HDL IP Core for Microchip Platforms **39-260**

Getting Started with Targeting Xilinx Versal Adaptive SoC Platform **39-265**

40 Target SoC Platforms and Speedgoat Boards

Model Design for AXI4 Slave Interface Generation **40-3**
- Considerations **40-3**
- Map Scalar Ports to AXI4 Slave Interface **40-3**
- Map Vector Ports to AXI4 Slave Interface **40-4**
- Map Double Data Types and Data Larger than 32 bits to AXI4-Slave Interfaces **40-5**
- Map Bus Data Types to AXI4 Slave Interface **40-8**
- Specify Initial Value of AXI4 Slave Registers **40-9**
- Read Back Value of AXI4 Slave Interfaces **40-10**
- Optimize AXI4 Slave Read Logic **40-13**

Model Design for AXI4-Stream Interface Generation **40-14**
Sample-Based Modeling **40-14**
Frame-Based Modeling **40-21**
Legacy Frame-Based Modeling **40-21**
Map Vector Ports to AXI4-Stream Interfaces **40-21**
Model Designs with Multiple Streaming Channels **40-25**
Model Designs That Have Multiple Sample Rates **40-25**
Interface Options for AXI4-Stream Data **40-26**
Restrictions **40-28**

Model Design for Frame-Based IP Core Generation **40-29**
Frame-Based Modeling for AXI4-Stream Interfaces **40-29**
Frame-Based Modeling for AXI4-Stream Video Interfaces **40-30**
Enable the Optimization **40-31**
Modeling Requirements **40-32**

Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces **40-33**
Why Use Multiple AXI4 Interfaces **40-33**
Specify Multiple AXI4 Interfaces in Generic IP Core Generation Workflow **40-33**
Specify Multiple AXI4 Interfaces in Custom Reference Designs . . . **40-34**
Ready Signal Mapping for Multiple Streaming Interfaces **40-36**
Restrictions **40-36**

Running Audio Filter with Multiple AXI4-Stream Channels on ZedBoard **40-38**

Inspect the Written Values of AXI4 Slave Registers by Using the Readback Methods **40-51**

Use MATLAB FPGA I/O Host Interface to Communicate with FPGA on Zynq-Based Radio **40-67**

Multirate IP Core Generation **40-85**

Board and Reference Design Registration System **40-89**
Board, IP Core, and Reference Design Definitions **40-89**
Board Registration Files **40-89**
Reference Design Registration Files **40-90**
Predefined Board and Reference Design Examples **40-91**

Register a Custom Board **40-92**
Define a Board **40-92**
Create a Board Plugin **40-93**
Define a Board Registration Function **40-93**

Register a Custom Reference Design **40-95**
Define a Reference Design **40-95**
Create a Reference Design Plugin **40-96**
Define a Reference Design Registration Function **40-96**

Define Custom Parameters and Callback Functions for Custom Reference Design **40-98**
Define Custom Parameters and Register Callback Function Handle **40-98**
Define Custom Callback Functions **40-102**

Customize Reference Design Dynamically Based on Reference Design Parameters **40-104**
Why Customize the Reference Design **40-104**
How Reference Design Customization Works **40-104**
Customizable Reference Design Parameters **40-105**
Example: Create Master Only or Slave Only or Both Slave and Master Reference Designs **40-106**

Define and Add IP Repository to Custom Reference Design **40-109**
Create an IP Repository Folder Structure **40-109**
Define IP List Function **40-110**
Add IP List Function to Reference Design Project **40-111**

FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules **40-113**

Model Design for AXI4-Stream Video Interface Generation **40-118**
Sample Based Modeling **40-118**
Protocol Signals and Timing Diagrams **40-119**
Model Data and Control Bus Signals **40-120**
Map DUT Ports to Multiple Channels **40-124**
Model Designs with Multiple Sample Rates **40-124**
Video Porch Insertion Logic **40-124**
Default Video System Reference Design **40-125**
Restrictions **40-126**
Frame-Based Modeling **40-126**

Model Design for AXI4 Master Interface Generation **40-128**
Simplified AXI4 Master Protocol - Write Channel **40-128**
Simplified AXI4 Master Protocol - Read Channel **40-130**
Base Address Register Calculation **40-132**
Specify Initial Value of AXI4 Master Read and Write Base Address **40-132**
Modeling for AXI4 Master Interfaces **40-132**
Map Vector Ports to AXI4 Master Interfaces **40-134**
Model ID Signals to Reduce the Number of AXI-4 Master Interfaces **40-136**
Model Designs with Multiple Sample Rates **40-138**
Reference Designs for IP Core Integration **40-138**
Restrictions **40-139**

IP Core Generation Workflow for Standalone FPGA Devices **40-141**
Targeting FPGA Reference Designs with AXI4 Interface **40-142**
Targeting FPGA Reference Designs Without AXI4 Interface **40-144**
Board Support **40-144**
Restrictions **40-144**

IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules 40-145
Supported I/O Modules 40-145
IP Core Generation Workflow 40-145
Restrictions 40-147

Map Bus Data Types to PCIe Interface 40-148
Model Bus Element 40-148

IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip 40-150

Running an Audio Filter on Live Audio Input Using Intel Board 40-170

Running an Audio Filter on Live Audio Input Using a Zynq Board 40-181

Deploy Model with AXI-Stream Interface in Zynq Workflow 40-192

Deploy Model with AXI4-Stream Video Interface on Zynq Hardware 40-207

Perform Matrix Operation Using External Memory 40-218

Authoring a Reference Design for Audio System on a Zynq Board 40-226

Authoring a Reference Design for Audio System on a ZYBO Board 40-236

Authoring a Reference Design for Audio System on Intel Board 40-242

Define Custom Board and Reference Design for Zynq Workflow 40-252

Define Custom Board and Reference Design for Intel SoC Workflow 40-270

Define Custom Board and Reference Design for Microchip Workflow 40-285

Define Custom Board and Reference Design for Microchip Pure FPGA Platforms 40-305

Dynamically Create Reference Design with Master Only or Slave Only AXI4-Stream Interface 40-320

Use JTAG AXI Manager to Control HDL Coder Generated IP Core 40-333

Debug a Zynq Design Using HDL Coder and Embedded Coder . . 40-346

Debug IP Core Using FPGA Data Capture 40-351

Field-Oriented Control of a Permanent Magnet Synchronous Machine on a Xilinx Zynq Platform **40-361**

Deploy a Frame-Based Model with AXI4-Stream Interfaces **40-378**

Deploy Frame-Based Models with AXI4-Stream Video Interfaces in Zynq-Based Hardware **40-386**

DAC and ADC Loopback Data Capture **40-392**

IQ Mixer Mode Capture **40-404**

PL-DDR4 ADC Data Capture **40-410**

DAC PL-DDR4 Transmit **40-419**

Polyphase Channelizer **40-426**

Multi-Tile Synchronization **40-430**

Enable Clock Domain Crossing on AXI4-Lite Interfaces **40-439**

Use Clock Domain Crossing to Run DUT Algorithm and AXI4-Lite Interface at Different Frequencies **40-442**

41 Device Tree Generation

Generate Device Tree for IP Core **41-2**
Get Started with Device Trees **41-2**
Use Device Trees with IP Core Generation Workflow **41-3**
Deploy Device Tree and Bitstream **41-8**
Sample Device Tree **41-9**

HDL Code Generation from MATLAB

MATLAB Algorithm Design

- “Functions Supported for HDL and HLS Code Generation” on page 1-2
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “Persistent Variables and Persistent Array Variables” on page 1-9
- “Complex Data Type Support” on page 1-11
- “Complex Data Type Support for High-Level Synthesis Code Generation” on page 1-14
- “HDL Code Generation for System Objects” on page 1-16
- “HDL Code Generation from System Objects” on page 1-18
- “HDL Code Generation for Streaming Matrix Inverse System Object” on page 1-22
- “HDL Code Generation for Streaming Matrix Multiply System Object” on page 1-31
- “HDL Code Generation from hdl.RAM System Object” on page 1-39
- “HDL Code Generation from a Non-Restoring Square Root System Object” on page 1-42
- “HDL Code Generation from Viterbi Decoder System Object” on page 1-46
- “Predefined System Objects Supported for HDL Code Generation” on page 1-50
- “Load constants from a MAT-File” on page 1-52
- “Generate Code for User-Defined System Objects” on page 1-53
- “Map Matrices to ROM” on page 1-55
- “Model State with Persistent Variables and System Objects” on page 1-56
- “Bitwise Operations in MATLAB for HDL and HLS Code Generation” on page 1-58
- “Mapping of Different Rounding and Overflow Methods from MATLAB to HLS” on page 1-61
- “Handling Constants in HDL and HLS Code Generation” on page 1-63
- “Structure Definition for HLS Code Generation” on page 1-65
- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Indexing Best Practices for HDL Code Generation” on page 1-68
- “For-Loop Best Practices for HDL Code Generation” on page 1-73
- “MATLAB Test Bench Requirements and Best Practices for Code Generation” on page 1-76

Functions Supported for HDL and HLS Code Generation

In this section...
"Supported MATLAB and Fixed Point Runtime Library Functions" on page 1-2
"Fixed-Point Function Limitations" on page 1-2

You can generate efficient code for a subset of MATLAB built-in functions and toolbox functions that you call from MATLAB code.

Supported MATLAB and Fixed Point Runtime Library Functions

The supported functions for HDL and HLS code generation are listed in the following tables. In these tables, a ⚠ icon before the name of a function indicates that there are specific usage notes and limitations related to HDL and HLS code generation for that function. To view these usage notes and limitations, in the corresponding reference page, scroll down to the **Extended Capabilities** section at the bottom and expand the **HDL Code Generation** section.

The table shows HDL and HLS code generation support for both MATLAB and fixed-point run-time library functions from the Fixed-Point Designer™ functions.

HDL and HLS code generation support for the functions is summarized in the following tables.

- Functions Supported for HDL and HLS Code Generation (Category List)
- Functions Supported for HDL and HLS Code Generation (Alphabetical List)

Fixed-Point Function Limitations

In addition to function-specific limitations listed in the table, the following general limitations apply to the use of Fixed-Point Designer functions in generated HDL and HLS code:

- `fipref` and `quantizer` objects are not supported.
- Slope and bias scaling are not supported.
- Dot notation is only supported for getting the values of `fimath` and `numerictype` properties. Dot notation is not supported for `fi` objects, and it is not supported for setting properties.
- Word lengths greater than `128` bits are not supported.
- You cannot change the `fimath` or `numerictype` of a given variable after that variable has been created.
- The `boolean` and `ScaledDouble` values of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.
- The `numel` function returns the number of elements of `fi` objects in the generated code.
- General limitations of C/C++ code generated from MATLAB apply. See "MATLAB Language Features That Code Generation Does Not Support".

See Also

`codegen` | `coder.HdlConfig`

More About

- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “Bitwise Operations in MATLAB for HDL and HLS Code Generation” on page 1-58
- “Functions for Programming and Data Types”

Supported MATLAB Data Types, Operators, and Control Flow Statements

In this section...

"Supported Data Types" on page 1-4
"Supported Operators" on page 1-5
"Control Flow Statements" on page 1-7

When you generate HDL and HLS code from your MATLAB algorithm, use the data types, operators, and control flow statements that HDL Coder supports.

Supported Data Types

HDL Coder does not support cell arrays and `Inf` data types. This table shows the supported subset of MATLAB data types.

Types	Supported Data Types	Restrictions
Integer	• `uint8`, `uint16`, `uint32`, `uint64` • `int8`, `int16`, `int32`, `int64`	In Simulink®, MATLAB Function block ports must use numeric types `sfix64` or `ufix64` for 64-bit data.
Real	• `double` • `single`	HDL code generated with `double` or `single` data types in your MATLAB code can be used for simulation, but is not synthesizable. You can generate synthesizable code when you use these data types in your Simulink model. For more information, see: • "Simulink Blocks Supported by Using Native Floating Point" on page 14-137 • "Generate Target-Independent HDL Code with Native Floating-Point" on page 14-111 • "Signal and Data Type Support" on page 14-3
Character	`char`	–
Logical	`logical`	–
Fixed point	• Scaled (binary point only) fixed-point numbers • Custom integers (zero binary point)	Fixed-point numbers with slope (not equal to 1.0) and bias (not equal to 0.0) are not supported. Maximum word size for fixed-point numbers is 128 bits.
Vectors	• unordered `{N}` • row `{1, N}` • column `{N, 1}`	The maximum number of vector elements allowed is 2^32. Before a variable is subscripted, it must be fully defined.

<table>
<tr><th>Types</th><th>Supported Data Types</th><th>Restrictions</th></tr>
<tr><td>Matrices</td><td><code>{N, M}</code></td><td>Matrices are supported in the body of the design algorithm and as inputs to the top-level design function.

Matrices are not supported with the following HDL workflows:
• Cosimulation model generation
• FPGA-in-the-Loop
• IP Core Generation</td></tr>
<tr><td>Structures</td><td><code>struct</code></td><td>Arrays of structures are not supported.

For the IP Core Generation workflow, structures are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.

Structures are not supported as inputs and outputs at the top-level DUT ports for HLS code generation.</td></tr>
<tr><td>Enumerations</td><td><code>enumeration</code></td><td>If your target language is Verilog®, all enumeration member names must be unique within the design.

Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods:
• IP Core Generation workflow
• FPGA-in-the-Loop
• HDL Cosimulation
• HLS Code Generation Workflow

Enumerations are not supported as inputs and outputs at the top-level DUT ports for HLS code generation.</td></tr>
</table>

Global variables are not supported for HDL and HLS code generation.

Supported Operators

Note HDL and HLS code generated for large vector and matrix inputs to arithmetic operations can result in inefficient code. The code for these operators is not automatically pipelined.

Arithmetic Operators

Operation	Operator Syntax	Equivalent Function	Restrictions
Binary addition	A+B	plus(A,B)	Neither A nor B can be data type logical.
Matrix multiplication	A*B	mtimes(A,B)	HDL code generated for matrix arithmetic operations is not pipelined, and can result in inefficient code.
Arraywise multiplication	A.*B	times(A,B)	Neither A nor B can be data type logical.
Matrix power	A^B	mpower(A,B)	A and B must be scalar, and B must be an integer. HDL code generated for matrix arithmetic operations is not pipelined, and can result in inefficient code.
Arraywise power	A.^B	power(A,B)	A and B must be scalar, and B must be an integer.
Complex transpose	A'	ctranspose(A)	–
Matrix transpose	A.'	transpose(A)	
Matrix concat	[A B]	None	–
Matrix index	A(r c)	None	Before you use a variable, you must fully define it.

Logical Operators

Operation	Operator Syntax	M Function Equivalent	Notes
Logical And	A&B	and(A,B)	–
Logical Or	A|B	or(A,B)	–
Logical Xor	A xor B	xor(A,B)	–
Logical And (short circuiting)	A&&B	N/A	Use short circuiting logical operators within conditionals.
Logical Or (short circuiting)	A||B	N/A	Use short circuiting logical operators within conditionals.
Element complement	~A	not(A)	–

Relational Operators

Relation	Operator Syntax	Equivalent Function
Less than	`A<B`	`lt(A,B)`
Less than or equal to	`A<=B`	`le(A,B)`
Greater than or equal to	`A>=B`	`ge(A,B)`
Greater than	`A>B`	`gt(A,B)`
Equal	`A==B`	`eq(A,B)`
Not equal	`A~=B`	`ne(A,B)`

Control Flow Statements

HDL Coder supports the following control flow statements and constructs with restrictions.

<table>
<tr><th>Control Flow Statement</th><th>Restrictions</th></tr>
<tr><td>for</td><td>Do not use for loops without static bounds. A for loop must run for a constant number of iterations, which means it cannot have control flow statements based on conditional information that changes the number of iterations for the for loop, such as break or return.

Do not use the & and | operators within conditions of a for statement. Instead, use the && and || operators.

HDL Coder does not support nonscalar expressions in the conditions of for statements. Instead, use the all or any functions to collapse logical vectors into scalars.</td></tr>
<tr><td>if</td><td>Do not use the & and | operators within conditions of an if statement. Instead, use the && and || operators.

HDL Coder does not support nonscalar expressions in the conditions of if statements. Instead, use the all or any functions to collapse logical vectors into scalars.</td></tr>
<tr><td>switch</td><td>The conditional expression in a switch or case statement must use only:

• uint8, uint16, uint32, int8, int16, or int32 data types
• Scalar data

If multiple case statements make assignments to the same variable, the numeric type and fimath specification for that variable must be the same in every case statement.</td></tr>
</table>

The following control flow statements are not supported:

- `while`
- `break`
- `continue`
- `return`

- `parfor`

Avoid using the following vector functions, as they may generate loops containing `break` statements:

- `isequal`
- `bitrevorder`

See Also

`codegen` | `coder.HdlConfig`

More About

- “Hexadecimal and Binary Values”
- “Functions Supported for HDL and HLS Code Generation” on page 1-2
- “Bitwise Operations in MATLAB for HDL and HLS Code Generation” on page 1-58
- “Functions for Programming and Data Types”
- “For-Loop Best Practices for HDL Code Generation” on page 1-73

Persistent Variables and Persistent Array Variables

Persistent Variables

Persistent variables enable you to model registers. If you need to preserve state between invocations of your MATLAB algorithm, use persistent variables.

Before you use a persistent variable, you must initialize it with a statement specifying its size and type. You can initialize a persistent variable with either a constant value or a variable, as in the following examples:

```
% Initialize with a constant
persistent p;
if isempty(p)
    p = fi(0,0,8,0);
end

% Initialize with a variable
initval = fi(0,0,8,0);

persistent p;
if isempty(p)
    p = initval;
end
```

Use a logical expression that evaluates to a constant to test whether a persistent variable has been initialized, as in the preceding examples. Using a logical expression that evaluates to a constant ensures that the generated HDL and High-Level Synthesis (HLS) code for the test is executed only once, as part of the reset process.

You can initialize multiple variables within a single logical expression, as in the following example:

```
% Initialize with  variables
initval1 = fi(0,0,8,0);
initval2 = fi(0,0,7,0);

persistent p;
if isempty(p)
    x = initval1;
    y = initval2;
end
```

Note If persistent variables are not initialized as described above, extra sentinel variables can appear in the generated code. These sentinel variables can translate to inefficient hardware.

Persistent Array Variables

Persistent array variables enable you to model RAM.

By default, the HDL Coder software optimizes the area of your design by mapping persistent array variables to RAM. If persistent array variables are not mapped to RAM, they map to registers. RAM mapping can therefore reduce the area of your design in the target hardware.

To learn how persistent array variables map to RAM, see “Map Persistent Arrays and dsp.Delay Objects to RAM” on page 8-6 for HDL code generation and “Map Persistent Arrays to RAM” on page 13-2 for HLS code generation.

See Also

`codegen` | `coder.HdlConfig`

More About

- “Hexadecimal and Binary Values”
- “Functions Supported for HDL and HLS Code Generation” on page 1-2
- “Bitwise Operations in MATLAB for HDL and HLS Code Generation” on page 1-58
- “Functions for Programming and Data Types”

Complex Data Type Support

In this section...

"Declaring Complex Signals" on page 1-11

"Conversion Between Complex and Real Signals" on page 1-12

"Support for Vectors of Complex Numbers" on page 1-12

Declaring Complex Signals

The following MATLAB code declares several local complex variables. `x` and `y` are declared by complex constant assignment; `z` is created using the using the `complex()` function.

```
function [x,y,z] = fcn

% create 8 bit complex constants
x = uint8(1 + 2i);
y = uint8(3 + 4j);
z = uint8(complex(5, 6));
```

The following code example shows VHDL® code generated from the previous MATLAB code.

```
ENTITY complex_decl IS
    PORT (
        clk : IN std_logic;
        clk_enable : IN std_logic;
        reset : IN std_logic;
        x_re : OUT std_logic_vector(7 DOWNTO 0);
        x_im : OUT std_logic_vector(7 DOWNTO 0);
        y_re : OUT std_logic_vector(7 DOWNTO 0);
        y_im : OUT std_logic_vector(7 DOWNTO 0);
        z_re : OUT std_logic_vector(7 DOWNTO 0);
        z_im : OUT std_logic_vector(7 DOWNTO 0));
END complex_decl;


ARCHITECTURE fsm_SFHDL OF complex_decl IS

BEGIN
    x_re <= std_logic_vector(to_unsigned(1, 8));
    x_im <= std_logic_vector(to_unsigned(2, 8));
    y_re <= std_logic_vector(to_unsigned(3, 8));
    y_im <= std_logic_vector(to_unsigned(4, 8));
    z_re <= std_logic_vector(to_unsigned(5, 8));
    z_im <= std_logic_vector(to_unsigned(6, 8));
END fsm_SFHDL;
```

As shown in the example, complex inputs, outputs and local variables declared in MATLAB code expand into real and imaginary signals. The naming conventions for these derived signals are:

- Real components have the same name as the original complex signal, suffixed with the default string `'_re'` (for example, `x_re`). To specify a different suffix, set the **Complex real part postfix** option (or the corresponding `ComplexRealPostfix` CLI property).
- Imaginary components have the same name as the original complex signal, suffixed with the string `'_im'` (for example, `x_im`). To specify a different suffix, set the **Complex imaginary part postfix** option (or the corresponding `ComplexImagPostfix` CLI property).

A complex variable declared in MATLAB code remains complex during the entire length of the program.

Conversion Between Complex and Real Signals

The MATLAB code provides access to the fields of a complex signal via the `real()` and `imag()` functions, as shown in the following code.

```
function [Re_part, Im_part]= fcn(c)
% Output real and imaginary parts of complex input signal

Re_part = real(c);
Im_part = imag(c);
```

HDL Coder supports these constructs, accessing the corresponding real and imaginary signal components in generated HDL code. In the following Verilog code example, the MATLAB complex signal variable `c` is flattened into the signals `c_re` and `c_im`. Each of these signals is assigned to the output variables `Re_part` and `Im_part`, respectively.

```
module Complex_To_Real_Imag (clk, clk_enable, reset, c_re, c_im, Re_part, Im_part );

    input clk;
    input clk_enable;
    input reset;
    input [3:0] c_re;
    input [3:0] c_im;
    output [3:0] Re_part;
    output [3:0] Im_part;

    // Output real and imaginary parts of complex input signal
    assign Re_part = c_re;
    assign Im_part = c_im;
```

Support for Vectors of Complex Numbers

You can generate HDL code for vectors of complex numbers. Like scalar complex numbers, vectors of complex numbers are flattened down to vectors of real and imaginary parts in generated HDL code.

For example in the following script `t` is a complex vector variable of base type `ufix4` and size `[1,2]`.

```
function y = fcn(u1, u2)

t = [u1 u2];
y = t+1;
```

In the generated HDL code the variable `t` is broken down into real and imaginary parts with the same two-element array.

```
VARIABLE t_re : vector_of_unsigned4(0 TO 3);
VARIABLE t_im : vector_of_unsigned4(0 TO 3);
```

The real and imaginary parts of the complex number have the same vector of type `ufix4`, as shown in the following code.

```
TYPE vector_of_unsigned4 IS ARRAY (NATURAL RANGE <>) OF unsigned(3 DOWNTO 0);
```

Complex vector-based operations (`+`,`-`,`*` etc.,) are similarly broken down to vectors of real and imaginary parts. Operations are performed independently on the elements of such vectors, following MATLAB semantics for vectors of complex numbers.

In the VHDL, Verilog or SystemVerilog code generated from MATLAB code, complex vector ports are always flattened. If complex vector variables appear on inputs and outputs, real and imaginary vector components are further flattened to scalars.

In the following code, `u1` and `u2` are scalar complex numbers and `y` is a vector of complex numbers.

```
function y = fcn(u1, u2)
```

```
t = [u1 u2];
y = t+1;
```

This generates the following port declarations in a VHDL entity definition.

```
ENTITY _MATLAB_Function IS
    PORT (
        clk : IN std_logic;
        clk_enable : IN std_logic;
        reset : IN std_logic;
        u1_re : IN vector_of_std_logic_vector4(0 TO 1);
        u1_im : IN vector_of_std_logic_vector4(0 TO 1);
        u2_re : IN vector_of_std_logic_vector4(0 TO 1);
        u2_im : IN vector_of_std_logic_vector4(0 TO 1);
        y_re : OUT vector_of_std_logic_vector32(0 TO 3);
        y_im : OUT vector_of_std_logic_vector32(0 TO 3));
END _MATLAB_Function;
```

See Also

`codegen` | `coder.HdlConfig`

More About

- "Hexadecimal and Binary Values"
- "Functions Supported for HDL and HLS Code Generation" on page 1-2
- "Bitwise Operations in MATLAB for HDL and HLS Code Generation" on page 1-58

Complex Data Type Support for High-Level Synthesis Code Generation

In this section...

"Declaring Complex Signals" on page 1-14
"Conversion Between Complex and Real Signals" on page 1-14
"Support for Vectors of Complex Numbers" on page 1-15

Declaring Complex Signals

The following MATLAB code declares several local complex variables. `x` and `y` are declared by complex constant assignment; `z` is created using the using the `complex()` function.

```
function [x,y,z] = fcn

% create 8 bit complex constants
x = uint8(1 + 2i);
y = uint8(3 + 4j);
z = uint8(complex(5, 6));
```

The following code example shows High-Level Synthesis (HLS) code generated from the previous MATLAB code.

```
class fcnClass
{
 public:
  void fcn(sc_fixed<16,3> &x_re, sc_fixed<16,3> &x_im, sc_fixed<16,4> &y_re,
           sc_fixed<16,4> &y_im, sc_fixed<16,4> &z_re, sc_fixed<16,4> &z_im)
  {
    x_re = sc_fixed<16,3>(1.0);
    x_im = sc_fixed<16,3>(2.0);
    y_re = sc_fixed<16,4>(3.0);
    y_im = sc_fixed<16,4>(4.0);
    z_re = sc_fixed<16,4>(5.0);
    z_im = sc_fixed<16,4>(6.0);
  }
};
```

As shown in the example, complex inputs, outputs and local variables declared in MATLAB code expand into real and imaginary signals. The naming conventions for these derived signals are:

- Real components have the same name as the original complex signal, suffixed with the default string `'_re'` (for example, `x_re`). To specify a different suffix, set the **Complex real part postfix** option (or the corresponding `ComplexRealPostfix` CLI property).
- Imaginary components have the same name as the original complex signal, suffixed with the string `'_im'` (for example, `x_im`). To specify a different suffix, set the **Complex imaginary part postfix** option (or the corresponding `ComplexImagPostfix` CLI property).

A complex variable declared in MATLAB code remains complex during the entire length of the program.

Conversion Between Complex and Real Signals

The MATLAB code provides access to the fields of a complex signal via the `real()` and `imag()` functions, as shown in the following code.

```
function [Re_part, Im_part]= fcn(c)
% Output real and imaginary parts of complex input signal
```

```
Re_part = real(c);
Im_part = imag(c);
```

HDL Coder supports these constructs, accessing the corresponding real and imaginary signal components in generated HLS code. In the following HLS code, the MATLAB complex signal variable `c` is flattened into the signals `c_re` and `c_im`. Each of these signals is assigned to the output variables `Re_part` and `Im_part`, respectively.

```
class fcnClass
{
 public:
  void fcn(sc_fixed<16,3> c_re, sc_fixed<16,3> c_im, sc_fixed<16,3> &Re_part,
           sc_fixed<16,3> &Im_part)
  {
    /*  Output real and imaginary parts of complex input signal */
    Re_part = c_re;
    Im_part = c_im;
  }
};
```

Support for Vectors of Complex Numbers

You can generate HLS code for vectors of complex numbers. Like scalar complex numbers, vectors of complex numbers are flattened down to vectors of real and imaginary parts in generated HLS code.

For example in the following script `t` is a complex vector variable of base type `ufix4` and size `[1,2]`.

```
function y = fcn(u1, u2)

t = [u1 u2];
y = t+1;
```

In the generated HLS code the variable `t` is broken down into real and imaginary parts with the same two-element array. The real and imaginary parts of the complex number have the same vector of type `ufix4`, as shown in the following code.

```
sc_uint<4> t_re[2];
sc_uint<4> t_im[2];
```

In the above MATLAB script, `u1` and `u2` are scalar complex numbers and `y` is a vector of complex numbers. This generates the following function definition in the HLS code.

```
void fcn(sc_uint<4> u1_re, sc_uint<4> u1_im, sc_uint<4> u2_re, sc_uint<4>
          u2_im, sc_uint<5> (&y_re)[2], sc_uint<5> (&y_im)[2])
```

Complex vector-based operations (`+`,`-`,`*` etc.,) are similarly broken down to vectors of real and imaginary parts. Operations are performed independently on the elements of such vectors, following MATLAB semantics for vectors of complex numbers.

See Also

`codegen` | `coder.HdlConfig`

More About

- “Hexadecimal and Binary Values”
- “Functions Supported for HDL and HLS Code Generation” on page 1-2
- “Bitwise Operations in MATLAB for HDL and HLS Code Generation” on page 1-58

HDL Code Generation for System Objects

In this section...

"Why Use System Objects?" on page 1-16
"Predefined System Objects" on page 1-16
"User-Defined System Objects" on page 1-16
"Limitations of HDL Code Generation for System Objects" on page 1-16
"System object Examples for HDL Code Generation" on page 1-17

HDL Coder supports both predefined and user-defined System objects for code generation.

Why Use System Objects?

System objects provide a design advantage because:

- You can save time during design and testing by using existing System object components.
- You can design and qualify custom System objects for reuse in multiple designs.
- You can define your algorithm in a System object once, and reuse multiple instances of it in a single MATLAB design.

  This idiom cannot be used with MATLAB functions that have state. For example, if the algorithm has state and requires the use of persistent variables, that function cannot be instantiated multiple times in a design. Instead, you would need to copy and rename the function for each instance.
- HDL code that you generate from System objects is modular and more readable.

Predefined System Objects

Predefined System objects that are available with MATLAB, DSP System Toolbox™, and Communications Toolbox™ are supported for HDL code generation. For a list, see "Predefined System Objects Supported for HDL Code Generation" on page 1-50.

User-Defined System Objects

You can create user-defined System objects for HDL code generation. For an example, see "Generate Code for User-Defined System Objects" on page 1-53.

Note To use a MATLAB class for HDL code generation, the class must be a System object.

Limitations of HDL Code Generation for System Objects

The following limitations apply to HDL code generation for System objects:

- Your design can call the System object only once.
- You must not call the object inside a nested conditional statement, such as a nested loop, `if` statement, or `switch` statement.

- You must not call the object inside a conditional statement that contains a matrix indexing operation.
- A System object must be declared persistent if it has state.

  A System object has state when it has a tunable private or public property, or a property with the `DiscreteState` attribute.
- You can use the `dsp.Delay` System object only in feed-forward delay modeling.
- System objects in cell arrays are not supported.
- Enumerations are not supported.
- Global variables are not supported.

Supported Methods

For predefined System objects, calling the object itself is supported for HDL code generation, but no other object functions are supported.

For user-defined System objects, either the `step` method, or the `output` and `update` methods, are supported for HDL code generation.

Additional Restrictions for Predefined System Objects

Predefined System objects are not supported for HDL code generation from within a MATLAB System block.

Additional Restrictions for User-Defined System Objects

In addition to the limitations for all System objects, the following restrictions apply to user-defined System objects for HDL code generation:

- In the `setupImpl` and `resetImpl` methods, if you assign values to properties or variables, the values must be constants.
- If your design uses the `output` and `update` methods, it can call each method only once per System object.
- Initial and reset values for properties must be compile-time constant.
- User-defined System objects must not be public properties.
- A System object with multiple outputs cannot be called within a conditional statement.

System object Examples for HDL Code Generation

To learn how to use System objects for HDL code generation, view the MATLAB designs in the following examples:

- “HDL Code Generation from System Objects” on page 1-18
- “Model State with Persistent Variables and System Objects” on page 1-56
- “Generate Code for User-Defined System Objects” on page 1-53
- “Integrate Custom HDL Code Into MATLAB Design” on page 5-15

HDL Code Generation from System Objects

This example shows how to generate HDL code from MATLAB® code that contains System objects. For more information on System objects, see "HDL Code Generation for System Objects" on page 1-16.

MATLAB Design

The MATLAB code used in this example implements a simple symmetric FIR filter and uses the `hdl.Delay` System object to model state. This example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_sysobj_ex';
testbench_name = 'mlhdlc_sysobj_ex_tb';
```

Look at the MATLAB design.

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
%
% Design pattern covered in this example:
%  Filter states modeled using HDL System object (hdl.Delay)
%  Filter coefficients passed in as parameters to the design
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [y_out, delayed_xout] = mlhdlc_sysobj_ex(x_in, h_in1, h_in2, h_in3, h_in4)
% Symmetric FIR Filter

persistent h1 h2 h3 h4 h5 h6 h7 h8;
if isempty(h1)
    h1 = hdl.Delay;
    h2 = hdl.Delay;
    h3 = hdl.Delay;
    h4 = hdl.Delay;
    h5 = hdl.Delay;
    h6 = hdl.Delay;
    h7 = hdl.Delay;
    h8 = hdl.Delay;
end

h1p = step(h1, x_in);
h2p = step(h2, h1p);
h3p = step(h3, h2p);
h4p = step(h4, h3p);
h5p = step(h5, h4p);
h6p = step(h6, h5p);
h7p = step(h7, h6p);
h8p = step(h8, h7p);

a1 = h1p + h8p;
```

```
a2 = h2p + h7p;
a3 = h3p + h6p;
a4 = h4p + h5p;

m1 = h_in1 * a1;
m2 = h_in2 * a2;
m3 = h_in3 * a3;
m4 = h_in4 * a4;

a5 = m1 + m2;
a6 = m3 + m4;

% filtered output
y_out = a5 + a6;
% delayout input signal
delayed_xout = h8p;

end
```

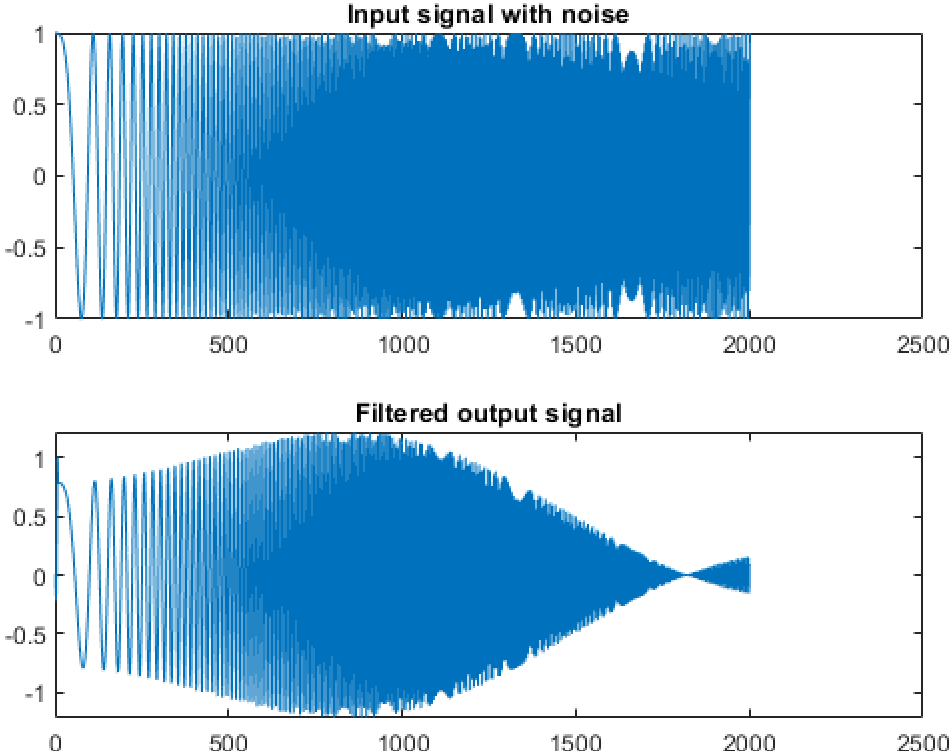
```
type(testbench_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB test bench for the FIR filter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2015 The MathWorks, Inc.

clear mlhdlc_sysobj_ex;

x_in = cos(2.*pi.*(0:0.001:2).*(1+(0:0.001:2).*75)).';


h1 = -0.1339;
h2 = -0.0838;
h3 = 0.2026;
h4 = 0.4064;

len = length(x_in);
y_out_sysobj = zeros(1,len);
x_out_sysobj = zeros(1,len);
a = 10;

for ii=1:len
    data = x_in(ii);
    % call to the design 'sfir' that is targeted for hardware
    [y_out_sysobj(ii), x_out_sysobj(ii)] = mlhdlc_sysobj_ex(data, h1, h2, h3, h4);
end


figure('Name', [mfilename, '_plot']);
subplot(2,1,1);
plot(1:len,x_in); title('Input signal with noise');
subplot(2,1,2);
plot(1:len,y_out_sysobj); title('Filtered output signal');
```

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sysobj_ex_tb
```



Create a New HDL Coder™ Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sysobj_prj
```

Next, add the file `mlhdlc_sysobj_ex.m` to the project as the MATLAB Function and `mlhdlc_sysobj_ex_tb.m` as the MATLAB Test Bench.

For a more complete tutorial on creating and populating MATLAB HDL Coder projects, see "Get Started with MATLAB to HDL Workflow".

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor. In the Workflow Advisor, right-click the **Code Generation** step. Choose the option **Run to selected task** to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

Supported System objects

For a list of System objects supported for HDL code generation, see “Predefined System Objects Supported for HDL Code Generation” on page 1-50.

See Also

`hdl.Delay` | `dsp.Delay`

See Also

Related Examples

- “HDL Code Generation from hdl.RAM System Object” on page 1-39

HDL Code Generation for Streaming Matrix Inverse System Object

This example shows how HDL Coder™ implements a streaming mode of matrix inverse operation with configurable sizes.

What is inverse of a matrix

A matrix X is invertible if there exists a matrix Y of the same size such that XY = YX = I, where I is the Identity matrix. The matrix Y is called inverse of X. A matrix that has no inverse is singular. A square matrix is singular only when its determinant is exactly zero.

Matrix inverse computation involves following steps:

1. Cofactor matrix calculation
2. Transpose of cofactor matrix
3. Multiply reciprocal of determinant of input matrix with transpose of cofactor matrix

Example:

```
             A = [4 12 -16;12 37 -43;-16 -43 98];
Cofactors of 'A' will be calculated from matrix of minors
        cof(A) = [1777 488 76;488 136 20;76 20 4];

Transpose of cofactor matrix will be
     (cof(A))' = [1777 488 76;488 136 20;76 20 4];

Multiply reciprocal of determinant of 'A' with transpose of cofactor matrix
          Ainv = (1/det(A)) * (cof(A))'
               = [49.3611 -13.5556 2.1111;-13.5556 3.7778 -0.5556;2.1111 -0.5556 0.1111];
```

Matrix Inverse: Gauss-Jordan elimination

To find the inverse of matrix A using Gauss-Jordan elimination, we must find elementary row operations that reduce A to identity matrix(I) and then perform the same operations on Identity matrix(I) to obtain Ainv.

Computation of Matrix Inverse using Gauss-Jordan elimination: Let start with matrix A, and write it down with an Identity matrix next to it: [A | I]

The goal is to make A an identity matrix by applying row transformations and right-hand side matrix I also participated in the row transformations, finally reduced to Ainv.

Computation of Ainv involves following steps:

1. swapping rows
2. make the diagonal elements as 1
3. make the non-diagonal elements as 0

Example:

```
                    (A)       (I)
                  1  2  3 | 1  0  0
```

```
      [A | I] =   2  5  3 | 0  1  0
                  1  0  8 | 0  0  1

Find the element with maximum value in the first column and swap the current
row with maximum element row
          swap R1 and R2 rows as R2 contains the largest values.

                  2  5  3 | 0  1  0
              =   1  2  3 | 1  0  0
                  1  0  8 | 0  0  1

Make the diagonal element in the first column as '1'
          R1 --> R1/2

                  1 2.5 1.5 | 0 0.5 0
              =   1   2   3 | 1   0 0
                  1   0   8 | 0   0 1
Make the non-diagonal elements in the first column as '0'
          R2 --> R2 - R1
          R3 --> R3 - R1

                  1  2.5 1.5 | 0  0.5 0
              =   0 -0.5 1.5 | 1 -0.5 0
                  0 -2.5 6.5 | 0 -0.5 1

Now column 1 has diagonal elements '1' and other elements as
'0'. This procedure is repeated for remaining columns and matrix A
will be reduced to identity matrix, Identity matrix will be reduced to Ainv.

                  1  0  0 | -40  16   9
              =   0  1  0 |  13  -5  -3
                  0  0  1 |   5  -2  -1
                    (I)        (Ainv)
```

Matrix Inverse: Cholesky decomposition

Matrix Inverse using cholesky decomposition supports only symmetric positive definite matrices. Positive definite means all the eigenvalues of the matrix should be positive.

Given a symmetric positive definite matrix A:

```
           A   = L * L',             L  is the lower triangular matrix
                                     L' is the transpose of L

       inv(A)  = inv(L * L')
               = inv(L') * inv(L)
               = (inv(L))' * inv(L)

       Ainv    = Linv' * Linv,       Linv is the inverse of lower triangular matrix
                                     Ainv is the inverse of input matrix
```

Computation of Ainv involves following steps:

1. Lower triangular matrix computation(L)
2. Inverse of lower triangular matrix(Linv)
3. Multiplication of transpose of Linv with Linv

Example:

```
                A = [4 12 -16;12 37 -43;-16 -43 98];

Lower triangular matrix(L) will be computed using cholesky decomposition
                L = [2 0 0;6 1 0;-8 5 3];

Linv will be computed using forward substitution method
            Linv = [0.5 0 0;-3 1 0;6.3333 -1.6667 0.3333];

Multiply transpose of Linv with Linv
            Ainv = Linv' * Linv
                  = [49.3611 -13.5556 2.1111;-13.5556 3.7778 -0.5556;2.1111 -0.5556 0.1111];
```

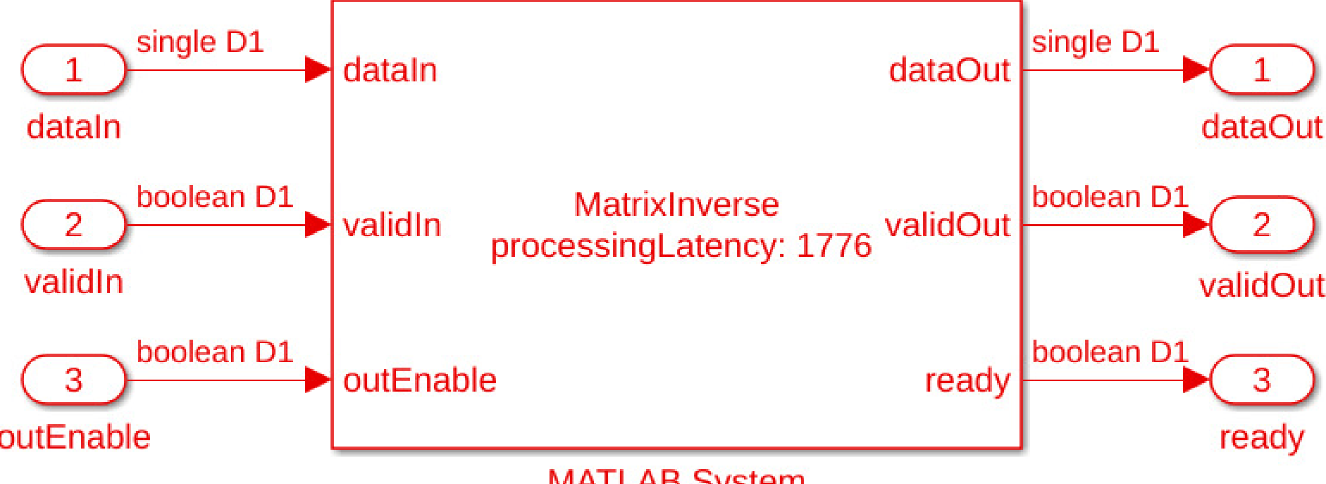
Benefits of using Gauss-Jordan Elimination

- Gauss-Jordan Elimination supports all square matrices.
- Gauss-Jordan Elimination supports both `single` and `double` data types.

Restrictions for Cholesky implementation

- Matrices for which inverse is to computed must be symmetric positive-definite.
- Input data types of the matrices must be `single` and block must be used in the `Native Floating Point` mode.
- Input matrices must not be larger than 64-by-64 in size.

Matrix Inverse Subsystem Interface:

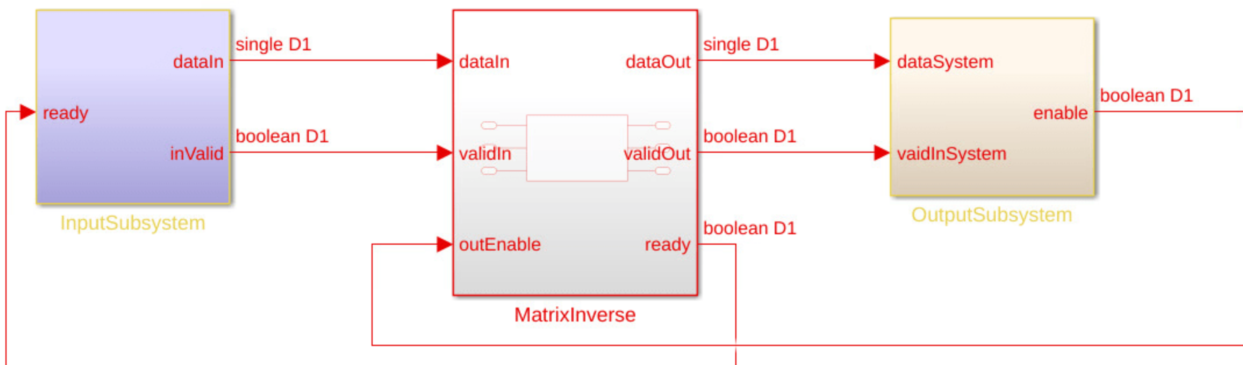


Matrix Inverse ports description:

Input ports		Output ports	
dataIn	Input data to the module	dataOut	Output data from the module
validIn	Valid signal for input data	validOut	Valid signal for output data
outEnable	Input signal that indicates downstream module is ready to take the output data from processing module	ready	Output signal that indicates processing module is ready to accept the input data in row major from upstream module

Matrix Inverse Implementation

This example shows streaming matrix inverse implementation

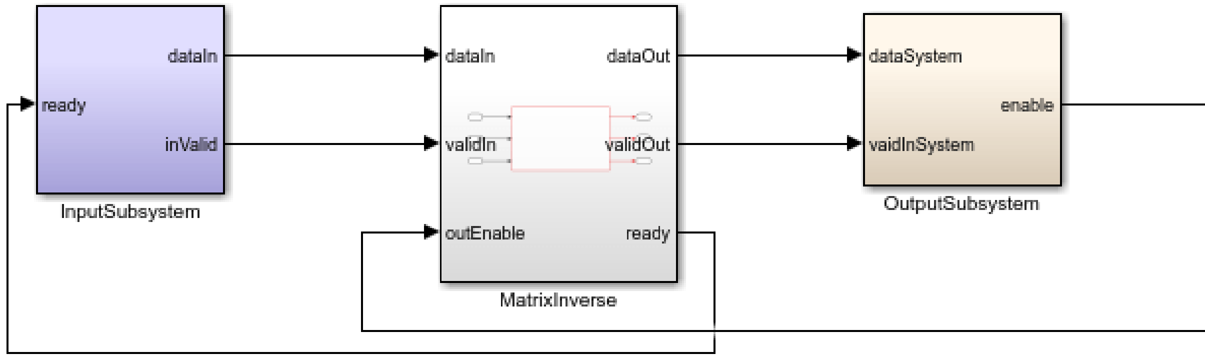


This example model contains three subsystems: `InputSubsystem`, `MatrixInverse`, and `OutputSubsystem`. The `InputSubsystem` is the upstream module that serializes the matrix input to the processing module when the `ready` signal is enabled. The `OutputSubsystem` is the downstream module that deserializes the data from the processing module to a matrix output when the `outEnable` signal is enabled. The `MatrixInverse` is a processing module that implements the matrix inverse operation.

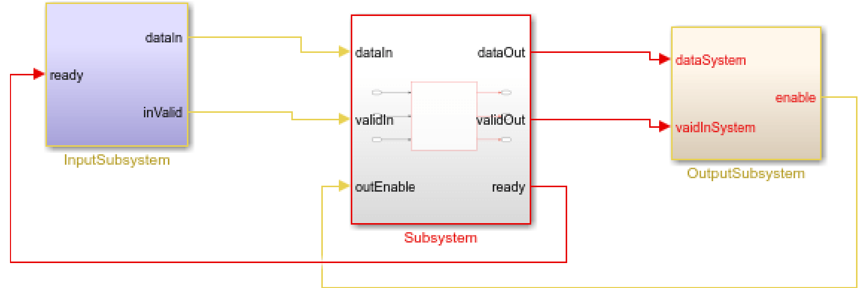
```
open_system('hdlcoder_streaming_mat_inv_max_lat_cholesky');
open_system('hdlcoder_streaming_mat_inv_max_lat_gauss_jordan');
```

Copyright 2017-2019 The MathWorks, Inc.

This example shows streaming matrix implementation using Cholesky decomposition

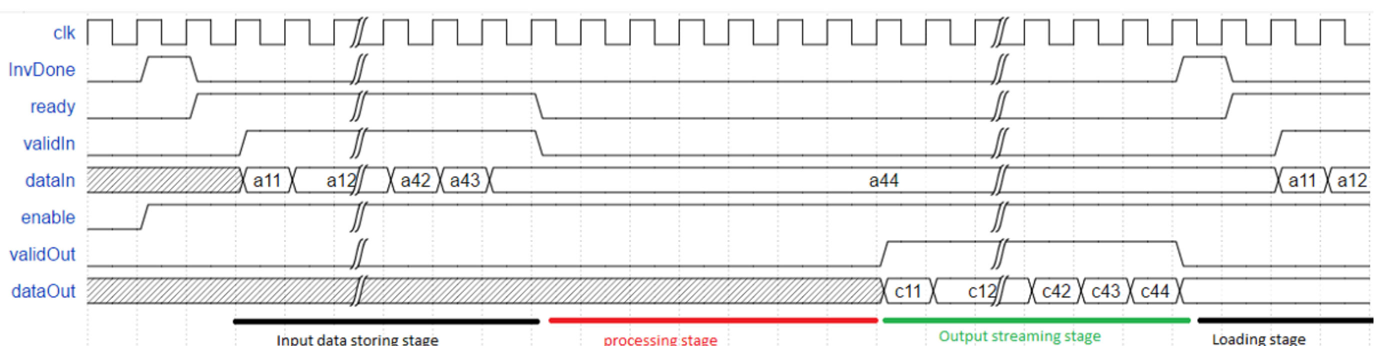


Copyright 2017-2019 The MathWorks, Inc. This example shows streaming matrix implementation using Gauss-Jordan elimination

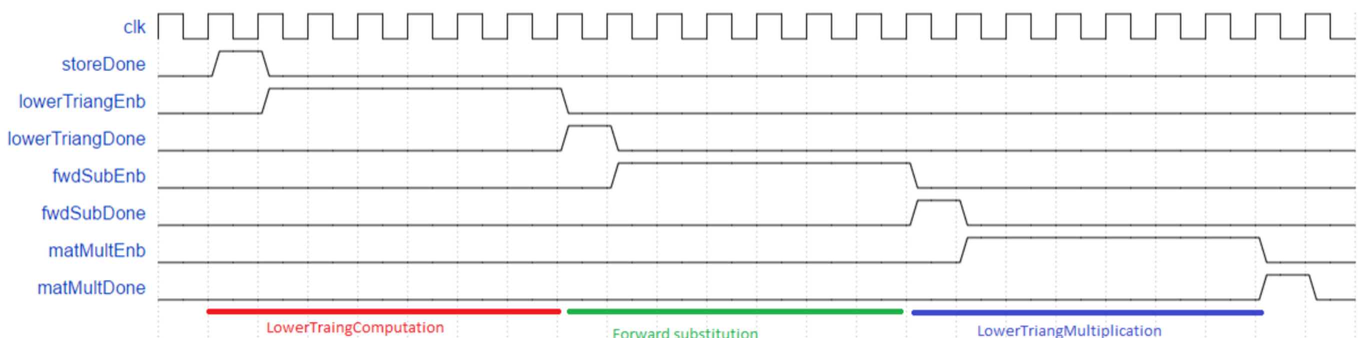


Matrix Inverse Timing diagrams:

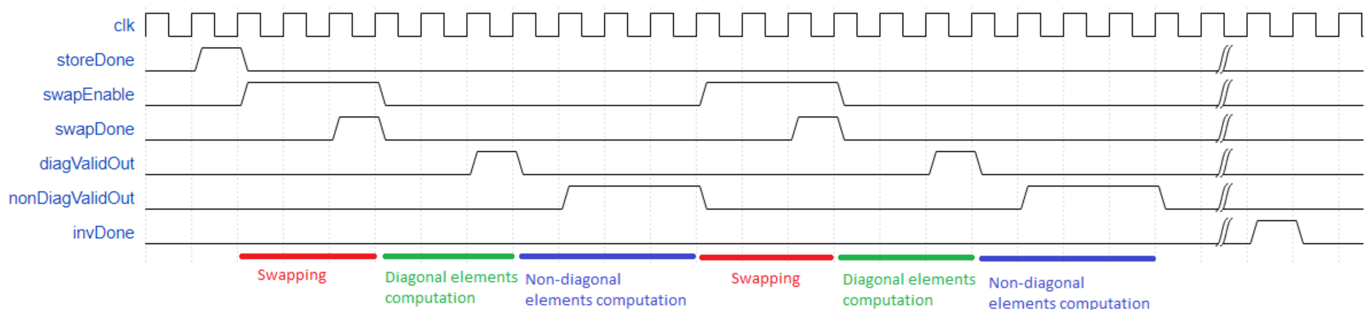
Timing diagram for complete system:



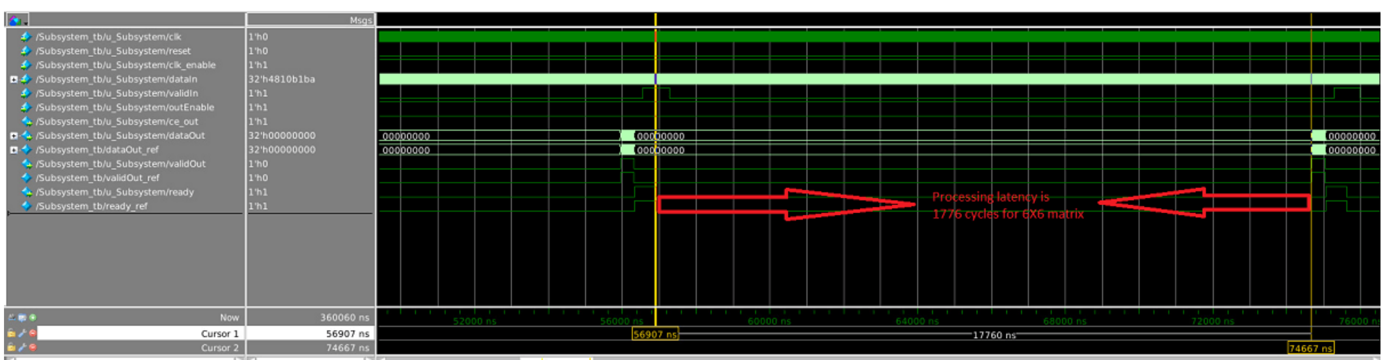
Timing diagram for processing stage(Cholesky decomposition):



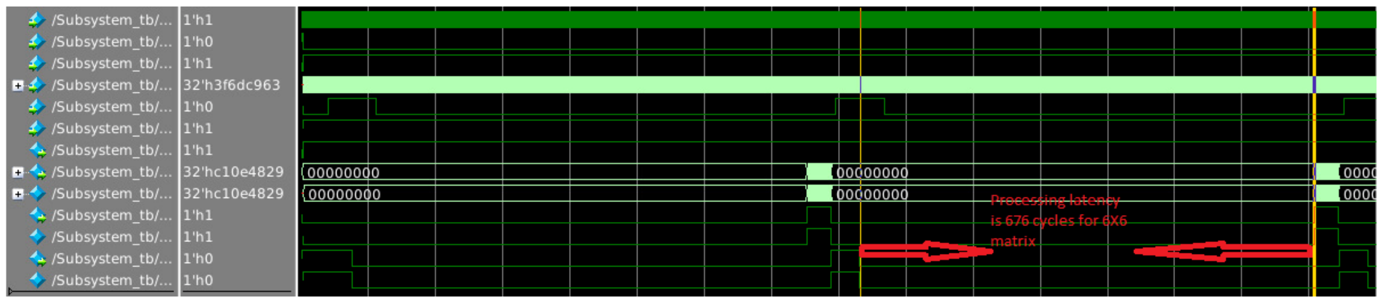
Timing diagram for processing stage(Gauss-Jordan elimination):



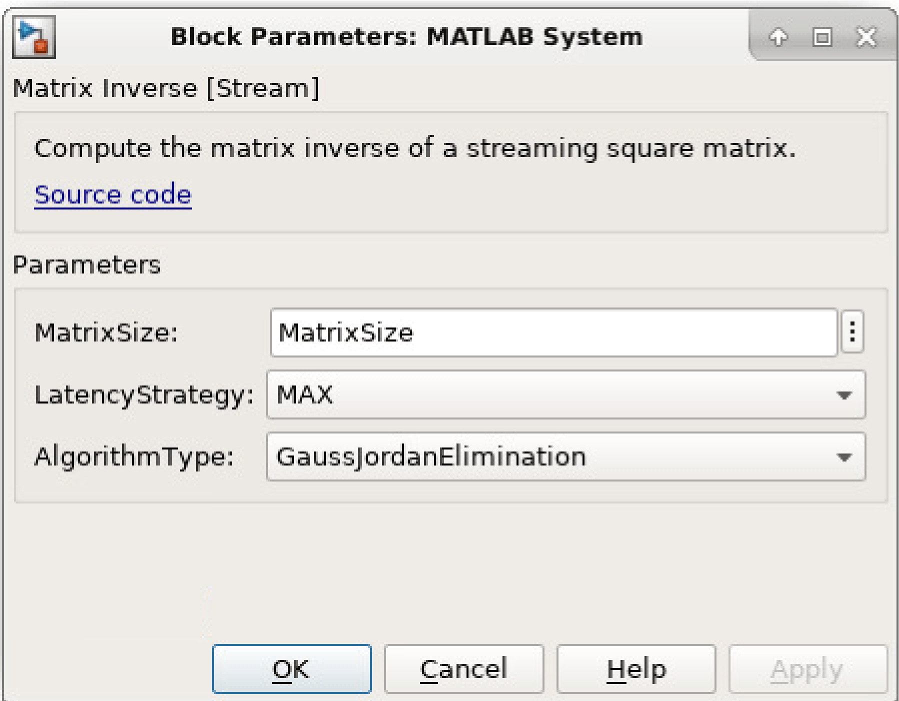
Modelsim result waveform(Cholesky decomposition)



Modelsim result waveform(Gauss-Jordan elimination)



Matrix Inverse Block parameters:




```
MatrixSize      : Enter size of input matrix as a positive integer.

LatencyStrategy : Select latency strategy from drop down menu
                  ({'ZERO, 'MIN', 'MAX'}) which should be same as HDL coder
                  latency strategy. User can see processing latency
                  based on the latency strategy.
AlgorithmType   : Select algorithm type from drop down menu({'CholeskyDecomposition',
                  'GaussJordanElimination'})
```

Matrix Inverse Block usage

1. Set block parameters of MATLAB System block.
2. Select input matrix based on the matrix size.
3. Generate HDL code for MatrixInverse subsystem.

Generated code and Generated model

After running code generation for MatrixInverse subsystem, generated code will be

```
>> makehdl(gcb, 'nat', 'on')
### Generating HDL for 'hdlcoder_streaming_matrix_inverse_max_latency/MatrixInverse'.
### Using the config set for model hdlcoder_streaming_matrix_inverse_max_latency for HDL code generation parameters.
### Starting HDL check.
### One or more feedback loops in the model are inhibiting optimizations. To highlight these loops in your model, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/clearhighlighting.m
### Begin Verilog Code Generation for 'hdlcoder_streaming_matrix_inverse_max_latency'.
### Working on RowColCounter as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/RowColCounter.v.
### Working on ReadySignalGenerator as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/ReadySignalGenerator.v.
### Working on StoringDone as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/StoringDone.v.
### Working on WrEnbStore as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/WrEnbStore.v.
### Working on WrAddrStore as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/WrAddrStore.v.
### Working on WrDataStore as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/WrDataStore.v.
### Working on InputDataStoreMemoryControl as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/InputDataStoreMemoryControl.v.
### Working on InputMatrixStoreControl as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/InputMatrixStoreControl.v.
### Working on LowerTriangEnable as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/LowerTriangEnable.v.
### Working on LowerTriangReadEnable as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/LowerTriangReadEnable.v.
### Working on LowerTriangDataValidIn as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/LowerTriangDataValidIn.v.
### Working on LTMemReadControl as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/LTMemReadControl.v.
### Working on LTRowCounter as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/LTRowCounter.v.
### Working on LTColumnCounter as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/LTColumnCounter.v.
### Working on LTRowColCounter as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/LTRowColCounter.v.
### Working on LTProcessController as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/LTProcessController.v.
### Working on DiagDataSelector as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/DiagDataSelector.v.
### Working on DiagDataComputation/nfp_mul_single as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/nfp_mul_single.v.
### Working on DiagDataComputation/nfp_add_single as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/nfp_add_single.v.
### Working on DiagDataComputation/nfp_sqrt_single as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/nfp_sqrt_single.v.
### Working on DiagDataComputation/nfp_relop_single as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/nfp_relop_single.v.
### Working on DiagDataComputation/nfp_sub_single as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/nfp_sub_single.v.
### Working on DiagDataComputation as hdlsrc/hdlcoder_streaming_matrix_inverse_max_latency/DiagDataComputation.v.
```

Generated model contains the MatrixInverse MATLAB System block. During modelsim simulation code generation outputs are compared with MATLAB System block outputs.

Synthesis statistics

Cholesky decomposition

SynthesisTool : Altera Quartus II 16.0
SyntheisToolChipFamily : Stratix V
SynthesisToolDeviceName : 5SEE9F45C2

Size	Fmax	ALMs	LABs	DSPs	Comb ALUTs	RAMs	Latency
4X4	316.36	7240	1223	12	11386	45	958
8X8	267.59	12889	2193	24	20415	76	2803
16X16	237.47	24219	4190	48	37884	138	9122
24X24	223.31	35738	6133	72	56826	205	18961
32X32	198.97	46949	8028	96	73918	266	32309

SynthesisTool : Xilinx Vivado 2017.4
SyntheisToolChipFamily : xc7v2000t
SynthesisToolDeviceName : fhg1761

Size	Fmax	Slices	LUTs	DSPs	Latency
4X4	233.81	4783	13968	24	958
8X8	261.85	8700	24362	48	2803
16X16	169.69	15466	44077	96	9122
24X24	221.29	25543	79260	144	18961
32X32	161.29	26371	80258	192	32309

Gauss-Jordan elimination

SynthesisTool : Altera Quartus II 18.1
SyntheisToolChipFamily : Stratix V
SynthesisToolDeviceName : 5SEE9F45C2

Size	Fmax	ALMs	LABs	DSPs	Comb ALUTs	RAMs	Latency
4X4	354.99	3509	650	2	4752	29	356
8X8	334.67	4793	826	2	6665	35	1156
16X16	288.77	7119	1191	2	9213	51	5636
24X24	250.88	9356	1617	2	11686	67	16516
32X32	237.98	12007	2072	2	14562	83	36868

SynthesisTool : Xilinx Vivado 2018.3
SyntheisToolChipFamily : xc7v2000t
SynthesisToolDeviceName : fhg1761

Size	Fmax	Slices	LUTs	DSPs	Latency
4X4	254.97	2462	6657	2	356
8X8	208.72	2885	7573	2	1156
16X16	151.03	4234	10648	2	5636
24X24	154.14	4828	11291	2	16516
32X32	113.39	6375	13130	2	36868

Related Links

- `inv`
- `chol`
- `eig`
- `rref`

HDL Code Generation for Streaming Matrix Multiply System Object

This example shows how HDL Coder™ implements a streaming mode of matrix multiplication with configurable sizes.

How to Multiply Matrices

Let A, B be two matrices then C= A * B is the matrix multiplication of A and B. If A is an m-by-p and B is an p-by-n matrix, then C is an m-by-n matrix defined by

```
C(i,j) =  A(i,1)B(1,j) + A(i,2)B(2,j) + -------- + A(i,p)B(p,j)
```

This inner definition says that C(i,j) is the inner product of ith row of A with the jth column of B

For non scalar A and B, the number of columns of A must equal to the number of rows of B

Example:

```
A = [1 3 5;2 4 7];             (2 X 3 matrix)

B = [-5 8 11;3 9 21;4 0 8];    (3 X 3 matrix)
```

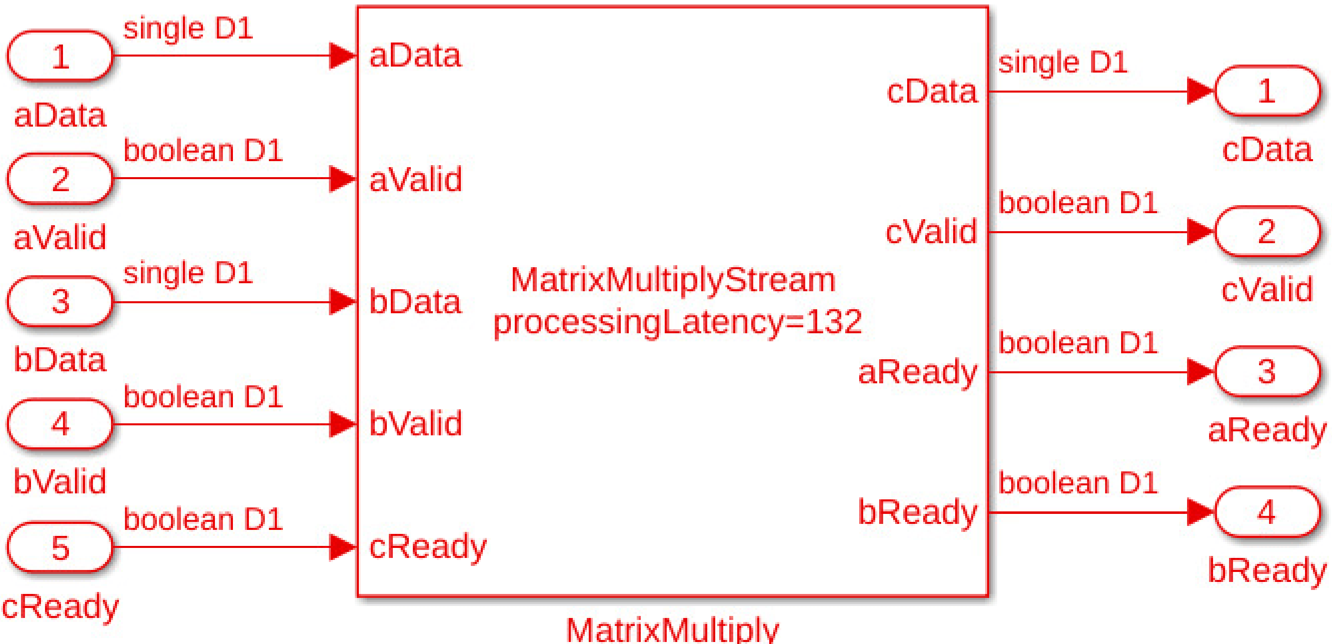
After calculating inner product of rows of A with columns of B

```
C = [24 35 114;30 52 162];     (2 X 3 matrix)
```

Introduction

Streaming Matrix Multiply supports multiplication of two matrices with configurable matrix sizes and dot product size. Dot product size is equal to the number of multipliers used for computation. This block can accepts serialized input data from matrix in row major or column major format.

Matrix Multiply Interface:

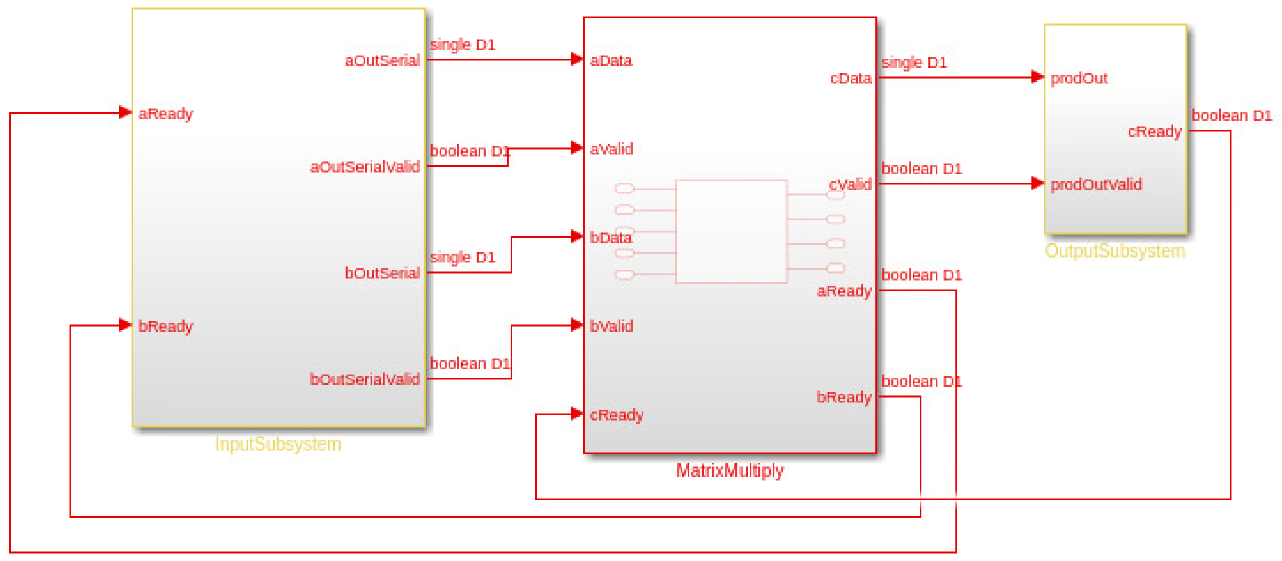


Matrix Multiply ports description:

Input ports		Output ports	
aData	Matrix-A input data to the module	cData	Matrix-C output data from processing module
aValid	Valid signal for matrix-A input data	cValid	Valid signal for matrix-C output data
bData	Matrix-B input data to the module	aReady	Output signal that indicates the processing module is ready to accept the matrix-A input data from upstream module
bValid	Valid signal for matrix-B input data	bReady	Output signal that indicates the processing module is ready to accept the matrix-B input data from upstream module
cReady	Input signal that indicates downstream module is ready to take output data from processing module		

MatrixMultiply Implementation

This example shows streaming matrix multiplication operation

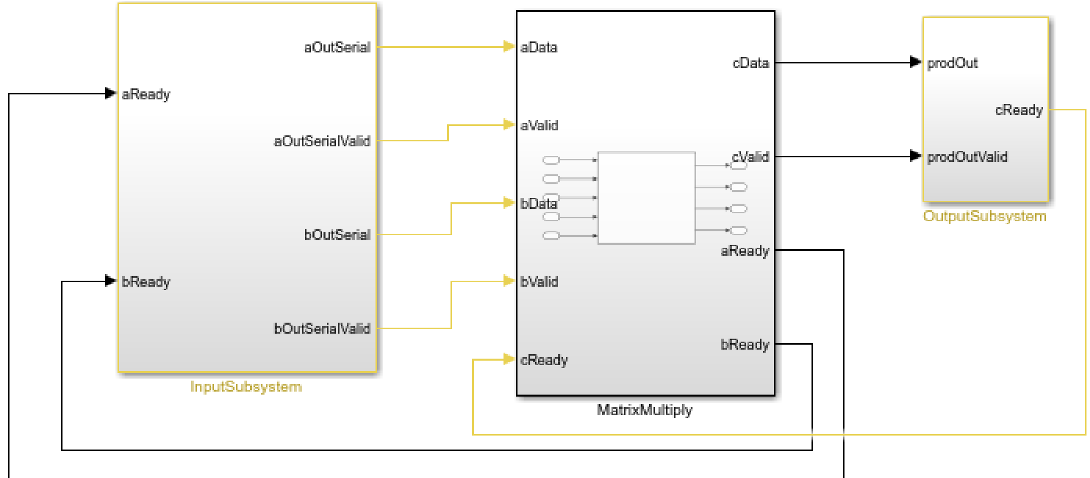


This example model contains three subsystems: `InputSubsystem`, `MatrixMultiply` and `OutputSubsystem`. The `InputSubsystem` is the upstream module that serializes the matrix inputs(A,B) to the processing module when `aReady` and `bReady` signals are enabled. The `OutputSubsystem` is the downstream module that deserializes the data from the processing module to matrix output(C) when the `cReady` signal is enabled. The `MatrixMultiply` is a processing module that implements the matrix multiplication.

```
open_system('hdlcoder_streaming_matrix_multiply_max_latency');
```

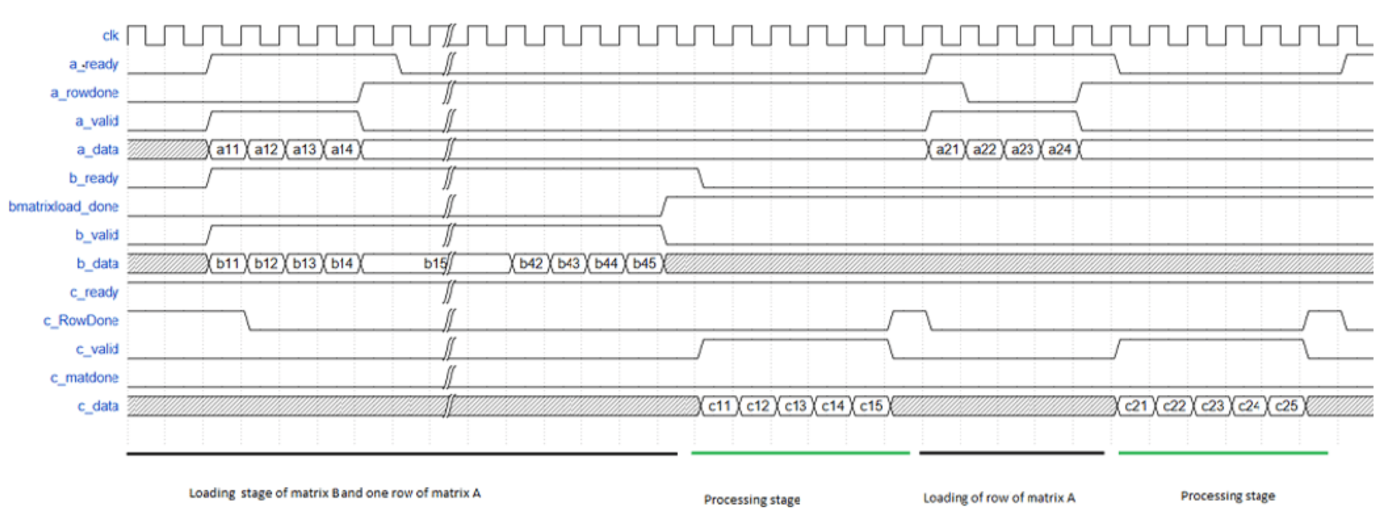
This example shows streaming matrix multiplication operation

Copyright 2018-2019 The MathWorks, Inc.

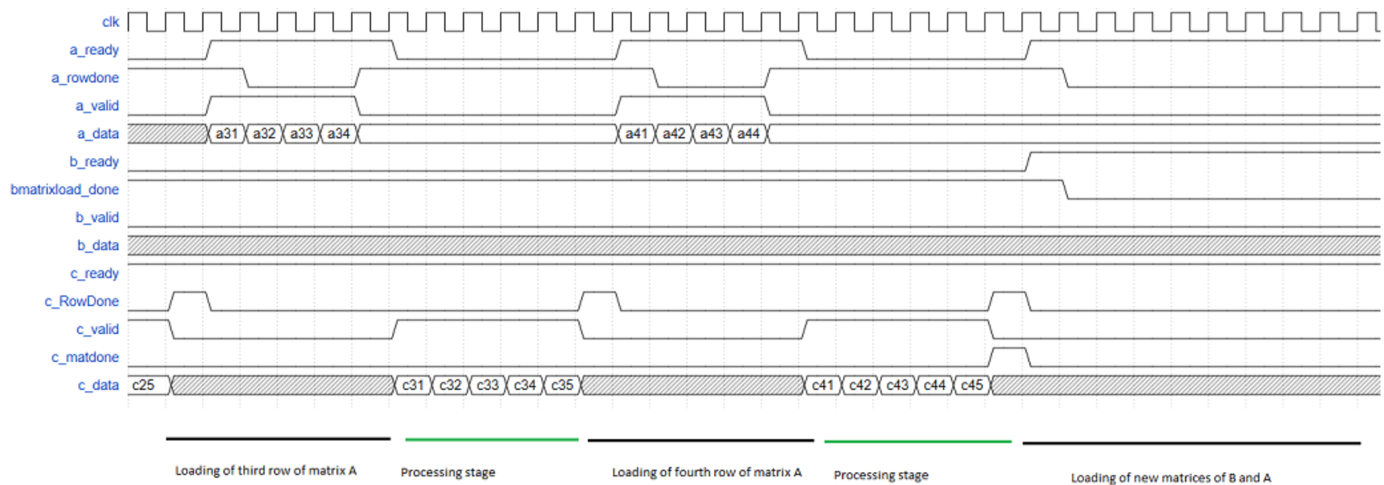


Matrix Multiply Timing diagrams

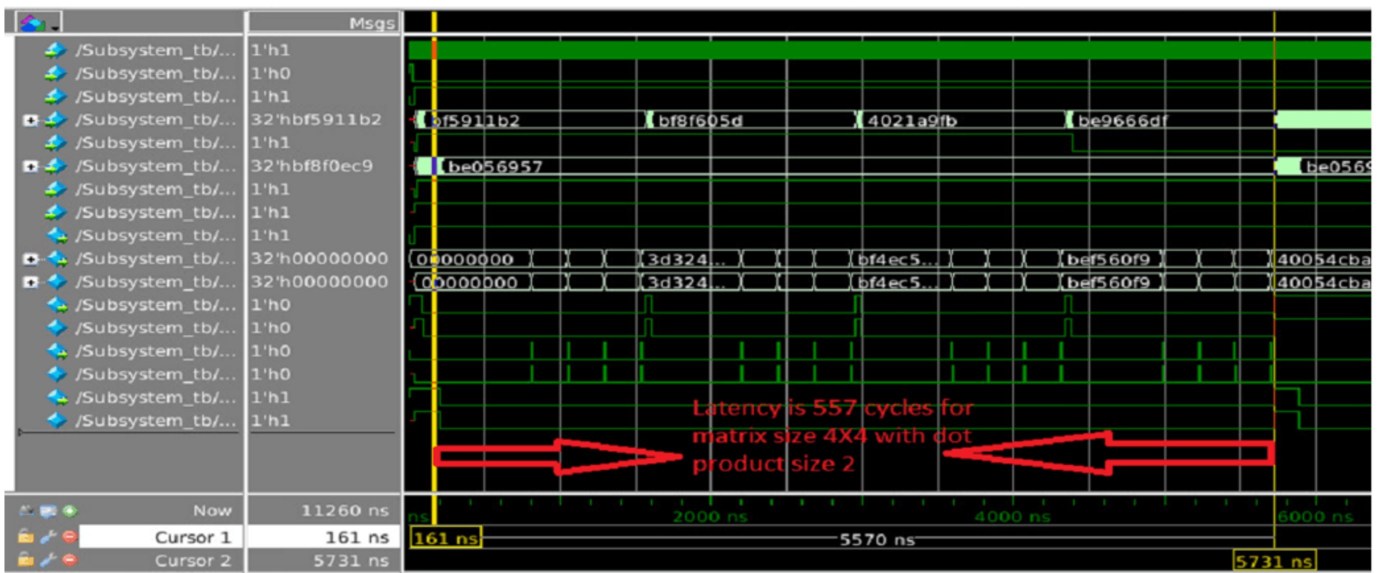
Timing diagram



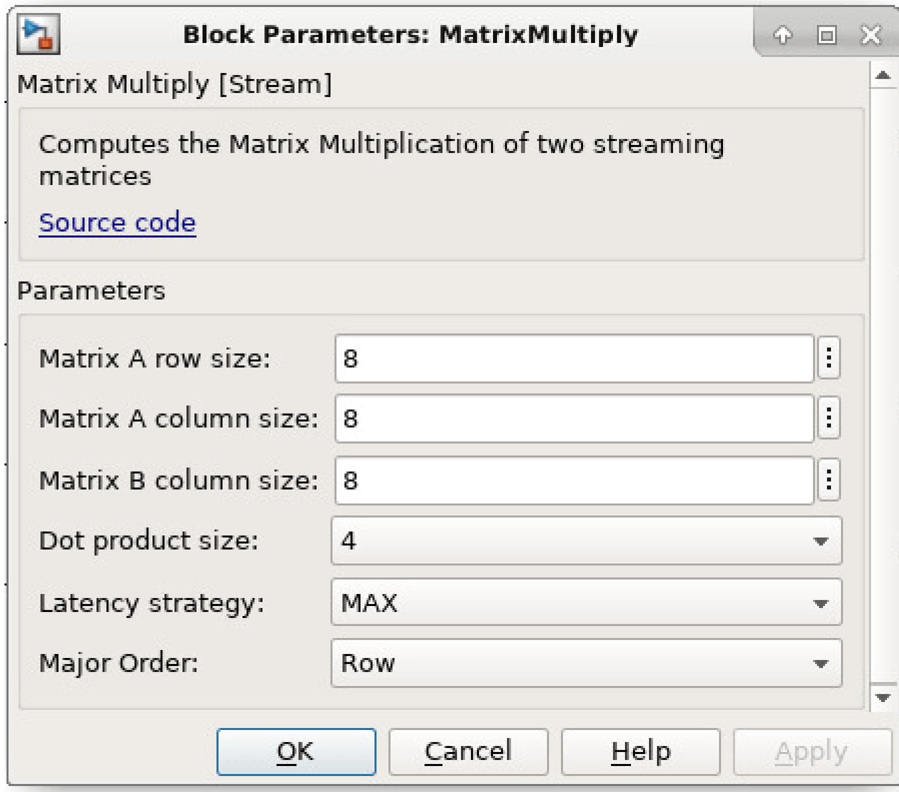
Timing diagram



Modelsim results for streaming matrix multiply



Matrix Multiply Block parameters:




```
Matrix-A Row Size          : Enter row size of input matrix A as a positive integer.
Matrix-A Column Size       : Enter column size of input matrix B as a positive integer
                             which is equals to input matrix B row size.
Matrix-B Column Size       : Enter column size of input matrix B as a positive integer.
Dot product size           : Select dot product size from drop
                             down menu(1,2,4,8,16,32,64) which should be
                             less than input matrix A column size.
LatencyStrategy           :  Select latency strategy from drop
                             down menu ({'ZERO', 'MIN', 'MAX'}) which
                             should be same as HDL coder latency strategy.
                             Processing latency depends on the latency strategy.
Major Order                : Select row major or column major
                             based on the input data streaming.
```

Matrix Multiply Block usage

1. Set block parameters of MATLAB System block.
2. Select input matrix sizes based on the values set in block parameters.
3. Generate HDL code for MatrixMultiply subsystem.

Generated code and Generated model

After running code generation for MatrixMultiply subsystem, generated code will be

```
>> makehdl(gcb, 'nat', 'on')
### Generating HDL for 'hdlcoder_streaming_matrix_multiply_max_latency/MatrixMultiply'.
### Using the config set for model hdlcoder_streaming_matrix_multiply_max_latency for HDL code generation parameters.
### Starting HDL check.
### One or more feedback loops in the model are inhibiting optimizations. To highlight these loops in your model, click the following MATLAB script: hdlsrc/hdlcoder_strea
### To clear highlighting, click the following MATLAB script: hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/clearhighlighting.m
### Begin VHDL Code Generation for 'hdlcoder_streaming_matrix_multiply_max_latency'.
### Working on matrixAStoreControl as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixAStoreControl.vhd.
### Working on matrixBStoreControl as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixBStoreControl.vhd.
### Working on matrixAMemoryReadAddress as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixAMemoryReadAddress.vhd.
### Working on matrixBMemoryReadAddress as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixBMemoryReadAddress.vhd.
### Working on readAddressValid as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/readAddressValid.vhd.
### Working on memoryReadAddressControl as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/memoryReadAddressControl.vhd.
### Working on matrixASubColumnControl as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixASubColumnControl.vhd.
### Working on matrixAMemoryWriteEnableDecoder as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixAMemoryWriteEnableDecoder.vhd.
### Working on matrixAMemory/ SimpleDualPortRAM_generic as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/alphaSimpleDualPortRAM_generic.vhd.
### Working on matrixAMemory as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixAMemory.vhd.
### Working on matrixAMemoryController as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixAMemoryController.vhd.
### Working on matrixBMemoryWriteControl as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixBMemoryWriteControl.vhd.
### Working on matrixBMemoryWriteEnableDecoder as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixBMemoryWriteEnableDecoder.vhd.
### Working on matrixBMemory as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixBMemory.vhd.
### Working on matrixBMemoryController as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixBMemoryController.vhd.
### Working on matrixBMemoryReadDataDecoder as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixBMemoryReadDataDecoder.vhd.
### Working on memoryController as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/memoryController.vhd.
### Working on dotProduct/nfp_mul_single as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/nfp_mul_single.vhd.
### Working on dotProduct/nfp_add_single as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/nfp_add_single.vhd.
### Working on dotProduct as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/dotProduct.vhd.
### Working on accumulator as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/accumulator.vhd.
### Working on processingSystem as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/processingSystem.vhd.
### Working on matrixMultiplyOutputControl as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixMultiplyOutputControl.vhd.
### Working on matrixMultiplyStream as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/matrixMultiplyStream.vhd.
### Working on hdlcoder_streaming_matrix_multiply_max_latency/MatrixMultiply as hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/MatrixMultiply.vhd.
### Generating package file hdlsrc/hdlcoder_streaming_matrix_multiply_max_latency/MatrixMultiply_pkg.vhd.
### Creating HDL Code Generation Check Report MatrixMultiply_report.html
### HDL check for 'hdlcoder_streaming_matrix_multiply_max_latency' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

Generated model contains the MatrixMultiply MATLAB System block. During modelsim simulation code generation outputs are compared with MATLAB System block outputs.

Synthesis statistics

Altera Stratix V 5SEE9F45C2:

Matrix size	Dot Product size	Frequency (MHz)	Latency	Utilization		
				ALMs	LABs	DSP slices
A (4,4),B(4,4)	2	500	557	1016	213	2
A (4,4),B(4,4)	4	426.62	200	1597	299	4
A (8,8),B(8,8)	2	500	3481	1061	209	2
A (8,8),B(8,8)	4	500	2057	1934	353	4
A (8,8),B(8,8)	8	465.77	585	3626	669	8

Xilinx Virtex 7 xc7v2000t:

Matrix size	Dot Product Size	Frequency (MHz)	Latency	Utilization		
				LUTS	Slices	DSP slices
A (4,4),B(4,4)	2	376.08	557	1960	745	4
A (4,4),B(4,4)	4	409.17	200	2788	1106	8
A (8,8),B(8,8)	2	343.29	3481	2073	818	4
A (8,8),B(8,8)	4	389.56	2057	3798	1548	8
A (8,8),B(8,8)	8	412.88	585	6124	2792	16

Restrictions

- Matrix dot product sizes can be `1` or a power of `2`. The allowed sizes are `1`, `2`, `4`, `8`, `16`, `32` and `64`.
- Input data types of the matrices must be `single` and block must be used in the `Native Floating Point` mode.
- Input matrices must not be larger than 64-by-64 in size.

Related Links

`mtimes`

HDL Code Generation from hdl.RAM System Object

This example shows how to generate HDL code from MATLAB® code from hdl.RAM System object™ in MATLAB and infer RAM in generated hardware.

MATLAB Design

This example shows implementation of a line delay that uses a memory in a ring structure, where data is written in one position and read from another position in such a way that the data written will be read after a specific number of cycles. An efficient implementation of this architecture on Virtex FPGAs uses the on-chip Dual Port Block RAMs and an address counter. The Block RAMs can be configured as 512x8 or 256x9 synchronous Dual Port RAMs. To parameterize the delay length, the RAM write address is generated by a counter and the read address is generated by adding a constant K to the write address. If the memory size is M, the input will be read M-K clock cycles after it was written to the memory, hence implementing M-K word shift behaviour.

```
design_name = 'mlhdlc_hdlram';
testbench_name = 'mlhdlc_hdlram_tb';
```

Look at the MATLAB design.

```
type(design_name);
```

```
%#codegen
function data_out = mlhdlc_hdlram(data_in)
%
% This example shows implementation of a line delay that uses a memory in a
% ring structure, where data is written in one position and read from
% another position in such a way that the data written will be read after a
% specific number of cycles. An efficient implementation of this
% architecture on Virtex FPGAs uses the on-chip Dual Port Block RAMs and an
% address counter. The Block RAMs can be configured as 512x8 or 256x9
% synchronous Dual Port RAMs. To parameterize the delay length, the RAM
% write address is generated by a counter and the read address is generated
% by adding a constant K to the write address. If the memory size is M, the
% input will be read M-K clock cycles after it was written to the memory,
% hence implementing M-K word shift behaviour.

%   Copyright 2012-2015 The MathWorks, Inc.

persistent hRam;
if isempty(hRam)
    hRam = hdl.RAM('RAMType', 'Dual port');
end

% read address counter
persistent rdAddrCtr;
if isempty(rdAddrCtr)
    rdAddrCtr = 0;
end

% ring counter length
ringCtrLength = 10;
ramWriteAddr = rdAddrCtr + ringCtrLength;
```

```
ramWriteData = data_in;
ramWriteEnable = true;

ramReadAddr = rdAddrCtr;

% execute single step of RAM
[~, ramRdDout] = step(hRam, ramWriteData, ramWriteAddr, ramWriteEnable, ramReadAddr);

rdAddrCtr = rdAddrCtr + 1;

data_out = ramRdDout;
```

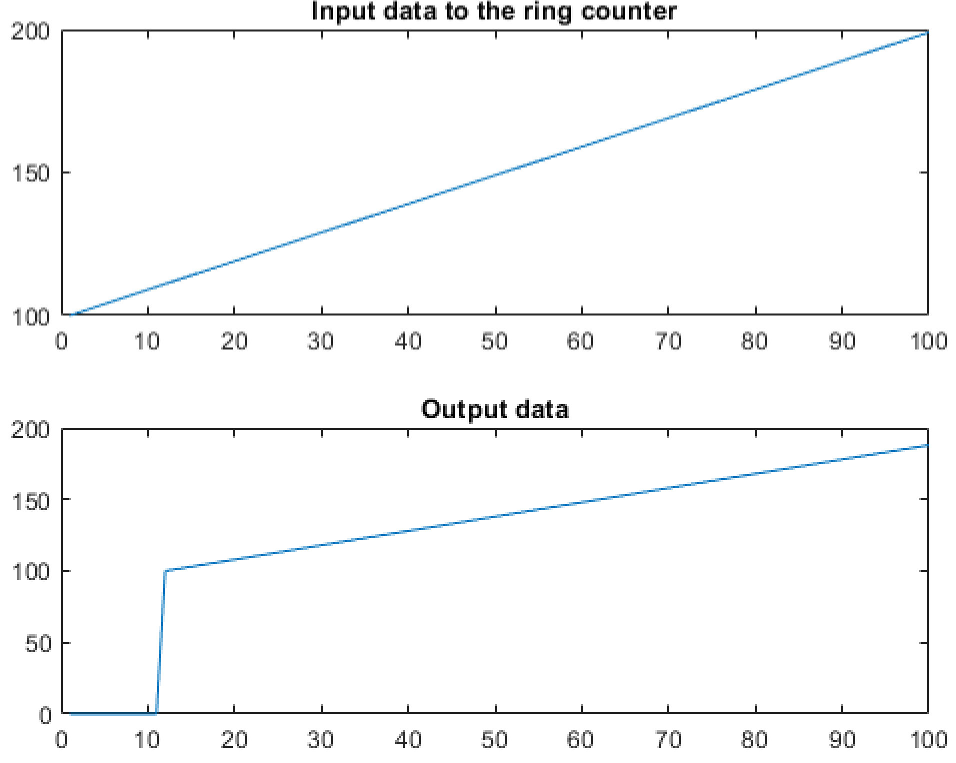
```
type(testbench_name);
```

```
function mlhdlc_hdlram_tb
%

%   Copyright 2012-2015 The MathWorks, Inc.

clear test_hdlram;

data = 100:200;
ring_out = zeros(1, length(data));

for ii=1:100
    ring_in = data(ii);
    ring_out(ii) = mlhdlc_hdlram(ring_in);
end


figure('Name', [mfilename, '_plot']);
subplot(2,1,1);
plot(1:100,data(1:100));
title('Input data to the ring counter')

subplot(2,1,2);
plot(1:100,ring_out(1:100));
title('Output data')

end
```

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_hdlram_tb
```



Create a New HDL Coder™ Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sysobj_prj
```

Next, add the file `mlhdlc_hdlram.m` to the project as the MATLAB Function and `mlhdlc_hdlram_tb.m` as the MATLAB Test Bench.

For a more complete tutorial on creating and populating MATLAB HDL Coder projects, see “Get Started with MATLAB to HDL Workflow”.

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

Supported System objects

For a list of System objects supported for HDL code generation, see “Predefined System Objects Supported for HDL Code Generation” on page 1-50.

HDL Code Generation from a Non-Restoring Square Root System Object

This example shows how to check, generate and verify HDL code from MATLAB® code that instantiates a non-restoring square root System object™.

MATLAB Design

The MATLAB code used in this example is a non-restoring square root engine suitable for implementation in an FPGA or ASIC. The engine uses a multiplier-free minimal area implementation based on [1] decision convolutional decoding, implemented as a System object. This example also shows a MATLAB test bench that tests the engine.

```
design_name = 'mlhdlc_sysobj_nonrestsqrt.m';
testbench_name = 'mlhdlc_sysobj_nonrestsqrt_tb.m';
sysobj_name = 'mlhdlc_msysobj_nonrestsqrt.m';
```

Look at the MATLAB design.

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Non-restoring Square Root
%
% Key Design pattern covered in this example:
% (1) Using a user-defined system object
% (2) Call the object only once per design iteration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [Q_o,Vld_o] = mlhdlc_sysobj_nonrestsqrt(D_i, Init_i)

%   Copyright 2014-2015 The MathWorks, Inc.

    persistent hSqrt;

    if isempty(hSqrt)
            hSqrt = mlhdlc_msysobj_nonrestsqrt();
    end

    [Q_o,Vld_o] = hSqrt(D_i,Init_i);
end
```

```
type(testbench_name);
```

```
% Nonrestoring Squareroot Testbench

%   Copyright 2014-2015 The MathWorks, Inc.

% Generate some random data
rng('default'); % set the random number generator to a consistent state
nsamp = 100; %number of samples
nbits = 32; % fixed-point word length
nfrac = 31; % fixed-point fraction length

data_i = fi(rand(1,nsamp), numerictype(0,nbits,nfrac));
```

```
% clear any persistent variables in the HDL function
clear mlhdlc_sysobj_nonrestsqrt

% Determine the "golden" sqrt results
data_go = sqrt(data_i);

% Commands for the sqrt engine
LOAD_DATA = true;
CALC_DATA = false;

% Pre-allocate the result array
data_o = zeros(1,nsamp, 'like', data_go);
% Load in a sample, then iterate until the results are ready
cyc_cnt = 0;
for i = 1:nsamp
    % Load the new sample into the sqrt engine
    [~, vld] = mlhdlc_sysobj_nonrestsqrt(data_i(i),LOAD_DATA);
    cyc_cnt = cyc_cnt + 1;
    while(vld == false)
        % Iterate until the result has been found
        [data_o(i), vld] = mlhdlc_sysobj_nonrestsqrt(data_i(i),CALC_DATA);
        cyc_cnt = cyc_cnt + 1;
    end
end

% find the integer representation of the result data
idt = numerictype(0,ceil(nbits/2),0);
% find the error in terms of integer bits
ierr = abs(double(reinterpretcast(data_o,idt))-double(reinterpretcast(data_go,idt)));
% find the error in terms of real-world values
derr = abs(double(data_o)- double(data_go));
pct_err = 100*derr ./ double(data_go);

fprintf('Maximum Error: %d (%0.3f %%)\n', max(derr), max(pct_err));
fprintf('Maximum Error (as unsigned integer): %d\n', max(ierr));
fprintf('Number of cycles: %d ( %d per sample)\n', cyc_cnt, cyc_cnt / nsamp);

%EOF
```

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

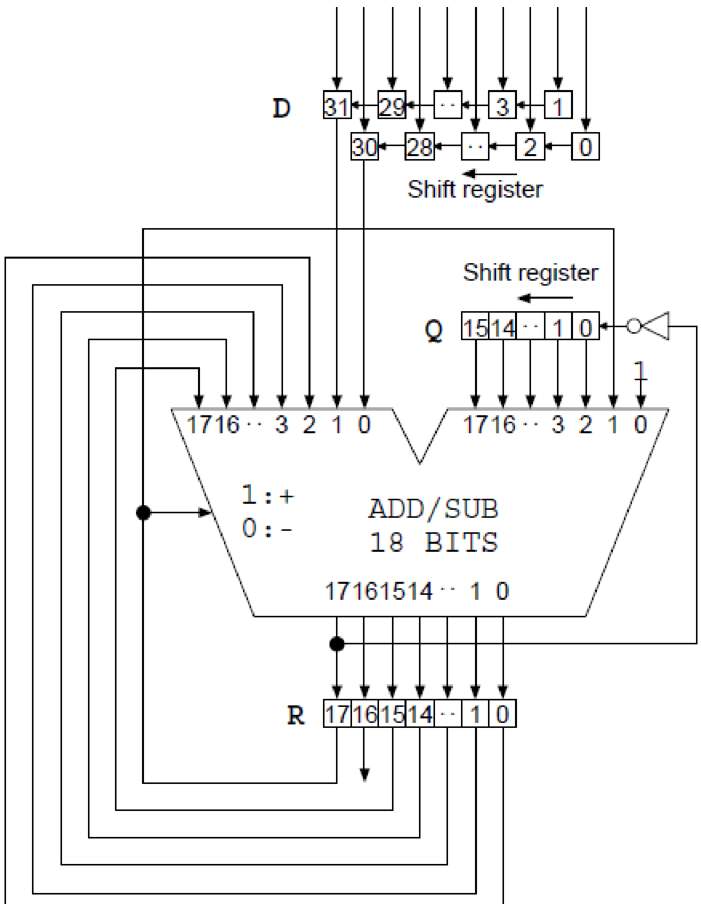
```
mlhdlc_sysobj_nonrestsqrt_tb
```

```
Maximum Error: 3.051758e-05 (0.028 %)
Maximum Error (as unsigned integer): 1
Number of cycles: 2000 ( 20 per sample)
```

Hardware Implementation of the Non-Restoring Square Root Algorithm

This algorithm implements the square root operation in a minimal area by using a single adder/subtractor with no mux (compared to a restoring algorithm that requires a mux). The square root is calculated using a series of shifts and adds/subs, so uses no multipliers (compared to other implementations which require a multiplier).

The overall architecture of the algorithm is shown below, as described in [1].



This implementation of the algorithm uses a minimal area approach that requires multiple cycles to calculate each result. The overall calculation time can be approximated as [Input Word Length / 2], with a few cycles of overhead to load the incoming data.

Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_nonrestsqrt
```

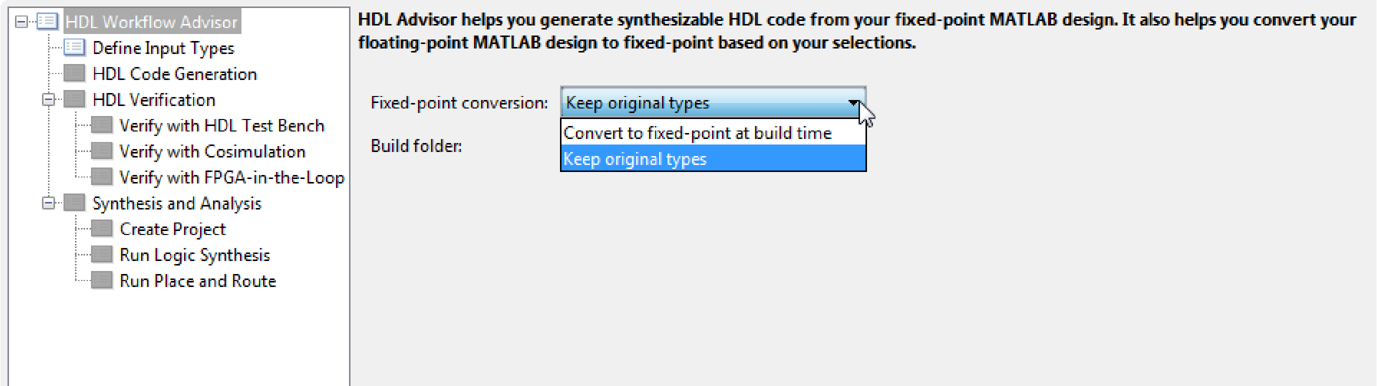
Next, add the file `mlhdlc_sysobj_nonrestsqrt.m` to the project as the MATLAB function and `mlhdlc_sysobj_nonrestsqrt_tb.m` as the MATLAB test bench.

For a more complete tutorial on creating and populating MATLAB HDL Coder projects, see “Get Started with MATLAB to HDL Workflow”.

Skip Fixed-Point Conversion

As the design is already in fixed-point, we do not need to perform automatic conversion.

Launch the HDL Advisor and choose `Keep original types` on the option **Fixed-point conversion**.



Run HDL Code Generation

Launch the Workflow Advisor. In the Workflow Advisor, right-click the **Code Generation** step and choose the option **Run to selected task** to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

Supported System objects

For a list of System objects supported for HDL code generation, see “Predefined System Objects Supported for HDL Code Generation” on page 1-50.

References

[1] Li, Y. and Chu, W. (1996) "A New Non-Restoring Square Root Algorithm and Its VLSI Implementations". IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD '96 Austin, Texas USA (7-9 October, 1996), pp. 538-544. doi: 10.1109/ICCD.1996.563604

HDL Code Generation from Viterbi Decoder System Object

This example shows how to check, generate and verify HDL code from MATLAB® code that instantiates a Viterbi Decoder System object™.

MATLAB Design

The MATLAB code used in this example is a Viterbi Decoder used in hard decision convolutional decoding, implemented as a System object. This example also shows a MATLAB test bench that tests the decoder.

```
design_name = 'mlhdlc_sysobj_viterbi';
testbench_name = 'mlhdlc_sysobj_viterbi_tb';
```

Look at the MATLAB design.

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Viterbi Decoder
%
% Key Design pattern covered in this example:
% (1) Using comm system toolbox ViterbiDecoder object
% (2) The object can be called only once per design iteration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2015 The MathWorks, Inc.

function decodedBits = mlhdlc_sysobj_viterbi(inputSymbol)

persistent hVitDec;

if isempty(hVitDec)
    hVitDec = comm.ViterbiDecoder('InputFormat','Hard', 'OutputDataType', 'Logical');
end

decodedBits = hVitDec(inputSymbol);
```

```
type(testbench_name);
```

```
% Viterbi_tb - testbench for Viterbi_dut

%   Copyright 2011-2015 The MathWorks, Inc.

numErrors = 0;
% rand stream
original_rs = RandStream.getGlobalStream;
rs = RandStream.create('mrg32k3a', 'seed', 25);
%RandStream.getGlobalStream(rs);
rs.reset;
% convolutional encoder
hConvEnc = comm.ConvolutionalEncoder;
% BER
hBER = comm.ErrorRate;
hBER.ReceiveDelay = 34;
```

```
reset(hBER);

% clear persistent variables in the design between runs of the testbench
clear mlhdlc_msysobj_viterbi;

for numSymbols = 1:10000
    % generate a random bit
    inputBit = logical(randi([0 1], 1, 1));

    % encode it with the Convolutional Encoder - rate 1/2
    encodedSymbol = hConvEnc(inputBit);

    % optional - add noise

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % call Viterbi Decoder DUT to decode the symbol
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    vitdecOut = mlhdlc_sysobj_viterbi(encodedSymbol);

    ber = hBER(inputBit, vitdecOut);
end

fprintf('%s\n', repmat('%', 1, 38));
fprintf('%%%%%%%%%%%%%% %s %%%%%%%%%%%%%%\n', 'Viterbi Decoder Output');
fprintf('%s\n', repmat('%', 1, 38));
fprintf('Number of bits %d, BER %g\n', numSymbols, ber(1));
fprintf('%s\n', repmat('%', 1, 38));

% EOF
```

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sysobj_viterbi_tb
```

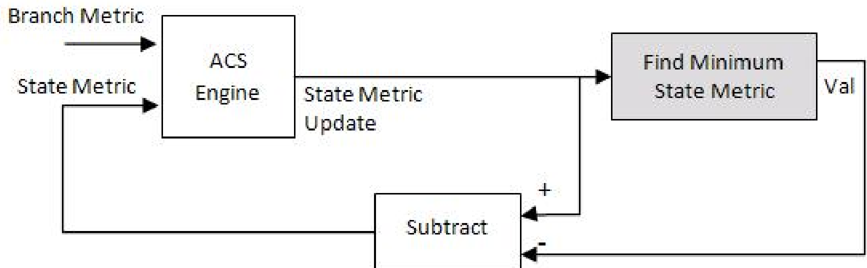
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% Viterbi Decoder Output %%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Number of bits 10000, BER 0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Hardware Implementation of Viterbi Decoding Algorithm

There are three main components in the Viterbi decoding algorithm. They are the branch metric computation (BMC), add-compare-select (ACS), and traceback decoding. The following diagram illustrates the three units in the Viterbi decoding algorithm.

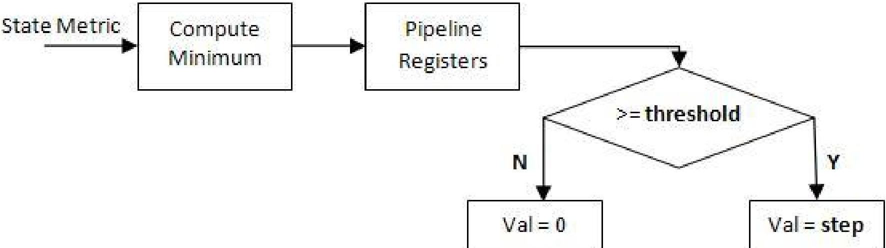
The Renormalization Method

The Viterbi decoder prevents the overflow of the state metrics in the ACS component by subtracting the minimum value of the state metrics at each time step, as shown in the following figure.



Obtaining the minimum value of all the state metric elements in one clock cycle results in a poor clock frequency for the circuit. The performance of the circuit may be improved by adding pipeline registers. However, simply subtracting the minimum value delayed by pipeline registers from the state metrics may still lead to overflow.

The hardware architecture modifies the renormalization method and avoids the state metric overflow in three steps. First, the architecture calculates values for the threshold and step parameters, based on the trellis structure and the number of soft decision bits. Second, the delayed minimum value is compared to the threshold. Last, if the minimum value is greater than or equal to the threshold value, the implementation subtracts the step value from the state metric; otherwise no adjustment is performed. The following figure illustrates the modified renormalization method.



Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_viterbi
```

Next, add the file `mlhdlc_sysobj_viterbi.m` to the project as the MATLAB function and `mlhdlc_sysobj_viterbi_tb.m` as the MATLAB test bench.

For a more complete tutorial on creating and populating MATLAB HDL Coder projects, see “Get Started with MATLAB to HDL Workflow”.

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor. In the Workflow Advisor, right-click the **Code Generation** step and choose the option **Run to selected task** to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

Supported System objects

For a list of System objects supported for HDL code generation, see “Predefined System Objects Supported for HDL Code Generation” on page 1-50.

Predefined System Objects Supported for HDL Code Generation

In this section...

"Predefined System Objects in MATLAB Code" on page 1-50
"Predefined System Objects in the MATLAB System Block" on page 1-51

Predefined System Objects in MATLAB Code

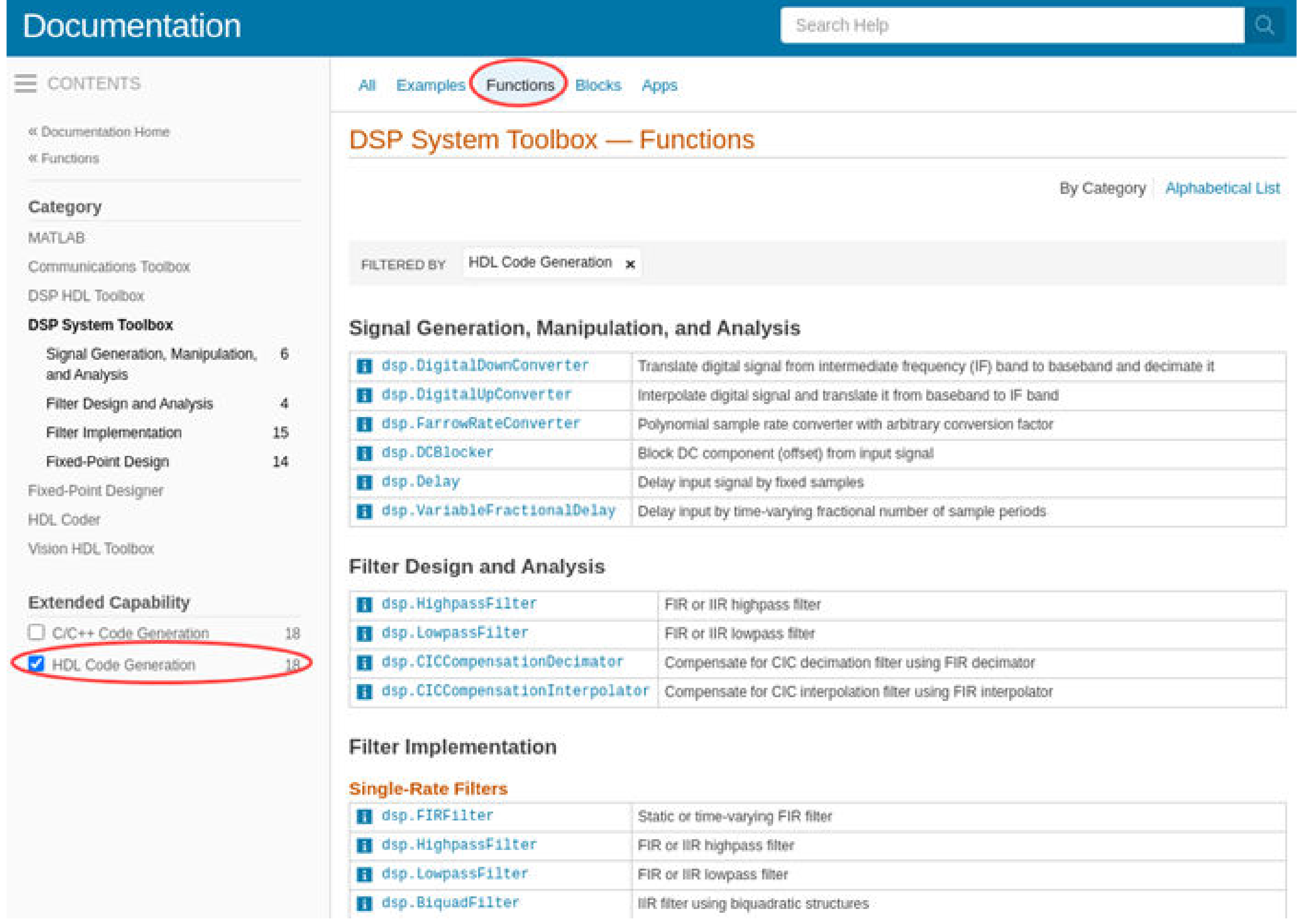
HDL Coder supports the following MATLAB System objects for HDL code generation:

- `hdl.RAM`
- `hdl.BlackBox`
- `hdl.Delay`

HDL Coder also supports HDL code generation from certain System objects in DSP System Toolbox and Communications Toolbox, and from HDL-optimized algorithms implemented with System objects in DSP HDL Toolbox™ and Vision HDL Toolbox™.

You can view System objects that are supported for HDL code generation in documentation by filtering the functions reference list. Click **Functions** in the blue bar at the top of the Help window, then select the **HDL code generation** check box at the bottom of the left column. The System objects are listed in their respective products. You can use the table of contents in the left column to navigate between products and categories.

Refer to the "Extended Capabilities > HDL Code Generation" section of each block page for restrictions for HDL code generation.



Predefined System Objects in the MATLAB System Block

A subset of these predefined System objects are supported for code generation when you use them in a MATLAB System block. To learn more, see “HDL Code Generation” on the MATLAB System page.

Load constants from a MAT-File

You can load compile-time constants from a MAT-file with the `coder.load` function in your MATLAB design.

For example, you can create a MAT-file, `sinvals.mat`, that contains fixed-point values of `sin` by entering the following commands in MATLAB:

```
sinvals = sin(fi(-pi:0.1:pi, 1, 16,15));
save sinvals.mat sinvals;
```

You can then generate HDL code from the following MATLAB code, which loads the constants from `sinvals.mat` into a persistent variable, `pConstStruct`, and assigns the values to a variable that is not persistent, `sv`.

```
persistent pConstStruct;
if isempty(pConstStruct)
    pConstStruct = coder.load('sinvals.mat');
end
sv = pConstStruct.sinvals;
```

See Also

`codegen` | `coder.HdlConfig`

More About

- "Functions Supported for HDL and HLS Code Generation" on page 1-2
- "Complex Data Type Support" on page 1-11
- "Supported MATLAB Data Types, Operators, and Control Flow Statements" on page 1-4

Generate Code for User-Defined System Objects

In this section...
“How To Create A User-Defined System object” on page 1-53
“User-Defined System object Example” on page 1-53

How To Create A User-Defined System object

To create a user-defined System object and generate code:

1. Create a class that subclasses from `matlab.System`.
2. Define one of the following sets of methods:
   - `setupImpl` and `stepImpl`
   - `setupImpl`, `outputImpl`, and `updateImpl`
3. Optionally, if your System object has private state properties, define the `resetImpl` method to initialize them to zero.
4. Write a top-level design function that creates an instance of your System object and calls the object, or calls the `output` and `update` methods.

   Note The `resetImpl` method runs automatically during System object initialization. For HDL code generation, you cannot call the public `reset` method.
5. Write a test bench function that exercises the top-level design function.
6. Generate HDL code.

User-Defined System object Example

This example shows how to generate HDL code for a user-defined System object that implements the `setupImpl` and `stepImpl` methods.

1. In a writable folder, create a System object, `CounterSysObj`, which subclasses from `matlab.System`. Save the code as `CounterSysObj.m`.

```
classdef CounterSysObj < matlab.System

    properties (Nontunable)
        Threshold = int32(1)
    end
    properties (Access=private)
        State
        Count
    end
    methods
        function obj = CounterSysObj(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end

    methods (Access=protected)
        function setupImpl(obj, ~)
```

```
            % Initialize states
            obj.Count = int32(0);
            obj.State = int32(0);
        end
        function y = stepImpl(obj, u)
            if obj.Threshold > u(1)
                obj.Count(:) = obj.Count + int32(1); % Increment count
            end
            y = obj.State;          % Delay output
            obj.State = obj.Count;  % Put new value in state
        end
    end
end
```

The `stepImpl` method implements the System object functionality. The `setupImpl` method defines the initial values for the persistent variables in the System object.

2 Write a function that uses this System object and save it as `myDesign.m`. This function is your DUT.

```
function y = myDesign(u)

persistent obj
if isempty(obj)
    obj = CounterSysObj('Threshold',5);
end

y = obj(u);

end
```

3 Write a test bench that calls the DUT function and save it as `myDesign_tb.m`.

```
clear myDesign
for ii=1:10
    y = myDesign(int32(ii));
end
```

4 Generate HDL code for the DUT function as you would for any other MATLAB code, but skip fixed-point conversion.

See Also

More About

- “HDL Code Generation for System Objects” on page 1-16

Map Matrices to ROM

To map a matrix constant to ROM:

- Read one matrix element at a time.
- The matrix size must be greater than or equal to the value specified for **RAM mapping threshold**.

  To learn how to set the RAM mapping threshold in MATLAB, see “Enable RAM Mapping” on page 8-6.
- Read accesses to the matrix must not be within a feedback loop.

If your MATLAB code meets these requirements, HDL Coder inserts a no-reset register at the output of the matrix in the generated code. Many synthesis tools infer a ROM from this code pattern.

Model State with Persistent Variables and System Objects

This example shows how to use persistent variables and System objects to model state and delays in a MATLAB® design for HDL code generation.

Introduction

Using System objects to model delay results in concise generated code.

In MATLAB, multiple calls to a function having persistent variables do not result in multiple delays. Instead, the state in the function gets updated multiple times.

In order to reuse code implemented in a function with states, you need to duplicate functions multiple times to create multiple instances of the algorithm with delay.

Examine the MATLAB Code

Look at the implementation of the Sobel algorithm.

Examine the design to see how the delays and line buffers are modeled using:

- Persistent variables: mlhdlc_sobel
- System objects: mlhdlc_sysobj_sobel

Notice that the `filterdelay` function is duplicated with different function names in `mlhdlc_sobel` code to instantiate multiple versions of the algorithm in MATLAB for HDL code generation.

The delay line implementation is more complicated when done using MATLAB persistent variables.

Now examine the simplified implementation of the same algorithm using System objects in `mlhdlc_sysobj_sobel`.

When used within the constraints of HDL code generation, the dsp.Delay objects always map to registers. For persistent variables to be inferred as registers, you have to be careful to read the variable before writing to it to map it to a register.

MATLAB Design

```
demo_files = {...
    'mlhdlc_sysobj_sobel', ...
    'mlhdlc_sysobj_sobel_tb', ...
    'mlhdlc_sobel', ...
    'mlhdlc_sobel_tb'
    };
```

Known Limitations

For predefined System Objects, HDL Coder™ only supports the `step` method and does not support `output` and `update` methods.

With support for only the step method, delays cannot be used in modeling feedback paths. For example, the following piece of MATLAB code cannot be supported using the `dsp.Delay` System object.

```
%#codegen
```

```
function y = accumulate(u)
persistent p;
if isempty(p)
   p = 0;
end
y = p;
p = p + u;
```

Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sobel
```

Next, add the file `mlhdlc_sobel.m` to the project as the MATLAB Function and `mlhdlc_sobel_tb.m` as the MATLAB Test Bench.

For a more complete tutorial on creating and populating MATLAB HDL Coder projects, see “Get Started with MATLAB to HDL Workflow”.

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor and right-click the **Code Generation** step. Choose the option **Run to selected task** to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the hyperlinks in the Code Generation Log window.

Now, create a new project for the system object design:

```
coder -hdlcoder -new mlhdlc_sysobj_sobel
```

Add the file `mlhdlc_sysobj_sobel.m` to the project as the MATLAB Function and `mlhdlc_sysobj_sobel_tb.m` as the MATLAB Test Bench.

Repeat the code generation steps and examine the generated fixed-point MATLAB and HDL code.

Additional Notes:

You can model integer delay using `dsp.Delay` object by setting the `Length` property to be greater than 1. These delay objects will be mapped to shift registers in the generated code.

If the optimization option **Map persistent array variables to RAMs** is enabled, delay System objects will get mapped to block RAMs under the following conditions:

- `InitialConditions` property of the `dsp.Delay` is set to zero.
- Delay input data type is not floating-point.
- `RAMSize` (DelayLength * InputWordLength) is greater than or equal to the **RAM Mapping Threshold**.

Bitwise Operations in MATLAB for HDL and HLS Code Generation

HDL Coder supports bit shift, bit rotate, bit slice operations that mimic HDL-specific operators without saturation and rounding logic.

Bit Shifting and Rotation

The following code implements a barrel shifter/rotator that performs a selected operation (based on the `mode` argument) on a fixed-point input operand.

```
function y   = fcn(u, mode)
% Multi Function Barrel Shifter/Rotator

% fixed width shift operation
fixed_width = uint8(3);

switch mode
    case 1
        % shift left logical
        y = bitsll(u, fixed_width);
    case 2
        % shift right logical
        y = bitsrl(u, fixed_width);
    case 3
        % shift right arithmetic
        y = bitsra(u, fixed_width);
    case 4
        % rotate left
        y = bitrol(u, fixed_width);
    case 5
        % rotate right
        y = bitror(u, fixed_width);
    otherwise
        % do nothing
        y = u;
end
```

This table shows the generated VHDL, Verilog and, HLS code.

Generated VHDL code	Generated Verilog code	Generated HLS code
In VHDL code generated for this function, the shift and rotate functions map directly to shift and rotate instructions in VHDL.	The corresponding Verilog code is similar, except that Verilog does not have native operators for rotate instructions.	The generated HLS code uses the native C++ bitwise operators to accomplish the shift operations. The rotate operations are written using shift operators.

Generated VHDL code:

```
     CASE mode IS
            WHEN "00000001" =>
                -- shift left logical
                --'<S2>:1:8'
                cr := signed(u) sll 3;
                y <= std_logic_vector(cr);
            WHEN "00000010" =>
                -- shift right logical
                --'<S2>:1:11'
                b_cr := signed(u) srl 3;
                y <= std_logic_vector(b_cr);
            WHEN "00000011" =>
                -- shift right arithmetic
                --'<S2>:1:14'
                c_cr := SHIFT_RIGHT(signed(u) , 3);
                y <= std_logic_vector(c_cr);
            WHEN "00000100" =>
                -- rotate left
                --'<S2>:1:17'
                d_cr := signed(u) rol 3;
                y <= std_logic_vector(d_cr);
            WHEN "00000101" =>
                -- rotate right
                --'<S2>:1:20'
                e_cr := signed(u) ror 3;
                y <= std_logic_vector(e_cr);
            WHEN OTHERS =>
                -- do nothing
                --'<S2>:1:23'
                y <= u;
        END CASE;
```

Generated Verilog code:

```
          case ( mode)
            1 :
              begin
                // shift left logical
                //'<S2>:1:8'
                cr = u <<< 3;
                y = cr;
              end
            2 :
              begin
                // shift right logical
                //'<S2>:1:11'
                b_cr = u >> 3;
                y = b_cr;
              end
            3 :
              begin
                // shift right arithmetic
                //'<S2>:1:14'
                c_cr = u >>> 3;
                y = c_cr;
              end
            4 :
              begin
                // rotate left
                //'<S2>:1:17'
                d_cr = {u[12:0], u[15:13]};
                y = d_cr;
              end
            5 :
              begin
                // rotate right
                //'<S2>:1:20'
                e_cr = {u[2:0], u[15:3]};
                y = e_cr;
              end
            default :
              begin
                // do nothing
                //'<S2>:1:23'
                y = u;
              end
        endcase
```

Generated HLS code:

```
#pragma once

#include "rtwtypes.hpp"

class fcnClass
{
 public:
  sc_ufixed<16,6> f(sc_ufixed<16,6> u, int8_T mod
  {
    sc_ufixed<16,6> y;
    sc_ufixed<16,6> c;
    sc_ufixed<16,6> c_0;
    sc_ufixed<16,6> c_1;
    sc_ufixed<16,6> c_2;
    c = sc_ufixed<16,6>(0.0);
    c_0 = sc_ufixed<16,6>(0.0);
    c_1 = sc_ufixed<16,6>(0.0);
    c_2 = sc_ufixed<16,6>(0.0);

    /*  Multi Function Barrel Shifter/Rotator */
    /*  fixed width shift operation */
    switch (mode) {
     case 1:
      /*  shift left logical */
      y = u << 3;
      break;

     case 2:
      /*  shift right logical */
      c_1 = u >> 3;
      y = c_1;
      break;

     case 3:
      /*  shift right arithmetic */
      c_2 = u >> 3;
      y = c_2;
      break;

     case 4:
      /*  rotate left */
      c = (sc_ufixed<16,6>)(u << 3) | (sc_ufixed<
      y = c;
      break;

     case 5:
      /*  rotate right */
      c_0 = (sc_ufixed<16,6>)(u >> 3) | (sc_ufixe
      y = c_0;
      break;

     default:
      /*  do nothing */
      y = u;
      break;
    }

    return y;
  }
};
```

Bit Slicing and Bit Concatenation

The `bitsliceget` and `bitconcat` functions map directly to slice and concatenate operators in both VHDL and Verilog.

You can use the functions `bitsliceget` and `bitconcat` to access and manipulate bit slices (fields) in a fixed-point or integer word. As an example, consider the operation of swapping the upper and lower 4-bit nibbles of an 8-bit byte. The following example accomplishes this task without resorting to traditional mask-and-shift techniques.

```
function y = fcn(u)
% NIBBLE SWAP
y = bitconcat( …
      bitsliceget(u, 4, 1),
      bitsliceget(u, 8, 5));
```

This table shows the generated VHDL, Verilog and, HLS code.

Generated VHDL code	Generated Verilog code	Generated HLS code
The following listing shows the corresponding generated VHDL code.	The following listing shows the corresponding generated Verilog code.	The following listing shows the corresponding generated HLS code.

```
ENTITY fcn IS
    PORT (
        clk : IN std_logic;
        clk_enable : IN std_logic;
        reset : IN std_logic;
        u : IN std_logic_vector(7 DOWNTO 0);
        y : OUT std_logic_vector(7 DOWNTO 0));
END nibble_swap_7b;


ARCHITECTURE fsm_SFHDL OF fcn IS


BEGIN
    -- NIBBLE SWAP
    y <= u(3 DOWNTO 0) & u(7 DOWNTO 4);
END fsm_SFHDL;
```

```
module fcn (clk, clk_enable, reset, u, y);
    input clk;
    input clk_enable;
    input reset;
    input [7:0] u;
    output [7:0] y;

    // NIBBLE SWAP
    assign y = {u[3:0], u[7:4]};

endmodule
```

```
#pragma once

#include "rtwtypes.hpp"

class fcnClass
{
 public:
  sc_uint<8> f(sc_ufixed<16,6> u)
  {
    sc_uint<8> y;
    sc_uint<16> tmp;
    sc_uint<16> tmp_0;

    /*  NIBBLE SWAP  */
    tmp = u();
    tmp_0 = u();
    y = (sc_uint<8>)(sc_uint<4>)((sc_uint<16>)tmp
      (sc_uint<8>)(sc_uint<4>)((sc_uint<16>)(tmp_
    return y;
  }
};
```

See Also

`codegen` | `coder.HdlConfig`

More About

- "Functions for Programming and Data Types"
- "Fixed-Point Function Limitations" on page 1-2
- "Supported MATLAB Data Types, Operators, and Control Flow Statements" on page 1-4

Mapping of Different Rounding and Overflow Methods from MATLAB to HLS

The specified fixed-point data types in MATLAB code are converted to High-Level Synthesis (HLS) fixed-point data types during code generation.

The tables show the mapping of different MATLAB fixed-point rounding and overflow methods to their equivalent HLS methods.

Rounding Methods

MATLAB Fixed-Point Rounding Methods	Description	Equivalent HLS Fixed-Point Rounding Methods
`Nearest` (default for MATLAB)	Rounds to the nearest representable number.	`SC_RND`
`Zero`	Rounds to the nearest representable number in the direction of zero.	`SC_TRN_ZERO`
`Floor`	Rounds to the nearest representable number in the direction of negative infinity. Equivalent to two's complement truncation.	`SC_TRN` (default for HLS)
`Round`	Rounds to the nearest representable number.	`SC_RND_INF`
`Convergent`	Rounds to the nearest representable number.	`SC_RND_CONV`

Note The `ceiling` rounding method is not supported for HLS code generation.

Overflow Methods

MATLAB Fixed-Point Overflow Methods	Description	Equivalent HLS Fixed-Point Overflow Methods
`Saturate` (default for MATLAB)	Saturate to maximum or minimum value of the fixed-point range on overflow.	`SC_SAT`
`Wrap`	Wrap on overflow. This mode is also known as two's complement overflow.	`SC_WRAP` (default for HLS)

The rounding and overflow methods are represented as typecasts on expressions in the generated HLS code.

For example, consider the MATLAB function `exampleFun` and the generated HLS code to understand the conversion of rounding and overflow methods.

MATLAB Code

```
function y = exampleFun(a, b)
    y = fi(a + b, 1, 5, 2, fimath('RoundingMethod','Nearest','OverflowAction','Saturate'));
end
```

Generated HLS Code

```
class exampleFunClass
{
    public:
    sc_fixed<5,3> exampleFun(sc_fixed<7,3> a, sc_fixed<7,3> b)
    {
        return (sc_fixed<5,3,SC_RND,SC_SAT>)((sc_fixed<8,4>)a + (sc_fixed<8,4>)b);
    }
};
```

Handling Constants in HDL and HLS Code Generation

When generating HDL or High-Level Synthesis (HLS) code from MATLAB, handling constants efficiently is important for optimizing performance and resource utilization. You can do this by using the functions `coder.const` and `coder.load`.

Specify Constants in Generated Code

To optimize the code, specify the constants in the generated code by using `coder.const`.

In the table, the MATLAB code consists of two functions, `filter_lowpass` and `calc_lowpass`. The `filter_lowpass` function applies a low-pass filter to an input sample in and outputs a single value representing the current filtered sample of the input signal. The `calc_lowpass` function calculates the filter coefficients for a low-pass filter given a normalized cutoff frequency `fc`.

Using `coder.const` in the `filter_lowpass` function instructs the code generator to treat the coefficients as constants.

MATLAB Code

```
% MATLAB code
function out=filter_lowpass(in)
    persistent delayline;
    if isempty(delayline)
        delayline=zeros(1,21);
    end

    fc = 0.3;

    coefficients=coder.const(calc_lowpass(fc));

    delayline=[in delayline(1:20)];
    out=sum(delayline.*coefficients);
end

function coeffs=calc_lowpass(fc)
    x = (-10:10)*fc;
    % sinc and gausswin are not supported with fi datatype
    % Cast it to single or double.
    coeffs=sinc(single(x)).*gausswin(single(21)).';
    coeffs=coeffs/sum(coeffs);
end
```

MATLAB Test Bench

```
n = 100;
t = 0:n;
signal=sin(2*pi*(t/n)/2.*(t));

filtered=zeros(size(signal));
for i=1:length(t)
    filtered(i)=filter_lowpass(signal(i));
end
```

Load Compile-Time Constants from MAT-file

You can also load compile-time constants from external files into your MATLAB design by using `coder.load`. The function `coder.load` loads data at compile time, not at run time.

In this example, the MATLAB code defines a low-pass filter `filter_lowpass` that uses coefficients loaded from a file `coeffs.mat` to filter an input signal. The MATLAB test bench generates the signals using `calc_lowpass` and then saves them in `coeffs.mat`.

<table>
<tr><th>MATLAB Code</th><th>MATLAB Test Bench</th></tr>
<tr><td>

```
% MATLAB code
function out=filter_lowpass(in)
    persistent delayline coeffs;
    if isempty(delayline)
        % load the mat file for the coeffs
        coeffs_struct = coder.load('coeffs.mat');
        coeffs = coeffs_struct.coeffs;
        delayline=zeros(1,21);
    end

    delayline=[in delayline(1:20)];
    out=sum(delayline.*coeffs);
end

function coeffs=calc_lowpass(fc)
    x = (-10:10)*fc;
    coeffs=sinc(single(x)).*gausswin(single(21)).';
    coeffs=coeffs/sum(coeffs);
end
```

</td><td>

```
n = 100;
t = 0:n;
signal=sin(2*pi*(t/n)/2.*(t));

% create a mat file for the constant
% coefficients from calc_lowpass function
fc = fi(0.3, 0, 14, 15, hdlfimath);
coeffs = calc_lowpass(fc);
save coeffs.mat coeffs;

filtered=zeros(size(signal));
for i=1:length(t)
    filtered(i)=filter_lowpass(signal(i));
end
```

</td></tr>
</table>

Generate HDL or HLS Code

To generate HDL or HLS code for the above defined MATLAB functions use these commands.

```
fname = 'filter_lowpass';

% HLS Codegen
cfg = coder.config('hdl');
% cfg.workflow = "Generic ASIC/FPGA"; % For HDL code generation
cfg.workflow = "High Level Synthesis";
tbname = [fname,'_tb'];
cfg.TestBenchName = tbname;

fixptCfg = coder.config('fixpt');
fixptCfg.TestBenchName = cfg.TestBenchName;

outname = [fname,'_sc_fixpt'];
codegen(fname,'-config',cfg,'-report','-float2fixed',fixptCfg);
```

Structure Definition for HLS Code Generation

To generate efficient standalone High-Level Synthesis (HLS) code for structures, you must define and use structures differently than you normally would when running your code in the MATLAB environment:

What's Different	More Information
Use a restricted set of operations.	"Structure Operations Allowed for Code Generation"
Observe restrictions on properties and values of scalar structures.	"Define Scalar Structures for Code Generation"
Reference structure fields individually during indexing.	"Index Substructures and Fields"
Avoid type mismatch when assigning values to structures and fields.	"Assign Values to Structures and Fields"

Limitations

MATLAB to HLS code generation does not support:

- Use of character arrays and strings in structures.
- Structures with cell arrays in fixed-point conversion.
- Structures with variable field lengths.
- Array of structures.
- Structures defined in external files.

Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code

MATLAB Design Requirements for HDL and HLS Code Generation

When you generate HDL or High-Level Synthesis (HLS) code from your MATLAB design, you are converting an algorithm into an architecture that must meet hardware area and speed requirements.

Your MATLAB design has these requirements:

- MATLAB code within the design must be supported by HDL or HLS code generation.
- Inputs and outputs must not be matrices or structures.

If you are generating code at the command line, verify your code readiness for code generation by using this command:

```
coder.screener('design_function_name')
```

If you use the HDL Workflow Advisor to generate code, this check runs automatically.

For a MATLAB language support reference, including supported functions from Fixed-Point Designer, see “Functions Supported for HDL and HLS Code Generation” on page 1-2.

Guidelines for Writing MATLAB Code

For more efficient and faster HDL and HLS code generation, design your MATLAB code by using these best practices:

- Serialize your input and output data. Parallel data processing structures require more hardware resources and a higher pin count.
- Use add and subtract algorithms instead of algorithms that use functions, such as sine, divide, and modulo. Add and subtract operations use fewer hardware resources.
- Avoid large arrays and matrices. Large arrays and matrices require more registers and more RAM for storage. Whenever you need to use large arrays for memory, consider using the RAM mapping optimization to map these memories to RAM.
- Convert your code from floating-point to fixed-point. Floating-point data types are inefficient for hardware realization. HDL Coder provides an automated workflow for floating-point to fixed-point conversion.
- Unroll loops to increase speed at the cost of higher area. For HDL code generation, unroll fewer loops and enable the loop streaming optimization to conserve area at the cost of lower throughput.
- Optimize generated code by reducing the number of bits used to represent a variable i.e., bitwidth of the operand inside the MATLAB code.
- Use hardware-friendly rounding and overflow methods. It can be achieved by using `hdlfimath` function in your MATLAB code.

  ```
  function out = add(a,b)
     out = fi(a+b,1,12,4,hdlfimath);
  end
  ```

Additional guidelines for efficient HLS code generation:

- Modularize the design into subfunctions as much as possible. Doing so improves the readability of the code and makes it easier to specify constraints, such as `coder.hdl.constrainlatency`, on a particular region of code.
- Use a zero-based indexing scheme followed by array access with the addition of 1 as MATLAB uses 1-based indexing: for example, use `arrayVar(index+1)` instead of `arrayVar(index)`. The extra indexing logic does not need to be inserted by the code generator, thereby reducing overall area.

See Also

Apps
HDL Coder

Objects
`coder.HdlConfig`

Functions
`coder.hdl.loopspec` | `coder.hdl.pipeline` | `coder.hdl.ramconfig`

More About

- “Optimize MATLAB Loops” on page 8-29
- “For-Loop Best Practices for HDL Code Generation” on page 1-73
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “Mapping of Different Rounding and Overflow Methods from MATLAB to HLS” on page 1-61
- “Indexing Best Practices for HDL Code Generation” on page 1-68

Indexing Best Practices for HDL Code Generation

In MATLAB, array indices start from 1 and follow a 1-based indexing approach. However, HDL code uses 0-based indexing, where array indices start from 0. When you generate HDL code from a MATLAB algorithm, HDL Coder converts 1-based MATLAB indexing to 0-based HDL indexing, which may result in excess logic that can consume more hardware area. Using these best practices can optimize your code and reduce the amount of hardware area used.

Minimize Automatic Index Conversions

Because HDL Coder converts 1-based MATLAB indexing to 0-based HDL indexing when you generate code, HDL Coder may generate excess logic in the VHDL, Verilog, or SystemVerilog code to assist with this conversion. This excess logic leads to increased hardware area consumption.

To minimize the number of automatic conversions HDL Coder introduces when translating your MATLAB algorithm to HDL code and meet your hardware area requirements, you can store a zero-based index in index variables. When you perform an index operation, add 1 to the value of index. HDL Coder optimizes the automatic conversion.

<table>
<tr><th>Unoptimized MATLAB Code</th><th>Optimized MATLAB Code</th></tr>
<tr><td>

```
arr = [3 5 7 9];
num = length(arr);
% Index ii is 1-based
for ii = 1:num
    % 1-based index when indexing
    disp(arr(ii))
end
```

</td><td>

```
arr = [3 5 7 9];
num = length(arr);
% Index ii is 0-based
for ii = 0:num-1
    % Add 1 to the value of the 0-based index when indexing
    disp(arr(ii+1))
end
```

</td></tr>
</table>

Example of Area Consumption Optimization

This MATLAB algorithm enhances the contrast of images by transforming the values in an intensity image so that the histogram of the output image is approximately flat. Apply the indexing best practice to the algorithm. For more information on this algorithm, see “Image Enhancement by Histogram Equalization” on page 2-51. When you convert the unoptimized algorithm to HDL code, the conversion process can lead to excess logic. Compare the hardware area usage of the unoptimized algorithm to the optimized algorithm.

Unoptimized MATLAB Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mlhdlc_heq.m
% Histogram Equalization Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out, y_out, pixel_out] = ...
    mlhdlc_heq(x_in, y_in, pixel_in, width, height)

% Copyright 2011-2024 The MathWorks, Inc.

persistent histogram
persistent transferFunc
persistent histInd
persistent cumSum

if isempty(histogram)
    histogram = zeros(1, 2^14);
    transferFunc = zeros(1, 2^14);
    histInd = 0;
    cumSum = 0;
end

% Figure out indexes based on where we are in the frame
if y_in < height && x_in < width % valid pixel data
    histInd = pixel_in + 1;
elseif y_in == height && x_in == 0 % first column of height+1
    histInd = 1;
elseif y_in >= height % vertical blanking period
    histInd = min(histInd + 1, 2^14);
elseif y_in < height % horizontal blanking - do nothing
    histInd = 1;
end

% Read histogram (must be outside conditional logic)
histValRead = histogram(histInd);

% Read transfer function (must be outside conditional logic)
transValRead = transferFunc(histInd);

% If valid part of frame add one to pixel bin
% and keep transfer func val
if y_in < height && x_in < width
    histValWrite = histValRead + 1; % Add pixel to bin
    transValWrite = transValRead; % Write back same value
    cumSum = 0;
elseif y_in >= height % In blanking time index through all bins and reset to zero
    histValWrite = 0;
    transValWrite = cumSum + histValRead;
    cumSum = transValWrite;
else
    histValWrite = histValRead;
    transValWrite = transValRead;
end

% Write histogram (must be outside conditional logic)
histogram(histInd) = histValWrite;

% Write transfer function (must be outside conditional logic)
transferFunc(histInd) = transValWrite;

pixel_out = transValRead;
x_out = x_in;
y_out = y_in;
```

Optimized MATLAB Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mlhdlc_heq_bp.m
% Histogram Equalization Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out, y_out, pixel_out] = ...
    mlhdlc_heq_bp(x_in, y_in, pixel_in, width, height)

% Copyright 2011-2024 The MathWorks, Inc.

persistent histogram
persistent transferFunc
persistent histInd
persistent cumSum

if isempty(histogram)
    histogram = zeros(1, 2^14);
    transferFunc = zeros(1, 2^14);
    histInd = 0;
    cumSum = 0;
end

% Adjust histInd index variable to be 0-based
if y_in < height && x_in < width
    histInd = pixel_in;
elseif y_in == height && x_in == 0
    histInd = 0;
elseif y_in >= height
    histInd = min(histInd + 1, 2^14-1);
elseif y_in < height
    histInd = 0;
end

% Add 1 to the value of the 0-based histInd when indexing
histValRead = histogram(histInd+1);

% Add 1 to the value of the 0-based histInd when indexing
transValRead = transferFunc(histInd+1);

if y_in < height && x_in < width
    histValWrite = histValRead + 1;
    transValWrite = transValRead;
    cumSum = 0;
elseif y_in >= height
    histValWrite = 0;
    transValWrite = cumSum + histValRead;
    cumSum = transValWrite;
else
    histValWrite = histValRead;
    transValWrite = transValRead;
end

% Add 1 to the value of the 0-based histInd when indexing
histogram(histInd+1) = histValWrite;

% Add 1 to the value of the 0-based histInd when indexing
transferFunc(histInd+1) = transValWrite;

pixel_out = transValRead;
x_out = x_in;
y_out = y_in;
```

1. Run the command `mlhdlc_demo_setup('heq')` in the MATLAB Command Window. This command copies the files `mlhdlc_heq.m` and `mlhdlc_heq_tb.m` into a temporary working folder.
2. Create a new MATLAB function in the temporary working folder named `mlhdlc_heq_bp.m`. Copy the optimized MATLAB algorithm from the table into the function.
3. Create a new MATLAB script in the temporary working folder named `mlhdlc_heq_bp_tb.m`. Copy the contents of the `mlhdlc_heq_tb.m` testbench into `mlhdlc_heq_bp_tb.m`. Change the line with the function call `mlhdlc_heq` to `mlhdlc_heq_bp` in order for the

mlhdlc_heq_bp_tb.m testbench to exercise the function design of the optimized MATLAB algorithm, mlhdlc_heq_bp.

4 Generate HDL code from the unoptimized algorithm, mlhdlc_heq. For more information on how to generate HDL code from the command line, see “Generate HDL Code from MATLAB Code Using the Command Line Interface”.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_heq_tb';

hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_heq_tb';
hdlcfg.MapPersistentVarsToRAM = 0;
hdlcfg.TargetLanguage = 'VHDL';

codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_heq
```

5 Open the generated VHDL file and observe the automatic conversion HDL Coder introduces when translating the 1-based MATLAB algorithm to 0-based VHDL indexing. For example, compare the index operations from the MATLAB algorithm to the generated VHDL code.

Unoptimized MATLAB Algorithm

```
...
...
%Read histogram (must be outside conditional logic)
histValRead = histogram(histInd);

%Read transfer function (must be outside conditional logic)
transValRead = transferFunc(histInd);
...
...
```

Generated VHDL Code

```
...
...
--Read histogram (must be outside conditional logic)
sub_cast := signed(resize(histInd_temp, 32));
-- HDL Coder automatically converts between the 1- and 0-based indicies
histValRead := mlhdlc_heq_fixpt_histogram(to_integer(sub_cast - 1));
--Read transfer function (must be outside conditional logic)
sub_cast_0 := signed(resize(histInd_temp, 32));
-- HDL Coder automatically converts between the 1- and 0-based indicies
transValRead := mlhdlc_heq_fixpt_transferFunc(to_integer(sub_cast_0 - 1));
...
...
```

6 Generate HDL code from the optimized MATLAB algorithm that follows the best practice, mlhdlc_heq_bp. Set the fixptcfg and hdlcfg object parameters and run the codegen command.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_heq_bp_tb';

hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_heq_bp_tb';
hdlcfg.MapPersistentVarsToRAM = 0;
hdlcfg.TargetLanguage = 'VHDL';

codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_heq_bp
```

7 Open the generated VHDL file. For example, compare the index operations from the optimized MATLAB algorithm to the generated VHDL code. Observe how the generated VHDL code includes less logic.

Optimized MATLAB Algorithm

```
...
...
% Add 1 to the value of the 0-based histInd when indexing
histValRead = histogram(histInd+1);

% Add 1 to the value of the 0-based histInd when indexing
transValRead = transferFunc(histInd+1);
...
...
```

Generated VHDL Code

```
...
...
-- Add 1 to the value of the 0-based histInd when indexing
-- HDL Coder optimizes the automatic conversion
histValRead := mlhdlc_heq_bp_fixpt_histogram(to_integer(histInd_temp));
-- Add 1 to the value of the 0-based histInd when indexing
-- HDL Coder optimizes the automatic conversion
transValRead := mlhdlc_heq_bp_fixpt_transferFunc(to_integer(histInd_temp));
...
...
```

8 Set the `fixptcfg` and `hdlcfg` object parameters and run the `codegen` command to place and route the HDL code from the unoptimized and optimized MATLAB algorithms. For more information, see “Synthesis and Analysis” on page 9-13. For example, set the `fixptcfg` and `hdlcfg` object parameters and run the `codegen` command to place and route the HDL code from the optimized algorithm, `mlhdlc_heq_bp`.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_heq_bp_tb';

hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_heq_bp_tb';
hdlcfg.MapPersistentVarsToRAM = 1;
hdlcfg.TargetLanguage = 'VHDL';

hdlcfg.SynthesisTool = 'Xilinx Vivado';
hdlcfg.SynthesisToolChipFamily = 'Zynq';
hdlcfg.SynthesisToolDeviceName = 'xc7z020';
hdlcfg.SynthesisToolPackageName = 'clg400';
hdlcfg.SynthesisToolSpeedValue = '-1';
hdlcfg.SynthesizeGeneratedCode = true;

codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_heq_bp
```

VHDL code generated from the optimized MATLAB algorithm uses less hardware area, as HDL Coder introduces fewer automatic index conversions to compensate for the differences in indexing. This table compares the place and route results for the unoptimized and optimized MATLAB algorithms obtained from the synthesis tool.

Place and Route Result for Unoptimized MATLAB Algorithm

```
Parsed resource report file: mlhdlc_heq_fixpt_utilization_synth.rpt.

        Resource           Usage     Available     Utilization (%)
    __________________    _______    _________    _______________

    {'Slice LUTs'     }    {'171'}    {'53200' }       {'0.32' }
    {'Slice Registers'}    {'277'}    {'106400'}       {'0.26' }
```

Place and Route Result for Optimized MATLAB Algorithm

```
Parsed resource report file: mlhdlc_heq_bp_fixpt_utilization_synth.rpt.

        Resource          Usage     Available     Utilization (%)
    ___________________    _______    __________    _______________

    {'Slice LUTs'     }    {'147'}    {'53200' }       {'0.28' }
    {'Slice Registers'}    {'262'}    {'106400'}       {'0.25' }
```

See Also

Functions
`codegen` | `coder.config`

Objects
`coder.HdlConfig` | `coder.FixPtConfig`

More About

- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Edit Configuration Parameters for HDL Coder” on page 5-13
- “Edit Configuration Parameters for Fixed-Point Code Generation” on page 4-24
- “Apply RAM Mapping to Optimize Area” on page 21-120
- “Map Persistent Arrays and dsp.Delay Objects to RAM” on page 8-6
- “Resolve Index Errors During Simulation” on page 6-6

For-Loop Best Practices for HDL Code Generation

When you generate HDL code from your MATLAB design, you are converting an algorithm into an architecture that must meet hardware area and speed requirements. Some best practices for using loops in MATLAB code for HDL code generation are:

- Use monotonically increasing loop counters, with increments of 1, to minimize the amount of hardware generated in the HDL code.
- When implementing a loop to find an index value, use a conditional `if-else` statement inside of the loop.
- If you want to use the loop streaming optimization:
  - When assigning new values to persistent variables inside a loop, do not use other persistent variables on the right side of the assignment. Instead, use an intermediate variable.
  - If a loop modifies any elements in a persistent array, the loop should modify all of the elements in the persistent array.

Monotonically Increasing Loop Counters

By using monotonically increasing loop counters with increments of 1, you can reduce the amount of hardware in the generated HDL code. The following loop is an example of a monotonically increasing loop counter with increments of 1.

```
a=1;
for i=1:10
    a=a+1;
end
```

If a loop counter increases by an increment other than 1, the generated HDL code can require additional adders. Due to this additional hardware, do not use the following type of loop.

```
a=1;
for i=1:2:10
    a=a+1;
end
```

If a loop counter decreases, the generated HDL code can require additional adders. Due to this additional hardware, do not use the following type of loop.

```
a=1;
for i=10:-1:1
    a=a+1;
end
```

Find Indices Using Loops

When generating HDL code, control flow-like breaks are not possible, see “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4. Because a `for` loop must run for a constant number of iterations, to find an index using loops, use a conditional inside the loop, such as an `if-else` statement. For example, to find the first index of a one in the array `u`, you can use this code:

```
function y = fcn(u)
```

```
y = fi(0, 0, ceil(log2(numel(u))), 0);

for ii = cast(1:numel(u), 'like', y)
    if y == 0 && u(ii) == true
        y = ii;
    end
end
end
```

The final value for y is the smallest possible fixed-point value of the index that contains a one in the array u.

Persistent Variables in Loops

If a loop contains multiple persistent variables, when you assign values to persistent variables, use intermediate variables that are not persistent on the right side of the assignment. This practice makes dependencies clear to the compiler and assists internal optimizations during the HDL code generation process. If you want to use the loop streaming optimization to reduce the amount of generated hardware, this practice is recommended.

In the following example, var1 and var2 are persistent variables. var1 is used on the right side of the assignment. Because a persistent variable is on the right side of an assignment, do not use this type of loop:

```
for i=1:10
    var1 = 1 + i;
    var2 = var1 * 2;
end
```

Instead of using var1 on the right side of the assignment, use an intermediate variable that is not persistent. This example demonstrates this with the intermediate variable var_intermediate.

```
for i=1:10
    var_intermediate = 1 + i;
    var1 = var_intermediate;
    var2 = var_intermediate * 2;
end
```

Persistent Arrays in Loops

If a loop modifies elements in a persistent array, make sure that the loop modifies all of the elements in the persistent array. If all elements of the persistent array are not modified within the loop, HDL Coder cannot perform the loop streaming optimization.

In the following example, a is a persistent array. The first element is modified outside of the loop. Do not use this type of loop.

```
for i=2:10
    a(i)=1+i;
 end
 a(1)=24;
```

Rather than modifying the first element outside the loop, modify all of the elements inside the loop.

```
for i=1:10
   if i==1
```

```
      a(i)=24;
   else
      a(i)=1+i;
   end
end
```

See Also

Apps
HDL Coder

Objects
`coder.HdlConfig`

Functions
`coder.hdl.loopspec` | `coder.hdl.pipeline`

More About

- “Optimize MATLAB Loops” on page 8-29
- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4

MATLAB Test Bench Requirements and Best Practices for Code Generation

What Is a MATLAB Test Bench?

A test bench is a MATLAB script or function that you write to test the algorithm in your MATLAB design function. The test bench varies the input data to the design to simulate real world conditions. It can also can check that the output data meets design specifications.

HDL Coder uses the data it gathers from running your test bench with your design to infer fixed-point data types for floating-point to fixed-point conversion. The coder also uses the data to generate HDL and High-Level Synthesis (HLS) test data for verifying your generated code. For more information on how to write your test bench for the best results, see "MATLAB Test Bench Requirements and Best Practices for Code Generation" on page 1-76.

MATLAB Test Bench Requirements

You can use any MATLAB data type and function in your test bench.

A MATLAB test bench has the following requirements:

- For floating-point to fixed-point conversion, the test bench must be a script or a function with no inputs. The test bench can have local helper functions that take inputs.
- The inputs and outputs in your MATLAB design interface must use the same data types, sizes, and complexity in each call site in your test bench.
- If you enable the **Accelerate test bench for faster simulation** option in the Float-to-Fixed Workflow, the MATLAB constructs in your test bench loop must be compilable.

MATLAB Test Bench Best Practices

Use the following MATLAB test bench best practices:

- *Design your test bench to cover the full numeric range of data that the design must handle.* HDL Coder uses the data that it accumulates from running the test bench to infer fixed-point data types during floating-point to fixed-point conversion.

  If you call the design function multiple times from your test bench, the coder uses the accumulated data from each instance to infer fixed-point types. Both the design and the test bench can call local functions within the file or other functions on the MATLAB path. The call to the design function can be at any level of your test bench hierarchy.
- *Before trying to generate code, run your test bench in MATLAB.* If simulation is slow, accelerate your test bench. To learn how to accelerate your simulation, see "Accelerate MATLAB Algorithms".
- If you have a loop that calls your design function, use only compilable MATLAB constructs within the loop and enable the **Accelerate test bench for faster simulation** option.
- Before each test bench simulation run, use the `clear variables` command to reset your persistent variables.

To see an example of a test bench, enter this command:

```
showdemo mlhdlc_tutorial_float2fixed_files
```

See Also

Apps
HDL Coder

Objects
`coder.HdlConfig`

Functions
`coder.hdl.loopspec` | `coder.hdl.pipeline`

More About

- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “For-Loop Best Practices for HDL Code Generation” on page 1-73
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4

MATLAB to HDL Examples for Communications and Signal Processing Applications

- “HDL Code Generation for LMS Filter” on page 2-2
- “Bisection Algorithm to Calculate Square Root of an Unsigned Fixed-Point Number” on page 2-8
- “Timing Offset Estimation” on page 2-12
- “Data Packetization” on page 2-16
- “Transmit and Receive FIFO Registers” on page 2-22
- “HDL Code Generation for Harris Corner Detection Algorithm” on page 2-29
- “HDL Code Generation for Adaptive Median Filter” on page 2-36
- “Contrast Adjustment” on page 2-43
- “Image Enhancement by Histogram Equalization” on page 2-51
- “HDL Code Generation for Image Format Conversion from RGB to YUV” on page 2-56
- “High Dynamic Range Imaging” on page 2-60
- “Accelerate Pixel-Streaming Designs Using MATLAB Coder” on page 2-65
- “Enhanced Edge Detection from Noisy Color Video” on page 2-68
- “Verify Sobel Edge Detection Algorithm in MATLAB-to-HDL Workflow” on page 2-71

HDL Code Generation for LMS Filter

This example shows how to generate HDL code from a MATLAB® design that implements an LMS filter. The example also illustrates how to design a test bench that cancels out the noise signal by using this filter.

LMS Filter MATLAB Design

The MATLAB design used in the example is an implementation of an LMS (Least Mean Squares) filter. The LMS filter is a class of adaptive filter that identifies an FIR filter signal that is embedded in the noise. The LMS filter design implementation in MATLAB consists of a top-level function `mlhdlc_lms_fcn` that calculates the optimal filter coefficients to reduce the difference between the output signal and the desired signal.

```
design_name = 'mlhdlc_lms_fcn';
testbench_name = 'mlhdlc_lms_noise_canceler_tb';
```

Review the MATLAB design:

```
open(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB Design: Adaptive Noise Canceler algorithm using Least Mean Square
% (LMS) filter implemented in MATLAB
%
% Key Design pattern covered in this example:
% (1) Use of function calls
% (2) Function inlining vs instantiation knobs available in the coder
% (3) Use of system objects in the testbench to stream test vectors into the design
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%#codegen
function [filtered_signal, y, fc] = mlhdlc_lms_fcn(input, ...
                                        desired, step_size, reset_weights)
% 'input'  : The signal from Exterior Mic which records the ambient noise.
% 'desired': The signal from Pilot's Mic which includes
%             original music signal and the noise signal
% 'err_sig': The difference between the 'desired' and the filtered 'input'
%            It represents the estimated music signal (output of this block)
%
% The LMS filter is trying to retrieve the original music signal('err_sig')
% from Pilot's Mic by filtering the Exterior Mic's signal and using it to
% cancel the noise in Pilot's Mic. The coefficients/weights of the filter
% are updated(adapted) in real-time based on 'input' and 'err_sig'.

% register filter coefficients
persistent filter_coeff;
if isempty(filter_coeff)
    filter_coeff = zeros(1, 40);
end

% Variable Filter: Call 'mtapped_delay_fcn' function on path to create
% 40-step tapped delay
delayed_signal = mtapped_delay_fcn(input);
```

```
% Apply filter coefficients
weight_applied = delayed_signal .* filter_coeff;

% Call treesum function on matlab path to sum up the results
filtered_signal = mtreesum_fcn(weight_applied);

% Output estimated Original Signal
td = desired;
tf = filtered_signal;
esig = td - tf;
y = esig;

% Update Weights: Call 'update_weight_fcn' function on MATLAB path to
% calculate the new weights
updated_weight = update_weight_fcn(step_size, esig, delayed_signal, ...
                                   filter_coeff, reset_weights);

% update filter coefficients register
filter_coeff = updated_weight;
fc = filter_coeff;

function y = mtreesum_fcn(u)
%Implement the 'sum' function without a for-loop
%  y = sum(u);

%  The loop based implementation of 'sum' function is not ideal for
%  HDL generation and results in a longer critical path.
%  A tree is more efficient as it results in
%  delay of log2(N) instead of a delay of N delay

%  This implementation shows how to explicitly implement the vector sum in
%  a tree shape to enable hardware optimizations.

%  The ideal way to code this generically for any length of 'u' is to use
%  recursion but it is not currently supported by MATLAB Coder


% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

% This implementation is hardwired for a 40tap filter.

level1 = vsum(u);
level2 = vsum(level1);
level3 = vsum(level2);
level4 = vsum(level3);
level5 = vsum(level4);
level6 = vsum(level5);
y = level6;

function output = vsum(input)

coder.inline('always');

vt = input(1:2:end);

for i = int32(1:numel(input)/2)
```

```
    k = int32(i*2);
    vt(i) = vt(i) + input(k);
end

output = vt;

function tap_delay = mtapped_delay_fcn(input)
% The Tapped Delay function delays its input by the specified number
% of sample periods, and outputs all the delayed versions in a vector
% form. The output includes current input

% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

persistent u_d;
if isempty(u_d)
    u_d = zeros(1,40);
end


u_d = [u_d(2:40), input];

tap_delay = u_d;

function weights = update_weight_fcn(step_size, err_sig, ...
            delayed_signal, filter_coeff, reset_weights)
% This function updates the adaptive filter weights based on LMS algorithm

%   Copyright 2007-2022 The MathWorks, Inc.

% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

step_sig = step_size .* err_sig;
correction_factor = delayed_signal .* step_sig;
updated_weight = correction_factor + filter_coeff;

if reset_weights
    weights = zeros(1,40);
else
    weights = updated_weight;
end
```

The MATLAB function is modular and uses functions:

- `mtapped_delay_fcn` to calculate delayed versions of the input signal in vector form.
- `mtreesum_fcn` to calculate the sum of the applied weights in a tree structure. The individual sum is calculated by using a `vsum` function.
- `update_weight_fcn` to calculate the updated filter weights based on the least mean square algorithm.

LMS Filter MATLAB Test Bench

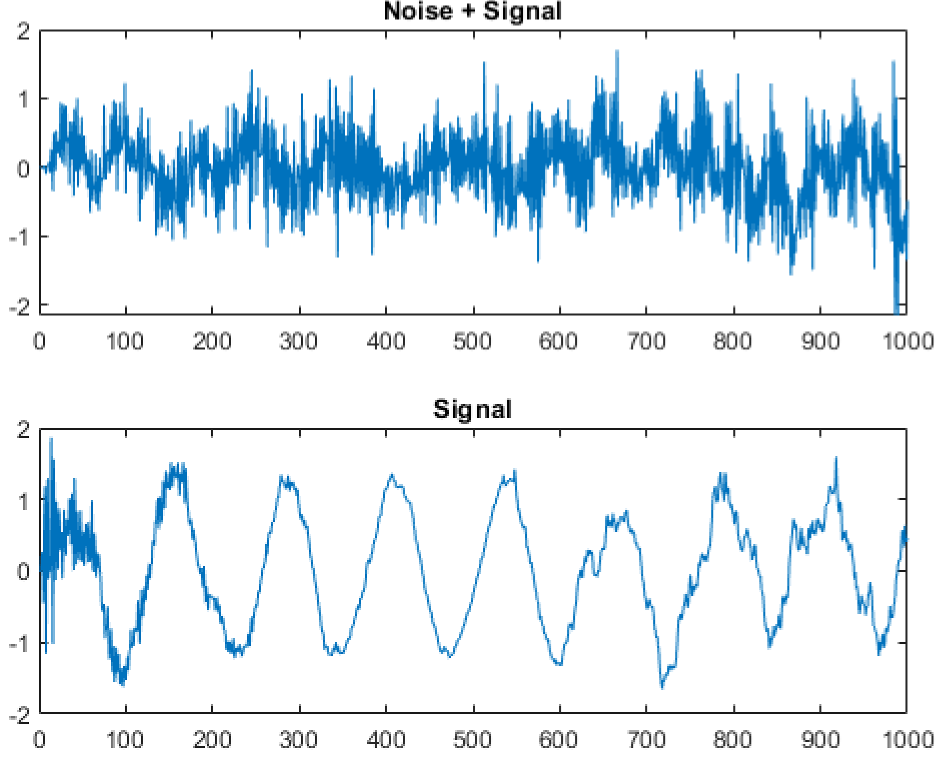
Review the MATLAB test bench:

```
open(testbench_name)


% Returns an adaptive FIR filter System object,
% HLMS, that computes the filtered output, filter error and the filter
% weights for a given input and desired signal using the Least Mean
% Squares (LMS) algorithm.

%   Copyright 2011-2022 The MathWorks, Inc.
clear('mlhdlc_lms_fcn');

hfilt2 = dsp.FIRFilter(...
        'Numerator', fir1(10, [.5, .75]));
rng('default'); % always default to known state
x = randn(1000,1);                              % Noise
d = step(hfilt2, x) + sin(0:.05:49.95)';         % Noise + Signal

stepSize = 0.01;
reset_weights =false;

hSrc = dsp.SignalSource(x);
hDesiredSrc = dsp.SignalSource(d);

hOut = dsp.SignalSink;
hErr = dsp.SignalSink;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Call to the design
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
while (~isDone(hSrc))
    [y, e] = mlhdlc_lms_fcn(step(hSrc), step(hDesiredSrc), ...
                                        stepSize, reset_weights);
    step(hOut, y);
    step(hErr, e);
end

figure('Name', [mfilename, '_signal_plot']);
subplot(2,1,1), plot(hOut.Buffer), title('Noise + Signal');
subplot(2,1,2),plot(hErr.Buffer), title('Signal');
```

Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

```
mlhdlc_lms_noise_canceler_tb
```



Create an HDL Coder Project

To generate HDL code from a MATLAB design:

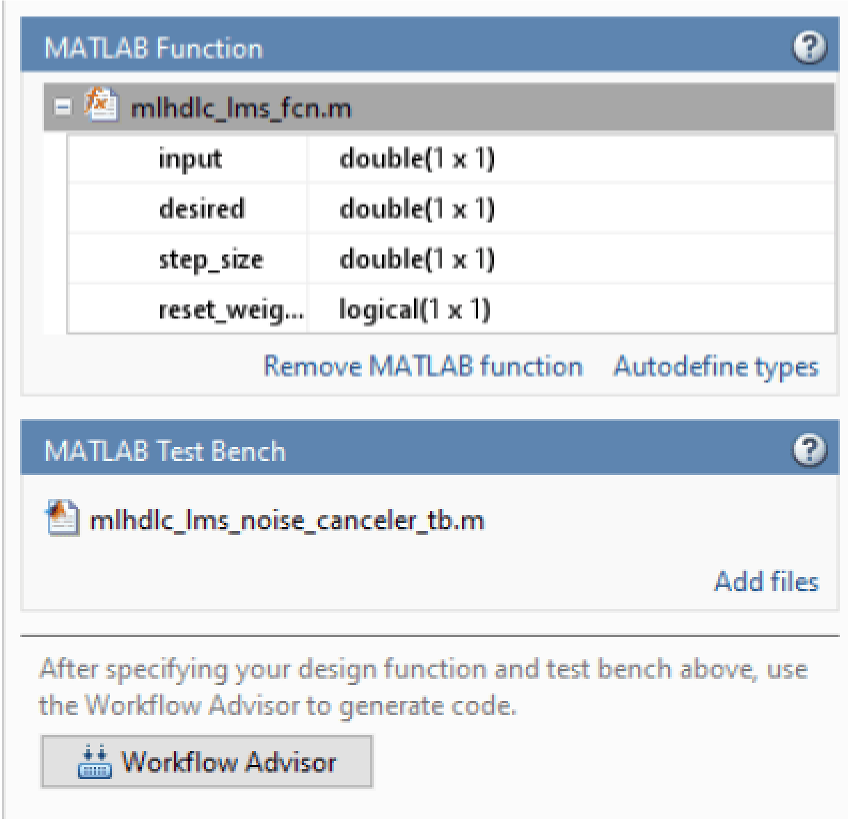
1. Create a HDL Coder project:

```
coder -hdlcoder -new mlhdlc_lms_nc
```

2. Add the file `mlhdlc_lms_fcn.m` to the project as the **MATLAB Function** and `mlhdlc_lms_noise_canceler_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefine types** to use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_lms_fcn`.

For a more complete tutorial on creating and populating MATLAB HDL Coder projects, see “Get Started with MATLAB to HDL Workflow”.



Run Fixed-Point Conversion and HDL Code Generation

1. Click the **Workflow Advisor** button to start the Workflow Advisor.
2. Right click the **HDL Code Generation** task and select **Run to selected task**.

A single HDL file `mlhdlc_lms_fcn_FixPt.vhd` is generated for the MATLAB design. To examine the generated HDL code for the filter design, click the hyperlinks in the Code Generation Log window.

If you want to generate a HDL file for each function in your MATLAB design, in the **Advanced** tab of the **HDL Code Generation** task, select the **Generate instantiable code for functions** check box. See also “Generate Instantiable Code for Functions” on page 5-12.

Bisection Algorithm to Calculate Square Root of an Unsigned Fixed-Point Number

This example shows how to generate HDL code from MATLAB® design implementing a bisection algorithm to calculate the square root of a number in fixed point notation.

Same implementation, originally using n-multipliers in HDL code, for wordlength `n`, under sharing and streaming optimizations, can generate HDL code with only one multiplier demonstrating the power of MATLAB® HDL Coder™ optimizations.

The design of the square-root algorithm shows the pipelining concepts to achieve a fast clock rate in resulting RTL design. Since this design is already in fixed point, you don't need to run fixed-point conversion.

MATLAB Design

```
% Design Sqrt
design_name = 'mlhdlc_sqrt';

% Test Bench for Sqrt
testbench_name = 'mlhdlc_sqrt_tb';
```

Lets look at the Sqrt Design

```
dbtype(design_name)
```

```
1     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2     % MATLAB design: Pipelined Bisection Square root algorithm
3     %
4     % Introduction:
5     %
6     % Implement SQRT by the bisection algorithm in a pipeline, for unsigned fixed
7     % point numbers (also why you don't need to run fixed-point conversion for this design).
8     % The demo illustrates the usage of a pipelined implementation for numerical algorithms.
9     %
10    % Key Design pattern covered in this example:
11    % (1) State of the bisection algorithm is maintained with persistent variables
12    % (2) Stages of the bisection algorithm are implemented in a pipeline
13    % (3) Code is written in a parameterized fashion, i.e. word-length independent, to work fo
14    %
15    % Ref. 1. R. W. Hamming, "Numerical Methods for Scientists and Engineers," 2nd, Ed, pp 67-6
16    %      2. Bisection method, http://en.wikipedia.org/wiki/Bisection_method, (accessed 02/18/
17    %
18
19    %   Copyright 2013-2015 The MathWorks, Inc.
20
21    %#codegen
22    function [y,z] = mlhdlc_sqrt( x )
23        persistent sqrt_pipe
24        persistent in_pipe
25       if isempty(sqrt_pipe)
26           sqrt_pipe = fi(zeros(1,x.WordLength),numerictype(x));
27           in_pipe = fi(zeros(1,x.WordLength),numerictype(x));
```

```
28      end
29
30      % Extract the outputs from pipeline
31      y = sqrt_pipe(x.WordLength);
32      z = in_pipe(x.WordLength);
33
34      % for analysis purposes you can calculate the error between the fixed-point bisection ro
35      %Q = [double(y).^2, double(z)];
36      %[Q, diff(Q)]
37
38      % work the pipeline
39      for itr = x.WordLength-1:-1:1
40          % move pipeline forward
41          in_pipe(itr+1) = in_pipe(itr);
42          % guess the bits of the square-root solution from MSB to the LSB of word length
43          sqrt_pipe(itr+1) = guess_and_update( sqrt_pipe(itr), in_pipe(itr+1), itr );
44      end
45
46      %% Prime the pipeline
47      % with new input and the guess
48      in_pipe(1) = x;
49      sqrt_pipe(1) = guess_and_update( fi(0,numerictype(x)), x, 1 );
50
51      %% optionally print state of the pipeline
52      %disp('************** State of Pipeline **********************')
53      %double([in_pipe; sqrt_pipe])
54
55      return
56   end
57
58   % Guess the bits of the square-root solution from MSB to the LSB in
59   % a binary search-fashion.
60   function update = guess_and_update( prev_guess, x, stage )
61       % Key step of the bisection algorithm is to set the bits
62       guess = bitset( prev_guess, x.WordLength - stage + 1);
63       % compare if the set bit is a candidate solution to retain or clear it
64       if ( guess*guess <= x )
65           update = guess;
66       else
67           update = prev_guess;
68       end
69       return
70   end
```

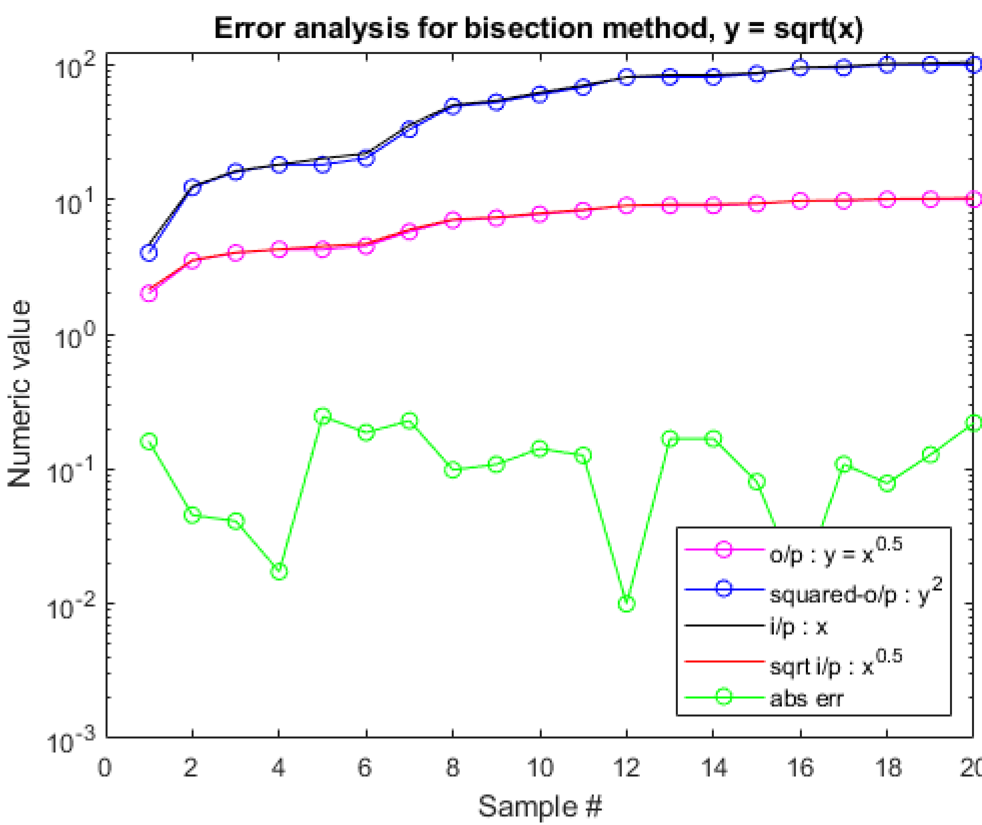
Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sqrt_tb
```

```
Iter = 01| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 02| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 03| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 04| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 05| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 06| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 07| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
```

```
Iter = 08| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 09| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 10| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 11| Input = 4.625| Output = 0000010000 (2.00) | actual = 2.150581 | abserror = 0.150581
Iter = 12| Input = 12.500| Output = 0000011100 (3.50) | actual = 3.535534 | abserror = 0.035534
Iter = 13| Input = 16.250| Output = 0000100000 (4.00) | actual = 4.031129 | abserror = 0.031129
Iter = 14| Input = 18.125| Output = 0000100010 (4.25) | actual = 4.257347 | abserror = 0.007347
Iter = 15| Input = 20.125| Output = 0000100010 (4.25) | actual = 4.486090 | abserror = 0.236090
Iter = 16| Input = 21.875| Output = 0000100100 (4.50) | actual = 4.677072 | abserror = 0.177072
Iter = 17| Input = 35.625| Output = 0000101110 (5.75) | actual = 5.968668 | abserror = 0.218668
Iter = 18| Input = 50.250| Output = 0000111000 (7.00) | actual = 7.088723 | abserror = 0.088723
Iter = 19| Input = 54.000| Output = 0000111010 (7.25) | actual = 7.348469 | abserror = 0.098469
Iter = 20| Input = 62.125| Output = 0000111110 (7.75) | actual = 7.881941 | abserror = 0.131941
Iter = 21| Input = 70.000| Output = 0001000010 (8.25) | actual = 8.366600 | abserror = 0.116600
Iter = 22| Input = 81.000| Output = 0001001000 (9.00) | actual = 9.000000 | abserror = 0.000000
Iter = 23| Input = 83.875| Output = 0001001000 (9.00) | actual = 9.158330 | abserror = 0.158330
Iter = 24| Input = 83.875| Output = 0001001000 (9.00) | actual = 9.158330 | abserror = 0.158330
Iter = 25| Input = 86.875| Output = 0001001010 (9.25) | actual = 9.320676 | abserror = 0.070676
Iter = 26| Input = 95.125| Output = 0001001110 (9.75) | actual = 9.753205 | abserror = 0.003205
Iter = 27| Input = 97.000| Output = 0001001110 (9.75) | actual = 9.848858 | abserror = 0.098858
Iter = 28| Input = 101.375| Output = 0001010000 (10.00) | actual = 10.068515 | abserror = 0.06851
Iter = 29| Input = 102.375| Output = 0001010000 (10.00) | actual = 10.118053 | abserror = 0.11805
Iter = 30| Input = 104.250| Output = 0001010000 (10.00) | actual = 10.210289 | abserror = 0.21028
```



Create a New HDL Coder Project

```
coder -hdlcoder -new mlhdlc_sqrt_prj
```

Next, add the file `mlhdlc_sqrt.m` to the project as the MATLAB Function and `mlhdlc_sqrt_tb.m` as the MATLAB Test Bench.

For a more complete tutorial on creating and populating MATLAB HDL Coder projects, see “Get Started with MATLAB to HDL Workflow”.

Run HDL Code Generation

This design is already in fixed point and suitable for HDL code generation. It is not desirable to run floating point to fixed point advisor on this design.

1. Click the **Workflow Advisor** button to start the HDL Workflow Advisor.
2. In the **HDL Workflow Advisor** task, set **Fixed-point conversion** to `Keep original types`.
3. In the **HDL Code Generation** task, select the **Optimizations** tab.
4. Clear the **MAP persistent array variables to RAMs** checkbox. Unchecking this option prevents the pipeline from being inferred as RAM.
5. Additionally, you can select **Distribute pipeline registers**, set **Resource sharing factor** to the `wordlength` (10 here), and select **Stream Loops**.
6. Right-click on the **HDL Code Generation** task and click **Run This Task**.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

Examine the Synthesis Results

1. Run the logic synthesis step with the following default options if you have ISE installed on your machine.
2. In the synthesis report, note the clock frequency reported by the synthesis tool without any optimization options enabled.
3. Typically the timing performance of this design, using Xilinx ISE synthesis tool for the `Virtex7` chip family, device `xc7v285t`, speed grade `-3`, is around `229MHz` with a maximum combinatorial path delay of `0.406ns`.
4. Optimizations for this design (loop streaming and multiplier sharing) work to reduce resource usage, with a moderate trade-off on timing. For the particular word-length size in test bench you see a reduction of `n` multipliers to one.

Timing Offset Estimation

This example shows how to generate HDL code from a basic lead-lag timing offset estimation algorithm implemented in MATLAB® code.

Introduction

In wireless communication systems, receive data is oversampled at the RF front end. This serves several purposes, including providing sufficient sampling rates for receive filtering.

However, one of the most important functions is to provide multiple sampling points on the received waveform such that data can be sampled near the maximum amplitude point in the received waveform. This example illustrates a basic lead-lag time offset estimation core, operating recursively.

The generated hardware core for this design operates at 1/os_rate where os_rate is the oversampled rate. That is, for 8 oversampled clock cycles this core iterates once. The output is at the symbol rate.

```
design_name = 'mlhdlc_comms_toe';
testbench_name = 'mlhdlc_comms_toe_tb';
```

Let us take a look at the MATLAB® design.

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Time Offset Estimation
%
%% Introduction:
%
% The generated hardware core for this design operates at 1/os_rate
% where os_rate is the oversampled rate. That is, for 8 oversampled clock cycles
% this core iterates once. The output is at the symbol rate.
%
% Key design pattern covered in this example:
% (1) Data is sent in a vector format, stored in a register and accessed
% multiple times
% (2) The core also illustrates basic mathematical operations
%

%   Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [tauh,q] = mlhdlc_comms_toe(r,mu)

persistent tau
persistent rBuf

os_rate = 8;
if isempty(tau)
    tau = 0;
    rBuf = zeros(1,3*os_rate);
end

rBuf = [rBuf(1+os_rate:end) r];
```

```
taur = round(tau);

% Determine lead/lag values and compute offset error
zl = rBuf(os_rate+taur-1);
zo = rBuf(os_rate+taur);
ze = rBuf(os_rate+taur+1);
offsetError = zo*(ze-zl);

% update tau
tau = tau + mu*offsetError;

tauh = tau;

q = zo;
```

```
type(testbench_name);
```

```
function mlhdlc_comms_toe_tb
%

%   Copyright 2011-2015 The MathWorks, Inc.

os_rate = 8;
Ns = 128;
SNR = 100;
mu = .5; % smoothing factor for time offset estimates

% create simulated signal
rng('default'); % always default to known state
b = round(rand(1,Ns));
d = reshape(repmat(b*2-1,os_rate,1),1,Ns*os_rate);

x = [zeros(1,Ns*os_rate) d zeros(1,Ns*os_rate)];
y = awgn(x,SNR);

w = fir1(3*os_rate+1,1/os_rate)';
z = filter(w,1,y);
r = z(4:end); % give it an offset to make things interesting

%tau = 0;
Nsym = floor(length(r)/os_rate);
tauh = zeros(1,Nsym-1); q = zeros(1,Nsym-1);
for i1 = 1:Nsym-1
    rVec = r(1+(i1-1)*os_rate:i1*os_rate);

    % Call to the Timing Offset Estimation Algorithm
    [tauh(i1),q(i1)] = mlhdlc_comms_toe(rVec,mu);
end

indexes = 1:os_rate:length(tauh)*os_rate;
indexes = indexes+tauh+os_rate-1-os_rate*2;

Fig1Loc=figposition([5 50 90 40]);
H_f1=figure(1); clf;
set(H_f1,'position',Fig1Loc);
subplot(2,1,1)
plot(r,'b');
```

```
hold on
plot(indexes,q,'ro');
axis([indexes(1) indexes(end) -1.5 1.5]);
title('Received Signal with Time Correct Detections');
subplot(2,1,2)
plot(tauh);
title('Estimate of Time Offset');

function y=figposition(x)
%FIGPOSITION Positions figure window irrespective of the screen resolution
% Y=FIGPOSITION(X) generates a vector the size of X.
% This specifies the location of the figure window in pixels
%
screenRes=get(0,'ScreenSize');
% Convert x to pixels
y(1,1)=(x(1,1)*screenRes(1,3))/100;
y(1,2)=(x(1,2)*screenRes(1,4))/100;
y(1,3)=(x(1,3)*screenRes(1,3))/100;
y(1,4)=(x(1,4)*screenRes(1,4))/100;
```

Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_comms_toe_tb
```

Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_toe
```

Next, add the file `mlhdlc_comms_toe.m` to the project as the MATLAB Function and `mlhdlc_comms_toe_tb.m` as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor from the Workflow Advisor button and right-click on the **Code Generation** step and choose the option **Run to selected task** to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

Data Packetization

This example shows how to generate HDL code from a MATLAB® design that packetizes a transmit sequence.

Introduction

In wireless communication systems receive data is oversampled at the RF front end. This serves several purposes, including providing sufficient sampling rates for receive filtering.

However, one of the most important functions is to provide multiple sampling points on the received waveform such that data can be sampled near the maximum amplitude point in the received waveform. This example illustrates a basic lead-lag time offset estimation core, operating recursively.

The generated hardware core for this design operates at 1/os_rate where os_rate is the oversampled rate. That is, for 8 oversampled clock cycles this core iterates once. The output is at the symbol rate.

```
design_name = 'mlhdlc_comms_data_packet';
testbench_name = 'mlhdlc_comms_data_packet_tb';
```

Let us take a look at the MATLAB design.

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Data packetization
%
% Introduction:
%
% This core is meant to illustrate packetization of a transmit sequence.
% There is a "pad" data section, which allows for the transmit amplifier to
% settle. This is then followed by a 65-bit training sequence. This is
% followed by the number of symbols beginning encoded into two bytes or
% 16-bits. This is then followed by a variable length data sequence and a
% CRC. All bits can optionally be differentially encoded.
%
% Key design pattern covered in this example:
% (1) Design illustrates the us of binary operands, such as bitxor
% (2) Shows how to properly segment persistent variables for register an
% BRAM access
% (3) Illustrates the use of fi math
% (4) Shows how to properly format and store ROM data, e.g., padData

%   Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [symbolOut, reByte] = ...
    mlhdlc_comms_data_packet(emptyFlag, byteValue, numberSymbols, diffOn, Nts, Npad)

persistent trainBits1 padData
persistent valueCRC crcVector bitPrev
persistent inPacketFlag  bitOfByteIndex symbolCount

fm = hdlfimath;
if isempty(symbolCount)
```

```
    symbolCount = 1;
    inPacketFlag = 0;
    valueCRC = fi(1, 0,16,0, fm);
    bitOfByteIndex = 1;
    bitPrev = fi(1, 0,1,0, fm);
    crcVector = zeros(1,16);
end
if isempty(trainBits1)
    % data-set already exists
    trainBits1 = TRAIN_DATA;
    padData = PAD_DATA;
end

%genPoly = 69665;
genPoly = fi(65535, 0,16,0, fm);
byteUint8 = uint8(byteValue);

reByte = 0;
symbolOut = fi(0, 0,1,0, fm);

%the first condition is whether or not we're currently processing a packet
if inPacketFlag == 1
    bitOut = fi(0, 0,1,0, fm);
    if symbolCount <= Npad
        bitOut(:) = padData(symbolCount);
    elseif symbolCount <= Npad+Nts
        bitOut(:) = trainBits1(symbolCount-Npad);
    elseif symbolCount <= Npad+Nts+numberSymbols
        bitOut(:) = bitget(byteUint8,9-bitOfByteIndex);
        bitOfByteIndex = bitOfByteIndex + 1;
        if bitOfByteIndex == 9 && symbolCount < Npad+Nts+numberSymbols
            bitOfByteIndex = 1;
            reByte = 1; % we've exhausted this one so pop new one off
        end
    elseif symbolCount <= Npad+Nts+numberSymbols+16
        bitOut(:) = 0;
    elseif symbolCount <= Npad+Nts+numberSymbols+32
        bitOut(:) = crcVector(symbolCount-(Npad+Nts+numberSymbols+16));
    else
        inPacketFlag = 0; %we're done
    end

    %leadValue = 0;
    % here we have the bit going out so if past Nts+Npad then form CRC.
    % Note that we throw 16 zeros on the end in order to flush the CRC
    if symbolCount > Npad+Nts && symbolCount <= Npad+Nts+numberSymbols+16

        valueCRCsh1 = bitsll(valueCRC, 1);
        valueCRCadd1 = bitor(valueCRCsh1, fi(bitOut, 0,16,0, fm));
        leadValue = bitget(valueCRCadd1,16);
        if leadValue == 1
            valueCRCxor = bitxor(valueCRCadd1, genPoly);
        else
            valueCRCxor = valueCRCadd1;
        end
        valueCRC = valueCRCxor;
        if symbolCount == Npad+Nts+numberSymbols+16
            crcVector(:) = bitget( valueCRC, 16:-1:1);
```

```
            end
        end

        if diffOn == 0 || symbolCount <= Npad+Nts
            symbolOut(:) = bitOut;
        else
            if bitPrev == bitOut
                symbolOut(:) = 1;
            else
                symbolOut(:) = 0;
            end
        end
        bitPrev(:) = symbolOut;

        symbolCount = symbolCount + 1; %total number of symbols transmitted
    else
        % we're not processing a packet and waiting for a new packet to arrive
        if emptyFlag == 0
            % reset everything
            inPacketFlag = 1;
            % toggle re to grab data
            reByte = 1;
            symbolCount = 1;
            bitOfByteIndex = 1;
            valueCRC(:) = 65535;
            bitPrev(:) = 0;
        end
    end

    end
```

```
type(testbench_name);
```

```
function mlhdlc_comms_data_packet_tb
%

%   Copyright 2011-2015 The MathWorks, Inc.

% generate transmit data, note the first two bytes are the data length
numberBytes = 8; % this is total number of symbols
numberSymbols = numberBytes*8;
rng(1); % always default to known state
data = [floor(numberBytes/2^8) mod(numberBytes,2^8) ...
    round(rand(1,numberBytes-2)*255)];

% generate training data helper function
make_train_data('TRAIN_DATA');

% make sure training data is generated
pause(2)
[~] = which('TRAIN_DATA');

trainBits1 = TRAIN_DATA;
Nts = length(trainBits1);

make_pad_data('PAD_DATA');
pause(2)
```

```
[~] = which('PAD_DATA');
Npad = 2^9;

% Give number of samples, where the start of the sequence flag will be
% (indicated by a zero), as well as an output buffer for generated symbols
Nsamp = 1000;
Noffset = 20;
emptyFlagHold = ones(1,Nsamp); emptyFlagHold(Noffset) = 0;
symbolOutHold = zeros(1,Nsamp);

dataIndex = 1;
byteValue = 0;
diffOn = 1; % 0 - regular encoding, 1 - differential encoding
for i1 = 1:Nsamp
    emptyFlag = emptyFlagHold(i1);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Call to the design
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    [symbolOut, reByte] = ...
        mlhdlc_comms_data_packet(emptyFlag, byteValue, numberSymbols, diffOn, Nts, Npad);

    % This set of code emulates the external FIFO interface
    if reByte == 1 % when high, pop a value off the input FIFO
        byteValue = data(dataIndex);
        dataIndex = dataIndex + 1;
    end
    symbolOutHold(i1) = symbolOut;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is all code to verify we did the encoding properly
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% grad training data - not differentially encoded
symbolTrain = symbolOutHold(1+Noffset+Npad:Noffset+Npad+Nts);

% grab user data and decode if necessary
symbolEst = zeros(1,numberSymbols);
symbolPrev = trainBits1(end);
if diffOn == 0
    symbolData = ...
        symbolOutHold(1+Noffset+Npad+Nts:Noffset+Npad+Nts+numberSymbols); %#ok<NASGU>
else
    % decoding is simply comparing adjacent received symbols
    symbolTemp = ...
       symbolOutHold(1+Noffset+Npad+Nts:Noffset+Npad+Nts+numberSymbols+32);
    for i1 = 1:length(symbolTemp)
        if symbolTemp(i1) == symbolPrev
            symbolEst(i1) = 1;
        else
            symbolEst(i1) = 0;
        end
        symbolPrev = symbolTemp(i1);
    end
end

% training data
```

```
trainDataEst = symbolTrain(1:Nts);
trainDiff = abs(trainDataEst-trainBits1');

% user data
userDataEst = symbolEst(1:numberSymbols);
dataEst = zeros(1,numberBytes);
for i1 = 1:numberBytes
    y = userDataEst((i1-1)*8+1:i1*8);
    dataEst(i1) = bin2dec(char(y+48));
end
userDiff = abs(dataEst-data);

disp(['Training Difference: ',num2str(sum(trainDiff)), ...
    ' User Data Difference: ',num2str(sum(userDiff))]);

% run it through and check CRC
genPoly = 69665;
c = symbolEst;
cEst = c(1,:);
cEst2 = [cEst(1:end-32) cEst(end-15:end)];
cEst = cEst2;

valueCRCc = 65535;
for i1 = 1:length(cEst)
    valueCRCsh1 = bitsll(uint16(valueCRCc), 1);
    valueCRCadd1 = bitor(uint16(valueCRCsh1), cEst(i1));
    leadValue = bitget( valueCRCadd1, 16);
    if (leadValue == 1)
        valueCRCxor = bitxor(uint16(valueCRCadd1), uint16(genPoly));
    else
        valueCRCxor = bitxor(uint16(valueCRCadd1), 0);
    end
    valueCRCc = valueCRCxor;
end
if valueCRCc == 0
    disp('CRC decoded correctly');
else
    disp('CRC check failed');
end

function make_train_data(filename)
x  = load('mlhdlc_dpack_train_data.txt');
fid = fopen([filename,'.m'],'w+');
fprintf(fid,['function y = ' filename '\n']);
fprintf(fid,'%%#codegen\n');
fprintf(fid,'y = [\n');
fprintf(fid,'%1.0e\n',x);
fprintf(fid,'];\n');
fclose(fid);

function make_pad_data(filename)
rng(1);
x = round(rand(1,2^9));
fid = fopen([filename,'.m'],'w+');
fprintf(fid,['function y = ' filename '\n']);
fprintf(fid,'%%#codegen\n');
fprintf(fid,'y = [\n');
fprintf(fid,'%1.0e\n',x);
```

```
fprintf(fid,'];\n');
fclose(fid);
```

Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_comms_data_packet_tb
```

```
Training Difference: 0 User Data Difference: 0
CRC decoded correctly
```

Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_dpack
```

Next, add the file `mlhdlc_comms_data_packet.m` to the project as the MATLAB Function and `mlhdlc_comms_data_packet_tb.m` as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor from the Workflow Advisor button and right-click on the **Code Generation** step and choose the option **Run to selected task** to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

Transmit and Receive FIFO Registers

This example shows how to generate HDL code from MATLAB® code that models the data transfer between a transmit and receive first-in, first-out (FIFO) register or buffer. This example contains two functions that represent a receive FIFO buffer and a transmit FIFO buffer, and a test bench `mlhdlc_fifo_tb` that simulates the data transfer that occurs between the two buffers. Each function is hardware-ready and exhibits good practices and guidelines to follow when writing MATLAB functions to generate efficient HDL code. For more information on guidelines to follow, see “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66.

View Example Functions and Test Bench

Open the MATLAB design for the transmit FIFO and the receive FIFO.

```
type('mlhdlc_rx_fifo');
```

```
function [dout, empty, byte_ready, full, bytes_available] = ...
    mlhdlc_rx_fifo(get_byte, store_byte, byte_in, reset_fifo, fifo_enable)
%

%   Copyright 2014-2015 The MathWorks, Inc.

%
%  First In First Out (FIFO) structure.
%  This FIFO stores integers.
%  The FIFO is actually a circular buffer.
%
persistent head tail fifo byte_out handshake

if (reset_fifo || isempty(head))
    head = 1;
    tail = 2;
    byte_out = 0;
    handshake = 0;
end

if isempty(fifo)
    fifo = zeros(1,1024);
end

full = 0;
empty = 0;
byte_ready = 0;

% Section for checking full and empty cases
if ((tail == 1 && head == 1024) || ((head + 1) == tail))
    empty = 1;
end
if ((head == 1 && tail == 1024) || ((tail + 1) == head))
    full = 1;
end

% handshaking logic
if get_byte == 0
    handshake = 0;
```

```
end
if handshake == 1
    byte_ready = 1;
end

if (fifo_enable == 1)
    %%%%%%%%%%%%%get%%%%%%%%%%%%%%%%%%
    if (get_byte && ~empty && handshake == 0 )
        head = head + 1;
        if head == 1025
            head = 1;
        end
        byte_out = fifo(head);
        byte_ready = 1;
        handshake = 1;
    end
    %%%%%%%%%%%%put%%%%%%%%%%%%%%%%%%%%
    if (store_byte && ~full)
        fifo(tail) = byte_in;
        tail = tail + 1;
        if tail == 1025
            tail = 1;
        end
    end
end

% Section for calculating num bytes in FIFO
if (head < tail)
    bytes_available = (tail - head) - 1;
else
    bytes_available = (1024 - head) + tail - 1;
end

dout = byte_out;
end
```

```
type('mlhdlc_tx_fifo');
```

```
function [dout, empty, byte_received, full, bytes_available, dbg_fifo_enable] = ...
    mlhdlc_tx_fifo(get_byte, store_byte, byte_in, reset_fifo, fifo_enable)
%

%   Copyright 2014-2015 The MathWorks, Inc.

%
%  First In First Out (FIFO) structure.
%  This FIFO stores integers.
%  The FIFO is actually a circular buffer.
%
persistent head tail fifo byte_out handshake

if (reset_fifo || isempty(head))
    head = 1;
    tail = 2;
    byte_out = 0;
    handshake = 0;
end
```

```
if isempty(fifo)
    fifo = zeros(1,1024);
end

full = 0;
empty = 0;
byte_received = 0;

% Section for checking full and empty cases
if ((tail == 1 && head == 1024) || ((head + 1) == tail))
    empty = 1;
end
if ((head == 1 && tail == 1024) || ((tail + 1) == head))
    full = 1;
end

% handshaking logic
if store_byte == 0
    handshake = 0;
end
if handshake == 1
    byte_received = 1;
end

if (fifo_enable == 1)
    %%%%%%%%%%%%%%get%%%%%%%%%%%%%%%%%%%%%
    if (get_byte && ~empty)
        head = head + 1;
        if head == 1025
            head = 1;
        end
        byte_out = fifo(head);
    end
    %%%%%%%%%%%%%put%%%%%%%%%%%%%%%%%%%%%
    if (store_byte && ~full && handshake == 0)
        fifo(tail) = byte_in;
        tail = tail + 1;
        if tail == 1025
            tail = 1;
        end
        byte_received = 1;
        handshake = 1;
    end
end

% Section for calculating num bytes in FIFO
if (head < tail)
    bytes_available = (tail - head) - 1;
else
    bytes_available = (1024 - head) + tail - 1;
end

dout = byte_out;
dbg_fifo_enable = fifo_enable;
end
```

Open the MATLAB design for the test bench that exercises both designs. This test bench test both the transmit and receive FIFOs. However, when generating HDL code, because you have individual functions for the transmit and receive FIFOs, you need individual test benches to test both functions and generate code. For simulation purposes, you can use `mlhdlc_fifo_tb`, but for HDL code generation, use the receive FIFO test bench `mlhdlc_rx_fifo_tb` with the receive FIFO function `mlhdlc_rx_fifo`, and use the transmit FIFO test bench `mlhdlc_tx_fifo` with the transmit FIFO function `mlhdlc_tx_fifo`.

```
type('mlhdlc_fifo_tb');
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simulation parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% data payload creation

%   Copyright 2014-2015 The MathWorks, Inc.

messageASCII = 'Hello World!';
message = double(unicode2native(messageASCII));
msgLength = length(message);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TX_FIFO core
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
numBytesToFifo = 1;
tx_get_byte = 0;
tx_full = 0;
tx_byte_received = 0;
i1 = 1;

while (numBytesToFifo <= msgLength && ~tx_full)
    % first thing the processor does is clear the internal tx fifo
    if i1 == 1
        tx_reset_fifo = 1;
        mlhdlc_tx_fifo(0, 0, 0, tx_reset_fifo, 1);
    else
        tx_reset_fifo = 0;
    end
    if (i1 > 1)
        tx_data_in = message(numBytesToFifo);
        numBytesToFifo = numBytesToFifo + 1;
        tx_store_byte = 1;
        while (tx_byte_received == 0)
            [tx_data_out, tx_empty, tx_byte_received, tx_full, tx_bytes_available] = ...
                mlhdlc_tx_fifo(tx_get_byte, tx_store_byte, tx_data_in, tx_reset_fifo, 1);
        end
        tx_store_byte = 0;
        while (tx_byte_received == 1)
            [tx_data_out, tx_empty, tx_byte_received, tx_full, tx_bytes_available] = ...
                mlhdlc_tx_fifo(tx_get_byte, tx_store_byte, tx_data_in, tx_reset_fifo, 1);
        end
    end
    i1 = i1 + 1;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Transfer Bytes from TX FIFO to RX FIFO
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
i1 = 1;

tx_get_byte = 0;
tx_store_byte = 0;
tx_data_in = 0;
tx_reset_fifo = 0;

rx_get_byte = 0;
rx_data_in = 0;
rx_reset_fifo = 0;

while (tx_bytes_available > 0)
        if i1 == 1
            rx_reset_fifo = 1;
            mlhdlc_rx_fifo(0, 0, 0, rx_reset_fifo, 1);
        else
            rx_reset_fifo = 0;
        end
        if (i1 > 1)
            tx_get_byte = 1;
            rx_store_byte = 1;
            [tx_data_out, tx_empty, tx_byte_received, tx_full, tx_bytes_available] = ...
                mlhdlc_tx_fifo(tx_get_byte, tx_store_byte, tx_data_in, tx_reset_fifo, 1);

            rx_data_in = tx_data_out;

            [rx_data_out, rx_empty, rx_byte_ready, rx_full, rx_bytes_available] = ...
                mlhdlc_rx_fifo(rx_get_byte, rx_store_byte, rx_data_in, rx_reset_fifo, 1);
        end
        i1 = i1 + 1;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RX_FIFO core
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
numBytesFromFifo = 1;
rx_store_byte = 0;
rx_byte_received = 0;
i1 = 1;
msgBytes = zeros(1,msgLength);

while (~rx_empty)
    % first thing the processor does is clear the internal rx fifo
    if (i1 > 1)
        rx_get_byte = 1;
        while (rx_byte_ready == 0)
            [rx_data_out, rx_empty, rx_byte_ready, rx_full, rx_bytes_available] = ...
                mlhdlc_rx_fifo(rx_get_byte, rx_store_byte, rx_data_in, rx_reset_fifo, 1);
        end
        msgBytes(i1-1) = rx_data_out;
        rx_get_byte = 0;
        while (rx_byte_ready == 1)
            [rx_data_out, rx_empty, rx_byte_ready, rx_full, rx_bytes_available] = ...
                mlhdlc_rx_fifo(rx_get_byte, rx_store_byte, rx_data_in, rx_reset_fifo, 1);
        end
    end
    i1 = i1 + 1;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
numRecBytes = numBytesFromFifo;
if sum(msgBytes-message) == 0
    disp('Received message correctly');
else
    disp('Received message incorrectly');
end
native2unicode(msgBytes)
```

Simulate the Design

Simulate the design with the test bench before generating code to ensure there are no runtime errors.

```
mlhdlc_fifo_tb
```

```
Received message correctly

ans =

    'Hello World!'
```

Create a New HDL Coder Project for the Receive FIFO

Run this command at the MATLAB command prompt to create a new HDL Coder project for the receive FIFO:

```
coder -hdlcoder -new mlhdlc_rx_fifo
```

When the **HDL Code Generation** pane opens, set the `mlhdlc_rx_fifo.m` function as the MATLAB function to generate HDL code for. Set the `mlhdlc_rx_fifo_tb.m` script as the MATLAB test bench. Click **Workflow Advisor**.

Refer to "Get Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder™ projects.

Generate HDL Code for the Receive FIFO

Right-click **HDL Code Generation** and select **Run to selected task** to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code for the receive FIFO by clicking the hyperlinks in the bottom pane.

Create a New HDL Coder Project for the Transmit FIFO

```
coder -hdlcoder -new mlhdlc_tx_fifo
```

When the **HDL Code Generation** pane opens, specify `mlhdlc_tx_fifo.m` as the MATLAB function to generate HDL code for the transmit FIFO function, and `mlhdlc_tx_fifo_tb.m` as the MATLAB test bench. Click **Workflow Advisor**.

Generate HDL Code for the Transmit FIFO

Right-click **HDL Code Generation** and select **Run to selected task** to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code for the transmit FIFO by clicking on the hyperlinks in the bottom pane.

See Also

Related Examples

- "Persistent Variables and Persistent Array Variables" on page 1-9
- "MATLAB Test Bench Requirements and Best Practices for Code Generation" on page 1-76

HDL Code Generation for Harris Corner Detection Algorithm

This example shows how to generate HDL code from a MATLAB® design that computes the corner metric by using Harris' technique.

Corner Detection Algorithm

A corner is a point in an image where two edges of the image intersect. The corners are robust to image rotation, translation, and illumination. Corners contain important features that you can use in many applications such as restoring image information, image registration, and object tracking.

Corner detection algorithms identify the corners by using a corner metric. This metric corresponds to the likelihood of pixels located at the corner of certain objects. Peaks of corner metric identify the corners. See also Corner Detection (Computer Vision Toolbox) in the Computer Vision Toolbox documentation. The corner detection algorithm:

1. Reads the input image.

```
Image_in = checkerboard(10);
```

2. Finds the corners.

```
cornerDetector = detectHarrisFeatures(Image_in);
```

3. Displays the results.

```
[~,metric] = step(cornerDetector,image_in);
figure;
subplot(1,2,1);
imshow(image_in);
title('Original');
subplot(1,2,2);
imshow(imadjust(metric));
title('Corner metric');
```

Corner Detection MATLAB Design

```
design_name = 'mlhdlc_corner_detection';
testbench_name = 'mlhdlc_corner_detection_tb';
```

Review the MATLAB design:

```
edit(design_name);
```

```
%#codegen
function [valid, ed, xfo, yfo, cm] = mlhdlc_corner_detection(data_in)
%   Copyright 2011-2022 The MathWorks, Inc.

[~, ed, xfo, yfo] = mlhdlc_sobel(data_in);

cm = compute_corner_metric(xfo, yfo);

% compute valid signal
persistent cnt
if isempty(cnt)
```

```
  cnt = 0;
end
cnt = cnt + 1;
valid = cnt > 3*80+3 && cnt <= 80*80+3*80+3;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function bm = compute_corner_metric(gh, gv)

cmh = make_buffer_matrix_gh(gh);
cmv = make_buffer_matrix_gv(gv);
bm = compute_harris_metric(cmh, cmv);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function bm = make_buffer_matrix_gh(gh)

persistent b1 b2 b3 b4;
if isempty(b1)
    b1 = dsp.Delay('Length', 80);
    b2 = dsp.Delay('Length', 80);
    b3 = dsp.Delay('Length', 80);
    b4 = dsp.Delay('Length', 80);
end

b1p = step(b1, gh);
b2p = step(b2, b1p);
b3p = step(b3, b2p);
b4p = step(b4, b3p);

cc = [b4p b3p b2p b1p gh];

persistent h1 h2 h3 h4;
if isempty(h1)
    h1 = dsp.Delay();
    h2 = dsp.Delay();
    h3 = dsp.Delay();
    h4 = dsp.Delay();
end

h1p = step(h1, cc);
h2p = step(h2, h1p);
h3p = step(h3, h2p);
h4p = step(h4, h3p);

bm = [h4p h3p h2p h1p cc];

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function bm = make_buffer_matrix_gv(gv)

persistent b1 b2 b3 b4;
if isempty(b1)
    b1 = dsp.Delay('Length', 80);
    b2 = dsp.Delay('Length', 80);
```

```
    b3 = dsp.Delay('Length', 80);
    b4 = dsp.Delay('Length', 80);
end

b1p = step(b1, gv);
b2p = step(b2, b1p);
b3p = step(b3, b2p);
b4p = step(b4, b3p);

cc = [b4p b3p b2p b1p gv];

persistent h1 h2 h3 h4;
if isempty(h1)
    h1 = dsp.Delay();
    h2 = dsp.Delay();
    h3 = dsp.Delay();
    h4 = dsp.Delay();
end

h1p = step(h1, cc);
h2p = step(h2, h1p);
h3p = step(h3, h2p);
h4p = step(h4, h3p);

bm = [h4p h3p h2p h1p cc];

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function cm = compute_harris_metric(gh, gv)

[g1, g2, g3] = gaussian_filter(gh, gv);
[s1, s2, s3] = reduce_matrix(g1, g2, g3);

cm = (((s1*s3) - (s2*s2)) - (((s1+s3) * (s1+s3)) * 0.04));

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [g1, g2, g3] = gaussian_filter(gh, gv)

%g=fspecial('gaussian',[5 5],1.5);
g = [0.0144    0.0281    0.0351    0.0281    0.0144
     0.0281    0.0547    0.0683    0.0547    0.0281
     0.0351    0.0683    0.0853    0.0683    0.0351
     0.0281    0.0547    0.0683    0.0547    0.0281
     0.0144    0.0281    0.0351    0.0281    0.0144];

g1 = (gh .* gh) .* g(:)';
g2 = (gh .* gv) .* g(:)';
g3 = (gv .* gv) .* g(:)';

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [s1, s2, s3] = reduce_matrix(g1, g2, g3)

s1 = sum(g1);
```

```
s2 = sum(g2);
s3 = sum(g3);

end
```

The MATLAB function is modular and uses several functions to compute the corners of the image. The function:

- `compute_corner_metric` computes the corner metric matrix by instantiating the function `compute_harris_metric`.
- `compute_harris_metric` detects the corner features in the input image by instantiating functions `gaussian_filter` and `reduce_matrix`. The function takes outputs of `make_buffer_matrix_gh` and `make_buffer_matrix_gv` as the inputs.

Corner Detection MATLAB Test Bench

Review the MATLAB test bench:

```
edit(testbench_name);
```

```
clear mlhdlc_corner_detection;
clear mlhdlc_sobel;

%   Copyright 2011-2022 The MathWorks, Inc.

image_in = checkerboard(10);
[image_height, image_width] = size(image_in);

% Pre-allocating y for simulation performance
y_cm = zeros(image_height, image_width);
y_ed = zeros(image_height, image_width);
gradient_hori = zeros(image_height,image_width);
gradient_vert = zeros(image_height,image_width);

dataValidOut = y_cm;

idx_in = 1;
idx_out = 1;
for i=1:image_width+3
    for j=1:image_height+3
        if idx_in <= image_width * image_height
            u = image_in(idx_in);
        else
            u = 0;
        end
        idx_in = idx_in + 1;

        [valid, ed, gh, gv, cm] = mlhdlc_corner_detection(u);

        if valid

            y_cm(idx_out) = cm;
            y_ed(idx_out) = ed;
            gradient_hori(idx_out)   = gh;
```

```
            gradient_vert(idx_out)   = gv;

            idx_out = idx_out + 1;
        end
    end
end

padImage = y_cm;
findLocalMaxima = vision.LocalMaximaFinder('MaximumNumLocalMaxima',100, ...
    'NeighborhoodSize', [11 11], ...
    'Threshold', 0.0005);
Corners = step(findLocalMaxima, padImage);
drawMarkers = vision.MarkerInserter('Size', 2); % Draw circles at corners
ImageCornersMarked = step(drawMarkers, image_in, Corners);

% Display results
% ...
%

nplots = 4;

scrsz = get(0,'ScreenSize');
figure('Name', [mfilename, '_plot'], 'Position',[1 300 700 200])

subplot(1,nplots,1);
imshow(image_in,[min(image_in(:)) max(image_in(:))]);
title('Checker Board')
axis square

subplot(1,nplots,2);
imshow(gradient_hori(3:end,3:end),[min(gradient_hori(:)) max(gradient_hori(:))]);
title(['Vertical',newline,' Gradient'])
axis square

subplot(1,nplots,3);
imshow(gradient_vert(3:end,3:end),[min(gradient_vert(:)) max(gradient_vert(:))]);
title(['Horizontal',newline,' Gradient'])
axis square

% subplot(1,nplots,4);
% imshow(y_ed);
% title('Edges')

subplot(1,nplots,4);
imagesc(ImageCornersMarked)
title('Corners');
axis square
```

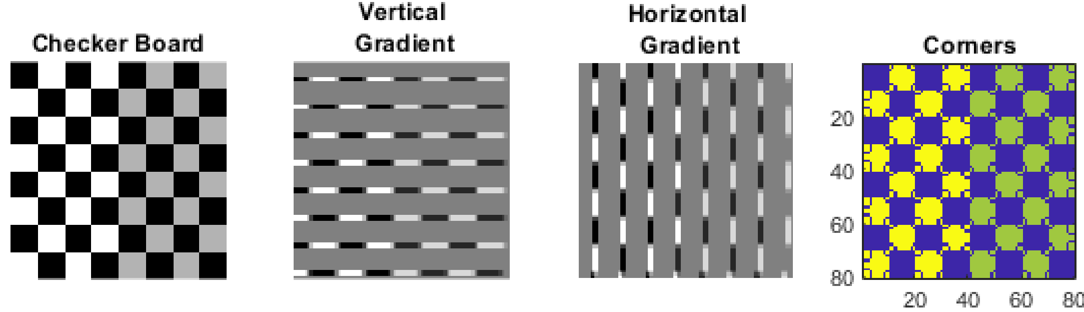
Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

```
mlhdlc_corner_detection_tb
```

```
Warning: vision.LocalMaximaFinder will be removed in a future release. Use the
houghpeaks and imregionalmax functions with equivalent functionality instead.
```

```
Warning: vision.MarkerInserter will be removed in a future release. Use the
insertMarker function with equivalent functionality instead.
```



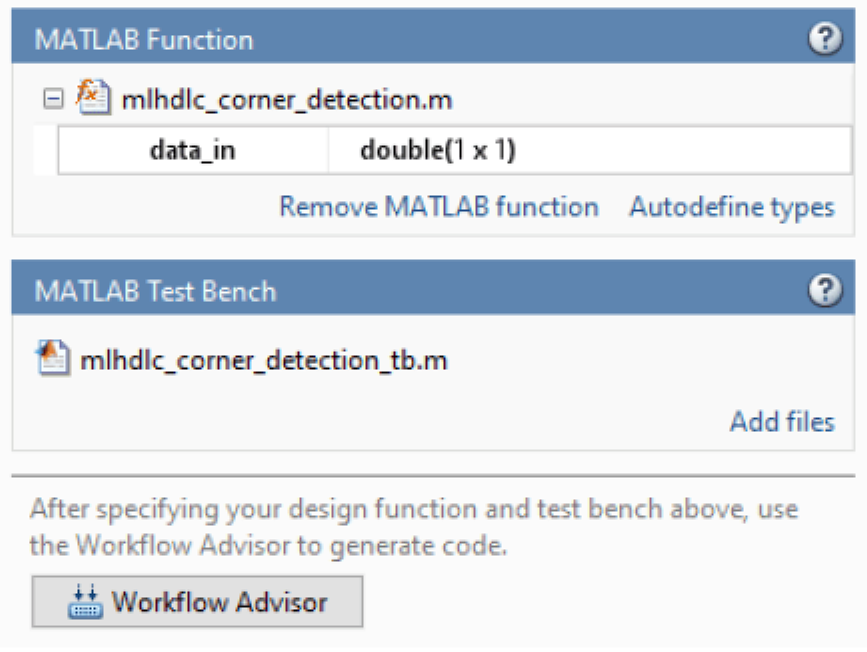
Create an HDL Coder™ Project

1. Create a HDL Coder project:

```
coder -hdlcoder -new mlhdlc_corner_detect_prj
```

2. Add the file `mlhdlc_corner_detection.m` to the project as the **MATLAB Function** and `mlhdlc_corner_detection_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefine types** to use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_corner_detection.m`.



Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Run Fixed-Point Conversion and HDL Code Generation

1. Click the **Workflow Advisor** button to start the Workflow Advisor.
2. Right click the **HDL Code Generation** task and select **Run to selected task**.

A single HDL file `mlhdlc_corner_detection_fixpt.vhd` is generated for the MATLAB design. To examine the generated HDL code for the filter design, click the hyperlinks in the Code Generation Log window.

If you want to generate a HDL file for each function in your MATLAB design, in the **Advanced** tab of the **HDL Code Generation** task, select the **Generate instantiable code for functions** check box. See also "Generate Instantiable Code for Functions" on page 5-12.

HDL Code Generation for Adaptive Median Filter

This example shows how to generate HDL code from a MATLAB® design that implements an adaptive median filter algorithm and generates HDL code.

Adaptive Filter MATLAB Design

An adaptive median filter performs spatial processing to reduce noise in an image. The filter compares each pixel in the image to the surrounding pixels. If one of the pixel values differ significantly from the majority of the surrounding pixels, the pixel is treated as noise. The filtering algorithm then replaces the noise pixel by the median values of the surrounding pixels. This process repeats until all noise pixels in the image are removed.

```
design_name = 'mlhdlc_median_filter';
testbench_name = 'mlhdlc_median_filter_tb';
```

Review the MATLAB design:

```
edit(design_name);
```

```
%#codegen
function [pixel_val, pixel_valid] = mlhdlc_median_filter(c_data, c_idx)
%   Copyright 2011-2019 The MathWorks, Inc.

smax = 9;
persistent window;
if isempty(window)
    window = zeros(smax, smax);
end

cp = ceil(smax/2); % center pixel;

w3 = -1:1;
w5 = -2:2;
w7 = -3:3;
w9 = -4:4;

r3 = cp + w3;      % 3x3 window
r5 = cp + w5;      % 5x5 window
r7 = cp + w7;      % 7x7 window
r9 = cp + w9;      % 9x9 window

d3x3 = window(r3, r3);
d5x5 = window(r5, r5);
d7x7 = window(r7, r7);
d9x9 = window(r9, r9);

center_pixel = window(cp, cp);


% use 1D filter for 3x3 region
outbuf = get_median_1d(d3x3(:)');
[min3, med3, max3] = getMinMaxMed_1d(outbuf);

% use 2D filter for 5x5 region
```

```
outbuf = get_median_2d(d5x5);
[min5, med5, max5] = getMinMaxMed_2d(outbuf);

% use 2D filter for 7x7 region
outbuf = get_median_2d(d7x7);
[min7, med7, max7] = getMinMaxMed_2d(outbuf);

% use 2D filter for 9x9 region
outbuf = get_median_2d(d9x9);
[min9, med9, max9] = getMinMaxMed_2d(outbuf);


pixel_val = get_new_pixel(min3, med3, max3, ...
    min5, med5, max5, ...
    min7, med7, max7, ...
    min9, med9, max9, ...
    center_pixel);


% we need to wait until 9 cycles for the buffer to fill up
% output is not valid every time we start from col1 for 9 cycles.
persistent datavalid
if isempty(datavalid)
    datavalid = false;
end
pixel_valid = datavalid;
datavalid = (c_idx >= smax);


% build the 9x9 buffer
window(:,2:smax) = window(:,1:smax-1);
window(:,1) = c_data;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [min, med, max] = getMinMaxMed_1d(inbuf)

max = inbuf(1);
med = inbuf(ceil(numel(inbuf)/2));
min = inbuf(numel(inbuf));

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [min, med, max] = getMinMaxMed_2d(inbuf)

[nrows, ncols] = size(inbuf);
max = inbuf(1, 1);
med = inbuf(ceil(nrows/2), ceil(ncols/2));
min = inbuf(nrows, ncols);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function new_pixel  = get_new_pixel(...
    min3, med3, max3, ...
    min5, med5, max5, ...
```

```
    min7, med7, max7, ...
    min9, med9, max9, ...
    center_data)

if (med3 > min3 && med3 < max3)
    new_pixel = get_center_data(min3, med3, max3,center_data);
elseif (med5 > min5 && med5 < max5)
    new_pixel = get_center_data(min5, med5, max5,center_data);
elseif (med7 > min7 && med7 < max7)
    new_pixel = get_center_data(min7, med7, max7,center_data);
elseif (med9 > min9 && med9 < max9)
    new_pixel = get_center_data(min9, med9, max9,center_data);
else
    new_pixel = center_data;
end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [new_data] = get_center_data(min,med,max,center_data)
if center_data <= min || center_data >= max
    new_data = med;
else
    new_data = center_data;
end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% perform median 1d computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function outbuf = get_median_1d(inbuf)

numpixels = length(inbuf);

tbuf = inbuf;

for ii=coder.unroll(1:numpixels)
    if bitand(ii,uint32(1)) == 1
        tbuf = compare_stage1(tbuf);
    else
        tbuf = compare_stage2(tbuf);
    end
end

outbuf = tbuf;

end

function outbuf = compare_stage1(inbuf)
numpixels = length(inbuf);
tbuf = compare_stage(inbuf(1:numpixels-1));
outbuf = [tbuf(:)' inbuf(numpixels)];
end

function outbuf = compare_stage2(inbuf)
numpixels = length(inbuf);
tbuf = compare_stage(inbuf(2:numpixels));
outbuf = [inbuf(1) tbuf(:)'];
```

```
end

function [outbuf] = compare_stage(inbuf)

step = 2;
numpixels = length(inbuf);

outbuf = inbuf;

for ii=coder.unroll(1:step:numpixels)
    t = compare_pixels([inbuf(ii), inbuf(ii+1)]);
    outbuf(ii) = t(1);
    outbuf(ii+1) = t(2);
end

end

function outbuf = compare_pixels(inbuf)
if (inbuf(1) > inbuf(2))
    outbuf = [inbuf(1), inbuf(2)];
else
    outbuf = [inbuf(2), inbuf(1)];
end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% perform median 2d computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function outbuf = get_median_2d(inbuf)

outbuf = inbuf;
[nrows, ncols] = size(inbuf);
for ii=coder.unroll(1:ncols)
    colData = outbuf(:, ii)';
    colDataOut = get_median_1d(colData)';
    outbuf(:, ii) = colDataOut;
end
for ii=coder.unroll(1:nrows)
    rowData = outbuf(ii, :);
    rowDataOut = get_median_1d(rowData);
    outbuf(ii, :) = rowDataOut;
end

end
```

The MATLAB function is modular and uses several functions to filter the noise in the image.

Adaptive Filter MATLAB Test Bench

A MATLAB test bench `mlhdlc_median_filter_tb` exercises the filter design by using a representative input range.

Review the MATLAB test bench:

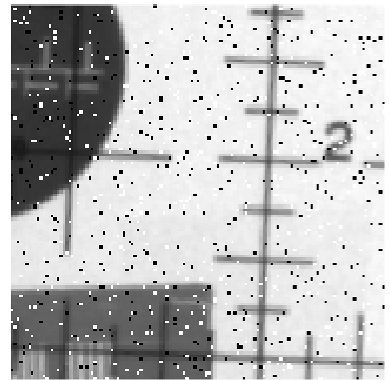
```
edit(testbench_name);
```

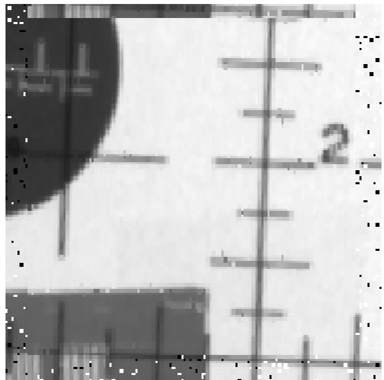
```
I = imread('mlhdlc_img_pattern_noisy.tif');
J = I;

%   Copyright 2011-2019 The MathWorks, Inc.

smax = 9;
[nrows, ncols] = size(I);
ll = ceil(smax/2);
ul = floor(smax/2);

for ii=1:ncols-smax
    for jj=1:nrows-smax

        c_idx = ii;
        c_data = double(I(jj:jj+smax-1, ii));

        [pixel_val, pixel_valid] = mlhdlc_median_filter(c_data, c_idx);

        if pixel_valid
            J(jj, ii) = pixel_val;
        end
    end
end

h = figure;
set( h, 'Name', [ mfilename, '_plot' ] );
subplot( 1, 2, 1 );
imshow( I, [  ] );
subplot( 1, 2, 2 );
imshow( J, [  ] );
```

Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

```
mlhdlc_median_filter_tb
```





Accelerating the Design for Faster Simulation

To simulate the test bench faster:

1. Create a MEX file by using MATLAB Coder™. The HDL Workflow Advisor automates these steps when running fixed-point simulations of the design.

```
codegen -o mlhdlc_median_filter -args {zeros(9,1), 0} mlhdlc_median_filter
[~, tbn] = fileparts(testbench_name);
```

2. Simulate the design by using the MEX file. When you run the test bench, HDL Coder uses the MEX file and runs the simulation faster.

```
mlhdlc_median_filter_tb
```

3. Clean up the MEX file.

```
clear mex;
rmdir('codegen', 's');
delete(['mlhdlc_median_filter', '.', mexext]);
```

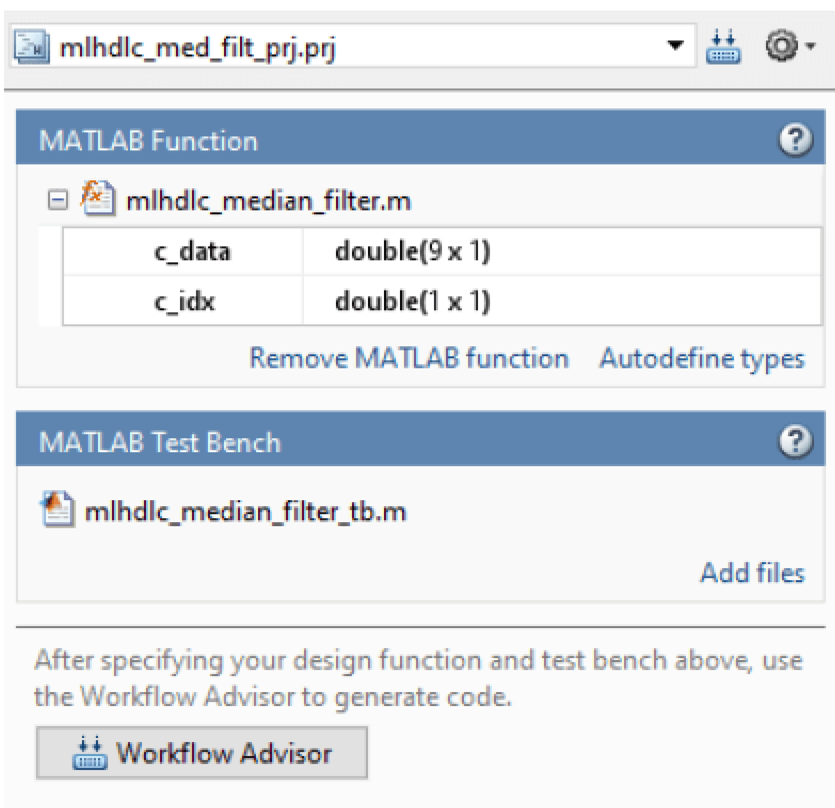
Create an HDL Coder Project

1. Create an HDL Coder project:

```
coder -hdlcoder -new mlhdlc_med_filt_prj
```

2. Add the file `mlhdlc_median_filter.m` to the project as the **MATLAB Function** and `mlhdlc_median_filter_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefine types** and use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_median_filter`.



Refer to "Get Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Run Fixed-Point Conversion and HDL Code Generation

1. Click the **Workflow Advisor** button to start the Workflow Advisor.
2. Right click the **HDL Code Generation** task and select **Run to selected task**.

A single HDL file `mlhdlc_median_filter_fixpt.vhd` is generated for the MATLAB design. To examine the generated HDL code for the filter design, click the hyperlinks in the Code Generation Log window.

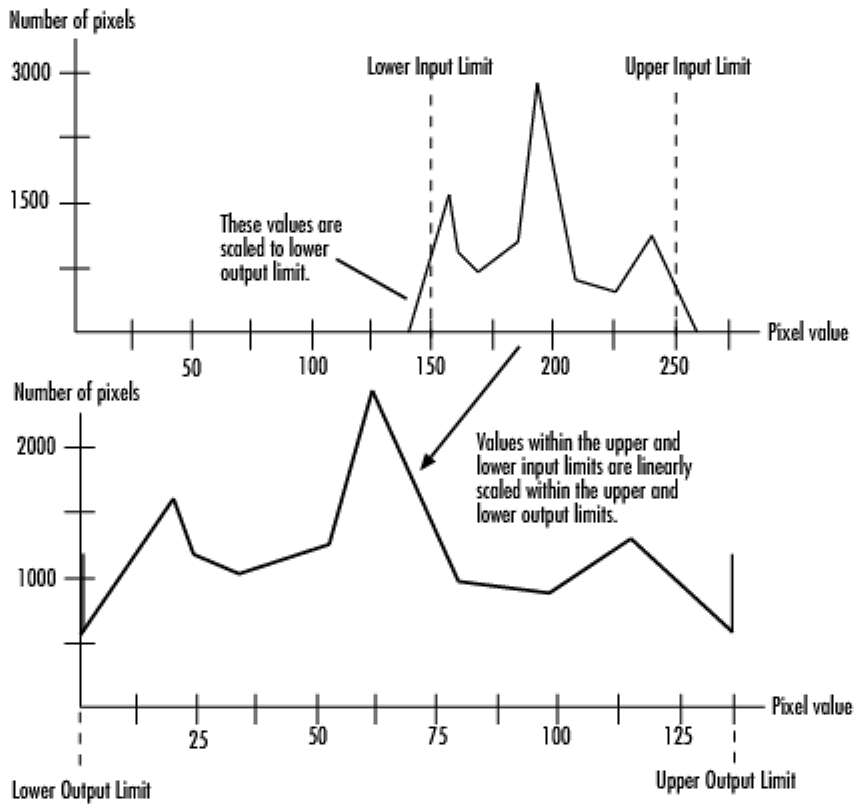
If you want to generate a HDL file for each function in your MATLAB design, in the **Advanced** tab of the **HDL Code Generation** task, select the **Generate instantiable code for functions** check box. See also "Generate Instantiable Code for Functions" on page 5-12.

Contrast Adjustment

This example shows how to generate HDL code from a MATLAB® design that adjusts image contrast by linearly scaling pixel values.

Algorithm

Contrast adjustment adjusts the contrast of an image by linearly scaling the pixel values between upper and lower limits. Pixel values that are above or below this range are saturated to the upper or lower limit value, respectively.



MATLAB Design

```
design_name = 'mlhdlc_image_scale';
testbench_name = 'mlhdlc_image_scale_tb';
```

Look at the MATLAB design:

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% scale.m
%
% Adjust image contrast by linearly scaling pixel values.
%
```

```
% The input pixel value range has 14bits and output pixel value range is
% 8bits.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out, y_out, pixel_out] = ...
    mlhdlc_image_scale(x_in, y_in, pixel_in, ...
          damping_factor_in, dynamic_range_in, ...
          tail_size_in, max_gain_in, ...
          width, height)

%   Copyright 2011-2022 The MathWorks, Inc.

persistent histogram1 histogram2
persistent low_count
persistent high_count
persistent offset
persistent gain
persistent limits_done
persistent damping_done
persistent reset_hist_done
persistent scaling_done
persistent hist_ind
persistent tail_high
persistent min_hist_damped    %Damped lower limit of populated histogram
persistent max_hist_damped    %Damped upper limit of populated histogram
persistent found_high
persistent found_low

DR_PER_BIN          = 8;
SF                  = 1./(1:(2^14/8)); % be nice to fix this
NR_OF_BINS          = (2^14/DR_PER_BIN) - 1;
MAX_DF              = 255;

if isempty(offset)
    offset               = 1;
    gain                 = 1;
    limits_done          = 1;
    damping_done         = 1;
    reset_hist_done      = 1;
    scaling_done         = 1;
    hist_ind             = 1;
    tail_high            = NR_OF_BINS;
    low_count              = 0;
    high_count               = 0;
    min_hist_damped      = 0;
    max_hist_damped      = (2^14/DR_PER_BIN) - 1;
    found_high           = 0;
    found_low            = 0;
end
if isempty(histogram1)
    histogram1           = zeros(1, NR_OF_BINS+1);
    histogram2           = zeros(1, NR_OF_BINS+1);
end

if y_in < height
    frame_valid = 1;
    if x_in < width
        line_valid = 1;
```

```
    else
        line_valid = 0;
    end
else
    frame_valid = 0;
    line_valid = 0;
end

% initialize at beginning of frame
if x_in == 0 && y_in == 0
    limits_done = 0;
    damping_done = 0;
    reset_hist_done = 0;
    scaling_done = 0;
    low_count = 0;
    high_count = 0;
    hist_ind = 1;
end

max_gain_frac = max_gain_in/2^4;
pix11 = floor(pixel_in/DR_PER_BIN);
pix_out_temp = pixel_in;

%**************************************************************************
%Check if valid part of frame. If pixel is valid remap pixel to desired
%output dynamic range (dynamic_range_in) by subtracting the damped offset
%(min_hist_damped) and applying the calculated gain calculated from the
%previous frame histogram statistics.
%**************************************************************************
% histogram read
histReadIndex1 = 1;
histReadIndex2 = 1;
if frame_valid && line_valid
    histReadIndex1 = pix11+1;
    histReadIndex2 = pix11+1;
elseif ~limits_done
    histReadIndex1 = hist_ind;
    histReadIndex2 = NR_OF_BINS - hist_ind;
end
histReadValue1 = histogram1(histReadIndex1);
histReadValue2 = histogram2(histReadIndex2);
histWriteIndex1 = NR_OF_BINS+1;
histWriteIndex2 = NR_OF_BINS+1;
histWriteValue1 = 0;
histWriteValue2 = 0;
if frame_valid
    if line_valid
        temp_sum = histReadValue1 + 1;
        ind = min(pix11+1, NR_OF_BINS);
        val = min(temp_sum, tail_size_in);
        histWriteIndex1 = ind;
        histWriteValue1 = val;
        histWriteIndex2 = ind;
        histWriteValue2 = val;

        %Scale pixel
        pix_out_offs_corr = pixel_in - min_hist_damped*DR_PER_BIN;
        pix_out_scaled = pix_out_offs_corr * gain;
```

```
            pix_out_clamp = max(min(dynamic_range_in, pix_out_scaled), 0);
            pix_out_temp = pix_out_clamp;
        end
    else
        %*********************************************************************
        %Ignore tail_size_in pixels and find lower and upper limits of the
        %histogram.
        %*********************************************************************
        if ~limits_done
            if hist_ind == 1
                tail_high = NR_OF_BINS-1;
                offset = 1;
                found_high = 0;
                found_low = 0;
            end

            low_count = low_count + histReadValue1;
            hist_ind_high = NR_OF_BINS - hist_ind;
            high_count = high_count + histReadValue2;

            %Found enough high outliers
            if high_count > tail_size_in && ~found_high
                tail_high = hist_ind_high;
                found_high = 1;
            end

            %Found enough low outliers
            if low_count > tail_size_in && ~found_low
                offset = hist_ind;
                found_low = 1;
            end

            hist_ind = hist_ind + 1;
            %All bins checked so limits must already be found
            if hist_ind >= NR_OF_BINS
                hist_ind = 1;
                limits_done = 1;
            end
            %*********************************************************************
            %Damp the limit change to avoid image flickering. Code below equivalent
            %to: max_hist_damped = damping_factor_in*max_hist_dampedOld +
            %(1-damping_factor_in)*max_hist_dampedNew;
            %*********************************************************************
        elseif ~damping_done
            min_hist_weighted_old = damping_factor_in*min_hist_damped;
            min_hist_weighted_new = (MAX_DF-damping_factor_in+1)*offset;
            min_hist_weighted = (min_hist_weighted_old + ...
                min_hist_weighted_new)/256;
            min_hist_damped = max(0, min_hist_weighted);
            max_hist_weighted_old = damping_factor_in*max_hist_damped;
            max_hist_weighted_new = (MAX_DF-damping_factor_in+1)*tail_high;
            max_hist_weighted = (max_hist_weighted_old + ...
                max_hist_weighted_new)/256;
            max_hist_damped = min(NR_OF_BINS, max_hist_weighted);
            damping_done = 1;
            hist_ind = 1;
            %*********************************************************************
            %Reset all bins to zero. More than one bin can be reset per function
```

```
        %call if blanking time is too short.
        %*******************************************************************
    elseif ~reset_hist_done
        histWriteIndex1 = hist_ind;
        histWriteValue1 = 0;
        histWriteIndex2 = hist_ind;
        histWriteValue2 = 0;
        hist_ind = hist_ind+1;
        if hist_ind == NR_OF_BINS
            reset_hist_done = 1;
        end
        %*******************************************************************
        %The gain factor is determined by comparing the measured damped actual
        %dynamic range to the desired user specified dynamic range. Input
        %dynamic range is measured in bins over DR_PER_BIN space.
        %*******************************************************************
    elseif ~scaling_done
        dr_in = round(max_hist_damped - min_hist_damped);
        gain_temp = dynamic_range_in*SF(dr_in);
        gain_scaled = gain_temp/DR_PER_BIN;
        gain = min(max_gain_frac, gain_scaled);
        scaling_done = 1;
        hist_ind = 1;
    end
end
histogram1(histWriteIndex1) = histWriteValue1;
histogram2(histWriteIndex2) = histWriteValue2;

x_out = x_in;
y_out = y_in;
pixel_out = pix_out_temp;
```

```
type(testbench_name);
```

```
%Test bench for scaling, analogous to automatic gain control (AGC)

%   Copyright 2011-2022 The MathWorks, Inc.

testFile = 'mlhdlc_img_peppers.png';
imgOrig = imread(testFile);
[height, width] = size(imgOrig);
imgOut = zeros(height,width);
hBlank = 20;
% make sure we have enough vertical blanking to filter the histogram
vBlank = ceil(2^14/(width+hBlank));

%df - Temporal damping factor of rescaling
%dr - Desired output dynamic range
df = 0;
dr = 255;
nrOfOutliers = 248;
maxGain = 2*2^4;

for frame = 1:2
    disp(['frame: ', num2str(frame)]);
    for y_in = 0:height+vBlank-1
        %disp(['frame: ', num2str(frame), ' of 2, row: ', num2str(y_in)]);
```

```
        for x_in = 0:width+hBlank-1
            if x_in < width && y_in < height
                pixel_in = double(imgOrig(y_in+1, x_in+1));
            else
                pixel_in = 0;
            end

            [x_out, y_out, pixel_out] = ...
                mlhdlc_image_scale(x_in, y_in, pixel_in, df, dr, ...
                    nrOfOutliers, maxGain, width, height);

            if x_out < width && y_out < height
                imgOut(y_out+1,x_out+1) = pixel_out;
            end
        end
    end

    figure('Name', [mfilename, '_scale_plot']);
    imgOut = round(255*imgOut/max(max(imgOut)));
    subplot(2,2,1); imshow(imgOrig, []);
    title('Original Image');
    subplot(2,2,2); imshow(imgOut, []);
    title('Scaled Image');
    subplot(2,2,3); histogram(double(imgOrig(:)),2^14-1);
    axis([0, 255, 0, 1500]);
    title('Histogram of original Image');
    subplot(2,2,4); histogram(double(imgOut(:)),2^14-1);
    axis([0, 255, 0, 1500]);
    title('Histogram of equalized Image');
end
```

Simulate the Design

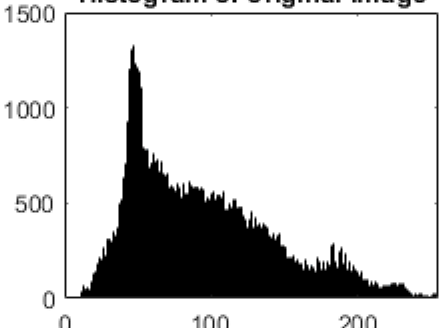
It is a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_image_scale_tb
```

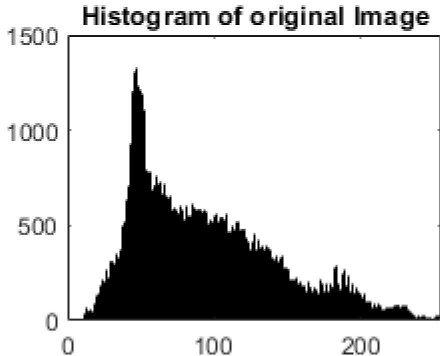
```
frame: 1
frame: 2
```

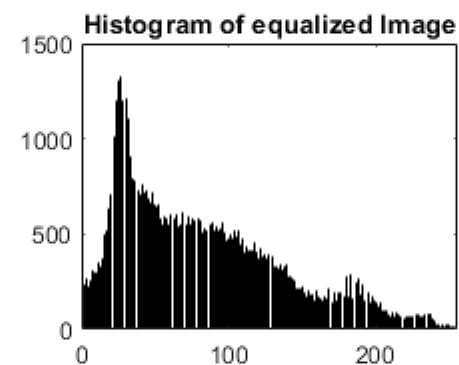
Original Image

Scaled Image


Histogram of original Image
1500
1000
500
0
0
100
200

Histogram of equalized Image
1500
1000
500
0
0
100
200





Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_scale_prj
```

Next, add the file `mlhdlc_image_scale.m` to the project as the MATLAB Function and `mlhdlc_image_scale_tb.m` as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor from the Workflow Advisor button and right-click on the **Code Generation** step and choose the option **Run to selected task** to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

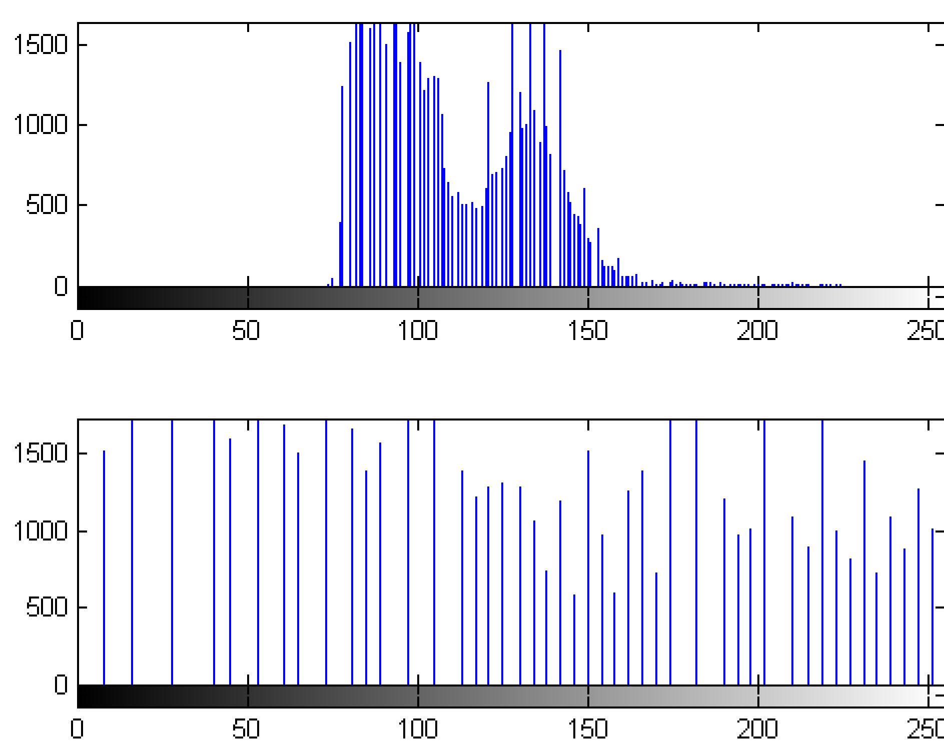
Image Enhancement by Histogram Equalization

This example shows how to generate HDL code from a MATLAB® design that does image enhancement using histogram equalization.

Algorithm

The Histogram Equalization algorithm enhances the contrast of images by transforming the values in an intensity image so that the histogram of the output image is approximately flat.

```
I = imread('pout.tif');
J = histeq(I);
subplot(2,2,1);
imshow( I );
subplot(2,2,2);
imhist(I)
subplot(2,2,3);
imshow( J );
subplot(2,2,4);
imhist(J)
```



MATLAB Design

```
design_name = 'mlhdlc_heq';
testbench_name = 'mlhdlc_heq_tb';
```

Let us take a look at the MATLAB design

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% heq.m
% Histogram Equalization Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out, y_out, pixel_out] = ...
    mlhdlc_heq(x_in, y_in, pixel_in, width, height)

%   Copyright 2011-2015 The MathWorks, Inc.

persistent histogram
persistent transferFunc
persistent histInd
persistent cumSum

if isempty(histogram)
    histogram = zeros(1, 2^14);
    transferFunc = zeros(1, 2^14);
    histInd = 0;
    cumSum = 0;
end

% Figure out indexes based on where we are in the frame
if y_in < height && x_in < width % valid pixel data
    histInd = pixel_in + 1;
elseif y_in == height && x_in == 0 % first column of height+1
    histInd = 1;
elseif y_in >= height % vertical blanking period
    histInd = min(histInd + 1, 2^14);
elseif y_in < height % horizontal blanking - do nothing
    histInd = 1;
end

%Read histogram (must be outside conditional logic)
histValRead = histogram(histInd);

%Read transfer function (must be outside conditional logic)
transValRead = transferFunc(histInd);

%If valid part of frame add one to pixel bin and keep transfer func val
if y_in < height && x_in < width
    histValWrite = histValRead + 1; %Add pixel to bin
    transValWrite = transValRead; %Write back same value
    cumSum = 0;
elseif y_in >= height %In blanking time index through all bins and reset to zero
    histValWrite = 0;
    transValWrite = cumSum + histValRead;
    cumSum = transValWrite;
else
    histValWrite = histValRead;
    transValWrite = transValRead;
end

%Write histogram (must be outside conditional logic)
```

```
histogram(histInd) = histValWrite;

%Write transfer function (must be outside conditional logic)
transferFunc(histInd) = transValWrite;

pixel_out = transValRead;
x_out = x_in;
y_out = y_in;
```

```
type(testbench_name);
```

```
%Test bench for Histogram Equalization

%   Copyright 2011-2018 The MathWorks, Inc.

testFile = 'mlhdlc_img_peppers.png';
imgOrig = imread(testFile);
[height, width] = size(imgOrig);
imgOut = zeros(height,width);
hBlank = 20;
% make sure we have enough vertical blanking to filter the histogram
vBlank = ceil(2^14/(width+hBlank));

for frame = 1:2
    disp(['working on frame: ', num2str(frame)]);
    for y_in = 0:height+vBlank-1
        %disp(['frame: ', num2str(frame), ' of 2, row: ', num2str(y_in)]);
        for x_in = 0:width+hBlank-1
            if x_in < width && y_in < height
                pixel_in = double(imgOrig(y_in+1, x_in+1));
            else
                pixel_in = 0;
            end

            [x_out, y_out, pixel_out] = ...
                mlhdlc_heq(x_in, y_in, pixel_in, width, height);

            if x_out < width && y_out < height
                imgOut(y_out+1,x_out+1) = pixel_out;
            end
        end
    end

    % normalize image to 255
    imgOut = round(255*imgOut/max(max(imgOut)));

    figure(1)
    subplot(2,2,1); imshow(imgOrig, [0,255]);
    title('Original Image');
    subplot(2,2,2); imshow(imgOut, [0,255]);
    title('Equalized Image');
    subplot(2,2,3); histogram(double(imgOrig(:)),2^14-1);
    axis([0, 255, 0, 1500])
    title('Histogram of original Image');
    subplot(2,2,4); histogram(double(imgOut(:)),2^14-1);
    axis([0, 255, 0, 1500])
```

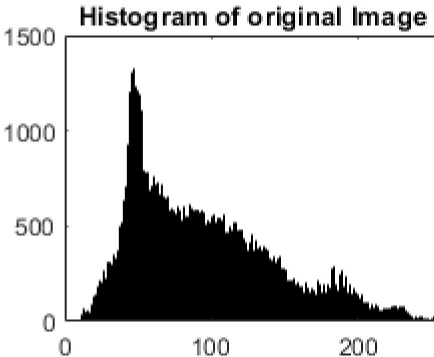
```
    title('Histogram of equalized Image');
end
```

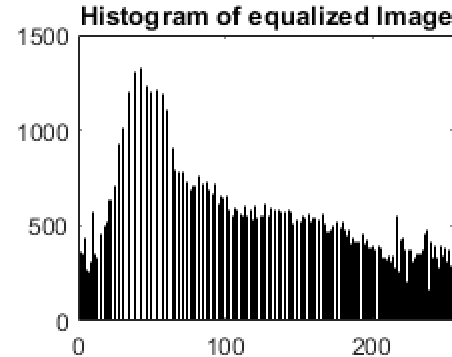
Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_heq_tb
```

```
working on frame: 1
working on frame: 2
```





Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_heq_prj
```

Next, add the file 'mlhdlc_heq.m' to the project as the MATLAB Function and 'mlhdlc_heq_tb.m' as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Run Fixed-Point Conversion and HDL Code Generation

Launch HDL Advisor and right click on the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

HDL Code Generation for Image Format Conversion from RGB to YUV

This example shows how to generate HDL code from a MATLAB® design that converts the image format from RGB to YUV.

MATLAB Design and Test Bench

```
design_name = 'mlhdlc_rgb2yuv';
testbench_name = 'mlhdlc_rgb2yuv_tb';
```

Review the MATLAB design:

```
open(design_name)
```

```
function [x_out, y_out, y_data_out, u_data_out, v_data_out] = ...
                        mlhdlc_rgb2yuv(x_in, y_in, r_in, g_in, b_in)
%#codegen

%   Copyright 2011-2019 The MathWorks, Inc.

persistent RGB_Reg YUV_Reg
persistent x1 x2 y1 y2

if isempty(RGB_Reg)
    RGB_Reg = zeros(3,1);
    YUV_Reg = zeros(3,1);
    x1 = 0; x2 = 0; y1 = 0; y2 = 0;
end

D = [.299  .587  .144; -.147 -.289  .436; .615 -.515 -.1];
C = [0; 128; 128];

RGB = [r_in; g_in; b_in];

YUV_1 = D*RGB_Reg;
YUV_2 = YUV_1 + C;
RGB_Reg = RGB;

y_data_out = round(YUV_Reg(1));
u_data_out = round(YUV_Reg(2));
v_data_out = round(YUV_Reg(3));
YUV_Reg = YUV_2;

x_out = x2; x2 = x1; x1 = x_in;
y_out = y2; y2 = y1; y1 = y_in;
```

Review the MATLAB test bench:

```
open(testbench_name);
```

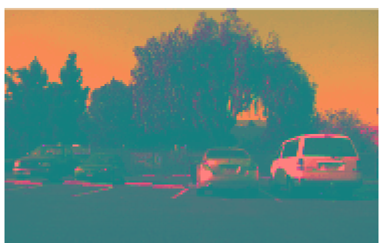
```
FRAMES = 1;
```

```
WIDTH = 752;
HEIGHT = 480;
HBLANK = 10;%748;
VBLANK = 10;%120;

%   Copyright 2011-2019 The MathWorks, Inc.

vidData = double(imread('mlhdlc_img_yuv.png'));

for f = 1:FRAMES
    vidOut = zeros(HEIGHT, WIDTH, 3);

    for y = 0:HEIGHT+VBLANK-1
        for x = 0:WIDTH+HBLANK-1
            if y >= 0 && y < HEIGHT && x >= 0 && x < WIDTH
                b = vidData(y+1,x+1,1);
                g = vidData(y+1,x+1,2);
                r = vidData(y+1,x+1,3);
            else
                b = 0; g = 0; r = 0;
            end

            [xOut, yOut, yData, uData, vData] = ...
                mlhdlc_rgb2yuv(x, y, r, g, b);

            if yOut >= 0 && yOut < HEIGHT && xOut >= 0 && xOut < WIDTH
                vidOut(yOut+1,xOut+1,:) = [yData vData uData];
            end
        end
    end

    figure(1);
    subplot(1,2,1);
    imshow(uint8(vidData));
    subplot(1,2,2);
    imshow(ycbcr2rgb(uint8(vidOut)));
    drawnow;
end
```

Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

```
mlhdlc_rgb2yuv_tb
```



Create an HDL Coder™ Project

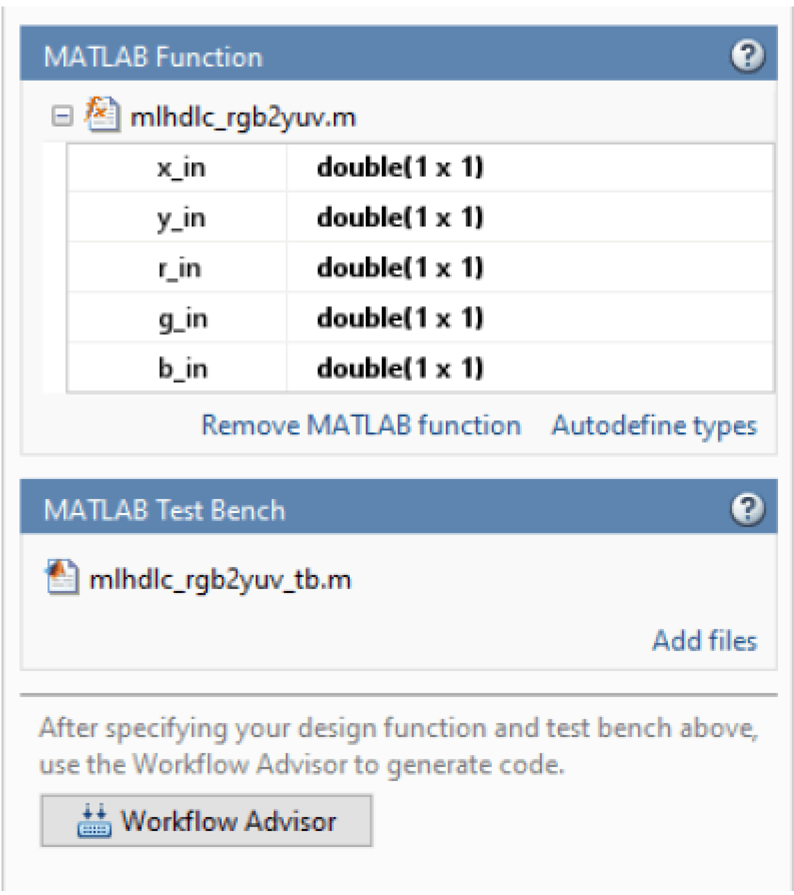
To generate HDL code from a MATLAB design:

1. Create a HDL Coder project:

```
coder -hdlcoder -new mlhdlc_rgb_prj
```

2. Add the file `mlhdlc_rgb2yuv.m` to the project as the **MATLAB Function** and `mlhdlc_rgb2yuv_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefine types** to use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_rgb2yuv`.



Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Run Fixed-Point Conversion and HDL Code Generation

1. Click the **Workflow Advisor** button to start the Workflow Advisor.
2. Right click the **HDL Code Generation** task and select **Run to selected task**.

A HDL file `mlhdlc_rgb2yuv_fixpt.vhd` is generated for the MATLAB design. To examine the generated HDL code for the filter design, click the hyperlink to the HDL file in the Code Generation Log window.

High Dynamic Range Imaging

This example shows how to generate HDL code from a MATLAB® design that implements a high dynamic range imaging algorithm.

Algorithm

High Dynamic Range Imaging (HDRI or HDR) is a set of methods used in imaging and photography to allow a greater dynamic range between the lightest and darkest areas of an image than current standard digital imaging methods or photographic methods. HDR images can represent more accurately the range of intensity levels found in real scenes, from direct sunlight to faint starlight, and is often captured by way of a plurality of differently exposed pictures of the same subject matter.

MATLAB Design

```
design_name = 'mlhdlc_hdr';
testbench_name = 'mlhdlc_hdr_tb';
```

Use the `dbtype` function to display the contents of the MATLAB design.

```
dbtype(design_name);
```

```
1     function [valid_out, x_out, y_out, ...
2         HDR1, HDR2, HDR3] = mlhdlc_hdr(YShort1, YShort2, YShort3, ...
3         YLong1, YLong2, YLong3, ...
4         plot_y_short_in, plot_y_long_in, ...
5         valid_in, x, y)
6     %
7
8     %   Copyright 2013-2015 The MathWorks, Inc.
9
10    % This design implements a high dynamic range imaging algorithm.
11
12    plot_y_short = plot_y_short_in;
13    plot_y_long = plot_y_long_in;
14
15    %% Apply Lum(Y) channels LUTs
16    y_short = plot_y_short(uint8(YShort1)+1);
17    y_long = plot_y_long(uint8(YLong1)+1);
18
19    y_HDR = (y_short+y_long);
20
21    %% Create HDR Chorm channels
22    % HDR per color
23
24    HDR1 = y_HDR * 2^-8;
25    HDR2 = (YShort2+YLong2) * 2^-1;
26    HDR3 = (YShort3+YLong3) * 2^-1;
27
28    %% Pass on valid signal and pixel location
29
30    valid_out = valid_in;
31    x_out = x;
32    y_out = y;
33
34    end
```

```
dbtype(testbench_name);
```

```
1
2     %
3
4     %   Copyright 2013-2015 The MathWorks, Inc.
5
6     % Clean screen and memory
7     close all
8     clear mlhdlc_hdr
9     set(0,'DefaultFigureWindowStyle','docked')
10
11
12    %% Read the two exposed images
13
14    short = imread('mlhdlc_hdr_short.tif');
15    long = imread('mlhdlc_hdr_long.tif');
16
17    % define HDR output variable
18    HDR = zeros(size(short));
19    [height, width, color] = size(HDR);
20
21    set(0,'DefaultFigureWindowStyle' , 'normal')
22    figure('Name', [mfilename, '_plot']);
23    subplot(1,3,1);
24    imshow(short, 'InitialMagnification','fit'), title('short');
25
26    subplot(1,3,2);
27    imshow(long, 'InitialMagnification','fit'), title('long');
28
29
30    %% Create the Lum(Y) channels LUTs
31    % Pre-process
32    % Luminance short LUT
33    ShortLut.x = [0    16    45    96   255];
34    ShortLut.y = [0    20    38    58   115];
35
36    % Luminance long LUT
37    LongLut.x = [ 0 255];
38    LongLut.y = [ 0  140];
39
40    % Take the same points to plot the joined Lum LUT
41    plot_x = 0:1:255;
42    plot_y_short = interp1(ShortLut.x,ShortLut.y,plot_x); %LUT short
43    plot_y_long = interp1(LongLut.x,LongLut.y,plot_x); %LUT long
44
45    %subplot(4,1,3);
46    %plot(plot_x, plot_y_short, plot_x, plot_y_long, plot_x, (plot_y_long+plot_y_short)), grid
47
48
49    %% Create the HDR Lum channel
50    % The HDR algorithm
51    % read the Y channels
52
53    YIQ_short = rgb2ntsc(short);
54    YIQ_long = rgb2ntsc(long);
55
```

```
56    %% Stream image through HDR algorithm
57
58    for x=1:width
59        for y=1:height
60            YShort1 = round(YIQ_short(y,x,1)*255); %input short
61            YLong1 = round(YIQ_long(y,x,1)*255); %input long
62
63            YShort2 = YIQ_short(y,x,2); %input short
64            YLong2 = YIQ_long(y,x,2); %input long
65
66            YShort3 = YIQ_short(y,x,3); %input short
67            YLong3 = YIQ_long(y,x,3); %input long
68
69            valid_in = 1;
70
71            [valid_out, x_out, y_out, HDR1, HDR2, HDR3] = mlhdlc_hdr(YShort1, YShort2, YShort3
72
73            % use x and y to reconstruct image
74            if valid_out == 1
75                HDR(y_out,x_out,1) = HDR1;
76                HDR(y_out,x_out,2) = HDR2;
77                HDR(y_out,x_out,3) = HDR3;
78            end
79        end
80    end
81
82    %% plot HDR
83    HDR_rgb = ntsc2rgb(HDR);
84    subplot(1,3,3);
85    imshow(HDR_rgb, 'InitialMagnification','fit'), title('hdr ');
```

Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_hdr_tb
```

short

long

hdr

Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_hdr_prj
```

Next, add the file 'mlhdlc_hdr.m' to the project as the MATLAB Function and 'mlhdlc_hdr_tb.m' as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Creating Constant Parameter Inputs

This example shows to use pass constant parameter inputs.

In this design the input parameters 'plot_y_short_in' and 'plot_y_long_in' are constant input parameters. You can define them accordingly by modifying the input types as 'constant(double(1x256))'

'plot_y_short_in' and 'plot_y_short_in' are LUT inputs. They are constant folded as double inputs to the design. You will not see port declarations for these two input parameters in the generated HDL code.

Note that inside the design 'mlhdlc_hdr.m' these variables are reassigned so that they get properly fixed-point converted. This is not necessary if these are purely used as constants for defining sizes of variables for example and not part of the logic.

Run Fixed-Point Conversion and HDL Code Generation

Launch HDL Advisor and right click on the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

Convert the Design to Fixed-Point and Generate HDL Code

The following script converts the design to fixed-point, and generate HDL code with a test bench.

```
exArgs = {0,0,0,0,0,0,coder.Constant(ones(1,256)),coder.Constant(ones(1,256)),0,0,0};
fc = coder.config('fixpt');
fc.TestBenchName = 'mlhdlc_hdr_tb';
hc = coder.config('hdl');
hc.GenerateHDLTestBench = true;
hc.SimulationIterationLimit = 1000; % Limit number of testbench points
codegen -float2fixed fc -config hc -args exArgs mlhdlc_hdr
```

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

Accelerate Pixel-Streaming Designs Using MATLAB Coder

This example shows how to accelerate a pixel-stream video processing algorithm in MATLAB® by using MATLAB Coder™.

You must have a MATLAB Coder license to run this example.

Acceleration with MATLAB Coder enables you to simulate large frame sizes, such as 1080p video, at practical speeds. Use this acceleration workflow after you have debugged the algorithm using a small frame size. Testing a design with a small image is demonstrated in the “Pixel-Streaming Design in MATLAB” (Vision HDL Toolbox) example.

How MATLAB Coder Works

MATLAB Coder generates C code from MATLAB code. Code generation accelerates simulation by locking-down the sizes and data types of variables. This process removes the overhead of the interpreted language checking for size and data type in every line of code. This example compiles both the test bench file `DesignAccelerationHDLTestBench.m` and the design file `DesignAccelerationHDLDesign.m` into a MEX function, and uses the resulting MEX file to speed up the simulation.

The directive (or pragma) **%#codegen** beneath the function signature indicates that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB code analyzer to help you diagnose and fix violations that would result in errors during code generation. The directive **%#codegen** does not affect interpreted simulation.

Best Practices

Debugging simulations with large frame sizes is impractical in interpreted mode due to long simulation time. However, debugging a MEX simulation is challenging due to lack of debug access into the code.

To avoid these scenarios, a best practice is to develop and verify the algorithm and test bench using a thumbnail frame size. In most cases, the HDL-targeted design can be implemented with no dependence on frame size. Once you are confident that the design and test bench are working correctly, then increase the frame size in the test bench, and use MATLAB Coder to accelerate the simulation. To increase the frame size, test bench only requires minor changes, as you can see by comparing `DesignAccelerationHDLTestBench.m` with the `PixelStreamingDesignHDLTestBench.m` in “Pixel-Streaming Design in MATLAB” (Vision HDL Toolbox).

Test Bench

In the test bench `DesignAccelerationHDLTestBench.m`, the **videoIn** object reads each frame from a video source which is converted from RGB to grayscale, and the **imresize** function interpolates this frame from 240p to 1080p. This 1080p image is passed to the **frm2pix** object, which converts the full image frame to a stream of pixels and control structures. The function `DesignAccelerationHDLDesign` is then called to process one pixel (and its associated control structure) at a time. After we process the entire pixel-stream and collect the output stream, the **pix2frm** object converts the output stream to full-frame video. The **DesignAccelerationHDLViewer** function displays the output and original images side-by-side.

The workflow above is implemented in the following lines of `DesignAccelerationHDLTestBench.m`.

```
for f = 1:numFrm
    frmFull = rgb2gray(readFrame(videoIn));            % Get a new frame
    frmIn = imresize(frmFull,[actLine actPixPerLine]); % Enlarge the frame

     [pixInVec,ctrlInVec] = frm2pix(frmIn);
     for p = 1:numPixPerFrm
         [pixOutVec(p),ctrlOutVec(p)] = DesignAccelerationHDLDesign(pixInVec(p),ctrlInVec(p));
     end
     frmOut = pix2frm(pixOutVec,ctrlOutVec);

     DesignAccelerationHDLViewer(actPixPerLine,actLine,[frmIn uint8(255*frmOut)]);
 end
```

The data type of frmIn is uint8 while that of frmOut, the edge detection output, is logical. Matrices of different data types cannot be concatenated, so **uint8(255*frmOut)** maps logical false and true to uint8(0) and uint8(255), respectively.

Both **frm2pix** and **pix2frm** are used to convert between full-frame and pixel-stream domains. The inner for-loop performs pixel-stream processing. The rest of the test bench performs full-frame processing (i.e., **videoIn**, **scaler**, and **viewer** inside the **DesignAccelerationHDLViewer** function).

Before the test bench terminates, frame rate is displayed to illustrate the simulation speed.

Not all functions used in the test bench support C code generation. For those that do not, such as `tic`, `toc`, `fprintf`, use **coder.extrinsic** to declare them as extrinsic functions. Extrinsic functions are excluded from MEX generation. The simulation executes them in the regular interpreted mode.

Pixel-Stream Design

The function defined in `DesignAccelerationHDLDesign.m` accepts a pixel stream and five control signals, and returns a modified pixel stream and control signals. For more information on the streaming pixel protocol used by System objects from the Vision HDL Toolbox™, see “Streaming Pixel Interface” (Vision HDL Toolbox).

In this example, the function contains the Edge Detector System object.

The focus of this example is the workflow, not the algorithm design itself. Therefore, the design code is quite simple. Once you are familiar with the workflow, it is straightforward to implement advanced video algorithms by taking advantage of the functionality provided by the System objects from Vision HDL Toolbox.

Create MEX File and Simulate the Design

Generate and execute the MEX file.

```
codegen('DesignAccelerationHDLTestBench');
DesignAccelerationHDLTestBench_mex;
```

```
Code generation successful.


10 frames have been processed in 10.56 seconds.
Average frame rate is 0.95 frames/second.
```

The **viewer** displays the original video on the left, and the output on the right.

HDL Code Generation

Enter the following command to create a new HDL Coder™ project in the temporary folder

```
coder -hdlcoder -new DesignAccelerationProject
```

Then, add the file `DesignAccelerationHDLDesign.m` to the project as the MATLAB Function and `DesignAccelerationHDLTestBench.m` as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a tutorial on creating and populating MATLAB HDL Coder projects.

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

Enhanced Edge Detection from Noisy Color Video

This example shows how to develop a complex pixel-stream video processing algorithm, accelerate its simulation using MATLAB® Coder™, and generate HDL code from the design. The algorithm enhances the edge detection from noisy color video.

You must have a MATLAB Coder license to run this example.

This example builds on the “Pixel-Streaming Design in MATLAB” (Vision HDL Toolbox) and the “Accelerate Pixel-Streaming Designs Using MATLAB Coder” (Vision HDL Toolbox) examples.

Test Bench

In the `EnhancedEdgeDetectionHDLTestBench.m` file, the **videoIn** object reads each frame from a color video source, and the `imnoise` function adds salt and pepper noise. This noisy color image is passed to the **frm2pix** object, which converts the full image frame to a stream of pixels and control structures. The function `EnhancedEdgeDetectionHDLDesign.m` is then called to process one pixel (and its associated control structure) at a time. After we process the entire pixel-stream and collect the output stream, the **pix2frm** object converts the output stream to full-frame video. A full-frame reference design `EnhancedEdgeDetectionHDLReference.m` is also called to process the noisy color image. Its output is compared with that of the pixel-stream design. The function `EnhancedEdgeDetectionHDLViewer.m` is called to display video outputs.

The workflow above is implemented in the following lines of `EnhancedEdgeDetectionHDLTestBench.m`.

```
...
frmIn = zeros(actLine,actPixPerLine,3,'uint8');
for f = 1:numFrm
    frmFull = readFrame(videoIn);             % Get a new frame
    frmIn = imnoise(frmFull,'salt & pepper'); % Add noise

     % Call the pixel-stream design
     [pixInVec,ctrlInVec] = frm2pix(frmIn);
     for p = 1:numPixPerFrm
         [pixOutVec(p),ctrlOutVec(p)] = EnhancedEdgeDetectionHDLDesign(pixInVec(p,:),ctrlInVec(p
     end
     frmOut = pix2frm(pixOutVec,ctrlOutVec);

     % Call the full-frame reference design
     [frmGray,frmDenoise,frmEdge,frmRef] = EnhancedEdgeDetectionHDLReference(frmIn);

     % Compare the results
     if nnz(imabsdiff(frmRef,frmOut))>20
         fprintf('frame %d: reference and design output differ in more than 20 pixels.\n',f);
         return;
     end

     % Display the results
     EnhancedEdgeDetectionHDLViewer(actPixPerLine,actLine,[frmGray frmDenoise uint8(255*[frmEdge
 end
 ...
```

Since frmGray and frmDenoise are uint8 data type while frmEdge and frmOut are logical, **uint8(255x[frmEdge frmOut])** maps logical false and true to uint8(0) and uint8(255), respectively, so that matrices can be concatenated.

Both **frm2pix** and **pix2frm** are used to convert between full-frame and pixel-stream domains. The inner for-loop performs pixel-stream processing. The rest of the test bench performs full-frame processing.

Before the test bench terminates, frame rate is displayed to illustrate the simulation speed.

For the functions that do not support C code generation, such as `tic`, `toc`, `imnoise`, and `fprintf` in this example, use **coder.extrinsic** to declare them as extrinsic functions. Extrinsic functions are excluded from MEX generation. The simulation executes them in the regular interpreted mode. Since `imnoise` is not included in the C code generation process, the compiler cannot infer the data type and size of frmIn. To fill in this missing piece, we add the statement **frmIn = zeros(actLine,actPixPerLine,3,'uint8')** before the outer for-loop.

Pixel-Stream Design

The function defined in `EnhancedEdgeDetectionHDLDesign.m` accepts a pixel stream and a structure consisting of five control signals, and returns a modified pixel stream and control structure. For more information on the streaming pixel protocol used by System objects from the Vision HDL Toolbox, see the “Streaming Pixel Interface” (Vision HDL Toolbox).

In this example, the **rgb2gray** object converts a color image to grayscale, **medfil** removes the salt and pepper noise. **sobel** highlights the edge. Finally, the **mclose** object performs morphological closing to enhance the edge output. The code is shown below.

```
[pixGray,ctrlGray] = rgb2gray(pixIn,ctrlIn);         % Convert RGB to grayscale
[pixDenoise,ctrlDenoise] = medfil(pixGray,ctrlGray); % Remove noise
[pixEdge,ctrlEdge] = sobel(pixDenoise,ctrlDenoise);  % Detect edges
[pixClose,ctrlClose] = mclose(pixEdge,ctrlEdge);     % Apply closing
```

Full-Frame Reference Design

When designing a complex pixel-stream video processing algorithm, it is a good practice to develop a parallel reference design using functions from the Image Processing Toolbox™. These functions process full image frames. Such a reference design helps verify the implementation of the pixel-stream design by comparing the output image from the full-frame reference design to the output of the pixel-stream design.

The function `EnhancedEdgeDetectionHDLReference.m` contains a similar set of four functions as in the `EnhancedEdgeDetectionHDLDesign.m`. The key difference is that the functions from Image Processing Toolbox process full-frame data.

Due to the implementation difference between `edge` function and `visionhdl.EdgeDetector` System object, reference and design output are considered matching if frmOut and frmRef differ in no greater than 20 pixels.

Create MEX File and Simulate the Design

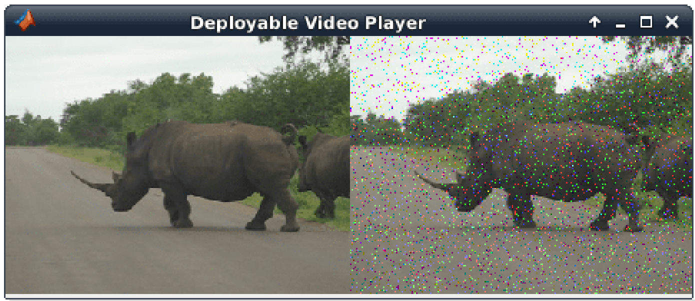
Generate and execute the MEX file.

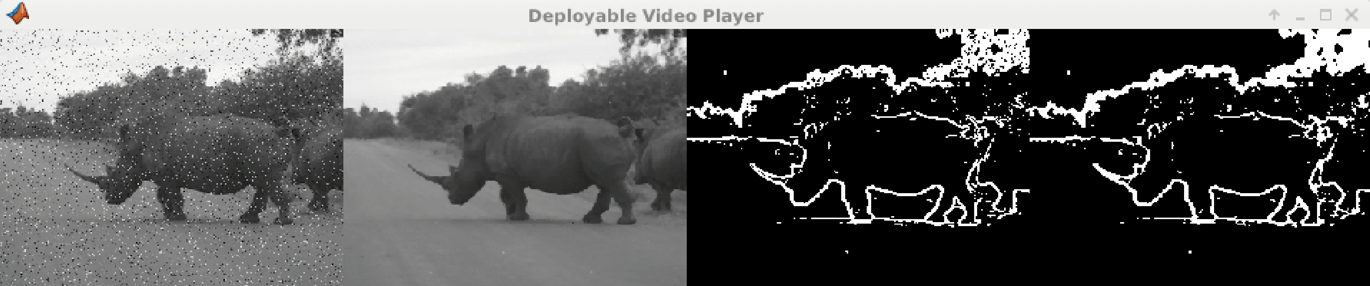
```
codegen('EnhancedEdgeDetectionHDLTestBench');
```

```
Code generation successful.
```

```
EnhancedEdgeDetectionHDLTestBench_mex;
```

```
frame 1: reference and design output differ in more than 20 pixels.
```





The upper video player displays the original color video on the left, and its noisy version after adding salt and pepper noise on the right. The lower video player, from left to right, represents: the grayscale image after color space conversion, the de-noised version after median filter, the edge output after edge detection, and the enhanced edge output after morphological closing operation.

Note that in the lower video chain, only the enhanced edge output (right-most video) is generated from pixel-stream design. The other three are the intermediate videos from the full-frame reference design. To display all of the four videos from the pixel-stream design, you would have written the design file to output four sets of pixels and control signals, and instantiated three more **visionhdl.PixelsToFrame** objects to convert the three intermediate pixel streams back to frames. For the sake of simulation speed and the clarity of the code, this example does not implement the intermediate pixel-stream displays.

HDL Code Generation

To create a new project, enter the following command in the temporary folder

```
coder -hdlcoder -new EnhancedEdgeDetectionProject
```

Then, add the file 'EnhancedEdgeDetectionHDLDesign.m' to the project as the MATLAB Function and 'EnhancedEdgeDetectionHDLTestBench.m' as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a tutorial on creating and populating MATLAB HDL Coder projects.

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

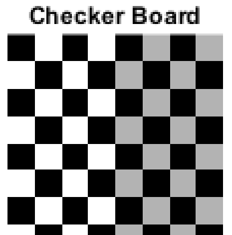
Verify Sobel Edge Detection Algorithm in MATLAB-to-HDL Workflow

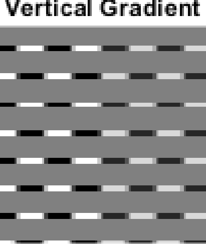
This example shows the HDL generation and verification of a MATLAB® design for the Sobel edge detection algorithm using the MATLAB HDL Coder™ Workflow Advisor. The MATLAB test bench is reused to verify the HDL using an automatically generated HDL cosimulation System object from HDL Verifier™. Any supported HDL simulator can be used, including Mentor Graphics ModelSim®/Questa®, Cadence Incisive®/Xcelium™, or Xilinx® Vivado® Simulator.

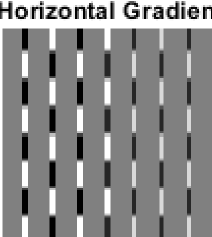
Simulate the Design

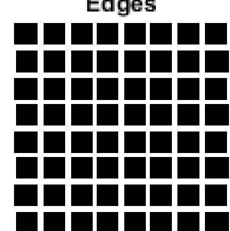
It is a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sobel_tb;
```









Create a New HDL Coder Project

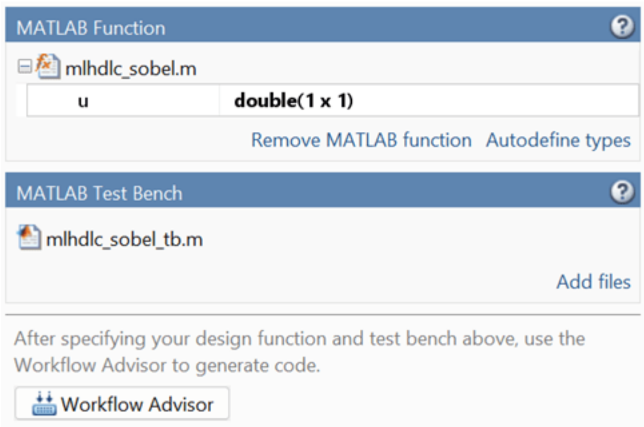
Run the following command to create the HDL code generation project.

```
coder -hdlcoder -new cosim_fil_sobel
```

This action opens a window titled **HDL Coder**.

Specify the Design and the Test Bench

1. Drag the file `mlhdlc_sobel.m` from the Current Folder Browser into the Entry Points tab of the HDL Coder window, under the **MATLAB Function** section.
2. Under the `mlhdlc_sobel.m` file, specify the data type of input argument `data_in` as `double (1 x 1)`
3. Drag the file `mlhdlc_sobel_tb.m` into the HDL Coder window, under **MATLAB Test Bench** section.



Generate HDL Code

1. Click **Workflow Advisor**.
2. Right click on the **Code Generation** step in Workflow Advisor.
3. Choose option **Run to selected task** to run all steps from the beginning of the workflow through to HDL code generation.

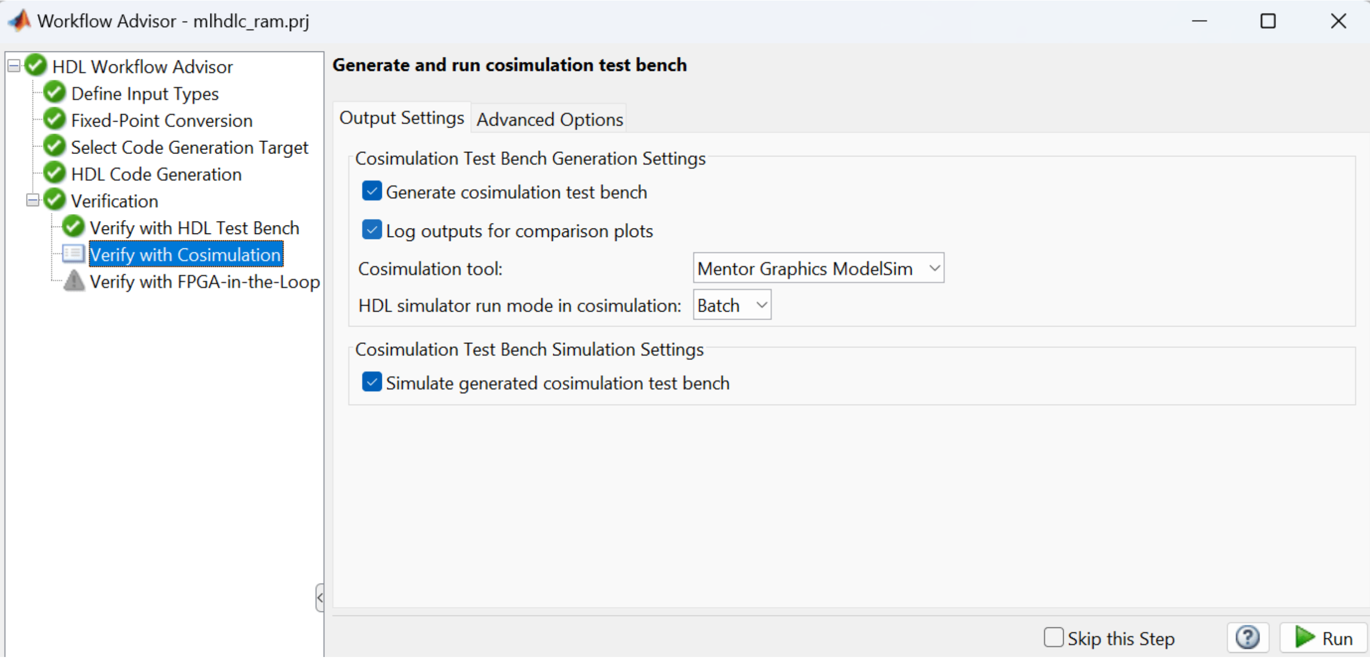
Verify Generated HDL Code with Cosimulation

To run this step, you must have one of the HDL simulators supported by HDL Verifier. See "Cosimulation Requirements" (HDL Verifier). You may skip this step if you do not have access to a supported simulator.

In the **Verify with Cosimulation** step, perform the following steps:

1. Select the **Generate cosimulation test bench** option.
2. Select the **Log outputs for comparison plots** option. This option generates the plotting of the HDL simulator output, the reference MATLAB algorithm output, and the differences between them.
3. For **Cosimulation tool** select your HDL simulator. The HDL simulator executable must be on your system path.
4. To view the waveform in the HDL simulator, select **GUI** mode in the **HDL simulator run mode in cosimulation** list. For Vivado simulator there is no interactive GUI environment. To debug the HDL design, after cosimulation, open the generated file `hdlverifier_cosim_waves.wdb` in Vivado.
5. Select **Simulate generated cosimulation test bench**.
6. Click **Run**.

When the simulation is complete, check the comparison plots in MATLAB. There should be no mismatch between the HDL simulator output and the reference MATLAB algorithm output.



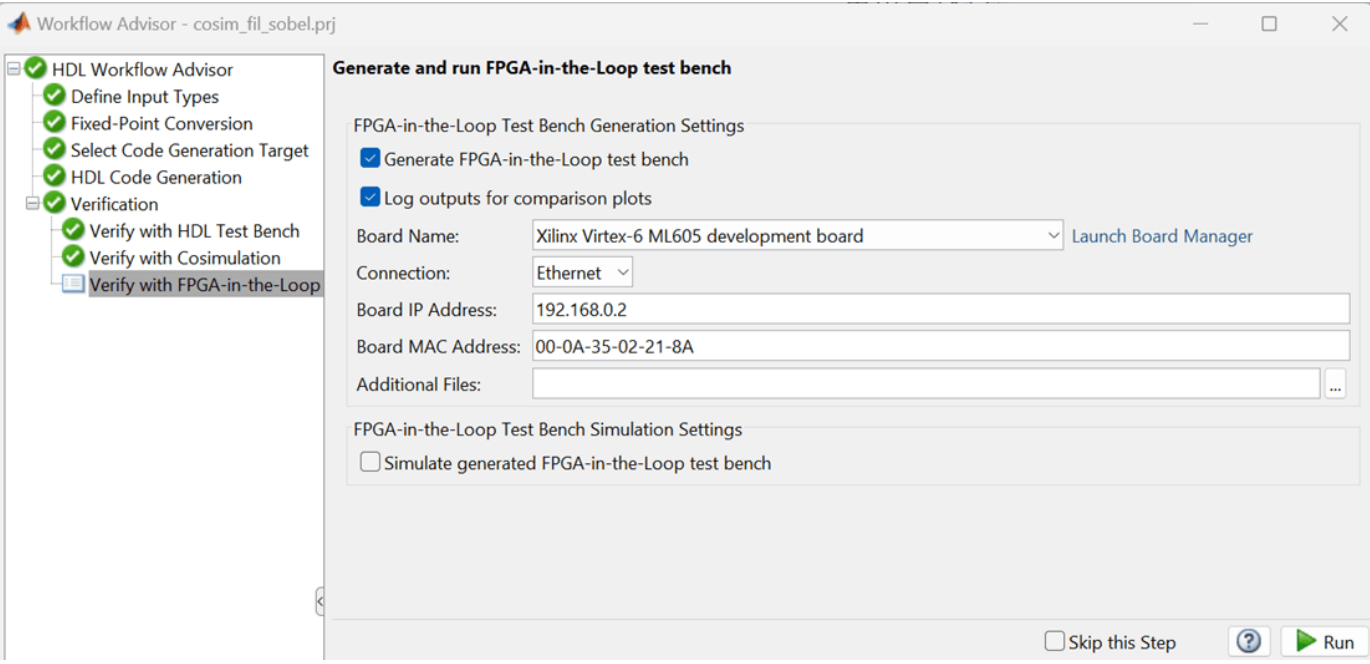
Verify Generated HDL Code with FPGA-in-the-Loop

To run this step, you must have one of the supported FPGA boards. See “FPGA Verification Requirements” (HDL Verifier) and “Set Up FPGA Design Software Tools” (HDL Verifier).

In the **Verify with FPGA-in-the-Loop** step, perform the following steps:

1. Select the **Generate FPGA-in-the-Loop test bench** option.
2. Select the **Log outputs for comparison plots** option. This option generates the plotting of the FPGA output, the reference MATLAB algorithm output, and the differences between them.
3. Select your FPGA board from the **Board Name** list. If your board is not on the list, click `Launch Board Manager` to create a new board entry. The board manager enables adding new boards in many ways including downloading add-on support packages, cloning from existing boards, or creating one from scratch. See “FPGA Board Manager” (HDL Verifier).
4. For boards using Ethernet connections enter your connection information in the **Board IP Address** and **Board MAC Address** fields.
5. Leave the **Additional Files** field empty.
6. Select **Simulate generated FPGA-in-the-Loop test bench**.
7. Click **Run**.

When the simulation is complete, check the comparison plots. There should be no mismatch between the FPGA output and the reference MATLAB algorithm output.


Workflow Advisor - cosim_fil_sobel.prj
HDL Workflow Advisor
Define Input Types
Fixed-Point Conversion
Select Code Generation Target
HDL Code Generation
Verification
Verify with HDL Test Bench
Verify with Cosimulation
Verify with FPGA-in-the-Loop
Generate and run FPGA-in-the-Loop test bench
FPGA-in-the-Loop Test Bench Generation Settings
Generate FPGA-in-the-Loop test bench
Log outputs for comparison plots
Board Name:
Xilinx Virtex-6 ML605 development board
Launch Board Manager
Connection:
Ethernet
Board IP Address:
192.168.0.2
Board MAC Address:
00-0A-35-02-21-8A
Additional Files:
FPGA-in-the-Loop Test Bench Simulation Settings
Simulate generated FPGA-in-the-Loop test bench
Skip this Step
Run

MATLAB Best Practices and Design Patterns for HDL Code Generation

- "Model a Counter for HDL and High-Level Synthesis Code Generation" on page 3-2
- "Model a State Machine for HDL and High-Level Synthesis Code Generation" on page 3-4
- "Generate Hardware Instances For Local Functions" on page 3-8
- "Implement RAM Using MATLAB Code" on page 3-10

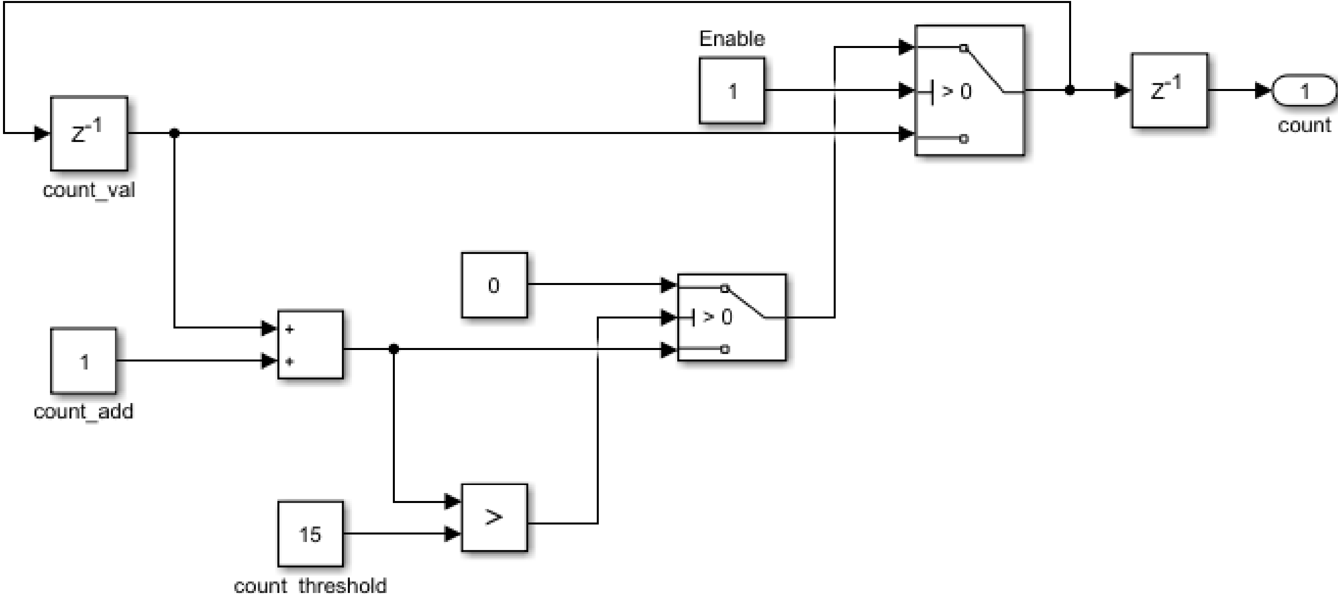
Model a Counter for HDL and High-Level Synthesis Code Generation

To write MATLAB code that models hardware and is suitable for HDL and High-Level Synthesis (HLS) code generation, use this design pattern.

This design pattern demonstrates best practices for writing MATLAB code for HDL and HLS code generation:

- Initialize persistent variables to a specific value. In this example, an `if` statement and the `isempty` function initialize the persistent variable. If you do not initialize the persistent variable, then you cannot generate HDL and HLS code.
- Inside a function, read persistent variables before they are modified so that the persistent variables are inferred as registers.

This Simulink model illustrates the MATLAB counter modeled in this example.



To learn how to model the counter in Simulink, see “Create HDL-Compatible Simulink Model”.

MATLAB Code for the Counter

The function `mlhdlc_counter` is a behavioral model of a 4-bit synchronous up counter. The input signal, `enable_ctr`, triggers the value of the count register, `count_val`, to increase by one. The counter continues to increase by one each time the input is nonzero, until the count reaches a limit of 15. After the counter reaches this limit, the counter returns to zero. A persistent variable, which is initialized to zero, represents the current value of the count. Two `if` statements determine the value of the count based on the input.

To define the `mldhlc_counter` and `mldhlc_counter_tb`, use these codes:

MATLAB Code

```
%#codegen
function count = mlhdlc_counter(enable_ctr)
% four bit synchronous up counter

% persistent variable for the state
persistent count_val;
if isempty(count_val)
    count_val = 0;
end

% counting up
if enable_ctr
    count_val=count_val+1;

    % limit to four bits
    if count_val>15
        count_val=0;
    end
end

count=count_val;

end
```

MATLAB Testbench

```
for i = 1:100
    if mod(i,5) == 0
        % do not increment the counter if mod(i,5) is
        val = mlhdlc_counter(false);
    else
        val = mlhdlc_counter(true);
    end
end
```

See Also

`codegen` | `coder.HdlConfig`

More About

- “Model a State Machine for HDL and High-Level Synthesis Code Generation” on page 3-4
- “Implement RAM Using MATLAB Code” on page 3-10
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4

Model a State Machine for HDL and High-Level Synthesis Code Generation

In this section...

"MATLAB Code for the Mealy State Machine" on page 3-4
"MATLAB Code for the Moore State Machine" on page 3-6

The following design pattern shows MATLAB examples of Mealy and Moore state machines which are suitable for HDL and High-Level Synthesis (HLS) code generation.

The MATLAB code in these models demonstrates best practices for writing MATLAB models for HDL and HLS code generation.

- With a `switch` block, use the `otherwise` statement to ensure that the model accounts for all conditions. If the model does not cover all conditions, the generated HDL code can contain errors.
- To designate the states in a state machine, use variables with numerical values.

MATLAB Code for the Mealy State Machine

In a Mealy state machine, the output depends on the state and the input. In a Moore state machine, the output depends only on the state.

The following MATLAB code defines the `mlhdlc_fsm_mealy` function. A persistent variable represents the current state. A `switch` block uses the current state and input to determine the output and new state. In each `case` in the `switch` block, an `if-else` statement calculates the new state and output.

MATLAB Code

```
%#codegen
function Z = mlhdlc_fsm_mealy(A)
% Mealy State Machine

% y = f(x,u) :
% all actions are condition actions and
% outputs are function of state and input

% define states
S1 = 0;
S2 = 1;
S3 = 2;
S4 = 3;

persistent current_state;
if isempty(current_state)
    current_state = S1;
end

% switch to new state based on the value state register
switch (current_state)

    case S1,
```

```
        % value of output 'Z' depends both on state and inputs
        if (A)
            Z = true;
            current_state = S1;
        else
            Z = false;
            current_state = S2;
        end

    case S2,

        if (A)
            Z = false;
            current_state = S3;
        else
            Z = true;
            current_state = S2;
        end

    case S3,

        if (A)
            Z = false;
            current_state = S4;
        else
            Z = true;
            current_state = S1;
        end

    case S4,

        if (A)
            Z = true;
            current_state = S1;
        else
            Z = false;
            current_state = S3;
        end

    otherwise,

        Z = false;
end
```

MATLAB Test Bench

```
for i = 1:100
    if mod(i,2) == 0
        val = mlhdlc_fsm_mealy(true);
    else
        val = mlhdlc_fsm_mealy(false);
    end
end
```

MATLAB Code for the Moore State Machine

The following MATLAB code defines the `mlhdlc_fsm_moore` function. A persistent variable represents the current state, and a `switch` block uses the current state to determine the output and new state. In each `case` in the `switch` block, an `if-else` statement calculates the new state and output. The value of the state is represented by numerical variables.

MATLAB Code

```
%#codegen
function Z = mlhdlc_fsm_moore(A)
% Moore State Machine

% y = f(x) :
% all actions are state actions and
% outputs are pure functions of state only

% define states
S1 = 0;
S2 = 1;
S3 = 2;
S4 = 3;


% using persistent keyword to model state registers in hardware
persistent curr_state;
if isempty(curr_state)
    curr_state = S1;
end

% switch to new state based on the value state register
switch (curr_state)

    case S1,

        % value of output 'Z' depends only on state and not on inputs
        Z = true;

        % decide next state value based on inputs
        if (~A)
            curr_state = S1;
        else
            curr_state = S2;
        end

    case S2,

        Z = false;

        if (~A)
            curr_state = S1;
        else
            curr_state = S3;
        end

    case S3,
```

```
        Z = false;

        if (~A)
            curr_state = S2;
        else
            curr_state = S4;
        end

    case S4,

        Z = true;
        if (~A)
            curr_state = S3;
        else
            curr_state = S1;
        end

    otherwise,
        Z = false;
end
```

MATLAB Test Bench

```
for i = 1:100
    if mod(i,2) == 0
        val = mlhdlc_fsm_moore(true);
    else
        val = mlhdlc_fsm_moore(false);
    end
end
```

See Also

`codegen` | `coder.HdlConfig`

More About

- “Functions Supported for HDL and HLS Code Generation” on page 1-2
- “Model a Counter for HDL and High-Level Synthesis Code Generation” on page 3-2
- “Implement RAM Using MATLAB Code” on page 3-10

Generate Hardware Instances For Local Functions

In this section...
“MATLAB Local Functions” on page 3-8
“MATLAB Code for mlhdlc_two_counters.m” on page 3-8

The following example shows how to use local functions in MATLAB, so that each execution of a local function corresponds to a separate hardware module in the generated HDL code.

MATLAB Local Functions

This example demonstrates best practices for writing local functions in MATLAB code that is suitable for HDL code generation.

- If your MATLAB code executes a local function multiple times, the generated HDL code does not necessarily instantiate multiple hardware modules. Rather than instantiating multiple hardware modules, multiple calls to a function typically update the state variable.
- If you want the generated HDL code to contain multiple hardware modules corresponding to each execution of a local function, specify two different local functions with the same code but different function names. If you want to avoid code duplication, consider using System objects to implement the behavior in the function, and instantiate the System object multiple times.
- If you want to specify a separate HDL file for each local function in the MATLAB code, in the Workflow Advisor, on the **Advanced** tab in the HDL Code Generation section, select **Generate instantiable code for functions** .

MATLAB Code for mlhdlc_two_counters.m

This function creates two counters and adds the output of these counters. To create two counters, there are two local functions with identical code, `counter` and `counter2`. The main method calls each of these local functions once. If the function were to call the `counter` function twice, separate hardware modules for the counters would not be generated in the HDL code.

```
%#codegen
function total_count = mlhdlc_two_counters(a,b)

%This function contains two different local functions with identical
%counters and calls each counter once.

total_count1=counter(a);

total_count2=counter2(b);

total_count=total_count1+total_count2;

function count = counter(enable_ctr)
%four bit synchronous up counter

%persistent variable for the state
persistent count_val;
if isempty(count_val)
    count_val = 0;
```

```
end

%counting up
if enable_ctr
    count_val=count_val+1;
end

%limit from four bits
if count_val>15
    count_val=0;
end

count=count_val;



function count = counter2(enable_ctr)
%four bit synchronous up counter

%persistent variable for the state
persistent count_val;
if isempty(count_val)
    count_val = 0;
end

%counting up
if enable_ctr
    count_val=count_val+1;
end

%limit from four bits
if count_val>15
    count_val=0;
end

count=count_val;
```

See Also

`codegen` | `coder.HdlConfig`

More About

- "Functions Supported for HDL and HLS Code Generation" on page 1-2
- "Model a Counter for HDL and High-Level Synthesis Code Generation" on page 3-2
- "Model a State Machine for HDL and High-Level Synthesis Code Generation" on page 3-4
- "Map Persistent Arrays and dsp.Delay Objects to RAM" on page 8-6

Implement RAM Using MATLAB Code

In this section...

"Implement RAM Using a Persistent Array or System object Properties" on page 3-10
"Implement RAM Using hdl.RAM" on page 3-11

You can write MATLAB code that maps to RAM during HDL code generation by using:

- Persistent arrays or private properties in a user-defined System object.
- `hdl.RAM` System objects.

The following examples model the same line delay in MATLAB. The line delay uses memory in a ring structure. Data is written to one location and read from another location in such a way that the data written is read after a delay of a specific number of cycles. The RAM read address is generated by a counter. The write address is generated by adding a constant value to the read address.

Implement RAM Using a Persistent Array or System object Properties

This example shows a line delay that implements the RAM behavior using a persistent array with the function `mlhdlc_hdlram_persistent`. Changing a specific value in the persistent array is equivalent to writing to the RAM. Accessing a specific value in the array is equivalent to reading from the RAM.

You can implement RAM by using user-defined System object private properties in the same way.

```
%#codegen
function data_out = mlhdlc_hdlram_persistent(data_in)

persistent hRam;
if isempty(hRam)
    hRam = zeros(128,1);
end

% read address counter
persistent rdAddrCtr;
if isempty(rdAddrCtr)
    rdAddrCtr = 1;
end

% ring counter length
ringCtrLength = 10;
ramWriteAddr = rdAddrCtr + ringCtrLength;

ramWriteData = data_in;
%ramWriteEnable = true;

ramReadAddr = rdAddrCtr;

% To write and read the memory, manipulate the values in the array

hRam(ramWriteAddr)=ramWriteData;
ramRdDout=hRam(ramReadAddr);

rdAddrCtr = rdAddrCtr + 1;
```

```
data_out = ramRdDout;
```

Implement RAM Using hdl.RAM

This example shows a line delay that implements the RAM behavior using `hdl.RAM` with the function, `mlhdlc_hdlram_sysobj`. In this function, the `step` method of the `hdl.RAM` System object reads and writes to specific locations in `hRam`. Code generation from `hdl.RAM` has the same restrictions as code generation from other System objects. For details, see “Limitations of HDL Code Generation for System Objects” on page 1-16.

```
%#codegen
function data_out = mlhdlc_hdlram_sysobj(data_in)
persistent hRam;
if isempty(hRam)
    hRam = hdl.RAM('RAMType', 'Dual port');
end

% read address counter
persistent rdAddrCtr;
if isempty(rdAddrCtr)
    rdAddrCtr = 0;
end

% ring counter length
ringCtrLength = 10;
ramWriteAddr = rdAddrCtr + ringCtrLength;

ramWriteData = data_in;
ramWriteEnable = true;

ramReadAddr = rdAddrCtr;

% To write and read the memory, call the RAM object
[~,ramRdDout] = hRam(ramWriteData,ramWriteAddr, ...
                      ramWriteEnable,ramReadAddr);

rdAddrCtr = rdAddrCtr + 1;

data_out = ramRdDout;
```

See Also

`codegen` | `coder.HdlConfig`

More About

- “Functions Supported for HDL and HLS Code Generation” on page 1-2
- “Model a Counter for HDL and High-Level Synthesis Code Generation” on page 3-2
- “Model a State Machine for HDL and High-Level Synthesis Code Generation” on page 3-4
- “Map Persistent Arrays and dsp.Delay Objects to RAM” on page 8-6

Fixed-Point Conversion

- “Specify Type Proposal Options” on page 4-2
- “Log Data for Histogram” on page 4-5
- “View and Modify Variable Information” on page 4-7
- “Automated Fixed-Point Conversion” on page 4-9
- “Custom Plot Functions” on page 4-23
- “Edit Configuration Parameters for Fixed-Point Code Generation” on page 4-24
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 4-26
- “Inspecting Data Using the Simulation Data Inspector” on page 4-31
- “Enable Plotting Using the Simulation Data Inspector” on page 4-33
- “Replacing Functions Using Lookup Table Approximations” on page 4-34
- “Replace a Custom Function with a Lookup Table” on page 4-35
- “Replace the exp Function with a Lookup Table” on page 4-41
- “Data Type Issues in Generated Code” on page 4-47
- “Working with Fixed-Point Code” on page 4-49
- “Floating-Point to Fixed-Point Conversion” on page 4-51
- “Fixed-Point Type Conversion and Refinement” on page 4-61
- “Working with Generated Fixed-Point Files” on page 4-68
- “Fixed-Point Type Conversion and Derived Ranges” on page 4-73
- “Generate HDL-Compatible Lookup Table Function Replacements Using coder.approximate” on page 4-78

Specify Type Proposal Options

Basic Type Proposal Settings	Values	Description
Fixed-point type proposal mode	Propose fraction lengths for specified word length	Use the specified word length for data type proposals and propose the minimum fraction lengths to avoid overflows.
	Propose word lengths for specified fraction length (default)	Use the specified fraction length for data type proposals and propose the minimum word lengths to avoid overflows.
Default word length	14 (default)	Default word length to use when **Fixed-point type proposal mode** is set to `Propose fraction lengths for specified word lengths`
Default fraction length	4 (default)	Default fraction length to use when **Fixed-point type proposal mode** is set to `Propose word lengths for specified fraction lengths`

Advanced Type Proposal Settings	Values	Description
When proposing types **Note** Manually-entered static ranges always take precedence over simulation ranges.	ignore simulation ranges	Propose data types based on derived ranges.
	ignore derived ranges	Propose data types based on simulation ranges.
	use all collected data (default)	Propose data types based on both simulation and derived ranges.
Propose target container types	Yes	Propose data type with the smallest word length that can represent the range and is suitable for C code generation (8,16,32, 64 ...). For example, for a variable with range `[0..7]`, propose a word length of 8 rather than 3.
	No (default)	Propose data types with the minimum word length needed to represent the value.
Optimize whole numbers	No	Do not use integer scaling for variables that were whole numbers during simulation.
	Yes (default)	Use integer scaling for variables that were whole numbers during simulation.

Advanced Type Proposal Settings	Values	Description
Signedness	Automatic (default)	Proposes signed and unsigned data types depending on the range information for each variable.
	Signed	Propose signed data types.
	Unsigned	Propose unsigned data types.
Safety margin for sim min/max (%)	0 (default)	Specify safety factor for simulation minimum and maximum values. The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of `55` specifies that you want a range at least 55 percent larger. A value of `-15` specifies that a range up to 15 percent smaller is acceptable.
Search paths	`''` (default)	Add paths to the list of paths to search for MATLAB files. Separate list items with a semicolon.

fimath Settings	Values	Description
Rounding method	Ceiling	Specify the `fimath` properties for the generated fixed-point data types. The default fixed-point math properties use the `Floor` rounding and `Wrap` overflow. These settings generate the most efficient code but might cause problems with overflow. After code generation, if required, modify these settings to optimize the generated code, or example, avoid overflow or eliminate bias, and then rerun the verification. For more information on `fimath` properties, see “fimath Object Properties”.
	Convergent	
	Floor (default)	
	Nearest	
	Round	
	Zero	
Overflow action	Saturate	
	Wrap (default)	
Product mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	

Generated File Settings	Value	Description
Generated fixed-point file name suffix	_fixpt (default)	Specify the suffix to add to the generated fixed-point file names.

Plotting and Reporting Settings	Values	Description
Custom plot function	'' (default)	Specify the name of a custom plot function to use for comparison plots.
Plot with Simulation Data Inspector	No (default)	Specify whether to use the Simulation Data Inspector for comparison plots.
	Yes	
Highlight potential data type issues	No (default)	Specify whether to highlight potential data types in the generated html report. If this option is turned on, the report highlights single-precision, double-precision, and expensive fixed-point operation usage in your MATLAB code.
	Yes	

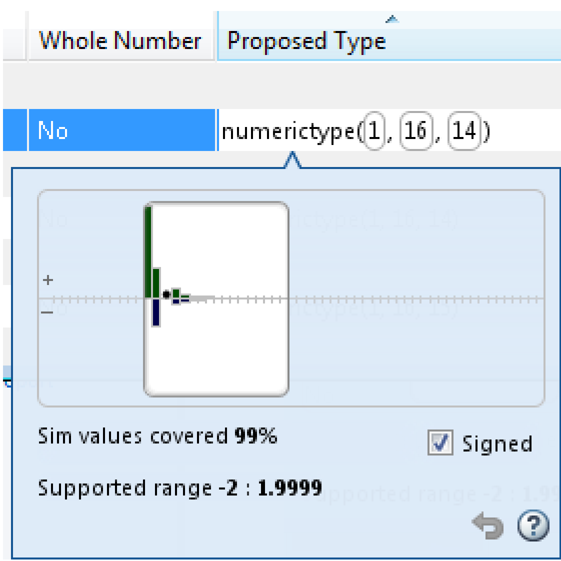
Log Data for Histogram

To log data for histograms:

1. In the Fixed-Point Conversion window, click **Run Simulation** and select `Log data for histogram`, and then click the Run Simulation button.

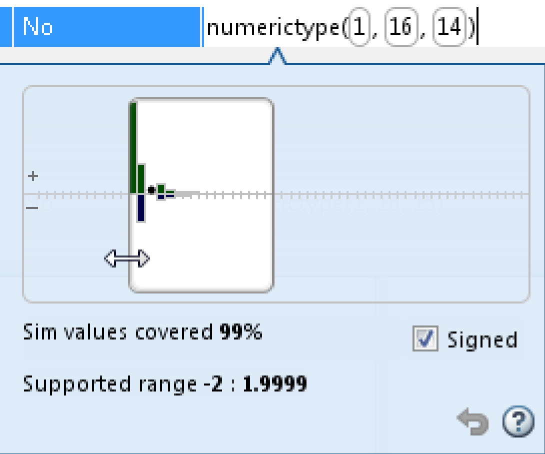
   The simulation runs and the simulation minimum and maximum ranges are displayed on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.

2. To view a histogram for a variable, click the variable's **Proposed Type** field.



3. You can view the effect of changing the proposed data types by:

   - Selecting and dragging the white bounding box in the histogram window. This action does not change the word length of the proposed data type, but modifies the position of the binary point within the word so that the fraction length of the proposed data type changes.
   - Selecting and dragging the left edge of the bounding box to increase or decrease the word length. This action does not change the fraction length or the position of the binary point.



- Selecting and dragging the right edge to increase or decrease the fraction length of the proposed data type. This action does not change the position of the binary point. The word length changes to accommodate the fraction length.
- Selecting or clearing **Signed**. Clear **Signed** to ignore negative values.

Before committing changes, you can revert to the types proposed by the automatic conversion by clicking .

View and Modify Variable Information

View Variable Information

In the Fixed-Point Conversion tool, you can view information about the variables in the MATLAB functions. To view information about the variables for the selected function, use the **Variables** tab or pause over a variable in the code window. For more information, see “Viewing Variables” on page 4-14.

You can view the variable information:

- **Variable**

  Variable name. Variables are classified and sorted as inputs, outputs, persistent, or local variables.
- **Type**

  The original size, type, and complexity of each variable.
- **Sim Min**

  The minimum value assigned to the variable during simulation.
- **Sim Max**

  The maximum value assigned to the variable during simulation.

To search for a variable in the MATLAB code window and on the **Variables** tab, use `Ctrl+F`.

Modify Variable Information

If you modify variable information, the app highlights the modified values using bold text. You can modify the following fields:

- **Static Min**

  You can enter a value for **Static Min** into the field or promote **Sim Min** information. See “Promote Sim Min and Sim Max Values” on page 4-8.

  Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.
- **Static Max**

  You can enter a value for **Static Max** into the field or promote **Sim Max** information. See “Promote Sim Min and Sim Max Values” on page 4-8.

  Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.
- **Whole Number**

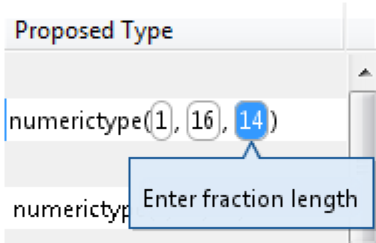
  The app uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

  Editing this field does not trigger static range analysis, but the app uses the edited value in subsequent analyses.

- **Proposed Type**

  You can modify the signedness, word length, and fraction length settings individually:

  - On the **Variables** tab, modify the value in the **ProposedType** field.



  - In the code window, select a variable, and then modify the **Proposed Type** field.

  If you selected to log data for a histogram, the histogram dynamically updates to reflect the modifications to the proposed type. You can also modify the proposed type in the histogram, see “Histogram” on page 4-19.

Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select `Reset entire table`.
- To revert the type of a selected variable to the type computed by the app, right-click the field and select `Undo changes`.
- To revert changes to variables, right-click the field and select `Undo changes for all variables`.
- To clear a static range value, right-click an edited field and select `Clear this static range`.
- To clear manually entered static range values, right-click anywhere on the **Variables** tab and select `Clear all manually entered static ranges`.

Promote Sim Min and Sim Max Values

With the app, you can promote simulation minimum and maximum values to static minimum and maximum values. This capability is useful if you have not specified static ranges and you have simulated the model with inputs that cover the full intended operating range.

To copy:

- A simulation range for a selected variable, select a variable, right-click, and then select `Copy sim range`.
- Simulation ranges for top-level inputs, right-click the Static Min or Static Max column, and then select `Copy sim ranges for all top-level inputs`.
- Simulation ranges for persistent variables, right-click the Static Min or Static Max column, and then select `Copy sim ranges for all persistent variables`.

Automated Fixed-Point Conversion

In this section...

"License Requirements" on page 4-9
"Automated Fixed-Point Conversion Capabilities" on page 4-9
"Code Coverage" on page 4-10
"Proposing Data Types" on page 4-12
"Locking Proposed Data Types" on page 4-14
"Viewing Functions" on page 4-14
"Viewing Variables" on page 4-14
"Histogram" on page 4-19
"Function Replacements" on page 4-20
"Validating Types" on page 4-21
"Testing Numerics" on page 4-21
"Detecting Overflows" on page 4-21

License Requirements

Fixed-point conversion requires the following licenses:

- Fixed-Point Designer
- MATLAB Coder™

Automated Fixed-Point Conversion Capabilities

You can convert floating-point MATLAB code to fixed-point code using the Fixed-Point Conversion tool in HDL Coder projects. You can choose to propose data types based on simulation range data, derived (also known as static) range data, or both.

You can manually enter static ranges. These manually-entered ranges take precedence over simulation ranges and the tool uses them when proposing data types. In addition, you can modify and lock the proposed type so that the tool cannot change it. For more information, see "Locking Proposed Data Types" on page 4-14.

For a list of supported MATLAB features and functions, see "MATLAB Language Features Supported for Automated Fixed-Point Conversion".

During fixed-point conversion, you can:

- Verify that your test files cover the full intended operating range of your algorithm using code coverage results.
- Propose fraction lengths based on default word lengths.
- Propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.

- Validate that you can build your project with the proposed data types.
- Test numerics by running the test bench with the fixed-point types applied.
- View a histogram of bits used by each variable.
- Detect overflows.

Code Coverage

By default, the Fixed-Point Conversion tool shows code coverage results. Your test files must exercise the algorithm over its full operating range so that the simulation ranges are accurate. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want. Reviewing code coverage results helps you verify that your test files are exercising the algorithm adequately. If the code coverage is inadequate, modify the test files or add more test files to increase coverage. If you simulate multiple test files in one run, the tool displays cumulative coverage. However, if you specify multiple test files but run them one at a time, the tool displays the coverage of the file that ran last.

The tool displays a color-coded coverage bar to the left of the code.

```
1   function y = ex_2ndOrder_filter(x) %#codegen
2     persistent z
3     if isempty(z)
4         z = zeros(2,1);
5     end
6     % [b,a] = butter(2, 0.25)
7     b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
8     a = [                 1, -0.942809041582063,  0.3333333333333333];
9
10
11    y = zeros(size(x));
12    for i=1:length(x)
13        y(i) = b(1)*x(i) + z(1);
14        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
15        z(2) = b(3)*x(i)        - a(3) * y(i);
16    end
17  end
```

This table describes the color coding.

Coverage Bar Color	Indicates
Green	One of the following situations: • The entry-point function executes multiple times and the code executes more than one time. • The entry-point function executes one time and the code executes one time. Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

When you pause over the coverage bar, the color highlighting extends over the code. For each section of code, the app displays the number of times that section executes.

```
1  function y = ex_2ndOrder_filter(x) %#codegen                    3 calls
2    persistent z
3    if isempty(z)
4        z = zeros(2,1);                                            1 calls
5    end                                                            3 calls
6    % [b,a] = butter(2, 0.25)
7    b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
8    a = [                 1, -0.942809041582063,  0.3333333333333333];
9
10
11   y = zeros(size(x));
12   for i=1:length(x)                                              768 calls
13       y(i) = b(1)*x(i) + z(1);
14       z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
15       z(2) = b(3)*x(i)        - a(3) * y(i);
16   end
17 end                                                              3 calls
```

To verify that your test files are testing your algorithm over the intended operating range, review the code coverage results.

Coverage Bar Color	Action
Green	If you expect sections of code to execute more frequently than the coverage shows, either modify the MATLAB code or the test files.
Orange	This behavior is expected for initialization code, for example, the initialization of persistent variables. If you expect the code to execute more than one time, either modify the MATLAB code or the test files.

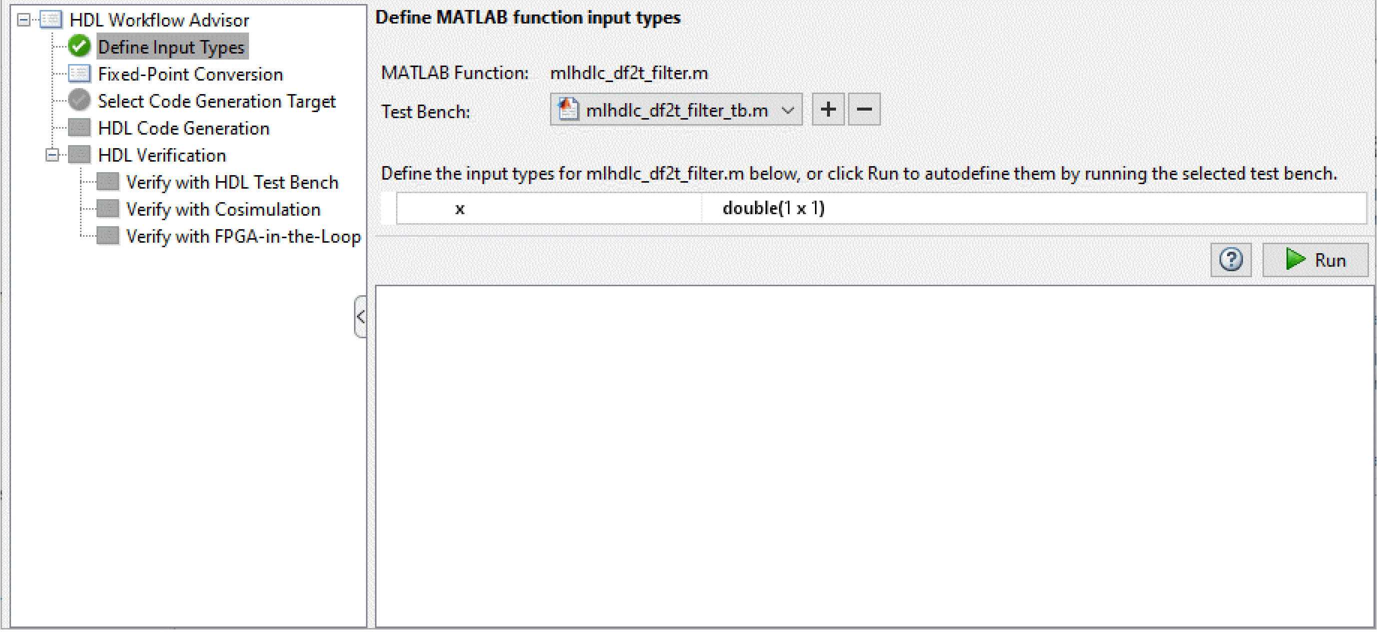
Coverage Bar Color	Action
Red	If the code that does not execute is an error condition, this behavior is acceptable. If you expect the code to execute, either modify the MATLAB code or the test files. If the code is written conservatively and has upper and lower boundary limits, and you cannot modify the test files to reach this code, add static minimum and maximum values. See “Computing Derived Ranges” on page 4-13.

Code coverage is on by default. Turn it off only after you have verified that you have adequate test file coverage. Turning off code coverage can speed up simulation. To turn off code coverage, in the Fixed-Point Conversion tool:

1. Click **Run Simulation**.
2. Clear `Show code coverage`.

Proposing Data Types

In the **Define Input Types** step, you specify a test bench that calls the MATLAB function. The tool runs the test file to analyze the code and infer the types for entry-point input arguments.



The Fixed-Point Conversion tool proposes fixed-point data types based on computed ranges and the word length or fraction length setting. The computed ranges are based on simulation range data, derived range data, or both. If you run a simulation and compute derived ranges, the conversion tool merges the simulation and derived ranges.

Note You cannot propose data types based on derived ranges for MATLAB classes.

You can manually enter static ranges. These manually-entered ranges take precedence over simulation ranges and the tool uses them when proposing data types. If you analyze ranges using derived range analysis alone, you must enter static ranges. In addition, you can modify and lock the proposed type so that the tool cannot change it. For more information, see “Locking Proposed Data Types” on page 4-14.

Running a Simulation

When you open the Fixed-Point Conversion tool, the tool generates an instrumented MEX function for your MATLAB design. If the build completes without errors, the tool displays compiled information (type, size, complexity) for functions and variables in your code. To navigate to local functions, click the **Functions** tab. If build errors occur, the tool provides error messages that link to the line of code that caused the build issues. You must address these errors before running a simulation. Use the link to navigate to the offending line of code in the MATLAB editor and modify the code to fix the issue. If your code uses functions that are not supported for fixed-point conversion, the tool displays them on the **Function Replacements** tab. See “Function Replacements” on page 4-20.

Before running a simulation, specify the test bench that you want to run. When you run a simulation, the tool runs the test bench, calling the instrumented MEX function. If you modify the MATLAB design code, the tool automatically generates an updated MEX function before running the test bench.

If the test bench runs successfully, the simulation minimum and maximum values and the proposed types are displayed on the **Variables** tab. If you manually enter static ranges for a variable, the manually-entered ranges take precedence over the simulation ranges. If you manually modify the proposed types by typing or using the histogram, the data types are locked so that the tool cannot modify them.

If the test bench fails, the errors are displayed on the **Simulation Output** tab.

The test bench should exercise your algorithm over its full operating range. The quality of the proposed fixed-point data types depends on how well the test bench covers the operating range of the algorithm with the desired accuracy.

Optionally, you can select to log data for histograms. After running a simulation, you can view the histogram for each variable. For more information, see “Histogram” on page 4-19.

Computing Derived Ranges

The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values or proposed data types for all input variables. To improve the analysis, enter as much static range information as possible for other variables. You can manually enter ranges or promote simulation ranges to use as static ranges. Manually-entered static ranges always take precedence over simulation ranges.

If you know what data type your hardware target uses, set the proposed data types to match this type. Manually-entered data types are locked so that the tool cannot modify them. The tool uses these data types to calculate the input minimum and maximum values and to derive ranges for other variables. For more information, see “Locking Proposed Data Types” on page 4-14.

When you select **Compute Derived Ranges**, the tool runs a derived range analysis to compute static ranges for variables in your MATLAB algorithm. When the analysis is complete, the static ranges are displayed on the **Variables** tab. If the run produces `+/-Inf` derived ranges, consider defining ranges for all persistent variables.

Optionally, you can select **Quick derived range analysis**. With this option, the conversion tool performs faster static analysis. The computed ranges might be larger than necessary. Select this option in cases where the static analysis takes more time than you can afford.

If the derived range analysis for your project is taking a long time, you can optionally set a timeout. The tool aborts the analysis when the timeout is reached.

Locking Proposed Data Types

You can lock proposed data types against changes by the Fixed-Point Conversion tool using one of the following methods:

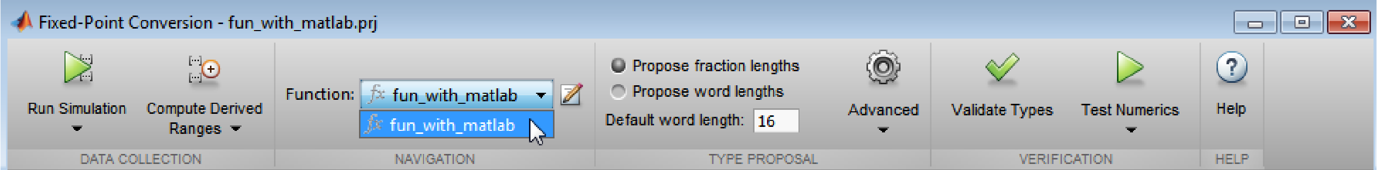
- Manually setting a proposed data type in the Fixed-Point Conversion tool.
- Right-clicking a type proposed by the tool and selecting `Lock computed value`.

The tool displays locked data types in bold so that they are easy to identify. You can unlock a type using one of the following methods:

- Manually overwriting it.
- Right-clicking it and selecting `Undo changes`. This action unlocks only the selected type.
- Right-clicking and selecting `Undo changes for all variables`. This action unlocks all locked proposed types.

Viewing Functions

You can view a list of functions in your project on the **Navigation** pane. This list also includes function specializations and class methods. When you select a function from the list, the MATLAB code for that function or class method is displayed in the Fixed-Point Conversion tool code window.



After conversion, the left pane also displays a list of output files including the fixed-point version of the original algorithm. If your function is not specialized, the conversion retains the original function name in the fixed-point filename and appends the fixed-point suffix. For example, the fixed-point version of `fun_with_matlab.m` is `fun_with_matlab_fixpt.m`.

Viewing Variables

The **Variables** tab provides the following information for each variable in the function selected in the **Navigation** pane:

- **Type** — The original data type of the variable in the MATLAB algorithm.
- **Sim Min** and **Sim Max** — The minimum and maximum values assigned to the variable during simulation.

  You can edit the simulation minimum and maximum values. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.
- **Static Min** and **Static Max** — The static minimum and maximum values.

  To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values for all input variables. To improve the analysis, enter as much static range information as possible for other variables.

  When you compute derived ranges, the Fixed-Point Conversion tool runs a static analysis to compute static ranges for variables in your code. When the analysis is complete, the static ranges are displayed. You can edit the computed results. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.
- **Whole Number** — Whether all values assigned to the variable during simulation are integers.

  The Fixed-Point Conversion tool determines whether a variable is always a whole number. You can modify this field. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.
- The proposed fixed-point data type for the specified word (or fraction) length. Proposed data types use the `numerictype` notation. For example, `numerictype(1,16,12)` denotes a signed fixed-point type with a word length of 16 and a fraction length of 12. `numerictype(0,16,12)` denotes an unsigned fixed-point type with a word length of 16 and a fraction length of 12.

  Because the tool does not apply data types to expressions, it does not display proposed types for them. Instead, it displays their original data types.

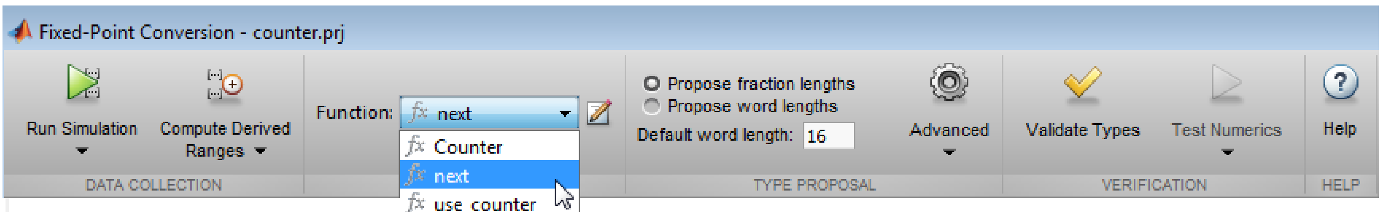
You can also view and edit variable information in the code pane by placing your cursor over a variable name.

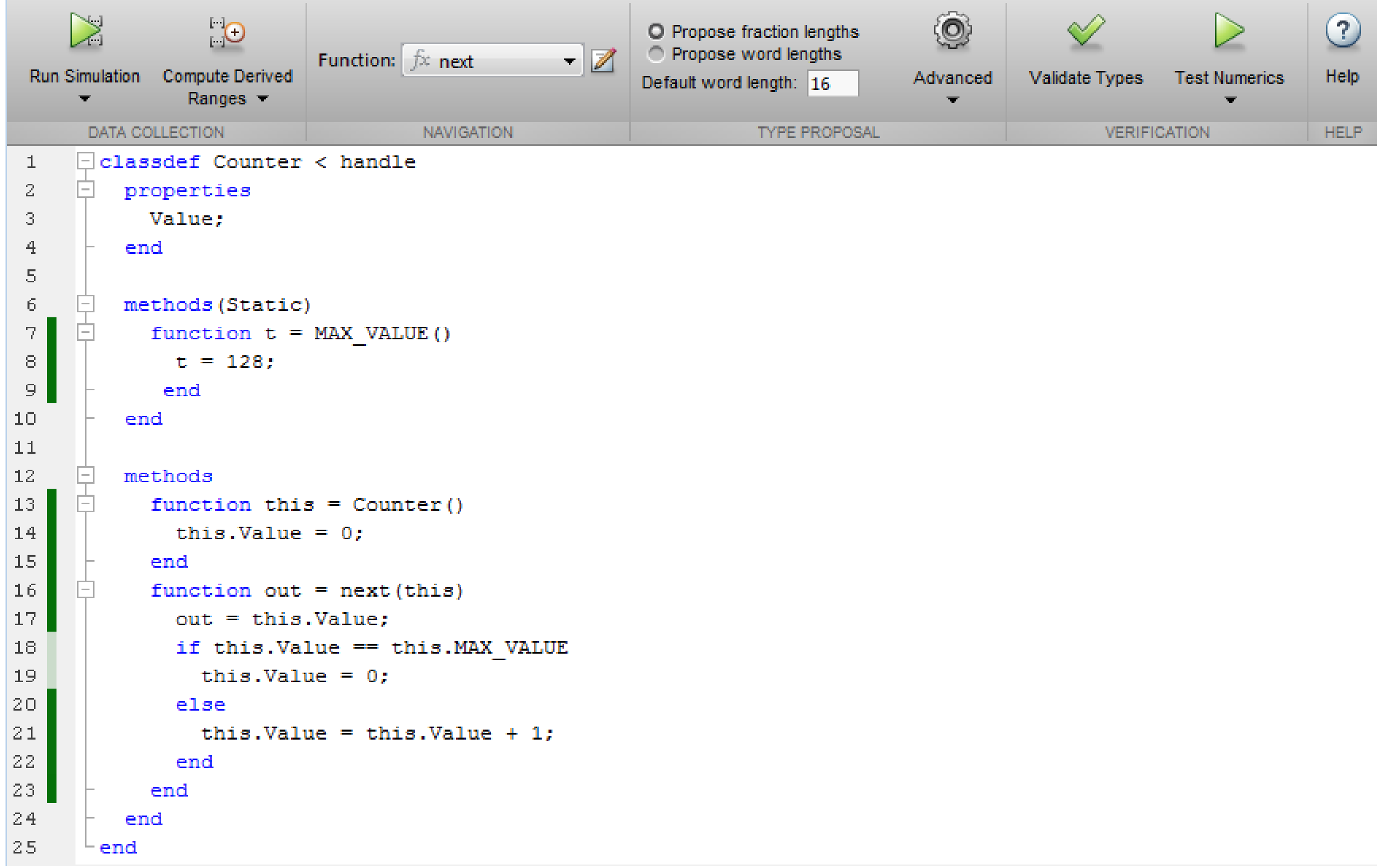
You can use `Ctrl+F` to search for variables in the MATLAB code and on the **Variables** tab. The tool highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

Viewing Information for MATLAB Classes

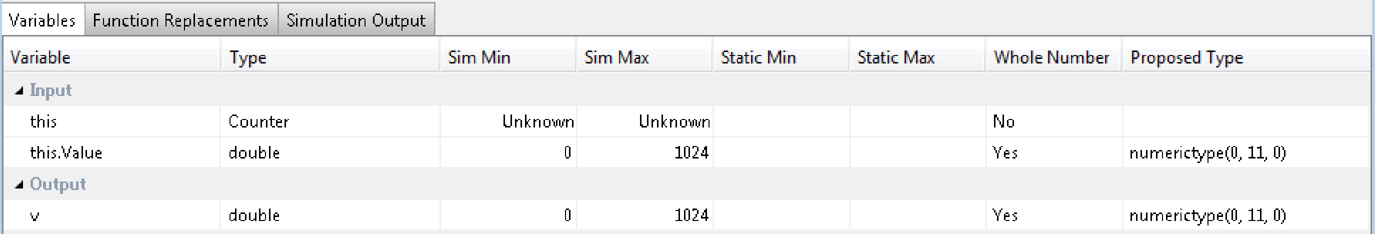
The tool displays:

- Code for MATLAB classes and code coverage for class methods in the code window. Use the **Function** list in the Navigation bar to select which class or class method to view.





- Information about MATLAB classes on the **Variables** tab.



Variables | Function Replacements | Simulation Output

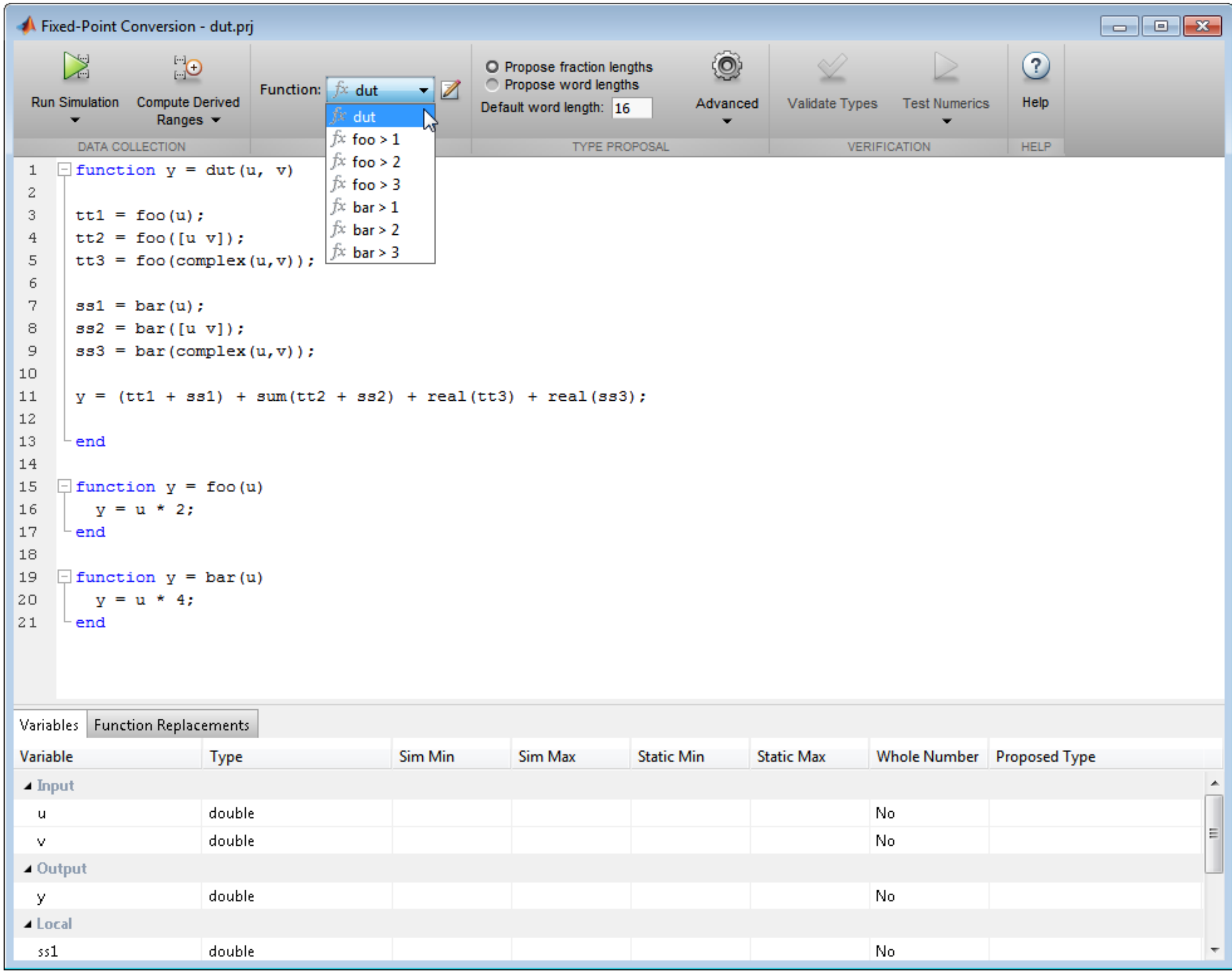
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
◢ Input							
this	Counter	Unknown	Unknown			No	
this.Value	double	0	1024			Yes	numerictype(0, 11, 0)
◢ Output							
v	double	0	1024			Yes	numerictype(0, 11, 0)

Specializations

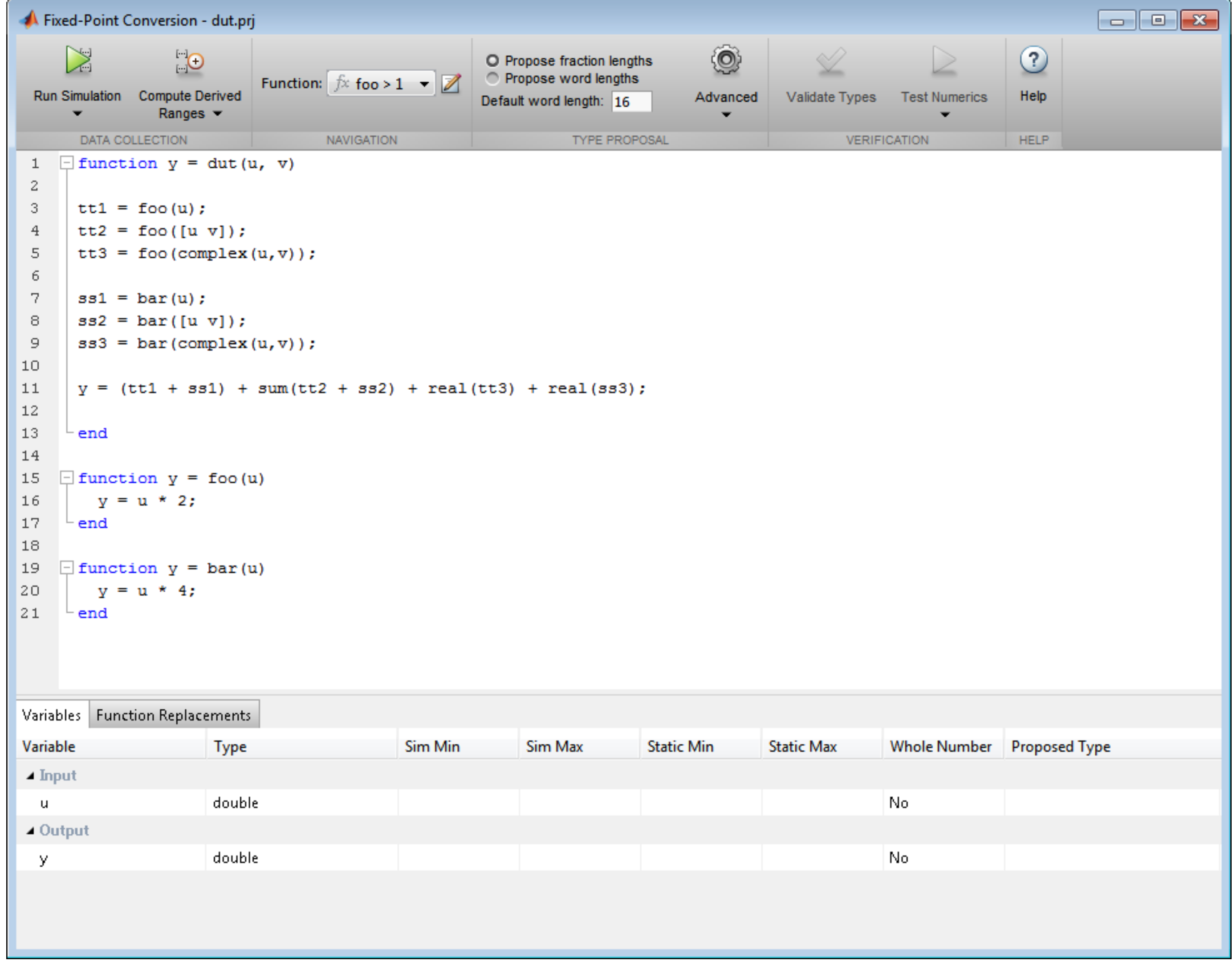
If a function is specialized, the tool lists each specialization and numbers them sequentially. For example, consider a function, `dut`, that calls subfunctions, `foo` and `bar`, multiple times with different input types.

```
function y = dut(u, v)

tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));
```

```
y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);

end

function y = foo(u)
  y = u * 2;
end

function y = bar(u)
  y = u * 4;
end
```



If you select a specialization, the app displays only the variables used by the specialization.



In the generated fixed-point code, the number of each fixed-point specialization matches the number in the Source Code list which makes it easy to trace between the floating-point and fixed-point versions of your code. For example, the generated fixed-point function for `foo > 1` is named `foo_s1`.

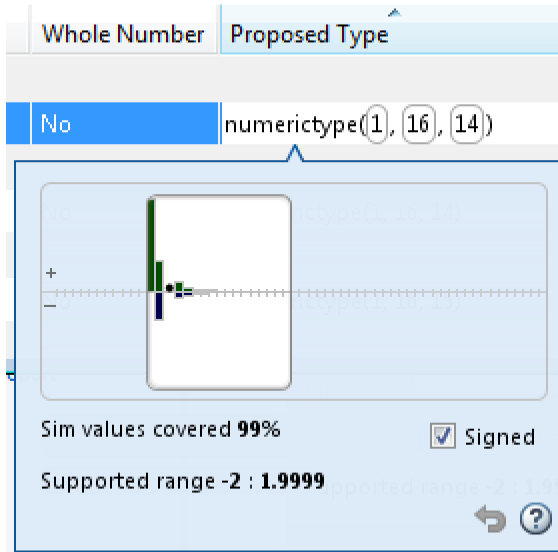
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                          %
%           Generated by MATLAB 8.4 and Fixed-Point Designer 4.3           %
%                                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen
function y = dut_fixpt(u, v)

fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full

tt1 = fi(foo_s1(u), 0, 5, 0, fm);
tt2 = fi(foo_s2([fi(u, 0, 5, 0, fm) v]), 0, 6, 0, fm);
tt3 = fi(foo_s3(complex(u,v)), 0, 6, 0, fm);

ss1 = fi(bar_s1(u), 0, 6, 0, fm);
ss2 = fi(bar_s2([fi(u, 0, 5, 0, fm) v]), 0, 7, 0, fm);
ss3 = fi(bar_s3(complex(u,v)), 0, 7, 0, fm);

y = fi((tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3), 0, 9, 0, fm);

end

function y = foo_s1(u)
  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Fu

  y = fi(u * fi(2, 0, 2, 0, fm), 0, 5, 0, fm);
end

function y = foo_s2(u)
  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Fu

  y = fi(u * fi(2, 0, 2, 0, fm), 0, 6, 0, fm);
end
```

Histogram

To log data for histograms, in the Fixed-Point Conversion window, click **Run Simulation** and select `Log data for histogram`, and then click the Run Simulation button.

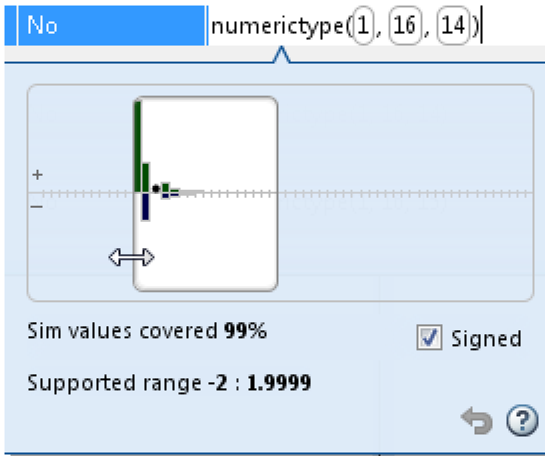
After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

The histogram provides the range of the proposed data type and the percentage of simulation values that the proposed data type covers. The bit weights are displayed along the X-axis, and the percentage of occurrences along the Y-axis. Each bin in the histogram corresponds to a bit in the binary word. For example, this histogram displays the range for a variable of type `numerictype(1,16,14)`.



You can view the effect of changing the proposed data types by:

- Dragging the edges of the bounding box in the histogram window to change the proposed data type.

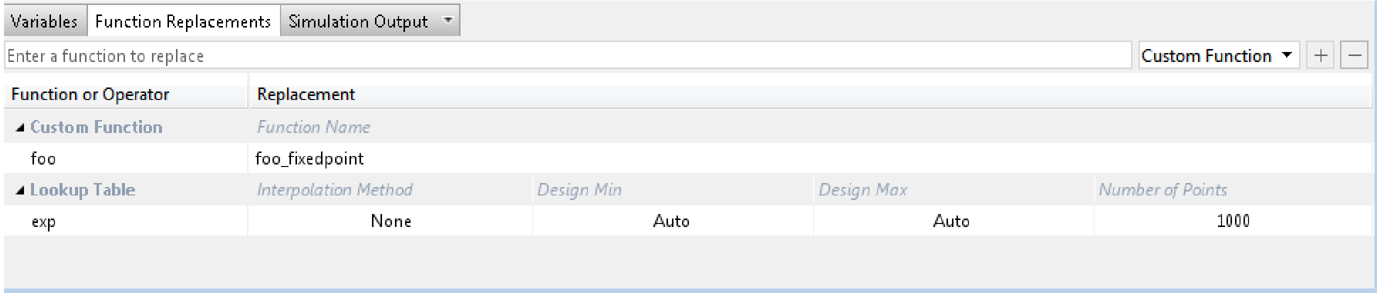


- Selecting or clearing **Signed**.

To revert to the types proposed by the automatic conversion, in the histogram window, click .

Function Replacements

If your MATLAB code uses functions that do not have fixed-point support, the tool lists these functions on the **Function Replacements** tab. You can choose to replace unsupported functions with a custom function replacement or with a lookup table.



You can add and remove function replacements from this list. If you enter a function replacements for a function, the replacement function is used when you build the project. If you do not enter a replacement, the tool uses the type specified in the original MATLAB code for the function.

Note Using this table, you can replace the names of the functions but you cannot replace argument patterns.

Validating Types

Selecting **Validate Types** validates the build using the proposed fixed-point data types. If the validation is successful, you are ready to test the numerical behavior of the fixed-point MATLAB algorithm.

If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. If errors or warning occur:

- On the **Variables** tab, inspect the proposed types and manually modified types to verify that they are valid.
- On the **Function Replacements** tab, verify that you have provided function replacements for unsupported functions.

Testing Numerics

After validating the proposed fixed-point data types, select **Test Numerics** to verify the behavior of the fixed-point MATLAB algorithm. By default, if you added a test bench to define inputs or run a simulation, the tool uses this test bench to test numerics. The tool compares the numerical behavior of the generated fixed-point MATLAB code with the original floating-point MATLAB code. If you select to log inputs and outputs for comparison plots, the tool generates an additional plot for each scalar output. This plot shows the floating-point and fixed-point results and the difference between them. For non-scalar outputs, only the error information is shown.

If the numerical results do not meet your desired accuracy after fixed-point simulation, modify fixed-point data type settings and repeat the type validation and numerical testing steps. You might have to iterate through these steps multiple times to achieve the desired results.

Detecting Overflows

When testing numerics, selecting **Use scaled doubles to detect overflows** enables overflow detection. When this option is selected, the conversion tool runs the simulation using scaled double versions of the proposed fixed-point types. Because scaled doubles store their data in double-

precision floating-point, they carry out arithmetic in full range. They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type.

If the tool detects overflows, on its Overflow tab, it provides:

- A list of variables and expressions that overflowed
- Information on how much each variable overflowed
- A link to the variables or expressions in the code window

Variables | Function Replacements | Overflows

	Function	Line	Description
⚠	overflow_fixpt	7	Overflow error in expression 'x'.
⚠	overflow_fixpt	7	Overflow error in expression 'y'.
⚠	overflow_fixpt	10	Overflow error in expression 'z'.
⚠	overflow_fixpt	10	Overflow error in expression 'z = fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'x'.
⚠	overflow_fixpt	10	Overflow error in expression 'x*y'.
⚠	overflow_fixpt	10	Overflow error in expression 'y'.
⚠	overflow_fixpt	11	Overflow error in expression 'z'.

If your original algorithm uses scaled doubles, the tool also provides overflow information for these expressions.

See Also

"Detect Overflows"

Custom Plot Functions

The Fixed-Point Conversion tool provides a default time series based plotting function. The conversion process uses this function at the test numerics step to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. For example, plots that show eye diagrams and bit error differences are more suitable in the communications domain and histogram difference plots are more suitable in image processing designs.

You can choose to use a custom plot function at the test numerics step. The Fixed-Point Conversion tool facilitates custom plotting by providing access to the raw logged input and output data before and after fixed-point conversion. You supply a custom plotting function to visualize the differences between the floating-point and fixed-point results. If you specify a custom plot function, the fixed-point conversion process calls the function for each input and output variable, passes in the name of the variable and the function that uses it, and the results of the floating-point and fixed-point simulations.

Your function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.

  Use this information to:

  - Customize plot headings and axes.
  - Choose which variables to plot.
  - Generate different error metrics for different output variables.
- A cell array to hold the logged floating-point values for the variable.

  This cell array contains values observed during floating-point simulation of the algorithm during the test numerics phase. You might need to reformat this raw data.
- A cell array to hold the logged values for the variable after fixed-point conversion.

  This cell array contains values observed during fixed-point simulation of the converted design.

For example, `function customComparisonPlot(varInfo, floatVarVals, fixedPtVarVals)`.

To use a custom plot function, in the Fixed-Point Conversion tool, select **Advanced**, and then set **Custom plot function** to the name of your plot function.

In the programmatic workflow, set the `coder.FixPtConfig` configuration object `PlotFunction` property to the name of your plot function. See “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 4-26.

Edit Configuration Parameters for Fixed-Point Code Generation

In this section...

"Create and Modify Configuration Objects" on page 4-24
"Additional Functionalities" on page 4-24

After you have created a fixed-point code generation configuration object at the command line, you can modify the properties of the object interactively by using the Fixed-point configuration setting dialog box.

Create and Modify Configuration Objects

1. Create a fixed-point configuration object.

   ```
   cfg = coder.config('fixpt');
   ```
2. Open the fixed-point configuration settings dialog box by using one of these methods:

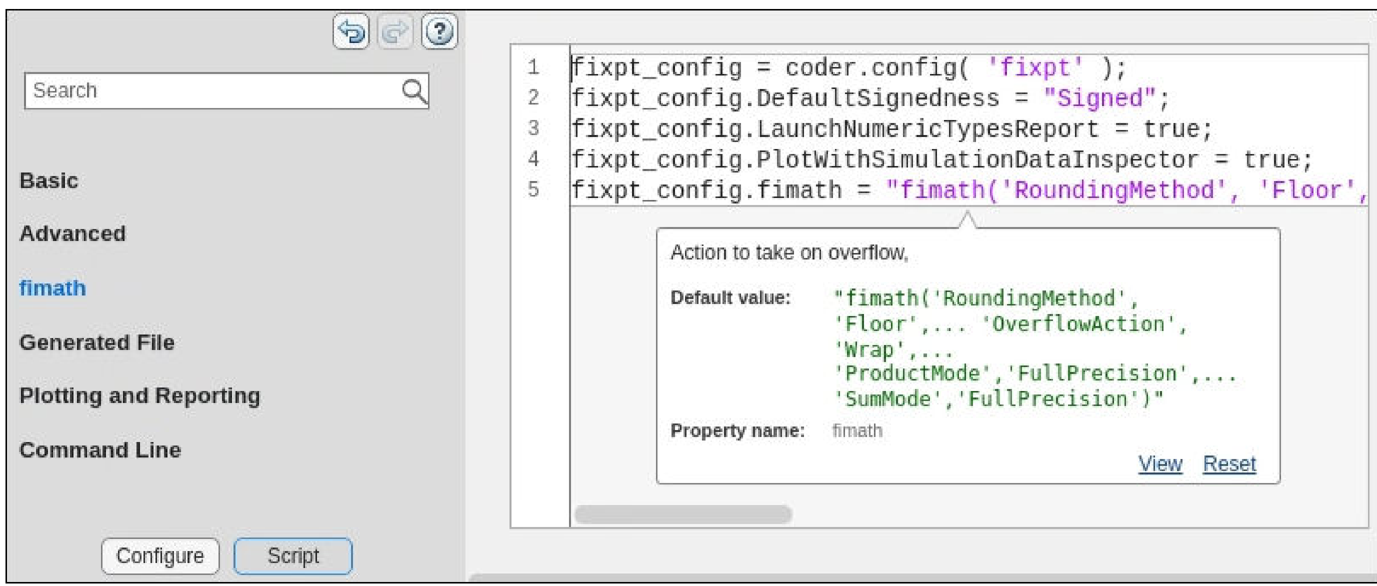
   - In the MATLAB workspace, double-click the configuration object `cfg`.
   - At the MATLAB command prompt, open the configuration object `cfg`.

     ```
     open cfg
     ```
3. In the dialog box, modify configuration parameters as required.

Additional Functionalities

Fixed-point configuration settings dialog box provides these functionalities.

- *Search*: When you search for a string, you see the filtered results across all settings categories. The search string might be present in a setting name, the name of an option for a setting, or in a tooltip.
- *Informative tooltips*: The tooltip for each setting contains the configuration object property name, a **Help** link for that property, and the name of any additional products that are required for using the property. If the property is disabled, the tooltip also contains links to other properties that you must set to enable this property. You can make that change in the tooltip itself.
- *Settings with nondefault values*: The dialog box shows settings that have nondefault values in bold font. To reset such a setting to its default values, click the **Reset** button in the tooltip.
- *Generate equivalent script*: You can view the command-line script that produces your current settings by clicking the **Script** button located at the bottom of the list of categories on the left side of the window. You can switch from script mode back to interactive mode by clicking the **Configure** button.



See Also

`coder.FixPtConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

Visualize Differences Between Floating-Point and Fixed-Point Results

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler

  See `https://www.mathworks.com/support/compilers/current_release/`.

  You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

Inspect Example Files

Type	Name	Description
Function code	`myFilter.m`	Entry-point MATLAB function
Test file	`myFilterTest.m`	MATLAB script that tests `myFilter.m`
Plotting function	`plotDiff.m`	Custom plot function
MAT-file	`filterData.mat`	Data to filter.

The myFilter Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
  b = complex(zeros(1,16));
  h = complex(zeros(1,16));
  h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

The myFilterTest File

```
% load data
data = load('filterData.mat');
d = data.symbols;

for idx = 1:4000
    y = myFilter(d(idx));
end
```

The plotDiff Function

```
% varInfo - structure with information about
% the variable. It has the following fields
%           i) name
%           ii) functionName
% floatVals - cell array of logged original values
% for the 'varInfo.name' variable
% fixedVals - cell array of logged values for
% the 'varInfo.name' variable after Fixed-Point conversion.
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % escape the '_'s because plot titles treat these as subscripts
    escapedVarName = regexprep(varName,'_','\\_');
    escapedFcnName = regexprep(fcnName,'_','\\_');

    % flatten the values
    flatFloatVals = floatVals(1:end);
    flatFixedVals = fixedVals(1:end);

    % build Titles
    floatTitle = [escapedFcnName ' > ' 'float : ' escapedVarName];
    fixedTitle = [escapedFcnName ' > ' 'fixed : ' escapedVarName];

    data = load('filterData.mat');

    switch varName
        case 'y'
            x_vec = data.symbols;

            figure('Name','Comparison plot','NumberTitle','off');

            % plot floating point values
            y_vec = flatFloatVals;
            subplot(1, 2, 1);
            plotScatter(x_vec, y_vec, 100, floatTitle);

            % plot fixed point values
            y_vec = flatFixedVals;
            subplot(1, 2, 2);
            plotScatter(x_vec, y_vec, 100, fixedTitle);

        otherwise
            % Plot only output 'y' for this example, skip the rest
    end

end

function plotScatter(x_vec, y_vec, n, figTitle)
    % plot the last n samples
    x_plot = x_vec(end-n+1:end);
    y_plot = y_vec(end-n+1:end);

    hold on
    scatter(real(x_plot),imag(x_plot), 'bo');
```

```
    hold on
    scatter(real(y_plot),imag(y_plot), 'rx');

    title(figTitle);
end
```

Set Up Configuration Object

1. Create a `coder.FixptConfig` object.

   ```
   fxptcfg = coder.config('fixpt');
   ```
2. Specify the test file name and custom plot function name. Enable logging and numerics testing.

   ```
   fxptcfg.TestBenchName = 'myFilterTest';
   fxptcfg.PlotFunction = 'plotDiff';
   fxptcfg.TestNumerics = true;
   fxptcfg.LogIOForComparisonPlotting = true;
   fxptcfg.DefaultWordLength = 16;
   ```

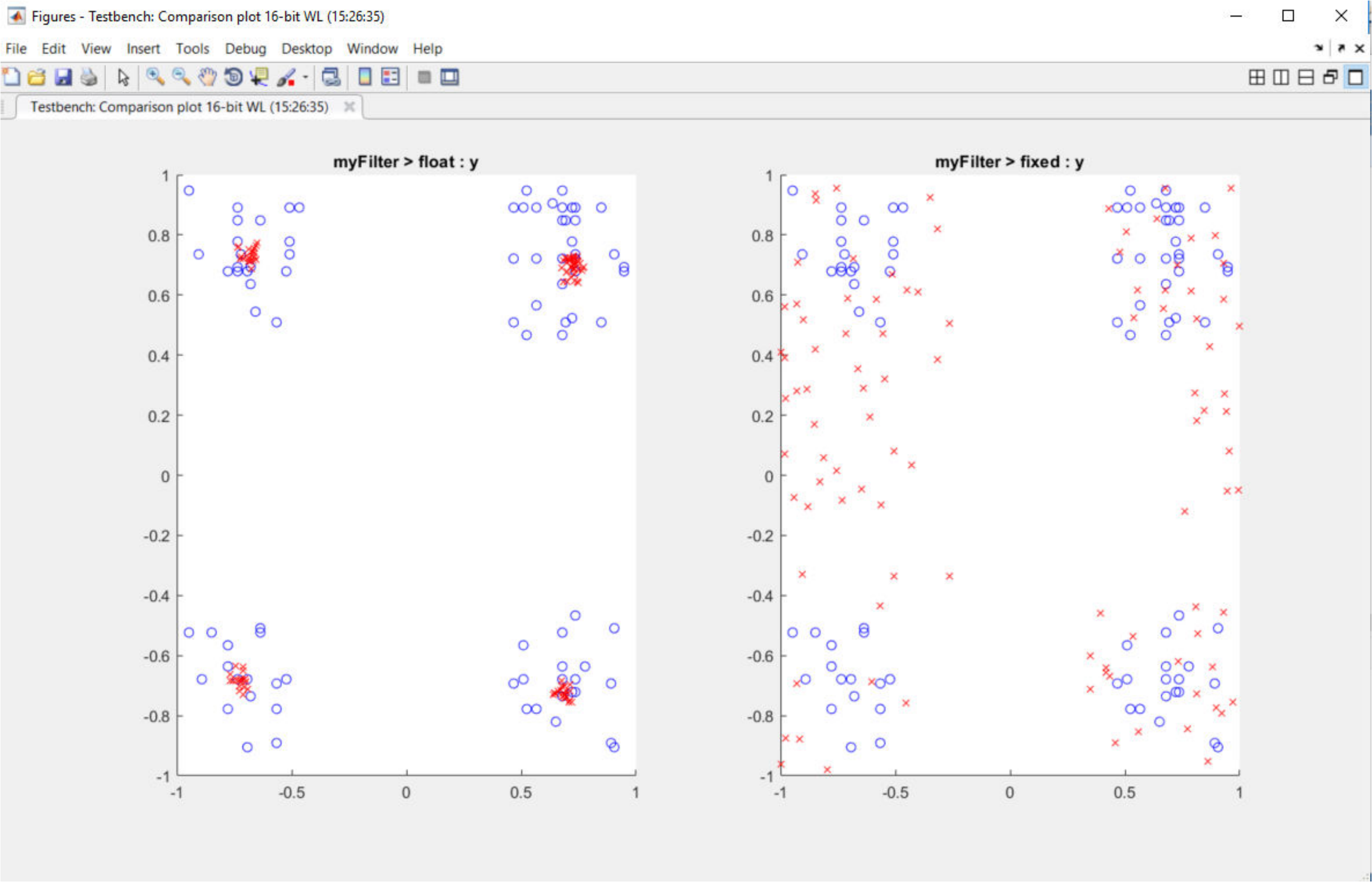
Convert to Fixed Point

Convert the floating-point MATLAB function, `myFilter`, to fixed-point MATLAB code. You do not need to specify input types for the `codegen` command because it infers the types from the test file.

```
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The conversion process generates fixed-point code using a default word length of `16` and then runs a fixed-point simulation by running the `myFilterTest.m` function and calling the fixed-point version of `myFilter.m`.

Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the conversion process uses this function to generate the comparison plot.

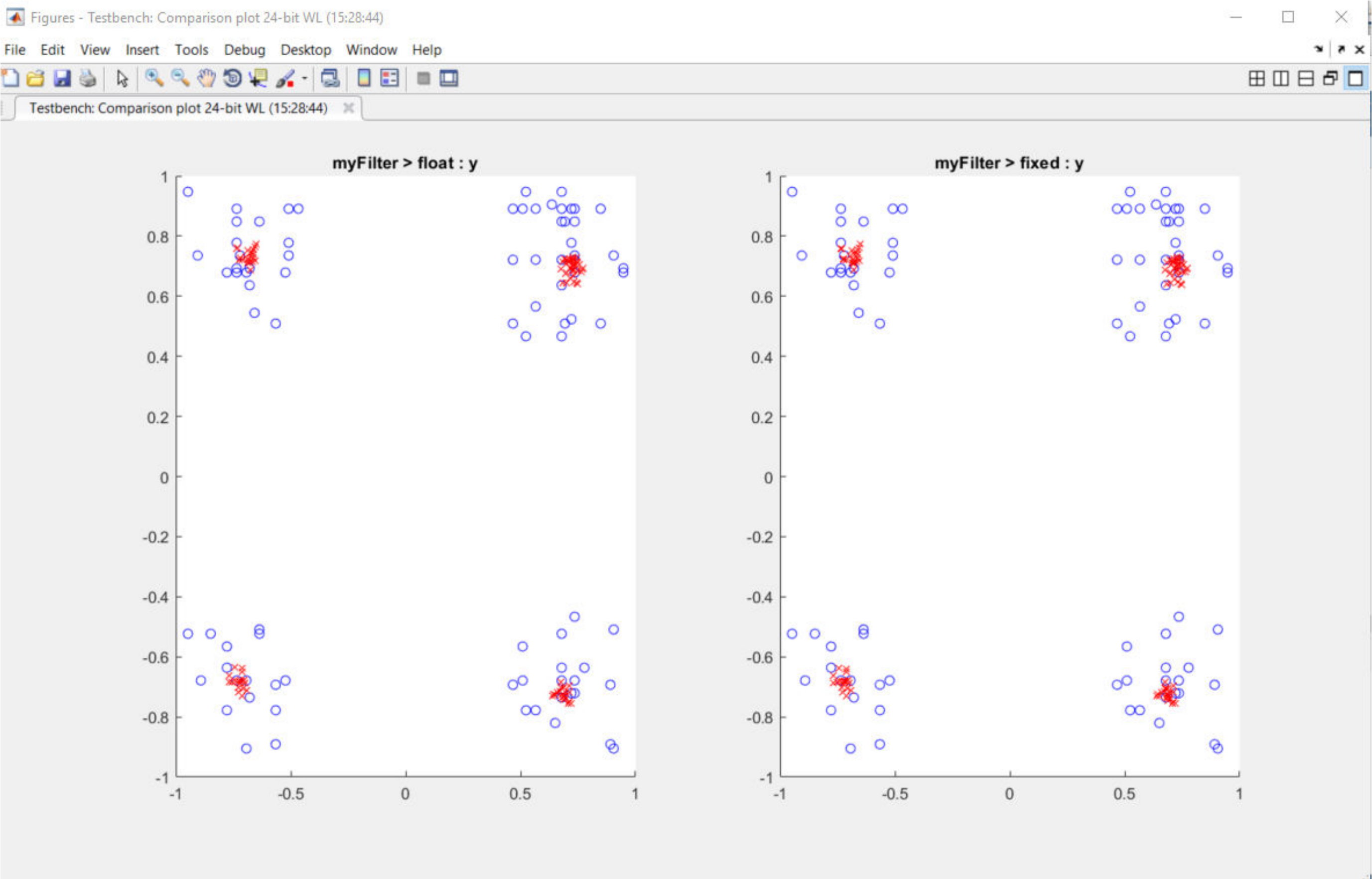


The plot shows that the fixed-point results do not closely match the floating-point results.

Increase the word length to 24 and then convert to fixed point again.

```
fxptcfg.DefaultWordLength = 24;
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The increased word length improved the results. This time, the plot shows that the fixed-point results match the floating-point results.


Figures - Testbench: Comparison plot 24-bit WL (15:28:44)
File Edit View Insert Tools Debug Desktop Window Help
Testbench: Comparison plot 24-bit WL (15:28:44)
myFilter > float : y
myFilter > fixed : y

Inspecting Data Using the Simulation Data Inspector

In this section...

"What Is the Simulation Data Inspector?" on page 4-31
"Import Logged Data" on page 4-31
"Export Logged Data" on page 4-31
"Group Signals" on page 4-31
"Run Options" on page 4-31
"Create Report" on page 4-32
"Comparison Options" on page 4-32
"Enabling Plotting Using the Simulation Data Inspector" on page 4-32
"Save and Load Simulation Data Inspector Sessions" on page 4-32

What Is the Simulation Data Inspector?

The Simulation Data Inspector allows you to view data logged during the fixed-point conversion process. You can use it to inspect and compare the inputs and outputs to the floating-point and fixed-point versions of your algorithm.

For fixed-point conversion, there is no programmatic interface for the Simulation Data Inspector.

Import Logged Data

Before importing data into the Simulation Data Inspector, you must have previously logged data to the base workspace or to a MAT-file.

Export Logged Data

The Simulation Data Inspector provides the capability to save data collected by the fixed-point conversion process to a MAT-file that you can later reload. The format of the MAT-file is different from the format of a MAT-file created from the base workspace.

Group Signals

You can customize the organization of your logged data in the Simulation Data Inspector **Runs** pane. By default, data is first organized by run. You can then organize your data by logged variable or no hierarchy.

Run Options

You can configure the Simulation Data Inspector to:

- Append New Runs

  In the Run Options dialog box, the default is set to add new runs to the bottom of the run list. To append new runs to the top of the list, select **Add new runs at top**.
- Specify a Run Naming Rule

  To specify run naming rules, in the Simulation Data Inspector toolbar, click **Run Options**.

Create Report

You can create a report of the runs or comparison plots. Specify the name and location of the report file. By default, the Simulation Data Inspector overwrites existing files. To preserve existing reports, select **If report exists, increment file name to prevent overwriting**.

Comparison Options

To change how signals are matched when runs are compared, specify the **Align by** and **Then by** parameters and then click **OK**.

Enabling Plotting Using the Simulation Data Inspector

To enable the Simulation Data Inspector, see "Enable Plotting Using the Simulation Data Inspector" on page 4-33.

Save and Load Simulation Data Inspector Sessions

If you have data in the Simulation Data Inspector and you want to archive or share the data to view in the Simulation Data Inspector later, save the Simulation Data Inspector session. When you save a Simulation Data Inspector session, the MAT-file contains:

- All runs, data, and properties from the **Runs** and **Comparisons** panes.
- Check box selection state for data in the **Runs** pane.

Save a Session to a MAT-File

1. On the **Visualize** tab, click **Save**.
2. Browse to where you want to save the MAT-file to, name the file, and click **Save**.

Load a Saved Simulation Data Inspector Simulation

1. On the **Visualize** tab, click **Open**.
2. Browse, select the MAT-file saved from the Simulation Data Inspector, and click **Open**.
3. If data in the session is plotted on multiple subplots, on the **Format** tab, click **Subplots** and select the subplot layout.

Enable Plotting Using the Simulation Data Inspector

In this section...
"From the UI" on page 4-33
"From the Command Line" on page 4-33

From the UI

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point logged input and output data. In the Fixed-Point Conversion tool:

1. Click **Advanced**.
2. In the Advanced Settings dialog box, set **Plot with Simulation Data Inspector** to `Yes`.
3. At the Test Numerics stage in the conversion process, click **Test Numerics**, select `Log inputs and outputs for comparison plots`, and then click .

For an example, see "Propose Fixed-Point Data Types Based on Derived Ranges".

From the Command Line

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point input and output data logged using the function. At the MATLAB command line:

1. Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'dti_test';
```

2. Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

```
fixptcfg.TestNumerics = true;
fixptcfg.LogIOForComparisonPlotting = true;
fixptcfg.PlotWithSimulationDataInspector = true;
```

3. Generate fixed-point MATLAB code using `codegen`.

```
codegen -float2fixed fixptcfg -config cfg dti
```

For an example, see "Propose Fixed-Point Data Types Based on Derived Ranges".

Replacing Functions Using Lookup Table Approximations

The software provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. These functions must be on the MATLAB path.

You can use this capability to handle functions that are not supported for fixed point and to replace your own custom functions. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. You can control the interpolation method and number of points in the lookup table. By adjusting these settings, you can tune the behavior of replacement function to match the behavior of the original function as closely as possible.

The fixed-point conversion process generates one lookup table approximation per call site of the function that needs replacement.

To use lookup table approximations, see:

- `coder.approximation`
- “Replace the exp Function with a Lookup Table” on page 4-41
- “Replace a Custom Function with a Lookup Table” on page 4-35

Replace a Custom Function with a Lookup Table

In this section...
"Using the HDL Coder App" on page 4-35
"From the Command Line" on page 4-38

With HDL Coder, you can generate lookup table approximations for functions that do not support fixed-point types, and replace your own functions. To replace a custom function with a Lookup Table, use the HDL Coder app, or the `fiaccel codegen` function.

Using the HDL Coder App

This example shows how to replace a custom function with a Lookup Table using the HDL Coder app.

Create Algorithm and Test Files

In a local, writable folder:

1. Create a MATLAB function, `custom_fcn`, which is the function that you want to replace.

```
function y = custom_fcn(x)
  y = 1./(1+exp(-x));
end
```

2. Create a wrapper function that calls `custom_fcn`.

```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

3. Create a test file, `custom_test`, which uses `call_custom_fcn`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
   y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Create and Set up a HDL Coder Project

1. Navigate to the work folder that contains the file for this example.
2. To open the HDL Coder app, in the MATLAB command prompt, enter `hdlcoder`. Set **Name** to `custom_project.prj` and click **OK**. The project opens in the MATLAB workspace.
3. In the project window, on the **MATLAB Function** tab, click the **Add MATLAB function** link. Browse to the file `call_custom_fcn.m`, and then click **OK** to add the file to the project.

Define Input Types

1. To define input types for `call_custom_fcn.m`, on the **MATLAB Function** tab, click **Autodefine types**.
2. Add `custom_test` as a test file, and then click **Run**.

   From the test file, HDL Coder determines that `x` is a scalar double.

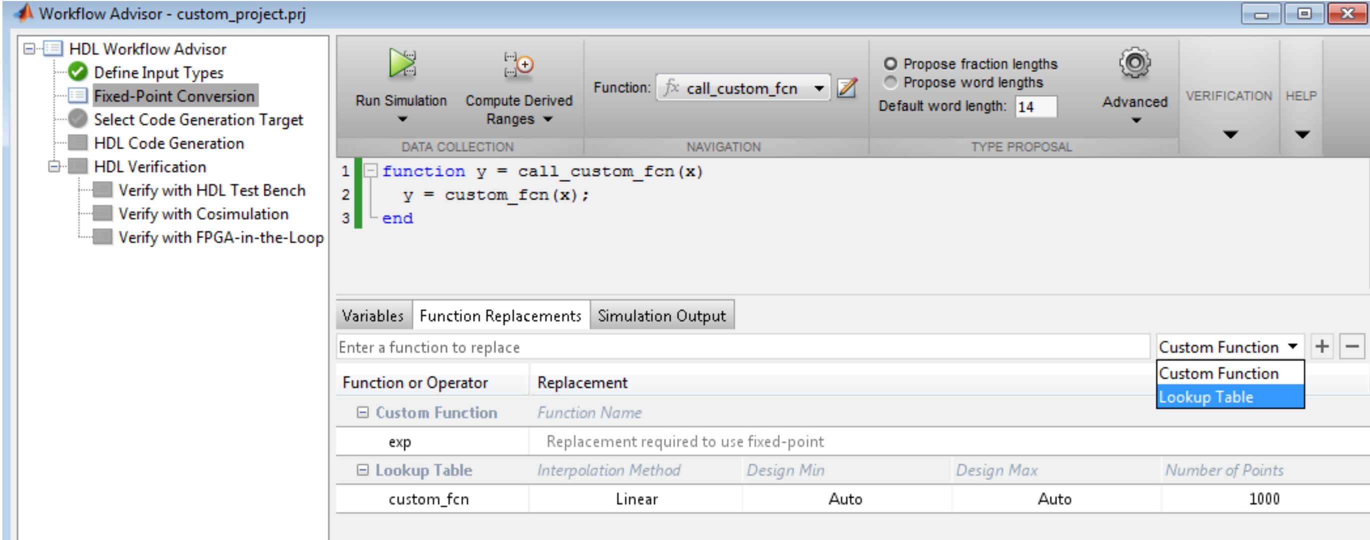
3 Click **Use These Types**.

Replace custom_fcn with Lookup Table

1 To open the HDL Workflow Advisor, click **Workflow Advisor**, and in the Workflow Advisor window, click **Fixed-Point Conversion**.

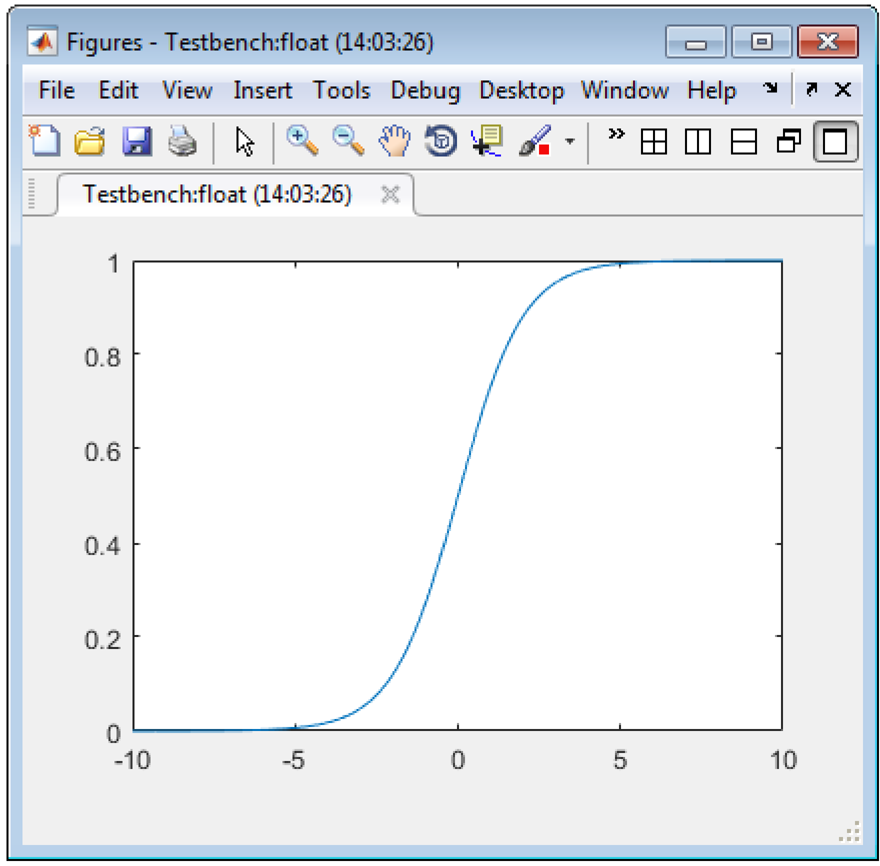
2 To replace `custom_fcn` with a Lookup Table, on the **Function Replacements** tab, enter `custom_fcn`, select `Lookup Table`, and then click +.

   By default, the lookup table uses linear interpolation, 1000 points, and design minimum and maximum values that the app detects by running a simulation or computing derived ranges.



3 Under **Run Simulation**, select `Log data for histogram`, and then click **Run Simulation**. Verify that `custom_test` file is selected as the test file.

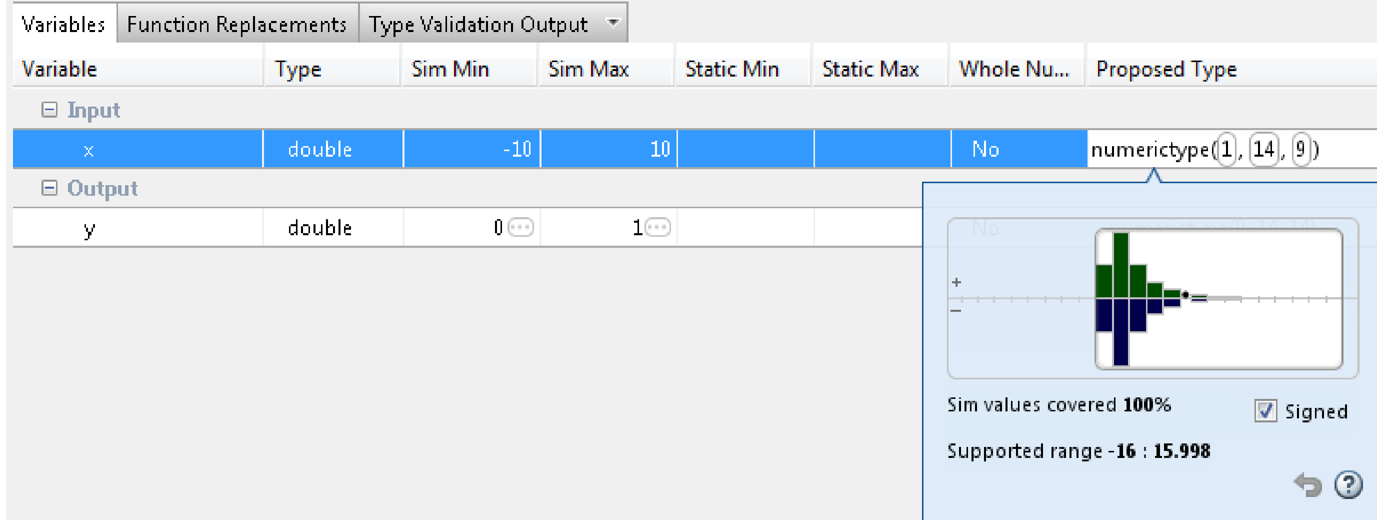
   The simulation runs and the tool displays simulation minimum and maximum ranges on the **Variables** tab. HDL Coder plots the simulation results in the MATLAB Editor.



Validate Fixed-Point Types

1 In the **Proposed Type** column, verify that the fixed-point types proposed by software cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.

   The histogram provides range information and the percentage of simulation range that the proposed data type covers.



2 To validate the build by using the proposed types, click **Validate Types**.

   The software validates the proposed types and generates a fixed-point code, `call_custom_fcn_fixpt`.

3 To view the generated fixed-point code, click the `call_custom_fcn_fixpt` link.

   The generated fixed-point function, `call_custom_fcn_fixpt.m`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end

function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor',...
         'OverflowAction', 'Wrap',...
         'ProductMode','FullPrecision',...
         'MaxProductWordLength', 128,...
         'SumMode','FullPrecision',...
         'MaxSumWordLength', 128);
end
```

From the Command Line

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler

  See `https://www.mathworks.com/support/compilers/current_release/`.

  You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

Create a MATLAB function, `custom_fcn.m`. This is the function that you want to replace.

```
function y = custom_fcn(x)
  y = 1./(1+exp(-x));
end
```

Create a wrapper function that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

Create a test file, `custom_test.m`, that uses `call_custom_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
```

```
   y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Create a function replacement configuration object to approximate `custom_fcn`. Specify the function handle of the custom function and set the number of points to use in the lookup table to 50.

```
q = coder.approximation('Function','custom_fcn',...
                        'CandidateFunction',@custom_fcn,...
                        'NumberOfPoints',50);
```

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'custom_test';
fixptcfg.TestNumerics = true;
fixptcfg.addApproximation(q);
```

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg call_custom_fcn
```

`codegen` generates fixed-point MATLAB code in `call_custom_fcn_fixpt.m`.

To view the generated fixed-point code, click the link to `call_custom_fcn_fixpt`.

The generated code contains a lookup table approximation, `replacement_custom_fcn`, for the `custom_fcn` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. The lookup table uses 50 points as specified. By default, it uses linear interpolation and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `call_custom_fcn_fixpt`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

See Also

`coder.approximation`

Related Examples

- “Replace the exp Function with a Lookup Table” on page 4-41

More About

- "Replacing Functions Using Lookup Table Approximations" on page 4-34

Replace the exp Function with a Lookup Table

With HDL Coder, you can handle functions that are not supported for fixed point and replace your own functions. To replace a custom function with a Lookup Table, use the HDL Coder App, or the `fiaccel` `codegen` function.

From the UI

This example shows how to replace a custom function with a Lookup Table using the HDL Coder app.

Create Algorithm and Test Files

In a local, writable folder:

1 Create a MATLAB function, `custom_fcn`, which is the function that you want to replace.

```
function y = custom_fcn(x)
  y = 1./(1+exp(-x));
end
```

2 Create a wrapper function that calls `custom_fcn`.

```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

3 Create a test file, `custom_test`, which uses `call_custom_fcn`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
   y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Create and Set up a HDL Coder Project

1 Navigate to the work folder that contains the file for this example.

2 To open the HDL Coder app, in the MATLAB command prompt, enter `hdlcoder`. Set **Name** to `custom_project.prj` and click **OK**. The project opens in the MATLAB workspace.

3 In the project window, on the **MATLAB Function** tab, click the **Add MATLAB function** link. Browse to the file `call_custom_fcn.m`, and then click **OK** to add the file to the project.

Define Input Types

1 To define input types for `call_custom_fcn.m`, on the **MATLAB Function** tab, click **Autodefine types**.

2 Add `custom_test` as a test file, and then click **Run**.

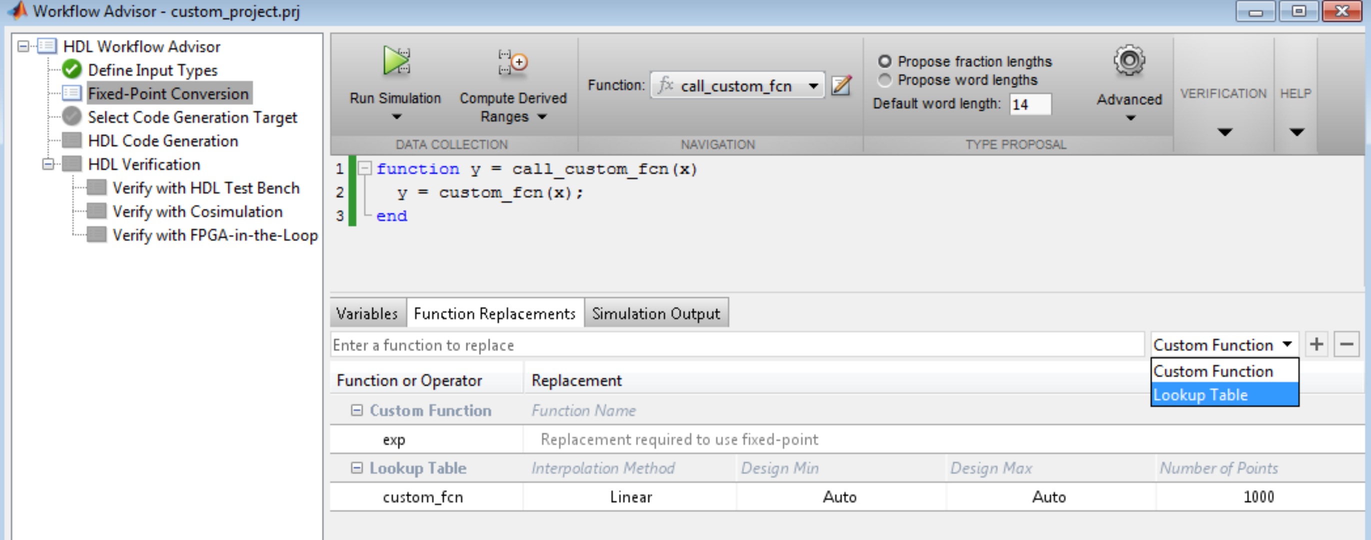
   From the test file, HDL Coder determines that `x` is a scalar double.

3 Click **Use These Types**.

Replace custom_fcn with Lookup Table

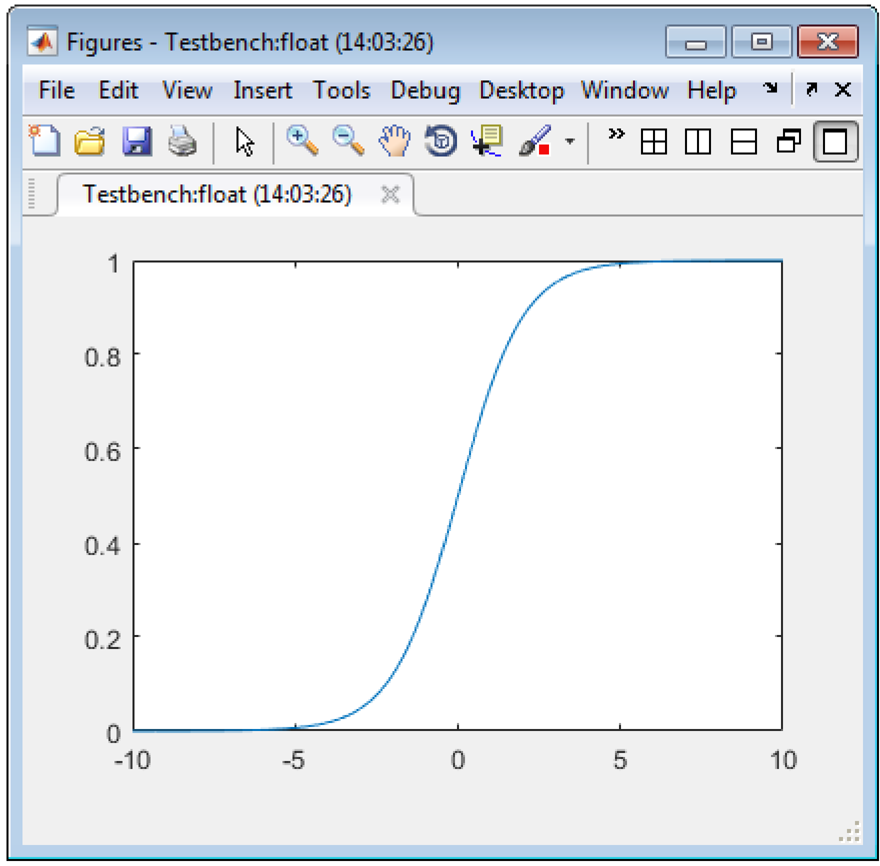
1. To open the HDL Workflow Advisor, click **Workflow Advisor**, and in the Workflow Advisor window, click **Fixed-Point Conversion**.
2. To replace `custom_fcn` with a Lookup Table, on the **Function Replacements** tab, enter `custom_fcn`, select `Lookup Table`, and then click +.

   By default, the lookup table uses linear interpolation, 1000 points, and design minimum and maximum values that the app detects by running a simulation or computing derived ranges.



3. Under **Run Simulation**, select `Log data for histogram`, and then click **Run Simulation**. Verify that `custom_test` file is selected as the test file.

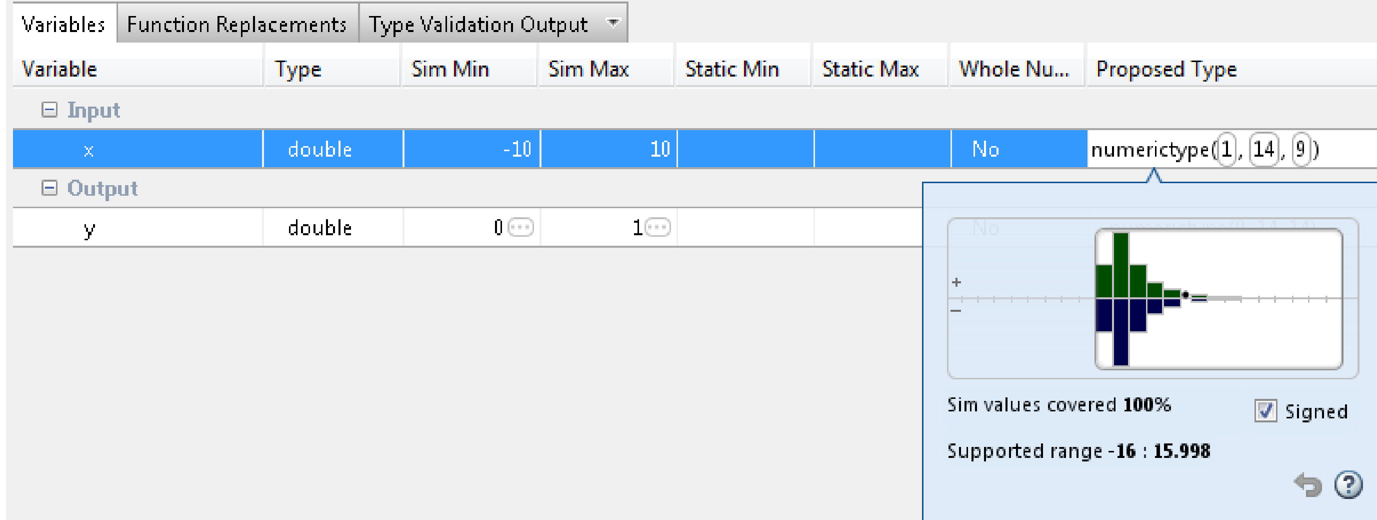
   The simulation runs and the tool displays simulation minimum and maximum ranges on the **Variables** tab. HDL Coder plots the simulation results in the MATLAB Editor.



Validate Fixed-Point Types

1 In the **Proposed Type** column, verify that the fixed-point types proposed by software cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.

   The histogram provides range information and the percentage of simulation range that the proposed data type covers.



2 To validate the build by using the proposed types, click **Validate Types**.

   The software validates the proposed types and generates a fixed-point code, `call_custom_fcn_fixpt`.

3 To view the generated fixed-point code, click the `call_custom_fcn_fixpt` link.

   The generated fixed-point function, `call_custom_fcn_fixpt.m`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end

function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor',...
         'OverflowAction', 'Wrap',...
         'ProductMode','FullPrecision',...
         'MaxProductWordLength', 128,...
         'SumMode','FullPrecision',...
         'MaxSumWordLength', 128);
end
```

From the Command Line

This example shows how to replace the `exp` function with a lookup table approximation in the generated fixed-point code using the function.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler

  See `https://www.mathworks.com/support/compilers/current_release/`.

  You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

Create Algorithm and Test Files

1 Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

```
function y = my_fcn(x)
  y = exp(x);
end
```

2 Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
   y(itr) = my_fcn( x(itr) );
```

```
    end
    plot( x, y );
```

Configure Approximation

Create a function replacement configuration object to approximate the `exp` function, using the default settings of linear interpolation and 1000 points in the lookup table.

```
q = coder.approximation('exp');
```

Set Up Configuration Object

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'my_fcn_test';
fixptcfg.TestNumerics = true;
fixptcfg.DefaultWordLength = 16;
fixptcfg.addApproximation(q);
```

Convert to Fixed Point

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg my_fcn
```

View Generated Fixed-Point Code

To view the generated fixed-point code, click the link to `my_fcn_fixpt`.

The generated code contains a lookup table approximation, `replacement_exp`, for the `exp` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `my_fcn_fixpt`, calls this approximation instead of calling `exp`.

```
function y = my_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_exp(x), 0, 16, 1, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

See Also

`coder.approximation`

Related Examples

- "Replace a Custom Function with a Lookup Table" on page 4-35

More About

- "Replacing Functions Using Lookup Table Approximations" on page 4-34

Data Type Issues in Generated Code

Within the fixed-point conversion report, you have the option to highlight MATLAB code that results in double, single, or expensive fixed-point operations. Consider enabling these checks when trying to achieve a strict single, or fixed-point design.

These checks are disabled by default.

Enable the Highlight Option in a Project

1. Open the **Settings** menu.
2. Under **Plotting and Reporting**, set **Highlight potential data type issues** to `Yes`.

Enable the Highlight Option at the Command Line

1. Create a fixed-point code configuration object:

   ```
   cfg = coder.config('fixpt');
   ```
2. Set the `HighlightPotentialDataTypeIssues` property of the configuration object to `true`.

   ```
   cfg.HighlightPotentialDataTypeIssues = true;
   ```

Stowaway Doubles

When trying to achieve a strict-single or fixed-point design, manual inspection of code can be time-consuming and error prone. This check highlights all expressions that result in a double operation.

Stowaway Singles

This check highlights all expressions that result in a single operation.

Expensive Fixed-Point Operations

The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, expensive rounding, expensive comparison, or multiword operations. For more information on optimizing generated fixed-point code, see “Tips for Making Generated Code More Efficient”.

Cumbersome Operations

Cumbersome operations most often occur due to insufficient range of output. Avoid inputs to a multiply or divide operation that has word lengths larger than the base integer type of your processor. Operations with larger word lengths can be handled in software, but this approach requires much more code and is much slower.

Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method. This check identifies expensive rounding operations in multiplication and division.

Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, when comparing an unsigned integer to a signed integer, one of the inputs must first be cast to the signedness of the other before the comparison operation can be performed. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

Multiword Operations

Multiword operations can be inefficient on hardware. When an operation has an input or output data type larger than the largest word size of your processor, the generated code contains multiword operations. You can avoid multiword operations in the generated code by specifying local `fimath` properties for variables. You can also manually specify input and output word lengths of operations that generate multiword code.

Working with Fixed-Point Code

This example shows HDL code generation from a fixed-point MATLAB® design that is ready for code generation.

Introduction

The MATLAB code used in the example is an implementation of viterbi decoder modeled using fixed-point constructs.

```
design_name = 'mlhdlc_viterbi';
testbench_name = 'mlhdlc_viterbi_tb';
```

1. MATLAB Design: mlhdlc_viterbi
2. MATLAB testbench: mlhdlc_viterbi_tb

Open the design function mlhdlc_viterbi by clicking on the above link to notice the use of Fixed-Point Designer functions:

1. use of 'fi', 'numerictype', and 'fimath' for modeling fixed-point data types
2. use of 'bitget', 'bitsliceget', 'bitconcat' for modeling bit-wise operations

Create a New HDL Coder™ Project

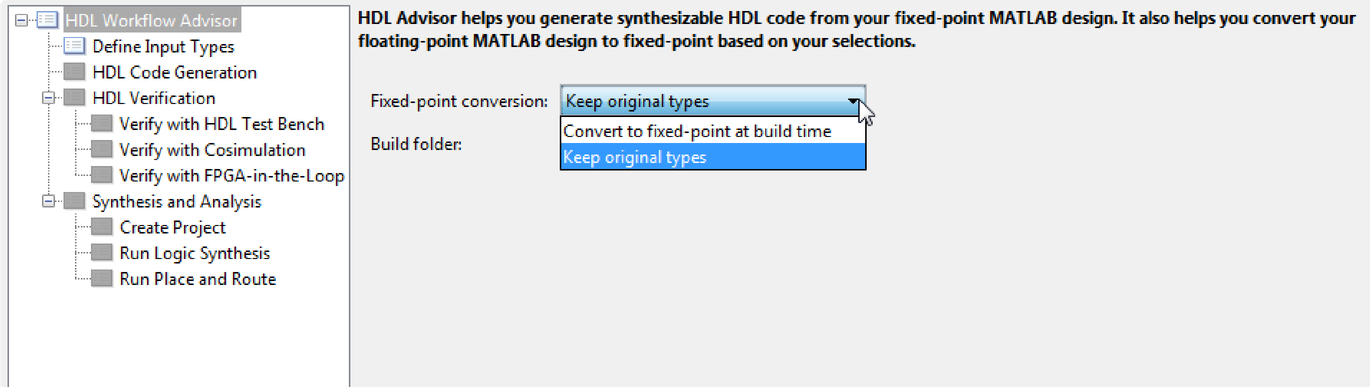
```
coder -hdlcoder -new fixpt_codegen
```

Next, add the file 'mlhdlc_viterbi.m' to the project as the MATLAB Function and 'mlhdlc_viterbi_tb.m' as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Skip Fixed-Point Conversion

Launch the HDL Advisor and choose 'Keep original types' on the option 'Fixed-point conversion:'.



The Floating-point to fixed-point conversion related step is removed from the workflow tree when we skip the conversion.

If your design is in floating-point, follow the instructions in “Floating-Point to Fixed-Point Conversion” on page 4-51 and convert your design to fixed-point before moving onto the HDL code generation steps.

Run HDL Code Generation

Right click on the 'Code Generation' step and choose the option 'Run this task' to run all code generation step directly.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

Try More Code Generation Options

As this is a large design with considerable number of functions you can try the option 'Generate instantiable code for functions' in the Advanced tab.

Re-examine the generated HDL code and compare it with the previous step.

Floating-Point to Fixed-Point Conversion

This example shows how to start with a floating-point design in MATLAB®, iteratively converge on an efficient fixed-point design in MATLAB, and verify the numerical accuracy of the generated fixed-point design.

Signal processing applications for reconfigurable platforms require algorithms that are typically specified using floating-point operations. However, for power, cost, and performance reasons, they are usually implemented with fixed-point operations either in software for DSP cores or as special-purpose hardware in FPGAs. Fixed-point conversion can be very challenging and time-consuming, typically demanding 25 to 50 percent of the total design and implementation time. Automated tools can simplify and accelerate the conversion process.

For software implementations, the aim is to define an optimized fixed-point specification which minimizes the code size and the execution time for a given computation accuracy constraint. This optimization is achieved through the modification of the binary point location (for scaling) and the selection of the data word length according to the different data types supported by the target processor.

For hardware implementations, the complete architecture can be optimized. An efficient implementation will minimize both the area used and the power consumption. Thus, the conversion process goal typically is focused around minimizing the operator word length.

The floating-point to fixed-point workflow is currently integrated in the HDL Workflow Advisor as described in “Get Started with MATLAB to HDL Workflow”.

Introduction

The floating-point to fixed-point conversion workflow in HDL Coder™ includes the following steps:

1. Verify that the floating-point design is compatible with code generation.
2. Compute fixed-point types based on the simulation of the testbench.
3. Generate readable and traceable fixed-point MATLAB code by applying proposed types.
4. Verify the generated fixed-point design.
5. Compare the numerical accuracy of the generated fixed-point code with the original floating point code.

MATLAB Design

The MATLAB code used in this example is a simple second-order direct-form 2 transposed filter. This example also contains a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_df2t_filter';
testbench_name = 'mlhdlc_df2t_filter_tb';
```

Examine the MATLAB design.

```
type(design_name);
```

```
%#codegen
function y = mlhdlc_df2t_filter(x)
```

```
%   Copyright 2011-2015 The MathWorks, Inc.

persistent z;
if isempty(z)
    % Filter states as a column vector
    z = zeros(2,1);
end

% Filter coefficients as constants
b = [0.29290771484375   0.585784912109375  0.292907714843750];
a = [1.0                0.0                0.171600341796875];

y    =  b(1)*x + z(1);
z(1) = (b(2)*x + z(2)) - a(2) * y;
z(2) =  b(3)*x - a(3) * y;

end
```

For the floating-point to fixed-point workflow, it is desirable to have a complete testbench. The quality of the proposed fixed-point data types depends on how well the testbench covers the dynamic range of the design with the desired accuracy.

For details on requirements for floating-point design and the testbench, see **Floating-Point Design Structure** structure section of “Working with Generated Fixed-Point Files” on page 4-68.

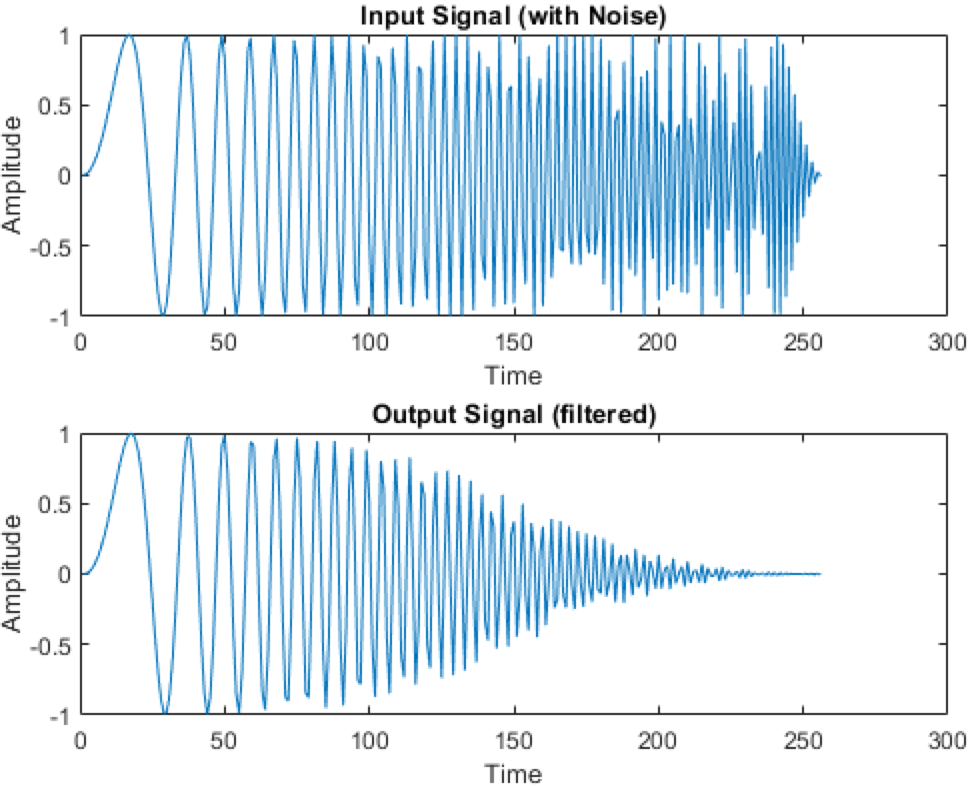
```
type(testbench_name);
```

```
%

%   Copyright 2011-2015 The MathWorks, Inc.


Fs = 256;              % Sampling frequency
Ts = 1/Fs;             % Sample time
t = 0:Ts:1-Ts;         % Time vector from 0 to 1 second
f1 = Fs/2;             % Target frequency of chirp set to Nyquist
in = sin(pi*f1*t.^2);  % Linear chirp from 0 to Fs/2 Hz in 1 second
out = zeros(size(in)); % Output the same size as the input

for ii=1:length(in)
    out(ii) = mlhdlc_df2t_filter(in(ii));
end

% Plot
figure('Name', [mfilename, '_plot']);
subplot(2,1,1);
plot(in);
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (with Noise)')

subplot(2,1,2);
plot(out);
xlabel('Time')
ylabel('Amplitude')
title('Output Signal (filtered)')
```

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_df2t_filter_tb
```



Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project
```

Next, add the file 'mlhdlc_filter.m' to the project as the MATLAB Function and 'mlhdlc_filter_tb.m' as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Fixed-Point Code Generation Workflow

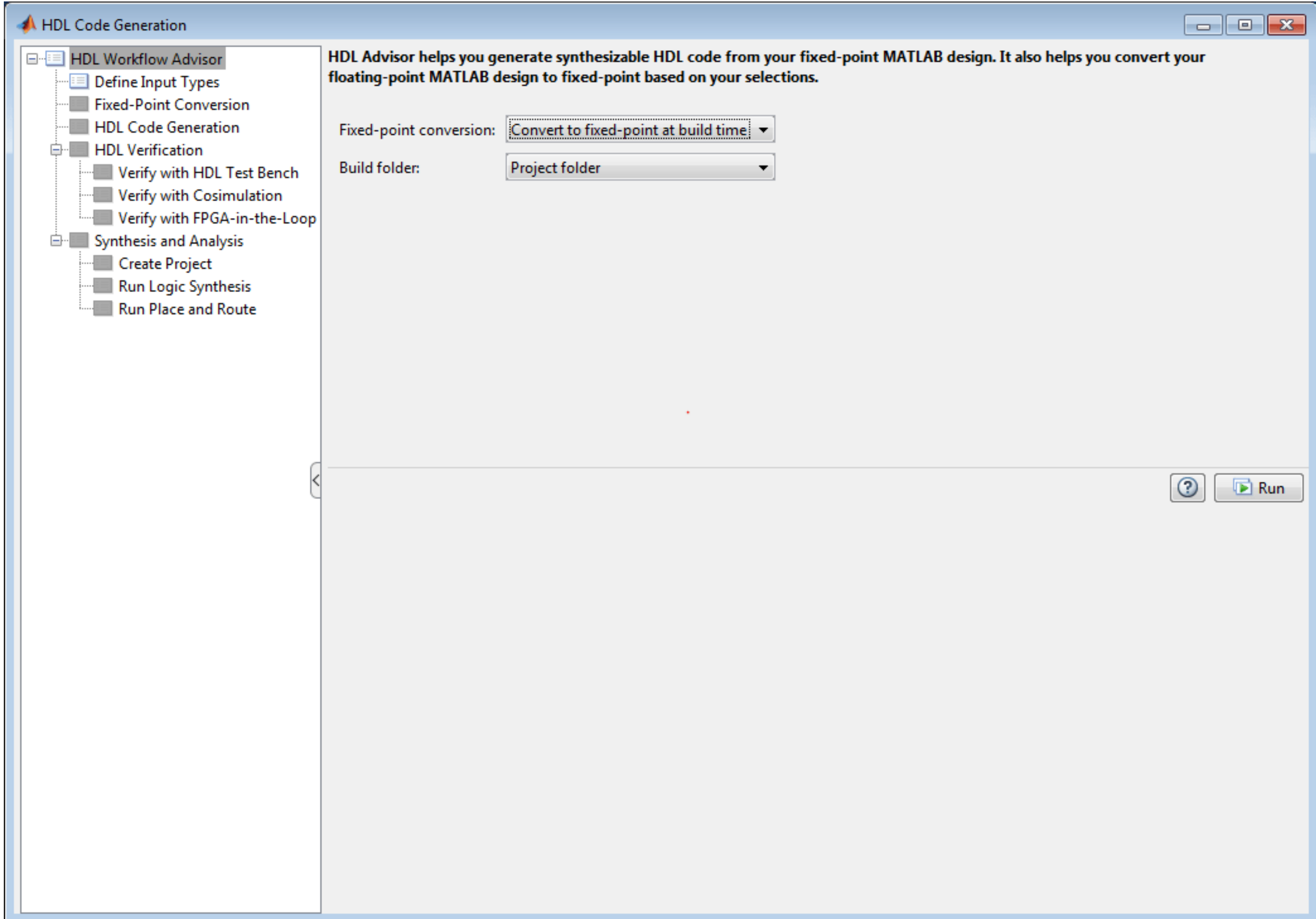
The floating-point to fixed-point conversion workflow allows you to:

- Verify that the floating-point design is code generation compliant
- Propose fixed-point types based on simulation data and word length settings
- Allow the user to manually adjust the proposed fixed-point types

- Validate the proposed fixed-point types
- Verify that the generated fixed-point MATLAB code has the desired numeric accuracy

Step 1: Launch Workflow Advisor

1. Click on the Workflow Advisor button to launch the HDL Workflow Advisor.
2. Choose **Convert to fixed-point at build time** for the option **Fixed-point conversion**.



Step 2: Define Input Types

In this step you can define input types manually or by specifying and running the testbench.

1. Click **Run** to execute this step.

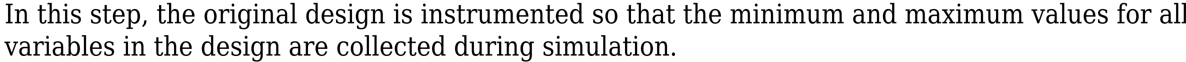
After simulation notice that the input variable `x` is defined as scalar double, `double(1x1)`.

Step 3: Run Simulation

1. Click on the **Fixed-Point Conversion** step.

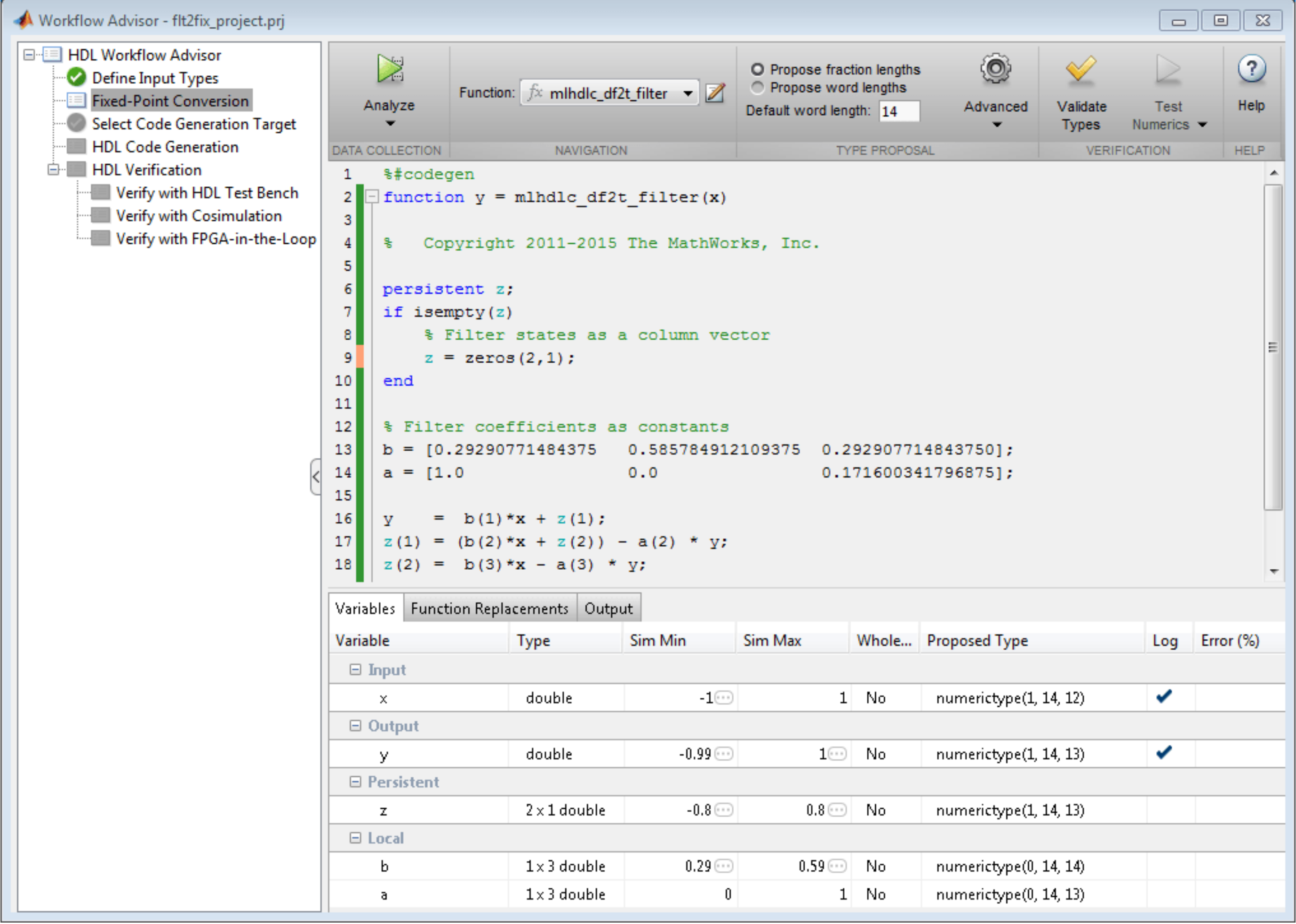
The design is compiled with the input types defined in the previous step and after the compilation is successful the variable table shows inferred types for all the functions in the design.

In this step, the original design is instrumented so that the minimum and maximum values for all variables in the design are collected during simulation.



1 Click on the 'Analyze' button.

Notice that the 'Sim Min' and 'Sim Max' table is now populated with simulation ranges. Fixed-point types are proposed based on the default word length settings.



At this stage, based on computed simulation ranges for all variables, you can compute:

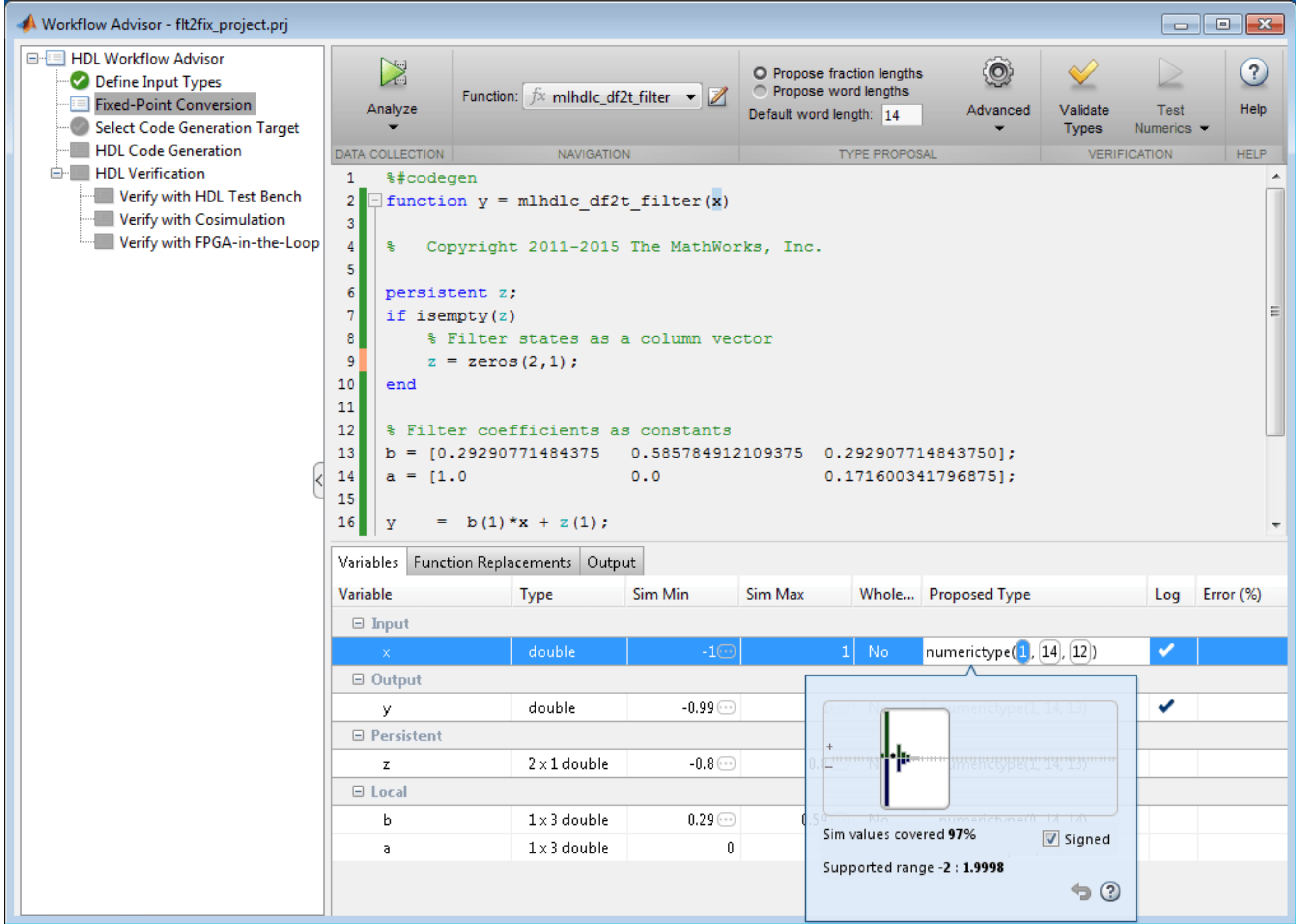
- Fraction lengths for a given fixed word length setting, or
- Word lengths for a given fixed fraction length setting.

The type table contains the following information for each variable existing in the floating-point MATLAB design, organized by function:

- Sim Min: The minimum value assigned to the variable during simulation.
- Sim Max: The maximum value assigned to the variable during simulation.
- Whole Number: Whether all values assigned during simulation are integers.

The type proposal step uses the above information and combines it with the user-specified word length settings to propose a fixed-point type for each variable.

You can also enable the **Log histogram data** option in the **Analyze** button drop-down menu to enable logging of histogram data.

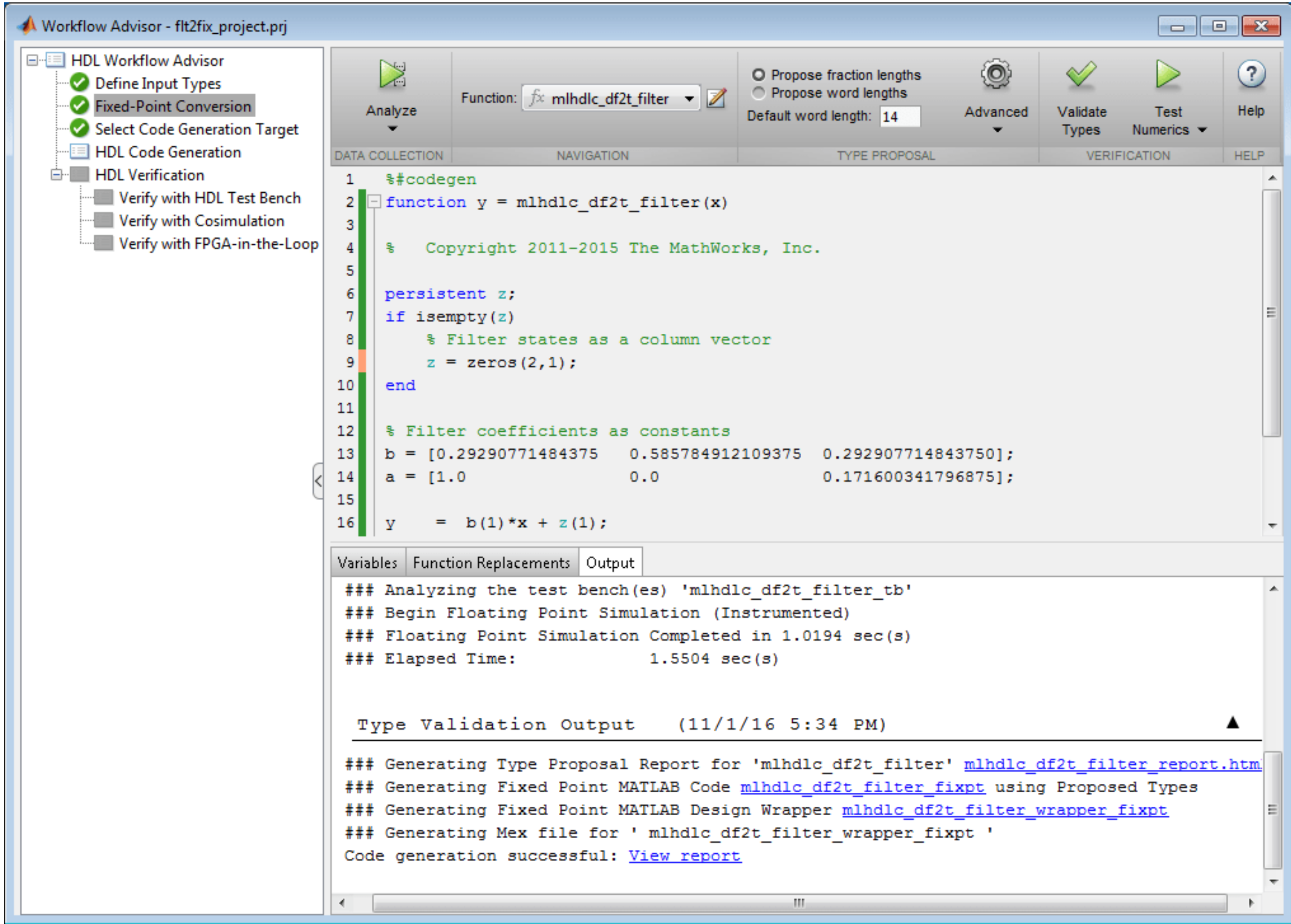


The histogram view concisely gives information about dynamic range of the simulation data for a variable. The x-axis correspond to bit weights and y-axis represents number of occurrences. The proposed numeric type information is overlaid on top of this graph and is editable. Moving the bounding white box left or right changes the position of binary point. Moving the right or left edges correspondingly change fraction length or wordlength. All the changes made to the proposed type are saved in the project.

Step 4: Validate types

In this step, the fixed-point types from the previous step are used to generate a fixed-point MATLAB design from the original floating-point implementation.

1 Click on the **Validate Types** button.



The generated code and other conversion artifacts are available via hyperlinks in the output window. The fixed-point types are explicitly shown in the generated MATLAB code.

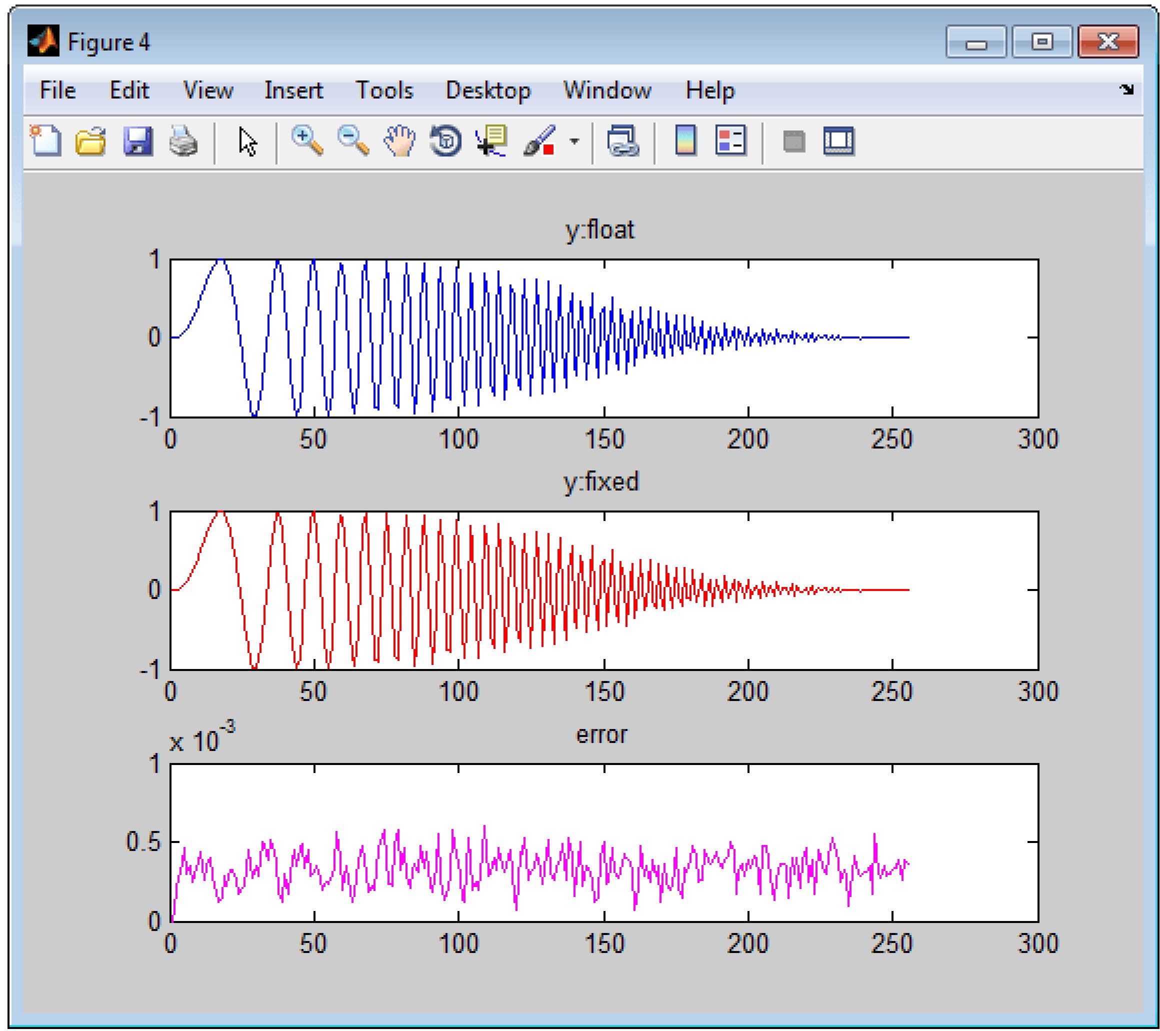
```
Editor - C:\Users\jilee\AppData\Local\Temp\mlhdlc_flt2fix_prj\codegen\mlhdlc_df2t_filter\fixpt\mlhdlc_df2t_filter_fixpt.m [Read Only]
mlhdlc_df2t_filter_fixpt.m

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                           %
%           Generated by MATLAB 9.1 and Fixed-Point Designer 5.3            %
%                                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen
function y = mlhdlc_df2t_filter_fixpt(x)

%   Copyright 2011-2015 The MathWorks, Inc.

fm = get_fimath();

persistent z;
if isempty(z)
    % Filter states as a column vector
    z = fi(zeros(2,1), 1, 14, 13, fm);
end

% Filter coefficients as constants
b = fi([0.29290771484375   0.585784912109375  0.292907714843750], 0, 14, 14, fm);
a = fi([1.0                0.0                0.171600341796875], 0, 14, 13, fm);

y    =  fi(b(1)*x + z(1), 1, 14, 13, fm);
z(1) = fi_signed((b(2)*x + z(2))) - a(2) * y;
z(2) =  fi_signed(b(3)*x) - a(3) * y;
```

Step 5: Test Numerics

1 Click on the **Test Numerics** button.

In this step, the generated fixed-point code is executed using MATLAB Coder.

If you enable the **Log inputs and outputs for comparison plots** option on the **Test Numerics** pane, an additional plot is generated for each scalar output that shows the floating point and fixed point results, as well as the difference between the two. For non-scalar outputs, only the error information is shown.



Step 6: Iterate on the Results

If the numerical results do not meet your desired accuracy after fixed-point simulation, you can return to the **Propose Fixed-Point Types** step in the Workflow Advisor. Adjust the word length settings or individually modify types as desired, and repeat the rest of the steps in the workflow until you achieve your desired results.

Refer to "Fixed-Point Type Conversion and Refinement" on page 4-61 for more details on how to iterate and refine the numerics of the algorithm in the generated fixed-point code.

Fixed-Point Type Conversion and Refinement

This example shows how to achieve your desired numerical accuracy when converting fixed-point MATLAB® code to floating-point code using the HDL Workflow Advisor.

Introduction

The floating-point to fixed-point conversion workflow in HDL Coder™ includes the following steps:

1. Verify the floating-point design is compatible for code generation.
2. Compute fixed-point types based on the simulation of the testbench.
3. Generate readable and traceable fixed-point MATLAB® code.
4. Verify the generated fixed-point design.

This tutorial uses Kalman filter suitable for HDL code generation to illustrate some key aspects of fixed-point conversion workflow, specifically steps 2 and 3 in the above list.

MATLAB Design

The MATLAB code used in this example implements a simple Kalman filter. This example also contains a MATLAB testbench that exercises the filter.

Kalman filter implementation suitable for HDL code generation

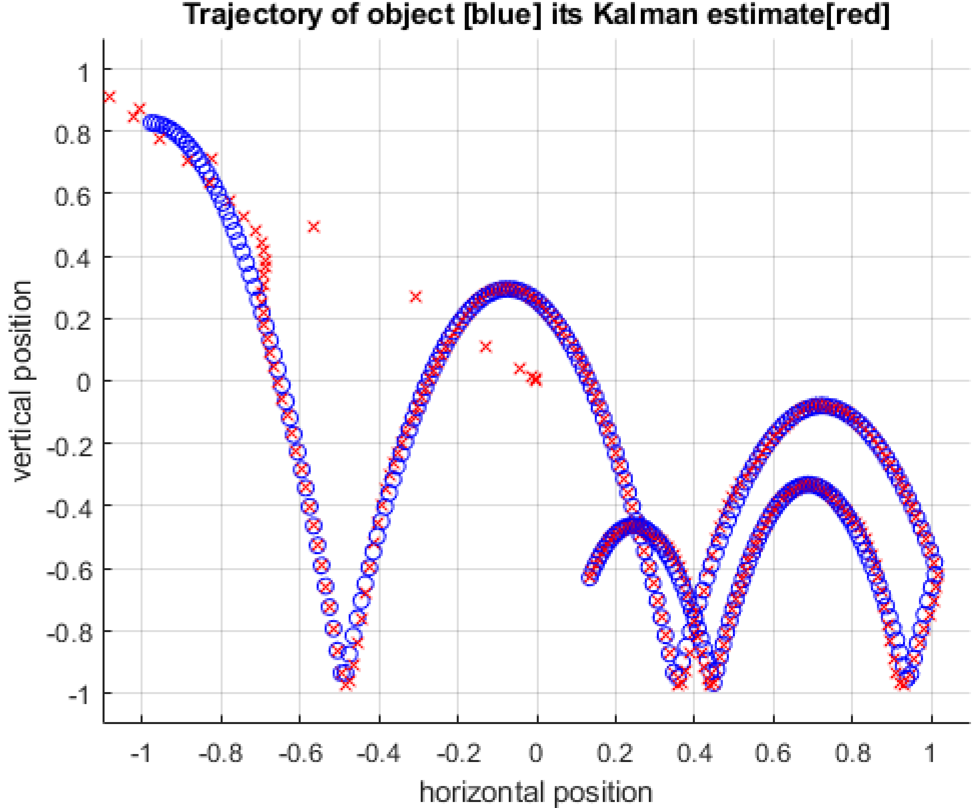
```
design_name = 'mlhdlc_kalman_hdl';
testbench_name = 'mlhdlc_kalman_hdl_tb';

edit('mlhdlc_kalman_hdl')
edit('mlhdlc_kalman_hdl_tb')
```

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_kalman_hdl_tb
```

```
Running --------> mlhdlc_kalman_hdl_tb

Current plot held
Current plot released
```



Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project
```

Next, add the file `mlhdlc_kalman_hdl.m` to the project as the MATLAB Function and `mlhdlc_kalman_hdl_tb.m` as the MATLAB Test Bench.

Refer to "Get Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Fixed-Point Code Generation Workflow

Perform the following tasks before moving on to the fixed-point type proposal step:

1. Click the **Workflow Advisor** button to launch the HDL Workflow Advisor.
2. Choose **Convert to fixed-point at build time** for the **Fixed-point conversion** option.
3. Click **Run** button to define input types for the design from the testbench.
4. Select the **Fixed-Point Conversion** workflow step.
5. Click **Analyze** to execute the instrumented floating-point simulation.

Refer to "Floating-Point to Fixed-Point Conversion" on page 4-51 for a more complete tutorial on these steps.

Determine the Initial Fixed Point Types

After instrumented floating-point simulation completes, Fixed-Point Types are proposed based on the simulation results.

At this stage of the conversion proposes fixed-point types for each variable in the design based on the recorded min/max values of the floating point variables and user input.

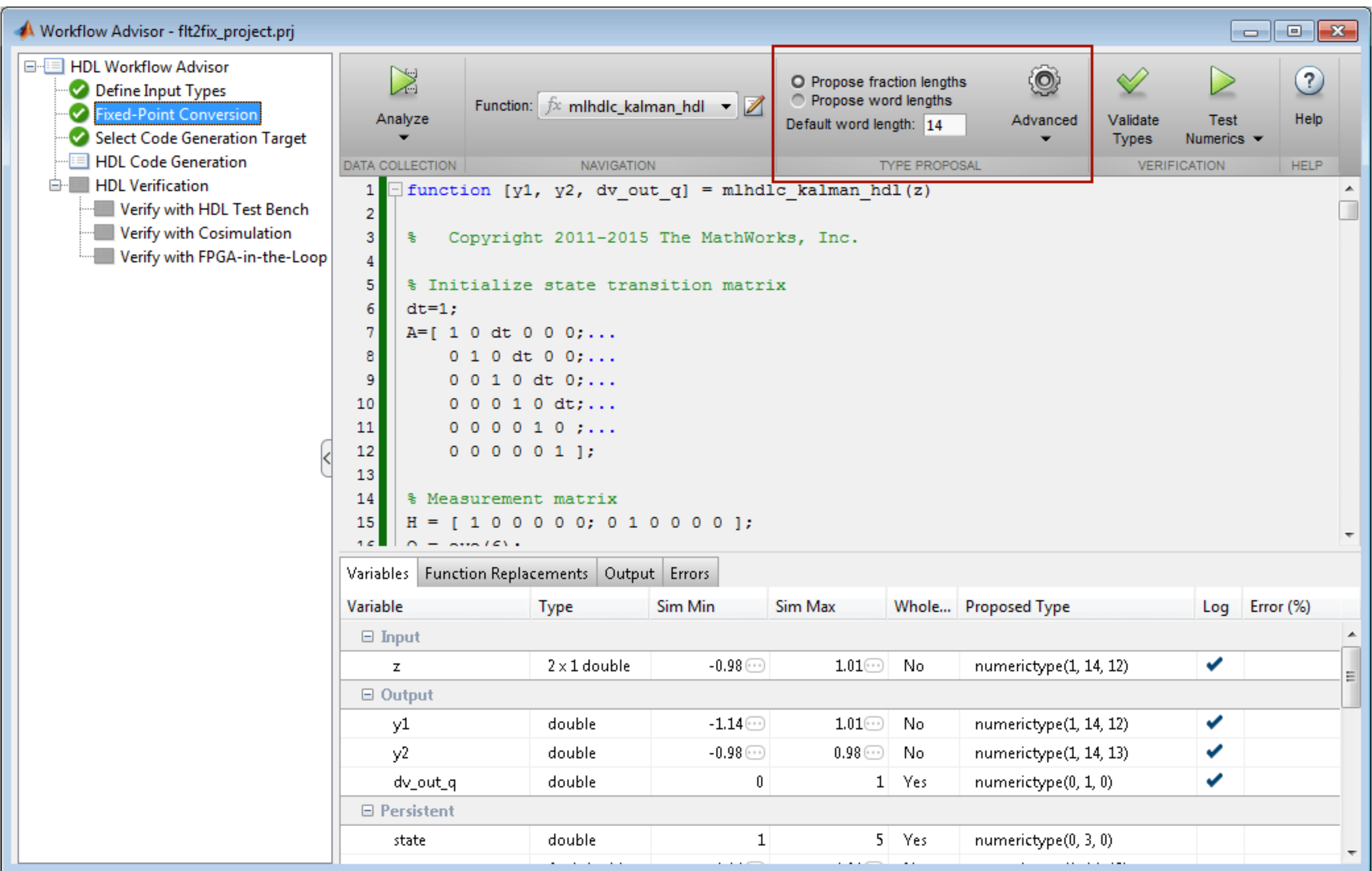
At this point, for all variables, you can (re)compute and propose:

- Fraction lengths for a given fixed word length setting, or
- Word lengths for a given fixed fraction length setting.

Choose the Word Length Setting

When you are starting with a floating-point design and going through the floating-point to fixed-point conversion for the first time, it is a good practice to start by specifying a 'Default Word Length' setting based on the largest dynamic range of all the variables in the design.

In this example, we start with a default word length of 22 and run the 'Propose Fixed-Point Types' step.



Explore the Proposed Fixed-Point Type Table

The type table contains the following information for each variable, organized by function, existing in the floating-point MATLAB design:

- Sim Min: The minimum value assigned to the variable during simulation.
- Sim Max: The maximum value assigned to the variable during simulation.
- Whole Number: Whether all values assigned during simulation are integer.

The type proposal step uses the above information and combines it with the user-specified word length settings to propose a fixed-point type for each variable.

You can also use **Compute Derived Range Analysis** to compute derived ranges and that is covered in detail in this tutorial “Fixed-Point Type Conversion and Derived Ranges” on page 4-73.

Interpret the Proposed Numeric Types for Variables

Based on the simulation range (min & max) values and the default word length setting, a numeric type is proposed for each variable.

The following table shows numeric type proposals for a **Default word length** of 22 bits.

Variables | Function Replacements | Output | Errors

Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
⊟ Input							
z	2 x 1 double	-0.98	1.01	No	numerictype(1, 14, 12)	✔	
⊟ Output							
y1	double	-1.14	1.01	No	numerictype(1, 14, 12)	✔	
y2	double	-0.98	0.98	No	numerictype(1, 14, 13)	✔	
dv_out_q	double	0	1	Yes	numerictype(0, 1, 0)	✔	
⊟ Persistent							
state	double	1	5	Yes	numerictype(0, 3, 0)		
x_est	6 x 1 double	-1.14	1.01	No	numerictype(1, 14, 12)		
p_est	6 x 6 double	0	472.78	No	numerictype(0, 14, 5)		
y	2 x 1 double	-1.14	1.01	No	numerictype(1, 14, 12)		
x_prd	6 x 1 double	-1.35	1.17	No	numerictype(1, 14, 12)		
p_prd	6 x 6 double	0	896.74	No	numerictype(0, 14, 4)		
z_prd	2 x 1 double	-1.35	1.17	No	numerictype(1, 14, 12)		
S	2 x 2 double	0	1896.74	No	numerictype(0, 14, 3)		
B	2 x 6 double	0	896.74	No	numerictype(0, 14, 4)		
klm_gain	6 x 2 double	0	0.47	No	numerictype(0, 14, 15)		
dv_out	double	0	1	Yes	numerictype(0, 1, 0)		
backslash_dv_out	double	0	1	Yes	numerictype(0, 1, 0)		
⊟ Local							
dt	double	1	1	Yes	numerictype(0, 1, 0)		
A	6 x 6 double	0	1	Yes	numerictype(0, 1, 0)		
H	2 x 6 double	0	1	Yes	numerictype(0, 1, 0)		
Q	6 x 6 double	0	1	Yes	numerictype(0, 1, 0)		
R	2 x 2 double	0	1000	Yes	numerictype(0, 10, 0)		

Examine the types proposed in the above table for variables instrumented in the top-level design.

Floating-Point Range for variable `B`:

- Simulation Info: SimMin: 0, SimMax: 896.74.., Whole Number: No
- Type Proposed: numerictype(0,22,12) (Signedness: Unsigned, WordLength: 22, FractionLength: 12)

The floating-point range:

- Has the same number of bits as the **Default word length**.
- Uses the minimum number of bits to completely represent the range.
- Uses the rest of the bits to represent the precision.

Integer Range for variable `A`:

- Simulation Info: SimMin: 0, SimMax: 1, Whole Number: Yes
- Type Proposed: numerictype(0,1,0) (Signedness: Unsigned, WordLength: 1, FractionLength: 0)

The integer range:

- Has the minimum number of bits to represent the whole integer range.
- Has no fractional bits.

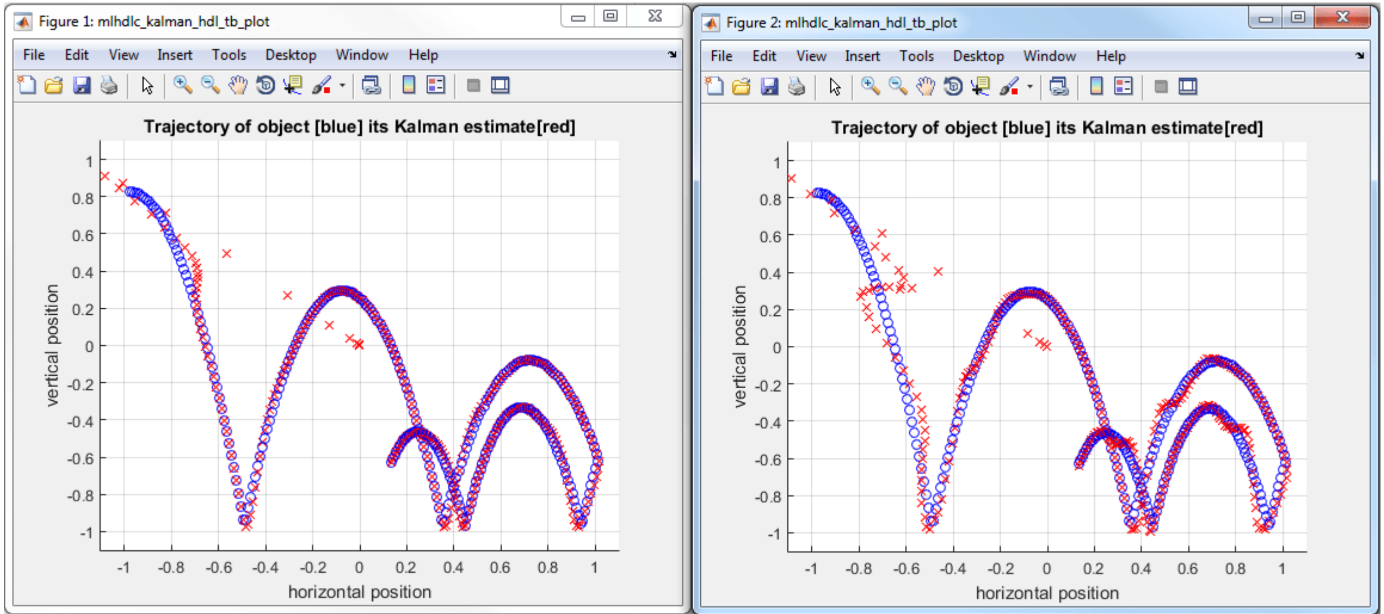
All the information in the table is editable, persists across iterations, and is saved with your code generation project.

Generate Fixed-Point Code and Verify the Generated Code

Based on the numeric types proposed for a default word length of 22, continue with fixed-point code generation and verification steps and observe the plots.

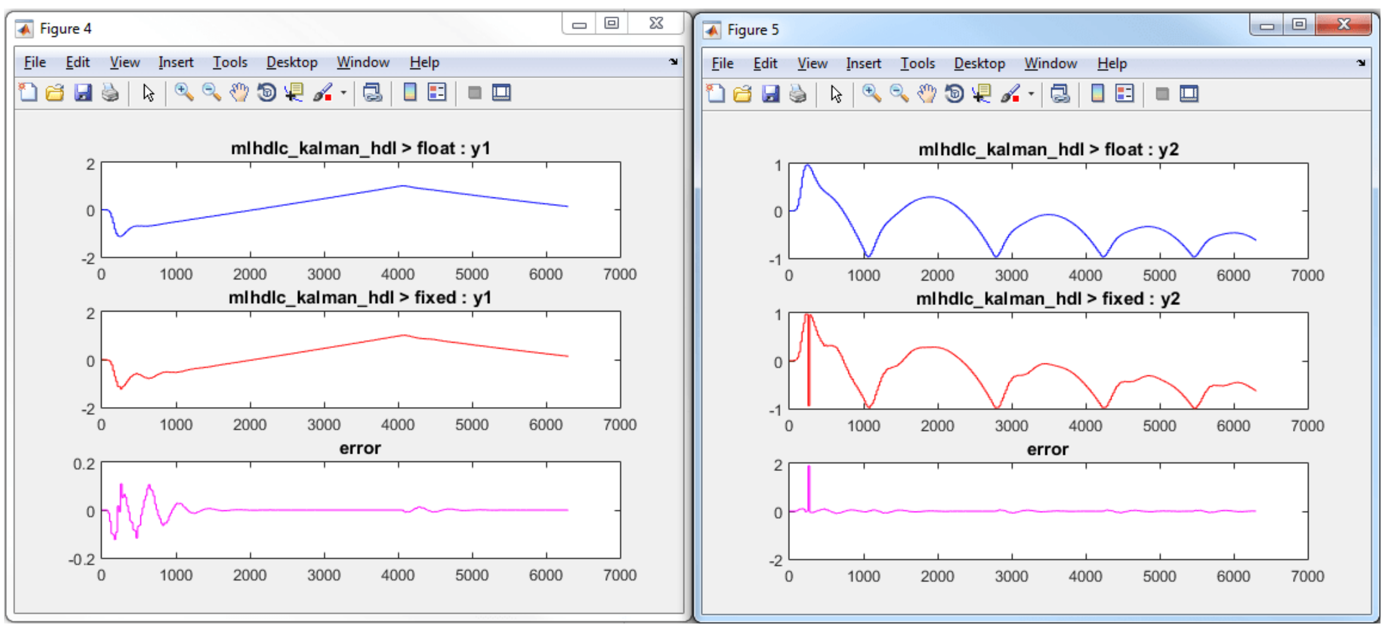
1. Click on **Validate Types** to apply computed fixed-point types.
2. Next choose the option **Log inputs and outputs for comparison plots** and then click on the **Test Numerics** to rerun the testbench on the fixed-point code.

The plot on the left is generated from testbench during the simulation of floating-point code, the one on the right is generated from the simulation of the generated fixed-point code. Notice, the plots do not match.



Having chosen comparison plots option you will see additional plots that compare the floating and fixed point simulation results for each output variable.

Examine the error graph for each output variable. It is very high for this particular design.



Iterate on the Results

One way to reduce the error is to increase **Default word length** and repeat the fixed-point conversion.

In this example design, when a word length of 22 bits is chosen there is a lot of truncation error when representing the precision. More bits are required to the right of the binary point to reduce the truncation errors.

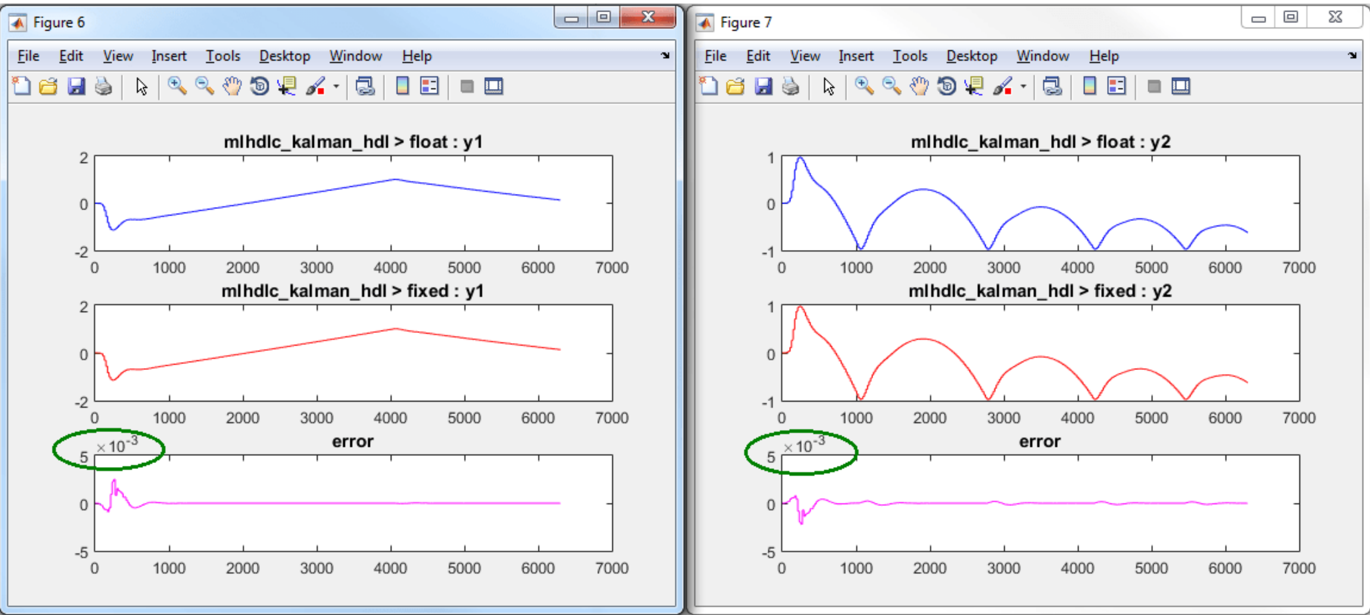
Let us now increase the default word length to 28 bits and repeat the type proposal and validation steps.

1 Select a **Default word length** of 28.

Changing default word length automatically triggers the type proposal step and new fixed-point types are proposed based on the new word length setting. Also notice that type validation needs to be rerun and numerics need to be verified again.

1 Click on **Validate Types**.
2 Click on **Test Numerics** to rerun the testbench on the fixed-point code.

Once these steps are complete, re-examine the comparison plots and notice that the error is now roughly three orders of magnitude smaller.



Working with Generated Fixed-Point Files

This example shows how to work with the files generated during floating-point to fixed-point conversion.

Introduction

This tutorial uses a simple filter implemented in floating-point and an associated testbench to illustrate the file structure of the generated fixed-point code.

```
design_name = 'mlhdlc_filter';
testbench_name = 'mlhdlc_filter_tb';
```

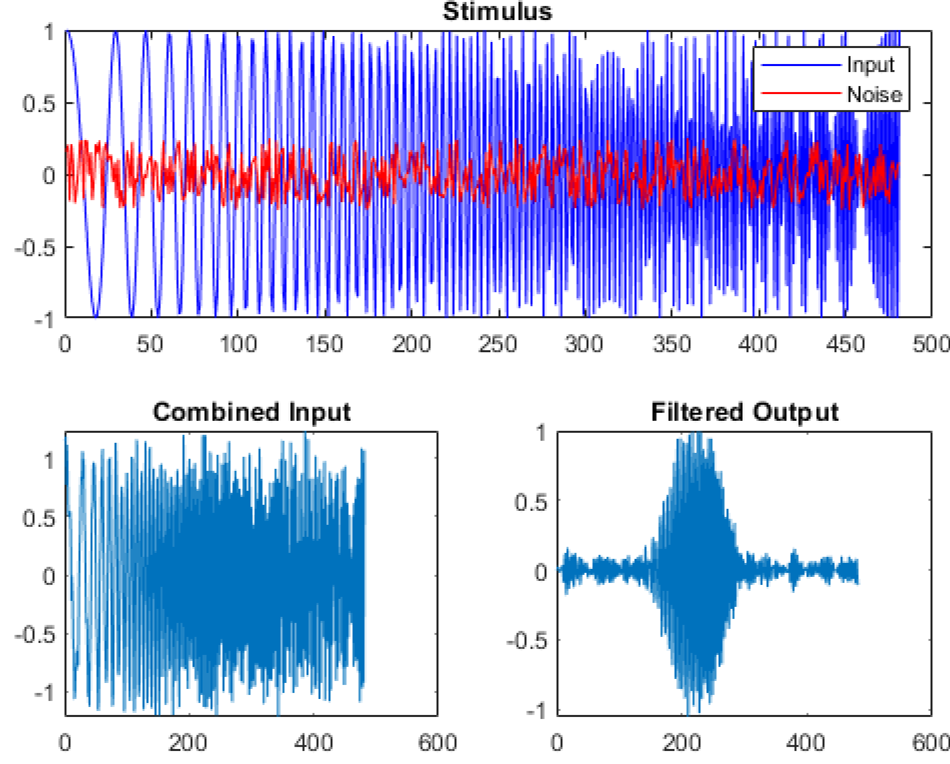
MATLAB® Code

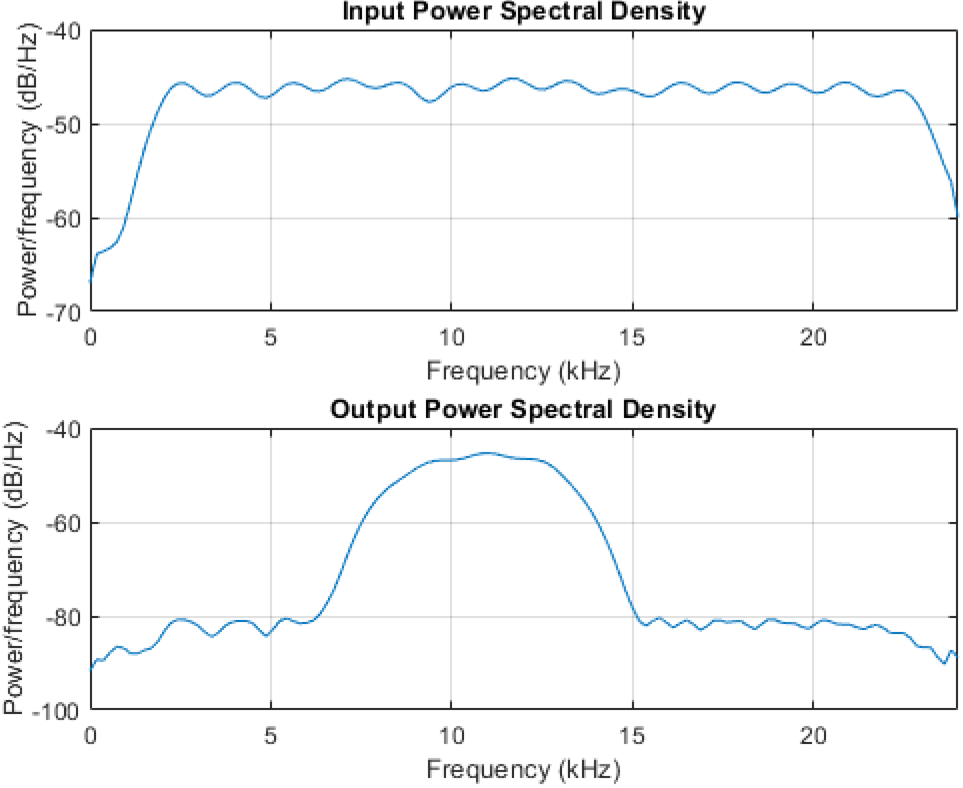
1. MATLAB Design: mlhdlc_filter
2. MATLAB testbench: mlhdlc_filter_tb

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_filter_tb
```





Create a New HDL Coder™ Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project
```

Next, add the file 'mlhdlc_filter' to the project as the MATLAB Function and 'mlhdlc_filter_tb' as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Fixed-Point Code Generation Workflow

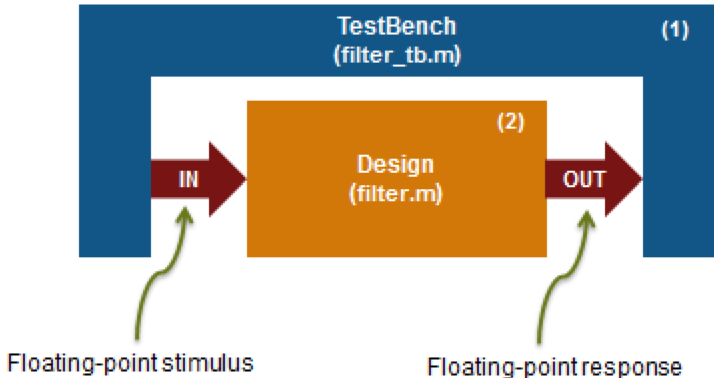
Perform the following tasks in preparation for the fixed-point code generation step:

1. Click the **Workflow Advisor** button to launch the Workflow Advisor.
2. Choose `Convert to fixed-point at build time` for the option **Fixed-point conversion**.
3. Right-click the **Fixed-Point Conversion** step and select **Run to Selected Task** to execute the instrumented floating-point simulation.

Refer to “Floating-Point to Fixed-Point Conversion” on page 4-51 for a more complete tutorial on these steps.

Floating-Point Design Structure

The original floating-point design and testbench have the following relationship.



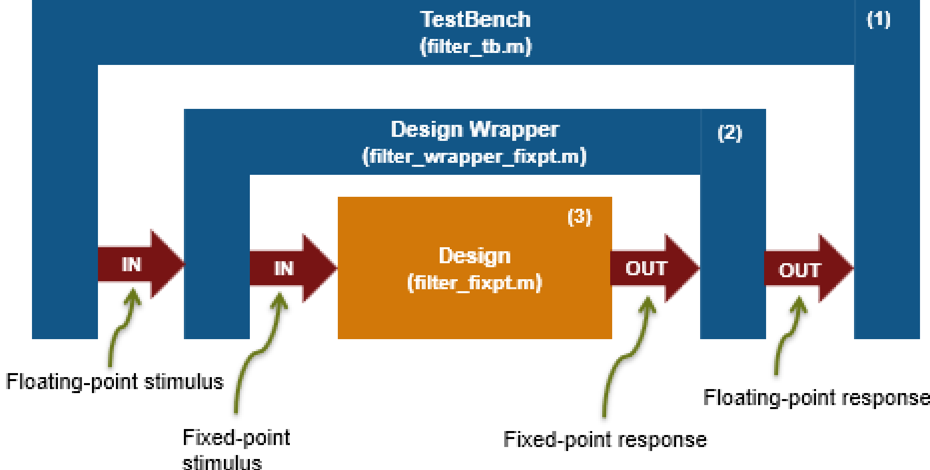
For floating-point to fixed-point conversion, the following requirements apply to the original design and the testbench:

- The testbench 'mlhdlc_filter_tb.m' (1) must be a script or a function with no inputs. The test bench can have local helper functions that take inputs.
- The design 'mlhdlc_filter.m' (2) must be a function.
- There must be at least one call to the design from the testbench. All call sites contribute when determining the proposed fixed-point types.
- Both the design and testbench can call other sub-functions within the file or other functions on the MATLAB path. Functions that exist within matlab/toolbox are not converted to fixed-point.

In the current example, the MATLAB testbench 'mlhdlc_filter_tb' has a single call to the design function 'mlhdlc_filter'. The testbench calls the design with floating-point inputs and accumulates the floating-point results for plotting.

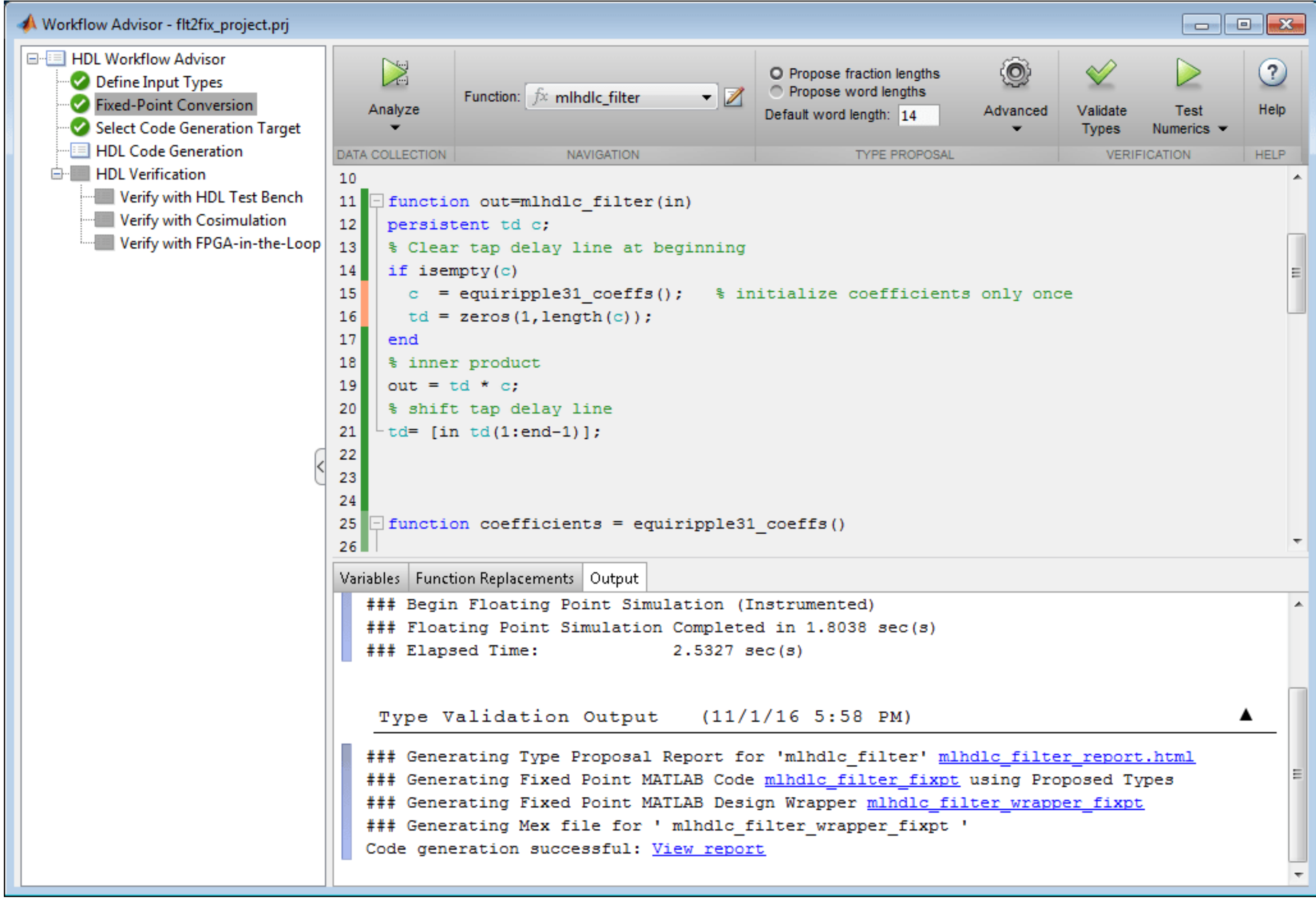
Validate Types

During the type validation step, fixed-point code is generated for this design and compiled to verify that there are no errors when applying the types. The output files will have the following structure.



The following steps are performed during fixed-point type validation process:

1. The design file 'mlhdlc_filter.m' is converted to fixed-point to generate fixed-point MATLAB code, 'mlhdlc_filter_fixpt.m' (3).
2. All user-written functions called in the floating-point design are converted to fixed point and included in the generated design file.
3. A new design wrapper file is created, called 'mlhdlc_filter_wrapper_fixpt.m' (2). This file converts the floating-point data values supplied by the testbench to the fixed-point types determined for the design inputs during the conversion step. These fixed point values are fed into the converted fixed-point design, 'mlhdlc_filter_fixpt.m'.
4. 'mlhdlc_filter_fixpt.m' will be used for HDL code generation.
5. All the generated fixed-point files are stored in the output directory 'codegen/mlhdlc_filter/fixpt'.



Click the links to the generated code in the Workflow Advisor log Window to examine the generated fixed-point design and wrapper.

Fixed-Point Type Conversion and Derived Ranges

This example shows how to achieve your desired numerical accuracy when converting fixed-point MATLAB® code to floating-point code using static range analysis which helps to compute derived ranges of the variables from design ranges.

Introduction

The floating-point to fixed-point conversion workflow in HDL Coder™ includes the following steps:

1. Verify the floating-point design is compatible for code generation.
2. Compute fixed-point types based on the simulation of the testbench.
3. Generate readable and traceable fixed-point MATLAB® code.
4. Verify the generated fixed-point design.

However, the fixed-point types proposed from the simulation depends on the quality of the testbench. Sometimes it is hard to write testbenches which completely cover paths of the design representing full design ranges of all the variables. Static analysis based workflow can be used in such cases to compute derived ranges from design ranges.

This tutorial uses a symmetric FIR filter whose output signal is integrated over time.

MATLAB Design

The MATLAB code used in this example implements a simple Kalman filter. This example also contains a MATLAB testbench that exercises the filter.

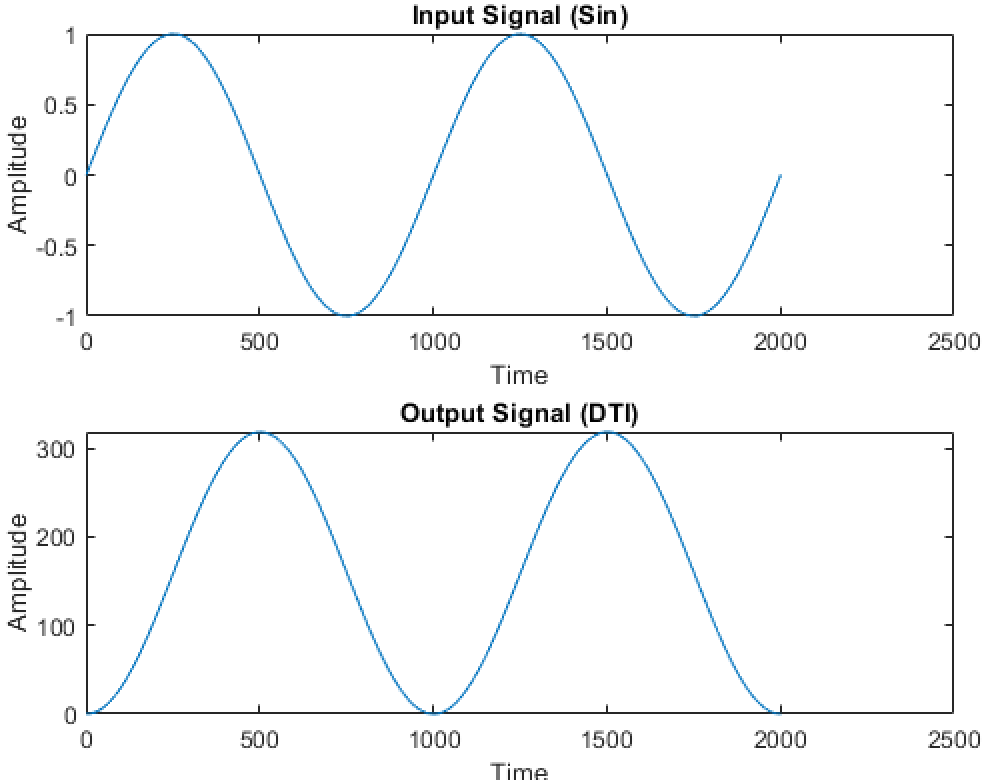
```
design_name = 'mlhdlc_dti';
testbench_name = 'mlhdlc_dti_tb';
```

1. MATLAB Design: mlhdlc_dti
2. MATLAB testbench: mlhdlc_dti_tb

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_dti_tb
```



Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project_dmm
```

Next, add the file 'mlhdlc_dti.m' to the project as the MATLAB Function and 'mlhdlc_dti_tb.m' as the MATLAB Test Bench.

Refer to "Get Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Fixed-Point Code Generation Workflow

Perform the following tasks before moving on to the fixed-point type proposal step:

1. Click the 'Workflow Advisor' button to launch the HDL Workflow Advisor.
2. Choose 'Convert to fixed-point at build time' for the 'Fixed-point conversion' option.
3. Click 'Run' button to define input types for the design from the testbench.
4. Select the 'Fixed-Point Conversion' workflow step.
5. Click 'Analyze' to execute the instrumented floating-point simulation.

Refer to "Floating-Point to Fixed-Point Conversion" on page 4-51 for a more complete tutorial on these steps.

Determine the Initial Fixed Point Types

After instrumented floating-point simulation completes, you will see 'Fixed-Point Types are proposed' based on the simulation results.

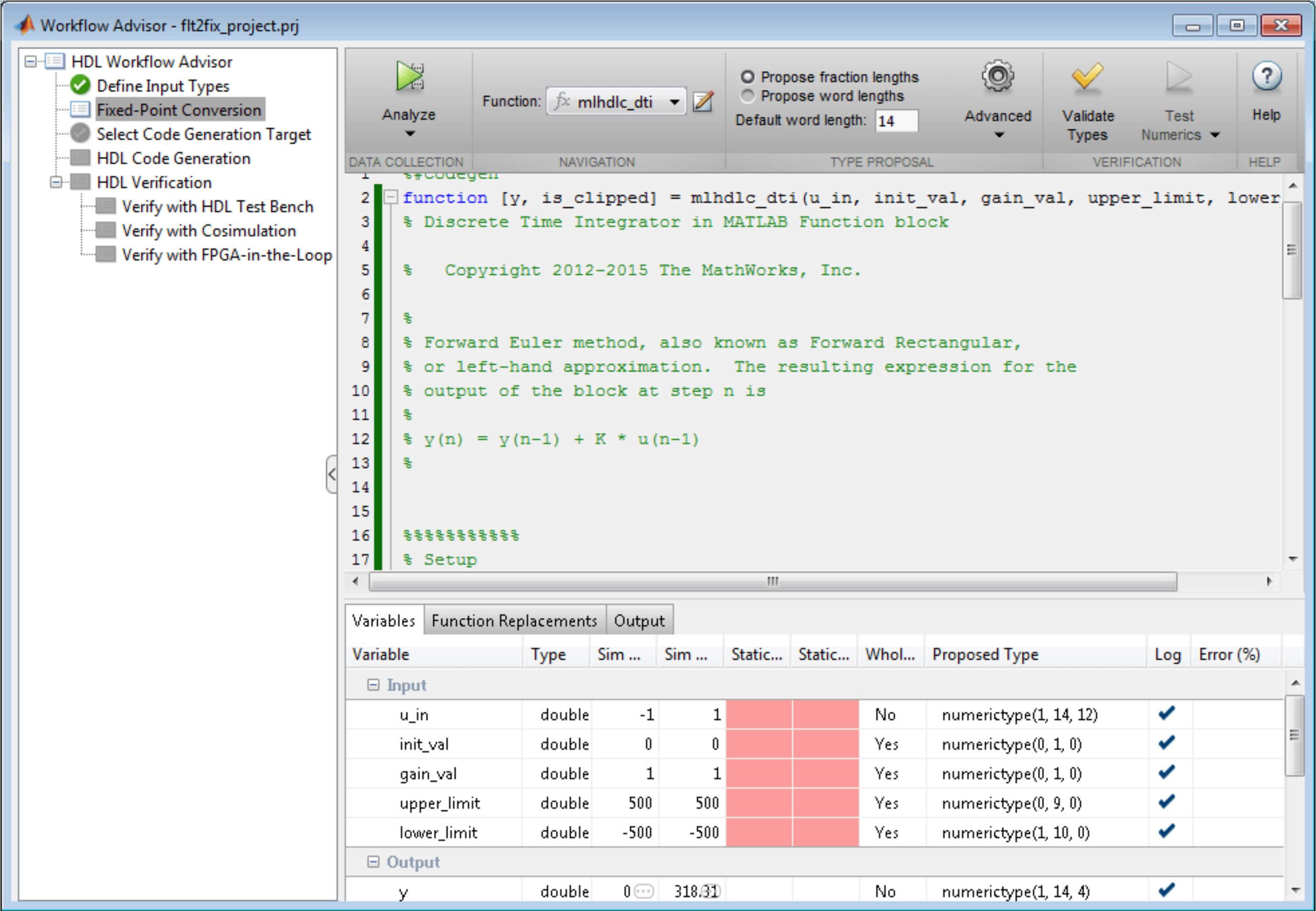
At this step fixed-point types for each variable in the design based on the recorded min/max values of the floating point variables and user input.

Observe the simulation range of the variable 'is_clipped' in the function 'mlhdlc_dti'. You will notice that the simulation range of this variable is a constant value 0. However, if you can observe the code to see that the variable can take values from -1 to -1.

The ranges for the variable can be fixed by updating the testbench. However, it may be desirable to compute program ranges through static analysis.

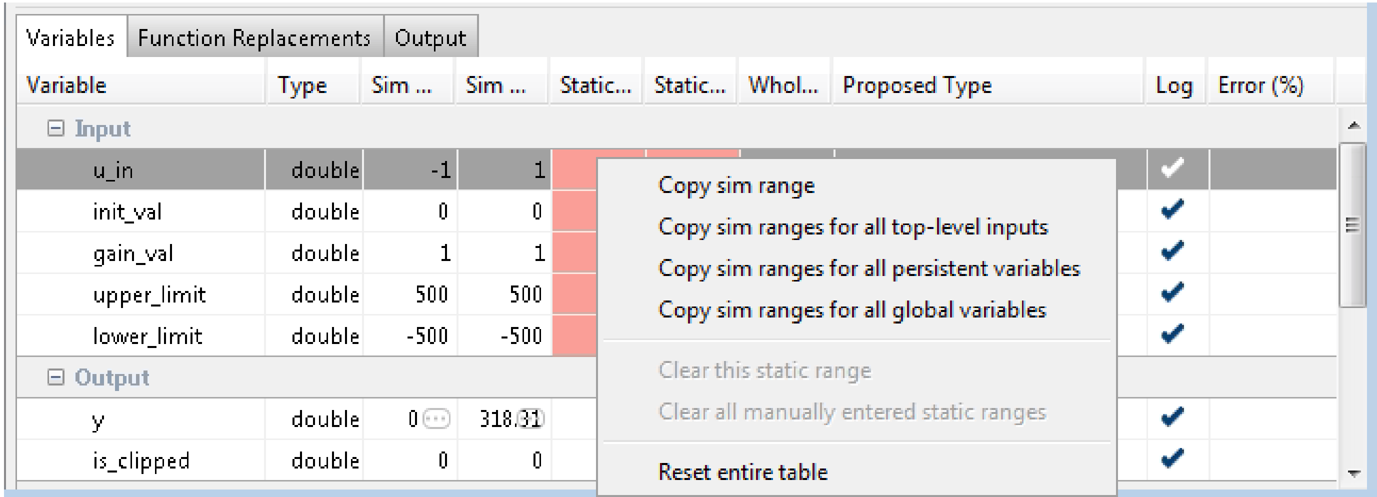
Entering Design Ranges and Computing Derived Ranges

In this step you can specify design ranges and compute derived ranges through static analysis. Enable derived range analysis by clicking the 'analyze ranges using derived range analysis' checkbox in the 'Analyze' button's menu. The tool will then prompt you to specify design ranges for the inputs variables in the Static Min and Static Max columns.

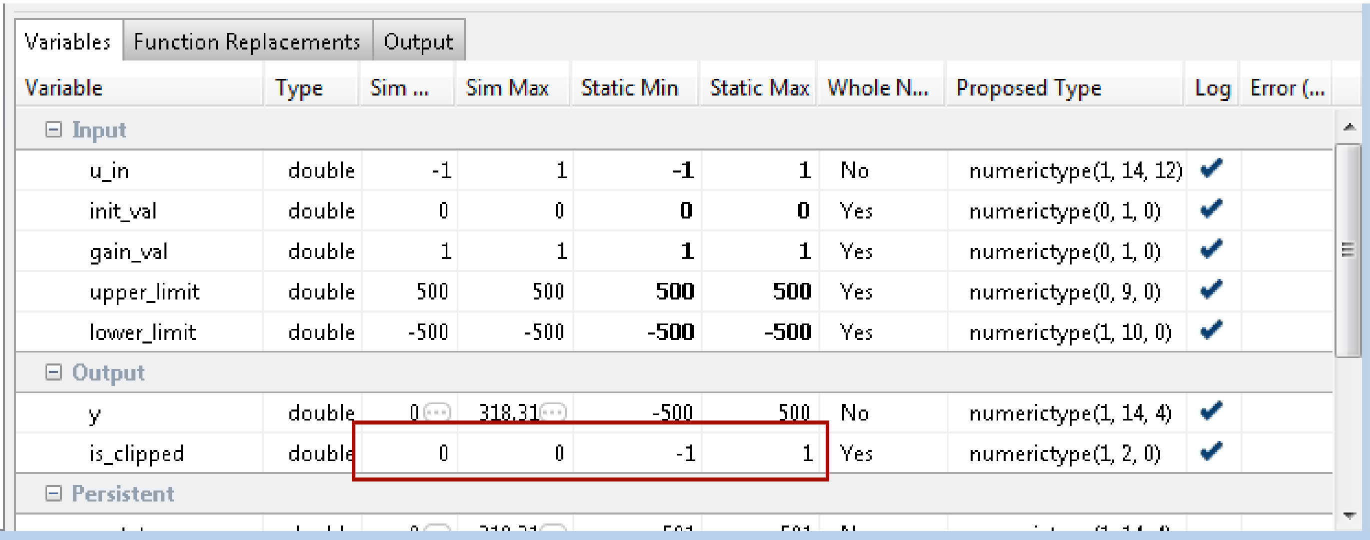


There are multiple ways you can enter design ranges.

1. You can manually edit the 'Static Min' and 'Static Max' entries in the table and specify design ranges.
2. You can copy the Sim Min and Sim Max for a variable via right-clicking on the table cell (or)
3. You can Lock or Specify the Output type to be used as the design range



Once all the necessary design ranges are specified you can click on the 'Analyze' button to use derived range analysis.



Notice that the derived range of the variable now includes values taken in all paths of the control flow.

Insufficient design ranges

Sometimes specifying ranges for input variables alone may not be sufficient for certain designs. For example in a MATLAB design implementing a counter using a persistent variable, the range of the variable depends on number of times the design is called. In such situations you will see computed

derived static ranges for the variable reported as -Inf or +Inf. When these imprecise ranges appear please consider specifying ranges for such persistent variables.

Generate HDL-Compatible Lookup Table Function Replacements Using coder.approximate

This example shows MATLAB® code generation from a floating-point MATLAB design that is not ready for code generation. Use `coder.approximate` function to generate a lookup table based MATLAB function. This newly generated function is ready for HDL code generation (not shown in this demo).

Introduction

The MATLAB code used in the example is sigmoid function, which is used for threshold detection and decision making problems. For example, neural networks use sigmoid functions with appropriate thresholds to train systems for learning patterns.

MATLAB Design

```
design_name = 'mlhdlc_approximate_sigmoid';
testbench_name = 'mlhdlc_approximate_sigmoid_tb';
```

This function call displays the file contents of `design_name`.

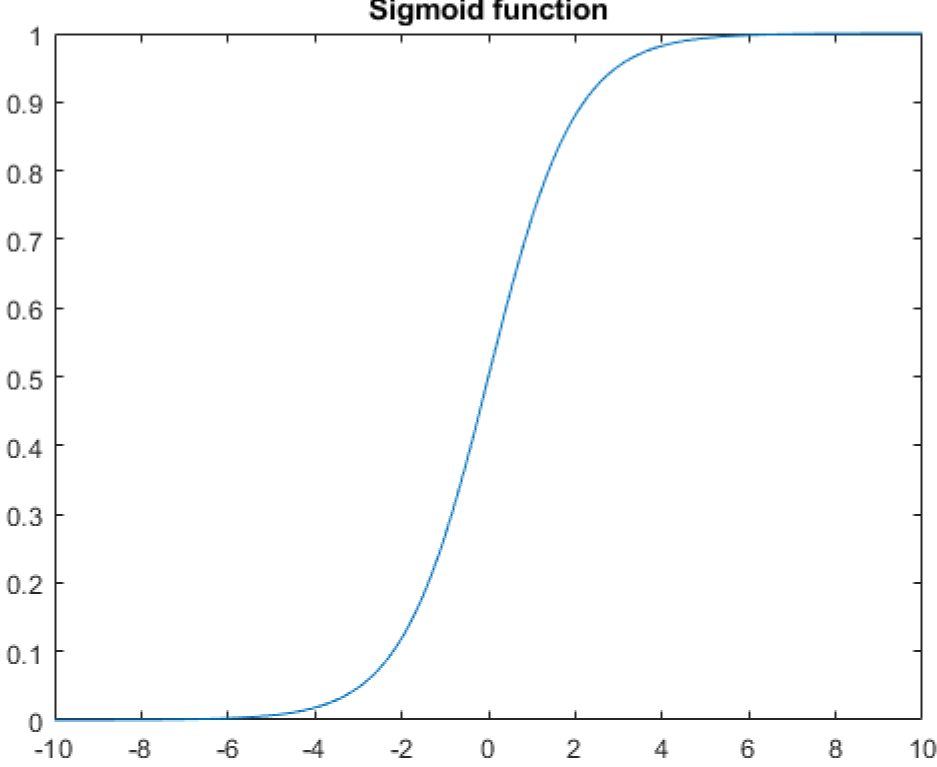
```
dbtype(design_name)
```

```
1     function y = mlhdlc_approximate_sigmoid( x )
2     %
3
4     %   Copyright 2014-2015 The MathWorks, Inc.
5
6         y = 1./(1+exp(-x));
7     end
```

Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_approximate_sigmoid_tb
```



1. MATLAB Design: mlhdlc_approximate_sigmoid
2. MATLAB testbench: mlhdlc_approximate_sigmoid_tb

Use `coder.approximate` to generate a lookup-table based replacement function for `mlhdlc_approximate_sigmoid`.

Generate fixed-point lookup-table replacements

```
repCfg = coder.approximation('Function','mlhdlc_approximate_sigmoid','CandidateFunction',@mlhdlc_
                             'NumberOfPoints',50,'InputRange',[-10,10],'FunctionNamePrefix','repsig_
coder.approximate(repCfg);
```

First the fixed-point conversion completes with appropriate function replacements, and following console message,

```
### Generating approximation for 'sigmoid' : repsiglookuptable.m
### Generating testbench for 'sigmoid' : repsiglookuptable_tb.m
### LookupTable replacement for function 'sigmoid' used 50 data points
```

This should generate the MATLAB files `repsig_lookuptable_tb`, and `repsig_lookuptable` containing the testbench and design respectively.

Test the replacement functions

To visually see the degree of match between lookup-table based replacement function and the original function use the testbench,

```
repsig_lookuptable_tb();
```

Code Generation

- “Create and Set Up Your Project” on page 5-2
- “Specify Properties of Entry-Point Function Inputs” on page 5-4
- “Code Generation Reports” on page 5-7
- “Generate Instantiable Code for Functions” on page 5-12
- “Edit Configuration Parameters for HDL Coder” on page 5-13
- “Integrate Custom HDL Code Into MATLAB Design” on page 5-15
- “Enable MATLAB Function Block Generation” on page 5-20
- “System Design with HDL Code Generation from MATLAB and Simulink” on page 5-21
- “Specify the Clock Enable Rate” on page 5-24
- “Specify Test Bench Clock Enable Toggle Rate” on page 5-26
- “Generate an HDL Coding Standard Report from MATLAB” on page 5-28
- “Generate an HDL Lint Tool Script” on page 5-31
- “Generate HDL Code from MATLAB Functions That Use Automated Lookup Table Generation” on page 5-33
- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-38
- “Minimize Clock Enables” on page 5-40

Create and Set Up Your Project

In this section...
“Create a New Project” on page 5-2
“Open an Existing Project” on page 5-3
“Add Files to the Project” on page 5-3

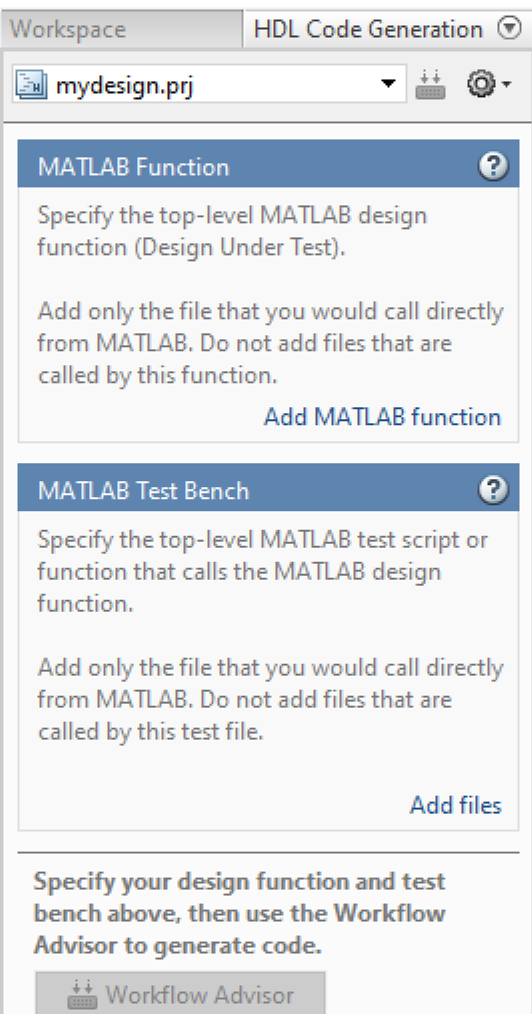
Create a New Project

1 At the MATLAB command line, enter:

```
hdlcoder
```

2 Enter a project name in the project dialog box and click **OK**.

HDL Coder creates the project in the local working folder, and, by default, opens the project in the right side of the MATLAB workspace.



Alternatively, you can create a new HDL Coder project from the apps gallery:

1. On the **Apps** tab, on the far right of the **Apps** section, click the arrow ▾.
2. Under **Code Generation**, click **HDL Coder**.
3. Enter a project name in the project dialog box and click **OK**.

Open an Existing Project

At the MATLAB command line, enter:

```
open project_name
```

where *`project_name`* specifies the full path to the project file.

Alternatively, navigate to the folder that contains your project and double-click the `.prj` file.

Add Files to the Project

Add the MATLAB Function (Design Under Test)

First, you must add the MATLAB file from which you want to generate code to the project. Add only the top-level function that you call from MATLAB (the Design Under Test). Do not add files that are called by this file. Do not add files that have spaces in their names. The path must not contain spaces, as spaces can lead to code generation failures in certain operating system configurations.

To add a file, do one of the following:

- In the project pane, under **MATLAB Function** , click the **Add MATLAB function** link and browse to the file.
- Drag a file from the current folder and drop it in the project pane under **MATLAB Function**.

If the functions that you added have inputs, and you do not specify a test bench, you must define these inputs. See “Specify Properties of Entry-Point Function Inputs” on page 5-4.

Add a MATLAB Test Bench

You must add a MATLAB test bench unless your design does not need fixed-point conversion and you do not want to generate an RTL test bench. If you do not add a test bench, you must define the inputs to your top-level MATLAB function. For more information, see “Specify Properties of Entry-Point Function Inputs” on page 5-4.

To add a test bench, do one of the following:

- In the project panel, under **MATLAB Test Bench**, click the **Add MATLAB test bench** link and browse to the file.
- Drag a file from the current folder and drop it in the project pane under **MATLAB Test Bench**.

Specify Properties of Entry-Point Function Inputs

In this section...

"When to Specify Input Properties" on page 5-4
"Why You Must Specify Input Properties" on page 5-4
"Methods for Defining Properties of Primary Inputs" on page 5-4
"Properties to Specify" on page 5-5
"Rules for Specifying Properties of Primary Inputs" on page 5-6

When to Specify Input Properties

If you supply a test bench for your MATLAB algorithm, you do not need to specify the primary function inputs manually. The HDL Coder software uses the test bench to infer the data types.

Why You Must Specify Input Properties

HDL Coder must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, HDL Coder must be able to identify the properties of the inputs to the *primary* function, also known as the *top-level* or *entry-point* function. Therefore, if your primary function has inputs, you must specify the properties of these inputs, to HDL Coder. If your primary function has no input parameters, HDL Coder can compile your MATLAB file without modification. You do not need to specify properties of inputs to local functions or external functions called by the primary function.

Note Your primary function cannot be within a MATLAB namespace. Create a wrapper function as the primary function outside the namespace. Call the desired function within the new function as the primary function.

If you use the tilde (~) character to specify unused function inputs in an HDL Coder project, and you want a different type to appear in the generated code, specify the type. Otherwise, the inputs default to real, scalar doubles.

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages

Method	Advantages	Disadvantages
"Define Input Properties by Example at the Command Line" **Note** If you define input properties programmatically in the MATLAB file, you cannot use this method	• Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a few primary inputs	• Must be specified at the command line every time you invoke (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
"Define Input Properties Using assert Statements in MATLAB Code"	• Integrated with MATLAB code; no need to redefine properties each time you invoke HDL Coder • Provides documentation of property specifications in the MATLAB code • Efficient for specifying memory-intensive inputs such as large structures	• Uses complex syntax • HDL Coder project files do not currently recognize properties defined programmatically. If you are using a project, you must reenter the input types in the project.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input.

For	Specify properties				
	Class	**Size**	**Complexity**	**numerictype**	**fimath**
Fixed-point inputs	✓	✓	✓	✓	✓
Other inputs	✓	✓	✓		

The following data types are not supported for primary function inputs, although you can use them within the primary function:

- structure
- matrix

Variable-size data is not supported in the test bench or the primary function.

Default Property Values

HDL Coder assigns the following default values for properties of primary function inputs.

Property	Default
class	`double`
size	`scalar`
complexity	`real`
`numerictype`	No default

Property	Default
fimath	hdlfimath

Supported Classes

The following table presents the class names supported by HDL Coder.

Class Name	Description
logical	Logical array of true and false values
char	Character array
int8	8-bit signed integer array
uint8	8-bit unsigned integer array
int16	16-bit signed integer array
uint16	16-bit unsigned integer array
int32	32-bit signed integer array
uint32	32-bit unsigned integer array
single	Single-precision floating-point or fixed-point number array
double	Double-precision floating-point or fixed-point number array
embedded.fi	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules:

- You must specify the class of all primary inputs. If you do not specify the size or complexity of primary inputs, they default to real scalars.
- For each primary function input whose class is fixed point (`fi`), you must specify the input `numerictype` and `fimath` properties.

Code Generation Reports

In this section...

"Report Generation" on page 5-7
"Report Location" on page 5-8
"Errors and Warnings" on page 5-8
"Files and Functions" on page 5-8
"MATLAB Source" on page 5-8
"MATLAB Variables" on page 5-9
"Additional Reports" on page 5-10
"Report Limitations" on page 5-10

HDL Coder produces a code generation report that helps you to:

- Debug code generation issues and verify that your MATLAB code is suitable for code generation.
- View generated HDL code.
- See how the code generator determines and propagates type information for variables and expressions in your MATLAB code.
- Access additional reports.

Report Generation

When you enable report generation, the code generator produces a code generation report. To control generation and opening of a code generation report, use app settings, `codegen` options, or configuration object properties.

To view the code generation report, click the **View report** link. If no build errors occur, this report provides links to your MATLAB code, generated C/C++ files, and compile-time type information for the variables in your MATLAB code. If build errors occur, the report lists errors and warnings.

In the HDL Coder app:

1. Open the HDL Coder Workflow Advisor.
2. In the HDL Code Generation step options, on the **Coding Style** tab, under **Generated Code Comments**, select the **Generate report** check box.

At the command line, use `codegen` options:

- To generate a report, use the `-report` option.
- To generate and open a report, use the `-launchreport` option.

Alternatively, use configuration object properties:

- To generate a report, set `GenerateReport` to `true`.
- If you want `codegen` to open the report for you, set `LaunchReport` to `true`.

Report Location

The code generation report is named `report.mldatx`. It is located in the `html` subfolder of the code generation output folder. If you have MATLAB R2018a or later, you can open the `report.mldatx` file by double-clicking it.

Errors and Warnings

View code generation error, warning, and information messages on the **All Messages** tab. To highlight the source code for an error or warning, click the message. It is a best practice to address the first message because subsequent errors and warnings can be related to the first message.

Files and Functions

The report lists MATLAB source functions and generated files. In the **MATLAB Source** pane, the **Function List** view organizes functions according to the containing file. To visualize functions according to the call structure, use the **Call Tree** view.

To view a function in the code pane of the report, click the function in the list. Clicking a function opens the file that contains the function. To edit the selected file in the MATLAB Editor, click **Edit in MATLAB** or click a line number in the code pane.

Specialized Functions or Classes

When a function is called with different types of inputs or a class uses different types for its properties, the code generator produces specializations. In the **MATLAB Source** pane, numbered functions (or classes) indicate specializations. For example:

fx fcn > 1
fx fcn > 2

MATLAB Source

To view a MATLAB function in the code pane, click the name of the function in the **MATLAB Source** pane. In the code pane, when you pause on a variable or expression, a tooltip displays information about its size, type, and complexity. Additionally, syntax highlighting helps you to identify MATLAB syntax elements and certain code generation attributes, such as whether a function is extrinsic or whether an argument is constant.

Extrinsic Functions

The report identifies an extrinsic function with purple text. The tooltip indicates that the function is extrinsic.

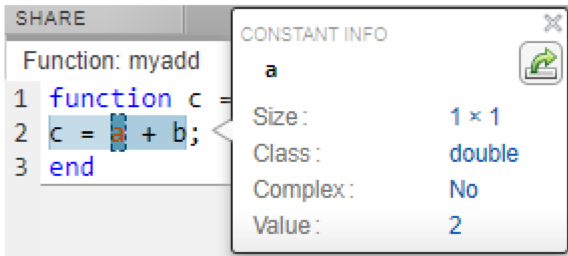
Function: callMyExtrinsic

```
1 function z = callMyExtrinsic(a,b)
2 %#codegen
3 coder.extrinsic('myExtr
4 z = 0;
5 z = myExtrinsic(a,b);
6 disp(z);
7 end
8
```

EXPRESSION INFO
myExtrinsic(a,b)
Size : 1 × 1
Class : mxArray
myExtrinsic *is an extrinsic function.*

Constant Arguments

Orange text indicates a compile-time constant argument to an entry-point function or a specialized function. The tooltip includes the constant value.



Knowing the value of a constant argument helps you to understand the generated function signatures. It also helps you to see when code generation creates function specializations for different constant argument values.

To export the value to a variable in the workspace, click the Export icon .

MATLAB Variables

The **Variables** tab provides information about the variables for the selected MATLAB function. To select a function, click the function in the **MATLAB Source** pane.

The variables table shows:

- Class, size, and complexity
- Properties of fixed-point types

This information helps you to understand type propagation and identify type mismatch errors.

Visual Indicators on the Variables Tab

This table describes the symbols, badges, and other indicators in the variables table.

Column in the Variables Table	Indicator	Description
Name	expander	Variable has elements or properties that you can see by clicking the expander.
Name	`{:}`	Heterogeneous cell array (all elements have the same properties)
Name	`{n}`	nth element of a heterogeneous cell array
Class	`v > n`	`v` is reused with a different class, size, and complexity. The number `n` identifies each unique reuse (a reuse with a unique set of properties). When you pause over a renamed variable, the report highlights only the instances of this variable that share the class, size, and complexity.
Class	`complex` prefix	Complex number
Class		Fixed-point type To see the fixed-point properties, click the badge.

Additional Reports

The **Summary** tab can have links to these additional reports:

- Conformance report
- Resource report
- “HDL Coding Standard Report” on page 24-2

Report Limitations

- The entry-point summary shows the individual elements of `varargin` and `varargout`, but the variables table does not show them.
- The report does not show full information for unrolled loops. It displays data types of one arbitrary iteration.
- The report does not show information about dead code.

See Also

More About

- “Basic HDL Code Generation and FPGA Synthesis from MATLAB”

- “Generate HDL Code from MATLAB Code Using the Command Line Interface”

Generate Instantiable Code for Functions

In this section...

"How to Generate Instantiable Code for Functions" on page 5-12
"Generate Code Inline for Specific Functions" on page 5-12
"Limitations for Instantiable Code Generation for Functions" on page 5-12

You can use the **Generate instantiable code for functions** option to generate a VHDL `entity`, or Verilog or SystemVerilog `module` for each function. The software generates code for each entity or module in a separate file.

How to Generate Instantiable Code for Functions

To enable instantiable code generation for functions in the UI:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task.
2. In the **Advanced** tab, select **Generate instantiable code for functions**.

To enable instantiable code generation for functions programmatically, in your `coder.HdlConfig` object, set the `InstantiateFunctions` property to true. For example, to create a `coder.HdlConfig` object and enable instantiable code generation for functions:

```
hdlcfg = coder.config('hdl');
hdlcfg.InstantiateFunctions = true;
```

Generate Code Inline for Specific Functions

If you want to generate instantiable code for some functions but not others, enable the option to generate instantiable code for functions, and use `coder.inline`. See `coder.inline` for details.

Limitations for Instantiable Code Generation for Functions

The software generates code inline when:

- Function calls are within conditional code or `for` loops.
- Any function is called with a nonconstant `struct` input.
- The function has state, such as a persistent variable, and is called multiple times.
- There is an enumeration anywhere in the design function.

See Also

`coder.HdlConfig` | `coder.FixPtConfig`

Related Examples

- "Generating Modular HDL Code for Functions"

Edit Configuration Parameters for HDL Coder

In this section...

"Create and Modify Configuration Objects" on page 5-13
"Additional Functionalities" on page 5-13

After you have created an HDL Coder code generation configuration object at the command line, you can modify the properties of the object interactively by using the HDL Coder Configuration dialog box.

Create and Modify Configuration Objects

1. Create a HDL Coder configuration object.

   ```
   cfg = coder.config("hdl")
   ```
2. Open the HDL Coder Configuration dialog box by using one of these methods:

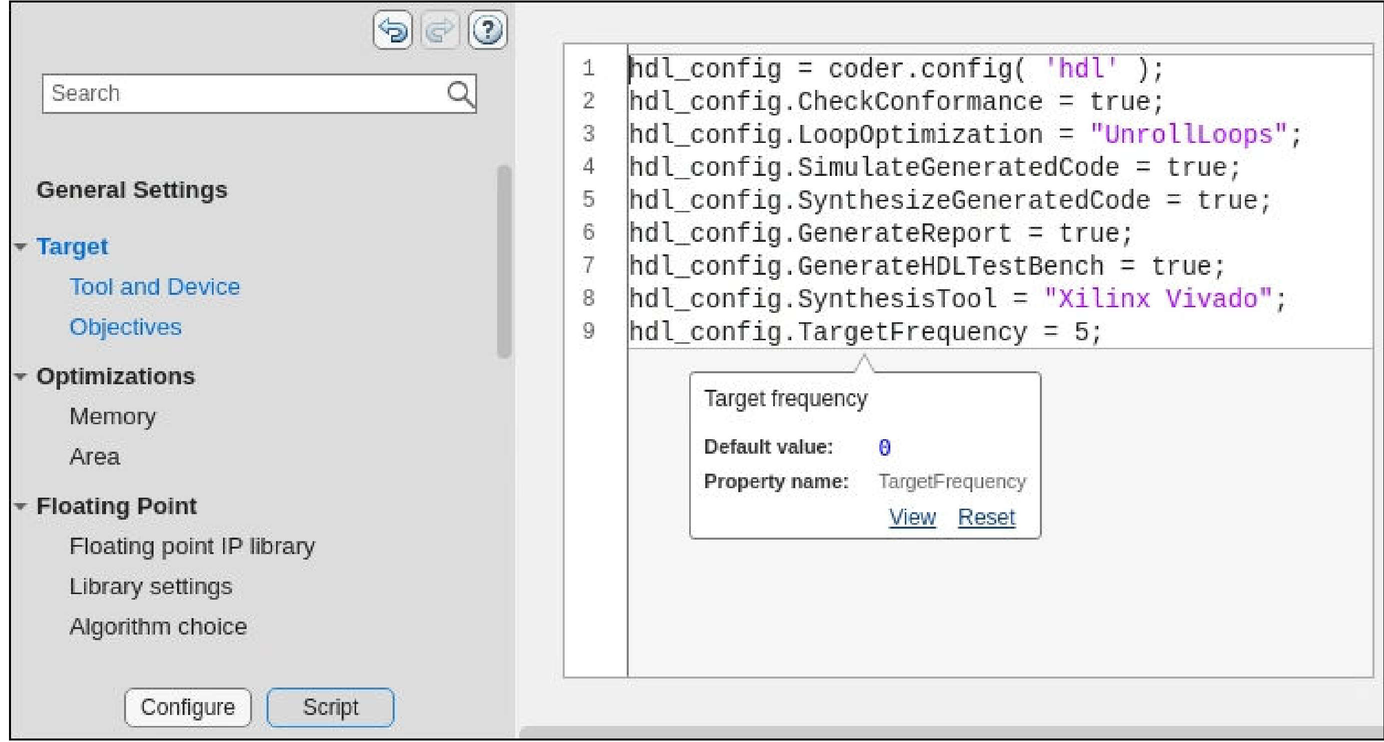
   - In the MATLAB command window, click the **Edit Configuration Object** hyperlink.
   - Double-click the configuration object `cfg` in the MATLAB workspace.
   - At the MATLAB command prompt, open the configuration object `cfg`.

     ```
     open cfg
     ```
3. In the dialog box, modify configuration parameters according to your requirements.

Additional Functionalities

HDL Coder Configuration dialog box provides these functionalities.

- Search: When you search for a string, you see the filtered results across all settings categories. The search string might be present in a setting name, the name of an option for a setting, or in a tooltip.
- Informative tooltips: The tooltip for each setting contains the configuration object property name, a **Help** link for that property, and the name of any additional products that are required for using the property. If the property is disabled, the tooltip also contains links to other properties that you must set to enable this property. You can make that change in the tooltip itself.
- Settings with nondefault values: The dialog box shows settings that have nondefault values in bold font. To reset such a setting to its default values, click the **Reset** button in the tooltip.
- Generate script: You can view the command-line script that produces your current settings by clicking the **Script** button located at the bottom of the list of categories on the left side of the window. You can switch from script mode back to interactive mode by clicking the **Configure** button.



See Also

coder.HdlConfig | coder.CodeConfig | coder.EmbeddedCodeConfig

Integrate Custom HDL Code Into MATLAB Design

`hdl.BlackBox` provides a way to include custom HDL code, such as legacy or handwritten HDL code, in a MATLAB design intended for HDL code generation.

When you create a user-defined System object that inherits from `hdl.BlackBox`, you specify a port interface and simulation behavior that matches your custom HDL code.

HDL Coder simulates the design in MATLAB using the behavior you define in the System object. During code generation, instead of generating code for the simulation behavior, the coder instantiates a module with the port interface you specify in the System object.

To use the generated HDL code in a larger system, you include the custom HDL source files with the rest of the generated code.

In this section...
“Define the hdl.BlackBox System object” on page 5-15
“Use System object In MATLAB Design Function” on page 5-16
“Generate HDL Code” on page 5-17
“Limitations for hdl.BlackBox” on page 5-19

Define the hdl.BlackBox System object

1. Create a user-defined System object that inherits from `hdl.BlackBox`.
2. Configure the black box interface to match the port interface for your custom HDL code by setting `hdl.BlackBox` properties in the System object.
3. Define the `step` method such that its simulation behavior matches the custom HDL code.

   Alternatively, the System object you define can inherit from `hdl.BlackBox` class, and you can define `output` and `update` methods to match the custom HDL code simulation behavior.

Example Code

For example, the following code defines a System object, `CounterBbox`, that inherits from `hdl.BlackBox` and represents custom HDL code for a counter that increments until it reaches a threshold. The `CounterBbox` `reset` and `step` methods model the custom HDL code behavior.

```
classdef CounterBbox < hdl.BlackBox % derive from hdl.BlackBox class
    %Counter: Count up to a threshold.
    %
    % This is an example of a discrete-time System object with state
    % variables.
    %
    properties (Nontunable)
        Threshold = 1
    end

    properties (DiscreteState)
        % Define discrete-time states.
        Count
    end
```

```
    methods
        function obj = CounterBbox(varargin)
            % Support name-value pair arguments
            setProperties(obj,nargin,varargin{:});
            obj.NumInputs = 1;   % define number of inputs
            obj.NumOutputs = 1;  % define number of inputs
        end
    end

    methods (Access=protected)
        % Define simulation behavior.
        % For code generation, the coder uses your custom HDL code instead.
        function resetImpl(obj)
            % Specify initial values for DiscreteState properties
            obj.Count = 0;
        end

        function myout = stepImpl(obj, myin)
            % Implement algorithm. Calculate y as a function of
            % input u and state.
            if (myin > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            myout = obj.Count;
        end
    end
end
```

Use System object In MATLAB Design Function

After you define your System object, use it in the MATLAB design function by creating an instance and calling its `step` method.

To generate code, you also need to create a test bench function that exercises the top-level design function.

Example Code

The following example code shows a top-level design function that creates an instance of the `CounterBbox` and calls its `step` method.

```
function [y1, y2] = topLevelDesign(u)

persistent mybboxObj myramObj
if isempty(mybboxObj)
    mybboxObj = CounterBbox; % instantiate the black box
    myramObj = hdl.RAM('RAMType', 'Dual port');
end

y1 = step(mybboxObj, u); % call the system object step method
[~, y2] = step(myramObj, uint8(10), uint8(0), true, uint8(20));
```

The following example code shows a test bench function for the `topLevelDesign` function.

```
clear topLevelDesign
y1 = zeros(1,200);
y2 = zeros(1,200);
```

```
for ii=1:200
    [y1(ii), y2(ii)] = topLevelDesign(ii);
end
plot([1:200], y2)
```

Generate HDL Code

Generate HDL code using the design function and test bench code.

When you use the generated HDL code, include your custom HDL code with the generated HDL files.

Example Code

In the following generated VHDL code for the `CounterBbox` example, you can see that the `CounterBbox` instance in the MATLAB code maps to an HDL component definition and instantiation, but HDL code is not generated for the `step` method.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY foo IS
  PORT( clk               :   IN    std_logic;
        reset             :   IN    std_logic;
        clk_enable        :   IN    std_logic;
        u                 :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
        ce_out            :   OUT   std_logic;
        y1                :   OUT   real;  -- double
        y2                :   OUT   std_logic_vector(7 DOWNTO 0)  -- uint8
        );
END foo;


ARCHITECTURE rtl OF foo IS

  -- Component Declarations
  COMPONENT CounterBbox
    PORT( clk             :   IN    std_logic;
          clk_enable      :   IN    std_logic;
          reset           :   IN    std_logic;
          myin            :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          myout           :   OUT   real  -- double
          );
  END COMPONENT;

  COMPONENT DualPortRAM_Inst0
    PORT( clk             :   IN    std_logic;
          enb             :   IN    std_logic;
          wr_din          :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          wr_addr         :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          wr_en           :   IN    std_logic;
          rd_addr         :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          wr_dout         :   OUT   std_logic_vector(7 DOWNTO 0);  -- uint8
          rd_dout         :   OUT   std_logic_vector(7 DOWNTO 0)  -- uint8
          );
  END COMPONENT;
```

```
  -- Component Configuration Statements
  FOR ALL : CounterBbox
    USE ENTITY work.CounterBbox(rtl);

  FOR ALL : DualPortRAM_Inst0
    USE ENTITY work.DualPortRAM_Inst0(rtl);

  -- Signals
  SIGNAL enb                : std_logic;
  SIGNAL varargout_1        : real := 0.0;  -- double
  SIGNAL tmp                : unsigned(7 DOWNTO 0);  -- uint8
  SIGNAL tmp_1              : unsigned(7 DOWNTO 0);  -- uint8
  SIGNAL tmp_2              : std_logic;
  SIGNAL tmp_3              : unsigned(7 DOWNTO 0);  -- uint8
  SIGNAL varargout_1_1      : std_logic_vector(7 DOWNTO 0);  -- ufix8
  SIGNAL varargout_2        : std_logic_vector(7 DOWNTO 0);  -- ufix8

BEGIN
  u_CounterBbox : CounterBbox
    PORT MAP( clk => clk,
              clk_enable => enb,
              reset => reset,
              myin => u,  -- uint8
              myout => varargout_1  -- double
              );

  u_DualPortRAM_Inst0 : DualPortRAM_Inst0
    PORT MAP( clk => clk,
              enb => enb,
              wr_din => std_logic_vector(tmp),  -- uint8
              wr_addr => std_logic_vector(tmp_1),  -- uint8
              wr_en => tmp_2,
              rd_addr => std_logic_vector(tmp_3),  -- uint8
              wr_dout => varargout_1_1,  -- uint8
              rd_dout => varargout_2  -- uint8
              );

  enb <= clk_enable;

  y1 <= varargout_1;

  --y2 = u;
  tmp <= to_unsigned(2#00001010#, 8);

  tmp_1 <= to_unsigned(2#00000000#, 8);

  tmp_2 <= '1';

  tmp_3 <= to_unsigned(2#00010100#, 8);

  ce_out <= clk_enable;

  y2 <= varargout_2;

END rtl;
```

Limitations for hdl.BlackBox

You cannot use `hdl.BlackBox` to assign values to a VHDL `generic`, or Verilog or SystemVerilog `parameter` in your custom HDL code.

See Also

`hdl.BlackBox`

Related Examples

- "Generate Board-Independent IP Core from MATLAB Algorithm" on page 5-38

Enable MATLAB Function Block Generation

In this section...

"Requirements for MATLAB Function Block Generation" on page 5-20
"Enable MATLAB Function Block Generation" on page 5-20
"Restrictions for MATLAB Function Block Generation" on page 5-20
"Results of MATLAB Function Block Generation" on page 5-20

Requirements for MATLAB Function Block Generation

During HDL code generation, your MATLAB algorithm must go through the floating-point to fixed-point conversion process, even if it is already a fixed-point algorithm.

Enable MATLAB Function Block Generation

Using the GUI

To enable MATLAB Function block generation using the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, on the left, click **Code Generation**.
2. In the **Advanced** tab, select the **Generate MATLAB Function Black Box** option.

Using the Command Line

To enable MATLAB Function block generation, at the command line, enter:

```
hdlcfg = coder.config('hdl');
hdlcfg.GenerateMLFcnBlock = true;
```

Restrictions for MATLAB Function Block Generation

The top-level MATLAB design function cannot have input or output arguments with the `struct` data type.

Results of MATLAB Function Block Generation

After you generate HDL code, an untitled model opens containing a MATLAB Function block.

You can use the MATLAB Function block as part of a larger model in Simulink for simulation and further HDL code generation.

To learn more about generating a MATLAB Function block from a MATLAB algorithm, see "System Design with HDL Code Generation from MATLAB and Simulink" on page 5-21.

System Design with HDL Code Generation from MATLAB and Simulink

This example shows how to generate a MATLAB Function block from a MATLAB® design for system simulation, code generation, and FPGA programming in Simulink®.

Introduction

HDL Coder can generate HDL code from both MATLAB and Simulink. The coder can also generate a Simulink component, the MATLAB Function block, from your MATLAB code.

This capability enables you to:

1. Design an algorithm in MATLAB;
2. Generate a MATLAB Function block from your MATLAB design;
3. Use the MATLAB component in a Simulink model of the system;
4. Simulate and optimize the system model;
5. Generate HDL code; and
6. Program an FPGA with the entire system design.

In this example, you will generate a MATLAB Function block from MATLAB code that implements a FIR filter.

MATLAB Design

The MATLAB code used in the example is a simple FIR filter. The example also shows a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';
```

1. Design: mlhdlc_fir
2. Test Bench: mlhdlc_fir_tb

Simulate the Design

To simulate the design with the test bench prior to code generation to make sure there are no runtime errors, enter the following command:

mlhdlc_fir_tb

Create a New Project

To create a new HDL Coder project, enter the following command:

```
coder -hdlcoder -new fir_project
```

Next, add the file `mlhdlc_fir.m` to the project as the MATLAB Function and `mlhdlc_fir_tb.m` as the MATLAB Test Bench.

Click the **Workflow Advisor** button to launch the HDL Workflow Advisor.

Enable the MATLAB Function Block Option

To generate a MATLAB Function block from a MATLAB HDL design, you must have a Simulink license. If the following command returns 1, Simulink is available:

```
license('test', 'Simulink')
```

In the HDL Workflow Advisor, select the **HDL Code Generation** task. On the right-side, click the **Advanced** tab and select the check box **Generate Simulink model (Simulink license is required)**.

Run Floating-Point to Fixed-Point Conversion and Generate Code

To generate a MATLAB Function block, convert your design from floating-point to fixed-point.

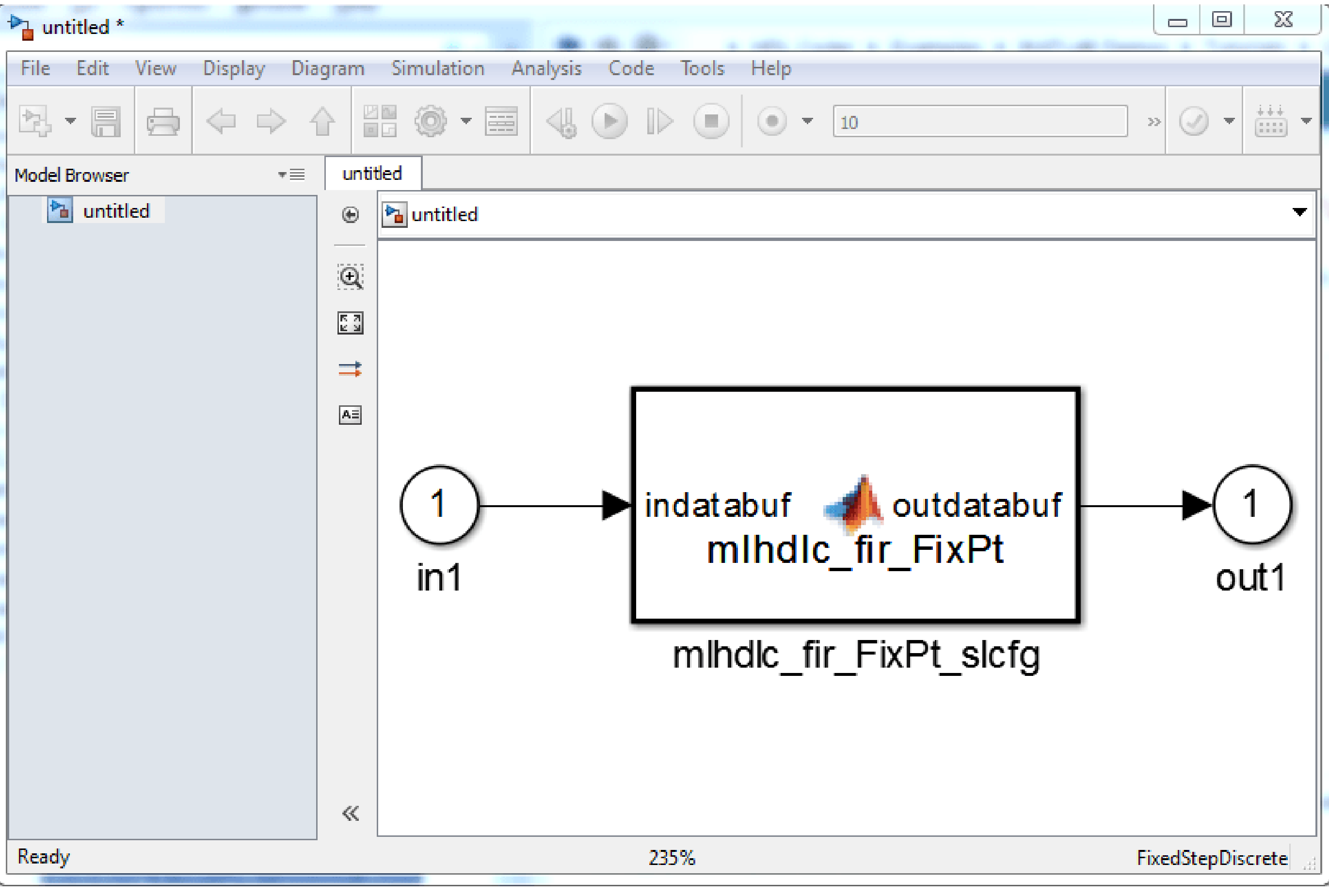
Right-click the **HDL Code Generation** step and choose the option **Run to selected task** to run all the steps from the beginning through HDL code generation.

Examine the Generated MATLAB Function Block

An untitled model opens after HDL code generation. It has a MATLAB Function block containing the fixed-point MATLAB code from your MATLAB HDL design. HDL Coder automatically applies settings to the model and MATLAB Function block so that they can simulate in Simulink and generate HDL code.

To generate HDL code from the MATLAB Function block, enter the following command:

```
makehdl('untitled');
```



You can rename and save the new block to use in a larger Simulink design.

Specify the Clock Enable Rate

In this section...
“Why Specify the Clock Enable Rate?” on page 5-24
“How to Specify the Clock Enable Rate” on page 5-24

Why Specify the Clock Enable Rate?

When HDL Coder performs area optimizations, it might upsample parts of your design (DUT), and thereby introduce an increase in your required DUT clock frequency.

If the coder upsamples your design, it generates a message indicating the ratio between the new clock frequency and your original clock frequency. For example, the following message indicates that your design’s new required clock frequency is 4 times higher than the original frequency:

```
The design requires 4 times faster clock with respect to the base rate = 1
```

This frequency increase introduces a rate mismatch between your input clock enable and output clock enable, because the output clock enable runs at the slower original clock frequency.

With the **Clock enable rate** option, you can choose whether to drive the input clock enable at the faster rate (**DUT base rate**) or at a rate that is less than or equal to the original clock enable rate (**Input data rate**).

How to Specify the Clock Enable Rate

1. In the HDL Workflow Advisor, select **MATLAB to HDL Workflow** > **Code Generation**. Click the **Clocks & Ports** tab.
2. For the **Clock enable rate** option, select **Input data rate** or **DUT base rate**.

Clock enable rate Option	Clock Enable Behavior
Input data rate (default)	Each assertion of the input clock enable produces an output clock enable assertion. You can assert the input clock enable at a maximum rate of once every N clocks. N = the upsampled clock rate / original clock rate. For example, if you see the message, “`The design requires 4 times faster clock with respect to the base rate = 1`”, your maximum input clock enable rate is once every 4 clocks.

Clock enable rate Option	Clock Enable Behavior
DUT base rate	Input clock enable rate does not match the output clock enable rate. You must assert the input clock enable with your input data N times to get 1 output clock enable assertion. N = the upsampled clock rate / original clock rate. For example, if you see the message, “`The design requires 4 times faster clock with respect to the base rate = 1`”, you must assert the input clock enable 4 times to get 1 output clock enable assertion.

Specify Test Bench Clock Enable Toggle Rate

In this section...
"When to Specify Test Bench Clock Enable Toggle Rate" on page 5-26
"How to Specify Test Bench Clock Enable Toggle Rate" on page 5-26

When to Specify Test Bench Clock Enable Toggle Rate

When you want the test bench to drive your input data at a slower rate than the maximum input clock enable rate, specify the test bench clock enable toggle rate.

This specification can help you to achieve better test coverage, and to simulate the real world input data rate.

Note The maximum input clock enable rate is once every N clock cycles. N = the upsampled clock rate / original clock rate. Refer to the clock enable behavior for **Input data rate**, in "Specify the Clock Enable Rate" on page 5-24.

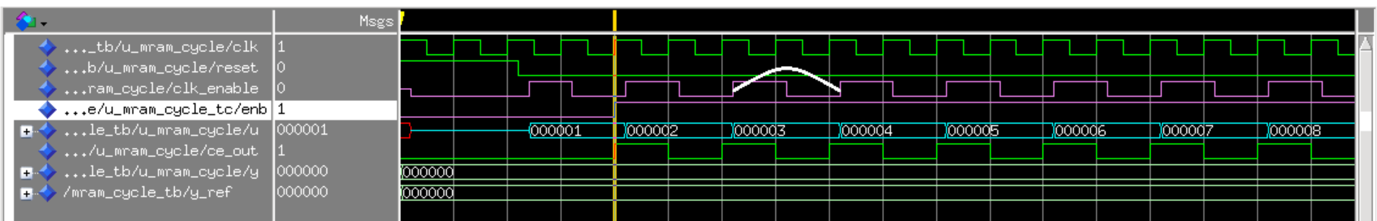
How to Specify Test Bench Clock Enable Toggle Rate

To set your test bench clock enable toggle rate:

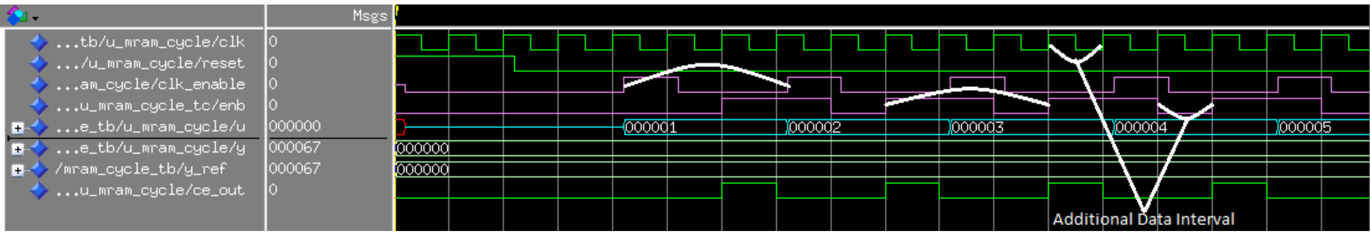
1. In the HDL Workflow Advisor, select **MATLAB to HDL Workflow** > **Code Generation**.
2. In the **Clocks & Ports** tab, for the **Clock enable rate** option, select **Input data rate**.
3. In the **Test Bench** tab, for **Input data interval**, enter 0 or an integer greater than the maximum input clock enable interval.

Input data interval, I	Test Bench Clock Enable Behavior
I = 0 (default)	Asserts at the maximum input clock enable rate, or once every N cycles. N = the upsampled clock rate / original clock rate.
I < N	Not valid; generates an error.
I = N	Same as I = 0.
I > N	Asserts every I clock cycles.

For example, this timing diagram shows clock enable behavior with **Input data interval** = 0. Here, the maximum input clock enable rate is once every 2 cycles.



The following timing diagram shows the same test bench and DUT with **Input data interval** = 3.


Msgs
...tb/u_mram_cycle/clk
.../u_mram_cycle/reset
...am_cycle/clk_enable
...u_mram_cycle_tc/enb
...e_tb/u_mram_cycle/u
...e_tb/u_mram_cycle/y
/mram_cycle_tb/y_ref
...u_mram_cycle/ce_out
000000
000067
000067
000001
000002
000003
000004
000005
Additional Data Interval

Generate an HDL Coding Standard Report from MATLAB

In this section...
“Using the HDL Workflow Advisor” on page 5-28
“Using the Command Line” on page 5-30

You can generate an HDL coding standard report that shows how well your generated code follows industry standards. You can optionally customize the coding standard report and the coding standard rules.

Using the HDL Workflow Advisor

To generate an HDL coding standard report using the HDL Workflow Advisor:

1. In the **HDL Code Generation** task, select the **Coding Standards** tab.
2. For **HDL coding standard**, select **Industry**.

Target | Coding Style | Coding Standards | Clocks & Ports | Optimizations | Advanced | Script Options

Choose coding standard

HDL coding standard: Industry

Report options

- [ ] Do not show passing rules in coding standard report

Basic coding rules

- [x] Check for duplicate names
- [x] Check for HDL keywords in design names
- [x] Check module, instance, entity name length

Minimum: 2

Maximum: 32

- [x] Check signal, port, parameter name length

Minimum: 2

Maximum: 40

RTL description rules

- [ ] Check for clock enable signals
- [ ] Check for reset signals
- [x] Check for asynchronous reset signals
- [ ] Minimize use of variables
- [x] Check for initial statements that set RAM initial values
- [x] Check number of conditional regions

Length: 1

- [x] Check if-else statement chain length

Length: 7

- [x] Check if-else statement nesting depth

Depth: 3

- [x] Check multiplier width

Maximum: 16

RTL design rules

- [x] Check for non-integer constants
- [x] Check line wrap length

3. Optionally, using the other options in the **Coding Standards** tab, customize the coding standard rules.
4. Click **Run** to generate code.

   After you generate code, the message window shows a link to the HTML compliance report.

Using the Command Line

To generate an HDL coding standard report using the command line interface, set the `HDLCodingStandard` property to `Industry` in the `coder.HdlConfig` object.

For example, to generate HDL code and an HDL coding standard report for a design, `mlhdlc_sfir`, with a testbench, `mlhdlc_sfir_tb`, enter the following commands:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_sfir_tb';
hdlcfg.HDLCodingStandard='Industry';
codegen -config hdlcfg mlhdlc_sfir
```

```
### Generating Resource Utilization Report resource_report.html
### Generating default Industry script file mlhdlc_sfir_mlhdlc_sfir_default.prj
### Industry Compliance report with 0 errors, 8 warnings, 4 messages.
### Generating Industry Compliance Report mlhdlc_sfir_Industry_report.html
```

To open the report, click the report link.

You can customize the coding standard report and coding standard rule checks by specifying an HDL coding standard customization object. For example, suppose you have a design, `mlhdlc_sfir`, and testbench, `mlhdlc_sfir_tb`. You can create an HDL coding standard customization object, *`cso`*, set the maximum if-else statement chain length to 5 by using the `IfElseChain` property, and generate code:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_sfir_tb';
hdlcfg.HDLCodingStandard='Industry';
cso = hdlcoder.CodingStandard('Industry');
cso.IfElseChain.length = 5;
hdlcfg.HDLCodingStandardCustomizations = cso;
codegen -config hdlcfg mlhdlc_sfir
```

See Also

Properties
HDL Coding Standard Customization

More About

- “HDL Coding Standard Report” on page 24-2
- “Basic Coding Practices” on page 24-7
- “RTL Description Rules and Checks” on page 24-18
- “RTL Design Methodology Guidelines” on page 24-44

Generate an HDL Lint Tool Script

You can generate a lint tool script to use with a third-party lint tool to check your generated HDL code.

HDL Coder can generate Tcl scripts for the following lint tools:

- Ascent Lint
- HDL Designer
- Leda
- SpyGlass
- Custom

If you specify one of the supported third-party lint tools, you can either generate a default tool-specific script, or customize the script by specifying the initialization, command, and termination names as a character vector. If you want to generate a script for a custom lint tool, you must specify the initialization, command, and termination names.

HDL Coder writes the initialization, command, and termination names to a Tcl script that you can use to run the third-party tool.

How To Generate an HDL Lint Tool Script

Using the HDL Workflow Advisor

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task.
2. In the **Script Options** tab, select **Lint**.
3. For **Choose lint tool**, select **Ascent Lint**, **HDL Designer**, **Leda**, **SpyGlass**, or **Custom**.
4. Optionally, enter text to customize the **Lint script initialization**, **Lint script command**, and **Lint script termination** fields. For a custom tool, you must specify these fields.

After you generate code, the command window shows a link to the lint tool script.

Using the Command Line

To generate an HDL lint tool script from the command line, set the `HDLLintTool` property to `AscentLint`, `HDLDesigner`, `Leda`, `SpyGlass` or `Custom` in your `coder.HdlConfig` object.

To disable HDL lint tool script generation, set the `HDLLintTool` property to `None`.

For example, to generate a default SpyGlass lint script using a `coder.HdlConfig` object, *`hdlcfg`*, enter:

```
hdlcfg.HDLLintTool = 'SpyGlass';
```

After you generate code, the command window shows a link to the lint tool script.

To generate an HDL lint tool script with custom initialization, command, and termination strings, use the `HDLLintTool`, `HDLLintInit`, `HDLLintCmd`, and `HDLLintTerm` properties.

For example, you can use the following command to generate a custom Leda lint script for a DUT subsystem, `sfir_fixed\symmetric_fir`, with custom initialization, termination, and command strings:

```
hdlcfg.HDLLintTool = 'Leda';
hdlcfg.HDLLintInit = 'myInitialization';
hdlcfg.HDLLintCmd = 'myCommand %s';
hdlcfg.HDLLintTerm = 'myTermination';
```

After you generate code, the command window shows a link to the lint tool script.

Custom Lint Tool Command Specification

If you want to generate a lint tool script for a custom lint tool, you must use `%s` as a placeholder for the HDL file name in the generated Tcl script.

For **Lint script command** or `HDLLintCmd`, specify the lint command in the following format:

```
custom_lint_tool_command -option1 -option2 %s
```

For example, to set the `HDLLintCmd` for a `coder.HdlConfig` object, *`hdlcfg`*, where the lint command is *`custom_lint_tool_command -option1 -option2`*, enter:

```
hdlcfg.HDLLintCmd = 'custom_lint_tool_command -option1 -option2 %s';
```

Generate HDL Code from MATLAB Functions That Use Automated Lookup Table Generation

This example shows how to generate HDL code from a floating-point MATLAB® design that is not ready for code generation. Use the fixed-point conversion process by using the `float2fixed` setting with the `codegen` function to generate a lookup table based MATLAB function replacement. Use the new MATLAB replacement function to generate the HDL code.

The MATLAB code in the example is an implementation of a variable exponent function.

Specify Design and Test Bench

Specify the MATLAB design that you are generating HDL code for and the test bench script that simulates and tests the MATLAB function. The `mlhdlc_replacement_exp.m` file contains the design and the exponent function calculations. The MATLAB test bench is in the file `mlhdlc_replacement_exp_tb.m`.

```
design_name = 'mlhdlc_replacement_exp';
testbench_name = 'mlhdlc_replacement_exp_tb';
```

Examine the MATLAB design. The use of the function `exp` is not a supported fixed-point function for HDL code generation. By using automated lookup table generation in the following steps, you can convert the `exp` function to a lookup table that is supported for HDL code generation. For a list of supported functions, see “Functions Supported for HDL and HLS Code Generation” on page 1-2.

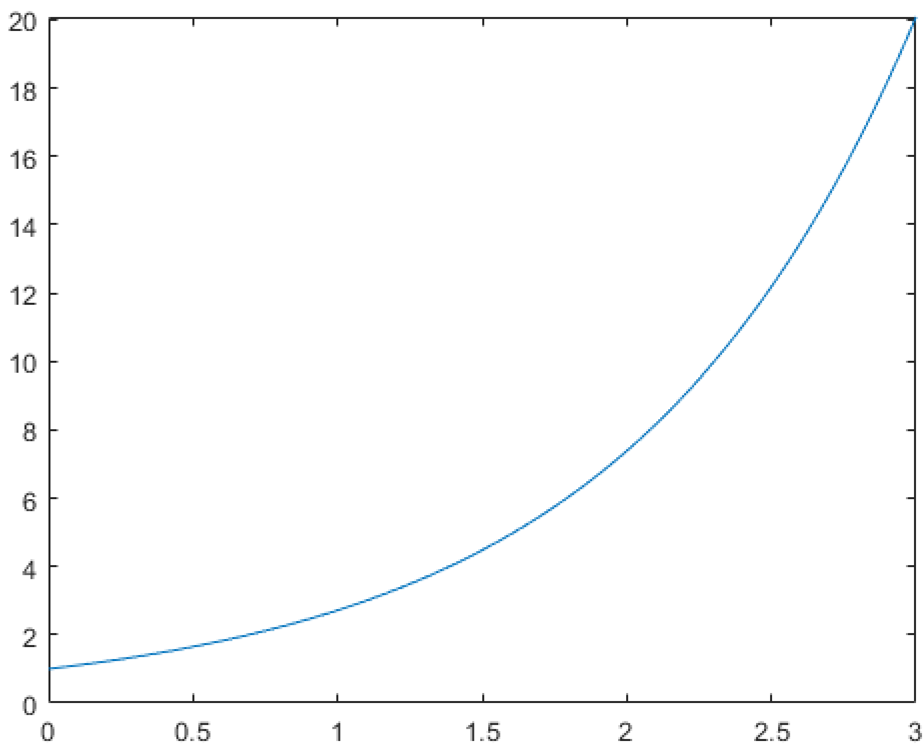
```
dbtype(design_name)
```

```
1     function y = mlhdlc_replacement_exp(u)
2     %
3
4     %   Copyright 2014-2015 The MathWorks, Inc.
5
6     y = exp(u);
7
8     end
```

Simulate the Design

Simulate the design with the test bench to ensure there are no run-time errors.

```
mlhdlc_replacement_exp_tb
```



Generate HDL Code by Using Implicit Fixed-Point Conversion

Create a new script called, copy and paste the following code into the script, and save the file as `runme.m`. The `runme` executes the automated lookup table conversion and generates HDL code by using implicit fixed-point conversion.

```
Set up the path to your installed synthesis tool. This example uses Vivado(R).
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.

clear design_name testbench_name fxpCfg hdlcfg interp_degree
design_name = 'mlhdlc_replacement_exp';
testbench_name = 'mlhdlc_replacement_exp_tb';

interp_degree = 0;

%%    fixed point converter config
fxpCfg = coder.config('fixpt');
fxpCfg.TestBenchName = 'mlhdlc_replacement_exp_tb';
fxpCfg.TestNumerics = true;

%    specify this - for optimized HDL
fxpCfg.DefaultWordLength = 10;

%%    exp - replacement config
mathFcnGenCfg = coder.approximation('exp');
%    generally use to increase accuracy; specify this as power of 2 for optimized HDL
mathFcnGenCfg.NumberOfPoints = 1024;
mathFcnGenCfg.InterpolationDegree = interp_degree; %         can be 0,1,2, or 3
fxpCfg.addApproximation(mathFcnGenCfg);
```

```
%%    HDL config object
hdlcfg = coder.config('hdl');

hdlcfg.TargetLanguage = 'Verilog';

hdlcfg.DesignFunctionName = design_name;
hdlcfg.TestBenchName = testbench_name;
hdlcfg.GenerateHDLTestBench=true;

hdlcfg.SimulateGeneratedCode=true;

%If you choose VHDL set the ModelSim compile options as well
%    hdlcfg.TargetLanguage = 'Verilog';
%    hdlcfg.HDLCompileVHDLCmd = 'vcom %s %s -noindexcheck \n';

hdlcfg.ConstantMultiplierOptimization = 'auto'; %optimize out any multipliers from interpolatio
hdlcfg.PipelineVariables = 'y u idx_bot x x_idx';%

hdlcfg.InputPipeline = 2;
hdlcfg.OutputPipeline = 2;
hdlcfg.RegisterInputs = true;
hdlcfg.RegisterOutputs = true;

hdlcfg.SynthesizeGeneratedCode = true;
hdlcfg.SynthesisTool = 'Xilinx Vivado';
hdlcfg.SynthesisToolChipFamily = 'Virtex7';
hdlcfg.SynthesisToolDeviceName = 'xc7vh580t';
hdlcfg.SynthesisToolPackageName = 'hcg1155';
hdlcfg.SynthesisToolSpeedValue = '-2G';

%codegen('-config',hdlcfg)

codegen('-float2fixed',fxpCfg,'-config',hdlcfg,'mlhdlc_replacement_exp')

%If you only want to do fixed point conversion and stop/examine the
%intermediate results you can use,

%only F2F conversion
codegen('-float2fixed',fxpCfg,'mlhdlc_replacement_exp')
```

Generate a Circuit with a High Clock Rate

To generate a high clock rate circuit for the generated lookup table replacement function, follow these recommendations:

- Use the number of points for the replacement function as a power.
- Set the **ConstantMultiplierOptimization** to `Auto` to allow HDL Coder™ to choose which Constant Multiplier Optimization yields the most area-efficient implementation in the generated HDL code. For more information, see “Constant Multiplier Optimization” on page 8-31.
- Use pipelined variables to minimize the clock delays and improve circuit frequency in the generated HDL code.

Output and Iterative Improvements

Run the `runme.m` script. After the fixed-point conversion completes the function replacements, the script outputs:

```
============= Step1: Analyze floating-point code ==============
```

```
Input types not specified, inferring types by simulating the test bench.

============= Step1a: Verify Floating Point ==============

### Analyzing the design 'mlhdlc_replacement_exp'
### Analyzing the test bench(es) 'mlhdlc_replacement_exp_tb'
### Begin Floating Point Simulation (Instrumented)
### Floating Point Simulation Completed in   1.8946 sec(s)
### Elapsed Time:             2.8361 sec(s)

============= Step2: Propose Types based on Range Information ==============

============= Step3: Generate Fixed Point Code ==============

### Generating Fixed Point MATLAB Code <a href="matlab:edit('codegen/mlhdlc_replacement_exp/fix
### Generating Fixed Point MATLAB Design Wrapper <a href="matlab:edit('codegen/mlhdlc_replaceme
### Generating Mex file for ' mlhdlc_replacement_exp_wrapper_fixpt '
Code generation successful: To view the report, open('codegen/mlhdlc_replacement_exp/fixpt/fxpt
### Generating Type Proposal Report for 'mlhdlc_replacement_exp' <a href="matlab:web('codegen/m

============= Step4: Verify Fixed Point Code ==============

### Begin Fixed Point Simulation : mlhdlc_replacement_exp_tb
### Fixed Point Simulation Completed in   1.9497 sec(s)
### Generating Type Proposal Report for 'mlhdlc_replacement_exp_fixpt' <a href="matlab:web('cod
### Elapsed Time:             2.6488 sec(s)

As this is a small design with only one replacement functions you can
try different number of points in approximation function generation.
Re-examine the generated HDL code and compare it with the previous step.

### Begin VHDL Code Generation
### Generating HDL Conformance Report <a href="matlab:web('codegen/mlhdlc_replacement_exp/hdlsr
### HDL Conformance check complete with 0 errors, 2 warnings, and 0 messages.
### Working on mlhdlc_replacement_exp_fixpt as <a href="matlab:edit('codegen/mlhdlc_replacement
### Generating package file <a href="matlab:edit('codegen/mlhdlc_replacement_exp/hdlsrc/mlhdlc_
### The DUT requires an initial pipeline setup latency. Each output port experiences these addi
### Output port 0: 12 cycles.
### Output port 1: 12 cycles.

### Generating Resource Utilization Report '<a href="matlab:web('codegen/mlhdlc_replacement_exp

### Begin TestBench generation.
### Accounting for output port latency: 12 cycles.'
### Collecting data...
### Begin HDL test bench file generation with logged samples
### Generating test bench: codegen/mlhdlc_replacement_exp/hdlsrc/mlhdlc_replacement_exp_fixpt_t
### Creating stimulus vectors ...

### Simulating the design 'mlhdlc_replacement_exp_fixpt' using 'ModelSim'.
### Generating Compilation Report codegen/mlhdlc_replacement_exp/hdlsrc/mlhdlc_replacement_exp_
### Generating Simulation Report codegen/mlhdlc_replacement_exp/hdlsrc/mlhdlc_replacement_exp_f
### Simulation successful.

### Creating Synthesis Project for 'mlhdlc_replacement_exp_fixpt'.
### Synthesis project creation successful.

### Synthesizing the design 'mlhdlc_replacement_exp_fixpt".
### Generating synthesis report codegen/mlhdlc_replacement_exp/hdlsrc/vivado_prj/mlhdlc_replace
### Synthesis successful.
```

Examine the Synthesis Results

Open up the synthesis results saved in the `mlhdlc_replacement_exp_fixpt_syn_results.txt` file. In the synthesis report, the clock frequency reported by the synthesis tool does not have any optimization options enabled. The synthesis report shows a high clock speed in the order of 300 MHz.

See Also

Related Examples

- “Replace the exp Function with a Lookup Table” on page 4-41
- “Generate HDL-Compatible Lookup Table Function Replacements Using coder.approximate” on page 4-78

Generate Board-Independent IP Core from MATLAB Algorithm

In this section...

"Requirements and Limitations for IP Core Generation" on page 5-38
"Generate Board-Independent IP Core" on page 5-38

When you open the HDL Workflow Advisor and run the `IP Core Generation` workflow for your Simulink model, you can specify a generic Xilinx® platform or a generic Intel® platform. The workflow then generates a generic IP core that you can integrate into any target platform of your choice. For IP core integration, define and register a custom reference design for your target board.

Requirements and Limitations for IP Core Generation

You cannot generate an HDL IP core without any AXI4 slave interface. At least one DUT port must map to an AXI4 or AXI4-Lite interface. To generate an HDL IP core without any AXI4 slave interfaces, use the Simulink IP core generation workflow. For more information, see "Generate Board-Independent HDL IP Core from Simulink Model" on page 39-33.

In the same IP core, you cannot map to both an AXI4 interface and AXI4-Lite interface.

AXi4-Lite Interface Restrictions

- The inputs and outputs must have a bit width less than or equal to 32 bits.
- The input and outputs must be scalar.

AXI4-Stream Video Interface Restrictions

- Ports must have a 32-bit width.
- Ports must be scalar.
- You can have a maximum of one input video port and one output video port.
- The AXI4-Stream Video interface is not supported in **Coprocessing – blocking Processor/FPGA synchronization** must be set to `Free running` mode. `Coprocessing – blocking` mode is not supported.

Generate Board-Independent IP Core

To generate a board-independent IP core to use in an embedded system integration environment, such as Intel Qsys, Xilinx EDK, or Xilinx IP Integrator:

1. Create an HDL Coder project containing your MATLAB design and test bench, or open an existing project.
2. In the HDL Workflow Advisor, define input types and perform fixed-point conversion.

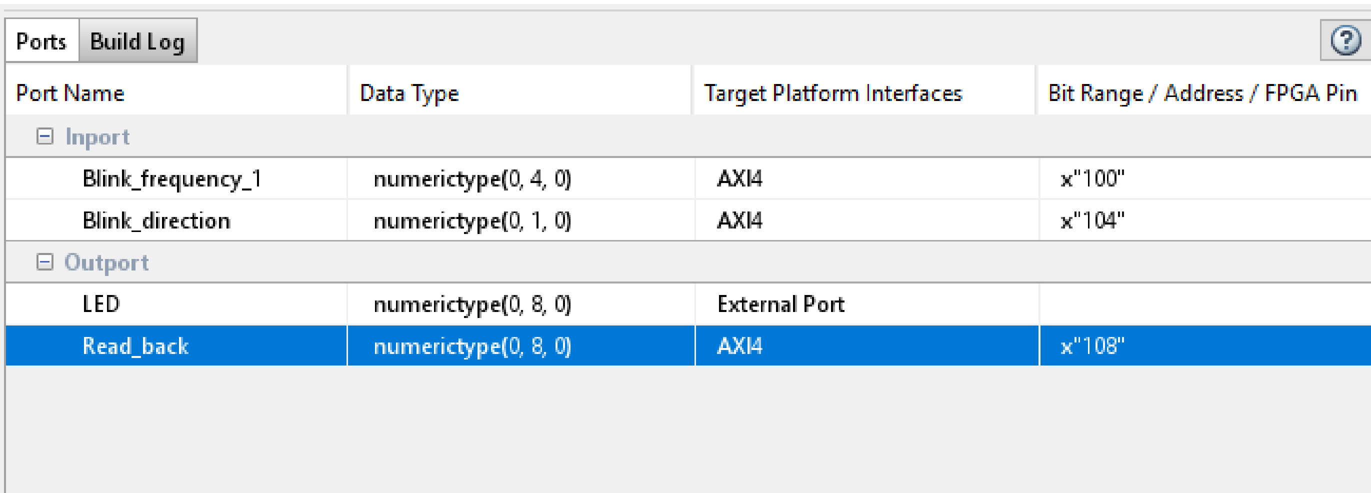
   To learn how to convert your design to fixed-point, see "Basic HDL Code Generation and FPGA Synthesis from MATLAB".
3. In the HDL Workflow Advisor, in the **Select Code Generation Target** task:

   - **Workflow**: Select `IP Core Generation`.
   - **Platform**: Select `Generic Xilinx Platform` or `Generic Altera Platform`.

Depending on your selection, the code generator automatically sets the **Synthesis tool**. For example, if you select `Generic Xilinx Platform`, **Synthesis tool** automatically changes to `Xilinx Vivado`.

- **Additional source files**: If you are using an `hdl.BlackBox` System object to include existing Verilog, SystemVerilog or VHDL code, enter the file names. Enter each file name manually, separated with a semicolon (`;`), or by using the **...** button.

4. In the **Set Target Interface** step, for each port, select an option from the **Target Platform Interfaces** drop-down list.



5. In the **HDL Code Generation** step, optionally specify code generation options, then click **Run**.

   In the HDL Workflow Advisor message pane, click the IP core report link to view detailed documentation for your generated IP core.

See Also

Classes
`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Generate IP Core from MATLAB for Blinking LEDs on FPGA Board” on page 39-180

More About

- “Custom IP Core Generation” on page 39-17
- “Board and Reference Design Registration System” on page 40-89

Minimize Clock Enables

In this section...

"Using the GUI" on page 5-40
"Using the Command Line" on page 5-40
"Limitations" on page 5-41

By default, HDL Coder generates code in a style that is intended to map to registers with clock enables, and the DUT has a top-level clock enable port.

If you do not want to generate registers with clock enables, you can minimize the clock enable logic. For example, if your target hardware contains registers without clock enables, you can save hardware resources by minimizing the clock enable logic.

The following VHDL code shows the default style of generated code, which uses clock enables. The `enb` signal is the clock enable:

```
Unit_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        Unit_Delay_out1 <= In1_signed;
      END IF;
    END IF;
  END PROCESS Unit_Delay_process;
```

The following VHDL code shows the style of code you generate if you minimize clock enables:

```
Unit_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      Unit_Delay_out1 <= In1_signed;
    END IF;
  END PROCESS Unit_Delay_process;
```

Using the GUI

To minimize clock enables, in the HDL Workflow Advisor, on the **HDL Code Generation** > **Clocks & Ports** tab, select **Minimize clock enables**.

Using the Command Line

To minimize clock enables, in the `coder.HdlConfig` configuration object, set the `MinimizeClockEnables` property to `true`. For example:

```
hdlCfg = coder.config('hdl')
hdlCfg.MinimizeClockEnables = true;
```

Limitations

If you specify area optimizations that the coder implements by increasing the clock rate in certain regions of the design, you cannot minimize clock enables. The following optimizations prevent clock enable minimization:

- Resource sharing
- RAM mapping
- Loop streaming

Clock enable minimization is also prevented by setting **RAM architecture** to **Generic RAM without clock enable** in the HDL Workflow Advisor on the **HDL Code Generation > Advanced** tab.

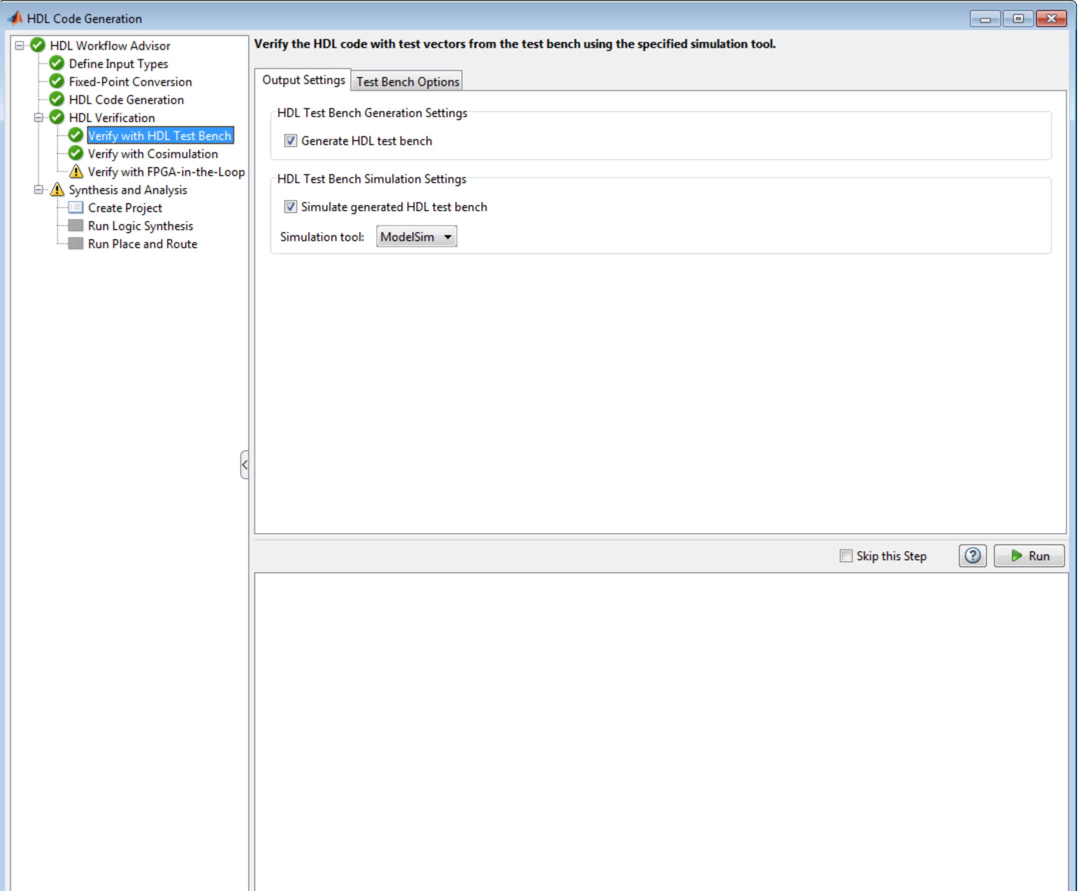
Verification

- “Verify Code with HDL Test Bench” on page 6-2
- “Test Bench Generation” on page 6-5
- “Resolve Index Errors During Simulation” on page 6-6

Verify Code with HDL Test Bench

Simulate the generated HDL design under test (DUT) with test vectors from the test bench using the specified simulation tool.

1 Start the MATLAB to HDL Workflow Advisor.



2 At step **HDL Verification**, click **Verify with HDL Test Bench**.
3 Select **Generate HDL test bench**.

This option enables HDL Coder to generate HDL test bench code from your MATLAB test script.

4 Optionally, select **Simulate generated HDL test bench**. This option enables MATLAB to simulate the HDL test bench with the HDL DUT.

If you select this option, you must also select the **Simulation tool**.

5 For **Test Bench Options**, select and set the optional parameters according to the descriptions in the following table.

HDL Test Bench Parameter	Description
Test bench name postfix	Specify the postfix for the test bench name.

HDL Test Bench Parameter	Description
Force clock	Enable for test bench to force clock input signals.
Clock high time (ns)	Specify the number of nanoseconds the clock is high.
Clock low time (ns)	Specify the number of nanoseconds the clock is low.
Hold time (ns)	Specify the hold time for input signals and forced reset signals.
Force clock enable	Enable to force clock enable.
Clock enable delay (in clock cycles)	Specify time (in clock cycles) between deassertion of reset and assertion of clock enable.
Force reset	Enable for test bench to force reset input signals.
Reset length (in clock cycles)	Specify time (in clock cycles) between assertion and deassertion of reset.
Hold input data between samples	Enable to hold subrate signals between clock samples.
Input data interval	Specifies the number of clock cycles between assertions of clock enable. For more information, see "Specify Test Bench Clock Enable Toggle Rate" on page 5-26.
Initialize test bench inputs	Enable to initialize values on inputs to test bench before test bench drives data to DUT.
Multi file test bench	Enable to divide generated test bench into helper functions, data, and HDL test bench code.
Test bench data file name postfix	Specify the character vector to append to name of test bench data file when generating multi-file test bench.
Test bench reference postfix	Specify the character vector to append to names of reference signals in test bench code.
Ignore data checking (number of samples)	Specify the number of samples at the beginning of simulation during which output data checking is suppressed.
Simulation iteration limit	Specify the maximum number of test samples to use during simulation of generated HDL code.

6. Optionally, select **Skip this step** if you don't want to use the HDL test bench to verify the HDL DUT.
7. Click **Run**.

If the test bench and simulation is successful, you should see messages similar to these in the message pane:

```
### Begin TestBench generation.
### Collecting data...
### Begin HDL test bench file generation with logged samples
### Generating test bench: mlhdlc_sfir_fixpt_tb.vhd
### Creating stimulus vectors...
### Simulating the design 'mlhdlc_sfir_fixpt' using 'ModelSim'.
### Generating Compilation Report mlhdlc_sfir_fixpt_vsim_log_compile.txt
### Generating Simulation Report mlhdlc_sfir_fixpt_vsim_log_sim.txt
### Simulation successful.
### Elapsed Time: 113.0315 sec(s)
```

If there are errors, those messages appear in the message pane. Fix errors and click **Run**.

Test Bench Generation

In this section...

"How Test Bench Generation Works" on page 6-5
"Test Bench Data Files" on page 6-5
"Test Bench Data Type Limitations" on page 6-5
"Use Constants Instead of File I/O" on page 6-5

How Test Bench Generation Works

HDL Coder writes the DUT stimulus and reference data from your MATLAB or Simulink simulation to data files (`.dat`).

During HDL simulation, the HDL test bench reads the saved stimulus from the `.dat` files. The test bench compares the actual DUT output with the expected output, which is also saved in .dat files. After you generate code, the message window displays links to the test bench data files.

Reference data is delayed by one clock cycle in the waveform viewer compared to default test bench generation due to the delay in reading data from files.

Test Bench Data Files

The coder saves stimulus and reference data for each DUT input and output in a separate test bench data file (`.dat`), with the following exceptions:

- Two files are generated for the real and imaginary parts of complex data.
- Constant DUT input data is written to the test bench as constants.

Vector input or output data is saved as a single file.

Test Bench Data Type Limitations

If you have double, single, or enumeration data types at the DUT inputs and outputs, the simulation data is generated as constants in the test bench code, instead of writing the simulation data to files.

Use Constants Instead of File I/O

You can generate test bench stimulus and reference data as constants in the test bench code instead of using file I/O. However, simulating a long running test bench that uses constants requires more memory than a test bench that uses file I/O.

Test bench generation automatically generates data as constants if your DUT inputs or outputs use data types that are not supported for file I/O. For details, see "Test Bench Data Type Limitations" on page 6-5.

To generate a test bench that uses constants instead of file I/O:

1. In the HDL Workflow Advisor, select the **HDL Verification** > **Verify with HDL Test Bench** task.
2. In the **Test bench Options** tab, disable the **Use file I/O for test bench** option.

Resolve Index Errors During Simulation

Issue

If you use a simulation tool to verify HDL code generated from a MATLAB algorithm or a Simulink model that uses a MATLAB Function block, the array indices may not always be resolved and may often be 0. MATLAB displays this error in the Command Window.

```
Simulation failed. See report.
```

The error originates from an error in the simulator. This error can take the form:

```
# ** Fatal: (vsim-3421) Value -1 is out of range 0 to 16383.
```

Possible Solutions

The error may occur during simulation when the simulator executes an index operation while propagating signals. Because the simulator is propagating signals, indices may be unresolved. Consequently, when the simulator performs a subtraction operation on an unresolved index and then accesses an array with an out-of-bounds index, the simulator generates index errors.

To resolve the error, try one of these solutions:

- Follow the indexing best practices for MATLAB code for HDL code generation to minimize automatic index conversions. For more information, see “Optimize Indexing When Verifying Code with Simulation Tools” on page 6-6. If you optimize the indexing in your MATLAB algorithm or MATLAB code in the MATLAB Function block, it can result in fewer index errors during simulation.
- Enable the **Suppress out of bounds access errors by generating simulation-only index checks** configuration parameter during HDL code generation. For more information, see “Use SimIndexCheck to Suppress Index Errors” on page 6-9.

Optimize Indexing When Verifying Code with Simulation Tools

Optimize the indexing in your MATLAB algorithm or MATLAB code in the MATLAB Function block. For more information, see “Indexing Best Practices for HDL Code Generation” on page 1-68.

For example, this MATLAB algorithm enhances the contrast of images by transforming the values in an intensity image so that the histogram of the output image is approximately flat. For more information on this algorithm, see “Image Enhancement by Histogram Equalization” on page 2-51. To illustrate this issue, simulate the HDL code of the unoptimized and optimized algorithms and compare the results.

Unoptimized MATLAB Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mlhdlc_heq.m
% Histogram Equalization Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out, y_out, pixel_out] = ...
    mlhdlc_heq(x_in, y_in, pixel_in, width, height)

% Copyright 2011-2024 The MathWorks, Inc.

persistent histogram
persistent transferFunc
persistent histInd
persistent cumSum

if isempty(histogram)
    histogram = zeros(1, 2^14);
    transferFunc = zeros(1, 2^14);
    histInd = 0;
    cumSum = 0;
end

% Figure out indexes based on where we are in the frame
if y_in < height && x_in < width % valid pixel data
    histInd = pixel_in + 1;
elseif y_in == height && x_in == 0 % first column of height+1
    histInd = 1;
elseif y_in >= height % vertical blanking period
    histInd = min(histInd + 1, 2^14);
elseif y_in < height % horizontal blanking - do nothing
    histInd = 1;
end

% Read histogram (must be outside conditional logic)
histValRead = histogram(histInd);

% Read transfer function (must be outside conditional logic)
transValRead = transferFunc(histInd);

% If valid part of frame add one to pixel bin
% and keep transfer func val
if y_in < height && x_in < width
    histValWrite = histValRead + 1; % Add pixel to bin
    transValWrite = transValRead; % Write back same value
    cumSum = 0;
elseif y_in >= height % In blanking time index through all bins and reset to zero
    histValWrite = 0;
    transValWrite = cumSum + histValRead;
    cumSum = transValWrite;
else
    histValWrite = histValRead;
    transValWrite = transValRead;
end

% Write histogram (must be outside conditional logic)
histogram(histInd) = histValWrite;

% Write transfer function (must be outside conditional logic)
transferFunc(histInd) = transValWrite;

pixel_out = transValRead;
x_out = x_in;
y_out = y_in;
```

Optimized MATLAB Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mlhdlc_heq_bp.m
% Histogram Equalization Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out, y_out, pixel_out] = ...
    mlhdlc_heq_bp(x_in, y_in, pixel_in, width, height)

% Copyright 2011-2024 The MathWorks, Inc.

persistent histogram
persistent transferFunc
persistent histInd
persistent cumSum

if isempty(histogram)
    histogram = zeros(1, 2^14);
    transferFunc = zeros(1, 2^14);
    histInd = 0;
    cumSum = 0;
end

% Adjust histInd index variable to be 0-based
if y_in < height && x_in < width
    histInd = pixel_in;
elseif y_in == height && x_in == 0
    histInd = 0;
elseif y_in >= height
    histInd = min(histInd + 1, 2^14-1);
elseif y_in < height
    histInd = 0;
end

% Add 1 to the value of the 0-based histInd when indexing
histValRead = histogram(histInd+1);

% Add 1 to the value of the 0-based histInd when indexing
transValRead = transferFunc(histInd+1);

if y_in < height && x_in < width
    histValWrite = histValRead + 1;
    transValWrite = transValRead;
    cumSum = 0;
elseif y_in >= height
    histValWrite = 0;
    transValWrite = cumSum + histValRead;
    cumSum = transValWrite;
else
    histValWrite = histValRead;
    transValWrite = transValRead;
end

% Add 1 to the value of the 0-based histInd when indexing
histogram(histInd+1) = histValWrite;

% Add 1 to the value of the 0-based histInd when indexing
transferFunc(histInd+1) = transValWrite;

pixel_out = transValRead;
x_out = x_in;
y_out = y_in;
```

1. Run the command `mlhdlc_demo_setup('heq')` in the MATLAB Command Window. This command copies the files `mlhdlc_heq.m` and `mlhdlc_heq_tb.m` into a temporary working folder.
2. Create a new MATLAB function in the temporary working folder named `mlhdlc_heq_bp.m`. Copy the optimized MATLAB algorithm from the table into the function.
3. Create a new MATLAB script in the temporary working folder named `mlhdlc_heq_bp_tb.m`. Copy the contents of the `mlhdlc_heq_tb.m` testbench into `mlhdlc_heq_bp_tb.m`. Change the line with the function call `mlhdlc_heq` to `mlhdlc_heq_bp` in order for the

mlhdlc_heq_bp_tb.m testbench to exercise the function design of the optimized MATLAB algorithm, mlhdlc_heq_bp.

4 Generate and simulate the HDL code from the unoptimized MATLAB algorithm, mlhdlc_heq. Set the fixptcfg and hdlcfg object parameters and run the codegen command:

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_heq_tb';

hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_heq_tb';
hdlcfg.MapPersistentVarsToRAM = 0;
hdlcfg.TargetLanguage = 'VHDL';

hdlcfg.GenerateHDLTestBench = true;
hdlcfg.SimulateGeneratedCode = true;
hdlcfg.SimulationTool = 'ModelSim';

codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_heq
```

Alternatively, you can use the MATLAB HDL Workflow Advisor. For more information, see “Verify Code with HDL Test Bench” on page 6-2.

5 Note the error in the MATLAB Command Window.

```
### Simulating the design 'mlhdlc_heq_fixpt' using 'ModelSim'.
### Generating Compilation Report mlhdlc_heq_fixpt_vsim_log_compile.txt
### Generating Simulation Report mlhdlc_heq_fixpt_vsim_log_sim.txt
Error occurred when running post codegeneration tasks
### Generating HDL Conformance Report mlhdlc_heq_fixpt_hdl_conformance_report.html.
### HDL Conformance check complete with 1 errors, 0 warnings, and 0 messages.
### Coder:hdl:post_codegen: Error: failed to run post code generation tasks:
Coder:FXPCONV:SimulationFailure Simulation failed. See report.
```

The error originates from an error in the simulator, in this instance ModelSim®. In the MATLAB Command Window, click mlhdlc_heq_fixpt_vsim_log_sim.txt to view the simulation report. Note the error from the simulator.

```
# ** Fatal: (vsim-3421) Value -1 is out of range 0 to 16383.
```

6 Generate and simulate the HDL code from the optimized MATLAB algorithm, mlhdlc_heq_bp. Set the fixptcfg and hdlcfg object parameters and run the codegen command:

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_heq_bp_tb';

hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_heq_bp_tb';
hdlcfg.MapPersistentVarsToRAM = 0;
hdlcfg.TargetLanguage = 'VHDL';

hdlcfg.GenerateHDLTestBench = true;
hdlcfg.SimulateGeneratedCode = true;
hdlcfg.SimulationTool = 'ModelSim';

codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_heq_bp
```

The code simulates without error.

```
### Simulating the design 'mlhdlc_heq_bp_fixpt' using 'ModelSim'.
### Generating Compilation Report mlhdlc_heq_bp_fixpt_vsim_log_compile.txt
### Generating Simulation Report mlhdlc_heq_bp_fixpt_vsim_log_sim.txt
### Simulation successful.
```

Use SimIndexCheck to Suppress Index Errors

HDL Coder may not optimize all instances of index conversions. If the index simulation error persists, you can use the **Suppress out of bounds access errors by generating simulation-only index checks** configuration parameter during HDL code generation. If you enable this parameter, HDL Coder generates additional logic that runs when you simulate your HDL code to prevent array indices from going out of bounds. Enabling this parameter disables all index checking during simulation.

See Also

Related Examples

- "Image Enhancement by Histogram Equalization" on page 2-51

7

Deployment

Generate Synthesis Scripts

You can generate customized synthesis scripts for the following tools:

- Xilinx Vivado®
- Xilinx ISE
- Microchip Libero
- Mentor Graphics® Precision
- Altera® Quartus II
- Synopsys® Synplify Pro®
- Intel Quartus® Pro

You can also generate a synthesis script for a custom tool by specifying the fields manually.

To generate a synthesis script:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task.
2. In the **Script Options** tab, select **Synthesis**.
3. For **Choose synthesis tool**, select a tool option.
4. If you want to customize your script, use the **Synthesis file postfix**, **Synthesis initialization**, **Synthesis command**, and **Synthesis termination** text fields to do so.

After you generate code, your synthesis Tcl script (`.tcl`) is in the same folder as your generated HDL code.

Optimization

- “Map Matrices to Block RAMs to Reduce Area” on page 8-2
- “Map Persistent Arrays and dsp.Delay Objects to RAM” on page 8-6
- “Pipelining MATLAB Code” on page 8-11
- “Pipeline MATLAB Expressions” on page 8-12
- “Distributed Pipelining” on page 8-14
- “Distributed Pipelining for Clock Speed Optimization” on page 8-15
- “Optimize Clock Speed for MATLAB Code by Using Adaptive Pipelining” on page 8-19
- “Optimize Feedback Loop Design and Maintain High Data Precision for HDL Code Generation” on page 8-26
- “Optimize MATLAB Loops” on page 8-29
- “Constant Multiplier Optimization” on page 8-31
- “Resource Sharing of Multipliers to Reduce Area” on page 8-33
- “Loop Streaming to Reduce Area” on page 8-39
- “Constant Multiplier Optimization to Reduce Area” on page 8-44

Map Matrices to Block RAMs to Reduce Area

This example shows how to use the RAM mapping optimization in HDL Coder™ to map persistent matrix variables to block RAMs in hardware.

Introduction

One of the attractive features of writing MATLAB code is the ease of creating, accessing, modifying and manipulating matrices in MATLAB.

When processing such MATLAB code, HDL Coder maps these matrices to wires or registers in HDL. For example, local temporary matrix variables are mapped to wires, whereas persistent matrix variables are mapped to registers.

The latter tends to be an inefficient mapping when the matrix size is large, since the number of register resources available is limited. It also complicates synthesis, placement and routing.

Modern FPGAs feature block RAMs that are designed to have large matrices. HDL Coder takes advantage of this feature and automatically maps matrices to block RAMs to improve area efficiency. For certain designs, mapping these persistent matrices to RAMs is mandatory if the design is to be realized. State-of-the-art synthesis tools may not be able to synthesize designs when large matrices are mapped to registers, whereas the problem size is more manageable when the same matrices are mapped to RAMs.

MATLAB Design

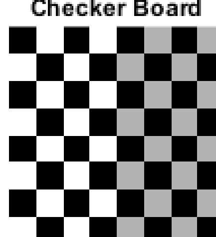
```
design_name = 'mlhdlc_sobel';
testbench_name = 'mlhdlc_sobel_tb';
```

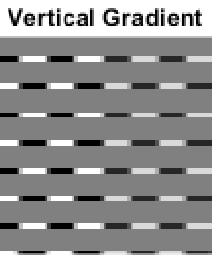
- MATLAB Design: mlhdlc_sobel
- MATLAB Testbench: mlhdlc_sobel_tb
- Input Image: stop_sign

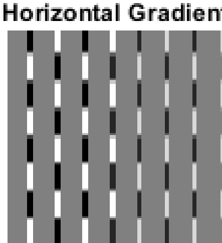
Simulate the Design

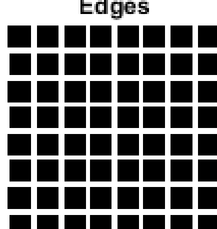
Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sobel_tb
```









Create a New HDL Coder™ Project

Run the following command to create a new project.

```
coder -hdlcoder -new mlhdlc_ram
```

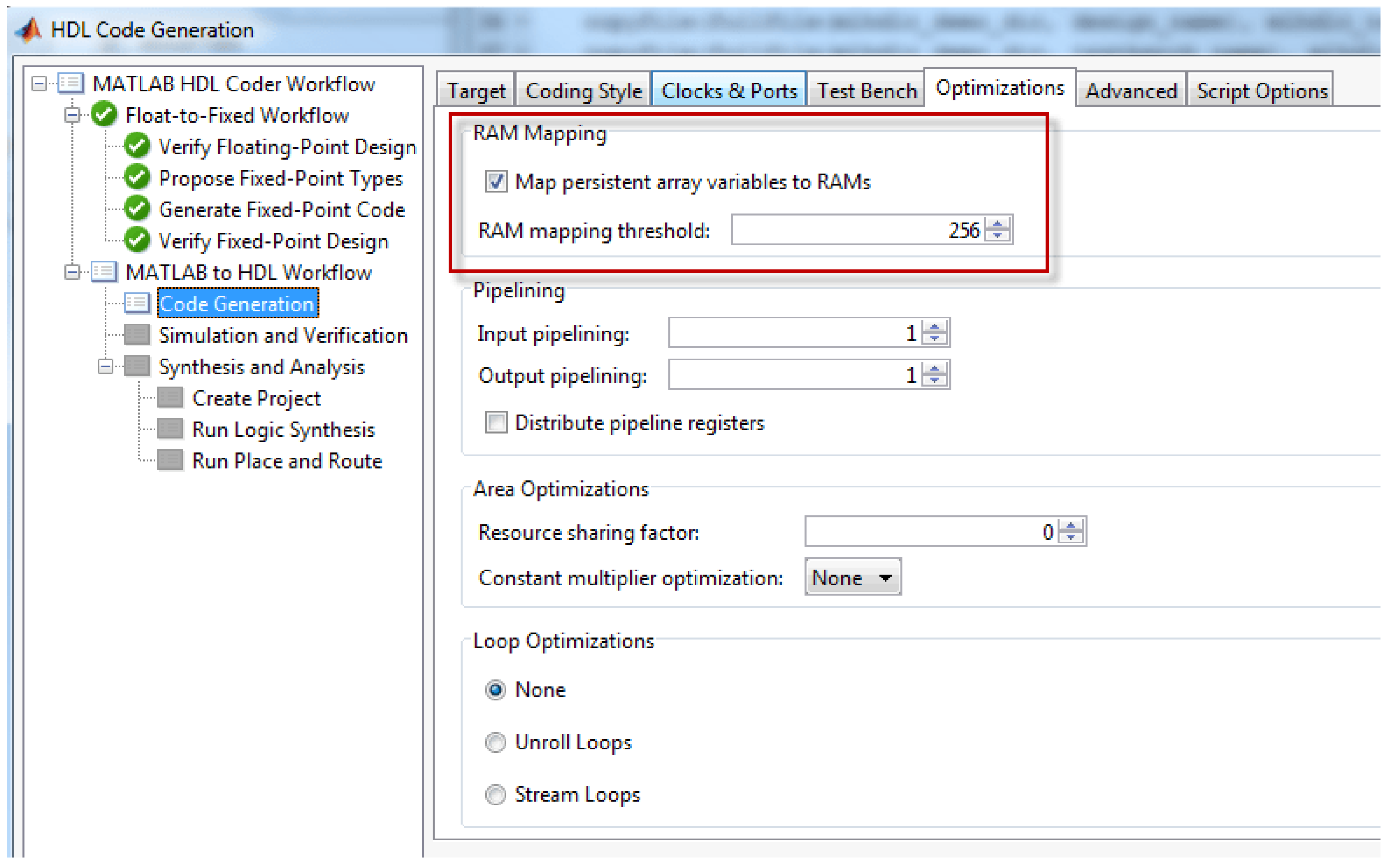
Next, add the file 'mlhdlc_sobel.m' to the project as the MATLAB function, and 'mlhdlc_sobel_tb.m' as the MATLAB test bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Turn On the RAM Mapping Optimization

Launch the Workflow Advisor.

The checkbox 'Map persistent array variables to RAMs' needs to be turned on to map persistent variables to block RAMs in the generated code.



Run Fixed-Point Conversion and HDL Code Generation

In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the Generated Code

Examine the messages in the log window to see the RAM files generated along with the design.

Run

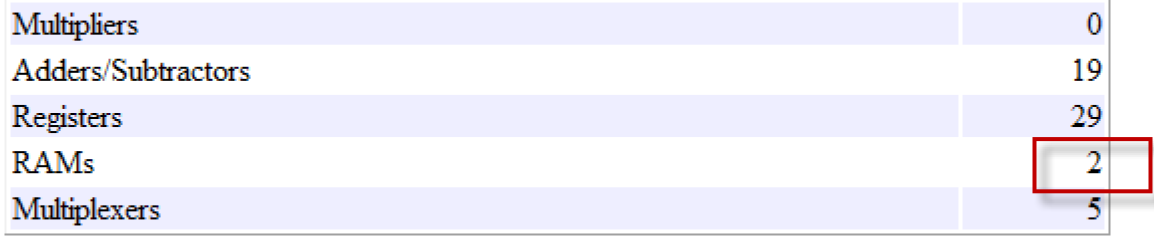
```
### Begin VHDL Code Generation
### Working on mlhdlc_sobel_FixPt/u_d_ram/DualPortRAM_128x9b as DualPortRAM_128x9b.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram as u_d_ram.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram/DualPortRAM_128x9b as DualPortRAM_128x9b_block.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram as u_d_ram_block.vhd
### Working on mlhdlc_sobel_FixPt as mlhdlc_sobel_FixPt.vhd
### Generating package file mlhdlc_sobel_FixPt_pkg.vhd
### The DUT requires an initial pipeline setup latency. Each output port experiences these
additional delays
### Output port 0: 4 cycles
### Output port 1: 4 cycles
### Generating Resource Utilization Report resource_report.html
### Elapsed Time: 33.1382 sec(s)
```

Help

A warning message appears for each persistent matrix variable not mapped to RAM.

Examine the Resource Report

Take a look at the generated resource report, which shows the number of RAMs inferred, by following the 'Resource Utilization report...' link in the generated code window.



Multipliers	0
Adders/Subtractors	19
Registers	29
RAMs	2
Multiplexers	5

Additional Notes on RAM Mapping

- Persistent matrix variable accesses must be in unconditional regions, i.e., outside any if-else, switch case, or for-loop code.
- MATLAB functions can have any number of RAM matrices.
- All matrix variables in MATLAB that are declared persistent and meet the threshold criteria get mapped to RAMs.
- A warning is shown when a persistent matrix does not get mapped to RAM.
- Read-dependent write data cycles are not allowed: you cannot compute the write data as a function of the data read from the matrix.

- Persistent matrices cannot be copied as a whole or accessed as a sub matrix: matrix access (read/write) is allowed only on single elements of the matrix.
- Mapping persistent matrices with non-zero initial values to RAMs is not supported.

Map Persistent Arrays and dsp.Delay Objects to RAM

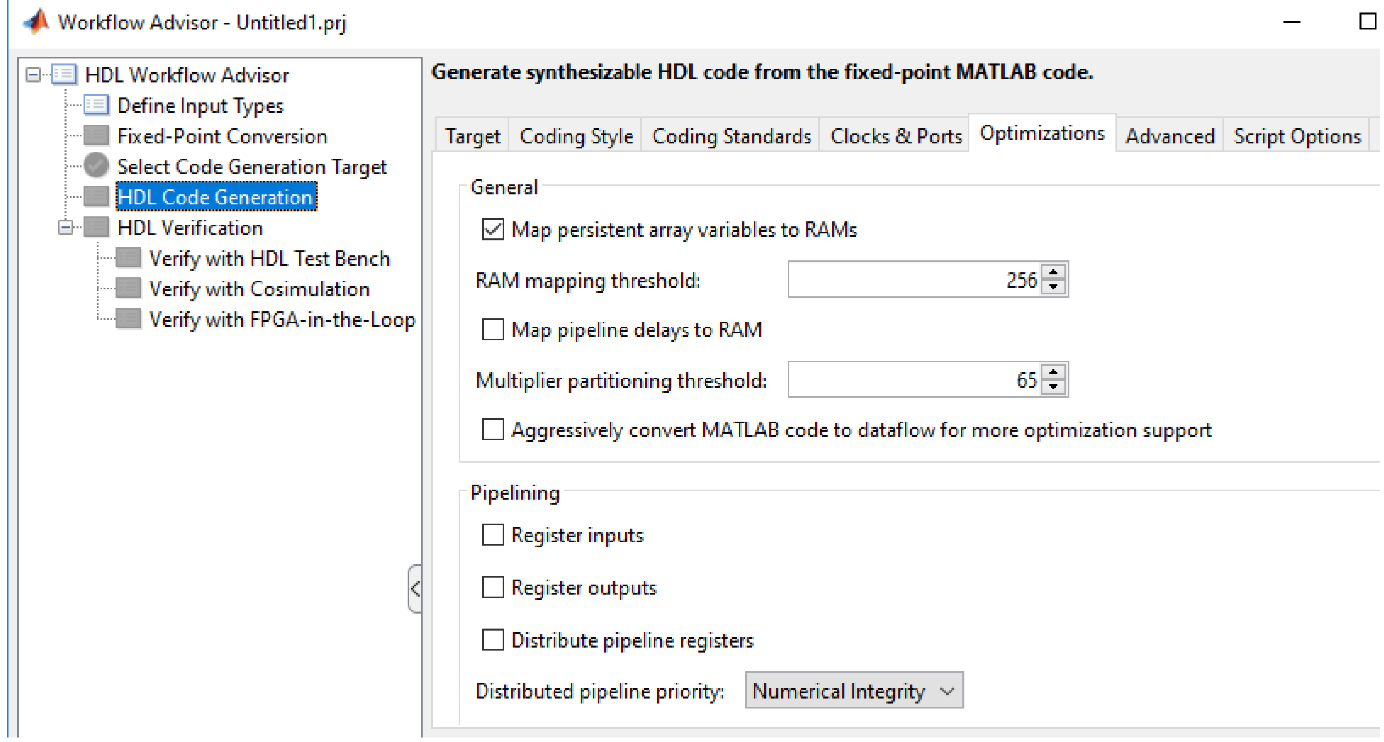
In this section...
"Enable RAM Mapping" on page 8-6
"RAM Mapping Requirements for Persistent Arrays and System object Properties" on page 8-7
"RAM Mapping Requirements for dsp.Delay System Objects" on page 8-9
"RAM Mapping Comparison" on page 8-9

Map persistent arrays and `dsp.Delay` objects to RAM on hardware to reduce the area used on your target device. For more information on general RAM mapping, see "Apply RAM Mapping to Optimize Area" on page 21-120.

Enable RAM Mapping

1. In the HDL Workflow Advisor, in the left pane, click **HDL Workflow Advisor** > **HDL Code Generation**. Then click the **Optimizations** tab.
2. Select **Map persistent array variables to RAMs**.
3. Set the **RAM mapping threshold** to either:

   - An integer that specifies the RAM size of the smallest persistent array, user-defined System object private property, or `dsp.Delay` object that you want to map to RAM.
   - A string of format `MxN` that specifies two thresholds to define the shape of the data to map to RAM, where `M` is the delay length (for delays) or array size (for persistent array variables) and `N` is the word length or bit width of the data type. Setting both thresholds excludes delays or persistent arrays from being mapped to RAM that inefficiently map to block RAM on your target hardware.



RAM Mapping Requirements for Persistent Arrays and System object Properties

The following table shows a summary of the RAM mapping behavior for persistent arrays and private properties of a user-defined System object.

Map Persistent Array Variables to RAMs Setting	Mapping Behavior
on	Map to RAM. For restrictions, see “RAM Mapping Restrictions” on page 8-7.
off	Map to registers in the generated HDL code.

RAM Mapping Restrictions

When you enable RAM mapping, a persistent array or user-defined System object private property maps to a block RAM when all of these conditions are true:

- Each read or write access is for a single element only. For example, submatrix access and array copies are not supported.
- Address computation logic is not read-dependent. For example, computation of a read or write address using the data read from the array is not supported.
- Persistent variables or user-defined System object private properties are initialized to 0 if they have a cyclic dependency. For example, if you have two persistent variables, A and B, you have a cyclic dependency if A depends on B, and B depends on A.

- If an access is within a conditional statement, the conditional statement uses only simple logic expressions (&&, `||`, ~) or relational operators. For example, in this code, `r1` does not map to RAM:

  ```
  if (mod(i,2) > 0)
      a = r1(u);
  else
      r1(i) = u;
  end
  ```

  You can rewrite complex conditions, such as conditions that call functions, by assigning them to temporary variables, and using the temporary variables in the conditional statement. For example, to map `r1` to RAM, rewrite the previous code as:

  ```
  temp = mod(i,2);
  if (temp > 0)
      a = r1(u);
  else
      r1(i) = u;
  end
  ```
- The persistent array or user-defined System object private property value depends on external inputs.

  For example, in the following code, `bigarray` does not map to RAM because it does not depend on `u`:

  ```
  function z = foo(u)

  persistent cnt bigarray
  if isempty(cnt)
      cnt = fi(0,1,16,10,hdlfimath);
      bigarray = uint8(zeros(1024,1));
  end
  z = u + cnt;
  idx = uint8(cnt);
  temp = bigarray(idx+1);
  cnt(:) = cnt + fi(1,1,16,0,hdlfimath) + temp;
  bigarray(idx+1) = idx;
  ```
- The RAM size is greater than or equal to the `RAMMappingThreshold` value. The RAM size is the product of `Array Size * Word Length * Complexity`, where:
  - `Array Size` is the number of elements in the array.
  - `Word Length` is the number of bits that represent the data type of the array.
  - `Complexity` is 2 for a complex data type or 1 for a real datatype.
- Access to the persistent variable that you are mapping to RAM is not in a loop, such as a `for` loop, unless the loop is unrolled. For more information, see `coder.unroll`.
- Access to the persistent variable that you are mapping to RAM is not in a nested conditional statement, such as a nested `if` statement or nested `switch` statement.

If any of the above conditions is false, the persistent array or user-defined System object private property maps to a register in the HDL code.

RAM Mapping Requirements for dsp.Delay System Objects

This table describes the `dsp.Delay` System object RAM mapping behavior:

Map Persistent Array Variables to RAMs Option	Mapping Behavior

on	A `dsp.Delay` System object maps to a block RAM when all of the following conditions are true:
	• `Length` property is greater than 4.
	• `InitialConditions` property is 0.
	• The delay input data type is one of these types:
	  • Real scalar with a non-floating-point data type
	  • Complex scalar with real and imaginary parts that are non-floating-point
	  • Vector where each element is either a non-floating-point real scalar or complex scalar
	• The RAM size is greater than or equal to the **RAM Mapping Threshold** value. The RAM size is the product of `DelayLength` * `WordLength`* `VectorLength` * `Complexity`, where
	  • `DelayLength` is the number of delays.
	  • `WordLength` is the number of bits that represent the input data type.
	  • `VectorLength` is the vector length of the input to the delay.
	  • `Complexity` is 2 for a complex data type or 1 for a real datatype.
	
	If any of the conditions are false, the `dsp.Delay` System object maps to registers in the HDL code.

off	A `dsp.Delay` System object maps to registers in the generated HDL code.

RAM Mapping Comparison

`hdl.RAM` objects, `dsp.Delay` objects, persistent array variables, and user-defined System object private properties can map to RAM, but have different attributes. This table summarizes the differences.

Attribute	`hdl.RAM` Objects	`dsp.Delay` Objects	Persistent Arrays and User-Defined System object Properties
RAM mapping criteria	Unconditionally maps to RAM	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for dsp.Delay System Objects” on page 8-9.	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for Persistent Arrays and System object Properties” on page 8-7.
Address generation and port mapping	User specified	Automatic	Automatic
Access scheduling	User specified	Automatically inferred	Automatically inferred
Overclocking	None	None	Local multirate, if the access schedule requires it.
Latency with respect to simulation in MATLAB.	0	0	2 cycles if local multirate; 1 cycle otherwise.
RAM type	User specified	Dual port	Dual port

See Also

Related Examples

- “Apply RAM Mapping to Optimize Area” on page 21-120
- “Map Matrices to Block RAMs to Reduce Area” on page 8-2

Pipelining MATLAB Code

Pipelining helps achieve a higher maximum clock rate by inserting registers at strategic points in the hardware to break the critical path. However, the higher clock rate comes at the expense of increased chip area and increased initial latency.

Port Registers

Input and output port registers for modules help partition a larger design so the critical path does not extend across module boundaries. Having a port register at each input and output port is a good design practice for synchronous interfaces. Distributed pipelining does not affect port registers. To insert input or output port registers:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task and select the **Optimizations** tab.
2. Enable **Register inputs**, **Register outputs**, or both.

Input and Output Pipeline Registers

You can insert multiple input and output pipeline stages. Distributed pipelining can move these input and output pipeline registers to help reduce your critical path within the module. If you insert input and output pipeline stages without applying distributed pipelining, the registers stay at the DUT inputs and outputs.

To insert input or output pipeline register stages:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task and select the **Optimizations** tab.
2. For **Input pipelining**, **Output pipelining**, or both, enter the number of pipeline register stages.

Operation Pipelining

Operation pipelining inserts one or more registers at the output of a specific expression in your MATLAB code. If you know a specific expression is part of the critical path, you can add a pipeline register at its output to reduce your critical path.

To learn how to insert a pipeline register at the output of a MATLAB expression, see "Pipeline MATLAB Expressions" on page 8-12.

Pipeline MATLAB Expressions

In this section...
"How To Pipeline a MATLAB Expression" on page 8-12
"Limitations of Pipelining for MATLAB Expressions" on page 8-12

With the `coder.hdl.pipeline` pragma, you can specify the placement and number of pipeline registers in the HDL code generated for a MATLAB expression.

If you insert pipeline registers and enable distributed pipelining, HDL Coder automatically moves the pipeline registers to break the critical path.

How To Pipeline a MATLAB Expression

To insert pipeline registers at the output of an expression in MATLAB code, place the expression in the `coder.hdl.pipeline` pragma. Specify the number of registers.

You can insert pipeline registers in the generated HDL code:

- At the output of the entire right side of an assignment statement.

  The following code inserts three pipeline registers at the output of a MATLAB expression, `a + b * c`:

  ```
  y = coder.hdl.pipeline(a + b * c, 3);
  ```
- At an intermediate stage within a longer MATLAB expression.

  The following code inserts five pipeline registers after the computation of `b * c` within a longer expression, `a + b * c`:

  ```
  y = a + coder.hdl.pipeline(b * c, 5);
  ```
- By nesting multiple instances of the pragma.

  The following code inserts five pipeline registers after the computation of `b * c`, and two pipeline registers at the output of the whole expression, `a + b * c`:

  ```
  y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c, 5),2);
  ```

Alternatively, to insert one pipeline register instead of multiple pipeline registers, you can omit the second argument in the pragma:

```
y = coder.hdl.pipeline(a + b * c);
```

```
y = a + coder.hdl.pipeline(b * c);
```

```
y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c));
```

Limitations of Pipelining for MATLAB Expressions

Note When you use the MATLAB code inside a MATLAB Function block and select the `MATLAB Datapath` architecture or enable the `AggresiveDataflowConversion` optimization, these limitations do not apply.

HDL Coder cannot insert a pipeline register at the output of a MATLAB expression if any of the variables in the expression are:

- A persistent variable that maps to a state element, like a state register or RAM.
- In a data feedback loop. For example, in the following code, you cannot pipeline an expression containing the `t` or `pvar` variables:

  ```
  persistent pvar;
  t = u + pvar;
  pvar = t + v;
  ```

See Also

`coder.hdl.pipeline`

More About

- “Pipelining MATLAB Code” on page 8-11

Distributed Pipelining

In this section...

"What is Distributed Pipelining?" on page 8-14
"Benefits and Costs of Distributed Pipelining" on page 8-14
"Selected Bibliography" on page 8-14

What is Distributed Pipelining?

Distributed pipelining, or register retiming, is a speed optimization that moves existing delays in a design to reduce the critical path while preserving functional behavior. This optimization moves the delays within a subsystem while preserving the hierarchy.

The HDL Coder software uses an adaptation of the Leiserson-Saxe retiming algorithm.

Benefits and Costs of Distributed Pipelining

Distributed pipelining can reduce your design's critical path, enabling you to use a higher clock rate and increase throughput.

However, distributed pipelining requires your design to contain a number of delays. If you need to insert additional delays in your design to enable distributed pipelining, this increases the area and the initial latency of your design.

Selected Bibliography

Leiserson, C.E, and James B. Saxe. "Retiming Synchronous Circuitry." *Algorithmica*. Vol. 6, Number 1, 1991, pp. 5-35.

Distributed Pipelining for Clock Speed Optimization

This example shows how to use the distributed pipelining and loop unrolling optimizations in HDL Coder™ to optimize clock speed.

Introduction

Distributed pipelining is a design-wide optimization supported by HDL Coder for improving clock frequency. When you turn on the 'Distribute Pipeline Registers' option in HDL Coder, the coder redistributes the input and output pipeline registers of the top level function along with other registers in the design in order to minimize the combinatorial logic between registers and thus maximize the clock speed of the chip synthesized from the generated HDL code.

Consider the following example design of a FIR filter. The combinatorial logic from an input or a register to an output or another register contains a sum of products. Loop unrolling and distributed pipelining moves the output registers at the design level to reduce the amount of combinatorial logic, thus increasing clock speed.

MATLAB® Design

The MATLAB code used in the example is a simple FIR filter. The example also shows a MATLAB test bench that exercises the filter.

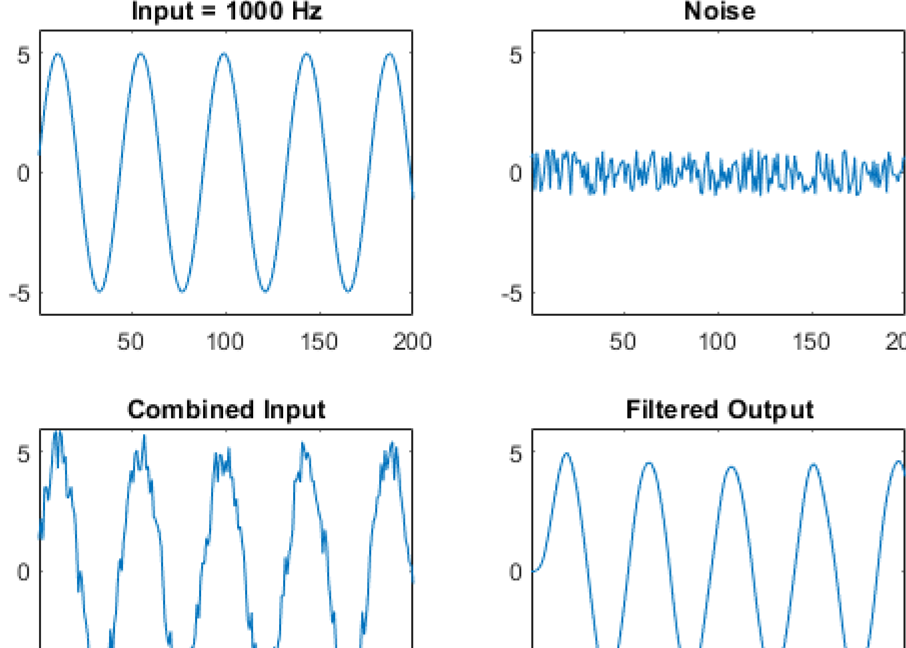
```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';
```

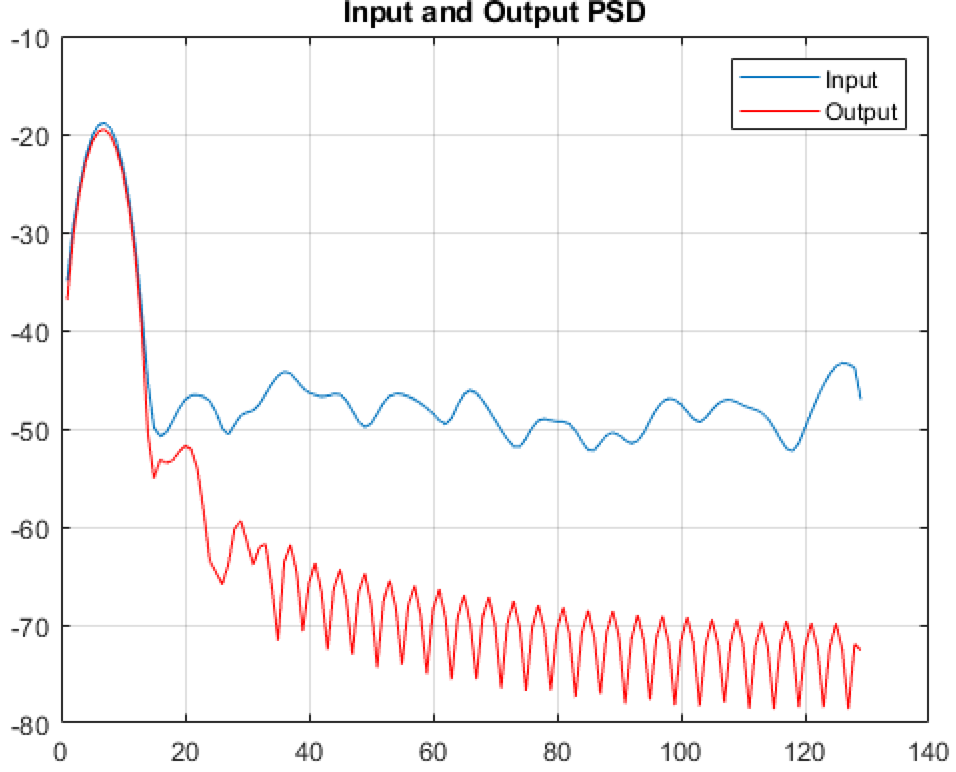
1. Design: mlhdlc_fir
2. Test Bench: mlhdlc_fir_tb

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no run-time errors.

```
mlhdlc_fir_tb
```


Input = 1000 Hz
Noise
Combined Input
Filtered Output
5
0
-5
50
100
150
200



Create a Fixed-Point Conversion Config Object

To perform fixed-point conversion, you need a 'fixpt' config object.

Create a 'fixpt' config object and specify your test bench name:

```
close all;
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_fir_tb';
```

Create an HDL Code Generation Config Object

To generate code, you must create an 'hdl' config object and set your test bench name:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_fir_tb';
```

Distributed Pipelining

To increase the clock speed, the user can set a number of input and output pipeline stages for any design. In this particular example Input pipelining option is set to '1' and Output pipelining option is set to '20'. Without any additional options turned on these settings will add one input pipeline register at all input ports of the top level design and 20 output pipeline registers at each of the output ports.

If the option 'Distribute pipeline registers' is enabled, HDL Coder tries to reposition the registers to achieve the best clock frequency.

In addition to moving the input and output pipeline registers, HDL Coder also tries to move the registers modeled internally in the design using persistent variables or with system objects like dsp.Delay.

Additional opportunities for improvements become available if you unroll loops. The 'Unroll Loops' option unrolls explicit for-loops in MATLAB code in addition to implicit for-loops that are inferred for vector and matrix operations. 'Unroll Loops' is necessary for this example to do distributed pipelining.

```
hdlcfg.InputPipeline = 1;
hdlcfg.OutputPipeline = 20;
hdlcfg.DistributedPipelining = true;
hdlcfg.LoopOptimization = 'UnrollLoops';
```

Examine the Synthesis Results

If you have ISE installed on your machine, run the logic synthesis step

```
hdlcfg.SynthesizeGeneratedCode = true;
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_fir
```

View the result report

```
edit codegen/mlhdlc_fir/hdlsrc/ise_prj/mlhdlc_fir_fixpt_syn_results.txt
```

In the synthesis report, note the clock frequency reported by the synthesis tool. When you synthesize the design with the loop unrolling and distributed pipelining options enabled, you see a significant clock frequency increase with pipelining options turned on.

Optimize Clock Speed for MATLAB Code by Using Adaptive Pipelining

This example shows how to use the adaptive pipelining optimization in HDL Coder™ to optimize clock speed for a MATLAB® design.

Introduction

Certain patterns of code that have registers can improve the achievable clock frequency and reduce the area usage on FPGA boards. The adaptive pipelining optimization creates these patterns by inserting pipeline registers in your design. To determine the optimal number of pipeline registers to insert in your design, the target device, target frequency, and multiplier word lengths are considered.

You can also use adaptive pipelining for:

- Automatically pipelining multiply operations for optimized Digital Signal Processors (DSP) mapping.
- Other optimizations, such as resource sharing, which saves area and timing because HDL Coder shares resources and inserts adaptive pipeline registers.

To insert adaptive pipelining into your design:

- Specify the target device by specifying a synthesis tool.
- Set your target frequency to be greater than zero.

Enable Adaptive Pipelining in a MATLAB Design

- Set adaptive pipelining by using an HDL code configuration object. For example, this MATLAB code creates an HDL code configuration object and sets the adaptive pipelining property to `true`:

```
hdlcfg = coder.config('hdl');
hdlcfg.AdaptivePipelining = true;
```

For more information, see `coder.HdlConfig`.

- Enable adaptive pipelining in the **HDL Workflow Advisor** > **HDL Code Generation** task > **Optimization** tab.

Adaptive Pipelining Example

Consider the following example design of a simple symmetric FIR filter. Adaptive pipelining inserts pipelines to reduce the amount of combinatorial logic and increase the clock speed. The example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_sfir';
testbench_name = 'mlhdlc_sfir_tb';
```

Review the MATLAB design.

```
open(design_name);
```

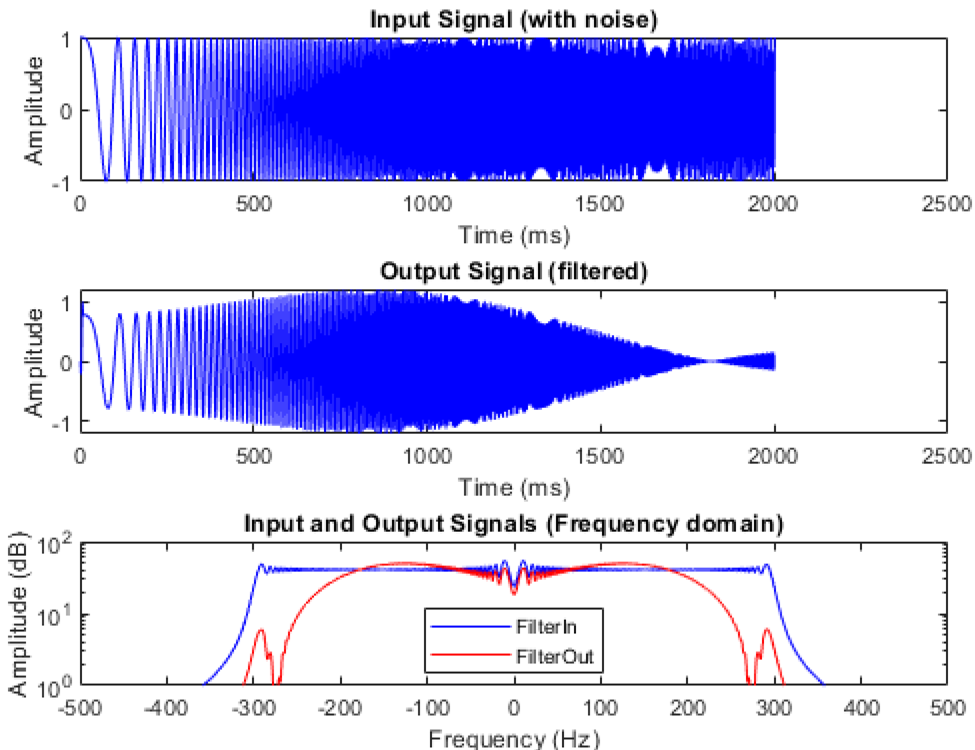
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
```

```
%
% Introduction:
%
% We can reduce the complexity of the FIR filter by leveraging its symmetry.
% Symmetry for an n-tap filter implies, coefficient h0 = coefficient hn-1,
% coefficient, h1 = coefficient hn-2, etc. In this case, the number of
% multipliers can be approximately halved. The key is to add the
% two data values that need to be multiplied with the same coefficient
% prior to performing the multiplication.
%
% Key Design pattern covered in this example:
% (1) Filter states represented using the persistent variables
% (2) Filter coefficients passed in as parameters

%   Copyright 2011-2019 The MathWorks, Inc.

%#codegen
function [y_out, delayed_xout] = mlhdlc_sfir(x_in,h_in1,h_in2,h_in3,h_in4)
% Symmetric FIR Filter

% declare and initialize the delay registers
persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
if isempty(ud1)
    ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
end

% access the previous value of states/registers
a1 = ud1 + ud8; a2 = ud2 + ud7;
a3 = ud3 + ud6; a4 = ud4 + ud5;

% multiplier chain
m1 = h_in1 * a1; m2 = h_in2 * a2;
m3 = h_in3 * a3; m4 = h_in4 * a4;

% adder chain
a5 = m1 + m2; a6 = m3 + m4;

% filtered output
y_out = a5 + a6;

% delayout input signal
delayed_xout = ud8;

% update the delay line
ud8 = ud7;
ud7 = ud6;
ud6 = ud5;
ud5 = ud4;
ud4 = ud3;
ud3 = ud2;
ud2 = ud1;
ud1 = x_in;
end
```

Simulate the Design

Before you generate code, simulate the design by using the test bench to make sure there are no run-time errors.

```
mlhdlc_sfir_tb
```



Set Up Synthesis Tool Path

Before you use HDL Coder to generate code, set up your synthesis tool path to synthesize the generated HDL code. Use the `hdlsetuptoolpath` function. For example, if your synthesis tool is Xilinx® Vivado®:

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
              'C:\Xilinx\Vivado\2018.3\bin\vivado.bat');
```

You must have already installed Xilinx Vivado. To check your Xilinx Vivado synthesis tool setup, start the tool by running this command:

```
!vivado
```

Create an HDL Coder Project

1. Create a project by running this command:

```
coder -hdlcoder -new fir_project
```

2. For **MATLAB Function**, add the MATLAB design `mlhdlc_sfir`. Add `mlhdlc_sfir_tb.m` as the MATLAB test bench.

3. Click **Autodefine types** and use the recommended types for the MATLAB design. HDL Coder infers data types by running the test bench.

Create Fixed-Point Versions of Algorithm and Test Bench

1. Click the **Workflow Advisor** button to open the Workflow Advisor. The **Define Input Types** task has passed.
2. Run the **Fixed-Point Conversion** task. The **Fixed-Point Conversion** tool opens in the right pane. For more information, see "Floating-Point to Fixed-Point Conversion" on page 4-51.

Select Code Generation Options

Before you generate HDL code, to deploy the code onto a target platform and use adaptive pipelining for your design, specify a synthesis tool. In the **Code Generation Target** task, leave **Workflow** set to `Generic ASIC/FPGA` and specify `Xilinx Vivado` as the **Synthesis Tool**. If you do not see the synthesis tool, click **Refresh list**. Specifying the **Synthesis tool** sets default values for the target hardware, such as **Chip family**, **Device**, **Package**, and **Speed**. Set **Speed** to `-1`. Leave the rest of the target hardware properties at their defaults.

To show the effects of adaptive pipelining, specify a target frequency that you are trying to achieve for your target device. For this example, set the **Target frequency (MHz)** to `200`. Run this task.

In the **HDL Code Generation** task, by using the tabs on the right side of this task, you can specify additional code generation options. For this example, in the **HDL Code Generation** task **>** **Optimization** tab:

- Specify `1` for **Input pipelining** and **Output pipelining**.
- Set **Loop Optimizations** to **Loop Unroll**. It is good practice to enable this option whenever you are using adaptive pipelining.
- For before and after comparison of adaptive pipelining effects on your design, keep the **Adaptive Pipelining** check box clear.
- Click **Run** to generate HDL code.

Examine the log window and click the links to explore the generated code and the reports. In the log window, the report shows two cycles of latency, one cycle each for the input pipeline and output pipeline added.

Synthesize Generated Code for Nonoptimized Design

After generating HDL code, in this example, go to **Synthesis and Analysis** and clear **skip this task** so that synthesis and implementation can be run on your design by using your specified **Synthesis Tool**, Xilinx Vivado. HDL Coder synthesizes the HDL code on the target platform and generates area and timing reports for your design based on the target device that you specify.

To synthesize the generated HDL code and generate a timing report from synthesis:

1. Go to the **Create project** task and add the `clock_constraint.xdc` file from the example folder by either:

- Providing the path to the file manually in the **Additional Project Files** option.

- By selecting the file using the ellipsis on the right-side of the **Additional Project Files** option. Click the right side of the drop-down list for another ellipsis that opens your file explorer window. Navigate to the folder containing `clock_constraint.xdc`, double-click the file, and click **OK**.

If you adjust your target frequency, adjust the clock constraint file accordingly. For example, because the target frequency is 200 MHz, and the clock constraint file requires the period in nanoseconds, the period is set to 1/200 MHz = 5 ns.

2. Run the **Create project** task.

This task creates a Xilinx Vivado synthesis project for the HDL code. HDL Coder uses this project in the next task to synthesize the design.

3. Select and run the **Run Synthesis** task.

This task starts the synthesis tool in the background, opens the synthesis project, compiles the HDL code, synthesizes the design, and generates netlists and area and timing reports. The timing reports take into account the `clock_constraint.xdc` file you previously specified to calculate slack between your required path delay and your actual path delay.

3. Select and run the **Run Implementation** task.

This task starts the synthesis tool in the background, runs place and route on the design, and generates pre- and post-route timing information for use in critical path analysis and back annotation of your source model.

The timing report in the output window of the **Run Implementation** task has the synthesis results show a negative slack of -2.47 ns, indicating that timing constraints are not met, and the clock frequency is 133 MHz, below the target frequency 200 MHz.

This figure shows how the algorithm is implemented in hardware when running synthesis on the generated code without adaptive pipelining. The input and output pipelines are shown in orange.

Symmetric FIR Filter

delayed_xout
x_in
h_in1
h_in2
h_in3
h_in4
y_out

Generate and Synthesize Code for Optimized Design with Adaptive Pipelining

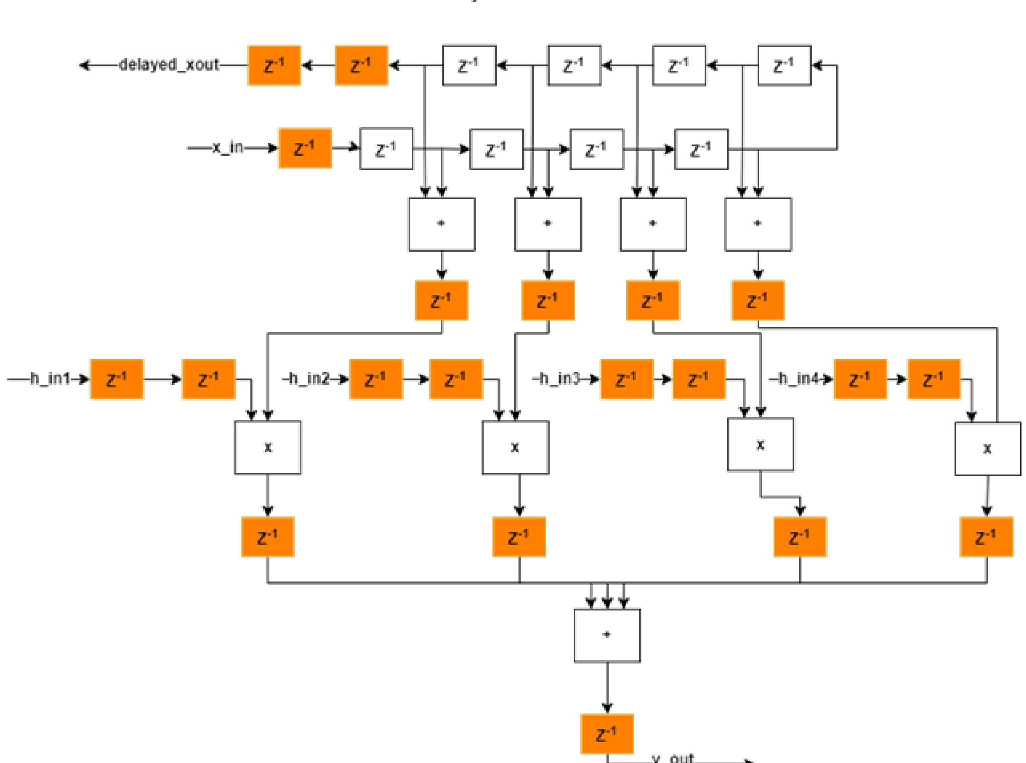
To generate HDL code with adaptive pipelining enabled, go to the **HDL Code Generation** task > **Optimization** tab, enable **Adaptive Pipelining**, and click **Run**.

In the log window, the report now shows four cycles of latency per output port, indicating that adaptive pipelining adds extra delays in the design.

When you run this task again, the subsequent tasks are reset. Because you have already specified your clock constraint file, right-click the **Synthesis and Analysis** task and select **Run This task** to run through the entire project creation, synthesis, and implementation workflow.

The timing report in the output window of the **Run Implementation** task has the synthesis results show a positive slack of .874 ns, indicating that timing constraints are now met as a result of enabling adaptive pipelining. The clock frequency is 242 MHz, above the target frequency 200 MHz.

This figure shows how the algorithm is implemented in hardware when running synthesis on the generated code with adaptive pipelining on. The additional delays added from adaptive pipelining and delay balancing are shown in orange, along with the original input and output pipelines.



Limitations

While adaptive pipelining can improve clock frequency and reduce area usage for your design, multiply operations might not be pipelined when adaptive pipelining is enabled if:

- The multiply operation is in a `for` loop that is not unrolled.
- The multiply operation is in a subfunction and **Generate instantiable code for functions** is not enabled. **Generate instantiable code for functions** is located in the **HDL Code Generation** task > **Advanced** tab.
- The multiply operation is in a subfunction and the function contains `coder.inline('never')`.

See Also

"Pipelining MATLAB Code" on page 8-11 | "Pipeline MATLAB Expressions" on page 8-12 | "Optimize MATLAB Loops" on page 8-29

Related Examples

- "Basic HDL Code Generation and FPGA Synthesis from MATLAB"
- "Distributed Pipelining for Clock Speed Optimization" on page 8-15

Optimize Feedback Loop Design and Maintain High Data Precision for HDL Code Generation

Optimize a feedback loop design and maintain high data precision for HDL code generation by using native floating point and clock-rate pipelining when generating HDL code from a MATLAB® design.

Open the MATLAB Code

The MATLAB function `feedback_fcn` describes a feedback loop that keeps track of the previous state value and adds it to the current value.

```
open('feedback_fcn')
```

```
function y = feedback_fcn(u)
persistent state;
if isempty(state)
    state = 0;
end
y = u + state + 1;
state = y;
```

Create HDL Code Generation Configuration Object and Apply Clock-Rate Pipelining

The variable `state` is a persistent variable. In HDL code generation, persistent variables act as delays and map to a register. As a result, persistent variables introduce latency in the feedback loop. To compensate for this latency in the generated HDL code, you can use clock-rate pipelining. For more information on clock-rate pipelining, see “Clock-Rate Pipelining” on page 21-148.

To programmatically apply clock-rate pipelining, first create a `coder.HDLConfig` object.

```
hdlcfg = coder.config('hdl');
```

Set the `DesignFunctionName` property to the MATLAB function and the `TestBenchName` property to the test bench script, `feedback_fcn_tb`.

```
hdlcfg.DesignFunctionName = 'feedback_fcn';
hdlcfg.TestBenchName = 'feedback_fcn_tb';
```

Enable `GenerateHDLTestBench` to generate HDL test bench code from `feedback_fcn_tb`.

```
hdlcfg.GenerateHDLTestBench = true;
```

Enable clock-rate pipelining and set an oversampling factor based on the amount of latency introduced in the design. In this example, the design requires a 22x faster clock than the base rate. Set the oversampling factor to 22.

```
hdlcfg.ClockRatePipelining = true;
hdlcfg.Oversampling = 22;
```

Use Native Floating Point for HDL Code Generation

The test bench function `feedback_fcn_tb` uses the `double` data type as the input to `feedback_fcn`.

```
open('feedback_fcn_tb')
```

```
for ii = 1:10
    y = feedback_fcn(ii);
end
```

Using a `double` data type as the input for HDL code generation produces non-synthesizable code unless you either convert the input to fixed-point or use native floating point. To maintain high data precision, use native floating point. The trade-off is that more hardware resources are needed to store floating-point values than fixed-point values.

To enable native floating point, first enable the `AggressiveDataflowConversion` property. This property transforms the control flow algorithm of the MATLAB code inside the MATLAB function to a dataflow representation, which native floating point uses.

```
hdlcfg.AggressiveDataflowConversion = true;
```

Use native floating point for HDL code generation by setting the `FloatingPointLibrary` property to `NativeFloatingPoint`. To produce an error during HDL code generation if there is any real type in the HDL code, you can set the diagnostic option `TreatRealsInGeneratedCodeAs` to `Error`. By default, the option `TreatRealsInGeneratedCodeAs` is set to `Error`.

```
hdlcfg.FloatingPointLibrary = 'NativeFloatingPoint';
hdlcfg.TreatRealsInGeneratedCodeAs
```

```
ans =

    'Error'
```

Generate a Simulink® model that contains Simulink blocks that is functionally equivalent to your MATLAB function design. The Simulink model performs the algorithm designed in your MATLAB function. This property requires a Simulink license.

```
hdlcfg.GenerateMLFcnBlock = true;
```

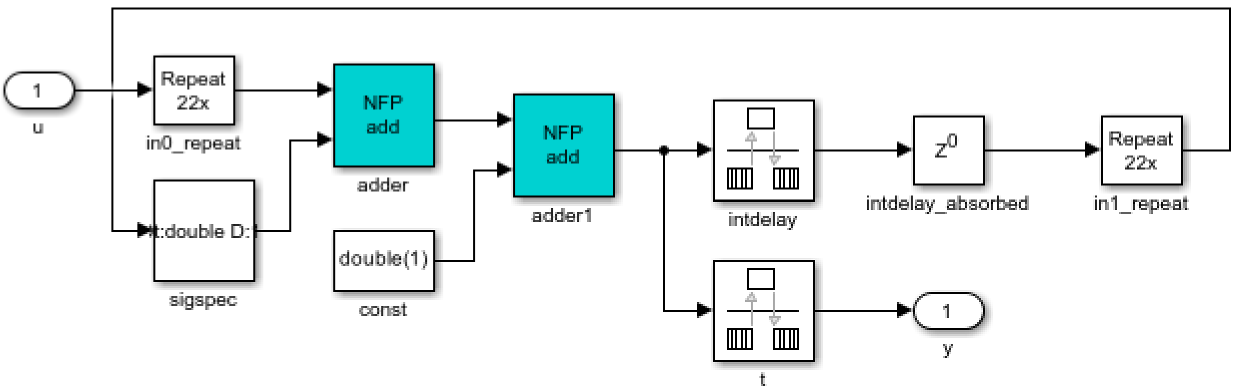
Generate HDL Code

Generate HDL code with an HDL code generation report by using the `codegen` function.

```
codegen -report -config hdlcfg
```

```
### Generating new model: '<a href="matlab:open_system('gm_feedback_fcn')">gm_feedback_fcn</a>'.
### Begin model generation 'gm_feedback_fcn'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### MESSAGE: The design requires 22 times faster clock with respect to the base rate = 1.
### Working on feedback_fcn_tc as C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\29\tp9de71946\hdlcoder-ex
### Begin VHDL Code Generation
### Working on feedback_fcn/nfp_add_double as <a href="matlab:edit('C:\TEMP\Bdoc24a_2528353_7604\
### Working on feedback_fcn_enb_bypass as <a href="matlab:edit('C:\TEMP\Bdoc24a_2528353_7604\ib46
### Working on feedback_fcn as <a href="matlab:edit('C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\29\tp9
### Generating package file <a href="matlab:edit('C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\29\tp9de
```

```
### Generating Resource Utilization Report <a href="matlab:hdlcoder.report.openDdg('C:\TEMP\Bdoc2

### Begin TestBench generation.
Code generation successful.

### Accounting for latency of output port : 1 cycles.
### Collecting data...
### Begin HDL test bench file generation with logged samples
### Generating test bench data file: C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\29\tp9de71946\hdlcode
### Generating test bench data file: C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\29\tp9de71946\hdlcode
### Working on feedback_fcn_tb as C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\29\tp9de71946\hdlcoder-e
### Generating package file C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\29\tp9de71946\hdlcoder-ex54382
### Generating HDL Conformance Report <a href="matlab:web('C:\TEMP\Bdoc24a_2528353_7604\ib462BFE
### HDL Conformance check complete with 0 errors, 0 warnings, and 1 messages.
### Code generation successful: To view the report, open('codegen\feedback_fcn\hdlsrc\html\repor
```



See Also

Related Examples

- “Basic HDL Code Generation and FPGA Synthesis from MATLAB”
- “Optimize MATLAB Loops” on page 8-29
- “Clock-Rate Pipelining” on page 21-148
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111

Optimize MATLAB Loops

In this section...

"Loop Streaming" on page 8-29
"Loop Unrolling" on page 8-29
"How to Optimize MATLAB Loops" on page 8-29
"Limitations for MATLAB Loop Optimization" on page 8-30

With loop optimization, you can stream or unroll loops in generated code. Loop streaming is an area optimization, and loop unrolling is a speed optimization. To optimize loops for MATLAB code that is inside a MATLAB Function block, use the `MATLAB Function` architecture. When you use the `MATLAB Datapath` architecture, the code generator unrolls loops irrespective of the loop optimization setting.

Loop Streaming

HDL Coder streams a loop by instantiating the loop body once and using that instance for each loop iteration. The code generator oversamples the loop body instance to keep the generated loop functionally equivalent to the original loop.

If you stream a loop, the advantage is decreased hardware resource usage because the loop body is instantiated fewer times. The disadvantage is the hardware implementation runs at a lower speed.

You can partially stream a loop. A partially streamed loop instantiates the loop body more than once, so it uses more area than a fully streamed loop. However, a partially streamed loop also uses less oversampling than a fully streamed loop.

Loop Unrolling

HDL Coder unrolls a loop by instantiating multiple instances of the loop body in the generated code. You can also partially unroll a loop. The generated code uses a loop statement that contains multiple instances of the original loop body and fewer iterations than the original loop.

The distributed pipelining and resource sharing can optimize the unrolled code. Distributed pipelining can increase speed. Resource sharing can decrease area.

When loop unrolling creates multiple instances, these instances are likely to increase area. Loop unrolling also makes the code harder to read.

How to Optimize MATLAB Loops

You can specify a global loop optimization by using the HDL Workflow Advisor, or at the command line.

You can also specify a local loop optimization for a specific loop by using the `coder.hdl.loopspec` pragma in the MATLAB code. If you specify both a global and local loop optimization, the local loop optimization overrides the global setting.

Global Loop Optimization

To specify a loop optimization in the Workflow Advisor:

1. In the HDL Workflow Advisor left pane, select **HDL Workflow Advisor > HDL Code Generation**.
2. In the **Optimizations** tab, for **Loop Optimizations**, select **None**, **Unroll Loops**, or **Stream Loops**.

To specify a loop optimization at the command line in the MATLAB to HDL workflow, specify the `LoopOptimization` property of the `coder.HdlConfig` object. For example, for a `coder.HdlConfig` object, `hdlcfg`, enter one of the following commands:

```
hdlcfg.LoopOptimization = 'UnrollLoops'; % unroll loops
```

```
hdlcfg.LoopOptimization = 'StreamLoops'; % stream loops
```

```
hdlcfg.LoopOptimization = 'LoopNone'; % no loop optimization
```

Local Loop Optimization

To learn how to optimize a specific MATLAB loop, see `coder.hdl.loopspec`.

Note If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

Limitations for MATLAB Loop Optimization

HDL Coder cannot stream a loop if:

- The loop index counts down. The loop index must increase by 1 on each iteration.
- There are two or more nested loops at the same level of hierarchy within another loop.
- Any particular persistent variable is updated both inside and outside a loop.
- A persistent variable that is initialized to a nonzero value is updated inside the loop.

HDL Coder can stream a loop when the persistent variable is:

- Updated inside the loop and read outside the loop.
- Read within the loop and updated outside the loop.

You cannot use the `coder.hdl.loopspec('stream')` pragma:

- In a subfunction. You must specify it in the top-level MATLAB design function.
- For a loop that is nested within another loop.
- For a loop containing a nested loop, unless the streaming factor is equal to the number of iterations.

See Also

`coder.hdl.loopspec`

Constant Multiplier Optimization

In this section...
“What is Constant Multiplier Optimization?” on page 8-31
“Specify Constant Multiplier Optimization” on page 8-31

What is Constant Multiplier Optimization?

The **Constant multiplier optimization** option enables you to specify use of canonical signed digit (CSD) or factored CSD (FCSD) optimizations for processing coefficient multiplier operations.

The following table shows the **Constant multiplier optimization** values.

Constant Multiplier Optimization Value	Description
None (default)	By default, HDL Coder does not perform CSD or FCSD optimizations. Code generated for the Gain block retains multiplier operations.
CSD	When you specify this option, the generated code decreases the area used by the model while maintaining or increasing clock speed, using canonical signed digit (CSD) techniques. CSD replaces multiplier operations with add and subtract operations. CSD minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.
FCSD	This option uses factored CSD (FCSD) techniques, which replace multiplier operations with shift and add/subtract operations on certain factors of the operands. These factors are generally prime but can also be a number close to a power of 2, which favors area reduction. This option lets you achieve a greater area reduction than CSD, at the cost of decreasing clock speed.
Auto	When you specify this option, HDL Coder chooses between the CSD or FCSD optimizations. The coder chooses the optimization that yields the most area-efficient implementation, based on the number of adders required. HDL Coder does not use multipliers, unless conditions are such that CSD or FCSD optimizations are not possible (for example, if the design uses floating-point arithmetic).

Specify Constant Multiplier Optimization

To specify constant multiplier optimization:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task and select the **Optimizations** tab.

2 For **Constant multiplier optimization**, select **CSD**, **FCSD**, or **Auto**.

Resource Sharing of Multipliers to Reduce Area

This example shows how to use the resource sharing optimization in HDL Coder™. This optimization identifies functionally equivalent multiplier operations in MATLAB® code and shares them in order to optimize design area. You have control over the number of multipliers to be shared in the design.

Introduction

Resource sharing is a design-wide optimization supported by HDL Coder™ for implementing area-efficient hardware.

This optimization enables users to share hardware resources by mapping `N` functionally-equivalent MATLAB operators, in this case multipliers, to a single operator.

The user specifies `N` using the `Resource Sharing Factor` option in the optimization panel.

Consider the following example model of a symmetric FIR filter. It contains 4 product blocks that are functionally equivalent and which are mapped to 4 multipliers in hardware. The Resource Utilization Report shows the number of multipliers inferred from the design.

In this example you run fixed-point conversion on the MATLAB design `mlhdlc_sharing` followed by HDL Coder. This prerequisite step normalizes all the multipliers used in the fixed-point code. You input proposed-type settings during this fixed-point conversion phase.

MATLAB Design

The MATLAB code used in the example is a simple symmetric FIR filter written in MATLAB and also has a testbench that exercises the filter.

```
design_name = 'mlhdlc_sharing';
testbench_name = 'mlhdlc_sharing_tb';
```

Look at the MATLAB design.

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
%
% Key Design pattern covered in this example:
% (1) Filter states represented using the persistent variables
% (2) Filter coefficients passed in as parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [y_out, x_out] = mlhdlc_sharing(x_in, h)
% Symmetric FIR Filter

persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
if isempty(ud1)
    ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
end
```

```
x_out = ud8;


a1 = ud1 + ud8;
a2 = ud2 + ud7;
a3 = ud3 + ud6;
a4 = ud4 + ud5;

% filtered output
y_out = (h(1) * a1 + h(2) * a2) + (h(3) * a3 + h(4) * a4);


% update the delay line
ud8 = ud7;
ud7 = ud6;
ud6 = ud5;
ud5 = ud4;
ud4 = ud3;
ud3 = ud2;
ud2 = ud1;
ud1 = x_in;

end
```

```
type(testbench_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB test bench for the FIR filter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2015 The MathWorks, Inc.

clear mlhdlc_sharing;

% input signal with noise
x_in = cos(3.*pi.*(0:0.001:2).*(1+(0:0.001:2).*75)).';

len = length(x_in);
y_out = zeros(1,len);
x_out = zeros(1,len);

% Define a regular MATLAB constant array:
%
% filter coefficients
h = [-0.1339 -0.0838 0.2026 0.4064];

for ii=1:len
    data = x_in(ii);
    % call to the design 'mlhdlc_sfir' that is targeted for hardware
    [y_out(ii), x_out(ii)] = mlhdlc_sharing(data, h);
end

figure('Name', [mfilename, '_plot']);
plot(1:len,y_out);
```

Create a New HDL Coder Project

Run the following command to create a new project:

```
coder -hdlcoder -new mlhdlc_sfir_sharing
```

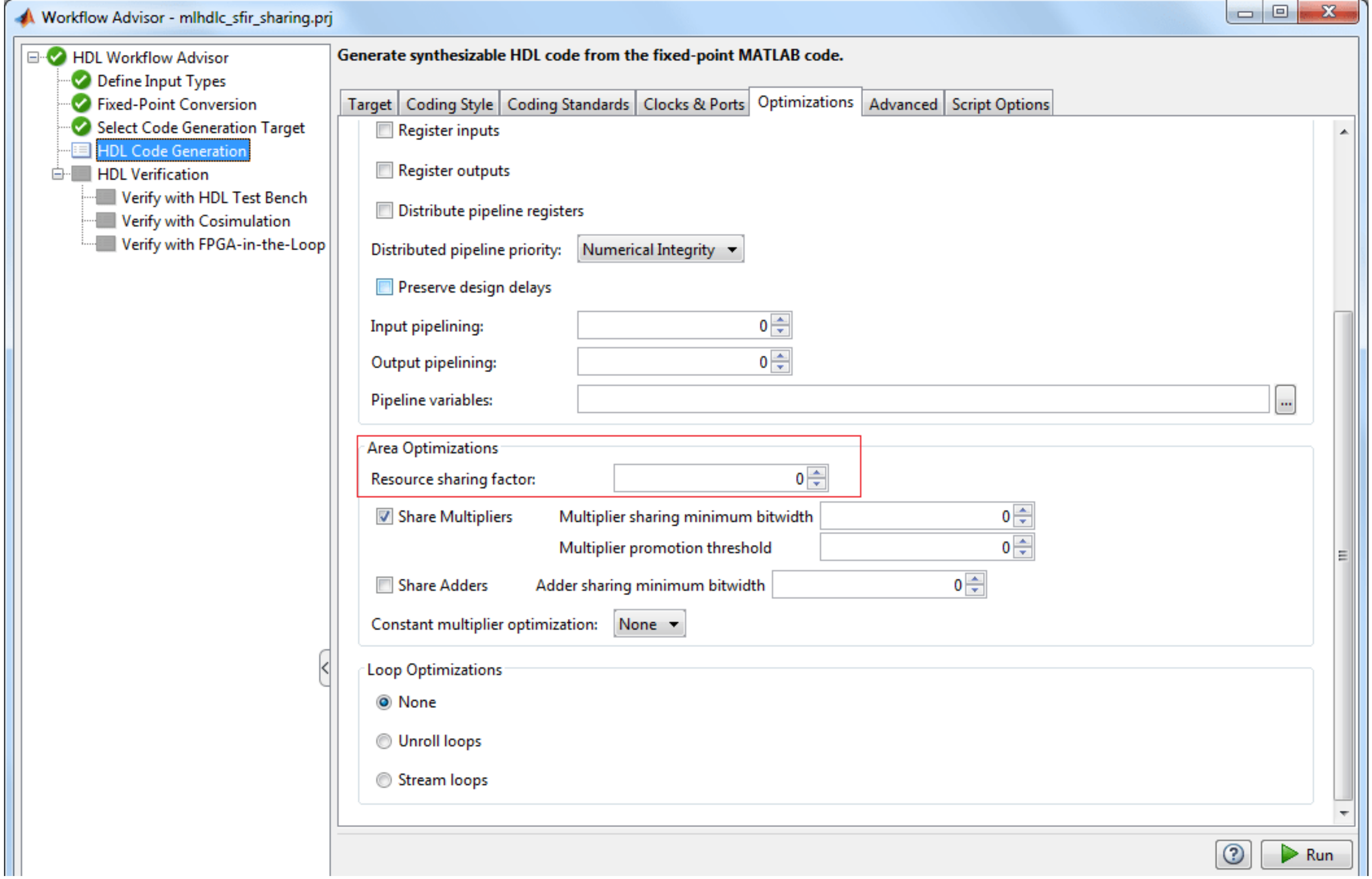
Next, add the file `mlhdlc_sharing.m` to the project as the MATLAB Function and `mlhdlc_sharing_tb.m` as the MATLAB Test Bench.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Realize an N-to-1 Mapping of Multipliers

Turn on the resource sharing optimization by setting the 'Resource Sharing Factor' to a positive integer value.

This parameter specifies `N` in the N-to-1 hardware mapping. Choose a value of N > 1.



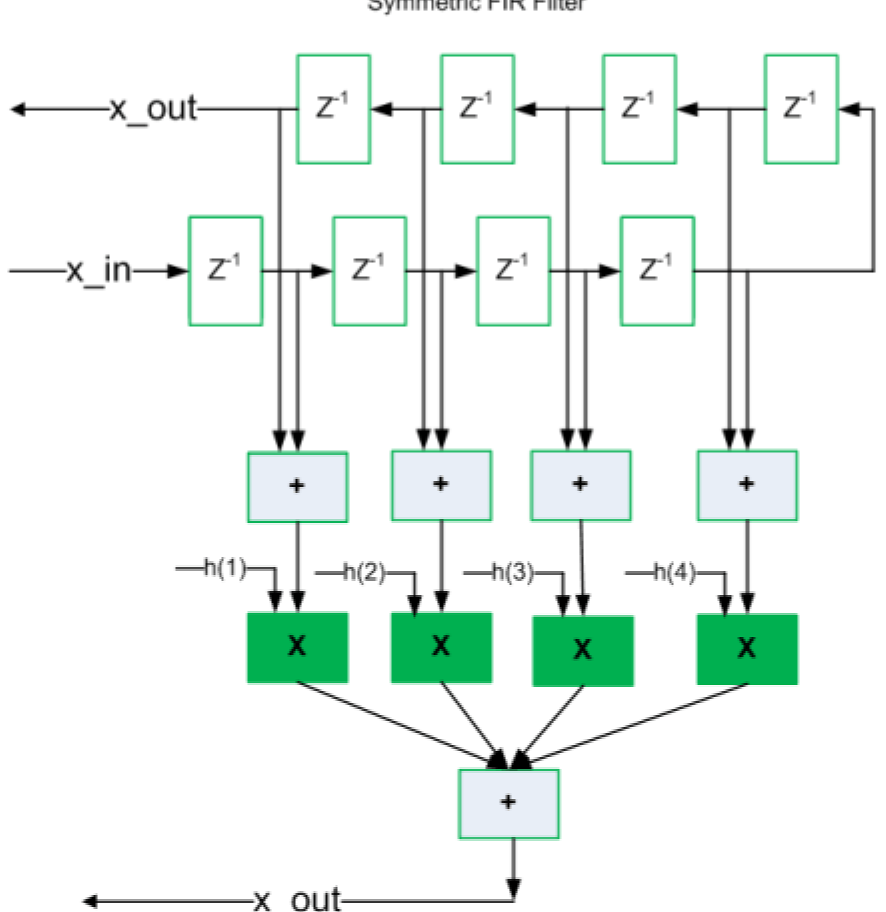
Examine the Resource Report

There are 4 multiplication operators in this example design. Generating HDL with a `SharingFactor` of 4 will result in only one multiplier in the generated code.

Multipliers	1
Adders/Subtractors	7
Registers	29
RAMs	0
Multiplexers	12

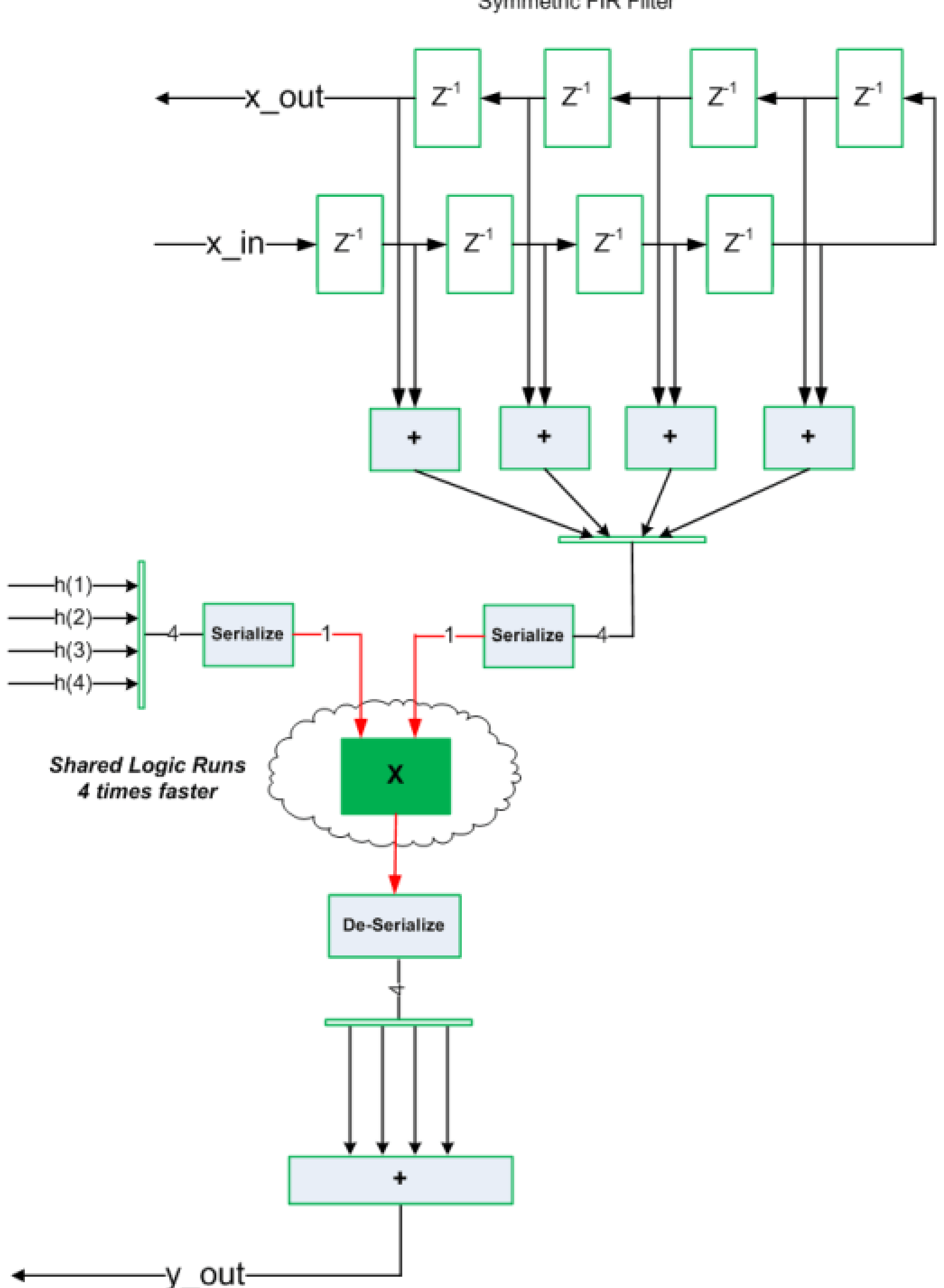
Sharing Architecture

The following figure shows how the algorithm is implemented in hardware when we synthesize the generated code without turning on the sharing optimization.



The following figure shows the sharing architecture automatically implemented by HDL Coder when the sharing optimization option is turned on.

The inputs to the shared multiplier are time-multiplexed at a faster rate (in this case 4x faster and denoted in red). The outputs are then routed to the respective consumers at a slower rate (in green).



Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor and right-click the **Code Generation** step. Choose the option **Run to selected task** to run all the steps from the beginning through the HDL code generation.

The detailed example “Fixed-Point Type Conversion and Derived Ranges” on page 4-73 provides a tutorial for updating the type proposal settings during fixed-point conversion.

Note that to share multipliers of different word-length, in the Optimization -> Resource Sharing tab of HDL Configuration Parameters, specify the **Multiplier promotion threshold**.

Run Synthesis and Examine Synthesis Results

Synthesize the generated code from the design with this optimization turned off, then with it turned on, and examine the area numbers in the resource report.

Loop Streaming to Reduce Area

This example shows how to use the design-level loop streaming optimization in HDL Coder™ to optimize area.

Introduction

A MATLAB® for loop generates a FOR_GENERATE loop in VHDL. Such loops are always spatially unrolled for execution in hardware. In other words, the body of the software loop is replicated as many times in hardware as the number of loop iterations. This results in inefficient area usage.

The loop streaming optimization creates an alternative implementation of a software loop, where the body of the loop is shared in hardware. Instead of spatially replicating copies of the loop body, HDL Coder™ creates a single hardware instance of the loop body that is time-multiplexed across loop iterations.

MATLAB Design

The MATLAB code used in this example implements a simple FIR filter. This example also shows a MATLAB testbench that exercises the filter.

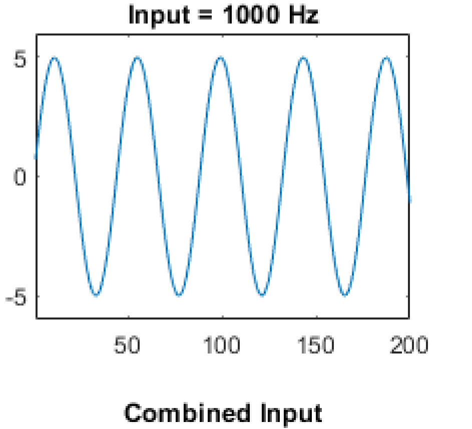
```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';
```

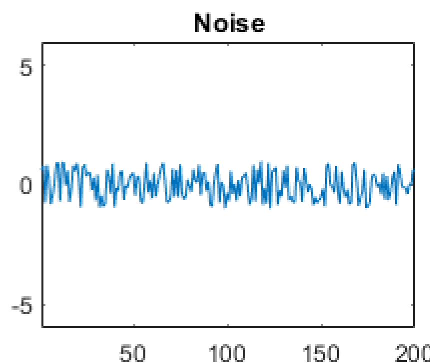
1. Design: mlhdlc_fir
2. Test Bench: mlhdlc_fir_tb

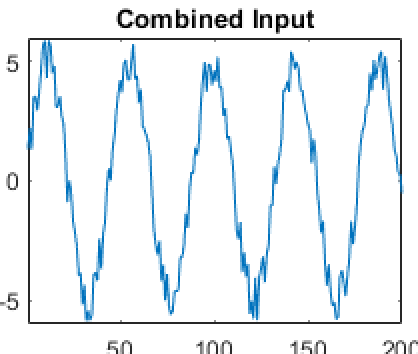
Simulate the Design

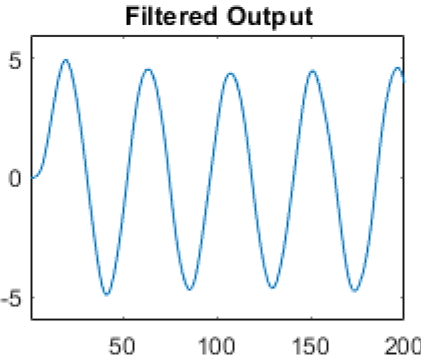
Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

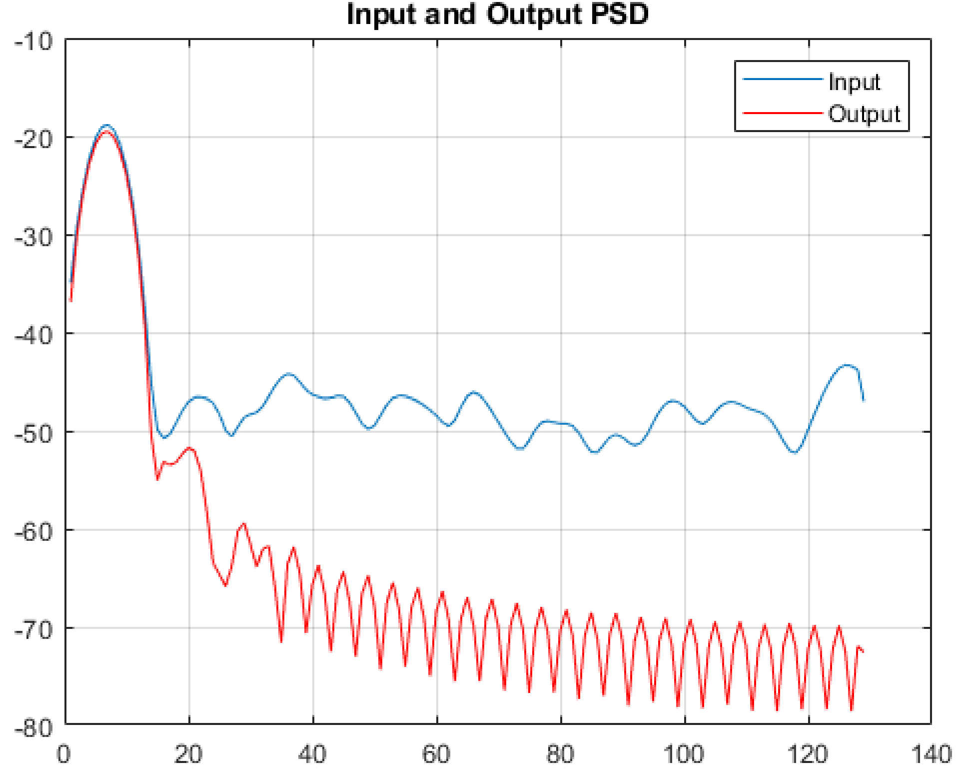
```
mlhdlc_fir_tb
```


Input = 1000 Hz
5
0
-5
50
100
150
200


Noise
5
0
-5
50
100
150
200


Combined Input
5
0
-5
50
100
150
200


Filtered Output
5
0
-5
50
100
150
200



Creating a New Project From the Command Line

To create a new project, enter the following command:

```
coder -hdlcoder -new fir_project
```

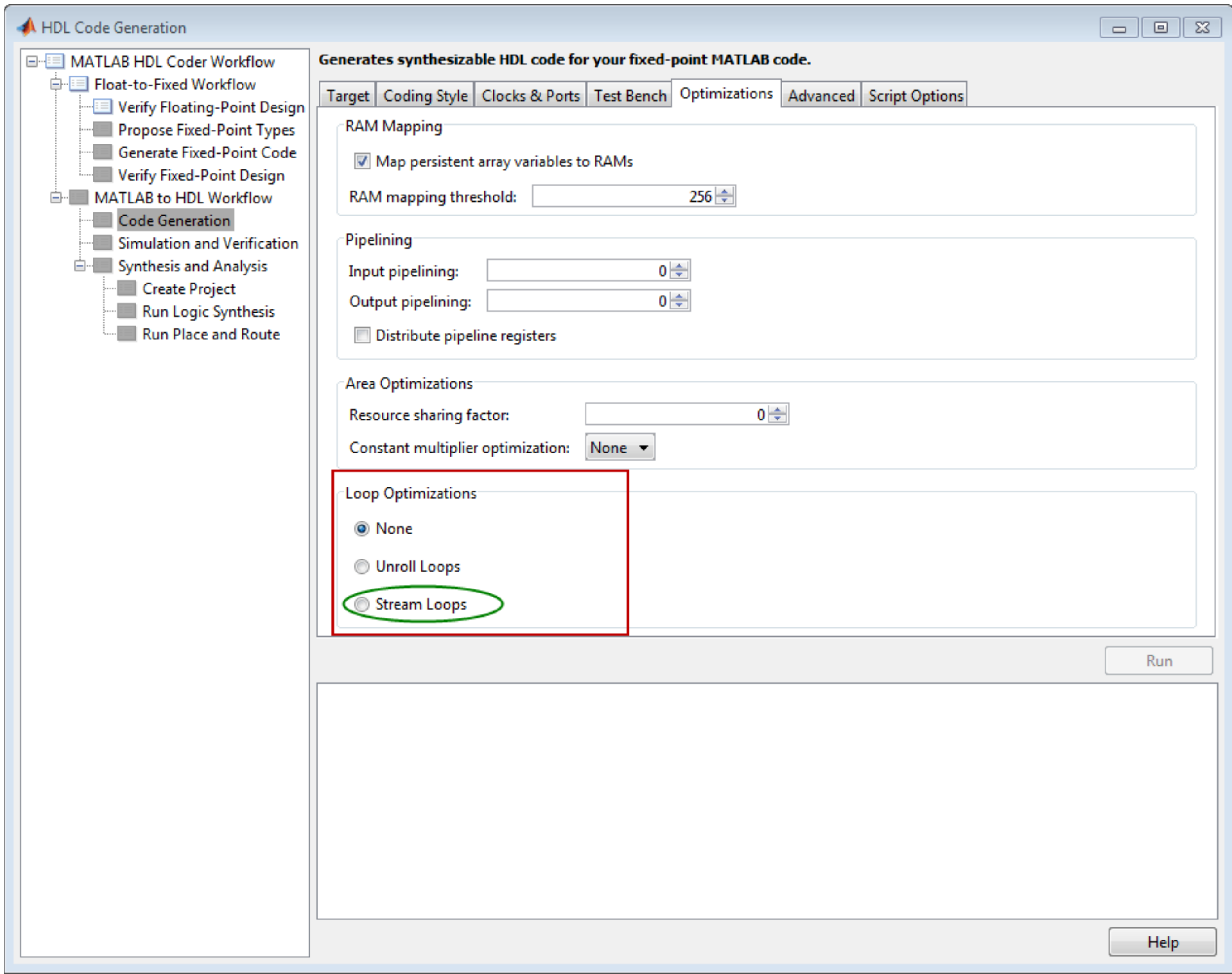
Next, add the file `mlhdlc_fir.m` to the project as the MATLAB Function and `mlhdlc_fir_tb.m` as the MATLAB Test Bench.

Launch the Workflow Advisor.

Refer to “Get Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Turn On Loop Streaming

The loop streaming optimization in HDL Coder converts software loops (either written explicitly using a for-loop statement, or inferred loops from matrix/vector operators) to area-friendly hardware loops.



Run Fixed-Point Conversion and HDL Code Generation

Right-click the **Code Generation** step. Choose the option **Run to selected task** to run all the steps from the beginning through HDL code generation.

Examine the Generated Code

When you synthesize the design with the loop streaming optimization, you see a reduction in area resources in the resource report. Try generating HDL code with and without the optimization.

The resource report without the loop streaming optimization:

Multipliers	16
Adders/Subtractors	31
Registers	106
RAMs	0
Multiplexers	0

The resource report with the loop streaming optimization enabled:

Multipliers	1
Adders/Subtractors	17
Registers	448
RAMs	0
Multiplexers	5

Limitations

Loops will be streamed only if they are regular nested loops. A regular nested loop structure is defined as one where:

- None of the loops in any level of nesting appear in a conditional flow region, i.e. no loop can be embedded within if-else or switch-else regions.
- Loop index variables are monotonically increasing.
- Total number of iterations of the loop structure is non-zero.
- There are no back-to-back loops at the same level of the nesting hierarchy.

Constant Multiplier Optimization to Reduce Area

This example shows how to perform a design-level area optimization in HDL Coder™ by converting constant multipliers into shifts and adds using canonical signed digit (CSD) techniques. The CSD representation of multiplier constants for example, in gain coefficients or filter coefficients) significantly reduces the area of the hardware implementation.

Canonical Signed Digit (CSD) Representation

A signed digit (SD) representation is an augmented binary representation with weights 0,1 and -1. -1 is represented in HDL Coder generated code as 1'.

$$X_{10} = \sum_{r=0}^{B-1} x_r \cdot 2^r$$

where

$$x_r = 0, 1, -1(\overline{1})$$

For example, here are a couple of signed digit representations for 93:

$$X_{10} = 64 + 16 + 13 = 01011101$$

$$X_{10} = 128 - 32 - 2 - 1 = 10\overline{1}000\overline{1}\overline{1}$$

Note that the signed digit representation is non-unique. A canonical signed digit (CSD) representation is an SD representation with the minimum number of nonzero elements.

Here are some properties of CSD numbers:

1. No two consecutive bits in a CSD number are nonzero
2. CSD representation uses minimum number of nonzero digits
3. CSD representation of a number is unique

CSD Multiplier

Let us see how a CSD representation can yield an implementation requiring a minimum number of adders.

Let us look at CSD example:

```
y = 231 * x
  = (11100111) * x                    % 231 in binary form
  = (1001'01001') * x                 % 231 in signed digit form
  = (256 - 32 + 8 - 1) * x            %
  = (x << 8) - (x << 5) + (x << 3) -x % cost of CSD: 3 Adders
```

HDL Coder CSD Implementation

HDL Coder uses a CSD implementation that differs from the traditional CSD implementation. This implementation preferentially chooses adders over subtractors when using the signed digit representation. In this representation, sometimes two consecutive bits in a CSD number can be nonzero. However, similar to the CSD implementation, the HDL Coder implementation uses the minimum number of nonzero digits. For example:

In the traditional CSD implementation, the number 1373 is represented as:

```
1373 = 0101'01'01'001'01
```

This implementation does not have two consecutive nonzero digits in the representation. The cost of this implementation is 1 adder and 4 subtractors.

In the HDL Coder CSD implementation, the number 1373 is represented as:

```
1373 = 00101011001'01
```

This implementation has two consecutive nonzero digits in the representation but uses the same number of nonzero digits as the previous CSD implementation. The cost of this implementation is 4 adders and 1 subtractor which shows that adders are preferred to subtractors.

FCSD Multiplier

A combination of factorization and CSD representation of a constant multiplier can lead to further reduction in hardware cost (number of adders).

FCSD can further reduce the number of adders in the above constant multiplier:

```
y  = 231 * x
y  = (7 * 33) * x
y_tmp = (x << 5) + x
y  = (y_tmp << 3) - y_tmp         % cost of FCSD: 2 Adders
```

CSD/FCSD Costs

This table shows the costs (C) of all 8-bit multipliers.

C	Coefficient
0	1, 2, 4, 8, 16, 32, 64, 128, 256
1	3, 5, 6, 7, 9, 10, 12, 14, 15, 17, 18, 20, 24, 28, 30, 31, 33, 34, 36, 40, 48, 56, 60, 62, 63, 65, 66, 68, 72, 80, 96, 112, 120, 124, 126, 127, 129, 130, 132, 136, 144, 160, 192, 224, 240, 248, 252, 254, 255
2	11, 13, 19, 21, 22, 23, 25, 26, 27, 29, 35, 37, 38, 39, 41, 42, 44, 46, 47, 49, 50, 52, 54, 55, 57, 58, 59, 61, 67, 69, 70, 71, 73, 74, 76, 78, 79, 81, 82, 84, 88, 92, 94, 95, 97, 98, 100, 104, 108, 110, 111, 113, 114, 116, 118, 119, 121, 122, 123, 125, 131, 133, 134, 135, 137, 138, 140, 142, 143, 145, 146, 148, 152, 156, 158, 159, 161, 162, 164, 168, 176, 184, 188, 190, 191, 193, 194, 196, 200, 208, 216, 220, 222, 223, 225, 226, 228, 232, 236, 238, 239, 241, 242, 244, 246, 247, 249, 250, 251, 253
3	43, 45, 51, 53, 75, 77, 83, 85, 86, 87, 89, 90, 91, 93, 99, 101, 102, 103, 105, 106, 107, 109, 115, 117, 139, 141, 147, 149, 150, 151, 153, 154, 155, 157, 163, 165, 166, 167, 169, 170, 172, 174, 175, 177, 178, 180, 182, 183, 185, 186, 187, 189, 195, 197, 198, 199, 201, 202, 204, 206, 207, 209, 210, 212, 214, 215, 217, 218, 219, 221, 227, 229, 230, 231, 233, 234, 235, 237, 243, 245
4	171, 173, 179, 181, 203, 205, 211, 213
	Minimum costs through factorization
2	$45 = 5 \times 9, 51 = 3 \times 17, 75 = 5 \times 15, 85 = 5 \times 17, 90 = 2 \times 9 \times 5, 93 = 3 \times 31, 99 = 3 \times 33, 102 = 2 \times 3 \times 17, 105 = 7 \times 15, 150 = 2 \times 5 \times 15, 153 = 9 \times 17, 155 = 5 \times 31, 165 = 5 \times 33, 170 = 2 \times 5 \times 17, 180 = 4 \times 5 \times 9, 186 = 2 \times 3 \times 31, 189 = 7 \times 9, 195 = 3 \times 65, 198 = 2 \times 3 \times 33, 204 = 4 \times 3 \times 17, 210 = 2 \times 7 \times 15, 217 = 7 \times 31, 231 = 7 \times 33$
3	$171 = 3 \times 57, 173 = 8 + 165, 179 = 51 + 128, 181 = 1 + 180, 211 = 1 + 210, 213 = 3 \times 71, 205 = 5 \times 41, 203 = 7 \times 29$

Reference: Digital Signal Processing with FPGAs by Uwe Meyer-Baese

MATLAB® Design

The MATLAB code used in this example implements a simple FIR filter. The example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_csd';
testbench_name = 'mlhdlc_csd_tb';
```

1. Design: mlhdlc_csd
2. Test Bench: mlhdlc_csd_tb

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_csd_tb
```

Create a Fixed-Point Conversion Config Object

To perform fixed-point conversion, you need a 'fixpt' config object.

Create a 'fixpt' config object and specify your test bench name:

```
close all;
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_csd_tb';
```

Create an HDL Code Generation Config Object

To generate code, you must create an 'hdl' config object and set your test bench name:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_csd_tb';
```

Generate Code without Constant Multiplier Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'None';
```

Enable the 'Unroll Loops' option to inline multiplier constants.

```
hdlcfg.LoopOptimization = 'UnrollLoops';
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_csd
```

Examine the generated code.

```
329    -- filtered output
330    --'mlhdlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
331    p22y_out_mul_temp <= (-2194) * a1;
332    p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
333    p22y_out_mul_temp_1 <= (-1373) * a2;
334    p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
335    p22y_out_add_temp <= p22y_out_add_cast + p22y_out_add_cast_1;
336    p22y_out_add_cast_2 <= resize(p22y_out_add_temp, 30);
337    p22y_out_mul_temp_2 <= 3319 * a3;
338    p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
339    p22y_out_mul_temp_3 <= 6658 * a4;
340    p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
341    p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
342    p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
343    p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
344    y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);
345
```

Take a look at the resource report for adder and multiplier usage without the CSD optimization.

Multipliers	4
Adders/Subtractors	7
Registers	23
RAMs	0
Multiplexers	0

Generate Code with CSD Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'CSD';
```

Enable the 'Unroll Loops' option to inline multiplier constants.

```
hdlcfg.LoopOptimization = 'UnrollLoops';
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_csd
```

Examine the generated code.

```
329     -- filtered output
330     --'mlhdlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
331     -- CSD Encoding (2194) : 0100010010010; Cost (Adders) = 3
332     p22y_out_mul_temp <=  - (((resize(a1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a1 & '0' &
333     p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
334     -- CSD Encoding (1373) : 0101011001'01; Cost (Adders) = 5
335     p22y_out_mul_temp_1 <=  - (((((resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a2 & '0' & '
336     p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
337     p22y_out_add_temp <= p22y_out_add_cast + p22y_out_add_cast_1;
338     p22y_out_add_cast_2 <= resize(p22y_out_add_temp, 30);
339     -- CSD Encoding (3319) : 0110100001'001'; Cost (Adders) = 4
340     p22y_out_mul_temp_2 <= (((resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a3 & '0' &
341     p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
342     -- CSD Encoding (6658) : 01101000000010; Cost (Adders) = 3
343     p22y_out_mul_temp_3 <= ((resize(a4 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a4 & '
344     p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
345     p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
346     p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
347     p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
348     y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);
```

Examine the code with comments that outline the CSD encoding for all the constant multipliers.

Look at the resource report and notice that with the CSD optimization, the number of multipliers is reduced to zero and multipliers are replaced by shifts and adders.

Multipliers	0
Adders/Subtractors	24
Registers	23
RAMs	0
Multiplexers	0

Generate Code with FCSD Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'FCSD';
```

Enable the 'Unroll Loops' option to inline multiplier constants.

```
hdlcfg.LoopOptimization = 'UnrollLoops';
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_csd
```

Examine the generated code.

```
331     -- filtered output
332     --'mlhdlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
333     -- FCSD for 2194 = 2 X 1097; Total Cost = 3
334     -- CSD Encoding (2) : 10; Cost (Adders) = 0
335     p22y_out_factor <= resize(a1 & '0', 28);
336     -- CSD Encoding (1097) : 010001001001; Cost (Adders) = 3
337     p22y_out_mul_temp <=  - (((resize(p22y_out_factor & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(p22y_out_factor & '0' & '0' & '0
338     p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
339     -- CSD Encoding (1373) : 0101011001'01; Cost (Adders) = 5
340     p22y_out_mul_temp_1 <=  - (((((resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' &
341     p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
342     p22y_out_add_temp <= p22y_out_add_cast + p22y_out_add_cast_1;
343     p22y_out_add_cast_2 <= resize(p22y_out_add_temp, 30);
344     -- CSD Encoding (3319) : 0110100001'001'; Cost (Adders) = 4
345     p22y_out_mul_temp_2 <= (((resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' &
346     p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
347     -- FCSD for 6658 = 2 X 3329; Total Cost = 3
348     -- CSD Encoding (2) : 10; Cost (Adders) = 0
349     p22y_out_factor_1 <= resize(a4 & '0', 28);
350     -- CSD Encoding (3329) : 0110100000001; Cost (Adders) = 3
351     p22y_out_mul_temp_3 <= ((resize(p22y_out_factor_1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(p22y_out_factor_1 & '0' &
352     p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
353     p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
354     p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
355     p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
356     y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);
357
358     y_out_2 <= y_out_1;
```

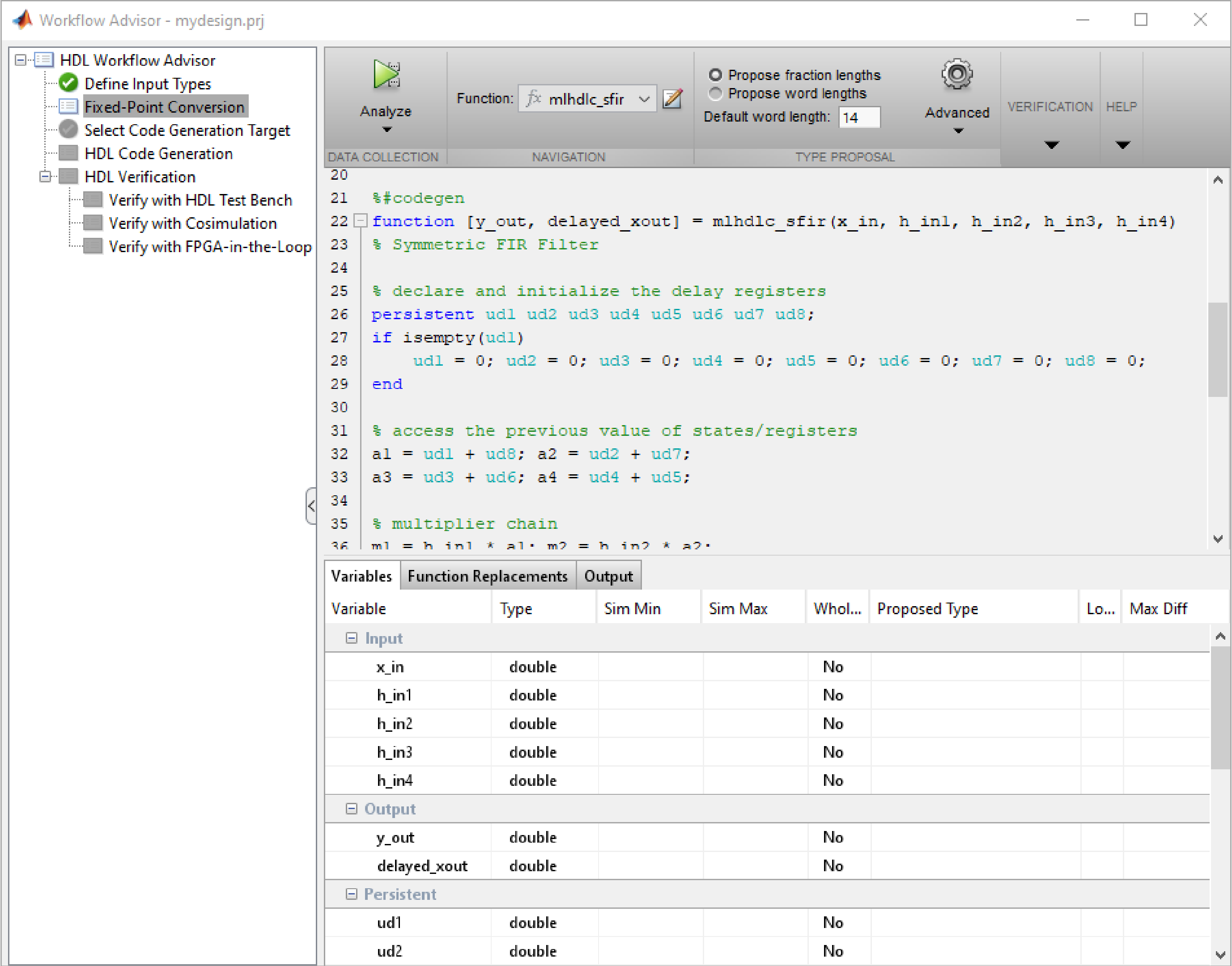
Examine the code with comments that outline the FCSD encoding for all the constant multipliers. In this particular example, the generated code is identical in terms of area resources for the multiplier constants. However, take a look at the factorizations of the constants in the generated code.

If you choose the 'Auto' option, HDL Coder will automatically choose between the CSD and FCSD options for the best result.

HDL Workflow Advisor Reference

- “HDL Workflow Advisor” on page 9-2
- “MATLAB to HDL Code and Synthesis” on page 9-6

HDL Workflow Advisor



Overview

The HDL Workflow Advisor is a tool that supports a suite of tasks covering the stages of the ASIC and FPGA design process, including converting floating-point MATLAB algorithms to fixed-point algorithms. Some tasks perform code validation or checking; others run the HDL code generator or third-party tools. Each folder at the top level of the HDL Workflow Advisor contains a group of related tasks that you can select and run.

Use the HDL Workflow Advisor to:

- Convert floating-point MATLAB algorithms to fixed-point algorithms.

  If you already have a fixed-point MATLAB algorithm, set **Design needs conversion to Fixed Point?** to No to skip this step.

- Generate HDL code from fixed-point MATLAB algorithms.
- Simulate the HDL code using a third-party simulation tool.
- Synthesize the HDL code and run a mapping process that maps the synthesized logic design to the target FPGA.
- Run a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.

HDL Workflow Advisor is not available in Simulink Online™.

Procedures

Automatically Run Tasks

To automatically run the tasks within a folder:

1. Click the **Run** button. The tasks run in order until a task fails.

   Alternatively, right-click the folder to open the context menu. From the context menu, select `Run` to run the tasks within the folder.
2. If a task in the folder fails:
   a. Fix the failure using the information in the results pane.
   b. Continue the run by clicking the **Run** button.

Run Individual Tasks

To run an individual task:

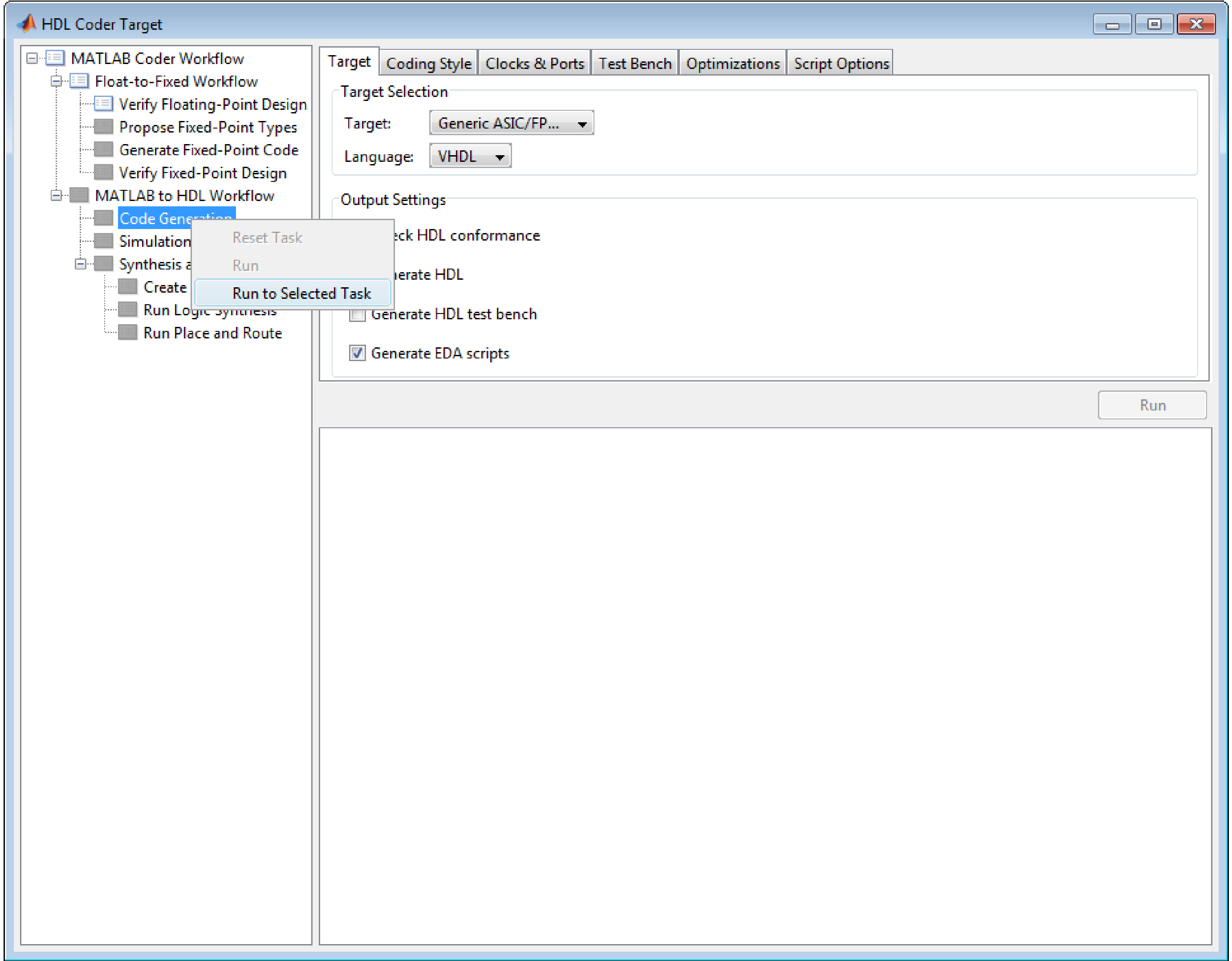
1. Click the **Run** button.

   Alternatively, right-click the task to open the context menu. From the context menu, select `Run` to run the selected task.
2. Review Results. The possible results are:
   Pass: Move on to the next task.
   Warning: Review results, decide whether to move on or fix.
   Fail: Review results, do not move on without fixing.
3. If required, fix the issue using the information in the results pane.
4. Once you have fixed a **Warning** or **Failed** task, rerun the task by clicking **Run**.

Run to Selected Task

To run the tasks up to and including the currently selected task:

1. Select the last task that you want to run.
2. Right-click this task to open the context menu.
3. From the context menu, select `Run to Selected Task`.



Note If a task before the selected task fails, the Workflow Advisor stops at the failed task.

Reset a Task

To reset a task:

1. Select the task that you want to reset.
2. Right-click this task to open the context menu.
3. From the context menu, select `Reset Task` to reset this and subsequent tasks.

Reset All Tasks in a Folder

To reset a task:

1. Select the folder that you want to reset.
2. Right-click this folder to open the context menu.

3. From the context menu, select `Reset Task` to reset the tasks this folder and subsequent folders.

MATLAB to HDL Code and Synthesis

In this section...
“MATLAB to HDL Code Conversion” on page 9-6
“Code Generation: Target Tab” on page 9-6
“Code Generation: Coding Style Tab” on page 9-7
“Code Generation: Clocks and Ports Tab” on page 9-8
“Code Generation: Test Bench Tab” on page 9-10
“Code Generation: Optimizations Tab” on page 9-11
“Simulation and Verification” on page 9-12
“Synthesis and Analysis” on page 9-13

MATLAB to HDL Code Conversion

The **MATLAB to HDL Workflow** task in the HDL Workflow Advisor generates HDL code from fixed-point MATLAB code, and simulates and verifies the HDL against the fixed-point algorithm. HDL Coder then runs synthesis, and optionally runs place and route to generate a circuit description suitable for programming an ASIC or FPGA.

Code Generation: Target Tab

Select target hardware and language and required outputs.

Input Parameters

Target

Target hardware. Select from the list:
- Generic ASIC/FPGA
- Xilinx
- Altera
- Simulation

Language

Select the language (VHDL, Verilog or SystemVerilog) in which code is generated. The selected language is referred to as the target language.

Default: VHDL

Check HDL Conformance

Enable HDL conformance checking.

Default: Off

Generate HDL

Enable generation of HDL code for the fixed-point MATLAB algorithm.

Default: On

Generate HDL Test Bench

Enable generation of HDL code for the fixed-point test bench.

Default: Off

Generate EDA Scripts

Enable generation of script files for third-party electronic design automation (EDA) tools. These scripts let you compile and simulate generated HDL code and synthesize generated HDL code.

Default: On

Code Generation: Coding Style Tab

Parameters that affect the style of the generated code.

Input Parameters

Preserve MATLAB code comments

Include MATLAB code comments in generated code.

Default: On

Include MATLAB source code as comments

Include MATLAB source code as comments in the generated code. The comments precede the associated generated code. Includes the function signature in the function banner.

Default: On

Generate Report

Enable a code generation report.

Default: Off

VHDL File Extension

Specify the file name extension for generated VHDL files.

Default: .vhd

Verilog File Extension

Specify the file name extension for generated Verilog files.

Default: .v

Comment in header

Specify comment lines in header of generated HDL and test bench files.

Default: None

Text entered in this field as a character vector generates a comment line in the header of the generated code. The code generator adds leading comment characters for the target language. When newlines or linefeeds are included in the text, the code generator emits single-line comments for each newline.

Package postfix

HDL Coder applies this option only if a package file is required for the design.

Default: _pkg

Entity conflict postfix

Specify the character vector to resolve duplicate VHDL entity, or Verilog or SystemVerilog module names in generated code.

Default: _block

Reserved word postfix

Specify a character vector to append to value names, postfix values, or labels that are VHDL, Verilog or SystemVerilog reserved words.

Default: _rsvd

Clocked process postfix

Specify a character vector to append to HDL clock process names.

Default: _process

Complex real part postfix

Specify a character vector to append to real part of complex signal names.

Default: '_re'

Complex imaginary part postfix

Specify a character vector to append to imaginary part of complex signal names.

Default: '_im'

Pipeline postfix

Specify a character vector to append to names of input or output pipeline registers.

Default: '_pipe'

Enable prefix

Specify the base name as a character vector for internal clock enables and other flow control signals in generated code.

Default: 'enb'

Code Generation: Clocks and Ports Tab

Clock and port settings

Input Parameters

Reset type

Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers.

Default: Asynchronous

Reset Asserted level

Specify whether the asserted (active) level of reset input signal is active-high or active-low.

Default: Active-high

Reset input port

Enter the name for the reset input port in generated HDL code.

Default: reset

Clock input port

Specify the name for the clock input port in generated HDL code.

Default: clk

Clock enable input port

Specify the name for the clock enable input port in generated HDL code.

Default: clk

Oversampling factor

Specify frequency of global oversampling clock as a multiple of the design under test (DUT) base rate (1).

Default: 1

Input data type

Specify the HDL data type for input ports.

For VHDL, the options are:

- std_logic_vector

  Specifies VHDL type STD_LOGIC_VECTOR
- signed/unsigned

  Specifies VHDL type SIGNED or UNSIGNED

Default: std_logic_vector

For Verilog, the options are:

- In generated Verilog code, the data type for all ports is 'wire'. Therefore, **Input data type** is disabled when the target language is Verilog.

Default: wire

Output data type

Specify the HDL data type for output data types.

For VHDL, the options are:

- Same as input data type

  Specifies that output ports have the same type specified by Input data type.
- std_logic_vector

  Specifies VHDL type STD_LOGIC_VECTOR
- signed/unsigned

  Specifies VHDL type SIGNED or UNSIGNED

Default: Same as input data type

For Verilog, the options are:

- In generated Verilog code, the data type for all ports is ‘wire’. Therefore, Output data type is disabled when the target language is Verilog.

  Default: wire

Clock enable output port

Specify the name for the clock enable input port in generated HDL code.

Default: clk_enable

Code Generation: Test Bench Tab

Test bench settings.

Input Parameters

Test bench name postfix

Specify a character vector appended to names of reference signals generated in test bench code.

Default: '_tb’

Force clock

Specify whether the test bench forces clock enable input signals.

Default: On

Clock High time (ns)

Specify the period, in nanoseconds, during which the test bench drives clock input signals high (1).

Default: 5

Clock low time (ns)

Specify the period, in nanoseconds, during which the test bench drives clock input signals low (0).

Default: 5

Hold time (ns)

Specify a hold time, in nanoseconds, for input signals and forced reset input signals.

Default: 2 (given the default clock period of 10 ns)

Setup time (ns)

Display setup time for data input signals.

Default: 0

Force clock enable

Specify whether the test bench forces clock enable input signals.

Default: On

Clock enable delay (in clock cycles)

Define elapsed time (in clock cycles) between deassertion of reset and assertion of clock enable.

Default: 1

Force reset

Specify whether the test bench forces reset input signals.

Default: On

Reset length (in clock cycles)

Define length of time (in clock cycles) during which reset is asserted.

Default: 2

Hold input data between samples

Specify how long subrate signal values are held in valid state.

Default: On

Initialize testbench inputs

Specify initial value driven on test bench inputs before data is asserted to device under test (DUT).

Default: Off

Multi file testbench

Divide generated test bench into helper functions, data, and HDL test bench code files.

Default: Off

Test bench data file name postfix

Specify suffix added to test bench data file name when generating multi-file test bench.

Default: '_data’

Test bench reference post fix

Specify a character vector to append to names of reference signals generated in test bench code.

Default: '_ref'

Ignore data checking (number of samples)

Specify number of samples during which output data checking is suppressed.

Default: 0

Use fiaccel to accelerate test bench logging

To generate a test bench, HDL Coder simulates the original MATLAB code. Use the Fixed-Point Designer `fiaccel` function to accelerate this simulation and accelerate test bench logging.

Default: On

Code Generation: Optimizations Tab

Optimization settings

Input Parameters

Map persistent array variables to RAMs

Select to map persistent array variables to RAMs instead of mapping to shift registers.

Default: Off

Dependencies:

- **RAM Mapping Threshold**
- **Persistent variable names for RAM Mapping**

RAM Mapping Threshold

Specify the minimum RAM size required for mapping persistent array variables to RAMs.

Default: 256

Persistent variable names for RAM Mapping

Provide the names of the persistent variables to map to RAMs.

Default: None

Input Pipelining

Specify number of pipeline registers to insert at top level input ports. Can improve performance and help to meet timing constraints.

Default: 0

Output Pipelining

Specify number of pipeline registers to insert at top level output ports. Can improve performance and help to meet timing constraints.

Default: 0

Distribute Pipeline Registers

Reduces critical path by changing placement of registers in design. Operates on all registers, including those inserted using the **Input Pipelining** and **Output Pipelining** parameters, and internal design registers.

Default: Off

Sharing Factor

Number of additional sources that can share a single resource, such as a multiplier. To share resources, set **Sharing Factor** to 2 or higher; a value of 0 or 1 turns off sharing.

In a design that performs identical multiplication operations, HDL Coder can reduce the number of multipliers by the sharing factor. This can significantly reduce area.

Default: 0

Simulation and Verification

Simulates the generated HDL code using the selected simulation tool.

Input Parameters

Simulation tool

Lists the available simulation tools.

Default: None

Skip this step

Default: Off

Results and Recommended Actions

Conditions	Recommended Action
No simulation tool available on system path.	Add your simulation tool path to the MATLAB system path, then restart MATLAB. For more information, see "Synthesis Tool Path Setup".

Synthesis and Analysis

This folder contains tasks to create a synthesis project for the HDL code. The task then runs the synthesis and, optionally, runs place and route to generate a circuit description suitable for programming an ASIC or FPGA.

Input Parameters

Skip this step

Default: Off

Skip this step if you are interested only in simulation or you do not have a synthesis tool.

Create Project

Create synthesis project for supported synthesis tool.

Description

This task creates a synthesis project for the selected synthesis tool and loads the project with the HDL code generated for your MATLAB algorithm.

You can select the family, device, package, and speed that you want.

When the project creation is complete, the HDL Workflow Advisor displays a link to the project in the right pane. Click this link to view the project in the synthesis tool's project window.

Input Parameters

Synthesis Tool

Select from the list:

- `Altera Quartus II`

  Generate a synthesis project for Altera Quartus II. When you select this option, HDL Coder sets:

- **Chip Family** to `Stratix II`
- **Device Name** to `EP2S60F1020C4`

You can manually change these settings.
- `Xilinx ISE`

Generate a synthesis project for Xilinx ISE. When you select this option, HDL Coder:

- Sets **Chip Family** to `Virtex4`
- Sets **Device Name** to `xc4vsx35`
- Sets **Package Name** to `ff6...`
- Sets **Speed Value** to `-...`

You can manually change these settings.

Default: `No Synthesis Tool Specified`

When you select `No Synthesis Tool Specified`, HDL Coder does not generate a synthesis project. It clears and disables the fields in the **Synthesis Tool Selection** pane.

Chip Family

Target device family.

Default: None

Device Name

Specific target device, within selected family.

Default: None

Package Name

Available package choices. The family and device determine these choices.

Default: None

Speed Value

Available speed choices. The family, device, and package determine these choices.

Default: None

Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails to create project.	Read the error message returned by synthesis tool, then check the synthesis tool version, and check that you have write permission for the project folder.
Synthesis tool does not appear in dropdown list.	Add your synthesis tool path to the MATLAB system path, then restart MATLAB. For more information, see “Synthesis Tool Path Setup”.

Run Logic Synthesis

Launch selected synthesis tool and synthesize the generated HDL code.

Description

This task:

- Launches the synthesis tool in the background.
- Opens the previously generated synthesis project, compiles HDL code, synthesizes the design, and emits netlists and related files.
- Displays a synthesis log in the **Result** subpane.

Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails when running place and route.	Read the error message returned by the synthesis tool, modify the MATLAB code, then rerun from the beginning of the HDL Coder workflow.

Run Place and Route

Launches the synthesis tool in the background and runs a Place and Route process.

Description

This task:

- Launches the synthesis tool in the background.
- Runs a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.
- Displays a log in the Result subpane.

Input Parameters

Skip this step

If you select **Skip this step**, the HDL Workflow Advisor executes the workflow, but omits the Perform Place and Route, marking it Passed. You might want to select **Skip this step** if you prefer to do place and route work manually.

Default: Off

Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails when running place and route.	Read the error message returned by the synthesis tool, modify the MATLAB code, then rerun from the beginning of the HDL Coder workflow.

SystemC Code Generation from MATLAB

MATLAB to HLS Examples for Communications and Signal Processing Applications

- “High-Level Synthesis Code Generation for LMS Filter” on page 10-2
- “High-Level Synthesis Code Generation for Bisection Algorithm” on page 10-8
- “High-Level Synthesis Code Generation for Data Packetization” on page 10-12
- “High-Level Synthesis Code Generation for DF2T Filter” on page 10-18
- “High-Level Synthesis Code Generation for Transmit and Receive FIFO Registers” on page 10-21
- “High-Level Synthesis Code Generation for Contrast Adjustment” on page 10-28
- “High-Level Synthesis Code Generation for Image Format Conversion from RGB to YUV” on page 10-37
- “High-Level Synthesis Code Generation for Advanced Encryption Standard” on page 10-41

High-Level Synthesis Code Generation for LMS Filter

This example shows how to generate High-Level Synthesis (HLS) code from a MATLAB® design that implements an LMS filter. The example also illustrates how to design a test bench that cancels out the noise signal by using this filter.

LMS Filter MATLAB Design

The MATLAB design used in the example is an implementation of an LMS (Least Mean Squares) filter. The LMS filter is a class of adaptive filter that identifies an FIR filter signal that is embedded in the noise. The LMS filter design implementation in MATLAB consists of a top-level function `mlhdlc_lms_fcn` that calculates the optimal filter coefficients to reduce the difference between the output signal and the desired signal.

```
design_name = 'mlhdlc_lms_fcn';
testbench_name = 'mlhdlc_lms_noise_canceler_tb';
```

Review the MATLAB design:

```
open(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB Design: Adaptive Noise Canceler algorithm using Least Mean Square
% (LMS) filter implemented in MATLAB
%
% Key Design pattern covered in this example:
% (1) Use of function calls
% (2) Function inlining vs instantiation knobs available in the coder
% (3) Use of system objects in the testbench to stream test vectors into the design
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%#codegen
function [filtered_signal, y, fc] = mlhdlc_lms_fcn(input, ...
                                        desired, step_size, reset_weights)
% 'input'  : The signal from Exterior Mic which records the ambient noise.
% 'desired': The signal from Pilot's Mic which includes
%            original music signal and the noise signal
% 'err_sig': The difference between the 'desired' and the filtered 'input'
%           It represents the estimated music signal (output of this block)
%
% The LMS filter is trying to retrieve the original music signal('err_sig')
% from Pilot's Mic by filtering the Exterior Mic's signal and using it to
% cancel the noise in Pilot's Mic. The coefficients/weights of the filter
% are updated(adapted) in real-time based on 'input' and 'err_sig'.

% register filter coefficients
persistent filter_coeff;
if isempty(filter_coeff)
    filter_coeff = zeros(1, 40);
end

% Variable Filter: Call 'mtapped_delay_fcn' function on path to create
% 40-step tapped delay
delayed_signal = mtapped_delay_fcn(input);
```

```
% Apply filter coefficients
weight_applied = delayed_signal .* filter_coeff;

% Call treesum function on matlab path to sum up the results
filtered_signal = mtreesum_fcn(weight_applied);

% Output estimated Original Signal
td = desired;
tf = filtered_signal;
esig = td - tf;
y = esig;

% Update Weights: Call 'update_weight_fcn' function on MATLAB path to
% calculate the new weights
updated_weight = update_weight_fcn(step_size, esig, delayed_signal, ...
                                   filter_coeff, reset_weights);

% update filter coefficients register
filter_coeff = updated_weight;
fc = filter_coeff;

function y = mtreesum_fcn(u)
%Implement the 'sum' function without a for-loop
%  y = sum(u);

%  The loop based implementation of 'sum' function is not ideal for
%  HDL generation and results in a longer critical path.
%  A tree is more efficient as it results in
%  delay of log2(N) instead of a delay of N delay

%  This implementation shows how to explicitly implement the vector sum in
%  a tree shape to enable hardware optimizations.

%  The ideal way to code this generically for any length of 'u' is to use
%  recursion but it is not currently supported by MATLAB Coder


% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

% This implementation is hardwired for a 40tap filter.

level1 = vsum(u);
level2 = vsum(level1);
level3 = vsum(level2);
level4 = vsum(level3);
level5 = vsum(level4);
level6 = vsum(level5);
y = level6;

function output = vsum(input)

coder.inline('always');

vt = input(1:2:end);

for i = int32(1:numel(input)/2)
```

```
    k = int32(i*2);
    vt(i) = vt(i) + input(k);
end

output = vt;

function tap_delay = mtapped_delay_fcn(input)
% The Tapped Delay function delays its input by the specified number
% of sample periods, and outputs all the delayed versions in a vector
% form. The output includes current input

% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

persistent u_d;
if isempty(u_d)
    u_d = zeros(1,40);
end


u_d = [u_d(2:40), input];

tap_delay = u_d;

function weights = update_weight_fcn(step_size, err_sig, ...
            delayed_signal, filter_coeff, reset_weights)
% This function updates the adaptive filter weights based on LMS algorithm

%   Copyright 2007-2022 The MathWorks, Inc.

% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

step_sig = step_size .* err_sig;
correction_factor = delayed_signal .* step_sig;
updated_weight = correction_factor + filter_coeff;

if reset_weights
    weights = zeros(1,40);
else
    weights = updated_weight;
end
```

The MATLAB function is modular and uses functions:

- `mtapped_delay_fcn` to calculate delayed versions of the input signal in vector form.
- `mtreesum_fcn` to calculate the sum of the applied weights in a tree structure. The individual sum is calculated by using a `vsum` function.
- `update_weight_fcn` to calculate the updated filter weights based on the least mean square algorithm.

LMS Filter MATLAB Test Bench

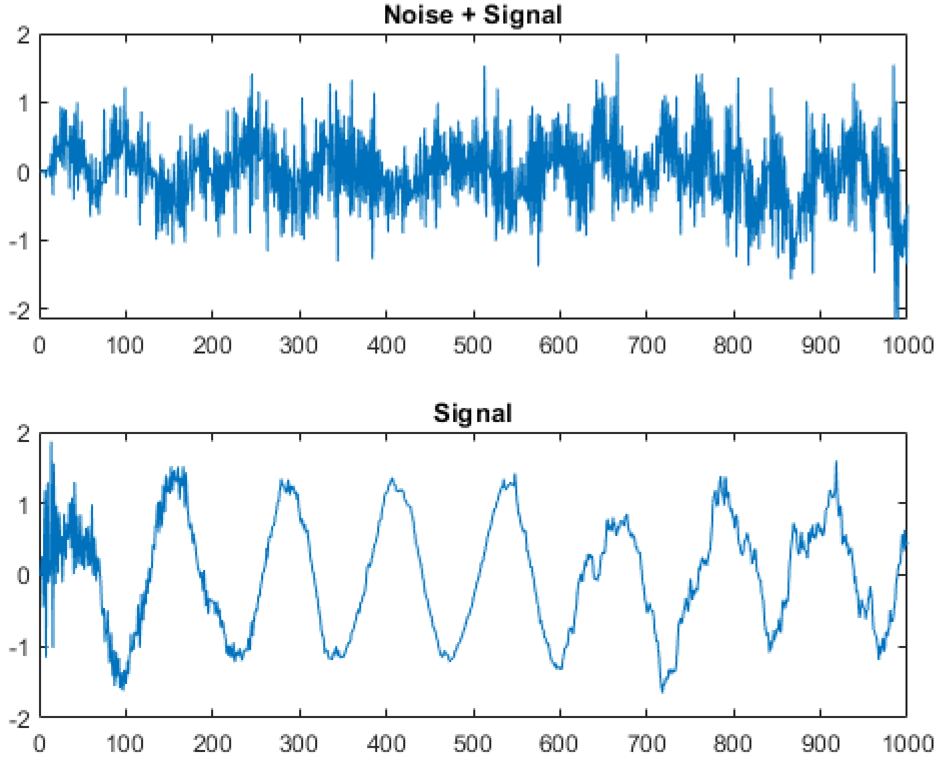
Review the MATLAB test bench:

```
open(testbench_name)


% Returns an adaptive FIR filter System object,
% HLMS, that computes the filtered output, filter error and the filter
% weights for a given input and desired signal using the Least Mean
% Squares (LMS) algorithm.

%   Copyright 2011-2022 The MathWorks, Inc.
clear('mlhdlc_lms_fcn');

hfilt2 = dsp.FIRFilter(...
        'Numerator', fir1(10, [.5, .75]));
rng('default'); % always default to known state
x = randn(1000,1);                                % Noise
d = step(hfilt2, x) + sin(0:.05:49.95)';         % Noise + Signal

stepSize = 0.01;
reset_weights =false;

hSrc = dsp.SignalSource(x);
hDesiredSrc = dsp.SignalSource(d);

hOut = dsp.SignalSink;
hErr = dsp.SignalSink;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Call to the design
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
while (~isDone(hSrc))
    [y, e] = mlhdlc_lms_fcn(step(hSrc), step(hDesiredSrc), ...
                                        stepSize, reset_weights);
    step(hOut, y);
    step(hErr, e);
end

figure('Name', [mfilename, '_signal_plot']);
subplot(2,1,1), plot(hOut.Buffer), title('Noise + Signal');
subplot(2,1,2),plot(hErr.Buffer), title('Signal');
```

Test the MATLAB Algorithm

To avoid run-time errors, simulate the design by using the HDL test bench.

```
mlhdlc_lms_noise_canceler_tb
```



Create HDL Coder Project

To generate High-Level Synthesis (HLS) code from a MATLAB design:

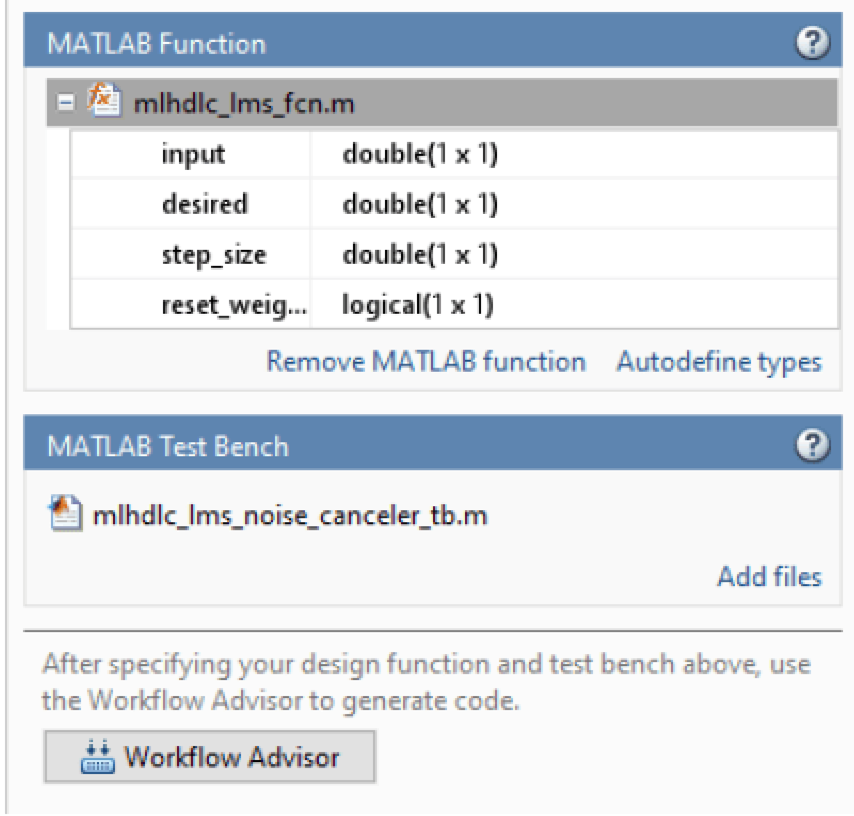
1. Create a HDL Coder project:

```
coder -hdlcoder -new mlhdlc_lms_nc
```

2. Add the file `mlhdlc_lms_fcn.m` to the project as the **MATLAB Function** and `mlhdlc_lms_noise_canceler_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefine types** to use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_lms_fcn`.

For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14 or “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.



Run Fixed-Point Conversion and HLS Code Generation

To generate HLS code from a MATLAB design:

1. At the MATLAB command line, setup the path for HLS code generation by using the function `hdlsetuphlstoolpath`.

2. Start the Workflow Advisor by clicking the **Workflow Advisor** button.

3. In the **HDL Workflow Advisor** step, select **Code Generation Workflow** as **MATLAB to HLS**.

4. Select **Cadence Stratus** as the **Synthesis tool** for **Select Code Generation Target**.

5. Right-click the **HLS Code Generation** and choose the option **Run to selected task** to run all the steps from the beginning through the HLS code generation.

A single HLS file `mlhdlc_lms_fcn_fixptClass.hpp` is generated for the MATLAB design. To examine the generated HLS code for the filter design, click the hyperlinks in the Code Generation Log window.

High-Level Synthesis Code Generation for Bisection Algorithm

You can generate High-Level Synthesis (HLS) code from a MATLAB® design that implements a bisection algorithm to calculate the square root of a number in fixed-point notation.

MATLAB Design

First, set up the `sqrt` model.

```
mlhdlc_demo_setup('sqrt');

% Design Sqrt
design_name = 'mlhdlc_sqrt';

% Test Bench for Sqrt
testbench_name = 'mlhdlc_sqrt_tb';
```

Review the `sqrt` design

```
dbtype(design_name)
```

```
1     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2     % MATLAB design: Pipelined Bisection Square root algorithm
3     %
4     % Introduction:
5     %
6     % Implement SQRT by the bisection algorithm in a pipeline, for unsigned fixed
7     % point numbers (also why you don't need to run fixed-point conversion for this design).
8     % The demo illustrates the usage of a pipelined implementation for numerical algorithms.
9     %
10    % Key Design pattern covered in this example:
11    % (1) State of the bisection algorithm is maintained with persistent variables
12    % (2) Stages of the bisection algorithm are implemented in a pipeline
13    % (3) Code is written in a parameterized fashion, i.e. word-length independent, to work fo
14    %
15    % Ref. 1. R. W. Hamming, "Numerical Methods for Scientists and Engineers," 2nd, Ed, pp 67-6
16    %      2. Bisection method, http://en.wikipedia.org/wiki/Bisection_method, (accessed 02/18/
17    %
18
19    %   Copyright 2013-2015 The MathWorks, Inc.
20
21    %#codegen
22    function [y,z] = mlhdlc_sqrt( x )
23        persistent sqrt_pipe
24        persistent in_pipe
25       if isempty(sqrt_pipe)
26           sqrt_pipe = fi(zeros(1,x.WordLength),numerictype(x));
27           in_pipe = fi(zeros(1,x.WordLength),numerictype(x));
28       end
29
30       % Extract the outputs from pipeline
31       y = sqrt_pipe(x.WordLength);
32       z = in_pipe(x.WordLength);
33
34       % for analysis purposes you can calculate the error between the fixed-point bisection ro
```

```
35      %Q = [double(y).^2, double(z)];
36      %[Q, diff(Q)]
37
38      % work the pipeline
39      for itr = x.WordLength-1:-1:1
40          % move pipeline forward
41          in_pipe(itr+1) = in_pipe(itr);
42          % guess the bits of the square-root solution from MSB to the LSB of word length
43          sqrt_pipe(itr+1) = guess_and_update( sqrt_pipe(itr), in_pipe(itr+1), itr );
44      end
45
46      %% Prime the pipeline
47      % with new input and the guess
48      in_pipe(1) = x;
49      sqrt_pipe(1) = guess_and_update( fi(0,numerictype(x)), x, 1 );
50
51      %% optionally print state of the pipeline
52      %disp('************** State of Pipeline **********************')
53      %double([in_pipe; sqrt_pipe])
54
55      return
56  end
57
58  % Guess the bits of the square-root solution from MSB to the LSB in
59  % a binary search-fashion.
60  function update = guess_and_update( prev_guess, x, stage )
61      % Key step of the bisection algorithm is to set the bits
62      guess = bitset( prev_guess, x.WordLength - stage + 1);
63      % compare if the set bit is a candidate solution to retain or clear it
64      if ( guess*guess <= x )
65          update = guess;
66      else
67          update = prev_guess;
68      end
69      return
70  end
```

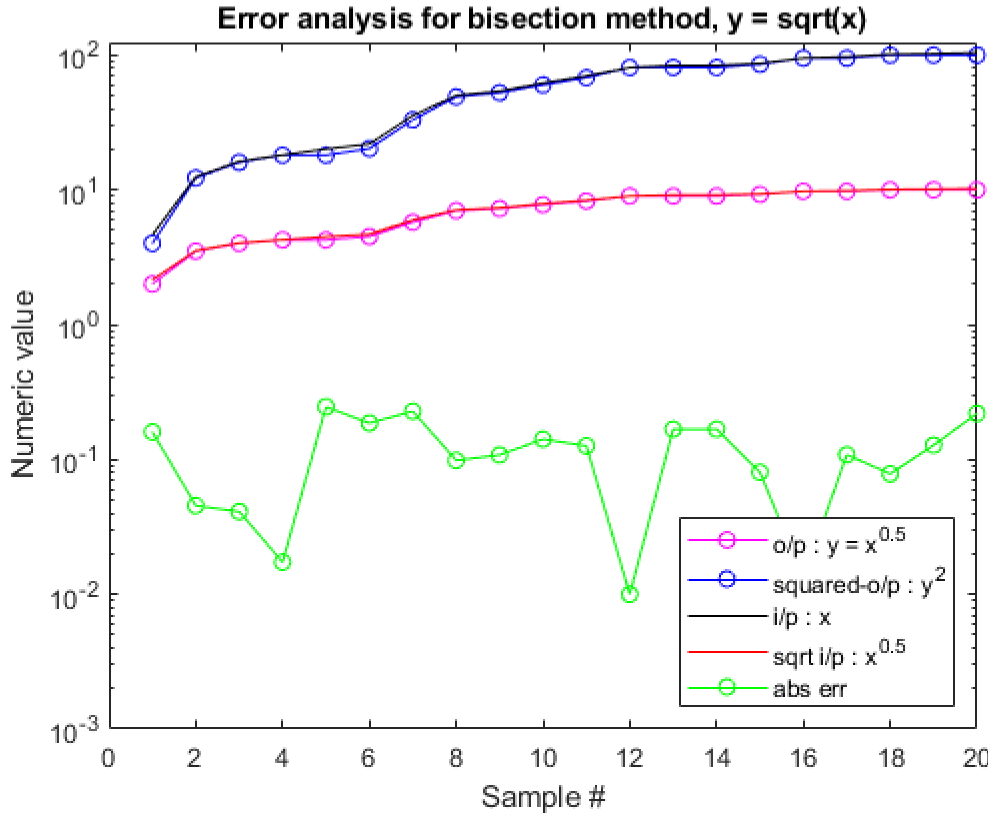
Simulate the Design

It is a best practice to simulate the design with the test bench prior to code generation to check for run-time errors.

```
mlhdlc_sqrt_tb
```

```
Iter = 01| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 02| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 03| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 04| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 05| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 06| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 07| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 08| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 09| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 10| Input = 0.000| Output = 0000000000 (0.00) | actual = 0.000000 | abserror = 0.000000
Iter = 11| Input = 4.625| Output = 0000010000 (2.00) | actual = 2.150581 | abserror = 0.150581
Iter = 12| Input = 12.500| Output = 0000011100 (3.50) | actual = 3.535534 | abserror = 0.035534
Iter = 13| Input = 16.250| Output = 0000100000 (4.00) | actual = 4.031129 | abserror = 0.031129
Iter = 14| Input = 18.125| Output = 0000100010 (4.25) | actual = 4.257347 | abserror = 0.007347
```

```
Iter = 15| Input = 20.125| Output = 0000100010 (4.25) | actual = 4.486090 | abserror = 0.236090
Iter = 16| Input = 21.875| Output = 0000100100 (4.50) | actual = 4.677072 | abserror = 0.177072
Iter = 17| Input = 35.625| Output = 0000101110 (5.75) | actual = 5.968668 | abserror = 0.218668
Iter = 18| Input = 50.250| Output = 0000111000 (7.00) | actual = 7.088723 | abserror = 0.088723
Iter = 19| Input = 54.000| Output = 0000111010 (7.25) | actual = 7.348469 | abserror = 0.098469
Iter = 20| Input = 62.125| Output = 0000111110 (7.75) | actual = 7.881941 | abserror = 0.131941
Iter = 21| Input = 70.000| Output = 0001000010 (8.25) | actual = 8.366600 | abserror = 0.116600
Iter = 22| Input = 81.000| Output = 0001001000 (9.00) | actual = 9.000000 | abserror = 0.000000
Iter = 23| Input = 83.875| Output = 0001001000 (9.00) | actual = 9.158330 | abserror = 0.158330
Iter = 24| Input = 83.875| Output = 0001001000 (9.00) | actual = 9.158330 | abserror = 0.158330
Iter = 25| Input = 86.875| Output = 0001001010 (9.25) | actual = 9.320676 | abserror = 0.070676
Iter = 26| Input = 95.125| Output = 0001001110 (9.75) | actual = 9.753205 | abserror = 0.003205
Iter = 27| Input = 97.000| Output = 0001001110 (9.75) | actual = 9.848858 | abserror = 0.098858
Iter = 28| Input = 101.375| Output = 0001010000 (10.00) | actual = 10.068515 | abserror = 0.06851
Iter = 29| Input = 102.375| Output = 0001010000 (10.00) | actual = 10.118053 | abserror = 0.11805
Iter = 30| Input = 104.250| Output = 0001010000 (10.00) | actual = 10.210289 | abserror = 0.21028
```



Create HDL Coder™ Project

Create an HDL Coder project.

```
coder -hdlcoder -new mlhdlc_sqrt_prj
```

Add the file `mlhdlc_sqrt.m` to the project as the MATLAB Function. Add the file `mlhdlc_sqrt_tb.m` as the MATLAB Test Bench.

For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14 or “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.

HLS Code Generation

This design is already in fixed point and suitable for HLS code generation. You do not need to run floating point to fixed point conversion on this design.

To generate HLS code from the MATLAB design:

1. At the MATLAB command line, set up the path for HLS code generation by using the function `hdlsetuphlstoolpath`.

2. Start the Workflow Advisor by clicking the **Workflow Advisor** button.

3. In the **HDL Workflow Advisor**, select **Code Generation Workflow** as **MATLAB to HLS**.

4. In the **Select Code Generation Target** step, from the **Synthesis tool** list, select **Cadence Stratus**.

5. Right-click the **HLS Code Generation** task and select **Run to selected task** to run all the steps from the beginning through the HLS code generation.

Examine the generated HLS code by clicking the hyperlinks in the **HLS Code Generation** log window.

See Also

- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “MATLAB to HLS Code Generation Options” on page 12-4

High-Level Synthesis Code Generation for Data Packetization

This example shows how to generate High-Level Synthesis (HLS) code from a MATLAB® design that packetizes a transmit sequence.

Introduction

In wireless communication, systems receive data that is oversampled at the radio frequency (RF) front end. This data serves several purposes, including providing sufficient sampling rates for receive filtering.

One of the most important functions is to provide multiple sampling points on the received waveform such that data can be sampled near the maximum amplitude point in the received waveform. This example illustrates a basic lead-lag time offset estimation core, operating recursively.

The generated hardware core for this design operates at 1/os_rate, where os_rate is the oversampled rate. That is, for 8 oversampled clock cycles, this core iterates once. The output is at the symbol rate.

```
design_name = 'mlhdlc_comms_data_packet';
testbench_name = 'mlhdlc_comms_data_packet_tb';
```

Review the MATLAB design.

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Data packetization
%
% Introduction:
%
% This core is meant to illustrate packetization of a transmit sequence.
% There is a "pad" data section, which allows for the transmit amplifier to
% settle. This is then followed by a 65-bit training sequence. This is
% followed by the number of symbols beginning encoded into two bytes or
% 16-bits. This is then followed by a variable length data sequence and a
% CRC. All bits can optionally be differentially encoded.
%
% Key design pattern covered in this example:
% (1) Design illustrates the us of binary operands, such as bitxor
% (2) Shows how to properly segment persistent variables for register an
% BRAM access
% (3) Illustrates the use of fi math
% (4) Shows how to properly format and store ROM data, e.g., padData

%   Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [symbolOut, reByte] = ...
    mlhdlc_comms_data_packet(emptyFlag, byteValue, numberSymbols, diffOn, Nts, Npad)

persistent trainBits1 padData
persistent valueCRC crcVector bitPrev
persistent inPacketFlag  bitOfByteIndex symbolCount

fm = hdlfimath;
```

```
if isempty(symbolCount)
    symbolCount = 1;
    inPacketFlag = 0;
    valueCRC = fi(1, 0,16,0, fm);
    bitOfByteIndex = 1;
    bitPrev = fi(1, 0,1,0, fm);
    crcVector = zeros(1,16);
end
if isempty(trainBits1)
    % data-set already exists
    trainBits1 = TRAIN_DATA;
    padData = PAD_DATA;
end

%genPoly = 69665;
genPoly = fi(65535, 0,16,0, fm);
byteUint8 = uint8(byteValue);

reByte = 0;
symbolOut = fi(0, 0,1,0, fm);

%the first condition is whether or not we're currently processing a packet
if inPacketFlag == 1
    bitOut = fi(0, 0,1,0, fm);
    if symbolCount <= Npad
        bitOut(:) = padData(symbolCount);
    elseif symbolCount <= Npad+Nts
        bitOut(:) = trainBits1(symbolCount-Npad);
    elseif symbolCount <= Npad+Nts+numberSymbols
        bitOut(:) = bitget(byteUint8,9-bitOfByteIndex);
        bitOfByteIndex = bitOfByteIndex + 1;
        if bitOfByteIndex == 9 && symbolCount < Npad+Nts+numberSymbols
            bitOfByteIndex = 1;
            reByte = 1; % we've exhausted this one so pop new one off
        end
    elseif symbolCount <= Npad+Nts+numberSymbols+16
        bitOut(:) = 0;
    elseif symbolCount <= Npad+Nts+numberSymbols+32
        bitOut(:) = crcVector(symbolCount-(Npad+Nts+numberSymbols+16));
    else
        inPacketFlag = 0; %we're done
    end

    %leadValue = 0;
    % here we have the bit going out so if past Nts+Npad then form CRC.
    % Note that we throw 16 zeros on the end in order to flush the CRC
    if symbolCount > Npad+Nts && symbolCount <= Npad+Nts+numberSymbols+16

        valueCRCsh1 = bitsll(valueCRC, 1);
        valueCRCadd1 = bitor(valueCRCsh1, fi(bitOut, 0,16,0, fm));
        leadValue = bitget(valueCRCadd1,16);
        if leadValue == 1
            valueCRCxor = bitxor(valueCRCadd1, genPoly);
        else
            valueCRCxor = valueCRCadd1;
        end
        valueCRC = valueCRCxor;
        if symbolCount == Npad+Nts+numberSymbols+16
```

```
                crcVector(:) = bitget( valueCRC, 16:-1:1);
            end
        end

        if diffOn == 0 || symbolCount <= Npad+Nts
            symbolOut(:) = bitOut;
        else
            if bitPrev == bitOut
                symbolOut(:) = 1;
            else
                symbolOut(:) = 0;
            end
        end
        bitPrev(:) = symbolOut;

        symbolCount = symbolCount + 1; %total number of symbols transmitted
    else
        % we're not processing a packet and waiting for a new packet to arrive
        if emptyFlag == 0
            % reset everything
            inPacketFlag = 1;
            % toggle re to grab data
            reByte = 1;
            symbolCount = 1;
            bitOfByteIndex = 1;
            valueCRC(:) = 65535;
            bitPrev(:) = 0;
        end
    end

    end
```

```
type(testbench_name);
```

```
function mlhdlc_comms_data_packet_tb
%

%   Copyright 2011-2015 The MathWorks, Inc.

% generate transmit data, note the first two bytes are the data length
numberBytes = 8; % this is total number of symbols
numberSymbols = numberBytes*8;
rng(1); % always default to known state
data = [floor(numberBytes/2^8) mod(numberBytes,2^8) ...
    round(rand(1,numberBytes-2)*255)];

% generate training data helper function
make_train_data('TRAIN_DATA');

% make sure training data is generated
pause(2)
[~] = which('TRAIN_DATA');

trainBits1 = TRAIN_DATA;
Nts = length(trainBits1);

make_pad_data('PAD_DATA');
```

```
pause(2)
[~] = which('PAD_DATA');
Npad = 2^9;

% Give number of samples, where the start of the sequence flag will be
% (indicated by a zero), as well as an output buffer for generated symbols
Nsamp = 1000;
Noffset = 20;
emptyFlagHold = ones(1,Nsamp); emptyFlagHold(Noffset) = 0;
symbolOutHold = zeros(1,Nsamp);

dataIndex = 1;
byteValue = 0;
diffOn = 1; % 0 - regular encoding, 1 - differential encoding
for i1 = 1:Nsamp
    emptyFlag = emptyFlagHold(i1);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Call to the design
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    [symbolOut, reByte] = ...
        mlhdlc_comms_data_packet(emptyFlag, byteValue, numberSymbols, diffOn, Nts, Npad);

    % This set of code emulates the external FIFO interface
    if reByte == 1 % when high, pop a value off the input FIFO
        byteValue = data(dataIndex);
        dataIndex = dataIndex + 1;
    end
    symbolOutHold(i1) = symbolOut;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is all code to verify we did the encoding properly
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% grad training data - not differentially encoded
symbolTrain = symbolOutHold(1+Noffset+Npad:Noffset+Npad+Nts);

% grab user data and decode if necessary
symbolEst = zeros(1,numberSymbols);
symbolPrev = trainBits1(end);
if diffOn == 0
    symbolData = ...
        symbolOutHold(1+Noffset+Npad+Nts:Noffset+Npad+Nts+numberSymbols); %#ok<NASGU>
else
    % decoding is simply comparing adjacent received symbols
    symbolTemp = ...
       symbolOutHold(1+Noffset+Npad+Nts:Noffset+Npad+Nts+numberSymbols+32);
    for i1 = 1:length(symbolTemp)
        if symbolTemp(i1) == symbolPrev
            symbolEst(i1) = 1;
        else
            symbolEst(i1) = 0;
        end
        symbolPrev = symbolTemp(i1);
    end
end
```

```
% training data
trainDataEst = symbolTrain(1:Nts);
trainDiff = abs(trainDataEst-trainBits1');

% user data
userDataEst = symbolEst(1:numberSymbols);
dataEst = zeros(1,numberBytes);
for i1 = 1:numberBytes
    y = userDataEst((i1-1)*8+1:i1*8);
    dataEst(i1) = bin2dec(char(y+48));
end
userDiff = abs(dataEst-data);

disp(['Training Difference: ',num2str(sum(trainDiff)), ...
    ' User Data Difference: ',num2str(sum(userDiff))]);

% run it through and check CRC
genPoly = 69665;
c = symbolEst;
cEst = c(1,:);
cEst2 = [cEst(1:end-32) cEst(end-15:end)];
cEst = cEst2;

valueCRCc = 65535;
for i1 = 1:length(cEst)
    valueCRCsh1 = bitsll(uint16(valueCRCc), 1);
    valueCRCadd1 = bitor(uint16(valueCRCsh1), cEst(i1));
    leadValue = bitget( valueCRCadd1, 16);
    if (leadValue == 1)
        valueCRCxor = bitxor(uint16(valueCRCadd1), uint16(genPoly));
    else
        valueCRCxor = bitxor(uint16(valueCRCadd1), 0);
    end
    valueCRCc = valueCRCxor;
end
if valueCRCc == 0
    disp('CRC decoded correctly');
else
    disp('CRC check failed');
end

function make_train_data(filename)
x  = load('mlhdlc_dpack_train_data.txt');
fid = fopen([filename,'.m'],'w+');
fprintf(fid,['function y = ' filename '\n']);
fprintf(fid,'%%#codegen\n');
fprintf(fid,'y = [\n');
fprintf(fid,'%1.0e\n',x);
fprintf(fid,'];\n');
fclose(fid);

function make_pad_data(filename)
rng(1);
x = round(rand(1,2^9));
fid = fopen([filename,'.m'],'w+');
fprintf(fid,['function y = ' filename '\n']);
fprintf(fid,'%%#codegen\n');
fprintf(fid,'y = [\n');
```

```
fprintf(fid,'%1.0e\n',x);
fprintf(fid,'];\n');
fclose(fid);
```

Simulate the Design

Before code generation, simulate the design by using the test bench to make sure that there are no run-time errors.

```
mlhdlc_comms_data_packet_tb
```

```
Training Difference: 0 User Data Difference: 0
CRC decoded correctly
```

Create HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_dpack
```

Add the file `mlhdlc_comms_data_packet.m` to the project as the MATLAB Function and `mlhdlc_comms_data_packet_tb.m` as the MATLAB Test Bench.

For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14 or “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.

Run Fixed-Point Conversion and HLS Code Generation

To generate HLS code from a MATLAB design:

1. At the MATLAB command line, setup the path for HLS code generation by using the function `hdlsetuphlstoolpath`.

2. Start the Workflow Advisor by clicking the **Workflow Advisor** button.

3. In the **HDL Workflow Advisor**, select **Code Generation Workflow** as **MATLAB to HLS**.

4. Select **Cadence Stratus** as the **Synthesis tool** for **Select Code Generation Target**.

5. Right-click the **HLS Code Generation** and choose the option **Run to selected task** to run all the steps from the beginning through the HLS code generation.

Examine the generated HLS code by clicking the hyperlinks in the Code Generation Log window.

High-Level Synthesis Code Generation for DF2T Filter

You can generate High-Level Synthesis (HLS) code from MATLAB® design for direct-form II transposed filter.

MATLAB Design

Set up the `df2t_filter` model and test bench for this example.

```
mlhdlc_demo_setup('df2t_filter');

% Design Sqrt
design_name = 'mlhdlc_df2t_filter';

% Test Bench for Sqrt
testbench_name = 'mlhdlc_df2t_filter_tb';
```

Review the `df2t_filter` design

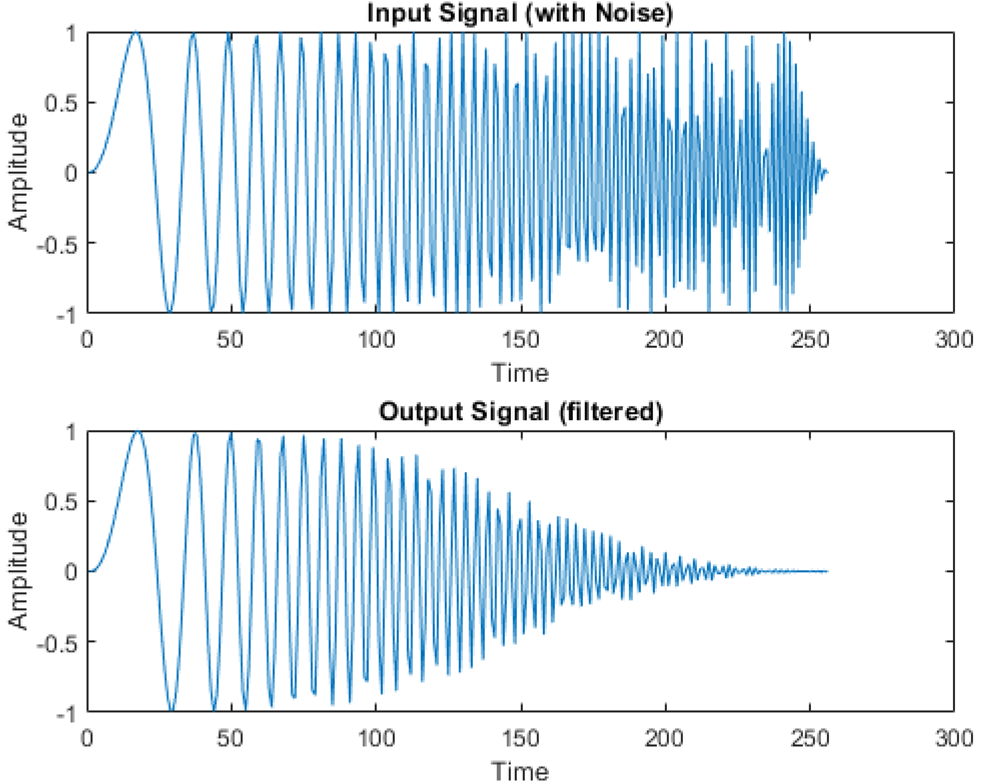
```
dbtype(design_name)
```

```
1     %#codegen
2     function y = mlhdlc_df2t_filter(x)
3
4     %   Copyright 2011-2015 The MathWorks, Inc.
5
6     persistent z;
7     if isempty(z)
8         % Filter states as a column vector
9         z = zeros(2,1);
10    end
11
12    % Filter coefficients as constants
13    b = [0.29290771484375   0.585784912109375  0.292907714843750];
14    a = [1.0                0.0                0.171600341796875];
15
16    y    =  b(1)*x + z(1);
17    z(1) = (b(2)*x + z(2)) - a(2) * y;
18    z(2) =  b(3)*x - a(3) * y;
19
20    end
```

Simulate the Design

It is a best practice to simulate the design with the test bench prior to code generation to check for run-time errors.

```
mlhdlc_df2t_filter_tb
```



Create a HDL Coder™ Project

Create an HDL Coder project.

```
coder -hdlcoder -new mlhdlc_df2t_prj
```

Next, add the file `mlhdlc_df2t_filter.m` to the project as the MATLAB Function and `mlhdlc_df2t_filter_tb.m` as the MATLAB Test Bench.

Run Fixed-Point Conversion and HLS Code Generation

To generate HLS code from the MATLAB design:

1. At the MATLAB command line, set up the path for HLS code generation by using the function `hdlsetuphlstoolpath`.

2. Start the Workflow Advisor by clicking the **Workflow Advisor** button.

3. In the **HDL Workflow Advisor**, select **Code Generation Workflow** as **MATLAB to HLS**.

4. Select **Cadence Stratus** as the **Synthesis tool** for **Select Code Generation Target**.

5. Right-click the **HLS Code Generation** task and select **Run to selected task** to run all the steps from the beginning through the HLS code generation.

Examine the generated HLS code by clicking the hyperlinks in the HLS Code Generation log window. For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.

See Also

- “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14
- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “MATLAB to HLS Code Generation Options” on page 12-4

High-Level Synthesis Code Generation for Transmit and Receive FIFO Registers

This example shows how to generate High-Level Synthesis (HLS) code from MATLAB® code that models the data transfer between a transmit and receive first-in, first-out (FIFO) register or buffer. This example contains two functions that represent a receive FIFO buffer and a transmit FIFO buffer, and a test bench `mlhdlc_fifo_tb` that simulates the data transfer that occurs between the two buffers. Each function is hardware-ready and exhibits good practices and guidelines to follow when writing MATLAB functions to generate efficient HLS code. For more information on guidelines to follow, see "Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code" on page 1-66.

View Example Functions and Test Bench

You can set up the model and open the MATLAB design for the transmit FIFO and the receive FIFO.

```
mlhdlc_demo_setup('rx_fifo');
type('mlhdlc_rx_fifo');
```

```
function [dout, empty, byte_ready, full, bytes_available] = ...
    mlhdlc_rx_fifo(get_byte, store_byte, byte_in, reset_fifo, fifo_enable)
%

%   Copyright 2014-2015 The MathWorks, Inc.

%
%  First In First Out (FIFO) structure.
%  This FIFO stores integers.
%  The FIFO is actually a circular buffer.
%
persistent head tail fifo byte_out handshake

if (reset_fifo || isempty(head))
    head = 1;
    tail = 2;
    byte_out = 0;
    handshake = 0;
end

if isempty(fifo)
    fifo = zeros(1,1024);
end

full = 0;
empty = 0;
byte_ready = 0;

% Section for checking full and empty cases
if ((tail == 1 && head == 1024) || ((head + 1) == tail))
    empty = 1;
end
if ((head == 1 && tail == 1024) || ((tail + 1) == head))
    full = 1;
end
```

```
% handshaking logic
if get_byte == 0
    handshake = 0;
end
if handshake == 1
    byte_ready = 1;
end

if (fifo_enable == 1)
    %%%%%%%%%%%%%%get%%%%%%%%%%%%%%%%%%%%%
    if (get_byte && ~empty && handshake == 0 )
        head = head + 1;
        if head == 1025
            head = 1;
        end
        byte_out = fifo(head);
        byte_ready = 1;
        handshake = 1;
    end
    %%%%%%%%%%%%%put%%%%%%%%%%%%%%%%%%%%%
    if (store_byte && ~full)
        fifo(tail) = byte_in;
        tail = tail + 1;
        if tail == 1025
            tail = 1;
        end
    end
end

% Section for calculating num bytes in FIFO
if (head < tail)
    bytes_available = (tail - head) - 1;
else
    bytes_available = (1024 - head) + tail - 1;
end

dout = byte_out;
end
```

```
mlhdlc_demo_setup('tx_fifo');
type('mlhdlc_tx_fifo');
```

```
function [dout, empty, byte_received, full, bytes_available, dbg_fifo_enable] = ...
    mlhdlc_tx_fifo(get_byte, store_byte, byte_in, reset_fifo, fifo_enable)
%

%   Copyright 2014-2015 The MathWorks, Inc.

%
%  First In First Out (FIFO) structure.
%  This FIFO stores integers.
%  The FIFO is actually a circular buffer.
%
persistent head tail fifo byte_out handshake

if (reset_fifo || isempty(head))
```

```
    head = 1;
    tail = 2;
    byte_out = 0;
    handshake = 0;
end

if isempty(fifo)
    fifo = zeros(1,1024);
end

full = 0;
empty = 0;
byte_received = 0;

% Section for checking full and empty cases
if ((tail == 1 && head == 1024) || ((head + 1) == tail))
    empty = 1;
end
if ((head == 1 && tail == 1024) || ((tail + 1) == head))
    full = 1;
end

% handshaking logic
if store_byte == 0
    handshake = 0;
end
if handshake == 1
    byte_received = 1;
end

if (fifo_enable == 1)
    %%%%%%%%%%%%%%get%%%%%%%%%%%%%%%%%%%%%
    if (get_byte && ~empty)
        head = head + 1;
        if head == 1025
            head = 1;
        end
        byte_out = fifo(head);
    end
    %%%%%%%%%%%%%put%%%%%%%%%%%%%%%%%%%%%
    if (store_byte && ~full && handshake == 0)
        fifo(tail) = byte_in;
        tail = tail + 1;
        if tail == 1025
            tail = 1;
        end
        byte_received = 1;
        handshake = 1;
    end
end

% Section for calculating num bytes in FIFO
if (head < tail)
    bytes_available = (tail - head) - 1;
else
    bytes_available = (1024 - head) + tail - 1;
end
```

```
dout = byte_out;
dbg_fifo_enable = fifo_enable;
end
```

Open the MATLAB design for the test bench that exercises both designs. This test bench test both the transmit and receive FIFOs. However, when generating HLS code, because you have individual functions for the transmit and receive FIFOs, you need individual test benches to test both functions and generate code. For simulation purposes, you can use `mlhdlc_fifo_tb`, but for HLS code generation, use the receive FIFO test bench `mlhdlc_rx_fifo_tb` with the receive FIFO function `mlhdlc_rx_fifo`, and use the transmit FIFO test bench `mlhdlc_tx_fifo` with the transmit FIFO function `mlhdlc_tx_fifo`.

```
type('mlhdlc_fifo_tb');
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simulation parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% data payload creation

%   Copyright 2014-2024 The MathWorks, Inc.

messageASCII = 'Hello World!';
message = double(unicode2native(messageASCII));
msgLength = length(message);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TX_FIFO core
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
numBytesToFifo = 1;
tx_get_byte = 0;
tx_full = 0;
tx_byte_received = 0;
i1 = 1;

while (numBytesToFifo <= msgLength && ~tx_full)
    % first thing the processor does is clear the internal tx fifo
    if i1 == 1
        tx_reset_fifo = 1;
        mlhdlc_tx_fifo(0, 0, 0, tx_reset_fifo, 1);
    else
        tx_reset_fifo = 0;
    end
    if (i1 > 1)
        tx_data_in = message(numBytesToFifo);
        numBytesToFifo = numBytesToFifo + 1;
        tx_store_byte = 1;
        while (tx_byte_received == 0)
            [tx_data_out, tx_empty, tx_byte_received, tx_full, tx_bytes_available] = ...
                mlhdlc_tx_fifo(tx_get_byte, tx_store_byte, tx_data_in, tx_reset_fifo, 1);
        end
        tx_store_byte = 0;
        while (tx_byte_received == 1)
            [tx_data_out, tx_empty, tx_byte_received, tx_full, tx_bytes_available] = ...
                mlhdlc_tx_fifo(tx_get_byte, tx_store_byte, tx_data_in, tx_reset_fifo, 1);
        end
    end
    i1 = i1 + 1;
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Transfer Bytes from TX FIFO to RX FIFO
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
i1 = 1;

tx_get_byte = 0;
tx_store_byte = 0;
tx_data_in = 0;
tx_reset_fifo = 0;

rx_get_byte = 0;
rx_data_in = 0;
rx_reset_fifo = 0;

while (tx_bytes_available > 0)
        if i1 == 1
            rx_reset_fifo = 1;
            mlhdlc_rx_fifo(0, 0, 0, rx_reset_fifo, 1);
        else
            rx_reset_fifo = 0;
        end
        if (i1 > 1)
            tx_get_byte = 1;
            rx_store_byte = 1;
            [tx_data_out, tx_empty, tx_byte_received, tx_full, tx_bytes_available] = ...
                mlhdlc_tx_fifo(tx_get_byte, tx_store_byte, tx_data_in, tx_reset_fifo, 1);

            rx_data_in = tx_data_out;

            [rx_data_out, rx_empty, rx_byte_ready, rx_full, rx_bytes_available] = ...
                mlhdlc_rx_fifo(rx_get_byte, rx_store_byte, rx_data_in, rx_reset_fifo, 1);
        end
        i1 = i1 + 1;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RX_FIFO core
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
numBytesFromFifo = 1;
rx_store_byte = 0;
rx_byte_received = 0;
i1 = 1;
msgBytes = zeros(1,msgLength);

while (~rx_empty)
    % first thing the processor does is clear the internal rx fifo
    if (i1 > 1)
        rx_get_byte = 1;
        while (rx_byte_ready == 0)
            [rx_data_out, rx_empty, rx_byte_ready, rx_full, rx_bytes_available] = ...
                mlhdlc_rx_fifo(rx_get_byte, rx_store_byte, rx_data_in, rx_reset_fifo, 1);
        end
        msgBytes(i1-1) = rx_data_out;
        rx_get_byte = 0;
        while (rx_byte_ready == 1)
            [rx_data_out, rx_empty, rx_byte_ready, rx_full, rx_bytes_available] = ...
                mlhdlc_rx_fifo(rx_get_byte, rx_store_byte, rx_data_in, rx_reset_fifo, 1);
        end
    end
```

```
    i1 = i1 + 1;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

numRecBytes = numBytesFromFifo;
if sum(msgBytes-message) == 0
    disp('Received message correctly');
else
    disp('Received message incorrectly');
end
native2unicode(msgBytes)
```

Simulate the Design

Simulate the design with the test bench before generating code to ensure there are no runtime errors.

```
mlhdlc_fifo_tb
```

```
Received message correctly

ans =

    'Hello World!'
```

Create HDL Coder™ Project for the Receive FIFO

At the MATLAB command line, set up the path for HLS code generation by using the function `hdlsetuphlstoolpath`.

Create a HDL Coder project for the receive FIFO.

```
coder -hdlcoder -new mlhdlc_rx_fifo
```

Add the file `mlhdlc_rx_fifo.m` to the project as the MATLAB Function and `mlhdlc_rx_fifo_tb.m` as the MATLAB Test Bench and then click **Workflow Advisor**.

Generate HLS Code for the Receive FIFO

In the **HDL Workflow Advisor**, select **MATLAB to HLS** as the **Code Generation Workflow** and **Cadence Stratus** as the **Synthesis tool** in **Select Code Generation Target** step.

Run all the steps from the beginning through the HLS code generation. Examine the generated HLS code for the receive FIFO by clicking the hyperlinks in the bottom pane.

For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.

Create HDL Coder Project for the Transmit FIFO

At the MATLAB command line, set up the path for HLS code generation by using the function `hdlsetuphlstoolpath`.

Create a HDL Coder project.

```
coder -hdlcoder -new mlhdlc_tx_fifo
```

Add `mlhdlc_tx_fifo.m` to the project as the MATLAB Function and `mlhdlc_tx_fifo_tb.m` as the MATLAB Test Bench and then click **Workflow Advisor**.

Generate HLS Code for the Transmit FIFO

In the **HDL Workflow Advisor**, select **MATLAB to HLS** as the **Code Generation Workflow** and **Cadence Stratus** as the **Synthesis tool** in **Select Code Generation Target** step.

Run all the steps from the beginning through the HLS code generation. Examine the generated HLS code for the transmit FIFO by clicking on the hyperlinks in the bottom pane.

See Also

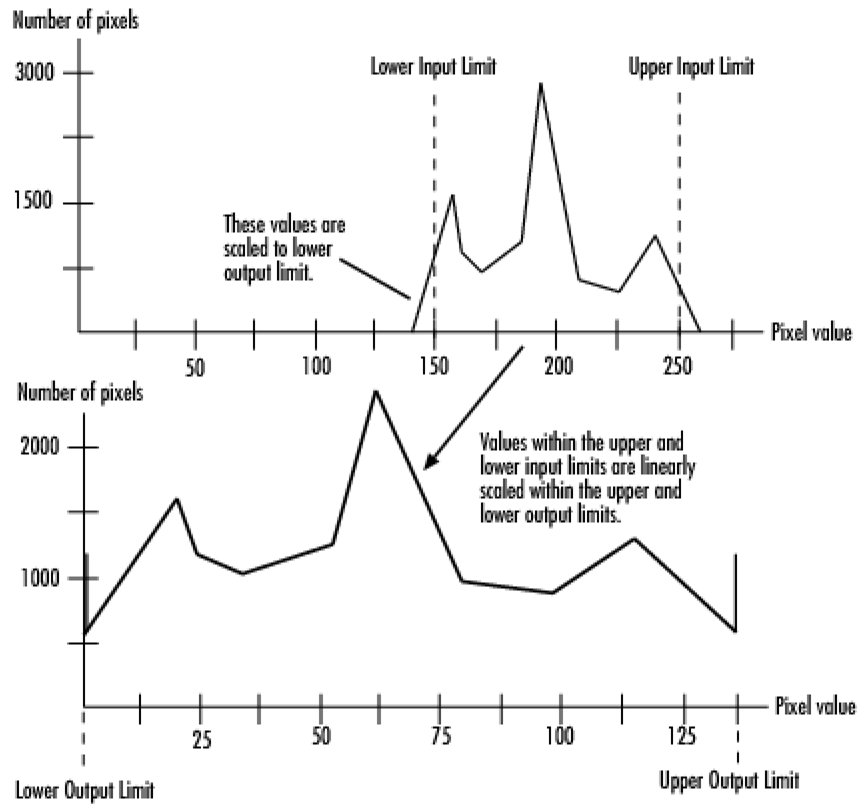
- “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “MATLAB to HLS Code Generation Options” on page 12-4

High-Level Synthesis Code Generation for Contrast Adjustment

This example shows how to generate High-Level Synthesis (HLS) code from a MATLAB® design that adjusts image contrast by linearly scaling pixel values.

Algorithm

Contrast adjustment adjusts the contrast of an image by linearly scaling the pixel values between upper and lower limits. Pixel values that are above or below this range are saturated to the upper or lower limit value, respectively.



MATLAB Design

```
design_name = 'mlhdlc_image_scale';
testbench_name = 'mlhdlc_image_scale_tb';
```

Review the MATLAB design:

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% scale.m
%
% Adjust image contrast by linearly scaling pixel values.
%
```

```
% The input pixel value range has 14bits and output pixel value range is
% 8bits.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out, y_out, pixel_out] = ...
    mlhdlc_image_scale(x_in, y_in, pixel_in, ...
          damping_factor_in, dynamic_range_in, ...
          tail_size_in, max_gain_in, ...
          width, height)

%   Copyright 2011-2022 The MathWorks, Inc.

persistent histogram1 histogram2
persistent low_count
persistent high_count
persistent offset
persistent gain
persistent limits_done
persistent damping_done
persistent reset_hist_done
persistent scaling_done
persistent hist_ind
persistent tail_high
persistent min_hist_damped    %Damped lower limit of populated histogram
persistent max_hist_damped    %Damped upper limit of populated histogram
persistent found_high
persistent found_low

DR_PER_BIN          = 8;
SF                  = 1./(1:(2^14/8)); % be nice to fix this
NR_OF_BINS          = (2^14/DR_PER_BIN) - 1;
MAX_DF              = 255;

if isempty(offset)
    offset              = 1;
    gain                = 1;
    limits_done         = 1;
    damping_done        = 1;
    reset_hist_done     = 1;
    scaling_done        = 1;
    hist_ind            = 1;
    tail_high           = NR_OF_BINS;
    low_count             = 0;
    high_count             = 0;
    min_hist_damped     = 0;
    max_hist_damped     = (2^14/DR_PER_BIN) - 1;
    found_high          = 0;
    found_low           = 0;
end
if isempty(histogram1)
    histogram1          = zeros(1, NR_OF_BINS+1);
    histogram2          = zeros(1, NR_OF_BINS+1);
end

if y_in < height
    frame_valid = 1;
    if x_in < width
        line_valid = 1;
```

```
    else
        line_valid = 0;
    end
else
    frame_valid = 0;
    line_valid = 0;
end

% initialize at beginning of frame
if x_in == 0 && y_in == 0
    limits_done = 0;
    damping_done = 0;
    reset_hist_done = 0;
    scaling_done = 0;
    low_count = 0;
    high_count = 0;
    hist_ind = 1;
end

max_gain_frac = max_gain_in/2^4;
pix11 = floor(pixel_in/DR_PER_BIN);
pix_out_temp = pixel_in;

%***************************************************************************
%Check if valid part of frame. If pixel is valid remap pixel to desired
%output dynamic range (dynamic_range_in) by subtracting the damped offset
%(min_hist_damped) and applying the calculated gain calculated from the
%previous frame histogram statistics.
%***************************************************************************
% histogram read
histReadIndex1 = 1;
histReadIndex2 = 1;
if frame_valid && line_valid
    histReadIndex1 = pix11+1;
    histReadIndex2 = pix11+1;
elseif ~limits_done
    histReadIndex1 = hist_ind;
    histReadIndex2 = NR_OF_BINS - hist_ind;
end
histReadValue1 = histogram1(histReadIndex1);
histReadValue2 = histogram2(histReadIndex2);
histWriteIndex1 = NR_OF_BINS+1;
histWriteIndex2 = NR_OF_BINS+1;
histWriteValue1 = 0;
histWriteValue2 = 0;
if frame_valid
    if line_valid
        temp_sum = histReadValue1 + 1;
        ind = min(pix11+1, NR_OF_BINS);
        val = min(temp_sum, tail_size_in);
        histWriteIndex1 = ind;
        histWriteValue1 = val;
        histWriteIndex2 = ind;
        histWriteValue2 = val;

        %Scale pixel
        pix_out_offs_corr = pixel_in - min_hist_damped*DR_PER_BIN;
        pix_out_scaled = pix_out_offs_corr * gain;
```

```
        pix_out_clamp = max(min(dynamic_range_in, pix_out_scaled), 0);
        pix_out_temp = pix_out_clamp;
    end
else
    %*****************************************************************
    %Ignore tail_size_in pixels and find lower and upper limits of the
    %histogram.
    %*****************************************************************
    if ~limits_done
        if hist_ind == 1
            tail_high = NR_OF_BINS-1;
            offset = 1;
            found_high = 0;
            found_low = 0;
        end

        low_count = low_count + histReadValue1;
        hist_ind_high = NR_OF_BINS - hist_ind;
        high_count = high_count + histReadValue2;

        %Found enough high outliers
        if high_count > tail_size_in && ~found_high
            tail_high = hist_ind_high;
            found_high = 1;
        end

        %Found enough low outliers
        if low_count > tail_size_in && ~found_low
            offset = hist_ind;
            found_low = 1;
        end

        hist_ind = hist_ind + 1;
        %All bins checked so limits must already be found
        if hist_ind >= NR_OF_BINS
            hist_ind = 1;
            limits_done = 1;
        end
        %*****************************************************************
        %Damp the limit change to avoid image flickering. Code below equivalent
        %to: max_hist_damped = damping_factor_in*max_hist_dampedOld +
        %(1-damping_factor_in)*max_hist_dampedNew;
        %*****************************************************************
    elseif ~damping_done
        min_hist_weighted_old = damping_factor_in*min_hist_damped;
        min_hist_weighted_new = (MAX_DF-damping_factor_in+1)*offset;
        min_hist_weighted = (min_hist_weighted_old + ...
            min_hist_weighted_new)/256;
        min_hist_damped = max(0, min_hist_weighted);
        max_hist_weighted_old = damping_factor_in*max_hist_damped;
        max_hist_weighted_new = (MAX_DF-damping_factor_in+1)*tail_high;
        max_hist_weighted = (max_hist_weighted_old + ...
            max_hist_weighted_new)/256;
        max_hist_damped = min(NR_OF_BINS, max_hist_weighted);
        damping_done = 1;
        hist_ind = 1;
        %*****************************************************************
        %Reset all bins to zero. More than one bin can be reset per function
```

```
        %call if blanking time is too short.
        %*********************************************************************
    elseif ~reset_hist_done
        histWriteIndex1 = hist_ind;
        histWriteValue1 = 0;
        histWriteIndex2 = hist_ind;
        histWriteValue2 = 0;
        hist_ind = hist_ind+1;
        if hist_ind == NR_OF_BINS
            reset_hist_done = 1;
        end
        %*********************************************************************
        %The gain factor is determined by comparing the measured damped actual
        %dynamic range to the desired user specified dynamic range. Input
        %dynamic range is measured in bins over DR_PER_BIN space.
        %*********************************************************************
    elseif ~scaling_done
        dr_in = round(max_hist_damped - min_hist_damped);
        gain_temp = dynamic_range_in*SF(dr_in);
        gain_scaled = gain_temp/DR_PER_BIN;
        gain = min(max_gain_frac, gain_scaled);
        scaling_done = 1;
        hist_ind = 1;
    end
end
histogram1(histWriteIndex1) = histWriteValue1;
histogram2(histWriteIndex2) = histWriteValue2;

x_out = x_in;
y_out = y_in;
pixel_out = pix_out_temp;
```

```
type(testbench_name);
```

```
%Test bench for scaling, analogous to automatic gain control (AGC)

%   Copyright 2011-2022 The MathWorks, Inc.

testFile = 'mlhdlc_img_peppers.png';
imgOrig = imread(testFile);
[height, width] = size(imgOrig);
imgOut = zeros(height,width);
hBlank = 20;
% make sure we have enough vertical blanking to filter the histogram
vBlank = ceil(2^14/(width+hBlank));

%df - Temporal damping factor of rescaling
%dr - Desired output dynamic range
df = 0;
dr = 255;
nrOfOutliers = 248;
maxGain = 2*2^4;

for frame = 1:2
    disp(['frame: ', num2str(frame)]);
    for y_in = 0:height+vBlank-1
        %disp(['frame: ', num2str(frame), ' of 2, row: ', num2str(y_in)]);
```

```
        for x_in = 0:width+hBlank-1
            if x_in < width && y_in < height
                pixel_in = double(imgOrig(y_in+1, x_in+1));
            else
                pixel_in = 0;
            end

            [x_out, y_out, pixel_out] = ...
                mlhdlc_image_scale(x_in, y_in, pixel_in, df, dr, ...
                    nrOfOutliers, maxGain, width, height);

            if x_out < width && y_out < height
                imgOut(y_out+1,x_out+1) = pixel_out;
            end
        end
    end

    figure('Name', [mfilename, '_scale_plot']);
    imgOut = round(255*imgOut/max(max(imgOut)));
    subplot(2,2,1); imshow(imgOrig, []);
    title('Original Image');
    subplot(2,2,2); imshow(imgOut, []);
    title('Scaled Image');
    subplot(2,2,3); histogram(double(imgOrig(:)),2^14-1);
    axis([0, 255, 0, 1500]);
    title('Histogram of original Image');
    subplot(2,2,4); histogram(double(imgOut(:)),2^14-1);
    axis([0, 255, 0, 1500]);
    title('Histogram of equalized Image');
end
```

Simulate the Design

It is a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

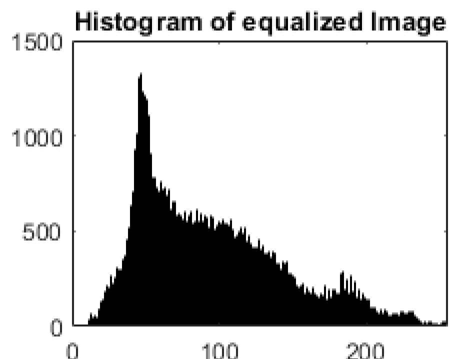
```
mlhdlc_image_scale_tb
```

```
frame: 1
frame: 2
```

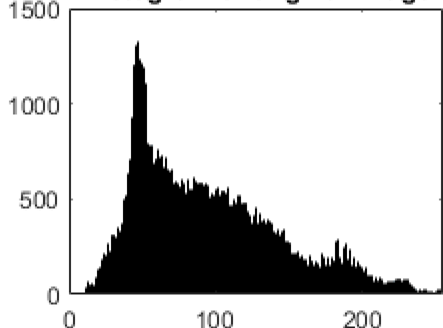
Original Image

Scaled Image

Histogram of original Image
1500
1000
500
0
0
100
200


Histogram of equalized Image
1500
1000
500
0
0
100
200

Original Image



Histogram of equalized Image
1500
1000
500
0
0
100
200

Create HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_scale_prj
```

Add the file `mlhdlc_image_scale.m` to the project as the MATLAB Function and `mlhdlc_image_scale_tb.m` as the MATLAB Test Bench.

For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14 or “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.

Run Fixed-Point Conversion and HLS Code Generation

To generate HLS code from a MATLAB design:

1. At the MATLAB command line, setup the path for HLS code generation by using the function `hdlsetuphlstoolpath`.

2. Start the Workflow Advisor by clicking the **Workflow Advisor** button.

3. In the **HDL Workflow Advisor**, select **Code Generation Workflow** as **MATLAB to HLS**.

4. Select **Cadence Stratus** as the **Synthesis tool** for **Select Code Generation Target**.

5. Right-click the **HLS Code Generation** and choose the option **Run to selected task** to run all the steps from the beginning through the HLS code generation.

Examine the generated HLS code by clicking the hyperlinks in the Code Generation Log window.

High-Level Synthesis Code Generation for Image Format Conversion from RGB to YUV

This example shows how to generate High-Level Synthesis (HLS) code from a MATLAB® design that converts the image format from RGB to YUV.

MATLAB Design and Test Bench

Set up the `rgb2yuv` model for this example.

```
design_name = 'mlhdlc_rgb2yuv';
testbench_name = 'mlhdlc_rgb2yuv_tb';
```

Review the `rgb2yuv` design.

```
open(design_name)
```

```
function [x_out, y_out, y_data_out, u_data_out, v_data_out] = ...
                        mlhdlc_rgb2yuv(x_in, y_in, r_in, g_in, b_in)
%#codegen

%   Copyright 2011-2019 The MathWorks, Inc.

persistent RGB_Reg YUV_Reg
persistent x1 x2 y1 y2

if isempty(RGB_Reg)
    RGB_Reg = zeros(3,1);
    YUV_Reg = zeros(3,1);
    x1 = 0; x2 = 0; y1 = 0; y2 = 0;
end

D = [.299  .587  .144; -.147 -.289  .436; .615 -.515 -.1];
C = [0; 128; 128];

RGB = [r_in; g_in; b_in];

YUV_1 = D*RGB_Reg;
YUV_2 = YUV_1 + C;
RGB_Reg = RGB;

y_data_out = round(YUV_Reg(1));
u_data_out = round(YUV_Reg(2));
v_data_out = round(YUV_Reg(3));
YUV_Reg = YUV_2;

x_out = x2; x2 = x1; x1 = x_in;
y_out = y2; y2 = y1; y1 = y_in;
```

Review the `rgb2yuv` test bench:

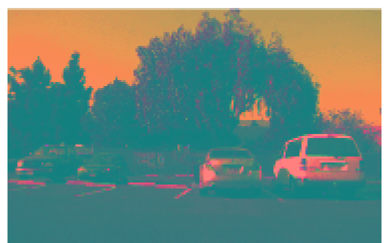
```
open(testbench_name);
```

```
FRAMES = 1;
WIDTH = 752;
HEIGHT = 480;
HBLANK = 10;%748;
VBLANK = 10;%120;

%   Copyright 2011-2019 The MathWorks, Inc.

vidData = double(imread('mlhdlc_img_yuv.png'));

for f = 1:FRAMES
    vidOut = zeros(HEIGHT, WIDTH, 3);

    for y = 0:HEIGHT+VBLANK-1
        for x = 0:WIDTH+HBLANK-1
            if y >= 0 && y < HEIGHT && x >= 0 && x < WIDTH
                b = vidData(y+1,x+1,1);
                g = vidData(y+1,x+1,2);
                r = vidData(y+1,x+1,3);
            else
                b = 0; g = 0; r = 0;
            end

            [xOut, yOut, yData, uData, vData] = ...
                mlhdlc_rgb2yuv(x, y, r, g, b);

            if yOut >= 0 && yOut < HEIGHT && xOut >= 0 && xOut < WIDTH
                vidOut(yOut+1,xOut+1,:) = [yData vData uData];
            end
        end
    end

    figure(1);
    subplot(1,2,1);
    imshow(uint8(vidData));
    subplot(1,2,2);
    imshow(ycbcr2rgb(uint8(vidOut)));
    drawnow;
end
```

Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

```
mlhdlc_rgb2yuv_tb
```



Create HDL Coder™ Project

To generate HLS code from a MATLAB design:

1. Create a HDL Coder project:

```
coder -hdlcoder -new mlhdlc_rgb_prj
```

2. Add the file `mlhdlc_rgb2yuv.m` to the project as the **MATLAB Function** and `mlhdlc_rgb2yuv_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefine types** to use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_rgb2yuv`.

For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14 or “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.

Run Fixed-Point Conversion and HLS Code Generation

To generate HLS code from the MATLAB design:

1. At the MATLAB command line, set up the path for HLS code generation by using the function `hdlsetuphlstoolpath`.

2. Start the Workflow Advisor by clicking the **Workflow Advisor** button.

3. In the **HDL Workflow Advisor**, select **Code Generation Workflow** as **MATLAB to HLS**.

4. In the **Select Code Generation Target** step, from the **Synthesis tool** list, select Cadence Stratus.

5. Right-click the **HLS Code Generation** task and select **Run to selected task** to run all the steps from the beginning through the HLS code generation.

Examine the generated HLS code by clicking the hyperlinks in the **HLS Code Generation** task log window.

See Also

- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “MATLAB to HLS Code Generation Options” on page 12-4

High-Level Synthesis Code Generation for Advanced Encryption Standard

This example shows how to generate High-Level Synthesis (HLS) code from MATLAB® code that implements an AES (Advanced Encryption Standard) algorithm.

AES MATLAB Design

AES is a symmetric block cipher that encrypts and decrypts confidential data using the same cipher key. You can design complicated algorithms such as AES using MATLAB code and then generate their equivalent HLS code.

In this example, the `mlhdlc_aes` function implements the AES encryption algorithm. This function accepts plain text to encrypt and a cipher key as inputs and outputs encrypted text. The `mlhdlc_aesd` function decrypts the encrypted text to plain text.

Set up the AES model for this example.

```
mlhdlc_demo_setup('aes');
```

To review the MATLAB code for encryption and decryption of text, open the files `mlhdlc_aes.m` and `mlhdlc_aesd.m`.

```
open('mlhdlc_aes');
open('mlhdlc_aesd');
```

You can also review the MATLAB test bench file.

```
open('mlhdlc_aes_tb')
```

Test the MATLAB Algorithm

To avoid run-time errors, simulate the design by using the test bench.

```
mlhdlc_aes_tb
```

```
1
Plain Text: D1 E8 21 EA A2 19 47 8C F5 F7 28 F8 F5 7C CD 24
Cipher Key: 6C EA CB F6 A8 09 D9 EF AE C2 BE 64 A8 2C B5 08
Cipher Text: 1F 9D 05 17 7B B0 5F 87 99 7A AE F3 9E 82 51 CC
Decrypted Plain Text:D1 E8 21 EA A2 19 47 8C F5 F7 28 F8 F5 7C CD 24
!!!!!Decrypted plain text matches the original text.!!!!!
2
Plain Text: 47 0C 19 D3 B2 51 F3 09 70 62 C4 CC 30 7D 72 A5
Cipher Key: B6 C1 47 AE A8 2A 1E 80 F6 57 96 39 C0 41 82 B3
Cipher Text: 2D CF CD F1 30 B8 A0 34 BB B1 BD 26 0B 0C 9D A0
Decrypted Plain Text:47 0C 19 D3 B2 51 F3 09 70 62 C4 CC 30 7D 72 A5
!!!!!Decrypted plain text matches the original text.!!!!!
3
Plain Text: E4 F6 8C 23 26 42 D7 41 D0 3E EE 5A 32 40 9E 79
Cipher Key: 5A D5 96 8D EB 49 C2 C1 61 91 13 0E 88 C7 EF 21
Cipher Text: 65 76 0D 70 C8 3A 59 D9 FD AD F0 E4 B5 61 3B 28
Decrypted Plain Text:E4 F6 8C 23 26 42 D7 41 D0 3E EE 5A 32 40 9E 79
!!!!!Decrypted plain text matches the original text.!!!!!
4
```

```
Plain Text: 92 78 03 56 2A CB 50 87 2A 9A 43 A7 B0 C0 73 15
Cipher Key: 3B EA 27 D3 8A FF 14 71 1B F6 01 C6 D1 DE 16 66
Cipher Text: 9B 77 32 8F 14 FD A5 F7 BB 2B 5B 45 CD 87 59 E5
Decrypted Plain Text:92 78 03 56 2A CB 50 87 2A 9A 43 A7 B0 C0 73 15
!!!!!Decrypted plain text matches the original text.!!!!!
5
Plain Text: 43 CD 6E E9 2F 44 25 23 DF 94 8D 25 DA 9F 5A 83
Cipher Key: 67 13 3D 20 2F 3D 6B 0D E7 F2 7E 7D 56 E6 5F 1C
Cipher Text: 96 9C D7 D1 FB 3C 26 B4 31 0F 7C 64 1A DE 4B BB
Decrypted Plain Text:43 CD 6E E9 2F 44 25 23 DF 94 8D 25 DA 9F 5A 83
!!!!!Decrypted plain text matches the original text.!!!!!
```

Generate HLS Code by Using HDL Workflow Advisor

HLS code generation with the HDL Workflow Advisor has the following basic steps:

1. At the MATLAB command line, set up the HLS tool path for HLS code generation by using the function `hdlsetuphlstoolpath`.
2. Create a HDL Coder project by adding the `mlhdlc_aes.m` design file and `mlhdlc_aes_tb.m` test bench file to the HDL workflow advisor.
3. Launch the HDL Workflow Advisor to generate HLS code. Do not run floating point to fixed point conversion on this design, as the AES MATLAB code is already in fixed point and suitable for HLS code generation.

For detailed information about these steps, see “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14 or “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.

See Also

- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4

High-Level Synthesis Code Generation

- “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5
- “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14
- “Replace Arithmetic Operation to Generate Efficient HDL and High-Level Synthesis Code” on page 11-18
- “HLS Code Generation Report” on page 11-29

coder.inline

Namespace: `coder`

Control inlining of current function in generated code

Syntax

```
coder.inline("always")
coder.inline("never")
```

Description

`coder.inline("always")` inlines the body of the function in which it is used, directly into the calling locations in the generated code.

The `coder.inline("always")` directive does not support the inlining of:

- Entry-point functions
- Recursive functions
- Functions that contain `parfor`-loops
- Functions called from `parfor`-loops

`coder.inline("never")` prevents inlining of the function in which it is used in the generated code. Use the `coder.inline("never")` optimization directive when you want to simplify the mapping between the MATLAB source code and the generated code.

The `coder.inline("never")` directive does not prevent the inlining of:

- Empty functions
- Functions that return constant output

To prevent inlining even in these situations, use the `coder.ignoreConst` function on an input at the function call site in your MATLAB code. For more information, see "Resolve Issue: coder.inline("never") and coder.nonInlineCall Do Not Prevent Function Inlining".

Examples

Use `coder.inline` in Flow-Control Statements

You can use `coder.inline` in flow-control code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

The `inline_division` function controls inlining based on whether it performs scalar division or vector division:

```
function out = inline_example(a)
  if (a > 10)
```

```
    out = inline_division(a,2);
  else
    out = 5;
  end
end

function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
   coder.inline("always");
else
% Vector division produces a for-loop.
% Prohibit inlining to reduce code size.
   coder.inline("never");
end

if any(divisor == 0)
   error("Cannot divide by 0");
end

y = dividend / divisor;
```

Prevent Function Inlining

Use `coder.inline("never")` in a function to prevent inlining in the generated code. In the generated code, calls to the `subfcn1` function remains as function calls.

```
function out = inline_example(a)
  if (a > 10)
    out = subfcn1(a);
  else
    out = 5;
  end
end

function out = subfcn1(a)
  coder.inline("never");
  out = a + 10;
end
```

Version History

Introduced in R2011a

R2024a: Support for High-Level Synthesis Code Generation

You can inline MATLAB functions in the generated HLS code.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

HDL Code Generation
Generate VHDL, Verilog and SystemVerilog code for FPGA and ASIC designs using HDL Coder™.

The `coder.inline` function supports MATLAB to High-Level Synthesis code generation.

See Also

Topics
"Resolve Issue: coder.inline("never") and coder.nonInlineCall Do Not Prevent Function Inlining"
"Generate Instantiable Code for Functions" on page 5-12

Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App

This example shows how to create an HDL Coder™ project and generate SystemC code This example shows how to create an HDL Coder™ project and generate SystemC® code This example shows how to create an HDL Coder™ project and generate High-Level Synthesis (HLS) code from a MATLAB® design for a symmetric finite impulse response (FIR) filter.

Set up the FIR model and test bench for this example.

```
mlhdlc_demo_setup('sfir');
```

FIR Filter MATLAB Design

The MATLAB design `mlhdlc_sfir` is a simple symmetric FIR filter. Ensure that the file path includes no spaces.

```
design_name = 'mlhdlc_sfir';
testbench_name = 'mlhdlc_sfir_tb';
```

Review the design.

```
open(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
%
% Introduction:
%
% We can reduce the complexity of the FIR filter by leveraging its symmetry.
% Symmetry for an n-tap filter implies, coefficient h0 = coefficient hn-1,
% coefficient, h1 = coefficient hn-2, etc. In this case, the number of
% multipliers can be approximately halved. The key is to add the
% two data values that need to be multiplied with the same coefficient
% prior to performing the multiplication.
%
% Key Design pattern covered in this example:
% (1) Filter states represented using the persistent variables
% (2) Filter coefficients passed in as parameters

%   Copyright 2011-2019 The MathWorks, Inc.

%#codegen
function [y_out, delayed_xout] = mlhdlc_sfir(x_in,h_in1,h_in2,h_in3,h_in4)
% Symmetric FIR Filter

% declare and initialize the delay registers
persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
if isempty(ud1)
    ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
end

% access the previous value of states/registers
a1 = ud1 + ud8; a2 = ud2 + ud7;
```

```
a3 = ud3 + ud6; a4 = ud4 + ud5;

% multiplier chain
m1 = h_in1 * a1; m2 = h_in2 * a2;
m3 = h_in3 * a3; m4 = h_in4 * a4;

% adder chain
a5 = m1 + m2; a6 = m3 + m4;

% filtered output
y_out = a5 + a6;

% delayout input signal
delayed_xout = ud8;

% update the delay line
ud8 = ud7;
ud7 = ud6;
ud6 = ud5;
ud5 = ud4;
ud4 = ud3;
ud3 = ud2;
ud2 = ud1;
ud1 = x_in;
end
```

FIR Filter MATLAB Test Bench

The MATLAB test bench `mlhdlc_sfir_tb` tests the filter design. Review this test bench.

```
open(testbench_name);
```

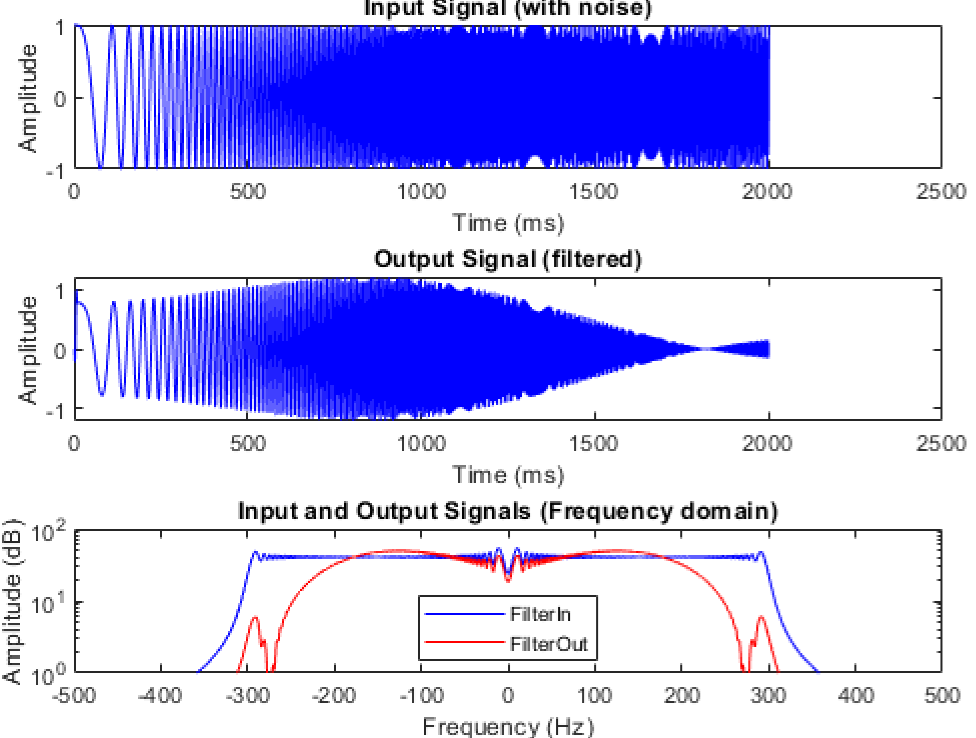
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB test bench for the FIR filter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2019 The MathWorks, Inc.
clear mlhdlc_sfir;
T = 2;
dt = 0.001;
N = T/dt+1;
sample_time = 0:dt:T;

df = 1/dt;
sample_freq = linspace(-1/2,1/2,N).*df;

% input signal with noise
x_in = cos(2.*pi.*(sample_time).*(1+(sample_time).*75)).';

% filter coefficients
h1 = -0.1339; h2 = -0.0838; h3 = 0.2026; h4 = 0.4064;

len = length(x_in);
y_out = zeros(1,len);
x_out = zeros(1,len);
```

```
for ii=1:len
    data = x_in(ii);
    % call to the design 'mlhdlc_sfir' that is targeted for hardware
    [y_out(ii), x_out(ii)] = mlhdlc_sfir(data, h1, h2, h3, h4);
end

figure('Name', [mfilename, '_plot']);
subplot(3,1,1);
plot(1:len,x_in,'-b');
xlabel('Time (ms)')

ylabel('Amplitude')
title('Input Signal (with noise)')
subplot(3,1,2); plot(1:len,y_out,'-b');
xlabel('Time (ms)')
ylabel('Amplitude')
title('Output Signal (filtered)')

freq_fft = @(x) abs(fftshift(fft(x)));

subplot(3,1,3); semilogy(sample_freq,freq_fft(x_in),'-b');
hold on
semilogy(sample_freq,freq_fft(y_out),'-r')
hold off
xlabel('Frequency (Hz)')
ylabel('Amplitude (dB)')
title('Input and Output Signals (Frequency domain)')
legend({'FilterIn', 'FilterOut'}, 'Location','South')
axis([-500 500 1 100])
```

Test MATLAB Algorithm

To avoid run-time errors, simulate the design by using the test bench.

```
mlhdlc_sfir_tb
```



Generate HLS Code by Using HDL Workflow Advisor

HLS code generation with the HDL Workflow Advisor has the following basic steps:

1. At the MATLAB command line, set up the high-level synthesis (HLS) tool path for HLS code generation by using the function `hdlsetuphlstoolpath`.
2. Create a MATLAB HDL Coder project.
3. Add the design and test bench files to the project.
4. Start the HDL Workflow Advisor for the MATLAB design.
5. Run fixed-point conversion and HLS code generation.

Create HDL Coder Project and Assign Files

To create an HDL Coder project:

1. In the MATLAB Editor, on the **Apps** tab, select **HDL Coder**. Enter `sfir_project` as the **Name** of the project.

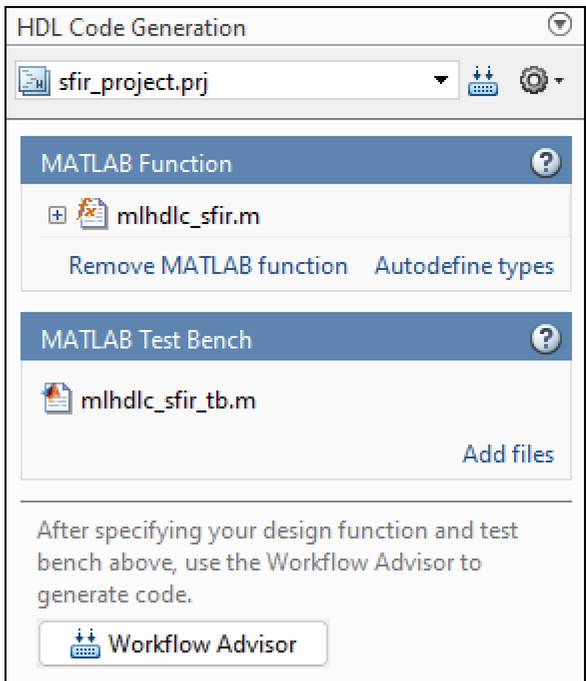
To create a project at the MATLAB command prompt, run the command:

```
coder -hdlcoder -new sfir_project
```

A `sfir_project.prj` file is created in the current folder.

2. For **MATLAB Function**, click the **Add MATLAB function** link and select the FIR filter MATLAB design `mlhdlc_sfir`. Under the **MATLAB Test Bench** section, click **Add files** and add the MATLAB test bench `mlhdlc_sfir_tb.m`.

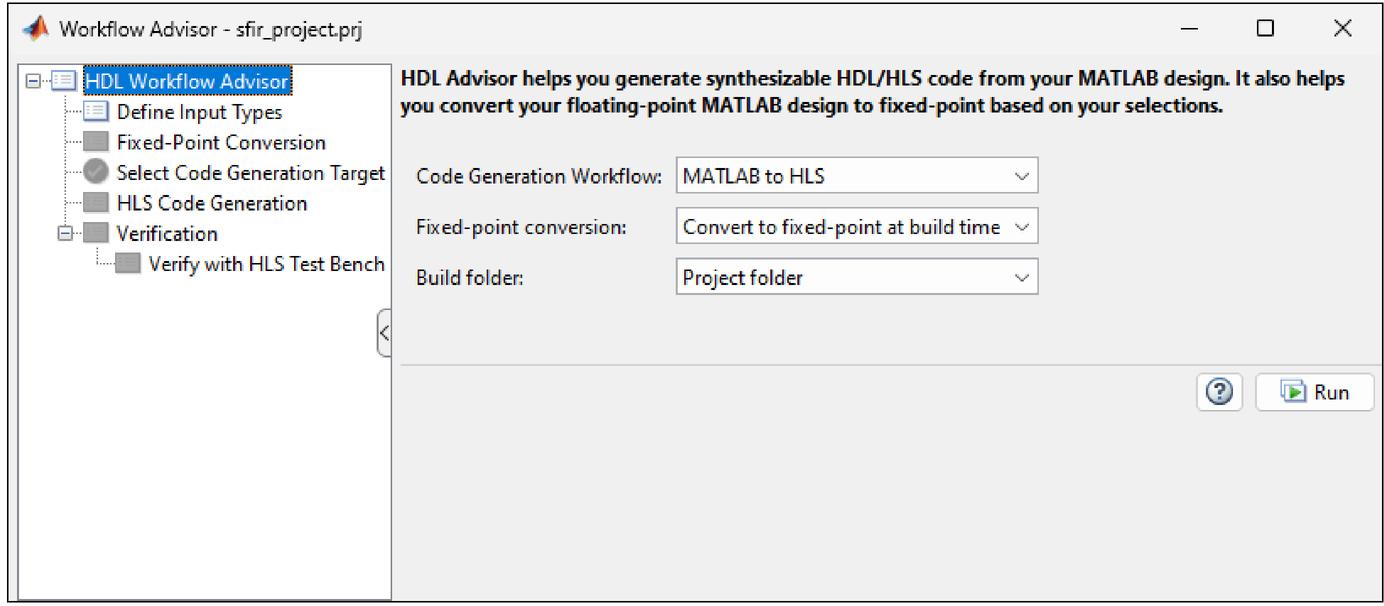
3. Click **Autodefine types** and use the recommended types for the MATLAB design. The code generator infers the input types from the MATLAB test bench.



Run Fixed-Point Conversion and HLS Code Generation in HDL Workflow Advisor

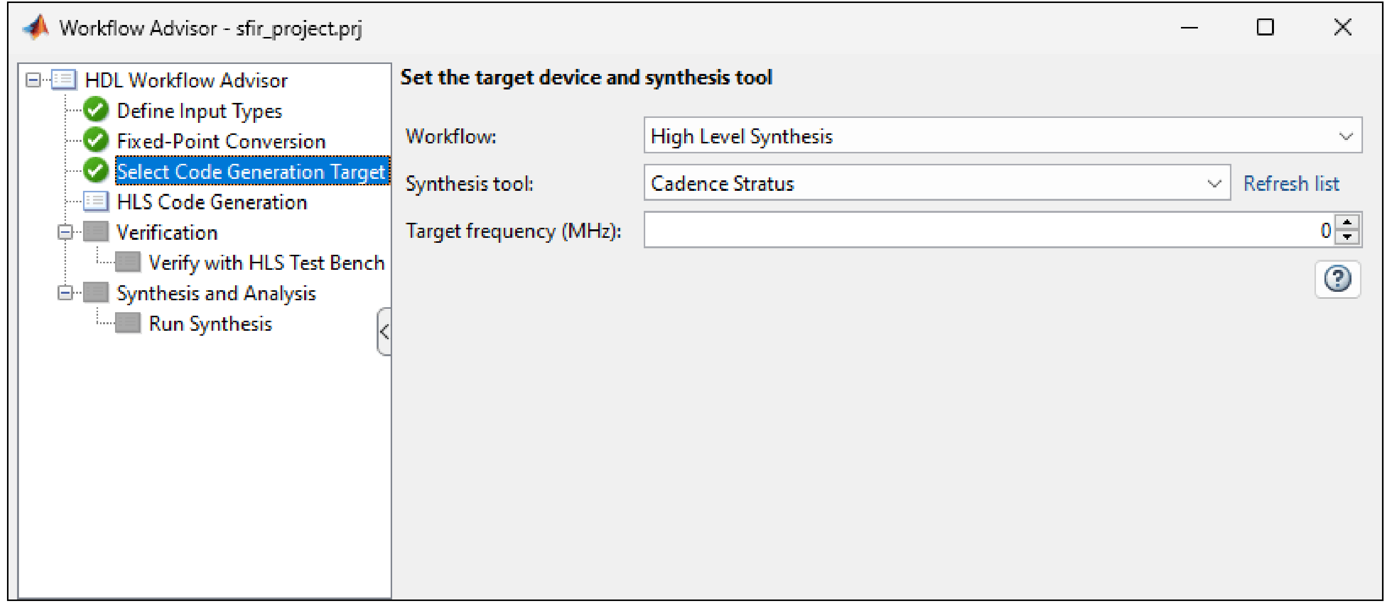
1. To start the HDL Workflow Advisor, click the **Workflow Advisor** button.

2. In **HDL Workflow Advisor** step, specify **Code Generation workflow** as **MATLAB to HLS**.



3. If your design does not use fixed-point types and functions, then translate your floating-point MATLAB design to a fixed-point design. To examine the generated fixed-point code from the floating-point design, click the **Fixed-Point Conversion** task. The generated fixed-point MATLAB code opens in the MATLAB editor. For details, see “Floating-Point to Fixed-Point Conversion” on page 4-51.

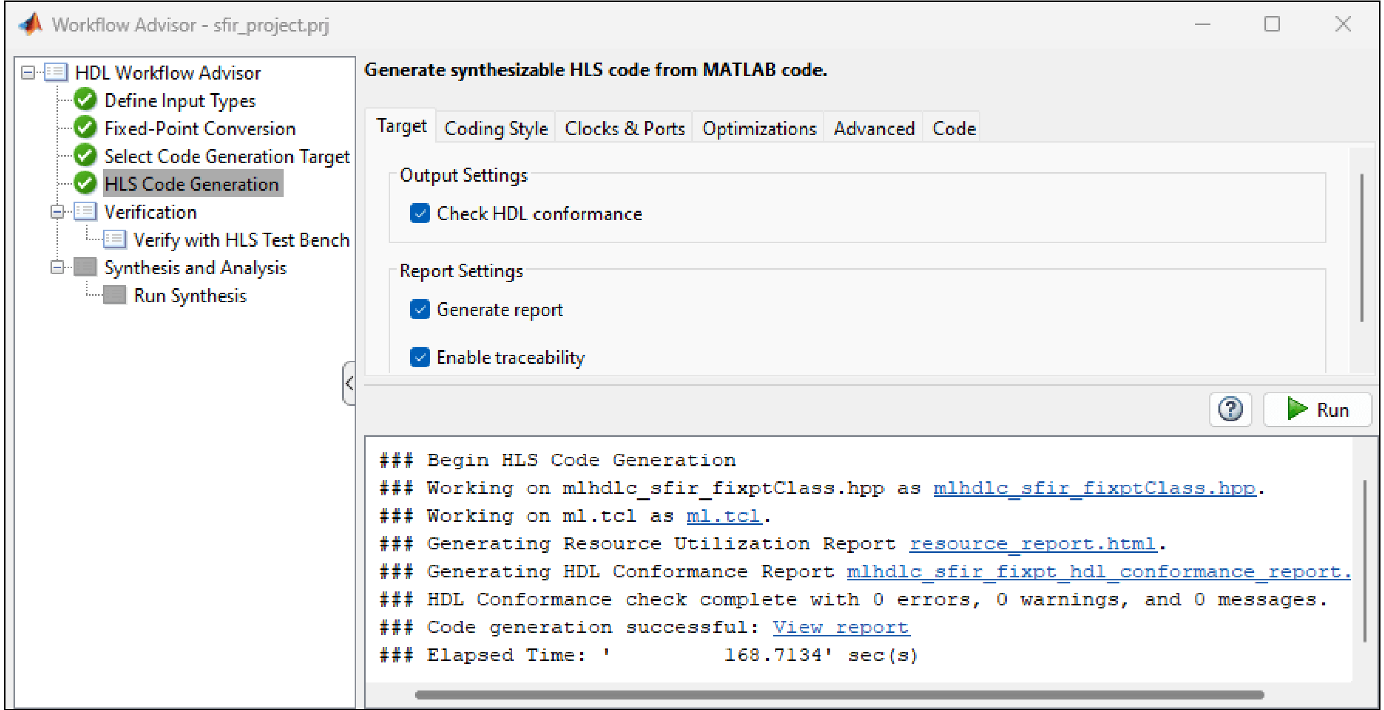
4. In **Select Code Generation Target** step, specify **Workflow** as **High Level Synthesis** and **Synthesis tool** as **Cadence Stratus**.



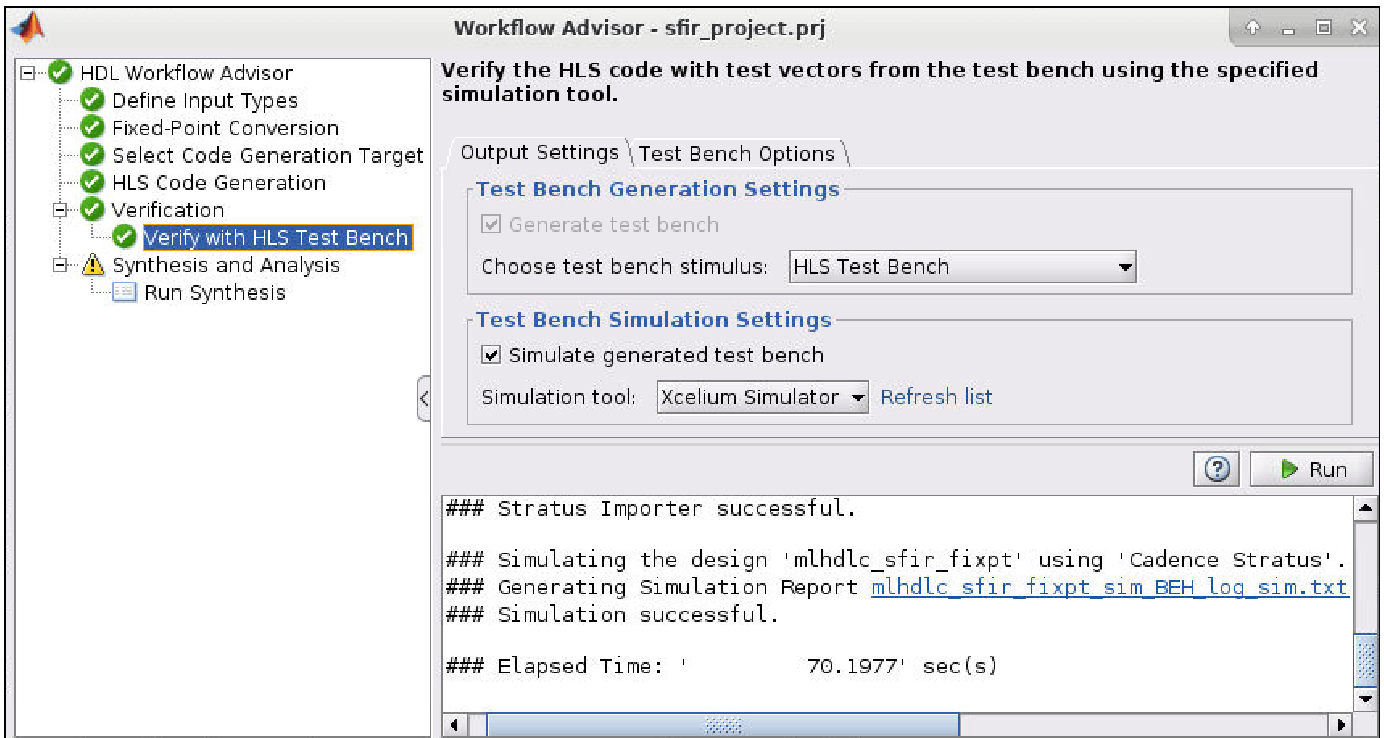
5. Right-click the **HLS Code Generation** task and select **Run to selected task**.

The code generator runs the Workflow Advisor tasks to generate HLS code for the filter design.

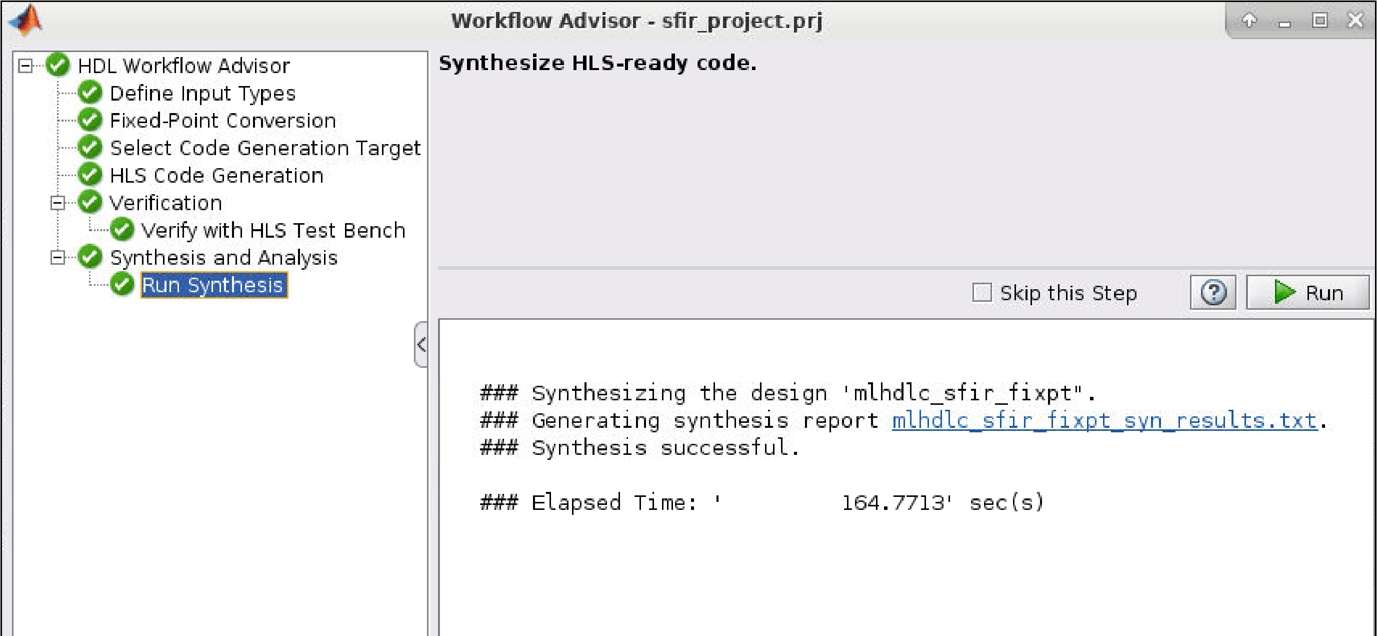
To examine the generated HLS code, click the hyperlink to `mlhdlc_sfir_fixptClass.hpp` in the **HLS Code Generation** log window.



6. You can compare the output of the generated HLS code to that of the MATLAB code in the **Verify with HDL Test Bench** step. This step also generates HLS test bench files and creates the Cadence Stratus project.



7. Synthesis is the final step in the code generation process. In the **Run Synthesis** step, the generated HLS code is synthesized using the HLS tool and Verilog or VHDL code is generated.



HLS Code Generation Report

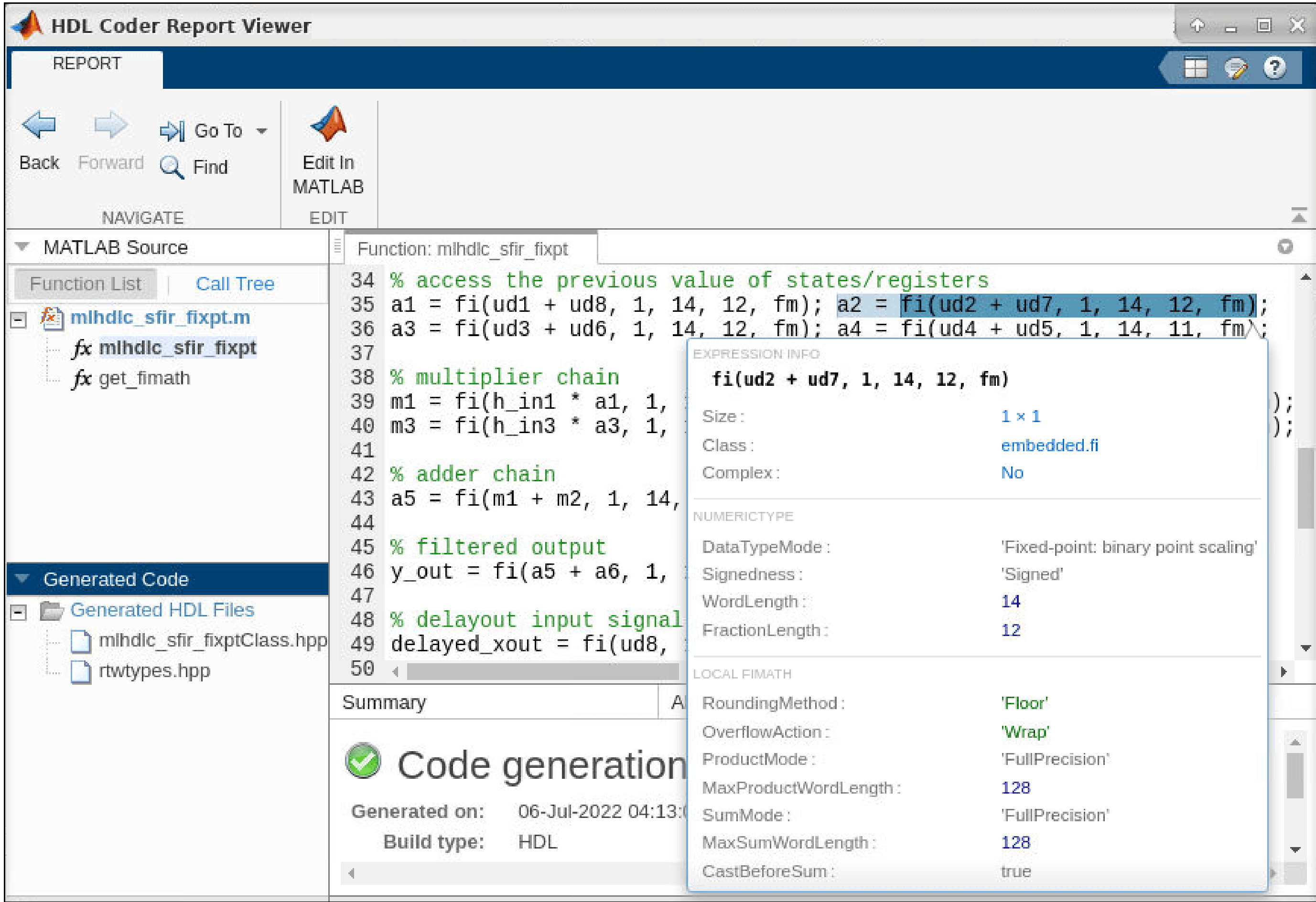
At the **HLS Code Generation** step, you can check the code generation report by clicking `View Report` in the log window.

The code generation report helps you to:

- Debug code generation issues and verify that your MATLAB code is suitable for code generation.
- View generated HLS code.
- Access additional reports like a conformance report and resource utiliztion report.
- See how the code generator determines and propagates type information for variables and expressions in your MATLAB code.

To view a MATLAB function in the code pane, click the name of the function in the MATLAB Source pane. In the code pane, when you pause on a variable or expression, a tooltip displays information about its size, type, and complexity.

For more information see, “HLS Code Generation Report” on page 11-29.



Limitations

- Only Cadence Stratus is supported as the high level synthesis tool. Cadence Stratus supports only point to point (p2p) communication for HLS code generation.
- Generating HLS code that runs on multiple threads is not supported.
- Systems objects are not supported for HLS code generation.
- Structures and enumerations are not supported as inputs and outputs at the top-level DUT ports for HLS code generation.

See Also

- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “Verify HLS Code That Has an HDL Test Bench” on page 12-2
- “HLS Code Generation Report” on page 11-29
- “Structure Definition for HLS Code Generation” on page 1-65

Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface

This example shows how to use the HDL Coder™ command-line interface to generate High-Level Synthesis (HLS) code from MATLAB® code, including floating-point to fixed-point conversion.

Overview

High-Level Synthesis (HLS) code generation using the command-line interface has the following basic steps:

1. Set up the HLS tool path for HLS code generation by using the function `hdlsetuphlstoolpath`.
2. Create a `fixpt` coder configuration object.
3. Create an `hdl` coder configuration object.
4. Set configuration object parameters.
5. Run the `codegen` command to generate HLS code.

The HDL Coder command-line interface can use two coder configuration objects with the `codegen` command. The `fixpt` coder configuration object configures the floating-point to fixed-point conversion of your MATLAB code. The `hdl` coder configuration object configures HLS code generation and programming options.

The example code implements a simple symmetric finite impulse response (FIR) filter and its test bench. Using this example, you can configure floating-point to fixed-point conversion and generate HLS code.

Set up the FIR model and test bench for this example. Ensure that the file path includes no spaces.

```
mlhdlc_demo_setup('sfir');
```

Create Floating-Point to Fixed-Point Conversion Configuration Object

If your design already uses fixed-point types and functions, then you can skip the fixed-point conversion step.

To perform floating-point to fixed-point conversion, you need a `fixpt` configuration object.

Create a `fixpt` configuration object and specify your test bench name.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_sfir_tb';
```

Fixed-Point Conversion Type Proposal Options

The code generator can propose fixed-point types based on your choice of either word length or fraction length. These two options are mutually exclusive.

Base the proposed types on a word length of `24`:

```
fixptcfg.DefaultWordLength = 24;
fixptcfg.ProposeFractionLengthsForDefaultWordLength = true;
```

Alternatively, you can base the proposed fixed-point types on fraction lengths. The following code configures the coder to propose types based on a fraction length of 10:

```
fixptcfg.DefaultFractionLength = 10;
fixptcfg.ProposeWordLengthsForDefaultFractionLength = true;
```

Safety Margin

The code generator increases the simulation data range on which it bases its fixed-point type proposal by the safety margin percentage. For example, the default safety margin is 4, which increases the simulation data range used for fixed-point type proposal by 4%.

Set the SafetyMargin to 10%:

```
fixptcfg.SafetyMargin = 10;
```

Data Logging

The code generator runs the test bench with the design before and after a floating-point to fixed-point conversion. You can enable simulation data logging to plot the quantization effects of the new fixed-point data types.

Enable data logging in the fixpt configuration object:

```
fixptcfg.LogIOForComparisonPlotting = true;
```

Numeric Type Proposal Report

Configure the code generator to start the type proposal report once the fixed-point types have been proposed:

```
fixptcfg.LaunchNumericTypesReport = true;
```

Create HLS Code Generation Configuration Object

To generate HLS code, you must create an hdl configuration object and set your test bench name:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_sfir_tb';
```

Target Language and Synthesis Tool

To generate HLS code, specify the workflow as High Level Synthesis and the synthesis tool as Cadence Stratus.

```
hdlcfg.Workflow = 'High Level Synthesis';
hdlcfg.SynthesisTool = 'Cadence Stratus';
```

Generation of HLS Test Bench Code

Configure the code generator to generate a HLS test bench from your MATLAB® test bench:

```
hdlcfg.GenerateHDLTestBench = true;
```

Simulation of the Generated HLS Code

If you want to simulate your generated HLS code, you must also configure the code generator to generate a HLS test bench.

```
hdlcfg.SimulateGeneratedCode = true;
```

Synthesis Tool for Generated HLS Code

You can synthesize your generated HLS code by using the synthesis tool **Cadence Stratus**. Configure the code generator to use this tool.

```
hdlcfg.SynthesizeGeneratedCode = true;
```

Run HLS Code Generation

Now that you have your `fixpt` and `hdl` configuration objects set up, run the `codegen` command to perform floating-point to fixed-point conversion and generate HLS code:

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_sfir -report
```

HLS Code Generation Report

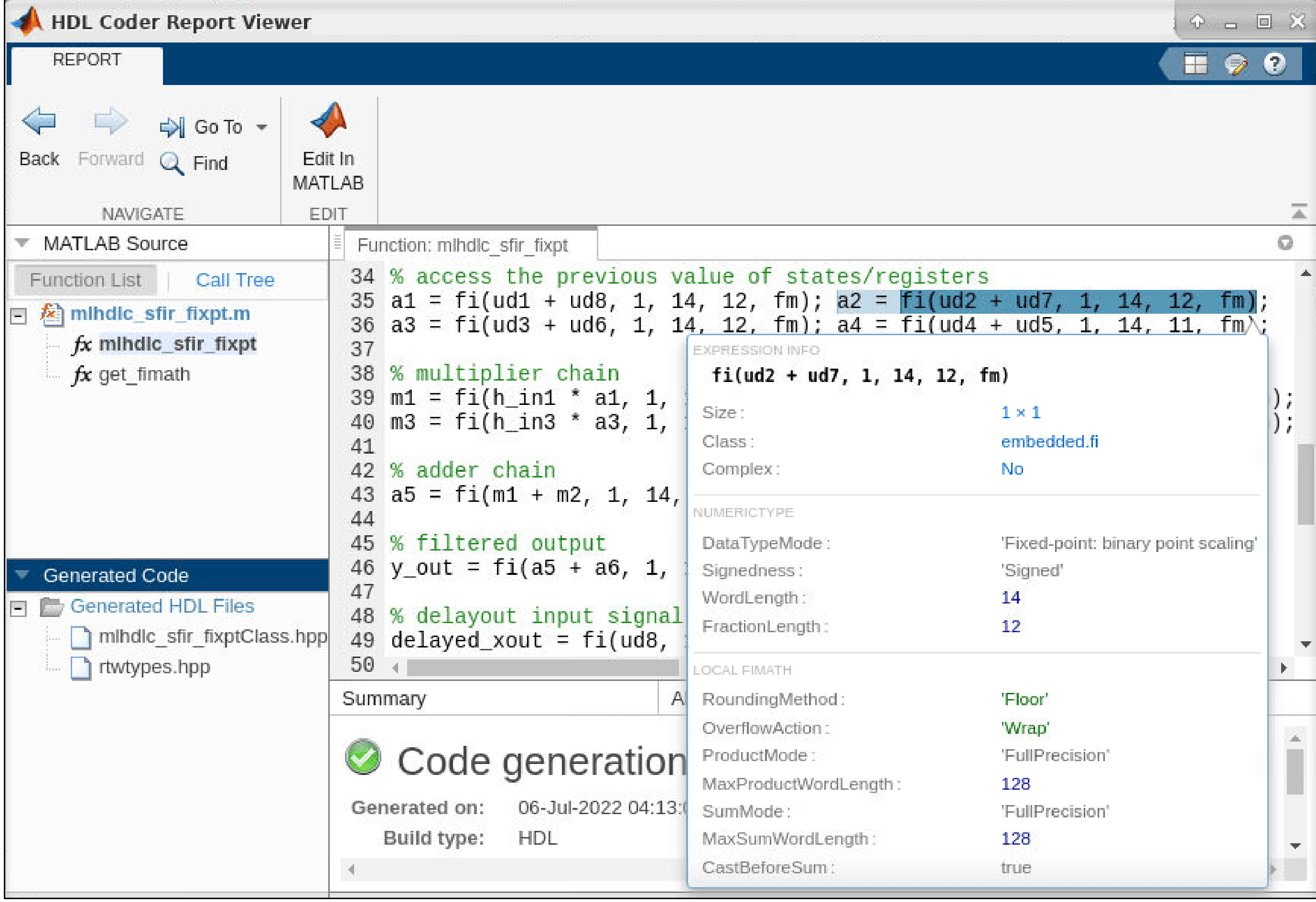
At the MATLAB command line window, you can check the code generation report by clicking `View Report`.

The code generation report helps you to:

- Debug code generation issues and verify that your MATLAB code is suitable for code generation.
- View generated HLS code.
- Access additional reports like a conformance report and resource utiliztion report.
- See how the code generator determines and propagates type information for variables and expressions in your MATLAB code.

To view a MATLAB function in the code pane, click the name of the function in the MATLAB Source pane. In the code pane, when you pause on a variable or expression, a tooltip displays information about its size, type, and complexity.

For more information see, “HLS Code Generation Report” on page 11-29.



Limitations

- Only Cadence Stratus is supported as the high level synthesis tool. Cadence Stratus supports only point to point (p2p) communication for HLS code generation.
- Generating HLS code that runs on multiple threads is not supported.
- Systems objects are not supported for HLS code generation.
- Structures and enumerations are not supported as inputs and outputs at the top-level DUT ports for HLS code generation.

See Also

- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “Verify HLS Code That Has an HDL Test Bench” on page 12-2
- “HLS Code Generation Report” on page 11-29
- “Structure Definition for HLS Code Generation” on page 1-65

Replace Arithmetic Operation to Generate Efficient HDL and High-Level Synthesis Code

Best practices when using division operation in your MATLAB® code for HDL and High-Level Synthesis (HLS) code generation.

Division Operation

One of the fundamental arithmetic operation, division is found in many applications including vector normalization and matrix decomposition. The example starts with floating-point division operation and use fixed-point conversion to map division into efficient implementations before generating code.

Division Using Cordic Algorithm

Cordic algorithm is one of the hardware efficient algorithms for division operations that require only shift-add operations. You can use `fixed.cordicDivide` method supported for fixed-point data type in MATLAB to generate efficient code.

To facilitate automatic replacement of division operation with `fixed.cordicDivide` function:

- Write the division operation into a separate function called `divide_op.`

```
function y = divide_example(u, v)
  y = divide_op(u, v);
end

function y = divide_op(u, v)
 y = u / v;
end
```

- Replace the `divide_op` function with `CordicDivideFcn` when converting from floating-point type to fixed-point data type.

```
function y = CordicDivideFcn(a, b)
 y = fixed.cordicDivide(a, b, numerictype(0,14,14));
end
```

Perform fixed-point conversion and generate HLS code by using these commands.

```
% Create HDL configuration object
hdlcfg = coder.config('hdl');
hdlcfg.Workflow = 'High Level Synthesis';
hdlcfg.TargetLanguage = 'SystemC';
hdlcfg.DesignFunctionName = 'divide_example';
hdlcfg.TestBenchName = 'divide_example_tb';

% Create float to fixed configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = { 'divide_example_tb.m' };
cfg.LogIOForComparisonPlotting = true;
cfg.TestNumerics = true;
cfg.PlotWithSimulationDataInspector = true;

% Replace divide_op with cordic divide function
cfg.addFunctionReplacement('divide_op', 'CordicDivideFcn');
```

```
% Invoke fixed-point conversion and generate SystemC code
codegen -float2fixed cfg divide_example -config hdlcfg -args {0,0}
```

```
===================================================
Design Name: divide_example
Test Bench Name: divide_example_tb
===================================================


============= Step1: Analyze floating-point code ==============

Code generation successful.



============= Step1a: Verify Floating Point ==============

### Analyzing the design 'divide_example'
### Analyzing the test bench(es) 'divide_example_tb'
### Begin Floating Point Simulation (Instrumented)
### Floating Point Simulation Completed in   3.2219 sec(s)
### Elapsed Time:             4.1406 sec(s)

============= Step2: Propose Types based on Range Information ==============


============= Step3: Generate Fixed Point Code ==============

### Generating Fixed Point MATLAB Code divide_example_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper divide_example_wrapper_fixpt
### Generating Mex file for ' divide_example_wrapper_fixpt '
Code generation successful: View report
### Generating Type Proposal Report for 'divide_example' divide_example_report.html

============= Step4: Verify Fixed Point Code ==============

### Analyzing the design 'divide_example'
### Analyzing the test bench(es) 'divide_example_tb'
### Begin Floating Point Simulation
### Floating Point Simulation Completed in   0.4898 sec(s)
### Begin Fixed Point Simulation : divide_example_tb
### Fixed Point Simulation Completed in   3.5850 sec(s)

Generating comparison plot(s) for 'divide_example' using Simulation Data Inspector.

--------------  Input variable : u  --------------
Generating comparison plot...

--------------  Input variable : v  --------------
Generating comparison plot...

--------------  Output variable : y  --------------
Generating comparison plot...

### Generating Fixed-point Types Report for 'divide_example_fixpt' divide_example_fixpt_report.h
### Elapsed Time:             8.6753 sec(s)
```

```
==================================================
Code generation successful.

### Begin HLS Code Generation
### Working on divide_example_fixptClass.hpp as divide_example_fixptClass.hpp.
### Working on divide_example_fixptModule.hpp as divide_example_fixptModule.hpp.
### Generating Resource Utilization Report resource_report.html.
### Generating HDL Conformance Report divide_example_fixpt_hdl_conformance_report.html.
### HDL Conformance check complete with 0 errors, 0 warnings, and 0 messages.
Code generation successful.
```

In the HDL workflow advisor, function replacements are done in the fixed-point converter step using the **Function Replacements** tab.

Division Using Shift-Add

Non-restoring division algorithm uses only shifts and addition operations and is advantageous to use in hardware. You can use a custom implementation of this division algorithm for fixed-point data types in function `shift_add_divider`. Write division operation in a separate function which can be replaced after fixed-point conversion with our custom implementation function `shift_add_divider`.

```
function y = divide_example(u, v)
  y = divide_op(u, v);
end

function y = divide_op(u, v)
 y = u / v;
end
```

Perform fixed-point conversion and generate HLS code by using these commands.

```
% Create configuration object of class 'coder.HdlConfig'.
hdlcfg = coder.config('hdl');
hdlcfg.Workflow = 'High Level Synthesis';
hdlcfg.TargetLanguage = 'SystemC'; % Use target language as 'HDL' for HDL code generation
hdlcfg.DesignFunctionName = 'divide_example';
hdlcfg.TestBenchName = 'divide_example_tb';

% Create configuration object of class 'coder.FixPtConfig'.
cfg = coder.config('fixpt');
cfg.TestBenchName = { 'divide_example_tb.m' };
cfg.LogIOForComparisonPlotting = true;
cfg.TestNumerics = true;
cfg.PlotWithSimulationDataInspector = true;
```

```
% Replace divide_op with shift-add divide function
cfg.addTypeSpecification('divide_example', 'u', numerictype(0,14,10));
cfg.addTypeSpecification('divide_op', 'u', numerictype(0,14,10));
cfg.addFunctionReplacement('divide_op', 'shift_add_divider');

% Invoke fixed-point conversion and generate SystemC
codegen -float2fixed cfg -config hdlcfg divide_example -args {0,0}
```

```
===================================================
Design Name: divide_example
Test Bench Name: divide_example_tb
===================================================


============= Step1: Analyze floating-point code ==============


============= Step1a: Verify Floating Point ==============

### Analyzing the design 'divide_example'
### Analyzing the test bench(es) 'divide_example_tb'
### Begin Floating Point Simulation (Instrumented)
### Floating Point Simulation Completed in   1.1100 sec(s)
### Elapsed Time:             1.5386 sec(s)

============= Step2: Propose Types based on Range Information ==============


============= Step3: Generate Fixed Point Code ==============

### Generating Fixed Point MATLAB Code divide_example_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper divide_example_wrapper_fixpt
### Generating Mex file for ' divide_example_wrapper_fixpt '
Code generation successful: View report
### Generating Type Proposal Report for 'divide_example' divide_example_report.html

============= Step4: Verify Fixed Point Code ==============

### Analyzing the design 'divide_example'
### Analyzing the test bench(es) 'divide_example_tb'
### Begin Floating Point Simulation
### Floating Point Simulation Completed in   0.5077 sec(s)
### Begin Fixed Point Simulation : divide_example_tb
### Fixed Point Simulation Completed in   3.5773 sec(s)

Generating comparison plot(s) for 'divide_example' using Simulation Data Inspector.

--------------  Input variable : u  --------------
Generating comparison plot...

--------------  Input variable : v  --------------
Generating comparison plot...

--------------  Output variable : y  --------------
Generating comparison plot...

### Generating Fixed-point Types Report for 'divide_example_fixpt' divide_example_fixpt_report.h
### Elapsed Time:             5.7933 sec(s)
```

```
====================================================
Code generation successful.

### Begin HLS Code Generation
### Working on divide_example_fixptClass.hpp as divide_example_fixptClass.hpp.
### Working on divide_example_fixptModule.hpp as divide_example_fixptModule.hpp.
### Generating Resource Utilization Report resource_report.html.
### Generating HDL Conformance Report divide_example_fixpt_hdl_conformance_report.html.
### HDL Conformance check complete with 0 errors, 0 warnings, and 0 messages.
Code generation successful.
```

Division Using Lookup Table

When the output of division does not have high dynamic range, use lookup table approach to compute reciprocal and then multiply the reciprocal with numerator to perform division operation.

For using a 1-D lookup table, split the division operation into a reciprocal operation and a multiplication. Implement the reciprocal using the lookup table.

```
function y = recip_example(u, v)
   recip = reciprocal_op(v);
   y = u * recip;
end
```

The function `reciprocal_op` need to be in a separate file so that the fixed-point converter can use the results of this function to create lookup table data.

```
function y = reciprocal_op(v)
  y = 1 / v;
end
```

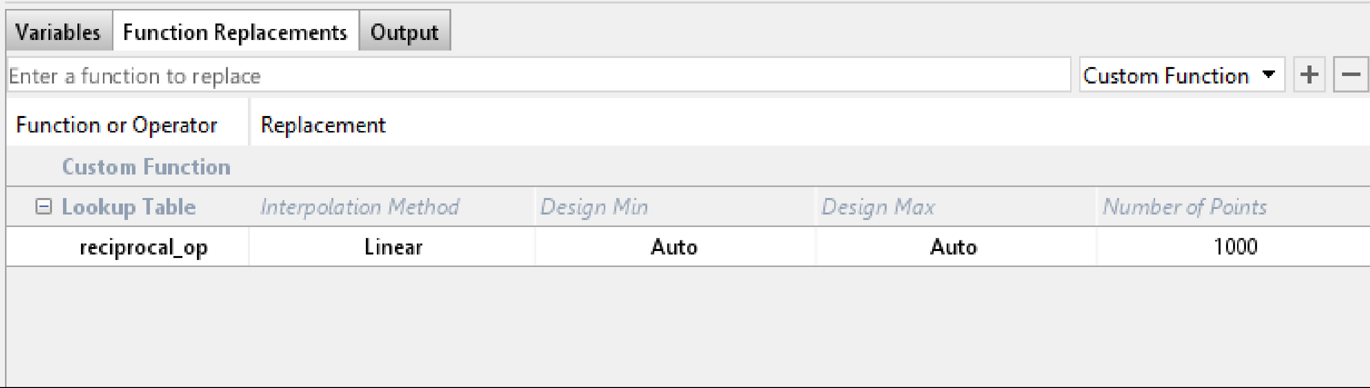
Perform fixed-point conversion and generate HLS code by using these commands.

```
% HDL Configuration
hdlcfg = coder.config('hdl');
hdlcfg.Workflow = 'High Level Synthesis';
hdlcfg.TargetLanguage = 'SystemC';
hdlcfg.DesignFunctionName = 'recip_example';
hdlcfg.TestBenchName = 'recip_example_tb';

% Fixpt conversion configuration
cfg = coder.config('fixpt');
cfg.TestBenchName = { 'recip_example_tb.m' };
cfg.LogIOForComparisonPlotting = true;
cfg.TestNumerics = true;
cfg.PlotWithSimulationDataInspector = true;

cfg.addTypeSpecification('recip_example', 'v', numerictype(0,6,0));

% Lookup table specification
approxConfig = coder.approximation('reciprocal_op');
approxConfig.CandidateFunction = 'reciprocal_op';
approxConfig.InterpolationDegree = 1;
approxConfig.NumberOfPoints = 1000;
cfg.addApproximation(approxConfig);

% Invoke code generation
codegen -float2fixed cfg recip_example -config hdlcfg -args {0,0}
```

```
===================================================
Design Name: recip_example
Test Bench Name: recip_example_tb
===================================================


============= Step1: Analyze floating-point code ==============

Code generation successful.



============= Step1a: Verify Floating Point ==============

### Analyzing the design 'recip_example'
### Analyzing the test bench(es) 'recip_example_tb'
### Begin Floating Point Simulation (Instrumented)
### Floating Point Simulation Completed in   1.0096 sec(s)
### Elapsed Time:             1.4969 sec(s)

============= Step2: Propose Types based on Range Information ==============


============= Step3: Generate Fixed Point Code ==============

### Generating Fixed Point MATLAB Code recip_example_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper recip_example_wrapper_fixpt
### Generating Mex file for ' recip_example_wrapper_fixpt '
Code generation successful: View report
### Generating Type Proposal Report for 'recip_example' recip_example_report.html

============= Step4: Verify Fixed Point Code ==============

### Analyzing the design 'recip_example'
### Analyzing the test bench(es) 'recip_example_tb'
### Begin Floating Point Simulation
### Floating Point Simulation Completed in   0.2613 sec(s)
### Begin Fixed Point Simulation : recip_example_tb
### Fixed Point Simulation Completed in   2.0110 sec(s)

Generating comparison plot(s) for 'recip_example' using Simulation Data Inspector.

--------------  Input variable : u  --------------
Generating comparison plot...

--------------  Input variable : v  --------------
Generating comparison plot...

--------------  Output variable : y  --------------
Generating comparison plot...

### Generating Fixed-point Types Report for 'recip_example_fixpt' recip_example_fixpt_report.html
### Elapsed Time:             3.7854 sec(s)

===================================================
Code generation successful.

### Begin HLS Code Generation
```

```
### Working on recip_example_fixptClass.hpp as recip_example_fixptClass.hpp.
### Working on recip_example_fixptModule.hpp as recip_example_fixptModule.hpp.
### Generating Resource Utilization Report resource_report.html.
### Generating HDL Conformance Report recip_example_fixpt_hdl_conformance_report.html.
### HDL Conformance check complete with 0 errors, 0 warnings, and 0 messages.
Code generation successful.
```

In the HDL workflow advisor, lookup table replacements are done in the fixed-point converter step using the **Function Replacements** tab. Lookup table parameters like table size and interpolation method can be customized in the workflow advisor.



Division Using Power of 2

Dividing a number by a power of two can be more efficiently done using a shift operation. For example, `a / 2^exponent` is replaced with `bitsra(a, exponent)`, which performs a right bitwise shift. Write division operation in a separate function `localDiv` which can be replaced after fixed-point conversion with our custom implementation function `bitsra`.

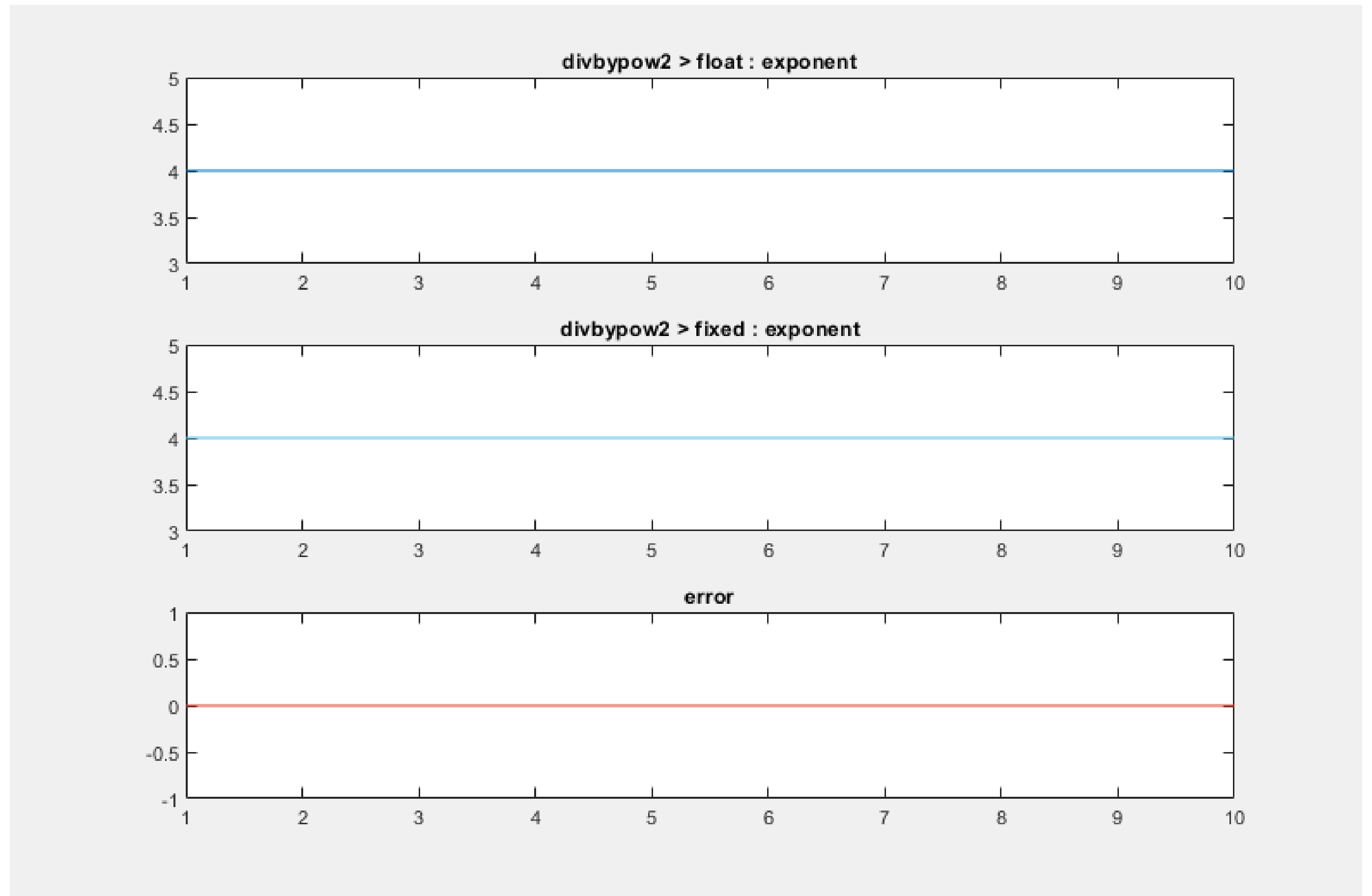
```
function y = divbypow2(u, exponent)
    y = localDiv(u, exponent);
end

function y = localDiv(u, exponent)
    y = u / 2^exponent;
end
```

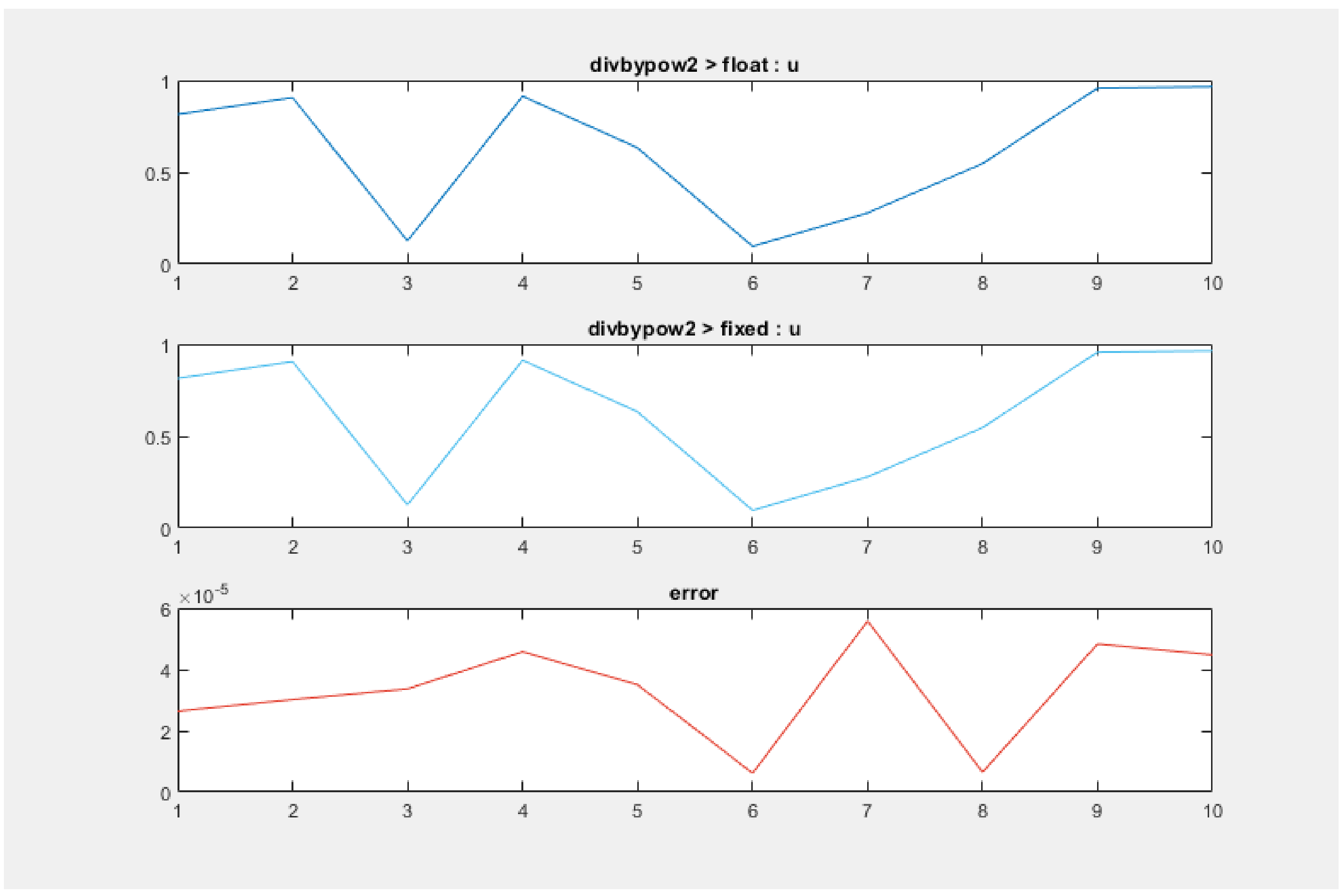
Perform fixed-point conversion and generate HLS code by using these commands.

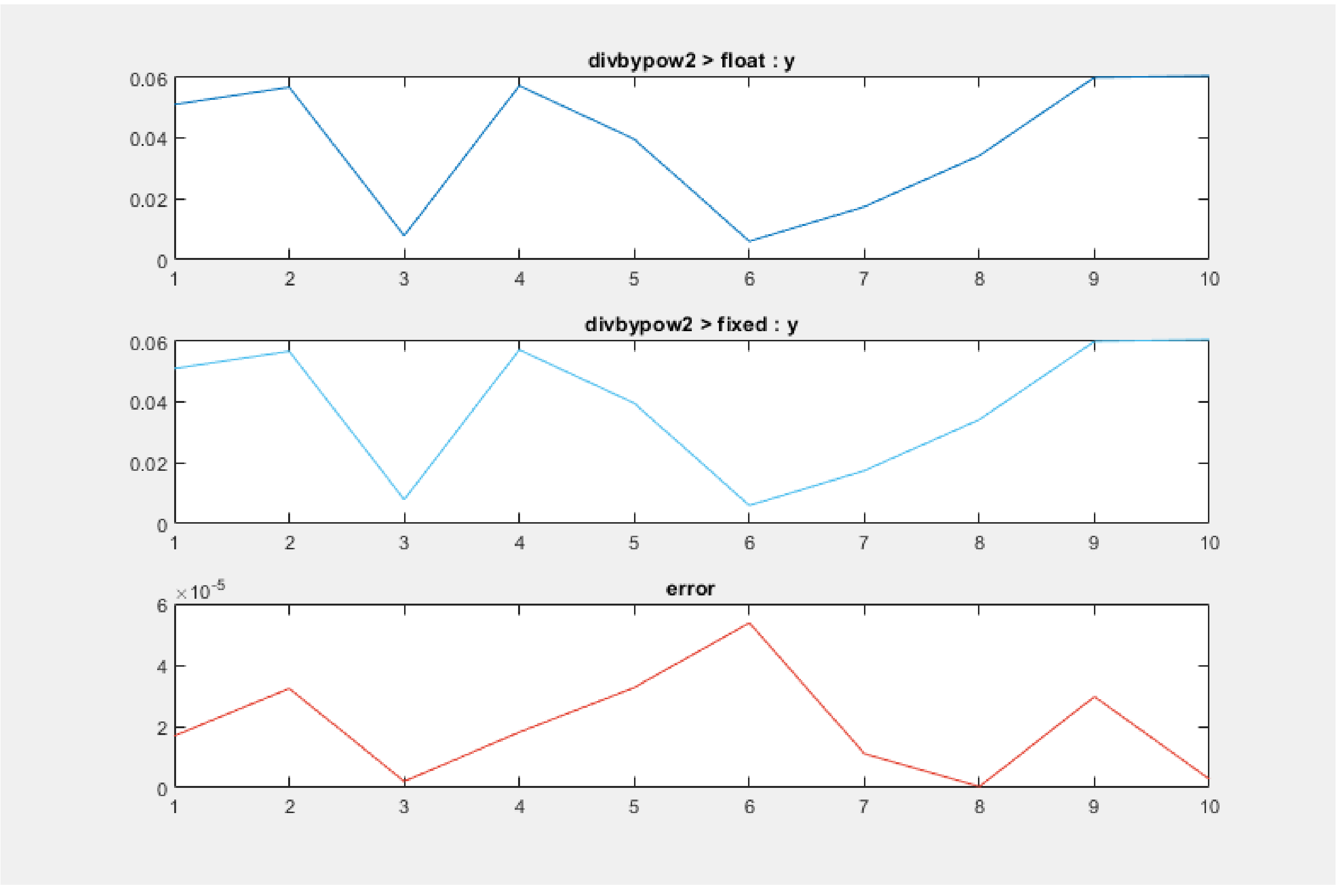
```
hdlCfg = coder.config('hdl');
hdlcfg.Workflow = 'High Level Synthesis';
hdlcfg.TargetLanguage = 'SystemC';
hdlCfg.DesignFunctionName = 'divbypow2';
hdlCfg.TestBenchName = 'divbypow2_tb';

% %% Define argument types for entry-point 'divbypow2_fixpt'.
% ARGS = cell(1,1);
% ARGS{1} = cell(2,1);
% ARGS{1}{1} = coder.typeof(fi(0,numerictype(0,14,14));
% ARGS{1}{2} = coder.typeof(fi(0,numerictype(0,3,0));
% codegen -config cfg  -args ARGS{1}
%% Create configuration object of class 'coder.FixPtConfig'.
fixptCfg = coder.config('fixpt');
fixptCfg.TestBenchName = { 'divbypow2_tb.m' };
```

```
fixptCfg.LogIOForComparisonPlotting = true;
fixptCfg.TestNumerics = true;
fixptCfg.addFunctionReplacement('localDiv', 'bitsra');

%% Invoke fixed-point conversion.
codegen -float2fixed fixptCfg divbypow2 -config hdlCfg -args {0, 0}

===================================================
Design Name: divbypow2
Test Bench Name: divbypow2_tb
===================================================


============= Step1: Analyze floating-point code ==============

Code generation successful.



============= Step1a: Verify Floating Point ==============

### Analyzing the design 'divbypow2'
### Analyzing the test bench(es) 'divbypow2_tb'
### Begin Floating Point Simulation (Instrumented)
### Floating Point Simulation Completed in   1.3807 sec(s)
### Elapsed Time:             1.9602 sec(s)

============= Step2: Propose Types based on Range Information ==============


============= Step3: Generate Fixed Point Code ==============

### Generating Fixed Point MATLAB Code divbypow2_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper divbypow2_wrapper_fixpt
### Generating Mex file for ' divbypow2_wrapper_fixpt '
Code generation successful: View report
### Generating Type Proposal Report for 'divbypow2' divbypow2_report.html

============= Step4: Verify Fixed Point Code ==============

### Analyzing the design 'divbypow2'
### Analyzing the test bench(es) 'divbypow2_tb'
### Begin Floating Point Simulation
### Floating Point Simulation Completed in   0.2857 sec(s)
### Begin Fixed Point Simulation : divbypow2_tb
### Fixed Point Simulation Completed in   1.2279 sec(s)

Generating comparison plot(s) for 'divbypow2' using PlotFunction : 'coder.internal.plotting.inBu

--------------  Input variable : u  --------------
Generating comparison plot...

Max Positive Error : 5.5836e-05
Max Negative Error : 0
Max Absolute Value : 0.96489
Max Percentage Error : 0.0057868
--------------  Input variable : exponent  --------------
Generating comparison plot...
```

```
Max Positive Error : 0
Max Negative Error : 0
Max Absolute Value : 4
Max Percentage Error : 0
-------------  Output variable : y  -------------
Generating comparison plot...

Max Positive Error : 5.3795e-05
Max Negative Error : 0
Max Absolute Value : 0.060306
Max Percentage Error : 0.089204
### Generating Fixed-point Types Report for 'divbypow2_fixpt' divbypow2_fixpt_report.html
### Elapsed Time:             4.1920 sec(s)
```




divbypow2 > float : u
divbypow2 > fixed : u
error
×10^-5




```
==========================================================
Code generation successful.

### Begin VHDL Code Generation
### Working on divbypow2_fixpt as divbypow2_fixpt.vhd.
### Generating Resource Utilization Report resource_report.html.
### Generating HDL Conformance Report divbypow2_fixpt_hdl_conformance_report.html.
### HDL Conformance check complete with 0 errors, 0 warnings, and 0 messages.
Code generation successful.
```

Summary

The above examples showed different ways of implementing division operation that are suitable for hardware. These techniques can be effectively used for HDL and HLS code generation from MATLAB.

HLS Code Generation Report

HDL Coder produces a code generation report that helps you to:

- Resolve code generation issues and verify that your MATLAB code is suitable for code generation.
- View generated High-Level Synthesis (HLS) code.
- See how the code generator determines and propagates type information for variables and expressions in your MATLAB code.
- Access additional reports such as, conformance report and resource utilization report.

Report Generation

You can use HDL Coder to produce a code generation report in one of these ways.

In the HDL Coder app:

1. Open the HDL Coder Workflow Advisor.
2. In the HDL Code Generation step options, on the **Coding Style** tab, under **Generated Code Comments**, select the **Generate report** check box.

At the command line, use `codegen` options.

- To generate a report, use the `-report` option.
- To generate and open a report, use the `-launchreport` option.

Alternatively, use configuration object properties.

- To generate a report, set `GenerateReport` to `true`.
- To open the report after generating it, set `LaunchReport` to `true`.

Report Location

The code generation report is named `report.mldatx`. It is located in the `html` subfolder of the code generation output folder. If you have MATLAB R2018a or later, you can open the `report.mldatx` file by double-clicking it.

Files and Functions

The report lists MATLAB source functions and generated files. In the **MATLAB Source** pane, the **Function List** view organizes functions according to the containing file. To visualize functions according to the call structure, use the **Call Tree** view.

To view a function in the code pane of the report, click the function in the list. Clicking a function opens the file that contains the function. To edit the selected file in the MATLAB Editor, click **Edit in MATLAB** or click a line number in the code pane.

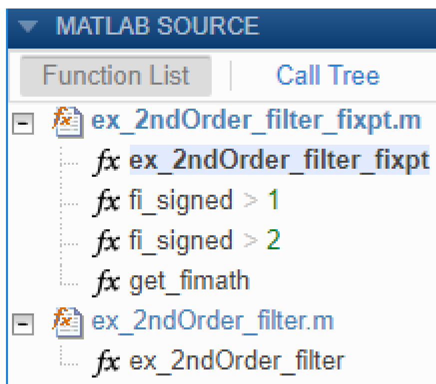
Specialized Functions or Classes

When a function is called with different types of inputs or a class uses different types for its properties, the code generator produces specializations. In the **MATLAB Source** pane, numbered functions (or classes) indicate specializations. For example:

fcn > 1
fcn > 2

Functions List After Fixed-Point Conversion

If you convert floating-point MATLAB code to fixed-point MATLAB code, and then generate fixed-point HLS code, the **MATLAB Source** pane lists the original MATLAB functions and the fixed-point MATLAB functions. For example:



MATLAB Source

To view a MATLAB function in the code pane, click the name of the function in the **MATLAB Source** pane. In the code pane, when you hover on a variable or expression, a tooltip displays information about its size, type, and complexity. Additionally, syntax highlighting helps you to identify MATLAB syntax elements and certain code generation attributes, such as whether a function is extrinsic or whether an argument is constant.

Extrinsic Functions

The report identifies an extrinsic function with purple text. The tooltip indicates that the function is extrinsic.

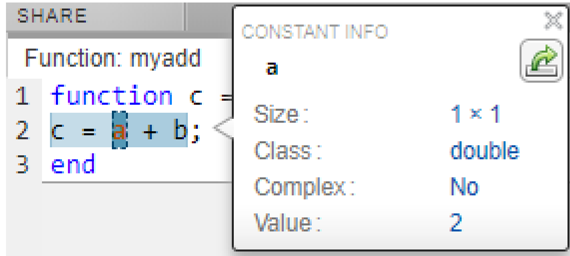
Function: callMyExtrinsic

```
1 function z = callMyExtrinsic(a,b)
2 %#codegen
3 coder.extrinsic('myExtr
4 z = 0;
5 z = myExtrinsic(a,b);
6 disp(z);
7 end
8
```

EXPRESSION INFO
myExtrinsic(a,b)
Size : 1 × 1
Class : mxArray
myExtrinsic is an extrinsic function.

Constant Arguments

Orange text indicates a compile-time constant argument to an entry-point function or a specialized function. The tooltip includes the constant value.



Knowing the value of a constant argument helps you to understand the generated function signatures. It also helps you to see when code generation creates function specializations for different constant argument values.

To export the value to a variable in the workspace, click the Export icon .

MATLAB Variables

The **Variables** tab provides information about the variables for the selected MATLAB function. To select a function, click the function in the **MATLAB Source** pane.

The variables table shows:

- Class, size, and complexity
- Properties of fixed-point types

This information helps you to understand type propagation and identify type mismatch errors.

Visual Indicators on the Variables Tab

This table describes the symbols, badges, and other indicators in the variables table.

Column in the Variables Table	Indicator	Description
Name	expander	Variable has elements or properties that you can see by clicking the expander.
Name	`{:}`	Heterogeneous cell array (all elements have the same properties)
Name	`{n}`	nth element of a heterogeneous cell array

Column in the Variables Table	Indicator	Description
Class	`v > n`	`v` is reused with a different class, size, and complexity. The number `n` identifies each unique reuse (a reuse with a unique set of properties). When you pause over a renamed variable, the report highlights only the instances of this variable that share the class, size, and complexity.
Class	`complex` prefix	Complex number
Class	fi	Fixed-point type To see the fixed-point properties, click the badge.

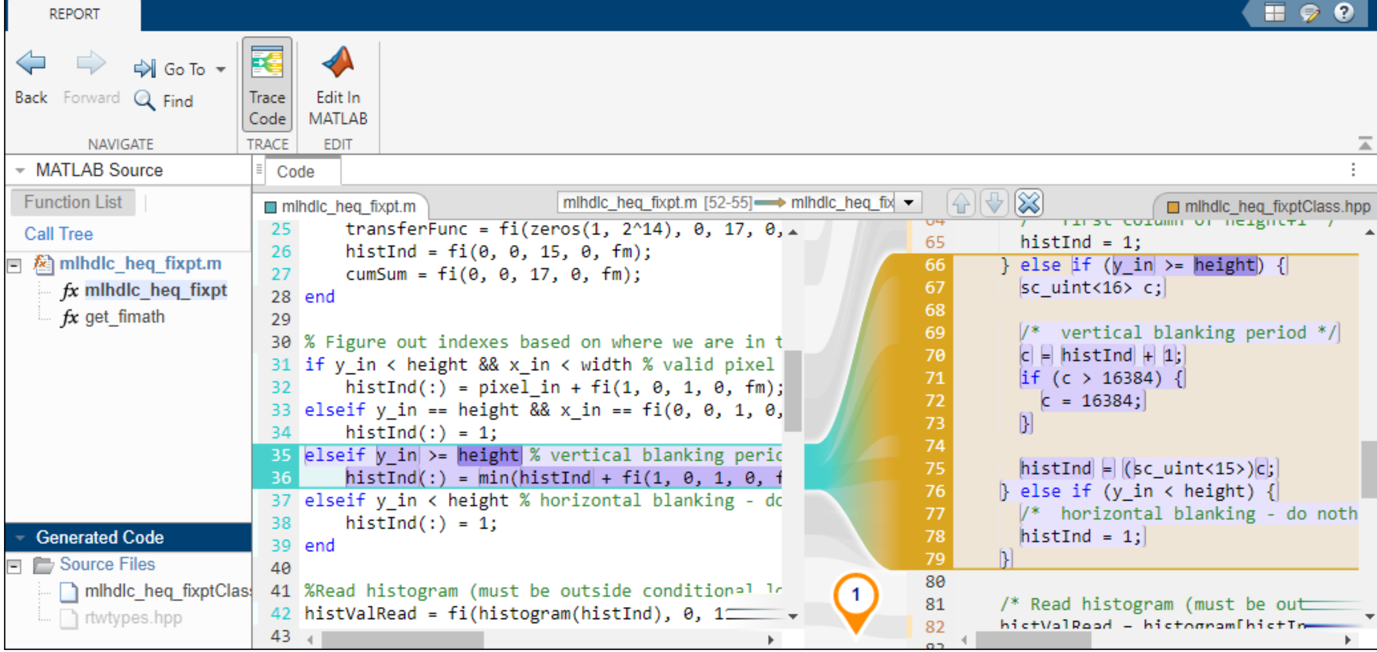
Tracing Code

You can trace between MATLAB source code and generated HLS code by using one of these methods:

- Interactively visualize the mapping between the MATLAB code and the generated code. To access interactive tracing, open the code generation report and click **Trace Code**.
- Include source code as comments in the generated HLS code.

When you click **Trace Code** on the HDL Coder Report Viewer, you see the MATLAB source code and the generated HLS code next to each other.

When you move your pointer over expressions that are part of larger expressions, different shades of purple help you to find the relevant expression in the corresponding HLS code.



Code Insights

The code generator can detect and report issues that can potentially occur in the generated code. View the messages on the **Code Insights** tab.

HLS code insights capture these inefficient loop constructs:

- Presence of unbounded loops in the MATLAB code.
- Presence of `break`, `continue`, or `return` statements in the MATLAB code.

Additional Reports

The **Summary** tab can have links to these additional reports:

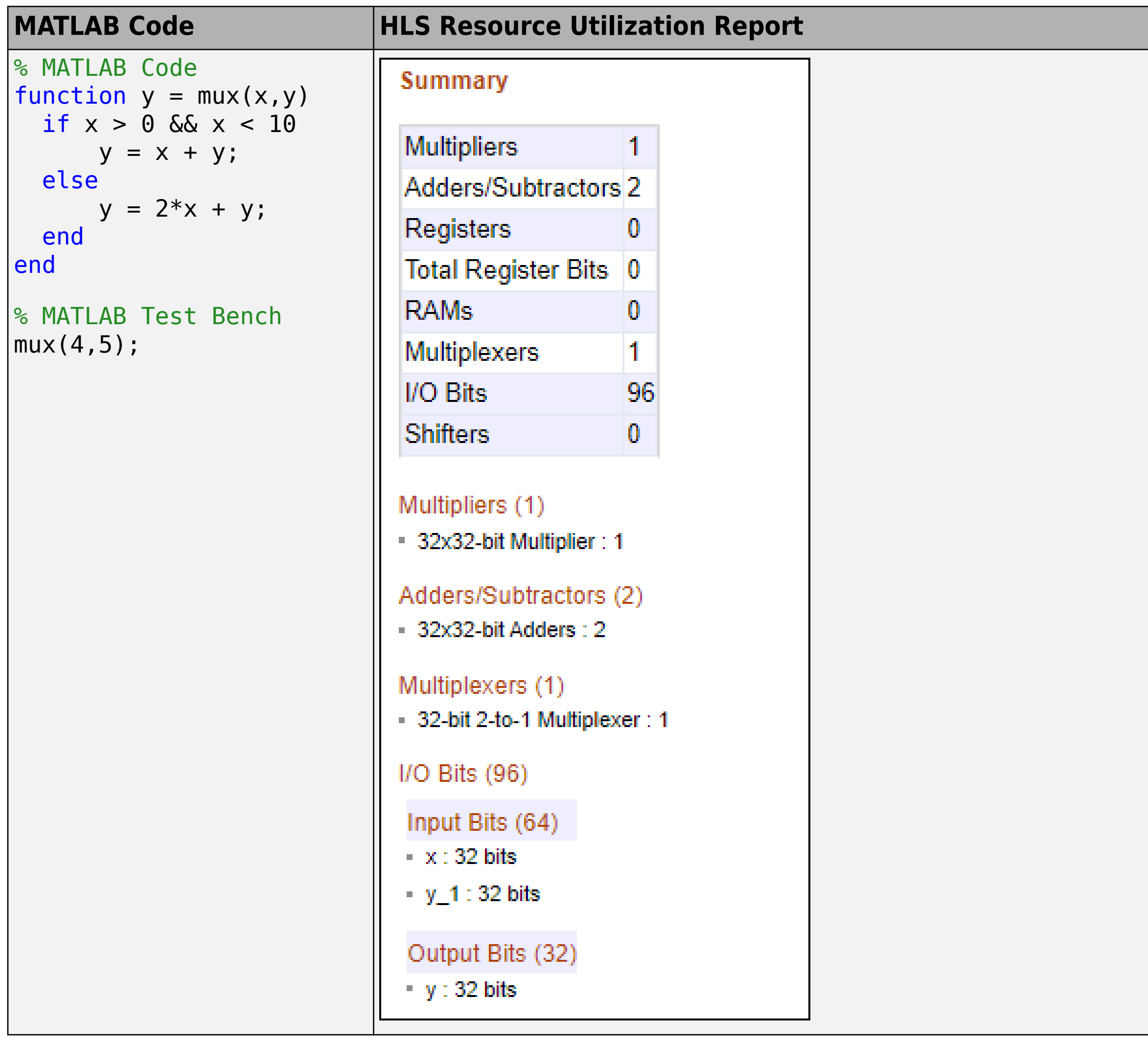
- **Conformance report**

  You can view the code generation error, warning, and information messages in the conformance report. To highlight the source code for an error or warning, click the message. It is a best practice to address the first message before later messages because subsequent errors and warnings can be related to the first message.

- **Resource utilization report**

  The report contains a summary and detailed information about the hardware resources that could be utilized during synthesis of the generated HLS code.

  For example, consider the MATLAB code and its tech bench.



MATLAB Code	HLS Resource Utilization Report

```
% MATLAB Code
function y = mux(x,y)
  if x > 0 && x < 10
      y = x + y;
  else
      y = 2*x + y;
  end
end

% MATLAB Test Bench
mux(4,5);
```


The table lists the mapping of operations/operators used in the MATLAB code to the hardware resources.

Resource Name	Description
Multipliers	Multiplication operations maps to multipliers on hardware.
Adders and Subtractors	Addition and subtraction operations maps to adders and subtractors on hardware.
Registers	MATLAB persistent variables maps to the registers on hardware.
RAMs	MATLAB persistent arrays maps to RAM on hardware to reduce the area on the target device.
Multiplexers	Conditional statements maps to the multiplexers on hardware.
I/O Bits	Specifies the total bit count of the input and output bits.
Shifters	Bit shift operations maps to the shifters on the hardware.

Report Limitations

- The entry-point summary shows the individual elements of `varargin` and `varargout`, but the variables table does not show them.
- The report does not show full information for unrolled loops. It displays data types of one arbitrary iteration.
- The report does not show information about dead code.

See Also

More About

- "Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App" on page 11-5

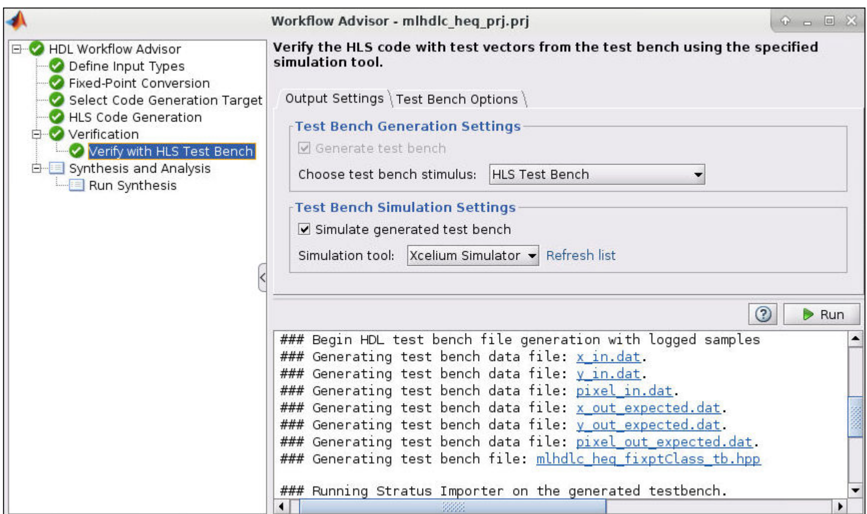
Verification

- “Verify HLS Code That Has an HDL Test Bench” on page 12-2
- “MATLAB to HLS Code Generation Options” on page 12-4
- “Floating-Point Tolerance Parameters” on page 12-8
- “Verify Generated HLS Code Using MATLAB Desktop Host” on page 12-9

Verify HLS Code That Has an HDL Test Bench

Simulate the generated High-Level Synthesis (HLS) design under test (DUT) with test vectors from the test bench files.

1. Start the MATLAB to HDL Workflow Advisor.
2. In the **HDL Workflow Advisor**, select **MATLAB to HLS** as the **Code generation workflow**.
3. Select the **Workflow** as **High Level Synthesis** and **Synthesis tool** as **Cadence Stratus** in the **Select Code Generation Target** step.
4. For **HDL Verification**, click **Verify with HDL Test Bench**.



5. Optionally, select **Simulate generated HDL test bench**. This option enables MATLAB to simulate the HLS test bench by using the HLS DUT.
6. Click **Run**.

   If the test bench and simulation executes without error, you see messages similar to these messages in the message pane:

```
### Begin TestBench generation.
Code generation successful.

### Collecting data...
### Begin HDL test bench file generation with logged samples
### Generating test bench data file: /tmp/mlhdlc_sfir/codegen/mlhdlc_sfir/hdlsrc/x_in.dat.
### Generating test bench data file: /tmp/mlhdlc_sfir/codegen/mlhdlc_sfir/hdlsrc/y_out_expected.dat.
### Generating test bench data file: /tmp/mlhdlc_sfir/codegen/mlhdlc_sfir/hdlsrc/delayed_xout_expected.dat.
### Generating test bench file: mlhdlc_sfir_fixptClass_tb.hpp

### Running Stratus Importer on the generated testbench.
### Working on mlhdlc_sfir_fixpt_bdw_import_log.txt as /tmp/mlhdlc_sfir/codegen/mlhdlc_sfir/hdlsrc/stratus_prj/mlhdl
### Stratus Importer successful.

### Simulating the design 'mlhdlc_sfir_fixpt' using 'Cadence Stratus'.
### Generating Simulation Report /tmp/mlhdlc_sfir/codegen/mlhdlc_sfir/hdlsrc/stratus_prj/mlhdlc_sfir_fixpt_sim_BEH_l
### Simulation successful.
### Elapsed Time: ' 64.6783' sec(s)
```

   If the errors appear in the message pane, fix the errors and click **Run**.

Additional Notes

- After clicking **Run**, the test bench generation starts generating the `.dat` files, such as `x_in.dat` and `y_out_expected.dat`, that contains the input and the expected output values to the top-level DUT and a test bench class file.
- When the test bench files are generated, the Stratus Importer is run by using Cadence Stratus HLS tool. It generates a Stratus HLS project in the current working folder.
- The simulation compiles and executes the generated HLS code and test bench with input values stored in the `x_in.dat` file. The simulation can verify the numeric match of the generated HLS code to the MATLAB code by comparing the HLS output with the expected output values stored in the `y_out_expected.dat` file.

See Also

More About

- “Verify Generated HLS Code Using MATLAB Desktop Host” on page 12-9
- “HLS Code Generation Report” on page 11-29

MATLAB to HLS Code Generation Options

In this section...

"MATLAB to HLS Code Generation" on page 12-4
"HLS Code Generation: Target Tab" on page 12-4
"HLS Code Generation: Coding Style Tab" on page 12-4
"HLS Code Generation: Clocks & Ports Tab" on page 12-5
"HLS Code Generation: Optimizations Tab" on page 12-6
"HLS Code Generation: Advance Tab" on page 12-6

MATLAB to HLS Code Generation

The **MATLAB to HLS Workflow** task in the HDL Workflow Advisor generates HLS code from MATLAB code. You can also simulate and verify the generated HLS code.

HLS Code Generation: Target Tab

Select target hardware and required outputs.

Input Parameters

Check HDL Conformance

Enable HDL conformance checking.

Default: Off

Generate Report

Enable to generate HLS code generation report. When you enable Generate report it automatically enables traceability.

Default: On

Enable traceability

Enable to trace the MATLAB code and the generated HLS code.

Default: On

HLS Code Generation: Coding Style Tab

Parameters that affect the style of the generated code.

Input Parameters

Preserve MATLAB code comments

Include MATLAB code comments in generated code.

Default: On

Include MATLAB source code as comments

Include MATLAB source code as comments in the generated code. The comments precede the associated generated code. Includes the function signature in the function banner.

Default: Off

Emit time/date stamp in the header

Enable to include time and date in output file header..

Default: Off

Comment in header

Specify comment lines in header of generated HLS and test bench files.

Default: None

Text entered in this field as a character vector generates a comment line in the header of the generated code. The code generator adds leading comment characters for the target language. When newlines or linefeeds are included in the text, the code generator emits single-line comments for each newline.

Complex real part postfix

Specify a character vector to append to the real part of complex signal names.

Default: `_re`

Complex imaginary part postfix

Specify a character vector to append to the imaginary part of complex signal names.

Default: `_im`

Reserved word postfix

Specify a character vector to append to value names, postfix values, or labels that are HLS reserved words.

Default: `_rsvd`

Create Regions automatically in the generated code

Enable to create regions in the generated HLS code. These regions can be used to perform design space exploration using the HLS tools. The generated region labels are stored inside `ml.tcl` file.

Default: Off

HLS Code Generation: Clocks & Ports Tab

Clocks & Ports settings

Input Parameters

Max number of I/O pins for FPGA deployment

Specify the maximum number of I/O pins for your target FPGA. If the DUT pin count in the generated code exceeds the value of this parameter, HDL Coder™ generates the message specified by the Check for DUT pin count exceeding I/O Threshold parameter in the Conformance Report.

Default: `5000`

Check for DUT pin count exceeding I/O Threshold

Specify the type of message to return if the DUT pin count exceeds the I/O threshold set by the Max number of I/O pins for FPGA deployment parameter.

Default: `Error`

HLS Code Generation: Optimizations Tab

Optimization settings

Input Parameters

Map persistent array variables to RAMs

Select to map persistent array variables to RAMs instead of mapping to shift registers.

Default: Off

Dependencies:

- **RAM Mapping Threshold**
- **Persistent variable names for RAM Mapping**

RAM Mapping Threshold

Specify the minimum RAM size required for mapping persistent array variables to RAMs.

Default: 256

Initialize Block Ram

Enable to initialize the RAM elements to zero.

Default: off

HLS Code Generation: Advance Tab

Advance settings

Input Parameters

Generate instantiable code for functions

Enable to generate the function hierarchy in the generated HLS code as in the MATLAB code.

Default: off

See Also

Functions
`coder.inline`

More About

- "Map Persistent Arrays to RAM" on page 13-2
- "HLS Code Generation Report" on page 11-29

Floating-Point Tolerance Parameters

Set the floating-point tolerance parameters in the **Verification** > **Verify with HLS Test Bench** > **Test Bench Options** tab of the HDL Workflow Advisor. Using the parameters in this tab, you can specify the floating-point tolerance strategy and the tolerance value.

Tolerance Strategy

When you generate HLS code from your MATLAB algorithm containing floating-point types, specify the floating-point tolerance strategy option.

Settings

Default: `Relative Error`

When you verify the generated code by using HLS Test bench, HDL Coder checks for the floating-point tolerance based on the relative error.

Dependency

Generate test bench enables this parameter. Enable this parameter by clicking **Generate Test Bench** in the **Verification** > **Verify with HLS Test Bench** > **Output Settings** pane of the HDL Workflow Advisor.

Limitation

ULP error is not supported for HLS code generation.

Command-Line Information

Parameter: `FPToleranceStrategy`
Type: character vector
Value: `"Relative Error"`
Default: `"Relative Error"`

Tolerance Value

Enter the tolerance value based on the floating-point tolerance check setting.

Settings

Default: `1e-07`

When you use this floating-point tolerance check setting, specify the tolerance value as a `double`.

Dependency

Generate test bench enables this parameter.

Command-Line Information

Parameter: `FPToleranceValue`
Type: `double`
Default: `1e-07`

Verify Generated HLS Code Using MATLAB Desktop Host

You can verify your generated HLS code by using the MATLAB® Desktop Host as the simulation tool. The verification ensures the generated HLS code for the DUT produces same results as the original MATLAB entry point for the same stimulus in the MATLAB test bench. Using MATLAB desktop host based MEX simulation eliminates the dependency on third-party EDA simulation tools for the verification of the generated HLS code. The following Discrete Cosine Transform (DCT) example shows how to generate HLS code and verify it using MATLAB Host.

Simulate the Generated HLS Code using MATLAB Desktop Host

The DCT example uses `mldhlc_dct` as the MATLAB® Design and `mlhdlc_dct_tb` as the MATLAB Test Bench to generate and simulate the code.

- Install the SystemC 2.3.3 library. When installing SystemC on the Windows® platform, use the `Release` option to build. You can download the SystemC library from accellera.org.
- Setup the SystemC library path by using `hdlsetuphlstoolpath`. For example:

```
hdlsetuphlstoolpath('ToolName','SystemC','SystemCIncludePath', ...
         'C:\Users\systemc-2.3.3\systemc-2.3.3\src','SystemCLibraryPath', ...
                 'C:\Users\systemc-2.3.3\systemc-2.3.3\msvc10\SystemC\x64\Release');
```

```
Adding the SystemC Include path to environment variable ML_SYSTEMC_INC_PATH :
C:\Users\systemc-2.3.3\systemc-2.3.3\src
Adding the SystemC Library path to environment variable ML_SYSTEMC_LIB_PATH:
C:\Users\systemc-2.3.3\systemc-2.3.3\msvc10\SystemC\x64\Release
```

- Simulate the generated HLS Code by using MATLAB Host as the simulation tool, with one of these options: the command Line Interface on page 12-9 or the HDL Workflow Advisor on page 12-10.

Note: To use MATLAB Host as the simulation tool, the synthesis tool should not be selected.

Command Line Interface

- Create an `hdl` configuration object and set the code generation workflow and the test bench name.

```
cfg = coder.config('hdl');
cfg.Workflow = "High Level Synthesis";
cfg.TestBenchName = "mlhdlc_dct_tb";
cfg.GenerateHDLTestBench = true;
cfg.TreatIOThresholdAs = "None";
```

- Enable the option to simulate generated code and specify the simulation tool.

```
cfg.SimulateGeneratedCode = true;
cfg.SimulationTool = "MATLAB Host";
```

- Generate HLS code.

```
codegen("mlhdlc_dct",'-config',cfg,'-report');
```

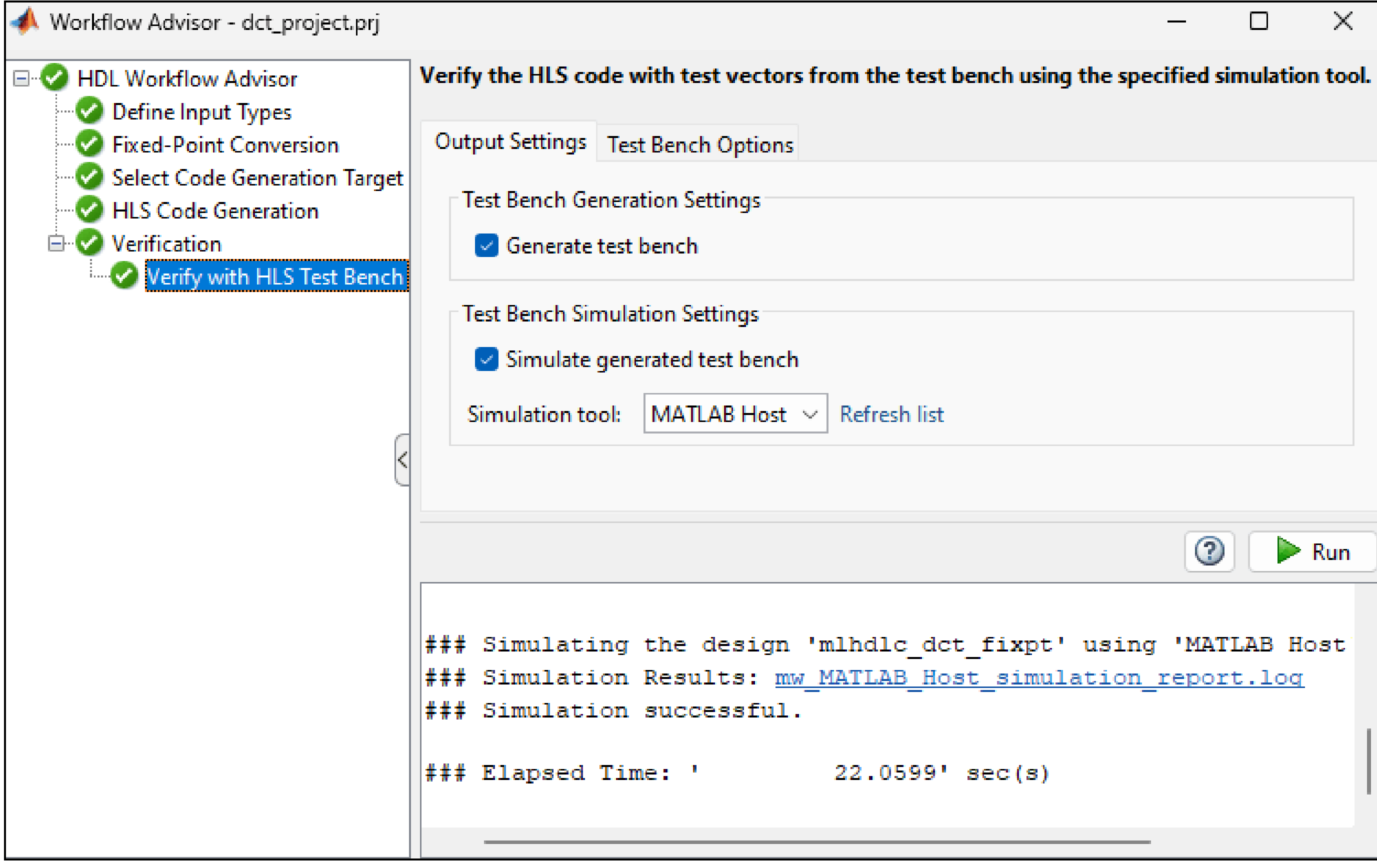
```
### Begin HLS Code Generation
### Working on mlhdlc_dctClass.hpp as mlhdlc_dctClass.hpp.
### Working on mlhdlc_dctModule.hpp as mlhdlc_dctModule.hpp.
### Generating Resource Utilization Report resource_report.html.

### Begin TestBench generation.
Code generation successful.

### Collecting data...
### Begin HDL test bench file generation with logged samples
### Generating test bench data file: input.dat.
### Generating test bench data file: output_expected.dat.
### Generating test bench file: mlhdlc_dctClass_tb.hpp
### Generating test bench module file: mlhdlc_dctModule_tb.hpp
Warning: C++ Compiler produced warnings. See the build log for further details.


### Simulating the design 'mlhdlc_dct' using 'MATLAB Host'.
### Simulation Results: mw_MATLAB_Host_simulation_report.log
### Simulation successful.

### Generating HDL Conformance Report mlhdlc_dct_hdl_conformance_report.html.
### HDL Conformance check complete with 0 errors, 1 warnings, and 0 messages.
### Code generation successful: View report
```

HDL Workflow Advisor

- Open the HDL Workflow Advisor using this command. A `dct_project.prj` file is created in the current folder.

```
coder -hdlcoder -new dct_project
```

- Select `mlhdlc_dct.m` as the **MATLAB function** and `mlhdlc_dct_tb.m` as the **MATLAB Test Bench**.
- To start the HDL Workflow Advisor, click the **Workflow Advisor** button.
- In **HDL Workflow Advisor** step, specify **Code Generation workflow** as **`MATLAB to HLS`**.
- In the **Verification** step, select **Generate test bench** and **Simulate generated test bench**. Specify **Simulation tool** as **`MATLAB Host`**.
- Run all the steps in the HDL Workflow Advisor to generate and simulate HLS code.



MATLAB Host Simulation Report

In the **Verify with HLS Test Bench** step, you can view the simulation results in `mw_MATLAB_Host_simulation_report.log`.

mw_MATLAB_Host_simulation_report.log

```
1  ## MATLAB 24.1.0.2470319 (R2024a) Prerelease Update 2 ##
2  ## Simulation of generated HLS SystemC code on MATLAB Desktop win64 Host Machine
3  ## HLS Synthesis Tool Selected: No Tool Selected
4  ## HLS Simulation Tool Selected: MATLAB Host
5  ## HLS SystemC Library path: C:\Users\systemc-2.3.3\systemc-2.3.3\msvc10\SystemC\x64\Release
6  ## HLS SystemC Include path: C:\Users\systemc-2.3.3\systemc-2.3.3\src
7  0 s: Info:  : SystemC 2.3.3-Accellera --- Jan 11 2024 16:02:55
8  0 s: Info:  : Copyright (c) 1996-2018 by all Contributors,
9  ALL RIGHTS RESERVED
10
11 89305 ns: Info: #####: #################################################
12 89305 ns: Info: Total Testpoints : 14884
13 89305 ns: Info: Tests Passed     : 14884
14 89305 ns: Info: Verification Result:     TEST COMPLETED (PASSED)
15 89305 ns: Info: #####: #################################################
16 89305 ns: Info: /OSCI/SystemC: Simulation stopped by user.
```

HLS Optimizations

- "Map Persistent Arrays to RAM" on page 13-2
- "Pipelining of for-Loops" on page 13-4
- "Map Persistent Variables to RAM for Histogram Equalization" on page 13-9
- "Create a Line Buffer Interface for SystemC Code Generation" on page 13-16

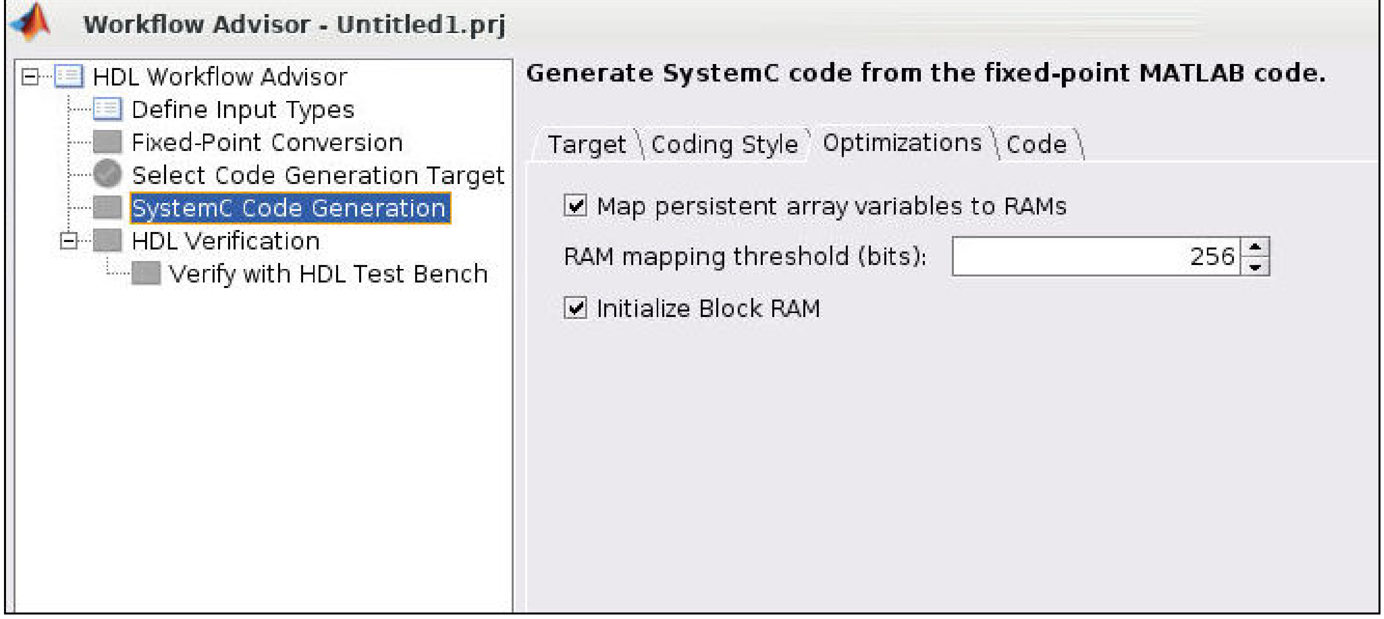
Map Persistent Arrays to RAM

In this section...
“Enable RAM Mapping” on page 13-2
“RAM Mapping Requirements for Persistent Arrays” on page 13-2

To map the persistent variables to RAMs in the generated High-Level Synthesis (HLS) code use the **RAM Mapping** optimization. Without this optimization, the variables are mapped to registers. RAM mapping is an area optimization. It reduces the area of your design in the target hardware.

Enable RAM Mapping

1. In the HDL Workflow Advisor, select **MATLAB to HLS Workflow > HLS Code Generation > Optimizations** tab.
2. Select the **Map persistent array variables to RAMs** option.
3. Set the **RAM mapping threshold** to the size (in bits) of the smallest persistent array that you want to map to RAM.



RAM Mapping Requirements for Persistent Arrays

This table lists a summary of the RAM mapping behavior for persistent arrays.

Map Persistent Array Variables to RAMs Setting	Mapping Behavior
on	Map to RAM in the generated HLS code.
off	Map to registers in the generated HLS code.

Additional Notes

- The large persistent arrays whose size in bits is greater than or equal to **RAM mapping threshold** are mapped to RAM. The size in bits is calculated as `NumElements` * `WordLength`.
  - `NumElements` is the number of elements in the array.
  - `WordLength` is the number of bits that represent the data type of the array.
- The list of RAM variables is populated inside the `ml.tcl` metadata file generated during **HLS Code Generation**. This metadata file is read by the Stratus HLS tool during project creation and maps these variables to RAM.
- If **Initialize Block RAM** option is turned off, the RAM variables that have a zero initial value are not initialized inside `initialize_ram_vars()` method.
- If the initial value of the variables is nonzero, then those RAM variables are initialized inside the `initialize_ram_vars()` method, irrespective of the **Initialize Block RAM** option value.

See Also

More About

- "Map Persistent Variables to RAM for Histogram Equalization" on page 13-9

Pipelining of for-Loops

Pipelining allows concurrent execution of multiple iterations. The next iteration of a loop can begin execution before the previous iteration completes its execution. Pipelining optimises the execution speed and improves the throughput of the code at the expense of increased resources.

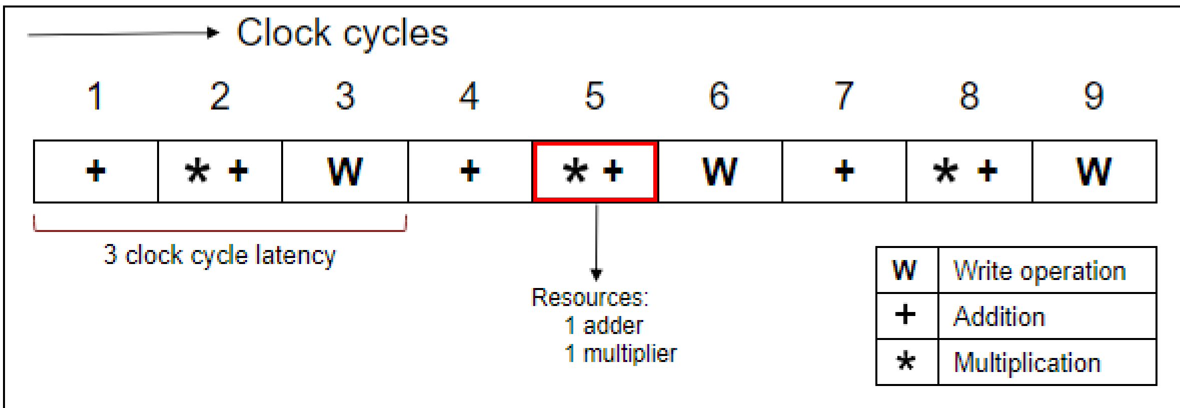
In MATLAB you can pipeline `for`-loops by using `coder.hdl.loopspec('pipeline')` or `coder.hdl.loopspec('pipeline',initiation_interval)`.

To understand the concept of pipelining of `for`-loops, consider the following MATLAB code. It consists of a `for`-loop and a persistent array `arr`, mapped to RAM during code generation.

```
for i = 1:20
    tmp = fi((a + b) * c + tmp, 0, 32, 0, hdlfimath);
    arr(i) = tmp;
end
```

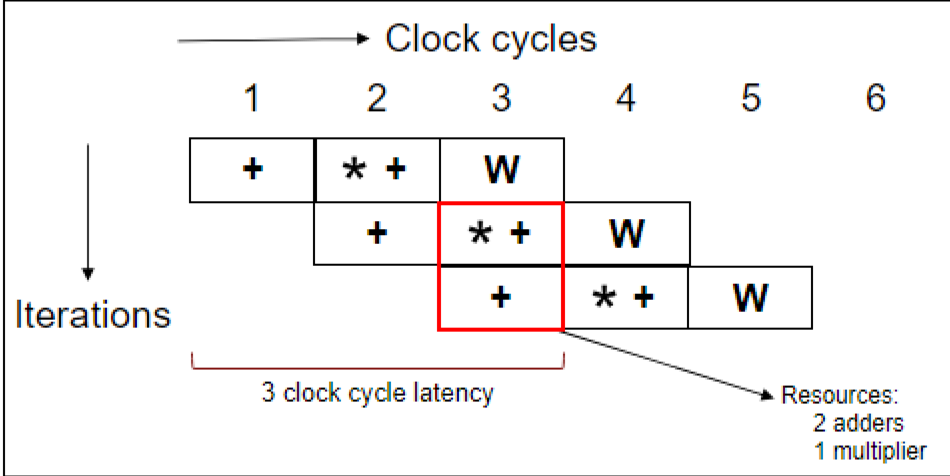
- **Non-pipelined for-loop**

  In a non-pipelined loop, all the iterations of the `for`-loop are scheduled serially. The next iteration executes after the previous iteration completes its execution. Also, there is no overlap in the execution of the iterations in a non-pipelined loop. The following diagram shows the schedule of a non-pipelined `for`-loop.



- **Pipelined for-loop**

  In a pipelined loop, the next iteration start its execution with a gap of an initiation interval. This can lead to overlap in the execution of the iterations. The initiation interval represents the number of clock cycles before the start of the next iteration of the `for`-loop. The following diagram shows the schedule of a pipelined `for`-loop with an initiation interval of `1`.



In the above schedules, a non-pipelined `for`-loop takes 9 clock cycles to complete 3 iterations whereas a pipelined `for`-loop takes only 5 clock cycles for 3 iterations. This shows the impact of pipelining in improving the throughput of the code at the expense of additional resources.

Issues with Pipelined for-Loops

Pipelining enables resource sharing between multiple iterations, thus leading to the following hazards:

1 Conflicting Read and Write Operations

If a pipelined `for`-loop is performing read and write operations in the same iteration, it can lead to overlapping of these operations. The memory read operation in the next iteration can be scheduled before the memory write operation in the current iteration. This leads to reading of incorrect values between the `for`-loop iterations of the shared resources.

Consider the following MATLAB code and its testbench. The code consists of a pipelined `for`-loop with an `initiation_interval` of `1`, that performs both read and write operations in each iteration.

MATLAB Code	Generated HLS Code
`function out = f(in)`	`class f_fixptClass`
`    persistent arr1;`	`{`
`    if isempty(arr1)`	`    public:`
`        arr1 = zeros(1,102);`	`        sc_uint<3> f_fixpt_arr1[102];`
`    end`	`        void f_fixpt_initialize_ram_vars()`
`    coder.hdl.loopspec('pipeline',1);`	`        {`
`    for i = 4:100`	`            int32_T t_0;`
`        y = arr1(i-3);`	`            L1:`
`        arr1(i) = in;`	`            for (t_0 = 0; t_0 < 102; t_0 = t_0 + 1) {`
`    end`	`                f_fixpt_arr1[t_0] = sc_uint<3>(0.0);`
`    out = y;`	`            }`
`end`	`        }`
	`        sc_uint<3> f_fixpt(sc_uint<3> in)`
	`        {`
`% MATLAB Testbench`	`            sc_uint<3> out;`
`a = fi(4,0,3,0);`	`            L2:`
`for i = 1:100`	`            for (int32_T i = 0; i < 97; i = i + 1) {`
`    out(i) = f(a);`	`                HLS_PIPELINE_LOOP( HARD_STALL, 1, "L2" );`
`end`	`                out = f_fixpt_arr1[i];`
	`                f_fixpt_arr1[i + 3] = in;`
	`            }`
	`            return out;`
	`        }`
	`};`

During synthesis, the high level synthesis (HLS) tool throws an error stating: `Unable to guarantee the safety of the schedule in the pipelined loop`.

Solution:

To overcome the conflicting read and write operations, you can use any one of the following solution based on your design requirement.

- Use `coder.hdl.arraydistance(arr1,'max',1)`, which ensures that read and write operations enclosed in the `for`-loop are separated with a maximum array distance of one clock cycle.

```
coder.hdl.loopspec('pipeline',1);
for i = 4:100
    coder.hdl.arraydistance(arr1,'max',1);
    y = arr1(i-3);
    arr1(i) = in;
end
```

- Alternatively, you can update the value of `initiation_interval` to 2 in the `coder.hdl.loopspec` pragma.

```
coder.hdl.loopspec('pipeline',2);
for i = 4:100
    y = arr1(i-3);
    arr1(i) = in;
end
```

2 Limited Memory Access

In many designs, it is common to have a single loop performing two or more accesses to a single memory (RAM). If such a loop is pipelined with `initiation_interval` as 1, these memory

accesses occur simultaneously during each clock cycle. Dual-port RAMs have two ports at maximum, so at most two parallel independent accesses can be scheduled to the memory in a single clock cycle.

However, with a pipelined `for`-loop, more than two memory accesses can occur in a single clock cycle. This scenario leads to a limited memory access issue.

Consider the following MATLAB code and its testbench. It has a persistent array `arr1` that is mapped to RAM. The `for`-loop is pipelined with an `initiation_interval` of `1`, and three write operations are performed on the RAM mapped variable `arr1` inside the `for`-loop body.

MATLAB Code

```
function out = f(in1, in2)
    persistent arr1;
    if isempty(arr1)
        arr1 = zeros(1,102);
    end

    coder.hdl.loopspec('pipeline',1);
    for i = 1:100
        arr1(i) = in1+in2;
        arr1(i+1) = in1*2;
        arr1(i+2) = in1+in2-2;
    end

    out = sum(arr1);
end


% MATLAB Testbench
a = int8(4);
b = int8(5);
out = f(a,b);
```

Generated HLS Code

```
class f_fixptClass
{
 public:
  sc_uint<4> f_fixpt_arr1[102];
  void f_fixpt_initialize_ram_vars()
  {
    int32_T t_0;
   L1:
    for (t_0 = 0; t_0 < 102; t_0 = t_0 + 1) {
      f_fixpt_arr1[t_0] = sc_uint<4>(0.0);
    }
  }
  sc_uint<10> f_fixpt(sc_uint<3> in1, sc_uint<3> in2)
  {
    sc_uint<10> out;
    sc_uint<11> Y;
   L2:
    for (int32_T i = 0; i < 100; i = i + 1) {
      HLS_PIPELINE_LOOP( HARD_STALL, 1, "L2" );
      f_fixpt_arr1[i] = (sc_uint<4>)in1 + (sc_uint<4>)in2;
      f_fixpt_arr1[i + 1] = (sc_uint<4>)(in1 * sc_uint<2>(2.0))
      f_fixpt_arr1[i + 2] = ((sc_uint<4>)in1 + (sc_uint<4>)in2)
        sc_uint<5>(2.0);
    }

    Y = (sc_uint<11>)f_fixpt_arr1[0];
   L3:
    for (int32_T k = 0; k < 101; k = k + 1) {
      Y = (sc_uint<11>)((sc_uint<12>)Y + (sc_uint<12>)f_fixpt_a
    }

    out = (sc_uint<10>)Y;
    return out;
  }
};
```

During synthesis, the HLS tool throws an error stating: `Unable to produce a valid schedule, found 1 Violation(s)`.

Solution:

To overcome the limited memory access create a pipelined `for`-loop with an `initiation_interval` of 2. This action ensures that two memory accesses occur in each iteration of the `for`-loop.

```
coder.hdl.loopspec('pipeline',2);
for i = 1:100
    arr1(i) = in1+in2;
    arr1(i+1) = in1*2;
    arr1(i+2) = in1+in2-2;
end
```

See Also

Functions
`coder.hdl.loopspec` | `coder.hdl.arraydistance`

More About

- “Map Persistent Arrays to RAM” on page 13-2
- “For-Loop Best Practices for HDL Code Generation” on page 1-73
- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14

Map Persistent Variables to RAM for Histogram Equalization

This example shows how to use the RAM mapping optimization in HDL Coder™ to map persistent matrix variables to block RAMs in hardware.

Introduction

In MATLAB, you can easily create, access, modify, and manipulate matrices.

When processing such MATLAB® code, HDL Coder maps these matrices to wires or registers in hardware. For example, local temporary matrix variables are mapped to wires, whereas persistent matrix variables are mapped to registers.

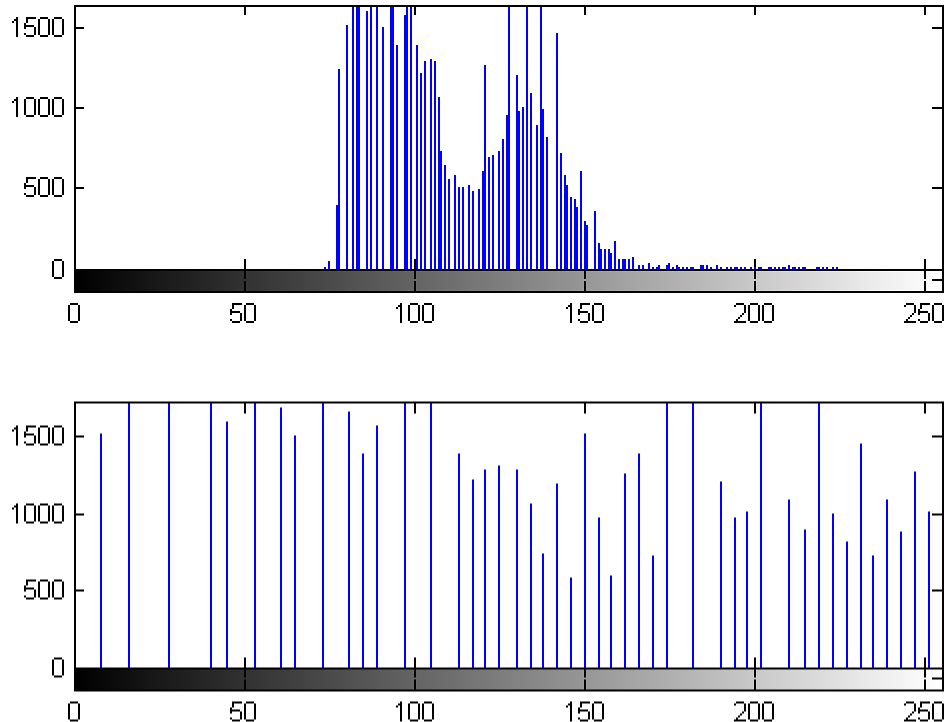
The latter tends to be an inefficient mapping when the matrix size is large because the number of register resources available is limited. It also complicates synthesis, placement, and routing.

Modern FPGAs feature block RAMs that are designed to have large matrices. HDL Coder takes advantage of this feature and maps matrices to block RAMs to improve area efficiency. For certain designs, mapping these persistent matrices to RAMs is mandatory if the design is to be realized. State-of-the-art synthesis tools might not be able to synthesize designs when large matrices are mapped to registers, whereas the size is more manageable when the same matrices are mapped to RAMs.

Algorithm

The Histogram Equalization algorithm enhances the contrast of images by transforming the values in an intensity image so that the histogram of the output image is approximately flat.

```
I = imread('pout.tif');
J = histeq(I);
subplot(2,2,1);
imshow( I );
subplot(2,2,2);
imhist(I)
subplot(2,2,3);
imshow( J );
subplot(2,2,4);
imhist(J)
```



MATLAB Design

The MATLAB code and MATLAB test bench used in the example is a Histogram Equalization algorithm.

```
design_name = 'mlhdlc_heq';
testbench_name = 'mlhdlc_heq_tb';
```

Review the MATLAB design

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% heq.m
% Histogram Equalization Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out, y_out, pixel_out] = ...
    mlhdlc_heq(x_in, y_in, pixel_in, width, height)

%   Copyright 2011-2015 The MathWorks, Inc.

persistent histogram
persistent transferFunc
persistent histInd
persistent cumSum

if isempty(histogram)
```

```
    histogram = zeros(1, 2^14);
    transferFunc = zeros(1, 2^14);
    histInd = 0;
    cumSum = 0;
end

% Figure out indexes based on where we are in the frame
if y_in < height && x_in < width % valid pixel data
    histInd = pixel_in + 1;
elseif y_in == height && x_in == 0 % first column of height+1
    histInd = 1;
elseif y_in >= height % vertical blanking period
    histInd = min(histInd + 1, 2^14);
elseif y_in < height % horizontal blanking - do nothing
    histInd = 1;
end

%Read histogram (must be outside conditional logic)
histValRead = histogram(histInd);

%Read transfer function (must be outside conditional logic)
transValRead = transferFunc(histInd);

%If valid part of frame add one to pixel bin and keep transfer func val
if y_in < height && x_in < width
    histValWrite = histValRead + 1; %Add pixel to bin
    transValWrite = transValRead; %Write back same value
    cumSum = 0;
elseif y_in >= height %In blanking time index through all bins and reset to zero
    histValWrite = 0;
    transValWrite = cumSum + histValRead;
    cumSum = transValWrite;
else
    histValWrite = histValRead;
    transValWrite = transValRead;
end

%Write histogram (must be outside conditional logic)
histogram(histInd) = histValWrite;

%Write transfer function (must be outside conditional logic)
transferFunc(histInd) = transValWrite;

pixel_out = transValRead;
x_out = x_in;
y_out = y_in;
```

```
type(testbench_name);
```

```
%Test bench for Histogram Equalization

%   Copyright 2011-2018 The MathWorks, Inc.

testFile = 'mlhdlc_img_peppers.png';
imgOrig = imread(testFile);
[height, width] = size(imgOrig);
```

```
imgOut = zeros(height,width);
hBlank = 20;
% make sure we have enough vertical blanking to filter the histogram
vBlank = ceil(2^14/(width+hBlank));

for frame = 1:2
    disp(['working on frame: ', num2str(frame)]);
    for y_in = 0:height+vBlank-1
        %disp(['frame: ', num2str(frame), ' of 2, row: ', num2str(y_in)]);
        for x_in = 0:width+hBlank-1
            if x_in < width && y_in < height
                pixel_in = double(imgOrig(y_in+1, x_in+1));
            else
                pixel_in = 0;
            end

            [x_out, y_out, pixel_out] = ...
                mlhdlc_heq(x_in, y_in, pixel_in, width, height);

            if x_out < width && y_out < height
                imgOut(y_out+1,x_out+1) = pixel_out;
            end
        end
    end

    % normalize image to 255
    imgOut = round(255*imgOut/max(max(imgOut)));

    figure(1)
    subplot(2,2,1); imshow(imgOrig, [0,255]);
    title('Original Image');
    subplot(2,2,2); imshow(imgOut, [0,255]);
    title('Equalized Image');
    subplot(2,2,3); histogram(double(imgOrig(:)),2^14-1);
    axis([0, 255, 0, 1500])
    title('Histogram of original Image');
    subplot(2,2,4); histogram(double(imgOut(:)),2^14-1);
    axis([0, 255, 0, 1500])
    title('Histogram of equalized Image');
end
```

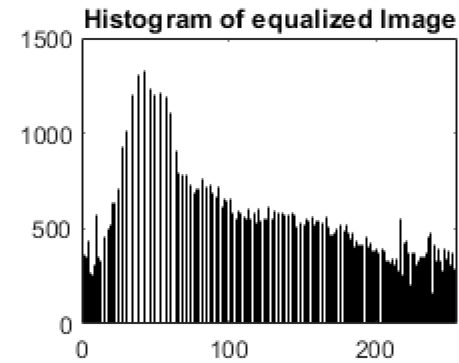
Simulate the Design

Before code generation, simulate the design by using the HDL test bench to make sure that there are no run-time errors.

```
mlhdlc_heq_tb
```

```
working on frame: 1
working on frame: 2
```

Histogram of original Image



Create HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_heq_prj
```

Add the file `mlhdlc_heq.m` to the project as the MATLAB Function and `mlhdlc_heq_tb.m` as the MATLAB Test Bench.

For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14 or “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5. For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14 or “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5. For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14 or “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.

HLS Code Generation by Using RAM Mapping Optimization

To generate HLS code from a MATLAB design:

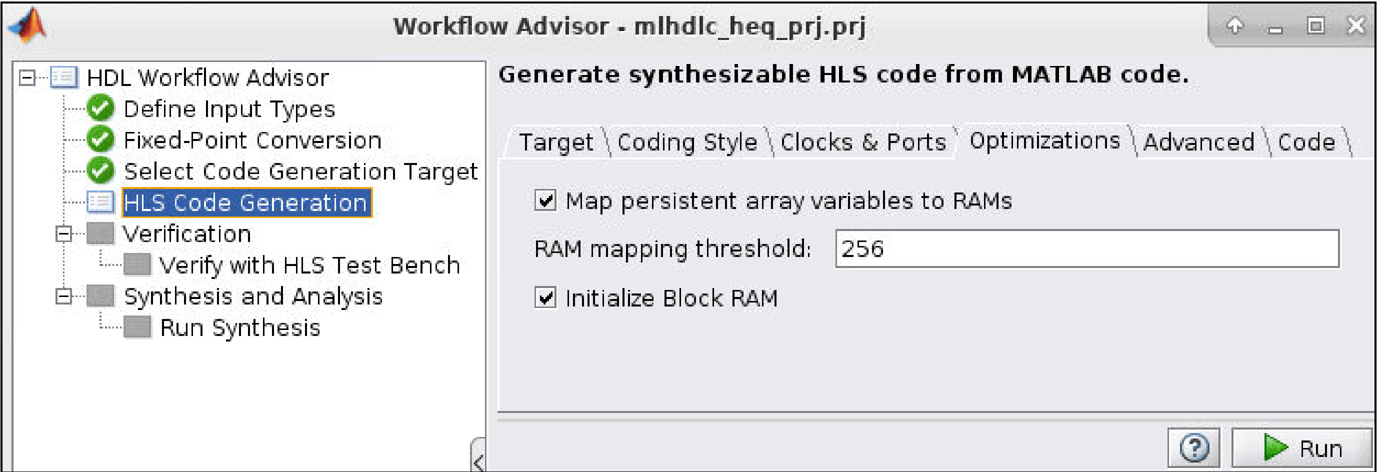
1. At the MATLAB command line, setup the high-level synthesis (HLS) tool path for HLS code generation by using the function `hdlsetuphlstoolpath`.

2. Start the HDL Workflow Advisor.

3. Select as **MATLAB to HLS** as the **Code Generation Workflow**.

4. In **Select Code Generation Target** step, select **Workflow** as **High Level Synthesis** and **Synthesis tool** as **Cadence Stratus**.

5. To map persistent variables to block RAMs in the generated code. Select **Map persistent array variables to RAMs** check box in the **Optimizations** tab of the **HLS Code Generation** step.



6. Right-click **HLS Code Generation** and choose the option **Run to selected task** to run all the steps from the beginning through the HLS code generation.

Examine the generated HLS code by clicking the hyperlinks in the Code Generation Log window.

Examine the Generated Code

Examine the messages in the log window to see the RAM files generated with the design.

```
### Begin HLS Code Generation
### Working on mlhdlc_heq_fixptClass.hpp as mlhdlc_heq_fixptClass.hpp.
### Working on ml.tcl as ml.tcl.
### Generating Resource Utilization Report resource_report.html.
### Generating HDL Conformance Report mlhdlc_heq_fixpt_hdl_conformance_report.html.
### HDL Conformance check complete with 0 errors, 0 warnings, and 0 messages.
Code generation successful.

### Elapsed Time: '         14.6244' sec(s)
```

Examine the Resource Report

Examine the generated resource report, which shows the number of RAMs inferred, by following the `mlhdlc_heq_fixpt_syn_results.txt` link in the generated code window.

The **Allocation Report** highlights the variables mapped to RAM.

```
       +--------------------------------------------------------------------------+
       |                                                                          |
00803: | Allocation Report for all threads:                                       |
00805: |                                         Area/Instance                    |
00805: |                                     ------------------------      Total |
00805: |                   Resource  Count    Seq(#FF)     Comb    BB       Area |
00805: | ---------------------------  -----  ----------  ------  ----   ------- |
00807: |               mux_14bx2i0c      3                 43.5            130.4 |
00807: |        dut_Add_17Ux11U_17U_4      1               120.4            120.4 |
00807: |          dut_N_Mux_17_2_5_4      2                 40.7             81.4 |
00807: |     dut_LessThan_16Sx16S_1U_4      1                 72.5             72.5 |
00807: |       dut_LessThan_9Ux9U_1U_4      2                 34.2             68.4 |
00807: | dut_GreaterThan_16Ux15U_1U_4      1                 61.9             61.9 |
00807: |         dut_Add_15Ux1U_16U_4      1                 61.9             61.9 |
.
.
.
00807: |               mux_17bx2i0c      1                 52.8             52.8 |
00807: |           dut_And_1Ux1U_1U_1      1                  8.9              8.9 |
00807: |       dut_OrReduction_9U_1U_4      1                  5.8              5.8 |
00807: |           dut_Xor_1Ux1U_1U_1      1                  4.4              4.4 |
00807: |              dut_Not_1U_1U_1      1                  4.1              4.1 |
00807: |            dut_Or_1Ux1U_1U_4      3                  1.4              4.1 |
00807: |                mux_1bx3i1c      1                  3.8              3.8 |
00807: |           dut_N_Muxb_1_2_9_4      1                  2.4              2.4 |
00807: |           dut_And_1Ux1U_1U_4      1                  1.4              1.4 |
00807: |              dut_Not_1U_1U_4      1                  0.7              0.7 |
00807: |             DPRAM_16384X11      1                          ?         0.0 |
00807: |             DPRAM_16384X17      1                          ?         0.0 |
00808: |                  registers     23                                        |
00809: |              register bits    106     7.5(1)      0.9            884.4 |
02604: |             estimated cntrl      1                 37.2             37.2 |
00811: | ---------------------------------------------------------------------- |
00812: |                 Total Area          791.4(106)  1437.7   0.0     2229.1 |
       |                                                                          |
       +--------------------------------------------------------------------------+
```

Additional Notes on RAM Mapping

- MATLAB functions can have any number of RAM matrices.
- All matrix variables in MATLAB that are declared persistent and meet the threshold criteria get mapped to RAMs.

Create a Line Buffer Interface for SystemC Code Generation

In this example, you will learn how to generate SystemC™ code using a line buffer interface from MATLAB® code. The MATLAB code implements a simple Sobel filter, which is used for edge detection in images.

The Sobel filter edge detection algorithm operates on grey scale image and produces an image that highlights the high spatial frequency regions corresponding to edges in the input image.

The example highlights these aspects of generating SystemC code from MATLAB Code:

- Line buffer interface - Use the `coder.hdl.interface` pragma in your MATLAB code to implement the line buffer interface. The line buffer interface supports only two-dimensional matrices for both working sets and the interface.
- Input matrix - The entry point function `hls_sobel` in the MATLAB code accepts two-dimensional matrices as input and returns the pixel value.
- Generating 2D working sets - Use the `hdl.WorkingSet` class in your MATLAB testbench to generate two-dimensional working sets from the input image.

Sobel Filter MATLAB Design

The `hls_sobel` function operates on a given working set by applying the Sobel filter kernel. The function returns the output pixel that corresponds to the position of the working set in the input image.

```
dbtype('hls_sobel')
```

```
1     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2     %% Sobel edge detection using Line buffer interface in HLS.
3     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4     % This example shows how to generate a line buffer interface from the MATLAB(R)
5     % design using MATLAB to HLS workflow. This MATLAB design implements the
6     % Sobel edge detection algorithm by the following steps.
7     %
8     % 1. Compute the horizontal and vertical gradients, gh and gv by convolving
9     %    the working set with the Sobel kernel and its transpose.
10    %      Sobel kernel = [-1 -2 -1
11    %                       0  0  0
12    %                       1  2  1];
13    % 2. Compute the gradient y as follows:
14    %      y = floor(abs(gh) + abs(gv)) / 4;
15    % 3. If the gradient is greater than the threshold, the pixel is considered
16    %    an edge pixel. In this example, the threshold is chosen as 255.
17
18    %#codegen
19
20    %   Copyright 2023 The MathWorks, Inc.
21    function y = hls_sobel(ws)
22
23    ORIGIN = [2 2];
24    IMAGE_SIZE = [480 752];
25    CONSTANT_FILL_VALUE = 0;
```

```
26
27    % Specify the line buffer properties using coder.hdl.interface pragma.
28    coder.hdl.interface(ws, 'Line Buffer', IMAGE_SIZE, 'Origin', ORIGIN, 'FillValue', CONSTANT_
29
30    ws1 = reshape(ws, [1 3*3]);
31
32    % Sobel kernel
33    Gx = [-1 0 1 -2 0 2 -1 0 1];
34    Gy = [-1 -2 -1 0 0 0 1 2 1];
35
36    % Compute horizontal and vertical gradients
37    gh = sum(ws1 .* Gx);
38    gv = sum(ws1 .* Gy);
39
40    % Compute the gradient.
41    y = floor(abs(gh) + abs(gv)) / 4;
42
43    % Determine whether the pixel is on the edge or not.
44    y = min(y,255);
45
46    end
```

Sobel Filter MATLAB Test Bench

In MATLAB testbench, use the `hdl.WorkingSet` class to generate working sets from the input image and pass them to the `hls_sobel` function. The dimensions of each working set matrix are determined by the `WINDOW_SIZE` parameter defined in the testbench.

```
dbtype('hls_sobel_tb')
```

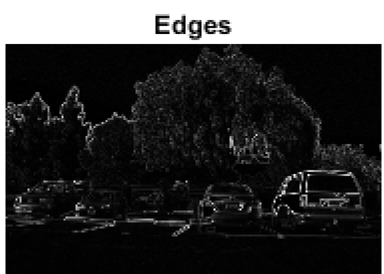
```
1     %
2
3     %   Copyright 2023 The MathWorks, Inc.
4
5     clear;
6
7     % Read the image for which the edge detection needs to be performed.
8     image3d = double(imread('mlhdlc_img_yuv.png'));
9     image = image3d(:,:,1);
10
11    [IMAGE_HEIGHT, IMAGE_WIDTH] = size(image);
12    WINDOW_SIZE = [3 3];  % Size of the working set/Window size
13    ORIGIN = [2 2];    % Working set origin
14    % Constant value to fill the pixels falling outside the image in the working set.
15    CONSTANT_FILL_VALUE = 0;
16
17    y = zeros(IMAGE_HEIGHT,IMAGE_WIDTH);
18
19    % Create the working set object. This helps in creating the 2-D working sets
20    % of specified size and origin from the input image.
21    ws = hdl.WorkingSet(image, WINDOW_SIZE, 'Origin', ORIGIN, 'FillValue', CONSTANT_FILL_VALUE
22
23    for i = 1:IMAGE_HEIGHT
24        for j = 1:IMAGE_WIDTH
25            % Get the working set at the pixel (i,j) in the image.
26            workingSet = ws.getWorkingSet(i,j);
27            y(i,j) = hls_sobel(workingSet);
```

```
28          end
29      end
30
31      % Plot the original image and the extracted edges.
32      figure('Name', [mfilename, '_plot'])
33
34      subplot(1,2,1);
35      imshow(uint8(image));
36      title('Original Image');
37
38      subplot(1,2,2);
39      imshow(uint8(y));
40      title('Edges');
```

Simulate the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

```
hls_sobel_tb
```



Generate SystemC Code by Using HDL Workflow Advisor

To generate SystemC code from the MATLAB design:

1. At the MATLAB command line, set up the high-level synthesis (HLS) tool path for SystemC code generation by using the function `hdlsetuphlstoolpath`.

2. Create an HDL Coder project by adding the `hls_sobel.m` design file and the `hls_sobel_tb.m` test bench file to the HDL workflow advisor.

3. Launch the HDL Workflow Advisor. Select **Code Generation Workflow** as `MATLAB to SystemC`.

4. In the **Select Code Generation Target** step, from the **Synthesis tool** list, select `Cadence Stratus`.

5. Right-click the **SystemC Code Generation** task and select the **Run to selected** task to run all the steps from the beginning through the SystemC code generation.

For more information, see “Get Started with MATLAB to High-Level Synthesis Workflow Using HDL Coder App” on page 11-5.

Examine the Generated Code

You can examine the generated SystemC code by clicking the hyperlinks in the **SystemC Code Generation** step log window.

```
### Begin SystemC Code Generation
### Working on hls_sobel_fixptClass.hpp as hls_sobel_fixptClass.hpp.
### Working on ml.tcl as ml.tcl.
### Generating Resource Utilization Report resource_report.html.
### Generating HDL Conformance Report hls_sobel_fixpt_hdl_conformance_report.html.
### HDL Conformance check complete with 0 errors, 1 warnings, and 0 messages.
### Code generation successful: View report
### Elapsed Time: '          24.3656' sec(s)
```

The line buffer properties specified in the `coder.hdl.interface` pragma are populated in the `ml.tcl` metadata file. These are read by the Stratus HLS tool to create appropriate line buffers.

ml.tcl

```
1   set BDW_IMPORT_ML_DUT_CLASS hls_sobel_fixptClass
2   set BDW_IMPORT_ML_DUT_FUNC hls_sobel_fixpt
3   set BDW_IMPORT_ML_DUT_INIT_FUNC {}
4   set BDW_IMPORT_ML_DUT_FUNC_INPUT_ARGS { ws }
5   set BDW_IMPORT_ML_DUT_FUNC_OUTPUT_ARGS { y }
6   set BDW_IMPORT_ML_DUT_ARRAYS_TO_SPMEMS {}
7   set BDW_IMPORT_ML_DUT_ARRAYS_TO_DPMEMS { }
8   set BDW_IMPORT_ML_DUT_MEM_INIT_FUNCS {}
9   set BDW_IMPORT_ML_DUT_FUNC_STABLE_ARGS { }
10  set BDW_IMPORT_ML_CLOCK_FREQ {0}
11  set BDW_IMPORT_ML_DUT_FUNC_INTERFACE { ws }
12  set BDW_IMPORT_ML_DUT_FUNC_INTERFACE_IMAGE_SIZE {480 752}
13  set BDW_IMPORT_ML_DUT_FUNC_INTERFACE_WORKING_SET_SIZE {3 3}
14  set BDW_IMPORT_ML_DUT_FUNC_INTERFACE_ORIGIN {1 1}
15  set BDW_IMPORT_ML_DUT_FUNC_INTERFACE_BOUNDARY_FILL_CONDITION {ConstantFill}
16  set BDW_IMPORT_ML_DUT_FUNC_INTERFACE_CONSTANT_FILL {0}
```

See Also

- `coder.hdl.interface`
- `hdl.WorkingSet`

Related Topics

- “Get Started with MATLAB to High-Level Synthesis Workflow Using the Command Line Interface” on page 11-14
- “Guidelines for Writing MATLAB Code to Generate Efficient HDL and HLS Code” on page 1-66
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “MATLAB to HLS Code Generation Options” on page 12-4

HDL Code Generation from Simulink

Model Design for HDL Code Generation

- “Signal and Data Type Support” on page 14-3
- “Use Simulink Templates for HDL Code Generation” on page 14-8
- “Generate DUT Ports for Tunable Parameters” on page 14-18
- “Generate Parameterized Code for Referenced Models” on page 14-21
- “Generating HDL Code for Subsystems with Array of Buses” on page 14-22
- “Generate HDL Code with Record or Structure Types for Bus Signals” on page 14-25
- “Implement Control Signal-Based Mathematical Functions by Using HDL Coder” on page 14-31
- “Implement Sqrt Block with Control Signals” on page 14-33
- “Implement Reciprocal Block with Control Signals” on page 14-37
- “Implement rSqrt Block with Control Signals” on page 14-41
- “Implement Divide Block with Control Signals” on page 14-45
- “Implement Sine and Cosine Block with Control Signals” on page 14-49
- “Implement Atan2 Block with Control Signals” on page 14-53
- “Using ForEach Subsystems in HDL Coder” on page 14-57
- “Generate HDL Code for Blocks Inside For Each Subsystem” on page 14-61
- “Field-Oriented Control of a Permanent Magnet Synchronous Machine” on page 14-66
- “Model and Debug Test Point Signals with HDL Coder” on page 14-71
- “Optimize Generated HDL Code for Multirate Designs with Large Rate Differentials” on page 14-80
- “Getting Started with HDL Coder Native Floating-Point Support” on page 14-88
- “Numeric Considerations for Native Floating-Point” on page 14-92
- “ULP Considerations of Native Floating-Point Operators” on page 14-95
- “Latency Values of Floating-Point Operators” on page 14-99
- “Latency Considerations with Native Floating Point” on page 14-104
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111
- “Floating Point Support: Field-Oriented Control Algorithm” on page 14-118
- “Design Model by Using HDL Coder Native Floating Point and Intel Hard Floating Point” on page 14-125
- “Verify the Generated Code from Native Floating-Point” on page 14-133
- “Simulink Blocks Supported by Using Native Floating Point” on page 14-137
- “Synthesis Benchmark of Common Native Floating Point Operators” on page 14-143
- “Supported Data Types and Scope” on page 14-155
- “Supported Synthesizable RTL Constructs and keywords in HDL Coder” on page 14-157
- “Import Verilog Code and Generate Simulink Model” on page 14-165
- “Supported Verilog Constructs for HDL Import” on page 14-169

- “Verilog Dataflow Modeling with HDL Import” on page 14-174
- “Simulate and Generate HDL Code for the Float Typecast Block” on page 14-185
- “Generate Simulink Model from CORDIC Atan2 Verilog Code” on page 14-187

Signal and Data Type Support

In this section...

"Buses" on page 14-3
"Enumerations" on page 14-4
"Matrices" on page 14-4
"Unsupported Signal and Data Types" on page 14-7

HDL Coder supports code generation for Simulink signal types and data types with a few special cases.

Buses

If your DUT or other blocks in your model have many input or output signals, you can create bus signals to improve the readability of your model. A bus signal or bus is a composite signal that consists of other signals that are called elements.

You can generate HDL code for designs that use virtual and nonvirtual buses. For example, you can generate code for designs that contain:

- DUT subsystem ports connected to buses.
- Simulink and Stateflow® blocks that support buses and HDL code generation.

Supported Blocks with Buses

Bus-capable blocks are blocks that can accept bus signals as input and produce bus signals as outputs. For a list of bus-capable blocks that Simulink supports, see "Bus-Capable Blocks". HDL Coder supports code generation for bus-capable blocks in the **HDL Coder** block library. For more details, see the "HDL Code Generation" section of each block page. The supported blocks include:

- Bus Assignment
- Bus Creator
- Bus Selector
- Constant
- Delay
- Multiport Switch
- Rate Transition
- Signal Conversion
- Signal Specification
- Switch
- Unit Delay
- Vector Concatenate, Matrix Concatenate
- Zero-Order Hold

In addition, subsystems, models, and these user-defined functions support buses for simulation and HDL code generation:

- Subsystem, Atomic Subsystem, CodeReuse Subsystem
- Model references, see “Model Referencing for HDL Code Generation” on page 25-2.
- Stateflow Chart
- MATLAB Function blocks
- MATLAB System blocks
- Vision HDL Toolbox blocks that accept a `pixelcontrol` bus for control input

Bus Support Limitations

Buses are not supported in the `IP Core Generation` workflow. In addition, you cannot generate code for designs that use:

- A Black box model reference connected to a bus.
- A bus input to a Delay block with nonzero **Initial condition**.

Enumerations

You can generate code for Simulink, MATLAB, or Stateflow enumerations within your design.

Requirements

- The enumeration strings must have unique names and must not use a reserved keyword in the Verilog, SystemVerilog or VHDL language.
- If your target language is Verilog or SystemVerilog, all enumeration member names must be unique within the design.

Restrictions

Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods:

- IP Core Generation workflow
- Simulink Real-Time FPGA I/O workflow
- Customization for the USRP Device workflow
- FPGA-in-the-loop
- HDL Cosimulation

Matrices

You can use matrix types with these blocks in your design. For more details, see the "HDL Code Generation" section of each block page.

HDL Coder Block Library	Supported blocks
Discontinuities	These blocks are supported: • Dead Zone • Relay • Saturation

<table>
<tr><th>HDL Coder Block Library</th><th>Supported blocks</th></tr>
<tr><td>Discrete</td><td>These blocks are supported:
• Delay
• Discrete-Time Integrator
• Memory
• Tapped Delay
• Unit Delay
• Unit Delay Enabled Synchronous
• Unit Delay Enabled Resettable Synchronous
• Unit Delay Resettable Synchronous
• Zero-Order Hold</td></tr>
<tr><td>HDL Floating Point Operations</td><td>The Rounding Function block is supported.</td></tr>
<tr><td>HDL Operations</td><td>All blocks in this library are supported.</td></tr>
<tr><td>HDL RAMs</td><td>Blocks in this library are not supported.</td></tr>
<tr><td>HDL Subsystems</td><td>Blocks in this library are not supported.</td></tr>
<tr><td>Logic and Bit Operations</td><td>These blocks are supported:
• Bit Clear
• Bit Concat
• Bit Reduce
• Bit Rotate
• Bit Set
• Bit Shift
• Bit Slice
• Extract Bits
• Logical Operator
• Relational Operator
• Shift Arithmetic</td></tr>
<tr><td>Lookup Tables</td><td>Blocks in this library are not supported.</td></tr>
</table>

HDL Coder Block Library	Supported blocks
Math Operations	These blocks are supported: • Abs • Add • Assignment • Bias • Complex to Real-Imag • Gain • Product, including matrix multiplication. • Matrix Concatenate • Math Function • Real-Imag to Complex • Reshape • Sign • Sum • Unary Minus • Increment Stored Integer • Increment Real World • Decrement Stored Integer • Decrement Real World • Sum of Elements • Product of Elements
Model Verification	All blocks in this library are supported.
Model-Wide Utilities	The DocBlock is supported. The Model Info block does not support matrix data types.
Ports & Subsystems	The Subsystem, Atomic Subsystem, CodeReuse Subsystem block is supported.
Signal Attributes	These blocks are supported: • Data Type Conversion • Data Type Duplicate • Probe • Rate Transition • Signal Conversion • Signal Specification

<table>
<tr><th>HDL Coder Block Library</th><th>Supported blocks</th></tr>
<tr><td>Signal Routing</td><td>These blocks are supported:
Mux
Multiport Switch
Selector
Switch</td></tr>
<tr><td>Sources</td><td>These blocks are supported:
Constant
Enumerated Constant
Inport</td></tr>
<tr><td>Sinks</td><td>These blocks are supported:
Display
Outport
Scope
To Workspace
To File</td></tr>
<tr><td>User-Defined Functions</td><td>The MATLAB Function block is supported.</td></tr>
</table>

Unsupported Signal and Data Types

- Arrays stored in row-major layout are not supported for HDL code generation
- Variable-size signals are not supported for code generation.

See Also

Related Examples

- “Generating HDL Code for Subsystems with Array of Buses” on page 14-22

More About

- “Signal Types”
- “About Data Types in Simulink”
- “Composite Signals”
- “Use Enumerated Data in Simulink Models”
- “Enumerated Data” (Stateflow)

Use Simulink Templates for HDL Code Generation

In this section...
“Create Model Using HDL Coder Model Template” on page 14-8
“HDL Coder Model Templates” on page 14-8

HDL Coder model templates in Simulink provide you with design patterns and best practices for models intended for HDL code generation. Models you create from one of the HDL Coder model templates have their configuration parameters and solver settings set up for HDL code generation. To configure an existing model for HDL code generation, use `hdlsetup`.

Create Model Using HDL Coder Model Template

To model hardware for efficient HDL code generation, create a model using an HDL Coder model template.

1. Open the Simulink Start Page. In the MATLAB Home tab, select the **Simulink** button. Alternatively, at the command line, enter:

   ```
   simulink
   ```
2. In the **HDL Coder** section, you see templates that are preconfigured for HDL code generation. Selecting the template opens a blank model in the Simulink Editor. To save the model, select **File > Save As**.
3. To open the Simulink Library Browser and then open the **HDL Coder** Block Library, select the **Library Browser** button in the Simulink Editor. Alternatively, at the command line, enter

   ```
   slLibraryBrowser
   ```

   To filter the Simulink Library Browser to show the block libraries that support HDL code generation, use the `hdllib` function:

   ```
   hdllib
   ```

HDL Coder Model Templates

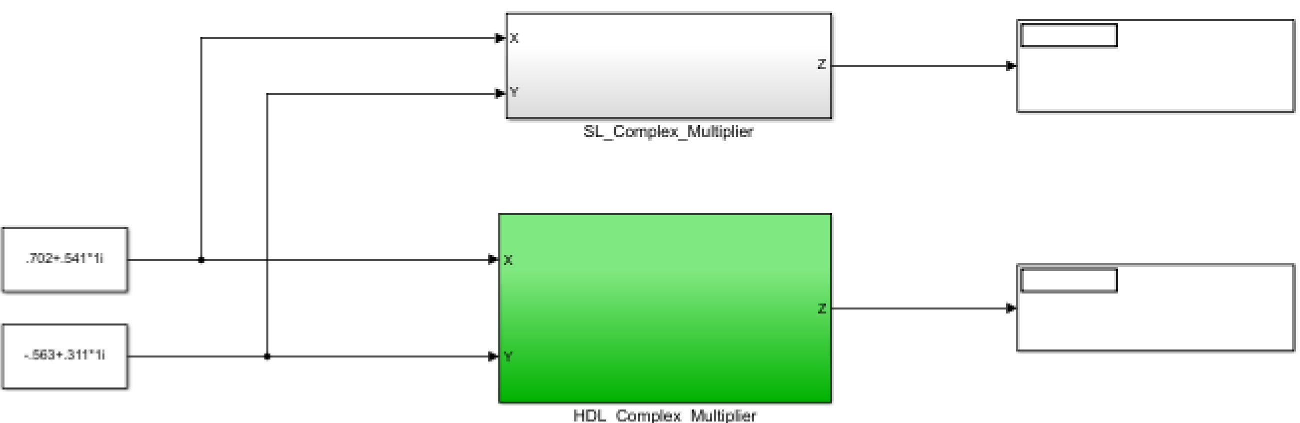
Complex Multiplier

The Complex Multiplier template shows how to model a complex multiplier-accumulator and manually pipeline the intermediate stages. The hardware implementation of complex multiplication uses four multipliers and two adders.

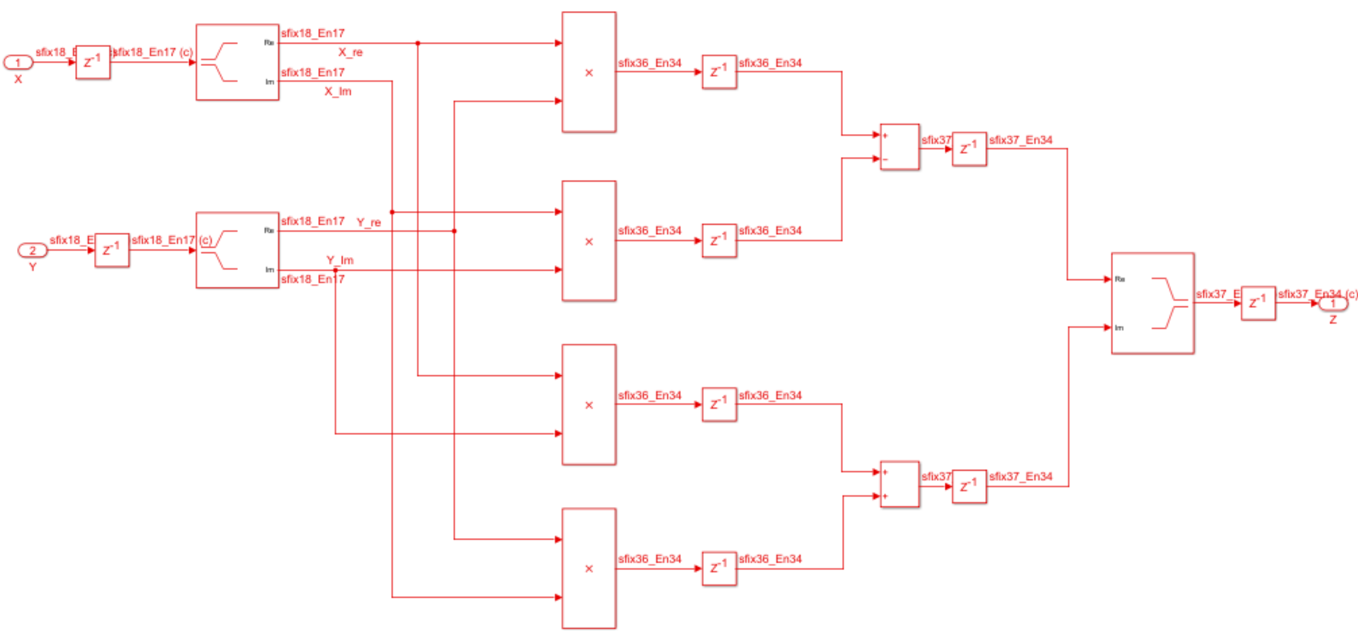
The template applies the following best practices:

- In the Configuration Parameters dialog box, in **HDL Code Generation > Global Settings**, **Reset type** is set to `Synchronous`.
- To improve speed, Delay blocks, which map to registers in hardware, are at the inputs and outputs of the multipliers and adders.
- To support the output data of a full-precision complex multiplier, the output data word length is manually specified to be (*operand_word_length* * 2) + 1.

  For example, in the template, the operand word length is 18, and the output word length is 37.

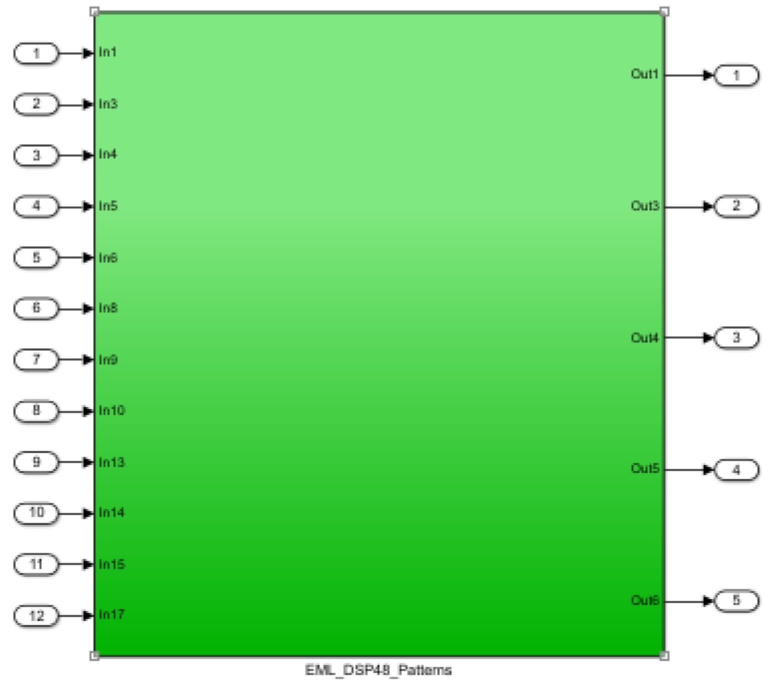


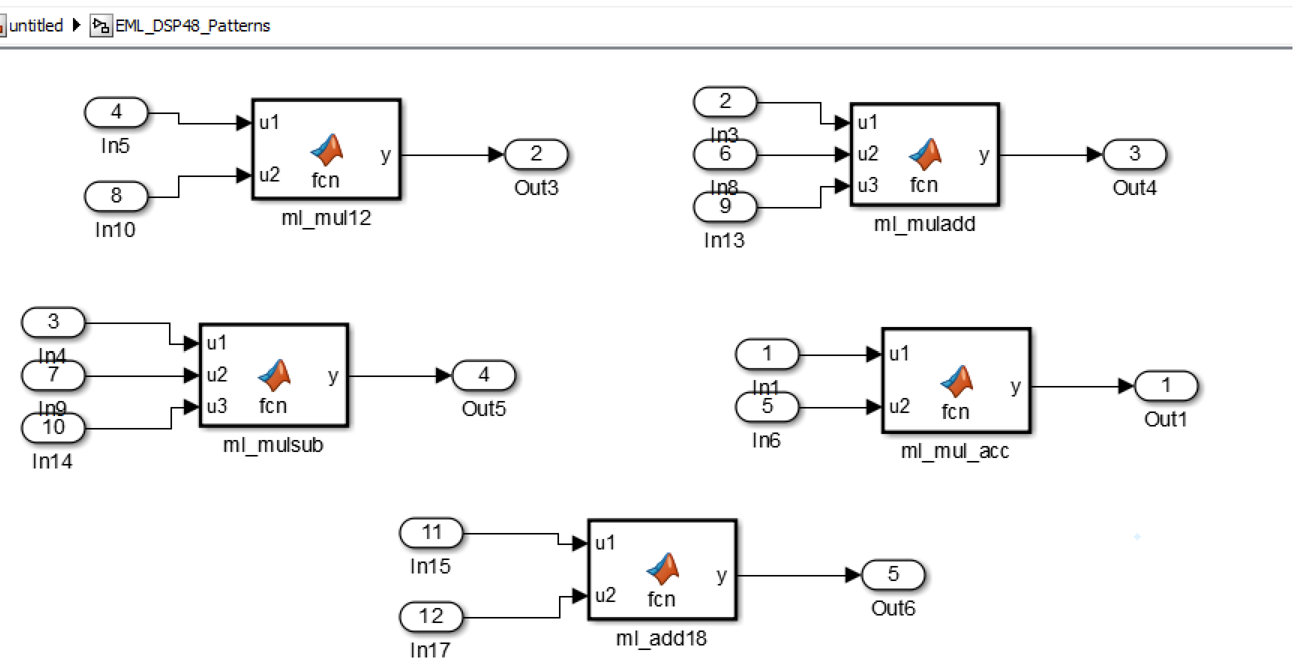
Copyright 2014-2016 The MathWorks, Inc.



MATLAB Arithmetic

The MATLAB Arithmetic template contains MATLAB arithmetic operations that infer DSP48s in hardware.





For example, the `ml_mul_acc` MATLAB Function block shows how to write a multiply-accumulate operation in MATLAB. hdlfimath on page 27-19 applies fixed-point math settings for HDL code generation.

```
function y = fcn(u1, u2)

% design of a 6x6 multipler
% same reset on inputs and outputs
% followed by an adder

nt = numerictype(0,6,0);
nt2 = numerictype(0,12,0);
fm = hdlfimath;

persistent u1_reg u2_reg mul_reg add_reg;
if isempty(u1_reg)
    u1_reg = fi(0, nt, fm);
    u2_reg = fi(0, nt, fm);
    mul_reg = fi(0, nt2, fm);
    add_reg = fi(0, nt2, fm);
end

mul = mul_reg;
mul_reg = u1_reg * u2_reg;
add = add_reg;
add_reg(:) = mul+add;
u1_reg = u1;
u2_reg = u2;

y = add;
```

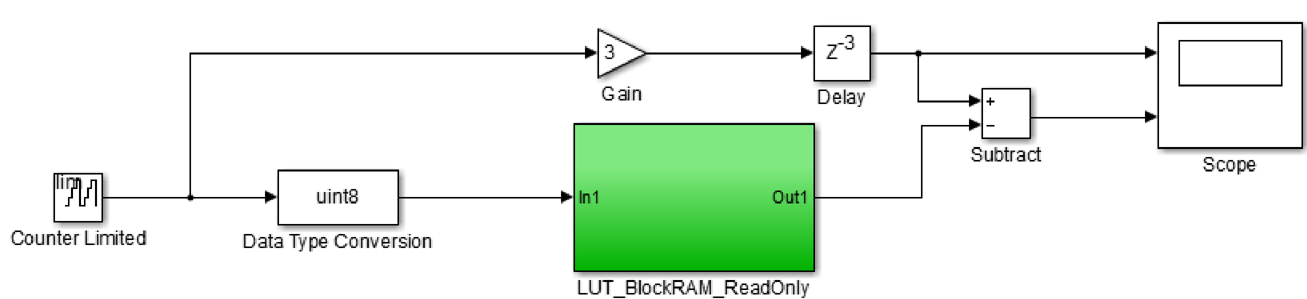
ROM

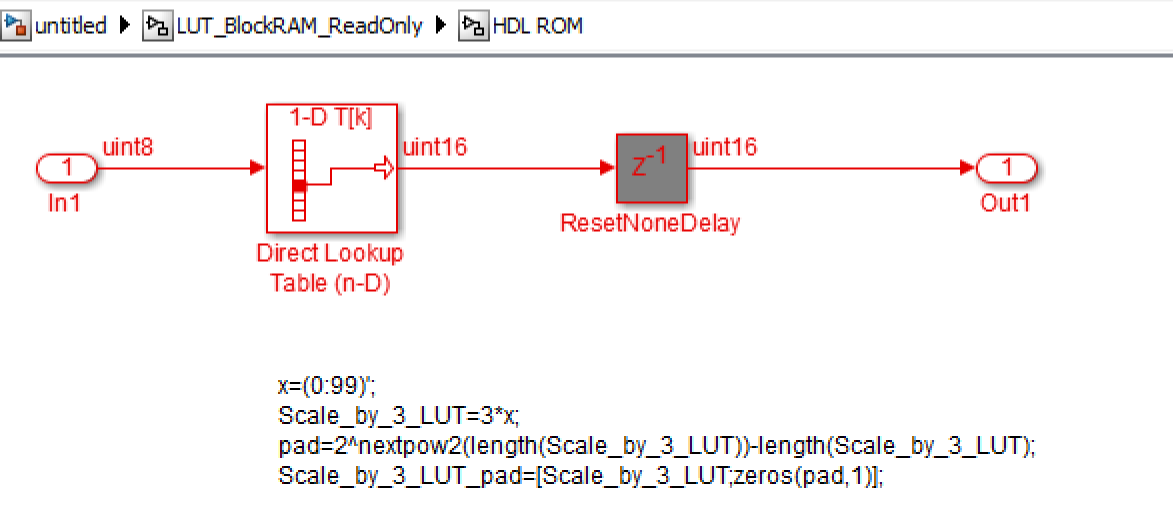
The ROM template is a design pattern that maps to a ROM in hardware.

The template applies the following best practices:

- At the output of the lookup table, there is a Delay block with `ResetType` = `none`.
- The lookup table is structured such that the spacing between breakpoints is a power of two.

  Using table dimensions that are a power of two enables HDL Coder to generate shift operations instead of division operations. If necessary, pad the table with zeros.
- The number of lookup table entries is a power of two. For some synthesis tools, a lookup table that has a power-of-two number of entries maps better to ROM. If necessary, pad the table with zeros.



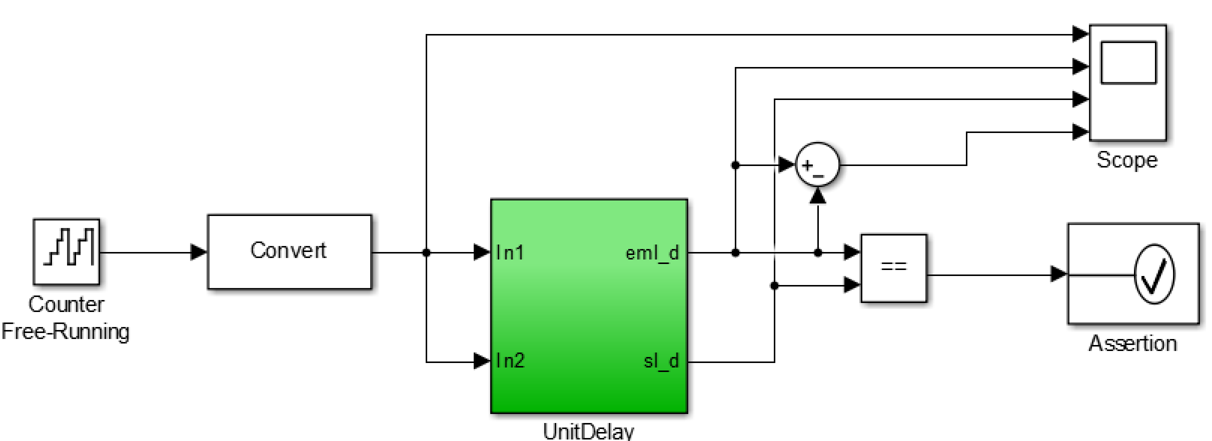


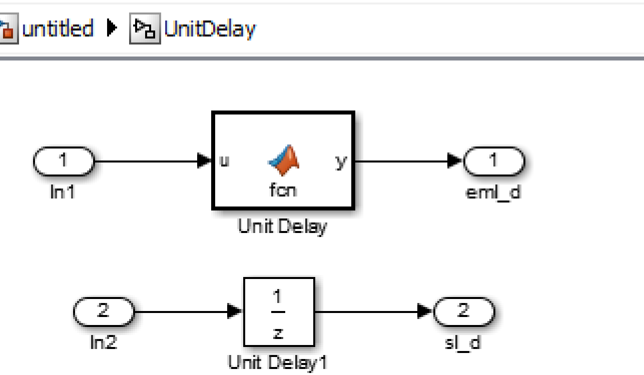
Register

The Register template shows how to model hardware registers:

- In Simulink, using the Delay block.
- In MATLAB, using persistent variables.

  This design pattern also shows how to use `cast` to propagate data types automatically.



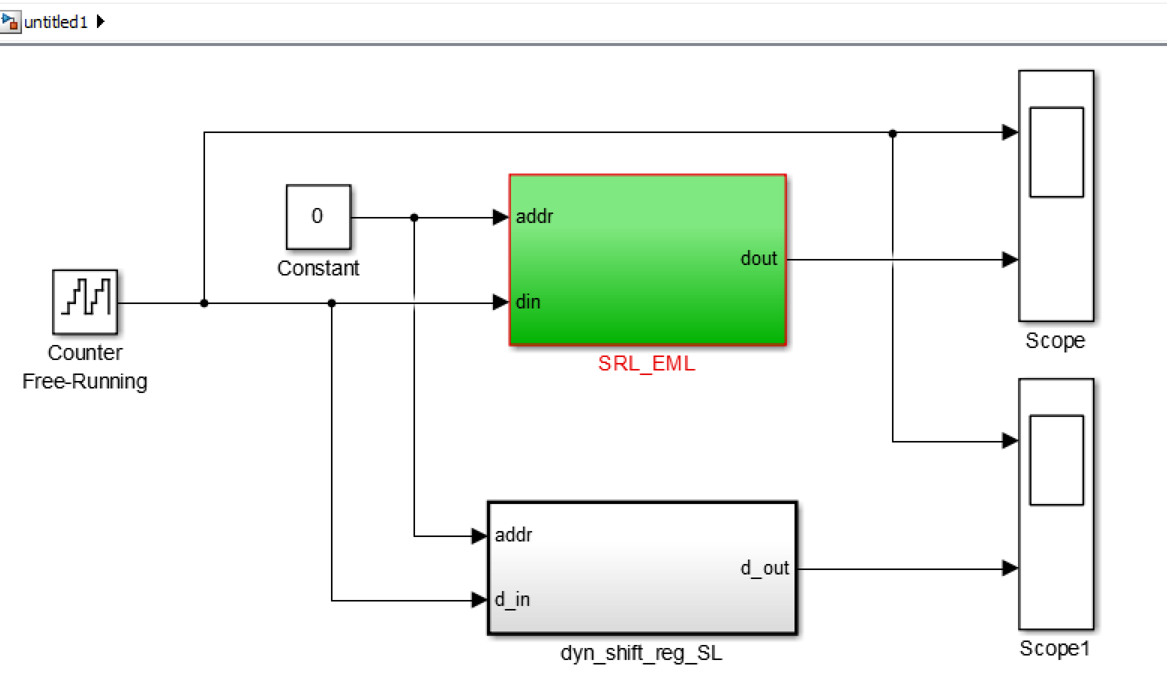


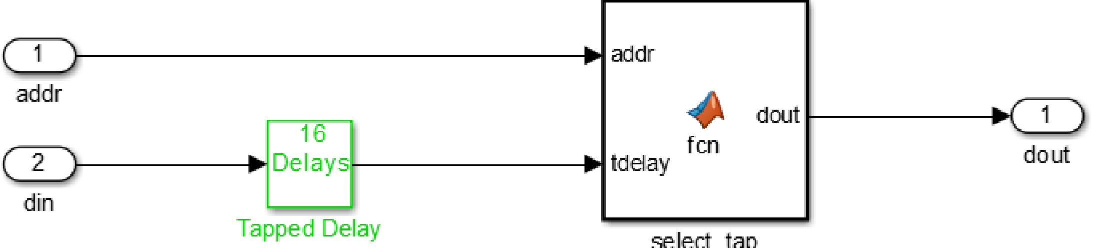
The MATLAB code in the MATLAB Function block uses a persistent variable to model the register.

```
function y = fcn(u)
% Unit delay implementation that maps to a register in hardware

persistent u_d;
if isempty(u_d)
    % defines initial value driven by unit delay at time step 0
    u_d = cast(0, 'like', u);
end

% return delayed input from last sample time hit
y = u_d;

% store the current input
u_d = u;
```

SRL

The SRL template shows how to implement a shift register that maps to an SRL16 in hardware. You can use a similar pattern to map to an SRL32.





To map to SRL16/32:
- Set ResetType = none for the tapped delay
- Use ML fcn block to create mux logic
- Flatten hierarchy for the subsystem to inline the ML code
- Do not use "include current input in output vector" option for the tapped delay

In the shift register subsystem, the Tapped Delay implements the shift operation, and the MATLAB Function, `select_tap`, implements the output mux.

In `select_tap`, the zero-based address, `addr` increments by 1 because MATLAB indices are one-based.

```
function dout = fcn(addr, tdelay)
%#codegen

addr1 = fi(addr+1,0,5,0);
dout = tdelay(addr1);
```

In the generated code, HDL Coder automatically omits the increment because Verilog, SystemVerilog and VHDL are zero-based.

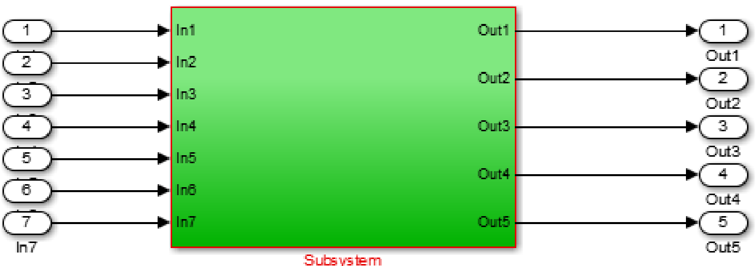
The template also applies the following best practices for mapping to an SRL16 in hardware:

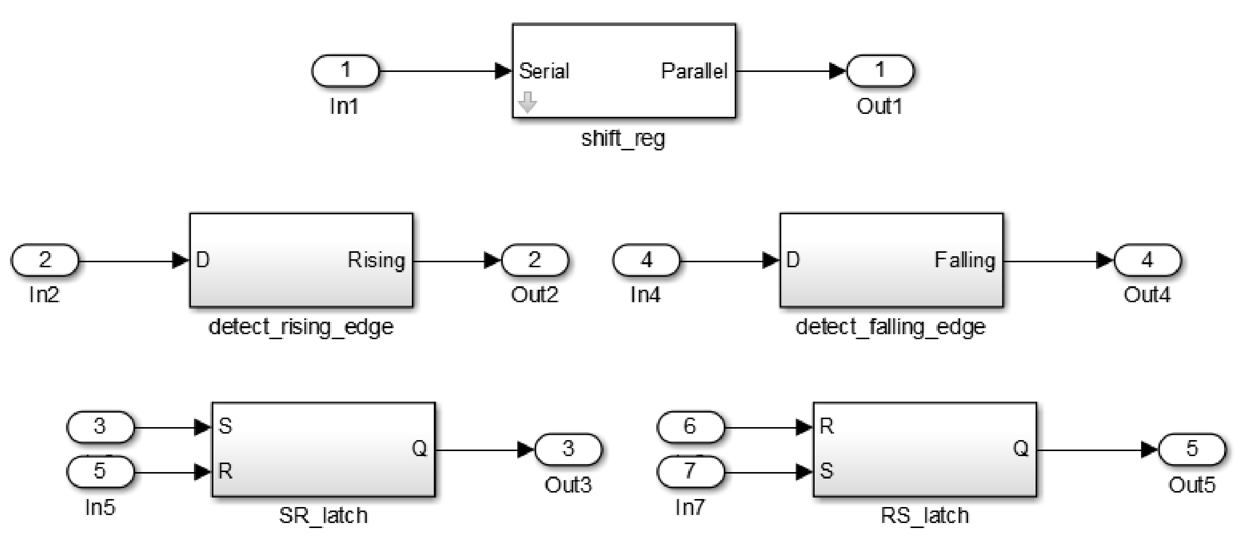
- For the Tapped Delay block:
  - In the Block Parameters dialog box, **Include current input in output vector** is not enabled.
  - In the HDL Block Properties dialog box, **ResetType** is set to none.
- For the Subsystem block, in the HDL Block Properties dialog box, **FlattenHierarchy** is set to on.

Simulink Hardware Patterns

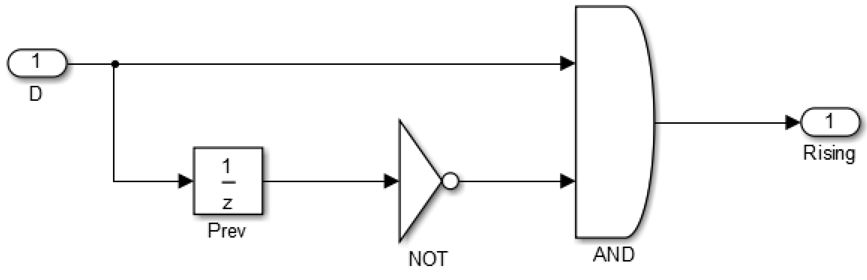
The Simulink Hardware Patterns template contains design patterns for common hardware operations:

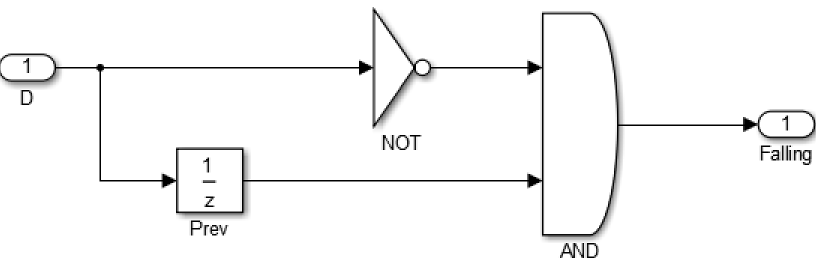
- Serial-to-parallel shift register
- Detect rising edge
- Detect falling edge
- SR latch
- RS latch





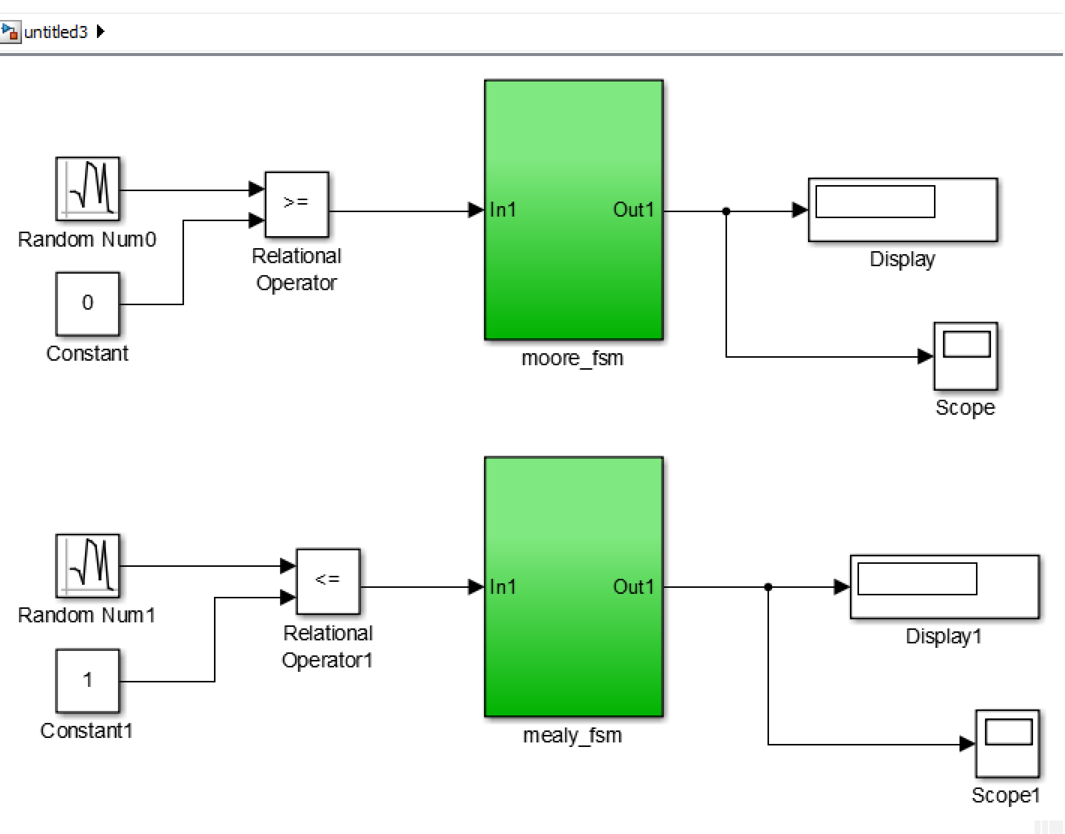
For example, the design patterns for rising edge detection and falling edge detection:





State Machine in MATLAB

The State Machine in MATLAB template shows how to implement Mealy and Moore state machines using the MATLAB Function block.



To learn more about best practices for modeling state machines, see “Model a State Machine for HDL and High-Level Synthesis Code Generation” on page 3-4.

See Also

`hdlsetup` | `makehdl`

More About

- “Create HDL-Compatible Simulink Model”
- “Generate HDL Code from Simulink Model”
- “Design Guidelines for the MATLAB Function Block” on page 27-19
- “Hardware Modeling with MATLAB Code”

Generate DUT Ports for Tunable Parameters

In this section...

"Prerequisites" on page 14-18
"Create and Add Tunable Parameter That Maps to DUT Ports" on page 14-18
"Generated Code" on page 14-19
"Limitations" on page 14-20
"Use Tunable Parameter in Other Blocks" on page 14-20

Tunable parameters that you use to adjust your model behavior during simulation can map to top-level DUT ports in your generated HDL code. HDL Coder generates one DUT port per tunable parameter.

You can generate a DUT port for a tunable parameter by using it in one of these blocks:

- Gain
- Constant
- MATLAB Function
- MATLAB System
- Chart
- Truth Table
- State Transition Table

These blocks with the tunable parameter can be at any level of the DUT hierarchy, including within a model reference.

You cannot use HDL cosimulation with a DUT that uses tunable parameters in any of these blocks. If you use a tunable parameter in a block other than these blocks, code is generated inline and does not map to DUT ports. To use the value of a tunable parameter in a Chart or Truth Table block, see "Use Tunable Parameter in Other Blocks" on page 14-20.

You can define and store the tunable parameters in the base workspace or a Simulink data dictionary. However, a Simulink data dictionary provides more capabilities. For details, see "What Is a Data Dictionary?".

Prerequisites

- The Simulink compiled data type for all instances of a tunable parameter must be the same.
- Simulink blocks that use tunable parameters with the same name must operate at the same data rate.

To learn more about Simulink compiled data types, see "Control Block Parameter Data Types".

Create and Add Tunable Parameter That Maps to DUT Ports

To generate a DUT port for a tunable parameter:

1. Create a tunable parameter with `StorageClass` set to `ExportedGlobal`.

For example, to create a tunable parameter, *myParam*, and initialize it to 5, at the command line, enter:

```
myParam = Simulink.Parameter;
myParam.Value = 5;
myParam.CoderInfo.StorageClass = 'ExportedGlobal';
```

Create a tunable parameter in Base Workspace using the Model Explorer:

- Add a Simulink.Parameter by selecting `Simulink.Parameter` from the **Add** menu to the Base Workspace.
- In the **StorageClass** column of the **Contents** pane, set **StorageClass** to `ExportedGlobal`.
- Alternatively, in the **Simulink.Parameter** pane, click **Code Generation** tab, set **StorageClass** to `ExportedGlobal`.

Alternatively, Create a tunable parameter in Model Workspace using the Model Explorer:

- Add a Simulink.Parameter by selecting `Simulink.Parameter` from the **Add** menu to the Model Workspace.
- In the **StorageClass** column of the **Contents** pane, click the **Configure** link.
- Alternatively in the **Simulink.Parameter** pane, click **Code Generation** tab, click **Configure in Coder App**.
- In the **Code Mappings** pane, click the **Parameters** tab. Under **Model Parameters**, set **StorageClass** to `ExportedGlobal`.

See “Create Data Objects from Built-In Data Class Package Simulink”.

2. In your Simulink design, use the tunable parameter as the:

   - **Constant value** in a Constant block.
   - **Gain** parameter in a Gain block.
   - MATLAB function argument in a MATLAB Function block.

Generated Code

The following VHDL code is an example of code that HDL Coder generates for a Gain block with its **Gain** field set to a tunable parameter, *myParam*. You see that the code generator creates a DUT port and adds a comment to indicate that the port corresponds to a tunable parameter.

```
ENTITY s IS
  PORT( In1      : IN   std_logic_vector(15 DOWNTO 0); -- sfix16_En5
        myParam  : IN   std_logic_vector(15 DOWNTO 0); -- sfix16_En5 Tunable port
        Out1     : OUT  std_logic_vector(31 DOWNTO 0) -- sfix32_En10
        );
END s;


ARCHITECTURE rtl OF s IS

  -- Signals
  SIGNAL myParam_signed  : signed(15 DOWNTO 0); -- sfix16_En5
  SIGNAL In1_signed      : signed(15 DOWNTO 0); -- sfix16_En5
  SIGNAL Gain_out1       : signed(31 DOWNTO 0); -- sfix32_En10

BEGIN
  myParam_signed <= signed(myParam);

  In1_signed <= signed(In1);
```

```
  Gain_out1 <= myParam_signed * In1_signed;

  Out1 <= std_logic_vector(Gain_out1);

END rtl;
```

Limitations

- Make sure that the Use trigger signal as clock check box is left cleared by default. When you use test point or tunable parameters in the IP core workflow, the DUT must be a subsystem level DUT and not the top-level model or model reference DUT.
- The tunable parameter uses the scalar value type, you cannot use complex, enumerated, and mathematical expressions value types.
- You cannot set value of a variable to an expression involving tunable parameter.

Use Tunable Parameter in Other Blocks

To use the value of a tunable parameter in a Chart or Truth Table block:

1. Create the tunable parameter and use it in a Constant block. on page 14-18
2. Add an input port to the block where you want to use the tunable parameter.
3. Connect the output of the Constant block to the new input port.

See Also

Related Examples

- "Generate Parameterized Code for Referenced Models" on page 14-21

Generate Parameterized Code for Referenced Models

In this section...

"Parameterize Referenced Model for HDL Code Generation" on page 14-21
"Restrictions" on page 14-21

To generate parameterized code for referenced models, use model arguments. You can use model arguments in a masked or unmasked Model block.

HDL Coder generates a single VHDL `entity`, Verilog or SystemVerilog `module` for the referenced model, even if the DUT has multiple instances of the referenced model. In the generated code, each model argument is a VHDL `generic` or a Verilog or SystemVerilog `parameter`.

Parameterize Referenced Model for HDL Code Generation

1. In the referenced model, create one or more model arguments.

   To learn how to create a model argument, see "Specify a Different Value for Each Instance of a Reusable Model".
2. In the referenced model, use each model argument parameter in a Gain or Constant block.
3. In the DUT, for each model reference, in the **Model arguments** table, enter values for each model argument.

   Alternatively, create a model mask for the referenced model. In the DUT, for each model reference, enter values for each model argument.
4. Generate code for the DUT.

Restrictions

Model argument values:

- Must be scalar.
- Cannot be complex.
- Cannot be enumerated data.

See Also

Related Examples

- "Generate Reusable Code for Subsystems" on page 25-18
- "Generate DUT Ports for Tunable Parameters" on page 14-18

More About

- "Specify a Different Value for Each Instance of a Reusable Model"
- "Model Referencing for HDL Code Generation" on page 25-2

Generating HDL Code for Subsystems with Array of Buses

In this section...

"How HDL Coder Generates Code for Array of Buses" on page 14-22

"Array of Buses Limitations" on page 14-24

An array of buses is an array whose elements are buses. Each element in an array of buses must be nonvirtual and must have the same data type.

The array of buses represents structured data compactly. The array:

- Reduces the model complexity
- Reduces maintenance by organizing and routing signals in your Simulink model for vectorized algorithms

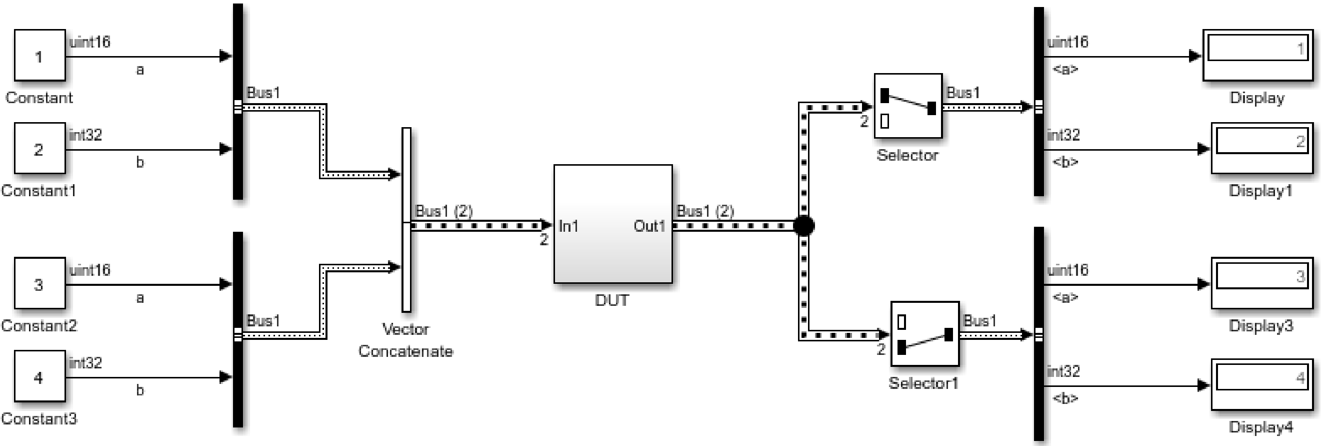
For more information, see "Group Nonvirtual Buses in Arrays of Buses".

You can generate HDL code for virtual and nonvirtual blocks that Simulink supports with an array of buses. For more information, see "Bus-Capable Blocks".

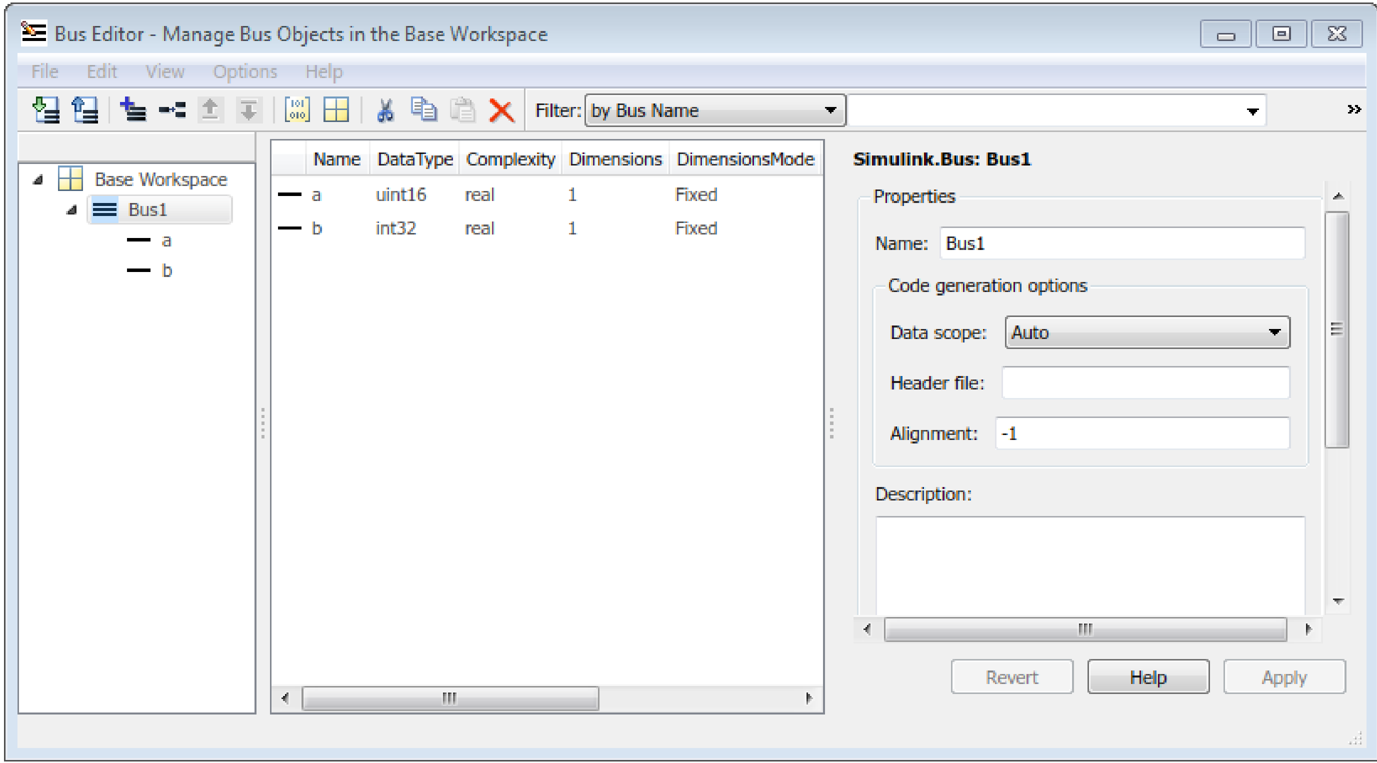
How HDL Coder Generates Code for Array of Buses

HDL Coder expands the array of buses in your Simulink model into the corresponding scalar signals in the generated code.

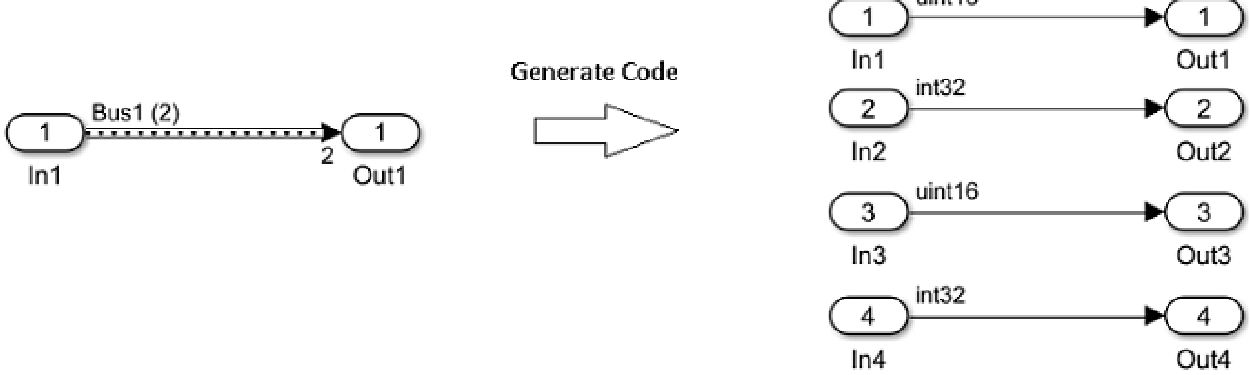
This Simulink model has an array of buses signal at the DUT interface.



The array of buses combines two nonvirtual bus elements, each having scalars `a` and `b` of types `uint16` and `int32` respectively.



The resulting HDL code expands the array of buses into scalars, and contains four scalar input and output ports.



In the generated code, the array of bus expansion results in four scalar signals at the input and output ports. For the first bus object, the input ports are `In_1_a` and `In_1_b`. For the second bus object, they are `In_2_a` and `In_2_b`. At the output, for the first bus object, they are `Out_1_a` and `Out_1_b`. For the second bus object, they are `Out_2_a` and `Out_2_b`.

```
ENTITY DUT IS
  PORT( In1_1_a    :   IN    std_logic_vector(15 DOWNTO 0);  -- uint16
        In1_1_b    :   IN    std_logic_vector(31 DOWNTO 0);  -- int32
        In1_2_a    :   IN    std_logic_vector(15 DOWNTO 0);  -- uint16
        In1_2_b    :   IN    std_logic_vector(31 DOWNTO 0);  -- int32
```

```
        Out1_1_a  :   OUT   std_logic_vector(15 DOWNTO 0);  -- uint16
        Out1_1_b  :   OUT   std_logic_vector(31 DOWNTO 0);  -- int32
        Out1_2_a  :   OUT   std_logic_vector(15 DOWNTO 0);  -- uint16
        Out1_2_b  :   OUT   std_logic_vector(31 DOWNTO 0)   -- int32
        );
END DUT;
```

HDL Coder generates code in accordance with the order in which you specify the bus elements and the array elements in your Simulink model. If you specify the VHDL target language for your Simulink model that contains a bus object with arrays, HDL Coder preserves the arrays in the generated code, and does not expand into scalars.

Array of Buses Limitations

- Do not use the array of buses inside other data types. You cannot use a bus signal that contains an array of buses.
- MATLAB System and MATLAB Function blocks that contain System objects are not supported with an array of buses.

See Also

More About

- “Signal and Data Type Support” on page 14-3
- “Signal Types”
- “About Data Types in Simulink”
- “Composite Signals”
- “Use Enumerated Data in Simulink Models”
- “Enumerated Data” (Stateflow)
- “Generate HDL Code with Record or Structure Types for Bus Signals” on page 14-25

Generate HDL Code with Record or Structure Types for Bus Signals

In this section...

"Requirements and Considerations" on page 14-25
"Use Record or Structure Types for Bus in HDL Code Generation" on page 14-26
"Generate Record Types for Bus Signals at Subsystem Interface" on page 14-26
"Generate Record Types for Array of Bus" on page 14-28
"Limitations" on page 14-30

For a Simulink model that contains the bus signals at the subsystem interface, you can generate code with record or structure types for those bus signals. Use the record or structure types to simplify your HDL code. The record or structure types is especially useful for simplifying interfaces and maintaining a large number of signals at the entity level.

To generate code with record or structure types for bus signals, enable the **Preserve Bus structure in the generated HDL code** parameter. This parameter is available in the Configuration Parameter dialog box, in the **HDL Code Generation > Global Settings > Coding Style > RTL Style** section.

HDL Coder generates record or structure types for the bus signals, an array of bus, the bus signals that have different data types, and the nested bus signals, for these interfaces:

- Bus signals at design-under-test (DUT) interface
- Bus signals at different subsystem-level interface
- Bus signals at the model reference interface
- Bus signals at black box interface
- Bus signals at controlled subsystem interface

Requirements and Considerations

- To generate code with record or structure types, set the target language to VHDL or SystemVerilog. Specify the **Target language** option in the **HDL Code Generation** pane of the Configuration Parameter dialog box.
- When you create a bus in your model, the `Bus` object must be specified. You can create the `Bus` object by using the Type Editor tool. To open Type Editor, run this command in the MATLAB Command Window:

  ```
  typeeditor
  ```

  After you create a `Bus` object and specify its attributes, you can associate it with any block that needs to use the bus definition that the object provides. To associate a block with a bus, in the Block Parameters dialog box, set **Output data type** or **Data type** to `Bus: <object name>`, replacing `<object name>` with the `Bus` object name and select the **Output as nonvirtual bus**. For more information, see "Specify Bus Properties with Bus Objects".
- When you use **Preserve Bus structure in the generated HDL code** option to generate record or structure for bus signal, you must turn off **Scalarize ports** option for your model. If **Scalarize ports** option is enabled, then HDL Coder does not generate record or structure types for bus signal due to the scalarization of bus ports. For more information, see Scalarize ports.

- To generate record or structure types for bus signal at model reference interface, enable **Generate VHDL or SystemVerilog code for model references into a single library** option. For more information, see Generate VHDL or SystemVerilog code for model references into a single library.

Use Record or Structure Types for Bus in HDL Code Generation

By default, the **Preserve Bus structure in the generated HDL code** option is disabled. The generated code does not have record or structure types. The bus signals are flattened at the interface and are defined as individual ports in an entity in the generated code.

If your model contains the bus signals at the DUT interface or different subsystem-level interface, you can generate the HDL code with record or structure types for those bus signals.

To generate record or structure types for the bus signals in the HDL code of your DUT subsystem:

1. Create a `Bus` object by using the Type Editor tool.
2. Specify **Output data type** as the Bus object in the Bus Creator block. For more information, see “Create Simulink Bus Objects”.
3. Specify **Target language** as `VHDL` or `SystemVerilog`.
4. Enable the generation of record tor structure types for bus. In the configuration parameter dialog box, go to the **HDL Code Generation > Global Settings > Coding Style > RTL Style** section, select **Preserve Bus structure in the generated HDL code**.
5. Generate HDL code for the DUT subsystem. In the **HDL Code** tab, set **Code for** subsystem to the DUT subsystem, and then click the **Generate HDL Code** button.

You can also generate HDL code for your DUT with record types by setting the `GenerateRecordType` argument of the `makehdl` function to "on".

```
makehdl(<DUT>,GenerateRecordType="on");
```

Generate Record Types for Bus Signals at Subsystem Interface

This example shows how to generate VHDL code with record types for a model that has bus signals at the design under test (DUT) interface. The **Preserve Bus structure in the generated HDL code** configuration option enables you to generate code with record types for the bus signals. Use the record type to simplify your VHDL code. The record type is especially useful for simplifying interfaces and maintaining a large number of signals at the entity level.

Open Model

The Simulink® model `BusSignalWithRecordTypes` has a bus signal at the DUT interface. The `DUT` subsystem consists of a Unit Delay block. The input and output of the `DUT` subsystem is a bus. This bus has two bus elements with the `double` datatype. The Bus Creator block uses bus object `BusRecordObj` for creating bus signals. To load the bus object `BusRecordObj`, run:

```
run('BusRecordInput.m')
```

Load and open the `BusSignalWithRecordTypes` model by running these commands:

```
load_system("BusSignalWithRecordTypes");
set_param("BusSignalWithRecordTypes",'SimulationCommand','Update')
open_system("BusSignalWithRecordTypes");
```

The bus object "BusRecordObj" is loaded
from PreLoadFcn model callback.

Copyright 2023 The MathWorks Inc.

Generate HDL code

You can generate the HDL code for a model by using HDL Coder™. To generate HDL code for `DUT` subsystem with record types by setting the `GenerateRecordType` argument of the `makehdl` to "on":

```
makehdl("BusSignalWithRecordTypes/DUT", GenerateRecordType="on")
```

Generated VHDL Code for DUT Subsystem

HDL Coder defines the record type for bus signals in the VHDL package file. HDL Coder uses this package file for the port declaration of the bus signals in an entity. This code snippet shows the VHDL code of a package file that specifies record types for two bus signal. The package file defines the record `BusRecordObj_record`, which consists of two bus elements with the double data type.

```
-- -------------------------------------------------
LIBRARY IEEE;
IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

PACKAGE DUT_pkg IS
    TYPE BusRecordObj_record IS RECORD
        signal1          : std_logic_vector(63 DOWNTO 0);
        signal2          : std_logic_vector(63 DOWNTO 0);
    END RECORD BusRecordObj_record;

END DUT_pkg;
```

These VHDL code of the `DUT` subsystem uses these record types for port and signals declaration of the bus signals. This code snippet shows VHDL code for a `DUT` subsystem which uses record types for the bus signals:

```
-- ------------------------------------------------------------
LIBRARY IEEE;
IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE work.Subsystem_pkg.ALL;

ENTITY DUT IS
  PORT( clk              :   IN    std_logic;
```

```
        reset           :   IN    std_logic;
        clk_enable      :   IN    std_logic;
        In1             :   IN    BusRecordObj_record;  -- record {double,double}
        ce_out          :   OUT   std_logic;
        Out1            :   OUT   BusRecordObj_record  -- record {double,double}
        );
END DUT;

ARCHITECTURE rtl OF Subsystem IS

    -- Signals
    SIGNAL enb              : std_logic;
    SIGNAL In1_signal1      : std_logic_vector(63 DOWNTO 0);  -- ufix64
    SIGNAL In1_signal2      : std_logic_vector(63 DOWNTO 0);  -- ufix64
    SIGNAL signal1          : std_logic_vector(63 DOWNTO 0);  -- ufix64
    SIGNAL signal2          : std_logic_vector(63 DOWNTO 0);  -- ufix64
    SIGNAL Unit_Delay_out1 : BusRecordObj_record;  -- record {double,double}
```

Generate Record Types for Array of Bus

This example shows model that has array of bus signal at the design under test (DUT) interface. Using **Preserve Bus structure in the generated HDL code** configuration option, you can generate code with array of record types for the array of bus signals.

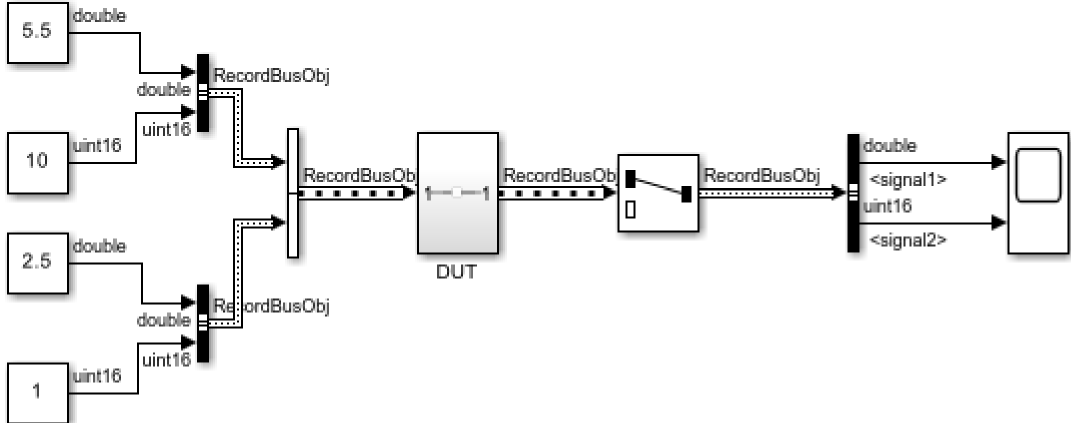
Open Model

The model `ArrayOfBusSignalWithRecordTypes` has array of bus signal at the `DUT` interface. The Bus Creator block uses bus object `RecordBusObj` for creating bus signals. This bus object has two bus elements of datatypes, `double` and `unit16`. To load the bus object `RecordBusObj`, run:

```
run('RecordArrayOfBusInput.m')
```

Load and open the `BusSignalWithRecordTypes` model by running these commands:

```
load_system("ArrayOfBusSignalWithRecordTypes");
set_param("ArrayOfBusSignalWithRecordTypes",'SimulationCommand','Update')
open_system("ArrayOfBusSignalWithRecordTypes");
```



The Vector Concatenate block generates an array of bus signal by concatenating two bus signals from the Bus Creator blocks.

Generated VHDL Code with Record Types for Array of Bus

Enable **Preserve Bus structure in the generated HDL code** option and generate HDL code for `DUT` subsystem by using this command.

```
makehdl("ArrayOfBusSignalWithRecordTypes/DUT",GenerateRecordType="on")
```

HDL Coder generates an array of record for array of bus signals in the VHDL package file. This code snippet shows the VHDL code of a package file that specifies `RecordBusObj` record types and vector of record for the array of bus signal.

```
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

PACKAGE DUT_pkg IS
TYPE RecordBusObjlogicType_record IS RECORD
    signal1                              : std_logic_vector(63 DOWNTO 0);
    signal2                              : std_logic_vector(15 DOWNTO 0);
END RECORD RecordBusObjlogicType_record;

TYPE vector_of_RecordBusObjlogicType_record IS ARRAY (NATURAL RANGE <>) OF BusRecordBusObjlo
TYPE RecordBusObj_record IS RECORD
    signal1                              : std_logic_vector(63 DOWNTO 0);
    signal2                              : unsigned(15 DOWNTO 0);
END RECORD RecordBusObj_record;

TYPE vector_of_RecordBusObj_record IS ARRAY (NATURAL RANGE <>) OF RecordBusObj_record;
END DUT_pkg;
```

These VHDL code of the `DUT` subsystem uses these vector record for port declaration of the bus array. This code snippet shows VHDL code for a `DUT` subsystem which uses vector record for the bus array:

```
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE work.DUT_pkg.ALL;

ENTITY DUT IS
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        clk_enable                        :   IN    std_logic;
        In1                               :   IN    vector_of_RecordBusObjlogicType_record(0
        ce_out                            :   OUT   std_logic;
        Out1                              :   OUT   vector_of_RecordBusObjlogicType_record(0
        );
END DUT;
```

Limitations

- HDL test bench generation does not support the record or structure type when you disable the **Use file I/O to read/write test bench data** option.
- HDL test bench generation does not support the record or structure type when you enable the **SystemVerilog DPI test bench** option.
- You cannot generate a cosimulation model for record or structure types.
- Target IP core generation does not support ports with the record or structure type.
- When your model contains the bus elements of complex or enumerated types, HDL Coder generates code without record or structure types for these buses.

See Also

Functions
`makehdl`

Objects
`Simulink.Bus`

Apps
Type Editor

More About

- Preserve Bus structure in the generated HDL code
- “Generating HDL Code for Subsystems with Array of Buses” on page 14-22
- “Specify Bus Properties with Bus Objects”
- “Bus-Capable Blocks”
- “Create Nonvirtual Buses”

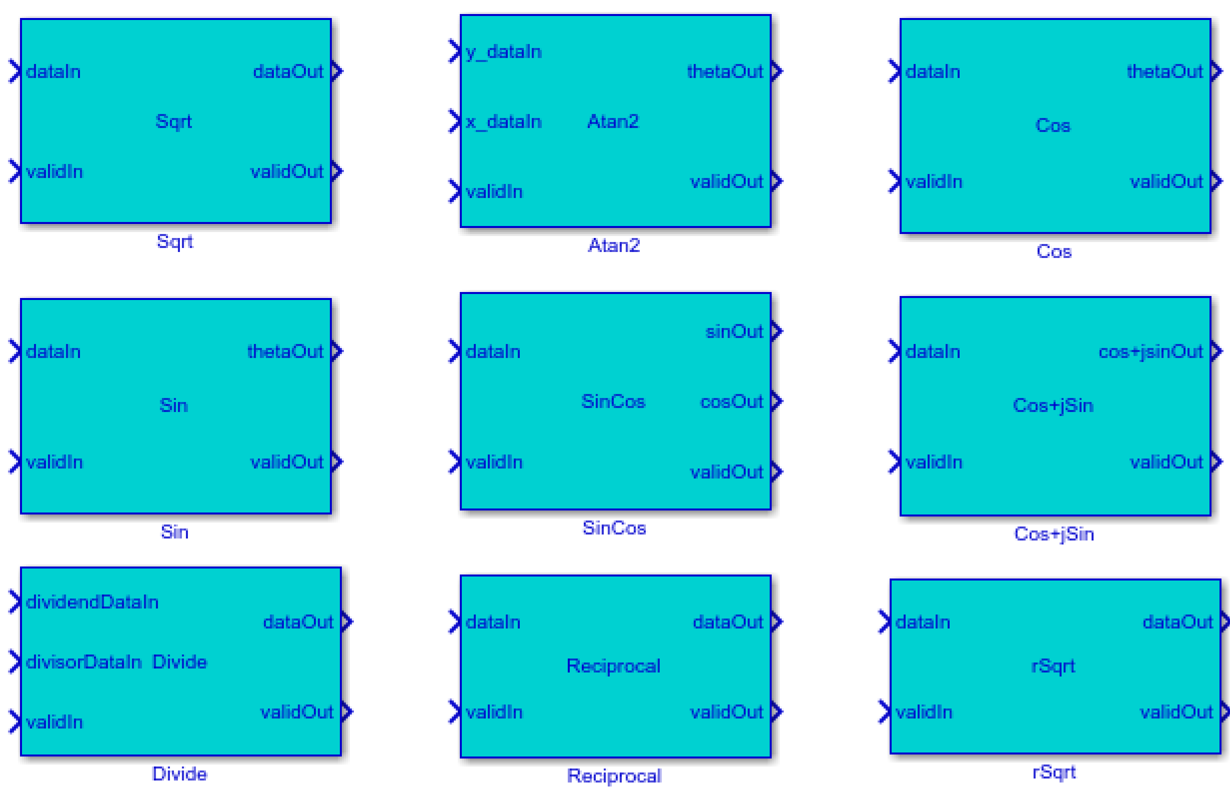
Implement Control Signal-Based Mathematical Functions by Using HDL Coder

HDL Coder™ provides `HDLMathLib` blocks that have control signal-based fixed-point mathematical functions. The blocks have control signals that indicate whether the input and output data are valid. You can also simulate these blocks with latency.

`HDLMathLib` Library With Control Ports for Mathematical Functions

To see all the mathematical function blocks in the HDLMathLib library, open the library by using this command.

```
open_system('HDLMathLib')
```



HDLMathLib include these blocks:

- Sqrt
- rSqrt
- Atan2
- Sin
- Cos

- SinCos
- Cos+jSin
- Reciprocal
- Divide

These blocks are a masked subsystem that contains a MATLAB Function block, `LumpLatency`. This MATLAB Function block is used to compute the latency. To see the latency computation for a block, view inside the masked subsystem and open `LumpLatency` block.

Related Examples

- “Implement Atan2 Block with Control Signals” on page 14-53
- “Implement Divide Block with Control Signals” on page 14-45
- “Implement Reciprocal Block with Control Signals” on page 14-37
- “Implement Sine and Cosine Block with Control Signals” on page 14-49
- “Implement Sqrt Block with Control Signals” on page 14-33
- “Implement rSqrt Block with Control Signals” on page 14-41

Implement Sqrt Block with Control Signals

This example shows how to implement the control-signal based Square root block and use it to generate HDL code.

Open and Run the Simulink Model

Specify the input data as a linear sweep. You can change these values according to your requirements.

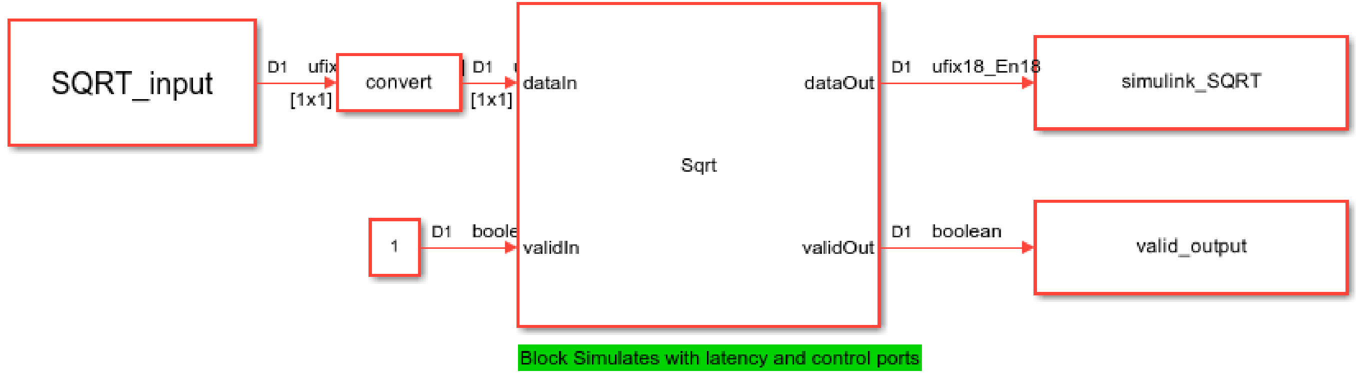
```
SQRT_input = fi(1/2^17:1/2^17:1,0,18,17)';
```

Specify the word length for fixed-point datatypes and the latency for the model. For more information on the latency calculation, see Sqrt.

```
WL = 18;  latency = 20;
```

Open the `hdlcoder_sqrt_bitset_control` model and specify a stop time sufficient to process all the input combinations.

```
stoptime = length(SQRT_input)-1+latency;
open_system('hdlcoder_sqrt_bitset_control')
sim('hdlcoder_sqrt_bitset_control')
```



This figure shows the simulation waveform for the model. You can see that `dataOut` output is valid when `validOut` is `1`.

Validate Simulink Output by Using Reference Output

To validate the output of the Simulink model, compare this output to a reference values. To obtain the reference values, use the `sqrt` function. Compute the reference output by using the `sqrt` function.

```
ref_SQRT = sqrt(double(SQRT_input));
```

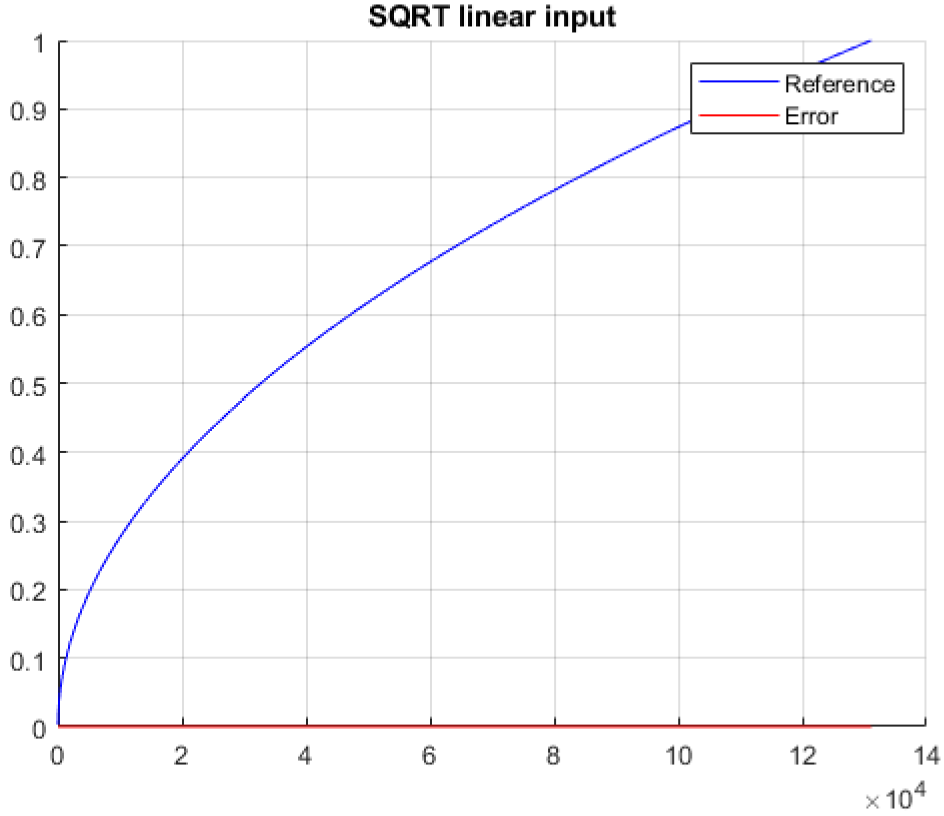
Use logical indexing to extract valid output.

```
implementation_SQRT = simulink_SQRT(valid_output);
```

Plot the comparison results by using the `comparison_plot_sqrt` function. The maximum error value is significantly smaller than the output of the model.

```
comparison_plot_sqrt(ref_SQRT,implementation_SQRT,1,'SQRT linear input');
```

```
Maximum Error SQRT linear input 3.814697e-06
Maximum PctError SQRT linear input 3.803159e-02
```



Generate HDL Code for Square Root Implementation

Check the HDL settings saved of the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_sqrt_bitset_control')
```

```
%% Set Model 'hdlcoder_sqrt_bitset_control' HDL parameters
hdlset_param('hdlcoder_sqrt_bitset_control', 'Backannotation', 'on');
hdlset_param('hdlcoder_sqrt_bitset_control', 'HDLSubsystem', 'hdlcoder_sqrt_bitset_control/Sqrt'
hdlset_param('hdlcoder_sqrt_bitset_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_sqrt_bitset_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
```

```
hdlset_param('hdlcoder_sqrt_bitset_control', 'TargetFrequency', 500);
hdlset_param('hdlcoder_sqrt_bitset_control', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt/LumpLatency', 'Architecture', 'MATLAB Datapath')
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt/LumpLatency', 'FlattenHierarchy', 'on');

% Set Sqrt HDL parameters
hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt/Sqrt', 'LatencyStrategy', 'Max');

hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt/ValidLine', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt/ValidLine', 'FlattenHierarchy', 'on');
```

To generate HDL code for the Sqrt block in the model, use the `makehdl` function.

```
makehdl('hdlcoder_sqrt_bitset_control/Sqrt')
close_system('hdlcoder_sqrt_bitset_control')
close all;
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_sqrt_bitset_control')">hdlcoder_sq
### Generating HDL for <a href="matlab:open_system('hdlcoder_sqrt_bitset_control/Sqrt')">hdlcoder
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_sqrt_bi
### Running HDL checks on the model 'hdlcoder_sqrt_bitset_control'.
### Begin compilation of the model 'hdlcoder_sqrt_bitset_control'...
### Begin compilation of the model 'hdlcoder_sqrt_bitset_control'...
### Working on the model 'hdlcoder_sqrt_bitset_control'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_sqrt_bitset_control', 'Genera
### Begin model generation 'gm_hdlcoder_sqrt_bitset_control'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_sqrt_bitset_con
### Begin VHDL Code Generation for 'hdlcoder_sqrt_bitset_control'.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_sqrt_bitset_control', 'Tracea
### Working on hdlcoder_sqrt_bitset_control/Sqrt/Sqrt as hdl_prj\hdlsrc\hdlcoder_sqrt_bitset_cont
### Working on hdlcoder_sqrt_bitset_control/Sqrt as hdl_prj\hdlsrc\hdlcoder_sqrt_bitset_control\S
### Generating package file hdl_prj\hdlsrc\hdlcoder_sqrt_bitset_control\Sqrt_pkg.vhd.
### Code Generation for 'hdlcoder_sqrt_bitset_control' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_sqrt_bitset_control' complete with 0 errors, 0 warnings, and 0 messag
### HDL code generation complete.
```

Sqrt Block Synthesis Performance

This figure shows the Sqrt block synthesis performance on the Xilinx® Virtex® 7 and Intel® Stratix® V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
234	499	1167	954

Intel Quartus Stratix V (5SEE9F45C2)

Fmax (MHz)	LABs	ALMs	Registers
185	110	823	848

Implement Reciprocal Block with Control Signals

This example shows how to implement the control-signal based Reciprocal block and use it to generate HDL code.

Open and Run Simulink Model

Specify the input data values as a linear sweep. You can change these values according to your requirements

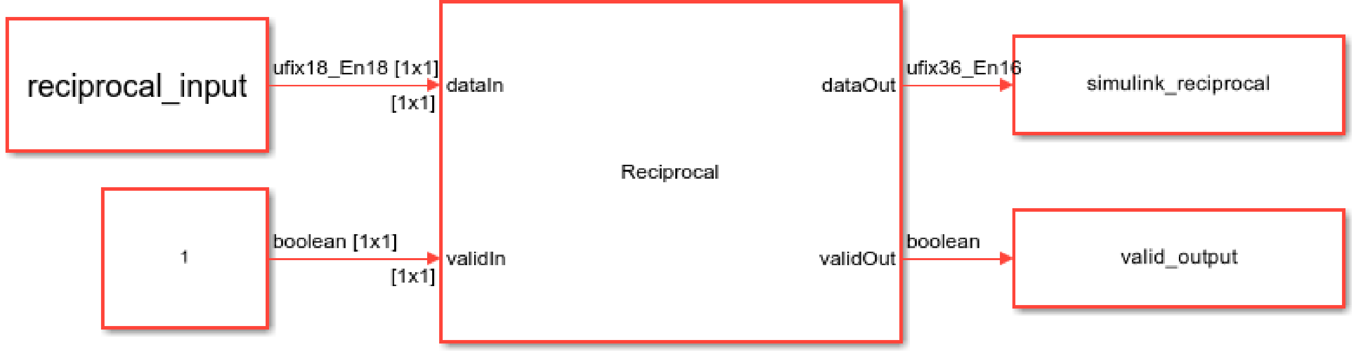
```
reciprocal_input = fi(1/2^10:1/2^18:1,0,18,18)';
```

Specify the word length for fixed-point datatypes and the latency for the model. For more information on the latency calculation, see Reciprocal.

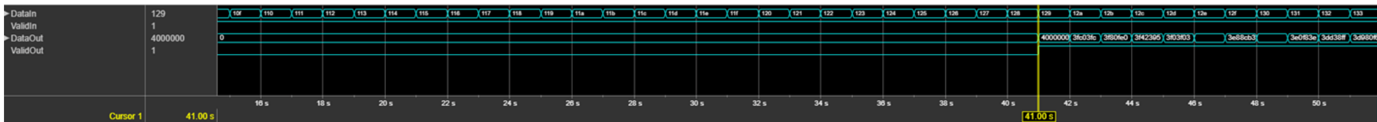
```
WL_recip = 18;  recip_latency = 41;
```

Open the `hdlcoder_reciprocal_shiftadd_control` model and specify a stop time sufficient to process all the input combinations.

```
stoptime_recip = length(reciprocal_input)-1+recip_latency;
open_system('hdlcoder_reciprocal_shiftadd_control')
sim('hdlcoder_reciprocal_shiftadd_control')
```



This figure shows the output waveform when you simulate the model. The `dataOut` output is valid when `validOut` is `1`.



Validate Simulink Output By Using Reference Output

To validate the output of the Simulink model, compare this output with a reference values. Compute the reference output by using the `reciprocal` operation.

```
ref_reciprocal = 1./double(reciprocal_input);
```

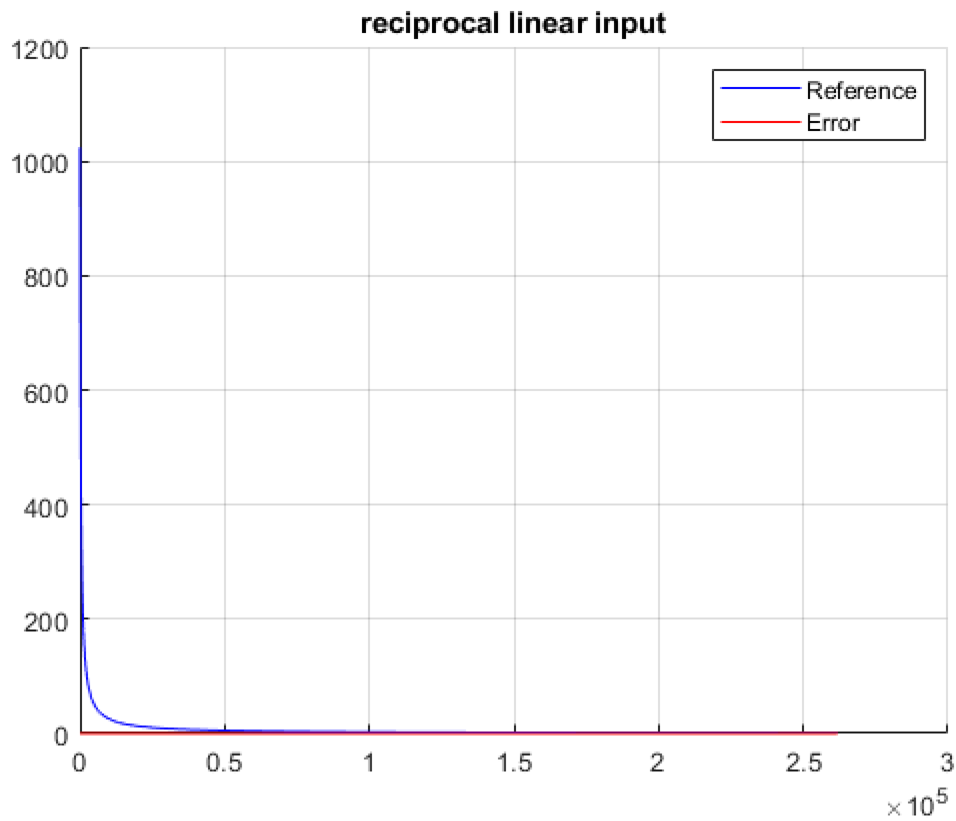
Use logical indexing to extract valid output.

```
implementation_reciprocal = simulink_reciprocal(valid_output);
```

Plot the comparison results by using the `comparison_plot_reciprocal` function. The maximum error value is significantly smaller than the output of the model.

```
comparison_plot_reciprocal(ref_reciprocal,implementation_reciprocal,9,'reciprocal linear input')
```

```
Maximum Error reciprocal linear input 1.525867e-05
Maximum PctError reciprocal linear input 1.522159e-03
```



Generate HDL Code for Reciprocal Implementation

Check HDL settings of the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_reciprocal_shiftadd_control')
```

```
%% Set Model 'hdlcoder_reciprocal_shiftadd_control' HDL parameters
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'Backannotation', 'on');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'HDLSubsystem', 'hdlcoder_reciprocal_shifta
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
```

```
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'TargetFrequency', 500);
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/LumpLatency', 'Architecture', 'MATI
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/LumpLatency', 'FlattenHierarchy',

hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/Reciprocal', 'Architecture', 'Shif
% Set Product HDL parameters
hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/Reciprocal', 'LatencyStrategy', 'Ma

hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/ValidLine', 'Architecture', 'MATLAI
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/ValidLine', 'FlattenHierarchy', 'or
```

To generate HDL code for the Reciprocal block in the model, use the `makehdl` function.

```
makehdl('hdlcoder_reciprocal_shiftadd_control/Reciprocal')
close_system('hdlcoder_reciprocal_shiftadd_control')
close all;
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_reciprocal_shiftadd_control')">hd
### Generating HDL for <a href="matlab:open_system('hdlcoder_reciprocal_shiftadd_control/Recipro
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_recipr
### Running HDL checks on the model 'hdlcoder_reciprocal_shiftadd_control'.
### Begin compilation of the model 'hdlcoder_reciprocal_shiftadd_control'...
### Begin compilation of the model 'hdlcoder_reciprocal_shiftadd_control'...
### Working on the model 'hdlcoder_reciprocal_shiftadd_control'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_reciprocal_shiftadd_control'
### Begin model generation 'gm_hdlcoder_reciprocal_shiftadd_control'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_reciprocal_shi
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdl_
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsr
### Begin VHDL Code Generation for 'hdlcoder_reciprocal_shiftadd_control'.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_reciprocal_shiftadd_control'
### Working on hdlcoder_reciprocal_shiftadd_control/Reciprocal/Reciprocal as hdl_prj\hdlsrc\hdlc
### Working on hdlcoder_reciprocal_shiftadd_control/Reciprocal as hdl_prj\hdlsrc\hdlcoder_recipr
### Code Generation for 'hdlcoder_reciprocal_shiftadd_control' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/t
### HDL check for 'hdlcoder_reciprocal_shiftadd_control' complete with 0 errors, 0 warnings, and
### HDL code generation complete.
```

Reciprocal Block Synthesis Performance

This figure shows the Reciprocal block synthesis performance on the Xilinx® Virtex® 7 and Intel® Stratix® V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
432	487	1364	2055

Intel Quartus Stratix V (5SEE9F45C2)

Fmax (MHz)	LABs	ALMs	Registers
265	181	1050	2819

Implement rSqrt Block with Control Signals

This example shows how to implement the control-signal based reciprocal square root block and use it to generate HDL code.

Open and Run Simulink Model

Specify the input data as a linear sweep. You can change these values according to your requirements.

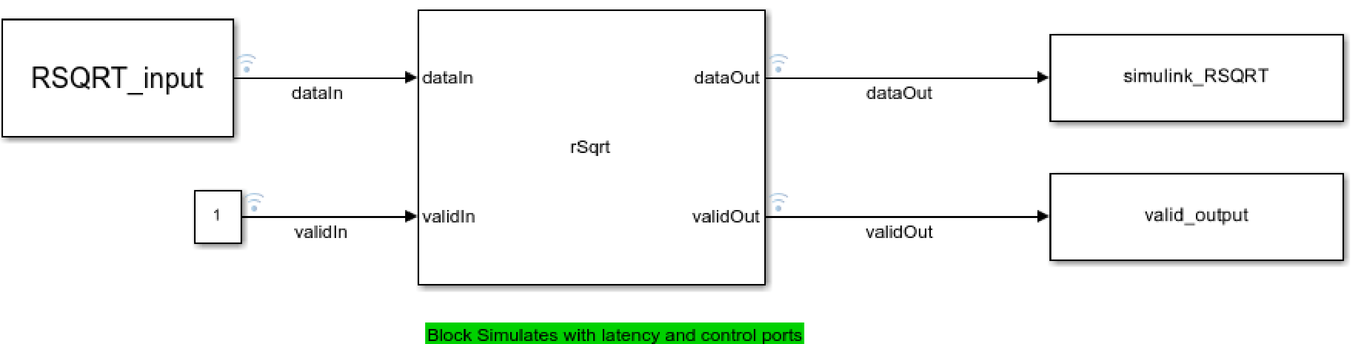
```
RSQRT_input = fi(1/10:1/10:100,0,16,8)';
```

Specify the word length for fixed-point datatypes and the latency for the model.

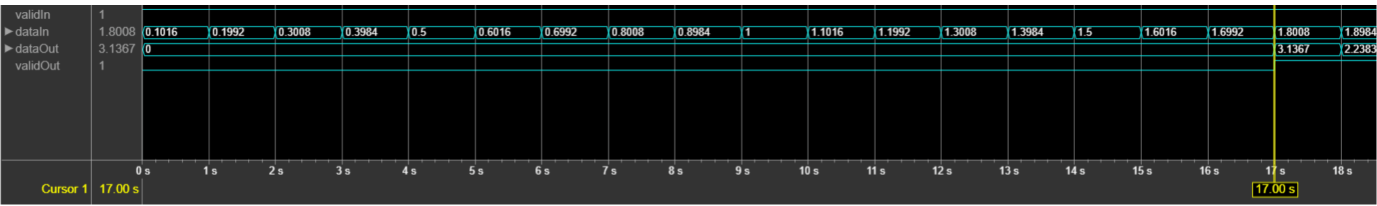
```
WL = 16;  latency = 17;
```

Open the `hdlcoder_rsqrt_bitset_control` model and specify a stop time sufficient to process all the input combinations.

```
stoptime = length(RSQRT_input)-1+latency;
open_system('hdlcoder_rsqrt_bitset_control')
sim('hdlcoder_rsqrt_bitset_control')
```



This figure shows the simulation waveform for the model. You can see that `dataOut` output is valid when `validOut` is `1`.



Validate Simulink Output by Using Reference Output

To validate the output of the Simulink model, compare this output to a reference value. Compute the reference output by using the 1/|sqrt| operation.

```
ref_RSQRT = sqrt(1./double(RSQRT_input));
```

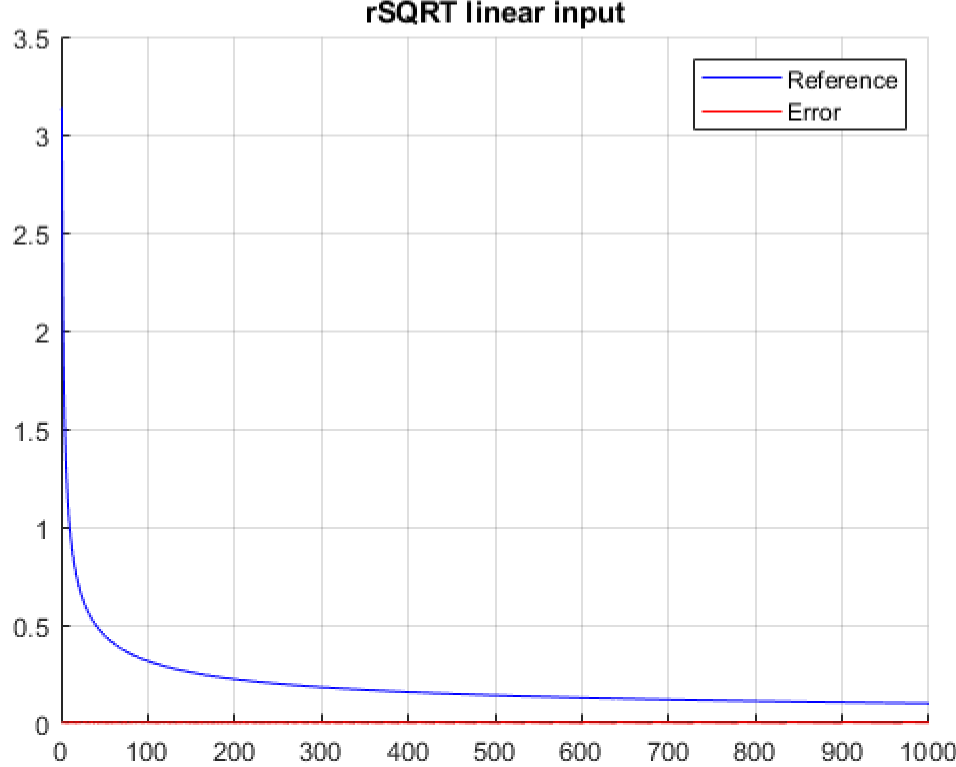
Use logical indexing to extract valid output.

```
implementation_RSQRT = simulink_RSQRT(valid_output);
```

Plot the comparison results by using the `comparison_plot_rsqrt` function. The maximum error value is significantly smaller than the output of the model.

```
comparison_plot_rsqrt(ref_RSQRT,implementation_RSQRT,1,'rSQRT linear input');
```

```
Maximum Error rSQRT linear input 3.906250e-03
Maximum PctError rSQRT linear input 3.819748e+00
```



Generate HDL Code for Reciprocal Square Root Implementation

Check the HDL settings of the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_rsqrt_bitset_control')
```

```
%% Set Model 'hdlcoder_rsqrt_bitset_control' HDL parameters
hdlset_param('hdlcoder_rsqrt_bitset_control', 'Backannotation', 'on');
hdlset_param('hdlcoder_rsqrt_bitset_control', 'HDLSubsystem', 'hdlcoder_rsqrt_bitset_control');
hdlset_param('hdlcoder_rsqrt_bitset_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_rsqrt_bitset_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_rsqrt_bitset_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_rsqrt_bitset_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_rsqrt_bitset_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_rsqrt_bitset_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_rsqrt_bitset_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
```

```
hdlset_param('hdlcoder_rsqrt_bitset_control', 'TargetFrequency', 500);
hdlset_param('hdlcoder_rsqrt_bitset_control', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_rsqrt_bitset_control/rSqrt', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_rsqrt_bitset_control/rSqrt/LumpLatency', 'Architecture', 'MATLAB Datapath
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_rsqrt_bitset_control/rSqrt/LumpLatency', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_rsqrt_bitset_control/rSqrt/ValidLine', 'Architecture', 'MATLAB Datapath')
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_rsqrt_bitset_control/rSqrt/ValidLine', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_rsqrt_bitset_control/rSqrt/rSqrt', 'Architecture', 'RecipSqrtNewtonSingleR
```

To generate HDL code for the rSqrt block in the model, use the `makehdl` function.

```
makehdl('hdlcoder_rsqrt_bitset_control/rSqrt')
close_system('hdlcoder_rsqrt_bitset_control')
close all;
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_rsqrt_bitset_control')">hdlcoder_r
### Generating HDL for <a href="matlab:open_system('hdlcoder_rsqrt_bitset_control/rSqrt')">hdlcod
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_rsqrt_b
### Running HDL checks on the model 'hdlcoder_rsqrt_bitset_control'.
### Begin compilation of the model 'hdlcoder_rsqrt_bitset_control'...
### Begin compilation of the model 'hdlcoder_rsqrt_bitset_control'...
### Working on the model 'hdlcoder_rsqrt_bitset_control'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_rsqrt_bitset_control', 'Gener
### Begin model generation 'gm_hdlcoder_rsqrt_bitset_control'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_rsqrt_bitset_co
### Begin VHDL Code Generation for 'hdlcoder_rsqrt_bitset_control'.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_rsqrt_bitset_control', 'Trace
### Working on hdlcoder_rsqrt_bitset_control/rSqrt/rSqrt/rSqrt_iv/NewtonPolynomialIVStage as hdl_
### Working on hdlcoder_rsqrt_bitset_control/rSqrt/rSqrt/rSqrt_iv as hdl_prj\hdlsrc\hdlcoder_rsqr
### Working on hdlcoder_rsqrt_bitset_control/rSqrt/rSqrt/rSqrt_core as hdl_prj\hdlsrc\hdlcoder_rs
### Working on hdlcoder_rsqrt_bitset_control/rSqrt/rSqrt as hdl_prj\hdlsrc\hdlcoder_rsqrt_bitset_
### Working on hdlcoder_rsqrt_bitset_control/rSqrt as hdl_prj\hdlsrc\hdlcoder_rsqrt_bitset_contro
### Generating package file hdl_prj\hdlsrc\hdlcoder_rsqrt_bitset_control\rSqrt_pkg.vhd.
### Code Generation for 'hdlcoder_rsqrt_bitset_control' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_rsqrt_bitset_control' complete with 0 errors, 0 warnings, and 1 messa
### HDL code generation complete.
```

rSqrt Block Synthesis Performance

This figure shows the rSqrt block synthesis performance on the Xilinx® Zynq®-7000 and Intel® Arria10® V devices.

Xilinx Vivado Zynq xc7z035-fbg676 (Speed Grade: -1)

Fmax (MHz)	Slice LUTs	Slice Registers	DSPs
136.30	673	842	40

Xilinx Vivado Zynq xc7z035-fbg676 (Speed Grade: -1)

Fmax (MHz)	Slice LUTs	Slice Registers	DSPs
136.30	673	842	40

Implement Divide Block with Control Signals

This example shows how to implement the control-signal based Divide block and use it to generate HDL code.

Open and Run Simulink Model

Specify the input data as a linear sweep. You can change these values according to your requirements.

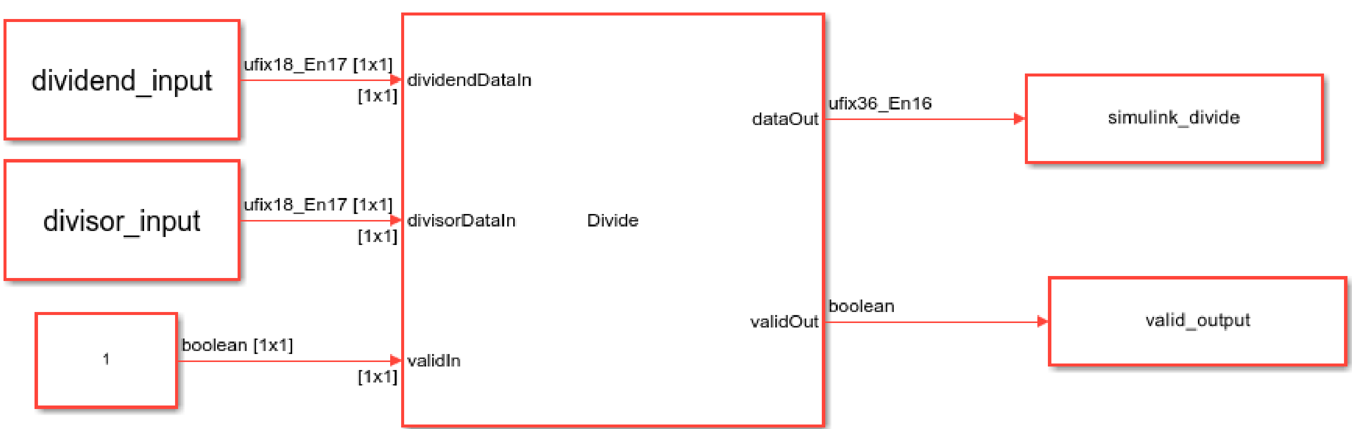
```
divisor_input = fi(1/2^17:1/2^17:1,0,18,17)';
dividend_input = fi((1/2^17 + 1/2^16):1/2^17:1,0,18,17)';
divisor_input_size = size(divisor_input);
dividend_input_size = size(dividend_input);
if(divisor_input_size(1) > dividend_input_size(1))
    data_size = dividend_input_size(1);
else
    data_size = divisor_input_size(1);
end
dividend_input = dividend_input(1:data_size,1);
divisor_input =  divisor_input(1:data_size,1);
```

Specify the word length for fixed-point datatypes and the latency for the model. For more information on the latency calculation, see Divide.

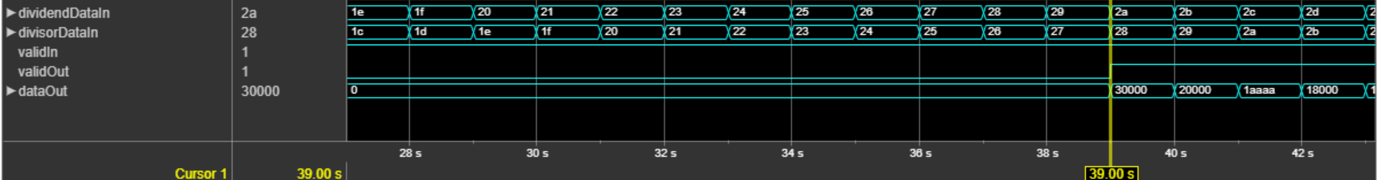
```
WL_divide = 18;  divide_latency = 39;
```

Open the `hdlcoder_divide_shiftadd_control` model and specify a stop time sufficient to process all the input combinations.

```
stoptime_divide = length(dividend_input)-1+divide_latency;
open_system('hdlcoder_divide_shiftadd_control')
sim('hdlcoder_divide_shiftadd_control')
```



The figure shows waveform when you simulate the above model. The `dataOut` output is valid when `validOut` is 1.



Validate Simulink Output By Using Reference Output

To validate the output of the Simulink model, compare this output with a reference value. Compute the reference output by using the `divide` function.

```
ref_divide = double(dividend_input)./double(divisor_input);
```

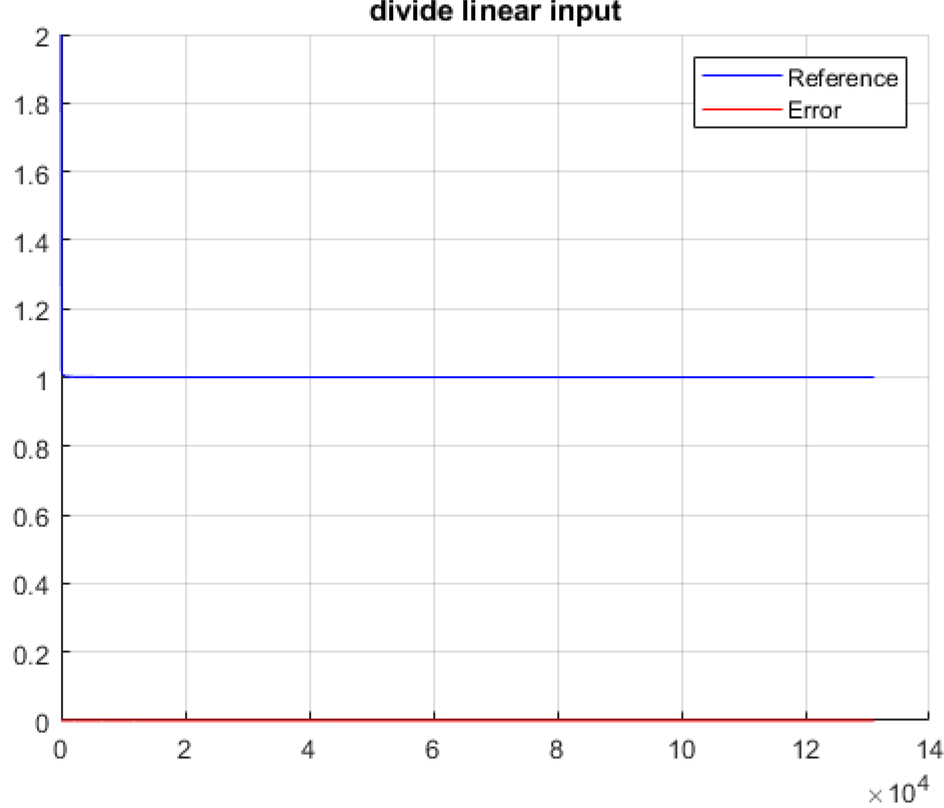
Use logical indexing to extract valid output.

```
implementation_divide = simulink_divide(valid_output);
```

Plot the comparison results by using the `comparison_plot_divide` function. The maximum error value is significantly smaller than the output of the model.

```
comparison_plot_divide(ref_divide,implementation_divide,11,'divide linear input');
```

```
Maximum Error divide linear input 1.525844e-05
Maximum PctError divide linear input 1.525786e-03
```



Generate HDL Code for Divide Implementation

Check HDL settings of the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_divide_shiftadd_control')
```

```
%% Set Model 'hdlcoder_divide_shiftadd_control' HDL parameters
hdlset_param('hdlcoder_divide_shiftadd_control', 'Backannotation', 'on');
hdlset_param('hdlcoder_divide_shiftadd_control', 'HDLSubsystem', 'hdlcoder_divide_shiftadd_contro
hdlset_param('hdlcoder_divide_shiftadd_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_divide_shiftadd_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_divide_shiftadd_control', 'TargetFrequency', 500);
hdlset_param('hdlcoder_divide_shiftadd_control', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_divide_shiftadd_control/Divide', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_divide_shiftadd_control/Divide/Divide', 'Architecture', 'ShiftAdd');
% Set Product HDL parameters
hdlset_param('hdlcoder_divide_shiftadd_control/Divide/Divide', 'LatencyStrategy', 'Max');

hdlset_param('hdlcoder_divide_shiftadd_control/Divide/LumpLatency', 'Architecture', 'MATLAB Datap
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_divide_shiftadd_control/Divide/LumpLatency', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_divide_shiftadd_control/Divide/ValidLine', 'Architecture', 'MATLAB Datapat
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_divide_shiftadd_control/Divide/ValidLine', 'FlattenHierarchy', 'on');
```

To generate HDL code for the Divide block in the model, use `makehdl` function.

```
makehdl('hdlcoder_divide_shiftadd_control/Divide')
close_system('hdlcoder_divide_shiftadd_control')
close all;
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_divide_shiftadd_control')">hdlcode
### Generating HDL for <a href="matlab:open_system('hdlcoder_divide_shiftadd_control/Divide')">hd
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_divide_
### Running HDL checks on the model 'hdlcoder_divide_shiftadd_control'.
### Begin compilation of the model 'hdlcoder_divide_shiftadd_control'...
### Begin compilation of the model 'hdlcoder_divide_shiftadd_control'...
### Working on the model 'hdlcoder_divide_shiftadd_control'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_divide_shiftadd_control', 'Ge
### Begin model generation 'gm_hdlcoder_divide_shiftadd_control'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_divide_shiftadd
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdl_
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsrc
### Begin VHDL Code Generation for 'hdlcoder_divide_shiftadd_control'.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_divide_shiftadd_control', 'Tr
### Working on hdlcoder_divide_shiftadd_control/Divide/Divide as hdl_prj\hdlsrc\hdlcoder_divide_s
```

```
### Working on hdlcoder_divide_shiftadd_control/Divide as hdl_prj\hdlsrc\hdlcoder_divide_shiftadd
### Code Generation for 'hdlcoder_divide_shiftadd_control' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_divide_shiftadd_control' complete with 0 errors, 0 warnings, and 1 me
### HDL code generation complete.
```

Divide Block Synthesis Performance

This figure shows the Divide block synthesis performance on the Xilinx® Virtex® 7 and Intel® Stratix® V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
432	530	1338	1972

Intel Quartus Stratix V (5SEE9F45C2)

Fmax (MHz)	LABs	ALMs	Registers
286	173	1035	2823

Implement Sine and Cosine Block with Control Signals

This example shows how to implement the control-signal based SinCos block and use it generate HDL code.

Open and Run Simulink Model

Specify the input data as a linear sweep through values in the range [-pi, pi]. You can change these values according to your requirements.

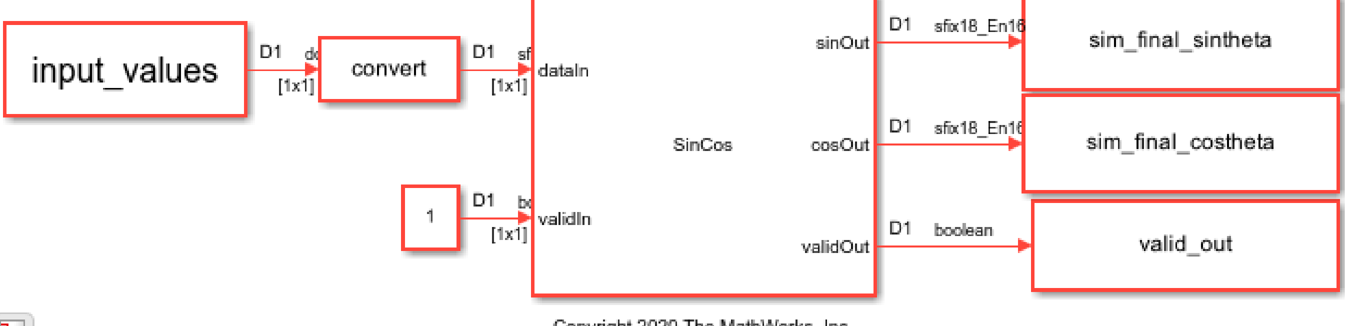
```
input_values = (-pi:.01/(2*pi):pi)';
```

Specify the word length for fixed-point datatypes and the latency for the model. The latency depends up on the number of iterations.

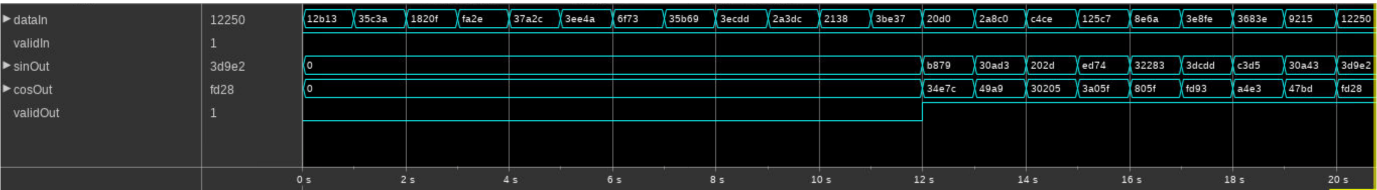
```
WL_SinCos = 18; latency_SinCos = 12;
```

Open the `hdlcoder_sincos_control` model and specify a stop time sufficient to process all the input combinations. The model has SinCos block that implements the SinCos using CORDIC algorithm. The other trigonometric function blocks, such as Sin, Cos, and Cos + jSin, use the same CORDIC approximation method.

```
stoptime_sincos = length(input_values)-1+latency_SinCos;
open_system('hdlcoder_sincos_control')
sim('hdlcoder_sincos_control')
```



This figure shows the waveform when you simulate the above model. The `dataOut` output is valid when `validOut` is `1`.

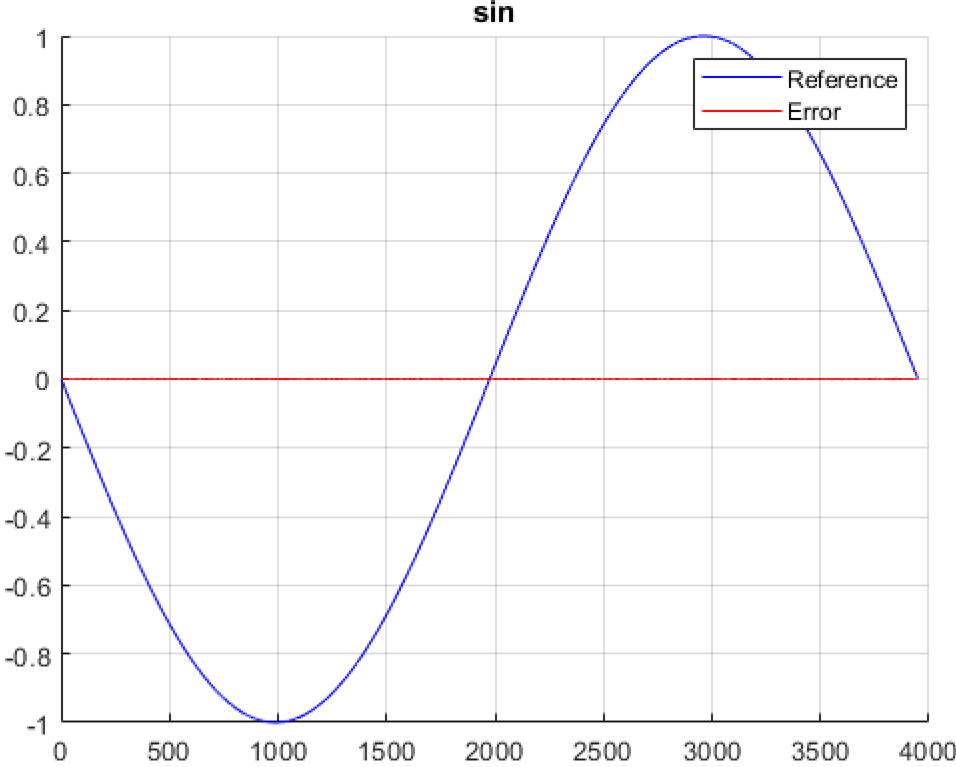


Validate Simulink Output By Using Reference Output

To validate the output of the Simulink model, compare the this output with a reference value. To obtain the reference values, use the `sin` and `cos` function. Compute the reference value for Sine output by using the `sin` function.

```
comparison_plot_sincos(sin(input_values),sim_final_sintheta(valid_out),5,'sin');
```

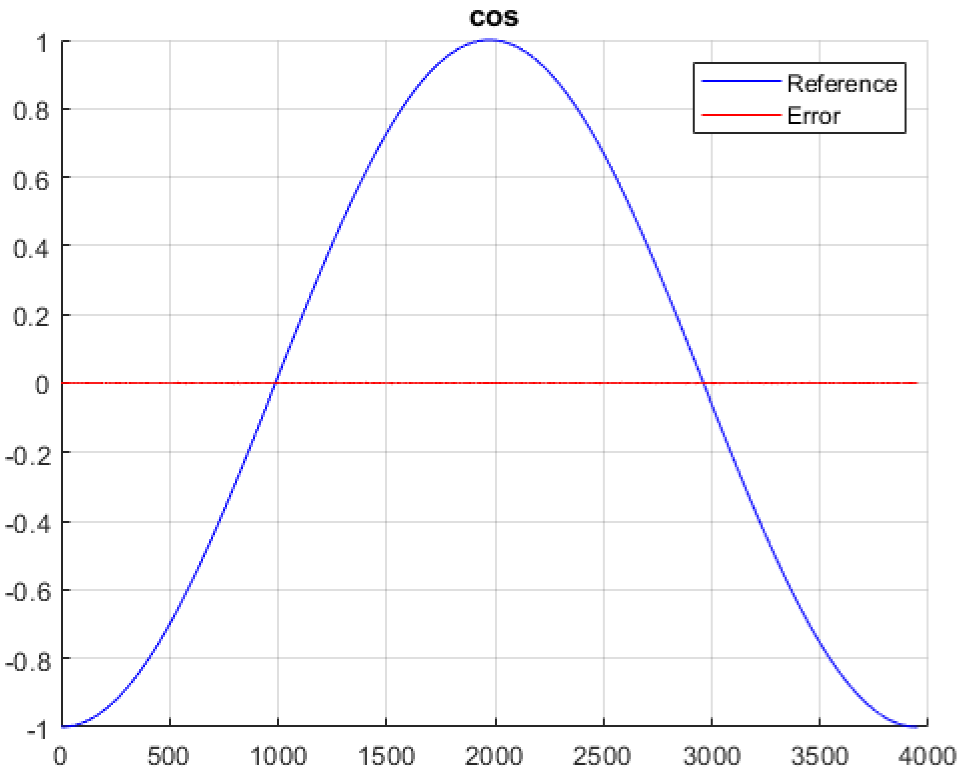
```
Maximum Error sin 1.005291e-03
```



Compute the reference value for Cosine output by using the `cos` function. The maximum error value is significantly smaller than the output of the model.

```
comparison_plot_sincos(cos(input_values),sim_final_costheta(valid_out),6,'cos');
```

```
Maximum Error cos 1.036532e-03
```



Generate HDL Code for SinCos Implementation

Check the HDL settings of the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_sincos_control')
```

```
%% Set Model 'hdlcoder_sincos_control' HDL parameters
hdlset_param('hdlcoder_sincos_control', 'HDLSubsystem', 'hdlcoder_sincos_control/SinCos');
hdlset_param('hdlcoder_sincos_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_sincos_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_sincos_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_sincos_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_sincos_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_sincos_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_sincos_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_sincos_control', 'TargetFrequency', 500);

hdlset_param('hdlcoder_sincos_control/SinCos/LumpLatency', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sincos_control/SinCos/LumpLatency', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_sincos_control/SinCos/LumpLatency1', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sincos_control/SinCos/LumpLatency1', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_sincos_control/SinCos/SinCos', 'Architecture', 'Cordic');

hdlset_param('hdlcoder_sincos_control/SinCos/ValidLine', 'Architecture', 'MATLAB Datapath');
```

```
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sincos_control/SinCos/ValidLine', 'FlattenHierarchy', 'on');
```

To generate HDL code for the SinCos block in the model, use the `makehdl` function.

```
makehdl('hdlcoder_sincos_control/SinCos')
close_system('hdlcoder_sincos_control')
close all;
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_sincos_control')">hdlcoder_sincos_
### Generating HDL for <a href="matlab:open_system('hdlcoder_sincos_control/SinCos')">hdlcoder_si
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_sincos_
### Running HDL checks on the model 'hdlcoder_sincos_control'.
### Begin compilation of the model 'hdlcoder_sincos_control'...
### Begin compilation of the model 'hdlcoder_sincos_control'...
### Working on the model 'hdlcoder_sincos_control'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_sincos_control', 'GenerateMod
### Begin model generation 'gm_hdlcoder_sincos_control'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_sincos_control\
### Begin VHDL Code Generation for 'hdlcoder_sincos_control'.
### Working on hdlcoder_sincos_control/SinCos/SinCos as hdl_prj\hdlsrc\hdlcoder_sincos_control\Si
### Working on hdlcoder_sincos_control/SinCos as hdl_prj\hdlsrc\hdlcoder_sincos_control\SinCos.vh
### Code Generation for 'hdlcoder_sincos_control' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_sincos_control' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

SinCos Block Synthesis Performance

This figure shows the SinCos block synthesis performance on the Xilinx Virtex 7 and Intel Stratix V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
573	226	680	606

Intel Quartus Stratix V (5SEE9F45C2)

Fmax (MHz)	LABs	ALMs	Registers
568	61	352	610

Implement Atan2 Block with Control Signals

This example shows how to implement the control-signal based Atan2 block and use it to generate HDL code.

Open and Run Simulink Model

Specify the input data as a linear sweep through values in the range [-pi, pi]. You can change these values according to your requirements.

```
input_values = (-pi:.01/(2*pi):pi)';
RADIUS = 10.^(-2.5:.25:0);
```

Specify the word length for fixed-point data types and the latency for the model. The latency depends on the number of iterations.

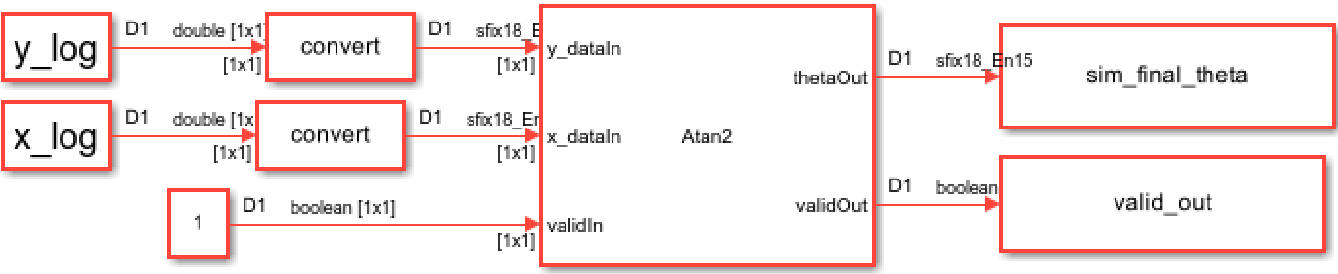
```
WL_atan2 = 18; latency_atan2 = 14;
```

Map the input data to x- and y-coordinate values.

```
x_log = zeros(length(input_values)*length(RADIUS),1);
y_log = zeros(length(input_values)*length(RADIUS),1);
for outerindex = 0:length(RADIUS)-1
    for index = 1:length(input_values)
        input = input_values(index);  % access current value
        y = RADIUS(outerindex+1)*sin(input);                % compute y
        x = RADIUS(outerindex+1)*cos(input);                % compute x
        addr = outerindex*length(input_values)+index;
        y_log(addr) = y;
        x_log(addr) = x;
    end
end
```

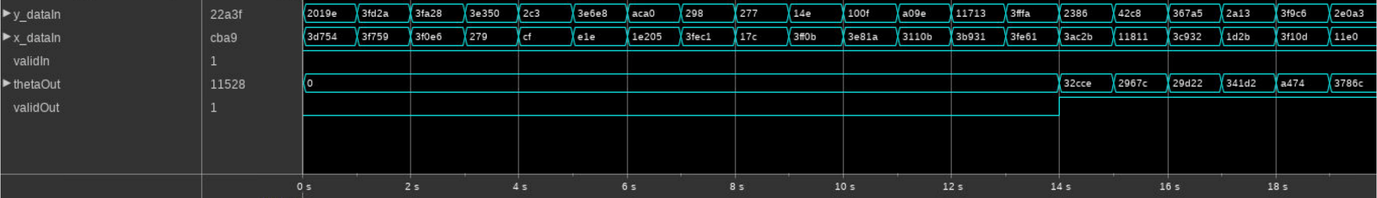
Open the `hdlcoder_atan2_control` model and specify a stop time sufficient to process all the input combinations. The model has an Atan2 block that implements the four-quadrant arctangent operation using the CORDIC algorithm.

```
stoptime_atan2 = length(x_log)-1+latency_atan2;
close all
open_system('hdlcoder_atan2_control')
sim('hdlcoder_atan2_control')
```



Copyright 2020 The MathWorks, Inc.

This figure shows the output waveform when you simulate the model. The `dataOut` output is valid when `validOut` is `1`.

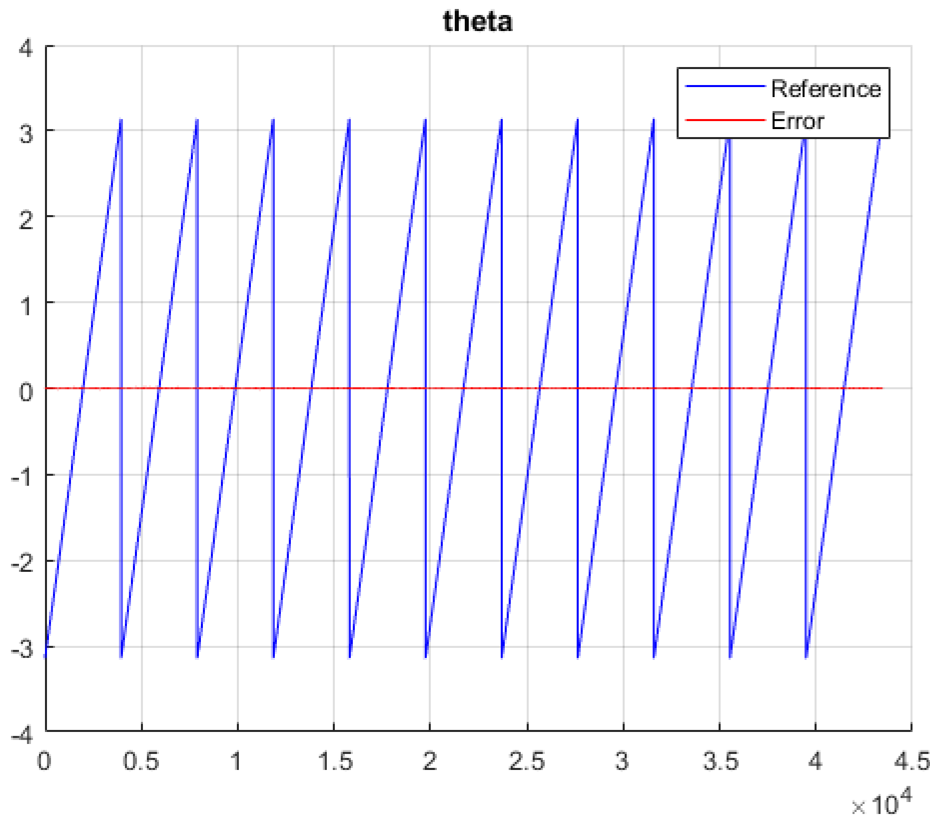


Validate Simulink Output By Using Reference Output

To validate the output of the Simulink model, compare this output with a reference value. To obtain the reference value, use the atan2 MATLAB function. The maximum error value is significantly smaller than the output of the model.

```
comparison_plot_atan2(atan2(y_log,x_log),sim_final_theta(valid_out),3,'theta');
```

```
Maximum Error theta 7.233221e-03
```



Generate HDL Code for Atan2 Implementation

Check the HDL settings of the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_atan2_control')
```

```
%% Set Model 'hdlcoder_atan2_control' HDL parameters
hdlset_param('hdlcoder_atan2_control', 'HDLSubsystem', 'hdlcoder_atan2_control/Atan2');
```

```
hdlset_param('hdlcoder_atan2_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_atan2_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_atan2_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_atan2_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_atan2_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_atan2_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_atan2_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_atan2_control', 'TargetFrequency', 500);

hdlset_param('hdlcoder_atan2_control/Atan2/Atan2', 'Architecture', 'Cordic');

hdlset_param('hdlcoder_atan2_control/Atan2/LumpLatency', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_atan2_control/Atan2/LumpLatency', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_atan2_control/Atan2/ValidLine', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_atan2_control/Atan2/ValidLine', 'FlattenHierarchy', 'on');
```

To generate HDL code for the Atan2 block in the model, use the `makehdl` function.

```
makehdl('hdlcoder_atan2_control/Atan2')
close_system('hdlcoder_atan2_control')
close all;
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_atan2_control')">hdlcoder_atan2_co
### Generating HDL for <a href="matlab:open_system('hdlcoder_atan2_control/Atan2')">hdlcoder_atan
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_atan2_c
### Running HDL checks on the model 'hdlcoder_atan2_control'.
### Begin compilation of the model 'hdlcoder_atan2_control'...
### Begin compilation of the model 'hdlcoder_atan2_control'...
### Working on the model 'hdlcoder_atan2_control'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_atan2_control', 'GenerateMode
### Begin model generation 'gm_hdlcoder_atan2_control'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_atan2_control\g
### Begin VHDL Code Generation for 'hdlcoder_atan2_control'.
### Working on hdlcoder_atan2_control/Atan2/Atan2 as hdl_prj\hdlsrc\hdlcoder_atan2_control\Atan2_
### Working on hdlcoder_atan2_control/Atan2 as hdl_prj\hdlsrc\hdlcoder_atan2_control\Atan2.vhd.
### Code Generation for 'hdlcoder_atan2_control' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlcoder_atan2_control' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

Atan2 Block Synthesis Performance

This figure shows the Atan2 block synthesis performance on the Xilinx® Virtex® 7 and Intel® Stratix® V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
544	231	725	620

Intel Quartus Stratix V (5SEE9F45C2)

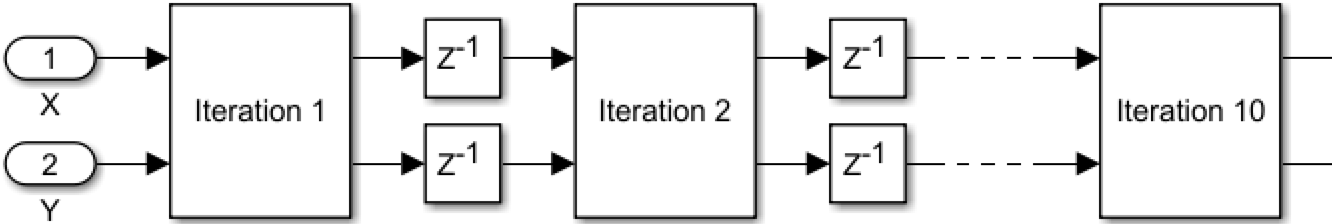
Fmax (MHz)	LABs	ALMs	Registers
565	61	342	674

Using ForEach Subsystems in HDL Coder

This example shows how you can use a For Each Subsystem to implement a streaming square root algorithm by cascading identical CORDIC iterations. You can then generate code for the algorithm by using HDL Coder™.

Using CORDIC Algorithm for Hardware Functions

CORDIC is an iterative algorithm that can be used to approximate fixed-point mathematics such as trigonometric functions, square root, and divide. The iterative core is composed of simple shift and add operations, allowing the algorithm to be implemented efficiently on FPGA or ASIC hardware. For low data rate applications, a single core can be reused to perform all iterations and achieve a very small area footprint. For applications requiring a new data sample to be processed at each clock cycle, a separate core can be used to calculate each iteration in a cascaded chain, as shown in the following diagram.



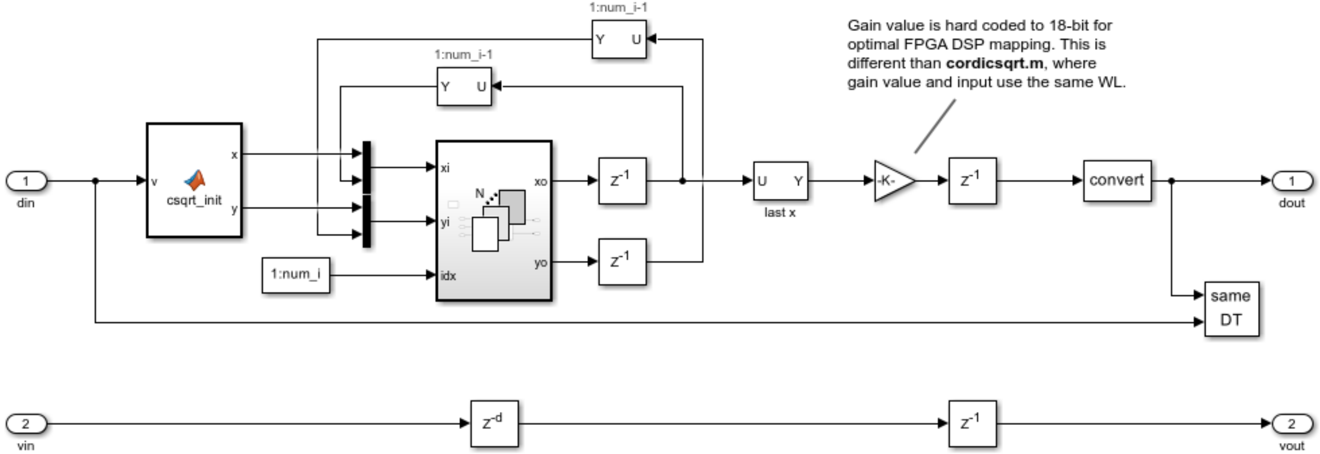
While it is straightforward to manually cascade the cores in Simulink®, the ability to automatically adjust the number of cores based on a parameter value would be highly desirable. You can do exactly that using a For Each Subsystem.

Cascade CORDIC Iterations Using For Each Subsystem

In this model, the iterative core is placed into a For Each Subsystem to be repeated *N* times, where *N* is the number of iterations defined in the upper-level block mask. The `N` core outputs form a vector at the For Each Subsystem output, where they are pipelined, and then fed back into the For Each Subsystem inputs. Outputs from core `(1:N-1)` are connected to inputs of core `(2:N)`, exactly the same as in the manually cascaded model.

A valid signal path is included to handle intermittent input data, and tested by inserting random gaps between valid data samples.

```
open_system('hdlcoder_foreach_cordic')
open_system('hdlcoder_foreach_cordic/For Each Cordic Sqrt','force')
```

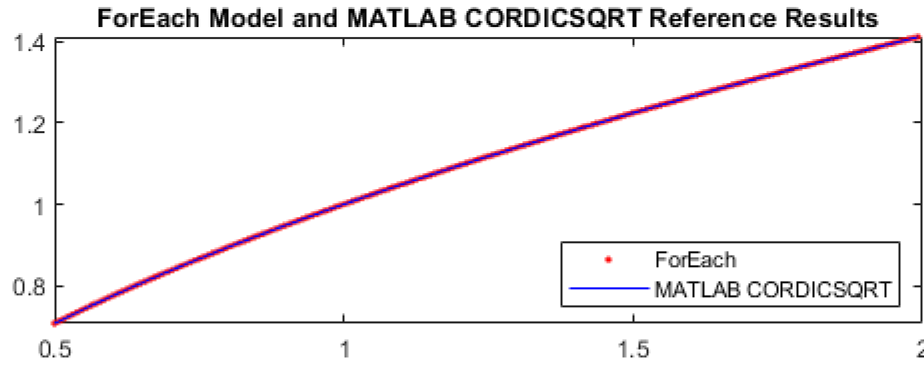


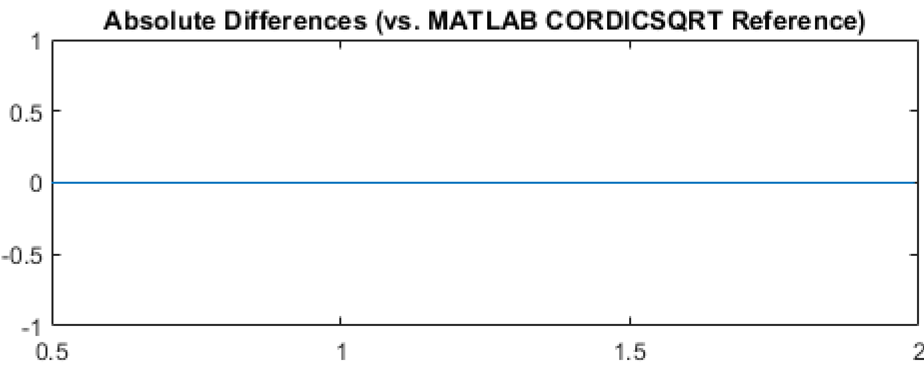
Compare Output to CORDIC Square Root Reference

Using a 14-bit signed input in the range of `[0.5,2)`, the output of the Simulink model matches the `cordicsqrt` reference function exactly. Input range outside of `[0.5,2)` is not expected to work because the example lacks a normalizer stage.

In addition, the final gain adjustment in the model uses an 18-bit gain parameter for optimal FPGA DSP mapping; while the `cordicsqrt` function matches the gain parameter word length to that of the gain input. This results in slight differences between the Simulink model output and the `cordicsqrt` function when other input data types are used.

```
slout = sim('hdlcoder_foreach_cordic');
data_out = slout.logsout.getElement('data out').Values.Data;
valid_out = slout.logsout.getElement('valid out').Values.Data;
data_out = data_out(valid_out);

ref_cordic = double(cordicsqrt(v_fix, niter));
data_in = double(v_fix);
data_out = double(data_out');

figure;
subplot(211);
plot(data_in, data_out, 'r.', data_in, ref_cordic, 'b-');
legend('ForEach', 'MATLAB CORDICSQRT', 'Location', 'SouthEast');
title('ForEach Model and MATLAB CORDICSQRT Reference Results');
subplot(212);
absErr = abs(ref_cordic - data_out);
plot(data_in, absErr);
title('Absolute Differences (vs. MATLAB CORDICSQRT Reference)');
```





Generate HDL Code

```
makehdl('hdlcoder_foreach_cordic/For Each Cordic Sqrt');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_foreach_cordic')">hdlcoder_foreach
### Generating HDL for <a href="matlab:open_system('hdlcoder_foreach_cordic/For Each Cordic Sqrt
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_foreach
### Running HDL checks on the model 'hdlcoder_foreach_cordic'.
### Begin compilation of the model 'hdlcoder_foreach_cordic'...
### Begin compilation of the model 'hdlcoder_foreach_cordic'...
### Working on the model 'hdlcoder_foreach_cordic'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_foreach_cordic', 'GenerateMod
### Begin model generation 'gm_hdlcoder_foreach_cordic'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_foreach_cordic\gm_hdlco
### Begin VHDL Code Generation for 'hdlcoder_foreach_cordic'.
### Unused logic removed during HDL code generation. To highlight the logic removed, click the fo
### To clear highlighting, click the following MATLAB script: hdlsrc\hdlcoder_foreach_cordic\clea
### Working on hdlcoder_foreach_cordic/For Each Cordic Sqrt/MATLAB Function2 as hdlsrc\hdlcoder_f
### Working on hdlcoder_foreach_cordic/For Each Cordic Sqrt/For Each Subsystem/MATLAB Function1 a
### Working on hdlcoder_foreach_cordic/For Each Cordic Sqrt/For Each Subsystem as hdlsrc\hdlcoder
### Working on hdlcoder_foreach_cordic/For Each Cordic Sqrt as hdlsrc\hdlcoder_foreach_cordic\For
### Generating package file hdlsrc\hdlcoder_foreach_cordic\For_Each_Cordic_Sqrt_pkg.vhd.
### Code Generation for 'hdlcoder_foreach_cordic' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
```

```
### HDL check for 'hdlcoder_foreach_cordic' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

Additional Modeling Guidelines

Observe the following guidelines when cascading blocks in your algorithm using For Each Subsystem:

- Since For Each Subsystem is atomic, the connection between output of block X and input of block X+1 creates an artificial algebraic loop. To break this loop, place pipeline registers between cascading blocks outside of the For Each Subsystem, as demonstrated in this example.
- A mux block is used to concatenate external input and outputs of block `(1:N-1)` to form the inputs of the For Each Subsystem. This requires the cascading blocks to use the same input and output data types.

Related Topics

- “Generate HDL Code for Blocks Inside For Each Subsystem” on page 14-61
- “Compute Square Root Using CORDIC”

Generate HDL Code for Blocks Inside For Each Subsystem

This example shows how to use blocks inside a For Each Subsystem in your Simulink® model, and then generate HDL code.

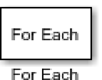
Why Use a For Each Subsystem?

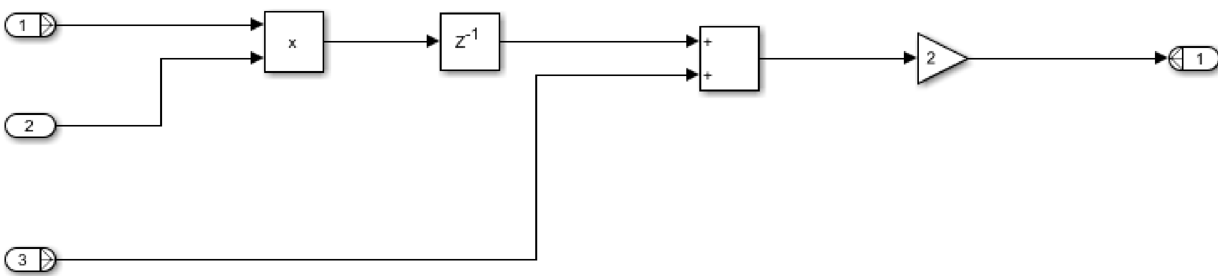
To repeatedly perform the same algorithm on individual elements or subarrays of the input signals, use the For Each Subsystem block. The set of blocks within the Subsystem replicate the algorithm that is applied to individual elements or equally divided subarrays of the input signals. Using the For Each Subsystem block, you do not have to create and connect replicas of a Subsystem block to model the same algorithm. The For Each Subsystem:

- Supports vector or 2-D matrix processing, which reduces the simulation time of your model. You can process individual elements or subarrays of an input signal simultaneously.
- Improves code readability by using a for-generate loop in the generated HDL code. The for-generate loop reduces the number of lines of code, which can otherwise result in hundreds of lines of code for large vector signals.
- Supports HDL code generation for all data types, Simulink blocks, and predefined and user-defined system objects.
- Supports optimizations on and inside the block, such as resource sharing and pipelining. The parallel processing capability of the For Each Subsystem block combined with the optimizations that you specify produces high performance on the target FPGA device.

Modeling With the For Each Subsystem

Open the `foreach_subsystem_example1` model. You see this simple algorithm modeled inside a For Each Subsystem block.



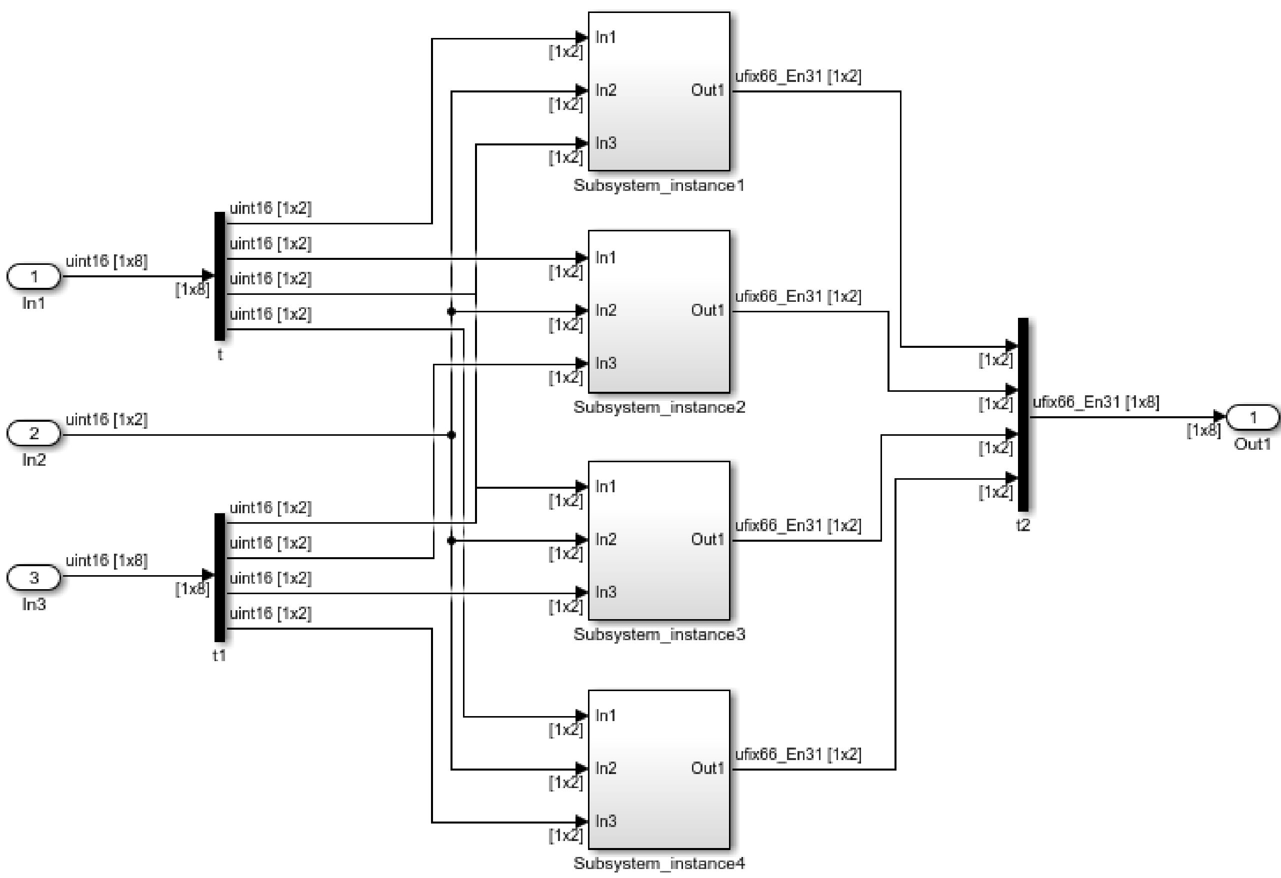


When you simulate the model, you see that the input signals `In1` and `In3` are partitioned into subarrays. To see this partitioning, double-click the For Each block. The block parameters **Partition Dimension** and **Partition Width** specify the dimension through which the input signal is partitioned and the width of each partition slice respectively. Based on the input signal sizes and the partitioning that you specify, the For Each Subsystem determines the number of iterations that it requires to compute the algorithm.

In this example, the input signals In1 and In3 of size 8 are partitioned into four subarrays, each of size 2. The input signal In2 of size 2 is not partitioned. To compute the algorithm, the For Each

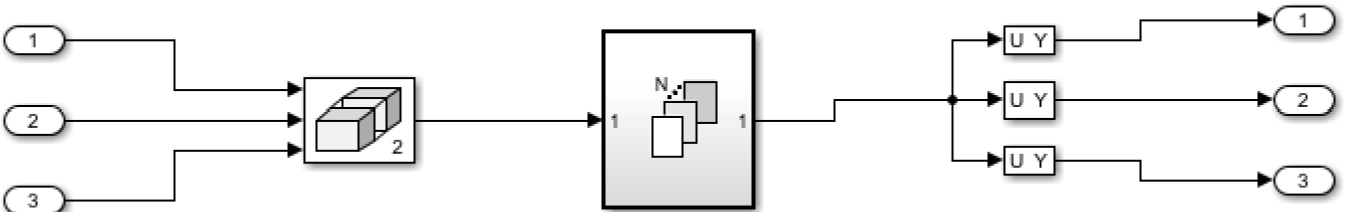
Subsystem requires four iterations, with each iteration repeating the algorithm on each of the four subarrays of In1 and In3.

The For Each Subsystem simplifies modeling of vectorized algorithms. This figure shows how you can model the same algorithm by creating multiple subsystem instances. This model can become graphically complex and difficult to maintain.

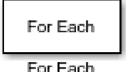


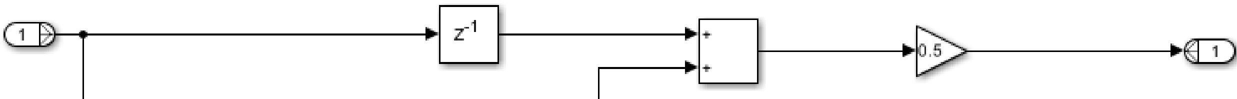
Using Matrix Input Signals

The For Each Subsystem supports 2-D matrix input for HDL code generation. For example, the `foreach_subsystem_example2` model shows a simple multi-channel filter operation. HDL code generation is not supported for matrices at the input and output ports of the HDL DUT, so the model separates the channels at the DUT subsystem boundary.



The For Each subsystem averages the samples on each channel. The generated HDL code will contain three copies of the logic inside the For Each Subsystem, and each operates on a 4x1 vector.





Using Complex Data Signals

The block does not support complex data types as inputs for HDL code generation. To input a complex signal, you can convert this signal to an array of signals, and then input to the block.

To perform the same algorithm on both real and imaginary parts of the signal:

1. Separate the signal into real and imaginary parts by using a Complex to Real-Imag block.
2. Create a vector signal that consists of the real and imaginary parts by using a Mux block.

You can then input this vector to the For Each Subsystem block and replicate the same computation on both the real and imaginary parts. At the output of the For Each Subsystem, you can convert the vector output back to a complex signal. Use a Demux block to separate the real and imaginary scalar parts, and then input the scalars to the Real-Imag to Complex block.

Generate HDL Code

To generate HDL code, in the `foreach_subsystem_example1` model, right-click the `Subsystem_Foreach` block and select **HDL Code > Generate HDL for Subsystem**.

To see the generated HDL code for the `Subsystem_Foreach` block, in the MATLAB® Command Window, click the `Subsystem_Foreach.vhd` file. In the VHDL® code snippet, you see this for-generate loop in the HDL code. This loop creates four subsystem instances, with each instance performing the algorithm on size 2 subarrays of inputs In1 and In3.

```
BEGIN
  -- <S2>/For Each Subsystem
  GEN_LABEL: FOR k IN 0 TO 3 GENERATE
    u_For_Each_Subsystem : For_Each_Subsystem
      PORT MAP( clk => clk,
                reset => reset,
                enb => clk_enable,
                In1 => In1(2*k TO 2*(k+1) - 1),  -- uint16 [2]
                In2 => In2,  -- uint16 [2]
                In3 => In3(2*k TO 2*(k+1) - 1),  -- uint16 [2]
                Out1 => For_Each_Subsystem_out1(2*k TO 2*(k+1) - 1)
                );
  END GENERATE;
```

You can specify optimizations that change the contents of the subsystems that the For Each Subsystem instantiates. In such cases, the code generator does not use for-generate loops in the HDL code. The HDL code does not contain for-generate loops if you have:

- Bus or complex input signals.
- Resource sharing and streaming optimizations on the subsystem.
- Vector inputs that get partitioned into nonscalar signals in the Verilog® code. To obtain for-generate loops in the Verilog code, partition the vector signal to scalars.

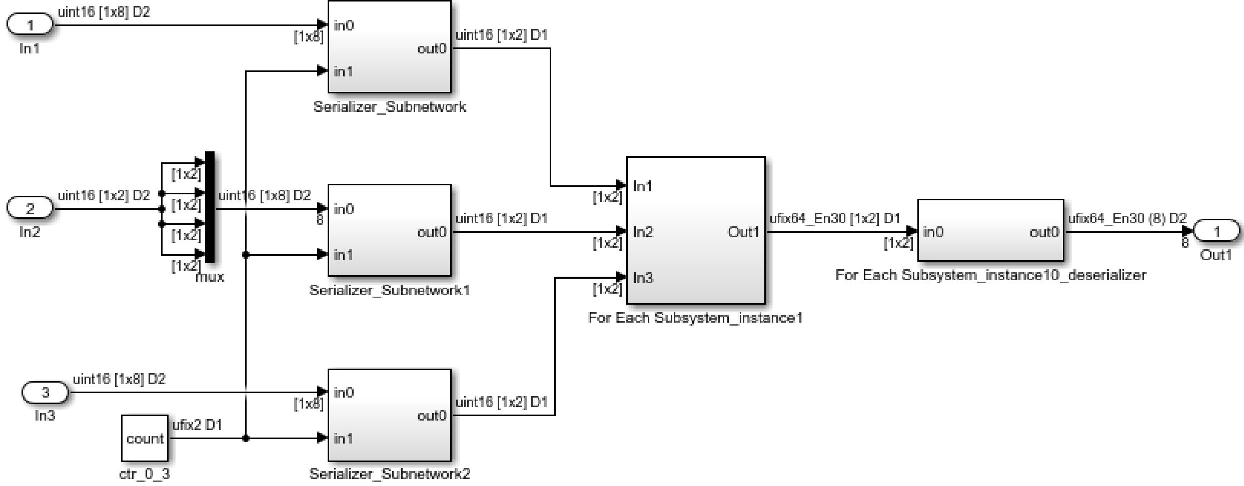
Optimize the For Each Subsystem Algorithm

To optimize the algorithm contained within the For Each Subsystem, you can enable optimizations such as resource sharing and streaming on the DUT that contains the For Each Subsystem. For example, by using the resource sharing optimization, you can share multiple Subsystem instances that are created by the For Each Subsystem. This optimization reuses the algorithm modeled by the Subsystem across multiple instances and reduces the area usage on the target device.

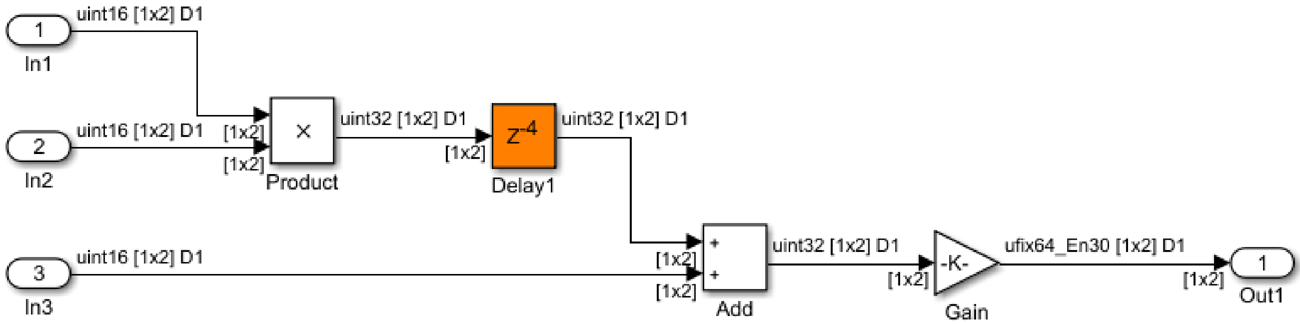
Note: When you enable optimizations on the For Each Subsystem, the generated HDL code does not contain for-generate loops.

This example shows how to use the resource sharing optimization on the For Each Subsystem. To share resources, select the Subsystem block that contains the For Each Subsystem and then specify the **Sharing Factor**. In this example, right-click the `Subsystem_Foreach` block and select **HDL Code > HDL Block Properties**. Set the **Sharing Factor** to 4, because the For Each Subsystem generates four Subsystem instances. Then, generate HDL code for the `Subsystem_Foreach` block.

To see the effect of the resource sharing optimization, at the command-line, enter `gm_foreach_subsystem_example1` to open the generated model. In the generated model, you see that the optimization shared the four subsystem instances generated by the For Each Subsystem into one Subsystem `For Each Subsystem_Instance1`.



If you double-click the `For Each Subsystem_Instance1` block, you see the algorithm computed for the size 2 subarrays of inputs In1 and In3.



To learn more about the resource sharing optimization, see “Resource Sharing” on page 21-45.

See Also

For Each Subsystem

Field-Oriented Control of a Permanent Magnet Synchronous Machine

In this example you will review a Field-Oriented Control (FOC) algorithm for a Permanent Magnet Synchronous Machine (PMSM). You will test the control algorithm with closed loop system simulation then generate HDL code for the control algorithm. You will also see how tunable parameter data is specified and how corresponding HDL port entities are generated.

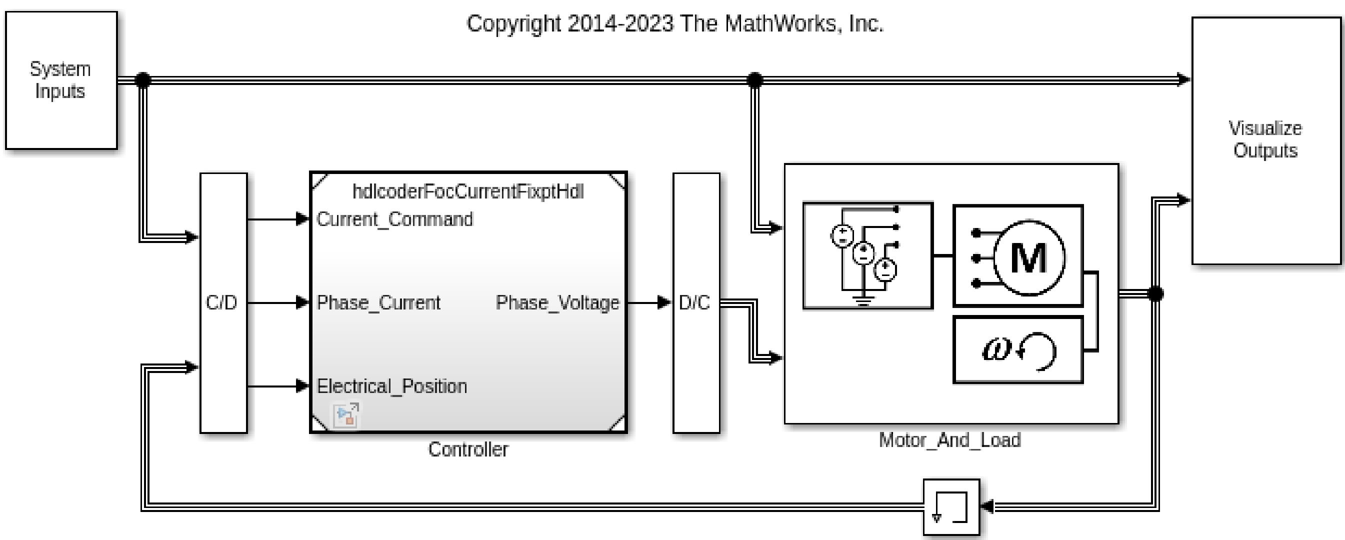
Introduction

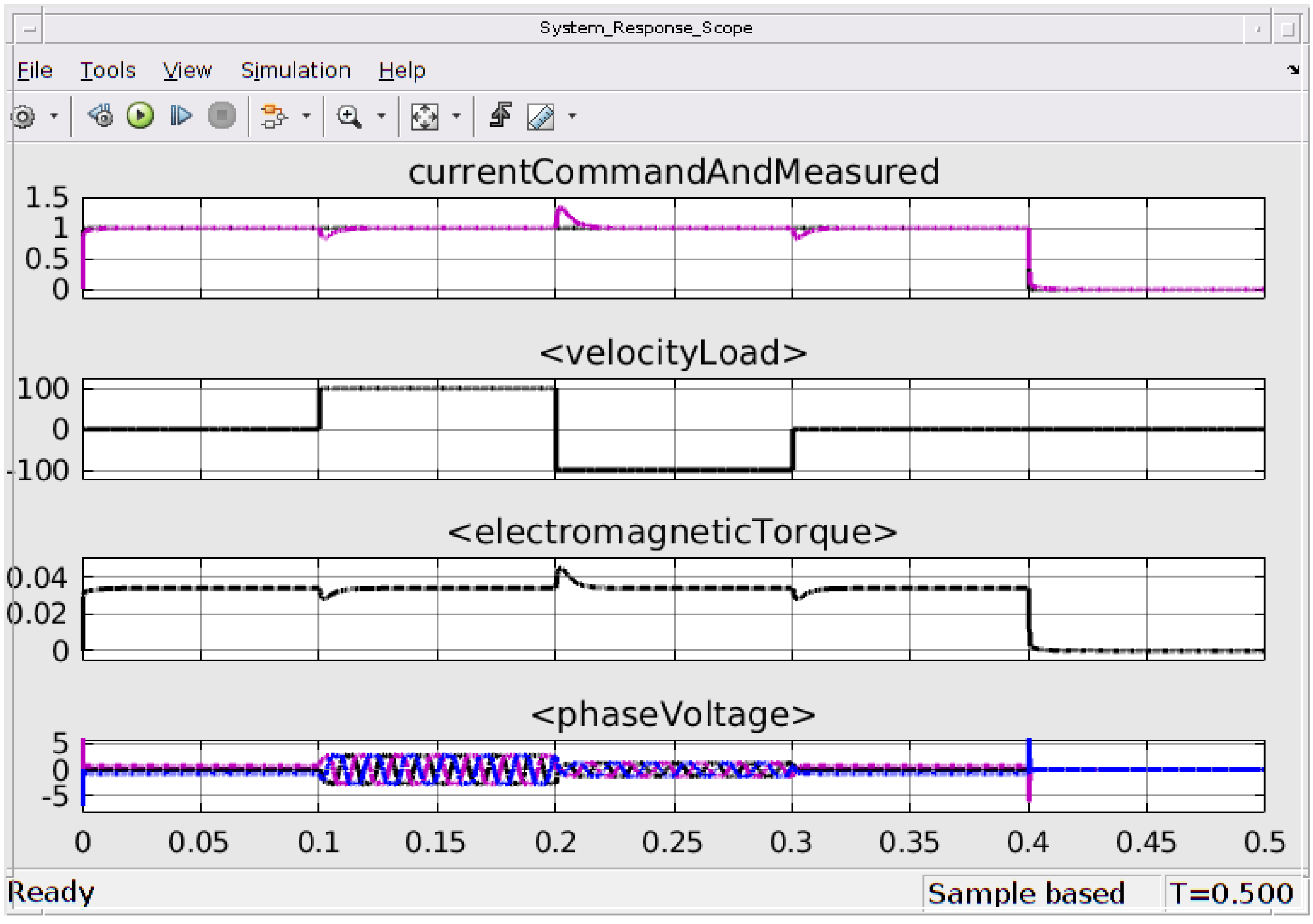
The example is partitioned such that you can generate code for the control algorithm as well as verify the behavior of the control algorithm using a simulation test bench. Simscape (TM) Electrical (TM) is required to run the system simulation test bench model `hdlcoderFocCurrentTestBench.slx` but is not required to generate code from the control algorithm model `hdlcoderFocCurrentFixptHdl.slx`.

Verify Behavior through Simulation

In this example FOC is used to regulate phase current to control torque of an electric machine. You can simulate a test bench to explore the behavior of the system. During the simulation, the solver may generate warnings related to zero crossing when the velocity load changes abruptly. You can disable these warnings during the simulation.

```
hasSimPowerSystems = license ('test', 'Power_System_Blocks');
if hasSimPowerSystems
   open_system('hdlcoderFocCurrentTestBench')
   set_param('hdlcoderFocCurrentTestBench','IgnoredZcDiagnostic','none');
   sim('hdlcoderFocCurrentTestBench')
   set_param('hdlcoderFocCurrentTestBench','IgnoredZcDiagnostic','warn');
end
```



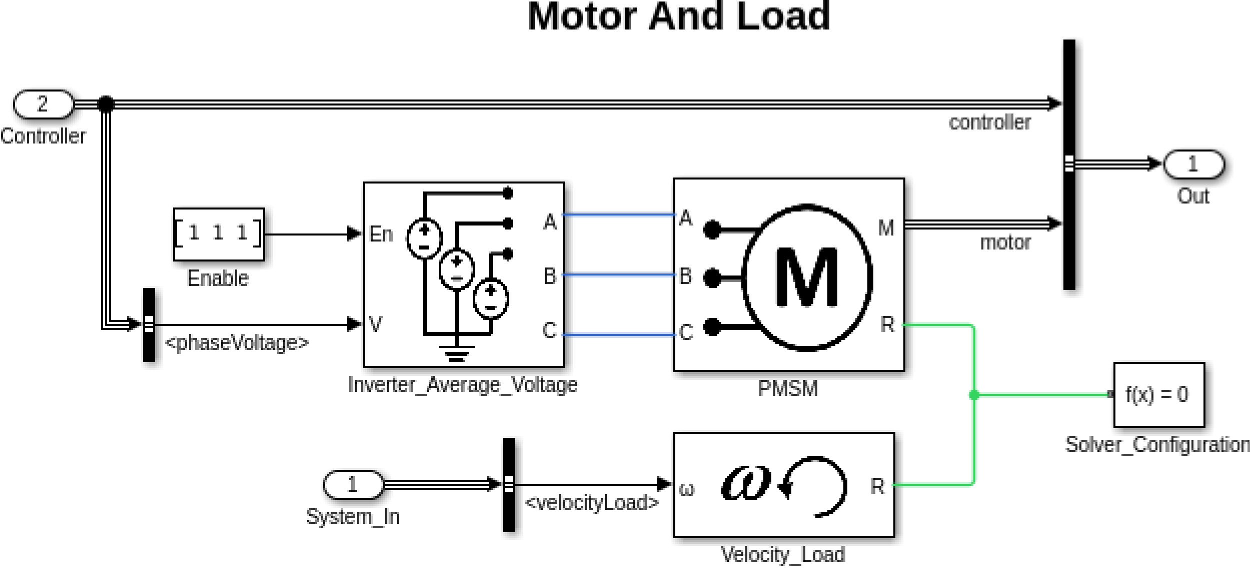


The scope shows that a 1 Amp step current command is requested and the load velocity changes between locked rotor (zero), +100 rad/sec, and -100 rad/sec. The current command represents a quadrature current command to a non-salient PMSM. (The controller regulates the direct current to zero.) Note that for this motor and controller, the electromagnetic torque closely follows the measured quadrature current of the motor.

Explore Plant Specification

In the Motor_And_Load subsystem you will see a mathematical model of the components being controlled. An average model of the inverter is used to drive a constant parameter dq voltage equation model of a PMSM which is connected to a velocity load.

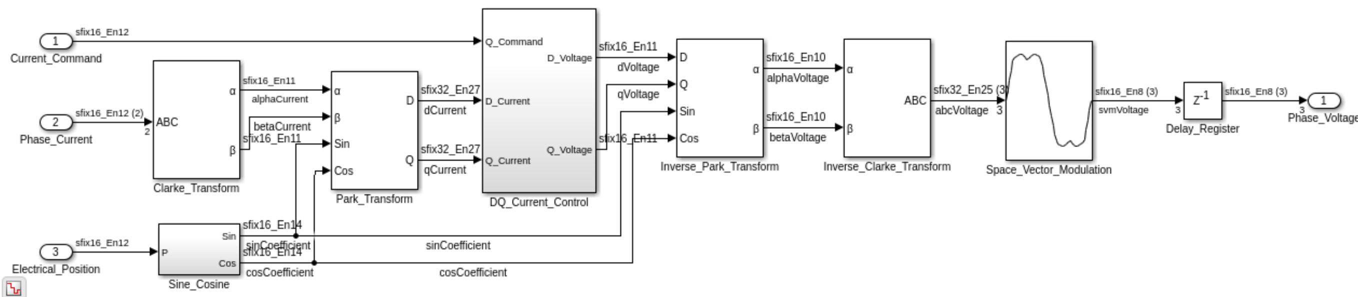
```
if hasSimPowerSystems
   open_system('hdlcoderFocCurrentTestBench/Motor_And_Load')
end
```



Explore Control Algorithm Specification

The FOC current control algorithm is specified in a separate model. In the control algorithm, the electrical equations of the machine are projected from the three-phase stationary reference frame onto a two phase rotating reference frame using Clarke and Park transforms. This simplifies the control by removing time and position dependencies. Space Vector Modulation enables the controller to achieve greater voltage across the phases than if just the sinusoidal outputs of the inverse Clarke transform were used.

```
load_system('hdlcoderFocCurrentFixptHdl');
open_system('hdlcoderFocCurrentFixptHdl/FOC_Current_Control')
```



Explore Data Specification

Both the controller and the plant (i.e. motor and load) reference data from the MATLAB workspace. A data definition file creates this data and is automatically run within the PreLoadFcn callback of the system test bench model.

```
%edit('hdlcoderFocCurrentFixptHdlData.m')
%
```

When you review this file, notice that the parameters paramCurrentControlP and paramCurrentControlI are specified as Simulink.Parameters whose storage class is set to `ExportedGlobal`. This tells HDL Coder to generate entity ports for these parameters instead of constant values.

Generate HDL Code for Control Algorithm

Before generating HDL code, it is important to ensure that the model adheres to certain important settings for HDL code generation. Below are some of the main steps:

- Create a DUT subsystem: For HDL code generation it is always better to create a DUT (Design Under Test) subsystem from which HDL code is generated. This subsystem serves several purposes including being a place-holder for HDL optimization settings.
- Setup for HDL: In order to get ready for HDL code generation, certain solver settings and model settings must be in place. The `hdlsetup` command takes care of all these settings and should be run before HDL code-generation.
- Checking sample times: Applying HDL optimizations requires all block sample times to be inferred as discrete. The main block-type to be cautious of are constants, which derive an 'inf' sample time, by default. We can find these blocks and explicitly set their sample-times to `-1` so they get the correct back-propagated sample times.

```
% You can generate and review the HDL code for the controller.
makehdl('hdlcoderFocCurrentFixptHdl/FOC_Current_Control');
```

```
### Generating HDL for 'hdlcoderFocCurrentFixptHdl/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurre
### Running HDL checks on the model 'hdlcoderFocCurrentFixptHdl'.
### Begin compilation of the model 'hdlcoderFocCurrentFixptHdl'...
### Begin compilation of the model 'hdlcoderFocCurrentFixptHdl'...
### Working on the model 'hdlcoderFocCurrentFixptHdl'...
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocCurrentFixptHdl', 'Generate
### Begin model generation 'gm_hdlcoderFocCurrentFixptHdl' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generating new validation model: <a href="matlab:open_system('hdlsrc/hdlcoderFocCurrentFixpt
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentFixptHdl'.
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Clarke_Transform as hdlsrc/hdlcode
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Contr
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Contr
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control as hdlsrc/hdlco
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Inverse_Clarke_Transform as hdlsrc
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Inverse_Park_Transform as hdlsrc/h
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Park_Transform as hdlsrc/hdlcoderF
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Sine_Cosine/Sine_Cosine_LUT as hdl
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Sine_Cosine as hdlsrc/hdlcoderFocC
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Space_Vector_Modulation as hdlsrc/
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control as hdlsrc/hdlcoderFocCurrentFixptH
### Generating package file hdlsrc/hdlcoderFocCurrentFixptHdl/FOC_Current_Control_pkg.vhd.
### Code Generation for 'hdlcoderFocCurrentFixptHdl' completed.
### Generating HTML files for code generation report at <a href="matlab:web('/home/amoses/Docume
### Creating HDL Code Generation Check Report file:///home/amoses/Documents/MATLAB/ExampleManage
### HDL check for 'hdlcoderFocCurrentFixptHdl' complete with 0 errors, 1 warnings, and 0 message
### HDL code generation complete.
```

Notice in the generated `hdlcoderFocCurrentFixptHdl.vhd` file that the entity has ports for paramCurrentControlP and paramCurrentControlI.

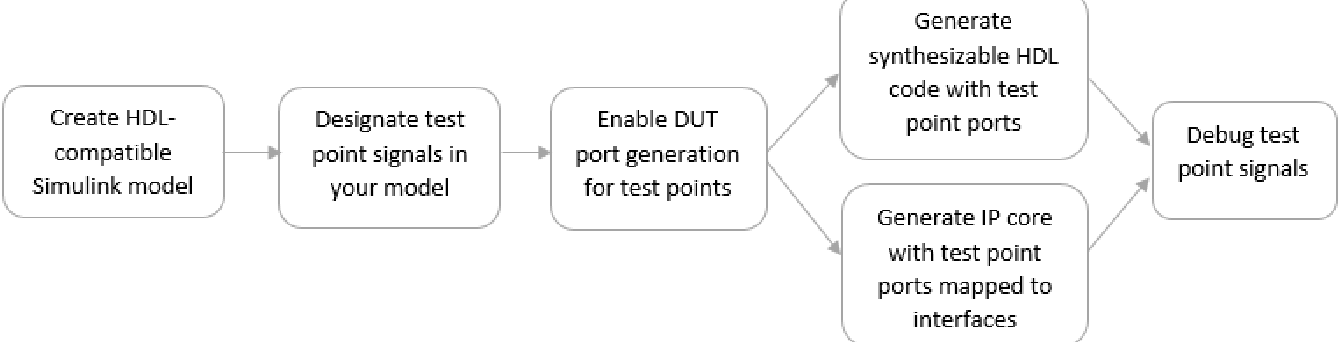
Model and Debug Test Point Signals with HDL Coder

This example shows how you can mark signals as test points in your Simulink® model and, after HDL code generation, debug the signals at the top level using the generated model or a test bench.

Why Use Test Points?

Test points are signals that you can use to easily debug and observe the simulation results at various points in your Simulink model. You can observe signals designated as test points with a Floating Scope block in a model. In Simulink, you can designate any signal in a model as a test point.

After code generation, you can observe test point signals at the DUT output ports and further debug the generated code in downstream workflows. This capability makes debugging your design easier because the code generator can propagate test point signals deep within the subsystem hierarchy to the DUT output ports.

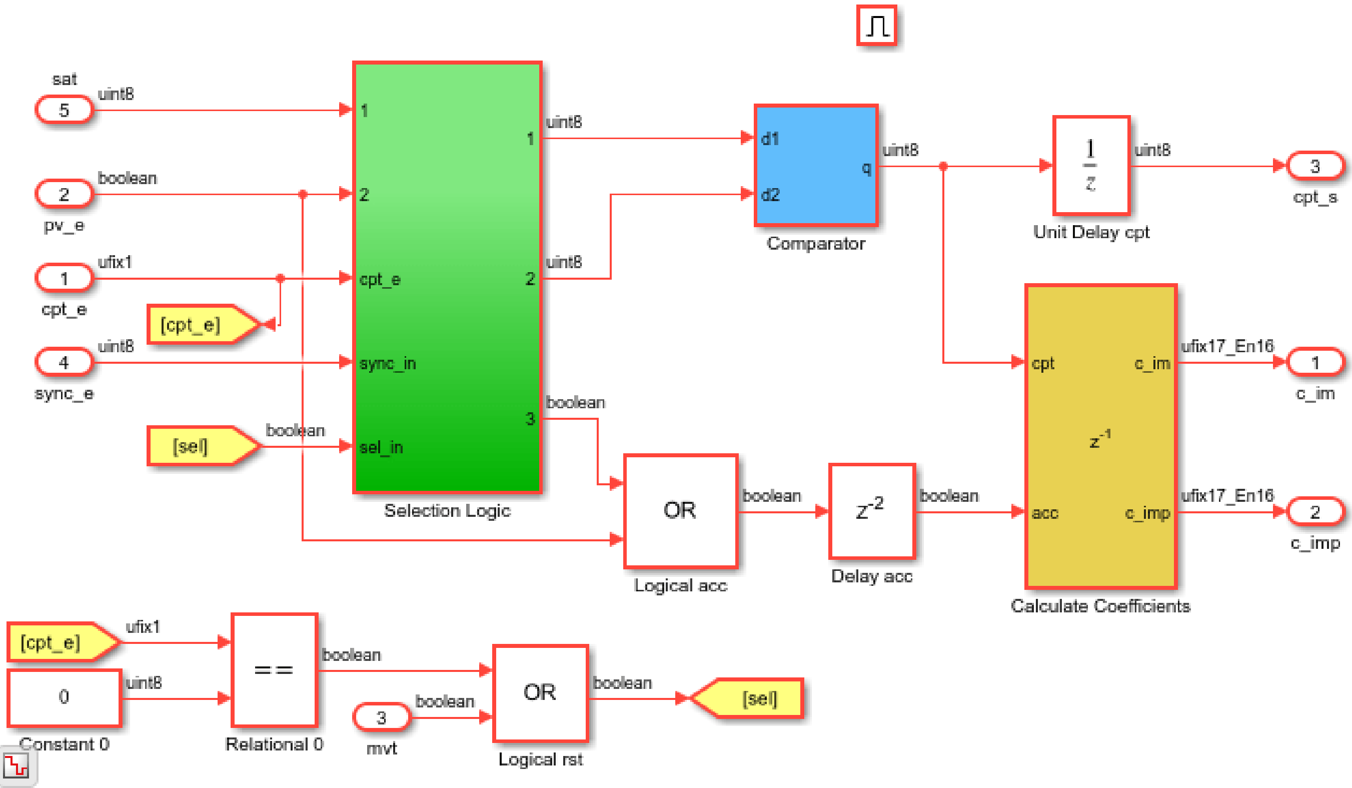


Create HDL-Compatible Model

Before you designate signals as test points and generate HDL code, make sure that the model you create is compatible for HDL code generation. See “Create HDL-Compatible Simulink Model”.

For this example, open the `hdlcoder_test_points` model that has been prepared for HDL code generation. The DUT is an Enabled Subsystem that calculates two coefficients based on inputs from a Selection Logic and a Comparator.

```
load_system('hdlcoder_test_points')
open_system('hdlcoder_test_points/DUT/MaJ Counter')
set_param('hdlcoder_test_points', 'SimulationCommand', 'update');
```



Designate Signals as Test Points

To debug internal signals in this model, mark them as test points in either of these ways:

- In the Simulink Editor, to open the Signal Properties dialog box, right-click the signal, and select **Properties**. Then, select **Test Point**.
- At the command line, get the handle to the output port of a block, and then set the port parameter `TestPoint` to `on`.

For example, enter these commands to designate the output signal from the Logical acc block that performs the OR operation as a test point.

```
portHandles = get_param('hdlcoder_test_points/DUT/MaJ Counter/Logical acc', 'portHandles');
outportHandle = portHandles.Outport;
set_param(outportHandle,'TestPoint','on');
```

If a block has more than one output port, specify the outport handle that you want to designate as testpoint. For example, to designate the second output of a Demux block as a test point, enter this command:

```
set_param(outportHandle(2),'TestPoint','on');
```

Simulink displays an indicator on each signal for which you enable the **Test point** setting. If you navigate the model, you see three additional test points. These test points are inside the Selection Logic subsystem, the Comparator Subsystem, and the Calculate Coefficients Subsystem blocks.

To learn more, see “Configure Signals as Test Points”.

Enable DUT Output Port Generation for Test Points

Before you generate HDL code, to debug signals that are designated as test points, enable HDL DUT port generation for the signals. When you generate code for the model, HDL Coder™ propagates these signals to the DUT as an additional output port.

To enable DUT output port generation for the `hdlcoder_simulink_test_points` model:

- In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Ports** tab, select **Enable HDL DUT port generation for test points**.
- At the command line, use the `EnableTestpoints` property.

```
hdlset_param('hdlcoder_test_points','EnableTestpoints','on')
```

To learn more about this parameter, see Enable HDL DUT output port generation for test points.

When you enable DUT output port generation for test points, you can disable delay balancing for the generated DUT output ports by using the Configuration Parameters dialog box and disabling **Balance delays for generated DUT output ports** or by using the command line and setting `BalanceDelaysForTestpoints` to `off`. To learn more about this parameter, see Balance delays for generated DUT output ports.

You can also treat a manually added output port as a test point and disable the HDL block property **BalanceDelays** on the DUT-level Outport block.

After you enable DUT port generation, you can run either of these workflows:

- Generate HDL code. To deploy the code onto a target FPGA, use the `Generic ASIC/FPGA` workflow in the HDL Workflow Advisor.
- Map test point ports to target platform interfaces, and generate an HDL IP core by using the `IP Core Generation` or `Simulink Real-Time FPGA I/O` workflows that use `Xilinx Vivado` or `Altera Quartus II` as the synthesis tools.

HDL Code Generation and FPGA Targeting

If you want to see the mapping between test point ports in the HDL code and the test point signals in your model, enable generation of the code generation report. The report displays the test point ports with links to the corresponding test point signals in your Simulink model.

For example, to enable report generation for the `hdlcoder_simulink_test_points` model:

- In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select **Generate resource utilization report**.
- To specify this setting at the command line, use the `ResourceReport` property.

```
hdlset_param('hdlcoder_test_points','ResourceReport','on')
```

To learn more about report generation, see “Create and Use Code Generation Reports” on page 23-2.

To generate HDL code:

- Right-click the `DUT` Subsystem and select **HDL Code > Generate HDL for Subsystem**.
- At the command line, run makehdl on the DUT Subsystem.

To deploy the code onto a target platform, use the `Generic ASIC/FPGA` workflow. In the HDL Workflow Advisor, on the **Set Target Device and Synthesis Tool** task, for **Target workflow**, select `Generic ASIC/FPGA`, specify the **Synthesis tool**, and then run the workflow.

When generating code, HDL Coder opens the Code Generation report. The Code Interface Report section contains links to the test point ports in the `Output Ports` section.

Output ports

Port Name	Datatype	Bits
ce_out	boolean	1
c_im	ufix17_En16	17
c_imp	ufix17_En16	17
cpt_s	uint8	8
tp_Sum_C_out1	ufix17_En16	17
tp_Relational_out1	boolean	1
tp_Sum_out1	uint8	8
tp_Logical_acc_out1	boolean	1
tp_Relational_0_out1	boolean	1

When you click the links in the test point ports, the code generator highlights the corresponding signals that you designated as test points in your Simulink model. Therefore, you can use the report to trace back from the test point port in the generated code to the test point signals in your Simulink model.

To see the test point ports in the generated HDL code, open the `DUT.v` file.

```
output  [16:0] c_imp;  // ufix17_En16
output  [7:0] cpt_s;  // uint8
output  [16:0] tp_Sum_C_out1;  // ufix17_En16 Testpoint port
output  tp_Relational_out1;  //  Testpoint port
output  [7:0] tp_Sum_out1;  // uint8 Testpoint port
output  tp_Logical_acc_out1;  //  Testpoint port
output  tp_Relational_0_out1;  //  Testpoint port
```

You can see the test point ports at the top level module declaration. These ports have the prefix `tp_` and a comment to indicate that they correspond to test point ports. If you specify VHDL® as the target language, you can see the test point ports in the entity declaration.

IP Core Generation and SoC Targeting

To generate an HDL IP core, open the HDL Workflow Advisor. In the Advisor:

1. On the **Set Target Device and Synthesis Tool** task, for **Target workflow**, select `IP Core Generation`, and specify a **Target platform** that uses `Xilinx Vivado` or `Altera Quartus II` as the **Synthesis tool**. If you use the `Simulink Real-Time FPGA I/O` workflow, specify a **Target platform** that uses `Xilinx Vivado` as the **Synthesis tool**

2. On the **Set Target Reference Design** task, you can specify the HDL Coder default reference designs, or a custom reference design that you want to integrate the HDL IP core into. If you do not specify unique names for test point signals, running this task can fail. To fix this error, in the **Result** subpane, select the link to generate unique names for test point signals. To verify that the task passes, rerun the task.
3. On the **Set Target Interface** task, you can enable HDL DUT port generation for testpoints by selecting the **Enable HDL DUT port generation for testpoints** checkbox and see the test point ports in the **Target platform interface table**. You can map the ports to AXI4, AXI4-Lite, or External Port interfaces. After you run this task, the code generator stores this testpoint interface mapping information on the DUT. To see this information, in the HDL Block Properties for the DUT Subsystem, on the **Target Specification** tab, look for the **TestPointMapping** block property. You can reload this information for the DUT across subsequent runs of the workflow.
4. On the **Generate RTL Code and IP Core task**, right-click and select **Run to Selected Task** to generate the IP core. The code generator opens an IP Core Generation report that displays the mapping of test point ports to interfaces.

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
cpt_e	Inport	ufix1	AXI4-Lite	x"100"
pv_e	Inport	boolean	AXI4-Lite	x"104"
mvt	Inport	boolean	AXI4-Lite	x"108"
sync_e	Inport	uint8	AXI4-Lite	x"110"
sat	Inport	uint8	DIP Switches [0:7]	[0:7]
Enable_In	Inport	boolean	AXI4-Lite	x"11C"
c_im	Outport	ufix17_En16	External Port	
c_imp	Outport	ufix17_En16	External Port	
cpt_s	Outport	uint8	AXI4-Lite	x"10C"
TestPoint_3	Test point	boolean	AXI4-Lite	x"114"
TestPoint	Test point	ufix17_En16	External Port	
TestPoint_1	Test point	boolean	AXI4-Lite	x"118"
TestPoint_2	Test point	uint8	LEDs General Purpose [0:7]	[0:7]

When you click the links in the test point ports, the code generator highlights the corresponding signals that you designated as test points in your Simulink model.

If you open the generated HDL source file, you see the test point signals connected to the IP core wrapper.

```
MaJCounte_ip_axi_lite u_MaJCounte_ip_axi_lite_inst (...
                                                    ...
                                                    .read_TestPoint_3(tp_TestPoint_3_sig),  // ufix1
                                                    .read_TestPoint_1(tp_TestPoint_1_sig),  // ufix1
                                                    ...
                                                    ...
                                                    );

MaJCounte_ip_dut u_MaJCounte_ip_dut_inst (...
                                          ...
                                          .tp_TestPoint(tp_TestPoint_sig),  // ufix17_En16
                                          .tp_TestPoint_1(tp_TestPoint_1_sig),  // ufix1
                                          .tp_TestPoint_2(tp_TestPoint_2_sig),  // ufix8
                                          .tp_TestPoint_3(tp_TestPoint_3_sig)  // ufix1
                                          );
```

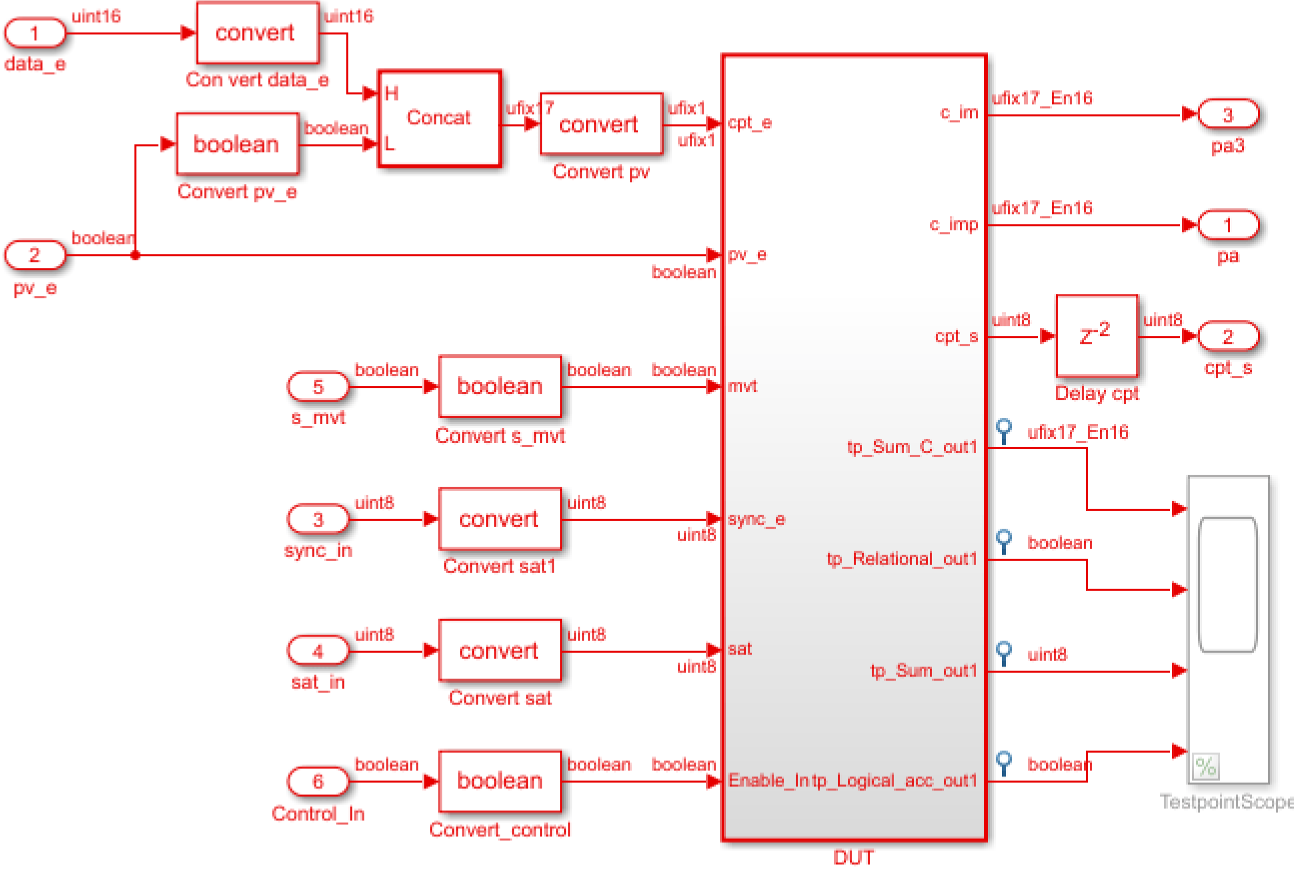
Run the workflow to generate the Software Interface model and integrate the IP core into the target reference design that you specified in the **Set Target Reference Design** task.

To learn more about the `IP Core Generation` workflow, see “Custom IP Core Generation” on page 39-17 and “Custom IP Core Report” on page 39-20.

Debug Test Point Signals

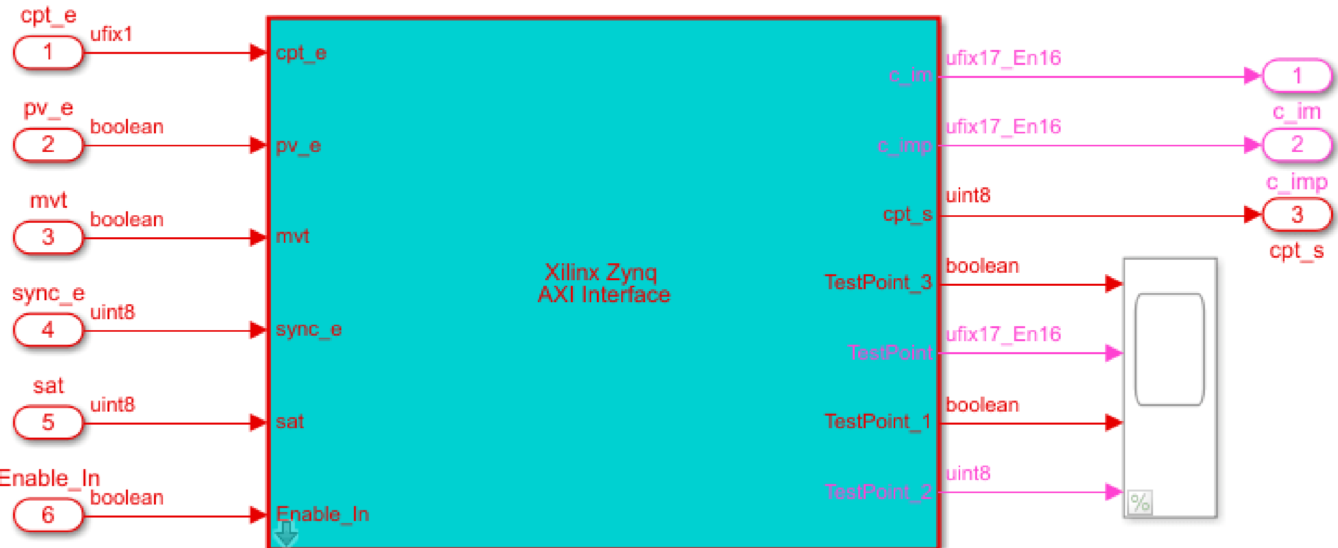
After you generate HDL code or generate an IP core, you can debug the test point signals.

If you generated HDL code for your model or ran the `Generic ASIC/FPGA` workflow, to debug the test point signals, generate a HDL test bench, or use the generated model. To open the generated model, at the command line, enter `gm_hdlcoder_test_points`.



In the generated model, you see the test points at the DUT output ports connected to a Scope block that is commented out. To observe the simulation results for these signals, uncomment the Scope block, and then run the simulation. If you navigate the generated model, you can see that the code generator creates an output port at the point where you designated the signal as a test point. HDL Coder then propagates these ports to the DUT as additional output ports.

If you run the `IP Core Generation` workflow to the **Generate Software Interface Model** task, the code generator opens the Software Interface model.



To observe the data on test point signals from the ARM® processor, uncomment the Scope block, and then run the Software Interface model.

Considerations

- Test point ports are considered similar to other output ports in the code generation process. Test point port generation works with all optimizations such as resource sharing, streaming, and distributed pipelining. To learn about various optimizations, see “Speed and Area Optimization”.
- If you generate a validation model, you see that the code generator does not compare the test point signals with the test point ports at the output. You can still observe the test point signals by uncommenting the Scope block and by running the simulation. To learn more about generated model and validation model, see “Generated Model and Validation Model” on page 21-10.
- If you generate a cosimulation model, you see the test point ports connected to a Terminator block. To observe the test points, remove the Terminator blocks, and connect the output ports to a Scope block, and then run cosimulation. You can also observe the waveforms in the HDL simulator that you run cosimulation with. To learn more about cosimulation, see “Generate a Cosimulation Model” on page 25-43.
- If you open the generated model, you see the Scope block commented out for performance considerations.
- You cannot specify the port ordering for the DUT test point ports. HDL Coder determines the port ordering when you generate code.
- **Target workflow** must be `Generic ASIC/FPGA`, `IP Core Generation`, or `Simulink Real-Time FPGA I/O`.
- If you use `IP Core Generation` or `Simulink Real-Time FPGA I/O` workflows, the **Synthesis tool** must be `Xilinx Vivado` or `Altera Quartus II`. `Xilinx ISE` is not supported.
- If you use `IP Core Generation` or `Simulink Real-Time FPGA I/O` workflows, you map the test point ports to AXI4, AXI4-Lite, or External Port interfaces. You cannot map the ports to AXI4-Stream or AXI4-Stream Video interfaces.

- When you use test point or tunable parameters in the IP core workflow, the DUT must be a subsystem level DUT and not the top-level model or model reference DUT.

See Also

More About

- “Configure Signals as Test Points”
- Enable HDL DUT output port generation for test points
- “Generated Model and Validation Model” on page 21-10

Optimize Generated HDL Code for Multirate Designs with Large Rate Differentials

In this section...

"Issue" on page 14-80
"Description" on page 14-80
"Recommendations" on page 14-82

Issue

When generating HDL code from your multirate algorithm in Simulink, HDL Coder might generate a large number of pipeline registers that can prevent the HDL design from fitting into an FPGA. This issue occurs due to modeling patterns that might result in large rate differentials. You can address this issue by using modeling techniques to manage sample time ratios.

Description

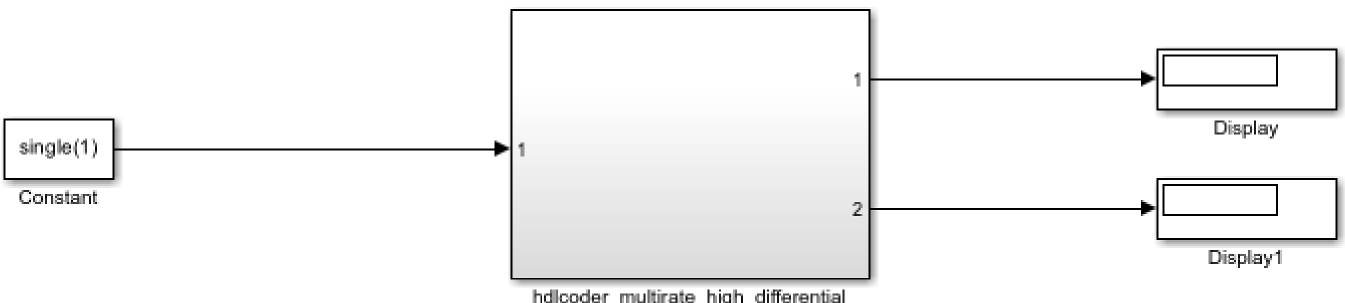
This issue occurs when your Simulink® model has a significantly large difference in sample rates or uses certain block implementations or optimizations that result in different clock-rate paths, such as:

- Multicycle block implementations
- Input and output pipelining
- Distributed pipelining
- Floating-point library mapping
- Native floating-point HDL code generation
- Fixed-point math functions such as reciprocal, sqrt, or divide
- Resource sharing
- Streaming

The additional pipelines result in a latency overhead that requires the insertion of matching delays across multiple signal paths operating at different rates. If the ratio of the fastest to the slowest clock rate is quite large, the code generator can potentially introduce a large number of registers in the resulting HDL code. The large number of pipeline registers can increase the size of the generated HDL files, and can prevent the design from fitting into an FPGA.

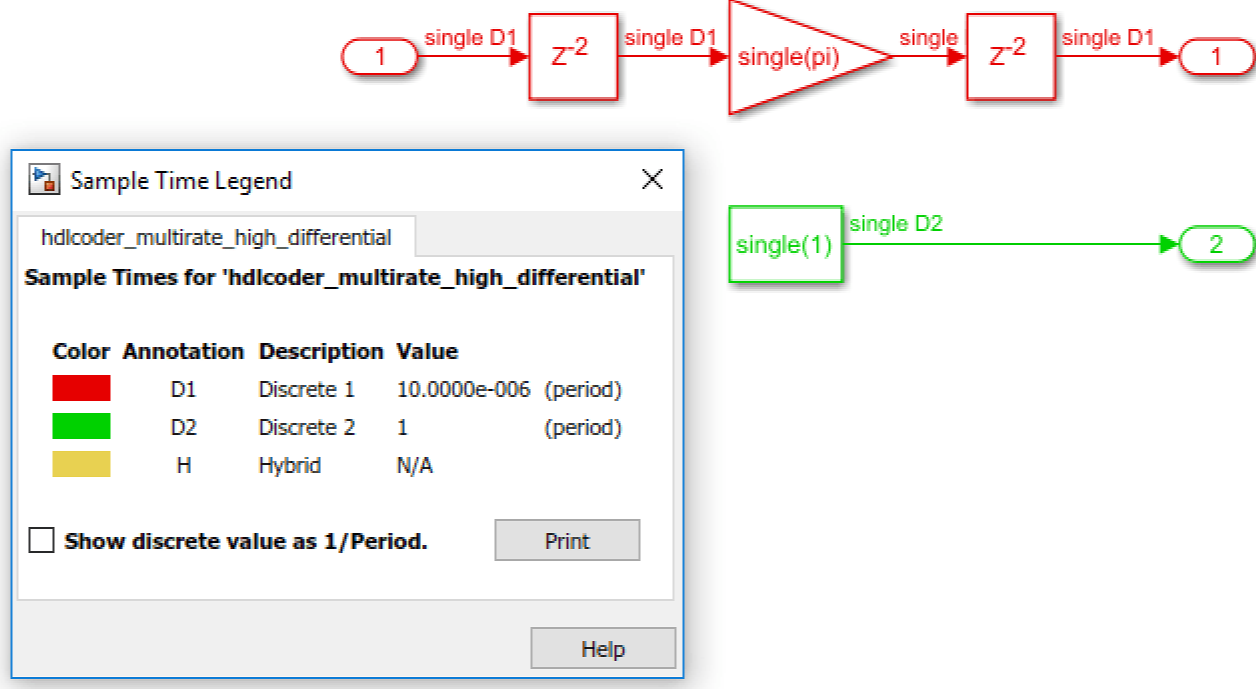
To see an example of how this issue occurs, open this Simulink model.

```
open_system('hdlcoder_multirate_high_differential')
```



Copyright 2017-2021 The MathWorks, Inc.

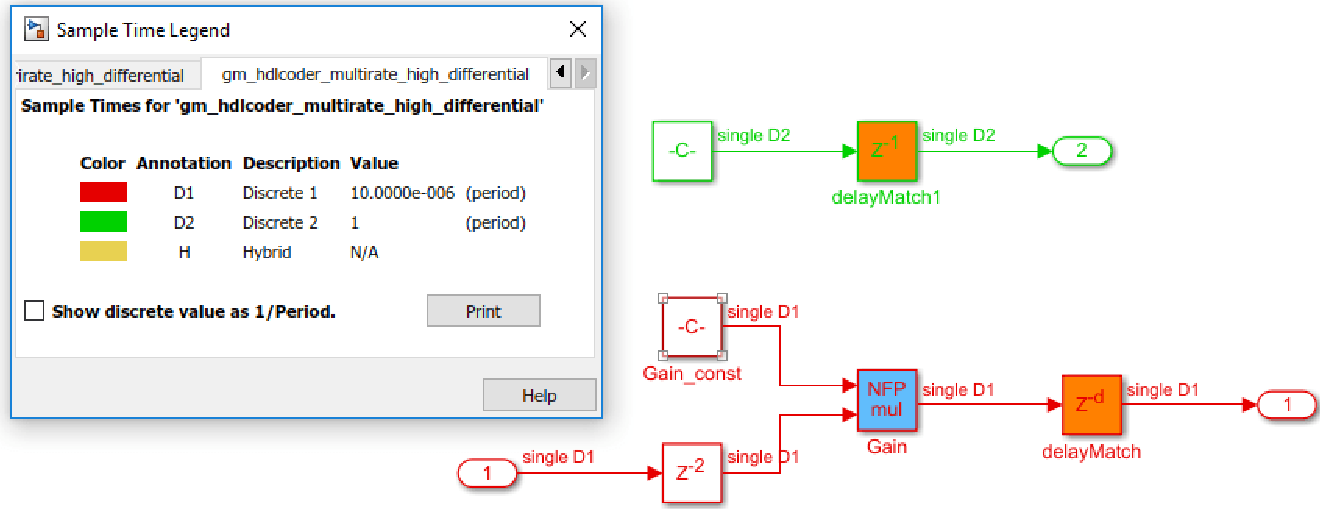
When you compile the model and double-click the `hdlcoder_multirate_high_differential` Subsystem, you can see that the model has a floating-point Gain block, a multicycle operator, in the fast clock-rate region.



Generate HDL code for the `hdlcoder_multirate_high_differential` Subsystem and check the output log.

```
### Generating HDL for 'hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential'.
### Using the config set for model hdlcoder_multirate_high_differential for HDL code generation parameters.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
### Output port 0: 100000 cycles.
### Output port 1: 1 cycles.
### Begin VHDL Code Generation for 'hdlcoder_multirate_high_differential'.
### Working on hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential/nfp_mul_comp as hdlsrc\hdlcoder_multirate_high_differential\nfp_mul_comp.vhd.
### Working on hdlcoder_multirate_high_differential_tc as hdlsrc\hdlcoder_multirate_high_differential\hdlcoder_multirate_high_differential_tc.vhd.
### Working on hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential as hdlsrc\hdlcoder_multirate_high_differential\hdlcoder_multirate_high_differential.vhd.
### Generating package file hdlsrc\hdlcoder_multirate_high_differential\hdlcoder_multirate_high_differential_pkg.vhd.
### Creating HDL Code Generation Check Report hdlcoder_multirate_high_differential_report.html
### HDL check for 'hdlcoder_multirate_high_differential' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

Open the generated model. At the command line, enter `gm_hdlcoder_multirate_high_differential`. When you compile the model and double-click the `hdlcoder_multirate_high_differential` Subsystem, the model looks as displayed by the sample time legend.



The large output latency on the fast clock rate region of the design is introduced by the code generator to balance delays across multiple output paths of the system. This large latency increases the size of the generated HDL files and reduces the efficiency of the generated code.

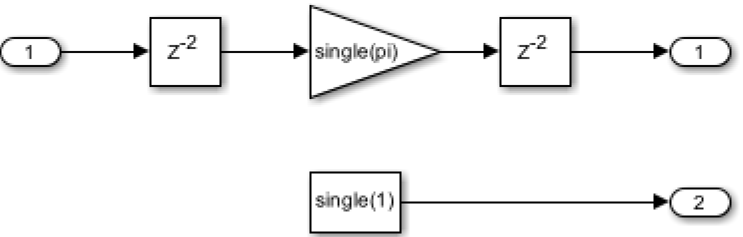
Recommendations

Recommendation 1: Use a Single-Rate Model

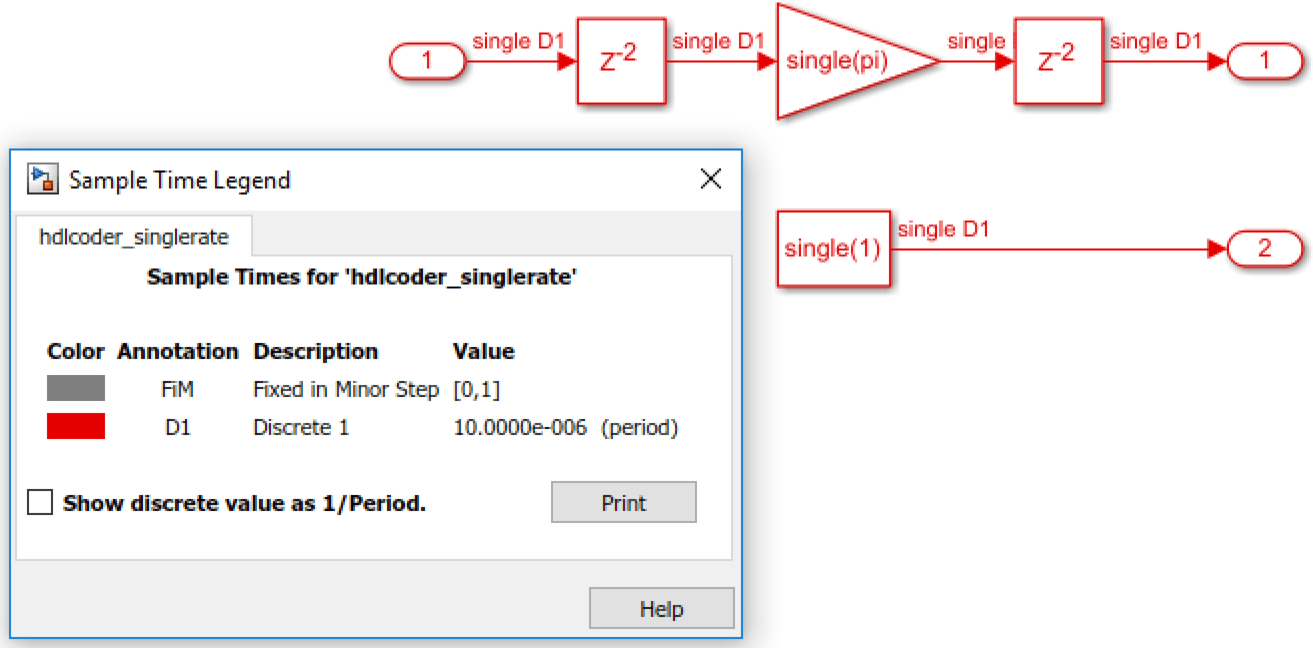
Most applications that you target the HDL code for might not require such a large rate differential. In that case, it is recommended that you use a single-rate model. In this example, you can change the sample rate of the Constant block inside the `hdlcoder_multirate_high_differential` subsystem to be the same as that of the base model.

Open this model that has the sample time of the Constant block changed to `10E-06`, which is the same sample time as the base sample time of the model.

```
open_system('hdlcoder_singlerate')
```



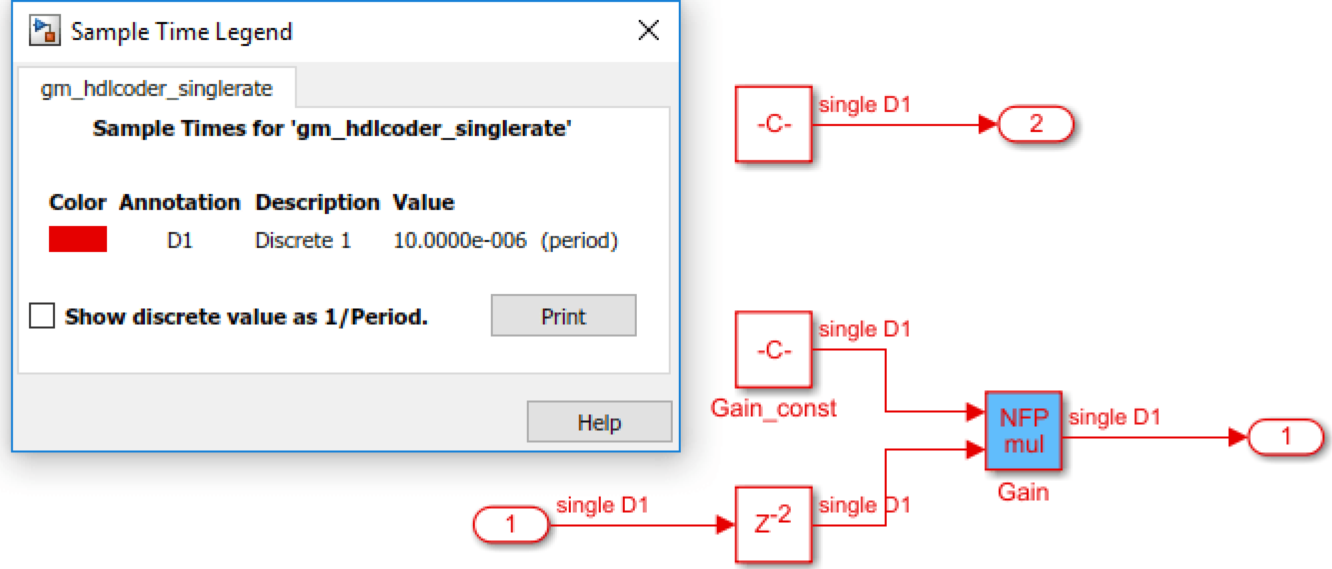
When you compile the model and double-click the `hdlcoder_singlerate` Subsystem, you see that the signal paths in the model operate at the same sample time of `10E-06`.



Set `UseFloatingPoint` property to `on` for the `hdlcoder_singlerate` model by using the `hdlset_param` function. Generate HDL code for the `hdlcoder_singlerate` subsystem and check the output log.

```
### Generating HDL for 'hdlcoder_singlerate/hdlcoder_singlerate'.
### Using the config set for model hdlcoder_singlerate for HDL code generation parameters.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
### Output port 0: 6 cycles.
### Output port 1: 6 cycles.
### Begin VHDL Code Generation for 'hdlcoder_singlerate'.
### Working on hdlcoder_singlerate/hdlcoder_singlerate/nfp_mul_comp as hdlsrc\hdlcoder_singlerate\nfp_mul_comp.vhd.
### Working on hdlcoder_singlerate/hdlcoder_singlerate as hdlsrc\hdlcoder_singlerate\hdlcoder_singlerate.vhd.
### Generating package file hdlsrc\hdlcoder_singlerate\hdlcoder_singlerate_pkg.vhd.
### Creating HDL Code Generation Check Report hdlcoder_singlerate_report.html
### HDL check for 'hdlcoder_singlerate' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

You see that the output latency has decreased significantly. Now open the generated model. At the MATLAB® command line, enter `gm_hdlcoder_singlerate`. When you compile the model and double-click the `hdlcoder_singlerate` Subsystem, the model looks as displayed by the sample time legend.



The generated HDL code is now optimal and uses few registers. Therefore, you can deploy the design to target FPGA platforms.

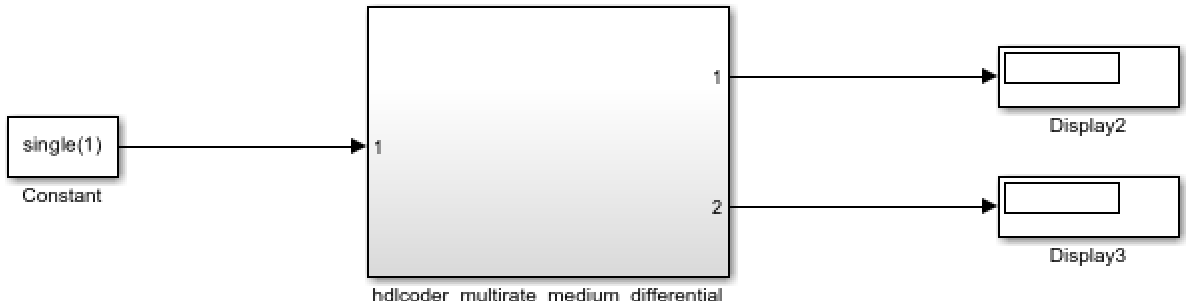
Recommendation 2: Reduce the Rate Differential

If you want to use a multirate model, it is recommended that you reduce the rate differential. Rate differential corresponds to the ratio of the fastest to the slowest clock rate in your design. If your target application requires two signal paths such that one signal path runs in time units of nanoseconds (ns) and the other signal path runs in time units of microseconds (us), you can choose to retain the multirate paths in your model. Be aware that delay balancing can introduce a significantly large number of registers to balance the signal paths.

In this example, you can change the sample rate of the Constant block inside the hdlcoder_multirate_high_differential Subsystem to reduce the rate differential.

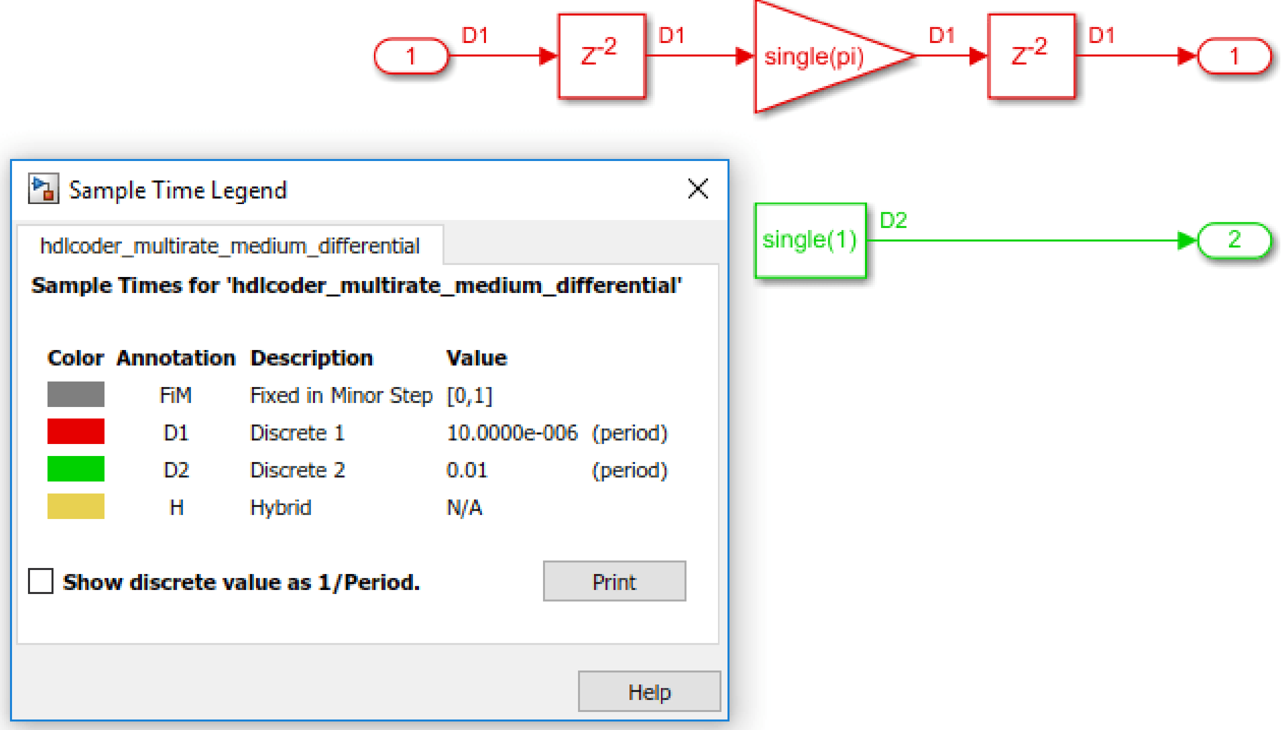
Open this model that has the sample time of the Constant block changed to 0.01.

```
open_system ('hdlcoder_multirate_medium_differential')
```



Copyright 2017-2021 The MathWorks, Inc.

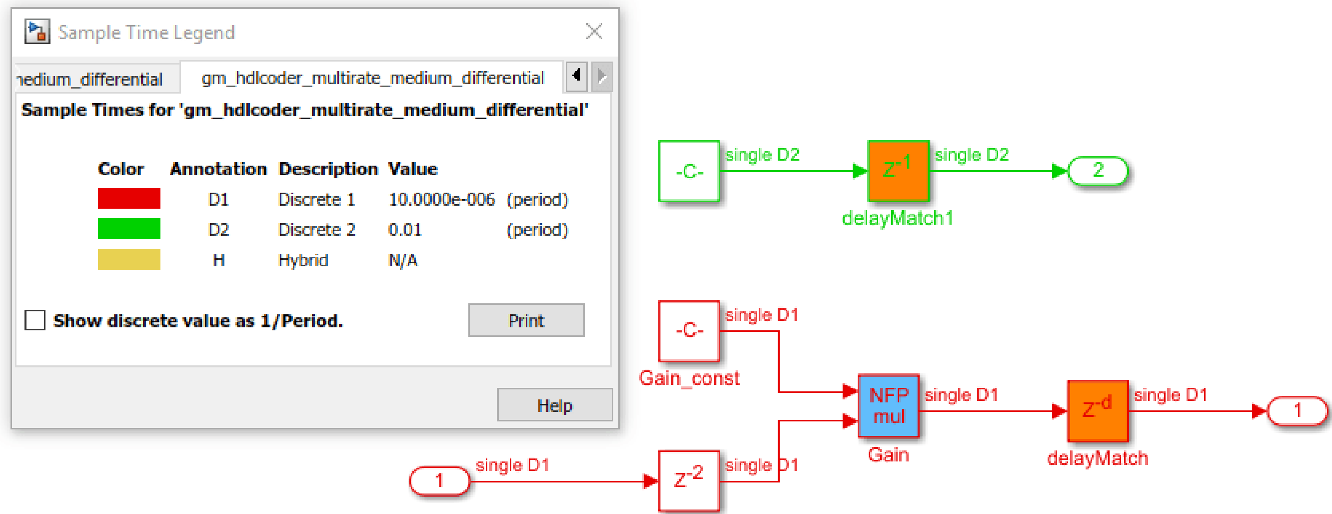
When you compile the model and double-click the `hdlcoder__multirate_medium_differential` Subsystem, you see that the rate differential between the two signal paths is equal to 1000.



Generate HDL code for the `hdlcoder_multirate_medium_differential` Subsystem and check the output log.

```
### Generating HDL for 'hdlcoder_multirate_medium_differential/hdlcoder_multirate_medium_differential'.
### Using the config set for model hdlcoder_multirate_medium_differential for HDL code generation parameters.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
### Output port 0: 1000 cycles.
### Output port 1: 1 cycles.
### Begin VHDL Code Generation for 'hdlcoder_multirate_medium_differential'.
### Working on hdlcoder_multirate_medium_differential/hdlcoder_multirate_medium_differential/nfp_mul_comp as hdlsrc\hdlcoder_multirate_medium_differential\nfp_mul_comp.vhd.
### Working on hdlcoder_multirate_medium_differential_tc as hdlsrc\hdlcoder_multirate_medium_differential\hdlcoder_multirate_medium_differential_tc.vhd.
### Working on hdlcoder_multirate_medium_differential/hdlcoder_multirate_medium_differential as hdlsrc\hdlcoder_multirate_medium_differential\hdlcoder_multirate_medium_differential.vhd.
### Generating package file hdlsrc\hdlcoder_multirate_medium_differential\hdlcoder_multirate_medium_differential_pkg.vhd.
### Creating HDL Code Generation Check Report hdlcoder_multirate_medium_differential_report.html
### HDL check for 'hdlcoder_multirate_medium_differential' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

Open the generated model. At the MATLAB® command line, enter `gm_hdlcoder_multirate_medium_differential`. When you compile the generated model and double-click the `hdlcoder_multirate_medium_differential` Subsystem, the model is as displayed by the sample time legend.



The model has a large number of registers, approximately 1000, in the fast clock rate path. The additional cost of registers is expected when you have a control logic that runs at a sample rate that is 1000 times faster than the sample rate of the system. When you deploy the generated code to a target platform, be aware of the constraints in hardware resources on the target platform. This recommendation offers a trade-off between generating optimal HDL code and targeting practical FPGA applications that might require an extremely large rate differential.

Recommendation 3: Map Pipeline Delays to RAM

To optimize the number of registers that your design uses on the target FPGA device, you can use the **Map Pipeline Delays to RAM** setting. This setting is a trade-off of the pipeline registers that are inserted in the HDL code with RAM resources to save area footprint on the target FPGA device. You can enable this setting in the **HDL Code Generation** > **Optimizations** > **General** tab of the Configuration Parameters dialog box.

You can also specify this setting at the command line by using the `MapPipelineDelaysToRAM` property with `hdlset_param` or `makehdl`. You can view the property value by using `hdlget_param`. Use either of these methods.

- Pass the property as an argument to the makehdl function.

```
makehdl('hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential', ...
                              'MapPipelineDelaysToRAM','on')
```

- When you use hdlset_param, you can set the parameter on the model, and then generate HDL code by using makehdl.

```
hdlset_param('hdlcoder_multirate_high_differential', ...
           'MapPipelineDelaysToRAM','on')
makehdl('hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential')
```

Use this setting in combination with the previous recommendations to further improve the efficiency of the generated HDL code and for deploying the code to the target platform.

See Also

createFloatingPointTargetConfig

Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 14-118

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 14-88
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111

Getting Started with HDL Coder Native Floating-Point Support

In this section...

"Key Features" on page 14-88
"Numeric Considerations and IEEE-754 Standard Compliance" on page 14-88
"Floating Point Types" on page 14-89
"Data Type Considerations" on page 14-90

Native floating-point support in HDL Coder enables you to generate code from your floating-point design. If your design has complex math and trigonometric operations or has data with a large dynamic range, use native floating point. You can use native floating point with a Simulink model or a MATLAB function.

Key Features

In your Simulink model or MATLAB function:

- You can have half-precision, single-precision, and double-precision floating-point data types and operations.
- You can have a combination of integer, fixed-point, and floating-point operations. By using Data Type Conversion blocks, you can perform conversions between floating-point and fixed-point data types.

The generated code:

- Complies with the IEEE-754 standard of floating-point arithmetic.
- Is target-independent. You can deploy the code on any generic FPGA or an ASIC.
- Does not require floating-point processing units or hard floating-point DSP blocks on the target ASIC or FPGA.

HDL Coder supports:

- Math and trigonometric functions
- Large subset of Simulink blocks
- Denormal numbers
- Customizing the latency of the floating-point operator

Numeric Considerations and IEEE-754 Standard Compliance

Native floating point technology in HDL Coder adheres to IEEE standard of floating-point arithmetic. For basic arithmetic operations such as addition, subtraction, multiplication, division, and reciprocal, when you generate HDL code in native floating-point mode, the numeric results obtained match the original Simulink model or MATLAB function.

Certain advanced math operations such as exponential, logarithm, and trigonometric operators have machine-specific implementation behaviors because these operators use recurring Taylor series and Remez expression based implementations. When you use these operators in native floating-point mode, the generated HDL code can have relatively small numeric differences from the Simulink

model or MATLAB function. These numeric differences are within a tolerance range and therefore indicate compliance with the IEEE-754 standard.

To generate code that complies with the IEEE-754 standard, HDL Coder supports:

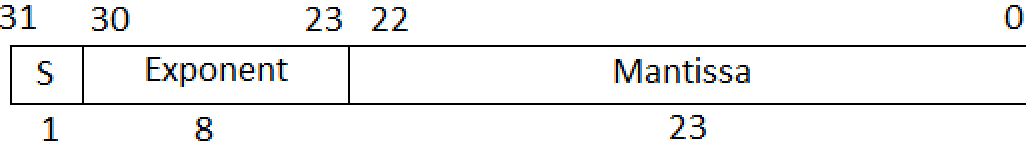
- Round to nearest rounding mode
- Denormal numbers
- Exceptions such as NaN (Not a Number), Inf, and Zero
- Customization of ULP (Units in the Last Place) and relative accuracy

For more information, see “Numeric Considerations for Native Floating-Point” on page 14-92.

Floating Point Types

Single Precision

In the IEEE 754–2008 standard, the single-precision floating-point number is 32-bits. The 32-bit number encodes a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa.



This graph is the normalized representation for floating-point numbers. You can compute the actual value of a normal number as:

$$value = (-1)^{sign} * (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{(e-127)}$$

The exponent field represents the exponent plus a bias of 127. The size of the mantissa is 24 bits. The leading bit is a 1, so the representation encodes the lower 23 bits.

Use single-precision types for applications that require larger dynamic range than half-precision types. Single-precision operations consume less memory and has lower latency than double-precision types.

Double Precision

In the IEEE 754–2008 standard, the single-precision floating-point number is 64-bits. The 64-bit number encodes a 1-bit sign, an 11-bit exponent, and a 52-bit mantissa.

63 62 52 51 0

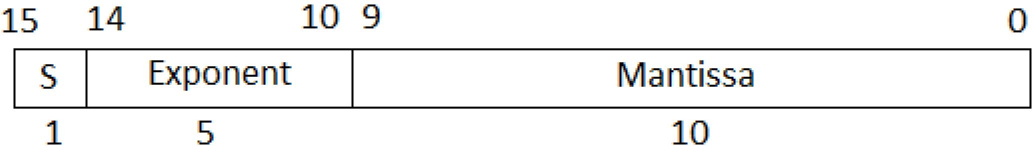
S	Exponent	Mantissa
1	11	52

The exponent field represents the exponent plus a bias of 1023. The size of the mantissa is 53 bits. The leading bit is a 1, so the representation encodes the lower 52 bits.

Use double-precision types for applications that require larger dynamic range, accuracy, and precision. These operations consume larger area on the FPGA and lower target frequency.

Half Precision

In the IEEE 754–2008 standard, the half-precision floating-point number is 16-bits. The 16-bit number encodes a 1-bit sign, a 5-bit exponent, and a 10-bit mantissa.



The exponent field represents the exponent plus a bias of 15. The size of the mantissa is 11 bits. The leading bit is a 1, so the representation encodes the lower 10 bits.

Use half-precision types for applications that require smaller dynamic range, consumes much less memory, has lower latency, and saves FPGA resources.

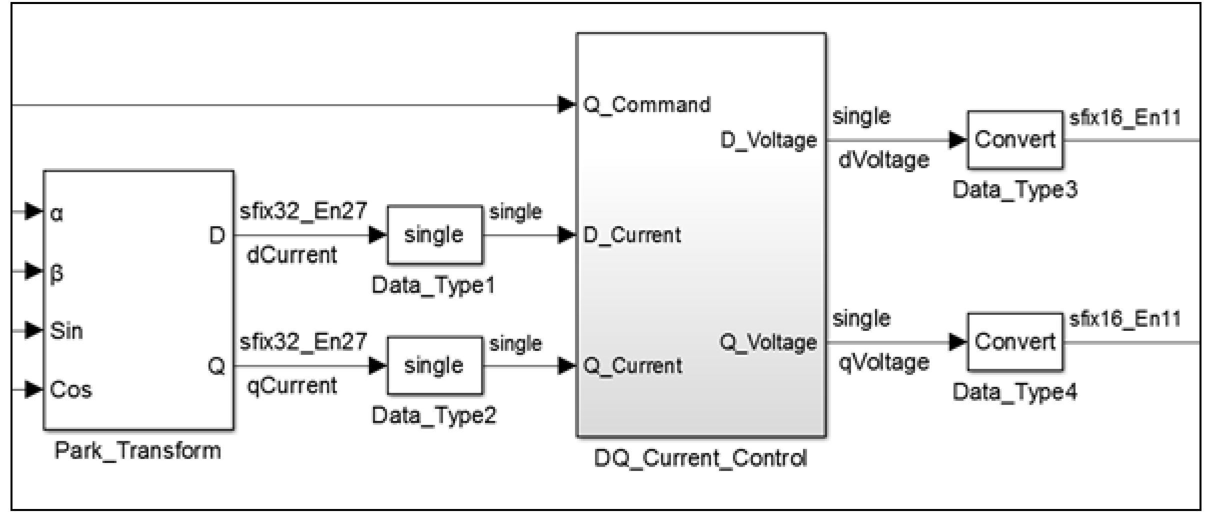
When using `half` types, you might want to explicitly set the **Output data type** of the blocks to `half` instead of the default setting `Inherit: Inherit via internal rule`. To learn how to change the parameters programmatically, see “Set HDL Block Parameters for Multiple Blocks Programmatically” on page 19-52.

Data Type Considerations

With native floating-point support, HDL Coder supports code generation from Simulink models or MATLAB functions that contain floating-point signals and fixed-point signals. You can model your design with floating-point types to:

- Implement algorithms that have a large or unknown dynamic range that can fall outside the range of representable fixed-point types.
- Implement complex math and trigonometric operations that are difficult to design in fixed point.
- Obtain a higher precision and better accuracy.

Floating-point designs can potentially occupy more area on the target hardware. In your Simulink model or MATLAB function, it is recommended to use floating-point data types in the algorithm data path and fixed-point data types in the algorithm control logic. This figure shows a section of a Simulink model that uses `single` and fixed-point types. By using Data Type Conversion blocks, you can perform conversions between the single and fixed-point types.



See Also

Modeling Guidelines
“Modeling with Native Floating Point” on page 18-65

Functions
`createFloatingPointTargetConfig`

Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 14-118

More About

- “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111
- “Simulink Blocks Supported by Using Native Floating Point” on page 14-137

Numeric Considerations for Native Floating-Point

In this section...

"Nearest Even Digit Rounding" on page 14-92
"Denormal Numbers" on page 14-92
"Exception Handling" on page 14-93
"Relative Accuracy and ULP Considerations" on page 14-93

Native floating-point technology can generate HDL code from your floating-point design. Floating-point designs have better precision, a higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex mathematical operations, use native floating-point technology.

HDL Coder generates code that complies with the IEEE® 754 standard for floating-point arithmetic. HDL Coder native floating-point supports:

- Rounding numbers to the nearest even digit
- Denormal numbers
- Exceptions such as `NaN` values, `Inf` values, and zero
- Customization of units in the last place (ULP) and relative accuracy

Nearest Even Digit Rounding

HDL Coder native floating-point supports rounding to the nearest even digit. This mode resolves all ties by rounding to the nearest even digit.

This rounding method requires at least three trailing bits after the 23 bits of the mantissa. The MSB is called Guard bit, the middle bit is called the Round bit, and the LSB is called the Sticky bit. The table shows the rounding action that HDL Coder performs based on different values of the three trailing bits. `x` denotes a *don't care* value and can take either a 0 or a 1.

Rounding bits	Rounding Action
`0xx`	No action performed.
`100`	A tie. If the mantissa bit that precedes the Guard bit is a 1, round up, otherwise no action is performed.
`101`	Round up.
`11x`	Round up.

Denormal Numbers

Denormal numbers are numbers that have an exponent field equal to zero and a nonzero mantissa field. The leading bit of the mantissa is zero.

$$value = (-1)^{sign} * (0 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{-126}$$

Denormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The presence of denormal numbers indicates a

loss of significant digits that can accumulate over multiple operations and result in unexpected values.

The logic that HDL Coder uses to handle denormal numbers involves counting the number of leading zeros and performing a left shift operation to obtain the normalized representation. Addition of this logic increases the area footprint on the target device and can affect the timing of your design.

When you use native floating-point support, you can specify your design handles denormal numbers.

Exception Handling

If you perform operations such as division by zero or computing the logarithm of a negative number, HDL Coder detects and reports exceptions. This table summarizes the mapping from the encoding of a floating-point number to the value of the number for different exceptions. An x denotes a don't care value, which can be a 0 or 1 without affecting the mapping.

Sign	Exponent	Significand	Value	Description
x	0xFF	0x00000000	$value = (-1)^S \infty$	Infinity
x	0xFF	A nonzero value	$value = NaN$	Not a Number
x	0x00	0x00000000	$value = 0$	Zero
x	0x00	A nonzero value	$value = (-1)^{sign} * (0 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{-126}$	Denormal
x	0x00 < E < 0xFF	x	$value = (-1)^{sign} * (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{(e-127)}$	Normal

Relative Accuracy and ULP Considerations

The representation of infinite real numbers with a finite number of bits requires an approximation. This approximation can result in rounding errors in floating-point computation. To measure the rounding errors, the floating-point standard uses a relative error and a ULP error.

ULP

If the exponent range is not upper-bounded, a ULP value of a floating-point number x is the distance between the two closest straddling floating-point numbers a and b nearest to x. The IEEE 754 standard requires that the result of an elementary arithmetic operation such as addition, multiplication, or division is correctly rounded. A correctly rounded result means that the rounded result is within 0.5 ULP of the exact result.

ULP value of 1 means adding a 1 to the decimal value of the stored integer. This table shows the approximation of π to nine decimal digits and how the ULP value of 1 changes the approximate value.

Floating-point number	Decimal Value of Stored Integer	IEEE-754 representation for Single Types
3.141592741	1078530011	0|10000000|10010010000111111011011
3.141592979	1078530012	0|10000000|10010010000111111011100

The gap between two consecutively representable floating-point numbers varies according to magnitude.

Floating-point number	Decimal Value of Stored Integer	IEEE-754 representation for Single Types
1234567	1234613304	0|10010011|00101101011010000111000
1234567.125	1234613305	0|10010011|00101101011010000111001

Relative Error

Relative error measures the relative difference between a floating-point number and the approximation of the real number. The relative error between the real numbers *a* and *b* is the ratio of absolute difference between numbers *a* and *b* to the maximum of *a* and *b*.

This table shows the relative error between two consecutive floating-point values that has a ULP value of 1.

Floating-point number	Decimal Value of Stored Integer	IEEE-754 representation for Single Types	Relative error
1234567	1234613304	0|10010011| 00101101011010000111000	1.0125e-07
1234567.125	1234613305	0|10010011| 00101101011010000111001	

See Also

Modeling Guidelines
“Modeling with Native Floating Point” on page 18-65

Functions
`createFloatingPointTargetConfig`

Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 14-118

More About

- “ULP Considerations of Native Floating-Point Operators” on page 14-95
- “Getting Started with HDL Coder Native Floating-Point Support” on page 14-88
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111

ULP Considerations of Native Floating-Point Operators

In this section...

"Adherence of Native Floating Point Operators to IEEE-754 Standard" on page 14-95
"ULP Values of Floating Point Operators" on page 14-95
"Considerations" on page 14-97

The representation of infinitely real numbers with a finite number of bits requires an approximation. This approximation can result in rounding errors in floating-point computation. To measure the rounding errors, the floating-point standard uses relative error and ULP (Units in the Last Place) error. To learn about relative error, see "Relative Accuracy and ULP Considerations" on page 14-93.

If the exponent range is not upper-bounded, Units in Last Place (ULP) of a floating-point number x is the distance between two closest straddling floating-point numbers a and b nearest to x. The IEEE-754 standard requires that the result of an elementary arithmetic operation such as addition, multiplication, and division is correctly round. A correctly rounded result means that the rounded result is within 0.5 ULP of the exact result.

Adherence of Native Floating Point Operators to IEEE-754 Standard

Native floating point technology in HDL Coder follows IEEE standard of floating-point arithmetic. Basic arithmetic operations such as addition, subtraction, multiplication, division, and reciprocal are mandated by IEEE to have zero ULP error. When you perform these operations in native floating-point mode, the numerical results obtained from the generated HDL code match the original Simulink model.

Certain advanced math operations such as exponential, logarithm, and trigonometric operators have machine-specific implementation behaviors because these operators use recurring taylor series and remez expression based implementations. When you use these operators in native floating-point mode, there can be relatively small differences in numerical results between the Simulink model and the generated HDL code.

You can measure the difference in numerical results as a relative error or ULP. A nonzero ULP for these operators does not mean noncompliance with the IEEE standard. A ULP of one is equivalent to a relative error of `10^-7`. You can ignore such relatively small errors by specifying a custom tolerance value for the ULP when generating a HDL test bench. For example, you can specify a custom floating-point tolerance of one ULP to ignore the error when verifying the generated code. For more information, see configuration parameters **Floating point tolerance check based on** and **Tolerance Value**.

ULP Values of Floating Point Operators

The table enumerates the ULP of floating-point operators that have a nonzero ULP. In addition to these operators, the HDL Reciprocal block has a ULP of five.

Math Functions

Simulink Blocks	Data Type	Units in the Last Place (ULP) error
exp	Single	1
	Half	1
log	Double	1
	Single	1
	Half	1
log10	Single	1
	Half	1
10^u	Single	1
pow	Single	1
hypot	Single	1

Math Operations

Simulink Blocks	Data Type	Units in the Last Place (ULP) error
HDL Reciprocal	Single	5
Reciprocal Sqrt	Double	1

Trigonometric Functions

Simulink Blocks	Data Type	Units in the Last Place (ULP) error
sin	Double	1
	Single	2
	Half	1
cos	Double	1
	Single	2
	Half	1
tan	Single	3
asin	Single	2
acos	Single	2
atan	Single	2
atan2	Single	5
sinh	Single	1
cosh	Single	1
tanh	Single	1
asinh	Single	2
acosh	Single	2
atanh	Single	3
sincos	Single	2

Considerations

For certain floating-point input values, some blocks can produce simulation results that vary from the MATLAB simulation results. To see the difference in results, before you generate code, enable generation of the validation model. In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select the **Generate validation model** check box.

- If you perform computations that involve complex numbers and an exception such as `Inf` or `NaN`, the HDL simulation result with native floating point can potentially vary from the Simulink simulation result. For example, if you multiply a complex input with `Inf`, the Simulink simulation result is `Infi` whereas the HDL simulation result is `NaN+Infi`.
- HDL Coder does not generate a mismatch error between reference and native floating-point values if both values are `NaN` irrespective of the sign.
- If you compute the square root or logarithm of a negative number, the HDL simulation result with native floating point is `0`. This result matches the simulation result when you verify the design with a SystemVerilog DPI test bench. In Simulink, the result obtained is `NaN`. According to the IEEE-754 standard, if you compute the square root or logarithm of a negative number, the result is that number itself.
- If the input to the Direct Lookup Table (n-D) is of floating-point data type, but the elements of the table use a smaller data type, the generated HDL code can be potentially incorrect. For example, the input is of `single` type and the elements use `uint8` type. To obtain accurate HDL simulation results, use the same data type for the input signal and the elements of the lookup table.

- If you use the Cosine block with the inputs `-7.729179E28` or `7.729179E28`, the generated HDL code has a ULP of 4. For all other inputs, the ULP is 2.
- When you use a Math Function block to compute `mod(a,b)` or `rem(a,b)`, where `a` is the dividend and `b` is the divisor, the simulation result in native floating-point point mode varies from the MATLAB simulation result in these cases:
  - If b is integer and $\frac{a}{b} > 2^{32}$, the simulation result in native floating-point mode is zero. For such significant difference in magnitude between the numbers `a` and `b`, this implementation saves area on the target FPGA device.
  - If $\frac{a}{b}$ is close to 2^{23}, the simulation result in native floating-point mode can potentially vary from the MATLAB simulation results.

See Also

Modeling Guidelines
"Modeling with Native Floating Point" on page 18-65

Functions
`createFloatingPointTargetConfig`

Related Examples

- "Floating Point Support: Field-Oriented Control Algorithm" on page 14-118

More About

- "Numeric Considerations for Native Floating-Point" on page 14-92
- "Getting Started with HDL Coder Native Floating-Point Support" on page 14-88
- "Generate Target-Independent HDL Code with Native Floating-Point" on page 14-111

Latency Values of Floating-Point Operators

In this section...

"Math Operations" on page 14-99
"Trigonometric Operations" on page 14-101
"Comparisons and Conversions" on page 14-102

HDL Coder native floating-point support can generate HDL code from your floating-point design. HDL Coder supports several Simulink blocks, Math and Trigonometric Functions blocks in native floating-point mode. The tables list the default latency values of these floating-point operations.

You can customize the latency settings for the blocks and design for trade-offs between latency and maximum frequency. For more information, see "NFPCustomLatency" on page 19-35.

You can also set the global custom latency of native floating-point IPs. Use the keywords listed in the tables to specify the global custom latency of the floating-point IPs. For more information on setting the custom latency of floating-point IPs, see "Latency Considerations with Native Floating Point" on page 14-104.

You can see the latency of these floating-point operators in MATLAB by entering these commands.

```
nfpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
nfpconfig.IPConfig
```

Math Operations

This table shows the list of basic math operations that are supported with native floating-point in HDL Coder and their latency information. The basic math operations include addition, subtraction, multiplication, and so on. You can use most of these blocks with both `single` and `double` data types. If you do not see an entry of `double` data type corresponding to a block, it means that the block does not support `double` types.

Basic Math Operators

Simulink Blocks	Keywords	Data Type	Minimum Output Latency	Maximum Output Latency	Custom Latency Support
Add	`ADDSUB`	Double	6	11	Yes
		Single	6	11	
		Half	4	8	
Subtract	`ADDSUB`	Double	6	11	Yes
		Single	6	11	
		Half	4	8	
Product or Gain	`MUL`	Double	6	9	Yes
		Single	6	8	
		Half	4	7	
Divide	`DIV`	Double	31	61	Yes
		Single	17	32	
		Half	10	19	
Reciprocal	`RECIP`	Double	30	60	Yes
		Single	16	31	
		Half	10	19	
Multiply-Add	`MULTADD`	Single	8	14	No
Sqrt	`SQRT`	Double	33	58	Yes
		Single	16	28	
		Half	6	12	
HDL Reciprocal	`HDLRECIP`	Single	14	21	No
Rounding Function	`ROUNDING`	Double	3	5	Yes
		Single	3	5	
Gain(Power of 2)	`GAINPOW2`	Double	1	2	Yes
		Single	1	2	
		Half	1	4	

This table shows the math functions that are supported with native floating-point in HDL Coder and their latency information. You can select the function using the **Function** setting of the Math Function block.

Math Functions

Simulink Blocks	Keywords	Data Type	Minimum Output Latency	Maximum Output Latency	Custom Latency Support
Rem	`REM`	Single	15	24	No
Mod	`MOD`	Single	16	26	No
Reciprocal Sqrt	`RSQRT`	Single	16	30	Yes
Hypot	`HYPOT`	Single	17	33	No
Magnitude Square	-	Double	6	9	Yes
		Single	6	8	

This table shows the exponential operations that are supported with native floating-point in HDL Coder and their latency information. You can select the function using the **Function** setting of the Math Function block. You can use these blocks with `single` data types. `Double` types are unsupported for the blocks except Log.

Exponent/Logarithm/Power

Simulink Blocks	Keywords	Data Type	Minimum Output Latency	Maximum Output Latency	Custom Latency Support
Exp	`EXP`	Single	16	26	No
Pow	`POW`	Single	33	54	No
Pow10	`POW10`	Single	16	26	No
Log	`LOG`	Double	34	44	No
		Single	20	27	
		Half	9	17	
Log10	`LOG10`	Single	17	27	No
		Half	10	18	

Trigonometric Operations

This table shows the trigonometric operations that are supported with native floating-point in HDL Coder and their latency information. You can select the function using the **Function** setting of the Trigonometric Function block. You can use these blocks with `single` data types.

Trigonometric Functions

Simulink Blocks	Keywords	Data Type	Minimum Output Latency	Maximum Output Latency	Custom Latency Support
Sin	SIN	Double	34	34	No
		Single	27	27	
		Half	8	14	
Cos	COS	Double	48	48	No
		Single	27	27	
		Half	9	14	
Tan	TAN	Single	33	33	No
Sincos	SINCOS	Single	27	27	No
Asin	ASIN	Single	17	23	No
Acos	ACOS	Single	17	23	No
Atan	ATAN	Single	36	36	No
Atan2	ATAN2	Single	42	42	No
Sinh	SINH	Single	18	30	No
Cosh	COSH	Single	17	27	No
Tanh	TANH	Single	25	43	No
Asinh	ASINH	Single	94	94	No
Acosh	ACOSH	Single	93	93	No
Atanh	ATANH	Single	67	67	No

Comparisons and Conversions

This table shows operations related to relational operators and data type conversions that are supported with native floating-point in HDL Coder and their latency information. You can use these blocks with both `single` and `double` data types except for the MinMax block. This block does not support `double` data type. For the Data Type Conversion block, you can convert between `double`, `half`, and `single` data types, and between floating-point and other fixed-point data types.

Comparisons and Conversions

Simulink Blocks	Keywords	Data Type	Minimum Output Latency	Maximum Output Latency	Custom Latency Support
Data Type Conversion	`CONVERT`	DOUBLE_TO_NUMERICTYPE	3	6	Yes
		DOUBLE_TO_SINGLE	3	6	
		HALF_TO_NUMERICTYPE	2	3	
		HALF_TO_SINGLE	1	2	
		NUMERICTYPE_TO_DOUBLE	3	6	
		NUMERICTYPE_TO_HALF	2	4	
		NUMERICTYPE_TO_SINGLE	6	6	
		SINGLE_TO_DOUBLE	3	5	
		SINGLE_TO_HALF	2	3	
		SINGLE_TO_NUMERICTYPE	6	6	
Relational Operator	`RELOP`	Double	1	3	Yes
		Single	1	3	
		Half	1	2	
MinMax	`MINMAX`	Single	3	3	No

See Also

Modeling Guidelines
"Modeling with Native Floating Point" on page 18-65

Functions
`createFloatingPointTargetConfig`

Related Examples

- "Floating Point Support: Field-Oriented Control Algorithm" on page 14-118

More About

- "Getting Started with HDL Coder Native Floating-Point Support" on page 14-88
- "Simulink Blocks Supported by Using Native Floating Point" on page 14-137

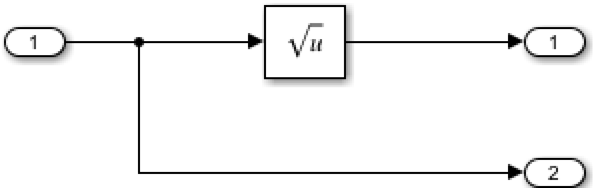
Latency Considerations with Native Floating Point

HDL Coder™ native floating-point technology can generate HDL code from your floating-point design. Native floating-point operators have a latency. When you generate HDL code, the code generator figures out this latency and adds matching delays to balance parallel paths.

View Latency of a Floating-Point Operator

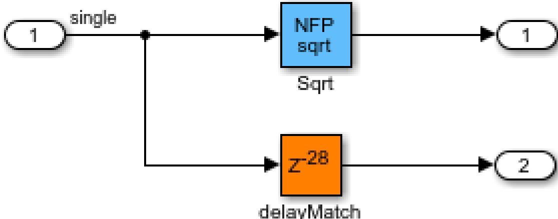
Open the `hdlcoder_nfp_delay_allocation` Simulink® model. The model uses `single` data types and computes the square root. The model has a parallel path to illustrate how the code generator balances delays.

```
load_system('hdlcoder_nfp_delay_allocation')
open_system('hdlcoder_nfp_delay_allocation/DUT')
```

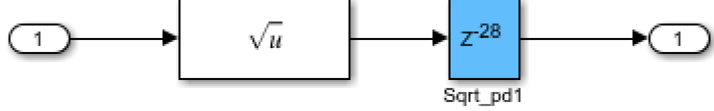


To generate HDL code:

1 Right-click the `DUT` Subsystem and select **HDL Code** > **Generate HDL for Subsystem**.
2 To see the generated model after HDL code generation, at the command line, enter `gm_hdlcoder_nfp_delay_allocation`.



The `NFP sqrt` block is the floating-point operator corresponding to the Sqrt block in your model, and has a latency of `28`. The code generator determines this latency and adds a matching delay of length `28` in the parallel path. To see the latency of the square root operation, double-click the `NFP Sqrt` block. The **Delay length** of the `Sqrt_pd1` block corresponds to the operator latency.



You can customize the latency of your design. Use custom latency settings to design for trade-offs between latency and throughput. You can then optimize your design implementation on the target FPGA device for area and speed. Customize the latency by using:

- Latency Strategy setting: Specify whether to map your entire Simulink model or individual blocks in your model to maximum, minimum, or zero latency of the floating-point operator.
- Custom Latency: You can specify a custom latency for certain blocks that you use in your Simulink model. The custom latency setting can take values from zero to the maximum latency of the floating-point operator.
- Oversampling factor: Increasing the **Oversampling factor** operates the design at a faster clock rate and absorbs the clock-rate pipelines with the latency of the floating-point operator.
- Delay blocks in the model: If your Simulink model has a latency, HDL Coder can absorb some or all of the latency with the native floating-point implementation.

Latency Strategy Setting for Model

You can specify the latency strategy setting for an entire model or for individual blocks in your model.

To specify this setting for a model:

1. In the `hdlcoder_nfp_delay_allocation` model, right-click the `DUT` Subsystem and select **HDL Code > HDL Coder Properties**.
2. On the **HDL Code Generation > Floating Point**, select **Use Floating Point**.
3. For **Latency Strategy**, select **MAX**, **MIN**, or **ZERO**.

To specify this setting from the command line:

1. Create a `hdlcoder.FloatingPointTargetConfig` object for native floating point by using the `hdlcoder.createFloatingPointTargetConfig` function.

```
nfpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint");
hdlset_param('hdlcoder_nfp_delay_allocation', 'FloatingPointTargetConfiguration', nfpconfig);
```

2. Specify the latency strategy by using the `LatencyStrategy property` of the `nfpconfig` object.

```
nfpconfig.LibrarySettings.LatencyStrategy = 'MAX'
```

```
nfpconfig = 

  FloatingPointTargetConfig with properties:

                Library: 'NATIVEFLOATINGPOINT'
        LibrarySettings: [1x1 fpconfig.NFPLatencyDrivenMode]
               IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
          VendorLibrary: []
  VendorLibrarySettings: []
         VendorIPConfig: []
```

To see the latency information, generate HDL code and then open the generated model. To open the generated model, enter the command `gm_hdlcoder_nfp_delay_allocation`.

Custom Latency Strategy for Blocks

For blocks in your Simulink model, you can selectively customize the latency strategy. By default, the blocks inherit the latency strategy setting you specify for the model. For certain blocks, you can specify a custom latency value that is between zero and the maximum latency of the floating-point operator.

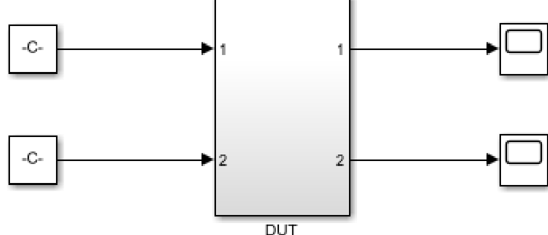
By specifying a custom latency, you can customize your design for trade-offs between:

- Clock frequency and power consumption: A higher latency value increases the maximum clock frequency (Fmax) that you can achieve, which increases the dynamic power consumption.
- Oversampling factor and sampling frequency: A combination of higher latency value and higher oversampling factor increases the Fmax that you can achieve but reduces the sampling frequency.

To learn more about this setting and how to specify the latency strategy for a block, see “LatencyStrategy” on page 19-33.

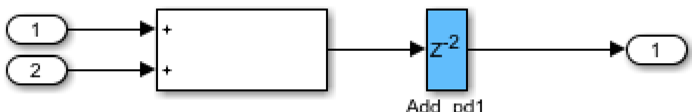
For example, if you have an Add block in the parallel path in your model, you can specify a custom latency value of 2 for the Add block by entering these commands.

```
load_system('hdlcoder_nfp_delay_allocation_custom')
open_system('hdlcoder_nfp_delay_allocation_custom')
hdlset_param('hdlcoder_nfp_delay_allocation_custom/DUT/Add','LatencyStrategy','Custom')
hdlset_param('hdlcoder_nfp_delay_allocation_custom/DUT/Add','NFPCustomLatency',2)
```



Copyright 2018-2021 The MathWorks, Inc.

To see the latency information, generate HDL code and then open the generated model. To open the generated model, enter the command `gm_hdlcoder_nfp_delay_allocation_custom`. In the generated model, you see that the `NFP Add` block has a latency of `2`.



Custom Latency Settings for Native Floating-Point IPs

For a model that has a large number of floating-point operators, you can customize the latency for native floating-point IPs by setting the global custom latency for NFP operators. The customization applies to all operators in the model.

For example, if you have a model that has multiple add and product blocks, by default, the block inherits the latency strategy settings specified for the model. By using these commands, you can customize the latency of all the `NFP add` blocks to 4 and all the `NFP mul` blocks to 3.

```
load_system('hdlcoder_nfp_delay_allocation_global_custom')
open_system('hdlcoder_nfp_delay_allocation_global_custom/Sample_DUT');
hdlset_param('hdlcoder_nfp_delay_allocation_global_custom', 'FloatingPointTargetConfiguration',
hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint', 'IPConfig', ...
{{ 'ADDSUB',  'SINGLE', 'CustomLatency', 4} ...
, { 'ADDSUB',  'DOUBLE', 'CustomLatency', 4} ...
, { 'MUL',  'SINGLE', 'CustomLatency', 3} ...
, { 'MUL',  'DOUBLE', 'CustomLatency', 3}}))
```

To see the latency information, generate HDL code and then open the generated model. To open the generated model, enter the command `gm_hdlcoder_nfp_delay_allocation_global_custom`. In the generated model, you see that the all the `NFP Add` block has a latency of 4 and all the `NFP mul` blocks has latency of 3.

For the list of keywords to use for native floating-point IPs in the API in the command, refer to the table in “Latency Values of Floating-Point Operators” on page 14-99.

Oversampling Factor

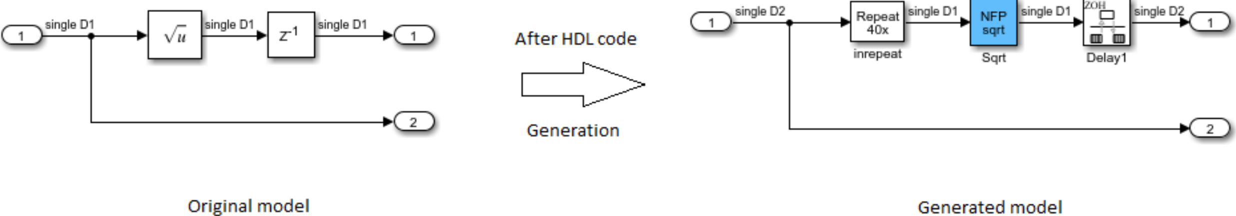
When you design the blocks in your Simulink model at the data rate, specify an **Oversampling factor** greater than one. The **Oversampling factor** inserts pipeline registers at a faster clock rate,

which improves clock frequency and reduces area usage. To learn more about clock-rate pipelining, see “Clock-Rate Pipelining” on page 21-148.

To see the effect of **Oversampling factor** on the model, in the `hdlcoder_nfp_delay_allocation` model:

1. Add a Delay block with a **Delay length** of `1` at the output of the `Sqrt` block.
2. Right-click the DUT and select **HDL Code > HDL Coder Properties**.
3. On the **HDL Code Generation > Global Settings** pane, enter a value of `40` for **Oversampling factor**.

After HDL code generation, the generated model shows the `NFP Sqrt` block operating at a clock rate that is 40 times faster than the `Sqrt` block in your model. The `NFP Sqrt` block absorbed the Delay block in your Simulink model. The Delay block now operates at the clock rate. This implementation saves area by absorbing the additional latency, and improves timing by operating at the faster clock rate.



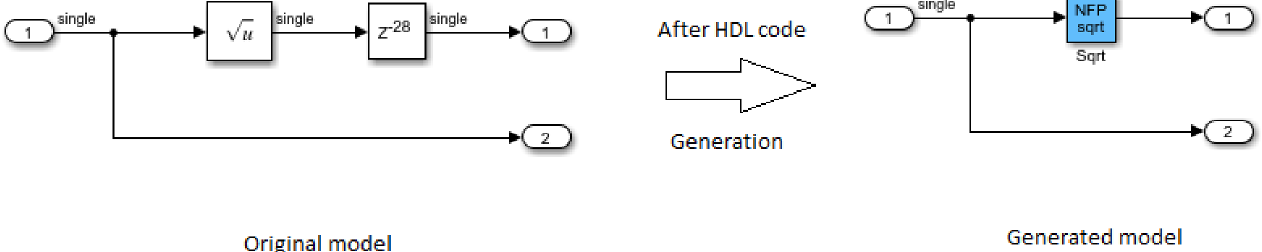
Delay Absorption in the Model

If your Simulink model has a Delay block with sufficient **Delay length** adjacent to an operator or separated from the operator by only a component that does not take zero input and output a non-zero value, such as a NOT Logical Operator block, HDL Coder absorbs the delays as part of the operator latency.

If the **Delay length** is equal to the latency of the floating-point operator, HDL Coder absorbs the delays and does not introduce any additional latency.

In the `hdlcoder_nfp_delay_allocation` model:

1. Double-click the Delay block at the output of the Sqrt block and change the **Delay length** to `28`.
2. Generate HDL code for the `DUT` Subsystem.
3. After HDL code generation, at the command line, enter `gm_hdlcoder_nfp_delay_allocation` to open the generated model.

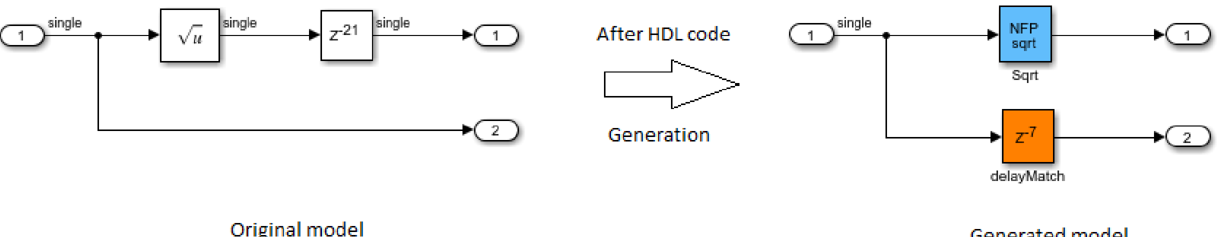


In the generated model, you see that the `NFP Sqrt` block absorbs the Delay block adjacent to the Sqrt block in your original model. This delay absorption occurs because the operator latency is equal to the **Delay length**. The code generator therefore avoids the additional latency in your model.

If the **Delay length** is less than the operator latency, HDL Coder absorbs the available delays and balances parallel paths by adding matching delays.

In the `hdlcoder_nfp_delay_allocation` model:

1. Double-click the Delay block at the output of the Sqrt block and change the **Delay length** to `21`.
2. Generate HDL code for the `DUT` Subsystem.
3. After HDL code generation, at the command line, enter `gm_hdlcoder_nfp_delay_allocation` to open the generated model.

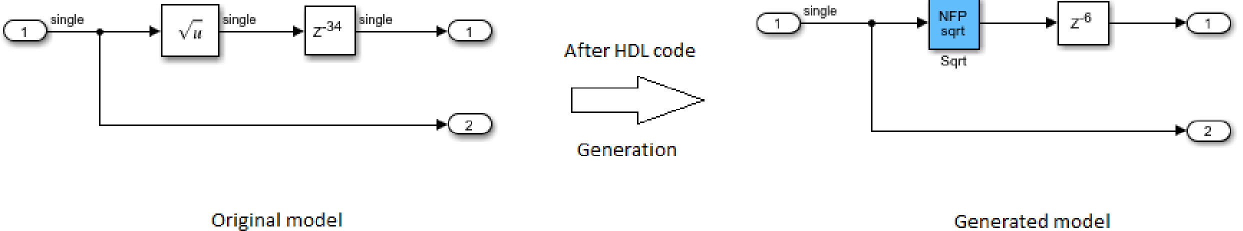


You see that the `NFP Sqrt` block absorbed a Delay of length `21` and added a matching delay of length `7` in the parallel path because the square root operation requires `28` delays.

If the delay length is greater than the operator latency, the code generator absorbs a certain number of delays equal to the latency and the excess delays appear outside the operator.

In the `hdlcoder_nfp_delay_allocation` model:

1. Double-click the Delay block at the output of the Sqrt block and change the **Delay length** to `34`.
2. Generate HDL code for the `DUT` Subsystem.
3. After HDL code generation, at the command-line, enter `gm_hdlcoder_nfp_delay_allocation` to open the generated model.



The NFP Sqrt block absorbed 28 delays because the square root operation has a latency of 28. The excess latency of 6 is outside the operator.

See Also

Modeling Guidelines
"Modeling with Native Floating Point" on page 18-65

Functions
`createFloatingPointTargetConfig`

Related Examples

- "Floating Point Support: Field-Oriented Control Algorithm" on page 14-118

More About

- "Getting Started with HDL Coder Native Floating-Point Support" on page 14-88
- "Latency Values of Floating-Point Operators" on page 14-99
- "Simulink Blocks Supported by Using Native Floating Point" on page 14-137

Generate Target-Independent HDL Code with Native Floating-Point

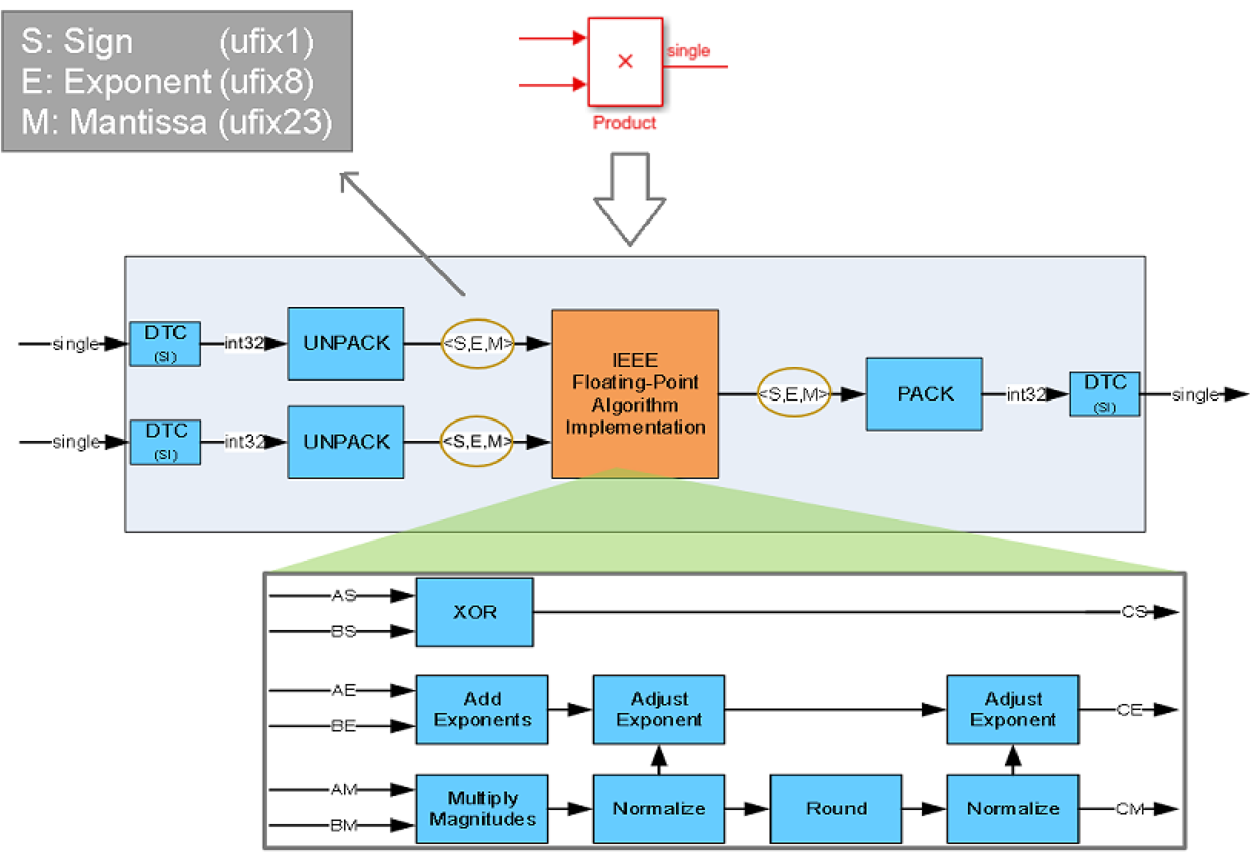
In this section...

“How HDL Coder Generates Target-Independent HDL Code” on page 14-111
“Enable Native Floating Point and Generate Code” on page 14-112
“View Code Generation Report” on page 14-113
“Analyze Results” on page 14-114
“Limitation” on page 14-116

HDL Coder native floating-point technology can generate target-independent HDL code from your floating-point design. You can synthesize your floating-point design on any generic FPGA or ASIC. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

How HDL Coder Generates Target-Independent HDL Code

This figure shows how HDL Coder generates code with the native floating-point technology.



The `Unpack` and `Pack` blocks convert the floating-point types to the sign, exponent, and mantissa. In the figure, *S*, *E*, and *M* represent the sign, exponent, and mantissa respectively. This interpretation is based on the IEEE-754 standard of floating-point arithmetic.

The **Floating-Point Algorithm Implementation** block performs computations on the *S*, *E*, and *M*. With this conversion, the generated HDL code is target-independent. You can deploy the design on any generic FPGA or an ASIC.

Enable Native Floating Point and Generate Code

You can enable native floating point and generate HDL code from a Simulink model or MATLAB function.

Specify Native Floating Point for a Simulink Model

You can specify the native floating-point settings for HDL code generation in the Configuration Parameters dialog box or at the command line.

To specify the native floating-point settings and generate HDL code in the Configuration Parameters dialog box:

1. In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
2. Click **Settings**. In the **HDL Code Generation** > **Floating Point** pane, select **Use Floating Point**.
3. Specify the **Latency Strategy** to map your design to maximum or minimum latency or no latency. See Latency Strategy.
4. If you have denormal numbers in your design, select **Handle Denormals**. Denormal numbers are numbers that have an exponent field equal to zero and a nonzero mantissa field. See Handle Denormals.
5. If your design has multipliers, to specify how you want HDL Coder to implement the multiplication operation, use the **Mantissa Multiplier Strategy**. See Mantissa Multiplier Strategy.
6. To share floating-point resources, on the **HDL Code Generation** > **Optimizations** > **Resource Sharing** tab, make sure that you select **Floating-point IPs**. The number of blocks that get shared depends on the **SharingFactor** that you specify for the subsystem.
7. Click **Apply**. In the **HDL Code** tab, click **Generate HDL Code**.

To apply native floating point at the command line for HDL code generation, use the `hdlcoder.createFloatingPointTargetConfig` function. You can use this function to create an `hdlcoder.FloatingPointTargetConfig` object for the native floating-point library.

```
nfpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT');
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', nfpconfig);
```

Optionally, you can specify the latency strategy and whether you want HDL Coder to handle denormal numbers in your design:

```
nfpconfig.LibrarySettings.HandleDenormals = 'on';
nfpconfig.LibrarySettings.LatencyStrategy = 'MAX';
```

To learn how you can verify the generated code, see “Verify the Generated Code from Native Floating-Point” on page 14-133.

Specify Native Floating Point for a MATLAB Function

You can specify the native floating-point settings for HDL code generation by using the MATLAB HDL Workflow Advisor or the command line.

To specify the native floating-point settings in the MATLAB HDL Workflow Advisor:

1. Open the MATLAB HDL Workflow Advisor. To get started with the MATLAB HDL Workflow Advisor, see “Basic HDL Code Generation and FPGA Synthesis from MATLAB”.
2. In the left pane, click the **HDL Code Generation** task. In the right pane, navigate to the **Floating Point** tab and set **Library** to `Native Floating Point`.
3. Click the **Clocks & Ports** tab and set **Oversampling factor** to a value greater than one.
4. Set **Latency Strategy** to `MIN`, `MAX`, or `ZERO` to map your design to minimum, maximum, or no latency, respectively. See Latency Strategy.
5. If you have denormal numbers in your design, select **Handle Denormals**. Denormal numbers are numbers that have an exponent field equal to zero and a nonzero mantissa field. See Handle Denormals.
6. If your design has multipliers, specify how you want HDL Coder to implement the multiplication operation by using the **Mantissa Multiplier Strategy** parameter. See Mantissa Multiplier Strategy.
7. In the left pane, right-click on the **HDL Code Generation** task and select **Run to Selected Task**.

To apply native floating point at the command line for HDL code generation, use the `coder.config` function to create a `coder.HdlConfig` object for HDL code generation and the `hdlcoder.createFloatingPointTargetConfig` function to create an `hdlcoder.FloatingPointTargetConfig` object for the native floating-point library.

```
hdlcfg = coder.config("hdl");
nfpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT');
hdlcfg.FloatingPointLibrary = 'NativeFloatingPoint';
hdlcfg.FloatingPointTargetConfiguration = nfpconfig;
```

Optionally, you can specify the latency strategy and whether you want HDL Coder to handle denormal numbers in your design.

```
nfpconfig.LibrarySettings.HandleDenormals = 'on';
nfpconfig.LibrarySettings.LatencyStrategy = 'MAX';
```

View Code Generation Report

To view the code generation reports of floating-point library mapping, before you begin code generation, enable generation of the Resource Utilization Report and Optimization Report. To enable the reports, on the **HDL Code** tab, click **Settings** > **Report Options**in the Configuration Parameters dialog box, on the **HDL Code Generation** pane, enable **Generate resource utilization report** and **Generate optimization report**. For more information, see “Create and Use Code Generation Reports” on page 23-2.

To see the list of native floating-point operators that HDL Coder supports and the floating-point operators to which your Simulink blocks mapped to, in the Code Generation Report, select **Native Floating-Point Resource Report**.

Native Floating-Point Resource Report for sfir_single

Summary of single precision native floating-point operators

adders	7
multipliers	4

A detailed report shows the various resources that the floating-point blocks use on the target device that you specify. HDL code generation from a native floating point design uses dynamic shift operations for aligning mantissa based on exponent values. Because these dynamic shifters are resource-expensive in the model, they are reported separately from static shift operators. For more information on dynamic and static shifters, see “Use Code Generation Reports to Evaluate Code Before Synthesis” on page 23-10.

Detailed Report

Module nfp_add_comp

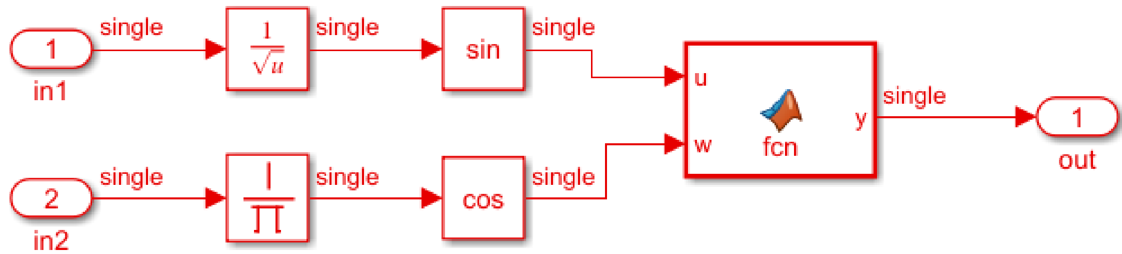
(Latency = "Max")

Multipliers	0
Adders/Subtractors	8
Registers	91
Total 1-Bit Registers	839
RAMs	0
Multiplexers	109
Static Shift operators	0
Dynamic Shift operators	2

To see the native floating-point settings that you applied to the model and whether HDL Coder successfully generated HDL code, in the Code Generation Report, select **Target Code Generation**.

Analyze Results

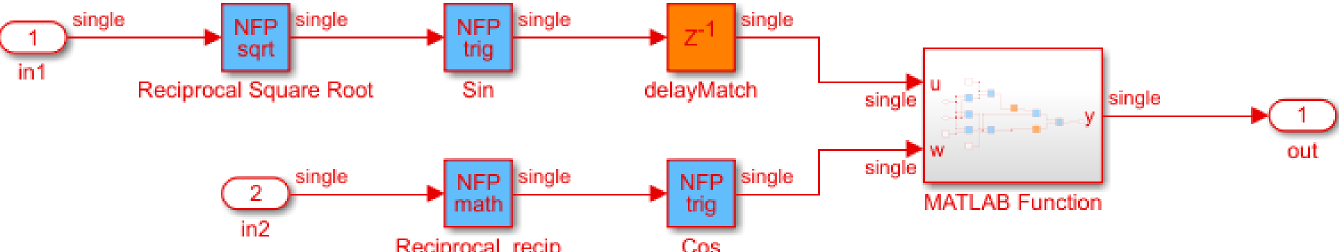
Floating point operators have a latency. If your Simulink model does not have delays, when you generate HDL code, the code generator figures out the operator latency and delay balances parallel paths. Consider this Simulink model that has two `single` inputs and gives a `single` output.



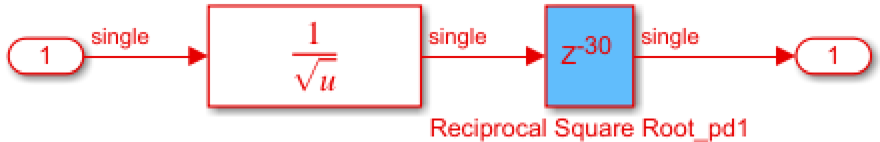
The MATLAB Function block in the Simulink model contains this code.

```
function y = fcn(u, w)
%#codegen

y1 = (u+w) * 20;
y2 = w^16;
y3 = (u-w) / 10;
y = y1 + y2 - y3;
```

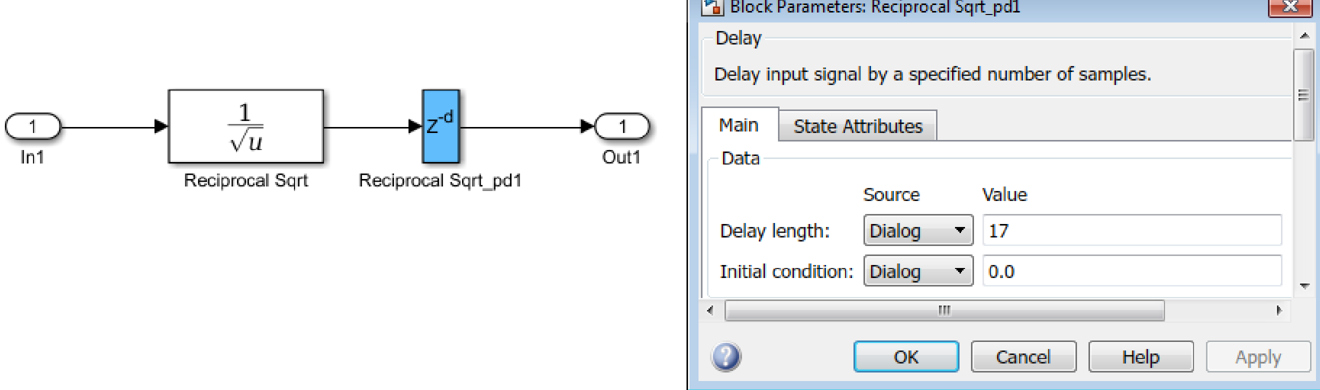
When you generate HDL code, the code generator maps the blocks in your Simulink model to synthesizable native floating-point operators. To see how the code generator implemented the floating-point operations, open the generated model. The blocks **NFP math**, **NFP Sqrt**, and **NFP trig** correspond to the floating-point implementation of the Reciprocal Sqrt, Reciprocal, sin, and cos blocks respectively in your original model.



Every floating-point operator has a latency. The code generator inserted an additional matching delay because the latency of the Reciprocal Sqrt is `30` and latency of Reciprocal is `31`. The operator latency is equal to the **Delay length** of the Delay block inside that `NFP` block. For example, if you double-click the `NFP sqrt` block, you can get the latency by looking at the **Delay length** of the `Delay` block. See “Latency Values of Floating-Point Operators” on page 14-99.



When you use MATLAB Function blocks with floating-point data types, HDL Coder uses the `MATLAB Datapath` architecture. This architecture treats the MATLAB Function block like a regular Subsystem block. When you generate code, the code generator maps the basic operations such as addition and multiplication to the corresponding native floating-point operators. Open the `MATLAB Function` subsystem to see how the code generator implemented the MATLAB Function block.



To learn more about the generated model, see "Generated Model and Validation Model" on page 21-10.

Limitation

To generate HDL code in native floating-point mode, use discrete sample times. Blocks operating at a continuous sample time are not supported.

See Also

Modeling Guidelines
"Modeling with Native Floating Point" on page 18-65

Functions
`createFloatingPointTargetConfig`

Related Examples

- "Floating Point Support: Field-Oriented Control Algorithm" on page 14-118

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 14-88
- “Simulink Blocks Supported by Using Native Floating Point” on page 14-137
- “Numeric Considerations for Native Floating-Point” on page 14-92

Floating Point Support: Field-Oriented Control Algorithm

In this example you review a Field-Oriented Control (FOC) algorithm for a Permanent Magnet Synchronous Machine (PMSM) implemented using single-precision and half-precision floating-point types.

Introduction

You have seen the fixed-point version of this design in “Field-Oriented Control of a Permanent Magnet Synchronous Machine” on page 14-66 that takes a deep dive into how to implement current control algorithm using fixed-point types. The model was converted to fixed-point before generating HDL code.

You can use floating-point single-precision types in your design and generate HDL code natively without converting to fixed-point types. This example shows design considerations when generating code from floating-point single-precision and half-precision variants of the fixed point model `hdlcoderFocCurrentFixptHdl`.

The testbench model `hdlcoderFocCurrentTestBench` has a reference block pointing to the DUT implemented in fixed-point or floating-point.

Salient features of Native Floating Point support

- Vendor independent and target agnostic RTL for FPGA and/or ASIC design
- Full range of IEEE-754 features including support for rounding modes, inf and nan data types, and optional support for denormal numbers.
- Extensive math block (`add,mul,div,recip,log,exp,sqrt,rsqrt`) and trigonometric block (`sin, cos, sincos, atan, atan2`) support.

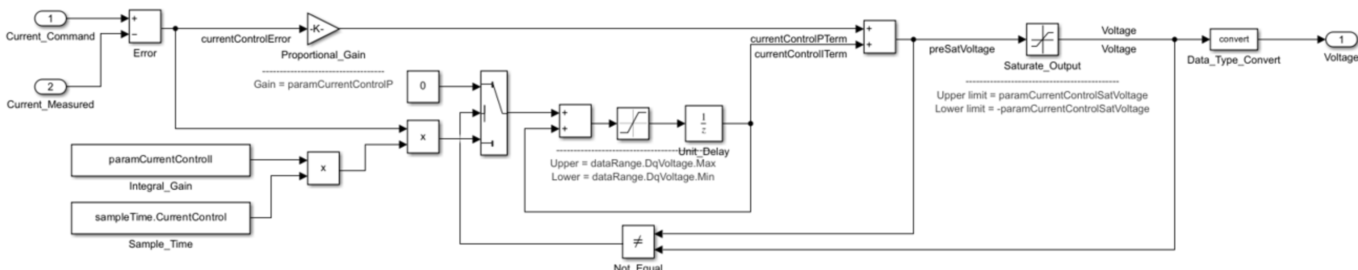
Why Use Floating-Point Types

Sometimes you might want to start in floating-point and stay in floating-point to target HDL, rather than convert to fixed-point, for the following reasons:

- Your algorithms have large or unknown dynamic ranges (for example integrators in feedback loops)
- Your algorithm uses operations that are difficult to design in fixed-point (ex: `atan2`)

In the fixed-point version of the example `hdlcoderFocCurrentFloatHDL.slx` you notice that several fixed-point rounding and saturation decisions are made to preserve the numerical behavior of the algorithm. For example the block `hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Saturate` which is a saturation block has been placed in the integrator loop so that results do not overflow due to accumulation in the loop.

```
open_system('hdlcoderFocCurrentFixptHdl');
open_system('hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control/
```



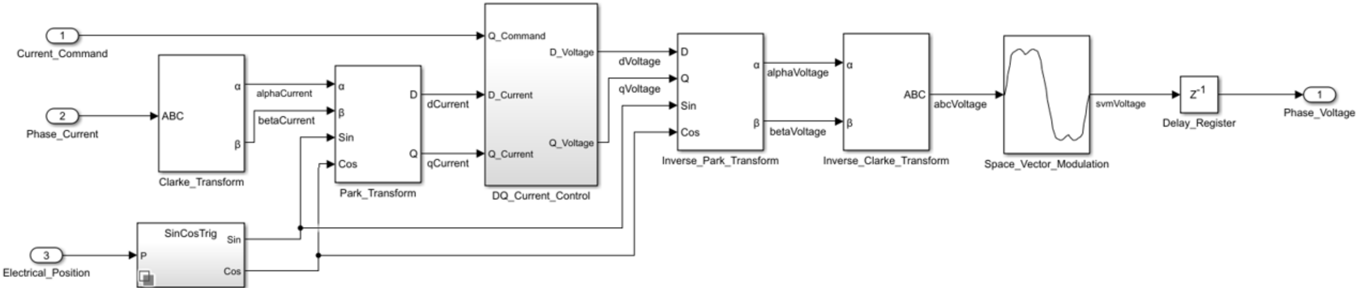
Sometimes, the task of fixed-point conversion could take several weeks to months with multiple levels of algorithm re-validation. It may also lead to undesirable loss of precision which might not be acceptable for some mission critical applications requiring very high accuracy.

In these situations, you can choose to use native floating point synthesis features available in HDL Coder.

Single-Precision FOC Model

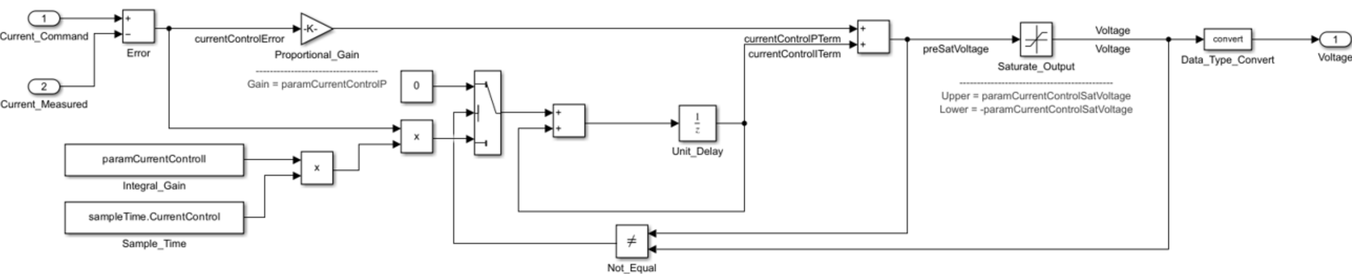
To open the single-precision version of the algorithm, run these commands:

```
load_system('hdlcoderFocCurrentFloatHdl');
open_system('hdlcoderFocCurrentFloatHdl/FOC_Current_Control')
```



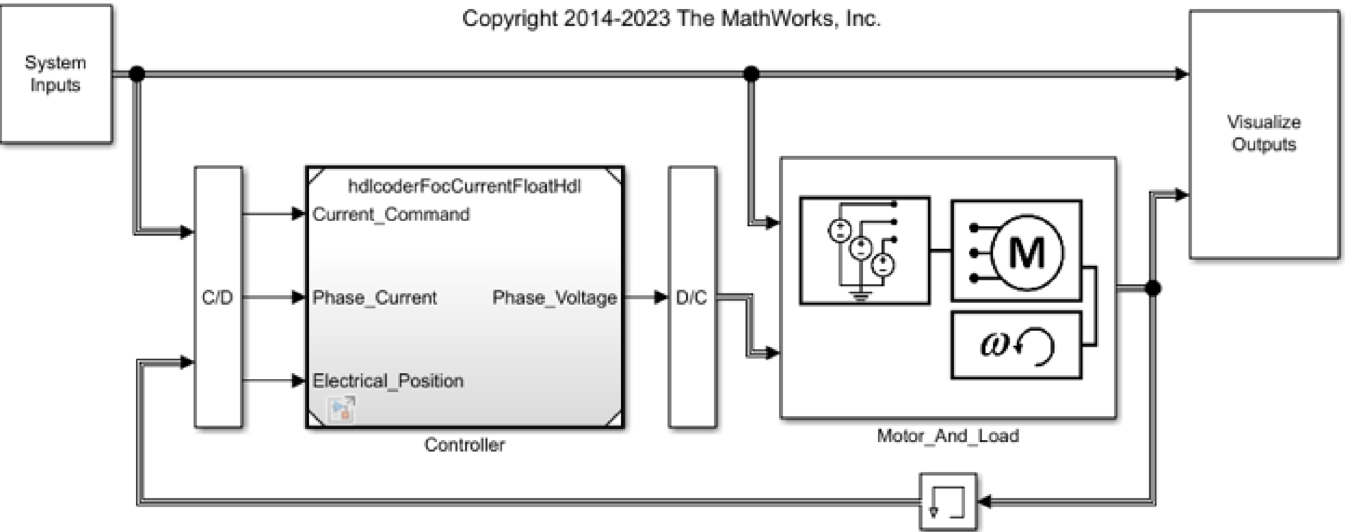
In comparison to the fixed-point algorithm, the single-precision model does not require additional rounding and saturation blocks and settings as can be seen in the floating-point version of the model `hdlcoderFocCurrentFloatHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control`

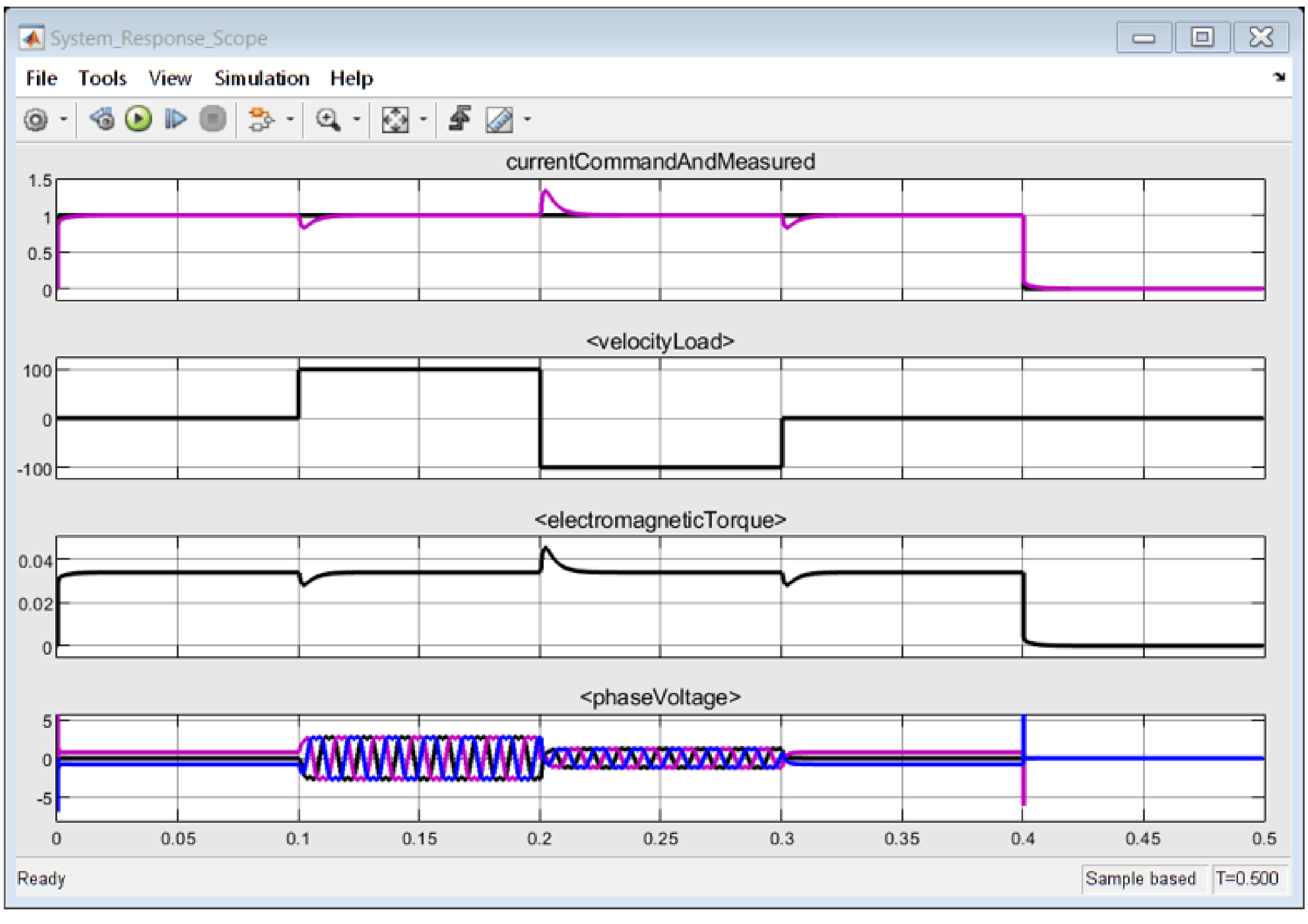
```
open_system('hdlcoderFocCurrentFloatHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control
```



To verify the behavior through simulation, run these commands:

```
hasSimPowerSystems = license ('test', 'Power_System_Blocks');
if hasSimPowerSystems
   open_system('hdlcoderFocCurrentTestBench')

   % set single-precision floating-point model in the testbench
   set_param('hdlcoderFocCurrentTestBench/Controller', 'ModelName', 'hdlcoderFocCurrentFloatHdl'

   set_param('hdlcoderFocCurrentTestBench','IgnoredZcDiagnostic','none');
   sim('hdlcoderFocCurrentTestBench')
   set_param('hdlcoderFocCurrentTestBench','IgnoredZcDiagnostic','warn');
end
```





You can generate HDL code and view the generated code for the controller.

```
hdlset_param('hdlcoderFocCurrentFloatHdl', 'FloatingPointTargetConfiguration', hdlcoder.createFlo
makehdl('hdlcoderFocCurrentFloatHdl/FOC_Current_Control');
```

```
### Begin compilation of the model 'hdlcoderFocCurrentFloatHdl'...
### Generating HDL for 'hdlcoderFocCurrentFloatHdl/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurre
### Running HDL checks on the model 'hdlcoderFocCurrentFloatHdl'.
### Working on the model 'hdlcoderFocCurrentFloatHdl'...
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocCurrentFloatHdl', 'Generate
### Begin model generation 'gm_hdlcoderFocCurrentFloatHdl' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: <a href="matlab:open_system('hdlsrc\hdlcoderFocCurrentFloatH
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentFloatHdl'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Begin VHDL Code Generation for 'FOC_Current_Control_tc'.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocCurrentFloatHdl\FOC_Current_Control_tc
### Code Generation for 'FOC_Current_Control_tc' completed.
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocCurrentFloatHdl', 'Traceabi
```

```
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_gain_pow2_single as hdlsrc\hdlc
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_mul_single as hdlsrc\hdlcoderFo
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_add_single as hdlsrc\hdlcoderFo
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_sub_single as hdlsrc\hdlcoderFo
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdlsrc\hdlcoder
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdlsrc\hdlcoder
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdlsrc\hdlcoder
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_uminus_single as hdlsrc\hdlcode
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_sincos_single as hdlsrc\hdlcode
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_add2_single as hdlsrc\hdlcoderF
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdlsrc\hdlcoder
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdlsrc\hdlcoder
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control as hdlsrc\hdlcoderFocCurrentFloatHd
### Generating package file hdlsrc\hdlcoderFocCurrentFloatHdl\FOC_Current_Control_pkg.vhd.
### Code Generation for 'hdlcoderFocCurrentFloatHdl' completed.
### Generating HTML files for code generation report at <a href="matlab:web('C:\Users\samule\OneD
### Creating HDL Code Generation Check Report file:///C:/Users/samule/OneDrive%20-%20MathWorks/Do
### HDL check for 'hdlcoderFocCurrentFloatHdl' complete with 0 errors, 0 warnings, and 2 message
### HDL code generation complete.
```

Half-Precision FOC Model

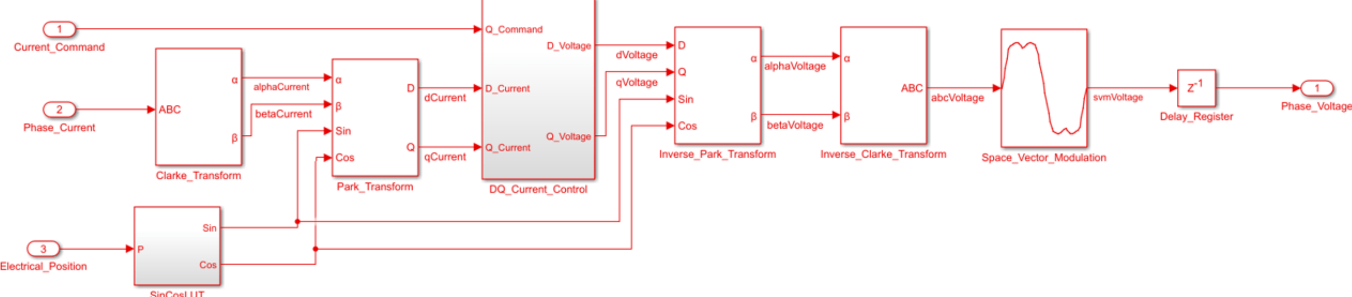
For applications that require smaller dynamic range, you can use `half` types without having to convert your design to use fixed-point types. Using `half` types consumes much less memory, has lower latency, and saves FPGA resources.

Advantages of half-precision floating point types in hardware perspective:

- Low latency
- Low Area
- High speed
- 16-bit floating point behavior which will be useful for optimal storage
- Wider dynamic range compared to integer or fixed point data types of same size

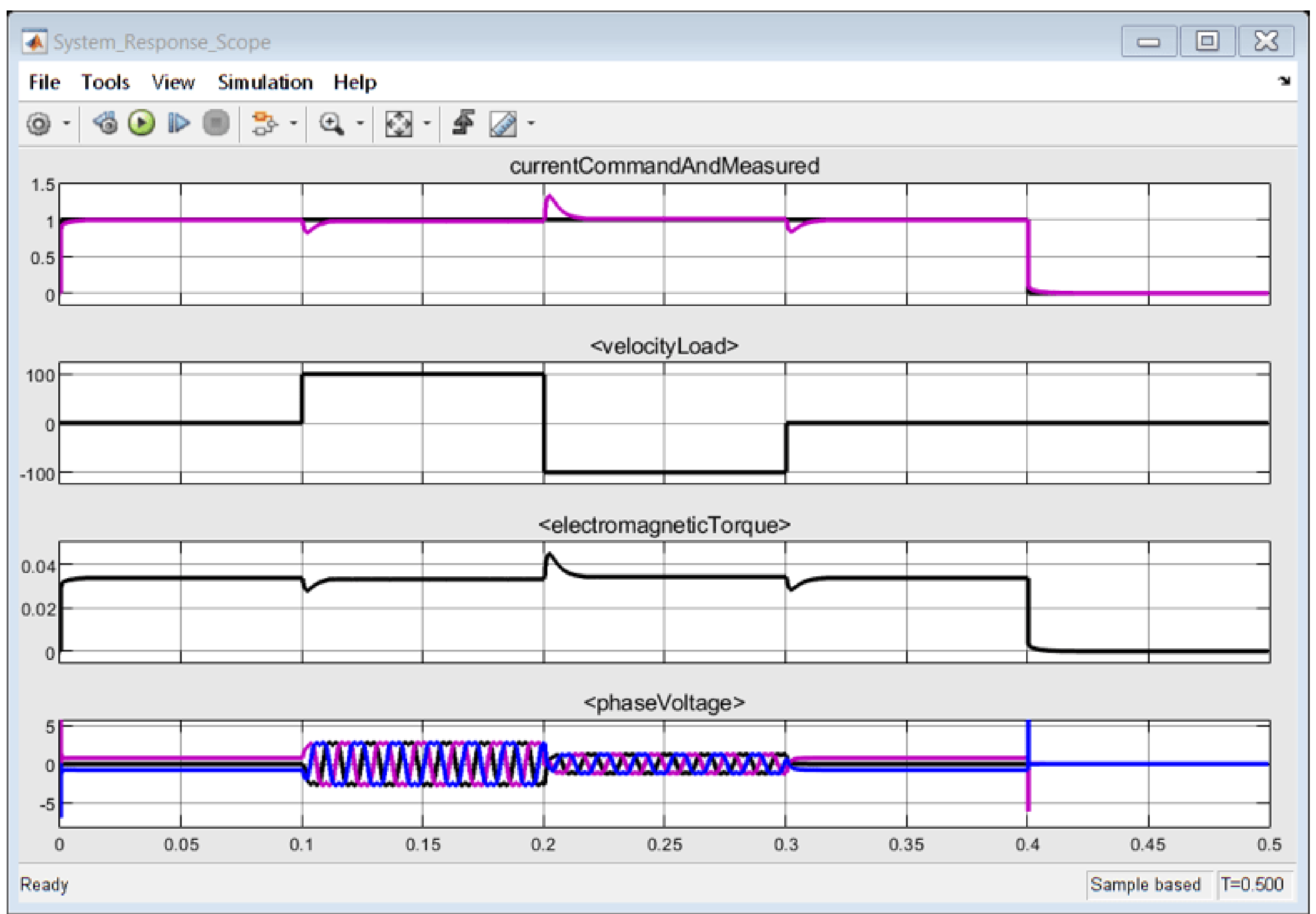
Half-precision types have same features as single- and double-precision floating-point types. Supported operators include basic arithmetic operators(EX: Add/Sub, Mul, Div/Recip) and Gain, Relational operators, Data type conversions.

```
load_system('hdlcoderFocCurrentFloatHalfHdl');
open_system('hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control')
```



To verify the behavior through simulation, run these commands:

```
hasSimPowerSystems = license ('test', 'Power_System_Blocks');
if hasSimPowerSystems
   open_system('hdlcoderFocCurrentTestBench')

   % set half-precision floating-point model in the testbench
   set_param('hdlcoderFocCurrentTestBench/Controller', 'ModelName', 'hdlcoderFocCurrentFloatHalfH

   set_param('hdlcoderFocCurrentTestBench','IgnoredZcDiagnostic','none');
   sim('hdlcoderFocCurrentTestBench')
   set_param('hdlcoderFocCurrentTestBench','IgnoredZcDiagnostic','warn');
end
```



You can generate HDL code and view the generated code for the controller.

```
hdlset_param('hdlcoderFocCurrentFloatHalfHdl', 'FloatingPointTargetConfiguration', hdlcoder.creat
makehdl('hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control');
```

```
### Generating HDL for 'hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurre
### Running HDL checks on the model 'hdlcoderFocCurrentFloatHalfHdl'.
### Begin compilation of the model 'hdlcoderFocCurrentFloatHalfHdl'...
### Begin compilation of the model 'hdlcoderFocCurrentFloatHalfHdl'...
### Working on the model 'hdlcoderFocCurrentFloatHalfHdl'...
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocCurrentFloatHalfHdl', 'Gene
```

```
### Begin model generation 'gm_hdlcoderFocCurrentFloatHalfHdl' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: <a href="matlab:open_system('hdlsrc\hdlcoderFocCurrentFloatH
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentFloatHalfHdl'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Begin VHDL Code Generation for 'FOC_Current_Control_tc'.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Contro
### Code Generation for 'FOC_Current_Control_tc' completed.
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_gain_pow2_half as hdlsrc\hd
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_add_half as hdlsrc\hdlcoder
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_mul_half as hdlsrc\hdlcoder
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_sub_half as hdlsrc\hdlcoder
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_relop_half as hdlsrc\hdlcod
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_relop_half as hdlsrc\hdlcod
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_uminus_half as hdlsrc\hdlco
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_relop_half as hdlsrc\hdlcod
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_convert_sfix_16_En14_to_hal
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_convert_half_to_sfix_16_En8
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_add2_half as hdlsrc\hdlcode
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_relop_half as hdlsrc\hdlcod
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_relop_half as hdlsrc\hdlcod
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control as hdlsrc\hdlcoderFocCurrentFlo
### Generating package file hdlsrc\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control_pkg.vhd.
### Code Generation for 'hdlcoderFocCurrentFloatHalfHdl' completed.
### Generating HTML files for code generation report at <a href="matlab:web('C:\Users\samule\OneD
### Creating HDL Code Generation Check Report file:///C:/Users/samule/OneDrive%20-%20MathWorks/Do
### HDL check for 'hdlcoderFocCurrentFloatHalfHdl' complete with 0 errors, 1 warnings, and 3 mess
### HDL code generation complete.
```

You can refer to documentation for full native floating-point capabilities available in HDL Coder. See link in the “Getting Started with HDL Coder Native Floating-Point Support” on page 14-88.

Design Model by Using HDL Coder Native Floating Point and Intel Hard Floating Point

This example shows how to create a model design that uses both HDL Coder™ native floating-point (NFP) and Intel® hard floating-point (HFP) IP. Using the NFP and HFP blocks together in a mixed design offers better FPGA resource utilizations. For complex models, this mixed design allows you to have bigger design into the FPGA fabric.

Generate HDL Code for Model with NFP Library

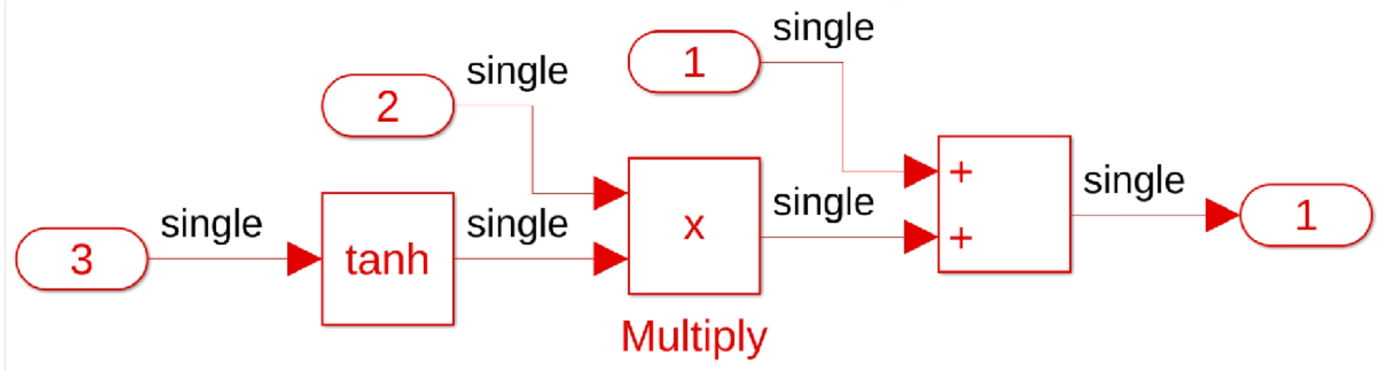
1. Set the synthesis tool path by using the function `hdlsetuptoolpath`. For this example, use Intel Quartus Pro as your synthesis tool. To set up tools in your environment, run the `hdlsetuptoolpath` command using the synthesis tool path on your computer. For example, the function `quartuspath` returns the Intel Quartus Pro synthesis tool path.

```
hdlsetuptoolpath('ToolName', 'Intel Quartus Pro','ToolPath', quartuspropath);
```

```
Prepending following Intel Quartus Pro path(s) to the system path:
B:\share\apps\HDLTools\Altera\21.3-mw-0_pro\Windows\quartus\bin64
```

2. Open the Simulink® model `hdlcoder_mixed_nfp_hfp`. The model consists of a Tanh, Multiply, and Add block. The input data type is single.

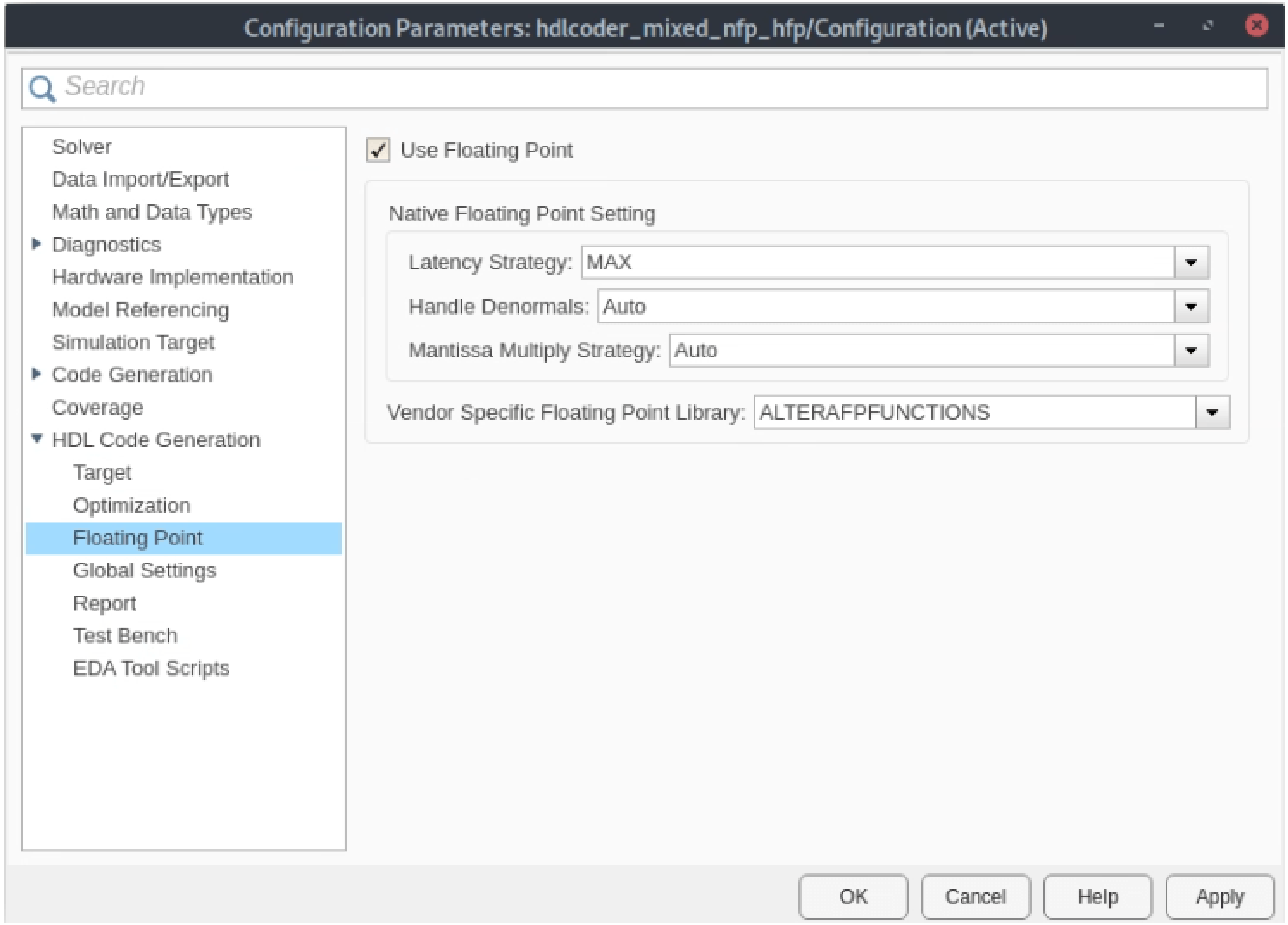
```
open_system('hdlcoder_mixed_nfp_hfp.slx');
```



3. Open the Configuration Parameters dialog box. In the **HDL Code Generation > Target** pane, set **Synthesis Tool** to `Intel Quartus Pro`, set **Family** to `Arria 10`, and set **Device** to `10AS016C3U19E2LG`.

4. In the **Floating Point** pane, select **Use Floating Point** to map the design to the native floating-point library. To map the design to a vendor-specific floating point library, set the **Vendor Specific Floating Point Library** to the library that aligns with your target hardware. In this example, the **Vendor Specific Floating Point Library** is set to `ALTERAFPFUNCTIONS`.

Click **OK** to save the changes.



5. To compile and simulate the generated code with QuestaSim, at the command line, set the Altera simulation library path by using the `hdlsetup_param` function. In this example, that path is saved in variable `alterasimulationlibpath`. For example, the simulation path can be `C:\HDLTools\Altera\Questa_SimLibs\15.1\`.

```
hdlset_param('hdlcoder_mixed_nfp_hfp', 'SimulationLibPath', alterasimulationlibpath);
```

6. Generate HDL code for the DUT subsystem, `hdlcoder_mixed_nfp_hfp/DUT`, by using the `makehdl` function.

```
makehdl('hdlcoder_mixed_nfp_hfp/DUT')
```

```
### Generating HDL for 'hdlcoder_mixed_nfp_hfp/DUT'.
### Using the config set for model hdlcoder_mixed_nfp_hfp for HDL code generation parameters.
### Running HDL checks on the model 'hdlcoder_mixed_nfp_hfp'.
### Begin compilation of the model 'hdlcoder_mixed_nfp_hfp'...
### Working on the model 'hdlcoder_mixed_nfp_hfp'...
### 'AdaptivePipelining' is set to 'Off' for the model. 'AdaptivePipelining' can improve the ach
### 'LUTMapToRAM' is set to 'On' for the model. This option is used to map lookup tables to a blo
### Using B:\share\apps\HDLTools\Altera\21.3-mw-0_pro\Windows\quartus\bin64\..\sopc_builder\bin\
***************************************************************
Quartus is a registered trademark of Intel Corporation in the
```

```
US and other countries.  Portions of the Quartus Prime software
code, and other portions of the code included in this download
or on this DVD, are licensed to Intel Corporation and are the
copyrighted property of third parties. For license details,
refer to the End User License Agreement at
http://fpgasoftware.intel.com/eula.
****************************************************************

2023.04.28.12:57:31 Info: alterafpf_add_single.alterafpf_add_single: B:/share/apps/hdltools/alte
2023.04.28.12:57:31 Info: alterafpf_add_single.alterafpf_add_single: B:/share/apps/hdltools/alte
2023.04.28.12:57:31 Info: Deploying alterafpf_add_single to C:\Users\clewis\OneDrive - MathWorks\
### Generating Altera(R) megafunction: alterafpf_add_single for target frequency of 100 MHz.
### alterafpf_add_single takes 3 cycles.
### Done.
****************************************************************
Quartus is a registered trademark of Intel Corporation in the
US and other countries.  Portions of the Quartus Prime software
code, and other portions of the code included in this download
or on this DVD, are licensed to Intel Corporation and are the
copyrighted property of third parties. For license details,
refer to the End User License Agreement at
http://fpgasoftware.intel.com/eula.
****************************************************************

2023.04.28.12:59:30 Info: alterafpf_mul_single.alterafpf_mul_single: B:/share/apps/hdltools/alte
2023.04.28.12:59:30 Info: alterafpf_mul_single.alterafpf_mul_single: B:/share/apps/hdltools/alte
2023.04.28.12:59:30 Info: Deploying alterafpf_mul_single to C:\Users\clewis\OneDrive - MathWorks\
### Generating Altera(R) megafunction: alterafpf_mul_single for target frequency of 100 MHz.
### alterafpf_mul_single takes 3 cycles.
### Done.
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 49 cycles.
### Working on... GenerateModel
### Begin model generation 'gm_hdlcoder_mixed_nfp_hfp' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_mixed_nfp_hfp'.
### Working on hdlcoder_mixed_nfp_hfp/DUT/nfp_tanh_single as hdl_prj_mixed\hdlsrc\hdlcoder_mixed_
### Working on hdlcoder_mixed_nfp_hfp/DUT as hdl_prj_mixed\hdlsrc\hdlcoder_mixed_nfp_hfp\DUT.vhd
### Generating package file hdl_prj_mixed\hdlsrc\hdlcoder_mixed_nfp_hfp\DUT_pkg.vhd.
### Code Generation for 'hdlcoder_mixed_nfp_hfp' completed.
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'hdlcoder_mixed_nfp_hfp' complete with 0 errors, 0 warnings, and 2 messages.
### HDL code generation complete.
```

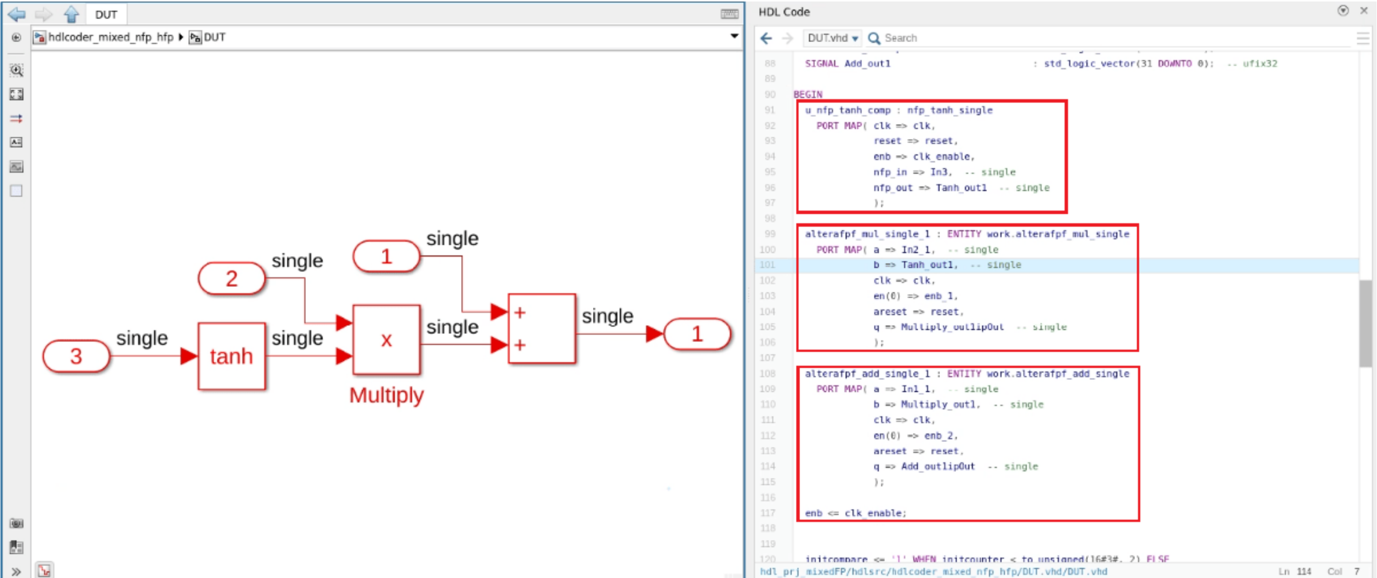
6. Generate the test bench for the Simulink model.

```
makehdltb('hdlcoder_mixed_nfp_hfp/DUT')
```

```
### Begin TestBench generation.
### Generating HDL TestBench for 'hdlcoder_mixed_nfp_hfp/DUT'.
### Begin compilation of the model 'hdlcoder_mixed_nfp_hfp'...
### Begin compilation of the model 'gm_hdlcoder_mixed_nfp_hfp'...
### Begin simulation of the model 'gm_hdlcoder_mixed_nfp_hfp'...

### Collecting data...
```

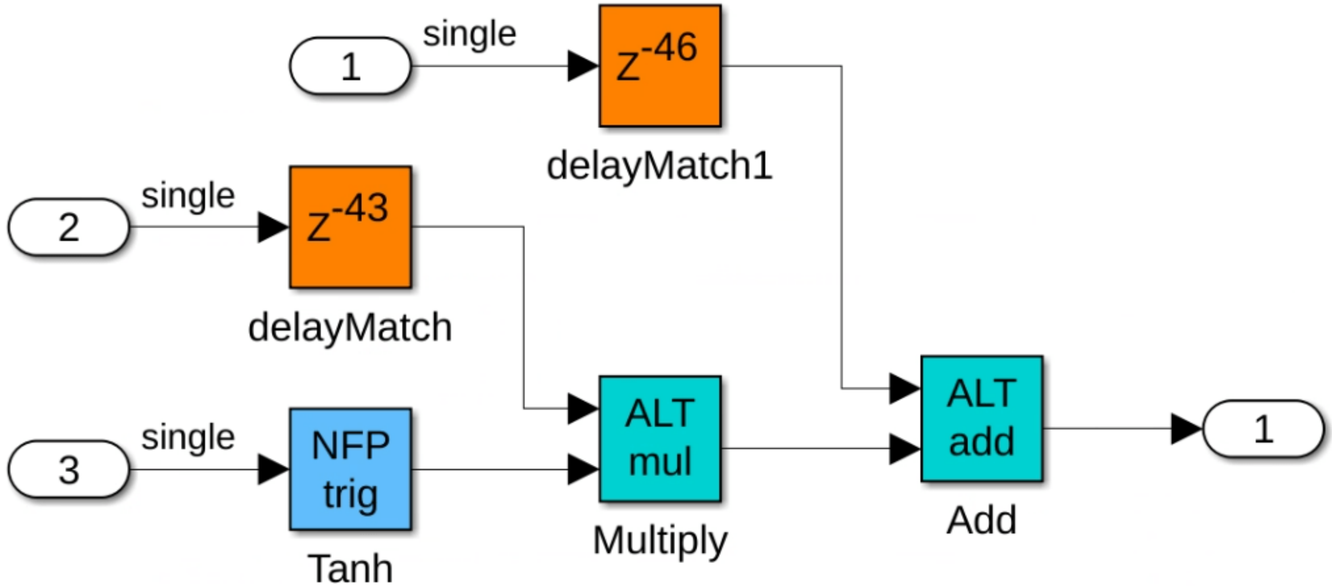
```
### Generating test bench data file: hdl_prj_mixed\hdlsrc\hdlcoder_mixed_nfp_hfp\Out1_expected.da
### Working on DUT_tb as hdl_prj_mixed\hdlsrc\hdlcoder_mixed_nfp_hfp\DUT_tb.vhd.
### Generating package file hdl_prj_mixed\hdlsrc\hdlcoder_mixed_nfp_hfp\DUT_tb_pkg.vhd.
### HDL TestBench generation complete.
```

7. After `makehdl` runs,the **HDL Code** pane opens and displays the VHDL code for the components of the model. The generated `DUT.vhd` file shows the trigonometric function component `tanh` maps to a native floating-point component, and the `Multiply` and `Add` components map to hard floating-point components.



8. Open the generated model file `gm_hdlcoder_mixed_nfp_hfp.slx`. The blocks that map to native floating-point components are light blue and display `NFP` on the block mask. The blocks that map to the vendor-specific floating point components are cyan and have initials associated with the specified vendor library on the block mask. In this example, the blocks that map to vendor-specific floating point IP from the `ALTERAFPFUNCTIONS` library display the initials `ALT`. The Tanh block maps to a native floating-point component and the Multiply and Add blocks map to hard floating-point components.

HDL Coder adds matching delays to balance the added latency from the floating-point operator blocks. Double-click each floating-point block to see the latency added for each. For more information on latency with floating-point operators, see “Latency Considerations with Native Floating Point” on page 14-104.



9. The `makehdl` function generates the `DUT_compile.do` file that contains these commands to compile generated HDL files.

- `vlib work`
- `vmap -c`
- `vmap -c -modelsimini /mathworks/devel/src/commercial/HDLTools/Altera/Questa_SimLibs/18.1_20.2/modelsim.ini`
- `vlib altera_fp_functions_1911`
- `vcom -work altera_fp_functions_1911 Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_add_single/altera_fp_functions_1911/dspba_library_package.vhd`
- `vcom -work altera_fp_functions_1911 Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_add_single/altera_fp_functions_1911/dspba_library.vhd`
- `vcom -work altera_fp_functions_1911 Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_add_single/altera_fp_functions_1911/alterafpf_add_single_altera_fp_functions_1911_y7txnay.vhd`
- `vcom -work altera_fp_functions_1911 Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_mul_single/altera_fp_functions_1911/alterafpf_mul_single_altera_fp_functions_1911_wfujeaa.vhd`
- `vlib work`
- `vcom DUT_pkg.vhd`
- `vcom Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_add_single.vhd`
- `vcom Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_mul_single.vhd`
- `vcom nfp_tanh_single.vhd`
- `vcom DUT.vhd`

10. The `makehdltb` function generates the `DUT_tb_compile.do` file that contains these commands to compile generated HDL files.

- `vlib work`
- `vmap -c`
- `vmap -c -modelsimini /mathworks/devel/src/commercial/HDLTools/Altera/Questa_SimLibs/18.1_20.2/modelsim.ini`
- `vlib altera_fp_functions_1911`
- `vcom -work altera_fp_functions_1911 Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_add_single/altera_fp_functions_1911/dspba_library_package.vhd`
- `vcom -work altera_fp_functions_1911 Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_add_single/altera_fp_functions_1911/dspba_library.vhd`
- `vcom -work altera_fp_functions_1911 Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_add_single/altera_fp_functions_1911/alterafpf_add_single_altera_fp_functions_1911_y7txnay.vhd`
- `vcom -work altera_fp_functions_1911 Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_mul_single/altera_fp_functions_1911/alterafpf_mul_single_altera_fp_functions_1911_wfujeaa.vhd`
- `vlib work`
- `vcom DUT_pkg.vhd`
- `vcom Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_add_single.vhd`
- `vcom Altera/Arria_10/10AS016C3U19E2LG/F100/synth/alterafpf_mul_single.vhd`
- `vcom nfp_tanh_single.vhd`
- `vcom DUT.vhd`
- `vcom DUT_tb_pkg.vhd`
- `vcom DUT_tb.vhd`

11. The `makehdltb` function generates the `DUT_tb_sim.do` file that contains these commands for simulation when using the ModelSim® software.

- `onbreak resume`
- `onerror resume`
- `vsim -t 1ps -L lpm -L altera_mf -voptargs=+acc work.DUT_tb`
- `add wave sim:/DUT_tb/u_DUT/clk`
- `add wave sim:/DUT_tb/u_DUT/reset`
- `add wave sim:/DUT_tb/u_DUT/clk_enable`
- `add wave sim:/DUT_tb/u_DUT/In1`
- `add wave sim:/DUT_tb/u_DUT/In2`
- `add wave sim:/DUT_tb/u_DUT/In3`
- `add wave sim:/DUT_tb/u_DUT/ce_out`
- `add wave sim:/DUT_tb/u_DUT/Out1`
- `add wave sim:/DUT_tb/Out1_ref`
- `run -all`

Verify Functionality by Using ModelSim Simulation

Open ModelSim and navigate to the folder containing the DO files.

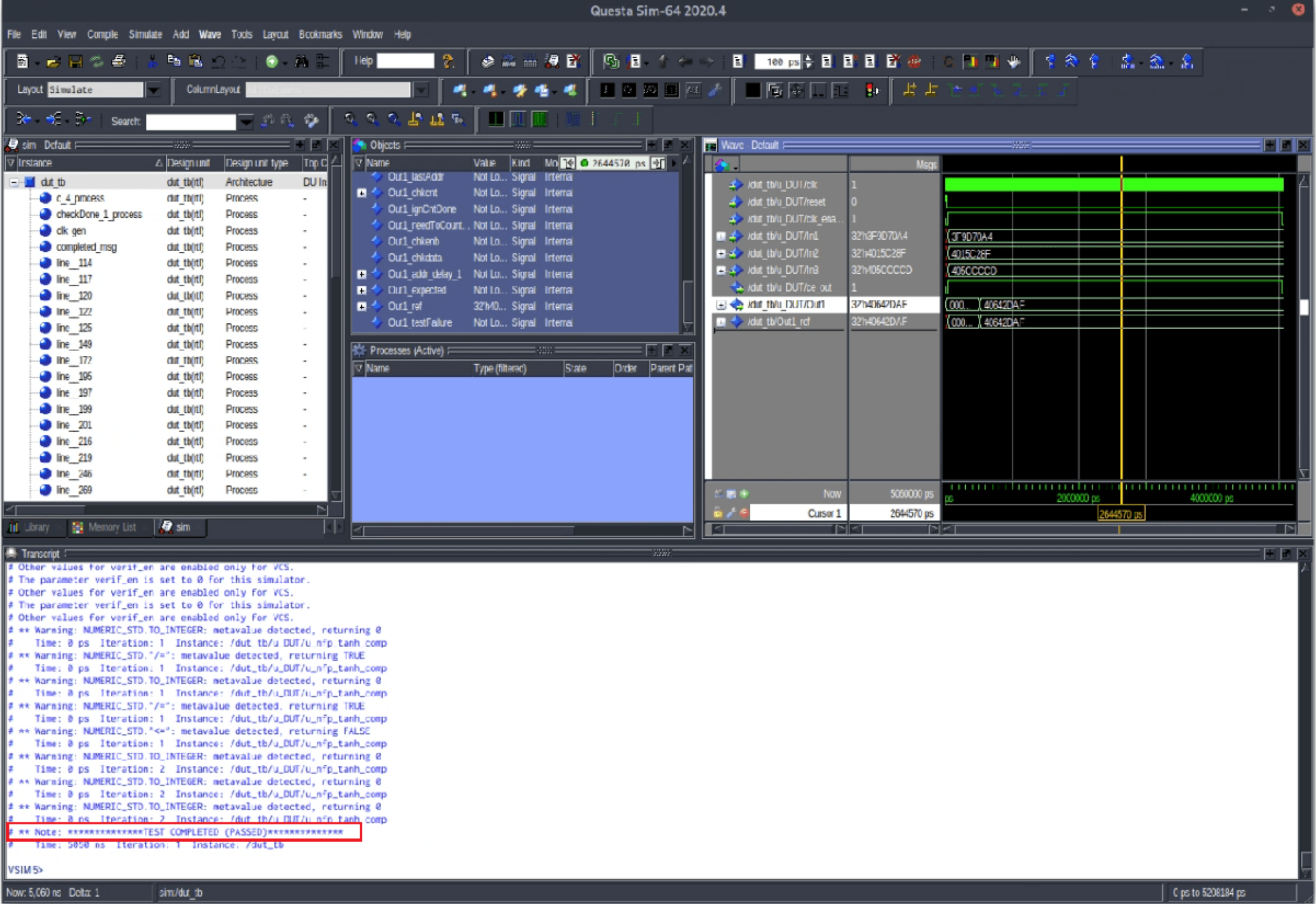
Run the DO files in this order:

```
do DUT_compile.do
```

```
do DUT_tb_compile.do
```

```
do DUT_tb_sim.do
```

This figure shows the ModelSim simulation with the a message at the end of the log to indicate that the tests passed.



Design Synthesis in Intel Quartus Pro

To perform design synthesis of the mixed NFP and HFP design in Intel Quartus Pro software, follow these steps:

1. Open Intel Quartus Pro and create a project.

2. Add the generated HDL files to the Quartus project, including the top module DUT and the other supporting files.

3. Set DUT as top module in the Quartus project.

4. Perform design synthesis by following the Intel Quartus Pro workflow.

See Also

`hdlcoder.FloatingPointTargetConfig` | `createFloatingPointTargetConfig` | `hdlcoder.FloatingPointTargetConfig.IPConfig` | `customize`

Related Examples

- “FPGA Floating-Point Library IP Mapping” on page 29-25
- “Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries” on page 29-19

More About

- “Customize Floating-Point IP Configuration” on page 29-36
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 29-41

Verify the Generated Code from Native Floating-Point

In this section...

"Specify the Tolerance Strategy" on page 14-133
"Verify the Generated Code with HDL Test Bench" on page 14-134
"Verify the Generated Code with Cosimulation" on page 14-134
"Limitation" on page 14-136

HDL Coder native floating-point technology can generate target-independent HDL code from your floating-point design. You can synthesize your floating-point design on any generic FPGA or ASIC. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

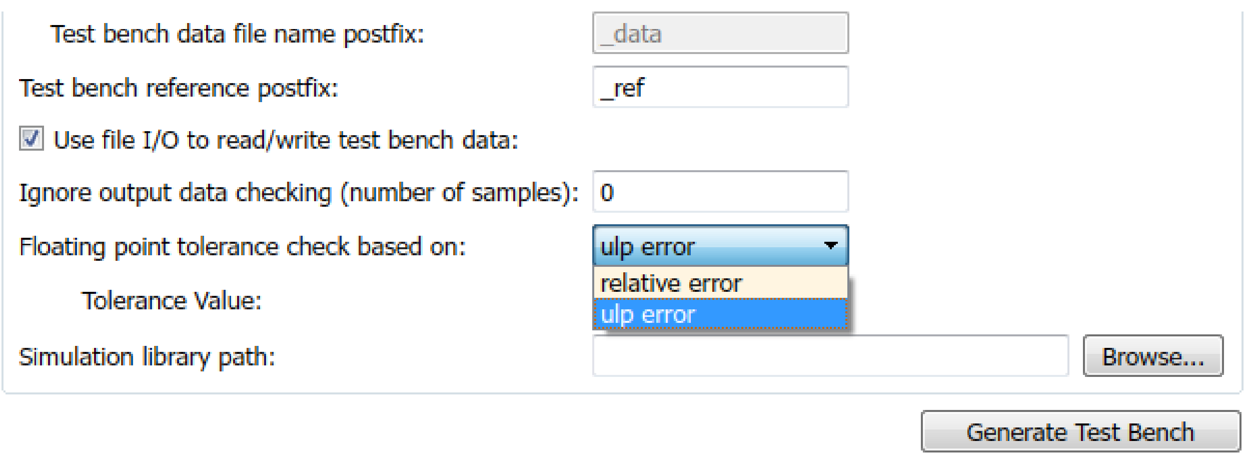
When representing infinitely real numbers with a finite number of bits, there can be rounding errors with the correct rounding range of values that the IEEE-754 standard specifies. To measure the rounding errors, you can specify the floating-point tolerance check based on `relative error` or `ulp error`. For more information about these rounding errors, see "Relative Accuracy and ULP Considerations" on page 14-93.

Specify the Tolerance Strategy

Before generating the testbench, specify the floating-point tolerance check for verifying the generated code.

To specify the tolerance check in the Configuration Parameters dialog box:

1. In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
2. Click **Settings**. In the **HDL Code Generation** > **Testbench** pane, for **Floating point tolerance check based on**, specify `relative error` or `ulp error`.



3. Enter the **Tolerance Value** and click **Apply**. If you choose `relative error`, the default is a tolerance value of `1e-07`. If you choose `ulp error`, the default tolerance value is zero. To learn more, see “Numeric Considerations for Native Floating-Point” on page 14-92.

To specify the tolerance strategy at the command-line, use:

1. Specify the floating point tolerance check setting by using `FPToleranceStrategy`.

```
% check for floating-point tolerance based on relative error
hdlset_param('sfir_single', 'FPToleranceStrategy', 'Relative');

% check for floating-point tolerance based on the ULP error
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP');
```
2. Based on the `FPToleranceStrategy` setting, enter the tolerance value by using `FPToleranceValue`.

```
 % if using relative error, enter custom tolerance value
hdlset_param('FP_test_16a', 'FPToleranceValue', 1e-06);

% if using ULP error, enter tolerance value greater
% than or equal to 1
hdlset_param('FP_test_16a', 'FPToleranceValue', 1);
```

Verify the Generated Code with HDL Test Bench

To generate an HDL test bench for verifying the generated code:

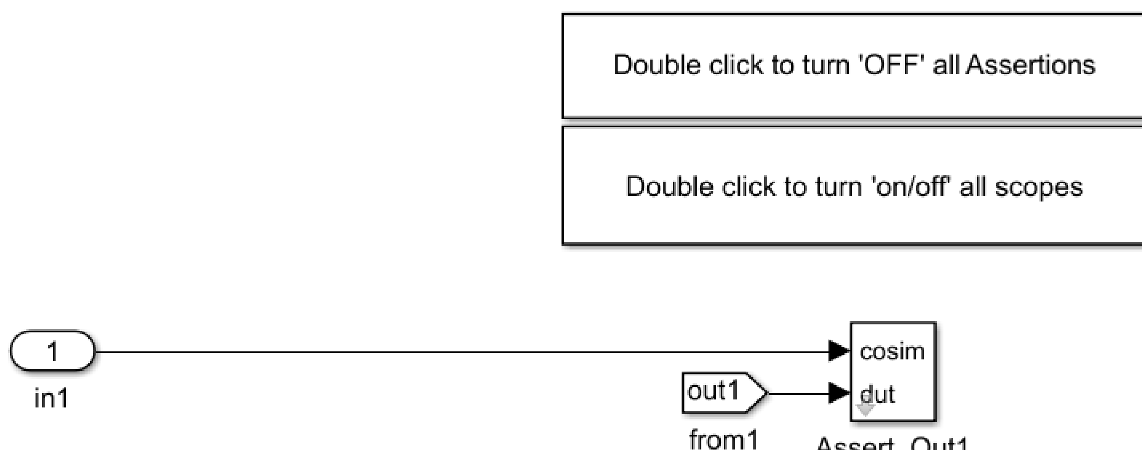
1. In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Test Bench** pane, in the Test Bench Generation Output section, select **HDL test bench**.
2. In the Configuration section, make sure that **Use file I/O to read/write test bench data** is enabled. To generate a test bench that uses constants instead of file I/O, clear **Use file I/O to read/write test bench data**.
3. Click **Apply**, and then click **Generate Test Bench**.

To learn more about how HDL test bench generation works, see “Test Bench Generation” on page 6-5.

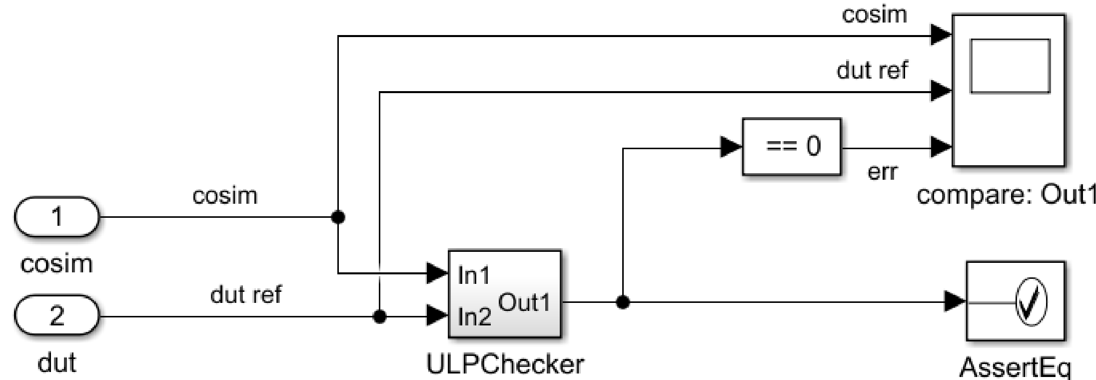
Verify the Generated Code with Cosimulation

To generate a cosimulation model for verifying the generated code:

1. In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Test Bench** pane, for **Cosimulation model for use with**, select the cosimulation tool.
2. Click **Apply**, and then click **Generate Test Bench**.
3. After test bench generation, save the cosimulation model. In the model, double-click the `Compare` subsystem.

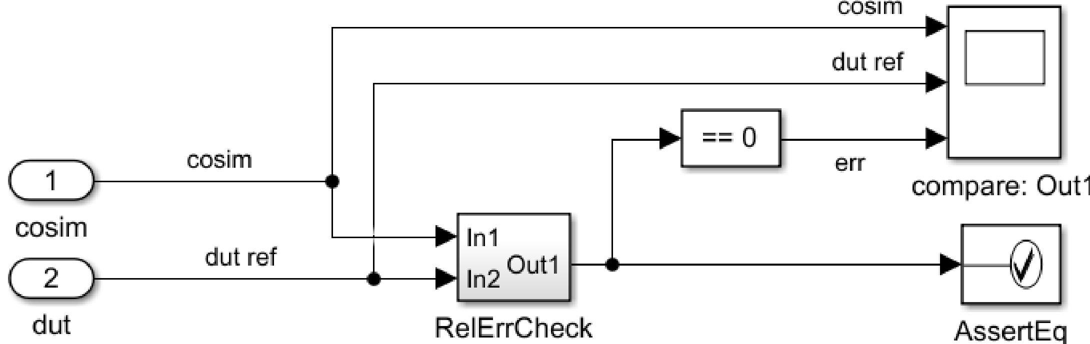


4. If you double-click the `Assert_Out1` block, the block parameters show the **Tolerance Value** that you specify.
5. To look inside the `Assert_Out1` block, click the mask. If you specify the floating-point tolerance check based on `ulp error`, the model shows a `ULPChecker` block.



The `ULPChecker` has a MATLAB Function block that shows how HDL Coder accounts for the ULP error when checking for numerical accuracy.

If you specify the floating-point tolerance check based on `relative error`, the model shows a `RelErrCheck` block.



`RelerrCheck` has a MATLAB Function block that shows how HDL Coder accounts for the relative error when checking for numerical accuracy.

6. In the Simulink Editor for the model, start simulation. At the end of cosimulation, check the `compare: Out1` scope.

   The scope compares the difference between the result signal from the cosimulation block and the reference signal from the DUT.

See also “Generate a Cosimulation Model” on page 25-43.

Limitation

When verifying the generated code, constructs that use IEEE standards prior to VHDL-2008 are not supported with native floating-point.

See Also

Modeling Guidelines
“Modeling with Native Floating Point” on page 18-65

Functions
`createFloatingPointTargetConfig`

Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 14-118

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 14-88
- “Numeric Considerations for Native Floating-Point” on page 14-92
- “Simulink Blocks Supported by Using Native Floating Point” on page 14-137

Simulink Blocks Supported by Using Native Floating Point

In this section...

"HDL Floating Point Operations Library" on page 14-137
"Supported Simulink Blocks in Math Operations Library" on page 14-138
"Supported Functions in Math Function Block" on page 14-139
"Supported Simulink Blocks in Other Libraries" on page 14-139
"Simulink Block Restrictions" on page 14-141

HDL Coder native floating point can generate target-independent HDL code from your floating-point design. You can synthesize your floating-point design on any generic FPGA or ASIC. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

HDL Coder supports several Simulink blocks, including math and trigonometric blocks by using native floating-point technology.

HDL Floating Point Operations Library

In the **HDL Floating Point Operations** library, HDL Coder supports blocks that have `single` and `double` data types in the `Native Floating Point` mode. For certain blocks, such as Discrete-Time Integrator and Discrete PID Controller, use zero latency strategy.

When you use `half` types, these blocks are supported in `Native Floating Point` mode.

- Add
- Bus to Vector
- Divide
- Dot Product
- Float Typecast
- Gain
- Logical Operator
- Multiply-Add
- Probe
- Product
- Product of Elements
- Reciprocal
- Relational Operator
- Subtract
- Sum of Elements
- Signal Conversion
- Terminator
- Transpose

- Unary Minus

Supported Simulink Blocks in Math Operations Library

In the `Math Operations` library, these blocks are supported for HDL code generation:

Block Name	Supported by Half Data Types	Supported by Single Data Types	Supported by Double Data types	Complex Data Support	Remarks
Abs	No	Yes	Yes	Yes	Not Applicable
Add	Yes	Yes	Yes	Yes	Not applicable
Assignment	No	Yes	Yes	Yes	Not applicable
Bias	No	Yes	Yes	Yes	Not applicable
Complex to Real-Imag	No	Yes	Yes	Yes	Not applicable
Divide	Yes	Yes	Yes	No	Not applicable
Dot Product	Yes	Yes	Yes	No	Not applicable
Gain	Yes	Yes	Yes	Yes	Not applicable
MinMax	No	Yes	Yes	No	Not applicable
Multiply-Add	Yes	Yes	Yes	Yes	Not applicable
Product	Yes	Yes	Yes	Yes	If you configure the block to perform matrix multiplication by setting the **Multiplication** block parameter to `Matrix(*)`, the **DotProductStrategy** must be set to `Fully Parallel`.
Product of Elements	Yes	Yes	Yes	Yes	Not applicable
Real-Imag to Complex	No	Yes	Yes	No	Not applicable
Reciprocal Sqrt	No	Yes	Yes	No	Not applicable
Reshape	Yes	Yes	Yes	Yes	Not applicable
Sqrt	No	Yes	Yes	No	Not applicable
Subtract	Yes	Yes	Yes	Yes	Not applicable
Sum	Yes	Yes	Yes	Yes	Not applicable
Sum of Elements	Yes	Yes	Yes	Yes	Not applicable
Trigonometric Function	No	Yes	No	No	Not applicable
Unary Minus	Yes	Yes	Yes	Yes	Not applicable

Block Name	Supported by Half Data Types	Supported by Single Data Types	Supported by Double Data types	Complex Data Support	Remarks
Vector Concatenate, Matrix Concatenate	Yes	Yes	Yes	Yes	Not applicable

Supported Functions in Math Function Block

In the Math Function block, these functions are supported for HDL code generation:

Math Functions	Supported Floating-Point Datatypes			Complex Data Support
	Half	Single	Double	
`exp`	No	Yes	No	No
`log`	No	Yes	Yes	No
`10^u`	No	Yes	No	No
`log10`	No	Yes	No	No
`magnitude^2`	No	Yes	Yes	Yes
`square`	No	Yes	No	Yes
`pow`	No	Yes	No	No
`conj`	No	Yes	No	Yes
`reciprocal`	Yes	Yes	Yes	No
`hypot`	No	Yes	No	No
`rem`	No	Yes	No	No
`mod`	No	Yes	No	No
`transpose`	Yes	Yes	Yes	Yes
`hermitian`	No	Yes	No	Yes

Supported Simulink Blocks in Other Libraries

The table shows the list of supported blocks for HDL code generation in other **HDL Coder** block libraries.

Block Library	Supported by Half Data Types	Supported by Single Data Types	Supported by Double Data Types
Discrete	The supported blocks include the set of delay blocks including the synchronous blocks, and Integer Delay, and Tapped Delay.	The supported blocks include: • Zero Order Hold • Set of delay blocks including the synchronous blocks, and Integer Delay, and Tapped Delay.	The supported blocks include: • Zero Order Hold • Set of delay blocks including the synchronous blocks, and Integer Delay, and Tapped Delay.
HDL Operations	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
HDL RAMs	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
HDL Subsystems	All blocks are supported.	All blocks are supported.	All blocks are supported.
Logic and Bit Operations	Supported blocks include: • Compare to Constant • Relational Operator • Logical Operator • Detect Change • Detect Increase • Detect Decrease	All blocks are supported.	All blocks are supported.
Lookup Tables	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
Model Verification	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
Model-Wide Utilities	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
Ports & Subsystems	Enable, reset, input, and output ports, model references, and subsystem blocks are supported.	Enable, reset, input, and output ports, model references, and subsystem blocks are supported.	Enable, reset, input, and output ports, model references, and subsystem blocks are supported.

Block Library	Supported by Half Data Types	Supported by Single Data Types	Supported by Double Data Types
Signal Attributes	Supported blocks include: • Data Type Conversion • Data Type Duplicate • Rate Transition • Signal Specification • Signal Conversion • Bus to Vector • Probe	All blocks are supported.	All blocks are supported.
Signal Routing	All blocks are supported.	All blocks are supported.	All blocks are supported.
Sources	The supported blocks include Inport, Constant, and Ground blocks.	The supported blocks include Inport, Constant, and Ground blocks.	The supported blocks include Inport, Constant, and Ground blocks.
Sinks	All blocks are supported.	All blocks are supported.	All blocks are supported.
User-Defined Functions	MATLAB Function blocks are supported.	MATLAB Function blocks are supported.	MATLAB Function blocks are supported.

Simulink Block Restrictions

In native floating-point mode, the code generator does not support these blocks or block architectures:

- Biquad Filter.
- Sum of Elements with complex input types.
- MATLAB System blocks.
- Dot Product in complex mode with **Architecture** as `Tree` or `Linear`.
- Discrete FIR Filter with **Architecture** other than `Fully Parallel`.
- Dead Zone and Dead Zone Dynamic.
- Polar to Cartesian.
- For the Data Type Conversion block:
  - `Stored Integer (SI)` mode for **Input and output to have equal** setting is not supported.
  - The **Saturate on integer overflow** check box must be left cleared.

See Also

Modeling Guidelines
“Modeling with Native Floating Point” on page 18-65

Functions
`createFloatingPointTargetConfig`

Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 14-118

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 14-88
- “Numeric Considerations for Native Floating-Point” on page 14-92
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111

Synthesis Benchmark of Common Native Floating Point Operators

This example shows how to access and generate synthesis benchmarks for common native floating-point operators with Xilinx® Vivado® and Intel® Quartus® tool.

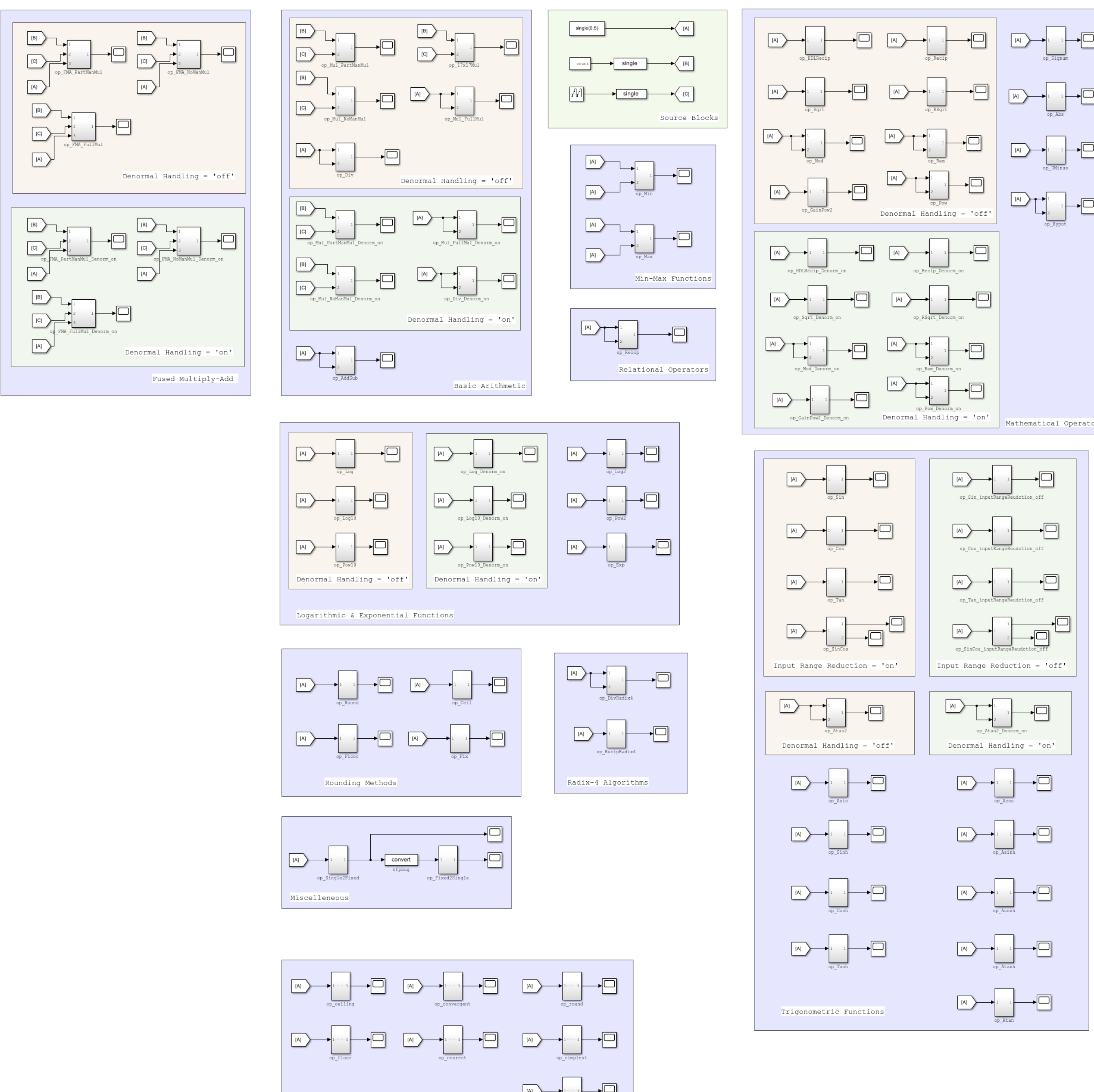
Access Generated Synthesis Results from a MAT File

Perform synthesis and timing analysis on common operators with Xilinx Vivado and Intel Quartus tool. These operators include basic math operators such as addition and subtraction, as well as more complex operators such as log, sin, cos, and atan. In this example, you configure the Simulink® models used for synthesis with native floating-point mode in single precision with different latency strategies. The strategies include maximum, minimum, zero, and custom latency values from zero to the maximum latency of the floating-point operator.

In this example, you use benchmarking data that has already been generated for the `hdl_nfp_single_ops_benchmark` model. The benchmarking data is in the `hdlcoder_synthesis_benchmark` MAT file. The MAT file contains the data for subsystems whose names start with `op_`. Each subsystem corresponds to a device under test (DUT).

Open the Simulink model.

```
addpath(fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoder', 'hdlutils', 'hdlBenchMarking'))
open_system('hdl_nfp_single_ops_benchmark')
```



To generate the synthesis result from the MAT file:

1. Make a copy of the Simulink model in a directory for which you have write permission.

2. Run the `runBenchmarkNFPSingle` function by running these commands.

```
addpath(fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoder', 'hdlutils', 'hdlBenchMarking'))
Results = runBenchmarkNFPSingle;
```

Generating synthesis benchmarks with the `hdl_nfp_single_ops_benchmark` model takes several days to finish benchmarking.

To view the MAT file that contains synthesis results.

```
load('hdlcoder_synthesis_benchmark.mat')
NFPSynthesisResults
```

```
NFPSynthesisResults = 

  struct with fields:

     Vivado: [1x1 struct]
    Quartus: [1x1 struct]
```

The MAT file contains a structure named `NFPSynthesisResults`, which has two fields. The Vivado field contains the synthesis results of the Xilinx Vivado. The Quartus field contains the synthesis results of the Intel Quartus. View synthesis results of Vivado tool.

```
NFPSynthesisResults.Vivado
```

```
ans = 

  struct with fields:

    HardwareDetails: [1x1 struct]
         MaxLatency: [82x13 table]
         MinLatency: [82x13 table]
        ZeroLatency: [82x13 table]
    TargetFrequency: 500
      CustomLatency: [369x13 table]
```

`NFPSynthesisResults.Vivado` and `NFPSynthesisResults.Quartus` contains these fields:

- `HardwareDetails` - Structure containing tool and device information
- `MaxLatency` - Table of synthesis results using the MAX latency strategy
- `MinLatency` - Table of synthesis results using the MIN latency strategy
- `ZeroLatency` - Table of synthesis results using the ZERO latency strategy
- `TargetFrequency` - Target frequency in MHz
- `CustomLatency` - Table of synthesis results using the latency values from zero to the maximum latency

The `NFPSynthesisResults.Vivado.CustomLatency` and `NFPSynthesisResults.Quartus.CustomLatency` contains list of performance and resource utilization of the floating-point operators for latency values from zero to max latency. View the complete list of performance and resource utilization results for Xilinx Vivado:

```
NFPSynthesisResults.Vivado.CustomLatency
```

```
ans = 
```

```
369x13 table

                                              Fmax     Slices    SliceRegs    LUTs    DSPs
                                             ______    ______    _________    ____    ____

    op_17x17Mul_vivado_latency0              109.05     123          96        372     1
    op_17x17Mul_vivado_latency1              121.36     134         164        400     1
    op_17x17Mul_vivado_latency2              132.59     153         260        403     1
    op_17x17Mul_vivado_latency3              272.41     158         314        375     1
    op_17x17Mul_vivado_latency4              263.99     180         357        398     1
    op_17x17Mul_vivado_latency5              259.81     145         441        385     1
    op_17x17Mul_vivado_latency6              735.29     148         448        399     1
    op_17x17Mul_vivado_latency7              656.17     155         543        408     1
    op_17x17Mul_vivado_latency8              653.59     171         673        403     1
    op_AddSub_vivado_latency0                83.326     162          96        532     0
    op_AddSub_vivado_latency1                104.91     148         127        486     0
    op_AddSub_vivado_latency2                161.34     144         208        426     0
    op_AddSub_vivado_latency3                242.54     167         281        484     0
    op_AddSub_vivado_latency4                266.95     187         338        521     0
    op_AddSub_vivado_latency5                306.94     180         387        483     0
    op_AddSub_vivado_latency6                308.45     215         434        536     0
    op_AddSub_vivado_latency7                296.38     184         501        533     0
    op_AddSub_vivado_latency8                379.94     193         562        505     0
    op_AddSub_vivado_latency9                367.78     190         561        562     0
    op_AddSub_vivado_latency10               383.29     188         646        529     0
    op_AddSub_vivado_latency11               504.03     209         667        545     0
    op_Div_vivado_latency0                   22.548     495          97       1542     0
    op_Div_vivado_latency1                   43.588     457         210       1388     0
    op_Div_vivado_latency2                   68.273     437         316       1313     0
    op_Div_vivado_latency3                   85.638     405         422       1208     0
    op_Div_vivado_latency4                   86.498     425         455       1201     0
    op_Div_vivado_latency5                   95.529     452         542       1315     0
    op_Div_vivado_latency6                   114.65     447         621       1238     0
    op_Div_vivado_latency7                   135.12     474         652       1327     0
    op_Div_vivado_latency8                   155.38     405         697       1290     0
    op_Div_vivado_latency9                   175.59     459         753       1335     0
    op_Div_vivado_latency10                  171.59     465         779       1298     0
    op_Div_vivado_latency11                  219.44     462         858       1409     0
    op_Div_vivado_latency12                  208.51     478         924       1417     0
    op_Div_vivado_latency13                  215.52     492         950       1390     0
    op_Div_vivado_latency14                  169.98     463        1012       1331     0
    op_Div_vivado_latency15                   299.4     456        1082       1353     0
    op_Div_vivado_latency16                  302.48     522        1138       1371     0
    op_Div_vivado_latency17                  317.76     509        1164       1358     0
    op_Div_vivado_latency18                  294.81     491        1248       1347     0
    op_Div_vivado_latency19                  315.26     483        1316       1344     0
    op_Div_vivado_latency20                  307.69     507        1345       1369     0
    op_Div_vivado_latency21                  303.77     505        1396       1390     0
    op_Div_vivado_latency22                  314.37     527        1447       1419     0
    op_Div_vivado_latency23                  308.26     518        1497       1444     0
    op_Div_vivado_latency24                  311.43     518        1548       1476     0
    op_Div_vivado_latency25                  318.47     509        1599       1501     0
    op_Div_vivado_latency26                   298.6     526        1650       1499     0
    op_Div_vivado_latency27                  300.48     527        1687       1553     0
    op_Div_vivado_latency28                  309.41     534        1763       1607     0
    op_Div_vivado_latency29                  341.06     553        1795       1650     0
    op_Div_vivado_latency30                  454.55     556        1862       1702     0
    op_Div_vivado_latency31                  478.01     575        1872       1753     0
```

op_Div_vivado_latency32	478.01	604	1956	1758	0
op_DivRadix4_vivado_latency0	30.254	686	97	2500	0
op_DivRadix4_vivado_latency1	47.886	559	236	1866	0
op_DivRadix4_vivado_latency2	70.373	555	367	1914	0
op_DivRadix4_vivado_latency3	64.779	559	484	1848	0
op_DivRadix4_vivado_latency4	91.058	583	544	1894	0
op_DivRadix4_vivado_latency5	121.6	581	630	1872	0
op_DivRadix4_vivado_latency6	157.73	595	701	1918	0
op_DivRadix4_vivado_latency7	159.36	606	772	1942	0
op_DivRadix4_vivado_latency8	157.43	590	820	1916	0
op_DivRadix4_vivado_latency9	236.69	607	942	1982	0
op_DivRadix4_vivado_latency10	232.07	630	969	2013	0
op_DivRadix4_vivado_latency11	228.99	625	1025	2030	0
op_DivRadix4_vivado_latency12	241.95	629	1107	2015	0
op_DivRadix4_vivado_latency13	241.49	623	1187	1988	0
op_DivRadix4_vivado_latency14	240.5	619	1267	1958	0
op_DivRadix4_vivado_latency15	239.41	613	1293	1973	0
op_DivRadix4_vivado_latency16	428.27	658	1459	2061	0
op_DivRadix4_vivado_latency17	444.05	663	1515	2104	0
op_DivRadix4_vivado_latency18	431.22	667	1583	2105	0
op_DivRadix4_vivado_latency19	447.23	670	1609	2078	0
op_DivRadix4_vivado_latency20	429	677	1693	2063	0
op_Div_Denorm_on_vivado_latency0	21.021	578	100	1819	0
op_Div_Denorm_on_vivado_latency1	42.541	545	213	1741	0
op_Div_Denorm_on_vivado_latency2	66.339	455	324	1451	0
op_Div_Denorm_on_vivado_latency3	85.273	463	433	1468	0
op_Div_Denorm_on_vivado_latency4	87.958	533	466	1460	0
op_Div_Denorm_on_vivado_latency5	95.914	492	506	1561	0
op_Div_Denorm_on_vivado_latency6	114.69	519	633	1480	0
op_Div_Denorm_on_vivado_latency7	136.56	500	664	1542	0
op_Div_Denorm_on_vivado_latency8	111.04	528	716	1558	0
op_Div_Denorm_on_vivado_latency9	170.79	520	764	1540	0
op_Div_Denorm_on_vivado_latency10	175.38	514	790	1534	0
op_Div_Denorm_on_vivado_latency11	201.13	560	869	1656	0
op_Div_Denorm_on_vivado_latency12	218.72	544	944	1644	0
op_Div_Denorm_on_vivado_latency13	215.84	555	970	1599	0
op_Div_Denorm_on_vivado_latency14	169.64	499	1032	1569	0
op_Div_Denorm_on_vivado_latency15	236.8	563	1102	1627	0
op_Div_Denorm_on_vivado_latency16	259.88	536	1158	1629	0
op_Div_Denorm_on_vivado_latency17	287.52	551	1184	1607	0
op_Div_Denorm_on_vivado_latency18	306.47	571	1268	1606	0
op_Div_Denorm_on_vivado_latency19	315.06	565	1336	1570	0
op_Div_Denorm_on_vivado_latency20	304.04	564	1365	1602	0
op_Div_Denorm_on_vivado_latency21	318.88	576	1416	1625	0
op_Div_Denorm_on_vivado_latency22	315.56	576	1467	1649	0
op_Div_Denorm_on_vivado_latency23	310.27	591	1517	1674	0
op_Div_Denorm_on_vivado_latency24	307.03	563	1568	1702	0
op_Div_Denorm_on_vivado_latency25	316.56	582	1619	1730	0
op_Div_Denorm_on_vivado_latency26	331.13	599	1670	1731	0
op_Div_Denorm_on_vivado_latency27	312.79	574	1697	1807	0
op_Div_Denorm_on_vivado_latency28	311.24	559	1773	1605	0
op_Div_Denorm_on_vivado_latency29	298.42	546	1815	1610	0
op_Div_Denorm_on_vivado_latency30	273.45	662	1882	1955	0
op_Div_Denorm_on_vivado_latency31	467.51	663	1892	2020	0
op_Div_Denorm_on_vivado_latency32	475.29	639	1976	1927	0
op_Fixed2Single_vivado_latency0	301.93	36	38	90	0
op_Fixed2Single_vivado_latency1	180.77	48	39	174	0
op_Fixed2Single_vivado_latency2	315.36	69	81	189	0

```
    op_Fixed2Single_vivado_latency3             319.69      70      104      205    0
    op_Fixed2Single_vivado_latency4             325.63      69      115      210    0
    op_Fixed2Single_vivado_latency5             455.79      69      155      214    0
    op_Fixed2Single_vivado_latency6             506.07      87      180      209    0
    op_Floor_vivado_latency0                    311.72      54       64      164    0
    op_Floor_vivado_latency1                    310.17      61      125      183    0
    op_Floor_vivado_latency2                    198.85      65      115      196    0
    op_Floor_vivado_latency3                     422.3      63      158      163    0
    op_Floor_vivado_latency4                       456      78      243      192    0
    op_Floor_vivado_latency5                    552.18      78      308      208    0
    op_Floor_vivado_latency6                    552.18      78      308      208    0
    op_GainPow2_vivado_latency0                 766.87      12       64       35    0
    op_GainPow2_vivado_latency1                 665.34      14       64       14    0
    op_GainPow2_vivado_latency2                 454.13      25       97       18    0
    op_GainPow2_Denorm_on_vivado_latency0       178.48      89       64      287    0
    op_GainPow2_Denorm_on_vivado_latency1       204.16      71       97      221    0
    op_GainPow2_Denorm_on_vivado_latency2       182.58     102       97      295    0
    op_Mul_FullMul_vivado_latency0              116.78      74       96      202    2
    op_Mul_FullMul_vivado_latency1              173.01      68      103      194    2
    op_Mul_FullMul_vivado_latency2              196.85      72      119      201    2
    op_Mul_FullMul_vivado_latency3              239.18      84      155      219    2
    op_Mul_FullMul_vivado_latency4              259.81      78      191      206    2
    op_Mul_FullMul_vivado_latency5              270.93     109      251      208    2
    op_Mul_FullMul_vivado_latency6              580.72      86      269      200    2
    op_Mul_FullMul_vivado_latency7               577.7      95      353      231    2
    op_Mul_FullMul_vivado_latency8              535.05     107      394      220    2
    op_Mul_FullMul_Denorm_on_vivado_latency0    68.259     217       96      780    2
    op_Mul_FullMul_Denorm_on_vivado_latency1    68.078     207      102      720    2
    op_Mul_FullMul_Denorm_on_vivado_latency2    81.706     198      154      681    2
    op_Mul_FullMul_Denorm_on_vivado_latency3    172.59     221      190      724    2
    op_Mul_FullMul_Denorm_on_vivado_latency4    170.65     245      218      736    2
    op_Mul_FullMul_Denorm_on_vivado_latency5    258.53     287      285      772    2
    op_Mul_FullMul_Denorm_on_vivado_latency6    171.79     278      320      720    2
    op_Mul_FullMul_Denorm_on_vivado_latency7    273.45     297      387      767    2
    op_Mul_FullMul_Denorm_on_vivado_latency8    286.45     285      398      769    2
    op_Mul_NoManMul_vivado_latency0             115.21     263       96      907    0
    op_Mul_NoManMul_vivado_latency1              108.3     306      302     1038    0
    op_Mul_NoManMul_vivado_latency2             164.72     276      273      912    0
    op_Mul_NoManMul_vivado_latency3             255.49     275      309      880    0
    op_Mul_NoManMul_vivado_latency4             267.74     301      444      961    0
    op_Mul_NoManMul_vivado_latency5             313.77     303      504      950    0
    op_Mul_NoManMul_vivado_latency6              378.5     303      518      883    0
    op_Mul_NoManMul_vivado_latency7             376.36     298      630      918    0
    op_Mul_NoManMul_vivado_latency8             548.25     327      850      947    0
    op_Mul_NoManMul_Denorm_on_vivado_latency0   65.656     422       96     1488    0
    op_Mul_NoManMul_Denorm_on_vivado_latency1   50.249     454      301     1546    0
    op_Mul_NoManMul_Denorm_on_vivado_latency2   70.947     428      227     1442    0
    op_Mul_NoManMul_Denorm_on_vivado_latency3   171.53     435      263     1462    0
    op_Mul_NoManMul_Denorm_on_vivado_latency4   175.44     464      397     1475    0
    op_Mul_NoManMul_Denorm_on_vivado_latency5   208.55     445      464     1459    0
    op_Mul_NoManMul_Denorm_on_vivado_latency6   248.76     438      479     1413    0
    op_Mul_NoManMul_Denorm_on_vivado_latency7   275.41     446      566     1401    0
    op_Mul_NoManMul_Denorm_on_vivado_latency8   264.83     516      793     1579    0
    op_Mul_PartManMul_vivado_latency0           109.05     123       96      372    1
    op_Mul_PartManMul_vivado_latency1           121.36     134      164      400    1
    op_Mul_PartManMul_vivado_latency2           132.59     153      260      403    1
    op_Mul_PartManMul_vivado_latency3           272.41     158      314      375    1
    op_Mul_PartManMul_vivado_latency4           263.99     180      357      398    1
```

```
op_Mul_PartManMul_vivado_latency5             259.81     145      441      385     1
op_Mul_PartManMul_vivado_latency6             735.29     148      448      399     1
op_Mul_PartManMul_vivado_latency7             656.17     155      543      408     1
op_Mul_PartManMul_vivado_latency8             613.12     174      673      404     1
op_Mul_PartManMul_Denorm_on_vivado_latency0   63.654     274       96      953     1
op_Mul_PartManMul_Denorm_on_vivado_latency1   57.571     288      163      929     1
op_Mul_PartManMul_Denorm_on_vivado_latency2   81.407     284      210      803     1
op_Mul_PartManMul_Denorm_on_vivado_latency3   166.58     285      246      876     1
op_Mul_PartManMul_Denorm_on_vivado_latency4   168.35     297      330      858     1
op_Mul_PartManMul_Denorm_on_vivado_latency5   258.67     291      397      853     1
op_Mul_PartManMul_Denorm_on_vivado_latency6   191.53     291      429      835     1
op_Mul_PartManMul_Denorm_on_vivado_latency7   278.01     311      499      824     1
op_Mul_PartManMul_Denorm_on_vivado_latency8   288.68     349      588      958     1
op_RSqrt_vivado_latency0                      15.533    1247       68     4426     0
op_RSqrt_vivado_latency1                      18.349    1188      183     4209     0
op_RSqrt_vivado_latency2                      28.384    1194      324     4179     0
op_RSqrt_vivado_latency3                      56.548    1126      466     3834     0
op_RSqrt_vivado_latency4                       59.51    1123      496     3898     0
op_RSqrt_vivado_latency5                      69.372    1095      553     3877     0
op_RSqrt_vivado_latency6                      84.048    1154      661     3926     0
op_RSqrt_vivado_latency7                      89.855    1263      748     4213     0
op_RSqrt_vivado_latency8                       83.57    1192      835     3997     0
op_RSqrt_vivado_latency9                      115.93    1159      916     3977     0
op_RSqrt_vivado_latency10                     130.26    1171      993     3915     0
op_RSqrt_vivado_latency11                     163.85    1211     1138     3991     0
op_RSqrt_vivado_latency12                     167.62    1213     1174     4018     0
op_RSqrt_vivado_latency13                     178.73    1186     1252     3954     0
op_RSqrt_vivado_latency14                     138.83    1140     1338     3785     0
op_RSqrt_vivado_latency15                     211.95    1146     1594     4036     0
op_RSqrt_vivado_latency16                     222.22    1200     1578     3967     0
op_RSqrt_vivado_latency17                     243.01    1178     1648     4014     0
op_RSqrt_vivado_latency18                     225.78    1235     1681     4020     0
op_RSqrt_vivado_latency19                     205.85    1189     1815     3993     0
op_RSqrt_vivado_latency20                        228    1257     1925     3971     0
op_RSqrt_vivado_latency21                     204.79    1206     2017     3946     0
op_RSqrt_vivado_latency22                     250.19    1213     2135     3928     0
op_RSqrt_vivado_latency23                     254.45    1165     2239     3901     0
op_RSqrt_vivado_latency24                     242.48    1170     2353     3879     0
op_RSqrt_vivado_latency25                     224.22    1210     2487     3853     0
op_RSqrt_vivado_latency26                     254.45    1232     2547     4130     0
op_RSqrt_vivado_latency27                      247.4    1243     2679     4168     0
op_RSqrt_vivado_latency28                     259.47    1233     2791     4184     0
op_RSqrt_vivado_latency29                     256.87    1296     2913     4204     0
op_RSqrt_vivado_latency30                     149.86    1313     3023     4221     0
op_RSqrt_Denorm_on_vivado_latency0            16.817    1218       64     4233     0
op_RSqrt_Denorm_on_vivado_latency1            19.525    1173      181     4232     0
op_RSqrt_Denorm_on_vivado_latency2            27.615    1212      323     4298     0
op_RSqrt_Denorm_on_vivado_latency3            53.813    1178      466     4024     0
op_RSqrt_Denorm_on_vivado_latency4            59.067    1135      496     4030     0
op_RSqrt_Denorm_on_vivado_latency5            70.512    1149      507     4006     0
op_RSqrt_Denorm_on_vivado_latency6              75.7    1163      661     4038     0
op_RSqrt_Denorm_on_vivado_latency7             89.63    1168      748     4068     0
op_RSqrt_Denorm_on_vivado_latency8            82.325    1191      813     4082     0
op_RSqrt_Denorm_on_vivado_latency9            123.21    1250      916     4096     0
op_RSqrt_Denorm_on_vivado_latency10           121.02    1199      993     4027     0
op_RSqrt_Denorm_on_vivado_latency11           153.14    1117     1138     3886     0
op_RSqrt_Denorm_on_vivado_latency12           150.85    1248     1174     4131     0
op_RSqrt_Denorm_on_vivado_latency13           178.54    1193     1252     4028     0
```

```
op_RSqrt_Denorm_on_vivado_latency14          136.56     1144       1338       3868      0
op_RSqrt_Denorm_on_vivado_latency15          224.16     1239       1572       4087      0
op_RSqrt_Denorm_on_vivado_latency16          208.03     1189       1578       4044      0
op_RSqrt_Denorm_on_vivado_latency17          235.79     1214       1648       4102      0
op_RSqrt_Denorm_on_vivado_latency18          242.48     1234       1682       4066      0
op_RSqrt_Denorm_on_vivado_latency19          214.82     1266       1816       4041      0
op_RSqrt_Denorm_on_vivado_latency20          246.06     1168       1926       4015      0
op_RSqrt_Denorm_on_vivado_latency21          215.42     1236       2018       3992      0
op_RSqrt_Denorm_on_vivado_latency22          241.72     1216       2136       3977      0
op_RSqrt_Denorm_on_vivado_latency23          246.12     1158       2240       3960      0
op_RSqrt_Denorm_on_vivado_latency24          215.61     1179       2354       3929      0
op_RSqrt_Denorm_on_vivado_latency25          250.63     1197       2488       3902      0
op_RSqrt_Denorm_on_vivado_latency26             249     1269       2548       4198      0
op_RSqrt_Denorm_on_vivado_latency27          236.29     1287       2680       4228      0
op_RSqrt_Denorm_on_vivado_latency28          234.91     1268       2792       4242      0
op_RSqrt_Denorm_on_vivado_latency29          253.23     1274       2914       4257      0
op_RSqrt_Denorm_on_vivado_latency30          265.75     1303       3024       4266      0
op_Relop_vivado_latency0                     575.37       27         69         39      0
op_Relop_vivado_latency1                     342.94       29        132         36      0
op_Relop_vivado_latency2                     633.71       31        134         41      0
op_Relop_vivado_latency3                     770.42       33        142         37      0
op_Round_vivado_latency0                     313.19       61         64        186      0
op_Round_vivado_latency1                     337.27       71        127        207      0
op_Round_vivado_latency2                     163.77       72        116        226      0
op_Round_vivado_latency3                     371.89       67        160        179      0
op_Round_vivado_latency4                      391.7       88        246        216      0
op_Round_vivado_latency5                     577.03       91        311        219      0
op_Round_vivado_latency6                     577.03       91        311        219      0
op_Single2Fixed_vivado_latency0              310.46      101         64        339      0
op_Single2Fixed_vivado_latency1              153.26      152         69        495      0
op_Single2Fixed_vivado_latency2              336.02       83         99        248      0
op_Single2Fixed_vivado_latency3              413.05       85        156        277      0
op_Single2Fixed_vivado_latency4              367.38       96        188        279      0
op_Single2Fixed_vivado_latency5              400.48      102        221        311      0
op_Single2Fixed_vivado_latency6              384.91      111        210        346      0
op_Sqrt_vivado_latency0                      31.903      222         65        676      0
op_Sqrt_vivado_latency1                      55.353      235        145        703      0
op_Sqrt_vivado_latency2                      79.917      255        222        730      0
op_Sqrt_vivado_latency3                      112.17      247        299        693      0
op_Sqrt_vivado_latency4                      90.506      282        334        738      0
op_Sqrt_vivado_latency5                      105.43      234        300        657      0
op_Sqrt_vivado_latency6                      129.52      242        426        657      0
op_Sqrt_vivado_latency7                      147.97      260        431        722      0
op_Sqrt_vivado_latency8                      186.05      254        391        653      0
op_Sqrt_vivado_latency9                      190.01      270        477        718      0
op_Sqrt_vivado_latency10                     193.61      265        495        725      0
op_Sqrt_vivado_latency11                     211.64      271        541        825      0
op_Sqrt_vivado_latency12                     223.36      294        574        830      0
op_Sqrt_vivado_latency13                     247.71      256        539        694      0
op_Sqrt_vivado_latency14                     308.26      270        637        750      0
op_Sqrt_vivado_latency15                     311.82      273        679        739      0
op_Sqrt_vivado_latency16                     280.43      290        695        762      0
op_Sqrt_vivado_latency17                     313.87      270        741        753      0
op_Sqrt_vivado_latency18                     316.26      283        773        769      0
op_Sqrt_vivado_latency19                     327.76      291        783        805      0
op_Sqrt_vivado_latency20                     335.68      280        816        799      0
op_Sqrt_vivado_latency21                     331.56      297        833        809      0
op_Sqrt_vivado_latency22                     339.33      309        893        839      0
```

op_Sqrt_vivado_latency23	372.72	296	937	862	0
op_Sqrt_vivado_latency24	341.18	314	932	855	0
op_Sqrt_vivado_latency25	335.35	312	953	868	0
op_Sqrt_vivado_latency26	355.49	312	988	884	0
op_Sqrt_vivado_latency27	362.71	314	1014	897	0
op_Sqrt_vivado_latency28	421.23	326	1050	913	0
op_Sqrt_Denorm_on_vivado_latency0	31.3	262	65	759	0
op_Sqrt_Denorm_on_vivado_latency1	58.976	250	123	743	0
op_Sqrt_Denorm_on_vivado_latency2	84.104	245	178	715	0
op_Sqrt_Denorm_on_vivado_latency3	112.56	248	233	726	0
op_Sqrt_Denorm_on_vivado_latency4	99.8	262	267	672	0
op_Sqrt_Denorm_on_vivado_latency5	100.5	258	304	738	0
op_Sqrt_Denorm_on_vivado_latency6	130.16	291	425	739	0
op_Sqrt_Denorm_on_vivado_latency7	143.14	268	430	807	0
op_Sqrt_Denorm_on_vivado_latency8	184.4	281	391	777	0
op_Sqrt_Denorm_on_vivado_latency9	188.79	304	482	806	0
op_Sqrt_Denorm_on_vivado_latency10	187.41	286	509	782	0
op_Sqrt_Denorm_on_vivado_latency11	236.52	305	552	838	0
op_Sqrt_Denorm_on_vivado_latency12	223.51	310	584	905	0
op_Sqrt_Denorm_on_vivado_latency13	214.87	297	539	815	0
op_Sqrt_Denorm_on_vivado_latency14	316.56	311	654	808	0
op_Sqrt_Denorm_on_vivado_latency15	300.48	305	696	802	0
op_Sqrt_Denorm_on_vivado_latency16	274.73	275	695	840	0
op_Sqrt_Denorm_on_vivado_latency17	320.1	314	758	816	0
op_Sqrt_Denorm_on_vivado_latency18	306.18	302	790	836	0
op_Sqrt_Denorm_on_vivado_latency19	291.89	314	800	846	0
op_Sqrt_Denorm_on_vivado_latency20	276.7	325	833	880	0
op_Sqrt_Denorm_on_vivado_latency21	272.33	331	854	888	0
op_Sqrt_Denorm_on_vivado_latency22	341.41	320	910	903	0
op_Sqrt_Denorm_on_vivado_latency23	363.64	338	958	923	0
op_Sqrt_Denorm_on_vivado_latency24	354.86	333	951	926	0
op_Sqrt_Denorm_on_vivado_latency25	339.1	325	974	929	0
op_Sqrt_Denorm_on_vivado_latency26	343.29	325	1009	953	0
op_Sqrt_Denorm_on_vivado_latency27	366.3	333	1035	963	0
op_Sqrt_Denorm_on_vivado_latency28	450.86	342	1071	978	0
op_ceiling_vivado_latency0	293.43	89	64	304	0
op_ceiling_vivado_latency1	94.742	137	68	477	0
op_ceiling_vivado_latency2	301.93	109	99	355	0
op_ceiling_vivado_latency3	353.36	110	157	363	0
op_ceiling_vivado_latency4	364.56	115	189	362	0
op_ceiling_vivado_latency5	360.23	121	222	378	0
op_ceiling_vivado_latency6	364.56	129	211	403	0
op_convergent_vivado_latency0	293.51	106	64	337	0
op_convergent_vivado_latency1	130.58	145	69	495	0
op_convergent_vivado_latency2	333	89	99	286	0
op_convergent_vivado_latency3	408.33	85	156	279	0
op_convergent_vivado_latency4	395.73	91	188	278	0
op_convergent_vivado_latency5	401.77	102	221	310	0
op_convergent_vivado_latency6	402.9	107	210	346	0
op_floor_vivado_latency0	293.51	106	64	337	0
op_floor_vivado_latency1	130.58	145	69	495	0
op_floor_vivado_latency2	333	89	99	286	0
op_floor_vivado_latency3	408.33	85	156	279	0
op_floor_vivado_latency4	395.73	91	188	278	0
op_floor_vivado_latency5	401.77	102	221	310	0
op_floor_vivado_latency6	402.9	107	210	346	0
op_nearest_vivado_latency0	293.51	106	64	337	0
op_nearest_vivado_latency1	130.58	145	69	495	0

```
    op_nearest_vivado_latency2                      333       89        99        286     0
    op_nearest_vivado_latency3                   408.33       85       156        279     0
    op_nearest_vivado_latency4                   395.73       91       188        278     0
    op_nearest_vivado_latency5                   401.77      102       221        310     0
    op_nearest_vivado_latency6                    402.9      107       210        346     0
    op_round_vivado_latency0                     293.51      106        64        337     0
    op_round_vivado_latency1                     130.58      145        69        495     0
    op_round_vivado_latency2                        333       89        99        286     0
    op_round_vivado_latency3                     408.33       85       156        279     0
    op_round_vivado_latency4                     395.73       91       188        278     0
    op_round_vivado_latency5                     401.77      102       221        310     0
    op_round_vivado_latency6                      402.9      107       210        346     0
    op_simplest_vivado_latency0                  293.51      106        64        337     0
    op_simplest_vivado_latency1                  130.58      145        69        495     0
    op_simplest_vivado_latency2                     333       89        99        286     0
    op_simplest_vivado_latency3                  408.33       85       156        279     0
    op_simplest_vivado_latency4                  395.73       91       188        278     0
    op_simplest_vivado_latency5                  401.77      102       221        310     0
    op_simplest_vivado_latency6                   402.9      107       210        346     0
    op_zero_vivado_latency0                      293.51      106        64        337     0
    op_zero_vivado_latency1                      130.58      145        69        495     0
    op_zero_vivado_latency2                         333       89        99        286     0
    op_zero_vivado_latency3                      408.33       85       156        279     0
    op_zero_vivado_latency4                      395.73       91       188        278     0
    op_zero_vivado_latency5                      401.77      102       221        310     0
    op_zero_vivado_latency6                       402.9      107       210        346     0
```

Generate Synthesis Results

To generate synthesis results for other Simulink models or different settings, you can pass input arguments to the `runBenchmarkNFPSingle` function. Specify name-value arguments, where `name` is the argument name and `value` is the corresponding value.

- `ModelName` - Simulink model name, specified as a character vector. The default value is `hdl_nfp_single_ops_benchmark`. The name of the subsystems in a model must start with `op_`.
- `TargetFrequency` - Target frequency in MHz. The default value is 500 MHz.
- `HardwareTool` - Hardware applications for generating results, specified as a cell array of character vectors. The default value is {|'Vivado','Quartus'|}.
- `LatencyStrategy` - Latency strategy, specified as a cell array of character vectors. The default value is {'Min','Max','Zero','Custom'}.
- `VivadoHardwareDetails` or `QuartusHardwareDetails` - Hardware details, specified as a structure containing these five fields:

`Tool` - Synthesis tool name, specified as character vector.

`ChipFamily` - Chip family, specified as character vector.

`DeviceName` - Device name, specified as character vector.

`PackageName` - Package name, specified as character vector.

`SpeedValue` - Hardware speed, specified as character vector.

For example, set the hardware details for Xilinx Vivado and Altera Quartus II.

```
vivado_hardware_details.Tool = 'Xilinx Vivado';
vivado_hardware_details.ChipFamily = 'Virtex7';
vivado_hardware_details.DeviceName = 'xc7v2000t';
vivado_hardware_details.PackageName = 'fhg1761';
vivado_hardware_details.SpeedValue = '-2';
vivado_hardware_details =

     struct with fields:

              Tool: 'Xilinx Vivado'
        ChipFamily: 'Virtex7'
        DeviceName: 'xc7v2000t'
       PackageName: 'fhg1761'
        SpeedValue: '-2'

quartus_hardware_details.Tool = 'Altera QUARTUS II';
quartus_hardware_details.ChipFamily = 'Stratix V';
quartus_hardware_details.DeviceName = '5SGSMD4E1H29C1';
quartus_hardware_details.PackageName = '';
quartus_hardware_details.SpeedValue = '';
quartus_hardware_details =

     struct with fields:

              Tool: 'Altera QUARTUS II'
        ChipFamily: 'Stratix V'
        DeviceName: '5SGSMD4E1H29C1'
       PackageName: ''
        SpeedValue: ''
```

Call the `runNFPBenchmarkResults` function for the specified hardware.

```
runNFPBenchmarkResults = runBenchmarkNFPSingle('ModelName','myModel',...
                                               'TargetFrequency',500,...
                                               'VivadoHardwareDetails', vivado_hardware_details,
                                               'QuartusHardwareDetails', quartus_hardware_detail
                                               'HardwareTool',{'Vivado','Quartus'},...
                                               'LatencyStrategy',{'Min','Max','Zero','Custom'});
```

Customize Synthesis Results

Generating synthesis benchmarks with the `hdl_nfp_single_ops_benchmark` model takes several days to finish benchmarking. To generate benchmarks for only a subset of the operators, make a copy of the model and delete any DUTs for which you do not require data.

To generate benchmark synthesis results with your own model using the `runBenchmarkNFPSingle` function, ensure that:

- Each DUT or subsystem name starts with `op_`.
- For custom latency benchmarking, each DUT name contains characters required to get the list of operators that support custom latency. Please see `purgeDutList` function in `runBenchmarkNFPSingle_Vivado` and `runBenchmarkNFPSingle_Quartus`. To view the list of blocks that supports custom latency, see “Latency Values of Floating-Point Operators” on page 14-99.

Related Links

- “Simulink Blocks Supported by Using Native Floating Point” on page 14-137
- “Numeric Considerations for Native Floating-Point” on page 14-92
- “Latency Considerations with Native Floating Point” on page 14-104

Supported Data Types and Scope

In this section...

"Supported Data Types" on page 14-155
"Unsupported Data Types" on page 14-156
"Scope for Variables" on page 14-156

Supported Data Types

HDL Coder supports the following subset of MATLAB data types.

Types	Supported Data Types	Restrictions
Integer	• `uint8`, `uint16`, `uint32`, `uint64` • `int8`, `int16`, `int32`, `int64`	In Simulink, MATLAB Function block ports must use numeric types `sfix64` or `ufix64` for 64-bit data.
Real	• `double` • `single`	HDL code generated with `double` or `single` data types can be used for simulation, but is not synthesizable. When you have floating-point data types, to generate synthesizable HDL code, use: • HDL Coder native floating-point when you want to deploy the generated code on any generic ASIC or FPGA. To learn more, see "Getting Started with HDL Coder Native Floating-Point Support" on page 14-88. • FPGA floating-point target libraries when you want to map the Simulink model to an Intel or Xilinx FPGA. To learn more, see "Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries" on page 29-19.
Character	`char`	
Logical	`logical`	
Fixed point	• Scaled (binary point only) fixed-point numbers • Custom integers (zero binary point)	Fixed-point numbers with slope (not equal to 1.0) and bias (not equal to 0.0) are not supported. Maximum word size for fixed-point numbers is 128 bits.
Vectors	• unordered `{N}` • row `{1, N}` • column `{N, 1}`	The maximum number of vector elements allowed is 2^32. Before a variable is subscripted, it must be fully defined.

Types	Supported Data Types	Restrictions
Matrices	`{N, M}`	Matrices are supported in the body of the design algorithm and as inputs to the top-level design function. Matrices are not supported with the following HDL workflows: • FPGA-in-the-Loop • IP Core Generation Cosimulation workflows that use Simulink and SystemVerilog now support vector data types.
Structures	`struct`	Arrays of structures are not supported. For the IP Core Generation workflow, structures are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.
Enumerations	`enumeration`	If your target language is Verilog, all enumeration member names must be unique within the design. Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods: • IP Core Generation workflow • FPGA-in-the-Loop Enumerations are supported for Simulink workflows and only for Verilog and SystemVerilog. Enumerations are not supported for VHDL.

Unsupported Data Types

The following data types are not supported:

- Cell array
- `Inf`

Scope for Variables

Global variables are not supported for HDL code generation.

Supported Synthesizable RTL Constructs and keywords in HDL Coder

Supported VHDL Constructs

The table lists the synthesizable VHDL constructs in HDL Coder .

Entity and Architecture Declaration

VHDL Constructs	Supported?	Comments
Entity declaration	Yes	–
Architecture declaration	Yes	–
Entity parameter port list	Yes	–
Entity signal declaration	Yes	–
Generic clause	Yes	–

Package and Library Declaration

VHDL Constructs	Supported?	Comments
Package declaration	No	–
Type declaration	No	–
Subtype declaration	No	–
Constant declaration	Yes	–
Variable declaration	Yes	–
Use clause	No	–
Library declaration	No	–

Component and Configuration Declaration

VHDL Constructs	Supported?	Comments
Configuration declaration	No	–
Configuration specification	No	–
Component declaration	Yes	–
Component instantiation	Yes	Supports ordered or named port map. Mixed and generic port mapping is not supported.
Generic clause	Yes	–

Data Types and Vectors

VHDL Constructs	Supported?	Comments
Integer declaration	Yes	–
Real declaration	No	–
String declaration	No	–
Bit_vector	Yes	–
Enumerated	Yes	–
Std_logic	Yes	Values other than 0 and 1 (U, X, Z, W, L, H) are not supported.
Vector declaration	Yes	–

Identifiers and Comments

VHDL Constructs	Supported?	Comments
Numbers (based, normal)	Yes	Based literals support bases 2, 8, 10, and 16.
Identifiers	Yes	–
Standard package functions	Yes	Supports signed, unsigned, to_std_logic_vector, resize, and to_integer.
Attribute instances	No	–
Attribute declaration	No	–
Records	No	–
Comments	Yes	–

Assignments

VHDL Constructs	Supported?	Comments
Assignment statements (signal, variable)	Yes	–
Selected signal assignment	Yes	–

Operators

VHDL Constructs	Supported?	Comments
Arithmetic operators (+,-,*, mod, rem)	Yes	–
Relational operators (<, >, <<. >>)	Yes	–
Unary operators (+,-)	Yes	–
Logical operators (and, or, xor, nand, and nor)	Yes	–
Absolute and exponential operators (abs, **)	Yes	–
Gates	Yes	–

Conditional and Looping Statements

VHDL Constructs	Supported?	Comments
If-else statement	Yes	–
Case statement	Yes	–
Conditional operators (?:)	Yes	–
Assertion statements	No	–
For loop	No	–
Loop statements	No	–
Generate statement	No	–

Process Statements and Procedure Definitions

VHDL Constructs	Supported?	Comments
Process statement	Yes	–
Functions	No	–
Blocks	No	–
Procedure definitions	No	–
Function calls	Yes	Recursive function calls are not supported.

Event Control Statements

VHDL Constructs	Supported?	Comments
Event control statements	Yes	–
Waveform condition	No	–
Wait statement	Yes	–
Exit statement	No	–
Null statement	No	–
Return statement	No	–

Supported Verilog Constructs

The table lists the supported synthesizable Verilog constructs in HDL Coder.

Module Definition and Instantiation

Verilog Constructs	Supported?	Comments
Library declaration	No	–
Configuration declaration	No	–
Module declaration	Yes	–
Module parameter port list	Yes	–
Port declarations	Yes	–
Module without ports	No	–
Local parameter declaration	Yes	–
Parameter declaration	Yes	--
Module instatiation	Yes	--

Data Types and Vectors

Verilog Constructs	Supported?	Comments
Net declaration (Wire, Supply0, Supply1)	Yes	–
Reg declaration	Yes	–
Integer declaration	Yes	–
Real declaration	No	–
String declaration	No	–
Vector declaration	Yes	–
Array support	Yes	Array indexing and multidimensional arrays are not supported.

Identifiers and Comments

Verilog Constructs	Supported?	Comments
Lexical tokens (Whitespace, operator, comment)	Yes	–
Numbers (Decimal, Binary, Hexadecimal, and Octal)	Yes	Does not support numbers such as x and z.
Identifiers (Simple, Escaped)	No	–
Compiler directives (`define,`undef, `ifndef, `else if)	Yes	–
System Functions ($signed, $unsigned)	Yes	–
Attribute instances	No	–
Comments	No	–

Assignments

Verilog Constructs	Supported?	Comments
Continuous assignment	Yes	–
Procedural assignment (Always block)	Yes	–
Blocking assignment	Yes	–
Non-blocking assignment	Yes	–

Operators

<table>
<tr><th>Verilog Constructs</th><th>Supported?</th><th>Comments</th></tr>
<tr><td>Arithmetic operators (+, -, *, **, /, <<<, >>>)</td><td>Yes</td><td>–</td></tr>
<tr><td>Reduction operators (&, ~&, |, ~|, ^, ~^, or ^~)</td><td>Yes</td><td>–</td></tr>
<tr><td>Logical operators (<<, >>, !, &&, | |, ==, !=)</td><td>Yes</td><td>–</td></tr>
<tr><td>Relational operators (>, <, >=, <=, ==, !=)</td><td>Yes</td><td>–</td></tr>
<tr><td>Bitwise operators (~, &, |, ^, ~^, ^~)</td><td>Yes</td><td>–</td></tr>
<tr><td>Unary operators (+, -)</td><td>Yes</td><td>–</td></tr>
<tr><td>Conditional operators (?:)</td><td>Yes</td><td>–</td></tr>
<tr><td>Concatenation</td><td>Yes</td><td>–</td></tr>
<tr><td>Bit Select</td><td>Yes</td><td>–</td></tr>
</table>

Conditional and Looping Statements

Verilog Constructs	Supported?	Comments
If-else statement	Yes	–
Case statement	Yes	–
Conditional operators (?:)	Yes	–
For loop	No	–
Loop Generate construct	No	–
Conditional Generate construct	No	–
Generate region	No	–
Genvar declaration	No	–

Procedural Blocks and Events

Verilog Constructs	Supported?	Comments
Initial construct (ROM modeling)	No	–
Always construct	Yes	–
Task declaration	No	–
Function declaration	No	–
Sequential blocks	Yes	–
Block declarations	Yes	–
Event control statements	Yes	–
Function calls	Yes	Does not supports recursive function calls.
Task enable	No	–

Others

Verilog Constructs	Supported?	Comments
Gate instantiation	No	–
Drive strength	No	–
Delays	No	–
Specparams	No	–
Specify block	No	–
Semantic verification (unused ports, correct module instantiation)	Yes	–
Clock bundle identification	Yes	Multiple sample rates and multiple clock signals are not supported.
Register inference	Yes	–
RAM inference	Yes	Supports Simple Dual Port RAM. Other types of RAM are not supported.
ROM inference	No	–
Counter inference	No	–

Supported VHDL Keywords

The table lists the supported synthesizable VHDL keywords in HDL Coder.

`abs`	`access`	`after`	`alias`	`all`
`and`	`architecture`	`array`	`assert`	`attribute`
`begin`	`block`	`body`	`buffer`	`bus`
`case`	`component`	`configuration`	`constant`	`disconnect`
`downto`	`else`	`elsif`	`end`	`entity`
`exit`	`file`	`for`	`function`	`generate`
`generic`	`group`	`guarded`	`if`	`impure`
`in`	`inertial`	`inout`	`is`	`label`
`library`	`linkage`	`literal`	`loop`	`map`
`mod`	`nand`	`new`	`next`	`nor`
`not`	`null`	`of`	`on`	`open`
`or`	`others`	`out`	`package`	`port`
`postponed`	`procedure`	`process`	`pure`	`range`
`record`	`register`	`reject`	`rem`	`report`
`return`	`rol`	`ror`	`select`	`severity`
`signal`	`shared`	`sla`	`sll`	`sra`

srl	subtype	then	to	transport
type	unaffected	units	until	use
variable	wait	when	while	with
xnor	xor			

Supported Verilog Keywords

The table lists the supported synthesizable Verilog keywords supported in HDL Coder.

always	and	assign	automatic	begin
buf	bufif0	bufif1	case	casex
casez	cell	cmos	config	deassign
default	defparam	design	disable	edge
else	end	endcase	endconfig	endfunction
endgenerate	endmodule	endprimitive	endspecify	endtable
endtask	event	for	force	forever
fork	function	generate	genvar	highz0
highz1	if	ifnone	incdir	include
initial	inout	input	instance	integer
join	large	liblist	library	localparam
macromodule	medium	module	nand	negedge
nmos	nor	noshowcancelled	not	notif0
notif1	or	output	parameter	pmos
posedge	primitive	pull0	pull1	pulldown
pullup	pulsestyle_onevent	pulsestyle_ondetect	rcoms	real
realtime	reg	release	repeat	rnmos
rpmos	rtran	rtranif0	rtranif1	scalared
showcancelled	signed	small	specify	specparam
strong0	strong1	supply0	supply1	table
task	time	tran	tranif0	tranif1
tri	tri0	tri1	triand	trior
trireg	unsigned	use	vectored	wait
wand	weak0	weak1	while	wire
wor	xnor	xor		

Import Verilog Code and Generate Simulink Model

In this section...
“HDL Import” on page 14-165
“HDL Import Requirements” on page 14-165
“Import HDL Code” on page 14-165
“Model Location” on page 14-166
“Errors and Warnings” on page 14-166
“Limitations of Verilog HDL Import” on page 14-167

To import synthesizable HDL code into the Simulink modeling environment, use HDL import. HDL import parses the input HDL file and generates a Simulink model. The model is a block diagram environment that visually represents the HDL code in terms of functionality and behavior. By importing the HDL code into Simulink, you can verify the functionality of the HDL code by compiling and running simulation on the model in a model-based simulation environment. You can also debug internal signals by logging the signals as test points.

HDL Import

Roundtrip code generation with HDL import is not recommended. Do not use HDL import to import the HDL code that was previously generated from a Simulink model by using the HDL Coder software. The Simulink model that you create is typically at a higher abstraction level. The model generated by HDL import might be at a lower abstraction level. The HDL code you generate from this model might not be usable for production code.

To generate production HDL code, develop your algorithm by using Simulink blocks, MATLAB code, or Stateflow charts. Then, use HDL Coder to generate code.

HDL Import Requirements

To generate a Simulink model, make sure that the HDL file you import:

- Is free of syntax errors.
- Is synthesizable.
- Uses supported Verilog constructs for the import.

Import HDL Code

To import the HDL code, in the MATLAB Command Window, run the `importhdl` function. The function parses the HDL input file that you specified and generates the corresponding Simulink model, and provides a link to open the model.

For example, consider a Verilog code of a comparator,

```
// File Name: comparator.v
// This module implements a simple comparator module

`define value 12
module comparator (clk, rst, a, b);
```

```
input clk, rst;
input [1:0] a;
output reg [1:0] b;

parameter d = 2'b11;

always@(posedge clk) begin
    if (rst)
        b <= 0;
    else if (a < `value)
        b <= a + 1;
end

endmodule
```

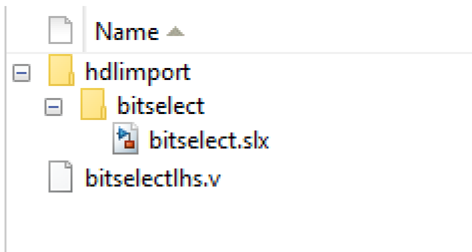
To generate a Simulink model, run the `importhdl` function and specify the HDL input file name. The file name can be specified in different ways, see input argument `'FileNames'` in `importhdl`.

```
importhdl('comparator.v')
```

The constructs that you use in the HDL code can infer simple Simulink blocks, such as Add and Product, to RAM blocks, such as Dual Rate Dual Port RAM. For more examples that illustrate various Simulink models that are inferred, see `importhdl`.

Model Location

The generated Simulink model is named after the top module in the input HDL file that you specify. The model is saved in the `hdlimport/TopModule` path relative to the current working folder. For example, if you input a file named `bitselectlhs.v` to the `importhdl` function that has `bitselect` as the top module name, the generated Simulink model has the name `bitselect.slx`, and is saved in the `hdlimport/bitselect` path relative to the current folder.



Errors and Warnings

When you run the `importhdl` function, HDL import verifies the syntax and semantics of the input HDL code. Semantic verification checks for module instantiation constructs, unused ports in the module definition, the sensitivity list of an `always` block, and so on. If HDL import fails, `importhdl` provides an error message and a link to the file name and line number.

For example, consider this Verilog code for a `bitselect` module:

```
module bitselect(a,c);

input [1:0] a;
output [1:0] c;

c[0] = 0;
assign c[1] = a[2];

endmodule
```

When you run the `importhdl` function, HDL import generates an error message:

```
Parser Error: bitselectlhs.v:6:2: error: Syntax Error near '['..
```

The error message indicates that there is a syntax error in line 6. To fix this error, change the syntax to an assignment statement.

```
assign c[0] = 0;
```

Limitations of Verilog HDL Import

HDL import does not support:

- Importing of VHDL files.
- On Mac Platforms.
- Importing of Verilog files from a read-only folder.
- Generation of the preprocessing files in a read-only file system that parses the HDL code you input to the `importhdl` function.
- Attribute instances and comments, which are ignored.
- (#)delay values, such as `#25`, which are ignored.
- Enumeration data types.
- More than one clock signal.
- Modules that are multirate.
- Recursive module instantiation.
- Multiport Switch inference with more than `1024` inputs. If you specify more than `1024` inputs to a Multiport Switch block that gets inferred from the Verilog code, Verilog import generates an error. The error is generated because the Simulink modeling environment does not support more than `1024` inputs for the block.
- ROM detection from the Verilog code.
- Importing of HDL files that use unsupported Verilog constructs. See “Supported Verilog Constructs for HDL Import” on page 14-169.
- Importing of HDL files that use unsupported dataflow modeling patterns. See “Unsupported Verilog Dataflow Patterns” on page 14-176.

See Also

Functions
`checkhdl` | `makehdl`

Related Examples

- "Generate Simulink Model from CORDIC Atan2 Verilog Code" on page 14-187

More About

- "Supported Verilog Constructs for HDL Import" on page 14-169
- "Verilog Dataflow Modeling with HDL Import" on page 14-174

Supported Verilog Constructs for HDL Import

In this section...

"Module Definition and Instantiations" on page 14-169
"Data Types and Vectors" on page 14-170
"Identifiers and Comments" on page 14-170
"Assignments" on page 14-171
"Operators" on page 14-171
"Conditional and Looping Statements" on page 14-172
"Procedural Blocks and Events" on page 14-172
"Other Constructs" on page 14-172

Use HDL import to import synthesizable HDL code into the Simulink modeling environment. To import the HDL code, use the `importhdl` function. Make sure that the constructs used in the HDL code are supported by HDL import.

These tables list the supported Verilog constructs that you can use when you import your HDL code. If you use an unsupported construct, HDL import generates an error when parsing the input HDL file. Verilog HDL import can sometimes ignore the presence of certain constructs in the HDL code. To learn more, see the **Comments** section of the table.

Module Definition and Instantiations

Verilog Constructs	Supported?	Comments
Library declaration	No	–
Configuration declaration	No	–
Module declaration	Yes	Multiple sample rates and multiple clock inputs are not supported.
Module parameter port list	Yes	–
Port declarations	Yes	INOUT ports are not supported.
Module without ports	No	–
Local parameter declaration	Yes	–
Parameter declaration	Yes	You can use parameters and constants that have a maximum size of 64 bits. By default, the parameter size is 32 bits.

Verilog Constructs	Supported?	Comments
Module instantiation	Yes	• Unconnected ports in the instantiated modules are removed when importing the Verilog code. • Recursive module instantiation is not supported. Instead, if your top module instantiates modules that are defined in recursive subfolders, `importhdl` parses all Verilog files. For example, in this figure, `importhdl` can parse both `DFF_Instantiation.v` and `TFF_Instantiation.v` that are instantiated in `DFF.v`. TopModule DFF.v ModInst1_DFF DFF_Instantiation.v ModInst2_TFF TFF_Instantiation.v

Data Types and Vectors

Verilog Constructs	Supported?	Comments
Net declaration (Wire, Supply0, Supply1)	Yes	–
Real declaration	No	–
String declaration	No	–
Vector declaration	Yes	–
Array support and array indexing	Yes	–
Reg declaration	Yes	–
Integer declaration	Yes	–

Identifiers and Comments

Verilog Constructs	Supported?	Comments
Lexical tokens (Whitespace, operator, comment)	Yes	–
Identifiers (Simple, Escaped)	Yes	–
System Functions ($signed, $unsigned)	Yes	–

Verilog Constructs	Supported?	Comments
Attribute instances	No	HDL import ignores these constructs.
Comments	No	HDL import ignores these constructs.
Numbers (Decimal, Binary, Hexadecimal, and Octal)	Yes	–
Compiler directives (`define,`undef, `ifndef, `else if)	Yes	–

Assignments

Verilog Constructs	Supported?	Comments
Continuous assignment	Yes	–
Blocking assignment	Yes	--
Nonblocking assignment	Yes	–
Procedural assignment (`Always` block)	Yes	–

Operators

<table>
<tr><th>Verilog Constructs</th><th>Supported?</th><th>Comments</th></tr>
<tr><td>Arithmetic operators (+, -, *, **, /, <<<, >>>)</td><td>Yes</td><td>–</td></tr>
<tr><td>Logical operators (<<, >>, !, &&, | |, ==, !=)</td><td>Yes</td><td>–</td></tr>
<tr><td>Relational operators (>, <, >=, <=, ==, !=)</td><td>Yes</td><td>–</td></tr>
<tr><td>Bitwise operators (~, &, |, ^, ~^, ^~)</td><td>Yes</td><td>–</td></tr>
<tr><td>Unary operators (+, -)</td><td>Yes</td><td>Supported for restricted data types</td></tr>
<tr><td>Power operators</td><td>Yes</td><td>Supported for restricted data types</td></tr>
<tr><td>Conditional operators (?:)</td><td>Yes</td><td>–</td></tr>
<tr><td>Concatenation</td><td>Yes</td><td>–</td></tr>
<tr><td>Bit Select</td><td>Yes</td><td>–</td></tr>
<tr><td>Reduction operators (&, ~&, |, ~|, ^, ~^, or ^~)</td><td>Yes</td><td>–</td></tr>
</table>

For an example that illustrates how to use different operators, see “Generate Simulink Model from Verilog Code for Various Operators”.

Conditional and Looping Statements

Verilog Constructs	Supported?	Comments
If-else statement	Yes	–
Conditional operators (?:)	Yes	–
For loop	Yes	–
Loop Generate construct	Yes	Supports loop generate constructs such as for-generate, case-generate, and if-generate constructs.
Conditional Generate construct	No	–
Generate region	No	–
Genvar declaration	No	–
Case statement	Yes	casex and casez statements are also supported.

Procedural Blocks and Events

Verilog Constructs	Supported?	Comments
Task declaration	No	–
Initial construct (ROM modeling)	No	–
Sequential blocks	Yes	–
Block declarations	Yes	–
Event control statements	Yes	–
Function calls	Yes	HDL import does not support recursive function calls.
Task enable	No	–
`Always` construct	Yes	–
Function declaration	Yes	–

Other Constructs

Verilog Constructs	Supported?	Comments
Gate instantiation	No	–
Specparams	No	–
Specify block	No	–
Semantic verification (unused ports, correct module instantiation)	Yes	–

Verilog Constructs	Supported?	Comments
Clock bundle identification	Yes	Multiple sample rates and multiple clock signals are not supported.
Register inference	Yes	–
Compare to Constant block inference	Yes	–
Gain block inference	Yes	–
RAM inference	Yes	–
ROM inference	No	–
Counter inference	No	–
Drive strength	No	–

See Also

Functions
`checkhdl` | `makehdl`

Related Examples

- “Generate Simulink Model from CORDIC Atan2 Verilog Code” on page 14-187

More About

- “Import Verilog Code and Generate Simulink Model” on page 14-165
- “Verilog Dataflow Modeling with HDL Import” on page 14-174

Verilog Dataflow Modeling with HDL Import

In this section...
"Supported Verilog Dataflow Patterns" on page 14-174
"Unsupported Verilog Dataflow Patterns" on page 14-176

Use HDL import to import synthesizable HDL code into the Simulink modeling environment. To import the HDL code, use the `importhdl` function. Make sure that the constructs used in the HDL code are supported by HDL import.

These tables list the supported Verilog HDL dataflow patterns that you can use when importing the HDL code. If your code uses an unsupported dataflow model such as code that infers a latch, `importhdl` generates an error message with a link to the file name and line number. You can then update the code as illustrated in the preceding examples.

Supported Verilog Dataflow Patterns

Verilog Dataflow Model	Example Verilog Code

Blocking assignments in sequential always blocks and nonblocking assignments in combinational always blocks.	For example, this Verilog code uses a sequential assignment for the variable `temp` in a combinational `always` block.
	
	`module dataconv(clk,a,b,c);`
	
	`input clk; integer i;`
	`input wire [7:0] a, b;`
	`output wire [7:0] c;`
	`reg [1:0] temp [0:7];`
	
	`always @(*) begin`
	`    for (i=0;i<=7;i=i+1) begin`
	`        temp[i] <= a[i] + b[i];`
	`    end`
	` end`
	
	`assign c = temp;`
	
	`endmodule`

<table>
<tr><th>Verilog Dataflow Model</th><th>Example Verilog Code</th></tr>
<tr><td>Multiple assignments to the same signal. You can use partial and complete assignments to that signal.</td><td>This example shows the Verilog code that performs both partial assignment and complete assignment to the variable <code>out1_reg</code>.
<pre>
module testPartialAndCompleteAssign
    (input [2:0] in1, in2,
     input cond, clk,
     output [2:0] out1);

  reg [2:0] out1_reg;

  always@(posedge clk) begin
    if(cond) begin
      out1_reg[0] = 1'b0;
      out1_reg[1] = 1'b0;
      out1_reg[2] = 1'b0;
    end
    else begin
      out_reg = in1 & in2;
    end
  end
  assign out1 = out1_reg;

endmodule
</pre></td></tr>
<tr><td>Multiple assignments to the same variable in the true or false path of a Switch block that gets inferred.</td><td>For example, this Verilog code performs multiple assignments to the variable <code>out1</code> inside both the if and else conditions of the always block.
<pre>
module testSwitchMuxing
    (input [2:0] in1,
     input cond,
     output reg [2:0] out1);

  always@(*) begin
      if (cond) begin
          out1 = in1;
      end
      else begin
          out1 = 3'd2;
          out1 = 3'd1;
      end
   end

endmodule
</pre></td></tr>
</table>

<table>
<tr><th>Verilog Dataflow Model</th><th>Example Verilog Code</th></tr>
<tr><td>Bit select, part select, and array indexing operations with signals. A Multiport Switch is inferred when array indexing is performed at the RHS. An array indexing operation on the LHS is inferred as an Assign block.</td><td>For example, this Verilog code uses multiple assignments to the variable <code>out1</code> in the false branch, which is not supported.
<pre>
module ArrayIndexing
    (In1, In2, In3, In4,
    Sel1, Sel2,
    Out1, Out2, Out3);

parameter w = 7;
input [w:0] In1, In2, In3, In4;
input [1:0] Sel1, Sel2;
output [w:0] Out1;
output [1:0] Out2;
output   Out3;

wire [w:0] v[3:0];

assign v[0] = In1;
assign v[1] = In2;
assign v[2] = In3;
assign v[3] = In4;

//Array indexing with signal Sel1
assign Out1 = v[Sel1];

//Part select on array index with signal Sel2
assign Out2 = v[Sel2][7:2];

//Bit select on array index with signal Sel
assign Out3 = v[Sel2][4];

endmodule
</pre></td></tr>
</table>

Unsupported Verilog Dataflow Patterns

These dataflow constructs are unsupported when you import the Verilog code. The *Description and Example Scenarios* column describes each scenario with an example and illustrates how you can avoid this scenario.

<table>
<tr><th>Verilog Dataflow Model</th><th>Description and Example Scenarios</th></tr>
<tr><td>Latch detection from the Verilog code.</td><td>Presence of latches in the HDL code can result in algebraic loops in the generated Simulink model and result in model compilation failures. To avoid latch inference, specify all branches in if-else conditions and in case statements.

For example, when you import this Verilog code, the if-condition is inferred as a Switch block. The block has a true path that is specified in the code. The output of the Switch block is fed back directly as input to the false path which results Fin an algebraic loop.

<pre>
module testAlgebraicLoop(input cond,
        input [2:0] in1,
        output reg [2:0] out1);

// causes latch inference - unsupported
always@(*) begin
    if (cond) begin
        out1 = in1;
    end
end

// Specify else branch to avoid latch inference
// always@(*) begin
//   if (cond) begin
//   out1 = in1;
//   end
//   else begin
//   out1 = in1 + 1;
//   end
// end

endmodule
</pre></td></tr>
</table>

<table>
<tr><th>Verilog Dataflow Model</th><th>Description and Example Scenarios</th></tr>
<tr><td>Performing operations on clock, reset, or clock enable signals in the Verilog code.</td><td>

When you define signals using names such as `clk`, `rst`, and `enb`, HDL import infers these signals to be the clock, global reset, and clock enable signals.

For example, this Verilog code uses an explicit assignment with the clock signal `clk`. This construct is not supported.

```
module testOperationOnClkBundle
    (input [2:0] in1,
     input clk,
     output reg [2:0] out1,
     output out2);

  reg [2:0] out1_reg;

  always@(posedge clk) begin
     out1_reg <= in1;
    end

  assign out2 = clk && 1'b1;

endmodule
```

Make sure that your Verilog code does not perform operations on any of the signals in the clock bundle. In addition, for signals that you use for performing computations in your code, make sure that the signal names do not match any of the names that are inferred as clock, reset, or enable signals. See the `clockBundle` name-value pair of the `importhdl` function for possible signal names that are inferred as signals in the clock bundle.

</td></tr>
</table>

<table>
<tr><th>Verilog Dataflow Model</th><th>Description and Example Scenarios</th></tr>
<tr><td>Sensitivity of clock or reset signal to different clock edges or different reset edges inside the same module.</td><td>You cannot have one always block with the clock signal sensitive to the positive edge and the other always block with the clock signal sensitive to the negative edge inside the same module.

For example, this Verilog code uses the positive and negative edges of the same clock, which is not supported.

<pre>
module testMultipleClockEdges
    (input [2:0] in1, in2,
     input clk,
     output [2:0] out1, out2);

  reg [2:0] out1_reg, out2_reg;

  /* clk sensitivity to posedge */
  always@(posedge clk) begin
     out1_reg <= in1 && in2;
    end

  /* clk sensitivity to neegedge */
  always@(negedge clk) begin
     out2_reg <= in1 || in2;
    end

  assign out1 = out1_reg;
  assign out2 = out2_reg;

endmodule
</pre>
Make sure that your Verilog code does not use both edges of the clock or reset signal in the same module. Use either posedge or negedge of the clock.</td></tr>
</table>

<table>
<tr><th>Verilog Dataflow Model</th><th>Description and Example Scenarios</th></tr>
<tr><td>Realization of synchronous and asynchronous circuits inside the same module.</td><td>You cannot use Verilog code realizes both circuits in the same module as shown in the code below. Use different modules for realization of asynchronous and synchronous circuits.

```
module testSynchronousAsynchronous
    (input [2:0] in1, in2,
     input clk, reset,
     output [2:0] out1, out2);

  reg [2:0] out1_reg, out2_reg;

  /* synchronous always block */
  always@(posedge clk) begin
     out1_reg <= in1 && in2;
    end

  /* asynchronous always block */
  always@(posedge clk or posedge reset)
  begin
     out2_reg <= in1 || in2;
    end

  assign out1 = out1_reg;
  assign out2 = out2_reg;

endmodule
```
</td></tr>
</table>

<table>
<tr><th>Verilog Dataflow Model</th><th>Description and Example Scenarios</th></tr>
<tr><td>Initialization of multiple RAMs that are inferred in the same module.</td><td>For example, this Verilog code infers <code>sample_store0</code> and <code>sample_store1</code> as RAMs. This construct is not supported. Separate each RAM inference into a single module.
<pre>
module testMultipleRAMs
    (input [2:0] in1, in2,
     input clk, reset,
     output reg [2:0] read_data0, read_data1);

  reg [2:0] out1_reg, out2_reg;

  /* Inference of RAM sample_store0 */
  always@(posedge clk) begin
    if (write_enable) begin
      sample_store0[write_address] <= write_data;
    end
    read_data0 = sample_store0[read_address0]
  end

  /* Inference of RAM sample_store1 */
  always@(posedge clk) begin
    if (write_enable) begin
      sample_store1[write_address] <= write_data;
    end
    read_data1 = sample_store0[read_address1]
  end

endmodule
</pre></td></tr>
</table>

<table>
<tr><th>Verilog Dataflow Model</th><th>Description and Example Scenarios</th></tr>
<tr><td>Usage of certain constructs when reading initial values from an <code>initial</code> block.</td><td>An <code>initial</code> block does not support constructs other than assignment statements or for loops with assignments. The RHS of the assignment statement must use only

For example, this Verilog code uses an if-else condition to assign a value to the variable <code>dout_a</code> and assigns a variable <code>write_data</code> to <code>dout_c</code>, which are not supported.

<pre>
module testInitial
        (input cond,
        input [3:0] write_data
        output reg [3:0] dout_a, dout_b, doutc);

        parameter AddrWidth = 3;

        integer i;
        reg [3:0] ram [7:0];

        initial begin
          for (i=0; i<=2**AddrWidth - 1; i=i+1) begin
              ram[i] = 0;
          end
          dout_b = 0;

          // if-else condition - not supported
          if (cond) begin
               dout_a = 0;
          end
          else begin
               dout_a = 4;
          end
          end

          // Variable assignment to RHS - not supported
          dout_a = write_data;

          // Use assignment statements instead for
          // dout_a and dout_c
          // dout_a = 4;
          // dout_c = 32;
          // end
          // end

endmodule
</pre></td></tr>
</table>

<table>
<tr><th>Verilog Dataflow Model</th><th>Description and Example Scenarios</th></tr>
<tr><td>All dimensions of signals not referenced at time of usage.</td><td>Use all dimensions of a signal in the input Verilog code. If you specify an additional dimension, it creates an indexed bit select.

For example, this Verilog code creates an `8-bit` variable `temp`. The assignment inside the `always` block generates an error because it does not use both dimensions of the variable.

```
module dataconv(clk,a,b,c);

input clk;
input wire [1:0] a, b;
output wire [1:0] c;

reg [7:0] temp_reg [1:0][1:0];

// temp_reg is not indexed
// with both dimensions - not supported
always @(posedge clk) begin
        temp_reg [0] <= a[0] + b[0];
        temp_reg [1] <= a[1] + b[1];
    end


// Use both dimensions when indexing a variable
// always @(posedge clk) begin
//        temp_reg [0][0] <= a[0] + b[0];
//        temp_reg [0][1] <= a[0] + b[1];
//        temp_reg [1][0] <= a[1] + b[0];
//        temp_reg [1][1] <= a[1] + b[1];

// You can also perform indexed bit select
//         temp_reg [1][0][1] = 1'b0;
//  end

assign c = temp_reg;

endmodule
```
</td></tr>
</table>

See Also

Functions
`checkhdl` | `makehdl`

Related Examples

- “Generate Simulink Model from CORDIC Atan2 Verilog Code” on page 14-187

More About

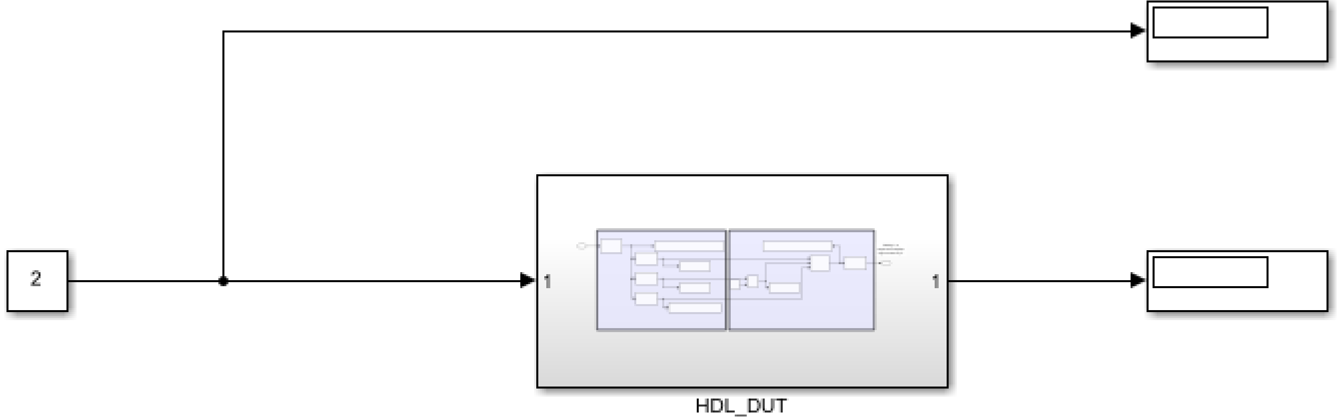
- “Import Verilog Code and Generate Simulink Model” on page 14-165
- “Supported Verilog Constructs for HDL Import” on page 14-169

Simulate and Generate HDL Code for the Float Typecast Block

This example shows how you can use the Float Typecast block to extract the sign, exponent, and mantissa bits from a floating-point input, and then convert the bits back to a floating-point output after performing any computations.

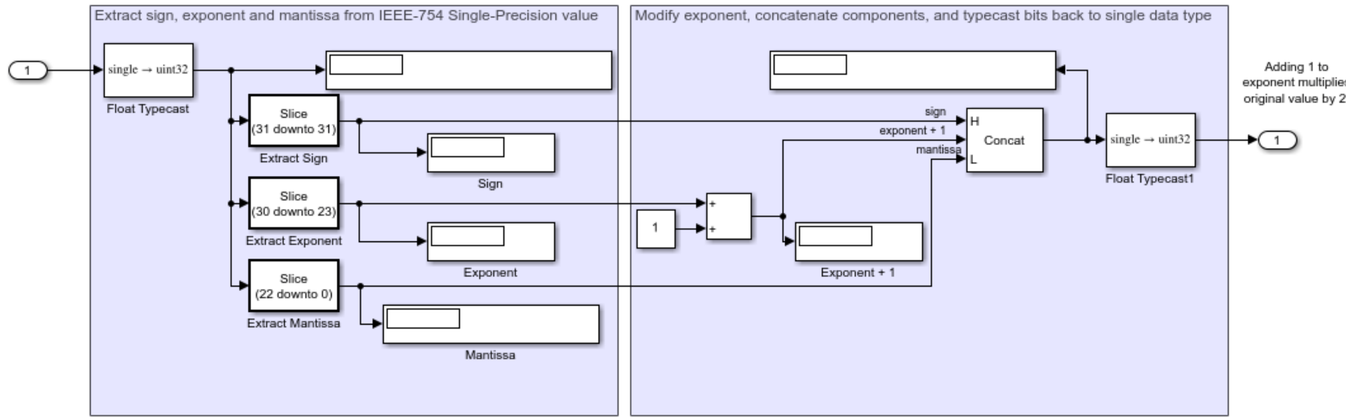
Open the hdlcoder_float_typecast_example model.

```
open_system('hdlcoder_float_typecast_example')
```



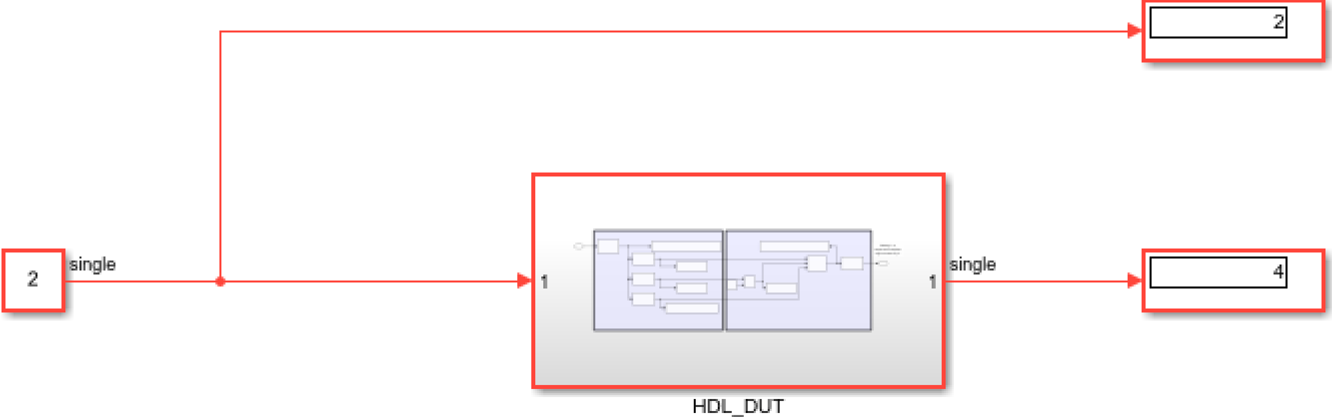
The model multiplies the floating-point input by two to produce the floating-point output. To multiply the input, the algorithm increments the exponent by one. Open the HDL_DUT subsystem.

```
open_system('hdlcoder_float_typecast_example/HDL_DUT')
```



The model is already configured for HDL compatibility by using the `hdlsetup` function. Simulate the model.

```
sim('hdlcoder_float_typecast_example')
open_system('hdlcoder_float_typecast_example')
```



Copyright 2020 The MathWorks, Inc.

Before you generate HDL code, enable the `Native Floating Point` mode.

```
nfpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT');
hdlset_param('hdlcoder_float_typecast_example', ...
                             'FloatingPointTargetConfiguration', nfpconfig);
```

Generate HDL code for the `HDL_DUT` subsystem.

```
makehdl('hdlcoder_float_typecast_example')
```

Generate Simulink Model from CORDIC Atan2 Verilog Code

This example shows how you can import a file containing Verilog® code and generate the corresponding Simulink® model by using the `importhdl` function. `importhdl` imports and parses the specified Verilog files to generate the corresponding Simulink model. The Verilog code in this example contains a CORDIC atan2 algorithm.

CORDIC 2-Argument Arctangent (atan2) Verilog Design

This Verilog input file implements a CORDIC atan2 algorithm.

```
cordic_atan2_verilog_file = 'cordic_atan2.v';
type(cordic_atan2_verilog_file);
```

```
`timescale 1ns / 1ps

module cordic_atan2(
    input clk,                      // clock
    input reset,                    // reset to the system
    input enable,                   // enable
    input signed [15:0] x_in,       // Input x value
    input signed [15:0] y_in,       // Input Y value
    output reg signed [15:0] theta  // Input theta value
);

// Pipeline the input values
reg signed [17:0]x_in_d; reg signed [17:0]y_in_d;
always @(posedge clk)
begin
    if(reset)    begin
        x_in_d <={18{1'b0}};
        y_in_d <={18{1'b0}};
    end else if(enable) begin
        // extend the input values for intermediate calculations
        x_in_d <= {{2{x_in[15]}},x_in};
        y_in_d <= {{2{y_in[15]}},y_in};
    end
end
// pre quad correction logic
reg x_qaud_adjust;  reg y_qaud_adjust;
reg y_non_zero;  reg signed [17:0]x_pre_quad_out;
reg signed [17:0]y_pre_quad_out;

always @(posedge clk)begin
    if(reset)begin
        x_pre_quad_out <= {18{1'b0}};
        y_pre_quad_out <= {18{1'b0}};
        x_qaud_adjust  <= 1'b0;
        y_qaud_adjust  <= 1'b0;
        y_non_zero     <=1'b0;
    end
    else if(enable)begin
        if(x_in_d[17] == 1'b1)begin
            x_pre_quad_out <= -x_in_d;
            x_qaud_adjust  <= 1'b1;
```

```
        end
        else begin
            x_pre_quad_out <= x_in_d;
            x_qaud_adjust  <= 1'b0;
        end
        if(y_in_d[17] == 1'b1)begin
            y_pre_quad_out <= -y_in_d;
            y_qaud_adjust  <= 1'b1;
            y_non_zero     <=1'b1;
        end
        else begin
            y_pre_quad_out <= y_in_d;
            y_qaud_adjust  <= 1'b0;
            y_non_zero     <= |y_in_d; // reduction or for test non zero or not
        end
    end
end

//LOOKUP TABLE FOR THE ANGLES
wire signed [15:0]LUT[0:14];
wire signed [17:0]x_k[0:15];
wire signed [17:0]y_k[0:15];
wire signed [17:0]z_k[0:15];
wire signed [15:0]theta_temp;
parameter PI = 16'sh6488;

assign LUT[0]  = 16'h1922;
assign LUT[1]  = 16'h0Ed6;
assign LUT[2]  = 16'h07D7;
assign LUT[3]  = 16'h03FB;
assign LUT[4]  = 16'h01FF;
assign LUT[5]  = 16'h0100;
assign LUT[6]  = 16'h0080;
assign LUT[7]  = 16'h0040;
assign LUT[8]  = 16'h0020;
assign LUT[9]  = 16'h0010;
assign LUT[10] = 16'h0008;
assign LUT[11] = 16'h0004;
assign LUT[12] = 16'h0002;
assign LUT[13] = 16'h0001;
assign LUT[14] = 16'h0000;

assign x_k[0]  = x_pre_quad_out;
assign y_k[0]  = y_pre_quad_out;

wire  signed [17:0]theta_temp0;
assign theta_temp0 = z_k[15];
assign theta_temp  = theta_temp0[15:0];
assign z_k[0] = 18'd0;

//cordic iterator
genvar  i;
generate
for (i = 0; i<15; i =i+1)
begin
    Kernel cordic_iterator (.clk (clk),
        .reset(reset),
        .enable  (enable ),
```

```
        .itr_num(i),
        .x_in(x_k[i]),
        .y_in(y_k[i]),
        .z_in(z_k[i]),
        .lut_constant(LUT[i]),
        .x_out(x_k[i+1]),
        .y_out(y_k[i+1]),
        .z_out(z_k[i+1])
    );
end
endgenerate

// matching delays for the control signls of the pre quadrant correction logic
reg x_qaud_adjust_match_delay[0:14];
reg y_qaud_adjust_match_delay[0:14];
reg y_non_zero_match_delay[0:14];
integer k;
always @(posedge clk)begin
    if(reset)begin
        x_qaud_adjust_match_delay[0] <= 1'b0; y_qaud_adjust_match_delay[0] <= 1'b0;
        y_non_zero_match_delay   [0] <= 1'b0; x_qaud_adjust_match_delay[1] <= 1'b0;
        y_qaud_adjust_match_delay[1] <= 1'b0; y_non_zero_match_delay   [1] <= 1'b0;
        x_qaud_adjust_match_delay[2] <= 1'b0; y_qaud_adjust_match_delay[2] <= 1'b0;
        y_non_zero_match_delay   [2] <= 1'b0; x_qaud_adjust_match_delay[3] <= 1'b0;
        y_qaud_adjust_match_delay[3] <= 1'b0; y_non_zero_match_delay   [3] <= 1'b0;
        x_qaud_adjust_match_delay[4] <= 1'b0; y_qaud_adjust_match_delay[4] <= 1'b0;
        y_non_zero_match_delay   [4] <= 1'b0; x_qaud_adjust_match_delay[5] <= 1'b0;
        y_qaud_adjust_match_delay[5] <= 1'b0; y_non_zero_match_delay   [5] <= 1'b0;
        x_qaud_adjust_match_delay[6] <= 1'b0; y_qaud_adjust_match_delay[6] <= 1'b0;
        y_non_zero_match_delay   [6] <= 1'b0; x_qaud_adjust_match_delay[7] <= 1'b0;
        y_qaud_adjust_match_delay[7] <= 1'b0; y_non_zero_match_delay   [7] <= 1'b0;
        x_qaud_adjust_match_delay[8] <= 1'b0; y_qaud_adjust_match_delay[8] <= 1'b0;
        y_non_zero_match_delay   [8] <= 1'b0; x_qaud_adjust_match_delay[9] <= 1'b0;
        y_qaud_adjust_match_delay[9] <= 1'b0; y_non_zero_match_delay   [9] <= 1'b0;
        x_qaud_adjust_match_delay[10] <= 1'b0; y_qaud_adjust_match_delay[10] <= 1'b0;
        y_non_zero_match_delay   [10] <= 1'b0; x_qaud_adjust_match_delay[11] <= 1'b0;
        y_qaud_adjust_match_delay[11] <= 1'b0; y_non_zero_match_delay   [11] <= 1'b0;
        x_qaud_adjust_match_delay[12] <= 1'b0; y_qaud_adjust_match_delay[12] <= 1'b0;
        y_non_zero_match_delay   [12] <= 1'b0; x_qaud_adjust_match_delay[13] <= 1'b0;
        y_qaud_adjust_match_delay[13] <= 1'b0; y_non_zero_match_delay   [13] <= 1'b0;
        x_qaud_adjust_match_delay[14] <= 1'b0; y_qaud_adjust_match_delay[14] <= 1'b0;
        y_non_zero_match_delay   [14] <= 1'b0;
    end
    else if(enable) begin
        x_qaud_adjust_match_delay[0] <=x_qaud_adjust;
        y_qaud_adjust_match_delay[0] <=y_qaud_adjust;
        y_non_zero_match_delay[0]    <=y_non_zero;
        for(k =0; k <14 ;k = k+1)begin
            x_qaud_adjust_match_delay[k+1] <=  x_qaud_adjust_match_delay[k];
            y_qaud_adjust_match_delay[k+1] <= y_qaud_adjust_match_delay[k];
            y_non_zero_match_delay[k+1] <= y_non_zero_match_delay[k];
        end
    end
end
// post quadrant correction logic

always @(posedge clk) begin
    if(reset)
```

```
            theta<=16'd0;
        else if(enable)begin
            if(y_non_zero_match_delay[14])begin
                if(x_qaud_adjust_match_delay[14])begin
                    if(y_qaud_adjust_match_delay[14])
                        theta <=theta_temp -PI;
                    else
                        theta <=PI -theta_temp;
                end
                else begin
                    if(y_qaud_adjust_match_delay[14])
                        theta <= -theta_temp;
                    else
                        theta <= theta_temp;
                end
            end
            else if(x_qaud_adjust_match_delay[14])begin
                theta <= PI;
            end
            else begin
                theta <= 16'd0;
            end
        end
    end

    endmodule

    module  Kernel(
        input clk,
        input reset,
        input enable,
        input signed [17:0]x_in,
        input signed [17:0]y_in,
        input signed [17:0]z_in,
        input [4:0]itr_num,
        input signed [15:0]lut_constant,
        output reg signed [17:0]x_out,
        output reg signed [17:0]y_out,
        output reg signed [17:0]z_out);

    wire signed [17:0] lut_constant_signExtension = {{2{lut_constant[15]}},lut_constant};

    always @(posedge clk)begin
        if(reset)begin
            x_out <={18{1'b0}};
            y_out <={18{1'b0}};
            z_out <= {18{1'b0}};
        end
        else if(enable)begin
            if(y_in[17])begin
                x_out <= x_in -(y_in>>>itr_num);
                y_out <= y_in + (x_in>>>itr_num);
                z_out <= z_in  - lut_constant_signExtension;
            end
            else begin
                x_out <= x_in +(y_in>>>itr_num);
                y_out <= y_in - (x_in>>>itr_num);
                z_out <= z_in  + lut_constant_signExtension;
```

```
        end
    end
end

endmodule
```

This Verilog design contains various commonly used Verilog constructs such as:

- Continuous assignments
- Always blocks
- Conditional Statements
- Module instantiation
- Generate Constructs
- For Loop

Import HDL File Containing CORDIC atan2 Algorithm

To import the Verilog file, specify the file name as an argument to the `importhdl` function.

```
importhdl(cordic_atan2_verilog_file);
```

```
### Parsing <a href="matlab:edit('cordic_atan2.v')">cordic_atan2.v</a>.
### Top Module name: 'cordic_atan2'.
### Identified ClkName::clk.
### Identified RstName::reset.
### Identified ClkEnbName::enable.
Warning: Unused signals detected in the Demux block created for vector index signal 'x_k'. A Demu
Warning: Unused signals detected in the Demux block created for vector index signal 'y_k'. A Demu
### Hdl Import parsing done.
### Removing unconnected components.
### Unconnected components detected when importing the HDL code. These components are removed fro
### Creating Target model cordic_atan2
### Begin model generation 'cordic_atan2'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Start Layout...
### Working on hierarchy at ---> 'cordic_atan2'.
### Laying out components.
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2'.
### Laying out components.
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator1'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator10'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator11'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator12'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator13'.
### Laying out components.
```

```
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator14'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator2'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator3'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator4'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator5'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator6'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator7'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator8'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator9'.
### Laying out components.
### Drawing block edges...
### Drawing block edges...
### Drawing block edges...
### Model generation complete.
### Setting model parameters.
### Generated model file C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\14\tpc26661ee\hdlcoder-ex38650911\
### Importhdl completed.
```

`importhdl` parses the input file and displays messages of the import process in the MATLAB™ Command Window. The import provides a link to the generated Simulink model `cordic_atan2.slx`. The generated model uses the same name as the top module in the input Verilog file.

Examine Generated Simulink Model

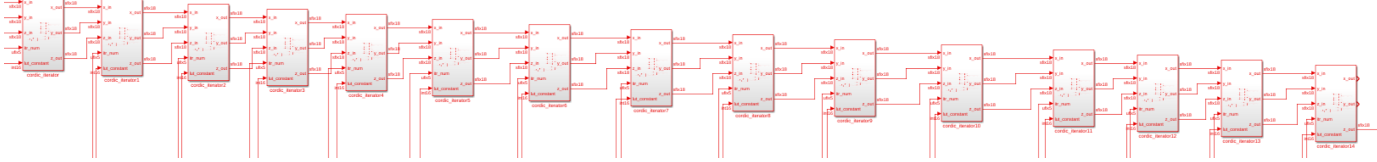
To open the generated Simulink model, click the link in the Command Window. The model is saved in the `hdlimport/cordic_atan2` path relative to the current folder. You can simulate the model and observe the simulation results.

```
open_system('hdlimport/cordic_atan2/cordic_atan2')
Simulink.BlockDiagram.arrangeSystem('cordic_atan2')
set_param('cordic_atan2', 'UnconnectedOutputMsg', 'None');
sim('hdlimport/cordic_atan2/cordic_atan2.slx');
```

Generated Module instances

The Verilog code instantiates 15 kernel modules by using the generate construct. In the generated Simulink model, 15 kernel modules are seen.

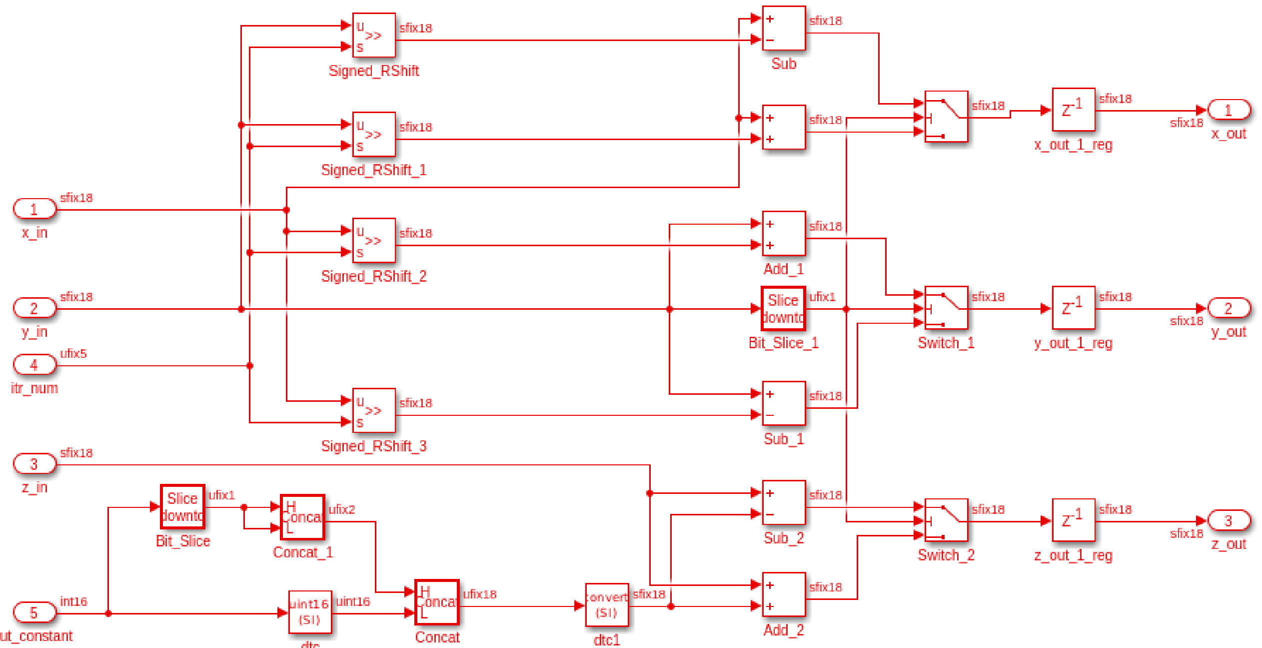
```
generate
    for (i = 0; i<15; i =i+1)
    begin
        Kernel cordic_iterator (.clk(clk),
                                .reset(reset),
                                .enable(enable),
                                .itr_num(i),
                                .x_in(x_k[i]),
                                .y_in(y_k[i]),
                                .z_in(z_k[i]),
                                .lut_constant(LUT[i]),
                                .x_out(x_k[i+1]),
                                .y_out(y_k[i+1]),
                                .z_out(z_k[i+1]));
    end
endgenerate
```



Simulink model for the Kernel Module

```
module  Kernel(
    input clk,
    input reset,
    input enable,
    input signed [17:0]x_in,
    input signed [17:0]y_in,
    input signed [17:0]z_in,
    input [4:0]itr_num,
    input signed [15:0]lut_constant,
    output reg signed [17:0]x_out,
    output reg signed [17:0]y_out,
    output reg signed [17:0]z_out);

wire signed [17:0] lut_constant_signExtension = {{2{lut_constant[15]}},lut_constant};

always @(posedge clk)begin
      if(reset)begin
         x_out <={18{1'b0}};
         y_out <={18{1'b0}};
         z_out <= {18{1'b0}};
      end
    else if(enable)begin
        if(y_in[17])begin
            x_out <= x_in -(y_in>>>itr_num);
            y_out <= y_in + (x_in>>>itr_num);
            z_out <= z_in  - lut_constant_signExtension;
        end
```

```
            else begin
                x_out <= x_in +(y_in>>>itr_num);
                y_out <= y_in - (x_in>>>itr_num);
                z_out <= z_in  + lut_constant_signExtension;
              end
          end
end

 endmodule
```



References

- `importhdl`
- “Import Verilog Code and Generate Simulink Model” on page 14-165

Simulink to HDL Examples for Communication and Signal Processing Applications

- "Programmable FIR Filter for FPGA" on page 15-2
- "Multichannel FIR Filter for FPGA" on page 15-7
- "High-Throughput Channelizer for FPGA" on page 15-10
- "Implement Digital Downconverter for FPGA" on page 15-20
- "Implement Digital Upconverter for FPGA" on page 15-38
- "HDL QAM Transmitter and Receiver" on page 15-58
- "Airplane Tracking with ADS-B Captured Data" on page 15-78
- "Generate HDL Code for Viterbi Decoder" on page 15-85
- "Design Video Processing Algorithms for HDL in Simulink" on page 15-95
- "Edge Detection and Image Overlay" on page 15-102
- "Lane Detection" on page 15-107
- "HDL QPSK Transmitter and Receiver" on page 15-125

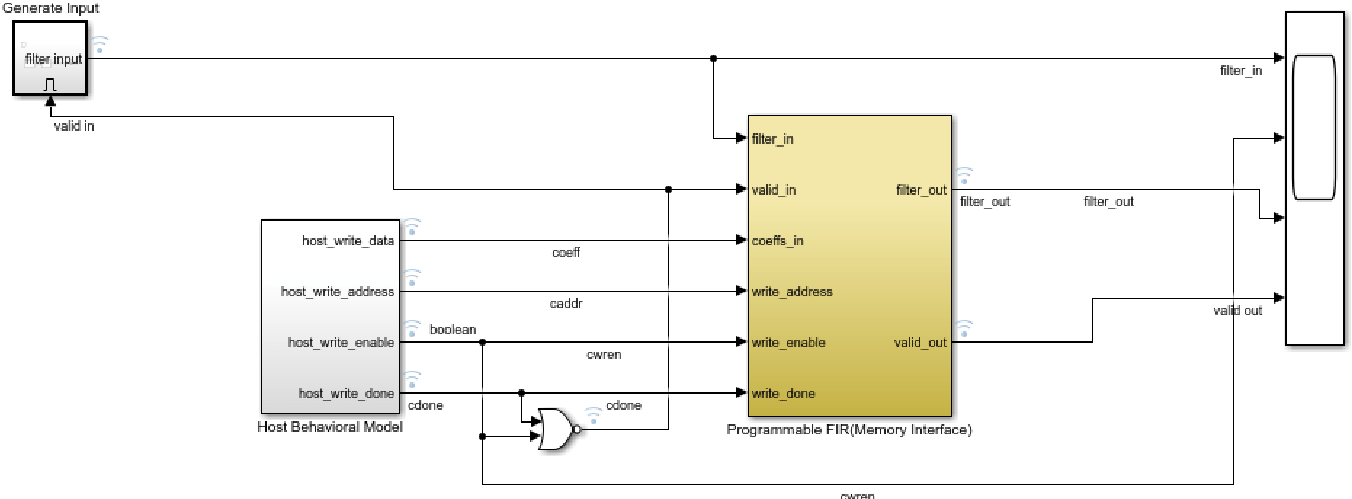
Programmable FIR Filter for FPGA

This example shows how to implement a programmable FIR filter for hardware. You can program the filter to the required response by loading the coefficients into internal registers using a memory-style interface.

This example implements a bank of two filters, each with a different response. Both filters have the same length and symmetry of coefficients. This pattern of coefficients allows the block to share multipliers for symmetric coefficients in the same way for both filters. The same filter hardware can be programmed with new coefficients to obtain a different filter response.

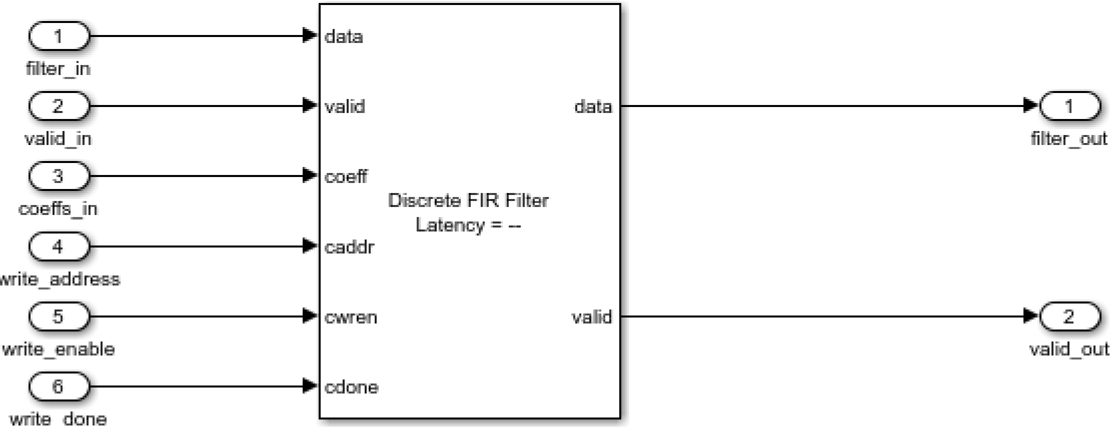
Model Programmable FIR Filter

This example implements two FIR filters, one with a lowpass response and the other with a highpass response. The coefficients are specified in the Model Properties>Callbacks>InitFcn function.



The Programmable FIR(Memory Interface) subsystem contains the Discrete FIR Filter block, with the **Coefficients source** parameter set to `Input port (Memory interface)`. This configuration enables a memory-style set of ports where you can write the coefficients into the filter. First, the Host Behavioral Model subsystem loads the lowpass coefficients to the FIR filter. The model delays the input chirp samples until the first coefficient write is complete. Then the Host Behavioral Model subsystem loads the highpass coefficients.

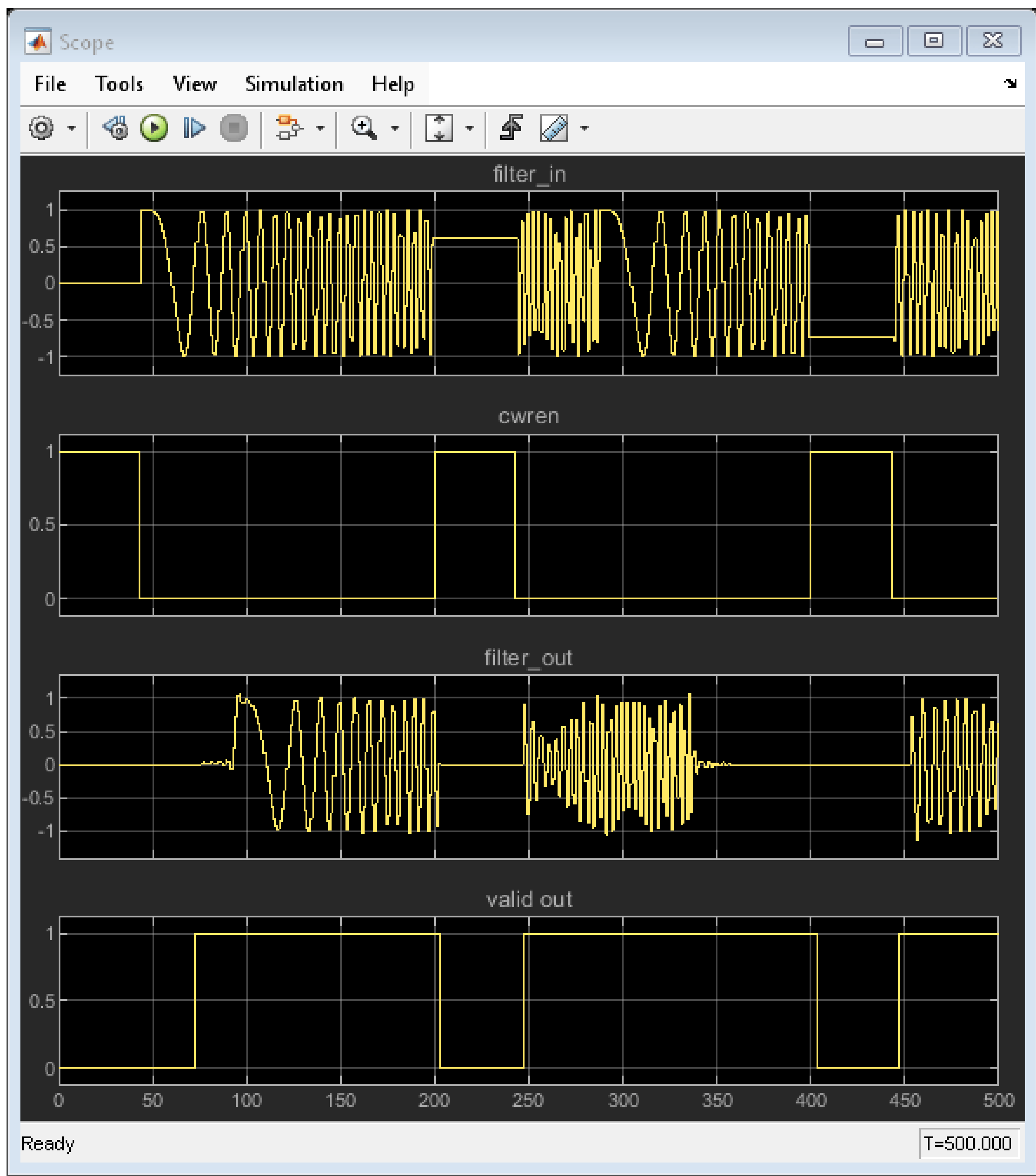
While you write coefficients to the filter, the filter ignores any input data samples. This model stops sending input data while the coefficients are changing. You can also use the output valid signal to determine when the output data is valid.



The Discrete FIR Filter block in this model is configured to use a fully parallel direct form systolic architecture. The block also supports using the memory-style programmable coefficients when the block uses the parallel transposed or serial architectures. The **Coefficient prototype** parameter is set to one of the coefficient vectors. This setting enables the block to optimize hardware resources based on the symmetry and position of zero-valued coefficients. When you specify a prototype, all input coefficient sets must match the prototype in symmetry and location of zero-valued coefficients. For more detail about how to use the prototype to optimize filter resources, see "Optimize Programmable FIR Filter Resources" (DSP HDL Toolbox).

Simulink Simulation Results

To view the input and output data and the signals of the coefficient interface, open the Scope and run the example model.

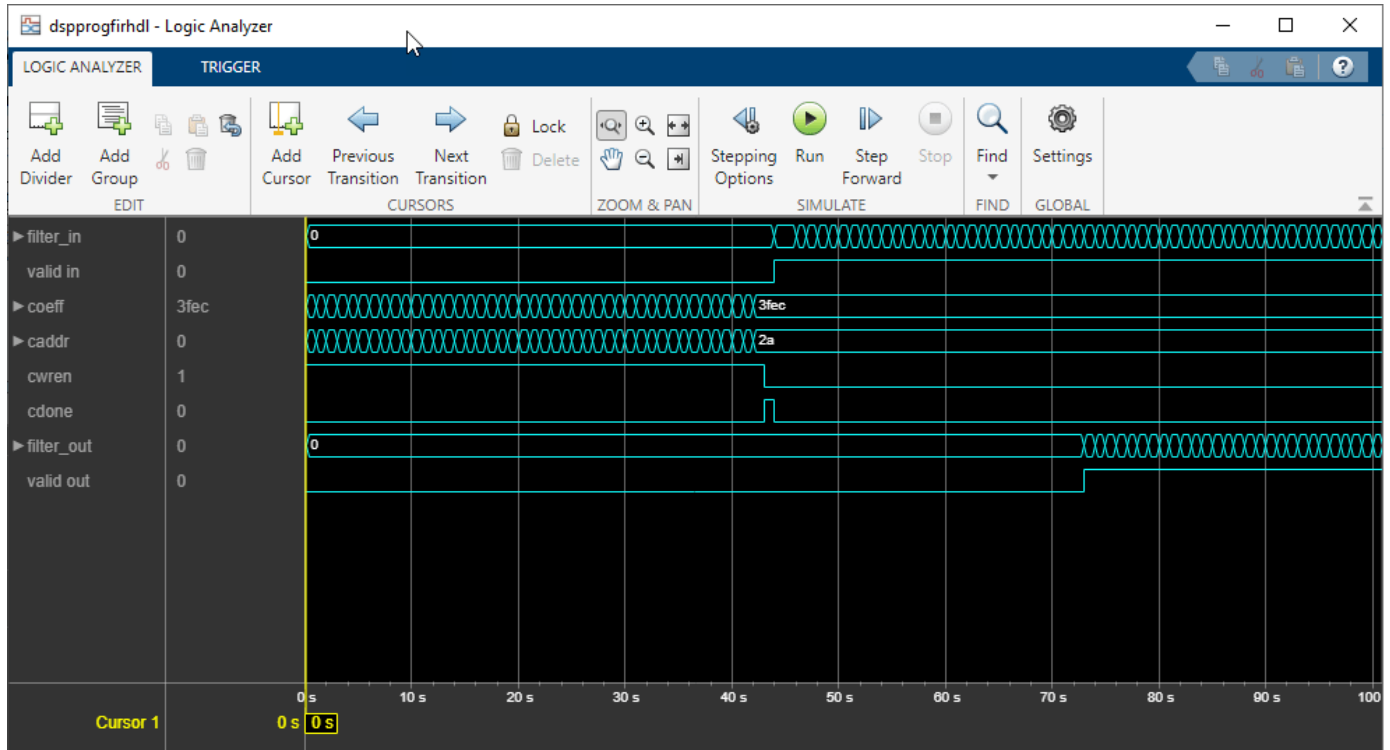


Using the Logic Analyzer

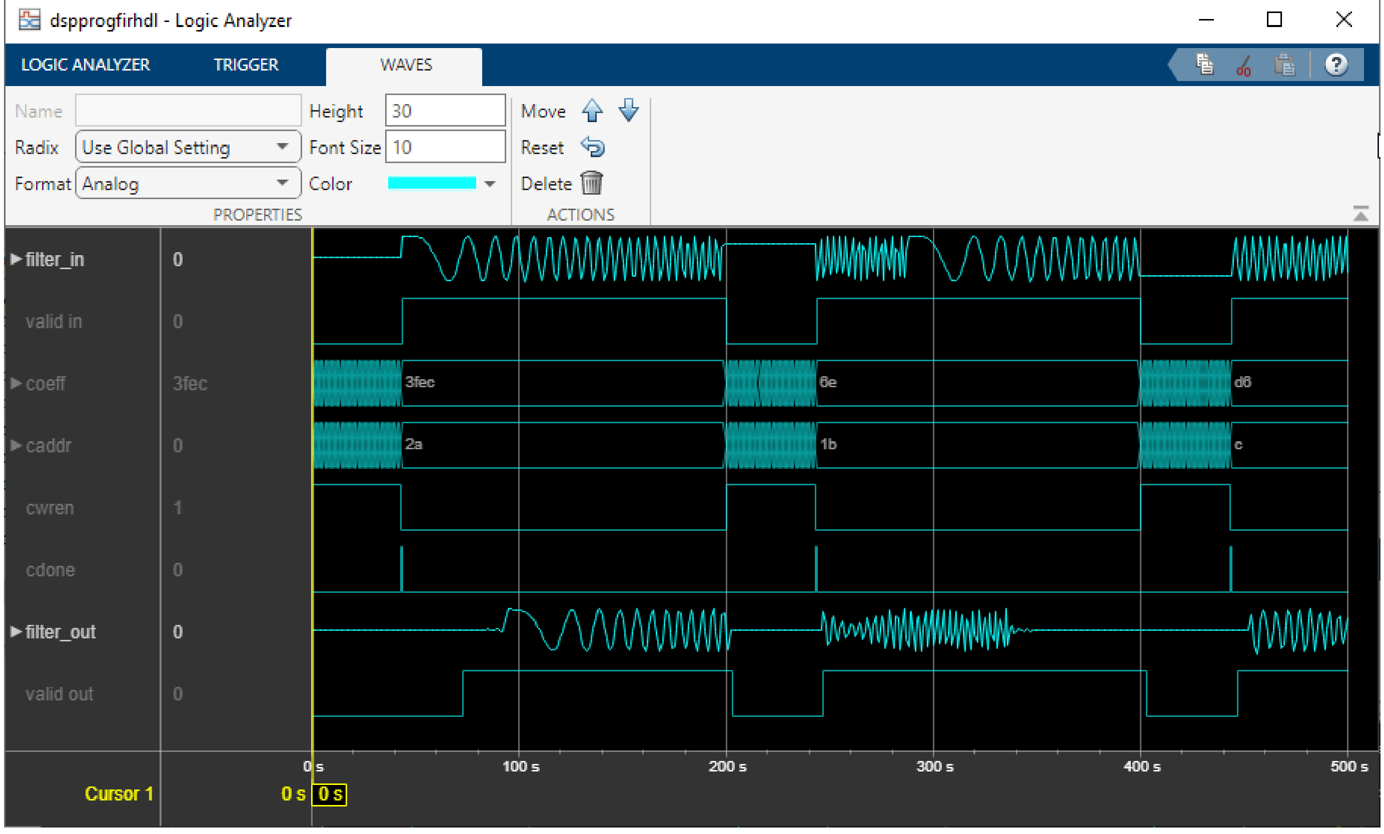
You can also view the signals from the model in the Logic Analyzer. The Logic Analyzer enables you to view multiple signals in one window and lets you find the transitions in the signals and measure the latency between signals.

Launch the Logic Analyzer from the model's toolstrip.

The model already has some signals of interest (input coefficients, write address, write enable, write done, filter in, filter out, and valid signals) added to the Logic Analyzer for observation.



The Logic Analyzer display can also be controlled on a per-wave or per-divider basis. To modify an individual wave or divider, select a wave or divider and then click the **Waves** tab. A useful mode of visualization in the Logic Analyzer is the `Analog` format.



For more information, see Logic Analyzer (DSP System Toolbox).

Generate HDL Code and Test Bench

You must have an HDL Coder™ license to generate HDL code for this example model. Use this command to generate HDL code.

```
systemname = [modelname '/Programmable FIR(Memory Interface)'];
makehdl(systemname);
```

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior.

```
makehdltb(systemname);
```

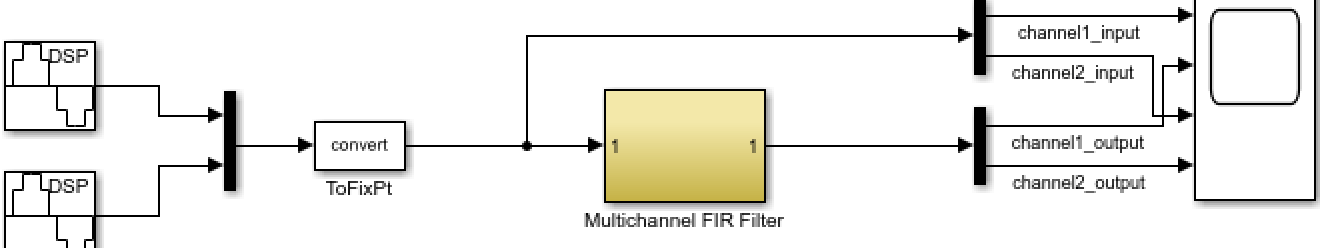
Multichannel FIR Filter for FPGA

This example shows how to implement a discrete FIR filter with multiple input data streams for hardware.

In many DSP applications, multiple data streams are filtered by the same filter. The straightforward solution is to implement a separate filter for each channel. You can create a more area-efficient structure by sharing one filter implementation across multiple channels. The resulting hardware requires a faster clock rate compared to the clock rate used for a single channel filter.

Model Multichannel FIR Filter

```
modelname = 'dspmultichannelhdl';
open_system(modelname);
```



Launch HDL Dialog

Copyright 2012 The MathWorks, Inc.

The model contains a two-channel FIR filter. The input data vector includes two streams of sinusoidal signal with different frequencies. The input data streams are processed by a lowpass filter whose coefficients are specified by the Model Properties `InitFcn` Callback function.

Select a fully parallel architecture for the Discrete FIR Filter block, and enable resource sharing across multiple channels.

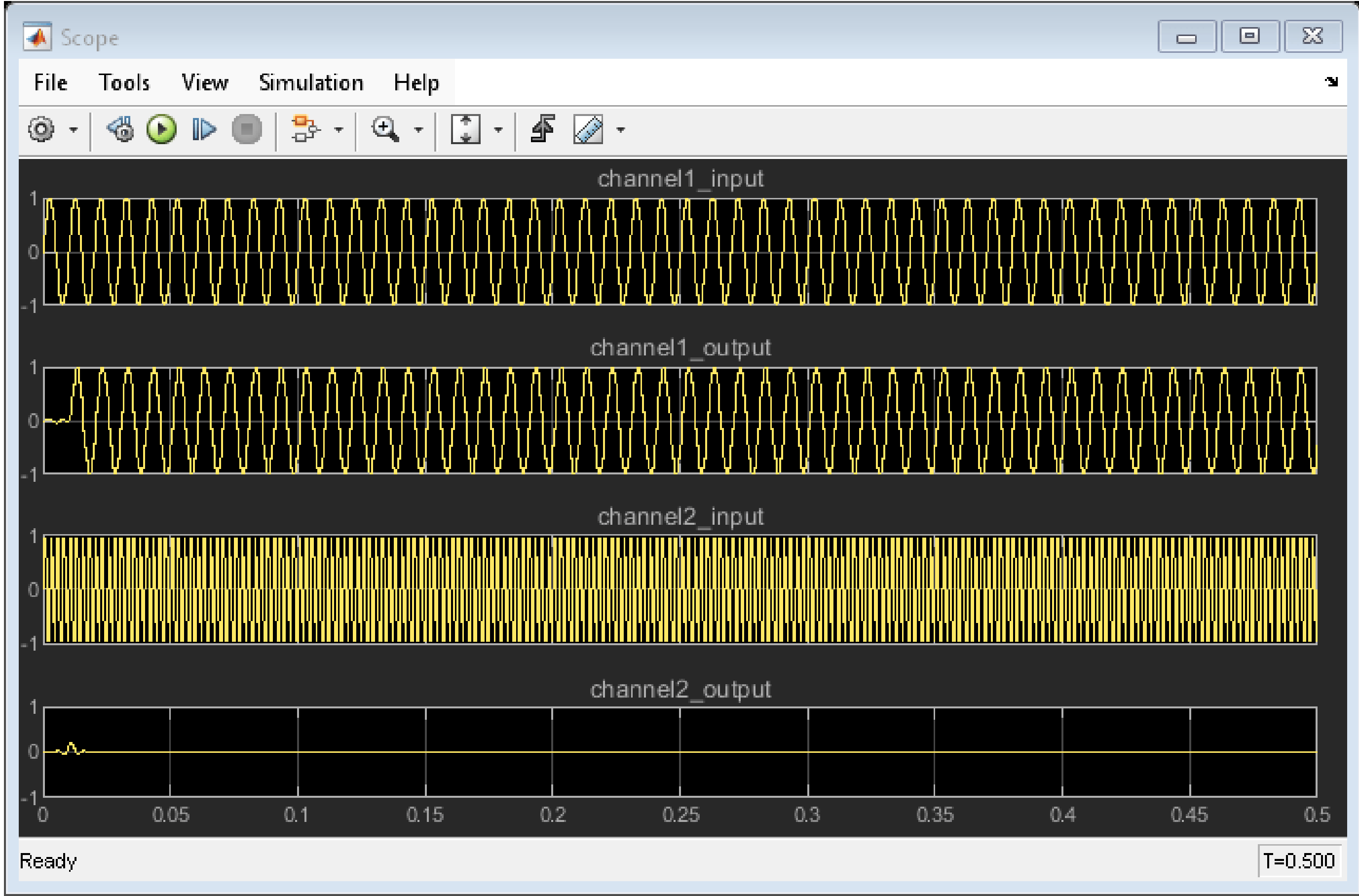
```
systemname = [modelname '/Multichannel FIR Filter'];
blockname = [systemname '/Discrete FIR Filter'];
set_param(blockname,'FilterStructure','Direct form symmetric');
hdlset_param(blockname,'Architecture','Fully Parallel');
hdlset_param(blockname,'ChannelSharing','On');
```

You can alternatively specify these settings on the **HDL Block Properties** menu, which you access by right-clicking a block and selecting **HDL Code > HDL Block Properties**.

Simulation Results

Run the example model and open the scope to compare the two data streams.

```
sim(modelname);
open_system([modelname '/Scope']);
```



Generate HDL Code and Test Bench

You must have an HDL Coder™ license to generate HDL code for this example model. Use this command to generate HDL code for the Multichannel FIR Filter subsystem. Enable the resource use report.

```
makehdl(systemname,'resource','on');
```

Use this command to generate a test bench that compares the HDL simulation results with the Simulink model results.

```
makehdltb(systemname);
```

Compare Resource Utilization

To compare resource use with and without sharing, you can disable sharing resources across channels and generate HDL code again, then compare the resource use reports.

```
hdlset_param(blockname,'ChannelSharing','Off');
makehdl(systemname,'resource','on');
```

Summary

Multipliers	44
Adders/Subtractors	86
Registers	86
RAMs	0
Multiplexers	0

Channels are not shared

Summary

Multipliers	22
Adders/Subtractors	43
Registers	91
RAMs	0
Multiplexers	1

Channels are shared

High-Throughput Channelizer for FPGA

This example shows how to implement a high-throughput, gigasamples-per-second (GSPS), channelizer for hardware by using a polyphase filter bank.

High speed signal processing is a requirement for applications such as radar, broadband wireless, and backhaul communication. Modern ADCs can sample signals at sample rates up to several GSPS, but the clock speeds for the fastest FPGA fall short of this sample rate. FPGAs typically run at hundreds of MHz. To perform GSPS processing on an FPGA, you can move from scalar processing to vector processing and process multiple samples in parallel at a much lower clock rate. Many modern FPGAs support the JESD204B standard interface, which accepts scalar input at a GHz clock rate and produces a vector of samples at a lower clock rate. FPGA algorithms that use vector input and parallel operations to achieve higher throughput are also referred to as super-sample processing.

This example shows how to design a signal processing application that supports GSPS throughput. These Simulink® models assume that the input data is vectorized by using a JESD204B interface, and is available at a lower clock rate in the FPGA. The algorithm processes four samples at a time. Both models have a polyphase filter bank that consists of a filter and an FFT. The polyphase filter bank technique minimizes leakage and scalloping loss from the FFT. For more information about polyphase filter banks, see “High Resolution Spectral Analysis in MATLAB” (DSP System Toolbox).

The first part of the example uses the Channelizer (DSP HDL Toolbox) block from the DSP HDL Toolbox™ library, configured for a 12-tap filter. The Channelizer block uses the polyphase filter bank technique and automatically chooses data types, applies pipelining, and uses other optimizations for hardware performance and resource use. With this block, you can easily explore variations on your design.

The second part of the example implements a polyphase filter bank that has a 4-tap filter. This second part uses basic Simulink blocks. It shows the polyphase filter bank architecture, and the challenges of filter design for hardware, such as choosing data types and inserting pipeline stages.

Implement Channelizer with 12-Tap Filter

This model uses the Channelizer block, configured with a 12-tap filter, that results in good spectrum performance. Using the Channelizer block from the DSP HDL Toolbox library makes it easy to change design parameters such as coefficients, vector size, and FFT length. The block also automatically shares multipliers, calculates fixed-point data types, and pipelines the filter.

The model uses these workspace variables to configure the FFT and filter. The input vector size for this model is 4 samples. The model uses a 512-point FFT and a 12-tap filter for each band. The number of coefficients for the channelizer is 512 frequency bands times 12 taps per frequency band. Generate all the coefficients by using the `tf(h)` function.

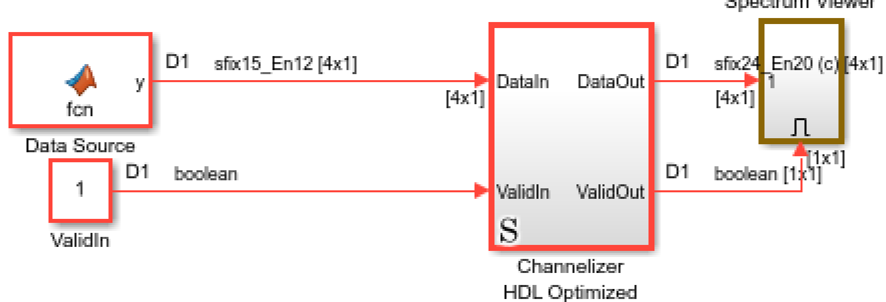
```
simTime = 4000;
FFTLength = 512;
InVect = 4;
hc = dsp.Channelizer;
hc.NumTapsPerBand = 12;
hc.NumFrequencyBands = FFTLength;
hc.StopbandAttenuation = 60;
coef12Tap = tf(hc);
```

The input data consists of two sine waves, 200 kHz and 206.5 kHz. The frequencies are close to each other to illustrate the spectrum resolution of the channelizer.

The `Channelizer HDL Optimized` subsystem contains the Channelizer block and a synchronous State Control block that generates hardware-optimized code for the enable logic within the channelizer.

The block implements pipeline stages around the multipliers so that the logic fits into FPGA DSP blocks, and implements the coefficient banks inside the channelizer with ROM blocks. These hardware-friendly options are the default behavior for the DSP HDL Toolbox blocks.

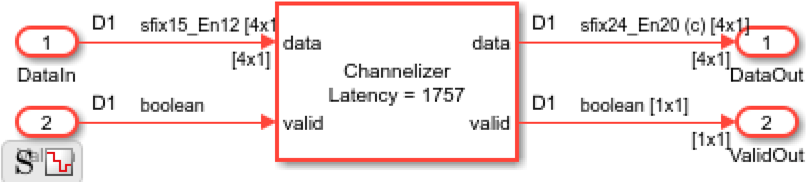
```
modelname = 'PolyphaseFilterBankHDLExample_HDLChannelizer';
open_system(modelname);
set_param(modelname,'SimulationCommand','Update');
```



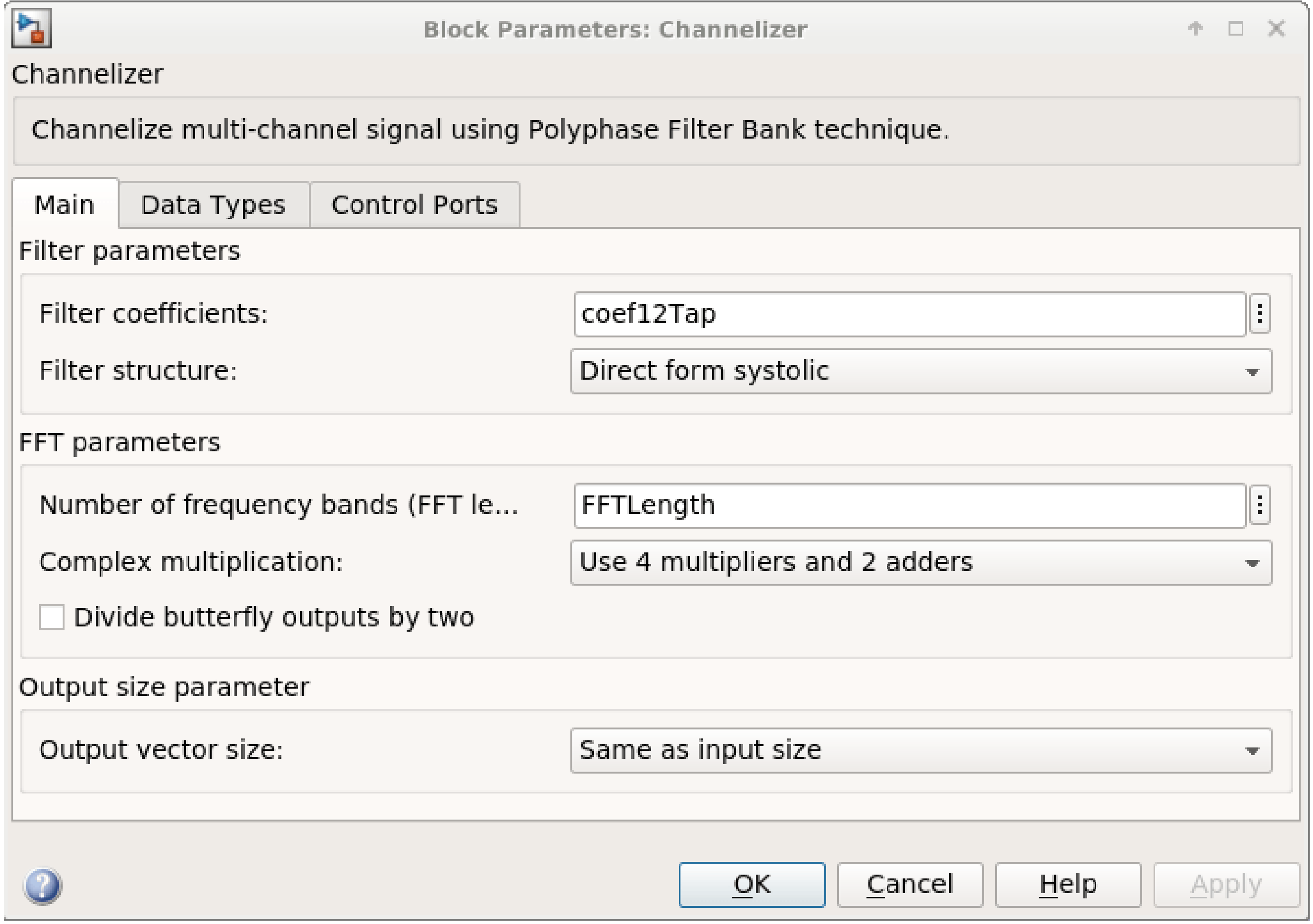
Copyright 2016-2021 The MathWorks, Inc.

This model diagram shows the Channelizer block inside the subsystem.

```
open_system([modelname,'/Channelizer HDL Optimized'])
```



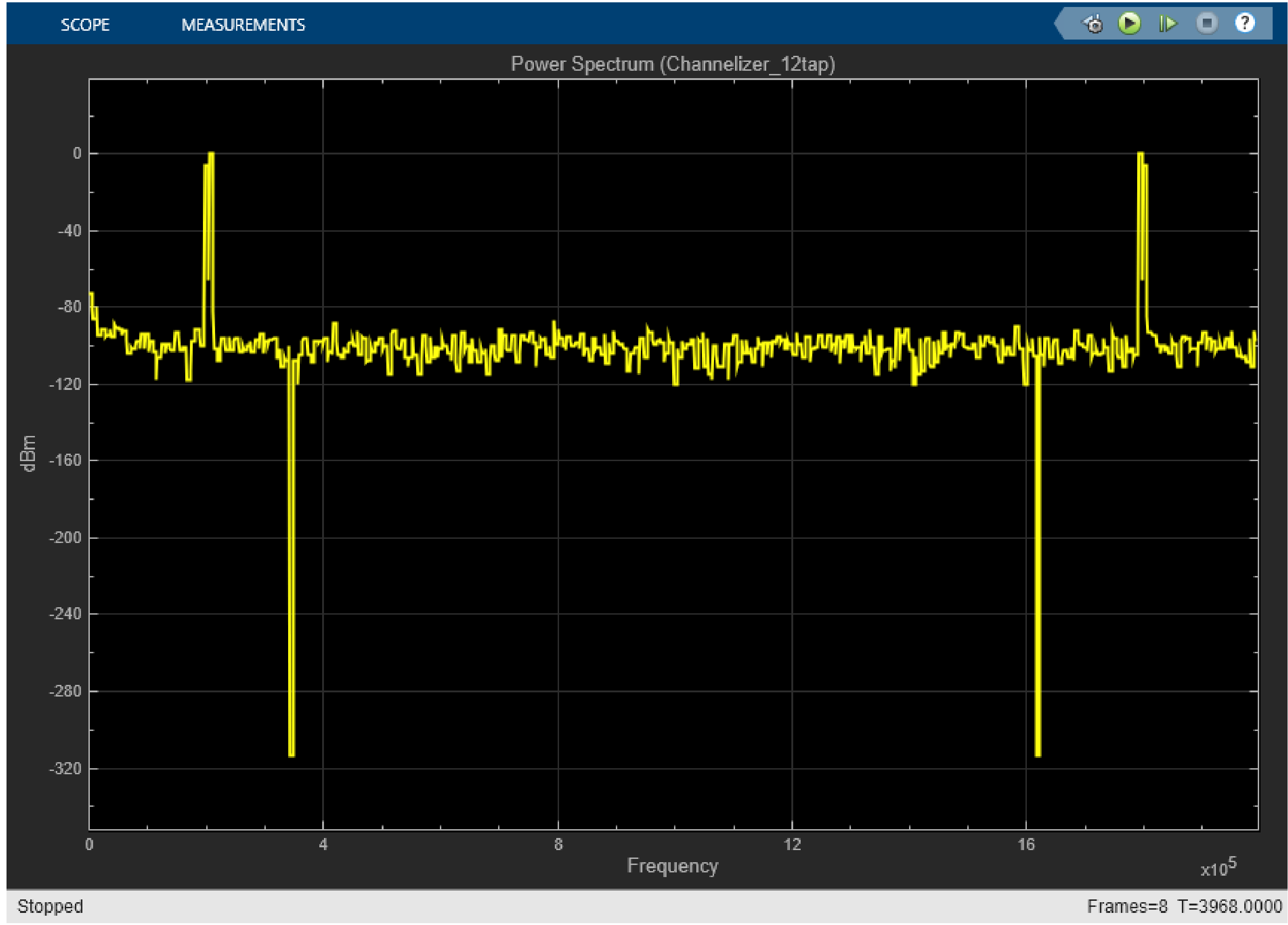
The block has parameters to configure the filter coefficients, FFT length, and other settings that enable you to explore different hardware implementations of the algorithm.



View Simulation Results

To visualize the spectrum result, open the spectrum viewer and run the model.

```
open_system([modelname,'/Spectrum Viewer/Power Spectrum viewer (Channelizer_12tap)']);
sim(modelname);
```



The spectrum viewer shows that the 12-tap filter separates the spectrum of the two signals. Zoom in between 100 kHz and 300 kHz to see where the channelizer detects two peaks. Two peaks is the expected result because the input signal has two frequency components.

Generate HDL Code

You must have the HDL Coder™ product to generate HDL code for this example model. Use this command to generate HDL code.

```
makehdl([modelname,'/Channelizer HDL Optimized']);
```

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior.

```
makehdltb([modelname,'/Channelizer HDL Optimized']);
```

The design was synthesized for Xilinx® Virtex 7 (xc7vx550t-ffg1158, speed grade 2). The design achieves a clock frequency of 361 MHz. At 4 samples per clock, this frequency results in 1.4 GSPS throughput.

The **Minimize clock enable** HDL code generation option is on in this model. The clock enable signal is a global signal, which is not recommended for high speed designs. In the model Configuration

Parameters, choose **HDL Code Generation**, **Global settings**, **Ports**, and then select **Minimize clock enable**. This option is supported when the model is single rate.

The FFT block uses 56 DSP blocks on the FPGA, and the filter uses 48 DSP blocks.

```
 T = table(...
    categorical({'LUT'; 'LUT RAM'; 'FF'; 'BRAM'; 'DSP'}), ...
    [26641; 14585; 24816; 2; 104], ...
    'VariableNames',{'Resource','Usage'});

disp(T);
```

```
    Resource    Usage
    ________    _____

    LUT         26641
    LUT RAM     14585
    FF          24816
    BRAM            2
    DSP           104
```

Implement 4-Tap Polyphase Filter Bank

To show the internal implementation of a polyphase filter bank, the second example model implements a 512-point FFT by using the DSP HDL Toolbox FFT block and a 4-tap filter for each band by using basic Simulink blocks. The 4-tap filter has lower frequency resolution than the 12-tap filter in the first model, but it is easier to see the structure of the filter. These MATLAB® variables configure the blocks in the model. Use the `dsp.Channelizer` (DSP System Toolbox) System object™ to generate the coefficients. The `polyphase` method of the channelizer object generates a 512-by-4 matrix. Each row represents the coefficients for one band. Cast the coefficients to fixed-point types that have the same word length as the input signal.

```
h = dsp.Channelizer;
h.NumTapsPerBand = 4;
h.NumFrequencyBands = FFTLength;
h.StopbandAttenuation = 60;
coef4Tap = fi(polyphase(h),1,15,14,'RoundingMethod','Convergent');
```

The algorithm requires 512 filters (one filter for each band). For a vector input of 4 samples, the model implements four parallel 4-tap filters. Each filter applies 128 sets of coefficients.

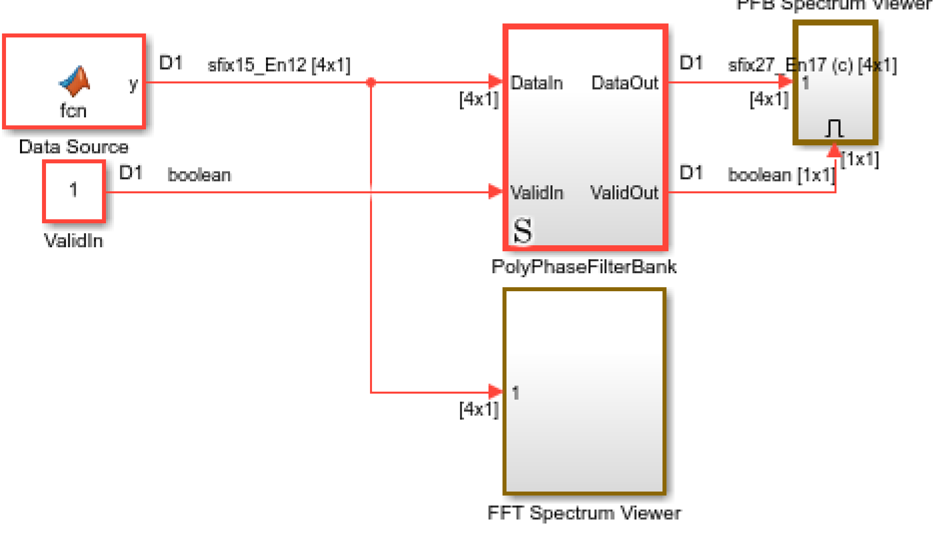
```
ReuseFactor = FFTLength/InVect;
```

These variables configure the model to pipeline the multipliers and the coefficient bank to fit the logic into DSP blocks on the FPGA. Using the DSP blocks enables synthesis to a higher clock rate.

```
Multiplication_PipeLine = 2;
CoefBank_PipeLine       = 1;
```

The input data consists of two sine waves, 200 kHz and 250 kHz. These two frequencies are farther apart than the previous model because the smaller filter has lower spectrum performance. The input and output of the `PolyPhaseFilterBank` subsystem are 4-by-1 vectors.

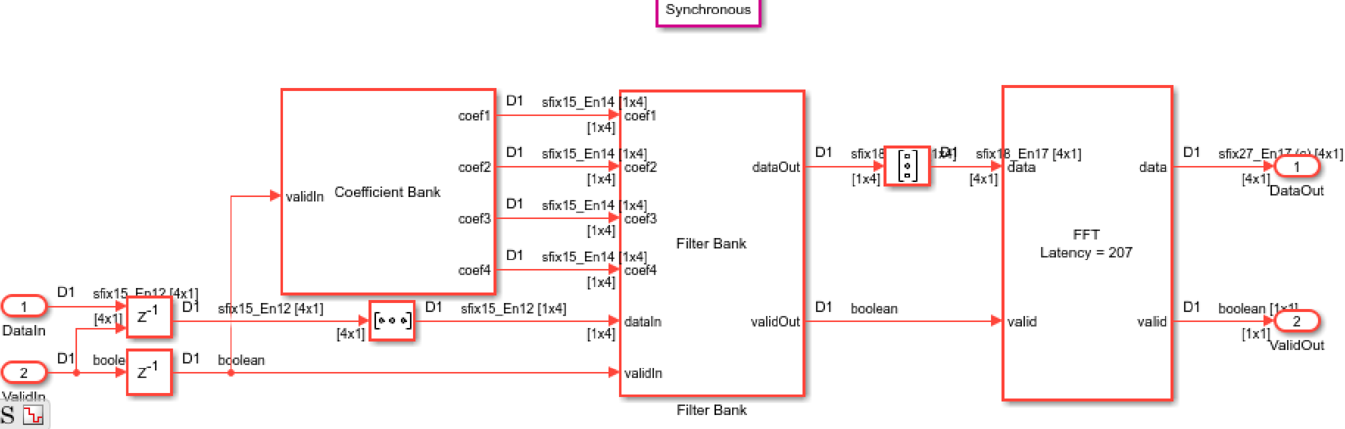
```
modelname = 'PolyphaseFilterBankHDLExample_4tap';
open_system(modelname);
set_param(modelname,'SimulationCommand','Update');
```



Copyright 2016-2021 The MathWorks, Inc.

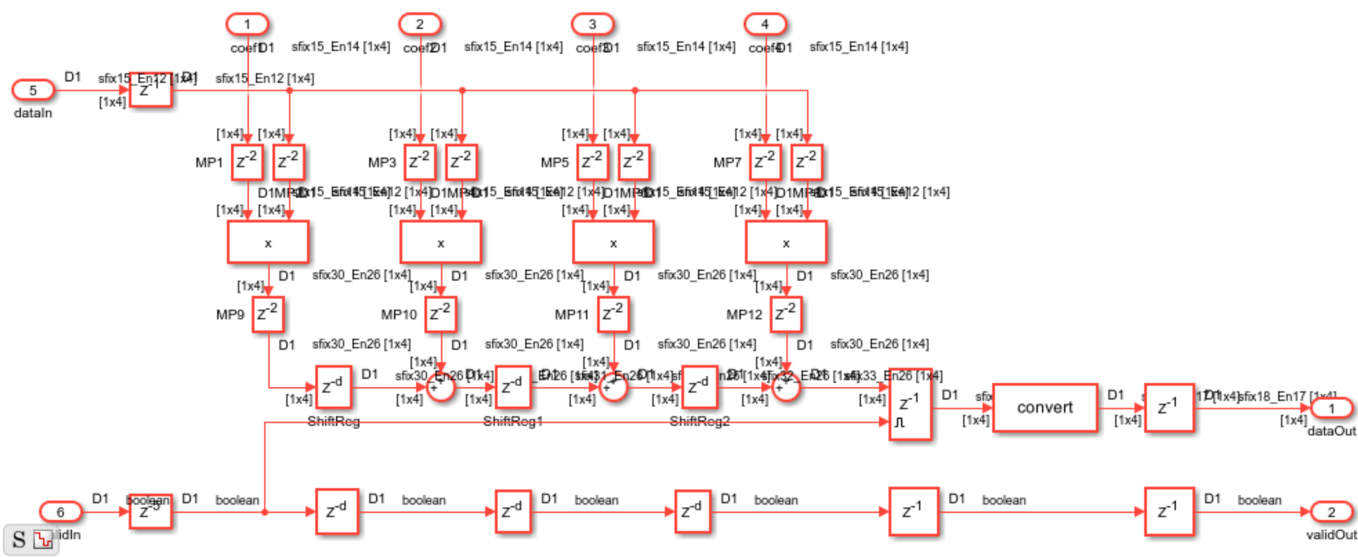
The PolyphaseFilterBank subsystem contains the Coefficient Bank subsystem that rotates over the coefficient sets. The Filter Bank subsystem accepts vectors of coefficients and data and returns a vector of filtered data. The FFT block also accepts and returns a vector of 4 samples, and implements a hardware-optimized architecture.

```
open_system([modelname,'/PolyPhaseFilterBank'])
```



This model diagram shows the pipelined 4-tap filter implementation.

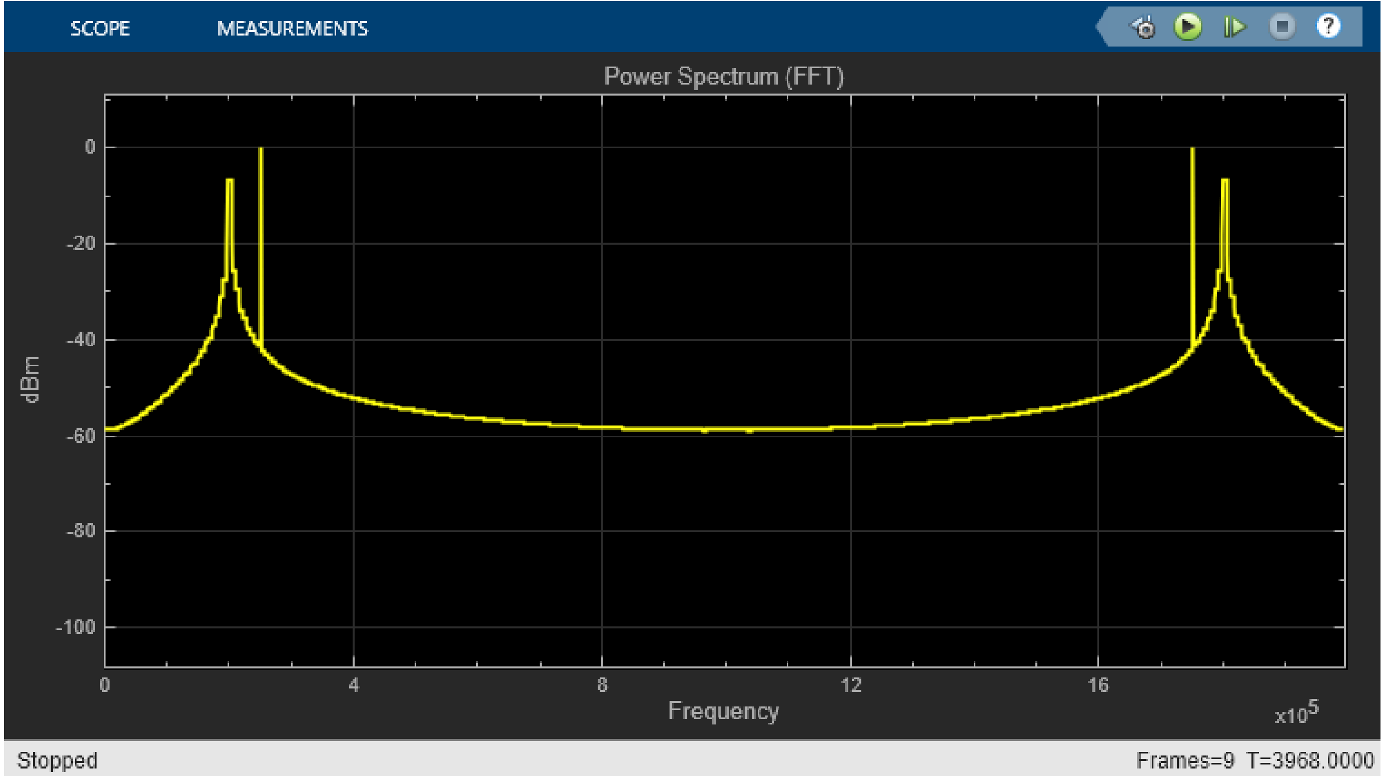
```
open_system([modelname,'/PolyPhaseFilterBank/Filter Bank'])
```

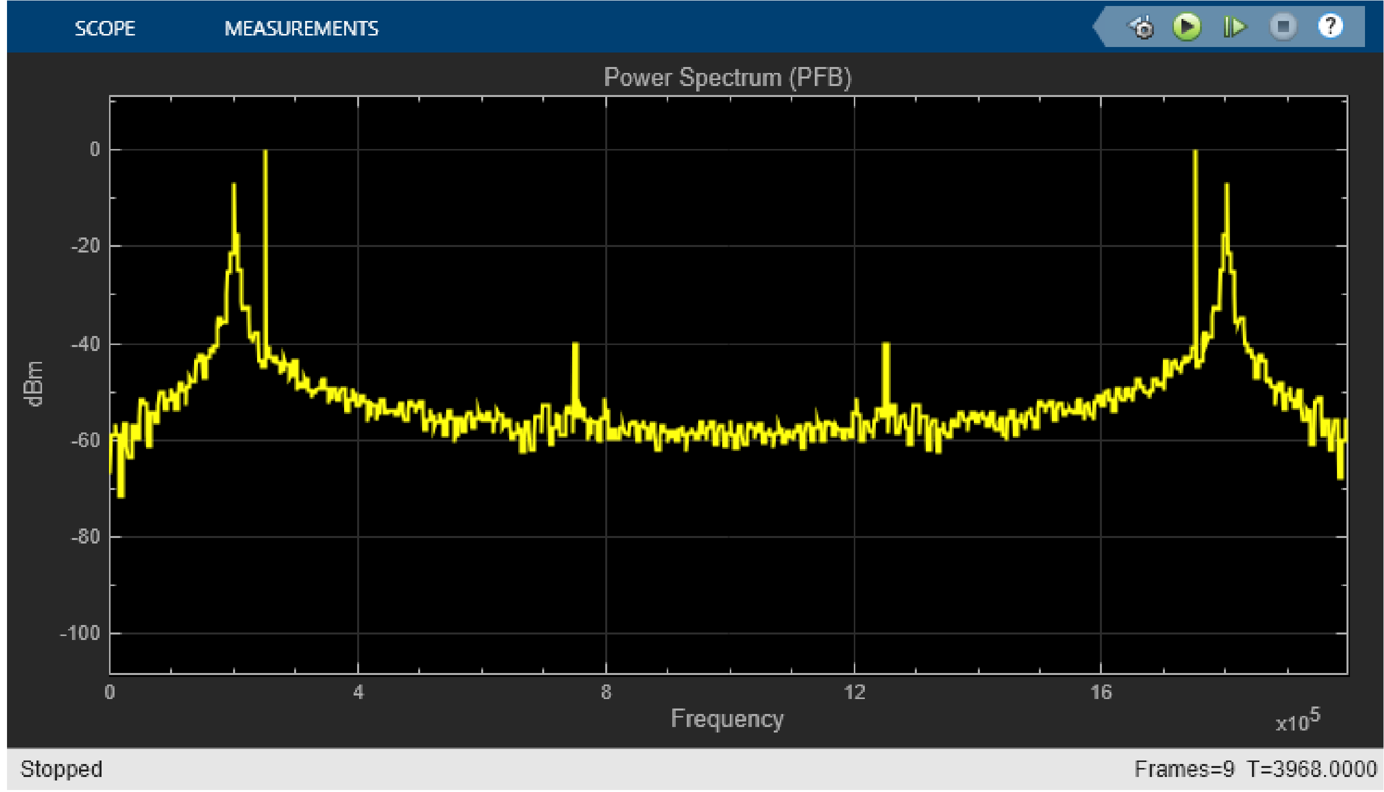


View Simulation Results

To visualize the result of the simulation, open the spectrum viewers and run the model.

```
open_system([modelname,'/FFT Spectrum Viewer/Power Spectrum viewer (FFT)']);
open_system([modelname,'/PFB Spectrum Viewer/Power Spectrum viewer (PFB)']);
sim(modelname);
```





The polyphase filter bank spectrum viewer shows the improvement in the power spectrum and minimization of frequency leakage and scalloping compared with using only an FFT. By comparing the two spectrums, and zooming in between 100 kHz and 300 kHz, you can see that the polyphase filter bank has fewer peaks over –40 dB than the classic FFT.

Considerations for Optimized Hardware

Data Type

Data word length affects both the accuracy of the result and the resources used in the hardware. This 4-tap filter uses full precision. With an input data type of `fixdt(1,15,13)`, the output is `fixdt(1,18,17)`. The absolute values of the filter coefficients are all smaller than 1, so the data does not grow after each multiplication operation. Each addition adds one bit to the data type. To keep the accuracy in the FFT, the data type grows one bit for each stage. This growth makes the twiddle factor multiplication larger at each stage. For many FPGAs, a multiplication size smaller than 18-by-18 is desirable. Because a 512-point FFT has 9 stages, the input of the FFT cannot be more than 11 bits. The first 8 binary digits of the maximum coefficient in this case are zero. Therefore, this example casts the coefficients to `fixdt(1,7,14)` instead of `fixdt(1,15,14)`. Also, the maximum value of the Datatype block output inside the polyphase filter bank has 7 leading zeros after the binary point, so the model casts the filter output to `fixdt(1,11,17)`. These adjustments keep the FFT internal multiplier size smaller than 18-by-18 and save hardware resources.

Design for Speed

The model uses these settings to enable the generated HDL code to synthesize to a faster clock rate.

1. Synchronous State Control block — Implements hardware-friendly enable logic for Delay blocks.

2. Minimize clock enable — Avoids implementing a global clock enable signal that could decrease the synthesized clock rate.
3. Use DSP block in FPGA — Maps multipliers into DSP blocks in the FPGA by including 2 delays before each multiplier and 2 delays after. These pipeline registers cannot have a reset signal. Set the reset type to none for each pipeline by right-clicking the Delay block and selecting **HDL Code** > **HDL Block Properties**, then setting **Reset Type** to `None`.
4. Use ROM in FPGA — Map the combinatorial logic inside the Coefficient MATLAB Function block (inside the `Coefficient Bank` subsystem) to a ROM by adding a register after the block. The delay length is set by `CoefBank_PipeLine`. Set the reset type for these delays to none.

Generate HDL Code

You must have the HDL Coder™ product to generate HDL code for this example model. Use this command to generate HDL code.

```
makehdl([modelname,'PolyPhaseFilterBank']);
```

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior.

```
makehdltb([modelname,'PolyPhaseFilterBank']);
```

When the design is synthesized for Xilinx® Virtex 7 (xc7vx550t-ffg1158, speed grade 2), the design achieves a clock frequency of 324 MHz (before place and route). At 4 samples per clock, this frequency results in 1.3 GSPS throughput.

The FFT block uses 56 DSP blocks in the FPGA, and the filter uses 16 DSP blocks.

```
 T = table(...
    categorical({'LUT'; 'LUT RAM'; 'FF'; 'BRAM'; 'DSP'}), ...
    [14713; 3570; 21514; 2; 72], ...
    'VariableNames',{'Resource','Usage'});

disp(T);
```

```
    Resource    Usage
    ________    _____

    LUT         14713
    LUT RAM      3570
    FF          21514
    BRAM            2
    DSP            72
```

Conclusion

The first part of the example shows how the Channelizer block from the DSP HDL Toolbox library makes it easy to implement an algorithm for hardware, and provides quick exploration of design options.

The second part of the example shows design considerations when implementing filters for hardware without using a hardware-optimized library block, such as choosing data types and inserting pipeline stages. When you use the library blocks from DSP HDL Toolbox, you do not have to consider these

factors yourself. The blocks implement hardware-optimized algorithms that are ready for HDL code generation and deployment to FPGAs or ASICs.

Implement Digital Downconverter for FPGA

This example shows how to design a digital downconverter (DDC) for radio communication applications such as LTE, and generate HDL code.

Introduction

DDCs are widely used in digital communication receivers to convert radio frequency (RF) or intermediate frequency (IF) signals to baseband. The DDC operation shifts the signal to a lower frequency and reduces its sampling rate to facilitate subsequent processing stages. The DDC in this example performs complex frequency translation followed by sample rate conversion using a four-stage filter chain. The example starts by designing the DDC with DSP System Toolbox™ functions in floating point. Then, each stage is converted to fixed point, and used in a Simulink® model that generates synthesizable HDL code. The example uses these two test signals to demonstrate and verify the DDC operation:

- A sinusoid that is modulated onto a 32 MHz IF carrier.
- An LTE downlink signal with a bandwidth of 1.4 MHz modulated onto a 32 MHz IF carrier.

The example compares the signal quality at the output of the floating-point DDC with the signal quality at the output of the fixed-point DDC.

Finally, the example presents an implementation of the filter chain for FPGAs, and synthesis results.

This example uses `DDCTestUtils`, a helper class that contains functions for generating stimulus and analyzing the DDC output. For more information, see the `DDCTestUtils.m` file.

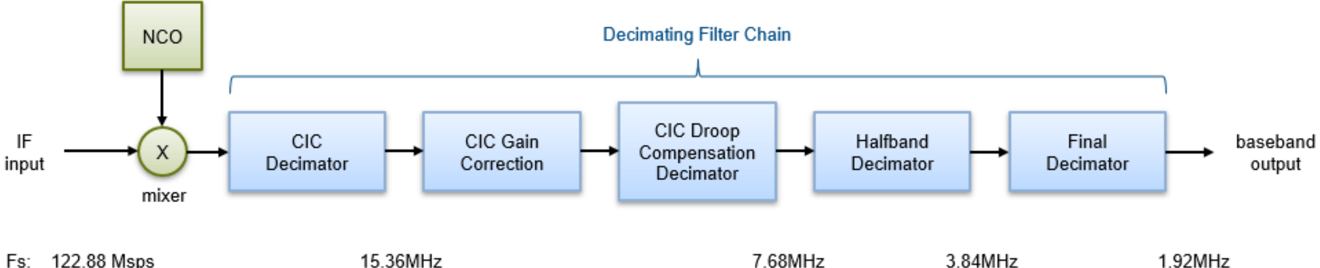
DDC Structure

The DDC consists of a numerically controlled oscillator (NCO), mixer, and decimating filter chain. The filter chain consists of a cascade integrator-comb (CIC) decimator, CIC gain correction, a CIC compensation decimator (FIR), a halfband FIR decimator, and a final FIR decimator.

The overall response of the filter chain is equivalent to that of a single decimation filter with the same specification. However, splitting the filter into multiple decimation stages results in a more efficient design that uses fewer hardware resources.

The CIC decimator provides a large initial decimation factor, which enables subsequent filters to work at lower rates. The CIC compensation decimator improves the spectral response by compensating for the CIC droop while decimating by two. The halfband is an intermediate decimator, and the final decimator implements the precise `Fpass` and `Fstop` characteristics of the DDC. The lower sampling rates near the end of the chain mean the later filters can optimize resource use by sharing multipliers.

This figure shows a block diagram of the DDC.



The sample rate of the input to the DDC is 122.88 Msps, and the output sample rate is 1.92 Msps. These rates give an overall decimation factor of 64. LTE receivers use 1.92 Msps as the typical sampling rate for cell search and master information block (MIB) recovery. The DDC filters are designed to suit this application. The DDC is optimized to run at a clock rate of 122.88 MHz.

DDC Design

This section explains how to design the DDC using floating-point operations and filter-design functions in MATLAB®.

DDC Parameters

This example designs the DDC filter characteristics to meet these specifications for the given input sampling rate and carrier frequency.

```
FsIn = 122.88e6;      % Sampling rate of DDC input
FsOut = 1.92e6;       % Sampling rate of DDC output
Fc = 32e6;            % Carrier frequency
Fpass = 540e3;        % Passband frequency, equivalent to 36x15kHz LTE subcarriers
Fstop = 700e3;        % Stopband frequency
Ap = 0.1;             % Passband ripple
Ast = 60;             % Stopband attenuation
```

CIC Decimator

The first filter stage is a CIC decimator because of its ability to efficiently implement a large decimation factor. The response of a CIC filter is similar to a cascade of moving average filters, but a CIC filter uses no multiplication or division. As a result, the CIC filter has a large DC gain.

```
cicParams.DecimationFactor = 8;
cicParams.DifferentialDelay = 1;
cicParams.NumSections = 3;
cicParams.FsOut = FsIn/cicParams.DecimationFactor;

cicFilt = dsp.CICDecimator(cicParams.DecimationFactor, ...
    cicParams.DifferentialDelay,cicParams.NumSections)
cicGain = gain(cicFilt)
```

```
cicFilt =

  dsp.CICDecimator with properties:

      DecimationFactor: 8
```

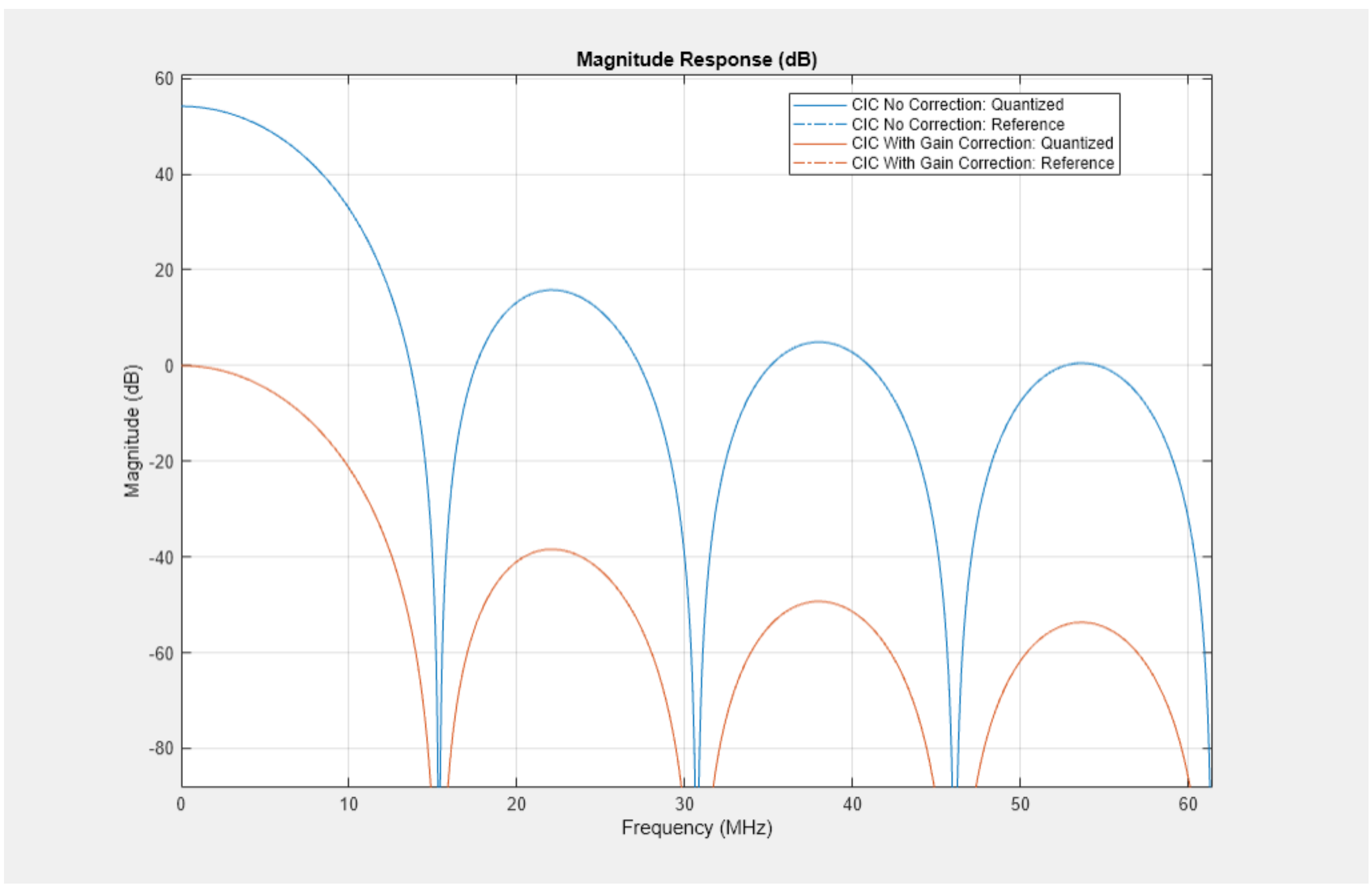
```
     DifferentialDelay: 1
           NumSections: 3
    FixedPointDataType: 'Full precision'


cicGain =

   512
```

Because the CIC gain is a power of two, a hardware implementation can easily correct for the gain factor by using a shift operation. For analysis purposes, the example represents the gain correction in MATLAB with a one-tap `dsp.FIRFilter` System object™.

```
cicGainCorr = dsp.FIRFilter('Numerator',1/cicGain)


cicGainCorr = 

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: 0.0020
    InitialConditions: 0

  Use get to show all properties
```

Display the magnitude response of the CIC filter with and without gain correction by using `fvtool`. For analysis, combine the CIC filter and the gain correction filter into a `dsp.FilterCascade` System object. CIC filters use fixed-point arithmetic internally, so `fvtool` plots both the quantized and unquantized responses.

```
ddcPlots.cicDecim = fvtool(...
    cicFilt, ...
    dsp.FilterCascade(cicFilt,cicGainCorr), ...
    'Fs',[FsIn,FsIn]);
legend(ddcPlots.cicDecim, ...
    'CIC No Correction', ...
    'CIC With Gain Correction');
```



CIC Droop Compensation Filter

Because the magnitude response of the CIC filter has a significant *droop* within the passband region, the example uses a FIR-based droop compensation filter to flatten the passband response. The droop compensator has the same properties as the CIC decimator. This filter implements decimation by a factor of two, so you must also specify bandlimiting characteristics for the filter. Use the `design` function to return a filter System object with the specified characteristics.

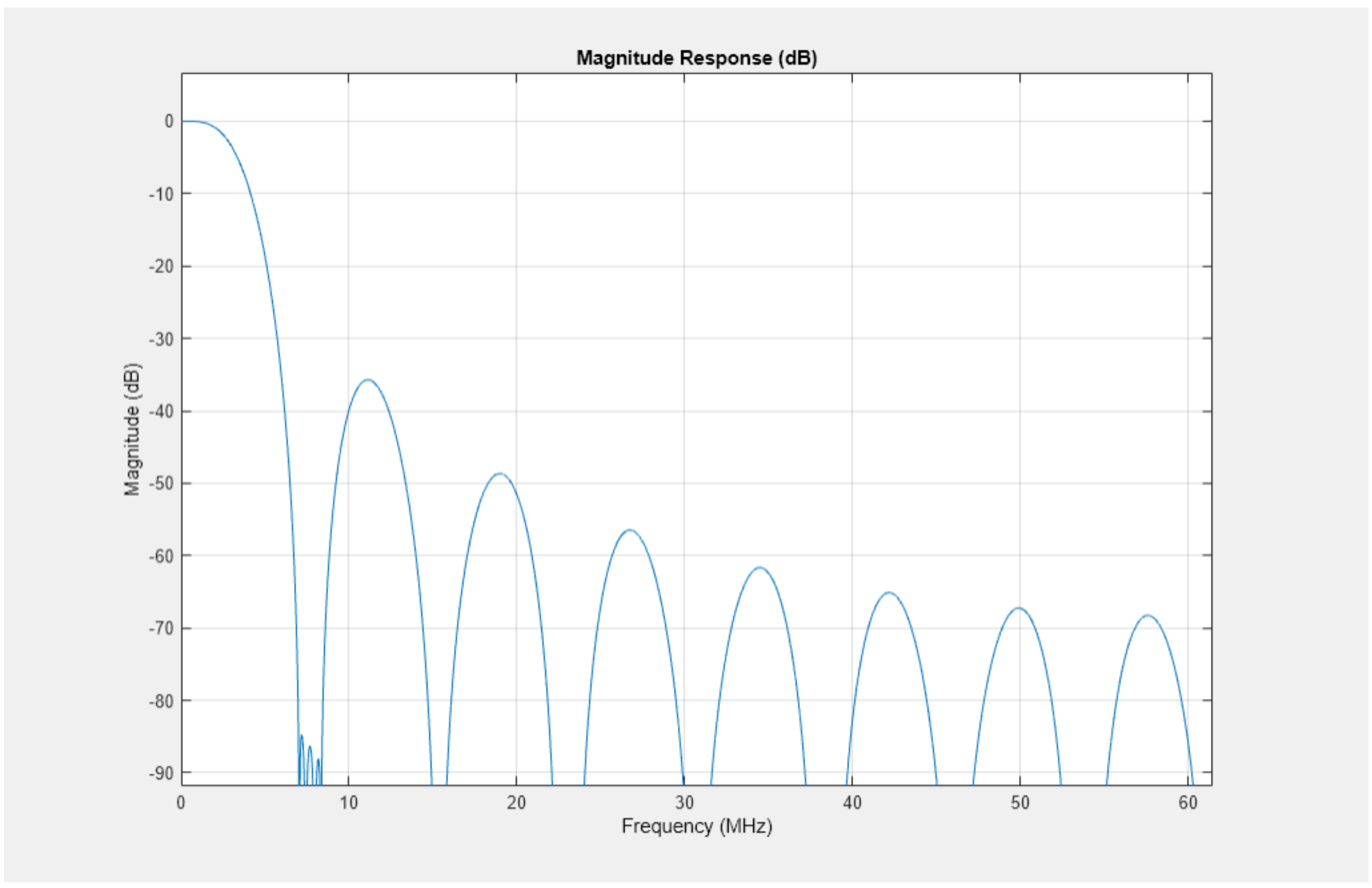
```
compParams.R = 2;                              % CIC compensation decimation factor
compParams.Fpass = Fstop;                      % CIC compensation passband frequency
compParams.FsOut = cicParams.FsOut/compParams.R; % New sampling rate
compParams.Fstop = compParams.FsOut - Fstop;     % CIC compensation stopband frequency
compParams.Ap = Ap;                            % Same passband ripple as overall filter
compParams.Ast = Ast;                          % Same stopband attenuation as overall filter

compSpec = fdesign.decimator(compParams.R,'ciccomp', ...
    cicParams.DifferentialDelay, ...
    cicParams.NumSections, ...
    cicParams.DecimationFactor, ...
    'Fp,Fst,Ap,Ast', ...
    compParams.Fpass,compParams.Fstop,compParams.Ap,compParams.Ast, ...
    cicParams.FsOut);
compFilt = design(compSpec,'SystemObject',true)
```

```
compFilt =
```

```
dsp.FIRDecimator with properties:

 Main
  DecimationFactor: 2
   NumeratorSource: 'Property'
         Numerator: [-0.0398 -0.0126 0.2901 0.5258 0.2901 -0.0126 -0.0398]
         Structure: 'Direct form'

Use get to show all properties
```

Plot the combined response of the CIC filter (with gain correction) and droop compensation.

```
ddcPlots.cicComp = fvtool(...
    dsp.FilterCascade(cicFilt,cicGainCorr,compFilt), ...
    'Fs',FsIn,'Legend','off');
```



Halfband Decimator

The halfband filter provides efficient decimation by two. Halfband filters are efficient because approximately half of their coefficients are equal to zero, and those multipliers are excluded from the hardware implementation.

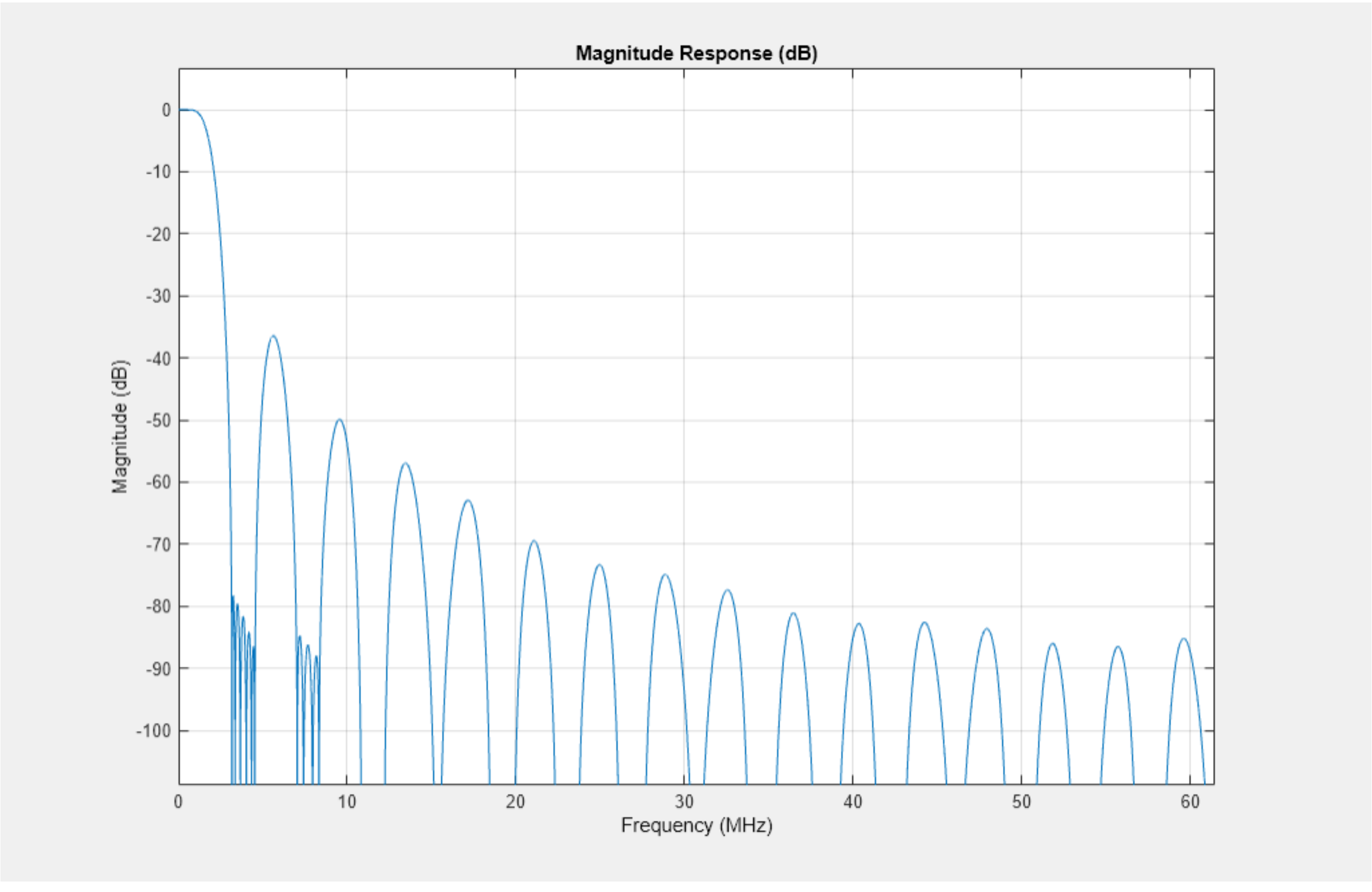
```
hbParams.FsOut = compParams.FsOut/2;
hbParams.TransitionWidth = hbParams.FsOut - 2*Fstop;
```

```
hbParams.StopbandAttenuation = Ast;

hbSpec = fdesign.decimator(2,'halfband',...
    'Tw,Ast', ...
    hbParams.TransitionWidth, ...
    hbParams.StopbandAttenuation, ...
    compParams.FsOut);
hbFilt = design(hbSpec,'SystemObject',true)
```

```
hbFilt = 

  dsp.FIRDecimator with properties:

   Main
    DecimationFactor: 2
     NumeratorSource: 'Property'
           Numerator: [0.0089 0 -0.0565 0 0.2977 0.5000 ... ] (1x11 double)
           Structure: 'Direct form'

  Use get to show all properties
```

Plot the response of the DDC up to the halfband filter output.

```
ddcPlots.halfbandFIR = fvtool(...
    dsp.FilterCascade(cicFilt,cicGainCorr,compFilt,hbFilt), ...
    'Fs',FsIn,'Legend','off');
```



Final FIR Decimator

The final FIR implements the detailed passband and stopband characteristics of the DDC. This filter has more coefficients than the earlier FIR filters, but because it operates at a lower sampling rate it can use resource sharing for an efficient hardware implementation.

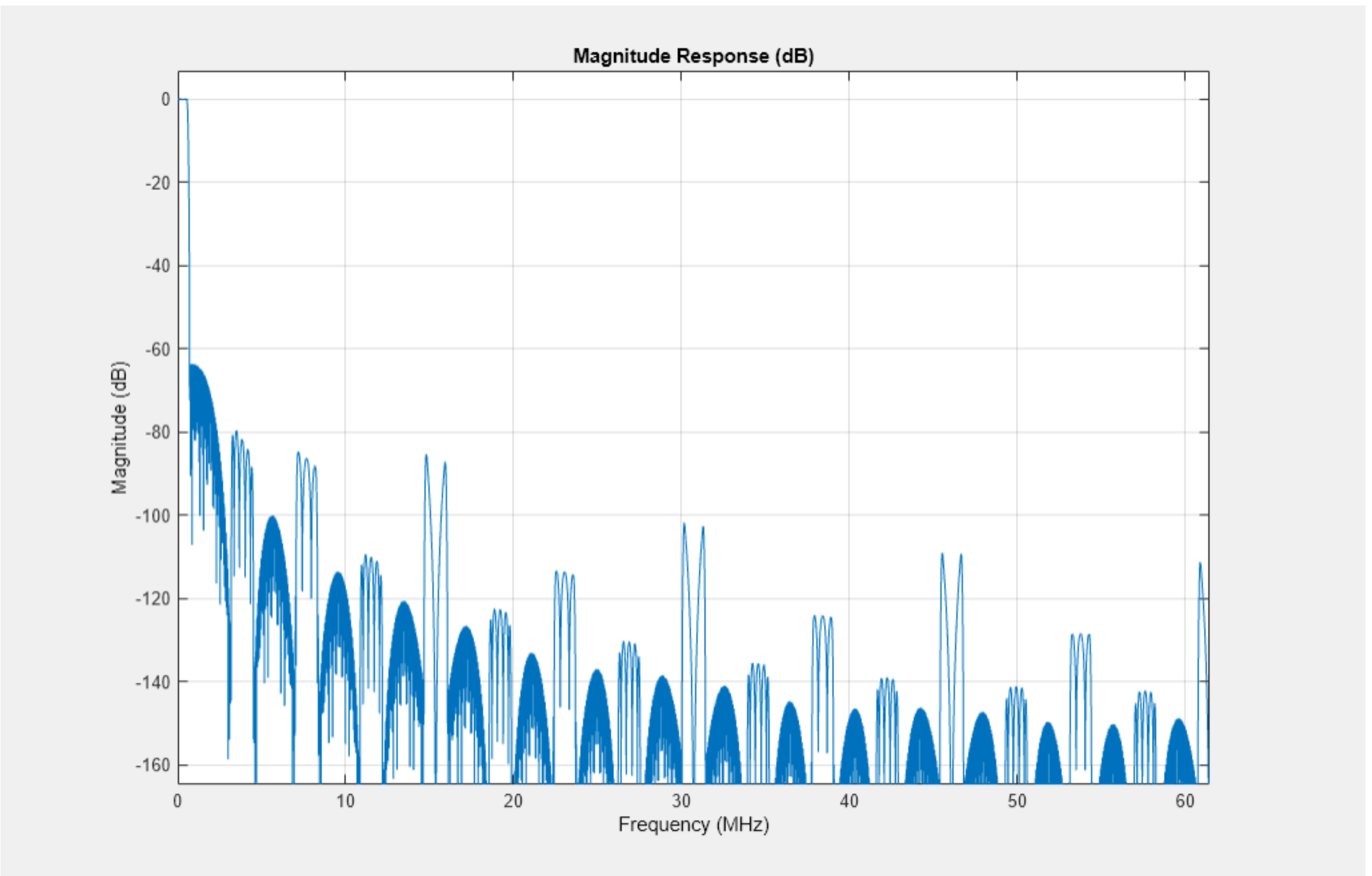
Add 3 dB of headroom to the stopband attenuation so that the DDC still meets the specification after fixed-point quantization. This value was found empirically by using `fvtool`.

```
finalSpec = fdesign.decimator(2,'lowpass', ...
    'Fp,Fst,Ap,Ast',Fpass,Fstop,Ap,Ast+3,hbParams.FsOut);
finalFilt = design(finalSpec,'equiripple','SystemObject',true)
```

```
finalFilt = 

  dsp.FIRDecimator with properties:

   Main
    DecimationFactor: 2
     NumeratorSource: 'Property'
           Numerator: [9.3365e-04 0.0013 9.3466e-04 ... ] (1x70 double)
           Structure: 'Direct form'

  Use get to show all properties
```

Visualize the overall magnitude response of the DDC.

```
ddcFilterChain           = dsp.FilterCascade(cicFilt,cicGainCorr,compFilt,hbFilt,finalFilt);
ddcPlots.overallResponse = fvtool(ddcFilterChain,'Fs',FsIn,'Legend','off');
```



Fixed-Point Conversion

The frequency response of the floating-point DDC filter chain now meets the specification. Next, quantize each filter stage to use fixed-point types and analyze them to confirm that the filter chain still meets the specification.

Filter Quantization

This example uses 16-bit coefficients, which are sufficient to meet the specification. Using fewer than 18 bits for the coefficients minimizes the number of DSP blocks that are required for an FPGA implementation. The input to the DDC filter chain is 16-bit data with 15 fractional bits. The filter outputs are 18-bit values, which provide extra headroom and precision in the intermediate signals.

For the CIC decimator, choosing the `'Minimum section word lengths'` fixed-point data type option automatically optimizes the internal wordlengths based on the output wordlength and other CIC parameters.

```
cicFilt.FixedPointDataType = 'Minimum section word lengths';
cicFilt.OutputWordLength = 18;
```

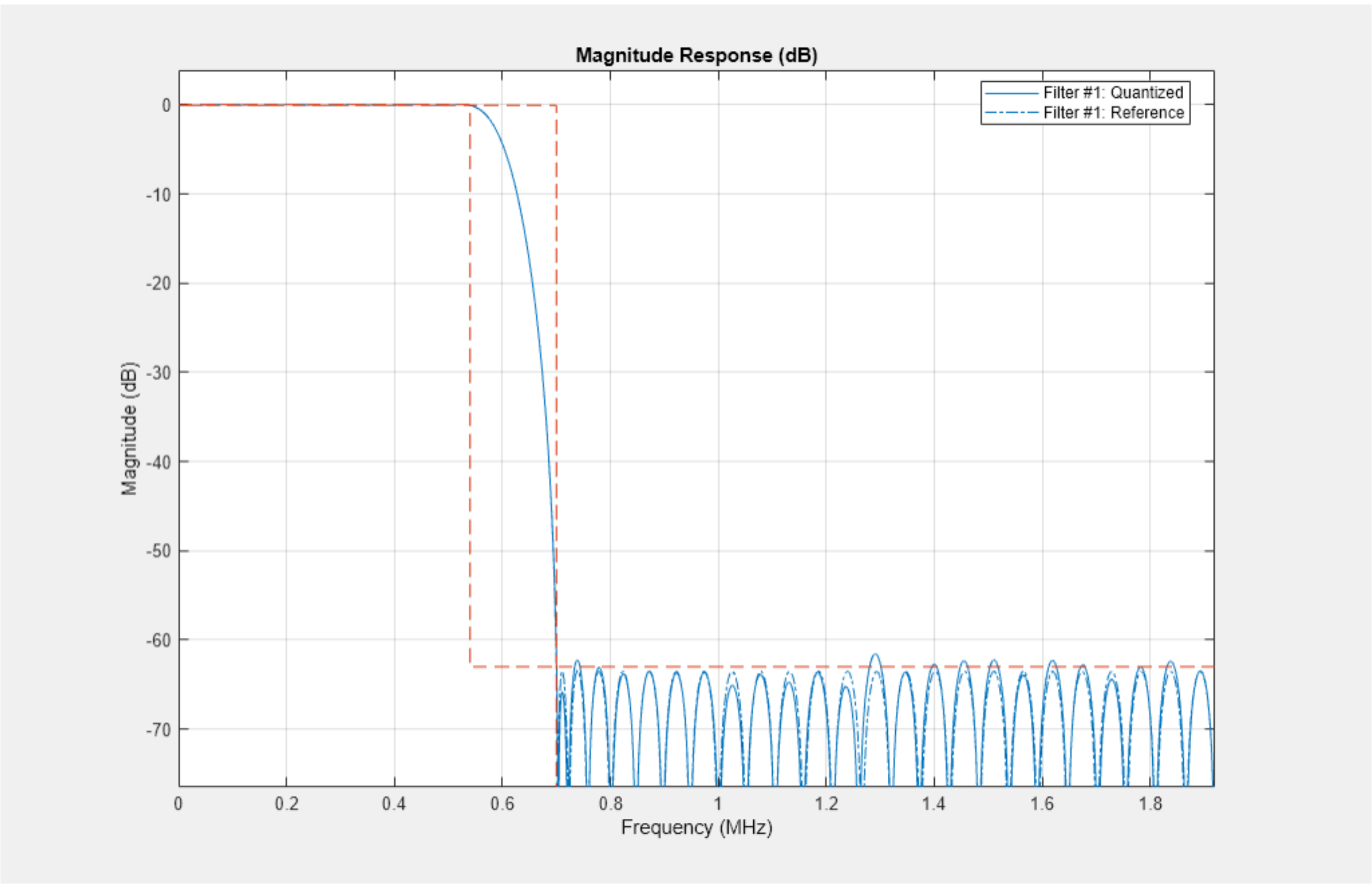
Configure the fixed-point properties of the gain correction and FIR-based System objects. The object uses the default RoundingMethod and OverflowAction property values ('Floor' and 'Wrap' respectively).

```
% CIC Gain Correction
cicGainCorr.FullPrecisionOverride = false;
cicGainCorr.CoefficientsDataType = 'Custom';
cicGainCorr.CustomCoefficientsDataType = numerictype(fi(cicGainCorr.Numerator,1,16));
cicGainCorr.OutputDataType = 'Custom';
cicGainCorr.CustomOutputDataType = numerictype(1,18,16);

% CIC Droop Compensation
compFilt.FullPrecisionOverride = false;
compFilt.CoefficientsDataType = 'Custom';
compFilt.CustomCoefficientsDataType = numerictype([],16,15);
compFilt.ProductDataType = 'Full precision';
compFilt.AccumulatorDataType = 'Full precision';
compFilt.OutputDataType = 'Custom';
compFilt.CustomOutputDataType = numerictype([],18,16);

% Halfband
hbFilt.FullPrecisionOverride = false;
hbFilt.CoefficientsDataType = 'Custom';
hbFilt.CustomCoefficientsDataType = numerictype([],16,15);
hbFilt.ProductDataType = 'Full precision';
hbFilt.AccumulatorDataType = 'Full precision';
hbFilt.OutputDataType = 'Custom';
hbFilt.CustomOutputDataType = numerictype([],18,16);

% FIR
finalFilt.FullPrecisionOverride = false;
finalFilt.CoefficientsDataType = 'Custom';
finalFilt.CustomCoefficientsDataType = numerictype([],16,15);
finalFilt.ProductDataType = 'Full precision';
finalFilt.AccumulatorDataType = 'Full precision';
finalFilt.OutputDataType = 'Custom';
finalFilt.CustomOutputDataType = numerictype([],18,16);
```

Fixed-Point Analysis

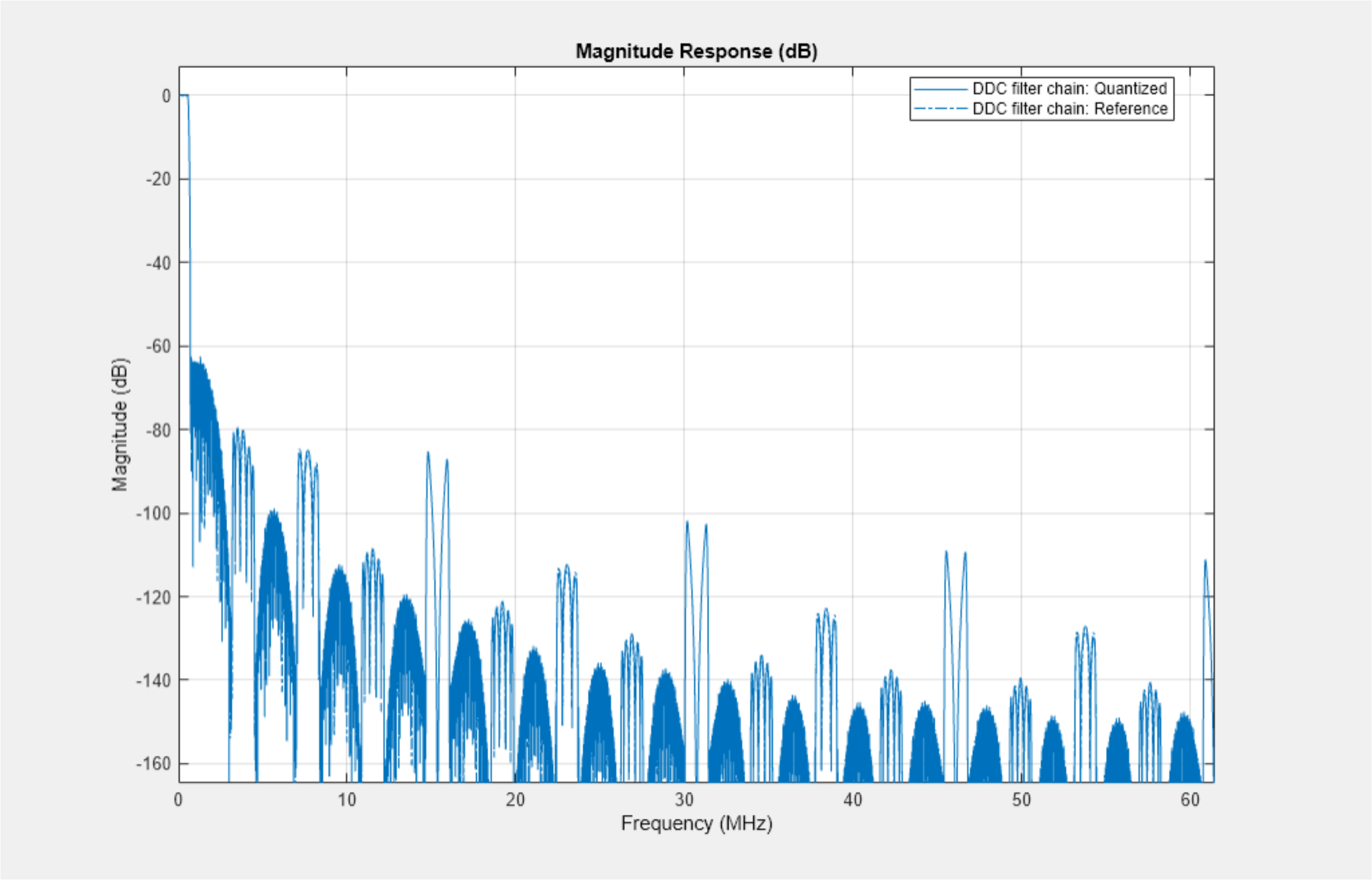
Inspect the quantization effects with fvtool. You can analyze the filters individually or in a cascade. fvtool shows the quantized and unquantized (reference) responses overlayed. For example, this figure shows the effect of quantizing the final FIR filter stage.

```
ddcPlots.quantizedFIR = fvtool(finalFilt, ...
    'Fs',hbParams.FsOut,'arithmetic','fixed');
```



Redefine the `ddcFilterChain` cascade object to include the fixed-point properties of the individual filters. Then, use `fvtool` to analyze the entire filter chain and confirm that the quantized DDC still meets the specification.

```
ddcFilterChain = dsp.FilterCascade(cicFilt, ...
    cicGainCorr,compFilt,hbFilt,finalFilt);
ddcPlots.quantizedDDCResponse = fvtool(ddcFilterChain, ...
    'Fs',FsIn,'Arithmetic','fixed');

legend(ddcPlots.quantizedDDCResponse, ...
    'DDC filter chain');
```



HDL-Optimized Simulink Model

The next step in the design flow is to implement the DDC in Simulink using blocks that support HDL code generation.

Model Configuration

The model relies on variables in the MATLAB workspace to configure the blocks and settings. It uses the same filter chain variables defined earlier in the example. Next, define the NCO characteristics and the input signal. The example uses these characteristics to configure the NCO block.

Specify the desired frequency resolution and calculate the number of accumulator bits that are required to achieve the desired resolution. Set the desired spurious free dynamic range, and then define the number of quantized accumulator bits. The NCO uses the quantized output of the accumulator to address the sine lookup table. Also compute the phase increment that the NCO uses to generate the specified carrier frequency. The NCO applies phase dither to those accumulator bits that are removed during quantization.

```
nco.Fd = 1;
nco.AccWL =  nextpow2(FsIn/nco.Fd)+1;
SFDR  = 84;
nco.QuantAccWL = ceil((SFDR-12)/6);
nco.PhaseInc = round((-Fc*2^nco.AccWL)/FsIn);
nco.NumDitherBits = nco.AccWL-nco.QuantAccWL;
```

The input to the DDC comes from the `ddcIn` variable. For now, assign a dummy value for `ddcIn` so that the model can compute its data types. During testing, `ddcIn` provides input data to the model.

```
ddcIn = 0;
```

You can create a sample-based signal by setting up the `FrameSize` to 1, and output each individual sample as it is received. For a higher input sampling frequency or power reducing consideration, this design could also realize frame-based processing, and the `FrameSize` should be modified accordingly. In this case, we're showing a case for the `FrameSize` of 4. Using vector input and implementing parallel FPGA operations to achieve higher throughput is referred to as super-sample processing.

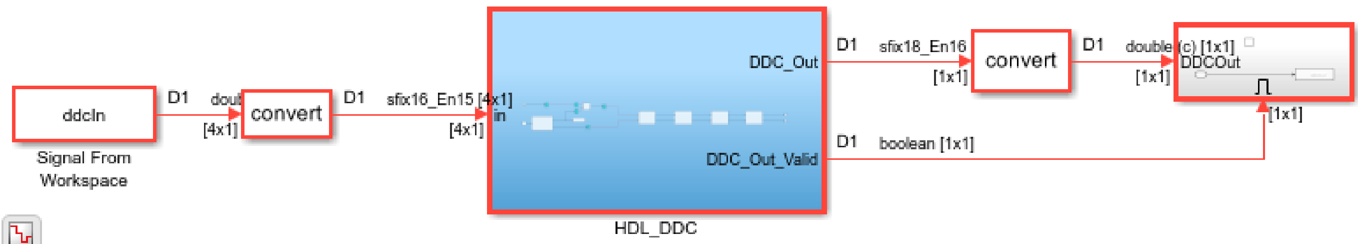
```
FrameSize = 4;
```

Model Structure

This figure shows the top level of the DDC Simulink model. The model imports the `ddcIn` variable from the MATLAB workspace by using a Signal From Workspace block, converts the input signal to 16-bit values, and applies the signal to the DDC. You can generate HDL code from the `HDL_DDC` subsystem.

```
modelName = 'DDCforLTEHDL';
open_system(modelName);
set_param(modelName,'SimulationCommand','Update');
set_param(modelName,'Open','on');
```

Implementation of a Digital Down-Converter for LTE in HDL

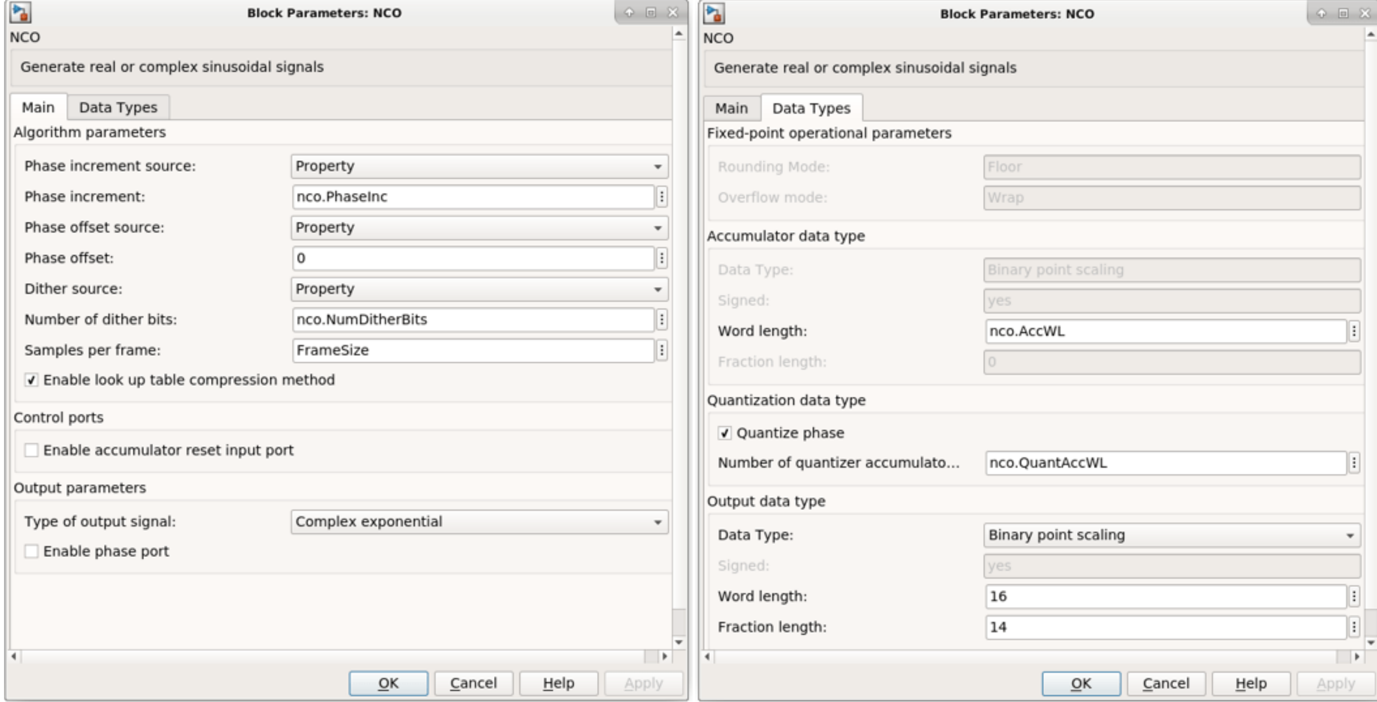


The `HDL_DDC` subsystem implements the DDC filter. First, the NCO block generates a complex phasor at the carrier frequency. This signal goes to a mixer that multiplies the phasor with the input signal. Then, the output of the mixer is passed to the filter chain and decimated to 1.92 Msps.

```
set_param([modelName '/HDL_DDC'],'Open','on');
```

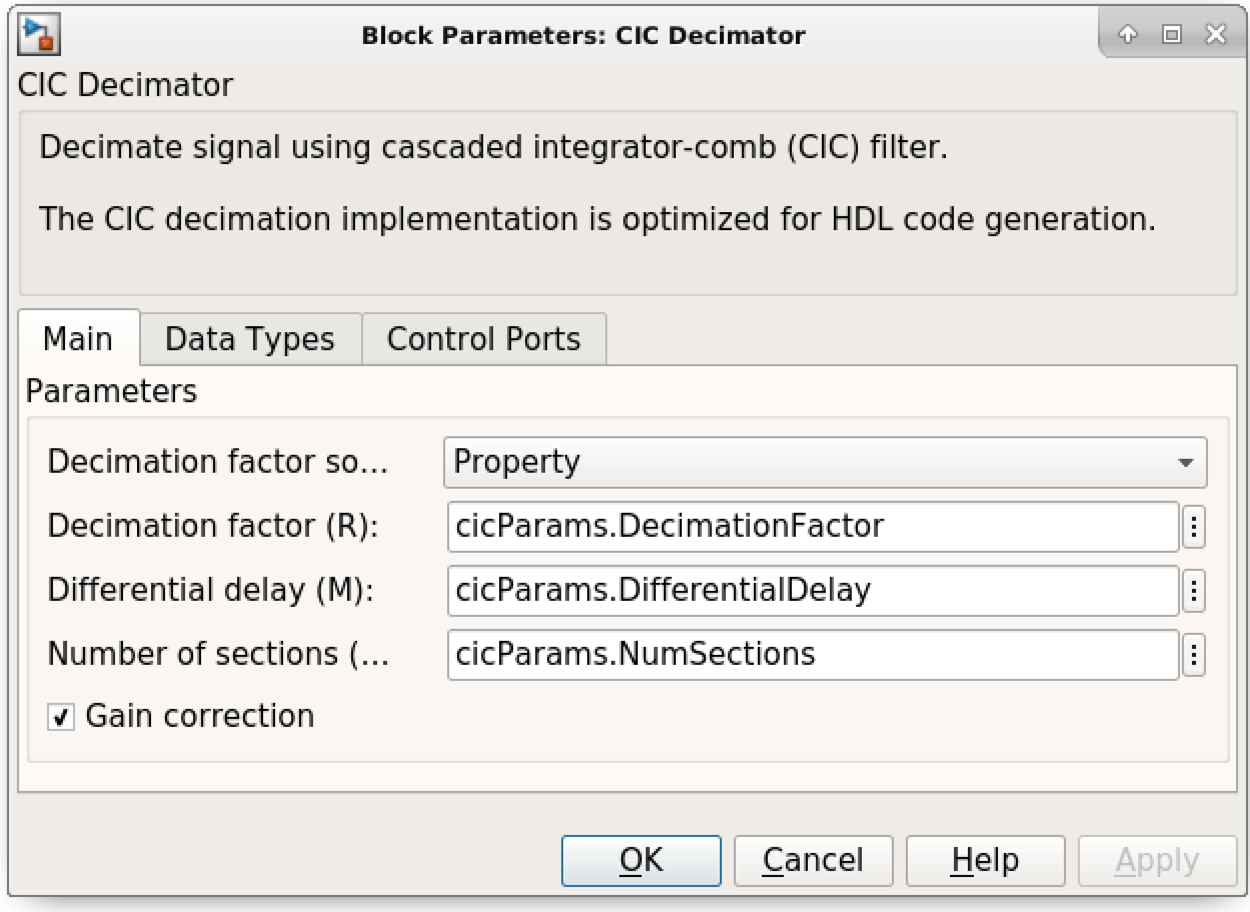
NCO Block Parameters

The NCO block in the model is configured with the parameters defined in the `nco` structure. This figure shows both tabs of the NCO block parameters dialog.



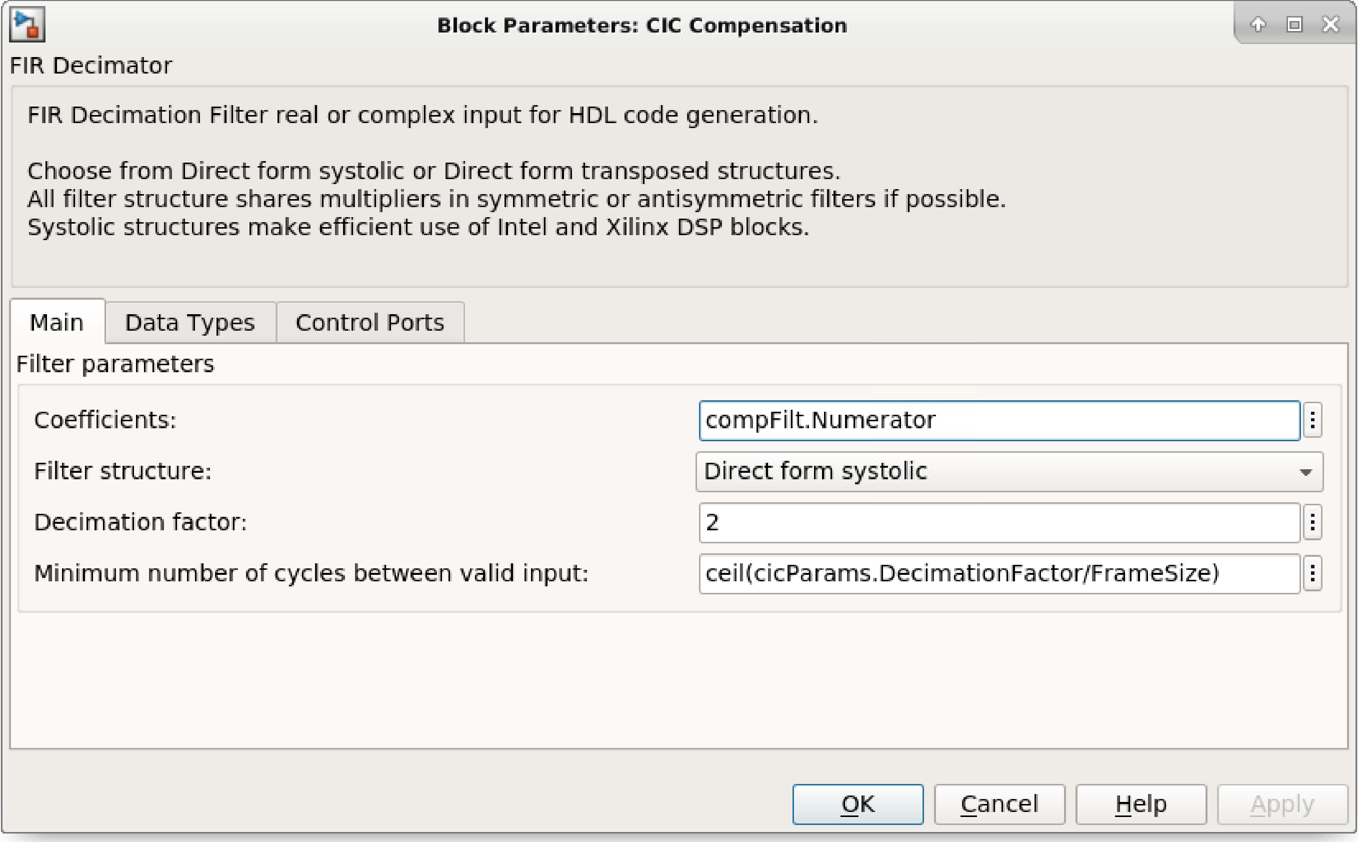
CIC Decimation and Gain Correction

The first filter stage is a CIC Decimator that is implemented with a CIC Decimator block. The block parameters are set to the `cicParams` structure values. To implement the gain correction, the model selects the **Gain correction** parameter. The image shows the block parameters for the CIC Decimator block.



The model configures the filters by using the properties of the corresponding System objects. The CIC compensation, halfband decimation, and final decimation filters operate at effective sample rates that are lower than the clock rate by factors of 8, 16, and 32, respectively. The model implements these sample rates by using the **valid** input signal to indicate which samples are valid at each rate. The signals in the filter chain all have the same Simulink sample time.

The CIC Compensation, Halfband Decimation, and Final Decimation filters are each implemented by an FIR Decimator. By setting the **Minimum number of cycles between valid input samples** parameter, we can use the invalid cycles between input samples. For example,the spacing between every input of CIC Compensation Decimator is `8`, which equals the decimation factor. So the CIC Compensation Decimator has the **Minimum number of cycles between valid input samples** set to `ceil(cicParams.DecimationFactor/FrameSize)`, which equals `2` cycles. The image shows the block parameters for the CIC Compensation Decimation block.



The FIR Decimator block fully reuses the multipliers in time over the number of clock cycles you specify. For `FrameSize` is 4, the CIC Compensation Decimation filter with complex input data would use 4 multipliers. The Halfband Decimation uses 4 multipliers, and the Final Decimation uses 12 multipliers. For `FrameSize` is 1, since the inputs spacing of CIC Compensation Decimation and Halfband Decimation are larger than their filter length, those two decimators only require 2 multipliers. And the Final Decimation needs 4 multipliers at that time.

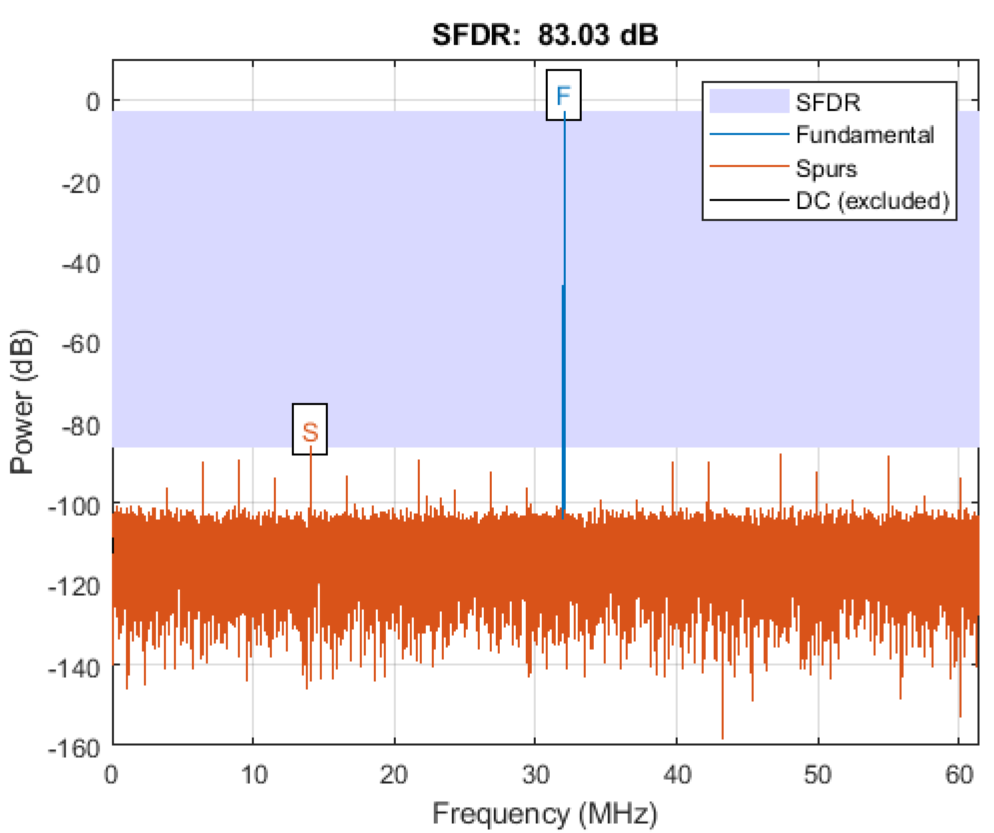
Sinusoid on Carrier Test and Verification

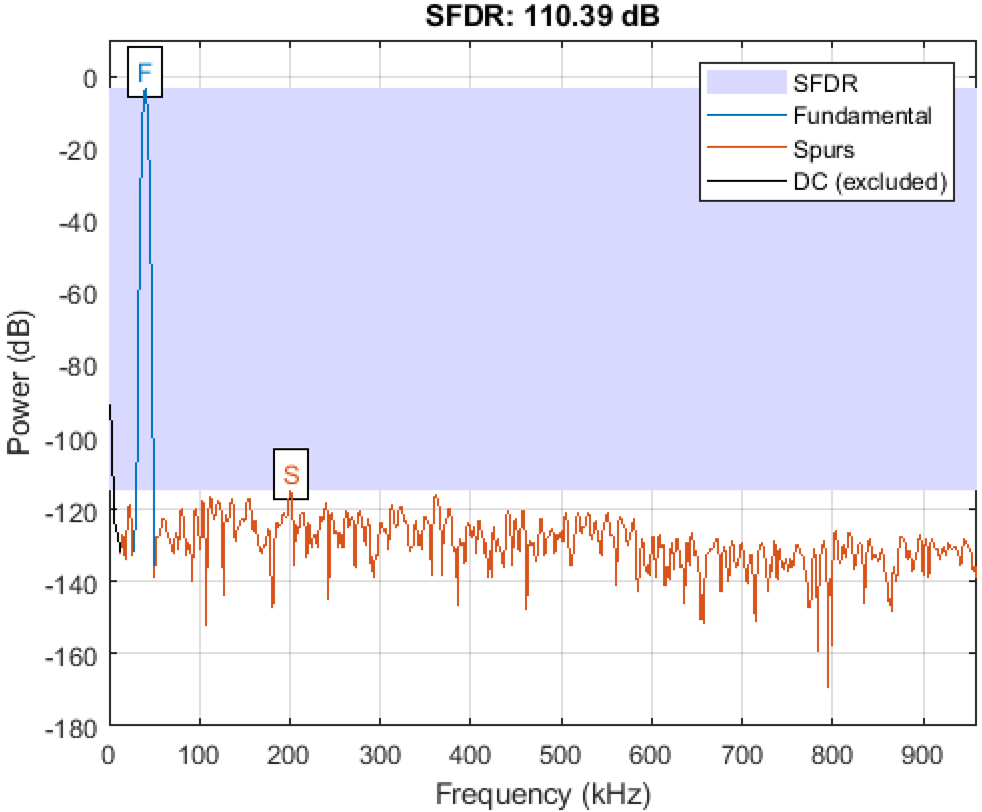
To test the DDC, modulate a 40 kHz sinusoid onto the carrier frequency and pass the modulated sine wave through the DDC. Then, measure the spurious- free dynamic range (SFDR) of the resulting tone and the SFDR of the NCO output. Plot the SFDR of the NCO and the fixed-point DDC output.

```
% Initialize random seed before executing any simulations.
rng(0);

% Generate a 40 kHz test tone, modulated onto the carrier.
ddcIn = DDCTestUtils.GenerateTestTone(40e3,Fc);

% Demodulate the test signal with the floating-point DDC.
ddcOut = DDCTestUtils.DownConvert(ddcIn,FsIn,Fc,ddcFilterChain);
release(ddcFilterChain);

% Demodulate the test signal by running the Simulink model.
out = sim(modelName);
```

```
% Measure the SFDR of the NCO, floating-point DDC outputs, and fixed-point
% DDC outputs.
results.sfdrNCO = sfdr(real(out.ncoOut),FsIn);
results.sfdrFloatDDC = sfdr(real(ddcOut),FsOut);
results.sfdrFixedDDC = sfdr(real(out.ddcFixedOut),FsOut);

disp('SFDR Measurements');
disp(['   Floating-point DDC SFDR: ',num2str(results.sfdrFloatDDC) ' dB']);
disp(['   Fixed-point NCO SFDR: ',num2str(results.sfdrNCO) ' dB']);
disp(['   Optimized fixed-point DDC SFDR: ',num2str(results.sfdrFixedDDC) ' dB']);
fprintf(newline);

% Plot the SFDR of the NCO and fixed-point DDC outputs.
ddcPlots.ncoOutSDFR = figure;
sfdr(real(out.ncoOut),FsIn);

ddcPlots.OptddcOutSFDR = figure;
sfdr(real(out.ddcFixedOut),FsOut);
```

```
SFDR Measurements
   Floating-point DDC SFDR: 291.4188 dB
   Fixed-point NCO SFDR: 83.0306 dB
   Optimized fixed-point DDC SFDR: 110.386 dB
```





LTE Signal Test

You can use an LTE test signal to perform more rigorous testing of the DDC. Generate a standard-compliant LTE waveform by using LTE Toolbox™ functions. Then, downconvert the waveform with the DDC model. Use LTE Toolbox functions to measure the error vector magnitude (EVM) of the resulting signals.

```
rng(0);
% Execute this test only if you have the LTE Toolbox product.
if license('test','LTE_Toolbox')

    % Generate a modulated LTE test signal by using the LTE Toolbox functions.
    [ddcIn,sigInfo] = DDCTestUtils.GenerateLTETestSignal(Fc);

    % Downconvert the signal with the floating-point DDC.
    ddcOut = DDCTestUtils.DownConvert(ddcIn,FsIn,Fc,ddcFilterChain);
    release(ddcFilterChain);

    % Downconvert the signal with the Simulink model, then measure and plot the
    % EVM of the floating-point and fixed-point results. Pad the input with zeros
    % to represent propagation latency and return the complete result.
    ddcIn = [ddcIn;zeros(2480*FrameSize,1)];
    out = sim(modelName);

    results.evmFloat = DDCTestUtils.MeasureEVM(sigInfo,ddcOut);
    results.evmFixed = DDCTestUtils.MeasureEVM(sigInfo,out.ddcFixedOut(1:length(ddcOut)));
```

```
    disp('LTE Error Vector Magnitude (EVM) Measurements');
    disp(['   Floating-point DDC RMS EVM: '  num2str(results.evmFloat.RMS*100,3) '%']);
    disp(['   Floating-point DDC Peak EVM: ' num2str(results.evmFloat.Peak*100,3) '%']);
    disp(['   Fixed-point DDC RMS EVM: '     num2str(results.evmFixed.RMS*100,3) '%']);
    disp(['   Fixed-point DDC Peak EVM: '    num2str(results.evmFixed.Peak*100,3) '%']);
    fprintf(newline);

end
```

```
LTE Error Vector Magnitude (EVM) Measurements
   Floating-point DDC RMS EVM: 0.633%
   Floating-point DDC Peak EVM: 2.44%
   Fixed-point DDC RMS EVM: 0.731%
   Fixed-point DDC Peak EVM: 2.69%
```

HDL Code Generation and FPGA Implementation

To generate the HDL code for this example you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `HDL_DDC` subsystem. The DDC was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place-and-route resource utilization results. The design met timing with a clock frequency of 331 MHz.

```
T = table(...
    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}),...
    categorical({'4341'; '383'; '8248'; '2.0'; '36'}),...
    'VariableNames',{'Resource','Usage'})
```

```
T =

  5x2 table

    Resource    Usage
    ________    _____

     LUT        4341 
     LUTRAM     383  
     FF         8248 
     BRAM       2.0  
     DSP        36   
```

Implement Digital Upconverter for FPGA

This example shows how to design a digital upconverter (DUC) for radio communication applications such as LTE, and generate HDL code.

Introduction

DUCs are widely used in digital communication transmitters to convert baseband signals to radio frequency (RF) or intermediate frequency (IF) signals. The DUC operation increases the sample rate of the signal and shifts it to a higher frequency to facilitate subsequent processing stages. The DUC in this example performs sample rate conversion using a four-stage filter chain followed by complex frequency translation. The example starts by designing the DUC with DSP System Toolbox™ functions in floating point. Then, each stage is converted to fixed point, and used in a Simulink® model that generates synthesizable HDL code. The example uses these two test signals to demonstrate and verify the DUC operation:

- A sinusoid that is modulated onto a 32 MHz IF carrier.
- An LTE downlink signal with a bandwidth of 1.4 MHz, modulated onto a 32 MHz IF carrier.

The example downconverts the outputs of the floating-point and fixed-point DUCs, and compares the signal quality of the two outputs.

Finally, the example presents an implementation of the filter chain for FPGAs, and synthesis results.

This example uses `DUCTestUtils`, a helper class that contains functions for generating stimulus and analyzing the DUC output. For more information, see the `DUCTestUtils.m` file.

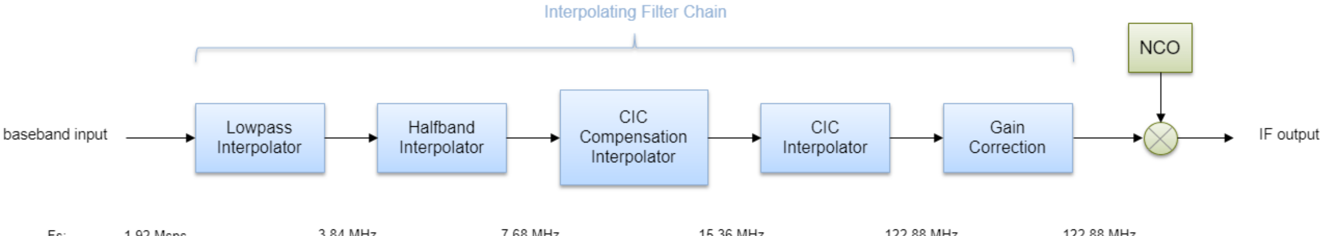
DUC Structure

The DUC consists of an interpolating filter chain, numerically controlled oscillator (NCO), and mixer. The filter chain consists of a lowpass interpolator, halfband interpolator, CIC compensation interpolator (FIR), CIC interpolator, and CIC gain correction.

The overall response of the filter chain is equivalent to that of a single interpolation filter with the same specification. However, splitting the filter into multiple interpolation stages results in a more efficient design that uses fewer hardware resources.

The first lowpass interpolator implements the precise `Fpass` and `Fstop` characteristics of the DUC. The halfband filter is an intermediate interpolator. The lower sampling rates at the beginning of the chain mean the earlier filters can optimize resource use by sharing multipliers. The CIC compensation interpolator improves the spectral response by compensating for the later CIC droop while interpolating by two. The CIC interpolator provides a large interpolation factor, which meets the filter chain upsampling requirements.

This figure shows a block diagram of the DUC.



The sample rate of the input to the DUC is 1.92 Msps, and the output sample rate is 122.88 Msps. These rates give an overall interpolation factor of 64. LTE receivers use 1.92 Msps as the typical sampling rate for cell search and master information block (MIB) recovery. The DUC filters are designed to suit this application. The DUC is optimized to run at a clock rate of 122.88 MHz.

DUC Design

This section explains how to design the DUC using floating-point operations and filter-design functions in MATLAB®. The DUC object enables you to specify several characteristics that define the response of the cascade for the four filters, including passband and stopband frequencies, passband ripple, and stopband attenuation.

DUC Parameters

This example designs the DUC filter characteristics to meet these desired response values for the given input sampling rate and carrier frequency.

```
FsIn = 1.92e6;       % Sampling rate at input to DUC
FsOut = 122.88;      % Sampling rate at the output
Fc = 32e6;           % Carrier frequency
Fpass = 540e3;       % Passband frequency, equivalent to 36*15kHz LTE subcarriers
Fstop = 700e3;       % Stopband frequency
Ap = 0.1;            % Passband ripple
Ast = 60;            % Stopband attenuation
```

First Lowpass Interpolator

This filter interpolates by two, and operates at the lowest sampling rate of the filter chain. The low sample rate means this filter can use resource sharing for an efficient hardware implementation.

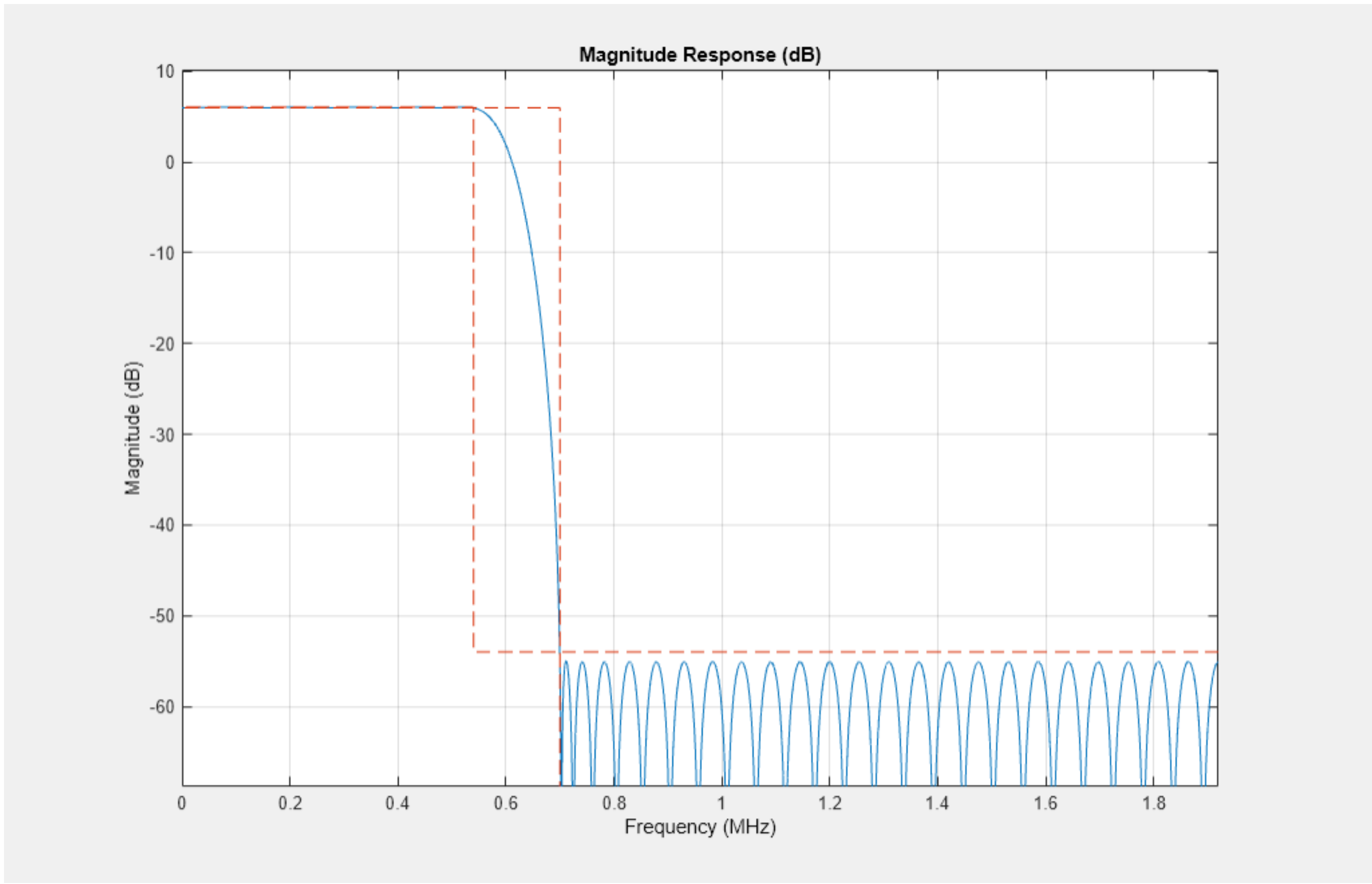
```
lowpassParams.FsIn = FsIn;
lowpassParams.InterpolationFactor = 2;
lowpassParams.FsOut = FsIn*lowpassParams.InterpolationFactor;

lowpassSpec = fdesign.interpolator(lowpassParams.InterpolationFactor, ...
    'lowpass','Fp,Fst,Ap,Ast',Fpass,Fstop,Ap,Ast,lowpassParams.FsOut);
lowpassFilt = design(lowpassSpec,'SystemObject',true)
```

```
lowpassFilt = 

  dsp.FIRInterpolator with properties:

    InterpolationFactor: 2
        NumeratorSource: 'Property'
              Numerator: [0.0020 0.0021 4.9115e-04 -0.0027 ... ] (1x69 double)
```

```
  Use get to show all properties
```

Display the magnitude response of the lowpass filter without gain correction.

```
ducPlots.lowpass = fvtool(lowpassFilt,'Fs',FsIn*2,'Legend','off');
```



Second Halfband Interpolator

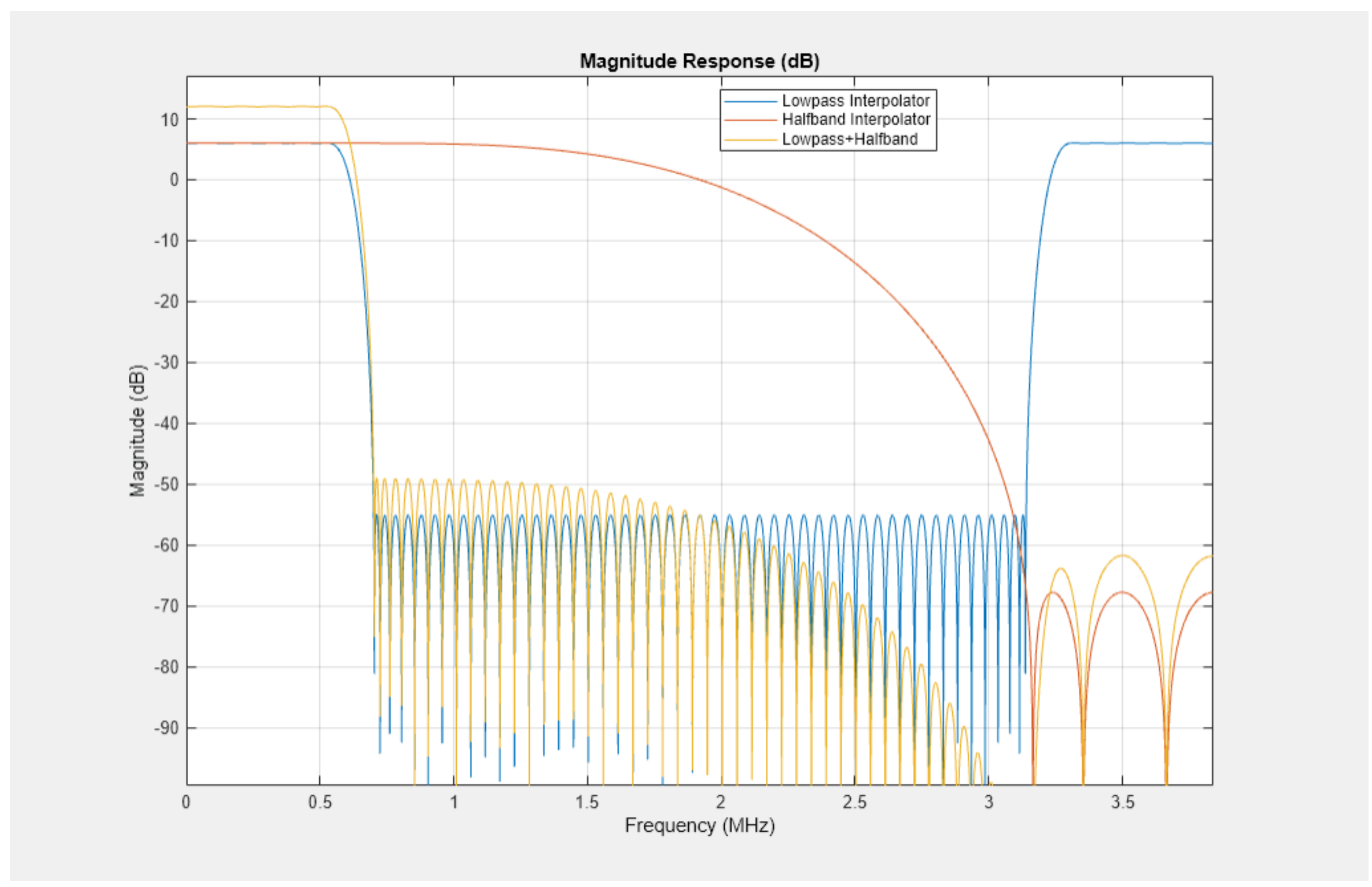
The halfband filter provides efficient interpolation by two. Halfband filters are efficient for hardware because approximately half of their coefficients are equal to zero, and those multipliers are excluded from the hardware implementation.

```
 hbParams.FsIn = lowpassParams.FsOut;
 hbParams.InterpolationFactor = 2;
 hbParams.FsOut = lowpassParams.FsOut*hbParams.InterpolationFactor;
 hbParams.TransitionWidth = hbParams.FsIn - 2*Fstop;
 hbParams.StopbandAttenuation = Ast;

 hbSpec = fdesign.interpolator(hbParams.InterpolationFactor,'halfband', ...
          'TW,Ast', ...
          hbParams.TransitionWidth, ...
          hbParams.StopbandAttenuation, ...
          hbParams.FsOut);

hbFilt = design(hbSpec,'SystemObject',true)
```

```
hbFilt = 

  dsp.FIRInterpolator with properties:

    InterpolationFactor: 2
        NumeratorSource: 'Property'
              Numerator: [0.0178 0 -0.1129 0 0.5953 1 ... ] (1x11 double)

  Use get to show all properties
```

Visualize the magnitude response of the halfband interpolation.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt);
ducPlots.hbFilt = fvtool(lowpassFilt,hbFilt,ducFilterChain, ...
                           'Fs',[FsIn*2,FsIn*4,FsIn*4]);

legend(ducPlots.hbFilt, ...
       'Lowpass Interpolator', ...
       'Halfband Interpolator', ...
       'Lowpass+Halfband');
```



CIC Compensation Interpolator

Because the magnitude response of the last CIC filter has a significant *droop* within the passband region, the example uses an FIR-based droop compensation filter to flatten the passband response. The droop compensator has the same properties as the CIC interpolator. This filter implements interpolation by a factor of two, so you must also specify bandlimiting characteristics for the filter. Also, specify the CIC interpolator properties that are used for this compensation filter as well as the later CIC interpolator.

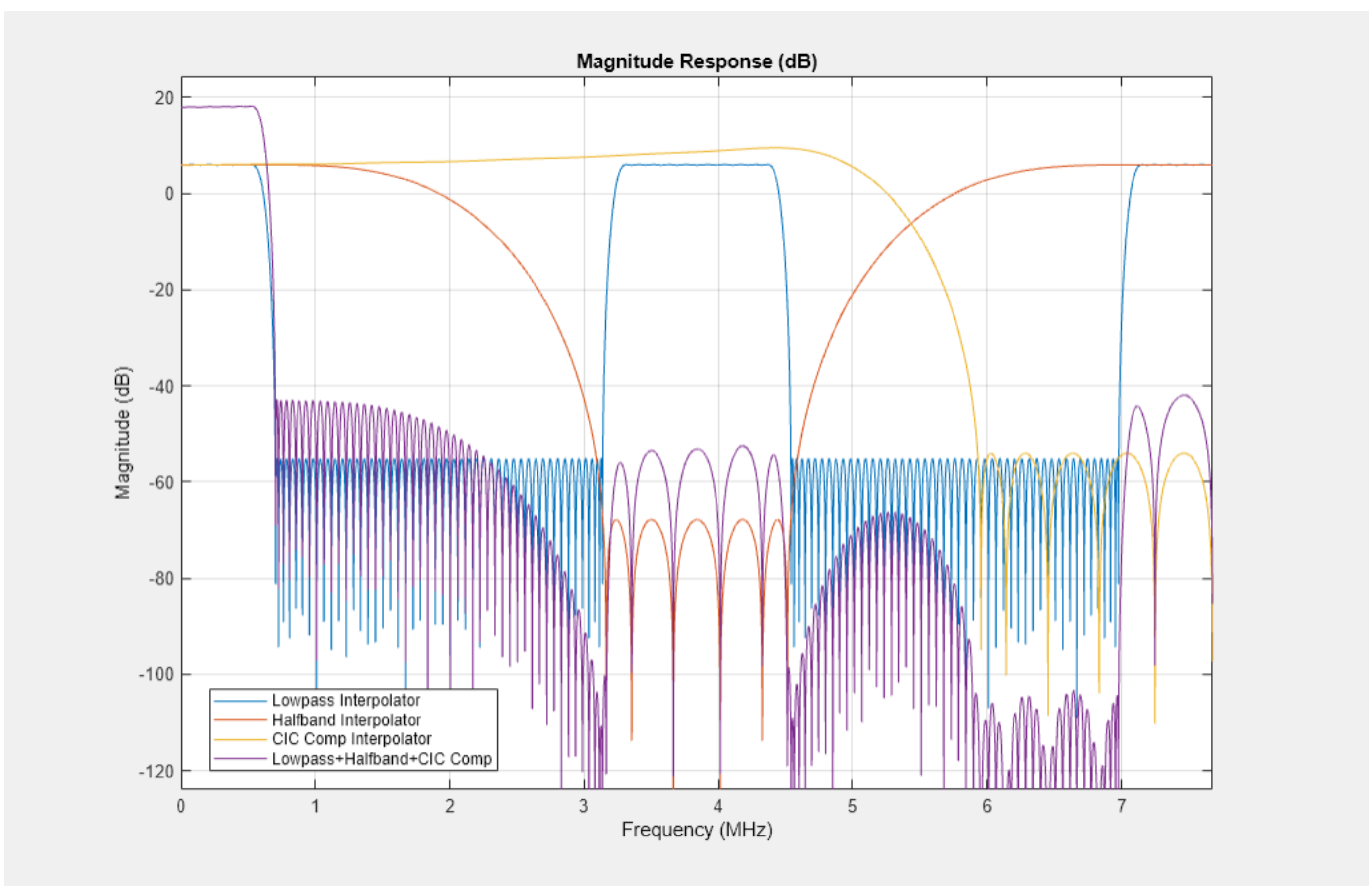
Use the `design` function to return a filter System object with the specified characteristics.

```
compParams.FsIn = hbParams.FsOut;
compParams.InterpolationFactor = 2;                                 % CIC compensation interpola
compParams.FsOut = compParams.FsIn*compParams.InterpolationFactor;  % New sampling rate
compParams.Fpass = 1/2*compParams.FsIn + Fpass;                     % CIC compensation passband 
compParams.Fstop = 1/2*compParams.FsIn + 1/4*compParams.FsIn;       % CIC compensation stopband 
compParams.Ap = Ap;                                                 % Same passband ripple as ove
compParams.Ast = Ast;                                               % Same stopband attenuation a
N = 31;                                                             % 32 tap filter to take advar

cicParams.InterpolationFactor = 8;  % CIC interpolation factor
cicParams.DifferentialDelay = 1;    % CIC interpolator differential delay
cicParams.NumSections = 3;          % CIC interpolator number of integrator and comb sections

compSpec = fdesign.interpolator(compParams.InterpolationFactor,'ciccomp', ...
           cicParams.DifferentialDelay, ...
           cicParams.NumSections, ...
           cicParams.InterpolationFactor, ...
           'N,Fp,Ap,Ast', ...
           N,compParams.Fpass,compParams.Ap,compParams.Ast, ...
           compParams.FsOut);
compFilt = design(compSpec,'SystemObject',true)
```

```
compFilt = 

  dsp.FIRInterpolator with properties:

    InterpolationFactor: 2
        NumeratorSource: 'Property'
              Numerator: [-6.9876e-04 -0.0099 0.0038 0.0134 ... ] (1x32 double)

  Use get to show all properties
```

Plot the response of the CIC compensation interpolator.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt,compFilt);
ducPlots.cicComp = fvtool(lowpassFilt,hbFilt,compFilt,ducFilterChain, ...
                          'Fs',[FsIn*2,FsIn*4,FsIn*8,FsIn*8]);

legend(ducPlots.cicComp, ...
       'Lowpass Interpolator', ...
       'Halfband Interpolator', ...
       'CIC Comp Interpolator', ...
       'Lowpass+Halfband+CIC Comp');
```



CIC Interpolator

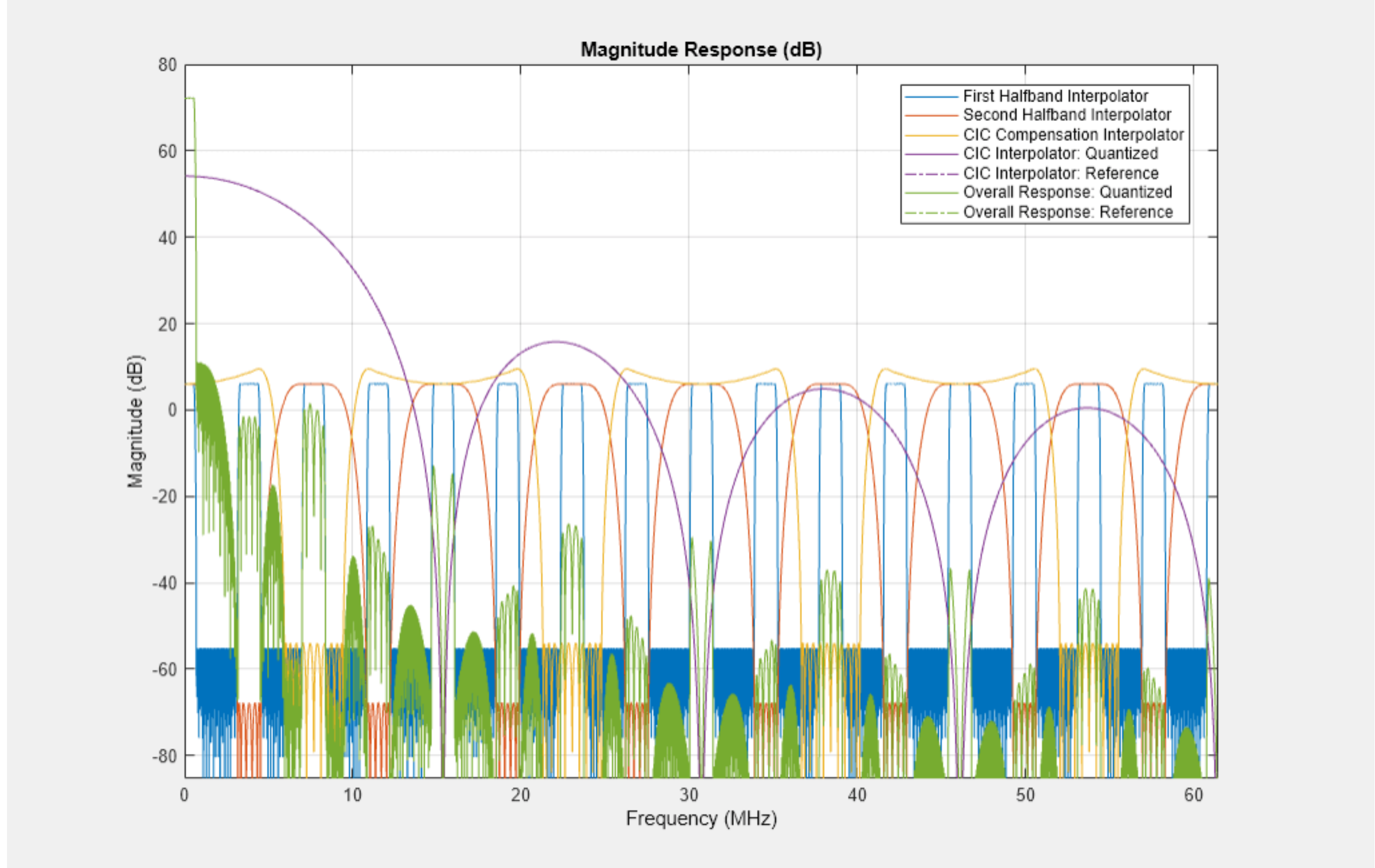
The last filter stage is implemented as a CIC interpolator because of this type of filter's ability to efficiently implement a large decimation factor. The response of a CIC filter is similar to a cascade of moving average filters, but a CIC filter uses no multiplication or division. As a result, the CIC filter has a large DC gain.

```
cicParams.FsIn = compParams.FsOut;
cicParams.FsOut = cicParams.FsIn*cicParams.InterpolationFactor;

cicFilt = dsp.CICInterpolator(cicParams.InterpolationFactor, ...
          cicParams.DifferentialDelay,cicParams.NumSections)
```

```
cicFilt = 

  dsp.CICInterpolator with properties:

    InterpolationFactor: 8
      DifferentialDelay: 1
            NumSections: 3
     FixedPointDataType: 'Full precision'
```

Visualize the magnitude response of the CIC interpolation. CIC filters use fixed-point arithmetic internally, so `fvtool` plots both the quantized and unquantized responses.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt,compFilt,cicFilt);
ducPlots.cicInter = fvtool(lowpassFilt,hbFilt,compFilt,cicFilt,ducFilterChain, ...
                      'Fs',[FsIn*2,FsIn*4,FsIn*8,FsIn*64,FsIn*64]);

legend(ducPlots.cicInter, ...
       'First Halfband Interpolator', ...
       'Second Halfband Interpolator', ...
       'CIC Compensation Interpolator', ...
       'CIC Interpolator',...
       'Overall Response');
```



Every interpolator has a DC gain that is determined by its interpolation factor. The CIC interpolator has a larger gain than other filters. Call the `gain` function to get the gain factor of this filter.

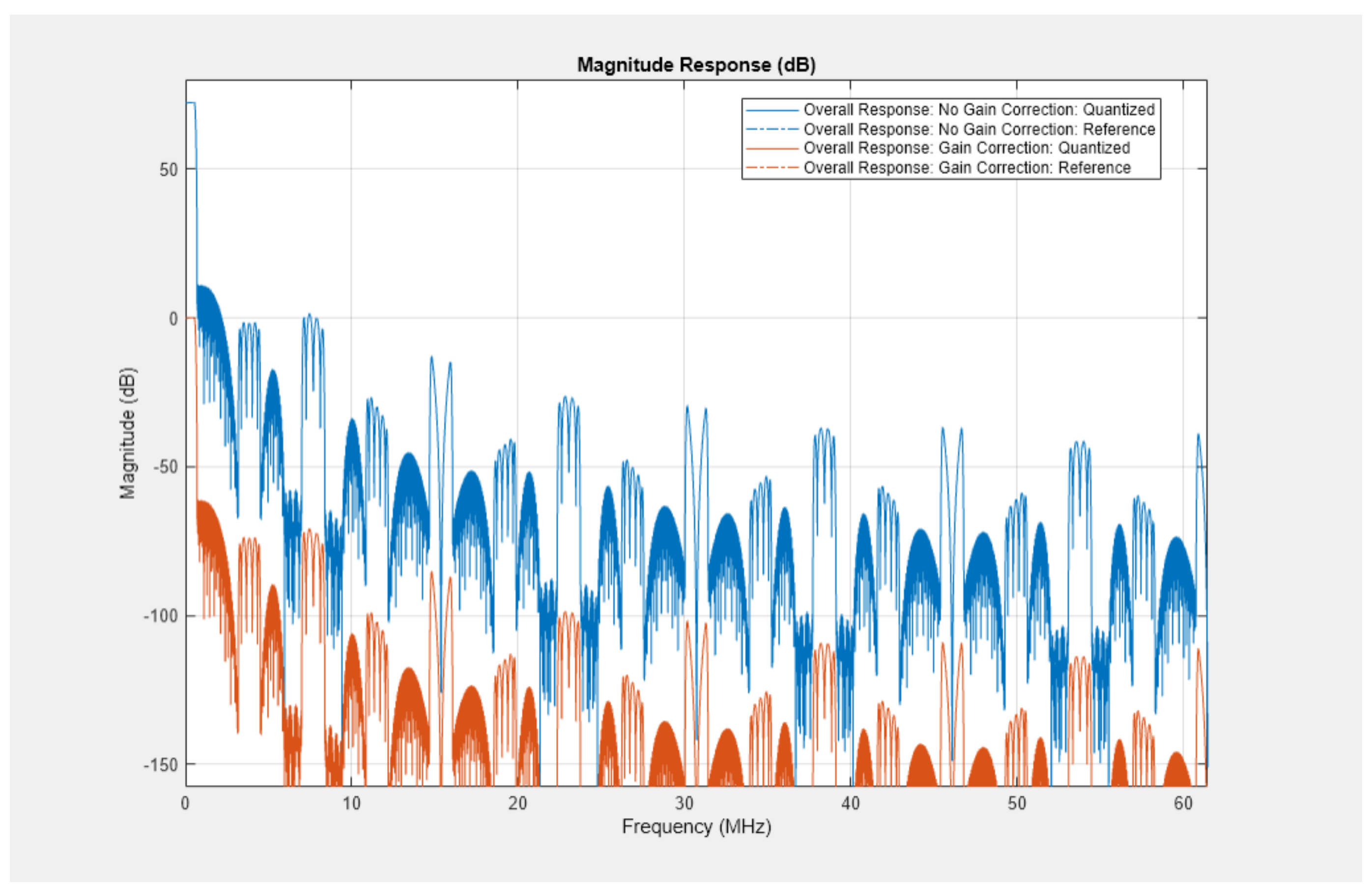
Because the CIC gain is a power of two, a hardware implementation can easily correct for the gain factor by using a shift operation. For analysis purposes, the example represents the gain correction by using a one-tap `dsp.FIRFilter` System object. Combine the filter chain and the gain correction filter into a `dsp.FilterCascade` System object.

```
cicGain = gain(cicFilt)
Gain = lowpassParams.InterpolationFactor* ...
    hbParams.InterpolationFactor*compParams.InterpolationFactor* ...
    cicParams.InterpolationFactor*cicGain;
GainCorr = dsp.FIRFilter('Numerator',1/Gain)
```

```
cicGain =

    64


GainCorr =

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: 2.4414e-04
    InitialConditions: 0

  Use get to show all properties
```

Plot the overall chain response with and without gain correction.

```
ducPlots.overallResponse = fvtool(ducFilterChain,dsp.FilterCascade(ducFilterChain,GainCorr), ...
                           'Fs',[FsIn*64,FsIn*64]);
legend(ducPlots.overallResponse, ...
       'Overall Response: No Gain Correction',...
       'Overall Response: Gain Correction');
```



Fixed-Point Conversion

The frequency response of the floating-point DUC filter chain now meets the specification. Next, quantize each filter stage to use fixed-point types and analyze them to confirm that the filter chain still meets the specification.

Filter Quantization

This example uses 16-bit coefficients, which are sufficient to meet the specification. Using fewer than 18 bits for the coefficients minimizes the number of DSP blocks that are required for an FPGA implementation. The input to the DUC filter chain is 16-bit data with 15 fractional bits. The filter outputs are 18-bit values, which provide extra headroom and precision in the intermediate signals.

```
% First Lowpass Interpolator
lowpassFilt.FullPrecisionOverride = false;
lowpassFilt.CoefficientsDataType = 'Custom';
lowpassFilt.CustomCoefficientsDataType = numerictype([],16,15);
lowpassFilt.ProductDataType = 'Full precision';
lowpassFilt.AccumulatorDataType = 'Full precision';
lowpassFilt.OutputDataType = 'Custom';
lowpassFilt.CustomOutputDataType = numerictype([],18,14);

% Halfband
hbFilt.FullPrecisionOverride = false;
hbFilt.CoefficientsDataType = 'Custom';
hbFilt.CustomCoefficientsDataType = numerictype([],16,14);
hbFilt.ProductDataType = 'Full precision';
hbFilt.AccumulatorDataType = 'Full precision';
hbFilt.OutputDataType = 'Custom';
hbFilt.CustomOutputDataType = numerictype([],18,14);

% CIC Compensation Interpolator
compFilt.FullPrecisionOverride = false;
compFilt.CoefficientsDataType = 'Custom';
compFilt.CustomCoefficientsDataType = numerictype([],16,14);
compFilt.ProductDataType = 'Full precision';
compFilt.AccumulatorDataType = 'Full precision';
compFilt.OutputDataType = 'Custom';
compFilt.CustomOutputDataType = numerictype([],18,14);
```

For the CIC interpolator, choosing the `'Minimum section word lengths'` fixed-point data type option automatically optimizes the internal wordlengths based on the output wordlength and other CIC parameters.

```
cicFilt.FixedPointDataType = 'Minimum section word lengths';
cicFilt.OutputWordLength = 18;
```

Configure the fixed-point properties of the gain correction and FIR-based System objects. The object uses the default `RoundingMethod` and `OverflowAction` property values (`'Floor'` and `'Wrap'` respectively).

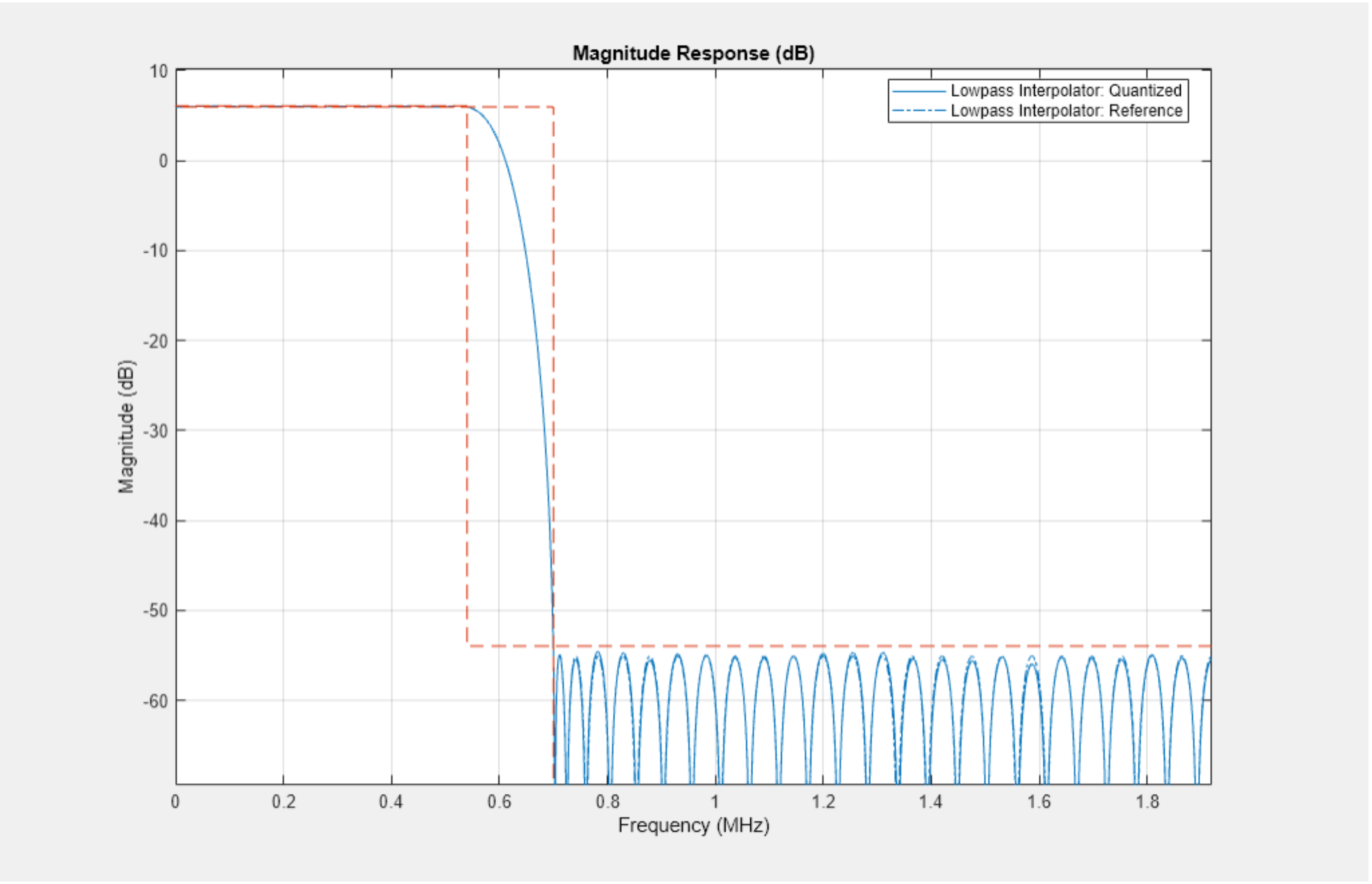
```
% CIC Gain Correction
GainCorr.FullPrecisionOverride = false;
GainCorr.CoefficientsDataType = 'Custom';
GainCorr.CustomCoefficientsDataType = numerictype(fi(GainCorr.Numerator,1,16));
```

```
GainCorr.OutputDataType = 'Custom';
GainCorr.CustomOutputDataType = numerictype(1,18,14);
```

Fixed-Point Analysis

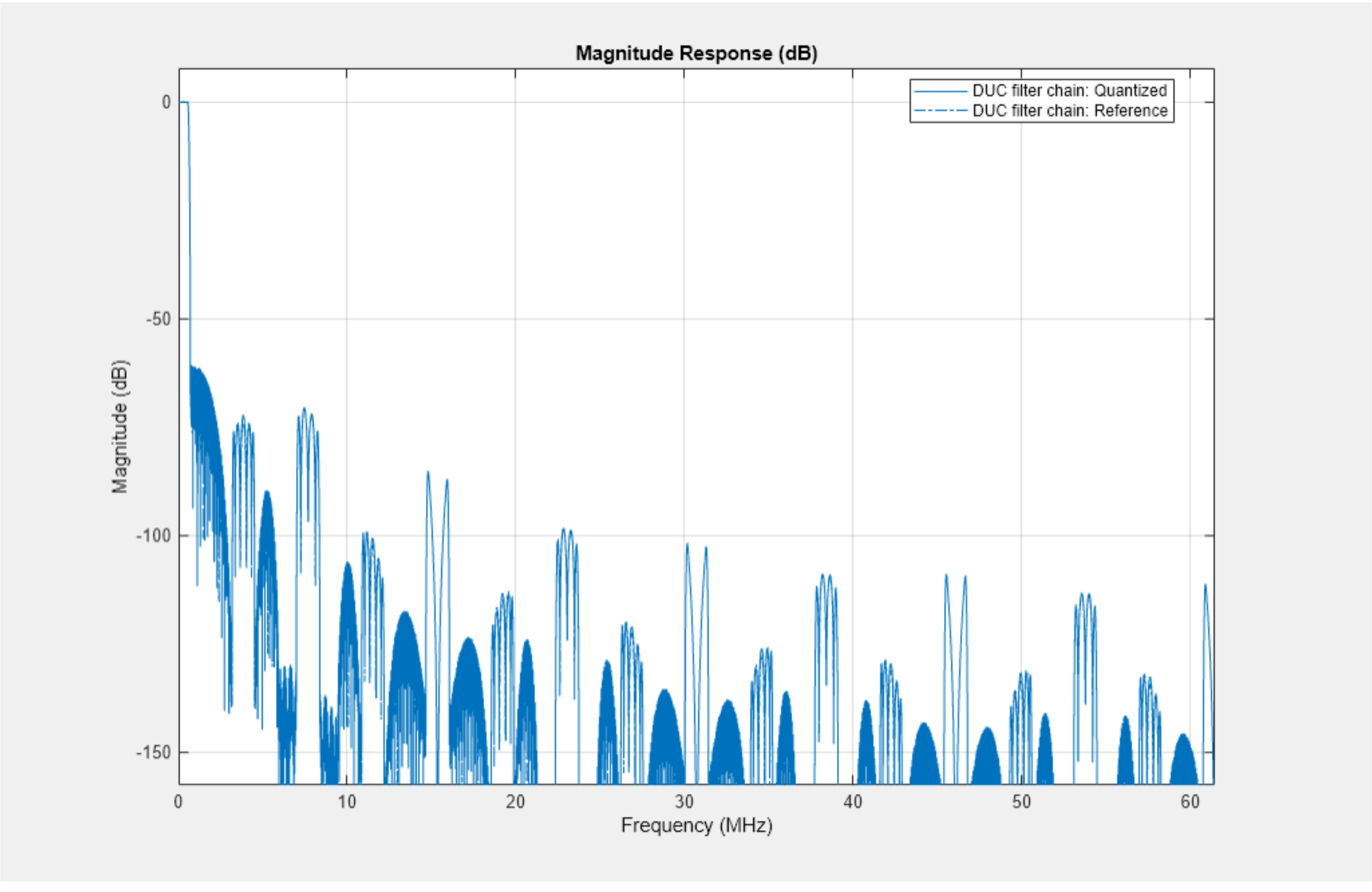
Inspect the quantization effects with `fvtool`. You can analyze the filters individually or in a cascade. `fvtool` shows the quantized and unquantized (reference) responses overlayed. For example, this figure shows the effect of quantizing the first FIR filter stage.

```
ducPlots.quantizedFIR = fvtool(lowpassFilt,'Fs',lowpassParams.FsIn*2,'arithmetic','fixed');

legend(ducPlots.quantizedFIR, ...
       'Lowpass Interpolator');
```



Redefine the `ducFilterChain` cascade object to include the fixed-point properties of the individual filters. Then use `fvtool` to analyze the entire filter chain and confirm that the quantized DUC still meets the specification.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt,compFilt,cicFilt,GainCorr);
ducPlots.quantizedDUCResponse = fvtool(ducFilterChain, ...
    'Fs',FsIn*64,'Arithmetic','fixed');

legend(ducPlots.quantizedDUCResponse, ...
       'DUC filter chain');
```



HDL-Optimized Simulink Model

The next step in the design flow is to implement the DUC in Simulink using blocks that support HDL code generation.

Model Configuration

The model relies on variables in the MATLAB workspace to configure the blocks and settings. The filter blocks are configured by using the filter chain objects defined earlier in the example.

The input to the DUC comes from the `ducIn` variable. For now, assign a dummy value for `ducIn` so that the model can compute its data types. During testing, `ducIn` provides input data to the model.

```
ducIn = 0;
```

The `outputFrame` parameter sets the frame size of the output based on the DAC requirement. `outputFrame` affects the input vector size and valid sample spacing, and it should be a power of two. Using vector input and implementing parallel FPGA operations to achieve higher throughput is referred to as super-sample processing.

```
outputFrame = 4;
```

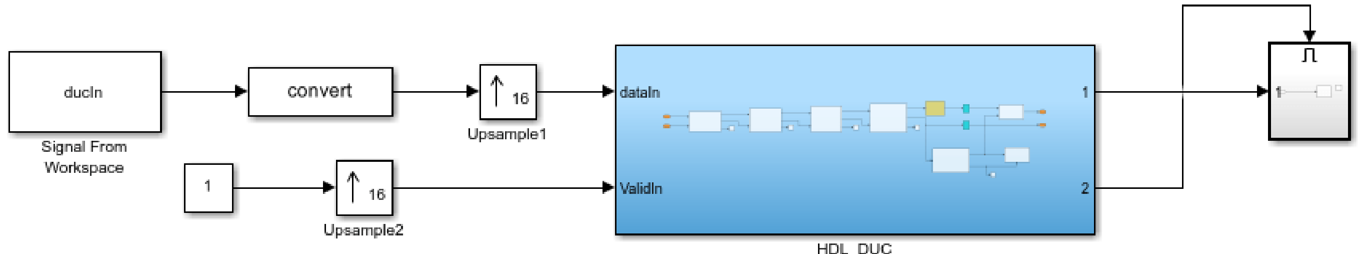
Model Structure

This figure shows the top level of the DUC Simulink model. The model imports the `ducIn` variable from the MATLAB workspace by using a **Signal From Workspace** block, converts the signal to 16-bit

values, and applies the signal to the DUC. The design is single rate, and uses a *valid* signal to convey the rate change from block to block. To simulate the input running 64 times slower than the clock, it is upsampled by 64 with zero insertion. You can generate HDL code from the `HDL_DUC` subsystem.

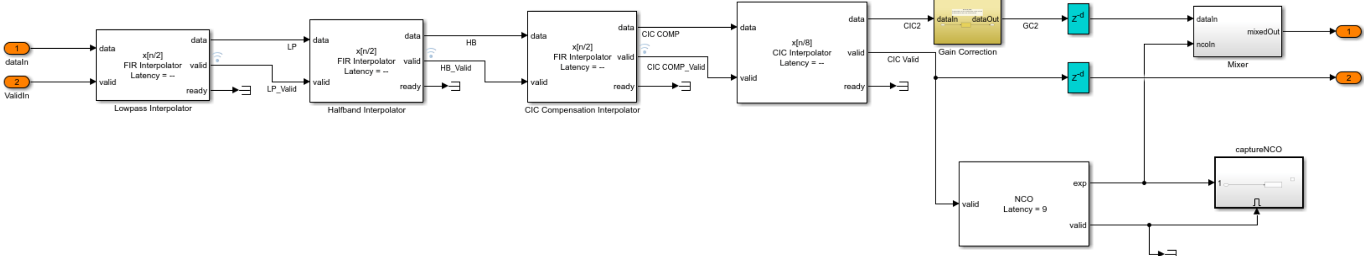
```
modelName = 'DUCforLTEHDL';
open_system(modelName);
set_param(modelName,'Open','on');
```

Implementation of a Digital Up-Converter for LTE in HDL



The DUC implementation is inside the `HDL_DUC` subsystem.

```
set_param([modelName '/HDL_DUC'],'Open','on');
```

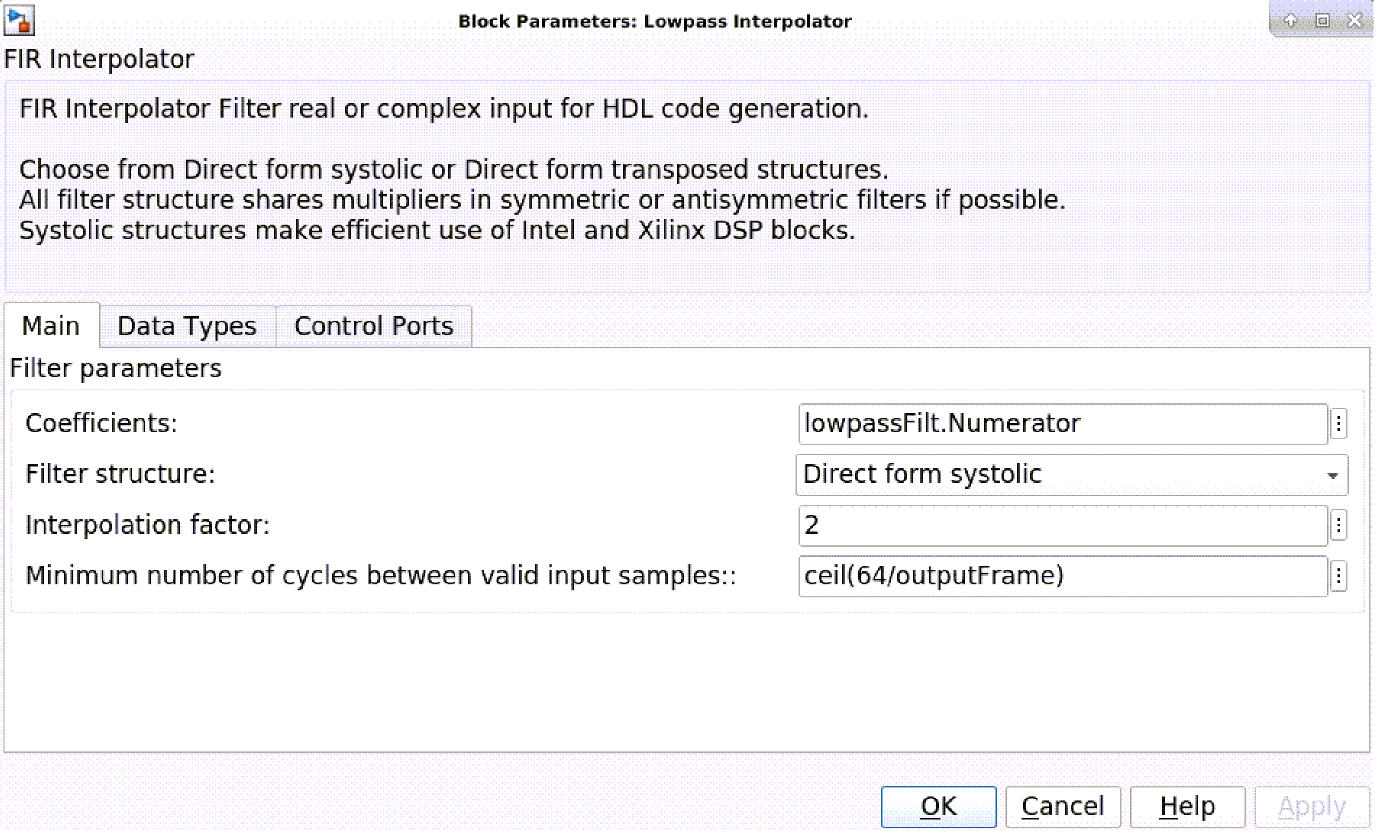


Filter Block Parameters

All of the filters are configured to inherit the coefficients of the corresponding System objects. Each block also has a "Minimum number of cycles between valid input" parameter that is used to optimize the resulting HDL code. The lowpass, halfband, and CIC compensation filters have cycles between valid inputs that can be used for resource sharing - 64, 32, and 16 cycles, respectively.

For example, because the sample rate of the input to the Lowpass Interpolation block is *Fclk*/64, 64 clock cycles are available to process each input sample.

The first filter interpolates by 2. Each polyphase branch is implemented by a separate FIR Filter. Because the number of cycles between valid input samples is greater than 1, each FIR is implemented using the partly serial systolic architecture. The filter has 69 coefficients in total, and after polyphase decomposition each branch has 35 coefficients. There are 64 cycles available for sharing, so each branch is implemented with a fully serial FIR. With complex input, each branch uses 2 multipliers, for a total of 4 multipliers in this filter.



The second filter is a halfband interpolator. This filter can also take advantage of cycles between valid input to perform resource sharing. These cycles are available because each interpolator output has idle cycles between valid samples. The first filter has 64 cycles and interpolates by 2. Therefore, it will output data every 32 cycles. The second filter has 6 coefficients per polyphase branch and so it can be implemented as a fully serial FIR. In this filter, the second branch only contains one non-zero coefficient, and it is a power of 2. The FIR Interpolator block implements this branch as a shift rather than a multiplier. The second filter then has only 2 multipliers.

The third filter is a CIC compensation filter which interpolates by 2. It has 32 coefficients in total, which we specified when designing the filter. The filter is implemented using 2 fully serial complex FIRs, giving a total of 4 multipliers for this filter.

Gain Correction

The gain correction divides the output by 4096, which is equivalent to shifting right by 12 bits. Because the input and output signals of the gain correction each are expressed with 18 bits, the model implements this shift by reinterpreting the data type of the output signal. The Conversion block reinterprets the 12-bit number to have 20 fractional bits rather than 8 fractional bits.

```
set_param([modelName '/HDL_DUC/Gain Correction'],'Open','on');
```

Divide by 4096
To right-shift by 12 bits, reinterpret the number to have
20 fractional bits rather than 8 fractional bits.

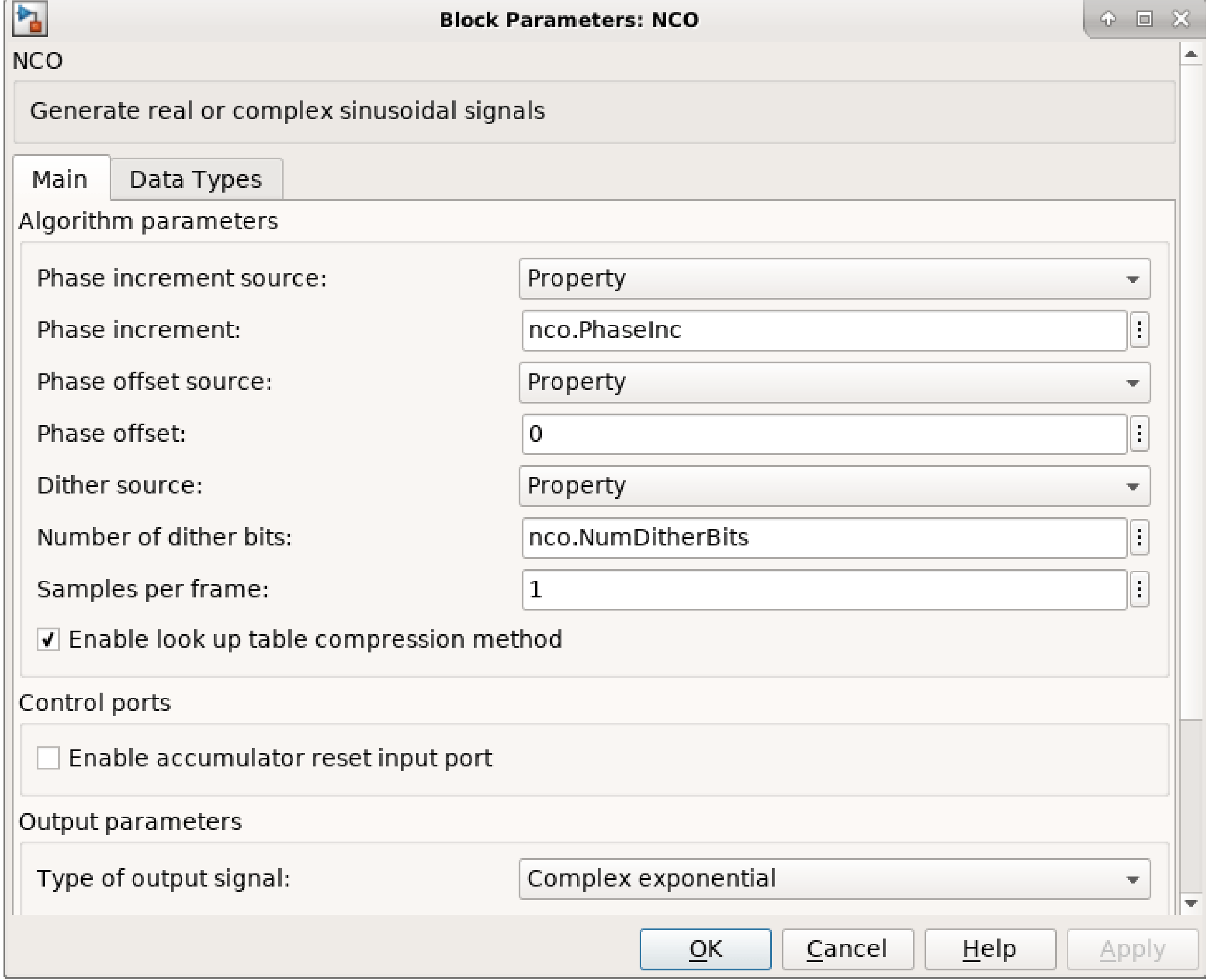
NCO Block Parameters

The NCO block generates a complex phasor at the carrier frequency. This signal goes to a mixer that multiplies the phasor with the output signal. The output of the mixer is sampled at 122.88 Msps.

Specify the desired frequency resolution, then calculate the number of accumulator bits required to achieve the desired resolution, and define the number of quantized accumulator bits. The NCO uses the quantized output of the accumulator to address the sine lookup table. Also compute the phase increment the NCO must use to generate the specified carrier frequency. The NCO applies phase dither to the accumulator bits that are removed during quantization.

```
nco.Fd = 1;
nco.AccWL = nextpow2(FsIn*64/nco.Fd) + 1;
nco.QuantAccWL = 12;
nco.PhaseInc = round((Fc*2^nco.AccWL)/(FsIn*64));
nco.NumDitherBits = nco.AccWL-nco.QuantAccWL;
```

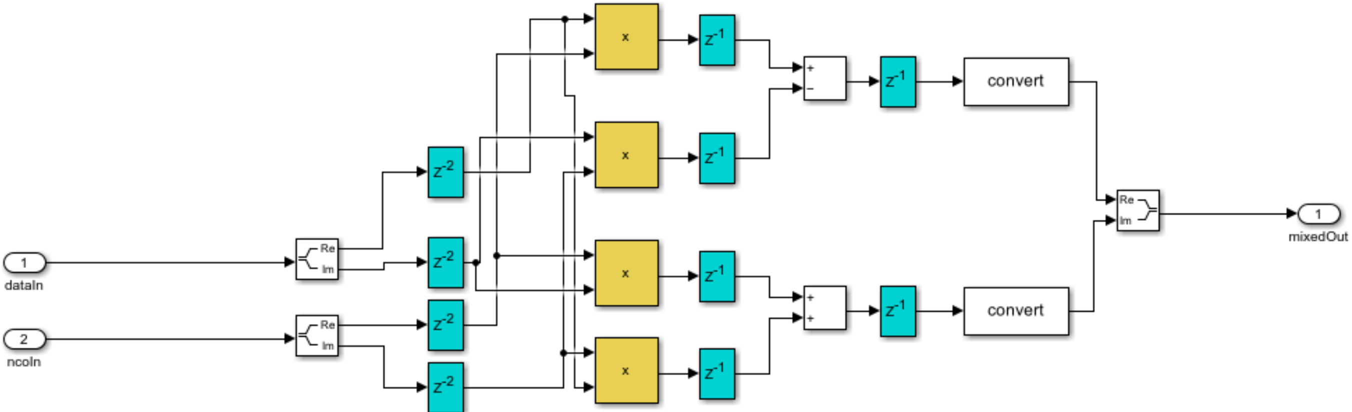
The NCO block in the model is configured with the parameters defined in the `nco` structure. This figure shows the NCO block parameters dialog.



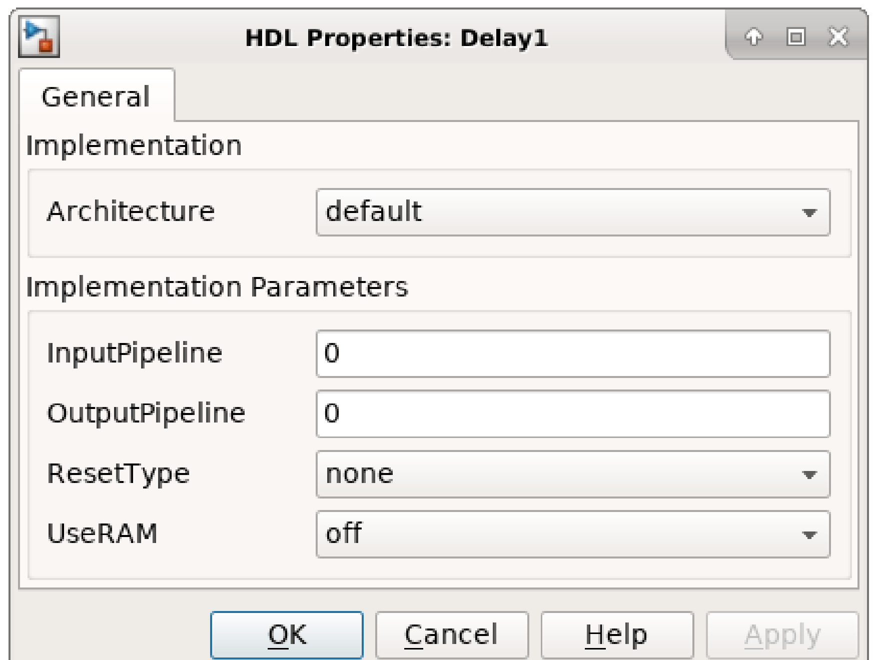
Mixer

The Mixer subsystem performs a complex multiply of the filter output and NCO.

```
set_param([modelName '/HDL_DUC/Mixer'],'Open','on');
```



To map the mixer to DSP slices on FPGA, the mixer is pipelined and the blocks have specific settings. The delay blocks in the mixer are configured to have reset turned off as shown in the image. There are 2 pipeline stages at the input, 1 between the multiplier and adder, and another post-adder. Also, the multipliers and adders are configured to use full-precision output. These blocks use `Floor` rounding method, and the saturate on overflow logic is disabled.



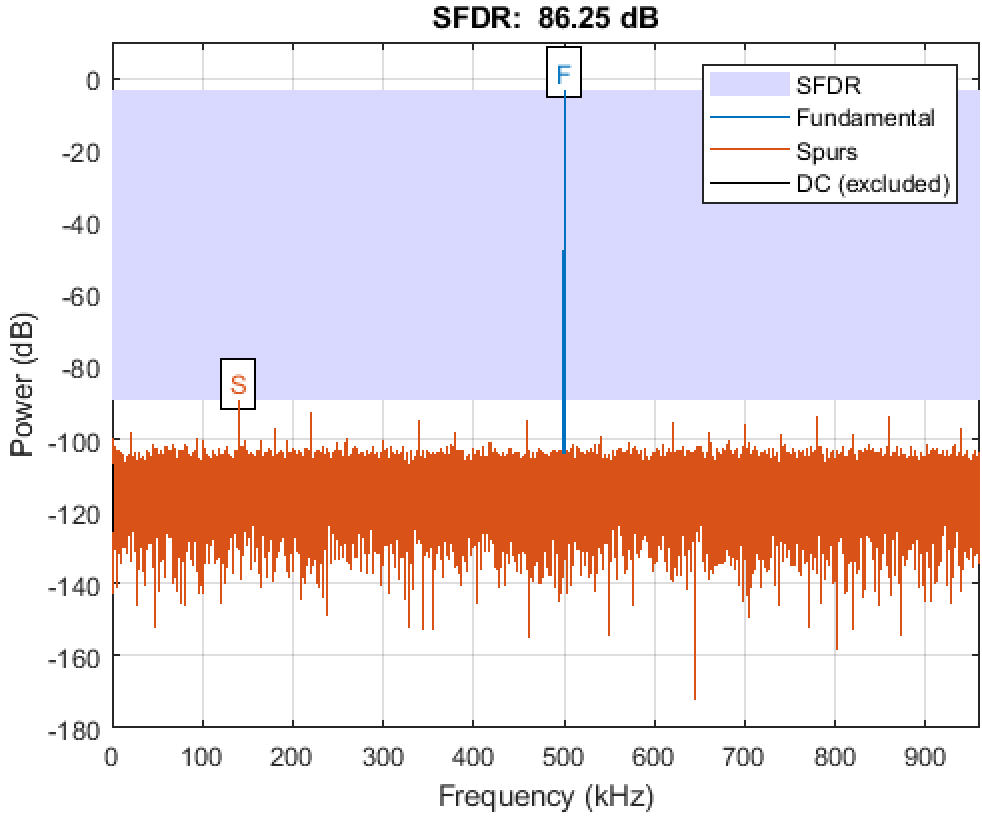
Sinusoid on Carrier Test

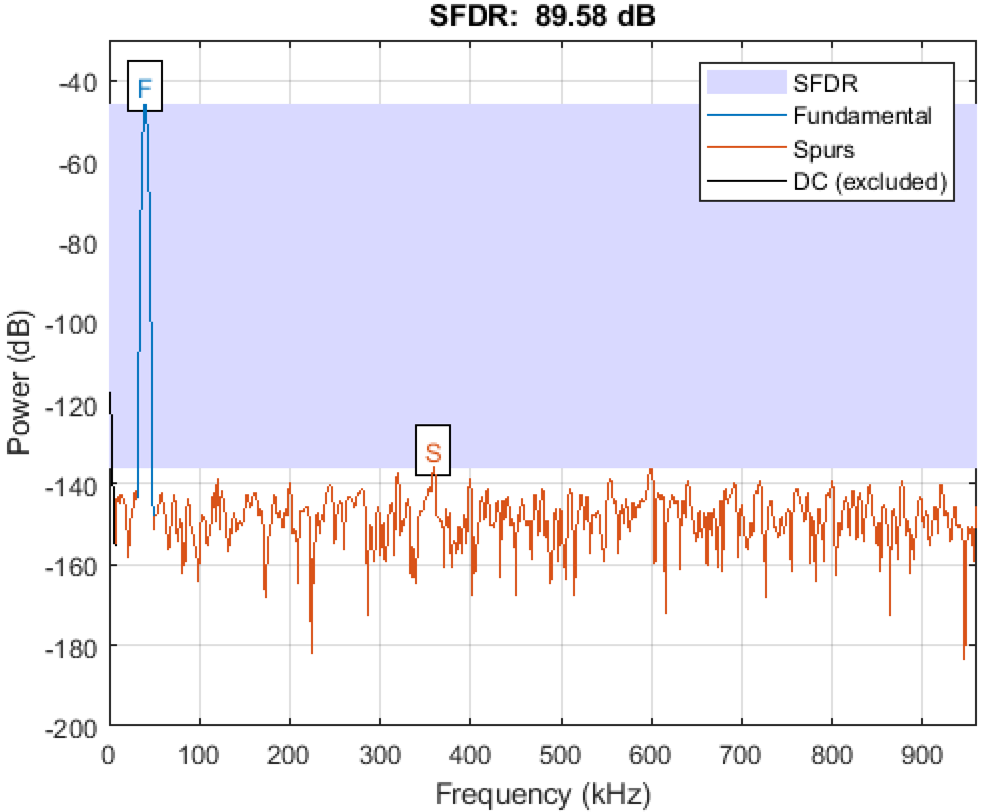
To test the DUC, pass a 40kHz sinusoid through the DUC and modulate the output signal onto the carrier frequency. Demodulate and resample the signal. Then measure the spurious free dynamic range (SFDR) of the resulting tone and the SFDR of the NCO output.

```
% Initialize random seed before executing any simulations.
rng(0);
```

```
% Generate a 40kHz test tone.
ducIn = DUCTestUtils.GenerateTestTone(40e3);

% Upconvert the test signal with the floating-point DUC.
ducTx = DUCTestUtils.UpConvert(ducIn,FsIn*64,Fc,ducFilterChain);
release(ducFilterChain);

% Down convert the output of DUC.
ducRx = DUCTestUtils.DownConvert(ducTx,FsIn*64,Fc);

% Upconvert the test signal by running the fixed-point Simulink model.
simOut = sim(modelName);

% Downconvert the output of DUC.
simTx = simOut.ducOut;
simRx = DUCTestUtils.DownConvert(simTx,FsIn*64,Fc);

% Measure the SFDR of the NCO, floating point DUC, and fixed-point DUC outputs.
results.sfdrNCO = sfdr(real(simOut.ncoOut),FsIn);
results.sfdrFloatDUC = sfdr(real(ducRx),FsIn);
results.sfdrFixedDUC = sfdr(real(simRx),FsIn);

disp('SFDR Measurements');
disp(['   Floating-point DUC SFDR: ',num2str(results.sfdrFloatDUC) ' dB']);
disp(['   Fixed-point NCO SFDR: ',num2str(results.sfdrNCO) ' dB']);
disp(['   Fixed-point DUC SFDR: ',num2str(results.sfdrFixedDUC) ' dB']);
fprintf(newline);

% Plot the SFDR of the NCO and fixed-point DUC outputs.
ducPlots.ncoOutSDFR = figure;
sfdr(real(simOut.ncoOut),FsIn);

ducPlots.ducOutSDFR = figure;
sfdr(real(simRx),FsIn);
```

```
SFDR Measurements
   Floating-point DUC SFDR: 287.9814 dB
   Fixed-point NCO SFDR: 86.2454 dB
   Fixed-point DUC SFDR: 89.5756 dB
```


SFDR: 86.25 dB
SFDR
Fundamental
Spurs
DC (excluded)
F
S
Power (dB)
Frequency (kHz)
0
-20
-40
-60
-80
-100
-120
-140
-160
-180
0
100
200
300
400
500
600
700
800
900



LTE Signal Test

You can use an LTE test signal to perform more rigorous testing of the DUC. Generate a standard-compliant LTE waveform by using LTE Toolbox™ functions. Then, upconvert the waveform with the DUC model. Use LTE Toolbox functions to measure the error vector magnitude (EVM) of the resulting signals.

```
rng(0);
% Execute this test only if you have the LTE Toolbox product.
if license('test','LTE_Toolbox')

    % Generate an LTE test signal by using LTE Toolbox functions.
    [ducIn, sigInfo] = DUCTestUtils.GenerateLTETestSignal();

    % Upconvert the signal with the floating-point DUC and modulate onto carrier.
    ducTx = DUCTestUtils.UpConvert(ducIn,FsIn*64,Fc,ducFilterChain);
    release(ducFilterChain);

    % Add noise to the transmit signal.
    ducTxAddNoise = DUCTestUtils.AddNoise(ducTx);

    % Downconvert the received signal.
    ducRx = DUCTestUtils.DownConvert(ducTxAddNoise,FsIn*64,Fc);

    % Upconvert the signal by using the Simulink model.
    simOut = sim(modelName);
```

```
    % Add noise to the transmit signal.
    simTx = simOut.ducOut;
    simTxAddNoise = DUCTestUtils.AddNoise(simTx);

    % Downconvert the received signal.
    simRx = DUCTestUtils.DownConvert(simTxAddNoise,FsIn*64,Fc);

    results.evmFloat = DUCTestUtils.MeasureEVM(sigInfo,ducRx);
    results.evmFixed = DUCTestUtils.MeasureEVM(sigInfo,simRx);

    disp('LTE EVM Measurements');
    disp(['   Floating-point DUC RMS EVM: '  num2str(results.evmFloat.RMS*100,3) '%']);
    disp(['   Floating-point DUC Peak EVM: ' num2str(results.evmFloat.Peak*100,3) '%']);
    disp(['   Fixed-point DUC RMS EVM: '     num2str(results.evmFixed.RMS*100,3) '%']);
    disp(['   Fixed-point DUC Peak EVM: '    num2str(results.evmFixed.Peak*100,3) '%']);
    fprintf(newline);

end
```

```
LTE EVM Measurements
   Floating-point DUC RMS EVM: 0.813%
   Floating-point DUC Peak EVM: 2.53%
   Fixed-point DUC RMS EVM: 0.816%
   Fixed-point DUC Peak EVM: 2.82%
```

HDL Code Generation and FPGA Implementation

To generate the HDL code for this example you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `HDL_DUC` subsystem. The DUC was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place-and-route resource utilization results for outputFrame of size 4. The design met timing with a clock frequency of 335 MHz.

```
T = table(...
    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}),...
    categorical({'4708'; '654'; '6849'; '2'; '32'}),...
    'VariableNames',{'Resource','Usage'})
```

```
T =

  5x2 table

    Resource    Usage
    ________    _____

     LUT        4708
     LUTRAM     654
     FF         6849
     BRAM       2
     DSP        32
```

HDL QAM Transmitter and Receiver

This example shows how to use Simulink® blocks that support HDL code generation to implement a 64-QAM transmitter and receiver for HDL code generation and hardware implementation.

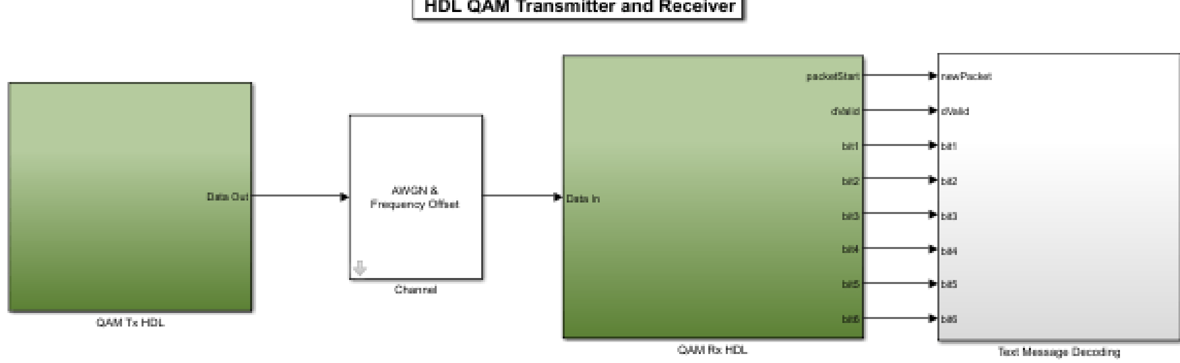
The `HDL QAM Tx` subsystem generates a complex valued, 64-QAM modulated constellation. A floating-point channel model, `Channel`, is used to add attenuation, channel noise, carrier frequency offset, and fractional delay in order to demonstrate the operation of the receiver subsystem. The `HDL QAM Rx` subsystem implements a practical digital receiver to mitigate the channel impairments using coarse frequency recovery, timing recovery, frame synchronization, and magnitude and phase recovery. The `Text Message Decoding` subsystem then receives the data packets, decodes the packets, and prints them to the MATLAB® Command Window.

Structure of the Example

To open the model, enter:

```
modelname = 'commqamtxrxhdl';
open_system(modelname);
set_param(modelname,'Open','on');
```

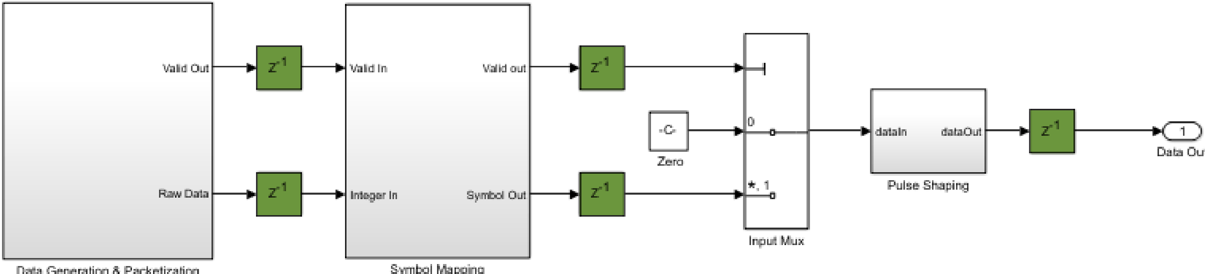
This image shows the top-level structure of the QAM receiver model. The `QAM Tx HDL` and `QAM Rx HDL` subsystems are optimized for HDL code generation.



To open the `QAM Tx HDL` subsystem, enter:

```
set_param(modelname,'Open','off');
set_param([modelname '/QAM Tx HDL'],'Open','on');
```

This image shows the detailed structure of the `QAM Tx HDL` subsystem.

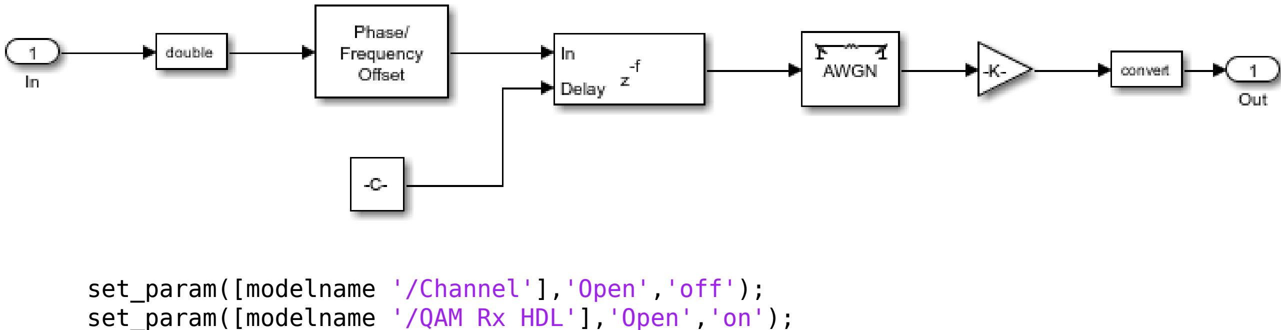


The `QAM Tx HDL` subsystem contains these components, which are described in more detail in the **HDL QAM Transmitter** section.

- **Data Generation & Packetization** - Generates the packets to be transmitted, grouping the bits for mapping to symbols
- **Symbol Mapping** - Maps the bits output from the **Data Generation & Packetization** subsystem to QAM symbols
- **Pulse Shaping** - Performs pulse shaping and upsampling of the symbols using an interpolating RRC (Root Raised Cosine) filter prior to transmission

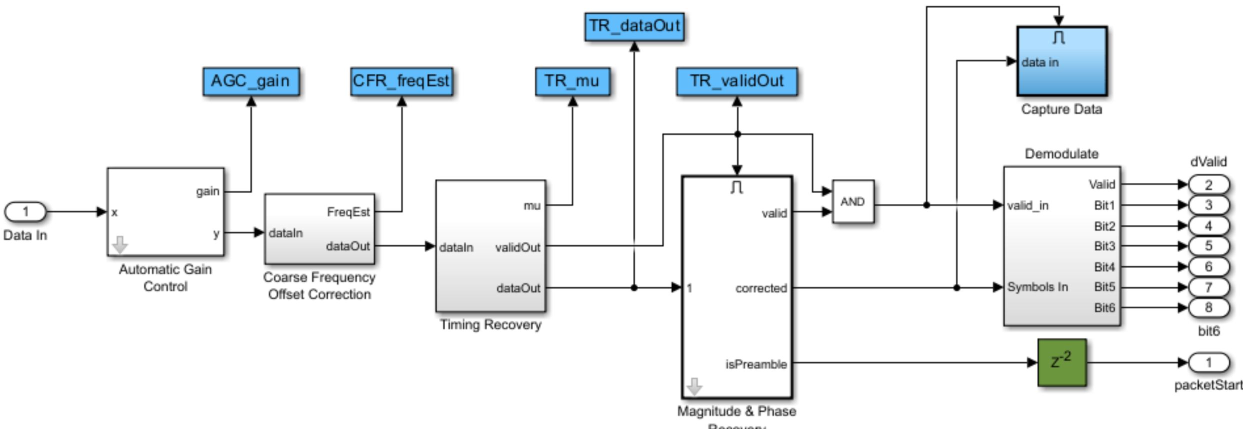
```
set_param([modelname '/QAM Tx HDL'],'Open','off');
set_param([modelname '/Channel'],'Open','on');
```

The structure of the `Channel` subsystem can be seen below. As the `Channel` subsystem is intended to be a rough approximation of a AWGN channel with attenuation and frequency offset it is intended to be run in software. As a result blocks which are not supported for HDL code generation can be used here, such as the **Phase/Frequency Offset** block. The **Phase/Frequency Offset** block does not support fixed point data types, hence the conversion to double at the input of the `Channel` subsystem. The signal is converted back to fixed point before being output from the `Channel` subsystem. A fractional delay and AWGN are applied to the transmitted signal and the **Gain** block attenuates the signal.




```
set_param([modelname '/Channel'],'Open','off');
set_param([modelname '/QAM Rx HDL'],'Open','on');
```

This image shows the detailed structure of the `QAM Rx HDL` subsystem.

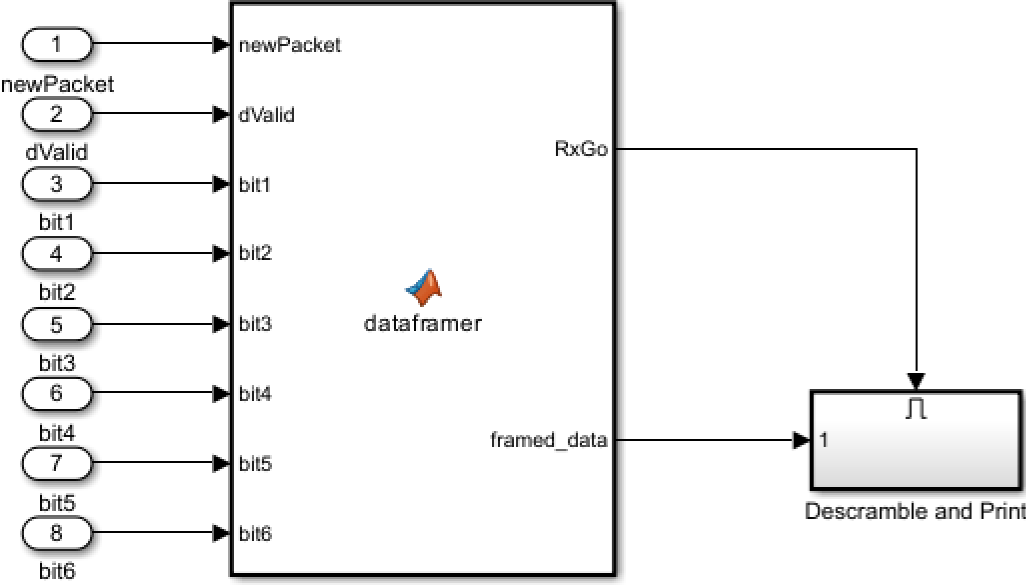


The `QAM Rx HDL` subsystem contains these components:

- **Automatic Gain Control** - Normalizes the received signal power.
- **Coarse Frequency Offset Correction** - Estimates the approximate frequency offset and corrects. The subsystem also contains the **Root Raised Cosine Receive Filter** block which downsamples by two.
- **Timing Recovery** - Resamples the input signal according to a recovered timing strobe so that symbol decisions are made at the optimum sampling instants.
- **Magnitude & Phase Recovery** - Performs packet detection, fine-grained phase and amplitude correction.
- **Demodulate** - Demodulates the signal and demaps symbols to bits.

```
set_param([modelname '/QAM Rx HDL'],'Open','off');
set_param([modelname '/Text Message Decoding'],'Open','on');
```

This image shows the structure of the `Text Message Decoding` subsystem.



This subsystem is designed to be run in software, therefore,the subsystem uses frame-based signals to speed up the computation. The `Text Message Decoding` subsystem has eight sample-based Boolean input signals: `dValid`, `packetStart` and signals `bit1` to `bit6`. The `dataframer` MATLAB Function block converts the sample-based signals to frame-based signals. The demodulated bits are valid only when `dValid` is set high. The `dataframer` block uses the `dValid` signal to fill up a delay line with the received bits and the `newPacket` signal to forward the data stored in the delay line to the output and reset the delay line.

The `Descramble and Print` subsystem processes the received data only when its enable signal goes high. This occurs when either the delay line accumulates 336 valid demodulated bits or the `newPacket` signal is high. This will cause the `dataframer` block to set the `RxGo` signal high. While the simulation is running, the `Descramble and Print` subsystem outputs the string "Hello world! ~64QAM test string~ `###`" to the MATLAB Command Window, where '`###`' is a repeating sequence of '000', '001, '002', ..., '099'. Every 50 packets the subsystem outputs the bit error rate of the data in the last 50 successfully received packets to the MATLAB Command Window.

HDL QAM Transmitter

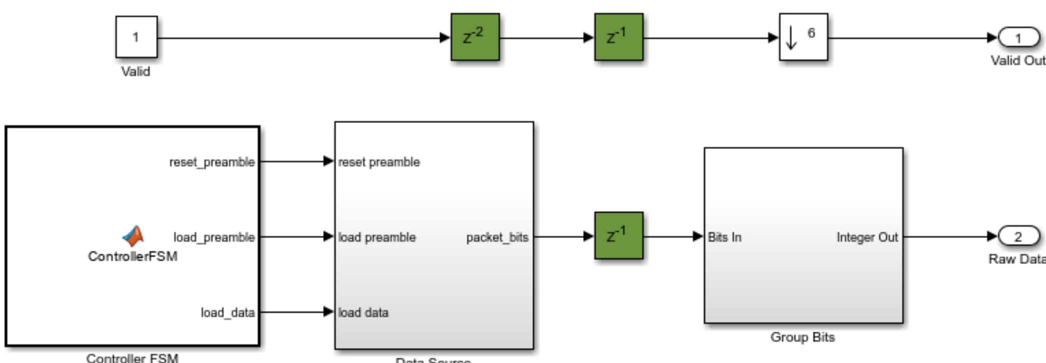
The `HDL QAM Tx` contains the `Data Generation & Packetization`, `Symbol Mapping`, and `Pulse Shaping` blocks.

Data Generation & Packetization

The `Controller FSM` and `Data Source` subsystems generate the preamble bits, data bits, performs scrambling and builds the packets. Each packet consists of an 84-bit Barker code preamble and 252 bits of scrambled data. The `Group Bits` block converts the input data bit stream into a six-bit integer at 1/6th of the input sampling rate, as required by the symbol mapper.

The `Data Source` subsystem has a pipeline delay of two samples. In addition, there is a pipeline delay between the data source and the bit pairing subsystem. The valid signal is therefore delayed to match the pipeline delay of the data path. The `Group Bits` subsystem reduces the sample rate by a factor of six. Placing a downsample by six in the valid control path ensures that the sample rate matches that of the signal path.

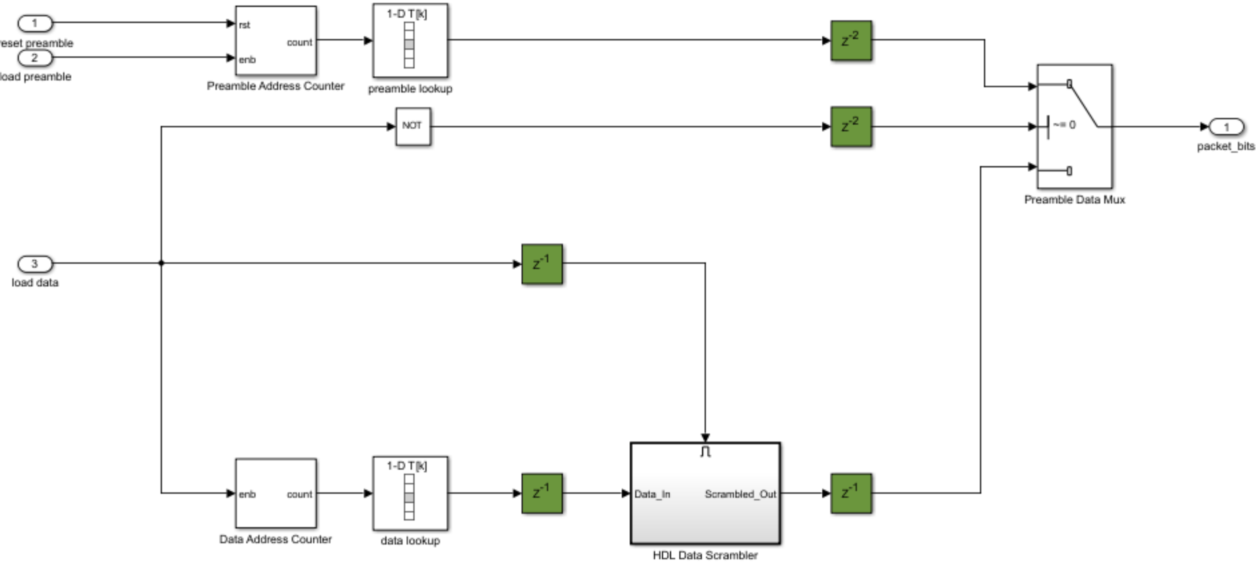
```
set_param([modelname '/Text Message Decoding'],'Open','off');
set_param([modelname '/QAM Tx HDL/Data Generation & Packetization'],'Open','on');
```



`Controller FSM` - The `Controller FSM` block is a MATLAB Function block that implements a control state machine. The FSM has two states, `Pack_Preamble` and `Append_Data`. The `Pack_Preamble` state asserts the `load_preamble` signal and de-asserts the `reset_preamble` and the `load_data` signals. The FSM remains in this state for 84 clock cycles. Then the FSM moves into the `Append_Data` state, asserts the `load_data` signal and the `reset_preamble` signal while releasing the `load_preamble` signal. The FSM remains in this state for 252 clock cycles. The `load_preamble` and `reset_preamble` are Boolean and are used to control the `Preamble Address Counter` which manages the load of the preamble at the start of each packet. The `load_data` signal is Boolean and enables the `Data Address Counter` which controls the loading of data into the packet.

`Data Source` - The `Data Source` subsystem contains two lookup tables (LUTs), storing the preamble and data bits. The `Preamble Address Counter` subsystem, which is controlled by the `reset_preamble` and `load_preamble` signals generated by the `Controller FSM` block, addresses the `preamble lookup` LUT.The `Data Access Counter`, which is enabled by the `load_data` signal generated by the `Controller FSM` block, addresses the `data lookup` LUT. The `Preamble Address Counter` subsystem has a reset signal, generated by the `Controller FSM` block, as the same preamble is inserted at the start of each packet. The `Data Address Counter` subsystem does not have a reset signal as the data address sequence is much longer and varies for each packet as different data bits are placed within each packet. In addition to enabling the counter for the data LUT, the `load data` input controls when the `HDL Data Scrambler` component enables selection of preamble or data bits via the `Preamble Data Mux` block.

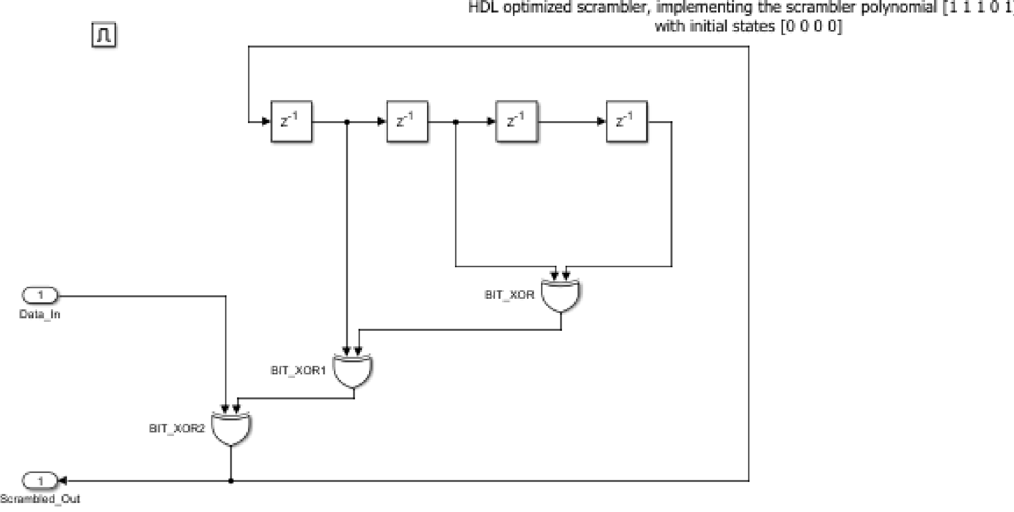
```
set_param([modelname '/QAM Tx HDL/Data Generation & Packetization'],'Open','off');
set_param([modelname '/QAM Tx HDL/Data Generation & Packetization/Data Source'],'Open','on');
```



`HDL Data Scrambler` - To view the `HDL Data Scrambler` block, enter:

```
set_param([modelname '/QAM Tx HDL/Data Generation & Packetization/Data Source'],'Open','off');
set_param([modelname '/QAM Tx HDL/Data Generation & Packetization/Data Source/HDL Data Scrambler
```

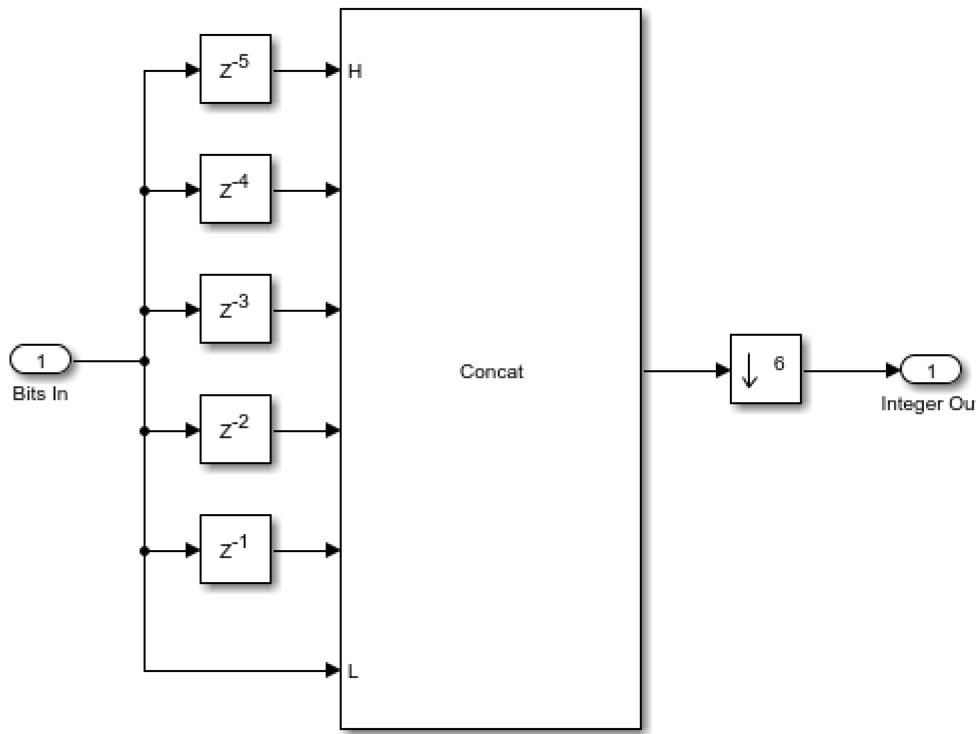
This image shows the `HDL Data Scrambler` subsystem. The subsystem uses XOR gates (for modulo 2 addition) and registers. The enabled subsystem ensures that the scrambler is only enabled when there is new input data to be processed.



`Group Bits` - The `Group Bits` subsystem groups six individual bits into a six-bit unsigned integer output which is the format expected by the symbol mapping component. The delays are used to align

six bits at the input of the **Bit Concat** block, which concatenates into a six-bit unsigned output. This output is then downsampled to select the correct grouping of bits.

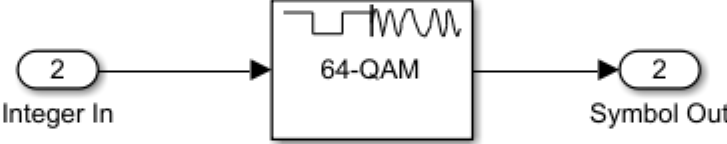
```
set_param([modelname '/QAM Tx HDL/Data Generation & Packetization/Data Source/HDL Data Scrambler
set_param([modelname '/QAM Tx HDL/Data Generation & Packetization/Group Bits'],'Open','on');
```



Symbol Mapping

The `Symbol Mapping` subsystem uses the **Rectangular QAM Modulator Baseband** block to map the integer input value onto the appropriate 64-QAM complex valued symbol. The block uses a Gray mapping scheme.

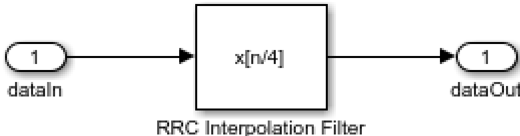
```
set_param([modelname '/QAM Tx HDL/Data Generation & Packetization/Group Bits'],'Open','off');
set_param([modelname '/QAM Tx HDL/Symbol Mapping'],'Open','on');
```



Pulse Shaping

The `Pulse Shaping` subsystem uses the **RRC Interpolation Filter** block with an upsampling factor of four. A matched filter is implemented in the receiver. The filter is pipelined.

```
set_param([modelname '/QAM Tx HDL/Symbol Mapping'],'Open','off');
set_param([modelname '/QAM Tx HDL/Pulse Shaping'],'Open','on');
```



HDL QAM Receiver

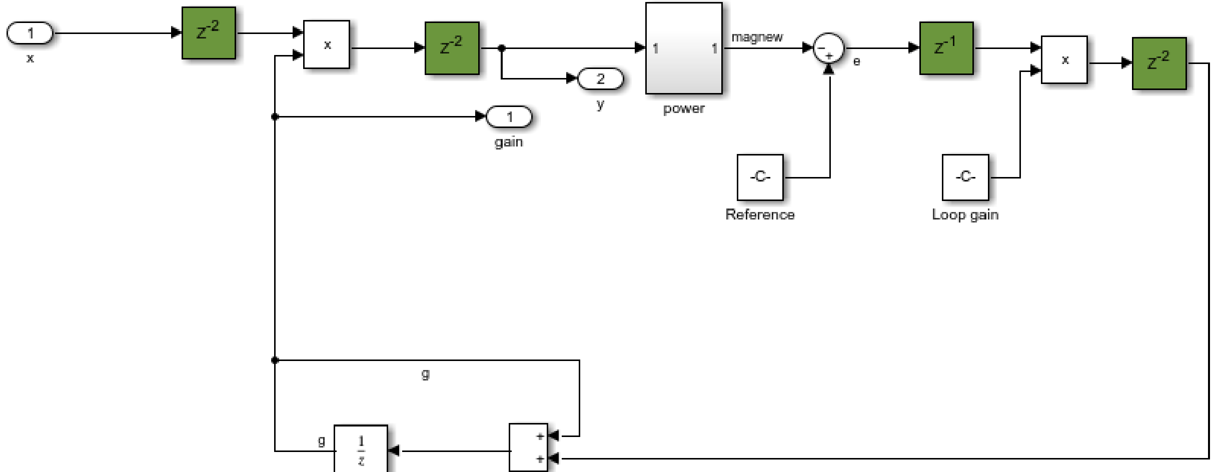
The `HDL QAM Rx` contains the `Automatic Gain Control, Coarse Frequency Offset Correction, Timing Recovery, Magnitude & Phase Recovery, and Demodulate` blocks.

Automatic Gain Control

The `Automatic Gain Control` subsystem ensures that the amplitude of the input of the `Coarse Frequency Compensation` is normalized to the range 1 to -1.

```
set_param([modelname '/QAM Tx HDL/Pulse Shaping'],'Open','off');
set_param([modelname '/QAM Rx HDL/Automatic Gain Control'],'Open','on');
```

This image shows the `Automatic Gain Control` subsystem structure with pipeline registers in green throughout the model.



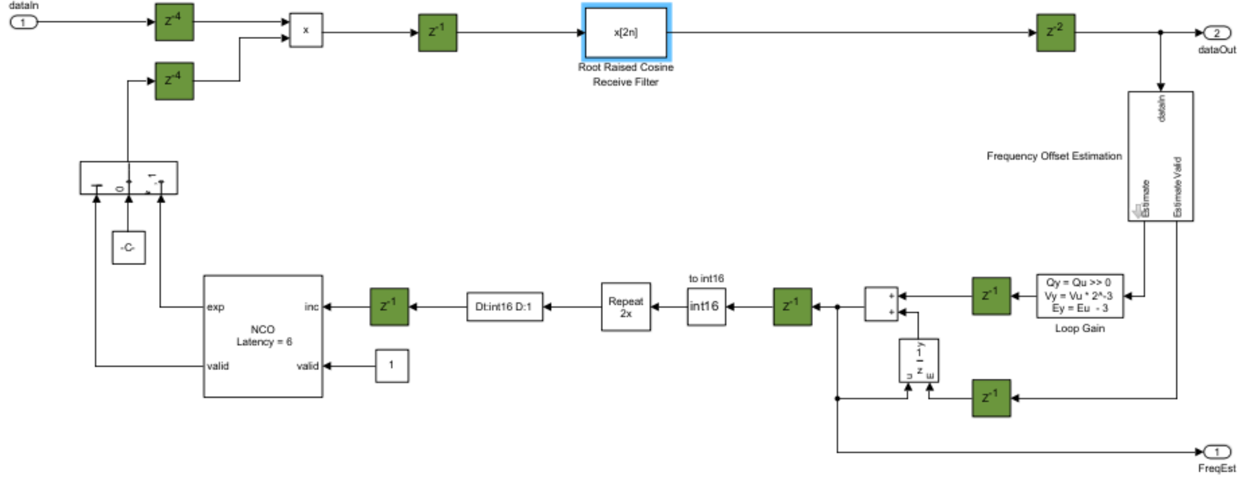
Coarse Frequency Offset Correction

The `Coarse Frequency Offset Correction` subsystem estimates and corrects for the frequency offset by using the Luise-Reggiannini algorithm [1 on page 15-77]. The **Frequency Offset Estimation** subsystem makes an estimate based on the output of the **Root Raised Cosine Receive Filter** block, then frequency offset correction based on this estimate is applied at the input to the **Root Raised Cosine Receive Filter**. This ensures that the desired portion of the received signal bandwidth is better aligned with the receiver filter frequency response, which improves the SNR compared to correcting at the output of the **Root Raised Cosine Receive Filter** block.

Because the estimation and correction algorithm is operating in a closed loop, and makes iterative updates to the previous estimates of the frequency offset, the system gradually converges towards a result. The **Loop Gain** averages the estimates. This architecture is described in [1 on page 15-77]. The **Root Raised Cosine Receive Filter** block implements a downsampling operation so it is necessary to upsample the feedback signal, using the repeat block, to match the rate at the input to the filter.

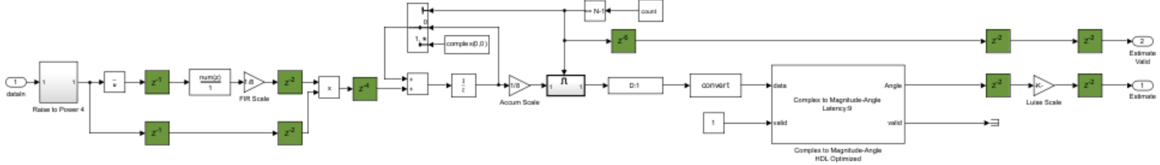
Note that there is a residual frequency offset at the output of the **Coarse Frequency Offset Correction** subsystem that varies over time, even if the frequency offset at the input to the subsystem remains the same as the `Coarse Frequency Offset Correction` subsystem makes new estimates. The **Magnitude and Phase Recovery** subsystem makes fine-grained correction of the residual offset.

```
set_param([modelname '/QAM Rx HDL/Automatic Gain Control'],'Open','off');
set_param([modelname '/QAM Rx HDL/Coarse Frequency Offset Correction'],'Open','on');
```



Frequency Offset Estimation: The **Frequency Offset Estimation** subsystem implements the Luise-Regiannini algorithm, described in [1 on page 15-77]. The subsystem raises the signal to the power four to implement a fourth power phase estimator as described in [2 on page 15-77]. Two cascaded product blocks, with pipelining added to improve hardware performance implement the estimator. The **Discrete FIR Filter** implements the filter with rectangular weights, made up of all ones, described in [1 on page 15-77]. The **FIR Scale** scales the FIR output to account for the filter gain. The **Complex To Magnitude-Angle HDL Optimized** block is used to implement the *angle* function, as required by the Luise-Reggiannini algorithm. This block computes the phase using the hardware friendly CORDIC algorithm. For more information, see the Complex to Magnitude-Angle (DSP HDL Toolbox) block. Before the **Frequency Offset Estimation** subsystem output, the signal is scaled as required by the Luise-Regiannini algorithm and, in addition, is scaled to match the word length of the NCO.

```
set_param([modelname '/QAM Rx HDL/Coarse Frequency Offset Correction'],'Open','off');
set_param([modelname '/QAM Rx HDL/Coarse Frequency Offset Correction/Frequency Offset Estimation
```

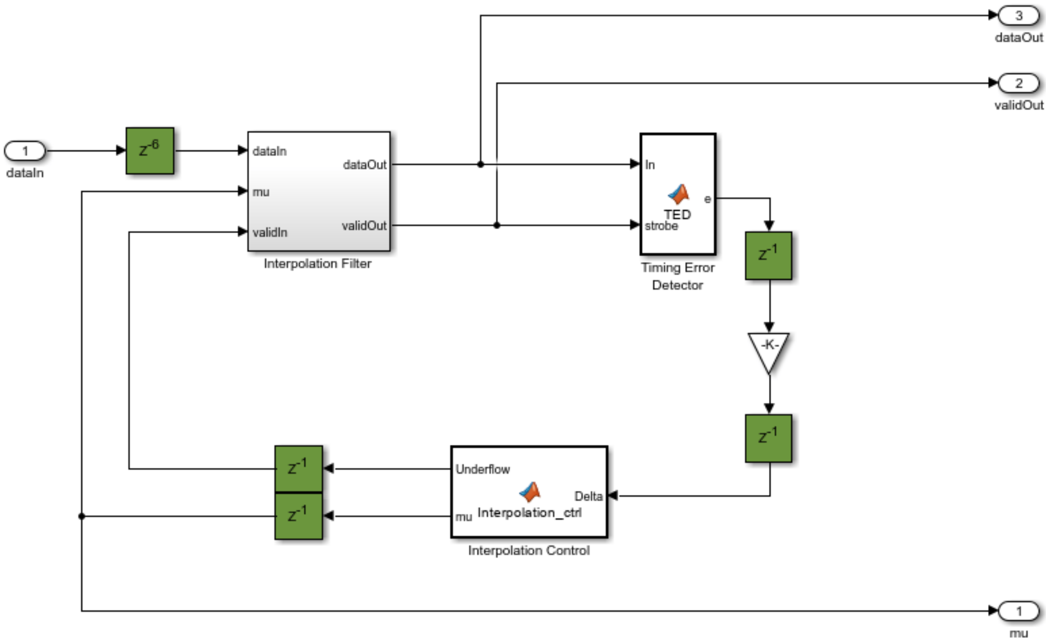


Timing Recovery

To open the `Timing Recovery` subsystem, enter:.

```
set_param([modelname '/QAM Rx HDL/Coarse Frequency Offset Correction/Frequency Offset Estimation
set_param([modelname '/QAM Rx HDL/Timing Recovery'],'Open','on');
```

This image shows the `Timing Recovery` subsystem.



The **Timing Recovery** subsystem implements a PLL, described in Chapter 8 of [3 on page 15-77], to correct the timing error in the received signal. On average, the **Timing Recovery** subsystem generates one output sample for every two input samples.

The **Interpolation Control** block implements a decrementing modulo-1 counter, described in Chapter 8.4.3 of [3 on page 15-77], to generate the control signal to facilitate the selection of the interpolants of the **Interpolation Filter**. This control signal also enables the **Timing Error Detector (TED)**, so that it calculates the timing errors at the correct timing instants. The **Interpolation Control** subsystem updates the timing difference, **mu**, for the **Interpolation Filter**, generating interpolants at optimum sampling instants.

The **Interpolation Filter** is a Farrow parabolic filter with $\alpha = 0.5$ as described in Chapter 8.4.2 of [3 on page 15-77]. The filter uses an α of 0.5 so that all the filter coefficients become 1, -1/2 and 3/2, which significantly simplifies the interpolator structure. Based on the interpolants the **Timing Error Detector,** generates timing errors during a zero crossing as described in Chapter 8.4.1 of [3 on page 15-77].

The **Interpolation Filter** introduces a fractional delay to the signal in order to compensate for the timing error. The fractional delay is controlled by the **mu** input signal. When the timing error (delay) reaches symbol boundaries, there is one extra or missing interpolant in the output. The **Timing Error Detector** implements bit stuffing or skipping to handle the extra or missing interpolants.

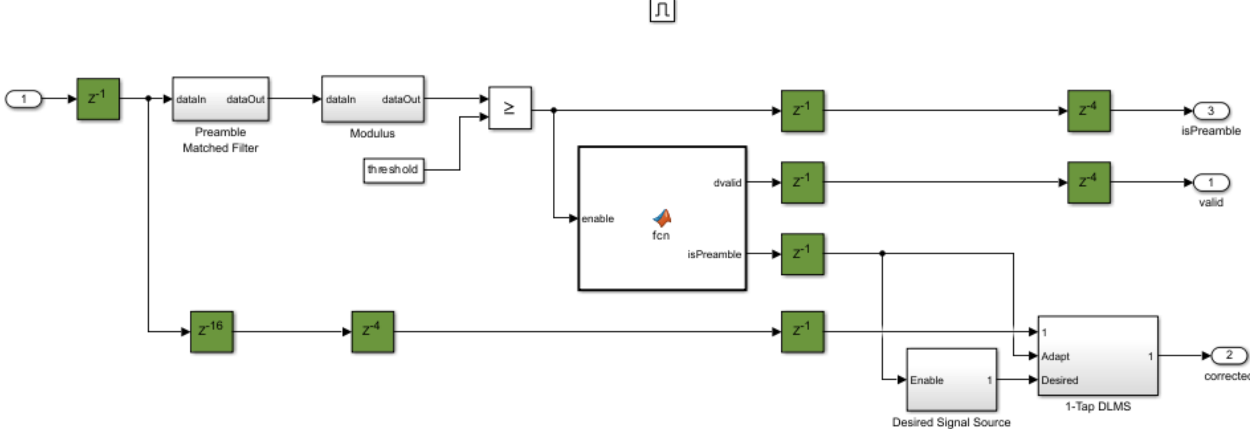
Refer to Chapter 8.4.4 of [3 on page 15-77] for details of bit stuffing and skipping. The timing recovery loop normally generates one output symbol for every two input samples. It also outputs a

timing strobe (**validOut** signal) that runs at the input sample rate. Under normal circumstances, the strobe value is simply a sequence of alternating ones and zeros. However, this occurs only when the relative delay between transmitter and receiver contains some fractional part of one symbol period and the integer part of the delay (in symbols) remains constant. If the integer part of the relative delay changes, the strobe value can have two consecutive zeros or two consecutive ones.

Magnitude & Phase Recovery

The **Magnitude & Phase Recovery** subsystem performs packet synchronization, fine grained frequency recovery and fine grained amplitude recovery.

```
set_param([modelname '/QAM Rx HDL/Timing Recovery'],'Open','off');
set_param([modelname '/QAM Rx HDL/Magnitude & Phase Recovery'],'Open','on');
```

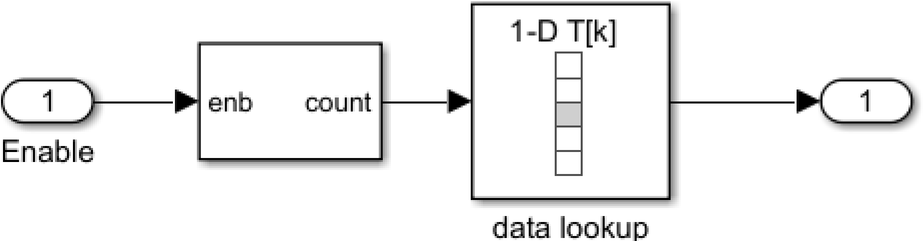


Packet Synchronization: The **Preamble Matched Filter** subsystem uses the time-reversed complex conjugate of the preamble as the filter weights. The modulus of the output of the **Preamble Matched Filter** subsystem is calculated using the **Modulus** subsystem. The output of the **Modulus** subsystem is then compared to a threshold to detect the preamble at the start of a packet. The MATLAB function block generates a signal, **isPreamble**, which is held high for the duration of the preamble of each packet. The MATLAB function block also generates the **dvalid** signal which is set high for the duration of the packet when a preamble has been detected.

Fine Grained Magnitude and Phase Recovery : The **1-Tap DLMS** (Delayed Least Mean Squares) filter subsystem, adapting over the preamble and using the reference signal generated by **Desired Signal Source**, corrects for both phase and magnitude errors. The **isPreamble** signal, generated by the MATLAB function block and set high for the 14 preamble symbols once a packet has been detected, is used to enable the desired signal source and to enable the **Adapt** input of the **1-Tap DLMS**. When the **isPreamble** signal is low, the weight in the **1-Tap DLMS** is held and the **Desired Signal Source** is reset. The Delayed LMS (DLMS) [4 on page 15-77] algorithm is used here to allow for more pipelining to be introduced and, therefore, reduce the critical path in the filter and increase the maximum clock rate achievable after being implemented in hardware.

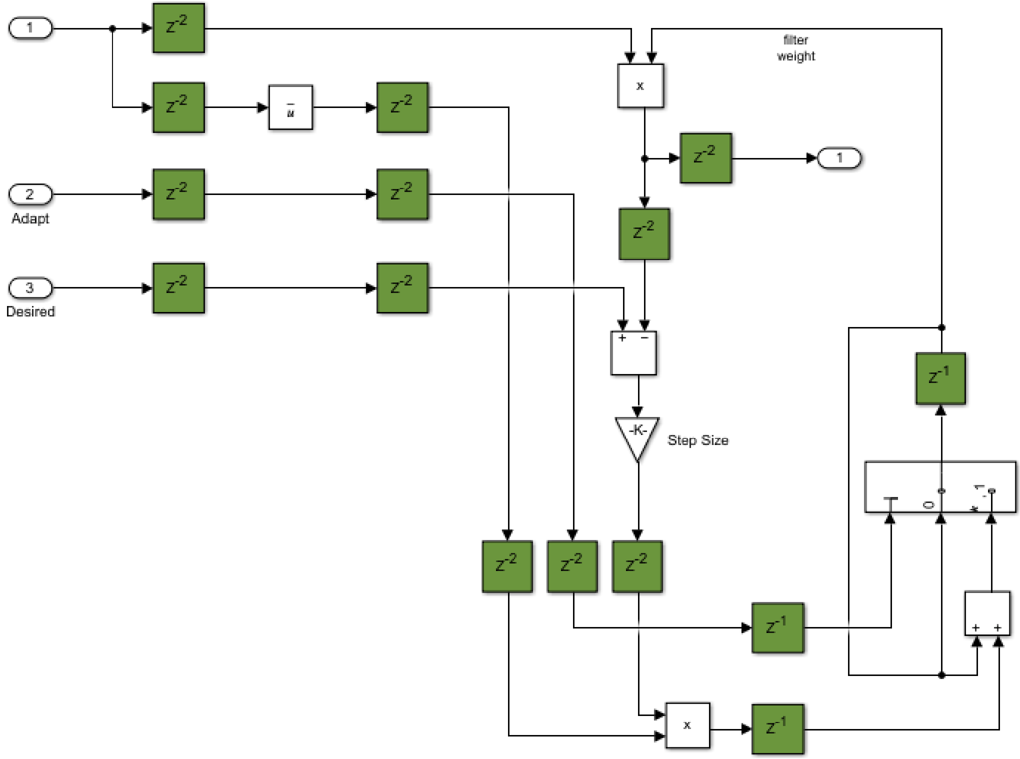
The internal structure of the **Desired Signal Source** subsystem is shown below. The **data lookup** LUT contains the preamble symbols.

```
set_param([modelname '/QAM Rx HDL/Magnitude & Phase Recovery'],'Open','off');
set_param([modelname '/QAM Rx HDL/Magnitude & Phase Recovery/Desired Signal Source'],'Open','on'
```



The internal structure of the **1-Tap DLMS** subsystem is shown below.

```
set_param([modelname '/QAM Rx HDL/Magnitude & Phase Recovery/Desired Signal Source'],'Open','off
set_param([modelname '/QAM Rx HDL/Magnitude & Phase Recovery/1-Tap DLMS'],'Open','on');
```

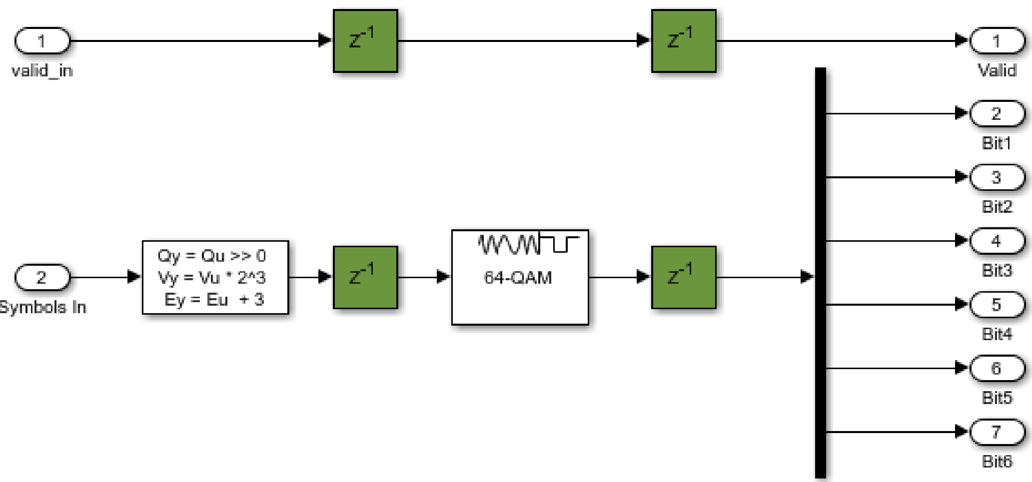


Demodulate

The **Demodulate** subsystem maps each 64-QAM input symbol to bits, outputting 6 bits for each input symbol. To generate HDL for the **Rectangular QAM Demodulator Baseband** block, the minimum distance between symbols must be set to 2. This is 8 times larger than the distance between the symbols generated in the transmitter. As a result, the symbols input to the **Demodulate** subsystem

must be scaled up appropriately. This is done using the **Shift Arithmetic** block which shifts the binary point left by 3 bits to achieve the required multiplication by 8.

```
set_param([modelname '/QAM Rx HDL/Magnitude & Phase Recovery/1-Tap DLMS'],'Open','off');
set_param([modelname '/QAM Rx HDL/Demodulate'],'Open','on');
```



Results and Displays

During the simulation, the model displays successfully received packets in the MATLAB Command Window. At every 50 packets, the MATLAB Command Windows displays the bit error rate of the data in the last 50 successfully received packets.

After running the simulation, the model displays six different figures that illustrate different aspects of the receiver performance. These are shown below, along with an explanation of each plot. The first five plots show the adaption, over the simulation duration, of the **Automatic Gain Control**, **Frequency Offset Estimation**, **Timing Recovery** position estimate, the real part of the constellation at the output of the **Timing Recovery** subsystem, and at the output of the **Magnitude & Phase Recover** subsystem. The last plot shows the constellation diagram at the output of **Magnitude & Phase Recovery** subsystem after any adaption has taken place.

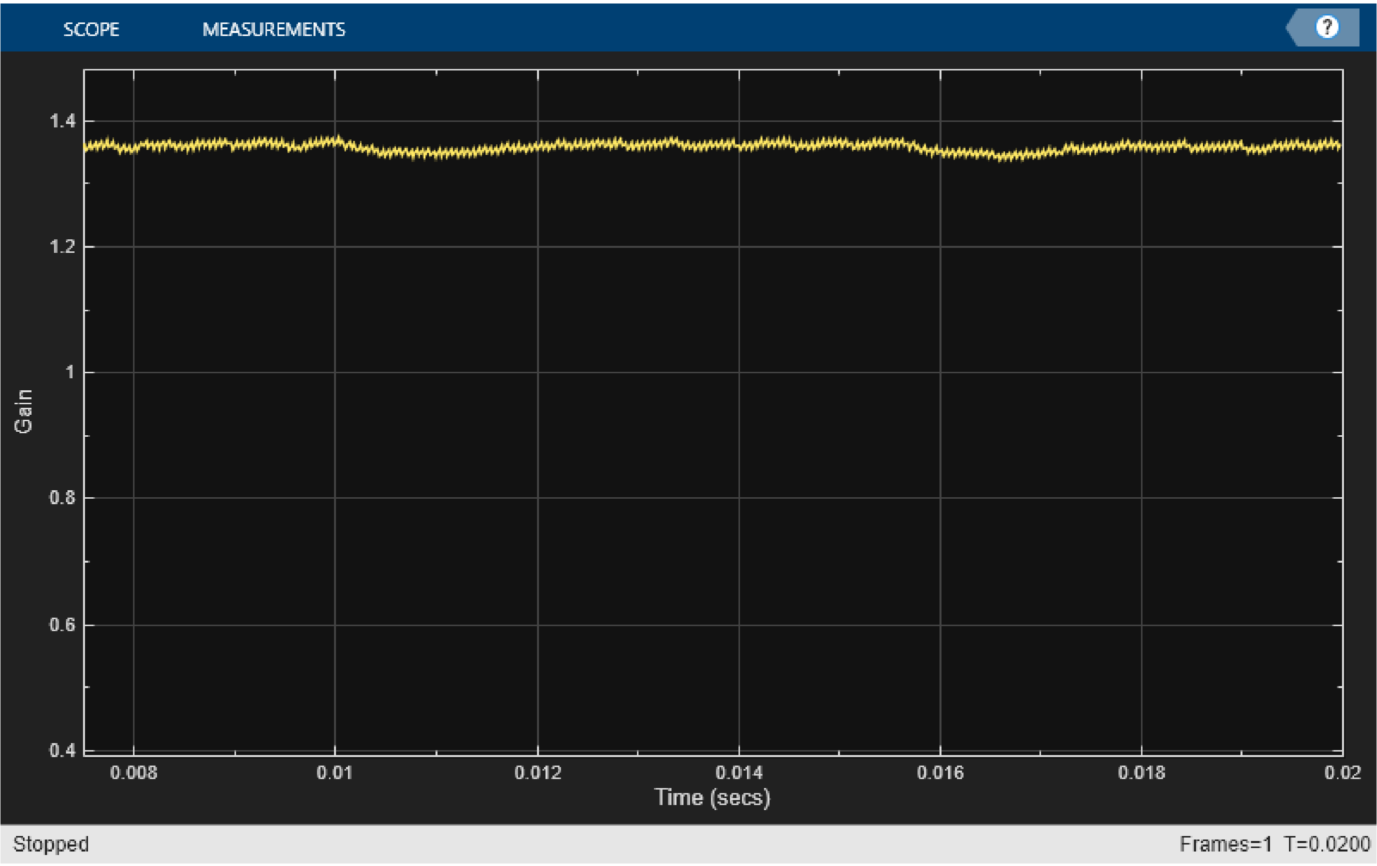
```
set_param([modelname '/QAM Rx HDL/Demodulate'],'Open','off');
print_cap = evalc('sim(modelname)'); %#ok<NASGU>
clear print_cap;
tscope1.hide;
tscope2.hide;
tscope3.hide;
constDiag1.hide;
constDiag2.hide;
constDiag3.hide;
```

Automatic Gain Control Plot

This plot illustrates the **Automatic Gain Control** subsystem adapting over time to normalize the output. A balance must be struck between how quickly the automatic gain control adapts and how

much ripple there is after the gain has reached a relatively constant level. Using a larger automatic gain control loop gain adapts faster, but the amplitude after adaption varies more. Using a smaller loop gain slows the adaption of the automatic gain control, smooths the level after adaption but takes longer to adapt.

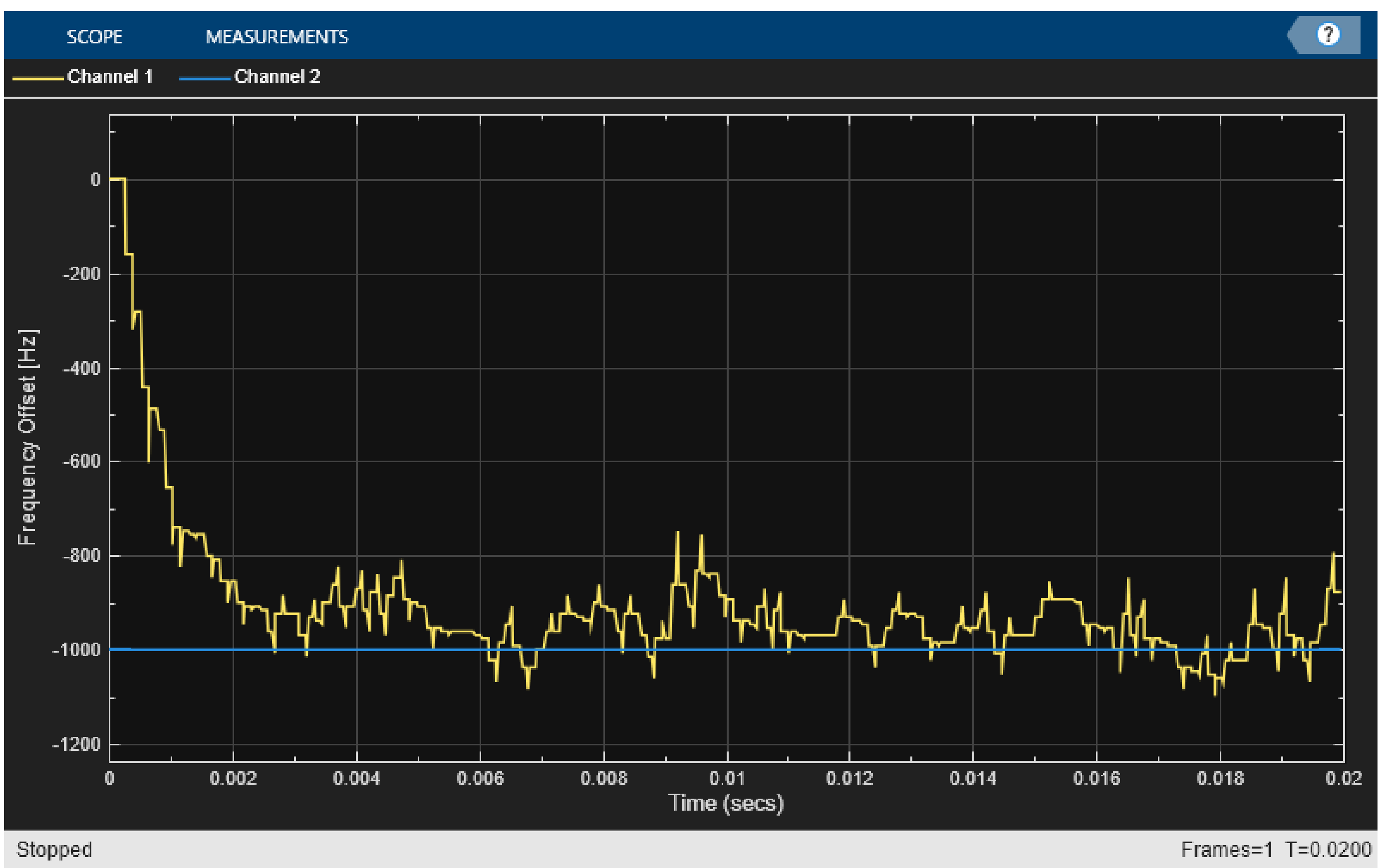
```
tscope1.show;
```



Frequency Offset Estimate Plot

This plot illustrates how the coarse frequency offset gradually adapts towards the frequency offset introduced by the system, as indicated by the blue horizontal line. This image shows that while the estimate comes close to the actual frequency offset, there is still a residual error that must be addressed later in the system.

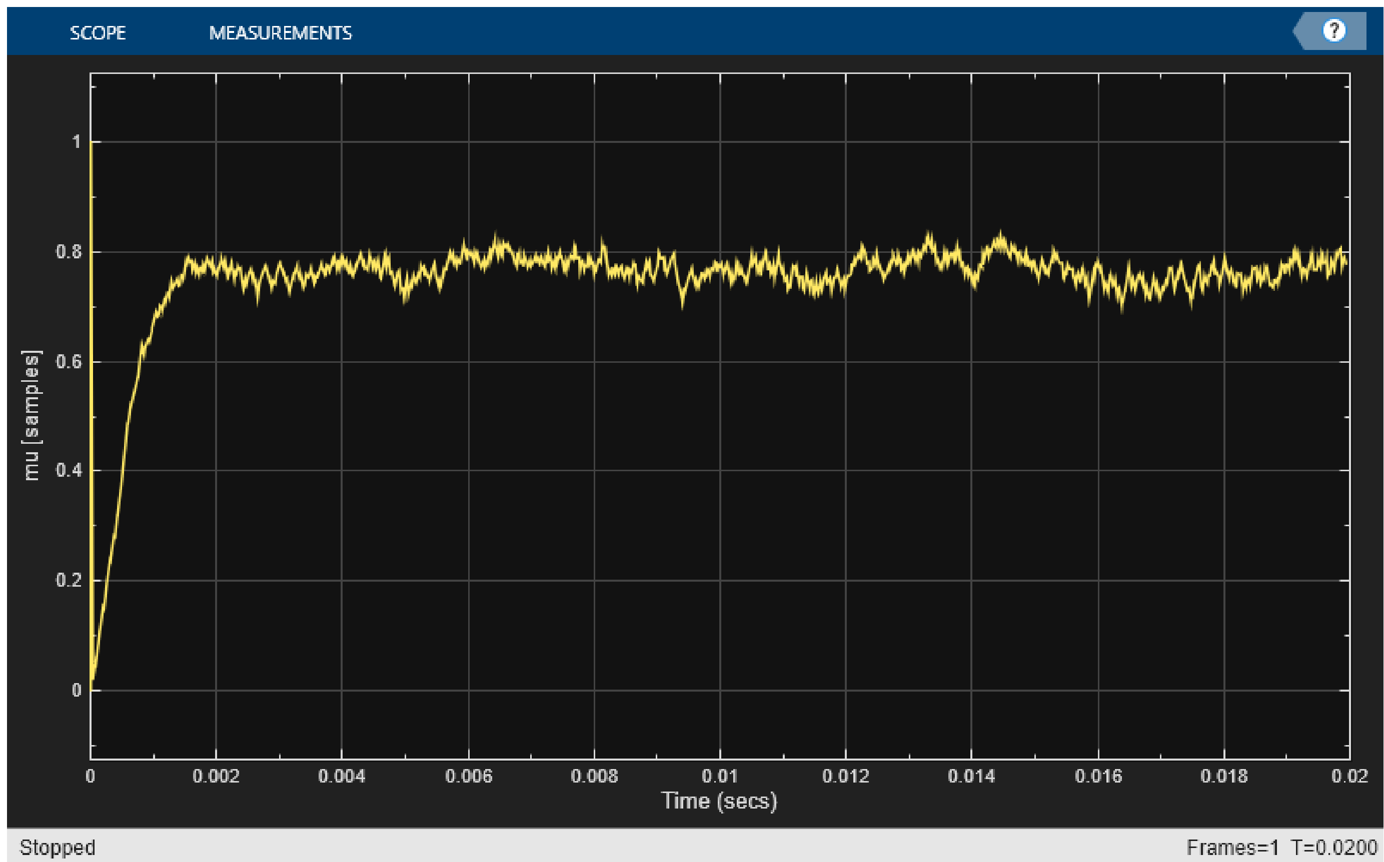
```
tscope1.hide;
tscope2.show;
```



Timing Recovery Position Plot

This plot shows the **mu** input to the **Interpolation Filter**. Note that **mu** converges to a steady state, with some ripple over time as the channel delay does not vary during the simulation.

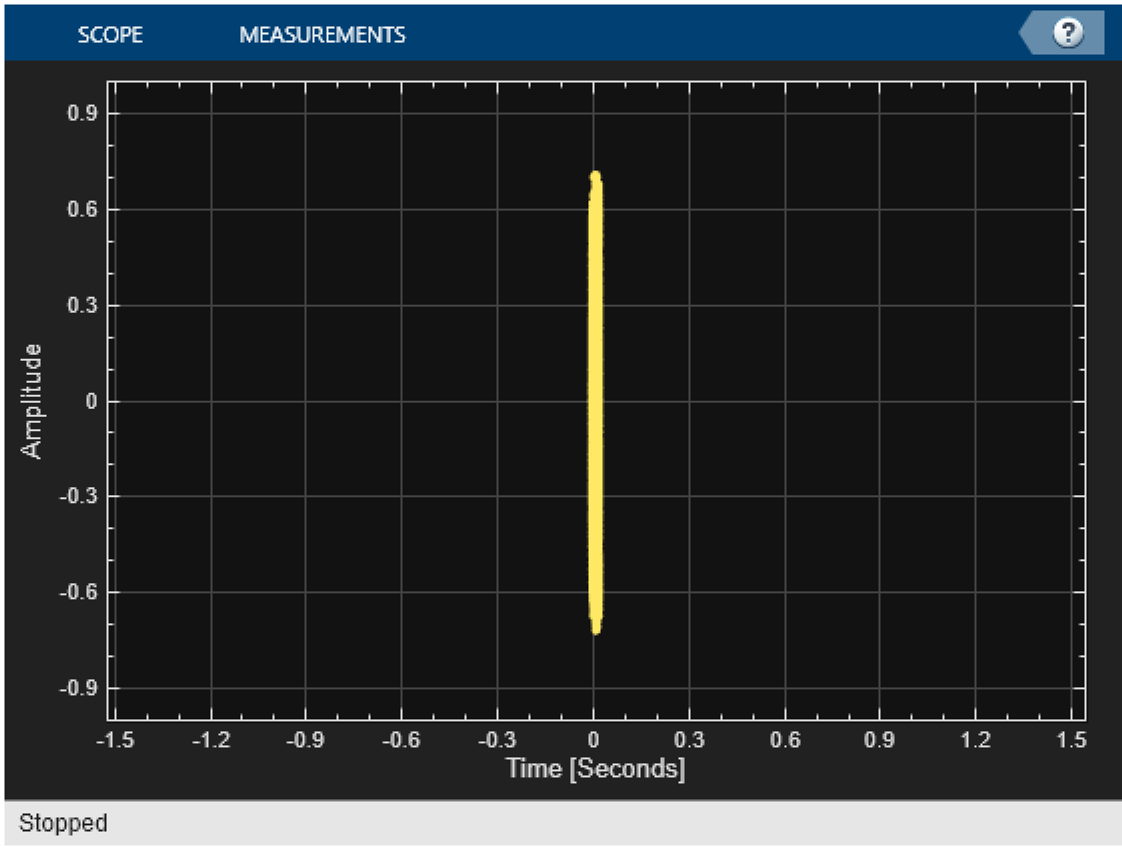
```
tscope2.hide;
tscope3.show;
```



Real Part of Timing Recovery Output Plot

This plot illustrates how the real part of the **Timing Recovery** subsystem output is beginning to converge towards the eight distinct amplitude levels expected for 64QAM. However, as the residual frequency offset remaining after the coarse frequency recovery has not yet been corrected at this point in the receiver, the quality of the signal varies with the distinct amplitude levels more clearly visible at some points than at others. The constellation still has some rotation at this point in the receiver.

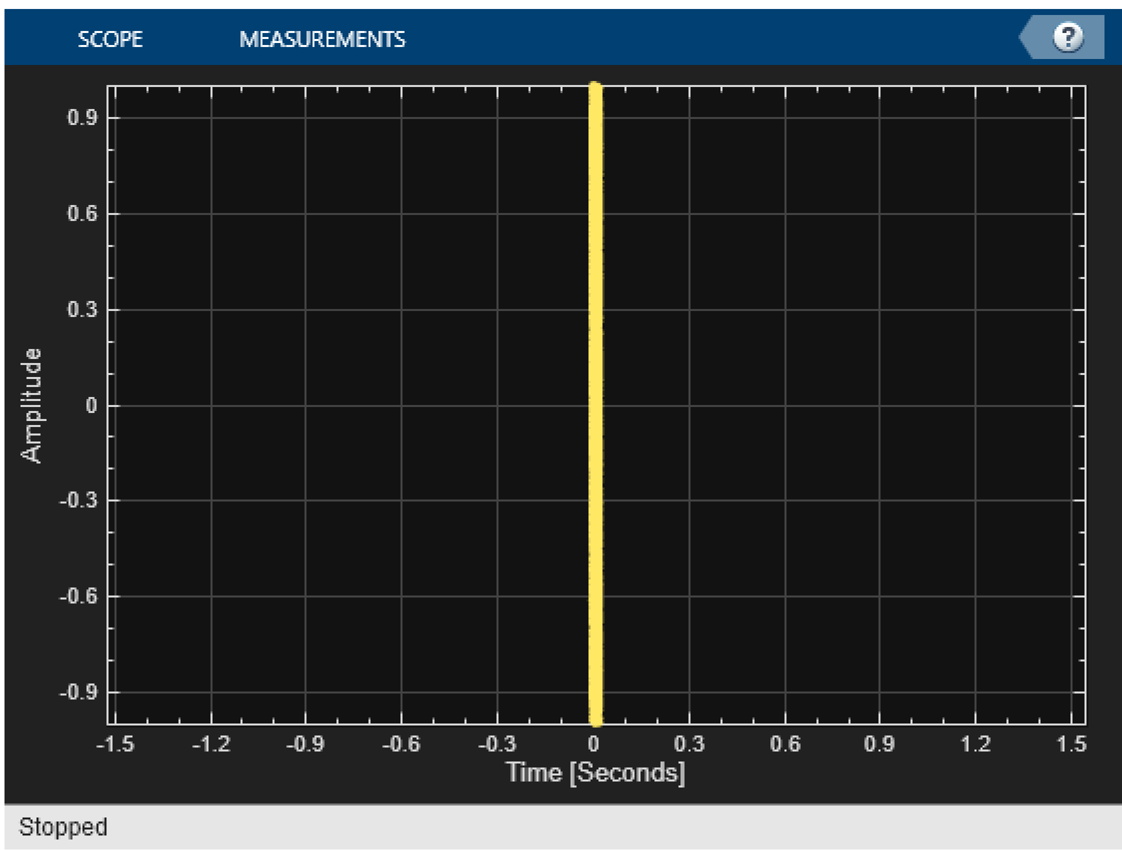
```
tscope3.hide;
constDiag1.show;
```



Real Part of Symbol Estimates Plot

This plot shows how the real part of output of the **Magnitude & Phase Recovery** subsystem adapts over time. Unlike the previous plot, this diagram is generated after the fine frequency recovery, therefore the constellation should not be rotating. There are no samples initially as the output from the block is not valid, and then eight clear amplitude levels should be seen - representing the eight real amplitude levels of the 64-QAM constellation.

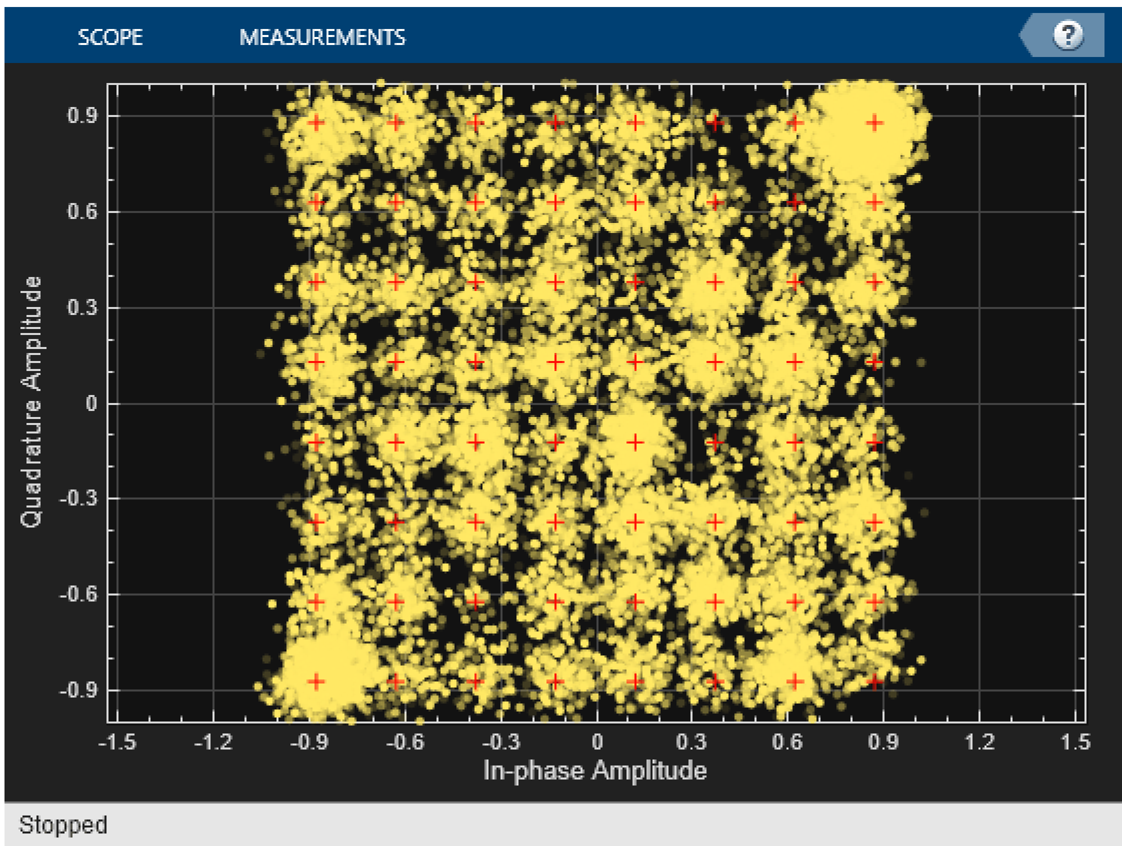
```
constDiag1.hide;
constDiag2.show;
```



Recovered Constellation Plot

This plot shows the constellation at the output of the **Magnitude & Phase Recovery** subsystem after the system has had time to adapt to the channel. Reducing the channel noise should reduce the size of each of the constellation points; increasing the channel noise begins to merge the distinct constellation points together. If the system has not successfully corrected for the frequency offset, then rotation of the constellation is visible here.

```
constDiag2.hide;
constDiag3.show;
```



References

1. Luise, M., and R. Reggiannini. “Carrier Frequency Recovery in All-Digital Modems for Burst-Mode Transmissions.” *IEEE Transactions on Communications* 43, no. 2/3/4 (February 1995): 1169–78. https://doi.org/10.1109/26.380149.

2. Moeneclaey, M., and G. de Jonghe. “ML-Oriented NDA Carrier Synchronization for General Rotationally Symmetric Signal Constellations.” *IEEE Transactions on Communications* 42, no. 8 (August 1994): 2531–33. https://doi.org/10.1109/26.310611.

3. Rice, Michael. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, N.J: Pearson/Prentice Hall, 2009.

4. Long, G., F. Ling, and J.G. Proakis. “The LMS Algorithm with Delayed Coefficient Adaptation.” *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37, no. 9 (September 1989): 1397–1405. https://doi.org/10.1109/29.31293..

```
constDiag3.hide;
close_system(modelname);
clear modelname;
```

Airplane Tracking with ADS-B Captured Data

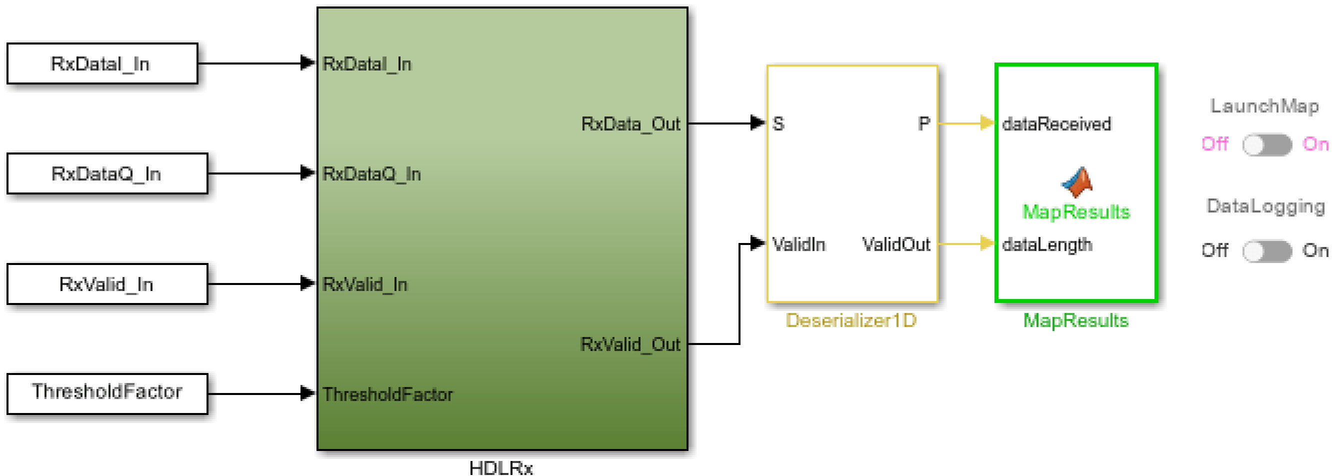
This example shows how to implement the Automatic Dependent Surveillance - Broadcast (ADS-B) receiver for HDL code generation and hardware implementation. This example decodes ADS-B extended squitter messages which can be used to track the airplane. The HDL-optimized model in this example uses Simulink® blocks that support HDL code generation to implement the ADS-B Receiver. This example model is used for real-time processing in “HW/SW Co-Design Implementation of ADS-B Receiver Using Analog Devices AD9361/AD9364” (SoC Blockset), which requires the SoC Blockset™ Support Package for Xilinx® Devices.

Introduction

ADS-B is an air traffic management and control surveillance system. The broadcast messages (approximately once per second) contain the flight information including position and velocity. For introduction on ADS-B technology and modes of transmission, see [1]. The **HDLRx** subsystem is optimized for HDL code generation. The captured received signal is streamed into the receiver (**HDLRx** subsystem) front end. The streaming output of the receiver is buffered and passed to the **MapResults** MATLAB® function to view the output.

Structure of the Example

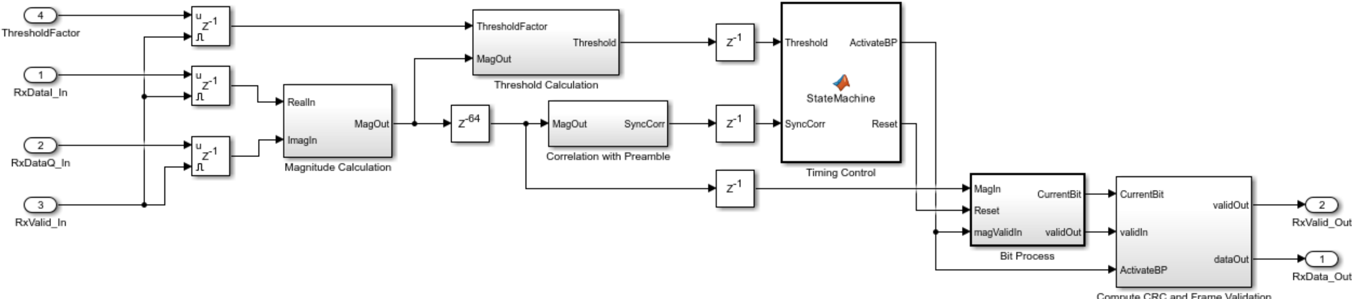
The model supports both Normal and Accelerator modes. The top-level structure of the ADS-B receiver model is shown in the following figure.



The receiver input data is captured using “HW/SW Co-Design Implementation of ADS-B Receiver Using Analog Devices AD9361/AD9364” (SoC Blockset) running on the Zynq® platform. The captured data represents the baseband received signal with a sampling rate of 4 MHz. The data contains 8 frames of extended squitter messages. The ADS-B transmitter modulates the 112-bit extended

squitter messages using 2-bit pulse-position modulation, and adds a 16-bit prefix. Then, to generate 4 MHz data, each 240-bit message is zero-padded and upsampled by 2.

This diagram shows the detailed structure of the **HDLRx** subsystem.



The subsystems listed here are described further in the following sections.

1. **Magnitude Calculation** - Finds the complex modulus of the received input signal

2. **Threshold Calculation** - Calculates the threshold value based on received input signal strength

3. **Correlation with Preamble** - Correlates the received signal with reference signal to detect the preamble

4. **Timing Control** - Provides timing synchronization for the receiver

5. **Bit Process** - Decodes symbols using PPM demodulation

6. **Compute CRC and Frame Validation** - Validates the frame by checking for CRC errors

HDL Optimized ADS-B Receiver

1. **Magnitude Calculation**

The inputs to the **Magnitude Calculation** subsystem are the in-phase (real) and quadrature (imaginary) phase samples. This subsystem outputs the modulus of the complex number. The $\sqrt(I^2 + Q^2)$ can be approximated by the $|L| + 0.4 * |S|$ algorithm described on page 238 of [2].

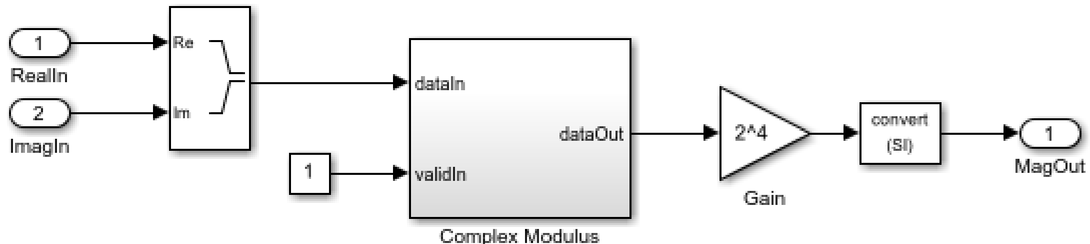
$$\sqrt{I^2 + Q^2} = |L| + 0.4 * |S|$$

where

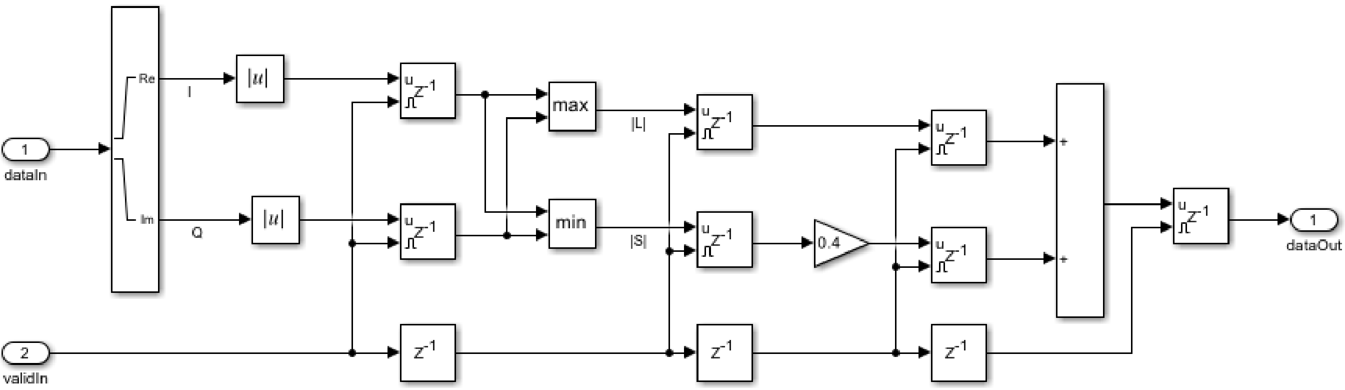
L is the larger value of I or Q

and S is the smaller value of I or Q.

The Gain block converts received input from 12-bit to 16-bit word length.

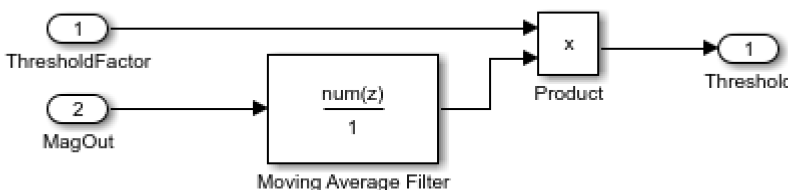


For the implementation of $|L| + 0.4|S|$ algorithm, see the following model.



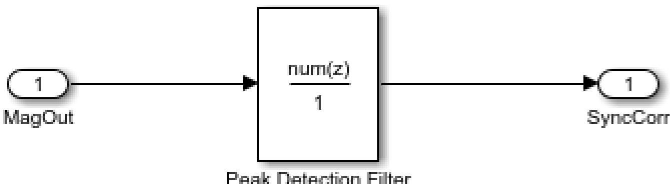
2. **Threshold Calculation**

The **Threshold Calculation** subsystem calculates the signal energy and applies a scaling factor to create a threshold for preamble detection. Moving Average Filter is a serial FIR filter architecture with 32 coefficients that operates on the magnitude values. The coefficients of the FIR filter are selected to find the average energy of the received signal. This example scales the signal energy by 5 to detect valid ADS-B preambles. For details on FIR filter, see Discrete FIR Filter.



3. **Correlation with Preamble**

The **Correlation with Preamble** subsystem correlates the received signal with the ADS-B reference/preamble sequence [1 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0] using a peak detection filter. The peak detection filter is a serial FIR Filter architecture, configured with coefficients that match the preamble sequence. Preamble correlation identifies potential ADS-B transmissions and aligns our bit detection algorithm with the first message bit. The preamble is detected if the peak amplitude exceeds the scaled threshold value. Once the preamble is detected, the correlation value is passed on as input(SyncCorr) to the **Timing Control** block.

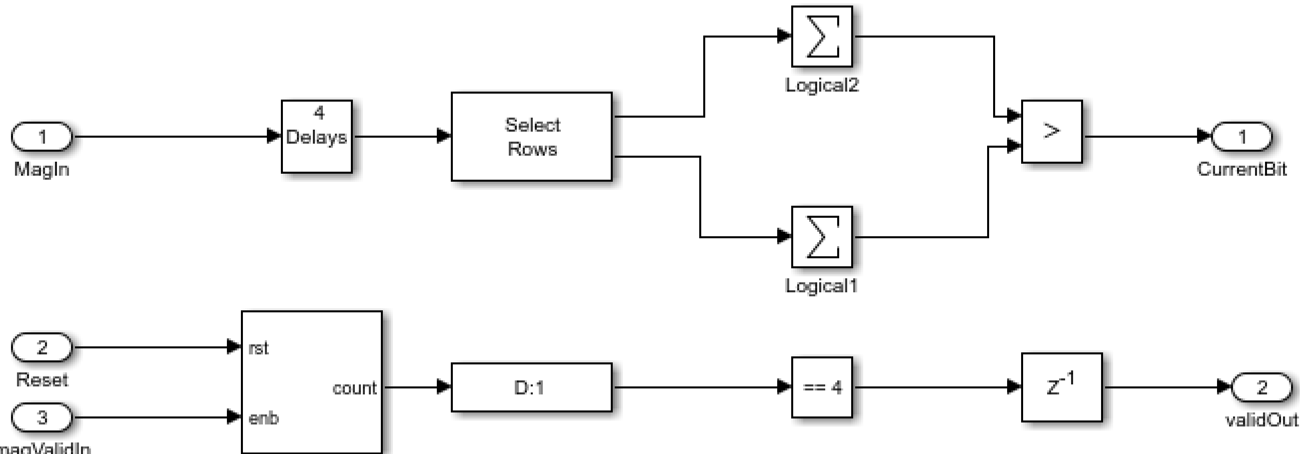


4. **Timing Control**

The **Timing Control** block is a state machine that detects the preamble and generates the control signals ActivateBP and Reset, that indicate the start of frame, end of frame and reset status to the **Bit Process** and **Compute CRC and Frame Validation** blocks.

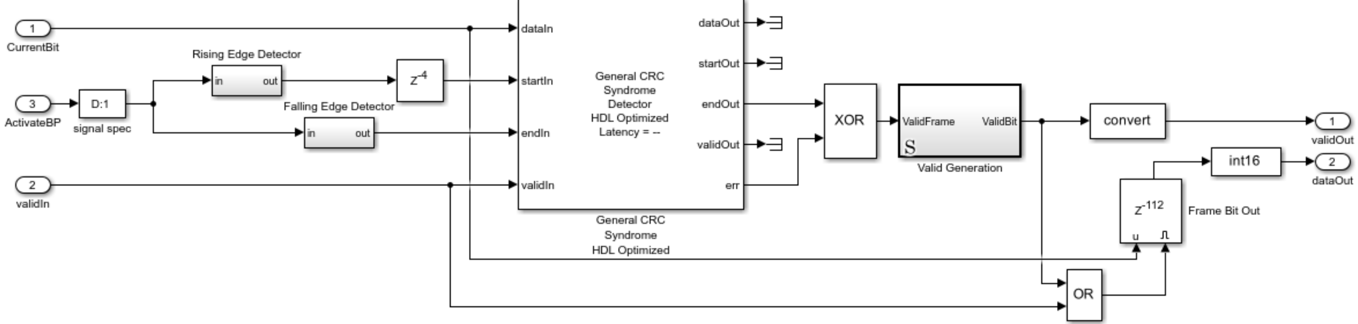
5. **Bit Process**

The **Bit Process** subsystem demodulates and down converts the 4 MHz received signal to a 1 MHz bit sequence. Each data bit is represented by four PPM bits. To demodulate, the block finds the sum of the first two bits and the last two bits of each quadruplet. Then, it compares the sums to determine the original bit value. The output valid signal is asserted every fourth cycle to align with 1 MHz bit sequence.



6. **Compute CRC and Frame Validation**

This subsystem checks for mismatches in the 24-bit checksum of each 88-bit message. The CRC block needs an indication of the frame boundaries to determine which bits are the checksum. The rising edge of the ActivateBP signal generated from the **Timing Control** block indicates the start of frame, and the falling edge indicates the end of the frame. The start signal is delayed to match the demod latency. When the block output err signal is zero, the frame is a valid ADS-B message. The subsystem buffers the message bits until the message is confirmed to have no CRC error.



Launch Map and Log Data

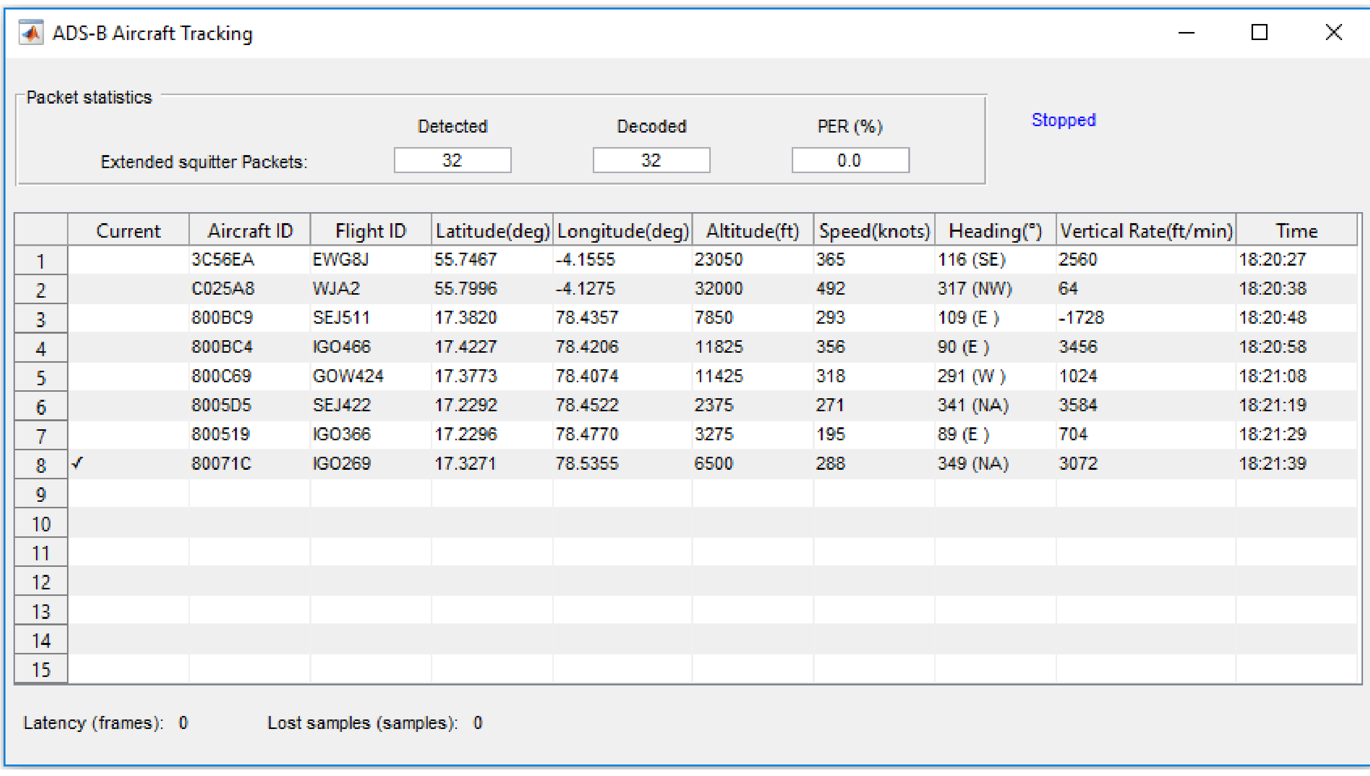
You can launch the map and start text file logging using the two slider switches (Launch Map and Data Logging).

Launch Map - Launch the map where the tracked flights can be viewed. **NOTE:** You must have a Mapping Toolbox™ license to use this feature.

Data Logging - Save the captured data in a TXT file. You can use the saved data for later for post processing.

Results and Displays

The **HDLRx** subsystem demodulates and decodes the ADS-B data and the output is streamed through **Deserializer1D** block and **MapResults** MATLAB function, which produces hexadecimal output information about the aircraft. Each extended squitter Mode S packet contains partial information (any of Aircraft ID, Flight ID, Altitude, Speed, and Location) about the aircraft and the table is built up from multiple messages. The output is obtained as shown in the following diagram. The packet statistics include the number of detected packets, the number of correctly decoded packets, and packet error rate (PER). These aircraft details match the transmitted values from the “HW/SW Co-Design Implementation of ADS-B Receiver Using Analog Devices AD9361/AD9364” (SoC Blockset) example.



HDL Code Generation and Synthesis Results

Pipeline registers have been added to the model to make sure that **HDLRx** subsystem does not have a long critical path. The HDL code generated from the **HDLRx** subsystem was synthesized using Xilinx® Vivado® on a **Zynq** FPGA with the device 7z045ffg900-2, and the design achieves **264.2 MHz** clock frequency, which is sufficient to decode the real-time ADS-B signals. The generated HDL code is tested and verified in the real-time example “HW/SW Co-Design Implementation of ADS-B Receiver Using Analog Devices AD9361/AD9364” (SoC Blockset). To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license. The following table shows the synthesis results of this example.

Synthesis Frequency	264.2 MHz	
Resources	Used	Utilization in %
Slice LUTs	831	0.38
Slice Registers	1689	0.39
DSP48E1	2	0.22
F7 Muxes	24	0.02
F8 Muxes	8	0.01

You can use the commands makehdl and makehdltb to generate HDL code and a test bench for the HDLRx subsystem. To generate the HDL code, use the following command:

```
makehdl('commadsbrxhdl/HDLRx')
```

To generate a test bench, use the following command:

```
makehdltb('commadsbrxhdl/HDLRx')
```

References

1. International Civil Aviation Organization, Annex 10, Volume 4. Surveillance and Collision Avoidance Systems.
2. Marvin E. Frerking, Digital Signal Processing in Communication Systems, Springer Science Business Media, New York,1994.

Generate HDL Code for Viterbi Decoder

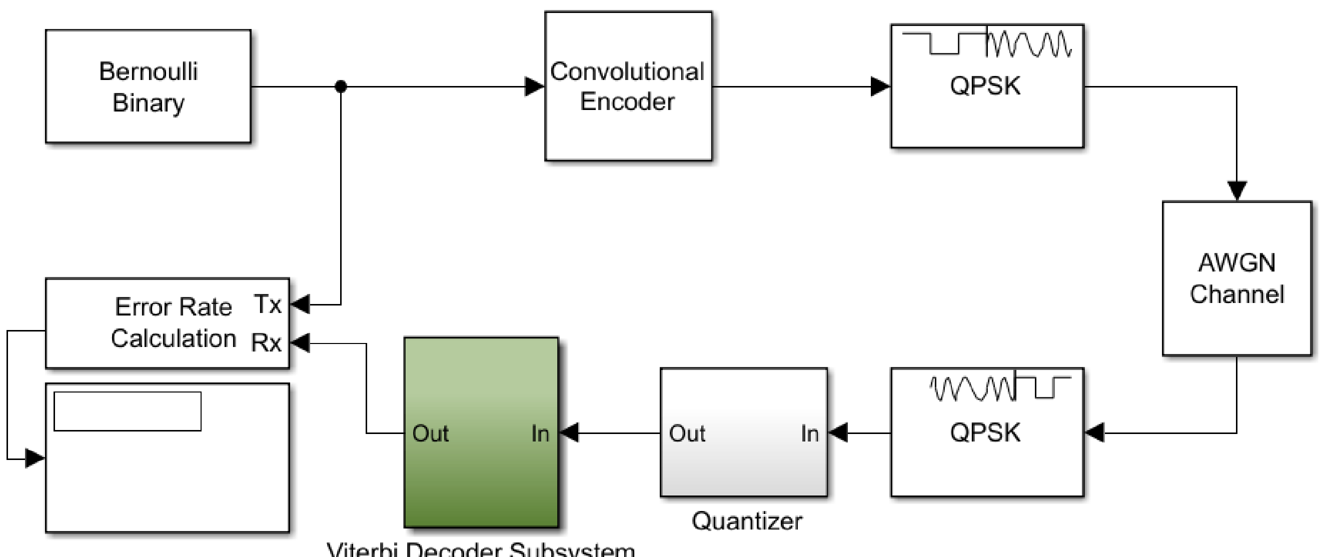
This example shows two different modeling patterns to implement the Viterbi decoding algorithm and generate HDL code. One of the patterns uses a MATLAB Function block. The other pattern uses the Viterbi Decoder block. You need a Communications Toolbox license to use the Viterbi Decoder block.

Model Algorithm with Viterbi Decoder Block

The model `hdlcoder_commviterbi` demonstrates how to generate code for a model that contains a fixed-point Viterbi Decoder block used in soft decision convolutional decoding. To learn more about HDL support for the Viterbi Decoder block see the “HDL Code Generation” (Communications Toolbox) section of the block page in documentation.

To open the model, run this command in the MATLAB Command Window:

```
open_system('hdlcoder_commviterbi');
```



Copyright 2014-2023 The MathWorks, Inc.

In this model, the top-level subsystem `Viterbi Decoder Subsystem` contains a Viterbi Decoder block. To open this subsystem, run these commands:

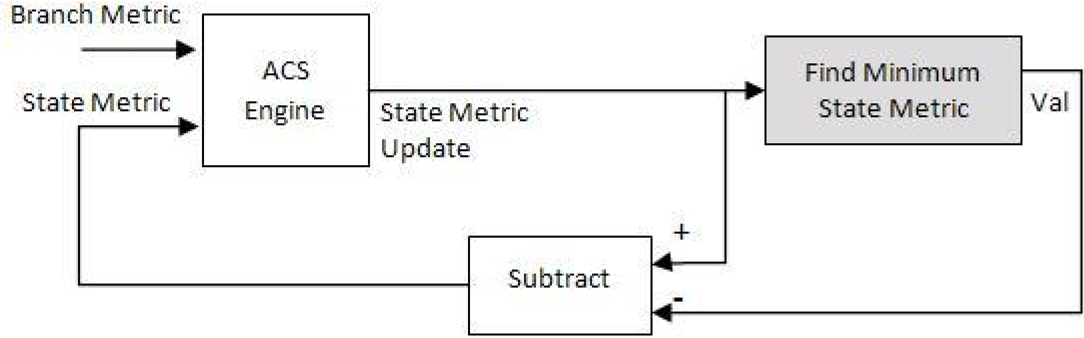
```
subsystem = 'hdlcoder_commviterbi/Viterbi Decoder Subsystem';
open_system(subsystem);
```

The Viterbi Decoding Algorithm

There are three main units in the Viterbi decoding algorithm, the branch metric computation (BMC), add-compare-select (ACS), and traceback decoding. This diagram illustrates the three units in the algorithm:

The Renormalization Method

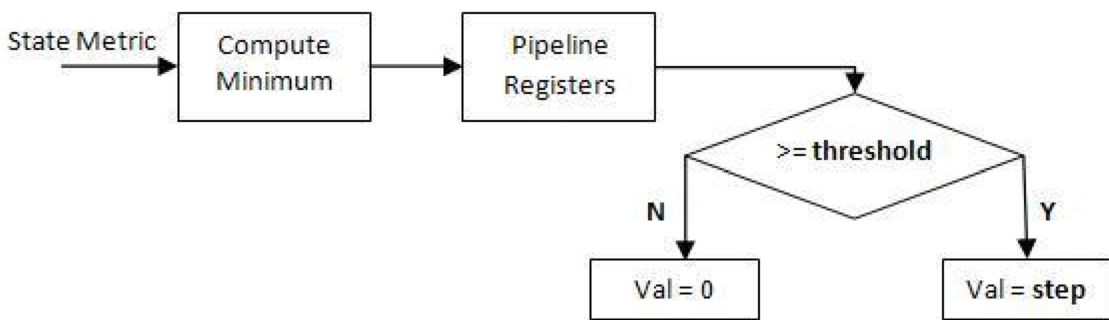
The Viterbi decoding algorithm prevents the overflow of the state metrics in the ACS component by subtracting the minimum value of the state metrics at each time step, as shown in this figure:



Obtaining the minimum value of all the state metric elements in one clock cycle results in a poor clock frequency for the circuit. You can improve the performance of the circuit by adding pipeline registers. However, subtracting the minimum value delayed by the pipeline registers from the state metrics may still lead to overflow. The hardware architecture modifies the renormalization method and avoids the state metric overflow in three steps:

1. The architecture calculates values for the threshold and step parameters based on the trellis structure and the number of soft decision bits.
2. The architecture compares the delayed minimum value to the threshold.
3. If the minimum value is greater than or equal to the threshold value, the implementation subtracts the step value from the state metric. Otherwise, it performs no adjustment.

This figure illustrates the modified renormalization method:



Optimal State Metric Word Length Calculation

The hardware implementation calculates the optimal word length of the state metric and compares it with the value you specify for the **State metric word length** parameter in the Viterbi Decoder block. The hardware architecture uses the optimal value if it is smaller than the one you specify. During code generation, HDL Coder displays a message to show the value in the Diagnostic Viewer. If the calculated value is larger than the value you specify, HDL Coder reports an error message and displays the optimal value.

Applying the calculated optimal state metric word length in the hardware implementation may significantly reduce the hardware resource if the value you specify is too large. For example, if you set 16 bits as the **State metric word length** but only 9 bits are required to achieve the same numerical results, applying the calculated optimal state metric word length in the hardware architecture saves approximately 40% of the register resources. The calculated optimal state metric word length for some typical trellises is displayed in this table:

Decoding rate	Free distance	Example Trellis	Number of soft decision bits	Optimal word length
1/2	10	7, [171 133]	3	8
1/2	10	7, [171 133]	1	5
1/3	15	7, [171 133 165]	3	9
1/3	15	7, [171 133 165]	1	6
1/4	24	9,[463 535 733 745]	3	9
1/4	24	9,[463 535 733 745]	1	7

This model decodes a convolutional code with a decoder rate of 1/2, constraint length of 7, and (171,133) encoding. It uses 3-bit soft decision decoding. The decoder runs at continuous mode with a traceback depth of 32. The state metric word length is 16 bits.

To validate the parameter settings of the Viterbi Decoder block, run these commands:

```
workingdir = tempname;
checkhdl(subsystem,'TargetDirectory',workingdir);
```

```
### Running HDL checks on the model 'hdlcoder_commviterbi'.
### Begin compilation of the model 'hdlcoder_commviterbi'...
```

```
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/5/tpc
### HDL check for 'hdlcoder_commviterbi' complete with 0 errors, 0 warnings, and 5 messages.
```

Running `checkhdl` generates messages that report:

- The default value of **TracebackStagesPerPipeline** HDL block property
- The state metric word length used in the HDL code compared with the one set for **State metric word length** parameter on the Viterbi Decoder block
- The total delay introduced by the pipeline registers with respect to the Viterbi Decoder block

To generate HDL for the subsystem that contains the Viterbi Decoder block, run this command:

```
makehdl(subsystem,'TargetDirectory',workingdir);
```

```
### Working on the model hdlcoder_commviterbi
### Generating HDL for hdlcoder_commviterbi/Viterbi Decoder Subsystem
### Using the config set for model hdlcoder_commviterbi for HDL code generation parameters.
### Running HDL checks on the model 'hdlcoder_commviterbi'.
### Begin compilation of the model 'hdlcoder_commviterbi'...
### Working on the model 'hdlcoder_commviterbi'...
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 20 cycles.
### Working on... GenerateModel
### Begin model generation 'gm_hdlcoder_commviterbi'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\5\tpc2ea2276_9511_4caa_b4c8_1f
### Begin VHDL Code Generation for 'hdlcoder_commviterbi'.
### Working on hdlcoder_commviterbi/Viterbi Decoder Subsystem/Viterbi Decoder/BranchMetric as C:\
### Working on hdlcoder_commviterbi/Viterbi Decoder Subsystem/Viterbi Decoder/ACS/ACSEngine/ACSUn
### Working on hdlcoder_commviterbi/Viterbi Decoder Subsystem/Viterbi Decoder/ACS/ACSEngine as C
### Working on hdlcoder_commviterbi/Viterbi Decoder Subsystem/Viterbi Decoder/ACS/ACSRenorm as C
### Working on hdlcoder_commviterbi/Viterbi Decoder Subsystem/Viterbi Decoder/ACS as C:\TEMP\Bdoc
### Working on hdlcoder_commviterbi/Viterbi Decoder Subsystem/Viterbi Decoder/Traceback/Traceback
### Working on hdlcoder_commviterbi/Viterbi Decoder Subsystem/Viterbi Decoder/Traceback as C:\TEM
### Working on hdlcoder_commviterbi/Viterbi Decoder Subsystem/Viterbi Decoder as C:\TEMP\Bdoc24a_
### Working on hdlcoder_commviterbi/Viterbi Decoder Subsystem as C:\TEMP\Bdoc24a_2528353_7604\ib4
### Generating package file C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\5\tpc2ea2276_9511_4caa_b4c8_1f9
### Code Generation for 'hdlcoder_commviterbi' completed.
### Generating HTML files for code generation report at hdlcoder_commviterbi_codegen_rpt.html
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/5/tpc
### HDL check for 'hdlcoder_commviterbi' complete with 0 errors, 0 warnings, and 5 messages.
### HDL code generation complete.
```

The top-level VHDL file name matches the name of the block in the model. The `Viterbi_Decoder` unit generated in the `Viterbi_Decoder.vhd` file contains three units: `BranchMetric`, `ACS`, and `Traceback`. The `ACS` and `Traceback` units instantiate the `ACSUnit` and `TracebackUnit` units multiple times, respectively. The package file `Viterbi_Decoder_Subsystem_pkg.vhd` includes the data type definitions.

To generate a test bench for the subsystem that contains the Viterbi Decoder block, run this command:

```
makehdltb(subsystem,'TargetDirectory',workingdir);
```

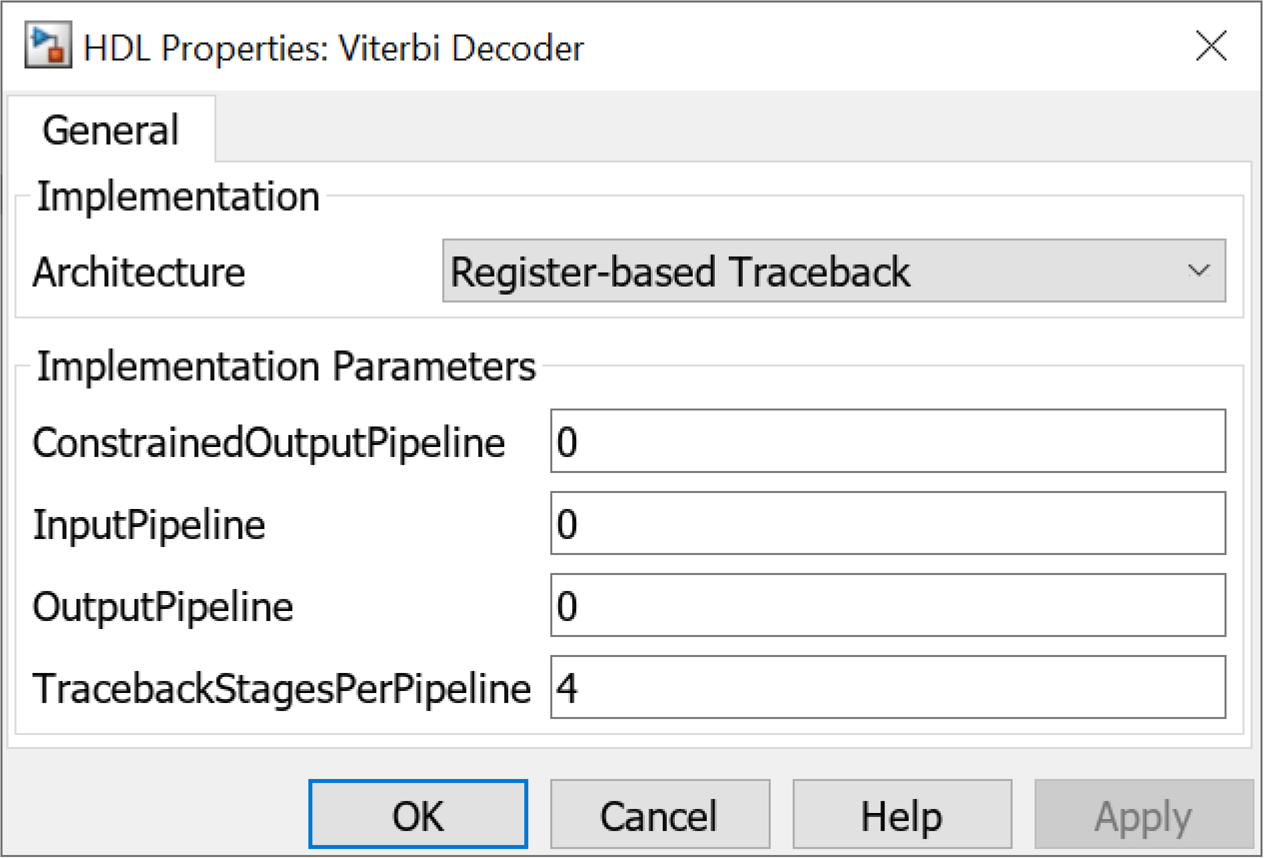
```
### Begin TestBench generation.
### Generating HDL TestBench for 'hdlcoder_commviterbi/Viterbi Decoder Subsystem'.
### Begin compilation of the model 'hdlcoder_commviterbi'...
### Begin compilation of the model 'gm_hdlcoder_commviterbi'...
### Begin simulation of the model 'gm_hdlcoder_commviterbi'...

### Collecting data...
### Generating test bench data file: C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\5\tpc2ea2276_9511_4caa
### Generating test bench data file: C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\5\tpc2ea2276_9511_4caa
### Working on Viterbi_Decoder_Subsystem_tb as C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\5\tpc2ea2276
### Generating package file C:\TEMP\Bdoc24a_2528353_7604\ib462BFE\5\tpc2ea2276_9511_4caa_b4c8_1f9
### HDL TestBench generation complete.
```

Optimizate Traceback Unit

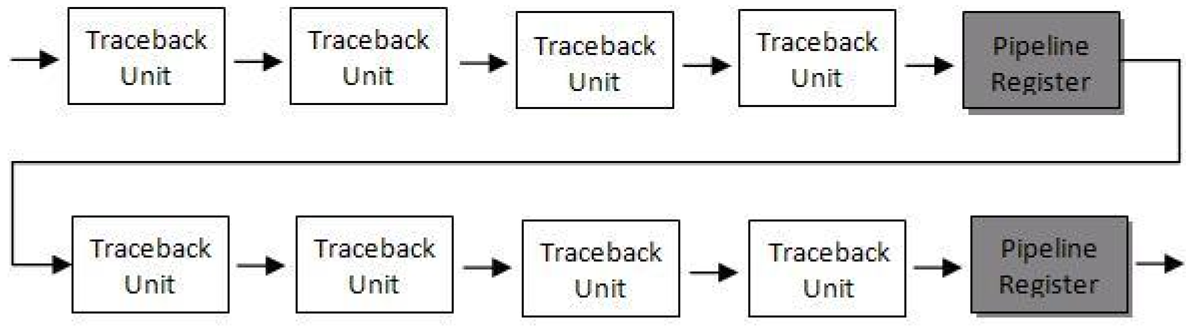
You can optimize the `Traceback` unit that HDL Coder generates in the `Viterbi_Decoder.vhd` file by either pipelining the register-based traceback unit or using the RAM-based traceback architecture.

Register-Based Traceback Pipelining

The Viterbi Decoder block decodes every bit by tracing through a traceback depth you define as the **TracebackStagesPerPipeline** property. Because the block implements a complete traceback for each decision bit, HDL Coder uses registers to store the minimum state index and branch decision in the `Traceback` unit. You can improve the performance of the generated circuit by pipelining this unit. Add pipeline registers to the `Traceback` unit by specifying the number of traceback stages for each pipeline registers. To add pipeline registers, right-click the Viterbi Decoder block and click **HDL Code** > **HDL Block Properties**. Set the **TracebackStagesPerPipeline** to 4.



This setting results in the insertion of a pipeline register for every four traceback units in the model, as illustrated in this figure:

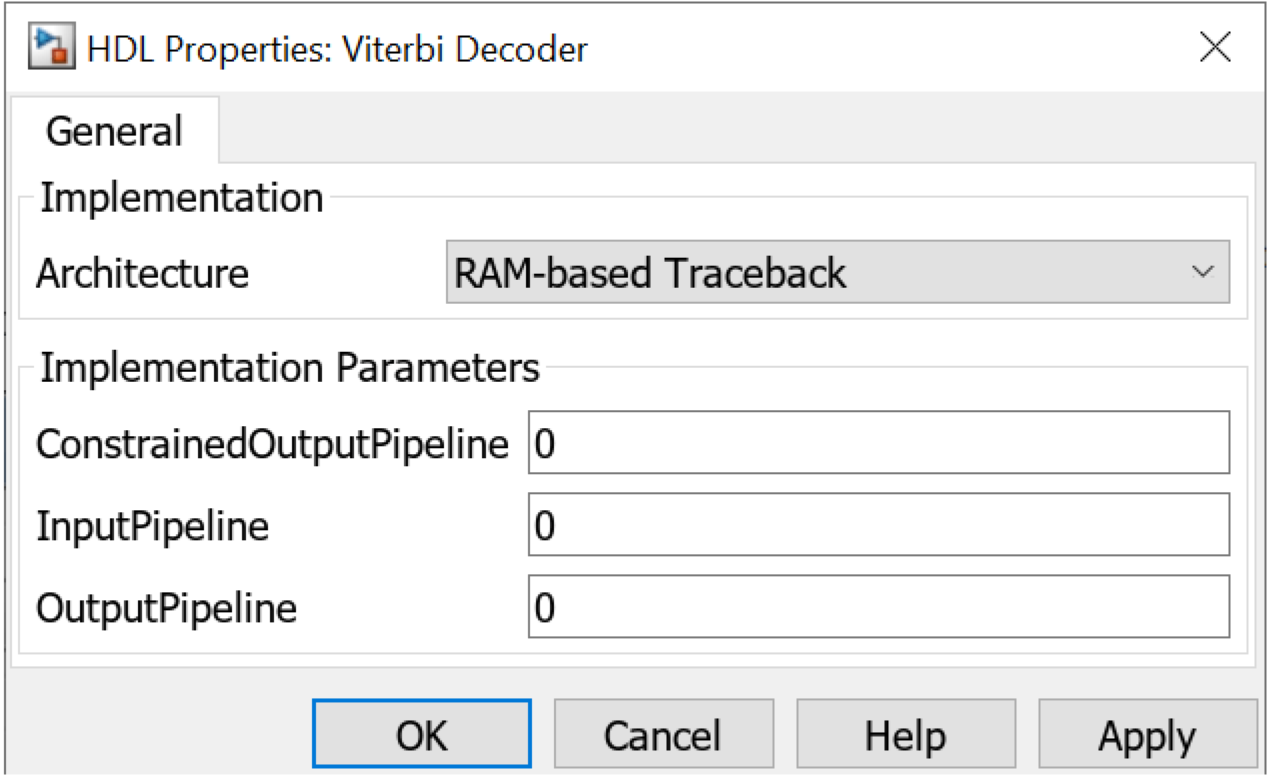


The **TracebackStagesPerPipeline** property balances the circuit performance based on system requirements. A smaller property value adds more registers and increases the speed of the traceback circuit. Increasing the number results in a lower number of registers and a decrease in the circuit speed.

In this example, adjusting the **TracebackStagesPerPipeline** property from 4 to 8 reduces the pipeline register usage in half and changes the circuit speed from 173MHz to 94 MHz.

RAM-Based Traceback Pipelining

Instead of using registers, you can use RAMs to save the survivor branch information. Right-click the Viterbi Decoder block and click **HDL Code > HDL Block Properties**. Set **Architecture** to `RAM-based Traceback`.



There are two differences between the register-based and the RAM-based traceback architectures:

1. The register-based implementation combines the traceback and decode operations into one step and uses the best state found from the minimum operation as the decoding initial state. The RAM-based implementation traces back through one set of data to find the initial state to decode the previous set of data.
2. The register-based implementation decodes one bit after a complete trace back. The RAM-based implementation traces back through *M* samples, decodes the previous *M* bits in reverse order, and releases one bit in order at each clock cycle.

Due to the differences in the two traceback algorithms, the RAM-based implementation produces different numerical results than the register-based traceback. A longer traceback depth, for example, 10 times of constraint length, is recommended in the RAM-based traceback to achieve a similar bit error rate (BER) as the register-based implementation.

The size of RAM required for the implementation depends on the trellis and the traceback depth. This table summarizes the RAM usage for some typical trellis structures.

Constraint length	Example trellis	Traceback depth	Memory size(bits)	Block RAMs
3	3,[5 7]	30	3x30x4	1
4	4,[15 17])	40	3x40x8	1
5	5,[37 27 33 25 35]	50	3x50x16	1
6	6,[73 75 55 65 47 57]	60	3x60x32	1
7	7,[171 133]	70	3x70x64	2
8	(8,[225 331 367])	80	3x80x128	4
9	9,[463 535 733 745]	90	3x90x256	8

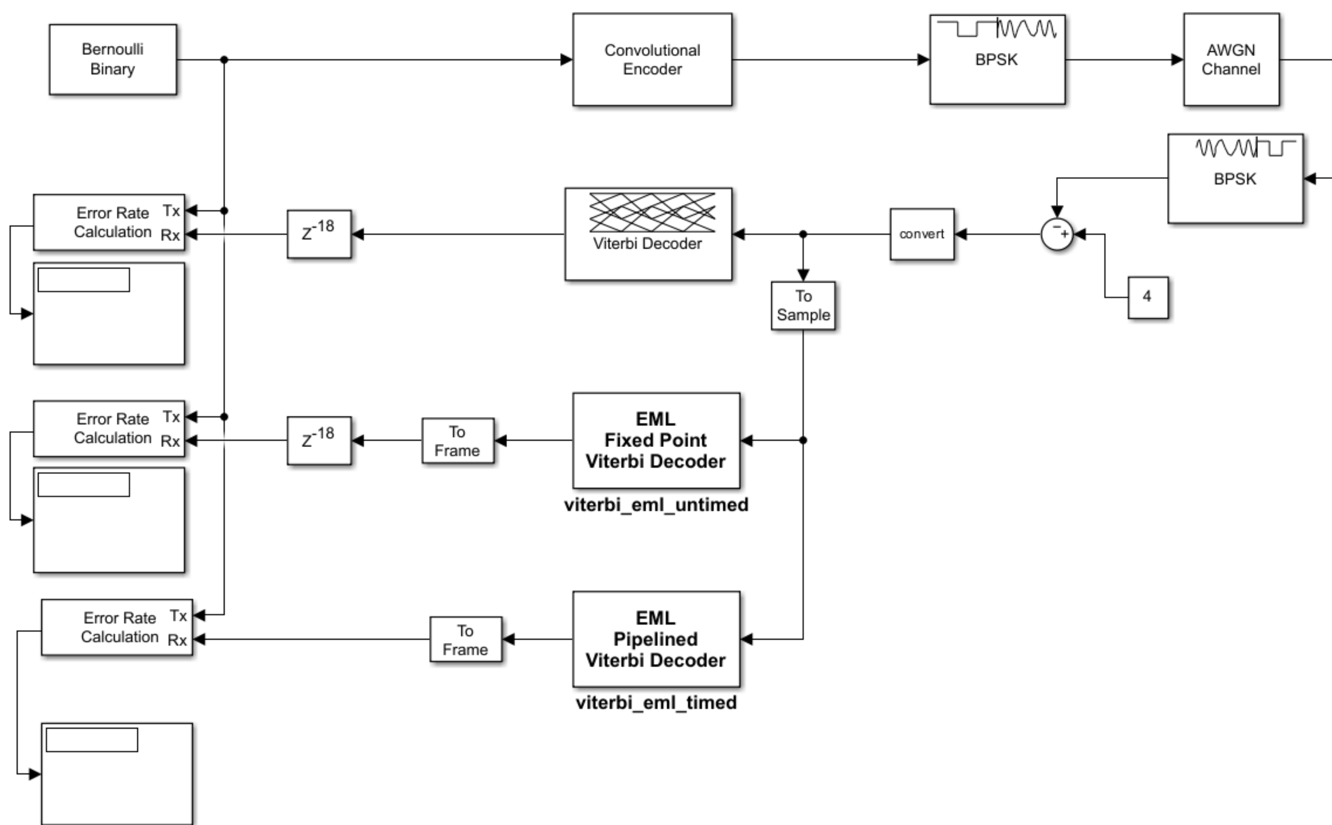
In this example, the RAM-based traceback unit uses 90% fewer registers than the register-based traceback unit with pipelining every 4 stages by using similar clock constraints in synthesis. The two implementations provide a register-RAM tradeoff you can tailor to the individual design.

Model Algorithm with MATLAB Function Block

The model `hdlcoderviterbi2` demonstrates an implementation of a Viterbi decoder that incorporates MATLAB Function blocks for use in simulation and HDL code generation.

To open the model, run this command:

```
open_system('hdlcoderviterbi2');
```



When your design contains MATLAB Function blocks, run the "Check for MATLAB Function block settings" on page 37-18 check in the HDL Code Advisor before you generate HDL code. This check verifies whether you use the recommended MATLAB Function block settings for HDL code generation.

Alternately, run this command:

```
checkhdl('hdlcoderviterbi2/viterbi_eml_timed')
```

```
### Running HDL checks on the model 'hdlcoderviterbi2'.
### Begin compilation of the model 'hdlcoderviterbi2'...
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/5/tp2
### HDL check for 'hdlcoderviterbi2' complete with 0 errors, 0 warnings, and 0 messages.
```

To simulate the model, in the Simulink Toolstrip, on the **Simulation** tab, click **Run**.

Alternately, run this command:

```
sim('hdlcoderviterbi2');
```

To generate HDL for the `viterbi_eml_timed` subsystem, open the **HDL Coder** app. Select the `viterbi_eml_timed` in your model. In the **HDL Code** tab, ensure that **Code for** is set to `viterbi_eml_timed`. To remember the selection, click the pin button to pin the subsystem `viterbi_eml_timed`. Click **Generate HDL Code**.

Alternately, run this command:

```
makehdl('hdlcoderviterbi2/viterbi_eml_timed');
```

```
### Working on the model hdlcoderviterbi2
### Generating HDL for hdlcoderviterbi2/viterbi_eml_timed
### Using the config set for model hdlcoderviterbi2 for HDL code generation parameters.
### Running HDL checks on the model 'hdlcoderviterbi2'.
### Begin compilation of the model 'hdlcoderviterbi2'...
### Working on the model 'hdlcoderviterbi2'...
### Working on... GenerateModel
### Begin model generation 'gm_hdlcoderviterbi2'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at hdlsrc\hdlcoderviterbi2\gm_hdlcoderviterbi2.slx
### Begin VHDL Code Generation for 'hdlcoderviterbi2'.
### Working on hdlcoderviterbi2/viterbi_eml_timed/ACS Unit/ACS as hdlsrc\hdlcoderviterbi2\ACS.vhd
### Working on hdlcoderviterbi2/viterbi_eml_timed/ACS Unit/Renormalize_pipelined as hdlsrc\hdlcod
### Working on hdlcoderviterbi2/viterbi_eml_timed/ACS Unit as hdlsrc\hdlcoderviterbi2\ACS_Unit.vh
### Working on hdlcoderviterbi2/viterbi_eml_timed/TraceBack Unit/TBU1 as hdlsrc\hdlcoderviterbi2\
### Working on hdlcoderviterbi2/viterbi_eml_timed/TraceBack Unit/TBU_OUT as hdlsrc\hdlcoderviterb
### Working on hdlcoderviterbi2/viterbi_eml_timed/TraceBack Unit as hdlsrc\hdlcoderviterbi2\Trace
### Working on hdlcoderviterbi2/viterbi_eml_timed/BMU as hdlsrc\hdlcoderviterbi2\BMU.vhd.
### Working on hdlcoderviterbi2/viterbi_eml_timed as hdlsrc\hdlcoderviterbi2\viterbi_eml_timed.vh
### Generating package file hdlsrc\hdlcoderviterbi2\viterbi_eml_timed_pkg.vhd.
### Code Generation for 'hdlcoderviterbi2' completed.
### Generating HTML files for code generation report at hdlcoderviterbi2_codegen_rpt.html
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/5/tp2
### HDL check for 'hdlcoderviterbi2' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

To generate a test bench for the `viterbi_eml_timed` subsystem, on the **HDL Code** tab, click **Generate Testbench**.

Alternately, run this command:

```
makehdltb('hdlcoderviterbi2/viterbi_eml_timed');
```

```
### Begin TestBench generation.
### Generating HDL TestBench for 'hdlcoderviterbi2/viterbi_eml_timed'.
### Begin compilation of the model 'hdlcoderviterbi2'...
### Begin compilation of the model 'gm_hdlcoderviterbi2'...
### Begin simulation of the model 'gm_hdlcoderviterbi2'...

### Collecting data...
### Generating test bench data file: hdlsrc\hdlcoderviterbi2\In1.dat.
### Generating test bench data file: hdlsrc\hdlcoderviterbi2\Out1_expected.dat.
### Working on viterbi_eml_timed_tb as hdlsrc\hdlcoderviterbi2\viterbi_eml_timed_tb.vhd.
### Generating package file hdlsrc\hdlcoderviterbi2\viterbi_eml_timed_tb_pkg.vhd.
### HDL TestBench generation complete.
```

References

1. Clark, George C., and J. Bibb Cain. *Error-Correction Coding for Digital Communications*. Springer US, 1981.
2. Feygin, G., and P. Gulak. "Architectural Tradeoffs for Survivor Sequence Memory Management in Viterbi Decoders." *IEEE Transactions on Communications*, vol. 41, no. 3, Mar. 1993, pp. 425–29.

See Also

Viterbi Decoder

Related Examples

- "HDL Code Generation from Viterbi Decoder System Object" on page 1-46

Design Video Processing Algorithms for HDL in Simulink

This example shows how to design a hardware-targeted image filter using Vision HDL Toolbox™ blocks. It also uses Computer Vision Toolbox™ blocks.

The key features of a model for hardware-targeted video processing in Simulink® are:

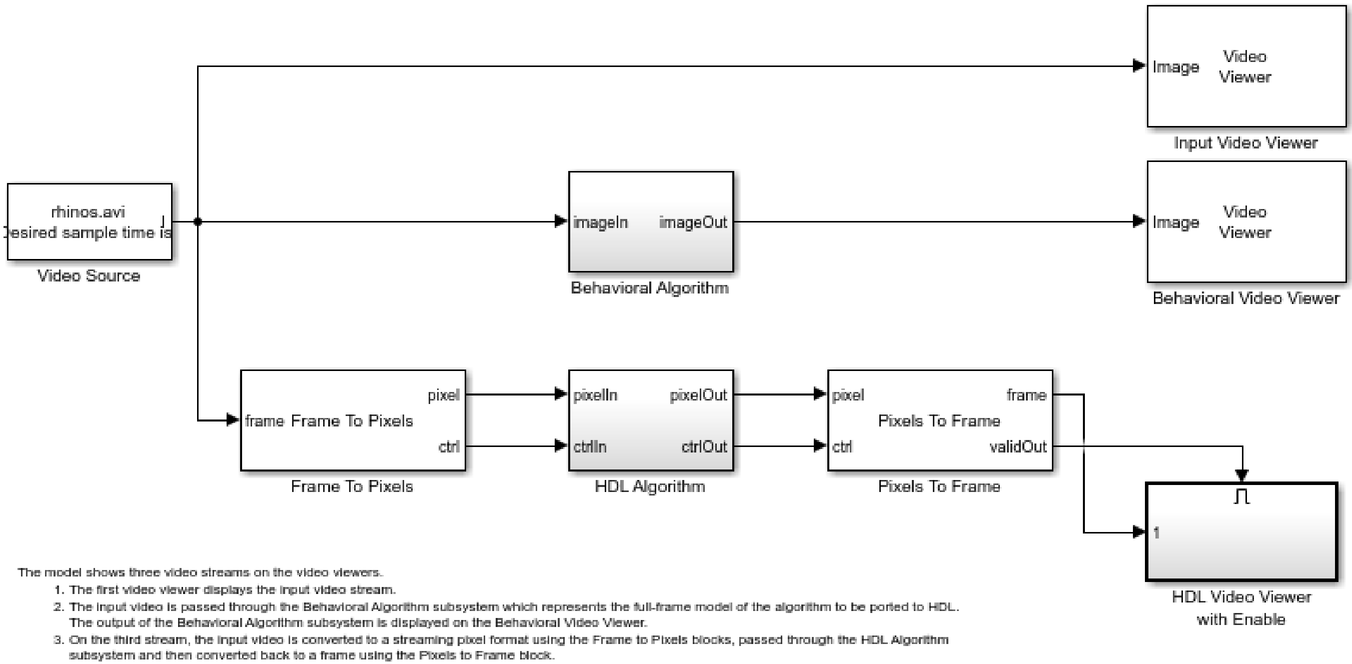
- **Streaming pixel interface**: Blocks in Vision HDL Toolbox use a streaming pixel interface. Serial processing is efficient for hardware designs, because less memory is required to store pixel data for computation. The serial interface allows the block to operate independently of image size and format and makes the design more resilient to video timing errors. For further information, see "Streaming Pixel Interface" (Vision HDL Toolbox).
- **Subsystem targeted for HDL code generation**: Design a hardware-friendly pixel-streaming video processing model by selecting blocks from the Vision HDL Toolbox libraries. The part of the design targeted for HDL code generation must be in a separate subsystem.
- **Conversion to frame-based video**: For verification, you can display frame-based video or compare the result of your hardware-compatible design with the output of a Simulink behavioral model. Vision HDL Toolbox provides a block that allows you to deserialize the output of your design.

Open Model Template

This tutorial uses a Simulink model template to get started.

Click the Simulink button, or type `simulink` at the MATLAB® command prompt. On the Simulink start page, find the Vision HDL Toolbox section, and click the Basic Model template.

The template creates a new model that you can customize. Save the model with a new name.



Import Data

The template includes a Video Source block that contains a 240p video sample. Each pixel is a scalar `uint8` value representing intensity. A best practice is to design and debug your design using a small frame size for quick debug cycles, before scaling up to larger image sizes. You can use this 240p source to debug a design targeted for 1080p video.

Serialize Data

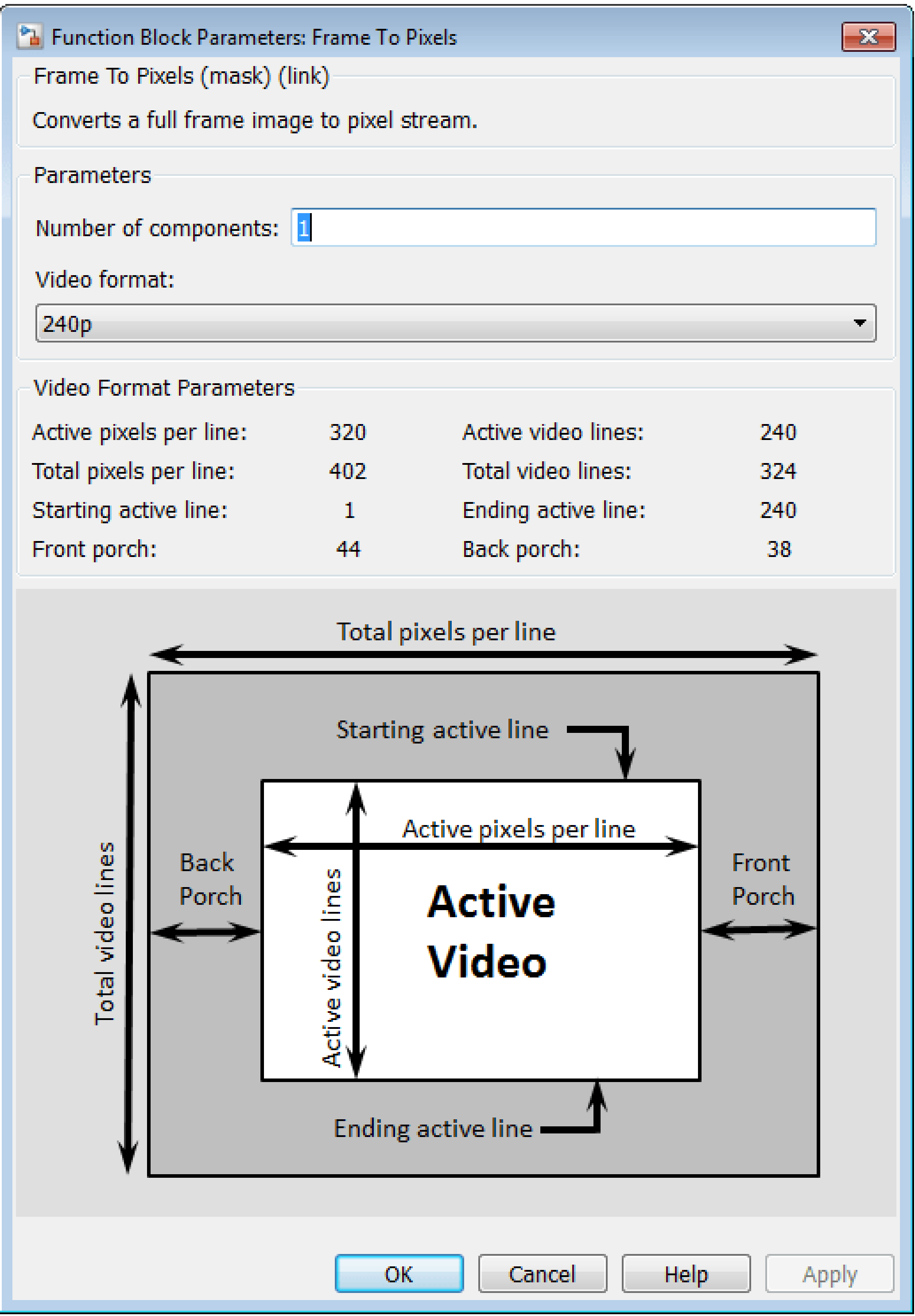
The Frame To Pixels block converts framed video to a stream of pixels and control structures. This block provides input for a subsystem targeted for HDL code generation, but it does not itself support HDL code generation.

The template includes an instance of this block. To simulate with a standard video format, choose a predefined video padding format to match your input source. To simulate with a custom-size image, choose the dimensions of the inactive regions that you want to surround the image with. This tutorial uses a standard video format.

Open the Frame To Pixels block dialog box to view the settings. The source video is in 240p grayscale format. A scalar integer represents the intensity value of each pixel. To match the input video, set **Number of components** to 1, and the **Video format** to 240p.

Note: The sample time of the video source must match the total number of pixels in the frame size you select in the Frame To Pixels block. Set the sample time to **Total pixels per line** × **Total lines**.

In the `InitFcn` callback, the template creates a workspace variable, `totalPixels`, for the sample time of a 240p frame.



Design HDL-Compatible Model

Design a subsystem targeted for HDL code generation, by modifying the HDL Algorithm subsystem. The subsystem input and output ports use the streaming pixel format described in the previous section. Open the HDL Algorithm subsystem to edit it.

In the Simulink Library Browser, click Vision HDL Toolbox. You can also open this library by typing `visionhdllib` at the MATLAB command prompt.

Select an image processing block. This example uses the Image Filter (Vision HDL Toolbox) block from the Filtering sublibrary. You can also access this library by typing `visionhdlfilter` at the MATLAB command prompt. Add the Image Filter block to the HDL Algorithm subsystem and connect the ports.

Open Image Filter block and make the following changes:

- Set **Filter coefficients** to `ones(4,4)/16` to implement a 4×4 blur operation.
- Set **Padding method** to `Symmetric`.
- Set **Line buffer size** to a power of 2 that accommodates the active line size of the largest required frame format. This parameter does not affect simulation speed, so it does not need to be reduced when simulating with a small test image. The default, 2048, accommodates 1080p video format.
- On the **Data Types** tab, under **Data Type**, set **Coefficients** to `fixdt(0,1,4)`.

Design Behavioral Model

You can visually or mathematically compare your HDL-targeted design with a behavioral model to verify the hardware design and monitor quantization error. The template includes a Behavioral Model subsystem with frame-based input and output ports for this purpose. Double-click on the Behavioral Model to edit it.

For this tutorial, add the 2-D FIR Filter (Computer Vision Toolbox) block from Computer Vision Toolbox™. This block filters the entire frame at once.

Open the 2-D FIR Filter block and make the following changes to match the configuration of the Image Filter block from Vision HDL Toolbox:

- Set **Coefficients** to `ones(4,4)/16` to implement a 4×4 blur operation.
- Set **Padding options** to `Symmetric`.
- On the **Data Types** tab, under **Data Type**, set **Coefficients** to `fixdt(0,2,4)`.

Deserialize Filtered Pixel Stream

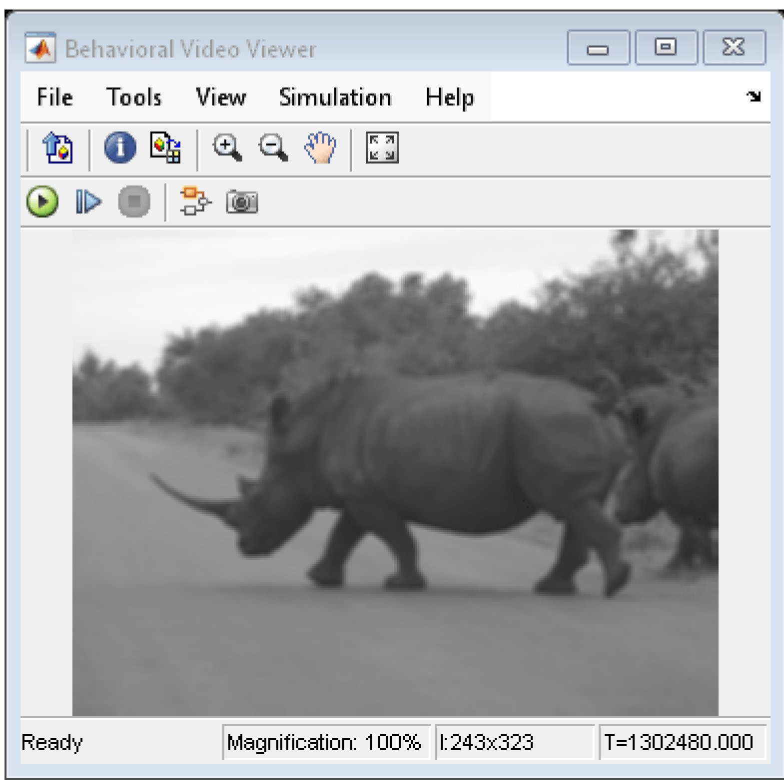
Use the Pixels To Frame block included in the template to deserialize the data for display.

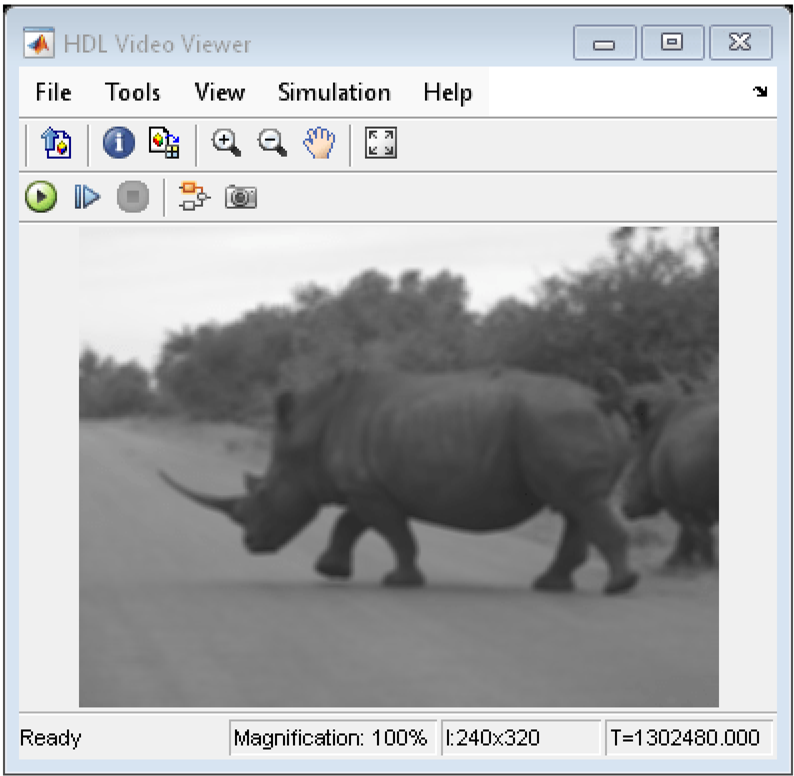
Open the Pixels To Frame block. Set the image dimension properties to match the input video and the settings you specified in the Frame To Pixels block. For this tutorial, the **Number of components** is

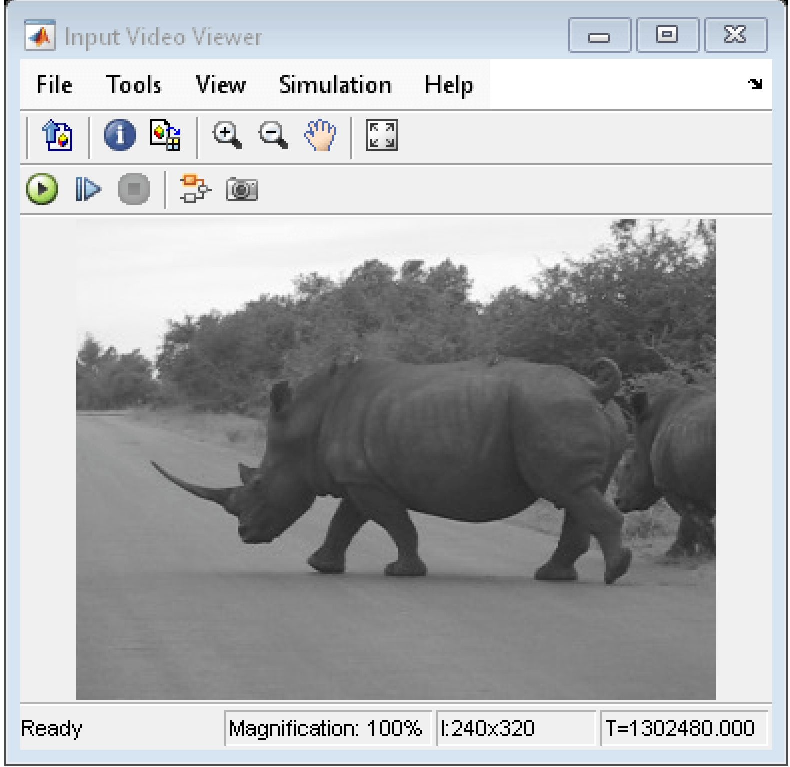
set to 1 and the **Video format** is set to 240p. The block converts the stream of output pixels and control signals back to a matrix representing a frame.

Display Results and Compare to Behavioral Model

Use the Video Viewer blocks included in the template to compare the output frames visually. The `validOut` signal of the Pixels To Frame block is connected to the `Enable` port of the viewer. Run the model to display the results.




HDL Video Viewer
File Tools View Simulation Help
Ready
Magnification: 100%
I:240x320
T=1302480.000


Input Video Viewer
File Tools View Simulation Help
Ready
Magnification: 100%
I:240x320
T=1302480.000

Generate HDL Code

Once your design is working in simulation, you can use HDL Coder™ to generate HDL code for the HDL Algorithm subsystem. See “Generate HDL Code from Simulink” (Vision HDL Toolbox).

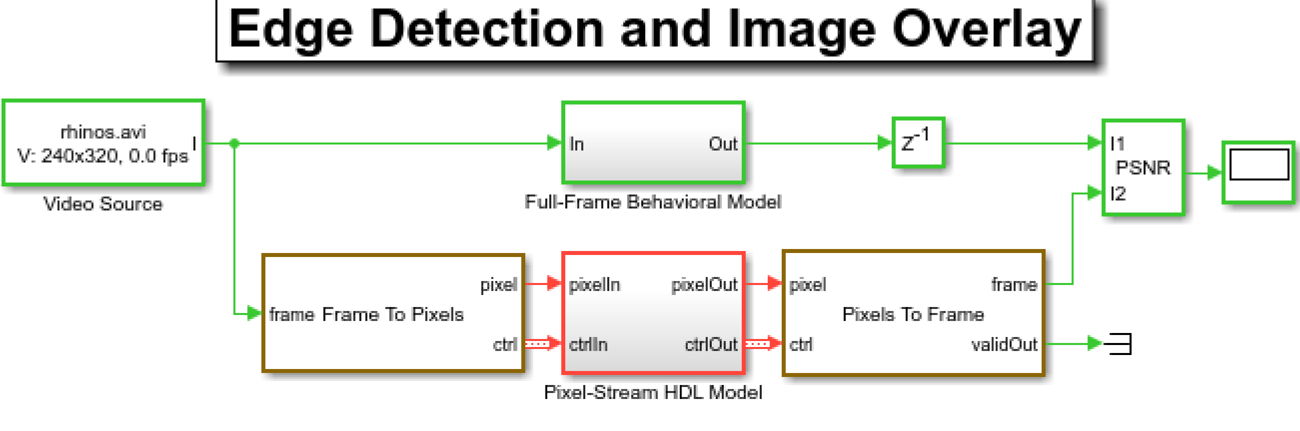
Edge Detection and Image Overlay

This example shows how to detect and highlight object edges in a video stream. The behavior of the pixel-stream Sobel Edge Detector block, video stream alignment, and overlay, is verified by comparing the results with the same algorithm calculated by the full-frame blocks from the Computer Vision Toolbox™.

This example model provides a hardware-compatible algorithm. You can implement this algorithm on a board using a Xilinx® Zynq® reference design. See “Developing Vision Algorithms for Zynq-Based Hardware” (SoC Blockset).

Structure of the Example

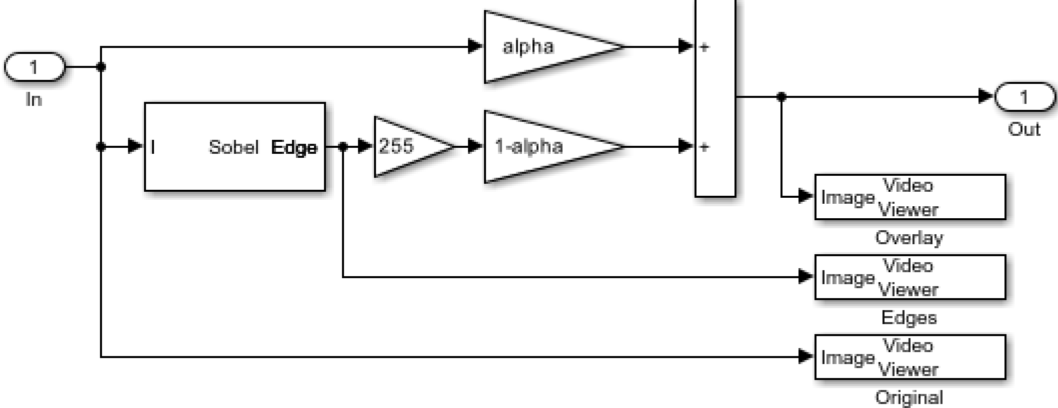
The EdgeDetectionAndOverlayHDL.slx system is shown below.



The difference in the color of the lines feeding the **Full-Frame Behavioral Model** and **Pixel-Stream HDL Model** subsystems indicates the change in the image rate on the streaming branch of the model. This rate transition is because the pixel stream is sent out in the same amount of time as the full video frames and therefore it is transmitted at a higher rate.

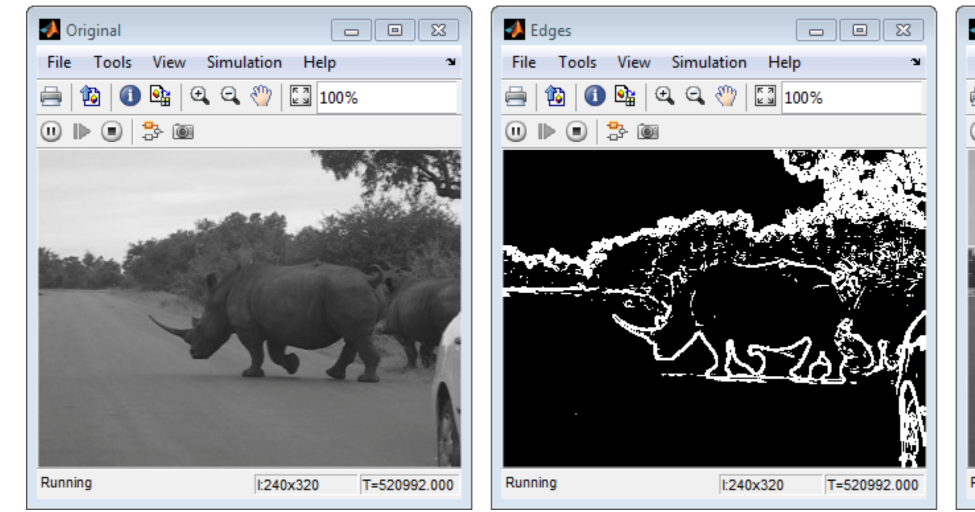
Full-Frame Behavioral Model

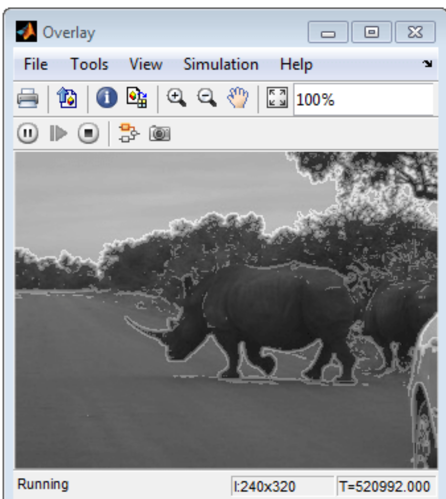
The following diagram shows the structure of the **Full-Frame Behavioral Model** subsystem, which employs the frame-based **Edge Detection** block.



Given that the frame-based **Edge Detection** block does not introduce latency, image overlay is performed by weighting the source image and the **Edge Detection** output image, and adding them together in a straightforward manner.

One frame of the source video, the edge detection result, and the overlaid image are shown from left to right in the diagram below.

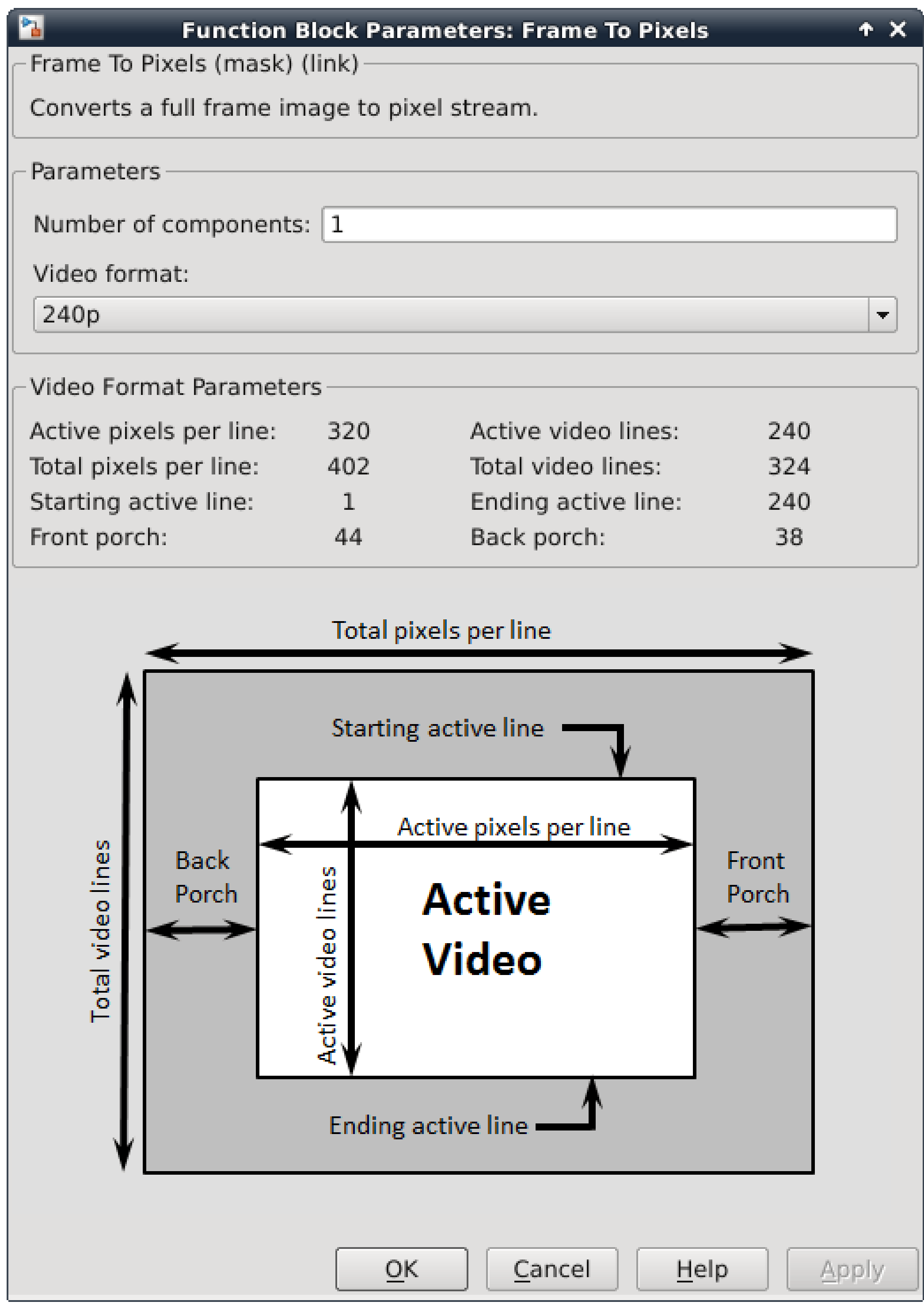




It is a good practice to develop a behavioral system using blocks that process full image frames, the **Full-Frame Behavioral Model** subsystem in this example, before moving forward to working on an FPGA-targeting design. Such a behavioral model helps verify the video processing design. Later on, it can serve as a reference for verifying the implementation of the algorithm targeted to an FPGA. Specifically, the **PSNR** (peak signal-to-noise ratio) block at the top level of the model compares the results from full-frame processing with those from pixel-stream processing.

Frame To Pixels: Generating a Pixel Stream

The task of the **Frame To Pixels** is to convert a full frame image to pixel stream. To simulate the effect of horizontal and vertical blanking periods found in real life hardware video systems, the active image is augmented with non-image data. For more information on the streaming pixel protocol, see “Streaming Pixel Interface” (Vision HDL Toolbox). The **Frame To Pixels** block is configured as shown:



The **Number of components** field is set to 1 for grayscale image input, and the **Video format** field is 240p to match that of the video source.

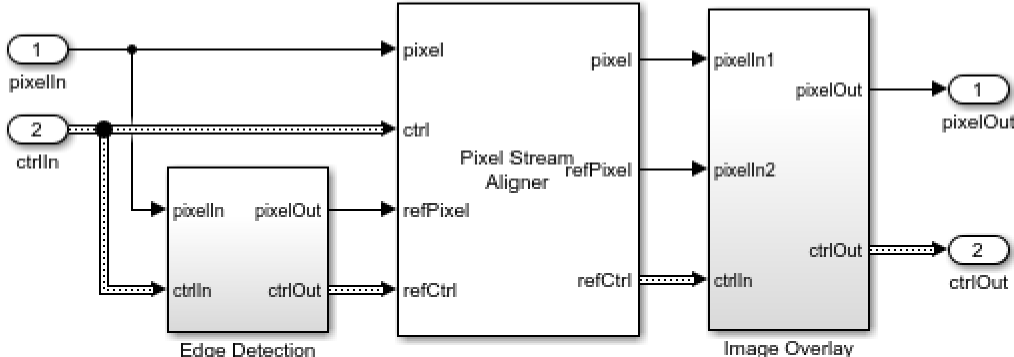
In this example, the Active Video region corresponds to the 240x320 matrix of the dark image from the upstream **Corruption** block. Six other parameters, namely, **Total pixels per line**, **Total video**

lines, **Starting active line**, **Ending active line**, **Front porch**, and **Back porch** specify how many non-image data will be augmented on the four sides of the Active Video. For more information, see the Frame To Pixels (Vision HDL Toolbox) block reference page.

Note that the sample time of the **Video Source** is determined by the product of **Total pixels per line** and **Total video lines**.

Pixel-Stream Edge Detection and Image Overlay

The **Pixel-Stream HDL Model** subsystem is shown in the diagram below. You can generate HDL code from this subsystem.



Due to the nature of pixel-stream processing, unlike the **Edge Detection** block in the **Full-Frame Behavioral Model**, the **Edge Detector** block from the Vision HDL Toolbox™ will introduce latency. The latency prevents us from directly weighting and adding two images to obtain the overlaid image. To address this issue, the **Pixel Stream Aligner** block is used to synchronize the two pixel streams before the sum.

To properly use this block, refPixel and refCtrl must be connected to the pixel and control bus that are associated with a delayed pixel stream. In our example, due to the latency introduced by the **Edge Detector**, the pixel stream coming out of the **Edge Detection** subsystem is delayed with respect to that feeding into it. Therefore, the upstream source of refPixel and refCtrl are the pixelOut and ctrlOut signals from the **Edge Detection** subsystem.

Pixels To Frame: Converting Pixel Stream Back to Full Frame

As a companion to **Frame To Pixels** that converts a full image frame to pixel stream, the **Pixels To Frame** block, reversely, converts the pixel stream back to the full frame by making use of the synchronization signals. Since the output of the **Pixels To Frame** block is a 2-D matrix of a full image, there is no need to further carry on the bus containing five synchronization signals.

The **Number of components** field and the **Video format** fields of both Frame To Pixels and Pixels To Frame are set at 1 and 240p, respectively, to match the format of the video source.

Verifying the Pixel Stream Processing Design

While building the streaming portion of the design, the **PSNR** block continuously verifies results against the original full-frame design. The **Delay** block on the top level of the model time-aligns the 2-D matrices for a fair comparison. During the course of the simulation, the **PSNR** block should give **inf** output, indicating that the output image from the **Full-Frame Behavioral Model** matches the image generated from the stream processing **Pixel-Stream HDL Model**.

Exploring the Example

The example allows you to experiment with different threshold and alpha values to examine their effect on the quality of the overlaid images. Specifically, two workspace variables $thresholdValue$ and $alpha$ with initial values 7 and 0.8, respectively, are created upon opening the model. You can modify their values using the MATLAB® command line as follows:

```
thresholdValue=8
alpha=0.5
```

The updated $thresholdValue$ will be propagated to the **Threshold** field of the **Edge Detection** block inside the **Full-Frame Behavioral Model** and the **Edge Detector** block inside **Pixel-Stream HDL Model/Edge Detection**. The $alpha$ value will be propagated to the **Gain1** block in the **Full-Frame Behavioral Model** and **Pixel-Stream HDL Model/Image Overlay**, and the value of $1 - alpha$ goes to **Gain2** blocks. Closing the model clears both variables from your workspace.

In this example, the valid range of $thresholdValue$ is between 0 and 256, inclusive. Setting $thresholdValue$ equal to or greater than 257 triggers a message **Parameter overflow occurred for 'threshold'**. The higher you set the $thresholdValue$, the smaller the amount of edges the example finds in the video.

The valid range of $alpha$ is between 0 and 1, inclusive. It determines the weights for edge detection output image and the original source image before adding them. The overlay operation is a linear interpolation according to the following formula.

```
overlaid image = alpha*source image + (1-alpha)*edge image.
```

Therefore, when $alpha = 0$, the overlaid image is the edge detection output, and when $alpha = 1$ it becomes the source image.

Generate HDL Code and Verify Its Behavior

To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command:

```
makehdl('EdgeDetectionAndOverlayHDL/Pixel-Stream HDL Model');
```

To generate a test bench, use the following command:

```
makehdltb('EdgeDetectionAndOverlayHDL/Pixel-Stream HDL Model');
```

Lane Detection

This example shows how to implement a lane-marking detection algorithm for FPGAs.

Lane detection is a critical processing stage in Advanced Driving Assistance Systems (ADAS). Automatically detecting lane boundaries from a video stream is computationally challenging and therefore hardware accelerators such as FPGAs and GPUs are often required to achieve real time performance.

In this example model, an FPGA-based lane candidate generator is coupled with a software-based polynomial fitting engine, to determine lane boundaries.

Download Input File

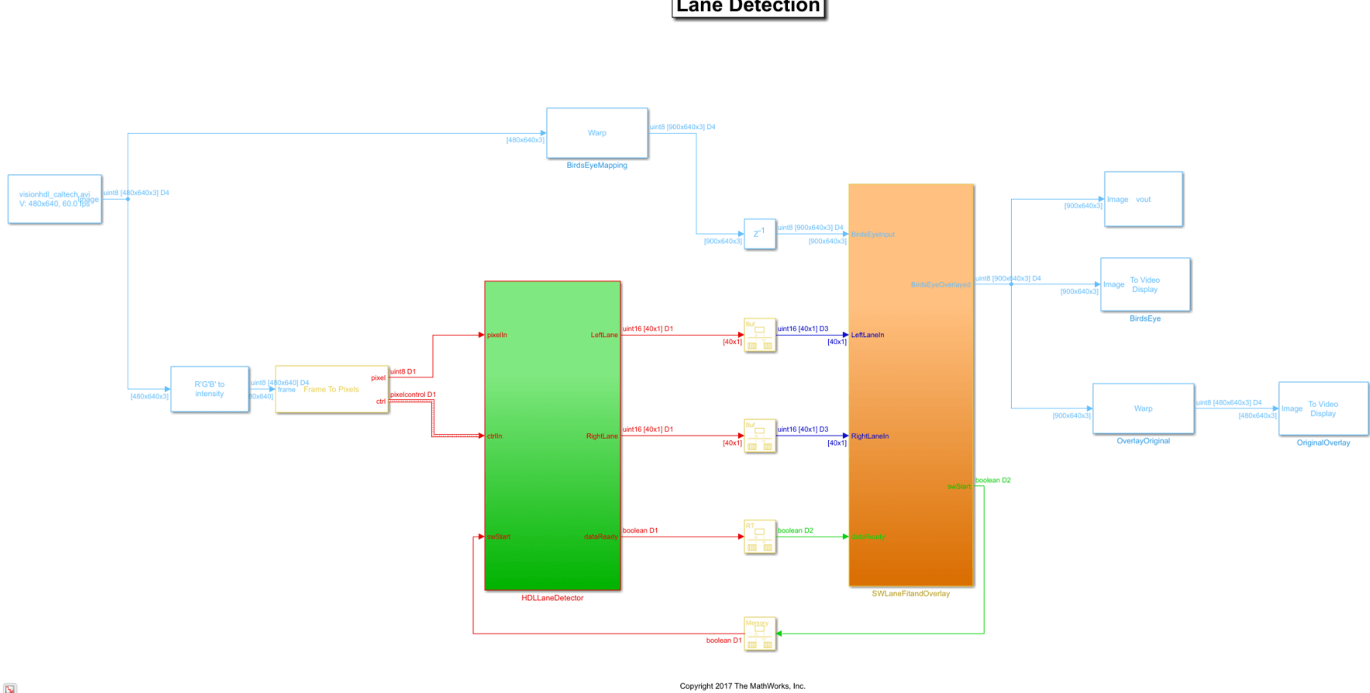
This example uses the visionhdl_caltech.avi file as an input. The file is approximately 19 MB in size. Download the file from the MathWorks website and unzip the downloaded file.

```
laneZipFile = matlab.internal.examples.downloadSupportFile('visionhdl_hdlcoder','caltech_dataset
[outputFolder,~,~] = fileparts(laneZipFile);
unzip(laneZipFile,outputFolder);
caltechVideoFile = fullfile(outputFolder,'caltech_dataset');
addpath(caltechVideoFile);
```

System Overview

The LaneDetectionHDL.slx system is shown below. The HDLLaneDetector subsystem represents the hardware accelerated part of the design, while the SWLaneFitandOverlay subsystem represent the software based polynomial fitting engine. Prior to the Frame to Pixels block, the RGB input is converted to intensity color space.

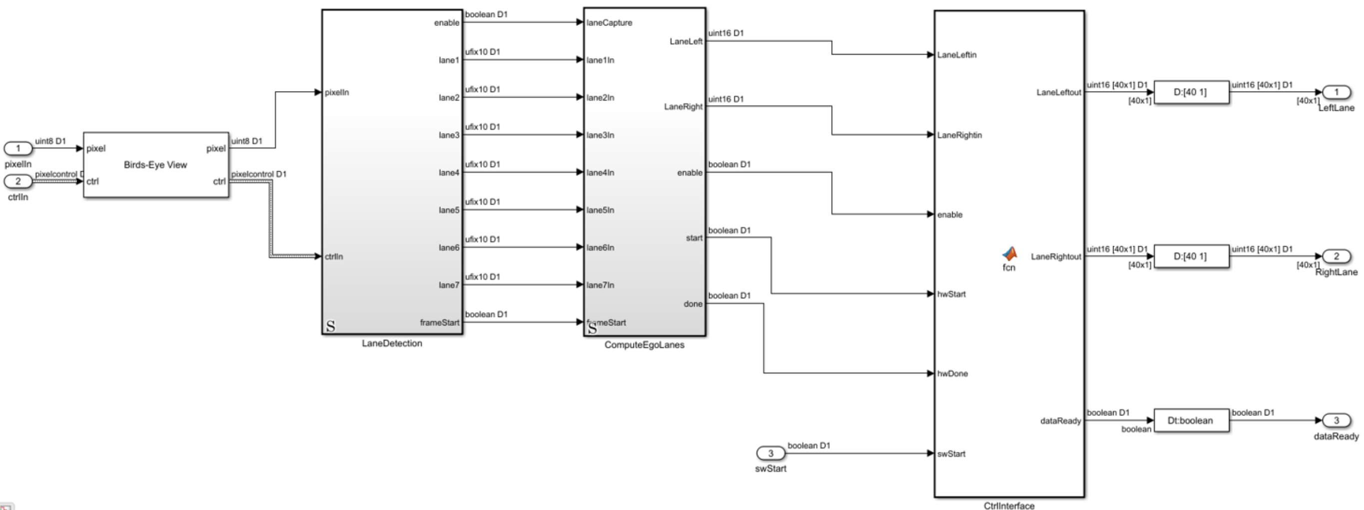
```
modelname = 'LaneDetectionHDL';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
```



HDL Lane Detector

The HDL Lane Detector represents the hardware-accelerated part of the design. This subsystem receives the input pixel stream from the front-facing camera source, transforms the view to obtain the birds-eye view, locates lane marking candidates from the transformed view and then buffers them up into a vector to send to the software side for curve fitting and overlay.

```
set_param(modelname, 'SampleTimeColors', 'off');
open_system([modelname '/HDLLaneDetector'],'force');
```



Birds-Eye View

The Birds-Eye View block transforms the front-facing camera view to a birds-eye perspective. Working with the images in this view simplifies the processing requirements of the downstream lane detection algorithms. The front-facing view suffers from perspective distortion, causing the lanes to converge at the vanishing point. The perspective distortion is corrected by applying an inverse perspective transform.

The Inverse Perspective Mapping (IPM) is given by the following expression:

$$(\hat{x}, \hat{y}) = round\left(\frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}, \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}\right)$$

The homography matrix, *h*, is derived from four intrinsic parameters of the physical camera setup, namely the focal length, pitch, height, and principle point (from a pinhole camera model). For more details, refer to the Computer Vision Toolbox™ documentation.

You can estimate the homography matrix by using the Computer Vision Toolbox™ `estgeotform2d` function or the Image Processing Toolbox™ `fitgeotform2d` function to create a `projtform2d` object. These functions require a set of matched points between the source frame and birds-eye view frame. The source frame points are taken as the vertices of a trapezoidal region of interest, and can extend past the source frame limits to capture a larger region. For the trapezoid shown the point mapping is:

c d
a b

$$sourcePoints = [c_x, c_y;\ d_x, d_y;\ a_x, a_y;\ b_x, b_y]$$

$$birdsEyePoints = [1, 1;\ bAPPL, 1;\ 1, bAVL;\ bAPPL, bAVL]$$

Where *bAPPL* and *bAVL* are the birds-eye view active pixels per line and active video lines respectively.

Direct evaluation of the source (front-facing) to destination (birds-eye) mapping in real time on FPGA/ASIC hardware is challenging. The requirement for division along with the potential for non-sequential memory access from a frame buffer mean that the computational requirements of this part of the design are substantial. Therefore instead of directly evaluating the IPM calculation in real time, an offline analysis of the input to output mapping has been performed and used to pre-compute a mapping scheme. This is possible as the homography matrix is fixed after factory calibration/installation of the camera, due to the camera position, height and pitch being fixed.

In this particular example, the birds-eye output image is a frame of [700x640] dimensions, whereas the front-facing input image is of [480x640] dimensions. There is not sufficient blanking available in order to output the full birds-eye frame before the next front-facing camera input is streamed in. The Birds-Eye view block will therefore not accept any new frame data until it has finished processing the current birds-eye frame.

```
open_system([modelname '/HDLLaneDetector'],'force');
```

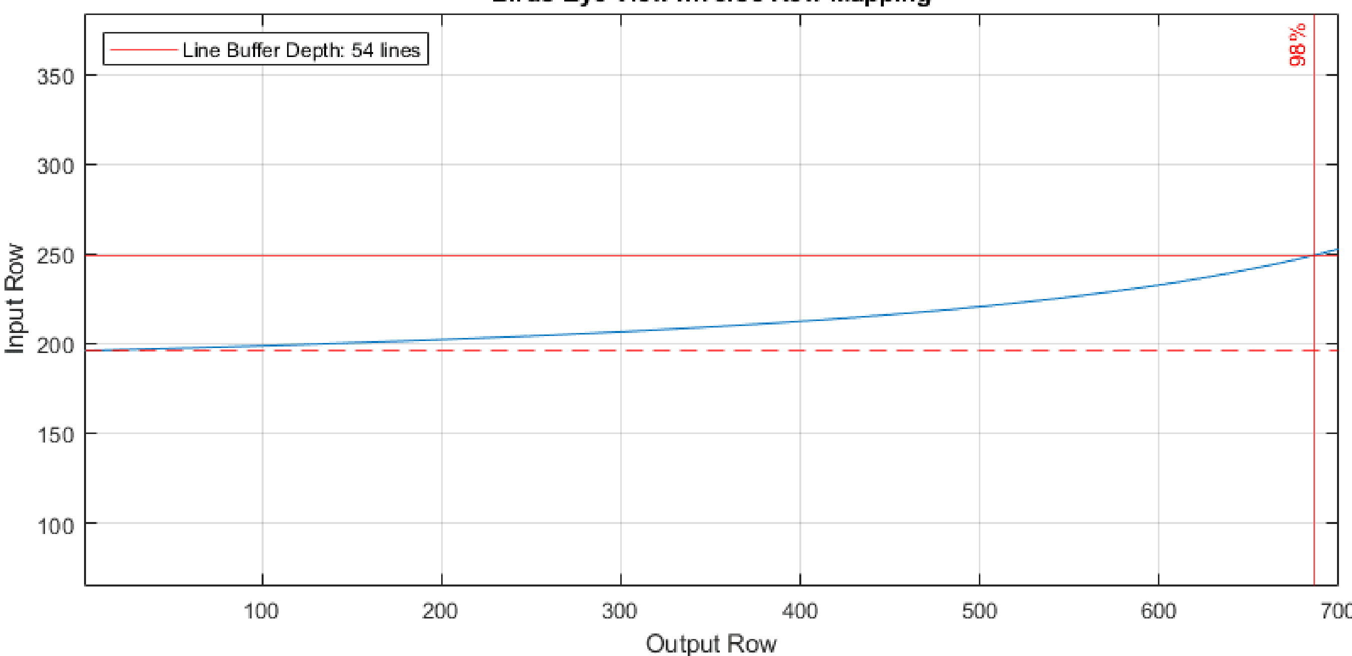
Line Buffering and Address Computation

A full sized projective transformation from input to output would result in a [900x640] output image. This requires that the full [480x640] input image is stored in memory, while the source pixel location is calculated using the source location and homography matrix. Ideally on-chip memory should be used for this purpose, removing the requirement for an off-chip frame buffer.

You can determine the number of lines to buffer on-chip by performing inverse row mapping using the homography matrix. The following script calculates the homography matrix from the point mapping, using it to an inverse transform to map the source frame rows to birds-eye view rows.

```
% Source & Birds-Eye Frame Parameters
%   AVL:  Active Video Lines, APPL: Active Pixels Per Line
sAVL  = 480;
sAPPL = 640;
% Birds-Eye Frame
bAVL  = 700;
bAPPL = 640;

% Determine Homography Matrix
%   Point Mapping [NW; NE; SW; SE]
sourcePoints   = [218,196; 421,196; -629,405; 1276,405];
birdsEyePoints = [001,001; 640,001;  001,900;  640,900];
%   Estimate Transform
tf = estgeotform2d(sourcePoints,birdsEyePoints,'projective');
%   Homography Matrix
h = tf.T;

% Visualize Birds-Eye ROI on Source Frame
vidObj   = VideoReader('visionhdl_caltech.avi');
vidFrame = readFrame(vidObj);
vidFrameAnnotated = insertShape(vidFrame,'Polygon',[sourcePoints(1,:) ...
    sourcePoints(2,:) sourcePoints(4,:) sourcePoints(3,:)],           ...
    'LineWidth',5,'Color','red');
vidFrameAnnotated = insertShape(vidFrameAnnotated,'FilledPolygon',    ...
    [sourcePoints(1,:) sourcePoints(2,:) sourcePoints(4,:)            ...
    sourcePoints(3,:)],'LineWidth',5,'Color','red','Opacity',0.2);
figure(1);
subplot(2,1,1);
imshow(vidFrameAnnotated)
title('Source Video Frame');

% Determine Required Birds-Eye Line Buffer Depth
%   Inverse Row Mapping at Frame Centre
x = round(sourcePoints(2,1)-((sourcePoints(2,1)-sourcePoints(1,1))/2));
Y = zeros(1,bAVL);
for ii = 1:1:bAVL
    [~,Y(ii)] = transformPointsInverse(tf,x,ii);
end
numRequiredRows = ceil(Y(0.98*bAVL) - Y(1));

% Visualize Inverse Row Mapping
subplot(2,1,2);
plot(Y,'HandleVisibility','off');   % Inverse Row Mapping
xline(0.98*bAVL,'r','98%','LabelHorizontalAlignment','left',          ...
    'HandleVisibility','off');      % Line Buffer Depth
yline(Y(1),'r--','HandleVisibility','off')
yline(Y(0.98*bAVL),'r')
```

```
title('Birds-Eye View Inverse Row Mapping');
xlabel('Output Row');
ylabel('Input Row');
legend(['Line Buffer Depth: ',num2str(numRequiredRows),' lines'],     ...
    'Location','northwest');
axis equal;
grid on;
```

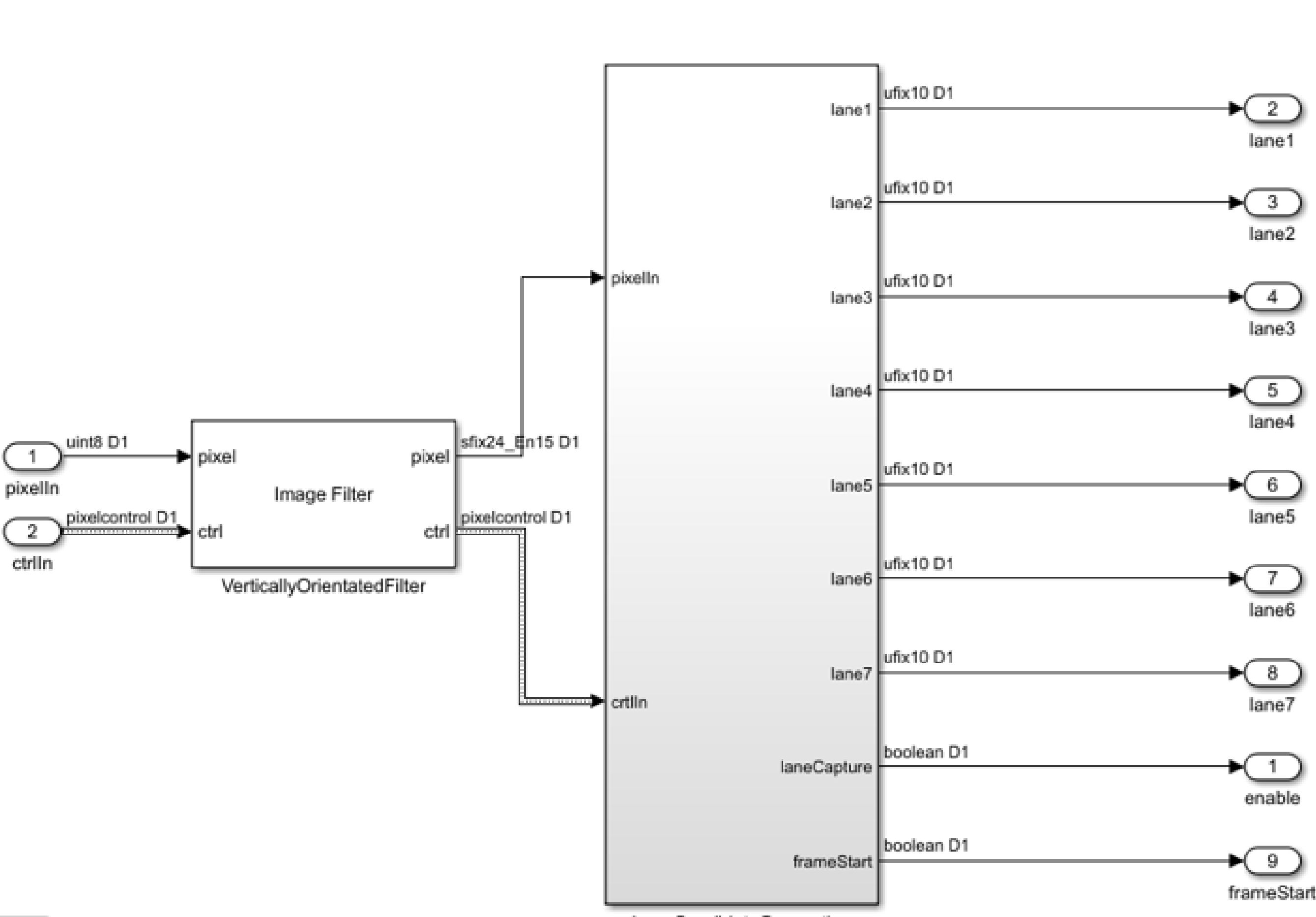


The plot shows the mapping of input line to output line revealing that in order to generate the first 700 lines of the top down birds eye output image, around 50 lines of the input image are required. This is an acceptable number of lines to store using on-chip memory.

Lane Detection

With the birds-eye view image obtained, the actual lane detection can be performed. There are many techniques which can be considered for this purpose. To achieve an implementation which is robust, works well on streaming image data and which can be implemented in FPGA/ASIC hardware at reasonable resource cost, this example uses the approach described in [1]. This algorithm performs a full image convolution with a vertically oriented first order Gaussian derivative filter kernel, followed by sub-region processing.

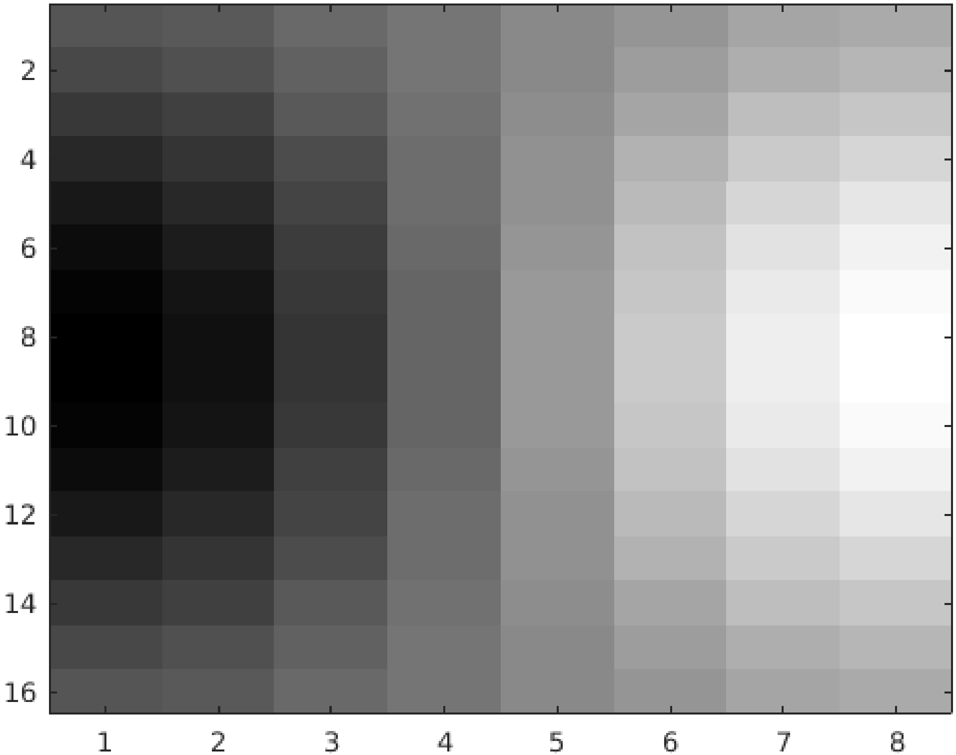
```
open_system([modelname '/HDLLaneDetector/LaneDetection'],'force');
```



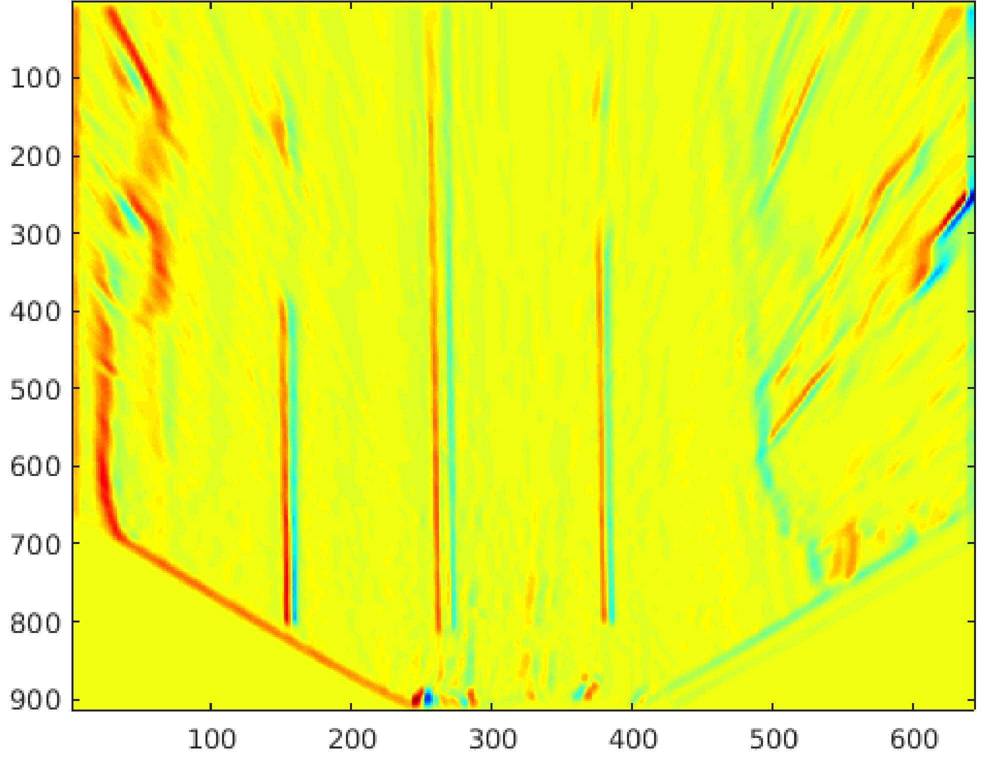
Vertically Oriented Filter Convolution

Immediately following the birds-eye mapping of the input image, the output is convolved with a filter designed to locate strips of high intensity pixels on a dark background. The width of the kernel is 8

pixels, which relates to the width of the lines that appear in the birds-eye image. The height is set to 16 which relates to the size of the dashed lane markings which appear in the image. As the birds-eye image is physically related to the height, pitch etc. of the camera, the width at which lanes appear in this image is intrinsically related to the physical measurement on the road. The width and height of the kernel may need to be updated when operating the lane detection system in different countries.



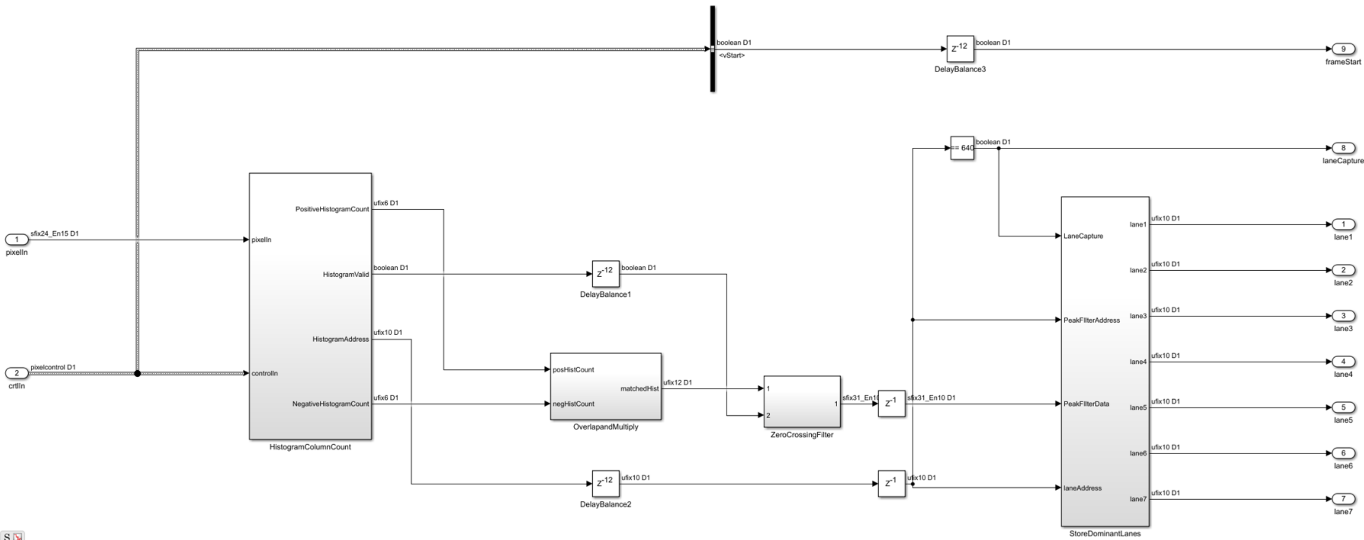
The output of the filter kernel is shown below, using jet colormap to highlight differences in intensity. Because the filter kernel is a general, vertically oriented Gaussian derivative, there is some response from many different regions. However, for the locations where a lane marking is present, there is a strong positive response located next to a strong negative response, which is consistent across columns. This characteristic of the filter output is used in the next stage of the detection algorithm to locate valid lane candidates.



Lane Candidate Generation

After convolution with the Gaussian derivative kernel, sub-region processing of the output is performed in order to find the coordinates where a lane marking is present. Each region consists of 18 lines, with a ping-pong memory scheme in place to ensure that data can be continuously streamed through the subsystem.

```
%Seeing as
open_system([modelname '/HDLLaneDetector/LaneDetection/LaneCandidateGeneration'],'force');
```



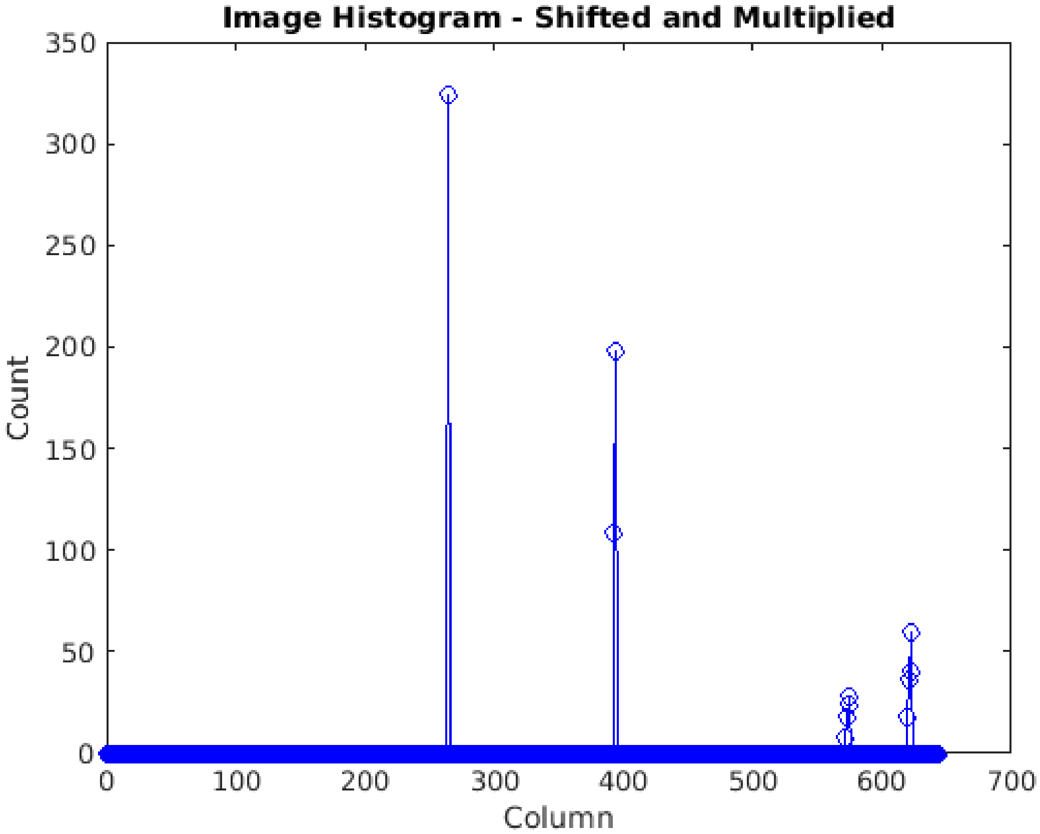
Histogram Column Count

Firstly, HistogramColumnCount counts the number of thresholded pixels in each column over the 18 line region. A high column count indicates that a lane is likely present in the region. This count is performed for both the positive and the negative thresholded images. The positive histogram counts are offset to account for the kernel width. Lane candidates occur where the positive count and negative counts are both high. This exploits the previously noted property of the convolution output where positive tracks appear next to negative tracks.

Internally, the column counting histogram generates the control signalling that selects an 18 line region, computes the column histogram, and outputs the result when ready. A ping-pong buffering scheme is in place which allows one histogram to be reading while the next is writing.

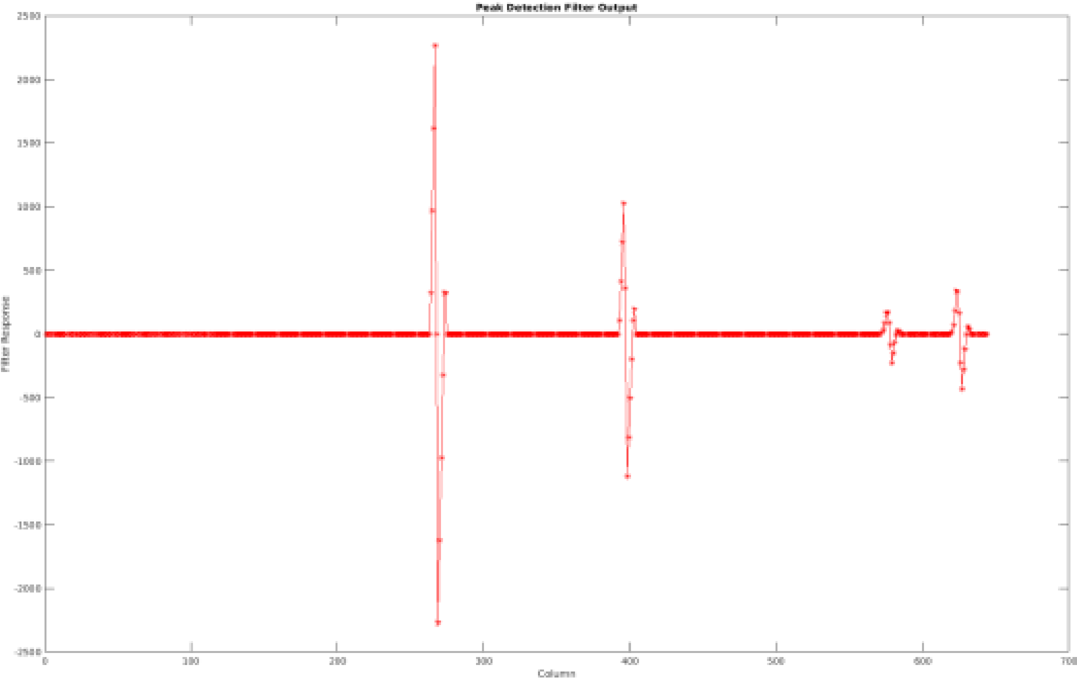
Overlap and Multiply

As noted, when a lane is present in the birds-eye image, the convolution result will produce strips of high-intensity positive output located next to strips of high-intensity negative output. The positive and negative column count histograms locate such regions. In order to amplify these locations, the positive count output is delayed by 8 clock cycles (an intrinsic parameter related to the kernel width), and the positive and negative counts are multiplied together. This amplifies columns where the positive and negative counts are in agreement, and minimizes regions where there is disagreement between the positive and negative counts. The design is pipelined in order to ensure high throughput operation.



Zero Crossing Filter

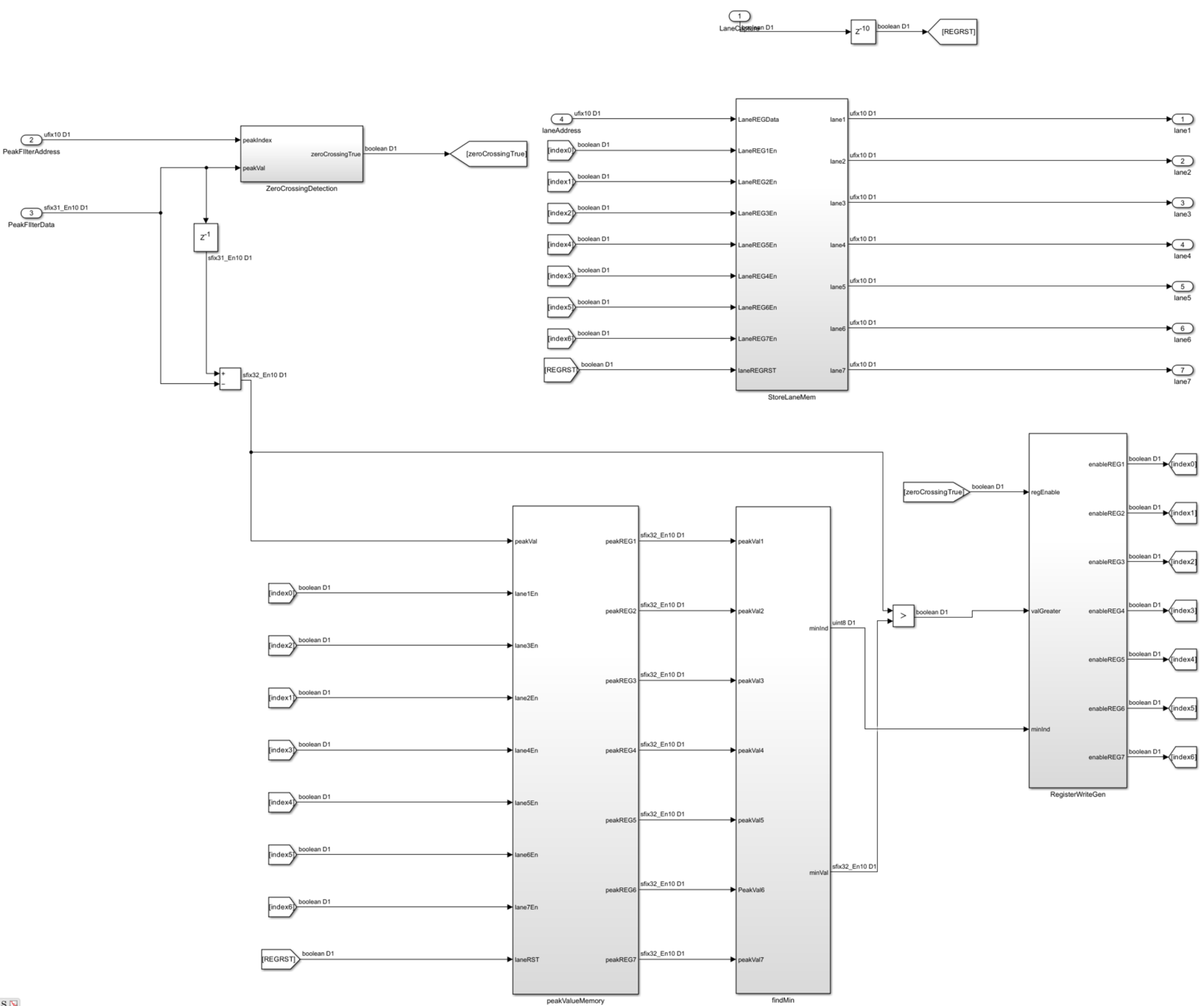
At the output of the Overlap and Multiply subsystem, peaks appear where there are lane markings present. A peak detection algorithm determines the columns where lane markings are present. Because the SNR is relatively high in the data, this example uses a simple FIR filtering operation followed by zero crossing detection. The Zero Crossing Filter is implemented using the Discrete FIR Filter block from DSP System Toolbox™. It is pipelined for high-throughput operation.



Store Dominant Lanes

The zero crossing filter output is then passed into the Store Dominant Lanes subsystem. This subsystem has a maximum memory of 7 entries, and is reset every time a new batch of 18 lines is reached. Therefore, for each sub-region 7 potential lane candidates are generated. In this subsystem, the Zero Crossing Filter output is streamed through, and examined for potential zero crossings. If a zero crossing does occur, then the difference between the address immediately prior to zero crossing and the address after zero crossing is taken in order to get a measurement of the size of the peak. The subsystem stores the zero crossing locations with the highest magnitude.

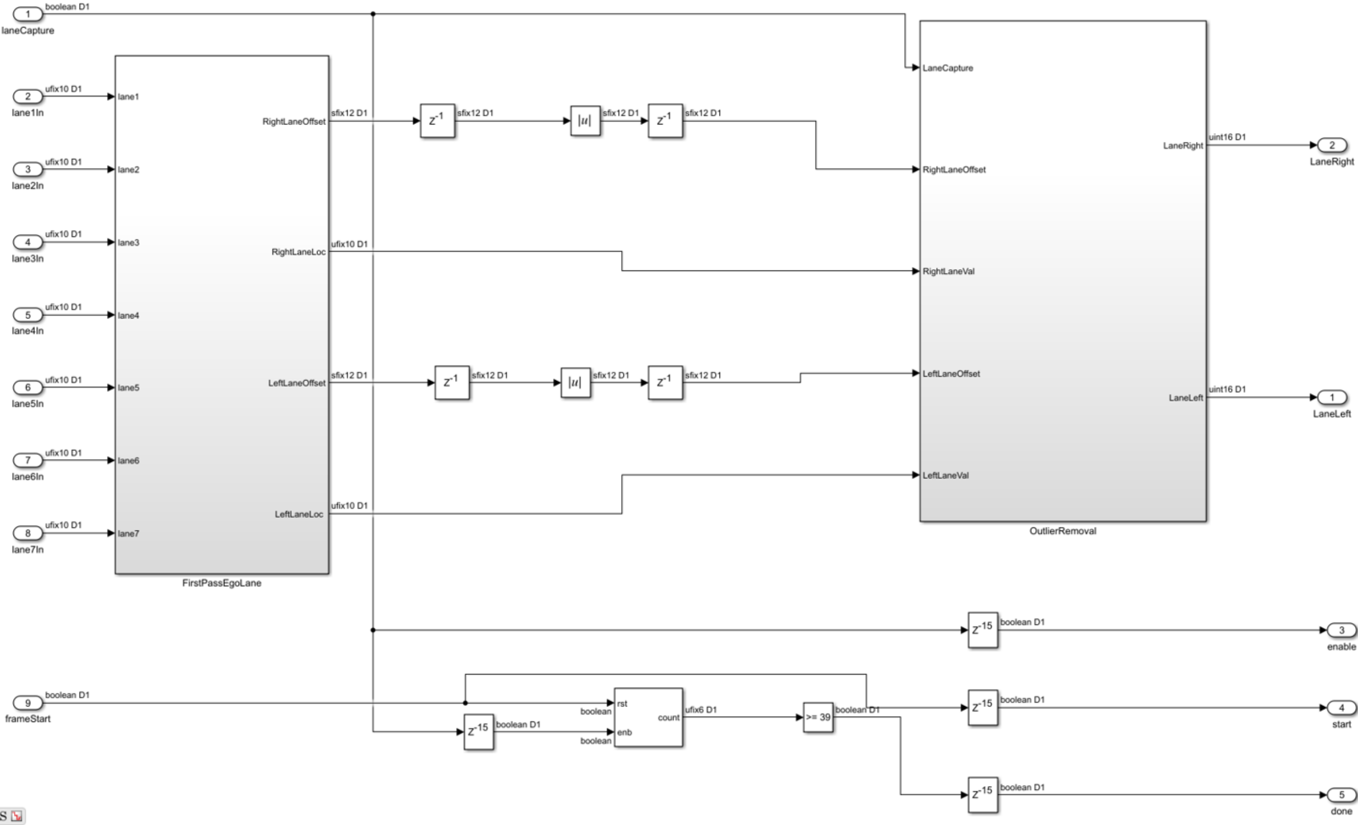
```
open_system([modelname '/HDLLaneDetector/LaneDetection/LaneCandidateGeneration/StoreDominantLanes
```



Compute Ego Lanes

The Lane Detection subsystem outputs the 7 most viable lane markings. In many applications, we are most interested in the lane markings that contain the lane in which the vehicle is driving. By computing the so called "Ego-Lanes" on the hardware side of the design, we can reduce the memory bandwidth between hardware and software, by sending 2 lanes rather than 7 to the processor. The Ego-Lane computation is split into two subsystems. The FirstPassEgoLane subsystem assumes that the centre column of the image corresponds to the middle of the lane, when the vehicle is correctly operating within the lane boundaries. The lane candidates which are closest to the center are therefore assumed as the ego lanes. The Outlier Removal subsystem maintains an average width of the distance from lane markings to centre coordinate. Lane markers which are not within tolerance of the current width are rejected. Performing early rejection of lane markers gives better results when performing curve fitting later on in the design.

```
open_system([modelname '/HDLLaneDetector/ComputeEgoLanes'],'force');
```



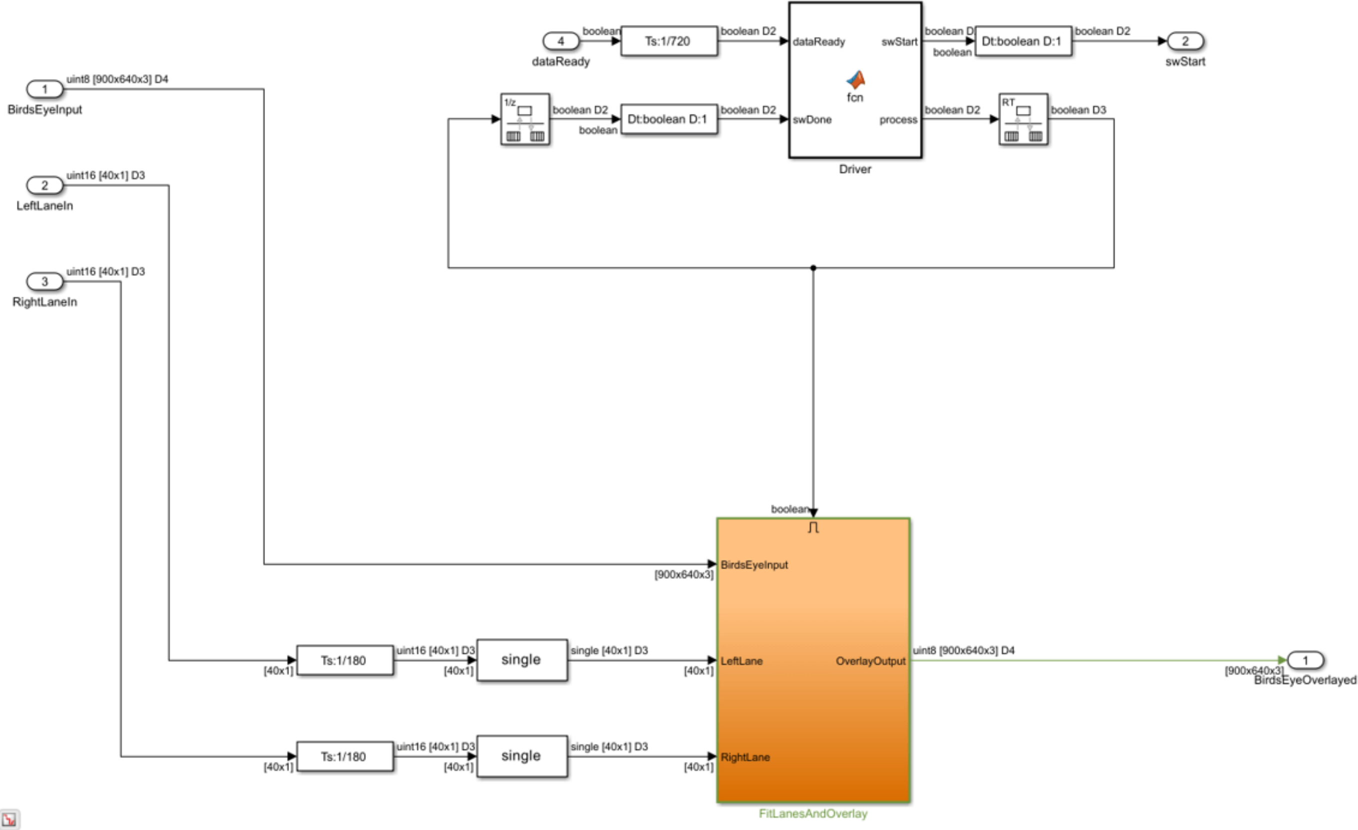
Control Interface

Finally, the computed ego lanes are sent to the CtrlInterface MATLAB function subsystem. This state machine uses the four control signal inputs - enable, hwStart, hwDone, and swStart to determine when to start buffering, accept new lane coordinate into the 40x1 buffer and finally indicate to the software that all 40 lane coordinates have been buffered and so the lane fitting and overlay can be performed. The dataReady signal ensures that software will not attempt lane fitting until all 40 coordinates have been buffered, while the swStart signal ensures that the current set of 40 coordinates will be held until lane fitting is completed.

Software Lane Fit and Overlay

The detected ego-lanes are then passed to the SW Lane Fit and Overlay subsystem, where robust curve fitting and overlay is performed. Recall that the birds-eye output is produced once every two frames or so rather than on every consecutive frame. The curve fitting and overlay is therefore placed in an enabled subsystem, which is only enabled when new ego lanes are produced.

```
open_system([modelname '/SWLaneFitandOverlay'],'force');
```



Driver

The Driver MATLAB Function subsystem controls the synchronization between hardware and software. Initially it is in a polling state, where it samples the dataReady input at regular intervals per frame to determine when hardware has buffered a full [40x1] vector of lane coordinates. Once this occurs, it transitions into software processing state where swStart and process outputs are held high. The Driver remains in the software processing state until swDone input is high. Seeing as the process output loops back to swDone input with a rate transition block in between, there is effectively a constant time budget specified for the FitLanesandOverlay subsystem to perform the fitting and overlay. When swDone is high, the Driver will transition into a synchronization state, where swStart is held low to indicate to hardware that processing is complete. The synchronization between software and hardware is such that hardware will hold the [40x1] vector of lane coordinates until the swStart signal transitions back to low. When this occurs, dataReady output of hardware will then transition back to low.

Fit Lanes and Overlay

The Fit Lanes and Overlay subsystem is enabled by the Driver. It performs the necessary arithmetic required in order to fit a polynomial onto the lane coordinate data received at input, and then draws the fitted lane and lane coordinates onto the Birds-Eye image.

Fit Lanes

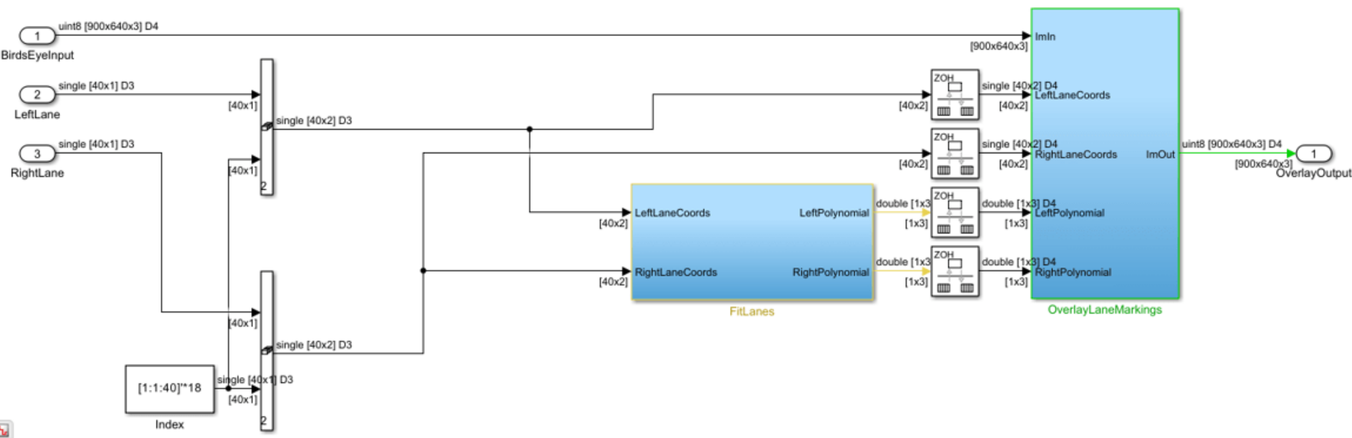
The Fit Lanes subsystem runs a RANSAC based line-fitting routine on the generated lane candidates. RANSAC is an iterative algorithm which builds up a table of inliers based on a distance measure

between the proposed curve, and the input data. At the output of this subsystem, there is a [3x1] vector which specifies the polynomial coefficients found by the RANSAC routine.

Overlay Lane Markings

The Overlay Lane Markings subsystem performs image visualization operations. It overlays the ego lanes and curves found by the lane-fitting routine.

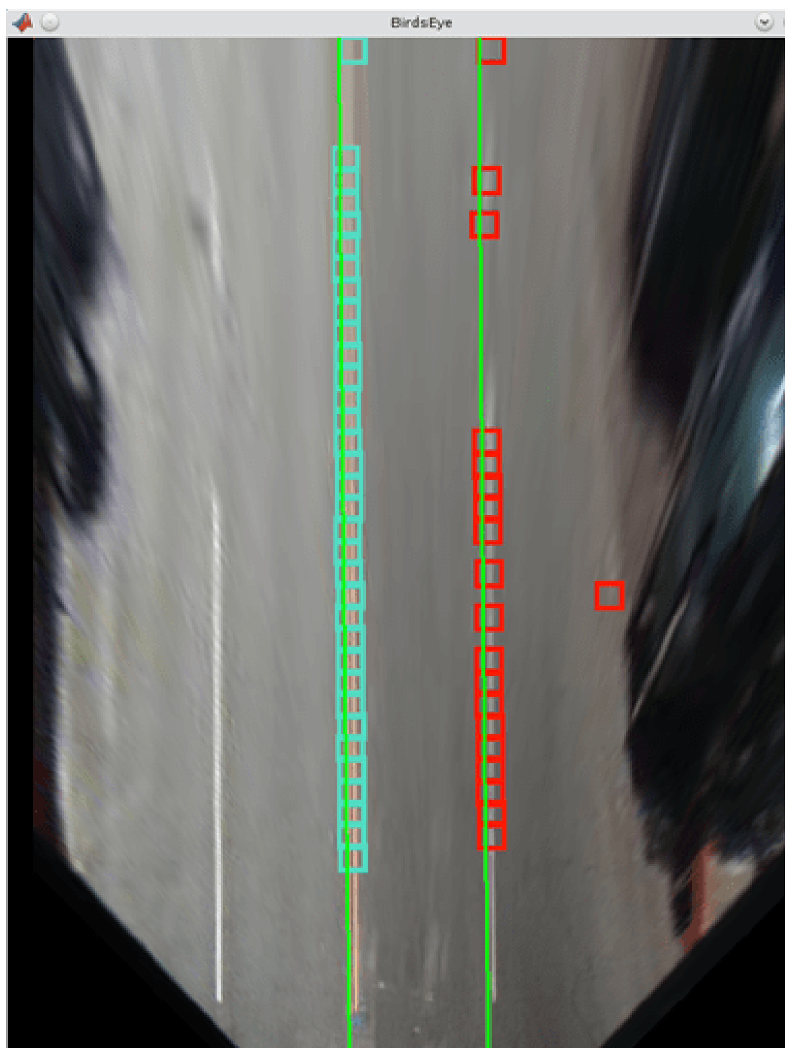
```
open_system([modelname '/SWLaneFitandOverlay/FitLanesAndOverlay'],'force');
```

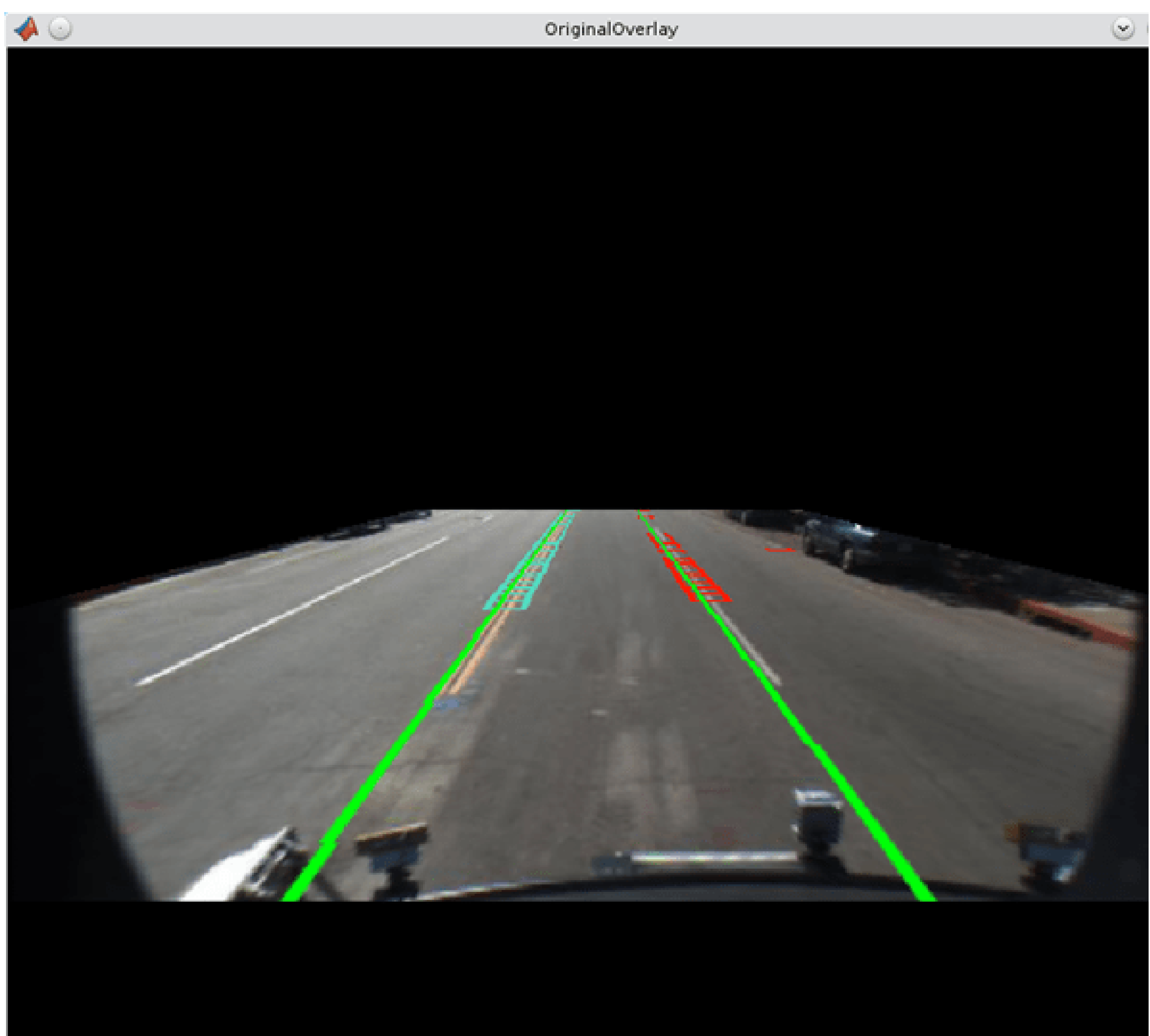


Results of the Simulation

The model includes two video displays shown at the output of the simulation results. The **BirdsEye** display shows the output in the warped perspective after lane candidates have been overlaid, polynomial fitting has been performed and the resulting polynomial overlaid onto the image. The **OriginalOverlay** display shows the **BirdsEye** output warped back into the original perspective.

Due to the large frame sizes used in this model, simulation can take a relatively long time to complete. If you have an HDL Verifier™ license, you can accelerate simulation speed by directly running the HDL Lane Detector subsystem in hardware using FPGA in the Loop.


BirdsEye



HDL Code Generation

To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command.

```
makehdl('LaneDetectionHDL/HDLLaneDetector')
```

To generate the test bench, use the following command. Note that test bench generation takes a long time due to the large data size. You may want to reduce the simulation time before generating the test bench.

```
makehdltb('LaneDetectionHDL/HDLLaneDetector')
```

For faster test bench simulation, you can generate a SystemVerilog DPIC test bench using the following command.

```
makehdltb('LaneDetectionHDL/HDLLaneDetector','GenerateSVDPITestBench','ModelSim')
```

Conclusion

This example has provided insight into the challenges of designing ADAS systems in general, with particular emphasis paid to the acceleration of critical parts of the design in hardware.

References

[1] R. K. Satzoda and Mohan M. Trivedi, "Vision based Lane Analysis: Exploration of Issues and Approaches for Embedded Realization", 2013 IEEE Conference on Computer Vision and Pattern Recognition.

[2] Video from Caltech Lanes Dataset - Mohamed Aly, "Real time Detection of Lane Markers in Urban Streets", 2008 IEEE Intelligent Vehicles Symposium - used with permission.

HDL QPSK Transmitter and Receiver

This example shows how to implement a QPSK transmitter and receiver in Simulink® that is optimized for HDL code generation and hardware implementation.

The model shown in this example modulates data based on quadrature phase shift keying (QPSK). The goal of this example is to model an HDL QPSK communication system that can transmit and recover information for a real-time system. The receiver implements symbol timing synchronization and carrier frequency and phase synchronization, which are essential in a single-carrier communication system.

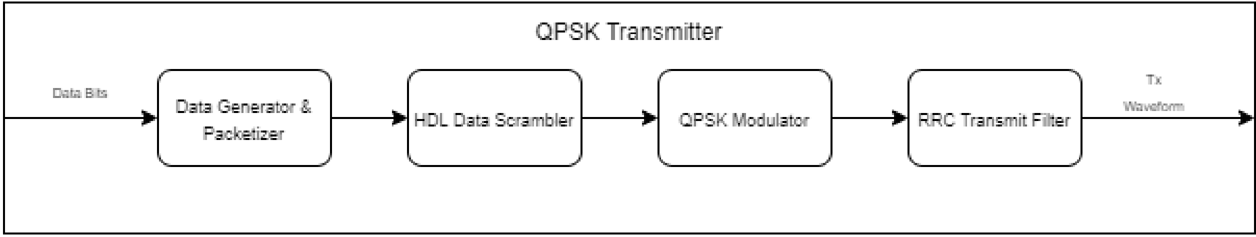
System Specifications

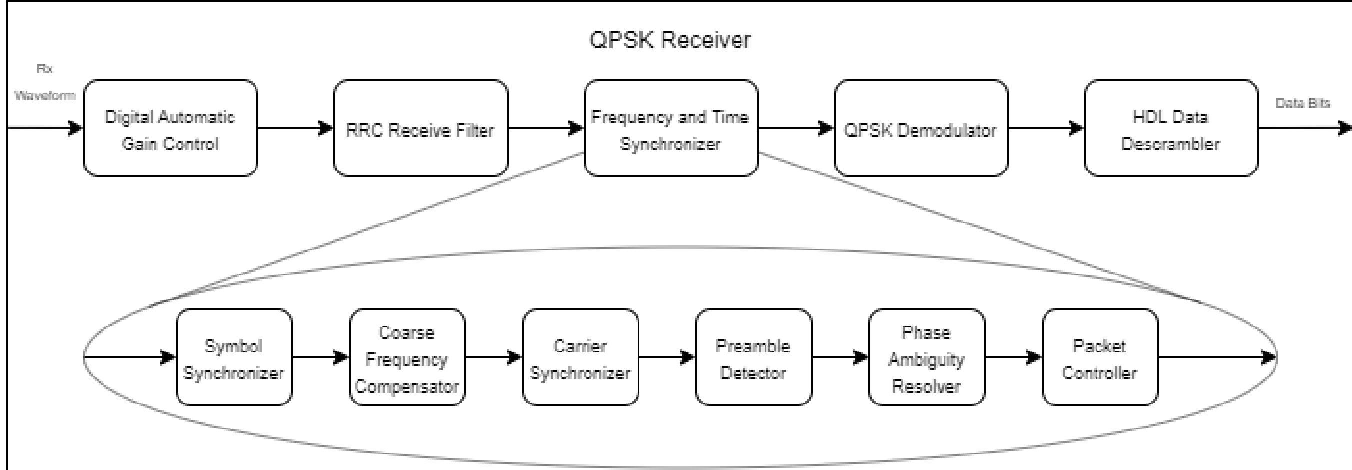
This section explains the specifications of the transmitter and receiver used in this example. The transmitter uses a packet-based frame format. To generate a preamble sequence, each bit of the 13-bit Barker sequence is repeated twice such that the same bit is modulated in the in-phase and quadrature-phase by the QPSK Modulator. The preamble sequence is followed by 2240 bits of payload data. The transmitter runs using a root raised cosine (RRC) pulse-shaping filter with a roll-off factor of 0.5, resulting in a bandwidth of 1.5 times the symbol rate. It uses four samples per symbol, resulting in a sample rate of four times the symbol rate. The RRC impulse response spans over four adjacent symbols. The bit rate is twice the symbol rate. The effective average bit rate is the bit rate times the frame efficiency. The frame efficiency is (2240/(2240+26)) , which equals to 0.9885.

The default symbol rate is set to 1.92 Mbaud, which results in a bandwidth of 1.5 times 1.92e6, which equals to 2.88 MHz, and a sample rate of 4 times 1.92e6, which equals to 7.68 Msps, bit rate of 2 times 1.92e6, which equals to 3.84 Mbps. The effective average bit rate supported by this system is 0.9885 times 3.84e6, which equals to 3.7959 Mbps. The entire system runs at a single maximum clock rate of 7.68 MHz. To achieve lower sample rates, the valid signal is handled accordingly. These specifications vary with the symbol rate.

Model Architecture

This section explains the high-level architecture of the QPSK transmitter and receiver as shown in the block diagram. The QPSK transmitter samples the input at a bit rate of twice the symbol rate. The Data Generator & Packetizer collects the data bits, generates the preamble bits, and forms the packet bits. The HDL Data Scrambler scrambles the data bits of each packet to increase bit transitions and avoid long running sequences of the same bit. The QPSK Modulator modulates the packet bits to generate QPSK symbols. The RRC Transmit Filter upsamples and pulse-shapes the QPSK symbols to generate the Tx Waveform at a sample rate of four times that of the symbol rate. The QPSK receiver samples the input at the transmission rate. The Digital AGC performs gain control to the desired amplitude level of the received waveform. The RRC Receive Filter performs matched filtering on the AGC output. The Frequency and Time Synchronizer performs synchronization operations and generates QPSK symbols for each packet. The QPSK Demodulator demodulates the QPSK symbols to generate packet bits. The HDL Data Descrambler descrambles the packet data bits that stream out of the receiver.





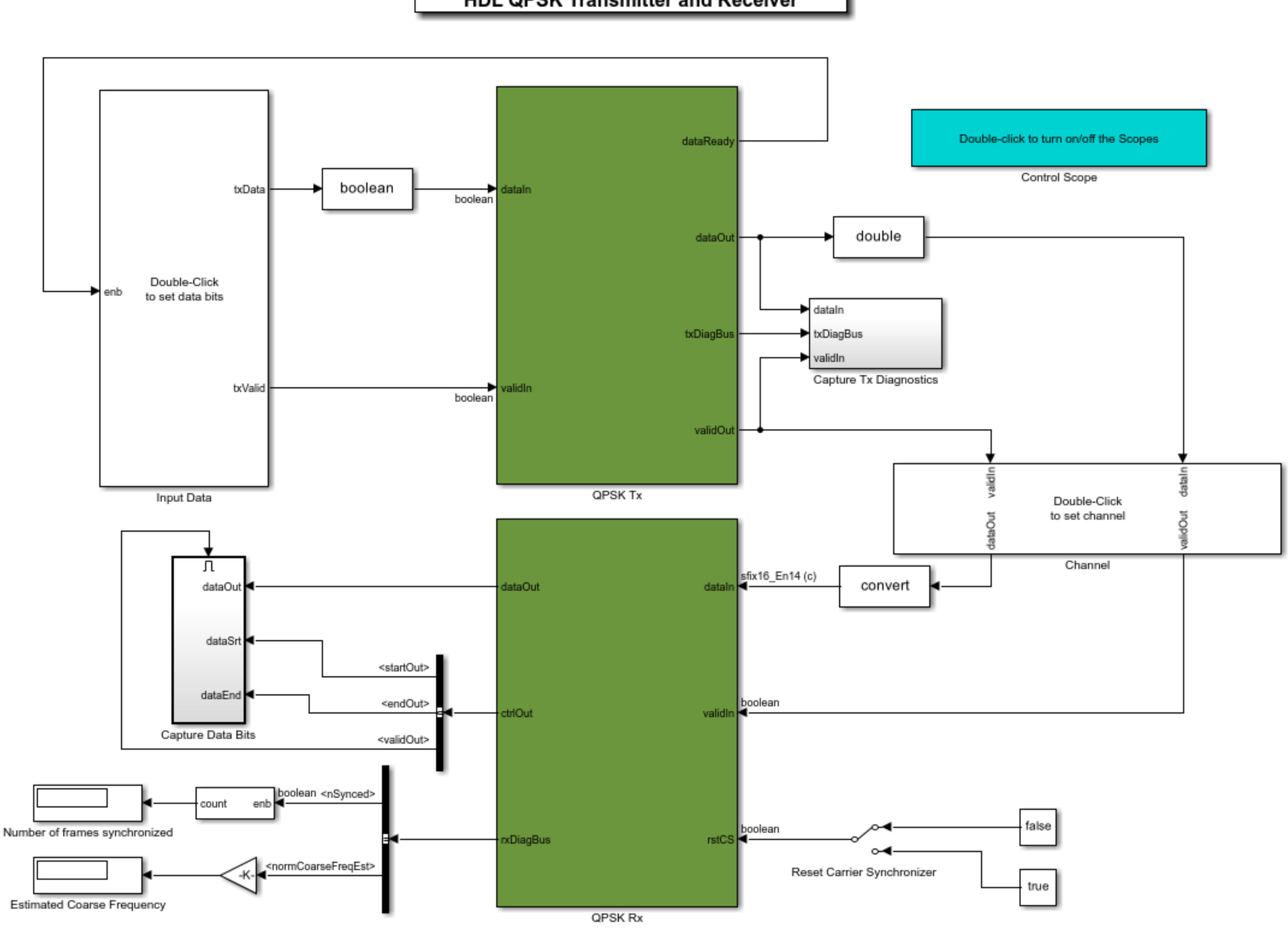
File Structure

The example contains the following Simulink model and MATLAB scripts.

- `commhdlQPSKTxRx` — Top-level Simulink model
- `commhdlQPSKTxRxParameters` — Generates parameters for QPSK Tx and QPSK Rx required for initialization
- `commhdlQPSKTxRxModelInit` — Initializes the `commhdlQPSKTxRx` model
- `generateHelloworldMsgBits` — Generates "Hello world *xxx* " message bits. *xxx* refers to values from 000 to 100.

System Interface

This figure shows the top-level model of the QPSK transmitter and receiver system.



Transmitter Inputs

- **dataIn** — Input data, specified as a Boolean scalar.
- **validIn** — Control signal to validate the **dataIn**, specified as a Boolean scalar.

Transmitter Outputs

- **dataOut** — Output transmitted waveform, returned as 16-bit complex data at a sample rate four times that of the symbol rate.
- **validOut** — Control signal to validate the **dataOut**, returned as a Boolean scalar.
- **txDiagBus** — Status signal with diagnostic outputs, returned as a Bus signal.
- **dataReady** — Signal to indicate a ready for the input signals, returned as a Boolean scalar.

Receiver Inputs

- **dataIn** — Input data, specified as a 16-bit complex data with sample rate as the transmitter output.

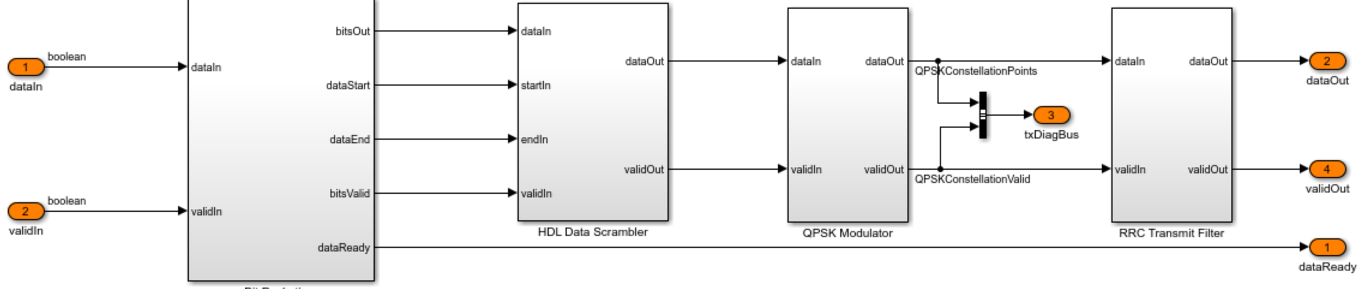
- **validIn** — Control signal to validate the **dataIn**, specified as a Boolean scalar.

Receiver Outputs

- **dataOut** — Decoded output data bits, returned as a Boolean scalar.
- **ctrlOut** — Bus signal with start, end, and valid signals, returned as a bus signal.
- **rxDiagBus** — Status signal with diagnostic outputs, returned as a bus signal.

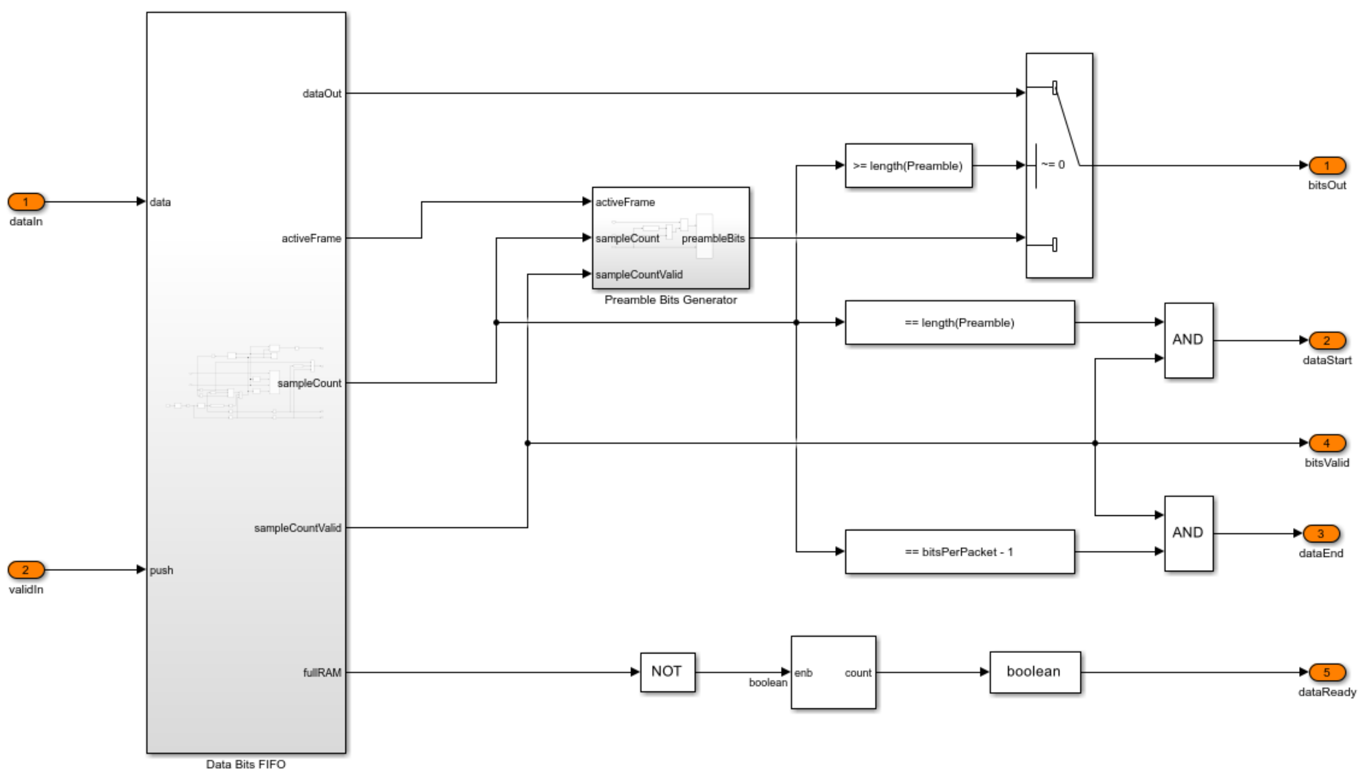
Transmitter Structure

This figure shows the top-level model of the `QPSK Tx` subsystem.

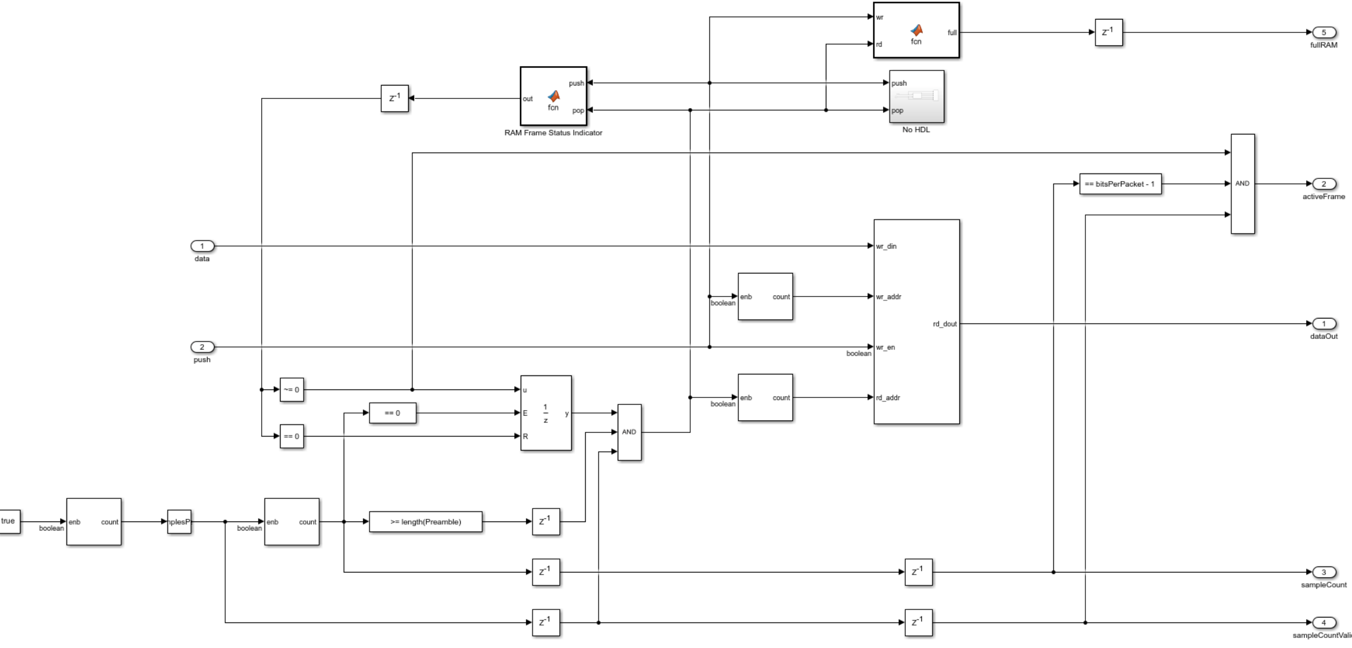


Bit Packetizer

The `Bit Packetizer` subsystem consists of a `Data Bits FIFO` subsystem and a `Preamble Bits Generator` subsystem. It stores input data and reads it out whenever required. It also generates the dataReady signal to indicate if the transmitter is ready to accept input data.



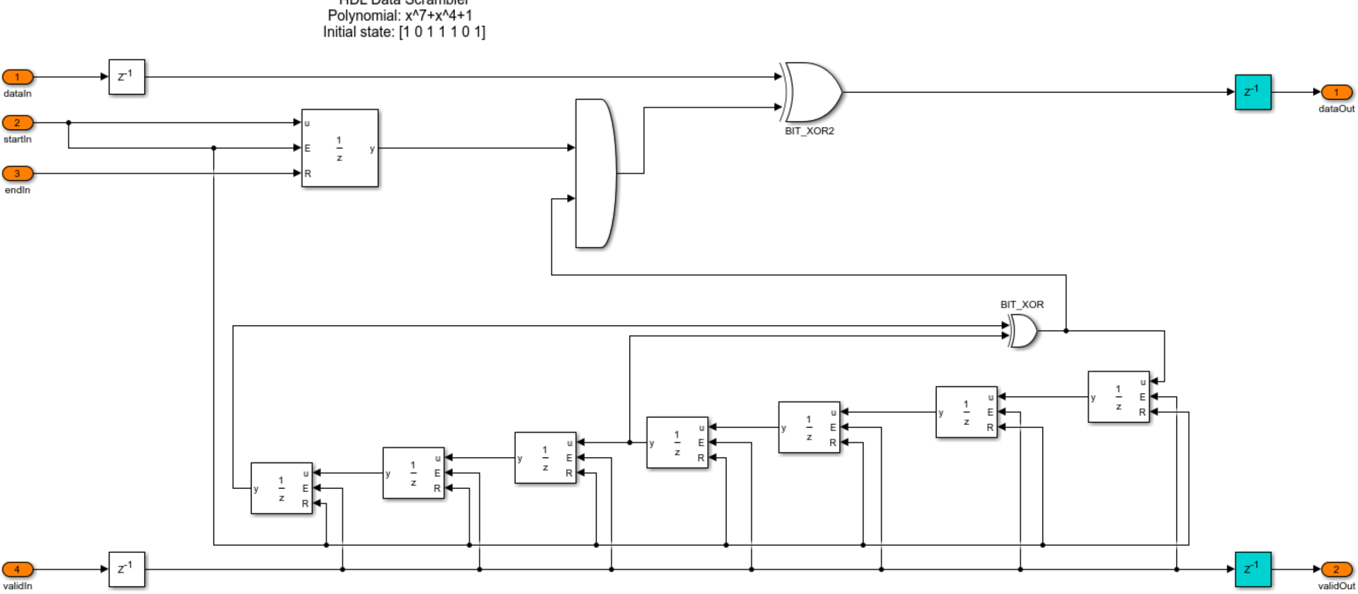
The Data Bits FIFO subsystem collects input data bits and stores them in a RAM. The RAM can fit two data packets to store the current data packet while reading out the previous packet. The RAM Frame Status Indicator function counts the number of packets currently stored in RAM. The subsystem reads data from the RAM only if at least one packet is available.



The `Preamble Bits Generator` subsystem gives out valid preamble sequence if there is at least one packet available in the RAM. Otherwise, it gives out random sequence.

HDL Data Scrambler

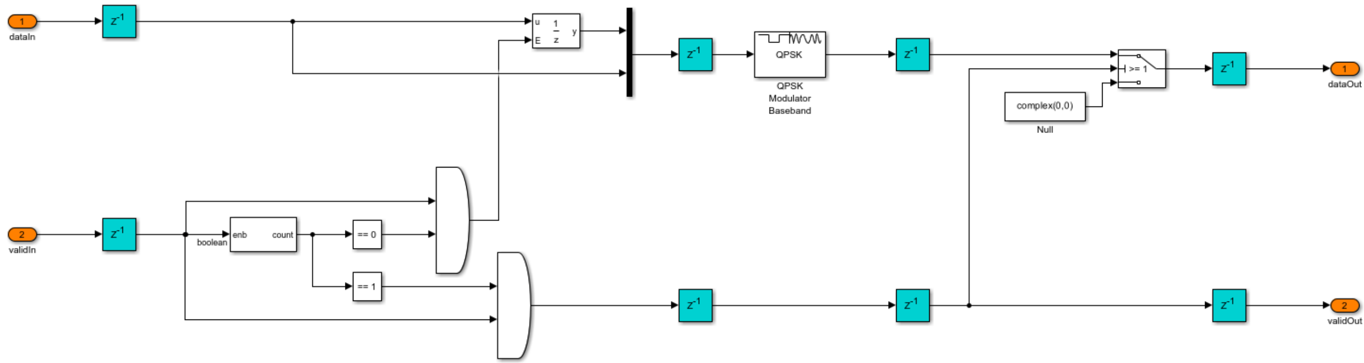
The `HDL Data Scrambler` subsystem scrambles the data bits in each packet by using the control signals generated by the `Bits Generator` subsystem.



QPSK Modulator

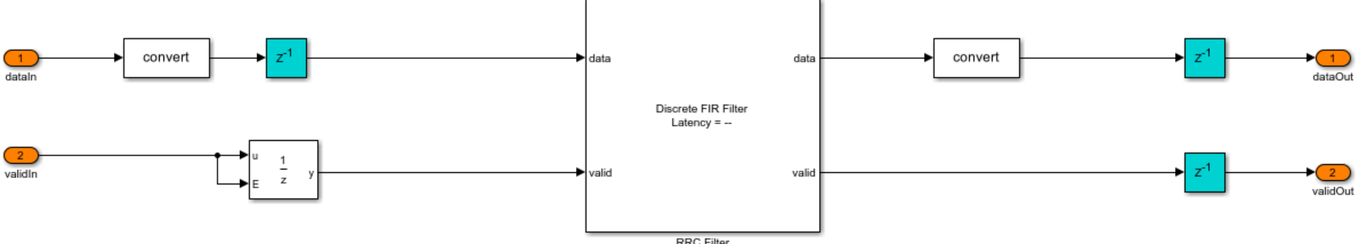
The `QPSK Modulator` subsystem uses the QPSK Modulator Baseband (Communications Toolbox) block to modulate the preamble and data bits to generate QPSK symbols. It uses a gray mapping as described in this table.

```
Bits         Mapping
____    _________________

 00     0.70711+0.70711i
 01     -0.70711+0.70711i
 11     -0.70711-0.70711i
 10     0.70711-0.70711i
```



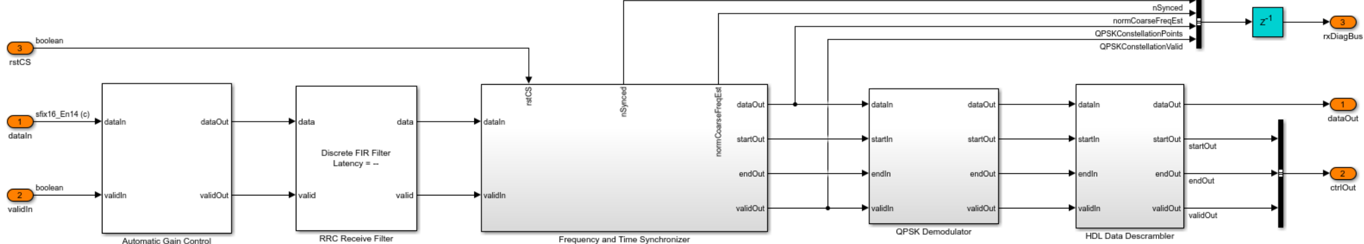
RRC Transmit Filter

The `RRC Transmit Filter` subsystem accepts input signals at a clock rate of 4 times the symbol rate that is a valid symbol followed by 3 zeros. This discrete valid signal is made continuous using the unit delay enabled block, which is equivalent to upsampling by 4. This upsampled signal is fed to the Discrete FIR Filter (DSP HDL Toolbox) block with an RRC impulse response to pulse-shape the transmitter waveform. The receive filter in the QPSK receiver forms a matched filter to this transmit filter.



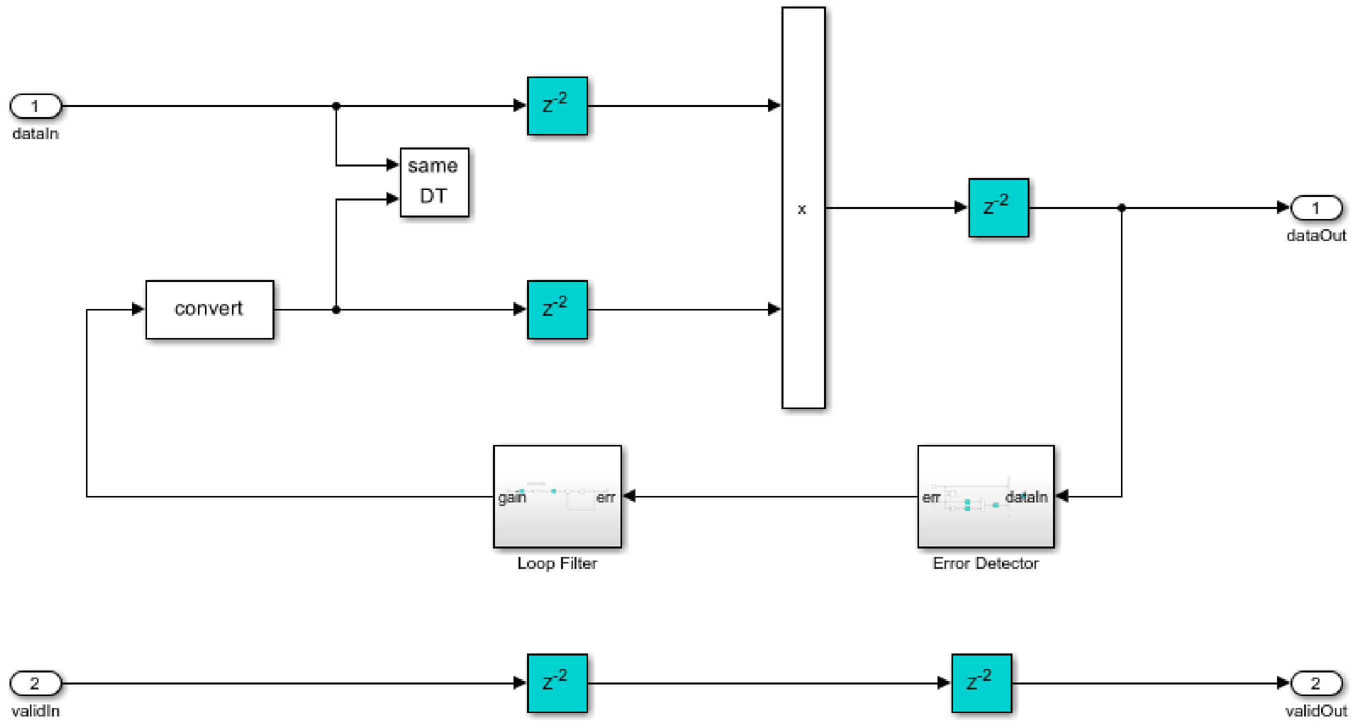
Receiver Structure

This figure shows the top-level overview of the `QPSK Rx` subsystem.



Automatic Gain Control

As the input signal amplitude affects the symbol and carrier synchronizer phase-locked loop (PLL) performance, the `Automatic Gain Control` subsystem is placed ahead of them. The magnitude squared output is compared with the AGC reference to generate an amplitude error. This error is multiplied with the loop gain and passed through an integrator to calculate the required gain. The resulted gain is multiplied with the AGC input to generate the AGC output. For more information, see Chapter 9.5 of [1].

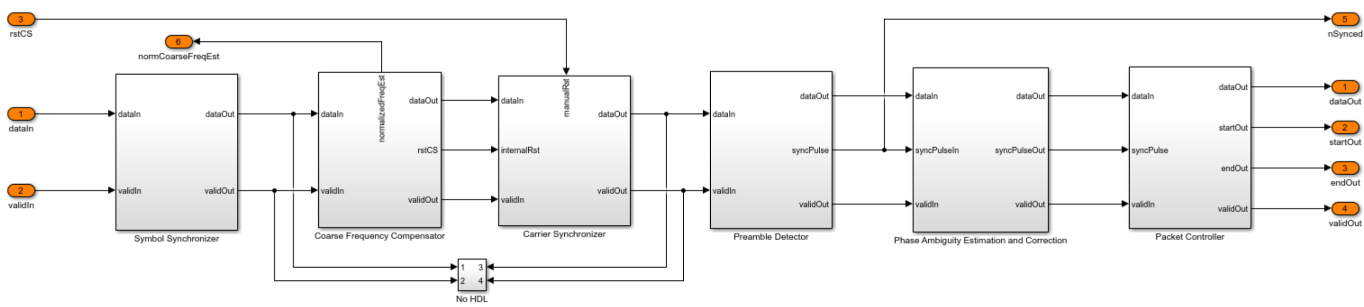


RRC Receive Filter

The RRC Receive Filter is a Discrete FIR Filter (DSP HDL Toolbox) block with matched filter coefficients of the filter used for pulse-shaping in the transmitter. The RRC matched filtering generates an RC pulse-shaped waveform, which has zero ISI characteristics at maximum eye opening in the eye diagram of the waveform. Also, the matched filtering process maximizes the signal to noise power ratio (SNR) of the filter output.

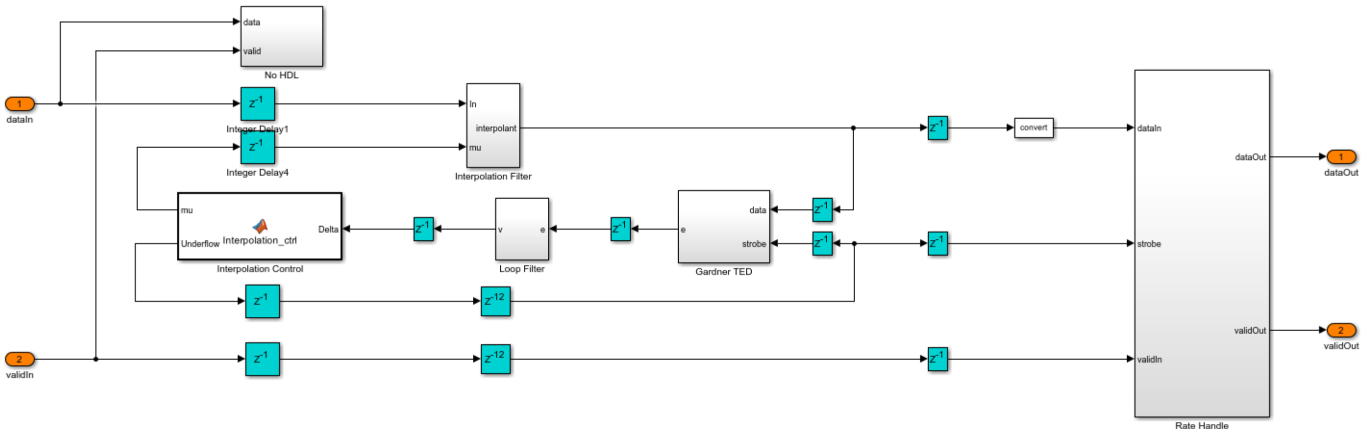
Frequency and Time Synchronizer

The `Frequency and Time Synchronizer` subsystem performs symbol synchronization, coarse frequency compensation, carrier synchronization, and preamble detection for packet synchronization. It also estimates and resolves the phase ambiguity that is left uncorrected in carrier synchronization.

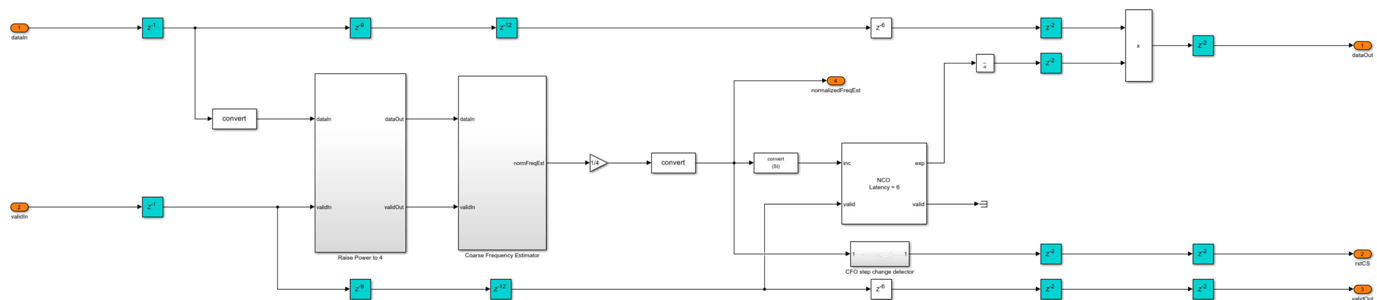


The `Symbol Synchronizer` subsystem is a PLL-based implementation. It generates samples at the optimum time instant (maximum eye opening instant) as described in Chapter 8.5 of [1]. The subsystem generates one output sample for every four input samples. The `Interpolation Filter`

subsystem implements a piecewise parabolic interpolator with a hardware resource efficient farrow structure as described in Chapter 8.4.2, and the farrow coefficients are tabulated in Table 8.4.1 (the free parameter α of the coefficients is taken as 0.5) of [1]. This filter introduces fractional delays in the input waveform. The `Gardner TED` subsystem implements a Gardner timing error detector. The timing error detector is described in Chapter 8.4.1 of [1]. The `Loop Filter` subsystem filters the timing error and the timing error is passed on to the `Interpolation Control` function block. This block implements a mod-1 decrementing counter to calculate fractional delays based on the loop filtered timing error as described in Chapter 8.4.3 of [1] to generate interpolants at optimum sampling instants. The `Rate Handle` subsystem selects samples if there is a strobe signal and stores them in a FIFO. These samples correspond to the maximum eye opening of the eye diagram before symbol synchronization.

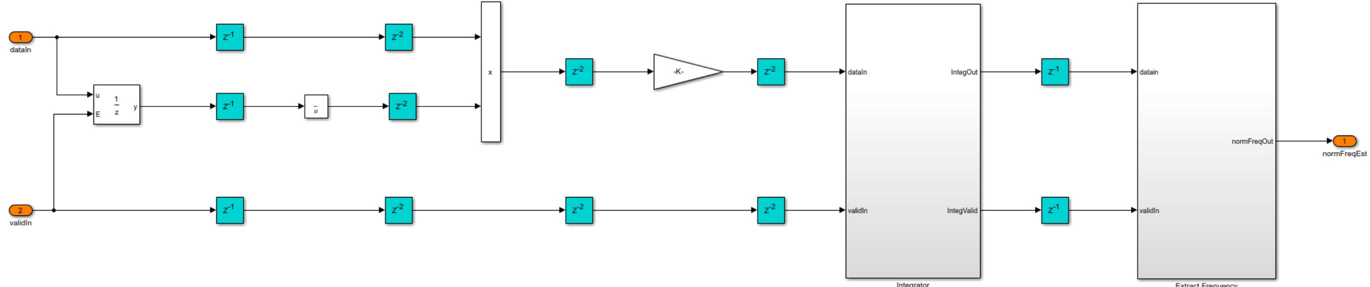


The `Coarse Frequency Compensator` subsystem raises the input sequence to a power of 4 in the `Raise Power to 4` subsystem. This eliminates the QPSK phase mapping dependency in the input sequence but reduces the estimation range by a factor of 4. This sequence streams into the `Coarse Frequency Estimator` subsystem. The estimate obtained from the `Coarse Frequency Estimator` subsystem is divided by 4 to remove the factor 4 due to raising to power 4 and get the normalized coarse frequency estimate. This estimate drives the NCO (DSP HDL Toolbox) block to generate complex exponential phase that is conjugated and multiplied with the input sequence to correct the frequency offset.

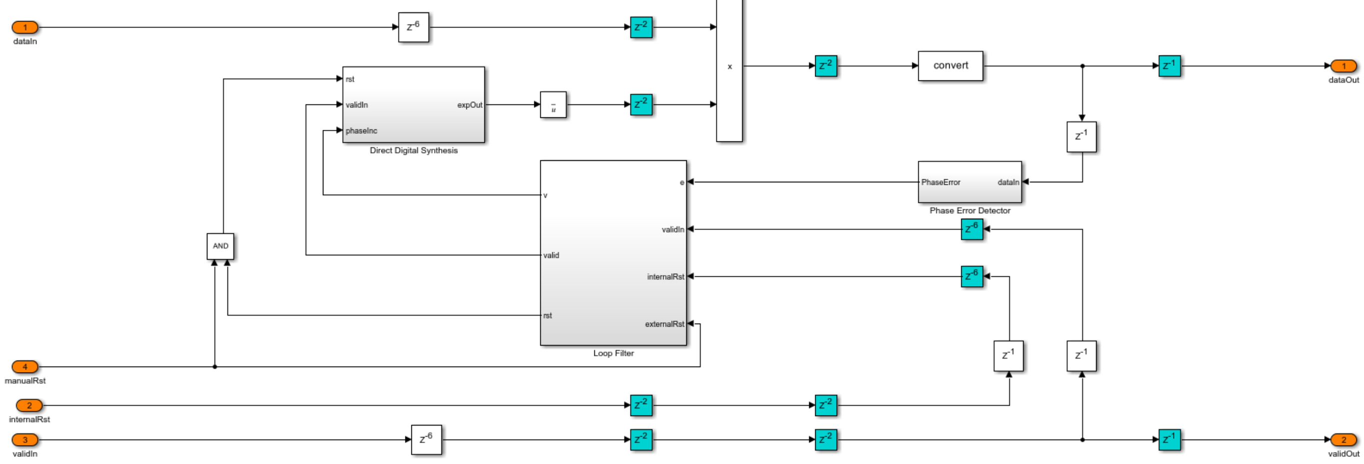


The `Coarse Frequency Estimator` subsystem differentially detects the input sequence and extracts the complex frequency offset estimate in the input. This estimate is averaged for 2^15 consecutive estimates in the `Integrator` subsystem to get the final complex estimate. The Complex to Magnitude-Angle (DSP HDL Toolbox) block extracts the frequency from the complex estimate in the `Extract Frequency` subsystem.

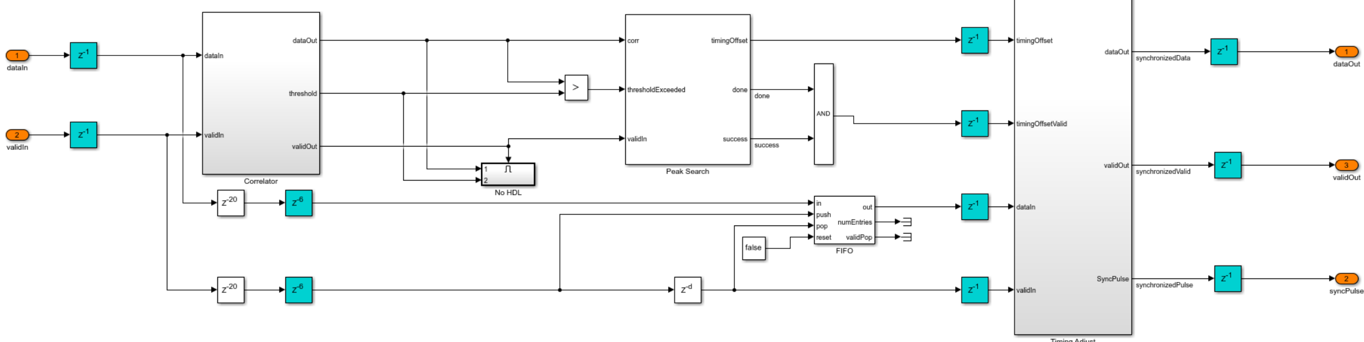
The frequency estimator estimates a normalized frequency (with respect to symbol rate) range of -0.125 to 0.125, which corresponds to a frequency offset range of -240 KHz to 240 KHz for a symbol rate of 1.92 Mbaud. The estimation accuracy is such that the residual frequency offset after coarse frequency offset correction is within the normalized frequency range of -0.0016 to 0.0016, which corresponds to a frequency offset range of -3 KHz to 3 KHz for a symbol rate of 1.92 Mbaud that the carrier synchronizer PLL converges.



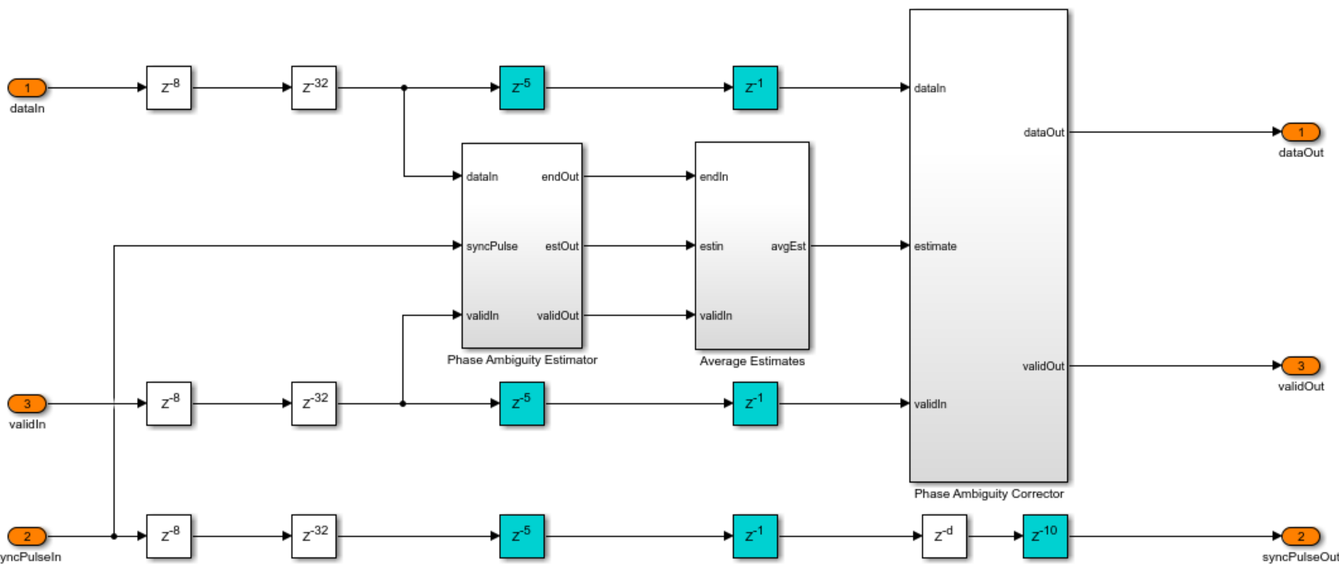
The `Carrier Synchronizer` subsystem is a TYPE II PLL with a sinusoidal phase error detector, which operates at a 45 degrees operating point. The phase error detector is described in Chapter 7.2.2, and the design equations are described in the Appendix C of [1]. A detailed analysis of TYPE II PLL with a zero operating point sinusoidal phase detector is described in Chapter 4 of [2]. The sign function of the phase detector in the real and imaginary parts converts all of the angles in the 4 quadrants into a first-quadrant angle (0 to 90 degrees), which creates an ambiguity of 90,180,270 degrees for second (90 to 180 degrees), third (-180 to -90 degrees) and fourth (-90 to 0 degrees) quadrant angles, respectively. The phase error is calculated as a deviation from the operating point (45 degrees) of the phase detector. The proportional plus integrator filter in the `Loop Filter` subsystem filters the phase error. The loop filter sets the normalized loop bandwidth (normalized by the sample rate) and the loop damping factor. The default normalized loop bandwidth is set to 0.005, and the default damping factor is set to 0.7071. The filtered error is given as a phase increment source to the `Direct Digital Synthesis` subsystem, which uses the NCO (DSP HDL Toolbox) block for complex exponential phase generation. The complex exponential phase is used to correct the frequency and phase of the input. A detailed analysis of direct digital synthesis is described in Chapter 9.2.2 of [1].



The `Preamble Detector` subsystem performs continuous correlation for the input with the Barker sequence. The correlation is implemented as convolution with the reversed Barker sequence as coefficients for the Discrete FIR Filter (DSP HDL Toolbox) block, and the magnitude of the correlated output is found using the Complex to Magnitude-Angle (DSP HDL Toolbox) block inside the `Correlator` subsystem. The magnitude of the correlation is compared with a threshold. The `Peak Search` subsystem begins searching for the maximum correlation peak that exceeded the threshold for every one frame duration and records the timing offset. The `Timing Adjust` subsystem synchronizes packet timing based on the timing offset to generate *syncPulse* signal, which indicates a packet synchronized sample to the subsequent subsystem.



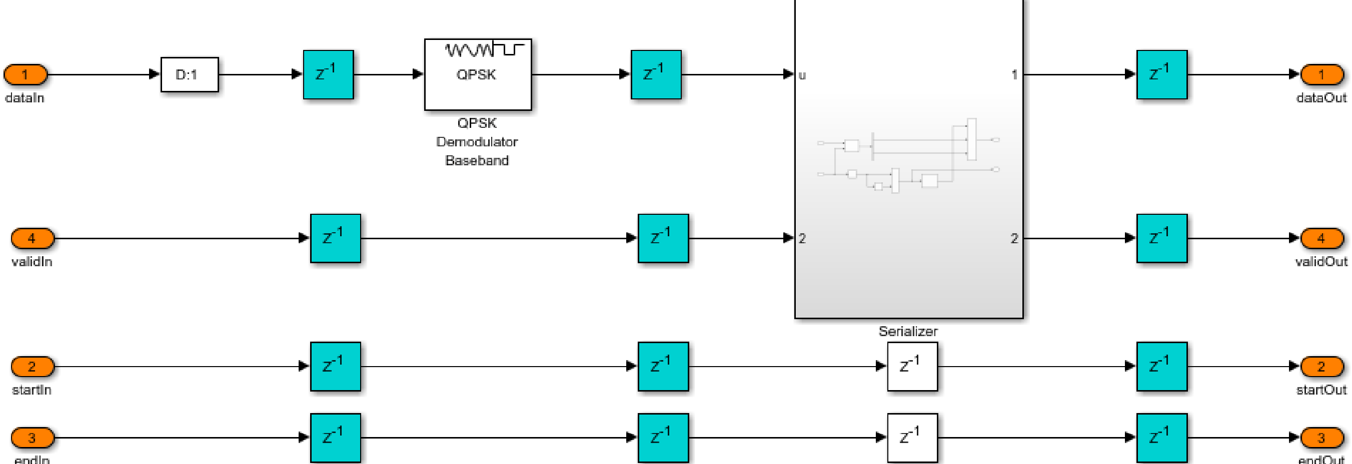
The `Phase Ambiguity Estimation and Correction` subsystem works based on the unique word method for phase ambiguity resolution described in Chapter 7.7.1 of [1]. This method uses the preamble sequence as the reference sequence. The reference sequence is conjugated and multiplied with the preamble sequence in the input, and the residual phase is extracted as the phase ambiguity estimate. This estimate is used to correct the ambiguity by rotating the constellation in the opposite direction of ambiguity.



The `Packet Controller` subsystem generates control signals for the packet boundaries.

QPSK Demodulator

The `QPSK Demodulator` subsystem uses the QPSK Demodulator Baseband (Communications Toolbox) block to demodulate the packet synchronized symbols and generate bits.



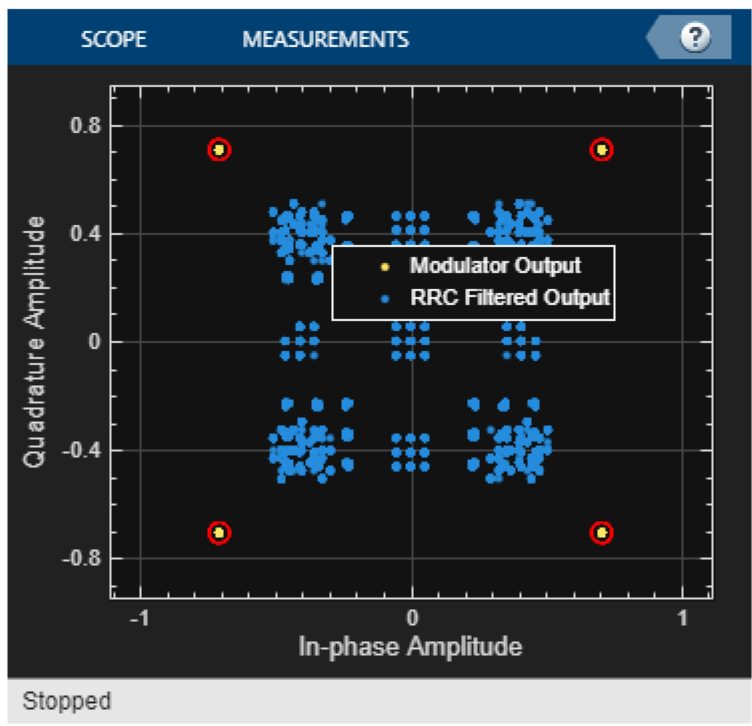
HDL Data Descrambler

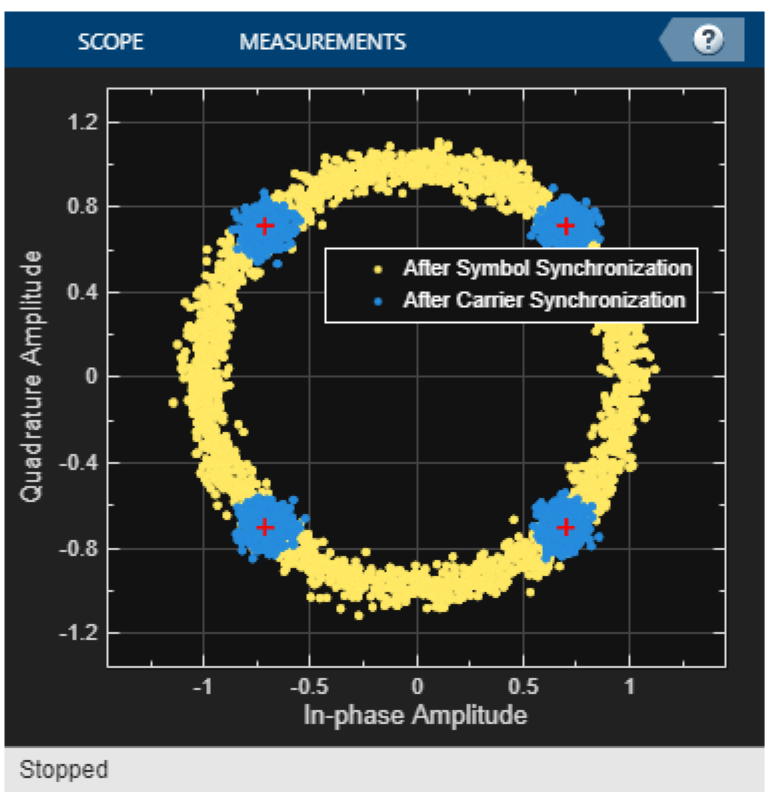
The `HDL Data Descrambler` subsystem descrambles the demodulated bits to generate the user bits. This subsystem is same as the scrambler used at the transmitter side.

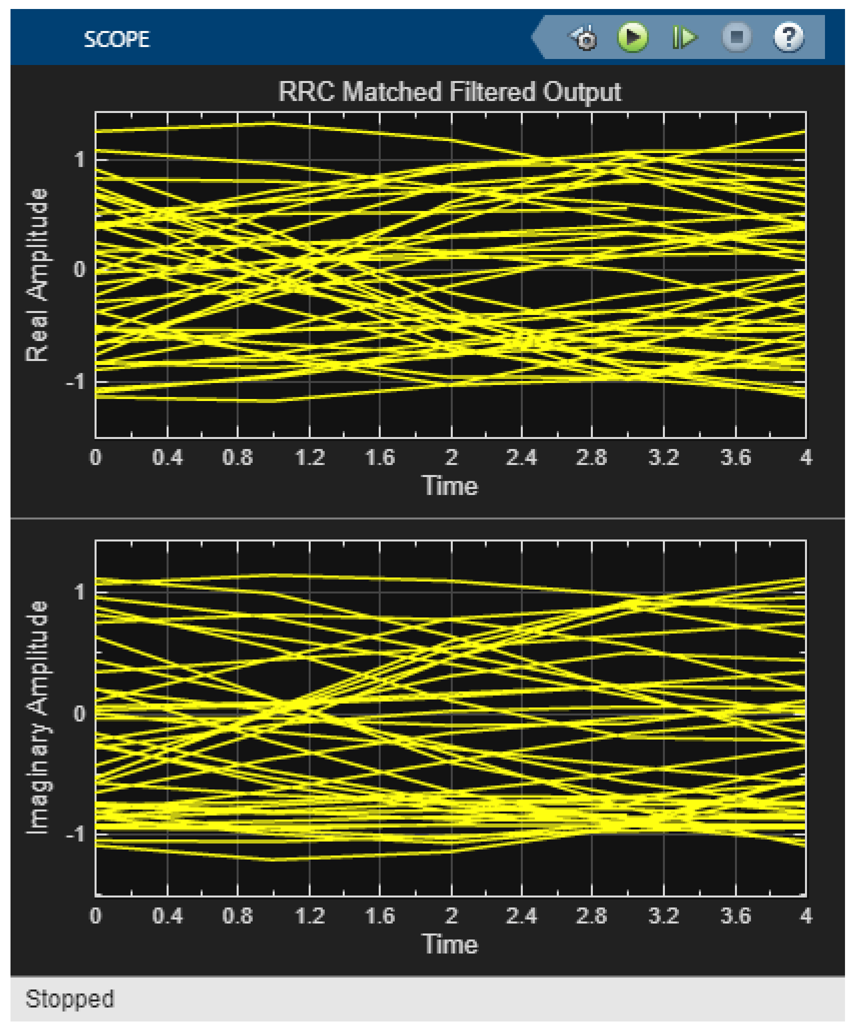
Run Model

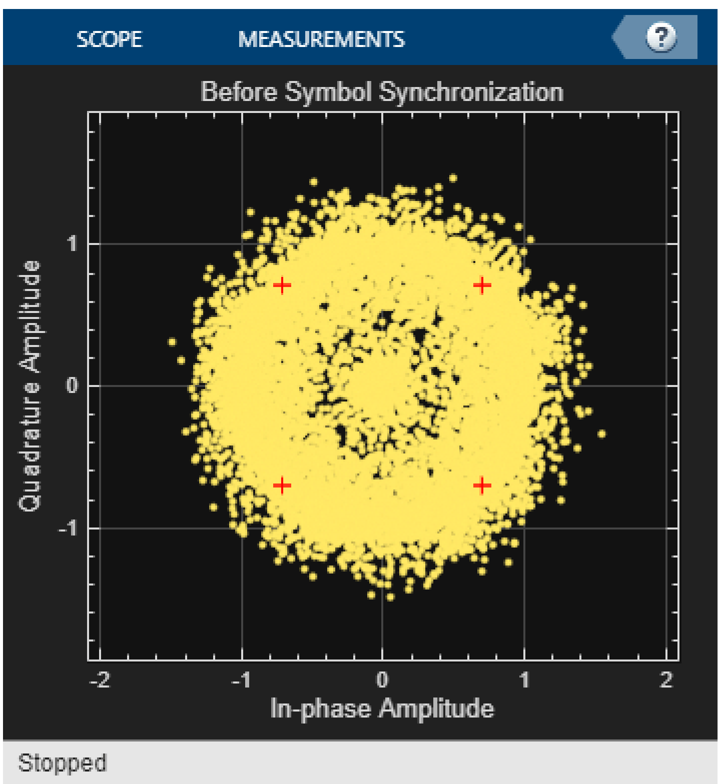
Open the `commhdlQPSKTxRx` model. You can set custom data on the `Input Data` subsystem mask and set the channel configuration on the `Channel` subsystem mask. Run the model. After simulation, the `QPSKTxRxVerification` script verifies the `commhdlQPSKTxRx` model outputs. This script generates a reference waveform to compare it with the transmitter output, and compares the transmitted bits with the receiver decoded user bits.

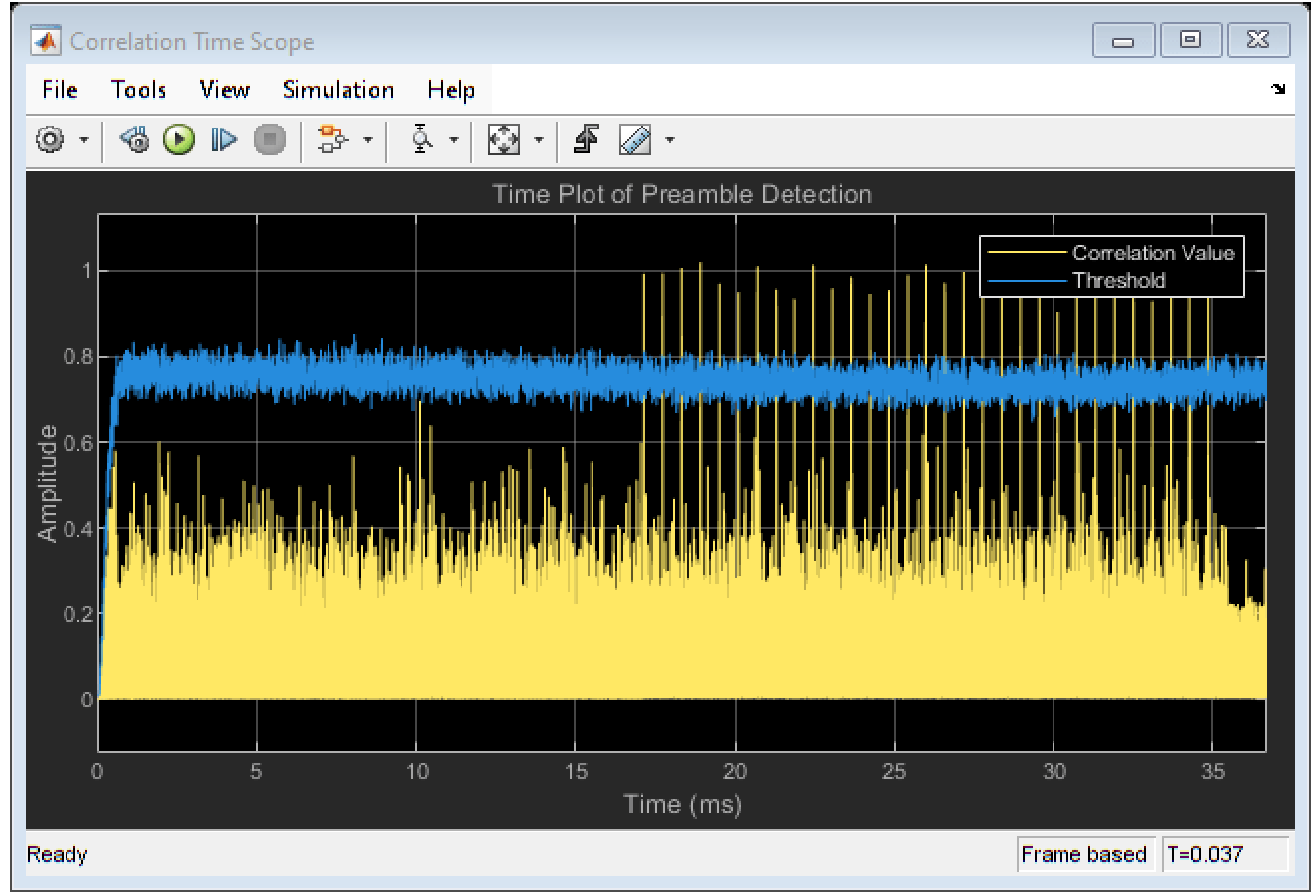
```
Simulation completed
Running the verification script

QPSK Tx:
Maximum absolute symbol error: Real:1.4496e-05 Imaginary:1.4496e-05

Maximum absolute RRC output error: Real:7.8708e-05 Imaginary:7.8708e-05

QPSK Rx:
Initial frames not compared : 29

Number of packets missed = 0 out of 30

Number of packets false detected = 0 out of 30

Number of bits errored = 0 out of 67200
```


SCOPE
MEASUREMENTS
0.8
0.4
0
-0.4
-0.8
Quadrature Amplitude
-1
0
1
In-phase Amplitude
Modulator Output
RRC Filtered Output
Stopped


SCOPE
MEASUREMENTS
1.2
0.8
0.4
0
-0.4
-0.8
-1.2
Quadrature Amplitude
-1
-0.5
0
0.5
1
In-phase Amplitude
After Symbol Synchronization
After Carrier Synchronization
Stopped


SCOPE
RRC Matched Filtered Output
Real Amplitude
1
0
-1
0
0.4
0.8
1.2
1.6
2
2.4
2.8
3.2
3.6
4
Time
Imaginary Amplitude
1
0
-1
0
0.4
0.8
1.2
1.6
2
2.4
2.8
3.2
3.6
4
Time
Stopped


SCOPE
MEASUREMENTS
Before Symbol Synchronization
Quadrature Amplitude
In-phase Amplitude
1
0
-1
-2
-1
0
1
2
Stopped



Generate HDL Code

To check and generate HDL code, you must have an HDL Coder™ license.

To generate the HDL code for the `QPSK Tx` and `QPSK Rx` subsystems, use the following commands:

`makehdl('commhdlQPSKTxRx/QPSK Tx')` and `makehdl('commhdlQPSKTxRx/QPSK Rx')`

To generate test bench, use the following commands:

`makehdltb('commhdlQPSKTxRx/QPSK Tx')` and `makehdltb('commhdlQPSKTxRx/QPSK Rx')`

Test bench generation time depends on the simulation time.

The resulting HDL code is synthesized for the Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization is shown in this table. The maximum frequency of operation is 320 MHz for the transmitter and 196 MHz for the receiver.

Resources	Tx Usage	Rx Usage
Slice Registers	250	14303
Slice LUT	137	8884
RAMB36	0	5
RAMB18	1	1

```
    DSP48                18          118
```

Further Exploration

You can modify the channel conditions by tuning the variables listed in the following table. You can change these values on the `Channel` subsystem mask in the `commhdlQPSKTxRx` model.

```
        Variable Name                                          Description
    ______________________    _________________________________________________________________

    fractionalTimingOffset    Normalized timing phase offset specified in the range >= 0 and < 1
    timingFrequencyOffset     Timing frequency offset specified in PPM
    EbN0dB                    Energy per information bit to single sided noise power spectral de
    CFO                       Carrier frequency offset specified in Hz
    CPO                       Carrier phase offset specified in degrees
```

References

1. Michael Rice, *Digital Communications - A Discrete-Time Approach*, Prentice Hall, April 2008.

2. Floyd M.Gardner, *Phaselock Techniques*, Third Edition, John Wiley & Sons, Inc., 2005.

See Also

Blocks
QPSK Modulator Baseband | QPSK Demodulator Baseband

Code Generation Options in the HDL Coder Dialog Boxes

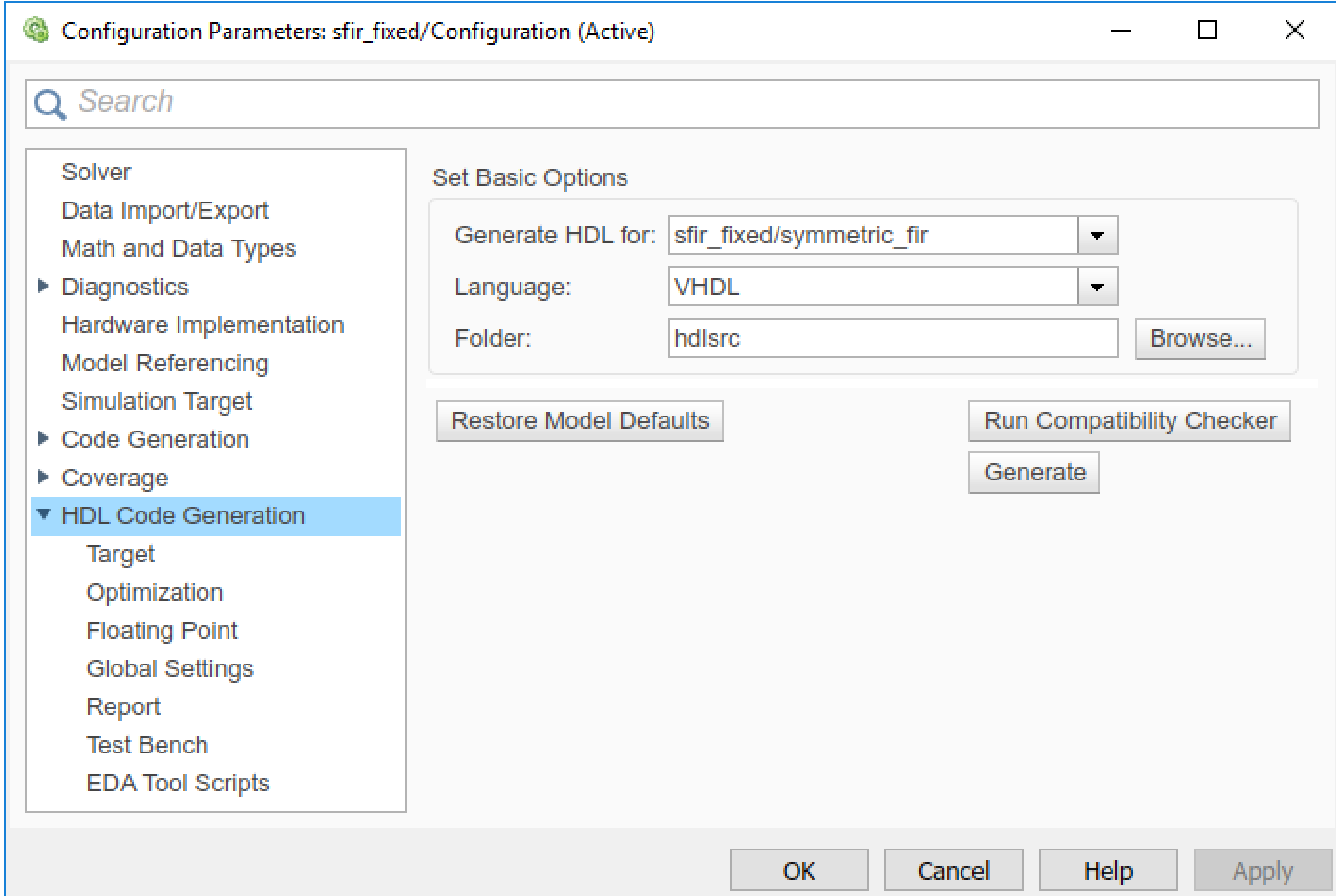
- "Set HDL Code Generation Options" on page 16-2
- "Generate HDL Code from Simulink Model Using Configuration Parameters" on page 16-6
- "Generate HDL Code from Simulink Model from Command Line" on page 16-10

Set HDL Code Generation Options

In this section...
"HDL Code Generation Options in the Configuration Parameters Dialog Box" on page 16-2
"HDL Code Tab in Simulink Toolstrip" on page 16-3
"HDL Code Options in the Block Context Menu" on page 16-4
"The HDL Block Properties Dialog Box" on page 16-4

HDL Code Generation Options in the Configuration Parameters Dialog Box

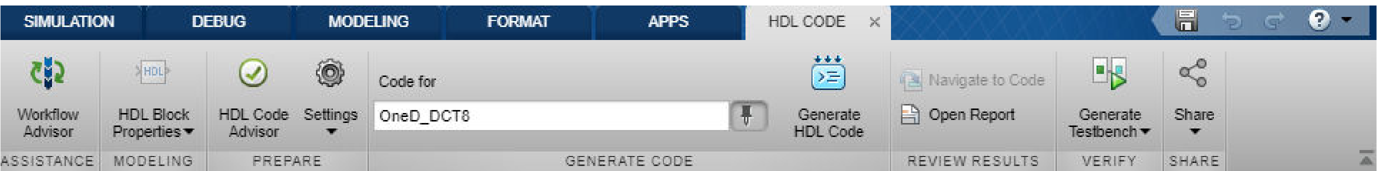
The following figure shows the top-level **HDL Code Generation** pane in the Configuration Parameters dialog box. To open this dialog box, in the Apps gallery, click **HDL Coder**. The **HDL Code** tab appears. In the **Prepare** section, click **Settings**.



Note When the **HDL Code Generation** pane of the Configuration Parameters dialog box appears, clicking the **Help** button displays general help for the Configuration Parameters dialog box.

HDL Code Tab in Simulink Toolstrip

The Simulink Toolstrip contains contextual tabs that appear only when you need to access them. To access the **HDL Code** tab, open the **HDL Coder** app from the **Apps** tab on the Simulink Toolstrip.



The **HDL Code** tab provides shortcuts to the HDL code generation options. You can also use this tab to initiate code generation.

Options include:

- **Workflow Advisor**: Open the HDL Workflow Advisor.
- **HDL Block Properties**: Open the HDL-compatible block library in the Simulink Library Browser or open the HDL Block Properties dialog box for a block that you select in your model.

  Note After you open the HDL-compatible block library, to restore the Library Browser to the default view, in the Library Browser, click the button.

- **HDL Code Advisor**: Open the HDL Code Advisor for the model or the selected Subsystem.
- **Settings**: Open the **HDL Code Generation** pane in the Configuration Parameters dialog box.
  - **Report Options**: Open the **HDL Code Generation** > **Report** pane.
  - **Remove HDL Configuration from Model**: The HDL configuration component is internal data that HDL Coder creates and attaches to a model. This component lets you view the **HDL Code Generation** pane in the Configurations Parameters dialog box, and use the **HDL Code Generation** pane to set HDL code generation options. To remove the HDL Code Generation configuration component to or from a model, select this option. For more information, see “Add or Remove the HDL Configuration Component” on page 23-37.
- **Code for**: Select the top-level Subsystem or model for which you want to generate HDL code. This option corresponds to the **Generate HDL for** option in the **HDL Code Generation** pane of the Configuration Parameters dialog box.
- **Generate HDL Code**: Initiate HDL code generation; equivalent to the **Generate HDL Code** check box in the **HDL Code Generation** > **Global Settings** > **Advanced** tab of the Configuration Parameters dialog box.
- **Navigate to Code**: Select a block in your model and navigate to the HDL code generated for that block. To use this setting, you must have generated a traceability report.
- **Open Report**: Opens the Code Generation Report if this report exists on the path. Otherwise, this button opens the HDL Check Report.
- **Generate Test Bench**: Initiate test bench code generation; equivalent to the **Generate Test Bench** button in the Configuration Parameters dialog box. To use this button, you If you do not select a subsystem in the **Generate HDL for** menu, the **Generate Test Bench** menu option is not available.

  If you have HDL Verifier™ installed, you can generate a HDL cosimulation model or a SystemVerilog DPI component.

- **Share**: Generate a protected model that you can share with a third party without revealing the intellectual property of the model.

HDL Code Options in the Block Context Menu

When you right-click a block that HDL Coder supports, the context menu for the block includes an **HDL Code** submenu. The code generator enables items in the submenu according to:

- The block type: for subsystems, the menu enables some options that are specific to subsystems.
- Whether or not code and traceability information has been generated for the block or subsystem.

Note You can also access the options in the context menu from the **HDL Code** tab in the Simulink Toolstrip. To access this tab, open the **HDL Coder** app from the **Apps** tab.

The following summary describes the **HDL Code** submenu options.

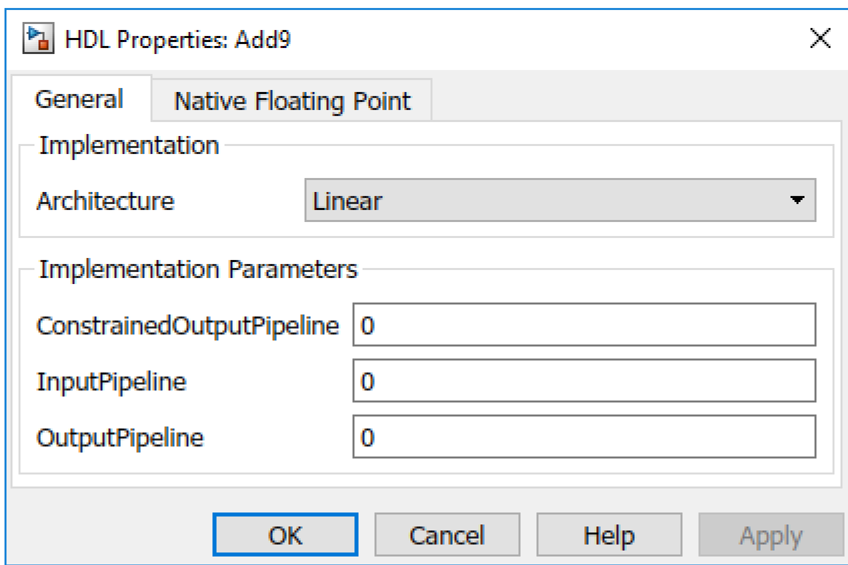
Option	Description	Availability
Check Subsystem Compatibility	Runs the HDL compatibility checker (`checkhdl`) on the subsystem.	Available only for subsystems.
Generate HDL for Subsystem	Runs the HDL code generator (`makehdl`) and generates code for the subsystem.	Available only for subsystems.
HDL Coder Properties	Opens the Configuration Parameters dialog box, with the top-level HDL Code Generation pane selected.	Available for blocks or subsystems.
HDL Block Properties	Opens a block properties dialog box for the block or subsystem. See "Set and View HDL Model and Block Parameters" on page 19-52 for more information.	Available for blocks or subsystems.
HDL Workflow Advisor	Opens the HDL Workflow Advisor for the subsystem.	Available only for subsystems.
Navigate to Code	Activates the HTML code generation report window, displaying the beginning of the code generated for the selected block or subsystem. For more information, see "Navigate Between Simulink Model and HDL Code by Using Traceability" on page 23-12.	Enabled when both code and a traceability report have been generated for the block or subsystem.

The HDL Block Properties Dialog Box

HDL Coder provides selectable alternate block implementations for many block types. Each implementation is optimized for different characteristics, such as speed or chip area. The HDL Properties dialog box lets you choose the implementation for a selected block.

Most block implementations support a number of implementation parameters that let you control further details of code generation for the block. The HDL Properties dialog box lets you set implementation parameters for a block.

The following figure shows the HDL Properties dialog box for a block.



There are a number of ways to specify implementations and implementation parameters for individual blocks or groups of blocks. See “Set and View HDL Model and Block Parameters” on page 19-52.

See Also

`makehdl` | `makehdltb`

Generate HDL Code from Simulink Model Using Configuration Parameters

In this section...
"FIR Filter Model" on page 16-6
"Create a Folder and Copy Relevant Files" on page 16-7
"Open HDL Code Generation Pane of Configuration Parameters Dialog Box" on page 16-8
"Generate HDL Code" on page 16-8

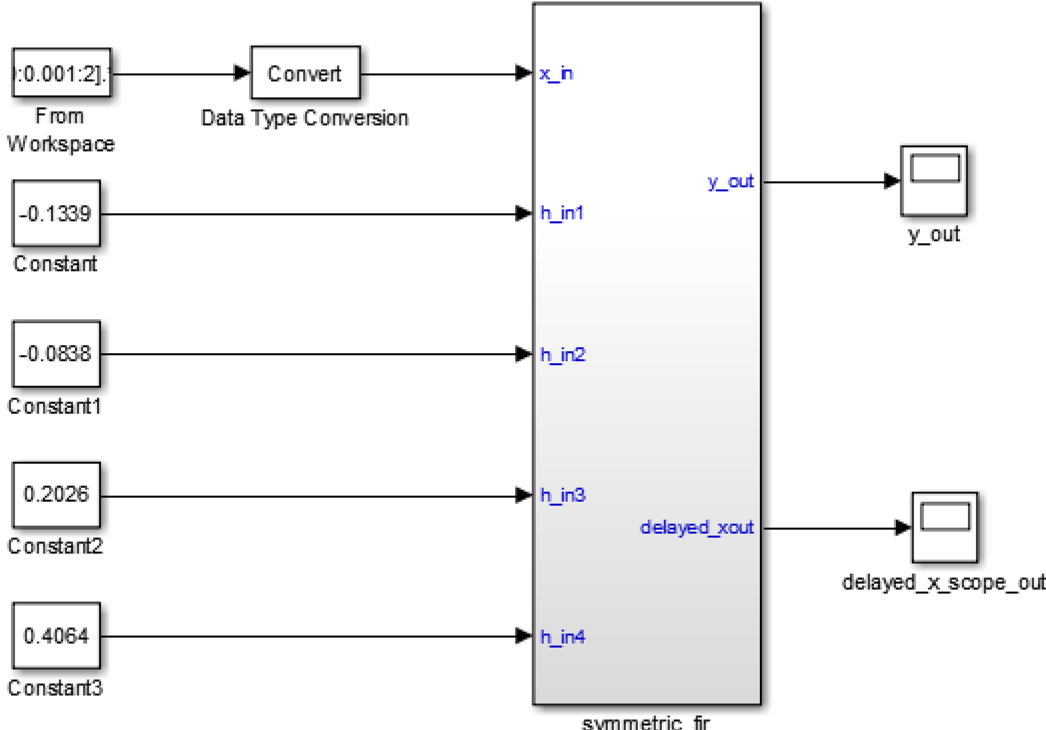
You can view and edit options and parameters that affect HDL code generation in the Configuration Parameters dialog box, or in the Model Explorer. This example illustrates how you can use the Configuration Parameters dialog box to generate HDL code for the Symmetric FIR filter model.

FIR Filter Model

Before you generate HDL code, the model must be compatible for HDL code generation. To check and update your model for HDL compatibility, see "Check HDL Compatibility of Simulink Model Using HDL Code Advisor" on page 38-2.

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



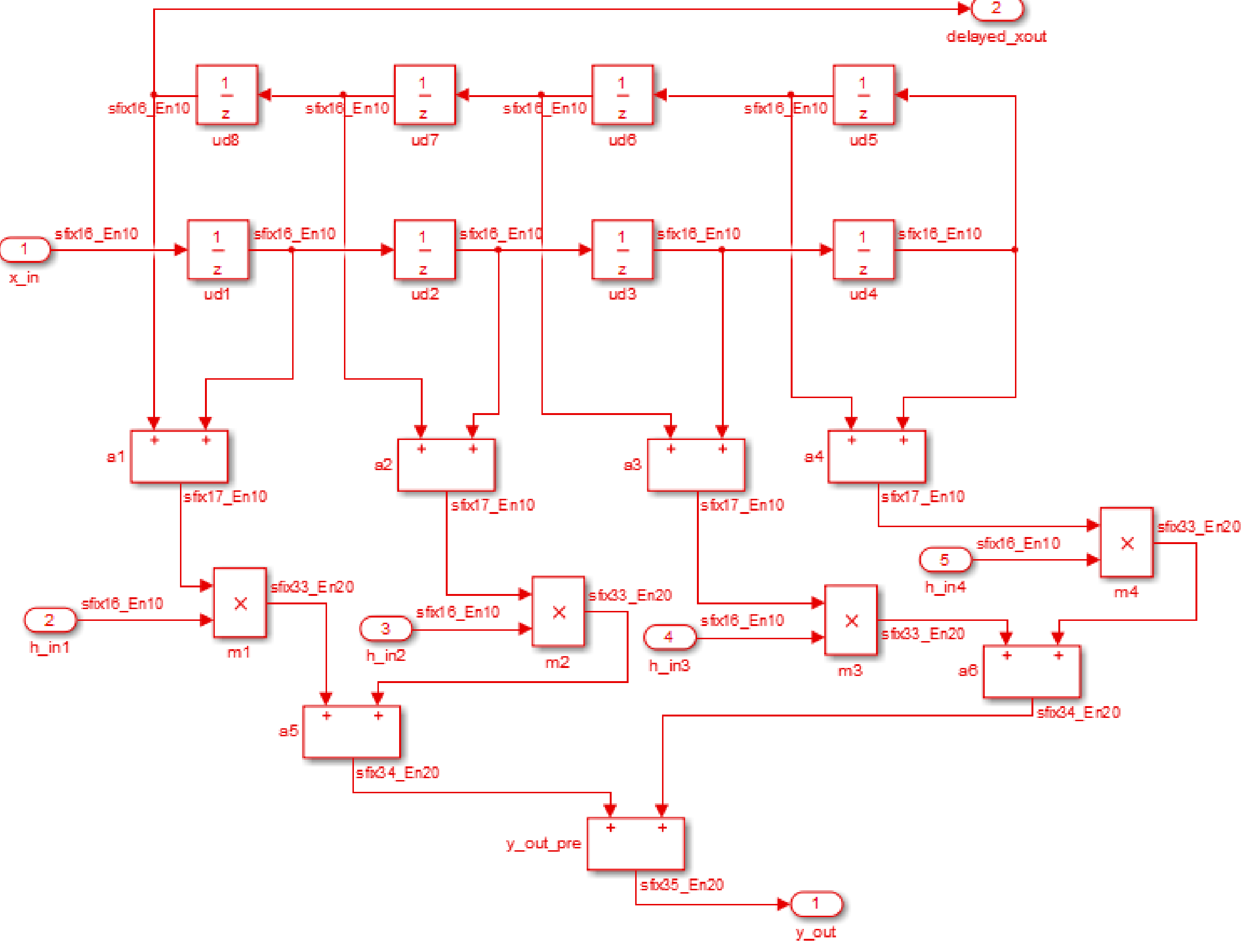
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



Create a Folder and Copy Relevant Files

In MATLAB:

1 Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

sl_hdlcoder_work stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

2. Make the sl_hdlcoder_work folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

3. Save a local copy of the sfir_fixed model to your current working folder. Leave the model open.

Open HDL Code Generation Pane of Configuration Parameters Dialog Box

To open the **HDL Code Generation** pane of the Configuration Parameters dialog box, in the Apps gallery, click **HDL Coder**. The **HDL Code** tab appears. In the **Prepare** section, click **Settings**.

The **HDL Code Generation** pane consists of basic options that specify the DUT that you want to generate code for, target language, and folder settings. The **Generate HDL for** setting is synchronized with the **Code for** menu in the **HDL Code** tab. You can also use the buttons in this pane to initiate code generation and perform compatibility checking. The **HDL Code Generation** pane consists of various subpanes that you can use to specify various settings related to clock and reset signals to reporting and optimization settings.

In the **HDL Code Generation** pane

- The **Generate HDL for** field specifies the sfir_fixed/symmetric_fir subsystem for code generation.
- The **Language** field specifies generation of VHDL code.
- The **Folder** field specifies a target folder that stores generated code files and scripts.

Generate HDL Code

To generate code, click the **Generate** button. By default, HDL Coder generates VHDL code in the target hdlsrc folder.

To generate Verilog code for the model:

1. In the **HDL Code** tab, click **Settings**.
2. In the **HDL Code Generation** pane, for **Language**, select Verilog. Leave other settings to the default. Click **Apply** and then click **Generate**.

HDL Coder compiles the model before generating code. Depending on model display options such as port data types, the model can change in appearance after code generation. As code generation proceeds, HDL Coder displays progress messages in the MATLAB command line with:

- Link to the Configuration Set that indicates the model for which the Configuration Parameters are applied.
- Links to the generated files. To view the files in the MATLAB Editor, click the links.
  - symmetric_fir.vhd: VHDL code. This file contains an entity definition and RTL architecture implementing the symmetric_fir.vhd filter.

- `symmetric_fir_compile.do`: Mentor Graphics ModelSim compilation script (vcom command) to compile the generated VHDL code.
- `symmetric_fir_synplify.tcl`: Synplify® synthesis script.
- `symmetric_fir_map.txt`: This report maps generated entities to the subsystems that generated them. See “Trace Code Using the Mapping File” on page 23-34

The process completes with the message:

```
### HDL Code Generation Complete.
```

See Also

`makehdl` | `makehdltb`

More About

- “Set HDL Code Generation Options” on page 16-2

Generate HDL Code from Simulink Model from Command Line

In this section...
“FIR Filter Model” on page 16-10
“Create a Folder and Copy Relevant Files” on page 16-11
“Generate HDL Code” on page 16-12

You can customize and edit HDL code generation options and then generate code at the command line. This example illustrates how you can use the Configuration Parameters dialog box to generate HDL code for the Symmetric FIR filter model.

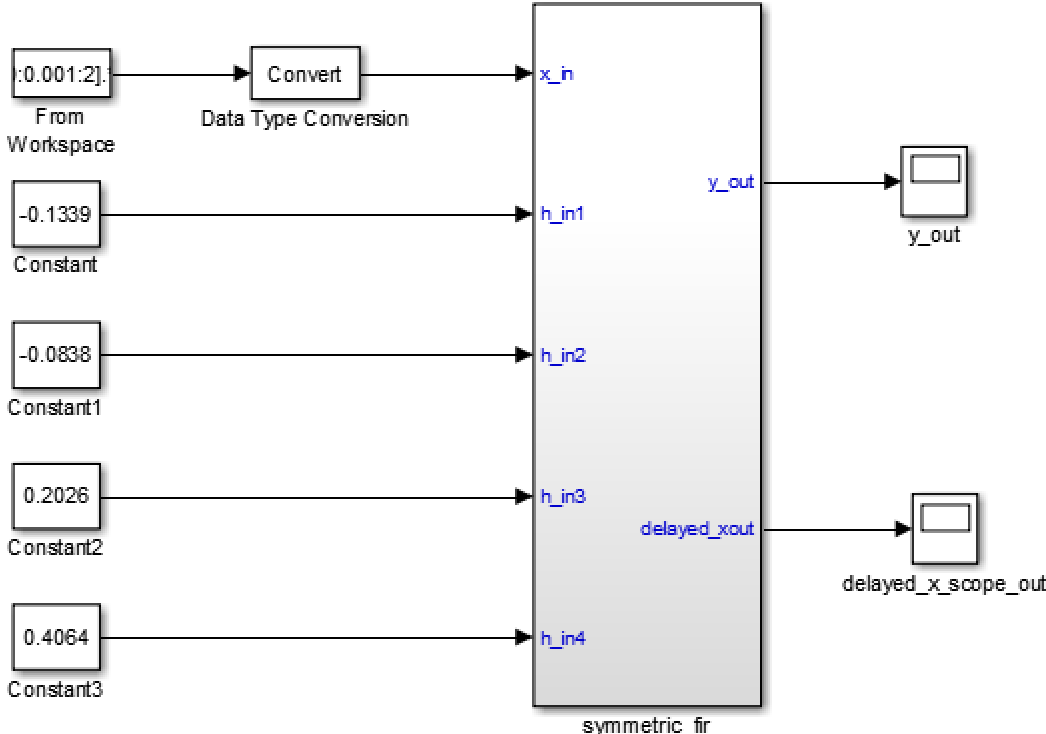
FIR Filter Model

Before you generate HDL code, the model must be compatible for HDL code generation. To check and update your model for HDL compatibility, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 38-2. You can also customize the model parameters by using the `hdlsetup` function.

```
hdlsetup(gcs)
```

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



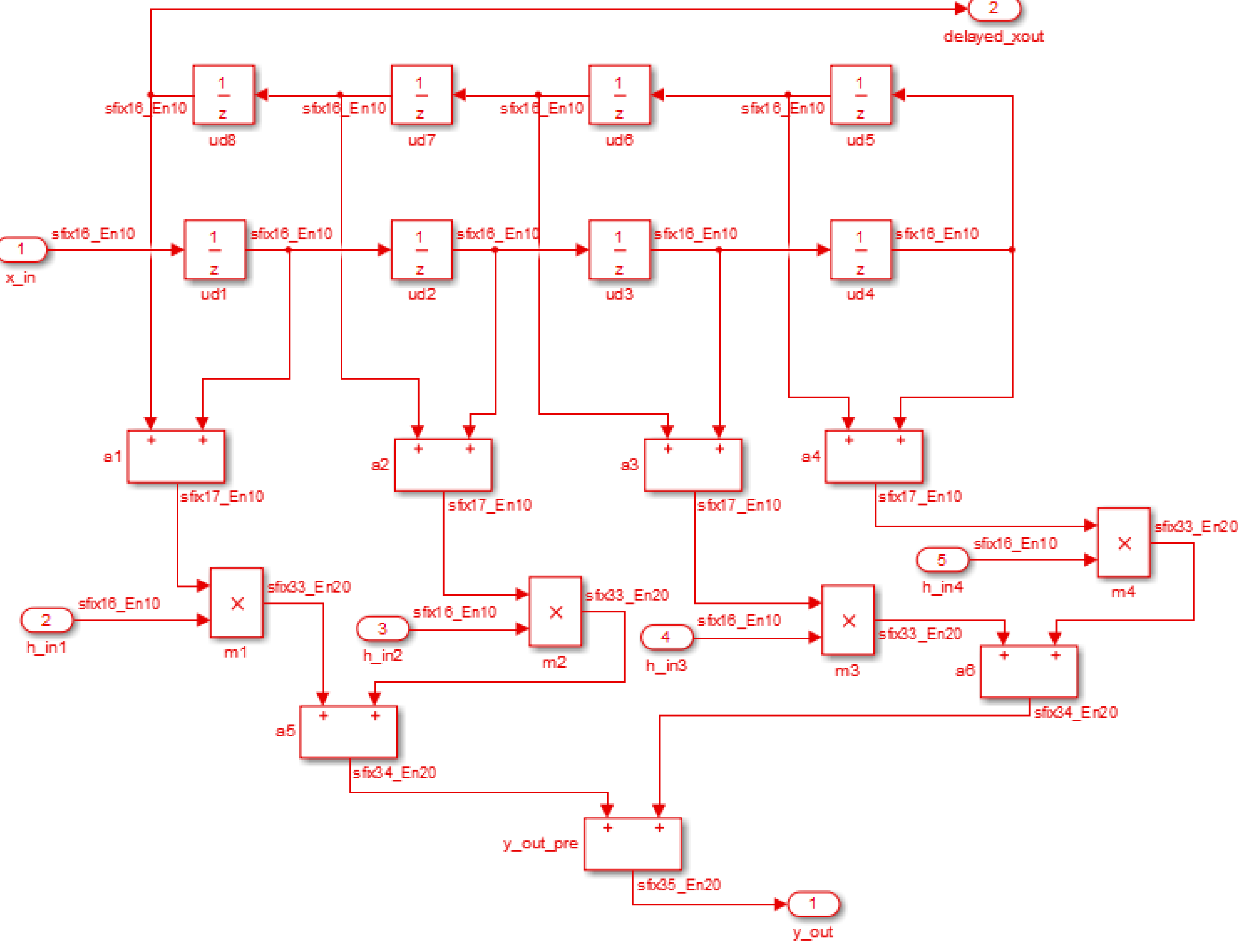
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



Create a Folder and Copy Relevant Files

In MATLAB:

1 Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

sl_hdlcoder_work stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

2. Make the sl_hdlcoder_work folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

3. Save a local copy of the sfir_fixed model to your current working folder. Leave the model open.

Generate HDL Code

To generate HDL code for the DUT, you use the makehdl function. For example, to generate HDL code for the symmetric_fir subsystem, enter:

```
makehdl('sfir_fixed/symmetric_fir')
```

To specify the customizations before you generate HDL code, use the hdlset_param function. You can also specify various name-value pair arguments with the makehdl function to customize HDL code generation options while generating HDL code. For example, to generate Verilog code, use the TargetLanguage property.

```
makehdl('sfir_fixed/symmetric_fir', 'TargetLanguage', 'Verilog')
```

Alternatively, if you are using hdlset_param, set this parameter on the model and then run the makehdl function.

```
hdlset_param('sfir_fixed', 'TargetLanguage', 'Verilog')
makehdl('sfir_fixed/symmetric_fir')
```

HDL Coder compiles the model before generating code. Depending on model display options such as port data types, the model can change in appearance after code generation. As code generation proceeds, HDL Coder displays progress messages in the MATLAB command line with:

- Link to the Configuration Set that indicates the model for which the Configuration Parameters are applied.
- Links to the generated files. To view the files in the MATLAB Editor, click the links.
  - symmetric_fir.vhd: VHDL code. This file contains an entity definition and RTL architecture implementing the symmetric_fir.vhd filter.
  - symmetric_fir_compile.do: Mentor Graphics ModelSim compilation script (vcom command) to compile the generated VHDL code.
  - symmetric_fir_synplify.tcl: Synplify synthesis script.
  - symmetric_fir_map.txt: This report maps generated entities to the subsystems that generated them. See “Trace Code Using the Mapping File” on page 23-34

The process completes with the message:

```
### HDL Code Generation Complete.
```

See Also

makehdl | makehdltb

More About

- "Set HDL Code Generation Options" on page 16-2

HDL Code Generation Pane: General

Modeling Guidelines

- “HDL Modeling Guidelines Severity Levels” on page 18-3
- “Model Design and Compatibility Guidelines - By Numbered List” on page 18-4
- “Guidelines for Supported Blocks and Data Types - By Numbered List” on page 18-7
- “Guidelines for Speed and Area Optimizations - By Numbered List” on page 18-12
- “Basic Guidelines for Modeling HDL Algorithm in Simulink” on page 18-14
- “Guidelines for Model Setup and Checking Model Compatibility” on page 18-20
- “Modeling with Simulink, Stateflow, and MATLAB Function Blocks” on page 18-24
- “Terminate Unconnected Block Outputs and Usage of Commenting Blocks” on page 18-27
- “Identify and Programmatically Change and Display HDL Block Parameters” on page 18-31
- “DUT Subsystem Guidelines” on page 18-37
- “Hierarchical Modeling Guidelines” on page 18-41
- “Design Considerations for Matrices and Vectors” on page 18-47
- “Use Bus Signals to Improve Readability of Model and Generate HDL Code” on page 18-52
- “Guidelines for Clock and Reset Signals” on page 18-58
- “Modeling with Native Floating Point” on page 18-65
- “Design Considerations for RAM Blocks and Blocks in HDL Operations Library” on page 18-68
- “Usage of Blocks in Logic and Bit Operations Library” on page 18-72
- “Generate FPGA Block RAM from Lookup Tables” on page 18-78
- “Recommended Block Parameter Settings of Multiport Switch Block for Numeric and Enumerated Types” on page 18-83
- “Guidelines for Using Selector Blocks to Extract Input Elements from Vector or Matrix Signals” on page 18-87
- “Guidelines for Using Assignment Blocks to Write Elements in Vectors, Matrices, and 3-D Arrays” on page 18-91
- “Usage of Different Subsystem Types” on page 18-93
- “Usage of Rate Change and Constant Blocks” on page 18-100
- “Guidelines for Using Delays and Goto and From Blocks for HDL Code Generation” on page 18-105
- “Modeling Efficient Multiplication and Division Operations for FPGA Targeting” on page 18-109
- “Using Persistent Variables and fi Objects Inside MATLAB Function Blocks for HDL Code Generation” on page 18-117
- “Guidelines for HDL Code Generation Using Stateflow Charts” on page 18-123
- “Simulink Data Type Considerations” on page 18-134
- “Guidelines for Using Rounding and Saturation Settings for Fixed-Point Data Types” on page 18-144
- “Guideline for Using Sqrt Block for HDL Code Generation” on page 18-146

- “Resource Sharing Settings for Various Blocks” on page 18-148
- “Resource Sharing of Subsystems and Floating-Point IPs” on page 18-152
- “Resource Sharing Guidelines for Vector Processing and Matrix Multiplication” on page 18-156
- “Distributed Pipelining and Clock-Rate Pipelining Guidelines” on page 18-161
- “Insert Distributed Pipeline Registers for Blocks with Vector Data Type Inputs” on page 18-164

HDL Modeling Guidelines Severity Levels

Each modeling guideline for HDL code generation has a different level of severity that indicates the levels of compliance requirements. This table illustrates what each severity level indicates.

Severity Levels

Category	Mandatory	Strongly Recommended	Recommended	Informative
Definition	Guidelines that are absolutely essential to follow. Models created must conform to these guidelines to 100%.	Guidelines that are agreed upon to be a good practice. Models created should conform to these guidelines to the greatest extent possible, but does not have to be 100%.	Guidelines that are recommended to improve the generated code and optimize the code on the target device, but are not critical	Guidelines that are meant to understand some modeling recommendations and best practices.
Impact	If you violate these guidelines, you cannot generate code and synthesize your design on the target hardware.	If you violate these guidelines, you get poor quality of results.	Violating these guidelines may impact the efficiency or ease of using the generated code with downstream synthesis tools	None

See Also

More About

- “Model Design and Compatibility Guidelines - By Numbered List” on page 18-4
- “Guidelines for Supported Blocks and Data Types - By Numbered List” on page 18-7
- “Guidelines for Speed and Area Optimizations - By Numbered List” on page 18-12

Model Design and Compatibility Guidelines - By Numbered List

The HDL modeling guidelines are a set of recommended guidelines that you can follow when creating Simulink model for code generation with HDL Coder. The model design and compatibility guidelines consist of guidelines for basic block usage, clock and reset signals, buses and vectors, and subsystem and hierarchical designing. Each modeling guideline for HDL code generation has a different level of severity that indicates the levels of compliance requirements. To learn more about these severity levels, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

These tables list the model design and compatibility guidelines in HDL Coder. These guidelines start from `1.1` and are divided into subsections. In the table, you see that certain guidelines have an associated model check. You can follow the modeling pattern recommended for these guidelines by running that check in the HDL Code Advisor. To learn more about the HDL Code Advisor, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 38-2.

Guidelines 1.1: Basic Settings

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.1.1	“Use HDL-Supported Blocks” on page 18-14	Recommended	None
1.1.2	“Partition Model into DUT and Test Bench” on page 18-15	Recommended	None
1.1.3	“Avoid Using Double-Byte Characters” on page 18-17	Mandatory	None
1.1.4	“Document Model Features and Attributes” on page 18-17	Recommended	None
1.1.5	“Customize hdlsetup Function Based on Target Application” on page 18-20	Recommended	Model Check: “Check for model parameters suited for HDL code generation” on page 37-5
1.1.6	“Check Subsystem for HDL Compatibility” on page 18-21	Recommended	None
1.1.7	“Run Model Checks for HDL Coder” on page 18-21	Recommended	None
1.1.8	“Modeling with Simulink, Stateflow, and MATLAB Function Blocks” on page 18-24	Informative	None
1.1.9	“Terminate Unconnected Block Outputs” on page 18-27	Mandatory	None
1.1.10	“Using Comment Out and Comment Through of Blocks” on page 18-28	Informative	None
1.1.11	“Adjust Sizes of Constant and Gain Blocks for Identifying Parameters” on page 18-31	Recommended	None

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.1.12	"Display Parameters That Affect HDL Code Generation" on page 18-31	Recommended	None
1.1.13	"Change Block Parameters by Using find_system and set_param" on page 18-36	Informative	None

Guidelines 1.2: DUT Subsystem and Hierarchical Modeling

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.2.1	"DUT Subsystem Considerations" on page 18-37	Strongly Recommended	Model Check: "Check for invalid top level subsystem" on page 37-13
1.2.2	"Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks" on page 18-37	Strongly Recommended	None
1.2.3	"Insert Handwritten Code into Simulink Modeling Environment" on page 18-39	Informative	None
1.2.4	"Avoid Constant Block Connections to Subsystem Port Boundaries" on page 18-41	Mandatory	None
1.2.5	"Generate Parameterized HDL Code for Constant and Gain Blocks" on page 18-42	Recommended	None
1.2.6	"Place Physical Signal Lines Inside a Subsystem" on page 18-44	Mandatory	None

Guidelines 1.3: Vectors, Matrices, and Buses

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.3.1	"Modeling Requirements for Matrices" on page 18-47	Mandatory	None
1.3.2	"Avoid Generating Ascending Bit Order in HDL Code From Vector Signals" on page 18-48	Strongly Recommended	None
1.3.3	"Use Bus Signals to Improve Readability of Model and Generate HDL Code" on page 18-52	Informative	None

Guidelines 1.4: Clock Bundle Signals

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.4.1	"Use Global Oversampling to Create Frequency-Divided Clock" on page 18-58	Informative	None
1.4.2	"Create Multirate Model with Integer Clock Multiples by Clock Division" on page 18-58	Mandatory	None
1.4.3	"Use Dual Rate Dual Port RAM for Noninteger Multiple Sample Times" on page 18-61	Mandatory	None
1.4.4	"Asynchronous Clock Modeling in HDL Coder" on page 18-62	Recommended	None
1.4.5	"Use Global Reset Type Setting Based on Target Hardware" on page 18-64	Strongly Recommended	Model check: "Check for global reset setting for Xilinx and Altera devices" on page 37-7

Guidelines 1.5: Native Floating Point

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.5.1	"Modeling with Native Floating Point" on page 18-65	Recommended	None

See Also

More About

- "Guidelines for Supported Blocks and Data Types - By Numbered List" on page 18-7
- "Guidelines for Speed and Area Optimizations - By Numbered List" on page 18-12

Guidelines for Supported Blocks and Data Types - By Numbered List

The HDL modeling guidelines are a set of recommended guidelines that you can follow when creating Simulink model for code generation with HDL Coder. The guidelines for supported blocks and data types consist of guidelines for using various blocks in the HDL Coder block library, and about the supported data types. Each modeling guideline for HDL code generation has a different level of severity that indicates the levels of compliance requirements. To learn more about these severity levels, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

These tables list the guidelines for supported data types in HDL Coder and for various blocks in the HDL Coder block library. The guidelines start from `2.1` and are divided into subsections. In the table, you see that certain guidelines have an associated model check. You can follow the modeling pattern recommended for these guidelines by running that check in the HDL Code Advisor. To learn more about the HDL Code Advisor, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 38-2.

Guidelines 2.1: HDL RAMs and HDL Operations Library

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.1.1	“RAM Block Access Considerations” on page 18-68	Recommended	None
2.1.2	“Serial to Parallel Conversion” on page 18-70	Recommended	None

Guidelines 2.2: Logic and Bit Operations Library

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.2.1	“Logical and Arithmetic Bit Shift Operations” on page 18-72	Informative	None
2.2.2	“Usage of Logical Operator, Bitwise Operator, and Bit Reduce Blocks” on page 18-74	Informative	None
2.2.3	“Use Boolean Output for Compare to Constant and Relational Operator Blocks” on page 18-76	Strongly Recommended	None

Guidelines 2.3: Lookup Table and Signal Routing Blocks

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.3.1	“Generate FPGA Block RAM from Lookup Tables” on page 18-78	Strongly Recommended	None

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.3.2	"Recommended Block Parameter Settings of Multiport Switch Block for Numeric and Enumerated Types" on page 18-83	Recommended	None
2.3.3	"Guidelines for Using Selector Blocks to Extract Input Elements from Vector or Matrix Signals" on page 18-87	Recommended	None
2.3.4	"Guidelines for Using Assignment Blocks to Write Elements in Vectors, Matrices, and 3-D Arrays" on page 18-91	Recommended	None

Guidelines 2.4: Ports and Subsystems

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.4.1	"Virtual Subsystem: Use as DUT" on page 18-93	Mandatory	Model Check: "Check for invalid top level subsystem" on page 37-13
2.4.2	"Atomic and Virtual Subsystems: Generate Reusable HDL Files" on page 18-93	Recommended	None
2.4.3	"Variant Subsystem: Using Variant Subsystems for HDL Code Generation" on page 18-94	Mandatory	None
2.4.4	"Model References: Build Model Design Using Smaller Partitions" on page 18-96	Recommended	None
2.4.5	"Block Settings of Enabled and Triggered Subsystems" on page 18-98	Mandatory	None

Guideline 2.5: Rate Change and Constant Blocks

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.5.1	"Usage of Rate Conversion Blocks" on page 18-100	Recommended	None
2.5.2	"Use Discrete and Finite Sample Time for Constant Block" on page 18-101	Recommended	Model Check: "Check for infinite and continuous sample time sources" on page 37-15

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.5.3	"Dynamically Change Sample Offset for Downsample Block" on page 18-103	Informative	None

Guideline 2.6: Delay Blocks

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.6.1	"Appropriate Usage of Delay Blocks as Registers" on page 18-105	Recommended	"Check for obsolete Unit Delay Enabled/Resettable Blocks" on page 37-20
2.6.2	"Absorb Delays to Avoid Timing Difference" on page 18-105	Recommended	None
2.6.3	"Map Large Delays to Block RAM" on page 18-106	Recommended	None
2.6.4	"Required HDL Settings for Goto and From Blocks" on page 18-107	Mandatory	None

Guideline 2.7: Multiplication and Accumulation Operations

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.7.1	"Designing Multipliers and Adders for Efficient Mapping to DSP Blocks on FPGA" on page 18-109	Strongly Recommended	None
2.7.2	"Set ConstMultiplierOptimization HDL Block Property to auto for Gain Block" on page 18-113	Recommended	None
2.7.3	"Use ShiftAdd Architecture of Divide Block for Fixed-Point Types" on page 18-115	Recommended	None
2.7.4	"Use Gain Block for Fixed-Point Constant Operations" on page 18-115	Strongly Recommended	None

Guideline 2.8: MATLAB Function Blocks

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.8.1	"Update Persistent Variables at End of MATLAB Function" on page 18-117	Strongly Recommended	None
2.8.2	"Avoid Algebraic Loop Errors from Persistent Variables inside MATLAB Function Blocks" on page 18-118	Mandatory	None
2.8.3	"Use hdlfimath Setting and Specify fi Objects inside MATLAB Function Block" on page 18-120	Strongly Recommended	"Check for MATLAB Function block settings" on page 37-18

Guideline 2.9: Stateflow Charts

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.9.1	"Choose State Machine Type based on HDL Implementation Requirements" on page 18-123	Strongly Recommended	None
2.9.2	"Specify Block Configuration Settings of Stateflow Chart" on page 18-123	Strongly Recommended	"Check for Stateflow chart settings" on page 37-19
2.9.3	"Insert Unconditional Transition State for Else Statement in HDL Code" on page 18-124	Recommended	None
2.9.4	"Data Type Settings and Casting in Stateflow Chart for HDL Code Generation" on page 18-127	Informative	None
2.9.5	"Using Absolute Time Temporal Logic in Stateflow Charts" on page 18-129	Mandatory	None
2.9.6	"Modeling Error (default) State in Stateflow Charts" on page 18-129	Informative	None
2.9.7	"Enable Clock-Driven Outputs of Stateflow Charts (Moore Charts Only)" on page 18-130	Informative	None
2.9.8	"Enumeration type for active state monitoring in a Stateflow chart with no default value" on page 18-131	Informative	None

Guidelines 2.10: Data Types

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.10.1	“Use Boolean for Logical Data and Ufix1 for Numerical Data” on page 18-134	Mandatory	None
2.10.2	“Specify Data Type of Gain Blocks” on page 18-134	Recommended	None
2.10.3	“Enumerated Data Type Restrictions” on page 18-135	Mandatory	None
2.10.4	“Choose Optimal Simulink Block to Compute Sine and Cosine Functions with Fixed-Point Data Types” on page 18-135	Recommended	None
2.10.5	“Choose Optimal Simulink Block to Compute Sine and Cosine Functions with Floating-Point Data Types” on page 18-140	Recommended	None
2.10.6	“Guidelines for Using Rounding and Saturation Settings for Fixed-Point Data Types” on page 18-144	Recommended	None

Guidelines 2.11: Square Root Operations

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.11.1	Use SqrtFunction Architecture for Square Root Block	Recommended	None

See Also

More About

- “Model Design and Compatibility Guidelines - By Numbered List” on page 18-4
- “Guidelines for Speed and Area Optimizations - By Numbered List” on page 18-12

Guidelines for Speed and Area Optimizations - By Numbered List

The HDL modeling guidelines are a set of recommended guidelines that you can follow when creating Simulink model for code generation with HDL Coder. In addition to providing architectural guidance, because the generated code targets hardware platforms such as FPGAs, ASICs, and SoCs, you can use these guidelines to optimize your design for speed or area on the target hardware.. Each modeling guideline for HDL code generation has a different level of severity that indicates the levels of compliance requirements. To learn more about these severity levels, see "HDL Modeling Guidelines Severity Levels" on page 18-3.

These tables list the guidelines for speed and area optimizations in HDL Coder. The guidelines start from `3.1` and are divided into subsections. These guidelines do not have an associated model check. You can follow the modeling pattern recommended for these guidelines by running that check in the HDL Code Advisor. To learn more about the HDL Code Advisor, see "Check HDL Compatibility of Simulink Model Using HDL Code Advisor" on page 38-2.

Guidelines 3.1: Resource Sharing and Streaming

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
3.1.1	"Resource Sharing of Add Blocks" on page 18-148	Recommended	None
3.1.2	"Resource Sharing of Gain Blocks" on page 18-149	Recommended	None
3.1.3	"Resource Sharing of Product Blocks" on page 18-150	Recommended	None
3.1.4	"Resource Sharing of Multiply-Add Blocks" on page 18-150	Recommended	None
3.1.5	"General Considerations for Sharing of Subsystems" on page 18-152	Recommended	None
3.1.6	"Use MATLAB Datapath Architecture for Sharing with MATLAB Function Blocks" on page 18-153	Recommended	None
3.1.7	"Sharing of Subsystems" on page 18-153	Recommended	None
3.1.8	"Resource Sharing of Floating-Point IPs" on page 18-154	Recommended	None
3.1.9	"Use StreamingFactor for Resource Sharing of Vector Signals" on page 18-156	Informative	None

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
3.1.10	“Use SharingFactor and HDL Block Properties for Sharing Matrix Multiplication Operations” on page 18-159	Informative	None

Guidelines 3.2: Clock Rate Pipelining and Distributed Pipelining

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
3.2.1	“Clock-Rate Pipelining Guidelines” on page 18-161	Informative	None
3.2.2	“Recommended Distributed Pipelining Settings” on page 18-161	Recommended	None
3.2.3	“Insert Distributed Pipeline Registers for Blocks with Vector Data Type Inputs” on page 18-164	Informative	None

See Also

More About

- “Model Design and Compatibility Guidelines - By Numbered List” on page 18-4
- “Guidelines for Supported Blocks and Data Types - By Numbered List” on page 18-7

Basic Guidelines for Modeling HDL Algorithm in Simulink

In this section...

"Use HDL-Supported Blocks" on page 18-14
"Partition Model into DUT and Test Bench" on page 18-15
"Avoid Using Double-Byte Characters" on page 18-17
"Document Model Features and Attributes" on page 18-17

Use these guidelines to develop your HDL algorithm in Simulink. The guidelines include using HDL-supported blocks when modeling your design and how to partition your design when developing the algorithm.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see "HDL Modeling Guidelines Severity Levels" on page 18-3.

Use HDL-Supported Blocks

Guideline ID

`1.1.1`

Severity

Strongly Recommended

Description

When you create your Simulink model, use blocks from the **Simulink Library Browser** > **HDL Coder** library. Several blocks in this library are pre-configured for HDL code generation. Blocks in this library are available with Simulink. If you do not have HDL Coder, you can simulate the blocks in your model, but cannot generate HDL code.

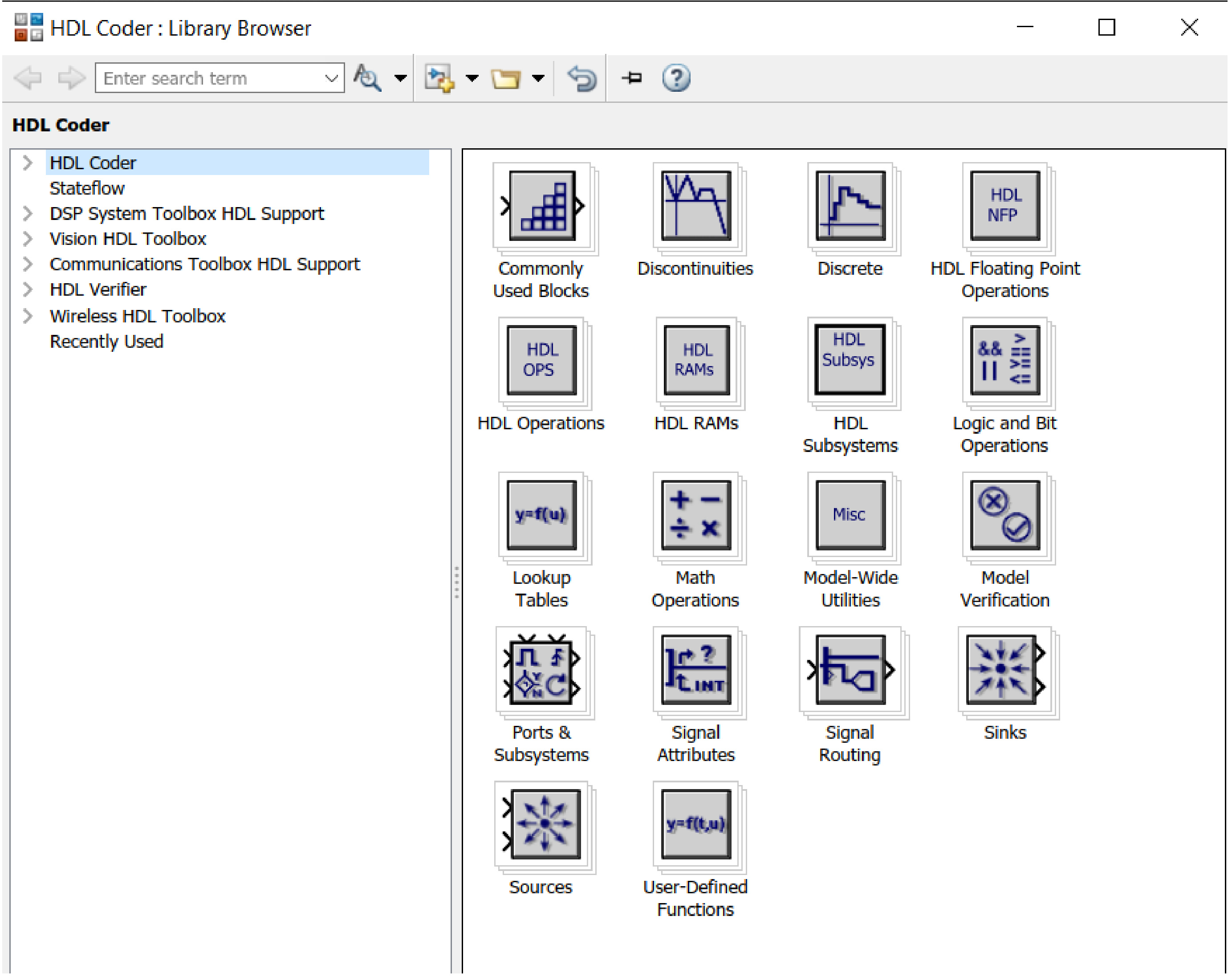
You can find additional HDL-supported blocks in these Simulink block libraries:

- **DSP System Toolbox HDL Support**
- **Communications Toolbox HDL Support**
- **Vision HDL Toolbox**
- **Wireless HDL Toolbox**

To display only HDL-supported blocks in the Library Browser:

- in the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select **HDL Block Properties** > **Open HDL Block Library**.
- Alternatively, at the MATLAB Command Window, enter `hdllib`.

```
hdllib
```



To restore the library browser to the default view, enter this command:

```
hdllib('off')
```

Note The set of supported blocks will change in future releases, so you should rebuild your supported blocks library each time you install a new version of this product.

Partition Model into DUT and Test Bench

Guideline ID

1.1.2

Severity

Recommended

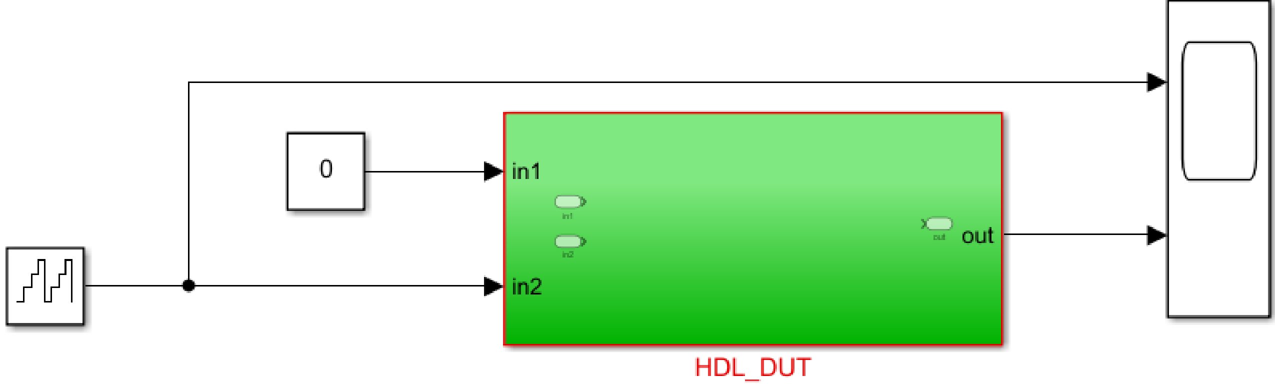
Description

When you create your Simulink model for HDL code generation, the Subsystem that you want to generate HDL code for is the Design-Under-Test (DUT). This Subsystem contains Simulink blocks that can be implemented on your target FPGA or ASIC device. You can further partition the logic inside the DUT into smaller subsystems based on functionality, sample rates in your design, and so on. When you generate HDL code, the DUT becomes the top-level module or entity, and the Subsystems inside the DUT become submodules or smaller entities.

Blocks outside the DUT Subsystem become part of the test bench. The test bench can consist of blocks that are not supported for HDL code generation. Simulate the test bench to:

- Verify the functionality of the DUT in your Simulink model.
- Verify functional equivalence of the generated model with your original model.

For example, if you open the Simulink model template **Blank_DUT**, this model opens in the Simulink Editor.



In this model, **HDL_DUT** Subsystem is the DUT and blocks outside this Subsystem form the test bench. You can develop your HDL algorithm inside the **HDL_DUT** Subsystem. This template model is preconfigured for HDL code generation.

Note You can also generate HDL code for the entire model instead of the DUT Subsystem. Replace the input signals and Constant blocks with Inport blocks. Replace the output signals and Scope blocks with Outport blocks.

Avoid Using Double-Byte Characters

Guideline ID

`1.1.3`

Severity

Strongly Recommended

Description

Downstream synthesis and simulation tools do not support double-byte characters such as Japanese and Chinese characters. HDL Coder does not support using:

- Double-byte characters in model and block names.
- Reserved words of your Operating System in model and block names such as `CR`, `con`, `prn`, `aux`, `ptr`, `null`, `ipt1`, `ipt2`, `ipt3`, and `ipt4`, `com1`, `com2`, `com3`, and `com4`.
- Double-byte characters in comments because the comments are propagated to the generated code. Use English comments instead.

Document Model Features and Attributes

Guideline ID

`1.1.4`

Severity

Recommended

Description

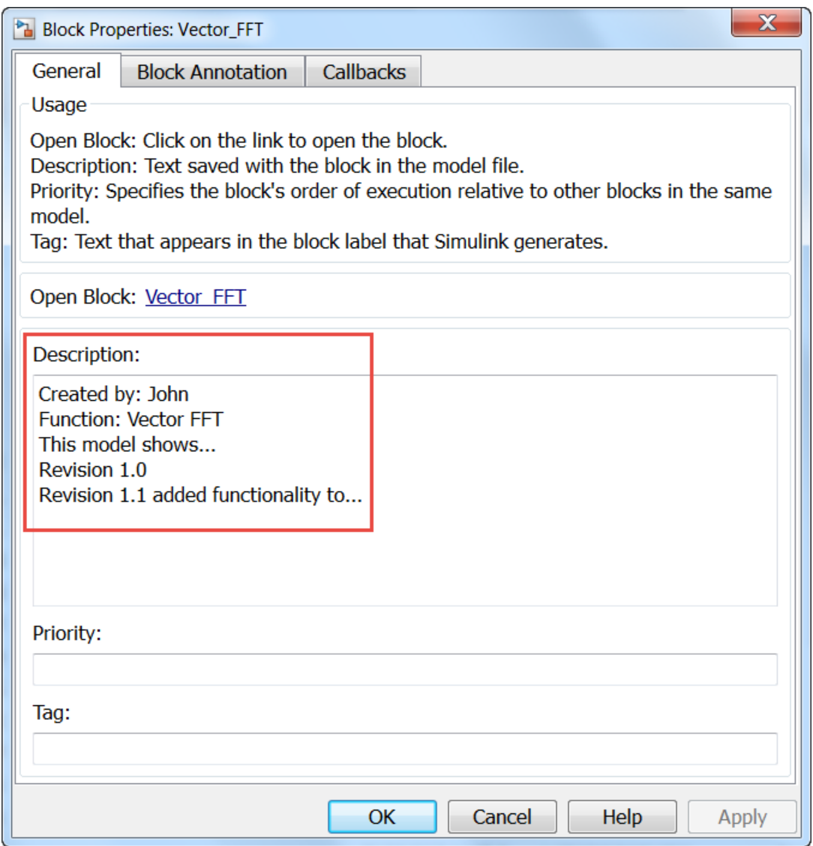
To make the generated HDL code easier to manage, you can document reference information as part of your model settings in these ways:

- **Custom File Headers and Footer Comments in HDL Code for Design and Testbench**

  In the **HDL Code Generation** > **Global Settings** > **Coding Style** tab of the Configuration Parameters dialog box, by using the **Custom File Header Comment** and **Custom File Footer Comment** parameters, you can enter your own custom comments to appear as headers or footers in all generated HDL files. To learn more, see Custom File Header Comment and Custom File Footer Comment.
- **Model and Block Annotations, Text Comments, and Requirement Comments**

  You can add annotations in the form of model annotations, text comments, or requirement comments to the generated code. For example, you can enter text directly on the block diagram as Simulink annotations, or insert text comments by placing a DocBlock in your model. To relate annotations in the block diagram to blocks in your model, use lines to connect the annotations to those blocks. These annotations appear as comments beside the blocks in the generated code. To learn more, see “Generate HDL Code with Annotations or Comments” on page 23-24.
- **Block Features and Attributes as Custom Header Comments for Each File**

In the **Description** section of the Block Properties for subsystems that you use in your design. This information appears as comment headers in the HDL code. For example, this figure illustrates block comments added for a `Vector FFT` Subsystem in your design.



The block comments appear as headers in the generated HDL code.

```
-- Simulink subsystem description for vector_fft_implementation_example/Vector_FFT:
--
-- Created by: John
-- Function: Vector FFT
-- This model shows...
-- Revision 1.0
-- Revision 1.1 added functionality to...
--
--
-------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Vector_FFT IS
```

See Also

Functions

`hdllib` | `checkhdl` | `hdlcodeadvisor`

Modeling Guidelines
"Guidelines for Model Setup and Checking Model Compatibility" on page 18-20 | "Modeling with Simulink, Stateflow, and MATLAB Function Blocks" on page 18-24

More About

- "Use Simulink Templates for HDL Code Generation" on page 14-8
- "Create HDL-Compatible Simulink Model"
- "Display Blocks for HDL Code Generation in Library Browser" on page 23-31

Guidelines for Model Setup and Checking Model Compatibility

In this section...

"Customize hdlsetup Function Based on Target Application" on page 18-20
"Check Subsystem for HDL Compatibility" on page 18-21
"Run Model Checks for HDL Coder" on page 18-21

Use these guidelines to set up your Simulink model for HDL code generation compatibility and verify that your design is ready to generate code.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see "HDL Modeling Guidelines Severity Levels" on page 18-3.

Customize hdlsetup Function Based on Target Application

Guideline ID

1.1.5

Severity

Strongly Recommended

Description

Before generating code, configure the model for HDL code generation by using `hdlsetup`. The `hdlsetup` function uses the `set_param` function to set up models for HDL code generation. To view the settings that `hdlsetup` function saves on the model, enter:

```
edit hdlsetup.m
```

You can customize the `hdlsetup.m` file to only edit parameters required for your target application. For example, you can disable some of the solver settings in the Configuration Parameters and enable certain model parameters such as displaying port data types.

```
% following config parameters are disabled.
%     'Solver',               'fixedstepdiscrete', ...
%     'SaveTime',             'off', ...
%     'SaveOutput',           'off', ...
%     'DataTypeOverride',     'ForceOff',...

% Following model parameters are enabled.
set_param(model, 'ShowLineDimensions', 'on')
set_param(model, 'ShowPortDataTypes', 'on')
set_param(model, 'SampleTimeColors', 'on')
set_param(model, 'WideLines', 'on')
```

To view a custom `hdlsetup` function, enter:

```
edit myhdlsetup.m
```

`myhdlsetup` saves some HDL-specific parameters by using `hdlset_param` on the model.

Check Subsystem for HDL Compatibility

Guideline ID

1.1.6

Severity

Strongly Recommended

Description

The compatibility checker generates a report specified system for compatibility problems, such as use of unsupported blocks, illegal data type usage, and so on.

To run the check for HDL compatibility:

- From the UI, right-click the DUT Subsystem and select **HDL Code > Check Subsystem for HDL compatibility**.
- At the command line, use the `checkhdl` function. Select the DUT Subsystem and then enter this command:

  ```
  checkhdl(gcb)
  ```

See also “Check Your Model for HDL Compatibility” on page 23-29.

When you run this command, the HDL compatibility checker generates an HDL Code Generation Check Report. The report is stored in the target `hdlsrc` folder. If the report does not display any errors, it indicates that your model is compatible for HDL code generation.

```
### Starting HDL Check.
### HDL Check Complete with 0 errors, warnings and messages.
```

Note `checkhdl` does not detect all compatibility issues. Even if HDL check completes without any errors or warnings, HDL Coder can generate errors during code generation.

Run Model Checks for HDL Coder

Guideline ID

1.1.7

Severity

Strongly Recommended

Description

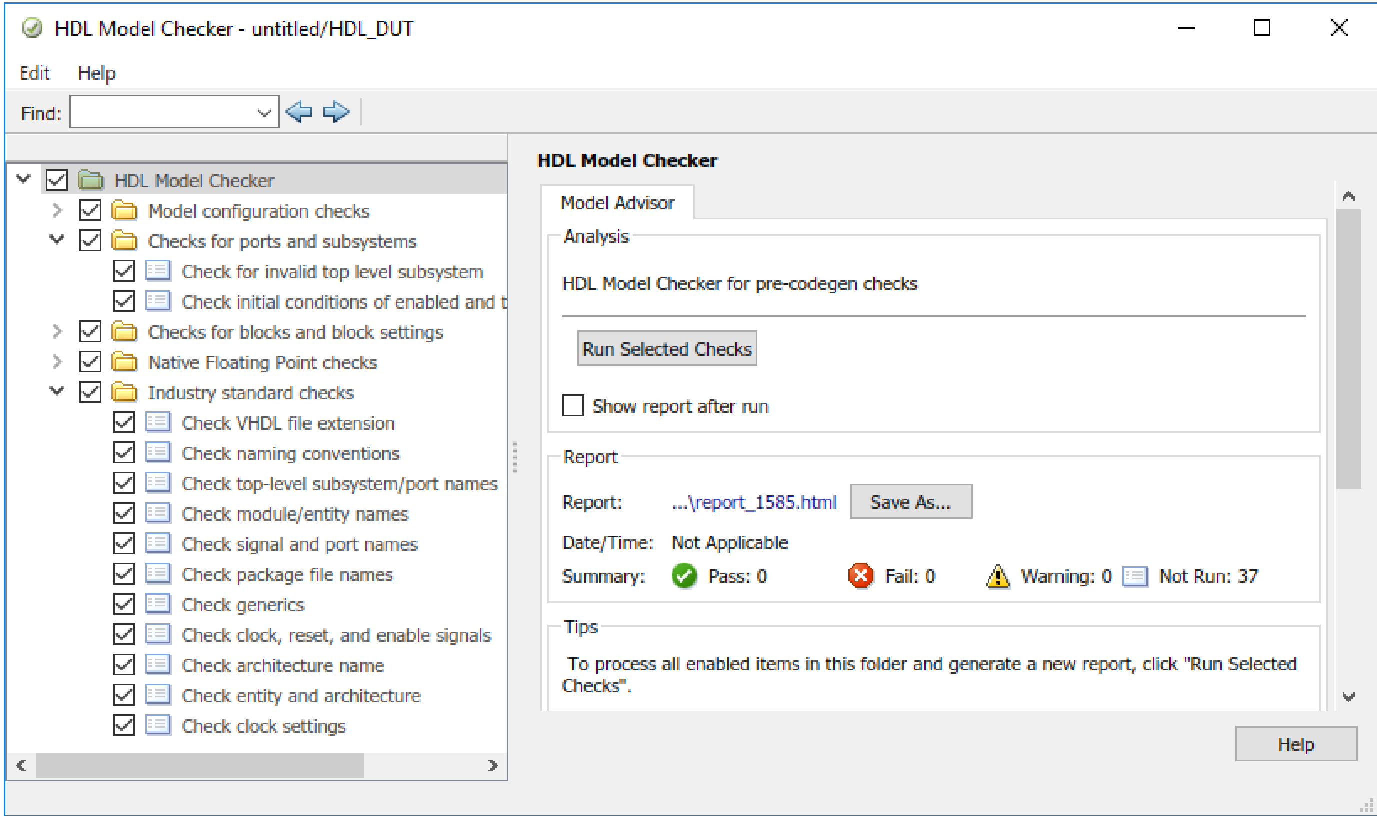
To see whether your DUT Subsystem is compatible for HDL code generation, run the checks in the HDL Code Advisor or the Simulink Model Advisor checks for **HDL Coder**.

To open the HDL Code Advisor:

- From the UI, in the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the DUT Subsystem and then click **HDL Code Advisor**.
- To run the model checks for the Subsystem you want to analyze, right-click that Subsystem, and in the context menu, select **HDL Code** > **HDL Code Advisor**.
- At the command line, use the `hdlcodeadvisor` function:

  ```
  hdlcodeadvisor(gcb)
  ```

When you run this command, the HDL Code Advisor appears.



You may not have to run all checks in the HDL Code Advisor. For example, if your model does not have single or double data types, you do not have to run the checks in the **Native Floating Point checks** folder. To learn more about each check and whether to run the check for your model, right-click that check and select **What's This?**.

See Also

Functions
`hdllib` | `checkhdl` | `hdlcodeadvisor`

Modeling Guidelines

"Basic Guidelines for Modeling HDL Algorithm in Simulink" on page 18-14

More About

- "Use Simulink Templates for HDL Code Generation" on page 14-8
- "Create HDL-Compatible Simulink Model"
- "Display Blocks for HDL Code Generation in Library Browser" on page 23-31

Modeling with Simulink, Stateflow, and MATLAB Function Blocks

In this section...
"Guideline ID" on page 18-24
"Severity" on page 18-24
"Description" on page 18-24

You can follow this guideline as a general practice for modeling your design with various blocks in the Simulink Library Browser.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see "HDL Modeling Guidelines Severity Levels" on page 18-3.

Guideline ID

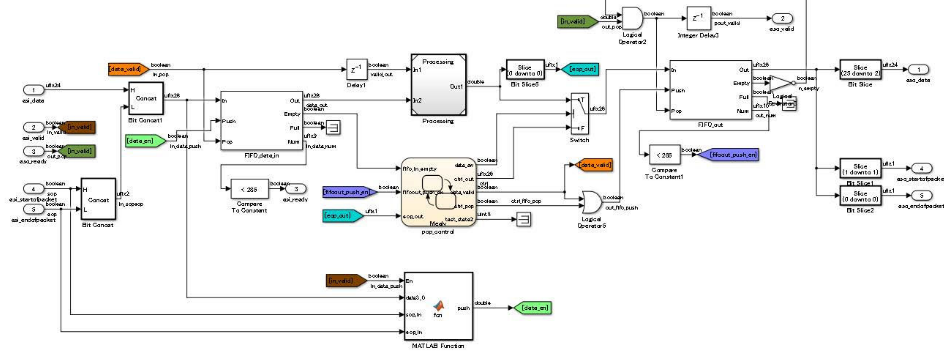
1.1.8

Severity

Informative

Description

When you create a Simulink model for HDL code generation, use Simulink blocks, MATLAB Function blocks, and Stateflow blocks based on the application. This figure shows an example of how you can use the various blocks inside your DUT.



Simulink Blocks

Use Simulink blocks to model arithmetic algorithms that perform numerical processing or contains feedback loops.

MATLAB Function Blocks

Use MATLAB Function blocks to model the control logic, conditional branches such as if-else statements, and simple state machines. You can also use MATLAB Function blocks to model an IP that is written using MATLAB code.

Stateflow Blocks

Use these Stateflow blocks to model your algorithm:

- State Transition Table: Use these blocks to model state machines that control the output using knowledge of the past and the present.
- Chart: Use these blocks to model flow charts using conditional if-else branches and state machines that control the output using knowledge of the past and the present.
- Truth Table: Use these blocks to model conditional if-else branches.

You can model combinational logic using Stateflow blocks. For more complex operations and operations that change timing such as pipeline insertion and processing, use Simulink blocks. You can then use the Stateflow logic to process the result calculated from the Simulink blocks

Model References

For significantly large algorithms that have complex computations, you can partition the design into a hierarchy of smaller designs. Use this partitioning for reuse, modular development, and accelerated simulation. You can reuse models by including them as Model blocks inside a top model. The model that reuses this block is called the top model and the block that is reused or included in the top model is called the referenced model.

Note When you generate HDL code for a Subsystem that is not at the top level of the model, HDL Coder converts the Subsystem to a model reference.

A referenced model is treated similar to an Atomic Subsystem. In some cases, an algebraic loop can potentially occur, and can prevent HDL code generation. To generate code, either remove the algebraic loop in your design, or, in the Configuration Parameters dialog box, specify the **Minimize algebraic loop occurrences** setting.

BlackBox Subsystems

For subsystems that you want to simulate in your design and to include the HDL code that you authored, use BlackBox subsystems. To create a **BlackBox** Subsystem, set the HDL Architecture of a Subsystem or Model reference to `BlackBox`. You can use this architecture to incorporate handwritten HDL code into a Simulink model. For more information, see “Verify the Combination of Hand-Written and Generated HDL Code” (HDL Verifier).

If you generate a Simulink model using the HDL code that you authored, use HDL import. To learn more, see “Import Verilog Code and Generate Simulink Model” on page 14-165.

HDL Cosimulation Blocks

If you have a HDL simulator such as Mentor Graphics ModelSim or Cadence Incisive®, you can use HDL Cosimulation blocks to simulate the HDL code for the DUT and instantiate this HDL code in the generated code.

See Also

Modeling Guidelines
"Basic Guidelines for Modeling HDL Algorithm in Simulink" on page 18-14

More About

- "Verify with HDL Cosimulation" on page 36-11
- "Display Blocks for HDL Code Generation in Library Browser" on page 23-31
- "Design Guidelines for the MATLAB Function Block" on page 27-19
- "Introduction to Stateflow HDL Code Generation" on page 26-2
- "Generate Black Box Interface for Subsystem" on page 25-4
- "Generate Black Box Interface for Referenced Model" on page 25-8

Terminate Unconnected Block Outputs and Usage of Commenting Blocks

You can follow these guidelines as recommended modeling practices such as making sure that block outputs are terminated and how you can comment out blocks for HDL code generation.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Terminate Unconnected Block Outputs

Guideline ID

1.1.9

Severity

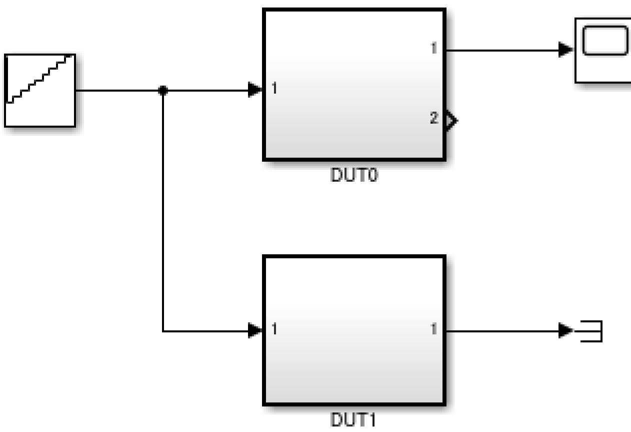
Mandatory

Description

If you generate HDL code for a Subsystem that has unconnected output ports, HDL Coder™ generates an error. For output ports that are not connected to downstream logic, connect them to a Terminator block.

This model illustrates a `DUT0` Subsystem that has an unconnected output port `Out2`.

```
open_system('hdlcoder_terminateout')
```



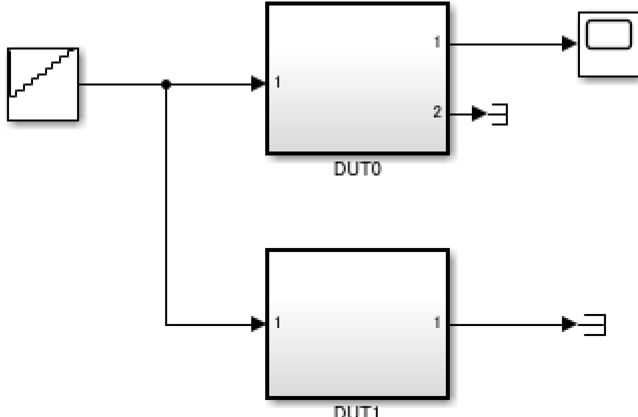
If you generate HDL code for this Subsystem, HDL Coder™ generates this error:

```
error in validation model generation: Failed to find source for outport 2 on
'DUT0' Please create a fully connected subsystem when generating the
cosimulation model.
```

```
close_system('hdlcoder_terminateout')
```

You can use the addterms function to add Terminator blocks to unconnected ports in your model.

```
load_system('hdlcoder_terminateout')
addterms('hdlcoder_terminateout')
open_system('hdlcoder_terminateout')
```



Using Comment Out and Comment Through of Blocks

Guideline ID

1.1.10

Severity

Informative

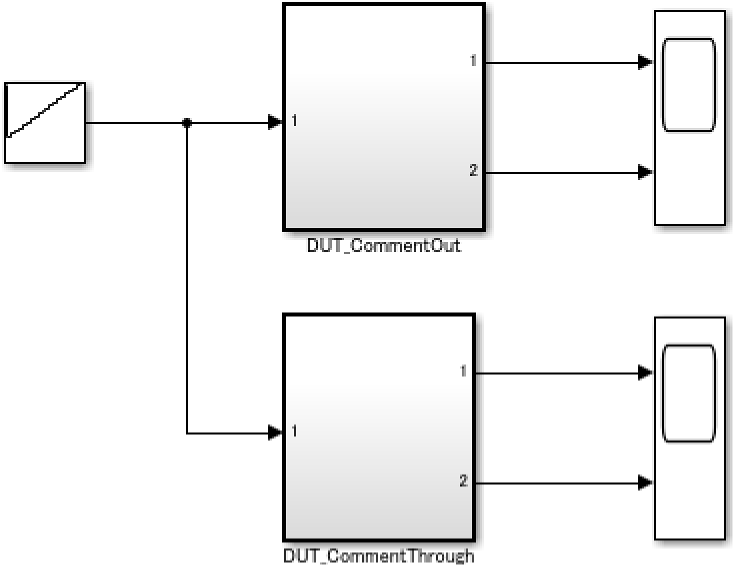
Description

To exclude blocks in your model from simulation without physically removing the blocks from your model, use **Comment Out** or **Comment Through**. When you use **Comment Out**, the signals are terminated and grounded. When you use **Comment Through**, the signals are passed through.

When you generate HDL code, you can use this capability to exclude certain blocks such as blocks that are not supported for HDL code generation.

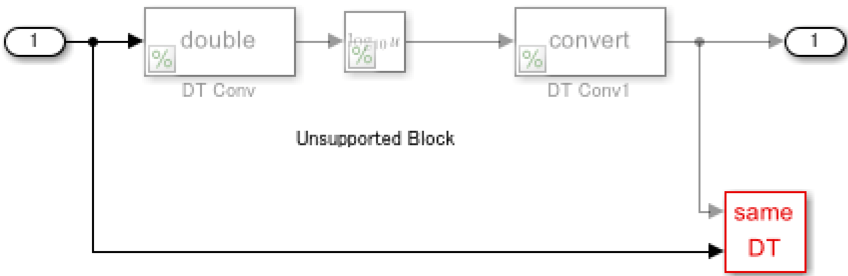
Open the model hdlcoder_comment_through_out.

```
open_system('hdlcoder_comment_through_out')
```



The code generator supports blocks that are comment out when the output signals are unused. The generated code assigns a constant value of `0` to the signal at the output. The `Dut_CommentOut` subsystem contains blocks that are commented out.

```
open_system('hdlcoder_comment_through_out/DUT_CommentOut/Generated_CommentOut')
```

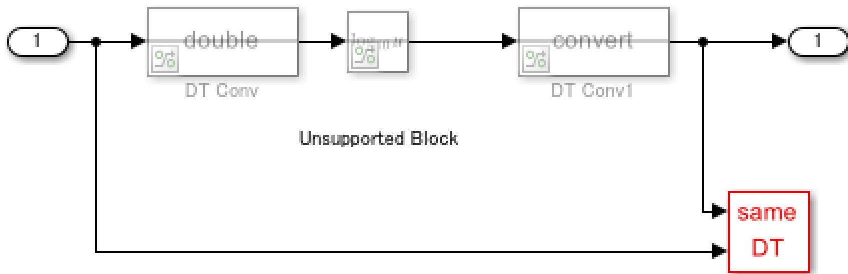


When you generate code, this VHDL® code generated for the `DUT_CommentOut` subsystem indicates a constant zero value assigned to `Out1`.

```
ARCHITECTURE rtl OF Generated IS

  -- Signals
  SIGNAL TmpGroundAtData_Type_DuplicateInport1_out1 : signed(15 DOWNTO 0);  -- sfix16_En6

BEGIN
-- Unsupported Block

TmpGroundAtData_Type_DuplicateInport1_out1 <= to_signed(16#0000#, 16);

Out1 <= std_logic_vector(TmpGroundAtData_Type_DuplicateInport1_out1);

END rtl;
```

The code generator supports blocks that are comment through. The generated code passes the input signal through to the output. The Dut_CommentThrough subsystem contains blocks that are comment through.

```
open_system('hdlcoder_comment_through_out/DUT_CommentThrough/Generated_CommentThrough')
```



When you generate code for Dut_CommentThrough subsystem, the VHDL code shows In1 passed through to Out1.

```
ARCHITECTURE rtl OF Generated_CommentThrough IS

BEGIN
-- Unsupported Block

Out1 <= In1;

END rtl;
```

See Also

Modeling Guidelines
"Basic Guidelines for Modeling HDL Algorithm in Simulink" on page 18-14

Identify and Programmatically Change and Display HDL Block Parameters

You can follow these guidelines to learn how you can identify block parameters in your design and programmatically update some of the parameters so that the model is compatible for HDL code generation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Adjust Sizes of Constant and Gain Blocks for Identifying Parameters

Guideline ID

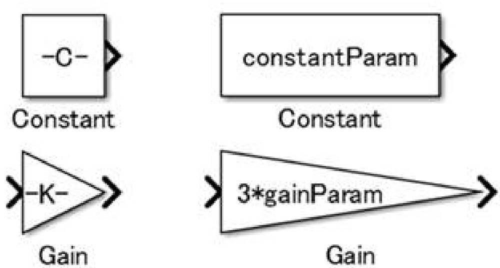
`1.1.11`

Severity

Recommended

Description

For Constant blocks and Gain blocks that have significantly large values or use parameter values, the **Constant** or **Gain** values may not be visible in the block mask. To increase readability, adjust the size of the block so that the parameter value can be displayed as shown in figure.



Display Parameters That Affect HDL Code Generation

Guideline ID

`1.1.12`

Severity

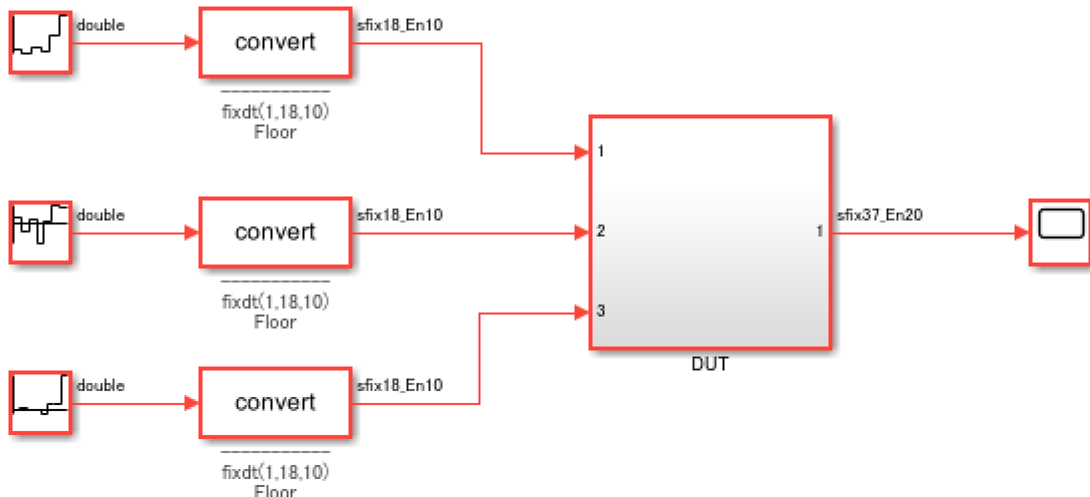
Recommended

Description

Certain HDL block properties such as `DistributedPipelining` and `SharingFactor` can significantly affect HDL code generation. If the block properties are enabled for a certain block or Subsystem, it is recommended that you annotate the block properties beside that block in the Simulink® diagram. When you annotate the model, use delimiters such as `--HDL--` to separate the annotation from the block name.

For example, open the model `hdlcoder_block_annotation_HDL_params.slx`.

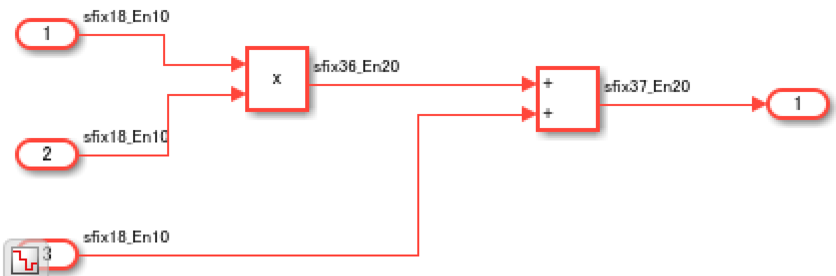
```
open_system('hdlcoder_block_annotation_HDL_params')
set_param('hdlcoder_block_annotation_HDL_params','SimulationCommand','Update')
```



Copyright 2018-2021 The MathWorks, Inc.

The DUT Subsystem performs a simple multiply-add operation.

```
open_system('hdlcoder_block_annotation_HDL_params/DUT')
```



There are HDL model and block parameters saved on the model. To see the non-default parameters, use the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_block_annotation_HDL_params/DUT')
```

```
%% Set Model 'hdlcoder_block_annotation_HDL_params' HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MaskParameterAsGeneric', 'on');
```

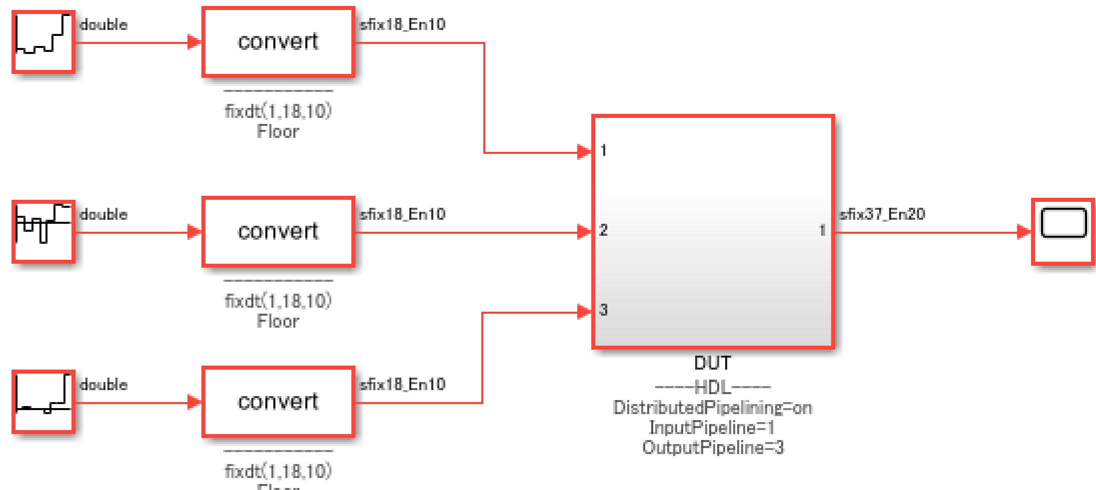
```
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MinimizeClockEnables', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MinimizeIntermediateSignals', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'TargetLanguage', 'Verilog');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'DistributedPipelining', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'InputPipeline', 1);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'OutputPipeline', 3);

% Set Sum HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Add', 'InputPipeline', 1);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Add', 'OutputPipeline', 1);

% Set Product HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Product', 'InputPipeline', 2);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Product', 'OutputPipeline', 1);
```

To annotate the blocks in the model with the non-default HDL block parameters saved and to display the non-default HDL model and block properties in the MATLAB® command window, use the `showHdlParams` script attached with the example.

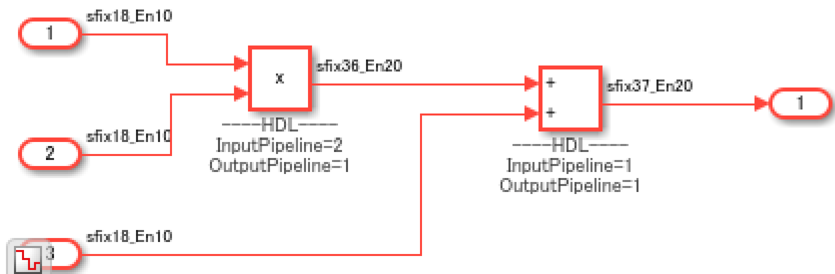
```
showHdlParams('hdlcoder_block_annotation_HDL_params/DUT','on')
```

```
%% Set Model 'hdlcoder_block_annotation_HDL_params' HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MaskParameterAsGeneric', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MinimizeClockEnables', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MinimizeIntermediateSignals', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'TargetLanguage', 'Verilog');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'DistributedPipelining', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'InputPipeline', 1);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'OutputPipeline', 3);

% Set Sum HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Add', 'InputPipeline', 1);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Add', 'OutputPipeline', 1);

% Set Product HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Product', 'InputPipeline', 2);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Product', 'OutputPipeline', 1);
```

```
open_system('hdlcoder_block_annotation_HDL_params')
```




```
open_system('hdlcoder_block_annotation_HDL_params/DUT')
```



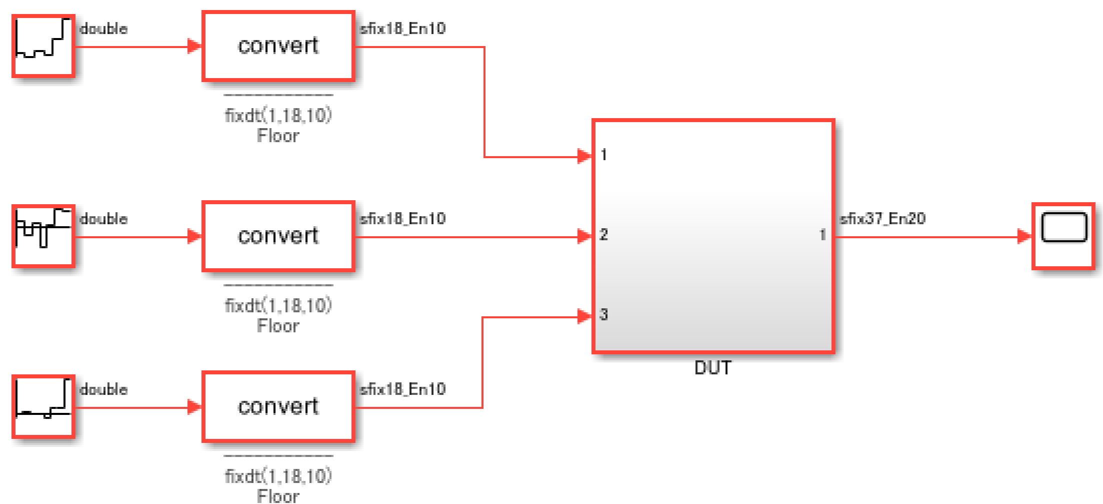
To remove the HDL block parameters annotation from the model, run the `showHdlParams` set to `off`.

```
showHdlParams('hdlcoder_block_annotation_HDL_params/DUT','off')
```

```
%% Set Model 'hdlcoder_block_annotation_HDL_params' HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MaskParameterAsGeneric', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MinimizeClockEnables', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MinimizeIntermediateSignals', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'TargetLanguage', 'Verilog');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'Traceability', 'on');

% Set SubSystem HDL parameters
```

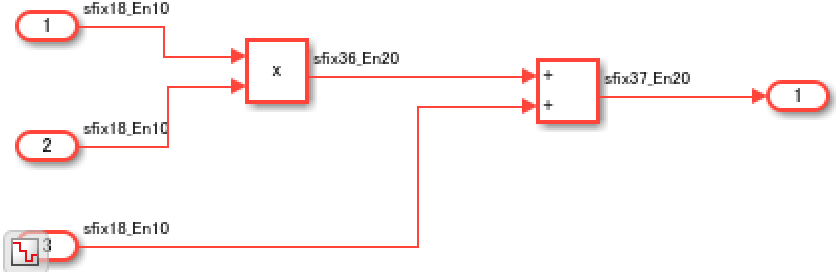
```
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'DistributedPipelining', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'InputPipeline', 1);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'OutputPipeline', 3);

% Set Sum HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Add', 'InputPipeline', 1);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Add', 'OutputPipeline', 1);

% Set Product HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Product', 'InputPipeline', 2);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Product', 'OutputPipeline', 1);
```

```
open_system('hdlcoder_block_annotation_HDL_params')
```



Copyright 2018-2021 The MathWorks, Inc.

```
open_system('hdlcoder_block_annotation_HDL_params/DUT')
```



Change Block Parameters by Using find_system and set_param

Guideline ID

1.1.13

Severity

Informative

Description

To modify the parameters of certain blocks, you can use the function `find_system` with the function `set_param`. For example, this script that detects all Constant blocks with a **Sample time** of `inf` and modifies it to `-1`:

```
modelname = ‘sfir_fixed’;
open_system (modelname)

% Detect all Constant blocks in the model
blockConstant = find_system(bdroot, 'blocktype', 'Constant')

% Detect the Constant blocks with sample time [inf], and change to [-1]
for  n  =  1:numel(blockConstant)
        sTime = get_param(blockConstant{n},'SampleTime')
        if strcmp(lower(sTime), 'inf')
            set_param(blockConstant{n}, 'SampleTime', '-1')
    end
end
```

See Also

Functions
`find_system` | `set_param` | `hdlsaveparams`

More About

- “Specify Block Properties”

DUT Subsystem Guidelines

You can follow these guidelines to learn some best practices on how you can model the DUT for HDL code and testbench generation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

DUT Subsystem Considerations

Guideline ID

`1.2.1`

Severity

Strongly Recommended

Description

The DUT is the Subsystem that contains the algorithm for which you want to generate code. Generally, you specify the top-level Subsystem as the DUT. See also “Partition Model into DUT and Test Bench” on page 18-15.

Consider using these recommended settings when you design the DUT Subsystem for HDL code generation.

- Make sure that the DUT is not a conditionally-executed subsystem, such as an Enabled Subsystem or a Triggered Subsystem. To verify that you are using a valid top-level Subsystem as the DUT, you can run this HDL model check “Check for invalid top level subsystem” on page 37-13.
- Make sure that the **HDL Architecture** of the DUT is not specified as a `BlackBox`. See “BlackBox Subsystems” on page 18-25.
- Connect output signals that are unconnected to a Terminator block. To learn more, see “Terminate Unconnected Block Outputs” on page 18-27.
- For a nontop DUT, specify the DUT as a nonvirtual Subsystem before generating HDL code to avoid numerical mismatches in the simulation results. To learn more, see “Usage of Different Subsystem Types” on page 18-93.

Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks

Guideline ID

`1.2.2`

Severity

Strongly Recommended

Description

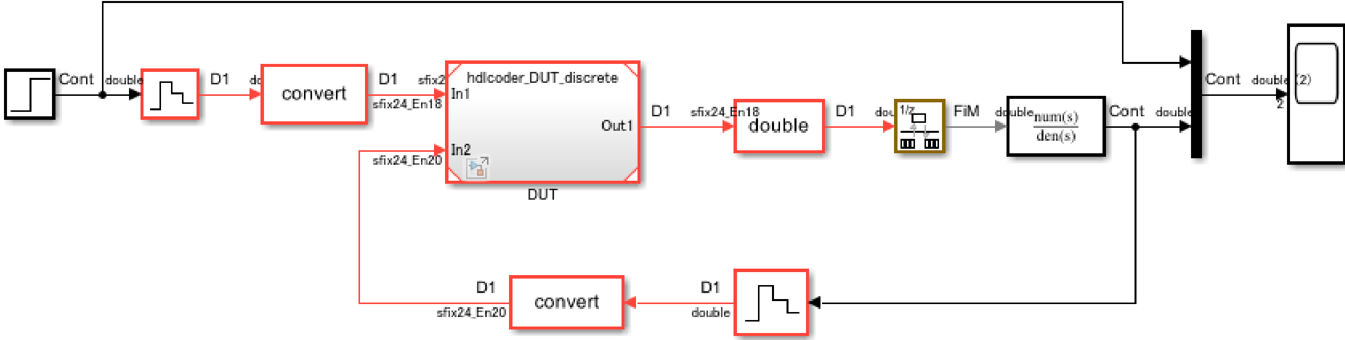
In some cases, parts of the Simulink™ testbench can contain Simscape™ blocks or other blocks from the Simulink library that operate at a continuous sample time. To simulate these blocks, you must

specify a continuous solver setting for your model. The solver settings that you specify applies to all blocks in your model. This means that the DUT Subsystem uses a continuous solver, which is not supported for HDL code generation. To generate HDL code, convert the DUT Subsystem to a model reference, and then use a fixed-step discrete solver for the referenced model. As the parent model and the referenced model use different solver settings, you must convert the sample time by inserting Zero-Order Hold and Rate Transition blocks at the DUT boundary.

For example, open the model `hdlcoder_testbench_continuous.slx`. The model uses `ode45`, which is a continuous solver setting. You see that the DUT is a Model block. Zero-Order Hold and Rate Transition blocks at the boundary convert the sample time.

```
open_system('hdlcoder_testbench_continuous')
set_param('hdlcoder_testbench_continuous','SimulationCommand','Update')
get_param('hdlcoder_testbench_continuous', 'Solver')
```

```
ans =

    'ode45'
```

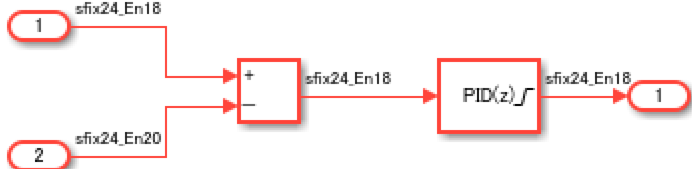


Copyright 2018-2021 The MathWorks, Inc.

To see the referenced model `hdlcoder_DUT_discrete`, double-click the `DUT` block. You see that the DUT uses a discrete solver setting.

```
open_system('hdlcoder_testbench_continuous/DUT')
get_param('hdlcoder_DUT_discrete', 'Solver')
```

```
ans =

    'FixedStepDiscrete'
```



Copyright 2018-2021 The MathWorks, Inc.

Insert Handwritten Code into Simulink Modeling Environment

Guideline ID

`1.2.3`

Severity

Informative

Description

You can reuse a pre-verified RTL IP or insert your handwritten HDL code into the Simulink modeling environment by using these methods:

- **Verilog HDL Import**

  If you have handwritten Verilog code, you can import the code into the Simulink environment. The import process generates a Simulink model that is functionally equivalent to your handwritten HDL code.

  HDL import supports a subset of Verilog constructs that you can use for importing your design to create the Simulink model. To learn more, see:

  - “Supported Verilog Constructs for HDL Import” on page 14-169
  - “Limitations of Verilog HDL Import” on page 14-167
- **BlackBox Subsystems**

  You can use BlackBox subsystems to insert your handwritten HDL code for a block in your Simulink model. You can then integrate BlackBox subsystems with other blocks in your Simulink model and then generate HDL code.

  To make the BlackBox Subsystem compatible with other blocks for HDL code generation and to include this block in your model, create the block in Simulink:

  - Name the block by using the same name as the VHDL entity, or Verilog or SystemVerilog module.
  - Define the same inputs and outputs, including the same types, sizes, and names.
  - Define the same clock, reset, and clock enable signals. A single block can have not more than one clock, reset, and clock enable signal.
  - Use a single sample rate for the block.
  - Specify the **Architecture** of the block as `BlackBox` in the HDL Block Properties.

  To learn more, see "Generate Black Box Interface for Subsystem" on page 25-4.
- **DocBlock in BlackBox Subsystems**

  To keep the HDL code with your model, instead of as a separate file, use a DocBlock to integrate custom HDL code. You can use your own handwritten VHDL, Verilog or SystemVerilog code as the text in the DocBlock.

  You include each DocBlock that contains custom HDL code by placing it in a black box subsystem, and including the black box subsystem in your DUT. One HDL file is generated per black box subsystem. For more information, see "Integrate Custom HDL Code by Using DocBlock" on page 25-10.
- **HDL Cosimulation Blocks**

  If you have a HDL simulator such as Mentor Graphics ModelSim or Cadence Incisive, you can use HDL Cosimulation blocks to simulate the HDL code for the DUT by using that HDL simulator.

  You can simulate the HDL code for the DUT in Simulink and instantiate the HDL code in the generated code for the DUT.

See Also

Functions
`importhdl` | `makehdl` | `makehdltb`

More About

- "Model Referencing for HDL Code Generation" on page 25-2
- "Customize Black Box or HDL Cosimulation Interface" on page 25-12
- "Generate a Cosimulation Model" on page 25-43

Hierarchical Modeling Guidelines

Consider using these recommended settings when you build your model hierarchically and generate HDL code for your design. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Avoid Constant Block Connections to Subsystem Port Boundaries

Guideline ID

1.2.4

Severity

Mandatory

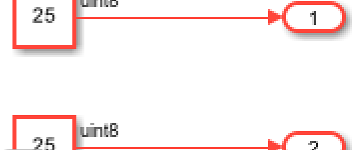
Description

It is recommended that you avoid directly connecting Constant blocks to the output ports of a Subsystem. Synthesis tools may optimize and remove the constants and create unconnected ports.

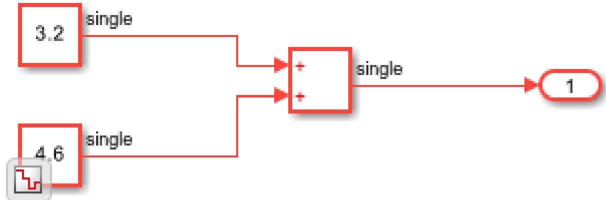
If you use floating-point data types with the `Native Floating Point` mode enabled, and input constant values to an arithmetic operator such as an Add block, HDL Coder™ replaces the Add block with a Constant block when generating code. This optimization can result in a Constant block directly connected to the output port. Therefore, it is recommended that you avoid such modeling constructs. See also “Simplify Constant Operations and Reduce Design Complexity in HDL Coder” on page 21-18.

For example, open the model `hdlcoder_constant_subsystem_boundary.slx`. The DUT contains two subsystems `Constant_subsys1` and `Constant_subsys2`, the outputs of which are inputs to a third Subsystem. `Constant_subsys1` contains Constant blocks directly connected to the output ports, and `Constant_subsys2` contains Constant blocks that have single data types as inputs to an Add block.

```
load_system('hdlcoder_constant_subsystem_boundary.slx')
set_param('hdlcoder_constant_subsystem_boundary','SimulationCommand','Update')
open_system('hdlcoder_constant_subsystem_boundary/DUT/Constant_subsys1')
```

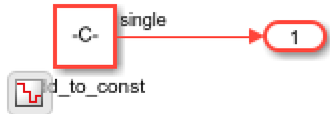



```
open_system('hdlcoder_constant_subsystem_boundary/DUT/Constant_subsys2')
```



As `Constant_subsys2` uses single data types and the model has `Native Floating Point` mode enabled, when you generate HDL code for the DUT, the `Constant_subsys2` becomes a candidate for the optimization that simplifies constant operations. When you open the generated model, you see a Constant block directly connected to the output port.

```
open_system('gm_hdlcoder_constant_subsystem_boundary.slx')
set_param('gm_hdlcoder_constant_subsystem_boundary','SimulationCommand','Update')
open_system('gm_hdlcoder_constant_subsystem_boundary/DUT/Constant_subsys2')
```



Generate Parameterized HDL Code for Constant and Gain Blocks

Guideline ID

1.2.5

Severity

Recommended

Description

To generate parameterized HDL code for Gain and Constant blocks:

- The Subsystem that contains the Gain and Constant blocks must be a masked subsystem. The Gain and Constant blocks use these mask parameter values. You define mask parameters of the Subsystem in the Mask Editor dialog box.
- The Subsystem that contains the Gain and Constant blocks must be an Atomic Subsystem. To make a Subsystem an Atomic Subsystem, right-click that Subsystem and select **Treat as atomic unit**.
- Enable the **Generate parameterized HDL code from masked subsystem** setting in the Configuration Parameters dialog box or set `MaskParameterAsGeneric` to `on` at the command line using `makehdl` or `hdlset_param`. For more information, see Generate parameterized HDL code from masked subsystem.

For an example, open the model `hdlcoder_masked_subsystems`. The `Top` Subsystem contains two atomic masked subsystems `MASKSUB` and `MASKSUB1` that are similar but for the masked parameter values.

```
load_system('hdlcoder_masked_subsystems')
set_param('hdlcoder_masked_subsystems', 'SimulationCommand', 'Update')
open_system('hdlcoder_masked_subsystems/TOP')
```

The model has the `MaskParameterAsGeneric` setting enabled. This setting corresponds to the **Generate parameterized HDL code from masked subsystem** setting that is enabled at the command line.

```
hdlsaveparams('hdlcoder_masked_subsystems')
```

```
%% Set Model 'hdlcoder_masked_subsystems' HDL parameters
hdlset_param('hdlcoder_masked_subsystems', 'HDLSubsystem', 'hdlcoder_masked_subsystems/TOP');
hdlset_param('hdlcoder_masked_subsystems', 'MaskParameterAsGeneric', 'on');
```

To generate VHDL® code for the `Top` Subsystem, run this command:

```
makehdl('hdlcoder_masked_subsystems/TOP')
```

In the generated code, you see that HDL Coder™ generates one HDL file `MaskedSub` with the different masked parameters mapped to generic ports.

```
-- -------------------------------------------------------------
--
-- File Name: hdlsrc\hdlcoder_masked_subsystems\TOP.vhd
-- Created: 2018-10-08 13:30:02
--
-- Generated by MATLAB 9.6 and HDL Coder 3.13
--
--
-- -------------------------------------------------------------
--
--

ARCHITECTURE rtl OF TOP IS

  -- Component Declarations
  COMPONENT MASKSUB
    GENERIC( m                            : integer;
             b                            : integer
             );
    PORT( x                               :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          y                               :   OUT   std_logic_vector(16 DOWNTO 0)  -- ufix17
          );
  END COMPONENT;
```

```
  -- Component Configuration Statements
  FOR ALL : MASKSUB
    USE ENTITY work.MASKSUB(rtl);

  -- Signals
  SIGNAL MASKSUB_out1                     : std_logic_vector(16 DOWNTO 0);  -- ufix17
  SIGNAL MASKSUB1_out1                    : std_logic_vector(16 DOWNTO 0);  -- ufix17

BEGIN
  u_MASKSUB : MASKSUB
    GENERIC MAP( m => 5,
                 b => 2
                 )
    PORT MAP( x => In1,  -- uint8
              y => MASKSUB_out1  -- ufix17
              );

  u_MASKSUB1 : MASKSUB
    GENERIC MAP( m => 6,
                 b => 4
                 )
    PORT MAP( x => In1,  -- uint8
              y => MASKSUB1_out1  -- ufix17
              );

  Out1 <= MASKSUB_out1;

  Out2 <= MASKSUB1_out1;

END rtl;
```

Place Physical Signal Lines Inside a Subsystem

Guideline ID

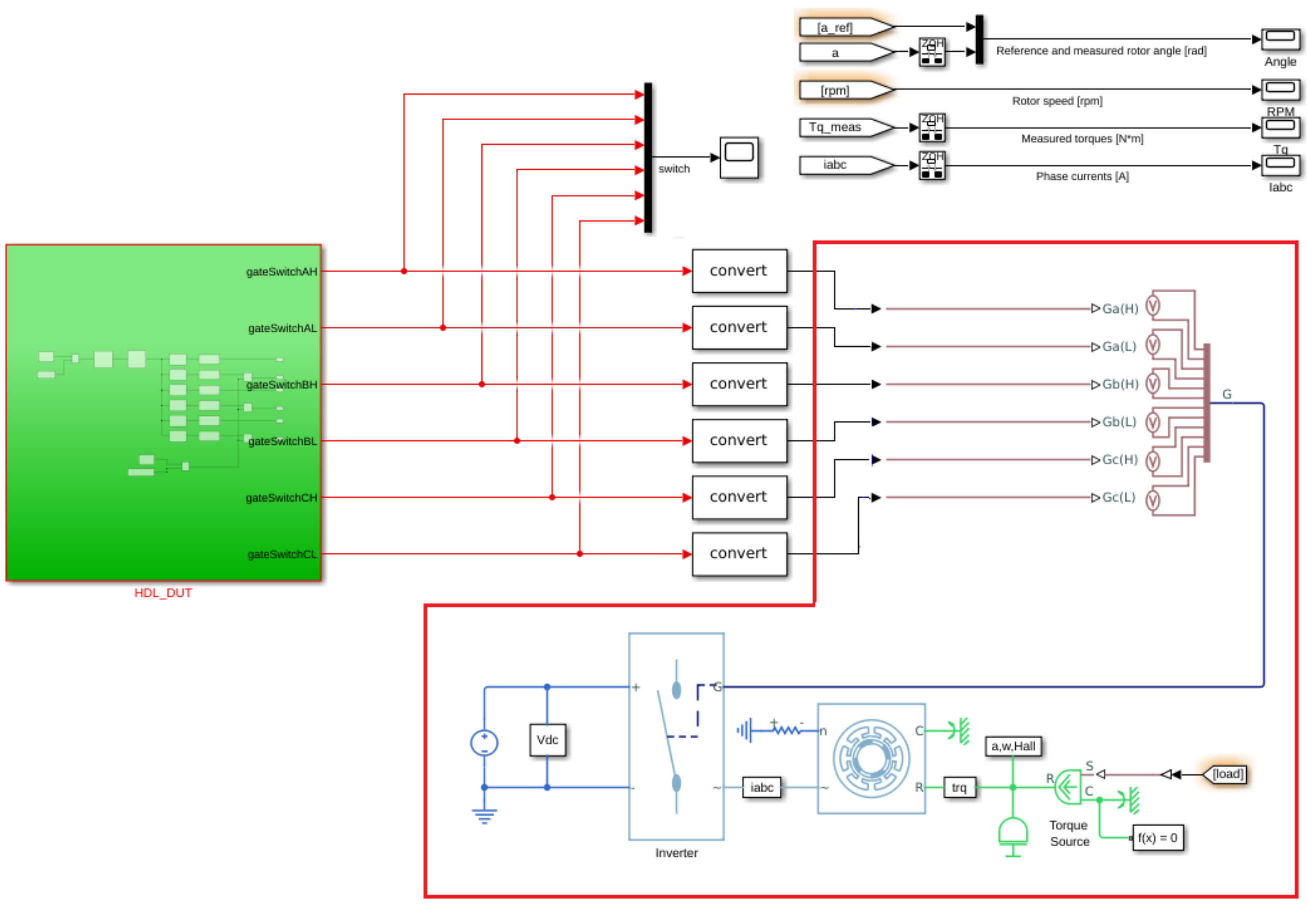
1.2.6

Severity

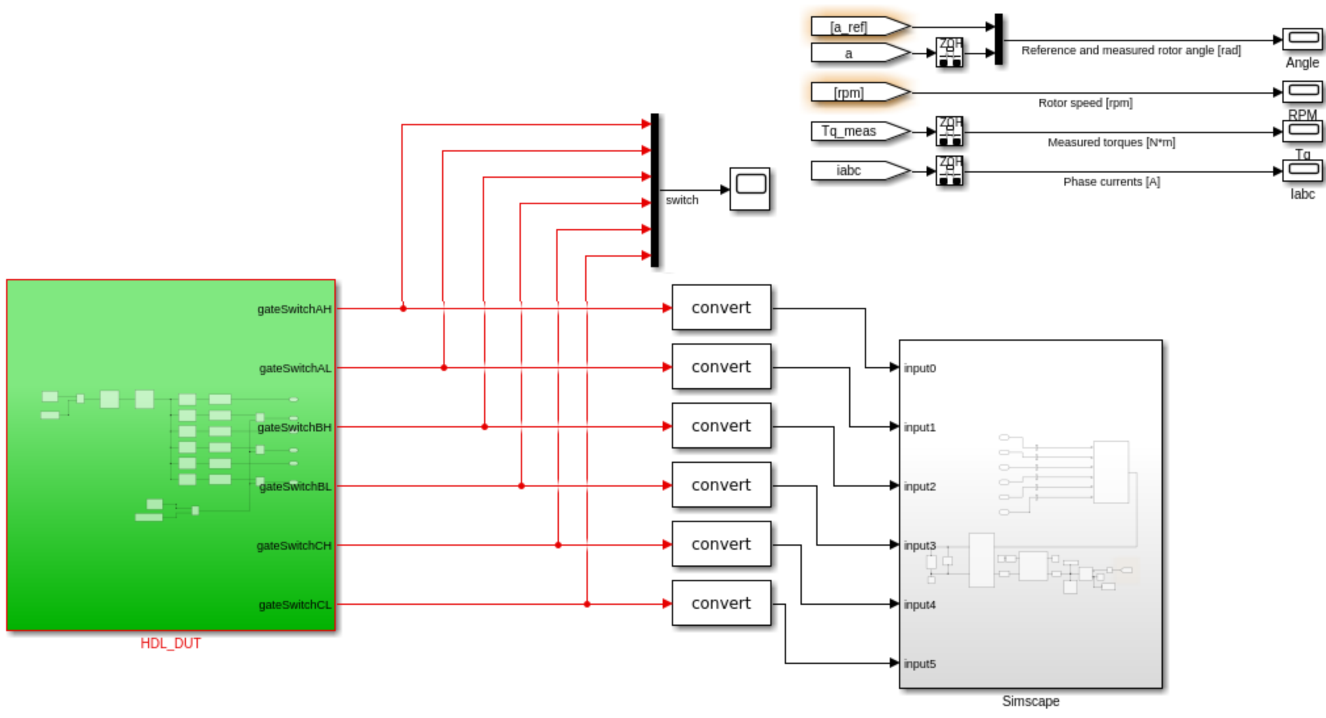
Mandatory

Description

To avoid errors when generating HDL test bench, physical signal lines that are present at the same level as the DUT subsystem must be placed inside a Subsystem block. For example, consider this Simulink model that has physical signal lines outside the DUT subsystem, `HDL_DUT`.



Place the physical signal lines and the blocks connected to it that are highlighted inside a Subsystem block. You can then generate HDL code and test bench for the DUT subsystem.



See Also

Functions

makehdl | makehdltb

More About

- “Generate Parameterized Code for Referenced Models” on page 14-21
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder” on page 21-18
- “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 21-224

Design Considerations for Matrices and Vectors

These guidelines recommend how you can use matrix and vector signals when modeling your design for HDL code generation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Modeling Requirements for Matrices

Guideline ID

`1.3.1`

Severity

Mandatory

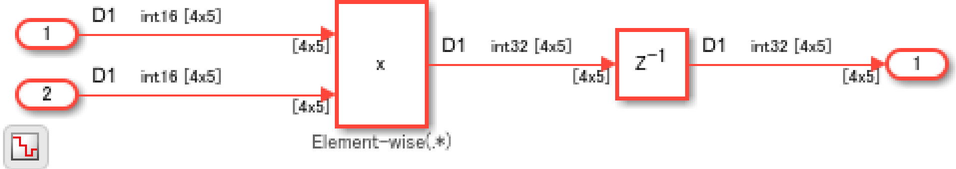
Description

HDL Coder™ support matrix data types at the DUT interfaces. You can use 2D or 3D matrices as a input to the DUT interface and generate HDL code for your model.

Example

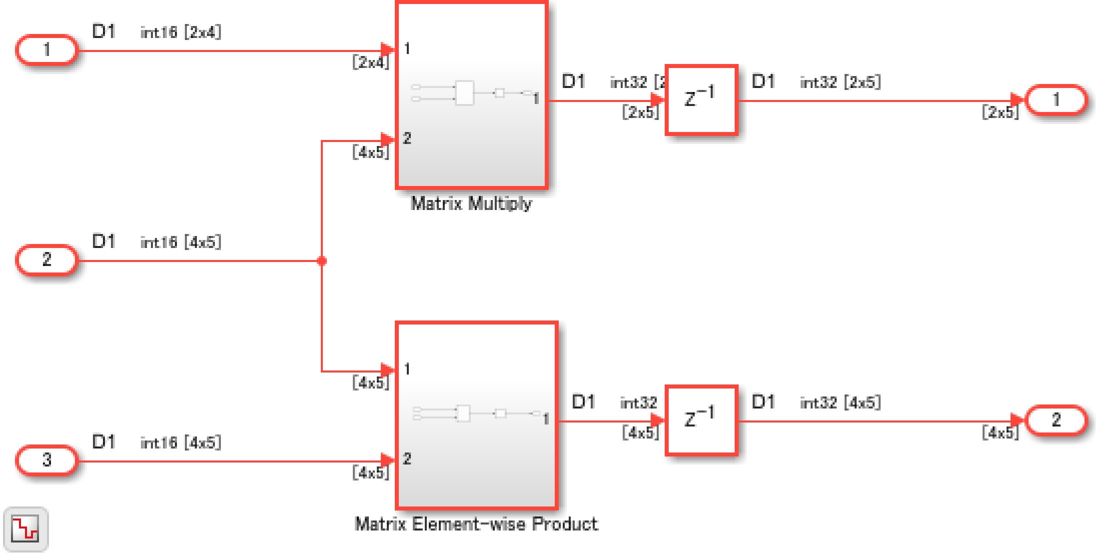
This example shows how to use matrix types in HDL Coder™. Open this model `hdlcoder_matrix_multiply`.

```
open_system('hdlcoder_matrix_multiply')
set_param('hdlcoder_matrix_multiply', 'SimulationCommand', 'update')
sim('hdlcoder_matrix_multiply')
```



If you open the DUT Subsystem, you see two subsystems. One subsystem uses a Matrix Multiply block and the other subsystem performs element-wise multiplication.

```
open_system('hdlcoder_matrix_multiply/DUT')
```



If you generate HDL code for the DUT Subsystem and open the generated model, you see how the multiplication operation is performed.

Modeling Considerations

- When you use the Product block, use the right **Multiplication** mode. By using this mode, you can perform either matrix multiplication or element-wise multiplication. The multiplied output can have different dimensions depending on the **Multiplication** mode.
- When you use the Product block to perform matrix multiplication, place the Matrix Multiply block inside a Subsystem block. When you generate code and open the generated model, you see that HDL Coder expands the matrix multiplication to multiple Product and Add blocks. Placing the Matrix Multiply block inside a subsystem makes the generated model easier to understand. In addition, make sure that you do not provide more than two inputs to the Matrix Multiply block.

Limitations

HDL code generation does not support matrix inputs in these HDL workflows:

- Cosimulation model generation
- FPGA-in-the-Loop
- IP Core Generation

Avoid Generating Ascending Bit Order in HDL Code From Vector Signals

Guideline ID

1.3.2

Severity

Strongly Recommended

Description

In MATLAB, the default bit ordering for arrays is ascending. The generated VHDL code in such cases uses a declaration of `std_logic_vector (0 to n)`. This signal declaration generates warnings by violating certain HDL coding standard rules. These are some scenarios:

Ascending Bit Order Scenarios

Scenario	Problem Example	Workaround

Delay block with a **Delay length** greater than 1.	This example illustrates the generated code for a Delay block with a **Delay length** of 5.
	
	ENTITY Subsystem1 IS
	  PORT(clk : IN std_logic;
	        reset : IN std_logic;
	        enb : IN std_logic;
	        In1 : IN std_logic; -- ufix1
	        Out1 : OUT std_logic -- ufix1
	        );
	END Subsystem1;
	
	ARCHITECTURE rtl OF Subsystem1 IS
	
	  -- Signals
	  SIGNAL Delay_reg : std_logic_vector(0 TO 4); -- ufix1 [5]
	  SIGNAL Delay_out1 : std_logic; -- ufix1
		Instead of using a Delay block with a **Delay length** of 5, you can connect five Delay blocks that have a **Delay length** of 1 in series.
		
		ENTITY Subsystem1 IS
		  PORT(clk : IN std_logic;
		        reset : IN std_logic;
		        enb : IN std_logic;
		        In1 : IN std_logic; -- ufix1
		        Out1 : OUT std_logic -- ufix1
		        );
		END Subsystem1;
		
		ARCHITECTURE rtl OF Subsystem1 IS
		
		  -- Signals
		  SIGNAL Delay_out1 : std_logic; -- ufix1
		  SIGNAL Delay1_out1 : std_logic; -- ufix1
		  SIGNAL Delay2_out1 : std_logic; -- ufix1
		  SIGNAL Delay3_out1 : std_logic; -- ufix1
		  SIGNAL Delay4_out1 : std_logic; -- ufix1

Combining multiple input signals to a vector signal using the Mux block.	This example illustrates the generated code when you use a Mux block to combine 4 input signals.
	
	ENTITY Subsystem IS
	  PORT(In1 : IN std_logic; -- ufix1
	        Out1 : OUT std_logic_vector(0 TO 3); -- ufix1 [4]
	        );
	END Subsystem;
	
	ARCHITECTURE rtl OF Subsystem IS
	
	  -- Signals
	  SIGNAL Mux_out1 : std_logic_vector(0 TO 3); -- ufix1 [4]
		Use a Bit Concat block to combine the input signals. This example illustrates the generated code for this block by concatenating 4 input signals.
		
		ENTITY Subsystem IS
		  PORT(In1 : IN std_logic; -- ufix1
		        Out1 : OUT std_logic_vector(3 DOWNTO 0) -
		        );
		END Subsystem;
		
		ARCHITECTURE rtl OF Subsystem IS
		
		  -- Signals
		  SIGNAL Bit_Concat_out1 : unsigned(3 DOWNTO 0); -- u

Scenario	Problem Example	Workaround

Using a Constant block to generate vector signals.	This example illustrates the generated code when you use a Constant block to generate a vector of 4 scalar boolean signals.
	`ENTITY Subsystem2 IS`
	`  PORT( Out1  :  OUT  std_logic_vector(0 TO 3)  -- boolean [4]`
	`        );`
	`END Subsystem;`
	
	`ARCHITECTURE rtl OF Subsystem2 IS`
	
	`  -- Signals`
	`  SIGNAL Constant_out1  : std_logic_vector(0 TO 3);  -- boolean [4]`
		Use a Demux block followed by a Bit Concat block after the Constant block. This example illustrates the generated code when you apply this modeling technique to the vector of 4 Constant block.
		`ENTITY Subsystem2 IS`
		`  PORT( Out1   :   OUT   std_logic_vector(3 DOWNTO 0)`
		`        );`
		`END Subsystem2;`
		
		`ARCHITECTURE rtl OF Subsystem2 IS`
		
		`  -- Signals`
		`  SIGNAL Constant_out1     : std_logic_vector(0 TO 3);`
		`  SIGNAL Constant_out1_0   : std_logic;`
		`  SIGNAL Constant_out1_1   : std_logic;`
		`  SIGNAL Constant_out1_2   : std_logic;`
		`  SIGNAL Constant_out1_3   : std_logic;`
		`  SIGNAL Bit_Concat_out1   : unsigned(3 DOWNTO 0);  --`
		`]`

See Also

Functions

`makehdl` | `makehdltb`

More About

- “Signal and Data Type Support” on page 14-3
- “RTL Description Rules and Checks” on page 24-18

Use Bus Signals to Improve Readability of Model and Generate HDL Code

You can follow these guidelines to learn about bus signals, how to model your design by using these signals, and generate HDL code. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see "HDL Modeling Guidelines Severity Levels" on page 18-3.

Guideline ID

`1.3.3`

Severity

Informative

Description

When to Use Buses?

If your DUT or other blocks in your model have many input or output signals, you can create bus signals to improve the readability of your model. A bus signal or bus is a composite signal that consists of other signals that are called elements. The bus signal can have a structure of different data types or a vector signal with the same data types. If all signals have the same data type, you generally use a Mux block. The constituent signals or elements of a bus can be:

- Mixed data type signals such as double, integer, and fixed-point
- Mixed scalar and vector elements
- Mixed real and complex signals
- Other buses nested to any level
- Multidimensional signals

HDL Coder Support for Buses

You can generate HDL code for designs that have:

- DUT subsystem ports connected to buses.
- Simulink® and Stateflow® blocks supported for HDL code generation.

HDL Coder™ supports code generation for bus-capable blocks in the **HDL Coder** block library. Bus-capable blocks are blocks that can accept bus signals as input and produce bus signals as outputs. For a list of bus-capable blocks that Simulink supports, see "Bus-Capable Blocks".

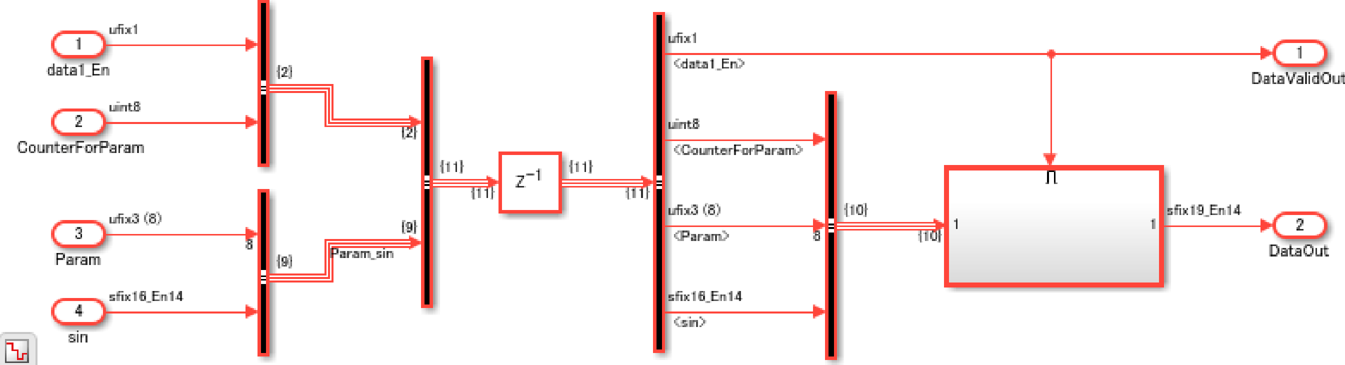
See "Signal and Data Type Support" on page 14-3 for blocks that support HDL code generation with buses.

Create Bus Signals

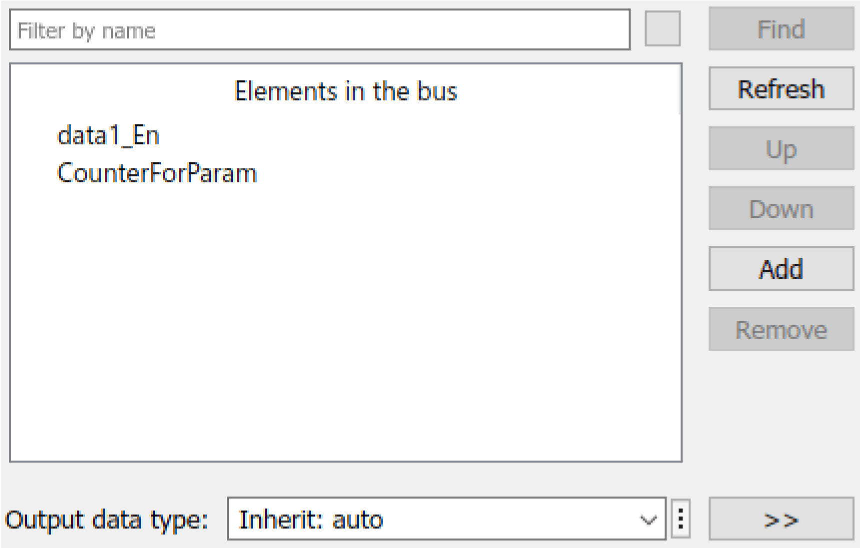
You can create bus signals by using Bus Creator blocks. A Bus Creator block assigns a name to each signal that it creates. You can then refer to signals by name when you search for their sources.

For an example that illustrates how to model with buses, open `hdlcoder_bus_nested.slx`. Double-click the `HDL_DUT` Subsystem.

```
open_system('hdlcoder_bus_nested')
set_param('hdlcoder_bus_nested','SimulationCommand','Update')
open_system('hdlcoder_bus_nested/HDL_DUT')
```



In this model, the Bus Creator blocks create two bus signals. One bus signal contains `data1_En` and `CounterForParam` signals. The other bus signal contains `Param` and `sin` signals. By default, each signal on the bus inherits the name of the signal connected to the bus. This figure shows the **Elements in the bus** block parameter for the Bus Creator block that takes inputs `data1_En` and `CounterForParam`.



Nest Buses

You see another Bus Creator block that combines these two bus signals. When one or more inputs to a Bus Creator block is a bus, the output is a nested bus.

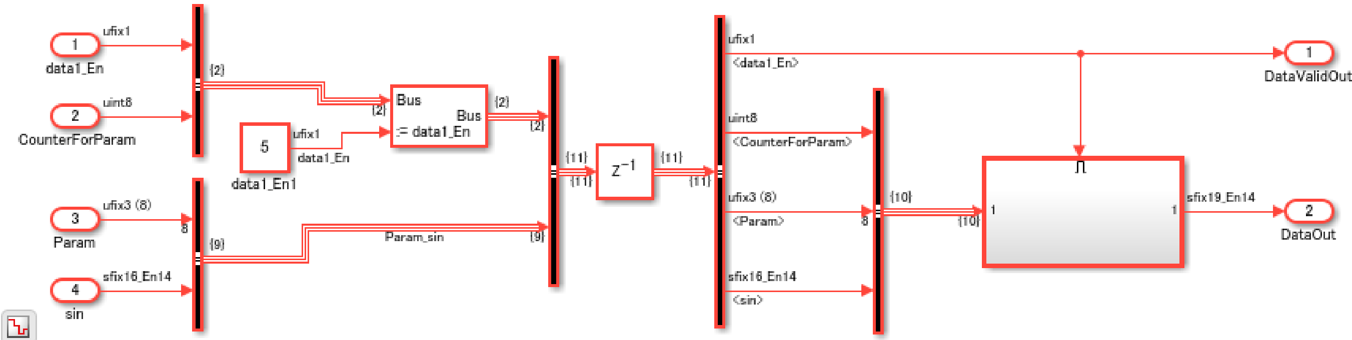
The Bus Creator block generates names for bus signals whose corresponding inputs do not have names. The names are in the form `signaln`, where n is the number of the port the input signal connects to. For example, if you open the Block Parameters dialog box for the second Bus Creator block, you see **Elements in the bus** as `signal1` and `Param_sin`.

Assign Signal Values to Bus

To change bus element values, use a Bus Assignment block. Use a Bus Assignment block to change bus element values without adding Bus Selector and Bus Creator blocks that select bus elements and reassemble them into a bus.

For example, open the model `hdlcoder_bus_nested_assignment`.

```
open_system('hdlcoder_bus_nested_assignment')
set_param('hdlcoder_bus_nested_assignment','SimulationCommand','Update')
open_system('hdlcoder_bus_nested_assignment/HDL_DUT')
```



In the model, you see a Bus Assignment block that assigns the value `5` to the `data1_En` signal in the bus.

Select Bus Outputs

To extract the signals from a bus that includes nested buses, use Bus Selector blocks. By default, the block outputs the specified bus elements as separate signals. You can also output the signals as another bus. You can use the `OutputSignals` block property to see the **Elements in the bus** that the block outputs. By using this property, you can track which signals are entering a Bus Selector block deep within your model hierarchy.

```
get_param('hdlcoder_bus_nested/HDL_DUT/Bus Selector5', 'OutputSignals')
```

```
ans =

    'signal1.data1_En,signal1.CounterForParam,Param_sin.Param,Param_sin.sin'
```

Generate HDL Code

To generate HDL code for this model, run this command:

```
makehdl('hdlcoder_bus_nested/HDL_DUT')
```

You see that the code generator expands the bus signals to scalar signals in the generated code. For example, if you open the generated Verilog file for the `HDL_DUT` Subsystem, for the Delay block that takes the two nested bus signals `signal1` and `Param_sin`, you see four always blocks created for each signal in the bus. For example, you see an always block for the `data1_En` signal that is part of `signal1`. This figure displays the scalar signals created for each bus signal in the module definition.

```
`timescale 1 ns / 1 ns

module HDL_DUT
     (clk, reset, clk_enable,
      data1_En, alphaCounterForParam,
      Param_0, Param_1, Param_2, Param_3,
      Param_4, Param_5, Param_6, Param_7,
      sin, ce_out, DataValidOut, DataOut);


           ......


   assign ce_out = clk_enable;

endmodule  // HDL_DUT
```

Simplify Subsystem Bus Interfaces

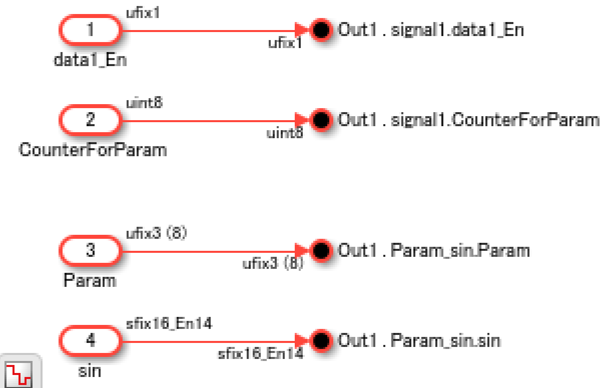
You can simplify the Subsystem bus interfaces by using Bus Element blocks. The In Bus Element and Out Bus Element blocks provide a simplified and flexible way to use bus signals as inputs and outputs to subsystems. The In Bus Element block is equivalent to an Inport block combined with a Bus Selector block. The Out Bus Element block is equivalent to an Outport block combined with a Bus Creator block. To refactor an existing model that uses Inport, Bus Selector, Bus Creator, and Outport blocks to use In Bus Element and Out Bus Element blocks, you can use Simulink Editor action bars.

For example, open the model `hdlcoder_bus_nested_simplified`. This model is functionally equivalent to the `hdlcoder_bus_nested` model but is a more simplified version.

```
open_system('hdlcoder_bus_nested_simplified')
set_param('hdlcoder_bus_nested_simplified','SimulationCommand','Update')
open_system('hdlcoder_bus_nested_simplified/HDL_DUT')
```

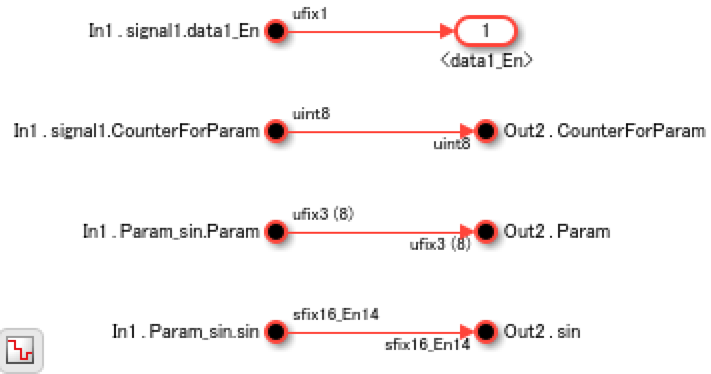
The model has two Subsystems that perform bus creation and bus selection by using Bus Element blocks. The `Bus_Creator_Subsystem` combines the Outport blocks with the Bus Creator blocks to create Out Bus Element blocks.

```
open_system('hdlcoder_bus_nested_simplified/HDL_DUT/Bus_Creator_Subsystem')
```



The `Bus_Selection_Subsystem` combines the Inport blocks with the Bus Selector blocks to create In Bus Element blocks.

```
open_system('hdlcoder_bus_nested_simplified/HDL_DUT/Bus_Selection_Subsystem')
```



To learn more, see “Simplify Subsystem and Model Interfaces with Bus Element Ports”.

Virtual and Nonvirtual Buses

The bus signals in the model `hdlcoder_bus_nested` created earlier by using Bus Creator and Bus Selector blocks are virtual buses. Each bus element signal is stored in memory, but the bus signal is not stored. The bus simplifies the graph but has no functional effect. In the generated HDL code, you see the constituent signals but not the bus signal.

To more easily track the correspondence between a bus signal in the model and the generated HDL code, use nonvirtual buses. Nonvirtual buses generate clean HDL code because it uses a structure to hold the bus signals. To convert a virtual bus to a nonvirtual bus, in the Block Parameters of the Bus Creator blocks, you specify the **Output data type** as `Bus: object_name` by replacing `object_name` with the name of the bus object and then select **Output as nonvirtual bus**.

See “Convert Virtual Bus to Nonvirtual Bus”.

Array of Buses

An array of buses is an array whose elements are buses. Each element in an array of buses must be nonvirtual and must have the same data type.

To learn more about modeling with array of buses, see “Generating HDL Code for Subsystems with Array of Buses” on page 14-22.

See Also

Functions
`makehdl` | `makehdltb`

More About

- “Signal and Data Type Support” on page 14-3
- “Group Nonvirtual Buses in Arrays of Buses”
- “Iteratively Process Nonvirtual Buses with Arrays of Buses”

Guidelines for Clock and Reset Signals

In the Simulink modeling environment, you do not create global signals such as clock, reset, and clock enable. These signals are created when you generate HDL code for your model. You can specify the clock cycle by using the sample time in Simulink.

If your model is single rate, it means all blocks operate at the same sample time. The synthesis tools infer that the registers or Delay blocks you add to your model run at the clock rate. For the synthesis tools, data propagates from the source register to the destination register in one clock cycle.

You can follow these guidelines to learn about generating clock signals in the HDL code. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Use Global Oversampling to Create Frequency-Divided Clock

Guideline ID

`1.4.1`

Severity

Informative

Description

You can set the frequency-divided clock rate used for HDL code generation as a multiple of the Simulink base sample rate. For example, if the Simulink base rate is `1 MHz` and you want the clock frequency of your target hardware to run at `50 MHz`, you can set **Target Frequency** to `50` and enable **Treat Simulink rates as actual hardware rates** to let HDL Coder determine an optimized oversampling value that achieves your desired clock frequency. For more information, see Treat Simulink rates as actual hardware rates. To learn more, see “Generate a Global Oversampling Clock” on page 20-9.

Create Multirate Model with Integer Clock Multiples by Clock Division

Guideline ID

`1.4.2`

Severity

Mandatory

Description

You can generate a multirate model by using clock-rate division or by using clock multiples. For a multirate model, the fastest sample time in your Simulink® model corresponds to the primary clock rate. A timing controller entity is created to control the clocking for blocks operating at slower sample rates. Clock enable signals that have the necessary rate and phase information control the clocking for these blocks in your design.

Multirate models are created when you use certain blocks in your Simulink model, specify certain block architectures, or use operations such as resource sharing. For example, these block-architecture combinations generate a multirate model:

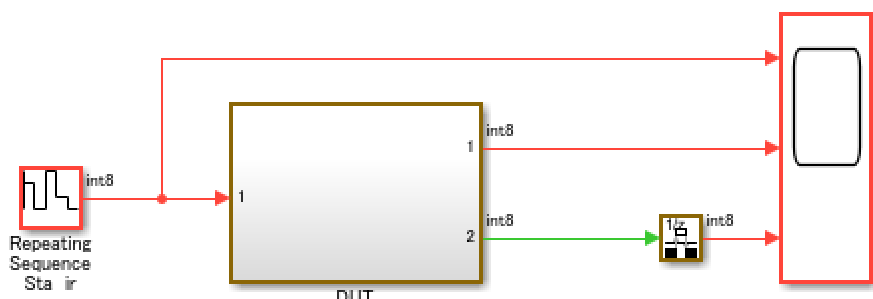
- Divide block with Newton-Raphson implementation.
- Reciprocal block with ReciprocalSqrtBasedNewton implementation.
- Sum of Elements and Product of Elements blocks with Cascade architecture.
- Sqrt with SqrtBasedNewton and Reciprocal Sqrt with ReciprocalSqrtBasedNewton implementation.

In addition, to model multirate designs in Simulink, use these blocks:

- In the **Simulink** > **Signal Attributes** Library, use the Rate Transition block.
- In the **DSP System Toolbox** > **Signal Operations** Library, you can use Upsample, Downsample, and Repeat blocks.
- In the **HDL Coder** > **HDL RAMs** Library, use the HDL FIFO block.

This model illustrates how to create a multirate design by using a Rate Transition block.

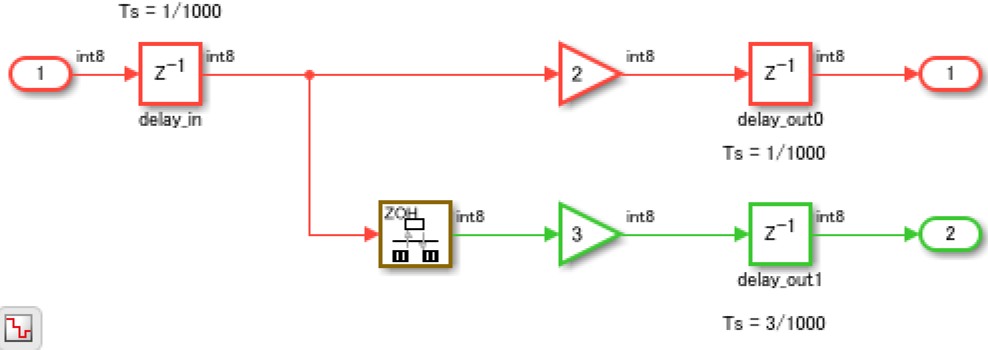
```
load_system('hdlcoder_multiclock')
set_param('hdlcoder_multiclock','SimulationCommand','Update')
open_system('hdlcoder_multiclock')
```



Copyright 2018-2021 The MathWorks, Inc.

The different colors in the model indicate that the model is multirate and has a faster rate D1 and a slower rate D2. To see the Rate Transition block that produces the different sample rates, double-click the DUT Subsystem.

```
open_system('hdlcoder_multiclock/DUT')
```



To see the sample times in your model, run this command:

```
ts = Simulink.BlockDiagram.getSampleTimes('hdlcoder_multiclock');
sampletime_D1 = ts(1)
sampletime_D2 = ts(2)
```

```
sampletime_D1 = 

  SampleTime with properties:

                  Value: [1.0000e-03 0]
            Description: 'Discrete 1'
          ColorRGBValue: [1 0.2706 0.2275]
             Annotation: 'D1'
             OwnerBlock: []
    ComponentSampleTimes: [0x0 Simulink.SampleTime]


sampletime_D2 = 

  SampleTime with properties:

                  Value: [0.0030 0]
            Description: 'Discrete 2'
          ColorRGBValue: [0.2275 0.7843 0.1922]
             Annotation: 'D2'
             OwnerBlock: []
    ComponentSampleTimes: [0x0 Simulink.SampleTime]
```

When you use a Rate Transition block in your model for multirate design, select the block parameters **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)**. Make sure the output sample rate is an integer multiple of the input sample rate.

For a multirate design, you can generate a single clock signal or multiple clock signals to control the clocking to blocks that operate at various sample rates. To specify this setting, in the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings** pane, specify the **Clock inputs** setting.

By default, **Clock inputs** is specified as `single`. A single clock is generated to control the clocking for all registers or Delay blocks in your model. The timing controller enable signals control the

clocking to various blocks in your design. This mode can increase the power dissipation as a single, fastest clock is connected to all registers in your design.

If you specify **Clock inputs** as `multiple`, a clock signal is generated for each sample rate in your design. However, this mode requires you to connect each of the clock, clock enable, and reset ports externally. This mode reduces power as the HDL design contains registers connected to slower clock signals. For more information, see “Using Multiple Clocks in HDL Coder” on page 20-14.

Use Dual Rate Dual Port RAM for Noninteger Multiple Sample Times

Guideline ID

`1.4.3`

Severity

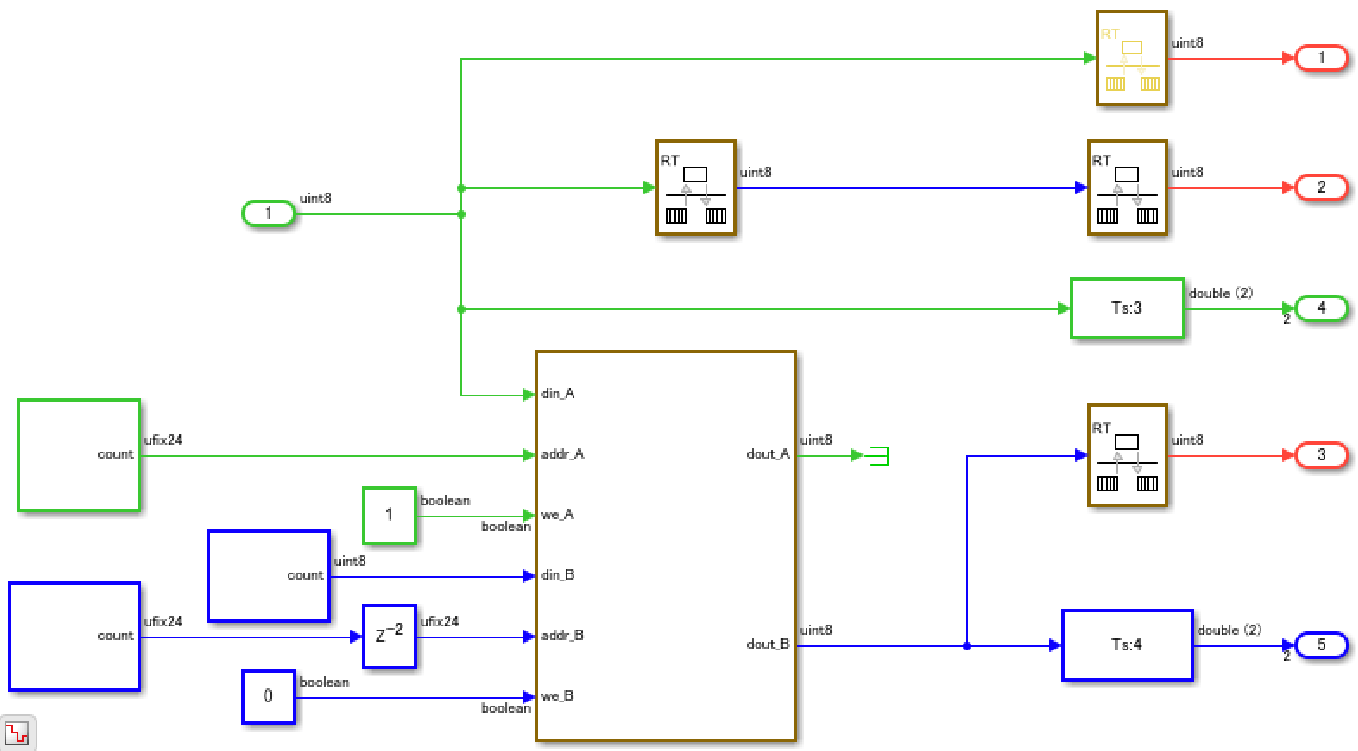
Mandatory

Description

When you use Rate Transition, Upsample, or Downsample blocks to create multirate models, the clock rates must be integer multiples of the base rate. To create a multirate model with clocks that are noninteger multiples, use a Dual Rate Dual Port RAM block. For integer clock multiplies, you can use the HDL FIFO or the Dual Rate Dual Port RAM block.

This model illustrates how you can create noninteger multiples of sample rates.

```
load_system('hdlcoder_dual_rate_dual_port_RAM')
set_param('hdlcoder_dual_rate_dual_port_RAM','SimulationCommand','Update')
open_system('hdlcoder_dual_rate_dual_port_RAM/DUT')
```



You cannot generate HDL code for this model because the Rate Transition blocks have the block parameter **Ensure data integrity during data transfer** cleared. To learn how you can manage address control when you use the RAM block, see “Design Considerations for RAM Blocks and Blocks in HDL Operations Library” on page 18-68.

Asynchronous Clock Modeling in HDL Coder

Guideline ID

1.4.4

Severity

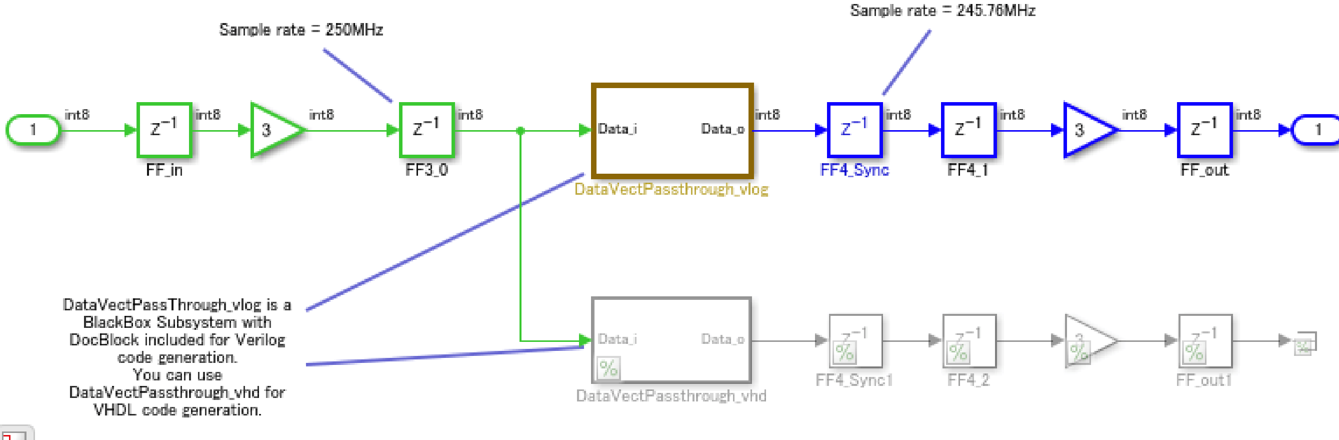
Recommended

Description

Most FPGA designs must have more than one clock domain with multiple parts of the design operating at various frequencies. You can model the various clock domains in Simulink® by using a pass-through implementation for transitioning between different sample rates. These sample rates correspond to the clock rates on the FPGA device.

For an example, open the model `hdlcoder_multi_clock_domain` and then open the DUT Subsystem.

```
load_system('hdlcoder_multi_clock_domain')
set_param('hdlcoder_multi_clock_domain', 'SimulationCommand', 'Update')
open_system('hdlcoder_multi_clock_domain/DUT')
```



You see a BlackBox Subsystem that contains a DocBlock, which is a text file that corresponds to the Verilog® code for a passthrough implementation. You can open the DocBlock to see the Verilog code. You see that the output of this Subsystem operates at a different sample rate or is in a clock domain that is different from the sample rate at the input of the Subsystem. The Subsystem also contains a commented out path that contains the VHDL® equivalent of the passthrough implementation. To generate VHDL code, uncomment this path and comment out the path that contains the Verilog BlackBox implementation.

To generate Verilog code for this model, run this command:

```
makehdl('hdlcoder_multi_clock_domain/DUT')
```

In the generated Verilog header file, you see the different clock domains in the model.

```
// -------------------------------------------------------------
//
// File Name: hdlsrc\hdlcoder_multi_clock_domain\DUT.v
// Created: 2018-10-05 11:30:21
//
// Generated by MATLAB 9.6 and HDL Coder 3.13
//
//
// -- -------------------------------------------------------------
// -- Rate and Clocking Details
// -- -------------------------------------------------------------
// Model base rate: 1.30208e-12
// Target subsystem base rate: 2.65428e-12
//
//
// Clock      Domain  Description
// -- -------------------------------------------------------------
// clk_1_3072 1       3072x slower than base rate clock
// clk_1_3125 2       3125x slower than base rate clock
// -- -------------------------------------------------------------
//
// Output Signal                 Clock      Domain  Sample Time
// -- -------------------------------------------------------------
// Output1                       (no clock) 0       4.06901e-09
```

```
// -- -------------------------------------------------------------
//
// -------------------------------------------------------------
```

Use Global Reset Type Setting Based on Target Hardware

Guideline ID

`1.4.5`

Severity

Recommended

Description

Matching the reset type to the FPGA architecture can improve resource utilization and the speed at which your design runs on the target hardware. To control this setting, in the Configuration Parameters dialog box, on the **HDL Code Generation** > **Global Settings** settings, specify the **Reset type**.

When you target Xilinx devices, set **Reset type** to `Synchronous`. For Intel or Altera devices, set **Reset type** to `Asynchronous`.

To make sure that you use the correct reset type for the hardware that you are targeting, in the HDL Code Advisor, run the model check “Check for global reset setting for Xilinx and Altera devices” on page 37-7.

Note Some Intel devices recommend using synchronous reset. For recommended reset settings, see the Intel or Xilinx documentation for that device.

See Also

Functions
`makehdl` | `makehdltb`

More About

- “Multirate Model Requirements for HDL Code Generation” on page 20-7
- “Code Generation from Multirate Models” on page 20-2
- “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 38-2

Modeling with Native Floating Point

HDL Coder native floating-point technology can generate HDL code from your floating-point design. These are some of the key features:

- Generation of target-independent HDL code that you can deploy on any FPGA or ASIC.
- Support for the full range of IEEE-754 features including denormal numbers, exceptions, and rounding modes.
- Extensive support for math and trigonometric blocks.

You can follow these guidelines as best practices when modeling your design for native floating-point code generation.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Guideline ID

`1.5.1`

Severity

Recommended

Description

Native floating-point support in HDL Coder generates code from your floating-point design. If your design has complex math and trigonometric operations or has data with a large dynamic range, use native floating-point. The generated HDL code is target-independent and complies with the IEEE-754 standard of floating-point arithmetic. To learn more, see “Getting Started with HDL Coder Native Floating-Point Support” on page 14-88.

You can use these modeling guidelines when using the native floating-point support in HDL Coder.

Use Blocks from HDL Floating Point Operations Library

The **HDL Floating Point Operations** block library consists of math and trigonometric functions and certain Simulink blocks that are configured for HDL code generation in native floating-point mode. For example, Discrete FIR Filter with **Architecture** set to `Fully Parallel`.

Use Floating-Point Types Based on Accuracy and Hardware Resource Usage Requirements

You can generate HDL code for models that contain floating-point and fixed-point data types in native floating-point mode. Floating point types have higher dynamic range but can potentially occupy more area on the target hardware. To design for these trade-offs, in your Simulink model, it is recommended to use floating-point data types to model the algorithm data path and fixed-point types to model the control logic. To switch between floating-point and fixed-point data types, use Data Type Conversion blocks.

See also “Data Type Considerations” on page 14-90.

Enable Optimizations such as Resource Sharing on Model

By enabling optimizations on the model, you can improve area and timing of your design on the target FPGA device. For example, to save area on the target FPGA device, use the resource sharing optimization. To share:

- Floating-point adders, enable **Share Adders**.
- Floating-point multipliers, enable **Share Multipliers**.
- Other floating-point resources, enable **Floating-Point IPs**.

See also “Resource Sharing” on page 21-45.

Simulate Latency of Blocks in Model

Floating-point designs have an inherent latency by default. This latency is added when generating HDL code for your model. It is recommended that you simulate latency in your model by adding this latency information to your original Simulink model. The code generator absorbs this latency during HDL code generation. To learn more, see “Latency Values of Floating-Point Operators” on page 14-99.

Customize Latency of Model or Blocks

You can customize the latency of an entire model, or selectively for certain blocks in your design. Using custom settings, you can specify a custom latency and design for trade-offs between latency and throughput.

To learn more, see “Latency Considerations with Native Floating Point” on page 14-104.

Use sincos Block Instead of Separate sin and cos Blocks

Certain modeling patterns that you use can optimize your model when you generate code with native floating-point technology. For example, if you are computing the trigonometric sine and cosine of the same input, in the `HDL Floating Point Operations` block library, use the Sincos block instead of separate Sin and Cos blocks. The Sincos block shares some of the logic that is used for computing the sine and cosine of the input. This implementation reduces the area footprint on the target FPGA device.

See also Trigonometric Function.

Use Tree as the HDL Architecture

To obtain a lower latency implementation, use `Tree` as the **HDL Architecture** for blocks such as the Sum of Elements and Product of Elements.

See also Sum of Elements and Product of Elements.

See Also

Functions
`makehdl` | `makehdltb`

More About

- “Numeric Considerations for Native Floating-Point” on page 14-92

- “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111
- “Verify the Generated Code from Native Floating-Point” on page 14-133

Design Considerations for RAM Blocks and Blocks in HDL Operations Library

Follow these guidelines to learn how you can use RAM blocks and blocks in the **HDL Operations** library when modeling your design.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

RAM Block Access Considerations

Guideline ID

`2.1.1`

Severity

Recommended

Description

In the **HDL RAMs** block library, there are seven different RAM blocks and a HDL FIFO block. If you see a RAM block that has the term `System` as part of the block name, such as Single Port RAM System, it is recommended that you use this block instead of the equivalent block that does not have `System` as part of the name, such as Single Port RAM. These blocks have `System` as part of the name because the block implementation is based on the `hdl.RAM` System object. The system blocks support vector inputs and yield much faster simulation results when used in your Simulink model.

When you use these blocks, make sure that the input sample time and output sample time are the same. This table illustrates the various RAM blocks that you can use and their purpose. The generated HDL code for these blocks maps to RAM in most FPGAs.

Block Name	Recommended Usage
Single Port RAM System	Use this block to replace the Single Port RAM block in your model. You obtain faster simulation results when using this block in your model. The block implementation uses a MATLAB System block that uses the `hdl.RAM` System object. Use this block to perform sequential read and write operations. In the Block Parameters dialog box of the block, you can specify an initial value for the RAM. To perform simultaneous read and write operations to different addresses, use the Simple Dual Port RAM System or the Dual Port RAM System block instead. The block does not support boolean inputs. Cast `boolean` types to `ufix1` for input to the block.

Block Name	Recommended Usage
Simple Dual Port RAM System	Use this block to replace the Simple Dual Port RAM block in your model. You obtain faster simulation results when using this block in your model. The block implementation uses a MATLAB System block that uses the `hdl.RAM` System object. Use this block to perform simultaneous read and write operations. It has a single output port to read data. In the Block Parameters dialog box of the block, you can specify an initial value for the RAM. The block does not support boolean inputs. Cast `boolean` types to `ufix1` for input to the block.
Dual Port RAM System	Use this block to replace the Dual Port RAM block in your model. You obtain faster simulation results when using this block in your model. The block implementation uses a MATLAB System block that uses the `hdl.RAM` System object. Use this block to perform simultaneous read and write operations. It has a read data output port and a write data output port. In the Block Parameters dialog box of the block, you can specify an initial value for the RAM. If you do not want to use the write data output port, to achieve better RAM inference, use the Simple Dual Port RAM System block instead. The block does not support boolean inputs. Cast `boolean` types to `ufix1` for input to the block.
True Dual Port RAM System	The block implementation uses a MATLAB System block that uses the `hdl.RAM` System object. Use this block to perform simultaneous read and write operations. It has two write data output ports and each write port also performs a read operation. In the Block Parameters dialog box of the block, you can specify an initial value for the RAM. The block does not support boolean inputs. Cast `boolean` types to `ufix1` for input to the block.
Simple Tri Port RAM System	The block implementation uses a MATLAB System block that uses the `hdl.RAM` System object. Use this block to perform simultaneous read and write operations. It has two output ports to read data. In the Block Parameters dialog box of the block, you can specify an initial value for the RAM. The block does not support boolean inputs. Cast `boolean` types to `ufix1` for input to the block.

<table>
<tr><th>Block Name</th><th>Recommended Usage</th></tr>
<tr><td>Dual Rate Dual Port RAM</td><td>This block does not have an equivalent System object-based implementation.

Use this block to perform simultaneous read and write operations to two different addresses that operate at different clock rates. You cannot perform concurrent access to the same address of the RAM.

To run the RAM ports at multiple clock rates, set Clock Inputs to <code>Multiple</code>. You can access this RAM twice in one clock cycle.</td></tr>
<tr><td>HDL FIFO</td><td>The HDL FIFO block stores a sequence of samples in a first in, first out (FIFO) register.

The inputs, <code>In</code> and <code>Push</code>, and the outputs, <code>Out</code> and <code>Pop</code> can run at different sample times. Specify the ratio of output to input sample time as a positive integer or <code>1/N</code> such that <code>N</code> is a positive integer. For example:

• If you specify the ratio as <code>2</code>, the output sample time is twice the input sample time. The outputs run slower than the input.
• If you specify the ratio as <code>1/2</code>, the output sample time is half the input sample time. The outputs run faster than the input.

The signals <code>Full</code>, <code>Empty</code>, and <code>Num</code> run at the fastest rate in your model. When you use the control output of the FIFO in an input, you may have to perform to a rate transition.

The input and output rates of the FIFO block are synchronous to each other.</td></tr>
</table>

Serial to Parallel Conversion

Guideline ID

`2.1.2`

Severity

Informative

Description

You can use the Serializer1D and Deserializer1D blocks to perform serial to parallel and parallel to serial conversion.

See Also

Functions
`makehdl`

Related Examples

- “Getting Started with RAM and ROM in Simulink” on page 19-58

More About

- “HDL Code Generation from hdl.RAM System Object” on page 1-39
- “Map Matrices to Block RAMs to Reduce Area” on page 8-2

Usage of Blocks in Logic and Bit Operations Library

These guidelines illustrate how to model your design for generating HDL-ready code from blocks in the **Logic and Bit Operations** Library. The Library contains blocks that perform logical and bitwise operations, bit reduction, and concatenation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Logical and Arithmetic Bit Shift Operations

Guideline ID

`2.2.1`

Severity

Informative

Description

You can use Simulink blocks to perform bit shifting operations. The blocks can perform logical and arithmetic bit shift. Left logical and arithmetic bit shift produce the same results but right logical shift and arithmetic shift operate differently as illustrated in this table.

Block/Function Name	Parameter/ Operation	Verilog or SystemVerilog Code Equivalent	VHDL code equivalent	Comments
Bit Shift	`Shift Left Logical`	<<<	`sll` (`sll` and `SHIFT_LEFT` are the same in VHDL.	This mode is the default mode for the block. The left shift operation does not preserve the sign bit. If the input uses a signed data type and has a positive value, the left shift operation shifts a `0` into the empty bit on the LSB (Least Significant Bit) side.
	`Shift Right Logical`	>>	`srl`	This mode does not preserve the sign bit. If the input uses a signed data type and has a positive value, the right shift operation shifts a `0` into the empty bit on the MSB (Most Significant Bit) side.
	`Shift Right Arithmetic`	>>>	`SHIFT_RIGHT`	When the input is a signed data type, the sign bit is preserved, and other bits shift to the right.

Block/Function Name	Parameter/ Operation	Verilog or SystemVerilog Code Equivalent	VHDL code equivalent	Comments
Shift Arithmetic block `bitshift` function	Positive value/shift right arithmetic	>>>	`SHIFT_RIGHT`	When the input is a signed data type, the sign bit is preserved, and other bits shift to the right.
	Negative value/ shift left arithmetic	<<<	`sll`	This mode does not preserve the sign bit. If the input uses a signed data type and has a positive value, the left shift operation shifts a `0` into the empty bit on the LSB side. This mode does not check underflows and overflows.
`bitsll` function	None/logical left shift	<<<	`sll`	This mode does not preserve the sign bit. The generated HDL code is the same as that of the `Shift Left Logical` mode of the Bit Shift block.
`bitsrl` function	None/logical right sift	>>	`srl`	This mode does not preserve the sign bit. The generated HDL code is the same as that of the `Shift Right Logical` mode of the Bit Shift block.
`bitsra` function	None/arithmetic right shift	>>>	`SHIFT_RIGHT`	When the input is a signed data type, the sign bit is preserved, and other bits shift to the right. The generated HDL code is the same as that of the `Shift Right Arithmetic` mode of the Bit Shift block.

The difference between a logical shift and an arithmetic shift is whether the sign bit is preserved. For signed data types, this bit is the MSB. In a logical right shift, the sign bit is shifted to the right and zero goes into the MSB side. In an arithmetic right shift, the MSB (sign bit) is preserved during the shift operation. For example, this code illustrates the difference between the functions.

```
A = fi([], 1, 4, 0, 'bin','1011');
B = bitsrl(A, 2)
```

```
B =

     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
             WordLength: 4
         FractionLength: 0

B.bin

ans =

    '0010'

C = bitsra(A, 2)

C =

    -2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 4
        FractionLength: 0

C.bin

ans =

    '1110'
```

Usage of Logical Operator, Bitwise Operator, and Bit Reduce Blocks

Guideline ID

2.2.2

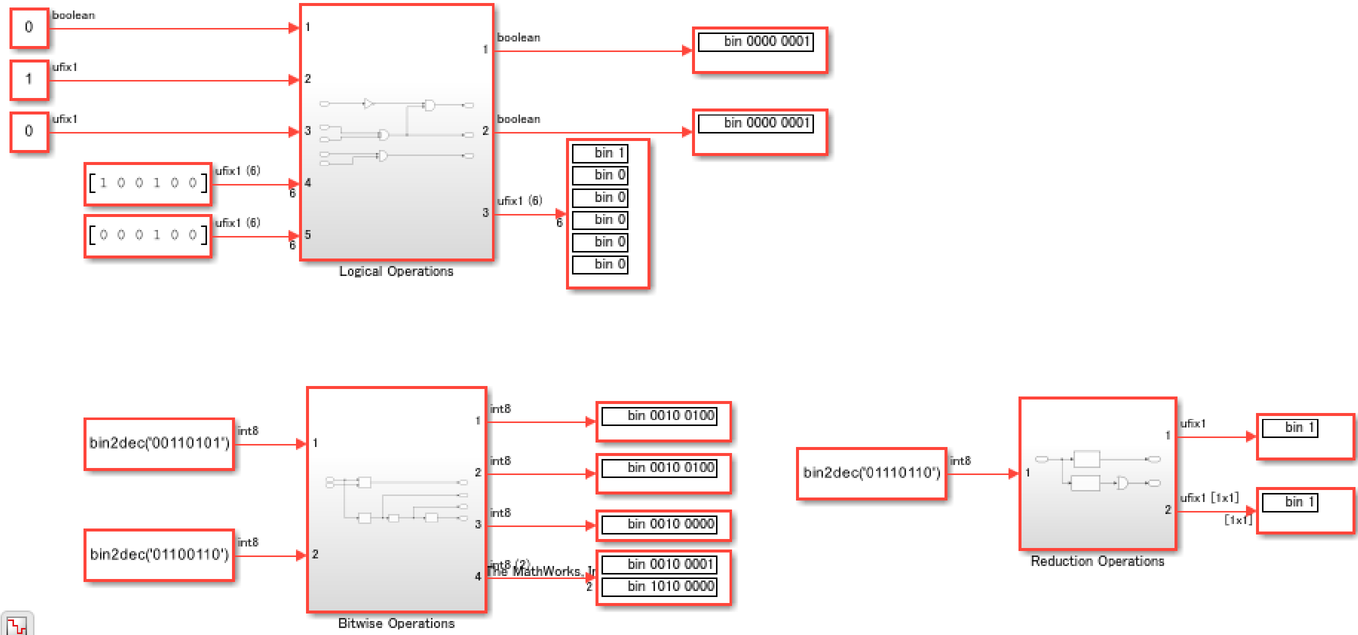
Severity

Informative

Description

For an example of logical and bitwise operations, open the model `hdlcoder_logical_bitwise_operations.slx`.

```
load_system('hdlcoder_logical_bitwise_operations')
sim('hdlcoder_logical_bitwise_operations')
open_system('hdlcoder_logical_bitwise_operations')
```

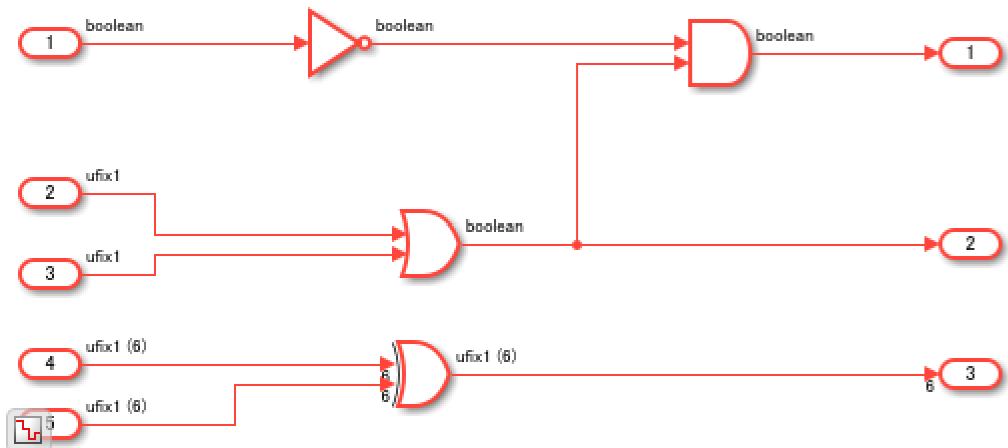


For single-bit operations that use `Boolean` or `ufix1` data types, use a Logical Operator block. To view the operation as a logical circuit symbol, in the Block Parameters dialog box of the block, specify the **Icon shape** as `Distinctive`. You can also input vectors that have `Boolean` or `ufix1` data types to the block.

`Boolean` and `ufix1` are different data types. Avoid using both data types within the same model, or using them interchangeably. See “Simulink Data Type Considerations” on page 18-134.

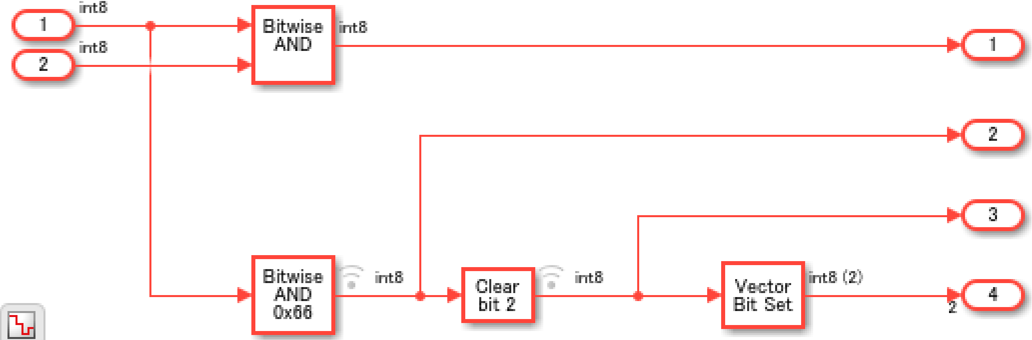
For an example of using the block, open the `Logical Operations` Subsystem.

```
open_system('hdlcoder_logical_bitwise_operations/Logical Operations')
```



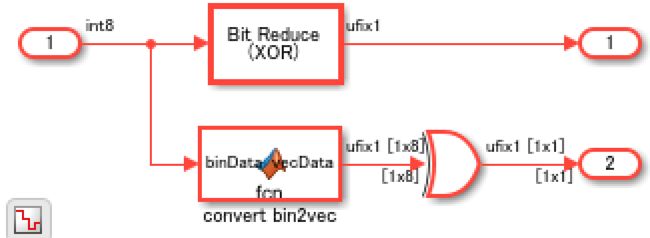
For bitwise operations on two or more bits that use integer or fixed-point data types, use the Bitwise Operator block. For an example, double-click the `Bitwise Operations` Subsystem.

```
open_system('hdlcoder_logical_bitwise_operations/Bitwise Operations')
```



To perform a bit-by-bit reduction operation on a vector that uses `Boolean` or `ufix1` and return a 1-bit value, use the Bit Reduce block. For an example, double-click the `Reduction Operations` Subsystem.

```
open_system('hdlcoder_logical_bitwise_operations/Reduction Operations')
```



The MATLAB Function block inside the Subsystem converts the 8-bit vector to a vector of 8 1-bit `ufix1` elements.

```
open_system('hdlcoder_logical_bitwise_operations/Reduction Operations/convert bin2vec')
```

Use Boolean Output for Compare to Constant and Relational Operator Blocks

Guideline ID

2.2.3

Severity

Strongly Recommended

Description

For Compare To Constant, Compare To Zero, and Relational Operator blocks, you can specify `uint8` or `boolean` as the **Output data type**. To generate efficient HDL code for models that contain these blocks, specify `boolean` as the **Output data type**, because the HDL code has to connect only the LSB.

For a Relational Operator block, make sure that both inputs are of the same data type. Using different data types for the inputs can result in unintended truncation of bits such as the sign bit, which can lead to simulation mismatches after HDL code generation.

To verify whether Relational Operator blocks in your model use the same input data type, and use `boolean` as the output data type, run the HDL model check "Check for Relational Operator block usage" on page 37-30.

See Also

Functions
`makehdl`

Blocks
Bitwise Operator | Extract Bits

More About

- "Bitwise Operations in MATLAB for HDL and HLS Code Generation" on page 1-58

Generate FPGA Block RAM from Lookup Tables

To map the lookup table blocks to random-access memory (RAM) to save area on your target Field Programmable Gate Array (FPGA) device, follow these guidelines.

Each guideline has a severity level that indicates the level of compliance requirements. For more information, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Guideline ID

`2.3.1`

Severity

Strongly Recommended

Description

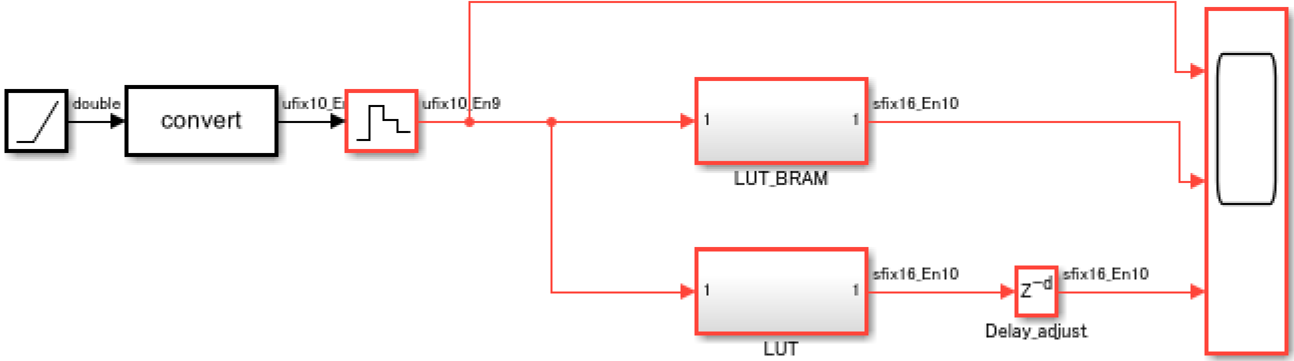
To map lookup tables to a block RAM, you can use the **Map lookup tables to RAM** parameter located in the **HDL Code Generation** tab > **Optimization** > **Pipelining** tab in the Model Configuration Parameters dialog box. This parameter is selected on by default. The optimization inserts a Delay block that has a **Delay length** of `1` and **ResetType** set to `none` immediately following the Lookup Table block. This modeling pattern efficiently maps your design to a Block RAM on the FPGA. To use the map lookup tables to RAM option, you must:

- Make sure that the Map lookup tables to RAM option is selected on for the model.
- Specify the synthesis tool.

Alternatively, you can selectively enable this optimization for certain subsystems in your design by using the `MapToRAM` HDL Block Property and disabling the **Map lookup tables to RAM** option for the model or create the modeling pattern in your design that is the same as the pattern otherwise generated by the optimization.

For an example, open the model `hdlcoder_LUT_BRAM_mapping.slx`.

```
open_system('hdlcoder_LUT_BRAM_mapping')
set_param('hdlcoder_LUT_BRAM_mapping','SimulationCommand','Update')
```



Copyright 2019 The MathWorks, Inc.

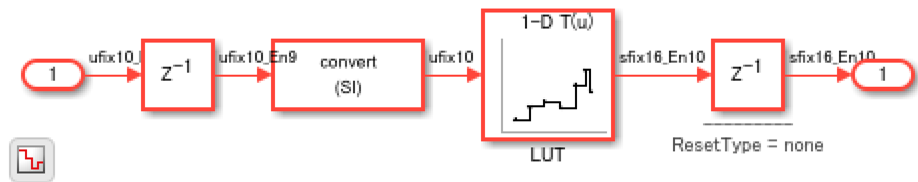
The **Map lookup tables to RAM** option is enabled on this model.

```
hdlget_param('hdlcoder_LUT_BRAM_mapping','LUTMapToRAM')
```

```
ans =

    'on'
```

The LUT_BRAM Subsystem contains a 1-D Lookup Table block followed by a Delay block that has a **Delay length** of 1 and **ResetType** set to none.

```
open_system('hdlcoder_LUT_BRAM_mapping/LUT_BRAM')
```



When you generate HDL code and synthesize the design on an FPGA, this modeling pattern efficiently maps to Block RAM. This figure displays the synthesis results for the LUT_BRAM Subsystem.

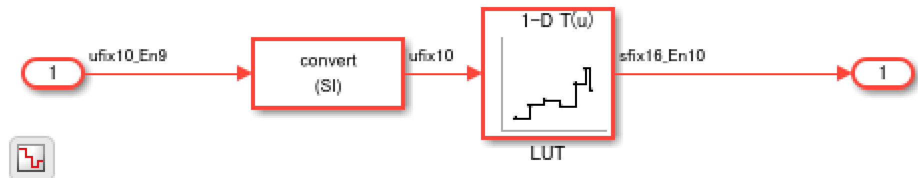
Parsed resource report file: LUT_OK_utilization_synth.rpt.

Resource summary	
Resource	**Usage**
Slice LUTs	0
Slice Registers	0
DSPs	0
Block RAM Tile	0.5

Parsed timing report file: timing_post_map.rpt.

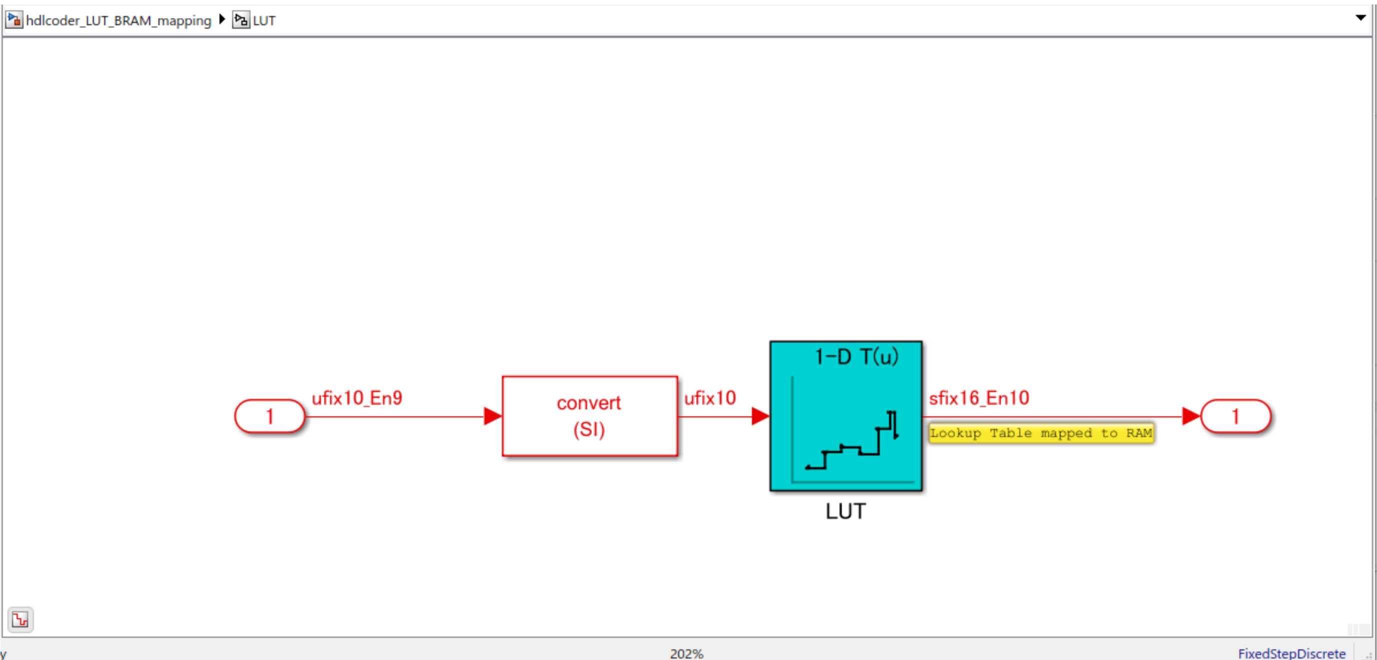
The `LUT` Subsystem in this model does not use this modeling pattern.

```
open_system('hdlcoder_LUT_BRAM_mapping/LUT')
```

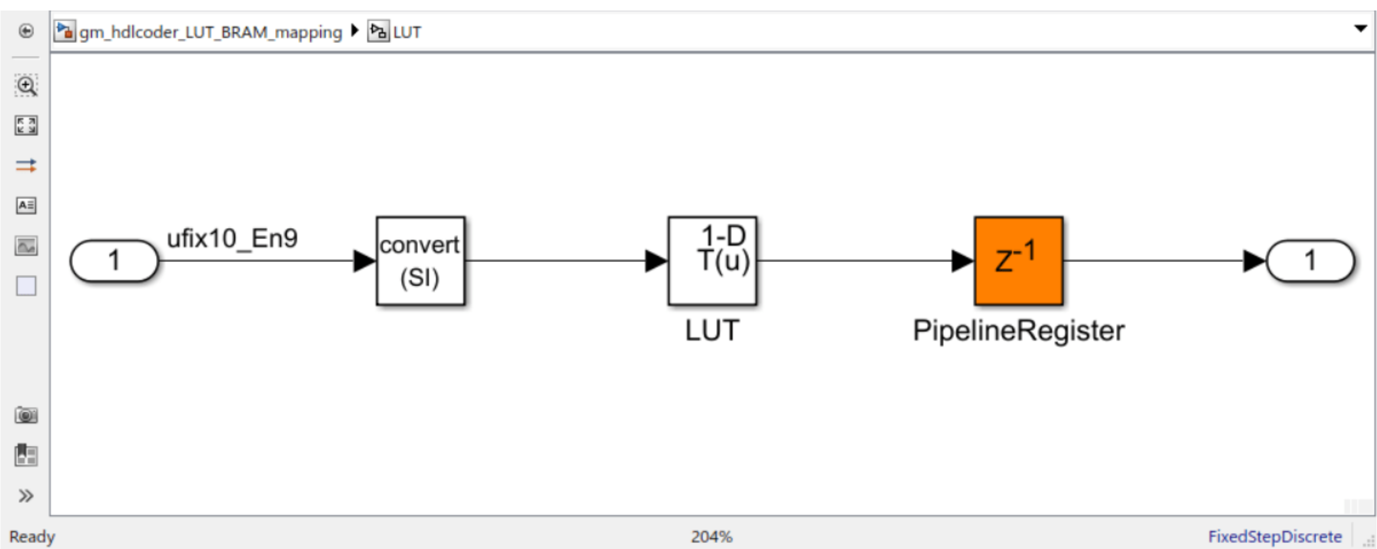


Because the **Map lookup tables to RAM** option is enabled for this model, this subsystem still maps the logic to Block RAM. Click the MATLAB® script linked in the MATLAB command window to highlight lookup tables mapped to RAM in your model.

```
### Generating HDL for 'hdlcoder_LUT_BRAM_mapping/LUT'.
### Using the config set for model hdlcoder_LUT_BRAM_mapping for HDL code generation parameters.
### Running HDL checks on the model 'hdlcoder_LUT_BRAM_mapping'.
### Begin compilation of the model 'hdlcoder_LUT_BRAM_mapping'...
### Applying HDL optimizations on the model 'hdlcoder_LUT_BRAM_mapping'...
### 'LUTMapToRAM' is set to 'On' for the model. This option is used to map lookup tables to a block RAM in hardware.  To disable pipeline insertion for mapping
### The code generation and optimization options you have chosen have introduced additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
### Output port 1: 1 cycles.
### Begin model generation.
### Model generation complete.
### To highlight look up tables mapped to RAM, click the following MATLAB script: hdl_prj\hdlsrc\hdlcoder_LUT_BRAM_mapping\highlightLUTPipeliningDiagnostic.m
### To clear highlighting, click the following MATLAB script: hdl_prj\hdlsrc\hdlcoder_LUT_BRAM_mapping\clearhighlighting.m
```



In order to map the lookup table to Block RAM, the generated model for the LUT Subsystem inserts a Delay block that has a **Delay length** of 1 and **ResetType** set to none immediately following the Lookup Table block.



See Also

Functions
makehdl

Properties
“MapToRAM” on page 19-19 | “MapPersistentVarsToRAM” on page 19-17

Model Settings
Map lookup tables to RAM | Map pipeline delays to RAM | RAM mapping threshold

More About

- “Apply RAM Mapping to Optimize Area” on page 21-120
- “RAM Mapping with the MATLAB Function Block” on page 21-125
- “Adaptive Pipelining” on page 21-181

Recommended Block Parameter Settings of Multiport Switch Block for Numeric and Enumerated Types

Guideline ID

2.3.2

Severity

Recommended

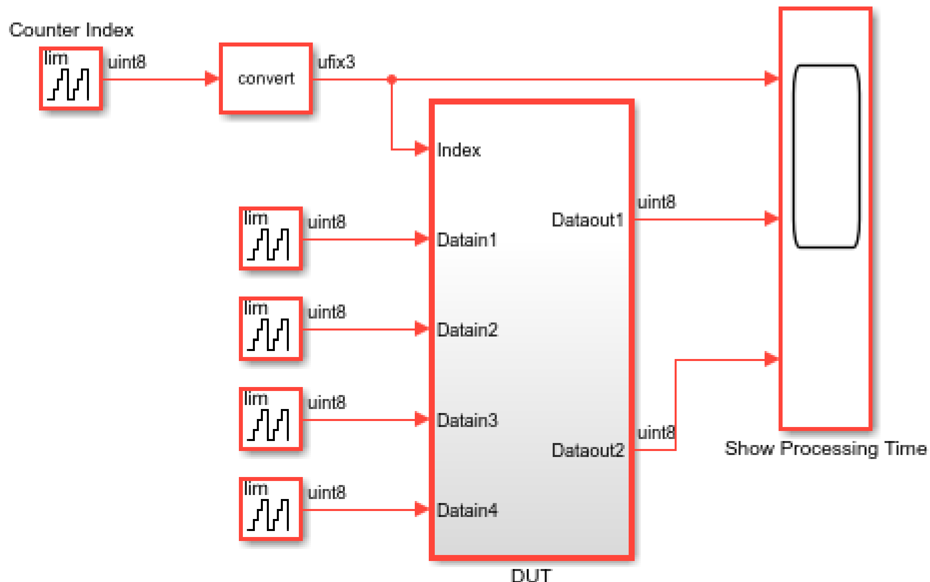
Description

To switch multiple input signals in your model for HDL code generation, you can use Switch, Multiport Switch, and Index Vector blocks. The Index Vector block is equivalent to the Multiport Switch block that has **Number of data terminals** set to 1.

You can use numeric and enumerated data types for the Multiport Switch block. When using numeric data types, in the Block Parameters dialog box of the Multiport Switch block, set **Data Port Order** to `Zero-based contiguous` and **Data port for default case** to `Last data port`. When number of input signals is a power of two, the `Zero-based contiguous` mode minimizes the number of bits of the control port.

For an example that uses Multiport Switch block with numeric types, open the model `hdlcoder_multiport_switch_numeric`.

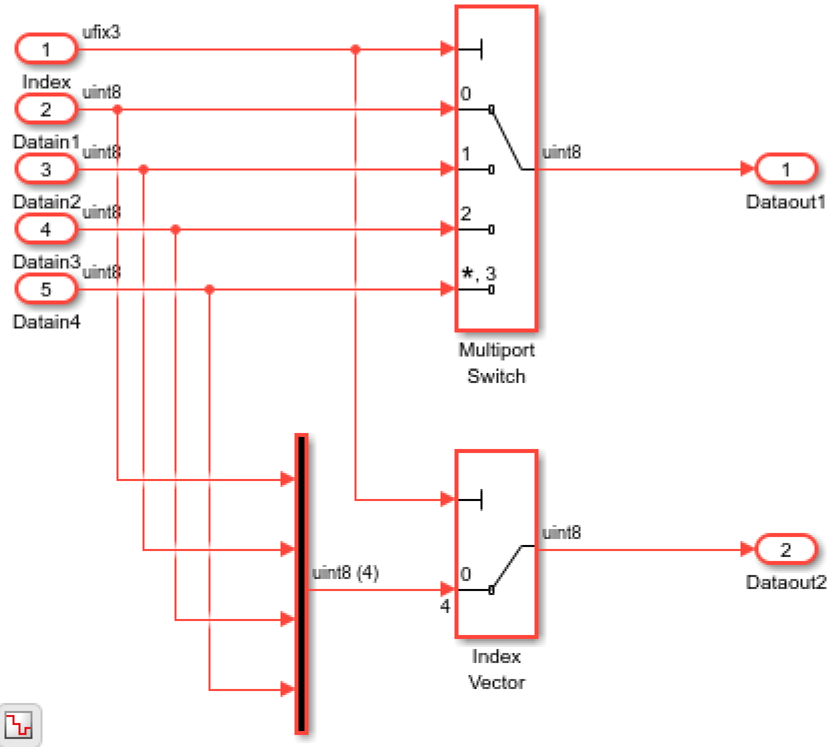
```
open_system('hdlcoder_multiport_switch_numeric')
set_param('hdlcoder_multiport_switch_numeric', 'SimulationCommand', 'Update')
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains a Multiport Switch block and an Index Vector block.

```
open_system('hdlcoder_multiport_switch_numeric/DUT')
```



To generate HDL code for the DUT, run the `makehdl` function.

```
makehdl('hdlcoder_multiport_switch_numeric/DUT')
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_multiport_switch_numeric')">hdlcod
### Generating HDL for <a href="matlab:open_system('hdlcoder_multiport_switch_numeric/DUT')">hdlc
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_multipo
### Running HDL checks on the model 'hdlcoder_multiport_switch_numeric'.
### Begin compilation of the model 'hdlcoder_multiport_switch_numeric'...
### Working on the model 'hdlcoder_multiport_switch_numeric'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_multiport_switch_numeric', 'G
### Begin model generation 'gm_hdlcoder_multiport_switch_numeric'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_multiport_switch_numeri
### Begin Verilog Code Generation for 'hdlcoder_multiport_switch_numeric'.
### Working on hdlcoder_multiport_switch_numeric/DUT as hdlsrc\hdlcoder_multiport_switch_numeric\
### Code Generation for 'hdlcoder_multiport_switch_numeric' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_multiport_switch_numeric' complete with 0 errors, 0 warnings, and 0 m
### HDL code generation complete.
```

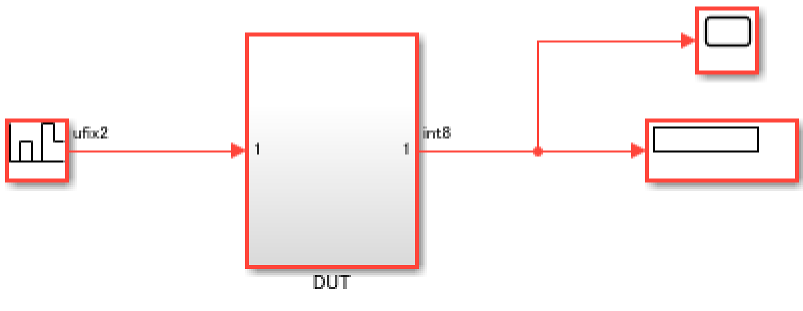
When you use enumerated types as input to the Multiport Switch, set **Data Port Order** to `Specify Indices`. You define the enumeration class in a MATLAB® file. When you use the default case, set the **Default case diagnostic** to `Warning` or `None`.

For an example that uses Multiport Switch block with enumerated types, open the model hdlcoder_multiport_switch_enum. This code shows the enumerated class defined in MATLAB.

```
classdef BasicColors < Simulink.IntEnumType

  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
  end

  methods (Static)
    function retVal = getDefaultValue()
      retVal = BasicColors.Blue;
    end
  end

end
```

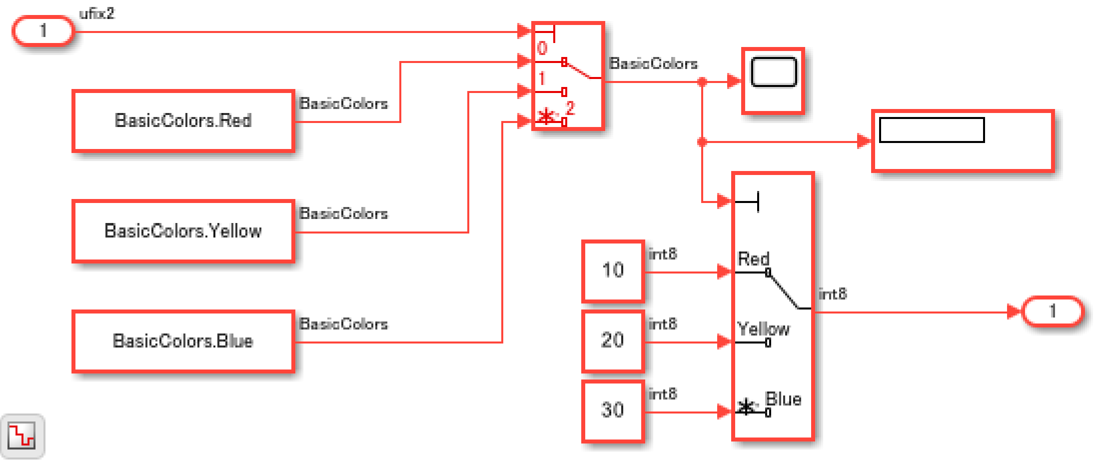
Open the model hdlcoder_multiport_switch_enum.

```
open_system('hdlcoder_multiport_switch_enum')
set_param('hdlcoder_multiport_switch_enum', 'SimulationCommand', 'Update')
```



The DUT subsystem contains two Multiport Switch blocks. The second Multiport Switch block has **Data Port Order** set to Specify Indices.

```
open_system('hdlcoder_multiport_switch_enum/DUT')
```



To generate HDL code for the DUT, run the `makehdl` function.

```
makehdl('hdlcoder_multiport_switch_enum/DUT')
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_multiport_switch_enum')">hdlcoder_
### Generating HDL for <a href="matlab:open_system('hdlcoder_multiport_switch_enum/DUT')">hdlcode
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_multipo
### Running HDL checks on the model 'hdlcoder_multiport_switch_enum'.
### Begin compilation of the model 'hdlcoder_multiport_switch_enum'...
### Begin compilation of the model 'hdlcoder_multiport_switch_enum'...
### Working on the model 'hdlcoder_multiport_switch_enum'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_multiport_switch_enum', 'Gene
### Begin model generation 'gm_hdlcoder_multiport_switch_enum'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_multiport_switch_enum\g
### Begin VHDL Code Generation for 'hdlcoder_multiport_switch_enum'.
### Working on hdlcoder_multiport_switch_enum/DUT as hdlsrc\hdlcoder_multiport_switch_enum\DUT.vh
### Generating package file hdlsrc\hdlcoder_multiport_switch_enum\DUT_pkg.vhd.
### Code Generation for 'hdlcoder_multiport_switch_enum' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_multiport_switch_enum' complete with 0 errors, 0 warnings, and 0 mess
### HDL code generation complete.
```

See Also

`makehdl` | `hdlset_param`

Related Examples

- "Set CodingStyle For Multiport Switch Block" on page 19-7
- "Use Bus Signals to Improve Readability of Model and Generate HDL Code" on page 18-52

Guidelines for Using Selector Blocks to Extract Input Elements from Vector or Matrix Signals

Follow these guidelines when you want to select input elements from vector or matrix signals using Selector blocks.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see "HDL Modeling Guidelines Severity Levels" on page 18-3.

Guideline ID

`2.3.3`

Severity

`Recommended`

Description

To extract scalar or partial vector input elements from vector or matrix signals in a model that you want to use for HDL Code generation, use Selector, Multiport Selector or Index Vector blocks. Use these parameter settings:

Intended Usage	Recommended Block	Block Parameter Settings	For Vector Inputs	For Matrix Inputs
Extracting one signal from vector or matrix inputs with a fixed or variable output range	Selector	**Index mode**: `Zero-based` **Index option**: Select index options with `(dialog)` when the output range is fixed. For variable output range, select options with `(port)`. • `Index vector (dialog)`: Specify the output range by entering index of each element. • `Start index (dialog)`: Select the output range by specifying the start index and output size. • `Index vector (port)`: Select the output range by specifying the index in the external port. • `Start index (port)`: Select the output range by specifying the index in the external port and a fixed output size. • `First and last index (port)`: HDL code generation is not	Set the **Number of input dimensions** to 1. The Selector block output is a: • Scalar when you specify only one index • Vector when you specify multiple indices	Set the **Number of input dimensions** to 2. The Selector output is a: • Scalar when you specify only one index for first and second dimension • Row vector when you specify the only one index for first dimension • Column vector when you specify the only one index for second dimension • Matrix when you specify multiple indices for first and second dimension.

Intended Usage	Recommended Block	Block Parameter Settings	For Vector Inputs	For Matrix Inputs
		supported for variable size.		
Extracting multiple signals from a vector signal with a fixed output range	Multiport Selector	Specify multiple subsets of scalar or vector indices to the **Indices to output** parameter. Example: `{1, [2,4]}` The block uses one-based indexing.	The block generates output signal for each index set respectively. Depending on whether each index set is a scalar or a vector, the corresponding output port is also a scalar or vector signal.	HDL code generation is not supported for matrix input. To extract multiple ranges from a matrix, convert the matrix input signal to a vector.
Extracting only one element as a signal from a vector signal	Index Vector	**Data port order**: `Zero-based contiguous` **Number of data ports**: `1` HDL Block Property**Coding Style**: Set to `case_stmt` to generate code as a `case` statement. Do not use the `ifelse_stmt` setting, because it brings deep logic combination using `if-elseif` structure.	You can select only one element in the input vector. The block output is a scalar signal.	Simulation and HDL code generation are not supported for matrix inputs.

Modeling Considerations

- Although you can use the Variable Selector block when the extraction range is variable and the inputs and outputs are vectors, the HDL code generated from this block has an `if-elseif` structure, and is therefore not recommended for this configuration.
- When you use Selector and Variable Selector blocks that has variable extraction range, you must use a built-in data type such as `uint8` as an input to the **Index port(Idx)**. This may cause small redundant circuit.
- Selector and Multiport Selector blocks support zero-based indices. When you use Selector or Multiport Selector blocks to select multiple elements, vector or matrix signal is output from one port. Though the HDL code generated from a Selector that uses zero-based index and Multiport

Selector are the same, the code from Selector block has better traceability because it keeps the same indices between the model and the generated code.

- Although the MATLAB code only supports one-based indices, the generated HDL code uses zero-based indexes. When you use Selector block to extract multiple sets of elements, these sets are output from multiple output ports in the generated HDL code.

See Also

Selector | Multiport Selector | Index Vector

Guidelines for Using Assignment Blocks to Write Elements in Vectors, Matrices, and 3-D Arrays

Follow these guidelines when writing input signals to an element in a vector, matrix, or 3-D array signal.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see "HDL Modeling Guidelines Severity Levels" on page 18-3.

Guideline ID

`2.3.4`

Severity

`Recommended`

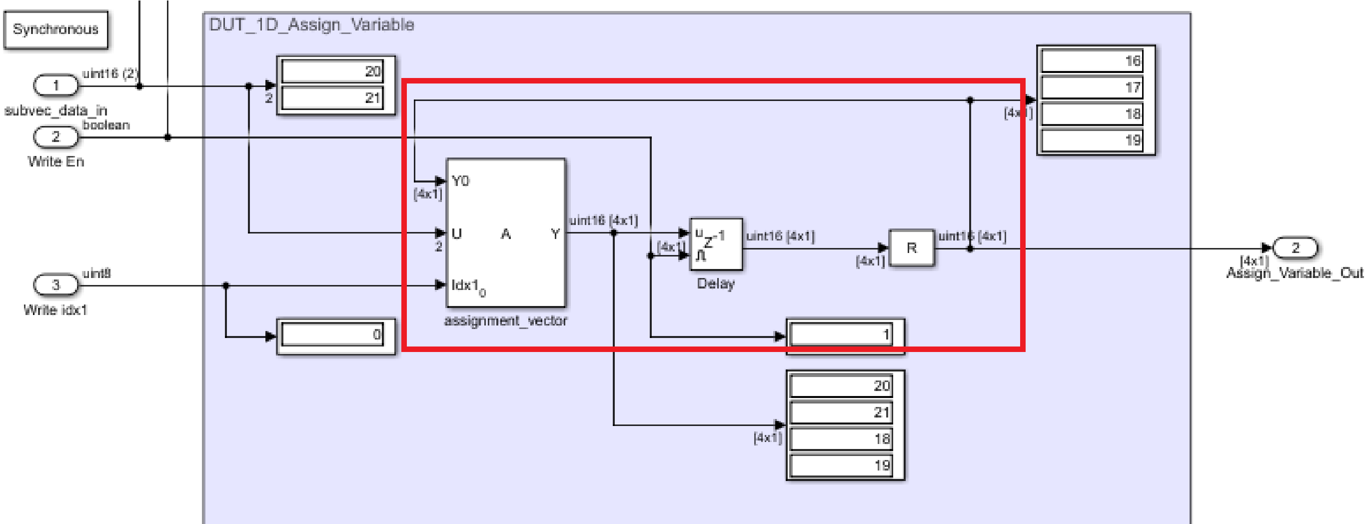
Description

You can use the Assignment block to write input signals to an element in a vector, matrix, or 3-D array signal. For HDL code generation, use these block parameter settings:

- **Number of output dimensions**: Set this parameter to `1` when the output is a vector, `2` for a 2-D array, or `3` for a 3-D array.
- **Index mode**: Use zero-based indexing so that the generated code matches the model.
- **Initialize output (Y)**: Set this parameter to `Initialize using input port <Y0>`, which initializes the output with the signal at the input port **Y0**. You cannot use the `Specify size for each dimension in table` setting for HDL code generation. To enable this parameter, set **Index Option** to `Index vector (port)` or `Starting index (port)` for one or more dimensions.
- **Index option**:
  - When using an Assignment block that has a variable index for an application such as register bank usage, choose a port-related setting, such as `Index Vector (port)`. For a fixed index, select from dialog-related settings, such as `Index vector (dialog)`.
  - When assigning individual values to multiple elements of the output port **Y**, use a input signal with the same size as the elements specified by setting of the **Index option** parameter.
  - When assigning the same value to multiple elements, use a scalar input signal regardless the setting of the Index option parameter.

Model a Register Bank by Using Assignment Block

This example shows how to model a register bank by using the Assignment block. The model in this figure contains an Assignment block that has variable index, a Delay block, and a Reshape block. To model as a register bank, the model employs a feedback loop from the output of the Reshape block to the input signal, **Y0**, of the Assignment block **Y0**. By specifying a scalar value to the input signal **U** as the register value and limiting the index from the port to one write address, the model behaves as a register bank. When you specify multiple elements to the input **Idx1**, the model simultaneously rewrites the elements in the output signals.



In the figure, when the column vector output `[16;17;18;19]` from the Delay block is input as the **Y0** value, a column vector `[20;21]` with two elements is input to the write input signal **U**. Because the start index of **Idx1** is `0`, the input signal **U** overwrites the 0th to 1st element of **Y0**, and the output signal **Y** becomes the column vector `[20;21;18;19]`. The Delay block outputs signal **Y** when the **Write En** signal is valid, or otherwise holds previous value.

See Also

Assignment

Usage of Different Subsystem Types

To learn how to use different types of subsystems in your design and model your algorithm hierarchically, use these guidelines. Each guideline has a severity level that indicates the level of compliance requirements. For more information, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Virtual Subsystem: Use as DUT

Guideline ID

2.4.1

Severity

Strongly Recommended

Description

A virtual subsystem is a subsystem that is not a conditionally-executed subsystem or an Atomic Subsystem. By default, a regular Subsystem block that you add to your Simulink model is a virtual subsystem. Non-virtual subsystem types include Atomic Subsystem, model reference, Variant Subsystem, and a variant model.

To determine whether a subsystem is virtual, use the `get_param` function with the parameter `IsSubsystemVirtual`. For example:

```
get_param('sfir_fixed/symmetric_fir', 'IsSubsystemVirtual')
```

Atomic and Virtual Subsystems: Generate Reusable HDL Files

Guideline ID

2.4.2

Severity

Recommended

Description

To generate a single HDL file for identical instances of the subsystems that you use at lower levels of a hierarchy, see “Generate Reusable Code for Subsystems” on page 25-18 and Code reuse.

To enable resource sharing on a subsystem unit, see “General Considerations for Sharing of Subsystems” on page 18-152.

You must turn off signal logging for generating reusable code from atomic and virtual subsystems. Generation of reusable code from atomic and virtual subsystem might not succeed if you log the signals inside the subsystems.

Variant Subsystem: Using Variant Subsystems for HDL Code Generation

Guideline ID

`2.4.3`

Severity

Mandatory

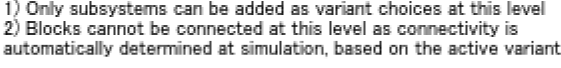
Description

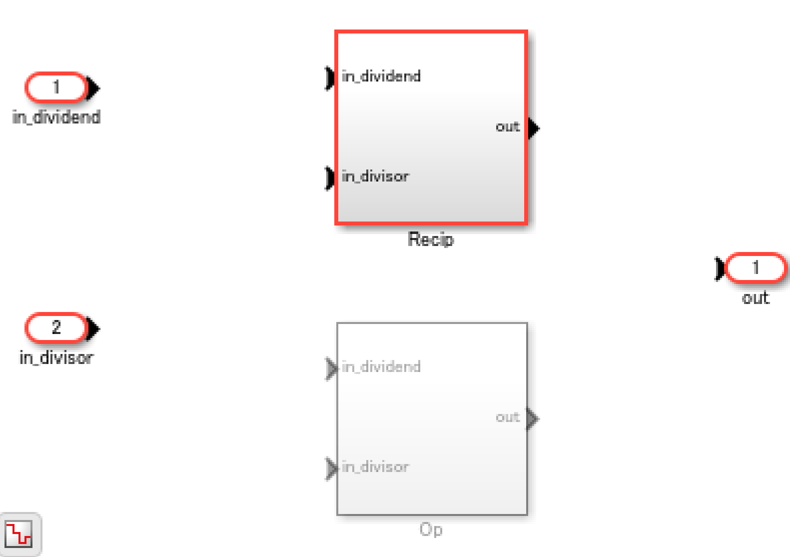
The Variant Subsystem block is a template preconfigured to contain two Subsystem blocks to use as Variant Subsystem choices. During simulation, a variant control decides which of the two Subsystem blocks is active. Therefore, you can use the Variant Subsystem to create two different configurations or subsystem behaviors and then specify the active configuration at the model compile stage or simulation-loop stage of simulation. For more information on the Variant Subsystem block, see Variant Subsystem, Variant Model, Variant Assembly Subsystem.

- You cannot use a Variant Subsystem as the DUT. To generate code, place the Variant Subsystem inside the Subsystem that you want to use as the DUT. The file name and instance name of the generated code is created only for the active configuration or both active and inactive configurations.
- You cannot share multiple Variant Subsystem blocks by using the Variant Subsystem optimization.
- You must make sure that when verifying the functionality of the generated code, the active variant that you used when simulating the model is the same as the active variant that you used for generating HDL code.

For an example, open the model `hdlcoder_variant_subsystem_design.slx`. If you open the DUT Subsystem, you see a Variant Subsystem block, `Divide`. The Variant Subsystem has two different subsystems, `Recip` and `Op`. If you open the Block Parameters dialog box for the `Divide` Subsystem, you see the **Variant control expression** and the **Condition** that determines which Subsystem to enable during simulation. The **Variant activation time** determines at which stage of simulation to enable the Subsystem. In this case, `Rcp` is `1` and the activation time is `update diagram`. The `Recip` Subsystem becomes active during simulation.

```
load_system('hdlcoder_variant_subsystem_design')
set_param('hdlcoder_variant_subsystem_design','SimulationCommand','Update')
open_system('hdlcoder_variant_subsystem_design/DUT/Divide')
```





Generated Code

To generate HDL code, run this command:

```
makehdl('hdlcoder_variant_subsystem_design/DUT');
```

An HDL file with the name `Recip.vhd` is generated because the `Recip` Subsystem is active at code generation time and therefore has code generated for it.

To generate HDL code for both `Recip` and `Op`, change the settings of these parameters on the Block Parameters dialog box of the `Divide` Subsystem, then run the `makehdl` command.

1. Set **Variant activation time** to `startup`.

```
set_param('hdlcoder_variant_subsystem_design/DUT/Divide','VariantActivationTime','startup')
```

2. Set the variant control variable `Rcp` of type `Simulink.VariantControl`, normal MATLAB® variable, or `Simulink.Parameter` and set its value to an integer value. In this example, `Rcp` is a normal MATLAB variable with an `int16` value.

```
Rcp = int16(1);
```

3. Set **Allow zero active variant controls** to `on`.

```
set_param('hdlcoder_variant_subsystem_design/DUT/Divide','AllowZeroVariantControls','on')
```

4. Set **Propagate conditions outside of variant subsystem** to `off`.

```
set_param('hdlcoder_variant_subsystem_design/DUT/Divide','PropagateVariantConditions','off')
```

Note: Ensure that **Treat as atomic unit** of `Recip` and `Op` Subsystems is set to `on`.

```
set_param('hdlcoder_variant_subsystem_design/DUT/Divide/Recip','TreatAsAtomicUnit','on')
set_param('hdlcoder_variant_subsystem_design/DUT/Divide/Op','TreatAsAtomicUnit','on')
```

Generate the HDL code using the following command.

```
makehdl('hdlcoder_variant_subsystem_design/DUT');
```

HDL files with the name `Recip.vhd` and `Op.vhd` are generated because the code is generated for both the `Recip` and `Op` Subsystems.

HDL Coder generates the following VHDL® code for the Variant Subsystem `Divide`, which has its variant control variable set to a tunable parameter, `Rcp`. The code generator creates a DUT port and adds a comment to indicate that the port corresponds to a tunable parameter. You can change the value of `Rcp` by providing the value to the pin on the chip that is mapped to `Rcp`.

```
ENTITY Divide IS
  PORT( clk        :   IN    std_logic;
    reset          :   IN    std_logic;
    enb            :   IN    std_logic;
    in_dividend    :   IN    std_logic_vector(15 DOWNTO 0);  -- sfix16_En14
    in_divisor     :   IN    std_logic_vector(15 DOWNTO 0);  -- sfix16_En13
    Rcp            :   IN    std_logic_vector(23 DOWNTO 0);  -- sfix24_En9 Tunable port
    out_rsvd       :   OUT   std_logic_vector(23 DOWNTO 0)  -- sfix24_En9
    );
END Divide;
```

```
VariantMerge_For_Outport_out_out1 <= Op_out1_signed WHEN Selected_Index_floor = to_signed(16#0001
  Recip_out1_signed WHEN Selected_Index_floor = to_signed(16#0002#, 15) ELSE
  GroundForVM_out_out1;
```

Model References: Build Model Design Using Smaller Partitions

Guideline ID

2.4.4

Severity

Recommended

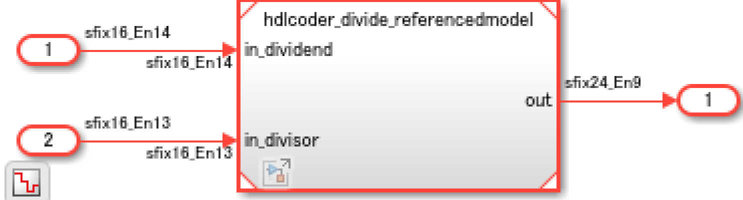
Description

Modeling a large design hierarchically can increase code generation time. If you specify generation of reports such as the traceability report, the code generation time can further increase significantly. To avoid such performance issues, it is recommended that you partition your design into smaller partitions. Use the Model block to unify a model that consists of smaller partitions. It also enables incremental code generation. You can generate HDL code for the parent model or the referenced model. To see the generated HDL code, in the `hdlsrc` folder, a folder is created for the parent model with a separate subfolder for the referenced model.

When generating the HDL test bench, if the test bench consists of blocks that operate with a continuous sample time, you can convert the DUT to a referenced model. This conversion enables the DUT to run at a fixed-step, discrete sample time. To learn more, see “Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks” on page 18-37.

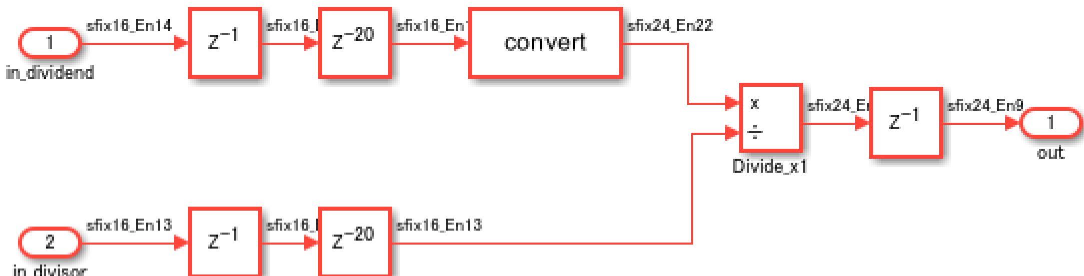
For an example, open the model `hdlcoder_divide_parentmodel.slx`. When you double-click the DUT Subsystem, you see a Model block that references the model `hdlcoder_divide_referencedmodel`.

```
load_system('hdlcoder_divide_parentmodel')
set_param('hdlcoder_divide_parentmodel', 'SimulationCommand', 'Update')
open_system('hdlcoder_divide_parentmodel/DUT')
```



To see the referenced model, double-click the Model block:

```
open_system('hdlcoder_divide_parentmodel/DUT/Model')
```



Copyright 2018-2021 The MathWorks, Inc.

To generate HDL code, enter this command:

```
makehdl('hdlcoder_divide_parentmodel/DUT')
```

For more information, see “Model Referencing for HDL Code Generation” on page 25-2.

Block Settings of Enabled and Triggered Subsystems

Guideline ID

`2.4.5`

Severity

Mandatory

Description

A Triggered Subsystem is a subsystem that receives a control signal via a Trigger block. The Triggered Subsystem executes for one cycle each time a trigger event occurs. When you generate HDL code for a triggered subsystem:

- Do not use the Triggered Subsystem block as the DUT. Place the Triggered Subsystem inside another Subsystem block, and use that Subsystem as the DUT.
- You can add unit delays to the output signals of the Triggered Subsystem, outside of the Triggered Subsystem block. The unit delays prevents HDL Coder from inserting additional bypass registers in the HDL code.
- Make sure that the **Use trigger signal as clock** setting does not result in timing mismatches when you simulate the testbench to verify the generated code. To learn more, see “Using Triggered Subsystems for HDL Code Generation” on page 20-19.

For other preferences when configuring the Triggered Subsystem block for HDL code generation, see “HDL Code Generation” on the Triggered Subsystem page.

An Enabled Subsystem is a subsystem that receives a control signal via an Enable block. The Enabled Subsystem executes at each simulation step where the control signal has a positive value. When you generate HDL code for an Enabled Subsystem:

- Do not use the Enabled Subsystem block as the DUT. Place the Enabled Subsystem inside another Subsystem block, and use that Subsystem as the DUT.
- You can add a State Control block in `Synchronous` mode inside the Enabled Subsystem to generate more efficient and hardware-friendly HDL code. The State Control block in `Synchronous` mode prevents HDL Coder from inserting additional bypass registers in the HDL code. The State Control block converts the Enabled Subsystem block to an Enabled Synchronous Subsystem block. To learn more, see “Synchronous Subsystem Behavior with the State Control Block” on page 25-75.
- If you want a State Control block in `Classic` mode or do not want a to add a State Control block, you can add unit delays to the output signals of the Enabled Subsystem, outside of the Enabled Subsystem block, to prevent HDL Coder from inserting additional bypass registers in the HDL code.

For other preferences when configuring the Enabled Subsystem block for HDL code generation, see “HDL Code Generation” on the Enabled Subsystem page.

See Also

Functions
makehdl

Model Settings
Use trigger signal as clock

Related Examples

- “Resettable Subsystem Support in HDL Coder” on page 25-87

More About

- “Using Enabled and Triggered Subsystems”

Usage of Rate Change and Constant Blocks

You can follow these guidelines to learn how to use blocks that can perform rate conversions in your model and blocks from the Sources library such as Constant blocks in your design. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Usage of Rate Conversion Blocks

Guideline ID

`2.5.1`

Severity

Recommended

Description

There are various ways in which you can model rate transitions. How you model rate transitions determine the timing and resource requirements of your design. This guideline shows various approaches for modeling rate transitions.

Increasing the Sample Rate

This table illustrates the blocks that you can use to increase the sample rate of your design. When you use these blocks, leave the block parameters to the default settings.

Rate Conversion Approach

Block	Generates Bypass Register?	Generates Zero Padding?	Notes
Repeat	No	No	To use this block, you must have DSP System Toolbox installed.
Rate Transition	No	No	None
Upsample	Yes	Yes	To use this block, you must have DSP System Toolbox installed. When you use this block, consider the impact of the bypass register and the logic that inserts zero padding on hardware resource usage.

For the Rate Transition block, to upsample the input signal without incurring a unit delay, in the Block Parameters dialog box of the Rate Transition block:

- Clear the **Ensure data integrity during data transfer** check box.

  Clearing this check box makes the **Ensure deterministic data transfer (maximum delay)** check box to disappear.

- Configure the output port sample time of the block to be an integer multiple of the input port sample time. Specify a fractional value of `1/n` for **Sample time multiple** where `n` is an integer. You can choose any value for the block parameter **Output port sample time options** as long as **Sample time multiple** uses a value `1/n`.

When the input and output clocks are not synchronous to each other, to avoid insertion of a bypass register in the HDL code generated for the Repeat and Rate Transition blocks, insert one unit delay following the Repeat and Rate Transition blocks in your model.

You can use the **Initial Condition** parameter of the block when increasing the sample rate. This parameter is used for upsampling when you leave the **Ensure data integrity during data transfer** check box selected. The initial value is propagated to the generated HDL code for the state that is created from a Rate Transition block.

Decreasing the Sample Rate

To reduce the sample rate, you can use a Downsample or a Rate Transition block. To use the Downsample block, you must have DSP System Toolbox installed. When you use these blocks, leave the block parameters to the default settings.

When downsampling the input signal, use the Rate Transition block because you can leave the block parameters to the default settings for HDL code generation. The **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** check boxes must be left selected. This mode generates an additional bypass register in the HDL code.

You can use the **Initial Condition** parameter of the block when decreasing the sample rate. When downsampling, this parameter is implemented as a bypass register and is not seen because it passes from the input to the output at initial clock cycle. The initial value is propagated to the generated HDL code for the state that is created from a Rate Transition block.

Use Discrete and Finite Sample Time for Constant Block

Guideline ID

`2.5.2`

Severity

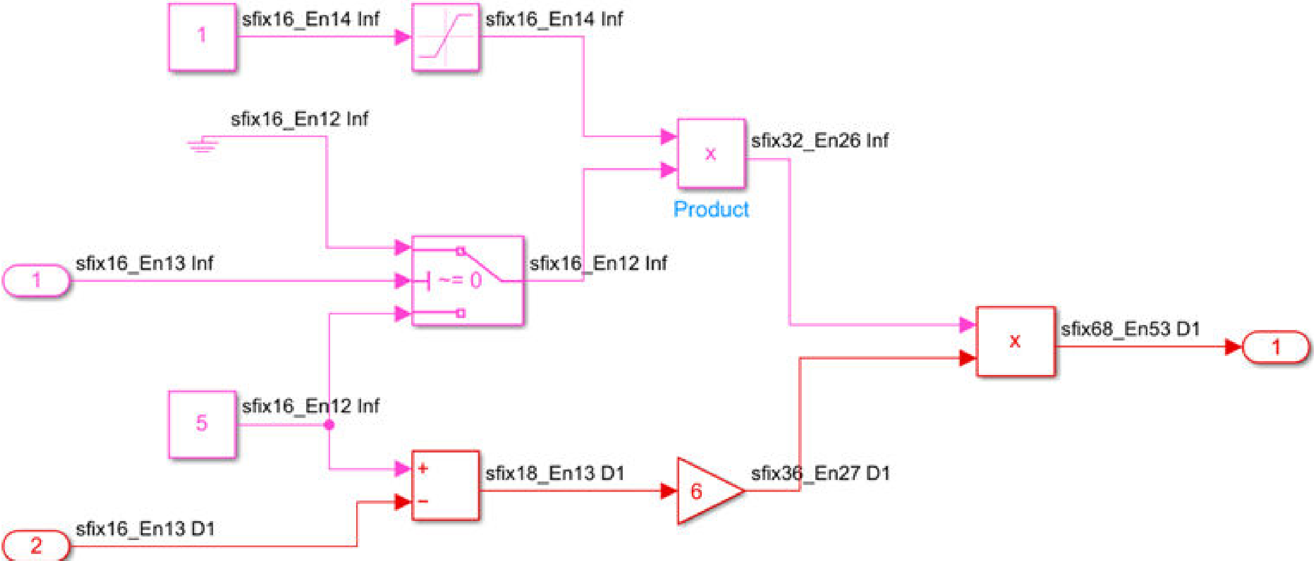
Recommended

Description

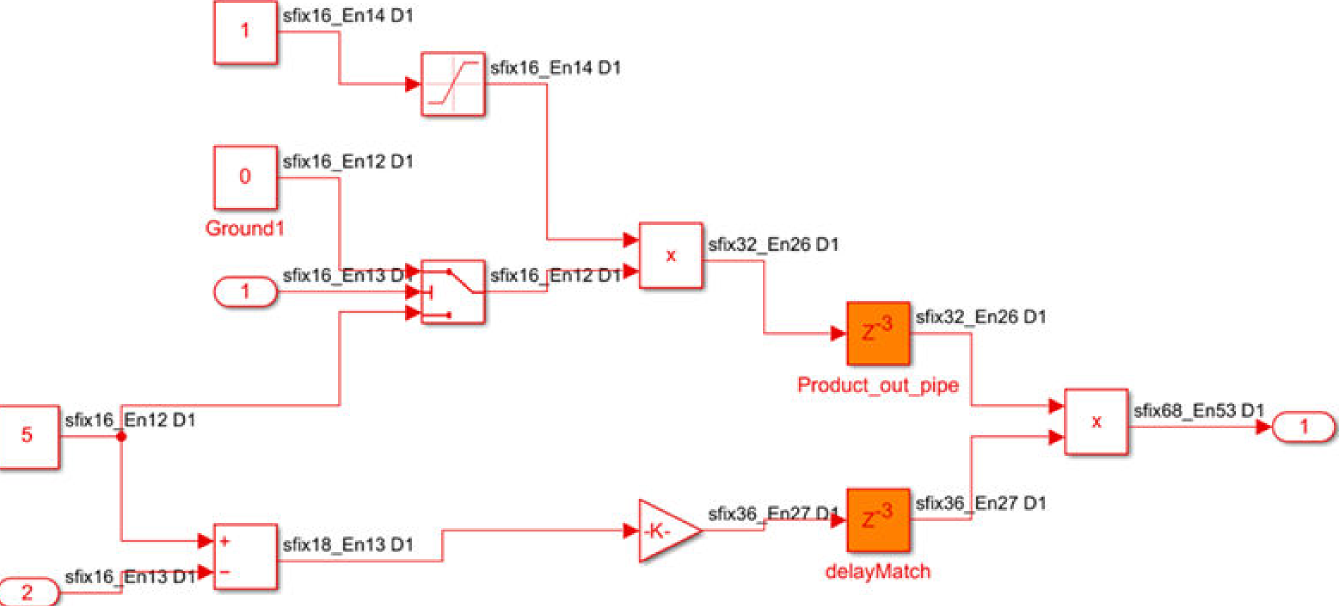
By default, the **Sample time** parameter of a Constant block is `inf`. When you use the Constant block, set the **Sample time** to `-1`. To identify Constant blocks that have infinite sample time in your design, in the Simulink model window, In the **Debug** tab, on the **Information Overlays > Sample Time** section, select **Colors**.

If you enable optimizations that add or handle latency in your design, such as streaming, sharing, clock-rate pipelining, or delay balancing, and then generate HDL code, HDL Coder resolves the sample times of Constant blocks with **Sample time** set to `inf`, if the sample times do not propagate to the device under test (DUT) output. For example, in this model DUT, you can generate HDL code with optimizations that add latency because the Product block merges the Constant and Ground blocks that have infinite sample times into a signal that has a fixed sample time, before to the DUT

output. The Sample Time Legend displays the infinite sample times in pink and the fixed sample times in red.



When the HDL block property **OutputPipeline** of the Product block is 3 and you generate HDL code and a generated model, the generated model shows the infinite sample times resolved to fixed sample times with the output pipeline delays applied and the balanced delays applied on the other signal path.



For blocks that have `inf` sample times propagate to the DUT output, you can identify and change the sample time of the blocks to `-1` by using either of these approaches:

- Run a script that can programmatically change the sample time of the blocks to `-1`. For an example script, see “Identify and Programmatically Change and Display HDL Block Parameters” on page 18-31.
- Run the check “Check for infinite and continuous sample time sources” on page 37-15 in the HDL Code Advisor. If running the check fails, it displays sources such as Constant blocks that have infinite sample time. Select **Modify Settings** to update the sample time to `-1` or to inherit via backpropagation.

Dynamically Change Sample Offset for Downsample Block

Guideline ID

`2.5.3`

Severity

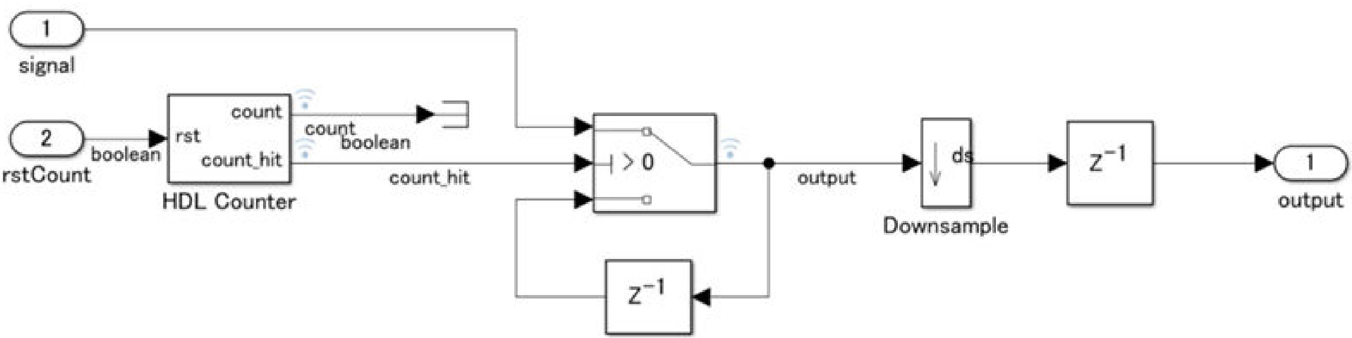
Informative

Description

You can use the **Sample offset** parameter to adjust the sample offset for the Downsample block. However, when you set the offset using this block parameter, the sample offset cannot be changed dynamically during simulation.

To change the sample offset for the Downsample block dynamically, you can adjust the sample offset for the input signal using the Simulink block instead of the **Sample offset** parameter.

This figure shows a model for a dynamically changing sample offset of the Downsample block by using HDL Counter and Switch blocks. The model uses HDL Counter block with the reset signal to latch the input signal at the desired downsampling period. Adjust the sample offset by varying the reset timing of the counter. In this model, a Delay block connects to the output port to suppress the generation of bypass registers.



You can also use a Rate Transition block in place of the Downsample block if you do not have the DSP System Toolbox licence for using Downsample block.

See Also

Functions
`makehdl`

More About

- "Multirate Model Requirements for HDL Code Generation" on page 20-7
- "Code Generation from Multirate Models" on page 20-2

Guidelines for Using Delays and Goto and From Blocks for HDL Code Generation

These guidelines illustrate the recommended settings for modeling delays in your model. You model delays by using blocks available in the **Discrete** Library. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see "HDL Modeling Guidelines Severity Levels" on page 18-3.

Appropriate Usage of Delay Blocks as Registers

Guideline ID

`2.6.1`

Severity

Recommended

Description

For blocks in your model to be inferred as a flipflop on the target FPGA, use Delay blocks instead of memory blocks. You can specify a local reset and enable signal for each Delay block.

By default, the **Delay length** of the block is set to `2`. In this case, the input to the block passes to the output after two time steps. If the **Delay length** is set to `0`, the input passes to the output without any delay. The generated HDL code treats the block as a wire. To infer a flipflop or register on the target device, set the **Source** to `dialog` and specify a **Delay length** greater than zero.

When using a Delay block that has an external enable port or that is inside the Enabled Subsystem block, it is recommended to use the State Control block in synchronous mode. Similarly, when you use a Delay block that has an external reset port or that is inside the Resettable Subsystem block, it is recommended to use the State Control block in Synchronous mode. To learn more, see HDL Code Generation in Delay.

Do not use the Unit Delay Enabled, Unit Delay Resettable, and Unit Delay Enabled Resettable blocks for HDL code generation. These blocks have been removed. Instead, replace these blocks with the Unit Delay Enabled Synchronous, Unit Delay Resettable Synchronous, and Unit Delay Enabled Resettable Synchronous blocks. These blocks use the State Control block in synchronous hardware mode. To perform this block replacement in your model, run the model check "Check for obsolete Unit Delay Enabled/Resettable Blocks" on page 37-20.

Absorb Delays to Avoid Timing Difference

Guideline ID

`2.6.2`

Severity

Recommended

Description

Certain block implementations, floating-point operations, and optimization settings such as distributed pipelining introduce latency in the generated HDL code and the generated model. The additional latency results in a timing difference between the original model and the generated model. To avoid this timing difference, such as when you are using a control system with feedback loop, use a modeling pattern that can absorb delays. To absorb the delays:

- Place a Delay block after the block that is introducing latency.
- Set the Delay block **Delay length** value equal to the block latency.

By adding the Delay block to your original model, you can simulate your original model with latency.

These blocks can introduce latency:

- Divide, Sqrt, and Reciprocal blocks that have custom latency value greater than zero.
- Trigonometric Function blocks that have **Function** set to `sin`, `cos`, `sincos`, `cos+jsin`, or `atan2` and HDL architecture set to `CORDIC`
- Native floating-point operators that have **LatencyStrategy** set to `Max`, `Min`, or a custom value greater than zero.

For more information, see "Use Delay Absorption While Modeling with Latency" on page 21-86.

Map Large Delays to Block RAM

Guideline ID

`2.6.3`

Severity

Recommended

Description

To save area, when your design contains large design delays and pipeline delays, you can map the delays to block RAM and UltraRAM resources on the FPGA. Design delays are delays that you manually insert in your design by using Delay blocks, or other blocks that have state including Queue, HDL FIFO, or Buffer blocks. Pipeline delays are delays that are generated by optimization settings or block implementation settings such as Newton-Raphson implementation.

To map design delays to RAM:

- In the HDL Block Properties dialog box of Delay blocks, set **UseRAM** to `on`. To learn how you can set this option on Delay blocks in your design programmatically, see "Set HDL Block Parameters for Multiple Blocks Programmatically" on page 19-52.
- To map significantly large delays to UltraRAM resources, you can specify the `ram_style` attribute in the generated HDL code.

```
-- This VHDL code shows the ramstyle attribute set to ultra:

 attribute ram_style: string;
 attribute ram_style of ram : signal is "ultra";
```

```
// This Verilog code shows the ramstyle attribute set to ultra:

(* ram_style = "ultra" *)
```

- As described in “Effects of Streaming and Distributed Pipelining” on page 19-27, even if **UseRAM** is `off`, you can map large delays that exceed a threshold value by using the RAM mapping threshold parameter. You can change this threshold value depending on how large a delay you want to map to RAM.
- When you use MATLAB Function blocks, you can map persistent variables in your MATLAB code to RAM by setting HDL architecture to `MATLAB Datapath` and `MapPersistentVarsToRAM` HDL block property to `on`.

For pipeline delays that are inserted by optimizations, delay balancing automatically inserts matching delays in parallel paths. If the delay length at the critical path and the number of vector elements in the parallel path take large values, the pipeline delays can also become significantly large.

To map these large delays to Block RAM:

- Enable the Map pipeline delays to RAM parameter.
- Adjust the **RAM mapping threshold (bits)** parameter to a value that is smaller than the required RAM size.

  To calculate the total RAM size, use this formula:

  ```
  RAMSize = Delay length * Word length * Vector length * Complexity
  ```

  `Complexity` is 2 for a complex data type or 1 for a real datatype.

For more information, see “Apply RAM Mapping to Optimize Area” on page 21-120.

Required HDL Settings for Goto and From Blocks

Guideline ID

`2.6.4`

Severity

Mandatory

Description

When you generate HDL code for the DUT Subsystem that uses From and Goto blocks:

- Do not use From and Goto blocks across the boundary of the DUT subsystem. To connect the input and output ports of the DUT, use Inport and Outport blocks instead.
- Do not use From and Goto blocks across the boundary of an Atomic Subsystem. To connect the input and output ports of the DUT, use Inport and Outport blocks instead.

Using From and Goto blocks across a subsystem hierarchy can impact the readability of the model. Before generating HDL code, it is recommended that you use From and Goto blocks in the same subsystem and use `local` or `Scoped` visibility. When you generate HDL code, in the generated model, each Goto and From block becomes a pair of From and Goto subsystems connected back to back.

See Also

Functions
makehdl

More About

- “Goto and From Blocks as a Signal Routing Alternative”

Modeling Efficient Multiplication and Division Operations for FPGA Targeting

These guidelines illustrate the recommended settings when using Divide and Product blocks in your model for improved area and timing on the target FPGA. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Designing Multipliers and Adders for Efficient Mapping to DSP Blocks on FPGA

Guideline ID

`2.7.1`

Severity

Strongly Recommended

Description

Digital signal processing (DSP) algorithms use several multipliers and accumulators. FPGA devices provided by vendors such as Xilinx® and Intel® contain dedicated DSP slices. These small size, high speed, DSP slices contain several multipliers and accumulators that make FPGA devices best suited for DSP applications.

The architecture of DSP slices varies widely across the different FPGA vendors and across different families of devices provided by the same vendor. To map your Simulink® model containing adders, multipliers, and delays to DSP slices, adapt your model to the DSP slice architecture by taking into consideration:

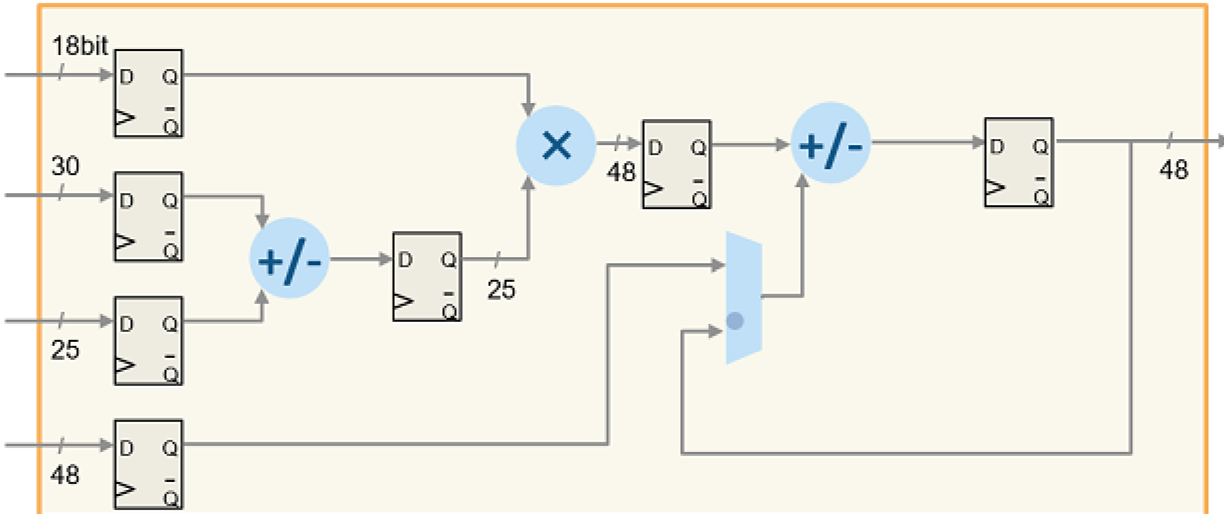
- Arrangement of flipflops, adders, and multipliers in the DSP slice.
- Rounding and saturation settings.
- Bit widths of the adders and multipliers. For efficient mapping, use bit widths in your model that are less than or equal to the bit widths of the DSP unit.

When the bit widths in your model become larger than the bit widths of the DSP, your design does not fit onto one DSP. In this case, multiple DSPs or additional logic is required.

You can map these blocks in your model to DSP blocks on an FPGA:

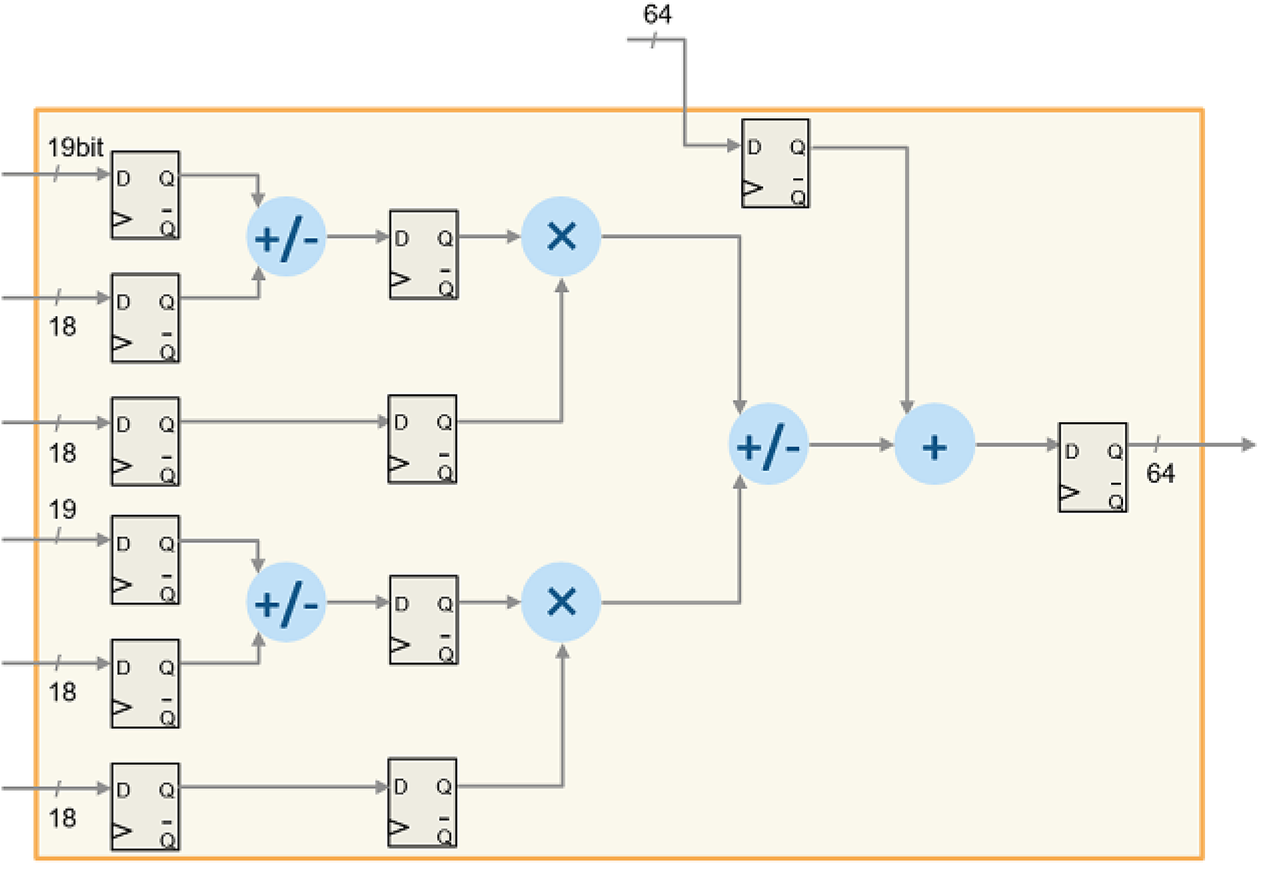
- Add and Sum
- Delay
- Product
- Multiply-Add
- Multiply-Accumulate

This figure illustrates the Xilinx DSP architecture. Xilinx 7 series FPGAs have dedicated DSP slices that use this architecture. The DSP architecture consists of input registers, pre-adder, 25x18 multiplier, intermediate registers, post-adder, and an output register.



For more information, see **DSP48E1 Slice Overview** in the Xilinx documentation.

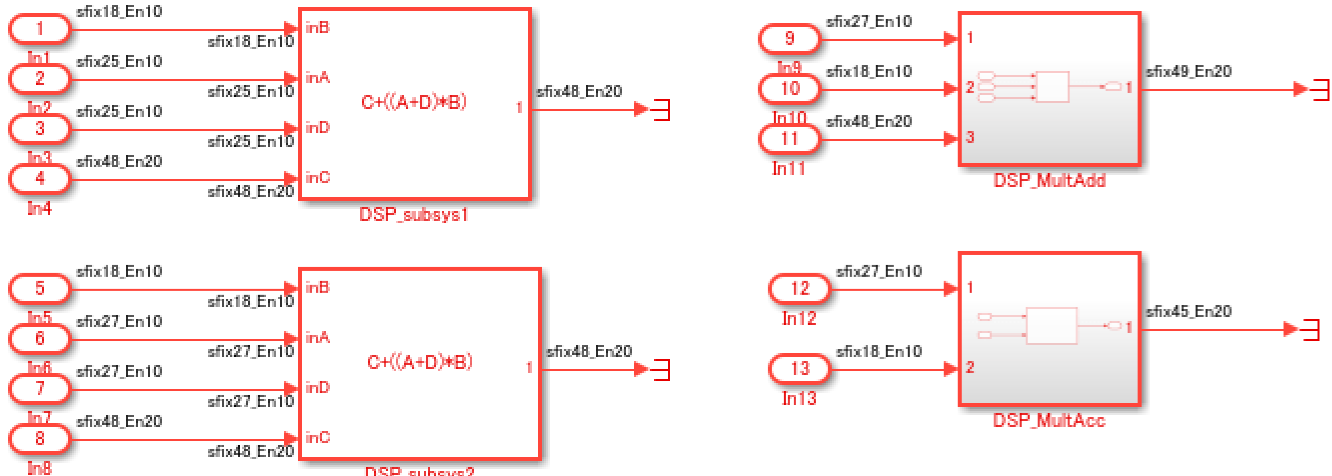
This figure illustrates the Intel DSP architecture. This DSP architecture for Stratix® V devices is a variable precision DSP architecture. The DSP blocks can have bit widths of 9, 18, 27, and 36 bits, and 18x25 complex multiplication for FFTs.



For more information, see DSP Block Architecture of Stratix V devices.

To learn how you can design your algorithm to map to this DSP unit, open the model `hdlcoder_multiplier_adder_dsp.slx`

```
open_system('hdlcoder_multiplier_adder_dsp')
set_param('hdlcoder_multiplier_adder_dsp', 'SimulationCommand', 'Update')
```



Copyright 2014-2019 The MathWorks, Inc.

The model consists of two subsystems dsp_subsys1 and dsp_subsys2 that implement the operation C+((A+D)*B. You can also implement this operation by using Multiply-Add or Multiply-Accumulate blocks as illustrated by subsystems DSP_MultAdd and DSP_MultAcc.

`dsp_subsys1` implements the operation C+((A+D)*B by using bit widths that equal the DSP on a Xilinx 7 series FPGA. If you open the HDL Workflow Advisor and deploy this Subsystem onto a Xilinx Virtex® 7 FPGA, the entire design fits exactly onto one DSP slice.

Passed Synthesis

Parsed resource report file: DSP_subsys1_utilization_synth.rpt.

Resource summary	
Resource	**Usage**
Slice LUTs	0
Slice Registers	0
DSPs	1
Block RAM Tile	0
URAM	0

`dsp_subsys2` implements the same operation by using bit widths that are larger than the DSP on a Xilinx FPGA. If you deploy this Subsystem onto a Xilinx Virtex 7 FPGA, you see that the entire design fits onto one DSP slice and uses additional slice logic.

Passed Synthesis

Parsed resource report file: DSP_subsys2_utilization_synth.rpt.

Resource summary	
Resource	**Usage**
Slice LUTs	141
Slice Registers	210
DSPs	1
Block RAM Tile	0
URAM	0

Set ConstMultiplierOptimization HDL Block Property to auto for Gain Block

Guideline ID

`2.7.2`

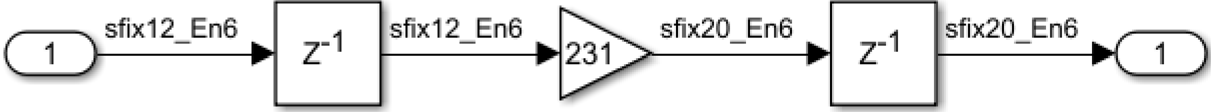
Severity

Strongly Recommended

Description

When you use a Gain block in your design, to achieve the most area-efficient implementation, set the "ConstMultiplierOptimization" on page 19-8 HDL block property to `auto`. The code generator chooses between `CSD` and `FCSD` implementations that yields the smallest circuit size and generates HDL code without using the multiplication (*) operator.

You can use this setting to avoid targeting DSP resources and reduce the number of logic circuits on Intel Quartus Prime when synthesizing your design on the target FPGA. For example, this table shows the generated HDL code for the Gain block depending on the HDL block property settings for the **ConstMultiplierOptimization** block.



ConstMultiplierOptimization Settings and Impact on Generated HDL Code

<table>
<tr><th>ConstMultiplierOptimization Setting</th><th>Operations</th><th>Generated HDL Code</th></tr>
<tr><td>CSD</td><td>Casts input data in parallel and adds or subtract the results.</td><td>This code shows the generated VHDL code.

<code>-- CSD Encoding(231): 1001'01001'; Cost (Adders)</code>
<code>DOUT_mul_temp <= ((resize(DIN & '0' & '0' & '0' &</code>
<code>& '0' & '0' & '0' & '0', 21) - resize(DIN & '0' &</code>
<code>& '0' & '0' & '0', 21)) + resize(DIN & '0' & '0'</code>
<code>'0', 21)) - resize(DIN, 21);</code>
<code>DOUT <= DOUT_mul_temp(19 DOWNTO 0);</code>

This code shows the generated Verilog code.

<code>// CSD Encoding (231) : 1001'01001'; Cost (Adders</code>
<code>assign Gain_1 = {DIN[11], {DIN, 8'b00000000}};</code>
<code>assign Gain_2 = {{4{DIN[11]}}, {DIN, 5'b00000}};</code>
<code>assign Gain_3 = {{6{DIN[11]}}, {DIN, 3'b000}};</code>
<code>assign Gain_4 = {{9{DIN[11]}}, DIN};</code>
<code>assign DOUT_mul_temp = ((Gain_1 - Gain_2) + Gain_</code>
<code>assign DOUT = DOUT_mul_temp[19:0];</code></td></tr>
<tr><td>FCSD</td><td>Adds the input data and its cast data in each cascading.</td><td>This code shows the generated VHDL code.

<code>-- FCSD for 231 = 33 X 7; Total Cost = 2</code>
<code>-- CSD Encoding (33) : 0100001; Cost (Adders) = 1</code>
<code>Gain_factor <= resize(DIN & '0' & '0' & '0' &</code>
<code>'0' & '0', 21) + resize(DIN, 21);</code>

<code>-- CSD Encoding (7) : 1001'; Cost (Adders) = 1</code>
<code>DOUT_mul_temp <= resize(Gain_factor & '0' &</code>
<code>'0' & '0', 21) - Gain_factor;</code>
<code>DOUT <= DOUT_mul_temp(19 DOWNTO 0);</code>

This code shows the generated Verilog code.

<code>// FCSD for 231 = 33 X 7; Total Cost = 2</code>
<code>// CSD Encoding (33) : 0100001; Cost (Adders) = 1</code>
<code>assign Gain_3 = {{4{DIN[11]}}, {DIN, 5'b00000}};</code>
<code>assign Gain_4 = {{9{DIN[11]}}, DIN};</code>
<code>assign Gain_factor = Gain_3 + Gain_4;</code>

<code>// CSD Encoding (7) : 1001'; Cost (Adders) = 1</code>
<code>assign Gain_1 = {Gain_factor, 3'b000};</code>
<code>assign Gain_2 = Gain_1[20:0];</code>
<code>assign DOUT_mul_temp = Gain_2 - Gain_factor;</code>
<code>assign DOUT = DOUT_mul_temp[19:0];</code></td></tr>
<tr><td>auto</td><td>Selects CSD or FCSD implementation that uses fewer adders.</td><td>Generated HDL code is same as CSD or FCSD implementation.</td></tr>
</table>

ConstMultiplierOptimization Setting	Operations	Generated HDL Code

none	Uses multiplication operator (*).	This code shows the generated VHDL code.
		`DOUT_mul_temp <= to_signed(2#011100111#, 9) * DIN`
		`DOUT <= DOUT_mul_temp(19 DOWNTO 0);`
		This code shows the generated Verilog code.
		`assign DOUT_mul_temp = 231 * DIN;`
		`assign DOUT = DOUT_mul_temp[19:0];`

Use ShiftAdd Architecture of Divide Block for Fixed-Point Types

Guideline ID

2.7.3

Severity

Recommended

Description

When you use fixed-point data types as inputs to the Divide block, specify the HDL Architecture of the block as `ShiftAdd`. In this architecture, the block computes the result by using multiple shift and add operations. The operations are pipelined to achieve higher clock frequencies on the target FPGA device.

When you use floating-point data types as inputs to the Divide block, select **Use Floating Point** configuration parameter.

Use Gain Block for Fixed-Point Constant Operations

Guideline ID

2.7.4

Severity

Strongly Recommended

Description

When you use the Constant block configured with fixed-point data types as an input to the Product, Divide, or Trigonometric operation blocks, you can replace those operators with a Gain or Constant block to reduce the circuit area of your design. You can use these design considerations for blocks with constant input:

- When one input of the Product block is a Constant block, replace the Product block with a Gain block and set the constant value as the **Gain** value of the Gain block.

- When the divisor input of the Divide block is a Constant block, replace the Divide block with a Gain block and set the reciprocal of the constant value to the **Gain** value of the Gain block.
- When the input to the Trigonometric Function block, such as `sin`, `cos`, or `atan2` is a Constant block, replace the block with a Constant block and set the calculated values to the Constant block. For example, if the input to the Sin block is `π/3`, replace the Sin block with a Constant block and set the constant value to `sin(pi/3)`.

The circuit area required for variable and constant operations are different. Using the constant operations can reduce the circuit area of your design.

See Also

Functions
`makehdl`

Blocks
Multiply-Add | Multiply-Accumulate

More About

- “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111

Using Persistent Variables and fi Objects Inside MATLAB Function Blocks for HDL Code Generation

These guidelines illustrate the recommended settings when using persistent variables inside MATLAB Function blocks in your model. The MATLAB Function block is available in the **User-Defined Functions** block library. A persistent variable in a MATLAB Function block acts similar to a delay element in your Simulink model.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Update Persistent Variables at End of MATLAB Function

Guideline ID

2.8.1

Severity

Strongly Recommended

Description

To make sure that the persistent variables inside the MATLAB Function block map to a register on the target FPGA device, update the persistent variable at the end of the MATLAB code inside the MATLAB Function block. Do not update the persistent variable before its value is read or used by the function.

For example, this MATLAB code is not recommended because the function updates the persistent variable `FF0` is updated before the value is read at the output.

```
function FF_out0 = fcn(FF_in)
%#codegen

persistent FF0

if isempty(FF0)
    FF0 = zeros(1, 'like', FF_in);
end

% Incorrect order of FF update
FF0 = FF_in

% Output FF0. FF_out0 is NOT delayed
FF_out0 = FF0;
```

This MATLAB code is recommended because the value is written to `FF0` at the end of the code.

```
function FF_out0 = fcn(FF_in)
%#codegen

persistent FF0

if isempty(FF0)
    FF0 = zeros(1, 'like', FF_in);
```

```
end

% Output FF0
FF_out0 = FF0;

% Write FF update at the end of the code
FF0 = FF_in
```

Avoid Algebraic Loop Errors from Persistent Variables inside MATLAB Function Blocks

Guideline ID

2.8.2

Severity

Mandatory

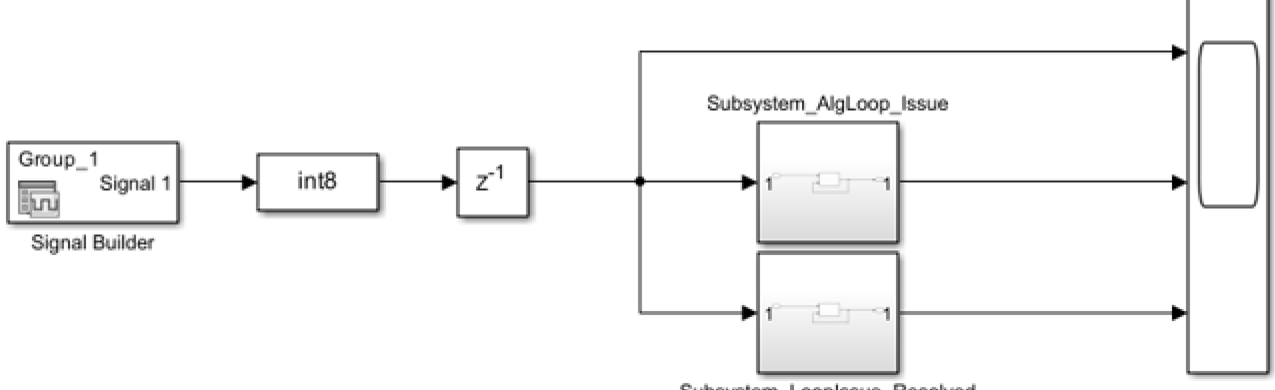
Description

When your Simulink® model contains MATLAB Function blocks inside a feedback loop and uses persistent variables, compiling or simulating the model might generate algebraic loop errors. To simulate the model and generate HDL code, use nondirect feedthrough.

In certain cases, the persistent delay in the MATLAB Function block inside a feedback loop causes an algebraic loop error. When you use direct feedthrough, the output of the block directly depends on the input. When **Allow direct feedthrough** is cleared, the output of the block depends on the internal state and properties and does not depend on the input. The nondirect feedthrough semantics prevents algebraic loops errors by making the outputs depend only on the state.

For an example, open the model `hdlcoder_MLFB_avoid_algebraic_loops`.

```
modelname = 'hdlcoder_MLFB_avoid_algebraic_loops';
blkname = 'hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLoop_Issue/MATLAB Function1';
open_system(modelname)
```



When you simulate the model, the algebraic loop error message is displayed. The MATLAB Function block `hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLoop_Issue/MATLAB Function` uses a persistent variable inside a MATLAB Function block.

```
open_system(blkname)
```

```
function y = fcn(u0, u1)
%#codegen

persistent tmp
if isempty(tmp)
    tmp = int8(0);
end

% There is a delay but algebraic loop is detected
y = tmp;

if u0 ~= 0
    tmp = int8(u1 + u0);
else
    tmp = int8(u1);
end
```

To avoid this error, use nondirect feedthrough. To specify nondirect feedthrough at the command line, create a `MATLABFunctionConfiguration` object by using `get_param` function, and then change the property value `AllowDirectFeedthrough`:

```
MLFBConfig = get_param(blkname, 'MATLABFunctionConfiguration');
MLFBConfig.AllowDirectFeedthrough = 0;
```

See also `MATLABFunctionConfiguration`.

To specify nondirect feedthrough from the UI:

1. Open the MATLAB Function block `MATLAB Function1`.
2. Open the Property Inspector. In the **Modeling** tab, in the **Design** section, click **Property Inspector**.
3. In the **Property Inspector**, open the **Advanced** section and clear the **Allow direct feedthrough** check box.

See also “Prevent Algebraic Loop Errors in MATLAB Function, Chart, and Truth Table Blocks”.

The model now simulates without algebraic errors. You can now generate HDL code for the Subsystem block `Subsystem_AlgLoop_Issue`.

```
open_system(modelname)
set_param('hdlcoder_MLFB_avoid_algebraic_loops', 'SimulationCommand', 'Update')
makehdl('hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLoop_Issue')
```

Use hdlfimath Setting and Specify fi Objects inside MATLAB Function Block

Guideline ID

2.8.3

Severity

Strongly Recommended

Description

`fimath` properties define the rules for performing arithmetic operations on fi objects. To specify `fimath` properties that govern arithmetic operations, use a `fimath` object. To see the default `fimath` property settings, run this command:

```
F = fimath
```

```
F =


        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

The default `fimath` settings reduce rounding errors and overflows. However, HDL code generation for a MATLAB Function block that uses these settings can incur additional resource usage on the target FPGA device. To avoid the additional logic, use `hdlfimath`. The `hdlfimath` function is a utility that defines `fimath` properties optimized for HDL code generation. To see the default `hdlfimath` settings, run this command:

```
H = hdlfimath
```

```
H =


        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

HDL code generation for a MATLAB Function block that uses these settings avoids the additional resource usage and saves area on the target FPGA device.

To specify these settings for a MATLAB Function block:

- Double-click the MATLAB Function block.
- In the **Modeling** tab, in the **Design** section, click the **Property Inspector**.
- In the **Property Inspector**, open the **Fixed-point properties** section. For:

- **Treat these inherited Simulink signal types as fi objects**, select `Fixed-point & Integer`.

  If you use the default `Fixed-point` setting, fixed-point data types specified by using fi objects and built-in integer types such as `int8` and `int16` are treated differently. When you use built-in integer types, the output data type for integer type calculations becomes the same as the input data type. The bit width is not expanded to perform numeric calculation.
- **MATLAB FUNCTION FIMATH**, select **Specify Other** and then enter `hdlfimath`.

To perform rounding operations that are different from the default hdlfimath settings, specify these settings explicitly by using the fi object as illustrated below.

```
A = fi(4.9, 1, 8)

A =

    4.8750

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4

B = fi(2.3, 1, 10)

B =

    2.2969

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 10
        FractionLength: 7

C = fi(A+B, 'RoundingMethod', 'Nearest', 'OverflowAction', 'Saturate')

C =

    7.1719

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 12
        FractionLength: 7

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

To make sure that the fimath settings are specified according to hdfimath for the MATLAB Function block, you can run the check “Check for MATLAB Function block settings” on page 37-18.

See Also

Blocks
MATLAB Function

Functions
`makehdl` | `fimath`

More About

- “Design Guidelines for the MATLAB Function Block” on page 27-19
- “Initialize Persistent Variables in MATLAB Functions” on page 26-22
- “Bitwise Operations in MATLAB for HDL and HLS Code Generation” on page 1-58

Guidelines for HDL Code Generation Using Stateflow Charts

These guidelines illustrate the recommended settings when using Stateflow charts in your model. The Stateflow Chart block is available in the **Stateflow** block library. By using Stateflow charts, you can model delays in your Simulink model.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Choose State Machine Type based on HDL Implementation Requirements

Guideline ID

`2.9.1`

Severity

Strongly Recommended

Description

HDL Coder supports code generation for Mealy and Moore Stateflow charts. MATLAB Function block is also available to model `Mealy` or `Moore` state machines.

To specify whether you want a `Mealy` or `Moore` state machine, in the Chart (Stateflow) properties, specify the **State Machine Type**. Do not use `Classic` because it affects readability of the generated HDL code. Choose the **State Machine Type** depending on how you want the Stateflow semantics to map to a hardware implementation. See “Hardware Realization of Stateflow Semantics” on page 26-6.

When you use `Mealy` charts, the outputs depend on the current state and inputs. By using `Mealy` charts, you can more easily define state transitions which makes these charts more flexible to use. The generated HDL code from `Mealy` charts may be less readable.

For `Moore` charts, the outputs depend only on the current state. The generated HDL code from `Moore` charts is more readable. `Moore` charts restrict flexibility in defining state transitions.

Specify Block Configuration Settings of Stateflow Chart

Guideline ID

`2.9.2`

Severity

Strongly Recommended

Description

When you use Stateflow Chart (Stateflow) blocks in your model for HDL code generation, use these recommended settings:

- For **Action Language**, use MATLAB
- For **Update method**, use Discrete or Inherited. Do not use Continuous.

Moore Chart

- If you disable **Initialize Outputs Every Time Chart Wakes Up**, the generated HDL code includes additional registers for the state machine output values.
- Disable **Support Variable-Size Arrays**.

Mealy Chart

- If you disable **Initialize Outputs Every Time Chart Wakes Up**, the generated HDL code includes additional registers for the state machine output values.
- Disable **Enable Super Step Semantics**.
- Disable **Support Variable-Size Arrays**.
- Enable **Execute (enter) chart at initialization**.

To make sure that these settings are specified for the Stateflow Chart, you can run the check "Check for Stateflow chart settings" on page 37-19.

Insert Unconditional Transition State for Else Statement in HDL Code

Guideline ID

2.9.3

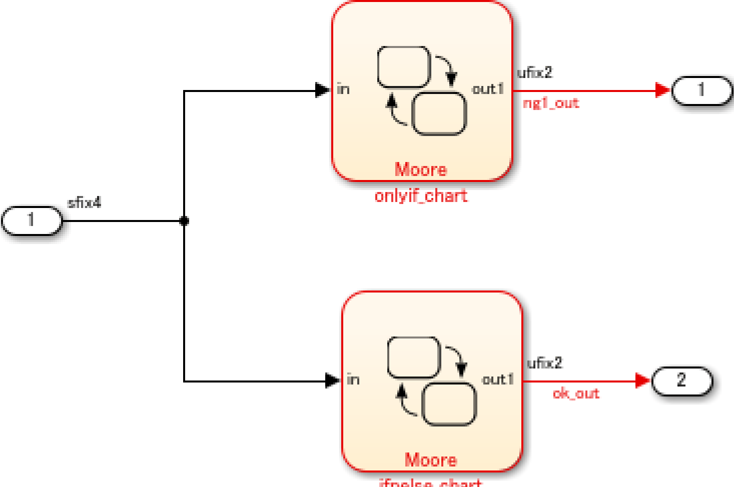
Severity

Recommended

Description

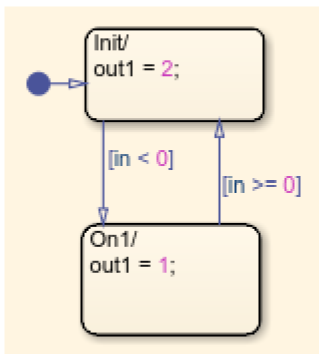
When you use Stateflow® charts for HDL code generation, insert unconditional states in the chart. The HDL code generated for such a chart contains an else branch with the if statement. The presence of an else branch prevents the third-party tool from inferring a latch when you deploy the HDL code. For example, open the model hdlcoder_chart_ifnelsecond.

```
open_system('hdlcoder_chart_ifnelsecond')
set_param('hdlcoder_chart_ifnelsecond', 'SimulationCommand', 'Update')
open_system('hdlcoder_chart_ifnelsecond/dut_chart')
```



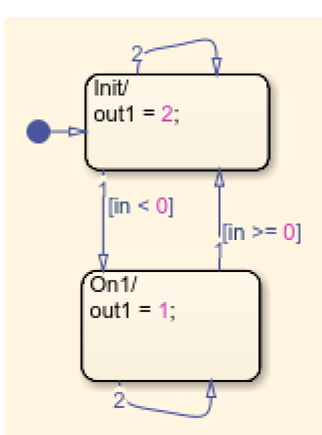
The model contains two Stateflow Moore Charts. The chart `onlyif_chart` implements a simple condition that outputs out1 based on in1.

```
open_system('hdlcoder_chart_ifnelsecond/dut_chart/onlyif_chart')
```



The Chart block `ifnelse_chart` is the same as `onlyif_chart` and has an unconditional transition state.

```
open_system('hdlcoder_chart_ifnelsecond/dut_chart/ifnelse_chart')
```



To generate HDL code for the DUT, run this command:

```
makehdl('hdlcoder_chart_ifnelsecond/dut_chart')
```

The HDL code generated for the `onlyif_chart` does not contain an else condition. Do not deploy this code to a target device because synthesis tools might infer a latch.

```
case (is_onlyif_chart)
  is_onlyif_chart_IN_Init :
    begin
      if (in < 4'sb0000) begin
        is_onlyif_chart_temp = is_onlyif_chart_IN_On1;
      end
    end
  default :
    begin
      //case IN_On1:
      if (in >= 4'sb0000) begin
        is_onlyif_chart_temp = is_onlyif_chart_IN_Init;
      end
    end
endcase
is_onlyif_chart <= is_onlyif_chart_temp;
```

The HDL code generated for the `ifnelse_chart` contains an else statement for the unconditional transition state. This code is recommended for deployment to the target FPGA device.

```
case (is_ifnelse_chart)
  is_ifnelse_chart_IN_Init :
    begin
      if (in < 4'sb0000) begin
        is_ifnelse_chart_temp = is_ifnelse_chart_IN_Onl;
      end
      else begin
        is_ifnelse_chart_temp = is_ifnelse_chart_IN_Init;
      end
    end
  default :
    begin
      //case IN_Onl:
      if (in >= 4'sb0000) begin
        is_ifnelse_chart_temp = is_ifnelse_chart_IN_Init;
      end
      else begin
        is_ifnelse_chart_temp = is_ifnelse_chart_IN_Onl;
      end
    end
endcase
is_ifnelse_chart <= is_ifnelse_chart_temp;
```

Data Type Settings and Casting in Stateflow Chart for HDL Code Generation

Guideline ID

`2.9.4`

Severity

Informative

Description

When you do not explicitly specify the data type of state and output variables in a Stateflow Chart that has `MATLAB` as the **Action Language**, the data type becomes double. If variables that have different data types are assigned to these Chart variables, a data type mismatch can occur, which can lead to simulation errors.

To avoid simulation errors, explicitly initialize the data type of the Chart variables by using fi objects, or specify their data type in Model Explorer. When performing assignments to variables of different types, you can perform data type conversion and initialization by using fi objects.

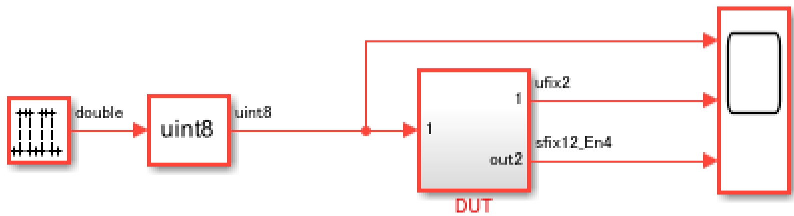
When you want to change the fi data type, you must specify the word lengths and fraction lengths for the types. For frequent data type conversions, you can instead cast the type to fi data types by using subscript(:). In this case, you can replace data type conversions with cast by subscript(:) in the Model

Explorer. The value of the substitution source is then type-converted to that of the substitution target variable.

Note: For assignments to intermediate variables, you do not have to cast the data type.

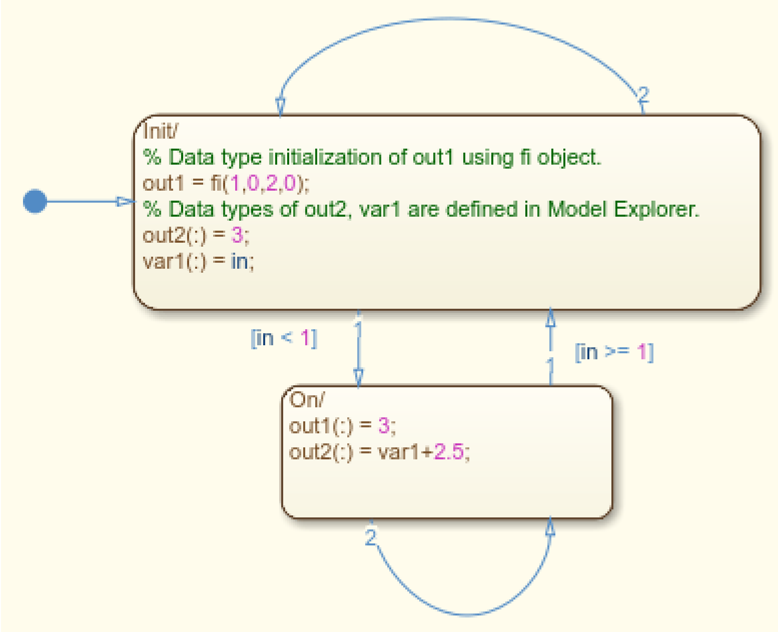
For an example that shows the different data type initialization methods, open the model `hdlcoder_chart_datatype_casting`.

```
open_system('hdlcoder_chart_datatype_casting')
set_param('hdlcoder_chart_datatype_casting', 'SimulationCommand', 'Update')
```



The `DUT` subsystem contains a Moore chart that shows how the output variables `out1` and `out2` and an internal variable `var1` are defined. `out1` type is explicitly specified by using a fi object, and `out2` and `var1` types are defined in the Model Explorer.

```
open_system('hdlcoder_chart_datatype_casting/DUT/Chart')
```



To generate HDL code for the `DUT` subsystem, run the `makehdl` function.

```
makehdl('hdlcoder_chart_datatype_casting/DUT')
```

Using Absolute Time Temporal Logic in Stateflow Charts

Guideline ID

2.9.5

Severity

Mandatory

Description

When you use absolute time temporal logic in your Stateflow Chart blocks in your model for HDL code generation, use these settings.

For the sample rate of the chart:

- If you use seconds (sec), then the sample time must be an integer 65535 or lower, or a decimal between 65.535 and 0.001 having no more than three decimal places.
- If you use milliseconds (msec), the sample time must be a decimal between 65.535 and 0.001 having no more than three decimal places, or a decimal between 0.065535 and 0.000001 having no more than six decimal places.
- If you use microseconds (usec), the sample time must be a decimal between 0.065535 and 0.000001 having no more than six decimal places, or a decimal between 0.000065535 and 0.000000001 having no more than nine decimal places.
- If the sample time is an integer below 2^16, use `'sec'`.
- If 1000 * sample time is an integer below 2^16, use `'sec'` or `'msec'`.
- If 1000000 * sample time is an integer below 2^16, use `'msec'` or `'usec'`.
- If 1000000000 * sample time is an integer below 2^16, use `'usec'`.

Modeling Error (default) State in Stateflow Charts

Guideline ID

2.9.6

Severity

Informative

Description

When generating HDL code for Stateflow charts that have exclusive state decomposition, states are represented by case statements. For example, a chart with three states, `A`, `B`, and `C`, results in a switch case:

```
CASE is_Chart IS
        WHEN IN_A =>
          is_Chart_next <= IN_B;
```

```
          y_tmp <= to_signed(16#00000002#, 32);
        WHEN IN_B =>
          is_Chart_next <= IN_C;
          y_tmp <= to_signed(16#00000003#, 32);
        WHEN OTHERS =>
          --case IN_C:
          y_tmp <= to_signed(16#00000003#, 32);
      END CASE;
```

It is useful to control which of the states corresponds to the default case of the switch. For example, you can use the default state to model error behavior. The state whose name is alphabetically last occurs in the default branch. In the example code, the default branch is `WHEN OTHERS=>`.

If you are using variant transitions, an unreachable default state can be eliminated in the HDL code. To avoid this, the selected default state must be reachable. For example, you can connect the default state to another reachable state and provide a transition that is always false, such as `1 == 0`. For more information, see “Control Indicator Lamp Dimmer Using Variant Conditions” (Stateflow).

Enable Clock-Driven Outputs of Stateflow Charts (Moore Charts Only)

Guideline ID

2.9.7

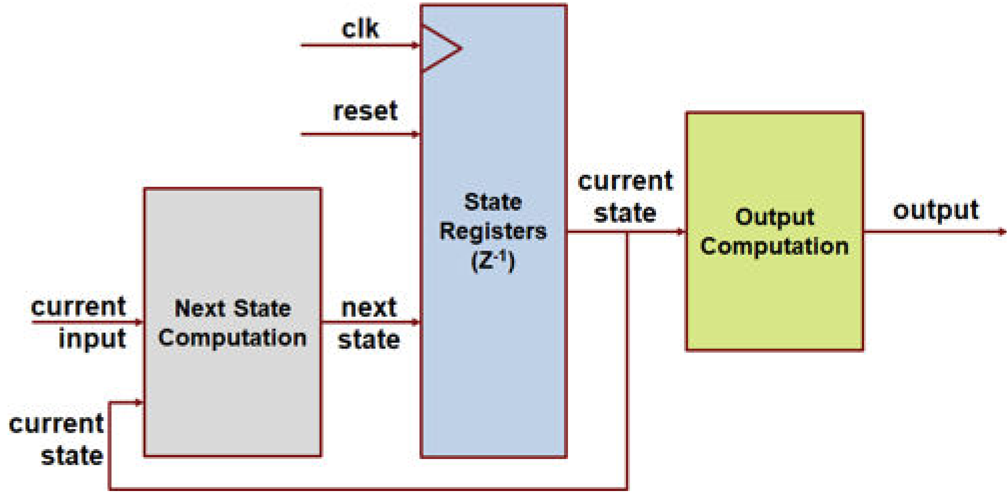
Severity

Informative

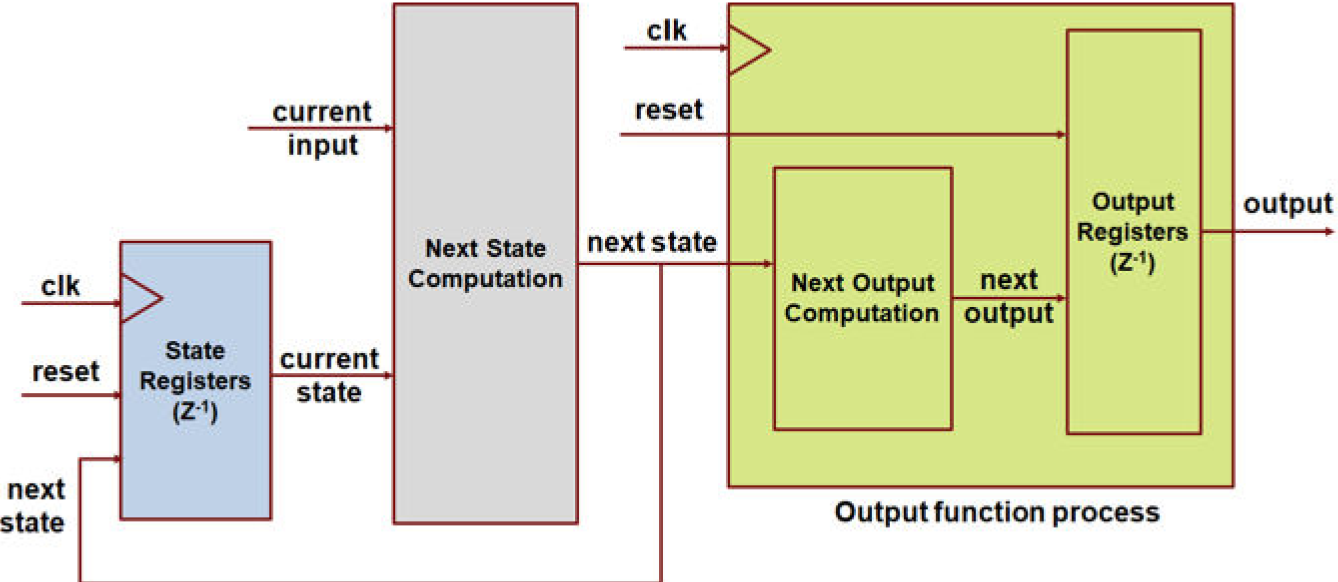
Description

You can generate HDL code that has clock-driven output signals for Moore charts in Stateflow. Clock-driven outputs prevent combinatorial logic from driving the output and allow an immediate output update when the clock signal and state change. You can enable clock-driven outputs for Stateflow charts by using the Stateflow chart HDL block property **ClockDrivenOutput**. This option is available only for Moore charts.

When **ClockDrivenOutput** is `off`, the HDL code generated from a Moore chart has this structure:



When you set **ClockDrivenOutput** to `on`, HDL Coder adds an output register that updates when the state updates. HDL Coder then assigns the final output variable a value from the clock-driven register. The generated HDL code has this structure:



The generated HDL code includes an additional output register for every output from the Moore chart.

Enumeration type for active state monitoring in a Stateflow chart with no default value

Guideline ID

2.9.8

Severity

Informative

Description

You can specify the enumeration type used to monitor state activity for a Stateflow chart that contains only literals that correspond to every state, without an extra literal for specifying the default `None` state, or any default state that does not represent a child state. The default value of the enumeration type can correspond to one of the states in the chart. Using an enumeration type that specifies only as many literals as there are states in the Stateflow chart allows you to generate HDL code that requires less area.

To monitor state activity for a Stateflow chart, enable active state data. See “Enable Active State Data” (Stateflow). To define the default value of the enumeration type as a state in your chart, select **Define enumerated type manually**. See “Define State Activity Enumeration Type” (Stateflow). If you use the option **Create enum definition from template** to create an enumeration definition template, the template creates the default state as `None`. Remove the `None` state.

For example, an enumerated type with the default state as `None` can have this definition:

```
classdef ChartModeType < Simulink.IntEnumType
    % MATLAB enumeration class definition generated from template
    %   to track the active child state of none_state_question2/Subsystem/Chart.

    enumeration
        A(0),
        B(1),
        None(3)
    end

    methods (Static)

        function defaultValue = getDefaultValue()
            % GETDEFAULTVALUE  Returns the default enumerated value.
            %   If this method is not defined, the first enumeration is used.
            defaultValue = ChartModeType.None;
        end

        % Unchanged methods not shown
   end
end
```

The section of generated HDL code from the enumerated type looks like this:

```
reg [1:0] ChartMode_reg;            // enum type ChartModeType (3 enums)
reg [1:0] ChartMode_reg_next;       // enum type ChartModeType (3 enums)
```

The registers `ChartMode_reg` and `ChartMode_reg_next` in the generated HDL code use two bits each to store the enumerated type.

When you remove the default `None` state from the enumerated definition and the function `getDefaultValue()`, this is the enumerated type:

```
classdef ChartModeType < Simulink.IntEnumType
    % MATLAB enumeration class definition generated from template
    %   to track the active child state of none_state_question2/Subsystem/Chart.

    enumeration
        A(0),
        B(1),
    end

    methods (Static)
        % Unchanged methods not shown
   end
end
```

The registers `ChartMode_reg` and `ChartMode_reg_next` in the generated HDL code now use only one bit each to store the enumerated type:

```
reg  ChartModeReg;              // enum type ChartModeEnum (2 enums)
reg  ChartModeReg_next;         // enum type ChartModeEnum (2 enums)
```

An enumerated type with no default `None` state for active state monitoring has these Stateflow limitations:

- You can only define an enumerated type with no `None` state to monitor the child activity of the top-most chart level.

- The Stateflow chart must have the property **Execute (enter) Chart At Initialization** enabled.
- You cannot use the `in` operator for the default child state.
- Local events are not allowed.
- You must set the configuration parameter **No unconditional default transitions** to `error`.

For more information on monitoring state activity in a Stateflow chart, see “Define State Activity Enumeration Type” (Stateflow).

See Also

Functions
`makehdl`

More About

- “Introduction to Stateflow HDL Code Generation” on page 26-2
- “Design Patterns Using Advanced Chart Features” on page 26-14

Simulink Data Type Considerations

You can follow these guidelines to learn the recommended data type settings that you want to use in your Simulink model for HDL code generation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see "HDL Modeling Guidelines Severity Levels" on page 18-3.

Use Boolean for Logical Data and Ufix1 for Numerical Data

Guideline ID

`2.10.1`

Severity

Mandatory

Description

`Boolean` and the fixed-point type, `ufix1`, are both `1-bit` data types in MATLAB and Simulink. These types are treated differently.

- Use `Boolean` for control logic signals such as enable and local reset signals. If you want to calculate a `Boolean` signal with a fixed-point data type, use a Data Type Conversion to convert the signal to a `fixdt(0,1,0)` type.
- To perform numeric calculations, use `fixdt(0,1,0)`. Sometimes, the output bit width can become larger than the bitwidth. To perform such operations, use the `Inherit: Inherit via internal rule` setting, because of the `numerictype` property of `fixdt(0,1,0)`.

Specify Data Type of Gain Blocks

Guideline ID

`2.10.2`

Severity

Recommended

Description

Gain blocks have a **Gain** parameter and an **Output data type** setting. It is recommended that you use fixed-point data types for these settings. In the Block Parameters dialog box of the Gain block:

- Specify a `Simulink.NumericType` object, such as `fixdt(1,16,8)`.
- Make sure that the **Gain** parameter of the block does not use a round parameter value. To avoid rounding of the gain value, you can specify a `fi` object, such as `fi(3.44,0,8,4)`.
- Avoid using `Inherit:Inherit via internal rule`. This setting can result in an erroneous data type being assigned to the block, thereby resulting in an HDL code generation error.

Enumerated Data Type Restrictions

Guideline ID

`2.10.3`

Severity

Mandatory

Description

Certain optimizations such as pipelining and resource sharing do not work seamlessly in the presence of enumerated data types. It is recommended that you use enumerated types on an as needed basis. HDL code generation has certain restrictions when modeling with enumerated types.

- You cannot use an enumerated data type for the input or output port of the top-level DUT.
- You cannot perform arithmetic operations such as `*`, `/`, `-`, and `+` with enumeration values.

Choose Optimal Simulink Block to Compute Sine and Cosine Functions with Fixed-Point Data Types

Guideline ID

`2.10.4`

Severity

Recommended

Description

You can compute Sine and Cosine functions with fixed-point data types by using the Trigonometric Function block, n-D Lookup Table block, Sine HDL Optimized block, Cosine HDL Optimized block, and NCO block. Each of these blocks adds different latencies and offer varying performance in speed and area. Choose the appropriate block based on your specific use case.

Follow these guidelines to determine the optimal block:

- Use the Trigonometric Function block for better precision in fixed-point applications. It provides the highest accuracy because it allows increased iterations of the CORDIC approximation.
- Use the n-D Lookup Table block for better speed. Because it has quick computation capabilities, it generates the outputs in one cycle. In comparison, the CORDIC approximation method of the Trigonometric Function block takes several cycles.
- Use the Sine HDL Optimized and Cosine HDL Optimized blocks to minimize circuit area size. They consumes only 1/4 of the memory compared to the n-D Lookup Table block.

Additionally, consider these limitations:

- When the required precision results in a lookup table (LUT) size that exceeds the available block RAM capacity, use the Trigonometric Function block with the **Approximate method** parameter set to `CORDIC` method.

- For the fixed-point Trigonometric Function block, the operating speed depends on the latency settings.

Guidelines for Block RAM Allocation:

- Allocate the n-D Lookup Table block, Sine HDL Optimized block, Cosine HDL Optimized block and NCO block to block RAM as required.
- To allocate the n-D Lookup Table block to block RAM, set the **MAPToRAM** parameter in the HDL Block Properties window or insert a Delay block after the n-D Lookup Table block.
- To allocate a Sine HDL Optimized block and Cosine HDL Optimized block to block RAM, do not place a Delay block immediately after it. This configuration prevents the assignment of table data to slice LUTs.

This table compares the Simulink blocks you can use to calculate sine and cosine functions with fixed-point data types:

Use Case	Block Name	Block Path	Parameter Settings	Supported Trigonometric Functions	Features	Limitations
Precision	Trigonometric Function	Simulink/ Math Operations/ Trigonometric Function	Set **Approximation method** to `CORDIC`. For more information about **LatencyStrategy**. See "LatencyStrategy" on page 19-33.	`sin`, `cos`, `cos+jsin`, `sincos`, `atan2`	This block uses the `CORDIC` approximate algorithm for calculations, which does not use memory but introduces delay due to pipelining. As the bit width increases, accuracy improves, and the circuit size remains smaller compared to other methods. For more information, see Trigonometric Function.	The number of iterations specified in the block parameters must be a natural number that is less than or equal to the word length of the fixed-point data.

Use Case	Block Name	Block Path	Parameter Settings	Supported Trigonometric Functions	Features	Limitations
Speed	n-D Lookup Table	HDL Coder/ Lookup tables/n-D lookup Table	Set **Interpolation method** to `Flat` or `Linear point-slope`.	`sin`, `cos`, `exp(j)`	This block is inefficient in terms of circuit area size because it occupies memory for entire period.	See Extended Capabilities section of n-D Lookup Table.
Circuit area size	Sine HDL Optimized and Cosine HDL Optimized	HDL Coder/ Lookup tables/Sine HDL Optimized an Cosine HDL Optimized	Default block settings	`sin`, `cos`, `exp(j)`, `sincos`	This block is efficient because it uses table data for 1/4 period or less.	To map to Block RAM, avoid placing a Delay block immediately after the Lookup Table block.
Speed and Circuit area size	NCO	DSP HDL Toolbox/ Signal Operations/ NCO	Set **Phase offset source** to `Input port` and **Phase increment** to `0`.	`sin`, `cos`, `exp(j)`, `sincos`	This block is efficient because it uses table data of ¼ period or less.	This block is designed for signal generation and can be used to calculate sine and cosine functions by configuring the appropriate parameters. See NCO (DSP HDL Toolbox).

The table compares the results from the operation of `sin` function on a logic synthesis with the single fixed-point data type on a Xilinx Kintex® 7, XC7K325T FFG900 device.

Block Name	Block Path	Parameter Settings	Number of Break Points	Number of Data Points	Maximum Frequency (in MHz)	Path Delay (in ns)	FPGA Resource Usage			
							LUTs	Registers	Block RAMs	DSPs
Trigonometric Function block	Simulink/Math Operations/ Trigonometric Function	Set **Approximation method** to `CORDIC`, **LatencyStrategy** to `Custom`, **CustomLatency** to `13`.	N/A	N/A	313	3.19	656	506	0	N/A
		Set **Approximation method** to `CORDIC`, **LatencyStrategy** to `Custom`, **CustomLatency** to `6`.	N/A	N/A	265	3.766	719	247	0	N/A

Block Name	Block Path	Parameter Settings	Number of Break Points	Number of Data Points	Maximum Frequency (in MHz)	Path Delay (in ns)	FPGA Resource Usage			
							LUTs	Registers	Block RAMs	DSPs
		Set **Approximation method** to `CORDIC`, **LatencyStrategy** to `Custom`, **CustomLatency** to `0`.	N/A	N/A	50	20.015	686	59	0	N/A
n-D Lookup Table block	HDL Coder/ Lookup tables/n-D lookup Table	Set **Interpolation method** to `Flat`.	1024	N/A	619	1.1614	21	12	0.5	N/A
		Set **Interpolation method** to `Linear point-slope`.	128	N/A	109	9.158	89	26	0	2
Sine HDL Optimized and Cosine HDL Optimized blocks	HDL Coder/ Lookup tables/ Sine HDL Optimized and Cosine HDL Optimized	Default block settings.	N/A	1024	374	2.672	52	13	0.5	N/A

Block Name	Block Path	Parameter Settings	Number of Break Points	Number of Data Points	Maximum Frequency (in MHz)	Path Delay (in ns)	FPGA Resource Usage			
							LUTs	Registers	Block RAMs	DSPs
NCO block	DSPHDL Toolbox/Signal Operations/NCO	Default block settings.	N/A	N/A	356	2.809	48	113	0.5	N/A

Choose Optimal Simulink Block to Compute Sine and Cosine Functions with Floating-Point Data Types

Guideline ID

`2.10.5`

Severity

Recommended

Description

You can compute Sine and Cosine functions with floating point data types by using the Trigonometric Function block or the n-D Lookup Table block. Each of these blocks adds different latencies and offer varying performance in speed and area. Choose the appropriate block based on your specific use case.

Follow these guidelines to determine the optimal block:

- Use Trigonometric Function block for superior accuracy, speed, and a smaller circuit area size than the n-D Lookup Table block.
- Use the n-D Lookup Table block in block RAM. The Trigonometric Function block is not suitable for RAM storage.

Additionally, consider these limitations:

- The **LatencyStrategy**: `ZERO, MIN, MAX` alter the clock frequency.
- The Trigonometric Function block supports a wide dynamic range with its floating-point logic circuit calculations, but, this may lead to an increase in circuit area size due to its complexity.

Guidelines for Block RAM Allocation

To allocate the n-D Lookup Table block to block RAM, set the **MAPToRAM** parameter in the HDL Block Properties window or insert a Delay block after the n-D Lookup Table block.

This table compares the Simulink blocks you can use to calculate sine and cosine functions with floating-point data types:

Use Case	Block Name	Block Path	Supported Data Types	Parameter Settings	Supported Trigonometric Function
Precision	Trigonometric Function	Simulink/Math Operations/ Trigonometric Function	Single	Set **Approximation method** to `None`. For LatencyStrategy settings, see "Latency Values of Floating-Point Operators" on page 14-99???.	`sin`, `cos`, `cos +jsin`, `sincos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`.
Speed (Implement to Block RAM)	n-D Lookup Table	HDL Coder/ Lookup Tables/n-D Lookup Table	Single, Double, Half	Set **Interpolation method** to `Linear point-slope` or `Flat` . For LatencyStrategy settings, see n-D Lookup Table.	`sin`, `cos`, `cos +jsin`, `sincos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`. The available functions depend on the table data settings.

This table compares the results from the `sin` function operation on a logic synthesis with the single floating-point on a Xilinx Kintex 7 XC7K325T FFG900 device.

Block Name	Block Path	Parameter Settings	Number of Break Points	Max Frequency (in MHz)	Path Delay (in ns)	FPGA Resource Usage			
						LUTs	Registers	Block RAMs	DSPs
Trigonometric Function	Simulink/Math Operations/ Trigonometric Function	Set **Approximation method** to `None`, **Latency Strategy** to `MAX`.	N/A	254	3.942	3749	2543	0	7

Block Name	Block Path	Parameter Settings	Number of Break Points	Max Frequency (in MHz)	Path Delay (in ns)	FPGA Resource Usage			
						LUTs	Registers	Block RAMs	DSPs
		Set **Approximation method** to None, **Latency Strategy** to ZERO.	N/A	23	43.6	3434	0	0	7
n-D Lookup Table	HDL Coder/ Lookup Tables/n-D Lookup Table	Set **Interpolation method** to Linear point-slope and **Latency Strategy** to ZERO. Numeric error 1e-3 order.	64	108	9.177	6202	88	1	1
		Set **Interpolation method** to Linear point-slope and **Latency Strategy** to MAX. Numeric error 1e-3 order.	64	103	9.708	6456	4593	1	1

Block Name	Block Path	Parameter Settings	Number of Break Points	Max Frequency (in MHz)	Path Delay (in ns)	FPGA Resource Usage			
						LUTs	Registers	Block RAMs	DSPs
		Set **Interpolation method** to Flat and **Latency Strategy** to MAX. Numeric error 1e-7 order.	1024	82	12.108	33240	490	1	1

See Also

Functions
makehdl

More About

- “Signal and Data Type Support” on page 14-3

Guidelines for Using Rounding and Saturation Settings for Fixed-Point Data Types

This guideline describes the design considerations when using rounding modes and saturate-on-integer overflow settings for fixed-point data types.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Guideline ID

`2.10.6`

Severity

Recommended

Description

When you use fixed-point data types in your design for HDL code generation, select the appropriate rounding mode and saturate-on-integer setting. Using the rounding or saturation settings may require additional circuits for computation and consume more hardware resources.

When you use blocks that have **Integer rounding mode** or **Saturate on integer overflow** parameters and use fixed-point data types, follow these considerations:

- Setting **Integer rounding mode** parameter to a setting other than `Floor` consumes more lookup tables (LUTs) and requires an additional adder operation. The table shows the LUT and adder count for different rounding modes when performing data type conversion for fixed-point data types. When you want to reduce the number of bits on least-significant bit (LSB) side, the generated code and hardware resources depend on the rounding mode setting.

Fixed-point Data Type Conversion	Rounding Mode	Hardware Resources	
		LUT	Adder Count
Reduce 2 bits on LSB side: Convert `sfix16_En14` to `sfix14_En10`	`Ceiling`	1	1
	`Convergent`	1	1
	`Floor`	0	0
	`Nearest`	1	1
	`Round`	1	1
	`Zero`	1	1
Reduce 4 bits on LSB side: Convert `sfix18_En14` to `sfix12_En8`	`Ceiling`	2	1
	`Convergent`	2	1
	`Floor`	0	0
	`Nearest`	1	1
	`Round`	2	1

Fixed-point Data Type Conversion	Rounding Mode	Hardware Resources	
		LUT	Adder Count
	Zero	2	1

- When you reduce the bits on most-significant bit (MSB) side, the generated code and hardware resources vary depending on the **Saturate on integer overflow** setting. If you enable **Saturate on integer overflow**, the block consumes more LUTs. The table shows the consumption of LUTs when you enable the **Saturate on integer overflow**.

Fixed-point Data Type Conversion	LUT
Reduce 2 bits on MSB side: Convert sfix36_En14 to sfix34_En14	33
Reduce 5 bits on MSB side: Convert sfix36_En14 to sfix31_En14	32
Reduce 6 bits on MSB side: Convert sfix36_En14 to sfix30_En14	32
Reduce 12 bits on MSB side: Convert sfix36_En14 to sfix24_En14	28

- Avoid setting the **Integer rounding mode** parameter to Simplest.
- You can set the Integer rounding mode parameter to either Zero or Floor depending on the settings of the **Device vendor** or **Signed integer division rounds to** model configuration parameters in the Configuration Parameter dialog. When you set **Device vendor** parameter to ASIC/FPGA, set the Integer rounding mode parameter to Zero.

See Also

More About

- “Rounding”
- “Saturation and Wrapping”

Guideline for Using Sqrt Block for HDL Code Generation

Follow these guidelines when you use Sqrt blocks in your model to improve performance in terms of frequency, area, and accuracy.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Use SqrtFunction Architecture for Square Root Block

Guideline ID

`2.11.1`

Severity

Recommended

Description

When you use the Sqrt block in your model, set the HDL architecture to `SqrtFunction` for improved circuit area, timing on FPGA target, and accuracy. Using this architecture for a Sqrt block has several advantages over the Newton-Raphson-based architectures. The table compares synthesis results between the different HDL architectures of the Sqrt block. The results in this table are from Sqrt operation on a logic synthesis with the uint8 data type on an Xilinx Kintex 7 `XC7K325T FFG900` device.

Sqrt Block Synthesis Results					
Architecture	Iterations	Maximum Frequency (in MHz)	FPGA Resource Usage		
			LUTs	Registers	DSPs
`SqrtFunction`	No iteration settings	354	20	49	0
`SqrtNewton`	3	189	290	137	3
	5		338	187	5
`RecipSqrtNewton`	3	185	141	164	4
	5		203	246	6
`SqrtNewtonSingleRate`	3	228	601	318	6
	5		839	468	10
`RecipSqrtNewtonSingleRate`	3	246	208	318	11
	5		317	484	17

By selecting the `SqrtFunction` architecture, you can also enable **UseMultiplier** property to compute the square root by using multipliers. This property is `off` by default and the square root is computed using the shift-add algorithm.

You can increase the design frequency and reduce resource utilization by setting the **UseMultiplier** to `off` and **LatencyStrategy** to `inherit`.

If you choose to use the Newton-Raphson-based architectures, follow these design guidelines:

- When designing a model for high frequency, use single-rate architectures such as `SqrtNewtonSingleRate` or `RecipSqrtNewtonSingleRate`. The resource consumption for single rate implementation is high because the implementation is performed without overclocking.
- You can increase the iterations for single-rate architecture to improve the accuracy. Recommended iteration values are between 3 to 10. However, even a greater number of iterations cannot make single-rate implementation more accurate than the `SqrtFunction` architecture. The simulation results of the Sqrt block with `SqrtFunction` architecture are the same as those of Simulink. By contrast, the simulation results for Newton-Raphson-based implementation can yield results that are less accurate and numerically different.
- Increasing number of iteration can increase latency and resource consumption. Additional cycles of latency depends on the architecture and iterations. For more information, see HDL Code Generation section in Sqrt.

See Also

Sqrt | rSqrt

Resource Sharing Settings for Various Blocks

Resource sharing is an area optimization in which HDL Coder identifies multiple functionally equivalent resources and replaces them with a single resource. The data is time-multiplexed over the shared resource to perform the same operations. To learn more about how resource sharing works, see “Resource Sharing” on page 21-45.

You can follow these guidelines to learn how to use the resource sharing optimization effectively with blocks such as Add and Product. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Resource Sharing of Add Blocks

Guideline ID

`3.1.1`

Severity

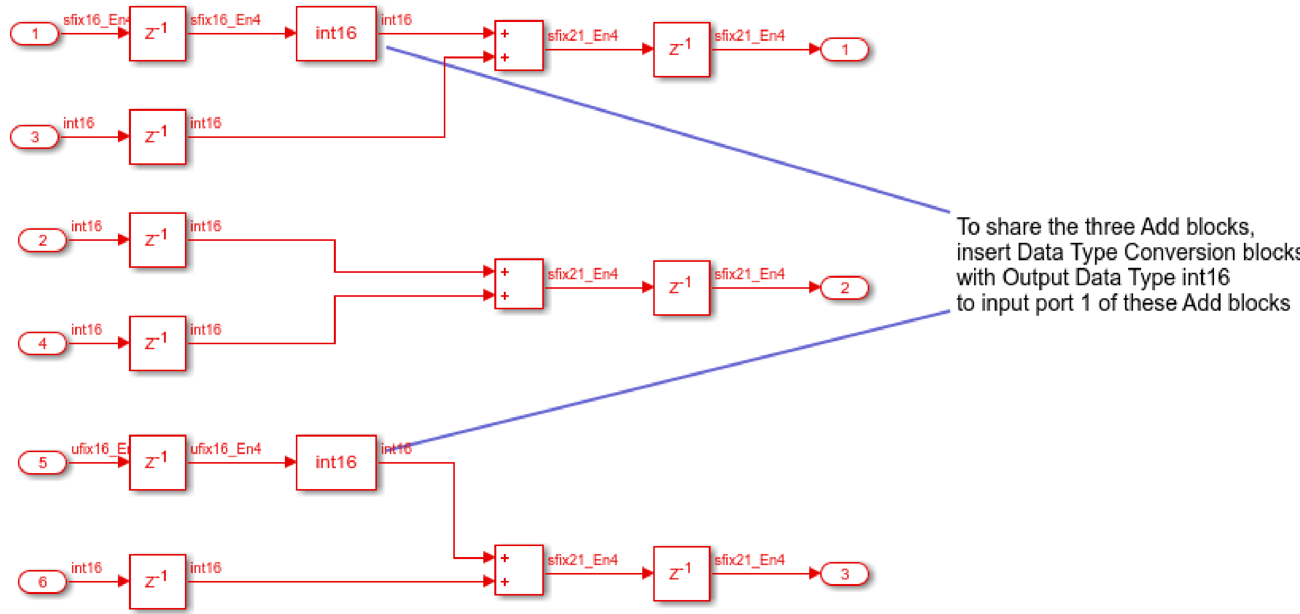
Recommended

Description

To share multiple Add blocks:

- Select the **Share Adders** setting.
- Leave the **Adder sharing minimum bitwidth** to `0`.
- Determine whether to perform resource sharing at the existing clock rate or at a higher clock rate. To use a higher clock rate, specify an **Oversampling factor** greater than `1`.
- Specify the “StreamingFactor” on page 19-25 for Add blocks with vector inputs or outputs.
- Specify the “SharingFactor” on page 19-23 for Add blocks with scalar inputs or outputs.
- Make sure that the input word-lengths of the Add blocks match.

  For example, this figure illustrates a model containing three Add blocks placed inside a Subsystem with **SharingFactor** of `3`. To share the Add blocks, you must insert Data Type Conversion blocks with **Output data type** set to `int16` so that the input word lengths match.



Resource Sharing of Gain Blocks

Guideline ID

`3.1.2`

Severity

Recommended

Description

When you share multiple Gain blocks in your design, the optimization inserts serialization and deserialization logic to share resources. This additional logic can become an area overhead if you are not sharing a large number of resources. Therefore, if your design does not contain a large number of Gain blocks to share, it is recommended that you disable the resource sharing optimization. To share multiple Gain blocks:

To share multiple Gain blocks:

- Determine how to implement the Gain block. HDL Coder does not share Gain blocks in either of these cases:

  - **ConstMultiplierOptimization** parameter set to `csd` or `fcsd`.
  - **Gain** parameter is a power of two.

  In both these cases, the code generator uses a cast operation to replace the multiplier operations with shift and add or subtract operations, which causes sharing to be unsuccessful. In addition, if the **Gain** parameter is `0` or `1`, then resource sharing requires no additional logic.
- Specify the "StreamingFactor" on page 19-25 for Gain blocks with vector inputs or outputs.

- Specify the “SharingFactor” on page 19-23 for Gain blocks with scalar inputs or outputs.
- Determine whether to perform resource sharing at the existing clock rate or at a higher clock rate. To use a higher clock rate, specify an **Oversampling factor** greater than `1`.
- Use the same synthesis attribute settings if you specify the `DSPStyle` block property for the Gain blocks. HDL Coder does not share multipliers that have different synthesis attribute settings.

Resource Sharing of Product Blocks

Guideline ID

`3.1.3`

Severity

Recommended

Description

To share multiple Product blocks:

- Specify `18` as the **Multiplier partitioning threshold** when targeting Xilinx devices and `25` as the threshold when targeting Intel devices. This setting creates more resource sharing opportunities for multipliers with a wide bit width, which reduces the use of DSPs on the FPGA.
- Specify the **Multiplier promotion threshold** if you want to share Product blocks that have different word-lengths. The multiplier promotion threshold is the maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers.
- Leave the **Share Multipliers** setting enabled and the **Multiplier sharing minimum bitwidth** to `0`.
- Specify the “StreamingFactor” on page 19-25 for the subsystems that contain Product blocks with vector inputs or outputs.
- Specify the “SharingFactor” on page 19-23 for the subsystems that contain Product blocks with scalar inputs or outputs.
- Use a Gain block instead of a Product block when one of the inputs to the Product block is a constant. Use the constant value as the **Gain** parameter of the Gain block. If you use floating-point data types in the `Native Floating Point` mode, HDL Coder converts the Product block to a Gain block automatically during code generation. To learn more, see “Simplify Constant Operations and Reduce Design Complexity in HDL Coder” on page 21-18.
- Determine whether to perform resource sharing at the existing clock rate or at a higher clock rate. To use a higher clock rate, specify an **Oversampling factor** greater than `1`.
- Use the same synthesis attribute settings if you specify the `DSPStyle` block property for the Product blocks. HDL Coder does not share multipliers that have different synthesis attribute settings.

Resource Sharing of Multiply-Add Blocks

Guideline ID

`3.1.4`

Severity

Recommended

Description

To share multiple Multiply-Add blocks:

- Leave the **Share Multiply-Add blocks** setting enabled and the **Multiply-Add block sharing minimum bitwidth** set to `0`.
- Determine whether to perform resource sharing at the existing clock rate or at a higher clock rate. To use a higher clock rate, specify an **Oversampling factor** greater than `1`.
- Specify the “SharingFactor” on page 19-23.

See Also

Related Examples

- “Resource Sharing of Multipliers to Reduce Area” on page 8-33
- “Resource Sharing for Area Optimization” on page 21-54
- “Single-Rate Resource Sharing Architecture” on page 21-65

More About

- “Streaming” on page 21-42
- “Resource Sharing of Subsystems and Floating-Point IPs” on page 18-152

Resource Sharing of Subsystems and Floating-Point IPs

Resource sharing is an area optimization in which HDL Coder identifies multiple functionally equivalent resources and replaces them with a single resource. The data is time-multiplexed over the shared resource to perform the same operations. To learn more about how resource sharing works, see “Resource Sharing” on page 21-45.

You can follow these guidelines to learn how to use the resource sharing optimization effectively for subsystems such as atomic subsystems and MATLAB Function blocks, and with floating-point IPs. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

General Considerations for Sharing of Subsystems

Guideline ID

`3.1.5`

Severity

Recommended

Description

To share resources for identical subsystems, such as when grouping Product, Add, and Delay blocks to map to one DSP slice, the subsystems to be shared must be Atomic Subsystem blocks, Virtual Subsystem blocks, or MATLAB Function blocks.

- Determine whether you want to share resources at the existing clock rate or at a higher clock rate.
- Sharing of enabled subsystems is not supported. For sharing resources, use atomic or virtual subsystems without enable semantics.
- Specify a “SharingFactor” on page 19-23 that is greater than or equal to the number of subsystems that you want to share.

  For example, if you have `10` subsystems, and you set the **SharingFactor** to `5`, HDL Coder cannot implement the resource sharing to `2` instances of the subsystem. To share the subsystems, divide the subsystems, and then share the instances of the smaller subsystems.
- Check the **SharingFactor** that you specify for various subsystems. The resource sharing optimization overclocks the shared resources by the LCM (Least Common Multiple) of the **SharingFactor** of various subsystems.

  For example, if you specify a **SharingFactor** of `5` for one Subsystem, and a **SharingFactor** of `7` for another Subsystem, the resource sharing optimization overclocks the shared resources by `35`. In such cases, it is recommended that you use the same **SharingFactor** for both subsystems, such as `5` or `7`. To learn more about this calculation, see “How Resource Sharing Works” on page 21-45.

Use MATLAB Datapath Architecture for Sharing with MATLAB Function Blocks

Guideline ID

`3.1.6`

Severity

Recommended

Description

HDL Coder shares MATLAB Function blocks that have:

- The same Simulink checksum. Use `Simulink.SubSystem.getChecksum` to determine the checksum.
- The same HDL block properties.

Make sure that the blocks do not use:

- Persistent variables
- Loop streaming
- Output pipelining

By using the `MATLAB Datapath` architecture, you can share resources inside the MATLAB Function block and across the MATLAB Function block with other blocks in your Simulink model. When you use this architecture, the code generator treats the MATLAB Function block like a regular Subsystem block. This capability enables you to more widely apply various speed and area optimizations with MATLAB Function blocks. See "HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture" on page 21-205.

Sharing of Subsystems

Guideline ID

`3.1.7`

Severity

Recommended

Description

HDL Coder can share Subsystem blocks that have the same Simulink checksum and the same HDL block properties.

To share Subsystem blocks, the state elements that the blocks can contain are:

- Delay
- Unit Delay
- Unit Delay Enabled Synchronous

- Unit Delay Resettable Synchronous
- Unit Delay Enabled Resettable Synchronous

The state elements must have the **Initial condition** parameter set to 0.

Sharing of subsystems inside enabled subsystems with synchronous semantics is not supported. To share resources, use enabled subsystems with classic semantics.

You cannot share subsystems that contain the following blocks or block implementations:

- Detect Change
- Discrete Transfer Fcn
- HDL FFT
- HDL FIFO
- Math Function (conj, hermitian, transpose)
- MATLAB Function blocks that contain persistent variables
- Sqrt
- Cascade architecture (MinMax, Product, Sum)
- Reciprocal Newton architecture
- Filter blocks including Discrete FIR Filter
- Communications Toolbox blocks
- DSP System Toolbox blocks, except Discrete FIR Filter
- Stateflow blocks
- Blocks that are not supported for delay balancing. For details, see “Delay Balancing Limitations” on page 21-84.

Limitations

Resource sharing of atomic subsystems might cause a mismatch in the initialization cycles of the validation model if your model has:

- Logic in the subsystems shared that produce non-zero output when the input is zero.
- Subsystems shared that are in a serial configuration in your design under test (DUT) subsystem.

Resource Sharing of Floating-Point IPs

Guideline ID

`3.1.8`

Severity

Recommended

Description

To share multiple:

- Floating-point adders, set `ShareAdders` to on.

- Floating-point multipliers, make sure `ShareMultipliers` is set to on.
- Other floating-point resources, set `ShareFloatingPointIP` to on.

See also "Modeling with Native Floating Point" on page 18-65.

See Also

Related Examples

- "Resource Sharing of Multipliers to Reduce Area" on page 8-33
- "Resource Sharing for Area Optimization" on page 21-54
- "Single-Rate Resource Sharing Architecture" on page 21-65

More About

- "Resource Sharing" on page 21-45
- "Streaming" on page 21-42

Resource Sharing Guidelines for Vector Processing and Matrix Multiplication

Resource sharing is an area optimization in which HDL Coder identifies multiple functionally equivalent resources and replaces them with a single resource. The data is time-multiplexed over the shared resource to perform the same operations. To learn more about how resource sharing works, see “Resource Sharing” on page 21-45.

You can follow these guidelines to learn how to use the resource sharing with streaming when processing 1-D vectors and 2-D matrices. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Use StreamingFactor for Resource Sharing of Vector Signals

Guideline ID

`3.1.9`

Severity

Informative

Description

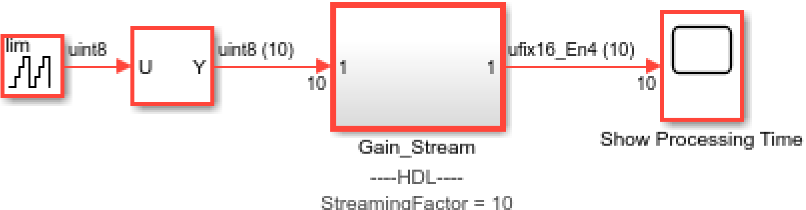
To reduce circuit area of a Subsystem block that performs the same computation on each element of a 1-D vector, use the Subsystem HDL block property **StreamingFactor**. For a vector signal that has N elements, set **StreamingFactor** to `N`. By using time-division multiplexing to process each element, you can process the result by using smaller number of operations. The clock frequency of the operators becomes N times faster than that of the original model.

When the subsystem containing resources to be shared uses multiple vector signals with different sizes, the clock frequency is multiplied by the least common multiple of the vector sizes, which can reduce the maximum achievable target frequency. To achieve the desired frequency:

- Add logic for demultiplexing the vector signal before it enters the subsystem and for multiplexing the signal that leaves the subsystem. You can then specify a **SharingFactor** on the subsystem instead of the **StreamingFactor**.
- Pad the different vector signals to make them the same size as the vector signal that has the maximum size, and then specify the **StreamingFactor**.

Open the model `hdlcoder_vector_stream_gain`.

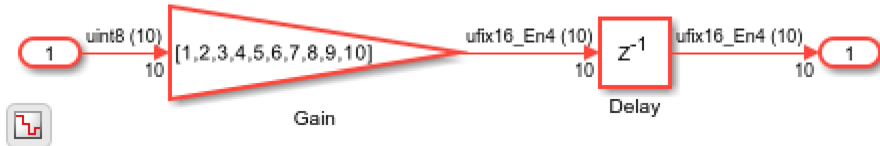
```
open_system('hdlcoder_vector_stream_gain')
set_param('hdlcoder_vector_stream_gain', 'SimulationCommand', 'Update')
```



Copyright 2020 The MathWorks, Inc.

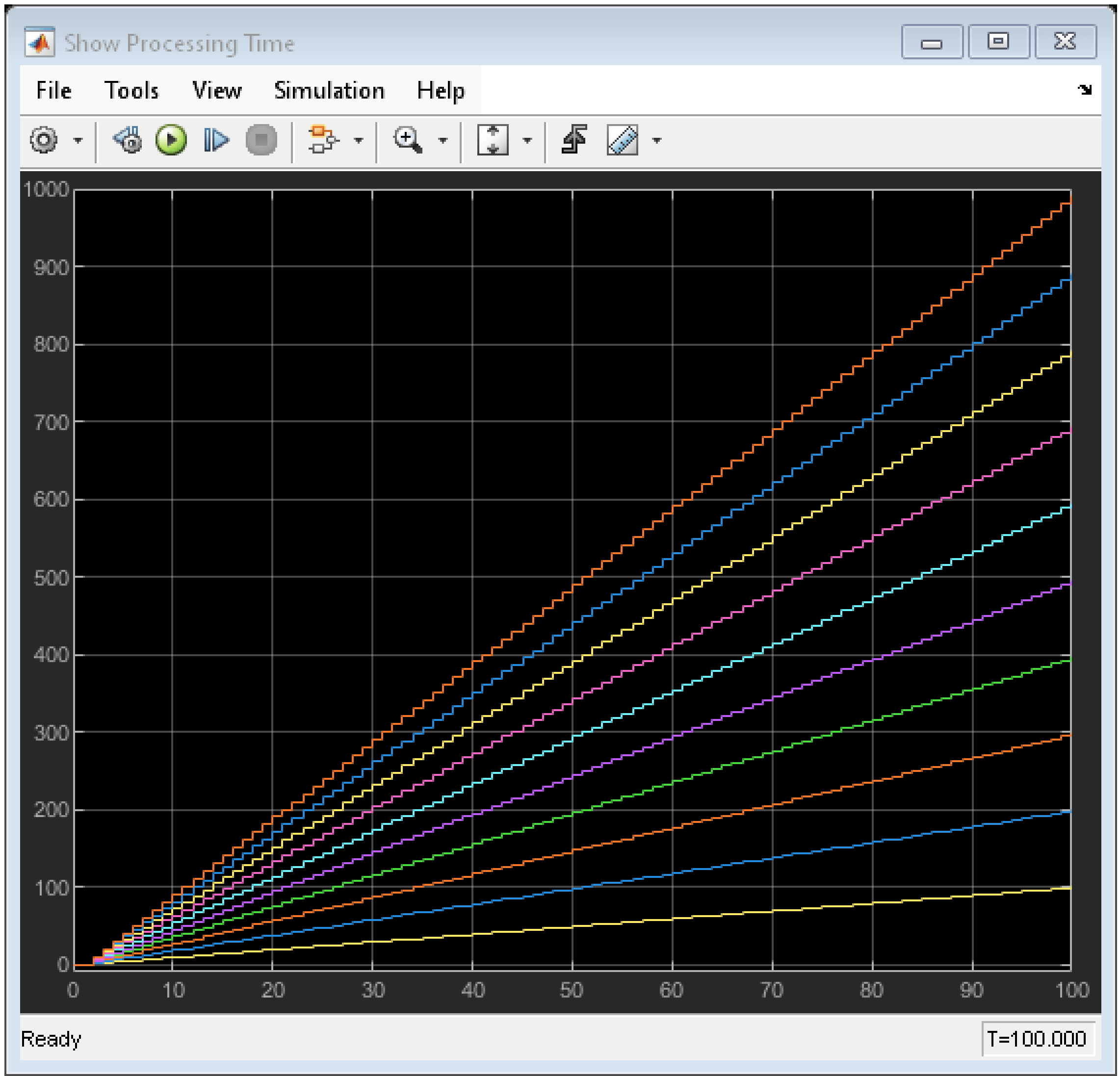
The model accepts a 10-element vector signal as input and multiplies each element by a gain value that is one more than the previous value.

```
open_system('hdlcoder_vector_stream_gain/Gain_Stream')
```



To see the simulation results, simulate the model and open the Scope block.

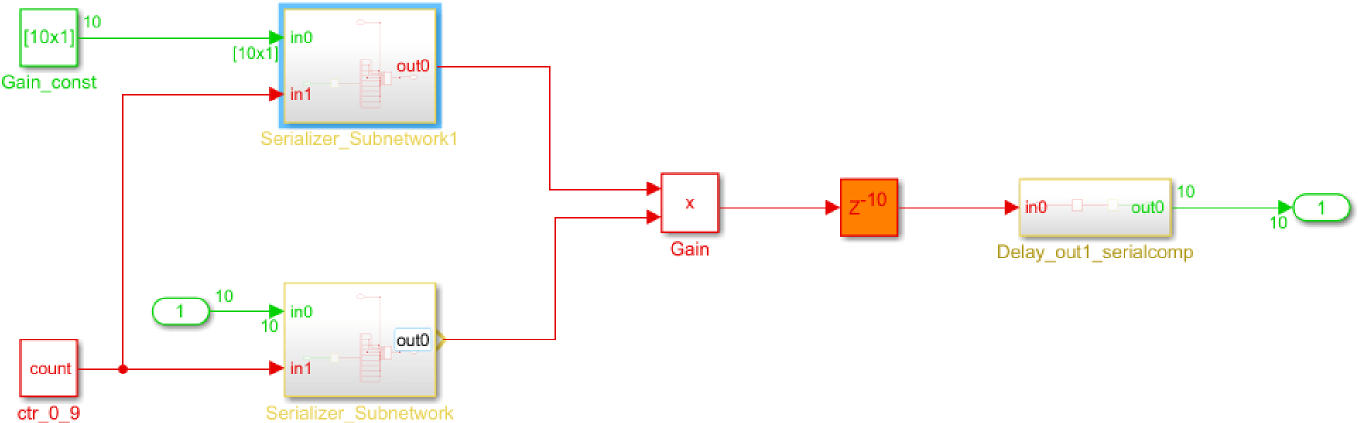
```
sim('hdlcoder_vector_stream_gain')
open_system('hdlcoder_vector_stream_gain/Show Processing Time')
```



The `Gain_Stream` subsystem has a **StreamingFactor** set to `10`. To generate HDL code for this subsystem, run the `makehdl` function:

```
makehdl('hdlcoder_vector_stream_gain/Gain_Stream')
```

After generating HDL code, to see the effect of the streaming optimization, open the generated model and navigate inside the `Gain_Stream` subsystem.



The vector data is serialized on the input side and the output size parallelizes the serial data. This optimization increases the total circuit size conversely when the target circuit size to be shared is small. The Gain block inside the shared subsystem is running at a rate that is 10 times faster than the model base rate, which avoids an increase in the subsystem latency and balances the reduction in maximum achievable frequency by the increase in area savings on the target hardware.

Use SharingFactor and HDL Block Properties for Sharing Matrix Multiplication Operations

Guideline ID

`3.1.10`

Severity

Informative

Description

The Matrix Multiply block is a Product block that has **Multiplication** block parameter set to `Matrix(*)`. In the HDL Block Properties dialog box, the HDL architecture is set to `Matrix Multiply` and you can specify the “DotProductStrategy” on page 19-10.

DotProductStrategy Settings

DotProductStrategy	Description
`'Fully Parallel'` (default)	Performs multiplication and addition operations in parallel. `[MxN]*[NxM]` matrix multiplication requires `N*M*M` multipliers.
`'Parallel Multiply-Accumulate'`	Uses the **Parallel** architecture of the Multiply-Accumulate block to implement the matrix multiplication. This architecture performs multiple Multiply-Add blocks in parallel with accumulation.
`'Serial Multiply-Accumulate'`	Uses the **Serial** architecture of the Multiply-Accumulate block to implement the matrix multiplication. This mode performs `N` times oversampling and number of multipliers becomes `M*M`.

To share resources and reduce the number of multipliers further, when you have multiple Matrix Multiply blocks in the same subsystem, set **DotProductStrategy** to `Fully Parallel` and specify the **SharingFactor** on the upper subsystem.

For multiplications involving complex and real numbers, the number of multipliers become doubled.

Number of Multipliers Generated by Multiplication of [MxN]*[NxM]

Multiplication Type	Fully Parallel/Parallel Multiply-Accumulate	Serial Multiply-Accumulate
Real x Real	`N*M*M`	`M*M`
Complex x Real	`N*M*M*2`	`M*M*2`
Complex x Complex	`N*M*M*4`	`M*M*4`

For floating-point matrix multiplication, select **Use Floating Point**. In this case, you must use the `Fully Parallel` **DotProductStrategy**. As this mode does not use element-wise operations and performs parallel multiplication and addition operations, use the **SharingFactor** instead of the **StreamingFactor** to share resources and save circuit area.

For an example that shows how to perform streaming matrix multiplication using floating-point types, see “HDL Code Generation for Streaming Matrix Multiply System Object” on page 1-31.

See Also

`makehdl`

More About

- “Resource Sharing Settings for Various Blocks” on page 18-148
- “Streaming” on page 21-42
- “Streaming: Area Optimization” on page 21-49
- “Single-Rate Resource Sharing Architecture” on page 21-65
- “HDL Code Generation for Streaming Matrix Inverse System Object” on page 1-22

Distributed Pipelining and Clock-Rate Pipelining Guidelines

The code generator introduces registers when you specify certain block implementations or use certain settings. You can follow these guidelines to learn more about these registers and how you can use them to optimize the timing of your design.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Clock-Rate Pipelining Guidelines

Guideline ID

`3.2.1`

Severity

Informative

Description

In most cases, the code generator introduces the registers in regions that run slower than the clock rate. To avoid or minimize additional latency, you can run these registers at the fast clock rate by using clock-rate pipelining. You can use clock-rate pipelining with these optimizations:

- Input and output pipelining
- Multi-cycle block implementations, such as complex math operations like Sqrt and Reciprocal.
- Floating-point library mapping
- Delay balancing
- Resource sharing and streaming

For designs with multiple hierarchies, when you want to perform certain system-level optimizations, such as sharing or distributed pipelining, it is recommended that you have the HDL block property **FlattenHierarchy** enabled on the top-level Subsystem.

To learn more about clock-rate pipelining and blocks that act as barriers to this optimization, see “Clock-Rate Pipelining” on page 21-148.

Recommended Distributed Pipelining Settings

Guideline ID

`3.2.2`

Severity

Recommended

Description

Distributed pipelining is a speed optimization that reduces the critical path by moving existing delays in your design while preserving the functional behavior. This optimization moves the delays within a subsystem while preserving the hierarchy.

To use this optimization for a design, in the Configuration Parameters dialog box, on the **HDL Code Generation** > **Optimization** pane, navigate to the **Pipelining** tab and select the **Distributed pipelining** check box.

To more effectively use this optimization, in the Configuration Parameters dialog box, on the **HDL Code Generation** > **Optimization** pane, you can specify these additional settings.

- **ConstrainedOutputPipeline**: Make sure that the total number of delays that are inserted including any input and output pipelining that you specify is greater than or equal to the value that you specify for **ConstrainedOutputPipeline** on the Subsystem.
- **Use synthesis estimates for distributed pipelining**: Select this option if you want to use synthesis timing estimates to calculate the propagation delays of the components in your design for distributed pipelining. This option can more accurately reflect how components function on hardware to better distribute pipelines in your design and maximize the clock frequency for your target device. See "Distributed Pipelining Using Synthesis Timing Estimates" on page 21-136.
- **Clock-rate pipelining**: Select this option if you want the code generator to insert registers at the clock rate instead of the data rate.
- **Allow clock-rate pipelining of DUT output ports**: Select this option if you want the code generator to insert registers at the clock rate instead of the data rate at the DUT output ports.
- **Allow design delay distribution**: Disable this option if you do not want the code generator to move the delays you added to your design. The optimization only moves pipeline registers.
- **Pipeline distribution priority**: Specify whether you want the priority to be `Numerical Integrity` or `Performance`. If you use `Performance`, make sure that the simulation results match. In some cases, this setting moves registers into blocks that have initial values such as constants, which can affect simulation results.

The subsystem to which you apply the optimization must meet these requirements:

- The subsystem that you apply this optimization on cannot contain any feedback loops.
- The subsystem must include only blocks that are supported for distributed pipelining. For a list of unsupported blocks, see "Limitations of Distributed Pipelining" on page 21-133. As a workaround:
  - Place some of the unsupported blocks, such as Dot Product blocks, inside another subsystem that does not have distributed pipelining enabled.
  - Change the **Pipeline distribution priority** parameter to `Performance` for blocks such as Enabled Subsystem blocks.
- The **Sample Time** parameter of the blocks must be discrete. If you have blocks with **Sample Time** set to `inf`, change them to `-1` if the infinite sample time propagates to the DUT output. To identify and change the sample time programmatically, see "Change Block Parameters by Using find_system and set_param" on page 18-36.

See Also

Related Examples

- “Distributed Pipelining: Speed Optimization” on page 21-139
- “Distributed Pipelining for Clock Speed Optimization” on page 8-15

More About

- “Distributed Pipelining” on page 21-130

Insert Distributed Pipeline Registers for Blocks with Vector Data Type Inputs

Distributed pipelining is a speed optimization that reduces the critical path by moving existing delays in your design while preserving the functional behavior. This guidelines illustrates how you can use the optimization effectively for vector inputs.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 18-3.

Guideline ID

`3.2.3`

Severity

Informative

Description

Blocks that Participate in Distributed Pipelining with Vector Types

By specifying certain settings, you can apply the distributed pipelining optimization to insert pipeline registers for these blocks when you input vectors that are larger than `3` in size. For details, see the "HDL Code Generation" section of each block page.

- Adders: Add, Subtract, and Sum of Elements
- Multipliers: Gain, Product, and Product of Elements
- MinMax
- Dot Product

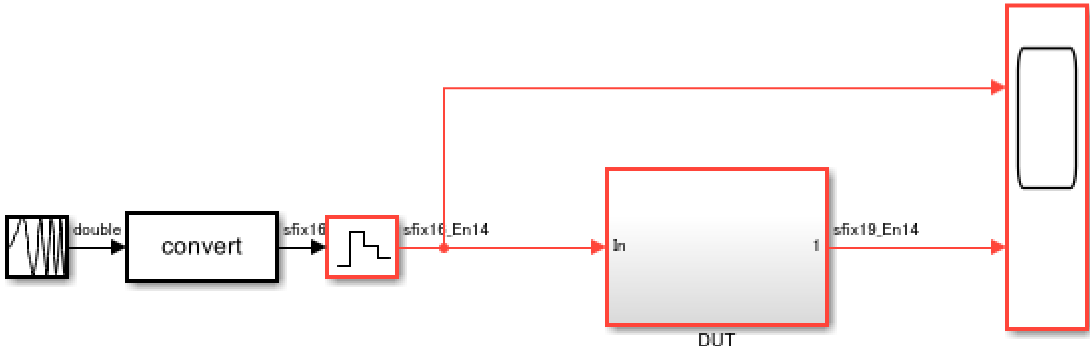
Block Settings and Requirements

1. In the HDL Block Properties for the blocks, set **Architecture** to :
   - `Tree` or `Linear` for adders, multipliers, and MinMax blocks. Distributed pipeline register insertion does not support `Cascade` architecture.
   - `Linear` for Dot Product. Distributed pipeline register insertion does not support `Tree` architecture for this block.
2. Specify the number of pipeline stages by using the **InputPipeline** and **OutputPipeline** properties in the HDL Block Properties dialog box, or by manually inserting Delay blocks.
3. Enable **DistributedPipelining** on the Subsystem for which you want to apply this optimization.
4. Open the Distributed Pipelining report.
5. Open and examine the generated model.

Distributed Pipelining Example for Vector Sum of Elements

This example shows how you can distribute pipeline registers at the output of a Sum of Elements block that accepts vector inputs.

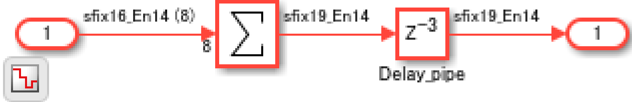
```
open_system('hdlcoder_distributed_pipelining_soe')
set_param('hdlcoder_distributed_pipelining_soe','SimulationCommand','Update')
```



Copyright 2018-2021 The MathWorks, Inc.

If you navigate the model, you see three pipeline stages for the Sum of Elements block.

```
open_system('hdlcoder_distributed_pipelining_soe/DUT/Add')
```



You see a Delay block of three added at the output of the Sum of Elements block. You can use distributed pipelining to distribute the delays.

1. Set **Architecture** to `Tree` for the Sum of Elements block.

```
hdlset_param('hdlcoder_distributed_pipelining_soe/DUT/Add/Add', ...
              'Architecture','Tree')
```

2. Enable **DistributedPipelining** on the Add Subsystem

```
hdlset_param('hdlcoder_distributed_pipelining_soe/DUT/Add', ...
              'DistributedPipelining','On')
```

3. Generate HDL code for the DUT Subsystem.

```
makehdl('hdlcoder_distributed_pipelining_soe/DUT')
```

4. Open the Code Generation Report to see the effect of the distributed pipelining optimization.

Detailed Report

Subsystem: *Add*

Implementation Parameters: *DistributedPipelining: 'on'; InputPipeline: 0; OutputPipeline: 0*

Status: Distributed Pipelining successful.

Before Distributed Pipelining : 3 registers *(57 flip-flops)*

Registers	Count
19-bit	3

After Distributed Pipelining : 7 registers *(123 flip-flops)*

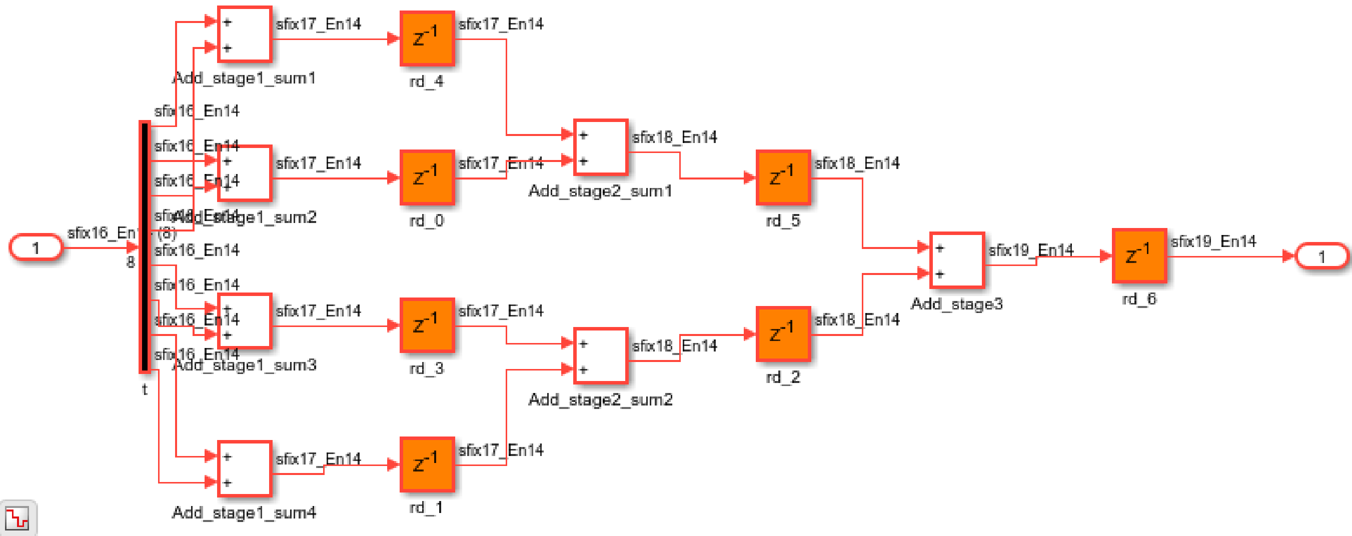
Registers	Count
17-bit	4
18-bit	2
19-bit	1

Generated Model

Generated model after the transformation:	gm_hdlcoder_distributed_pipelining_soe

5. To see the effect of the transformation and how the pipeline registers are distributed, open the generated model and navigate to the Add Subsystem.

```
load_system('gm_hdlcoder_distributed_pipelining_soe')
set_param('gm_hdlcoder_distributed_pipelining_soe','SimulationCommand','Update')
open_system('gm_hdlcoder_distributed_pipelining_soe/DUT/Add')
```



See Also

Related Examples

- “Distributed Pipelining: Speed Optimization” on page 21-139
- “Distributed Pipelining for Clock Speed Optimization” on page 8-15

More About

- “Distributed Pipelining” on page 21-130

Supported Blocks Library and Block Properties

- "View HDL-Supported Blocks and HDL-Specific Block Documentation" on page 19-2
- "HDL Block Properties: General" on page 19-3
- "HDL Block Properties: Native Floating Point" on page 19-29
- "HDL Filter Block Properties" on page 19-39
- "HDL Filter Architectures" on page 19-45
- "Distributed Arithmetic for HDL Filters" on page 19-50
- "Set and View HDL Model and Block Parameters" on page 19-52
- "Pass through and No HDL Implementations" on page 19-56
- "Build a ROM Block with Simulink Blocks" on page 19-57
- "Getting Started with RAM and ROM in Simulink" on page 19-58
- "Wireless Communications Design for ASICs, FPGAs, and SoCs" on page 19-61

View HDL-Supported Blocks and HDL-Specific Block Documentation

View HDL-Supported Blocks and Documentation

You can generate efficient HDL code for a number of blocks in Simulink and other product libraries. To see the product libraries that support HDL code generation use the `hdllib` function. This function filters the library browser to show blocks that are supported for HDL code generation. To learn more, see “Display Blocks for HDL Code Generation in Library Browser” on page 23-31.

This table shows blocks in various product libraries supported for HDL code generation. To view usage notes and limitations, in the corresponding reference page, scroll down to the **Extended Capabilities** section at the bottom and expand the **HDL Code Generation** section.

HDL code generation support for the blocks is summarized in the following tables.

- Blocks Supported for HDL Code Generation (Category List)
- Blocks Supported for HDL Code Generation (Alphabetical List)

View HDL-Specific Block Documentation

The HDL Block Properties dialog box contains HDL-specific properties for each block and subsystem in your model. On this dialog box, you can click the **Help** button to navigate to the documentation for that block. See also “HDL Block Properties: General” on page 19-3 and “HDL Block Properties: Native Floating Point” on page 19-29.

To view HDL-specific block documentation, either:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the block for which you want to see the help documentation and then select **HDL Block Properties**. To view the block documentation, click **Help**.
- Right-click the block and select **HDL Code > HDL Block Properties**. To view the block documentation, click **Help**.

You see the documentation in the **Extended Capabilities > HDL Code Generation** section of the block page in the product that owns the block.

See Also

`hdllib`

Related Examples

- “Set and View HDL Model and Block Parameters” on page 19-52
- “Display Blocks for HDL Code Generation in Library Browser” on page 23-31

More About

- “HDL Block Properties: General” on page 19-3
- “HDL Filter Block Properties” on page 19-39

HDL Block Properties: General

In this section...

“Overview” on page 19-3
“AdaptivePipelining” on page 19-4
“AllowDelayDistribution” on page 19-5
“BalanceDelays” on page 19-5
“ClockRatePipelining” on page 19-6
“CodingStyle” on page 19-7
“ConstMultiplierOptimization” on page 19-8
“ConstrainedOutputPipeline” on page 19-8
“DistributedPipelining” on page 19-9
“DotProductStrategy” on page 19-10
“DSPStyle” on page 19-11
“FlattenHierarchy” on page 19-13
“GuardIndexVariables” on page 19-14
“InputPipeline” on page 19-14
“InstantiateFunctions” on page 19-15
“InstantiateStages” on page 19-16
“LoopOptimization” on page 19-16
“LUTRegisterResetType” on page 19-17
“MapPersistentVarsToRAM” on page 19-17
“MapToRAM” on page 19-19
“OutputPipeline” on page 19-19
“PreserveUpstreamLogic” on page 19-20
“RAMDirective” on page 19-20
“ResetType” on page 19-21
“SerialPartition” on page 19-23
“SharingFactor” on page 19-23
“SoftReset” on page 19-23
“StreamingFactor” on page 19-25
“UseRAM” on page 19-25
“VariablesToPipeline” on page 19-28

Overview

Block implementation parameters enable you to control details of the code generated for specific block implementations. See “Set and View HDL Model and Block Parameters” on page 19-52 to learn how to select block implementations and parameters in the GUI or the command line.

Property names are specified as character vectors. The data type of a property value is specific to the property. This section describes the syntax of each block implementation parameter and how the parameter affects generated code.

HDL Block Properties of Library Blocks

HDL block properties of library blocks are treated similar to mask parameters. When you instantiate library blocks in your model, the current HDL block properties of that library block are copied to instances of that block in your model. The HDL block properties of these instances are not synchronized with the HDL block properties of the library block. That is, if you change the HDL block property of the library block, the change does not get propagated to instances of the library block that you already added to your Simulink model. If you want the HDL block properties of a library block to be synchronized with its instances in the model, create a Subsystem and then place this block inside that Subsystem. The HDL block properties of blocks that reside inside the library block are synchronized with the corresponding instances in your model.

Suppose a library contains a Subsystem block with HDL architecture set to `Module`. When you instantiate this block in your model, the block instance uses `Module` as the HDL architecture. If you change the HDL architecture of the Subsystem block in the library to `BlackBox`, existing instances of that Subsystem block in your model still use `Module` as the HDL architecture. If you now add instances of the Subsystem block from the library in your model, the new block instances get a copy of the current HDL block properties, and therefore use `BlackBox` as the HDL architecture. If you want the HDL architecture of the Subsystem block in the library to be synchronized with its instances in the model, create a wrapper subsystem with the HDL architecture that you want inside this Subsystem.

AdaptivePipelining

The `AdaptivePipelining` subsystem parameter enables you to set adaptive pipelining on a subsystem within a model.

Adaptive Pipelining Setting	Description
`'inherit'` (default)	Use the adaptive pipelining setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the adaptive pipelining setting for the model.
`'on'`	Insert adaptive pipelines for this subsystem.
`'off'`	Do not insert adaptive pipelines for this subsystem, even if the parent subsystem has adaptive pipelining enabled.

To disable adaptive pipelining for a subsystem within a model, set the adaptive pipelining parameter, `AdaptivePipelining`, to `'off'` for that subsystem.

To learn how to set model-level adaptive pipelining, see Adaptive pipelining.

Set Adaptive Pipelining For a Subsystem

To set adaptive pipelining for a subsystem from the HDL Block Properties dialog box:

1. Right-click the subsystem and select **HDL Code** > **HDL Block Properties**.

2 For **AdaptivePipelining**, select **inherit**, **on**, or **off**.

To set adaptive pipelining for a subsystem from the command line, use `hdlset_param`. For example, to turn off adaptive pipelining for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'AdaptivePipelining', 'off')
```

See also `hdlset_param`.

AllowDelayDistribution

Use the `AllowDelayDistribution` property of a Delay or Unit Delay block to allow the delay block to be moved for the distributed pipelining optimization or absorbed during delay absorption.

Allow Delay Distribution Setting	Description
`'inherit'` (default)	Use the Allow design delay distribution setting of the model.
`'on'`	Allow the delay block to be moved or absorbed in optimizations.
`'off'`	Do not allow the delay block to be moved or absorbed in optimizations.

Set Allow Delay Distribution For a Delay or Unit Delay Block

To set whether a Delay or Unit Delay block is distributed or absorbed during distributed pipelining or delay absorption from the HDL Block Properties dialog box:

1 Right-click the subsystem and select **HDL Code** > **HDL Block Properties**.
2 Set **AllowDelayDistribution** to `inherit`, `on`, or `off`.

To set delay distribution for a Delay or Unit Delay block from the command line, use `hdlset_param`. For example, to not allow delay distribution for a Delay block, `'my_DUT/Delay1'`:

```
hdlset_param('my_DUT/Delay1','AllowDelayDistribution','off')
```

See also `hdlset_param`.

See Also

- Allow design delay distribution
- “Distributed Pipelining” on page 21-130
- “Delay Absorption During Delay Balancing” on page 21-83

BalanceDelays

The `BalanceDelays` subsystem parameter enables you to set delay balancing on a subsystem within a model.

BalanceDelays Setting	Description
'inherit' (default)	Use the delay balancing setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the delay balancing setting for the model.
'on'	Balance delays for this subsystem.
'off'	Do not balance delays for this subsystem, even if the parent subsystem has delay balancing enabled.

To learn more about delay balancing, see “Delay Balancing” on page 21-81.

Set Delay Balancing For a Subsystem

To set delay balancing for a subsystem using the HDL Block Properties dialog box:

1. Right-click the subsystem.
2. Select **HDL Code** > **HDL Block Properties**.
3. For **BalanceDelays**, select **inherit**, **on**, or **off**.

To set delay balancing for a subsystem from the command line, use `hdlset_param`. For example, to turn off delay balancing for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'BalanceDelays', 'off')
```

See also `hdlset_param`.

ClockRatePipelining

The `ClockRatePipelining` subsystem parameter enables you to set clock-rate pipelining on a subsystem within a model.

Clock-Rate Pipelining Setting	Description
'inherit' (default)	Use the clock-rate pipelining setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the clock-rate pipelining setting for the model.
'on'	Insert clock-rate pipelines for this subsystem.
'off'	Do not insert clock-rate pipelines for this subsystem, even if the parent subsystem has clock-rate pipelining enabled.

To disable clock-rate pipelining for a subsystem within a model, set `ClockRatePipelining` to `off` for that subsystem.

To learn how to set model-level clock-rate pipelining, see Clock-rate pipelining.

Set Clock-Rate Pipelining For a Subsystem

To set clock-rate pipelining for a subsystem using the HDL Block Properties dialog box:

1. Right-click the subsystem.
2. Select **HDL Code** > **HDL Block Properties**.
3. Set **ClockRatePipelining** to `inherit`, `on`, or `off`.

To set clock-rate pipelining for a subsystem from the command line, use `hdlset_param`. For example, to turn off clock-rate pipelining for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'ClockRatePipelining', 'off')
```

See Also

- “Clock-Rate Pipelining” on page 21-148
- “Increase Clock Frequency Using Clock-Rate Pipelining” on page 21-153

CodingStyle

When you use Multiport Switch blocks, use the `CodingStyle` parameter to specify whether you want to generate HDL code with if-else or case statements. By default, HDL Coder generates if-else statements. If you have several Multiport Switch blocks in your model, you can choose to specify a different `CodingStyle` for each block.

CodingStyle Setting	Description
`'ifelse_stmt'`(*Default*)	Generate if-else statements in the Verilog or SystemVerilog code or when-else statements in the VHDL code for a Multiport Switch block.
`'case_stmt'`	Generate case statements in the Verilog or SystemVerilog code or case-when statements in the VHDL code for a Multiport Switch block.

Set CodingStyle For Multiport Switch Block

To set CodingStyle for a Multiport Switch using the HDL Block Properties dialog box:

1. Right-click the Multiport Switch block.
2. Select **HDL Code** > **HDL Block Properties**.
3. For **CodingStyle**, select `ifelse_stmt` or `case_stmt`.

To see the `CodingStyle` specified for a subsystem from the command line, use `hdlget_param`. For example, to see the settings specified for a Multiport Switch block inside a subsystem, `my_dut`:

```
hdlget_param('my_dut/Multiport Switch', 'CodingStyle')
```

```
ans =

    'case_stmt'
```

See also `hdlset_param`.

ConstMultiplierOptimization

The ConstMultiplierOptimization implementation parameter lets you specify use of canonical signed digit (CSD) or factored CSD optimizations for processing coefficient multiplier operations in the generated code.

The following table shows the ConstMultiplierOptimization parameter values.

ConstMultiplierOptimization Setting	Description
'none' (*Default*)	By default, HDL Coder does not perform CSD or FCSD optimizations. Code generated for the Gain block retains multiplier operations.
'CSD'	When you specify this option, the generated code decreases the area used by the model while maintaining or increasing clock speed, using canonical signed digit (CSD) techniques. CSD replaces multiplier operations with add and subtract operations. CSD minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.
'FCSD'	This option uses factored CSD (FCSD) techniques, which replace multiplier operations with shift and add/subtract operations on certain factors of the operands. These factors are generally prime but can also be a number close to a power of 2, which favors area reduction. This option lets you achieve a greater area reduction than CSD, at the cost of decreasing clock speed.
'auto'	When you specify this option, HDL Coder chooses between the CSD or FCSD optimizations. The coder chooses the optimization that yields the most area-efficient implementation, based on the number of adders required. When you specify 'auto', the coder does not use multipliers, unless conditions are such that CSD or FCSD optimizations are not possible (for example, if the design uses floating-point arithmetic).

The ConstMultiplierOptimization parameter is available for the following blocks:

- Gain
- Stateflow chart
- Truth Table
- MATLAB Function
- MATLAB System

ConstrainedOutputPipeline

Use the ConstrainedOutputPipeline parameter to specify a nonnegative number of registers to place at the block outputs.

HDL Coder moves existing delays within your design to try to meet your constraint. New registers are not added. If there are fewer registers than the coder needs to satisfy your constraint, the coder reports the difference between the number of desired and actual output registers. You can add delays to your design using input or output pipelining.

Distributed pipelining does not redistribute registers you specify with constrained output pipelining.

How to Specify Constrained Output Pipelining

To specify constrained output pipelining for a block using the GUI:

1. Right-click the block and select **HDL Code > HDL Block Properties**.
2. For **ConstrainedOutputPipeline**, enter the number of registers you want at the output ports.

To specify constrained output pipelining, at the command line, enter:

```
hdlset_param(path_to_block,
             'ConstrainedOutputPipeline', number_of_output_registers)
```

For example, to constrain 6 registers at the output ports of a subsystem, `subsys`, in your model, `mymodel`, enter:

```
hdlset_param('mymodel/subsys','ConstrainedOutputPipeline', 6)
```

See Also

- “Constrained Output Pipelining” on page 21-146

DistributedPipelining

The `DistributedPipelining` subsystem parameter enables pipeline register distribution, which is a speed optimization that increases your clock speed by reducing your critical path on a subsystem. This optimization moves the delays within a subsystem while preserving the hierarchy.

DistributedPipelining Setting	Description
`'inherit'` (default)	Use the distributed pipelining setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the distributed pipelining setting for the model.
`'off'`	HDL Coder does not distribute the pipeline registers placed manually or by using the HDL block properties InputPipeline or OutputPipeline.
`'on'`	HDL Coder distributes registers inside the subsystem, MATLAB Function block, or Stateflow chart based on critical path analysis. This setting distributes the pipelines in the subsystem that are already placed manually or by using the HDL Block Properties InputPipeline or OutputPipeline.

Tip Output data might be in an invalid state initially if you insert pipeline registers. To avoid test bench errors resulting from initial invalid samples, disable output checking for those samples. For more information, see **Ignore output data checking (number of samples)**.

To enable distributed pipelining for a subsystem within a model, set `DistributedPipelining` to `on` for that subsystem.

To learn how to set model-level distributed pipelining, see Distributed pipelining.

Set Distributed Pipelining For a Subsystem

To set distributed pipelining for a subsystem using the HDL Block Properties dialog box:

1. Right-click the subsystem.
2. Select **HDL Code > HDL Block Properties**.
3. Set **DistributedPipelining** to `inherit`, `on`, or `off`.

To set distributed pipelining for a subsystem from the command line, use the `hdlset_param` function. For example, enter:

```
hdlset_param('path/to/subsystem', 'DistributedPipelining', 'on')
```

See Also

- “Distributed Pipelining” on page 21-130
- “Distributed Pipelining: Speed Optimization” on page 21-139
- “Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks” on page 27-28

DotProductStrategy

If you use the Product block for matrix multiplication in your design, use the `DotProductStrategy` to specify how you want to implement the matrix multiplication.

The `DotProductStrategy` options are listed in the following table.

DotProductStrategy Value	Description
`'Fully Parallel'` (default)	Expands the matrix multiplication operation into multipliers and adders. For example, if you multiply two 2x2 matrices, the implementation uses eight multipliers and four adders to compute the result. To share the multipliers to optimize area, use the HDL block property StreamingFactor. **Note** The **DotProductStrategy** must be set to `'Fully Parallel'` when you use the `Native Floating Point` mode.

DotProductStrategy Value	Description
`'Fully Parallel Scalarized'`	Expands the matrix multiplication operation into multipliers and adders. For example, if you multiply two 2x2 matrices, the implementation uses eight multipliers and four adders to compute the result. Use this option for smaller sized matrices and when you want to enable sharing on both the multipliers and adders. To share the multipliers to optimize area, use the HDL block property SharingFactor.
`'Serial Multiply-Accumulate'`	Uses the **Serial** architecture of the Multiply-Accumulate block to implement the matrix multiplication. In this architecture, the clock rate must be faster than the clock rate that you specify with **Parallel** architecture. You can see the clock rate in the Clock Summary information of the Code Generation report.
`'Parallel Multiply-Accumulate'`	Uses the **Parallel** architecture of the Multiply-Accumulate block to implement the matrix multiplication.

DSPStyle

`DSPStyle` enables you to generate code that includes synthesis attributes for multiplier mapping in your design. You can choose whether to map a particular block’s multipliers to DSPs or logic in hardware.

For Xilinx targets, the generated code uses the `use_dsp` attribute. For Altera targets, the generated code uses the `multstyle` attribute.

The `DSPStyle` options are listed in the following table.

DSPStyle Value	Description
`'none'` (default)	Do not insert a DSP mapping synthesis attribute.
`'on'`	Insert synthesis attribute that directs the synthesis tool to map to DSPs in hardware.
`'off'`	Insert synthesis attribute that directs the synthesis tool to map to logic in hardware.

The `DSPStyle` parameter is available for the following blocks:

- Gain
- Product
- Product of Elements with Architecture set to Tree
- Subsystem

- Atomic Subsystem
- Variant Subsystem
- Enabled Subsystem
- Triggered Subsystem
- Model with Architecture set to `ModelReference`

Hierarchy Flattening Behavior

If you specify hierarchy flattening for a subsystem that also has a nondefault `DSPStyle` setting, HDL Coder propagates the `DSPStyle` setting to the parent subsystem.

If the flattened subsystem contains Gain, Product, or Product of Elements blocks, the coder keeps their nondefault `DSPStyle` settings, and replaces default `DSPStyle` settings with the flattened subsystem `DSPStyle` setting.

Synthesis Attributes in Generated Code

The generated code for synthesis attributes depends on:

- Target language
- `DSPStyle` value
- `SynthesisTool` value

The following table shows examples of synthesis attributes in generated code.

DSPStyle Value	TargetLanguage Value	SynthesisTool Value	
		`'Altera Quartus II'`	`'Xilinx ISE'` `'Xilinx Vivado'`
`'none'`	`'Verilog'`	`wire signed [32:0] m4_out1;`	`wire signed [32:0] m4_out1;`
	`'VHDL'`	`m4_out1 : signal;`	`m4_out1 : signal;`
`'on'`	`'Verilog'`	`(* multstyle = "dsp" *) wire signed [32:0] m4_out1;`	`(* use_dsp = "yes" *) wire signed [32:0] m4_out1;`

	`'VHDL'`	`attribute multstyle : string ;`
		
		`attribute multstyle of m4_out1 : signal is "dsp" ;`
			`attribute use_dsp : string ;`
			
			`attribute use_dsp of m4_out1 : signal is "yes" ;`

`'off'`	`'Verilog'`	`(* multstyle = "logic" *) wire signed [32:0] m4_out1;`	`(* use_dsp = "no" *) wire signed [32:0] m4_out1;`

	`'VHDL'`	`attribute multstyle : string ;`
		
		`attribute multstyle of m4_out1 : signal is "logic" ;`
			`attribute use_dsp : string ;`
			
			`attribute use_dsp of m4_out1 : signal is "no" ;`

Requirement For Synthesis Attribute Specification

You must specify a synthesis tool by using the `SynthesisTool` property.

How To Specify a Synthesis Attribute

To specify a synthesis attribute using the HDL Block Properties dialog box:

1. Right-click the block.
2. Select **HDL Code > HDL Block Properties**.
3. For **DSPStyle**, select **on**, **off**, or **none**.

To specify a synthesis attribute from the command line, use `hdlset_param`. For example, suppose you have a model, `my_model`, with a DUT subsystem, `my_dut`, that contains a Gain block, `my_multiplier`. To insert a synthesis attribute to map `my_multiplier` to a DSP, enter:

```
hdlset_param('my_model/my_dut/my_multiplier', 'DSPStyle', 'on')
```

See also `hdlset_param`.

Limitations For Synthesis Attribute Specification

- When you specify a nondefault `DSPStyle` block property, the `ConstMultiplierOptimization` property must be set to `'none'`.
- Inputs to multiplier components cannot use the `double` data type.
- Gain constant cannot be a power of 2.

FlattenHierarchy

`FlattenHierarchy` enables you to remove subsystem hierarchy from the HDL code generated from your design.

FlattenHierarchy Setting	Description
`'inherit'` (default)	Use the hierarchy flattening setting of the parent subsystem. If this subsystem is the highest-level subsystem, do not flatten.
`'on'`	Flatten this subsystem.
`'off'`	Do not flatten this subsystem, even if the parent subsystem is flattened.

To flatten hierarchy, you must also have the `MaskParameterAsGeneric` global property set to `'off'`. For more information, see Generate parameterized HDL code from masked subsystem.

How To Flatten Hierarchy

To set hierarchy flattening using the HDL Block Properties dialog box:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the Subsystem and then click **HDL Block Properties**. For **FlattenHierarchy**, select **on**, **off**, or **inherit**.
- Right-click the Subsystem and select **HDL Code > HDL Block Properties**. For **FlattenHierarchy**, select **on**, **off**, or **inherit**.

To set hierarchy flattening from the command line, use `hdlset_param`. For example, to turn on hierarchy flattening for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'FlattenHierarchy', 'on')
```

See also `hdlset_param`.

Limitations For Hierarchy Flattening

A subsystem cannot be flattened if the subsystem is:

- A Synchronous Subsystem or uses the State Control block in `Synchronous` mode.
- A model reference implementation.
- A Triggered Subsystem when Use trigger signal as clock is enabled.

GuardIndexVariables

`GuardIndexVariables` enables you to specify whether to hoist array indices out of conditional statements or not. When you enable certain optimizations such as RAM Mapping, loop streaming, sharing, and so on, expressions are moved out of the array indices. A temporary variable is created for the expression that might result in an index out-of- bounds error during simulation. When you enable this option the generated code might be inefficient for your target hardware.

GuardIndexVariables Setting	Description
`'off'` (default)	Move the expression for array indices out of the conditional statement and create a temporary variable for the expression.
`'on'`	Do not move the expression for array indices out of the conditional statement and do not create a temporary variable for the expression.

This image shows the generated code with the option enabled and disabled. When the `GuardIndexVariable` option is `off`, the array index variable `i` is not decided by the conditional loop and might go out of bounds. When the `GuardIndexVariable` option is `on`, the array index variable `idx` is decided by the conditional loop preventing an array index out of bounds error during simulation.

```
if(u)
    x = A(i) * b;
else
    x = 0;
end
```

Source Program

```
tmp <= A(i) * b;
x1 <= tmp if u else 0;
```

GuardIndexVariables 'off'

```
idx <= i if u else 0;
tmp <= A(idx) * b;
x1 <= tmp if u else 0;
```

GuardIndexVariables 'on'

If you get an index access violation error during simulation, use this option.

InputPipeline

`InputPipeline` lets you specify a implementation with input pipelining for selected blocks. The parameter value specifies the number of input pipeline stages (pipeline depth) in the generated code.

Distributed pipelining can move input pipelines to optimize your design. To prevent distributed pipelining from moving input pipelines from a specified point in your design, use the ConstrainedOutputPipeline parameter.

The following code specifies an input pipeline depth of two stages for each Sum block in the model:

```
sblocks = find_system(gcb, 'BlockType', 'Sum');
for ii=1:length(sblocks),hdlset_param(sblocks{ii},'InputPipeline', 2), end;
```

Note The `InputPipeline` setting does not have any effect on blocks that do not have an input port.

When generating code for pipeline registers, HDL Coder appends a postfix string to names of input or output pipeline registers. The default postfix string is `_pipe`. To customize the postfix string, use the **Pipeline postfix** option in the **Global Settings / General** pane in the **HDL Code Generation** pane of the Configuration Parameters dialog box. Alternatively, you can pass the desired postfix as a character vector in the `makehdl` property `PipelinePostfix`. For an example, see Pipeline postfix.

InstantiateFunctions

For the MATLAB Function block, you can use the **InstantiateFunctions** parameter to generate a VHDL `entity`, Verilog or SystemVerilog `module` for each function. HDL Coder generates code for each `entity` or `module` in a separate file.

The **InstantiateFunctions** options for the MATLAB Function block are listed in the following table.

InstantiateFunctions Setting	Description
`'off'` (default)	Generate code for functions inline.
`'on'`	Generate a VHDL `entity`, Verilog or SystemVerilog `module` for each function, and save each `module` or `entity` in a separate file.

How To Generate Instantiable Code for Functions

To set the **InstantiateFunctions** parameter using the HDL Block Properties dialog box:

1. Right-click the MATLAB Function block.
2. Select **HDL Code** > **HDL Block Properties**.
3. For **InstantiateFunctions**, select **on**.

To set the **InstantiateFunctions** parameter from the command line, use `hdlset_param`. For example, to generate instantiable code for functions in a MATLAB Function block, `myMatlabFcn`, in your DUT subsystem, `myDUT`, enter:

```
hdlset_param('my_DUT/my_MATLABFcnBlk', 'InstantiateFunctions', 'on')
```

Generate Code Inline for Specific Functions

If you want to generate instantiable code for some functions but not others, enable the option to generate instantiable code for functions, and use `coder.inline`. See `coder.inline` for details.

Limitations for Instantiable Code Generation for Functions

The software generates code inline when:

- Function calls are within conditional code or `for` loops.
- Any function is called with a nonconstant `struct` input.
- The function has state, such as a persistent variable, and is called multiple times.
- There is an enumeration anywhere in the design function.

InstantiateStages

For a `Cascade` architecture, you can use the **InstantiateStages** parameter to generate a VHDL `entity` or Verilog `module` for each computation stage. HDL Coder generates code for each `entity` or `module` in a separate file.

InstantiateStages Setting	Description
`'off'` (default)	Generate cascade stages in a single VHDL `entity` or Verilog `module`.
`'on'`	Generate a VHDL `entity` or Verilog `module` for each cascade stage, and save each `module` or `entity` in a separate file.

LoopOptimization

`LoopOptimization` enables you to stream or unroll loops in code generated from a MATLAB Function block. Loop streaming optimizes for area; loop unrolling optimizes for speed.

Note If you specify the `MATLAB Datapath` architecture of the MATLAB Function block, you can only unroll loops. To stream loops, you can use the streaming optimization by specifying a **StreamingFactor**. See "HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture" on page 21-205.

LoopOptimization Setting	Description
`'none'` (default)	Do not optimize loops.
`'Unrolling'`	Unroll loops.
`'Streaming'`	Stream loops.

How to Optimize MATLAB Function Block For Loops

To select a loop optimization using the HDL Block Properties dialog box:

1. Right-click the MATLAB Function block.
2. Select **HDL Code** > **HDL Block Properties**.
3. For **LoopOptimization**, select `none`, `Unrolling`, or `Streaming`.

To select a loop optimization from the command line, use `hdlset_param`. For example, to turn on loop streaming for a MATLAB Function block, `my_mlfn`:

```
hdlset_param('my_mlfn', 'LoopOptimization', 'Streaming')
```

See also `hdlset_param`.

Limitations for MATLAB Function Block Loop Optimization

HDL Coder cannot stream a loop if:

- The loop index counts down. The loop index must increase by 1 on each iteration.
- There are 2 or more nested loops at the same level of hierarchy within another loop.
- Any particular persistent variable is updated both inside and outside a loop.

HDL Coder can stream a loop when the persistent variable is:

- Updated inside the loop and read outside the loop.
- Read within the loop and updated outside the loop.

LUTRegisterResetType

Use the `LUTRegisterResetType` block parameter to control synthesis of a LUT into a ROM structure on an FPGA.

LUTRegisterResetType Value	Description
`default`	LUT output register has default reset logic. When you generate HDL, the LUT will be synthesized as registers.
`none`	LUT output register has no reset logic. When you generate HDL, the LUT will be synthesized as a ROM.

You can specify `LUTRegisterResetType` for the following blocks:

- Gamma Correction
- Lookup Table

MapPersistentVarsToRAM

With the `MapPersistentVarsToRAM` implementation parameter, you can use RAM-based mapping for persistent arrays of a MATLAB Function block instead of mapping to registers.

MapPersistentVarsToRAM Setting	Mapping Behavior
`off`	Persistent arrays map to registers in the generated HDL code.
`on`	Persistent array variables map to RAM. For restrictions, see “RAM Mapping Restrictions” on page 19-17.

RAM Mapping Restrictions

When you enable RAM mapping, a persistent array or user-defined System object private property maps to a block RAM when all of these conditions are true:

- Each read or write access is for a single element only. For example, submatrix access and array copies are not supported.

- Address computation logic is not read-dependent. For example, computation of a read or write address using the data read from the array is not supported.
- Persistent variables or user-defined System object private properties are initialized to 0 if they have a cyclic dependency. For example, if you have two persistent variables, A and B, you have a cyclic dependency if A depends on B, and B depends on A.
- If an access is within a conditional statement, the conditional statement uses only simple logic expressions (&&, `||`, ~) or relational operators. For example, in this code, `r1` does not map to RAM:

  ```
  if (mod(i,2) > 0)
      a = r1(u);
  else
      r1(i) = u;
  end
  ```

  You can rewrite complex conditions, such as conditions that call functions, by assigning them to temporary variables, and using the temporary variables in the conditional statement. For example, to map `r1` to RAM, rewrite the previous code as:

  ```
  temp = mod(i,2);
  if (temp > 0)
      a = r1(u);
  else
      r1(i) = u;
  end
  ```
- The persistent array or user-defined System object private property value depends on external inputs.

  For example, in the following code, `bigarray` does not map to RAM because it does not depend on `u`:

  ```
  function z = foo(u)

  persistent cnt bigarray
  if isempty(cnt)
      cnt = fi(0,1,16,10,hdlfimath);
      bigarray = uint8(zeros(1024,1));
  end
  z = u + cnt;
  idx = uint8(cnt);
  temp = bigarray(idx+1);
  cnt(:) = cnt + fi(1,1,16,0,hdlfimath) + temp;
  bigarray(idx+1) = idx;
  ```
- The RAM size is greater than or equal to the `RAMMappingThreshold` value. The RAM size is the product of `Array Size * Word Length * Complexity`, where:
  - `Array Size` is the number of elements in the array.
  - `Word Length` is the number of bits that represent the data type of the array.
  - `Complexity` is 2 for a complex data type or 1 for a real datatype.
- Access to the persistent variable that you are mapping to RAM is not in a loop, such as a `for` loop, unless the loop is unrolled. For more information, see `coder.unroll`.
- Access to the persistent variable that you are mapping to RAM is not in a nested conditional statement, such as a nested `if` statement or nested `switch` statement.

If any of the above conditions is false, the persistent array or user-defined System object private property maps to a register in the HDL code.

RAMMappingThreshold

The default value of RAMMappingThreshold is 256. To change the threshold, use hdlset_param. For example, the following command changes the mapping threshold for the sfir_fixed model to 128 bits:

```
hdlset_param('sfir_fixed', 'RAMMappingThreshold', 128);
```

You can also change the RAM mapping threshold in the Configuration Parameters dialog box. For more information, see **RAM mapping threshold**.

Example

For an example that shows how to map persistent array variables to RAM in a MATLAB Function block, see “RAM Mapping with the MATLAB Function Block” on page 21-125.

MapToRAM

Use the MapToRAM property to map lookup tables (LUT) to RAM.

When **Simulate RAM Delay** is enabled, the MapToRAM property is disabled for the Sine HDL Optimized and Cosine HDL Optimized block.

MapToRAM Setting	Mapping Behavior
inherit (default)	The model setting **Map lookup tables to RAM** behavior is used.
off	The block lookup tables (LUTs) are mapped to logic slices on the FPGA.
on	The block lookup tables (LUTs) are mapped to RAM.

MapToRAM Considerations for Added Latency

When you enable either the block property MapToRAM or the global option LUTMapToRAM and the synthesis tool for the model is Xilinx or Altera, a unit delay without reset is added after the Lookup Table block. Because of the added unit delay, additional latency is inserted and delay balanced in the generated HDL Code and generated model. To avoid extra latency, you can add a Delay block with the HDL Block property **ResetType** set to **none** after the Lookup Table block in the original model.

OutputPipeline

OutputPipeline lets you specify a implementation with output pipelining for selected blocks. The parameter value specifies the number of output pipeline stages (pipeline depth) in the generated code. Distributed pipelining can move output pipelines to optimize your design. To prevent distributed pipelining from moving output pipelines from a specified point in your design, use the ConstrainedOutputPipeline parameter.

This code specifies an output pipeline depth of two stages for each Sum block in the model:

```
sblocks = find_system(gcb, 'BlockType', 'Sum');
for ii=1:length(sblocks),hdlset_param(sblocks{ii},'OutputPipeline', 2), end;
```

Note The `OutputPipeline` setting does not have any effect on blocks that do not have an output port.

When generating code for pipeline registers, HDL Coder appends a postfix string to names of input or output pipeline registers. The default postfix string is `_pipe`. To customize the postfix string, use the **Pipeline postfix** option in the Configuration Parameters dialog box, in the **HDL Code Generation > Global Settings > General** tab. Alternatively, you can use the `PipelinePostfix` property with `makehdl`. For an example, see Pipeline postfix.

See also “Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks” on page 27-28.

PreserveUpstreamLogic

Control the removal of unconnected logic. Unconnected logic is logic that is connected upstream from a terminator block. This property is available for Terminator blocks only.

PreserveUpstreamLogic Options	PreserveUpstreamLogic Behavior
`off` (default)	Logic connected upstream from the Terminator block is not preserved in HDL code generation.
`on`	Logic connected upstream from the Terminator block is preserved in HDL code generation.

For an example, see the "Upstream Logic Preservation of Unused Port" section of “Optimize Unconnected Ports in HDL Code for Simulink Models” on page 21-246.

RAMDirective

Using **RAMDirective**, you can specify whether you want to map the Random Access Memory (RAM) blocks in your Simulink model to FPGA RAM memory blocks. You can map large memory blocks such as `ultra` from the Xilinx family and M144k from the Quartus family. Specify these **RAMDirective** values for the RAM blocks in your design based on the synthesis tool.

Synthesis Tool	RAM Style Attribute	RAMDirective Values
Quartus	`ramstyle`	`logic| M9K| M10K| M20K| M144K| MLAB`
Xilinx	`ram_style`	`block| distributed| register| ultra`
Microchip	`syn_ramstyle`	`block_ram| registers| lsram| uram`

When you specify a value for this setting, HDL Coder generates a `ramstyle` attribute in the HDL code. This attribute specifies the type of RAM memory unit that you want the synthesis tool to use when inferring the RAM blocks in your design.

When you do not specify a value for this setting, HDL Coder does not generate the `ramstyle` attribute in the HDL code. The synthesis tool determines the type of inferred RAM for mapping the RAM blocks in your model.

Set RAMDirective for RAM Blocks

In the **HDL RAMs** library, except for the Dual Rate Dual Port RAM, you can specify the `RAMDirective` property for all other RAM blocks.

To set `RAMDirective` for a RAM block in the HDL Block Properties dialog box:

1. Right-click the RAM block.
2. Select **HDL Code > HDL Block Properties**.
3. In **RAMDirective** property, specify the values as listed in the table.

You can also set the **RAMDirective** by using `hdlset_param` function.

```
hdlset_param(<ram_block_name>, ‘RAMDirective’, <attribute_value>);
```

To specify these attributes in the MATLAB HDL workflow, use the `RAMDirective` parameter value pair for `hdl.RAM` instantiation. Set this property by using this command:

```
hRam = hdl.RAM(‘RAMType’, ‘Single Port RAM’, ‘RAMDirective’, ‘ultra’);
```

For example, generate an HDL attribute for mapping the RAM blocks in your model to `block RAM`. A `block RAM` is a dedicated memory unit on the FPGA. Sizes of block RAMs can be `4kb`, `8kb`, `16kb`, and `32kb`.

To map your RAM blocks to `block RAM`:

- Specify the synthesis tool. You must target a Xilinx device that contains `block RAM` resources. If the target device does not contain block RAMs, the synthesis tool ignores this attribute and might infer the RAM as distributed RAMs or Lookup Table (LUT) slices.
- Enter the RAMDirective value as `block` for your Simulink RAM blocks in HDL Block Properties.
- Generate the HDL code for your model.

This generated VHDL code shows the `ram_style` attribute is set to `block`:

```
 attribute ram_style: string;
 attribute ram_style of ram : signal is "block";
```

This generated Verilog or SystemVerilog code shows the `ram_style` attribute is set to `block`:

```
(* ram_style = "block" *)
```

Dependency

When using **RAMDirective** property, make sure to select the synthesis tool for your design.

ResetType

Use the `ResetType` block parameter to suppress reset logic generation.

ResetType Value	Description
`default`	Generate reset logic.
`none`	Do not generate reset logic. Reset is not applied to generated registers. Therefore, mismatches between Simulink and the generated code occur for some number of samples during the initial phase, when registers are not fully loaded. To avoid test bench errors during the initial phase, determine the number of samples required to fully load the registers. Then, set the **Ignore output data checking (number of samples)** option accordingly. See also Ignore output data checking (number of samples).

You can specify `ResetType` for the following blocks:

- Chart
- Convolutional Deinterleaver
- Convolutional Interleaver
- Delay
- Delay (DSP System Toolbox)
- General Multiplexed Deinterleaver
- General Multiplexed Interleaver
- MATLAB Function
- MATLAB System
- Memory
- Tapped Delay
- Truth Table
- Unit Delay Enabled
- Unit Delay

Reset Logic for Optimizations in the MATLAB Function Block

When you set **ResetType** to `none` for a MATLAB Function block, HDL Coder does not generate reset logic for persistent variables in the MATLAB code.

However, if you specify other optimizations for the block, the coder may insert registers that use reset logic. The coder does not suppress reset logic generation for these registers. Therefore, if you set **ResetType** to `none` along with other block optimizations, your generated code may have a reset port at the top level.

How to Suppress Reset Logic Generation

To suppress reset logic generation for a block using the UI:

1. Right-click the block and select **HDL Code > HDL Block Properties**.
2. For **ResetType**, select `none`.

To suppress reset logic generation, on the command line, enter:

```
hdlset_param(path_to_block,'ResetType','none')
```

For example, to suppress reset logic generation for a Unit Delay block, `UnitDelay1`, within a subsystem, `mySubsys`, on the command line, enter:

```
hdlset_param('mySubsys/UnitDelay1','ResetType','none');
```

Specify Synchronous or Asynchronous Reset

To specify a synchronous or asynchronous reset, use the `ResetType` model-level parameter. For details, see Reset type.

SerialPartition

Use this parameter on Min/Max blocks to specify partitions for a serial cascade architecture. The default setting uses the minimum number of partitions.

To Generate This Architecture...	Set SerialPartition to...
Cascade-serial with explicitly specified partitioning	`[p1 p2 p3...pN]`: a vector of `N` integers, where `N` is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the input data vector. The values of the vector elements must be in descending order, except the last two elements can be equal. For example, for an input of 8 elements, partitions `[5 3]` or `[4 2 2]` are legal, but the partitions `[2 2 2 2]` or `[3 2 3]` raise an error at code generation time.
Cascade-serial with automatically optimized partitioning	`0`

This property is also used for serial filter architectures. For how to configure filter blocks, see “SerialPartition” on page 19-43.

SharingFactor

Use `SharingFactor` to specify the number of functionally equivalent resources to map to a single shared resource. The default is 0. See “Resource Sharing” on page 21-45.

SoftReset

Use the `SoftReset` block parameter to specify whether to generate hardware-friendly synchronous reset logic, or local reset logic that matches the Simulink simulation behavior. This property is available for the Unit Delay Resettable block or Unit Delay Enabled Resettable block.

SoftReset Value	Description
off (default)	Generate local reset logic that matches the Simulink simulation behavior.
on	Generate synchronous reset logic for the block. This option generates code that is more efficient for synthesis, but does not match the Simulink simulation behavior.

When SoftReset set to 'off', the following code is generated for a Unit Delay Resettable block :

```
always @(posedge clk or posedge reset)
  begin : Unit_Delay_Resettable_process
    if (reset == 1'b1) begin
      Unit_Delay_Resettable_zero_delay <= 1'b1;
      Unit_Delay_Resettable_switch_delay <= 2'b00;
    end
    else begin
      if (enb) begin
        Unit_Delay_Resettable_zero_delay <= 1'b0;
        if (UDR_reset == 1'b1) begin
          Unit_Delay_Resettable_switch_delay <= 2'b00;
        end
        else begin
          Unit_Delay_Resettable_switch_delay <= In1;
        end
      end
    end
  end

assign Unit_Delay_Resettable_1 =
            (UDR_reset ||
             Unit_Delay_Resettable_zero_delay ? 1'b1 : 1'b0);
assign out0 = (Unit_Delay_Resettable_1 == 1'b1 ? 2'b00 :
            Unit_Delay_Resettable_switch_delay);
```

When SoftReset set to 'on', the following code is generated for a Unit Delay Resettable block :

```
always @(posedge clk or posedge reset)
  begin : Unit_Delay_Resettable_process
    if (reset == 1'b1) begin
      Unit_Delay_Resettable_reg <= 2'b00;
    end
    else begin
      if (enb) begin
        if (UDR_reset != 1'b0) begin
          Unit_Delay_Resettable_reg <= 2'b00;
        end
        else begin
          Unit_Delay_Resettable_reg <= In1;
        end
      end
    end
  end

assign out0 = Unit_Delay_Resettable_reg;
```

StreamingFactor

Number of parallel data paths, or vectors, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming” on page 21-42.

UseRAM

The `UseRAM` implementation parameter enables using RAM-based mapping for a Delay block instead of mapping to a shift register.

<table>
<tr><th>UseRAM Setting</th><th>Mapping Behavior</th></tr>
<tr><td><code>off</code></td><td>The delay maps to a shift register in the generated HDL code, except in one case. For details, see “Effects of Streaming and Distributed Pipelining” on page 19-27.</td></tr>
<tr><td><code>on</code></td><td>The delay maps to a dual-port RAM block when the following conditions are true:

Initial value of the delay is zero.
The Delay block does not have an external reset or enable port.
Delay length greater than or equal to 4.
Delay has one of the following set of numeric and data type attributes:

(a) Real scalar with a non-floating-point data type (such as signed integer, unsigned integer, fixed point, or Boolean)
(b) Complex scalar with real and imaginary parts that use non-floating-point data type
(c) Vector where each element is either (a) or (b)

<code>RAMSize</code> is greater than or equal to the <code>RAMMappingThreshold</code> value. <code>RAMSize</code> is the product of <code>DelayLength * WordLength * VectorLength * Complexity</code>.

<code>DelayLength</code> is the number of delays that the Delay block specifies.
<code>WordLength</code> is the number of bits that represent the data type of the delay.
<code>VectorLength</code> is the vector length of the input to the Delay block.
<code>Complexity</code> is 2 for a complex data type or 1 for a real datatype.

For more information, see RAM mapping threshold.

If any condition is false, the delay maps to a shift register in the HDL code unless it merges with other delays to map to a single RAM. For more information, see “Mapping Multiple Delays to RAM” on page 19-26.</td></tr>
</table>

This implementation parameter is available for the Delay block in the Simulink Discrete library and the Delay block in the DSP System Toolbox Signal Operations library.

Mapping Multiple Delays to RAM

HDL Coder can also merge several delays of equal length into one delay and then map the merged delay to a single RAM. This optimization provides the following benefits:

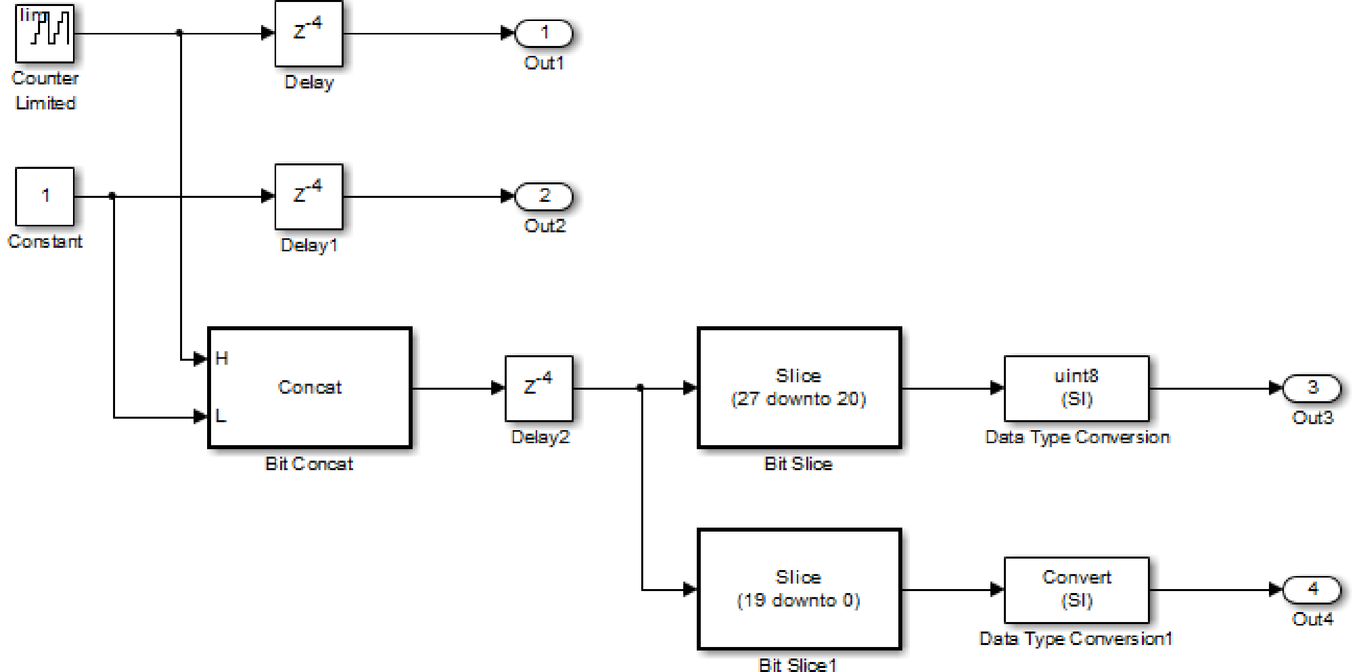
- Increased occupancy on a single RAM
- Sharing of address generation logic, which minimizes duplication of identical HDL code
- Mapping of delays to a RAM when the *individual* delays do not satisfy the threshold

The following rules control whether or not multiple delays can merge into one delay:

- The delays must:
  - Be at the same level of the subsystem hierarchy.
  - Use the same compiled sample time.
  - Have `UseRAM` set to `on`, or be generated by streaming or resource sharing.
  - Have the same `ResetType` setting, which cannot be `none`.
- The total word length of the merged delay cannot exceed 128 bits.
- The `RAMSize` of the merged delay is greater than or equal to the `RAMMappingThreshold` value. `RAMSize` is the product of `DelayLength * WordLength * VectorLength * Complexity`.

Example of Multiple Delays Mapping to a Block RAM

`RAMMappingThreshold` for this model is `100` bits.



The Delay and Delay1 blocks merge and map to a dual-port RAM in the generated HDL code by satisfying the following conditions:

- Both delay blocks:
  - Are at the same level of the hierarchy.
  - Use the same compiled sample time.
  - Have **UseRAM** set to `on` in the HDL block properties dialog box.
  - Have the same **ResetType** setting of `default`.
- The total word length of the merged delay is 28 bits, which is below the 128-bit limit.
- The `RAMSize` of the merged delay is 112 bits (4 delays * 28-bit word length), which is greater than the mapping threshold of 100 bits.

When you generate HDL code for this model, HDL Coder generates additional files to specify RAM mapping. HDL Coder stores these files in the same source location as other generated HDL files, for example, the `hdlsrc` folder.

Effects of Streaming and Distributed Pipelining

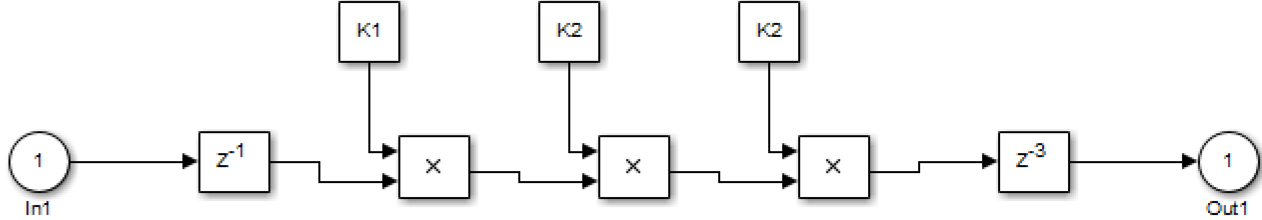
When `UseRAM` is `off` for a Delay block, HDL Coder maps the delay to a shift register by default. However, the coder changes the `UseRAM` setting to `on` and tries to map the delay to a RAM under the following conditions:

- Streaming is *enabled* for the subsystem with the Delay block.
- Distributed pipelining is *disabled* for the subsystem with the Delay block.

Suppose that distributed pipelining is *enabled* for the subsystem with the Delay block.

- When `UseRAM` is `off`, the Delay block participates in retiming.
- When `UseRAM` is `on`, the Delay block does not participate in retiming. HDL Coder does not break up a delay marked for RAM mapping.

  Consider a subsystem with two Delay blocks, three Constant blocks, and three Product blocks:



  When `UseRAM` is `on` for the Delay block on the right, that delay does not participate in retiming.

The following summary describes whether or not HDL Coder tries to map a delay to a RAM instead of a shift register.

UseRAM Setting for the Delay Block	Optimizations Enabled for Subsystem with Delay Block		
	Distributed Pipelining Only	Streaming Only	Both Distributed Pipelining and Streaming
On	Yes	Yes	Yes

UseRAM Setting for the Delay Block	Optimizations Enabled for Subsystem with Delay Block		
	Distributed Pipelining Only	Streaming Only	Both Distributed Pipelining and Streaming
Off	No	Yes, because mapping to a RAM instead of a shift register can provide an area-efficient design.	No

VariablesToPipeline

Warning `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

The VariablesToPipeline parameter enables you to insert a pipeline register at the output of one or more MATLAB variables. Specify a list of variables as a character vector, with spaces separating the variables.

See also “Pipeline MATLAB Expressions” on page 8-12.

HDL Block Properties: Native Floating Point

In this section...

"Overview" on page 19-29
"CheckResetToZero" on page 19-30
"DivisionAlgorithm" on page 19-30
"HandleDenormals" on page 19-31
"InputRangeReduction" on page 19-32
"LatencyStrategy" on page 19-33
"CustomLatency" on page 19-34
"NFPCustomLatency" on page 19-35
"MantissaMultiplyStrategy" on page 19-36
"MaxIterations" on page 19-37

Overview

Block implementation parameters enable you to control details of the code generated for specific block implementations. See "Set and View HDL Model and Block Parameters" on page 19-52 to learn how to select block implementations and parameters in the GUI or the command line.

Property names are specified as character vectors. The data type of a property value is specific to the property. This section describes the syntax of each block implementation parameter that you can specify in the **Native Floating Point** tab of the HDL Block Properties. You can see how specifying the parameter affects the generated code.

HDL Block Properties of Library Blocks

HDL block properties of library blocks are treated similar to mask parameters. When you instantiate library blocks in your model, the current HDL block properties of that library block are copied to instances of that block in your model. The HDL block properties of these instances are not synchronized with the HDL block properties of the library block. That is, if you change the HDL block property of the library block, the change does not get propagated to instances of the library block that you already added to your Simulink model. If you want the HDL block properties of a library block to be synchronized with its instances in the model, create a Subsystem and then place this block inside that Subsystem. The HDL block properties of blocks that reside inside the library block are synchronized with the corresponding instances in your model.

Suppose a library contains a Subsystem block with HDL architecture set to `Module`. When you instantiate this block in your model, the block instance uses `Module` as the HDL architecture. If you change the HDL architecture of the Subsystem block in the library to `BlackBox`, existing instances of that Subsystem block in your model still use `Module` as the HDL architecture. If you now add instances of the Subsystem block from the library in your model, the new block instances get a copy of the current HDL block properties, and therefore use `BlackBox` as the HDL architecture. If you want the HDL architecture of the Subsystem block in the library to be synchronized with its instances in the model, create a wrapper subsystem with the HDL architecture that you want inside this Subsystem.

CheckResetToZero

You can use the **CheckResetToZero** property for the mod and rem functions of the Math Function block in native floating-point mode. If you have numbers `a` and `b` such that the quotient `a/b` is close to an integer, this setting treats `a` as an integral multiple of `b`, and rem(a,b)=0. This result is numerically accurate and matches the Simulink simulation result. However, computing this result uses additional resources and increases the area footprint on the target FPGA device.

For example, for these sets of numbers, you get different simulation results when you enable and disable the **CheckResetToZero** setting.

CheckResetToZero Setting	Description
`'on'` (default)	When you compute mod or rem of two numbers whose quotient is closer to an integer, and has a precision greater than that of the floating point data type you use, HDL Coder adds the required logic to output the result of mod or rem as zero when the quotient of the numbers is close to an integer.
`'off'`	HDL Coder does not insert the additional logic to calculate the quotient, which saves area on the target FPGA device.

Set CheckResetToZero For the Math Function Block

To set **CheckResetToZero** for a block from the HDL Block Properties dialog box:

1. Right-click the block.
2. Select **HDL Code HDL Block Properties**.
3. For **CheckResetToZero**, select **on** or **off**.

To set **CheckResetToZero** for the Math Function block inside a subsystem, `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Math', 'CheckResetToZero, 'on')
```

See also `hdlset_param`.

DivisionAlgorithm

You can use the `DivisionAlgorithm` property when you enable `Native Floating Point` mode for the Divide block and the Math Function block in Reciprocal mode.

DivisionAlgorithm Setting	Description
`Radix-2` (default)	The default Radix-2 mode performs repeated subtractions by computing one bit of the quotient in each iteration. To design for lower area usage while trading off for latency, use the Radix-2 mode.

DivisionAlgorithm Setting	Description
`Radix-4`	The Radix-4 mode performs repeated subtractions by computing two bits of the quotient in each iteration. To compute the result, the Radix-4 mode uses half the number of iterations that is required by the Radix-2 mode. To design for lower latency while trading off for area, use the Radix-4 mode.

Single-Precision Division Resource Utilization and Maximum Clock Frequency on Xilinx Virtex-7

DivisionAlgorithm Mode	LatencyStrategy	Latency	Fmax	LUTs	Registers
Radix-2	MIN	17	334.4MHz	1248	1011
	MAX	32	454.5MHz	1294	1797
Radix-4	MIN	11	245.5MHz	1956	865
	MAX	20	453.1MHz	1854	1522

Specify DivisionAlgorithm For the Math Function or Division Block

To specify **DivisionAlgorithm** for a block from the HDL Block Properties dialog box:

1. Right-click the block.
2. Select **HDL Code** > **HDL Block Properties**.
3. In the **Native Floating Point** tab, specify the **DivisionAlgorithm**.

To specify **DivisionAlgorithm** for the block at the command line, use `hdlset_param`. For example, this command specifies Radix-4 mode for a Divide block inside a subsystem, `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Divide, 'DivisionAlgorithm', 'Radix-4')
```

HandleDenormals

You can use the `HandleDenormals` property for certain blocks that support HDL code generation in `Native Floating Point` mode. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. With this setting, you can specify whether you want HDL Coder to insert additional logic to handle the denormal numbers in your design. For more information, see “Denormal Numbers” on page 14-92.

HandleDenormals Setting	Description
`'inherit'` (default)	Use the handle denormals setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the handle denormals setting for the model.

HandleDenormals Setting	Description
'on'	If you have denormal numbers at these block inputs, HDL Coder adds the logic to normalize the denormal numbers.
'off'	HDL Coder does not insert additional logic to handle denormal numbers in your design. The code generator treats the denormal value as zero before performing any computation.

To enable `HandleDenormals` for a block within a model, set the parameter, `HandleDenormals`, to `'on'` for that block.

Set Handle Denormals For a Block

To set handle denormals for a block from the HDL Block Properties dialog box:

1. Right-click the block.
2. Select **HDL Code** > **HDL Block Properties**.
3. For **HandleDenormals**, select **inherit**, **on**, or **off**.

To set handle denormals for a block from the command line, use `hdlset_param`. For example, to enable handle denormals for a Product block inside a subsystem, `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'HandleDenormals', 'on')
```

See also Handle Denormals.

InputRangeReduction

You can use the **InputRangeReduction** property for the sin, cos, tan, sincos, and cos+jsin functions of the Trigonometric Function block in `Native Floating Point` mode. By default, this setting is enabled for the block, and it assumes that your input range is unbounded. If your input to the block is bounded in the range `[-pi, pi]`, your design does not require the logic to reduce the input range. In that case, you can disable this setting, and the block implementation incurs a lower latency and uses fewer resources on the target hardware. When you disable the setting, the generated model contains a block that verifies whether the inputs are bounded in the range `[-pi, pi]`.If you have unbounded inputs, the generated model triggers an assertion during simulation.

InputRangeReduction Setting	Description
'on' (default)	Assumes that the input range is unbounded and inserts additional logic to reduce the input argument range to `[-pi, pi]` before computing the algorithm.
'off'	Assumes that the input argument is bounded in the range `[-pi, pi]` and does not insert the additional logic to reduce the input argument range. This implementation reduces the latency and saves area on the target platform.

Set InputRangeReduction For the Trigonometric Function Block

To set **InputRangeReduction** for a block from the HDL Block Properties dialog box:

1. Right-click the block.
2. Select **HDL Code HDL Block Properties**.
3. In the **Native Floating Point** tab, for **InputRangeReduction**, select **on** or **off**.

To disable **InputRangeReduction** for the Trigonometric Function block inside a subsystem, `my_trigonometric` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/my_trigonometric', ...
                        'InputRangeReduction, 'off')
```

See also `hdlset_param`.

LatencyStrategy

You can use the `LatencyStrategy` property for certain blocks that support HDL code generation for fixed-point and floating-point types. When you use floating-point types, select the model configuration parameter **Use Floating Point**. For fixed-point types, the property specifies zero, maximum, or custom latency. For floating-point types, the property specifies whether the blocks map to minimum, maximum, or a custom latency for the operator.

LatencyStrategy Setting	Description
`'inherit'` (default)	Use the latency strategy setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the latency strategy setting for the model.
`'Max'`	During code generation, HDL Coder uses the maximum latency value for the native floating point operator.
`'Min'`	During code generation, HDL Coder uses the minimum latency value for the native floating point operator.
`'Zero'`	During code generation, HDL Coder does not add any latency for the native floating point operator.
`'Custom'`	During code generation, HDL Coder adds latency equal to the value that you specify for **CustomLatency** or **NFPCustomLatency** settings of the native floating point operator. You can use this setting for certain blocks in the native floating-point mode. To see the blocks for which you can specify the setting, see “NFPCustomLatency” on page 19-35.

To specify the minimum latency option for a block within a model, set the parameter, `LatencyStrategy`, to `'MIN'` for that block.

To learn how to set model-level latency strategy setting, see “Latency Considerations with Native Floating Point” on page 14-104.

Set Latency Strategy for Fixed-Point Blocks

When you use fixed-point types, you can specify the **LatencyStrategy** for these blocks.

- Divide and Reciprocal blocks that have `ShiftAdd` as the HDL architecture.
- Sqrt block that has `SqrtFunction` as the HDL architecture.
- Trigonometric Function block that has **Function** set to `sin`, `cos`, `sincos`, `cos+jsin`, or `atan2` and **Approximation method** as `CORDIC`.

To set latency strategy for a subsystem from the HDL Block Properties dialog box:

1. In the Simulink toolstrip, on the **Apps** tab, select **HDL Coder**.
2. Select the block, and on the **HDL Code** tab, click the **HDL Block Properties** button.
3. In the **General** tab, specify the **LatencyStrategy**. If you set **LatencyStrategy** to **Custom**, you must specify a value for the **CustomLatency**.

To specify the latency strategy for a block from the command line, use `hdlset_param`. For example, to specify the minimum latency for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'LatencyStrategy', 'MAX')
```

See also `hdlset_param`.

Set Latency Strategy for Floating-Point Block

To set latency strategy for a subsystem from the HDL Block Properties dialog box:

1. In the Simulink toolstrip, on the **Apps** tab, select **HDL Coder**.
2. Select the block, and on the **HDL Code** tab, click the **HDL Block Properties** button.
3. In the **Native Floating Point** tab, specify the **LatencyStrategy**. If you set **LatencyStrategy** to **Custom**, you must specify a value for the **NFPCustomLatency**.

For details, see the "HDL Code Generation" section of each block page. To learn about blocks for which you can specify a custom latency, see “NFPCustomLatency” on page 19-35.

To specify the latency strategy for a block from the command line, use `hdlset_param`. For example, to specify the minimum latency for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'LatencyStategy', 'MIN')
```

See also `hdlset_param`.

CustomLatency

You can specify a custom latency for certain blocks for fixed-point types. By using the custom latency strategy, you can trade-off between clock frequency and power consumption. To specify a custom latency strategy, set **LatencyStrategy** to `CUSTOM` and specify a value for **CustomLatency**. For details, see the "HDL Code Generation" section of each block page.

You can specify the **CustomLatency** setting for these blocks with fixed-point types.

- Divide and Reciprocal blocks that have `ShiftAdd` as the HDL architecture.
- Sqrt block that has `SqrtFunction` as the HDL architecture.
- Trigonometric Function block that has **Function** set to `sin`, `cos`, `sincos`, `cos+jsin`, or `atan2` and **Approximation method** as `CORDIC`.

Set Custom Latency Value For a Block

To set custom latency value for a subsystem from the HDL Block Properties dialog box:

1. In the Simulink toolstrip, on the **Apps** tab, select **HDL Coder**.
2. Select the block, and on the **HDL Code** tab, click the **HDL Block Properties** button.
3. In the **General** tab, set **LatencyStrategy** to **Custom** and specify a value for **CustomLatency**.

To specify the latency strategy for a block from the command line, use `hdlset_param`. For example, to specify a custom latency of four for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'LatencyStrategy', 'Custom')
hdlset_param('my_design/my_dut/Product', 'CustomLatency', 4)
```

See also `hdlset_param`.

NFPCustomLatency

You can specify a custom latency for certain blocks in the native floating-point mode. By using the custom latency strategy, you can trade-off between clock frequency and power consumption. To specify a custom latency strategy, set **LatencyStrategy** to `Custom` and specify a value for **NFPCustomLatency**. You can specify the custom latency value from 1 to `Max` latency of the block. For more information on latency values of floating-point operators, see “Latency Values of Floating-Point Operators” on page 14-99.

You can specify the **NFPCustomLatency** setting for these blocks with both `single` and `double` data types.

- Add
- Subtract
- Product
- Math Function in Reciprocal mode
- Gain
- Divide
- Relational Operator
- Data Type Conversion
- Rounding Function

You can also specify a **NFPCustomLatency** setting for these blocks with `single` data types.

- Sqrt
- Reciprocal Sqrt
- Sum of Elements

- Product of Elements

Set Custom Latency Value For a Block

To set custom latency value for a subsystem from the HDL Block Properties dialog box:

1. Right-click the block.
2. Select **HDL Code > HDL Block Properties**.
3. In the **Native Floating Point** tab, for **LatencyStrategy**, select **Custom**.
4. Specify a value for the **NFPCustomLatency**.

To specify the latency strategy for a block from the command line, use `hdlset_param`. For example, to specify a custom latency of four for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'LatencyStategy', 'Custom')
hdlset_param('my_design/my_dut/Product', 'NFPCustomLatency', 4)
```

See also `hdlset_param`.

MantissaMultiplyStrategy

You can use the `MantissaMultiplyStrategy` property for multipliers that support HDL code generation in native floating-point mode. Blocks that have this setting include Product, Divide, Math Function (in Reciprocal mode), and so on. By using this setting, you can specify how you want HDL Coder to implement the mantissa multiplication operation for the blocks.

MantissaMultiplyStrategy Setting	Description
`'inherit'` (default)	Use the mantissa multiply strategy setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the mantissa multiply strategy setting for the model.
`'FullMultiplier'`	HDL Coder uses multipliers to perform the mantissa multiplication operation for the native floating point operator. The multipliers can utilize DSP units on the target device.
`'PartMultiplierPartAddShift'`	HDL Coder splits the implementation into two parts. One part is implemented with multipliers. The other part is implemented with a combination of adders and shifters. The multipliers can utilize the DSP units on the target device. The combination of adders and shifters does not utilize the DSP.
`'NoMultiplierFullAddShift'`	HDL Coder uses adders and shifters to implement the mantissa multiplication. This option does not utilize DSP units on the target device. You can also use this option if your target device does not contain DSP units.

To implement the mantissa multiplication with adders and shifters, set `MantissaMultiplyStrategy`, to `'NoMultiplierFullAddShift'` for that block.

Set Mantissa Multiply Strategy For a Block

To set adaptive pipelining for a subsystem from the HDL Block Properties dialog box:

1 Right-click the block.
2 Select **HDL Code** > **HDL Block Properties**.
3 In the **Native Floating Point** tab, for **MantissaMultiplyStrategy**, select **inherit**, **FullMultiplier**, **PartMultiplierPartAddShift**, or **NoMultiplierFullAddShift**.

To specify the mantissa multiply strategy for a block from the command line, use `hdlset_param`. For example, to implement the mantissa multiplication using adders and shifters for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', ...
                'MantissaMultiplyStrategy', 'PartMultiplierPartAddShift')
```

See also `hdlset_param`.

MaxIterations

You can use the `MaxIterations` property for the mod and rem functions of the Math Function block in `Native Floating Point` mode. If you have numbers `a` and `b` that are significantly large integers, you can increase the **MaxIterations** setting to match the Simulink simulation result. However, computing this result uses additional resources and increases the area footprint on the target FPGA device.

MaxIterations Setting	Description
32 (default)	The default number of iterations to compute the result of mod and rem functions in `Native Floating Point` mode. This implementation can potentially result in a numerical mismatch with the Simulink simulation results for large integers.
64	Specify `64` as the number of iterations to compute the result of mod and rem functions in `Native Floating Point` mode. In this mode, the implementation has higher probability of matching the Simulink simulation result for significantly large integers but can use more hardware resources.
128	Specify `128` as the number of iterations to compute the result of mod and rem functions in `Native Floating Point` mode. In this mode, the implementation matches the Simulink simulation result for large integers but uses more hardware resources.

Set MaxIterations For the Math Function Block

To set **MaxIterations** for a block from the HDL Block Properties dialog box:

1. Right-click the block.
2. Select **HDL Code** > **HDL Block Properties**.
3. In the **Native Floating Point** tab, for **MaxIterations**, select **32**, **64**, or **128**.

To set handle denormals for a block from the command line, use `hdlset_param`. For example, to enable adaptive pipelining for a Product block inside a subsystem, `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'HandleDenormals', 'on')
```

See also `hdlset_param`.

See Also

Related Examples

- "Floating Point Support: Field-Oriented Control Algorithm" on page 14-118

More About

- "Getting Started with HDL Coder Native Floating-Point Support" on page 14-88

HDL Filter Block Properties

In this section...

"AdderTreePipeline" on page 19-39
"AddPipelineRegisters" on page 19-39
"ChannelSharing" on page 19-40
"CoeffMultipliers" on page 19-40
"DALUTPartition" on page 19-40
"DARadix" on page 19-41
"FoldingFactor" on page 19-42
"MultiplierInputPipeline" on page 19-42
"MultiplierOutputPipeline" on page 19-42
"NumMultipliers" on page 19-43
"ReuseAccum" on page 19-43
"SerialPartition" on page 19-43

AdderTreePipeline

This property applies to frame-based filters. It specifies how many pipeline registers the architecture includes between levels of the adder tree. These pipeline stages increase filter throughput while adding latency. The default value is `0`. To improve the speed of this architecture, the recommended setting is `2`.

Pipeline stages introduce delays along the path in the model that contains the affected filter. When you enable this pipeline option, the coder automatically adds balancing delays on parallel data paths.

For more information on the frame-based filter architecture, see "Frame-Based Architecture" on page 19-47.

AddPipelineRegisters

This property applies to scalar input filters. When you enable this property, the default linear adder of the filter is implemented as a pipelined tree adder instead. This architecture increases filter throughput while adding latency. The default value is `off`.

The following limitations apply to `AddPipelineRegisters`:

- If you use `AddPipelineRegisters`, the code generator forces full precision in the HDL and the generated filter model. This option implements a pipelined adder tree structure in the HDL code for which only full precision is supported. If you generate a validation model, you must use full precision in the original model to avoid validation mismatches.
- Pipeline stages introduce delays along the path in the model that contains the affected filter. When you enable this pipeline option, the coder automatically adds balancing delays on parallel data paths.

Note When you use this property with the CIC Interpolation block, delays in parallel paths are not automatically balanced. Manually add delays where required by your design.

For filter architecture diagrams that indicate where the pipeline stages are added, see “HDL Filter Architectures” on page 19-45.

ChannelSharing

You can use the `ChannelSharing` implementation parameter with a multichannel filter to enable sharing a single filter implementation among channels for a more area-efficient design. This parameter is either `'on'` or `'off'`. The default is `'off'`, and a separate filter will be implemented for each channel.

See “Multichannel FIR Filter for FPGA” (DSP System Toolbox).

CoeffMultipliers

The `CoeffMultipliers` implementation parameter lets you specify use of canonical signed digit (CSD) or factored CSD optimizations for processing coefficient multiplier operations in code generated for certain filter blocks. Specify the `CoeffMultipliers` parameter using one of the following options:

- `'csd'`: Use CSD techniques to replace multiplier operations with shift-and-add operations. CSD techniques minimize the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits. This representation decreases the area used by the filter while maintaining or increasing clock speed.
- `'factored-csd'`: Use factored CSD techniques, which replace multiplier operations with shift-and-add operations on prime factors of the coefficients. This option lets you achieve a greater filter area reduction than CSD, at the cost of decreasing clock speed.
- `'multipliers'` (default): Retain multiplier operations.

HDL Coder supports `CoeffMultipliers` for fully-parallel filter implementations. It is not supported for fully-serial and partly-serial architectures.

DALUTPartition

The size of the LUT grows exponentially with the order of the filter. For a filter with `N` coefficients, the LUT must have `2^N` values. For higher order filters, LUT size must be reduced to reasonable levels. To reduce the size, you can subdivide the LUT into a number of LUTs, called LUT partitions. Each LUT partition operates on a different set of taps. The results obtained from the partitions are summed.

For example, for a 160-tap filter, the LUT size is `(2^160)*W` bits, where `W` is the word size of the LUT data. Dividing this into 16 LUT partitions, each taking 10 inputs (taps), the total LUT size is reduced to `16*(2^10)*W` bits.

Although LUT partitioning reduces LUT size, more adders are required to sum the LUT data.

You can use `DALUTPartition` to enables DA code generation and specify the number and size of LUT partitions.

Specify LUT partitions as a vector of integers `[p1 p2...pN]` where:

- `N` is the number of partitions.
- Each vector element specifies the size of a partition. The maximum size for an individual partition is 12.

- The sum of all vector elements equals the filter length `FL`. `FL` is calculated differently depending on the filter type. You can find how FL is calculated for different filter types in the next section.

See "Distributed Arithmetic for HDL Filters" on page 19-50.

Specifying DALUTPartition for Single-Rate Filters

To determine the LUT partition for one of the supported single-rate filter types, calculate `FL` as shown in the following table. Then, specify the partition as a vector whose elements sum to `FL`.

Filter Type	Filter Length (FL) Calculation
Direct-form FIR	`FL = length(find(Hd.numerator ~= 0))`
Direct-form asymmetrical FIR, direct-form symmetrical FIR	`FL = ceil(length(find(Hd.numerator ~= 0))/2)`

You can also specify generation of DA code for your filter design without LUT partitioning. To do so, specify a vector of one element, whose value is equal to the filter length.

Specifying DALUTPartition for Multirate Filters

For supported multirate filters (FIR Decimation and FIR Interpolation), you can specify the LUT partition as

- A vector defining a partition for LUTs for all polyphase subfilters.
- A matrix of LUT partitions, where each row vector specifies a LUT partition for a corresponding polyphase subfilter. In this case, the `FL` is uniform for all subfilters. This approach provides fine control for partitioning each subfilter.

The following table shows the `FL` calculations for each type of LUT partition.

LUT Partition	Filter Length (FL) Calculation
Vector: Determine `FL` as shown in the **Filter Length (FL) Calculation** column to the right. Specify the LUT partition as a vector of integers whose elements sum to `FL`.	`FL = size(polyphase(Hm), 2)`

Matrix: Determine the subfilter length `FL`*i* based on the polyphase decomposition of the filter, as shown in the **Filter Length (FL) Calculation** column to the right. Specify the LUT partition for each subfilter as a row vector whose elements sum to `FL`*i*.	`p = polyphase(Hm);`
	`FLi = length(find(p(i,:)));`
	
	where *i* is the index to the *i*th row of the polyphase matrix of the multirate filter. The *i*th row of the matrix `p` represents the *i*th subfilter.

DARadix

The inherently bit-serial nature of DA can limit throughput. To improve throughput, the basic DA algorithm can be modified to compute more than one bit sum at a time. The number of simultaneously computed bit sums is expressed as a power of two called the DA radix. For example, a DA radix of 2 (`2^1`) indicates that one bit sum is computed at a time. A DA radix of 4 (`2^2`) indicates that two bit sums are computed at a time, and so on.

To compute more than one bit sum at a time, the LUT is replicated. For example, to perform DA on 2 bits at a time (radix 4), the odd bits are fed to one LUT and the even bits are simultaneously fed to an

identical LUT. The LUT results corresponding to odd bits are left-shifted before they are added to the LUT results corresponding to even bits. This result is then fed into a scaling accumulator that shifts its feedback value by 2 places.

Processing more than one bit at a time introduces a degree of parallelism into the operation, improving speed at the expense of area.

You can use `DARadix` to specify the number of bits processed simultaneously in DA. The number of bits is expressed as `N`, which must be:

- A nonzero positive integer that is a power of two
- Such that `mod(W, log2(N)) = 0`, where `W` is the input word size of the filter

The default value for `N` is 2, specifying processing of one bit at a time, or fully serial DA, which is slow but low in area. The maximum value for `N` is `2^W`, where `W` is the input word size of the filter. This maximum specifies fully parallel DA, which is fast but high in area. Values of `N` between these extrema specify partly serial DA.

Note When setting a `DARadix` value for symmetrical and asymmetrical filters, see “Considerations for Symmetrical and Asymmetrical Filters” on page 19-51.

See “Distributed Arithmetic for HDL Filters” on page 19-50.

FoldingFactor

`FoldingFactor` specifies the total number of clock cycles taken for the computation of filter output in an IIR SOS filter with serial architecture. It is complementary with “NumMultipliers” on page 19-43. You must select one property or the other; you cannot use both. If you do not specify either `FoldingFactor` or `NumMultipliers`, HDL code for the filter is generated with fully parallel architecture.

MultiplierInputPipeline

You can use this parameter to generate a specified number of pipeline stages at multiplier inputs for FIR filter structures. The default value is 0.

The following limitation applies to `MultiplierInputPipeline`:

- Pipeline stages introduce delays along the path in the model that contains the affected filter. When you enable this pipeline option, the coder automatically adds balancing delays on parallel data paths.

For diagrams of where these pipeline stages occur in the filter architecture, see “HDL Filter Architectures” on page 19-45.

MultiplierOutputPipeline

You can use this parameter to generate a specified number of pipeline stages at multiplier outputs for FIR filter structures. The default value is 0.

The following limitation applies to `MultiplierOutputPipeline`:

- Pipeline stages introduce delays along the path in the model that contains the affected filter. When you enable this pipeline option, the coder automatically adds balancing delays on parallel data paths.

For diagrams of where these pipeline stages occur in the filter architecture, see “HDL Filter Architectures” on page 19-45.

NumMultipliers

`NumMultipliers` specifies the total number of multipliers used for the filter implementation in an IIR SOS filter with serial architecture. It is complementary with “FoldingFactor” on page 19-42 property. You must select one property or the other; you cannot use both. If you do not specify either `FoldingFactor` or `NumMultipliers`, HDL code for the filter is generated with fully parallel architecture.

ReuseAccum

You can use this parameter to enable or disable accumulator reuse in a serial HDL architecture. The default is a fully parallel architecture.

To Generate This Architecture...	Set ReuseAccum to...
Fully parallel	Omit this property
Fully serial	Not specified, or `'off'`
Partly serial	`'off'`
Cascade-serial with explicitly specified partitioning	`'on'`
Cascade-serial with automatically optimized partitioning	`'on'`

For more information on parallel and serial filter architectures, see “HDL Filter Architectures” on page 19-45

SerialPartition

Use this parameter to specify partitions for a serial filter architecture. The default is a fully parallel architecture.

To Generate This Architecture...	Set SerialPartition to...
Fully parallel	Omit this property
Fully serial	`N`, where `N` is the length of the filter

To Generate This Architecture...	Set SerialPartition to...
Partly serial	[p1 p2 p3...pN]: A vector of integers having N elements, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter. When you define the partitioning for a partly serial architecture, consider the following: • The filter length should be divided as uniformly as possible into a vector equal in length to the number of multipliers intended. For example, if your design requires a filter of length 9 with 2 multipliers, the recommended partition is [5 4]. If your design requires 3 multipliers, the recommended partition is[3 3 3] rather than some less uniform division such as [1 4 4] or [3 4 2]. • If your design is constrained by the need to compute each output value (corresponding to each input value) in an exact number N of clock cycles, use N as the largest partition size and partition the other elements as uniformly as possible. For example, if the filter length is 9 and your design requires exactly 4 cycles to compute the output, define the partition as [4 3 2]. This partition executes in 4 clock cycles, at the cost of 3 multipliers.
Cascade-serial with explicitly specified partitioning	[p1 p2 p3...pN]: A vector of N integers, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter. The values of the vector elements must be in descending order, except the last two elements, which can be equal. For example, for a filter length of 8, partitions [5 3] or [4 2 2] are valid, but the partitions [2 2 2 2] and [3 2 3] raise an error at code generation time.
Cascade-serial with automatically optimized partitioning	Omit this property.

For more information on parallel and serial filter architectures, see “HDL Filter Architectures” on page 19-45.

This property is also used for Min/Max blocks with cascade-serial architectures. For how to configure Min/Max cascades, see “SerialPartition” on page 19-23.

See Also

More About

- “Set and View HDL Model and Block Parameters” on page 19-52
- “HDL Block Properties: General” on page 19-3

HDL Filter Architectures

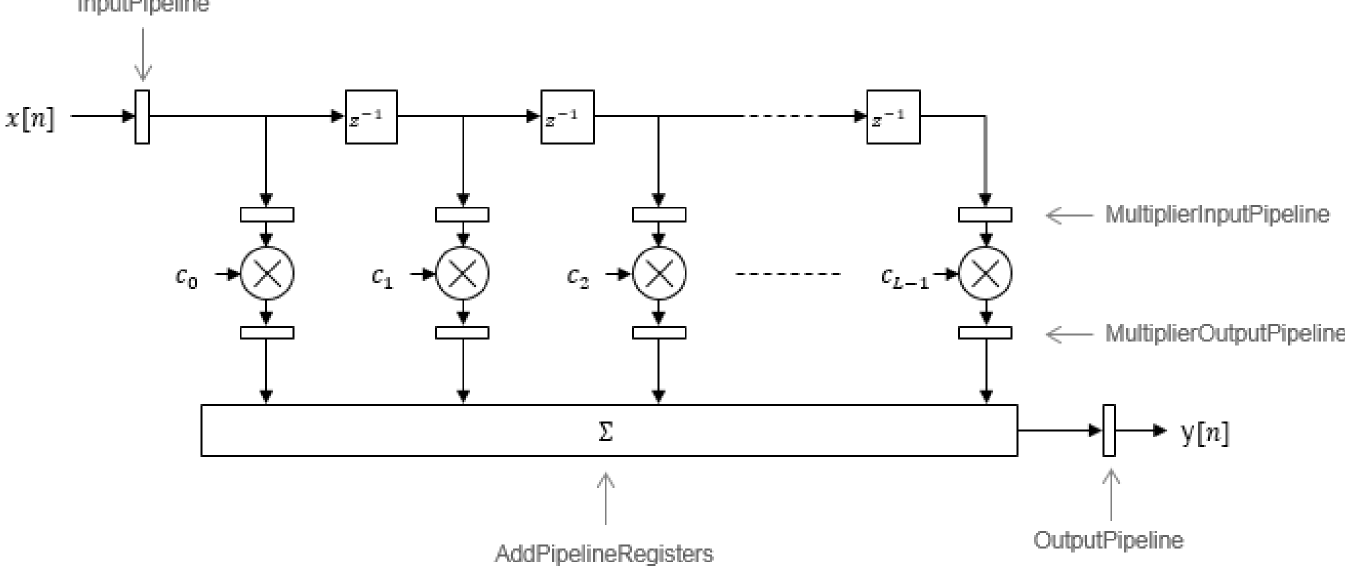
The HDL Coder software provides architecture options that extend your control over speed vs. area tradeoffs in the realization of filter designs. To achieve the desired tradeoff for generated HDL code, you can either specify a fully parallel architecture, or choose one of several serial architectures. Configure a serial architecture using the “SerialPartition” on page 19-43 and “ReuseAccum” on page 19-43 parameters. You can also choose a frame-based filter for increased throughput.

Use pipelining parameters to improve speed performance of your filter designs. Add pipelines to the adder logic of your filter using AddPipelineRegisters on page 19-39 for scalar input filters, and “AdderTreePipeline” on page 19-39 for frame-based filters. Specify pipeline stages before and after each multiplier with MultiplierInputPipeline on page 19-42 and MultiplierOutputPipeline on page 19-42. Set the number of pipeline stages before and after the filter using “InputPipeline” on page 19-14 and “OutputPipeline” on page 19-19. The architecture diagrams show the locations of the various configurable pipeline stages.

Fully Parallel Architecture

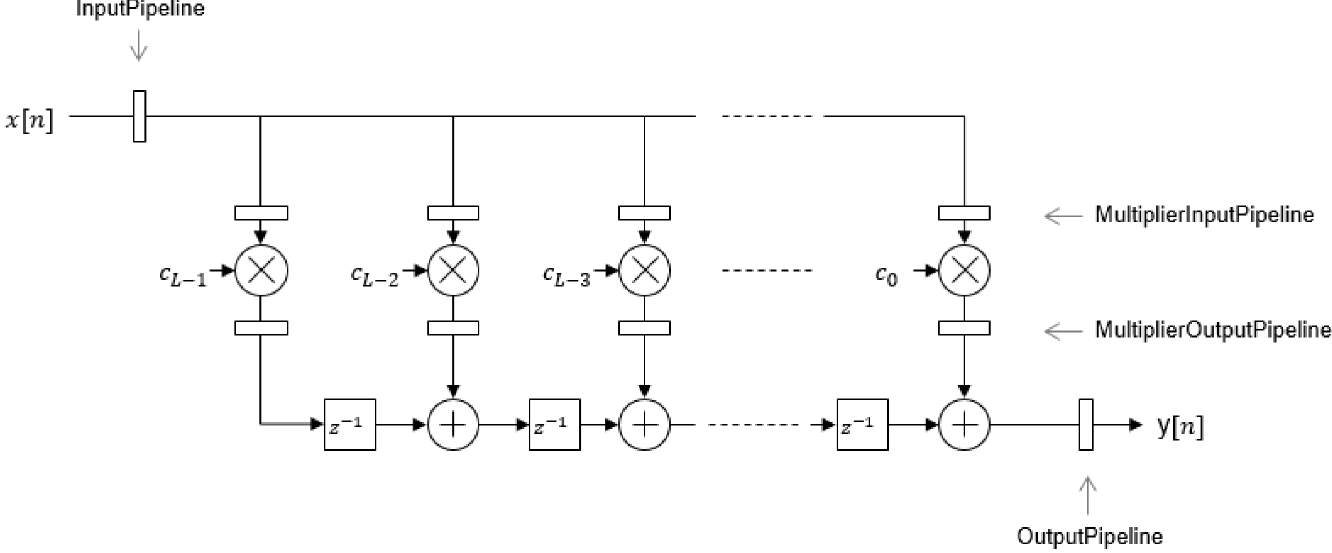
This option is the default architecture. A fully parallel architecture uses a dedicated multiplier and adder for each filter tap. The taps execute in parallel. A fully parallel architecture is optimal for speed. However, it requires more multipliers and adders than a serial architecture, and therefore consumes more chip area. The diagrams show the architectures for direct form and for transposed filter structures with fully parallel implementations, and the location of configurable pipeline stages.

Direct Form



By default, the block implements linear adder logic. When you enable `AddPipelineRegisters`, the adder logic is implemented as a pipelined adder tree. The adder tree uses full-precision data types. If you generate a validation model, you must use full precision in the original model to avoid validation mismatches.

Transposed



The `AddPipelineRegisters` parameter has no effect on a transposed filter implementation.

Serial Architectures

Serial architectures reuse hardware resources in time, saving chip area. Configure a serial architecture using the “SerialPartition” on page 19-43 and “ReuseAccum” on page 19-43 parameters. The available serial architecture options are *fully serial*, *partly serial*, and *cascade serial*.

Fully Serial

A fully serial architecture conserves area by reusing multiplier and adder resources sequentially. For example, a four-tap filter design uses a single multiplier and adder, executing a multiply-accumulate operation once for each tap. The multiply-accumulate section of the design runs at four times the filter's input/output sample rate. This design saves area at the cost of some speed loss and higher power consumption.

In a fully serial architecture, the system clock runs at a much higher rate than the sample rate of the filter. Thus, for a given filter design, the maximum speed achievable by a fully serial architecture is less than that of a parallel architecture.

Partly Serial

Partly serial architectures cover the full range of speed vs. area tradeoffs that lie between fully parallel and fully serial architectures.

In a partly serial architecture, the filter taps are grouped into a number of serial partitions. The taps within each partition execute serially, but the partitions execute in parallel with respect to one another. The outputs of the partitions are summed at the final output.

When you select a partly serial architecture, you specify the number of partitions and the length (number of taps) of each partition. Suppose you specify a four-tap filter with two partitions, each having two taps. The system clock runs at twice the filter's sample rate.

Cascade Serial

A cascade-serial architecture closely resembles a partly serial architecture. As in a partly serial architecture, the filter taps are grouped into a number of serial partitions that execute in parallel with respect to one another. However, the accumulated output of each partition is cascaded to the accumulator of the previous partition. The output of all partitions is therefore computed at the accumulator of the first partition. This technique is termed *accumulator reuse*. A final adder is not required, which saves area.

The cascade-serial architecture requires an extra cycle of the system clock to complete the final summation to the output. Therefore, the frequency of the system clock must be increased slightly with respect to the clock used in a noncascade partly serial architecture.

To generate a cascade-serial architecture, specify a partly serial architecture with accumulator reuse enabled. If you do not specify the serial partitions, HDL Coder automatically selects an optimal partitioning.

Latency in Serial Architectures

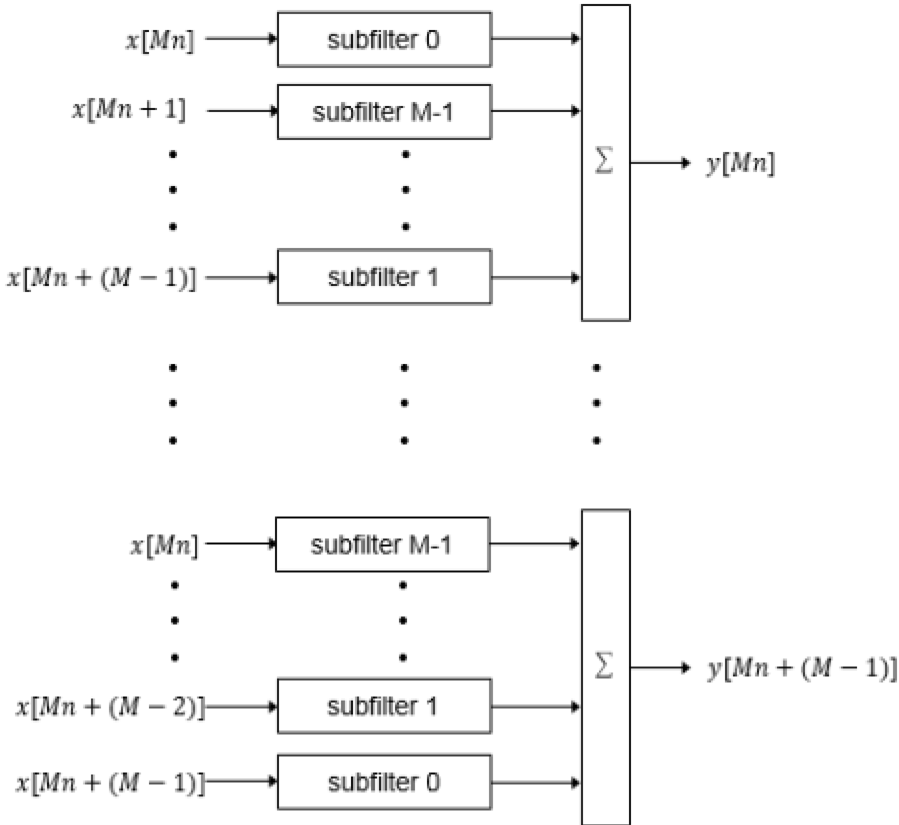
Serialization of a filter increases the total latency of the design by one clock cycle. The serial architectures use an accumulator (an adder with a register) to add the products sequentially. An additional final register is used to store the summed result of all the serial partitions, requiring an extra clock cycle for the operation. To model this latency, HDL Coder inserts a Delay block into the generated model after the filter block.

Full-Precision for Serial Architectures

When you choose a serial architecture, the code generator uses full precision in the HDL code. HDL Coder therefore forces full precision in the generated model. If you generate a validation model, you must use full precision in the original model to avoid validation mismatches.

Frame-Based Architecture

When you select a frame-based architecture and provide an *M*-sample input frame, the coder implements a fully parallel filter architecture. The filter includes *M* parallel subfilters for each input sample.

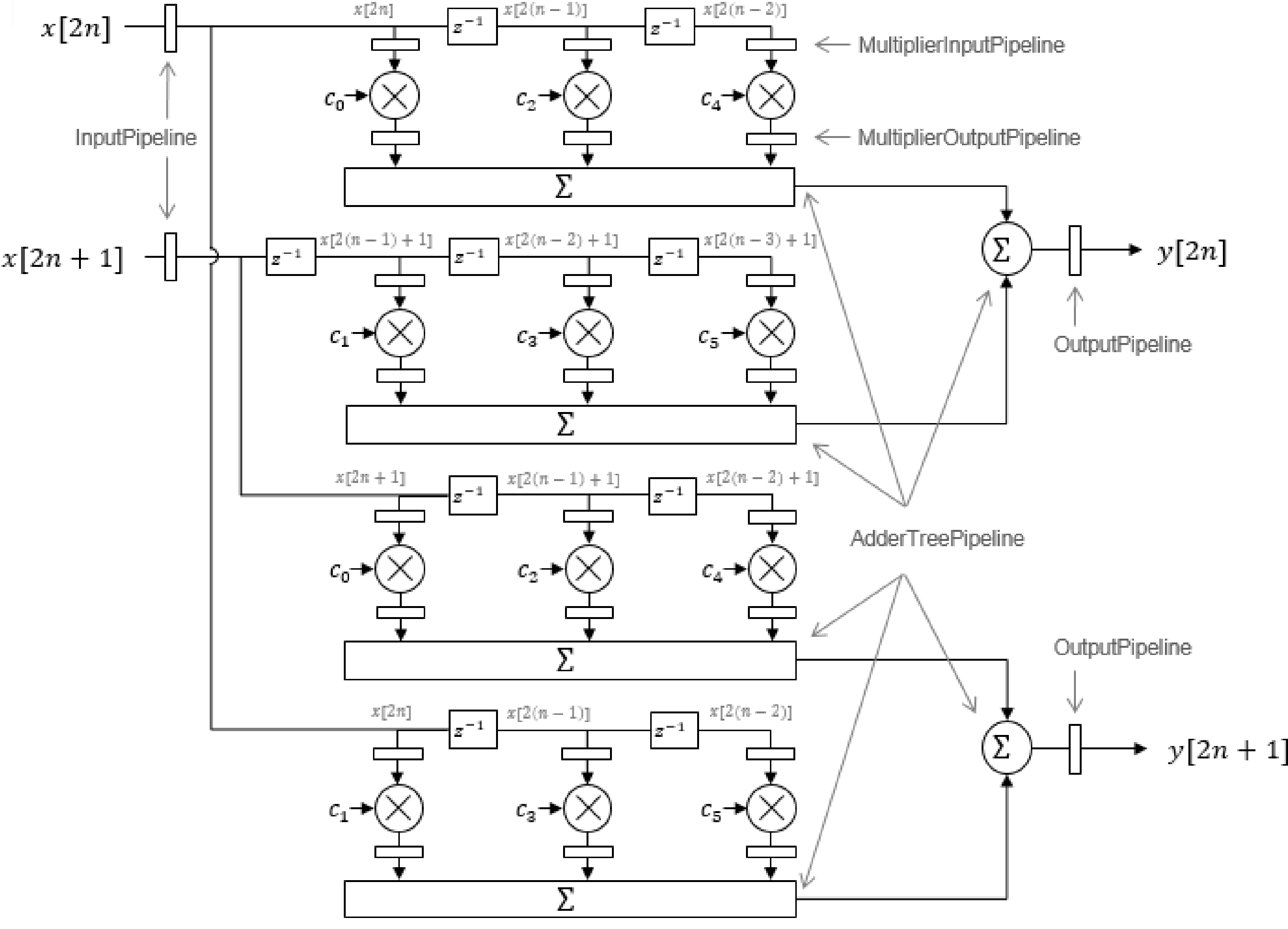


Each of the subfilters includes every *M*th coefficient. The subfilter results are added so that each output sample is the sum of each of the coefficients multiplied with one input sample.

Subfilter	Coefficients
0	c_0, c_M, …
1	c_a, c_{M+1}, …
M–1	c_{M-1}, c_{2M-1}, …

The diagram shows the filter architecture for a frame size of two samples (*M* = 2), and a filter length of six coefficients. The input is a vector with two values representing samples in time. The input samples, *x[2n]* and *x[2n+1]*, represent the *n*th input pair. Every second sample from each stream is fed to two parallel subfilters. The four subfilter results are added together to create two output samples. In this way, each output sample is the sum of each of the coefficients multiplied with one of the input samples.

The sums are implemented as a pipelined adder tree. Set “AdderTreePipeline” on page 19-39 to specify the number of pipeline stages between levels of the adder tree. To improve clock speed, it is recommended that you set this parameter to 2. To fit the multipliers into DSP blocks on your FPGA, add pipeline stages before and after the multipliers using MultiplierInputPipeline on page 19-42 and MultiplierOutputPipeline on page 19-42.



For symmetric or antisymmetric coefficients, the filter architecture reuses the coefficient multipliers and adds design delay between the multiplier and summation stages as required.

See Also

More About

- "HDL Filter Block Properties" on page 19-39
- "Distributed Arithmetic for HDL Filters" on page 19-50

Distributed Arithmetic for HDL Filters

Distributed Arithmetic (DA) is a widely used technique for implementing sum-of-products computations without the use of multipliers. Designers frequently use DA to build efficient Multiply-Accumulate Circuitry (MAC) for filters and other DSP applications. The main advantage of DA is its high computational efficiency. DA distributes multiply and accumulate operations across shifters, lookup tables (LUTs) and adders in such a way that conventional multipliers are not required.

In a DA realization of a FIR filter structure, a sequence of input data words of width `W` is fed through a parallel to serial shift register, producing a serialized stream of bits. The serialized data is then fed to a bit-wise shift register. This shift register serves as a delay line, storing the bit serial data samples.

The delay line is tapped (based on the input word size `W`), to form a `W`-bit address that indexes into a lookup table (LUT). The LUT stores all possible sums of partial products over the filter coefficients space. The LUT is followed by a shift and adder (scaling accumulator) that adds the values obtained from the LUT sequentially.

A table lookup is performed sequentially for each bit (in order of significance starting from the LSB). On each clock cycle, the LUT result is added to the accumulated and shifted result from the previous cycle. For the last bit (MSB), the table lookup result is subtracted, accounting for the sign of the operand.

This basic form of DA is fully serial, operating on one bit at a time. If the input data sequence is `W` bits wide, then a FIR structure takes `W` clock cycles to compute the output. Symmetric and asymmetric FIR structures are an exception, requiring `W+1` cycles, because one additional clock cycle is needed to process the carry bit of the preadders.

You can control how DA code is generated by using the `DALUTPartition` and `DARadix` implementation parameters. The `DALUTPartition` and `DARadix` parameters have certain requirements and restrictions that are specific to different filter types. These requirements are included in the discussions of each parameter.

- Reduce LUT Size: “DALUTPartition” on page 19-40
- Improve Performance with Parallelism: “DARadix” on page 19-41

For information on the theoretical foundations of DA, see “Further References” on page 19-51.

Requirements and Considerations for Generating Distributed Arithmetic Code

Fixed-Point Quantization Required

Generation of DA code is supported only for fixed-point filter designs.

Specifying Filter Precision

The data path in HDL code generated for the DA architecture is carefully optimized for full precision computations. The filter result is cast to the output data size only at the final stage when it is presented to the output.

Distributed arithmetic merges the product and accumulator operations and does computations at full precision. This approach ignores the **Product output** and **Accumulator** properties of the Digital Filter block and sets these properties to full precision.

Coefficients with Zero Values

DA ignores taps that have zero-valued coefficients and reduces the size of the DA LUT accordingly.

Considerations for Symmetrical and Asymmetrical Filters

For symmetrical and asymmetrical filters:

- A bit-level preadder or presubtractor is required to add tap data values that have coefficients of equal value and/or opposite sign. One extra clock cycle is required to compute the result because of the additional carry bit.
- HDL Coder takes advantage of filter symmetry where possible. This reduces the DA LUT size substantially, because the effective filter length for these filter types is halved.

Further References

Detailed discussions of the theoretical foundations of DA appear in the following publications:

- Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, Second Edition, Springer, pp 88–94, 128–143
- White, S.A., *Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review*. IEEE ASSP Magazine, Vol. 6, No. 3

Set and View HDL Model and Block Parameters

In this section...

"Set HDL Block Parameters" on page 19-52

"Set HDL Block Parameters for Multiple Blocks Programmatically" on page 19-52

"View All HDL Block Parameters" on page 19-54

"View Non-Default HDL Block Parameters" on page 19-54

"View HDL Model Parameters" on page 19-54

You can view and set HDL-related block properties, such as implementation and implementation parameters, at the model level and at the individual block level.

Set HDL Block Parameters

To set the HDL Block parameters from the UI, open the HDL Block Properties dialog box, and modify the block properties. To open the HDL Properties dialog box, either:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the block for which you want to see the HDL parameters and then select **HDL Block Properties**.
- Right-click the block and select **HDL Code > HDL Block Properties**.

To set the HDL-related parameters at the command line, use `hdlset_param`. `hdlset_param(`*`path`*`,`*`Name`*`, `*`Value`*`)` sets HDL-related parameters in the block or model referenced by *`path`*. One or more *`Name,Value`* pair arguments specify the parameters to be set, and their values. You can specify several name and value pair arguments in any order as *`Name1,Value1,…,NameN,ValueN`*.

For example, to set the sharing factor to 2 and the architecture to `Tree` for a block in your model:

1. Open the model and select the block.
2. Enter the following at the command line:

   ```
   hdlset_param (gcb, 'SharingFactor', 2, 'Architecture', 'Tree')
   ```

To view the HDL parameters specified for a block, use `hdlget_param`. For example, to see the HDL architecture setting for a block, at the command line, enter:

```
hdlget_param(gcb, 'Architecture')
```

You can also assign the returned HDL block parameters to a cell array. In the following example, `hdlget_param` returns all HDL block parameters and values to the cell array `p`.

```
p = hdlget_param(gcb, 'all')


p =

    'Architecture'   'Linear'   'InputPipeline'   [0]   'OutputPipeline'   [0]
```

Set HDL Block Parameters for Multiple Blocks Programmatically

For models that contain a large number of blocks, using the **HDL Block Properties** dialog box to select block implementations or set implementation parameters for individual blocks may not be

practical. It is more efficient to set HDL-related model or block parameters for multiple blocks programmatically. You can use the `find_system` function to locate the blocks of interest. Then, use a loop to call `hdlset_param` to set the desired parameters for each block.

The following example uses the `sfir_fixed` model to demonstrate how to locate a group of blocks in a subsystem and specify the same output pipeline depth for all the blocks.

```
open_system('sfir_fixed')

% Find all Product blocks in the model
prodblocks = find_system('sfir_fixed/symmetric_fir', ...
                            'BlockType', 'Product')

% Set the output pipeline to 2 for the blocks
for ii=1:length(prodblocks)
    hdlset_param(prodblocks{ii}, 'OutputPipeline', 2)
end
```

```
prodblocks =

  4×1 cell array

    {'sfir_fixed/symmetric_fir/m1'}
    {'sfir_fixed/symmetric_fir/m2'}
    {'sfir_fixed/symmetric_fir/m3'}
    {'sfir_fixed/symmetric_fir/m4'}
```

To verify the settings, use `hdlget_param` to display the value of the `OutputPipeline` parameter for the blocks.

```
% Get the output pipeline to 2 for the blocks
for ii=1:length(prodblocks)
    hdlget_param(prodblocks{ii}, 'OutputPipeline')
end
```

```
ans =

     2


ans =

     2


ans =

     2


ans =

     2
```

View All HDL Block Parameters

`hdldispblkparams` displays the HDL block parameters available for a specified block.

The following example displays HDL block parameters and values for the currently selected block.

```
hdldispblkparams(gcb,'all')
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL Block Parameters ('simplevectorsum/vsum/Sum of
Elements')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Implementation

    Architecture  : Linear

Implementation Parameters

    InputPipeline : 0
    OutputPipeline : 0
```

See also `hdldispblkparams`.

View Non-Default HDL Block Parameters

The following example displays only HDL block parameters that have non-default values for the currently selected block.

```
hdldispblkparams(gcb)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL Block Parameters ('simplevectorsum/vsum/Sum of
Elements')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Implementation

    Architecture : Linear

Implementation Parameters

    OutputPipeline : 3
```

View HDL Model Parameters

To display the names and values of HDL-related properties in a model, use the `hdldispmdlparams` function.

The following example displays HDL-related properties and values of the current model, in alphabetical order by property name.

```
hdldispmdlparams(bdroot,'all')
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
HDL CodeGen Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%

AddPipelineRegisters                  : 'off'
Backannotation                        : 'on'
BlockGenerateLabel                    : '_gen'
CheckHDL                              : 'off'
ClockEnableInputPort                  : 'clk_enable'
.
.
.
VerilogFileExtension                  : '.v'
```

The following example displays only HDL-related properties that have non-default values.

```
hdldispmdlparams(bdroot)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL CodeGen Parameters (non-default)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CodeGenerationOutput           : 'GenerateHDLCodeAndDisplayGeneratedModel'
HDLSubsystem                   : 'simplevectorsum/vsum'
ResetAssertedLevel             : 'Active-low'
Traceability                   : 'on'
```

See Also

Functions

hdlset_param | hdlget_param | hdldispblkparams | hdldispmdlparams

More About

- “HDL Block Properties: General” on page 19-3
- “HDL Block Properties: Native Floating Point” on page 19-29

Pass through and No HDL Implementations

Pass-through and No HDL Implementations

Implementation	Description
Pass-through implementations	Provides a pass-through implementation in which the block's inputs are passed directly to its outputs. HDL Coder supports the following blocks with a pass-through implementation: • Convert 1-D to 2-D • Reshape • Signal Conversion • Signal Specification
No HDL	The `NoHDL` implementation completely removes the block from the generated code. Thus, you can use the block in simulation but treat it as a “no-op” in the HDL code. This implementation is used for many blocks (such as Scopes and Assertions) that are significant in simulation but would be meaningless in HDL code. You can also use this implementation as an alternative implementation for subsystems. Logic driving a `NoHDL` subsystem might cause unconnected logic that is eliminated in the HDL code. For more information, see “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 21-224. Output ports of `NoHDL` subsystems are not recommended. `NoHDL` subsystems appear in the generated model but are removed, along with any logic that only drives the subsystem, before code generation. To generate HDL for the surrounding logic of a `NoHDL` subsystem, comment through the subsystem in Simulink instead. For more information, see “Using Comment Out and Comment Through of Blocks” on page 18-28.

For more information related to special-purpose implementations, see “External Component Interfaces”.

Build a ROM Block with Simulink Blocks

HDL Coder does not provide a ROM block, but you can easily build one using basic Simulink blocks. The Getting Started with RAM and ROM example includes a ROM built using a 1-D Lookup Table block and a Unit Delay block. To open the example, type the following command at the MATLAB prompt:

```
openExample('hdlcoder/GettingStartedWithRAMAndROMInSimulinkExample',...
'supportingFile','hdlcoderrom.slx');
```

Getting Started with RAM and ROM in Simulink

This example shows how to utilize RAM resources in your FPGA design using HDL Coder™.

Introduction

Dedicated RAM blocks on an FPGA are valuable resources for digital designs. It is easy to design with RAM and ROM in Simulink® and utilize the dedicated RAM blocks available on your FPGA using HDL Coder.

RAM Blocks in the HDL Example Library

HDL Coder provides these types of RAM blocks in the HDL RAMs block library. Use `hdllib` to display blocks that are compatible with HDL Coder™, and then select the `HDL RAMs` library under `HDL Coder`:

- Single Port RAM
- Single Port RAM System
- Simple Dual Port RAM
- Simple Dual Port RAM System
- Dual Port RAM
- Dual Port RAM System
- Dual Rate Dual Port RAM

Run this command to navigate to the RAM blocks in the HDL library.

```
hdllib
```

The RAM blocks are masked subsystems built using Simulink blocks for behavioral simulation. For code generation, HDL Coder generates predefined templates that describe RAM structures in HDL. Most synthesis tools recognize the RAM structures in the templates, and map them to available RAM resources on the FPGA.

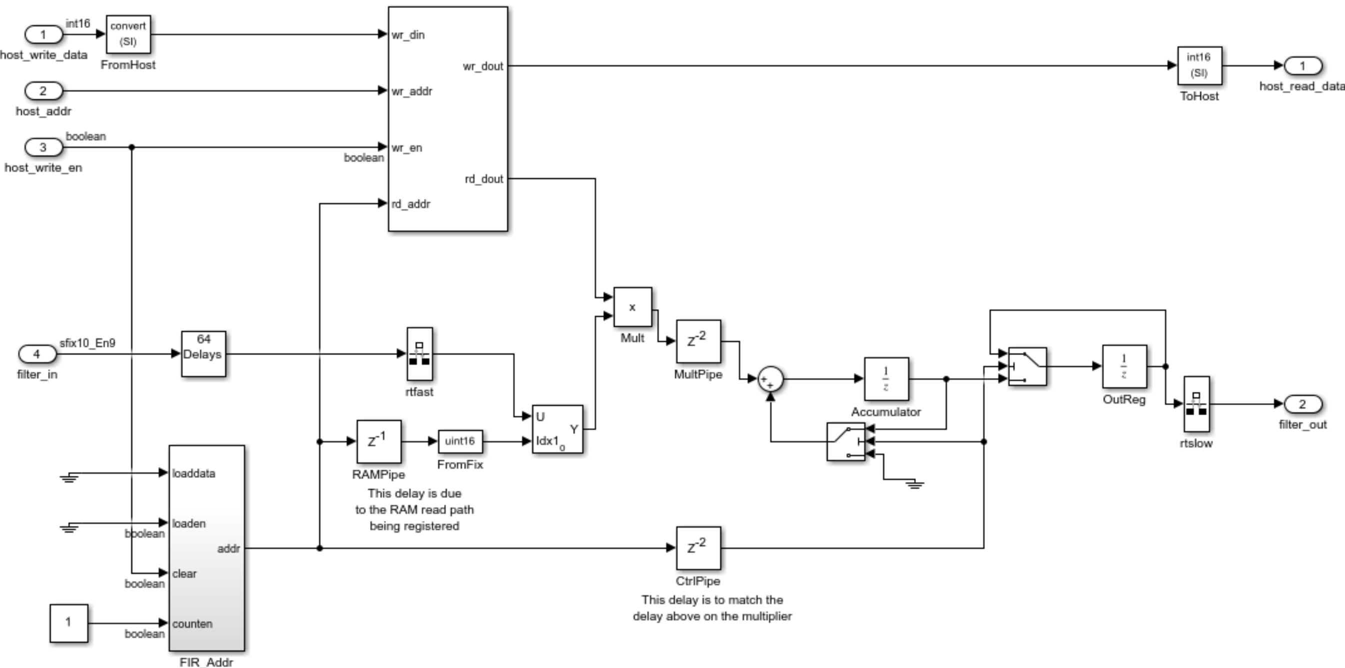
Using Generic RAM Coding Style for RAM Blocks

By default, HDL Coder provides RAM templates that use clock enable for the RAM structures. As an alternative, HDL Coder also provides a style of generic template that does not use clock enable. The generic RAM style template implements clock enable with logic in a wrapper around the RAM. You can control this using **RAM Architecture** option in the HDL Coder global settings panel.

You may want to use the generic RAM style if your synthesis tool does not support RAM structures with a clock enable and cannot map the HDL to FPGA RAM resources as a result.

The example `hdlcoderfirram` shows how to use the generic RAM style for your design.

```
open_system('hdlcoderfirram');
open_system('hdlcoderfirram/FIR_RAM');
```



Selecting a RAM Coding Style

You can select a RAM coding style from the Configuration Parameters window. In the Simulink Toolstrip, in the **Apps** tab, select **HDL Coder**. In the **HDL Code** tab, select **Settings**. In **HDL Code Generation > Global Settings > Coding style**, set the RAM Architecture parameter to your desired RAM coding style.

The **RAM Architecture** parameter controls the generation of clock enable logic for RAM in a subsystem and determines how HDL Coder implements clock enable signals when it generates HDL code for RAM.

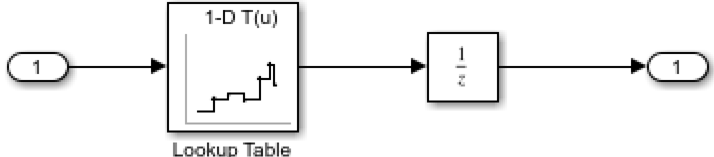
By default, **RAM Architecture** is set to `RAM with clock enable`. HDL Coder generates HDL code for RAM that uses a dedicated clock enable signal. The availability of a dedicated clock enable signal depends on your FPGA hardware specifications.

If you set **RAM Architecture** to `Generic RAM without clock enable`, HDL Coder generates HDL code for RAM that does not support a clock enable signal. Instead, HDL Coder generates a RAM wrapper that implements the clock enable logic. Use this setting if you use a synthesis tool that does not support RAM with a clock enable signal or target hardware that does not properly map the generated HDL code for RAM to the FPGA RAM resources.

Building ROM Using Simulink Blocks

HDL Coder does not provide a ROM block, but you can build one using a Lookup Table block and a Unit Delay block from Simulink, as shown in the following example.

```
open_system('hdlcoderrom');
open_system('hdlcoderrom/ROM');
```



Follow these modeling guidelines when building ROM from Simulink:

- For an n-bit address, specify all 2^n entries of the Lookup Table data. Otherwise, your synthesis tool may not map the generated code to RAM, and the code may not match your Simulink model.
- Place the Lookup Table and Unit Delay blocks in the same model hierarchy.
- Support of RAM reset logic varies among FPGA devices and synthesis tools. For best synthesis results, suppress the generation of reset logic for the Unit Delay block by setting its **ResetType** property to `none`, in the HDL Block Properties dialog box. Also set the **IgnoreDataChecking** property to 1 in the HDL Test Bench configuration parameters to ignore the initial simulation mismatch caused by suppressing the reset logic.

If you follow these guidelines, most synthesis tools will implement the ROM using dedicated RAM blocks in an FPGA.

Minimum RAM Size Requirements

If the size of the RAM or ROM in your design is small, your synthesis tool may map the generated code to registers instead of dedicated RAM resources for better speed performance. Check your synthesis tool for minimum RAM size requirements, and if desired, how you may override those requirements.

Wireless Communications Design for ASICs, FPGAs, and SoCs

In this section...

"From Mathematical Algorithm to Hardware Implementation" on page 19-61
"HDL-Optimized Blocks" on page 19-63
"Reference Applications" on page 19-63
"Generate HDL Code and Prototype on FPGA" on page 19-63

Deploying algorithmic models to ASIC, FPGA, or SoC hardware makes it possible to do over-the-air testing and verification. However, designing wireless communications systems for hardware requires design tradeoffs between hardware resources and throughput. You can speed up hardware design and deployment by using HDL-optimized blocks that have hardware-suitable interfaces and architectures, reference applications that implement portions of the LTE, 5G NR, satellite communication, WLAN, and custom OFDM-based communications physical layer, and automatic HDL code generation. You can also use hardware support packages to assist with deploying and verifying your design on real hardware.

MathWorks® HDL products, such as Wireless HDL Toolbox, allow you to start with a mathematical model, such as MATLAB code from LTE Toolbox™, 5G Toolbox™, WLAN Toolbox™, or Satellite Communications Toolbox and design a hardware implementation of that algorithm that is suitable for ASICs, FPGAs, and SoCs.

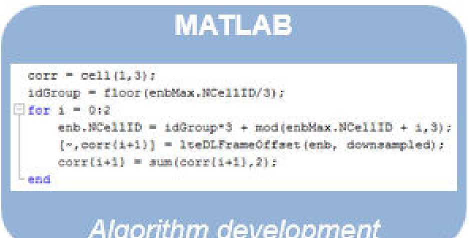
From Mathematical Algorithm to Hardware Implementation

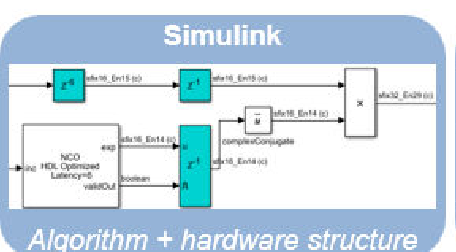
Wireless communications design starts with algorithm development and testing using MATLAB functions. MATLAB code, which usually operates on matrices of floating-point data, is good for developing mathematical algorithms, manipulating large data sets, and visualizing data.

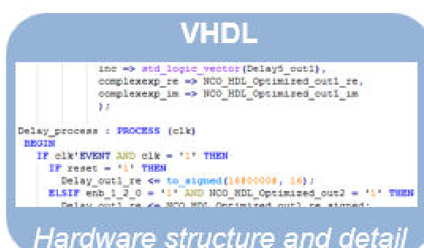
Hardware engineers typically receive a mathematical specification from an algorithm team, and reimplement the algorithm for hardware. Hardware designs require tradeoffs of resource usage for clock speed and overall throughput. This tradeoff means operating on streaming data, and designing logic to control the storage and flow of data. Hardware engineers also work in hardware description languages (HDLs), like VHDL and Verilog, that provide cycle-based modeling and parallelism.

To bridge this gap between mathematical algorithm and hardware implementation, use the MATLAB algorithm model as a starting point for hardware implementation. Make incremental changes to the design to make it suitable for hardware, and progress towards a Simulink model that you can use to automatically generate HDL code by using HDL Coder.

This diagram shows the design progression from mathematical algorithm in MATLAB, to hardware-compatible implementation in Simulink, and then the generated VHDL code.

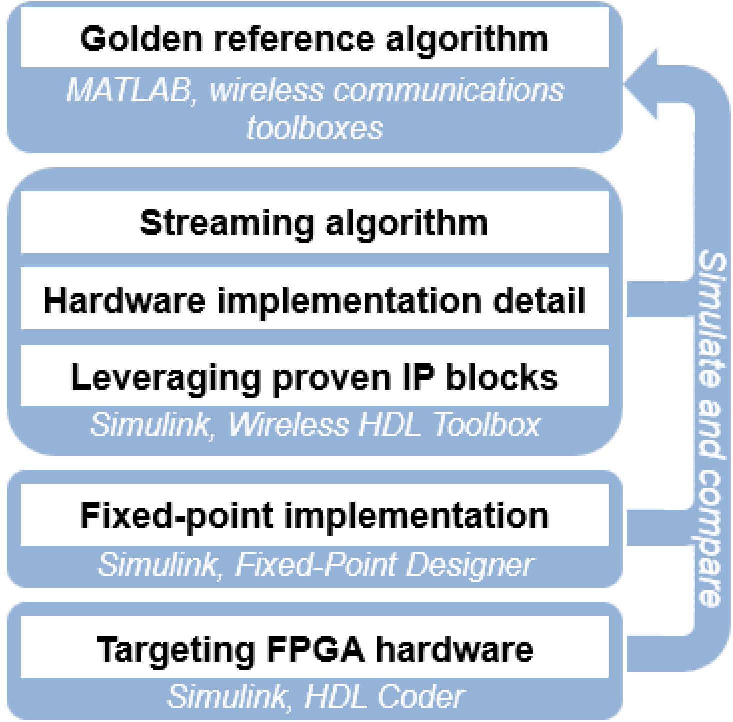




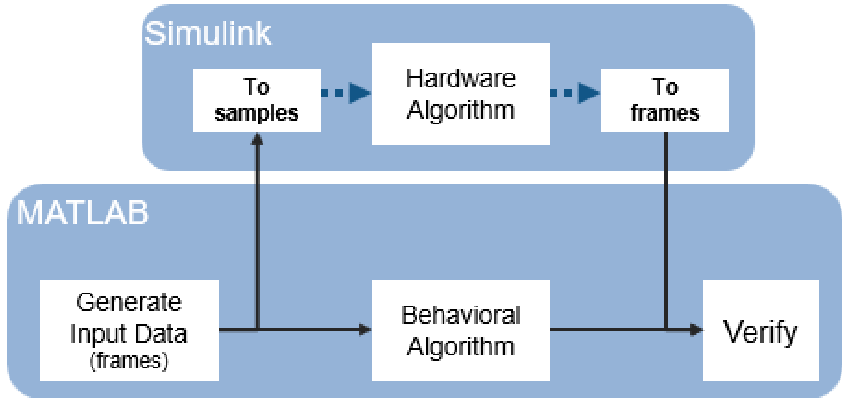


While both MATLAB and Simulink support automatic generation of HDL code, you must construct your design with hardware requirements in mind, and Simulink is better-suited for cycle-based modeling for hardware. It can represent parallel data paths and streaming data with control signals to manage the timing of the data stream. To aid in fixed-point type choices, it clearly visualizes data type propagation in the design. It also allows for easy pipelining of mathematical operations to improve maximum clock frequency in hardware.

While you create your hardware-ready design, use the MATLAB algorithm as a "golden reference" to verify that each version of the design still meets the mathematical requirements. The workflow shown in the diagram uses MATLAB and Simulink as collaboration and communication tools between the algorithm and hardware design teams.



For instance, when designing for LTE, 5G, WLAN, and satellite communication wireless standards, you can use LTE Toolbox, 5G Toolbox, WLAN Toolbox, and Satellite Communications Toolbox functions to create a golden reference in MATLAB. Then transition to Simulink and create a hardware-compatible implementation by using library blocks from Wireless HDL Toolbox and DSP HDL Toolbox that support HDL code generation. You can reuse test and data generation infrastructure from MATLAB by importing data from MATLAB to your Simulink model and returning the output of the model to MATLAB to verify it against the "golden reference".



HDL-Optimized Blocks

Library blocks from Wireless HDL Toolbox implement encoders, decoders, modulators, demodulators, and sequence generators for use in an LTE, 5G, WLAN, satellite communications, or custom OFDM-based wireless communications system. These blocks use a standard streaming data interface for hardware. This interface makes it easy to connect parts of the algorithm together, and includes control signals that manage the flow of data and mark frame boundaries. These blocks support automatic HDL code generation with HDL Coder. You can also use blocks from DSP HDL Toolbox that support HDL code generation.

The blocks provide hardware-suitable architectures that optimize resource use, such as including adder and multiplier pipelining to fit well into FPGA DSP slices. They also support automatic and configurable fixed-point data types. Using predefined blocks also allows you to try different parameter configurations without changing the rest of the design.

For lists of blocks that support HDL code generation, see Wireless HDL Toolbox Block List (HDL Code Generation) and DSP HDL Toolbox Block List (HDL Code Generation).

Reference Applications

Wireless HDL Toolbox provides reference applications that contain hardware-ready implementations of large parts of the LTE, 5G NR, Satellite, WLAN, and custom OFDM-based communications physical layer. The subsystems in these reference applications have also been tested on hardware boards. These designs are verified against the "golden reference" functions provided by LTE Toolbox, 5G Toolbox, WLAN Toolbox, or Satellite Communications Toolbox. They are designed to be modular, scalable, and extensible so you can insert additional physical channels.

These reference applications can be used as-is to deliver packet information to your unique application and to generate synthesizable VHDL or Verilog with HDL Coder. They also serve as examples to illustrate recommended practices for implementing communications algorithms on FPGA or ASIC hardware.

Generate HDL Code and Prototype on FPGA

Wireless HDL Toolbox provides blocks that support HDL code generation. To generate HDL code from designs that use these blocks, you must have an HDL Coder license. HDL Coder produces device-independent code with signal names that correspond to the Simulink model. HDL Coder also provides

a tool to drive the FPGA synthesis and targeting process, and enables you to generate scripts and test benches for use with third-party HDL simulators.

To assist with the setup and targeting of programmable logic on a prototype board, and to verify your wireless communications system design on hardware, download the required hardware support package SoC Blockset™ Support Package for Xilinx Devices.

See Also

External Websites

- Wireless HDL Toolbox
- HDL Coder

See Also

Related Examples

- “Prototype Wireless Communications Algorithms on Hardware” (Wireless HDL Toolbox)

Generating HDL Code for Multirate Models

- “Code Generation from Multirate Models” on page 20-2
- “Timing Controller for Multirate Models” on page 20-4
- “Generate Reset for Timing Controller” on page 20-6
- “Multirate Model Requirements for HDL Code Generation” on page 20-7
- “Generate a Global Oversampling Clock” on page 20-9
- “Using Multiple Clocks in HDL Coder” on page 20-14
- “Using Triggered Subsystems for HDL Code Generation” on page 20-19
- “Use Triggered Subsystem for Asynchronous Clock Domain” on page 20-25
- “Meet Timing Requirements Using Enable-Based Multicycle Path Constraints” on page 20-32
- “Use Multicycle Path Constraints to Meet Timing for Slow Paths” on page 20-38

Code Generation from Multirate Models

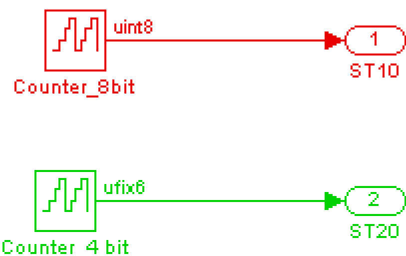
HDL Coder supports HDL code generation for single-clock and multiple clock multirate models. Your model can include blocks running at multiple sample rates:

- Within the device under test (DUT).
- In the test bench driving the DUT. In this case, the DUT inherits multiple sample rates from its inputs or outputs.
- In both the test bench and the DUT.

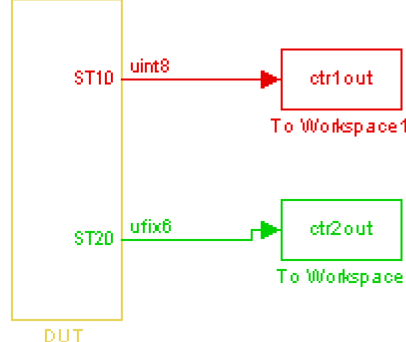
In general, generating HDL code for a multirate model does not differ greatly from generating HDL code for a single-rate model. However, there are a few requirements and restrictions on the configuration of the model and the use of specialized blocks (such as Rate Transitions) that apply to multirate models. For details, see “Multirate Model Requirements for HDL Code Generation” on page 20-7.

Clock Enable Generation for a Multirate DUT

The following block diagram shows the interior of a subsystem containing blocks that are explicitly configured with different sample times. The upper and lower Counter Free-Running blocks have sample times of 10 s and 20 s respectively. The counter output signals are routed to output ports `ST10` and `ST20`, which inherit their sample times. The signal path terminating at `ST10` runs at the base rate of the model; the signal path terminating at `ST20` is a subrate signal, running at half the base rate of the model.



As shown in the next figure, the outputs of the multirate DUT drive To Workspace blocks in the test bench. These blocks inherit the sample times of the DUT outputs.

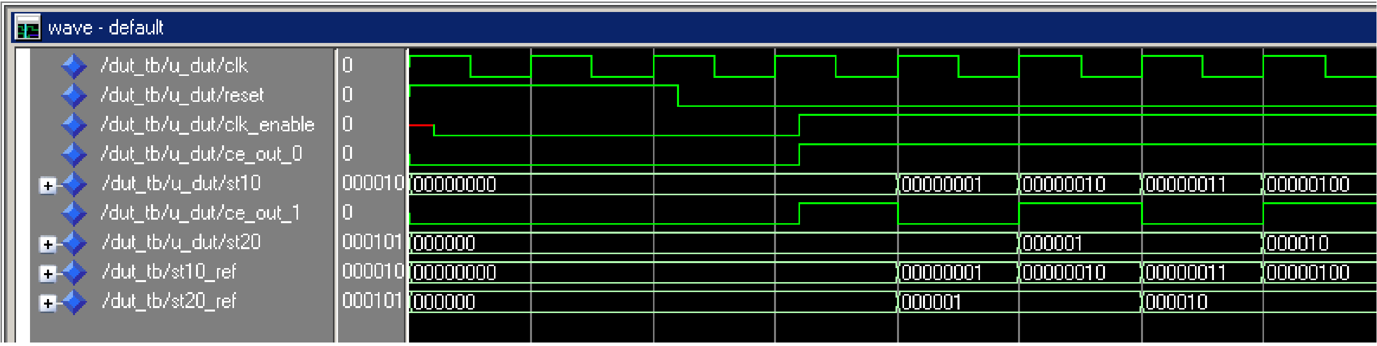


The following listing shows the VHDL entity declaration generated for the DUT.

```
ENTITY DUT IS
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        clk_enable                        :   IN    std_logic;
        ce_out_0                          :   OUT   std_logic;
        ce_out_1                          :   OUT   std_logic;
        ST10                              :   OUT   std_logic_vector(7 DOWNTO 0);  -- uint8
        ST20                              :   OUT   std_logic_vector(5 DOWNTO 0)  -- ufix6
        );
END DUT;
```

The entity has the standard clock, reset, and clock enable inputs and data outputs for the `ST10` and `ST20` signals. In addition, the entity has two clock enable outputs (`ce_out_0` and `ce_out_1`). These clock enable outputs replicate internal clock enable signals maintained by the timing controller entity.

The following figure, showing a portion of a Mentor Graphics ModelSim simulation of the generated VHDL code, lets you observe the timing relationship of the base rate clock (`clk`), the clock enables, and the computed outputs of the model.



After the assertion of `clk_enable` (replicated by `ce_out_0`), a new value is computed and output to `ST10` for every cycle of the base rate clock.

A new value is computed and output for subrate signal `ST20` for every other cycle of the base rate clock. An internal signal, `enb_1_2_1` (replicated by `ce_out_1`) governs the timing of this computation.

Timing Controller for Multirate Models

A multirate model is model that contains blocks running at multiple sample rates. You can create a multirate model by adding blocks that run at different sample rates or by enabling an HDL optimization, such as streaming, sharing, or clock-rate pipelining, for a single rate model. In a multirate model, HDL Coder creates a timing controller entity to define timing signals for the model. These timing signals are signals such as a clock, reset, external clock enable inputs and a clock enable output. A timing controller generates the required rates from a single primary clock by using one or more counters to create multiple clock enables.

In single clock mode:

- HDL Coder generates a timing controller if you have a multirate model.
- The primary clock rate is the fastest rate in the model.
- HDL code generated from multirate models employs a single primary clock input that corresponds to the base rate of the DUT.

In multiple clock mode:

- HDL Coder generates a timing controller if the DUT contains downsampling operations that require a clock divider. The outputs of the timing controller are clock enable signals that run at rates that are an integer multiple slower than the timing controller primary clock.
- There is no primary clock rate in multiple clock mode.
- HDL code generated from multirate models employs one clock input for each rate in the DUT. The number of clock dividers generated in multiple clock mode depends on how many unique downsampling requests there are across various rates.

Timing Controller Naming

The timing controller generates a set of clock enables that contain the necessary rate and phase information to control the clocking for the design in the name of each clock enable. The naming of the clock enables shows the clock enable in relation to the model base rate. The model base rate might not be in the DUT. This naming convention keeps the naming consistent for mapping sample rate colors to the generated HDL code. For information on sample rate colors, see “View Sample Time Information”.

For example, if the model base rate is not used inside the DUT, the clock in a timing controller can be labeled as `enb_1_2_0` instead of `enb` because the model base rate is the fastest rate in the model. In this case, `enb_1_2_0` means the model base rate is twice as fast as the DUT base rate.

Each timing controller entity definition is written to a separate code file. The timing controller file and entity names are derived from the name of the DUT. To form the timing controller name, HDL Coder appends the value of the `TimingControllerPostfix` property to the DUT name. For example, if the name of your DUT is `my_test`, in the default case HDL Coder adds the `TimingControllerPostfix` value, `_tc`, to form the timing controller name `my_test_tc`.

See Also

Functions
`makehdl` | `makehdltb`

Model Settings
Timing controller postfix | Timing controller architecture | Optimize timing controller

Related Examples

- “Using Multiple Clocks in HDL Coder” on page 20-14
- “Create Multirate Model with Integer Clock Multiples by Clock Division” on page 18-58

Generate Reset for Timing Controller

In this section...

"Requirements for Timing Controller Reset Port Generation" on page 20-6
"How To Generate Reset for Timing Controller" on page 20-6
"Limitations for Timing Controller Reset Port Generation" on page 20-6

You can generate a reset port for the timing controller, which generates the clock, clock enable, and reset signals in a multirate DUT. In the generated code, the reset for the timing controller is a DUT input port.

Requirements for Timing Controller Reset Port Generation

Your design must use single-clock mode. That is, the `ClockInputs` property value must be `'Single'`.

How To Generate Reset for Timing Controller

To generate a reset port for the timing controller, set the `TimingControllerArch` property to `'resettable'` using `makehdl` or `hdlset_param`.

To disable reset port generation for the timing controller, set the `TimingControllerArch` property to `'default'`.

For example, for a model, *`sfir_fixed`*, specify a reset port for the timing controller by entering:

```
hdlset_param('sfir_fixed','TimingControllerArch','resettable')
```

Limitations for Timing Controller Reset Port Generation

The following workflows are not compatible with timing controller reset port generation:

- FPGA-in-the-Loop
- Custom IP core generation

Multirate Model Requirements for HDL Code Generation

In this section...

"Model Configuration Parameters" on page 20-7
"Sample Rate" on page 20-7
"Blocks To Use For Rate Transitions" on page 20-7

Model Configuration Parameters

Before generating HDL code, configure the parameters of your model using the `hdlsetup` command. This sets up your multirate model for HDL code generation. This section summarizes settings applied to the model by `hdlsetup` that are relevant to multirate code generation. These include:

- **Solver** options that are recommended or required for HDL code generation:
  - **Type**: `Fixed-step`.
  - **Solver**: `Discrete (no continuous states)`. Other fixed-step solvers could be selected, but this option is usually best for simulating discrete systems.
  - **Treat each discrete rate as a separate task**: Clear for single tasking mode.
- `hdlsetup` configures the following **Diagnostics** / **Sample time** options for all models:
  - **Multitask data transfer**: `error`
  - **Single task data transfer**: `error`

  In multirate models intended for HDL code generation, Rate Transition blocks must be explicitly inserted when blocks running at different rates are connected. Set **Multitask data transfer** and **Single task data transfer** to `error` to detect illegal rate transitions before code is generated.

To learn more about the settings that `hdlsetup` configures, see "Check for model parameters suited for HDL code generation" on page 37-5.

Sample Rate

HDL Coder requires that at least one valid sample rate (sample time > 0) must exist in the model. If all rates are 0, –1, or –2, the code generator (`makehdl`) and compatibility checker (`checkhdl`) terminates with an error message.

Blocks To Use For Rate Transitions

Use Rate Transition blocks, rather than the following blocks, to create rate transitions in models intended for HDL code generation:

- Delay
- Tapped Delay
- Unit Delay
- Unit Delay Enabled
- Zero-Order Hold

The Delay blocks listed should be configured to have the same input and output sample rates.

Zero-Order Hold blocks must be configured with inherited (-1) sample times.

Generate a Global Oversampling Clock

In this section...

"Specifying the Oversampling Value" on page 20-9
"Requirements for the Oversampling Factor" on page 20-10
"Resolving Oversampling Rate Conflicts" on page 20-10

In many designs, the DUT is not self-contained. For example, consider a DUT that is part of a larger system that supplies timing signals to its components under control of a global clock. The global clock typically runs at a higher rate than some of the components under its control. By specifying a global oversampling clock, you can integrate your DUT into a larger system without using Upsample or Downsample blocks.

To generate global clock logic, you specify an oversampling value. The oversampling value expresses the desired rate of the global oversampling clock as a multiple of the base rate of your model. When you specify an oversampling value, HDL Coder generates the global oversampling clock and derives the required timing signals from clock signal. The global oversampling clock affects only the generated HDL code. The clock does not affect the simulation behavior of your model.

Specifying the Oversampling Value

You can use two different parameters to set an oversampling value for your model, depending on your design:

- If you are modeling with actual hardware rates, enable the **Treat Simulink rates as actual hardware rates** parameter to set an oversampling value for your model automatically.
- If you are modeling with relative rates in Simulink, set the **Oversampling factor** parameter to a value greater than `1` to set an oversampling value for your model.

When you enable **Treat Simulink rates as actual hardware rates**, HDL Coder can automatically adjust the oversampling value for your model if changes in Simulink rate or target frequency occur, without requiring you to update the **Oversampling factor** manually.

Specify the Oversampling Value Using the Configuration Parameters Dialog Box

When modeling with actual hardware rates which means that the Simulink rates reflect data rates on hardware, you can specify the oversampling value for a global clock from the Configuration Parameter dialog box by:

1. Setting a value greater than zero for the **Target Frequency** parameter in the **HDL Code Generation** > **Target** pane.
2. Selecting the **Treat Simulink rates as actual hardware rates** check box in the **HDL Code Generation** > **Global Settings** pane.

When modeling with relative rates which means that the Simulink rates are normalized to a base rate of one, you can specify the oversampling value for a global clock from the Configuration Parameter dialog box by setting the **Oversampling factor** parameter to a value greater than `1` in the **HDL Code Generation** > **Global Settings** pane.

Specify the Oversampling Value From the Command Line

When modeling with actual hardware rates, you can specify the oversampling value for a global clock from the command line by setting the `TargeFrequency` property for your model and enabling the

TreatRatesAsHardwareRates property by using the hdlset_param or makehdl functions. For example, to specify a target frequency of 200MHz and enable HDL Coder to set an oversampling value for your current model automatically, use these commands:

```
hdlset_param(gcs,'TargetFrequency',200)
hdlset_param(gcs,'TreatRatesAsHardwareRates','on')
```

When modeling with relative rates, you can specify the oversampling value for a global clock from the command line by setting the Oversampling property with the hdlset_param or makehdl functions. For example, to specify an oversampling factor of 7 for your current model, use this command:

```
hdlset_param(gcs,'Oversampling',7)
```

Requirements for the Oversampling Factor

When you specify the oversampling value manually by using the **Oversampling factor** parameter:

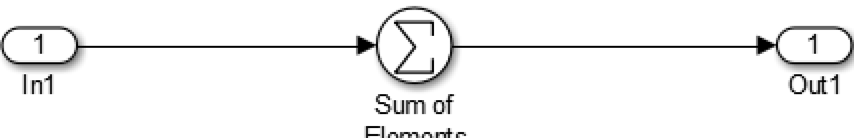
- The **Oversampling factor** must be an integer greater than or equal to 1.
- The default value for the **Oversampling factor** parameter is 1. In the default case, HDL Coder does not generate a global oversampling clock.
- Some DUTs require multiple sampling rates for their internal operations. In such cases, the other rates in the DUT must divide evenly into the global oversampling rate. For more information, see “Resolving Oversampling Rate Conflicts” on page 20-10.

Resolving Oversampling Rate Conflicts

The HDL realization of some designs is inherently multirate, even though the original Simulink model is single-rate. As an example, consider the simplevectorsum_cascade model.

This model consists of a subsystem, vsum, driven by a vector input of width 10, with a scalar output. The following figure shows the root level of the model.

The device under test is the `vsum` subsystem, shown in the following figure. The subsystem contains a Sum block, configured for vector summation.



The `simplevectorsum_cascade` model specifies a cascaded implementation (`SumCascadeHDLEmission`) for the Sum block. The generated HDL code for a cascaded vector Sum block implementation runs at two effective rates: a faster (oversampling) rate for internal computations and a slower rate for input/output. HDL Coder reports that the inherent oversampling rate for the DUT is five times the base rate:

```
dut = 'simplevectorsum_cascade/vsum';
makehdl(dut);
```

```
### Generating HDL for 'simplevectorsum_cascade/vsum'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.
```

```
### The code generation and optimization options you have chosen have introduced
    additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for
    compensation.
### The DUT requires an initial pipeline setup latency. Each output port
    experiences these additional delays
### Output port 0: 1 cycles

### Begin VHDL Code Generation
### MESSAGE: The design requires 5 times faster clock with respect to the
    base rate = 1.
...
```

In some cases, the clock requirements for such a DUT conflict with the global oversampling rate. To avoid oversampling rate conflicts, verify that subrates in the model divide evenly into the global oversampling rate.

For example, if you request a global oversampling rate of 8 for the `simplevectorsum_cascade` model, the coder displays a warning and ignores the requested oversampling factor. The coder instead respects the oversampling factor that the DUT requests:

```
dut = 'simplevectorsum_cascade/vsum';
makehdl(dut,'Oversampling',8);
```

```
### Generating HDL for 'simplevectorsum_cascade/vsum'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### The code generation and optimization options you have chosen have introduced
    additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for
    compensation.
### The DUT requires an initial pipeline setup latency. Each output port
    experiences these additional delays
### Output port 0: 1 cycles

### Begin VHDL Code Generation
### WARNING: The design requires 5 times faster clock with respect to
        the base rate = 1, which is incompatible with the oversampling
        value (8). Oversampling value is ignored.
...
```

An oversampling factor of 10 works in this case:

```
dut = 'simplevectorsum_cascade/vsum';
makehdl(dut,'Oversampling',10);
```

```
### Generating HDL for 'simplevectorsum_cascade/vsum'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.


### The code generation and optimization options you have chosen have introduced
    additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for
    compensation.
### The DUT requires an initial pipeline setup latency. Each output port
    experiences these additional delays
### Output port 0: 1 cycles

### Begin VHDL Code Generation
### MESSAGE: The design requires 10 times faster clock with respect to
    the base rate = 1.
...
```

See Also

Related Examples

- "Clock-Rate Pipelining" on page 21-148
- "Optimization with Constrained Overclocking" on page 21-23

Using Multiple Clocks in HDL Coder

This example shows how to instantiate multiple top-level synchronous clock input ports in HDL Coder™.

Overview of Clocking Modes

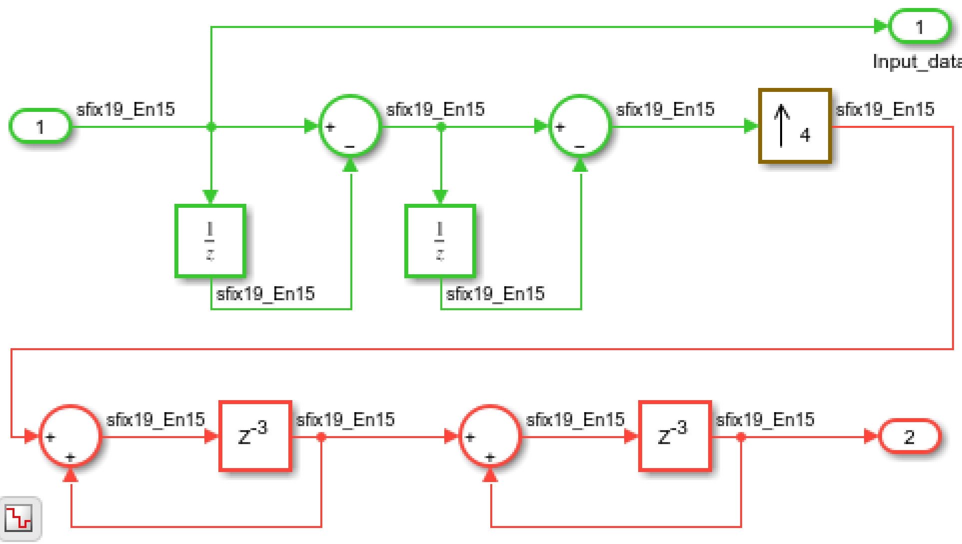
HDL Coder has two clocking modes. One mode generates a single clock input to the Device Under Test (DUT). The other mode generates a synchronous primary clock input for each Simulink® rate in the DUT. By default, HDL Coder creates an HDL design that uses a single clock port for the DUT. In single clock mode, if multiple rates exist in the Simulink model, a timing controller is created to control the clocking to the portions of the model that run at a slower rate. The timing controller generates a set of clock enables with the necessary rate and phase information to control the clocking for the design. Each generated clock enable is an integer multiple slower than the primary clock rate. Each output signal rate is associated with a clock enable output signal that indicates the correct timing to sample the output data.

In synchronous multiple clock mode, the generated code has a set of clock ports as primary inputs to the DUT. Each clock port corresponds to a separate rate in the model. Transitions between rates require clock enables at a given rate that are out of phase with that rate's clock. These out of phase signals are generated with a timing controller. A multiple clock model may require multiple timing controllers.

When using multiple clocks for multirate model, it is recommended to add sequential logic such as delay block at each Simulink rate. If the sequential logic is not present at a particular Simulink rate, HDL Coder does not generate separate clock port for that rate. For example, in a multirate model consisting of Downsample block, add a unit delay block after the Downsample block to generate the clock port of that downsampling rate.

The first example uses a multirate CIC Interpolation filter in single clock mode. The filter's input is also presented as an output for this example to present a model with output signals running at different rates.

```
load_system('hdlcoder_clockdemo');
open_system('hdlcoder_clockdemo/DUT');
set_param('hdlcoder_clockdemo', 'SimulationCommand', 'update');
```



Single Clock Mode DUT Timing Interface

In single clock mode, the HDL code for the DUT has a set of three signals that do not appear in the Simulink diagram added to it. Collectively, these are a clock bundle that contains signals for clock, primary clock enable, and reset. These signals appear in the VHDL Entity declaration and are used throughout the generated code.

```
hdlset_param('hdlcoder_clockdemo', 'Traceability', 'on');
makehdl('hdlcoder_clockdemo/DUT');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_clockdemo')">hdlcoder_clockdemo</a
### Generating HDL for <a href="matlab:open_system('hdlcoder_clockdemo/DUT')">hdlcoder_clockdemo/
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_clockde
### Running HDL checks on the model 'hdlcoder_clockdemo'.
### Begin compilation of the model 'hdlcoder_clockdemo'...
### Working on the model 'hdlcoder_clockdemo'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_clockdemo', 'GenerateModel')"
### Begin model generation 'gm_hdlcoder_clockdemo'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_clockdemo\gm_hdlcoder_c
### Begin VHDL Code Generation for 'hdlcoder_clockdemo'.
### Begin VHDL Code Generation for 'DUT_tc'.
### Working on DUT_tc as hdlsrc\hdlcoder_clockdemo\DUT_tc.vhd.
### Code Generation for 'DUT_tc' completed.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_clockdemo', 'Traceability')">
### Working on hdlcoder_clockdemo/DUT as hdlsrc\hdlcoder_clockdemo\DUT.vhd.
### Generating package file hdlsrc\hdlcoder_clockdemo\DUT_pkg.vhd.
### Code Generation for 'hdlcoder_clockdemo' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_clockdemo' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

Clock Summary Reporting in Single Clock Mode

The file comment block in the HDL DUT code contains Clock Summary information. In single clock mode, this report contains a table detailing the sample rates for each clock enable output signal. The report also contains a table listing each user output signal and its associated clock enable output signal. Any time a HTML report is generated, the Clock Summary Report is also generated.

Generating Synchronous Multiclock HDL Code

To generate multiple synchronous clocks for this design, the `ClockInputs` property must be set to `multiple`. Either change the `ClockInputs` property at the command line using `makehdl` or change the Clock inputs setting to `Multiple` on the **HDL Code Generation > Global Settings** tab of the Configuration Parameters dialog box.

```
makehdl('hdlcoder_clockdemo/DUT', 'ClockInputs', 'multiple');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_clockdemo')">hdlcoder_clockdemo</a
### Generating HDL for <a href="matlab:open_system('hdlcoder_clockdemo/DUT')">hdlcoder_clockdemo/
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_clockde
### Running HDL checks on the model 'hdlcoder_clockdemo'.
### Begin compilation of the model 'hdlcoder_clockdemo'...
### Working on the model 'hdlcoder_clockdemo'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_clockdemo', 'GenerateModel')"
### Begin model generation 'gm_hdlcoder_clockdemo'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_clockdemo\gm_hdlcoder_c
### Begin VHDL Code Generation for 'hdlcoder_clockdemo'.
### Begin VHDL Code Generation for 'DUT_tc'.
### Working on DUT_tc as hdlsrc\hdlcoder_clockdemo\DUT_tc.vhd.
### Code Generation for 'DUT_tc' completed.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_clockdemo', 'Traceability')">
### Working on hdlcoder_clockdemo/DUT as hdlsrc\hdlcoder_clockdemo\DUT.vhd.
### Generating package file hdlsrc\hdlcoder_clockdemo\DUT_pkg.vhd.
### Code Generation for 'hdlcoder_clockdemo' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_clockdemo' complete with 0 errors, 1 warnings, and 1 messages.
### HDL code generation complete.
```

Clock Summary Information in Multiclock Mode

The contents of the Clock Summary are different in multiple clock mode. The report now contains a clock table. This table has one entry for each primary DUT clock. It describes the relative clock ratio between each clock and the fastest clock in the model. As with single clock mode, this information is presented both in the HDL DUT file comment block and the HTML report.

Multiclock Mode and HDL Coder Optimizations

Multiple synchronous clocks can be useful even for a design with only a single Simulink rate. Various optimizations can require clock rates faster than indicated in the original model. The following example demonstrates an audio filtering model that applies the same filter on the left and right channels. By default, HDL Coder would generate two filter modules in hardware. With this configuration, multiple clock mode still only generates one clock, just as single clock mode does.

```
bdclose hdlcoder_clockdemo;
load_system('hdlcoder_audiofiltering');
```

```
open_system('hdlcoder_audiofiltering/Audio filter');
hdlset_param('hdlcoder_audiofiltering', 'ClockInputs', 'Multiple');
hdlset_param('hdlcoder_audiofiltering/Audio filter', 'SharingFactor', 0);
makehdl('hdlcoder_audiofiltering/Audio filter', 'Traceability', 'on');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_audiofiltering')">hdlcoder_audiof
### Generating HDL for <a href="matlab:open_system('hdlcoder_audiofiltering/Audio filter')">hdlc
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_audiof
### Running HDL checks on the model 'hdlcoder_audiofiltering'.
### Begin compilation of the model 'hdlcoder_audiofiltering'...
### Working on the model 'hdlcoder_audiofiltering'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_audiofiltering', 'GenerateMod
### Begin model generation 'gm_hdlcoder_audiofiltering'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_audiofiltering\gm_hdlc
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_audiofiltering', 'Traceabilit
### Working on hdlcoder_audiofiltering/Audio filter/Filter_left as hdlsrc\hdlcoder_audiofiltering
### Working on hdlcoder_audiofiltering/Audio filter as hdlsrc\hdlcoder_audiofiltering\Audio_filte
### Code Generation for 'hdlcoder_audiofiltering' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/t
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 1 warnings, and 0 messages.
### HDL code generation complete.
```

Using Multiple Clock Mode with Resource Sharing

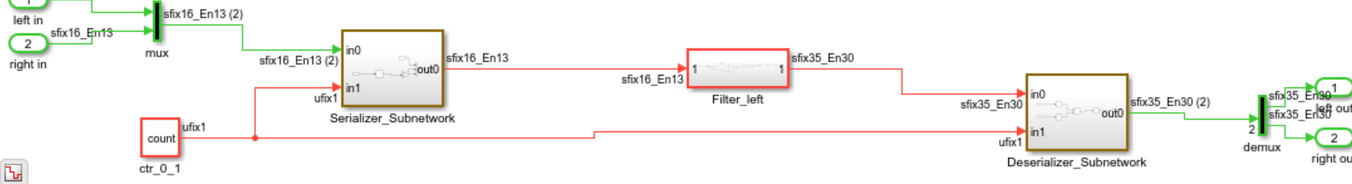
With resource sharing applied to the identical left and right channel atomic subsystems, only one filter is generated. To meet the Simulink timing requirements, the single filter is run at twice the clock rate as the original Simulink model, as is shown below. Since the resource sharing optimization creates a second clock rate, the user can use synchronous multiple clock mode to provide external clocks for both rates. In this configuration, multiple clock mode still only generates one clock. You see the message:

```
The design requires 2 times faster clock with respect to the base rate =
0.00012207.
```

```
bdclose gm_hdlcoder_audiofiltering;
hdlset_param('hdlcoder_audiofiltering/Audio filter', 'SharingFactor', 2);
makehdl('hdlcoder_audiofiltering/Audio filter', 'Traceability', 'on');
open_system('gm_hdlcoder_audiofiltering/Audio filter');
set_param('gm_hdlcoder_audiofiltering', 'SimulationCommand', 'update');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_audiofiltering')">hdlcoder_audiof
### Generating HDL for <a href="matlab:open_system('hdlcoder_audiofiltering/Audio filter')">hdlc
```

```
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_audiofi
### Running HDL checks on the model 'hdlcoder_audiofiltering'.
### Begin compilation of the model 'hdlcoder_audiofiltering'...
### Working on the model 'hdlcoder_audiofiltering'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 1 cycles.
### Output port 2: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_audiofiltering', 'GenerateMod
### Begin model generation 'gm_hdlcoder_audiofiltering'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_audiofiltering\gm_hdlco
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### MESSAGE: The design requires 2 times faster clock with respect to the base rate = 0.00012207
### Working on... <a href="matlab:configset.internal.open('hdlcoder_audiofiltering', 'Traceabilit
### Working on hdlcoder_audiofiltering/Audio filter/Filter_left as hdlsrc\hdlcoder_audiofiltering
### Working on hdlcoder_audiofiltering/Audio filter as hdlsrc\hdlcoder_audiofiltering\Audio_filte
### Generating package file hdlsrc\hdlcoder_audiofiltering\Audio_filter_pkg.vhd.
### Code Generation for 'hdlcoder_audiofiltering' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 1 warnings, and 1 messages.
### HDL code generation complete.
```



See Also

- “Multirate Model Requirements for HDL Code Generation” on page 20-7

Using Triggered Subsystems for HDL Code Generation

In this section...

"Best Practices" on page 20-19
"Using the Signal Editor Block" on page 20-19
"Using Trigger As Clock" on page 20-19
"Specify Trigger As Clock" on page 20-20
"Trigger As Clock Without Synchronous Registers" on page 20-20
"Model Trigger Signal As Clock in Triggered Subsystem" on page 20-20
"Use Triggered and Resettable Subsystem to Model Clock and Reset Signals" on page 20-21
"Model Single Clock and Reset Signal Using Triggered and Resettable Subsystems" on page 20-22
"Limitations" on page 20-23

The Triggered Subsystem block is a Subsystem block that executes each time the control signal has a trigger value. To learn more about the block, see Triggered Subsystem.

Best Practices

When using triggered subsystems in models targeted for HDL code generation, consider the following:

- For synthesis results to match Simulink results, drive the trigger port with registered logic (with a synchronous clock) on the FPGA.
- It is good practice to put unit delays on Triggered Subsystem output signals. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
- The use of triggered subsystems can affect synthesis results in the following ways:
  - In some cases, the system clock speed can drop by a small percentage.
  - Generated code uses more resources, scaling with the number of triggered subsystem instances and the number of output ports per subsystem.

Using the Signal Editor Block

When you connect outputs from a Signal Editor block to a triggered subsystem, you might need to use a Rate Transition block. To run all triggered subsystem ports at the same rate:

- If the trigger source is a Signal Editor block, but the other triggered subsystem inputs come from other sources, insert a Rate Transition block into the signal path before the trigger input.
- If all inputs (including the trigger) come from a Signal Editor block, they have the same rate, so special action is not required.

Using Trigger As Clock

Using the trigger as clock in triggered subsystems enables you to partition your design into different clock regions in the generated code. Make sure that the **Clock edge** setting in the Configuration Parameters dialog box matches the **Trigger type** of the Trigger block inside the triggered subsystem.

For example, you can model:

- A design with clocks that run at the same rate, but out of phase.
- Clock regions driven by an external or internal clock divider.
- Clock regions driven by clocks whose rates are not integer multiples of each other.
- Internally generated clocks.
- Clock gating for low-power design.

Note Using the trigger as clock for triggered subsystems can result in timing mismatches of one cycle during testbench simulation.

Specify Trigger As Clock

- In **HDL Code Generation** > **Global Settings** > **Ports** tab, select **Use trigger signal as clock**.
- Set the `TriggerAsClock` property using `makehdl` or `hdlset_param`. For example, to generate HDL code that uses the trigger signal as clock for triggered subsystems in a DUT subsystem, `myDUT`, in a model, `myModel`, enter:

  ```
  makehdl ("myModel/myDUT",TriggerAsClock="on")
  ```

Trigger As Clock Without Synchronous Registers

When you use the trigger as clock in triggered subsystem, each triggered subsystem input or output requires synchronization delays immediately outside and immediately inside the subsystem. These delays act as a synchronization interface between the regions running at different rates. HDL Coder can allows you to generate HDL code without adding the synchronization delays by enabling the "trigger as clock without synchronous register" option. By default, this option is `on`.

You can enable or disable this option by using `makehdl` or `hdlset_param` function. For example, to generate an HDL code for a DUT subsystem, `myDUT` in a model, `myModel`, that uses trigger as a clock for triggered subsystem without having synchronization delays, enter:

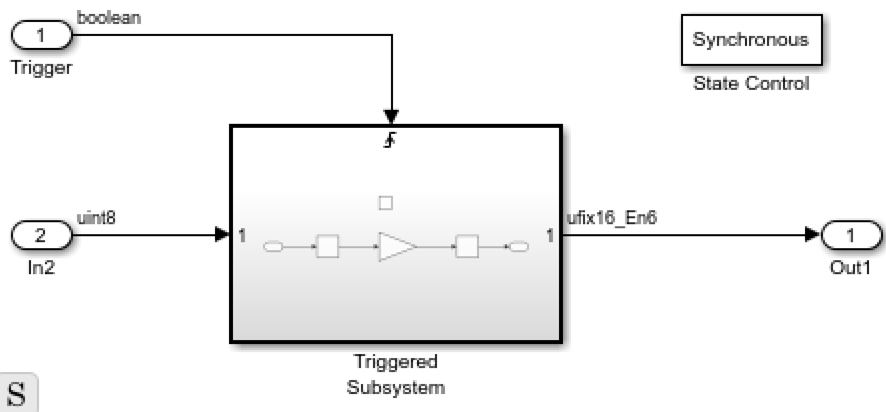
```
makehdl("myModel/myDUT",TriggerAsClockWithoutSyncRegisters="on")
```

Model Trigger Signal As Clock in Triggered Subsystem

This example shows how to model trigger port of the Triggered Subsystem as a clock signal. Using trigger as clock functionality in the Triggered Subsystem enables you to use trigger signal as a clock in your generated HDL code.

The model `TriggerAsClockSingle` has a `DUT` subsystem which contains a Triggered Subsystem. Load and open the `TriggerAsClockSingle` model by running these commands:

```
load_system("TriggerAsClockSingle");
set_param("TriggerAsClockSingle",'SimulationCommand','Update')
open_system("TriggerAsClockSingle/DUT");
```



To use trigger signal as clock in your generated HDL code, enable **Use trigger signal as clock** option in the **HDL Code Generation > Global Settings > Ports** tab of the configuration settings. Then, generate the HDL code for a DUT subsystem using makehdl command or HDL Coder™ app:

```
makehdl("TriggerAsClockSingle/DUT")
```

In the generated HDL code, HDL Coder maps the trigger port of the Triggered Subsystem to the clock. This code snippet shows the HDL code of Triggered Subsystem which uses Trigger signal as a clock.

```
    Delay1_process : PROCESS (Trigger, reset)
    BEGIN
        IF reset = '1' THEN
            Delay1_out1 <= to_unsigned(16#0000#, 16);
        ELSIF Trigger'EVENT AND Trigger = '1' THEN
            Delay1_out1 <= Gain_out1;
        END IF;
    END PROCESS Delay1_process;
```

Use Triggered and Resettable Subsystem to Model Clock and Reset Signals

You can model control signals, such as clock and reset, by using the triggered and resettable subsystem. You can use trigger as clock functionality to model trigger port from triggered subsystem as a clock and use resettable subsystem to model reset port from Simulink.

When you use triggered or resettable subsystem in your model targeted for HDL code generation, you can:

- Model a clock and reset signal from Simulink by including a resettable subsystem inside the triggered subsystem.
- Use a triggered subsystem with synchronous semantics.
- Generate a code with a single clock and reset for a nested resettable subsystem inside a triggered subsystem by using the **Use trigger signal as clock** and **Minimize global resets** parameters.
- Generate code that has multiple clock and reset signals for a model consisting of multiple triggered and resettable subsystem.
- Use Unit Delay Enabled Synchronous block inside the Triggered subsystem.

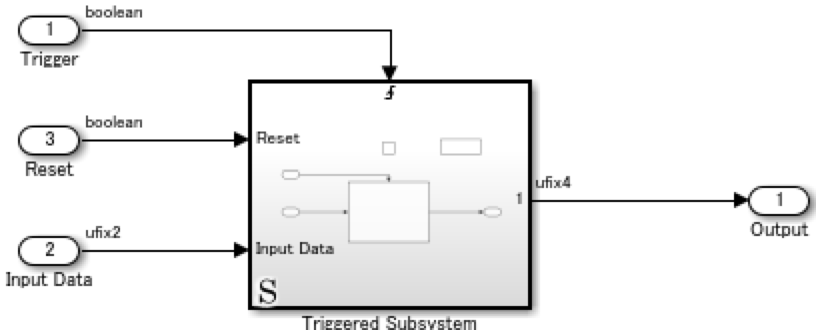
- Model a Unit Delay Resettable Synchronous block in the triggered subsystem by adding the Unit Delay block to a resettable subsystem with synchronous semantics and placing the resettable subsystem inside triggered subsystem.
- Model a Unit Delay Enabled Resettable Synchronous block in the triggered subsystem by adding the Unit Delay Enabled block to a resettable subsystem with synchronous semantics and placing the resettable subsystem inside triggered subsystem.

Model Single Clock and Reset Signal Using Triggered and Resettable Subsystems

This example shows how to model clock and reset signal using Triggered and Resettable subsystems. You can use trigger as clock functionality to model trigger port from triggered subsystem as a clock and use resettable subsystem to model reset port.

The model `ClockAndResetModellingUsingTriggerAsClock` has a `DUT` subsystem which contains a Triggered Subsystem. Load and open the `ClockAndResetModellingUsingTriggerAsClock` model by running these commands:

```
load_system("ClockAndResetModellingUsingTriggerAsClock");
set_param("ClockAndResetModellingUsingTriggerAsClock",'SimulationCommand','Update')
open_system("ClockAndResetModellingUsingTriggerAsClock/DUT");
```



A resettable subsystem is placed within the triggered subsystem. Using resettable subsystem, you can model a reset port for your model. Minimize the global reset option so that model has single reset signal from resettable subsystem. To minimize global reset option, enable **Minimize global resets** in the **HDL Code Generation > Global Settings > Ports** tab of the configuration settings.

To use trigger signal as clock in your generated HDL code, enable **Use trigger signal as clock** option in the **HDL Code Generation > Global Settings > Ports** tab. Then, generate the HDL code for a `DUT` subsystem using `makehdl` command or HDL Coder™ app:

```
makehdl("ClockAndResetModellingUsingTriggerAsClock/DUT")
```

In the generated HDL code, HDL Coder maps the trigger port of the Triggered Subsystem to the clock. Also, the reset signal is generated from the Resettable subsystem. The HDL code of triggered subsystem shows mapping of trigger port to clock signal in the resettable subsystem.

```
module Triggered_Subsystem
          (Trigger,
```

```
                Reset_1,
                Input_Data,
                Output_rsvd);

        input   Trigger;
        input   Reset_1;
        input   [1:0] Input_Data;  // ufix2
        output  [3:0] Output_rsvd;  // ufix4

        wire [3:0] Resettable_Subsystem_out1;  // ufix4

        Resettable_Subsystem u_Resettable_Subsystem (.clk(Trigger),
                                                     .Input_Data(Input_Data),  // ufix2
                                                     .Reset_1(Reset_1),
                                                     .Output_rsvd(Resettable_Subsystem_out1)  //
                                                     );
        assign Output_rsvd = Resettable_Subsystem_out1;

    endmodule  // Triggered_Subsystem
```

Limitations

HDL Coder supports HDL code generation for triggered subsystems that meet the following conditions:

- The triggered subsystem is not the DUT.
- The subsystem is not *both* triggered *and* enabled.
- The trigger signal is a scalar.
- If the output of the subsystem is a bus then initial value of the outport must be 0.
- All inputs and outputs of the triggered subsystem (including the trigger signal) run at the same rate.
- The **Show output port** parameter of the Trigger block is set to `Off`.
- The **Latch input by delaying outside signal** check box is not selected on the Inport block inside the Triggered Subsystem.
- If the DUT contains the following blocks, `RAMArchitecture` is set to `WithClockEnable`:
  - Dual Port RAM
  - Simple Dual Port RAM
  - Single Port RAM
- The triggered subsystem does not contain the following blocks:
  - Discrete-Time Integrator
  - CIC Decimation
  - CIC Interpolation
  - FIR Decimation
  - FIR Interpolation
  - Downsample
  - Upsample
  - HDL Cosimulation blocks for HDL Verifier

- Rate Transition
- Pixel Stream FIFO (Vision HDL Toolbox)
- PN Sequence Generator, if the **Use trigger signal as clock** option is selected.

See Also

"Use Triggered Subsystem for Asynchronous Clock Domain" on page 20-25

Related Examples

- "Generate Multiple Clocks Using Trigger As Clock" on page 20-25
- "Generate Multiple Clocks and Resets Using Triggered and Resettable Subsystems" on page 20-28

Use Triggered Subsystem for Asynchronous Clock Domain

You can design a model for an asynchronous clock domain using a triggered subsystem. An asynchronous clock domain design operates at different clock regions whose clock rates are not integer multiples of one another. You can model an asynchronous clock domain design in Simulink by using multiple triggered subsystems. You can use a trigger as a clock functionality to generate separate clock signals for each triggered subsystem.

Using a triggered subsystem and a trigger as a clock, you can design a model that has one of these characteristics:

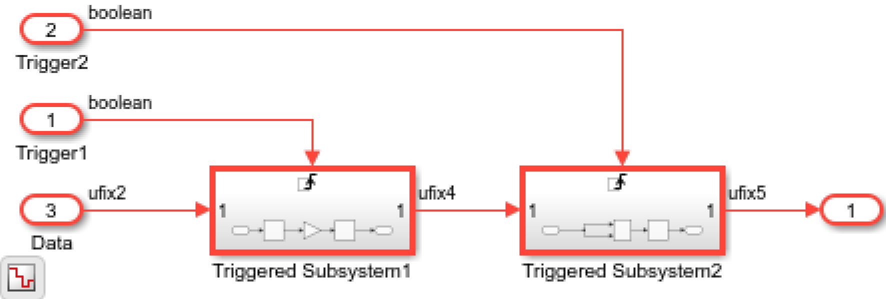
- Clock regions driven by clocks whose rates are not integer multiples of one another.
- Clocks that run at the same rate but out of phase.
- Multiple clocks that operate at different rates.
- Multiple clocks and resets modeled using nested resettable subsystems.
- Synchronous and asynchronous external resets modeled using nested triggered and resettable subsystems.

Generate Multiple Clocks Using Trigger As Clock

This example shows how to model multiple clock signals using triggered subsystems. Using trigger as clock functionality in the triggered subsystem enables you to use trigger signal as a clock in your generated HDL code. You can model an asynchronous clock domain design in Simulink® by using multiple triggered subsystem and use trigger as a clock functionality to generate separate clock signal for each triggered subsystem.

The model `AsynchronousClockUsingTriggerAsClock` has a `DUT` subsystem which contains a two Triggered Subsystem. Load and open the `AsynchronousClockUsingTriggerAsClock` model by running these commands:

```
load_system("AsynchronousClockUsingTriggerAsClock");
set_param("AsynchronousClockUsingTriggerAsClock",'SimulationCommand','Update')
open_system("AsynchronousClockUsingTriggerAsClock/DUT");
```



To use trigger signal as clock in your generated HDL code, enable **Use trigger signal as clock** option in the **HDL Code Generation > Global Settings > Ports** tab of the configuration settings. Then, generate the HDL code for a `DUT` subsystem using `makehdl` command or HDL Coder™ app:

```
makehdl("AsynchronousClockUsingTriggerAsClock/DUT")
```

In the generated HDL code, HDL Coder generates two trigger ports for the DUT subsystem which are used as clock signal in the triggered subsystems.

```
    module DUT
            (Trigger1,
            Trigger2,
            Data,
            Out1);

    input   Trigger1;
    input   Trigger2;
    input   [1:0] Data;  // ufix2
    output  [7:0] Out1;  // uint8

    wire [7:0] Triggered_Subsystem1_out1;  // uint8
    wire [7:0] Triggered_Subsystem2_out1;  // uint8

    Triggered_Subsystem1 u_Triggered_Subsystem1 (.Trigger(Trigger1),
                                                 .In1(Data),  // ufix2
                                                 .Out1(Triggered_Subsystem1_out1)  // uint8
                                                 );

    Triggered_Subsystem2 u_Triggered_Subsystem2 (.Trigger(Trigger2),
                                                 .In1(Triggered_Subsystem1_out1),  // uint8
                                                 .Out1(Triggered_Subsystem2_out1)  // uint8
                                                 );

    assign Out1 = Triggered_Subsystem2_out1;

endmodule  // DUT
```

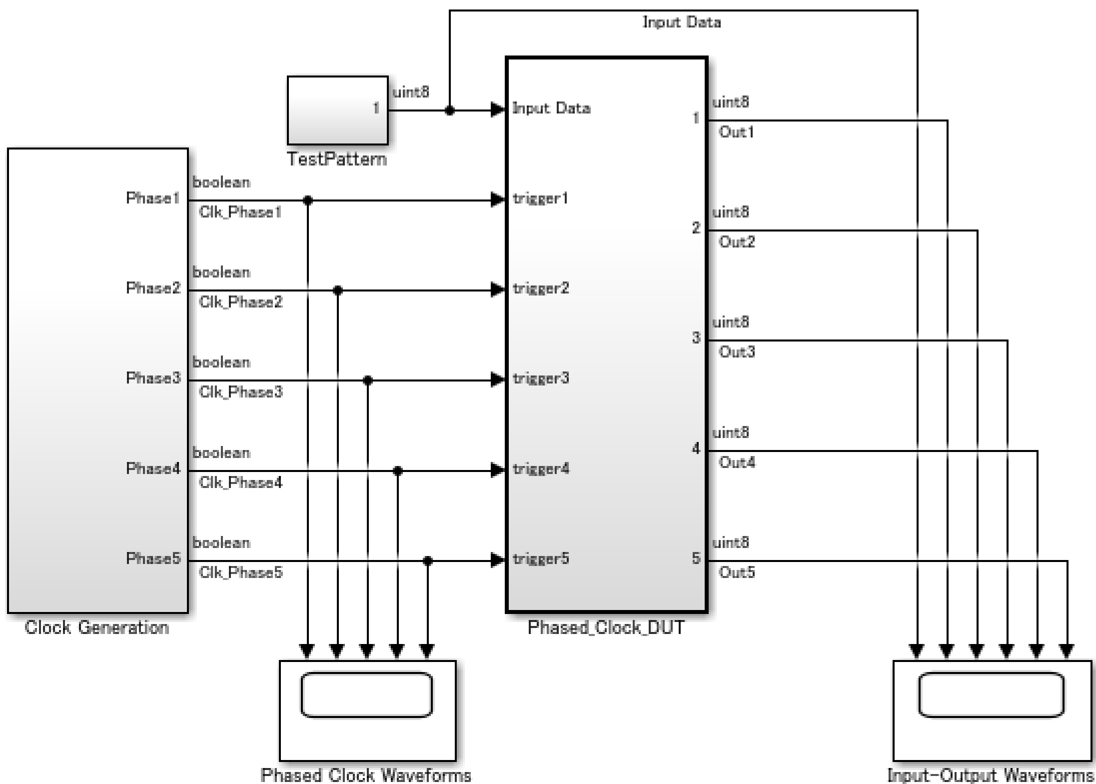
Each triggered subsystem is driven by the separate clock connected to the trigger port of triggered subsystem. You can use these clock signals to operate the subsystems at different clock rates. In the HDL code for triggered subsystem, the trigger signals are used as clock as shown in code snippet below.

```
module Triggered_Subsystem1
            (Trigger,
            In1,
            Out1);

    input   Trigger;
    input   [1:0] In1;  // ufix2
    output  [7:0] Out1;  // uint8
...
...
always @(posedge Trigger)
begin : Delay1_process
    Delay1_out1 <= Gain_out1;
end
```

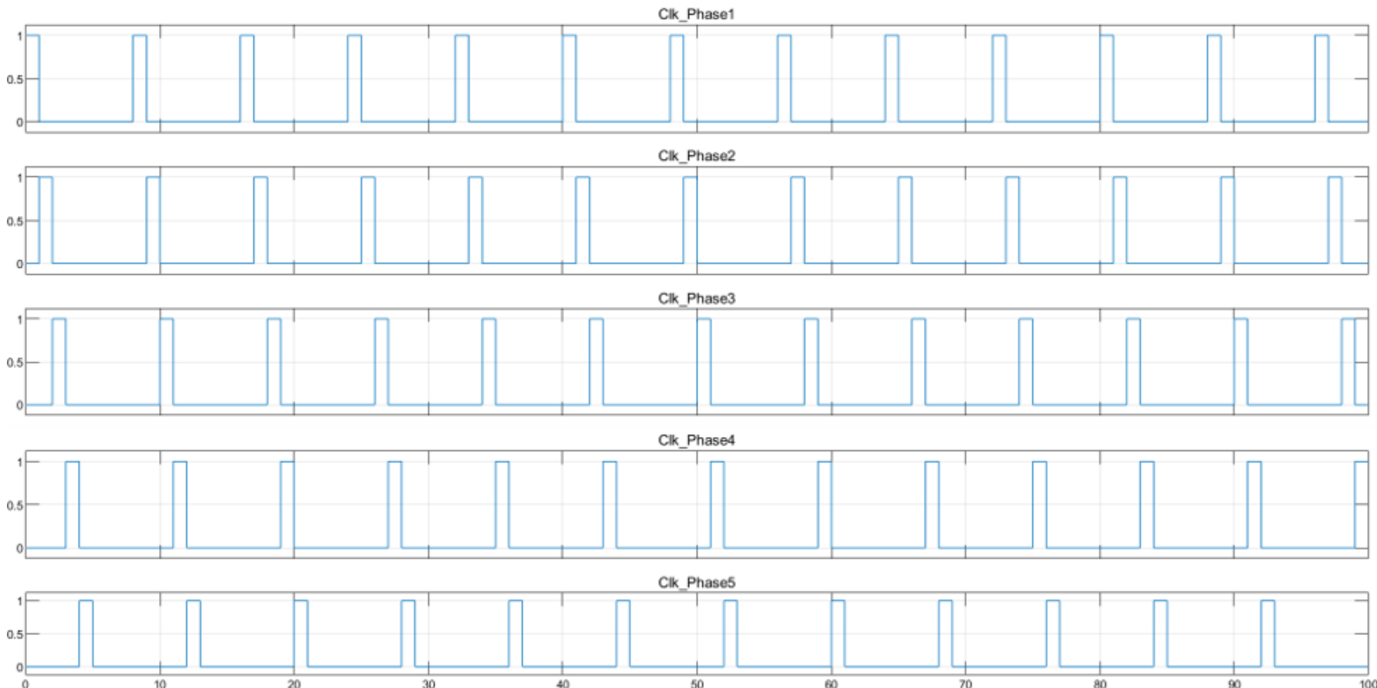
Design a Model With Clocks That Run at Same Rate, But Out of Phase

Using trigger as clock for triggered subsystem, you can model a design that has clocks which operate at same rate, but out of phase. For example, load and open the `PhasedClocksUsingTriggeredSubsystem` model.

```
load_system("PhasedClocksUsingTriggeredSubsystem");
set_param("PhasedClocksUsingTriggeredSubsystem",'SimulationCommand','Update')
open_system("PhasedClocksUsingTriggeredSubsystem");
```

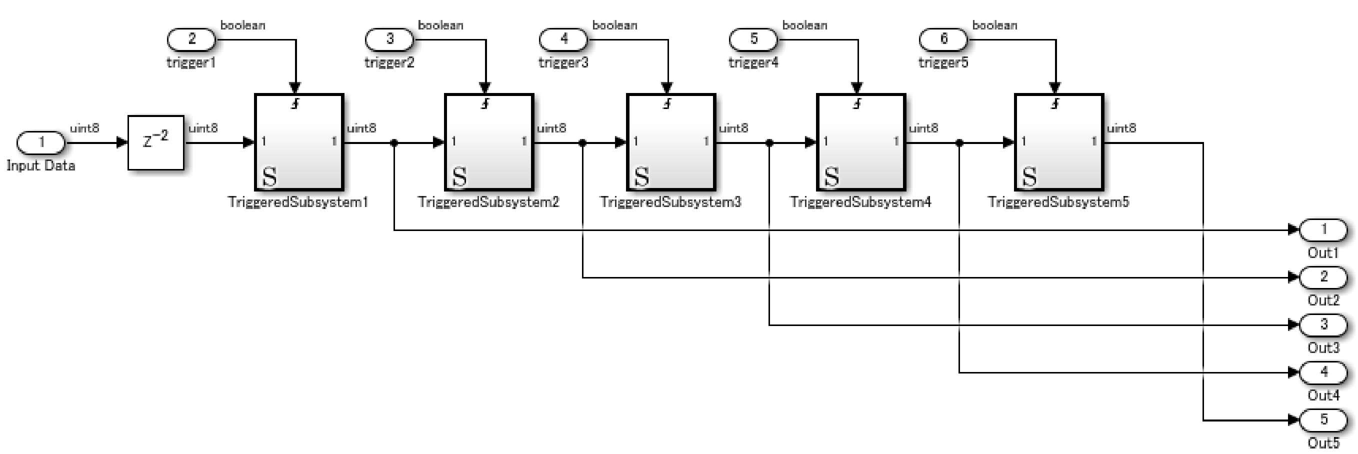


The model contains the `Clock Generation` subsystem which outputs five clock signals that run at same rate, but are out of phase. Run the simulation for the model and view waveforms of the phased clocks.



These clock signals are connected to trigger port of the triggered subsystems placed in the `Phased_Clock_DUT` subsystem. Open `Phased_Clock_DUT` subsystem.

```
open_system("PhasedClocksUsingTriggeredSubsystem/Phased_Clock_DUT");
```



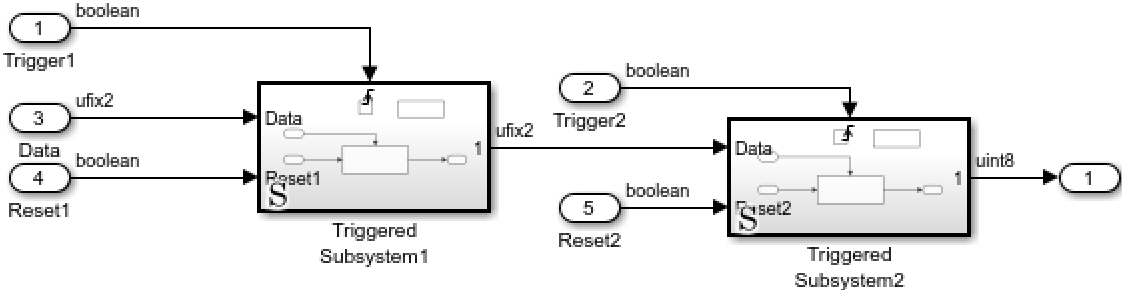
Enable **Use trigger signal as clock** option and generate HDL code for `Phased_Clock_DUT` subsystem. In the generated code, trigger port of the triggered subsystem acts as a clock which is driven by the corresponding clock signal.

Generate Multiple Clocks and Resets Using Triggered and Resettable Subsystems

This example shows how to model multiple clock and reset signals using triggered and resettable subsystems. Using trigger as clock functionality, you can model multiple clock signals from multiple triggered subsystems. By placing the resettable subsystem inside each triggered subsystem, you can model multiple reset signals in your Simulink model.

The model `AsynchronousClockandResetUsingTriggerAsClock` has a `DUT` subsystem which contains a two Triggered Subsystem. Load and open the `AsynchronousClockandResetUsingTriggerAsClock` model by running these commands:

```
load_system("AsynchronousClockandResetUsingTriggerAsClock");
set_param("AsynchronousClockandResetUsingTriggerAsClock",'SimulationCommand','Update')
open_system("AsynchronousClockandResetUsingTriggerAsClock/DUT");
```



A resettable subsystem is placed within the triggered subsystem. Using resettable subsystem, you can model a reset port for your model. Minimize the global reset option so that model has reset ports from resettable subsystems. To minimize global reset option, enable **Minimize global resets** in the **HDL Code Generation > Global Settings > Ports** tab of the configuration settings.

To use trigger signal as clock in your generated HDL code, enable **Use trigger signal as clock** option in the **HDL Code Generation > Global Settings > Ports** tab of the configuration settings. Then, generate the HDL code for a `DUT` subsystem using `makehdl` command or HDL Coder™ app:

```
makehdl("AsynchronousClockandResetUsingTriggerAsClock/DUT")
```

The generated code for a `DUT` subsystem is shown in code snippet below. HDL Coder generates two trigger ports for the `DUT` subsystem which are used as clock signal in the triggered subsystems and two reset ports from the resettable subsystems.

```
ENTITY DUT IS
  PORT( Trigger1                          :   IN    std_logic;
        Trigger2                          :   IN    std_logic;
        Data                              :   IN    std_logic_vector(1 DOWNTO 0);  -- ufix2
        Reset1                            :   IN    std_logic;
        Reset2                            :   IN    std_logic;
        Out2                              :   OUT   std_logic_vector(7 DOWNTO 0)  -- uint8
        );
END DUT;

...
...

BEGIN
    u_Triggered_Subsystem1 : Triggered_Subsystem1
      PORT MAP( Trigger => Trigger1,
```

```
                Data => Data,  -- ufix2
                Reset1 => Reset1,
                Out1 => Triggered_Subsystem1_out1  -- ufix2
                );

    u_Triggered_Subsystem2 : Triggered_Subsystem2
      PORT MAP( Trigger => Trigger2,
                Data => Triggered_Subsystem1_out1,  -- ufix2
                Reset2 => Reset2,
                Out1 => Triggered_Subsystem2_out1  -- uint8
                );

    Out2 <= Triggered_Subsystem2_out1;

END rtl;
```

Model Asynchronous and Synchronous Reset for a Simulink Model

You can generate synchronous and asynchronous external resets for your Simulink model by using nested triggered and resettable subsystems. You can use trigger-as-clock functionality to model the trigger port from a Triggered Subsystem as a clock and use Resettable Subsystem to model external reset port.

Simulink Model	Model Parameters	Type of External Reset	Generated HDL Code

Resettable subsystem is placed inside the triggered subsystem	**Use trigger signal as clock**: on	Synchronous	`always @(posedge Trigger)`
			`    begin : Delay_process`
			`        if (Reset_1 == 1'b1) be`
			`            Delay_out1 <= 2'b00`
			`        end`
			`...`
			
			In this generated HDL code, `Reset_1` is a synchronous reset.

Triggered subsystem is placed inside the resettable subsystem	**Use trigger signal as clock**: on	Asynchronous	`always @(posedge Trigger or pos`
			`    begin : Delay_process`
			`        if (Reset_1 == 1'b1) be`
			`            Delay_out1 <= 2'b00`
			`        end`
			`...`
			
			In this generated HDL code, `Reset_1` is an asynchronous reset.

<table>
<tr><th>Simulink Model</th><th>Model Parameters</th><th>Type of External Reset</th><th>Generated HDL Code</th></tr>
<tr><td>Model the subsystem hierarchy as Resettable Subsystem > Triggered Subsystem > Resettable Subsystem</td><td>Use trigger signal as clock: <code>on</code></td><td>Both synchronous and asynchronous</td><td><pre>always @(posedge Trigger or pos
    begin : Delay_process
        if (reset == 1'b1) begi
            Delay_out1 <= 2'b00
        end
        else begin
            if (Reset_1 == 1'b1
                Delay_out1 <= 2
            end
...</pre>In this generated HDL code, <code>reset</code> is an asynchronous reset and <code>Reset_1</code> is a synchronous reset.</td></tr>
</table>

See Also

“Using Triggered Subsystems for HDL Code Generation” on page 20-19

Related Examples

- “Model Trigger Signal As Clock in Triggered Subsystem” on page 20-20
- “Model Single Clock and Reset Signal Using Triggered and Resettable Subsystems” on page 20-22

Meet Timing Requirements Using Enable-Based Multicycle Path Constraints

In this section...

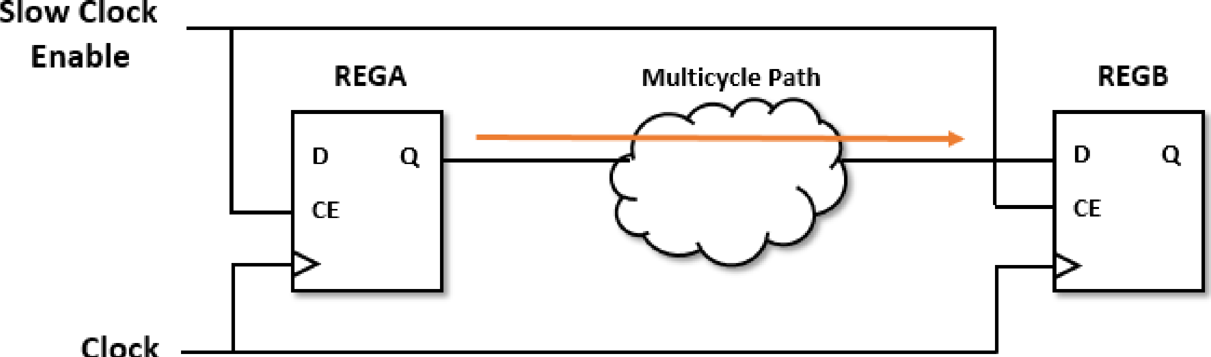
"How Enable-Based Multicycle Path Constraints Work" on page 20-32
"Specify Enable-Based Constraints" on page 20-33
"Benefits of Using Enable-Based Constraints" on page 20-34
"Modeling Guidelines" on page 20-34
"Multicycle Path Constraints for Various Synthesis Tools" on page 20-35
"Preserve Enable Signals in Timing Control Logic" on page 20-36
"Limitations" on page 20-37

If your Simulink model contains multiple sample rates or uses speed and area optimizations that insert pipeline registers, your design can have multicycle paths. Multicycle paths are data paths between two registers that operate at a sample rate slower than the FPGA clock rate and therefore take multiple clock cycles to complete their execution. To synchronize the clock rate to the sample rates of various paths in your design, you can use a single clock mode or a multiple clock mode. By default, HDL Coder uses a single clock mode that generates a single primary clock at the fastest sample rate and creates a timing controller entity to control the clock rate to the multicycle paths. The timing controller generates a set of clock enables with the required rate and phase information to control the sequential elements such as Delay blocks that operate at different sample rates.

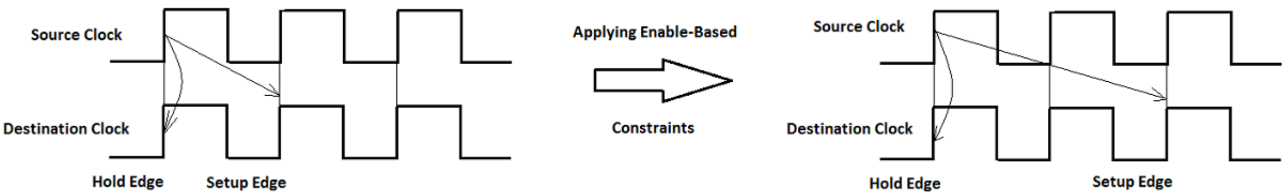
When you synthesize the generated HDL code, synthesis tools can fail to meet the timing requirements of multicycle paths. The timing failure occurs because synthesis tools cannot infer the various sample rates in your design from the generated HDL code. The synthesis tools assume that the registers in your design run at the primary clock rate and requires data to travel between the registers within one clock cycle. However, the multicycle paths are not required to complete their execution within one clock cycle and therefore cannot meet the timing requirements. To meet the timing requirements, specify generation of enable-based multicycle path constraints.

How Enable-Based Multicycle Path Constraints Work

Synthesis tools require that data propagates from a source register to a destination register within one clock cycle. Multicycle path constraints relax this timing requirement by allowing multiple clock cycles for data to propagate between the registers. The code generator uses the timing controller enable signals to create enable-based register groups, with registers in each group driven by the same clock enable. When you apply the enable-based constraints and generate HDL code, the code generator outputs a constraints file with the naming convention `dutname_constraints`. The file defines the timing requirements of multicycle paths and contains information about the setup and hold constraints that needs to be met.



This figure shows a multicycle path that takes a certain number of clock cycles, say *N*, for the data to propagate from REGA to REGB. By default, the synthesis tools define the setup edge at the next active clock edge and the hold edge at the same active clock edge with respect to the destination clock signal. For a multicycle path that takes N clock cycles, the constraints redefine the setup and hold edge to allow for the longer data propagation time.



For example, consider a multicycle path takes two clock cycles for data top propagate from the source to the destination register. This waveform shows how applying enable-based constraints redefines the setup and hold edges. This code snippet shows this setup and hold requirement in the constraints file that gets generated when you enable multicycle path constraints.

```
set_multicycle_path 2 -setup -from $REGA -to $REGB
set_multicycle_path 1 -hold  -from $REGA -to $REGB
```

Specify Enable-Based Constraints

Before you generate the enable-based constraints, you must:

- Preserve the multicycle paths in your design. Before you enable generation of multicycle path constraints, make sure that you disable optimizations such as clock rate pipelining and adaptive pipelining in those regions where you want to apply multicycle path constraints.

- Make sure that the region that operates at a slower clock rate is bounded by timing controller based clock enable signals operating at zero phase.
- Specify the **Synthesis Tool**. The format of the multicycle path constraints file that gets generated depends on the **Synthesis tool** that you specify. If you do not specify the synthesis tool and the **Generate EDA scripts** check box is selected, HDL Coder does not generate multicycle path constraints.
- Use the single clock mode. In the **HDL Code Generation** > **Global Settings** pane, set **Clock inputs** to `Single`.

You can specify generation of multicycle constraints in the Configuration Parameters dialog box, or in the HDL Workflow Advisor UI, or at the command line.

- In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Target and Optimizations** pane, select the **Enable-based constraints** check box.
- In the HDL Workflow Advisor, on the **HDL Code Generation** > **Set Code Generation Options** > **Set Optimization Options** task, select the **Enable-based constraints** check box.
- At the command line, use the `MulticyclePathConstraints` property with `hdlset_param` or `makehdl`.

Benefits of Using Enable-Based Constraints

If the synthesis tools identify the multicycle path constraints, you can:

- Realize higher clock rates and improve the timing of your design.
- Reduce the area footprint on the target FPGA device because multicycle path constraints do not introduce any pipeline registers.
- Reduce HDL code generation time because the code generator does not have to run many optimization settings.
- Reduce synthesis time since multicycle path constraints relax the timing requirements on the synthesis tool.
- Skip verification of your design after generating HDL code as the generated model with the constraints is identical to the original model.

When you use the enable-based constraints setting:

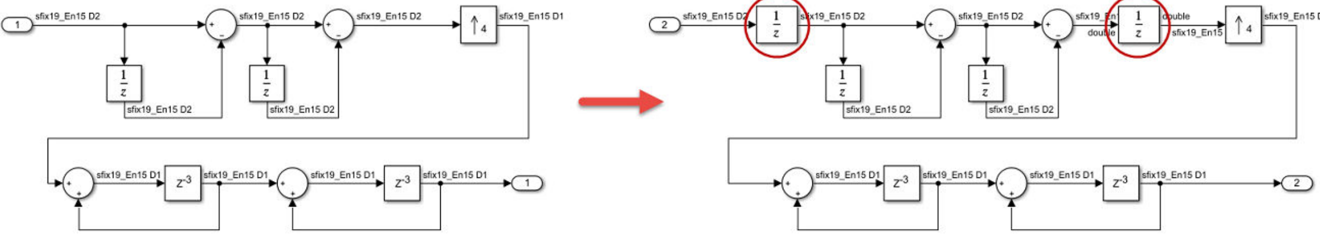
- The generated constraints are more robust to name changes in synthesis tools.
- HDL code generation becomes faster.
- The **Target workflow** can be `Generic ASIC/FPGA`, `IP Core Generation`, and `Simulink Real-Time FPGA I/O`.
- The constraint file format is supported with `Xilinx ISE`, `Xilinx Vivado`, and `Altera QUARTUS II`.

Modeling Guidelines

When you specify generation of enable-based constraints, use these modeling patterns in your design. If your model contains slow-rate regions that are not bounded by registers, then add delays at the same slow rate to the input and output of the slow-rate regions. For example, if you enter this command in the MATLAB Command Window, you see a multirate CIC Interpolation filter implemented in single clock mode:

```
openExample('hdlcoder_clockdemo');
```

This figure shows how to bound the input and output of the slow-rate region annotated by the slow sample time D2 in the model with Unit Delay blocks so that the enable-based constraints can identify the slow-rate path.



Note You can use Rate Transition blocks to introduce the input and output registers but make sure that the registers are slow rate and have zero phase.

Multicycle Path Constraints for Various Synthesis Tools

Enable-based multicycle path constraints have various file formats that depend on the **Synthesis tool** that you specify.

Altera Quartus II

HDL Coder generates the constraints in the form of an SDC file. This code snippet shows the SDC file generated for Altera Quartus II.

```
# Multicycle constraints for clock enable: DUT_tc.u1_d4_o0
set enbreg [get_registers *u_DUT_tc|phase_0]
set_multicycle_path 4 -to [get_fanouts $enbreg -through [get_pins -hier *|ena]] -end -setup
set_multicycle_path 3 -to [get_fanouts $enbreg -through [get_pins -hier *|ena]] -end -hold
```

Xilinx Vivado

HDL Coder generates the constraints in the form of an XDC file. This code snippet shows the XDC file generated for Xilinx Vivado.

```
# Multicycle constraints for clock enable: DUT_tc.u1_d4_o0
set enbregcell [get_cells -hier -filter {mcp_info=="DUT_tc.u1_d4_o0"}]
set enbregnet [get_nets -of_objects [get_pins -of_objects $enbregcell -filter {DIRECTION == OUT}]
set reglist [get_cells -of [filter [all_fanout -flat -endpoints_only $enbregnet] IS_ENABLE]]
set_multicycle_path 4 -setup -from $reglist -to $reglist -quiet
set_multicycle_path 3 -hold  -from $reglist -to $reglist -quiet
```

The multicycle path constraints form enable-based register groups by querying the synthesis netlist for the ATTRIBUTE keyword. When you run any of the supported target workflows, this code snippet shows this keyword in the synthesis netlist.

```
...
ATTRIBUTE mcp_info: string
```

```
ATTRIBUTE mcp_info OF phase_0 : SIGNAL IS "DUT_tc.u1_d4_o0";
...
```

The constraints file that is generated for Xilinx Vivado is more robust than pattern matching on module or signal names.

Xilinx ISE

HDL Coder generates the constraints in the form of a `UCF` file. This code snippet shows the `UCF` file generated for a model that has one slow-rate region controlled by a clock enable signal and has a target frequency of `300MHz`. The multicycle path constraints depend on the **Target Frequency** that you specify.

```
# Multicycle constraints for clock enable: DUT_tc.u1_d4_o0
NET "*u_DUT_tc/phase_0" TNM_NET = FFS "TN_u_DUT_tc_phase_0";
TIMESPEC "TS_u_DUT_tc_phase_0" = FROM "TN_u_DUT_tc_phase_0" TO "TN_u_DUT_tc_phase_0" TS_FPGA_CLK/
```

The clock constraints that are generated when you run the `Generic ASIC/FPGA`, or the `Simulink Real-Time FPGA I/O` workflow with Xilinx ISE are shown in this code.

```
# Timing Specification Constraints

NET "clk" TNM_NET = "TN_clk";
TIMESPEC "TS_FPGA_CLK" = PERIOD "TN_clk" 300 MHz;
```

To use the multicycle path constraints when you generate HDL code by using the `makehdl` function, make sure that you add a `TS_FPGA_CLK` constraint to the `UCF` file.

Preserve Enable Signals in Timing Control Logic

When you apply enable-based multicycle path (MCP) constraints to relax the timing constraints on multicycle data paths, HDL Coder generates MCP with timing controller logic. If you source enable-based constraints to the synthesis tool, the synthesis tool optimizes this timing controller logic to convert it to the single clock domain design. The MCP constraints are not applied in the generated HDL code. To preserve this timing control logic in the synthesis tool, HDL Coder adds an attribute `direct_enable` to all of the enable signals in your generated timing controller HDL code.

This generated VHDL code shows the `direct_enable` attribute is added to the enable signal in timing controller logic:

```
...
ATTRIBUTE direct_enable OF enb_1_1_1: SIGNAL IS "yes";
...
```

This generated Verilog code shows the `direct_enable` attribute is added to the enable signal in timing controller logic:

```
...
(* direct_enable = "yes" *) output enb_1_1_1;
...
```

Due to the `direct_enable` attribute, the MCP constraints are applied to your design when you source MCP constraints in the synthesis tool.

Limitations

- The multicycle path constraints file is not supported with the `FPGA-in-the-Loop` workflow.
- If the slow-rate region is not bounded by registers, multicycle path constraints require that you to add two Delay blocks at the slow rate, which increases the latency of your design.
- The code generator does not add constraints on paths between registers that have a nonzero phase value for the timing controller-based enable signals. For the code generator to add constraints, use registers that derive from phase `0` clock enable signals, such as Delay blocks.
- The generated multicycle constraints can be less effective if you apply the constraints to regions that have optimizations such as clock-rate pipelining and adaptive pipelining enabled. With clock-rate pipelining, the registers operate at the faster clock rate and might not retain the slow-rate registers in your design.
- HDL Coder does not generate multicycle path constraints for single-rate models.
- If you use the multiple clock mode, the code generator does not output the multicycle path constraints file.

See Also

Related Examples

- "Use Multicycle Path Constraints to Meet Timing for Slow Paths" on page 20-38

More About

- "Clock-Rate Pipelining" on page 21-148
- "Adaptive Pipelining" on page 21-181

Use Multicycle Path Constraints to Meet Timing for Slow Paths

This example shows how to apply multicycle path constraints in your design to meet timing requirements. Using multicycle path constraints can save area and reduce synthesis run times. For more information, see “Meet Timing Requirements Using Enable-Based Multicycle Path Constraints” on page 20-32.

Algorithms modeled in Simulink® for HDL code generation can have multiple sample rates. These multiple rates can be part of the Simulink model or can be introduced with HDL Coder™ options, such as oversampling. When you enable oversampling, the generated HDL code can run on the FPGA at a faster clock rate. This faster rate allows additional optimizations to take effect. To learn more about specifying an oversampling value, see “Specifying the Oversampling Value” on page 20-9.

When HDL Coder is configured to use a single clock, it generates a timing controller to control clocked elements, such as delays, at different sample rates with clock enables. These clock enables are synchronous to the single clock and toggle at rates that are multiple times slower than the base clock. The data paths between slow-clocked element pairs are called *multicycle paths* because they allow data to take multiple clock cycles to travel. However, synthesis tools cannot infer this acceptable delay directly from the HDL code. The tools assume that data changes every cycle and must travel from one register to the next within one clock cycle. Synthesis tools have to take more effort to meet the excessive timing requirement and therefore can fail timing. By declaring a set of data paths as multicycle paths and providing the actual timing of these paths to the downstream synthesis tool, HDL Coder can simplify and accelerate the process of meeting the desired timing constraints for a design.

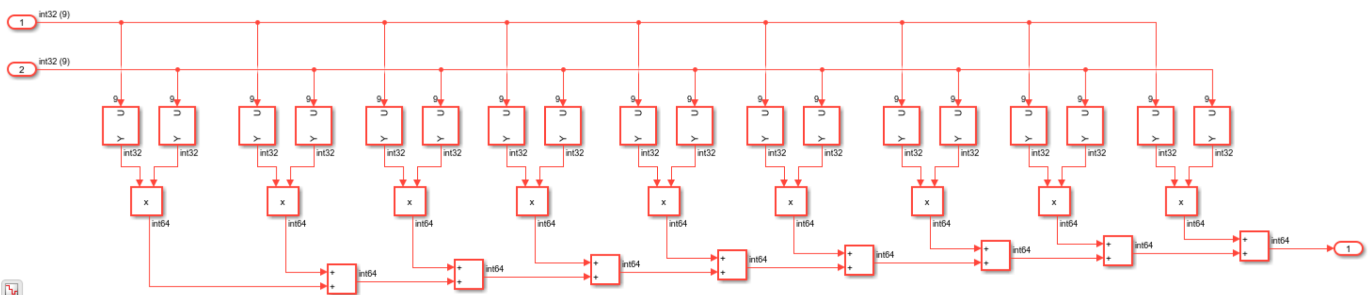
This example shows how to generate multicycle path constraints in HDL Coder so that you can specify timing requirements and enable efficient timing analysis in the synthesis tool.

Apply Multicycle Constraints Using HDL Coder

This example uses Xilinx® Vivado® 2020.2 post place and routing static timing analysis results for a Zynq® UltraScale+(TM) device (xazu11eg-ffvf1517-1-i) to show the impact of enabled multicycle path constraints. HDL Coder generates constraint files in XDC format for Xilinx Vivado, UCF format for Xilinx ISE, and SDC format for Altera Quartus II.

In this example, the `hdlcoder_multi_cycle_path_constraints` model contains a direct-form FIR filter with an adder chain in the critical path. The input data rate of this filter is 2MHz, but the goal of this design is to run as fast as possible so that it can be integrated with other IPs that require high frequency. First, set a 130MHz clock frequency without any timing optimization by setting the `TargetFrequency` to 130. Because you are modeling with actual hardware rates by setting the **Sample Time** of the Random Number block to `.5e-6`, or 2MHz, enable `TreatRatesAsHardwareRates` to allow HDL Coder to calculate an oversampling value needed to meet your target frequency.

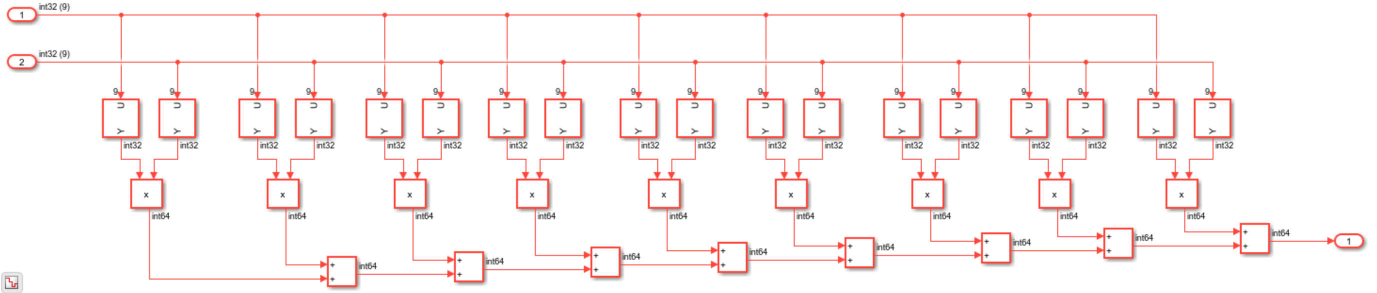
```
load_system('hdlcoder_multi_cycle_path_constraints');
open_system('hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product');
hdlset_param('hdlcoder_multi_cycle_path_constraints','TargetFrequency',130);
hdlset_param('hdlcoder_multi_cycle_path_constraints','TreatRatesAsHardwareRates','on');
set_param('hdlcoder_multi_cycle_path_constraints','SimulationCommand','update');
```



Generate HDL code with these settings and inspect the generated model. The generated model is identical to the original model.

```
makehdl('hdlcoder_multi_cycle_path_constraints/Subsystem');
load_system('gm_hdlcoder_multi_cycle_path_constraints');
set_param('gm_hdlcoder_multi_cycle_path_constraints','SimulationCommand','update');
open_system('gm_hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_multi_cycle_path_constraints')">hd
### Generating HDL for <a href="matlab:open_system('hdlcoder_multi_cycle_path_constraints/Subsyst
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_multi_c
### Running HDL checks on the model 'hdlcoder_multi_cycle_path_constraints'.
### Begin compilation of the model 'hdlcoder_multi_cycle_path_constraints'...
### Working on the model 'hdlcoder_multi_cycle_path_constraints'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_multi_cycle_path_constraints
### Begin model generation 'gm_hdlcoder_multi_cycle_path_constraints'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_multi_cycle_pat
### Begin VHDL Code Generation for 'hdlcoder_multi_cycle_path_constraints'.
### MESSAGE: The design requires 65 times faster clock with respect to the base rate = 5e-07.
### Begin VHDL Code Generation for 'Subsystem_tc'.
### Working on Subsystem_tc as hdl_prj\hdlsrc\hdlcoder_multi_cycle_path_constraints\Subsystem_tc
### Code Generation for 'Subsystem_tc' completed.
### Working on hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product as hdl_prj\hdlsrc\hdlc
### Working on hdlcoder_multi_cycle_path_constraints/Subsystem as hdl_prj\hdlsrc\hdlcoder_multi_c
### Generating package file hdl_prj\hdlsrc\hdlcoder_multi_cycle_path_constraints\Subsystem_pkg.vh
### Code Generation for 'hdlcoder_multi_cycle_path_constraints' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Writing Vivado multicycle constraints XDC file <a href="matlab:edit('hdl_prj\hdlsrc\hdlcoder_
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_multi_cycle_path_constraints' complete with 0 errors, 0 warnings, and
### HDL code generation complete.
```

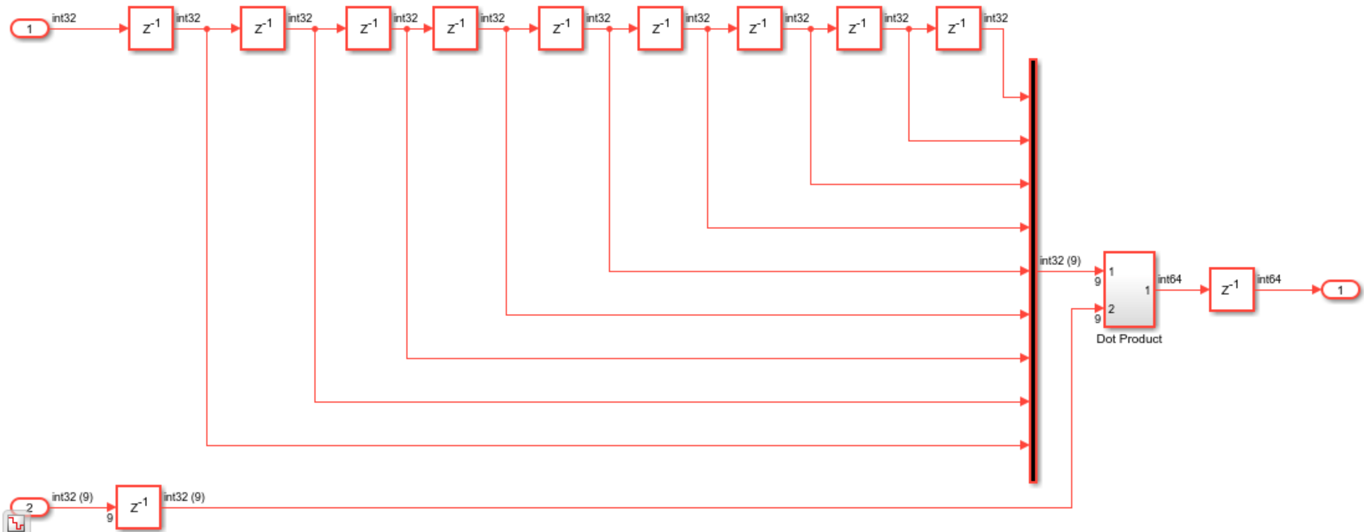


The generated HDL code fails to meet the 130MHz timing requirement for the clock. The timing requirement is 7.692 ns, or 1/130MHz, and a negative slack indicates a timing violation of this requirement.

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (VIOLATED) :        -1.979ns  (required time - arrival time)
  Source:                 Delay6_out1_reg[8]/C
                            (rising edge-triggered cell FDCE clocked by MWCLK  {rise@0.000ns fall@3.846ns period=7.692ns}
  Destination:            Delay1_out1_reg[61]/D
                            (rising edge-triggered cell FDCE clocked by MWCLK  {rise@0.000ns fall@3.846ns period=7.692ns}
  Path Group:             MWCLK
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            7.692ns  (MWCLK rise@7.692ns - MWCLK rise@0.000ns)
  Data Path Delay:        9.686ns  (logic 7.297ns (75.333%)  route 2.389ns (24.667%))
```

To meet timing requirements, use multicycle path constraints. Check the original model.

```
open_system('hdlcoder_multi_cycle_path_constraints/Subsystem');
```



The Dot Product subsystem is surrounded by delays that run at the desired 2MHz data rate. The design can tolerate multiple clock cycles for the data to propagate through it because of the oversampling value set by enabling `TreatRatesAsHardwareRates`. HDL Coder requires multicycle regions to be surrounded by slow-clocked elements, such as these delays, so that the constraints can define the paths among them as multicycle paths. Enable `MulticyclePathConstraints` and HDL Coder generates an additional file during HDL code generation.

```
hdlset_param('hdlcoder_multi_cycle_path_constraints', 'MulticyclePathConstraints', 'on');
```

Increase the target frequency to 300MHz to increase the clock rate.

```
hdlset_param('hdlcoder_multi_cycle_path_constraints', 'TargetFrequency', 300);
```

Generate HDL code and the multicycle path constraints.

```
makehdl('hdlcoder_multi_cycle_path_constraints/Subsystem');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_multi_cycle_path_constraints')">hc
### Generating HDL for <a href="matlab:open_system('hdlcoder_multi_cycle_path_constraints/Subsys
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_multi_
```

```
### Running HDL checks on the model 'hdlcoder_multi_cycle_path_constraints'.
### Begin compilation of the model 'hdlcoder_multi_cycle_path_constraints'...
### Working on the model 'hdlcoder_multi_cycle_path_constraints'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_multi_cycle_path_constraints
### Begin model generation 'gm_hdlcoder_multi_cycle_path_constraints'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_multi_cycle_pa
### Begin VHDL Code Generation for 'hdlcoder_multi_cycle_path_constraints'.
### MESSAGE: The design requires 150 times faster clock with respect to the base rate = 5e-07.
### Begin VHDL Code Generation for 'Subsystem_tc'.
### Working on Subsystem_tc as hdl_prj\hdlsrc\hdlcoder_multi_cycle_path_constraints\Subsystem_tc
### Code Generation for 'Subsystem_tc' completed.
### Working on hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product as hdl_prj\hdlsrc\hdl
### Working on hdlcoder_multi_cycle_path_constraints/Subsystem as hdl_prj\hdlsrc\hdlcoder_multi_
### Generating package file hdl_prj\hdlsrc\hdlcoder_multi_cycle_path_constraints\Subsystem_pkg.v
### Code Generation for 'hdlcoder_multi_cycle_path_constraints' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Writing Vivado multicycle constraints XDC file <a href="matlab:edit('hdl_prj\hdlsrc\hdlcoder_
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/t
### HDL check for 'hdlcoder_multi_cycle_path_constraints' complete with 0 errors, 0 warnings, an
### HDL code generation complete.
```

Inspect the generated constraint XDC file.

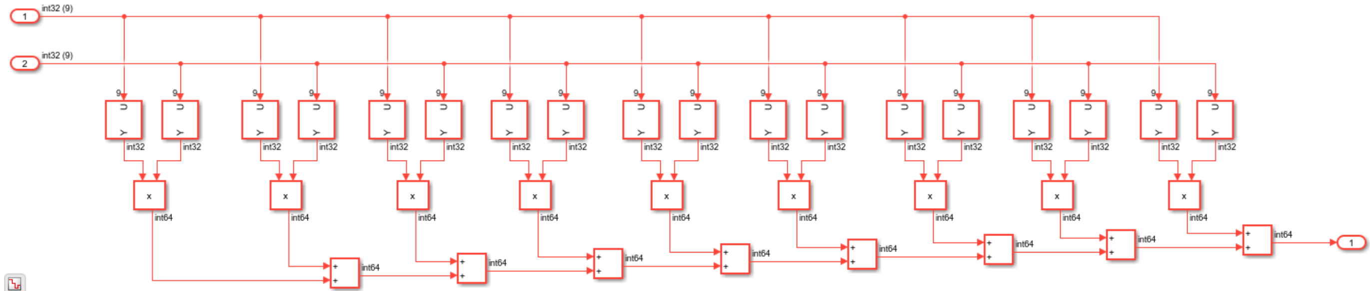
```
dbtype('hdl_prj/hdlsrc/hdlcoder_multi_cycle_path_constraints/Subsystem_constraints.xdc');
```

```
1     # Multicycle constraints for clock enable: Subsystem_tc.u1_d150_o0
2     set enbregcell [get_cells -hier -filter {mcp_info=="Subsystem_tc.u1_d150_o0"}]
3     set enbregnet [get_nets -of_objects [get_pins -of_objects $enbregcell -filter {DIRECTION ==
4     set reglist1 [get_cells -of [filter [all_fanout -flat -endpoints_only $enbregnet] IS_ENABL
5     set_multicycle_path 150 -setup -from $reglist1 -to $reglist1 -quiet
6     set_multicycle_path 149 -hold -from $reglist1 -to $reglist1 -quiet
7
8     # Multicycle constraints for clock enable: Subsystem_tc.u1_d150_o1
9     set enbregcell [get_cells -hier -filter {mcp_info=="Subsystem_tc.u1_d150_o1"}]
10    set enbregnet [get_nets -of_objects [get_pins -of_objects $enbregcell -filter {DIRECTION ==
11    set reglist2 [get_cells -of [filter [all_fanout -flat -endpoints_only $enbregnet] IS_ENABL
12    set_multicycle_path 150 -setup -from $reglist2 -to $reglist2 -quiet
13    set_multicycle_path 149 -hold -from $reglist2 -to $reglist2 -quiet
14
15    # Multicycle constraints from clock enable: Subsystem_tc.u1_d150_o1 to clock enable: Subsys
16    set_multicycle_path 149 -setup -from $reglist2 -to $reglist1 -quiet
17    set_multicycle_path 149 -hold -from $reglist2 -to $reglist1 -quiet
18
19    # Multicycle constraints from clock enable: Subsystem_tc.u1_d150_o0 to clock enable: Subsys
20    set_multicycle_path 151 -setup -from $reglist1 -to $reglist2 -quiet
21    set_multicycle_path 149 -hold -from $reglist1 -to $reglist2 -quiet
22
```

These constraints first find flip flops driven by the 2MHz clock enable signal. Then, they define the paths among these flip flops to allow up to 150 cycles for data to propagate.

Inspect the generated model.

```
open_system('gm_hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product');
set_param('gm_hdlcoder_multi_cycle_path_constraints', 'SimulationCommand', 'update');
```



The generated model and HDL code are identical to the previous results because generating enabled based multicycle path constraints does not alter the HDL architecture. However, the multicycle path constraints enable the design to run at 300MHz on the FPGA.

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             0.612ns  (required time - arrival time)
  Source:                 u_Subsystem_tc/phase_0_reg/C
                            (rising edge-triggered cell FDCE clocked by MWCLK  {rise@0.000ns fall@1.666ns period=3.333ns}
  Destination:            Delay6_out1_reg[4]/CE
                            (rising edge-triggered cell FDCE clocked by MWCLK  {rise@0.000ns fall@1.666ns period=3.333ns}
  Path Group:             MWCLK
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            3.333ns  (MWCLK rise@3.333ns - MWCLK rise@0.000ns)
  Data Path Delay:        2.408ns  (logic 0.399ns (16.570%)  route 2.009ns (83.430%))
```

Additional Information About Multicycle Path Constraints

Multicycle path constraints are required for synthesis tools to understand timing requirements. This information is extracted from the Simulink model since it cannot be inferred from the generated HDL code. Multicycle path constraints identify paths between clocked elements driven by the same clock enable. It can fail to meet timing requirements in certain cases. For example, a data path is not recognized as a multicycle path, if it is not gated with both input and output delays or is between two delays of different rates. Therefore, if you want to use multicycle path constraints for certain parts in your design, it is important to retain the multicycle paths in that region from being altered by optimizations introducing pipelines, such as input and output pipelining, clock rate pipelining, adaptive pipelining resource sharing, streaming, pipelined math operations, e.g. Newton-Raphson method for sqrt or recip, Cordic algorithm for trigonometric functions, and floating-point IP mapping.

Optimization

- “Speed and Area Optimizations in HDL Coder” on page 21-3
- “Automatic Iterative Optimization” on page 21-7
- “Generated Model and Validation Model” on page 21-10
- “Locate Numeric Differences After Speed Optimization” on page 21-13
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder” on page 21-18
- “Optimization with Constrained Overclocking” on page 21-23
- “Resolve Numeric Mismatch with Delay Balancing” on page 21-25
- “Resolve Simulation Mismatch When Pipelining with a Feedback Loop Outside the DUT” on page 21-31
- “Streaming” on page 21-42
- “Resource Sharing” on page 21-45
- “Streaming: Area Optimization” on page 21-49
- “Resource Sharing for Area Optimization” on page 21-54
- “Single-Rate Resource Sharing Architecture” on page 21-65
- “Improve Resource Sharing with Design Modifications” on page 21-69
- “Improve Resource Sharing with Clone Detection and Replacement” on page 21-75
- “Delay Balancing” on page 21-81
- “Use Delay Absorption While Modeling with Latency” on page 21-86
- “Delay Balancing and Validation Model Workflow in HDL Coder” on page 21-94
- “Control the Scope of Delay Balancing” on page 21-102
- “Delay Balancing on Multirate Designs” on page 21-107
- “Find Feedback Loops” on page 21-115
- “Hierarchy Flattening” on page 21-117
- “Apply RAM Mapping to Optimize Area” on page 21-120
- “RAM Mapping with the MATLAB Function Block” on page 21-125
- “Distributed Pipelining” on page 21-130
- “Distributed Pipelining Using Synthesis Timing Estimates” on page 21-136
- “Distributed Pipelining: Speed Optimization” on page 21-139
- “Constrained Output Pipelining” on page 21-146
- “Clock-Rate Pipelining” on page 21-148
- “Increase Clock Frequency Using Clock-Rate Pipelining” on page 21-153
- “Iteratively Maximize Clock Frequency by Using Speed Optimizations” on page 21-167
- “Adaptive Pipelining” on page 21-181
- “Design Patterns That Require Adaptive Pipelining” on page 21-188
- “Critical Path Estimation Without Running Synthesis” on page 21-192

- “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 21-205
- “Subsystem Optimizations for Filters” on page 21-214
- “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 21-224
- “Optimize Unconnected Ports in HDL Code for Simulink Models” on page 21-246

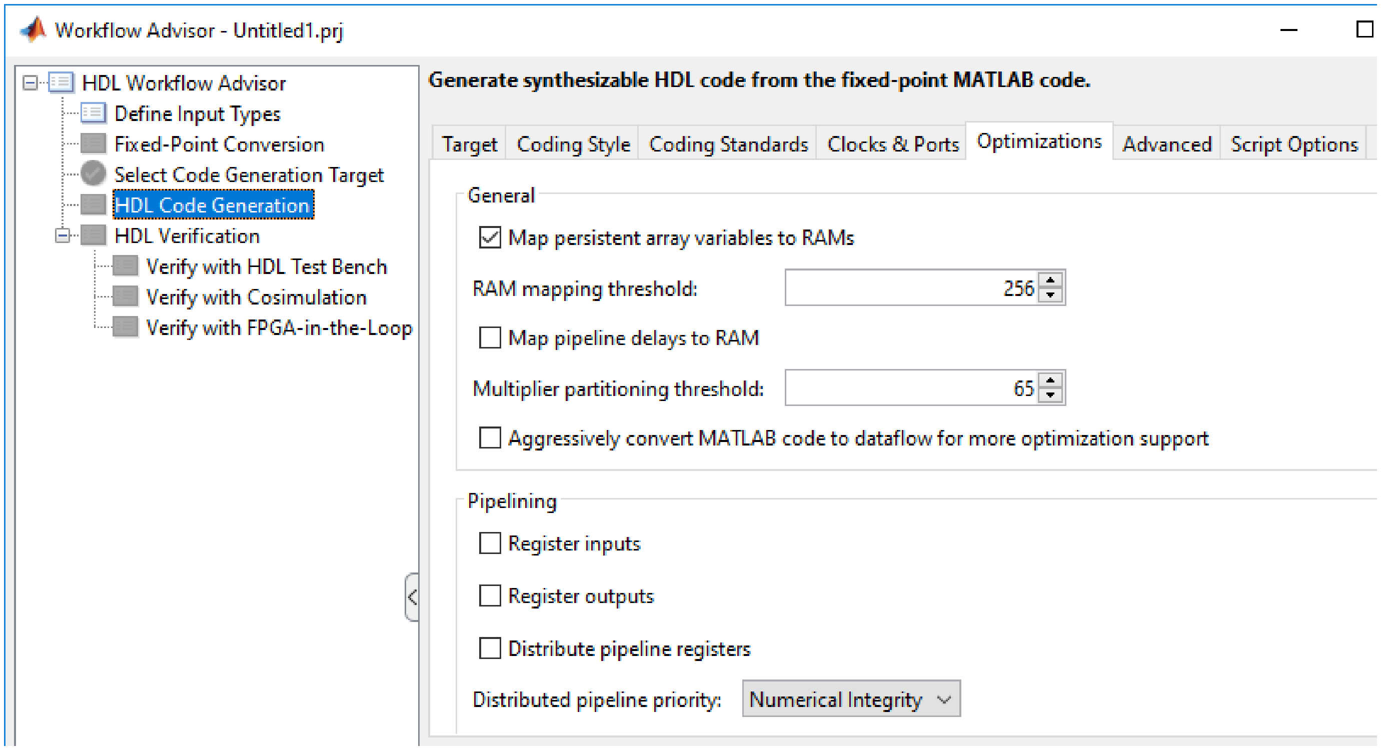
Speed and Area Optimizations in HDL Coder

Use area and speed optimizations in HDL Coder to save resources and improve the timing of your design on the target FPGA device. The optimizations do not change the functional behavior of your algorithm but can optimize certain resources in your design, introduce latency, or cause difference in sample rates.

You can initially generate HDL code and synthesize your design on your FPGA platform without enabling optimizations. If the design does not meet the timing requirements, you can enable the optimizations and rerun the workflow until your design meets the area and speed requirements. See "Basic HDL Code Generation Workflow".

Optimizations in MATLAB HDL Code Generation

To enable optimizations on your MATLAB code, open the Workflow Advisor from MATLAB. In the Advisor, on the **HDL Code Generation** task, enable the settings in the **Optimization** tab.



Optimizations in Simulink HDL Code Generation

You can enable optimizations at the model level and at the block level. Specify model-level optimizations:

- In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Optimization** pane.

- At the command line by using the makehdl or hdlset_param function to set the property value.
- In the Simulink HDL Workflow Advisor, on the **Set HDL Options**task, the **HDL Code Generation Settings** button opens the HDL code generation settings in the Configuration Parameters dialog box. You can then navigate to the **HDL Code Generation > Optimization** pane.

Subsystems in your model inherit the model-level optimization settings. You can change the subsystem level settings in the HDL Block Properties dialog box for the subsystems or by using the hdlset_param function. You can also specify certain additional settings for certain blocks in your model such as adding pipelines at the input and output. This table illustrates various optimizations that are available at the block level and model level.

Optimization	Model Level?	Subsystem Level?	Comments
Delay balancing	Yes	Yes	–
RAM mapping	Yes	No	–
Adaptive pipelining	Yes	Yes	–
Clock rate pipelining	Yes	Yes	–
Distributed pipelining	Yes	Yes	–
Resource sharing	Yes	Yes	At the model level, you specify the type of resources you want to share such as adders and multipliers. At the block level, you specify the **SharingFactor**.
Streaming	No	Yes	–

To see the effect of the optimizations:

- You can generate an optimization report with the HDL code. To learn how to enable this report, see “Create and Use Code Generation Reports” on page 23-2.
- Open the generated model or generate the validation model. The generated model is a behavioral model of the HDL code that shows the effect of block implementations and optimizations that you enabled. To verify the numerics of the generated model with the original model, you can generate a validation model. See “Generated Model and Validation Model” on page 21-10.

Tip To effectively use optimizations, change the sample time setting for Constant blocks from Inf to -1.

General Optimizations

Your model can have design delays and pipeline delays. Design delays are delays that you manually add to your model. Pipeline delays are delays that are introduced by pipelining settings specified on the blocks, block implementations such as Newton-Raphson method, native floating-point operators, or speed optimizations. You see these delays in the generated HDL code, generated model, and validation model.

General optimization parameters includes:

- RAM mapping: Use RAM mapping parameters to map large delays, persistent variables in MATLAB code, and pipeline delays to RAM based on a threshold bit width. See also “Apply RAM

Mapping to Optimize Area” on page 21-120 and Map pipeline delays to RAM and RAM mapping threshold.

- Delay balancing: Enabled by default, this optimization balances pipeline delays by inserting matching delays in parallel paths. The optimization matches numerics of the generated model with the original model. You see the effect of this optimization in the **Delay Balancing** section of the optimization report. See “Delay Balancing” on page 21-81.

Speed Optimizations

Speed optimizations improve the timing of your design on the target FPGA by optimizing the critical path. To identify the critical path, you can run the `Generic ASIC/FPGA` workflow for your FPGA device and then annotate the critical path or use the timing reports.

To identify the critical path more quickly and speed up the iterative process of finding and optimizing the critical path, use critical path estimation. You do not have to run synthesis or generate HDL code. Critical path estimation uses static timing analysis with timing data from target-specific timing databases. You see the effect of this optimization in the **Critical Path Estimation** section of the optimization report. See “Critical Path Estimation Without Running Synthesis” on page 21-192.

Speed optimizations include:

- Clock rate pipelining: A Simulink optimization that is enabled by default, and runs pipeline registers at a faster clock rate when you specify an **Oversampling factor** greater than one. Use clock-rate pipelining with hierarchy flattening to remove hierarchical boundaries in a subsystem, thereby improving retiming. See “Clock-Rate Pipelining” on page 21-148.
- Distributed pipelining: An optimization that retimes registers that are existing delays, or specified by using **InputPipeline** and **OutputPipeline** block settings. To preserve existing delays, disable the Allow design delay distribution setting. To more accurately reflect how components function on hardware to better distribute pipelines and increase clock speed for your target device, use synthesis timing estimates for distributed pipelining. See Use synthesis estimates for distributed pipelining. Enable distributed pipelining on the model for retiming registers across hierarchies. You see the effect of this optimization in the **Distributed Pipelining** section of the optimization report. See “Distributed Pipelining” on page 21-130.
- Adaptive pipelining: A Simulink optimization that inserts pipeline registers at input or output or both ports of certain blocks to create patterns that efficiently map blocks to DSP units on the target FPGA device. The optimization considers the target device, target frequency, multiplier word lengths, and the HDL Block Property settings. You see the effect of this optimization in the **Adaptive Pipelining** section of the optimization report. See “Adaptive Pipelining” on page 21-181.
- Loop Unrolling: A MATLAB optimization that unrolls a loop by instantiating multiple instances of the loop body in the generated code. You can also partially unroll a loop. See “Optimize MATLAB Loops” on page 8-29

Area Optimizations

Area optimizations reduce resource usage of your design. Optimizing your design for area can reduce the speed at which your design runs on the FPGA.

Area optimizations include:

- Resource Sharing: An optimization that identifies multiple functionally equivalent resources and replaces them with a single resource. At the model level, you specify resources you want to share

such as adders and multipliers. At the subsystem level, you specify a **SharingFactor** depending on the number of shareable resources in your design. By using the optimization with clock-rate pipelining, you can specify how to overclock the shared resources. See “Resource Sharing” on page 21-45

- Streaming: A Simulink optimization that splits a vector data path into multiple smaller vector data paths based on the **StreamingFactor** that you specify on the subsystems, thereby reducing hardware resource consumption. See “Streaming” on page 21-42.
- Loop Streaming: A MATLAB optimization that streams a loop by instantiating the loop body once and using that instance for each loop iteration. The code generator oversamples the loop body instance to keep the generated loop functionally equivalent to the original loop. See “Optimize MATLAB Loops” on page 8-29

See Also

`makehdl`

More About

- “Increase Clock Frequency Using Clock-Rate Pipelining” on page 21-153
- “Distributed Pipelining: Speed Optimization” on page 21-139
- “Resource Sharing for Area Optimization” on page 21-54
- “Streaming: Area Optimization” on page 21-49
- “Distributed Pipelining for Clock Speed Optimization” on page 8-15

Automatic Iterative Optimization

In this section...

"How Automatic Iterative Optimization Works" on page 21-7
"Automatic Iterative Optimization Output" on page 21-8
"Automatic Iterative Optimization Report" on page 21-8
"Automatic Iterative Optimization Synthesis Tool and Hardware" on page 21-9
"Limitations of Automatic Iterative Optimization" on page 21-9

Automatic iterative optimization enables you to optimize your clock frequency without specifying individual optimization options, such as input or output pipelining, distributed pipelining, or loop unrolling. Clock frequency is determined by the critical path of your design. Automatic iterative optimization improves clock frequency by inserting pipeline registers to break and shorten the critical path.

You can use `hdlcoder.optimizeDesign` to optimize your clock frequency in these ways:

- Best clock frequency: Specify the maximum number of iterations that you want HDL Coder to perform. HDL Coder iterates the optimization to minimize the critical path in your design.
- Target clock frequency: Specify a clock frequency target for your design and the maximum number of iterations you want HDL Coder to perform. HDL Coder iterates the optimization until it meets your target clock frequency or reaches the maximum number of iterations.

  HDL Coder can determine that your target clock frequency is not achievable because your target clock period is less than the latency of the largest atomic combinational group of logic in your design.

How Automatic Iterative Optimization Works

You can specify your clock frequency goal, the maximum number of iterations, and the timing strategy for the iterative optimization by using the `hdlcoder.OptimizationConfig` object properties. Depending on the timing strategy that you choose, synthesis (the default) or critical path estimation, the steps that HDL Coder performs for each iteration vary.

Running the automatic iterative optimization with synthesis can take a long time, depending on the complexity of your design. For example, when running the `hdl.optimizeDesign` function that has synthesis as the timing strategy, synthesis can occupy nearly 94% of the run time of this function. To help mitigate the runtime and still use synthesis, the `hdlcoder.optimizeDesign` function can regenerate code from a previous run or resume from an interrupted run.

To help mitigate the runtime most effectively, you can choose critical path estimation as your timing strategy. Critical path estimation provides an estimated critical path rather than the actual critical path determined by synthesis. When using critical path estimation, automatic iterative optimization avoids model generation and HDL code generation to help increase the speed of every iteration. Critical path estimation as the timing strategy is most effective for placement of delays and reducing the critical path when you generate a timing database for your target device by first using the `genhdltdb` function. For an example that runs the `hdlcoder.optimizeDesign` function using both strategies, see "Use Critical Path Estimation for Faster Optimization".

If you choose critical path estimation as your timing strategy, the automatic iterative optimization performs these steps:

1. Analyzes the logic in your design.
2. Estimates the critical path, and obtains timing analysis data by using a timing database. For more information, see “Critical Path Estimation Without Running Synthesis” on page 21-192.
3. Inserts pipeline registers to break the critical path.
4. Balances delays.
5. Saves iteration data in a new folder.

If you choose synthesis as your timing strategy, the automatic iterative optimization performs these steps:

1. Analyzes the logic in your design.
2. Generates code.
3. Uses the synthesis tool to analyze the generated code and obtains post-map timing analysis data.
4. Back-annotates the design by using the timing analysis data.
5. Inserts pipeline registers to break the critical path.
6. Balances delays.
7. Saves iteration data in a new folder.

When HDL Coder has met your clock frequency goal or it has reached the maximum number of iterations, it saves the generated code and iteration data in a new folder and generates a report that describes the final critical path.

Automatic Iterative Optimization Output

When HDL Coder exits the optimization loop, it saves the results of the final iteration in a folder, `hdlsrc/`*`your_model_name`*`/hdlexpl/Final-`*`timestamp`*.

The final iteration folder contains:

- The generated HDL code, in the `hdlsrc/`*`your_model_name`* folder.
- A data file, `cpGuidance.mat`, that you can use with your original model to regenerate code without rerunning the iterative optimization.
- The optimization report, `summary.html`.

Automatic Iterative Optimization Report

HDL Coder generates a report for the final optimization iteration and saves it in the final iteration folder, `hdlsrc/`*`your_model_name`*`/hdlexpl/Final-`*`timestamp`*.

The final optimization report, `summary.html`, contains:

- Summary Section containing:
  - Final critical path latency.
  - Critical path latency and elapsed time for each iteration.
- Diagnostic Section containing:
  - Reason for stopping at the final iteration.

- Model or block settings that can reduce the accuracy of the critical path analysis.

  If your model has these settings, remove them where possible, and rerun the `hdlcoder.optimizeDesign` function. Some optimizations, such as distributed pipelining and constrained output pipeline, change the placement of pipeline registers after the coder analyzes the critical path.
- Critical path description, which shows signals and components in the original model and generated model that are part of the critical path.

  You might see a message that a signal or component on the critical path cannot be traced back to the original model. HDL Coder might not be able to map its internal representation of your design back to the original design. Each optimization iteration changes the internal representation. The final representation can have a structure that is different from your original design.

Automatic Iterative Optimization Synthesis Tool and Hardware

If you are using synthesis as the timing strategy, your synthesis tool must be Xilinx ISE or Xilinx Vivado. Your target device must be a Xilinx FPGA.

Limitations of Automatic Iterative Optimization

- In the current release, automatic iterative optimization does not support Altera hardware if you are using synthesis as the timing strategy.
- Automatic iterative optimization is available from the command line only.
- When the timing strategy is set to synthesis, HDL Coder uses post-map timing information, which the synthesis tool generates before performing place and route. Post-map timing information is less accurate than timing information that the synthesis tool generates after place and route, but is faster to obtain.
- When the timing strategy is set to critical path estimation, HDL Coder uses a timing database to estimate the critical path. Critical path estimation yields a less accurate critical path result than synthesis does, but is the fastest option by a substantial amount (about 10x faster depending on your design) for iterative optimization.
- When using critical path estimation as the timing strategy, if there are blocks in the model that are not supported by critical path estimation, the results might be less optimal than using synthesis as the timing strategy. The blocks in the model that are not supported by critical path estimation appear as a message in the HDL Code Generation Check Report and the Resource Utilization Report, and as a link to a script to highlight the blocks in the MATLAB Command Window.
- You cannot compare the iterative optimization timing value for the critical path when using synthesis as the timing strategy to the result when using critical path estimation as the timing strategy. To compare the results, run synthesis on the final design after running `hdlcoder.optimizeDesign` with critical path estimation as the timing strategy.

See Also

`hdlcoder.optimizeDesign` | `hdlcoder.OptimizationConfig`

Generated Model and Validation Model

In this section...
“Generated Model” on page 21-10
“Validation Model” on page 21-11

HDL Coder enables you to view the effect of HDL optimizations and block settings in your generated model.

Generated Model

Before generating code, HDL Coder creates a behavioral model of the HDL code called the generated model. The generated model is a model created during HDL code generation that captures cycle-accurate and bit-true behavior of the generated code in area and timing optimizations during code generation. It shows latency and numeric differences between your Simulink design under test (DUT) and the generated HDL code. Delays that HDL Coder inserts are highlighted in the generated model in various colors. Different delays in code generation, the corresponding highlight color, and how the delays are named in the generated model are listed in this table.

Delays	Highlight Color	Naming Convention
• Block implementation • RAM mapping	Cyan	The block is highlighted in cyan. Delay blocks inside this block use the default name `Delay` and are not highlighted.
“Constrained Output Pipelining” on page 21-146	Green	Constrained output pipelining: `rd_n`
• “Distributed Pipelining” on page 21-130 • “InputPipeline” on page 19-14 and “OutputPipeline” on page 19-19 • “Delay Balancing” on page 21-81 • “Clock-Rate Pipelining” on page 21-148 • “Adaptive Pipelining” on page 21-181	Orange	• Distributed pipelining: `rd_n` • Input pipelining: `in_n_pipe_in_pipe` and output pipelining: `out_n_pipe_in_pipe` • Delay balancing: `delayMatch` • Clock-rate pipelining: `rd_n` • Adaptive pipelining: `HwModeRegister` at block inputs and `PipelineRegister` at the block output.

With timing and area optimizations, the generated model is substantially different from the original model. For example, there are additional integer delays next to blocks if you request optimizations and additional balanced delays to maintain the accuracy of the algorithm. If you request resource sharing or streaming optimization where the same operator is time multiplexed across multiple operations, there are additional rates in your model. When the original model does not have any optimizations, the generated model is the same as the original model. The generated model simulates the generated HDL code. The generated model is useful for visualizing changes in latency, rates, and dataflow from the original model.

While the original model and generated model have the same simulation settings, these models are not required to have the same code generation settings. For example, if you set **ResetType** to `none` for a delay block in your model, this setting might not appear in the generated model equivalent delay

block because it does not affect simulation. To see the delay block with **ResetType** set to `none` mapped to the generated HDL code, see "Navigate Between Simulink Model and HDL Code by Using Traceability" on page 23-12.

The generated model is used in Register Transfer Level (RTL) test bench generation. The input stimulus and output response are captured from the generated model instead of the original model because the generated model reflects the algorithm timing changes required for optimizations. If you disable model generation, you cannot generate a test bench in HDL Coder.

After code generation, the generated model is saved in the target folder. By default, the generated model prefix is `gm_`. For example, if your model name is *`myModel`*, your generated model name is `gm_`*`myModel`*.

Customize the Generated Model

To customize the prefix of the generated model name, use the `GeneratedModelNamePrefix` property with the functions `makehdl` or `hdlset_param`. See Prefix for generated model name.

You can also specify various options for the naming and layout of the generated model. See Prefix for generated model name.

Validation Model

Because the generated model is often substantially different from the original model, the coder can also create a validation model to compare the original model to the generated model. The validation model inserts delays at the outputs of the original model to compensate for latency differences and compares the outputs of the two models. When you simulate the validation model, numeric differences in the output data trigger an assertion.

Using the validation model, you can verify that the output of the optimized DUT is bit-true to the results produced by the original DUT.

A validation model contains:

- A generated model.
- An original model that has compensating delays inserted.
- Original inputs, routed to the original model and the generated model.
- Scopes for comparing and viewing the outputs of the original model and generated model.

Latency Differences

Some block architectures and optimizations introduce latency. For example, for the Reciprocal block, you can specify HDL block architectures that implement the Newton-Raphson method. The Newton-Raphson method is iterative, so block architectures that use it are multicycle and introduce latency at the block rate.

Similarly, the resource sharing area optimization time-multiplexes data over a shared hardware resource, which introduces local multirate and latency at the upsampled rate.

Numeric Differences

HDL block architectures can introduce numeric differences between the original and generated model. For example:

- An HDL block property, such as “InputPipeline” on page 19-14 specified on the block, or certain HDL architectures or optimizations, such as distributed pipelining, that move delays to the input of the block.
- The Newton-Raphson method is an approximation. If you select a Newton-Raphson block implementation, the generated model shows a change in numerics.
- HDL implementations for signal processing blocks, such as filters, can change numerics.

See also “Locate Numeric Differences After Speed Optimization” on page 21-13.

Generate a Validation Model

- In the Configuration Parameters dialog box, in the **HDL Code Generation** > **Global Settings** > **Model Generation** pane, select **Validation Model**.
- In the HDL Workflow Advisor, in the **HDL Code Generation** > **Generate RTL Code and Testbench** pane, enable **Generate validation model**.
- Use the `GenerateValidationModel` property with the functions `makehdl` or `hdlset_param`.

Customize the Validation Model

To customize the suffix of the generated validation name, use the `ValidationModelNameSuffix` property with the functions `makehdl` or `hdlset_param`. See Suffix for validation model name.

Restrictions

- To generate a validation model, you must generate HDL code for the DUT Subsystem. Model generation is not supported for generating code for the entire model instead of the DUT Subsystem.
- Make sure that the DUT subsystem has no unconnected output ports. See “Terminate Unconnected Block Outputs and Usage of Commenting Blocks” on page 18-27.

See Also

More About

- “Delay Balancing” on page 21-81

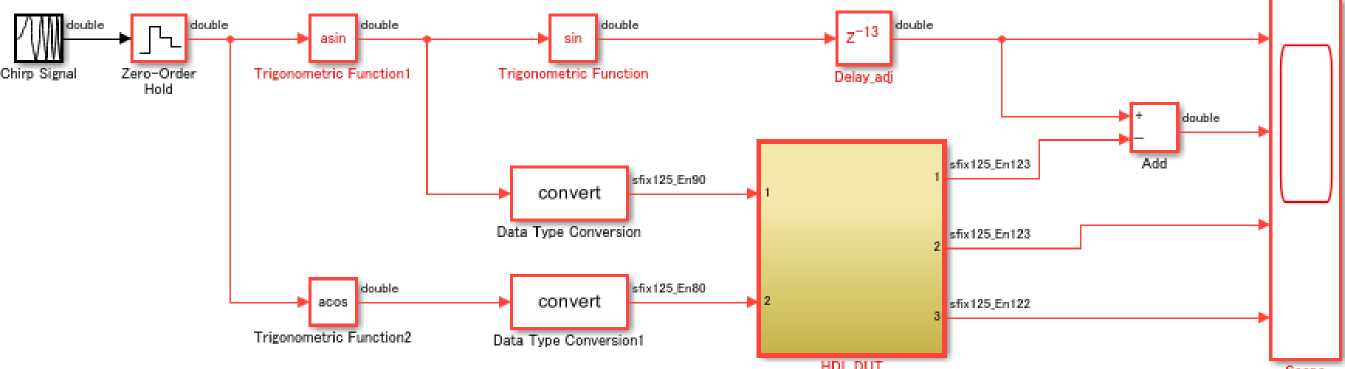
Locate Numeric Differences After Speed Optimization

This example shows how a model that contains Trigonometric Function blocks might have differences in numeric results after HDL code generation. You can observe these numeric differences in the generated validation model. The validation model compares the original model with the generated model that shows the effect of block implementations and speed and area optimizations.

Trigonometric Function Model

Open the model `hdlcoder_sincos_cordic_optimization`.

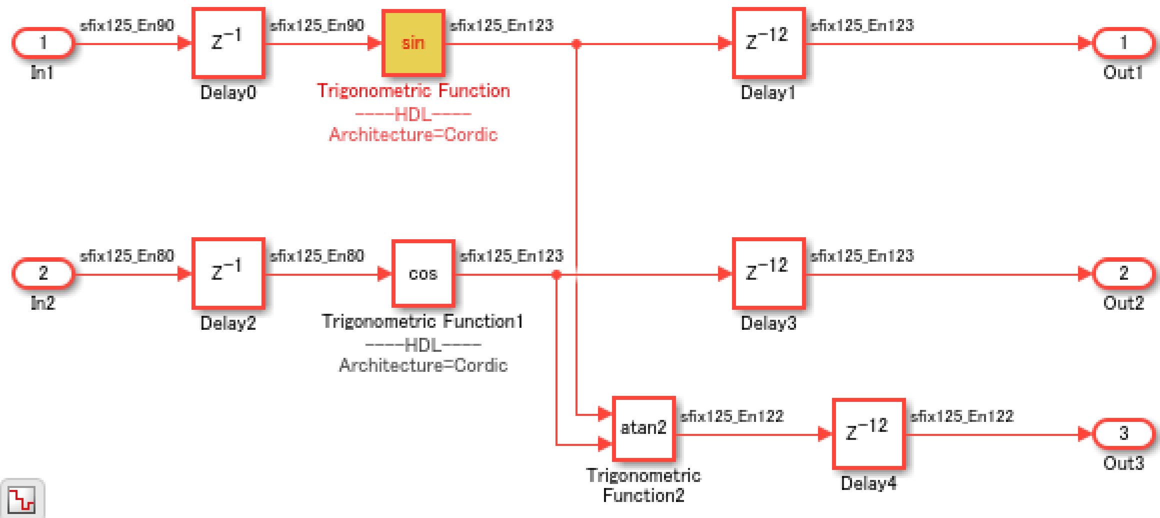
```
open_system('hdlcoder_sincos_cordic_optimization')
set_param('hdlcoder_sincos_cordic_optimization', 'SimulationCommand', 'Update')
```



Copyright 2020-2021 The MathWorks, Inc.

Inside the `HDL_DUT` subsystem, this model uses Trigonometric Function blocks that have HDL architecture set to `CORDIC` and **LatencyStrategy** set to `MAX`. The block settings introduce pipelines at the input of the Trigonometric Function blocks.

```
open_system('hdlcoder_sincos_cordic_optimization/HDL_DUT')
```



The model has various optimizations enabled on the model. To see the HDL parameters saved on the model, use the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_sincos_cordic_optimization')

%% Set Model 'hdlcoder_sincos_cordic_optimization' HDL parameters
hdlset_param('hdlcoder_sincos_cordic_optimization', 'ClockRatePipelining', 'off');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'EDAScriptGeneration', 'off');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'EnableTestpoints', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLGenerateWebview', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSubsystem', 'hdlcoder_sincos_cordic_opti
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthCmd', 'set_global_assignment -name \
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthFilePostfix', '_quartus.tcl');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthInit', 'load_package flow\nset top_
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthTerm', 'execute_flow -compile\nproje
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthTool', 'Quartus');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'MaskParameterAsGeneric', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'MinimizeClockEnables', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'MinimizeIntermediateSignals', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'ShareAdders', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'TargetLanguage', 'Verilog');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'Traceability', 'on');

hdlset_param('hdlcoder_sincos_cordic_optimization/HDL_DUT/Trigonometric Function', 'Architecture

hdlset_param('hdlcoder_sincos_cordic_optimization/HDL_DUT/Trigonometric Function1', 'Architecture
```

Generate HDL Code and Validation Model

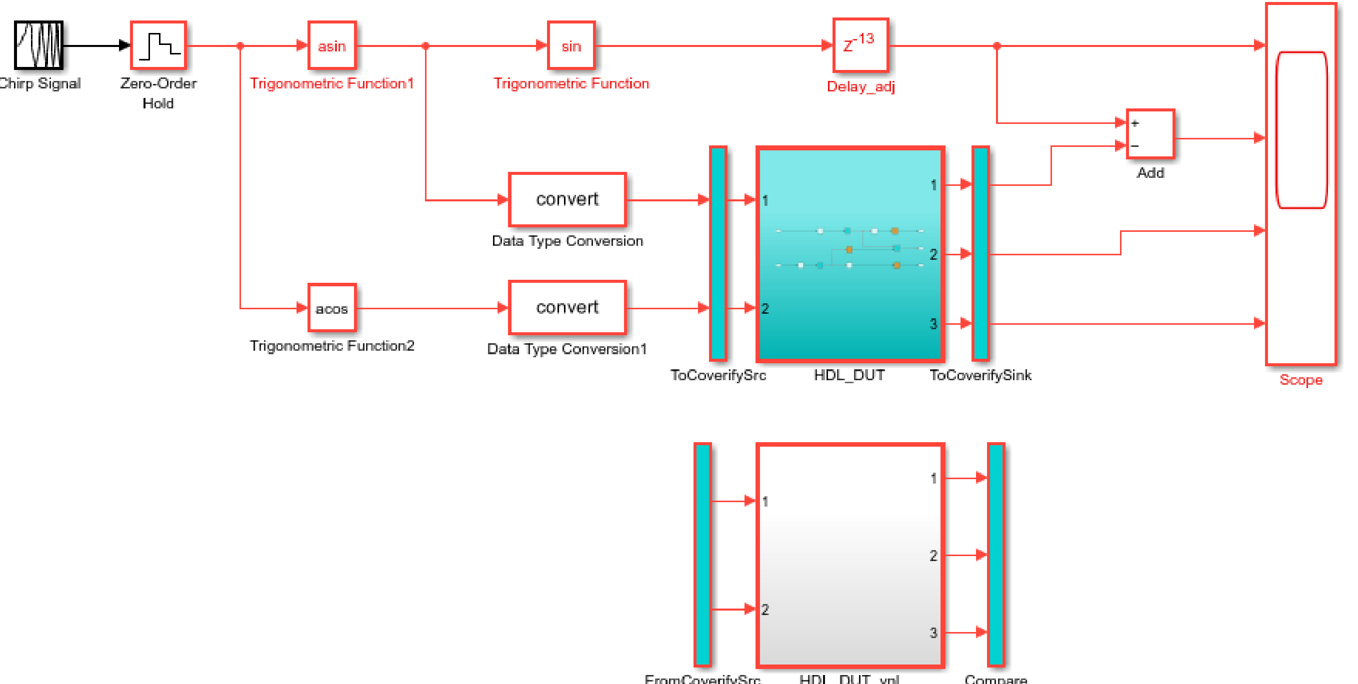
To see the effect of the optimization, generate HDL code and validation model for the `HDL_DUT` subsystem by using the `makehdl` function.

```
makehdl('hdlcoder_sincos_cordic_optimization')
```

When you open the HDL Check Report, you see a warning message displayed that indicates delays introduced at the inputs of the blocks, which might cause a numeric mismatch in the initial cycles when simulating the validation model.

After code generation, you see the model `gm_hdlcoder_sincos_cordic_optimization_vnl`. In this example, the model has been saved with the name `hdlcoder_sincos_cordic_optimization_validation`.

```
open_system('hdlcoder_sincos_cordic_optimization_validation')
set_param('hdlcoder_sincos_cordic_optimization_validation', 'SimulationCommand', 'Update')
```



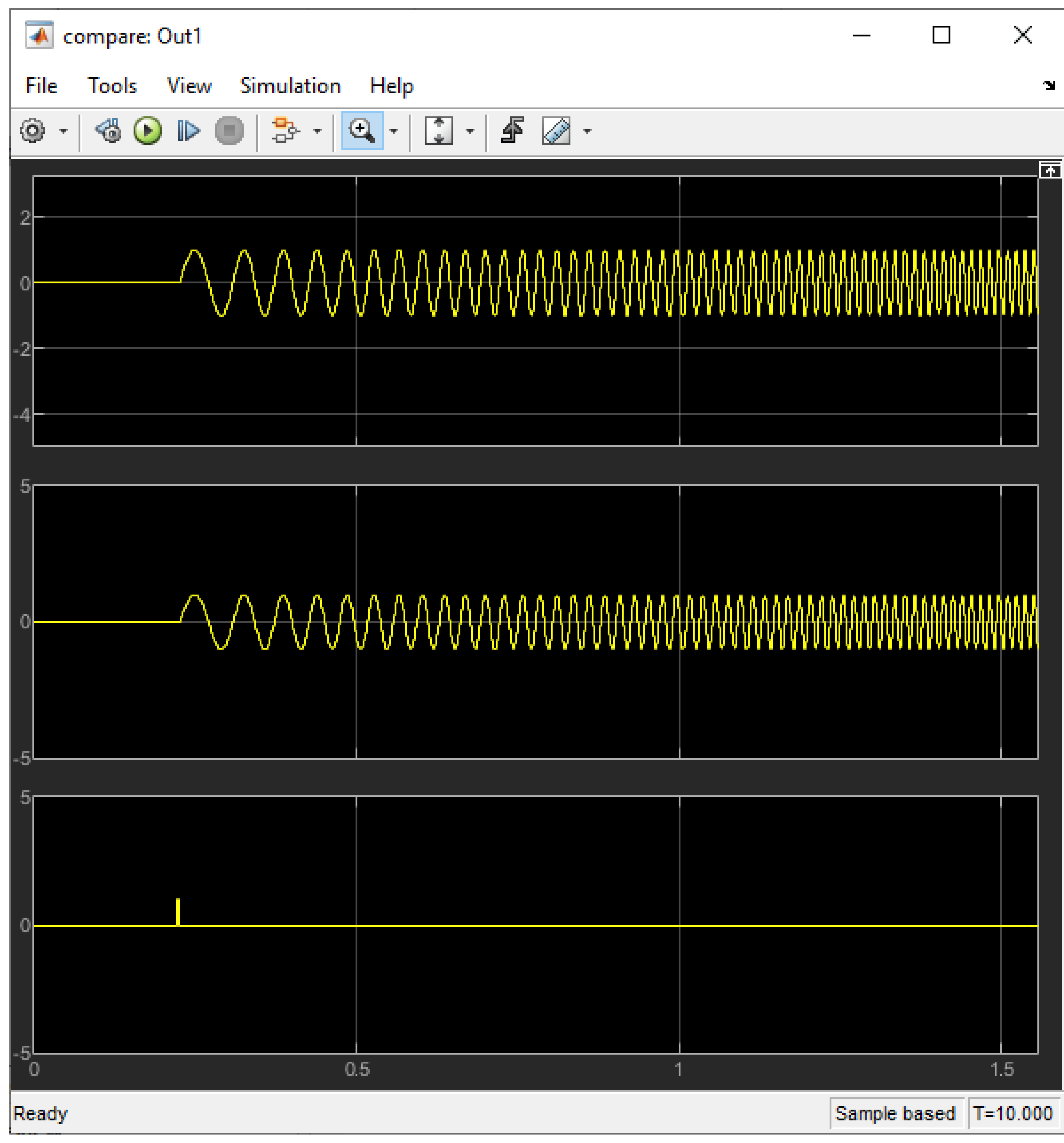
Copyright 2020-2021 The MathWorks, Inc.

Observe Numeric Differences

The `HDL_DUT` subsystem highlighted in cyan indicates that this subsystem is different from the subsystem in the original model `HDL_DUT_vnl`. The `HDL_DUT` subsystem is part of the generated model after HDL code generation, and shows the effect of optimizations. You also see the pipelines introduced by the Trigonometric Function blocks.

```
open_system('hdlcoder_sincos_cordic_optimization_validation/HDL_DUT')
```

When you simulate the model, you see assertions detected in the initial cycles of simulation, which indicates a numeric mismatch. The mismatch is caused by pipelines introduced at the input of the block. To fix the mismatch, avoid using block implementations, or HDL block properties such as **InputPipeline**, or optimizations that introduce pipelines at the input of the blocks.



See Also

More About

- "Generated Model and Validation Model" on page 21-10
- "Resolve Numeric Mismatch with Delay Balancing" on page 21-25

Simplify Constant Operations and Reduce Design Complexity in HDL Coder

HDL Coder™ performs certain optimization techniques that improve the quality of the generated HDL code. When you use floating-point data types in `Native Floating Point` mode and generate code from your model, at compile-time, HDL Coder searches for a subset of blocks that fit a certain pattern. When the code generator recognizes the pattern, it automatically performs certain optimization techniques to replace the blocks in the subset with other, simpler blocks.

The optimization techniques:

- Remove redundant run-time computations
- Simplify your design
- Improve quality and efficiency of generated HDL code
- Reduce latency and area footprint
- Improve timing of your design on target hardware

Simplification of Constant Operations

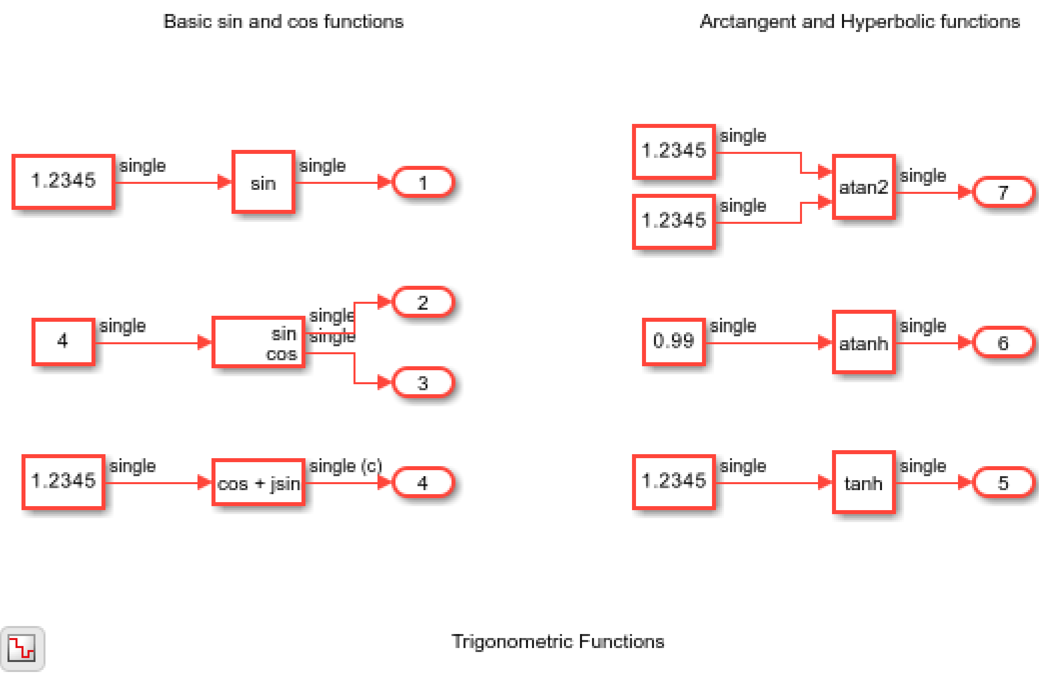
HDL Coder removes redundant operations in your design by evaluating constant subexpressions in advance. This optimization technique identifies Simulink® blocks in your model that have constant values at all input ports, and then replaces the blocks with Constant blocks. The code generator propagates the input constants inside the blocks to compute the resulting Constant value.

```
For example, c = 3 * 5 becomes c = 15.
```

This optimization works with any Simulink block that supports HDL code generation. For example:

- Open the model `hdlcoder_constant_simplification`. Double-click the `Trigonometric Functions` Subsystem.

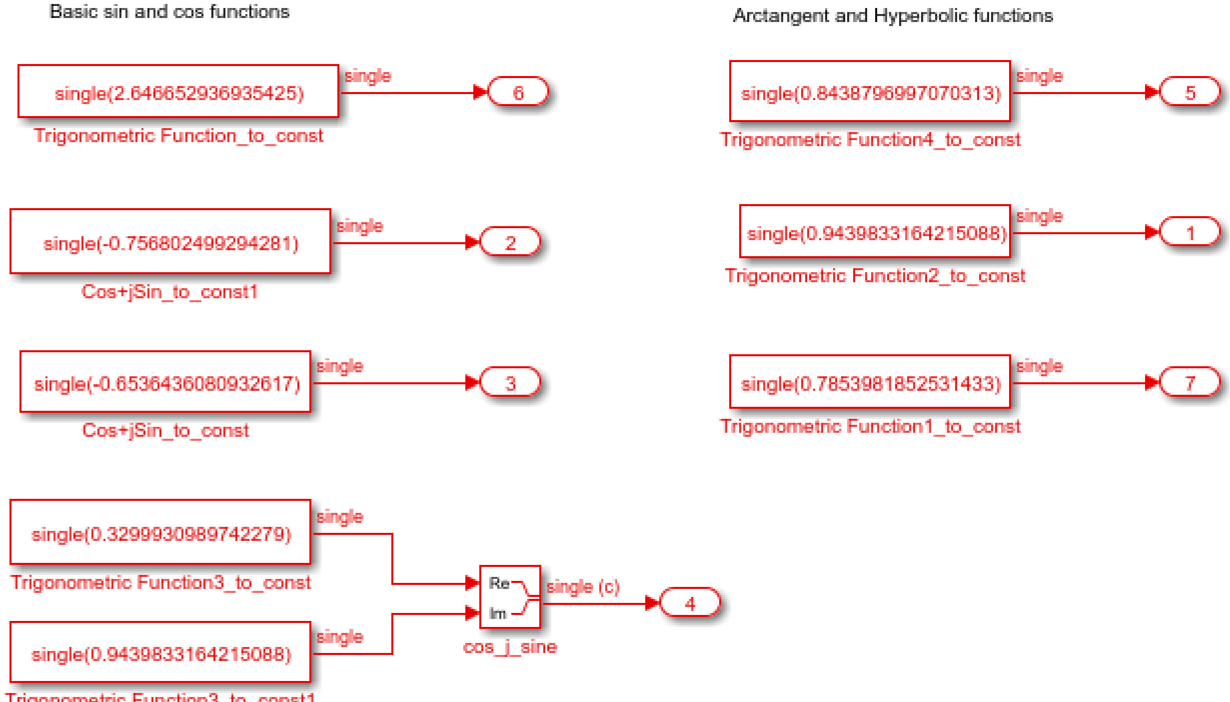
```
open_system('hdlcoder_constant_simplification')
open_system('hdlcoder_constant_simplification/Trigonometric Functions')
set_param('hdlcoder_constant_simplification', 'SimulationCommand', 'update');
```



- To generate HDL code for the `hdlcoder_constant_simplification` model, enter this command.

```
makehdl('hdlcoder_constant_simplification')
```

- Open the generated model, and double-click the `Trigonometric Functions` subsystem.



HDL Coder recognized the modeling pattern and replaced the constant single-precision trigonometric operations with constants. This optimization results in significant area reduction and improvements to timing when you deploy the code onto the target platform.

Replacement of Slower Operations with Faster Equivalents

This optimization automatically replaces slower operations with faster equivalents.

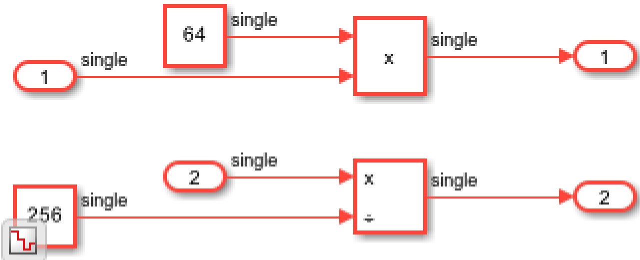
```
For example, r1 = r2 / 2 becomes r1 = r2 >> 1.
```

Examples of this optimization technique include replacement of a Product block or a Divide block by a Gain block. If one of the inputs to a Product block or a Divide block is a constant and a power of two, the code generator replaces that block by a Gain block. The code generator propagates the **Constant value** inside the Product block or the Divide block to compute the **Gain** parameter. This optimization works with single data types in the `Native Floating Point` mode.

For example:

- Open the model `hdlcoder_slow_operation_replacement`. Double-click the DUT Subsystem.

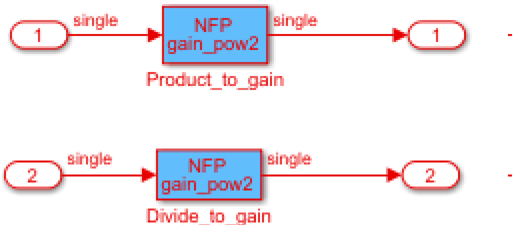
```
open_system('hdlcoder_slow_operation_replacement')
open_system('hdlcoder_slow_operation_replacement/DUT')
set_param('hdlcoder_slow_operation_replacement', 'SimulationCommand', 'update');
```



- To generate HDL code for the DUT Subsystem, enter this command:

```
makehdl('hdlcoder_slow_operation_replacement/DUT')
```

- Open the generated model, and double-click the `DUT` subsystem.



HDL Coder recognized the modeling pattern and replaced the Product block and the Divide block by a Gain block. This optimization significantly reduces the latency of your design, and improves area and timing on the target FPGA.

Combination of Multiple Operations

This optimization replaces several operations with one equivalent operation. Examples of this optimization technique include replacement of a Sin block and a Cos block by a Sincos block. If you provide the same input signal to a Sin block and a Cos block in your model, and when the HDL Block Properties of the Sin and Cos blocks match, the code generator replaces the blocks with a Sincos block. This optimization works with single data types in the `Native Floating Point` mode.

For example:

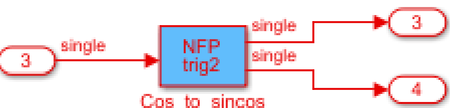
- Open the model `hdlcoder_combine_operations`. Double-click the DUT Subsystem.

```
open_system('hdlcoder_combine_operations')
open_system('hdlcoder_combine_operations/DUT')
set_param('hdlcoder_combine_operations', 'SimulationCommand', 'update');
```

- To generate HDL code for the DUT Subsystem, enter this command:

```
makehdl('hdlcoder_combine_operations/DUT')
```

- Open the generated model, and double-click the DUT Subsystem. The generated model appears as below.



HDL Coder recognized the modeling pattern and replaced the Sin block and the Cos block by a Sincos block. This optimization technique significantly improves the performance of your design on the target platform.

Considerations

- The optimizations work with floating-point data types. Fixed-point designs are not affected by this optimization. When you use single data types, enable the `Native Floating Point` mode. In the Configuration Parameters dialog box, in the **HDL Code Generation > Floating Point** pane, select **Use Floating Point**. To learn about native floating-point support in HDL Coder, see "Generate Target-Independent HDL Code with Native Floating-Point" on page 14-111.
- The optimizations preserve all comments from the blocks in the generated code. To learn about specifying comments to blocks, see "Generate HDL Code with Annotations or Comments" on page 23-24.
- The optimizations do not optimize blocks that use tunable parameters or generic inputs because the tunable parameters are not treated as constant values. To make these blocks participate in the optimization, in the Mask Editor for the blocks, clear the Tunable check box. To learn about tunable parameters, see "Generate DUT Ports for Tunable Parameters" on page 14-18.
- The optimizations treat enumeration values and constants from the Workspace browser as constant values. The code generator therefore propagates these values inside various components and simplifies the constant operations.

See Also

More About

- "Generated Model and Validation Model" on page 21-10
- "Remove Redundant Logic and Unused Blocks in Generated HDL Code" on page 21-224
- "Optimize Unconnected Ports in HDL Code for Simulink Models" on page 21-246

Optimization with Constrained Overclocking

In this section...

"Optimizations that Overclock Resources" on page 21-23
"How to Use Constrained Overclocking" on page 21-23
"Constrained Overclocking Limitations" on page 21-24

Area and timing optimizations that you specify can result in upsampled rates, or overclocking, in your design. For example, when you use the resource sharing optimization, HDL Coder overclocks the shared resources by an overclocking factor (OCF). The OCF depends on the number of shareable resources and the value for the **SharingFactor** parameter that you specify. If your clock rate is high, overclocking can cause your design clock rate to exceed the maximum clock rate of your target hardware. To constrain the overclocking of your design, specify the an oversampling value using the **Treat Simulink rates as actual hardware rates** parameter or the **Oversampling factor** parameter in conjunction with clock-rate pipelining.

Optimizations that Overclock Resources

Area and speed optimizations and certain block implementations result in overclocking the resources in your design. For example, these optimizations and implementations can result in upsampled rates in your design:

- RAM mapping
- Streaming
- Resource sharing
- Loop streaming
- Specific block implementations, such as cascade architectures, Newton-Raphson architectures, and some filter implementations

How to Use Constrained Overclocking

When using area and speed optimizations, you can specify constraints on overclocking by specifying an oversampling value for your model. To learn how to specify an oversampling value, see "Specifying the Oversampling Value" on page 20-9. If you want a single-rate design, you can use these parameters to prevent overclocking or limit overclocking within a range.

Suppose that you have a design that does not currently fit in the target hardware, but is already running at the target device maximum clock frequency, and you know that the inputs to your design can change at most every *N* cycles. You can enable area optimizations, such as resource sharing, and specify a single-rate implementation using the oversampling value of your model.

By default, the clock-rate pipelining optimization is enabled, and it works in conjunction with the oversampling value to make the DUT sample time slower than the actual clock rate. You can design your model at the sample time of your actual hardware, the data rate, and then enable the **Treat Simulink rates as actual hardware rates** parameter to let HDL Coder set an optimized oversampling value *N* for your design based on your model base rate and the clock rate. HDL Coder then has a latency budget of *N* cycles to perform the computation. In this situation, HDL Coder can reuse the shared resource at the original clock rate over *N* cycles, instead of implementing the sharing optimization by overclocking the shared resource.

Constrained Overclocking Limitations

When you constrain overclocking by enabling the **Treat Simulink rates as actual hardware rates** parameter or by setting the **Oversampling factor** parameter to a value greater than `1`, the **Clock inputs** parameter must be `Single`.

See Also

More About

- "Clock-Rate Pipelining" on page 21-148
- "Generate a Global Oversampling Clock" on page 20-9

Resolve Numeric Mismatch with Delay Balancing

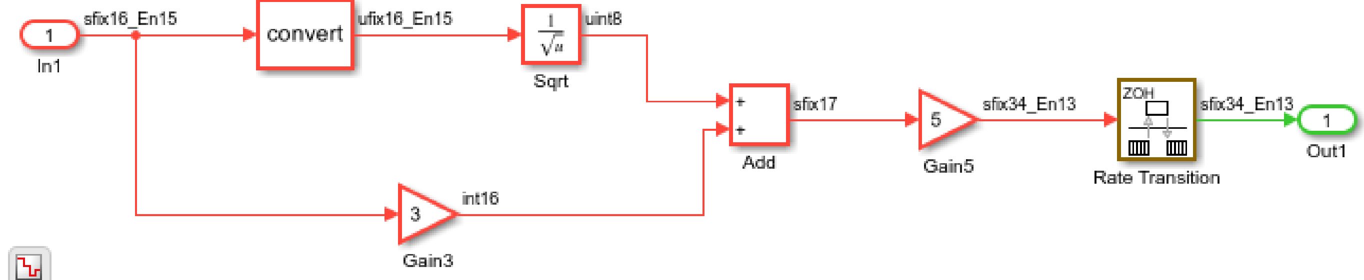
This example shows how to use delay balancing to resolve a numeric mismatch between the generated model and original model after HDL code generation.

Problem

The issue is that simulating the validation model results in a numeric mismatch between the original model and the generated model after HDL code generation. To illustrate this issue:

1. Open the `hdlcoder_resolve_delaybalancing` model. The DUT is a simple multirate design.

```
modelname = 'hdlcoder_resolve_delaybalancing';
dutname = 'hdlcoder_resolve_delaybalancing/Subsystem';
load_system(modelname)
open_system(dutname)
set_param(modelname, 'SimulationCommand', 'update');
```



2. Generate HDL code and validation model for the DUT.

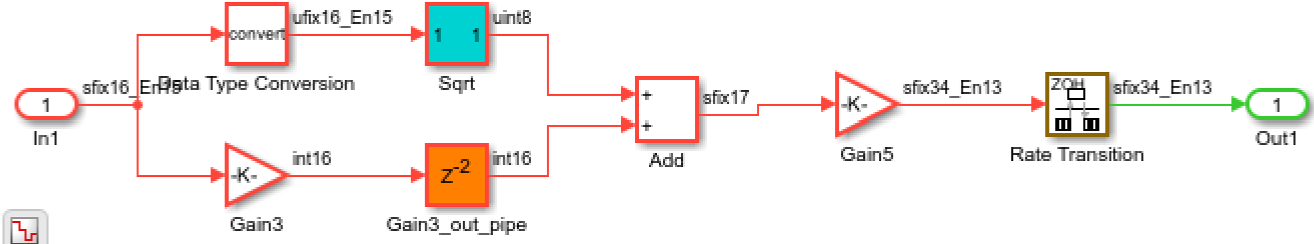
```
makehdl(dutname,'TreatBalanceDelaysOffAs', 'None', 'GenerateValidationModel', 'on', ...
                    'TargetDirectory','C:/Temp/hdlsrc')
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_resolve_delaybalancing')">hdlcode
### Generating HDL for <a href="matlab:open_system('hdlcoder_resolve_delaybalancing/Subsystem')">
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_resolve
### Running HDL checks on the model 'hdlcoder_resolve_delaybalancing'.
### Begin compilation of the model 'hdlcoder_resolve_delaybalancing'...
### Working on the model 'hdlcoder_resolve_delaybalancing'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_resolve_delaybalancing', 'Ge
### Begin model generation 'gm_hdlcoder_resolve_delaybalancing'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('C:\Temp\hdlsrc\hdlcoder_resolve_delayba
### Generating new validation model: '<a href="matlab:open_system('C:\Temp\hdlsrc\hdlcoder_resolv
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_resolve_delaybalancing'.
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 0.1.
### Begin VHDL Code Generation for 'Subsystem_tc'.
### Working on Subsystem_tc as C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_tc.vhd.
### Code Generation for 'Subsystem_tc' completed.
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_iv as C:\Temp\hdlsrc\hdlcoder_
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_core as C:\Temp\hdlsrc\hdlcode
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt as C:\Temp\hdlsrc\hdlcoder_resolve_
```

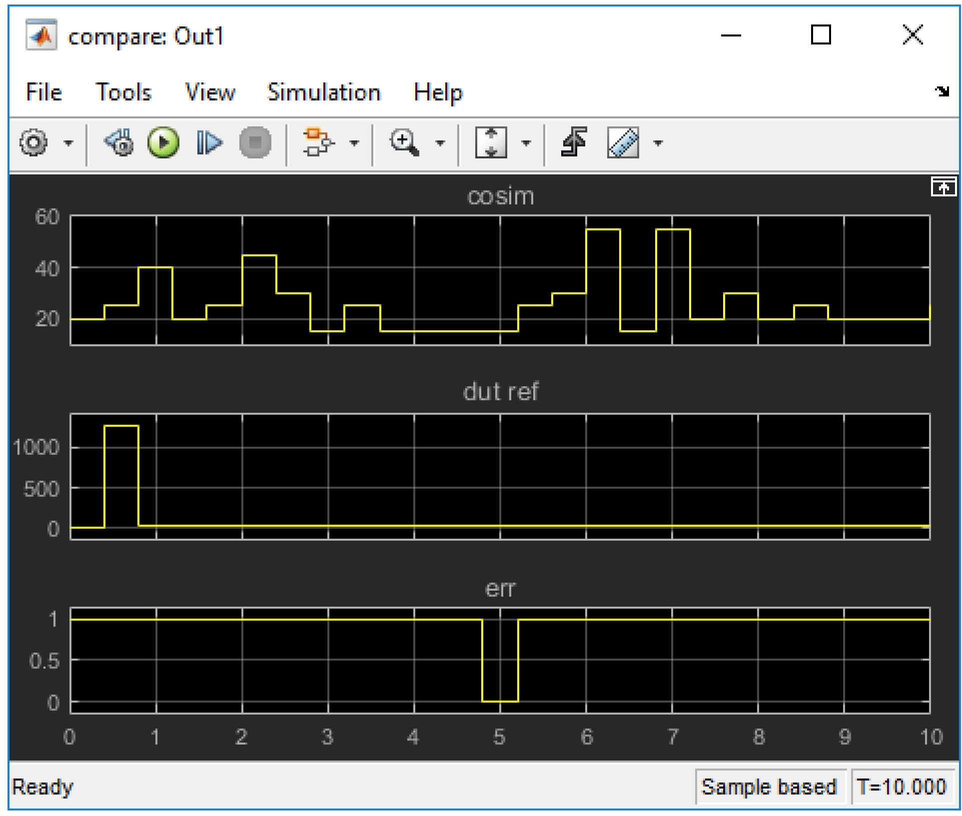
```
### Working on hdlcoder_resolve_delaybalancing/Subsystem as C:\Temp\hdlsrc\hdlcoder_resolve_delay
### Generating package file C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_pkg.vhd.
### Code Generation for 'hdlcoder_resolve_delaybalancing' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/Temp/hdlsrc/hdlcoder_resolve_delaybalanc
### HDL check for 'hdlcoder_resolve_delaybalancing' complete with 0 errors, 0 warnings, and 5 mes
### HDL code generation complete.
```

3. View the validation model. The validation model compares the generated model with the original model. The generated model displays the effect of optimizations and block-specific architectures that you specify. Use the validation model to verify that the DUT in the generated model is bit-true to the numeric results produced by the original DUT.

```
valmodelname = 'gm_hdlcoder_resolve_delaybalancing_vnl';
valmodelsubsys = 'gm_hdlcoder_resolve_delaybalancing_vnl/Subsystem';
load_system(valmodelname)
open_system(valmodelsubsys)
set_param(valmodelname, 'SimulationCommand', 'update');
```



4. Simulate the validation model. HDL Coder™ generates warnings that indicate an assertion detected at various time stamps. If you navigate through the validation model by double-clicking the `Compare` Subsystem and then the `Assert_Out1` Subsystem, you see a `compare: Out1` Scope block. This Scope block compares the output of the original model DUT with the generated model DUT and displays numeric differences as an error signal. When you double-click the Scope block, you see a nonzero error, which indicates a numerical mismatch.



Cause

To diagnose this issue:

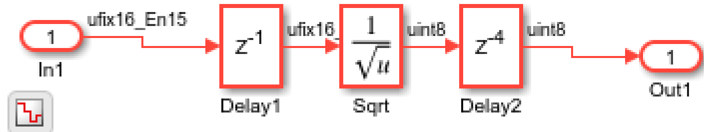
1. Observe the parameters saved on the original model. You see that `BalanceDelays` is set to `off` on the model.

```
hdlsaveparams(modelname)
```

```
%% Set Model 'hdlcoder_resolve_delaybalancing' HDL parameters
hdlset_param('hdlcoder_resolve_delaybalancing', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_resolve_delaybalancing', 'GenerateHDLTestBench', 'off');
hdlset_param('hdlcoder_resolve_delaybalancing', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_resolve_delaybalancing', 'HDLSubsystem', 'hdlcoder_resolve_delaybalancing/

% Set Gain HDL parameters
hdlset_param('hdlcoder_resolve_delaybalancing/Subsystem/Gain3', 'OutputPipeline', 2);

hdlset_param('hdlcoder_resolve_delaybalancing/Subsystem/Sqrt', 'Architecture', 'RecipSqrtNewton')
```

2. Inspect the validation model. Inside the DUT Subsystem, you see that the code generator implemented the reciprocal square root operation as a Subsystem. If you double-click the Sqrt Subsystem, you see that the implementation has a latency. This latency arises due to the Newton-Raphson implementation of reciprocal square root.

```
open_system('gm_hdlcoder_resolve_delaybalancing_vnl/Subsystem/Sqrt')
```



The simulation mismatch occurred because the Newton-Raphson choice for implementing the Reciprocal Sqrt block results in a latency difference between the original model and the generated model. In addition, the downsampling introduced by the Rate Transition block drops samples. As delay balancing is disabled on the model, the code generator did not add matching delays to account for this latency.

Solution

To fix this issue, enable delay balancing on the model. In the original model, set `BalanceDelays` to `on`. When you enable delay balancing, the code generator detects introduction of delays along one path and adds matching delays on other, parallel signal paths.

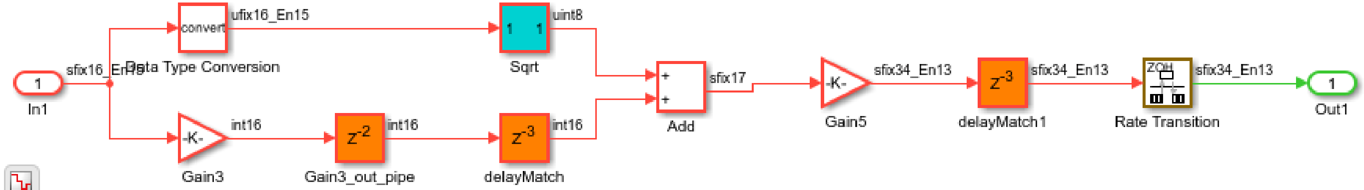
1. Enable `BalanceDelays` on the model and generate HDL code and validation model.

```
load_system(modelname)
makehdl(dutname,'TreatBalanceDelaysOffAs', 'Error','BalanceDelays','on', ...
                'GenerateValidationModel', 'on', ...
                    'TargetDirectory','C:/Temp/hdlsrc')
```

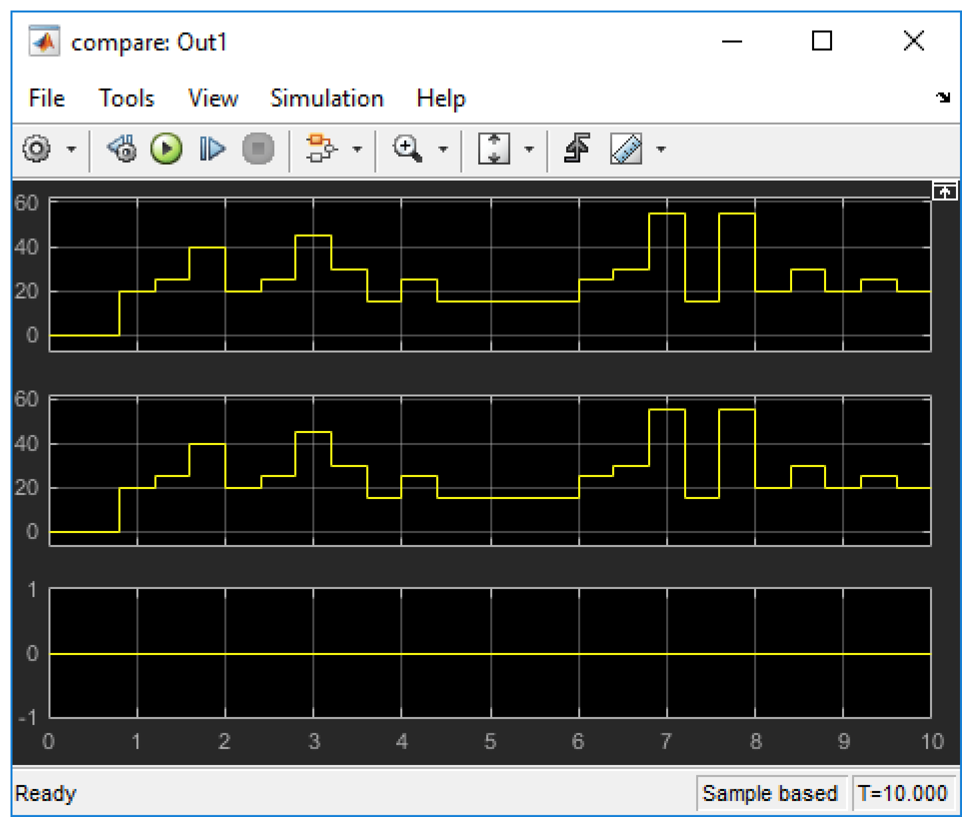
```
### Working on the model <a href="matlab:open_system('hdlcoder_resolve_delaybalancing')">hdlcoder
### Generating HDL for <a href="matlab:open_system('hdlcoder_resolve_delaybalancing/Subsystem')">
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_resolve
### Running HDL checks on the model 'hdlcoder_resolve_delaybalancing'.
### Begin compilation of the model 'hdlcoder_resolve_delaybalancing'...
### Working on the model 'hdlcoder_resolve_delaybalancing'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_resolve_delaybalancing', 'Ge
### Begin model generation 'gm_hdlcoder_resolve_delaybalancing'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('C:\Temp\hdlsrc\hdlcoder_resolve_delayba
### Generating new validation model: '<a href="matlab:open_system('C:\Temp\hdlsrc\hdlcoder_resolv
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_resolve_delaybalancing'.
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 0.1.
### Begin VHDL Code Generation for 'Subsystem_tc'.
### Working on Subsystem_tc as C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_tc.vhd.
### Code Generation for 'Subsystem_tc' completed.
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_iv as C:\Temp\hdlsrc\hdlcoder_
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_core as C:\Temp\hdlsrc\hdlcode
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt as C:\Temp\hdlsrc\hdlcoder_resolve_
### Working on hdlcoder_resolve_delaybalancing/Subsystem as C:\Temp\hdlsrc\hdlcoder_resolve_delay
### Generating package file C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_pkg.vhd.
### Code Generation for 'hdlcoder_resolve_delaybalancing' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/Temp/hdlsrc/hdlcoder_resolve_delaybalanc
### HDL check for 'hdlcoder_resolve_delaybalancing' complete with 0 errors, 0 warnings, and 4 mes
### HDL code generation complete.
```

2. Open the validation model. You see that the code generator introduced matching delays to balance the latency introduced by the Sqrt block and to offset the effect of downsampling. The additional delays account for the latency difference.

```
load_system(valmodelname)
open_system(valmodelsubsys)
set_param(valmodelname, 'SimulationCommand', 'update');
```



3. Simulate the validation model and open the `compare: Out1` Scope block. You see that the numerical mismatch has been resolved.



See Also

Related Examples

- "Delay Balancing and Validation Model Workflow in HDL Coder" on page 21-94

More About

- "Delay Balancing" on page 21-81
- "Generated Model and Validation Model" on page 21-10
- "Check delay balancing setting" on page 37-11

Resolve Simulation Mismatch When Pipelining with a Feedback Loop Outside the DUT

This example shows how to use pipelining optimizations to resolve a simulation mismatch between the generated model and original model after HDL code generation.

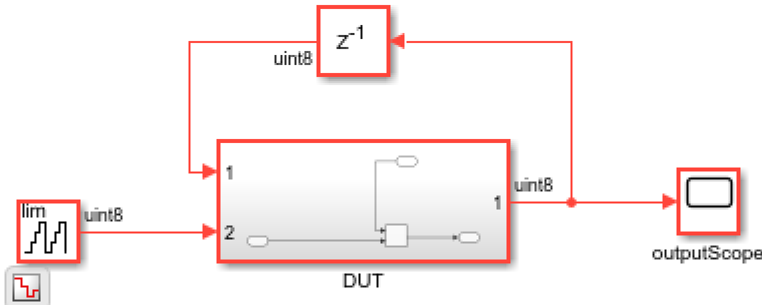
Issue

If you have a feedback loop around your device under test (DUT) subsystem and pipeline optimizations enabled in the DUT, you might have a simulation mismatch between your generated and original model. This mismatch is due to added latency from delays or pipeline registers in the DUT of the generated model that are not present in the original model DUT. A simulation mismatch shows that your original and generated model, and your original model and generated HDL code, are not functionally equivalent.

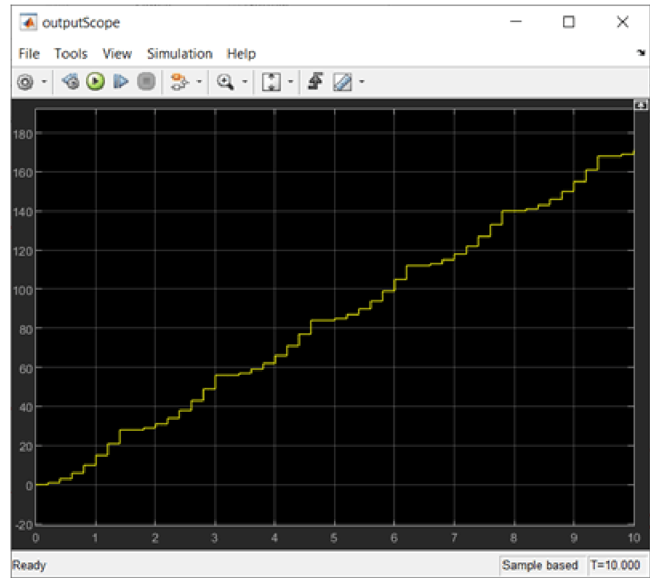
HDL Coder™ generates code only for your DUT and nothing outside of that in your model. As a result, the simulation mismatch does not cause code generation errors. The mismatch is also not seen in the validation model because it is comparing only the original model DUT to the generated model DUT.

This issue can be seen in the `feedback_loop_outsideDUT` model. In the model, the feedback delay is outside of the DUT subsystem that HDL Coder generates code for.

```
modelname = "feedback_loop_outsideDUT";
dutname = "feedback_loop_outsideDUT/DUT";
open_system(modelname);
sim(modelname);
```



Open the `outputScope` of the original model.



In the model, the HDL Block properties `InputPipeline` and `OutputPipeline` are set to 1 for the `Add` block. These properties place a delay block, which acts as a pipeline register, before and after the `Add` block to add pipelines in the combinatorial path and increase the clock frequency of your design. Because there are options enabled to add pipelines in the generated model and HDL code, delay balancing is enabled (its default) for the model. It is recommended to keep delay balancing enabled to balance delays added in your design. For more information, see “Delay Balancing” on page 21-81.

```
hdlsaveparams(modelname)
```

```
%% Set Model 'feedback_loop_outsideDUT' HDL parameters
hdlset_param('feedback_loop_outsideDUT', 'GenerateHDLCode', 'off');
hdlset_param('feedback_loop_outsideDUT', 'GenerateValidationModel', 'on');
hdlset_param('feedback_loop_outsideDUT', 'HDLSubsystem', 'feedback_loop_outsideDUT/DUT');

% Set SubSystem HDL parameters
hdlset_param('feedback_loop_outsideDUT/DUT', 'ClockRatePipelining', 'off');

% Set Sum HDL parameters
hdlset_param('feedback_loop_outsideDUT/DUT/Add', 'InputPipeline', 1);
hdlset_param('feedback_loop_outsideDUT/DUT/Add', 'OutputPipeline', 1);
```

Generate HDL code. In the MATLAB Command Window, a message during code generation shows that two cycles of latency have been introduced in the generated model. The latency is not compensated for during code generation because the feedback loop is outside of the DUT. Because of the latency introduced, the original model and generated model have a simulation mismatch.

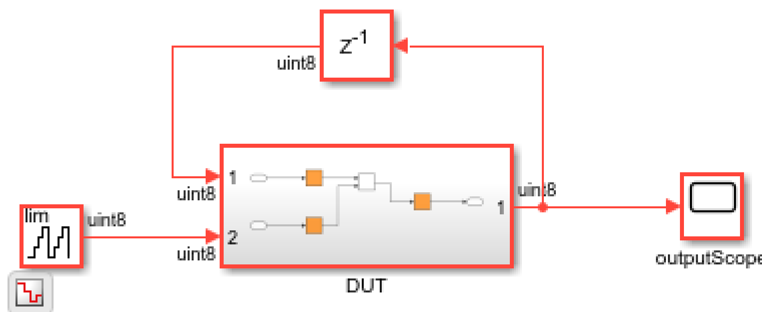
```
makehdl(dutname);
```

```
### Working on the model <a href="matlab:open_system('feedback_loop_outsideDUT')">feedback_loop_o
### Generating HDL for <a href="matlab:open_system('feedback_loop_outsideDUT/DUT')">feedback_loop
### Using the config set for model <a href="matlab:configset.showParameterGroup('feedback_loop_ou
### Running HDL checks on the model 'feedback_loop_outsideDUT'.
### Begin compilation of the model 'feedback_loop_outsideDUT'...
### Working on the model 'feedback_loop_outsideDUT'...
```

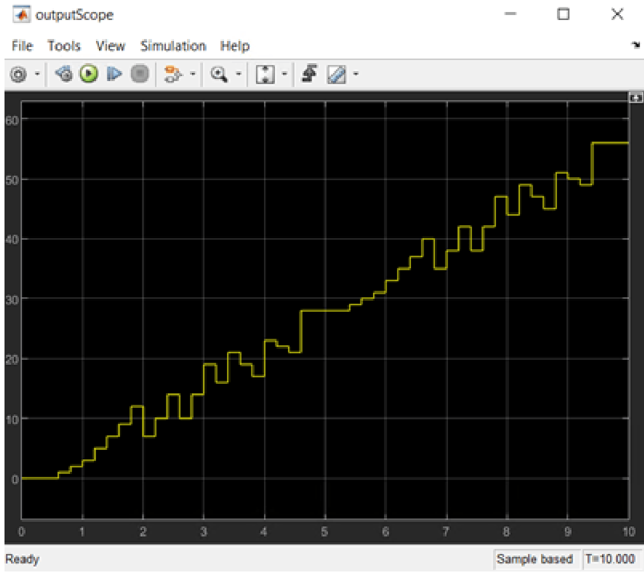
```
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('feedback_loop_outsideDUT', 'GenerateMo
### Begin model generation 'gm_feedback_loop_outsideDUT'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\feedback_loop_outsideDUT\gm_feed
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\feedback_loop_outsideDU
### Validation model generation complete.
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'feedback_loop_outsideDUT' complete with 0 errors, 0 warnings, and 1 messages.
```

Open and run the generated model.

```
open_system("gm_feedback_loop_outsideDUT");
sim("gm_feedback_loop_outsideDUT");
```



In the `outputScope` of the generated model, compare the scopes of the original and generated model to see the simulation mismatch between the two scopes.

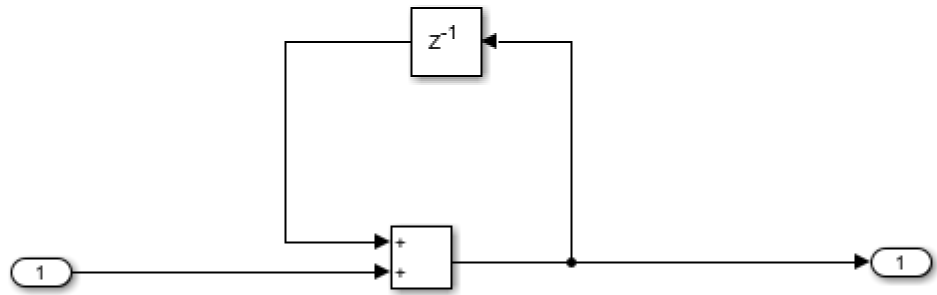


Potential Solutions

Use Clock-Rate Pipelining

To fix this issue, it is recommended to bring the design delay inside of the DUT subsystem so it can be taken into account during HDL code generation. The feedback delay is brought into the DUT in the model `feedback_loop_inDUT`. HDL Coder then takes the entire feedback loop into account during code generation.

```
modelname = "feedback_loop_inDUT";
load_system(modelname);
dutname = "feedback_loop_inDUT/DUT";
open_system(dutname);
```



When you generate code for this DUT by using the `makehdl` command, this delay balancing error is displayed in the MATLAB Command Window:

```
Delay balancing unsuccessful because Delay introduced in feedback loop cannot
be path balanced. Offending Block:feedback_loop_inDUT/DUT/Add_out_pipe.
```

While the error in the HDL code generation is about unsuccessful delay balancing, the delay balancing issue is a result of pipelining in feedback loops that introduce latency and break functional equivalence between the original and generated model. To pipeline your design inside a feedback loop by using optimization options, such as the HDL Block properties `InputPipeline` and `OutputPipeline` for blocks in your DUT, use a faster clock rate and enable the clock-rate pipelining optimization. For more information, see “Clock-Rate Pipelining” on page 21-148.

In the `feedback_loop_inDUT` model, to use clock-rate pipelining:

- Set an oversampling factor of 10 to increase the clock rate to be 10x the model base rate, or data rate.
- Enable clock-rate pipelining for the DUT.

```
hdlset_param(modelname, "Oversampling", 10)
hdlset_param(dutname, "ClockRatePipelining", "on");
```

Now, when you generate code, there are no errors and the latency decreases from two data rate cycles to one cycle. The latency that remains is a result of the feedback loop in the DUT. The latency is not added during code generation.

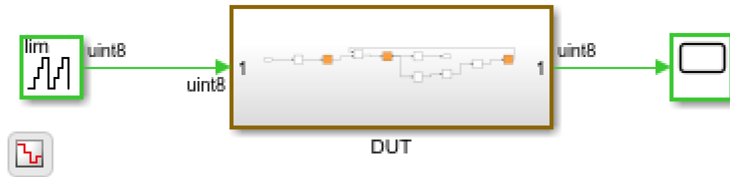
```
makehdl(dutname);
```

```
### Working on the model <a href="matlab:open_system('feedback_loop_inDUT')">feedback_loop_inDUT<
### Generating HDL for <a href="matlab:open_system('feedback_loop_inDUT/DUT')">feedback_loop_inDU
```

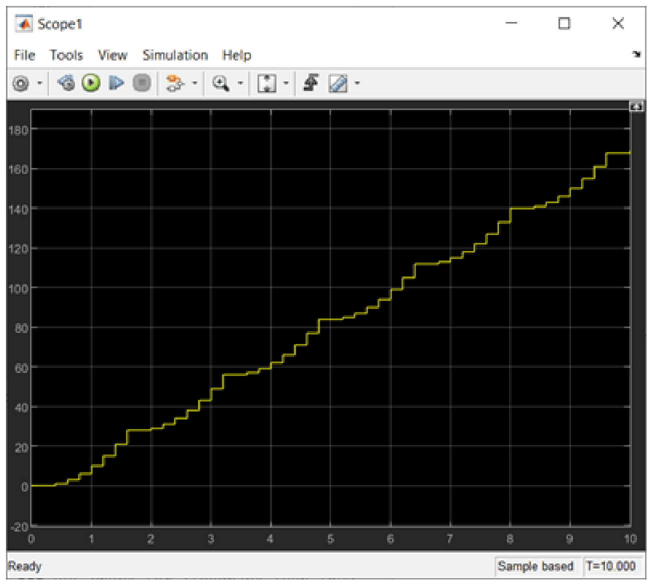
```
### Using the config set for model <a href="matlab:configset.showParameterGroup('feedback_loop_in
### Running HDL checks on the model 'feedback_loop_inDUT'.
### Begin compilation of the model 'feedback_loop_inDUT'...
### Working on the model 'feedback_loop_inDUT'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('feedback_loop_inDUT', 'GenerateModel')
### Begin model generation 'gm_feedback_loop_inDUT'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\feedback_loop_inDUT\gm_feedback_
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\feedback_loop_inDUT\gm_
### Validation model generation complete.
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'feedback_loop_inDUT' complete with 0 errors, 0 warnings, and 2 messages.
```

Open and run the generated model.

```
open_system("gm_feedback_loop_inDUT");
sim("gm_feedback_loop_inDUT");
```



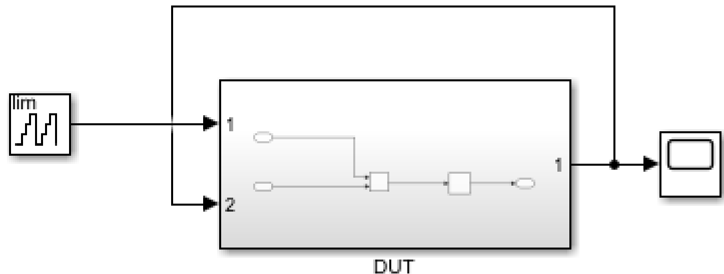
You can now compare the original model `feedback_loop_inDUT` to the generated model `gm_feedback_loop_inDUT` by using the output scopes of the original and generated models. The simulation of the original model with the feedback loop inside or outside the DUT is the same simulation. The generated model with the feedback loop inside the DUT now has a functionally matching output scope, delayed by one data rate cycle of latency.



The output scope shows that the original and generated model are now functionally equivalent. This functional equivalence can also be seen by using the validation model because the feedback loop is inside the DUT and you can directly compare the two DUTs.

Now that the issue is fixed by using clock-rate pipelining, for design purposes, you can move the feedback loop wire outside of the DUT while keeping the delay inside the DUT. This option enables you to maintain your original design as much as possible, while fixing the simulation mismatch with clock-rate pipelining.

```
modelname = "delay_inDUT";
load_system(modelname);
dutname = "delay_inDUT/DUT";
open_system(modelname);
```



Applying this solution means that the delay balancing error during code generation is not generated. As a result, the simulation issue is not indicated during code generation when you move only the delay inside the DUT and have disabled clock-rate pipelining.

```
makehdl(dutname);
```

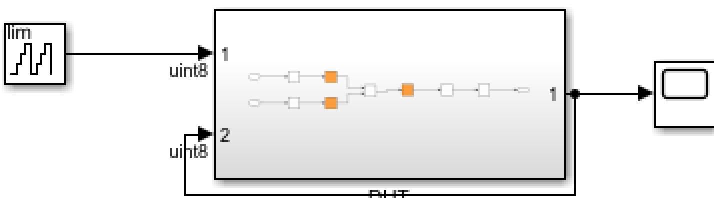
```
### Working on the model <a href="matlab:open_system('delay_inDUT')">delay_inDUT</a>
### Generating HDL for <a href="matlab:open_system('delay_inDUT/DUT')">delay_inDUT/DUT</a>
### Using the config set for model <a href="matlab:configset.showParameterGroup('delay_inDUT', {
### Running HDL checks on the model 'delay_inDUT'.
### Begin compilation of the model 'delay_inDUT'...
### Working on the model 'delay_inDUT'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('delay_inDUT', 'GenerateModel')">Genera
### Begin model generation 'gm_delay_inDUT'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\delay_inDUT\gm_delay_inDUT.slx')
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\delay_inDUT\gm_delay_in
### Validation model generation complete.
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'delay_inDUT' complete with 0 errors, 0 warnings, and 1 messages.
```

Even though there is no delay balancing error to fix during code generation, the simulation mismatch still needs to be fixed. This requires the same solution of setting the oversampling factor and enabling clock-rate pipelining. Removing the feedback loop from the DUT but keeping the design delay inside allows the design delay to be absorbed and reduces the latency to zero.

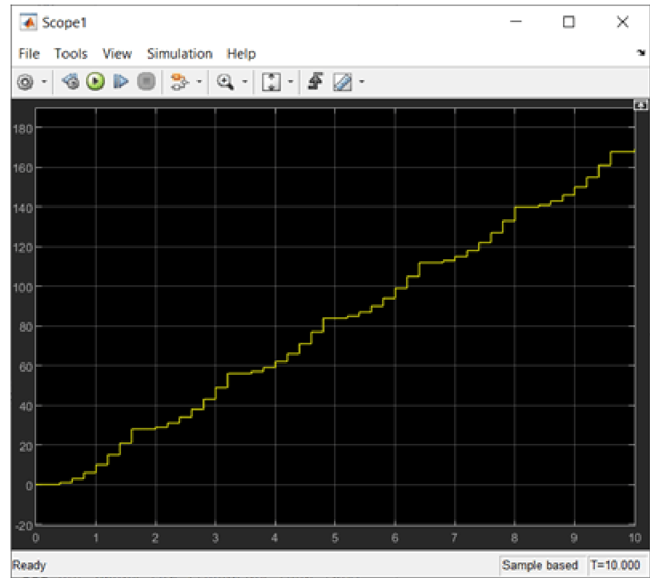
```
hdlset_param(modelname, "Oversampling", 10)
hdlset_param(dutname, "ClockRatePipelining", "on");

makehdl(dutname);

### Working on the model <a href="matlab:open_system('delay_inDUT')">delay_inDUT</a>
### Generating HDL for <a href="matlab:open_system('delay_inDUT/DUT')">delay_inDUT/DUT</a>
### Using the config set for model <a href="matlab:configset.showParameterGroup('delay_inDUT', {
### Running HDL checks on the model 'delay_inDUT'.
### Begin compilation of the model 'delay_inDUT'...
### Working on the model 'delay_inDUT'...
### Working on... <a href="matlab:configset.internal.open('delay_inDUT', 'GenerateModel')">Genera
### Begin model generation 'gm_delay_inDUT'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\delay_inDUT\gm_delay_inDUT.slx')
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\delay_inDUT\gm_delay_ir
### Validation model generation complete.
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'delay_inDUT' complete with 0 errors, 0 warnings, and 1 messages.
```

Open and run the generated model. You can compare the output scopes of the original and generated model after code generation to see the issue is resolved and there is no added latency during code generation.

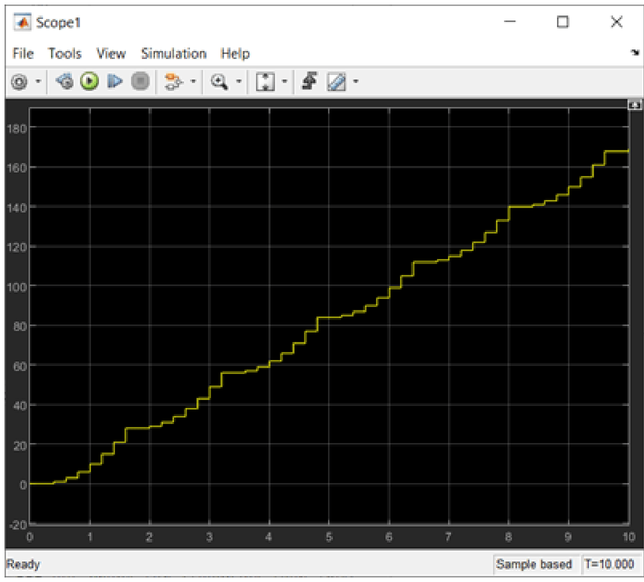
```
open_system("gm_delay_inDUT");
sim("delay_inDUT");
```



This is the output scope of the original model `delay_inDUT`.



This is the output scope of the generated model `gm_delay_inDUT`.



Disable Optimizations That Add Latency

To keep the feedback loop outside of the DUT with no added latency to the generated model, remove all optimizations that add pipelines to your generated model and HDL code. In this example, remove the input and output pipelines optimization options that are specified for the `Add` block in your DUT subsystem in the `feedback_loop_outsideDUT` model.

```
modelname = "feedback_loop_outsideDUT";
dutname = "feedback_loop_outsideDUT/DUT";
open_system(modelname);

hdlset_param("feedback_loop_outsideDUT/DUT/Add", "InputPipeline", 0);
hdlset_param("feedback_loop_outsideDUT/DUT/Add", "OutputPipeline", 0);
```

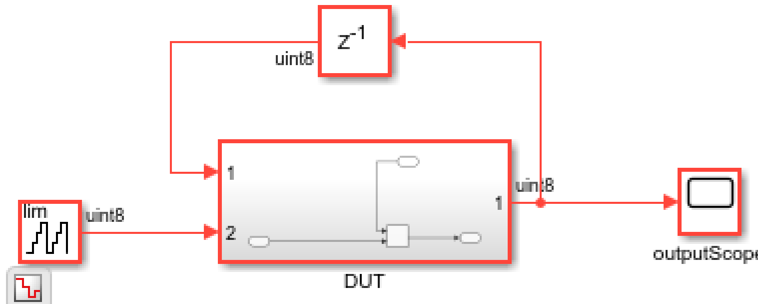
Removing the input and output pipelines removes the added latency to the generated model. The original and generated model are now functionally equivalent with no simulation mismatch. To compare the original and generated model scope outputs, generate HDL code.

```
makehdl(dutname)
```

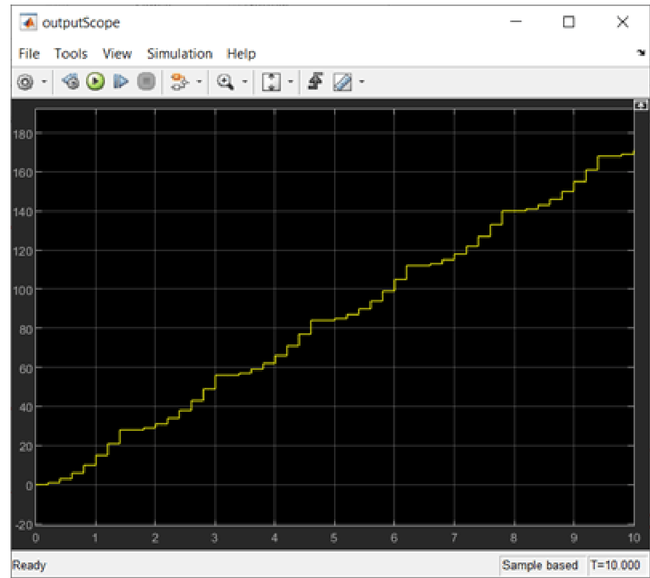
```
### Working on the model <a href="matlab:open_system('feedback_loop_outsideDUT')">feedback_loop_o
### Generating HDL for <a href="matlab:open_system('feedback_loop_outsideDUT/DUT')">feedback_loop
### Using the config set for model <a href="matlab:configset.showParameterGroup('feedback_loop_ou
### Running HDL checks on the model 'feedback_loop_outsideDUT'.
### Begin compilation of the model 'feedback_loop_outsideDUT'...
### Working on the model 'feedback_loop_outsideDUT'...
### Working on... <a href="matlab:configset.internal.open('feedback_loop_outsideDUT', 'GenerateMo
### Begin model generation 'gm_feedback_loop_outsideDUT'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\feedback_loop_outsideDUT\gm_feed
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\feedback_loop_outsideDU
### Validation model generation complete.
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'feedback_loop_outsideDUT' complete with 0 errors, 0 warnings, and 1 messages.
```

Open and run the generated model.

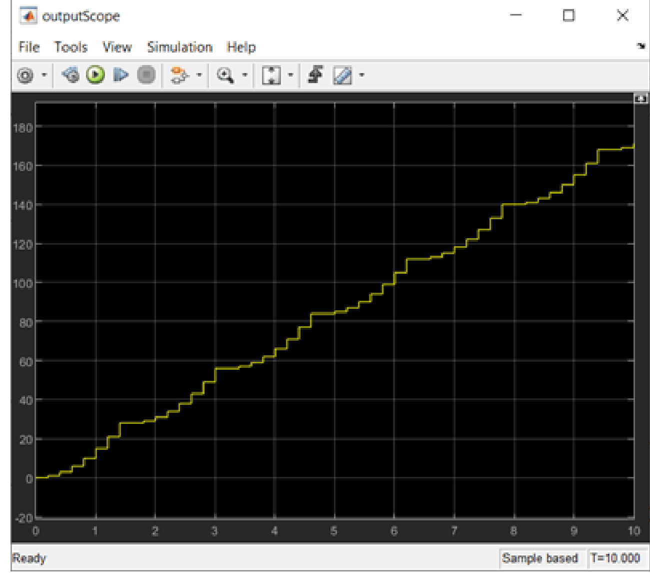
```
open_system("gm_feedback_loop_outsideDUT");
sim("gm_feedback_loop_outsideDUT");
```



This is the output scope of the original model `feedback_loop_outsideDUT`.



This is the output scope of the generated model `gm_feedback_loop_outsideDUT`.



Redesign the Model to Handle Added Latency

If you prefer to keep the pipeline optimization in your design, you can redesign your model to handle the added latency in order to remove the simulation mismatch. For example, you can redesign your model to no longer contain a feedback loop. If you remove the feedback loop, HDL Coder can balance delays to handle the added latency and maintain functional equivalence between the original and generated model to avoid a simulation mismatch.

See Also

"Guidelines for Using Delays and Goto and From Blocks for HDL Code Generation" on page 18-105

Related Examples

- "Resolve Numeric Mismatch with Delay Balancing" on page 21-25

Streaming

In this section...

"What Is Streaming?" on page 21-42
"Specify Streaming" on page 21-42
"How to Determine Streaming Factor and Sample Time" on page 21-43
"Determine Blocks That Support Streaming" on page 21-43
"Requirements for Streaming Subsystems" on page 21-43
"Streaming Report" on page 21-44

What Is Streaming?

Streaming is an area optimization in which HDL Coder transforms a vector data path to a scalar data path (or to several smaller-sized vector data paths). By default, HDL Coder generates fully parallel implementations for vector computations. For example, the code generator realizes a vector sum as several adders, executing in parallel during a single clock cycle. This technique can consume many hardware resources. With streaming, the generated code saves chip area by multiplexing the data over a smaller number of shared hardware resources.

By specifying a streaming factor for a subsystem, you can control the degree to which such resources are shared within that subsystem. When the ratio of streaming factor (N_{st}) to subsystem data path width (V_{dim}) is 1:1, HDL Coder implements an entirely scalar data path. A streaming factor of 0 (the default) produces a fully parallel implementation (that is, without sharing) for vector computations.

If you know the maximal vector dimensions and the sample rate for a subsystem, you can compute the possible streaming factors and resulting sample rates for the subsystem. However, even if the requested streaming factor is mathematically possible, the subsystem must meet other criteria for streaming.

By default, when you apply the streaming optimization, HDL Coder oversamples the shared hardware resource to generate an area-optimized implementation with the original latency. If the streamed data path is operating at a rate slower than the base rate, the code generator implements the data path at the base rate. You can also limit the oversampling ratio to meet target hardware clock constraints. To learn more, see "Clock-Rate Pipelining" on page 21-148.

You can generate and use the validation model to verify that the output of the optimized DUT is bit-true to the results produced by the original DUT. To learn more about the validation model, see "Generated Model and Validation Model" on page 21-10.

Specify Streaming

To specify streaming from the UI:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the subsystem, model reference, or MATLAB Function block and then click **HDL Block Properties**. In the **StreamingFactor** field, enter the number of resources that you want to stream.

Note For MATLAB Function blocks, to specify the **StreamingFactor**, in the HDL Block Properties dialog box, you must set the HDL architecture of the block to `MATLAB Datapath`.

- Right-click the subsystem, model reference, or MATLAB Function block and select **HDL Code > HDL Block Properties**. In the **StreamingFactor** field, enter the number of resources that you want to stream.

At the command-line, you can set `StreamingFactor` using the `hdlset_param` function, as in the following example.

```
modelname = 'sfir_fixed'
dut = 'sfir_fixed/symmetric_fir';
open_system(modelname)
hdlset_param(dut,'StreamingFactor', 4);
```

How to Determine Streaming Factor and Sample Time

In a given subsystem, if N_{st} is the streaming factor, and V_{dim} is the maximum vector dimension, then the data path of the resultant streamed subsystem is one of the following:

- Of width $V_{stream} = (V_{dim}/N_{st})$, if $V_{dim} > N_{st}$.
- Of width $V_{stream} = (N_{st}/V_{dim})$, if $N_{st} > V_{dim}$.
- Scalar.

If the original data path operated with a sample time, `S`, that is equal to the base sample time, then the streamed subsystem operates with a sample time of:

- S / N_{st}, if $V_{dim} > N_{st}$.
- S / V_{dim}, if $N_{st} > V_{dim}$.

If the original data path operated with a sample time, `S`, that is greater than the base sample time,S_{base}, then the streamed subsystem operates with a sample time of S_{base} / `Oversampling`. Notice that the streamed sample time is independent of the original sample time, `S`.

Determine Blocks That Support Streaming

HDL Coder supports many blocks for streaming. If you initiate streaming code generation for a subsystem that contains incompatible blocks, the coder works around those blocks and generates non-streaming code for them.

HDL Coder cannot apply the streaming optimization to a model reference.

Requirements for Streaming Subsystems

Before applying streaming, HDL Coder performs a series of checks on the subsystems to be streamed. You can stream a subsystem if:

- The streaming factor N_{st} is a perfect divisor of the vector width V_{dim}, or the vector width must be a perfect divisor of the streaming factor.
- All inputs to the subsystem have the same vector size. If the inputs have different vector sizes, you can stream the subsystem by flattening the subsystem hierarchy. When you flatten the hierarchy, the streaming optimization identifies regions with different vector sizes and creates streaming groups for these regions. These groups have different streaming factors that are inferred from the vector sizes.

Streaming Report

To see the streaming information in the report, before you generate code for each subsystem or model reference, enable the optimization report. To enable this report, in the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

When you generate an optimization report, in the **Streaming and Sharing** section, you see the effect of the streaming optimization. If streaming is unsuccessful, the report shows diagnostic messages and offending blocks that caused streaming to fail. When the requested streaming factor cannot be implemented, HDL Coder generates non-streaming code.

If streaming is successful, the report displays the **StreamingFactor** that was inferred, and a table that specifies:

- **Group**: A unique group ID for a group of Simulink blocks that belong to a streaming group.
- **Inferred Streaming Factor**: Streaming factor inferred by HDL Coder with the **Streaming Factor** that you specify in the HDL Block Properties.

To see groups of blocks that belong to a streaming group in your Simulink model and in the generated model, click the **Highlight streaming groups and diagnostics** link in the report.

See Also

More About

- “Resource Sharing” on page 21-45
- “Clock-Rate Pipelining” on page 21-148
- “Create and Use Code Generation Reports” on page 23-2

Resource Sharing

In this section...

"How Resource Sharing Works" on page 21-45

"Benefits and Costs of Resource Sharing" on page 21-46

"Shareable Resources in Different Blocks" on page 21-46

"Specify Resource Sharing" on page 21-46

"Block Requirements for Resource Sharing" on page 21-47

"Resource Sharing Report" on page 21-47

"Limitations for Resource Sharing" on page 21-47

Resource sharing is an area optimization in which HDL Coder identifies multiple functionally equivalent resources and replaces them with a single resource. The data is time-multiplexed over the shared resource to perform the same operations.

How Resource Sharing Works

You can specify a sharing factor for a subsystem or a MATLAB Function block. HDL Coder tries to identify a certain number of identical, shareable resources up to the value of the sharing factor. How HDL Coder shares these resources depends on the number of resources, the sharing factor, and the oversampling value for your design. To learn how to set an oversampling value for your design, see "Specifying the Oversampling Value" on page 20-9.

By default, the oversampling value is 1, and resource sharing overclocks the shared resources by an overclocking factor that depends on the remainder of the sharing factor and the number of resources. In this code, `OCF` is the overclocking factor, `N` is the number of sharable resources, and `SF` is the sharing factor:

```
if rem(SF,N) == 0
    OCF = N;
else
    OCF = SF;
end
```

If you specify an oversampling value greater than 1, either automatically by using the **Treat Simulink rates as actual hardware rates** parameter or manually by using the **Oversampling factor** parameter, your design operates a faster clock rate on the target hardware when clock-rate pipelining is enabled. When you specify a **SharingFactor**, the resource sharing optimization tries to share up to `N` resources and overclocks the shared resources by a factor given by:

Overclocking factor = (*block_rate* ÷ *DUT_base_rate*) × Oversampling value

You can use the validation model to verify that the output of the optimized DUT is bit-true to the results produced by the original DUT. To learn more about the validation model, see "Generated Model and Validation Model" on page 21-10.

Benefits and Costs of Resource Sharing

Resource sharing can substantially reduce your chip area. For example, the generated code can use one multiplier to perform the operations of several identically configured multipliers from the original model. However, resource sharing has the following costs:

- Uses more multiplexers and can use more registers.
- Reduces opportunities for distributed pipelining or retiming, because HDL Coder does not pipeline across clock rate boundaries.
- Multiplies the clock rate of the target hardware by the sharing factor.

Shareable Resources in Different Blocks

If you specify a nonzero sharing factor for a MATLAB Function block, HDL Coder identifies and shares functionally equivalent multipliers.

If you specify a nonzero sharing factor for a Subsystem, HDL Coder identifies and shares functionally equivalent instances of the following types of blocks:

- Gain
- Product
- Multiply-Add
- Add or Sum with two inputs
- Atomic Subsystem
- MATLAB Function

The code generator shares functionally equivalent MATLAB Function blocks with fixed-point types. When you use floating-point types or use the `MATLAB Datapath` architecture for MATLAB Function blocks with fixed-point types, HDL Coder treats the MATLAB Function block as a regular Subsystem. You can then share functionally equivalent resources inside the MATLAB Function block. To learn more, see “Use MATLAB Datapath Architecture for Sharing with MATLAB Function Blocks” on page 18-153.

Specify Resource Sharing

To specify resource sharing from the UI:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the subsystem, model reference, or MATLAB Function block and then click **HDL Block Properties**. In the **SharingFactor** field, enter the number of shareable resources.
- Right-click the subsystem, model reference, or MATLAB Function block and select **HDL Code > HDL Block Properties**. In the **SharingFactor** field, enter the number of shareable resources.

At the command-line, set the `SharingFactor` using `hdlset_param`, as in the following example.

```
modelname = 'sfir_fixed'
dut = 'sfir_fixed/symmetric_fir';
open_system(modelname)
hdlset_param(dut,'SharingFactor', 4);
```

Block Requirements for Resource Sharing

For blocks to be shared, they must meet these requirements:

- Single rate.
- The DUT must not contain blocks with `inf` sample time. During HDL code generation, HDL Coder only resolves `inf` sample times when the sample times do not propagate to the DUT output. For more information, see "Use Discrete and Finite Sample Time for Constant Block" on page 18-101.
- No bus inputs or outputs.
- No tunable mask parameters. To share these blocks, in the Mask Editor, clear the **Tunable** check box.
- If the block is within a feedback loop, at least one Unit Delay or Delay block must be connected to each output port.

To learn about block-specific settings and requirements for resource sharing, see:

- "Resource Sharing Settings for Various Blocks" on page 18-148
- "Resource Sharing of Subsystems and Floating-Point IPs" on page 18-152

Resource Sharing Report

To see the resource sharing information in the report, before you generate code for each subsystem or model reference, enable the optimization report. To enable this report, in the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

When you generate the optimization report, in the **Streaming and Sharing** section, you see the effect of the resource sharing optimization. If resource sharing is unsuccessful, the report shows diagnostic messages and offending blocks that cause resource sharing to fail.

If resource sharing is successful, the report displays the **SharingFactor**, and a table that contains groups of blocks that shared resources. The table contains:

- **Group Id**: A unique ID for a group of similar Simulink blocks, such as add or product blocks, that share resources.
- **Resource Type**: The type of Simulink block in a sharing group.
- **I/O Wordlengths**: Word lengths of inputs to and output from the block in a sharing group.
- **Group size**: Number of blocks of the same type in a sharing group.
- **Block name**: Name of a block that belongs to a sharing group.
- **Color Legend**: Color that highlights all the blocks in a sharing group.

To see the shared resources in your Simulink model and in the generated model, click the **Highlight shared resources and diagnostics** link.

Limitations for Resource Sharing

- Multirate sharing cannot share resources that have a different number of pipelines inserted from adaptive pipelining.
- Model references are not supported for resource sharing.

See Also

Related Examples

- Resource Sharing For Area Optimization on page 21-54
- “Single-Rate Resource Sharing Architecture” on page 21-65

More About

- “Clock-Rate Pipelining” on page 21-148
- “Streaming” on page 21-42
- “Create and Use Code Generation Reports” on page 23-2

Streaming: Area Optimization

This example shows how to use the subsystem level streaming optimization in HDL Coder™.

Introduction

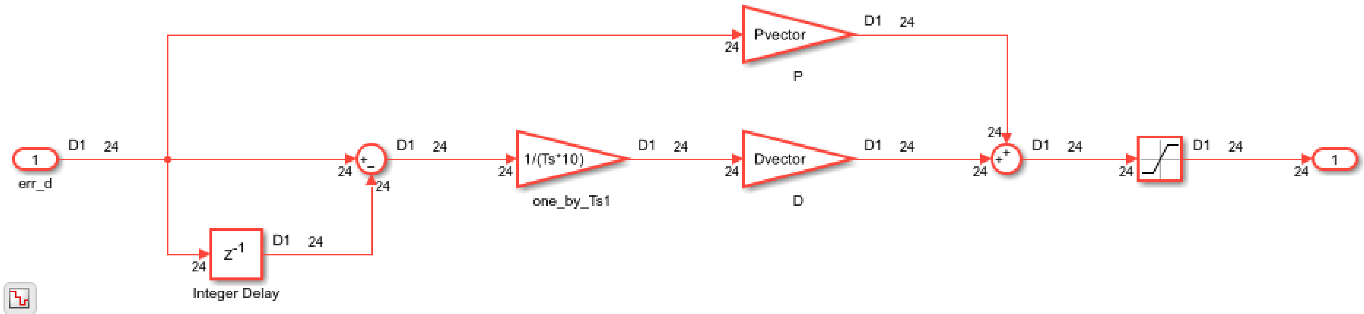
Streaming is a subsystem-wide optimization supported by HDL Coder for implementing area-efficient hardware. By default, the coder implements hardware that is bit-accurate and cycle-accurate to the Simulink® model. This implies that vector datapaths in Simulink map inefficiently to hardware. Consider a product block in Simulink that operates on two 64-element vector inputs and generates a 64-element vector output. This block executes 64 multiplications in a single Simulink time step. To remain cycle-accurate, HDL Coder maps this block to 64 parallel multipliers in the generated HDL code. Given that multipliers are expensive on FPGAs, this is an inefficient hardware implementation.

Streaming is an optimization that flattens a vector datapath to either a scalar or a smaller sized vector datapath. The idea is to serialize the execution of parallel hardware, so that resources can be shared and the vector data can be time-multiplexed over the shared resources.

Consider the following example model that operates on a 24-element vector datapath. This model contains three vector gains and two vector adds, resulting in a hardware implementation containing 72 multipliers and 24 adders. This can be confirmed by generating the resource utilization report when generating HDL code.

```
load_system('hdl_areaopt1');
open_system('hdl_areaopt1/Controller');
hdlset_param('hdl_areaopt1/Controller', 'StreamingFactor', 0);
hdlset_param('hdl_areaopt1', 'ResourceReport', 'on');
makehdl('hdl_areaopt1/Controller');
```

```
### Working on the model <a href="matlab:open_system('hdl_areaopt1')">hdl_areaopt1</a>
### Generating HDL for <a href="matlab:open_system('hdl_areaopt1/Controller')">hdl_areaopt1/Contr
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_areaopt1', {
### Running HDL checks on the model 'hdl_areaopt1'.
### Begin compilation of the model 'hdl_areaopt1'...
### Working on the model 'hdl_areaopt1'...
### Working on... <a href="matlab:configset.internal.open('hdl_areaopt1', 'GenerateModel')">Gener
### Begin model generation 'gm_hdl_areaopt1'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdl_areaopt1\gm_hdl_areaopt1.slx
### Begin VHDL Code Generation for 'hdl_areaopt1'.
### Working on hdl_areaopt1/Controller as hdlsrc\hdl_areaopt1\Controller.vhd.
### Generating package file hdlsrc\hdl_areaopt1\Controller_pkg.vhd.
### Code Generation for 'hdl_areaopt1' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdl_areaopt1' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```



Streaming to Scalarize the Datapath

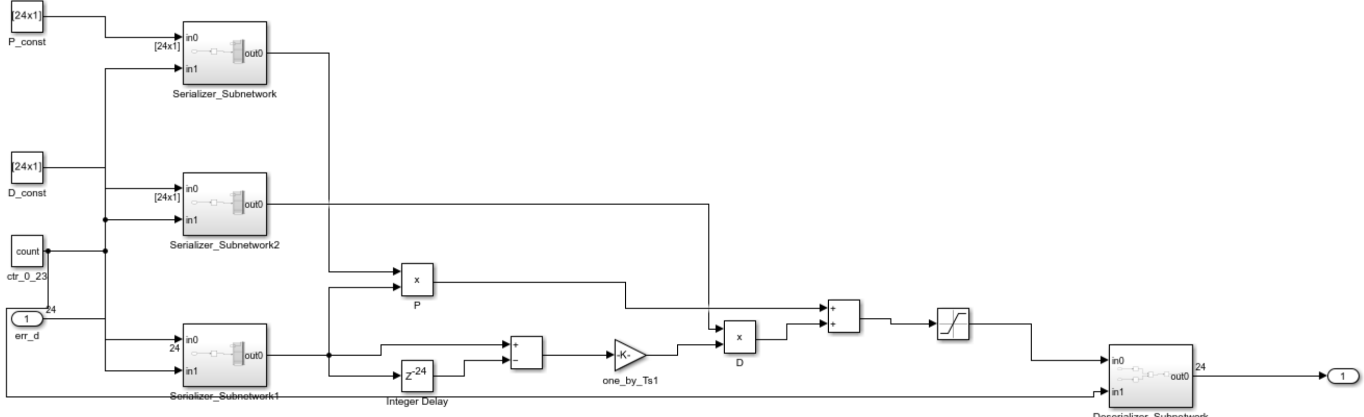
An efficient area implementation of the same model can be realized by setting a positive integer value to the `StreamingFactor` implementation parameter on the subsystem. This parameter specifies the extent to which the datapath is scalarized - the higher the value, the greater the area savings. In this example, we have a 24-element vector datapath; to fully scalarize it, specify a `StreamingFactor` value of 24. This can be done either through the HDL block properties dialog (opened by right-clicking on the `Controller` subsystem) or through the command `hdlset_param`.

Generating HDL code with `StreamingFactor` set to 24, generates HDL that uses only three multipliers and two adders (see the resource report after HDL code generation). The streamed architecture is implemented as local multi-rate or in single-rate mode depending on the context of the subsystem being streamed. If the subsystem logic is operating at a slower sample rate or if the Oversampling factor is set to a value greater than one, then clock-rate pipelining kicks in and a streamed subsystem is implemented as a multi-cycle, single-rate architecture. See “Single-Rate Resource Sharing Architecture” on page 21-65 for more details. In all other cases, a local multi-rate implementation is created, as described in this example. The elements of the vector datapath are streamed at a faster rate (in this case 24 times faster and denoted in red) and all computations operate on a scalar datapath. At the output, the vector is reconstructed using a deserializer and the output is sampled at the slower rate (as seen in the generated model in green).

```
hdlset_param('hdl_areaopt1/Controller', 'StreamingFactor', 24);
hdlset_param('hdl_areaopt1', 'GenerateValidationModel', 'on');
makehdl('hdl_areaopt1/Controller');
open_system('gm_hdl_areaopt1/Controller');
%set_param('gm_hdl_areaopt1', 'SimulationCommand', 'update');
```

```
### Working on the model <a href="matlab:open_system('hdl_areaopt1')">hdl_areaopt1</a>
### Generating HDL for <a href="matlab:open_system('hdl_areaopt1/Controller')">hdl_areaopt1/Cont
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_areaopt1', 
### Running HDL checks on the model 'hdl_areaopt1'.
### Begin compilation of the model 'hdl_areaopt1'...
### Working on the model 'hdl_areaopt1'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdl_areaopt1', 'GenerateModel')">Gene
### Begin model generation 'gm_hdl_areaopt1'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdl_areaopt1\gm_hdl_areaopt1.sl
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdl_areaopt1\gm_hdl_are
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdl_areaopt1'.
### MESSAGE: The design requires 24 times faster clock with respect to the base rate = 2.
```

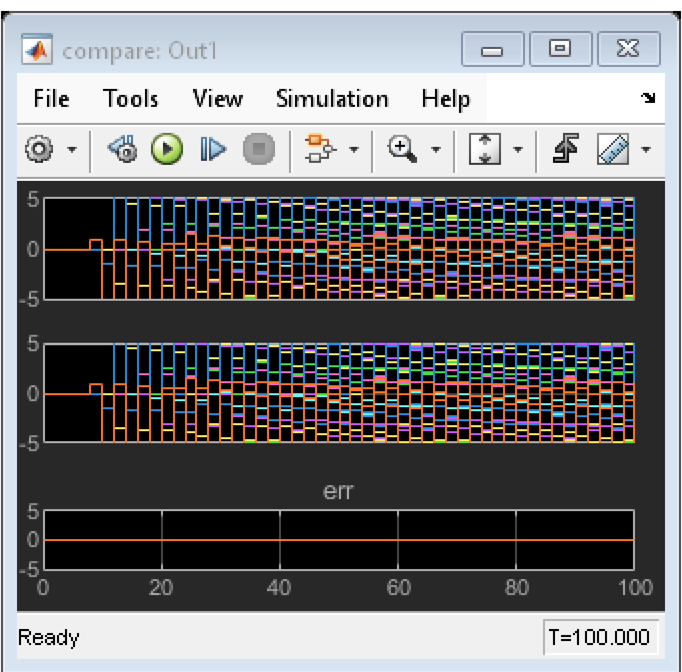
```
### Begin VHDL Code Generation for 'Controller_tc'.
### Working on Controller_tc as hdlsrc\hdl_areaopt1\Controller_tc.vhd.
### Code Generation for 'Controller_tc' completed.
### Working on hdl_areaopt1/Controller as hdlsrc\hdl_areaopt1\Controller.vhd.
### Generating package file hdlsrc\hdl_areaopt1\Controller_pkg.vhd.
### Code Generation for 'hdl_areaopt1' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/t
### HDL check for 'hdl_areaopt1' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```



Delay Balancing and Functional Equivalence

The rate transitions that implement time-multiplexing in the streaming architecture introduce a cycle of additional latency. To maintain functional fidelity, this delay must be balanced across all cut-sets that this path is a member of. When the streaming option is turned on, the coder automatically also turns on the delay balancing option (`BalanceDelays`) to automatically balance this additional delay. The coder also automatically turns on the validation model generation option so the user can verify that functional equivalence is maintained with respect to the original model.

```
sim('gm_hdl_areaopt1_vnl');
open_system('gm_hdl_areaopt1_vnl/Compare/Assert_Out1/compare: Out1')
```



Parameterizability for More Flexibility

By tuning the `StreamingFactor` parameter, one can explore the design space along the datapath size dimension. A value of 1 implies no streaming (or fully parallel implementation), and a value of 24 (or the full vector length) implies maximal streaming (or fully serial implementation). By picking values between these two extremes, one can explore the design space from fully parallel to fully serial implementations.

If we set `StreamingFactor` to 6 in this example model, we get a four-element vector datapath in the generated HDL. This results in the use of 12 multipliers and 8 adders as shown in the resource report.

```
hdlset_param('hdl_areaopt1/Controller', 'StreamingFactor', 6);
makehdl('hdl_areaopt1/Controller');
open_system('gm_hdl_areaopt1/Controller');
%set_param('gm_hdl_areaopt1', 'SimulationCommand', 'update');
```

```
### Working on the model <a href="matlab:open_system('hdl_areaopt1')">hdl_areaopt1</a>
### Generating HDL for <a href="matlab:open_system('hdl_areaopt1/Controller')">hdl_areaopt1/Cont
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_areaopt1', -
### Running HDL checks on the model 'hdl_areaopt1'.
### Begin compilation of the model 'hdl_areaopt1'...
### Working on the model 'hdl_areaopt1'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdl_areaopt1', 'GenerateModel')">Gene
### Begin model generation 'gm_hdl_areaopt1'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdl_areaopt1\gm_hdl_areaopt1.sl
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdl_areaopt1\gm_hdl_are
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdl_areaopt1'.
```

```
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 2.
### Begin VHDL Code Generation for 'Controller_tc'.
### Working on Controller_tc as hdlsrc\hdl_areaopt1\Controller_tc.vhd.
### Code Generation for 'Controller_tc' completed.
### Working on hdl_areaopt1/Controller as hdlsrc\hdl_areaopt1\Controller.vhd.
### Generating package file hdlsrc\hdl_areaopt1\Controller_pkg.vhd.
### Code Generation for 'hdl_areaopt1' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdl_areaopt1' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

Resource Sharing for Area Optimization

This example shows how to use the subsystem level sharing optimization in HDL Coder™.

Introduction

To implement area-efficient hardware, use resource sharing, which is a subsystem-level optimization supported by HDL Coder.

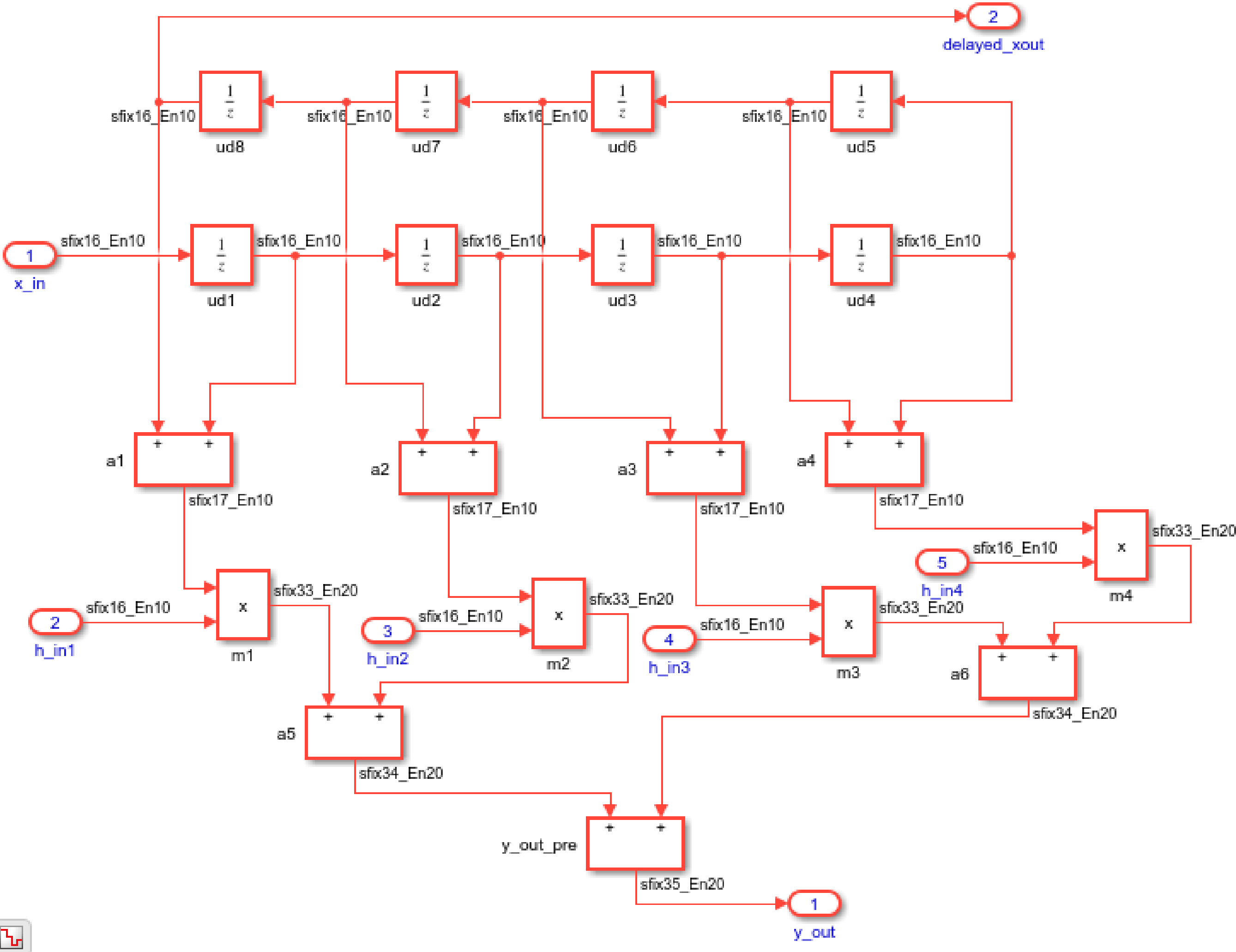
By default, HDL Coder implements hardware that is a one-to-one mapping of Simulink® blocks to hardware module implementations. The resource sharing optimization enables you to share hardware resources by enabling an N-to-1 mapping of `N` functionally equivalent Simulink blocks to a single hardware module. The user specifies `N` by using the `SharingFactor` implementation parameter.

Because a time-multiplexed architecture incurs a longer latency to complete the area optimization, HDL Coder manages the timing discrepancy depending on what resources are being shared. Suppose that the shared resources are operating at the base sample rate, and then resource sharing is implemented as a local multirate architecture, which is described in this example. If the shared resources are operating at a slower sample rate than the base sample rate, then HDL Coder invokes clock-rate pipelining to synthesize an implementation that uses the latency budget defined in the rate differential. In this case, the resource shared architecture is a single rate implementation and takes multiple time steps to complete all the shared operations. “Single-Rate Resource Sharing Architecture” on page 21-65 describes the details of this implementation.

This example also illustrates the local multirate architecture of resource sharing. Consider the following symmetric FIR filter model. It contains four product blocks that are functionally equivalent and that map to four multipliers in hardware. The Resource Utilization Report lists the hardware resources used.

```
load_system('sfir_fixed');
open_system('sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'ResourceReport', 'on');
makehdl('sfir_fixed/symmetric_fir');
```

```
### Working on the model <a href="matlab:open_system('sfir_fixed')">sfir_fixed</a>
### Generating HDL for <a href="matlab:open_system('sfir_fixed/symmetric_fir')">sfir_fixed/symme
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_fixed', {
### Running HDL checks on the model 'sfir_fixed'.
### Begin compilation of the model 'sfir_fixed'...
### Working on the model 'sfir_fixed'...
### Working on... <a href="matlab:configset.internal.open('sfir_fixed', 'GenerateModel')">Genera
### Begin model generation 'gm_sfir_fixed'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\sfir_fixed\gm_sfir_fixed.slx')">
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Code Generation for 'sfir_fixed' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/t
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```



Share Resources for an N-to-1 Mapping

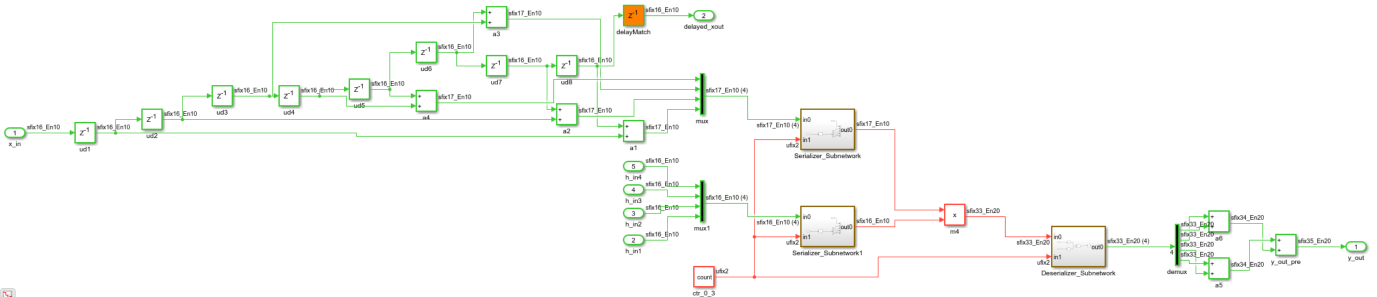
To reduce area resources, you can invoke the sharing optimization by setting the `SharingFactor` parameter on the subsystem to a positive integer value. This parameter specifies `N` in the N-to-1 hardware mapping. In this example, there are four product blocks, so generating HDL with `SharingFactor` set to 4 generates HDL code that has one multiplier.

The code generation model reflects the sharing architecture. The inputs to the shared blocks are time-multiplexed over the shared resource at a faster rate (in this case 4x faster, shown in red). The outputs are then routed to the respective consumers at a slower rate (shown in green).

```
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 4);
hdlset_param('sfir_fixed', 'GenerateValidationModel', 'on');
makehdl('sfir_fixed/symmetric_fir');
open_system('gm_sfir_fixed/symmetric_fir');
set_param('gm_sfir_fixed', 'SimulationCommand', 'update');
```

```
### Working on the model <a href="matlab:open_system('sfir_fixed')">sfir_fixed</a>
### Generating HDL for <a href="matlab:open_system('sfir_fixed/symmetric_fir')">sfir_fixed/symme
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_fixed', {
```

```
### Running HDL checks on the model 'sfir_fixed'.
### Begin compilation of the model 'sfir_fixed'...
### Working on the model 'sfir_fixed'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### Output port 2: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_fixed', 'GenerateModel')">Genera
### Begin model generation 'gm_sfir_fixed'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\sfir_fixed\gm_sfir_fixed.slx')">
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\sfir_fixed\gm_sfir_fixe
### Validation model generation complete.
### Begin VHDL Code Generation for 'sfir_fixed'.
### MESSAGE: The design requires 4 times faster clock with respect to the base rate = 1.
### Begin VHDL Code Generation for 'symmetric_fir_tc'.
### Working on symmetric_fir_tc as hdlsrc\sfir_fixed\symmetric_fir_tc.vhd.
### Code Generation for 'symmetric_fir_tc' completed.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Generating package file hdlsrc\sfir_fixed\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_fixed' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/t
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

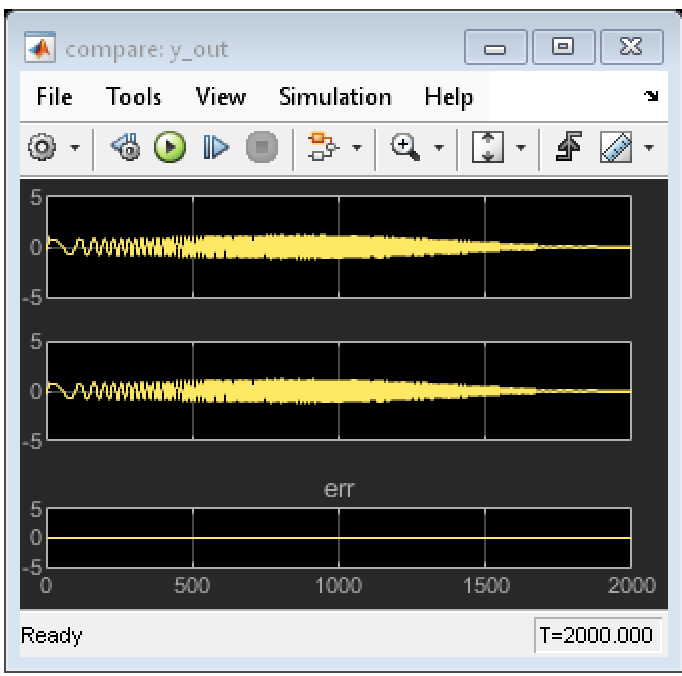


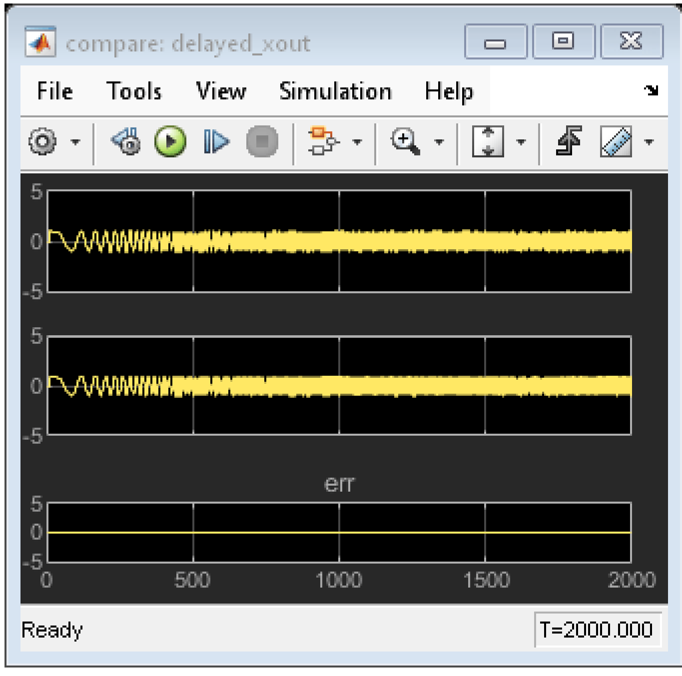
The sharing optimization is implemented by using time-division multiplexing. Simulink requires the outputs of the shared resource to be sampled at the predefined sample rate. HDL Coder overclocks the shared resource at a faster rate than the data rate. In the preceding example, the shared architecture, which includes the shared resources multiplexer-serializer at the inputs, and demultiplexer-deserializer at the outputs, operates at four times the rate of the input data, because the `Sharingfactor` is set to 4.

Delay Balancing and Functional Equivalence

The rate transitions that implement time-multiplexing in the resource sharing architecture introduce additional latency. To maintain functional equivalence, delay balancing inserts matching delays in parallel merging paths. The generated validation model enables you to verify functional equivalence by comparing the operation of the shared hardware architecture to the original model.

```
sim('gm_sfir_fixed_vnl');
open_system('gm_sfir_fixed_vnl/Compare/Assert_y_out/compare: y_out')
open_system('gm_sfir_fixed_vnl/Compare/Assert_delayed_xout/compare: delayed_xout')
```



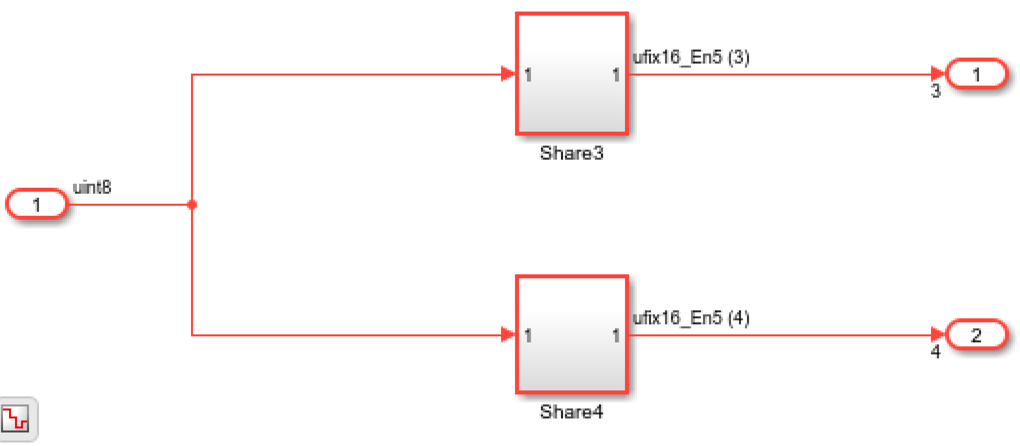


Control Multiplicative Oversampling Through SharingFactor

The net oversampling for the whole design is equivalent to the LCM of all `SharingFactor` values set on the model. Consider the example `hdlcoder_uniform_oversampling.slx`. It has two subsystems: subsystem `Share3` has three gain blocks that can be shared and `Share4` has four gain blocks that can be shared.

```
saved_warning_state = warning('off', 'hdlcoder:makehdl:DeprecateMaxOverSampling');
warning('off', 'hdlcoder:makehdl:DeprecateMaxComputationLatency');
```

```
load_system('hdlcoder_uniform_oversampling');
open_system('hdlcoder_uniform_oversampling/Subsystem');
set_param('hdlcoder_uniform_oversampling', 'SimulationCommand', 'update');
hdlsaveparams('hdlcoder_uniform_oversampling/Subsystem');

%% Set Model 'hdlcoder_uniform_oversampling' HDL parameters
hdlset_param('hdlcoder_uniform_oversampling', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_uniform_oversampling', 'HDLSubsystem', 'hdlcoder_uniform_oversampling');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_uniform_oversampling/Subsystem/Share3', 'SharingFactor', 3);

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_uniform_oversampling/Subsystem/Share4', 'SharingFactor', 4);
```



`Share3` sets its `SharingFactor` to 3 and `Share4` sets its `SharingFactor` to 4. HDL Coder applies local resource sharing to each subsystem and as a result, the HDL implementation requires LCM(3, 4)| = 12x oversampling. This oversampling is reported during HDL code generation.

```
makehdl('hdlcoder_uniform_oversampling/Subsystem');

### Working on the model <a href="matlab:open_system('hdlcoder_uniform_oversampling')">hdlcoder_u
### Generating HDL for <a href="matlab:open_system('hdlcoder_uniform_oversampling/Subsystem')">hd
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_unifor
### Running HDL checks on the model 'hdlcoder_uniform_oversampling'.
### Begin compilation of the model 'hdlcoder_uniform_oversampling'...
### Working on the model 'hdlcoder_uniform_oversampling'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### Output port 2: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_uniform_oversampling', 'Gene
### Begin model generation 'gm_hdlcoder_uniform_oversampling'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_uniform_oversampling\g
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoder_uniform_overs
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_uniform_oversampling'.
### MESSAGE: The design requires 12 times faster clock with respect to the base rate = 0.1.
### Begin VHDL Code Generation for 'Subsystem_tc'.
```

```
### Working on Subsystem_tc as hdlsrc\hdlcoder_uniform_oversampling\Subsystem_tc.vhd.
### Code Generation for 'Subsystem_tc' completed.
### Working on hdlcoder_uniform_oversampling/Subsystem/Share3 as hdlsrc\hdlcoder_uniform_oversamp
### Working on hdlcoder_uniform_oversampling/Subsystem/Share4 as hdlsrc\hdlcoder_uniform_oversamp
### Working on hdlcoder_uniform_oversampling/Subsystem as hdlsrc\hdlcoder_uniform_oversampling\Su
### Generating package file hdlsrc\hdlcoder_uniform_oversampling\Subsystem_pkg.vhd.
### Code Generation for 'hdlcoder_uniform_oversampling' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_uniform_oversampling' complete with 0 errors, 0 warnings, and 1 messa
### HDL code generation complete.
```

One way to circumvent this multiplicative effect of oversampling is to set the `SharingFactor` of all subsystems to the available oversampling budget. In the above example, if the oversampling budget is only 4x, then set `SharingFactor` set to 4 for both `Share3` and `Share4`. In this case, HDL Coder can share fewer resources than the `SharingFactor` and stay idle for the remaining cycles.

```
hdlset_param('hdlcoder_uniform_oversampling/Subsystem/Share3', 'SharingFactor', 4);
makehdl('hdlcoder_uniform_oversampling/Subsystem');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_uniform_oversampling')">hdlcoder_u
### Generating HDL for <a href="matlab:open_system('hdlcoder_uniform_oversampling/Subsystem')">hd
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_uniform
### Running HDL checks on the model 'hdlcoder_uniform_oversampling'.
### Begin compilation of the model 'hdlcoder_uniform_oversampling'...
### Working on the model 'hdlcoder_uniform_oversampling'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 1 cycles.
### Output port 2: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_uniform_oversampling', 'Gener
### Begin model generation 'gm_hdlcoder_uniform_oversampling'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_uniform_oversampling\gm
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoder_uniform_oversa
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_uniform_oversampling'.
### MESSAGE: The design requires 4 times faster clock with respect to the base rate = 0.1.
### Begin VHDL Code Generation for 'Subsystem_tc'.
### Working on Subsystem_tc as hdlsrc\hdlcoder_uniform_oversampling\Subsystem_tc.vhd.
### Code Generation for 'Subsystem_tc' completed.
### Working on hdlcoder_uniform_oversampling/Subsystem/Share3 as hdlsrc\hdlcoder_uniform_oversamp
### Working on hdlcoder_uniform_oversampling/Subsystem/Share4 as hdlsrc\hdlcoder_uniform_oversamp
### Working on hdlcoder_uniform_oversampling/Subsystem as hdlsrc\hdlcoder_uniform_oversampling\Su
### Generating package file hdlsrc\hdlcoder_uniform_oversampling\Subsystem_pkg.vhd.
### Code Generation for 'hdlcoder_uniform_oversampling' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_uniform_oversampling' complete with 0 errors, 0 warnings, and 1 messa
### HDL code generation complete.
```

The oversampling factor is now LCM(4,4) = 4. This value is the value reported during code generation. It is a best practice to set the `SharingFactor` values to the available oversampling budget. If your design contains fewer shareable resources than the `SharingFactor` value you specify, HDL Coder shares the shareable resources available and overclocks them by the `SharingFactor` value. If you want to apply resource sharing and other optimizations that use

overclocking, such as streaming, or apply resource sharing in multiple nested subsystems, this general guideline might result in a higher oversampling factor.

Block Support, Atomic Subsystems, and Extensions

HDL Coder supports resource sharing of four block types: Product, Gain, Atomic Subsystem, and MATLAB Function. For MATLAB Function blocks, use the `MATLAB Datapath` architecture with fixed-point types. This architecture is the default setting for floating-point types. You can specify the **HDL Architecture** in the HDL Block Properties dialog box of the MATLAB Function block.

Sharing functionally equivalent Product and Gain blocks means that the multipliers in the HDL implementation are shared. Two Product blocks are functionally equivalent if:

- The data types of their inputs and outputs are identical.
- Their block parameter settings are identical.
- Their HDL block properties are identical.

For a Gain block, functional equivalence additionally requires that the constant value and data types are also identical. If the gain constant data types are identical for two Gain blocks that have different gain constant values, HDL Coder can share them. Similarly, if a Gain block can be implemented as a Product block that has constant input, and it has the same data types as another Product block in the design, HDL Coder can share them.

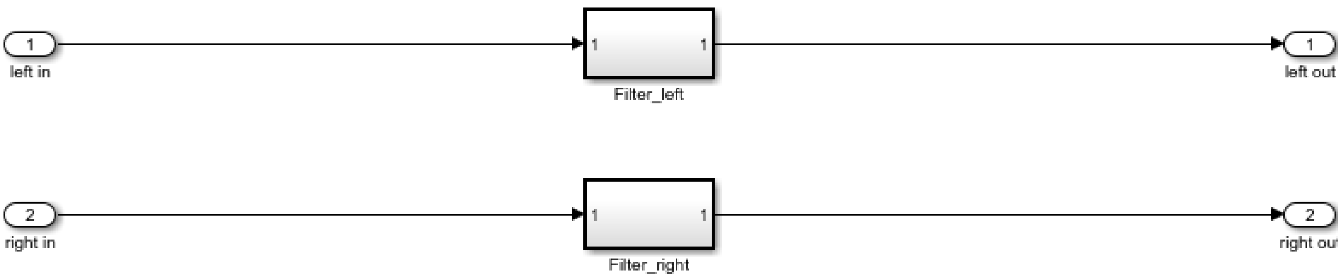
The third block type, Atomic Subsystem, is useful for sharing functionally equivalent logic encapsulated inside atomic subsystems. Two atomic subsystems are functionally equivalent and can be shared if:

- Their Simulink checksums are identical.
- Their HDL block properties are identical.

Share Atomic Subsystems

The following example demonstrates an audio filtering model that applies the same filter on the left and right channels. By default, HDL Coder generates two filter modules in hardware.

```
load_system('hdlcoder_audiofiltering');
open_system('hdlcoder_audiofiltering/Audio filter');
```

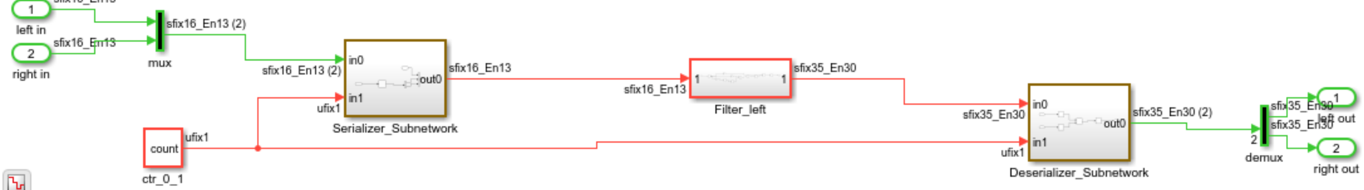


The filters on the two audio channels can be shared by specifying a `SharingFactor` value of 2 on the encompassing subsystem. This generates an architecture that uses only one filter, as shown below.

```
hdlset_param('hdlcoder_audiofiltering/Audio filter', 'SharingFactor', 2);
makehdl('hdlcoder_audiofiltering/Audio filter');
```

```
open_system('gm_hdlcoder_audiofiltering/Audio filter');
set_param('gm_hdlcoder_audiofiltering', 'SimulationCommand', 'update');
```

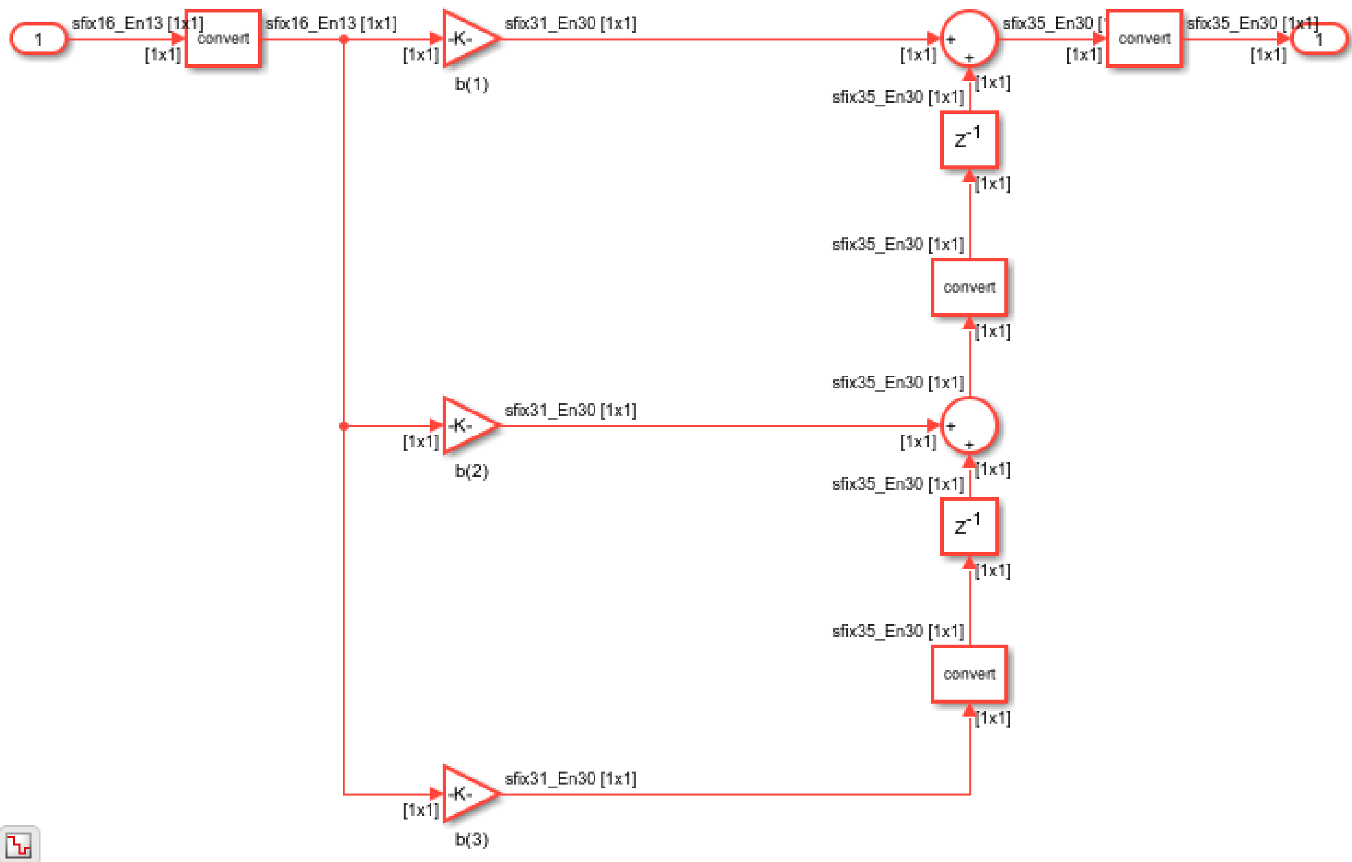
```
### Working on the model <a href="matlab:open_system('hdlcoder_audiofiltering')">hdlcoder_audiof
### Generating HDL for <a href="matlab:open_system('hdlcoder_audiofiltering/Audio filter')">hdlco
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_audiof
### Running HDL checks on the model 'hdlcoder_audiofiltering'.
### Begin compilation of the model 'hdlcoder_audiofiltering'...
### Working on the model 'hdlcoder_audiofiltering'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### Output port 2: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_audiofiltering', 'GenerateMod
### Begin model generation 'gm_hdlcoder_audiofiltering'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_audiofiltering\gm_hdlco
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### MESSAGE: The design requires 2 times faster clock with respect to the base rate = 0.00012207
### Begin VHDL Code Generation for 'Audio_filter_tc'.
### Working on Audio_filter_tc as hdlsrc\hdlcoder_audiofiltering\Audio_filter_tc.vhd.
### Code Generation for 'Audio_filter_tc' completed.
### Working on hdlcoder_audiofiltering/Audio filter/Filter_left as hdlsrc\hdlcoder_audiofiltering
### Working on hdlcoder_audiofiltering/Audio filter as hdlsrc\hdlcoder_audiofiltering\Audio_filte
### Generating package file hdlsrc\hdlcoder_audiofiltering\Audio_filter_pkg.vhd.
### Code Generation for 'hdlcoder_audiofiltering' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```



Opportunities to Resource Share Across Hierarchies

Because `SharingFactor` is a subsystem-level parameter, different subsystems at different levels of the hierarchy can specify different sharing values. In the audio filter example, the filters on each channel use three gain blocks respectively.

```
open_system('hdlcoder_audiofiltering/Audio filter/Filter_left');
```



Share Resources at Higher Level of Hierarchy

You can specify a `SharingFactor` value of 2 at the top-level of the DUT to share the filters on the two channels. You can specify a `SharingFactor` value of 3 on each of the filter subsystems to enable sharing of the three gain blocks in each channel. When HDL code is now generated, the left and right filters have been shared. Only one filter is at the top-level of the hierarchy.

```
hdlset_param('hdlcoder_audiofiltering/Audio filter', 'SharingFactor', 2);
hdlset_param('hdlcoder_audiofiltering/Audio filter/Filter_left', 'SharingFactor', 3);
hdlset_param('hdlcoder_audiofiltering/Audio filter/Filter_right', 'SharingFactor', 3);
makehdl('hdlcoder_audiofiltering/Audio filter');
open_system('gm_hdlcoder_audiofiltering/Audio filter');
set_param('gm_hdlcoder_audiofiltering', 'SimulationCommand', 'update');
```

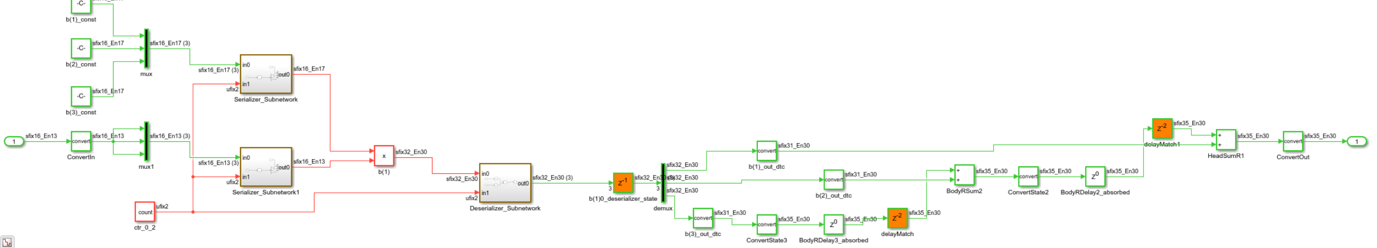
```
### Working on the model <a href="matlab:open_system('hdlcoder_audiofiltering')">hdlcoder_audiofi
### Generating HDL for <a href="matlab:open_system('hdlcoder_audiofiltering/Audio filter')">hdlco
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_audiofi
### Running HDL checks on the model 'hdlcoder_audiofiltering'.
### Begin compilation of the model 'hdlcoder_audiofiltering'...
### Working on the model 'hdlcoder_audiofiltering'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 2 cycles.
### Output port 2: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_audiofiltering', 'GenerateMod
### Begin model generation 'gm_hdlcoder_audiofiltering'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
```

```
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_audiofiltering\gm_hdlco
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 0.00012207
### Begin VHDL Code Generation for 'Audio_filter_tc'.
### Working on Audio_filter_tc as hdlsrc\hdlcoder_audiofiltering\Audio_filter_tc.vhd.
### Code Generation for 'Audio_filter_tc' completed.
### Working on hdlcoder_audiofiltering/Audio filter/Filter_left as hdlsrc\hdlcoder_audiofiltering
### Working on hdlcoder_audiofiltering/Audio filter as hdlsrc\hdlcoder_audiofiltering\Audio_filte
### Generating package file hdlsrc\hdlcoder_audiofiltering\Audio_filter_pkg.vhd.
### Code Generation for 'hdlcoder_audiofiltering' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

Share Resources at Lower Level of Hierarchy

The three gain blocks in the lower level of the hierarchy (within the filter subsystem) have also been shared. You have reduced the resource usage from six multipliers to just one.

```
open_system('gm_hdlcoder_audiofiltering/Audio filter/Filter_left');
set_param('gm_hdlcoder_audiofiltering', 'SimulationCommand', 'update');
```

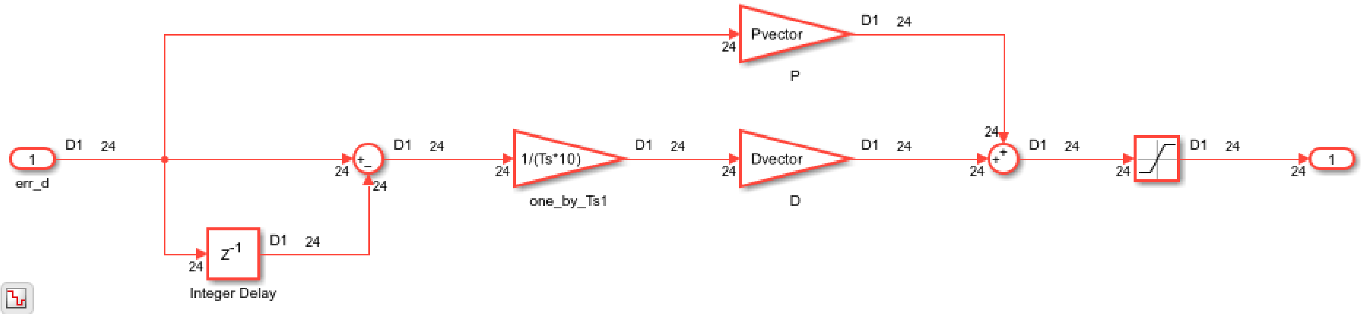


Combine Optimizations and Sharing

Resource sharing can be combined with other optimizations, such as the streaming optimization.

Consider this model. it contains a 24-element vector datapath and three vector gain blocks, which map to 72 multipliers, by default. Streaming can scalarize the vector datapath while sharing can share the three Gain blocks.

```
load_system('hdl_areaopt1');
open_system('hdl_areaopt1/Controller');
set_param('hdl_areaopt1', 'SimulationCommand', 'update');
```

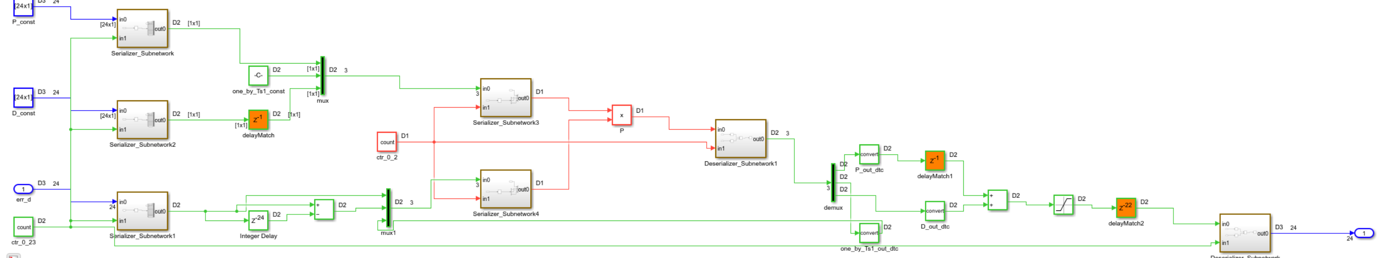


Streaming and Sharing to Reduce Design to Use One Multiplier

To invoke both optimizations, set `StreamingFactor` to 24 and `SharingFactor` to 3. The former reduces the number of multipliers from 72 to 3. The latter reduces the 3 scalar multipliers to 1.

```
hdlset_param('hdl_areaopt1/Controller', 'StreamingFactor', 24);
hdlset_param('hdl_areaopt1/Controller', 'SharingFactor', 3);
makehdl('hdl_areaopt1/Controller');
open_system('gm_hdl_areaopt1/Controller');
set_param('gm_hdl_areaopt1', 'SimulationCommand', 'update');
```

```
### Working on the model <a href="matlab:open_system('hdl_areaopt1')">hdl_areaopt1</a>
### Generating HDL for <a href="matlab:open_system('hdl_areaopt1/Controller')">hdl_areaopt1/Contr
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_areaopt1', {
### Running HDL checks on the model 'hdl_areaopt1'.
### Begin compilation of the model 'hdl_areaopt1'...
### Working on the model 'hdl_areaopt1'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('hdl_areaopt1', 'GenerateModel')">Gener
### Begin model generation 'gm_hdl_areaopt1'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdl_areaopt1\gm_hdl_areaopt1.slx
### Begin VHDL Code Generation for 'hdl_areaopt1'.
### MESSAGE: The design requires 72 times faster clock with respect to the base rate = 2.
### Begin VHDL Code Generation for 'Controller_tc'.
### Working on Controller_tc as hdlsrc\hdl_areaopt1\Controller_tc.vhd.
### Code Generation for 'Controller_tc' completed.
### Working on hdl_areaopt1/Controller as hdlsrc\hdl_areaopt1\Controller.vhd.
### Generating package file hdlsrc\hdl_areaopt1\Controller_pkg.vhd.
### Code Generation for 'hdl_areaopt1' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdl_areaopt1' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```



Single-Rate Resource Sharing Architecture

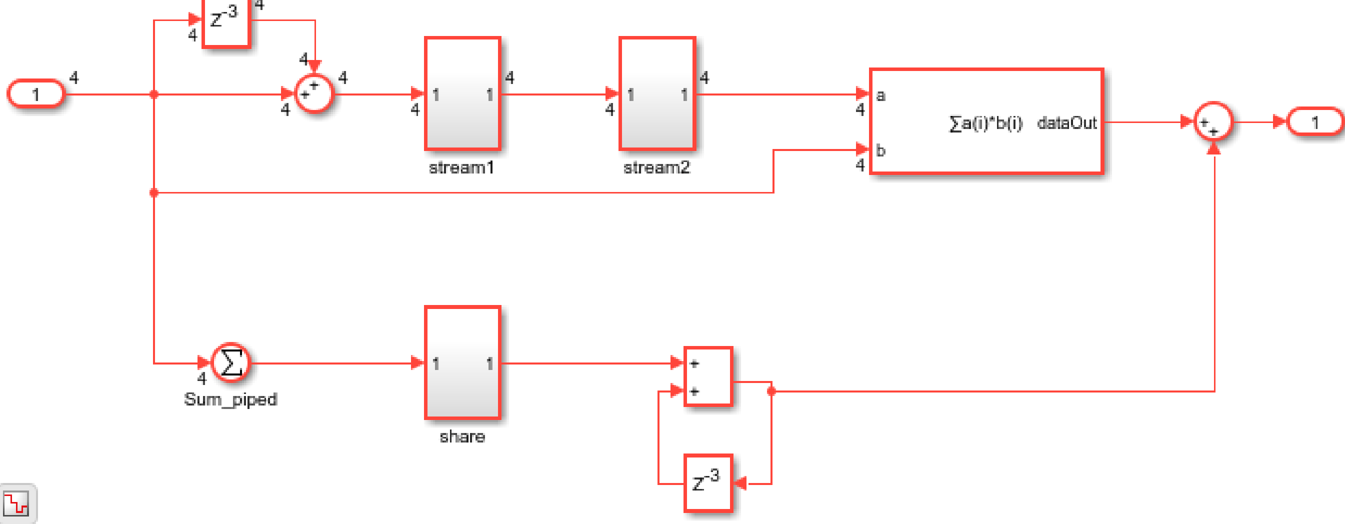
This example shows how HDL Coder™ generates a single-rate architecture from a model to manage the execution of operations in the context of clock-rate pipelining and resource sharing. When you apply resource sharing to blocks in your design, HDL Coder decides the architecture to use based on the sample rate of the regions where sharing occurs. If resource sharing occurs in a region of the design that operates at the fastest base sample rate, then HDL Coder synthesizes a local multi-rate architecture, as described in "Resource Sharing for Area Optimization" on page 21-54. In this example, the shared resources are in a region that operates at a slower sample rate, so HDL Coder applies clock-rate pipelining and synthesizes a single-rate architecture.

Clock-rate pipelining is an optimization that finds separate regions of logic in a design that operate on data at a slower sample rate and inserts pipelining and resource sharing logic at the faster clock rate. In these cases, resource sharing is implemented as a time-multiplexed architecture that operates at a single rate and incurs a latency. In order to execute dependent operations and manage the additional latency, HDL Coder synthesizes appropriate scheduling logic.

Inspect Model

Open and update the model `hdlcoder_singlerate_sharing`.

```
load_system('hdlcoder_singlerate_sharing');
open_system('hdlcoder_singlerate_sharing/Subsystem');
set_param('hdlcoder_singlerate_sharing', 'SimulationCommand', 'update');
```



This model has an oversampling constraint defined by the Oversampling factor parameter, which specifies how much faster the FPGA clock rate runs with respect to the Simulink base sample time. This model sets the oversampling factor to `30` by using the command-line to set the property `Oversampling`, which means that the clock-rate pipelined region can consume 30 clock cycles to complete execution.

Generate HDL Code and Inspect Validation Model

List the optimization options for individual blocks and subsystems by using the function `hdlsaveparams`. Generate code and a validation model to see single-rate sharing architecture.

```
hdlsaveparams('hdlcoder_singlerate_sharing/Subsystem');
makehdl('hdlcoder_singlerate_sharing/Subsystem');
```

```
%% Set Model 'hdlcoder_singlerate_sharing' HDL parameters
hdlset_param('hdlcoder_singlerate_sharing', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_singlerate_sharing', 'HDLSubsystem', 'hdlcoder_singlerate_sharing');
hdlset_param('hdlcoder_singlerate_sharing', 'Oversampling', 30);

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem', 'DistributedPipelining', 'on');

hdlset_param('hdlcoder_singlerate_sharing/Subsystem/Sum_piped', 'Architecture', 'Tree');
% Set Sum HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/Sum_piped', 'OutputPipeline', 2);

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/share', 'SharingFactor', 2);

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/stream1', 'StreamingFactor', 2);

% Set Delay HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/stream1/Delay', 'UseRAM', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/stream2', 'StreamingFactor', 4);

### Begin compilation of the model 'hdlcoder_singlerate_sharing'...
### Working on the model <a href="matlab:open_system('hdlcoder_singlerate_sharing')">hdlcoder_si
### Generating HDL for <a href="matlab:open_system('hdlcoder_singlerate_sharing/Subsystem')">hdl
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_single
### Running HDL checks on the model 'hdlcoder_singlerate_sharing'.
### Begin compilation of the model 'hdlcoder_singlerate_sharing'...
### Working on the model 'hdlcoder_singlerate_sharing'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_singlerate_sharing', 'Genera
### Begin model generation 'gm_hdlcoder_singlerate_sharing'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_singlerate_sharing\gm_
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcod
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoder_singlerate_sh
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_singlerate_sharing'.
### MESSAGE: The design requires 30 times faster clock with respect to the base rate = 0.1.
### Begin VHDL Code Generation for 'Subsystem_tc'.
### Working on Subsystem_tc as hdlsrc\hdlcoder_singlerate_sharing\Subsystem_tc.vhd.
### Code Generation for 'Subsystem_tc' completed.
### Working on hdlcoder_singlerate_sharing/Subsystem/share as hdlsrc\hdlcoder_singlerate_sharing
### Working on hdlcoder_singlerate_sharing/Subsystem/stream1 as hdlsrc\hdlcoder_singlerate_shari
### Working on hdlcoder_singlerate_sharing/Subsystem/stream2 as hdlsrc\hdlcoder_singlerate_shari
```

```
### Working on crp_temp_MAC as hdlsrc\hdlcoder_singlerate_sharing\crp_temp_MAC.vhd.
### Working on hdlcoder_singlerate_sharing/Subsystem as hdlsrc\hdlcoder_singlerate_sharing\Subsys
### Generating package file hdlsrc\hdlcoder_singlerate_sharing\Subsystem_pkg.vhd.
### Code Generation for 'hdlcoder_singlerate_sharing' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_singlerate_sharing' complete with 0 errors, 3 warnings, and 4 message
### HDL code generation complete.
```

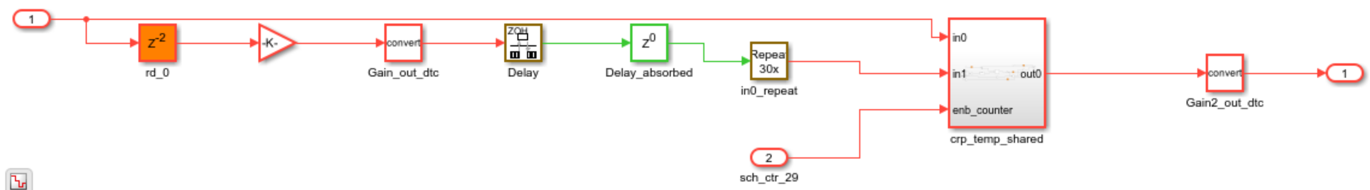
At the global level in the generated validation model, HDL Coder schedules each of these locally shared and streamed subsystems according to their latency. The unit of scheduling is a clock-rate pipelined region that has been automatically identified by HDL Coder. For each region, a Counter Limited block is used as a sequencer for the scheduling logic. In order to avoid generating identical counters, HDL Coder generates one global scheduling counter per model for all clock-rate pipelining regions that require the same size counter. The counter counts from zero to $clockratebudget - 1$, where the budget is the ratio of the shared resource sample rate to the FPGA clock rate.

In this example, the budget is 30 because the `Oversampling` property is `30`. HDL Coder assigns a time interval that each streamed and shared subsystem executes within. The subsystem is encapsulated in an enabled subsystem so that it is only active during that time interval. The counter or sequencer value specifies the current time step, and the logic that computes the time interval drives the enable inputs to these subsystems.

For each subsystem in the validation model that specifies resource sharing or streaming, a single-rate resource-shared architecture implements the time-division multiplexing. For example, see the validation model subsystem `gm_hdlcoder_singlerate_sharing_vnl/Subsystem/share`. If `SharingFactor` = `N`, it takes `N-1` cycles to execute the shared architecture per cycle of the original computation.

Open the validation model subsystem `gm_hdlcoder_singlerate_sharing_vnl/Subsystem/share`.

```
open_system('gm_hdlcoder_singlerate_sharing_vnl/Subsystem/share');
set_param('gm_hdlcoder_singlerate_sharing_vnl', 'SimulationCommand', 'update');
```



The subsystem `gm_hdlcoder_singlerate_sharing_vnl/Subsystem/share` is assigned the time interval `[2, 3]` because the Sum of Elements block, `hdlcoder_singlerate_sharing/Subsystem/Sum_piped` has the HDL block property **OutputPipeline** set to `2` and is on the path between the DUT inputs and the inputs to this subsystem. The shared subsystem starts execution in time step two, and because the resource sharing property `SharingFactor` is set to `3`, takes $3 - 1 = 2$ cycles to complete. The enable input to `gm_hdlcoder_singlerate_sharing_vnl/Subsystem/share/crp_temp_shared` is asserted only when the global counter is greater than or equal to two and less than or equal to three.

In addition to streamed and shared subsystems, HDL Coder also schedules any blocks or subsystems that contain state or implement multi-cycle operations. For example, the design uses Multiply-Accumulate block in the `gm_hdlcoder_singlerate_sharing_vnl/Subsystem/crp_temp_MAC`

subsystem that computes the dot-product on two four-element vectors. This logic takes four cycles to execute and is scheduled in the time interval `[4, 7]` because there are two streaming regions on the path from the inputs to the Multiply-Accumulate block. The first streaming region, `gm_hdlcoder_singlerate_sharing_vnl/Subsystem/stream1`, is scheduled in time interval `[0, 1]` due to a streaming factor of two. The second streaming region, `gm_hdlcoder_singlerate_sharing_vnl/Subsystem/stream2`, is scheduled in time interval `[1, 4]` due to a streaming factor of four.

The generated validation model has non-trivial changes from the original model, but models the single-rate sharing architecture that HDL Coder synthesizes and generates for the HDL code. This model also compares the numerics of the synthesized architecture with the original model except for differences accounted for by added latency. For more details, see “Delay Balancing and Validation Model Workflow in HDL Coder” on page 21-94. Run the validation model to compare the numerics between the generated and the original model in each time step. The model throws an error on mismatches.

See Also

Related Examples

- “Resource Sharing for Area Optimization” on page 21-54
- “Increase Clock Frequency Using Clock-Rate Pipelining” on page 21-153

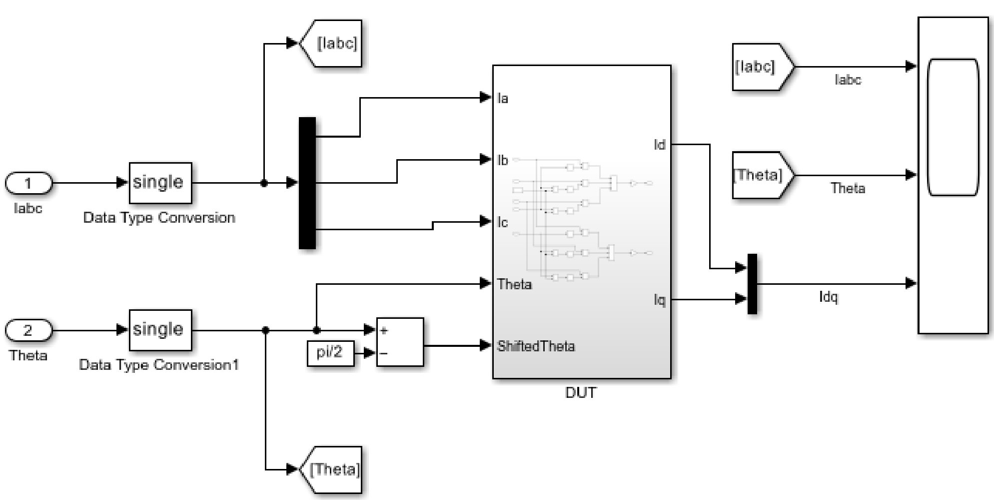
Improve Resource Sharing with Design Modifications

This example shows how to improve opportunities for resource sharing to optimize your model design by making certain modifications to your design. Resource sharing is an HDL Coder optimization that improves area utilization in the design on the target FPGA device. The optimization identifies multiple functionally equivalent resources and replaces them with a single resource. For more information, see “Resource Sharing” on page 21-45.

Analyze the Current Model

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
design_name = 'hdlcoderParkTransform';
design_new_name = 'hdlcoderParkTransformCopy';

copyfile(fullfile([design_name,'.slx']), fullfile([design_new_name,'.slx']), 'f');


% Open the model.
open_system(design_new_name);
```



Copyright 2020-2021 The MathWorks, Inc.

```
% Set a SharingFactor of 6 on the subsystem of interest and generate
% HDL along with the corresponding reports from the model.
subsystem = [design_new_name '/DUT'];
hdlset_param(subsystem, 'SharingFactor', 6);
makehdl(subsystem);
```

```
### Working on the model <a href="matlab:open_system('hdlcoderParkTransformCopy')">hdlcoderParkT
### Generating HDL for <a href="matlab:open_system('hdlcoderParkTransformCopy/DUT')">hdlcoderPar
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderParkTra
### Running HDL checks on the model 'hdlcoderParkTransformCopy'.
### Begin compilation of the model 'hdlcoderParkTransformCopy'...
### Working on the model 'hdlcoderParkTransformCopy'...
```

```
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 42 cycles.
### Output port 2: 42 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoderParkTransformCopy', 'GenerateM
### Begin model generation 'gm_hdlcoderParkTransformCopy'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoderParkTransformCop
### Begin VHDL Code Generation for 'hdlcoderParkTransformCopy'.
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 1e-05.
### Begin VHDL Code Generation for 'DUT_tc'.
### Working on DUT_tc as hdl_prj\hdlsrc\hdlcoderParkTransformCopy\DUT_tc.vhd.
### Code Generation for 'DUT_tc' completed.
### Working on hdlcoderParkTransformCopy/DUT/nfp_sin_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_add2_single as hdl_prj\hdlsrc\hdlcoderParkTrans
### Working on hdlcoderParkTransformCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT as hdl_prj\hdlsrc\hdlcoderParkTransformCopy\DUT.vhd
### Generating package file hdl_prj\hdlsrc\hdlcoderParkTransformCopy\DUT_pkg.vhd.
### Code Generation for 'hdlcoderParkTransformCopy' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoderParkTransformCopy' complete with 0 errors, 0 warnings, and 1 messages
### HDL code generation complete.
```

In the generated Streaming and Sharing report, notice that there are specific groups that were identified to be eligible for sharing:

Resource Type	Group Size	Block Name
Trigonometry	6	sine
Sum	2	AddSub2
Sum	2	AddSub1
Product	6	Product
Product	2	Gain

The Native Floating-Point Resource Report shows the operators needed:

Resource	Usage
Adders	4
Multipliers	2
Sin	1
Subtractors	2

The High-level Resource Report shows that the resource utilization for the design was estimated to be:

Resource	Usage
Multipliers	9
Adders/Subtractors	152
Registers	1121
Total 1-Bit Registers	15896

```
RAMs                        0
Multiplexers                857
I/O Bits                    228
Static Shift operators      5
Dynamic Shift operators     13
```

You can also run synthesis workflow on the model. The synthesis results for this model are:

```
        Resource             Usage
_________________________   ________
Slice LUTs                  29381
Slice Registers             24140
DSPs                        50
Block Ram Tile              0
URAM                        0
```

For more details regarding synthesis workflow, see "HDL Code Generation and FPGA Synthesis from Simulink Model".

Detect Patterns in Model

Upon further analysis of the model, you can see that there are a few patterns that can be modified to improve resource sharing.

The first such pattern is with the Add and Subtract blocks whose names start with `AddSub` and whose output goes to Sin blocks.

The second pattern corresponds to a Sum (as an addition or subtraction) block followed by a Gain block.

In both of these patterns, there are some dissimilarities in the highlighted blocks that prevent resource sharing from happening optimally. The dissimilarities are as follows:

First, the signs of the Sum blocks in the region near the Gain blocks are different. Going from top to bottom, the signs are `+++` and `---`.

You can fix this inconsistency with the following changes:

1. Switch the signs of the bottom Sum block from `---` to `+++`
2. Make the three bottom Sin blocks to Cos blocks.
3. Change the input to the Cos block and the two bottom Sum blocks from `Inport 5 ShiftedTheta` to `Inport 4 Theta`.

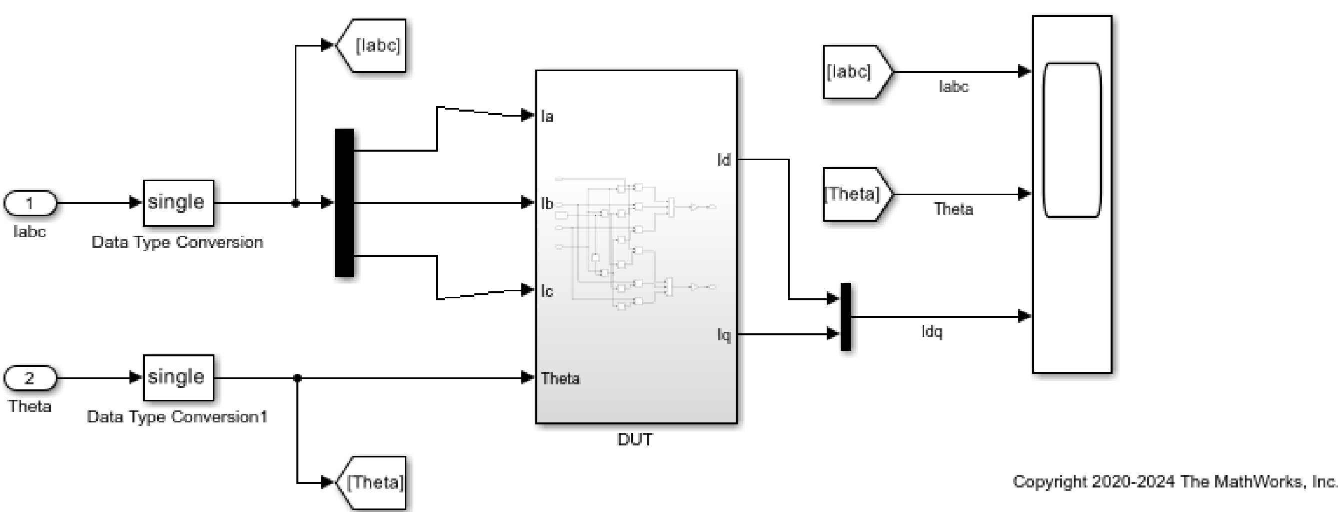
Second, the signs of the Sum blocks in the different regions are different. Going from the top to bottom, the signs are `+-`, `++`, `+-` and `++`.

The blocks with both positive signs are being shared separately from the blocks with alternating signs. This separation between the blocks with both positive signs and the blocks with alternating signs requires more resources. You can simplify your model further to reduce the number of Sum blocks in your model while improving resource sharing. The input to the Sum blocks are the same, and you can reduce the number of Sum blocks you need to two. Then, the inconsistency between the signs is easily fixed by introducing a Unary Minus block.

The discrepancy is then resolved, and this model now shares more resources than the original.

This edited model is saved as `hdlcoderParkTransformShare.slx`. Copy this version of the model into the temporary directory using the following commands:

```
design_name = 'hdlcoderParkTransformShare';
design_new_name = 'hdlcoderParkTransformShareCopy';

copyfile(fullfile([design_name,'.slx']), fullfile([design_new_name,'.slx']), 'f');


% Open the model.
open_system(design_new_name);
```



Compare Results from Optimized Model

Since this model has been improved for resource sharing, more blocks are shared and results in an improvement in resource utilization. Set a **SharingFactor** of 3 on the subsystem of interest and generate HDL along with the corresponding reports from the optimized model

```
subsystem = [design_new_name '/DUT'];

hdlset_param(subsystem, 'SharingFactor', 6);
makehdl(subsystem);
```

```
### Working on the model <a href="matlab:open_system('hdlcoderParkTransformShareCopy')">hdlcoderP
### Generating HDL for <a href="matlab:open_system('hdlcoderParkTransformShareCopy/DUT')">hdlcode
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderParkTran
### Running HDL checks on the model 'hdlcoderParkTransformShareCopy'.
### Begin compilation of the model 'hdlcoderParkTransformShareCopy'...
### Working on the model 'hdlcoderParkTransformShareCopy'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 38 cycles.
### Output port 2: 38 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoderParkTransformShareCopy', 'Gene
### Begin model generation 'gm_hdlcoderParkTransformShareCopy'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoderParkTransformSha
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdl_
```

```
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsrc
### Generating new validation model: '<a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoderParkTra
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderParkTransformShareCopy'.
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 6.
### Begin VHDL Code Generation for 'DUT_tc'.
### Working on DUT_tc as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_tc.vhd.
### Code Generation for 'DUT_tc' completed.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sincos_single as hdl_prj\hdlsrc\hdlcoderPar
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_uminus_single as hdl_prj\hdlsrc\hdlcoderPar
### Working on hdlcoderParkTransformShareCopy/DUT as hdl_prj\hdlsrc\hdlcoderParkTransformShareCop
### Generating package file hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_pkg.vhd.
### Code Generation for 'hdlcoderParkTransformShareCopy' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoderParkTransformShareCopy' complete with 0 errors, 0 warnings, and 2 mess
### HDL code generation complete.
```

After making those design modifications, the Native Floating-Point Resource Report shows the updated operators needed:

Resource	Usage
Adders	2
Multipliers	2
SinCos	1
Subtractors	1
Unary Minus	1

Notice that the **Adders**, **Multipliers**, and **Subtractors** count have reduced after sharing.

After sharing, the High-level Resource Report shows that the resource utilization for the design was estimated to be:

Resource	Usage
Multipliers	13
Adders/Subtractors	178
Registers	1012
Total 1-Bit Registers	16888
RAMs	0
Multiplexers	610
I/O Bits	196
Static Shift operators	8
Dynamic Shift operators	8

After running synthesis on the model, you can see the following results:

Resource	Usage
Slice LUTs	8792
Slice Registers	8419
DSPs	13
Block Ram Tile	0
URAM	0

Notice that the **Slice LUTs**, **Slice Registers**, and **DSPs** counts have reduced after sharing.

To learn how you can further improve resource sharing with automatically replacement of recurring patterns using the Clone Detection application, see “Improve Resource Sharing with Clone Detection and Replacement” on page 21-75.

Improve Resource Sharing with Clone Detection and Replacement

This example shows how you can automatically identify and replace recurring patterns by using the Clone Detector App. Consequently, it also improves opportunities for optimizing the model by using resource sharing. The Clone Detector App refactors models by identifying clones in a design and replacing clones with links to subsystem blocks in a library. This example uses a model that has already been modified for optimal resource sharing by using the Clone Detector App. To learn more about these modifications, see "Improve Resource Sharing with Design Modifications."

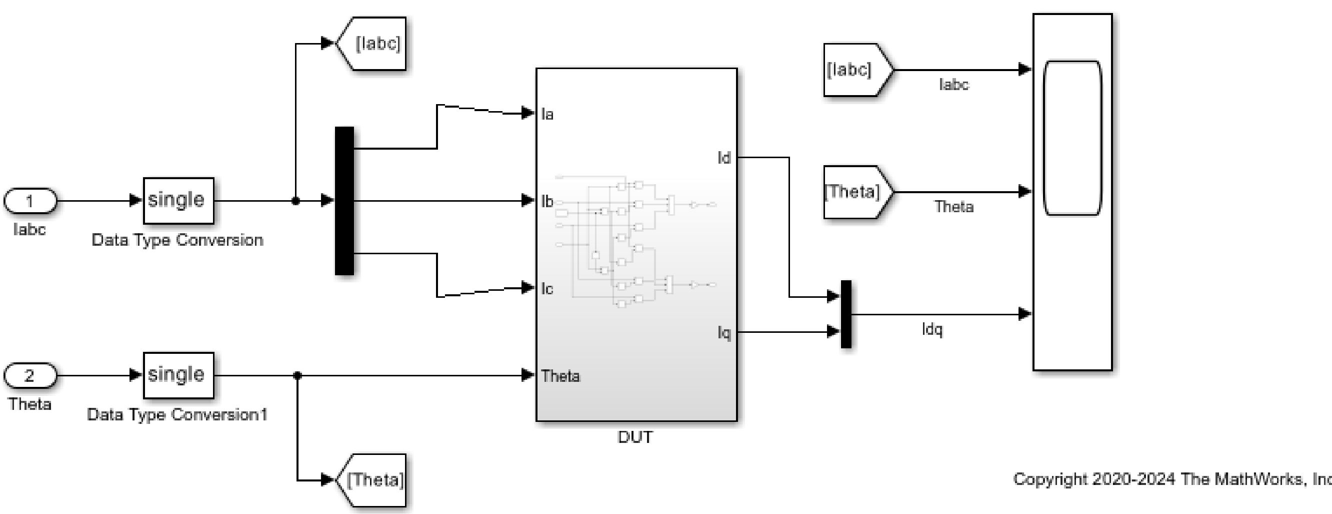
Clone Detection

Clone Detection is a tool that can be used to identify modeling patterns in a design that are similar to a few sample patterns provided as inputs through a custom library file. Once the Clone Detector App identifies these patterns, it replaces the patterns with atomic subsystems. This enables the generation of optimal code through code reuse and results in better resource utilization through sharing of resources among the atomic subsystems. For more information regarding Clone Detection, please refer to “Enable Component Reuse by Using Clone Detection” (Simulink Check).

Set Up Model for Resource Sharing

This edited model is saved as hdlcoderParkTransformShare.slx. Copy this version of the model into the temporary directory using the following commands:

```
design_name = 'hdlcoderParkTransformShare';
design_new_name = 'hdlcoderParkTransformShareCopy';

copyfile(fullfile([design_name,'.slx']), fullfile([design_new_name,'.slx']), 'f');

% Open the model.
open_system(design_new_name);
```



Set a **SharingFactor** of 6 on the subsystem of interest and generate HDL along with the corresponding reports from the model.

```
subsystem = [design_new_name '/DUT'];
hdlset_param(subsystem, 'SharingFactor', 6);
makehdl(subsystem);

### Working on the model <a href="matlab:open_system('hdlcoderParkTransformShareCopy')">hdlcoderP
### Generating HDL for <a href="matlab:open_system('hdlcoderParkTransformShareCopy/DUT')">hdlcode
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderParkTran
### Running HDL checks on the model 'hdlcoderParkTransformShareCopy'.
### Begin compilation of the model 'hdlcoderParkTransformShareCopy'...
### Working on the model 'hdlcoderParkTransformShareCopy'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 38 cycles.
### Output port 2: 38 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoderParkTransformShareCopy', 'Gene
### Begin model generation 'gm_hdlcoderParkTransformShareCopy'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoderParkTransformSha
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdl_
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsrc
### Generating new validation model: '<a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoderParkTra
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderParkTransformShareCopy'.
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 6.
### Begin VHDL Code Generation for 'DUT_tc'.
### Working on DUT_tc as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_tc.vhd.
### Code Generation for 'DUT_tc' completed.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sincos_single as hdl_prj\hdlsrc\hdlcoderPar
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_uminus_single as hdl_prj\hdlsrc\hdlcoderPar
### Working on hdlcoderParkTransformShareCopy/DUT as hdl_prj\hdlsrc\hdlcoderParkTransformShareCop
### Generating package file hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_pkg.vhd.
### Code Generation for 'hdlcoderParkTransformShareCopy' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoderParkTransformShareCopy' complete with 0 errors, 0 warnings, and 2 mess
### HDL code generation complete.
```

The High-level Resource Report shows that the resource utilization for the design was estimated to be:

Resource	Usage
Multipliers	13
Adders/Subtractors	178
Registers	1012
Total 1-Bit Registers	16888
RAMs	0
Multiplexers	610
I/O Bits	196
Static Shift operators	8
Dynamic Shift operators	8

You can also run synthesis workflow on the model. The synthesis results for this model are:

```
        Resource             Usage
_________________________   ________
Slice LUTs                  8792
Slice Registers             8419
DSPs                        13
Block Ram Tile              0
URAM                        0
```

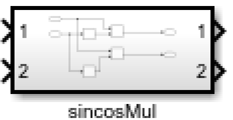
For more details regarding synthesis workflow, see “HDL Code Generation and FPGA Synthesis from Simulink Model”.

Use Clone Detection to Optimize the Model for Resource Sharing

Before you use the Clone Detector App, create a customer library that contains each repeating pattern in the model as an Atomic Subsystem. For this example, you are given a custom library called hdlcoderParkTransformShareLib.slx.

```
% Copy the library into the temporary directory
lib_name = 'hdlcoderParkTransformShareLib';
lib_new_name = 'hdlcoderParkTransformShareLibCopy';

copyfile(([lib_name,'.slx']), fullfile([lib_new_name,'.slx']), 'f');

% Open the custom library
open_system('hdlcoderParkTransformShareLibCopy');
```

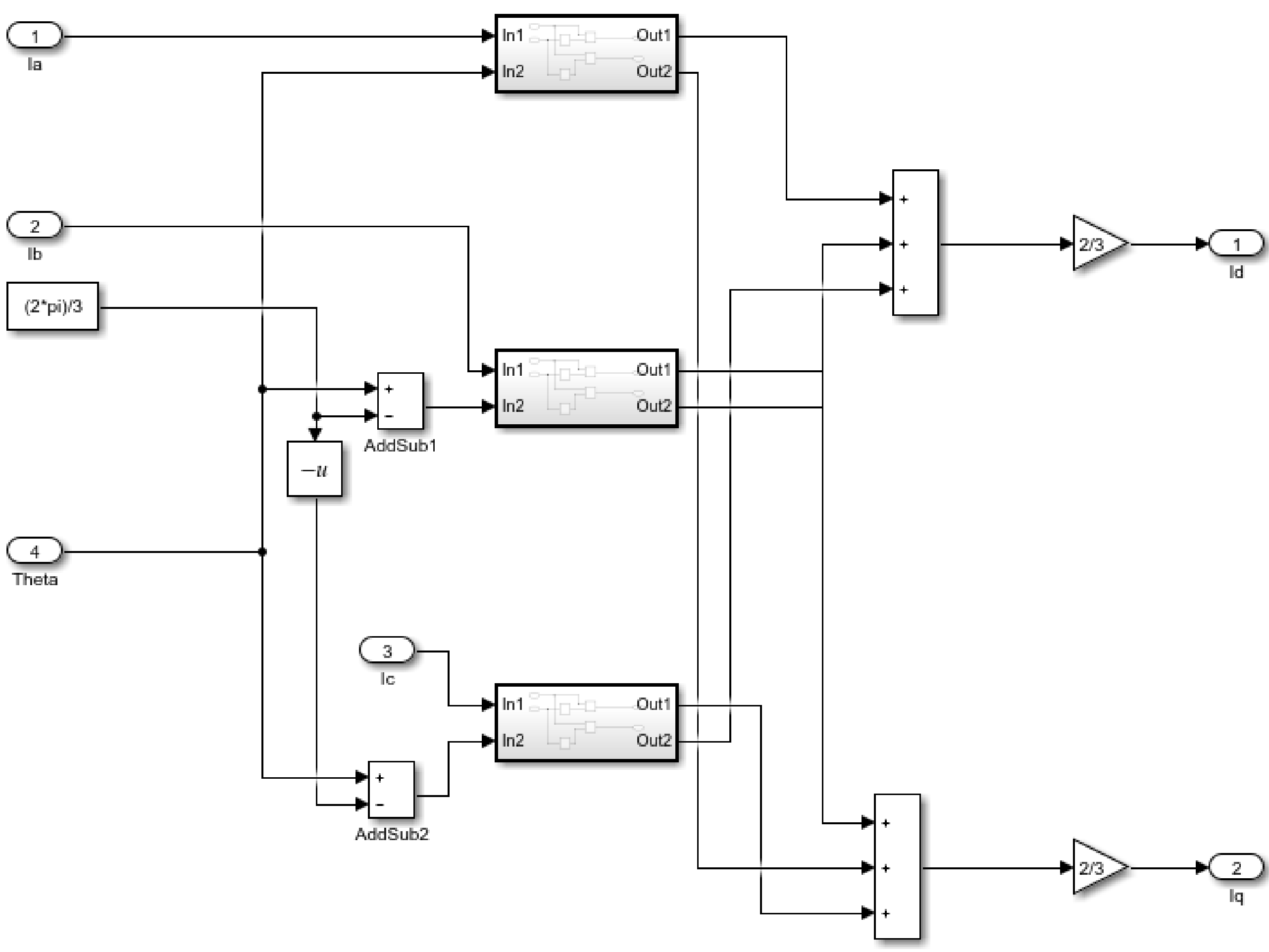
Copyright 2020-2021 The MathWorks, Inc.



Replace patterns of the model with the library containing the Atomic Subsystem. To configure the Clone Detector App and replace the clones, refer to “Enable Component Reuse by Using Clone Detection” (Simulink Check). Specifically, follow the instructions in the *Set the Parameters for Clone Detection* section to link the library for clone detection, and set the **Maximum number of different parameters** value to `0`.

Then, follow the *Replace Clones* section to apply this library to the model for exact clone replacement.

Notice that all the patterns in the original model has been replaced with atomic subsystems from the library as shown below:



Save the changes made to this system. In the Simulink Editor, on the **Simulation** tab, click **Save**.

Compare Results from Optimized Model

Since this model contains the Sin, Cos, and Product blocks inside atomic subsystems, they can now be shared and results in an improvement in resource utilization.

```
subsystem = [design_new_name '/DUT'];
hdlset_param(subsystem, 'SharingFactor', 3);
% Generate HDL code and the corresponding reports for the optimized model
makehdl(subsystem);
```

```
### Working on the model <a href="matlab:open_system('hdlcoderParkTransformShareCopy')">hdlcoderP
### Generating HDL for <a href="matlab:open_system('hdlcoderParkTransformShareCopy/DUT')">hdlcode
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderParkTran
### Running HDL checks on the model 'hdlcoderParkTransformShareCopy'.
### Begin compilation of the model 'hdlcoderParkTransformShareCopy'...
### Working on the model 'hdlcoderParkTransformShareCopy'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 32 cycles.
### Output port 2: 32 cycles.
```

```
### Working on... <a href="matlab:configset.internal.open('hdlcoderParkTransformShareCopy', 'Gene
### Begin model generation 'gm_hdlcoderParkTransformShareCopy'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoderParkTransformSha
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdl_
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsrc
### Generating new validation model: '<a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoderParkTra
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderParkTransformShareCopy'.
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 6.
### Begin VHDL Code Generation for 'DUT_tc'.
### Working on DUT_tc as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_tc.vhd.
### Code Generation for 'DUT_tc' completed.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sincos_single as hdl_prj\hdlsrc\hdlcoderPar
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkTr
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_uminus_single as hdl_prj\hdlsrc\hdlcoderPar
### Working on hdlcoderParkTransformShareCopy/DUT as hdl_prj\hdlsrc\hdlcoderParkTransformShareCop
### Generating package file hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_pkg.vhd.
### Code Generation for 'hdlcoderParkTransformShareCopy' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoderParkTransformShareCopy' complete with 0 errors, 0 warnings, and 2 mess
### HDL code generation complete.
```

After sharing, the High-level Resource Report shows that the resource utilization for the design was estimated to be:

Resource	Usage
Multipliers	13
Adders/Subtractors	177
Registers	1004
Total 1-Bit Registers	16657
RAMs	0
Multiplexers	610
I/O Bits	196
Static Shift operators	8
Dynamic Shift operators	8

Notice that the **Adders/Subtractors**, **Registers**, and **Total 1-Bit Registers** counts have reduced after the atomic subsystems are shared.

You can also run synthesis workflow on the model. The synthesis results for this model are:

Resource	Usage
Slice LUTs	8848
Slice Registers	8308
DSPs	13
Block Ram Tile	0
URAM	0

Notice that the **Slice Registers** count has reduced after sharing.

Delay Balancing

In this section...
“Specify Delay Balancing” on page 21-81
“Delay Balancing Considerations” on page 21-83
“Delay Absorption During Delay Balancing” on page 21-83
“Delay Balancing Report” on page 21-84
“Delay Balancing Limitations” on page 21-84

To achieve more efficient hardware usage and higher clock rates in your model, HDL Coder has several optimizations, block implementations, and options that introduce discrete delays into your model. Optimizations such as output pipelining, streaming, or resource sharing can introduce delays. Some block implementations, such as the Newton-Raphson and CORDIC architecture, inherently introduce delays in the generated code.

When optimizations or block implementation options introduce delays along the critical path in a model, the numerical calculations of the original model and generated model or HDL code can differ from each other because equivalent delays are not introduced on other, parallel signal paths. Manual insertion of compensating delays along other paths is possible, but is prone to error and does not scale well to large models that have many signal paths or multiple sample rates.

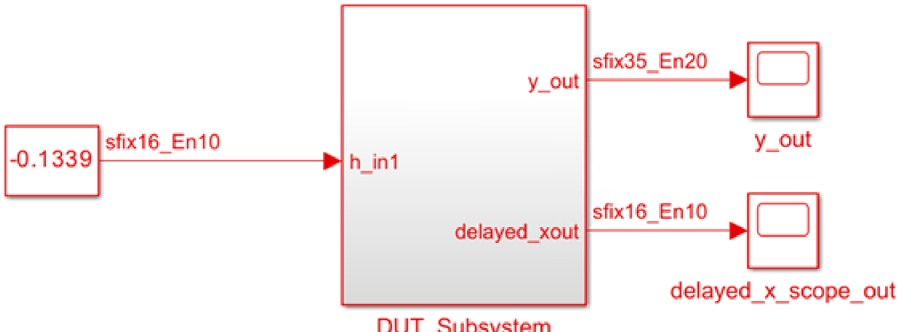
To avoid this issue, HDL Coder provides delay balancing. HDL Coder detects introduction of new delays along one path, and then inserts matching delays on the other paths.

Specify Delay Balancing

You can set delay balancing for the ports in your device under test (DUT) subsystem. You can disable delay balancing for specific parts of your design by using the HDL block property **BalanceDelays** for input and output ports. See “Delay Balancing Considerations” on page 21-83.

Disable Delay Balancing for Constant Sources

A stable path is a path where the initial source to the path is constant. If your model contains a stable path outside the DUT and connects to the top-level DUT subsystem and you want to reduce the amount of resources needed, you can disable delay balancing for the stable path by disabling the HDL block property **BalanceDelays** on the DUT-level Inport block. The **BalanceDelays** property affects only DUT-level Inport blocks. For example, in this model, you can disable **BalanceDelays** for the `DUT_Subsystem`'s Inport block, which is also input port `h_in1`, because the input to the Inport block is a Constant block.



To disable the HDL block property **BalanceDelays** for an Inport block using the HDL Block Properties dialog box:

1. Right-click the Inport block.
2. Select **HDL Code** > **HDL Block Properties**.
3. Set **BalanceDelays** to `off`.

To disable delay balancing for an Inport block from the command line, use the `hdlset_param` function.

If your DUT includes a subsystem hierarchy and you disable **BalanceDelays** on the DUT-level Inport block, the delay balancing on the stable path does not occur within the lower-level hierarchy if the stable path extends to that lower-level hierarchy. For an example, see “Control the Scope of Delay Balancing” on page 21-102.

Note When you disable **BalanceDelays** on DUT-level Inport blocks for stable paths and distributed pipelining is enabled for the model, change the **Pipeline distribution priority** configuration parameter from `Numerical Integrity` to `Performance` if you see delays inserted on stable paths in the generated model. For more information, see Pipeline distribution priority.

To disable the HDL block property **BalanceDelays** for an Outport block using the HDL Block Properties dialog box:

1. Right-click the Outport block.
2. Select **HDL Code** > **HDL Block Properties**.
3. Set **BalanceDelays** to `off`.

To disable delay balancing for an Outport block from the command line, use the `hdlset_param` function.

If your model contains a stable path that originates in the DUT, meaning that there is a Constant block as an input in the DUT, and a parallel path to the stable path has latency introduced from optimizations, delay balancing does not insert a matching delay on the stable path that contains the Constant block because the output value of the block is a constant. Not inserting the delay results in less resources needed for the design, but also results in an initial simulation mismatch. To resolve the simulation mismatch, add matching delays to the Constant block path by setting the HDL block

property **OutputPipeline** to the number of matching delays needed at the output of the Constant block.

Delay Balancing Considerations

When you generate HDL code, delay balancing is an essential part of code optimizations. If any optimizations are enabled, you must balance pipelines introduced as a part of those optimizations. Failure to balance automatically inserted pipeline delays causes issues in generated code deployed in hardware.

For multirate models, HDL Coder might generate a large number of pipeline registers that can prevent the HDL design from fitting onto an FPGA. Before you disable delay balancing for the model, see “Delay Balancing on Multirate Designs” on page 21-107 and “Optimize Generated HDL Code for Multirate Designs with Large Rate Differentials” on page 14-80.

Resolving Delay Balancing Errors in a Feedback Loop Design

When you enable delay balancing on a model and that model contains a feedback loop in the DUT subsystem, during HDL code generation, you may see this error:

An extra 10 cycles of latency introduced by optimizations in the feedback loop cannot be offset using design delays for the loop latency budget. If you are modeling at data rate (high-level description of algorithm without hardware implementation details), refer to the clock-rate pipelining report for more details. If you are modeling at clock rate, consider increasing the latency budget by adding more design delays in the feedback loop. Refer to the delay balancing report for more details. Offending block: <modelname>/Subsystem/Add1 . The error message includes links to the clock-rate pipelining report and delay balancing report.

In this situation, the error is a result of pipelining in feedback loops that introduces latency and breaks the functional equivalence between the original and generated model. However, disabling delay balancing for the entire model results in a simulation mismatch between your original model and the generated model. To resolve this issue, either:

- Use clock-rate pipelining. See the Use Clock-Rate Pipelining section in “Resolve Simulation Mismatch When Pipelining with a Feedback Loop Outside the DUT” on page 21-31.
- Increase the latency budget by adding more design delays. See the Model with Latency in a Feedback Loop section in “Use Delay Absorption While Modeling with Latency” on page 21-86.

Delay Absorption During Delay Balancing

The delay balancing optimization applies delay absorption to your model, which uses design delays in place of pipeline delays introduced from optimizations to prevent unused latency from being added in your design. Similar to other pipeline optimizations, such as distributed pipelining, delay absorption moves existing delays to the point where an optimization introduces latency. Delay absorption allows you to model with latency, which means that you can add design delays in the original design to simulate the extra delays introduced by optimizations in the generated design. If there are enough design delays, the extra delays introduced by optimizations are absorbed and no additional cycles of latency are reported at the DUT output ports. Modeling with latency prevents timing differences between the original model and generated model and allows HDL Coder to generate a functionally equivalent model to the generated HDL code. Modeling with latency at the clock rate is useful for

high-frequency, high-throughput applications, such as signal processing and wireless system design. For more information, see "Use Delay Absorption While Modeling with Latency" on page 21-86.

For delay absorption to run during HDL code generation:

- Place a Delay block with a delay length equal to or greater than the block latency after the block that introduces latency. You can place the Delay block anywhere downstream on the data path from the block that introduces latency. For example, if you have a series of floating-point operators that accumulate to create a large amount of latency, you can place a single Delay block with a delay length equal to the total latency from the operators at the end of the series of operators.
- For delay absorption to take place inside of a triggered subsystem, enable the model parameter **Use trigger signal as clock**.

For more information on floating-point operations and delay absorption, see "Latency Considerations with Native Floating Point" on page 14-104.

Delay Balancing Report

When you generate code for a design under test (DUT) HDL Coder produces the optimization report. Select the **Delay Balancing** section of the report. The DUT can be a subsystem, model, or model reference.

The Delay Balancing Report shows latency changes, pipeline and phase delay at the output ports, delay absorption information, and the number of pipelines added at the output ports to match the delays. If delay balancing fails, the report displays the criteria that was violated and the link to any block or subsystem that caused delay balancing to fail.

Delay Balancing Limitations

Subsystem-Level Restrictions

HDL Coder does not support delay balancing if:

- A subsystem with `BlackBox` architecture has the **ImplementationLatency** block property set to a negative value.

  To fix this error, for **ImplementationLatency**, enter a nonnegative integer.

Sample Time Restrictions

HDL Coder does not support delay balancing if you introduce latency from optimizations and:

- The sample time is continuous.
- Your design has an infinite sample time output.

See Also

Related Examples

- "Delay Balancing and Validation Model Workflow in HDL Coder" on page 21-94
- "Use Delay Absorption While Modeling with Latency" on page 21-86

- "Control the Scope of Delay Balancing" on page 21-102
- "Delay Balancing on Multirate Designs" on page 21-107

More About

- "Resolve Numeric Mismatch with Delay Balancing" on page 21-25
- "Create and Use Code Generation Reports" on page 23-2
- "Generated Model and Validation Model" on page 21-10
- "Check for blocks that have nonzero output latency" on page 37-21

Use Delay Absorption While Modeling with Latency

Learn how to model your design with latency to apply the delay absorption optimization that runs when you generate HDL code with delay balancing.

Delay absorption is part of the delay balancing optimization. Delay absorption uses design delays in place of pipeline delays introduced from optimizations to prevent unused latency from being added to your design. You can use delay absorption by modeling with latency, which means that you add design delays to your model to take the place of introduced latency from optimizations. By modeling your design with latency, HDL Coder™ can use delay absorption to absorb the design delays and not introduce extra pipeline delays that add latency to your model, which prevents timing differences between the original model and generated model and allows HDL Coder to generate a functionally equivalent model to the generated HDL code.

Delays in your design can either be design delays or pipeline delays. Design delays are delays that you manually insert in your design by using Delay blocks, or other blocks that have state, including Queue, HDL FIFO, or Buffer blocks. Pipeline delays are delays that are generated by optimization settings, such as input or output pipelining options or block implementation settings, such as the `ShiftAdd` implementation for a Divide block and the `CORDIC` approximation for a Trigonometric Function block.

Certain block implementations, floating-point operations, and optimization settings, such as input pipelining or resource sharing, introduce latency to the generated HDL code and generated model. The additional latency results in a timing difference between the original model and the generated model. To avoid this timing difference, such as when you are using a control system with a feedback loop, you can add design delays to your model. For delay absorption to absorb the design delays:

- Place a Delay block after the block that is introducing latency. You can place the Delay block anywhere downstream on the data path from the block that introduces latency.
- Set the Delay block **Delay length** parameter to a value equal to the block latency.

By adding Delay blocks to your original model, you can simulate your original model with latency.

These blocks can introduce latency:

- Divide, Sqrt, and Reciprocal blocks that have custom latency value greater than zero
- Trigonometric Function blocks that have **Function** set to `sin`, `cos`, `sincos`, `cos+jsin`, or `atan2` and **Approximation method** set to `CORDIC`
- Native floating-point operators that have the HDL block property **LatencyStrategy** set to `Max`, `Min`, or a custom value greater than zero

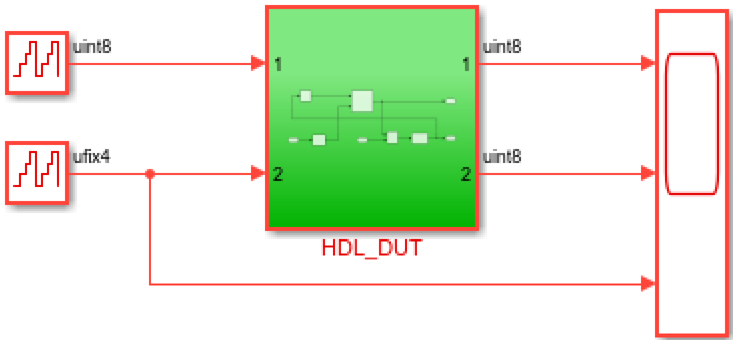
To learn about blocks that have custom latency with fixed-point types, open the `HDLMathLib` library. The library contains fixed-point blocks that have control signals.

```
open_system('HDLMathLib')
```

Model with Latency When Blocks Introduce Latency

To learn how HDL Coder absorbs delays, open the model `hdlcoder_absorb_delays`.

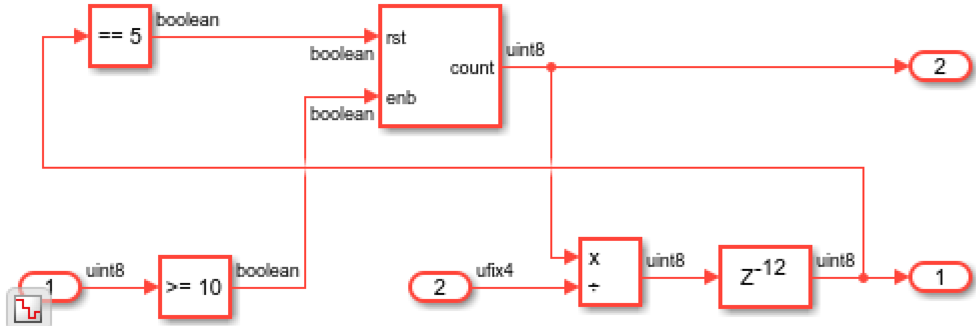
```
open_system('hdlcoder_absorb_delays')
set_param('hdlcoder_absorb_delays', 'SimulationCommand', 'Update')
```



Copyright 2020 The MathWorks, Inc.

Inside the `HDL_DUT` subsystem, you see a Delay block that has **Delay length** equal to `12` beside the Divide block. This **Delay length** corresponds to the latency of the division operation for fixed-point data types. In this case, the required **Delay length** is the sum of the bitwidth, `8`, and `4`, which is equal to `12`.

```
load_system('hdlcoder_absorb_delays')
open_system('hdlcoder_absorb_delays/HDL_DUT')
```



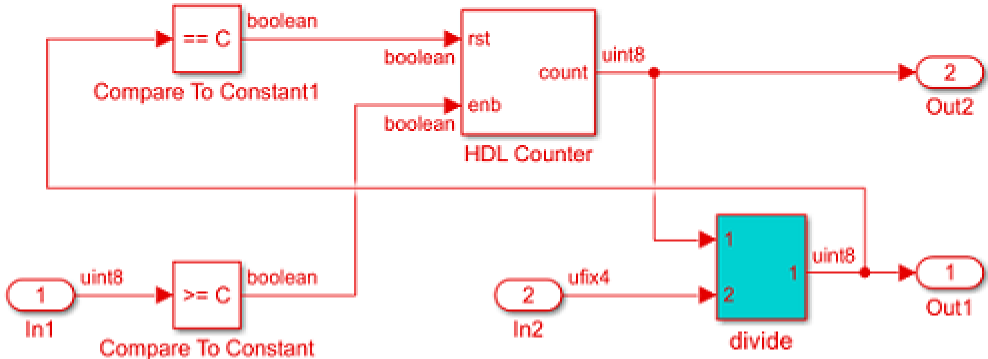
To generate HDL code for the DUT subsystem, use the `makehdl` function.

```
makehdl('hdlcoder_absorb_delays/HDL_DUT')
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_absorb_delays')">hdlcoder_absorb_d
### Generating HDL for <a href="matlab:open_system('hdlcoder_absorb_delays/HDL_DUT')">hdlcoder_ab
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_absorb_
### Running HDL checks on the model 'hdlcoder_absorb_delays'.
### Begin compilation of the model 'hdlcoder_absorb_delays'...
### Begin compilation of the model 'hdlcoder_absorb_delays'...
### Working on the model 'hdlcoder_absorb_delays'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_absorb_delays', 'GenerateMode
### Begin model generation 'gm_hdlcoder_absorb_delays'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_absorb_delays\gm_hdlco
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdls
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
```

```
### Begin VHDL Code Generation for 'hdlcoder_absorb_delays'.
### Working on hdlcoder_absorb_delays/HDL_DUT/Divide as hdlsrc\hdlcoder_absorb_delays\Divide.vhd
### Working on hdlcoder_absorb_delays/HDL_DUT as hdlsrc\hdlcoder_absorb_delays\HDL_DUT.vhd.
### Generating package file hdlsrc\hdlcoder_absorb_delays\HDL_DUT_pkg.vhd.
### Code Generation for 'hdlcoder_absorb_delays' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/t
### HDL check for 'hdlcoder_absorb_delays' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

In the generated model, the delays beside the Divide block are absorbed into the block latency. When you double-click this `Divide` block, you see the original Divide block and the Delay block with **Delay length** of 12.



For an example that shows delay absorption for floating-point operations, see “Latency Considerations with Native Floating Point” on page 14-104.

Model with Latency When Optimizations Introduce Latency

Optimizations such as streaming can introduce latency. For example, open the model `hdlcoder_absorb_delay_streaming`. The DUT contains a Delay block to model the original design with latency and a subsystem that contains a Product block. The inputs are vectors of size 10.

```
load_system('hdlcoder_absorb_delay_streaming');
open_system('hdlcoder_absorb_delay_streaming/DUT');
set_param('hdlcoder_absorb_delay_streaming','SimulationCommand','Update');
```

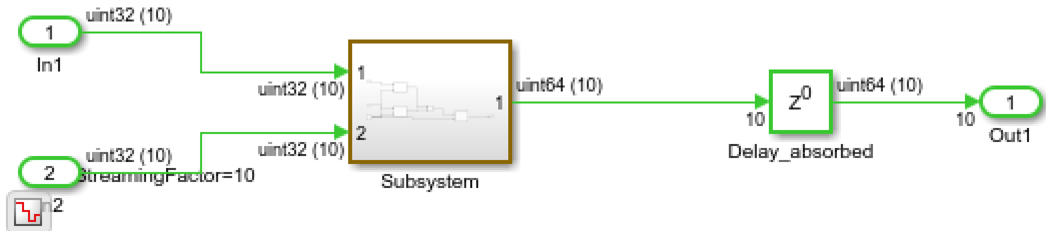
To stream the vector computations, set the HDL block property `StreamingFactor` to 10 on the lower-level subsystem. To model with latency, allow delay absorption to absorb Delay blocks in your model by using the model property `AllowDelayDistribution`, which is enabled by default. Then, generate HDL code for the DUT subsystem using the `makehdl` function.

```
hdlset_param('hdlcoder_absorb_delay_streaming/DUT/Subsystem', 'StreamingFactor', 10);
hdlget_param('hdlcoder_absorb_delay_streaming', 'AllowDelayDistribution');
makehdl('hdlcoder_absorb_delay_streaming/DUT');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_absorb_delay_streaming')">hdlcoder
### Generating HDL for <a href="matlab:open_system('hdlcoder_absorb_delay_streaming/DUT')">hdlcod
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_absorb_
### Running HDL checks on the model 'hdlcoder_absorb_delay_streaming'.
### Begin compilation of the model 'hdlcoder_absorb_delay_streaming'...
### Working on the model 'hdlcoder_absorb_delay_streaming'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_absorb_delay_streaming', 'Ger
### Begin model generation 'gm_hdlcoder_absorb_delay_streaming'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_absorb_delay_streaming\
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoder_absorb_delay_s
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_absorb_delay_streaming'.
### MESSAGE: The design requires 10 times faster clock with respect to the base rate = 0.1.
### Begin VHDL Code Generation for 'DUT_tc'.
### Working on DUT_tc as hdlsrc\hdlcoder_absorb_delay_streaming\DUT_tc.vhd.
### Code Generation for 'DUT_tc' completed.
### Working on hdlcoder_absorb_delay_streaming/DUT/Subsystem as hdlsrc\hdlcoder_absorb_delay_stre
### Working on hdlcoder_absorb_delay_streaming/DUT as hdlsrc\hdlcoder_absorb_delay_streaming\DUT
### Generating package file hdlsrc\hdlcoder_absorb_delay_streaming\DUT_pkg.vhd.
### Code Generation for 'hdlcoder_absorb_delay_streaming' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlcoder_absorb_delay_streaming' complete with 0 errors, 0 warnings, and 1 mes
### HDL code generation complete.
```

When streaming is enabled, a Deserializer block that has 1 cycle of delay is inserted in the generated design. If the original model did not include a Delay block, the generated design would have 1 cycle of extra latency compared to the original design. The delay absorption optimization uses the Delay block in the original model to offset the delay introduced by streaming so that the generated design does not have any added latency. Open the generated model to see the original delay block absorbed after code generation.

```
open_system('gm_hdlcoder_absorb_delay_streaming/DUT');
set_param('gm_hdlcoder_absorb_delay_streaming','SimulationCommand','Update');
```



To disable delay absorption from using Delay blocks in the original model, set the model property `AllowDelayDistribution` to `off`. Notice that because the delay absorption is disabled, there is now 1 cycle of extra latency in the generated model.

```
hdlset_param('hdlcoder_absorb_delay_streaming', 'AllowDelayDistribution', 'off');
makehdl('hdlcoder_absorb_delay_streaming/DUT');
open_system('gm_hdlcoder_absorb_delay_streaming/DUT');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_absorb_delay_streaming')">hdlcoder
### Generating HDL for <a href="matlab:open_system('hdlcoder_absorb_delay_streaming/DUT')">hdlcod
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_absorb_
### Running HDL checks on the model 'hdlcoder_absorb_delay_streaming'.
### Begin compilation of the model 'hdlcoder_absorb_delay_streaming'...
### Working on the model 'hdlcoder_absorb_delay_streaming'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_absorb_delay_streaming', 'Ger
### Begin model generation 'gm_hdlcoder_absorb_delay_streaming'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_absorb_delay_streaming\
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoder_absorb_delay_s
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_absorb_delay_streaming'.
### MESSAGE: The design requires 10 times faster clock with respect to the base rate = 0.1.
### Begin VHDL Code Generation for 'DUT_tc'.
### Working on DUT_tc as hdlsrc\hdlcoder_absorb_delay_streaming\DUT_tc.vhd.
### Code Generation for 'DUT_tc' completed.
### Working on hdlcoder_absorb_delay_streaming/DUT/Subsystem as hdlsrc\hdlcoder_absorb_delay_stre
### Working on hdlcoder_absorb_delay_streaming/DUT as hdlsrc\hdlcoder_absorb_delay_streaming\DUT
### Generating package file hdlsrc\hdlcoder_absorb_delay_streaming\DUT_pkg.vhd.
### Code Generation for 'hdlcoder_absorb_delay_streaming' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlcoder_absorb_delay_streaming' complete with 0 errors, 0 warnings, and 1 mes
### HDL code generation complete.
```

You can also disable delay absorption for a specific Delay block by disabling the HDL block property `AllowDelayDistribution` for the block. For more information, see “AllowDelayDistribution” on page 19-5.

Model with Latency in a Feedback Loop

When you model with latency in a feedback loop, you can either:

- Use clock-rate pipelining. For more information see “Clock-Rate Pipelining” on page 21-148.
- Use delay absorption. For example, you can model a feedback loop at the clock rate with some of amount of latency expressed as design delays to absorb the latency introduced from native floating-point operators or pipeline optimizations.

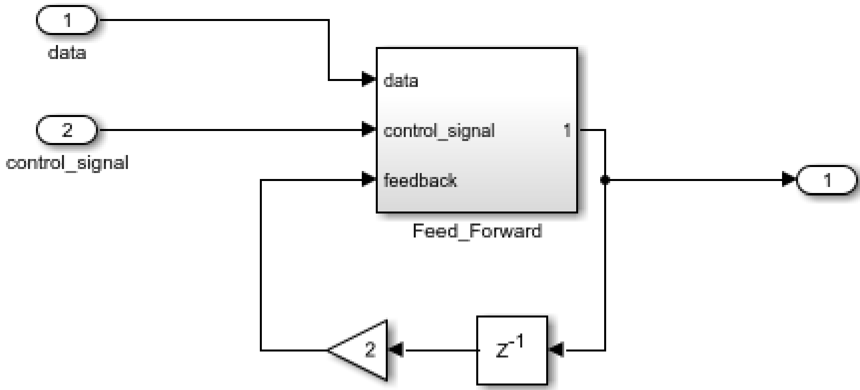
In this example, use delay absorption to model with latency in a feedback loop.

This example contains output pipelines to help optimize the speed of the design, but at the cost of introducing an extra cycle of latency for each pipeline. This additional latency is balanced during delay balancing so that the numerics and functionality of the algorithm are preserved. However, in a feedback loop, delay balancing requires you to model with latency in order to absorb the design delays to prevent a timing mismatch as a result of the pipeline optimizations.

For example, if you generate code with output pipelines added in a feedback loop without modeling with latency, HDL Coder generates a delay balancing error. You can observe this in the `delay_absorption_feedback_loop` model, where the subsystem `delay_absorption_feedback_loop/DUT/Subsystem` is in a feedback loop and the `delay_absorption_feedback_loop/DUT/Feed_Forward/Sum` block has the HDL block property `OutputPipeline` set to 2.

Load and open the model to its DUT subsystem.

```
load_system('delay_absorption_feedback_loop');
open_system('delay_absorption_feedback_loop/DUT');
```



View the non-default parameters by using `hdlsaveparams` to see the `OutputPipeline` setting for the Sum block.

```
hdlsaveparams('delay_absorption_feedback_loop');
```

```
%% Set Model 'delay_absorption_feedback_loop' HDL parameters
hdlset_param('delay_absorption_feedback_loop', 'GenerateValidationModel', 'on');
hdlset_param('delay_absorption_feedback_loop', 'HDLSubsystem', 'delay_absorption_feedback_loop/DU

% Set Sum HDL parameters
hdlset_param('delay_absorption_feedback_loop/DUT/Feed_Forward/Sum', 'OutputPipeline', 2);
```

If you generate code by using the `makehdl` function, there is a delay balancing error in the HDL Code Generation Report. To fix this error, you need to add enough latency in your design to match the amount of latency added from the output pipelining optimization.

In this example, you need two unit delays total, or a single design delay with a **Delay Length** of two. The error mentions only one extra cycle of latency because the current Delay block in the feedback path is absorbed to handle one of the two extra cycles of latency. Increase the feedback delay block from a **Delay Length** of one to a **Delay Length** of two. You can do this by setting the parameter in the Delay block or by using this command:

```
set_param('delay_absorption_feedback_loop/DUT/Delay', 'DelayLength', '2');
```

Generate HDL code by using the `makehdl` function.

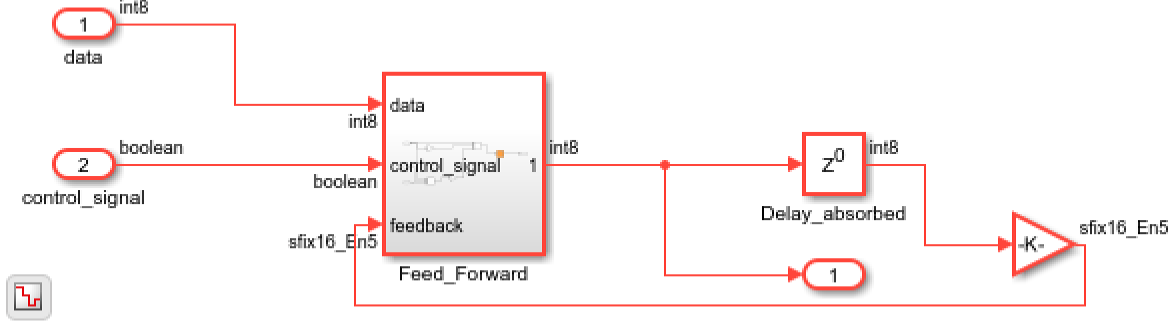
```
makehdl('delay_absorption_feedback_loop/DUT');
```

```
### Working on the model <a href="matlab:open_system('delay_absorption_feedback_loop')">delay_abs
### Generating HDL for <a href="matlab:open_system('delay_absorption_feedback_loop/DUT')">delay_a
### Using the config set for model <a href="matlab:configset.showParameterGroup('delay_absorptio
### Running HDL checks on the model 'delay_absorption_feedback_loop'.
### Begin compilation of the model 'delay_absorption_feedback_loop'...
### Working on the model 'delay_absorption_feedback_loop'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
```

```
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('delay_absorption_feedback_loop', 'Gene
### Begin model generation 'gm_delay_absorption_feedback_loop'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\delay_absorption_feedback_loop\g
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\delay_absorption_feedba
### Validation model generation complete.
### Begin VHDL Code Generation for 'delay_absorption_feedback_loop'.
### Working on delay_absorption_feedback_loop/DUT/Feed_Forward as hdlsrc\delay_absorption_feedbac
### Working on delay_absorption_feedback_loop/DUT as hdlsrc\delay_absorption_feedback_loop\DUT.vh
### Generating package file hdlsrc\delay_absorption_feedback_loop\DUT_pkg.vhd.
### Code Generation for 'delay_absorption_feedback_loop' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'delay_absorption_feedback_loop' complete with 0 errors, 0 warnings, and 0 mess
### HDL code generation complete.
```

Open the generated model to view the absorbed design delays. The feedback delays were absorbed and moved into the Feed_Forward subsystem to handle the latency added from output pipelining. No extra latency is introduced as a result of delay absorption.

```
open_system('gm_delay_absorption_feedback_loop/DUT');
set_param('gm_delay_absorption_feedback_loop', 'SimulationCommand', 'update');
```



When applying delay absorption in a feedback loop, follow these guidelines:

- Add the design delays to be absorbed at the same subsystem level as the feedback loop.
- Delay absorption cannot absorb upstream latency from a feedback loop, which means that if there are design delays placed before a feedback loop in your model, delay absorption does not absorb those upstream design delays. If you do have upstream latency that is not absorbed, HDL code generation produces a delay balancing error message in the HDL code generation report. Use clock-rate pipelining to model with latency when there is upstream latency in your design.

- Delay absorption cannot absorb design delays separated from the block that introduces latency by a component that takes in a zero-input value and outputs a non-zero value, such as a NOT Logical Operator block.

See Also

Related Examples

- “Latency Considerations with Native Floating Point” on page 14-104
- “Delay Balancing” on page 21-81
- “Absorb Delays to Avoid Timing Difference” on page 18-105

Delay Balancing and Validation Model Workflow in HDL Coder

This example shows how HDL Coder™ can automatically balance delays within a model. HDL Coder may introduce additional delays in the HDL implementation for a given model. These delays may be introduced by either certain block implementations or by optimizations for the purpose of improving the efficiency of the hardware implementations. However, introducing delays on only certain paths can result functional behavior that is different from the original intent of the user model, thereby violating functional equivalence between the original user model and the HDL implementation.

Delay Balancing is a feature supported by HDL Coder for automatically balancing such newly introduced delays across all parallel paths, ensuring that functional integrity is preserved with reference to the original model. This equivalence relationship can be confirmed by invoking the validation model workflow that enables the user to visualize the HDL Code-generation model, the delays introduced by implementations and those introduced by delay balancing and verify the equivalence relationship with the original model.

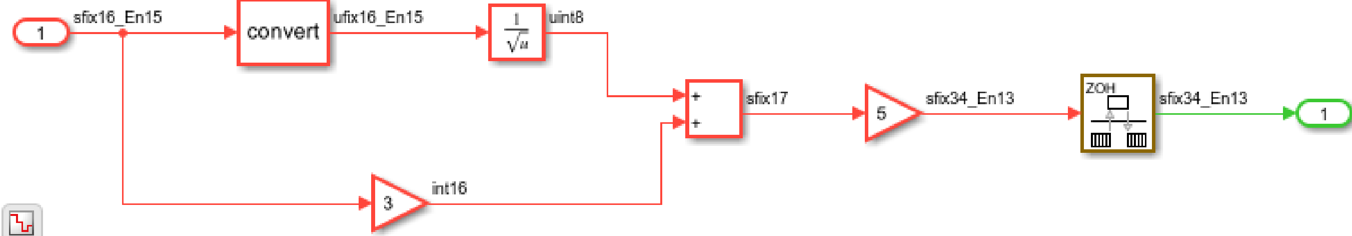
Implementations and Optimizations introduce Latency

Some of the arithmetic blocks in Simulink® require complex hardware algorithms. Consider, for example, the reciprocal square root block. This block computes its answer in a single time step in Simulink. If the corresponding hardware implementation should stay cycle-accurate with Simulink, the hardware algorithm for this block must compute in a single clock cycle. However, this results in a long critical path that degrades the clock frequency and efficiency of hardware. Thus, HDL Coder implements this block with a 5-cycle latency, which means that every path containing this block will introduce a 5-cycle delay.

Certain optimizations supported by HDL Coder may also introduce additional delays. For example, specifying `InputPipeline` or `OutputPipeline` as an implementation parameter on a block introduces additional pipeline delays in the generated HDL. This is again unmatched across parallel paths and will result in functional differences with the original model.

Consider an example model that contains a square root block implementing the `rSqrt` function and the `Gain3` block along the parallel path has the `OutputPipeline` HDL Block property set to 2.

```
load_system('hdl_delaybalancing');
open_system('hdl_delaybalancing/Subsystem');
set_param('hdl_delaybalancing', 'SimulationCommand', 'update');
```



Validation Model Generation

Due to changes in latency, the HDL Coder always generates a Code Generation model that captures the added delays during implementation. The RTL verification and automatic co-simulation model generation features validate that the RTL simulation of the generated HDL code is bit-accurate and

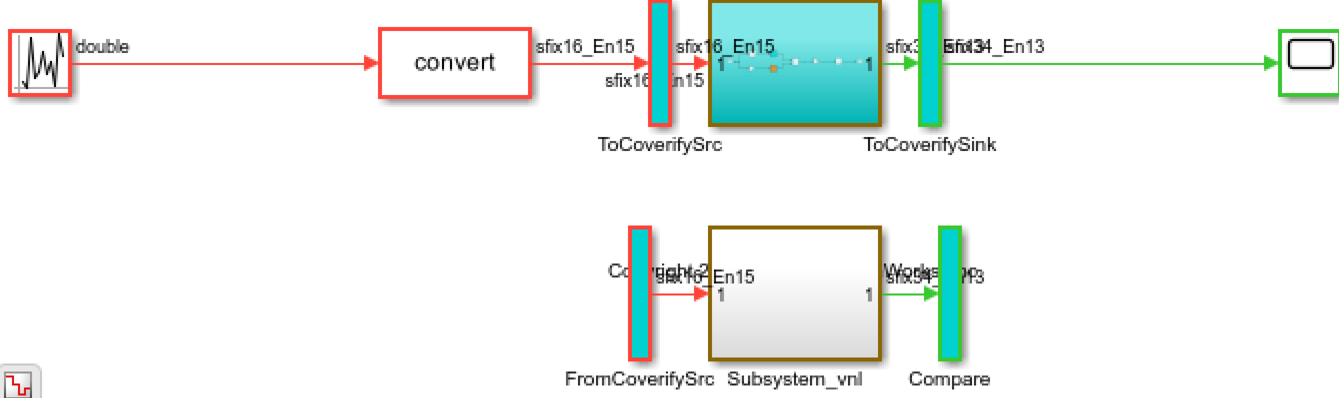
cycle-accurate with the Simulink simulation of the Code-generation model. However, this does not say anything about the functional relationship with the original, user model.

The Validation model enables the user to verify that the functional equivalence of the original, user model with the Code-generation model. This feature is turned on by the model-level parameter, `GenerateValidationModel`. This parameter can be set by either the `hdlset_param` command or can be supplied as a `makehdl` argument. Then, during code generation, notice a message that says that validation model has been generated.

The Validation Model consists of two parts: the DUT from the original model (`gm_hdl_delaybalancing_vnl/Subsystem_vnl`) and the DUT from the Code-generation model (`gm_hdl_delaybalancing_vnl/Subsystem`).

```
hdlset_param('hdl_delaybalancing', 'GenerateValidationModel', 'on');
hdlset_param('hdl_delaybalancing', 'BalanceDelays', 'off');
makehdl('hdl_delaybalancing/Subsystem');
open_system('gm_hdl_delaybalancing_vnl');
set_param('gm_hdl_delaybalancing_vnl', 'SimulationCommand', 'update');
```

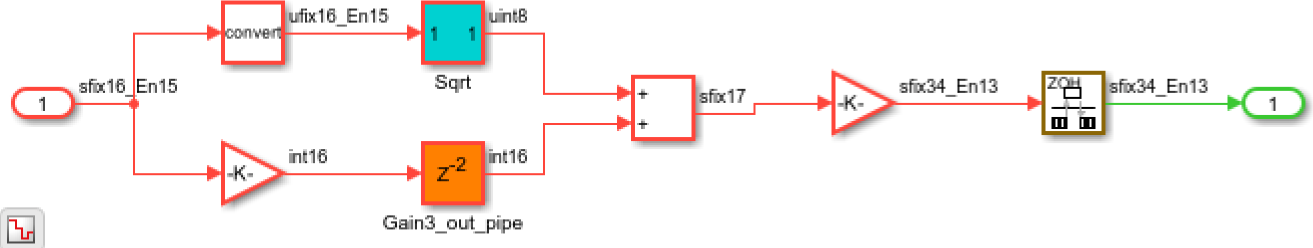
```
### Working on the model <a href="matlab:open_system('hdl_delaybalancing')">hdl_delaybalancing</a
### Generating HDL for <a href="matlab:open_system('hdl_delaybalancing/Subsystem')">hdl_delaybala
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_delaybalanci
### Running HDL checks on the model 'hdl_delaybalancing'.
### Begin compilation of the model 'hdl_delaybalancing'...
### Working on the model 'hdl_delaybalancing'...
### Working on... <a href="matlab:configset.internal.open('hdl_delaybalancing', 'GenerateModel')"
### Begin model generation 'gm_hdl_delaybalancing'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdl_delaybalancing\gm_hdl_delayb
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdl_delaybalancing\gm_h
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdl_delaybalancing'.
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 0.1.
### Begin VHDL Code Generation for 'Subsystem_tc'.
### Working on Subsystem_tc as hdlsrc\hdl_delaybalancing\Subsystem_tc.vhd.
### Code Generation for 'Subsystem_tc' completed.
### Working on hdl_delaybalancing/Subsystem/Sqrt/Sqrt_iv as hdlsrc\hdl_delaybalancing\Sqrt_iv.vhd
### Working on hdl_delaybalancing/Subsystem/Sqrt/Sqrt_core as hdlsrc\hdl_delaybalancing\Sqrt_core
### Working on hdl_delaybalancing/Subsystem/Sqrt as hdlsrc\hdl_delaybalancing\Sqrt.vhd.
### Working on hdl_delaybalancing/Subsystem as hdlsrc\hdl_delaybalancing\Subsystem.vhd.
### Generating package file hdlsrc\hdl_delaybalancing\Subsystem_pkg.vhd.
### Code Generation for 'hdl_delaybalancing' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdl_delaybalancing' complete with 0 errors, 0 warnings, and 5 messages.
### HDL code generation complete.
```



Code Generation DUT: Output Pipeline Insertion

The top subsystem (`gm_hdl_delaybalancing_vnl/Subsystem`) in the Validation model is the DUT as implemented for HDL code generation and this is reference DUT when performing RTL testbench verification and Cosimulation block based verification. Notice that `OutputPipeline` parameter on the `Gain3` block is implemented by an integer delay of length 2.

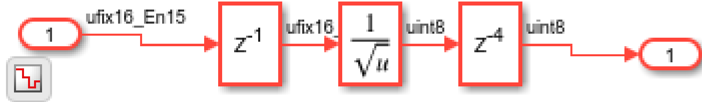
```
open_system('gm_hdl_delaybalancing_vnl/Subsystem');
```



Code Generation DUT: Sqrt Block Implementation

Implementing the square root function in on clock cycle is not efficient for hardware. The coder implements a pipelined architecture and this is reflected in the Code-Generation DUT model (under the square root subsystem) by the five additional delays.

```
open_system('gm_hdl_delaybalancing_vnl/Subsystem/Sqrt');
```



Equivalence Checking with the Validation Model

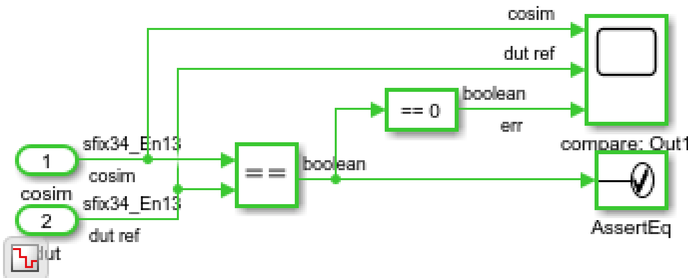
Validation model performs equivalence checking by routing the same inputs to both (the original and code-generation) DUTs using From and Goto blocks. This is encapsulated in the `ToCoverifySrc` and `FromCoverifySrc` subsystems. Both DUTs now respond to the same stimuli in each time step. The outputs from both DUTs are then sampled in each time step and their equivalence is checked. This is done by comparing the outputs from each output port, computing their difference, which should always be zero for functional equivalence.

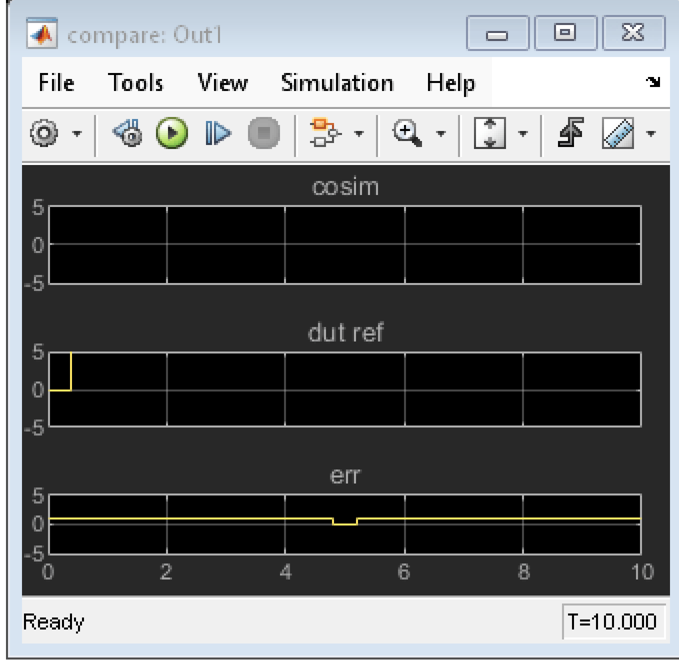
In the current example, however, notice that functional equivalence is violated. The difference between the two outputs is non-zero in several time steps. This results in mismatch assertions and is also reflected in the last panel of the comparison scope.

```
open_system('gm_hdl_delaybalancing_vnl/Compare/Assert_Out1');
sim('gm_hdl_delaybalancing_vnl');
open_system('gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/compare: Out1')
```

```
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 0
Warning: Division by zero occurred. Quotient was saturated. This originated from
'gm_hdl_delaybalancing_vnl/Subsystem/Sqrt/Sqrt'
Suggested Actions:
    •  - Suppress

Warning: Saturate on overflow detected. This originated from
'gm_hdl_delaybalancing_vnl/Subsystem/Sqrt/Sqrt'
Suggested Actions:
    •  - Suppress

Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 0.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 0.8
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 1.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 1.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 2.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 2.8
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 3.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 3.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 4.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 5.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 5.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 6.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 6.8
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 7.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 7.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 8
```

```
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 8.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 8.8
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 9.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 9.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 10
```





Automatic Delay Balancing

To solve the functional equivalence problem, you can turn on the delay balancing feature by setting the model-level `BalanceDelays` option to `on`. This can be done through either the `hdlset_param` command or as a `makehdl` argument.

With `BalanceDelays` turned on, HDL Coder automatically identifies the locations where matching delays need to be added to guarantee functional equivalence. After taking into account all of the delays introduced from implementation and optimizations, `BalanceDelays` adds delays that cover regular cut-sets, multi-rate boundaries, and subsystem boundaries.

Now when we observe the Code-generation DUT from the validation model notice that several additional delays have been added for matching delays introduced by the Sqrt block and OutputPipeline option. The names of these delays are typically prefixed with `delayMatch`. Notice that the coder also automatically computes the appropriate delays needed when crossing rate boundaries.

```
hdlset_param('hdl_delaybalancing', 'BalanceDelays', 'on');
makehdl('hdl_delaybalancing/Subsystem');
open_system('gm_hdl_delaybalancing_vnl/Subsystem')
set_param('gm_hdl_delaybalancing_vnl', 'SimulationCommand', 'update');
```

```
### Working on the model <a href="matlab:open_system('hdl_delaybalancing')">hdl_delaybalancing</a
### Generating HDL for <a href="matlab:open_system('hdl_delaybalancing/Subsystem')">hdl_delaybala
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_delaybalanci
### Running HDL checks on the model 'hdl_delaybalancing'.
### Begin compilation of the model 'hdl_delaybalancing'...
### Working on the model 'hdl_delaybalancing'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('hdl_delaybalancing', 'GenerateModel')"
### Begin model generation 'gm_hdl_delaybalancing'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdl_delaybalancing\gm_hdl_delayb
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdl_delaybalancing\gm_h
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdl_delaybalancing'.
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 0.1.
### Begin VHDL Code Generation for 'Subsystem_tc'.
### Working on Subsystem_tc as hdlsrc\hdl_delaybalancing\Subsystem_tc.vhd.
### Code Generation for 'Subsystem_tc' completed.
### Working on hdl_delaybalancing/Subsystem/Sqrt/Sqrt_iv as hdlsrc\hdl_delaybalancing\Sqrt_iv.vhd
### Working on hdl_delaybalancing/Subsystem/Sqrt/Sqrt_core as hdlsrc\hdl_delaybalancing\Sqrt_core
### Working on hdl_delaybalancing/Subsystem/Sqrt as hdlsrc\hdl_delaybalancing\Sqrt.vhd.
### Working on hdl_delaybalancing/Subsystem as hdlsrc\hdl_delaybalancing\Subsystem.vhd.
### Generating package file hdlsrc\hdl_delaybalancing\Subsystem_pkg.vhd.
### Code Generation for 'hdl_delaybalancing' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/t
### HDL check for 'hdl_delaybalancing' complete with 0 errors, 0 warnings, and 4 messages.
### HDL code generation complete.
```


Initial Latency and Functional Validation

The delays introduced by implementations essentially construct a pipelined hardware architecture to improve clock frequency and hardware efficiency. The pipeline however introduces an initial latency and the first output sample is generated after this initial latency. While these pipeline delays are

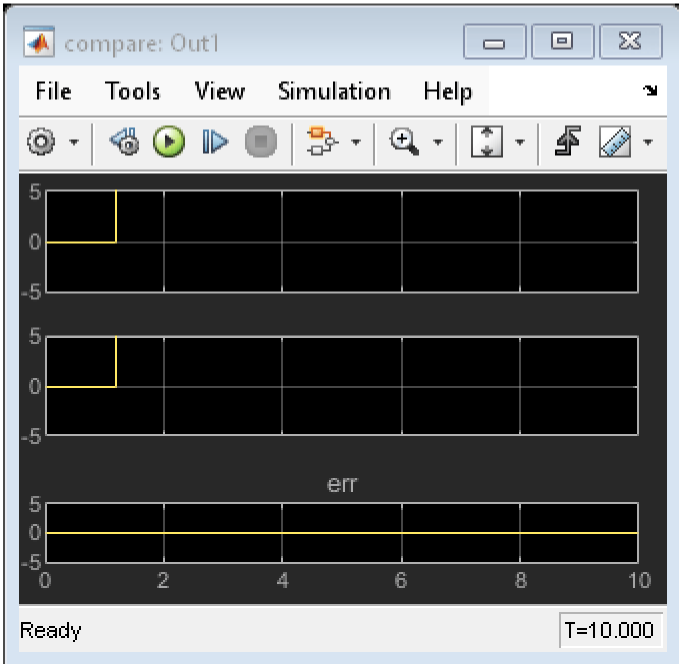
automatically balanced inside the DUT, it is important to manually balance delays outside the DUT in the rest of the model. The amount of delay (or initial latency) is communicated to the user during code generation as follows:

```
### Some latency changes occurred in the DUT. Each output port experiences these additional de
### Output port 0: 2 cycles
```

The equivalence checking in the Validation model uses this initial latency information for delaying the output from the original DUT. This is an example of balancing the delay outside DUT, since the balancing occurs at the inputs of the equivalence checking subsystem. Now, when we simulate the Validation model, note that there are no assertions and thus functional equivalence is preserved. While the pipeline delays are automatically balanced inside the DUT, it is important to manually balance delays outside the DUT in the rest of the model.

```
close_system('gm_hdl_delaybalancing_vnl/Subsystem')
sim('gm_hdl_delaybalancing_vnl');
open_system('gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/compare: Out1')
```

```
Warning: Division by zero occurred. Quotient was saturated. This originated from
'gm_hdl_delaybalancing_vnl/Subsystem/Sqrt/Sqrt'
Suggested Actions:
    •  - Suppress

Warning: Saturate on overflow detected. This originated from
'gm_hdl_delaybalancing_vnl/Subsystem/Sqrt/Sqrt'
Suggested Actions:
    •  - Suppress
```



Control the scope of delay balancing

The examples above describe the delay balancing feature as applied to the whole DUT. Sometimes, the design may explicitly model control and data paths, and you may not want to insert matching

delays on the control path during delay balancing. The examples in “Control the Scope of Delay Balancing” on page 21-102 show how this option can be applied locally to individual subsystems instead of the entire DUT.

Control the Scope of Delay Balancing

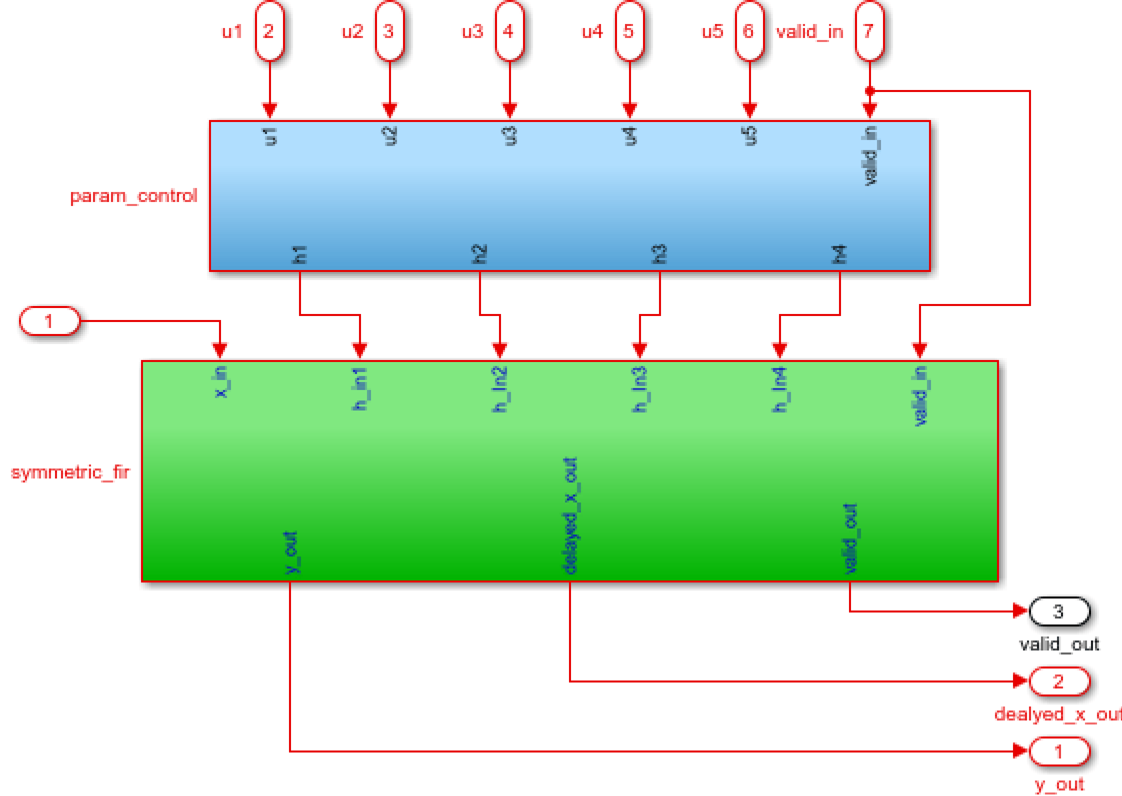
This example shows how to balance delays in specific parts of a design, without balancing delays on the entire design.

You can programmatically balance delays by using the `BalanceDelays` HDL input and output port block property to balance the additional delays introduced by HDL Coder™. You can enable or disable this property on individual input and output ports in your model. For example, in a design containing a data path and a control path, delay balancing can be applied exclusively to the data path of the design. This means focusing on the paths that necessitate data synchronization. You can use `BalanceDelays` at the design under test (DUT) port level to control how and where HDL Coder balances delays. `hdlcoder_localdelaybalancing.slx` shows how to disable delay balancing on user-defined stable control paths.

Constrain Delay Balancing to the Data Path

The example model `hdlcoder_localdelaybalancing.slx` has two subsystems under the device under test (DUT) subsystem, `hdlcoder_localdelaybalancing/Subsystem`, `param_control` and `symmetric_fir`, which contain the control logic and the data path, respectively.

```
open_system('hdlcoder_localdelaybalancing');
open_system('hdlcoder_localdelaybalancing/Subsystem');
```



To optimize timing results, each subsystem has one block that has one output pipeline register.

```
hdldispblkparams('hdlcoder_localdelaybalancing/Subsystem/param_control/And');
hdldispblkparams('hdlcoder_localdelaybalancing/Subsystem/symmetric_fir/Add');
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL Block Parameters ('hdlcoder_localdelaybalancing/Subsystem/param_control/And')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Implementation

    Architecture : default

Implementation Parameters

    OutputPipeline : 1


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL Block Parameters ('hdlcoder_localdelaybalancing/Subsystem/symmetric_fir/Add')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Implementation

    Architecture : Linear

Implementation Parameters

    OutputPipeline : 1
```

Enable the generation of the validation model and generate HDL code using the `makehdl` function.

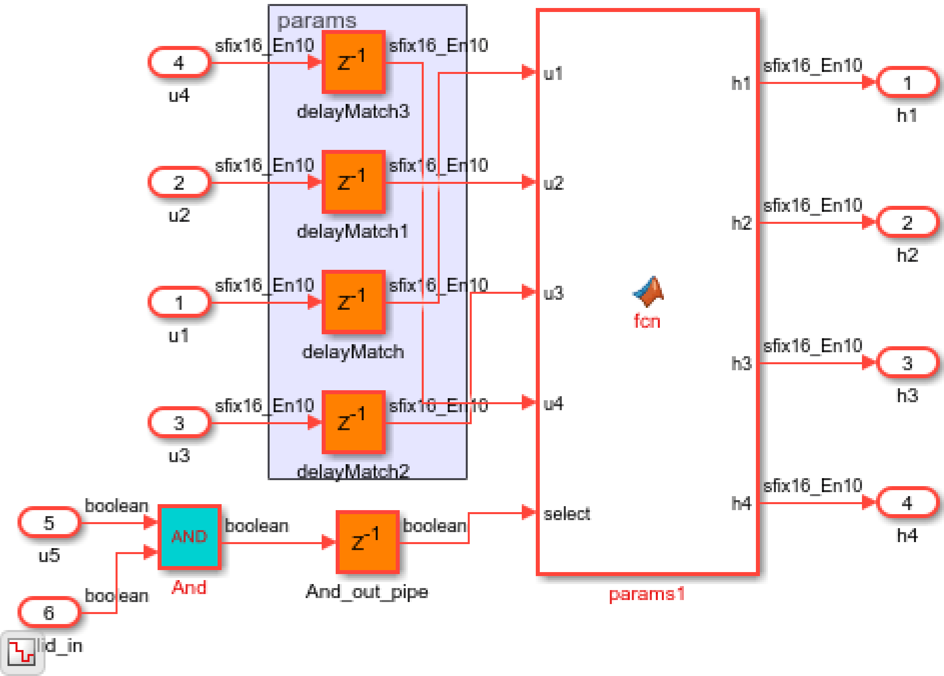
```
hdlset_param('hdlcoder_localdelaybalancing', 'GenerateValidationModel', 'on');
makehdl('hdlcoder_localdelaybalancing/Subsystem');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_localdelaybalancing')">hdlcoder_lo
### Generating HDL for <a href="matlab:open_system('hdlcoder_localdelaybalancing/Subsystem')">hdl
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_localde
### Running HDL checks on the model 'hdlcoder_localdelaybalancing'.
### Begin compilation of the model 'hdlcoder_localdelaybalancing'...
### Working on the model 'hdlcoder_localdelaybalancing'...
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 1 cycles.
### Output port 2: 1 cycles.
### Output port 3: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_localdelaybalancing', 'Genera
### Begin model generation 'gm_hdlcoder_localdelaybalancing'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_localdelaybalancing\gm_
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdls
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoder_localdelaybala
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_localdelaybalancing'.
### Working on hdlcoder_localdelaybalancing/Subsystem/param_control/params as hdlsrc\hdlcoder_loc
```

```
### Working on hdlcoder_localdelaybalancing/Subsystem/param_control as hdlsrc\hdlcoder_localdelay
### Working on hdlcoder_localdelaybalancing/Subsystem/symmetric_fir as hdlsrc\hdlcoder_localdelay
### Working on hdlcoder_localdelaybalancing/Subsystem as hdlsrc\hdlcoder_localdelaybalancing\Subs
### Code Generation for 'hdlcoder_localdelaybalancing' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_localdelaybalancing' complete with 0 errors, 0 warnings, and 1 messag
### HDL code generation complete.
```

Load and update the validation model, rearrange the generated model layout, and open the validation model to the `param_control` subsystem.

```
load_system('gm_hdlcoder_localdelaybalancing_vnl');
set_param('gm_hdlcoder_localdelaybalancing_vnl', 'SimulationCommand', 'update');
arrangeLayout('gm_hdlcoder_localdelaybalancing_vnl/Subsystem/param_control')
open_system('gm_hdlcoder_localdelaybalancing_vnl/Subsystem/param_control');
```



Delay balancing is enabled globally by default and inserts matching delays on both the control path and the data path, as shown in the validation model. For more information on the validation model, see “Generated Model and Validation Model” on page 21-10.

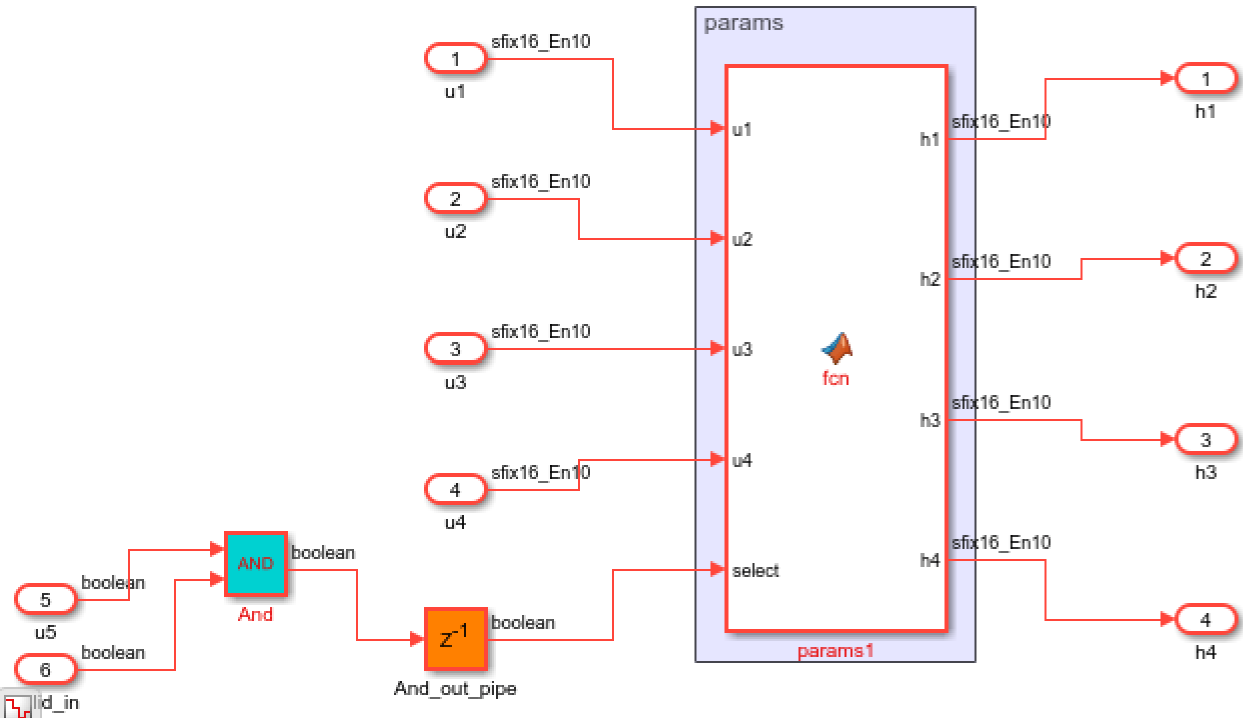
In this example, only the data path, `symmetric_fir`, requires data synchronization. The inputs `u1` through `u4` to the `param_control` subsystem are constant inputs that are coefficients to the FIR filter. The input paths through the DUT subsystem for these coefficients are stable and do not have to synchronize with each other or with the processed data. As a result, you can disable delay balancing for the input ports with constant inputs to the control logic to save on hardware resources. To achieve this, set the HDL block property `BalanceDelays` for Inport blocks `u1` through `u4` of the DUT subsystem to `off`. Because the stable paths propagate through to the lower-level hierarchy in the DUT subsystem, you do not need to disable this property for lower-level Inport blocks inside the `param_control` or `symmetric_fir` subsystems.

```
hdlset_param('hdlcoder_localdelaybalancing/Subsystem/u1', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_localdelaybalancing/Subsystem/u2', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_localdelaybalancing/Subsystem/u3', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_localdelaybalancing/Subsystem/u4', 'BalanceDelays', 'off');
```

Close the validation model, generate HDL code using the `makehdl` function, and open the new validation model to the `param_control` subsystem.

```
bdclose('gm_hdlcoder_localdelaybalancing_vnl');
hdlset_param('hdlcoder_localdelaybalancing', 'ValidationModelNameSuffix', '_vnl_DBoff')
makehdl('hdlcoder_localdelaybalancing/Subsystem');
load_system('gm_hdlcoder_localdelaybalancing_vnl_DBoff');
set_param('gm_hdlcoder_localdelaybalancing_vnl_DBoff', 'SimulationCommand', 'update');
Simulink.BlockDiagram.expandSubsystem('gm_hdlcoder_localdelaybalancing_vnl_DBoff/Subsystem/param_
open_system('gm_hdlcoder_localdelaybalancing_vnl_DBoff/Subsystem/param_control');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_localdelaybalancing')">hdlcoder_lo
### Generating HDL for <a href="matlab:open_system('hdlcoder_localdelaybalancing/Subsystem')">hdl
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_localde
### Running HDL checks on the model 'hdlcoder_localdelaybalancing'.
### Begin compilation of the model 'hdlcoder_localdelaybalancing'...
### Working on the model 'hdlcoder_localdelaybalancing'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 1 cycles.
### Output port 2: 1 cycles.
### Output port 3: 1 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoder_localdelaybalancing', 'Genera
### Begin model generation 'gm_hdlcoder_localdelaybalancing'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_localdelaybalancing\gm_
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdls
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoder_localdelaybala
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_localdelaybalancing'.
### Working on hdlcoder_localdelaybalancing/Subsystem/param_control/params as hdlsrc\hdlcoder_loc
### Working on hdlcoder_localdelaybalancing/Subsystem/param_control as hdlsrc\hdlcoder_localdelay
### Working on hdlcoder_localdelaybalancing/Subsystem/symmetric_fir as hdlsrc\hdlcoder_localdelay
### Working on hdlcoder_localdelaybalancing/Subsystem as hdlsrc\hdlcoder_localdelaybalancing\Subs
### Code Generation for 'hdlcoder_localdelaybalancing' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_localdelaybalancing' complete with 0 errors, 0 warnings, and 1 messag
### HDL code generation complete.
```



Now delay balancing is only active in the data path subsystem and the generated model and code do not contain any delays in the control path subsystem.

See Also

Related Examples

- “Delay Balancing and Validation Model Workflow in HDL Coder” on page 21-94
- “Use Delay Absorption While Modeling with Latency” on page 21-86

More About

- “Delay Balancing” on page 21-81
- “Resolve Numeric Mismatch with Delay Balancing” on page 21-25

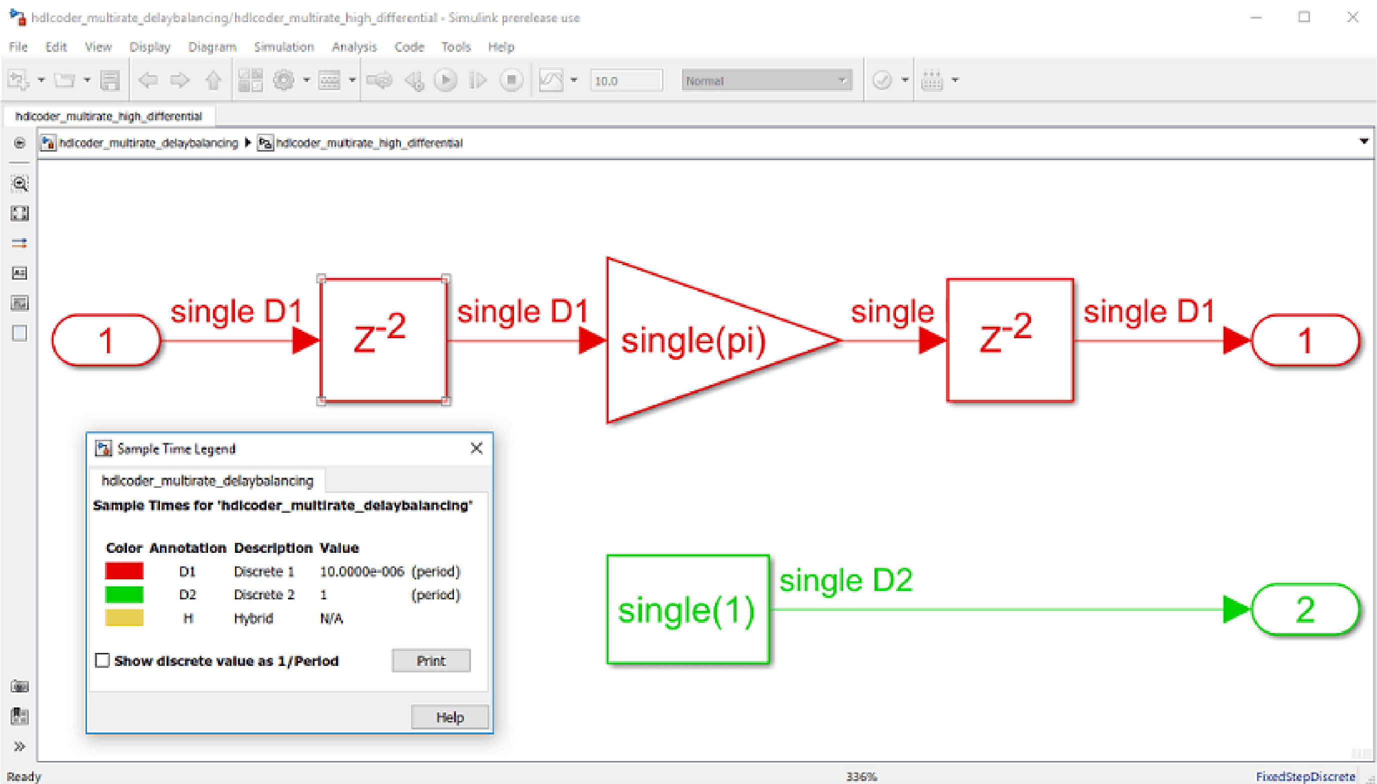
Delay Balancing on Multirate Designs

This example shows how an indiscrete usage of Simulate rates on a multirate design can generate an undesirable HDL code, and provides few recommendations for optimal code generation.

Introduction

This example model contains 3 subsystems, the first one demonstrates the issue and the others provide practical ways of resolving the issue.

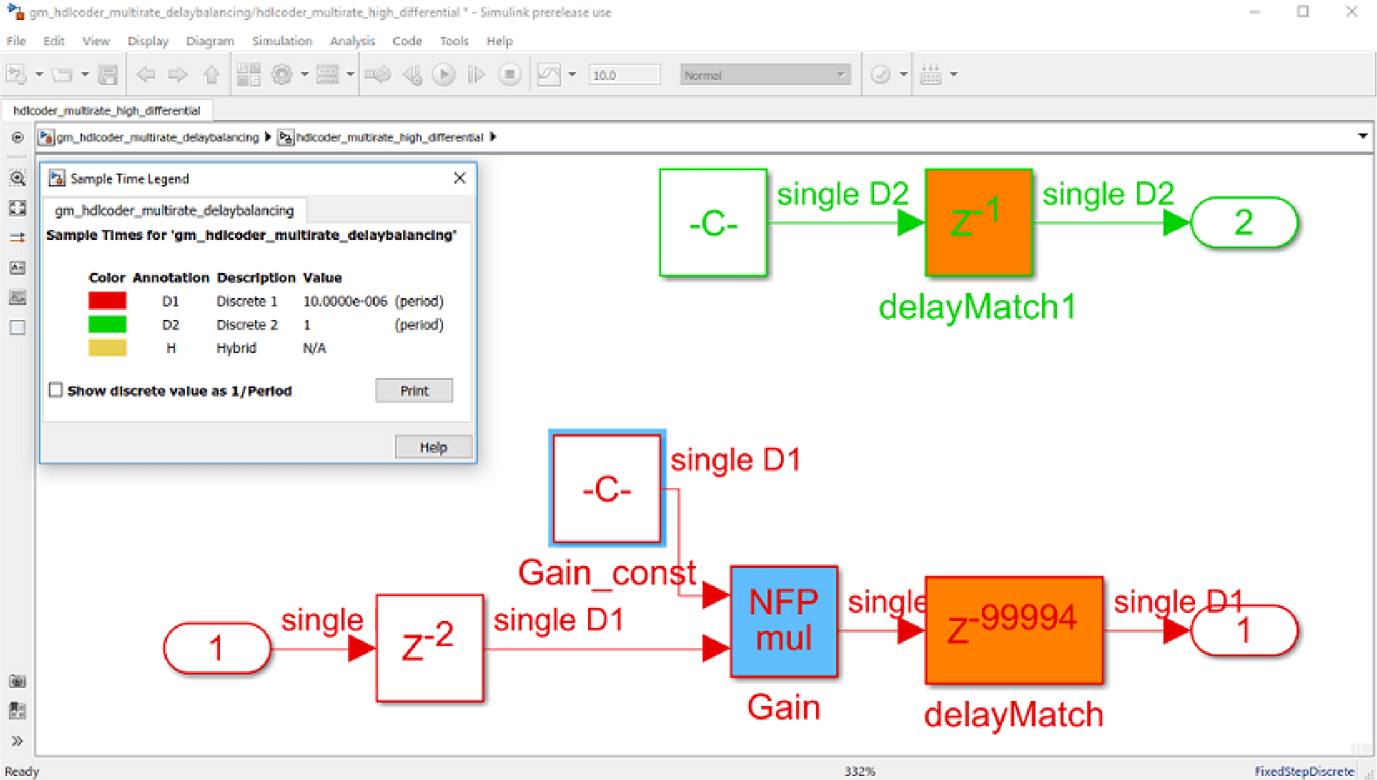
Note in the below design there are two islands of logics, both running at different rates. The rate differential between the two rates is 10E-06, which is a very high number and possibly unrealistic for practical FPGA design. This model has a floating-point Gain block, a multi-cycle operator, in the fast-clock region.



Running code generation on this model, we get:

```
>> makehdl(gcb)
### Generating HDL for 'hdlcoder_multirate_delaybalancing/hdlcoder_multirate_high_differential'.
### Using the config set for model hdlcoder_multirate_delaybalancing for HDL code generation parameters.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
### Output port 0: 100000 cycles.
### Output port 1: 1 cycles.
### Begin VHDL Code Generation for 'hdlcoder_multirate_delaybalancing'.
### Working on hdlcoder_multirate_delaybalancing/hdlcoder_multirate_high_differential/nfp_mul_comp as hdlsrc\hdlcoder_multirate_delaybalancing\nfp_mul_comp.vhd.
### Working on hdlcoder_multirate_high_differential_tc as hdlsrc\hdlcoder_multirate_delaybalancing\hdlcoder_multirate_high_differential_tc.vhd.
### Working on hdlcoder_multirate_delaybalancing/hdlcoder_multirate_high_differential as hdlsrc\hdlcoder_multirate_delaybalancing\hdlcoder_multirate_high_differential.vhd.
### Generating package file hdlsrc\hdlcoder_multirate_delaybalancing\hdlcoder_multirate_high_differential_pkg.vhd.
### Creating HDL Code Generation Check Report hdlcoder_multirate_high_differential_report.html
### HDL check for 'hdlcoder_multirate_delaybalancing' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The compiled generated model looks as below. Note that the high output latency on the fast clock rate region of the design are added to balance delays across multiple output paths of the system.

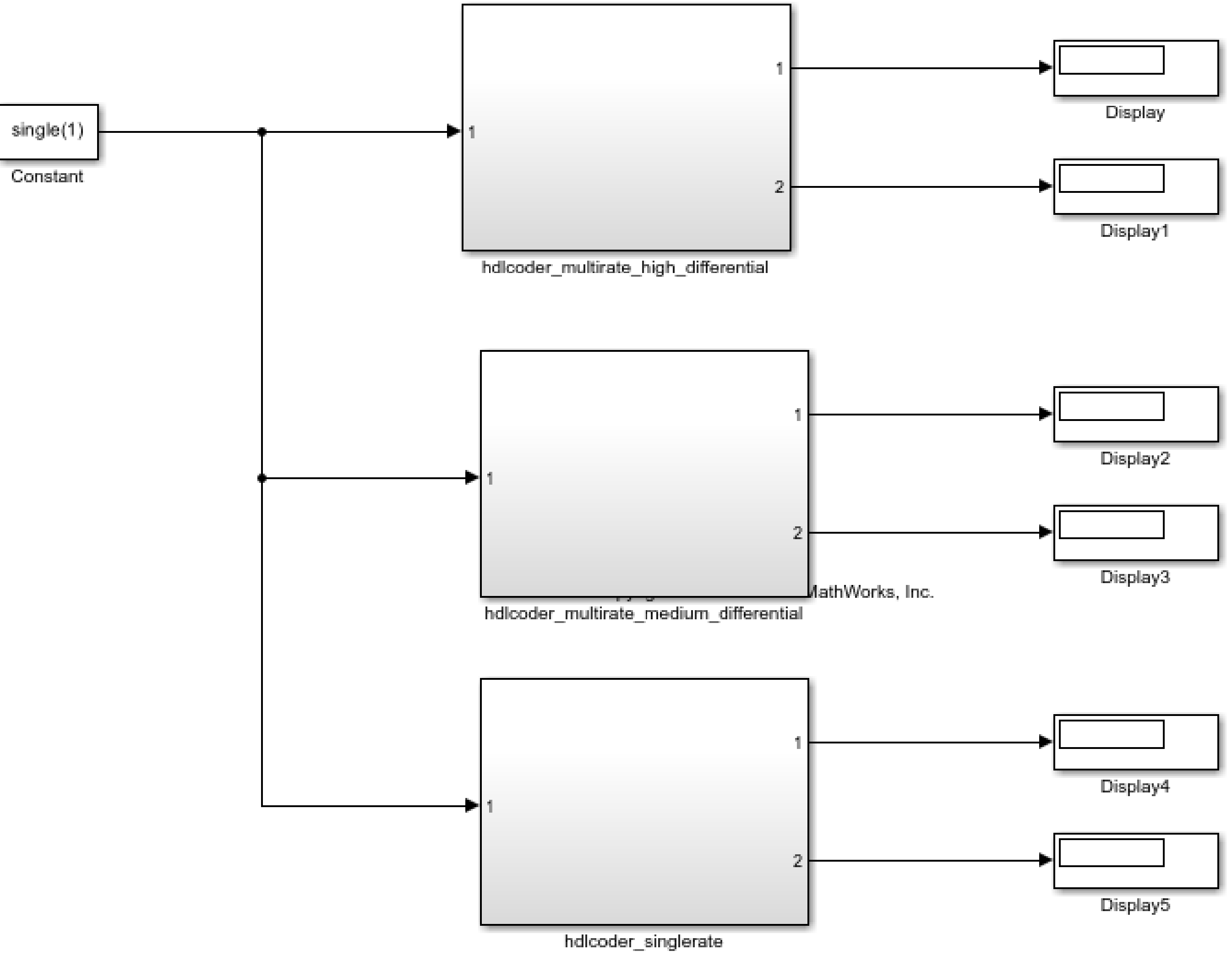


The high number of registers in the fast clock rate region has an undesired effect post HDL-code generation:

1. Generated HDL files are by itself very large.
2. The large number of pipeline registers will make fitting the design into an FPGA improbable.

The following sections of this document create a general awareness of the resource constraint that multi-rate models can create when used in the presence of multi-cycle operations, and provides a few recommendations for optimum resource usage.

```
open_system('hdlcoder_multirate_delaybalancing');
```



Recommended Guidelines

Your Simulink models might have different clock-rate paths due to different modeling reasons. In the presence of optimizations, like I/O pipelining, distributed pipelining, streaming and/or sharing, or multi-cycle operations, like floating-point IPs, fixed-point math functions like sqrt or divide, pipelines are introduced which are applied at the same rate at which the signal path operates.

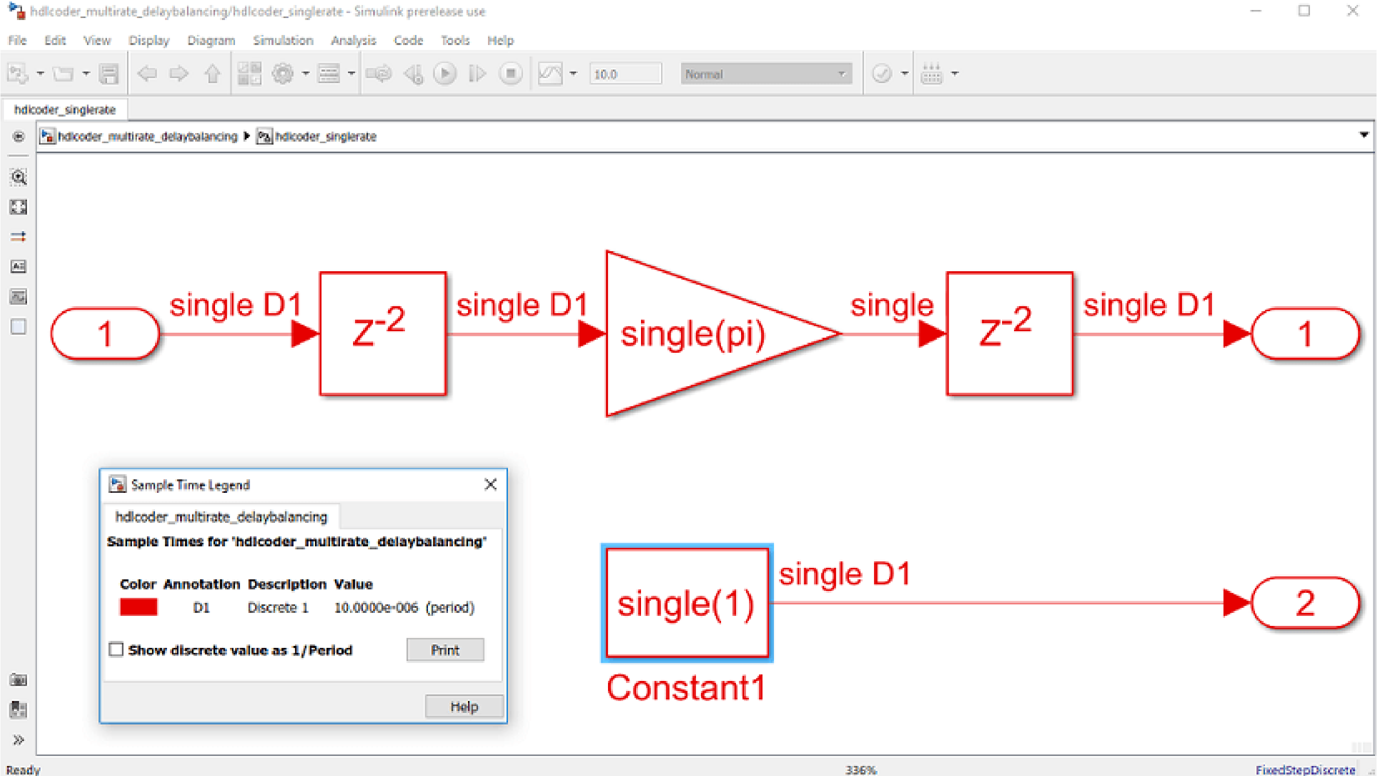
Introducing any additional pipelining introduces undesirable latency overhead that needs to be balanced across multiple output paths, operating at different rates. If the ratio difference between the fastest and slowest clock rate in the Simulink model is very large, it causes a large number of registers to be generated in the final HDL code. The HDL files become large and the design may not even fit into an FPGA.

Recommendation #1: Remove unintentional multirates

There might be an undesirable effect that the rate differential of the model has on the generated HDL code. For instance, in the above model, the sample rate specified on the constant block was not given

consideration and set to a value that caused a rate differential of 10E06 with the base model rate. Such a high rate differential appears unintentional.

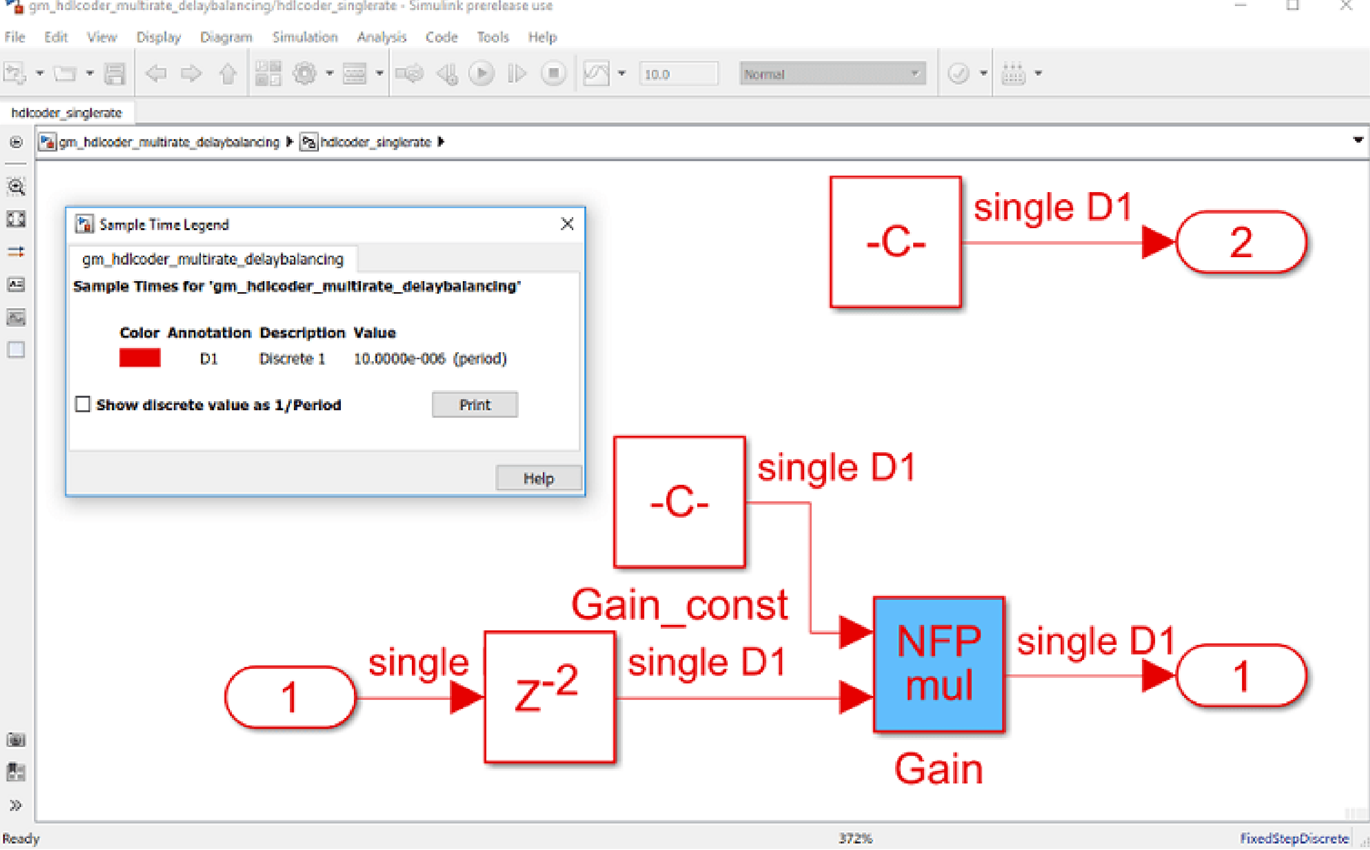
A recommendation is to change the sample rate of the constant block to run at the same rate as the base model, for such a situation.



Running code generation on this model, you get:

```
>> makehdl(gcb)
### Generating HDL for 'hdlcoder_multirate_delaybalancing/hdlcoder_singlerate'.
### Using the config set for model hdlcoder_multirate_delaybalancing for HDL code generation parameters.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
### Output port 0: 6 cycles.
### Output port 1: 6 cycles.
### Begin VHDL Code Generation for 'hdlcoder_multirate_delaybalancing'.
### Working on hdlcoder_multirate_delaybalancing/hdlcoder_singlerate/nfp_mul_comp as hdlsrc\hdlcoder_multirate_delaybalancing\nfp_mul_comp.vhd.
### Working on hdlcoder_multirate_delaybalancing/hdlcoder_singlerate as hdlsrc\hdlcoder_multirate_delaybalancing\hdlcoder_singlerate.vhd.
### Generating package file hdlsrc\hdlcoder_multirate_delaybalancing\hdlcoder_singlerate_pkg.vhd.
### Creating HDL Code Generation Check Report hdlcoder_singlerate_report.html
### HDL check for 'hdlcoder_multirate_delaybalancing' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

Note that the output latency numbers have decreased significantly. The compiled generated model looks like the image below.

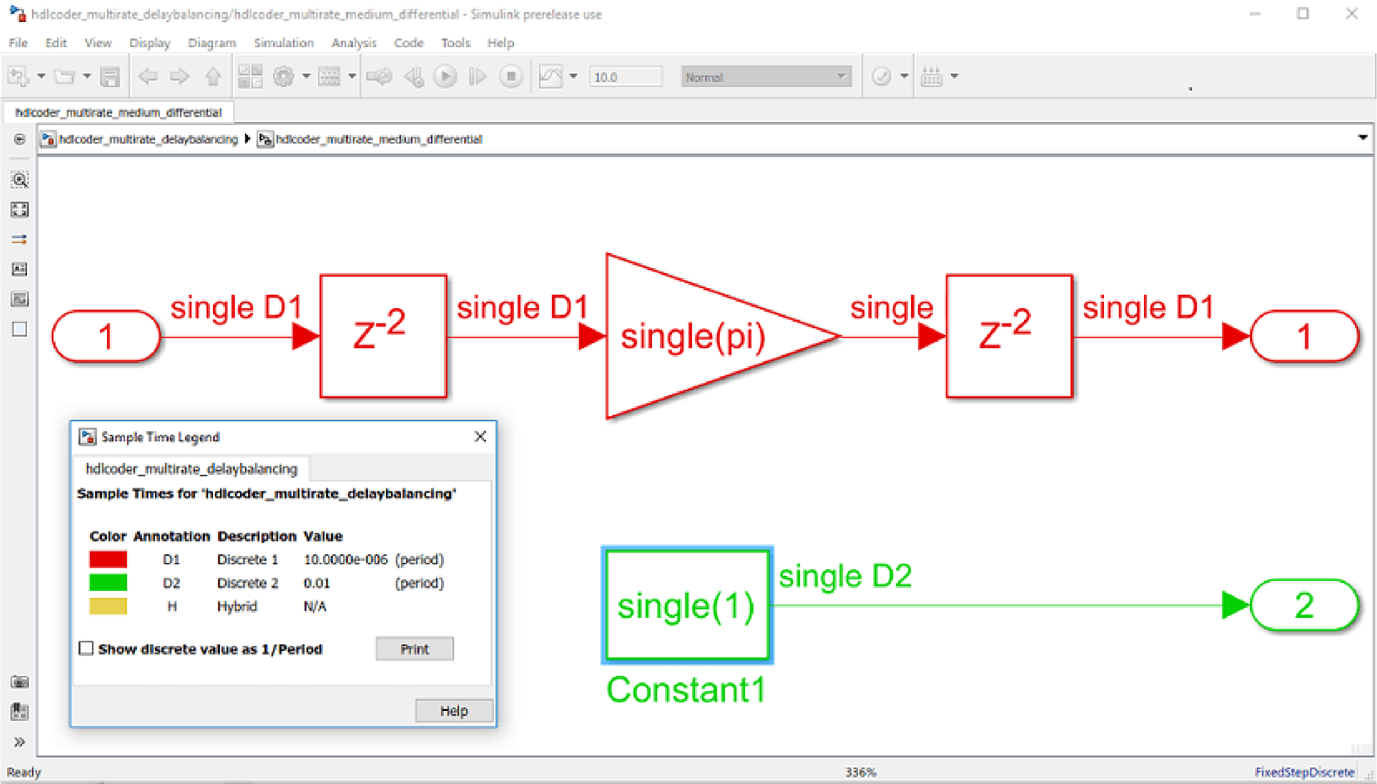


There is no undesirable high number of registers.

Recommendation #2: Keep rate differential practical

If multi-rate is a desirable property that you need, consider making the rate differential as practical as possible.

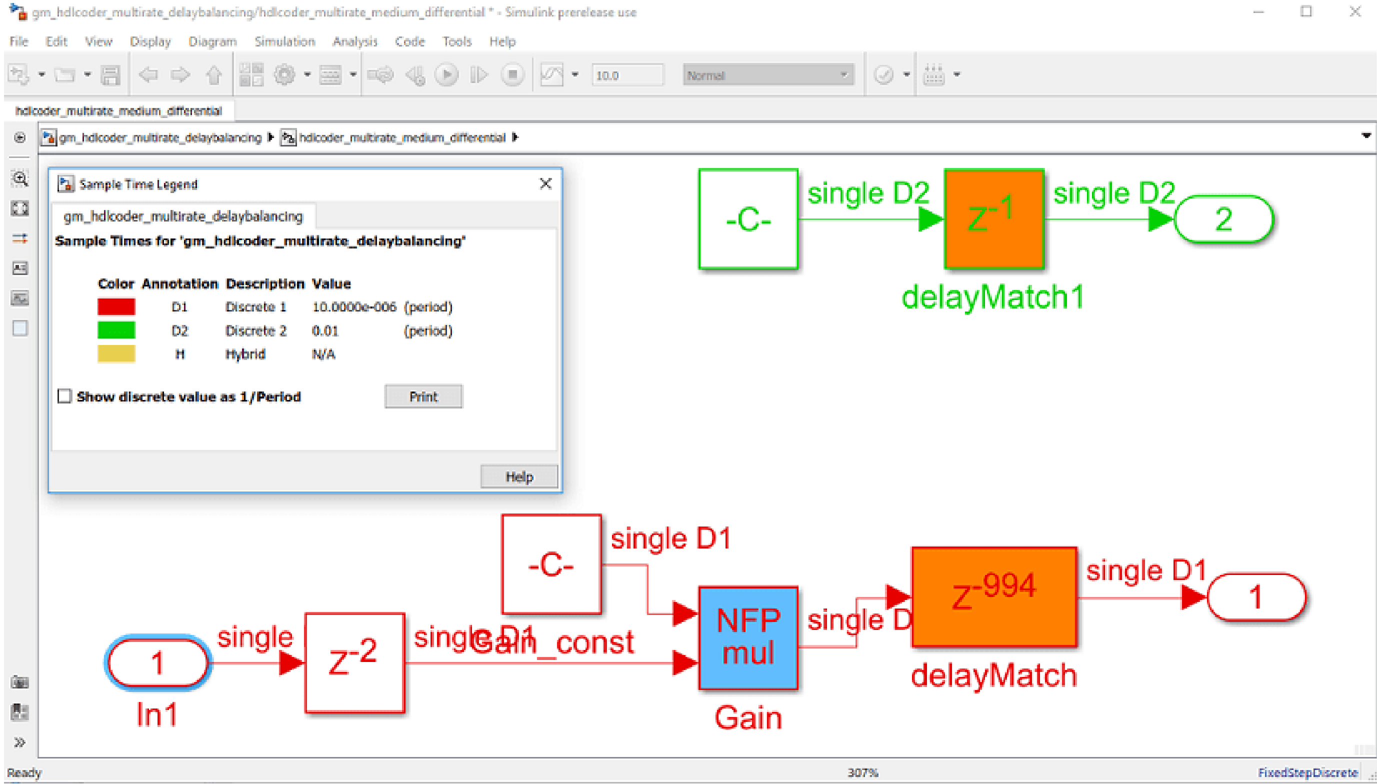
For example, if one path of the design running at 'ns' and other path of your design is at 'us' and is a desirable feature of the design, you can still choose to have multi-rate paths in his model with the awareness that delay balancing may cause high number of registers.



Running code generation on this model, we get:

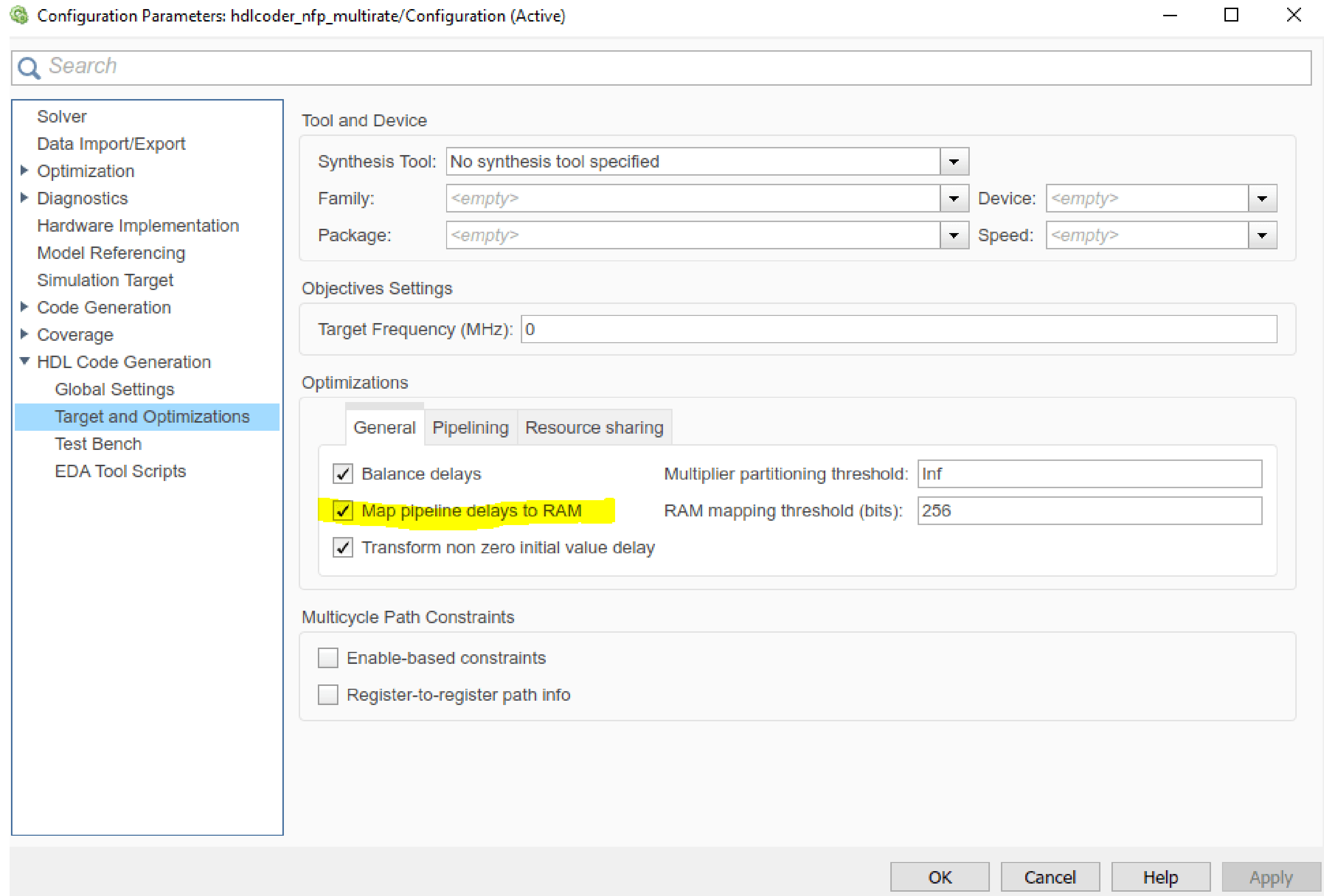
```
>> makehdl(gcb)
### Generating HDL for 'hdlcoder_multirate_delaybalancing/hdlcoder_multirate_medium_differential'.
### Using the config set for model hdlcoder_multirate_delaybalancing for HDL code generation parameters.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
### Output port 0: 1000 cycles.
### Output port 1: 1 cycles.
### Begin VHDL Code Generation for 'hdlcoder_multirate_delaybalancing'.
### Working on hdlcoder_multirate_delaybalancing/hdlcoder_multirate_medium_differential/nfp_mul_comp as hdlsrc\hdlcoder_multirate_delaybalancing\nfp_mul_comp.vhd.
### Working on hdlcoder_multirate_medium_differential_tc as hdlsrc\hdlcoder_multirate_delaybalancing\hdlcoder_multirate_medium_differential_tc.vhd.
### Working on hdlcoder_multirate_delaybalancing/hdlcoder_multirate_medium_differential as hdlsrc\hdlcoder_multirate_delaybalancing\hdlcoder_multirate_medium_differential.vhd.
### Generating package file hdlsrc\hdlcoder_multirate_delaybalancing\hdlcoder_multirate_medium_differential_pkg.vhd.
### Creating HDL Code Generation Check Report hdlcoder_multirate_medium_differential_report.html
### HDL check for 'hdlcoder_multirate_delaybalancing' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The compiled generated model looks like the figure below. In the generated model and HDL code, we will have close to 1000 registers in the fast clock rate output path. The additional cost of registers is **not unusual** for control logics that are running 1000x faster than the system. It is important to be aware of the hardware resource constraints for such a model.



To optimize on the total number of registers in FPGA, you can also use the model parameter **Map pipeline delays to RAM**. Doing this can tradeoff RAM resources to save on logic area.

```
>> hdlset_param(gcs, 'MapPipelineDelaysToRAM', 'on');
```


Configuration Parameters: hdlcoder_nfp_multirate/Configuration (Active)
Search
Solver
Data Import/Export
Optimization
Diagnostics
Hardware Implementation
Model Referencing
Simulation Target
Code Generation
Coverage
HDL Code Generation
Global Settings
Target and Optimizations
Test Bench
EDA Tool Scripts
Tool and Device
Synthesis Tool: No synthesis tool specified
Family: <empty>
Device: <empty>
Package: <empty>
Speed: <empty>
Objectives Settings
Target Frequency (MHz): 0
Optimizations
General
Pipelining
Resource sharing
Balance delays
Multiplier partitioning threshold: Inf
Map pipeline delays to RAM
RAM mapping threshold (bits): 256
Transform non zero initial value delay
Multicycle Path Constraints
Enable-based constraints
Register-to-register path info
OK
Cancel
Help
Apply

Find Feedback Loops

In this section...

"Specify Highlighting of Feedback Loops" on page 21-115
"Remove Highlighting" on page 21-115
"Limitations" on page 21-115

Feedback loops in your Simulink design can inhibit delay balancing and optimizations such as resource sharing and streaming.

To find feedback loops in your design that are inhibiting optimizations, you can generate and run a MATLAB script that highlights one or more feedback loops in your original model and the generated model. When you run the script, different feedback loops are highlighted in different colors. The feedback loop highlighting script is saved in the same target folder as the HDL code.

After you generate code, if feedback loops are inhibiting optimizations, the command window shows a link that you can click to highlight feedback loops. If you generate an Optimization Report, the report also contains a link you can click to highlight feedback loops.

The script can highlight feedback loops that are inhibiting the following optimizations:

- Resource sharing
- Streaming
- MATLAB variable pipelining
- Delay balancing

Specify Highlighting of Feedback Loops

By default, highlighting of feedback loops is enabled. To generate a feedback loop highlighting script programmatically, use the `HighlightFeedbackLoops` property with `makehdl` or `hdlset_param`. For example, to generate a feedback loop highlight script for a model, *`myModel`*, enter:

```
hdlset_param ('myModel', 'HighlightFeedbackLoops', 'on');
```

Remove Highlighting

By default, HDL Coder generates a script to highlight feedback loops and a script to clear the highlighting of feedback loops in your model. You can turn off highlighting using either of these ways:

- Click the `clearhighlighting` script in the MATLAB Command Window
- In the Simulink Toolstrip, select **Debug** > **Trace Signal**.

Limitations

- Feedback loop highlighting cannot highlight blocks that have names that contain a single quote (`'`).
- In some cases, feedback loop highlighting might highlight a subsystem or one block instead of the lowest-level block.

See Also

More About

- “Create and Use Code Generation Reports” on page 23-2
- “Generated Model and Validation Model” on page 21-10

Hierarchy Flattening

In this section...
“What Is Hierarchy Flattening?” on page 21-117
“When to Flatten Hierarchy” on page 21-117
“Considerations” on page 21-117
“How to Flatten Hierarchy” on page 21-117
“Hierarchy Flattening Report” on page 21-118
“Limitations for Hierarchy Flattening” on page 21-118

What Is Hierarchy Flattening?

Hierarchy flattening enables you to remove subsystem hierarchy from the HDL code generated from your design.

HDL Coder considers blocks within a flattened subsystem to be at the same level of hierarchy, and no longer grouped into separate subsystems. This consideration allows the coder to reorganize blocks for optimization across the original hierarchical boundaries, while preserving functionality.

When to Flatten Hierarchy

To preserve the modularity of the design and have a one-to-one mapping from subsystem name to corresponding HDL `module` or `entity` name, do not flatten the hierarchy. The generated HDL code is more readable when you don't flatten the hierarchy.

Flatten hierarchy to:

- Enable more extensive area and speed optimization.
- Reduce the number of HDL output files. For every subsystem that you flatten, HDL Coder generates one less HDL output file.

Considerations

- Before you flatten the hierarchy, you must have the `MaskParameterAsGeneric` property set to `off`. For more information, see Generate parameterized HDL code from masked subsystem.
- When you use optimizations such as resource sharing or streaming with hierarchy flattening, in certain cases, HDL Coder might retain the subsystem hierarchy in the generated model. However, the HDL code generated for the flattened subsystems is inlined, which reduces the number of HDL files.
- When you use floating-point data types in `Native Floating Point` mode, HDL Coder might not flatten the hierarchy. This is because floating-point designs generate hundreds of lines of code and inlining the HDL files makes the generated code less readable.

How to Flatten Hierarchy

By default, a subsystem inherits its hierarchy flattening setting from the parent subsystem. However, you can enable or disable flattening for individual subsystems. This table lists options you can specify for hierarchy flattening options for a subsystem are listed in the following table.

Hierarchy Flattening Setting	Description
inherit (default)	Use the hierarchy flattening setting of the parent subsystem. If this subsystem is the highest-level subsystem, do not flatten.
on	Flatten this subsystem.
off'	Do not flatten this subsystem, even if the parent subsystem is flattened.

To set hierarchy flattening using the HDL Block Properties dialog box:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the Subsystem and then click **HDL Block Properties**. For **FlattenHierarchy**, select **on**, **off**, or **inherit**.
- Right-click the Subsystem and select **HDL Code > HDL Block Properties**. For **FlattenHierarchy**, select **on**, **off**, or **inherit**.

To set hierarchy flattening from the command line, use `hdlset_param`. For example, to turn on hierarchy flattening for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'FlattenHierarchy', 'on')
```

See also `hdlset_param`.

Hierarchy Flattening Report

To see the hierarchy flattening information in the report, before you generate code for each subsystem or model reference, enable the optimization report. To enable this report, in the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

The report displays subsystems in your model that have **FlattenHierarchy** set to `on` and `off`, hierarchy flattening status, and the HDL files that are inlined. You can use the report to more effectively flatten the subsystem hierarchy and improve opportunities for optimizations such as clock-rate pipelining on the model.

If hierarchy flattening is unsuccessful, the report shows a table that contains subsystems that are not flattened, and reasons for not flattening the subsystem. Subsystems that have a * highlighted beside it indicates whether the HDL files are inlined though hierarchy flattening failed.

Limitations for Hierarchy Flattening

A subsystem cannot be flattened if the subsystem is:

- A Synchronous Subsystem or uses the State Control block in `Synchronous` mode.
- A model reference implementation.
- A Triggered Subsystem when Use trigger signal as clock is enabled.

See Also

`makehdl`

More About

- “Clock-Rate Pipelining” on page 21-148

- “Resource Sharing” on page 21-45
- “Streaming” on page 21-42
- “Create and Use Code Generation Reports” on page 23-2

Apply RAM Mapping to Optimize Area

In this section...

“RAM Mapping for a Simulink Model” on page 21-120
“RAM Mapping for a MATLAB Design” on page 21-121
“Use the RAM Mapping Threshold” on page 21-121
“Exclude Inefficient RAM Mapping” on page 21-122

RAM mapping is an area optimization that maps storage and delay elements in your Simulink model or MATLAB code to random access memory (RAM) rather than to individual registers on hardware. RAM mapping can reduce the amount of area that your design uses in the target hardware by reducing the number of registers that elements consume and store those elements in RAM blocks.

RAM Mapping for a Simulink Model

You can map these Simulink model elements to RAM:

- Delay blocks. To map delays to RAM, set the HDL block property UseRAM. For guidelines, see “Map Large Delays to Block RAM” on page 18-106.
- Persistent variables in MATLAB Function blocks. To map persistent arrays in a MATLAB Function block to RAM, use the pragma `coder.hdl.ramconfig` or set the HDL block property MapPersistentVarsToRAM. For an example, see “RAM Mapping with the MATLAB Function Block” on page 21-125.
- Lookup tables. To map lookup tables to RAM, set the HDL block property MapToRAM.
- RAM blocks from the HDL Operations library:
  - Single Port RAM
  - Single Port RAM System
  - Dual Port RAM
  - Dual Port RAM System
  - Simple Dual Port RAM
  - Simple Dual Port RAM System
  - Dual Rate Dual Port RAM
  - Simple Tri Port RAM System
  - True Dual Port RAM System
- Blocks with a RAM implementation.

Specify RAM Mapping for a Simulink Model

To specify RAM mapping for persistent arrays in a MATLAB Function block or Delay blocks:

1. Set the associated RAM mapping property for the elements that you want to map to RAM. For example, to map a Delay block to RAM, set the **UseRAM** HDL block property to `on`. To map persistent arrays, set the MATLAB Function block HDL block property **MapPersistentVarsToRAM** to `on` or use the pragma `coder.hdl.ramconfig` in your MATLAB Function block after defining the persistent variables.

2. Specify a minimum RAM mapping threshold by using the **RAM mapping threshold** configuration parameter. See Map pipeline delays to RAM and RAM mapping threshold.

RAM Mapping for a MATLAB Design

You can map these MATLAB code elements to RAM:

- Persistent array variables.
- Pipeline registers in the generated HDL code.
- `dsp.Delay` System object.
- `hdl.RAM` System object.

Specify RAM Mapping for a MATLAB Design

To specify RAM mapping for delays or persistent arrays in a MATLAB function, in the **Optimizations** section of the **HDL Code Generation** tab of the MATLAB HDL Workflow Advisor:

1. Set the associated RAM mapping property for the elements that you want to map to RAM. For example, to map persistent arrays, select the **Map persistent array variables to RAMs** option. For more information, see “Map Persistent Arrays and dsp.Delay Objects to RAM” on page 8-6. To map pipelines delays to RAM, select the **Map pipeline delays to RAM** option.
2. Specify a value for the **RAM mapping threshold** option. For more information, see “RAMThreshold”.

For an example, see “Map Matrices to Block RAMs to Reduce Area” on page 8-2.

You can also map persistent array variables to RAM by using the pragma `coder.hdl.ramconfig` in your MATLAB code after you define and assign persistent variables in the code. `coder.hdl.ramconfig` allows you to select persistent variables to map to RAM and to map persistent variables to a specific RAM architecture, such as a simple tri port RAM. For more information, see `coder.hdl.ramconfig`.

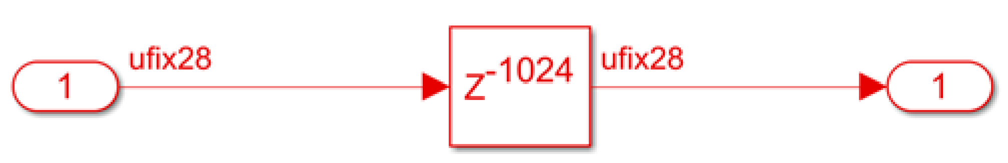
Use the RAM Mapping Threshold

When you use the configuration parameter **RAM mapping threshold (bits)**, you can specify a single integer to define the mapping threshold that maps any delay or persistent array greater than that threshold bit size to RAM. For example, if your design contains a Delay block with a delay length of 1024 cycles and a word length of 28 bits and you set the HDL block property **UseRAM** to `on` and the **RAM mapping threshold** parameter to `256` total bits, the total RAM size of 28672 bits of the Delay block exceeds the threshold and HDL Coder maps 1024x28 bits to the RAM on hardware. To calculate the total RAM size, use this formula:

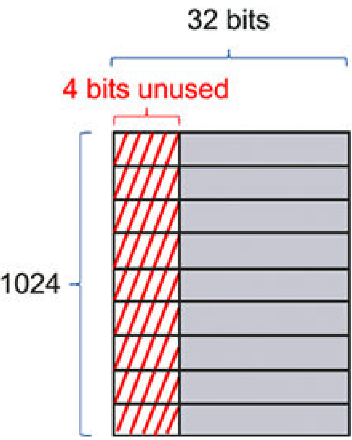
```
RAMSize = Delay length * Word length * Vector length * Complexity
```

In this case, the input vector length is 1 and the complexity of a real data type is 1, so the total RAM size equation is:

```
RAMSize = Delay length * Word length = 1024*28
```



With a block RAM (BRAM) size on the target hardware of 1024x32 bits, the delay is mapped with 4 bits of unused width for each row of RAM. As a result, depending on the shape of the data defined by delay length and word length, RAM mapping might be inefficient.



RAM Merging

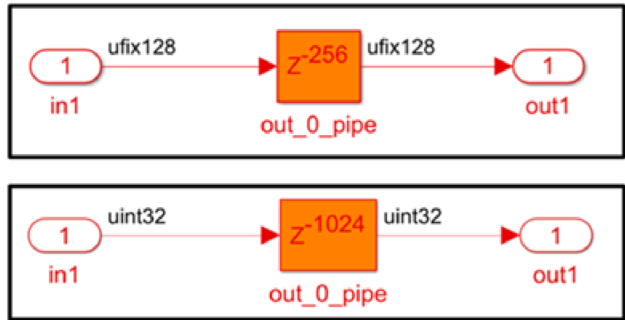
RAM merging is a process that merges several delays and maps them to RAM together. During HDL code generation, RAM merging occurs when you have multiple Delay blocks with the same delay length in the same subsystem. RAM merging fuses the delays into a single delay that has a RAM size equal to the sum of the original RAM sizes. If this new RAM size is greater than the **RAM mapping threshold** value, the merged delays are mapped to RAM. For example, if you have three Delay blocks in a subsystem that each have a total RAM size of 40 bits and the **RAM mapping threshold** is `100`, all three delays map to RAM because they have a total of `3x40=120` bits when they merge.

Exclude Inefficient RAM Mapping

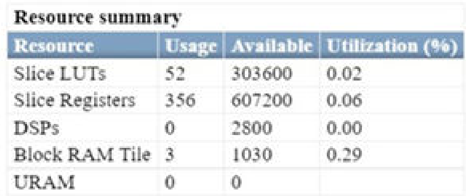
To exclude inefficient mapping, you can specify two thresholds to define the shape of the data to map to RAM, one for delay length (for delays) or array size (for persistent array variables) and one for word length or bit width of the data type. Setting both thresholds maps delays or persistent arrays more efficiently to RAM by excluding delays or persistent arrays that inefficiently map to block RAM on the target hardware for your design. These thresholds allow you to selectively map data that has a shape similar to specific block RAM configuration on your target hardware.

Set Total RAM Size for RAM Mapping Threshold

For example, for a model with **RAM Mapping Threshold** set to `256`, both of these subsystems in the DUT in the generated model are mapped to block RAM on the Xilinx RAMB36E1 because both subsystems contain delays with a total bit size of 32768 bits, which is greater than the threshold, even with different delay lengths and bit widths.



To determine how many block RAMs are used from the design, run synthesis on the model. For more information on how to run synthesis, see “HDL Code Generation and FPGA Synthesis from Simulink Model”. This is the Resource summary report from synthesis:



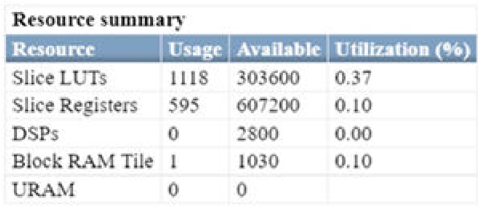
Resource summary

Resource	Usage	Available	Utilization (%)
Slice LUTs	52	303600	0.02
Slice Registers	356	607200	0.06
DSPs	0	2800	0.00
Block RAM Tile	3	1030	0.29
URAM	0	0	

The BRAM on the Xilinx RAMB36E1 can be configured for both 512x64 bits and 1024x32 bits size. The first subsystem inefficiently maps to block RAM and takes up two BRAM of size 512x64 bits, while the second subsystem maps more efficiently to one BRAM of size 1024x32 bits, resulting in a total of 3 block RAM being utilized.

Set Delay Length and Word Length for RAM Mapping Threshold

For this example, the **RAM Mapping Threshold** is changed to `1024x1`, where 1024 is the delay length threshold and 1 is the word length threshold. To determine how many block RAMs are used, run synthesis on the model. This is the Resource summary report from synthesis:



Resource summary

Resource	Usage	Available	Utilization (%)
Slice LUTs	1118	303600	0.37
Slice Registers	595	607200	0.10
DSPs	0	2800	0.00
Block RAM Tile	1	1030	0.10
URAM	0	0	

Now, only the second subsystem is mapped to BRAM because the first subsystem has a delay length of 256 cycles, which is less than the delay length threshold of 1024. As a result, only one BRAM is used to map the second subsystem. The first subsystem is mapped to LUTs and registers.

See Also

Model Settings
Map pipeline delays to RAM | **RAM mapping threshold**

More About

- "MATLAB Function Block Design Patterns for HDL" on page 27-14
- "RAM Mapping with the MATLAB Function Block" on page 21-125

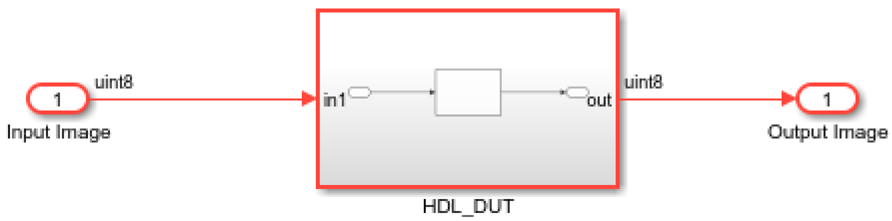
RAM Mapping with the MATLAB Function Block

This example shows how to map persistent arrays to RAM by using the MapPersistentVarsToRAM block-level parameter. The RAM size must be greater than or equal to the RAMMappingThreshold. The resource report shows the difference in area improvements resulting from RAM mapping.

Line Buffer Model

Open the model hdlcoder_ram_mapping_matlab_function.

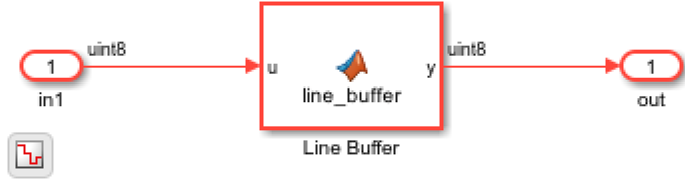
```
open_system('hdlcoder_ram_mapping_matlab_function')
set_param('hdlcoder_ram_mapping_matlab_function', 'SimulationCommand', 'Update')
```



Copyright 2019-2021 The MathWorks, Inc.

The DUT Subsystem in the model drives a Line Buffer MATLAB Function block.

```
open_system('hdlcoder_ram_mapping_matlab_function/HDL_DUT')
```



To see the MATLAB® code implementation of the line buffer, open the MATLAB Function block.

```
open_system('hdlcoder_ram_mapping_matlab_function/HDL_DUT/Line Buffer')
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Line buffer: Uses a presistent array to store the image
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function y = line_buffer(u)
%3codegen

persistent u_d ctr;

if isempty(u_d)
    u_d = uint8(zeros(1,80)); % You can map this to RAM
    ctr = uint8(1);
end

y = u_d(ctr);
u_d(ctr) = u;

if ctr == uint8(80)
    ctr = uint8(1);
else
    ctr = ctr + 1;
end

end
```

Generate HDL Code

1. Enable generation of the resource utilization report. The report displays the number of adders, subtractors, multipliers, registers, and RAMs that the design consumes.

```
hdlset_param('hdlcoder_ram_mapping_matlab_function', 'resourcereport', 'on')
```

2. Generate HDL code for the `HDL_DUT` Subsystem.

```
makehdl('hdlcoder_ram_mapping_matlab_function/HDL_DUT')
```

HDL Coder™ displays the Code Generation Report. In the report, select the **High-Level Resource Report** section. The design consumes `81` registers and `648` 1-bit registers. By default, the `MapPersistentVarsToRAM` property is disabled and the code generator does not infer or consume RAM resources.

Multipliers	0
Adders/Subtractors	3
Registers	81
Total 1-Bit Registers	648
RAMs	0
Multiplexers	1
I/O Bits	20
Static Shift operators	0
Dynamic Shift operators	0

Enable RAM Mapping and Generate HDL Code

1. Enable the `MapPersistentVarsToRAM` HDL parameter on the MATLAB Function block.

```
ml_subsys = 'hdlcoder_ram_mapping_matlab_function/HDL_DUT/Line Buffer';
hdlset_param(ml_subsys, 'MapPersistentVarsToRAM', 'on')
```

2. Generate HDL code for the `HDL_DUT` Subsystem.

```
makehdl('hdlcoder_ram_mapping_matlab_function/HDL_DUT')
```

In the Code Generation Report, select the **High-Level Resource Report** section. The design consumes one register, eight 1-bit registers, and one RAM. The number of RAMs inferred depends on the `RAMMappingThreshold` that you specify. See RAM mapping threshold.

Multipliers	0
Adders/Subtractors	3
Registers	1
Total 1-Bit Registers	8
RAMs	1
Multiplexers	3
I/O Bits	20
Static Shift operators	0
Dynamic Shift operators	0

RAM Mapping with MATLAB Datapath Architecture

The `MATLAB Datapath` architecture treats the MATLAB Function block like a regular Subsystem. The architecture converts the MATLAB code that you wrote to a dataflow representation in

Simulink®. HDL Coder can then more widely use optimizations across the MATLAB Function block with other Simulink blocks in your model.

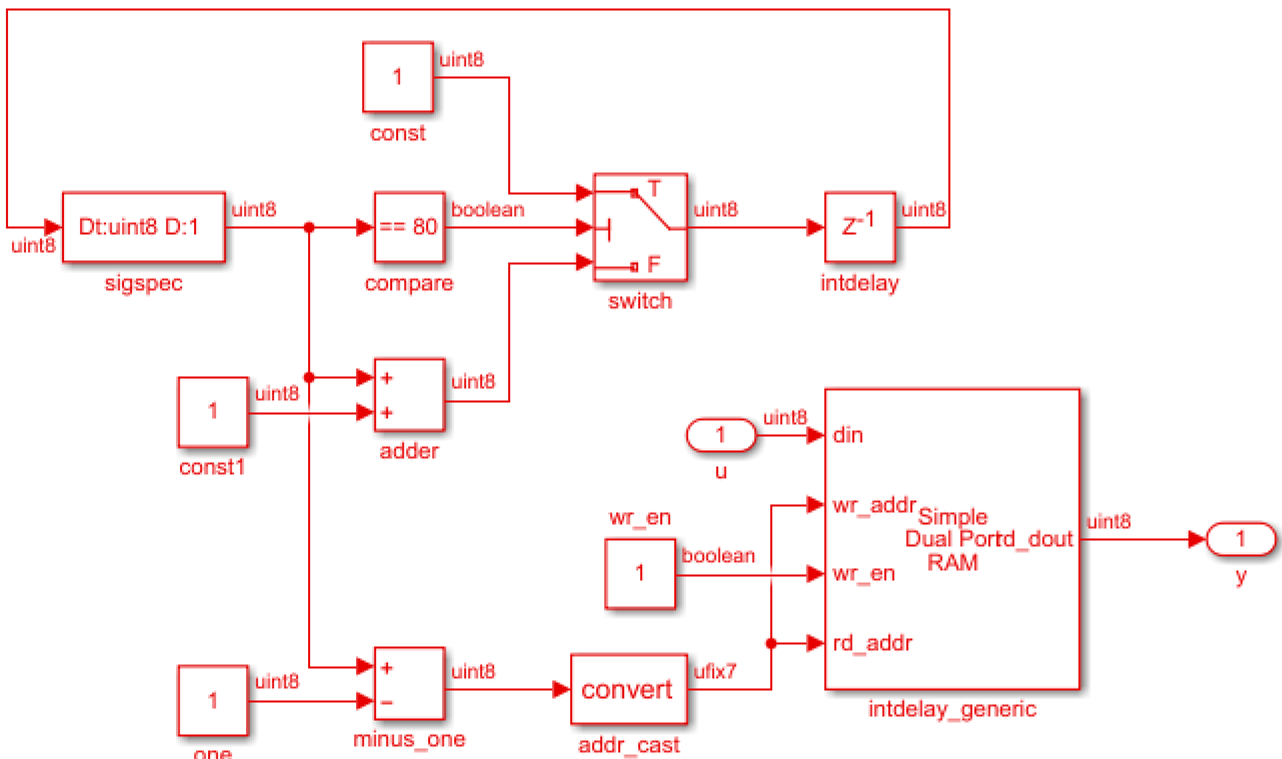
1. Enable the `MATLAB Datapath` HDL architecture and then set the `MapPersistentVarsToRAM` parameter on the MATLAB Function block.

```
hdlset_param(ml_subsys, 'Architecture', 'MATLAB Datapath')
hdlset_param(ml_subsys, 'MapPersistentVarsToRAM', 'on')
```

2. Generate HDL code for the `HDL_DUT` Subsystem.

```
makehdl('hdlcoder_ram_mapping_matlab_function/HDL_DUT')
```

The **High-Level Resource Report** indicates that the design consumes the same number of resources as the design that used the default architecture of the MATLAB Function block. To see how `MATLAB Datapath` architecture modifies the MATLAB code to a Simulink dataflow representation, open the generated model `gm_hdlcoder_ram_mapping_matlab_function` and navigate to the `HDL_DUT` Subsystem. There is a `Line Buffer` Subsystem in place of the MATLAB Function block. Inside the Subsystem block is the dataflow representation that displays a RAM block inferred.



To learn about design patterns that enable efficient RAM mapping of persistent arrays in MATLAB Function blocks, see the `eml_hdl_design_patterns/RAMs` library.

See Also

Model Settings
Map pipeline delays to RAM | RAM mapping threshold

More About

- “MATLAB Function Block Design Patterns for HDL” on page 27-14
- “MapPersistentVarsToRAM” on page 19-17

Distributed Pipelining

In this section...

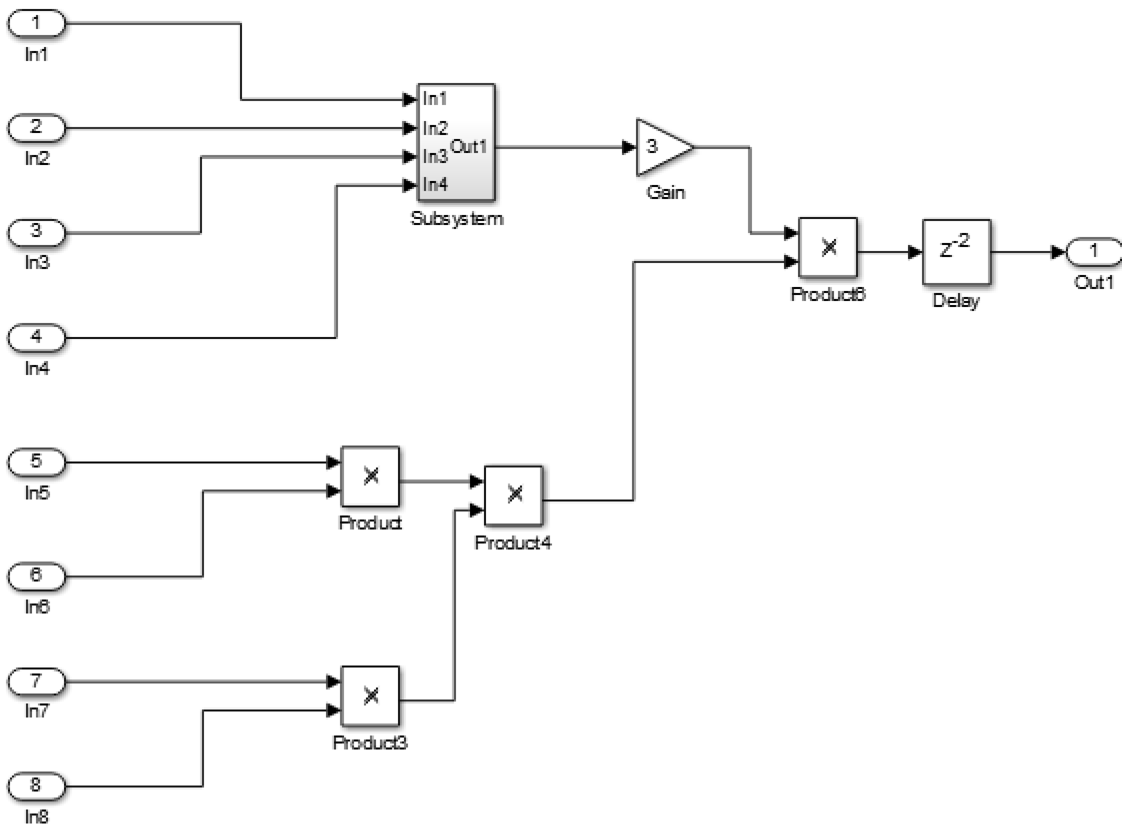
"What Is Distributed Pipelining?" on page 21-130
"Benefits and Costs of Distributed Pipelining" on page 21-131
"How Distributed Pipelining Works" on page 21-131
"Requirements for Distributed Pipelining" on page 21-132
"Specify Distributed Pipelining" on page 21-132
"Distributed Pipelining Report" on page 21-133
"Limitations of Distributed Pipelining" on page 21-133
"Selected Bibliography" on page 21-135

What Is Distributed Pipelining?

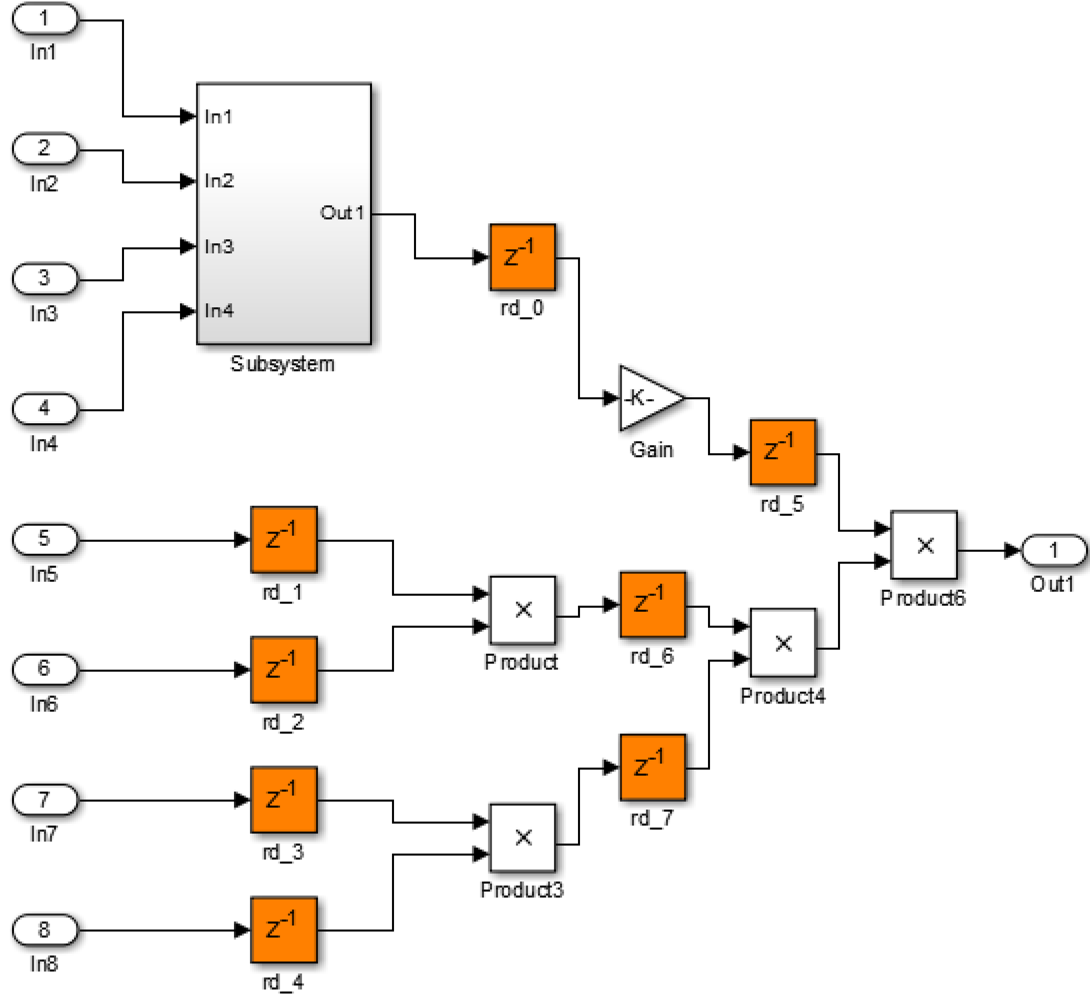
Distributed pipelining, or register retiming, is a speed optimization that moves existing delays in a design to reduce the critical path while preserving functional behavior. This optimization moves the delays within a subsystem while preserving the hierarchy.

The HDL Coder software uses an adaptation of the Leiserson-Saxe retiming algorithm.

For example, in this model, there is a delay of 2 at the output.



The diagram shows the generated model after distributed pipelining redistributes the delay to reduce the critical path.



Benefits and Costs of Distributed Pipelining

Distributed pipelining can reduce your design's critical path, enabling you to use a higher clock rate and increase throughput.

However, distributed pipelining requires your design to contain a number of delays. If you need to insert additional delays in your design to enable distributed pipelining, this increases the area and the initial latency of your design.

How Distributed Pipelining Works

HDL Coder applies distributed pipelining with other optimization options to move the existing delays in your design to reduce the critical path. The existing delays in your design can either be design delays or pipeline delays. Design delays are delays that you manually insert in your design by using Delay blocks, or other blocks that have state, including Queue, HDL FIFO, or Buffer blocks. Pipeline

delays are delays that are generated by optimization settings, such as input or output pipelining options, or block implementation settings, such as the `ShiftAdd` implementation for a Divide block and the `CORDIC` approximation for a Trigonometric Function block. See “InputPipeline” on page 19-14 and “OutputPipeline” on page 19-19.

When you generate code, before distributed pipelining runs, the input and output pipelines you specify are inserted at the input and outputs ports in your design. HDL Coder uses these pipelines to determine latency requirements and an outline for ideal pipelining for your design.

HDL Coder begins distributed pipelining by calculating the propagation delay for the components in your design by assigning each component an equal weight, except for wire components, such as Selector blocks and Bit Concat blocks, that are assigned zero-propagation delay. Distributed pipelining then redistributes the design delays and pipeline delays based on the weight assigned to each component in your design. Distributed pipelining takes into consideration all of the pipeline stages and distributes them as evenly as possible in order to obtain the shortest critical path. The inserted pipelines might not appear in the place you originally requested them because distributed pipelining can move these to improve the critical path.

If pipelines are required at specific point after a block, you can set the HDL Block Property **ConstrainedOutputPipeline** on the block at the specified point to the number of pipelines that you want to keep as output pipelines after that block. Constrained output pipelining does not insert extra pipelines, but instead provides information to distributed pipelining to first prioritize pipelines moved to or left at that specific point in your design. This option is used only to prevent distributed pipelining from moving specified pipelines or to require distributed pipelining to move pipelines to that specified point. For more information, see “Constrained Output Pipelining” on page 21-146.

Note If you want to disable distributed pipelining from running on your subsystem and you have set constrained output pipelining on blocks in your design, disable distributed pipelining for your subsystem and reset the **ConstrainedOutputPipeline** property on the blocks to zero.

If you do not want the design delays you have in your design to be moved by distributed pipelining, disable **Allow design delay distribution**. See Allow design delay distribution.

Requirements for Distributed Pipelining

Distributed pipelining requires your design to contain delays or registers that can be redistributed. You can use input pipelining or output pipelining to insert more registers. You can insert these delays manually or insert them by setting the HDL block properties **InputPipeline** or **OutputPipeline** for the subsystem or block that you plan to set distributed pipelining. For more information, see “InputPipeline” on page 19-14 and “OutputPipeline” on page 19-19.

If your design does not meet your timing requirements at first, try adding more delays or registers to improve your results.

Specify Distributed Pipelining

You can set distributed pipelining on a model or the top-level DUT subsystem. For finer control, you can set distributed pipelining on subsystems, Stateflow charts, and MATLAB Function blocks in the top-level DUT subsystem. See “Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks” on page 27-28.

To enable distributed pipelining for the model from the UI:

1. In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
2. Click **Settings**. In the **HDL Code Generation** > **Optimization** pane, in the **Pipelining** tab, select **Distributed pipelining** and click **OK**.

To enable distributed pipelining for a model, on the command line, enter:

```
hdlset_param('modelname', 'DistributedPipelining', 'on')
```

If you have a top-level device under test (DUT) that contains a subsystem hierarchy, such as lower-level subsystems, and you want distributed pipelining to run throughout the DUT and the lower-level subsystems, enable the model configuration parameter **Distributed pipelining** and leave the HDL block property **DistributedPipelining** as `inherit` for the DUT and all lower-level subsystems.

To prevent distributed pipelining from running in a specific lower-level subsystem in the DUT, set the HDL block property **DistributedPipelining** to `off` for that subsystem.

Note If you insert pipeline registers, output data can be in an invalid state initially. To avoid test bench errors resulting from initial invalid samples, disable output checking for those samples. For more information, see Ignore output data checking (number of samples).

Distributed Pipelining Report

To see the distributed pipelining information in the report, before you generate code for each subsystem or model reference, enable the optimization report. In the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

When you generate the optimization report, in the **Distributed Pipelining** section, you see the effect of the distributed pipelining optimization for the subsystems that have distributed pipelining enabled in the generated model and in the **Distributed Pipelining Summary**. If distributed pipelining is unsuccessful, the report shows diagnostic messages and blocks that caused distributed pipelining to fail.

When distributed pipelining encounters barriers, the HDL Coder generates a highlighting script named `highlightDistributedPipeliningBarriers.m`. This script highlights all the barriers present in the model.

If you set the HDL block property **ConstrainedOutputPipeline** for a block in your design, the **Constrained Output Pipeline Summary** displays the requested constrained output pipelines and if those requests are met during code generation.

If distributed pipelining is successful, the **Detailed Report** in the **Distributed Pipelining** section displays comparative listings of registers before and after you apply the distributed pipelining transform.

Limitations of Distributed Pipelining

The distributed pipelining optimization has these limitations:

- Your pipelining results might not be optimal in hardware because the operator latencies in your target hardware might differ from the estimated operator latencies used by the distributed pipelining algorithm.
- HDL Coder generates pipeline registers at the outputs instead of distributing the registers to reduce critical path in these situations:
  - Stateflow chart containing a state, local variable, or a matrix with statically unresolvable index.
- HDL Coder distributes pipeline registers around these blocks instead of within them:
  - Model
  - Sum (`Cascade` implementation)
  - Divide
  - Reciprocal
  - Sqrt
  - Sine HDL Optimized and Cosine HDL Optimized
  - Product (`Cascade` implementation)
  - MinMax
  - Upsample
  - Downsample
  - Rate Transition
  - Zero-Order Hold
  - Reciprocal Sqrt (`RecipSqrtNewton` implementation)
  - Trigonometric Function (`CORDIC` Approximation)
  - Single Port RAM
  - Dual Port RAM
  - Simple Dual Port RAM
  - Blocks with floating-point IP (i.e. native floating-point)
- If you enable distributed pipelining for a subsystem that contains these blocks, HDL Coder generates a warning message during code generation in the HDL Code Generation Check Report. This message identifies blocks that act as barriers for distributed pipelining in the subsystem. HDL Coder distributes pipeline registers around nested subsystems.
  - M-PSK Demodulator Baseband
  - M-PSK Modulator Baseband
  - QPSK Demodulator Baseband
  - QPSK Modulator Baseband
  - BPSK Demodulator Baseband
  - BPSK Modulator Baseband
  - PN Sequence Generator
  - Repeat
  - HDL Counter
  - LMS Filter

- Sine Wave
- Viterbi Decoder
- Triggered Subsystem
- Counter Limited
- Counter Free-Running

Selected Bibliography

Leiserson, C.E, and James B. Saxe. “Retiming Synchronous Circuitry.” *Algorithmica*. Vol. 6, Number 1, 1991, pp. 5-35.

See Also

Properties

“DistributedPipelining” on page 19-9

Model Settings

Distributed pipelining | Pipeline distribution priority | Use synthesis estimates for distributed pipelining

See Also

Related Examples

- “Distributed Pipelining: Speed Optimization” on page 21-139
- “Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks” on page 27-28
- “Iteratively Maximize Clock Frequency by Using Speed Optimizations” on page 21-167

More About

- “Recommended Distributed Pipelining Settings” on page 18-161
- “Distributed Pipelining Using Synthesis Timing Estimates” on page 21-136
- “Create and Use Code Generation Reports” on page 23-2

Distributed Pipelining Using Synthesis Timing Estimates

Distributed pipelining is a speed optimization that increases clock frequency for your target device by distributing pipelines in your generated HDL code. For more information, see “Distributed Pipelining” on page 21-130. Synthesis timing estimates can more accurately reflect how components function on hardware. You can use them for distributed pipelining to more effectively maximize the clock frequency for your specific target device.

Synthesis Timing Estimates in Distributed Pipelining

When applying distributed pipelining to a subsystem, HDL Coder calculates an approximate propagation delay for each component in your design to determine where to place the added delays. Synthesis timing estimates are propagation delay estimates for each component in your design.

Without synthesis timing estimates, HDL Coder calculates propagation delay for the components in your design by assigning each component an equal weight, except for wire components, such as Selector blocks and Bit Concat blocks, that are assigned zero propagation delay.

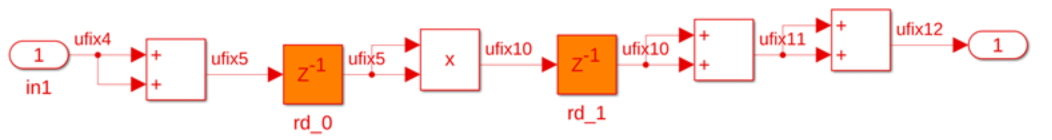
Using synthesis timing estimates, HDL Coder calculates propagation delays for the components in your design by assigning different weights to varying components, based on how the components function on hardware. This information is derived from a target-specific timing database that HDL Coder supports, or that you create by using the `genhdltdb` function. Critical path estimation uses the same timing databases to estimate the critical path in a design. See “Critical Path Estimation Without Running Synthesis” on page 21-192. Using synthesis timing estimates for distributed pipelining is typically applied when you specify a synthesis tool for the model because the optimization option is platform-specific.

How Distributed Pipelining Works By Using Synthesis Timing Estimates

For an example of how distributed pipelining works using by synthesis timing estimates, in the following model, there is a delay of 2 at the output. Multipliers on hardware take longer to compute than adders, meaning their actual propagation delay is longer than adder propagation delay. To simplify the calculations for this example, assume adders have a propagation delay of one nanosecond and multipliers have twice as long of a propagation delay as adders, two nanoseconds.

Without using synthesis timing estimates for distributed pipelining, the adders and multipliers are given equal weights for their propagation delays and the distributed pipelining algorithm cannot tell the difference between the components. The pipelines are distributed as shown in the figure. The critical path length then becomes the delay of the first adder and multiplier, three nanoseconds.

Using synthesis timing estimates for distributed pipelining, the adders and multiplier are now given weights for their propagation delays that more accurately reflect how the components function on hardware. The delays are redistributed so that the critical path contains just the multiplier, making the critical path length the propagation delay of just the multiplier, two nanoseconds.



Using synthesis timing estimates for distributed pipelining most effectively minimizes the critical path to maximize the clock frequency for your design when:

- Your synthesis tool and target hardware have been characterized by using a timing database that HDL Coder supports or by using a timing database that you create by using the `genhdltdb` function.
- All of the blocks in your design are characterized, meaning the blocks are part of the timing database for each supported target device. See “Characterized Blocks” on page 21-195.

Requirements for Synthesis Timing Estimates for Distributed Pipelining

To use synthesis timing estimates for distributed pipelining, specify distributed pipelining for at least one subsystem or MATLAB Function block in your design.

If you specify a synthesis tool using the model parameter `SynthesisTool`, HDL Coder sets default values for the model parameters `SynthesisToolChipFamily` and `SynthesisToolSpeedValue`. If you do not specify a value for `SynthesisTool` or for `SynthesisToolChipFamily` and `SynthesisToolSpeedValue`, HDL Coder generates a warning when generating HDL code. HDL Coder sets the default values for `SynthesisToolChipFamily` and `SynthesisToolSpeedValue` as `virtex7` and `-1` respectively. To prevent the warning, specify the `SynthesisToolChipFamily` and `SynthesisToolSpeedValue` parameters.

Specify Distributed Pipelining to Use Synthesis Timing Estimates

To use synthesis timing estimates for distributed pipelining, you can:

- In the Configuration Parameters dialog box, enable the parameter **Use synthesis estimates for distributed pipelining**, located in **HDL Code Generation** > **Optimization** > **Pipelining** tab.
- Use the `hdlset_param` function. Set the parameter to `on` at the command line. For example, to enable this parameter for the model `sfir_fixed`, use this command:

  ```
  hdlset_param('sfir_fixed','UseSynthesisEstimatesForDistributedPipelining','on')
  ```

For more information, see the Use Synthesis Timing Estimates for Distributed Pipelining section in the “Distributed Pipelining: Speed Optimization” on page 21-139 example.

Limitations

- The run time to generate HDL code from your model is approximately 10x more than the run time of using equal weights to calculate the propagation delays of the components in your design.

- If there are uncharacterized blocks in the model that are not supported, the distributed pipelining results might be less optimal than not using synthesis timing estimates for distributed pipelining. The uncharacterized blocks in the model appear as a message in the HDL Code Generation Check Report and the Resource Utilization Report, and as a link to a script to highlight the blocks in the MATLAB Command Window.
- If you have a MATLAB function block in your design with the HDL Block Property **Architecture** set to `MATLAB Function`, when using synthesis timing estimates for distributed pipelining, you might see the MATLAB function block appear as an uncharacterized block in the reports. To characterize the MATLAB function block, set the **Architecture** to `MATLAB Datapath`.

See Also

Properties
Use synthesis estimates for distributed pipelining | “DistributedPipelining” on page 19-9

Related Examples

- “Distributed Pipelining: Speed Optimization” on page 21-139
- “Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks” on page 27-28
- “Iteratively Maximize Clock Frequency by Using Speed Optimizations” on page 21-167

More About

- “Distributed Pipelining” on page 21-130
- “Critical Path Estimation Without Running Synthesis” on page 21-192

Distributed Pipelining: Speed Optimization

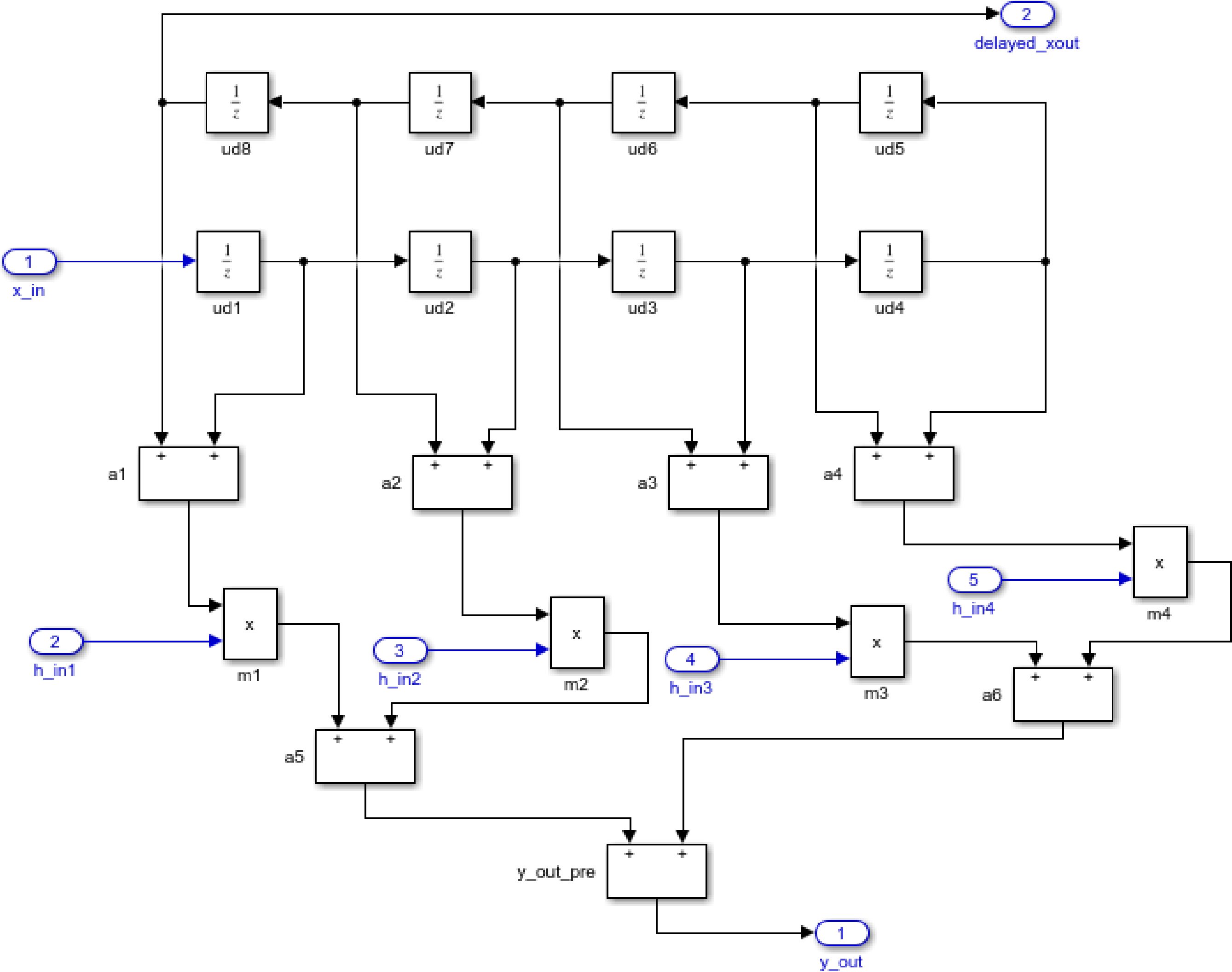
This example shows how to optimize a Simulink® design for speed by using the distributed pipelining optimization.

Introduction

Distributed pipelining is a subsystem-wide optimization supported by HDL Coder for reducing the critical path and achieving high clock speed hardware. By turning on distributed pipelining, HDL Coder redistributes the input pipeline registers, output pipeline registers of the subsystem, and the registers in the subsystem to positions that minimize the combinatorial logic between registers and maximize the clock speed of the chip synthesized from the generated HDL code. For more information on distributed pipelining, see “Distributed Pipelining” on page 21-130.

Consider the following example model of a symmetric FIR filter. The combinatorial logic from an input or a register to an output or another register contains a product block and an adder tree. Distributed pipelining moves the output registers set at the subsystem level to reduce the levels of the combinatorial logic.

```
load_system('sfir_fixed');
open_system('sfir_fixed/symmetric_fir');
```



Set Output Pipeline Stage

To increase the clock speed, you can set a number of pipeline stages for any subsystem. Without turning on distributed pipelining, the specified number of registers are added to each of the output ports of the subsystem. Some synthesis tools support optimizations, such as retiming, that optimize the position of the registers during synthesis.

To see the effects of distributed pipelining in reducing the critical path and increasing the clock frequency, enable `CriticalPathEstimation` to estimate a critical path for your design. When you enable critical path estimation, HDL Coder uses a target-specific timing database to estimate the critical path. If you do not set a Synthesis Tool Chip Family and Synthesis Tool Speed Value to specify the specific target timing database, HDL Coder sets default values for both parameters for critical path estimation, and generates a warning when generating HDL code. To prevent the warning, specify the Synthesis Tool Chip Family and Synthesis Tool Speed Value. You can do this with or without a Synthesis Tool specified. For more information, see “Critical Path Estimation Without Running Synthesis” on page 21-192.

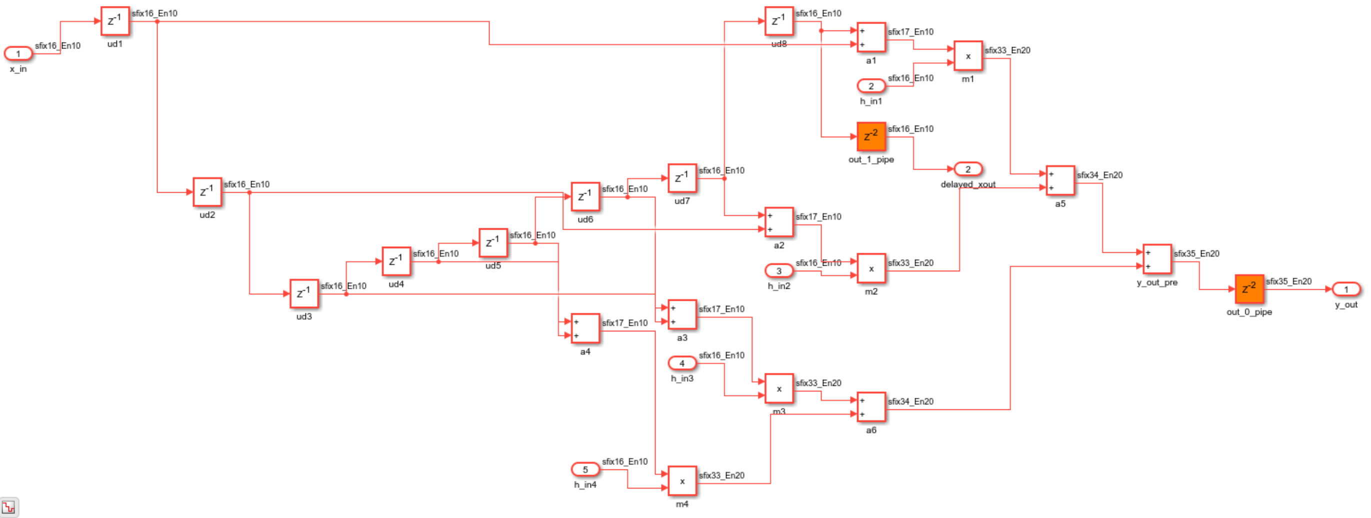
```
hdlset_param('sfir_fixed', 'CriticalPathEstimation', 'on');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'virtex7', 'SynthesisToolSpeedValue', '-1'
```

In this example, the subsystem output pipeline register is set to 2.

The code generation model explicitly reflects the inserted register at output ports of the subsystem (highlighted in orange).

```
hdlset_param('sfir_fixed/symmetric_fir', 'OutputPipeline', 2);
makehdl('sfir_fixed/symmetric_fir');
open_system('gm_sfir_fixed/symmetric_fir');
set_param('gm_sfir_fixed', 'SimulationCommand', 'update');
```

```
### Working on the model <a href="matlab:open_system('sfir_fixed')">sfir_fixed</a>
### Generating HDL for <a href="matlab:open_system('sfir_fixed/symmetric_fir')">sfir_fixed/symmet
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_fixed', {
### Running HDL checks on the model 'sfir_fixed'.
### Begin compilation of the model 'sfir_fixed'...
### Working on the model 'sfir_fixed'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 2 cycles.
### Output port 2: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_fixed', 'GenerateModel')">Genera
### Begin model generation 'gm_sfir_fixed'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\sfir_fixed\gm_sfir_fixed.slx')">
### Estimated critical path for design: <a href="matlab:run('hdlsrc\sfir_fixed\criticalPathEstima
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\sfir_fi
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Generating package file hdlsrc\sfir_fixed\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_fixed' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```



The critical path estimated without distributed pipelining enabled is 10.269 ns. The estimated critical path appears in the **Code Generation Report > Timing and Area Report > Critical Path Estimation** tab.

Set Distributed Pipelining

Enable distributed pipelining for the model by entering:

```
hdlset_param('sfir_fixed', 'DistributedPipelining', 'on');
```

Generate HDL code with a specified generated model name, and open the generated model. The generated model explicitly reflects the distributed registers in the subsystem (highlighted in orange).

```
makehdl('sfir_fixed/symmetric_fir', 'GeneratedModelNamePrefix', 'gm2_');
open_system('gm2_sfir_fixed/symmetric_fir');
set_param('gm2_sfir_fixed', 'SimulationCommand', 'update');
```

```
### Working on the model <a href="matlab:open_system('sfir_fixed')">sfir_fixed</a>
### Generating HDL for <a href="matlab:open_system('sfir_fixed/symmetric_fir')">sfir_fixed/symmet
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_fixed', {
### Running HDL checks on the model 'sfir_fixed'.
### Begin compilation of the model 'sfir_fixed'...
### Working on the model 'sfir_fixed'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 2 cycles.
### Output port 2: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_fixed', 'GenerateModel')">Genera
### Begin model generation 'gm2_sfir_fixed'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\sfir_fixed\gm2_sfir_fixed.slx')
### Estimated critical path for design: <a href="matlab:run('hdlsrc\sfir_fixed\criticalPathEstima
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\sfir_f
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Generating package file hdlsrc\sfir_fixed\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_fixed' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/t
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

The critical path estimated with distributed pipelining on is now 7.443 ns.

Use Synthesis Timing Estimates for Distributed Pipelining

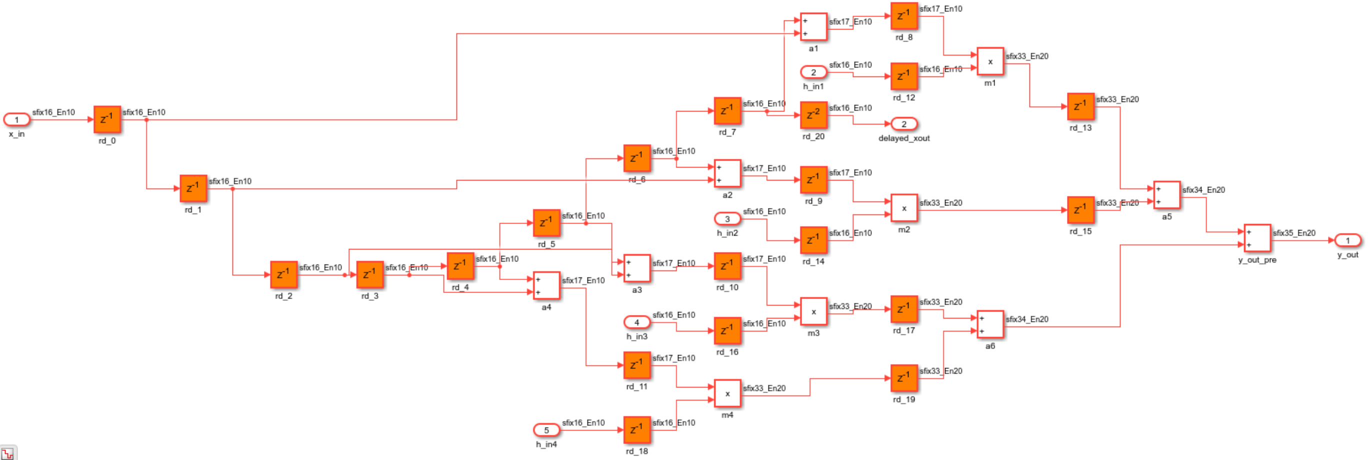
To more accurately reflect how components function on hardware to better distribute pipelines in your design and maximize the clock frequency for your target device, enable synthesis timing estimates for distributed pipelining using the model configuration parameter **Use synthesis estimates for Distributed Pipelining** or through the command-line.

```
hdlset_param('sfir_fixed', 'UseSynthesisEstimatesForDistributedPipelining', 'on');
```

Generate HDL code with a specified generated model name, and open the generated model.

```
makehdl('sfir_fixed/symmetric_fir', 'GeneratedModelNamePrefix', 'gm3_');
open_system('gm3_sfir_fixed/symmetric_fir');
set_param('gm3_sfir_fixed', 'SimulationCommand', 'update');
```

```
### Working on the model <a href="matlab:open_system('sfir_fixed')">sfir_fixed</a>
### Generating HDL for <a href="matlab:open_system('sfir_fixed/symmetric_fir')">sfir_fixed/symmet
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_fixed', {
### Running HDL checks on the model 'sfir_fixed'.
### Begin compilation of the model 'sfir_fixed'...
### Working on the model 'sfir_fixed'...
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 2 cycles.
### Output port 2: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_fixed', 'GenerateModel')">Generat
### Begin model generation 'gm3_sfir_fixed'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\sfir_fixed\gm3_sfir_fixed.slx')"
### Estimated critical path for design: <a href="matlab:run('hdlsrc\sfir_fixed\criticalPathEstima
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\sfir_fi
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Generating package file hdlsrc\sfir_fixed\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_fixed' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```



Using synthesis timing estimates causes an increase in run time of the function `makehdl`, but it can reduce the critical path and increase the clock frequency of your design. For more information, see "Distributed Pipelining Using Synthesis Timing Estimates" on page 21-136.

The critical path estimation report shows that critical path is now 6.320 ns. The critical path estimation is just an estimated critical path based on a target-specific timing database. If you want the actual critical path on your target hardware, you must run the model through synthesis.

Synthesis Comparison of Distributed Pipelining With and Without Synthesis Timing Estimates

In the HDL Workflow Advisor, generate HDL code and perform FPGA synthesis using the Generic ASIC/FPGA workflow with these settings:

- **Synthesis tool** set to `Xilinx Vivado`
- **Family** of the synthesis tool set to `virtex7`
- **Speed** of the synthesis tool set to `-1`
- **Target Frequency (MHz)** set to `120`

For more information on the code generation and synthesis steps, see "HDL Code Generation and FPGA Synthesis from Simulink Model".

The synthesis results without distributed pipelining enabled are:

Timing summary	
	Value
Requirement	8.3333 ns
Data Path Delay	7.368 ns
Slack	-0.764 ns
Clock Frequency	

The synthesis results shows negative slack, indicating that timing constraints are not met. The clock frequency is not calculated when timing constraints are not met.

The synthesis results with distributed pipelining enabled are:

Timing summary	
	Value
Requirement	8.3333 ns
Data Path Delay	1.35 ns
Slack	2.717 ns
Clock Frequency	178.05 MHz

The synthesis results show positive slack and a clock frequency of 178.05 MHz, indicating that the timing constraints are met and the target clock frequency of 120 MHz is met.

The synthesis results with distributed pipelining using synthesis timing estimates are:

Timing summary	
	Value
Requirement	8.3333 ns
Data Path Delay	1.273 ns
Slack	3.837 ns
Clock Frequency	222.40 MHz

The synthesis results show a clock frequency of 222.40 MHz, indicating that the timing constraints are met. The clock speed has increased by using synthesis timing estimates for distributed pipelining.

Distributed Pipelining Across Subsystem Hierarchies

If your DUT subsystem has subsystem hierarchy, meaning it contains lower-level subsystems, set the model parameter **Distributed pipelining** to `on` to have distributed pipelining run through the entire DUT without needing to enable `DistributedPipelining` for the DUT and each subsystem inside the DUT. The subsystem HDL block property **DistributedPipelining** is set to `inherit` by default, which means that each subsystem takes the **DistributedPipelining** value of its parent subsystem. The top-level DUT subsystem takes the value specified by the model parameter **Distributed pipelining**.

You can specifically enable or disable distributed pipelining for a lower-level subsystem if you set the HDL block property **DistributedPipelining** to `On` or `Off` for the subsystem. For more information, “DistributedPipelining” on page 19-9.

See Also

Related Examples

- “Iteratively Maximize Clock Frequency by Using Speed Optimizations” on page 21-167

More About

- “Distributed Pipelining” on page 21-130

Constrained Output Pipelining

In this section...

"What Is Constrained Output Pipelining?" on page 21-146
"When to Use Constrained Output Pipelining" on page 21-146
"Requirements for Constrained Output Pipelining" on page 21-146
"Specify Constrained Output Pipelining" on page 21-146
"Limitations of Constrained Output Pipelining" on page 21-147

What Is Constrained Output Pipelining?

With constrained output pipelining, you can specify a nonnegative number of registers at the outputs of a block.

Constrained output pipelining does not add registers, but instead redistributes existing delays within your design to try to meet the constraint. If HDL Coder cannot meet the constraint with existing delays, it reports the difference between the number of desired and actual output registers in the timing report.

Distributed pipelining does not move registers you specify with constrained output pipelining.

When to Use Constrained Output Pipelining

Use constrained output pipelining when you want to place registers at specific locations in your design. This can enable you to optimize the speed of your design.

For example, if you know where the critical path is in your design and want to reduce it, you can use constrained output pipelining to place registers at specific locations along the critical path.

Requirements for Constrained Output Pipelining

Your design must contain existing delays or registers. When there are fewer registers than HDL Coder needs to satisfy your constraint, the coder reports the difference between the number of desired and actual output registers.

You can add registers to your design using input or output pipelining.

Specify Constrained Output Pipelining

To specify constrained output pipelining for a block using the UI:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the Subsystem and then click **HDL Block Properties**. For **ConstrainedOutputPipeline**, enter the number of registers you want at the output ports.
- Right-click the block and select **HDL Code > HDL Block Properties**. For **ConstrainedOutputPipeline**, enter the number of registers you want at the output ports.

To specify constrained output pipelining, on the command line, enter:

```
hdlset_param(path_to_block,...
        'ConstrainedOutputPipeline',number_of_output_registers)
```

For example, to constrain six registers at the output ports of a subsystem, *subsys*, in your model, *mymodel*, enter:

```
hdlset_param('mymodel/subsys','ConstrainedOutputPipeline', 6)
```

Limitations of Constrained Output Pipelining

HDL Coder does not constrain output pipeline register placement:

- Within a DUT subsystem, if the DUT contains a subsystem, model reference, or model reference with black box implementation.
- At the outputs of any type of delay block or the top-level DUT subsystem.

Clock-Rate Pipelining

In this section...

"Rationale for Clock-Rate Pipelining" on page 21-148
"How Clock-Rate Pipelining Works in a Simulink Model" on page 21-148
"How Clock-Rate Pipelining Works in a MATLAB Function" on page 21-149
"Clock-Rate Pipelining and Hierarchy Flattening" on page 21-149
"Clock-Rate Pipelining for DUT Output Ports" on page 21-150
"Specify Clock-Rate Pipelining" on page 21-150
"Clock-Rate Pipelining Report" on page 21-151
"Limitations for Clock-Rate Pipelining" on page 21-151

Clock-rate pipelining is an optimization framework in HDL Coder that allows other speed and area optimizations to introduce latency at the clock rate. When modeling at a rate slower than the clock rate, clock-rate pipelining automatically upsamples these slower regions such that the latency introduced by other optimizations is either minimized or offset by data rate delay. Clock-rate pipelining does not attempt to optimize a design without other speed and area optimization requests. You can use clock-rate pipelining with a Simulink model or a MATLAB function.

Rationale for Clock-Rate Pipelining

HDL Coder introduces pipelines when you specify certain block implementations or enable some optimizations on the Simulink model or MATLAB function, such as:

- Multi-cycle block implementations
- Input and output pipelining
- Distributed pipelining (includes setting the HDL block property **ConstrainedOutputPipeline**)
- Adaptive pipelining
- Floating-point library mapping
- Native floating-point HDL code generation
- Resource sharing
- Streaming
- Mapping of lookup table (LUT) to RAM
- Blocks which add latency during code generation automatically (For example, Product, Divide, and Math Function blocks with HDL architecture set to `ShiftAdd`, Sqrt)

By default, in slow paths, these pipeline registers operate at the slow data rate. When you enable clock-rate pipelining, the pipeline registers operate at the faster clock rate. Clock-rate pipelining does not affect existing design delays in your model. It is an alternative to using multicycle path constraints with your synthesis tool.

How Clock-Rate Pipelining Works in a Simulink Model

The clock-rate pipelining optimization identifies slow paths or regions in the model by analyzing the block sample times. Blocks that have a sample time greater than the device under test (DUT) base

sample time are part of the slow path, and are potential candidates for clock-rate pipelining. In these slow paths, HDL Coder enables optimizations to introduce pipeline delays at the clock rate.

If you set the **Oversampling factor** configuration parameter to a value greater than `1` or you enable the **Treat Simulink rates as actual hardware rates** configuration parameter, the DUT sample time becomes slower than the actual clock rate. HDL Coder determines the maximum number of clock-rate pipelines that it can insert based on the DUT-to-block sample time ratio and the oversampling value:

Maximum number of clock-rate delays = (*block_rate* ÷ *DUT_base_rate*) × `Oversampling`

If you are modeling with actual hardware rates, enable the **Treat Simulink rates as actual hardware rates** parameter to set an oversampling value for your model automatically. If you are modeling with relative rates in Simulink, set the **Oversampling factor** parameter to a value greater than `1` to set an oversampling value for your model. When you enable the **Treat Simulink rates as actual hardware rates** parameter, HDL Coder can automatically adjust the oversampling value for your model if changes in Simulink rate or target frequency occur, without requiring you to update the **Oversampling factor** manually.

Clock-rate pipelining identifies regions in the model that have the same slow data rate and are delimited by either Delay blocks or blocks that introduce a rate transition. HDL Coder converts these regions to the faster clock rate by introducing Repeat blocks at the input of the region and Rate Transition blocks at the output of the region. If the output of a clock-rate region is a Delay block at the data rate, HDL Coder absorbs that Delay block. To accommodate the delay, HDL Coder introduces several clock-rate pipelines that correspond to the ratio of the data rate to the clock rate.

How Clock-Rate Pipelining Works in a MATLAB Function

A MATLAB function, which is the DUT when generating HDL code from MATLAB, runs at a single data rate. If you want a clock rate that is faster than the data rate, you can use clock-rate pipelining. You specify how much faster you want your clock rate to be than the DUT data rate by specifying an oversampling factor greater than one. The maximum number of clock-rate delays that HDL Coder can insert is equal to the oversampling factor.

You can also use clock-rate pipelining to optimize speed if you have other code generation options that introduce latency in a feedback loop, such as the use of persistent variables or native floating point.

Clock-Rate Pipelining and Hierarchy Flattening

You can use the clock-rate pipelining optimization with or without flattening the subsystem hierarchy. Flatten the subsystem hierarchy when you want to maximize opportunities for sharing resources in your design. To flatten the subsystem hierarchy, enable **FlattenHierarchy** on the top-level Subsystem. By default, all Subsystem blocks inside the top-level subsystem inherit this **FlattenHierarchy** setting. Hierarchy flattening brings several clock-rate regions to the same level in the hierarchy and combines them, which increases opportunities for clock-rate pipelining. However, it breaks the modularity of your design and affects the readability of the generated HDL code. See also “Hierarchy Flattening” on page 21-117.

To apply clock-rate pipelining without flattening the hierarchy, on the top-level subsystem in your model, disable **FlattenHierarchy**. If your design uses fixed-point data types, enable some optimizations on the underlying subsystems. In this case, HDL Coder introduces clock-rate pipelines in your design while preserving the subsystem hierarchy, which:

- Improves the modularity of your design and makes navigation through the generated model easier especially in large designs with complex hierarchies.
- Improves readability of the generated HDL code by creating multiple Verilog, SystemVerilog or VHDL files for the various Subsystem blocks in your design.

Clock-Rate Pipelining for DUT Output Ports

To produce DUT outputs as soon as possible by passing the outputs from the DUT at the clock rate rather than the data rate, select the **Allow clock-rate pipelining of DUT output ports** configuration parameter or use the `ClockRatePipelineOutputPorts` property. This property changes the timing of your DUT interface by changing the sample time of the DUT output ports from a slower data rate to the clock rate. To adjust for the difference in timing, HDL Coder generates messages that provide the phase offset of each output port. For example, this message means that the output data from *portname* is valid after 31 clock cycles: `Phase of output port` *`portname`*`: 31 clock cycles.` To set this parameter, see Allow clock-rate pipelining of DUT output ports.

The validation model adjusts for the timing difference by inserting a Rate Transition block at the DUT output and comparing the output of the Rate Transition block with the original output. The RTL test bench logs the output data at the input of the Rate Transition block and compares it with the DUT output in the RTL simulation.

When producing DUT outputs at the clock rate, the outputs are ready as soon as possible, even if one output is ready before another. For example, this is useful if you run a hardware-in-the-loop simulation with a control system and have a plant model running at a slow rate and discrete logic such as control signals in the DUT that need to run at a faster rate or with a short response delay time. To synchronize the outputs while still satisfying the highest-latency requirements of the outputs, you can balance clock-rate pipelined DUT output ports by selecting the **Balance clock-rate pipelined DUT output ports** configuration parameter or using the `BalanceClockRateOutputPorts` property. You can apply this property when interfacing your logic with a valid signal interface to align the output of your logic path and the output of the valid signal path. To set this parameter, see Balance clock-rate pipelined DUT output ports.

Specify Clock-Rate Pipelining

You can generate HDL code with the clock-rate pipelining optimization from a Simulink model or MATLAB function.

Specify Clock-Rate Pipelining for a Simulink Model

You can set clock-rate pipelining on a model or, for finer control, on subsystems within the top-level DUT subsystem. By default, clock-rate pipelining is enabled on the model. To disable clock-rate pipelining from the UI:

1. In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
2. Click **Settings**. In the **HDL Code Generation** > **Optimization** > **Pipelining** tab, clear **Clock-rate pipelining** and click **OK**.

At the command line, use the `makehdl` or `hdlset_param` function to set the `ClockRatePipelining` property to `off`.

You can use clock-rate pipelining for a subsystem within the top-level DUT subsystem. To model a control path in your design at the data rate instead of the clock rate, put the control path in a

subsystem, and disable clock-rate pipelining for that subsystem. To disable clock-rate pipelining for a subsystem within the top-level DUT subsystem, set **ClockRatePipelining** to `off` for that subsystem. See also "Set Clock-Rate Pipelining For a Subsystem" on page 19-6.

Specify Clock-Rate Pipelining for a MATLAB Function

To enable the clock-rate pipelining for a MATLAB function:

1. Open the MATLAB HDL Workflow Advisor. To get started with the MATLAB HDL Workflow Advisor, see "Basic HDL Code Generation and FPGA Synthesis from MATLAB".
2. In the left pane, click the **HDL Code Generation** task. In the right pane, navigate to the **Optimization** tab and select **Clock Rate Pipelining**.
3. Click the **Clocks & Ports** tab and set **Oversampling factor** to a value greater than one.

Clock-Rate Pipelining Report

To see the clock-rate pipelining information in the report, before you generate code for each subsystem or model reference, enable the optimization report. In the**HDL Code Generation > Report** section of the Configuration Parameters dialog box, select **Generate optimization report**.

When you generate the optimization report, you can use the **Clock-Rate Pipelining** section of the report to see how the clock-rate pipelining optimization performed in your model. The **Clock-Rate Pipelining** section includes:

- Whether clock-rate pipelining and its associated optimizations ran in the model successfully
- The oversampling factor used in the model
- The clock rate used for clock-rate pipelining
- A list of the clock-rate pipelining obstacles and a link that highlights the obstacles in the original and generated models
- The suggested oversampling factor, if possible
- Latency budgets for loops, if possible

For more information on how to create and use the code generation report, see "Create and Use Code Generation Reports" on page 23-2.

Limitations for Clock-Rate Pipelining

These blocks do not support clock-rate pipelining, and therefore delimit clock-rate pipelining regions:

- Deserializer1D
- Serializer1D
- Dual Port RAM
- Dual Rate Dual Port RAM
- Simple Dual Port RAM
- Single Port RAM
- HDL FIFO
- HDL Cosimulation

- Hit Crossing
- MATLAB System, if it uses persistent variables
- Reusable Subsystem (CodeReuseSubsystem), if `FlattenHierarchy` is not enabled
- Model. Use a Subsystem Reference instead.

HDL Coder does not support clock-rate pipelining for:

- Subsystem or Model block with `HDLArchitecture` set to `BlackBox`.

  Black box subsystem or black box model reference blocks.
- Subsystems that contain blocks not supported for clock-rate pipelining.
- DSP Builder for Intel FPGAs subsystems.
- System Generator for DSP subsystems.
- Communications Toolbox blocks.
- DSP System Toolbox blocks, except for Delay and Discrete FIR Filter.
- Wireless HDL Toolbox blocks.
- Vision HDL Toolbox blocks.
- Stateflow blocks.

HDL Coder does not support applying both the streaming and sharing optimizations on the same resource when you use the clock-rate pipelining optimization. Either disable clock-rate pipelining or use either the streaming optimization or the sharing optimization on the same resource when clock-rate pipelining is enabled.

HDL Coder does not support clock-rate pipelining when **Clock inputs** is set to `Multiple`.

See Also

Model Settings
Clock-rate pipelining | Allow clock-rate pipelining of DUT output ports | Balance clock-rate pipelined DUT output ports

Related Examples

- "Increase Clock Frequency Using Clock-Rate Pipelining" on page 21-153
- "Optimize Feedback Loop Design and Maintain High Data Precision for HDL Code Generation" on page 8-26

More About

- "ClockRatePipelining" on page 19-6
- "Delay Balancing" on page 21-81
- "Hierarchy Flattening" on page 21-117
- "Create and Use Code Generation Reports" on page 23-2

Increase Clock Frequency Using Clock-Rate Pipelining

This example shows how to apply clock-rate pipelining to optimize slow paths in your design and thereby reduce latency, increase clock frequency and decrease area usage. For more information on how to use clock-rate pipelining, see “Clock-Rate Pipelining” on page 21-148.

Introduction

Algorithmic design with Simulink can introduce slow-rate datapaths in the generated HDL design. These paths correspond to slower Simulink sample time operations because the algorithmic data rate operates at a slower rate than the HDL clock rate.

Clock-rate pipelining identifies the maximal subregions in the model that operate at the same data rate and that are delimited either by rate-change blocks or Delay blocks. These subregions are called *clock-rate regions* because they are good candidates for clock-rate pipelining. If the output of a clock-rate region is a Delay block at the data rate, HDL Coder absorbs that Delay block, which results in a budget of several clock-rate pipelines equal to the ratio of data rate to the clock rate.

Consider the field-oriented control example “Field-Oriented Control of a Permanent Magnet Synchronous Machine” on page 14-66, which describes a motor-control design mapped to an FPGA. The input samples arrive every 20 μs or 50 KHz. In a closed control loop, the controller latency must be within the desired response time. There is a delay on the output port that results in a latency of 20 μs.

To meet design constraints like timing and area, you can apply several optimizations like input and output pipelining, distributed pipelining, streaming, and/or sharing. Additionally, you can implement non-trivial math functions like `sqrt` or `divide` as multi-cycle pipelined operations. Because the pipelines introduced by these optimizations are applied at the same rate at which the signal path operates, which is 20 μs, introducing additional pipelining creates undesirable latency overhead and can violate the closed loop latency budget.

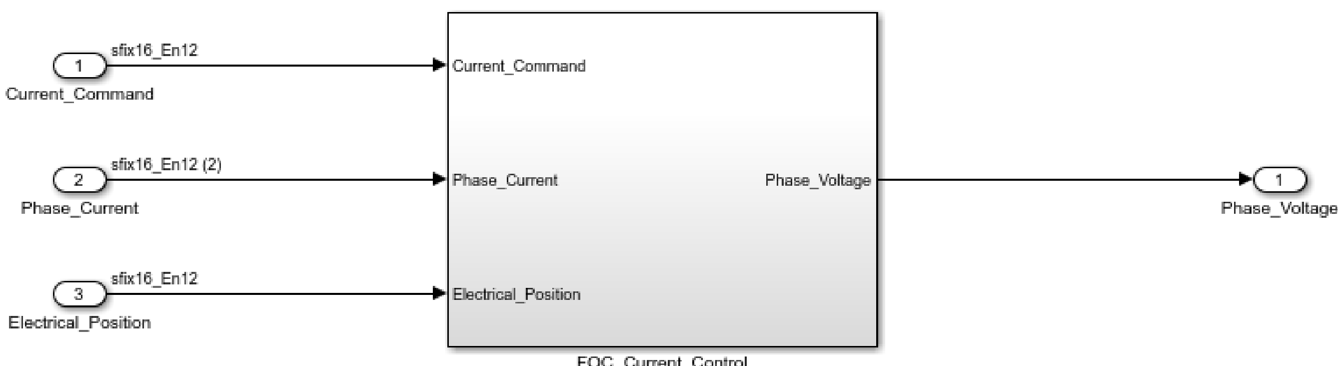
However, the FPGA can implement this controller in the order of MHz, which means that the introduced pipelines can operate at the MHz rate and minimize the impact on latency. Clock-rate pipelining leverages this rate differential and pipelines the controller to improve its area and timing characteristics on the FPGA. This example shows how to incrementally apply timing and area optimizations using clock-rate pipelining.

Prepare the Model

Prepare the model so that it is ready for clock-rate pipelining.

Open the `hdlcoderFocCurrentFixptHdl` model, then define the rate differential. Signal paths in Simulink end up on slow paths in HDL either because the signal path operates at a slower sample time than the base sample time of the model, or because the Simulink base sample time corresponds to the data rate instead of the clock rate. The base sample time in this model is 20 μ secs. The final FPGA implementation of the controller targets 40 MHz, or 25 ns.

```
open_system('hdlcoderFocCurrentFixptHdl')
```



Setting the model sample time to 25 ns slows down Simulink simulation performance. HDL Coder can automatically determine how much faster the FPGA clock rate runs with respect to the Simulink base sample time when you select the configuration parameter Treat Simulink rates as actual hardware rates. In this case, HDL Coder calculates the oversampling value needed for your design by dividing the target frequency of the FPGA 40MHZ by the model base frequency 50KHz.

Set optimizations on subsystems. For fixed-point designs, HDL Coder applies clock-rate pipelining to subsystems only when it needs to insert pipelining. The HDL Coder options that introduce pipelining are distributed pipelining, sharing, streaming, input/output pipelining, constrained output pipeline, adaptive pipelining, and any block implementations that introduce multi-cycle implementations including floating point implementations. You can apply optimizations either locally, to maintain the subsystem hierarchies, or globally, if the underlying subsystems are flattened. In the local case, apply pipelining and optimization settings on individual subsystems. To apply global optimizations, set them on the top-level subsystem. For more information on prerequisites for hierarchy flattening, see “Hierarchy Flattening” on page 21-117.

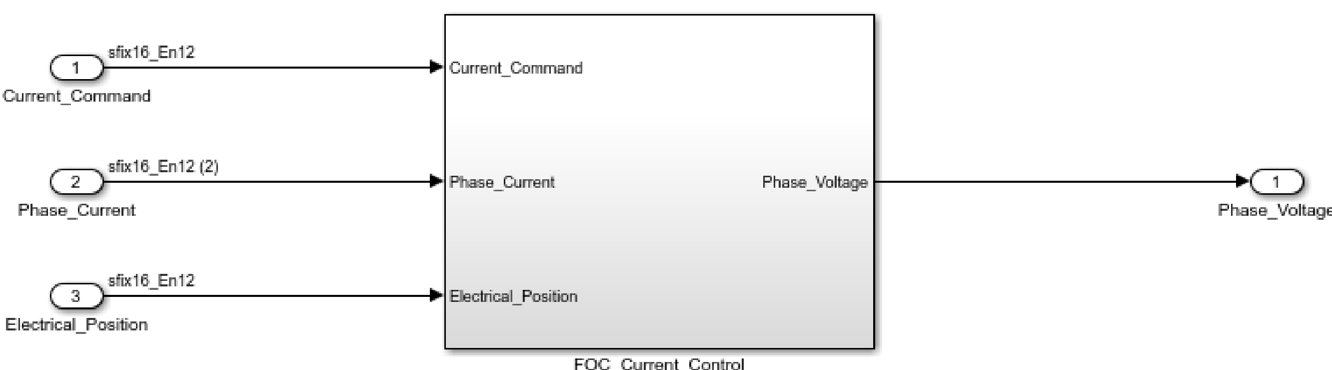
Apply Clock-Rate Pipelining

Create a copy of the original model, `hdlcoderFocCurrentFixptHdl`, and save it as `hdlcoderFocClockRatePipelining`. In this example, you apply clock-rate pipelining optimizations to the new model and compare it to the original model.

```
srcHdlModel = 'hdlcoderFocCurrentFixptHdl';
dstHdlModel = 'hdlcoderFocClockRatePipelining';
dstHdlDut   = [dstHdlModel '/FOC_Current_Control'];
gmHdlModel  = ['gm_' dstHdlModel];
gmHdlDut    = ['gm_' dstHdlDut];

open_system(srcHdlModel);
save_system(srcHdlModel,dstHdlModel);
```

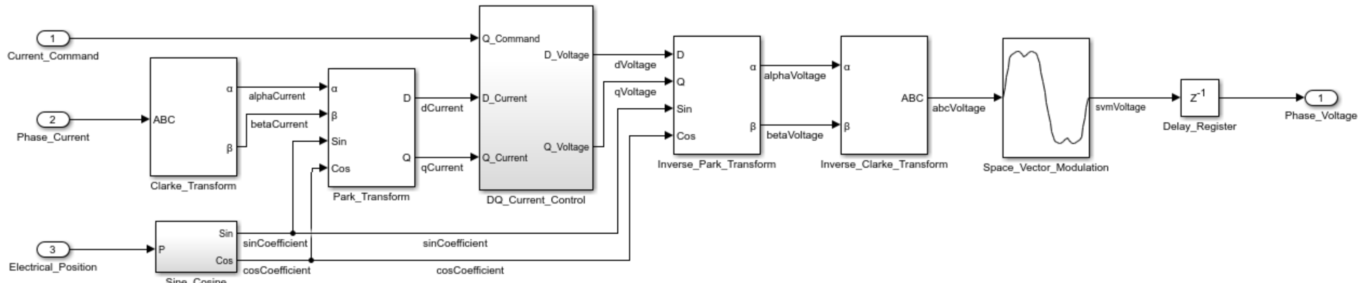
FOC Current Control

Copyright 2014-2017 The MathWorks, Inc.



The subsystem `FOC_Current_Control` that contains the algorithm for which HDL code is generated is the device under test (DUT). Open the DUT subsystem.

```
open_system(dstHdlDut);
```



Apply clock-rate pipelining. Clock-rate pipelining is enabled by default and automatically finds clock-rate regions when you generate HDL code. For more information, see “Clock-Rate Pipelining” on page 21-148.

```
hdlset_param(dstHdlModel, 'ClockRatePipelining', 'on');
```

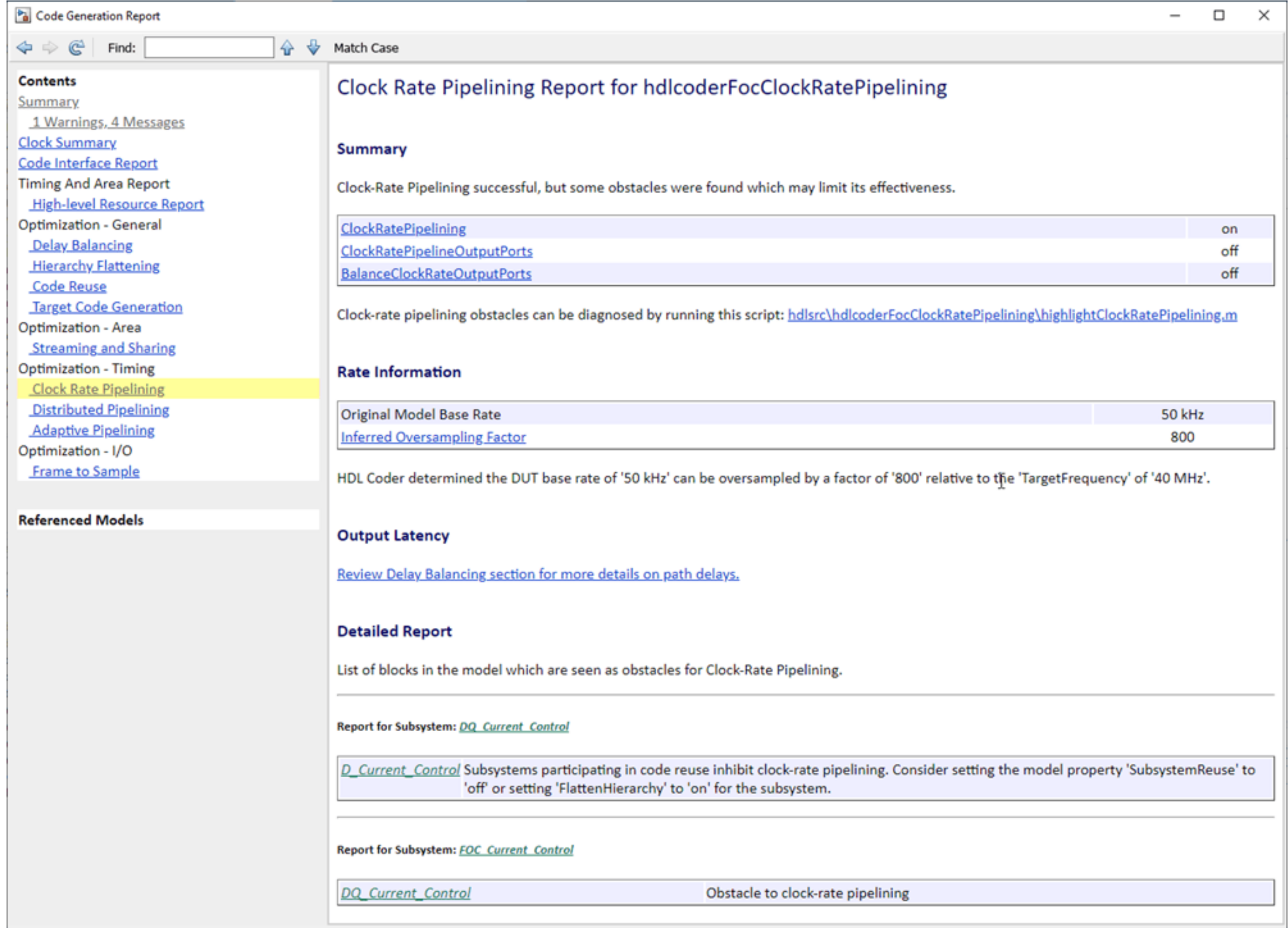
Configure the model to increase clock speed by setting `TargetFrequency` to `40` MHz for the model, enabling `TreatRatesAsHardwareRates` to set an oversampling value for clock-rate pipelining, and enabling distributed pipelining for the model. The subsystems inherit the top-level distributed pipelining setting by default.

```
hdlset_param(dstHdlModel, 'TargetFrequency', 40);
hdlset_param(dstHdlModel, 'TreatRatesAsHardwareRates', 'on');
hdlset_param(dstHdlModel, 'DistributedPipelining', 'on');

save_system(dstHdlModel);
```

To see the impact of clock-rate pipelining, generate HDL code and open the Code Generation report to the **Clock-Rate Pipelining** section.

```
makehdl(dstHdlDut);
```

```
### Working on the model <a href="matlab:open_system('hdlcoderFocClockRatePipelining')">hdlcoderF
### Generating HDL for <a href="matlab:open_system('hdlcoderFocClockRatePipelining/FOC_Current_Co
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocClock
### Running HDL checks on the model 'hdlcoderFocClockRatePipelining'.
### Begin compilation of the model 'hdlcoderFocClockRatePipelining'...
### Begin compilation of the model 'hdlcoderFocClockRatePipelining'...
### Working on the model 'hdlcoderFocClockRatePipelining'...
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocClockRatePipelining', 'Gene
### Begin model generation 'gm_hdlcoderFocClockRatePipelining'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoderFocClockRatePipelining\g
### Clock-rate pipelining obstacles can be diagnosed by running this script: <a href="matlab:run
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoderFocClockRatePip
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocClockRatePipelining'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Begin VHDL Code Generation for 'FOC_Current_Control_tc'.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocClockRatePipelining\FOC_Current_Contro
### Code Generation for 'FOC_Current_Control_tc' completed.
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/DQ_Current_Control/D_Current_Co
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/DQ_Current_Control/D_Current_Co
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/DQ_Current_Control as hdlsrc\hd
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Clarke_Transform as hdlsrc\hdlc
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Sine_Cosine/Sine_Cosine_LUT as
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Sine_Cosine as hdlsrc\hdlcoderF
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Park_Transform as hdlsrc\hdlcod
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Inverse_Park_Transform as hdlsr
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Inverse_Clarke_Transform as hdl
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Space_Vector_Modulation as hdls
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control as hdlsrc\hdlcoderFocClockRateP
### Generating package file hdlsrc\hdlcoderFocClockRatePipelining\FOC_Current_Control_pkg.vhd.
### Code Generation for 'hdlcoderFocClockRatePipelining' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlcoderFocClockRatePipelining' complete with 0 errors, 2 warnings, and 4 mess
### HDL code generation complete.
```



In the **Clock-Rate Pipelining** section, under **Detailed Report**, there is a list of clock-rate pipelining obstacles. To remove the obstacles, you can apply the solutions listed in the report. For this example, disable the block parameter `TreatAsAtomicUnit` for both subsystems that are obstacles to treat the blocks in the subsystem as being at the same level in the model hierarchy as the subsystem. Clock-rate pipelining can then run across these subsystems and optimize your model.

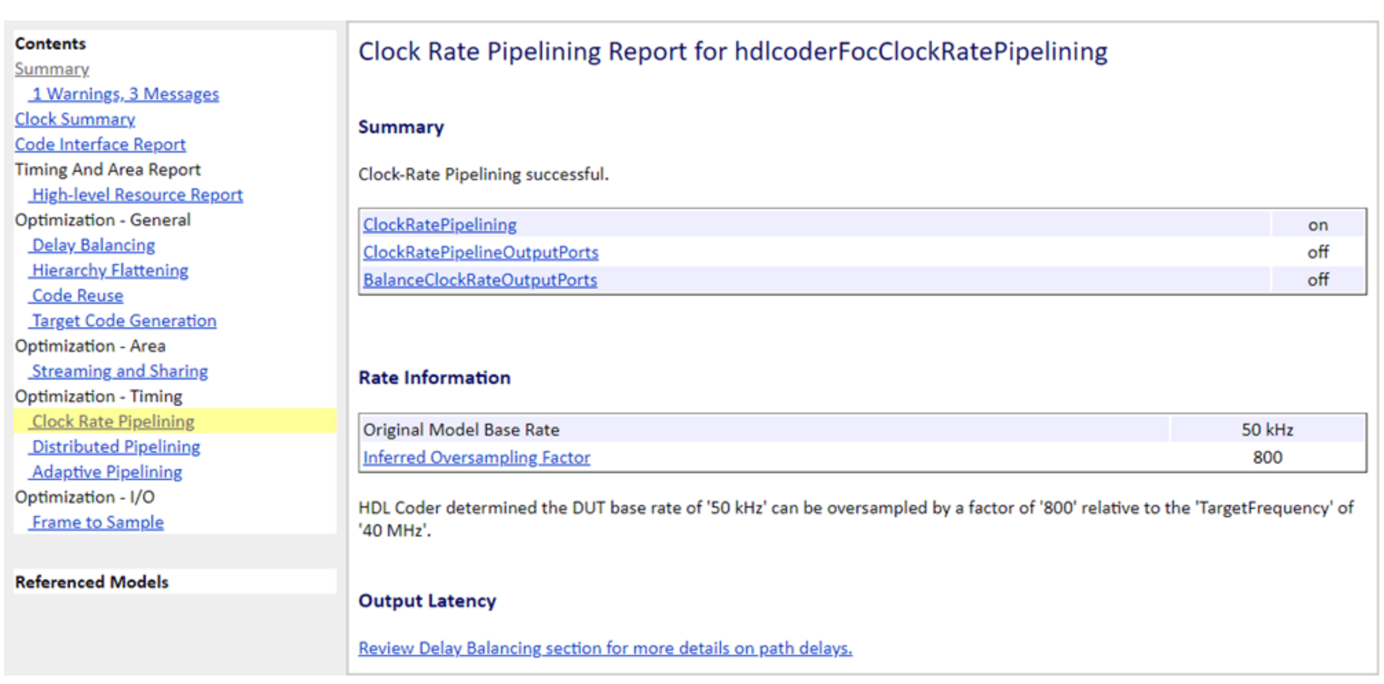
```
set_param([dstHdlDut '/DQ_Current_Control/D_Current_Control'], 'TreatAsAtomicUnit', 'off');
set_param([dstHdlDut '/DQ_Current_Control/Q_Current_Control'], 'TreatAsAtomicUnit', 'off');
```

Generate HDL code and check the **Clock-Rate Pipelining** section of the Code Generation report to see that there are no longer clock-rate pipelining obstacles. For more information on the clock-rate pipelining section of the Code Generation report, see “Clock-Rate Pipelining Report” on page 21-151.

```
makehdl(dstHdlDut);
```

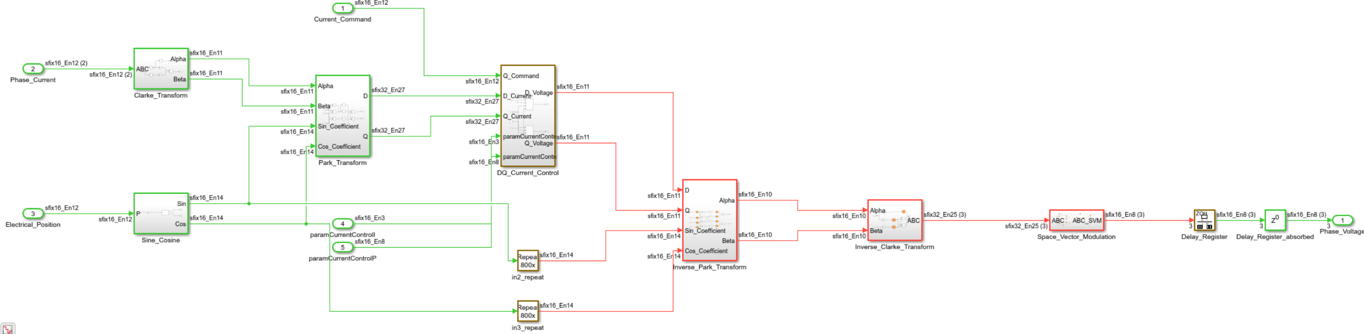
```
### Working on the model <a href="matlab:open_system('hdlcoderFocClockRatePipelining')">hdlcoderF
### Generating HDL for <a href="matlab:open_system('hdlcoderFocClockRatePipelining/FOC_Current_Co
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocClock
### Running HDL checks on the model 'hdlcoderFocClockRatePipelining'.
### Begin compilation of the model 'hdlcoderFocClockRatePipelining'...
### Begin compilation of the model 'hdlcoderFocClockRatePipelining'...
### Working on the model 'hdlcoderFocClockRatePipelining'...
```

```
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocClockRatePipelining', 'Gene
### Begin model generation 'gm_hdlcoderFocClockRatePipelining'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoderFocClockRatePipelining\g
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoderFocClockRatePip
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocClockRatePipelining'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Begin VHDL Code Generation for 'FOC_Current_Control_tc'.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocClockRatePipelining\FOC_Current_Contro
### Code Generation for 'FOC_Current_Control_tc' completed.
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/DQ_Current_Control/D_Current_Co
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/DQ_Current_Control/D_Current_Co
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/DQ_Current_Control/Q_Current_Co
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/DQ_Current_Control/Q_Current_Co
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/DQ_Current_Control as hdlsrc\hd
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Clarke_Transform as hdlsrc\hdlc
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Sine_Cosine/Sine_Cosine_LUT as
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Sine_Cosine as hdlsrc\hdlcoderF
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Park_Transform as hdlsrc\hdlcod
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Inverse_Park_Transform as hdlsr
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Inverse_Clarke_Transform as hdl
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control/Space_Vector_Modulation as hdls
### Working on hdlcoderFocClockRatePipelining/FOC_Current_Control as hdlsrc\hdlcoderFocClockRateP
### Generating package file hdlsrc\hdlcoderFocClockRatePipelining\FOC_Current_Control_pkg.vhd.
### Code Generation for 'hdlcoderFocClockRatePipelining' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlcoderFocClockRatePipelining' complete with 0 errors, 2 warnings, and 3 mess
### HDL code generation complete.
```



Look inside the top-level subsystem of the generated model. The generated model shows that the design has been clock-rate pipelined and is running at the fast rate. If there are subsystems in the generated model that are not clock-rate pipelined, then check if there were optimizations set on the subsystem in the original model.

```
open_system(gmHdlDut);
set_param(gmHdlModel, 'SimulationCommand', 'update');
set_param(gmHdlDut, 'ZoomFactor', 'FitSystem');
```



Rate transitions are introduced on the design inputs to bring them to the clock rate, which is set by the target frequency, 1/40 MHz, or 25 ns. All pipelines are introduced at this rate and are operating at the clock rate. Finally, the output-side delay has been replaced by a down-sampling rate transition, which brings the signal back to the data rate. Compared to the original model, the pipelines inserted at the clock-rate improved the clock frequency of the new design, without incurring any additional sample time delays.

Verify that the functional behavior of the design is unchanged by using the validation model and co-simulation model. For more information, see “Verification”.

Apply Local Area Optimizations to Subsystems

The rate differential on the slow path implies that computation along this path can take several clock cycles. Specifically, the allowed latency is defined by the clock-rate budget. Apart from adding pipelines to improve clock frequency, you can reuse hardware resources by leveraging the latency budget. Apply resource sharing optimizations, such as setting the `StreamingFactor` and `SharingFactor` options in a slow-path region. This section demonstrates how resource sharing is applied within clock-rate regions.

When resource sharing is applied to a clock-rate path, HDL Coder oversamples the shared resource architecture for time-multiplexing as illustrated in “Resource Sharing for Area Optimization” on page 21-54. However, if sharing or streaming is requested in a slow datapath, then HDL Coder implements resource sharing without oversampling.

The `Park_Transform` subsystem and `Inverse_Park_Transform` subsystem each use four multipliers within them that can be potentially shared. Additionally, the `Clarke_Transform` subsystem and `Inverse_Clarke_Transform` subsystem each use two gains, which may be potentially shared, unless they are simply power-of-2 gains, which results in shifts instead of multiplications. As a result, the gain in `Inverse_Clarke_Transform` cannot be shared. Open the current model and highlight the subsystems where sharing can be applied.

```
srcHdlModel = 'hdlcoderFocClockRatePipelining';
dstHdlModel = 'hdlcoderFocSharing';
```

```
dstHdlDut   = [dstHdlModel '/FOC_Current_Control'];
gmHdlModel  = ['gm_' dstHdlModel];
gmHdlDut    = ['gm_' dstHdlDut];

open_system(srcHdlModel);
save_system(srcHdlModel,dstHdlModel);

open_system(dstHdlDut);
hilite_system([dstHdlDut '/Park_Transform']);
hilite_system([dstHdlDut '/Inverse_Park_Transform']);
hilite_system([dstHdlDut '/Clarke_Transform']);
```

Set the sharing factors to the number of multipliers to share for each of the subsystems in order to apply resource sharing.

```
hdlset_param([dstHdlDut '/Park_Transform'], 'SharingFactor', 4);
hdlset_param([dstHdlDut '/Inverse_Park_Transform'], 'SharingFactor', 4);
hdlset_param([dstHdlDut '/Clarke_Transform'], 'SharingFactor', 2);

save_system(dstHdlModel);
```

Generate HDL code using the `makehdl` command.

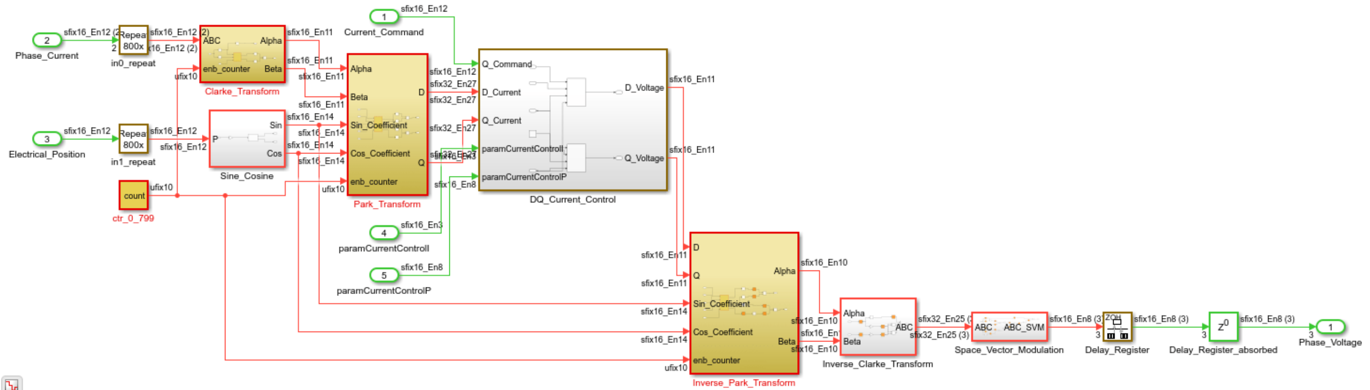
```
makehdl(dstHdlDut);
```

```
### Working on the model <a href="matlab:open_system('hdlcoderFocSharing')">hdlcoderFocSharing</a
### Generating HDL for <a href="matlab:open_system('hdlcoderFocSharing/FOC_Current_Control')">hdl
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocShar
### Running HDL checks on the model 'hdlcoderFocSharing'.
### Begin compilation of the model 'hdlcoderFocSharing'...
### Begin compilation of the model 'hdlcoderFocSharing'...
### Working on the model 'hdlcoderFocSharing'...
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocSharing', 'GenerateModel')"
### Begin model generation 'gm_hdlcoderFocSharing'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoderFocSharing\gm_hdlcoderFo
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoderFocSharing\gm_h
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocSharing'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Begin VHDL Code Generation for 'FOC_Current_Control_tc'.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocSharing\FOC_Current_Control_tc.vhd.
### Code Generation for 'FOC_Current_Control_tc' completed.
```

```
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Satura
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control/D_Current_Control as hdl
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control/Q_Current_Control/Satura
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control/Q_Current_Control as hdl
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control as hdlsrc\hdlcoderFocSha
### Working on hdlcoderFocSharing/FOC_Current_Control/Clarke_Transform as hdlsrc\hdlcoderFocShari
### Working on hdlcoderFocSharing/FOC_Current_Control/Sine_Cosine/Sine_Cosine_LUT as hdlsrc\hdlco
### Working on hdlcoderFocSharing/FOC_Current_Control/Sine_Cosine as hdlsrc\hdlcoderFocSharing\Si
### Working on hdlcoderFocSharing/FOC_Current_Control/Park_Transform as hdlsrc\hdlcoderFocSharing
### Working on hdlcoderFocSharing/FOC_Current_Control/Inverse_Park_Transform as hdlsrc\hdlcoderFo
### Working on hdlcoderFocSharing/FOC_Current_Control/Inverse_Clarke_Transform as hdlsrc\hdlcode
### Working on hdlcoderFocSharing/FOC_Current_Control/Space_Vector_Modulation as hdlsrc\hdlcoder
### Working on hdlcoderFocSharing/FOC_Current_Control as hdlsrc\hdlcoderFocSharing\FOC_Current_Co
### Generating package file hdlsrc\hdlcoderFocSharing\FOC_Current_Control_pkg.vhd.
### Code Generation for 'hdlcoderFocSharing' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlcoderFocSharing' complete with 0 errors, 2 warnings, and 3 messages.
### HDL code generation complete.
```

Review the generated model and observe that HDL Coder implements time-multiplexing at the clock rate using knowledge of the available latency budget due to the slow datapath.

```
load_system(gmHdlDut);
set_param(gmHdlModel, 'SimulationCommand', 'update');
set_param(gmHdlDut, 'ZoomFactor', 'FitSystem');
hilite_system([gmHdlDut '/ctr_0_799']);
hilite_system([gmHdlDut '/Clarke_Transform/Clarke_Transform_shared']);
hilite_system([gmHdlDut '/Park_Transform/Park_Transform_shared']);
hilite_system([gmHdlDut '/Inverse_Park_Transform/Inverse_Park_Transform_shared']);
open_system(gmHdlDut);
Simulink.BlockDiagram.arrangeSystem(gmHdlDut);
```



The time-multiplexing architecture, also known as the single-rate sharing architecture is described in “Single-Rate Resource Sharing Architecture” on page 21-65. A global scheduler is created to enable and disable different regions of the design using enabled subsystems. The enable/disable control is implemented using a Limited Counter block `ctr_0_799` that counts to the latency budget (0 to 799). The shared regions are implemented as enabled subsystems that are enabled according to an automatically determined schedule order. In this design, there are two subsystems containing groups of multipliers that are shared in four places and one subsystem containing a group of multipliers that are shared in two places. As a result of resource sharing, the multiplier count for the design has reduced from 20 to 13 without any latency penalties.

Apply Global Optimizations by Flattening the Model

Global cross-subsystem optimizations can be applied by leveraging the subsystem-flattening feature. With flattening, there are more number of resources that can be shared at the same level of hierarchy. To trigger such sharing, set either sharing or streaming on the top-level subsystem. The sharing factor value chosen must be an upper bound. To determine a good value, the resource usage of the design must be analyzed.

```
srcHdlModel = 'hdlcoderFocCurrentFixptHdl';
dstHdlModel = 'hdlcoderFocSharingWithFlattening';
dstHdlDut   = [dstHdlModel '/FOC_Current_Control'];
gmHdlModel  = ['gm_' dstHdlModel];
gmHdlDut    = ['gm_' dstHdlDut];

open_system(srcHdlModel);
save_system(srcHdlModel,dstHdlModel);

open_system(dstHdlDut);

hdlset_param(dstHdlModel, 'ClockRatePipelining', 'on');
hdlset_param(dstHdlModel, 'TargetFrequency', 40);
hdlset_param(dstHdlModel, 'TreatRatesAsHardwareRates', 'on');
hdlset_param(dstHdlModel, 'DistributedPipelining', 'on');
hdlset_param(dstHdlDut, 'FlattenHierarchy', 'on');

hilite_system([dstHdlDut '/Park_Transform']);
hilite_system([dstHdlDut '/Inverse_Park_Transform']);
hilite_system([dstHdlDut '/Clarke_Transform']);
hilite_system([dstHdlDut '/Inverse_Clarke_Transform']);
```

The `Park_Transform` subsystem and the `Inverse_Park_Transform` subsystem each use four multipliers within them that can be potentially shared. Additionally, the `Clarke_Transform` subsystem and the `Inverse_Clarke_Transform` subsystem each use two gains, which may be potentially shared, unless they are simply power-of-2 gains, which results in shifts instead of multiplications. Choose the upper-bound value of four as the `SharingFactor` and generate HDL code.

```
hdlset_param(dstHdlDut, 'SharingFactor', 4);
save_system(dstHdlModel);

makehdl(dstHdlDut);
```

```
### Working on the model <a href="matlab:open_system('hdlcoderFocSharingWithFlattening')">hdlcode
### Generating HDL for <a href="matlab:open_system('hdlcoderFocSharingWithFlattening/FOC_Current_
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocShar
```

```
### Running HDL checks on the model 'hdlcoderFocSharingWithFlattening'.
### Begin compilation of the model 'hdlcoderFocSharingWithFlattening'...
### Begin compilation of the model 'hdlcoderFocSharingWithFlattening'...
### Working on the model 'hdlcoderFocSharingWithFlattening'...
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocSharingWithFlattening', 'Ge
### Begin model generation 'gm_hdlcoderFocSharingWithFlattening'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoderFocSharingWithFlattening
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoderFocSharingWithF
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocSharingWithFlattening'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Begin VHDL Code Generation for 'FOC_Current_Control_tc'.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocSharingWithFlattening\FOC_Current_Con
### Code Generation for 'FOC_Current_Control_tc' completed.
### Working on hdlcoderFocSharingWithFlattening/FOC_Current_Control as hdlsrc\hdlcoderFocSharingW
### Generating package file hdlsrc\hdlcoderFocSharingWithFlattening\FOC_Current_Control_pkg.vhd.
### Code Generation for 'hdlcoderFocSharingWithFlattening' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlcoderFocSharingWithFlattening' complete with 0 errors, 2 warnings, and 3 me
### HDL code generation complete.
```

Review the generated model and observe that HDL Coder implements time-multiplexing at the clock rate using knowledge of the available latency budget due to the slow datapath.

```
open_system(gmHdlDut);
set_param(gmHdlModel, 'SimulationCommand', 'update');
set_param(gmHdlDut, 'ZoomFactor', 'FitSystem');
hilite_system([gmHdlDut '/ctr_0_799']);
hilite_system([gmHdlDut '/crp_temp_shared']);
hilite_system([gmHdlDut '/crp_temp_shared1']);
hilite_system([gmHdlDut '/crp_temp_shared2']);
hilite_system([gmHdlDut '/crp_temp_shared3']);
hilite_system([gmHdlDut '/crp_temp_shared4']);
```

In this design, there are five subsystems containing groups of multipliers that are shared in four places or less across the model. These five subsystems have `crp_temp_shared` as part of their names.

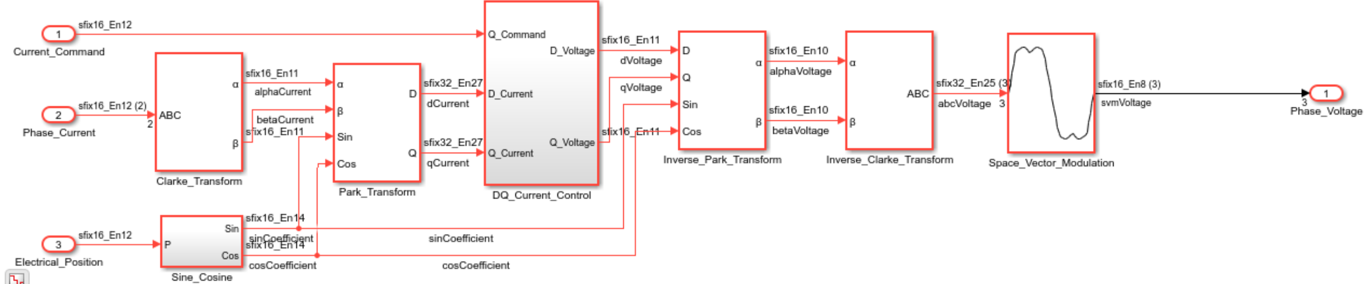
In summary, with the design flattened, the multiplier count for the design has reduced from 20 to 7 without any latency penalties.

Minimize Latency

As an advanced maneuver, it is possible to reduce the output latency by removing the output `Delay_Register` and instead using the option to allow clock-rate pipelining of DUT output ports.

Create a new model as a copy of the `hdlcoderFocSharing` model and remove the output `Delay_Register`.

```
srcHdlModel = 'hdlcoderFocSharing';
dstHdlModel = 'hdlcoderFocMinLatency';
dstHdlDut   = [dstHdlModel '/FOC_Current_Control'];
gmHdlModel  = ['gm_' dstHdlModel];
gmHdlDut    = ['gm_' dstHdlDut];

open_system(srcHdlModel);
save_system(srcHdlModel,dstHdlModel);

delete_line(dstHdlDut,'Space_Vector_Modulation/1','Delay_Register/1');
delete_line(dstHdlDut,'Delay_Register/1','Phase_Voltage/1');
delete_block([dstHdlDut,'/Delay_Register'])
add_line(dstHdlDut,'Space_Vector_Modulation/1','Phase_Voltage/1');

open_system(dstHdlDut);
```



The clock-rate pipelining for output ports option is available in the configuration parameters dialog under the **HDL Code Generation > Optimization > Pipelining** tab: check the **Allow clock-rate pipelining of DUT output ports** option. The command-line property name for this option is `ClockRatePipelineOutputPorts`. When the `ClockRatePipelineOutputPorts` option is turned on and the output register is removed, the generated HDL code does not wait for the full sample step to generate the output. It instead generates the output within a few clock cycles as soon as the data is ready. The generated HDL code generates the output at the clock rate without waiting for the next sample step.

```
hdlset_param(dstHdlModel, 'ClockRatePipelineOutputPorts', 'on');
save_system(dstHdlModel);

makehdl(dstHdlDut);
```

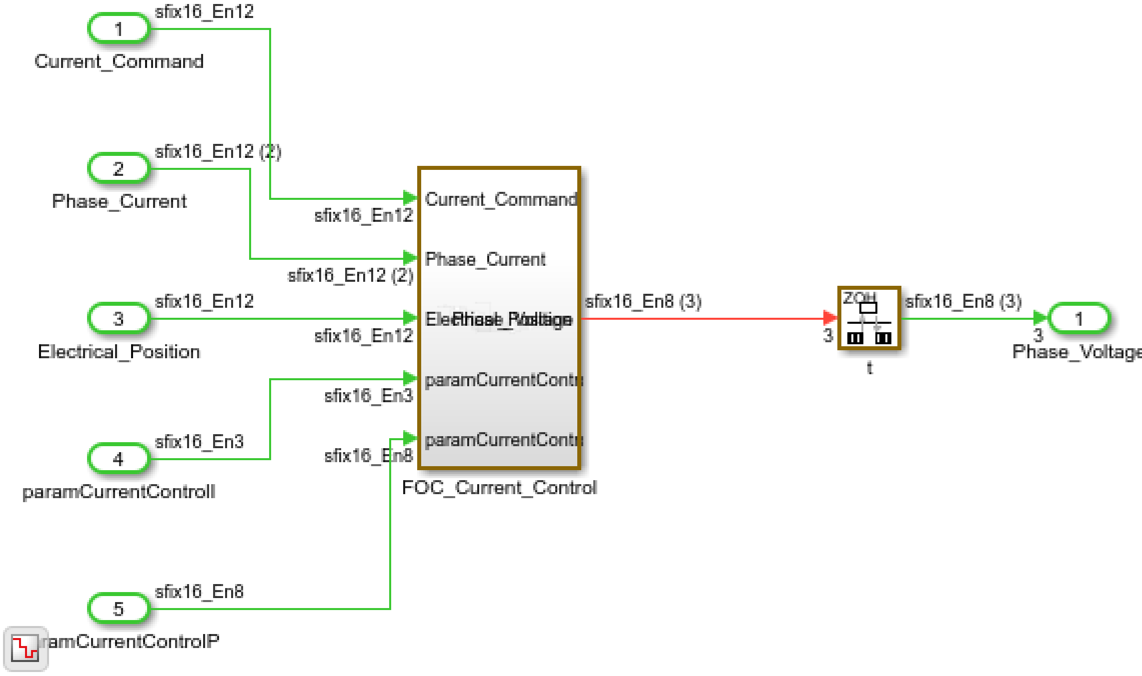
```
### Working on the model <a href="matlab:open_system('hdlcoderFocMinLatency')">hdlcoderFocMinLate
### Generating HDL for <a href="matlab:open_system('hdlcoderFocMinLatency/FOC_Current_Control')">
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocMinLa
### Running HDL checks on the model 'hdlcoderFocMinLatency'.
### Begin compilation of the model 'hdlcoderFocMinLatency'...
### Begin compilation of the model 'hdlcoderFocMinLatency'...
### Working on the model 'hdlcoderFocMinLatency'...
### Clock-rate pipelining was applied on signals connected to the DUT's output ports. The DUT out
### Phase of output port 1: 19 clock cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocMinLatency', 'GenerateMode
### Begin model generation 'gm_hdlcoderFocMinLatency'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
```

```
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoderFocMinLatency\gm_hdlcode
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: '<a href="matlab:open_system('hdlsrc\hdlcoderFocMinLatency\g
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocMinLatency'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Begin VHDL Code Generation for 'FOC_Current_Control_tc'.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control_tc.vhd
### Code Generation for 'FOC_Current_Control_tc' completed.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Clarke_Transform as hdlsrc\hdlcoderFocMi
### Working on hdlcoderFocMinLatency/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Sa
### Working on hdlcoderFocMinLatency/FOC_Current_Control/DQ_Current_Control/D_Current_Control as
### Working on hdlcoderFocMinLatency/FOC_Current_Control/DQ_Current_Control/Q_Current_Control/Sa
### Working on hdlcoderFocMinLatency/FOC_Current_Control/DQ_Current_Control/Q_Current_Control as
### Working on hdlcoderFocMinLatency/FOC_Current_Control/DQ_Current_Control as hdlsrc\hdlcoderFo
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Inverse_Clarke_Transform as hdlsrc\hdlc
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Inverse_Park_Transform as hdlsrc\hdlcod
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Park_Transform as hdlsrc\hdlcoderFocMin
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Sine_Cosine/Sine_Cosine_LUT as hdlsrc\h
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Sine_Cosine as hdlsrc\hdlcoderFocMinLat
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Space_Vector_Modulation as hdlsrc\hdlco
### Working on hdlcoderFocMinLatency/FOC_Current_Control as hdlsrc\hdlcoderFocMinLatency\FOC_Cur
### Generating package file hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control_pkg.vhd.
### Code Generation for 'hdlcoderFocMinLatency' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/t
### HDL check for 'hdlcoderFocMinLatency' complete with 0 errors, 2 warnings, and 3 messages.
### HDL code generation complete.
```

Notice that the `makehdl` command has generated a message, `### Phase of output port 1:`, explaining how to sample the DUT outputs. The number of clock cycles specified in this message is how quickly the DUT outputs can be sampled and, in essence, is the latency of the design. The total latency of the design is down from a data-rate sample step of 20 μs to a few nanoseconds.

Review the generated model to observe that a new DUT subsystem is created whose output operates at the clock rate, which is 25 ns.

```
open_system(gmHdlDut);
set_param(gmHdlModel, 'SimulationCommand', 'update');
set_param(gmHdlDut, 'ZoomFactor', 'FitSystem');
```



This option introduces additional latency into the generated HDL code that was not in the original simulation model. In doing this, the sample time of the output port has changed to the clock rate. This introduces a possible discrepancy in results during the validation and verification flow since the test-harness expects the design to generate outputs at the data rate. The validation model addresses this problem by inserting a down-sampling rate-transition to bring the output back to the data rate. Thus, the validation model still compares outputs at the data rate. The HDL testbench compares the new DUT outputs at the clock rate because the generated HDL outputs are emitted at the clock rate.

Summary

Clock-rate pipelining is a technique to optimize and pipeline slow paths in your design. Clock-rate pipelining ensures that pipelines are introduced at the clock rate for these HDL Coder constructs and optimizations:

- Pipelined math operations: Several math blocks implement a multi-cycle, pipelined HDL implementation, e.g., Newton-Rhapson method for sqrt or recip, Cordic algorithm for trigonometric functions. These pipelines are introduced at clock rate if the block operates on a slow path.
- Floating point mapping: As described above, floating point library mapping utilizes clock-rate pipelines when implementing floating point math.
- Pipelining optimizations: All pipelining optimizations including input/output pipelining, adaptive pipelining and distributed pipelining use clock-rate registers on slow paths.
- Resource sharing and streaming: Time-multiplexing of resource-shared architectures are implemented at the clock rate.

Slow paths are identified as paths using a slower Simulink sample time or when the oversampling value is greater than one. Using clock-rate pipelining, the design speed and area properties can be improved without compromising the total latency of the design.

Iteratively Maximize Clock Frequency by Using Speed Optimizations

This example shows how to iteratively maximize the clock frequency for a model by using various speed optimizations and minimize the resources needed for the design. In this example, you first apply speed optimizations to achieve a desired target frequency, then apply area optimizations to reduce the resources needed for your design on hardware.

The algorithm in this model detects a peak pulse in a noisy signal. The model is designed for deployment onto hardware but it is not yet optimized for speed.

To optimize the model:

1. You use Simulink® HDL optimizations to generate optimized HDL code from the model. These optimizations can increase the clock speed of your design and reduce the amount of hardware components needed for the design.
2. For each iteration of optimizations that you apply, you use the HDL Workflow Advisor to run synthesis on your model and to generate a Resource and Timing summary to determine the clock frequency achieved and the resources used.
3. From the summary, you can determine which optimizations to apply next. The target clock frequency in this example is 230 MHz.

Prerequisites

This example requires Xilinx Vivado® as the synthesis tool. Set up the synthesis tool before beginning this example by using the function `hdlsetuptoolpath`.

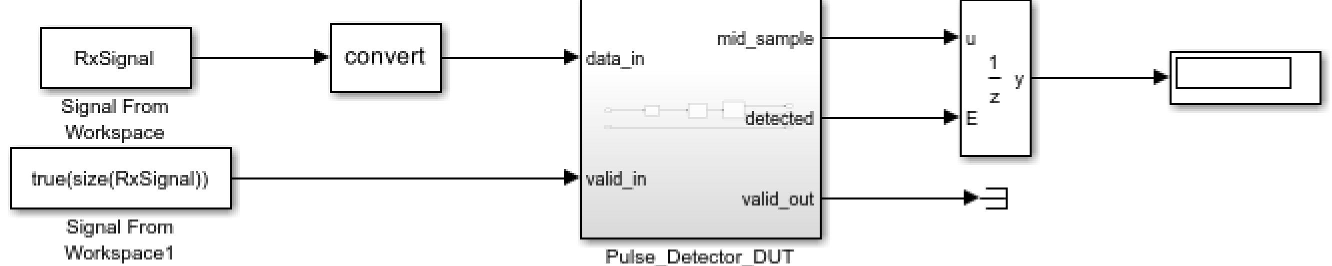
This example targets the Xilinx Zynq® hardware with a speed grade of -1. HDL Coder™ has a default timing database for Xilinx Zynq with a speed grade -1. If you are targeting an FPGA that HDL Coder does not have a characterized timing database for, use the function `genhdltdb` to create your own database. To see the list of characterized timing databases, see “Critical Path Estimation Without Running Synthesis” on page 21-192.

If you generate your own timing database, you can use the model parameter `TimingDatabaseDirectory` to set the timing database for the model to your custom timing database created. See Custom Timing Database Directory.

Open and Simulate the Model

Open and simulate the model to see the peak found using the peak detection algorithm.

```
modelname = 'pulse_detector';
DUTname = [modelname '/Pulse_Detector_DUT'];
open_system(modelname);
```




```
sim(modelname);
```

Set the `Pulse_Detector_DUT` as the HDL subsystem so you generate HDL code for the `Pulse_Detector_DUT` subsystem and not the entire model.

```
hdlset_param(modelname,'HDLSubsystem', DUTname);
```

Apply Base Speed Optimizations

When you want to deploy a model to an FPGA, it is a best practice to apply some base speed optimizations. For this example, enable adaptive pipelining and add I/O pipeline registers. Applying an input and output pipeline enables you to clock the I/O for the FPGA. Adaptive pipelining enables you to pipeline and target DSPs. For more information, see “Adaptive Pipelining” on page 21-181.

```
hdlset_param(modelname, 'AdaptivePipelining', 'on');
hdlset_param(DUTname, 'OutputPipeline', 1);
hdlset_param(DUTname, 'InputPipeline', 1);
```

When you apply optimizations that can affect timing or numerical changes in the model, such as adaptive pipelining, it is a best practice to generate a validation model. Enable the generation of the validation model. The validation model compares the generated model and original model and displays any numerical mismatches between the two.

```
hdlset_param(modelname, 'GenerateValidationModel', 'on');
```

To view the nondefault HDL code generation parameters for the model, use the function `hdlsaveparams`.

```
hdlsaveparams(modelname)
```

```
%% Set Model 'pulse_detector' HDL parameters
hdlset_param('pulse_detector', 'AdaptivePipelining', 'on');
hdlset_param('pulse_detector', 'Backannotation', 'on');
hdlset_param('pulse_detector', 'CriticalPathEstimation', 'on');
hdlset_param('pulse_detector', 'GenerateHDLTestBench', 'off');
hdlset_param('pulse_detector', 'GenerateValidationModel', 'on');
hdlset_param('pulse_detector', 'HDLSubsystem', 'pulse_detector/Pulse_Detector_DUT');
hdlset_param('pulse_detector', 'MinimizeClockEnables', 'on');
hdlset_param('pulse_detector', 'ResetType', 'Synchronous');
hdlset_param('pulse_detector', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('pulse_detector', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('pulse_detector', 'SynthesisToolDeviceName', 'xc7z045');
hdlset_param('pulse_detector', 'SynthesisToolPackageName', 'ffg900');
```

```
hdlset_param('pulse_detector', 'SynthesisToolSpeedValue', '-1');
hdlset_param('pulse_detector', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('pulse_detector', 'TargetFrequency', 230);
hdlset_param('pulse_detector', 'TargetLanguage', 'Verilog');
hdlset_param('pulse_detector', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'InputPipeline', 1);
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'OutputPipeline', 1);

hdlset_param('pulse_detector/Pulse_Detector_DUT/Discrete FIR Filter', 'Architecture', 'Fully Para
```

Note that the Discrete FIR Filter block has the HDL block property `Architecture` set to `Fully Parallel`. This option enables subsystem level optimizations to be applied to the filter block. For more information, see “Subsystem Optimizations for Filters” on page 21-214.

Generate code by using the `makehdl` command.

```
makehdl(DUTname)
```

```
### Working on the model <a href="matlab:open_system('pulse_detector')">pulse_detector</a>
### Generating HDL for <a href="matlab:open_system('pulse_detector/Pulse_Detector_DUT')">pulse_de
### Using the config set for model <a href="matlab:configset.showParameterGroup('pulse_detector'
### Running HDL checks on the model 'pulse_detector'.
### Begin compilation of the model 'pulse_detector'...
### Working on the model 'pulse_detector'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 9 cycles.
### Output port 2: 9 cycles.
### Output port 3: 9 cycles.
### Working on... <a href="matlab:configset.internal.open('pulse_detector', 'GenerateModel')">Ge
### Begin model generation 'gm_pulse_detector'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\pulse_detector\gm_pulse_
### Estimated critical path for design: <a href="matlab:run('hdl_prj\hdlsrc\pulse_detector\criti
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdl_
### Blocks that are not characterized for Critical Path Estimation: <a href="matlab:run('hdl_prj
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsr
### Generating new validation model: '<a href="matlab:open_system('hdl_prj\hdlsrc\pulse_detector
### Validation model generation complete.
### Begin Verilog Code Generation for 'pulse_detector'.
### Unused logic removed during HDL code generation. To highlight the logic removed, click the fo
### To clear highlighting, click the following MATLAB script: hdl_prj\hdlsrc\pulse_detector\clea
### Working on... <a href="matlab:configset.internal.open('pulse_detector', 'Traceability')">Tra
### Working on pulse_detector/Pulse_Detector_DUT/Compute Power as hdl_prj\hdlsrc\pulse_detector\
### Working on pulse_detector/Pulse_Detector_DUT/Local Peak/MATLAB Function as hdl_prj\hdlsrc\pu
### Working on pulse_detector/Pulse_Detector_DUT/Local Peak as hdl_prj\hdlsrc\pulse_detector\Loc
### Working on pulse_detector/Pulse_Detector_DUT as hdl_prj\hdlsrc\pulse_detector\Pulse_Detector
### Code Generation for 'pulse_detector' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/t
### HDL check for 'pulse_detector' complete with 0 errors, 0 warnings, and 4 messages.
### HDL code generation complete.
```

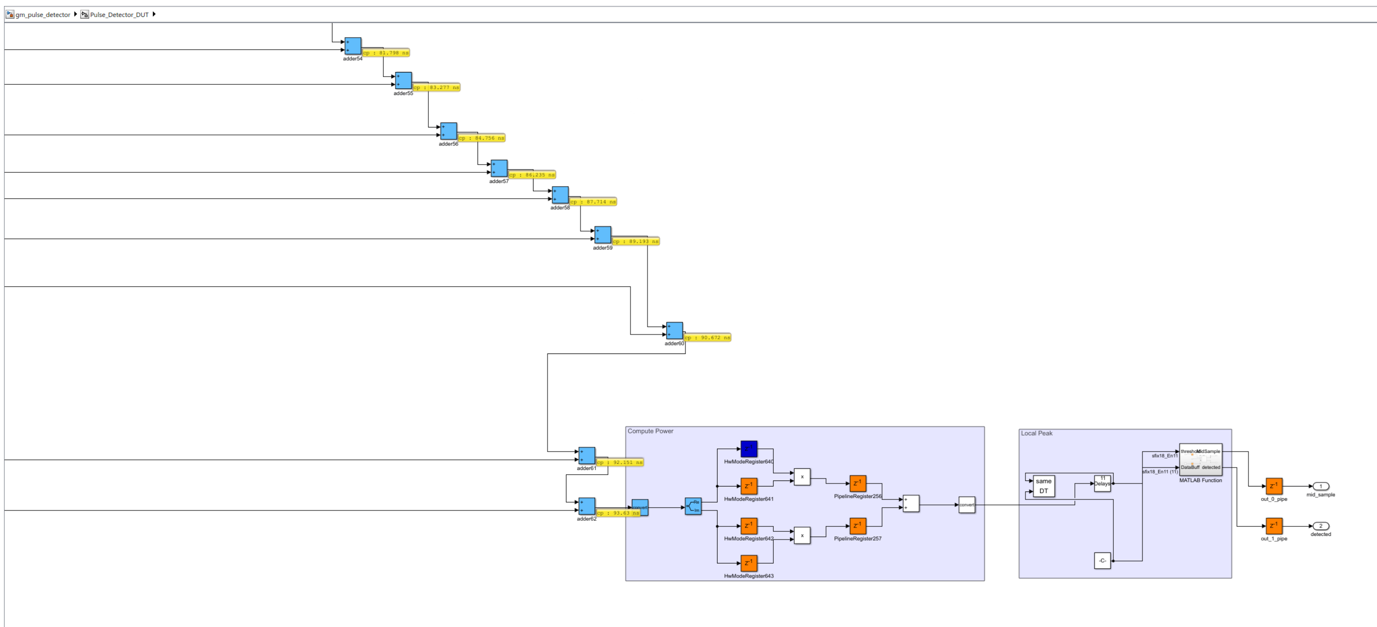
Because the `Traceability` parameter for this model is `on`, the Code Generation Report window opens after the `makehdl` function completes. This report summarizes the generated code and the HDL code options applied to the model. For more information, see “Create and Use Code Generation Reports” on page 23-2.

In the left pane, click **Delay Balancing**. The pipelines added from the base speed optimizations applied add 9 cycles of latency to the generated model.

You can find out more information about the optimizations applied in the **Optimization Report** section. Each link in this section characterizes an optimization applied to the model. Click **High-level resource report** to see the initial estimate of resources needed to deploy the model to your target hardware.

Open the generated model `gm_pulse_detector` to see the optimizations applied to the model. The generated model is located in the `hdl_prj\hdlsrc\pulse_detector` folder.

Click on the highlighting script `hdl_prj\hdlsrc\pulse_detector\criticalPathEstimated.m` in the output of the `makehdl` command to highlight the estimated critical path of the design. This image shows a snippet of the generated model with part of estimated critical path highlighted and the subsystems expanded. The critical path estimation shows the critical path running through a long chain of adders from the Discrete FIR Filter block.



You can also see the estimated critical path in the Code Generation Report by clicking **Critical Path Estimation**. For this example, the critical path estimated is 93.872 ns. The critical path estimation is an estimation of the critical path of your model when it is implemented on hardware. To accurately capture the critical path, you must run synthesis on your model.

In the **Critical Path Estimation** pane, you can also click scripts to highlight the estimated critical path of your design and the blocks not characterized by the critical path estimation in both the original model and generated model. Click the script that highlights the uncharacterized blocks, `highlightCriticalPathEstimationOffendingBlocks.m`. The script highlights the MATLAB Function block. When the HDL block property **Architecture** is set to `MATLAB Function`, critical

path estimation cannot characterize the MATLAB Function block. While you can still generate code, to most accurately capture the critical path of your design and apply subsystem level optimizations to your MATLAB Function block, consider changing the **Architecture** of your MATLAB Function block to `MATLAB Datapath`. This option is explored in the next set of optimizations applied to the model.

To determine if the speed optimizations applied to the model achieve the target frequency, run synthesis on the model to check if the timing constraints are met.

To run synthesis on the model:

1. In Simulink, the in **HDL Code** tab, click **Workflow Advisor**.
2. In the left pane, click **4. FPGA Synthesis and Analysis > 4.2 Perform Synthesis and P/R > 4.2.2. Run Implementation**. Clear **Skip This Task**.
3. Right-click **4.2.2 Run Implementation** and click **Run to Selected Task**.

The HDL Workflow Advisor runs through synthesis and implementation for the model. The Result section displays the Resource and Timing summary from synthesis. The Timing summary shows that there is negative slack, which indicates that the timing constraint, the target clock frequency, is not met. When timing is not met, the clock frequency is not calculated in this report.

Parsed resource report file: Pulse_Detector_DUT_utilization_placed.rpt.

Resource summary

Resource	Usage	Available	Utilization (%)
Slice LUTs	4241	218600	1.94
Slice Registers	6349	437200	1.45
DSPs	258	900	28.67
Block RAM Tile	0	545	0.00
URAM	0	0	

Parsed timing report file: timing_post_route.rpt.

Timing summary

	Value
Requirement	4.3478 ns
Data Path Delay	8.615 ns
Slack	-4.731 ns
Clock Frequency	

For reference, you can calculate the clock frequency manually by using this equation:

$$ClockFrequency = 1/(Requirement - Slack)$$

The calculated clock frequency is 110 MHz.

For more information on the code generation and synthesis steps, see “HDL Code Generation and FPGA Synthesis from Simulink Model”.

Apply Distributed Pipelining

To apply Simulink HDL optimizations to the MATLAB Function block, set the HDL block property **Architecture** to `MATLAB Datapath`. This option characterizes the MATLAB Function block for

critical path estimation, which helps you get a more accurate estimate of your critical path. For more information on when to set this property to MATLAB Datapath, see “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 21-205.

```
hdlset_param([DUTname '/Local Peak/MATLAB Function'], 'Architecture', 'MATLAB Datapath');
```

To increase the clock frequency and reduce the critical path, you can enable the HDL optimization distributed pipelining. In order to use distributed pipelining, there already needs to be pipelines in the design that distributed pipelining can move to minimize the critical path. In this example, to break up the long chain of adders causing a long critical path, increase the number of pipeline stages for distributed pipelining to move by increasing the number of output pipelines. For more information on distributed pipelining, see “Distributed Pipelining” on page 21-130.

Increase the number of output pipeline stages in the DUT to 23 and enable distributed pipelining for the model.

```
hdlset_param(DUTname, 'OutputPipeline', 23);
hdlset_param(modelname, 'DistributedPipelining', 'on');
```

Pipeline distribution priority is by default set as Numerical Integrity. Set the **Pipeline distribution priority** to Performance to allow distributed pipelining to move pipelines with the goal of trying to maximize clock speed.

It is important to note that when you set **Pipeline distribution priority** to Performance, the model behavior might not strictly match the generated code behavior. If strict numerical integrity is a requirement, keep **Pipeline distribution priority** set to Numerical Integrity. Instead, consider moving the placement of initial pipelines set in the design, in this case the output pipelines stages, to locations where the pipelines are not blocked by distributed pipelining barriers.

```
hdlset_param(modelname, 'PipelineDistributionPriority', 'Performance');
```

To apply distributed pipelining across all subsystems in the DUT and see all blocks at the same level in the generated model, enable the FlattenHierarchy parameter on the DUT subsystem. Alternatively, to keep the subsystem hierarchy while still applying distributed pipelining across subsystems, enable **Distributed pipelining** for the model.

```
hdlset_param(DUTname, 'FlattenHierarchy', 'on')
```

Report the nondefault HDL parameters for the model to see the new applied optimizations for the model.

```
hdlsaveparams(modelname)
```

```
%% Set Model 'pulse_detector' HDL parameters
hdlset_param('pulse_detector', 'AdaptivePipelining', 'on');
hdlset_param('pulse_detector', 'Backannotation', 'on');
hdlset_param('pulse_detector', 'CriticalPathEstimation', 'on');
hdlset_param('pulse_detector', 'DistributedPipelining', 'on');
hdlset_param('pulse_detector', 'GenerateHDLTestBench', 'off');
hdlset_param('pulse_detector', 'GenerateValidationModel', 'on');
hdlset_param('pulse_detector', 'HDLSubsystem', 'pulse_detector/Pulse_Detector_DUT');
hdlset_param('pulse_detector', 'MinimizeClockEnables', 'on');
hdlset_param('pulse_detector', 'PipelineDistributionPriority', 'Performance');
hdlset_param('pulse_detector', 'ResetType', 'Synchronous');
hdlset_param('pulse_detector', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('pulse_detector', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('pulse_detector', 'SynthesisToolDeviceName', 'xc7z045');
```

```
hdlset_param('pulse_detector', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('pulse_detector', 'SynthesisToolSpeedValue', '-1');
hdlset_param('pulse_detector', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('pulse_detector', 'TargetFrequency', 230);
hdlset_param('pulse_detector', 'TargetLanguage', 'Verilog');
hdlset_param('pulse_detector', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'FlattenHierarchy', 'on');
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'InputPipeline', 1);
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'OutputPipeline', 23);

hdlset_param('pulse_detector/Pulse_Detector_DUT/Discrete FIR Filter', 'Architecture', 'Fully Para

hdlset_param('pulse_detector/Pulse_Detector_DUT/Local Peak/MATLAB Function', 'Architecture', 'MAT
```

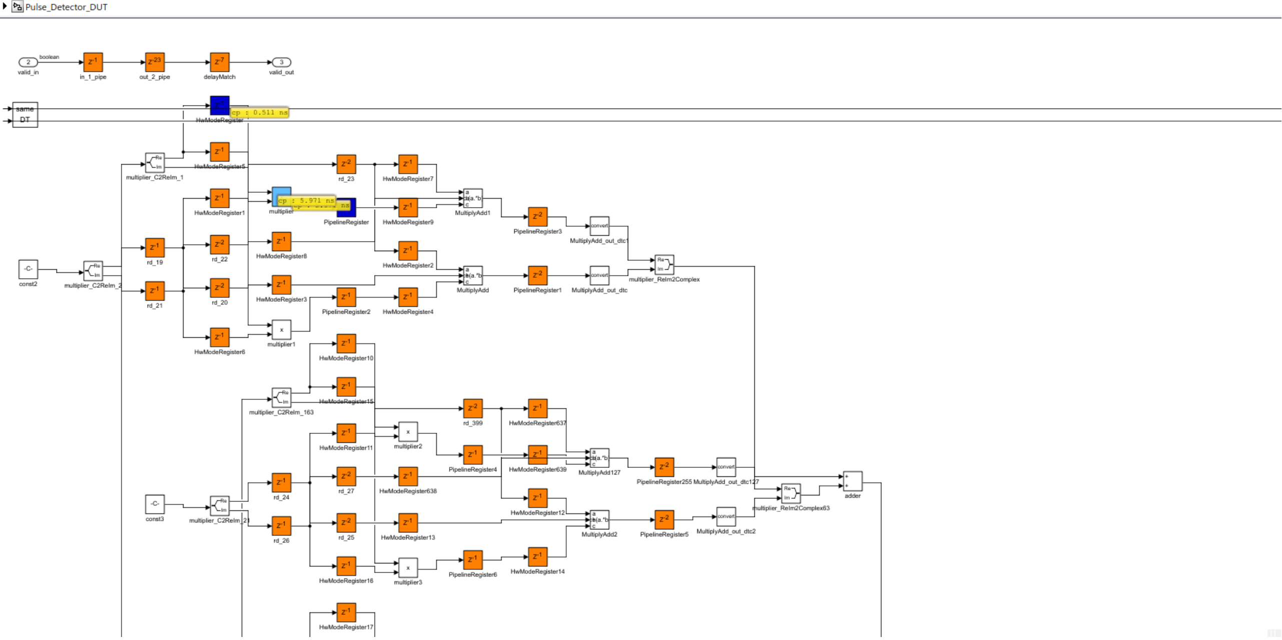
Generate HDL code and open the generated model `gm_pulse_detector`.

```
makehdl(DUTname)
```

```
### Working on the model <a href="matlab:open_system('pulse_detector')">pulse_detector</a>
### Generating HDL for <a href="matlab:open_system('pulse_detector/Pulse_Detector_DUT')">pulse_de
### Using the config set for model <a href="matlab:configset.showParameterGroup('pulse_detector'
### Running HDL checks on the model 'pulse_detector'.
### Begin compilation of the model 'pulse_detector'...
### Working on the model 'pulse_detector'...
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 31 cycles.
### Output port 2: 31 cycles.
### Output port 3: 31 cycles.
### Working on... <a href="matlab:configset.internal.open('pulse_detector', 'GenerateModel')">Gen
### Begin model generation 'gm_pulse_detector'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\pulse_detector\gm_pulse_
### Estimated critical path for design: <a href="matlab:run('hdl_prj\hdlsrc\pulse_detector\critic
### Blocks that are not characterized for Critical Path Estimation: <a href="matlab:run('hdl_prj\
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsrc
### Generating new validation model: '<a href="matlab:open_system('hdl_prj\hdlsrc\pulse_detector\
### Validation model generation complete.
### Begin Verilog Code Generation for 'pulse_detector'.
### Unused logic removed during HDL code generation. To highlight the logic removed, click the fo
### To clear highlighting, click the following MATLAB script: hdl_prj\hdlsrc\pulse_detector\clear
### Working on... <a href="matlab:configset.internal.open('pulse_detector', 'Traceability')">Trac
### Working on pulse_detector/Pulse_Detector_DUT as hdl_prj\hdlsrc\pulse_detector\Pulse_Detector_
### Code Generation for 'pulse_detector' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'pulse_detector' complete with 0 errors, 0 warnings, and 3 messages.
### HDL code generation complete.
```

The optimization options applied to the model add 31 cycles of latency to the generated model. This increase in latency is due to the large number of output pipelines, which reduced the critical path.

Below is a snippet of the generated model with the estimated critical path highlighted. The estimated critical path is now only through a single multiplier block because distributed pipelining broke up the long chain of adders in the Discrete FIR Filter by inserting pipelines throughout the original critical path. As a result, the adder chain with pipelines now has a shorter path than the multiplier block by itself.



The estimated critical path is now 5.971 ns, which is much smaller than the initial value of 93.872 ns. This decrease in estimated critical path is a result of the speed optimizations applied across the entire DUT, including its subsystems and the MATLAB Function block.

Check if your timing constraints have been met by running the HDL Workflow Advisor through step 4.2.2. The synthesis results show that there is positive slack, indicating that the timing constraint is met. The clock frequency is now 255 MHz and is higher than the target frequency of 230 MHz and the initial clock frequency of 110 MHz.

Parsed resource report file: Pulse_Detector_DUT_utilization_placed.rpt.

Resource summary			
Resource	Usage	Available	Utilization (%)
Slice LUTs	8798	218600	4.02
Slice Registers	13834	437200	3.16
DSPs	258	900	28.67
Block RAM Tile	0	545	0.00
URAM	0	0	

Parsed timing report file: timing_post_route.rpt.

Timing summary	
	Value
Requirement	4.3478 ns
Data Path Delay	3.647 ns
Slack	0.435 ns
Clock Frequency	255.57 MHz

Apply Distributed Pipelining Using Synthesis Timing Estimates

While the clock frequency is achieved from the previous optimizations applied, another speed optimization you can use specifically for distributed pipelining is using synthesis timing estimates. For more information, see “Distributed Pipelining Using Synthesis Timing Estimates” on page 21-136.

```
hdlset_param(modelname, 'UseSynthesisEstimatesForDistributedPipelining','on');
```

Report the nondefault HDL parameters for the model.

```
hdlsaveparams(modelname)
```

```
%% Set Model 'pulse_detector' HDL parameters
hdlset_param('pulse_detector', 'AdaptivePipelining', 'on');
hdlset_param('pulse_detector', 'Backannotation', 'on');
hdlset_param('pulse_detector', 'CriticalPathEstimation', 'on');
hdlset_param('pulse_detector', 'DistributedPipelining', 'on');
hdlset_param('pulse_detector', 'GenerateHDLTestBench', 'off');
hdlset_param('pulse_detector', 'GenerateValidationModel', 'on');
hdlset_param('pulse_detector', 'HDLSubsystem', 'pulse_detector/Pulse_Detector_DUT');
hdlset_param('pulse_detector', 'MinimizeClockEnables', 'on');
hdlset_param('pulse_detector', 'PipelineDistributionPriority', 'Performance');
hdlset_param('pulse_detector', 'ResetType', 'Synchronous');
hdlset_param('pulse_detector', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('pulse_detector', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('pulse_detector', 'SynthesisToolDeviceName', 'xc7z045');
hdlset_param('pulse_detector', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('pulse_detector', 'SynthesisToolSpeedValue', '-1');
hdlset_param('pulse_detector', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('pulse_detector', 'TargetFrequency', 230);
hdlset_param('pulse_detector', 'TargetLanguage', 'Verilog');
hdlset_param('pulse_detector', 'Traceability', 'on');
hdlset_param('pulse_detector', 'UseSynthesisEstimatesForDistributedPipelining', 'on');

% Set SubSystem HDL parameters
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'FlattenHierarchy', 'on');
```

```
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'InputPipeline', 1);
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'OutputPipeline', 23);

hdlset_param('pulse_detector/Pulse_Detector_DUT/Discrete FIR Filter', 'Architecture', 'Fully Para

hdlset_param('pulse_detector/Pulse_Detector_DUT/Local Peak/MATLAB Function', 'Architecture', 'MAT
```

Generate HDL code and open the generated model `gm_pulse_detector`.

```
makehdl(DUTname)
```

```
### Working on the model <a href="matlab:open_system('pulse_detector')">pulse_detector</a>
### Generating HDL for <a href="matlab:open_system('pulse_detector/Pulse_Detector_DUT')">pulse_de
### Using the config set for model <a href="matlab:configset.showParameterGroup('pulse_detector'
### Running HDL checks on the model 'pulse_detector'.
### Begin compilation of the model 'pulse_detector'...
### Working on the model 'pulse_detector'...
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 31 cycles.
### Output port 2: 31 cycles.
### Output port 3: 31 cycles.
### Working on... <a href="matlab:configset.internal.open('pulse_detector', 'GenerateModel')">Gen
### Begin model generation 'gm_pulse_detector'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\pulse_detector\gm_pulse_
### Estimated critical path for design: <a href="matlab:run('hdl_prj\hdlsrc\pulse_detector\critic
### Blocks that are not characterized for Critical Path Estimation: <a href="matlab:run('hdl_prj\
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsrc
### Generating new validation model: '<a href="matlab:open_system('hdl_prj\hdlsrc\pulse_detector\
### Validation model generation complete.
### Begin Verilog Code Generation for 'pulse_detector'.
### Unused logic removed during HDL code generation. To highlight the logic removed, click the fo
### To clear highlighting, click the following MATLAB script: hdl_prj\hdlsrc\pulse_detector\clear
### Working on... <a href="matlab:configset.internal.open('pulse_detector', 'Traceability')">Trac
### Working on pulse_detector/Pulse_Detector_DUT as hdl_prj\hdlsrc\pulse_detector\Pulse_Detector_
### Code Generation for 'pulse_detector' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'pulse_detector' complete with 0 errors, 1 warnings, and 3 messages.
### HDL code generation complete.
```

The optimizations applied to the model add 31 cycles of latency to the generated model.

The estimated critical path is 5.971 ns, which is the same as the previous model. Because critical path estimation is an estimate, you need to run synthesis to determine how this option changed your clock frequency and resources.

Depending on your design, using synthesis timing estimates for distributed pipelining can improve your clock speed or keep it similar to the clock frequency when using synthesis estimates for distributed pipelining is not enabled.

Using synthesis timing estimates for distributed pipelining can increase your clock speed if your design has different types of components with nonzero propagation delay, and the delays of each type of block are different from one another. For example, if you have Add and Multiply blocks in your

design that have nonzero propagation delays, but the numerical value of the delays are significantly different, using synthesis timing estimates for distributed pipelining can increase your clock frequency.

Using synthesis timing estimates for distributed pipelining might not improve your clock frequency if there is such a large number of delays in the model that regardless of how distributed pipelining moves the delays, they are placed in the same or similar locations in the design.

Check if your timing constraints have been met by running the HDL Workflow Advisor through step 4.2.2. The synthesis results show that there is positive slack, which indicates that the timing constraint is met. The clock frequency is now 236 MHz and is higher than the target frequency of 230 MHz.

Parsed resource report file: Pulse_Detector_DUT_utilization_placed.rpt.

Resource summary			
Resource	Usage	Available	Utilization (%)
Slice LUTs	9212	218600	4.21
Slice Registers	13998	437200	3.20
DSPs	258	900	28.67
Block RAM Tile	0	545	0.00
URAM	0	0	

Parsed timing report file: timing_post_route.rpt.

Timing summary	
	Value
Requirement	4.3478 ns
Data Path Delay	4.225 ns
Slack	0.127 ns
Clock Frequency	236.92 MHz

Apply Area Optimizations

The target frequency is now met due the applied speed optimizations, but the model uses significant resources. The **Resource summary** table displays the resource usage for the model.

To reduce the area used by the design, apply sharing to the DUT. If you look at the generated model you can see a significant portion of the FIR Filter block uses adders. Ensure resource sharing is applied to the adders by enabling the sharing of adders. You can increase or decrease the sharing factor depending on the area requirements for your design. For this example, set the `SharingFactor` to `8` to reduce the DSP usage by about a factor of eight.

```
hdlset_param(DUTname,'SharingFactor', 8);
hdlset_param(modelname, 'ShareAdders','on');
```

Resource sharing creates a multirate model due to the serializer subnetworks added in the generated model. Synthesis tools require a constraint file to indicate if a path is using a slower clock enable where it can tolerate multiple clock cycles. If a synthesis tool does not have a constraint file when there are multiple clock rates, the synthesis tool might report that the critical path is in a slow part of the logic that does not need to be executed every single clock cycle. You can generate this constraint file by enabling the generation of a multicycle path constraint for your model. Generating this constraint file can also improve the timing of your design and reduce synthesis time. For more

information, see “Meet Timing Requirements Using Enable-Based Multicycle Path Constraints” on page 20-32.

```
hdlset_param(modelname, 'MulticyclePathConstraints', 'on');
```

Report the nondefault HDL parameters for the model.

```
hdlsaveparams(modelname)
```

```
%% Set Model 'pulse_detector' HDL parameters
hdlset_param('pulse_detector', 'AdaptivePipelining', 'on');
hdlset_param('pulse_detector', 'Backannotation', 'on');
hdlset_param('pulse_detector', 'CriticalPathEstimation', 'on');
hdlset_param('pulse_detector', 'DistributedPipelining', 'on');
hdlset_param('pulse_detector', 'GenerateHDLTestBench', 'off');
hdlset_param('pulse_detector', 'GenerateValidationModel', 'on');
hdlset_param('pulse_detector', 'HDLSubsystem', 'pulse_detector/Pulse_Detector_DUT');
hdlset_param('pulse_detector', 'MinimizeClockEnables', 'on');
hdlset_param('pulse_detector', 'MulticyclePathConstraints', 'on');
hdlset_param('pulse_detector', 'PipelineDistributionPriority', 'Performance');
hdlset_param('pulse_detector', 'ResetType', 'Synchronous');
hdlset_param('pulse_detector', 'ShareAdders', 'on');
hdlset_param('pulse_detector', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('pulse_detector', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('pulse_detector', 'SynthesisToolDeviceName', 'xc7z045');
hdlset_param('pulse_detector', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('pulse_detector', 'SynthesisToolSpeedValue', '-1');
hdlset_param('pulse_detector', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('pulse_detector', 'TargetFrequency', 230);
hdlset_param('pulse_detector', 'TargetLanguage', 'Verilog');
hdlset_param('pulse_detector', 'Traceability', 'on');
hdlset_param('pulse_detector', 'UseSynthesisEstimatesForDistributedPipelining', 'on');

% Set SubSystem HDL parameters
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'FlattenHierarchy', 'on');
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'InputPipeline', 1);
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'OutputPipeline', 23);
hdlset_param('pulse_detector/Pulse_Detector_DUT', 'SharingFactor', 8);

hdlset_param('pulse_detector/Pulse_Detector_DUT/Discrete FIR Filter', 'Architecture', 'Fully Para

hdlset_param('pulse_detector/Pulse_Detector_DUT/Local Peak/MATLAB Function', 'Architecture', 'MA
```

Generate the HDL code and open the generated model `gm_pulse_detector`.

```
makehdl(DUTname)
```

```
### Working on the model <a href="matlab:open_system('pulse_detector')">pulse_detector</a>
### Generating HDL for <a href="matlab:open_system('pulse_detector/Pulse_Detector_DUT')">pulse_de
### Using the config set for model <a href="matlab:configset.showParameterGroup('pulse_detector'
### Running HDL checks on the model 'pulse_detector'.
### Begin compilation of the model 'pulse_detector'...
### Working on the model 'pulse_detector'...
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 96 cycles.
### Output port 2: 96 cycles.
### Output port 3: 96 cycles.
### Working on... <a href="matlab:configset.internal.open('pulse_detector', 'GenerateModel')">Gen
```

```
### Begin model generation 'gm_pulse_detector'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\pulse_detector\gm_pulse_
### Estimated critical path for design: <a href="matlab:run('hdl_prj\hdlsrc\pulse_detector\critic
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### Blocks that are not characterized for Critical Path Estimation: <a href="matlab:run('hdl_prj\
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsrc
### Generating new validation model: '<a href="matlab:open_system('hdl_prj\hdlsrc\pulse_detector\
### Validation model generation complete.
### Begin Verilog Code Generation for 'pulse_detector'.
### Unused logic removed during HDL code generation. To highlight the logic removed, click the fo
### To clear highlighting, click the following MATLAB script: hdl_prj\hdlsrc\pulse_detector\clear
### MESSAGE: The design requires 8 times faster clock with respect to the base rate = 1.
### Begin Verilog Code Generation for 'Pulse_Detector_DUT_tc'.
### Working on Pulse_Detector_DUT_tc as hdl_prj\hdlsrc\pulse_detector\Pulse_Detector_DUT_tc.v.
### Code Generation for 'Pulse_Detector_DUT_tc' completed.
### Working on... <a href="matlab:configset.internal.open('pulse_detector', 'Traceability')">Trac
### Working on pulse_detector/Pulse_Detector_DUT as hdl_prj\hdlsrc\pulse_detector\Pulse_Detector_
### Code Generation for 'pulse_detector' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Writing Vivado multicycle constraints XDC file <a href="matlab:edit('hdl_prj\hdlsrc\pulse_det
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'Pulse_Detector_DUT_tc' complete with 0 errors, 1 warnings, and 0 messages.
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'pulse_detector' complete with 0 errors, 3 warnings, and 5 messages.
### HDL code generation complete.
```

The optimizations add 96 cycles of latency to the generated model. This increase in added latency is due to the resource sharing area optimizations. Sharing resources for multiple operations serializes the design and as a result, the optimized design requires more clock cycles to produce the output. In this example because there is no requirement to maximize throughput, reducing the area consumed at the cost of added latency is a good trade-off.

The critical path estimated is 6.213 ns, which is slightly larger than without area optimizations applied.

Check if your timing constraints have been met by running the HDL Workflow Advisor through step 4.2.2. The synthesis results show that there is positive slack, which indicates that the timing constraint is met. The clock frequency is now 234 MHz and is higher than the target frequency of 230 MHz.

As indicated by the resource summary, the DSP usage decreased significantly from 28.67% of the DSPs used to 3.67% used. The trade-off for this decrease in area is a minor decrease in clock frequency and a slight increase in the usage of slice LUTs and slice Registers.

Parsed resource report file: Pulse_Detector_DUT_utilization_placed.rpt.

Resource summary			
Resource	**Usage**	**Available**	**Utilization (%)**
Slice LUTs	17129	218600	7.84
Slice Registers	51213	437200	11.71
DSPs	33	900	3.67
Block RAM Tile	0	545	0.00
URAM	0	0	

Parsed timing report file: timing_post_route.rpt.

Timing summary	
	Value
Requirement	4.3478 ns
Data Path Delay	3.953 ns
Slack	0.082 ns
Clock Frequency	234.42 MHz

Summary

Below is a table summarizing the impact of the HDL optimizations applied in each section.

	LUTs (% used)	Registers (% used)	DSPs (% used)	RAMs (% used)	Slack (ns)	Clock Frequency (MHz)	Latency (cycles)
Apply Base Speed Optimizations	1.94%	1.45%	28.67%	0%	-4.731	110	9
Apply Distributed Pipelining	4.02%	3.16%	28.67%	0%	0.435	255.57	31
Apply Distributed Pipelining Using Synthesis Timing Estimates	4.21%	3.20%	28.67%	0%	0.127	236.92	31
Apply Area Optimizations	7.84%	11.71%	3.67%	0%	0.082	234.42	96

See Also

“Critical Path Estimation Without Running Synthesis” on page 21-192 | “Generated Model and Validation Model” on page 21-10

Adaptive Pipelining

In this section...

"Requirements" on page 21-181
"Specify Adaptive Pipelining" on page 21-182
"Supported Blocks" on page 21-182
"Pipeline Insertion for Rate Transition and Downsample Blocks" on page 21-182
"Pipeline Insertion for Product and Gain Blocks" on page 21-183
"Pipeline Insertion for Multiply-Add and Multiply-Accumulate Blocks" on page 21-184
"Pipeline Insertion for MATLAB Function Blocks" on page 21-186
"Adaptive Pipelining Report" on page 21-186

Certain patterns or combination of blocks with registers can improve the achievable clock frequency and reduce the area usage on the FPGA boards. The adaptive pipelining optimization creates these patterns by inserting pipeline registers to the blocks in your design. To determine the optimal number of pipeline registers to insert in your design, the optimization considers the target device, target frequency, multiplier word lengths, and the settings in the HDL Block Properties. Use adaptive pipelining with:

- Clock-rate pipelining to insert pipeline registers at a faster clock-rate instead of the slower data-rate. With clock-rate pipelining, you can design your DUT at one rate and then specify an oversampling value for your design using either the **Treat Simulink rates as actual hardware rates** or **Oversampling factor** parameter. To learn more about these parameters, see Treat Simulink rates as actual hardware rates and Oversampling factor.
- Resource sharing which saves area and timing because the code generator shares resources and inserts adaptive pipeline registers.

By default, the adaptive pipelining optimization is disabled on the model. In certain situations, you must enable this optimization before generating HDL code. See "Design Patterns That Require Adaptive Pipelining" on page 21-188.

Requirements

- For HDL Coder to insert adaptive pipelines, specify the target device. When your design has multipliers, specify the target device and the target frequency.

  Note If you use a target device that is not characterized for adaptive pipelining, the optimization uses Xilinx Virtex®-7 when `Xilinx Vivado` is specified as the **Synthesis Tool**, and uses Intel Stratix® V when the **Synthesis Tool** is `Altera Quartus II` or `Intel Quartus Pro`.

- Make sure that delay balancing is enabled for the subsystem that you want HDL Coder to insert adaptive pipelines for. If you disable delay balancing, the code generator does not insert adaptive pipelines.
- Make sure that your design does not have floating-point data types or operations.

Note In some cases, when you have blocks inside a feedback loop, adaptive pipelining is unable to insert the required number of pipeline registers at the output. Delay balancing can then fail.

Specify Adaptive Pipelining

You can set adaptive pipelining for an entire model or, for finer control, you can set adaptive pipelining for subsystems within the top-level DUT subsystem.

Enable Adaptive Pipelining for a Model

By default, adaptive pipelining is disabled at the model level. You can enable adaptive pipelining in one of the following ways:

- In the HDL Workflow Advisor, on the **HDL Code Generation** > **Set Code Generation Options** > **Set Optimization Options** > **Pipelining** tab, select **Adaptive pipelining**.
- In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Optimization** > **Pipelining** tab, select **Adaptive pipelining** and click **OK**.
- At the command line, use the `makehdl` or `hdlset_param` function to set Adaptive pipelining to `on`.

  ```
  hdlset_param(gcs, 'AdaptivePipelining', 'on')
  ```

Enable Adaptive Pipelining for a Subsystem

By default, subsystems in your model inherit the model-level adaptive pipelining setting. If you want HDL Coder to selectively enable adaptive pipelines for a subsystem in your model, set **AdaptivePipelining** to `on` for that subsystem.

To learn how to set adaptive pipelining for a subsystem, see “Set Adaptive Pipelining For a Subsystem” on page 19-4.

Supported Blocks

Adaptive pipelining supports these multipliers, multiply accumulate, and rate transition blocks for automatic pipeline insertion.

- Downsample
- Rate Transition
- Product
- Gain
- Multiply-Add
- Multiply-Accumulate
- MATLAB Function

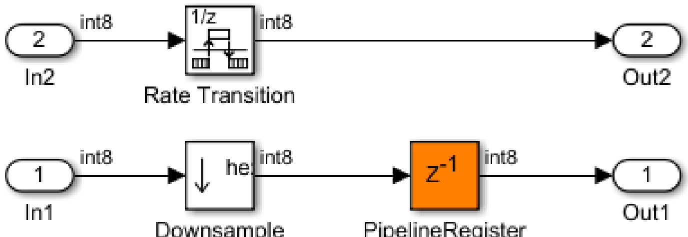
Pipeline Insertion for Rate Transition and Downsample Blocks

To insert adaptive pipelines for the Rate Transition and Downsample blocks:

1. Specify the target device.
2. Make sure that Downsample blocks have a **Downsample factor** greater than two.

When generating code, HDL Coder inserts a pipeline register at the output port of the Downsample block. Addition of the pipeline register can avoid the bypass register logic, which saves area on the target FPGA.

This figure is the generated model for the blocks with Xilinx Virtex7 as the target FPGA device.



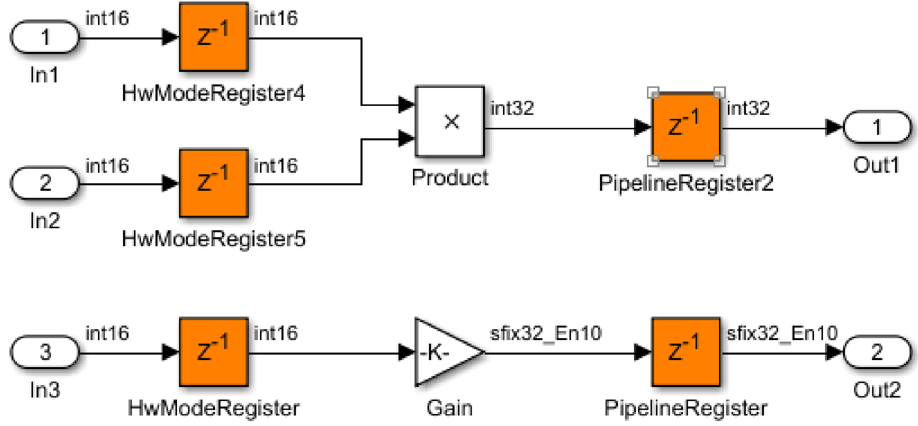
Pipeline Insertion for Product and Gain Blocks

To insert adaptive pipelines for these blocks:

1. Specify the target device.
2. Specify a target frequency greater than zero.

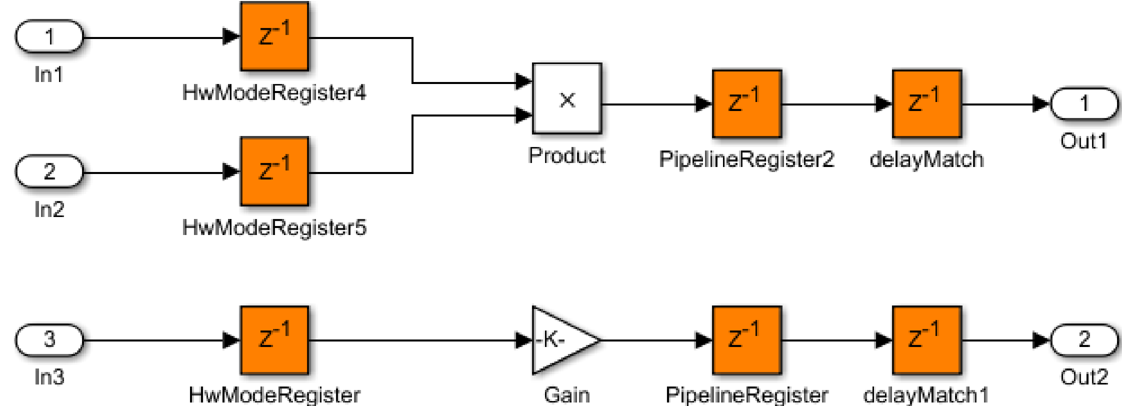
When generating code, HDL Coder inserts registers at the input and output ports of the blocks. The combination of multipliers with the registers can potentially map to DSP units on the target device.

This figure is the generated model for the Product and Gain blocks with Intel Arria10 as the target FPGA device and a target frequency of 500 MHz. The inputs to the blocks are of type `int16`.



The pattern and number of pipeline registers that HDL Coder inserts can vary depending on the target device, target frequency, and the multiplier word lengths.

This figure is the generated model for the blocks with Xilinx Virtex7 as the target FPGA device and a target frequency of 1500 MHz. The inputs are of type `int8`.



The blocks have a different number of pipeline registers at the output ports. To match the delays, HDL Coder adds a delay at the output of the Product and Gain blocks.

Pipeline Insertion for Multiply-Add and Multiply-Accumulate Blocks

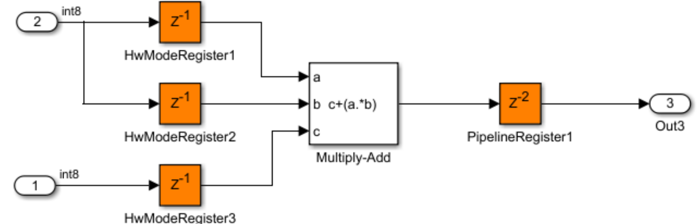
To insert adaptive pipelines for these blocks:

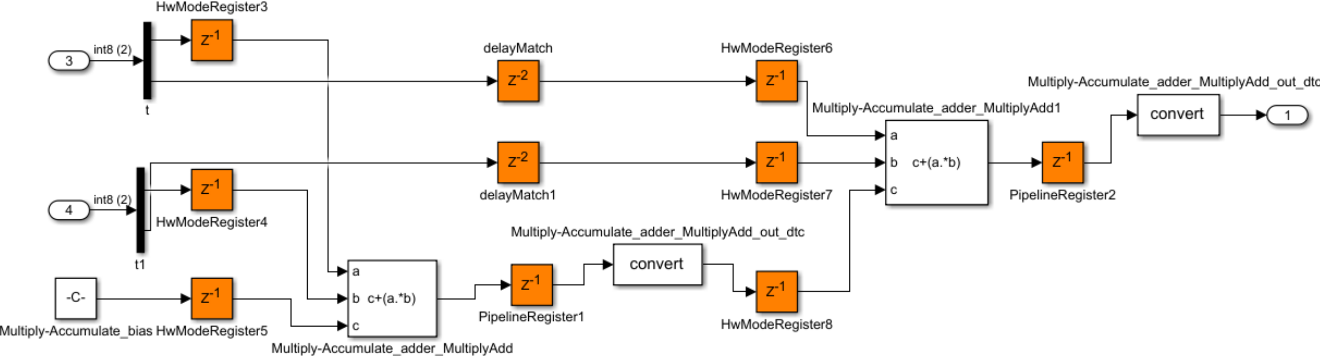
1. Specify the target device.
2. Specify a target frequency greater than zero.
3. Use the `Parallel` HDL architecture for the Multiply-Accumulate block. For an input vector of size `N`, this architecture uses `N` Multiply-Add blocks in series to compute the result.

Caution The Multiply-Add block with **PipelineDepth** set to `auto` or a value greater than zero and the Multiply-Accumulate block with HDL architecture specified as `Parallel` ignore the adaptive pipelining setting. If you specify the target FPGA device and a target frequency greater than zero, the code generator inserts pipeline registers at the inputs and outputs of the block even when adaptive pipelining is disabled.

When generating code, HDL Coder inserts registers at the input and output ports of the blocks. The combination of the blocks with the registers can potentially map to DSP units on the target device.

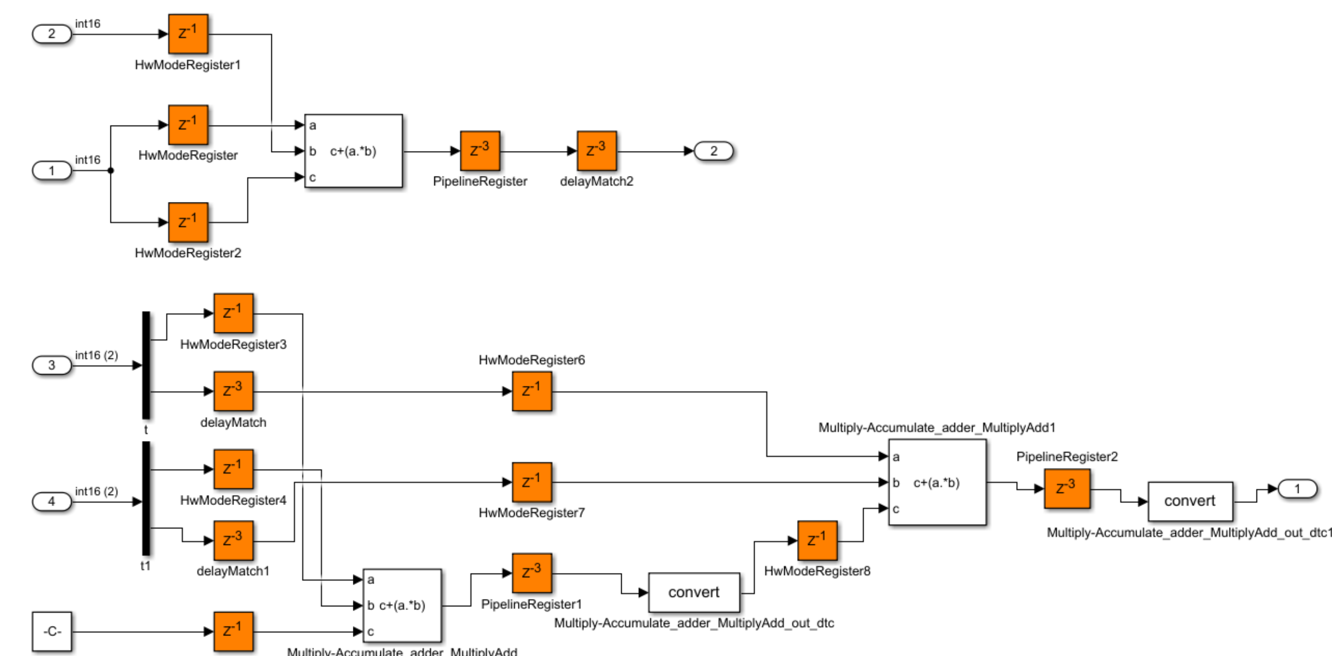
This figure is the generated model for the Multiply-Add and Multiply-Accumulate with Intel Arria10 as the target FPGA device and a target frequency of 500 MHz. The inputs to the blocks are of type `int8`.





The pattern and number of pipeline registers that HDL Coder inserts can vary depending on the target device, target frequency, and the multiplier word lengths.

This figure is the generated model for the blocks with Xilinx Virtex7 as the target FPGA device and a target frequency of 1500 MHz. The inputs are of type `int16`.



Pipeline Insertion for MATLAB Function Blocks

To insert adaptive pipelines for MATLAB Function blocks:

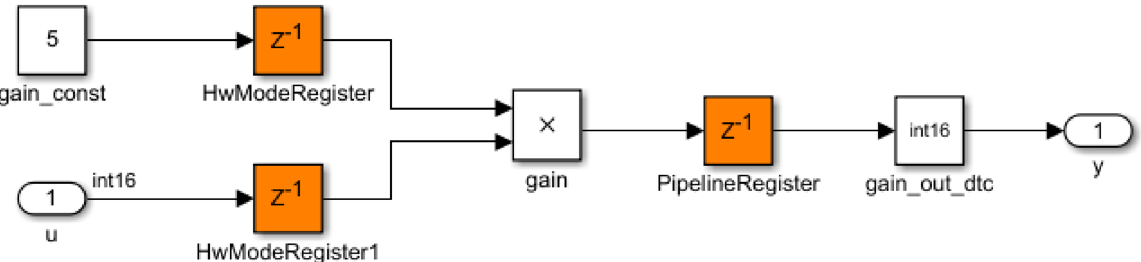
1. Specify the target device.
2. Specify a target frequency greater than zero.
3. Set the HDL architecture for the MATLAB Function blocks to `MATLAB Datapath`.

HDL Coder treats MATLAB Function blocks that have their architecture set to `MATLAB Datapath` like regular Subsystems. The code generator converts the MATLAB algorithm to a Simulink block diagram. If the Simulink diagram uses blocks that are supported by adaptive pipelining, such as Product or Add blocks, the code generator inserts pipeline registers at the input and output ports of the blocks. The combination of multipliers with the registers can potentially map to DSP units on the target device.

Consider a MATLAB Function block that uses the `MATLAB Datapath` architecture. This code is the algorithm inside the MATLAB Function block.

```
function y = fcn(u)

y = u*5;
```

This figure is the generated model for the MATLAB Function block that has Intel Arria10 as the target FPGA device and a target frequency of 500 MHz. The inputs to the blocks are of type `int16`. The code generator inferred the algorithm as a multiplication by a constant and inserted adaptive pipelines at the input and output.



See “HDL Applications for the MATLAB Function Block” on page 27-2.

Adaptive Pipelining Report

To see the adaptive pipelining information in the report, before you generate code for each Subsystem or model reference, enable the Code Generation report. In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select **Generate optimization report**.

When you generate HDL code, the Code Generation report opens when code generation is complete. Select the **Adaptive Pipelining** section of the Optimization report.

The Adaptive Pipelining report displays the status of the adaptive pipelining optimization and whether adaptive pipelines were inserted in your design.

When you enable adaptive pipelining and generate HDL code, the report displays:

- The blocks that have inserted pipeline registers. To see the pipeline registers inserted to the blocks in your design, click the link to the block.
- The number of pipeline registers inserted.
- Additional notes.
- If adaptive pipelining is not completed, the criteria that caused it to fail.

See Also

"AdaptivePipelining" on page 19-4 | "Clock-Rate Pipelining" on page 21-148 | Balance delays

Related Examples

- "Design Patterns That Require Adaptive Pipelining" on page 21-188
- "Iteratively Maximize Clock Frequency by Using Speed Optimizations" on page 21-167

More About

- "Create and Use Code Generation Reports" on page 23-2

Design Patterns That Require Adaptive Pipelining

Certain patterns or combination of blocks with registers can improve the achievable clock frequency and reduce the area usage on FPGAs. The adaptive pipelining optimization creates these patterns by inserting pipeline registers to the blocks in your design. To determine the optimal number of pipeline registers to insert in your design, the optimization considers the target device, target frequency, multiplier word lengths, and the settings in the HDL Block Properties. See “Adaptive Pipelining” on page 21-181.

By default, the adaptive pipelining optimization is disabled on a model. If you decide to use this optimization, you must enable it.

In many situations, you can selectively insert pipelines in your model and generate efficient HDL code without enabling the adaptive pipelining optimization. For example, you can selectively add pipelines next to a multiplier.

For certain design patterns, you must enable the adaptive pipelining optimization before generating code. For example, you might not be able to selectively add pipelines to the internal logic of a Discrete-Time Integrator block.

Guidelines on whether to enable adaptive pipelining:

- For releases previous to R2021a, the adaptive pipelining report for your model showed that the code generator inserted adaptive pipelines. Enable adaptive pipelining when you generate code for that model.
- If your model uses the blocks that are supported by the adaptive pipelining optimization and you do not selectively insert pipeline registers for these blocks, enable adaptive pipelining.
- If your model uses the blocks that are supported by the adaptive pipelining optimization and you enable any of the following optimizations, also enable adaptive pipelining:
  - Resource sharing
  - Streaming
  - Clock-rate pipelining.

See “Adaptive Pipelining Report” on page 21-186 and “Supported Blocks” on page 21-182.

The examples “Audio System That Uses Low Pass, Band Pass, and High Pass Filters” on page 21-188 and “Discrete FIR Filter That Uses Resource Sharing” on page 21-188 illustrate some situations in which you must enable adaptive pipelining before generating code.

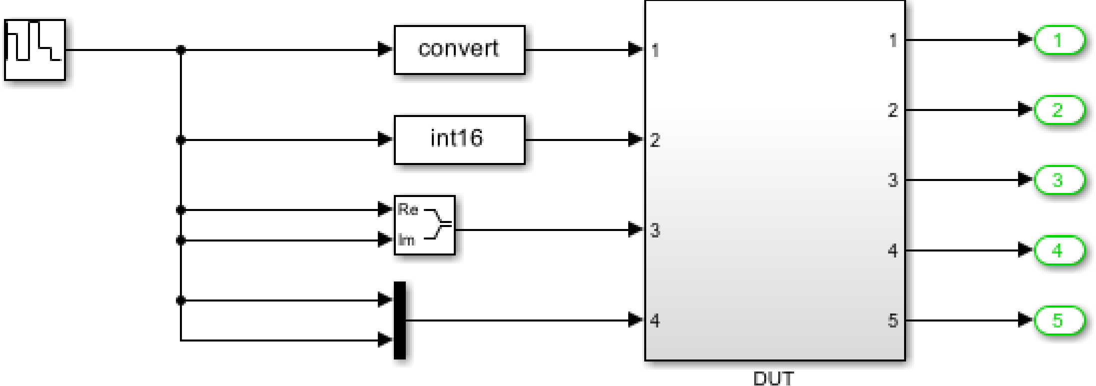
Audio System That Uses Low Pass, Band Pass, and High Pass Filters

See “Running an Audio Filter on Live Audio Input Using a Zynq Board” on page 40-181.

The model in this example has adaptive pipelining enabled. If you disable adaptive pipelining, the generated HDL code does not meet the timing constraints.

Discrete FIR Filter That Uses Resource Sharing

Open and explore this model for a Discrete FIR filter.



Copyright 2017 The MathWorks, Inc.

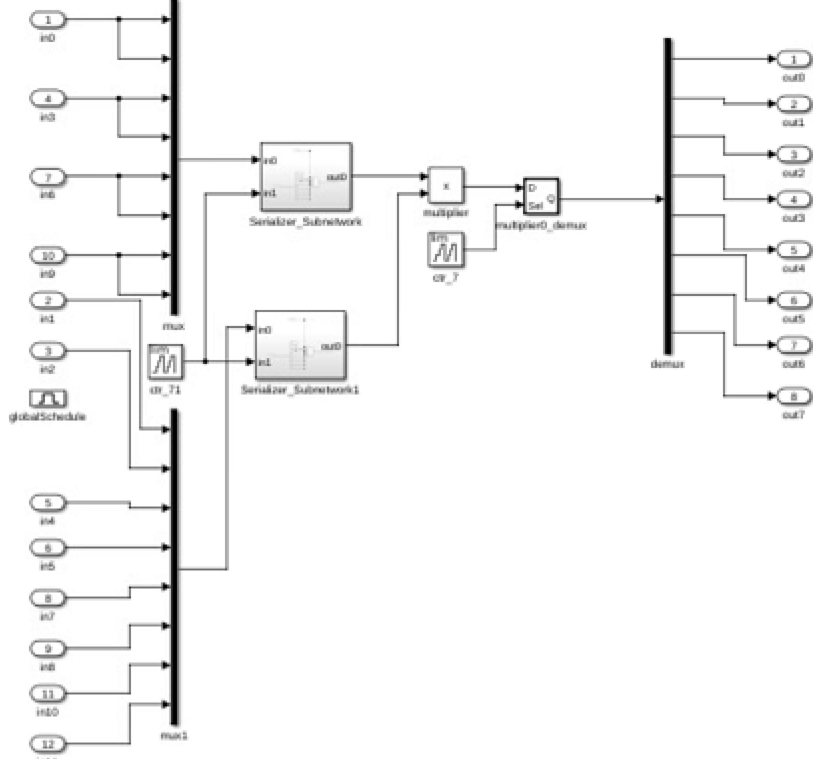
For the DUT Subsystem block, the HDL block property **SharingFactor** is set to `8`. So, this subsystem uses the resource sharing optimization.

Generate Code With Adaptive Pipelining Disabled

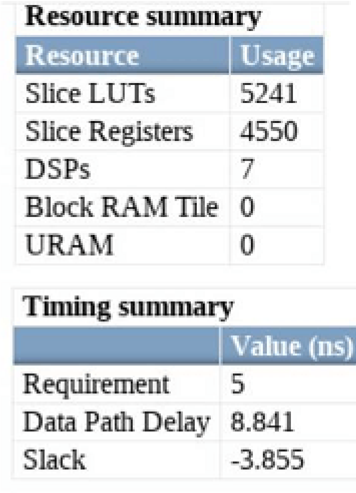
In the Configuration Parameters dialog box, on the **HDL Code Generation > Optimization > Pipelining** tab, clear **Adaptive pipelining**.

In the HDL Workflow Advisor, set **Target Frequency (MHz)** to `200` and **Synthesis tool** to `Xilinx Vivado`. Generate HDL code and perform FPGA synthesis. For more information on the code generation and synthesis steps, see “HDL Code Generation and FPGA Synthesis from Simulink Model”.

In the generated model, no adaptive pipelines are inserted.



The synthesis result shows a negative slack, indicating that timing constraints are not met.



Resource summary

Resource	Usage
Slice LUTs	5241
Slice Registers	4550
DSPs	7
Block RAM Tile	0
URAM	0

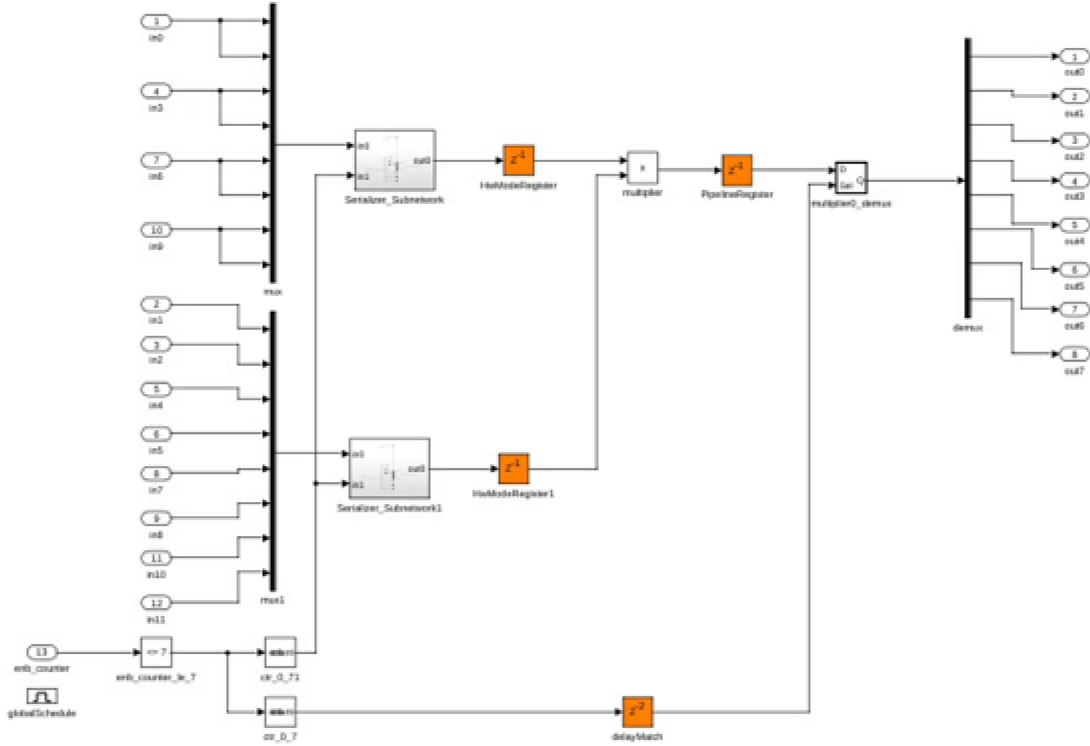
Timing summary

	Value (ns)
Requirement	5
Data Path Delay	8.841
Slack	-3.855

Generate Code With Adaptive Pipelining Enabled

In the Configuration Parameters dialog box, on the **HDL Code Generation > Optimization > Pipelining** tab, select **Adaptive pipelining**. Generate HDL code and perform FPGA synthesis as before.

In the generated model, you can see that the code generator has inserted adaptive pipelines for the multiplier blocks.



The synthesis result shows a positive slack, indicating that timing constraints are satisfied.

Resource summary

Resource	Usage
Slice LUTs	5062
Slice Registers	4951
DSPs	7
Block RAM Tile	0
URAM	0

Timing summary

	Value (ns)
Requirement	5
Data Path Delay	4.528
Slack	0.41

See Also

"Adaptive Pipelining" on page 21-181 | "Resource Sharing" on page 21-45 | "Streaming" on page 21-42

Related Examples

- "Running an Audio Filter on Live Audio Input Using a Zynq Board" on page 40-181

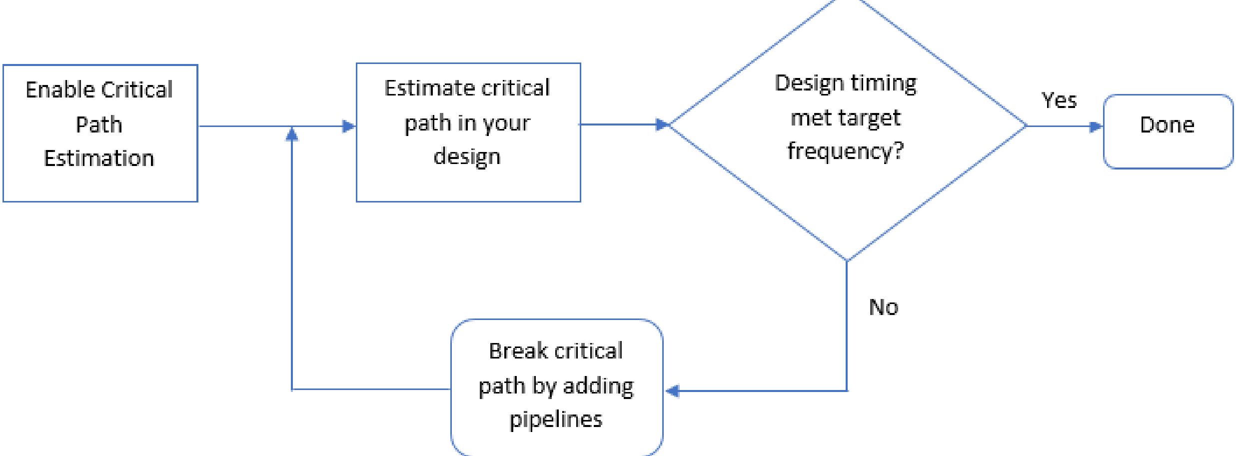
Critical Path Estimation Without Running Synthesis

In this section...

"Critical Path Estimation Process" on page 21-192
"Use Critical Path Estimation" on page 21-194
"Characterized Blocks" on page 21-195
"Considerations" on page 21-201
"Generate Custom Timing Database for Custom Tools and Devices" on page 21-202

The critical path is the combinational path between an input and output that has the maximum timing delay. To find the critical path in your design, use HDL Coder. To make the critical path timing meet the target frequency that you want your design to achieve, break the critical path by adding delays. The additional delays increase the latency and register usage on the target FPGA.

To quickly identify the most likely critical path in your design, use critical path estimation. You then do not have to run synthesis or generate HDL code. Critical path estimation speeds up this iterative process of finding the critical path. It optimizes the critical path until your design timing meets the target frequency that you want.



Critical path estimation speeds up the design iteration process. Critical path estimation is an alternative to annotating the critical path by performing **FPGA Synthesis and Analysis** with the HDL Workflow Advisor.

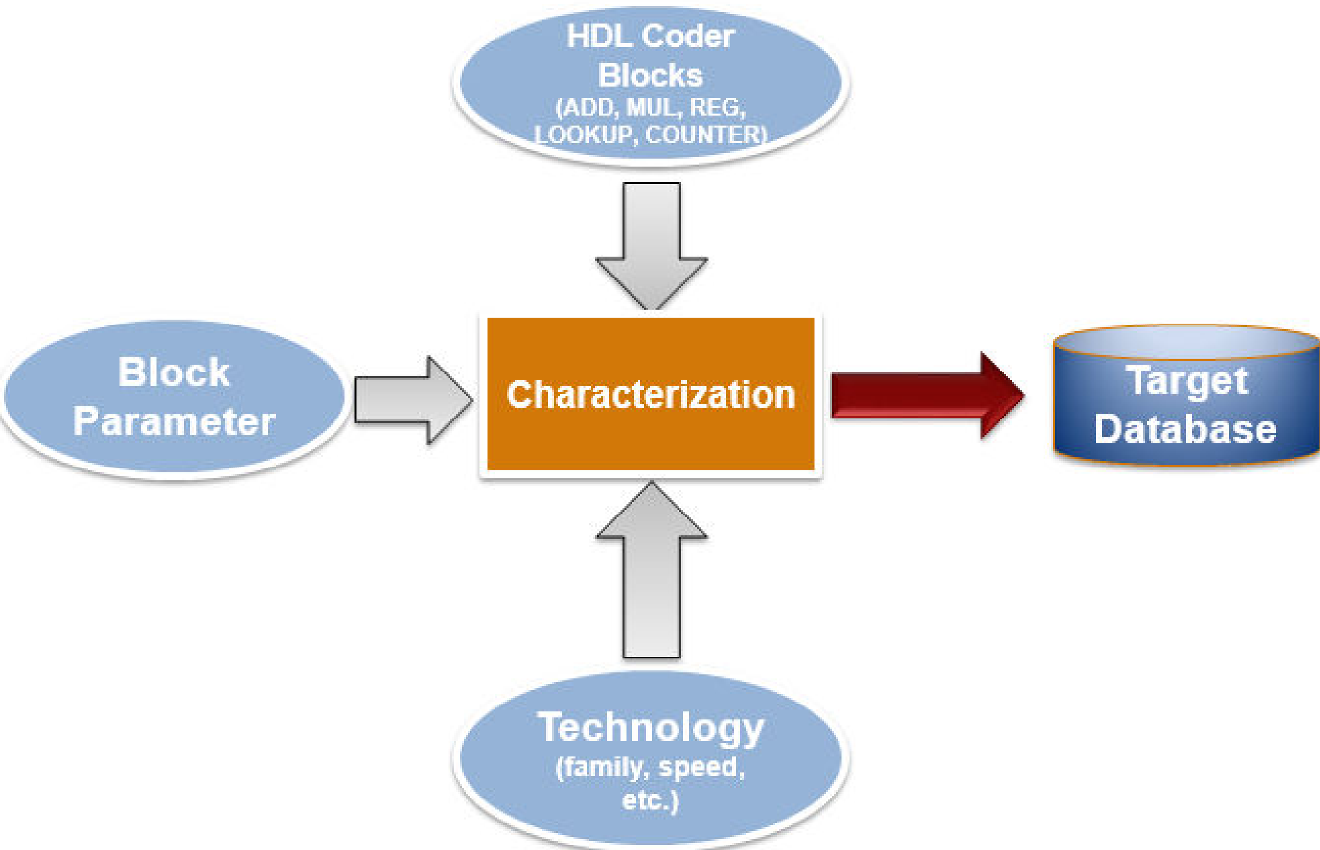
Critical Path Estimation Process

HDL Coder finds the estimated critical path by performing static timing analysis with timing data from target-specific timing databases. Generate timing databases for a specified target device family, target device speed grade, and target tool by using the `genhdltdb` function. By default, HDL Coder has timing databases for these target devices:

- Altera Cyclone V
- Intel Stratix V

- Xilinx Artix®-7, speed grade -1
- Xilinx Kintex-7, speed grade -1
- Xilinx Kintex UltraScale™, speed grade -1
- Xilinx Virtex-4, speed grade -10
- Xilinx Virtex-7, speed grade -1
- Xilinx Zynq®, speed grade -1
- Xilinx Zynq UltraScale+™, speed grade -1

To create timing databases, HDL Coder characterizes basic design components, such as Simulink blocks, block architectures, and subcomponents of those blocks, for specific target devices.



The code generator analyzes your model design to decompose it into the blocks and subcomponents in the timing databases. If your design consists of blocks or subcomponents in the timing databases, the code generator can estimate the timing critical path more accurately. If your design uses components that are not in the timing databases, a separate highlighting script is generated to show the uncharacterized blocks. If the timing data is incomplete for parts of your design, it is possible that the estimated critical path does not match your actual critical path.

If your target hardware is one of the target devices supported for critical path estimation, the timing numbers and estimated critical path are more accurate. If your target hardware is not a supported

device, or is not in the same device family, you can estimate the critical path, but it is possible that the timing numbers are not accurate.

Use Critical Path Estimation

You can estimate the critical path for your design in the Configuration Parameters dialog box or at the command line. To estimate the critical path in the Configuration Parameters dialog box:

1. Enable generation of critical path estimation report.
   a. In the **Apps** tab, select **HDL Coder**.
   b. In the HDL Code tab, select **Settings** > **Report Options**, and then select **Generate high-level timing critical path report**.
2. Disable HDL code generation for your model. In the **HDL Code Generation** > **Global Settings** > **Advanced** tab, clear the **Generate HDL Code** check box.

   To estimate the critical path in your design, you do not have to run the complete code generation process. When you disable HDL code generation, you run the process until HDL Coder creates the generated model and displays the critical path estimation script. You avoid running a larger portion of the code generation process, which saves time in estimating the critical path, especially for large models.
3. If your design contains floating-point data types, enable the `Native Floating Point` mode. In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Floating Point** pane, select **Use Floating Point**.
4. Set the path of the generated timing databases for your target device. In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Report** pane, select the **Generate high-level timing critical path report** parameter, and then set the path of your generated timing databases by clicking **Browse** and selecting the target folder.

   By default, the target folder shows some of the saved timing database folders based on your target configuration. If the **Custom Timing Database Directory** box is empty or the target configuration has no timing databases, by default, HDL Coder uses timing databases for the Xilinx Virtex -7, speed grade -1 device to generate the critical path estimation report.

   You can also generate timing databases for a specified target device family, target device speed grade, and target tool by using the `genhdltdb` function.
5. Generate a critical path estimation report. In the **HDL Code Generation** pane, click **Apply**, and then click **Generate**.

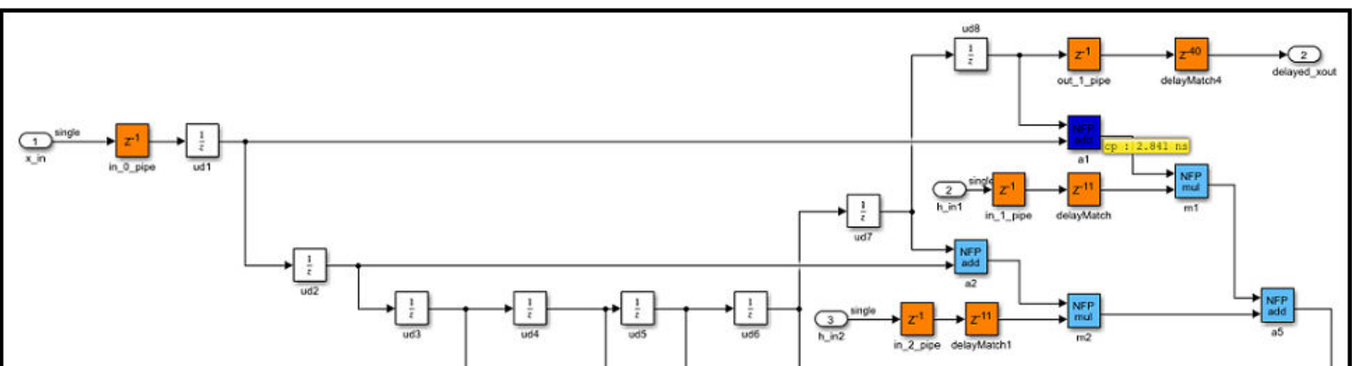
HDL Coder generates a critical path estimation report and displays messages in the MATLAB Command Window that include a link to a highlighting script and a script that clears the highlighting.

To generate the report at the command line, use this code. Specify the `modelname` and `dutname` variables based on the design for which you want to estimate the critical path. Set the path of the generated timing databases for your target device by using the `hdlset_param` function. When you enable the generation of the critical path estimation report and do not set the timing database path for a target device, HDL Coder searches for the default timing databases for the specified target device family and target device speed grade. If timing databases for the specified target device are not available, by default, HDL Coder uses timing databases for the Xilinx Virtex-7, speed grade -1 device to generate the critical path estimation report. This example uses the `sfir_single` model.

```
% Specify model and subsystem names
modelname = 'sfir_single';
```

```
dutname = 'sfir_single/symmetric_fir';
open_system(modelname)

% Disable HDL code generation for faster generation
% of critical path estimation report
hdlset_param(modelname,'CriticalPathEstimation','on');
hdlset_param(modelname,'GenerateHDLCode','off');

% If design contains single data types,
% enable native floating-point support
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
hdlset_param(modelname,'FloatingPointTargetConfig',fpconfig);

% Set path of generated timing databases for target device
hdlset_param(modelname,'TimingDatabaseDirectory','C:\Work\Database');

% Generate report
makehdl(dutname)
```

When you click the link to the `criticalpathestimated` script, the code generator highlights the critical path in the generated model. In the generated model, you see the critical path timing information and blocks that are on this path. This figure shows a section of a Simulink model that has the critical path annotated. The native floating-point operators are highlighted in light blue and the delays are highlighted in orange. The block that is part of the critical path is highlighted in dark blue with the critical path value annotated beside the block. For more information, see “Generated Model and Validation Model” on page 21-10.



You can clear the highlighting by clicking the link to the `clearhighlighting` script.

To optimize the critical path, break the critical path by adding pipeline registers. If you select **Use Floating Point** to use native floating point, set the **LatencyStrategy** to `Max` to improve timing. Regenerate the critical path estimation report and the script that highlights the critical path in your design. You can repeat this process until your design timing meets the target frequency that you want.

Characterized Blocks

This table lists blocks that are characterized with fixed-point and single-precision native floating-point types. These blocks are part of the timing database for each supported target device.

Math Operations

Simulink Blocks	Fixed-Point Types	Single-Precision (Native Floating Point)
Abs	✓	✓
Add	✓ (The block cannot have more than two inputs)	✓
Subtract	✓	✓
Product	✓	✓
Gain	✓	✓
Gain to power of two	Not applicable	✓
Divide	Not applicable	✓
Rounding Function	Not applicable	✓
Unary Minus	✓	✓
Sign	✓	✓
Reshape	Not applicable	✓
Complex to Real-Imag	✓	Not applicable

Math Functions

Simulink Blocks	Fixed-Point Types	Single-Precision Types (Native Floating Point)
Reciprocal	✓	✓
Hypot	✓	✓
Rem	✓	✓
Mod	✓	✓
Sqrt	✓	✓
Reciprocal Sqrt	✓	✓
Exp	Not applicable	✓
Log	Not applicable	✓
Log10	Not applicable	✓
10^u	Not applicable	✓
Magnitude Square	✓	✓
Square	✓	✓
Pow	Not applicable	✓
Conj	Not applicable	Not applicable
Transpose	Not applicable	Not applicable
Hermitian	Not applicable	Not applicable

Trigonometric Functions

Simulink Blocks	Fixed-Point Types	Single-Precision Types (Native Floating Point)
Sin	Not applicable	✓
Cos	Not applicable	✓
Tan	Not applicable	✓
Sincos	Not applicable	✓
Asin	Not applicable	✓
Acos	Not applicable	✓
Atan	Not applicable	✓
Atan2	Not applicable	✓
Sinh	Not applicable	✓
Cosh	Not applicable	✓
Tanh	Not applicable	✓
Atanh	Not applicable	✓

Conversions and Comparisons

Simulink Blocks	Fixed-Point Types	Single-Precision Types (Native Floating Point)
Data Type Conversion	✓	✓
Float Typecast	Not applicable	✓
Relational Operator	✓	✓
Compare To Constant	✓	✓
MinMax	✓	✓

Logic and Bit Operations

Simulink Blocks	Fixed-Point Types	Single-Precision Types (Native Floating Point)
Bit Concat	✓	Not applicable
Extract Bits	✓	Not applicable
Bit Shift	✓	Not applicable
Bit Slice	✓	Not applicable
Bitwise Operator	✓	Not applicable
Logical Operator	✓	Not applicable
Detect Change	✓	Not applicable
Detect Decrease	✓	Not applicable
Detect Increase	✓	Not applicable
Detect Fall Negative	✓	Not applicable
Detect Fall Nonpositive	✓	Not applicable
Detect Rise Positive	✓	Not applicable
Detect Rise Nonnegative	✓	Not applicable
Interval Test	✓	Not applicable
Interval Test Dynamic	✓	Not applicable

Discrete and Signal Routing

Simulink Blocks	Fixed-Point Types	Single-Precision Types (Native Floating Point)
Unit Delay	✓	✓
Delay	✓	✓
Bus Creator	✓	✓
Bus Selector	✓	✓
Demux	✓	✓
Multiport Switch	✓	✓
Selector	✓	✓
Switch	✓	✓
Multiport Switch	✓	Not applicable
Bus Assignment	✓	Not applicable
Index Vector	✓	Not applicable
Vector Concatenate	✓	Not applicable
Resettable Delay	✓	Not applicable
Tapped Delay	✓	Not applicable
Unit Delay Enabled Synchronous	✓	Not applicable
Unit Delay Resettable Synchronous	✓	Not Applicable
Unit Delay Enabled Resettable Synchronous	✓	Not applicable
Discrete FIR Filter	✓	Not applicable
Discrete Transfer Fcn	✓	Not applicable
Zero-Order Hold	✓	Not applicable
Memory	✓	Not applicable

HDL Operations and HDL RAMs

Simulink Blocks	Fixed-Point Types	Single-Precision Types (Native Floating Point)
Counter Free-Running	✓	✓
Counter Limited	✓	✓
HDL Counter	✓	✓
Dual Port RAM	✓	✓
Dual Rate Dual Port RAM	✓	✓
Simple Dual Port RAM	✓	✓
Single Port RAM	✓	✓
Deserializer1D	✓	✓
Serializer1D	✓	✓

Signal Attributes and Lookup Tables

Simulink Blocks	Fixed-Point Types	Single-Precision Types (Native Floating Point)
Constant	✓	✓
1-D Lookup Table	✓	✓
2-D Lookup Table	✓	✓
n-D Lookup Table	✓	✓
Rate Transition	✓	✓
Signal Conversion	✓	✓
Signal Specification	✓	✓

User-Defined Functions

Simulink Blocks	Fixed-Point Types	Single-Precision Types (Native Floating Point)
MATLAB Function	Not Applicable	✓

If you have a MATLAB function block in your design and the HDL Block Property **Architecture** is set to `MATLAB Function`, when estimating the critical path and generating HDL code, you might see the MATLAB function block appear as an uncharacterized block in the code generation reports. To characterize the MATLAB function block, set the **Architecture** to `MATLAB Datapath`.

When you use MATLAB Function blocks and generate code by using the `MATLAB Datapath` architecture, HDL Coder converts the MATLAB algorithm to a Simulink block diagram in the generated model. In the generated model, critical path estimation can annotate the critical path inside the MATLAB Function block and across the MATLAB Function block boundary with other Simulink blocks. See also “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 21-205.

Considerations

Critical Path Estimation for Multirate Models

Critical path estimation does not consider the clock-gating information to different sequential elements in your design.

If your model contains multiple sample rates or uses speed and area optimizations that insert pipeline registers, your design becomes multirate and can have multicycle paths. For multirate models, critical path estimation treats the slow and fast data paths as running at the same rate. A data path that has a faster clock rate might be highlighted as the critical path when the design has another data path at a slower rate. This issue might cause critical path estimation to report inaccurate timing results.

To verify the estimated critical path information, open the HDL Workflow Advisor and run the `Generic ASIC/FPGA` workflow for your target device to the **Annotate model with synthesis result** task.

Critical Path Estimation in Native Floating Point Mode

If you have `single` data types in your design and you use the `Native Floating Point` mode by selecting **Use Floating Point**, the critical path estimation script sometimes highlights a single floating-point operator in the generated model. The code generator highlights a single block because floating-point algorithms are computation-intensive. The critical path can be an internal register-to-register path within the floating-point operator.

In this case, to optimize the critical path timing, set the **LatencyStrategy** to `Max` for the Simulink block corresponding to that operator.

Simulink blocks that use half-precision data types do not participate in critical path estimation. These blocks are highlighted by using the `0 ns` timing delay.

HDL Code Generation Behavior

When you enable critical path estimation, it is possible that the generated HDL code is different from the report for a Delay block that has an external reset or an enable port. For blocks such as MinMax, the number of generated HDL files might differ when you enable critical path estimation. This change occurs due to certain optimizations performed by the code generator when you enable this optimization. The optimization changes only how the code appears and does not affect the functionality.

Following are the Simulink blocks for which the generated HDL code can potentially be different.

- Delay block that has external reset or enable port
- MinMax
- Unit Delay Enabled Synchronous
- Unit Delay Resettable Synchronous
- Unit Delay Enabled Resettable Synchronous
- Enabled Delay
- Resettable Delay
- Tapped Delay

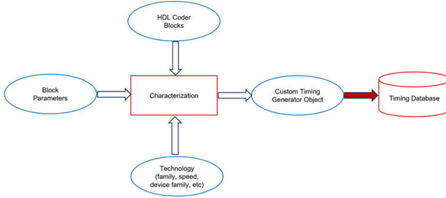
- Discrete FIR Filter
- Biquad Filter
- MATLAB Function

Inaccuracy in Critical Path Estimation

- Critical path estimation tries to account for routing delays by using an estimation factor. Without running place and route, it is difficult to accurately account for routing delays.
- HDL Coder infers uncharacterized blocks that are combinational in nature as zero-delay combinational blocks. The code generator treats other blocks as registers.
- If your target device does not have timing characteristics that are similar to one of the supported target devices, critical path estimation cannot accurately compute your critical path.

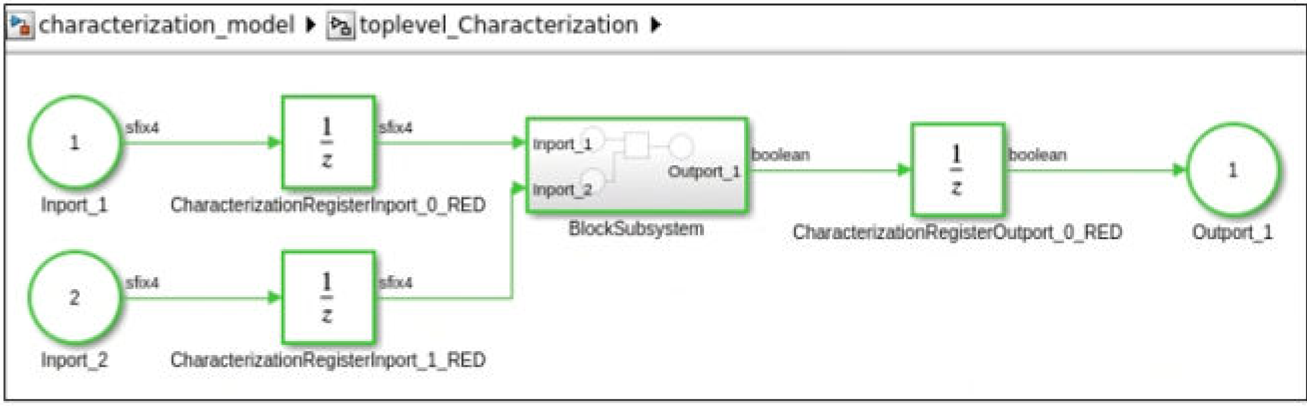
Generate Custom Timing Database for Custom Tools and Devices

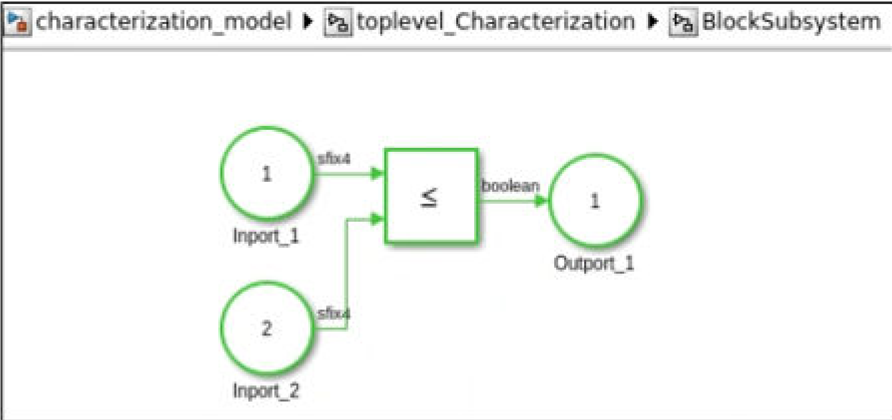
To generate a custom timing database for tools and devices, HDL Coder uses the characterization of basic design components, such as Simulink blocks, block architectures, and subcomponents of those blocks, for specific target devices and passes this information to a `hdlcoder.TimingGenerator` object. The `hdlcoder.TimingGenerator` object generates a timing information file with static timing information. The `genhdltdb` function uses the timing information file to generate a custom timing database.



You can generate a timing generator object by using the `hdlcoder.TimingGenerator` handle class. HDL Coder includes timing generator objects for Intel Quartus and Xilinx Vivado that you can use to create a custom timing generator for your tool or device. For more information, see `hdlcoder.TimingGenerator`.

The timing generator object outputs a comma-separated value (CSV) text file that contains the propagation delays in nanoseconds (ns). For example, suppose you use the model in this image





The timing generator outputs this CSV text file:

```
DelayInfo, mw_internal_registers, mw_internal_registers, inf
DelayInfo, mw_inport_0, mw_internal_registers, inf
DelayInfo, mw_inport_1, mw_internal_registers, inf
DelayInfo, mw_internal_registers, mw_outport_0, inf
DelayInfo, mw_inport_0, mw_outport_0, 0.494
DelayInfo, mw_inport_1, mw_outport_0, 0.451
```

The file contains information in this format: `DelayInfo, source_type, destination_type, propagation delay (ns)`.

The `source_type` or `destination_type` fields are of type:

- `mw_internal_registers` — Any internal register inside the subsystem that contains the block to be characterized.
- `mw_inport_#` — Numbered input to the operator starting at index zero. For example, `mw_inport_0`, `mw_inport_1`, and so on.
- `mw_outport_#` — Numbered output from the operator starting at index zero. For example, `mw_outport_0`, `mw_outport_1`, and so on.
- `propagation delay (ns)` — The value in nanoseconds for either internal-to-internal, input-to-internal, internal-to-output, or input-to-output propagation delay. If a timing path does not exist, this value is `inf`. This field contains the timing information for every path.

See Also

`hdlcoder.FloatingPointTargetConfig` | `makehdl` | `genhdltdb` | `hdlcoder.TimingGenerator`

More About

- “Create and Use Code Generation Reports” on page 23-2
- “Generated Model and Validation Model” on page 21-10
- “Distributed Pipelining” on page 8-14
- “Getting Started with HDL Coder Native Floating-Point Support” on page 14-88

HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture

This example shows how to use various optimizations inside the MATLAB Function block and across the MATLAB Function block boundary with other blocks in your Simulink® model. The example also illustrates the difference in area and timing when you use different HDL architecture settings of the MATLAB Function block.

Why Use MATLAB Datapath Architecture?

HDL code generation for a MATLAB Function block supports two HDL architectures: `MATLAB Function` and `MATLAB Datapath`. Specify the **HDL Architecture** in the HDL Block Properties dialog box of the MATLAB Function block.

Use the `MATLAB Datapath` architecture to:

- Model complex fixed-point and floating-point MATLAB® algorithms inside MATLAB Function blocks and interface this algorithm with other Simulink blocks in your model.
- Improve area and timing of your design significantly by optimizing the algorithm inside the MATLAB Function block and across the MATLAB Function block boundary with other Simulink blocks in your model.

The `MATLAB Datapath` architecture is the default setting for MATLAB Function blocks with floating-point types. By enabling this architecture for fixed-point operations, you can use various optimizations that include:

- Hierarchy flattening
- Resource sharing and streaming
- Clock-rate pipelining
- Adaptive pipelining
- Distributed pipelining
- Critical path estimation

How MATLAB Datapath Architecture Works

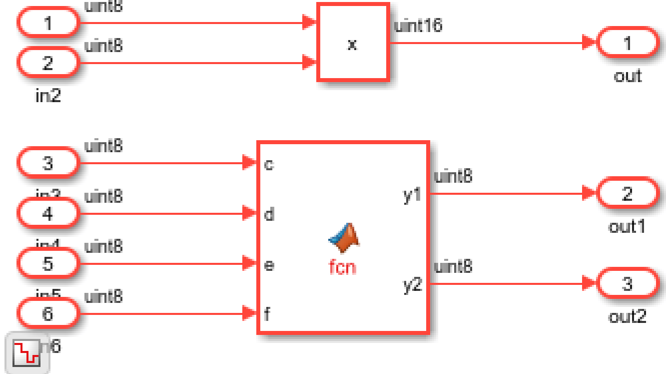
Fixed-point Simulink models use the `MATLAB Function` architecture by default. Certain HDL optimizations such as resource sharing and distributed pipelining that you enable with this architecture optimize the blocks surrounding the MATLAB Function block and the algorithm inside the MATLAB Function block. To see the effect of the optimizations inside the MATLAB Function block, examine the generated HDL code for the block. This architecture does not apply the optimizations across the MATLAB Function block boundary with other Simulink blocks.

Floating-point Simulink models use the `MATLAB Datapath` architecture even if you specify `MATLAB Function` as the architecture setting for the block. When you use floating-point types, specify the native floating-point mode. With this architecture, the code generator treats the block like a regular Subsystem block. HDL Coder transforms the control flow algorithm of the MATLAB code inside the MATLAB Function block to a dataflow representation that uses Simulink blocks. The `MATLAB Datapath` architecture unrolls loops in your code due to this transformation. If you want to stream loops, either use the loop streaming optimization with the `MATLAB Function` architecture or use the streaming optimization with `MATLAB Datapath` as the HDL architecture.

By using the MATLAB Datapath architecture, you can more effectively perform various HDL Coder optimizations with the MATLAB Function block that you would otherwise perform with a Subsystem block. The MATLAB Datapath architecture applies the optimization settings that you specify on the algorithm inside the MATLAB Function block and across the MATLAB Function block boundary with other blocks in your Simulink model.

For example, consider this model with a DUT Subsystem that consists of a Product block and a MATLAB Function block.

```
open_system('hdlcoder_MLFB_simple_datapath')
set_param('hdlcoder_MLFB_simple_datapath', 'SimulationCommand', 'Update')
open_system('hdlcoder_MLFB_simple_datapath/HDL_DUT')
```



The MATLAB Function block implements two multiplications.

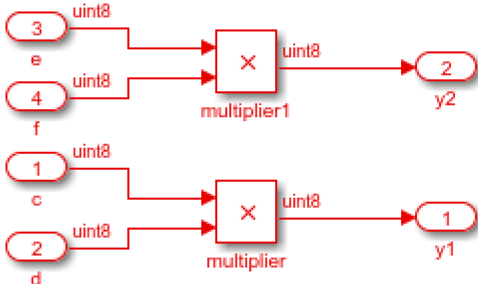
```
open_system('hdlcoder_MLFB_simple_datapath/HDL_DUT/MATLAB Function')
```

```
function [y1, y2] = fcn(c, d, e, f)
%#codegen

y1 = c * d;
y2 = e * f;
```

The HDL architecture of the MATLAB Function block is set to MATLAB Datapath. To generate HDL code for the HDL_DUT Subsystem, run this command:

```
makehdl('hdlcoder_MLFB_simple_datapath/HDL_DUT')
```

When you generate HDL code, the code generator replaces the MATLAB Function block with a subsystem that performs the multiplications c * d and e * f.



With the `MATLAB Datapath` architecture, you can perform optimizations inside the MATLAB Function block and across the MATLAB Function block with other Simulink blocks. In this example, you can share the two multipliers inside the MATLAB Function block. To optimize the blocks, set the **SharingFactor** to 2 on the MATLAB Function block.

```
mlsubsys = 'hdlcoder_MLFB_simple_datapath/HDL_DUT/MATLAB Function';
hdlset_param(mlsubsys, 'SharingFactor', 2)
```

When you generate HDL code, the code generator shares the multiplications inside the MATLAB Function block. The sharing group is displayed in the Optimization Report. When you click the links in the sharing group, HDL Coder displays the shared multipliers inside the MATLAB Function block in the generated model and the original model.

Sharing Report

Subsystem: *MATLAB Function*

SharingFactor: 2

Highlight shared resources and diagnostics

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	8x8 -> 16	2	multiplier	

You can apply the optimization across the MATLAB Function block with other Simulink blocks. In this example, you can share the Product block outside the MATLAB Function block with the multipliers inside the MATLAB Function block. To share these resources, remove the **SharingFactor** on the MATLAB Function block, and on the parent subsystem, `HDL_DUT`, enable **FlattenHierarchy** and set **SharingFactor** to 3.

```
hdlset_param(mlsubsys, 'SharingFactor', 0)
hdlset_param('hdlcoder_MLFB_simple_datapath/HDL_DUT', ...
                    'FlattenHierarchy', 'on', 'SharingFactor', 3)
```

Note: Do not use the `InlineMATLABCode` property with the `MATLAB Datapath` architecture of the block. Use `FlattenHierarchy` instead.

When you generate HDL code, the code generator shares the multiplications inside the MATLAB Function block with the Product block outside. You see the sharing group of three multipliers in the

Optimization Report. When you click the links in the sharing group, the shared multipliers are highlighted in the generated model and the original model.

Sharing Report

Subsystem: HDL_DUT

SharingFactor: 3

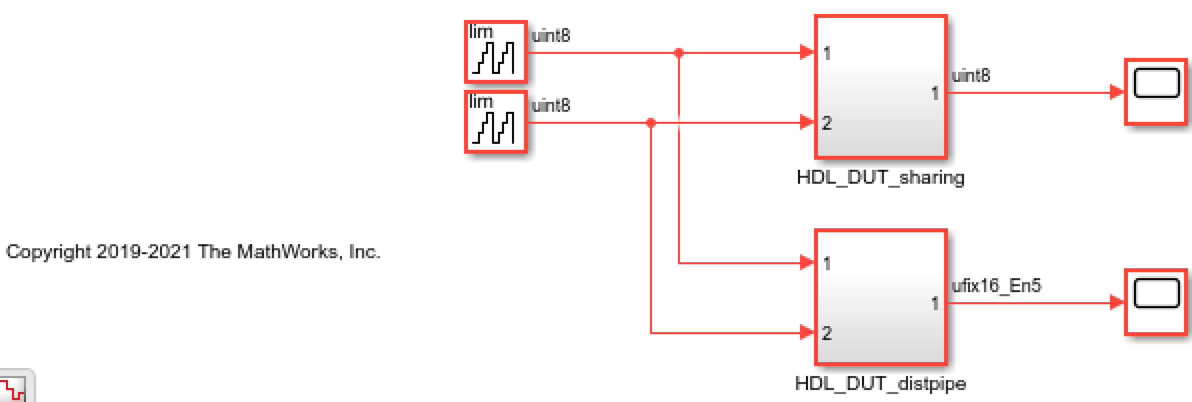
Highlight shared resources and diagnostics

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	8x8 -> 16	3	Product	

MATLAB Function Block Model with Default MATLAB Function Architecture

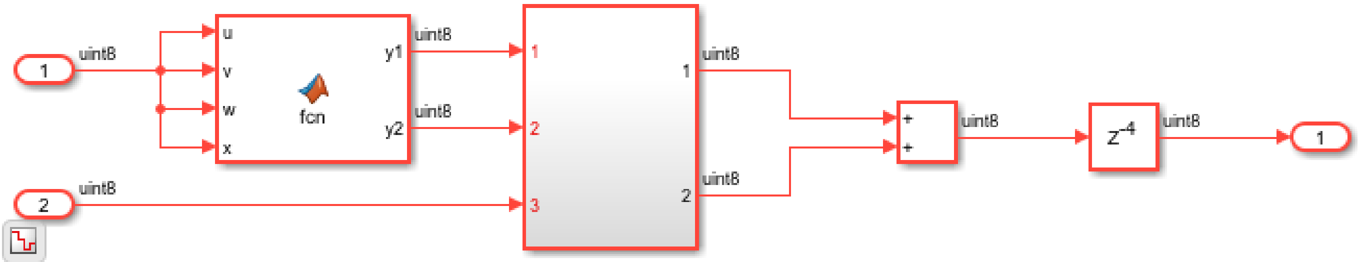
For an example model that illustrates the `MATLAB Datapath` architecture and how it differs from the `MATLAB Function` architecture, open the model `hdlcoder_MLFB_share_pipeline`. The model uses integer types. For an example that illustrates how you use the `MATLAB Datapath` architecture with floating-point types, see “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111.

```
open_system('hdlcoder_MLFB_share_pipeline')
set_param('hdlcoder_MLFB_share_pipeline','SimulationCommand','Update')
```

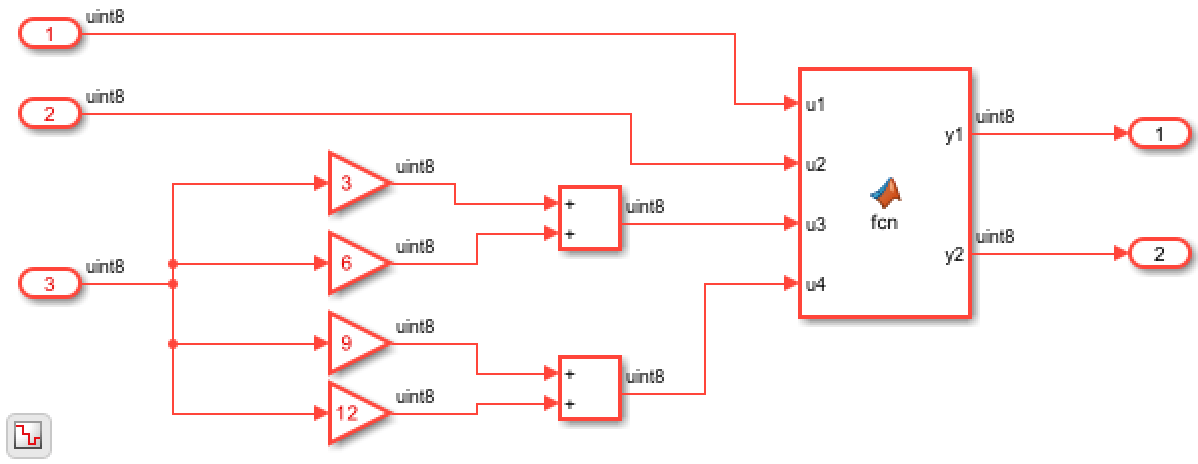


The model contains two DUT subsystems at the top level, `HDL_DUT_sharing` and `HDL_DUT_distpipe`. The subsystems illustrate how you can use resource sharing and distributed pipelining optimizations across the MATLAB Function block boundary with other blocks. Both subsystems perform basic additions and multiplications inside and outside the MATLAB Function block.

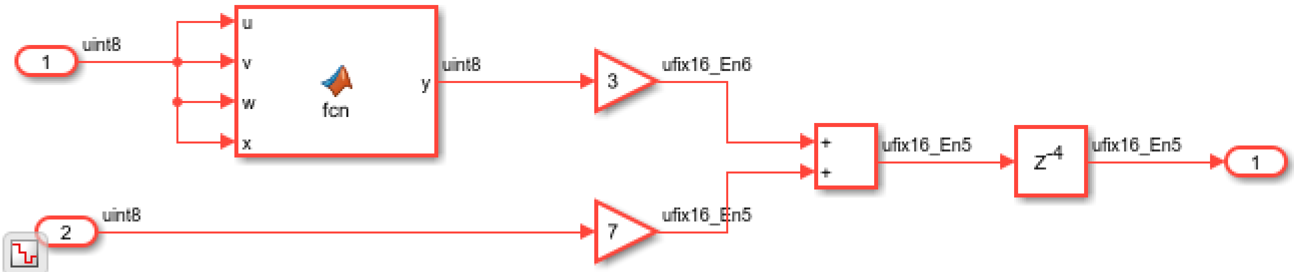
```
open_system('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing')
```




```
open_system('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing/Subsystem')
```




```
open_system('hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe')
```



To see the HDL parameters that are saved on the model, run the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_MLFB_share_pipeline')
```

```
%% Set Model 'hdlcoder_MLFB_share_pipeline' HDL parameters
hdlset_param('hdlcoder_MLFB_share_pipeline', 'CriticalPathEstimation', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'HDLSubsystem', 'hdlcoder_MLFB_share_pipeline/HDL_D
hdlset_param('hdlcoder_MLFB_share_pipeline', 'Oversampling', 40);
hdlset_param('hdlcoder_MLFB_share_pipeline', 'ShareAdders', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe', 'DistributedPipelining', 'on');
```

```
hdlset_param('hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe', 'FlattenHierarchy', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing', 'FlattenHierarchy', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing', 'SharingFactor', 8);
```

You see that the default `MATLAB Function` HDL architecture is saved for the MATLAB Function blocks in this model.

Generate HDL Code Using MATLAB Function Architecture

To generate HDL code for the sharing DUT, run this command:

```
makehdl('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing')
```

When you open the Streaming and Sharing Report, the report displays four multipliers and three adders as shared resources.

Sharing Report

Subsystem: *HDL_DUT_sharing*

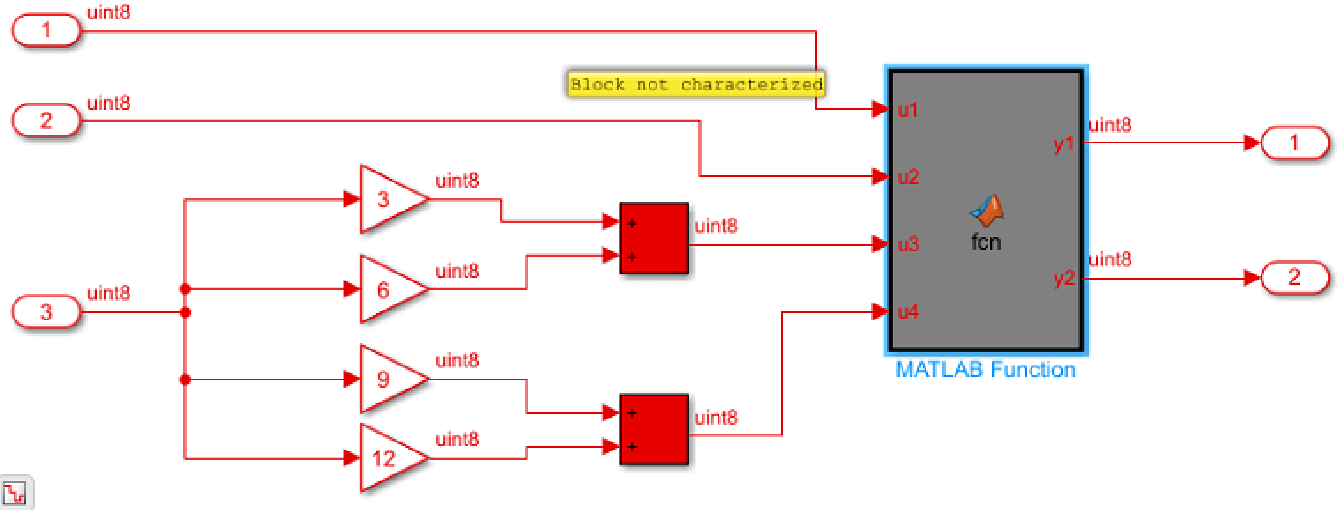
SharingFactor: 8

Highlight shared resources and diagnostics

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	8x8 -> 16	4	Gain2	
2	Sum	8+8 -> 8	3	Add1	

When you click the second sharing group, the code generator highlights three adders surrounding the MATLAB Function block. The sharing group includes the two adders inside the Subsystem and the Add block outside. The code generator did not share the multipliers and adders that are inside the MATLAB Function block.

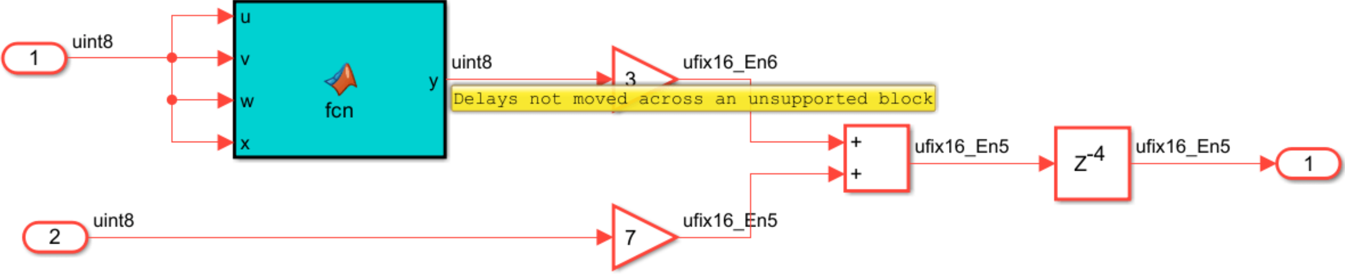
Critical path estimation is enabled on the model. When you annotate the critical path, the MATLAB Function block acts as a barrier to this optimization. If the critical path is inside the MATLAB Function block and if you want to highlight the critical path, use the `MATLAB Datapath` architecture.



To generate HDL code for `HDL_DUT_distpipe`, run this command:

```
makehdl('hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe')
```

When you open the Distributed Pipelining Report and click on the "Highlight blocks inhibiting distributed pipelining" link, you see that the code generator moved pipelines inside the `HDL_DUT_distpipe` subsystem but did not distribute pipelines inside the MATLAB Function block.



Apply Optimizations Across MATLAB Function Block and Other Simulink Blocks

To improve area and timing of your design, use the `MATLAB Datapath` architecture. For the `HDL_DUT_sharing` subsystem, you can combine resource sharing with clock-rate pipelining and share the resources inside the MATLAB Function block and across the MATLAB Function block with other blocks.

To share resources:

1. Enable **FlattenHierarchy** and specify a **SharingFactor** on the parent subsystem `HDL_DUT_sharing`. Set the **SharingFactor** to 8.

```
share_subsys = 'hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing';
hdlset_param(share_subsys, 'FlattenHierarchy', 'on', 'SharingFactor', 8);
```

2. Specify the `MATLAB Datapath` architecture for the MATLAB Function blocks inside the `HDL_DUT_sharing` subsystem.

```
share_mlfcn1 = 'hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing/MATLAB Function';
share_mlfcn2 = 'hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing/Subsystem/MATLAB Function';
hdlset_param(share_mlfcn1, 'architecture', 'MATLAB Datapath');
hdlset_param(share_mlfcn2, 'architecture', 'MATLAB Datapath');
```

To generate HDL code for `HDL_DUT_distpipe`, run this command:

```
makehdl('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing')
```

When you open the Streaming and Sharing report, the report displays a sharing group of eight multipliers and two sharing groups of adders.

Sharing Report

Subsystem: *HDL_DUT_sharing*

SharingFactor: 8

Highlight shared resources and diagnostics

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	9x9 -> 17	8	multiplier	
2	Sum	8+8 -> 8	4	adder	
3	Sum	8+8 -> 8	2	Add1	

When you select the first sharing group, you see that the optimization shared the multipliers inside the MATLAB Function with the four Gain blocks outside. The second sharing group consists of adders inside each of the two MATLAB Function blocks.

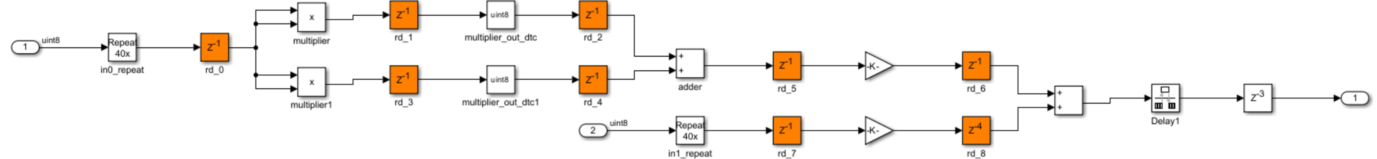
You can use the distributed pipelining optimization with the `HDL_DUT_distpipe` subsystem. Enable distributed pipelining at the top level and set the HDL architecture of the MATLAB Function to `MATLAB Datapath` with **DistributedPipelining** set to `on`.

```
dist_subsys = 'hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe/MATLAB Function';
hdlset_param(dist_subsys, 'architecture', 'MATLAB Datapath');
hdlset_param(dist_subsys, 'DistributedPipelining', 'on');
```

Generate HDL code for the `HDL_DUT_distpipe`:

```
makehdl('hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe')
```

After you generate HDL code, open the generated model. HDL Coder uses distributed pipelining to move the pipeline registers across the blocks.



See Also

Blocks
MATLAB Function

Functions
`makehdl` | `hdlsaveparams`

More About

- "Design Guidelines for the MATLAB Function Block" on page 27-19
- "Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks" on page 27-28
- "RAM Mapping with the MATLAB Function Block" on page 21-125
- Functions Supported for HDL Code Generation (Category List)
- Functions Supported for HDL Code Generation (Alphabetical List)

Subsystem Optimizations for Filters

The Discrete FIR Filter (when used with scalar or multichannel input data) and Biquad Filter blocks participate in subsystem-level optimizations. To set optimization properties, right-click on the subsystem and open the **HDL Properties** dialog box.

For these blocks to participate in subsystem-level optimizations, you must leave the block-level **Architecture** set to the default, `Fully parallel`.

You cannot use these subsystem optimizations when using the Discrete FIR Filter in frame-based input mode.

Sharing

These filter blocks support sharing resources within the filter and across multiple blocks in the subsystem. When you specify a **SharingFactor**, the optimization tools generate a filter implementation in HDL that shares resources using time-multiplexing. To generate an HDL implementation that uses the minimum number of multipliers, set the **SharingFactor** to a number greater than or equal to the total number of multipliers. The sharing algorithm shares multipliers that have the same input and output data types. To enable sharing between blocks, you may need to customize the internal data types of the filters. Alternatively, you can target a particular system clock rate with your choice of **SharingFactor**.

Resource sharing applies to multipliers by default. To share adders, select the check box under **Resource sharing** on the **Configuration Parameters > HDL Code Generation > Global Settings > Optimizations** dialog box.

For more information, see “Resource Sharing” on page 21-45 and the “Area Reduction of Multichannel Filter Subsystem” on page 21-215 example.

You can also use a **SharingFactor** with multichannel filters. See “Area Reduction of Filter Subsystem” on page 21-220.

Streaming

Streaming refers to sharing an atomic part of the design across multiple channels. To generate a streaming HDL implementation of a multichannel subsystem, set **StreamingFactor** to the number of channels in your design.

If the subsystem contains a single filter block, the block-level **ChannelSharing** option and the subsystem-level **StreamingFactor** option result in similar HDL implementations. Use **StreamingFactor** when your subsystem contains either more than one filter block or additional multichannel logic that can participate in the optimization. You must set block-level **ChannelSharing** to `off` to use **StreamingFactor** at the subsystem level.

See “Streaming” on page 21-42 and the “Area Reduction of Filter Subsystem” on page 21-220 example.

Pipelining

You can enable **DistributedPipelining** at the subsystem level to allow the filter to participate in pipeline optimizations. The optimization tools operate on the **InputPipeline** and **OutputPipeline**

pipeline stages specified at subsystem level. The optimization tools also operate on these block-level pipeline stages:

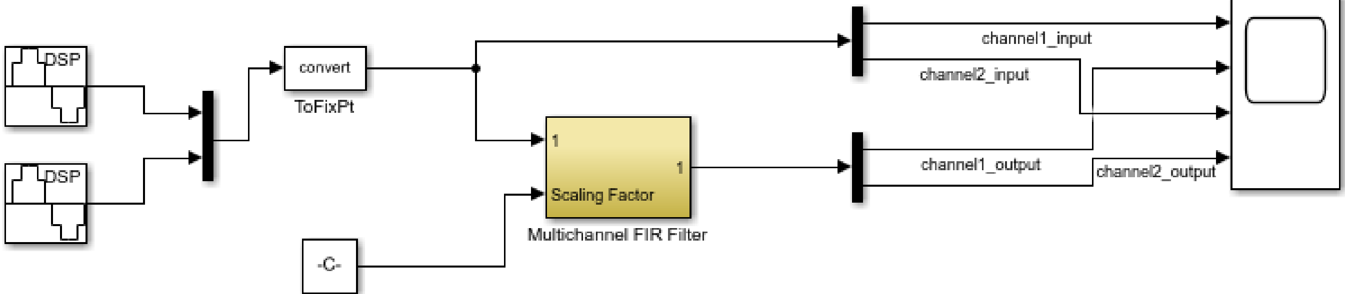
- **InputPipeline** and **OutputPipeline**
- **MultiplierInputPipeline** and **MultiplierOutputPipeline**
- **AddPipelineRegisters**

The optimization tools do not move design delays within the filter architecture. See “Distributed Pipelining” on page 8-14.

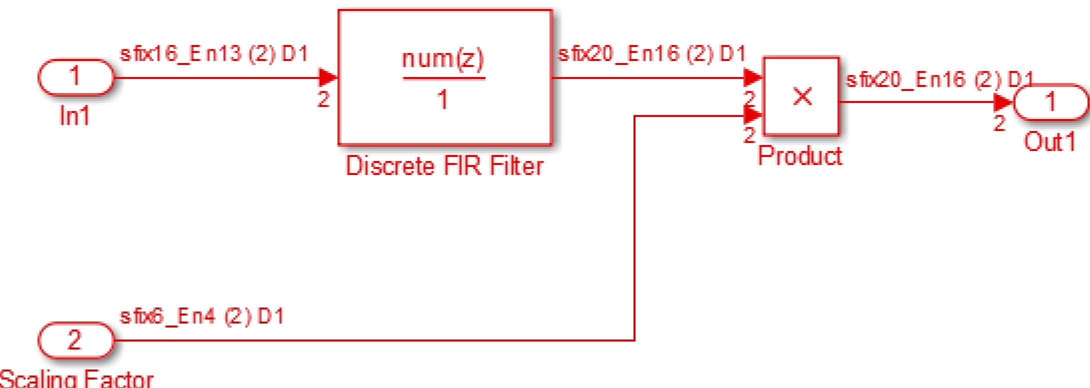
The filter block also participates in clock-rate pipelining, if enabled in **Configuration Parameters**. This feature is enabled by default. See “Clock-Rate Pipelining” on page 21-148.

Area Reduction of Multichannel Filter Subsystem

To reduce the number of multipliers in the HDL implementation of a multichannel filter and surrounding logic, use the **StreamingFactor** HDL Coder™ optimization.

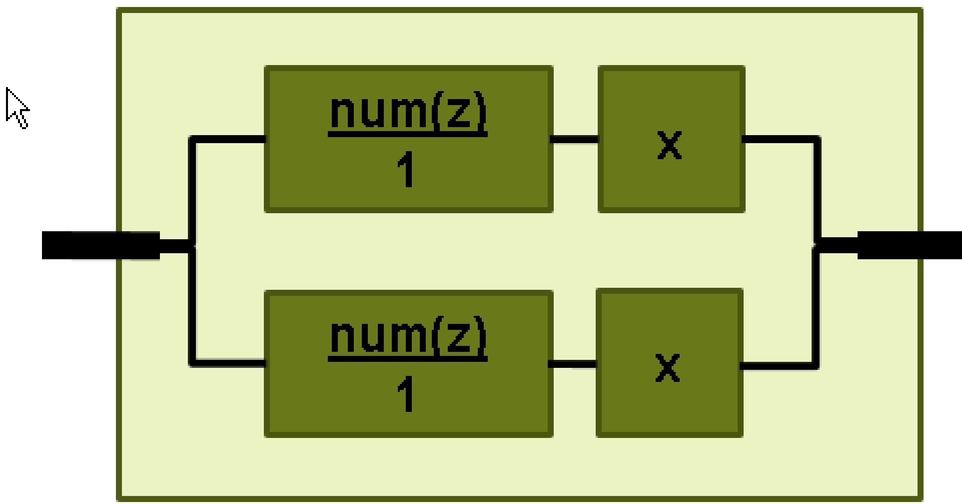


The model includes a two-channel sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

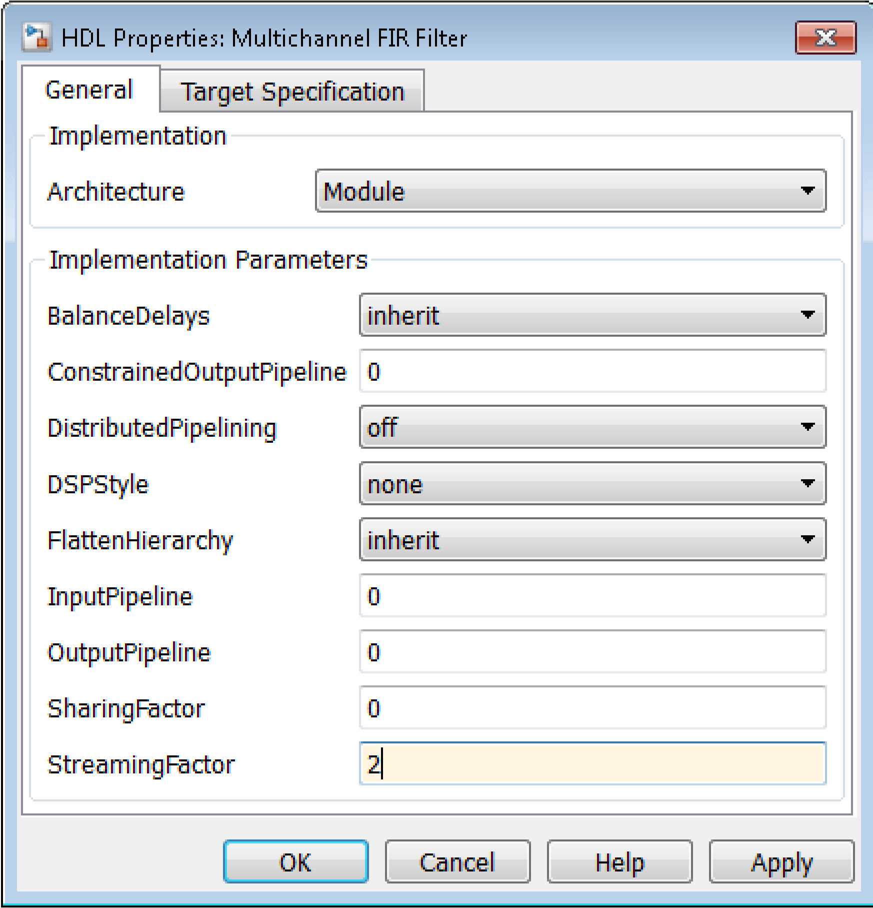


The subsystem contains a Discrete FIR Filter block and a constant multiplier. The multiplier is included to show the optimizations operating over all eligible logic in a subsystem.

The filter has 44 symmetric coefficients. With no optimizations enabled, the generated HDL code takes advantage of symmetry. The nonoptimized HDL implementation uses 46 multipliers: 22 for each channel of the filter and 1 for each channel of the Product block.



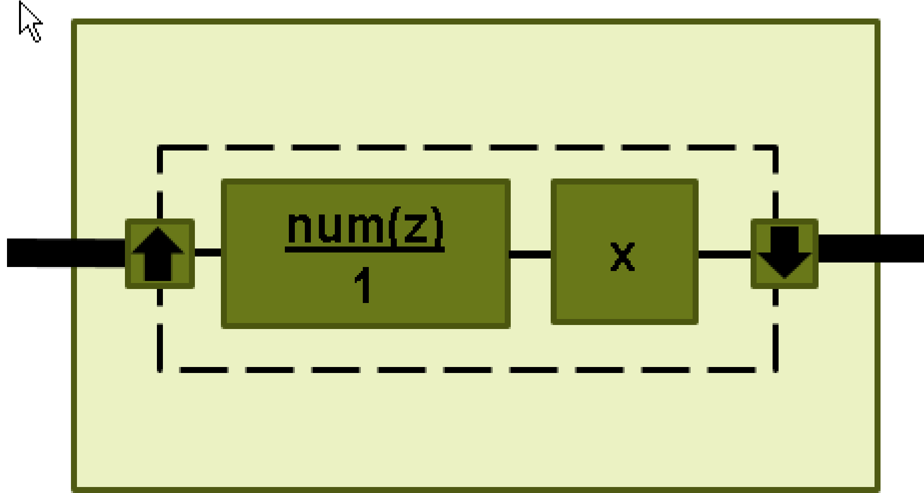
To enable streaming optimization for the Multichannel FIR Filter Subsystem, right-click the subsystem and select **HDL Code > HDL Block Properties**.



Set the **StreamingFactor** to 2, because this design is a two-channel system.

To observe the effect of the optimization, under **Configuration Parameters > HDL Code Generation**, select **Generate resource utilization report** and **Generate optimization report**. Then, to generate HDL code, right-click the Multichannel FIR Filter Subsystem and select **HDL Code > Generate HDL for Subsystem**.

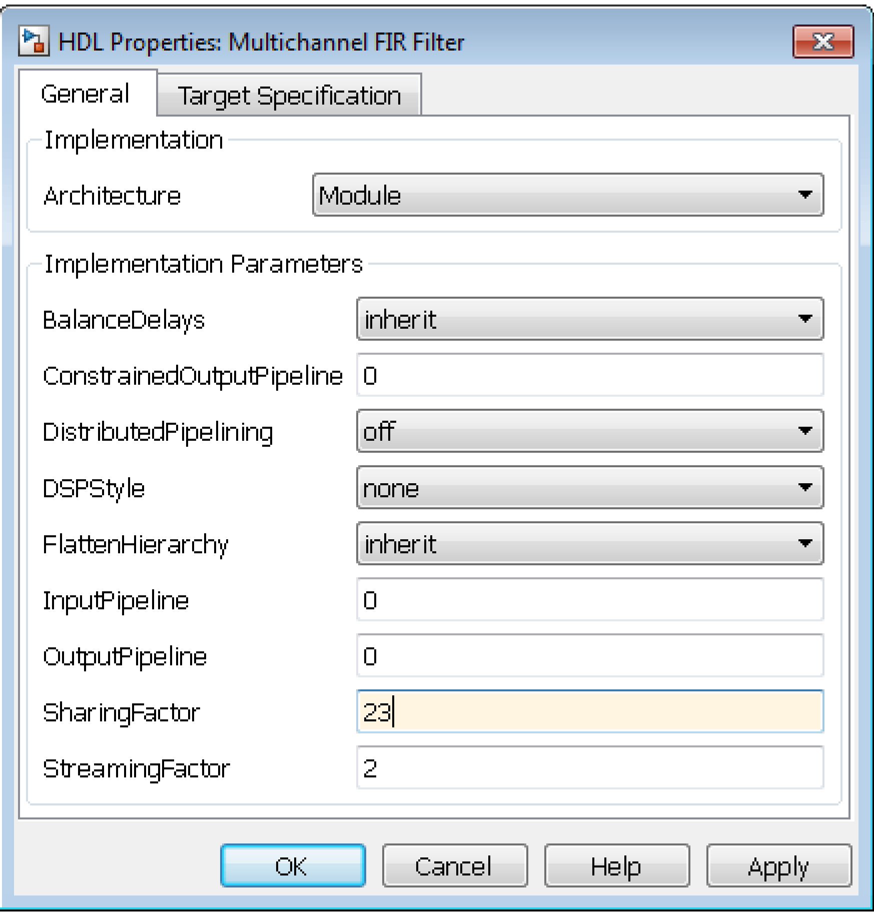
With the streaming factor applied, the logic for one channel is instantiated once and run at twice the rate of the original model.



In the **Code Generation Report** window, click **High-level Resource Report**. The generated HDL code now uses 23 multipliers, compared to 46 in the nonoptimized code. The multipliers in the filter kernel and subsequent scaling are shared between the channels.

Multipliers	23
Adders/Subtractors	44
Registers	92
RAMs	0
Multiplexers	28

To apply **SharingFactor** to multichannel filters, set the **SharingFactor** to 23.



The optimized HDL now uses only 2 multipliers. The optimization tools do not share multipliers of different sizes.

Summary

Multipliers	2
Adders/Subtractors	47
Registers	166
Total 1-Bit Registers	3566
RAMs	0
Multiplexers	37
I/O Bits	80

Detailed Report

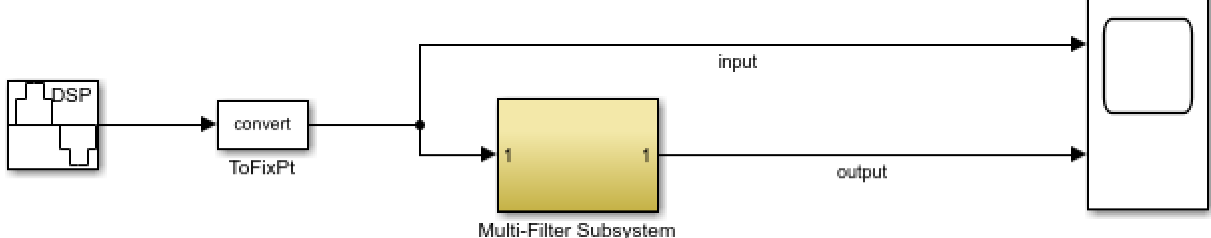
Report for Subsystem: *Multichannel FIR Filter*

Multipliers (2)

```
16x16-bit Multiply : 1
[+] 16x6-bit Multiply : 1
```

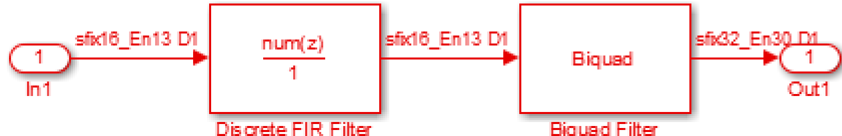
Area Reduction of Filter Subsystem

To reduce the number of multipliers in the HDL implementation of a multifilter design, use the **SharingFactor** HDL Coder™ optimization.



Copyright 2016 The MathWorks, Inc.

The model includes a sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

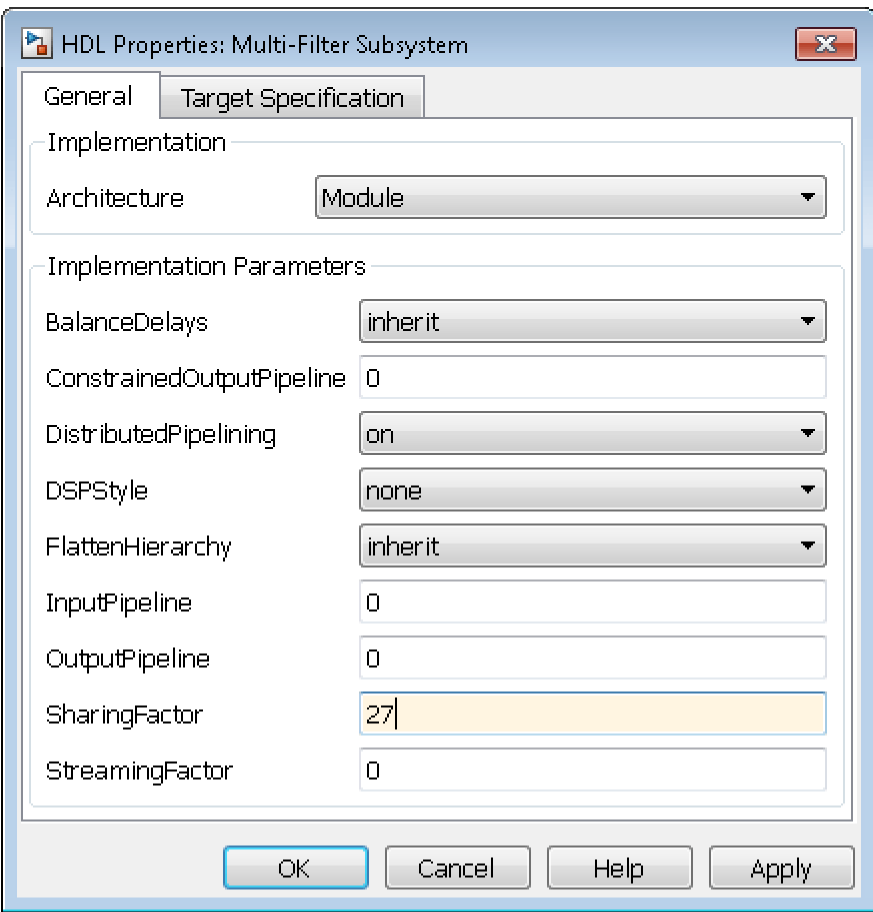


The subsystem contains a Discrete FIR Filter block and a Biquad Filter block. This design demonstrates how the optimization tools share resources between multiple filter blocks.

The Discrete FIR Filter block has 43 symmetric coefficients. The Biquad Filter block has 6 coefficients, two of which are unity. With no optimizations enabled, the generated HDL code takes advantage of symmetry and unity coefficients. The nonoptimized HDL implementation of the subsystem uses 27 multipliers.

Multipliers	27
Adders/Subtractors	46
Registers	49
Total 1-Bit Registers	800
RAMs	0
Multiplexers	0
I/O Bits	52

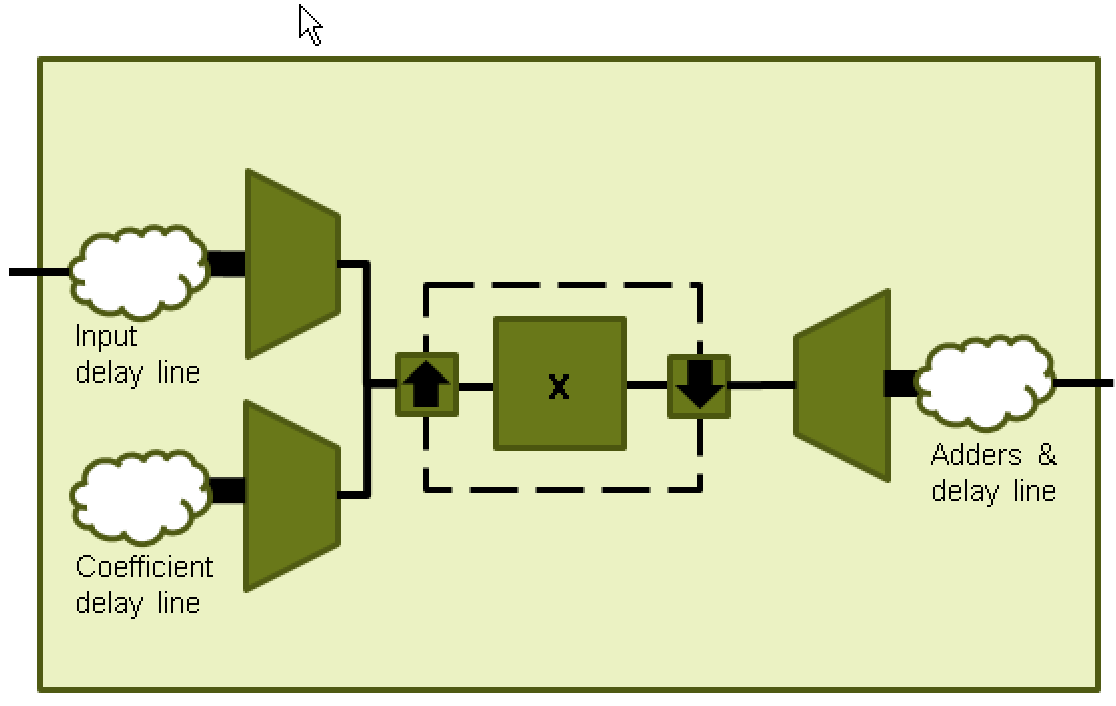
To enable streaming optimization for the **Multi-Filter Subsystem**, right-click the subsystem and select **HDL Code > HDL Block Properties**.



Set the **SharingFactor** to 27 to reduce the design to a single multiplier. The optimization tools attempt to share multipliers with matching data types. To reduce to a single multiplier, you must set the internal data types of the filter blocks to match each other.

To observe the effect of the optimization, under **Configuration Parameters** > **HDL Code Generation**, select **Generate resource utilization report** and **Generate optimization report**. Then, to generate HDL code, right-click the Multi-Filter Subsystem and select **HDL Code** > **Generate HDL for Subsystem**.

With the **SharingFactor** applied, the subsystem upsamples the rate by 27 to share a single multiplier for all the coefficients.



In the **Code Generation Report** window, click **High-level Resource Report**. The generated HDL code now uses one multiplier.

Multipliers	1
Adders/Subtractors	48
Registers	102
Total 1-Bit Registers	2457
RAMs	0
Multiplexers	7
I/O Bits	52

See Also

More About

- "Resource Sharing" on page 21-45
- "Streaming" on page 21-42
- "Clock-Rate Pipelining" on page 21-148

Remove Redundant Logic and Unused Blocks in Generated HDL Code

If your design contains redundant logic or unused blocks, HDL Coder™ removes the blocks, components, or part of the HDL code that does not contribute to the output.

Redundant Logic Considerations

Components that do not contribute to the output in the design are removed during HDL code generation. Removing redundant logic reduces code size and avoids potential synthesis failures with downstream tools when deploying your code onto a target platform. If a component or logic is preserved during HDL code generation, that component or logic is considered *active*. This optimization improves the performance of your design on the target hardware. The optimization does not affect the traceability support.

Redundant logic in your design is removed in conjunction with unused port deletion optimization. To learn about this optimization, see “Optimize Unconnected Ports in HDL Code for Simulink Models” on page 21-246.

HDL Coder does not treat a component or logic as redundant in these cases:

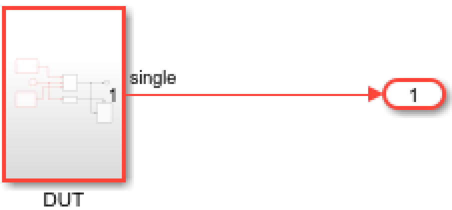
- The component has at least one output port that contributes to the evaluation of the DUT output.
- The component has at least one output port that contributes to the evaluation of the control port of a Subsystem block that is active.
- The component has at least one output port that contributes to the evaluation of input of a component that is preserved during HDL code generation.
- The component is an active black box subsystem, or contains an active black box subsystem, or is connected to an active black box subsystem, as described below.
- The component is an FPGA Data Capture block.

In other cases, the code generator treats the logic as redundant and removes the associated blocks or components during code generation. In addition, blocks that have HDL architecture set to `No HDL`, such as Scope, Assertion, Terminator, and To Workspace blocks are considered redundant, and are removed during code generation.

How Redundant Logic Removal Works

During code generation, HDL Coder removes the redundant logic or blocks that do not contribute to the DUT output. Open the model `hdlcoder_remove_redundant_logic`.

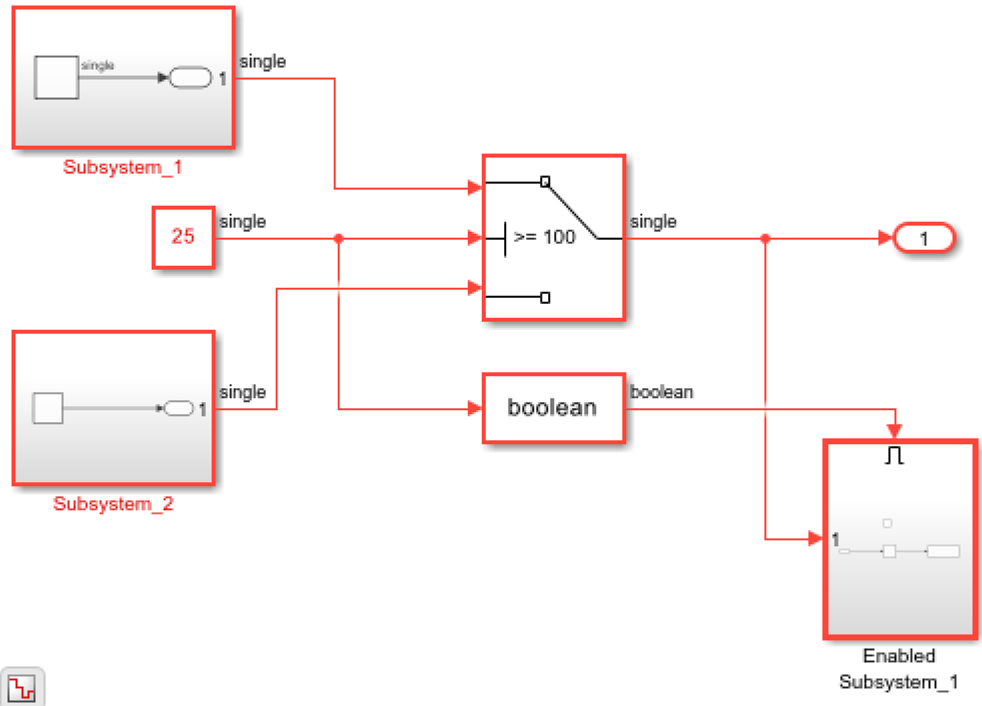
```
open_system('hdlcoder_remove_redundant_logic')
set_param('hdlcoder_remove_redundant_logic', 'SimulationCommand', 'update');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem block contains a Switch block and an Enabled Subsystem block. Based on the control input to the Switch block, the false path from Subsystem_2 is passed to the output. The EnabledSubsystem_1 block output is terminated by a Display block and does not actively contribute to the output.

```
open_system('hdlcoder_remove_redundant_logic/DUT')
```



To generate HDL code for the design, at the MATLAB® command prompt, enter:

```
makehdl('hdlcoder_remove_redundant_logic/DUT')
```

The generated VHDL® code shows that HDL Coder evaluated the Switch block condition at compile time to pass the input from Subsystem_2 to the output, and eliminated Subsystem_1 input branch. The EnabledSubsystem_1 block is removed during HDL code generation since it does not have an active output.

```
ARCHITECTURE rtl OF DUT IS
```

```
-- Component Declarations
COMPONENT Subsystem_2
  PORT( Out1        :   OUT   std_logic_vector(31 DOWNTO 0)  -- single
        );
END COMPONENT;

-- Component Configuration Statements
FOR ALL : Subsystem_2
  USE ENTITY work.Subsystem_2(rtl);

-- Signals
SIGNAL Subsystem_2_out1    : std_logic_vector(31 DOWNTO 0);  -- ufix32

BEGIN
 u_Subsystem_2 : Subsystem_2
   PORT MAP( Out1 => Subsystem_2_out1  -- single
             );

Out1 <= Subsystem_2_out1;

END rtl;
```

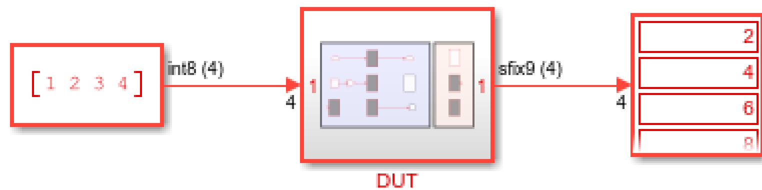
Redundant Logic in Black Box Subsystems

Black box subsystems are subsystem blocks that have HDL architecture set to `BlackBox`. A black box interface for a subsystem is the generated VHDL component or Verilog® or SystemVerilog module that includes only the HDL input and output port definitions for the subsystem. Use the generated interface to integrate existing manually written HDL code, third-party IP, or other code generated by HDL Coder. See “Generate Black Box Interface for Subsystem” on page 25-4.

Black box subsystems that have at least one input port are considered valid and preserved during HDL code generation. The input port can be unconnected. In this case, Simulink® inserts a virtual signal that has a constant zero value as input to the block. In the case of output ports, the black box subsystems must have at least one output port that is connected to a downstream block. When the output port is connected, the code generator identifies the black box subsystem as a source that contributes to the output value computation, and preserves it during code generation.

Open the model `hdlcoder_blackbox_redundant_logic`.

```
open_system('hdlcoder_blackbox_redundant_logic')
sim('hdlcoder_blackbox_redundant_logic');
```

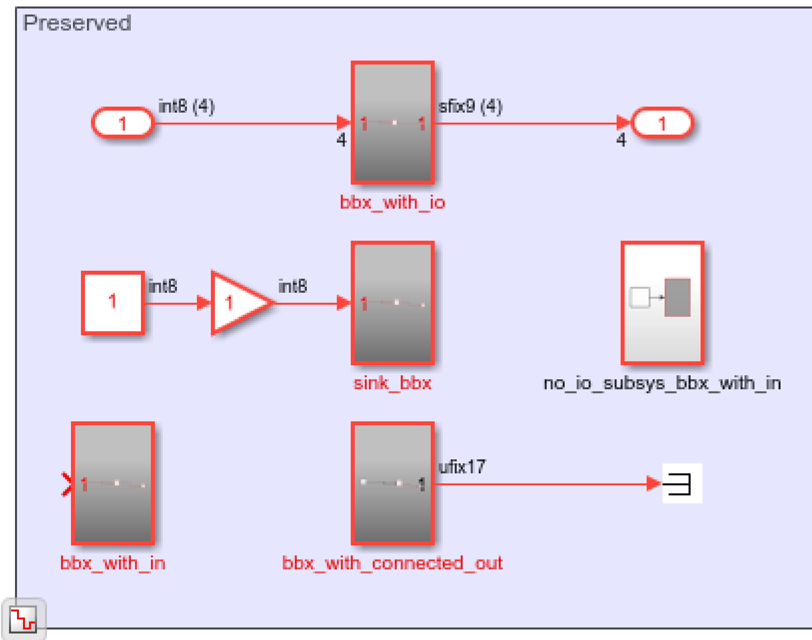


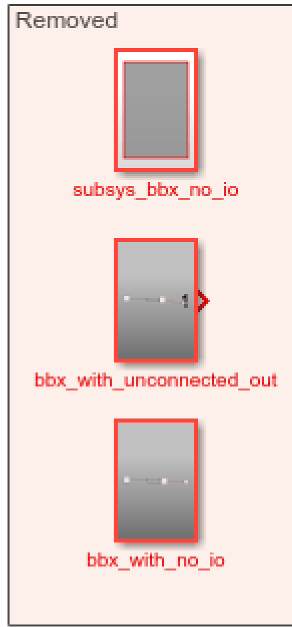
Copyright 2020 The MathWorks, Inc.

The `DUT` subsystem contains black box subsystems inside boxes labeled `Preserved` and `Removed`. Black box subsystems inside `Preserved` are not removed during code generation because they have

at least one input port. The other black box subsystems that do not have an input port or have an unconnected output port are removed during code generation.

```
open_system('hdlcoder_blackbox_redundant_logic/DUT')
```





To generate HDL code for the `DUT` subsystem, run this command:

```
makehdl('hdlcoder_blackbox_redundant_logic/DUT')
```

The generated HDL code shows the subsystems in the `Preserved` section. The unconnected input port is automatically connected to a constant zero.

```
ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT bbx_with_io
  PORT( clk             :   IN    std_logic;
        clk_enable      :   IN    std_logic;
        reset           :   IN    std_logic;
        In1             :   IN    vector_of_std_logic_vector8(0 TO 3);  -- int8 [4]
        Out1            :   OUT   vector_of_std_logic_vector9(0 TO 3)  -- sfix9 [4]
        );
END COMPONENT;

COMPONENT no_io_subsys_bbx_with_in
  PORT( clk             :   IN    std_logic;
        reset           :   IN    std_logic;
        enb             :   IN    std_logic
        );
END COMPONENT;

COMPONENT bbx_with_in
  PORT( clk             :   IN    std_logic;
        clk_enable      :   IN    std_logic;
```

```
        reset         :   IN    std_logic;
        In1           :   IN    std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;

COMPONENT sink_bbx
  PORT( clk           :   IN    std_logic;
        clk_enable    :   IN    std_logic;
        reset         :   IN    std_logic;
        In1           :   IN    std_logic_vector(7 DOWNTO 0)  -- int8
        );
END COMPONENT;

COMPONENT bbx_with_connected_out
  PORT( clk           :   IN    std_logic;
        clk_enable    :   IN    std_logic;
        reset         :   IN    std_logic;
        Output        :   OUT   std_logic_vector(16 DOWNTO 0)  -- ufix17
        );
END COMPONENT;


...

END rtl;
```

Redundant Logic in Subsystem Blocks

Subsystem blocks are preserved in the generated HDL code if the blocks have at least one output port that contributes to the evaluation of a DUT output. The Subsystem blocks are also preserved if they contain a black box subsystem that is valid. A black box subsystem is valid if it contains an input port, or an output port that is connected, as described above.

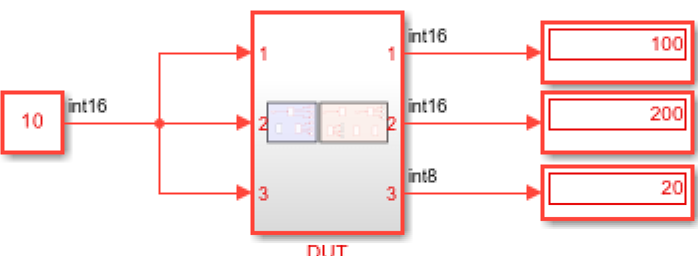
The different kinds of subsystem blocks follow this convention.

- Subsystem
- Atomic Subsystem
- Model References
- Variant Subsystem
- ForEach Subsystem
- Triggered Subsystem
- Enabled Subsystem
- Synchronous subsystems
- Masked subsystems

For individual subsystem ports, the removal of redundant logic also varies depending on whether you specify the Remove Unused Ports setting in the Configuration Parameters dialog box.

Open the model `hdlcoder_subsys_redundant_logic`.

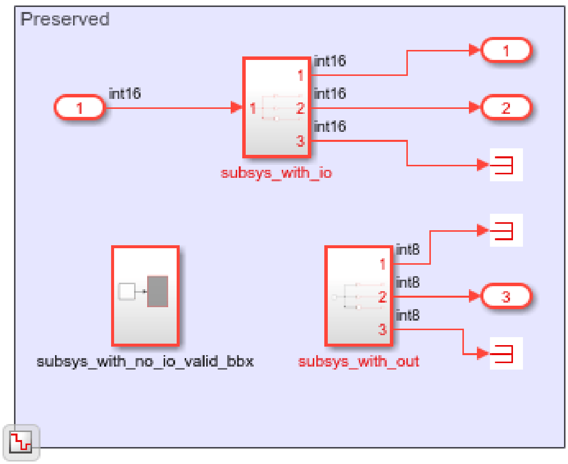
```
open_system('hdlcoder_subsys_redundant_logic')
sim('hdlcoder_subsys_redundant_logic');
```

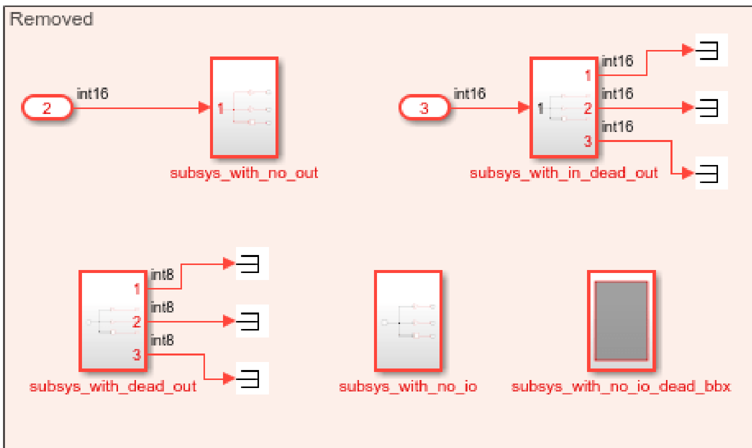


Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains subsystem blocks inside boxes labeled `Preserved` and `Removed`. Subsystem blocks inside `Preserved` are not removed during code generation because they have at least one output port that contributes to the evaluation of the DUT output, or have a valid black box subsystem. The other black box subsystems that do not have an active output port are removed during code generation.

```
open_system('hdlcoder_subsys_redundant_logic/DUT')
```





To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_subsys_redundant_logic/DUT')
```

The generated HDL code shows the subsystems in the `Preserved` section.

```
ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT subsys_with_io
  PORT( In1         :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1        :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2        :   OUT   std_logic_vector(15 DOWNTO 0)   -- int16
        );
END COMPONENT;
```

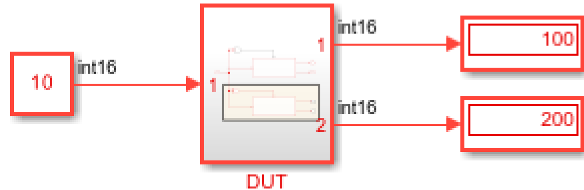
```
COMPONENT subsys_with_out
  PORT( Out2        :   OUT   std_logic_vector(7 DOWNTO 0)  -- int8
        );
END COMPONENT;

COMPONENT subsys_with_no_io_valid_bbx
  PORT( clk         :   IN    std_logic;
        reset       :   IN    std_logic;
        enb         :   IN    std_logic
        );
END COMPONENT;

...

END rtl;
```

Redundant Logic in Subsystem Ports

For subsystem data ports, the removal of redundant logic also depends on whether you specify the **Remove Unused Ports** setting. See “Optimize Unconnected Ports in HDL Code for Simulink Models” on page 21-246.

Control ports are not affected by the **Remove Unused Ports** setting. The control ports and components that contribute to evaluation of the control ports are preserved in the generated HDL code only if the entire subsystem instance is considered active.

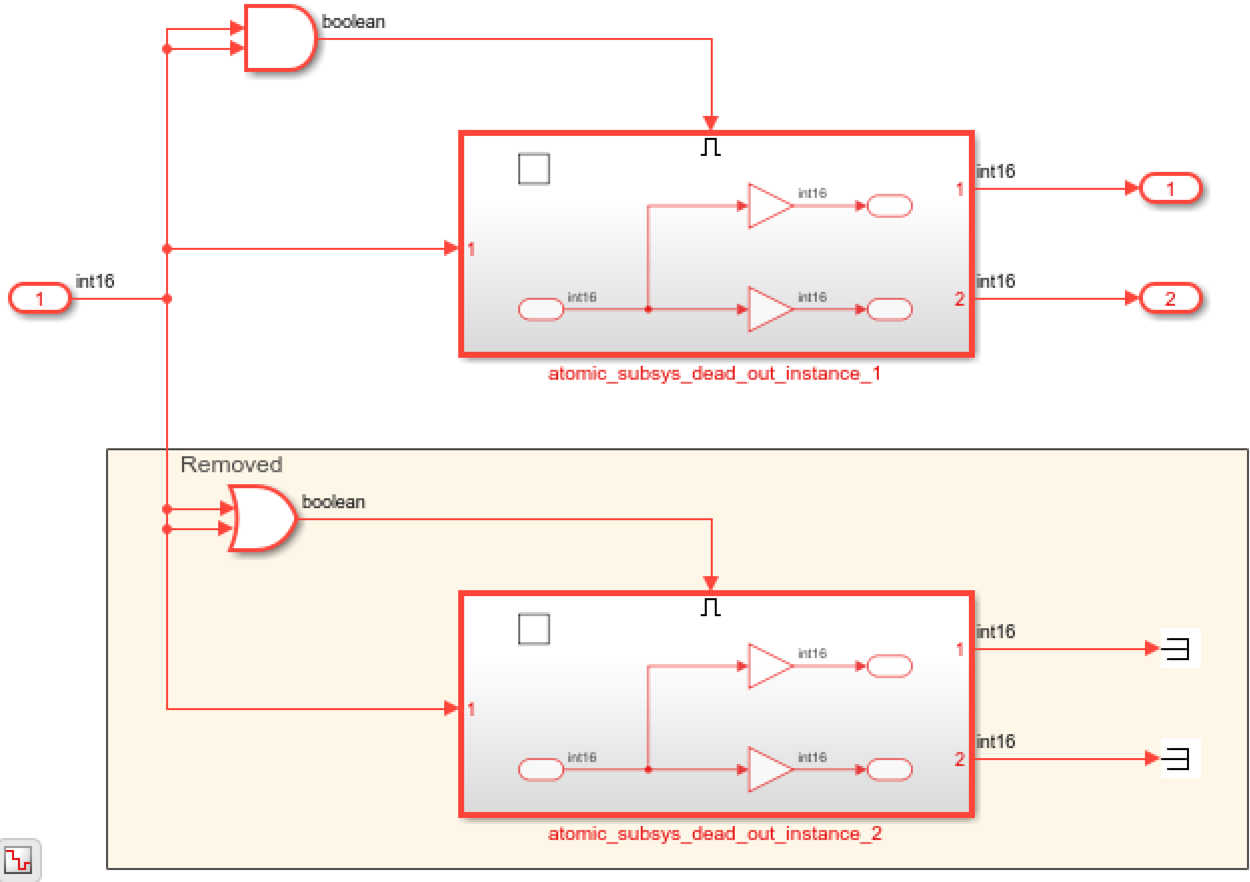
```
open_system('hdlcoder_control_redundant_logic')
sim('hdlcoder_control_redundant_logic');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains two atomic subsystems with an input that drives the Enable port. The subsystem `atomic_subsys_dead_out_instance_2` is not active as the outputs are terminated.

```
open_system('hdlcoder_control_redundant_logic/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_control_redundant_logic/DUT')
```

The atomic_subsys_dead_out_instance_2 including the control port and input signal is removed in the generated HDL code.

```
ENTITY DUT IS
 PORT( clk            :   IN    std_logic;
       reset          :   IN    std_logic;
       clk_enable     :   IN    std_logic;
       In1            :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       ce_out         :   OUT   std_logic;
       Out1           :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2           :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT atomic_subsys_dead_out_instance_1
  PORT( clk          :   IN    std_logic;
        reset        :   IN    std_logic;
        enb          :   IN    std_logic;
        In1          :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
```

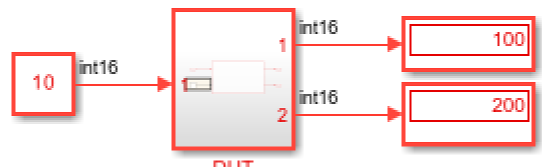
```
        Enable        :   IN    std_logic;
        Out1          :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2          :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;

...

END rtl;
```

Redundant Logic in Atomic Subsystems and Model References

Redundant logic in atomic subsystems, model references, and ForEach subsystem blocks are treated in the same manner during HDL code generation. Redundant logic at the boundary of atomic subsystems are removed during HDL code generation.

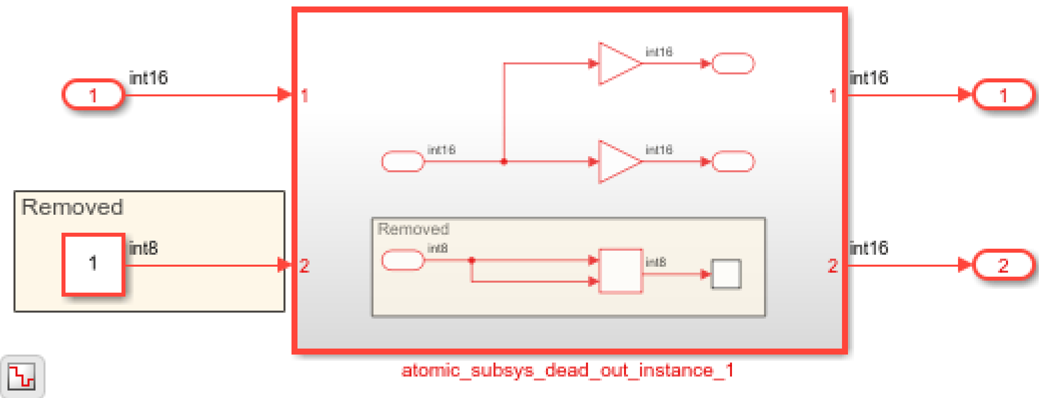
Open the model `hdlcoder_atomic_subsys2_redundant`.

```
open_system('hdlcoder_atomic_subsys2_redundant')
sim('hdlcoder_atomic_subsys2_redundant');
```



The `DUT` subsystem contains a single Atomic Subsystem block. The Constant block is input to the subsystem that has an Add block connected to a Terminator block.

```
open_system('hdlcoder_atomic_subsys2_redundant/DUT')
```



To generate HDL code for the `DUT` subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys2_redundant/DUT')
```

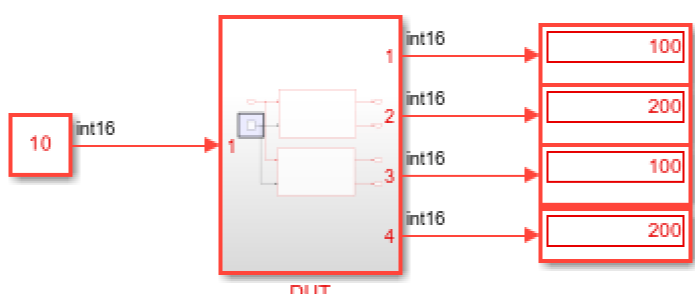
When you generate HDL code, the Add block and the input port are removed because the blocks do not contribute to the evaluation of a DUT outport.

```
ENTITY atomic_subsys_dead_out_instance_1 IS
 PORT( In1     :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1    :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2    :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END atomic_subsys_dead_out_instance_1;

ARCHITECTURE rtl OF atomic_subsys_dead_out_instance_1 IS

...

In1_signed <= signed(In1);

Gain_mul_temp <= to_signed(16#000A#, 16) * In1_signed;
Gain_out1 <= Gain_mul_temp(15 DOWNTO 0);

Out1 <= std_logic_vector(Gain_out1);

Gain1_mul_temp <= to_signed(16#0014#, 16) * In1_signed;
Gain1_out1 <= Gain1_mul_temp(15 DOWNTO 0);

Out2 <= std_logic_vector(Gain1_out1);

END rtl;
```

If there are more than one active instances of the Atomic Subsystem blocks, the redundant logic computation does not cross the subsystem boundary, and the blocks are preserved in the generated HDL code.

Open the model `hdlcoder_atomic_subsys1_redundant`.

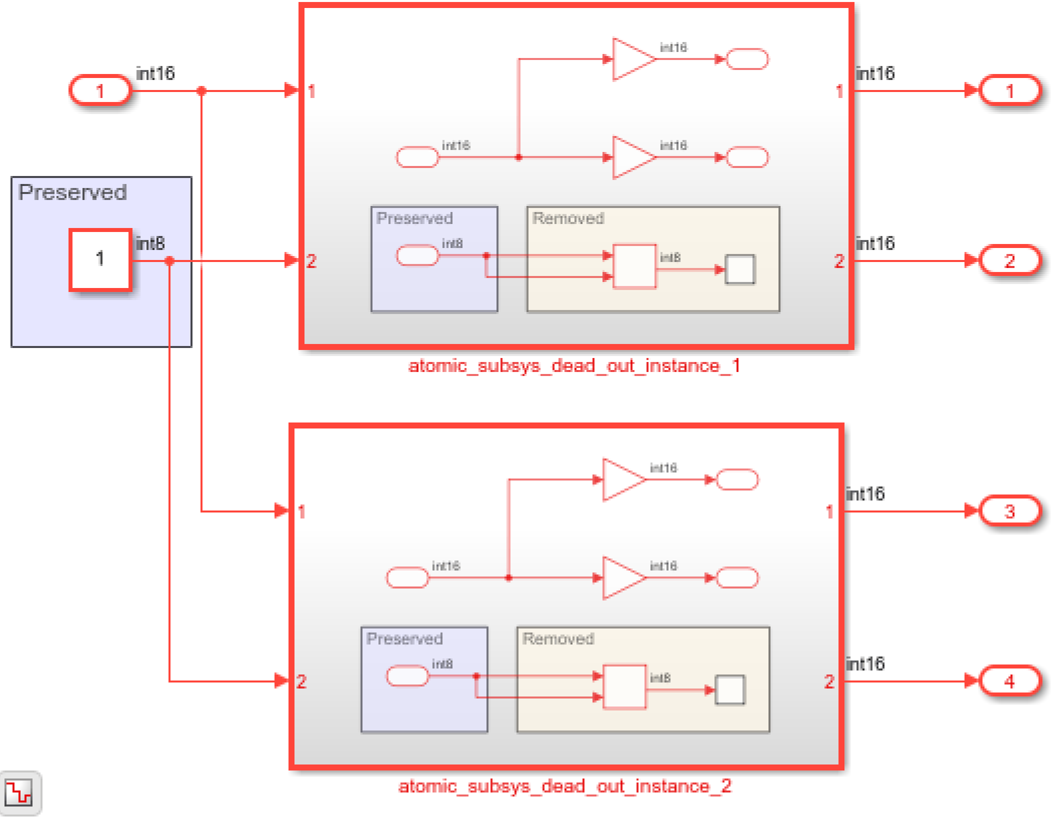
```
open_system('hdlcoder_atomic_subsys1_redundant')
sim('hdlcoder_atomic_subsys1_redundant');
```



Copyright 2020 The MathWorks, Inc.

The `DUT` subsystem contains two atomic subsystems. Inside these subsystems, an Add block is connected to a Terminator block.

```
open_system('hdlcoder_atomic_subsys1_redundant/DUT')
```



To generate HDL code for the `DUT` subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys1_redundant/DUT')
```

When you generate HDL code, the Add block is removed because it does not contribute to the evaluation of a DUT outport. As there are two atomic subsystem instances that are active, the input port to the Add block is preserved during HDL code generation.

A single HDL file `atomic_subsys_dead_out_instance_1` is generated for the atomic subsystems. This file contains the `In2` port declaration but is unused in the HDL code. At the DUT level, `DUT.vhd`, the Constant block is preserved though it is feeding into an input port that does not drive any component inside the Atomic Subsystem.

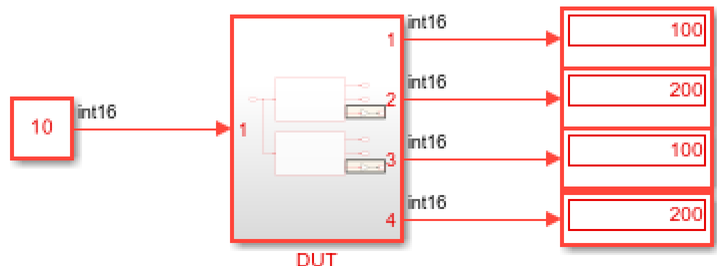
```
ENTITY atomic_subsys_dead_out_instance_1 IS
 PORT( In1       :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       In2       :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
       Out1      :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2      :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END atomic_subsys_dead_out_instance_1;

ARCHITECTURE rtl OF atomic_subsys_dead_out_instance_1 IS

-- Signals
SIGNAL In1_signed          : signed(15 DOWNTO 0);  -- int16
SIGNAL Gain_mul_temp       : signed(31 DOWNTO 0);  -- sfix32
SIGNAL Gain_out1           : signed(15 DOWNTO 0);  -- int16
```

```
SIGNAL Gain1_mul_temp    : signed(31 DOWNTO 0);  -- sfix32
SIGNAL Gain1_out1        : signed(15 DOWNTO 0);  -- int16

...

END rtl;
```

When you have multiple instances of Atomic Subsystem blocks that are active, these instances are preserved in the generated code.

Open the model `hdlcoder_atomic_subsys1_ports_redundant`.

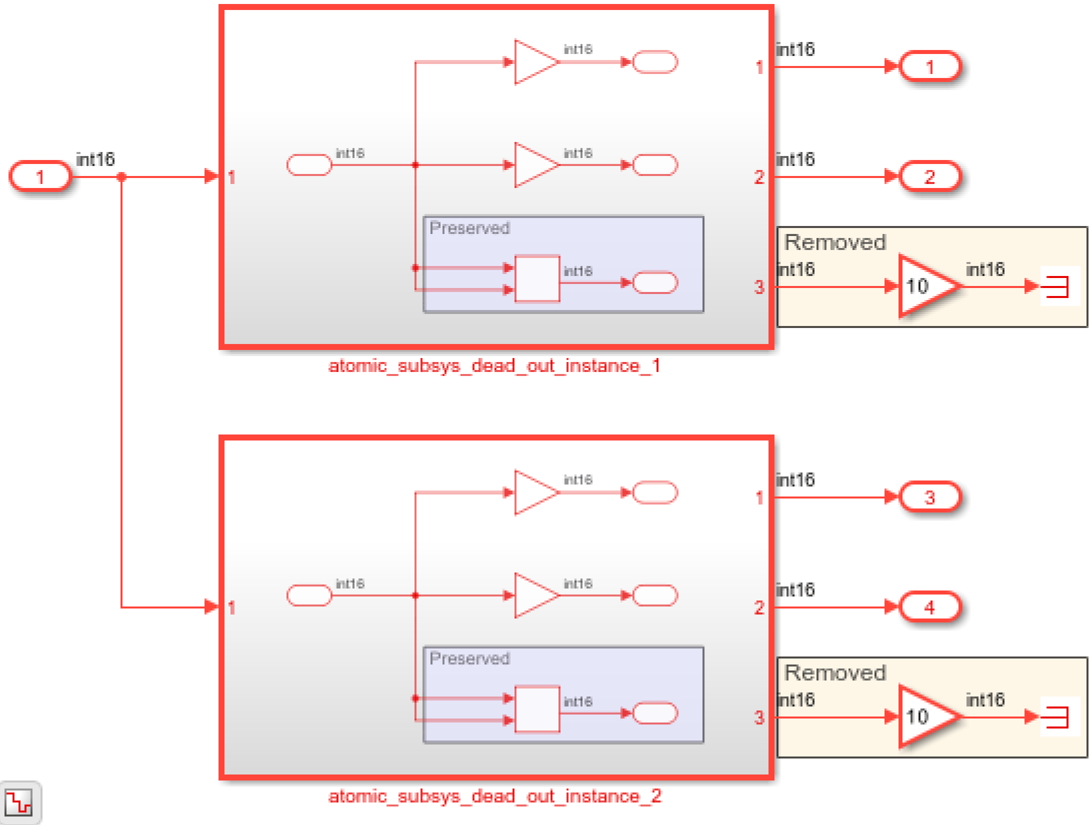
```
open_system('hdlcoder_atomic_subsys1_ports_redundant')
sim('hdlcoder_atomic_subsys1_ports_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains two atomic subsystems that are active.

```
open_system('hdlcoder_atomic_subsys1_ports_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys1_ports_redundant/DUT')
```

The atomic subsystems are preserved in the generated HDL code but the Gain block calculation is removed from the code. In this case, the multiple atomic subsystem instances are active an thus the redundant logic is not removed across the port boundary.

```
ENTITY DUT IS
 PORT( In1      :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1     :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2     :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out3     :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out4     :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT atomic_subsys_dead_out_instance_1
  PORT( In1    :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1   :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2   :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out3   :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;
```

```
...
Out1 <= atomic_subsys_dead_out_instance_1_out1;
Out2 <= atomic_subsys_dead_out_instance_1_out2;
Out3 <= atomic_subsys_dead_out_instance_2_out1;
Out4 <= atomic_subsys_dead_out_instance_2_out2;
END rtl;
```

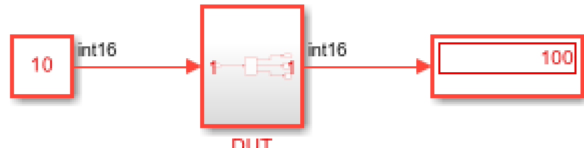
When at least one instance of Atomic Subsystem block is not active and when there are unused output ports outside the Atomic Subsystem blocks, the generated HDL code varies depending on the **Remove Unused Port** setting. See "Optimize Unconnected Ports in HDL Code for Simulink Models" on page 21-246.

Redundant Logic in Masked Subsystems

Redundant logic in masked subsystems are removed in the same manner as for regular Subsystem blocks. When generating mask parameters as generics in the HDL code by setting `MaskParameterAsGeneric` to `on`, the generic ports are preserved in the code.

Open the model `hdlcoder_masked_subsys_redundant`.

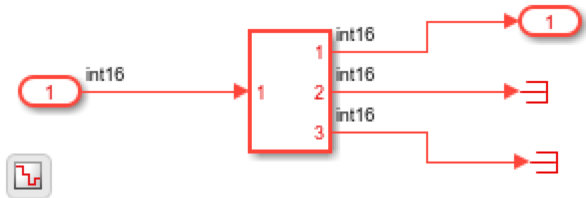
```
open_system('hdlcoder_masked_subsys_redundant')
sim('hdlcoder_masked_subsys_redundant');
```



Copyright 2020 The MathWorks, Inc.

The model contains a masked subsystem that has two mask parameters `Gain` and `Gain1`. `Gain` has the value `10` and }Gain1| has the value |20}.

```
open_system('hdlcoder_masked_subsys_redundant/DUT')
```



To generate HDL code for the `DUT` subsystem, run this command:

```
makehdl('hdlcoder_masked_subsys_redundant/DUT')
```

The generated code shows that generic ports `Gain` and `Gain1` are preserved but the output port `Out2` is removed.

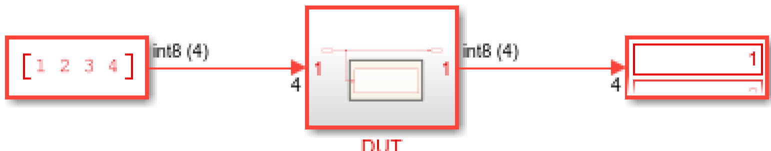
```
ENTITY Subsystem IS
 GENERIC( Gain      : integer := 10;
          Gain1     : integer := 20
          );
 PORT( In1          :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1         :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END Subsystem;

ARCHITECTURE rtl OF Subsystem IS

...

kconst <= to_signed(Gain, 16);

In1_signed <= signed(In1);

Gain_mul_temp <= kconst * In1_signed;
Gain_out1 <= Gain_mul_temp(15 DOWNTO 0);

Out1 <= std_logic_vector(Gain_out1);

END rtl;
```

Redundant Logic in DocBlock and Annotations

Annotations or DocBlock blocks that are inside active subsystems are preserved in the generated HDL code. A subsystem is active if it contains at least one output port that leads to the evaluation of the DUT output, or has an active black box subsystem, as described above. If HDL Coder determines that a subsystem containing an annotation or DocBlock is redundant, then that annotation or DocBlock is also removed from the generated code.

Open the model `hdlcoder_annotations_redundant_logic`.

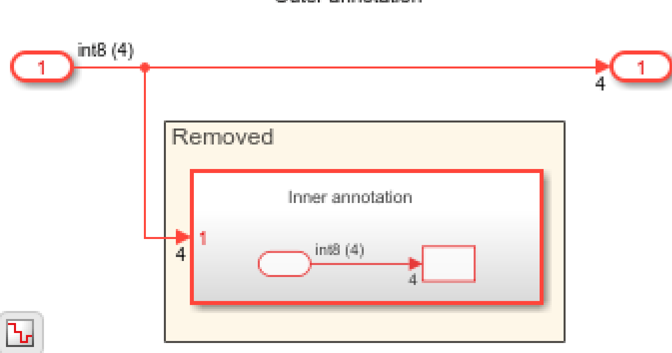
```
open_system('hdlcoder_annotation_redundant_logic')
sim('hdlcoder_annotation_redundant_logic');
```



Copyright 2020 The MathWorks, Inc.

The `DUT` subsystem contains an `Inner annotation` subsystem. This subsystem is redundant because it does not have an active output port.

```
open_system('hdlcoder_annotation_redundant_logic/DUT')
```



To generate HDL code for the `DUT` subsystem, run this command:

```
makehdl('hdlcoder_annotation_redundant_logic/DUT')
```

The generated HDL code shows the `Inner annotation` subsystem including the annotation `Inner annotation` removed from the generated HDL code.

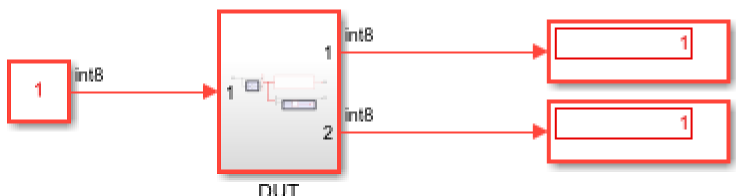
```
ENTITY DUT IS
 PORT( In1   :   IN    vector_of_std_logic_vector8(0 TO 3);  -- int8 [4]
       Out2  :   OUT   vector_of_std_logic_vector8(0 TO 3)  -- int8 [4]
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

BEGIN
 -- Removed
 --
 -- Outer annotation

Out2 <= In1;

END rtl;
```

Redundant Logic in Bus Signals and Bus Element Ports

Redundant logic and unused blocks are removed from the model when it contains buses. The redundant logic optimization applies in the same manner to virtual buses, nonvirtual buses, and bus element ports.

Open the model `hdlcoder_virtual_bus_redundant`.

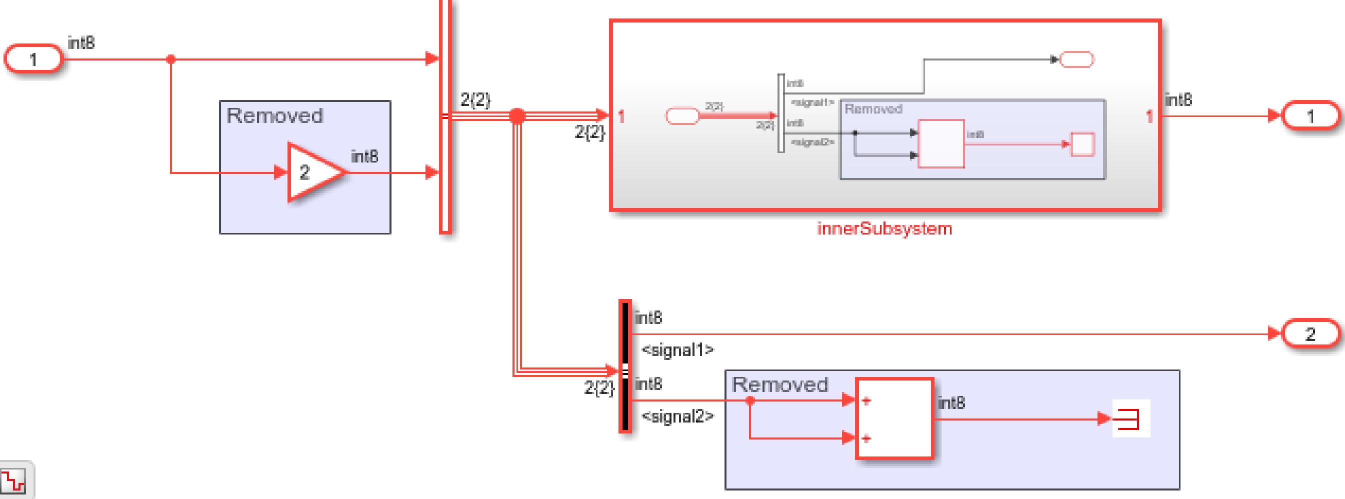
```
open_system('hdlcoder_virtual_bus_redundant')
sim('hdlcoder_virtual_bus_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains a virtual bus that drives an innerSubsystem block and an Add block. One of the bus signals is connected to Terminator blocks at the output.

```
open_system('hdlcoder_virtual_bus_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_virtual_bus_redundant/DUT')
```

The generated HDL code shows that the redundant logic is removed in the design and the input port and output ports that are active are preserved.

```
ENTITY DUT IS
 PORT( In1                                 :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
       Out1                                :   OUT   std_logic_vector(7 DOWNTO 0);  -- int8
       Out2                                :   OUT   std_logic_vector(7 DOWNTO 0)   -- int8
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT innerS
  PORT( In1_signal1                       :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
        Out1                              :   OUT   std_logic_vector(7 DOWNTO 0)   -- int8
```

```
        );
END COMPONENT;

...

END rtl;
```

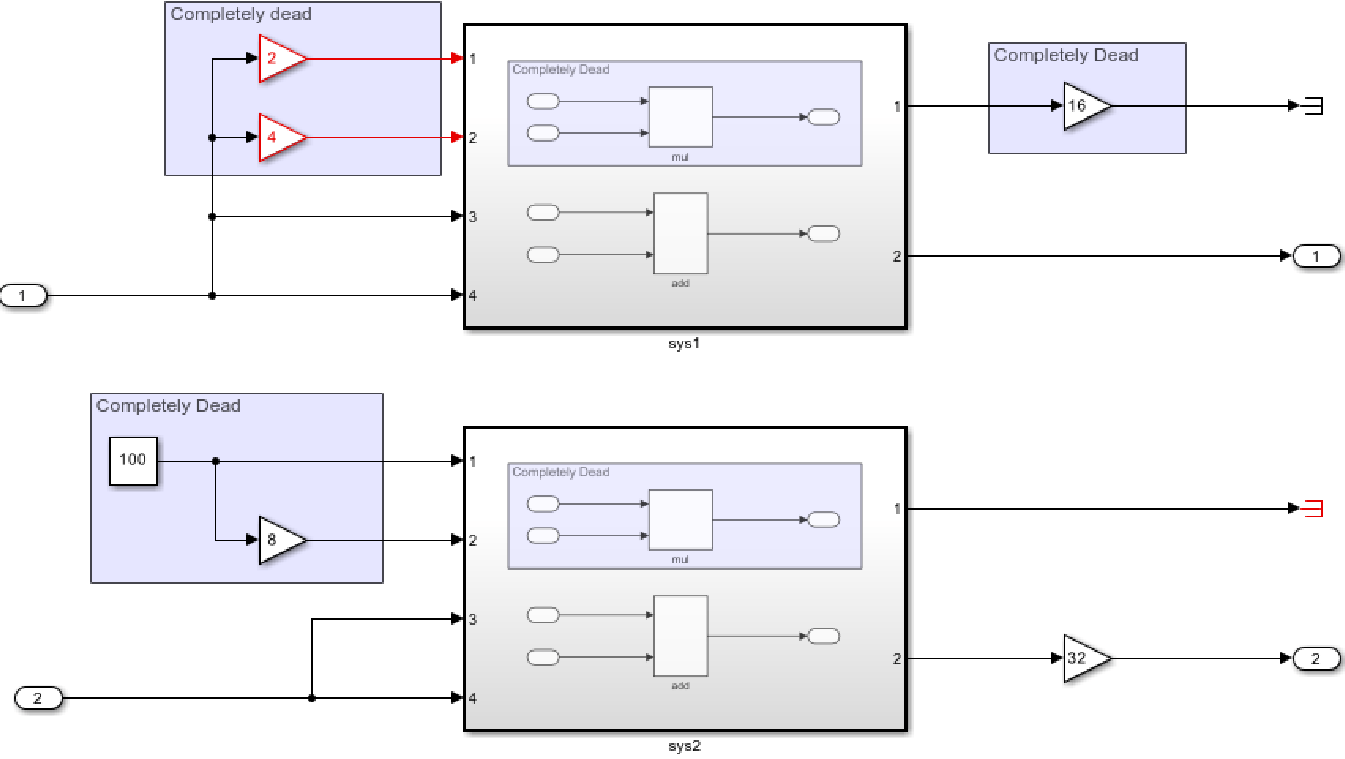
Inside the innerSubsystem block, the Add block connected to the Terminator block is removed.

```
ENTITY innerSubsystem IS
 PORT( In1_signal1                       :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
       Out1                              :   OUT   std_logic_vector(7 DOWNTO 0)  -- int8
       );
END innerSubsystem;

ARCHITECTURE rtl OF innerSubsystem IS

-- Signals
SIGNAL signal1                          : signed(7 DOWNTO 0);  -- int8

BEGIN
 -- Removed

signal1 <= signed(In1_signal1);

Out1 <= std_logic_vector(signal1);

END rtl;
```

Block Connectivity Definition

HDL Coder classifies block connectivity as completely unconnected or partially connected to detect and removed unused logic. Completely unconnected: For modules generated from identical atomic subsystems, model blocks, or ForEach subsystem blocks that are reused across multiple instances, the internal module logic and logic for blocks connected to the module are classified as completely unconnected only if all instances of the module do not use the logic. For example, open the model atomic_complete_unconnected.slx.

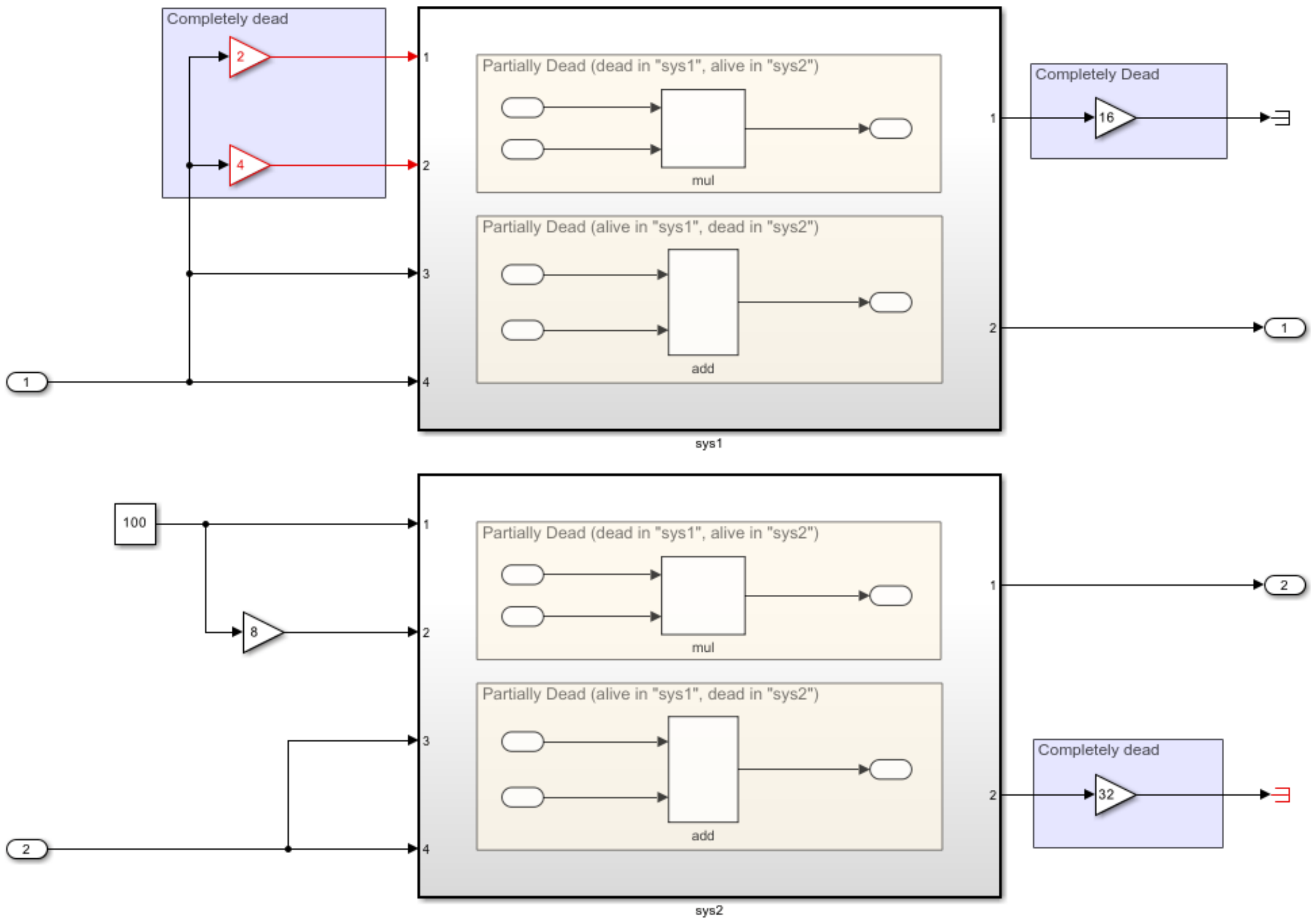
```
open_system('atomic_complete_unconnected')
open_system('atomic_complete_unconnected/DUT')
```



The `mul` block is unused in both `sys1` and `sys2` subsystems and is eliminated from the generated code. Inputs to the `mul` block are completely unconnected as well and are eliminated from the generated code.

Partially connected: For modules generated from identical atomic subsystems, model blocks, or ForEach subsystem blocks that are reused across multiple instances, the internal module logic and logic for blocks connected to the module are classified as partially connected even if one instance of the module uses the repeated logic. For example, open the model `atomic_partial_connected.slx`.

```
open_system('atomic_partial_connected')
```



The `mul` block is used in `sys2`, classifying the block as partially connected and the block logic is retained in the generated code. The inputs to the `mul` block are completely unused and eliminated from the generated code.

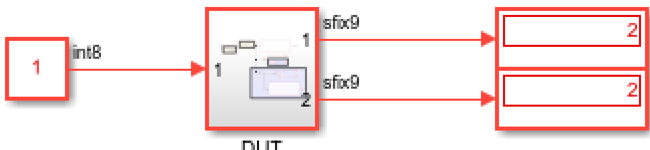
Limitations

1. When your model contains multiple instances of atomic subsystems, model references, or ForEach subsystem blocks, if these blocks are determined to be partially connected during HDL code generation, then all ports are preserved in the generated code.

2. For models that have vector signals, Demux blocks act as boundaries for the redundant logic optimization. Redundant logic and components that are downstream of the Demux block are removed during HDL code generation. The optimization does not cross the Demux block boundary and therefore preserves components that are upstream of the Demux block. This limitation when using vectors also applies when you convert buses to vectors.

Open the model `hdlcoder_vector_redundant_logic`.

```
open_system('hdlcoder_vector_redundant_logic')
sim('hdlcoder_vector_redundant_logic');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains an innerSubsystem block. This subsystem has one active output port and the other port is terminated.

```
open_system('hdlcoder_vector_redundant_logic/DUT')
```

To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_vector_redundant_logic/DUT')
```

The generated HDL code shows that the Add block performs a vector Add calculation. Both the Gain block and Add block are preserved in the generated HDL code as the redundant logic optimization does not cross the Demux block boundary.

```
ENTITY DUT IS
 PORT( In1       :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
       Out1      :   OUT   std_logic_vector(8 DOWNTO 0);  -- sfix9
       Out2      :   OUT   std_logic_vector(8 DOWNTO 0)  -- sfix9
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT innerSubsystem
  PORT( In1      :   IN    vector_of_std_logic_vector9(0 TO 1);  -- sfix9 [2]
        Out1     :   OUT   std_logic_vector(8 DOWNTO 0)  -- sfix9
        );
END COMPONENT;


...

In1_signed <= signed(In1);

Gain1_cast <= resize(In1_signed & '0' & '0', 16);
Gain1_out1 <= Gain1_cast(7 DOWNTO 0);

Mux_out1(0) <= signed(In1);
Mux_out1(1) <= Gain1_out1;

Add_out1_gen: FOR t_0 IN 0 TO 1 GENERATE
  Add_out1(t_0) <= resize(Mux_out1(t_0), 9) + resize(Mux_out1(t_0), 9);
END GENERATE Add_out1_gen;

outputgen: FOR k IN 0 TO 1 GENERATE
  Add_out1_1(k) <= std_logic_vector(Add_out1(k));
END GENERATE;

Out2 <= std_logic_vector(Add_out1(0));


...

END rtl;
```

See Also

More About

- “Generated Model and Validation Model” on page 21-10
- “Optimize Unconnected Ports in HDL Code for Simulink Models” on page 21-246
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder” on page 21-18

Optimize Unconnected Ports in HDL Code for Simulink Models

HDL code generation improves code readability, reduces code size, and reduces area usage by removing unconnected ports from the generated code. This improvement includes removing unconnected vector and scalar ports, bus element ports, and bus ports. Unconnected port removal can help avoid synthesis failure caused by unused ports in the generated code.

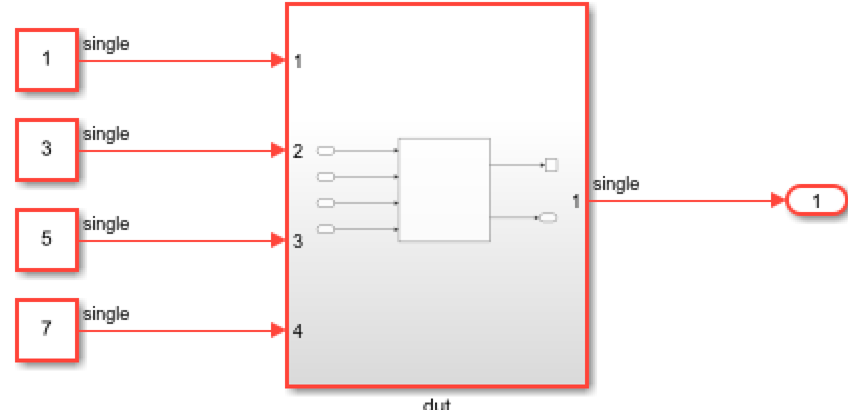
Unused Port Removal

Unused port removal works in conjunction with removal of unused blocks in your design. See “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 21-224.

You can see the effect of unused port removal in the generated HDL code. Ports are not removed from top-level DUT models or subsystems, implementation models, or validation models.

Open the model `hdlcoder_RemoveUnconnectedPorts` containing Bus Element ports and a port connected to an inactive output.

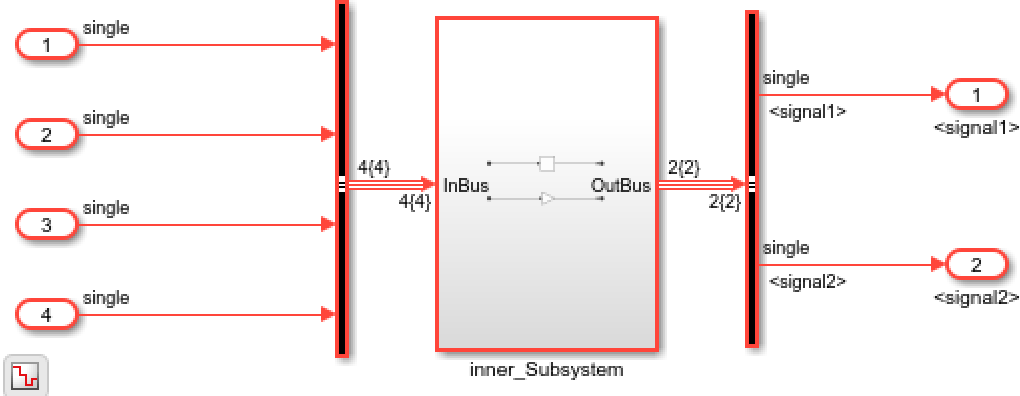
```
open_system('hdlcoder_RemoveUnconnectedPorts')
set_param('hdlcoder_RemoveUnconnectedPorts', 'SimulationCommand', 'update');
```



Copyright 2019-2021 The MathWorks, Inc.

Open the `dut` Subsystem block, and then open the `mid_Subsystem` block. The `mid_Subsystem` contains the Bus Element ports. One of the output signals is connected to a Terminator block.

```
open_system('hdlcoder_RemoveUnconnectedPorts/dut/mid_Subsystem')
```



To generate HDL code for the design, at the MATLAB® command prompt, enter:

```
makehdl('hdlcoder_RemoveUnconnectedPorts/dut')
```

The generated code `mid_Subsystem.vhd` shows that unconnected ports are removed during HDL code generation. The input `InBus_signal3` at the DUT input port is multiplied by a Gain block, and then connected to the output port `OutBus_signal2`, which is then passed to the DUT output port. Because the other input and output ports are unused at the DUT level, these ports are removed from the generated HDL code.

```
ARCHITECTURE rtl OF mid_Subsystem IS

-- Component Declarations
COMPONENT inner_Subsystem
  PORT( clk               :   IN    std_logic;
        reset             :   IN    std_logic;
        enb               :   IN    std_logic;
        InBus_signal3     :   IN    std_logic_vector(31 DOWNTO 0);  -- single
        OutBus_signal2    :   OUT   std_logic_vector(31 DOWNTO 0)  -- single
        );
END COMPONENT;

...

END rtl;
```

Disable Unused Port Deletion Optimization

By default, the optimization is enabled and unused ports are removed in the generated HDL code.

If you do not want unconnected ports to be removed from your design:

1. In the Configuration Parameters dialog box, clear the Remove Unused Ports check box.
2. When you run the HDL Workflow Advisor, in the **Set Code Generation Options > Set Optimization Options** task, clear the **Remove Unused Ports** check box.
3. At the command line, set `DeleteUnusedPorts` to `off` with `hdlset_param` or `makehdl`. For example, to specify that you want to preserve the unused ports in the `hdlcoder_RemoveUnconnectedPorts` model, run this command:

```
makehdl('hdlcoder_RemoveUnconnectedPorts/dut', 'DeleteUnusedPorts', 'off')
```

The generated HDL code preserves the unused Bus Elements ports.

```
ARCHITECTURE rtl OF mid_Subsystem IS

-- Component Declarations
COMPONENT inner_Subsystem
  PORT( clk                 :   IN    std_logic;
        reset               :   IN    std_logic;
        enb                 :   IN    std_logic;
        InBus_signal1       :   IN    std_logic_vector(31 DOWNTO 0);  -- single
        InBus_signal2       :   IN    std_logic_vector(31 DOWNTO 0);  -- single
        InBus_signal3       :   IN    std_logic_vector(31 DOWNTO 0);  -- single
        InBus_signal4       :   IN    std_logic_vector(31 DOWNTO 0);  -- single
        OutBus_signal1      :   OUT   std_logic_vector(31 DOWNTO 0);  -- single
        OutBus_signal2      :   OUT   std_logic_vector(31 DOWNTO 0)  -- single
        );
END COMPONENT;

...

END rtl;
```

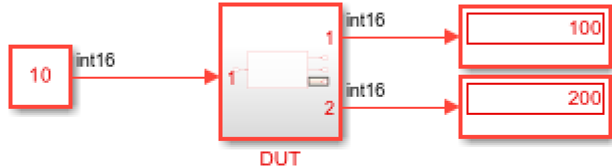
Unused Port Deletion for Subsystem Ports

This optimization can remove unused subsystem data ports. Control ports and ports of a referenced model are not removed.

A subsystem data port is considered to be active when it contributes to a DUT output port downstream or is connected to an active black box subsystem, or a component that is preserved during HDL code generation.

Open the model `hdlcoder_subsys_ports_unused`.

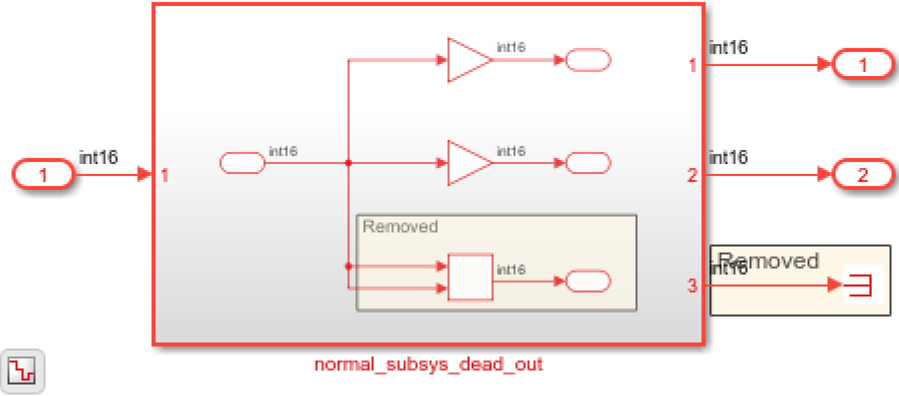
```
open_system('hdlcoder_subsys_ports_unused')
sim('hdlcoder_subsys_ports_unused');
```



Copyright 2020 The MathWorks, Inc.

The model contains a `normal_subsys_dead_out` subsystem that contains an output port `out3` that is terminated and does not contribute to the DUT output.

```
open_system('hdlcoder_subsys_ports_unused/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_subsys_ports_unused/DUT')
```

By default, when `DeleteUnusedPorts` is `on`, the Add block calculation and output port `Out3` are removed in the generated HDL code.

```
ENTITY normal_subsys_dead_out IS
 PORT( In1       :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1      :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2      :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END normal_subsys_dead_out;

ARCHITECTURE rtl OF normal_subsys_dead_out IS

...

  Out1 <= std_logic_vector(Gain_out1);

...

  Out2 <= std_logic_vector(Gain1_out1);

END rtl;
```

To disable `DeleteUnusedPorts` optimization, run this command:

```
makehdl('hdlcoder_subsys_ports_unused/DUT', 'DeleteUnusedPorts', 'off')
```

When you set `DeleteUnusedPorts` to `off`, this port and the Add block calculation are preserved in the generated HDL code.

```
ENTITY normal_subsys_dead_out IS
 PORT( In1     :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1    :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2    :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out3    :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END normal_subsys_dead_out;

ARCHITECTURE rtl OF normal_subsys_dead_out IS

...
```

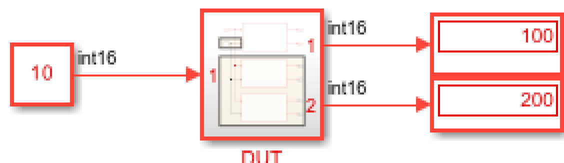
```
  Out1 <= std_logic_vector(Gain_out1);

...

  Out2 <= std_logic_vector(Gain1_out1

  Add_out1 <= to_signed(16#0000#, 16);

  Out3 <= std_logic_vector(Add_out1);

END rtl;
```

Unused Port Deletion for Atomic Subsystems

For unused ports outside atomic subsystems, atomic subsystem instances are removed from the generated HDL code.

Open the model `hdlcoder_atomic_subsys3_redundant`

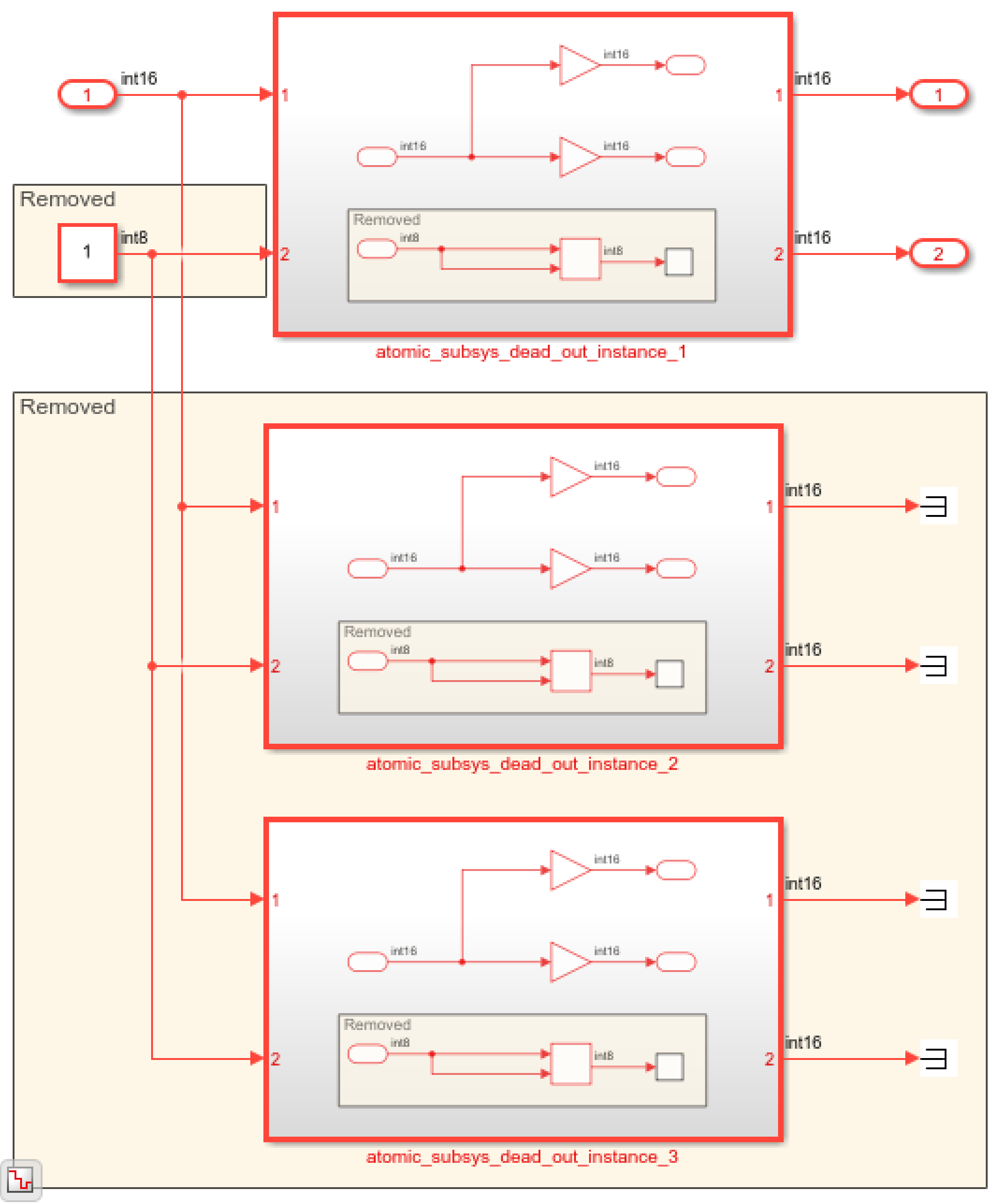
```
open_system('hdlcoder_atomic_subsys3_redundant')
sim('hdlcoder_atomic_subsys3_redundant');
```



Copyright 2020 The MathWorks, Inc.

The `DUT` subsystem contains three atomic subsystem instances. The outputs of two of the atomic subsystem instances are connected to Terminator blocks.

```
open_system('hdlcoder_atomic_subsys3_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys3_redundant/DUT')
```

```
ENTITY DUT IS
 PORT( In1       :   IN    std_logic_vector(15 DOWNTO 0);   -- int16
       Out1      :   OUT   std_logic_vector(15 DOWNTO 0);   -- int16
       Out2      :   OUT   std_logic_vector(15 DOWNTO 0)   -- int16
       );
END DUT;

ARCHITECTURE rtl OF DUT IS
```

```
-- Component Declarations
COMPONENT atomic_subsys_dead_out_instance_1
  PORT( In1    :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1   :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2   :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;

...

END rtl;
```

To set `DeleteUnusedPorts` to `off`, run this command:

```
makehdl('hdlcoder_atomic_subsys3_redundant/DUT', 'DeleteUnusedPorts', 'off')
```

If you set `DeleteUnusedPorta` to `off`, the input port `In2` is preserved in the generated HDL code but it is feeding into an input port that is driving an unused component.

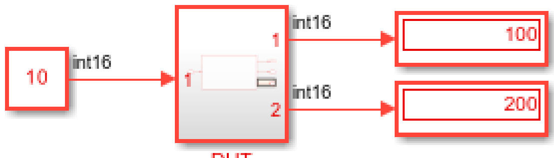
```
ENTITY DUT IS
 PORT( In1        :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1       :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2       :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT atomic_subsys_dead_out_instance_1
  PORT( In1      :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In2      :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
        Out1     :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2     :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;

...

END rtl;
```

Unused Port Deletion for Output Ports of Atomic Subsystems

When you have multiple active atomic subsystem instances, the redundant logic across the boundary including any ports is preserved in the HDL code independent of the `DeleteUnusedPorts` setting. When you have only one Atomic Subsystem instance, by default, when `DeleteUnusedPorts` is `on`, any redundant logic and ports across the subsystem boundary are removed. If you set `DeleteUnusedPorts` to `off`, any unused port is preserved though the logic is redundant.

Open the model `hdlcoder_atomic_subsys2_ports_redundant`.

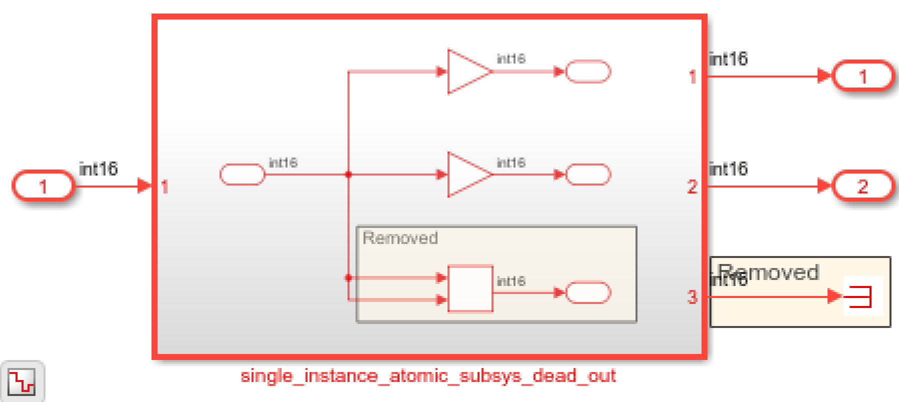
```
open_system('hdlcoder_atomic_subsys2_ports_redundant')
sim('hdlcoder_atomic_subsys2_ports_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains an Atomic Subsystem block that contains an Add block connected to an output port terminated outside the subsystem.

```
open_system('hdlcoder_atomic_subsys2_ports_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys2_ports_redundant/DUT')
```

In the generated HDL code, the Add block and corresponding output port Out3 is removed because it does not contribute to an active output.

```
ENTITY DUT IS
 PORT( In1    :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1   :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2   :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT single_instance_atomic_subsys_dead_out
  PORT( In1    :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1   :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2   :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;

...

END rtl;
```

To set DeleteUnusedPorts to off, run this command:

```
makehdl('hdlcoder_atomic_subsys2_ports_redundant/DUT', 'DeleteUnusedPorts', 'off')
```

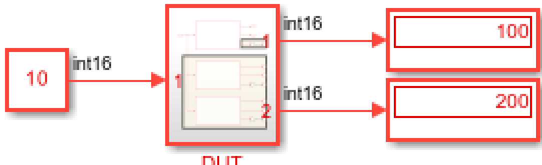
If you set DeleteUnusedPorts to off, the output port Out3 is preserved in the generated HDL code.

```
ENTITY DUT IS
 PORT( In1    :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1   :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2   :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT single_instance_atomic_subsys_dead_out
  PORT( In1    :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1   :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2   :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        Out3   :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;

...

END rtl;

ENTITY single_instance_atomic_subsys_dead_out IS
 PORT( In1      :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1     :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2     :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out3     :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END single_instance_atomic_subsys_dead_out;

ARCHITECTURE rtl OF single_instance_atomic_subsys_dead_out IS


...

Add_out1 <= to_signed(16#0000#, 16);

Out3 <= std_logic_vector(Add_out1);

END rtl;
```

If there are multiple atomic subsystem instances that are not active, the unused port and logic are removed or preserved depending on the DeleteUnusedPorts setting.

Open the model hdlcoder_atomic_subsys3_ports_redundant.

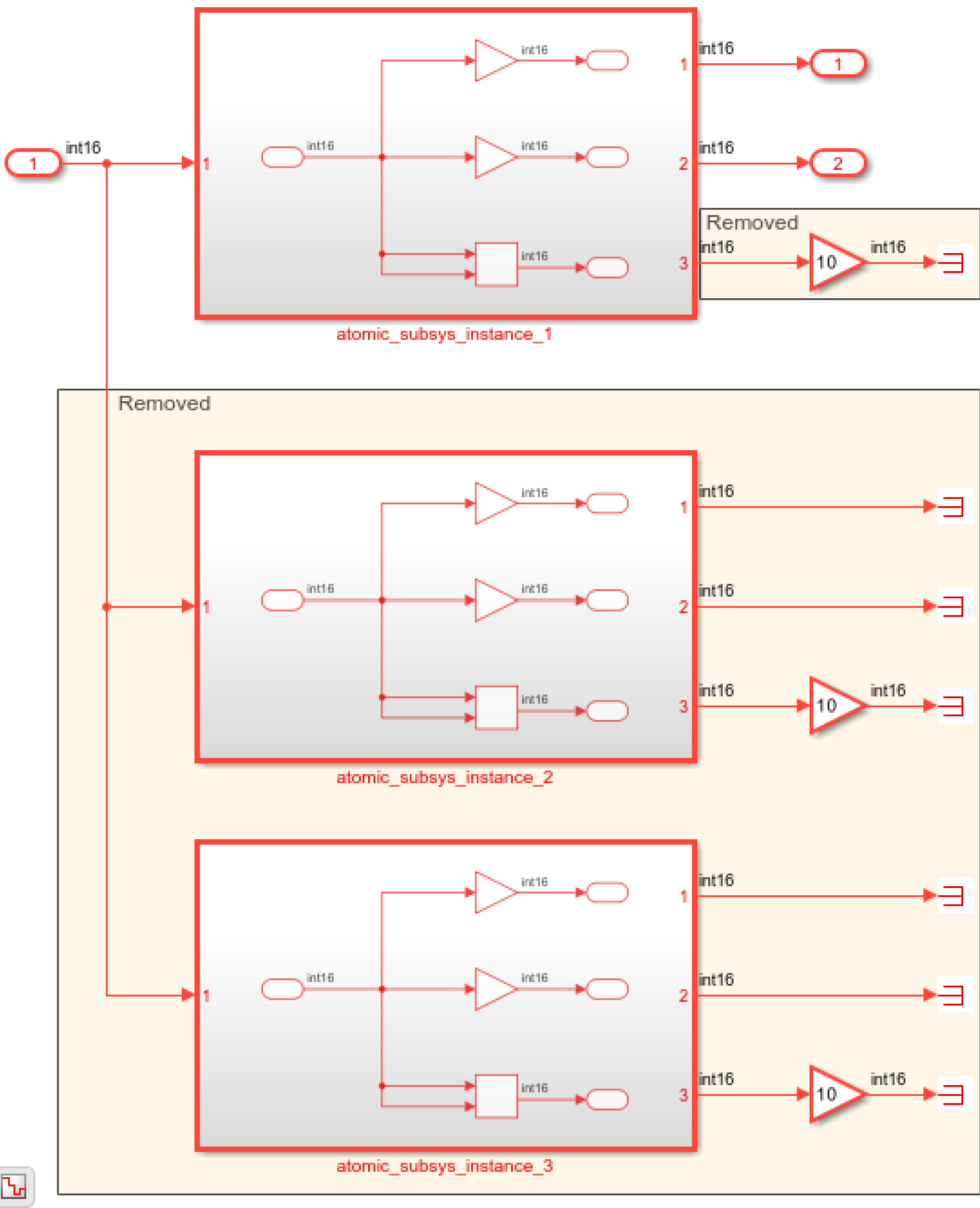
```
open_system('hdlcoder_atomic_subsys3_ports_redundant')
sim('hdlcoder_atomic_subsys3_ports_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains an Atomic Subsystem block that contains an Add block connected to an output port terminated outside the subsystem.

```
open_system('hdlcoder_atomic_subsys3_ports_redundant/DUT')
```



The two atomic subsystem instances that have the output ports terminated are removed in the generated HDL code. In the other Atomic Subsystem block, the Add block calculation and its output is removed.

```
ENTITY DUT IS
 PORT( In1           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1          :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2          :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
```

```
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT atomic_subsys_instance_1
  PORT( In1         :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1        :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2        :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;

...

-- Removed
--
-- Removed

u_atomic_subsys_instance_1 : atomic_subsys_instance_1
  PORT MAP( In1 => In1,  -- int16
            Out1 => atomic_subsys_instance_1_out1,  -- int16
            Out2 => atomic_subsys_instance_1_out2  -- int16
            );

Out1 <= atomic_subsys_instance_1_out1;

Out2 <= atomic_subsys_instance_1_out2;

END rtl;
```

To set `DeleteUnusedPorts` to `off`, run this command:

```
makehdl('hdlcoder_atomic_subsys3_ports_redundant/DUT', 'DeleteUnusedPorts', 'off')
```

If you set `DeleteUnusedPorts` to `off`, the output port `Out3` is preserved in the generated HDL code.

```
ENTITY DUT IS
 PORT( In1          :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
       Out1         :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
       Out2         :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
       );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT atomic_subsys_instance_1
  PORT( In1         :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1        :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2        :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out3        :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;
```

If you see the generated HDL code for the Atomic Subsystem instance, it shows that the Add block computation and Out3 is preserved in the generated HDL code.

```
ENTITY atomic_subsys_instance_1 IS
 PORT( In1           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
```

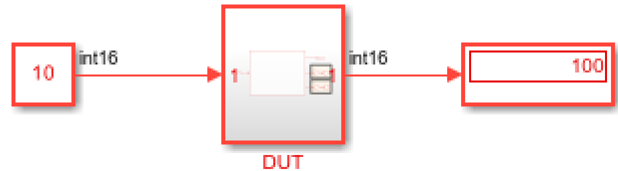
```
        Out1          :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2          :   OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out3          :   OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END atomic_subsys_instance_1;

ARCHITECTURE rtl OF atomic_subsys_instance_1 IS

...

Add_out1 <= In1_signed + In1_signed;

Out3 <= std_logic_vector(Add_out1);

END rtl;
```

Upstream Logic Preservation of Unused Port

You can preserve logic upstream from a Terminator block, its unused logic, while still removing other unconnected ports and unused logic. You can choose specific unused logic that you want to preserve for generated HDL code while still optimizing the rest of your model. Enable the HDL block property `PreserveUpstreamLogic` for the Terminator block for which you want to preserve the logic upstream. See “PreserveUpstreamLogic” on page 19-20.

Open the model `hdlcoder_subsys_port_preserve`.

```
open_system('hdlcoder_subsys_ports_preserved')
sim('hdlcoder_subsys_ports_preserved');
```

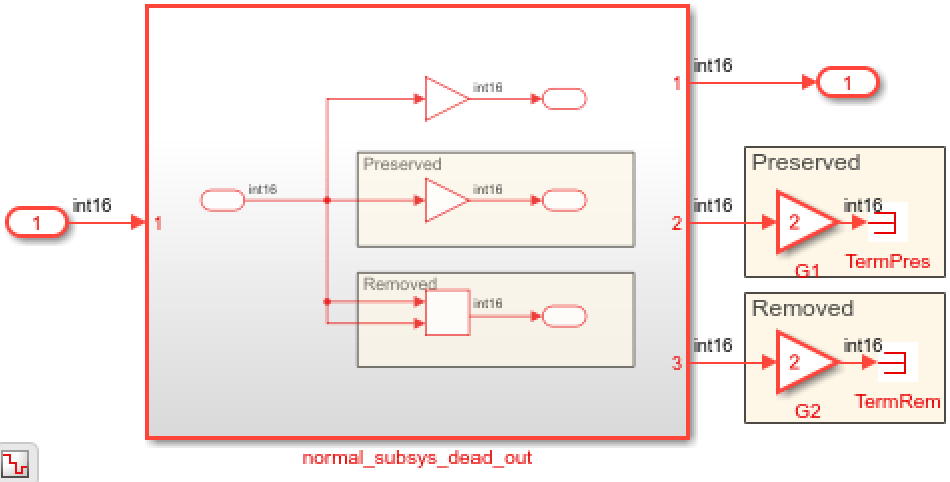


Copyright 2020 The MathWorks, Inc.

The model contains a `DUT` subsystem that has two Terminator blocks as output ports for the subsystem. The Terminator block `TermPres` has the HDL block property `PreserveUpstreamLogic` set to on. The Terminator block `TermRem` does not (the default setting). The model property `DeleteUnusedPorts` is enabled by default.

```
open_system('hdlcoder_subsys_ports_preserved/DUT')
hdlget_param('hdlcoder_subsys_ports_preserved/DUT/TermPres', 'PreserveUpstreamLogic')
```

```
ans =

    'on'
```

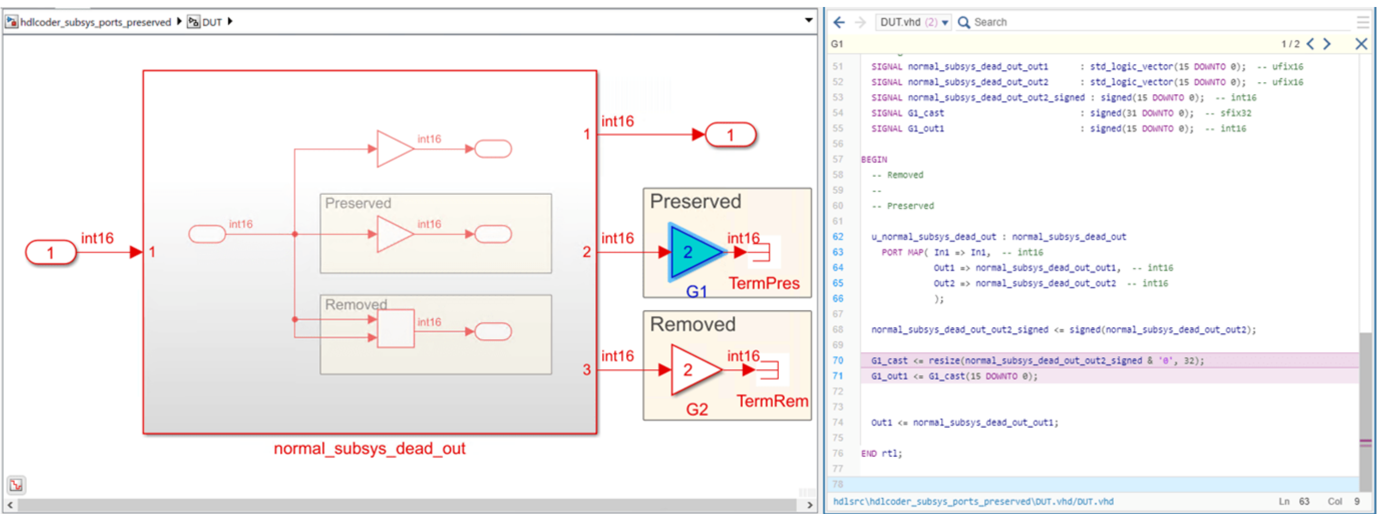


When you generate HDL code, the logic connected upstream from the `TermPres` block is preserved in the HDL code, including the outport `Out2` from the `normal_subsys_dead_out` subsystem. Enabling `PreserveUpstreamLogic` for a Terminator block takes priority over the enabled `DeleteUnusedPorts` option. The logic connected upstream from the `TermRem` block is not preserved, including not preserving the outport from the `normal_subsys_dead_out` subsystem.

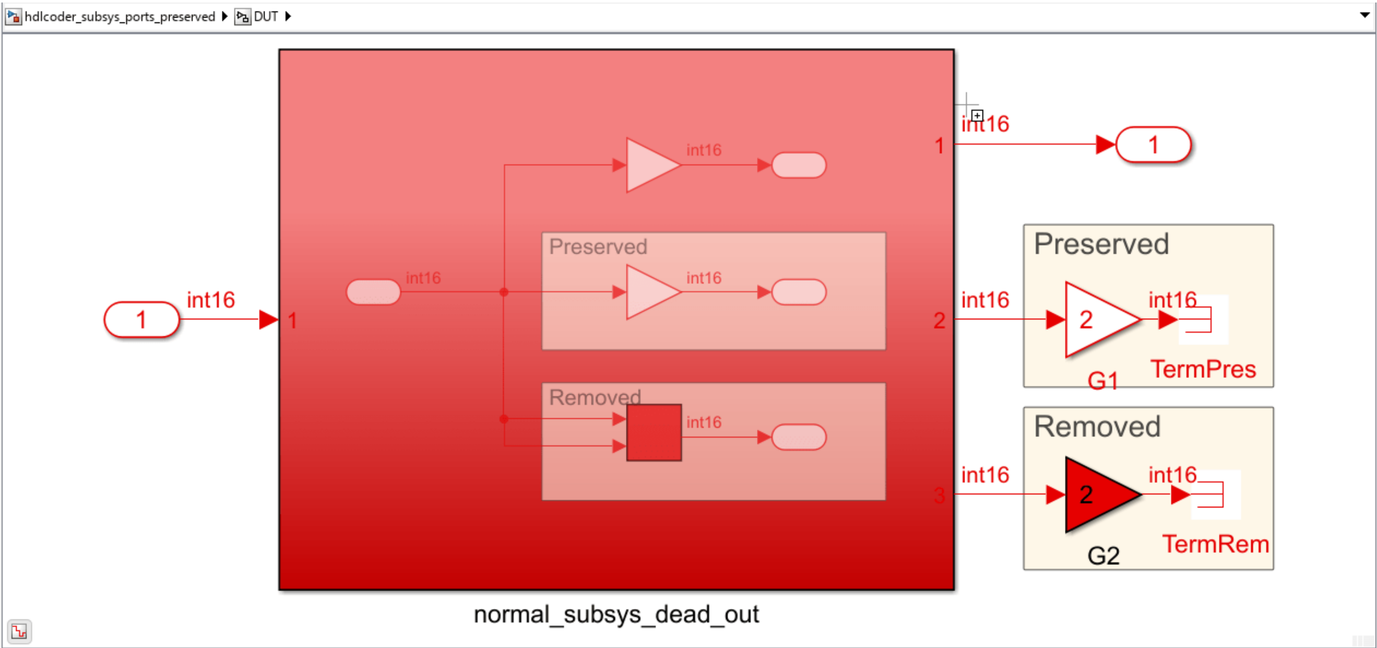
To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_subsys_ports_preserved/DUT')
```

In the code traceability report, click the `G1` gain connected to `TermPres`. The corresponding HDL code shows that the logic connected upstream from `TermPres`, such as `G1`, is preserved. No corresponding HDL code exists for the logic upstream from `TermRem`.



You can highlight the removed logic blocks upstream from `TermRem` by clicking the script `highlightRemovedDeadBlocks.m` in the output of the `makehdl` command. The removed blocks appear in red in the model. To clear the highlighting, click the script `clearHighlightingRemovedDeadBlocks.m`.



You can also view all eliminated and virtual blocks in the code generation report. Click **Traceability Report**, then view the **Eliminated / Virtual Blocks** section.

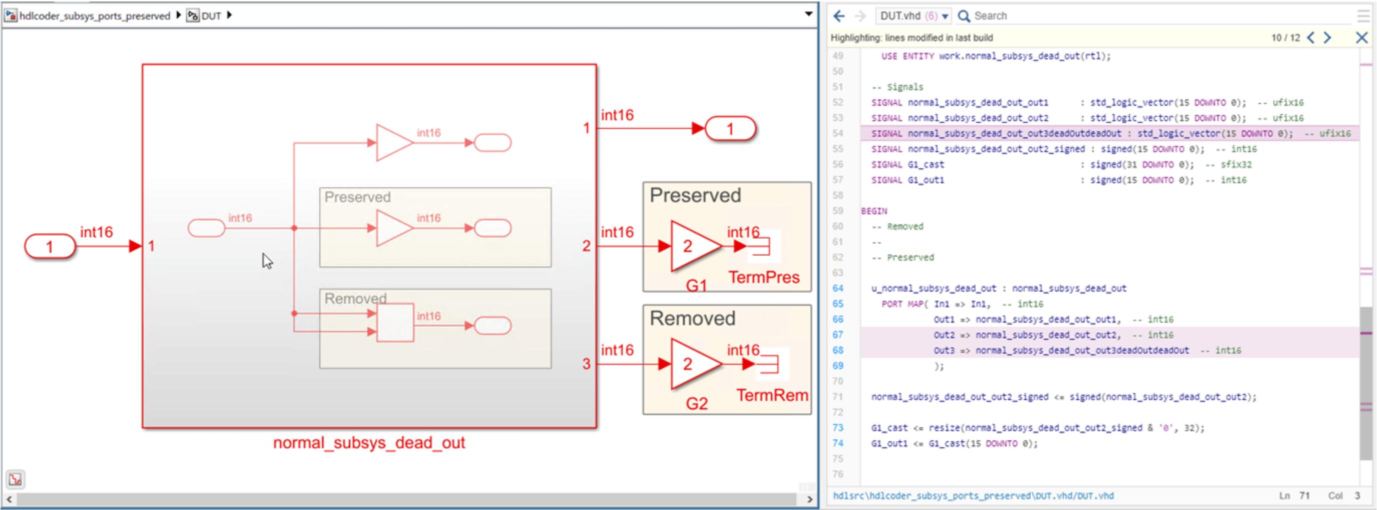
You can preserve the output port from the `normal_subsys_dead_out` subsystem connected to `TermRem` but still remove any upstream logic by disabling `DeleteUnusedPorts` for the model.

```
hdlset_param('hdlcoder_subsys_ports_preserved', 'DeleteUnusedPorts', 'off')
```

Generate HDL code for the DUT subsystem:

```
makehdl('hdlcoder_subsys_ports_preserved/DUT');
```

The traceability report now highlights the difference between the last HDL code generated and this HDL code. The logic upstream from `TermRem`, such as the gain block `G2`, is still removed because `PreserveUpstreamLogic` is off. The outport `Out3` is now in the generated HDL code because `DeleteUnusedPorts` is set to `off` for the model.



Limitations

If your model contains multiple instances of atomic subsystems, model references, or Foreach Subsystem blocks, and these blocks are active during HDL code generation, then ports are preserved in the generated code. Components connected upstream to these ports are also considered active. The ports are preserved independently of whether you enable or disable the `DeleteUnusedPorts` setting.

This limitation also applies to bus signals. In this case, the entire bus is preserved in the generated HDL code. For an example, see “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 21-224.

See Also

More About

- “Generated Model and Validation Model” on page 21-10
- “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 21-224
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder” on page 21-18

I/O Optimizations

- “HDL Code Generation from Frame-Based Algorithms” on page 22-2
- “Use Neighborhood, Reduction, and Iterator Patterns with a Frame-Based Model or Function for HDL Code Generation” on page 22-10
- “Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods” on page 22-17
- “Use Sample-Based Inputs and Frame-Based Inputs in an Algorithm” on page 22-22
- “Synthesize Code for Frame-Based Model” on page 22-26
- “Offload Large Delays from Frame-Based Models to External Memory” on page 22-32
- “Optimize Area Usage for Frame-Based Algorithms with Tall Array Inputs” on page 22-40
- “Compute Image Characteristics with a Frame-Based Model for HDL Code Generation” on page 22-46
- “Generate HDL Code from Frame-Based Model by Using Neighborhood Processing with States” on page 22-52

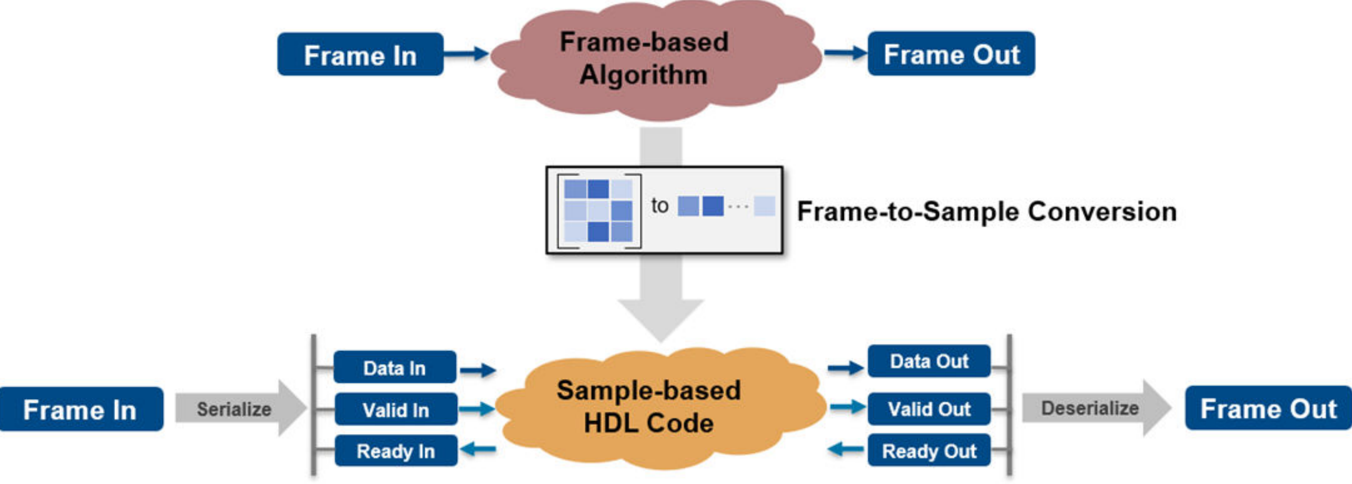
HDL Code Generation from Frame-Based Algorithms

Platforms that have limited I/O like FPGA or ASIC devices typically process large datasets as streaming pixels or samples. To deploy a frame-based model onto these devices, you must manually translate your algorithms to operate on streams of data. You can automate this process and generate HDL code from frame-based models or MATLAB functions with matrix inputs by using the frame-to-sample optimization in HDL Coder. This optimization converts frame-based vector or matrix inputs to smaller-sized samples or pixels for HDL code generation to target stream-based hardware and reduce the FPGA I/O needed to handle large input and output signals. You can optimize designs for hardware while reducing algorithm development time for various use cases in domains that have large inputs, such as image processing, digital signal processing, radar applications, and audio processing.

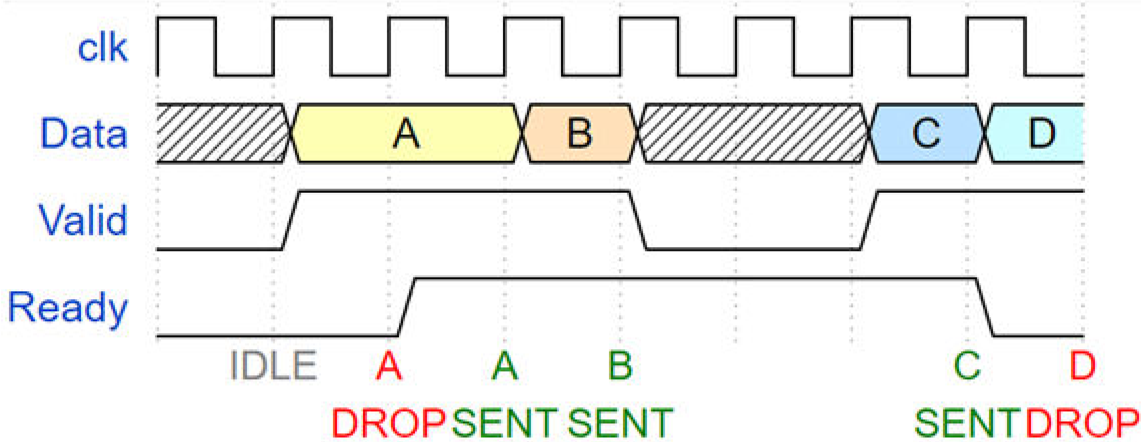
When you use the frame-to-sample optimization, HDL Coder generates hardware-ready HDL code from frame-based algorithms that has the necessary logic to store samples inside the DUT in line buffers, align streams, and balance data paths. You can use multiple modeling patterns, such as element-wise operations, neighborhood operations, and iterative and reduction operations, to author frame-based algorithms supported by the frame-to-sample optimization.

Generating HDL Code from a Frame-based Algorithm

When you use the frame-to-sample conversion to generate HDL code from a frame-based algorithm, HDL Coder transforms your frame-based algorithm into synthesizable HDL code with sample-based logic that has valid and ready control signals and the logic to handle and align the data streams directly from the frame-based algorithm. You can use the frame-to-sample conversion with a Simulink model or a MATLAB function.



When you generate HDL code from a frame-based algorithm, the stream-based HDL code and generated model contain sample-based logic, which includes data signals, valid and ready control signals. This timing diagram maps the relationship between the Valid, Data, and Ready signals.



A Valid signal indicates when data is available. The Ready signal indicates that the DUT can accept and process data. Transfer of data occurs only when both the Valid and Ready signals are high. This is represented by data packets A,B, and C in the image. When the Ready signal is low, the DUT cannot process more data. If you send a Valid and Data signal before the Ready signal assertion, the Data signal is dropped. You can de-assert the Valid signal and not send data while the Ready signal is high. If the Ready signal de-asserts and the Valid signal is high, the Data signal is dropped. This is represented by data packet D in the image.

You can create matrix or frame-based algorithms by using element-wise operations, neighborhood operations, and iterative and reduction operations supported for HDL code generation. You can create:

- Neighborhood operations by using the `hdl.npufun` function a in MATLAB function or the Neighborhood Processing Subsystem block in a Simulink model. You can use `hdl.npufun` in a MATLAB Function block in your DUT to apply neighborhood processing and element-wise operations to an incoming image or matrix, such as filtering with a kernel.
- Iterative operations using the `hdl.iteratorfun` function. For example, you can use `hdl.iteratorfun` in a MATLAB Function block in your DUT to loop over arrays to produce a single output to an incoming image or matrix for histogram equalization and to compute statistics such as `min` and `max`.
- Element-wise operations by using element-wise functions or blocks such as the Gain, Product, Sum, Subtract, and Divide blocks.

In vision or image processing, you can use these frame operations to model 2-D based algorithms, such as filtering, histogram creation, histogram equalization, and edge detection. In signal processing, you can calculate a moving average computation on a large input signal.

Specify the Frame-to-Sample Conversion Optimization

You can generate HDL code with the frame-to-sample conversion optimization from a Simulink model or MATLAB function.

Specify the Frame-to-Sample Conversion Optimization from Simulink

To enable the frame-to-sample conversion optimization for a Simulink model:

1. Individually enable the HDL block property **ConvertToSamples** for Inport blocks on the DUT to convert the incoming vector or matrix input signal to samples by using the frame to sample conversion optimization. To specify an Inport block as a input signal for the frame-to-sample conversion, enter:

   ```
   hdlset_param("<path/to/Inport>", ConvertToSamples = "on")
   ```

2. Select the **Enable frame to sample conversion** parameter in the Configuration Parameters window. Use the frame-to-sample conversion parameters to apply frame-to-sample optimization options to your design. To enable frame-to-sample conversion on a Simulink model, on the command line, enter:

   ```
   hdlset_param("<model_name>", FrameToSampleConversion = "on")
   ```

   For more information, see **Enable frame to sample conversion**.

3. If you design your frame-based algorithm using a MATLAB Function block, set the HDL block property **Architecture** of the MATLAB Function block to `MATLAB Datapath`.

After generating HDL code, you can check the generated model and HDL code to see that each of the matrix inputs streamed contain a sample, valid, and ready bundle. You can use the validation model to compare the original frame-based model and the generated sample-based model. You can also deploy the generated HDL code to an FPGA using the IP core generation workflow. For an example, see “Deploy a Frame-Based Model with AXI4-Stream Interfaces” on page 40-378.

Specify Frame-to-Sample Conversion from MATLAB

To enable the frame-to-sample conversion optimization for a MATLAB function:

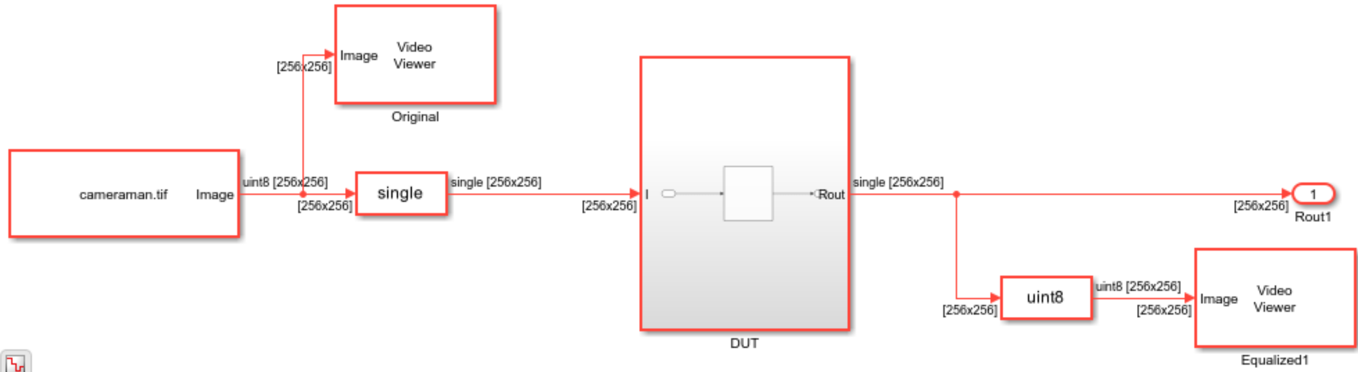
1. Open the MATLAB HDL Workflow Advisor. To get started with the MATLAB HDL Workflow Advisor, see “Basic HDL Code Generation and FPGA Synthesis from MATLAB”.
2. In the left pane, click the **HDL Code Generation** task. In the right pane, navigate to the **Optimization** tab and select **Aggressive Dataflow Conversion**.
3. Click the **Frame to Sample Conversion** tab and select **Enable Frame to Sample Conversion**. Use the frame-to-sample conversion parameters in this tab to apply optimization options to your design.

Generate HDL Code from a Frame-Based Model Example

The Simulink model `hdlFrame_Blur_2D_MLFB` models a common image-processing frame-based blurring algorithm which uses a neighborhood processing pattern.

Open the model to view the frame-based blurring algorithm. The model applies an image-blurring kernel to the image `cameraman.tif`.

```
open_system("hdlFrame_Blur_2D_MLFB");
set_param(gcs,'SimulationCommand','Update');
```



The DUT contains a MATLAB function called `image_blur` in the MATLAB Function block. The `image_blur` function calls the frame-to-sample supported function `hdl.npufun` that applies the blurring kernel in the `blur` function to the input image. For more information, see `hdl.npufun`.

```
open_system("hdlFrame_Blur_2D_MLFB/DUT/MATLAB Function")
```

```
function I_out = image_blur(I)
% Blur input image

I_out = hdl.npufun(@blur, [3 3], I);

end

function y = blur(N)

y = sum(N(:),'native')/9;

end
```

Apply the frame-to-sample conversion optimization by setting the model configuration parameter `FrameToSampleConversion` to `on` from the command line.

```
hdlset_param("hdlFrame_Blur_2D_MLFB", FrameToSampleConversion = "on");
```

Specify which incoming frame-based signal to be convert to a sample-based signal by setting the HDL Inport block property `ConvertToSamples` of the input to `on`. In this example, set the `ConvertToSamples` property to `on` for the only Inport block to the `DUT` subsystem, `I`.

```
hdlset_param("hdlFrame_Blur_2D_MLFB/DUT/I", ConvertToSamples = "on");
```

To see the conversion from the frame-based model to the sample-based version, enable model generation. Generate HDL code and the generated model from the `DUT` subsystem.

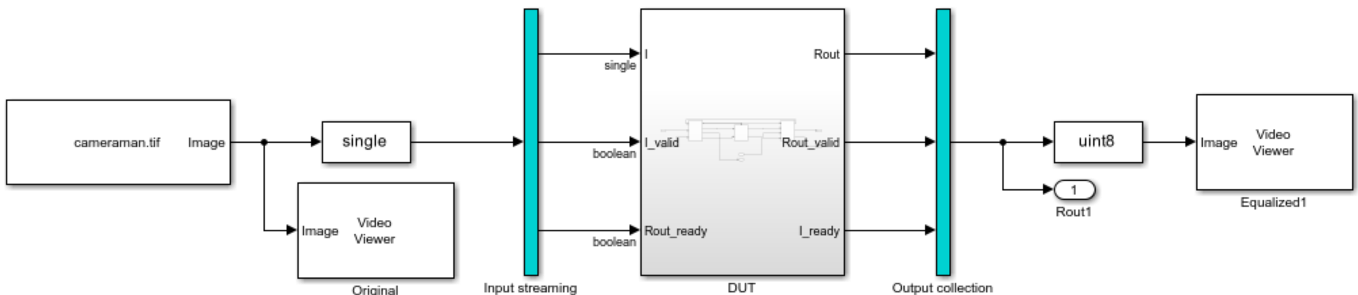
```
hdlset_param("hdlFrame_Blur_2D_MLFB", GenerateModel = "on");
makehdl('hdlFrame_Blur_2D_MLFB/DUT');

bdclose('hdlFrame_Blur_2D_MLFB/Original');
bdclose('hdlFrame_Blur_2D_MLFB/Equalized1');
```

```
### Working on the model <a href="matlab:open_system('hdlFrame_Blur_2D_MLFB')">hdlFrame_Blur_2D_M
### Generating HDL for <a href="matlab:open_system('hdlFrame_Blur_2D_MLFB/DUT')">hdlFrame_Blur_2D
```

```
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlFrame_Blur_2D
### Running HDL checks on the model 'hdlFrame_Blur_2D_MLFB'.
### Begin compilation of the model 'hdlFrame_Blur_2D_MLFB'...
### Working on the model 'hdlFrame_Blur_2D_MLFB'...
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 127 cycles.
### Output port 1: The first valid output of this port will be after an initial latency of 257 va
### Output port 2: 127 cycles.
### Output port 2: The first valid output of this port will be after an initial latency of 257 va
### Working on... <a href="matlab:configset.internal.open('hdlFrame_Blur_2D_MLFB', 'GenerateModel
### Begin model generation 'gm_hdlFrame_Blur_2D_MLFB'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlFrame_Blur_2D_MLFB\gm_hdlFram
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdls
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlFram
### Begin VHDL Code Generation for 'hdlFrame_Blur_2D_MLFB'.
### MESSAGE: The design requires 65536 times faster clock with respect to the base rate = 1.
### Working on counterNetwork as hdlsrc\hdlFrame_Blur_2D_MLFB\counterNetwork.vhd.
### Working on NeighborhoodCreator_3x3/row3_col2 as hdlsrc\hdlFrame_Blur_2D_MLFB\row3_col2.vhd.
### Working on NeighborhoodCreator_3x3/row2_linebuffer/SimpleDualPortRAM_generic as hdlsrc\hdlFra
### Working on NeighborhoodCreator_3x3/row2_linebuffer as hdlsrc\hdlFrame_Blur_2D_MLFB\row2_lineb
### Working on NeighborhoodCreator_3x3 as hdlsrc\hdlFrame_Blur_2D_MLFB\NeighborhoodCreator_3x3.vh
### Working on boundaryCounters_3_3 as hdlsrc\hdlFrame_Blur_2D_MLFB\boundaryCounters_3_3.vhd.
### Working on BoundaryCheck_3x3 as hdlsrc\hdlFrame_Blur_2D_MLFB\BoundaryCheck_3x3.vhd.
### Working on I_NeighborhoodCreator as hdlsrc\hdlFrame_Blur_2D_MLFB\I_NeighborhoodCreator.vhd.
### Working on hdlFrame_Blur_2D_MLFB/DUT/MATLAB Function/blur/nfp_div_single as hdlsrc\hdlFrame_B
### Working on hdlFrame_Blur_2D_MLFB/DUT/MATLAB Function/blur/nfp_add_single as hdlsrc\hdlFrame_B
### Working on hdlFrame_Blur_2D_MLFB/DUT/MATLAB Function/blur as hdlsrc\hdlFrame_Blur_2D_MLFB\blu
### Working on hdlFrame_Blur_2D_MLFB/DUT/MATLAB Function as hdlsrc\hdlFrame_Blur_2D_MLFB\MATLAB_F
### Working on hdlFrame_Blur_2D_MLFB/DUT/Input_FIFOs/I_FIFO as hdlsrc\hdlFrame_Blur_2D_MLFB\I_FIF
### Working on hdlFrame_Blur_2D_MLFB/DUT/Input_FIFOs as hdlsrc\hdlFrame_Blur_2D_MLFB\Input_FIFOs
### Working on hdlFrame_Blur_2D_MLFB/DUT/Output_FIFOs/I_out_FIFO as hdlsrc\hdlFrame_Blur_2D_MLFB\
### Working on hdlFrame_Blur_2D_MLFB/DUT/Output_FIFOs as hdlsrc\hdlFrame_Blur_2D_MLFB\Output_FIFO
### Working on hdlFrame_Blur_2D_MLFB/DUT as hdlsrc\hdlFrame_Blur_2D_MLFB\DUT.vhd.
### Generating package file hdlsrc\hdlFrame_Blur_2D_MLFB\DUT_pkg.vhd.
### Code Generation for 'hdlFrame_Blur_2D_MLFB' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlFrame_Blur_2D_MLFB' complete with 0 errors, 0 warnings, and 2 messages.
### HDL code generation complete.
```

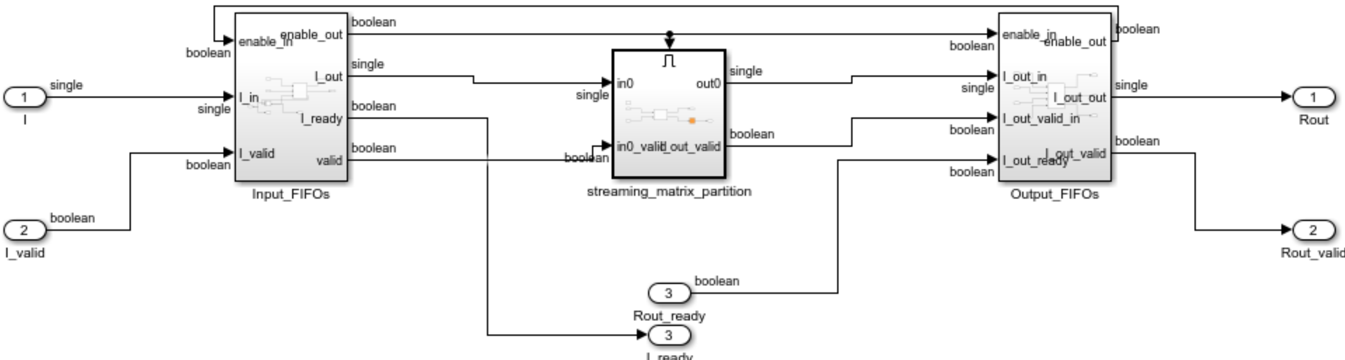
Open the generated model, `gm_hdlFrame_Blur_2D_MLFB`. This model contains the sample-based version of the DUT and contains Valid, Ready, and Data signals.

```
open_system('gm_hdlFrame_Blur_2D_MLFB');
Simulink.BlockDiagram.arrangeSystem('gm_hdlFrame_Blur_2D_MLFB');
```



Open the generated model DUT. There is significantly more hardware detail in the sampled-based generated model than in the original frame-based model. In the `DUT` subsystem, there is an input and output FIFO that handles and stores the data from the generated streaming matrix partition. The streaming matrix partition contains the blurring algorithm and other sample-based logic needed to deploy the algorithm to stream-based hardware.

```
open_system('gm_hdlFrame_Blur_2D_MLFB/DUT');
```



If you run the generated model, notice that the time it takes to get the output image is longer than the original model. Converting a frame-based algorithm to a sample-based algorithm that streams pixels of the image one at a time requires more time to process each pixel than the frame-based version. The latency to process a valid output pixel for a valid input pixel from the original input frame depends on a few factors, such as size of the input image, the samples per cycle, and the algorithm. The output of the `makehdl` command displays the added latency in the MATLAB command window. In this example, the first valid output is accessible after an initial latency of 257 valid inputs.

The sampling rate, latency, and image size determine the simulation stop time needed for the generated model. To generate the blurred output image from the sample-based algorithm in the generated model, the original and subsequent generated model have stop times of 2 seconds, which is the time needed to display the entire output image after latency is introduced.

Hardware Considerations

Prior to generating HDL code from frame-based algorithms for hardware deployment:

- Check that the target hardware should have enough memory to store frames in memory before and after the stream-based algorithm is applied. Sample-based algorithms require more memory, but use significantly less I/O.

- Ensure your design can handle some latency. Sample-based algorithms add latency and delay depending on your algorithm, design, and input data size.

Supported Blocks and Operations

Frame-to-sample conversion supports these blocks and operations:

- MATLAB Function block with the HDL block property **Architecture** set to `MATLAB Datapath`. In the MATLAB function, you can use:
  - Frame-to-sample-focused functions, such as `hdl.npufun` and `hdl.iteratorfun`.
  - Element-wise operations between streamed data.
  - Operations between a scalar and streamed data.
- Neighborhood Processing Subsystem block.
- Element-wise operations. For example, you can use blocks that support element-wise operations such as the Gain, Product, Sum, Subtract, and Divide blocks.
- Square matrix product operations. For example, you can use the Matrix Multiply block with square input matrices.
- 3-D matrices.
- Constant inputs or sources.
- Saturation.
- Trigonometric operations.
- Bit-wise operations, such as bit-slice, bit-and, and bit-or.
- Comparison operations.
- Bias block.
- Abs block.
- Complex to Real-Imag block and Real-Imag to Complex block.
- Sqrt block.
- Data type conversion block.
- Switch block.

Limitations

Frame-to-sample conversion does not support:

- Column vectors.
- Persistent variables in `hdl.iteratorfun` function.
- Non-square matrix product operations.
- Matrix operations on streamed data. For example, frame-to-sample conversion in Simulink does not support using the Selector, Assignment, or Reshape block on streamed data.
- Sum of elements operations.
- Product of elements operations.
- MATLAB System Objects such as `hdl.RAM`.
- For Each Subsystems.

- Delay blocks.
- Blocks that contain design delays.
- `Bus` objects as a datatype.
- IP core generation from a MATLAB function.

See Also

Functions
`hdl.npufun` | `hdl.iteratorfun`

Blocks
Neighborhood Processing Subsystem

Model Settings
Enable frame to sample conversion | **Samples per cycle**

Related Examples

- "Synthesize Code for Frame-Based Model" on page 22-26
- "Offload Large Delays from Frame-Based Models to External Memory" on page 22-32
- "Deploy a Frame-Based Model with AXI4-Stream Interfaces" on page 40-378
- "Deploy Frame-Based Models with AXI4-Stream Video Interfaces in Zynq-Based Hardware" on page 40-386
- "Compute Image Characteristics with a Frame-Based Model for HDL Code Generation" on page 22-46
- "Optimize Area Usage for Frame-Based Algorithms with Tall Array Inputs" on page 22-40
- "Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods" on page 22-17
- "Use Sample-Based Inputs and Frame-Based Inputs in an Algorithm" on page 22-22
- "Use Neighborhood, Reduction, and Iterator Patterns with a Frame-Based Model or Function for HDL Code Generation" on page 22-10

Use Neighborhood, Reduction, and Iterator Patterns with a Frame-Based Model or Function for HDL Code Generation

In this example, you use frame-based modeling to model an image processing algorithm that rectifies fog in an image. You then generate synthesizable HDL code by using the HDL Coder™ frame-to-sample conversion optimization to generate pixel-based code from the frame-based model.

The fog rectification model uses neighborhood, reduction, and iterator patterns. You can model neighborhood patterns by using the `hdl.npufun` function in a MATLAB® Function block or the Neighborhood Processing Subsystem block. You can implement reduction and iterator patterns by using `hdl.iteratorfun` in a MATLAB Function block.

Additionally, as an alternative from generating HDL code from a Simulink® model, you can use a MATLAB function to generate HDL code that contains the frame-based fog rectification algorithm.

Model Algorithm by Using a MATLAB Function Block

In the `hdlFrame_FogRectification_2D_MLFB` model, the device under test (DUT) contains a MATLAB Function block that includes the neighborhood processing, reduction, and iterator operations needed for fog rectification. To generate synthesizable HDL code from the frame-based model, the neighborhood processing algorithm uses the frame-to-sample conversion supported function `hdl.npufun`. The iterator and reduction algorithms use the frame-to-sample conversion supported function `hdl.iteratorfun`. For more information, see `hdl.npufun` and `hdl.iteratorfun`. Using these two functions in a frame-based model enables you to contain an entire fog rectification algorithm in a single MATLAB Function block.

Open the `hdlFrame_FogRectification_2D_MLFB/DUT/MATLAB Function` subsystem to see the fog rectification algorithm.

```
load_system("hdlFrame_FogRectification_2D_MLFB");
open_system("hdlFrame_FogRectification_2D_MLFB/DUT/MATLAB Function");
```

```
function [r_out,g_out,b_out]  = fog_rectification(Ru, Gu, Bu)

[im_gray, restoreR, restoreG, restoreB] = fog_stage1(Ru, Gu, Bu);

[r_out,g_out,b_out] = fog_rectification_m(im_gray, restoreR, restoreG, restoreB);

end

function [im_gray, restoreR, restoreG, restoreB] = fog_stage1(Ru, Gu, Bu)
coder.inline('never')


%% Dark channel Estimation from input
darkChannel = hdl.npufun(@min_kernel, [1 1], Ru, Gu, Bu);

% diff_im is used as input and output variable for anisotropic diffusion
diff_im = fi(fi(0.9,0,16,14)*darkChannel, 1, 32, 16);

%% Refine dark channel using Anisotropic diffusion.
diff_im(:) = hdl.npufun(@filter_kernel, [3 3], diff_im);
diff_im(:) = hdl.npufun(@filter_kernel, [3 3], diff_im);
```

```
diff_im(:) = hdl.npufun(@filter_kernel, [3 3], diff_im);

%% Reduction with min
% diff_im = min(darkChannel,diff_im);
diff_im(:) = hdl.npufun(@min_kernel2, [1 1], darkChannel, diff_im);

diff_im(:) = cast(fi(0.6,0,16,14)*diff_im, 'like', diff_im);

%% Parallel element-wise math to compute
%  Restoration with inverse Koschmieder's law
factor = hdl.npufun(@calc_restoration_factor, [1 1], diff_im);

restoreRf = (Ru-diff_im).*factor;
restoreGf = (Gu-diff_im).*factor;
restoreBf = (Bu-diff_im).*factor;

% Convert to integer
restoreR = uint8(restoreRf);
restoreG = uint8(restoreGf);
restoreB = uint8(restoreBf);

%%
% Stretching performs the histogram stretching of the image.
% im is the input color image and p is cdf limit.
% out is the contrast stretched image and cdf is the cumulative prob.
% density function and T is the stretching function.
% RGB to grayscale conversion
im_gray = hdl.npufun(@convert_gray, [1 1], restoreR, restoreG, restoreB);

% [restoreR, restoreG, restoreB] = hdl.npufun(@zero_to_one, [1 1], restoreR, restoreG, restoreB)
% restoreR(restoreR == 0) = 1;
% restoreG(restoreG == 0) = 1;
% restoreB(restoreB == 0) = 1;

% Rewriting 0-1 setting to avoid reshape
zeros_to_one = (restoreR == 0);
restoreR = restoreR + uint8(zeros_to_one);
zeros_to_one = (restoreG == 0);
restoreG = restoreG + uint8(zeros_to_one);
zeros_to_one = (restoreB == 0);
restoreB = restoreB + uint8(zeros_to_one);

end

function out = filter_kernel(in)
hN = fi([0.0625 0.1250 0.0625; 0.1250 0.2500 0.1250; 0.0625 0.1250 0.0625], 0, 16, 15);
out = cast(sum(in(:).*hN(:)),'like',in);
end

function out = min_kernel(R, G, B)
    out = min([R, G, B]);
end

function out = convert_gray(R, G, B)

out = uint8(round(fi(0.2989,1,16,15) * R + fi(0.5870,1,16,15) * G + fi(0.1140,1,16,15) * B));

end
```

```
function out = min_kernel2(A, B)
    out = min(A, B);
end

function factor = calc_restoration_factor(diff_im)

ds = bitshift(diff_im, -8); % divide by 256 to make value < 1
% compute factor with taylor series expansion
% 1/(1-x) = 1 + x + x^2 + x^3 + x^4 + x^5;
ds2 = fi(ds*ds,1,32,16);
ds3 = fi(ds2*ds,1,32,16);
ds4 = fi(ds3*ds,1,32,16);
ds5 = fi(ds4*ds,1,32,16);
factor = fi(1 + ds + ds2 + ds3 + ds4 + ds5,1,32,24);

end

function [r_out,g_out,b_out] = fog_rectification_m(im_gray, r, g, b)

coder.inline('never')
%FOG_RECTIFICATION_HDL_3_STAGE2
% histogram computation
% Find cdf
% Find breakpoints of table
% Compute table
% Use table on input image to create output image

% histogram calculation
hist = zeros(1, 256, 'uint32');
hist = hdl.iteratorfun(@hist_kernel_fcn, im_gray, hist);

cdf_init = zeros(256,1,'uint32');
cdf = hdl.iteratorfun(@cdf_compute, hist, cdf_init);

N = numel(im_gray);
p1 = fi(5*N/100);
p2 = fi(N-(5*N/100));

i1i2 = zeros(1,2,'uint8');
i1i2 = hdl.iteratorfun(@find_break_points, cdf, i1i2, p1, p2);
i1 = i1i2(1);
i2 = 255-i1i2(2);

o1 = fi(255*0.10,1,32,16,'RoundingMethod','Zero','OverflowAction','Saturate');
o2 = fi(255*0.90,1,32,16,'RoundingMethod','Zero','OverflowAction','Saturate');

t1 = o1/fi(i1,'RoundingMethod','Zero','OverflowAction','Saturate');
t2 = (o2-o1)/fi(i2-i1,'RoundingMethod','Zero','OverflowAction','Saturate');
t3 = (255-o2)/fi(255-i2,'RoundingMethod','Zero','OverflowAction','Saturate');

t1f = fi(t1, 0, 32, 24);
t2f = fi(t2, 0, 32, 24);
t3f = fi(t3, 0, 32, 24);

T = zeros(1,256, 'uint8');
T = hdl.iteratorfun(@createTable, T, T, i1, i2, t1f, t2f, t3f, o1, o2);
```

```
[r_out, g_out, b_out] = hdl.npufun(@table_lookup, [1,1], r, g, b, 'KernelArg', T);

end

function count = hist_kernel_fcn(pix, count, idx) %#ok<*INUSD>

count(pix+1) = count(pix+1) +1;

end

function cdf = cdf_compute(hist, cdf, idx)

if idx > 1
    cdf(idx) = cdf(idx-cast(1,'like',idx)) + hist;
else
    cdf(1) = hist;
end

end

function i1i2 = find_break_points(cdf, i1i2, idx, p1, p2)
    if (cdf <= p1)
        i1i2(1) = i1i2(1) + 1;
    end

    if (cdf >= p2)
        i1i2(2) = i1i2(2) + 1;
    end
end

function T = createTable(unused, T, idx, i1, i2, t1f, t2f, t3f, o1, o2)

    is = uint16(idx);
    if is <= i1+1
        T(idx) = uint8(t1f*(is-1));
    elseif is >= i1+2 && is <= i2+1
        T(idx) = uint8(((t2f)*(is-1))-((t2f)*i1)+o1);
    else
        T(idx) = uint8(((t3f)*(is-1))-((t3f)*i2)+o2);
    end

end

function [r_out, g_out, b_out] = table_lookup(r, g, b, T)
    out1=T(r);
    out2=T(g);
    out3=T(b);

    r_out = uint8(out1);
    g_out = uint8(out2);
    b_out = uint8(out3);
end
```

Run the Model

The model uses 2-D matrices as inputs to the DUT. These inputs signals are the separated R, G, and B components of the input image. Each input signal is a frame input matrix composed of 240x320 pixels. Simulate the model to see the frame size and simulation results.

```
sim("hdlFrame_FogRectification_2D_MLFB");
```

Generate HDL Code

Generate synthesizable HDL code by using the frame-to-sample conversion. Set the HDL block property `ConvertToSamples` on the Inport blocks of the DUT that connect to the R, G, B input signals to convert the input signals from frame-based to sample-based inputs.

```
hdlset_param('hdlFrame_FogRectification_2D_MLFB/DUT/r_in', 'ConvertToSamples', 'on');
hdlset_param('hdlFrame_FogRectification_2D_MLFB/DUT/g_in', 'ConvertToSamples', 'on');
hdlset_param('hdlFrame_FogRectification_2D_MLFB/DUT/b_in', 'ConvertToSamples', 'on');
```

For the MATLAB Function block that contains the fog rectification algorithm, set the HDL block property `Architecture` to `MATLAB Datapath`. Enable the frame-to-sample conversion optimization and generate HDL code using the `makehdl` command. For more information on the frame-to-sample conversion optimization, see “HDL Code Generation from Frame-Based Algorithms” on page 22-2.

```
hdlset_param('hdlFrame_FogRectification_2D_MLFB/DUT/MATLAB Function', 'Architecture', 'MATLAB Da
hdlset_param('hdlFrame_FogRectification_2D_MLFB', 'FrameToSampleConversion', 'on')
```

```
makehdl('hdlFrame_FogRectification_2D_MLFB/DUT')
```

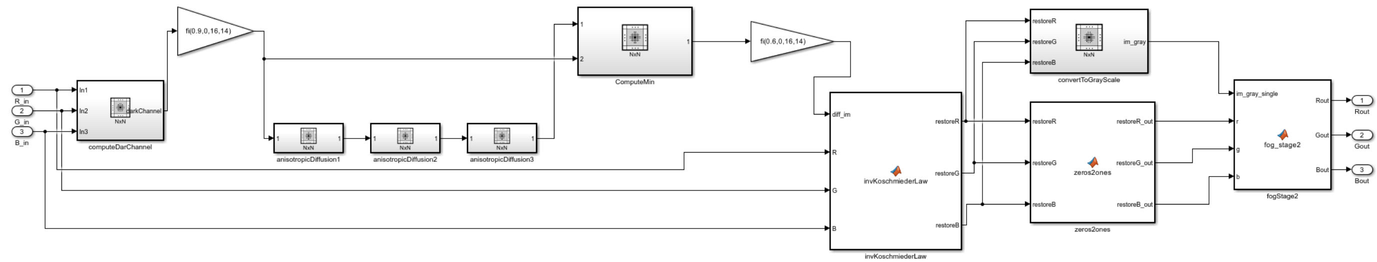
The frame-to-sample conversion separates the frame-based inputs into sample, valid, and ready signals for a sample-based hardware-targeted interface.

Model Algorithm by Using Neighborhood Processing Subsystem and MATLAB Function Blocks

In the `hdlFrame_FogRectification_2D_SLBlock` model, the DUT models the fog rectification algorithm using multiple Neighborhood Processing Subsystem and MATLAB Function blocks that use the `hdl.npufun` function. The MATLAB Function block `fogStage2` and the `hdl.iteratorfun` function model the reduction and iterator functions. Separating the fog rectification algorithm into various Simulink blocks enables you to visualize and model the different aspects of the fog rectification in a more modular way than with a single MATLAB Function block.

Open the `hdlFrame_FogRectification_2D_SLBlock/DUT` subsystem to see the fog rectification algorithm.

```
load_system("hdlFrame_FogRectification_2D_SLBlock");
open_system("hdlFrame_FogRectification_2D_SLBlock/DUT");
```



Run the Model

The model uses 2-D matrices as inputs to the DUT. These inputs signals are the separated R, G, and B components of the input image. Each input signal is a frame input matrix composed of 240x320 pixels. Simulate the model to see the frame size and simulation results.

```
sim("hdlFrame_FogRectification_2D_SLBlock");
```

Although the `hdlFrame_FogRectification_2D_SLBlock` and `hdlFrame_FogRectification_2D_MLFB` models differ in design, the output is the same. Both models can also generate synthesizable HDL code for sample-based hardware.

Generate HDL Code

Generate synthesizable HDL code by using the frame-to-sample conversion. Set the HDL block property `ConvertToSamples` on the Inport blocks of the DUT that connect to the R, G, B input signals to convert the input signals from frame-based to sample-based inputs.

```
hdlset_param('hdlFrame_FogRectification_2D_SLBlock/DUT/R_in', 'ConvertToSamples', 'on');
hdlset_param('hdlFrame_FogRectification_2D_SLBlock/DUT/G_in', 'ConvertToSamples', 'on');
hdlset_param('hdlFrame_FogRectification_2D_SLBlock/DUT/B_in', 'ConvertToSamples', 'on');
```

Enable the frame-to-sample conversion optimization and generate HDL code using the `makehdl` command.

```
hdlset_param('hdlFrame_FogRectification_2D_SLBlock', 'FrameToSampleConversion', 'on')
```

```
makehdl('hdlFrame_FogRectification_2D_SLBlock/DUT')
```

The frame-to-sample conversion separates the frame-based inputs into sample, valid, and ready signals for a sample-based hardware-targeted interface.

Model Algorithm and Generate HDL Code by Using a MATLAB Function

You can also generate HDL code directly from MATLAB functions by using the MATLAB-to-HDL workflow. For more information on the MATLAB-to-HDL workflow, see “Generate HDL Code from MATLAB Code Using the Command Line Interface”. In this example, the MATLAB function `fog_rectification` contains the MATLAB code from the MATLAB Function block in the `hdlFrame_FogRectification_2D_MLFB/DUT` subsystem.

```
open("fog_rectification");
```

Open the test bench function, `fog_rectification_tb`, to see the input fog rectification image separated into the R,G, and B components.

```
open("fog_rectification_tb");
```

The `fog_rectification_tb` script tests the function by inputting the original fog rectification image and splitting the input signal into the R, G, and B frame-based components of the image.

The test bench uses code generation to run the algorithm faster. To specify the input types and generate HDL code quickly without running the test bench, use the `-args` option.

To generate HDL code from a MATLAB function, create a `coder.HdlConfig` object, `hdlcfg`. Set the `DesignFunctionName` property to the `fog_rectification` function.

```
hdlcfg = coder.config('hdl');
hdlcfg.DesignFunctionName = 'fog_rectification';
```

To enable frame-to-sample conversion, first enable AggressiveDataflowConversion. This property transforms the control flow algorithm of the MATLAB code in the MATLAB function to a dataflow representation that is used for the frame-to-sample conversion optimization.

```
hdlcfg.AggressiveDataflowConversion = true;
```

Enable frame-to-sample conversion for the coder.HdlConfig object.

```
hdlcfg.FrameToSampleConversion = true;
```

Specify the input image and separate the image into its R, G, and B frame-based input components.

```
I = imread('inputFogRectification.png');
R = I(:,:,1);
G = I(:,:,2);
B = I(:,:,3);
```

Generate HDL code by using the codegen function. Specify the input arguments in the codegen command by using the -args option.

```
codegen -config hdlcfg -args { R,G,B }
```

See Also

hdl.npufun | hdl.iteratorfun

Related Examples

- “Deploy a Frame-Based Model with AXI4-Stream Interfaces” on page 40-378
- “Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods” on page 22-17
- “Use Sample-Based Inputs and Frame-Based Inputs in an Algorithm” on page 22-22

More About

- “HDL Code Generation from Frame-Based Algorithms” on page 22-2

Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods

If you have a frame-based model, you can model an optical flow algorithm by using multiple neighborhood processing methods. In this example, you use the HDL Coder™ frame-to-sample conversion optimization to generate synthesizable pixel-based HDL code from two frame-based models that demonstrate different modeling patterns.

This example shows two different modeling patterns for a neighborhood processing algorithm for optical flow. One of the patterns uses a MATLAB Function block with the neighborhood processing function `hdl.npufun`. The other pattern uses the Neighborhood Processing Subsystem block.

Model Algorithm with MATLAB Function Block

In the `hdlFrame_OpticalFlow_LK_MLFB` model, there is a single MATLAB Function block inside the device under test (DUT) that contains neighborhood processing operations needed for the Lucas Kanade method for optical flow. To generate synthesizable HDL code from the frame-based model, HDL Coder uses the frame-to-sample conversion supported function, `hdl.npufun` to create a streaming sample-based neighborhood processing algorithm. Using this function in a frame-based model enables you to contain the entire image processing algorithm in a single MATLAB Function block.

Open the `hdlFrame_OpticalFlow_LK_MLFB/DUT_LK/MATLAB Function LK` subsystem to see the optical flow algorithm.

```
load_system("hdlFrame_OpticalFlow_LK_MLFB");
open_system("hdlFrame_OpticalFlow_LK_MLFB/DUT_LK/MATLAB Function LK");
```

```
function flowVector  = opticalFlowForHDL_lk(I, Idelay)
% Implements the Lucas Kanade method for optical flow

%   Copyright 2021-2023 The MathWorks, Inc.

Ix = hdl.npufun(@imfilter_ComputIx, [1 5], I);
Iy = hdl.npufun(@imfilter_ComputIy, [5 1], I);
It = I - Idelay;

WIxx = hdl.npufun(@imfilter_kernel, [5 5], Ix.*Ix);
WIxy = hdl.npufun(@imfilter_kernel, [5 5], Ix.*Iy);
WIyy = hdl.npufun(@imfilter_kernel, [5 5], Iy.*Iy);
WIxt = hdl.npufun(@imfilter_kernel, [5 5], Ix.*It);
WIyt = hdl.npufun(@imfilter_kernel, [5 5], Iy.*It);

[Vx, Vy] = hdl.npufun(@calc_roots_pixel, [1 1], WIxx, WIxy, WIyy, WIxt, WIyt);
flowVector = complex(Vx,Vy);

end

function WI_1x1 = imfilter_ComputIx(I_1x5)

coder.inline('always');

d5 = [-1 8 0 -8 1]/12;
```

```
d5 = fliplr(d5(:)');
[WI_1x1] = sum(d5 .* I_1x5);

end

function WI_1x1 = imfilter_ComputIy(I_5x1)

coder.inline('always');

d5 = [-1 8 0 -8 1]/12;
d5 = flipud(d5(:));
[WI_1x1] = sum(d5(:) .* I_5x1);

end

function WI_1x1 = imfilter_kernel(I_5x5)

coder.inline('always');

p5 = [1 4 6 4 1]/16;
W = p5(:)*p5(:)';
[WI_1x1] = sum(W(:) .* I_5x5(:));

end

function [VxPix, VyPix] = calc_roots_pixel(aPix, bPix, cPix, ...
    txPix, tyPix)

coder.inline('always');

r = aPix + cPix;
s = sqrt(single(4 * bPix * bPix + (aPix - cPix) * (aPix - cPix)));
eig1 = (single(r) + s);
eig2 = (single(r) - s);

vx = single(0);
vy = single(0);
thresh = 0.0039;
thresh2 = 0;

if (eig1 > thresh && eig2 > thresh)
    d = bPix * bPix - aPix * cPix;
    iDelta = 1/d;
    vx =  single( -(tyPix * bPix - txPix * cPix)*iDelta);
    vy =  single( -(txPix * bPix - aPix * tyPix)*iDelta);
elseif (eig1 > thresh && eig2 < thresh)
    rr = bPix+aPix;
    cc = cPix+bPix;
    norm = cc*cc + rr*rr;

    if (norm > thresh2)
        invnorm = 1/norm;
        tmp = -(txPix + tyPix) * invnorm;
        vx =  single(rr * tmp);
        vy =  single(cc * tmp);
    end
end
```

```
VxPix = vx;
VyPix = vy;

end
```

Run the Model

The input video is split into a previous frame and current frame. The input ports of the DUT, `CurrFrame` and `PrevFrame`, connect to the 2-D matrices that contain the data for the current frame and previous frame. Each input signal is a frame composed of 360x640 pixels. Simulate the model to see the frame size and simulation results of the optical flow output.

```
sim("hdlFrame_OpticalFlow_LK_MLFB");
```

Generate HDL Code

Generate synthesizable HDL code by using the frame-to-sample conversion. Set the HDL block property `ConvertToSamples` on the Inport blocks of the DUT that connect to the frame-input signals to convert the input signals from frame-based to sample-based inputs.

```
hdlset_param('hdlFrame_OpticalFlow_LK_MLFB/DUT_LK/PrevFrame','ConvertToSamples','on');
hdlset_param('hdlFrame_OpticalFlow_LK_MLFB/DUT_LK/CurrFrame','ConvertToSamples','on');
```

For the MATLAB Function block that contains the optical flow algorithm, set the HDL block property `Architecture` to `MATLAB Datapath`. Enable the frame-to-sample conversion optimization and generate HDL code using the `makehdl` command. For more information on the frame-to-sample conversion optimization, see “HDL Code Generation from Frame-Based Algorithms” on page 22-2.

```
hdlset_param('hdlFrame_OpticalFlow_LK_MLFB/DUT_LK/MATLAB Function LK','Architecture','MATLAB Data
hdlset_param('hdlFrame_OpticalFlow_LK_MLFB','FrameToSampleConversion','on');

makehdl('hdlFrame_OpticalFlow_LK_MLFB/DUT_LK');
```

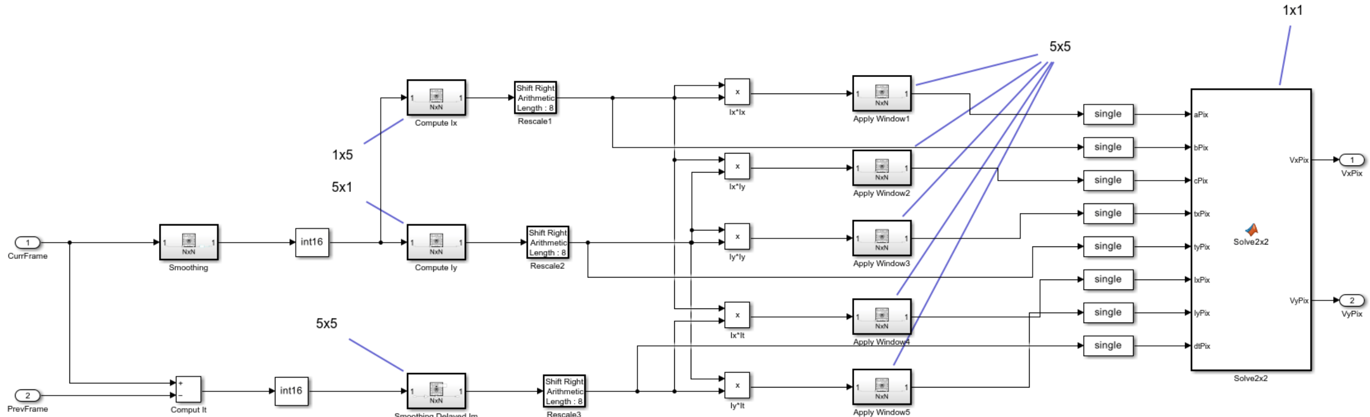
The frame-to-sample conversion separates the frame-based inputs into sample, valid, and ready signals for a sample-based hardware-targeted interface.

Model Algorithm with the Neighborhood Processing Subsystem Block

In the `hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem` model, the DUT contains multiple Neighborhood Processing Subsystem blocks that contain the neighborhood processing operations needed for the Lucas Kanade optical flow algorithm. Separating the optical flow algorithm into various Simulink blocks enables you to visualize and model the different aspects of the algorithm in a more modular way than with a single MATLAB Function block.

Open the `hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem/DUT` subsystem to see the optical flow algorithm.

```
load_system("hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem");
open_system("hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem/DUT");
```



Run the Model

The input video is split into a previous frame and current frame. The input ports of the DUT, `CurrFrame` and `PrevFrame`, connect to the 2-D matrices that contain the data for the current frame and previous frame. Each input signal is a frame composed of 360x640 pixels. Simulate the model to see the frame size and simulation results of the optical flow output.

```
sim("hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem");
```

Although the `hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem` and `hdlFrame_OpticalFlow_LK_MLFB` models differ in design, the output is the same. Both models can also generate synthesizable HDL code for sample-based hardware.

Generate HDL Code

Generate synthesizable HDL code by using the frame-to-sample conversion. Set the HDL block property `ConvertToSamples` on the Inport blocks of the DUT that connect to the frame-input signals to convert the input signals from frame-based to sample-based inputs.

```
hdlset_param('hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem/DUT/PrevFrame','ConvertToSample
hdlset_param('hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem/DUT/CurrFrame','ConvertToSample
```

Enable the frame-to-sample conversion optimization and generate HDL code using the `makehdl` command.

```
hdlset_param('hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem','FrameToSampleConversion','on
```

```
makehdl('hdlFrame_OpticalFlow_LK_NeighborProcessingSubsystem/DUT');
```

The frame-to-sample conversion separates the frame-based inputs into sample, valid, and ready signals for a sample-based hardware-targeted interface.

See Also

`hdl.npufun`

Related Examples

- "Deploy a Frame-Based Model with AXI4-Stream Interfaces" on page 40-378
- "Use Neighborhood, Reduction, and Iterator Patterns with a Frame-Based Model or Function for HDL Code Generation" on page 22-10

- “Use Sample-Based Inputs and Frame-Based Inputs in an Algorithm” on page 22-22

More About

- “HDL Code Generation from Frame-Based Algorithms” on page 22-2

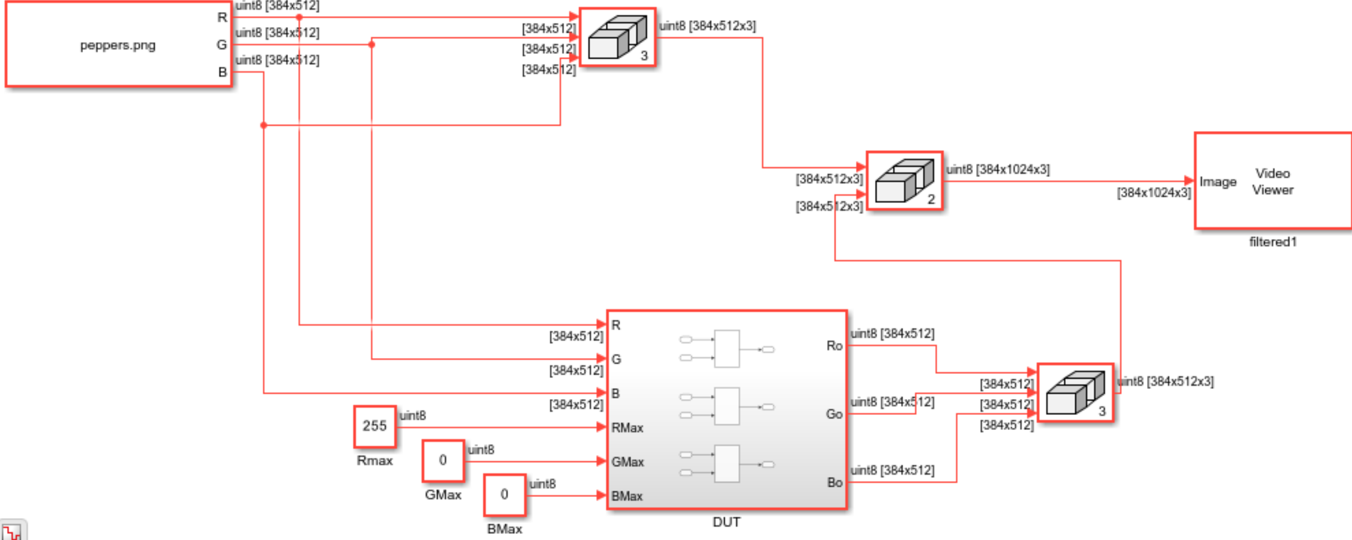
Use Sample-Based Inputs and Frame-Based Inputs in an Algorithm

You can use matrix or frame-based signals with scalar or sample-based signals in the same algorithm when you use the frame-to-sample conversion operation. You can use non-streamed scalar inputs with streamed frame inputs to generate synthesizable HDL code for algorithms where you apply a threshold value or gain to an incoming streamed signal. This example uses an RGB image as an input and applies a color filter based on the threshold signals.

Model Frame Input and Sample Input Interaction

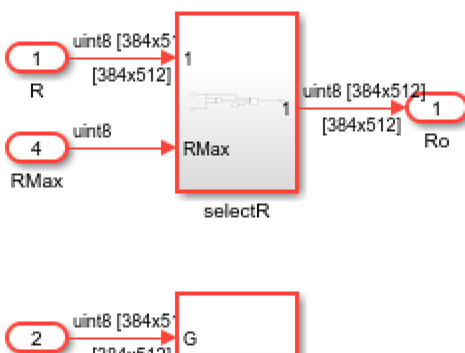
This example takes an RGB input image, splits the image into three individual RGB inputs signals, and applies a color filter on each signal based on the threshold signals specific to each color input. The threshold signals are inputs to the DUT as non-streamed scalar values.

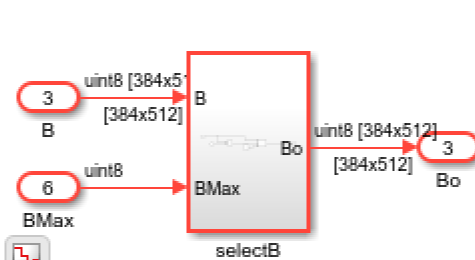
```
open_system("hdlFrame_RGBFilter");
set_param('hdlFrame_RGBFilter', 'SimulationCommand', 'update');
```



Open the DUT subsystem to see the blocks that perform the filtering operation on the RGB frame inputs.

```
open_system('hdlFrame_RGBFilter/DUT');
```





Run the Model

The model uses 2-D matrices as inputs to the DUT. These inputs signals are the separated R,G,B components of the input image. Each streamed-input signal is a frame input matrix composed of 384x512 pixels. Each non-streamed threshold input is a `uint8` scalar value. Simulate the model to see the color-filtered output image.

```
sim("hdlFrame_RGBFilter");
```

Generate HDL Code

Generate synthesizable HDL code by using the frame-to-sample conversion. Set the HDL block property `ConvertToSamples` on the Inport blocks of the DUT that connect to the R, G, B input signals to convert the input signals from frame-based to sample-based inputs. You do not need to enable the `ConvertToSamples` property on the threshold inputs because they are scalar sample-based values.

```
hdlset_param('hdlFrame_RGBFilter/DUT/R', 'ConvertToSamples', 'on');
hdlset_param('hdlFrame_RGBFilter/DUT/G', 'ConvertToSamples', 'on');
hdlset_param('hdlFrame_RGBFilter/DUT/B', 'ConvertToSamples', 'on');
```

Enable the frame-to-sample conversion optimization and generate HDL code using the `makehdl` command. For more information on the frame-to-sample conversion optimization, see “HDL Code Generation from Frame-Based Algorithms” on page 22-2.

```
hdlset_param('hdlFrame_RGBFilter', 'FrameToSampleConversion', 'on')
```

```
makehdl('hdlFrame_RGBFilter/DUT')
```

The frame-to-sample conversion separates the frame-based inputs into sample, valid, and ready signals for a sample-based hardware-targeted interface.

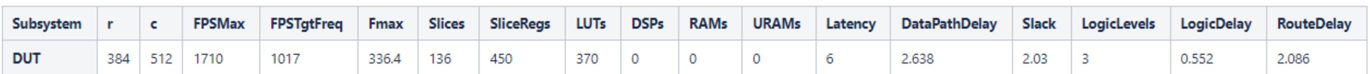
Perform FPGA Synthesis and Analysis

In the HDL Workflow Advisor, perform FPGA synthesis using the generic ASIC/FPGA workflow. To generate HDL code and run synthesis on your design using the HDL Workflow Advisor, see “HDL Code Generation and FPGA Synthesis from Simulink Model”.

Use these settings located in the **HDL Code Generation > Target** pane in the Model Configuration Parameters dialog box:

- **Synthesis tool** set to `Xilinx Vivado`
- **Family** set to `Zynq`
- **Device** set to `xc7z045`
- **Package** set to `ffg900`
- **Speed** set to `-2`
- **Target Frequency (MHz)** set to `200`

This table shows the synthesis results for the `DUT` subsystem in `hdlFrame_RBGFilter`. The results report the frames per seconds (FPS) for each individual RGB channel, which assumes that each RGB input is streamed independently of one another.



Subsystem	r	c	FPSMax	FPSTgtFreq	Fmax	Slices	SliceRegs	LUTs	DSPs	RAMs	URAMs	Latency	DataPathDelay	Slack	LogicLevels	LogicDelay	RouteDelay
DUT	384	512	1710	1017	336.4	136	450	370	0	0	0	6	2.638	2.03	3	0.552	2.086

The synthesis results show a positive slack of 2.03 ns, which indicates that the timing constraints are met. The `r` and `c` columns correspond to the number of rows and columns of the frame, respectively. The `FPSMax` and `FPSTgtFreq` columns correspond to the average FPS at the output when using the maximum achievable frequency `Fmax` and the Target Frequency `200 MHz`, respectively. To calculate the average FPS, use this equation:

$$\mathrm{FPS} = \frac{\mathrm{Frequency\ (Hz)}}{r \cdot c}$$

See Also

Related Examples

- “Deploy a Frame-Based Model with AXI4-Stream Interfaces” on page 40-378
- “Use Neighborhood, Reduction, and Iterator Patterns with a Frame-Based Model or Function for HDL Code Generation” on page 22-10
- “Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods” on page 22-17

More About

- “HDL Code Generation from Frame-Based Algorithms” on page 22-2

Synthesize Code for Frame-Based Model

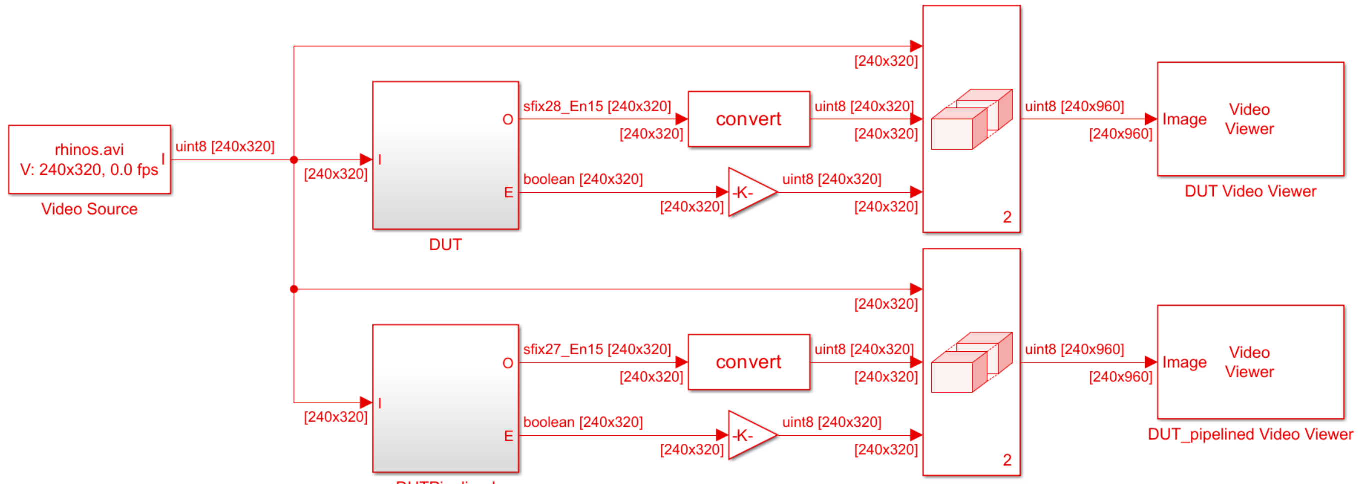
This example shows how you can generate HDL code for a Sobel edge detection frame algorithm and calculate the frames per second of your design by synthesizing the generated code with the Simulink® HDL Workflow Advisor.

In this example, you can optimize the speed of the design by either manually pipelining the algorithm or by applying distributed pipelining.

Sobel Edge Detection Model

Open the model to see a frame-based implementation of the Sobel edge detection algorithm.

```
open_system('FrameBasedEdgeDetectionForHDLCodeGenExample')
set_param('FrameBasedEdgeDetectionForHDLCodeGenExample','SimulationCommand','Update')
```



The model consists of two designs under test (DUT) and a testbench. The `DUT` subsystem contains a MATLAB® Function block that implements a Sobel edge detection algorithm using `hdl.npufun`.

```
function [O, E] = edgeDetectionAndOverlay(I)

E = hdl.npufun(@sobel_kernel, [3 3], I);
O = hdl.npufun(@mix_kernel, [1 1], E, I);

end

function e = sobel_kernel(in)

u = fi(in);
hGrad = u(1) + fi(2)*u(2) + u(3) - (u(7) + fi(2)*u(8) + u(9));
vGrad = u(1) + fi(2)*u(4) + u(7) - (u(3) + fi(2)*u(6) + u(9));

hGrad = bitshift(hGrad, -3); % Divide by 8
vGrad = bitshift(vGrad, -3); % Divide by 8
```

```
thresholdValueSq = fi(49); % Threshold parameter
e = (hGrad*hGrad + vGrad*vGrad) > thresholdValueSq;

end

function O = mix_kernel(E, I)

alpha = fi(0.8); % Parameter for combining images
scaleE = E*fi(255,0,8,0);
O = scaleE * (fi(1)-alpha) + I*alpha;

end
```

The `DUTPipelined` subsystem contains a MATLAB Function block with a manually pipelined version of the previous algorithm that uses the `coder.hdl.pipeline` pragma. For example, you can modify the `mix_kernel` function to insert pipeline registers for the overlay operation.

```
function O = mix_kernel(E, I)

alpha = fi(0.8); % Parameter for combining images
scaleE = E*fi(255,0,8,0);
scaleEdelay = coder.hdl.pipeline(scaleE,2);
O1 = scaleEdelay*(1-alpha);
O1delay = coder.hdl.pipeline(O1,2);
O2 = I*alpha;
O2delay = coder.hdl.pipeline(O2,4);
O = O1delay + O2delay;

end
```

Perform FPGA Synthesis and Analysis

Use the HDL Workflow Advisor to synthesize the DUTs and compare the resources and timing achieved by each of the implementations. To generate HDL code and run synthesis on your design using the HDL Workflow Advisor, see “HDL Code Generation and FPGA Synthesis from Simulink Model”.

This table shows the results achieved for each DUT subsystem when you use these settings in the HDL Workflow Advisor:

- **Synthesis Tool:** `Xilinx Vivado`
- **Family:** `Zynq`
- **Device:** `xc7z045`
- **Package:** `ffg900`
- **Speed:** `-2`
- **Target Frequency:** `200`

The `FPSMax` and `FPSTgtFreq` columns correspond to the average frames per second (FPS) at the output when using the maximum achievable frequency (`Fmax`) and the Target Frequency (`200 MHz`), respectively, and `r` and `c` correspond to the number of rows and columns of the frame. To calculate the average frames per second, you can use this equation:

$$FPS = \frac{\text{Frequency (Hz)}}{r \cdot c}.$$

DUT	r	c	FPSMax	FPSTgtFreq	Fmax	Slices	SliceRegs	LUTs	DSPs	RAMs	Latency	DataPathDelay	Slack	LogicLevels	LogicDelay	RouteDelay
DUT	240	320	926	NA	72.84	274	530	712	10	1.5	8	12.541	-9.058	25	7.01	5.531
DUT_pipelined	240	320	3549	2604	272.63	302	1006	724	10	1.5	23	3.316	1.332	4	0.496	2.82

Converting a frame-based algorithm to a sample-based algorithm requires more time to process each pixel than the frame-based version. This conversion adds additional latency between the input and output frames. In this example, the first pixel of the manually pipelined DUT output frame is available after an initial latency of 321 valid input pixels, plus the latency of 23 clock cycles reported in the table. For more information, see “HDL Code Generation from Frame-Based Algorithms” on page 22-2.

Optimized Speed by Using Speed Optimizations

You can use the distributed pipelining and the adaptive pipelining optimizations from HDL Coder™ with the frame-to-sample optimization to reduce the critical path of the design and achieve higher clock speed hardware for the `DUT` subsystem. Distributed pipelining moves existing design delays and pipeline registers in your design to reduce the critical path and adaptive pipelining inserts pipeline registers to blocks in the design that could result in area or speed optimizations. For this example, add input and output pipelines for distributed pipelining to redistribute and enable adaptive pipelining to obtain an optimal result. For more information on distributed pipelining and adaptive pipelining, see “Distributed Pipelining” on page 21-130 and “Adaptive Pipelining” on page 21-181 respectively.

Enable distributed pipelining, adaptive pipelining, and set the MATLAB Function block input and output pipeline registers to 16 by entering:

```
hdlset_param('FrameBasedEdgeDetectionForHDLCodeGenExample', 'DistributedPipelining', 'on');
hdlset_param('FrameBasedEdgeDetectionForHDLCodeGenExample', 'AdaptivePipelining', 'on');
hdlset_param('FrameBasedEdgeDetectionForHDLCodeGenExample/DUT/MATLAB Function', 'DistributedPipe
hdlset_param('FrameBasedEdgeDetectionForHDLCodeGenExample/DUT/MATLAB Function', 'OutputPipeline'
hdlset_param('FrameBasedEdgeDetectionForHDLCodeGenExample/DUT/MATLAB Function', 'InputPipeline',
```

This table shows the resources and timing achieved when using distributed pipelining for different video formats. The highlighted row corresponds to the video format used in the previous section.

Format	r	c	FPSMax	FPSTgtFreq	Fmax	Slices	SliceRegs	LUTs	DSPs	RAMs	Latency	DataPathDelay	Slack	LogicLevels	LogicDelay	RouteDelay
8KUHDTV	4320	7680	7	6	246.43	412	1155	847	10	10	23	3.001	0.942	0	3.001	0
4KUHDTV	2160	3840	29	24	246.43	403	1133	848	10	5	23	3.001	0.942	0	3.001	0
2KCinema	1080	2048	111	90	246.43	377	1118	829	10	3	23	3.001	0.942	0	3.001	0
1200p	1200	1600	118	96	246.43	377	1111	819	10	2.5	23	3.001	0.942	0	3.001	0
1080p	1080	1920	128	104	246.43	381	1111	850	10	2.5	23	3.001	0.942	0	3.001	0
1024p	1024	1280	188	152	246.43	380	1107	842	10	2.5	23	3.001	0.942	0	3.001	0
768p	768	1024	267	217	246.43	389	1104	843	10	2.5	23	3.001	0.942	0	3.001	0
720p	720	1280	313	254	246.43	360	1096	851	10	1.5	23	3.001	0.942	0	3.001	0
576p	576	720	594	482	246.43	378	1089	888	10	1.5	23	3.001	0.942	0	3.001	0
480pH	480	720	713	578	246.43	356	1085	885	10	1.5	23	3.001	0.942	0	3.001	0
480p	480	640	802	651	246.43	356	1085	869	10	1.5	23	3.001	0.942	0	3.001	0
240p	**240**	**320**	**3208**	**2604**	**246.43**	**362**	**1282**	**923**	**10**	**1.5**	**45**	**3.001**	**0.942**	**0**	**3.001**	**0**

For all video formats, the synthesis results show a clock frequency of 246.43 MHz, indicating that the timing constraints are met.

Optimized Speed by Modifying the Samples per Cycle

You can vary the video size and achieve higher FPS by using the **SamplesPerCycle** parameter in the Configuration Parameters window:

$$\mathrm{FPS} = \frac{\mathrm{Frequency\,(Hz)}}{r \cdot \left(\frac{c}{\mathrm{SamplesPerCycle}}\right)}.$$

This table shows the FPS obtained for the `DUTPipelined` subsystem (manually pipelined) when you vary the size of the frames and the Samples per cycle parameter.

Video Format	r	c	FPS @ 1,2,4,8 pixels per cycle			
			1	2	4	8
8KUHDTV	4320	7680	7	15	29	57
4KUHDTV	2160	3840	29	63	119	233
2KCinema	1080	2048	111	226	451	853
1200p	1200	1600	118	253	473	912
1080p	1080	1920	128	270	501	1033
1024p	1024	1280	188	385	739	1413
768p	768	1024	255	549	1055	2061
720p	720	1280	313	673	1249	2432
576p	576	720	594	1191	2341	4430
480pH	480	720	713	1541	3013	5325
480p	480	640	802	1729	3175	5940
240p	240	320	3549	6636	12496	24191

See Also

Related Examples

- "Deploy a Frame-Based Model with AXI4-Stream Interfaces" on page 40-378

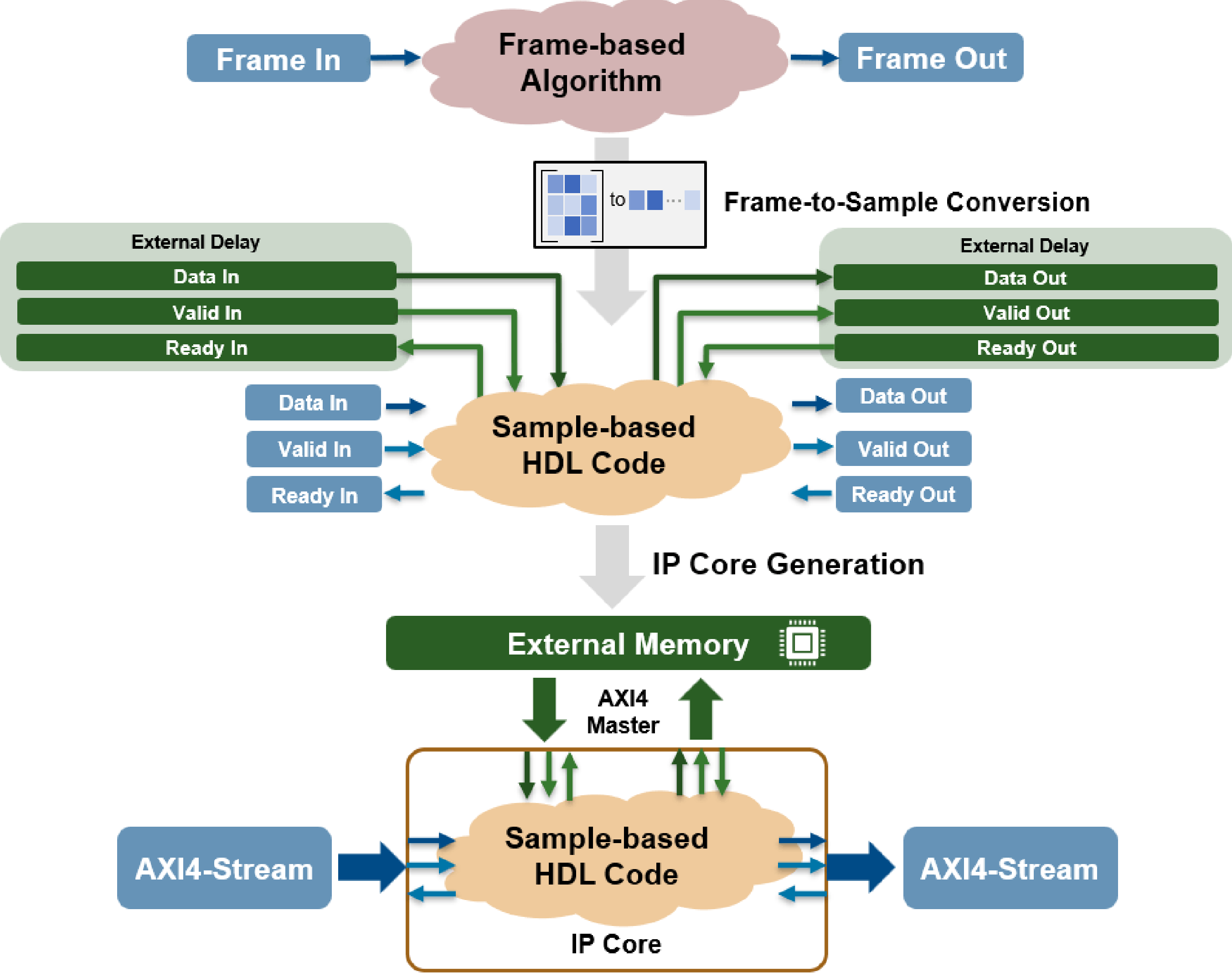
- “Use Neighborhood, Reduction, and Iterator Patterns with a Frame-Based Model or Function for HDL Code Generation” on page 22-10
- “Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods” on page 22-17

More About

- “HDL Code Generation from Frame-Based Algorithms” on page 22-2

Offload Large Delays from Frame-Based Models to External Memory

Frame-based algorithms often require storing large amounts of data in external memory for future processing. When you use the frame-to-sample conversion, HDL Coder™ transforms your frame-based algorithm into sample-based HDL code and generates additional ports to offload the delays that the design needs for pipeline computations. During the IP core generation, HDL Coder can map these ports to an AXI4 Master interface to store the data in external memory.



Offloading a large delay is also useful in signal processing algorithms that require a large amount of data to process the input signal. A common image processing application is histogram equalization, which requires building a histogram from an entire input frame in order to equalize the image. This example shows how to leverage the frame-to-sample optimization to generate a sample-based IP core with AXI4-Stream interfaces from a frame-based histogram equalization model.

To run this example, you must have the following software and hardware boards:

- HDL Coder Support Package for Xilinx® FPGA and SoC Devices.
- Xilinx Vivado®. To view the supported versions, see “HDL Language Support and Supported Third-Party Tools and Hardware”.
- Xilinx Zynq ZC706 Evaluation Kit.

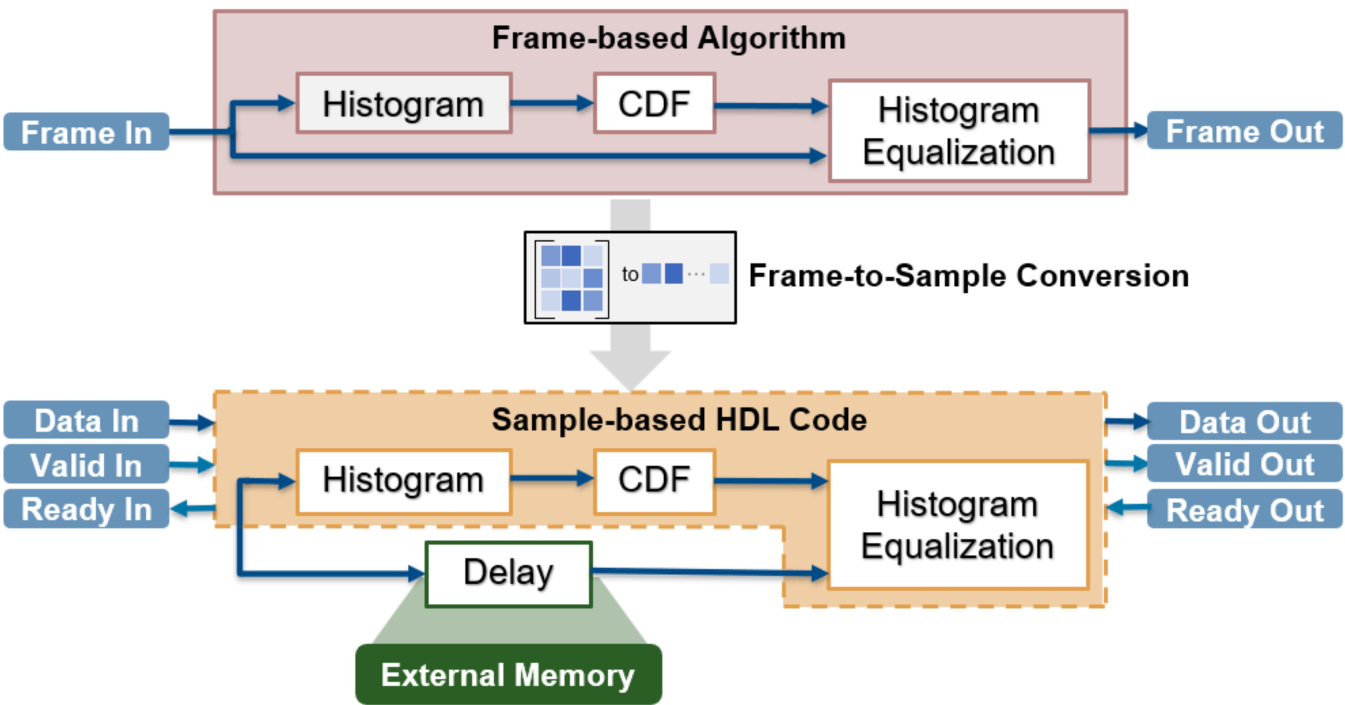
Model a Histogram Equalization Algorithm Using Iterative Operations

Open the `HistogramEq` MATLAB Function block in the `hdlFrame_Zynq_Histogram/DUT` subsystem to see the histogram equalization algorithm.

```
load_system('hdlFrame_Zynq_Histogram')
open_system('hdlFrame_Zynq_Histogram/DUT/HistogramEq')

function im_out = histeq(im_gray)

% Histogram calculation
hist = zeros(1, 256, 'uint16');
hist = hdl.iteratorfun(@hist_kernel_fcn, im_gray, hist);

[row,col] = size(im_gray);
factor = coder.const(fi(255/(row*col)));

% Cumulative distribution function calculation
cdf_init = zeros(1,256,'uint8');
cdf = hdl.iteratorfun(@cdf_compute, hist, cdf_init, factor);

% Equalize input frame: replace the value from look up table
im_out = hdl.npufun(@table_lookup, [1,1], im_gray, 'NonSampleInput', cdf);

end

function count = hist_kernel_fcn(pix, count, idx) %#ok<INUSD>

count(pix+1) = count(pix+1) +1;

end

function cdf = cdf_compute(hist, cdf, idx, factor)

if idx > 1
    cdf(idx) = uint8(cdf(idx-cast(1,'like',idx)) + hist*factor);
else
    cdf(1) = uint8(hist*factor);
end

end

function out = table_lookup(in, cdf)
    out = cdf(in);
end
```

In the `hdlFrame_Zynq_Histogram` model, there is a single MATLAB® Function block inside the device under test (DUT) that uses the `hdl.iteratorfun` function to compute the cumulative distribution function (CDF) of the incoming frame, and the `hdl.npufun` function to equalize the frame using the CDF result. For more information on modeling iterative and neighborhood operations in frame-based models, see “HDL Code Generation from Frame-Based Algorithms” on page 22-2.

When you use the frame-to-sample optimization and set the Delay size threshold for external memory (bits) configuration parameter to a specified threshold in kilobytes, HDL Coder generates sample-based HDL code from the frame-based algorithm and offloads delays greater than the threshold to external memory..



Generate HDL IP Core

When you generate an IP core for this model, you can connect the streaming I/O of your algorithm to a streaming interface. HDL Coder handles the external memory mapping process by generating the frame management logic and read and write controllers to write the delay to external memory using an AXI4 Master interface. The IP core can then write the incoming frame data to DDR memory and read the data to perform the equalization once the histogram has been calculated. This process reduces modeling and development time because HDL Coder handles the complex frame management of multiple delays in external memory and does not require you to model the simplified AXI4 master protocol to connect the IP core to external memory.

To generate an IP core from the frame-based DUT and deploy this design on the Zynq hardware:

1. Enable the frame-to-sample conversion:

```
hdlset_param('hdlFrame_Zynq_Histogram','FrameToSampleConversion','on');
```

2. Enable the HDL block property `ConvertToSamples` for the input image to be streamed, `ImageIn`:

```
hdlset_param('hdlFrame_Zynq_Histogram/DUT/ImageIn','ConvertToSamples','on');
```

3. To offload large delays to external memory outside of the FPGA, set the `DelaySizeThreshold` parameter to a delay size threshold in kilobytes. For this example, the delay needed for the histogram equalization algorithm is of similar size to the image, which is 262x216x8 or 56.6 kilobytes. To map

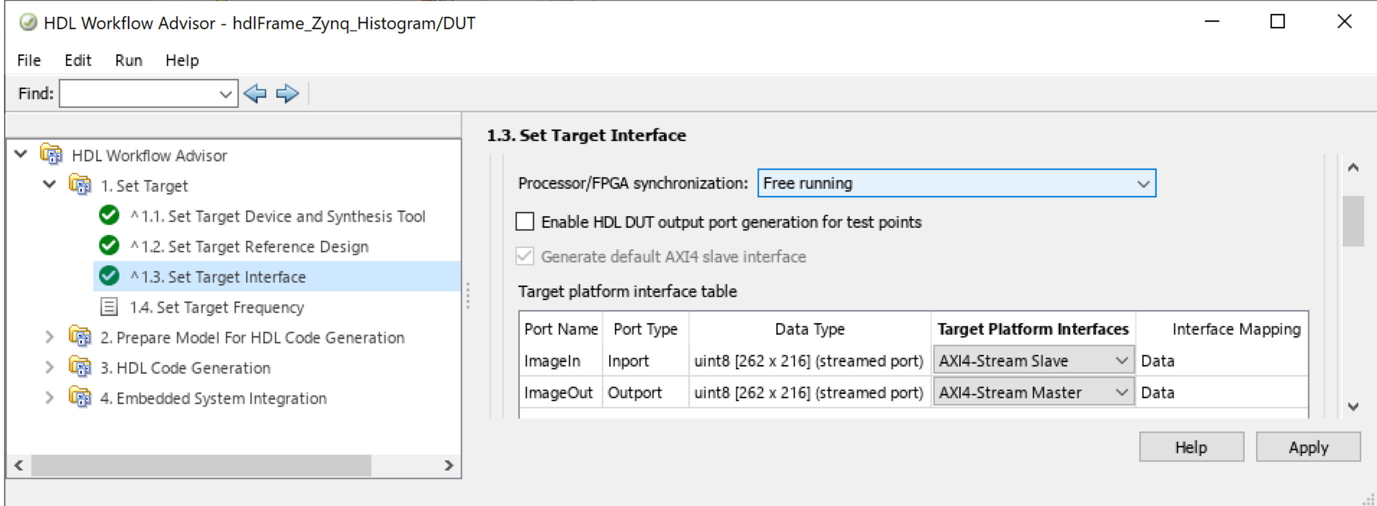
the large delay to external memory, set the parameter to a value lower than the image size in kilobytes. In this case, set the `DelaySizeThreshold` to 10 kilobytes.

```
hdlset_param('hdlFrame_Zynq_Histogram','DelaySizeThreshold',10)
```

4. Set up the Xilinx Vivado synthesis tool path by using the `hdlsetuptoolpath` command. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2022.1\bin\vivado.bat')
```

5. Open the HDL Workflow Advisor and generate an IP core from the DUT subsystem, `hdlFrame_Zynq_Histogram/DUT`. In task **1.1 Set Target Device and Synthesis Tool,** set **Target workflow** to `IP Core Generation` and **Target platform** to `Xilinx Zynq ZC706 evaluation kit`. In task **1.2 Set Target Reference Design**, set **Reference Design** to `Default System with External DDR3 memory access`. In task **1.3 Set Target Interface**, set **Target platform interface table** to the settings shown in this image.



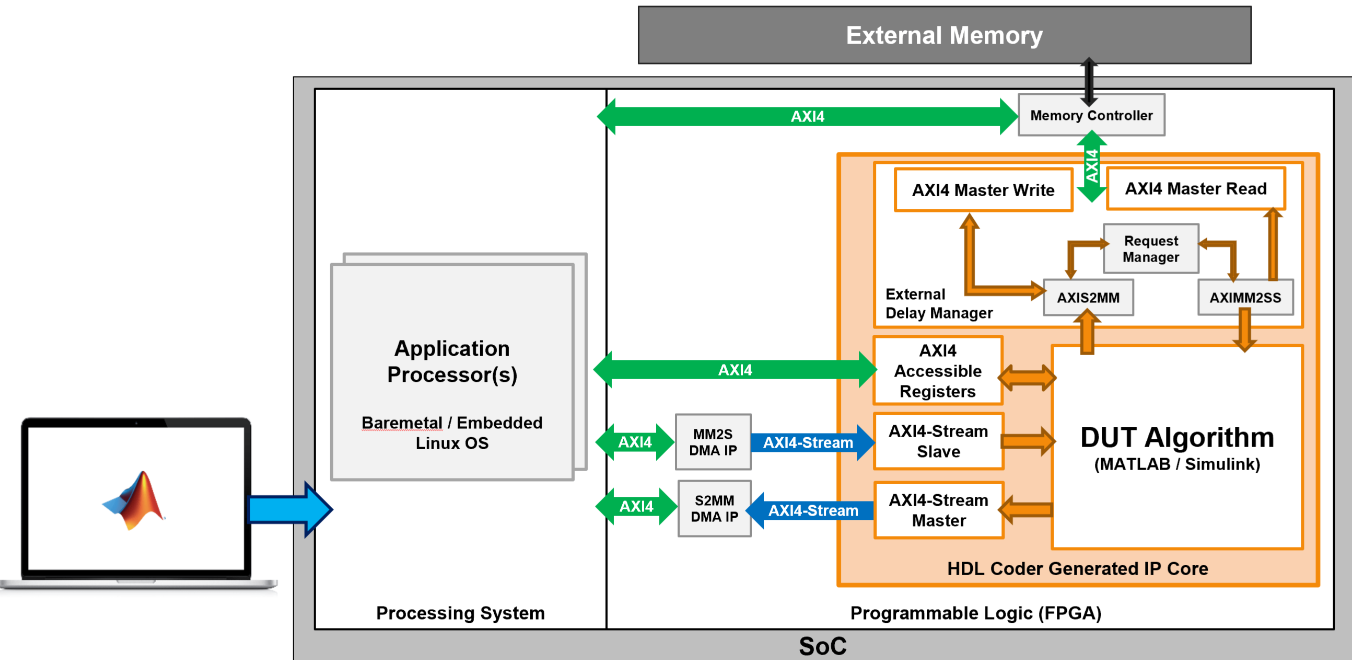
6. Right-click task **3.2 Generate RTL Code and IP Core** and select **Run to Selected Task** to generate the IP core. Because the frame-to-sample optimization is enabled, HDL Coder generates additional DUT ports to offload the necessary delays needed for the histogram calculation. During the IP core generation, HDL Coder maps these ports to an AXI4 Master interface to store the data in the DDR memory. You can find the register address mapping, the necessary frame size for the external delay, and other information about the IP core generated in the IP core report.

7. In task **4.2 Generate Software Interface**, select the **Generate host interface script** check box and click **Run this Task**. The HDL Workflow Advisor generates two MATLAB files in your current folder that you can use to prototype the generated IP core.

8. Right-click task **4.3 Build FPGA Bitstream** and select **Run to Selected Task** to generate the Vivado project and build the FPGA bitstream.

During the project creation, the generated DUT IP core is integrated into the `Default System with External DDR3 Memory Access` reference design. This reference design contains a Xilinx Memory Interface Generator IP, which communicates with the on-board external DDR3 memory on the ZC706 platform, and the `AXI Manager` IP, which enables MATLAB to control the DUT IP and initialize and verify the DDR memory content. The DMA IPs transfer AXI4-Stream data between the processing system and the FPGA.

To view the generated Vivado project, click the link in the result window in task **4.1 Create Project**. Open the Vivado block design. The generated reference design project looks similar to this architecture diagram.



9. After the bitstream generates, right-click task **4.4 Program Target Device** and click **Run this Task** to program the target device.

For a more information on IP core generation for Xilinx hardware, see “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98.

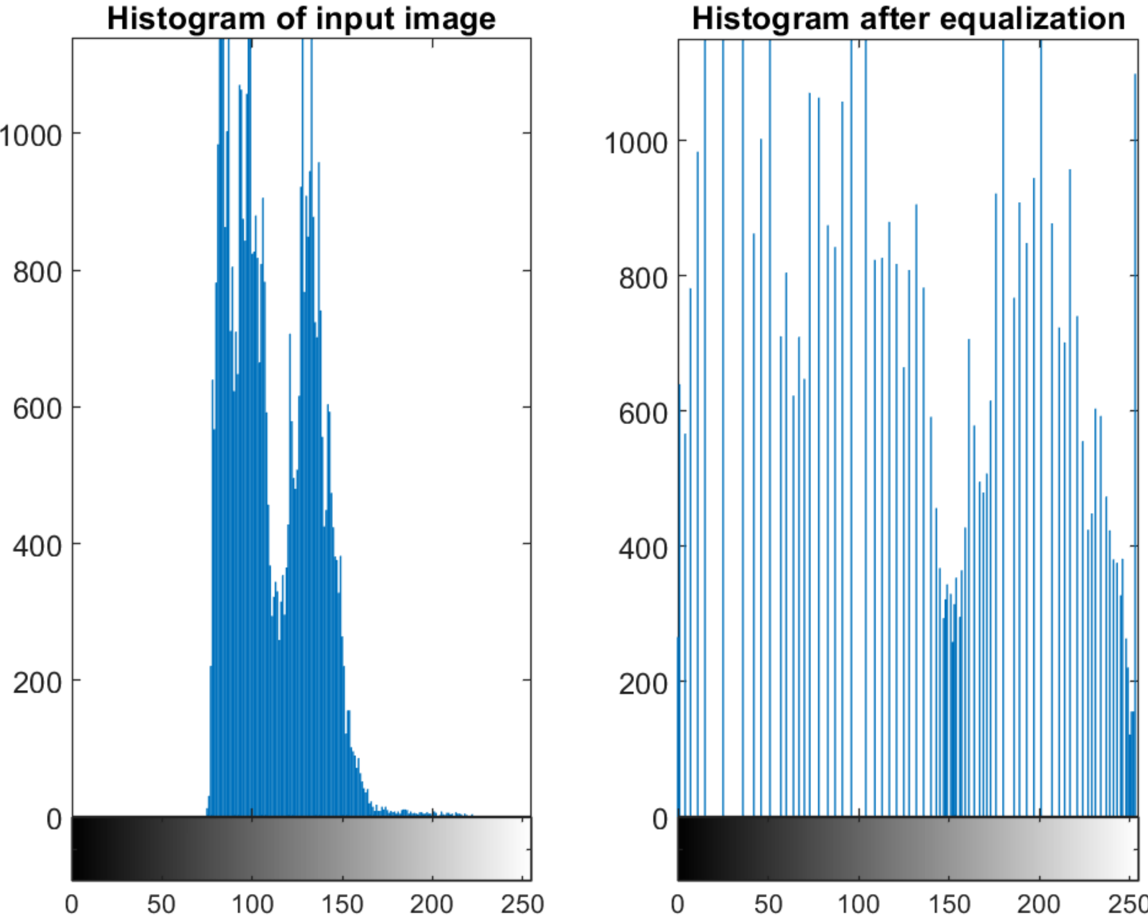
Run FPGA Implementation on Xilinx Zynq ZC706 Evaluation Kit

You can interact with the FPGA design by reading and writing data from MATLAB on the host computer as described in the Interact with FPGA Design from Host Computer section in “Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands” on page 39-241.

You can use these commands as a starting point to test the frame-based model deployed on the FPGA.

```
% load image
I = imread('hdlc_pout.tif');

%Write image to FPGA
wrValid1 = writePort(hFPGA, "ImageIn", I);
wrValid2 = writePort(hFPGA, "ImageIn", I);
% Read result from FPGA
[outputFrame1, rdValid1] = readPort(hFPGA, "ImageOut");
% Display result
figure
imagesc([I outputFrame1],[0 255]); colormap(gray)
title('(left) Input image, (right) Output image read from FPGA')
% Display histograms
figure
```

```
subplot(1,2,1);imhist(I,256);
title('Histogram of input image')
subplot(1,2,2);imhist(outputFrame1,256);
title('Histogram after equalization')
```



Limitations

When you generate an IP core from a frame-based algorithm and enable delay mapping to external memory, these limitations apply:

- You can map at most one large delay to external memory. If there are multiple large delays over the threshold set by the **Delay size threshold for external memory** parameter, the largest delay is mapped to external memory while the rest of the delays are mapped to memory on the FPGA.
- HDL Coder only maps FIFO blocks as delays generated during HDL code generation to external memory. Delay blocks created from optimizations that add pipelines to the generated model and code cannot be moved outside the DUT and mapped to external memory.

See Also

Related Examples

- "Deploy a Frame-Based Model with AXI4-Stream Interfaces" on page 40-378
- "Use Neighborhood, Reduction, and Iterator Patterns with a Frame-Based Model or Function for HDL Code Generation" on page 22-10

- “Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods” on page 22-17

More About

- “HDL Code Generation from Frame-Based Algorithms” on page 22-2
- “Model Design for AXI4-Stream Interface Generation” on page 40-14

Optimize Area Usage for Frame-Based Algorithms with Tall Array Inputs

This example shows how to generate area-efficient HDL code from a frame-based algorithm that has input data with significantly more rows than columns. This type of input data is common in audio and digital signal processing algorithms. You can use the frame-to-sample conversion optimization and adjust the order in which the input data is processed for your target hardware to reduce the amount of area needed for the algorithm. For more information on the frame-to-sample conversion optimization, see “HDL Code Generation from Frame-Based Algorithms” on page 22-2.

The Input processing order configuration parameter allows you to select the order in which the frame-to-sample optimization processes an incoming matrix. By selecting the input processing order, you can align the way your algorithm is processed with the type of hardware you are using or the application you are targeting. For example, if you have an input hardware sensor or an upstream component that requires the column-major ordering, you do not need to use an extra transpose. Similarly, if you have a digital signal processing algorithm that requires column-major ordering, you do not need to transpose the matrix to make the algorithm compatible with the frame-to-sample optimization.

Inspect the Model

For this example, the input data is daily data of average temperature over the course of 10 years, stored as 365 rows for each day by 10 columns for the 10 years. The algorithm calculates a moving average within each year, which requires computing the moving average along each column.

Load the input data `inputTemp` from the `inputData.mat` file.

```
load('inputData.mat');
```

Load the model and open the MATLAB Function block that contains the moving average algorithm.

```
mdl = 'moving_avg_col_major';
dut = [mdl '/DUT'];

load_system(mdl);
open_system([dut '/MATLAB Function'])
```

```
function out_data = movingAvgfcn(inp_data)
out_data = hdl.npufun(@kernelAdd, [5 1], inp_data);
end

function y = kernelAdd(x)
y = single(1/5)*sum(x(:), 'native');
end
```

The `hdl.npufun` function computes the moving average algorithm and is compatible with the frame-to-sample conversion optimization. You could also use the Neighboring Processing Subsystem block to apply the same algorithm and generate HDL code with the frame-to-sample optimization. For an example, see “Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods” on page 22-17.

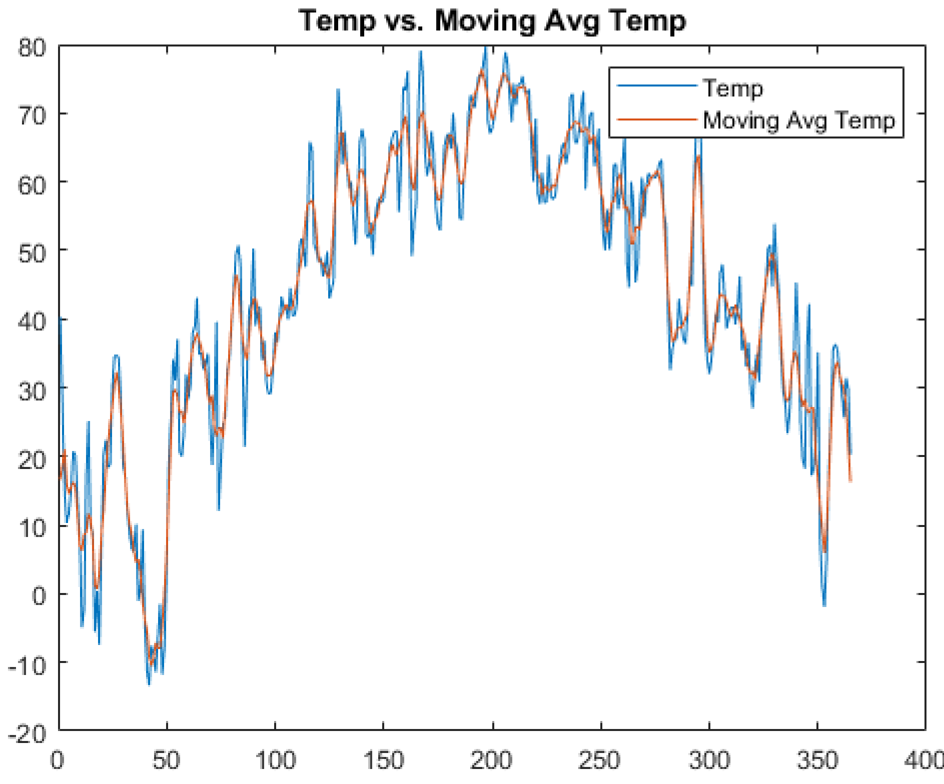
Run Model and Plot Output

Run the model to save the output data to your MATLAB workspace as `outputTemp`.

```
sim(mdl)
```

Plot the first year of input temperature data against the output temperature data to see the moving average filter applied to the data.

```
dayLength = 1:size(inputTemp,1);
yearofChoice = 1;

figure;
plot(dayLength,inputTemp(:,1));
title('Temp vs. Moving Avg Temp')
hold on;
plot(dayLength,outputTemp(:,1));
legend('Temp', 'Moving Avg Temp')
```



Generate HDL Code and View Area Usage

To generate HDL code using the frame-to-sample conversion optimization, first set the HDL subsystem to the DUT you want to generate code from.

```
hdlset_param(mdl,'HDLSubsystem', dut);
```

To convert the input matrix from a frame-based to a sample-based input, set the HDL block property `ConvertToSamples` to `on` on the Inport block of the DUT that connects to the frame-input signal.

```
hdlset_param([dut '/in_sig'], 'ConvertToSamples', 'on');
```

Enable the frame-to-sample conversion optimization and generate HDL code using the `makehdl` command.

```
hdlset_param(mdl, 'FrameToSampleConversion', 'on');
```

```
makehdl(dut)
```

```
### Working on the model <a href="matlab:open_system('moving_avg_col_major')">moving_avg_col_majo
### Generating HDL for <a href="matlab:open_system('moving_avg_col_major/DUT')">moving_avg_col_ma
### Using the config set for model <a href="matlab:configset.showParameterGroup('moving_avg_col_m
### Running HDL checks on the model 'moving_avg_col_major'.
### Begin compilation of the model 'moving_avg_col_major'...
### Working on the model 'moving_avg_col_major'...
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 59 cycles.
### Output port 1: The first valid output of this port will be after an initial latency of 20 val
### Output port 2: 59 cycles.
### Output port 2: The first valid output of this port will be after an initial latency of 20 val
### Working on... <a href="matlab:configset.internal.open('moving_avg_col_major', 'GenerateModel
### Begin model generation 'gm_moving_avg_col_major'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\moving_avg_col_major\gm_
### Begin VHDL Code Generation for 'moving_avg_col_major'.
### MESSAGE: The design requires 3650 times faster clock with respect to the base rate = 0.2.
### Working on counterNetwork as hdl_prj\hdlsrc\moving_avg_col_major\counterNetwork.vhd.
### Working on NeighborhoodCreator_5x1/row4_linebuffer/SimpleDualPortRAM_generic as hdl_prj\hdlsr
### Working on NeighborhoodCreator_5x1/row4_linebuffer as hdl_prj\hdlsrc\moving_avg_col_major\row
### Working on NeighborhoodCreator_5x1 as hdl_prj\hdlsrc\moving_avg_col_major\NeighborhoodCreator
### Working on boundaryCounters_5_1 as hdl_prj\hdlsrc\moving_avg_col_major\boundaryCounters_5_1.v
### Working on BoundaryCheck_5x1 as hdl_prj\hdlsrc\moving_avg_col_major\BoundaryCheck_5x1.vhd.
### Working on inp_data_NeighborhoodCreator as hdl_prj\hdlsrc\moving_avg_col_major\inp_data_Neigh
### Working on moving_avg_col_major/DUT/MATLAB Function/kernelAdd/nfp_add_single as hdl_prj\hdlsr
### Working on moving_avg_col_major/DUT/MATLAB Function/kernelAdd/nfp_mul_single as hdl_prj\hdlsr
### Working on moving_avg_col_major/DUT/MATLAB Function/kernelAdd as hdl_prj\hdlsrc\moving_avg_co
### Working on moving_avg_col_major/DUT/MATLAB Function as hdl_prj\hdlsrc\moving_avg_col_major\MA
### Working on moving_avg_col_major/DUT/Input_FIFOs/in_sig_FIFO as hdl_prj\hdlsrc\moving_avg_col_
### Working on moving_avg_col_major/DUT/Input_FIFOs as hdl_prj\hdlsrc\moving_avg_col_major\Input_
### Working on moving_avg_col_major/DUT/Output_FIFOs/out_data_FIFO as hdl_prj\hdlsrc\moving_avg_c
### Working on moving_avg_col_major/DUT/Output_FIFOs as hdl_prj\hdlsrc\moving_avg_col_major\Outpu
### Working on moving_avg_col_major/DUT as hdl_prj\hdlsrc\moving_avg_col_major\DUT.vhd.
### Generating package file hdl_prj\hdlsrc\moving_avg_col_major\DUT_pkg.vhd.
### Code Generation for 'moving_avg_col_major' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'moving_avg_col_major' complete with 0 errors, 1 warnings, and 1 messages.
### HDL code generation complete.
```

Run Synthesis

By default, the **Input Processing Order** parameter is set to `RowMajor`. Run synthesis to determine the initial area usage from the model without applying column-major ordering for optimization. To run synthesis on the model, you need to have a synthesis tool installed and on the MATLAB™ path. For more information, see “Tool Setup”.

To run synthesis on the model:

1. In Simulink, in the **HDL Code** tab, click **Workflow Advisor**.

2. In the left pane, click **1. Set Target > 1.1 Set Target Device and Synthesis Tool**. Set:

- **Target workflow** to `Generic ASIC/FPGA`
- **Synthesis tool** to `Xilinx Vivado`
- **Family** to `Virtex7`
- **Device** to `xc7vx485t`
- **Package** to `ffg1761`
- **Speed** to `-2`

3. Click **1.2 Set Target Frequency**, and set **Target Frequency (MHz)** to `300`.

4. In the left pane, click **4. FPGA Synthesis and Analysis > 4.2 Perform Synthesis and P/R > 4.2.2. Run Implementation**. Clear **Skip This Task**.

5. Right-click **4.2.2 Run Implementation** and click **Run to Selected Task**.

The HDL Workflow Advisor runs through synthesis and implementation for the model. The synthesis results display the **Resource summary** and **Timing summary**.

Resource summary

Resource	Usage	Available	Utilization (%)
Slice LUTs	3046	0	
Slice Registers	3740	0	
DSPs	1	0	
Block RAM Tile	0	0	
URAM	0	0	

Parsed timing report file: timing_post_map.rpt.

Timing summary

	Value
Requirement	3.3333 ns (300 MHz)
Data Path Delay	0.715 ns
Slack	-0.183 ns
Clock Frequency	284.39 MHz

For more information on the code generation and synthesis steps, see “HDL Code Generation and FPGA Synthesis from Simulink Model”.

Optimize Area Usage

Typically, for incoming frame data that has a significantly larger number of rows than columns, also called a *tall matrix*, you can use column-major ordering to optimize the area used on hardware. For incoming frame data that has a significantly larger number of columns than rows, also called a *long matrix*, you can use row-major ordering to optimize the area used on hardware. The output of either ordering method is the same, but the ordering can generate less or more efficient mapping onto the target FPGA hardware.

For this example, to optimize the area, you either change the input processing order to column-major ordering or manually transpose the data. It is less time consuming and requires no extra manual

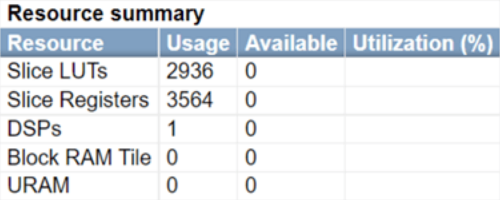
effort to use the **Input Processing Order** parameter instead of manually transposing the data. Set the input processing order to column major by using the command-line.

```
hdlset_param(mdl, 'InputProcessingOrder', 'ColumnMajor');
```

Generate HDL code using the `makehdl` command and run through synthesis with the same settings as you did before.

```
makehdl(dut)
```

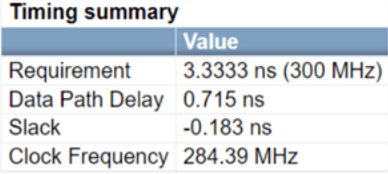
```
### Working on the model <a href="matlab:open_system('moving_avg_col_major')">moving_avg_col_majo
### Generating HDL for <a href="matlab:open_system('moving_avg_col_major/DUT')">moving_avg_col_ma
### Using the config set for model <a href="matlab:configset.showParameterGroup('moving_avg_col_m
### Running HDL checks on the model 'moving_avg_col_major'.
### Begin compilation of the model 'moving_avg_col_major'...
### Working on the model 'moving_avg_col_major'...
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 59 cycles.
### Output port 1: The first valid output of this port will be after an initial latency of 2 vali
### Output port 2: 59 cycles.
### Output port 2: The first valid output of this port will be after an initial latency of 2 vali
### Working on... <a href="matlab:configset.internal.open('moving_avg_col_major', 'GenerateModel
### Begin model generation 'gm_moving_avg_col_major'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\moving_avg_col_major\gm_
### Begin VHDL Code Generation for 'moving_avg_col_major'.
### MESSAGE: The design requires 3650 times faster clock with respect to the base rate = 0.2.
### Working on counterNetwork as hdl_prj\hdlsrc\moving_avg_col_major\counterNetwork.vhd.
### Working on NeighborhoodCreator_5x1/col1_row4 as hdl_prj\hdlsrc\moving_avg_col_major\col1_row4
### Working on NeighborhoodCreator_5x1 as hdl_prj\hdlsrc\moving_avg_col_major\NeighborhoodCreator
### Working on boundaryCounters_5_1 as hdl_prj\hdlsrc\moving_avg_col_major\boundaryCounters_5_1.v
### Working on BoundaryCheck_5x1 as hdl_prj\hdlsrc\moving_avg_col_major\BoundaryCheck_5x1.vhd.
### Working on inp_data_NeighborhoodCreator as hdl_prj\hdlsrc\moving_avg_col_major\inp_data_Neigh
### Working on moving_avg_col_major/DUT/MATLAB Function/kernelAdd/nfp_add_single as hdl_prj\hdlsr
### Working on moving_avg_col_major/DUT/MATLAB Function/kernelAdd/nfp_mul_single as hdl_prj\hdlsr
### Working on moving_avg_col_major/DUT/MATLAB Function/kernelAdd as hdl_prj\hdlsrc\moving_avg_co
### Working on moving_avg_col_major/DUT/MATLAB Function as hdl_prj\hdlsrc\moving_avg_col_major\MA
### Working on moving_avg_col_major/DUT/Input_FIFOs/in_sig_FIFO/FIFO/FIFO_classic/SimpleDualPortR
### Working on moving_avg_col_major/DUT/Input_FIFOs/in_sig_FIFO as hdl_prj\hdlsrc\moving_avg_col_
### Working on moving_avg_col_major/DUT/Input_FIFOs as hdl_prj\hdlsrc\moving_avg_col_major\Input_
### Working on moving_avg_col_major/DUT/Output_FIFOs/out_data_FIFO as hdl_prj\hdlsrc\moving_avg_c
### Working on moving_avg_col_major/DUT/Output_FIFOs as hdl_prj\hdlsrc\moving_avg_col_major\Outpu
### Working on moving_avg_col_major/DUT as hdl_prj\hdlsrc\moving_avg_col_major\DUT.vhd.
### Generating package file hdl_prj\hdlsrc\moving_avg_col_major\DUT_pkg.vhd.
### Code Generation for 'moving_avg_col_major' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'moving_avg_col_major' complete with 0 errors, 1 warnings, and 1 messages.
### HDL code generation complete.
```



Resource summary

Resource	Usage	Available	Utilization (%)
Slice LUTs	2936	0	
Slice Registers	3564	0	
DSPs	1	0	
Block RAM Tile	0	0	
URAM	0	0	

Parsed timing report file: timing_post_map.rpt.



Timing summary

	Value
Requirement	3.3333 ns (300 MHz)
Data Path Delay	0.715 ns
Slack	-0.183 ns
Clock Frequency	284.39 MHz

The **Resource summary** in the synthesis results shows a reduction in LUTs and register hardware used to map the algorithm as a result of altering the order in which input data is processed.

See Also

Related Examples

- “Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods” on page 22-17

More About

- “HDL Code Generation from Frame-Based Algorithms” on page 22-2
- Input processing order

Compute Image Characteristics with a Frame-Based Model for HDL Code Generation

In this example, you use frame-based modeling to model an image processing algorithm that detects an object based on color. The model takes 2-D images as input and produces a two-element location for the detected object. You then generate synthesizable HDL code by using the HDL Coder™ frame-to-sample conversion optimization, which generates pixel-based code from a frame-based model. For more info on the frame-to-sample conversion optimization, see "HDL Code Generation from Frame-Based Algorithms" on page 22-2.

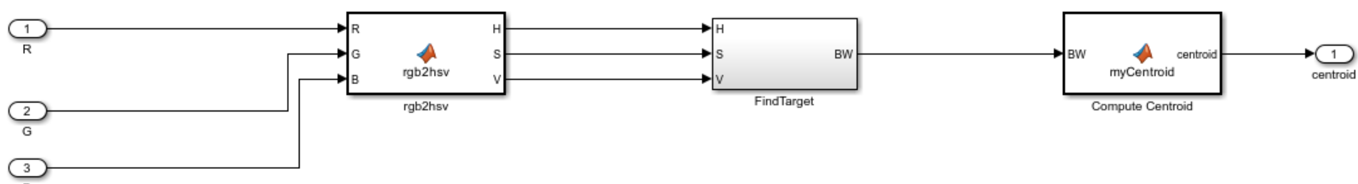
The object detection model uses neighborhood, iterator, and element-wise operator patterns. You can model neighborhood patterns by using the `hdl.npufun` function in a MATLAB® Function block or the Neighborhood Processing Subsystem block. You can implement iterator patterns by using `hdl.iteratorfun` in a MATLAB Function block. You can model element-wise operations using Simulink® blocks. Neighborhood processing operations, element-wise operations using Simulink blocks, and iterator operations support the generation of synthesizable HDL code by using the frame-to-sample conversion optimization. This model uses Sum, Abs, Relational, and Logical Operator blocks for element-wise operations. The input to the model is a frame-based RGB video signal. The output from the model is the sample-based location of the detected object.

Inspect and Run the Model

In the `hdlFrame_ObjectDetecting` model, the device under test (DUT) contains a MATLAB Function block `rgb2hsv` that uses neighborhood processing to convert the input RGB color space to the HSV color space. The HSV color space output is passed into the `FindTarget` subsystem that contains Simulink blocks that detect the target color area. Finally, the `Compute Centroid` MATLAB Function block computes a centroid value from the detected area in the image by using `hdl.iteratorfun`.

Open the `hdlFrame_ObjectDetecting/DUT` subsystem to see the object detection algorithm.

```
load_system('hdlFrame_ObjectDetecting');
open_system('hdlFrame_ObjectDetecting/DUT');
```



Open the `hdlFrame_ObjectDetecting/DUT/rgb2hsv` MATLAB Function block to see the RGB to HSV color space conversion algorithm.

```
open_system('hdlFrame_ObjectDetecting/DUT/rgb2hsv');
```

```
function [H, S, V] = rgb2hsv_frame(R, G, B)

[H, S, V] = hdl.npufun(@rgb2hsvKernel, [1 1], R, G, B);

end
```

```
function [h, s, v] = rgb2hsvKernel(r, g, b)

maxVal = max([r g b]);
minVal = min([r g b]);
delta = maxVal - minVal;

h = single(0.0);
if (r == maxVal)
    h = (g - b) / delta;
end

if (g == maxVal)
    h = (b - r) / delta + 2.0;
end

if (b == maxVal)
    h = (r - g) / delta + 4.0;
end

h = h/6.0;
if h < 0.0
    h = h + 1;
end

tmp = delta / maxVal;
if (delta == 0.0)
    h = single(0.0);
end

if (maxVal ~= 0.0)
    s = tmp;
else
    s = single(0.0);
end

v = maxVal;

end
```

Open the `hdlFrame_ObjectDetecting/DUT/Compute Centroid` block to see the centroid computation algorithm.

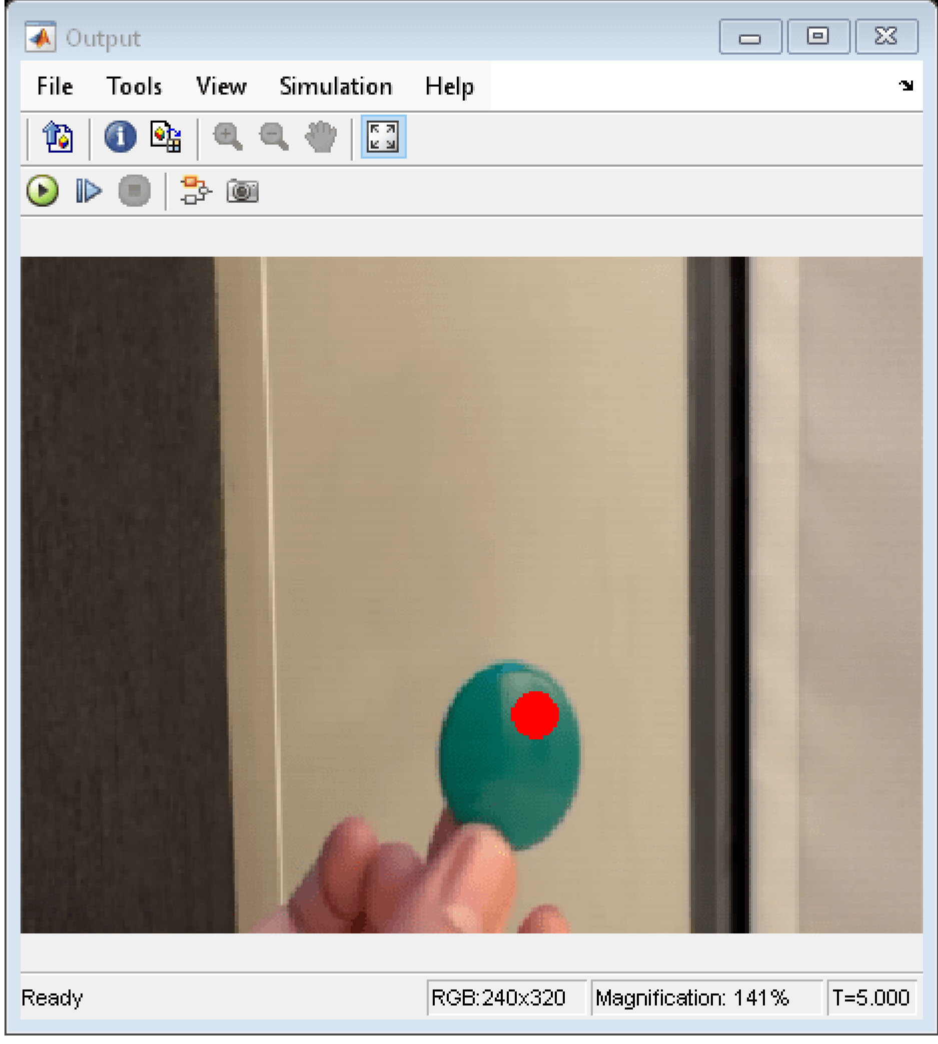
```
open_system('hdlFrame_ObjectDetecting/DUT/Compute Centroid');
```

```
function centroid = myCentroid_frame(BW)

C = coder.const(int32(size(BW, 2)));
sumRC = int32([0 0 0 0 0]); % centroidR, centroiC, RowCounter, ColCounter totalPoints

centRC = hdl.iteratorfun(@kernel, BW, sumRC, C);
recip = single(1)/single(centRC(5));
centR = int32(single(centRC(1))*recip);
centC = int32(single(centRC(2))*recip);
centroid = [centR, centC];
```

```
end

function sumRC = kernel(bw, sumRC, idx, C)
sumRC(3) = sumRC(3) + 1; % counter 1 rows

% check for boundaries
if sumRC(3) > C
    sumRC(3) = 1;
    sumRC(4) = sumRC(4) + 1; % counter 2 cols
end

if bw
    sumRC(1) = sumRC(1) + sumRC(3);
    sumRC(2) = sumRC(2) + sumRC(4);
    sumRC(5) = sumRC(5) + 1;
end

end
```

The model uses 2-D matrices as inputs to the DUT. These inputs signals are the separated R, G, and B components of the input image. Each input signal is a frame input matrix composed of 240x320 pixels. To see the frame size and simulation results, open the `hdlFrame_ObjectDetecting/Output` block and simulate the model.

```
open_system('hdlFrame_ObjectDetecting/Output')
sim("hdlFrame_ObjectDetecting");
```



Identify Streamed Signals

To generate synthesizable HDL code using the frame-to-sample conversion, identify the signals to be converted from frames to samples during HDL code generation. For this example, you convert the `R`, `G`, and `B` input signals. The R, G, and B input signals are streamed signals in this example because they go through the frame-to-sample conversion optimization during HDL code generation. The frame-to-sample conversion detects if the output of the DUT is a streamed output signal, which is a converted frame-to-sample signal, or a non-streamed signal, which is any output that does not get transformed from frame to samples. In this example, the centroid value is a characteristic of the image, and is characterized as a non-streamed signal.

Close the `hdlFrame_ObjectDetecting/Output` block and set the HDL block property `ConvertToSamples` on the Inport blocks of the DUT that connect to the R, G, and B input signals to convert the input signals from frame-based to sample-based inputs during HDL code generation.

```
close_system('hdlFrame_ObjectDetecting/Output')
hdlset_param('hdlFrame_ObjectDetecting/DUT/R', 'ConvertToSamples', 'on');
hdlset_param('hdlFrame_ObjectDetecting/DUT/G', 'ConvertToSamples', 'on');
hdlset_param('hdlFrame_ObjectDetecting/DUT/B', 'ConvertToSamples', 'on');
```

Generate HDL Code

Set the HDL block property `Architecture` to `MATLAB Datapath` on the MATLAB Function blocks inside DUT that convert image from RGB to HSV color space and compute the centroid.

```
hdlset_param('hdlFrame_ObjectDetecting/DUT/rgb2hsv', 'Architecture', 'MATLAB Datapath');
hdlset_param('hdlFrame_ObjectDetecting/DUT/Compute Centroid', 'Architecture', 'MATLAB Datapath')
```

Enable the frame-to-sample conversion optimization and generate HDL code using the `makehdl` command.

```
hdlset_param('hdlFrame_ObjectDetecting', 'FrameToSampleConversion', 'on');

makehdl('hdlFrame_ObjectDetecting/DUT')
```

```
### Working on the model <a href="matlab:open_system('hdlFrame_ObjectDetecting')">hdlFrame_Object
### Generating HDL for <a href="matlab:open_system('hdlFrame_ObjectDetecting/DUT')">hdlFrame_Obje
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlFrame_ObjectD
### Running HDL checks on the model 'hdlFrame_ObjectDetecting'.
### Begin compilation of the model 'hdlFrame_ObjectDetecting'...
### Working on the model 'hdlFrame_ObjectDetecting'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 1: 177 cycles.
### Output port 1: The first valid output of this port will be after an initial latency of 76799
### Output port 2: 177 cycles.
### Output port 2: The first valid output of this port will be after an initial latency of 76799
### Working on... <a href="matlab:configset.internal.open('hdlFrame_ObjectDetecting', 'GenerateMo
### Begin model generation 'gm_hdlFrame_ObjectDetecting'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlFrame_ObjectDetecting
### Estimated critical path for design: <a href="matlab:run('hdl_prj\hdlsrc\hdlFrame_ObjectDetect
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdl_
### Blocks that are not characterized for Critical Path Estimation: <a href="matlab:run('hdl_prj\
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdl_prj\hdlsrc
### Generating new validation model: '<a href="matlab:open_system('hdl_prj\hdlsrc\hdlFrame_Object
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlFrame_ObjectDetecting'.
### MESSAGE: The design requires 76800 times faster clock with respect to the base rate = 0.01666
### Working on... <a href="matlab:configset.internal.open('hdlFrame_ObjectDetecting', 'Traceabili
### Working on hdlFrame_ObjectDetecting/DUT/FindTarget/HSVtoBW/nfp_sub_single as hdl_prj\hdlsrc\h
### Working on hdlFrame_ObjectDetecting/DUT/rgb2hsv/rgb2hsvKernel/nfp_relop_single as hdl_prj\hdl
### Working on hdlFrame_ObjectDetecting/DUT/rgb2hsv/rgb2hsvKernel/nfp_add_single as hdl_prj\hdlsr
### Working on hdlFrame_ObjectDetecting/DUT/rgb2hsv/rgb2hsvKernel/nfp_div_single as hdl_prj\hdlsr
### Working on hdlFrame_ObjectDetecting/DUT/FindTarget/HSVtoBW/nfp_relop_single as hdl_prj\hdlsrc
### Working on hdlFrame_ObjectDetecting/DUT/rgb2hsv/rgb2hsvKernel/nfp_relop_single as hdl_prj\hdl
### Working on hdlFrame_ObjectDetecting/DUT/rgb2hsv/rgb2hsvKernel/nfp_relop_single as hdl_prj\hdl
### Working on hdlFrame_ObjectDetecting/DUT/rgb2hsv/rgb2hsvKernel as hdl_prj\hdlsrc\hdlFrame_Obje
### Working on hdlFrame_ObjectDetecting/DUT/rgb2hsv as hdl_prj\hdlsrc\hdlFrame_ObjectDetecting\rg
### Working on hdlFrame_ObjectDetecting/DUT/FindTarget/HSVtoBW/nfp_abs_single as hdl_prj\hdlsrc\h
### Working on hdlFrame_ObjectDetecting/DUT/FindTarget/HSVtoBW as hdl_prj\hdlsrc\hdlFrame_ObjectD
### Working on hdlFrame_ObjectDetecting/DUT/FindTarget as hdl_prj\hdlsrc\hdlFrame_ObjectDetecting
### Working on hdlFrame_ObjectDetecting/DUT/Compute Centroid/kernel as hdl_prj\hdlsrc\hdlFrame_Ob
### Working on hdlFrame_ObjectDetecting/DUT/Compute Centroid/nfp_convert_single_to_sfix_32_En0 as
### Working on hdlFrame_ObjectDetecting/DUT/Compute Centroid/nfp_mul_single as hdl_prj\hdlsrc\hdl
### Working on hdlFrame_ObjectDetecting/DUT/Compute Centroid/nfp_recip_single as hdl_prj\hdlsrc\h
```

```
### Working on hdlFrame_ObjectDetecting/DUT/Compute Centroid/nfp_convert_sfix_32_En0_to_single as
### Working on hdlFrame_ObjectDetecting/DUT/Compute Centroid as hdl_prj\hdlsrc\hdlFrame_ObjectDet
### Working on hdlFrame_ObjectDetecting/DUT/Input_FIFOs/R_FIFO/FIFO/FIFO_classic/SimpleDualPortRA
### Working on hdlFrame_ObjectDetecting/DUT/Input_FIFOs/R_FIFO as hdl_prj\hdlsrc\hdlFrame_ObjectD
### Working on hdlFrame_ObjectDetecting/DUT/Input_FIFOs/G_FIFO as hdl_prj\hdlsrc\hdlFrame_ObjectD
### Working on hdlFrame_ObjectDetecting/DUT/Input_FIFOs/B_FIFO as hdl_prj\hdlsrc\hdlFrame_ObjectD
### Working on hdlFrame_ObjectDetecting/DUT/Input_FIFOs as hdl_prj\hdlsrc\hdlFrame_ObjectDetectin
### Working on hdlFrame_ObjectDetecting/DUT/Output_FIFOs/centroid_memory as hdl_prj\hdlsrc\hdlFra
### Working on hdlFrame_ObjectDetecting/DUT/Output_FIFOs as hdl_prj\hdlsrc\hdlFrame_ObjectDetecti
### Working on hdlFrame_ObjectDetecting/DUT as hdl_prj\hdlsrc\hdlFrame_ObjectDetecting\DUT.vhd.
### Generating package file hdl_prj\hdlsrc\hdlFrame_ObjectDetecting\DUT_pkg.vhd.
### Code Generation for 'hdlFrame_ObjectDetecting' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlFrame_ObjectDetecting' complete with 0 errors, 0 warnings, and 4 messages.
### HDL code generation complete.
```

The frame-to-sample conversion separates the frame-based inputs into sample, valid, and ready signals for a sample-based hardware-targeted interface.

See Also

Related Examples

- “HDL Code Generation from Frame-Based Algorithms” on page 22-2
- “Generate HDL Code from Frame-Based Models by Using Neighborhood Modeling Methods” on page 22-17

Generate HDL Code from Frame-Based Model by Using Neighborhood Processing with States

You can represent an algorithm that converts Bayer pattern encoding to a true-color, RGB image by using a frame-based model and neighborhood processing methods. In this example, you use the HDL Coder™ frame-to-sample conversion optimization to generate synthesizable pixel-based HDL code from a frame-based model.

This example uses neighborhood processing operations that are needed for the demosaic algorithm. The algorithm contains persistent variables, that model states in a MATLAB™ design, to track whether a pixel location is odd or even. The model uses a MATLAB Function block with the neighborhood processing function `hdl.npufun`.

Examine MATLAB Function Block Algorithm

In the `hdlFrame_Demosaic` model, there is a MATLAB Function block inside the device under test (DUT) that contains the neighborhood processing operations needed for the demosaic algorithm. To generate synthesizable HDL code from the frame-based model, HDL Coder uses `hdl.npufun` to create a streaming sample-based neighborhood processing algorithm. You can use this function in a frame-based model to contain the entire image processing algorithm in a single MATLAB Function block.

Open the MATLAB Function block in the `hdlFrame_Demosaic/DUT` subsystem to see the demosaic algorithm.

```
load_system("hdlFrame_Demosaic");
open_system("hdlFrame_Demosaic/DUT/MATLAB Function");
imshow("hdlFrame_demosaic.tif");
```

Examine the Kernel Function for Neighborhood Processing

This code shows a section of kernel function that this model uses for neighborhood processing. This code shows how the function uses persistent variables inside the kernel function `demosaic_bilinear` to track the row and column location in the input image. The function uses the row and column values to decide the computations to get the `R`, `G`, and `B` output values.

```
function [R,G,B] = demosaic_bilinear(height, width, im)

persistent row col
if isempty(row)
    row = uint16(1);
    col = uint16(1);
end
```

```
isRowOdd = bitget(row, 1);
isColOdd = bitget(col, 1);
```

Run the Model

The input data `hdlFrame_demosaic.tif` is a image encoded by the Bayer pattern encoded with a size of size 500-by-500 pixels. Simulate the model to see the decoded RGB color data an as output color image.

```
sim("hdlFrame_Demosaic");
```

Generate HDL Code

Generate synthesizable HDL code by using the frame-to-sample conversion. On the Inport block of the DUT, set the HDL block property `ConvertToSamples` to convert the input signals from frame-based to sample-based inputs.

```
hdlset_param('hdlFrame_Demosaic/DUT/I','ConvertToSamples','on');
```

For the MATLAB Function block that contains the optical flow algorithm, set the HDL block property `Architecture` to `MATLAB Datapath`. Enable the frame-to-sample conversion optimization and generate HDL code using the `makehdl` command. For more information on the frame-to-sample conversion optimization, see "HDL Code Generation from Frame-Based Algorithms" on page 22-2.

```
hdlset_param('hdlFrame_Demosaic/DUT/MATLAB Function','Architecture','MATLAB Datapath');
hdlset_param('hdlFrame_Demosaic','FrameToSampleConversion','on');
```

```
makehdl('hdlFrame_Demosaic/DUT');
```

The frame-to-sample conversion separates the frame-based inputs into sample, valid, and ready signals for a sample-based hardware-targeted interface.

See Also

Functions
`hdl.npufun`

Related Examples

- "Compute Image Characteristics with a Frame-Based Model for HDL Code Generation" on page 22-46
- "Synthesize Code for Frame-Based Model" on page 22-26

More About

- "HDL Code Generation from Frame-Based Algorithms" on page 22-2

Code Generation Reports, HDL Compatibility Checker, Block Support Library, and Code Annotation

- “Create and Use Code Generation Reports” on page 23-2
- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 23-12
- “Generate Web View of Model in Code Generation Report” on page 23-21
- “Generate HDL Code with Annotations or Comments” on page 23-24
- “Check Your Model for HDL Compatibility” on page 23-29
- “Display Blocks for HDL Code Generation in Library Browser” on page 23-31
- “Trace Code Using the Mapping File” on page 23-34
- “Add or Remove the HDL Configuration Component” on page 23-37

Create and Use Code Generation Reports

HDL Coder generates and displays an HTML code generation report when you generate HDL code from a Simulink model that has at least one of the report configuration parameters enabled. The report contains information about the generated code such as:

- Resource utilization estimates
- Timing estimates
- The impact of HDL Coder optimizations
- Links tracing code to Simulink blocks

Customize the code generation reports by using the model configuration parameters in the **HDL Code Generation > Report** settings of the Configuration Parameters dialog box.

Generate Reports

These tables list the sections you can add to your code generation reports. HDL Coder generates the **Summary**, **Clock Summary**, and **Code Interface Report** sections by default.

Report Section	Description	Configuration Parameter	Dependencies
Summary	Contains information about the model, design under test (DUT), date of code generation, and non-default HDL Coder settings.	Enable at least one of the parameters in **HDL Code Generation > Report** section of the Configuration Parameters dialog box.	In the **HDL Code Generation > Global Settings** settings, enable **Generate HDL code**.

Report Section	Description	Configuration Parameter	Dependencies
Clock Summary	Contains information about the: • Base rate of the model • Base rate of the DUT • Sample time of the outputs • Latency of the outputs The report also displays information based on the number of clocks in the model, including: • The clock enable signals for single-rate models • A clock table for multirate models For more information, see "Using Multiple Clocks in HDL Coder" on page 20-14.		
Code Interface report	Lists the names, data types, and bit lengths of the input and output ports to the DUT. The report displays links to each input port and output port in your Simulink model.		

Timing and Area Reports

Report Section	Description	Configuration Parameter	Dependencies
High-level Resource Report	Summarizes the adders, subtractors, multipliers, registers, and other resources the DUT consumes.	**Generate resource utilization report**	In the **HDL Code Generation > Global Settings** settings, enable **Generate HDL code**.

Report Section	Description	Configuration Parameter	Dependencies
Target-specific Report	Shows the resource utilization report when you generate target-specific code with an FPGA floating-point library mapping.		• In the **HDL Code Generation > Global Settings** settings, enable **Generate HDL code**. • In the **HDL Code Generation > Floating Point** settings, set **Vendor Specific Floating Point Library** to a setting other than `None`.
Native Floating-Point Resource Report	Lists the floating-point operators that your Simulink blocks map to.		• In the **HDL Code Generation > Global Settings** settings, enable **Generate HDL code**. • In the **HDL Code Generation > Floating Point** settings, enable **Use Floating Point**.
Critical Path Estimation	Estimates the critical path and the propagation delay along the path. This section estimates the maximum frequency the DUT can run at.	**Generate high-level timing critical path report**	• In the **HDL Code Generation > Global Settings** settings, enable **Generate HDL code**. • In the **HDL Code Generation > Global Settings**, in the **Model Generation** tab, enable **Generated model**.

Optimization Reports

Report Section	Description	Configuration Parameter	Dependencies
Delay Balancing	Provides detailed information on the: • Pipeline latency • Phase delays added at the output ports to match delays along parallel paths • Delay absorption For more information, see “Delay Balancing Report” on page 21-84.	**Generate optimization report**	• In the **HDL Code Generation > Global Settings** settings, enable **Generate HDL code**. • In the **HDL Code Generation > Global Settings**, in the **Model Generation** tab, enable **Generated model**.
Hierarchy Flattening	Displays the hierarchy flattening status, subsystems that have **FlattenHierarchy** set to `on` or `off`, and the inline HDL files. For more information, see “Hierarchy Flattening Report” on page 21-118.		
Code Reuse	Summarizes where HDL Coder reused generated subsystem code.		
Target Code Generation	Displays target device summary and target mapping status when the generated code uses floating-point types.		
Streaming and Sharing	Summarizes information about the subsystems you specify sharing or streaming for. For more information, see “Streaming Report” on page 21-44 and “Resource Sharing Report” on page 21-47.		
Clock Rate Pipelining	Details how clock-rate pipelining performed in your model. For more information, see “Clock-Rate Pipelining Report” on page 21-151.		

Report Section	Description	Configuration Parameter	Dependencies
Distributed Pipelining	Displays comparative listings of the registers before and after you apply the distributed pipelining transform. For more information, see "Distributed Pipelining Report" on page 21-133.		
Adaptive Pipelining	Displays the status of the adaptive pipelining optimization, blocks for which pipeline registers are inserted, and the number of pipeline registers. For more information, see "Adaptive Pipelining Report" on page 21-186.		
Frame to Sample	Contains information about the: • Frame-to-sample conversion parameters • Input ports and output ports that HDL Coder converted to samples • Output latency • Input and output ports that map large delays to external memory		

Traceability Report and Model Web View

Report Section	Description	Configuration Parameter	Dependencies
Traceability Report	Links lines of the generated code to the corresponding blocks in the Simulink model when possible. Enable this report to use the code view in Simulink to trace between the code and model. For more information, see “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 23-12.	**Generate traceability report**	In the **HDL Code Generation > Global Settings** settings, enable **Generate HDL code**.
Model Web View	Displays the Simulink model in the HTML code generation report. The model web view requires Simulink Report Generator™. For more information, see “Generate Web View of Model in Code Generation Report” on page 23-21.	**Generate model Web view**	In the **HDL Code Generation > Global Settings** settings, enable **Generate HDL code**.

Alternatively, you can programmatically customize the code generation reports by setting the properties in this table `on` or `off`. Use `hdlset_param` or `makehdl` to set these properties.

Report Section	`makehdl` Property	Dependencies
High-level Resource Report	`ResourceReport`	Set `GenerateHDLCode` to `on`.
Target-specific Report		Set `GenerateHDLCode` to `on`, and set `FloatingPointTargetConfiguration` to a vendor-specific floating point library.
Native Floating-Point Resource Report		Set `GenerateHDLCode` and `UseFloatingPoint` to `on`.
Critical Path Estimation	`CriticalPathEstimation`	Set `GenerateHDLCode` and `GenerateModel` to `on`.
Delay Balancing	`OptimizationReport`	Set `GenerateHDLCode` and `GenerateModel` to `on`.
Hierarchy Flattening		
Code Reuse		

Report Section	makehdl Property	Dependencies
Target Code Generation		
Streaming and Sharing		
Clock Rate Pipelining		
Distributed Pipelining		
Adaptive Pipelining		
Frame to sample		
Traceability Report	`Traceability`	Set `GenerateHDLCode` to `on`.
Model Web View	`HDLGenerateWebView`	Set `GenerateHDLCode` to `on`.

Navigate the Code Generation Report

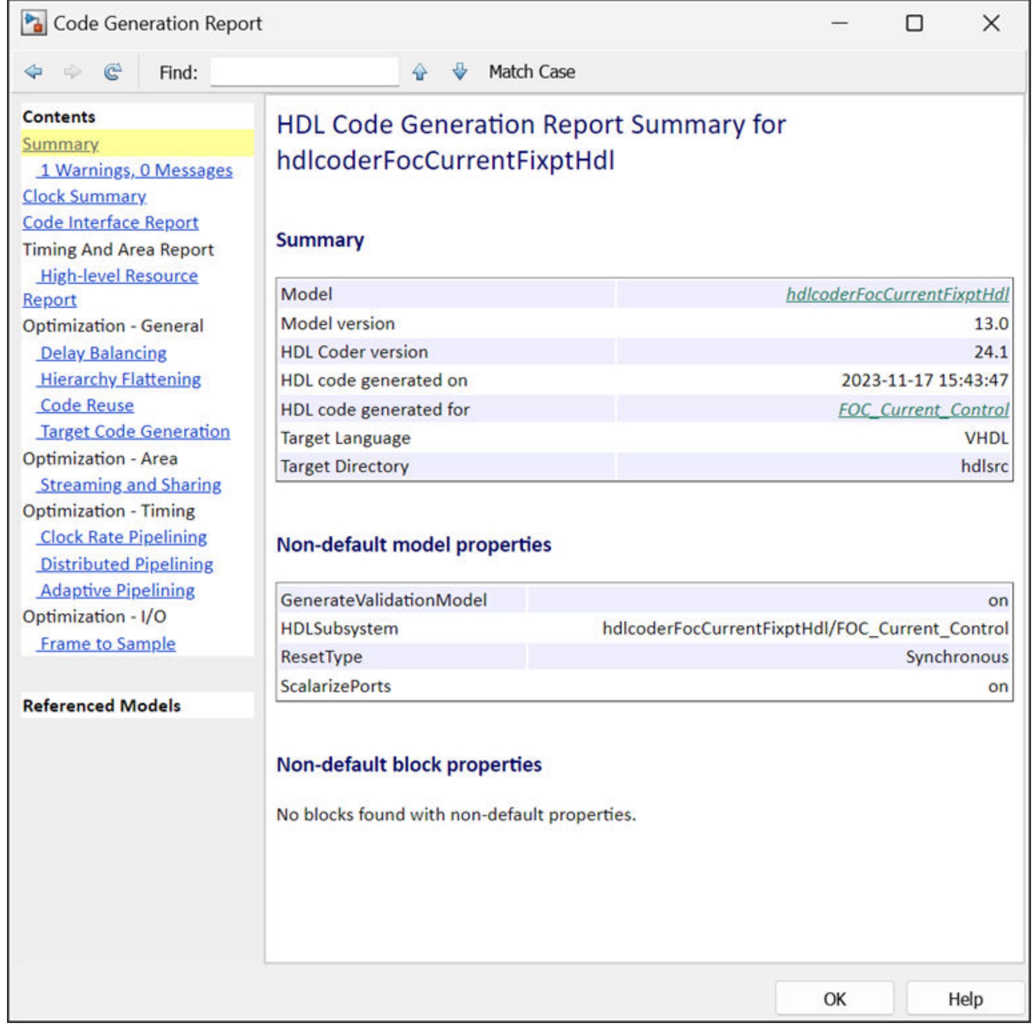
You can navigate to the code generation report from the Simulink model. For example, suppose you generate a report for the model `hdlcoderFocCurrentFixptHdl.slx`. Open the model by using this command:

```
openExample('hdlcoder/ClockRatePipeliningExample','supportingFile','hdlcoderFocCurrentFixptHdl.s
```

In this model, the configuration parameters **Generate resource utilization reports** and **Generate optimization reports** are enabled. Build HDL code from the subsystem `FOC_Current_Control` by using this command:

```
makehdl('hdlcoderFocCurrentFixptHdl/FOC_Current_Control')
```

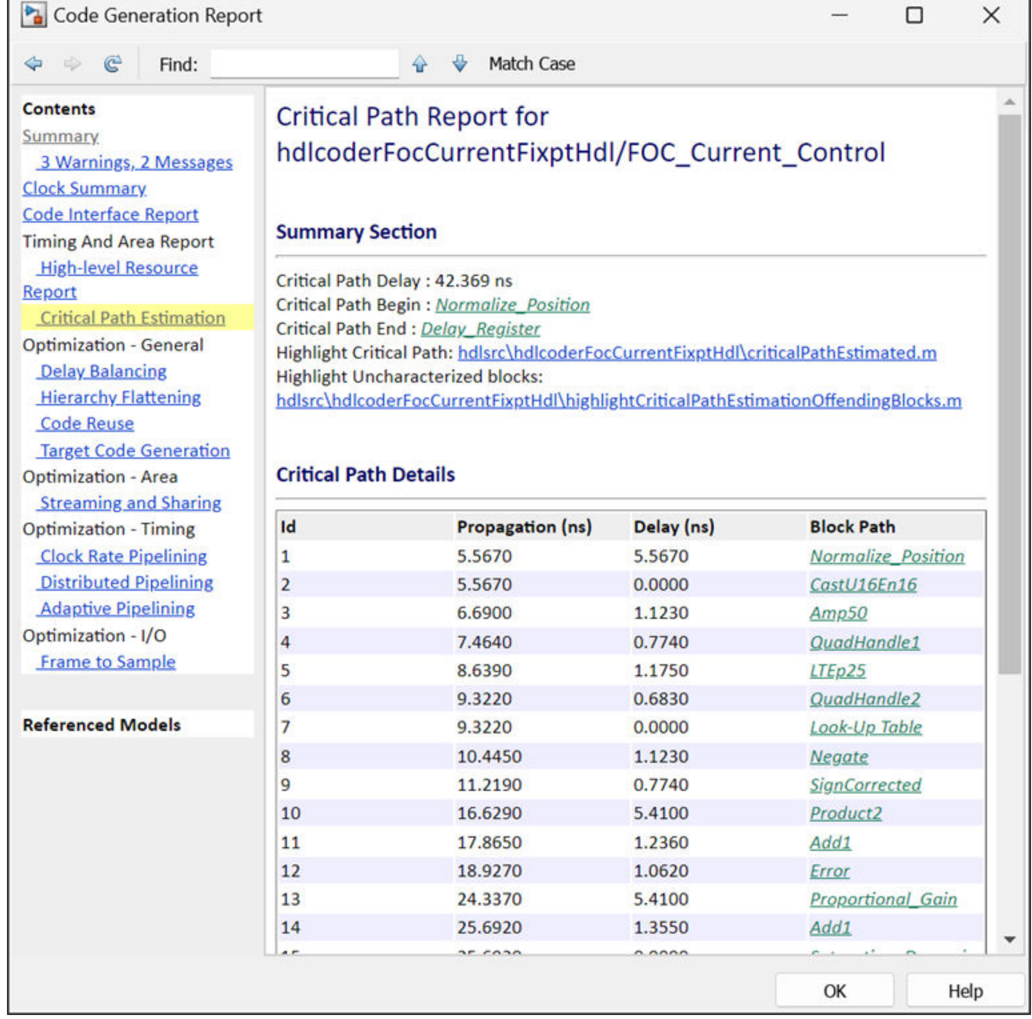
The report opens automatically after code generation. You can also open the report by clicking the link to `hdlcoderFoxCurrentFixptHdl_codegen_rpt.html` in the MATLAB Command Window or by opening the **HDL Coder** app and clicking **Open Report**.



Click the links in the **Contents** pane to read the different code generation report sections. The generated report contains a **High-level Resource Report** section that captures the resource utilization of the generated code. It also contains sections for clock-rate pipelining, distributed pipelining, and other optimizations that capture how HDL Coder optimized the generated code.

You can customize the reports to include only the sections you need. For example, enable the `CriticalPathEstimation` property to generate a report that includes a **Critical Path Estimation** section.

```
makehdl('hdlcoderFocCurrentFixptHdl/FOC_Current_Control', 'CriticalPathEstimation', 'on')
```



Id	Propagation (ns)	Delay (ns)	Block Path
1	5.5670	5.5670	Normalize_Position
2	5.5670	0.0000	CastU16En16
3	6.6900	1.1230	Amp50
4	7.4640	0.7740	QuadHandle1
5	8.6390	1.1750	LTEp25
6	9.3220	0.6830	QuadHandle2
7	9.3220	0.0000	Look-Up Table
8	10.4450	1.1230	Negate
9	11.2190	0.7740	SignCorrected
10	16.6290	5.4100	Product2
11	17.8650	1.2360	Add1
12	18.9270	1.0620	Error
13	24.3370	5.4100	Proportional_Gain
14	25.6920	1.3550	Add1

Use Code Generation Reports to Evaluate Code Before Synthesis

When you select the **Generate resource utilization report** or **Generate high-level timing critical path report** parameters, HDL Coder adds a **Timing and Area Report** section to the code generation report. You can use the timing and area report to evaluate whether the generated code can run with the frequency and resources your hardware requires.

Assess the Area of the Generated Code

The **Summary** section of the high-level resource report estimates the usage of:

- Multipliers
- Adders and subtractors
- 1-bit registers
- RAMs
- Multiplexors
- I/O bits

- Static shift operators
- Dynamic shift operators

Use the resource report to track and reduce the estimated resource usage of the generated HDL code.

The **High-Level Resource Report** section also contains the following sections:

- The **Registers** section displays the total number of 1-bit registers. The total is the sum of products over the bit widths of the registers and their frequency of occurrence.
- **Static Shift Operators** and **Dynamic Shift operators** sections. A static shift is a shift value that is a mask constant. The shift logic does not change. A dynamic shift is a shift value specified as an input to a block. Dynamic shifts are more resource expensive than static shifts.

If the resource usage in the reports exceeds what is available on your hardware, consider applying area optimizations such as resource sharing and streaming to the DUT. Resource sharing and streaming optimize the generated code to use shared hardware resources that reduce the area. For more information, see “Area Optimizations” on page 21-5.

Assess the Timing of the Generated Code

The **Critical Path Estimation** section includes an estimate of the combinatorial path in the model with the longest timing delay. HDL Coder uses target-specific timing databases to estimate the critical path of the generated code. If HDL Coder does not have a timing database for your target configuration, generate a target-specific timing database using the `genhdltdb` function. For more information, see “Critical Path Estimation Without Running Synthesis” on page 21-192.

The propagation delay of the critical path limits the frequency that the generated code can run at on the target. If the estimated delay of the critical path estimation is too long for the DUT to run at your target frequency, consider optimizing the estimated critical path. The critical path estimation links to a `criticalPathEstimated` script that highlights the critical path in the generated model. Use this script to identify the estimated critical path in your generated model, and then add delays in your original Simulink model to break the critical path.

Additionally, HDL Coder optimizations can optimize the timing of the critical path and increase the maximum frequency of the DUT. For more information, see “Speed Optimizations” on page 21-5.

Assess the Impact of Optimizations

If the reported timing or area usage do not meet requirements, you can optimize the original model and generate code from it again before synthesis. After generating code, use optimization reports to understand how HDL Coder optimized the DUT or why it did not apply specific optimizations. Use the optimization reports to identify obstacles to optimization and improve the timing and area of the DUT.

See Also

Related Examples

- “Increase Clock Frequency Using Clock-Rate Pipelining” on page 21-153
- “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 21-205
- “Improve Resource Sharing with Design Modifications” on page 21-69

Navigate Between Simulink Model and HDL Code by Using Traceability

In this section...

"How Traceability Works" on page 23-12
"Generate Traceability Report" on page 23-13
"Report Location" on page 23-13
"View the Traceability Report" on page 23-14
"Traceability by Using Code View" on page 23-14
"Code-to-Model Navigation" on page 23-15
"Model-to-Code Navigation" on page 23-18
"Traceability Report Limitations" on page 23-20

Even a relatively small model can generate hundreds of lines of HDL code. To identify the mapping between your source model and the generated HDL code more easily, use the traceability support in HDL Coder.

How Traceability Works

When you enable traceability support and generate HDL code for your model, the code generator creates and displays an HTML code generation report.

By default, the code generator uses the line-level style to generate a traceability report. The report generated by using this style contains hyperlinks for each line of HDL code to navigate between code and model. You can customize the traceability style to generate a comment-based report. This style contains hyperlinked comments above a block of code that correspond to a searchable tag for a certain block in your model. To learn more about the two traceability styles, see Traceability style.

You can generate reports for the root-level model or for subsystems, blocks, Stateflow charts, or MATLAB Function blocks. By default, HDL Coder generates a report for the top-level model.

After you generate the report, you can navigate from:

- Model to code: Select a certain block in your model and navigate to corresponding lines of HDL code in the report.
- Code to model: Select a line of code in the report and navigate to Simulink blocks corresponding to that line of code.

HDL Coder provides this two-way navigation or bidirectional traceability. With traceability support, you can:

- Verify that the generated code is as you expect. You can identify which model elements correspond to a line of code, and track code from different model elements that you have or have not reviewed.
- Verify whether the generated code meets the design requirements. You can assign the requirements to model elements and include the requirements as hyperlinks in the traceability report.

Generate Traceability Report

You can generate the report in the Configuration Parameters dialog box or at the command-line.

1. Enable generation of the traceability report.

   - In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select **Settings > Report Options**, and then select **Generate traceability report**.
   - At the command line, use `hdlset_param` to set the `Traceability` property on the model.

   To learn more about this parameter, see Generate traceability report.
2. Specify the traceability style. To generate a line-level traceability report, leave this setting as the default. To generate a comment-based traceability report:

   - On the **HDL Code Generation > Report** pane, specify the **Traceability style**.
   - At the command line, use `hdlset_param` to specify the `TraceabilityStyle` property on the model.

   To learn more about this parameter, see Traceability style.
3. Generate HDL code and the traceability report. Either select the DUT Subsystem and click **Generate HDL Code** on the Simulink Toolstrip, or run `makehdl` on the DUT Subsystem at the command line.

When HDL code generation is complete, the HTML code generation report appears in a new window.

Report Location

By default, HDL Coder writes the code generation report files to a folder in the `hdlsrc\html\` folder of the build folder. If you close the report, you can navigate to this folder to reopen the report.

Before generating code, you can customize the target folder that stores the HDL code and the report files.

- In the Configuration Parameters dialog box, specify the target folder by using the **Target** setting.
- At the command line, use the `TargetDirectory` property.

To learn how to specify this parameter, see Code Generation Folder.

To keep your traceability report up to date, regenerate the HDL code and report after modifying the source model.

View the Traceability Report

In the HTML code generation report window, select the **Traceability Report** section. In the left pane of the report, click the names of **Generated Source Files** to view their contents in a MATLAB web browser window.

This figure shows a typical traceability report.

Eliminated / Virtual Blocks

Block Name	Comment
<S2>/x_in	Inport
<S2>/h_in1	Inport
<S2>/h_in2	Inport
<S2>/h_in3	Inport
<S2>/h_in4	Inport
<S2>/y_out	Outport
<S2>/delayed_xout	Outport

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

Subsystem: sfir_fixed/symmetric_fir

Object Name	Code Location
<S2>/a1	symmetric_fir.vhd:208, 209, 210
<S2>/a2	symmetric_fir.vhd:212, 213, 214

The traceability report has several subsections that indicate the blocks or subsystems from which the code was generated:

- The **Eliminated / Virtual Blocks** section accounts for blocks that are untraceable because they are not included in the generated HDL code.
- The **Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions** section provides a complete mapping between model elements and code.

If you assigned block requirements, you can see the requirements as hyperlinked comments in the traceability report. For more information, see Include requirements in block comments.

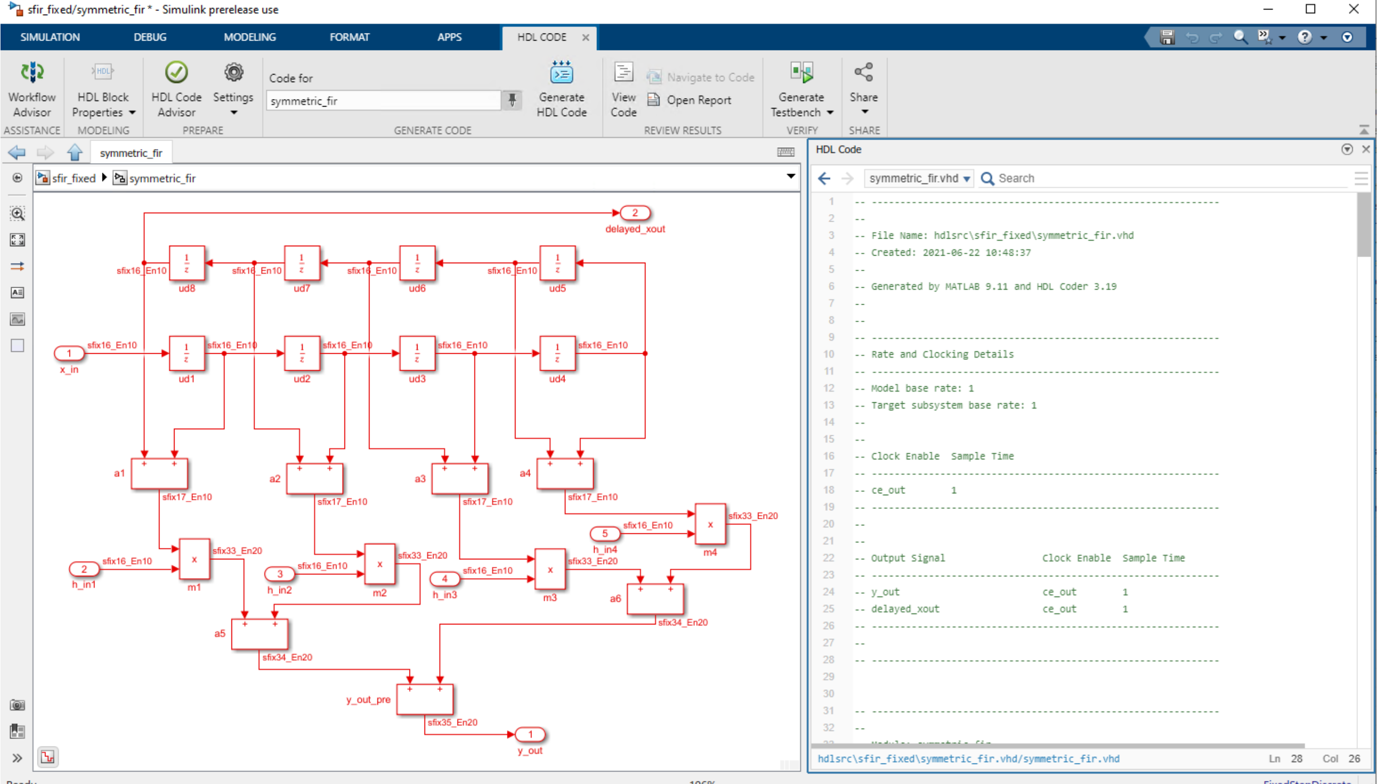
Traceability by Using Code View

By default, after you enable generation of the traceability report and generate HDL code for your model, the Code view displays the generated code to the right of your model. You can use the Code view to trace generated HDL code to your model and your model to the generated HDL code.

To manually open the Code view, open the **HDL Coder** app. On the Simulink toolstrip, click the **View Code** button. Select the file that you want to display by using the drop-down list at the top of the

Code view. You can dock or undock the Code view from the editor and minimize or expand the Code view using the down arrow in the upper right corner of the Code view. You can also use rich text capabilities, such as code folding and hiding comments.

This figure shows the Code view opened from the **View Code** button on the Simulink toolstrip in **HDL Coder** app.



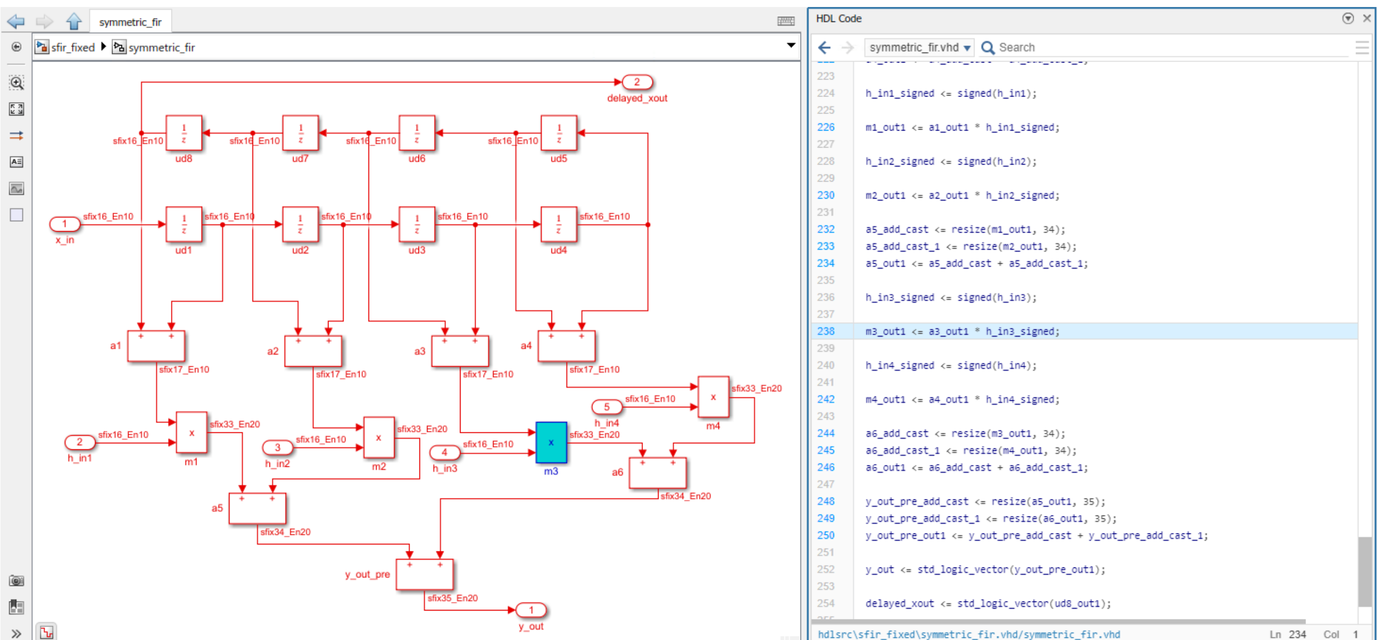
Code-to-Model Navigation

To navigate from the HDL code to the model, follow either of these workflows:

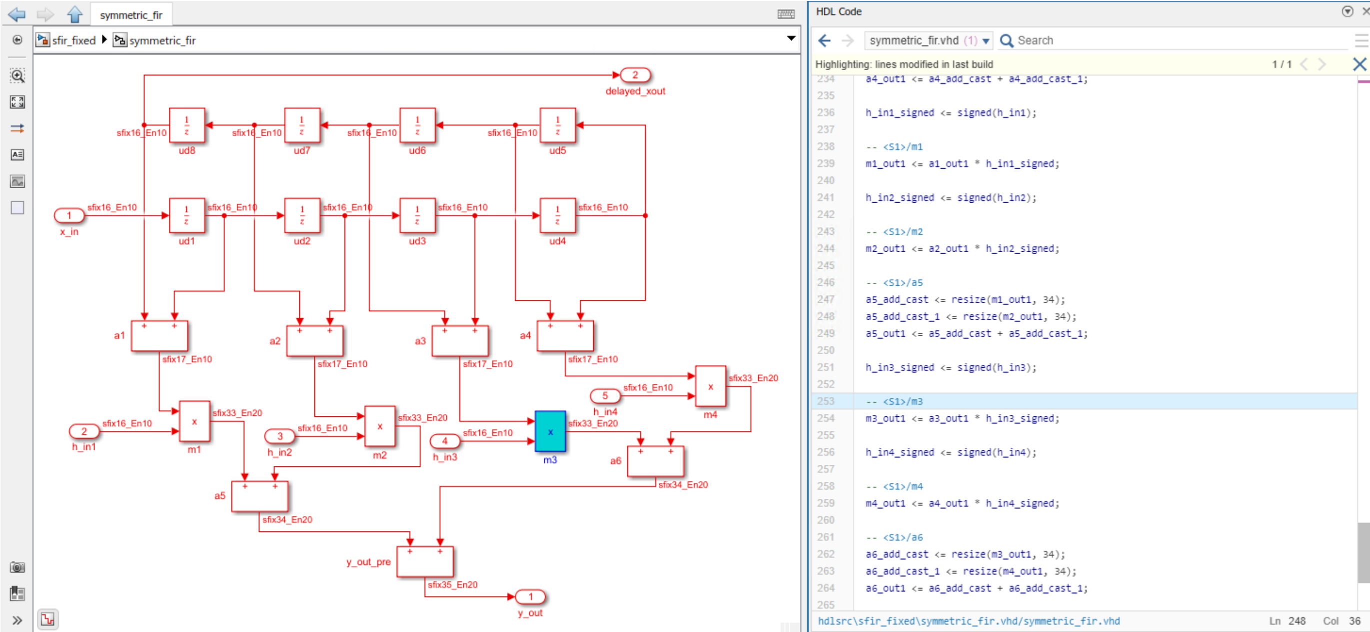
Use the Code view:

1. Navigate to the Code view by clicking the automatically generated Code view panel on the right or by clicking the **View Code** button on the Simulink toolstrip in the **HDL Coder** app.
2. In the Code view, click the line number hyperlink or code comment link to highlight the block that the code line traces to. You can trace lines of code to the model elements from which they were generated.

This figure shows how the generated HDL code on line 238 in `symmetric_fir.vhd` in the Code view maps to the block highlighted in blue in the model when the **Traceability style** is specified as `Line Level`.



This figure shows how the comment `<S1>/m3` generated in HDL code on line 253 in `symmetric_fir.vhd` in the Code view maps to the block highlighted in blue in the model when the **Traceability style** is specified as `Comment Based`.



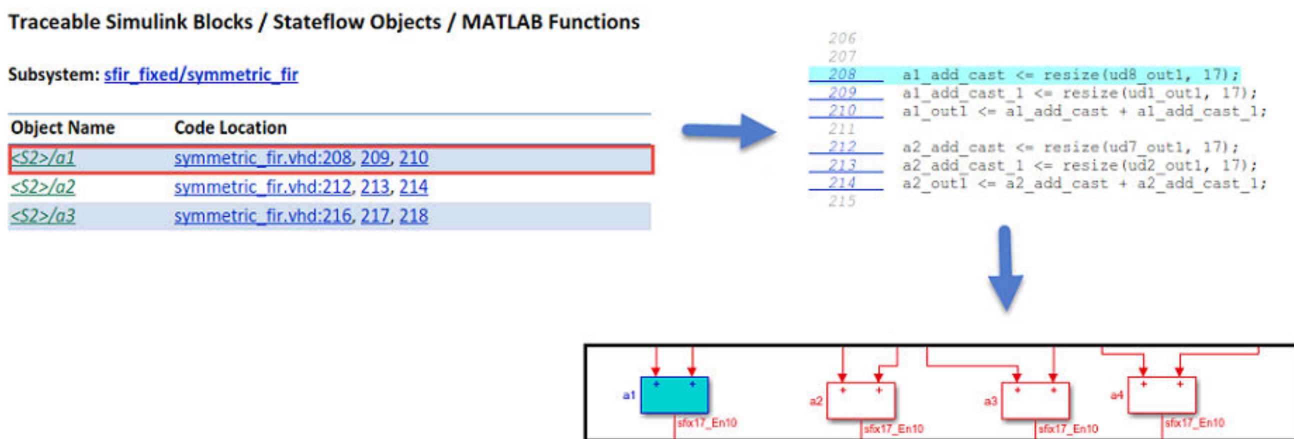
Use the Code Generation Report:

1. In the traceability report, on the **Code Location** column, click any hyperlink.

   The code generator highlights that line of HDL code in the generated source file.
2. Select the link corresponding to that line of code in the source file.

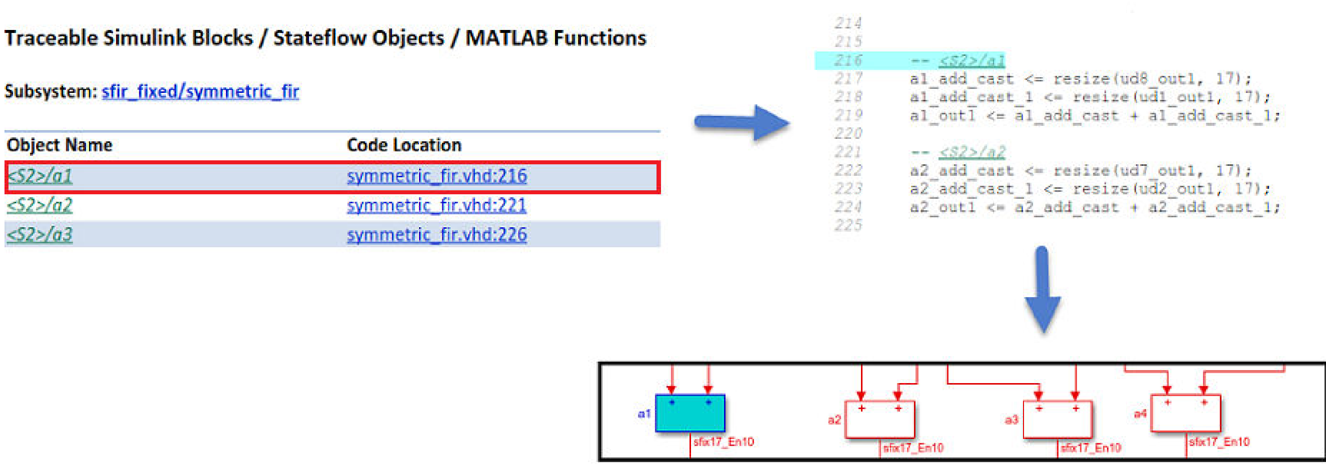
The code generator opens a separate window that displays the highlighted Simulink block corresponding to that line of code.

This figure shows how to navigate from the HDL code to the model by using the traceability report when you specify `Line Level` as the **Traceability style**.



In the traceability report, you see that HDL Coder generates line-level hyperlinks to the HDL code in the **Code Location** column. Click the link to highlight that line of code in the HDL source file, and then click the hyperlink for that line of code in the source file to highlight the corresponding block in your model.

This figure shows how to navigate from the HDL code to the model using the traceability report when you specify `Comment Based` as the **Traceability style**.



In the traceability report, when you select a hyperlink in the **Code Location** column, you see that HDL Coder highlights a hyperlinked comment `<S2>/a1` in the HDL code. When you click the hyperlinked comment in the HDL source file, the code generator highlights the corresponding block `a1` in your model.

Model-to-Code Navigation

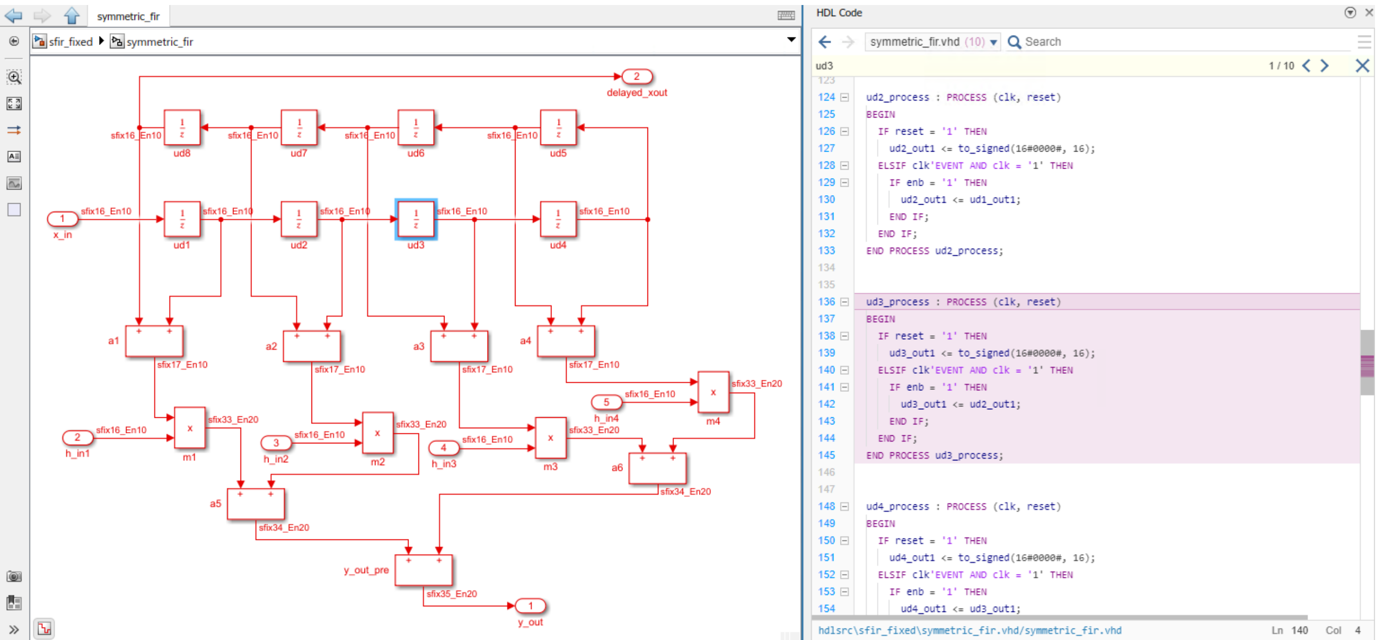
Use model-to-code traceability to select a component at any level of the model and view the code references to that component in the traceability report. For tracing, you can select these objects:

- Subsystem
- Simulink block
- MATLAB Function block
- Stateflow chart, or these elements of a Stateflow chart:
  - State
  - Transition
  - Truth table
  - MATLAB function inside a chart

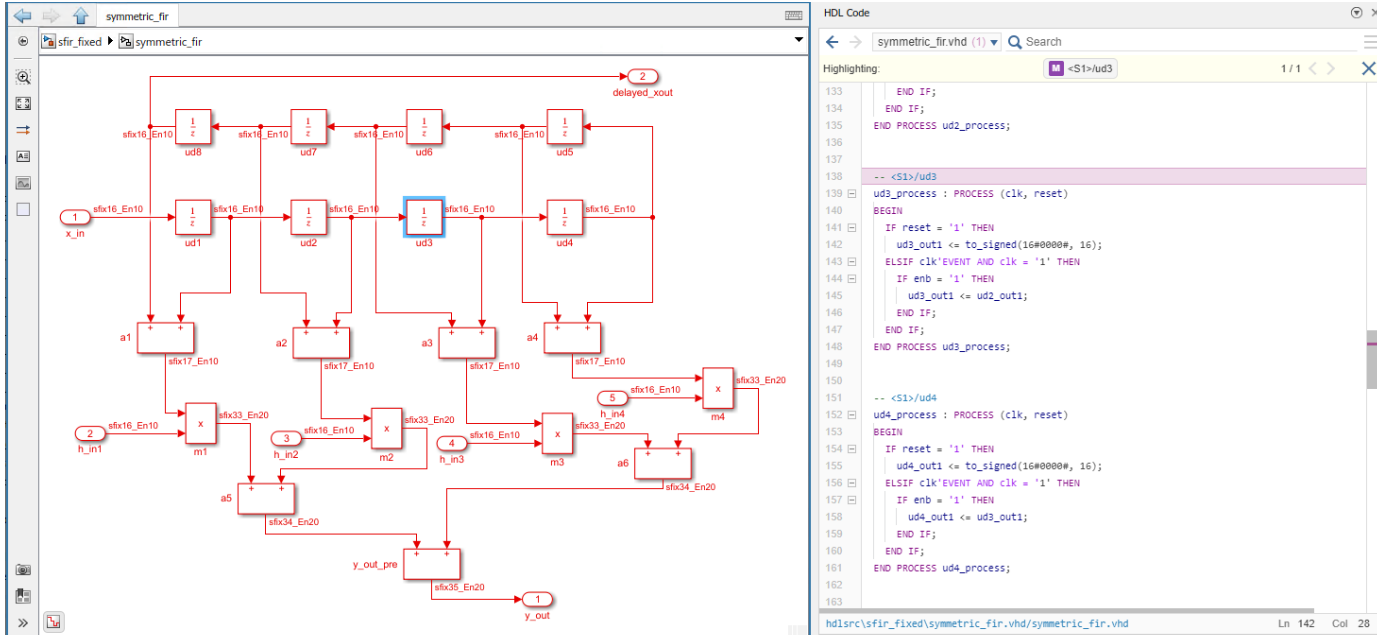
You can navigate from a certain block in the model to the HDL code generated for that block by using any of these approaches.

- Click that block in the model. The Code view highlights the code for the block and scrolls to the highlighted code lines.

  This figure shows an example of how a block highlighted in the model maps to the corresponding highlighted HDL code in the script `symmetric_fir.vhd` when the **Traceability style** is specified as `Line Level`.

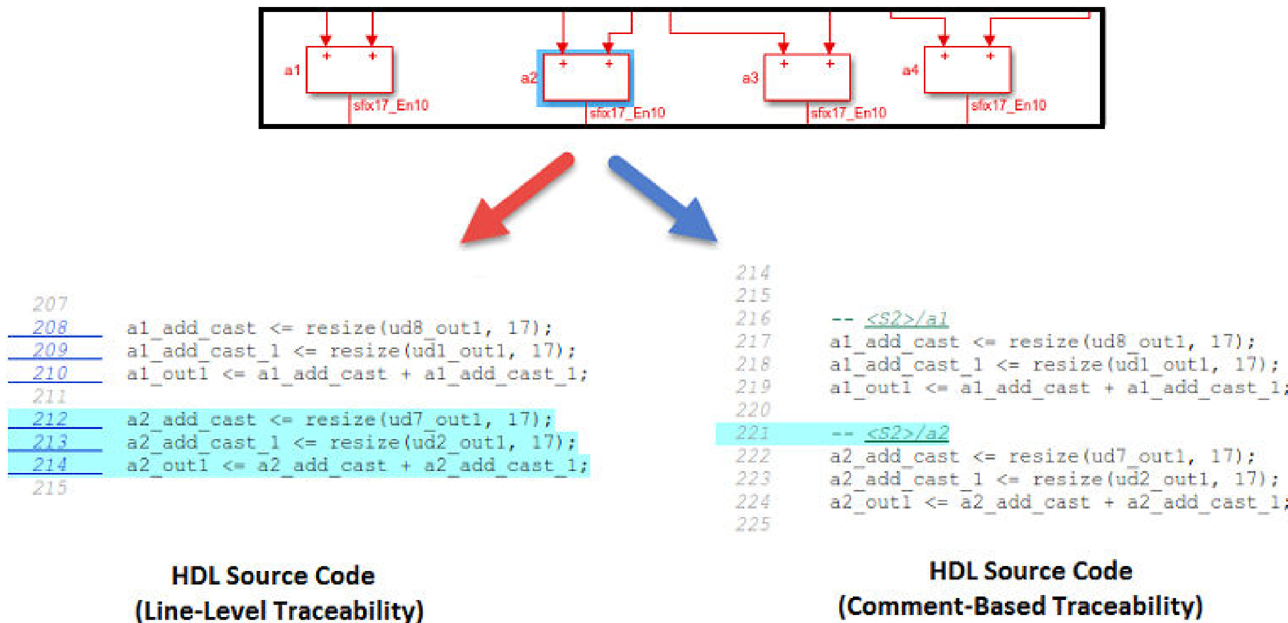


  This figure shows an example of how a block highlighted in the model maps to the corresponding highlighted comment in the HDL code in the script `symmetric_fir.vhd` when the **Traceability style** is specified as `Comment Based`.



- Select that block and click **Navigate to Code** on the **HDL Code** tab.
- Right-click that block in your Simulink model and select **HDL Code** > **Navigate to Code**.

This figure shows the model-to-code navigation for both line-level and comment-based traceability style.



If you use `Line Level` as the **Traceability style** and navigate from the model to the HDL code, the traceability report highlights all lines of HDL code corresponding to that block.

If you use `Comment Based` as the **Traceability style** and navigate from the model to the HDL code, the traceability report highlights the traceable block comment in the HDL code.

Traceability Report Limitations

- If a block name in your model contains a single quote (`'`), code-to-model and model-to-code traceability are disabled for that block.
- If an asterisk (*) in a block name in your model causes a name-mangling ambiguity relative to other names in the model, code-to-model highlighting and model-to-code highlighting are disabled for that block. This is most likely to occur if an asterisk precedes or follows a slash (`/`) in a block name or appears at the end of a block name.
- If a block name in your model contains the character ÿ (`char(255)`), code-to-model highlighting and model-to-code highlighting are disabled for that block.
- If you use certain subsystem types, the Subsystem block is not traceable from the model to the HDL code at the subsystem level. It is possible that you can trace individual blocks within the Subsystem block. You cannot trace from the model to the code for these subsystem types:
  - Virtual
  - Masked
  - Nonvirtual for which code has been optimized away
- Traceability does not support Model Reference as the top-level Subsystem block.

See Also

More About

- "Create and Use Code Generation Reports" on page 23-2

Generate Web View of Model in Code Generation Report

To review and analyze the generated code, it is helpful to navigate between the code and model. If you have a Simulink Report Generator license, you can include a Web view of the model within the HTML code generation report. You can then share your model and generated code outside of the MATLAB environment. When you generate the report, the Web view includes the block diagram attributes displayed in the Simulink Editor, such as, block sorted execution order, signal properties, and port data types. You can display the generated Web views in a supported Web browser. See “Web Views” (Simulink Report Generator) for browser requirements and to read more about Web views.

Generate HTML Code Generation Report with Model Web View

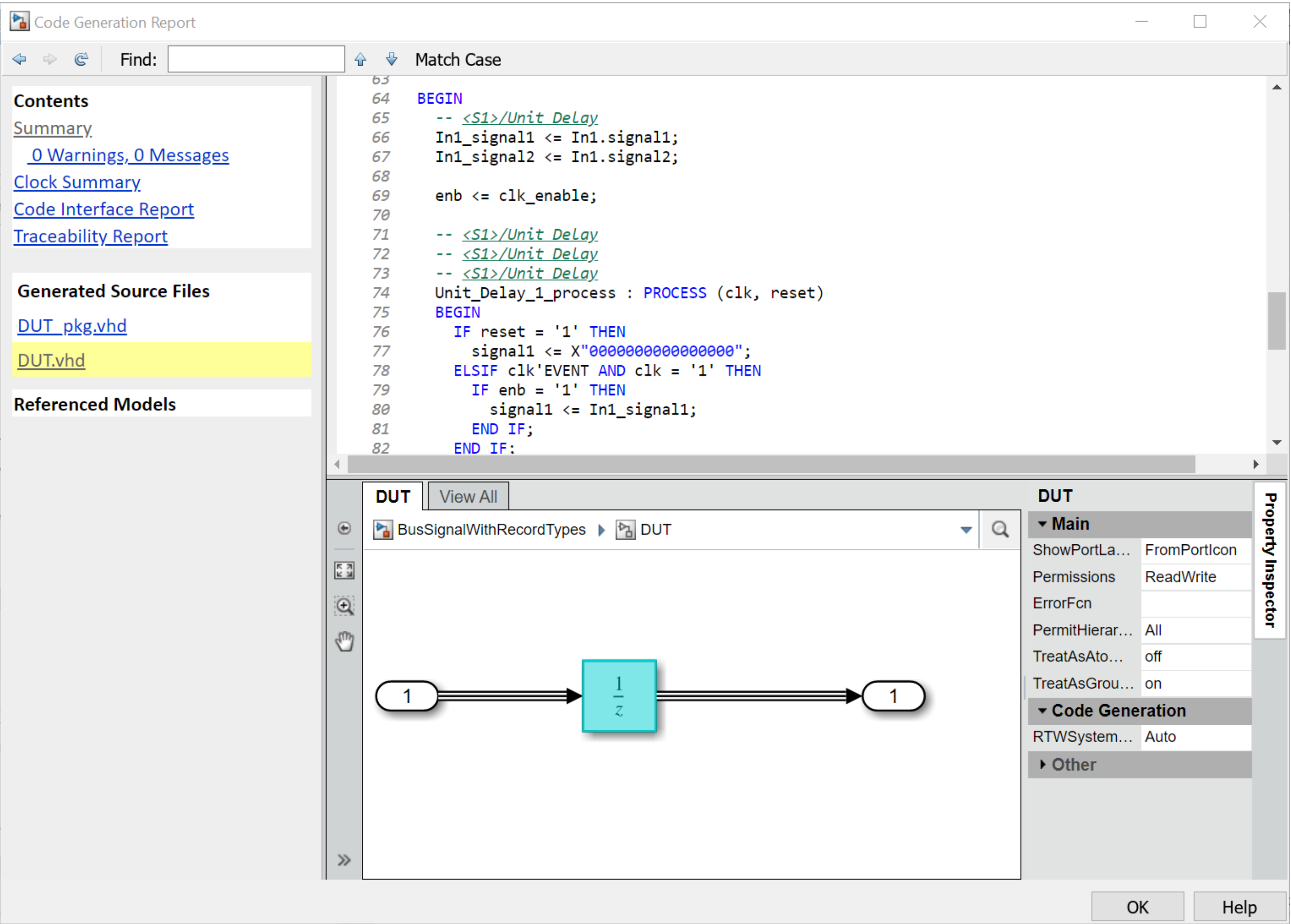
This example shows how to create an HTML code generation report which includes a Web view of the model diagram.

1. From Simulink Toolstrip, open the HDL Coder app.
2. Open the `RunningSum` model. At the command line, enter:

   ```
   openExample('hdlcoder/GenerateRecordTypesForBusSignalsAtSubsystemInterfaceExample',...
   'supportingFile','BusSignalWithRecordTypes');
   ```
3. Open the Configuration Parameters dialog box and navigate to the **HDL Code Generation** > **Report** pane.
4. Configure these parameters:

   - Select **Generate traceability report**.
   - Set **Traceability style** to `Comment Based`.
   - Select **Generate model Web view**.
5. Click **Apply** and save the model.
6. In the **HDL Code** tab, click the **Generate HDL Code** button.

   After building the model and generating code, the code generation report opens in a MATLAB Web browser.
7. In the left navigation pane, select a source code file. The corresponding source code is displayed in the right pane and includes hyperlinked comments.



8 Click a link in the code. The model Web view displays and highlights the corresponding block in the model.
9 To highlight the generated code for a block in your model, click the block. The corresponding code is highlighted in the source code pane.

For more information about exploring a model in a Web view, see “Explore Web Views” (Simulink Report Generator).

Model Web View Limitations

The HTML code generation report includes the following limitations when using the model Web view:

- Code is not generated for virtual blocks. In the model Web view of the code generation report, when tracing between the model and the code, when you click a virtual block, it is highlighted yellow.
- In the model Web view, you cannot open a referenced model diagram by double-clicking the Model block in the top model. Instead, open the code generation report for the referenced model by using the drop-down menu at the top of the report.
- Stateflow truth tables, events, and links to library charts are not supported in the model Web view.
- Searching in the code generation report does not find or highlight text in the model Web view.

- If you navigate from the actual model diagram (not the model Web view in the report), to the source code in the HTML code generation report, the model Web view is disabled and not visible. To enable the model Web view, open the report again.
- For a subsystem build, the traceability hyperlinks of the root level inport and outport blocks are disabled.
- “Traceability Limitations” (Embedded Coder) that apply to tracing between the code and the actual model diagram.

See Also

More About

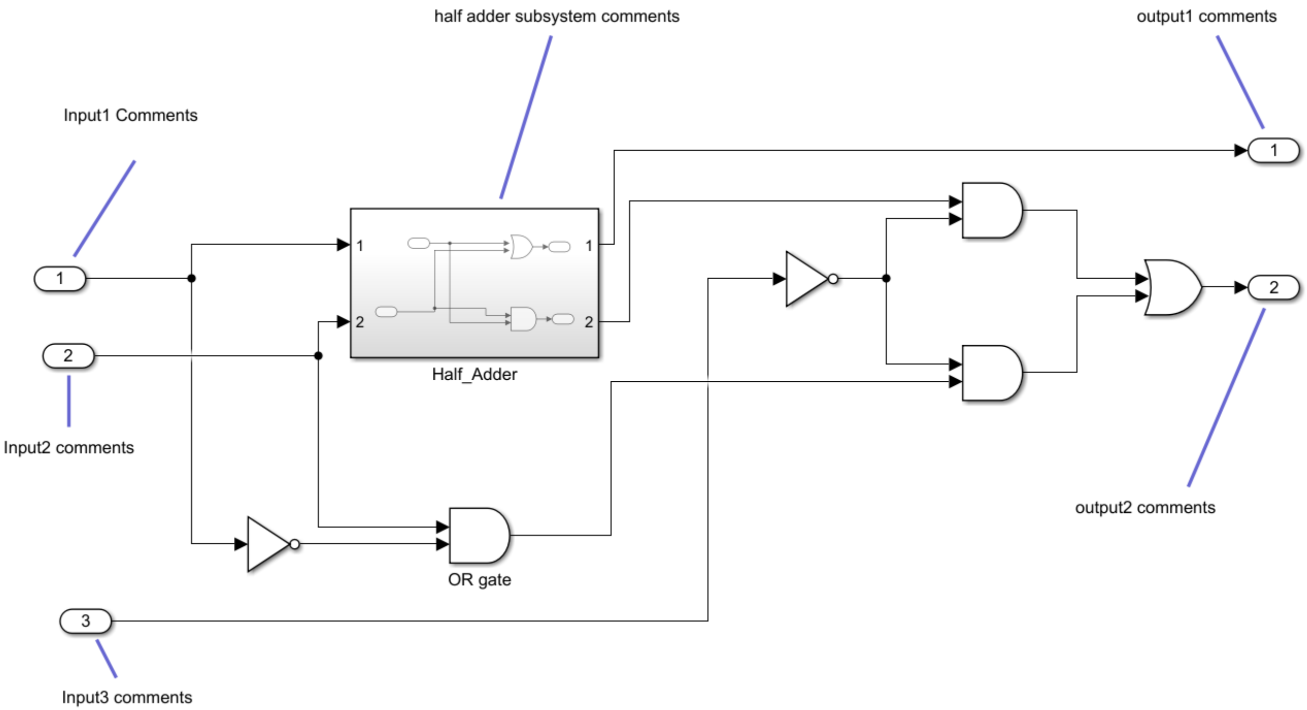
- “Web Views” (Simulink Report Generator)
- “Reports for Code Generation” (Embedded Coder)
- “Generate Code Generation Report” (Embedded Coder)
- “Configure and Generate Code Generation Report Programmatically” (Embedded Coder)

Generate HDL Code with Annotations or Comments

You can use the HDL Coder software to create text annotations to generated HDL code from the model annotations, text comments, or requirements in Simulink.

Simulink Annotations

You can generate code with comments for blocks, including Inport and Outport blocks, by using Simulink annotations. When you add Simulink annotations to the model, HDL Coder renders the text from these annotations as plain text comments in generated code. The comments are generated at the same level in the model hierarchy as the subsystem that contains the annotations. The figure shows a model that has Simulink annotations for Inports, Outports, and other blocks.



This code snippet displays the generated VHDL code, which includes text comments for the blocks.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY DUT IS
  PORT( -- inport1 comment
        In1                               :   IN    std_logic;
        -- inport2 comment
        In2                               :   IN    std_logic;
         -- outport1 comment
        Out1                              :   OUT   std_logic;
        -- outport2 comment
        Out2                              :   OUT   std_logic
        );
END DUT;
```

```
ARCHITECTURE rtl OF DUT IS
  -- Component Declarations
  COMPONENT Subsystem
    PORT( In1                             :   IN    std_logic;
          In2                             :   IN    std_logic;
          Out1                            :   OUT   std_logic
          );
  END COMPONENT;
BEGIN

-- instantiated module comment

  u_Subsystem : Subsystem
    PORT MAP( In1 => Not_gate_out1,
              In2 => Logical_Operator_out1,
              Out1 => Subsystem_out1
              );

  Logical_Operator_out1 <= In1 OR In2;
  Not_gate_out1 <=  NOT Logical_Operator_out1;
  Out1 <= Not_gate_out1;
  Out2 <= Subsystem_out1;

END rtl;
```

For Constant blocks, to convert annotations to comments in the HDL code, in the Configuration Parameters window:

- In the **HDL Code Generation > Global Settings** pane, in the **Coding style** tab, clear the **Minimize Intermediate Signals** check box.
- In the **HDL Code Generation > Report** pane,set **Traceability style** to `Comment Based`

For more information about annotations, See “Annotate Models”.

Signal Descriptions

When you provide a description for the signals in your Simulink model, the generated HDL code displays these descriptions as comments above the signal declaration statements. To specify a description for the signal, right-click the signal and select **Properties** to open the Signal Properties dialog box. Select the **Documentation** tab and enter a description for the signal in the **Description** field. For the signal description, use ASCII characters. Non-ASCII characters in the generated code may interfere with downstream synthesis and lint tools. In some cases, due to optimizations that act on the signals, the generated code may not translate all signal descriptions to HDL comments or may create replicas of HDL comments for certain signal descriptions.

Text Comments

If you enter text comments in your model by using a DocBlock block, HDL Coder renders the text from the DocBlock block in the generated code as plain text comments. The comments are generated at the same level in the model hierarchy as the subsystem that contains the DocBlock block.

Set the **Document type** parameter of the DocBlock block to `Text`. HDL Coder does not support the `HTML` or `RTF` options.

For more information about the DocBlockblock, see DocBlock.

Requirement Comments and Hyperlinks

By using Requirements Toolbox™, you can assign requirements to model elements and generate them as comments in the generated code. For example:

1. Open the model `hdlcoder_simple_up_counter` using this command:

   ```
   openExample("hdlcoder_simple_up_counter");
   ```
2. Open the Requirements Editor and create a new requirement set named `Delay_requirements.slreqx`.

   For more information, see “Author Requirements in MATLAB or Simulink” (Requirements Toolbox).

   Create requirements with these details:

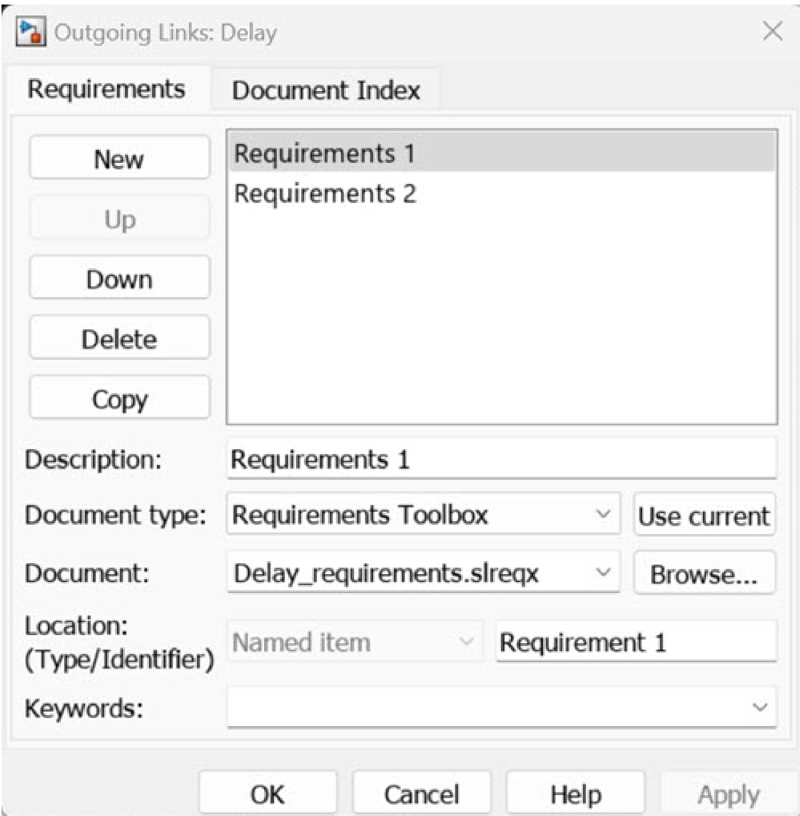
   - **Custom ID**: 1
   - **Summary**: Requirement 1
   - **Description**: Sample text 1

   Similarly, create another requirement with these details:

   - **Custom ID**: 2
   - **Summary**: Requirement 2
   - **Description**: Sample text 2

   Tip You can create links without leaving the Simulink Editor by using the Requirements Perspective. For more information, see “View and Link Requirements in Simulink” (Requirements Toolbox).

   ---
3. Open the `HDL_DUT` subsystem.
4. Right-click the Delay block named `Delay`. From the context menu, select **Requirements** > **Open Outgoing Links dialog**.
5. Link the requirements. Click the **New** button. Set the **Description** to `Requirements 1`, set **Document type** to `Requirements Toolbox`, set **Document** to `Delay_requirements.slreqx`, and **Location (Type/Identifier)** to `Requirement 1`.



Repeat the steps 4 and 5 and link the Requirement 2 to the Subsystem.

6. Click **Apply** and **OK**.
7. To include requirements as hyperlinked comments in the generated code, in the Configuration Parameters dialog box:

   - In the **HDL Code Generation > Report** pane, select **Generate traceability report**
   - In the **HDL Code Generation > Global settings** pane, in the **Comments** tab, select **Include Requirements in Block Comments**

   Alternatively, if you generate code from the command line, set the `Traceability` and `RequirementComments` properties using the `makehdl` function.

   ```
   makehdl("HDL_DUT","Traceability","on","RequirementComments","on");
   ```

   The generated HDL code includes comments for the Delay block and `HDL_DUT` subsystem within the `hdlcoder_simple_up_counter` model, which contains links to the associated requirements for each block and subsystem from the original model.

   ```
   //-------------------------------------------------------------
   // Module: HDL_DUT
   // Source Path: hdlcoder_simple_up_counter/HDL_DUT
   // Hierarchy Level: 0
   // Model version: 9.2
   //
   //  Block requirements for <Root>/HDL_DUT
   //  1. Delay_requirements.slreqx:Requirement 2
   //
   ```

```
// -------------------------------------------------------------

module HDL_DUT
          (clk,
           reset,
           clk_enable,
           count_threshold,
           Enable,
           ce_out,
           out);
  input   clk;
  input   reset;
  input   clk_enable;
  input   [7:0] count_threshold;  // uint8
  input   Enable;
  output  ce_out;
  output  [7:0] out;  // uint8

  // Block requirements for <S1>/Delay1
   //  1. Delay_requirements.slreqx:Requirement 1

   always @(posedge clk or posedge reset)
    begin : Delay1_process
      if (reset == 1'b1) begin
        Delay1_out1 <= 8'b00000000;
      end
      else begin
        if (enb) begin
          Delay1_out1 <= Switch1_out1;
        end
      end
    end

  assign out = Delay1_out1;
  assign ce_out = clk_enable;

endmodule  // HDL_DUT
```

See Also

Related Examples

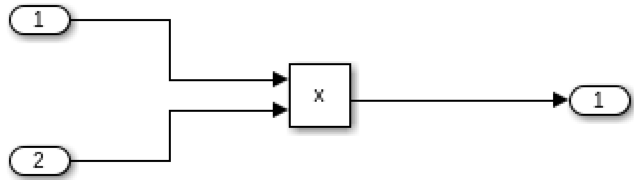
- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 23-12
- “Link Directly to Requirements in Third-Party Applications” (Requirements Toolbox)

Check Your Model for HDL Compatibility

This example shows how to check whether a subsystem or model is compatible for HDL code generation by using the HDL compatibility checker. The HDL compatibility checker examines the specified system for compatibility problems, such as use of unsupported blocks, illegal data type usage, and so on. The HDL compatibility checker generates an HDL Code Generation Check Report. The report is stored in the target `hdlsrc` folder. The report file naming convention is system_report.html, where system is the name of the subsystem or model that is passed to the HDL compatibility checker. The HDL Code Generation Check Report is displayed in a MATLAB™ web browser window. Each entry in the HDL Code Generation Check Report has hyperlinks to the block or subsystem that is not compatible for HDL code generation.

Open this Simulink™ model that has a Product block inside a `DUT` Subsystem. The inputs to the block are a mix of double and integer data types.

```
load_system('hdlcoder_product_mixed_types')
open_system('hdlcoder_product_mixed_types/DUT')
```



To check whether the DUT Subsystem is compatible for HDL code generation, run the compatibility checker. To run the checker from the command line, use the checkhdl function. To learn more about the checkhdl function, see `checkhdl`.

```
checkhdl('hdlcoder_product_mixed_types/DUT', ...
    'TargetDirectory','C:/HDL_Checks/hdlsrc')
```

```
### Running HDL checks on the model 'hdlcoder_product_mixed_types'.
### Begin compilation of the model 'hdlcoder_product_mixed_types'...
### Creating HDL Code Generation Check Report file:///home/jdirner/Documents/MATLAB/Examples/hdl
### HDL check for 'hdlcoder_product_mixed_types' complete with 2 errors, 0 warnings, and 0 messa
```

HDL check for 'hdlcoder_product_mixed_types' complete with 1 errors, 1 warnings, and 0 messages.

The following table describes blocks for which errors, warnings or messages were reported.

Simulink Blocks and resources	Level	Description
hdlcoder_product_mixed_types/DUT/Product	Error	Unhandled mixed double, single, and non-real datatypes at ports of block.
hdlcoder_product_mixed_types/DUT	Warning	Signals of type 'Double' will not generate synthesizable HDL. Consider enabling native floating-point mode and retyping all 'Double' typed signals to 'Single' to generate synthesizable code. More information

Click the `hdlcoder_product_mixed_types/DUT/Product` link to highlight the Product block inside the DUT Subsystem.

To run the compatibility checker from the UI:

1. Open the Configuration Parameters dialog box or the Model Explorer. Select the **HDL Code Generation** pane.
2. From the **Generate HDL for** dropdown, select the `DUT` Subsystem you want to check.
3. Click the **Run Compatibility Checker** button.

For a Subsystem that passes the HDL compatibility check, the HDL Code Generation Check Report contains a hyperlink to that subsystem.

Display Blocks for HDL Code Generation in Library Browser

The `hdllib` function displays the blocks that are compatible with HDL code generation in the Library Browser in standalone mode. If you construct models by using blocks from this Library Browser, your models are compatible with HDL code generation.

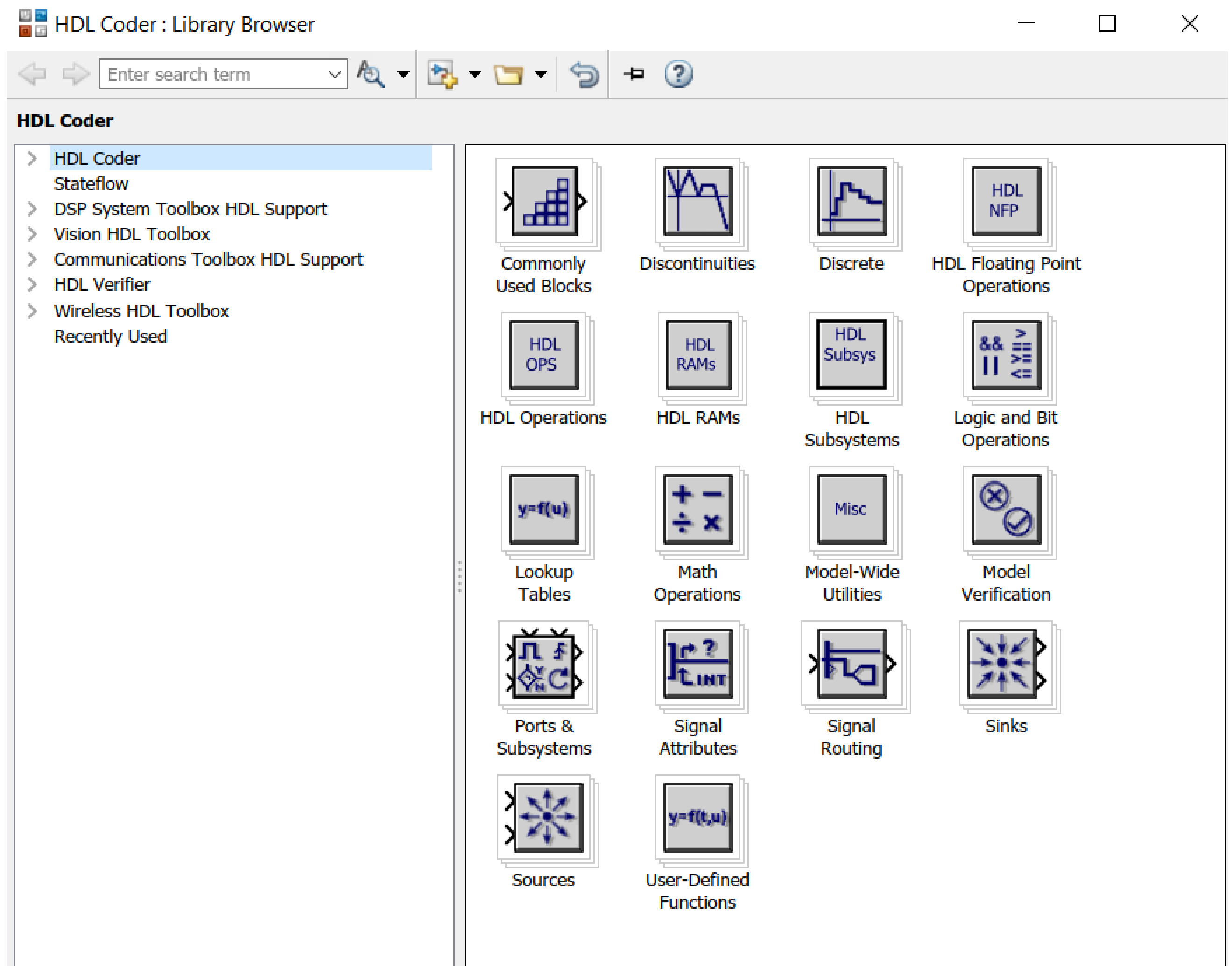
In the Library Browser, parameter settings for blocks are compatible with HDL code generation, and therefore can differ from the default settings.

To display the blocks that are compatible with HDL code generation:

- In the **Apps** tab, select **HDL Coder**. In the **HDL Code** tab, select **HDL Block Properties** > **Open HDL Block Library**.
- Alternatively, at the command prompt, enter:

```
hdllib
```

The Library Browser opens in standalone mode showing blocks compatible with HDL code generation.

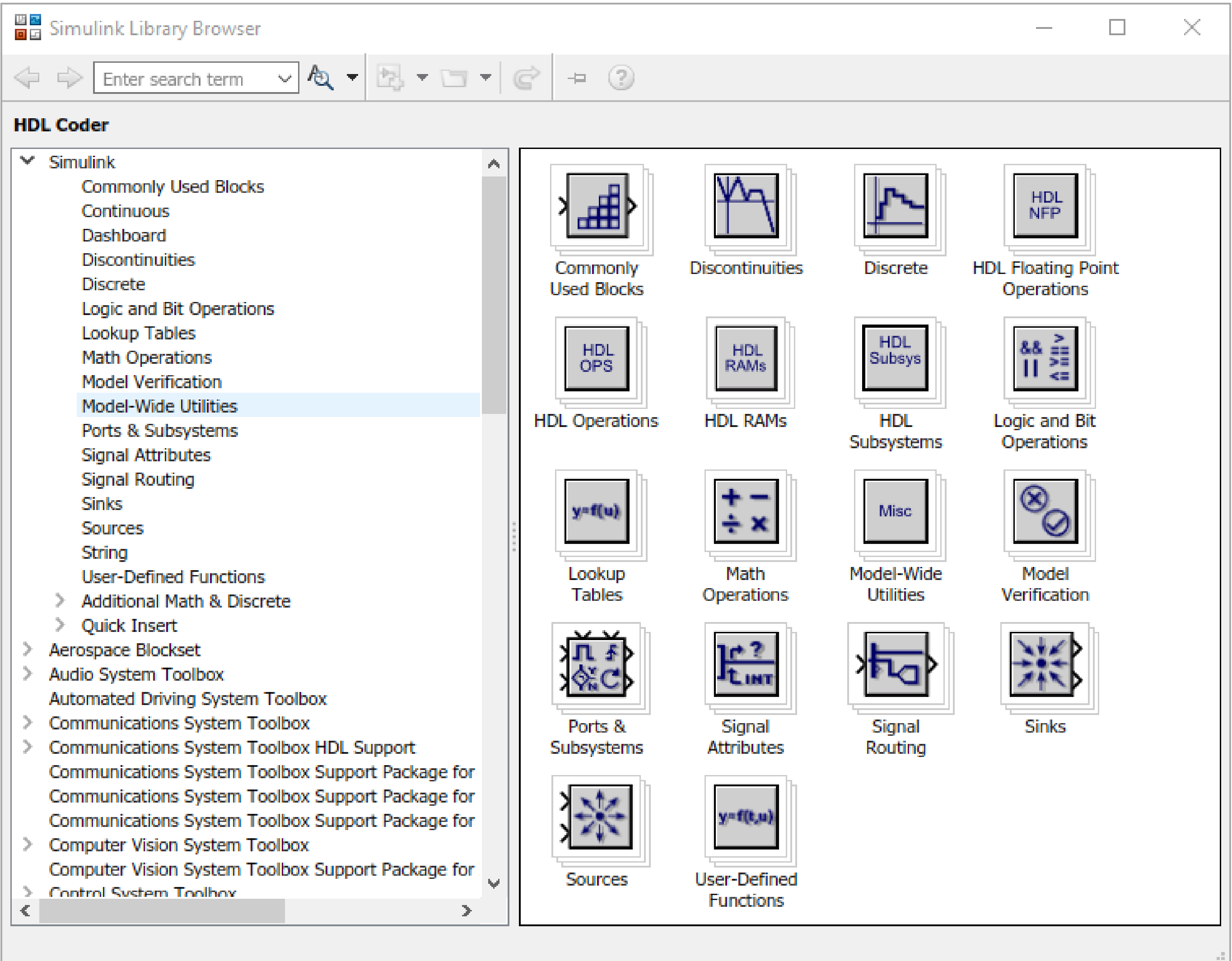


You can drag the blocks from the Library Browser into your model.

If you close and reopen the Library Browser in the same MATLAB session, you continue to see only the blocks that are compatible with HDL code generation.

To reset the Library Browser to display all blocks, regardless of HDL code generation compatibility, enter this command in the MATLAB command window:

```
hdllib('off')
```



To change the Library Browser to display only those blocks that are compatible with HDL code generation, click the button.

Generate Table of Supported Blocks

To generate an HTML table that lists the blocks that are compatible with HDL Code generation:

1. At the command prompt, enter:

```
hdllib('html')
```

`hdllib` creates the `hdlsupported` library and these HTML reports:

```
### HDL supported block list hdlblklist.html
### HDL implementation list hdlsupported.html
```

2. To see the generated list of blocks, click the `hdlblklist.html` link.

See Also

Tools
`hdllib`

Tools
Library Browser in Standalone Mode

See Also

More About

- “Create HDL-Compatible Simulink Model”
- “View HDL-Supported Blocks and HDL-Specific Block Documentation” on page 19-2

Trace Code Using the Mapping File

Note This section refers to generated VHDL entities, Verilog or SystemVerilog modules generically as “entities.”

A mapping file is a text report file generated by `makehdl`. Mapping files are generated as an aid in tracing generated HDL entities back to the corresponding systems in the model.

A mapping file shows the relationship between systems in the model and the VHDL entities, Verilog or SystemVerilog modules that were generated from them. A mapping file entry has the form

```
path --> HDL_name
```

where *`path`* is the full path to a system in the model and *`HDL_name`* is the name of the VHDL entity, Verilog or SystemVerilog module that was generated from that system. The mapping file contains one entry per line.

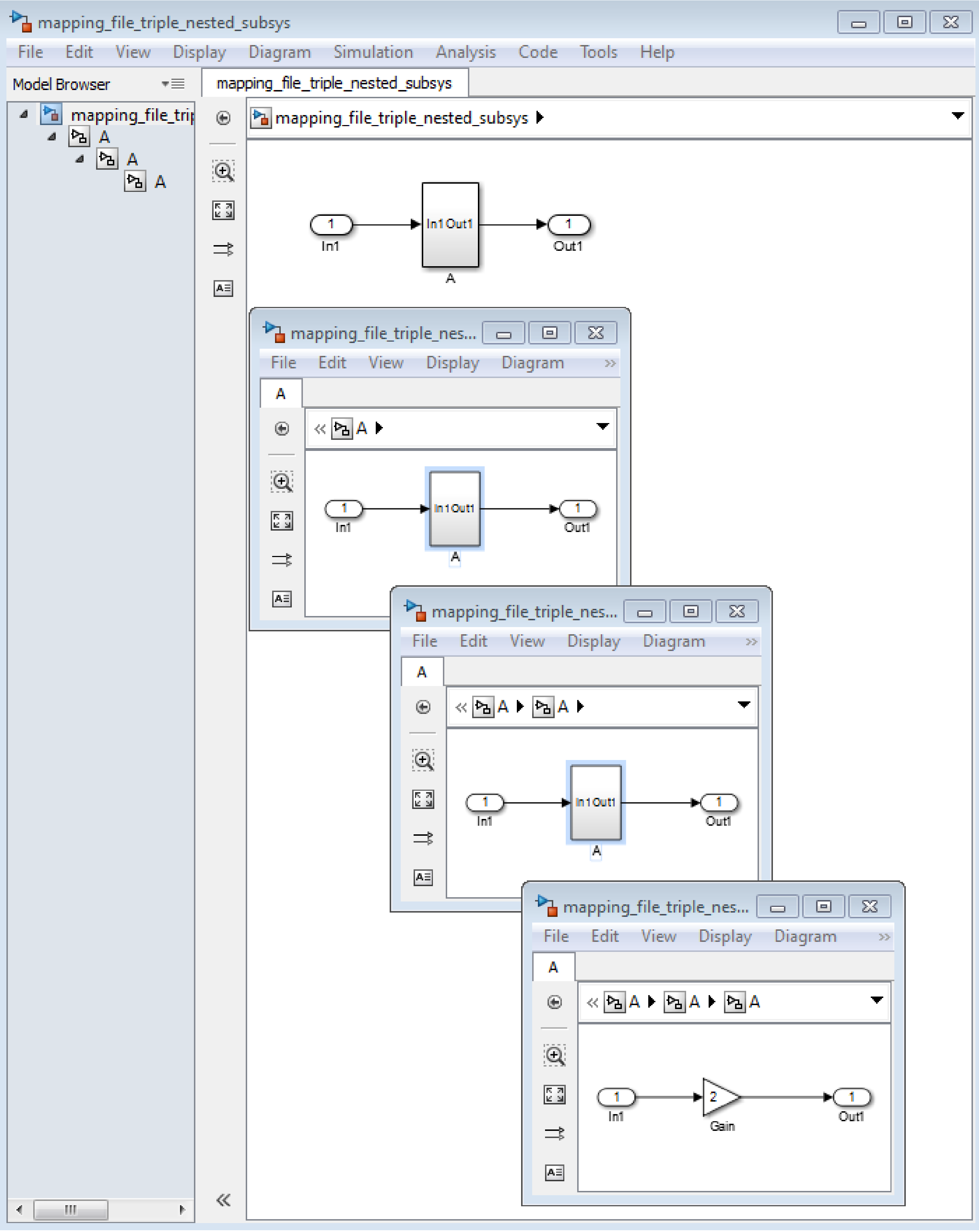
In simple cases, the mapping file may contain only one entry. For example, the `symmetric_fir` subsystem of the `sfir_fixed` model generates the following mapping file:

```
sfir_fixed/symmetric_fir --> symmetric_fir
```

Mapping files are more useful when HDL code is generated from complex models where multiple subsystems generate many entities, and in cases where conflicts between identically named subsystems are resolved by HDL Coder.

If a subsystem name is unique within the model, HDL Coder simply uses the subsystem name as the generated entity name. Where identically named subsystems are encountered, the coder attempts to resolve the conflict by appending a postfix string (by default, `'_entity'`) to the conflicting subsystem. If subsequently generated entity names conflict in turn with this name, incremental numerals (`1,2,3,...`*`n`*) are appended.

As an example, consider the model shown in the following figure. The top-level model contains subsystems named `A` nested to three levels.


mapping_file_triple_nested_subsys
File Edit View Display Diagram Simulation Analysis Code Tools Help
Model Browser
mapping_file_triple_nested_subsys
In1
Out1
A
mapping_file_triple_nes...
Gain

When code is generated for the top-level subsystem A, makehdl works its way up from the deepest level of the model hierarchy, generating unique entity names for each subsystem.

```
makehdl('mapping_file_triple_nested_subsys/A')
### Working on mapping_file_triple_nested_subsys/A/A/A as A_entity1.vhd
### Working on mapping_file_triple_nested_subsys/A/A as A_entity2.vhd
### Working on mapping_file_triple_nested_subsys/A as A.vhd

### HDL Code Generation Complete.
```

The following example lists the contents of the resultant mapping file.

```
mapping_file_triple_nested_subsys/A/A/A --> A_entity1
mapping_file_triple_nested_subsys/A/A --> A_entity2
mapping_file_triple_nested_subsys/A --> A
```

Given this information, you can trace a generated entity back to its corresponding subsystem by using the open_system command, for example:

```
open_system('mapping_file_triple_nested_subsys/A/A')
```

Each generated entity file also contains the path for its corresponding subsystem in the header comments at the top of the file, as in the following code excerpt.

```
-- Module: A_entity2
-- Simulink Path: mapping_file_triple_nested_subsys/A
-- Hierarchy Level: 0
```

Add or Remove the HDL Configuration Component

In this section...

"HDL Configuration Component" on page 23-37
"Add the HDL Configuration Component to a Model" on page 23-37
"Remove the HDL Configuration Component from a Model" on page 23-37

HDL Configuration Component

The HDL configuration component is an internal data structure that HDL Coder creates and attaches to a model. This component lets you view and set HDL code generation options in the **HDL Code Generation** pane in the Configurations Parameters dialog box. These are situations where you may want to add or remove the HDL configuration component:

- If you create a model on a system that does not have HDL Coder installed, the model does not have the HDL configuration component attached. In this case, you may want to add the HDL configuration component to the model.
- If a previous user removed the HDL configuration component from a model, you may want to add the HDL configuration component back to the model.
- If you run a model on a system that has HDL Coder installed, and on another system that does not, you may want to keep the model consistent between both environments. If so, you may want to remove the HDL configuration component from the model.

Add the HDL Configuration Component to a Model

To add the HDL configuration component to a model, in the Simulink Toolstrip, in the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. In the **HDL Code** tab, select **Add HDL Configuration**.

Remove the HDL Configuration Component from a Model

To remove the HDL configuration component from a model, in the **HDL Code** tab, select **Settings > Remove HDL Configuration from model**.

HDL Coding Standards

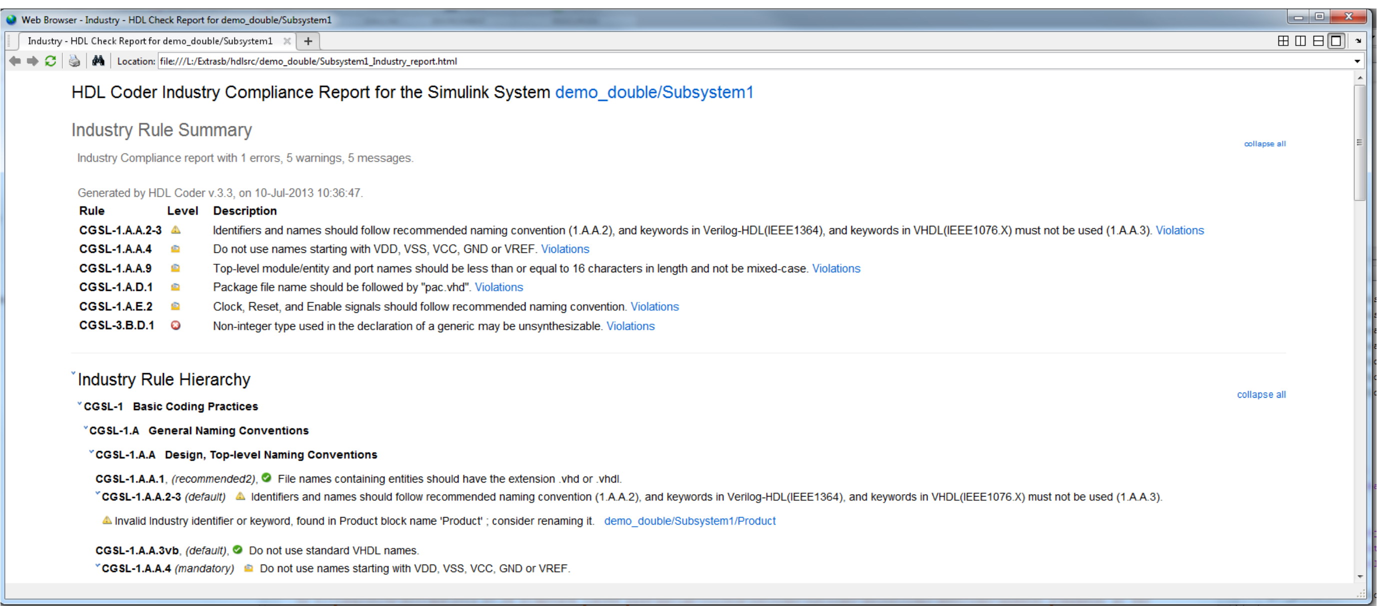
- “HDL Coding Standard Report” on page 24-2
- “HDL Coding Standards” on page 24-4
- “Generate HDL Coding Standard Report from Simulink” on page 24-5
- “Basic Coding Practices” on page 24-7
- “RTL Description Rules and Checks” on page 24-18
- “RTL Design Methodology Guidelines” on page 24-44
- “Generate HDL Lint Tool Script” on page 24-49
- “Cascaded Conditional Region Variable Assignments” on page 24-51

HDL Coding Standard Report

The HDL coding standard report shows how your generated HDL code conforms to an industry coding standard you select when generating code.

The report can contain errors, warnings, and messages. Errors and warnings in the report link to elements in your original design so you can fix problems, then regenerate code. Messages show where HDL Coder automatically corrected the code to conform to the coding standard.

The report also lists the rules in the coding standard with which the generated code complies. You can inspect the report to see which coding standard rules the coder checks.



To learn more about HDL coding standards, see “HDL Coding Standards” on page 24-4.

Rule Summary

The rule summary section shows the total numbers of errors, warnings, and messages, and lists the corresponding rules. Each rule shown in the summary links to the rule in the detailed rule hierarchy section.

Rule Hierarchy

The rule hierarchy section lists every rule HDL Coder checks, within three categories:

- Basic coding practices, including rules for names, clocks, and reset.
- RTL description techniques, including rules for combinatorial and synchronous logic, operators, and finite state machines.
- RTL design methodology guidelines, including rules for ports, function libraries, files, and comments.

If your HDL code does not conform to a specific rule, the rule shows either the automated correction, or a link to the original design element causing the error or warning. When you click a link, the

design opens with the design element highlighted. You can fix the problem in your design, then regenerate code.

Rule and Report Customization

You can configure the report so that it does not display passing rules by using the `ShowPassingRules` property of the HDL coding standard customization object. You can also disable or customize coding standard rules. See HDL Coding Standard Customization.

How to Fix Warnings and Errors

To learn more about warnings and errors you can fix by modifying your design, see:

- "Basic Coding Practices" on page 24-7
- "RTL Description Rules and Checks" on page 24-18
- "RTL Design Methodology Guidelines" on page 24-44

See Also

Related Examples

- "Generate an HDL Coding Standard Report from MATLAB" on page 5-28
- "Generate HDL Coding Standard Report from Simulink" on page 24-5

HDL Coding Standards

Industry coding standards recommend using certain HDL coding guidelines. HDL Coder generates code that follows industry standard rules and generates a report that shows how well your generated HDL code conforms to industry coding standards. See “HDL Coding Standard Report” on page 24-2.

HDL Coder checks for conformance of your Simulink model or MATLAB algorithm to the HDL coding standard rules.

The coder can also generate third-party lint tool scripts to use to check your generated HDL code. The industry standard rules fall under the following three sections:

- Section 1: “Basic Coding Practices” on page 24-7.
- Section 2: “RTL Description Rules and Checks” on page 24-18.
- Section 3: “RTL Design Methodology Guidelines” on page 24-44.

When generating a coding standard report, HDL Coder adds a prefix to the rules. The rule prefix depends on whether you generate the report from MATLAB or Simulink. The rule prefix for MATLAB is CGML and for Simulink is CGSL.

To fix errors or warnings related to these rules, update your model design. You can customize some of the coding standard rules. See HDL Coding Standard Customization.

HDL coding standards provide language-specific code usage rules to help you generate more efficient, portable, and synthesizable HDL code, such as coding guidelines for:

- Names
- Ports, reset, and clocks
- Combinatorial and synchronous logic
- Finite state machines
- Conditional statements and operators

See Also

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-28
- “Generate HDL Coding Standard Report from Simulink” on page 24-5

Generate HDL Coding Standard Report from Simulink

In this section...

"Using the Configuration Parameters Dialog Box" on page 24-5
"Using the Command Line" on page 24-5

You can generate an HDL coding standard report that shows how well your generated code follows industry standards. You can optionally customize the coding standard report and the coding standard rules.

Using the Configuration Parameters Dialog Box

To generate an HDL coding standard report using the Configuration Parameters dialog box:

1. To open the Configuration Parameters dialog box, in the Apps gallery, click **HDL Coder**. The **HDL Code** tab in the Simulink toolstrip appears. In the **Prepare** section, click **Settings**.
2. In the **HDL Code Generation** > **Global Settings** pane of the Configuration Parameters dialog box, select the **Coding standards** tab.
3. For the **HDL coding standard** parameter, select **Industry** and click **Apply**.
4. Optionally, use the other options in the **Coding standards** tab to customize the coding standard rules and click **Apply**.
5. You can then generate HDL code by clicking the **Generate HDL Code** button in the **HDL Code** tab in the Simulink toolstrip.

After you generate code, the message window shows a link to the HTML compliance report. To open the report, click the report link.

Using the Command Line

To generate an HDL coding standard report using the command-line interface, set the `HDLCodingStandard` property to `Industry` by using `makehdl` or `hdlset_param`.

For example, to generate HDL code and an HDL coding standard report for a subsystem, `sfir_fixed/symmetric_sfir`, enter the following command:

```
makehdl('sfir_fixed/symmetric_fir','HDLCodingStandard','Industry')
```

```
### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.vhd
### Industry Compliance report with 4 errors, 18 warnings, 5 messages.
### Generating Industry Compliance Report symmetric_fir_Industry_report.html
### Generating SpyGlass script file sfir_fixed_symmetric_fir_spyglass.prj
### HDL code generation complete.
```

To open the report, click the report link.

You can customize the coding standard report and coding standard rule checks by specifying an HDL coding standard customization object. For example, for a subsystem, `sfir_fixed/`

`symmetric_sfir`, you can create an HDL coding standard customization object, *`cso`*, set the maximum if-else statement chain length to 5 by using the `IfElseChain` property, and generate code:

```
cso = hdlcoder.CodingStandard('Industry');
cso.IfElseChain.length = 5;
makehdl('sfir_fixed/symmetric_fir','HDLCodingStandard','Industry', ...
        'HDLCodingStandardCustomizations',cso)
```

See Also

Properties
HDL Coding Standard Customization

Model Settings
HDL coding standard | Show passing rules in coding standard report

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-28

More About

- “HDL Coding Standard Report” on page 24-2
- “Basic Coding Practices” on page 24-7
- “RTL Description Rules and Checks” on page 24-18
- “RTL Design Methodology Guidelines” on page 24-44

Basic Coding Practices

In this section...

"1.A General Naming Conventions" on page 24-8

"1.B General Guidelines for Clocks and Resets" on page 24-14

"1.C Guidelines for Initial Reset" on page 24-15

"1.D Guidelines for Clocks" on page 24-16

"1.F Guidelines for Hierarchical Design" on page 24-17

HDL Coder conforms to the following naming conventions and basic coding guidelines and checks for modeling constructs that violate these rules. HDL Coder reports potential rule violations in the HDL coding standard report. To avoid these violations, see the rule recommendations.

1.A General Naming Conventions

1.A.A Design and Top-Level Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.A.1 Warning	`Verilog/ SystemVerilog: Source file name should be same as the name of the module in the file.`	By default, HDL Coder generates code that has the same module and file name. If you use `BlackBox` architecture for your subsystem and generate code, the source names and file names can be different.	If you use `BlackBox` architecture for your subsystem, make sure that the source file name and module name are the same.
	`VHDL: File names containing entities should have the extension .vhd or .vhdl.`	Source file name has to use certain recommended naming conventions and file extensions.	Use the **VHDL file extension option** in the HDL Workflow Advisor, or the `VHDLFileExtension` property from the command line.
1.A.A.2 Message	`Verilog/VHDL/ SystemVerilog: Identifiers and names should follow recommended naming convention.`	A name in the design does not start with a letter or contains a character other than a number, letter, or underscore.	Update the names in your design so that they start with a letter of the alphabet (`a-z`, `A-Z`), and contain only alphanumeric characters (`a-z`, `A-Z`, `0-9`) and underscores (_).
1.A.A.3 Message	`Verilog/VHDL/ SystemVerilog: Keywords in Verilog-HDL(IEEE1364), SystemVerilog(v3.1a) , and keywords in VHDL(IEEE1076.X) must not be used.`	There are Verilog, SystemVerilog, or VHDL keywords within the names in your design.	Update the names in your design so that they do not contain Verilog, SystemVerilog, or VHDL keywords. You can disable this rule checking by using the `HDLKeywords` property of the HDL coding standard customization object.
1.A.A.3vb Message	`VHDL: Do not use standard VHDL names.`	HDL Coder does not use standard VHDL names.	No action required.
1.A.A.4 Error	`Verilog/VHDL/ SystemVerilog: Do not use names starting with VDD, VSS, VCC, GND or VREF.`	A name or names in the design are not using the standard naming convention.	Update the names in your design so that they start with a letter of the alphabet (`a-z`, `A-Z`), and contain only alphanumeric characters (`a-z`, `A-Z`, `0-9`) and underscores (_).

Rule / Severity	Message	Problem	Recommendations
1.A.A.5 Error	`Verilog/VHDL/ SystemVerilog: Do not use case variants of name in the same scope.`	Two or more names in your design, within the same scope, are identical except for case. For example, the names `foo` and `Foo` cannot be in the same scope.	Update the names in your design so that no two names within the same scope differ only in case. You can disable this rule checking by using the `DetectDuplicateNamesCheck` property of the HDL coding standard customization object.

1.A.A.6 Warning	`Verilog/ SystemVerilog: Primary port names or module names must follow recommended naming convention.`
	
	`VHDL: Component name should be same as its corresponding entity name.`
		HDL Coder generates code that complies with this rule for Verilog, SystemVerilog and VHDL.	No action required.

1.A.A.9 Warning	`Verilog/VHDL/ SystemVerilog: Top-level module/entity and port names should be less than or equal to 16 characters in length and not be mixed-case.`	A top-level module, entity, or port name in the generated code is longer than 16 characters, or uses letters with mixed case.	Update the indicated name in your design so that it is less than or equal to 16 characters long, and all letters are lowercase. all letters must be either all uppercase or all lowercase. You can customize this rule by using the `ModuleInstanceEntityNameLength` property of the HDL coding standard customization object.

1.A.B Module Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.B.1-1b Error	`Verilog/ SystemVerilog: Module and Instance names should be between 2 and 32 characters in length. The instance names including hierarchy should be less than or equal to 128 characters in length.` `VHDL: Entity names and instance names should be between 2 and 32 characters in length. The instance names including hierarchy should be less than or equal to 128 characters in length.`	A module, instance, or entity name in the generated code is fewer than 2 characters or more than 32 characters in length.	Update the indicated name in your design so that it is from 2 through 32 characters in length. You can customize this rule by using the `ModuleInstanceEntityNameLength` property of the HDL coding standard customization object.

1.A.C Signal Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.C.3 Error	`Verilog/ SystemVerilog: Signal names, port names, parameter names, define names and function names should be between 2 and 40 characters in length.` `VHDL: Signal names, variable names, type names, label names, and function names should be between 2 and 40 characters in length.`	A signal, port, parameter, define, or function name in the generated code is fewer than 2 characters, or more than 40 characters in length.	Update function names or subsystem names in your design to be from 2 through 40 characters in length. You can customize this rule by using the `SignalPortParamNameLength` property of the HDL coding standard customization object.

1.A.D File, Package, and Parameter Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.D.1 Warning	`Verilog/ SystemVerilog: Include files must have extensions that match ".h", ".vh",".inc", and ".h", ".inc", "ht", ".tsk" for testbench.`	The generated include files match these extensions for the testbench.	No action required.
	`VHDL: Package file name should be followed by "pac.vhd".`	By default, the generated package file postfix is `_pkg`.	In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > General** pane, specify the **Package postfix** to `_pac`.
1.A.D.4 Warning	`Verilog/ SystemVerilog: Macros defined outside a module must not be used in the module.`	HDL Coder does not generate macros in the Verilog or SystemVerilog code, or redefine constants in the VHDL code.	No action required.
	`VHDL: Constants should not be redefined.`		
1.A.D.9 Warning	`Verilog/ SystemVerilog: Bit-width must be specified for parameters with more than 32 bits.`	HDL Coder does not specify a bit-width greater than 32 bits in the generated code.	No action required.
	`VHDL: Generic must not be used at top-level module.`	If you use generics at top-level module or if you have mask parameters in your design and set the `MaskParameterAsGeneric` property, HDL Coder reports this violation.	If you have mask parameters in your design, set the `MaskParameterAsGeneric` to `off`.

1.A.E Register and Clock Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.E.2 Warning	`Verilog/VHDL/ SystemVerilog: Clock, Reset, and Enable signals should follow recommended naming convention.`	The clock, reset, and enable signals are not using the recommended naming convention.	In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings** pane, using the **clock input port**, **reset input port**, and **clock enable input port** options, update the names for the clock, reset, and enable signals respectively. Follow these naming conventions: • Clock signal names must be `clk` or `ck`. • Clock enable signal names must be `en`. • For 'active-high' reset, reset signal names must be `rst` or `reset`. • For 'active-low' reset, reset signal names must be `rst`, `reset`, `rst_n`, `reset_n`, `rst_x`, or `reset_x`.

1.A.F Architecture Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.F.1 Warning	`VHDL: Architecture name must contain RTL.`	In the generated VHDL code, the architecture name does not contain RTL.	In **HDL Code Generation > Global Settings > General** tab, update the **VHDL architecture name** to use an architecture name that contains RTL.
1.A.F.4 Warning	`VHDL: An entity and its architecture must be described in the same file.`	By default, HDL Coder describes the entity and architecture of the VHDL code in the same file. If you set the `SplitEntityArch` property to `on`, the generated VHDL code describes the entity and architecture in separate files, so HDL Coder reports a warning.	Set `SplitEntityArch` to off so that HDL Coder describes the entity and architecture of the VHDL code in the same file.

1.B General Guidelines for Clocks and Resets

1.B.A Clock Constraints

Rule / Severity	Message	Problem	Recommendations
1.B.A.1 Message	`VHDL: Design should have only a single clock and use only one edge of the clock.`	Your design uses multiple edges of the clock or contains more than one clock signals. If you set the **ClockInputs** property to `multiple` or use `TriggerAsClock` to use the trigger signal for a triggered subsystem as clock, HDL Coder generates this message.	Update your design to use a single clock signal. In the **HDL Code Generation > Global Settings** panel, set **Clock inputs** to `Single`, and **Clock edge** to `Rising` or `Falling`.
1.B.A.2 Error	`Verilog/VHDL/SystemVerilog: Do not create an RS latch or flip-flop using primitive cells such as AND, OR.`	HDL Coder does not create latches, and complies with this rule.	No action required.
1.B.A.3 Error	`Verilog/VHDL/SystemVerilog: Remove combinational loops.`	HDL Coder does not create combinational loops.	No action required.

1.C Guidelines for Initial Reset

1.C.A Flip-Flop Clock Constraints

Rule / Severity	Message	Problem	Recommendations
1.C.A.3 Warning	`Verilog/VHDL/ SystemVerilog: Do not use asynchronous set/reset signals other than initial reset.`	HDL Coder does not use asynchronous reset signals as non-reset or synchronous reset signals.	No action required.
1.C.A.6 Error	`Verilog/VHDL/ SystemVerilog: Signals must not be used as both asynchronous reset and synchronous reset.`	HDL Coder adds the reset control logic outside the DUT and does not generate both asynchronous reset and synchronous reset signals.	No action required.
1.C.A.7 Warning	`Verilog/VHDL/ SystemVerilog: A flip-flop must not have both asynchronous set and asynchronous reset.`	HDL Coder does not generate code with both asynchronous set and reset signals.	No action required.

1.C.B Reset Conventions

Rule / Severity	Message	Problem	Recommendations
1.C.B.1a Message	`Verilog/VHDL/ SystemVerilog: Asynchronous resets or sets must not be gated.`	HDL Coder does not gate asynchronous set or reset signals.	No action required.
1.C.B.1b Message	`Verilog/VHDL/ SystemVerilog: Reset must be generated in separate module instantiated at top-level.`	The generated code complies with this rule, because the DUT does not contain reset instantiation.	No action required.
1.C.B.2 Warning	`Verilog/VHDL/ SystemVerilog: Do not use signals other than initial reset for asynchronous reset input of flip-flop.`	HDL Coder uses only initial reset signals for asynchronous reset input of flip-flop.	No action required.

1.D Guidelines for Clocks

1.D.A Clock Packaging Constraints

Rule / Severity	Message	Problem	Recommendations
1.D.A.1 Warning	`Verilog/VHDL/ SystemVerilog: Clock should be generated in separate module or entity instantiated at top-level.`	HDL Coder generates code that complies with this rule, because the DUT does not contain clock instantiation.	No action required.

1.D.C Clock Gating Constraints

Rule / Severity	Message	Problem	Recommendations
1.D.C.2–4 Message	`Verilog/VHDL/ SystemVerilog: Do not use flip-flop outputs as clocks of other flip-flops and flip-flop clock signals as non-clock signals.`	HDL Coder does not use the output of flip-flops as clocks of other flip-flops, or flip-flop clock signals as nonclock signals.	No action required.
1.D.C.6 Message	`Verilog/VHDL/ SystemVerilog: Do not use flip-flops with inverted edges.`	If your Simulink model uses a Triggered Subsystem block with rising and falling triggers and has `TriggerAsClock` enabled, HDL Coder violates this rule.	Disable `TriggerAsClock` or do not use Triggered Subsystem blocks with both rising and falling triggers in your Simulink model.

1.D.D Clock Hierarchy Constraints

Rule / Severity	Message	Problem	Recommendations
1.D.D.2 Message	`Verilog/ SystemVerilog: One hierarchical level should have a single clock only.`	Your Simulink model uses multiple clock signals.	Update your design to use a single clock signal. In the **HDL Code Generation > Global Settings** panel, set **Clock inputs** to `Single`.

1.F Guidelines for Hierarchical Design

1.F.A Basic Block Size Guidelines

Rule / Severity	Message	Problem	Recommendations
1.F.A.4 **Error**	`Verilog/VHDL/ SystemVerilog: Clock generation, reset generation, RAM, Setup/Hold ensure buffers, and I/O cells must be a module at top-level.`	HDL Coder generates separate modules for the DUT, RAM, timing controller, so that it complies with this rule.	No action required.

See Also

Properties
HDL Coding Standard Customization

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-28
- “Generate HDL Coding Standard Report from Simulink” on page 24-5

More About

- “HDL Coding Standard Report” on page 24-2
- “RTL Description Rules and Checks” on page 24-18
- “RTL Design Methodology Guidelines” on page 24-44

RTL Description Rules and Checks

In this section...

"2.A Guidelines for Combinational Logic" on page 24-18
"2.B Guidelines for "Always" Constructs of Combinational Logic" on page 24-23
"2.C Guidelines for Flip-Flop Inference" on page 24-25
"2.D Guidelines for Latch Description" on page 24-29
"2.E Guidelines for Tristate Buffer" on page 24-30
"2.F Guidelines for Always/Process Construct with Circuit Structure into Account" on page 24-32
"2.G Guidelines for "IF" Statement Description" on page 24-33
"2.H Guidelines for "CASE" Statement Description" on page 24-35
"2.I Guidelines for "FOR" Statement Description" on page 24-38
"2.J Guidelines for Operator Description" on page 24-39
"2.K Guidelines for Finite State Machine Description" on page 24-43

HDL Coder conforms to the following RTL description rules and checks for modeling constructs that violate these rules. HDL Coder reports potential rule violations in the HDL coding standard report. To avoid these violations, see the rule recommendations.

2.A Guidelines for Combinational Logic

2.A.A Combinatorial Logic Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.A.1 Reference	`VHDL: Package IEEE.std_logic_1164 must be included in each entity.`	HDL Coder includes the package in each entity in the generated VHDL code.	No action required.
2.A.A.2 Warning	`Verilog/ SystemVerilog: A function description must assign return values to all possible states of the function.`	HDL Coder does not generate functions for DUT.	No action required.
2.A.A.3 Warning	`Verilog/ SystemVerilog: Check using RTL parsing tool for error prevention.`	HDL Coder generates VHDL, Verilog, SystemVerilog code with the correct syntax and complies with this rule.	No action required.

2.A.B Function Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.B.1 Error	`Verilog/ SystemVerilog: Function statement should not be used for asynchronous reset line logic in an always construct for FF inference.`	HDL Coder does not generate functions for DUT.	No action required.
	`VHDL: Use std_logic or std_logic_vector data types to describe ports of an entity.`	At the inputs and outputs, HDL Coder uses std_logic or std_logic_vector to describe the ports.	No action required.
2.A.B.2-3 Error	`Verilog/ SystemVerilog: Do not use nonblocking assignment, or input argument as input in function description.`	The generated HDL code complies with this rule for Verilog and SystemVerilog.	No action required.
	`VHDL: Use range specification for integer types.`	By default, HDL Coder specifies the range for integer types in the generated code.	No action required.
2.A.B.4 Error	`Verilog/ SystemVerilog: Task constructs should not be used in the design.`	HDL Coder does not use tasks or fork-join constructs in the Verilog and SystemVerilog code.	No action required.
	`VHDL: Do not use bit and bit vector data types in the design.`	HDL Coder does not use bit or bit vector data types in the generated code.	No action required.
2.A.B.5 Error	`Verilog/ SystemVerilog: Clock edges should not be used in a task description.`	When generating Verilog or SystemVerilog code, HDL Coder does not use clock edges in a task description.	No action required.
2.A.B.6 Error	`VHDL: Specify range for std_logic_vector.`	HDL Coder complies with this rule, because the generated VHDL code specifies the range that std_logic_vector uses.	No action required.

2.A.C Bit Width Matching Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.C.1-2 Error	`Verilog/ SystemVerilog: Ensure that bitwidth of function arguments matches that of corresponding function inputs, and bitwidth of function return value matches that of assignment destination signal.`	At module instantiation, HDL Coder enforces type matching, so that it complies with this rule.	No action required.
	`VHDL: Use only 'in', 'out', and 'inout' ports. Do not use buffer and linkage.`	When generating VHDL code, HDL Coder specifies 'IN', 'OUT', or 'INOUT' ports, and does not use buffer or linkage.	No action required.
2.A.C.3 Error	`Verilog/ SystemVerilog: Use concatenation when assigning to multiple signals.`	HDL Coder complies with this rule.	No action required.
	`VHDL: Port mode must be explicitly specified.`	When generating VHDL code, HDL Coder specifies 'IN', 'OUT', or 'INOUT' ports and does not use buffer or linkage.	No action required.
2.A.C.4-5 Error	`Verilog/ SystemVerilog: In function description, do not assign global signals, and return value assignment must be the last statement.`	HDL Coder generates Verilog or SystemVerilog code that complies with this rule.	No action required.
	`VHDL: Input port must not be described with initial value.`	In the generated VHDL code, HDL Coder does not specify an initial value to the input port.	No action required.

2.A.D Operators Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.D.5 Message	`Verilog/ SystemVerilog: Bit-wise operators must be used instead of logical operators in multi-bit operations.`	In the generated Verilog or SystemVerilog code, HDL Coder complies with this rule for multibit operators.	No action required.
2.A.D.6 Message	`Verilog/ SystemVerilog: Reduction of a single-bit or large expression should not be performed.`	By default, HDL Coder does not reduce a single-bit or a large expression. If your design performs bit-reduction operations, the resulting HDL code can perform reduction of a large expression.	Update your design so that there are no calls to bit reduction operations.

2.A.E Conditional Statement Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.E.3 Message	`Verilog/ SystemVerilog: Ensure that conditional expressions evaluate to a scalar.`	HDL Coder complies with this rule.	No action required.

2.A.F Array, Vector, Matrix Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.F.2 Warning	`Verilog/VHDL/ SystemVerilog: LSB of vectors/memory should be zero.`	Your design contains vectors whose LSB has a nonzero value.	Update your design so that the generated code contains vectors or memory whose LSB value is zero.
2.A.F.4 Warning	`Verilog/VHDL/ SystemVerilog: Index variable width should not be too short.`	HDL Coder enforces type matching and ensures that the index variable width is not too short.	No action required.
2.A.F.5 Error	`Verilog/VHDL/ SystemVerilog: Do not use x and z for an array index.`	In the generated code, HDL Coder does not use x or z for an array index.	No action required.

2.A.G Assignment Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.G.1 Error	`VHDL: Direct assignment must be used for aggregates.`	HDL Coder directly assigns aggregates in the generated code without performing any intervening operations.	No action required.

2.A.H Function Return Value Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.H.1 Reference	`VHDL: Constrained arrays should not be used as sub-program description.`	In the generated code, HDL Coder does not use constrained arrays in subprogram description.	No action required.
2.A.H.2 Reference	`VHDL: Specify range for return values in function description when return type is array.`	In function description, when the return type is array, HDL Coder specifies the range for return values in function in the generated code.	No action required.
2.A.H.4-6 Error	`VHDL: In a sub-program description, use only OTHERS clause when specifying aggregates, not use or call a nested subprogram description, and not read Global signals.`	HDL Coder complies with this rule.	No action required.
2.A.H.9-10 Warning	`VHDL: A function must have a return statement, return a valid value in all possible states, and not have any other statement following the return statement.`	HDL Coder complies with this rule.	No action required.

2.A.I Built-in Attribute Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.I.4-5 **Error**	`VHDL: Do not use user-defined attributes, or built-in attributes except range, length, left, right, high, low, reverse_range, and event.`	By default, HDL Coder does not use user-defined attributes in the generated code. If you set HDL block properties, such as **DSPStyle** in your design, the generated code uses synthesis directives.	To fix this error, in your design, clear the HDL block property that you have set for using synthesis directives in the generated code.

2.A.J VHDL Specific Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.J.1-6 **Warning**	`VHDL: In a design, do not use block statements, objects of type record, shared variables, while-loop statements, procedures, or selected signal assignments.`	If your design uses loop statements, HDL Coder generates this warning.	To avoid this warning, update your design so that there are no looping statements.
2.A.J.8-13 **Error**	`VHDL: In a design, do not use access types, alias declarations, bus and register signals, disconnect specifications, waveforms, and attributes that are defined in Synopsys library.`	HDL Coder complies with this rule.	No action required.

2.B Guidelines for “Always” Constructs of Combinational Logic

2.B.A Latch Constraints

Rule / Severity	Message	Problem	Recommendations
2.B.A.2 **Reference**	`Verilog/VHDL/SystemVerilog: Check latch creation from RTL lint checker and synthesis tools; Design should not have latches.`	HDL Coder does not create latches.	No action required.

2.B.B Signal Constraints - I

Rule / Severity	Message	Problem	Recommendations
2.B.B.2-3 Message	`Verilog/VHDL/ SystemVerilog: In the sensitivity list of a process or always block, do not define constants, use wait statements, or include a signal that is not read inside that block.`	HDL Coder generates code that complies with the use of these constructs inside a `process` block (VHDL) or an `always` block (Verilog and SystemVerilog).	No action required.
	`Verilog/ SystemVerilog: Do not describe multiple event expressions with always constructs.`	HDL Coder does not describe more than one event expression in an `always` construct.	No action required.

2.B.C Signal Constraints - II

Rule / Severity	Message	Problem	Recommendations
2.B.C.1-2 Error	`Verilog/ SystemVerilog: Do not use nonblocking assignments in combinational always blocks, or when assigning initial values in always constructs of sequential blocks.`	Your design uses constructs that generate Verilog code with nonblocking assignments in combinational `always` blocks or assigns initial values in `always` constructs of sequential blocks.	Update your MATLAB algorithm or Stateflow design so that the generated Verilog code does not use these constructs.
2.B.C.3 Message	`Verilog/VHDL/ SystemVerilog: Do not assign a signal more than once in an always construct for sequential circuits.`	In an `always` construct for sequential circuits, HDL Coder does not perform multiple assignments to a signal.	No action required.

2.C Guidelines for Flip-Flop Inference

2.C.A Assignment Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.A.1-2c Error	`Verilog/VHDL/ SystemVerilog: In flip-flop description, do not use quasi-continuous assignments, deassign statements, blocking assignments, variable assignment statements, or stable attribute.`	HDL Coder does not introduce any additional data or add these constructs when generating flip-flops in `process` blocks (VHDL) or `always` blocks. (Verilog or SystemVerilog)	No action required.
2.C.A.4-5b Warning	`Verilog/VHDL/ SystemVerilog: Only flip-flop data paths can have delays. The delay values must be integral and non-negative.`	HDL Coder does not generate code that uses `DELAY` attributes for the DUT. The generated testbench can contain `DELAY` attributes.	No action required.
2.C.A.6 Error	`Verilog/VHDL/ SystemVerilog: Check the logic level of the reset signal as specified in the sensitivity list of the always block.`	HDL Coder uses `posedge` or `negedge` to denote transitions at clock edges in the generated code.	No action required.
2.C.A.7 Message	`Verilog/VHDL/ SystemVerilog: A flip-flop should not have two asynchronous resets. Do not use functions in the asynchronous reset description.`	HDL Coder does not generate multiple asynchronous resets. The generated code can contain multiple synchronous resets.	No action required.
2.C.A.8 Error	`VHDL: Do not use wait constructs.`	HDL Coder does not use wait constructs.	No action required.

Rule / Severity	Message	Problem	Recommendations
2.C.A.9 Error	`VHDL: Functions 'rising_edge' or 'falling_edge' should not be used in the design.`	By default, HDL Coder uses the event syntax for clock events. By using the `UseRisingEdge` property, you can specify whether to use the `rising_edge` or `falling_edge` to detect clock transitions.	To fix this error, you can control the `UseRisingEdge` property such that the generated code uses the event syntax.

2.C.B Blocking Statement Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.B.1-2 Warning	`Verilog/VHDL/ SystemVerilog: Use blocking assignment in flip-flop description. Do not use blocking and nonblocking assignments together in the same always block.`	HDL Coder complies with this rule.	No action required.
2.C.B.4 Error	`VHDL: Variables, if used, must be assigned to a signal before the end of the process.`	The generated HDL code does not contain dead code, so HDL Coder complies with this rule.	No action required.

2.C.C Clock Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.C.1-2b Error	`Verilog/VHDL/ SystemVerilog: Do not use edges of multiple clocks or both edges of the same clock in an always block. Do not describe multiple clock edges in a single process/ always block for same edge of a single clock.`	HDL Coder uses the rising edge or falling edge of the clock, but does not use both edges of the clock.	No action required.
2.C.C.4-5 Error	`Verilog/VHDL/ SystemVerilog: Minimize, and if possible, remove clock enable signals and reset signal on networks.`	If your design generates code that uses clock enables and reset signals on networks, HDL Coder generates an error.	To minimize clock enables in the generated HDL code, in the HDL coding standard customization properties, enable the **MinimizeClockEnableCheck** property. To remove reset signals on the networks, in the HDL coding standard customization properties, enable the **RemoveResetCheck** setting.
2.C.C.6 Warning	`Verilog/VHDL/ SystemVerilog: Do not use asynchronous reset signals.`	Your Simulink model design or MATLAB code uses asynchronous reset signals.	To avoid this violation, use synchronous reset signals for your design. In the Configuration Parameters dialog box, set **Reset type** to `Synchronous`.

2.C.D Initial Value Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.D.1 Error	`Verilog/VHDL/SystemVerilog: Do not specify flip-flop or RAM initial value using initial construct.`	The generated HDL code for your design contains an unsynthesizable `initial` statement.	Disable the **Initialize block RAM** or **Initialize all RAM blocks** option in the HDL Workflow Advisor. You can disable this rule checking by using the `InitialStatements` property of the HDL coding standard customization object.

2.C.F Mixed Timing Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.F.1-2a Warning	`Verilog/VHDL/SystemVerilog: Do not use multiple resets or mix descriptions of flip-flops with and without asynchronous reset in the same process/always block.`	HDL Coder complies with this rule.	No action required.

2.D Guidelines for Latch Description

2.D.A Module Constraints

Rule / Severity	Message	Problem	Recommendations
2.D.A.2-3 Warning	`Verilog/VHDL/ SystemVerilog: Latch descriptions should not have asynchronous set or asynchronous reset, or be mixed with other descriptions in the same module.`	HDL Coder does not create latches in the generated code.	No action required.
2.D.A.4-5 Error	`Verilog/VHDL/ SystemVerilog: Do not use combinational loops that contain latches or level two latches in the same phase clock.`	By default, HDL Coder does not create combinational loops. If your MATLAB algorithm contains combinational loops, the generated HDL code can use combinational loops.	Update your MATLAB code so that the generated HDL code does not contain any combinational loops.

2.E Guidelines for Tristate Buffer

2.E.A Module Constraints

Rule / Severity	Message	Problem	Recommendations
2.E.A.1-2 Warning	`Verilog/VHDL/ SystemVerilog: Tristate descriptions must not be mixed with other descriptions in the same module and should not contain logic in tristate enable conditions.`	HDL Coder does not create latches or tristate buffers in the generated code.	No action required.
2.E.A.4-5b Reference	`Verilog/VHDL/ SystemVerilog: Tristate bus must not be driven by more than specified number of drivers. A net that is not tristated or a signal without a resolution function must not have multiple drivers.`	HDL Coder does not create latches or tristate buffers in the generated code.	No action required.
2.E.A.6-9 Error	`Verilog/VHDL/ SystemVerilog: Inout port should not be directly connected to input/output. Do not use tristate output in an if conditional expression or in the selection expression of a case statement that assigns a fixed value in others choice.`	By default, HDL Coder does not connect input or output ports directly to bidirectional ports. In your Simulink model, on the HDL block properties for the input or output port, if you set **BidirectionalPort** to `on`, the generated HDL code can directly connect inout to input or output ports.	In your Simulink model, on the HDL block properties for the input or output port, set **BidirectionalPort** to `off`.

2.E.B Connectivity Constraints

Rule / Severity	Message	Problem	Recommendations
2.E.B.1 Warning	`Verilog/VHDL/ SystemVerilog: Logic directly driven by tristate nets should be in a separate module.`	HDL Coder does not have tristate nets in the generated HDL code.	No action required.

2.F Guidelines for Always/Process Construct with Circuit Structure into Account

2.F.B Constraints on Number of Conditional Statements

Rule / Severity	Message	Problem	Recommendations
2.F.B.1 Error	`Verilog/VHDL/ SystemVerilog: Do not describe more than one statement (if/case/while/for/ loop) separately within a single always or process block.`	The generated HDL code for your design contains more than one conditional statement (if-else, case, and loops) that is described separately within a `process` block for VHDL code or an `always` block for Verilog and SystemVerilog code.	Update your design so that there is not more than one conditional statement that is described separately in a process block. You can customize this rule by using the `ConditionalRegionCheck` property of the HDL coding standard customization object.
2.F.B.1.a Error	`Verilog/VHDL/ SystemVerilog: Do not write to same signal for VHDL or register for Verilog in multiple cascaded conditional (if/ case) regions within the same process block for VHDL code or always construct for Verilog code.`	The generated HDL code for your design contains the same signal for VHDL code or register for Verilog and SystemVerilog code written to in multiple cascaded conditional regions (such as `if/case` regions) in the same `process` block for VHDL code or `always` construct for Verilog and SystemVerilog code.	Update your design so that the signal or register is not written to more than once in cascaded conditional regions in the same process block or always construct. For more information, see "Cascaded Conditional Region Variable Assignments" on page 24-51. You can customize this rule by using the `CascadedConditionalAssignmentCheck` property of the HDL coding standard customization object.
2.F.B.2 Error	`Verilog/VHDL/ SystemVerilog: A variable in the sensitivity list is modified inside the same process or always block.`	HDL Coder does not modify the variables in the sensitivity list, including clock, reset, and enable signals.	No action required.

2.G Guidelines for “IF” Statement Description

2.G.B Common Sub-Expression Constraints

Rule / Severity	Message	Problem	Recommendations
2.G.B.2 Warning	`Verilog/VHDL/ SystemVerilog: Avoid describing conditions that will not be executed.`	The generated HDL code does not contain dead code, or result in conditions that are not executed.	No action required.

2.G.C Nesting Depth Constraints

<table>
<tr><th>Rule / Severity</th><th>Message</th><th>Problem</th><th>Recommendations</th></tr>
<tr><td>2.G.C.1a-b
Message</td><td><code>Verilog/VHDL/ SystemVerilog: Nesting in if-else constructs should not be deeper than N levels. Where feasible case statements should be used, rather than if-else statements, if performance is important.</code></td><td>The MATLAB code contains an <code>if-elseif</code> statement with more than N levels of nesting. By default, N is 3.</td><td>Modify <code>if-elseif</code> statements in your MATLAB code so there are N or fewer levels of nesting.

For example, the following <code>if-elseif</code> pseudocode contains three levels of nesting:

<pre>if ...
   if ...
      if ...
      else
   else
else</pre>
You can customize this rule by using the <code>IfElseNesting</code> property of the HDL coding standard customization object.</td></tr>
<tr><td>2.G.C.1c
Message</td><td><code>Verilog/VHDL/ SystemVerilog: Chain of if...else if constructs must not be exceed default number of levels.</code></td><td>The generated HDL code contains an <code>if-elseif</code> statement with more than seven branches.</td><td>Modify <code>if-elseif</code> statements in your MATLAB code so that the number of branches is seven or fewer.

For example, the following <code>if-elseif</code> pseudocode contains three branches:

<pre>if ...
elseif ...
elseif ...
else</pre>
You can customize this rule by using the <code>IfElseChain</code> property of the HDL coding standard customization object.</td></tr>
</table>

2.G.D Begin-End Decorator Constraints

Rule / Severity	Message	Problem	Recommendations
2.G.D.2-3 Message	`Verilog/VHDL/ SystemVerilog: Attach begin-end to "if" statements.`	The generated HDL code complies with these code constructs.	No action required.
	`Verilog/ SystemVerilog: Do not use fork-join constructs.`		

2.H Guidelines for “CASE” Statement Description

2.H.A CASE Structure Constraints

Rule / Severity	Message	Problem	Recommendations
2.H.A.3-5 Reference	`Verilog/VHDL/ SystemVerilog: case constructs should not have overlapping clause conditions. Do not use full_case directive.`	The generated HDL code complies with these constructs for case statements and does not use the full_case directive.	No action required.

2.H.C Default Value Constraints

Rule / Severity	Message	Problem	Recommendations
2.H.C.3 Warning	`Verilog/ SystemVerilog: Do not use //synposys full_case pragma when all conditions are not described as case clause or the default clause is missing.`	HDL Coder describes all possible cases in a case statement so that the synthesis tool does not infer a latch.	No action required.
2.H.C.4 Message	`Verilog/VHDL/ SystemVerilog: A signal that is assigned don't care value in a case default clause should not be used in if conditions, ternary and case constructs.`	HDL Coder does not use a signal that is assigned a *don't care* value in the default clause.	No action required.
2.H.C.5 Warning	`Verilog/VHDL/ SystemVerilog: Default clause in case construct must be the last clause.`	To avoid latch inference, HDL Coder describes all possible cases, including the default clause.	No action required.
2.H.C.6-7 Message	`Verilog/VHDL/ SystemVerilog: Do not use a signal to which don't care is assigned for selection expression of casex statements or case statements that do not assign 'X' in default clause.`	HDL Coder does not use *don't care* values, and explores the entire space of an n-bit select signal.	No action required.

2.H.D Don’t Care Constraints

Rule / Severity	Message	Problem	Recommendations
2.H.D.1-4 Message	`Verilog/ SystemVerilog: Design should not use casex or casez constructs. casex or casez constructs must contain a dont-care condition, and not have complex clause conditions. The don't care condition in casex or casez branches must follow proper coding style.`	HDL Coder does not generate casex or casez constructs, so that it complies with this rule.	No action required.

2.H.E Additional CASE Constraints

Rule / Severity	Message	Problem	Recommendations
2.H.E.1-4 Message	`Verilog/ SystemVerilog: Do not use parallel_case directive. In a case clause condition, do not use fixed values, variables, expressions, and logical, arithmetic, bitwise, or reduction operations.`	HDL Coder does not use the `parallel_case` directive and generates code that complies with these constructs.	No action required.

2.I Guidelines for “FOR” Statement Description

2.I.A Loop Body Constraints

Rule / Severity	Message	Problem	Recommendations
2.I.A.2a-b Message	`Verilog/ SystemVerilog: Loop variable and terminating condition of "for" construct must have constant initial value.`	HDL Coder does not generate casex or casez constructs so that it complies with this rule.	No action required.
2.I.A.2c-e Message	`Verilog/ SystemVerilog: Loop variable of "for" construct must have a constant value inside the construct and must not be used outside the construct.` `Verilog/ SystemVerilog: The loop termination condition must not be a constant.`	HDL Coder generates the right loop constructs and complies with this rule.	No action required.

2.I.B Non-Constant Operation Constraints

Rule / Severity	Message	Problem	Recommendations
2.I.B.4 Error	`Verilog/VHDL/ SystemVerilog: Separate for loops must be used in reset and logic parts of flip-flop descriptions.`	HDL Coder uses separate for loops in the reset and logic parts of flip-flop descriptions.	No action required.

2.I.C Exit Constraints

Rule / Severity	Message	Problem	Recommendations
2.I.C.1 Error	`VHDL: Exit or next statement must not be used in a for loop.`	The generated code contains for loops only when HDL Coder knows the number of iterations. When the loop is executing, HDL Coder does not exit from the for loop,	No action required.

2.J Guidelines for Operator Description

2.J.A Comparison and Precedence Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.A.4a-c **Message**	`Verilog/ SystemVerilog: Signals must not be compared with X or Z, or values containing X or Z.`	By default, HDL Coder does not generate code that contains these constructs. If your Simulink model design uses Constant blocks with **Architecture** set to `Logic Value` and uses these constructs, the coder displays this message.	Update your Simulink model design so that the Constant blocks do not use these constructs when **Architecture** is set to `Logic Value`. Alternatively, change the **Architecture** to `Constant`.
2.J.A.4v **Error**	`Verilog/VHDL/ SystemVerilog: Do not assign X except for the others clause of case statements.`	By default, HDL Coder does not use X in the others clause of case statements. In certain cases, if the generated code does not comply with **2.J.A.4a-c**, HDL Coder can assign X in the others clause.	Update your Simulink model design so that the generated HDL code does not use constructs that rule **2.J.A.4a-c** specifies.
2.J.A.5-6 **Warning**	`Verilog/ SystemVerilog: Do not use values containing 'X' or 'Z'.`	If your design uses unknown or high-impedance constants, HDL Coder displays a warning.	Update your Simulink model or MATLAB algorithm so that there are no high-impedance constants.
	`VHDL: Do not use values including 'X','Z','U','-','W', 'H','L', or constants that contain the values 'X','Z','U','-','W', 'H','L'.`		
2.J.A.7-8 **Message**	`Verilog/ SystemVerilog: Do not use RAM output signals for a conditional expression of if statements, or selection expression of case statements that assign 'x' in the default clause.`	By default, HDL Coder complies with this rule. If your Simulink model uses RAM output signals with a Switch or Multiport switch block, the generated HDL code can use these constructs.	Update your Simulink model so that there are no RAM output signals to Switch or Multiport switch blocks.

2.J.B Vector Operator Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.B.3 Message	`Verilog/VHDL/ SystemVerilog: Do not perform logical negation on vectors.`	HDL Coder does not perform logical negation on vectors.	No action required.

2.J.C Relational Operator Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.C.1-6 Error	`Verilog/VHDL/ SystemVerilog: Bitwidths of operands of a relational or logical operator must match.`	HDL Coder ensures that the data types of the operands match in a relational or logical expression.	No action required.
	`Verilog/VHDL/ SystemVerilog: Bitwidths should be specified for conditional expression.`		

2.J.D Signed Signal, Data Type Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.D.3-5 Warning	`Verilog/VHDL/ SystemVerilog: Take care when assigning integer to reg or wire, and when comparing negative value reg and integer variables. Integer objects must not be assigned negative values.`	HDL Coder complies with this rule.	No action required.
2.J.D.6 Warning	`VHDL: Signed data type must be used in signed operation and std_logic_vector calling std_logic_unsigned package must be used in unsigned operation.`	HDL Coder complies with this rule.	No action required.
2.J.D.8 Warning	`VHDL: Function To_stdlogicvector should not be used in the design.`	HDL Coder does not use the function `To_stdlogicvector` in the code.	No action required.

2.J.E Number of Operator Repetition Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.E.5 Warning	`Verilog/ SystemVerilog: Do not describe arithmetic operators with conditional operators(?) in assign statement.`	HDL Coder complies with this rule.	No action required.

2.J.F Precision Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.F.5 Warning	`Verilog/VHDL/ SystemVerilog: Large multipliers must not be described using the multiplication operator with RTL.`	The generated HDL code contains a multiplication operator (*) where the output of the multiplication has a bitwidth of 16 or greater.	In your design, implement multiplications by using a shift-and-add algorithm, or ensure that the data size of the output of a multiplication does not require a bitwidth of 16 or greater. You can customize this rule by using the `MultiplierBitWidth` property of the HDL coding standard customization object.

2.J.G Common Sub-Expression Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.G.2 Warning	`Verilog/VHDL/ SystemVerilog: common operational expressions should be described separately.`	HDL Coder identifies the common operational expressions and describes them separately.	No action required.

2.J.H Division Operator Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.H.1 Message	`Verilog/VHDL/ SystemVerilog: Do not use arithmetic and logical expressions in the right and left sides of the division or modulus operator.`	HDL Coder homogenizes the division operator into a separate statement and complies with this rule.	No action required.
2.J.H.2-3 Message	`Verilog/VHDL/ SystemVerilog: Keep the left side of the division or modulus operator within 12 bits. If right side of the division or modulus operator is not a power of two, keep it within 8 bits.`	In your design, the left side of the modulus or division operation is greater than 12 bits, or the right side is not a power of two and greater than eight bits.	Update your design so that the number of bits in the operands of the division or modulus operation are within the bounds that the rule specifies.

2.K Guidelines for Finite State Machine Description

2.K.A State Transition Constraints

Rule / Severity	Message	Problem	Recommendations
2.K.A.4 Warning	`Verilog/VHDL/ SystemVerilog: Number of states of an FSM should be within 40.`	Your model design contains a Stateflow Chart or State Transition Table that uses more than 40 states.	Update your model design so that there are not more than 40 states.

2.K.C Logic Separation Constraints

Rule / Severity	Message	Problem	Recommendations
2.K.C.1 Reference	`Verilog/VHDL/ SystemVerilog: Ensure that sequential and combinational parts of an FSM are in separate always block.`	By default, HDL Coder puts the sequential and combinational parts of a Finite State Machine (FSM) in separate `always` blocks.	No action required.

2.K.E Encoding Constraints

Rule / Severity	Message	Problem	Recommendations
2.K.E.2 Warning	`VHDL: Do not assign state encoding by attaching attributes to the state variable which is declared as a type.`	HDL Coder does not attach attributes to state variables in the generated code.	No action required.

See Also

Properties
HDL Coding Standard Customization

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-28
- “Generate HDL Coding Standard Report from Simulink” on page 24-5

More About

- “HDL Coding Standard Report” on page 24-2
- “Basic Coding Practices” on page 24-7
- “RTL Design Methodology Guidelines” on page 24-44

RTL Design Methodology Guidelines

In this section...

"3.A Guidelines for Creating Function Libraries" on page 24-44

"3.B Guidelines for Using Function Libraries" on page 24-45

"3.C Guidelines for Test Facilitation Design" on page 24-47

HDL Coder conforms to the following RTL design methodology guidelines, and checks for modeling constructs that violate these rules. HDL Coder reports potential rule violations in the HDL coding standard report. To avoid these violations, see the rule recommendations.

3.A Guidelines for Creating Function Libraries

3.A.C Signal, Port Constraints - I

Rule / Severity	Message	Problem	Recommendations
3.A.C.1 Warning	`Verilog/ SystemVerilog: The order of module port decalarations and instance port connections lists should be same as the order in the module port map.`	HDL Coder preserves the order of module port declarations and instance port connections as they appear in the original Simulink DUT.	No action required.
3.A.C.4a Message	`Verilog/VHDL/ SystemVerilog: Define only one port or signal per line in I/O, reg, and wire declaration.`	HDL Coder complies with this rule.	No action required.

3.A.D Signal, Port Constraints - II

Rule / Severity	Message	Problem	Recommendations

3.A.D.4–5 Warning	`Verilog/VHDL/ SystemVerilog: Multiple assignments should not be made in one line.`
	
	`Verilog/VHDL/ SystemVerilog: The maximum number of characters in one line should not be more than N.`
		The generated HDL code contains multiple assignments in one line or lines greater than N characters. You have a name or identifier in your original design that contains more than N characters.	Shorten names in your design that are longer than N characters.
			
			You can also customize N by using the `LineLength` property of the HDL coding standard customization object.
			
			HDL Coder folds the long lines in the design only so far as the HDL code syntax is not broken.

3.A.F Generic Usage Constraints

Rule / Severity	Message	Problem	Recommendations
3.A.F.1 Reference	`Verilog/ SystemVerilog: Generic should be used in conditional expression of if generate statement.`	HDL Coder does not generate if-generate statements, but can generate for-generate statements in the generated HDL code.	No action required.

3.B Guidelines for Using Function Libraries

3.B.B Parameters, Constant Constraints

Rule / Severity	Message	Problem	Recommendations
3.B.B.2b-4 Message	`Verilog/ SystemVerilog: Define macros should be read using include files. Include files must be specified with more than 1 level higher relative path.`	HDL Coder does not generate macros in the HDL code.	No action required.
3.B.B.5-7 Message	`Verilog/ SystemVerilog: Text macros should not be nested, and constants should be defined using parameters only.`	HDL Coder does not generate macros in the HDL code.	No action required.

3.B.C Port Constraints

Rule / Severity	Message	Problem	Recommendations
3.B.C.1 Message	`Verilog/VHDL/SystemVerilog: Port/Generic connections in instantiations must be made by named association rather than position association.`	HDL Coder preserves the association of ports, so that it complies with this rule.	No action required.
3.B.C.2 Message	`Verilog/SystemVerilog: Bit-width of the component port and its connected net must match.`	HDL Coder enforces type and bit-width matching, so that it complies with this rule.	No action required.
3.B.C.3 Message	`VHDL: Do not use entity instantiation in the design.`	HDL Coder does not use entity instantiation in the design. The generated HDL code is generic and reusable.	No action required.

3.B.D Generic Constraints

Rule / Severity	Message	Problem	Recommendations
3.B.D.1 Error	`Verilog/VHDL/SystemVerilog: Non-integer type used in the declaration of a generic may be unsynthesizable.`	The generated HDL code contains a noninteger data type.	If you have floating-point data types in your design, you can map them to HDL Coder native floating-point libraries so that the generated code does not use floating-point data types. Alternatively, modify your design so that it does not use floating-point data types. You can disable this rule checking by using the `NonIntegerTypes` property of the HDL coding standard customization object.
3.B.D.3 Error	`Verilog/SystemVerilog: Do not use defparam statements.`	HDL Coder complies with this rule.	No action required.

3.C Guidelines for Test Facilitation Design

3.C.A Clock Constraints - I

Rule / Severity	Message	Problem	Recommendations
3.C.A.1-4 **Error**	`Verilog/VHDL/ SystemVerilog: Internal clocks and aynschronous sets/ resets must be controllable from external pins.`	In the generated HDL code, you can control clocks from external pins. If you have a triggered subsystem and enable `TriggerAsClock`, then the trigger signal becomes a clock signal that you can control from external pins. For reset signals that you model in Simulink, the generated VHDL code can have a load port, which is a primary input in the generated code.	To avoid this rule violation, disable the `TriggerAsClock`.

3.C.B Black Box Constraints

Rule / Severity	Message	Problem	Recommendations
3.C.B.3 **Error**	`Verilog/VHDL/ SystemVerilog: Do not connect the outputs of a black box to clock, reset, or tristate enable pins.`	HDL Coder connects the clock bundle to the entity or blackbox and does not modify it, so the generated code complies with this rule.	No action required.

3.C.C Clock Constraints - II

Rule / Severity	Message	Problem	Recommendations
3.C.C.1 **Error**	`Verilog/VHDL/ SystemVerilog: A clock must not be connected to the D input of a flip-flop.`	HDL Coder does not use clock as data.	No action required.

3.C.F Clock Constraints - III

Rule / Severity	Message	Problem	Recommendations
3.C.F.2 Error	`Verilog/VHDL/ SystemVerilog: Do not mix clock and reset lines.`	HDL Coder connects the clock bundle to the entity or blackbox and does not modify it, so the generated code complies with this rule.	No action required.

See Also

Properties
HDL Coding Standard Customization

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-28
- “Generate HDL Coding Standard Report from Simulink” on page 24-5

More About

- “HDL Coding Standard Report” on page 24-2
- “Basic Coding Practices” on page 24-7
- “RTL Description Rules and Checks” on page 24-18

Generate HDL Lint Tool Script

You can generate a lint tool script to use with a third-party lint tool to check your generated HDL code.

HDL Coder can generate Tcl scripts for the following lint tools:

- Ascent Lint
- HDL Designer
- Leda
- SpyGlass
- Custom

If you specify one of the supported third-party lint tools, you can either generate a default tool-specific script, or customize the script by specifying the initialization, command, and termination names as a character vector. If you want to generate a script for a custom lint tool, you must specify the initialization, command, and termination names.

HDL Coder writes the initialization, command, and termination names to a Tcl script that you can use to run the third-party tool.

How to Generate an HDL Lint Tool Script

Using the Configuration Parameters Dialog Box

1. In the Configuration Parameters dialog box, select **HDL Code Generation > EDA Tool Scripts**.
2. Select the **Lint script** option.
3. For **Choose lint tool**, select **Ascent Lint**, **HDL Designer**, **Leda**, **SpyGlass**, or **Custom**.
4. Optionally, enter text to customize the **Lint initialization**, **Lint command**, and **Lint termination** strings. For a custom tool, specify these fields.

After you generate code, the message window shows a link to the lint tool script.

Using the Command Line

To generate an HDL lint tool script from the command line, set the `HDLLintTool` parameter to `AscentLint`, `HDLDesigner`, `Leda`, `SpyGlass`, or `Custom` using `makehdl` or `hdlset_param`.

To disable HDL lint tool script generation, set the `HDLLintTool` parameter to `None`.

For example, to generate HDL code and a default SpyGlass lint script for a DUT subsystem, `sfir_fixed\symmetric_fir`, enter the following:

```
makehdl('sfir_fixed/symmetric_fir','HDLLintTool','SpyGlass')
```

After you generate code, the message window shows a link to the lint tool script.

To generate an HDL lint tool script with custom initialization, command, and termination names, use the `HDLLintTool`, `HDLLintInit`, `HDLLintTerm`, and `HDLLintCmd` parameters.

For example, you can use the following command to generate a custom Leda lint script for a DUT subsystem, `sfir_fixed\symmetric_fir`, with custom initialization, command, and termination names:

```
makehdl('sfir_fixed/symmetric_fir','HDLLintTool','Leda',...
        'HDLLintInit','myInitialization','HDLLintCmd','myCommand %s',...
        'HDLLintTerm','myTermination')
```

Custom Lint Tool Command Specification

If you want to generate a lint tool script for a custom lint tool, you must use `%s` as a placeholder for the HDL file name in the generated Tcl script.

Specify the **Lint command** or `HDLLintCmd` using the following format:

```
hdlset_param ('HDLLintCmd', 'custom_lint_tool_command -option1 -option2 %s')
```

For example, to set `HDLLintCmd`, where the lint command is *`custom_lint_tool_command -option1 -option2`*, at the command line, enter:

```
hdlset_param ('HDLLintCmd', 'custom_lint_tool_command -option1 -option2 %s')
```

Cascaded Conditional Region Variable Assignments

In R2021b, HDL Coder provides a coding standard (Guideline 2.F.B.1.a in "RTL Description Rules and Checks" on page 24-18) to check for assignments to the same variable in multiple cascaded control regions within the same process block. HDL Coder points to the blocks that generate such a coding style. If the style is not recommended for use in your production workflow, consider an alternative coding style or replace the block pointed to by the rule. The check displays an error if the generated HDL code for your design contains the same variable/signal for `VHDL` or register/wire for `Verilog` written to in multiple cascaded conditional regions, such as `switch case` or `if/else` cascaded regions, within the same process block. To turn on this check for your model, see Check for assignments to the same variable in multiple cascaded control regions.

Example Patterns that Fail the Check for Guideline 2.F.B.1.a

These examples show HDL Code patterns that fail the check for the presence of assignments to the same variable in multiple cascaded conditional regions.

Parallel Cascaded 'if' Regions

```
always @(u) begin
    if(u < 8'sd4) begin
        y = -8'sd1;
    end
    if(u == 8'sd1) begin
        y = 8'sd10;
    else
        y =8'sd2;
    end
end
```

Parallel Cascaded 'if' Regions with Nesting

```
always @(u, k) begin
    if (u < 8’sd4) begin
        if(k == 8’sd1) begin
            y = 8'sd10;
        end
    end
    if (u < 8’sd2) begin
        y = 8'sd1;
    else
        y = 8’sd2;
    end
end
```

Parallel Cascaded 'if' and 'switch' Regions

```
always @(u) begin
    if (u < 8’sd4) begin
        y = 8'sd10;
    end
    case (u)
            8’sd1:
                y = 8’sd11;
            default:
                y = 8’sd4;
```

```
    endcase
end
```

Parallel Cascaded 'switch' Regions

```
always @(u) begin
    case (u)
            8’sd2:
                y = 8’sd15;
            default:
                y = 8’sd4;
    endcase
    case (u)
            8’sd1:
                y = 8’sd11;
            default:
                y = 8’sd4;
    endcase
end
```

Parallel Cascaded Region Inside Nested Region

```
always @(u) begin
    case (u)
            8’sd2:
                y = 8’sd15;
            default:
                 if (u < 8’sd4) begin
                     y = 8'sd10;
                 end
    if (u == 8’sd1) begin
                     y = 8'sd10;
                 else
                     y = 8’sd2;
                 end
    endcase
end
```

Example Patterns that Pass the Check for Guideline 2.F.B.1.a

These examples show HDL Code patterns that pass the check for the presence of assignments to the same variable in multiple cascaded conditional regions if you enable the check.

Default Value and Conditional Assignment

```
always @(u) begin
    y = 8’sd1;
    if (u < 8’sd4) begin
        y = 8'sd10;
    end
end
```

Parallel cascaded “if” Regions Writing to Different Outputs

```
always @(u) begin
    if (u < 8’sd4) begin
        y1 = -8'sd1;
```

```
    else
        y1 = 8'sd3;
    end
    if (u == 8'sd1) begin
        y2 = 8'sd10;
    else
        y2 = 8'sd2;
    end
end
```

Assignment in All Paths of Condition Regions

```
always @(u) begin
    if (u == 8'sd1) begin
        y = 8'sd10;
    else
        y = 8'sd2;
    end
end
```

Simulink Blocks and Modeling Patterns that Fail the Check for Guideline 2.F.B.1.a

Some Simulink blocks and modeling patterns can fail this check (guideline 2.F.B.1.a) and cause an error during HDL code generation. If a block or modeling pattern you use in your code fails this check, consider disabling the check if it is not needed in your production workflow or changing the block or modeling pattern producing the code failure.

MATLAB Function Block

In some cases, MATLAB code written inside MATLAB Function blocks might generate HDL Code that fails this check.

Conditional Initialization of Persistent Variables

If the persistent variables in a MATLAB Function block are conditionally initialized and then overwritten in a conditional region elsewhere in the code, the generated code might fail this check.

For example, if you have a MATLAB Function block in your DUT with the MATLAB code:

```
function y = fcn(u)

persistent loc;

if(isempty(loc) || u == int8(2))
    loc = int8(-1);
end

% persistent variable overwritten in conditional code region
if(u<int8(4))
    loc = int8(10);
end

y = loc;
```

This Verilog code snippet demonstrates the violation of a value assignment to the variable *loc_temp* in two parallel conditional `if` statements.

```
always @(loc, loc_not_empty, u) begin
    loc_temp = loc;
    loc_not_empty_next = loc_not_empty;
    if ( ! loc_not_empty || (u == 8'sd2)) begin
        loc_temp = -8'sd1;
        loc_not_empty_next = 1'b1;
    end
    // persistent variable overwritten in conditional code region
    if (u < 8'sd4) begin
        loc_temp = 8'sd10;
    end
    ...

end
```

Explicit modeling in the MATLAB Function block

If you set the **HDL Architecture** in the HDL Block Properties dialog box of the MATLAB Function block as `MATLAB Function` and design your MATLAB code in a coding style that assigns multiple values to the same variable in cascading conditional regions, an error might occur from the resulting violating patterns in the generated HDL code.

For example, if you have a MATLAB Function block in your DUT with the MATLAB code:

```
function y = fcn(u)

% if region 1
if(u<4)

    % if region 1-1
    if(u==1)
        y = int8(0);
    else
        y = int8(4);
    end

    % if region 1-2
    if(u==2)
        y = int8(5);
    end
else
    y = int8(6);
end

...

end
```

Region 1 in the preceding code contains two `if` statements that assign different values to the same output variable *y*. This pattern causes a violation in the generated Verilog code.

```
module mlfb
            (u,
             y);

    ...

    always @(u) begin
```

```
        // if region 1
        if (u < 8'sd4) begin
            // if region 1-1
            if (u == 8'sd1) begin
                y_1 = 8'sd0;
            end
            else begin
                y_1 = 8'sd4;
            end
            // if region 1-2
            if (u == 8'sd2) begin
                y_1 = 8'sd5;
            end
        end
        else begin
            y_1 = 8'sd6;
        end

        ...

    end

    assign y = y_1;

endmodule // mlfb
```

Conditional Assignments of Matrix Variables

If you assign different indices of a single matrix in multiple cascading conditional regions in a MATLAB Function block, a violation can occur.

For example, if you have a MATLAB Function block in your DUT with the MATLAB code:

```
function y = fcn(u)

y = int8(zeros(1,2));

% if region 1
if(u==1)
    y(1) = int8(2);
end

% if region 2
if(u==2)
    y(2) = int8(5);
end
```

Although two different indices of the matrix *y* are being assigned values in cascading `if` statements, this code generates Verilog code that causes a violation.

```
ARCHITECTURE rtl OF mlfb IS
    -- Signals
    SIGNAL u_signed : signed(7 DOWNTO 0); -- int8
    SIGNAL y_tmp : vector_of_signed8(0 TO 1); -- int8 [2]

BEGIN
    u_signed <= signed(u);
```

```
    mlfb_1_output : PROCESS (u_signed)
    BEGIN
        FOR t_0 IN 0 TO 1 LOOP
            y_tmp(t_0) <= to_signed(16#00#, 8);
        END LOOP;

        -- if region 1
        IF u_signed = to_signed(16#00000001#, 8) THEN
            y_tmp(0) <= to_signed(16#02#, 8);
        END IF;
        -- if region 2
        IF u_signed = to_signed(16#00000002#, 8) THEN
            y_tmp(1) <= to_signed(16#05#, 8);
        END IF;
    END PROCESS mlfb_1_output;

    outputgen: FOR k IN 0 TO 1 GENERATE
        y(k) <= std_logic_vector(y_tmp(k));
    END GENERATE;

END rtl;
```

Workarounds

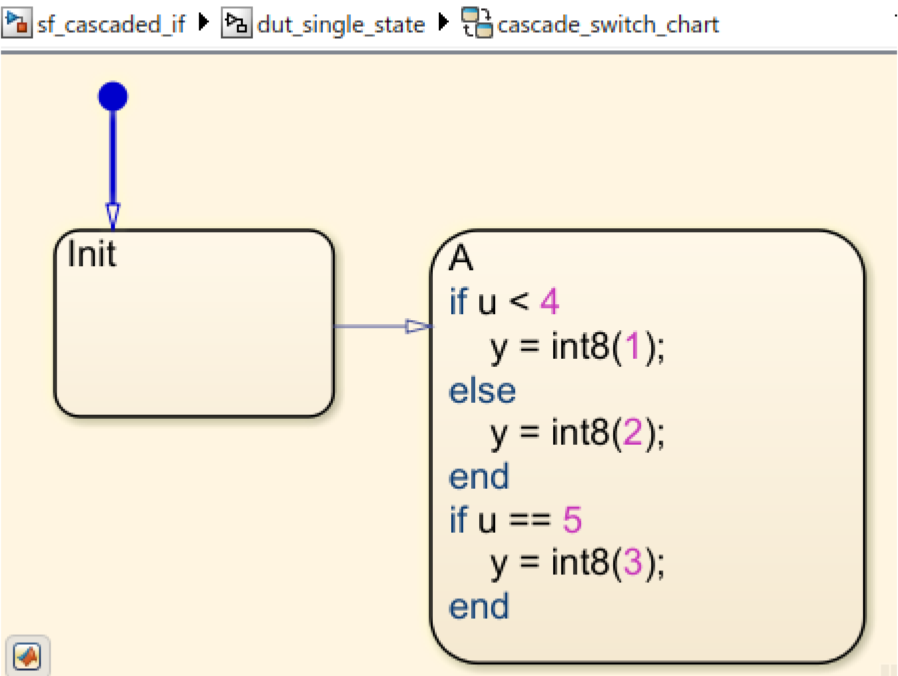
- Design MATLAB code in your MATLAB Function block in a way that avoids patterns that assign multiple values to the same variable in multiple cascaded conditional regions.
- Consider specifying the HDL Block property of the MATLAB Function block **HDL Architecture** to be `MATLAB Datapath`.

Stateflow Charts

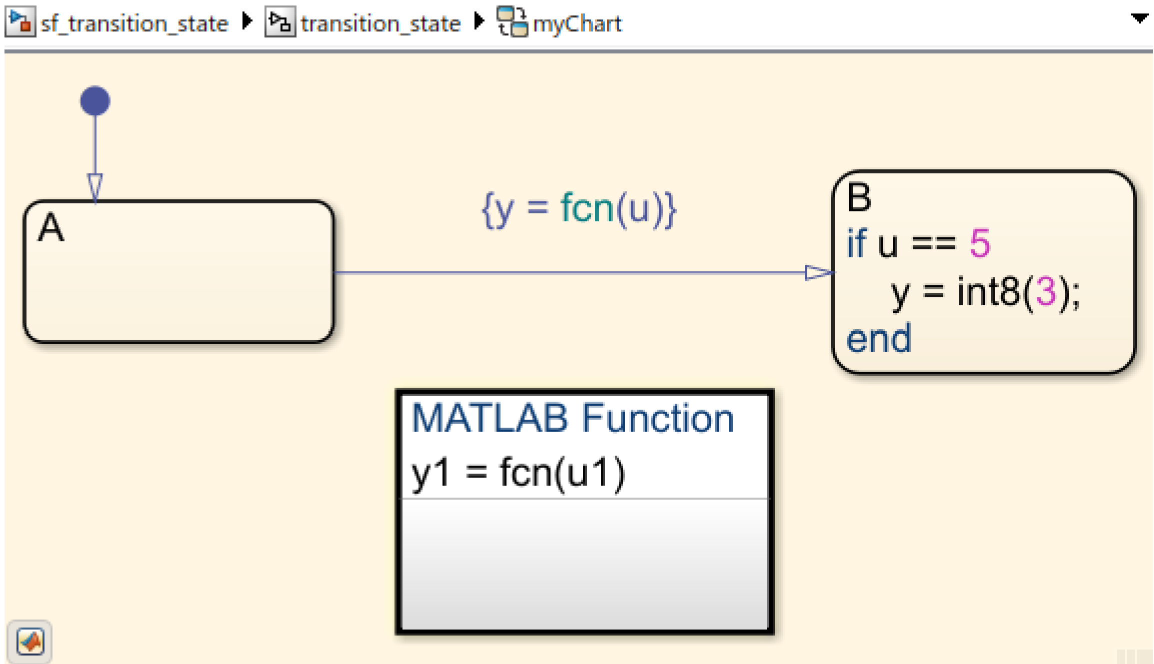
In some cases, MATLAB code written in Stateflow chart can generate HDL code that violates this check.

Explicit Chart Patterns

Cascaded `if` regions containing assignments to the same variable in a Stateflow chart can produce the equivalent pattern in the generated HDL code.



One `if` region in a transition followed by another `if` region in a destination state can cause a violation in the generated HDL code. This pattern is seen in this Stateflow Chart:



With the MATLAB Function block in the transition containing the code:

```
function y1 = fcn(u1)
    if(u1<4)
        y1 = int8(5);
```

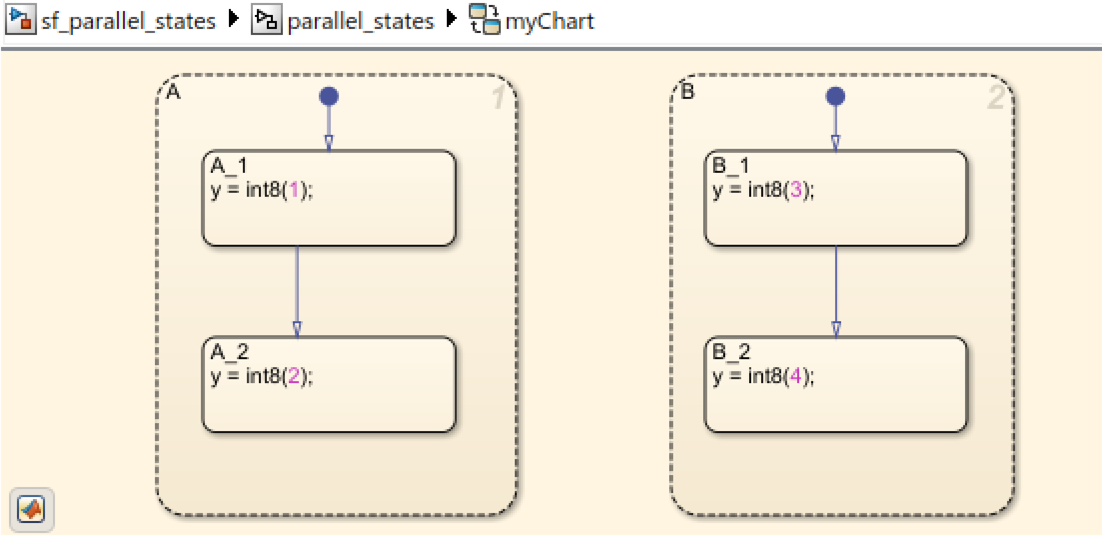
```
    else
        y1 = int8(6);
    end
end
```

The workaround for both of these explicit chart pattern violations is to change the design pattern used in creating the Stateflow charts. Avoid using explicit `if/else` or `switch/case` regions in your Stateflow charts. Instead, redesign the required control flows by using Stateflow chart semantics, such as transitions and states.

Parallel Decomposition

In Stateflow charts, you have an option to choose **Parallel (AND)** mode for the **Decomposition** property of the chart that changes the chart execution to parallel style. For more information, see "Execution Order for Parallel States" (Stateflow).

For example, when you specify parallel decomposition in a chart, it results in the generation of cascaded `switch` regions in the same `always` process.



If the same output variable is assigned a value in multiple parallel states, a violation occurs in the generated HDL code. This Verilog code snippet demonstrates the violation of a value assignment to the variable *y_1* in two parallel conditional `case` statements in the same `always` process, generated as a result of parallel decomposition.

```
always @(is_A, is_B) begin
        is_A_next = is_A;
        is_B_next = is_B;
        case ( is_A)
            state_type_is_A_IN_A_1 :
            begin
                is_A_next = state_type_is_A_IN_A_2;
                y_1 = 8'sd2;
            end
            default :
            begin
                //case IN_A_2:
                y_1 = 8'sd2;
```

```
            end
        endcase
        case ( is_B)
            state_type_is_B_IN_B_1 :
            begin
                is_B_next = state_type_is_B_IN_B_2;
                y_1 = 8'sd4;
            end
            default :
            begin
                //case IN_B_2:
                y_1 = 8'sd4;
            end
        endcase
    end
```

Temporal Logic

Use of temporal logic in your Stateflow chart generally results in violations. Under some conditions, generated code does not violate the coding guideline. Avoid using temporal logic to generate code that does not result in a violation.

LUT-Based Sine, Cosine Block

When using the LUT-based Sine, Cosine block in your model to generate HDL code, a violation of this check might occur in the generated HDL code. This Verilog code snippet generated from a Simulink model containing a LUT-based Sine, Cosine block demonstrates the violation of a value assignment to the variable *Look_Up_Table_k* in multiple parallel conditional `if` statements.

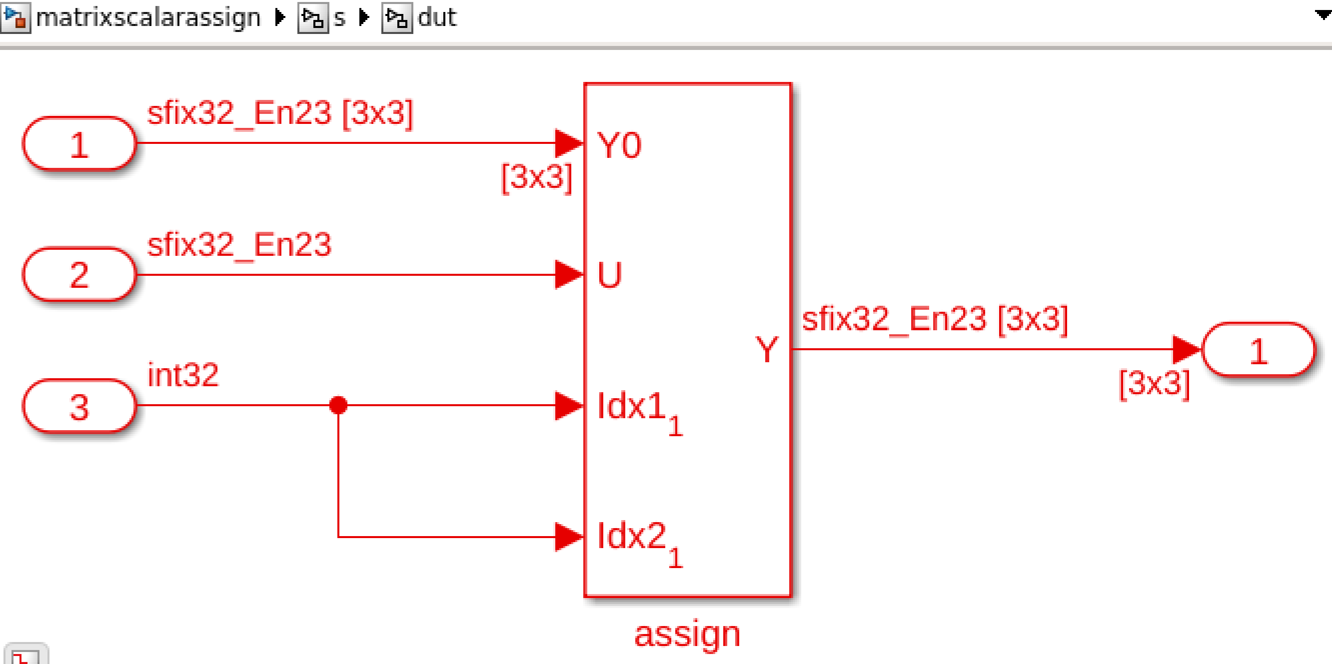
```
always @(QuadHandle2_out1) begin

    Look_Up_Table_t_0_0[0] = 16'sb0000000000000000;
    ...
    Look_Up_Table_t_0_0[31] = 16'sb0011111111101100;
    Look_Up_Table_t_0_0[32] = 16'sb0100000000000000;
    Look_Up_Table_in0_0 = 18'sb000000000000000000;
    Look_Up_Table_in0_0_0 = 18'sb000000000000000000;

    if (QuadHandle2_out1 <= 18'sb000000000000000000) begin
        Look_Up_Table_k = 6'b000000;
    end
    else if (QuadHandle2_out1 >= 18'sb000100000000000000) begin
        Look_Up_Table_k = 6'b100000;
    end
    else begin
        Look_Up_Table_in0_0 = QuadHandle2_out1 >>> 8'd9;
        Look_Up_Table_k = Look_Up_Table_in0_0[5:0];
    end

    ...

    Look_Up_Table_dout_low = Look_Up_Table_t_0_0[Look_Up_Table_k];
    if ( ! (Look_Up_Table_k == 6'b100000)) begin
        Look_Up_Table_k = Look_Up_Table_k + 6'b000001;
    end
    ...
end
```

For more information, see Sine, Cosine.

Assignment Block

Similar to the violation caused by conditional assignments of matrix variables in MATLAB Function blocks, the Assignment block might cause a similar violation in the generated HDL code.

For example, a model containing an Assignment block in the DUT with the Assignment block property **Index Option** having more than one dimension and set to `Index vector (port)`:



produces this Verilog code snippet:

```
always @* begin
    if ((In3 == 32'sb00000000000000000000000000000001) && (In3 == 32'sb000000000000000000000000000
    begin
        assign_out1[0][0] = In2;
    end
    else begin
        assign_out1[0][0] = In1[0][0];
    end
    if ((In3 == 32'sb00000000000000000000000000000010) && (In3 == 32'sb000000000000000000000000000
    begin
        assign_out1[1][0] = In2;
    end
    else begin
        assign_out1[1][0] = In1[1][0];
    end

    ...

    if ((In3 == 32'sb00000000000000000000000000000011) && (In3 == 32'sb000000000000000000000000000
    begin
        assign_out1[2][2] = In2;
```

```
    end
    else begin
        assign_out1[2][2] = In1[2][2];
    end
end
```

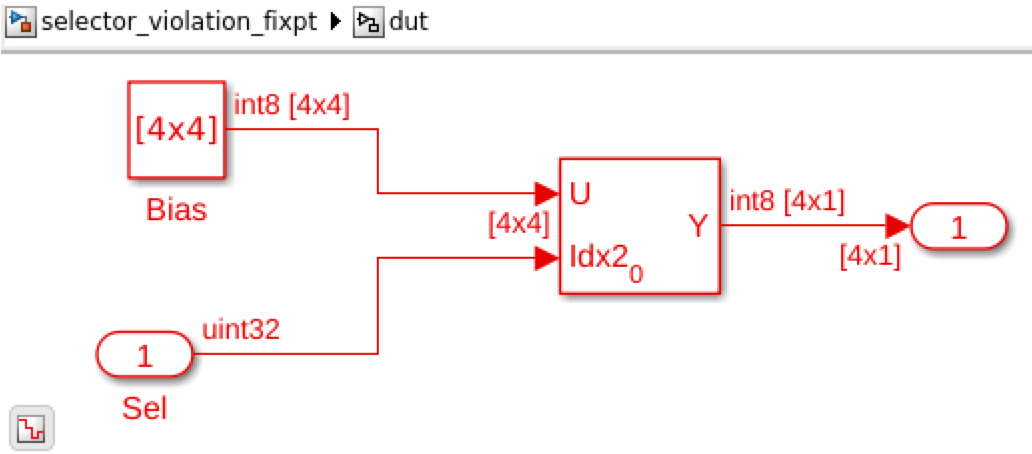
This Verilog code violates the check because even though the index values assigned of the matrix variable *assign_out1* are different between the two cascaded `if` statements, the matrix variable is still being assigned multiple values in cascading conditional regions.

For more information on the Assignment block, see Assignment.

Selector Block

If a Selector block is in a model that generates HDL code, a violation of this check might occur.

For example, a model containing a Selector block in the DUT with the Selector block property **Index Option** having more than one dimension and set to `Index vector (port)`:



produces this generated Verilog code that causes a violation by assigning the same variable *Selector_out1* multiple values for each of its indices in multiple cascaded `if` statements.

```
always @* begin
    Selector_out1[0] = Bias_out1[0][3];
    Selector_out1[1] = Bias_out1[1][3];
    Selector_out1[2] = Bias_out1[2][3];
    Selector_out1[3] = Bias_out1[3][3];
    if (Sel == 32'b00000000000000000000000000000010) begin
        Selector_out1[0] = Bias_out1[0][2];
        Selector_out1[1] = Bias_out1[1][2];
        Selector_out1[2] = Bias_out1[2][2];
        Selector_out1[3] = Bias_out1[3][2];
    end
    if (Sel == 32'b00000000000000000000000000000001) begin
        Selector_out1[0] = Bias_out1[0][1];
        Selector_out1[1] = Bias_out1[1][1];
        Selector_out1[2] = Bias_out1[2][1];
        Selector_out1[3] = Bias_out1[3][1];
    end
    if (Sel == 32'b00000000000000000000000000000000) begin
        Selector_out1[0] = Bias_out1[0][0];
```

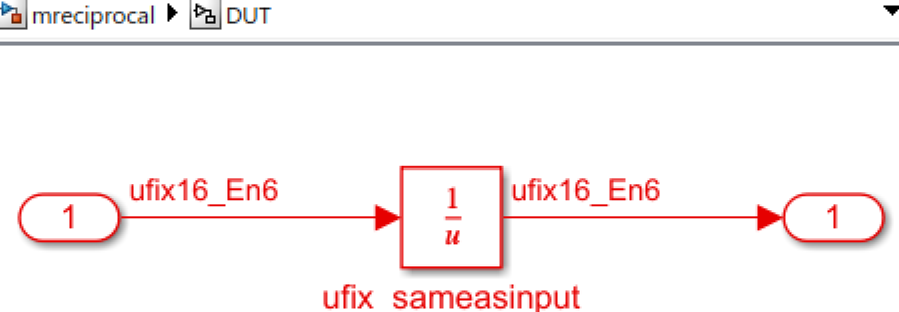
```
        Selector_out1[1] = Bias_out1[1][0];
        Selector_out1[2] = Bias_out1[2][0];
        Selector_out1[3] = Bias_out1[3][0];
    end
end
```

For more information on the Selector block, see Selector.

Math Function Block Set as Reciprocal

If a Math Function block with the **Function** set as `reciprocal`, the **Algorithm method** set as `Exact`, and the **Saturate on integer overflow** selected as `on` is in a DUT in a model that generates HDL code, a violation of this check might occur.

For example, a model containing the Math Function block with the preceding settings



produces this Verilog Code snippet that demonstrates the violation of a value assignment to the variable *ufix_sameasinput_out1* in two parallel conditional `if` statements:

```
always @(In1) begin
    ufix_sameasinput_div_temp = 16'b0000000000000000;
    if (In1 == 16'b0000000000000000) begin
        ufix_sameasinput_out1 = 16'b1111111111111111;
    end
    else begin
        ufix_sameasinput_div_temp = 13'b1000000000000 / In1;
        ufix_sameasinput_out1 = ufix_sameasinput_div_temp;
    end
    if (In1 == 16'b0000000000000000) begin
        ufix_sameasinput_out1 = 16'b1111111111111111;
    end
end
```

For more information on the Math Function block, see Math Function.

See Also

Model Settings
Check for conditional statements in processes | **Check for assignments to the same variable in multiple cascaded control regions** | **Check if-else statement chain length** | **Check if-else statement nesting depth**

More About

- "RTL Description Rules and Checks" on page 24-18
- HDL Coding Standard Customization Properties

Interfacing Subsystems and Models to HDL Code

- “Model Referencing for HDL Code Generation” on page 25-2
- “Generate Black Box Interface for Subsystem” on page 25-4
- “Generate Black Box Interface for Referenced Model” on page 25-8
- “Integrate Custom HDL Code by Using DocBlock” on page 25-10
- “Customize Black Box or HDL Cosimulation Interface” on page 25-12
- “Specify Bidirectional Ports” on page 25-17
- “Generate Reusable Code for Subsystems” on page 25-18
- “Scalarization of Vector Ports in Generated VHDL Code” on page 25-25
- “Create a Xilinx System Generator Subsystem” on page 25-29
- “Create an Altera DSP Builder Subsystem” on page 25-31
- “Using Altera DSP Builder Advanced Blockset with HDL Coder” on page 25-33
- “Using Xilinx System Generator for DSP with HDL Coder” on page 25-38
- “Choose a Test Bench for Generated HDL Code” on page 25-41
- “Generate a Cosimulation Model” on page 25-43
- “HDL Verifier Cosimulation Model Generation in HDL Coder” on page 25-57
- “Verify HDL Design Using SystemVerilog DPI Test Bench” on page 25-68
- “Pass-Through and No-Op Implementations” on page 25-74
- “Synchronous Subsystem Behavior with the State Control Block” on page 25-75
- “Using the State Control Block to Generate More Efficient Code with HDL Coder” on page 25-80
- “Resettable Subsystem Support in HDL Coder” on page 25-87

Model Referencing for HDL Code Generation

Model referencing in your DUT subsystem enables you to:

- Partition a large design into a hierarchy of smaller designs for reuse, modular development, and accelerated simulation.
- Incrementally generate and test code.

  HDL Coder incrementally generates code for referenced models according to the **Configuration Parameters dialog box > Model Referencing pane > Rebuild** options. For more information, see Rebuild.

  However, HDL Coder treats `If changes detected` and `If changes in known dependencies detected` as the same. For example, if you set **Rebuild** to either `If changes detected` or `If changes in known dependencies detected`, HDL Coder regenerates code for referenced models only when the referenced models have changed. The referenced model targets are rebuild when the software detects a change that could affect simulation results. A structural checksum is a computation used to detect changes in the model that can affect simulation results. For more information about the kinds of changes that affect the structural checksum, see `Simulink.BlockDiagram.getChecksum`.

How To Generate Code for a Referenced Model

By default, Generate VHDL or SystemVerilog code for model references into a single library is enabled. The VHDL code is generated in a single library instead of separate libraries. In this case, set the `ScalarizePorts` property to `off` before generating HDL code.

When generating code, if you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, use the `ScalarizePorts` property to generate non-conflicting port definitions. For more information, see Scalarize ports.

Use of vector or matrix ports at the model reference boundary requires ports to be scalarized in generated VHDL code, or the type definition or code to be placed in single library. When generating VHDL code, either set Scalarize ports to `on` or enable Generate VHDL or SystemVerilog code for model references into a single library option.

You can generate HDL code for the referenced model using the UI or the command line.

Using the UI

1. Right-click the Model block and select **HDL Code > HDL Block Properties**.
2. For **Architecture**, select **ModelReference**.
3. Generate HDL code from your DUT subsystem.

Using the Command Line

1. Set the `Architecture` property of the Model block to `ModelReference`. For example, for a DUT subsystem, `mydut`, that includes a model reference, `referenced_model`, enter this command:

```
hdlset_param ('mydut/referenced_model', ...
              'Architecture', 'ModelReference');
```

2 Generate HDL code for your DUT subsystem.

```
makehdl ('mydut');
```

Generate Code for Model Arguments

To generate a single Verilog or SystemVerilog `module` or VHDL `entity` for instances of a referenced model with different model argument values, see “Generate Parameterized Code for Referenced Models” on page 14-21.

Generate Comments

If you enter text in the Model Block Properties dialog box **Description** field, HDL Coder generates a comment in the HDL code.

Limitations

- Model block must have default values for the Block parameters.
- Multiple model references that refer to the same model must have the same HDL block properties.

HDL Coder cannot move registers across a model reference. Therefore, referenced models can inhibit these optimizations:

- Distributed pipelining
- Constrained output pipelining
- Streaming
- Clock-rate pipelining
- Resource sharing

To use these optimizations, consider using subsystem references instead of model references. For more information, see “Create and Use Referenced Subsystems in Models”.

When you have model references and generate HDL code, the generated model, validation model, and cosimulation model can fail to compile or simulate. To fix compilation or simulation errors, make sure that the referenced models are loaded or are on the search path.

The coder can apply the resource sharing optimization to share referenced model instances. However, you can apply this optimization only when all model references that point to the same referenced model have the same rate after optimizations and rate propagation. The model reference final rate may differ from the original rate, but all model references that point to the same referenced model must have the same final rate.

Generate Black Box Interface for Subsystem

In this section...

"What Is a Black Box Interface?" on page 25-4
"Requirements" on page 25-4
"Generate a Black Box Interface for a Subsystem" on page 25-4
"Generate Code for a Black Box Subsystem Implementation" on page 25-6

What Is a Black Box Interface?

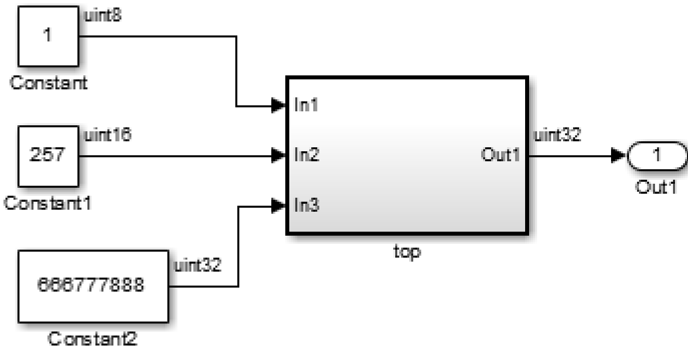
A *black box* interface for a subsystem is a generated VHDL component or Verilog or SystemVerilog module that includes only the HDL input and output port definitions for the subsystem. By generating such a component, you can use a subsystem in your model to generate an interface to existing manually written HDL code, third-party IP, or other code generated by HDL Coder.

Requirements

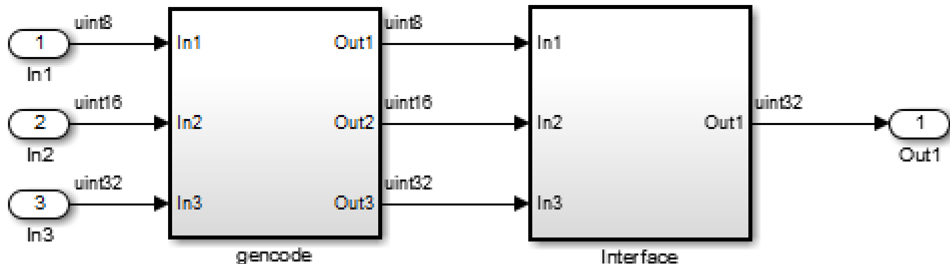
- The black box implementation is available only for subsystem blocks below the level of the DUT. Virtual and atomic subsystem blocks of custom libraries that are below the level of the DUT also work with black box implementations.

Generate a Black Box Interface for a Subsystem

To generate the interface, select the `BlackBox` implementation for one or more Subsystem blocks. Consider the following model that contains a subsystem `top`, which is the device under test.



The subsystem `top` contains two lower-level subsystems:

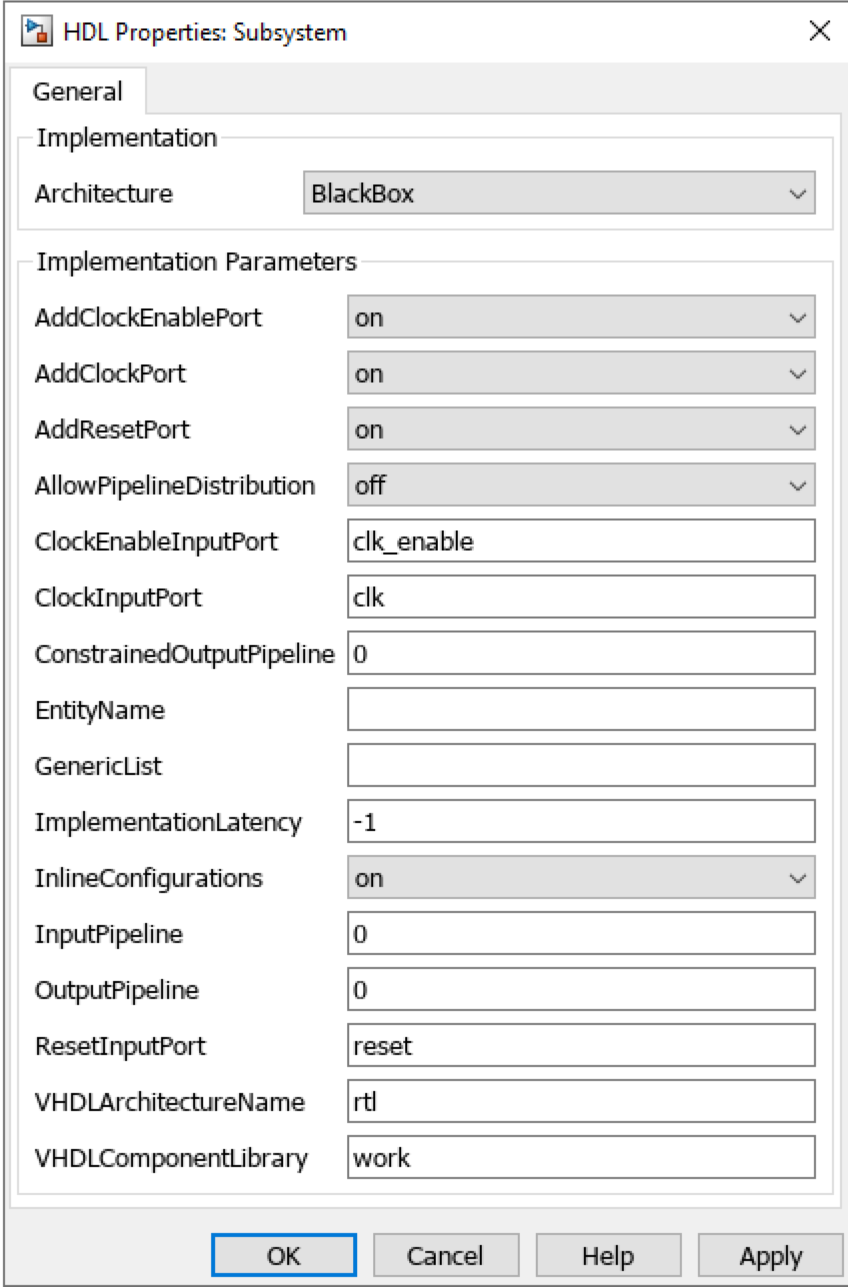


Suppose that you want to generate HDL code from `top`, with a black box interface from the `Interface` subsystem. To specify a black box interface:

1. Right-click the `Interface` subsystem and select **HDL Code** > **HDL Block Properties**.

   The HDL Properties dialog box appears.
2. Set **Architecture** to `BlackBox`.

   The following parameters are available for the black box implementation:



The HDL block parameters available for the black box implementation enable you to customize the generated interface. See "Customize Black Box or HDL Cosimulation Interface" on page 25-12 for information about these parameters.

3. Change parameters as desired, and click **Apply**.
4. Click **OK** to close the HDL Properties dialog box.

Generate Code for a Black Box Subsystem Implementation

When you generate code for the DUT in the `ex_blackbox_subsys` model, the following messages appear:

```
>> makehdl('ex_blackbox_subsys/top')
### Generating HDL for 'ex_blackbox_subsys/top'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### Begin VHDL Code Generation
### Working on ex_blackbox_subsys/top/gencode as hdlsrc\gencode.vhd
### Working on ex_blackbox_subsys/top as hdlsrc\top.vhd
### HDL Code Generation Complete.
```

In the progress messages, observe that the `gencode` subsystem generates a separate file, `gencode.vhd`, for its VHDL entity definition. The `Interface` subsystem does not generate such a file. The interface code for this subsystem is in `top.vhd`, generated from `ex_blackbox_subsys/top`. The following code listing shows the component definition and instantiation generated for the `Interface` subsystem.

```
  COMPONENT Interface
    PORT( clk           :   IN    std_logic;
          clk_enable    :   IN    std_logic;
          reset         :   IN    std_logic;
          In1           :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          In2           :   IN    std_logic_vector(15 DOWNTO 0);  -- uint16
          In3           :   IN    std_logic_vector(31 DOWNTO 0);  -- uint32
          Out1          :   OUT   std_logic_vector(31 DOWNTO 0)  -- uint32
          );
  END COMPONENT;
...
  u_Interface : Interface
    PORT MAP( clk => clk,
              clk_enable => enb,
              reset => reset,
              In1 => gencode_out1,  -- uint8
              In2 => gencode_out2,  -- uint16
              In3 => gencode_out3,  -- uint32
              Out1 => Interface_out1  -- uint32
              );

  enb <= clk_enable;

  ce_out <= enb;

  Out1 <= Interface_out1;
```

By default, the black box interface generated for subsystems includes clock, clock enable, and reset ports. “Customize Black Box or HDL Cosimulation Interface” on page 25-12 describes how you can rename or suppress generation of these signals, and customize other aspects of the generated interface.

See Also

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 25-12
- “Generate Black Box Interface for Referenced Model” on page 25-8
- “Integrate Custom HDL Code by Using DocBlock” on page 25-10

Generate Black Box Interface for Referenced Model

In this section...

"When to Generate a Black Box Interface" on page 25-8
"How to Generate a Black Box Interface" on page 25-8
"Caveats and Limitations" on page 25-8

When to Generate a Black Box Interface

Specify a black box implementation for the Model block when you already have legacy or manually-written HDL code. HDL Coder generates the HDL code that is required to interface to the referenced HDL code.

Code is generated with the following assumptions:

- Every HDL entity or module requires clock, clock enable, and reset ports. Therefore, these ports are defined for each generated entity or module.
- Use of Simulink data types is assumed. For VHDL code, port data types are assumed to be `STD_LOGIC` or `STD_LOGIC_VECTOR`.

If you want to generate code for a multirate, multiclock DUT that includes a referenced model, see "Model Referencing for HDL Code Generation" on page 25-2.

How to Generate a Black Box Interface

To instantiate an HDL wrapper, or black box interface, for a referenced model:

1. Right-click the Model block and select **HDL Code** > **HDL Block Properties**.

   In the HDL Block Properties dialog box:

   - For **Architecture**, select **BlackBox**.
   - Customize the ports and other implementation parameters. To learn more about customizing the ports, see "Customize Black Box or HDL Cosimulation Interface" on page 25-12.
2. Generate HDL code for your DUT subsystem.

Caveats and Limitations

- If you run the `checkhdl` function to check the compatibility of your model for HDL code generation, the function does not check the port data types within the referenced model.
- If you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, use the `ScalarizePorts` property to generate nonconflicting port definitions. For more information, see Scalarize ports.

See Also

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 25-12
- “Generate Black Box Interface for Subsystem” on page 25-4
- “Integrate Custom HDL Code by Using DocBlock” on page 25-10

Integrate Custom HDL Code by Using DocBlock

In this section...

"When to Use DocBlock for Integrating Custom Code" on page 25-10

"Use DocBlock to Integrate Custom Code" on page 25-10

"Restrictions" on page 25-10

"Include Custom HDL Code Using Doc Block" on page 25-11

You can use one or more DocBlock blocks to integrate custom HDL code into your design.

When to Use DocBlock for Integrating Custom Code

If you want to keep the HDL code in your model, instead of as a separate file, use a DocBlock to integrate custom HDL code. The text in the DocBlock is your custom VHDL, Verilog or SystemVerilog code.

You include each DocBlock that contains custom HDL code by placing it in a black box subsystem, and then including the black box subsystem in your design under test (DUT). One HDL file is generated for each black box subsystem.

Alternatives for Custom Code Integration

If you want to keep your custom HDL code separate from your model, such as when the custom code is intellectual property (IP) or a library from a third party vendor, use a black box subsystem on page 25-4 or black box model reference on page 25-8.

Use DocBlock to Integrate Custom Code

1. In your DUT, at any level of hierarchy, add a Subsystem block.
2. For the Subsystem block, in the HDL Block Properties dialog box:
   - Set **Architecture** to `BlackBox`.
   - Customize the black box subsystem interface so that it matches your custom HDL code interface. See "Customize Black Box or HDL Cosimulation Interface" on page 25-12.
3. In the subsystem, add a DocBlock block.
4. For the DocBlock, in the HDL Block Properties dialog box:
   - Set **Architecture** to `HDLText`.
   - Set **TargetLanguage** to your target language, such as `Verilog`, `SystemVerilog` or `VHDL`.
5. In the DocBlock, enter the HDL code for your custom Verilog or SystemVerilog `module`, or VHDL `entity`.

   The language must match the DocBlock **TargetLanguage** setting.

Restrictions

- The black box subsystem that contains the DocBlock cannot be the top-level DUT.

- You can have a maximum of two DocBlock blocks in the black box subsystem. If you have two DocBlock blocks, one has **TargetLanguage** set to `VHDL` and the other must have **TargetLanguage** set to either `Verilog` or `SystemVerilog`.

  When generating code, HDL Coder integrates the code from the DocBlock that matches the target language for code generation.
- When a black box subsystem that contains the DocBlock is placed in another black box subsystem, HDL Coder does not integrate the HDL code from this DocBlock during code generation.

Include Custom HDL Code Using Doc Block

The `hdlcoderIncludeCustomHdlUsingDocBlockExample` model shows how to integrate custom VHDL and Verilog code into your design with the DocBlock block.

To open `hdlcoderIncludeCustomHdlUsingDocBlockExample` model, run these commands:

```
load_system('hdlcoderIncludeCustomHdlUsingDocBlockExample.slx');
open_system('hdlcoderIncludeCustomHdlUsingDocBlockExample/DUT');
```

The `LegacyCodeSubs` subsystem has two DocBlock blocks which contains legacy VHDL and Verilog code. You can generate the HDL code of the `DUT` subsystem by using `makehdl` command.

See Also

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 25-12
- “Generate Black Box Interface for Subsystem” on page 25-4
- “Generate Black Box Interface for Referenced Model” on page 25-8

Customize Black Box or HDL Cosimulation Interface

You can customize port names and set attributes of the external component when you generate an interface from the following blocks:

- Model with black box implementation
- Subsystem with black box implementation
- HDL Cosimulation

Interface Parameters

Open the HDL Block Properties dialog box to see the interface generation parameters.

The following table summarizes the names, value settings, and purpose of the interface generation parameters.

Note You cannot specify clock, reset, and clock enable signals explicitly in your Simulink model by using the **AddClockEnablePort**, **AddClockPort**, and **AddResetPort** parameters. Instead, use these parameters to add a clock, reset, or clock enable port in the generated HDL code.

Parameter Name	Values	Description
AddClockEnablePort	`on` | `off` Default: `on`	If `on`, add a clock enable input port to the interface generated for the block. The name of the port is specified by **ClockEnableInputPort**.
AddClockPort	`on` | `off` Default: `on`	If `on`, add a clock input port to the interface generated for the block. The name of the port is specified by **ClockInputPort**. If **AddClockPort** is `off`, then **AddClockEnablePort** and **AddResetPort** is considered as `off`.
AddResetPort	`on` | `off` Default: `on`	If `on`, add a reset input port to the interface generated for the block. The name of the port is specified by **ResetInputPort**.
AllowPipelineDistribution	`on` | `off` Default: `off`	If `on`, allow HDL Coder to move registers across the block, from input to output or output to input.

Parameter Name	Values	Description
ClockEnableInputPort	Default: `clk_enable`	Specifies HDL name for block's clock enable input port. To generate the clock enable input ports with the HDL names specified in this property, set **AddClockEnablePort** to `on`. You can specify multiple clock enable names. The mapping of clock enables depends on input and output sample rates. The clock enable ports are mapped from fastest to slowest sample rate. For example, if your model has two sample rates and you specify `["clken1" "clken2"]` as the clock enable names to this property. The `clken1` is mapped to the faster sample rate and `clken2` is mapped to the slower rate. When **Clock inputs** is `Single`, specify either one or *N* clock enable port names. Where *N* is number of sample rates. When **Clock inputs** is `Multiple`, specify *N* clock enable port names.
ClockInputPort	Default: `clk`	Specifies HDL name for block's clock input signal. To generate the clock input ports with the HDL name specified in this property, set **AddClockPort** to `on`. You can specify multiple clock port names. The mapping of clock ports depends on input and output sample rates. The clock ports are mapped from fastest to slowest sample rate. For example, if your model has two sample rates and you specify `["clk1" "clk2"]` as the clock names to this property. The `clk1` is mapped to the faster sample rate and `clk2` is mapped to the slower rate. When **Clock inputs** is `Single`, specify one clock input port name. When **Clock inputs** is `Multiple`, specify *N* clock input port names. Where *N* is number of sample rates.

Parameter Name	Values	Description
ConstrainedOutputPipeline	Default: 0	Specifies the number of delays that you want the code generator to insert at the output of the interface by redistributing existing delays in your design.
EntityName	Default: Entity name string is derived from the block name, and modified when necessary to generate a legal VHDL entity name.	Specifies VHDL `entity`, or Verilog or SystemVerilog `module` name generated for the block.

GenericList	Pass a cell array variable that contains cell arrays each with two or three character arrays, or enter a cell array of cell arrays that each contain two or three character arrays. The character arrays represent the name, value, and optional data type of a VHDL `generic`, or Verilog or SystemVerilog `parameter`. The default data type is `integer`.
	
	Default: none
		Specifies a list of VHDL `generic`, or Verilog or SystemVerilog `parameter` name-value pairs, each with an optional data type specification, to pass to a subsystem with a `BlackBox` implementation.
		
		For example, in the HDL Block Properties dialog box, enter `{'`*`name`*`','`*`value`*`','`*`type`*`'}`, or, if the data type is `integer`, enter `{'`*`name`*`','`*`value`*`'}`.
		
		To set `GenericList` using `hdlset_param`, at the command line, enter:
		
		`hdlset_param (`*`blockname`*`,'GenericList','{''`*`name`*`'',''`*`value`*`'',''`*`type`*`''}');`
		
		If the data type is `integer`, at the command line, enter:
		
		`hdlset_param (`*`blockname`*`,'GenericList','{''`*`name`*`'',''`*`value`*`''}');`

Parameter Name	Values	Description
ImplementationLatency	-1 | 0 | positive integer Default: -1	Specifies the additional latency of the external component in time steps, relative to the Simulink block. If 0 or greater, this value is used for delay balancing. Your inputs and outputs must operate at the same rate. If -1, latency is unknown. This disables delay balancing. For Black Box subsystem that has multiple sample rates, the product of **ImplementationLatency** and slowest output sample rate must be divisible by all the output sample rates.
InlineConfigurations (VHDL only)	`on` | `off` Default: If this parameter is unspecified, defaults to the value of the global `InlineConfigurations` property.	If `off`, suppress generation of a configuration for the block, and require a user-supplied external configuration.
InputPipeline	Default: 0	Specifies the number of input pipeline stages (pipeline depth) in the generated code.
OutputPipeline	Default: 0	Specifies the number of output pipeline stages (pipeline depth) in the generated code.

Parameter Name	Values	Description
ResetInputPort	Default: `reset`	Specifies HDL name for block's reset input. To generate the reset input ports with the HDL name specified in this property, set **AddResetPort** to `on`. You can specify multiple reset port names. The mapping of reset ports depends on input and output sample rates. The reset ports are mapped from fastest to slowest sample rate. For example, if your model has two sample rates and you specify `["rst1" "rst2"]` as the reset names to this property. The `rst1` is mapped to the faster sample rate and `rst2` is mapped to the slower rate. When **Clock inputs** is `Single`, specify one reset input port name. When **Clock inputs** is `Multiple`, specify *N* reset input port names. Where *N* is number of sample rates.
VHDLArchitectureName (VHDL only)	Default: `rtl`	Specifies RTL architecture name generated for the block. The architecture name is generated only if **InlineConfigurations** is `on`.
VHDLComponentLibrary (VHDL only)	Default: `work`	Specifies the library from which to load the VHDL component.

See Also

More About

- "Generate Black Box Interface for Subsystem" on page 25-4
- "Generate Black Box Interface for Referenced Model" on page 25-8
- "Integrate Custom HDL Code by Using DocBlock" on page 25-10
- "Specify Bidirectional Ports" on page 25-17

Specify Bidirectional Ports

You can specify bidirectional ports for Subsystem blocks with black box implementation. In the generated code, the bidirectional ports have the Verilog, SystemVerilog or VHDL `inout` keyword.

Requirements

- The bidirectional port must be a black box subsystem port.
- There must be no logic between the bidirectional port and the corresponding top-level DUT subsystem port. Otherwise, the generated code does not compile.

How To Specify a Bidirectional Port

To specify a bidirectional port using the UI:

1. In the black box Subsystem, right-click the Inport or Outport block that represents the bidirectional port. Select **HDL Code > HDL Block Properties**.
2. For **BidirectionalPort**, select `on`.

To specify a bidirectional port at the command line, set the `BidirectionalPort` property to `'on'` using `hdlset_param` or `makehdl`.

For example, suppose you have a model, *`my_model`*, that contains a DUT subsystem, *`dut_subsys`*, and the DUT subsystem contains a black box subsystem, *`blackbox_subsys`*. If *`blackbox_subsys`* has an Inport, *`input_A`*, specify *`input_A`* as bidirectional by entering:

```
hdlset_param('mymodel/dut_subsys/blackbox_subsys/input_A','BidirectionalPort','on');
```

Limitations

- You cannot generate a Verilog or SystemVerilog test bench if there is a bidirectional port within your DUT subsystem.
- HDL Coder does not support bidirectional ports for masked subsystems that use `BlackBox` as the **HDL Architecture**.
- Simulink does not support bidirectional ports, so you cannot simulate the bidirectional behavior in Simulink.

See Also

More About

- “Generate Black Box Interface for Subsystem” on page 25-4
- “Generate Black Box Interface for Referenced Model” on page 25-8
- “Integrate Custom HDL Code by Using DocBlock” on page 25-10
- “Customize Black Box or HDL Cosimulation Interface” on page 25-12

Generate Reusable Code for Subsystems

In this section...

"Requirements for Generating Reusable Code for Atomic Subsystems" on page 25-18
"Requirements for Generating Reusable Code for Virtual Subsystems" on page 25-18
"Generate Reusable Code for Atomic Subsystems" on page 25-19
"Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters" on page 25-21

HDL Coder can detect atomic subsystems that are identical, or identical except for their mask parameter values, at any level of the model hierarchy, and generate a single reusable HDL `module` or `entity`. The reusable HDL code is generated as a single file and instantiated multiple times.

Requirements for Generating Reusable Code for Atomic Subsystems

- The `DefaultParameterBehavior` Simulink Configuration Parameter must be `Inlined`. You can set this parameter at the command line by using the `set_param` or `hdlsetup` function. To specify this setting in the Configuration Parameters dialog box, you must have Simulink Coder.

  Note Using `hdlsetup` sets `InlineParams` property to `on`. Enabling this parameter is the same as setting `DefaultParameterBehavior` to `Inlined`. Setting `InlineParams` to `off` changes `DefaultParameterBehavior` value to `Tunable`.

- Do not use functionality such as signal logging or using blocks like To Workspace or To File.
- The atomic subsystems must be identical, or identical except for their mask parameter values.

  - `MaskParameterAsGeneric` must be `on` if the reusable subsystem has mask parameters with different values. This parameter has no effect on virtual subsystems. For more information, see Generate parameterized HDL code from masked subsystem.
  - Mask parameters must be tunable. The code generator does not share atomic subsystems with mask parameters that are nontunable.
  - Mask parameters must be scalar.
  - Mask parameter data types must be `integer` or `fixed-point` with a word length of less than or equal to `32`.
  - The tunable parameter must be used in only Constant, Gain, or Compare To Constant blocks.
  - Port data types must match.

    If you change the value of the tunable mask parameter, the output port data type can change. If one of the atomic subsystems has a different port data type, the code generated for that subsystem also differs.

Requirements for Generating Reusable Code for Virtual Subsystems

- The `DefaultParameterBehavior` Simulink Configuration Parameter must be `Inlined`. You can set this parameter at the command line by using the `set_param` or `hdlsetup` function. To specify this setting in the Configuration Parameters dialog box, you must have Simulink Coder.

Note Using `hdlsetup` sets `InlineParams` property to `on`. Enabling this parameter is the same as setting `DefaultParameterBehavior` to `Inlined`. Setting `InlineParams` to `off` changes `DefaultParameterBehavior` value to `Tunable`.

- Do not use logging functionality such as signal logging or blocks like To Workspace or To File.
- The virtual subsystems must be identical.
  - `SubsystemReuse` must be set to `'Atomic and Virtual'`.
    - Setting `SubsystemReuse` to `'Atomic and Virtual'` reduces artificial algebraic errors and improves the recognition of identical subsystems, irrespective of their topology within the rest of the design. Identification of similar subsystems can help resource sharing.
    - To set these values to your required setting, in the MATLAB Command Window, enter:

      ```
      hdlset_param('myHDLModel', 'SubsystemReuse', 'Atomic and Virtual')
      ```
    - Alternatively, you can set this option from the top-level **HDL Code Generation** pane in the Configuration Parameters dialog box. Under **Global Settings** > **Coding style**, you can change the **Code reuse** setting to the required option.
    - The previous commands set the `SubsystemReuse` option for your project. To set this option for only the current code generation session, enter:

      ```
      makehdl(<DUT system>, 'SubsystemReuse', 'Atomic and Virtual')
      ```
  - The tunable parameter must be used in only Constant, Gain, or Compare To Constant blocks.
  - Port data types must match.
  - If the value of the tunable parameter changes during runtime, the code generator does not reuse the code.

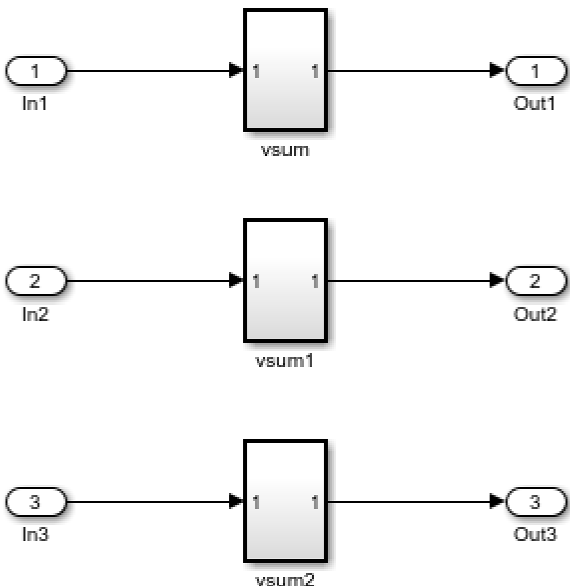
Generate Reusable Code for Atomic Subsystems

This example shows HDL code generation of a model that has identical atomic subsystems. If your design contains identical atomic subsystems, the coder generates one HDL module or entity for the subsystem and instantiates it multiple times.

Open Model

Open the `hdlcoder_reusable_code_identical_subsystem` model to see an example of a DUT subsystem containing three identical atomic subsystems.

```
load_system('hdlcoder_reusable_code_identical_subsystem');
open_system('hdlcoder_reusable_code_identical_subsystem/DUT');
```



Generate HDL Code

You can generate the HDL code for the model by using `makehdl` function. To generate HDL code for `DUT` subsystem, run this command:

```
makehdl('hdlcoder_reusable_code_identical_subsystem/DUT')
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_reusable_code_identical_subsystem
### Generating HDL for <a href="matlab:open_system('hdlcoder_reusable_code_identical_subsystem/DU
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_reusab
### Running HDL checks on the model 'hdlcoder_reusable_code_identical_subsystem'.
### Begin compilation of the model 'hdlcoder_reusable_code_identical_subsystem'...
### Working on the model 'hdlcoder_reusable_code_identical_subsystem'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_reusable_code_identical_subsy
### Begin model generation 'gm_hdlcoder_reusable_code_identical_subsystem'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_reusable_code_i
### Generating new validation model: '<a href="matlab:open_system('hdl_prj\hdlsrc\hdlcoder_reusab
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_reusable_code_identical_subsystem'.
### Working on hdlcoder_reusable_code_identical_subsystem/DUT/vsum as hdl_prj\hdlsrc\hdlcoder_reu
### Working on hdlcoder_reusable_code_identical_subsystem/DUT as hdl_prj\hdlsrc\hdlcoder_reusable
### Generating package file hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\DUT_pkg.vhd
### Code Generation for 'hdlcoder_reusable_code_identical_subsystem' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlcoder_reusable_code_identical_subsystem' complete with 0 errors, 0 warnings
### HDL code generation complete.
```

HDL Coder generates a single VHDL® file, `vsum.vhd`, for the three subsystems. The generated code for the DUT subsystem, `DUT.vhd`, contains three instantiations of the `vsum` component.

```
ARCHITECTURE rtl OF DUT IS
```

```
-- Component Declarations
COMPONENT vsum
  PORT( In1                               :   IN    vector_of_std_logic_vector16(0 TO 9);  -- int16
        Out1                              :   OUT   std_logic_vector(19 DOWNTO 0)  -- sfix20
        );
END COMPONENT;

-- Component Configuration Statements
FOR ALL : vsum
  USE ENTITY work.vsum(rtl);

-- Signals
SIGNAL vsum_out1                          : std_logic_vector(19 DOWNTO 0);  -- ufix20
SIGNAL vsum1_out1                         : std_logic_vector(19 DOWNTO 0);  -- ufix20
SIGNAL vsum2_out1                         : std_logic_vector(19 DOWNTO 0);  -- ufix20

BEGIN
u_vsum : vsum
  PORT MAP( In1 => In1,  -- int16 [10]
            Out1 => vsum_out1  -- sfix20
            );

u_vsum1 : vsum
  PORT MAP( In1 => In2,  -- int16 [10]
            Out1 => vsum1_out1  -- sfix20
            );

u_vsum2 : vsum
  PORT MAP( In1 => In3,  -- int16 [10]
            Out1 => vsum2_out1  -- sfix20
            );

Out1 <= vsum_out1;

Out2 <= vsum1_out1;

Out3 <= vsum2_out1;

END rtl;
```

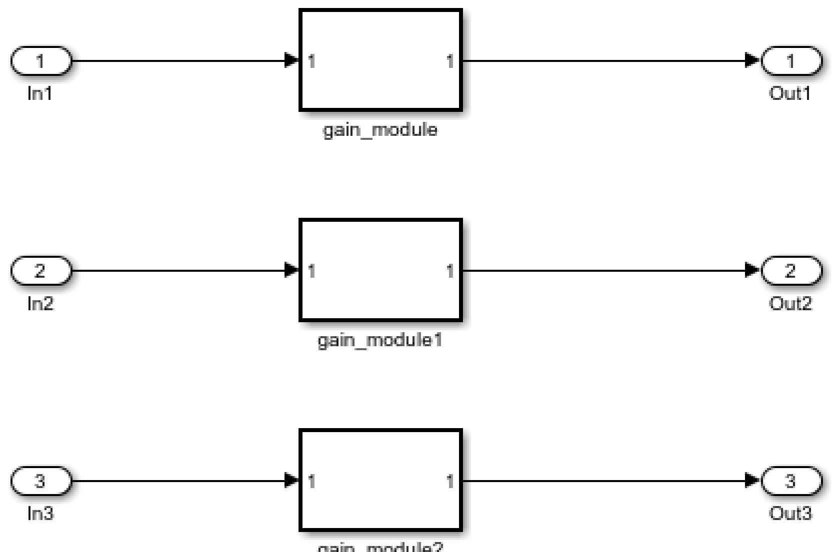
Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters

Using this example, you can generate HDL code for your design containing atomic subsystems that are identical except for their tunable mask parameter values. You can generate one HDL module or entity for the subsystem. In the generated code, the module or entity is instantiated multiple times.

Open Model

The `hdlcoder_reusable_code_parameterized_subsystem` model shows an example of a DUT subsystem containing atomic subsystems that are identical except for their tunable mask parameter values.

```
load_system('hdlcoder_reusable_code_parameterized_subsystem');
open_system('hdlcoder_reusable_code_parameterized_subsystem/DUT');
```



In `hdlcoder_reusable_code_parameterized_subsystem/DUT`, the gain modules are subsystems with gain values represented by tunable mask parameters. Gain values are: 4 for `gain_module`, 5 for `gain_module1`, and 7 for `gain_module2`.

Generate HDL Code

To generate reusable code for identical atomic subsystems, enable `MaskParameterAsGeneric` for the model. By default, `MaskParameterAsGeneric` is disabled.

To enable the generation of reusable code for the atomic subsystems with tunable parameters, set `MaskParameterAsGeneric` option to "on" for a model by using `hdlset_param` function.

```
hdlset_param('hdlcoder_reusable_code_parameterized_subsystem','MaskParameterAsGeneric','on');
```

Alternatively, in the Configuration Parameters dialog box, in the **HDL Code Generation > Global Settings > Coding Style** tab, enable the **Generate parameterized HDL code from masked subsystem** option.

You can also enable `MaskParameterAsGeneric` option by using `makehdl` function and generate HDL code for a DUT subsystem.

```
makehdl('hdlcoder_reusable_code_parameterized_subsystem/DUT','MaskParameterAsGeneric','on',...
        'TargetLanguage','Verilog')
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_reusable_code_parameterized_subsys
### Generating HDL for <a href="matlab:open_system('hdlcoder_reusable_code_parameterized_subsyste
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_reusab
### Running HDL checks on the model 'hdlcoder_reusable_code_parameterized_subsystem'.
### Begin compilation of the model 'hdlcoder_reusable_code_parameterized_subsystem'...
### Working on the model 'hdlcoder_reusable_code_parameterized_subsystem'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_reusable_code_parameterized_s
### Begin model generation 'gm_hdlcoder_reusable_code_parameterized_subsystem'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_reusable_code_paramete
```

```
### Begin Verilog Code Generation for 'hdlcoder_reusable_code_parameterized_subsystem'.
### Working on hdlcoder_reusable_code_parameterized_subsystem/DUT/gain_module as hdlsrc\hdlcoder_
### Working on hdlcoder_reusable_code_parameterized_subsystem/DUT as hdlsrc\hdlcoder_reusable_coc
### Code Generation for 'hdlcoder_reusable_code_parameterized_subsystem' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdlcoder_reusable_code_parameterized_subsystem' complete with 0 errors, 0 warn
### HDL code generation complete.
```

With `MaskParameterAsGeneric` enabled, HDL Coder generates a single source file, `gain_module.v`, for the three gain module subsystems. The generated code for the DUT subsystem, `DUT.v`, contains three instantiations of the `gain_module` component.

```
module DUT
        (
         In1,
         In2,
         In3,
         Out1,
         Out2,
         Out3
        );

input   [7:0] In1;  // uint8
input   [7:0] In2;  // uint8
input   [7:0] In3;  // uint8
output  [31:0] Out1;  // uint32
output  [31:0] Out2;  // uint32
output  [31:0] Out3;  // uint32

wire [31:0] gain_module_out1;  // uint32
wire [31:0] gain_module1_out1;  // uint32
wire [31:0] gain_module2_out1;  // uint32

gain_module   #  (.myGain(4)
                   )
              u_gain_module   (.In1(In1),  // uint8
                               .Out1(gain_module_out1)  // uint32
                               );

assign Out1 = gain_module_out1;

gain_module   #  (.myGain(5)
                   )
              u_gain_module1   (.In1(In2),  // uint8
                                .Out1(gain_module1_out1)  // uint32
                                );

assign Out2 = gain_module1_out1;

gain_module   #  (.myGain(7)
                   )
              u_gain_module2   (.In1(In3),  // uint8
                                .Out1(gain_module2_out1)  // uint32
                                );

assign Out3 = gain_module2_out1;

endmodule  // DUT
```

In gain_module.v, the myGain Verilog® parameter is generated for the tunable mask parameter.

```
module gain_module
        (
         In1,
         Out1
        );

input   [7:0] In1;  // uint8
output  [31:0] Out1;  // uint32

parameter [31:0] myGain = 4;  // ufix32

wire [31:0] kconst;  // ufix32
wire [39:0] Gain_mul_temp;  // ufix40
wire [31:0] Gain_out1;  // uint32

assign kconst = myGain;

assign Gain_mul_temp = kconst * In1;
assign Gain_out1 = Gain_mul_temp[31:0];

assign Out1 = Gain_out1;

endmodule  // gain_module
```

If the input model has subsystems that contains mask information such as mask variables and mask initialization code, the mask information of those subsystems is preserved in the generated model. If you do not need this information, you can remove it.

See Also

More About

- Generate parameterized HDL code from masked subsystem
- “Generate Parameterized Code for Referenced Models” on page 14-21
- “Create and Add Tunable Parameter That Maps to DUT Ports” on page 14-18

Scalarization of Vector Ports in Generated VHDL Code

This example shows how to flatten the vector signals in a Simulink® model into a structure of scalar signals in the generated VHDL® code.

Specify Scalarization of Vector Ports

When you generate Verilog code, by default, the vector signals are flattened into scalars by default. When you generate VHDL code, you can specify whether to flatten the vector signals on the entire model or at the DUT level. Flattening the vector ports at only the DUT level speeds up code generation, especially for large models that have many vector inputs.

When generating HDL code from MATLAB® and Simulink, you can scalarize the vector ports into scalars. For the MATLAB® to HDL workflow, in the **HDL Code Generation** task, on the **Clocks & Ports** tab, set **Scalarize ports** to `dutlevel` or `on`.

To scalarize the vector ports when generating HDL code for a Simulink model:

- In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Ports** tab, set **Scalarize Ports** parameter to `dutlevel` or `on`.
- In the HDL Workflow Advisor, on the **HDL Code Generation > Set Code Generation Options > Set Advanced Options > Ports** tab, set Scalarize ports parameter to `dutlevel` or `on`.
- At the MATLAB command prompt, set the `ScalarizePorts` property to `on` or `dutlevel` by using `hdlset_param` or `makehdl`.

Vector Sum Model

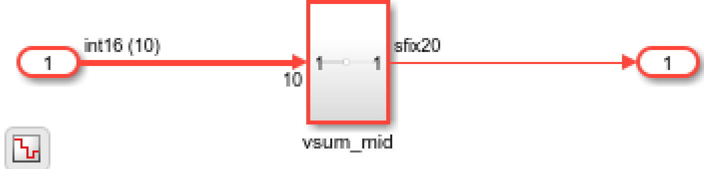
To see the flattening of vector ports, open the model `hdlcoder_hdlcoder_vector_sum_nested`. The model is driven by a vector input of width `10` and has a scalar output.

```
open_system('hdlcoder_vector_sum_nested')
set_param('hdlcoder_vector_sum_nested','SimulationCommand','update')
```

Copyright 2020-2021 The MathWorks, Inc.

This model is the same as the `simplevectorsum` model and consists of a Subsystem, `vsum_mid`, inside the `vsum` subsystem.

```
open_system('hdlcoder_vector_sum_nested/vsum')
```



The `vsum_mid` subsystem contains a Sum of Elements block that is configured for vector summation. The model is configured to use the `Tree` implementation when generating HDL code for the Sum of Elements block within the `vsum` subsystem. This implementation is optimized for minimal latency, generates a tree-shaped structure of adders for the block.

```
open_system('hdlcoder_vector_sum_nested/vsum/vsum_mid')
```

Scalarize Vector Ports

By default, the `ScalarizePorts` property is `off`. HDL Coder™ generates a type definition and port declaration for the vector port `In1`, as shown in this code.

```
PACKAGE simplevectorsum_pkg IS
  TYPE vector_of_std_logic_vector16 IS ARRAY (NATURAL RANGE <>)
     OF std_logic_vector(15 DOWNTO 0);
  TYPE vector_of_signed16 IS ARRAY (NATURAL RANGE <>) OF signed(15 DOWNTO 0);
END simplevectorsum_pkg;

...

ENTITY vsum IS
  PORT( In1     :   IN    vector_of_std_logic_vector16(0 TO 9);  -- int16 [10]
        Out1    :   OUT   std_logic_vector(19 DOWNTO 0)  -- sfix20
      );
END vsum;
```

To scalarize the vector ports when generating HDL code, either set the **Scalarize ports** parameter in the Configuration Parameters dialog box or set the `ScalarizePorts` property to `on` or `dutlevel` by using the `hdlset_param` or `makehdl` functions. When you set `ScalarizePorts` to `dutlevel`, only the vector signals at the DUT are flattened into scalars. The scalars are input to the `vsim_mid` subsystem as vectors.

```
makehdl('hdlcoder_vector_sum_nested/vsum', 'ScalarizePorts', 'dutlevel')
```

```
ENTITY vsum IS
  PORT( In1_0                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_1                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_2                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_3                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_4                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_5                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_6                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_7                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_8                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_9                   :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
```

```
        Out1                 :   OUT   std_logic_vector(19 DOWNTO 0)  -- sfix20
        );
END vsum;

ENTITY vsum_mid IS
  PORT( In1                             :   IN    vector_of_std_logic_vector16(0 TO 9);  -- int
        Out1                            :   OUT   std_logic_vector(19 DOWNTO 0)  -- sfix20
        );
END vsum_mid;
```

You can change the starting index for the names of scalarized vector ports from `Zero-based` to `One-based` by setting the model configuration parameter Indexing for scalarized port naming to `One-based`.

To flatten the vector ports on the entire model, set `ScalarizePorts` to `on`. The vector ports at `vsum_mid` and the inputs to the Sum of Elements block are also flattened into scalars.

```
makehdl('hdlcoder_vector_sum_nested/vsum', 'ScalarizePorts', 'on')
```

```
ENTITY vsum_mid IS
  PORT( In1_0                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_1                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_2                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_3                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_4                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_5                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_6                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_7                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_8                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_9                           :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1                            :   OUT   std_logic_vector(19 DOWNTO 0)  -- sfix20
        );
END vsum_mid;
```

Usage Notes and Restrictions

- When you use the `ScalarizePorts` property for a protected model, you must use the same value for this property as the value specified for this parameter in the top model from which it is referenced.
- When you use the `IP Core Generation` and `FPGA-in-the-Loop` workflows, vector ports are not supported. Set `ScalarizePorts` as `on` or `dutlevel`. For faster code generation, set `ScalarizePorts` to `dutlevel`.
- Vector or matrix ports as input to a model reference interface must be scalarized before generating HDL code. Set `ScalarizePorts` to `dutlevel`.
- By default, **Generate VHDL code for model references into a single library** is enabled. The VHDL code is generated in a single library instead of separate libraries. In this case, set the `ScalarizePorts` property to `off` before generating HDL code.
- When generating code, if you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, use the `ScalarizePorts` property to generate non-conflicting port definitions.

See Also

`makehdl`

More About

- "Model Referencing for HDL Code Generation" on page 25-2
- "Generate Black Box Interface for Referenced Model" on page 25-8

Create a Xilinx System Generator Subsystem

In this section...

"Why Use Xilinx System Generator Subsystems" on page 25-29

"Requirements for Xilinx System Generator Subsystems" on page 25-29

"Create a Xilinx System Generator Subsystem" on page 25-29

"Limitations for Code Generation from Xilinx System Generator Subsystems" on page 25-30

Why Use Xilinx System Generator Subsystems

You can generate HDL code from a model with both Simulink and Xilinx blocks using Xilinx System Generator (XSG) subsystems. Using Simulink and Xilinx blocks in your model provides:

- A single platform for combined Simulink and Xilinx System Generator simulation, code generation, and synthesis.
- Targeted code generation: Xilinx System Generator for DSP generates code from Xilinx blocks and HDL Coder generates code from Simulink blocks.
- HDL Coder area and speed optimizations for Simulink components.

Requirements for Xilinx System Generator Subsystems

Group your Xilinx blocks into one or more Xilinx System Generator (XSG) subsystems for code generation. An XSG subsystem can contain a hierarchy of subsystems.

To generate code from a Xilinx System Generator subsystem:

- Use Vivado or ISE Design Suite 13.4 or later.
- If your design uses Boolean data types, select the **Use STD_LOGIC type for Boolean or 1 bit wide gateways** setting in the Xilinx System Generator window. By default, Xilinx System Generator uses `std_logic_vector` to represent Boolean types whereas HDL Coder uses `std_logic`, which can result in a mismatch.

An XSG subsystem is a Subsystem block that has:

- Architecture set to **Module**.
- One System Generator token, placed at the top level of the XSG subsystem hierarchy.
- Xilinx blocks.
- Simulink blocks not requiring code generation.
- Input and output ports connected directly to Gateway In or Gateway Out blocks.
- **Propagate data type to output** option enabled on Gateway Out blocks.
- Matching input and output data types on Gateway In blocks. See "Limitations for Code Generation from Xilinx System Generator Subsystems" on page 25-30.

Create a Xilinx System Generator Subsystem

1. Create a subsystem containing the Xilinx blocks and set its architecture to `Module`.

2. Add a System Generator token at the top level of the subsystem.

   You can have subsystem hierarchy in a Xilinx System Generator subsystem, but there must be a System Generator token at the top level of the hierarchy.
3. Connect each subsystem input or output port directly to a Gateway In or Gateway Out block.
4. On each Gateway Out block, select the **Propagate data type to output** option.

Limitations for Code Generation from Xilinx System Generator Subsystems

Code generation from Xilinx System Generator (XSG) subsystems has these limitations:

- `ConstrainedOutputPipeline`, `InputPipeline`, and `OutputPipeline` are the only valid block properties for an XSG subsystem.
- HDL Coder does not generate code for blocks within an XSG subsystem, including Simulink blocks.
- Gateway In blocks must not do nontrivial data type conversion. For example, a Gateway In block can convert between the `sfix8_en6` and `Fix_8_6` data types, but changing data sign, word length, or fraction length is not allowed.
- For Verilog code generation, Simulink block names in your design cannot be the same as Xilinx names. Similarly, Xilinx blocks in your design cannot have the same name as other Xilinx blocks. HDL Coder cannot resolve these name conflicts and generates an error late in the code generation process.

See Also
`makehdl` | `makehdltb`

Related Examples
- “Using Xilinx System Generator for DSP with HDL Coder” on page 25-38

Create an Altera DSP Builder Subsystem

In this section...

"Why Use Altera DSP Builder Subsystems?" on page 25-31
"Requirements for Altera DSP Builder Subsystems" on page 25-31
"How to Create an Altera DSP Builder Subsystem" on page 25-31
"Determine Clocking Requirements for Altera DSP Builder Subsystems" on page 25-32
"Limitations for Code Generation from Altera DSP Builder Subsystems" on page 25-32

Why Use Altera DSP Builder Subsystems?

You can generate HDL code from a model with both Simulink and Altera DSP Builder Advanced blocks using Altera DSP Builder (DSPB) subsystems.

Using both Simulink and Altera blocks in your model provides the following benefits:

- A single platform for combined Simulink and Altera DSP Builder simulation, code generation, and synthesis.
- Targeted code generation: Altera DSP Builder generates code from Altera blocks; HDL Coder generates code from Simulink blocks.
- HDL Coder area and speed optimizations for Simulink components.

Requirements for Altera DSP Builder Subsystems

You must group your Altera blocks into one or more Altera DSP Builder (DSPB) subsystems for code generation. A DSPB subsystem can contain a hierarchy of subsystems.

To generate code from a Altera DSP Builder subsystem, you must use Quartus II 13.0 or later.

A DSPB subsystem is a Subsystem block with:

- Architecture set to **Module**.
- A valid DSP Builder Advanced Blockset design, including a top-level Device block and DSP Builder Advanced blocks, as defined in the Altera DSP Builder documentation.

How to Create an Altera DSP Builder Subsystem

1. Create an Altera DSP Builder Advanced Blockset design as defined in the Altera DSP Builder documentation.
2. Create a subsystem containing the Altera DSP Builder Advanced Blockset design, and set its **Architecture** to `Module`.

To see an example that shows HDL code generation for an Altera DSP Builder subsystem, see "Using Altera DSP Builder Advanced Blockset with HDL Coder" on page 25-33.

Determine Clocking Requirements for Altera DSP Builder Subsystems

DSPB subsystems must either run at the DUT subsystem base rate, or you can provide a custom clock.

Determining the DUT subsystem base rate can be an iterative process. Area optimizations, such as RAM mapping or resource sharing, may cause HDL Coder to oversample area-optimized parts of the design. Therefore, the DUT subsystem initial base rate may differ from the final base rate, and you may not know the model base rate until you generate code.

To determine the model base rate, iteratively generate code until your model converges on a base rate:

1. Generate code for the DUT subsystem that contains your DSPB subsystem.
2. If HDL Coder displays an error message that says that your DSPB subsystem rate is slower than the base rate, modify the DSPB subsystem inputs so that the DSPB subsystem runs at the base rate in the message.

   For example, you can insert an Upsample block.
3. Repeat these steps until your DSPB subsystem rate matches the base rate.

To provide a custom clock for your DSPB subsystem:

1. In the HDL Workflow Advisor, for **HDL Code Generation** > **Set Code Generation Options** > **Set Advanced Options** > **Clock inputs**, select **Multiple**.
2. In the generated HDL code, connect your custom clocks to the DUT clock input ports that corresponds to your DSPB subsystems clock.

Limitations for Code Generation from Altera DSP Builder Subsystems

Code generation for Altera DSP Builder (DSPB) subsystems has the following limitations:

- The DUT subsystem cannot be a DSPB subsystem.
- DSPB subsystems must run at the Simulink model base rate. You may need to iteratively generate code to determine the base rate, because area optimizations can cause local multirate. See “Determine Clocking Requirements for Altera DSP Builder Subsystems” on page 25-32 for a workflow.
- Altera blocks with bus interfaces are not supported.
- Altera DSP Builder does not generate Verilog code.
- Test bench simulation mismatches can occur because the Simulink data comparison does not take Altera valid signals into account. For an example and workaround, see “Using Altera DSP Builder Advanced Blockset with HDL Coder” on page 25-33.

Using Altera DSP Builder Advanced Blockset with HDL Coder

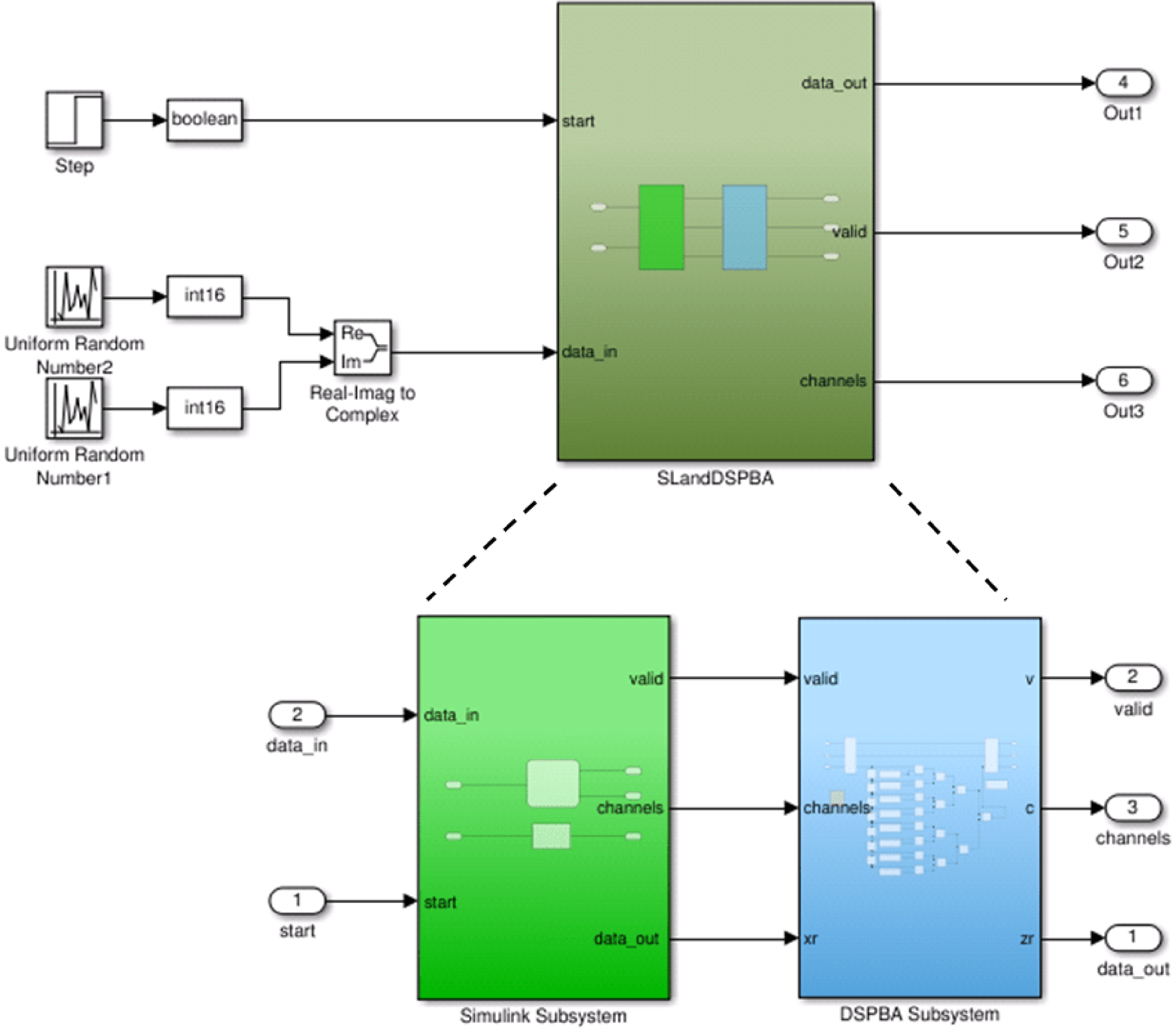
This example shows how to use the Altera® DSP Builder Advanced Blockset with HDL Coder™.

Introduction

Using the Altera® DSP Builder Advanced Blockset Subsystem block, or DSPBA Subsystem block, enables you to model designs using blocks from both Simulink® and Altera®, and to automatically generate integrated HDL code. HDL Coder™ generates HDL code from the Simulink® blocks, and uses Altera® DSP Builder to generate HDL code from the DSPBA Subsystem blocks.

In this example, the design, or code generation subsystem, contains two parts: one with Simulink® native blocks, and one with Altera® DSP Builder Advanced blocks. The Altera® blocks are grouped in a DSPBA Subsystem (hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem). Altera® DSP Builder optimizes these blocks for Altera® FPGAs. In the rest of the design, Simulink® blocks and HDL Coder™ offer many model-based design features, such as distributed pipelining and delay balancing, to perform model-level optimizations.

```
open_system('hdlcoder_sldspba');
open_system('hdlcoder_sldspba/SLandDSPBA');
```



Setup for Altera® DSP Builder Advanced Blockset

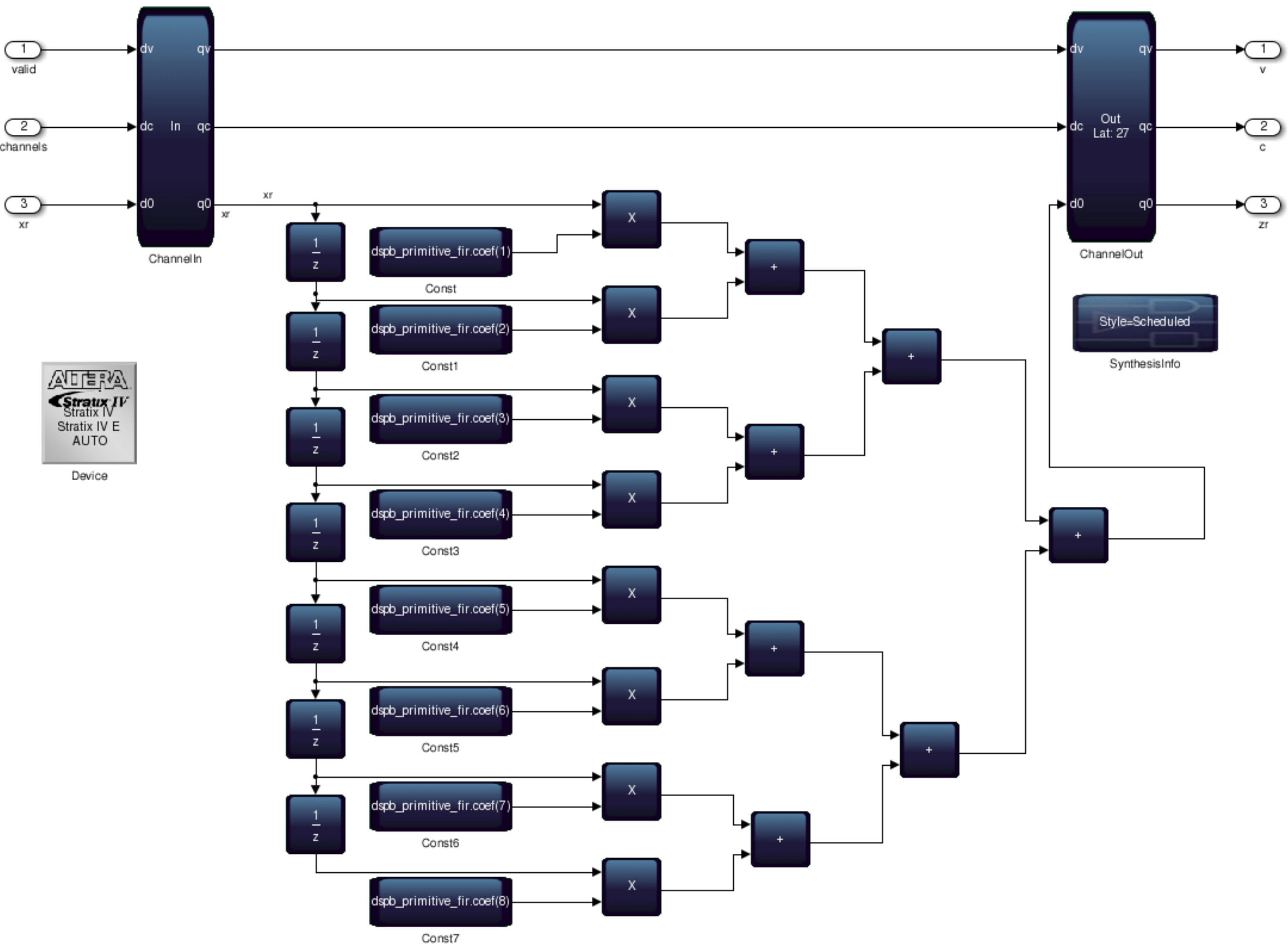
In order to use the Altera® DSP Builder Advanced Blockset Subsystem block, you must have Altera® Quartus II set up with Simulink®. For version compatibility, see "HDL Language Support and Supported Third-Party Tools and Hardware".

Create Altera® DSP Builder Advanced Blockset Subsystem

To create a DSPBA Subsystem:

1. Put the Altera® blocks in one subsystem and set its architecture to "Module" (the default value).
2. Place a Device block at the top level of the subsystem. You can have subsystem hierarchy in a DSPBA Subsystem, but there must be a Device block at the top level of the hierarchy.

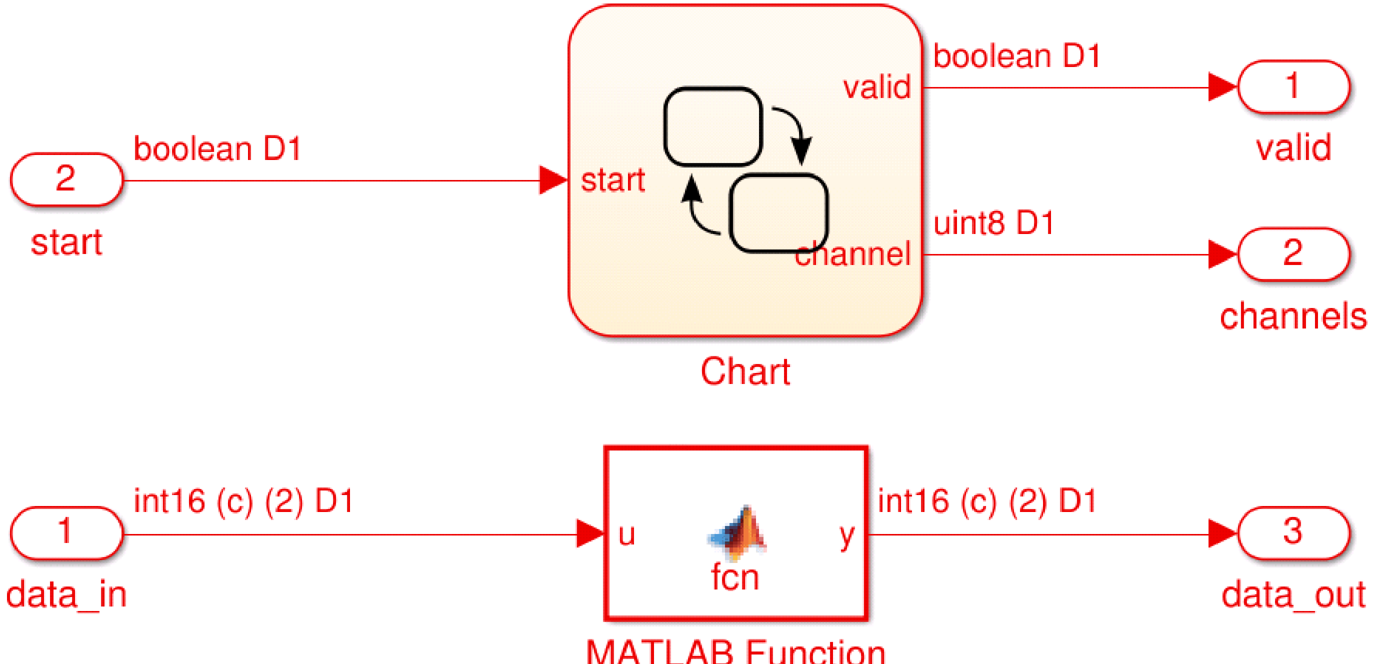
```
open_system('hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem');
```



Simulink® Components

In this example, a Stateflow chart generates the channel and valid signals to drive the DSPBA subsystem.

```
open_system('hdlcoder_sldspba/SLandDSPBA/Simulink Subsystem');
```



Generate HDL Code

You can use either makehdl at the command line or HDL Workflow Advisor to generate HDL code. To use makehdl:

```
makehdl('hdlcoder_sldspba/SLandDSPBA');
```

You can also generate a testbench, simulate, and synthesize the design as you would for any other model.

Handle Simulation Mismatch When Valid Signal Not Asserted

The DSPBA Subsystem simulation may not match its generated code's behavior when the valid signal is not asserted under certain circumstances, such as when the folding option in both hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem/ChannelIn and hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem/ChannelOut are turned on. The mismatch affects the downstream Simulink design and causes a test bench simulation failure.

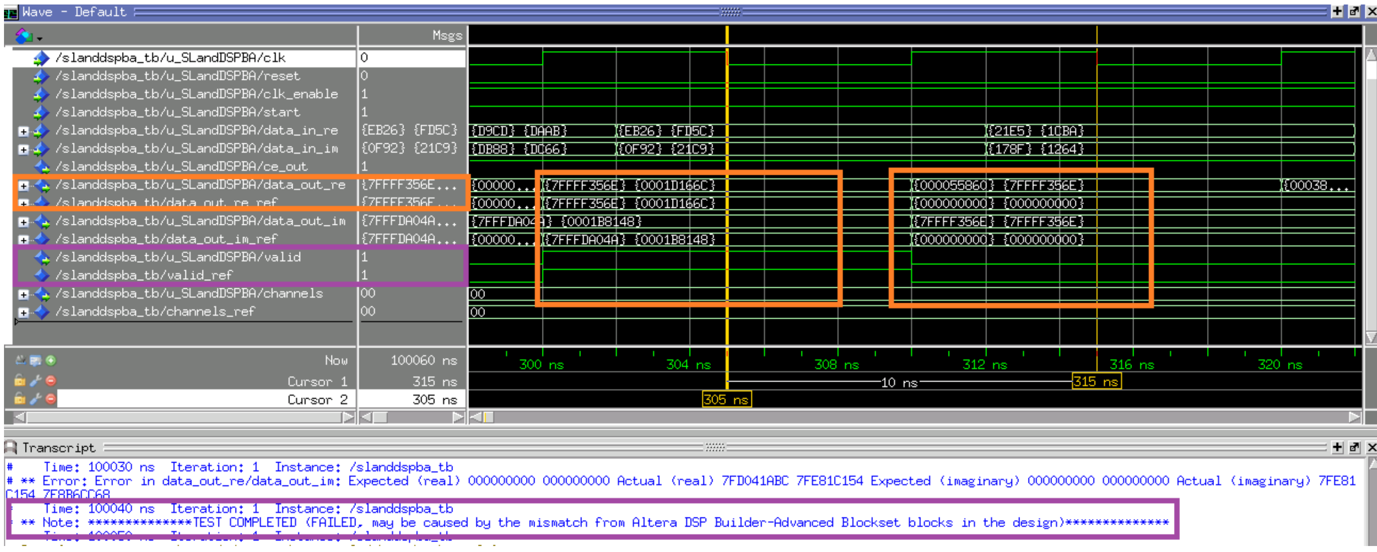
To see the mismatch, you can turn the folding setting on the ChannelIn and ChannelOut blocks:

```
set_param('hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem/ChannelIn', 'FoldingEnabled', 1);
set_param('hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem/ChannelOut', 'FoldingEnabled', 1);
```

Then, generate the HDL code and test bench again:

```
makehdl('hdlcoder_sldspba/SLandDSPBA');
makehdltb('hdlcoder_sldspba/SLandDSPBA');
```

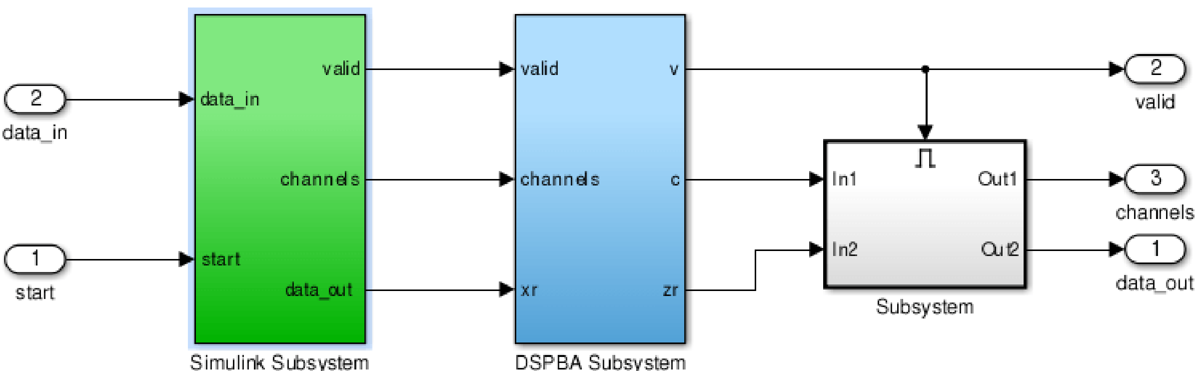
After simulating the generated code and test bench, you can see that the outputs from HDL coder match the reference data only when the valid signal is asserted.



As the message from the test bench indicates, the mismatch is expected.

To avoid this simulation mismatch, insert an enabled subsystem at the DSPBA Subsystem output signals, before they reach the Simulink part of your design or the output ports of the overall design. The following subsystem shows how to connect signals to the enabled subsystem.

```
open_system('hdlcoder_sldspba/SLandDSPBA2');
```



Using Xilinx System Generator for DSP with HDL Coder

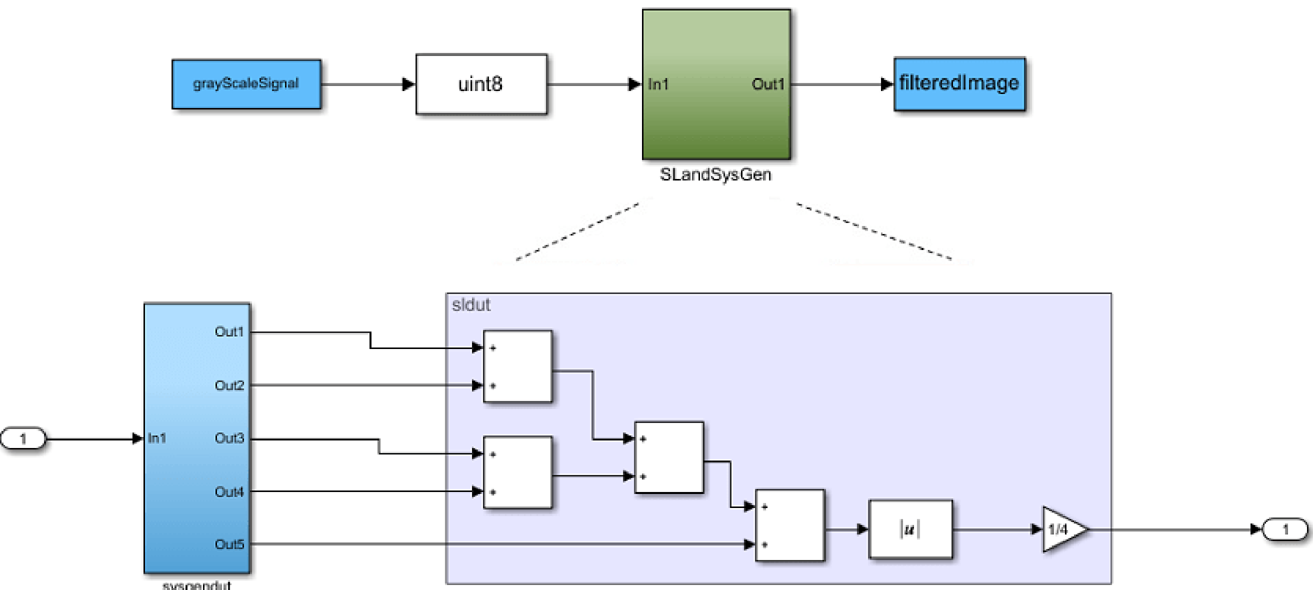
This example shows how to use Xilinx® System Generator for DSP with HDL Coder™.

Introduction

Using the Xilinx System Generator Subsystem block enables you to model designs using blocks from both Simulink® and Xilinx, and to automatically generate integrated HDL code. HDL Coder™ generates HDL code from the Simulink blocks, and uses Xilinx System Generator to generate HDL code from the Xilinx System Generator Subsystem blocks.

In this example, the design, or code generation subsystem, contains two parts: one with Simulink native blocks, and one with Xilinx blocks. The Xilinx blocks are grouped into a Xilinx System Generator Subsystem `sysgendut` that is inside a `SLandSysGen` Subsystem at the top level of the model `hdlcoder_slsysgen`. System Generator optimizes these blocks for Xilinx FPGAs. In the rest of the design, Simulink blocks and HDL Coder offer model-based design capabilities and HDL optimizations, such as distributed pipelining and delay balancing.

```
open_system('hdlcoder_slsysgen');
open_system('hdlcoder_slsysgen/SLandSysGen');
```

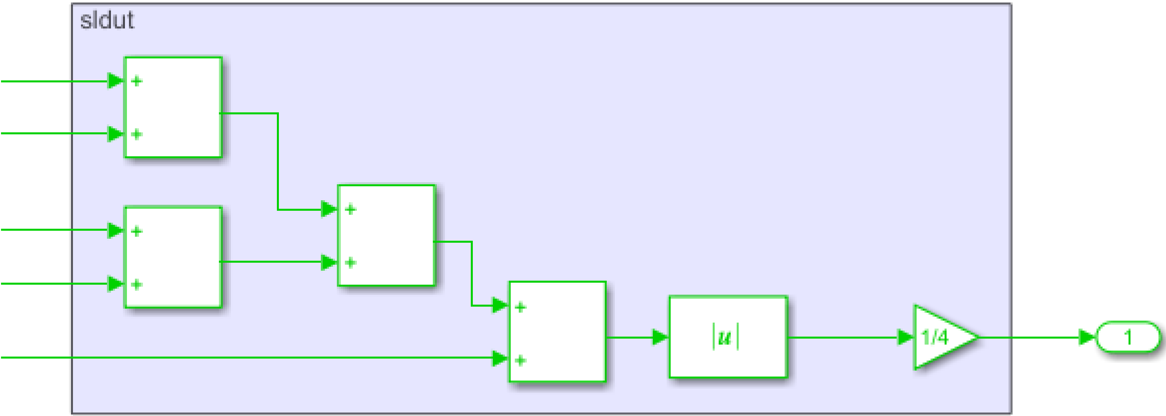


Perform Model-Level Optimizations for Simulink® Components

In this example, a sum tree indicated by the `sldut` section inside the `SLandSysGen` Subsystem is modeled with Simulink blocks. You can use distributed pipelining feature to take care of the speed optimization.

Distributed pipelining can move pipeline registers into the sum tree to reduce the critical path without changing the model function. Other optimizations, such as resource sharing, are also available, but not used in this example.

```
open_system('hdlcoder_slsysgen/SLandSysGen');
```

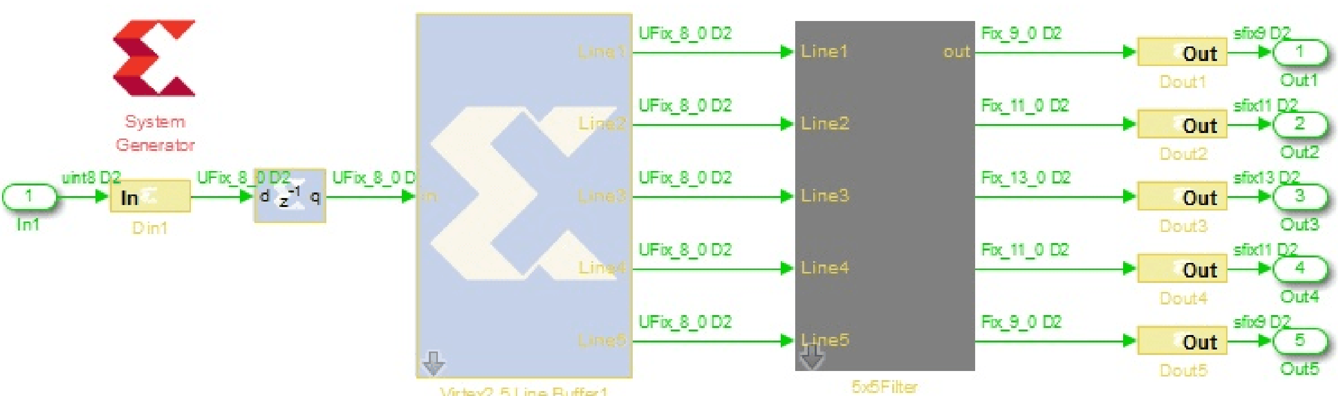


Create Xilinx System Generator Subsystem

To create a Xilinx System Generator subsystem:

1. Put the Xilinx blocks in one subsystem and leave the HDL architecture set to default value of `Module`.
2. Place a System Generator token at the top level of the subsystem. You can have subsystem hierarchy in a Xilinx System Generator Subsystem, but there must be a System Generator token at the top level of the hierarchy.

```
open_system('hdlcoder_slsysgen/SLandSysGen/sysgendut');
```



Configure Gateway In and Gateway Out Blocks

In each Xilinx System Generator subsystem, you must connect input and output ports directly to Gateway In and Gateway Out blocks.

Gateway In blocks must not do non-trivial data type conversion. For example, a Gateway In block can convert between uint8 and UFix_8_0, but changing data sign, word length, or fraction length is not allowed.

Generate HDL Code

You can use either makehdl at the command line or HDL Workflow Advisor to generate HDL code. To use makehdl:

```
makehdl('hdlcoder_slsysgen/SLandSysGen');
```

You can also generate a testbench, simulate, and synthesize the design as you would for any other model.

Choose a Test Bench for Generated HDL Code

When you generate HDL code with HDL Coder, you can optionally generate a test bench as well. The coder also generates build-and-run scripts for the HDL simulator you specify. The test bench options are:

- HDL test bench — An HDL test bench that includes the generated HDL DUT and files containing input and output data vectors. This test bench verifies the generated HDL DUT against test vectors generated from your Simulink model. See “Test Bench Generation” on page 34-15.
- Cosimulation model — A Simulink model that includes an HDL Cosimulation block that runs your generated HDL code in an HDL simulator. The model also includes your original Simulink stimulus generation, your behavioral model, and any blocks for display or analysis of the output data. The model compares the output of the HDL Cosimulation block against the output of the source subsystem. See “Generate a Cosimulation Model” on page 25-43.
- SystemVerilog DPI test bench — An HDL test bench that includes the generated HDL DUT and a generated C-language component. The C component creates input stimuli and runs a behavioral model of the DUT subsystem. The test bench uses a direct programming interface (DPI) to run the C component inside an HDL simulation. This test bench verifies the generated HDL DUT against a C representation of the source Simulink model. See “Verify HDL Design Using SystemVerilog DPI Test Bench” on page 25-68.
- FPGA-in-the-loop — A Simulink model that includes an FPGA-in-the-Loop block that communicates with your HDL design while it runs on the FPGA board. The model also includes your original Simulink stimulus generation, your behavioral model, and any blocks for display or analysis of the output data. The model compares the output of the FPGA-in-the-Loop block against the output of the source subsystem. See “FIL Simulation with HDL Workflow Advisor for Simulink” (HDL Verifier).

Select test bench options in HDL Workflow Advisor under **HDL Code Generation** > **Set Testbench Options**, or in the Model Configuration Parameters dialog box, under **HDL Code Generation** > **Test Bench**. Alternatively, for command-line access, select your test bench using the properties of `makehdltb`.

For FPGA-in-the-loop, select the target workflow in HDL Workflow Advisor under **Set Target** > **Set Target Device and Synthesis Tool**. Then select your FPGA and synthesis tool. You can also generate an FPGA-in-the-loop model for existing HDL code by using FPGA-in-the-Loop Wizard.

Test Bench	License Requirements	Pros	Cons
HDL test bench		• Fast compile time in HDL simulator	• Runs simulation to generate data files, which can take a long time for large data sets • File I/O can slow down simulation for large data sets • Run test in HDL simulator • Fixed input stimulus

Test Bench	License Requirements	Pros	Cons
Cosimulation model	• HDL Verifier	• Fast compile time in HDL simulator • Run tests from Simulink, including changing parameters to affect input stimulus • Automatic test bench execution from HDL Workflow Advisor	
SystemVerilog DPI test bench	• HDL Verifier • Simulink Coder	• Fast generation time because the coder does not run a simulation • Fast simulation time for large data sets, because the stimulus comes from generated code rather than files	• Run test in HDL simulator • No tunable parameters in stimulus generation
FPGA-in-the-loop	• HDL Verifier • HDL Verifier Support Package for Xilinx FPGA Boards or HDL Verifier Support Package for Intel FPGA Boards	• Run tests from Simulink, including changing parameters to affect input stimulus • Prototype hardware implementation of your DUT	• Long generation time due to synthesis into FPGA • Hardware setup

See Also

More About

- “Set HDL Code Generation Options” on page 16-2

Generate a Cosimulation Model

In this section...

“Requirements” on page 25-43
“What Is A Cosimulation Model?” on page 25-43
“Generating a Cosimulation Model using the Model Configuration Parameters” on page 25-44
“Structure of the Generated Model” on page 25-46
“Launching a Cosimulation” on page 25-50
“The Cosimulation Script File” on page 25-52
“Complex and Vector Signals in the Generated Cosimulation Model” on page 25-54
“Generating a Cosimulation Model from the Command Line” on page 25-55
“Naming Conventions for Generated Cosimulation Models and Scripts” on page 25-55
“Limitations for Cosimulation Model Generation” on page 25-56

Requirements

- To use this feature, your installation must include an HDL Verifier license.
- Make sure the DUT subsystem has no unconnected output ports. See “Terminate Unconnected Block Outputs and Usage of Commenting Blocks” on page 18-27.

What Is A Cosimulation Model?

A cosimulation model is an automatically generated Simulink model configured for both Simulink simulation and cosimulation of your design with an HDL simulator. HDL Coder supports automatic generation of a cosimulation model as a part of the test bench generation process.

The cosimulation model includes:

- A behavioral model of your design, realized in a Simulink subsystem.
- A corresponding HDL Cosimulation block, configured to cosimulate the design using HDL Verifier. HDL Coder configures the HDL Cosimulation block for use with either Mentor Graphics ModelSim, Cadence Incisive, or Xilinx Vivado simulator.
- Test input data, calculated from the test bench stimulus that you specify.
- Scope blocks, which let you observe and compare the DUT and HDL cosimulation outputs, and any error between these signals.
- Goto and From blocks that capture the stimulus and response signals from the DUT and use these signals to drive the cosimulation.
- A comparison/assertion mechanism that reports discrepancies between the original DUT output and the cosimulation output.

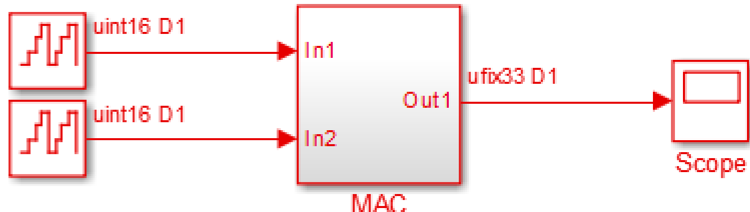
In addition to the generated model, HDL Coder generates a TCL script that launches and configures your cosimulation tool. Comments in the script file document clock, reset, and other timing signal information defined by the coder for the cosimulation tool.

Generating a Cosimulation Model using the Model Configuration Parameters

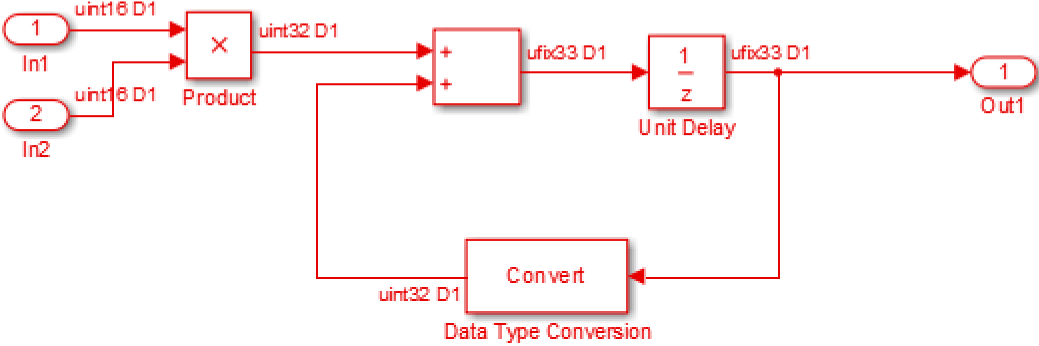
This example demonstrates the process for generating a cosimulation model. The example model, `hdl_cosim_demo1`, implements a simple multiply and accumulate (MAC) algorithm. Open the model by entering the name at the MATLAB command line:

```
openExample('hdl_cosim_demo1');
```

The following figure shows the top-level model.



The DUT is the MAC subsystem.



Cosimulation model generation takes place during generation of the test bench. As a best practice, generate HDL code before generating a test bench, as follows:

1. Open the Configuration Parameters dialog box. Right-click on the white space of the Simulink model and select **Model Configuration Parameters**.
2. In the **HDL Code Generation** pane of the Configuration Parameters dialog box, set **Generate HDL for** to `hdl_cosim_demo1/MAC`.
3. Click **Apply**.
4. Click **Generate**. HDL Coder displays progress messages, as shown in the following listing:

```
### Applying HDL Code Generation Control Statements
### Starting HDL Check.
### HDL Check Complete with 0 error, 0 warning and 0 message.

### Begin VHDL Code Generation
```

```
### Working on hdl_cosim_demo1/MAC as hdlsrc\MAC.vhd
### HDL Code Generation Complete.
```

Next, configure the test bench options to include generation of a cosimulation model:

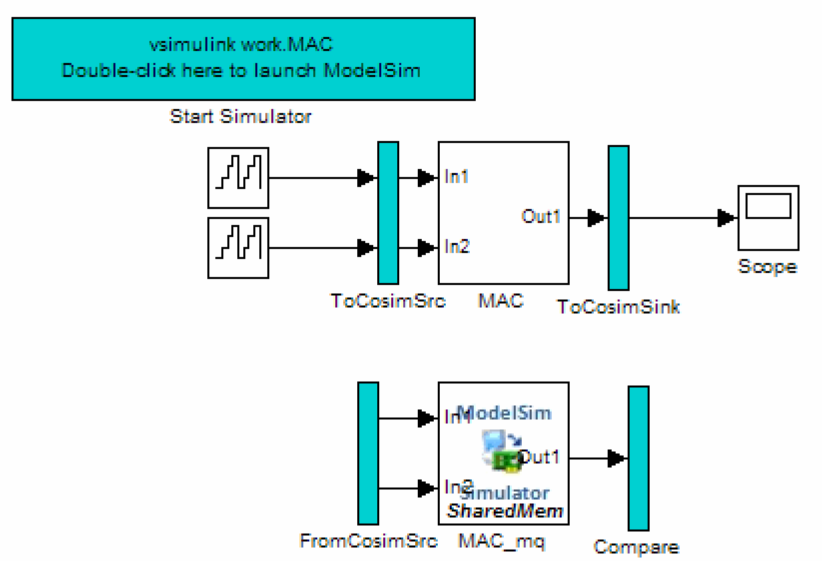
1. Select the **HDL Code Generation** > **Test Bench** pane of the Configuration Parameters dialog box.
2. Select the **Cosimulation model** check box. Then select your **Simulation tool** in the drop-down menu.
3. Configure required test bench options. HDL Coder records option settings in a generated script file (see "The Cosimulation Script File" on page 25-52).
4. Click **Apply**.

Next, generate test bench code and the cosimulation model:

1. At the bottom of the **Test Bench** pane, click **Generate Test Bench**. HDL Coder displays progress messages as shown in the following listing:

   ```
   ### Begin TestBench Generation
   ### Generating new cosimulation model: gm_hdl_cosim_demo1_mq0.mdl
   ### Generating new cosimulation tcl script: hdlsrc/gm_hdl_cosim_demo1_mq0_tcl.m
   ### Cosimulation Model Generation Complete.

   ### Generating Test bench: hdlsrc\MAC_tb.vhd
   ### Please wait ...
   ### HDL TestBench Generation Complete.
   ```

   When test bench generation completes, HDL Coder opens the generated cosimulated model. The following figure shows the generated model.



2. Save the generated model. The generated model exists only in memory unless you save it.

As indicated by the code generation messages, HDL Coder generates the following files in addition to the usual HDL test bench file:

- A cosimulation model (`gm_hdl_cosim_demo1_mq`)
- A file that contains a TCL cosimulation script and information about settings of the cosimulation model (`gm_hdl_cosim_demo1_mq_tcl.m`)

Generated file names derive from the model name, as described in “Naming Conventions for Generated Cosimulation Models and Scripts” on page 25-55.

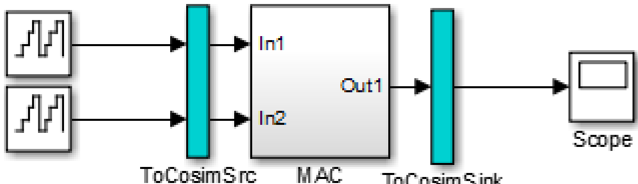
The next section, “Structure of the Generated Model” on page 25-46, describes the features of the model. Before running a cosimulation, become familiar with these features.

Structure of the Generated Model

You can set up and launch a cosimulation using controls located in the generated model. This section examines the model generated from the example MAC subsystem.

Simulation Path

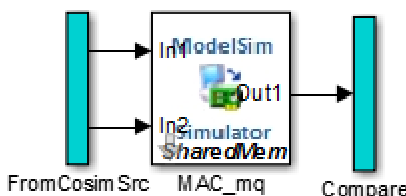
The model comprises two parallel signal paths. The simulation path, located in the upper half of the model window, is nearly identical to the original DUT. The purpose of the simulation path is to execute a normal Simulink simulation and provide a reference signal for comparison to the cosimulation results. The following figure shows the simulation path.



The two subsystems labelled `ToCosimSrc` and `ToCosimSink` do not change the performance of the simulation path. Their purpose is to capture stimulus and response signals of the DUT and route them to and from the HDL cosimulation block (see “Signal Routing Between Simulation and Cosimulation Paths” on page 25-48).

Cosimulation Path

The cosimulation path, located in the lower half of the model window, contains the generated HDL Cosimulation block. The following figure shows the cosimulation path.

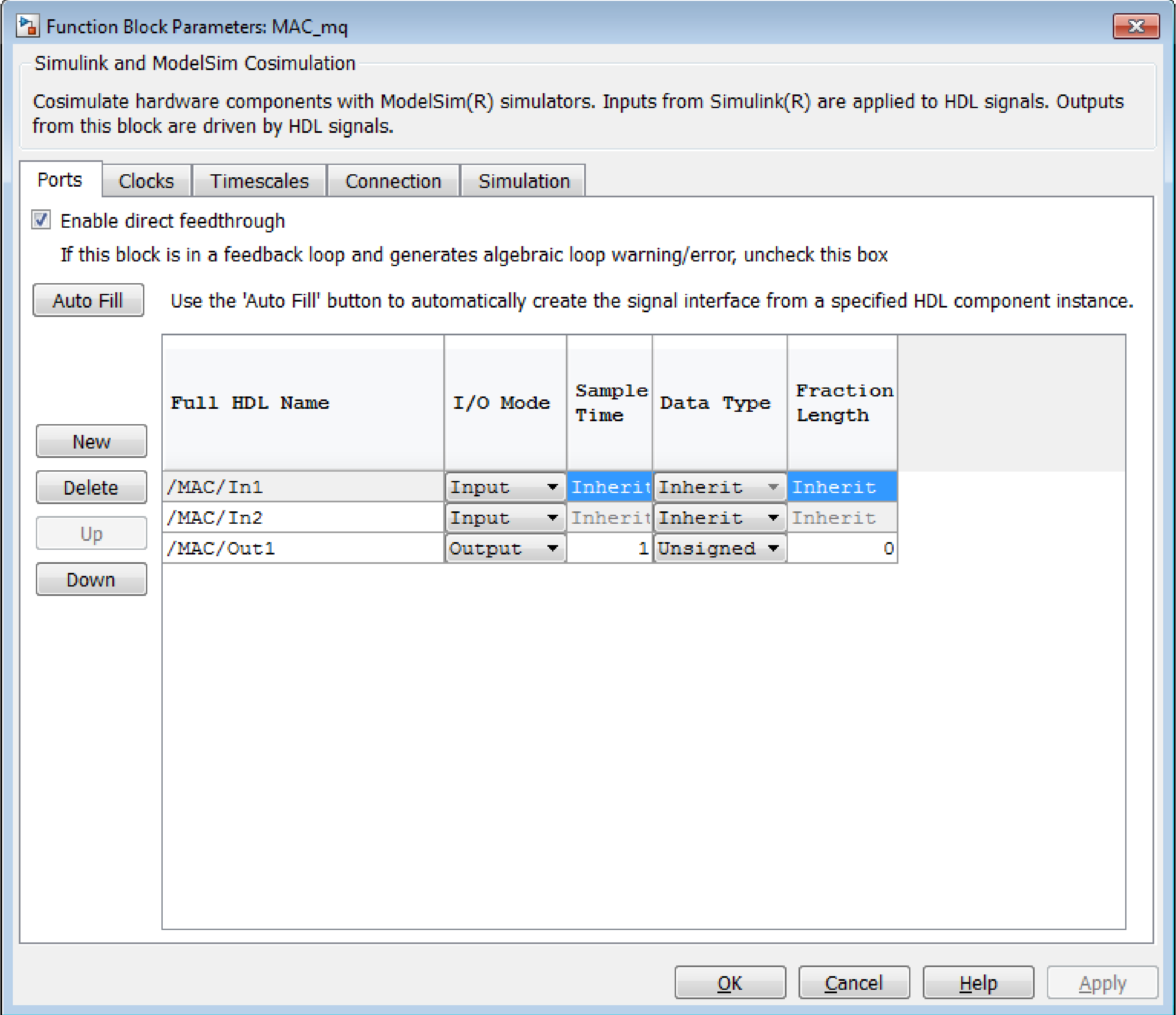


The `FromCosimSrc` subsystem receives the same input signals that drive the DUT. In the `gm_hdl_cosim_demo1_mq0` model, the subsystem simply passes the inputs on to the HDL Cosimulation block. Signals of some other data types require further processing at this stage (see “Signal Routing Between Simulation and Cosimulation Paths” on page 25-48).

The `Compare` subsystem at the end of the cosimulation path compares the cosimulation output to the reference output produced by the simulation path. If the comparison detects a discrepancy, an Assertion block in the `Compare` subsystem displays a warning message. If desired, you can disable

assertions and control other operations of the `Compare` subsystem. See “Controlling Assertions and Scope Displays” on page 25-49 for details.

HDL Coder populates the HDL Cosimulation block with the compiled I/O interface of the DUT. The following figure shows the **Ports** pane of the `Mac_mq` HDL Cosimulation block.

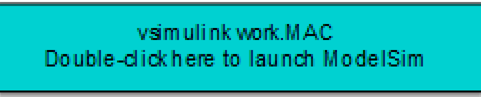


HDL Coder sets the **Full HDL Name**, **Sample Time**, **Data Type**, and other fields as required by the model. HDL Coder also configures other HDL Cosimulation block parameters under the **Timescales** and **Tcl** panes.

Tip HDL Coder configures the generated HDL Cosimulation block for the `Shared Memory` connection method.

Start Simulator Control

ModelSim or Incisive® users — When you double-click the Start Simulator control, it launches the selected cosimulation tool and passes in a startup command to the tool. The Start Simulator icon displays the startup command, as shown in the following figure.



The commands executed when you double-click the Start Simulator icon launch and set up the cosimulation tool, but they do not start the actual cosimulation. “Launching a Cosimulation” on page 25-50 describes how to run a cosimulation with the generated model.

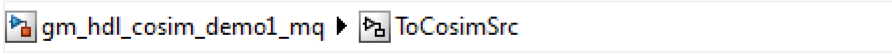
Vivado simulator users — When you double-click the Start Simulator control, it regenerates a shared library (DLL file). The Start Simulator icon displays that information, as shown in the following figure.

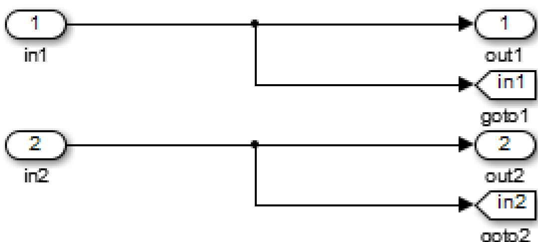
Double-click to
regenerate Vivado HDL DLL
for design MAC

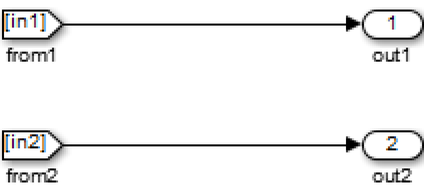
Start Simulator

Signal Routing Between Simulation and Cosimulation Paths

The generated model routes signals between the simulation and cosimulation paths using Goto and From blocks. For example, the Goto blocks in the `ToCosimSrc` subsystem route each DUT input signal to a corresponding From block in the `FromCosimSrc` subsystem. The following figures show the Goto and From blocks in each subsystem.



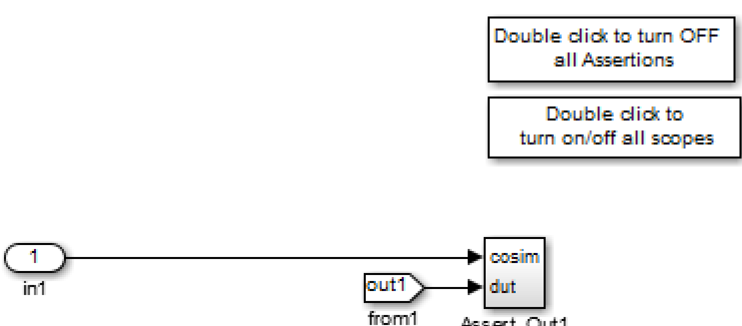




The preceding figures show simple scalar inputs. Signals of complex and vector data types require further processing. See "Complex and Vector Signals in the Generated Cosimulation Model" on page 25-54 for further information.

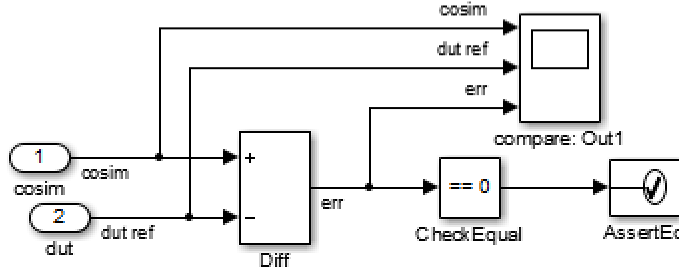
Controlling Assertions and Scope Displays

The `Compare` subsystem lets you control the display of signals on scopes, and warning messages from assertions. The following figure shows the `Compare` subsystem for the `gm_hdl_cosim_demo1_mq0` model.



For each output of the DUT, HDL Coder generates an assertion checking subsystem (`Assert_OutN`). The subsystem computes the difference (`err`) between the original DUT output (`dut ref`) and the corresponding cosimulation output (`cosim`). The subsystem routes the comparison result to an Assertion block. If the comparison result is not zero, the Assertion block reports the discrepancy.

The following figure shows the `Assert_Out1` subsystem for the `gm_hdl_cosim_demo1_mq0` model.



This subsystem also routes the `dut ref`, `cosim`, and `err` signals to a Scope for display at the top level of the model.

By default, the generated cosimulation model enables all assertions and displays all Scopes. Use the buttons on the `Compare` subsystem to disable assertions or hide Scopes.

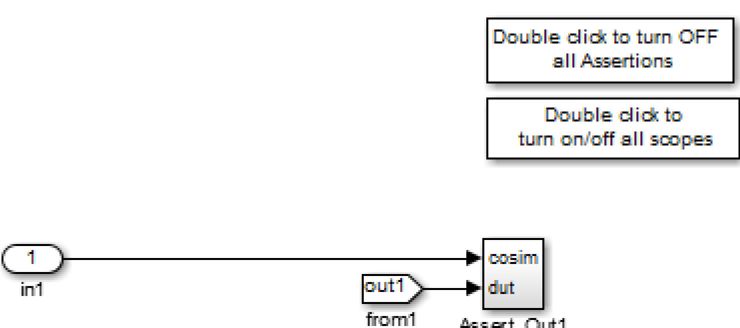
Tip Assertion messages are warnings and do not stop simulation.

Launching a Cosimulation

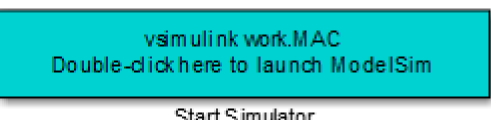
To run a cosimulation with the generated model:

1. Double-click the `Compare` subsystem to configure Scopes and assertion settings.

   If you want to disable Scope displays or assertion warnings before starting your cosimulation, use the buttons on the `Compare` subsystem (shown in the following figure).



2 Double-click the Start Simulator control.



The Start Simulator control launches your HDL simulator (in this case, HDL Verifier for use with Mentor Graphics ModelSim).

The HDL simulator in turn executes a startup script. In this case the startup script consists of the TCL commands located in `gm_hdl_cosim_demo1_mq0_tcl.m`. When the HDL simulator finishes executing the startup script, it displays a message like the following.

```
# Ready for cosimulation...
```

3 In the Simulink Editor for the generated model, start simulation.

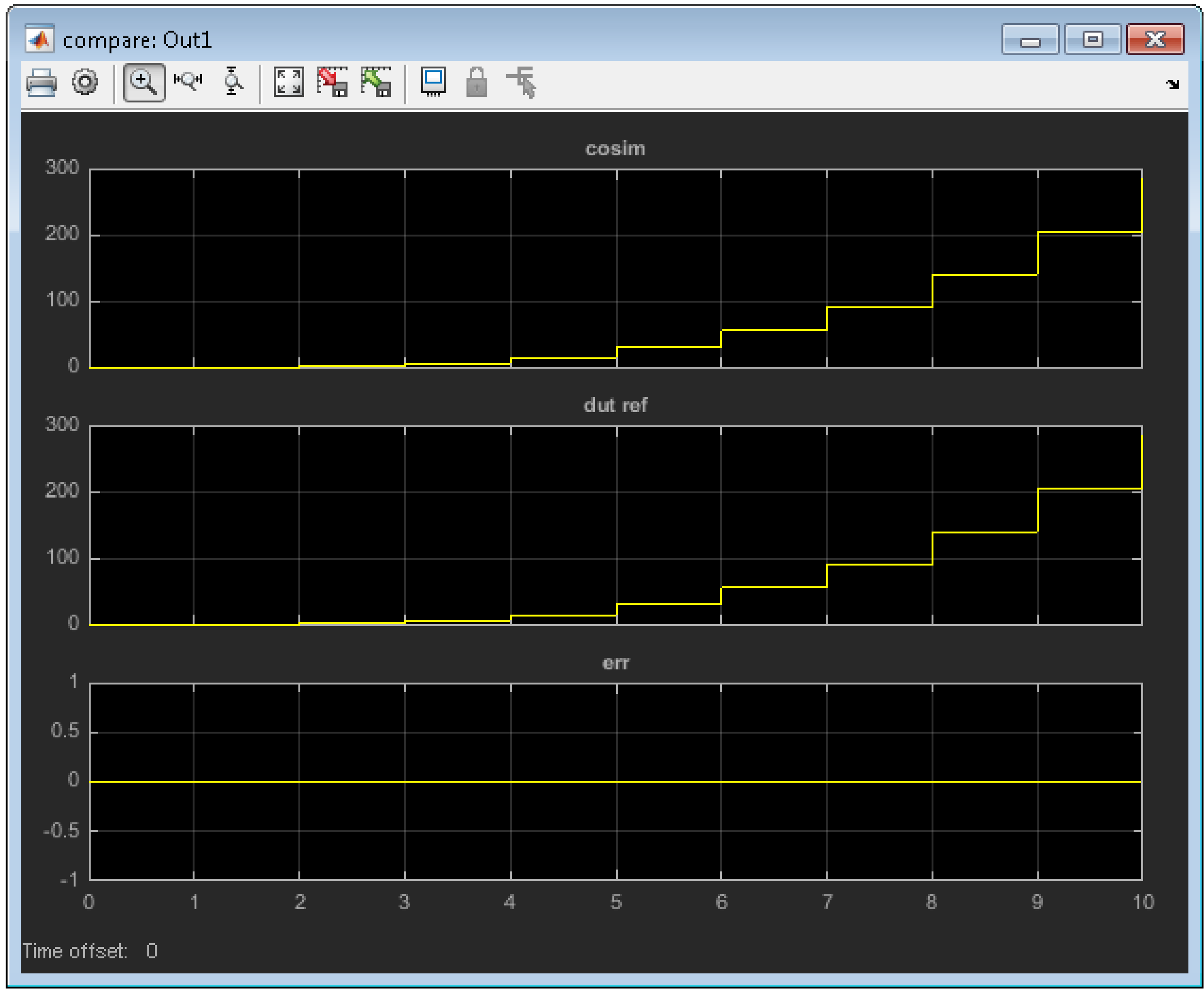
As the cosimulation runs, the HDL simulator displays messages like the following.

```
# Running Simulink Cosimulation block.
# Chip Name: --> hdl_cosim_demo1/MAC
# Target language: --> vhdl
# Target directory: --> hdlsrc
# Fri Jun 05 4:26:34 PM Eastern Daylight Time 2009
# Simulation halt requested by foreign interface.
# done
```

At the end of the cosimulation, if you have enabled Scope displays, the compare scope displays the following signals:

- `cosim`: The result signal output by the HDL Cosimulation block.
- `dut ref`: The reference output signal from the DUT.
- `err`: The difference (error) between these two outputs.

The following figure shows these signals.



The Cosimulation Script File

The generated script file has two sections:

- A comment section that documents model settings that are relevant to cosimulation.
- A function that stores several lines of TCL code into a variable, `tclCmds`. The cosimulation tools execute these commands when you launch a cosimulation.

Header Comments Section

The following listing shows the comment section of a script file generated for the `hdl_cosim_demo1` model:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auto generated cosimulation 'tclstart' script
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Source Model         : hdl_cosim_demo1.mdl
%  Generated Model      : gm_hdl_cosim_demo1.mdl
%  Cosimulation Model   : gm_hdl_cosim_demo1_mq.mdl
%
%  Source DUT           : gm_hdl_cosim_demo1_mq/MAC
%  Cosimulation DUT     : gm_hdl_cosim_demo1_mq/MAC_mq
```

```
%
%  File Location        : hdlsrc/gm_hdl_cosim_demo1_mq_tcl.m
%  Created              : 2009-06-16 10:51:01
%
%  Generated by MATLAB 7.9 and HDL Coder 1.6
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  ClockName           : clk
%  ResetName           : reset
%  ClockEnableName     : clk_enable
%
%  ClockLowTime        : 5ns
%  ClockHighTime       : 5ns
%  ClockPeriod         : 10ns
%
%  ResetLength         : 20ns
%  ClockEnableDelay    : 10ns
%  HoldTime            : 2ns
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  ModelBaseSampleTime   : 1
%  OverClockFactor     : 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Mapping of DutBaseSampleTime to ClockPeriod
%
%  N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
%  1 sec in Simulink corresponds to 10ns in the HDL
%  Simulator(N = 10)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  ResetHighAt          : (ClockLowTime + ResetLength + HoldTime)
%  ResetRiseEdge        : 27ns
%  ResetType            : async
%  ResetAssertedLevel   : 1
%
%  ClockEnableHighAt    : (ClockLowTime + ResetLength + ClockEnableDelay + HoldTime)
%  ClockEnableRiseEdge  : 37ns
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The comments section comprises the following subsections:

- *Header comments*: This section documents the files names for the source and generated models and the source and generated DUT.
- *Test bench settings*: This section documents the `makehdltb` property values that affect cosimulation model generation. The generated TCL script uses these values to initialize the cosimulation tool.
- *Sample time information*: The next two sections document the base sample time and oversampling factor of the model. HDL Coder uses `ModelBaseSampleTime` and `OverClockFactor` to map the clock period of the model to the HDL cosimulation clock period.
- *Clock, clock enable, and reset waveforms*: This section documents the computations of the duty cycle of the `clk`, `clk_enable`, and `reset` signals.

TCL Commands Section

The following listing shows the TCL commands section of a script file generated for the `hdl_cosim_demo1` model:

```
function tclCmds = gm_hdl_cosim_demo1_mq_tcl
tclCmds = {
    'do MAC_compile.do',...% Compile the generated code
    'vsimulink work.MAC',...% Initiate cosimulation
    'add wave  /MAC/clk',...% Add wave commands for chip input signals
    'add wave  /MAC/reset',...
    'add wave  /MAC/clk_enable',...
    'add wave  /MAC/In1',...
```

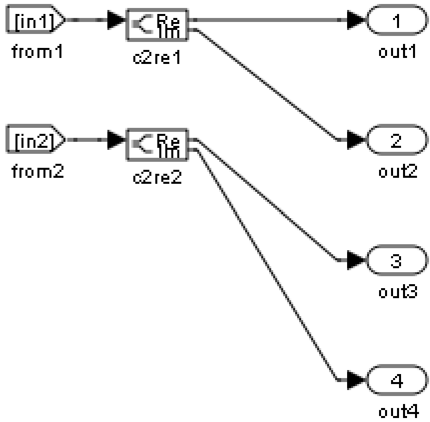
```
    'add wave  /MAC/In2',...
    'add wave  /MAC/ce_out',...% Add wave commands for chip output signals
    'add wave  /MAC/Out1',...
    'set UserTimeUnit ns',...% Set simulation time unit
    'puts ""',...
    'puts "Ready for cosimulation..."',...
};
end
```

Complex and Vector Signals in the Generated Cosimulation Model

Input signals of complex or vector data types require insertion of additional elements into the cosimulation path. this section describes these elements.

Complex Signals

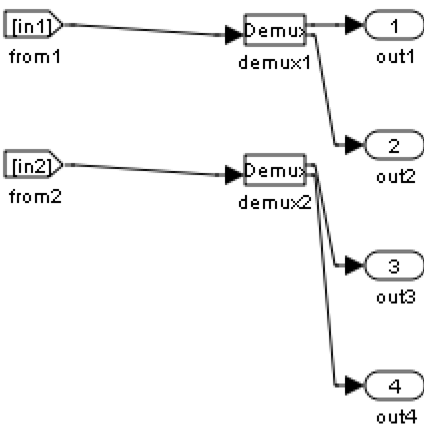
The generated cosimulation model automatically breaks complex inputs into real and imaginary parts. The following figure shows a `FromCosimSrc` subsystem that receives two complex input signals. The subsystem breaks the inputs into real and imaginary parts before passing them to the subsystem outputs.



The model maintains the separation of real and imaginary components throughout the cosimulation path. The `Compare` subsystem performs separate comparisons and separate scope displays for the real and imaginary signal components.

Vector Signals

The generated cosimulation model flattens vector inputs. The following figure shows a `FromCosimSrc` subsystem that receives two vector input signals of dimension 2. The subsystem flattens the inputs into scalars before passing them to the subsystem outputs.



Generating a Cosimulation Model from the Command Line

To generate a cosimulation model from the command line, pass the `GenerateCosimModel` property to the `makehdltb` function. `GenerateCosimModel` takes one of the following property values:

- `'ModelSim'` : generate a cosimulation model configured for HDL Verifier for use with Mentor Graphics ModelSim.
- `'Incisive'`: generate a cosimulation model configured for HDL Verifier for use with Cadence Incisive.
- `'Vivado Simulator'`: generate a cosimulation model configured for HDL Verifier for use with Xilinx Vivado.

In the following command, `makehdltb` generates a cosimulation model configured for HDL Verifier for use with Mentor Graphics ModelSim.

```
makehdltb('hdl_cosim_demo1/MAC','GenerateCosimModel','ModelSim');
```

Naming Conventions for Generated Cosimulation Models and Scripts

The naming convention for generated cosimulation models is

prefix_modelname_toolid_suffix, where:

- *prefix* is the string `gm`.
- *modelname* is the name of the generating model.
- *toolid* is an identifier indicating the HDL simulator chosen by the **Cosimulation model for use with:** option. Valid *toolid* strings are `'mq'` , `'in'`, or `'vs'`.
- *suffix* is an integer that provides each generated model with a unique name. The suffix increments with each successive test bench generation for a given model. For example, if the original model name is `test`, then the sequence of generated cosimulation model names is `gm_test_toolid_0`, `gm_test_toolid_1`, and so on.

The naming convention for generated cosimulation scripts is the same as that for models, except that the file name extension is `.m`.

Limitations for Cosimulation Model Generation

When you configure a model for cosimulation model generation, observe the following limitations:

- Explicitly specify the sample times of source blocks to the DUT in the simulation path. Use of the default sample time (`-1`) in the source blocks may cause sample time propagation problems in the cosimulation path of the generated model.
- The HDL Coder software does not support continuous sample times for cosimulation model generation. Do not use sample times `0` or `Inf` in source blocks in the simulation path.
- If you set **Clock Inputs** to `Multiple`, HDL Coder does not support generation of a cosimulation model.
- Combinatorial output paths (caused by absence of registers in the generated code) have a latency of one extra cycle in cosimulation. To avoid discrepancy in the comparison between the simulation and cosimulation outputs, the **Allow direct feedthrough** option on the **Ports** pane of the HDL Cosimulation block is automatically selected.

  Alternatively, you can avoid the latency by specifying output pipelining (see “OutputPipeline” on page 19-19). This will fully register outputs during code generation.
- Double data types are not supported for the HDL Cosimulation block. Avoid use of double data types in the simulation path when generating HDL code and a cosimulation model.

HDL Verifier Cosimulation Model Generation in HDL Coder

This example shows how to reuse an existing Simulink® model to verify HDL Coder™ generated hardware designs using an HDL Verifier™ cosimulation test bench.

Introduction

Cosimulation is a challenging task, especially with automatically generated code. It is important to keep in-sync various aspects of the source model including sample rates, feedforward/feedthrough systems, and other various parameters and settings used during code generation, while setting up the HDL Verifier cosimulation block and the target HDL simulator.

The automated cosimulation model generation takes the guess-work out of the HDL cosimulation block and simulator setup by deciphering all the compiled model and code generation information. All of the automated settings are documented in the generated scripts. The end result is a cosimulation model that is ready to verify the generated code.

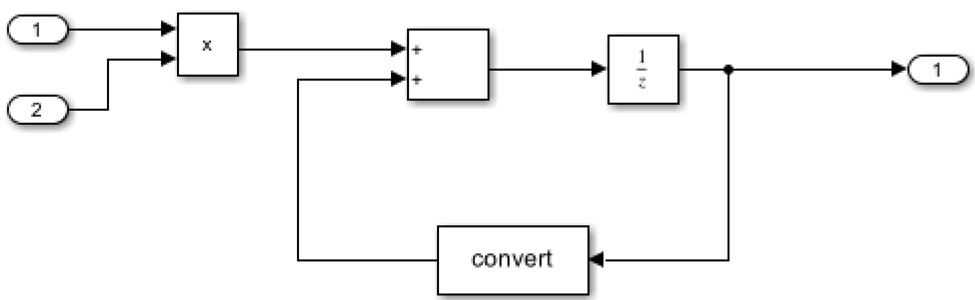
The main examples show using Mentor Graphics® ModelSim®/Questa®, but other supported simulators include Cadence® Xcelium™ (formerly Incisive®) and Xilinx® Vivado™ Simulator. Specific `makehdltb` examples are given for those simulators at the end.

Generating HDL and the Cosimulation Model

This sections shows a basic multiply-accumulate design in Simulink. We generate the HDL design and generate a cosimulation test bench model. We then go through some details of how the test bench model operates.

Generate the HDL Design

Take a simple accumulator design in Simulink and automatically generate a cosimulation model for it as a part of test bench generation.




```
% Generate VHDL code using the |makehdl| function.
makehdl('hdl_cosim_demo1/MAC', 'targetlang', 'vhdl')
```

```
### Working on the model <a href="matlab:open_system('hdl_cosim_demo1')">hdl_cosim_demo1</a>
### Generating HDL for <a href="matlab:open_system('hdl_cosim_demo1/MAC')">hdl_cosim_demo1/MAC</a
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_cosim_demo1
### Running HDL checks on the model 'hdl_cosim_demo1'.
### Begin compilation of the model 'hdl_cosim_demo1'...
### Working on the model 'hdl_cosim_demo1'...
```

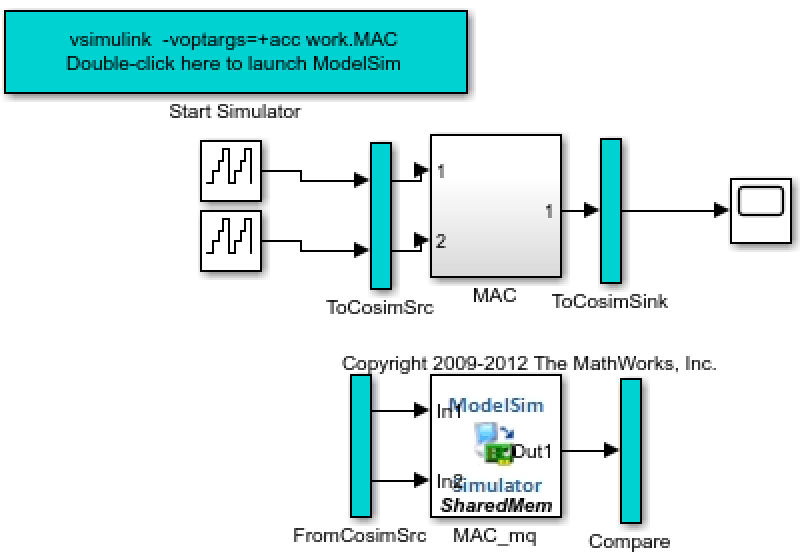
```
### Working on... <a href="matlab:configset.internal.open('hdl_cosim_demo1', 'GenerateModel')">Ge
### Begin model generation 'gm_hdl_cosim_demo1'...
### Copying DUT to the generated model....
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_dem
### Begin VHDL Code Generation for 'hdl_cosim_demo1'.
### Working on hdl_cosim_demo1/MAC as hdlsrc\hdl_cosim_demo1\MAC.vhd.
### Code Generation for 'hdl_cosim_demo1' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/14/tp
### HDL check for 'hdl_cosim_demo1' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

Generate the Cosimulation Model as an HDL Test Bench

Using the model used for creating the Simulink design as a starting point, generate a cosimulation model to use as an HDL test bench. Set the makehdl parameter `GenerateCosimModel` to choose the HDL simulator. Supported values include `ModelSim`, `Incisive`, or `Vivado Simulator`.

```
% Generate the cosimulation test bench for the MAC HDL
makehdltb('hdl_cosim_demo1/MAC', 'targetlang', 'vhdl', 'GenerateCosimModel', 'ModelSim')
```

```
### Begin TestBench generation.
### Generating HDL TestBench for 'hdl_cosim_demo1/MAC'.
### Begin compilation of the model 'hdl_cosim_demo1'...
### Begin compilation of the model 'gm_hdl_cosim_demo1'...
### Generating new cosimulation model: '<a href="matlab:open_system('gm_hdl_cosim_demo1_mq')">gm_
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_demo1_mq_tcl.m.
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_demo1_mq_batch_tc
### Note: Option 'Allow direct feedthrough' has been set to 'on' on 'gm_hdl_cosim_demo1_mq/MAC_mo
### Begin simulation of the model 'gm_hdl_cosim_demo1'...

### Collecting data...
### Generating test bench data file: hdlsrc\hdl_cosim_demo1\In1.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo1\In2.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo1\Out1_expected.dat.
### Working on MAC_tb as hdlsrc\hdl_cosim_demo1\MAC_tb.vhd.
### Generating package file hdlsrc\hdl_cosim_demo1\MAC_tb_pkg.vhd.
### HDL TestBench generation complete.
```



New Code Generation Messages

As seen from the additional code generation messages in the command window a cosimulation model `gm_hdl_cosim_demo1_mq` is generated; In addition to the code generated in the target directory, `hdlsrc`, an additional cosimulation script `gm_hdl_cosim_demo1_mq_tcl.m` is generated to prepare the target simulator for cosimulation with Simulink.

```
### Generating new cosimulation model: gm_hdl_cosim_demo1_mq
### Generating new cosimulation tcl script: hdlsrc/gm_hdl_cosim_demo1_mq_tcl.m
### Cosimulation Model Generation Complete.
```

(Optional) Generate HDL Code Coverage Report and Database

To instrument the HDL Simulator to generate a code coverage database, either:

a) On the **HDL Code Generation > Test Bench** pane, select the check box labeled **HDL code coverage**.

b) When you call `makehdltb`, set `HDLCodeCoverage` to `on`. For example:

```
makehdltb('hdl_cosim_demo1/MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'ModelSim','HDLCodeCo
```

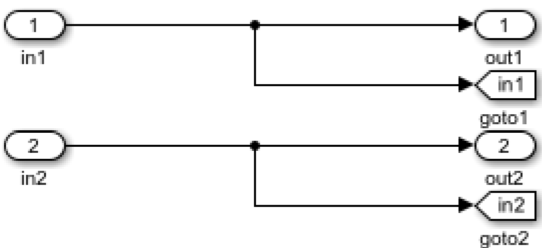
The HDL code coverage artifacts are generated in the source directory after the test bench is simulated.

Examining the Cosimulation Model Test Bench Features

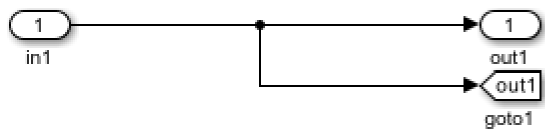
Stimulus and Response Capture

As you can see from the cosimulation model the original device under test (DUT) is intercepted by two subsystems `ToCosimSrc` and `ToCosimSink`. The purpose of these two subsystems is to capture the stimulus and the response of the DUT and use it for driving the cosimulation using `Goto` blocks. The number of `Goto` blocks in each of the following subsystems match the number of inputs and outputs of the DUT.

`ToCosimSrc`:



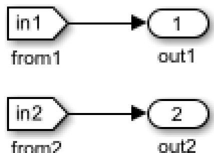
`ToCosimSink`:



Stimulus to the HDL Cosimulation Block

The stimulus that is originally driving the DUT is fed to the fully configured HDL cosimulation block using the `From` block as shown below. In some cases input stimulus signals cannot be directly fed to the HDL Cosimulation block. For example, the HDL design might have flattened structured datatypes such as complex signals and vectored signals and the HDL cosimulation block reflects the RTL interface. In such cases further transformation of the input stimulus signals is done automatically. In the current model, the `From` blocks directly feed the contents of corresponding `Goto` blocks.

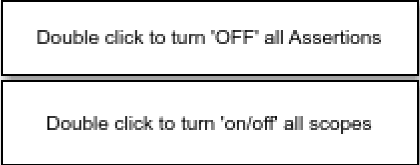
`FromCosimSrc`:

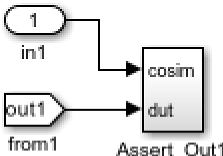


Comparison of the Results

The response from the original DUT is compared with the response from the HDL Cosimulation block in HDL Verifier using the Sink blocks provided by Simulink for visualization of the response data.

`Compare`:

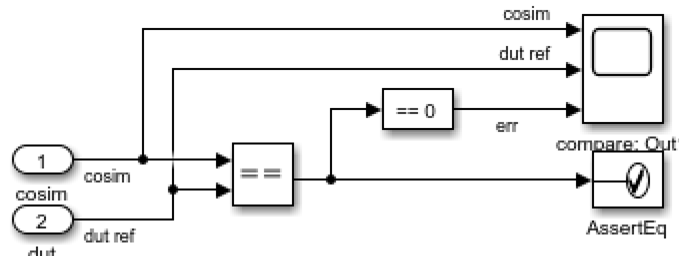




Assertion Checking in the Generated Model

For each output of the device under test subsystem the following assertion-checking model is generated that checks the original output (`dut ref`) with cosimulation output (`cosim`) and generates assertion messages when the input to the assertion block detects a mismatch.

`Assert_Out1`:



Using Assertion Blocks

Assertions are enabled in the Assertion block but do not stop simulation. If as a part of cosimulation there are any assertions from the following block you should see a warning from the Assertion block:

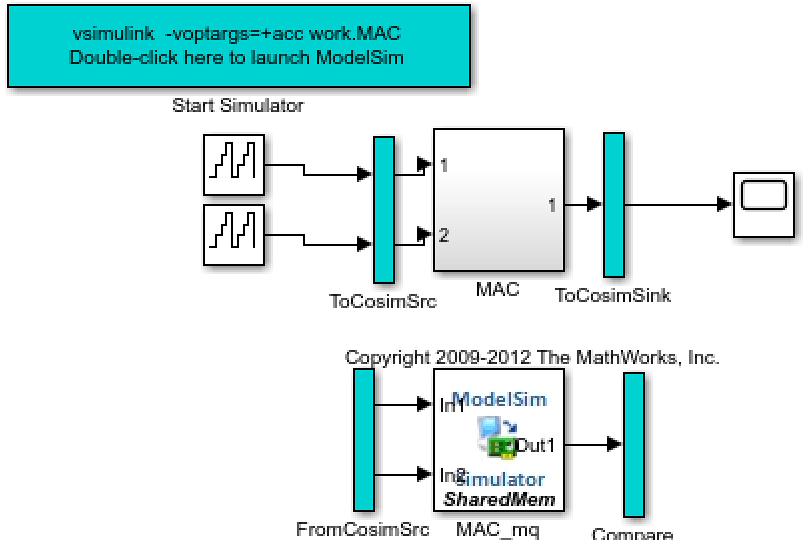
```
Warning: Assertion detected in 'gm_hdl_cosim_demo1_mq/Compare/Assert_Out1/AssertEq' at time 1.000
```

Examining the HDL Simulator Specific Support

HDL Cosimulation Block Setup

The HDL Cosimulation block is automatically populated with the compiled input output interface of the DUT. The **Ports** panel is fully populated with **Full HDL Name**, **Sample Time** and **Data type** information. Similarly, various HDL Cosimulation block setup parameters such as **Timescale** and **tcl port** panes are automatically populated. Note that **Connection method** of the cosimulation model is always configured as `Shared Memory`.

`MAC_mq`:



Target Simulator Launch and Setup

Now look at the automation associated with the launch and setup of the target simulator (ModelSim/Questa, Incisive/Xcelium, or Vivado Simulator). As can be seen in the top level of the generated model, a subsystem with the name `Start Simulator` is generated with the following callback function. This subsystem is used to launch the target simulator of choice.

`OpenFcn`:

```
ans =

    'try
       cosimDirName = pwd;
       cd 'hdlsrc\hdl_cosim_demo1';
       vsim('tclstart',gm_hdl_cosim_demo1_mq_tcl);
       cd (cosimDirName);
       clear cosimDirName;
     catch me
       disp('Failed to launch cosimulator with "vsim"');
       disp (me.message);
       cd (cosimDirName);
       clear cosimDirName;
     end'
```

Simulation of the Cosimulation Model

The code associated with the callback is simulator specific and executes the necessary commands to get the HDL simulator up and running. For ModelSim/Questa an invocation of `vsim` is made. For Incisive/Xcelium, an invocation of `nclaunch` is made. For Vivado Simulator, there is no separate process or debug environment; instead a system command compiles the design into a shared-library for the cosimulation.

The following show the details for ModelSim/Questa.

```
vsim('tclstart',gm_hdl_cosim_demo1_mq_tcl)
```

The MATLAB command `vsim` for ModelSim launches the target simulator from within MATLAB environment with the necessary setup for cosimulation. The `vsim` command is invoked with the `tclstart` option that accepts an tcl string that configures the simulator on its launch. The file `gm_hdl_cosim_demo1_mq_tcl` is also automatically generated by HDL Coder along with the cosimulation model.

Contents of the Generated tclstart Command File

The generated `tclstart` file contains commands for configuring the launched simulator as well as comments about how various settings of Cosimulation model are generated.

`hdlsrc/hdl_cosim_demo1/gm_hdl_cosim_demo1_mq_tcl`:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auto generated cosimulation 'tclstart' script
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Source Model         : hdl_cosim_demo1
%  Generated Model      : gm_hdl_cosim_demo1
%  Cosimulation Model   : gm_hdl_cosim_demo1_mq
%
%  Source DUT           : gm_hdl_cosim_demo1_mq/MAC
%  Cosimulation DUT     : gm_hdl_cosim_demo1_mq/MAC_mq
%
%  File Location        : hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_demo1_mq_tcl.m
%  Created              : 2024-02-13 21:03:18
%
%  Generated by MATLAB 24.1, HDL Coder 24.1, and Simulink 24.1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  ClockName           : clk
%  ResetName           : reset
%  ClockEnableName     : clk_enable
%
%  ClockLowTime        : 5ns
%  ClockHighTime       : 5ns
%  ClockPeriod         : 10ns
%
%  ResetLength         : 20ns
%  ClockEnableDelay    : 10ns
%  HoldTime            : 2ns
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  ModelBaseSampleTime   : 1
%  DutBaseSampleTime     : 1
%  OverClockFactor     : 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Mapping of DutBaseSampleTime to ClockPeriod
%
%  N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
%  1 sec in Simulink corresponds to 10ns in the HDL Simulator(N = 10)
```

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  ResetHighAt          : (ClockLowTime + ResetLength + HoldTime)
%  ResetRiseEdge        : 27ns
%  ResetType            : async
%  ResetAssertedLevel   : 1
%
%  ClockEnableHighAt    : (ClockLowTime + ResetLength + ClockEnableDelay + HoldTime)
%  ClockEnableRiseEdge  : 37ns
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function tclCmds = gm_hdl_cosim_demo1_mq_tcl
tclCmds = {
    'do MAC_compile.do',...% Compile the generated code
    'vsimulink -voptargs=+acc  work.MAC',...% Initiate cosimulation
    'add wave  /MAC/clk',...% Add wave commands for chip input signals
    'add wave  /MAC/reset',...
    'add wave  /MAC/clk_enable',...
    'add wave  /MAC/In1',...
    'add wave  /MAC/In2',...
    'add wave  /MAC/ce_out',...% Add wave commands for chip output signals
    'add wave  /MAC/Out1',...
    'set UserTimeUnit ns',...% Set simulation time unit
    'puts ""',...
    'puts "Ready for cosimulation..."',...
};
end
```

Header Comments in the tclstart File

The comments in the `tclstart` file specify the source model, generated model, cosimulation model, and the source and cosimulation DUT subsystems. The cosimulation DUT is placed in parallel with the source model DUT in the cosimulation model to capture any differences between the generated HDL code and the source model DUT. The cosimulation model validates the cycle-accurate and bit-true behavior of the generated code.

Visualizing HDL Signals

Wave commands are added for all top-level interface signals.

Pre-simulation Commands

In the HDL Cosimulation block, the **Pre-simulation Tcl commands** parameter contains force commands that drive the clock bundle (clock, clock-enable, reset). The **Time to run HDL simulator before cosimulation starts** parameter initiates simulation with a run time necessary to bring the chip out of reset.

Cosim block `TclPreSimCommand`:

```
ans =

    'puts "Running Simulink Cosimulation block.";
      puts "Chip Name: --> hdl_cosim_demo1/MAC";
      puts "Target language: --> vhdl";
```

```
     puts "Target directory: --> hdlsrc\hdl_cosim_demo1";
     puts [clock format [clock seconds]];
    # Clock force command;
    force /MAC/clk 0 0ns, 1 5ns -r 10ns;
    # Clock enable force command;
    force /MAC/clk_enable 0 0ns, 1 37ns;
    # Reset force command;
    force /MAC/reset 1 0ns, 0 27ns;

    '
```

Test Bench Options Affecting Cosimulation Model Generation

The next part of the tclstart script file shows all the makehdltb test bench parameters supported by HDL Coder and their initial values used in cosimulation scripts.

```
ClockName, ResetName, ClockEnableName
ClockLowTime, ClockHighTime, ClockPeriod
ResetLength, ClockEnableDelay, HoldTime
```

Model Sample Times and Mapping of DutBaseSampleTime to ClockPeriod

The next part of the comment section covers sample times in the model and how they influenced clocking of the HDL Cosimulation block in HDL Verifier.

```
  N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
  1 sec in Simulink corresponds to 10ns in the HDL Simulator(N = 10)
```

Generated tclstart Script Output

The function in `gm_hdl_cosim_demo1_mq_tcl` generates the necessary tcl command string (tclCmds).

If the `EDAScriptGeneration` option is turned `on` and compilation `do` files are generated for ModelSim as part of `makehdl`, then a single `do` command is generated. If the `EDAScriptGeneration` option is turned `off`, then explicit compilation commands are added for compiling the generated HDL code for the DUT.

Launching the Simulator and Running the Cosimulation Test Bench

Double clicking the Start Simulator block launches the simulator with the tcl commands in the generated `tclstart` MATLAB script. Once the simulator is launched all the generated code is compiled and the HDL Cosimulation block is ready for simulation.

To run the test bench, press the **Run** button. Any data value or cycle timing differences between the original Simulink design and the HDL design are flagged via assertions.

Support for Other HDL Interface Translations

Support for Complex Signals

The model `hdl_cosim_demo2` contains a MAC subsystem using complex data types. The cosimulation test bench generation automatically handles translating the Simulink types to match the HDL port interface.

```
% Generate the HDL for a MAC design using complex data types and its
```

```
% corresponding cosimulation test bench model:
load_system('hdl_cosim_demo2');
open_system('hdl_cosim_demo2/Complex MAC');
makehdl('hdl_cosim_demo2/Complex MAC', 'targetlang', 'vh');
makehdltb('hdl_cosim_demo2/Complex MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'ModelSim')
```

Observe the FromCosimSrc subsystems: the input complex signal is automatically broken into real and imaginary pieces before driving the HDL Cosimulation block.

Observe that the comparison section checks the results for real and imaginary parts of complex outputs separately.

Support for Vector Signals

The model `hdl_cosim_demo3` contains a MAC subsystem using vectored data signals. The cosimulation test bench generation automatically handles translating the Simulink types to match the flattened HDL port interface.

```
% Generate the HDL for a MAC design using vectored signals and its
% corresponding cosimulation test bench model:
load_system('hdl_cosim_demo3');
open_system('hdl_cosim_demo3/Vector MAC');
makehdl('hdl_cosim_demo3/Vector MAC', 'targetlang', 've');
makehdltb('hdl_cosim_demo3/Vector MAC', 'targetlang', 've', 'GenerateCosimModel', 'ModelSim')
```

Observe how vectored signals are handled in the FromCosimSrc and Compare subsystems.

Support for Local Multi-Rate

The model `hdl_cosim_demo4` contains a MAC subsystem with a Sum of Elements block that is configured with a Cascade implementation and requires overclocking as can be seen in the code generation messages. The cosimulation test bench generation automatically handles translating the timing interface in Simulink to that required of the HDL implementation.

```
% Generate the HDL for a MAC design with 5x overclocking and its
% corresponding cosimulation test bench model:
load_system('hdl_cosim_demo4');
open_system('hdl_cosim_demo4/LocalMR MAC');
makehdl('hdl_cosim_demo4/LocalMR MAC', 'targetlang', 'vh');
makehdltb('hdl_cosim_demo4/LocalMR MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'ModelSim');
```

Note how the time-scale settings change to offset the overclocking in the multi-rate system.

The code generation messages show an overclocking that require a five times faster clock with respect to base rate of the model. This info is encapsulated in the cosimulation model as a part of the time scale setting as per the following message:

```
N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
1 sec in Simulink corresponds to 50ns in the HDL Simulator(N = 50)
```

Support for Incisive/Xcelium

The following are the concrete commands to create an Incisive/Xcelium cosimulation test bench for the first model.

```
% Generate the HDL and cosimulation model for Incisive/Xcelium
load_system('hdl_cosim_demo1')
makehdl('hdl_cosim_demo1/MAC', 'targetlang', 'vh')
makehdltb('hdl_cosim_demo1/MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'Incisive')
type hdlsrc/hdl_cosim_demo1/gm_hdl_cosim_demo1_in_tcl
```

Support for Vivado Simulator

The following are the concrete commands to create a Vivado Simulator cosimulation test bench for the first model.

```
% Generate the HDL and cosimulation model for Vivado Simulator
load_system('hdl_cosim_demo1')
makehdl('hdl_cosim_demo1/MAC', 'targetlang', 'vh')
makehdltb('hdl_cosim_demo1/MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'Vivado Simulator')
type hdlsrc/hdl_cosim_demo1/gm_hdl_cosim_demo1_vs.tcl
```

Verify HDL Design Using SystemVerilog DPI Test Bench

This example shows how to use SystemVerilog DPI test bench for verification of HDL code where a large data set is required.

In certain applications, simulation of a large number of samples is required to verify the HDL code generated by HDL Coder™ for your algorithm. For instance, these applications require a large number of samples for algorithm verification:

a) Calculation of radar astronomy frequency channels using a polyphase filter bank.

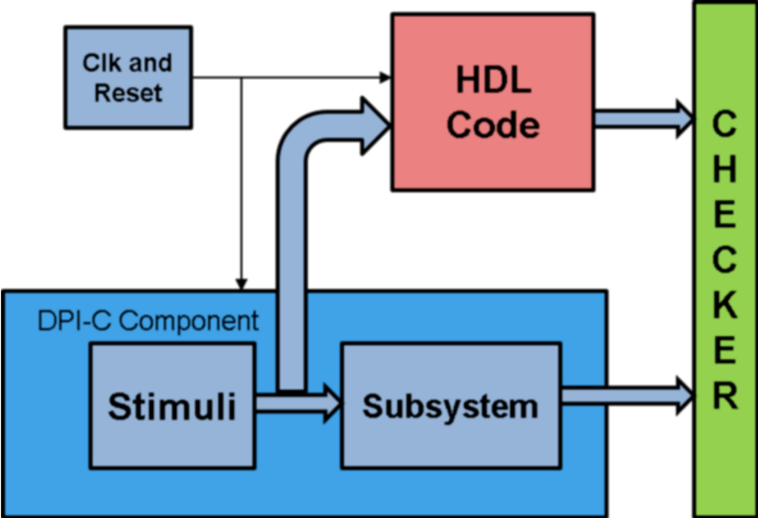
b) Obtaining the Bit Error Rate (BER) from a Viterbi decoder in a communications system.

c) Pixel-streaming video processing algorithms on high-resolution video.

Generating an HDL test bench to verify such a design is time consuming because the coder must simulate the model in Simulink® to capture the test bench data.

A faster generated test bench alternative is the HDL Verifier™ SystemVerilog DPI test bench. The SystemVerilog DPI test bench does not require a Simulink simulation, so for large data sets it generates a test bench in a shorter time than the HDL test bench. This feature requires the ASIC Testbench for HDL Verifier add-on.

SystemVerilog DPI test bench integrates with Simulink Coder™ to export a Simulink system as generated C code inside a SystemVerilog component with a Direct Programming Interface (DPI). Within the DPI-C component the stimuli is generated and applied to the C subsystem and also applied to the generated HDL code for the Simulink system. The test bench compares the output of the HDL simulation with the output of the DPI-C component to verify the HDL design.

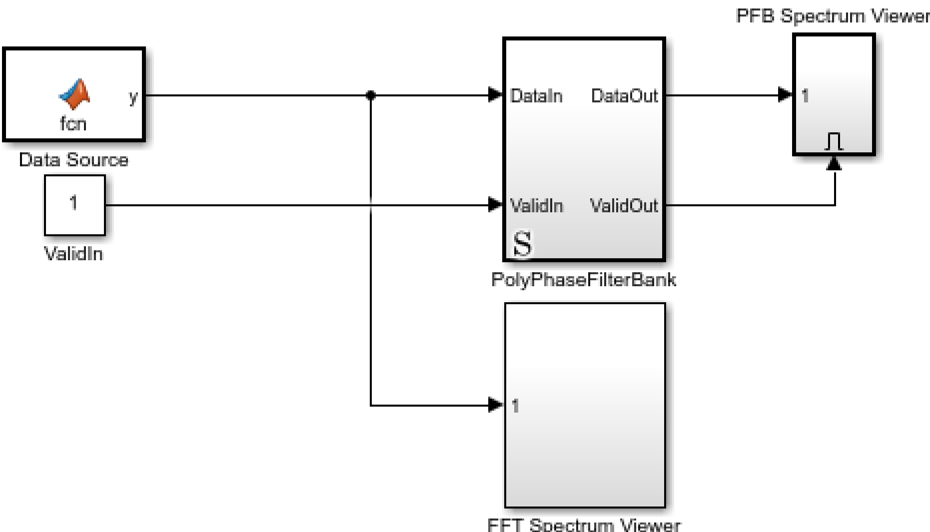


Polyphase Filter Bank

Polyphase filter bank is a widely used technique to reduce inaccuracy in FFT due to leakage and scalloping losses. A polyphase filter bank produces a flatter response as compared to a normal DFT by suppressing out-of-band signals significantly.

The model is a Polyphase Filter Bank which consists of a filter and an FFT that processes 16 samples at a time. For more information about the polyphase filter bank see “High-Throughput Channelizer for FPGA” on page 15-10.

```
modelname = 'PolyphaseFilterBankHDLExample_4tap';
open_system(modelname);
```



Set Up the Model

The example model implements a 512-point FFT by using the DSP HDL Toolbox FFT block and a 4-tap filter for each band by using basic Simulink blocks. These MATLAB variables configure the blocks in the model. Use the dsp.Channelizer System object™ to generate the coefficients. The polyphase method of the channelizer object generates a 512-by-4 matrix. Each row represents the coefficients for one band. Cast the coefficients to fixed-point types that have the same word length as the input signal.

```
simTime = 1000;
FFTLength = 512;
InVect = 4;
h = dsp.Channelizer;
h.NumTapsPerBand = 4;
h.NumFrequencyBands = FFTLength;
h.StopbandAttenuation = 60;
coef4Tap = fi(polyphase(h),1,15,14,'RoundingMethod','Convergent');
%
% The algorithm requires 512 filters (one filter for each band). For a
% vector input of 4 samples, the model implements four parallel 4-tap
% filters. Each filter applies 128 sets of coefficients.
%
ReuseFactor = FFTLength/InVect;
%
% These variables configure the model to pipeline the multipliers and
% the coefficient bank to fit the logic into DSP blocks on the FPGA. Using
```

```
% the DSP blocks enables synthesis to a higher clock rate.
%
Multiplication_PipeLine = 2;
CoefBank_PipeLine       = 1;
%
% The input data consists of two sine waves, 200 kHz and 250 kHz. The
% input and output of the PolyPhaseFilterBank subsystem are 4-by-1 vectors.
```

Generate HDL Code, HDL Test Bench, and SystemVerilog DPI Test Bench

Check the PolyphaseFilterBank subsystem for HDL code generation compatibility:

```
checkhdl('PolyphaseFilterBankHDLExample_4tap/PolyPhaseFilterBank');
```

Run the following command to generate HDL code:

```
makehdl('PolyphaseFilterBankHDLExample_4tap/PolyPhaseFilterBank');
```

Run the following command to generate the test bench:

```
makehdltb('PolyphaseFilterBankHDLExample_4tap/PolyPhaseFilterBank');
```

This command generates an HDL test bench by simulating the model in Simulink and then capturing the test bench data.

Run the following command to generate SystemVerilog DPI test bench:

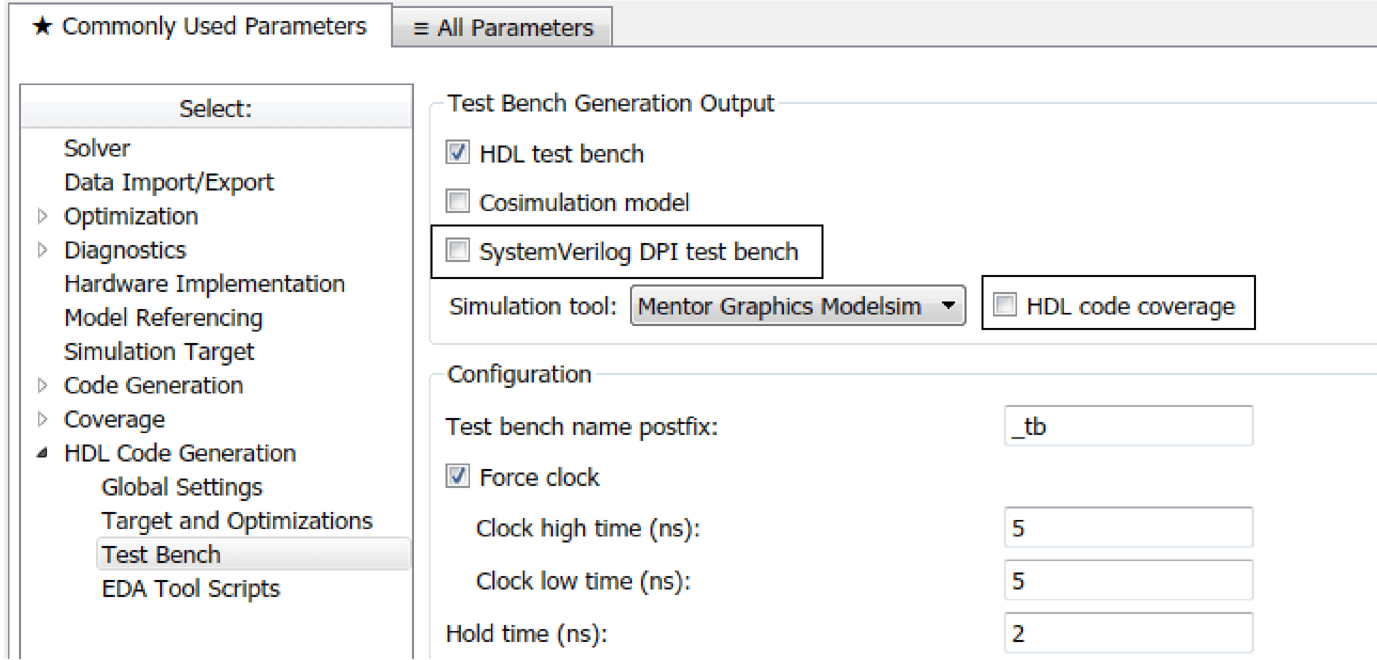
```
HDLSimulator = 'ModelSim'; % Supported Simulator Options = 'ModelSim', 'Incisive', 'VCS', 'Vivado
```

```
makehdltb('PolyphaseFilterBankHDLExample_4tap/PolyPhaseFilterBank', ...
          'GenerateSVDPITestBench', HDLSimulator, ...
          'GenerateHDLTestBench', 'Off');
```

This command generates a SystemVerilog test bench without running a Simulink simulation. Instead of a simulation, the code exports the Simulink system as generated C code inside a SystemVerilog component. The test bench verifies the output data by comparing it with the output of the HDL design. The makehdltb function also generates simulator-specific scripts for compilation and simulation.

SystemVerilog DPI test bench can be used to verify HDL designs of both target languages - VHDL® and Verilog®.

Alternatively, you can set SystemVerilog DPI test bench options on the 'HDL Code Generation > Test Bench' pane in Configuration Parameters.



Generated SystemVerilog DPI Test Bench Artifacts

When you request a SystemVerilog DPI test bench, the coder generates the following artifacts:

a.) PolyPhaseFilterBank_dpi_tb.sv - This is the SystemVerilog test bench that verifies the HDL code.

b.) PolyPhaseFilterBank_dpi_tb.do - This is the macro file that Mentor Graphics ModelSim® uses to compile the HDL code and run the test bench simulation.

Based on the selected simulator, the coder generates a different file for compilation and test bench simulation. For instance, if you select 'Incisive', the coder generates 'PolyPhaseFilterBank_dpi_tb.sh' for compilation and simulation on Cadence® Incisive®.

(Optional) Generate HDL Code Coverage Report and Database

To instrument the HDL Simulator to generate a HDL code coverage report and database, either:

a.) On the 'HDL Code Generation > Test Bench' pane, select the check box labeled 'HDL code coverage'.

b.) When you call 'makehdltb', set 'HDLCodeCoverage' to 'on'. For example:

```
makehdltb('PolyphaseFilterBankHDLExample_4tap/PolyPhaseFilterBank', ...
          'GenerateSVDPITestBench', HDLSimulator, ...
          'GenerateHDLTestBench', 'Off', ...
          'HDLCodeCoverage', 'On');
```

The HDL code coverage artifacts are generated in the source directory after the test bench is simulated.

Generation Time Comparison of HDL Test Bench and SystemVerilog DPI Test Bench

The example model runs for 1000 time steps, specified by the simTime variable. The sampling frequency is 2e+6 Hz, which means that the simulation to generate the HDL testbench collects 8e+9 samples.

For certain applications, it takes more samples to obtain the right frequency from the polyphase filter. An increase in simulation time also increases the time required to generate an HDL test bench.

A solution for such applications is to use the SystemVerilog DPI test bench. The generation time for the test bench remains the same no matter how many samples your test scenario requires.

You can increase the Simulation Time by changing the 'simTime' variable. For instance to generate an HDL test bench for 2e+12 samples, set:

```
simTime = 1000000;
```

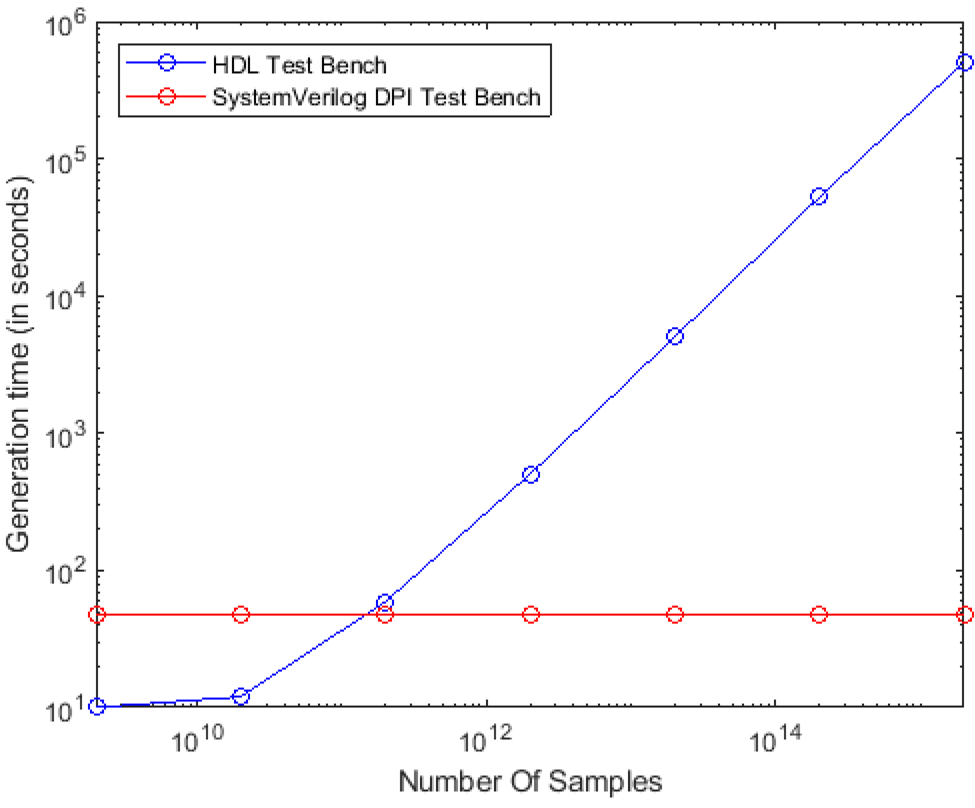
The table shows a comparison of time taken (in seconds) for generation of HDL test bench and SystemVerilog DPI test bench for increasing numbers of samples (from 2e+9 to 2e+15) :

```
columns = {'NumberOfSamples';'GenerationTimeHDLTestBench';'GenerationTimeSystemVerilogDPITestbenc
numSamples = [2e9;2e10;2e11;2e12;2e13;2e14;2e15];
HDLTBtime= [10;12;59;504;4994;52200;505506];
DPICTBtime=[47;47;47;47;47;47;47];
CompareTestBenchTimes = table(numSamples,HDLTBtime,DPICTBtime,'VariableNames',columns);
disp(CompareTestBenchTimes);
```

```
    NumberOfSamples    GenerationTimeHDLTestBench    GenerationTimeSystemVerilogDPITestbench
    _______________    __________________________    _______________________________________

         2e+09                        10                               47
         2e+10                        12                               47
         2e+11                        59                               47
         2e+12                       504                               47
         2e+13                      4994                               47
         2e+14                     52200                               47
         2e+15               5.0551e+05                               47
```

A log plot of generation time for both these test bench types with respect to the Number of samples, shows that while HDL test bench requires more generation time with an increase in the number of samples, generation time for the SystemVerilog DPI test bench remains constant irrespective of the number of samples.

```
loglog(numSamples,HDLTBtime,'b-o', numSamples,DPICTBtime, 'r-o' );
xlim([2e09 2e15]);
legend('HDL Test Bench','SystemVerilog DPI Test Bench','Location','northwest');
xlabel('Number Of Samples');
ylabel('Generation time (in seconds)');
```



Conclusion

While HDL test bench is very efficient for a small number of samples, if your test scenario requires a large number of samples, HDL Verifier SystemVerilog DPI test bench provides faster test bench generation.

Pass-Through and No-Op Implementations

HDL Coder provides a pass-through or no-op implementation for some blocks. A pass-through implementation generates a wire in the HDL; a no-op implementation omits code generation for the block or subsystem. These implementations are useful in cases where you need a block for simulation, but do not need the block or subsystem in your generated HDL code.

The pass-through and no-op implementations are summarized in the following table.

Implementation	Description
Pass-through implementations	Provides a pass-through implementation in which the block's inputs are passed directly to its outputs. HDL Coder supports the following blocks with a pass-through implementation: • Convert 1-D to 2-D • Reshape • Signal Conversion • Signal Specification
No HDL	This implementation completely removes the block from the generated code. This enables you to use the block in simulation but treat it as a “no-op” in the HDL code. This implementation is used for many blocks (such as Scopes and Assertions) that are significant in simulation but are meaningless in HDL code.

Synchronous Subsystem Behavior with the State Control Block

In this section...

"What Is a State Control Block?" on page 25-75
"State Control Block Modes" on page 25-75
"Synchronous Badge for Subsystems by Using Synchronous Mode" on page 25-76
"Generate HDL Code with the State Control Block" on page 25-77

What Is a State Control Block?

When you have blocks with state, and have enable or reset ports inside a subsystem, use the `Synchronous` mode of the State Control block to:

- Provide efficient enable and reset simulation behavior on hardware.
- Generate cleaner HDL code and use fewer resources on hardware.

You can add the State Control block to your Simulink model at any level in the model hierarchy. How you set the State Control block affects the simulation behavior of other blocks inside the subsystem that have state.

- For synchronous hardware simulation behavior, set **State control** to `Synchronous`.
- For default Simulink simulation behavior, set **State control** to `Classic`.

State Control Block Modes

Functionality	Synchronous mode	Classic mode
State Control block setting	Default block setting when you add the block from the HDL Subsystems block library.	The simulation behavior is the same as a subsystem that does not use the State Control block.
Simulink simulation behavior • Initialize method: Initializes states. • Update method: Updates states. • Output method: Computes output values.	The update method only updates states. The output method computes the output values at each time step. For example, when you have enabled subsystems, the output value changes when the enable signal is low as it processes new input values. The output value matches the output from `Classic` mode when enable signal becomes high.	The update method updates states and computes the output values. For example, when you have enabled subsystems, the output value is held steady when the enable signal is low and changes only when the enable signal becomes high.
HDL simulation behavior	More efficient on hardware.	Less efficient on hardware.

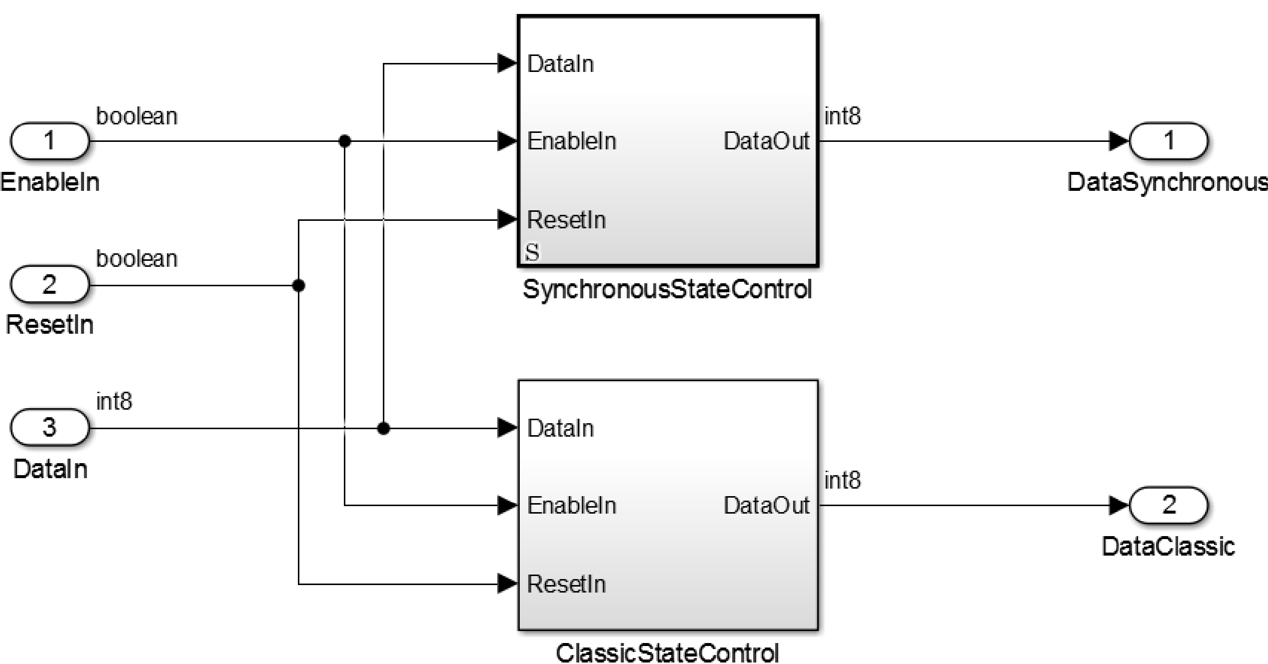
Functionality	Synchronous mode	Classic mode
HDL code generation behavior	Generated HDL code is cleaner and uses fewer resources on hardware. For example, when you have enabled subsystems, HDL Coder does not generate bypass registers for each state update and uses fewer hardware resources.	Generated HDL code is not as clean and uses more hardware resources. For example, when you have enabled subsystems, HDL Coder generates bypass registers for each state update and uses more resources.

To learn more about when you can use the State Control block, see State Control.

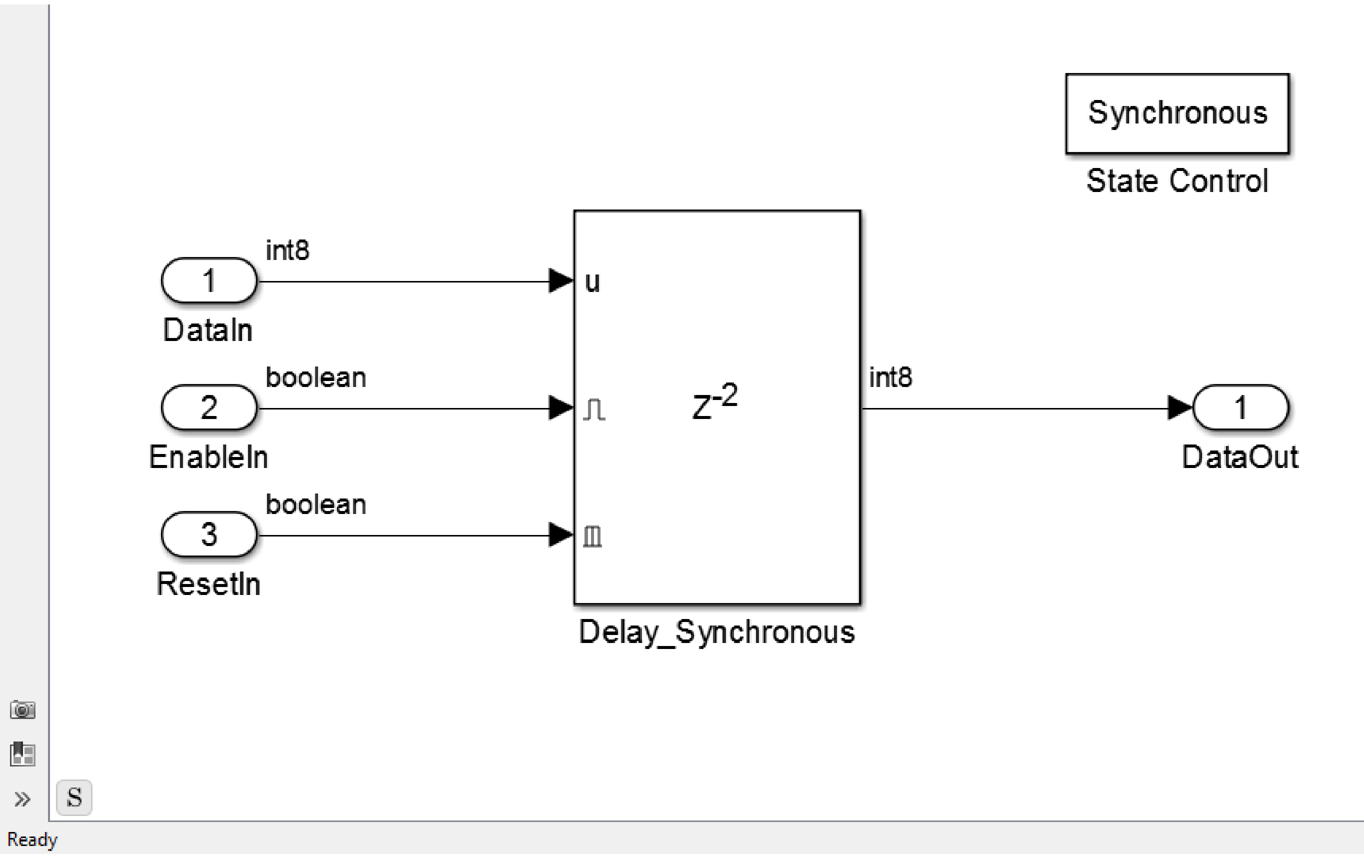
Synchronous Badge for Subsystems by Using Synchronous Mode

To see if a subsystem in your Simulink model uses synchronous semantics:

- A symbol **S** is displayed on the subsystem to indicate synchronous behavior.



- If you double-click the **SynchronousStateControl** subsystem, a badge **S** is displayed in the Simulink editor to indicate that blocks inside the subsystem are using synchronous hardware semantics.



The **SynchronousStateControl** and **ClassicStateControl** subsystems use a Delay block with an external reset and an enable port in `Synchronous` and `Classic` modes respectively.

Generate HDL Code with the State Control Block

The following table shows a comparison of the HDL code generated from the Delay block for `Classic` and `Synchronous` modes of the State Control block.

<table>
<tr><th>Functionality</th><th>Synchronous mode</th><th>Classic mode</th></tr>
<tr>
<td>HDL code generation. Settings applied:

Language: <code>Verilog</code>
Reset type: <code>Synchronous</code>

</td>
<td>
<pre><code>`timescale 1 ns / 1 ns

module SynchronousStateControl
          (
           clk,
           reset,
           enb,
           DataIn,
           EnableIn,
           ResetIn,
           DataOut
          );

  input   clk;
  input   reset;
  input   enb;
  input   signed [7:0] DataIn; // int8
  input   EnableIn;
  input   ResetIn;
  output signed [7:0] DataOut; // int8

  reg signed [7:0] Delay_Synchronous_reg [0:1]; // sfix8 [2]
  wire signed [7:0] Delay_Synchronous_reg_next [0:1]; // sfix8 [2]
  wire signed [7:0] Delay_Synchronous_out1; // int8

  always @(posedge clk)
    begin : Delay_Synchronous_process
      if (reset == 1'b1 || ResetIn == 1'b1) begin
        Delay_Synchronous_reg[0] <= 8'sb00000000;
        Delay_Synchronous_reg[1] <= 8'sb00000000;
      end
      else begin
        if (enb && EnableIn) begin
          Delay_Synchronous_reg[0] <= Delay_Synchronous_reg_next[0];
          Delay_Synchronous_reg[1] <= Delay_Synchronous_reg_next[1];
        end
      end
    end

  assign Delay_Synchronous_out1 = Delay_Synchronous_reg[1];
  assign Delay_Synchronous_reg_next[0] = DataIn;
  assign Delay_Synchronous_reg_next[1] = Delay_Synchronous_reg[0];

  assign DataOut = Delay_Synchronous_out1;

endmodule // SynchronousStateControl</code></pre>

Generated HDL code is cleaner and requires fewer hardware resources as HDL Coder does not generate bypass registers.
The update method only updates the states.

</td>
<td>
<pre><code>`timescale 1 ns / 1 ns

module ClassicStateControl
          (
           clk,
           reset,
           enb,
           DataIn,
           EnableIn,
           ResetIn,
           DataOut
          );

  input   clk;
  input   reset;
  input   enb;
  input   signed [7:0] DataIn; // int8
  input   EnableIn;
  input   ResetIn;
  output signed [7:0] DataOut; // int8

  reg signed [7:0] Delay_Synchronous_bypass; // sfix8
  reg signed [7:0] Delay_Synchronous_reg [0:1]; // sfi
  wire signed [7:0] Delay_Synchronous_bypass_next; //
  wire signed [7:0] Delay_Synchronous_reg_next [0:1];
  wire signed [7:0] Delay_Synchronous_delay_out; // sf
  wire signed [7:0] Delay_Synchronous_out1; // int8

  always @(posedge clk)
    begin : Delay_Synchronous_process
      if (reset == 1'b1 || ResetIn == 1'b1) begin
        Delay_Synchronous_bypass <= 8'sb00000000;
        Delay_Synchronous_reg[0] <= 8'sb00000000;
        Delay_Synchronous_reg[1] <= 8'sb00000000;
      end
      else begin
        if (enb && EnableIn) begin
          Delay_Synchronous_bypass <= Delay_Synchronous_
          Delay_Synchronous_reg[0] <= Delay_Synchronous_
          Delay_Synchronous_reg[1] <= Delay_Synchronous_
        end
      end
    end

  assign Delay_Synchronous_delay_out = (ResetIn == 1'b1
            Delay_Synchronous_reg[1]);
  assign Delay_Synchronous_out1 = (EnableIn == 1'b1 ? D
            Delay_Synchronous_bypass);
  assign Delay_Synchronous_bypass_next = Delay_Synchron
  assign Delay_Synchronous_reg_next[0] = DataIn;
  assign Delay_Synchronous_reg_next[1] = Delay_Synchron

  assign DataOut = Delay_Synchronous_out1;

endmodule // ClassicStateControl</code></pre>

Generated HDL code is less cleaner and requires more hardware

</td>
</tr>
</table>

Functionality	Synchronous mode	Classic mode
		resources as HDL Coder generates bypass registers. • The update method updates states and computes the output values.

See Also

State Control

Using the State Control Block to Generate More Efficient Code with HDL Coder

This example shows how to use the State Control block to generate hardware-friendly HDL code using HDL Coder™.

Introduction to the State Control block

The State Control block is a block that modifies the Simulink® simulation behavior for its containing subsystem and all subsystems nested beneath it. Its purpose is to more closely model the synchronous behavior of clocked digital hardware, particularly with respect to blocks that have state and use explicit enable and reset signals.

When a State Control block is placed in a subsystem and has its parameter set to `Synchronous`, the generated HDL code will be more hardware friendly. When a subsystem is in the Synchronous mode it is marked with a graphic S in its lower left corner. A State Control block with its parameter set to `Classic` behaves identically to when there is no State Control block in the subsystem.

The simulation behavior difference between the two modes is small, but significant to generating efficient HDL code. The differences focus around the simulation behavior involving explicit reset and enable signals. For example, in Synchronous mode, the explicit block reset input has priority over the block enable input signal.

Classic mode behavior for Delay with explicit enable input port

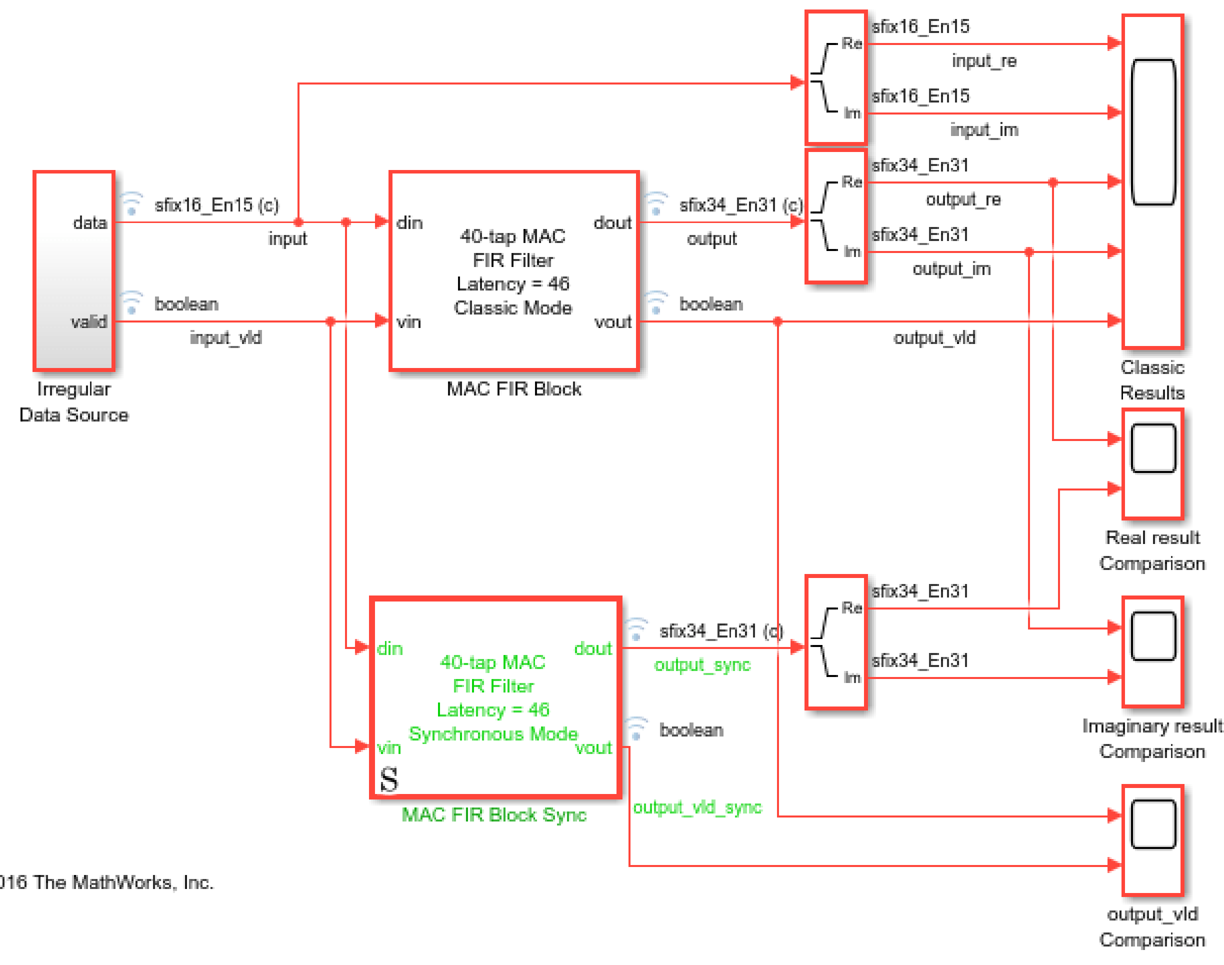
HDL code generated by HDL Coder simulates identically to the model that it is generated from. In Classic State Control mode, the generated code for certain constructs implements sub-optimal hardware due to this requirement. For example, a Delay block with explicit enable input will generate a bypass register, comprised of a register and a multiplexer, in addition to the modeled register, to capture the Simulink Classic mode behavior.

Examine the contents of Enabled_Delay.vhd to observe the additional register signals and the bypass register.

```
load_system('hdlcoder_statecontrol_model');
open_system('hdlcoder_statecontrol_model');
set_param('hdlcoder_statecontrol_model', 'SimulationCommand', 'update');
makehdl('hdlcoder_statecontrol_model/MAC FIR Block', 'TargetDirectory', 'hdlsrc_classic');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_statecontrol_model')">hdlcoder_sta
### Generating HDL for <a href="matlab:open_system('hdlcoder_statecontrol_model/MAC FIR Block')">
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_statec
### Running HDL checks on the model 'hdlcoder_statecontrol_model'.
### Begin compilation of the model 'hdlcoder_statecontrol_model'...
### Begin compilation of the model 'hdlcoder_statecontrol_model'...
### Working on the model 'hdlcoder_statecontrol_model'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_statecontrol_model', 'Genera
### Begin model generation 'gm_hdlcoder_statecontrol_model'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc_classic\hdlcoder_statecontrol_mo
### Begin VHDL Code Generation for 'hdlcoder_statecontrol_model'.
### Unused logic removed during HDL code generation. To highlight the logic removed, click the fo
### To clear highlighting, click the following MATLAB script: hdlsrc_classic\hdlcoder_statecontro
```

```
### Working on hdlcoder_statecontrol_model/MAC FIR Block/Coeff ROM as hdlsrc_classic\hdlcoder_sta
### Working on hdlcoder_statecontrol_model/MAC FIR Block/Enabled_Delay as hdlsrc_classic\hdlcode
### Working on hdlcoder_statecontrol_model/MAC FIR Block/RAM delay line/circular buffer logic as
### Working on hdlcoder_statecontrol_model/MAC FIR Block/RAM delay line/SimpleDualPortRAM_generi
### Working on hdlcoder_statecontrol_model/MAC FIR Block/RAM delay line as hdlsrc_classic\hdlcod
### Working on hdlcoder_statecontrol_model/MAC FIR Block as hdlsrc_classic\hdlcoder_statecontrol_
### Generating package file hdlsrc_classic\hdlcoder_statecontrol_model\MAC_FIR_Block_pkg.vhd.
### Code Generation for 'hdlcoder_statecontrol_model' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/t
### HDL check for 'hdlcoder_statecontrol_model' complete with 0 errors, 1 warnings, and 1 message
### HDL code generation complete.
```




```
type hdlsrc_classic/hdlcoder_statecontrol_model/Enabled_Delay.vhd
```

```
-- -------------------------------------------------------------
--
-- File Name: hdlsrc_classic\hdlcoder_statecontrol_model\Enabled_Delay.vhd
-- Created: 2024-02-13 21:38:39
--
-- Generated by MATLAB 24.1, HDL Coder 24.1, and Simulink 24.1
--
```

```
-- -------------------------------------------------------------


-- -------------------------------------------------------------
--
-- Module: Enabled_Delay
-- Source Path: hdlcoder_statecontrol_model/MAC FIR Block/Enabled_Delay
-- Hierarchy Level: 1
-- Model version: 15.0
--
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Enabled_Delay IS
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        enb                               :   IN    std_logic;
        din_re                            :   IN    std_logic_vector(33 DOWNTO 0);  -- sfix34_En3
        din_im                            :   IN    std_logic_vector(33 DOWNTO 0);  -- sfix34_En3
        LocalEnable                       :   IN    std_logic;
        Out1_re                           :   OUT   std_logic_vector(33 DOWNTO 0);  -- sfix34_En3
        Out1_im                           :   OUT   std_logic_vector(33 DOWNTO 0)  -- sfix34_En31
        );
END Enabled_Delay;


ARCHITECTURE rtl OF Enabled_Delay IS

  -- Signals
  SIGNAL din_re_signed                    : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL din_im_signed                    : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL Enabled_Delay_bypass_delay_re    : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL Enabled_Delay_bypass_delay_im    : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL Enabled_Delay_reg_re             : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL Enabled_Delay_reg_im             : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL dout_re                          : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL dout_im                          : signed(33 DOWNTO 0);  -- sfix34_En31

BEGIN
  din_re_signed <= signed(din_re);

  din_im_signed <= signed(din_im);

  Enabled_Delay_1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Enabled_Delay_bypass_delay_re <= to_signed(0, 34);
      Enabled_Delay_bypass_delay_im <= to_signed(0, 34);
      Enabled_Delay_reg_re <= to_signed(0, 34);
      Enabled_Delay_reg_im <= to_signed(0, 34);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' AND LocalEnable = '1' THEN
        Enabled_Delay_bypass_delay_im <= Enabled_Delay_reg_im;
        Enabled_Delay_reg_im <= din_im_signed;
        Enabled_Delay_bypass_delay_re <= Enabled_Delay_reg_re;
        Enabled_Delay_reg_re <= din_re_signed;
```

```
      END IF;
    END IF;
  END PROCESS Enabled_Delay_1_process;


  dout_re <= Enabled_Delay_reg_re WHEN LocalEnable = '1' ELSE
      Enabled_Delay_bypass_delay_re;

  dout_im <= Enabled_Delay_reg_im WHEN LocalEnable = '1' ELSE
      Enabled_Delay_bypass_delay_im;

  Out1_re <= std_logic_vector(dout_re);

  Out1_im <= std_logic_vector(dout_im);

END rtl;
```

Synchronous mode behavior for Delay with explicit enable input port

A Delay block with explicit enable in Synchronous State Control mode will generate HDL code that creates more efficient hardware. The implementation does not contain a bypass register.

Examine Enabled_Delay_Sync.vhd and note the improvement in the generated code as compared to the Classic mode output.

```
makehdl('hdlcoder_statecontrol_model/MAC FIR Block Sync', 'TargetDirectory', 'hdlsrc_sync');
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_statecontrol_model')">hdlcoder_sta
### Generating HDL for <a href="matlab:open_system('hdlcoder_statecontrol_model/MAC FIR Block Syn
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_stateco
### Running HDL checks on the model 'hdlcoder_statecontrol_model'.
### Begin compilation of the model 'hdlcoder_statecontrol_model'...
### Begin compilation of the model 'hdlcoder_statecontrol_model'...
### Working on the model 'hdlcoder_statecontrol_model'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_statecontrol_model', 'Genera
### Begin model generation 'gm_hdlcoder_statecontrol_model'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc_sync\hdlcoder_statecontrol_mode
### Begin VHDL Code Generation for 'hdlcoder_statecontrol_model'.
### Unused logic removed during HDL code generation. To highlight the logic removed, click the fo
### To clear highlighting, click the following MATLAB script: hdlsrc_sync\hdlcoder_statecontrol_m
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/Coeff ROM as hdlsrc_sync\hdlcoder_s
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/Enabled Delay Sync as hdlsrc_sync\h
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line/circular buffer logi
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line/SimpleDualPortRAM_ge
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line as hdlsrc_sync\hdlco
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync as hdlsrc_sync\hdlcoder_statecontro
### Generating package file hdlsrc_sync\hdlcoder_statecontrol_model\MAC_FIR_Block_Sync_pkg.vhd.
### Code Generation for 'hdlcoder_statecontrol_model' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_statecontrol_model' complete with 0 errors, 1 warnings, and 1 message
### HDL code generation complete.
```

```
type hdlsrc_sync/hdlcoder_statecontrol_model/Enabled_Delay_Sync.vhd
```

```
-- -------------------------------------------------------------
-- 
-- File Name: hdlsrc_sync\hdlcoder_statecontrol_model\Enabled_Delay_Sync.vhd
-- Created: 2024-02-13 21:39:06
-- 
-- Generated by MATLAB 24.1, HDL Coder 24.1, and Simulink 24.1
-- 
-- -------------------------------------------------------------


-- -------------------------------------------------------------
-- 
-- Module: Enabled_Delay_Sync
-- Source Path: hdlcoder_statecontrol_model/MAC FIR Block Sync/Enabled Delay Sync
-- Hierarchy Level: 1
-- Model version: 15.0
-- 
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Enabled_Delay_Sync IS
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        enb                               :   IN    std_logic;
        din_re                            :   IN    std_logic_vector(33 DOWNTO 0);  -- sfix34_En3
        din_im                            :   IN    std_logic_vector(33 DOWNTO 0);  -- sfix34_En3
        LocalEnable                       :   IN    std_logic;
        Out1_re                           :   OUT   std_logic_vector(33 DOWNTO 0);  -- sfix34_En3
        Out1_im                           :   OUT   std_logic_vector(33 DOWNTO 0)  -- sfix34_En31
        );
END Enabled_Delay_Sync;


ARCHITECTURE rtl OF Enabled_Delay_Sync IS

  -- Signals
  SIGNAL din_re_signed                    : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL din_im_signed                    : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL dout_re                          : signed(33 DOWNTO 0);  -- sfix34_En31
  SIGNAL dout_im                          : signed(33 DOWNTO 0);  -- sfix34_En31

BEGIN
  din_re_signed <= signed(din_re);

  din_im_signed <= signed(din_im);

  Enabled_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      dout_re <= to_signed(0, 34);
      dout_im <= to_signed(0, 34);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' AND LocalEnable = '1' THEN
        dout_re <= din_re_signed;
        dout_im <= din_im_signed;
```

```
      END IF;
    END IF;
  END PROCESS Enabled_Delay_process;


  Out1_re <= std_logic_vector(dout_re);

  Out1_im <= std_logic_vector(dout_im);

END rtl;
```

Enabled Subsystems

When a model has an Enabled subsystem in Synchronous mode, the code generated for it will also be improved. A Synchronous mode Enabled subsystem will no longer generate bypass registers on the subsystem outputs. In addition, any registers inside the Enabled subsystem that have explicit enable inputs will also show the same improvements as discussed previously.

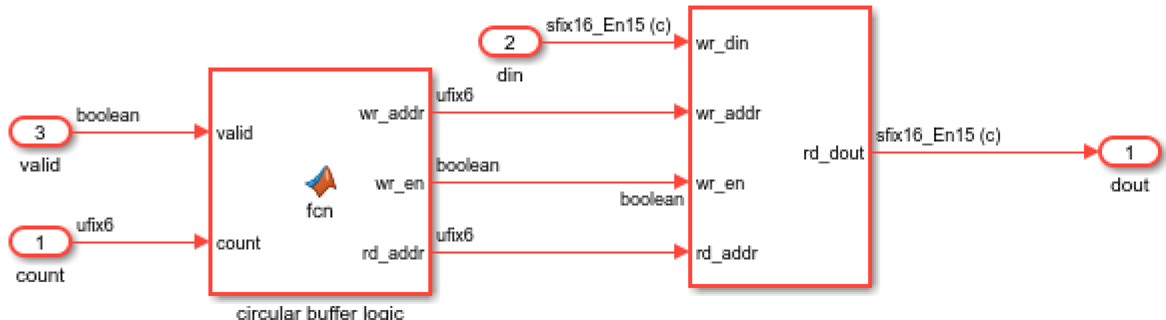
MATLAB Function Blocks and Synchronous Mode

MATLAB Function Blocks require more precise configuration in order to be used in Synchronous mode. If the block contains a direct combinatorial path from block input to output, you must enable direct feedthrough:

1. Open the Property Inspector. In the **Modeling** tab, in the **Design** section, click the **Property Inspector**.
2. Click the MATLAB Function block.
3. Expand the **Advanced** section. Select **Allow direct feedthrough** for each block containing a combinatorial output path.

This setting allows the code to be generated in Synchronous mode from a MATLAB Function block, when that block has both combinatorial and sequential paths in its code.

```
open_system('hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line')
%   open_system('hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line/circular buffer lo
```



The RAM based delay line has 2 latency:
- 1 from count to rd_addr
- 1 from rd_addr to rd_dout

For additional information

To learn more about the State Control block, see State Control.

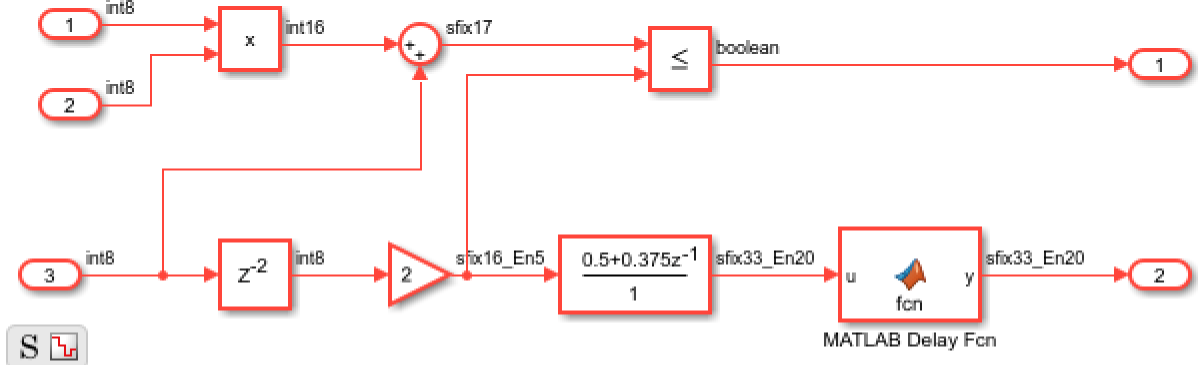
Resettable Subsystem Support in HDL Coder

This example shows how to use Resettable Subsystems in HDL Coder™.

Introduction to Resettable Subsystems

A Resettable Subsystem is a subsystem that will reset all states within the subsystem hierarchy based on a boolean control signal. It does this without requiring wiring the reset signal to each stateful block in Simulink®. This feature allows resetting blocks such as the MATLAB Function Block, which does not have an available reset port. For support in HDL Coder, a Resettable Subsystem is supported only within a Synchronous State Control region.

```
load_system('hdlcoder_resettable_subsystem');
open_system('hdlcoder_resettable_subsystem/DUT/Resettable Subsystem');
set_param('hdlcoder_resettable_subsystem', 'SimulationCommand', 'update');
```



The Reset Block

A Resettable Subsystem looks similar to an Enabled Subsystem or any other Simulink conditionally executed subsystem in that it has a specialized Reset Port block inside it. This control port block has several trigger types available. HDL Coder supports the "level hold" trigger type.

```
open_system('hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/LocalReset');
```

Resettable Subsystem Effects on Generated HDL code

Resettable Subsystems allow resetting the state of all blocks with state inside the subsystem to their initial value. In the generated HDL code, each design delay, a delay modeled explicitly in Simulink, has a reset added. Hardware implementation delays such as pipeline delays are not reset. The reset signal is a synchronous signal and is entirely independent from the global reset signal.

```
close_system('hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/LocalReset');
makehdl('hdlcoder_resettable_subsystem/DUT');
type hdlsrc/hdlcoder_resettable_subsystem/DUT.vhd
```

```
### Working on the model <a href="matlab:open_system('hdlcoder_resettable_subsystem')">hdlcoder_r
### Generating HDL for <a href="matlab:open_system('hdlcoder_resettable_subsystem/DUT')">hdlcoder
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_resetta
### Running HDL checks on the model 'hdlcoder_resettable_subsystem'.
### Begin compilation of the model 'hdlcoder_resettable_subsystem'...
### Working on the model 'hdlcoder_resettable_subsystem'...
### Working on... <a href="matlab:configset.internal.open('hdlcoder_resettable_subsystem', 'Gener
### Begin model generation 'gm_hdlcoder_resettable_subsystem'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoder_resettable_subsystem\gm
### Begin VHDL Code Generation for 'hdlcoder_resettable_subsystem'.
### Working on hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/Discrete FIR Filter as hdls
### Working on hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/MATLAB Delay Fcn as hdlsrc\
### Working on hdlcoder_resettable_subsystem/DUT/Resettable Subsystem as hdlsrc\hdlcoder_resettab
### Working on hdlcoder_resettable_subsystem/DUT as hdlsrc\hdlcoder_resettable_subsystem\DUT.vhd
### Generating package file hdlsrc\hdlcoder_resettable_subsystem\DUT_pkg.vhd.
### Code Generation for 'hdlcoder_resettable_subsystem' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc24a_2528353_7604/ib462BFE/29/tp
### HDL check for 'hdlcoder_resettable_subsystem' complete with 0 errors, 0 warnings, and 0 messa
### HDL code generation complete.

-- -------------------------------------------------------------
--
-- File Name: hdlsrc\hdlcoder_resettable_subsystem\DUT.vhd
-- Created: 2024-02-13 21:27:35
--
-- Generated by MATLAB 24.1, HDL Coder 24.1, and Simulink 24.1
--
--
-- -------------------------------------------------------------
-- Rate and Clocking Details
-- -------------------------------------------------------------
-- Model base rate: 1
-- Target subsystem base rate: 1
--
--
-- Clock Enable  Sample Time
-- -------------------------------------------------------------
-- ce_out        1
-- -------------------------------------------------------------
--
--
-- Output Signal                 Clock Enable  Sample Time
-- -------------------------------------------------------------
-- Out1                          ce_out        1
-- Out2                          ce_out        1
-- -------------------------------------------------------------
--
-- -------------------------------------------------------------



-- -------------------------------------------------------------
--
-- Module: DUT
-- Source Path: hdlcoder_resettable_subsystem/DUT
-- Hierarchy Level: 0
```

```
-- Model version: 16.0
--
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY DUT IS
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        clk_enable                        :   IN    std_logic;
        LocalReset                        :   IN    std_logic;
        In2                               :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
        In3                               :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
        In4                               :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
        ce_out                            :   OUT   std_logic;
        Out1                              :   OUT   std_logic;
        Out2                              :   OUT   std_logic_vector(32 DOWNTO 0)  -- sfix33_En2
        );
END DUT;


ARCHITECTURE rtl OF DUT IS

  -- Component Declarations
  COMPONENT Resettable_Subsystem
    PORT( clk                             :   IN    std_logic;
          reset                           :   IN    std_logic;
          enb                             :   IN    std_logic;
          In1                             :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
          In2                             :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
          In3                             :   IN    std_logic_vector(7 DOWNTO 0);  -- int8
          LocalReset                      :   IN    std_logic;
          Out1                            :   OUT   std_logic;
          Out2                            :   OUT   std_logic_vector(32 DOWNTO 0)  -- sfix33_En2
          );
  END COMPONENT;

  -- Component Configuration Statements
  FOR ALL : Resettable_Subsystem
    USE ENTITY work.Resettable_Subsystem(rtl);

  -- Signals
  SIGNAL Resettable_Subsystem_out1        : std_logic;
  SIGNAL Resettable_Subsystem_out2        : std_logic_vector(32 DOWNTO 0);  -- ufix33

BEGIN
  u_Resettable_Subsystem : Resettable_Subsystem
    PORT MAP( clk => clk,
              reset => reset,
              enb => clk_enable,
              In1 => In2,  -- int8
              In2 => In3,  -- int8
              In3 => In4,  -- int8
              LocalReset => LocalReset,
              Out1 => Resettable_Subsystem_out1,
              Out2 => Resettable_Subsystem_out2  -- sfix33_En20
              );
```

```
  ce_out <= clk_enable;

  Out1 <= Resettable_Subsystem_out1;

  Out2 <= Resettable_Subsystem_out2;

END rtl;
```

The MATLAB Function Block does not have support for an explicit reset port. When placed in a Resettable Subsystem, HDL Coder will generate a synchronous external reset signal to control the resetting of persistent variables inside the function.

```
  function y = fcn(u)

  persistent state;

  if isempty(state)
      state = fi(0, 1, 33, 20);
  end

  y = state;
  state = u;
  end
```

```
type hdlsrc/hdlcoder_resettable_subsystem/MATLAB_Delay_Fcn.vhd
```

```
-- -------------------------------------------------------------
--
-- File Name: hdlsrc\hdlcoder_resettable_subsystem\MATLAB_Delay_Fcn.vhd
-- Created: 2024-02-13 21:27:35
--
-- Generated by MATLAB 24.1, HDL Coder 24.1, and Simulink 24.1
--
-- -------------------------------------------------------------


-- -------------------------------------------------------------
--
-- Module: MATLAB_Delay_Fcn
-- Source Path: hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/MATLAB Delay Fcn
-- Hierarchy Level: 2
-- Model version: 16.0
--
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY MATLAB_Delay_Fcn IS
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        enb                               :   IN    std_logic;
        u                                 :   IN    std_logic_vector(32 DOWNTO 0);  -- sfix33_En2
        LocalReset                        :   IN    std_logic;
        y                                 :   OUT   std_logic_vector(32 DOWNTO 0)  -- sfix33_En2(
        );
```

```
END MATLAB_Delay_Fcn;


ARCHITECTURE rtl OF MATLAB_Delay_Fcn IS

  -- Signals
  SIGNAL u_signed                         : signed(32 DOWNTO 0);  -- sfix33_En20
  SIGNAL y_tmp                            : signed(32 DOWNTO 0);  -- sfix33_En20

BEGIN
  u_signed <= signed(u);

  MATLAB_Delay_Fcn_1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      y_tmp <= to_signed(0, 33);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        IF LocalReset = '1' THEN
          y_tmp <= to_signed(0, 33);
        ELSE
          y_tmp <= u_signed;
        END IF;
      END IF;
    END IF;
  END PROCESS MATLAB_Delay_Fcn_1_process;


  y <= std_logic_vector(y_tmp);

END rtl;
```

A synchronous delay signal named `LocalDelay` has been added to the VHDL code generated for the delay implemented in the MATLAB Function block.

Stateflow HDL Code Generation Support

- “Introduction to Stateflow HDL Code Generation” on page 26-2
- “Hardware Realization of Stateflow Semantics” on page 26-6
- “Generate HDL for Mealy and Moore Finite State Machines” on page 26-7
- “Design Patterns Using Advanced Chart Features” on page 26-14
- “Initialize Persistent Variables in MATLAB Functions” on page 26-22

Introduction to Stateflow HDL Code Generation

In this section...

"Example" on page 26-2
"Chart Initialization" on page 26-2
"Tunable Parameters" on page 26-3
"Comments in Stateflow Charts" on page 26-3
"Restrictions" on page 26-3

Stateflow charts provide concise descriptions of complex system behavior by using hierarchical finite state machine (FSM) theory, flow diagram notation, and state-transition diagrams.

You use a chart to model a finite state machine or a complex control algorithm intended for realization as an ASIC or FPGA. When the model meets design requirements, you then generate HDL code (VHDL, Verilog or SystemVerilog) that implements the design embodied in the model. You can simulate and synthesize generated HDL code by using industry standard tools, and then map your system designs into FPGAs and ASICs. For more information on how to generate HDL code for finite state machines, see "Generate HDL for Mealy and Moore Finite State Machines" on page 26-7.

Generation of VHDL, Verilog or SystemVerilog code from a model containing a chart does not differ greatly from HDL code generation from other models. The HDL code generator is designed to:

- Support the largest possible subset of chart semantics consistent with HDL code. This broad subset means that you can generate HDL code from existing models without significant remodeling.
- Generate bit-true, cycle-accurate HDL code that is fully compatible with Stateflow simulation semantics.

Example

Using the hdlcodercfir model, the example shows how to generate HDL code for a subsystem that includes Stateflow charts.

To open the model, at the command line, enter:

```
hdlcodercfir
```

Chart Initialization

Enabling the **Execute (enter) Chart at Initialization** property executes the update chart function immediately following chart initialization. You can keep the chart property **Execute (enter) Chart at Initialization** disabled. This property can significantly alter the Stateflow Chart behavior and the generated HDL Code. "Execution of a Chart at Initialization" (Stateflow) describes the effect of this property on a Stateflow Chart.

You cannot perform arithmetic in initialization actions because reset actions cannot handle the delay of combinatorial logic.

You can select or clear **Initialize Outputs Every Time Chart Wakes Up**. If you clear it, the generated HDL code includes an additional register for the state machine output values.

Tunable Parameters

You can use a tunable parameter in a Stateflow Chart intended for HDL code generation.

For more information, see “Generate DUT Ports for Tunable Parameters” on page 14-18.

Comments in Stateflow Charts

When your Simulink model contains a Stateflow Chart that uses comments, HDL Coder generates the comments in the HDL code.

When you generate Verilog code from the model, HDL Coder displays the comments in the Stateflow Chart inline beside the corresponding Stateflow object.

Restrictions

HDL Coder does not support Stateflow blocks that contain messages for HDL code generation.

Location of Charts in the Model

A chart intended for HDL code generation must be part of a Simulink subsystem. If the chart for which you want to generate code is at the root level of your model, embed the chart in a subsystem. Connect the relevant signals to the subsystem inputs and outputs.

Data Types

The code generator supports a subset of MATLAB data types in charts that include:

- Signed and unsigned integer
- Fixed point
- Boolean
- Enumeration

Note Except for data types assigned to ports, multidimensional arrays of these types are supported. Port data types must be either scalar or vector.

If you use single and double data types, HDL Coder generates real data types in the HDL code. You can simulate and verify the code by using third-party simulators such as ModelSim.

Real types are not synthesizable on the target FPGA device. The code generator does not support generation of HDL code for the Stateflow Chart in `Native Floating Point` mode. To generate synthesizable HDL code when you use floating-point data types, develop an algorithm by using MATLAB Function on page 14-114 blocks or other “Simulink Blocks Supported by Using Native Floating Point” on page 14-137.

Imported Code

A chart intended for HDL code generation must be entirely self-contained. These restrictions apply:

- Do not call MATLAB functions other than `min` or `max`.

- Do not use MATLAB System objects in a Chart block.
- Do not use MATLAB workspace data.
- Do not call C math functions. HDL does not have a counterpart to the C math library.
- If the **Enable C-bit operations** property is disabled, do not use the exponentiation operator (`^`). The exponentiation operator is implemented with the C Math Library function `pow`.
- Do not include custom code. Information entered on the **Simulation Target > Custom Code** pane in the Configuration Parameters dialog box is ignored.
- Do not share data (through Data Store Memory blocks) between charts. HDL Coder does not map such global data to HDL because HDL does not support global data.

Vector of Tunable Parameters

Vector of Tunable Parameters as data types for Chart blocks is not supported.

Input and Output Events

HDL Coder supports the use of input and output events with Stateflow charts, subject to these constraints:

- You can define and use only one input event per Stateflow chart. There is no restriction on the number of output events that you can use.
- The coder does not support HDL code generation for charts that have a single input event, and which also have nonzero initial values on the chart's output ports.
- All input and output events must be edge-triggered.

For detailed information on input and output events, see “Activate a Stateflow Chart by Sending Input Events” (Stateflow) and “Activate a Simulink Block by Sending Output Events” (Stateflow).

Messages

Stateflow messages are not supported for HDL code generation.

Loops

Other than `for` loops, do not explicitly use loops in a chart intended for HDL code generation. Observe the following restrictions on `for` loops:

- The data type of the loop counter variable must be `int32`.
- HDL Coder supports only constant-bounded loops.

The `for` loop example, `sf_for`, shows a design pattern for a `for` loop that uses a graphical function.

Additional Restrictions

HDL Coder imposes additional restrictions on the use of classic chart features. These limitations exist because HDL does not support some features of general-purpose sequential programming languages.

- Make separate copies for each instance of each atomic subchart. HDL Coder does not support code generation for atomic subcharts. For more information, see “Convert an Atomic Subchart to a Normal Subchart” (Stateflow).
- Do not generate HDL code for Simulink Function block.

- Do not define local events in a chart from which HDL code is generated.

  Do not use these implicit events:

  - `enter`
  - `exit`
  - `change`

  You can use these implicit events:

  - `wakeup`
  - `tick`

  If the base events are limited to these types of implicit events, you can use temporal logic.
- Do not use recursion through graphical functions. HDL Coder does not support recursion.
- Avoid unstructured code. Although charts allow unstructured code through transition flow diagrams and graphical functions, this usage results in `goto` statements and multiple function return statements. HDL does not support either `goto` statements or multiple function return statements. Therefore, do not use unstructured flow diagrams.
- If you have not selected the **Initialize Outputs Every Time Chart Wakes Up** chart option, do not read from output ports.
- Do not use Data Store Memory objects.
- Do not use pointer (&) or indirection (*) operators. See “Pointer and Address Operations” (Stateflow).
- If a chart gets a run-time overflow error during simulation, it is possible to disable data range error checking and generate HDL code for the chart. In such cases, some results obtained from the generated HDL code might not be bit-true to results from the simulation. The recommended practice is to enable overflow checking and eliminate overflow conditions from the model during simulation.

See Also

State Transition Table | Truth Table | Sequence Viewer

Related Examples

- “Generate HDL for Mealy and Moore Finite State Machines” on page 26-7
- “Design Patterns Using Advanced Chart Features” on page 26-14

More About

- “Hardware Realization of Stateflow Semantics” on page 26-6

Hardware Realization of Stateflow Semantics

A mapping from Stateflow semantics to an HDL implementation has the following requirements:

- **Requirement 1**: Hardware designs require separability of output and state update functions.
- **Requirement 2**: HDL is a concurrent language. To achieve the goal of bit-true simulation, execution must be in order.

To meet Requirement 1, an FSM is coded in HDL as two concurrent blocks that execute under different conditions. One block evaluates the transition conditions, computes outputs and computes the next state variables. The other block updates the current state variables from the available next state and performs the actual state transitions. This second block is activated only on the trigger edge of the clock signal, or an asynchronous reset signal.

Stateflow sequential semantics map to HDL sequential statements, and local chart variables in function scope map to VHDL variables in process scope. In VHDL, variable assignment is sequential. Therefore, statements in a Stateflow function that uses local variables can map to statements in a VHDL process that uses corresponding variables. The VHDL assignments execute in the same order as the assignments in the Stateflow function.

See Also

State Transition Table | Truth Table | Sequence Viewer

Related Examples

- “Generate HDL for Mealy and Moore Finite State Machines” on page 26-7
- “Design Patterns Using Advanced Chart Features” on page 26-14

More About

- “Introduction to Stateflow HDL Code Generation” on page 26-2

Generate HDL for Mealy and Moore Finite State Machines

In this section...

"Generate HDL Code for Moore Finite State Machine" on page 26-7
"Generate HDL for Mealy Finite State Machine" on page 26-8
"Initialize Outputs Every Time Chart Wakes Up" on page 26-10

Stateflow charts support modeling of three types of state machines:

- Classic (default)
- Mealy
- Moore

For HDL code generation, use Mealy or Moore type state machines. Mealy and Moore state machines differ in these ways.

- The output of a Mealy state machine is a function of the current state and inputs.
- The output of a Moore state machine is a function of the current state only.

The principal advantages of using Mealy or Moore charts as an alternative to Classic charts are:

- Moore charts can generate more efficient code than Classic charts.
- At compile time, Mealy and Moore charts are validated for conformance to their formal definitions and semantic rules, and violations are reported.

To learn more about HDL code generation guidelines for charts, see the Chart block.

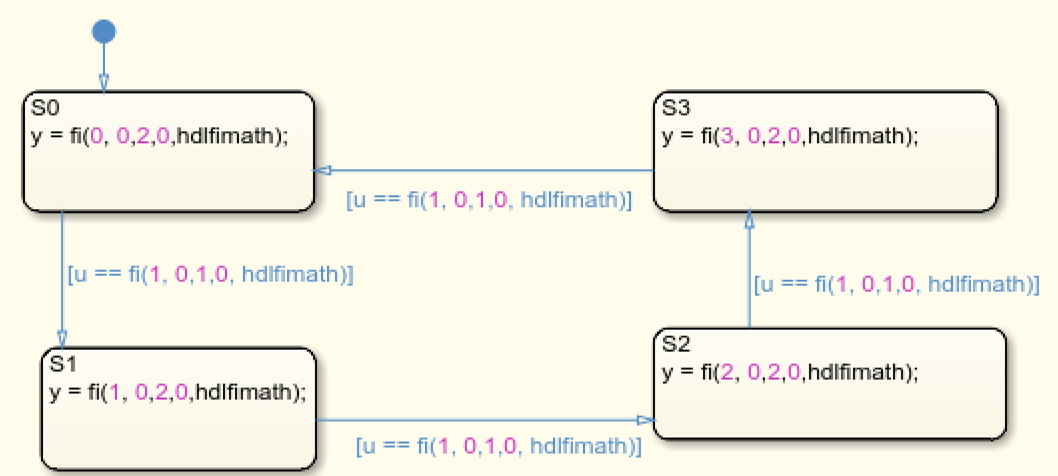
Generate HDL Code for Moore Finite State Machine

This example shows Stateflow chart of a Moore state machine that uses MATLAB® as the action language.

Open model

Load `hdlcoder_fsm_mealy_moore` model. The `Moore` subsystem in `hdlcoder_fsm_mealy_moore` contains the Stateflow chart `Moore_Chart` that models a Moore state machine. To open the `Moore_Chart`, run these commands:

```
load_system('hdlcoder_fsm_mealy_moore.slx');
open_system('hdlcoder_fsm_mealy_moore/Moore/Moore_Chart');
```



When generating HDL code for a chart that models a Moore state machine, these conditions apply.

- The chart must meet the general code generation requirements as described in the Chart block.
- Actions must occur in states only. These actions must be unlabeled. Moore actions must be associated with states, because the output computation is dependent only on states, not input. The configuration of active states at time step `t` determines the output. If state `S` is active when a chart wakes up at time `t`, then state `S` contributes to the output whether or not the state remains active into time `t+1`.
- Do not call Simulink® functions. This restriction prevents the output from depending on the input in ways that are difficult for the HDL code generator to verify.
- If you disable the **Initialize Outputs Every Time Chart Wakes Up** parameter, the generated HDL code includes additional registers of the state machine output values.
- You can enable clock-driven outputs for a Moore chart by setting the Stateflow chart HDL block property **ClockDrivenOutput** to `on`. For more information, see “Guidelines for HDL Code Generation Using Stateflow Charts” on page 18-123.

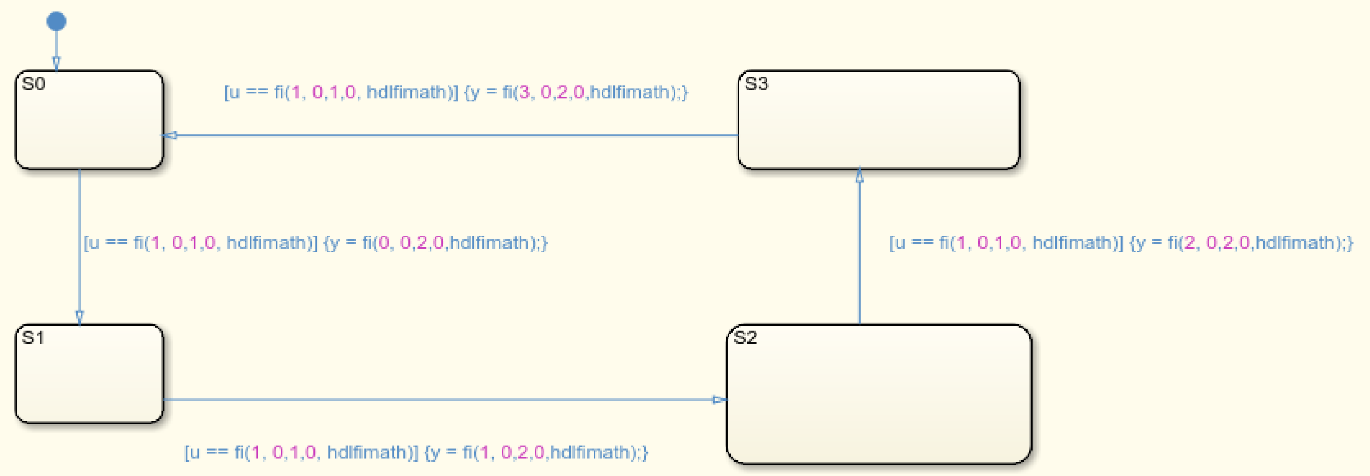
Generate HDL for Mealy Finite State Machine

This example shows Stateflow chart that models a Mealy state machine using MATLAB as the action language. You can also generate the HDL code for Stateflow using HDL Coder™.

Open model

Load `hdlcoder_fsm_mealy_moore` model. The `Mealy` subsystem in `hdlcoder_fsm_mealy_moore` contains the Stateflow chart `Mealy_Chart` that model a Mealy state machine. To open the `Mealy_Chart`, run these commands:

```
load_system('hdlcoder_fsm_mealy_moore.slx');
open_system('hdlcoder_fsm_mealy_moore/Mealy/Mealy_Chart');
```



When generating HDL code for a chart that models a Mealy state machine, these conditions apply.

- The chart must meet the general code generation requirements as described in the Chart block.
- Actions must be associated with inner and outer transitions only.
- If you disable the **Initialize Outputs Every Time Chart Wakes Up** parameter, the generated HDL code includes additional registers of the state machine output values.

Mealy actions are associated with transitions. In Mealy machines, the output computation is driven by a change of input values. The dependence of the output on the input is the fundamental distinguishing factor between the formal definitions of Mealy and Moore machines. The requirement that actions result from transitions is to some degree stylistic, rather than required, to enforce Mealy semantics. Because transition conditions are primarily input conditions in any machine type, the output computation ultimately follows input conditions in either type.

Generated HDL Code of the Mealy Chart

This code is the Verilog® code generated for the Mealy chart.

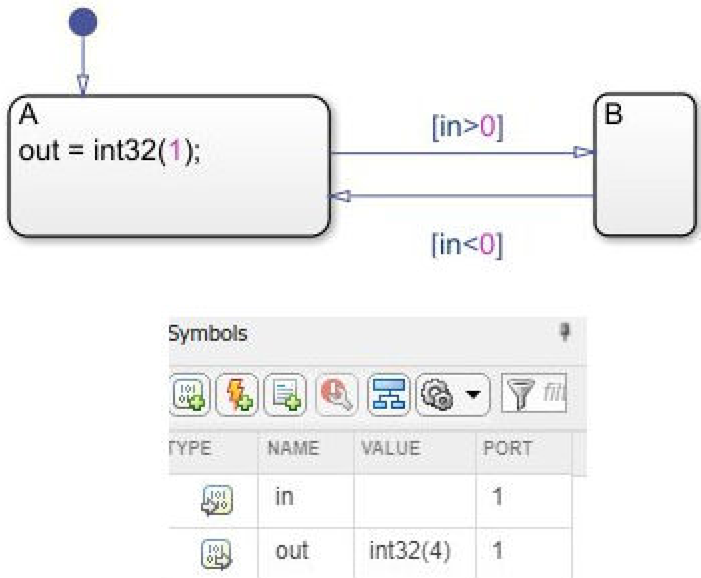
```
always @(posedge clk or posedge reset)
  begin : Mealy_Chart_1_process
    if (reset == 1'b1) begin
      is_Mealy_Chart <= is_Mealy_Chart_IN_S0;
    end
    else begin
      if (enb) begin
        is_Mealy_Chart <= is_Mealy_Chart_next;
      end
    end
  end

always @(is_Mealy_Chart, u) begin
  is_Mealy_Chart_next = is_Mealy_Chart;
  y_1 = 2'b00;
  case ( is_Mealy_Chart)
    is_Mealy_Chart_IN_S0 :
      begin
        if (u == 8'sb00000001) begin
```

```
          y_1 = 2'b00;
          is_Mealy_Chart_next = is_Mealy_Chart_IN_S1;
        end
      end
    is_Mealy_Chart_IN_S1 :
      begin
        if (u == 8'sb00000001) begin
          y_1 = 2'b01;
          is_Mealy_Chart_next = is_Mealy_Chart_IN_S2;
        end
      end
    is_Mealy_Chart_IN_S2 :
      begin
        if (u == 8'sb00000001) begin
          y_1 = 2'b10;
          is_Mealy_Chart_next = is_Mealy_Chart_IN_S3;
        end
      end
    default :
      begin
        if (u == 8'sb00000001) begin
          y_1 = 2'b11;
          is_Mealy_Chart_next = is_Mealy_Chart_IN_S0;
        end
      end
  endcase
end

assign y = y_1;
```

Initialize Outputs Every Time Chart Wakes Up

Mealy and Moore charts have an option to return the output signals to their initial values when the signals are not driven by a state or transition. Select the **Initialize Outputs Every Time Chart Wakes Up** parameter to enable this behavior. If you clear this parameter, the generated HDL code includes additional registers to store the state machine output values.

This figure shows a simple Moore chart with two states and one output. The output is set to 1 in state A, and the initial value of the output is 4.



When you select the **Initialize Outputs Every Time Chart Wakes Up** parameter, the output value returns to 4 unless the state machine is in state A. State A sets the output to 1. When you clear the **Initialize Outputs Every Time Chart Wakes Up** parameter, the output value remains at 1 after the machine passes through state A, and does not return to 4.

This figure shows the Verilog code generated for this Moore chart, with **Initialize Outputs Every Time Chart Wakes Up** selected and then with it cleared.

☑ Initialize outputs every time chart wakes up

```
always @(posedge clk or posedge reset)
  begin : MooreChart_1_process
    if (reset == 1'b1) begin
      is_MooreChart <= is_MooreChart_IN_A;
    end
    else begin
      if (enb) begin
        is_MooreChart <= is_MooreChart_next;
      end
    end
  end

always @(in, is_MooreChart) begin
  is_MooreChart_next = is_MooreChart;
  case ( is_MooreChart)
    is_MooreChart_IN_A :
      begin
        if (in > 32'sb00000000000000000000000000000000) begin
          is_MooreChart_next = is_MooreChart_IN_B;
        end
      end
    default :
      begin
        //case IN_B:
        if (in < 32'sb00000000000000000000000000000000) begin
          is_MooreChart_next = is_MooreChart_IN_A;
        end
      end
  endcase
end

always @(is_MooreChart) begin
  out_1 = 32'sd4;
  case ( is_MooreChart)
    is_MooreChart_IN_A :
      begin
        out_1 = 32'sd1;
      end
    default :
      begin
        //case IN_B:
      end
  endcase
end
```

☐ Initialize outputs every time chart wakes up

```
always @(posedge clk or posedge reset)
  begin : MooreChart_1_process
    if (reset == 1'b1) begin
      out_reg <= 32'sd4;
      is_MooreChart <= is_MooreChart_IN_A;
    end
    else begin
      if (enb) begin
        is_MooreChart <= is_MooreChart_next;
        out_reg <= out_reg_next;
      end
    end
  end

always @(in, is_MooreChart) begin
  is_MooreChart_next = is_MooreChart;
  case ( is_MooreChart)
    is_MooreChart_IN_A :
      begin
        if (in > 32'sb00000000000000000000000000000000) begin
          is_MooreChart_next = is_MooreChart_IN_B;
        end
      end
    default :
      begin
        //case IN_B:
        if (in < 32'sb00000000000000000000000000000000) begin
          is_MooreChart_next = is_MooreChart_IN_A;
        end
      end
  endcase
end

always @(is_MooreChart, out_reg) begin
  out_reg_next = out_reg;
  case ( is_MooreChart)
    is_MooreChart_IN_A :
      begin
        out_reg_next = 32'sd1;
      end
    default :
      begin
        //case IN_B:
      end
  endcase
end

assign out = out_reg_next;
```

This table shows the resource usage of these two Moore machines synthesized for a Xilinx Vivado Virtex 7 device. When you clear **Initialize Outputs Every Time Chart Wakes Up**, the generated HDL code includes additional registers for the output state.

Resource	Initialize Outputs Every Time Chart Wakes UpSelected	Initialize Outputs Every Time Chart Wakes UpCleared
LUTs	18	20
Registers	1	3
DSPs	0	0
Block RAM	0	0

See Also

Chart

More About

- "Design Patterns Using Advanced Chart Features" on page 26-14
- "Introduction to Stateflow HDL Code Generation" on page 26-2
- "Hardware Realization of Stateflow Semantics" on page 26-6

Design Patterns Using Advanced Chart Features

In this section...

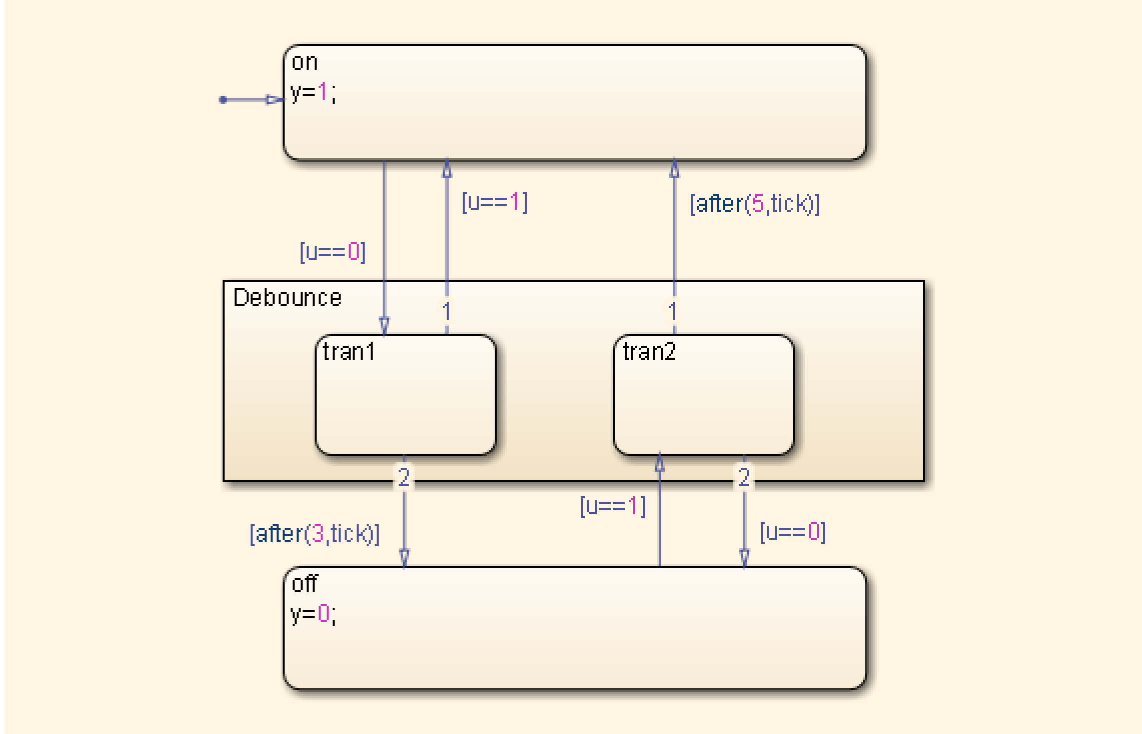
"Temporal Logic" on page 26-14
"Graphical Function" on page 26-15
"Hierarchy and Parallelism" on page 26-16
"Stateless Charts" on page 26-17
"Truth Tables" on page 26-18

Temporal Logic

Stateflow temporal logic operators (such as `after`, `before`, or `every`) are Boolean operators that operate on recurrence counts of Stateflow events. Temporal logic operators can appear only in conditions on transitions that originate from states, and in state actions. Although temporal logic does not introduce new events into a Stateflow model, it is useful to think of the change of value of a temporal logic condition as an event. You can use temporal logic operators in many cases where a counter is required. A common use case would be to use temporal logic to implement a time-out counter.

For detailed information, see "Control Chart Execution by Using Temporal Logic" (Stateflow).

The chart shown in the following figure uses temporal logic in a design for a debouncer. Instead of instantaneously switching between on and off states, the chart uses two intermediate states and temporal logic to ignore transients. The transition is committed based on a time-out.



By default, states in a Stateflow Chart are ordered alphabetically. The ordering of states in the HDL code can potentially vary if you enable active state output port generation in the HDL code. To enable this setting, open the Chart properties and select the **Create output port for monitoring** check box. See also “Simplify Stateflow Charts by Incorporating Active State Output” (Stateflow).

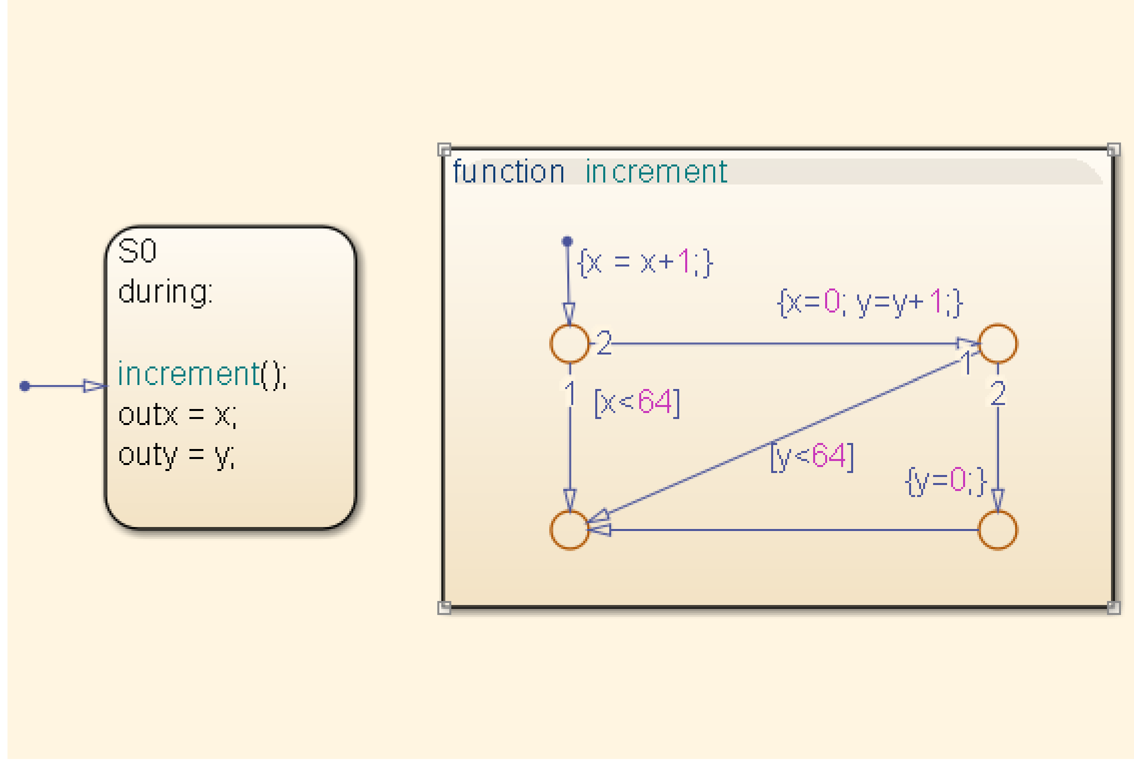
When you generate VHDL code, the recently added state is selected as the `OTHERS` state in the HDL code. The following code excerpt shows VHDL code generated from this Chart.

```
Chart_1_output : PROCESS (is_Chart, u, temporalCounter_i1, y_reg)
    VARIABLE temporalCounter_i1_temp : unsigned(7 DOWNTO 0);
  BEGIN
    temporalCounter_i1_temp := temporalCounter_i1;
    is_Chart_next <= is_Chart;
    y_reg_next <= y_reg;
    IF temporalCounter_i1 < 7 THEN
      temporalCounter_i1_temp := temporalCounter_i1 + 1;
    END IF;

    CASE is_Chart IS
      WHEN IN_tran1 =>
        IF u = 1.0 THEN
          is_Chart_next <= IN_on;
          y_reg_next <= 1.0;
        ELSIF temporalCounter_i1_temp >= 3 THEN
          is_Chart_next <= IN_off;
          y_reg_next <= 0.0;
        END IF;
      WHEN IN_tran2 =>
        IF temporalCounter_i1_temp >= 5 THEN
          is_Chart_next <= IN_on;
          y_reg_next <= 1.0;
        ELSIF u = 0.0 THEN
          is_Chart_next <= IN_off;
          y_reg_next <= 0.0;
        END IF;
      WHEN IN_off =>
        IF u = 1.0 THEN
          is_Chart_next <= IN_tran2;
          temporalCounter_i1_temp := to_unsigned(0, 8);
        END IF;
      WHEN OTHERS =>
        IF u = 0.0 THEN
          is_Chart_next <= IN_tran1;
          temporalCounter_i1_temp := to_unsigned(0, 8);
        END IF;
    END CASE;

    temporalCounter_i1_next <= temporalCounter_i1_temp;
  END PROCESS Chart_1_output;
```

Graphical Function

A graphical function is a function defined graphically by a flow diagram. Graphical functions reside in a chart along with the diagrams that invoke them. Like MATLAB functions and C functions, graphical functions can accept arguments and return results. Graphical functions can be invoked in transition and state actions.

The following figure shows a graphical function that implements a 64–by–64 counter.



The following code excerpt shows VHDL code generated for this graphical function.

```
 x64_counter_sf : PROCESS (x, y, outx_reg, outy_reg)
    -- local variables
    VARIABLE x_temp : unsigned(7 DOWNTO 0);
    VARIABLE y_temp : unsigned(7 DOWNTO 0);
BEGIN
    outx_reg_next <= outx_reg;
    outy_reg_next <= outy_reg;
    x_temp := x;
    y_temp := y;
    x_temp := tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(x_temp, 9), 10)
            + tmw_to_unsigned(to_unsigned(1, 9), 10), 8);

    IF x_temp < to_unsigned(64, 8) THEN
        NULL;
    ELSE
        x_temp := to_unsigned(0, 8);
        y_temp := tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(y_temp, 9), 10)
              + tmw_to_unsigned(to_unsigned(1, 9), 10), 8);

        IF y_temp < to_unsigned(64, 8) THEN
            NULL;
        ELSE
            y_temp := to_unsigned(0, 8);
        END IF;

    END IF;

    outx_reg_next <= x_temp;
    outy_reg_next <= y_temp;
    x_next <= x_temp;
    y_next <= y_temp;
END PROCESS x64_counter_sf;
```

Hierarchy and Parallelism

Stateflow charts support both hierarchy (states containing other states) and parallelism (multiple states that can be active simultaneously).

In Stateflow semantics, parallelism is not synonymous with concurrency. Parallel states can be active simultaneously, but they are executed sequentially according to their execution order. (Execution order is displayed on the upper right corner of a parallel state).

For detailed information on hierarchy and parallelism, see “Overview of Stateflow Objects” (Stateflow) and “Execution Order for Parallel States” (Stateflow).

For HDL code generation, an entire chart maps to a single output computation process. Within the output computation process:

- The execution of parallel states proceeds sequentially.
- Nested hierarchical states map to nested `CASE` statements in the generated HDL code.

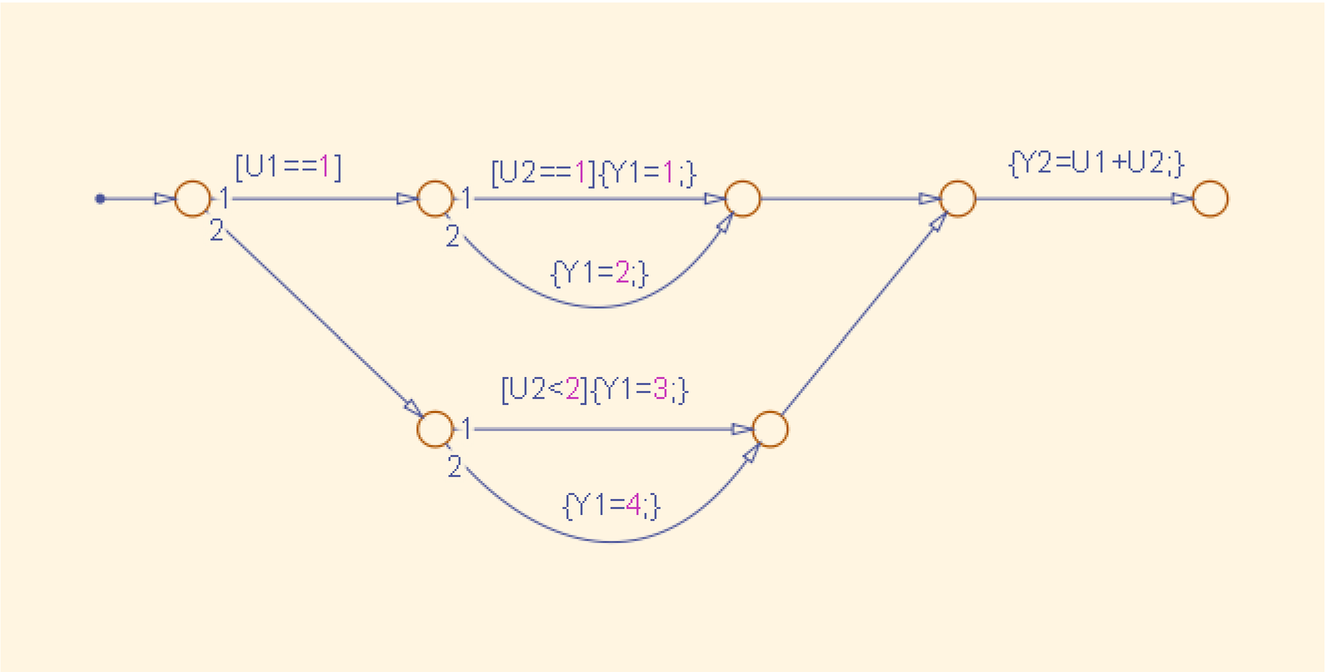
Stateless Charts

Charts consisting of pure flow diagrams (i.e., charts without states) are useful in capturing `if-then-else` constructs used in procedural languages like C.

As an example, consider the following logic, expressed in C-like pseudocode.

```
if(U1==1) {
    if(U2==1) {
       Y = 1;
    }else{
      Y = 2;
    }
}else{
   if(U2<2) {
      Y = 3;
   }else{
      Y = 4;
       }
      }
```

The following figure shows the flow diagram that implements the `if-then-else` logic.



The following generated VHDL code excerpt shows the nested IF-ELSE statements obtained from the flow diagram.

```
Chart : PROCESS (Y1_reg, Y2_reg, U1, U2)
        -- local variables
    BEGIN
        Y1_reg_next <= Y1_reg;
        Y2_reg_next <= Y2_reg;

        IF unsigned(U1) = to_unsigned(1, 8) THEN

            IF unsigned(U2) = to_unsigned(1, 8) THEN
                Y1_reg_next <= to_unsigned(1, 8);
            ELSE
                Y1_reg_next <= to_unsigned(2, 8);
            END IF;

        ELSIF unsigned(U2) < to_unsigned(2, 8) THEN
            Y1_reg_next <= to_unsigned(3, 8);
        ELSE
            Y1_reg_next <= to_unsigned(4, 8);
        END IF;

        Y2_reg_next <= tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(unsigned(U1), 9),10)
        + tmw_to_unsigned(tmw_to_unsigned(unsigned(U2), 9), 10), 8);
    END PROCESS Chart;
```

Truth Tables

HDL Coder supports HDL code generation for:

- Truth Table functions within a Stateflow chart
- Truth Table blocks in Simulink models

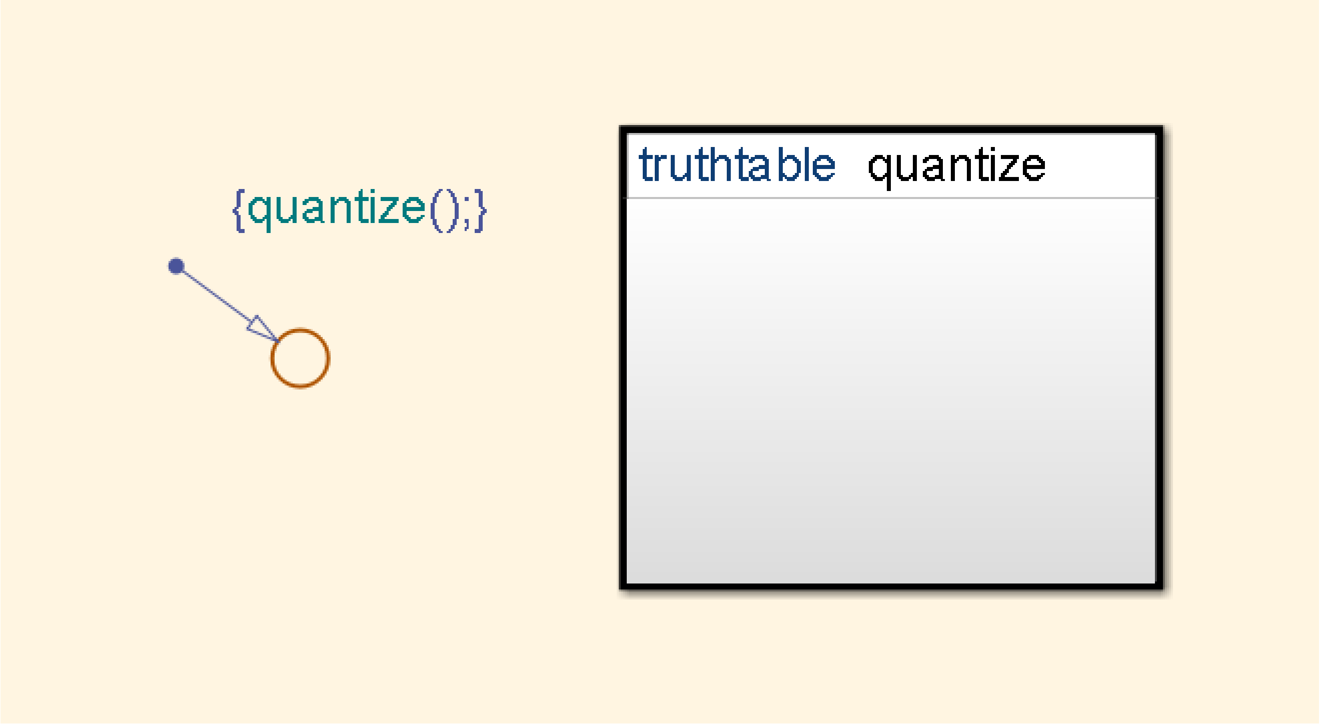
This section examines a Truth Table function in a chart, and the VHDL code generated for the chart.

Truth Tables are well-suited for implementing compact combinatorial logic. A typical application for Truth Tables is to implement nonlinear quantization or threshold logic. Consider the following logic:

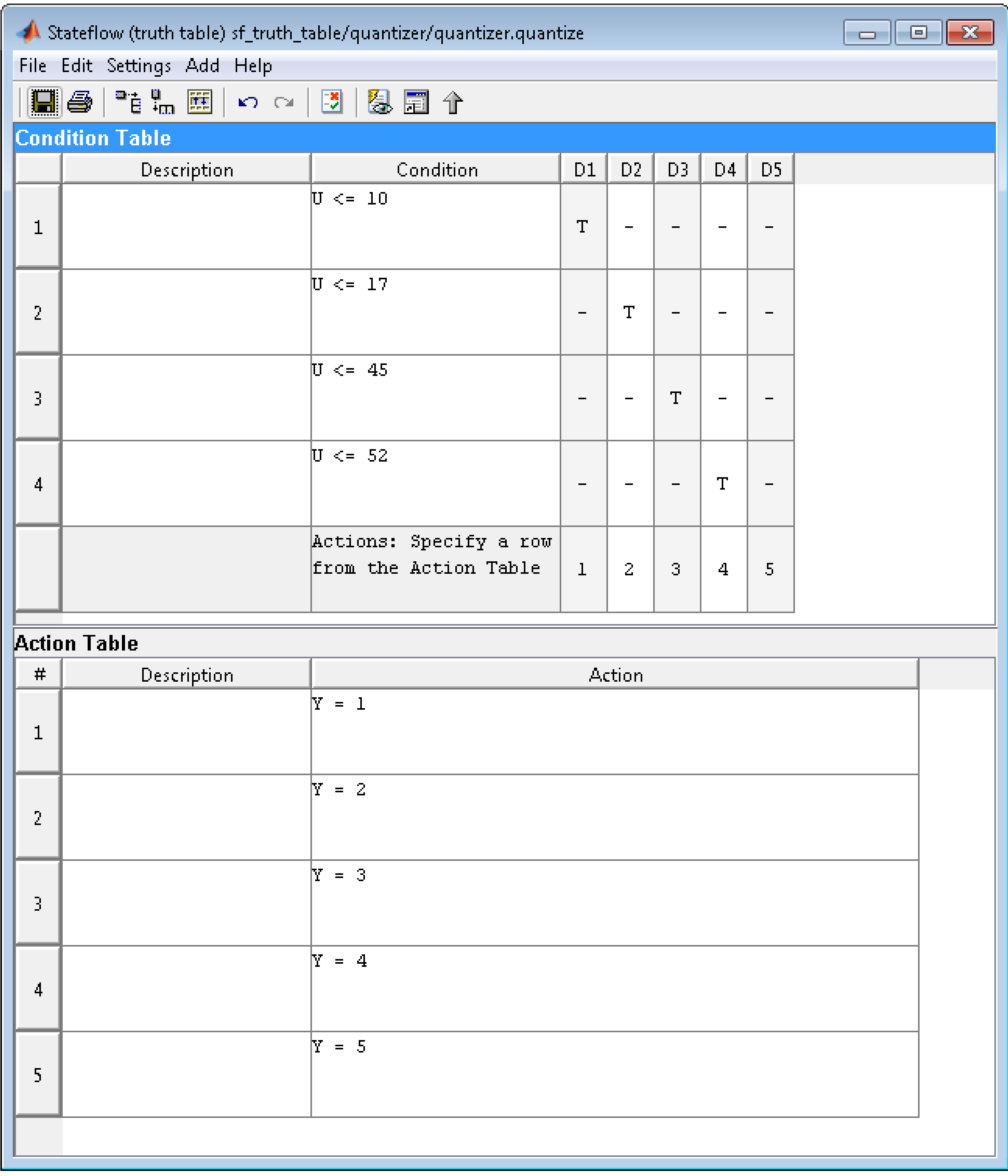
```
Y = 1 when 0  <= U <= 10
Y = 2 when 10 <  U <= 17
Y = 3 when 17 <  U <= 45
Y = 4 when 45 <  U <= 52
Y = 5 when 52 <  U
```

A stateless chart with a single call to a Truth Table function can represent this logic succinctly.

The following figure shows the `quantizer` chart, containing the Truth Table.



The following figure shows the threshold logic, as displayed in the Truth Table Editor.


Stateflow (truth table) sf_truth_table/quantizer/quantizer.quantize

File Edit Settings Add Help

Condition Table

	Description	Condition	D1	D2	D3	D4	D5
1		U <= 10	T	-	-	-	-
2		U <= 17	-	T	-	-	-
3		U <= 45	-	-	T	-	-
4		U <= 52	-	-	-	T	-
		Actions: Specify a row from the Action Table	1	2	3	4	5

Action Table

#	Description	Action
1		Y = 1
2		Y = 2
3		Y = 3
4		Y = 4
5		Y = 5

The following code excerpt shows VHDL code generated for the `quantizer` chart.

```
quantizer : PROCESS (Y_reg, U)
    -- local variables
    VARIABLE aVarTruthTableCondition_1 : std_logic;
    VARIABLE aVarTruthTableCondition_2 : std_logic;
```

```
        VARIABLE aVarTruthTableCondition_3 : std_logic;
        VARIABLE aVarTruthTableCondition_4 : std_logic;
    BEGIN
        Y_reg_next <= Y_reg;
        -- Condition #1
        aVarTruthTableCondition_1 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(10, 8));
        -- Condition #2
        aVarTruthTableCondition_2 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(17, 8));
        -- Condition #3
        aVarTruthTableCondition_3 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(45, 8));
        -- Condition #4
        aVarTruthTableCondition_4 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(52, 8));

        IF tmw_to_boolean(aVarTruthTableCondition_1) THEN
            -- D1
            -- Action 1
            Y_reg_next <= to_unsigned(1, 8);
        ELSIF tmw_to_boolean(aVarTruthTableCondition_2) THEN
            -- D2
            -- Action 2
            Y_reg_next <= to_unsigned(2, 8);
        ELSIF tmw_to_boolean(aVarTruthTableCondition_3) THEN
            -- D3
            -- Action 3
            Y_reg_next <= to_unsigned(3, 8);
        ELSIF tmw_to_boolean(aVarTruthTableCondition_4) THEN
            -- D4
            -- Action 4
            Y_reg_next <= to_unsigned(4, 8);
        ELSE
            -- Default
            -- Action 5
            Y_reg_next <= to_unsigned(5, 8);
        END IF;

    END PROCESS quantizer;
```

Note When generating code for a Truth Table block in a Simulink model, HDL Coder writes a separate entity/architecture file for the Truth Table code. The file is named `Truth_Table.vhd` (for VHDL) , `Truth_Table.v` (for Verilog) or `Truth_Table.sv` (for SystemVerilog).

See Also

State Transition Table | Truth Table | Sequence Viewer

Related Examples

- “Generate HDL for Mealy and Moore Finite State Machines” on page 26-7

More About

- “Hardware Realization of Stateflow Semantics” on page 26-6
- “Introduction to Stateflow HDL Code Generation” on page 26-2

Initialize Persistent Variables in MATLAB Functions

A persistent variable is a local variable in a MATLAB function that retains its value in memory between calls to the function. If you generate code from your model, you must initialize a persistent variable for your MATLAB functions. For more information, see `persistent`.

When using MATLAB functions that contain persistent variables in Simulink models, you should follow these guidelines:

- Initialize the persistent variables in functions only by accessing constants.
- Ensure the control flow of the function does not depend on whether the initialization occurs.

If you do not follow these guidelines, several conditions produce an initialization error:

- MATLAB Function blocks with persistent variables where the **Allow direct feedthrough** property is cleared
- MATLAB Function blocks with persistent variables in models with State Control blocks where **State control** is set to `Synchronous`
- Stateflow charts that implement Moore machine semantics and that use MATLAB functions with persistent variables

For example, the function `fcn` below uses a persistent variable, `n`. `fcn` violates both guidelines. The initial value of `n` depends on the input `u` and the `return` statement interrupts the normal control flow of the function. Consequently, this code produces an error when used in a model that has one of the conditions described above.

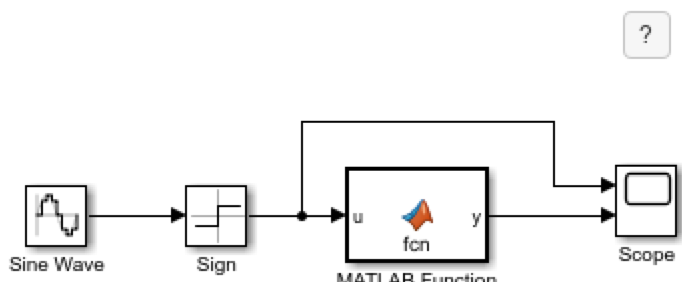
```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = u;
        y = 1;
        return
    end

    y = n;
    n = n + u;
end
```

To prevent the error, initialize the persistent variable by setting it to a constant value and removing the `return` statement. This modified version of `fcn` initializes the persistent variable without producing an error:

```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = 1;
    end

    y = n;
    n = n + u;
end
```

MATLAB Function Block with No Direct Feedthrough

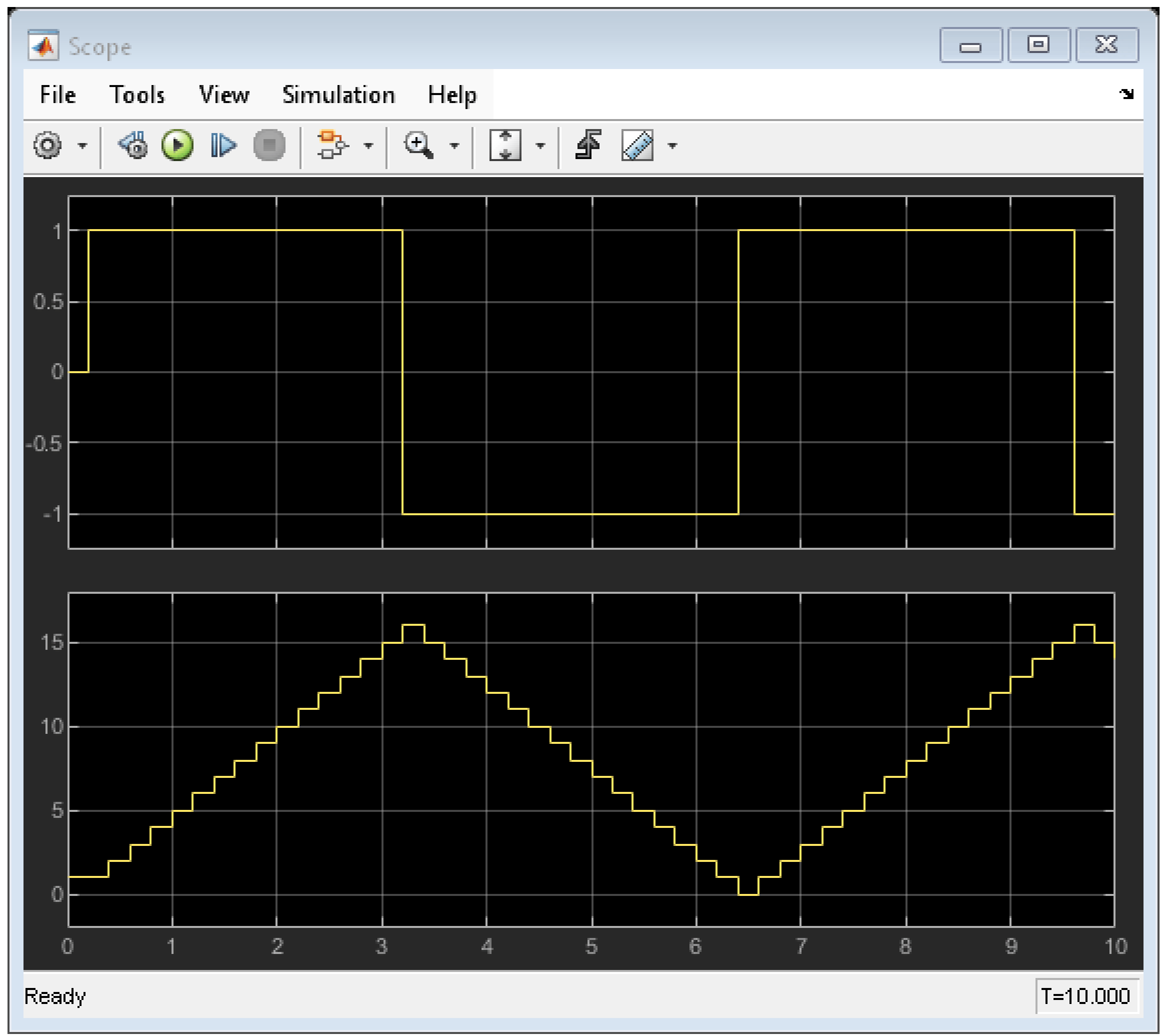
This model contains a MATLAB Function block that uses the first version of `fcn`, described previously. The MATLAB Function block input is a square wave, which is provided by a Sign and Sine Wave block. The MATLAB Function block adds the value of `u` to the persistent variable `n` at each time step.



Simulate the model. The simulation returns an error because:

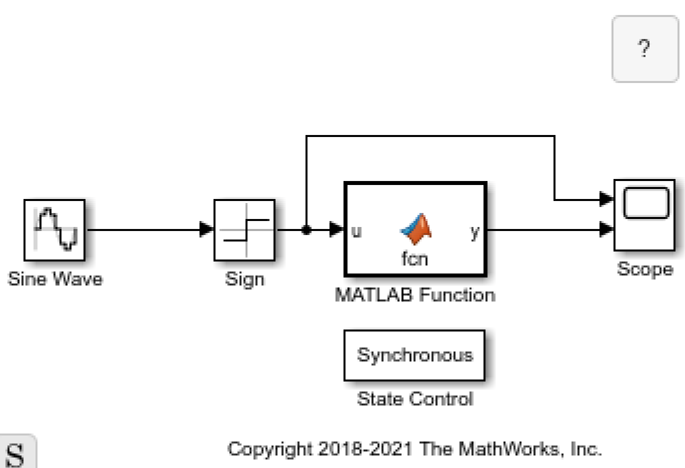
- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.
- The **Allow direct feedthrough** property of the MATLAB Function block is cleared.

Modify the MATLAB Function block code, as shown in the corrected version of `fcn`. Simulate the model again.



State Control Block in Synchronous Mode

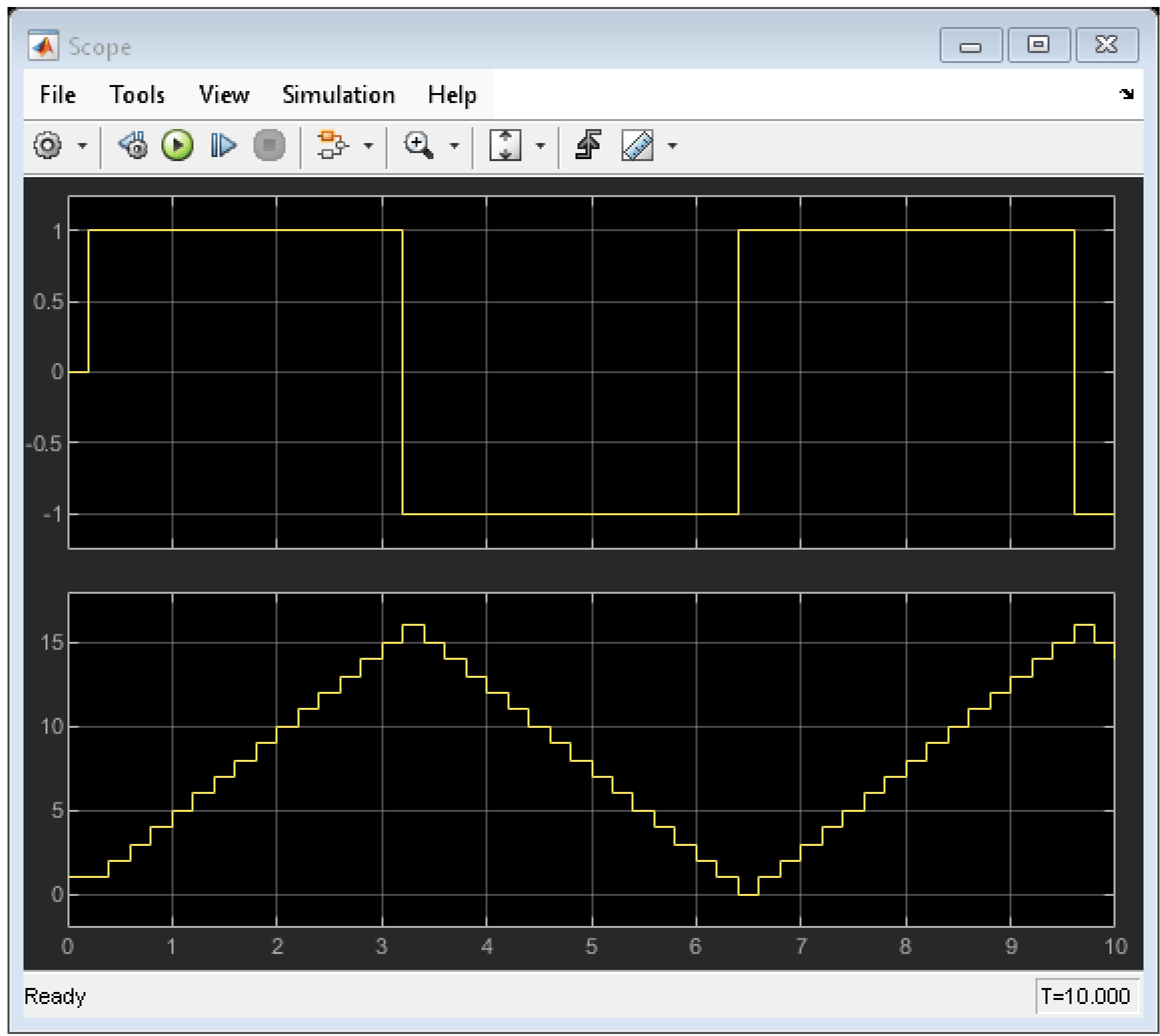
This model contains a MATLAB Function block that uses the first version of `fcn`, described previously. The MATLAB Function block input is a square wave, which is provided by a Sign and Sine Wave block. The MATLAB Function block adds the value of `u` to the persistent variable `n` at each time step. The model contains a State Control block where **State control** is set to `Synchronous`.



Simulate the model. The simulation returns an error because:

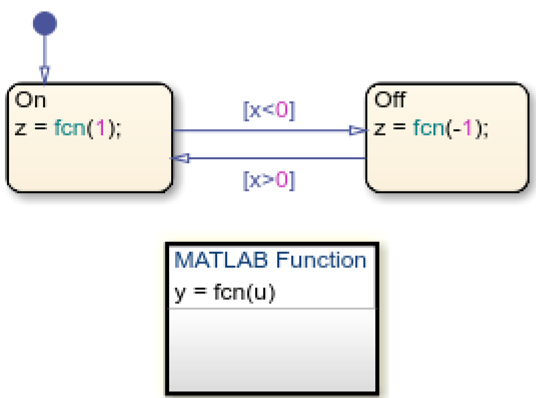
- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.
- The model contains a State Control block where **State control** is set to `Synchronous`.

Modify the MATLAB Function block code, as shown in the corrected version of `fcn`. Simulate the model again.



Stateflow Chart Implementing Moore Semantics

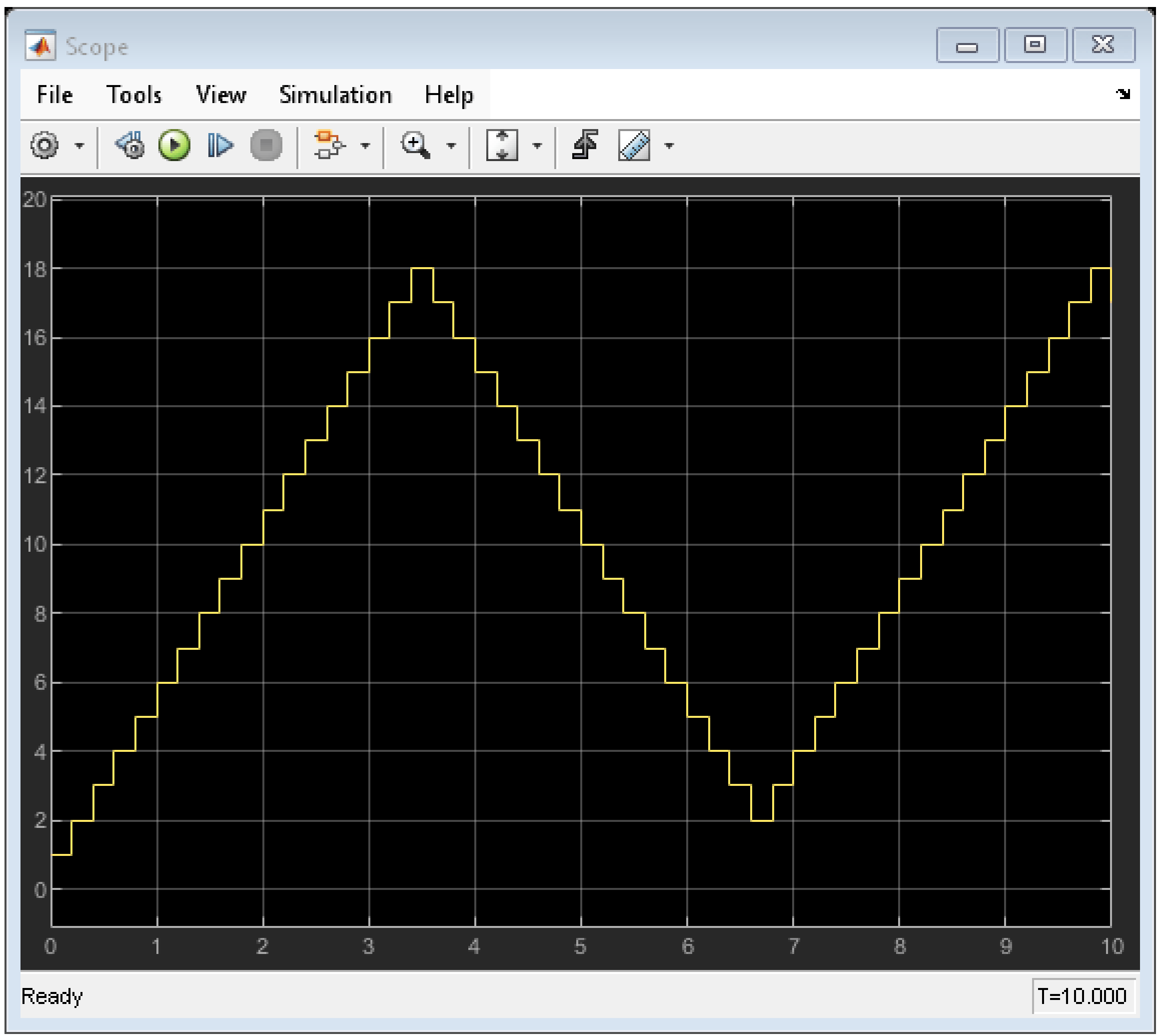
This model contains a Stateflow Chart with a MATLAB function that uses the first version of `fcn`, described previously. The MATLAB function adds the value (`1` or `-1`) determined by the active state to the persistent variable `n` at each time step.



Simulate the model. The simulation returns an error because:

- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.
- The chart implements Moore semantics.

Modify the MATLAB function code, as shown in the corrected version of `fcn`. Simulate the model again.



See Also

Blocks
MATLAB Function | State Control | Chart

Functions
`persistent`

More About

- "Use Nondirect Feedthrough in a MATLAB Function Block"
- "Synchronous Subsystem Behavior with the State Control Block" on page 25-75
- "Design Considerations for Moore Charts" (Stateflow)

Generating HDL Code with the MATLAB Function Block

- “HDL Applications for the MATLAB Function Block” on page 27-2
- “Generate HDL Code from a MATLAB Function Block” on page 27-4
- “Generate Instantiable Code for Functions” on page 27-12
- “MATLAB Function Block Design Patterns for HDL” on page 27-14
- “Design Guidelines for the MATLAB Function Block” on page 27-19
- “CORDIC Algorithm Using the MATLAB Function Block” on page 27-22
- “Create Hardware Design Patterns Using the MATLAB Function Block For HDL Code Generation” on page 27-23
- “Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks” on page 27-28

HDL Applications for the MATLAB Function Block

In this section...
“Structure of Generated HDL Code” on page 27-2
“HDL Applications” on page 27-2

Structure of Generated HDL Code

The MATLAB Function block contains a MATLAB function in a model. The inputs and outputs of the function are represented by the ports on the block, which allow you to interface your model to the function code. When you generate HDL code for a MATLAB Function block, HDL Coder generates two HDL files:

- A file containing entity and architecture code that implement the actual algorithm or computations generated for the MATLAB Function block.
- A file containing an entity definition and RTL architecture that provide a black box interface to the algorithmic code generated for the MATLAB Function block.

The structure of these code files is analogous to the structure of the model, in which a subsystem provides an interface between the root model and the function in the MATLAB Function block.

HDL Applications

The MATLAB Function block supports a subset of the MATLAB language that is well-suited to HDL implementation of various DSP and telecommunications algorithms, such as:

- Sequence and pattern generators
- Encoders and decoders
- Interleavers and de-interleavers
- Modulators and demodulators
- Multipath channel models; impairment models
- Timing recovery algorithms
- Viterbi algorithm; Maximum Likelihood Sequence Estimation (MLSE)
- Adaptive equalizer algorithms

You can also use the MATLAB Function block in a wide variety of floating-point applications. Both `single` and `double` types are supported. To learn more, see “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 21-205.

See Also

“Check for MATLAB Function block settings” on page 37-18

More About

- “Design Guidelines for the MATLAB Function Block” on page 27-19
- “Generate HDL Code from a MATLAB Function Block” on page 27-4

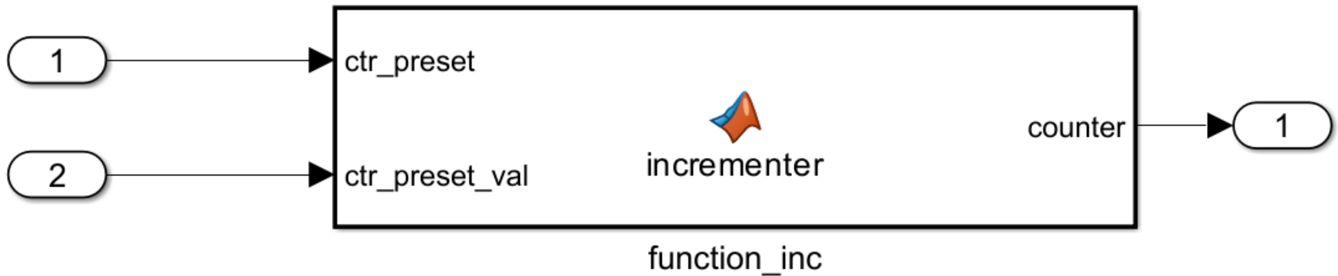
- “MATLAB Function Block Design Patterns for HDL” on page 27-14
- “Generate DUT Ports for Tunable Parameters” on page 14-18

Generate HDL Code from a MATLAB Function Block

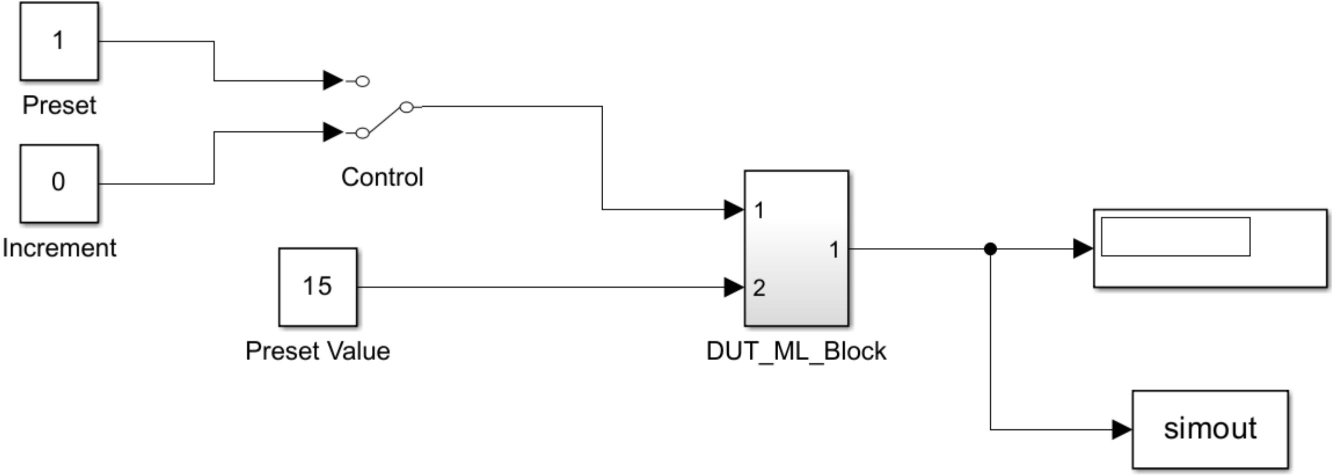
This example shows how to construct and configure a simple model and then generate VHDL code from the model.

The `MLHDLIncrementer` model in this example includes a MATLAB Function block that implements a simple fixed-point counter function, `incrementer`. The `incrementer` function is invoked once during each sample period. The function maintains a persistent variable count that the function either increments or reinitializes to a preset value, `ctr_preset_val`, depending on the value passed to the `ctr_preset` input of the MATLAB Function block. The function returns the counter value, `counter`, at the output of the MATLAB Function block.

The MATLAB Function block resides in a subsystem named `DUT_ML_Block`. The subsystem functions as a device under test (DUT) from which you generate HDL code.



The root-level model drives the subsystem and includes Display and To Workspace blocks. The Display and To Workspace blocks do not generate HDL code.



Create and Configure the Model

Create a model and run the `hdlsetup` command to set some model configuration parameters to values recommended for HDL code generation. The `hdlsetup` command uses the `set_param` function to set up models for HDL code generation. To customize the `hdlsetup` command, see "Customize hdlsetup Function Based on Target Application" on page 18-20.

Before you begin building the example model, set up a working folder for your model and generated code. This folder stores the model you create, and also contains subfolders of the generated code. The location of the folder does not matter, except that it should not be within the MATLAB tree.

1. Create a new model.

2. Save the model as `MLHDLIncrementer`.

3. In the MATLAB Command Window, enter:

```
hdlsetup('MLHDLIncrementer');
```

4. In Simulink, open the Configuration Parameters dialog box.

5. In the Solver pane, set these configuration parameters:

- **Fixed step size (fundamental sample time)**: 1
- **Stop time**: 5

6. Click **OK** to save your changes and close the Configuration Parameters dialog box.

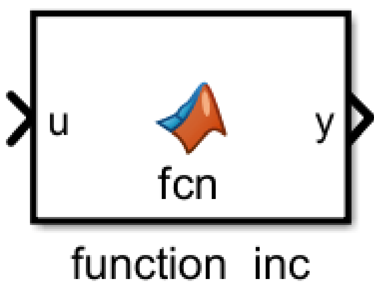
7. Save your model.

Add a MATLAB Function Block to the Model

1. Open the Library Browser. Then, select the **Simulink > User-Defined Functions** library.

2. Select the MATLAB Function block and add it to the model.

3. Change the block label to `function_inc`.



4. Close the Library Browser.

5. Save the model.

Set Fixed-Point Settings for the MATLAB Function Block

Configure the MATLAB Function block with the recommended `fimath` specifications. To configure the block:

1. Double-click the MATLAB Function block. The MATLAB Function block editor opens.

2. In the **Modeling** tab, in the **Design** section, click the **Property Inspector**. The Property Inspector displays the default `fimath` specification and other properties for the MATLAB Function block.

3. Expand the **Fixed-point properties** section and select **Specify Other**. Selecting this option enables the **MATLAB FUNCTION FIMATH** text entry field.

4. The `hdlfimath` function is a utility that defines a `fimath` specification optimized for HDL code generation. Replace the default specification in the **MATLAB FUNCTION FIMATH** field with:

```
hdlfimath;
```

This function sets these `fimath` specification properties to these settings:

- `ProductMode` property: `'FullPrecision'`
- `SumMode` property: `'FullPrecision'`

5. Set the **Treat these inherited signal types as fi objects** property to `Fixed-point`.

6. Save the model.

Program the MATLAB Function Block

Next, add the code to the MATLAB Function block to define the `incrementer` function, and then use diagnostics to check for errors.

1. In the MATLAB Function block editor, delete the default code.

2. Add this code into the editor:

```
function counter = incrementer(ctr_preset, ctr_preset_val)
% The function incrementer implements a preset counter that counts
% how many times this block is called.
%
% This example function shows how to model memory with persistent variables,
% using fimath settings suitable for HDL. It also demonstrates MATLAB
% operators and other language features that HDL Coder supports
% for code generation from Embedded MATLAB Function block.
%
% On the first call, the result 'counter' is initialized to zero.
% The result 'counter' saturates if called more than 2^14-1 times.
% If the input ctr_preset receives a nonzero value, the counter is
% set to a preset value passed in to the ctr_preset_val input.


persistent current_count;
if isempty(current_count)
    % zero the counter on first call only
    current_count = uint32(0);
end
```

```
counter = getfi(current_count);

if ctr_preset
    % set counter to preset value if input preset signal is nonzero
    counter = ctr_preset_val;
else
    % otherwise count up
    inc = counter + getfi(1);
    counter = getfi(inc);
end

% store counter value for next iteration
current_count = uint32(counter);

end

function hdl_fi = getfi(val)

nt = numerictype(0,14,0);
fm = hdlfimath;
hdl_fi = fi(val, nt, fm);

end
```

3. Save the model. Doing so updates the model window and the MATLAB Function block.

Changing the function header of the MATLAB Function block makes these changes to the block icon:

- The function name of the block changes to `incrementer`.
- The arguments `ctr_preset` and `ctr_preset_val` appear as input ports to the block.
- The return value `counter` appears as an output port from the block.

4. Resize the block to make the port labels more readable.

5. Save the model again.

Construct and Connect the Subsystem

You construct a subsystem containing the MATLAB Function block to serve as a device under test (DUT) from which you generate HDL code. You then set the port data types and connect the subsystem ports to the model.

To construct the subsystem:

1. Click the `incrementer` block.
2. On the **Modeling** tab of the Simulink Toolstrip, select **Create Subsystem**. A subsystem, labeled `Subsystem`, is created in the model window.
3. Change the label of the Subsystem block to `DUT_ML_Block`.

To set the port data types for the MATLAB Function block:

1. Double-click the subsystem to view its contents. The subsystem contains the `incrementer` MATLAB Function block and Inport and Outport blocks connected to its ports.
2. Double-click the `incrementer` block to open the MATLAB Function block editor.

3. In the **Modeling** tab, in the **Design** section, click **Model Explorer**.
4. Click **`function_inc`** and select **`ctr_preset`** in the middle pane. In the right pane, click the Show data type assistant button >> to display the **Data Type Assistant** section. Set **Mode** to `Built in` and set the data type to `boolean`. Click **Apply**.
5. In the middle pane, click `counter`. Open the Data Type Assistant. Verify that **Mode** is set to `Inherit: Same as Simulink`.
6. Return to the top-level model and save it.

To connect the subsystem ports to the model:

1. Open the Library Browser.
2. From the **Sources** library, add a Constant block to the model. Set the value of the Constant block to 1, and the **Output data type** to `boolean`. Change the block label to `Preset`.
3. Make a copy of the `Preset` Constant block. Set its value to 0, and change its block label to `Increment`.
4. From the **Signal Routing** library, add a Manual Switch block to the model. Change its label to `Control`. Connect its output to the input port named **1** of the `DUT_ML_Block` subsystem.
5. Connect the `Preset` Constant block to the top input port of the `Control` switch block. Connect the `Increment` Constant block to the bottom input port of the `Control` switch block.
6. Add a third Constant block to the model. Set the value of the Constant to 15, and the **Output data type** to `Inherit via back propagation`. Change the block label to `Preset Value`.
7. Connect the `Preset Value` Constant block to the input port named **2** of the `DUT_ML_Block` subsystem.
8. From the **Sinks** library, add a Display block to the model. Connect it to the output port named **1** of the `DUT_ML_Block` subsystem.
9. From the **Sinks** library, add a To Workspace block to the model. Branch the output signal from the `DUT_ML_Block` subsystem to the To Workspace block.
10. Save the model.

Use the built-in diagnostics of the MATLAB Function block to test for syntax errors:

1. Open the `DUT_ML_Block` subsystem.
2. Double-click the MATLAB Function block.
3. In the Function tab, click **Update Model** to compile and build the MATLAB Function block code.

The build process displays some progress messages.

The build process builds an S-function for use in simulation. During the build process, the software generates C code for this S-function. The code generation messages you see during the build process refer to generation of C code, not HDL code generation.

When the build concludes without encountering an error, a message window indicates that parsing was successful. If errors are found, the Diagnostics Manager lists them. For more information on debugging MATLAB Function block build errors, see MATLAB Function.

Compile the Model and Display Port Data Types

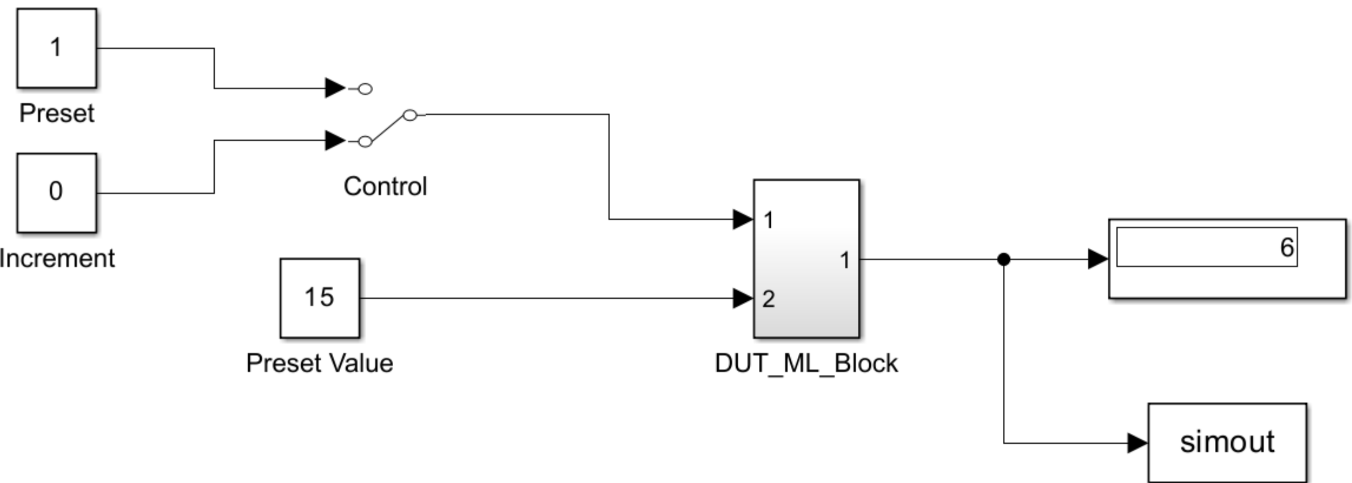
When you compile the model, Simulink verifies the model structure and settings, and updates the model display.

1. In the **Debug** tab of the Simulink Toolstrip, click **Information Overlays** > **Ports**. In the Ports section, select **Base data types**.
2. Press **Ctrl+D** to compile and update the model. This action rebuilds the code. After the model compiles, the block diagram updates to show the port data types.
3. Save the model.

Simulate the Model

Start simulation by clicking the **Run** button on the **Simulation** tab. If required, the code rebuilds before the simulation starts.

After the simulation completes, the Display block shows the final output value returned by the `incrementer` function block. For example, given a **Start time** of 0, a **Stop time** of 5, and a zero value at the input port **1**, the simulation returns a value of 6.



Copyright 2006-2023 The MathWorks, Inc.

You can experiment with the results of toggling the `Control` switch, changing the `Preset Value` constant, and changing the total simulation time. You can also examine the workspace variable `simout`, which is bound to the To Workspace block.

Generate HDL Code

To generate HDL code for the model, you first select the DUT_ML_Block subsystem for HDL code generation, set basic code generation configurations, and then generate VHDL code for the subsystem.

Select the `DUT_ML_Block` subsystem and configure it for HDL code generation:

1. Open the Configuration Parameters dialog box and click the **HDL Code Generation** pane.

2. Ensure that these parameters are set as follows:

- **Generate HDL for**: MLHDLIncrementer/DUT_ML_Block
- **Language**: VHDL
- **Folder**: hdlsrc

The **Folder** field specifies the code generation target folder. This folder is a subfolder of your working folder. Before generating code, make sure you are in the current working folder.

To generate code:

1. In Simulink Toolstrip, from the **Apps** gallery, click **HDL Coder**. In the **HDL Code** tab, click **Generate HDL Code**.

HDL Coder™ compiles the model before generating code. Depending on model display options, such as the port data types, the appearance of the model might change after code generation.

2. As code generation proceeds, the Diagnostic Viewer displays progress messages. The process should complete with a message like the following:

```
### HDL Code Generation Complete.
```

In the Diagnostic Viewer, the names of generated VHDL files in the progress messages are hyperlinked. After code generation completes, you can click these hyperlinks to view the files in the MATLAB editor.

3. The code generation process created the `hdlsrc` folder as a subfolder of your working folder. To view generated code and script files, double-click the `hdlsrc` folder in Current Folder browser.

4. Observe that two VHDL files were generated. The structure of HDL code generated for MATLAB Function blocks is similar to the structure of code generated for Stateflow® charts and Digital Filter blocks. The VHDL files that were generated in the `hdlsrc` folder are:

- `function_inc.vhd`: VHDL code. This file contains entity and architecture code that implements the computations generated for the MATLAB Function block.
- `DUT_ML_Block.vhd`: VHDL code. This file contains an entity definition and RTL architecture that provides a black box interface to the code generated in `function_inc.vhd`.

The structure of these code files is analogous to the structure of the model, in which the `DUT_ML_Block` subsystem provides an interface between the root model and the `incrementer` function in the MATLAB Function block.

The other files generated in the `hdlsrc` folder are:

- `DUT_ML_Block_compile.do`: Mentor Graphics® ModelSim® compilation script (`vcom` command) that compiles the VHDL code in the two `.vhd` files.
- `DUT_ML_Block_map.txt`: Mapping file. This report file maps generated entities, or modules, to the subsystems that generated them. See “Trace Code Using the Mapping File” on page 23-34.

To view the generated VHDL code in the MATLAB editor, double-click the `DUT_ML_Block.vhd` or `function_inc.vhd` files in the Current Folder browser.

See Also

“Check for MATLAB Function block settings” on page 37-18

More About

- “Design Guidelines for the MATLAB Function Block” on page 27-19
- “HDL Applications for the MATLAB Function Block” on page 27-2
- “MATLAB Function Block Design Patterns for HDL” on page 27-14
- “Generate DUT Ports for Tunable Parameters” on page 14-18

Generate Instantiable Code for Functions

In this section...

"How To Generate Instantiable Code for Functions" on page 27-12
"Generate Code Inline for Specific Functions" on page 27-12
"Limitations for Instantiable Code Generation for Functions" on page 27-12

For the MATLAB Function block, you can use the **InstantiateFunctions** parameter to generate a VHDL `entity`, Verilog or SystemVerilog `module` for each function. HDL Coder generates code for each `entity` or `module` in a separate file.

The **InstantiateFunctions** options for the MATLAB Function block are listed in the following table.

InstantiateFunctions Setting	Description
`'off'` (default)	Generate code for functions inline.
`'on'`	Generate a VHDL `entity`, Verilog or SystemVerilog `module` for each function, and save each `module` or `entity` in a separate file.

How To Generate Instantiable Code for Functions

To set the **InstantiateFunctions** parameter using the HDL Block Properties dialog box:

1. Right-click the MATLAB Function block.
2. Select **HDL Code** > **HDL Block Properties**.
3. For **InstantiateFunctions**, select **on**.

To set the **InstantiateFunctions** parameter from the command line, use `hdlset_param`. For example, to generate instantiable code for functions in a MATLAB Function block, `myMatlabFcn`, in your DUT subsystem, `myDUT`, enter:

```
hdlset_param('my_DUT/my_MATLABFcnBlk', 'InstantiateFunctions', 'on')
```

Generate Code Inline for Specific Functions

If you want to generate instantiable code for some functions but not others, enable the option to generate instantiable code for functions, and use `coder.inline`. See `coder.inline` for details.

Limitations for Instantiable Code Generation for Functions

The software generates code inline when:

- Function calls are within conditional code or `for` loops.
- Any function is called with a nonconstant `struct` input.
- The function has state, such as a persistent variable, and is called multiple times.
- There is an enumeration anywhere in the design function.

See Also

"Check for MATLAB Function block settings" on page 37-18

More About

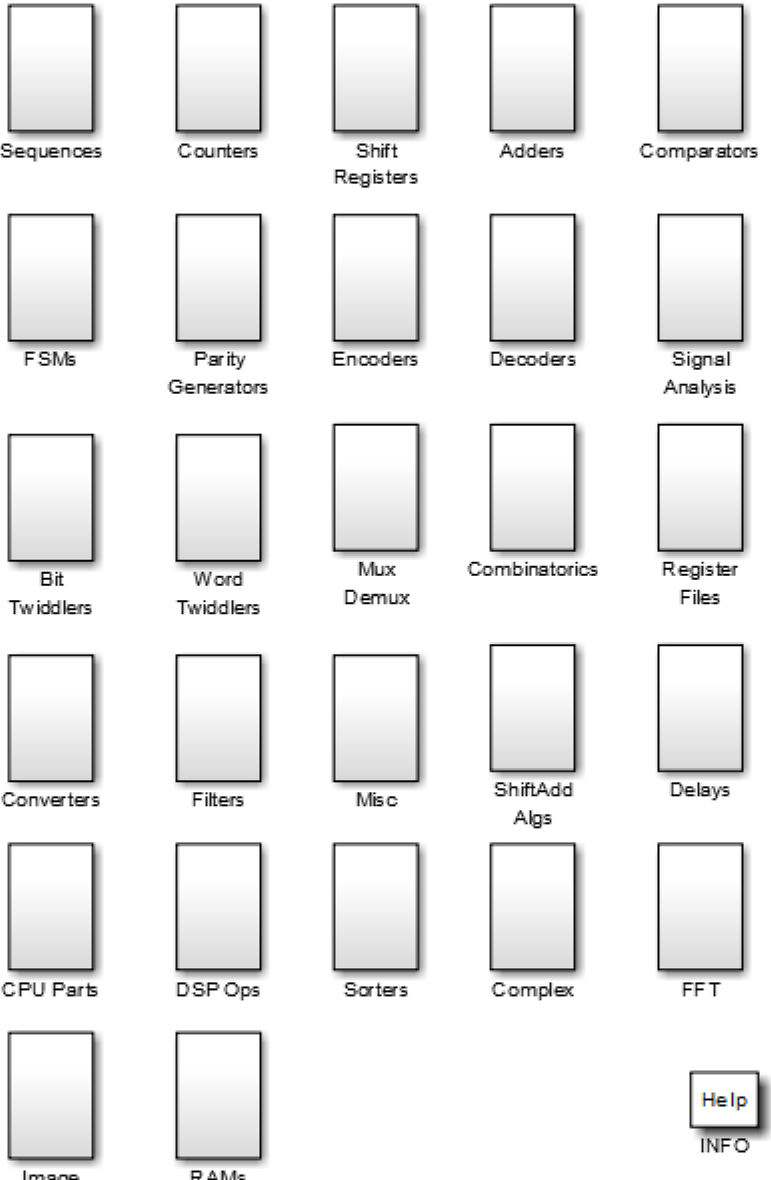
- "Design Guidelines for the MATLAB Function Block" on page 27-19
- "Generate HDL Code from a MATLAB Function Block" on page 27-4
- "MATLAB Function Block Design Patterns for HDL" on page 27-14
- "Generate DUT Ports for Tunable Parameters" on page 14-18

MATLAB Function Block Design Patterns for HDL

In this section...

"HDL Design Pattern Blocks" on page 27-16
"Using Blocks in this Library for HDL Code Generation" on page 27-17
"Fixed-Point Algorithm Support" on page 27-17

You can use MATLAB Function blocks to write algorithms for implementation in Simulink and Stateflow, then use this code for HDL code generation. You can use the HDL design patterns library `eml_hdl_design_patterns` to implement common hardware modeling algorithms using the MATLAB Function block. The library contains examples that demonstrate hardware-friendly applications of the MATLAB Function block for HDL code generation.



To open the library, at the MATLAB command prompt, enter:

```
eml_hdl_design_patterns
```

You can use the blocks in the library to model various hardware elements by:

- Copying a block from the library to your model and using it as a computational unit.
- Copying the code from the block and using it as a local function in an existing MATLAB Function block.

When you create custom blocks, you can control whether to inline or instantiate the HDL code generated from MATLAB Function blocks. To inline or instantiate the HDL code, in the Configuration

Parameters dialog box, click **HDL Code Generation > Global Settings**. In the right pane, in the **Coding style** tab, select **Inline MATLAB Function block code**.

Note Do not use the **Inline MATLAB Function block code** setting when you set the **Architecture** HDL block property of the MATLAB Function block to `MATLAB Datapath`. Enable **FlattenHierarchy** instead. For more information, see "HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture" on page 21-205.

HDL Design Pattern Blocks

This table summarizes some of the categories of HDL design patterns in the `eml_hdl_design_patterns` library. For more information on the different blocks, see "Create Hardware Design Patterns Using the MATLAB Function Block For HDL Code Generation" on page 27-23.

Library Subsystem	Purpose
`Combinatorics`	Combinatorial fixed-point algorithms, such as expressions containing addition, subtraction, and multiplication operators with different fixed-point data types.
`Adders`	Algorithms that model different adder logic.
`Delays`	Algorithms to model integer delay, tap delay, and tap delay vector blocks by using vectors of persistent variables. Use these design patterns to implement sequential algorithms that carry state between executions of the MATLAB Function block in a model.
`Comparators`	Algorithms for finding the minimum value of a vector.
`FSMs`	MATLAB Function block control constructs such as `switch/case` and `if-else`, coupled with fixed point arithmetic operations to model control logic quickly. The `FSMs/mealy_fsm_blk` and `FSMs/moore_fsm_blk` blocks provide example implementations of Mealy and Moore finite state machines in the MATLAB Function block.
`Counters`	Counter logic that shows how to model state and quantize data elements within loops.

Library Subsystem	Purpose
Shift Registers	Algorithms that model shift register hardware elements. The shift_reg_1by32 block shows how to model shift registers by using the bitsliceget and bitconcat functions. This function implements a serial input and output shifters with a 32–bit fixed-point operand input. The shift_reg_1by64 block shows a 64 bit shifter. In this case, the shifter uses two fixed-point words to represent the operand, overcoming the 32-bit word length limitation for fixed-point integers.
Word Twiddlers	Algorithms that perform word conversions, such as conversions from an integer to bits and bits to an integer.

Using Blocks in this Library for HDL Code Generation

To build models using blocks in this library and generate HDL code:

- Create or open a model.
- Copy a block from eml_hdl_design_patterns library to your model.
- Place the block in the device under test (DUT) subsystem.
- Run the hdlsetup command to set up your model for HDL code generation.
- Run the makehdl function to generate HDL code for the DUT subsystem.

For more information on generating HDL code from a MATLAB Function block, see “Generate HDL Code from a MATLAB Function Block” on page 27-4.

Fixed-Point Algorithm Support

The MATLAB Function block supports floating-point arithmetic and fixed-point arithmetic by using the Fixed-Point Designer fi function. This function supports rounding and saturation modes that you can use to code algorithms that manipulate arbitrary word and fraction lengths. HDL Coder supports all fi rounding and overflow modes. HDL code generated from the MATLAB Function block is bit-true to MATLAB semantics. Generated code uses bit manipulation and bit access operators, such as slice, extend, reduce, and concatenate that are native to VHDL and Verilog. The eml_hdl_design_patterns library includes example blocks that use the fi object, such as the blocks in the Adders subsystem.

See Also

“Check for MATLAB Function block settings” on page 37-18

More About

- “Create Hardware Design Patterns Using the MATLAB Function Block For HDL Code Generation” on page 27-23

- "Design Guidelines for the MATLAB Function Block" on page 27-19
- "Generate HDL Code from a MATLAB Function Block" on page 27-4
- "Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks" on page 27-28
- "Generate DUT Ports for Tunable Parameters" on page 14-18

Design Guidelines for the MATLAB Function Block

In this section...

"Use Compiled External Functions With MATLAB Function Blocks" on page 27-19
"Build the MATLAB Function Block Code First" on page 27-19
"Use the hdlfimath Utility for Optimized FIMATH Settings" on page 27-19
"Use Optimal Fixed-Point Option Settings" on page 27-20
"Set the Output Data Type of MATLAB Function Blocks Explicitly" on page 27-20
"Using Tunable Parameters" on page 27-20
"Run HDL Model Check for MATLAB Function Blocks" on page 27-20
"Use MATLAB Datapath Architecture for Enhanced HDL Optimizations" on page 27-21

Use Compiled External Functions With MATLAB Function Blocks

The HDL Coder software supports HDL code generation from MATLAB Function blocks that include compiled external functions. This feature enables you to write reusable MATLAB code and call it from multiple MATLAB Function blocks.

Such functions must be defined in files that are on the MATLAB Function block path. Use the `%#codegen` compilation directive to indicate that the MATLAB code is suitable for code generation. See "Function Definition" for information on how to create, compile, and invoke external functions.

Build the MATLAB Function Block Code First

Before generating HDL code for a subsystem containing a MATLAB Function block, build the MATLAB Function block code to check for errors. To build the code, click the **Build Model** button in the function editor.

Use the hdlfimath Utility for Optimized FIMATH Settings

The `hdlfimath` function is a utility that defines a FIMATH specification that is optimized for HDL code generation.

The following listing shows the `fimath` setting defined by `hdlfimath`.

```
hdlfm = fimath(...
    'RoundMode', 'floor',...
    'OverflowMode', 'wrap',...
    'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
    'SumMode', 'FullPrecision', 'SumWordLength', 32,...
    'CastBeforeSum', true);
```

The HDL division operator supports these rounding modes:

- `'zero'` — For signed and unsigned fixed point numbers.
- `'floor'` — For unsigned division only.

When the `fimath` `OverflowMode` property of the `fimath` specification is set to `'Saturate'`, HDL code generation is disallowed for the following cases:

- `SumMode` is set to `'SpecifyPrecision'`
- `ProductMode` is set to `'SpecifyPrecision'`

Use Optimal Fixed-Point Option Settings

Use the default (`Fixed-point`) setting for the **Treat these inherited signal types as fi objects** option.

Clear the **Saturate on integer overflow** setting for the MATLAB Function block.

Set the Output Data Type of MATLAB Function Blocks Explicitly

By setting the output data type of a MATLAB Function block explicitly, you enable optimizations for RAM mapping and pipelining. Avoid inheriting the output data type for a MATLAB Function block for which you want to enable optimizations.

Using Tunable Parameters

HDL Coder supports both tunable and non-tunable parameters with the following data types:

- Scalar
- Vector
- Complex
- Structure
- Enumeration

When using tunable parameters with the MATLAB Function block:

- The tunable parameter should be a Simulink.Parameter object with the `StorageClass` set to `ExportedGlobal`.

  ```
  x = Simulink.Parameter
  x.Value = 1
  x.CoderInfo.StorageClass = 'ExportedGlobal'
  ```
- Double-click the MATLAB Function block and open the **Symbols** pane and **Property Inspector** in the **Design** section in the **Modeling** tab. In the Symbols Pane, select the tunable parameter. In the Property Inspector, select the **Tunable** check box.

For details, see “Generate DUT Ports for Tunable Parameters” on page 14-18.

Run HDL Model Check for MATLAB Function Blocks

When your design contains MATLAB Function blocks, before you generate HDL code, you can run the check “Check for MATLAB Function block settings” on page 37-18 in the HDL Code Advisor. This check verifies whether you use the recommended MATLAB Function blocks for HDL code generation. The settings include whether `fimath` settings are defined as per `hdlfimath` and **Saturate on integer overflow** check box is cleared.

See also “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 38-2.

Use MATLAB Datapath Architecture for Enhanced HDL Optimizations

In the HDL Block Properties dialog box for the MATLAB Function blocks, you can specify whether to use `MATLAB Function` or `MATLAB Datapath` as the HDL architecture. Floating-point models use the `MATLAB Datapath` architecture even if you specify the HDL architecture as `MATLAB Function` on the block. Fixed-point models use the `MATLAB Function` architecture by default.

To perform various speed and area optimizations such as sharing and distributed pipelining inside the MATLAB Function block and across the MATLAB Function block boundary with other Simulink blocks, use the `MATLAB Datapath` architecture. When you use this architecture, the code generator treats the MATLAB Function block like a regular Subsystem block. This capability enables you to optimize the algorithm inside the MATLAB Function block and across the MATLAB Function block with other blocks in your model.

See “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 21-205.

See Also

“Check for MATLAB Function block settings” on page 37-18

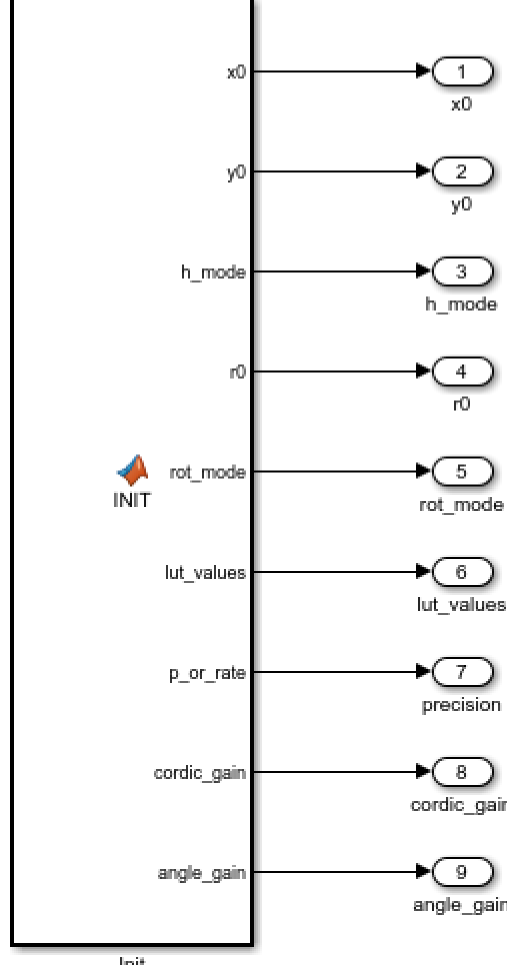
More About

- “Generate HDL Code from a MATLAB Function Block” on page 27-4
- “MATLAB Function Block Design Patterns for HDL” on page 27-14
- “Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks” on page 27-28
- “Generate DUT Ports for Tunable Parameters” on page 14-18

CORDIC Algorithm Using the MATLAB Function Block

This example shows how to use HDL Coder™ to check, generate and verify HDL for a fixed-point CORDIC model implementing sin and cos trigonometric functions using the MATLAB Function Block.

```
open_system("hdlcordic_eml.slx");
```



Create Hardware Design Patterns Using the MATLAB Function Block For HDL Code Generation

You can use the HDL design patterns library `em_hdl_design_patterns` to implement patterns that use the MATLAB Function block to solve common hardware modeling problems. For more information on the `em_hdl_design_patterns` library, see "MATLAB Function Block Design Patterns for HDL" on page 27-14.

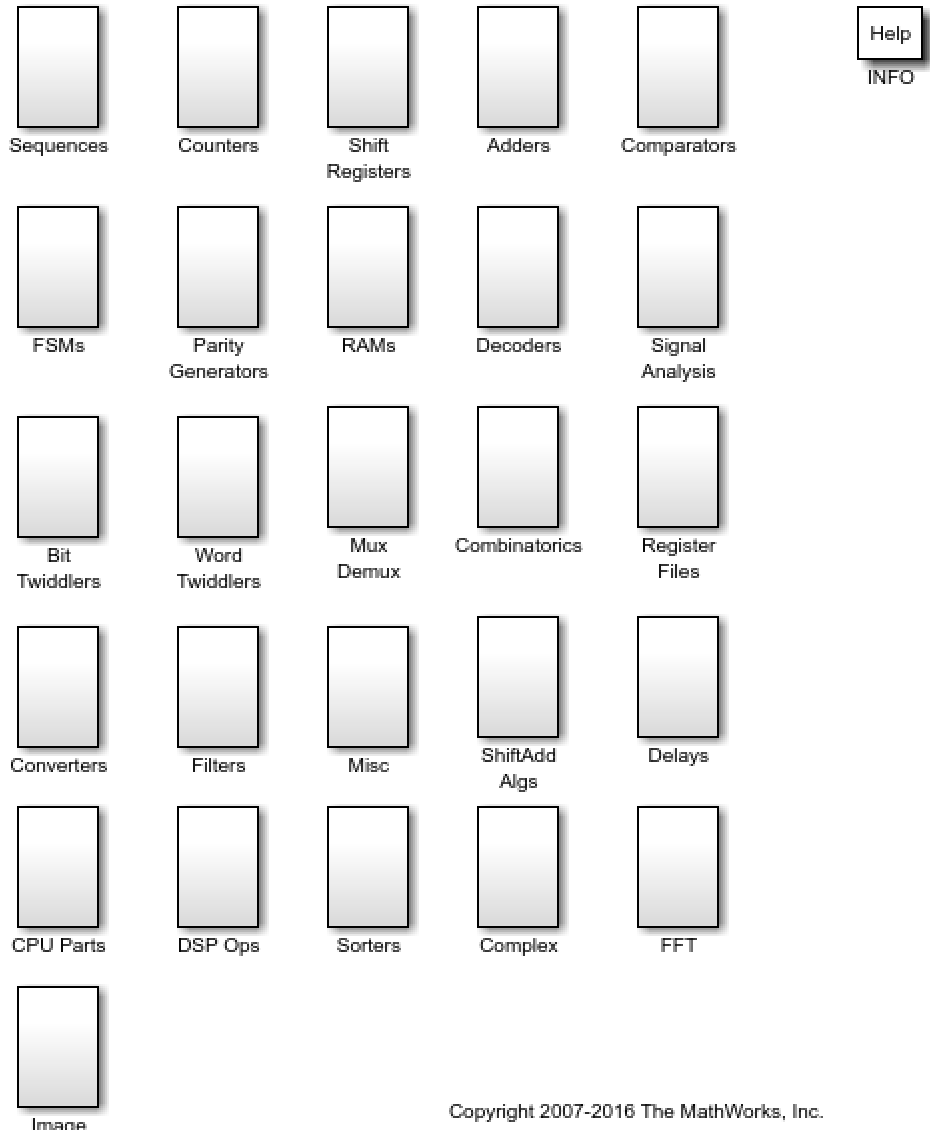
HDL Coder™ supports a subset of MATLAB Function block features that you can use in HDL implementations of various digital signal processing (DSP) and telecommunication applications, such as sequence detectors, pattern generators, encoders, and decoders. Some of the features supported by the MATLAB Function block in HDL Coder are:

- Numeric classes, including `int`, `uint`, `logical`, `single`, and `double`
- Fixed-point arithmetic using the `fi` object
- Arithmetic, logical, relational, and bitwise operator support
- MATLAB® expressions that use the preceding operators
- 1-D and 2-D matrix operations
- Matrix subscripting
- Control flow using `if`, `switch`, and `for` statements
- Subfunctions
- Persistent variables that model state
- Fixed point and integer MATLAB library functions

Open the Pattern Library

The MATLAB Function block expresses algorithm behavior in a concise and textual way, which enables you to model hardware algorithms at a high level. You can use HDL Coder to implement these MATLAB Function block hardware-friendly algorithms. The sample patterns library `em_hdl_design_patterns` shows how to model common, HDL-supported hardware modeling constructs using MATLAB Function blocks. Open this sample library model at the MATLAB command prompt by entering:

```
open_system('eml_hdl_design_patterns')
```



Model Arithmetic Expressions Using MATLAB Function Blocks

In the `em_hdl_design_patterns` library, you can use the blocks in the `Adders` and `Misc` subsystems to model arithmetic expressions. To model arbitrary length arithmetic operations using signed and unsigned logic vectors, use the `fi` object. When using `fi` objects in a MATLAB Function block, the `fi` function helps you to define a fixed-point object with a customized `numerictype` object and `fimath` object. The `numerictype` object defines the sign, word length, and fraction length attributes of the `fi` object and the `fimath` object defines rounding and saturation modes. For more information, see `fi`. To generate HDL code using the `fi` function, use these `fimath` settings:

```
fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
'SumMode', 'FullPrecision', 'SumWordLength', 32);
```

When modeling algorithms that require more complicated rounding and saturation logic, you can use other `fimath` modes for `rounding`, such as `floor`, `ceil`, `fix`, or `nearest`, and `overflow`, such as `wrap`, or `saturate`.

To see the affect of `fimath` properties on generated HDL code, generate HDL code for `eml_hdl_design_patterns/Adders/add_with_carry` and `eml_hdl_design_patterns/Misc/eml_expr`.

Model Combinatorics Using Fixed-Point Algorithms

Open the `eml_hdl_design_patterns` library and select the `Combinatorics/eml_expr` block. The `eml_expr` MATLAB Function block implements a simple expression that contains addition, subtraction, and multiplication operators with different fixed-point data types. Generate HDL code for the MATLAB Function block by following the process in “Generate HDL Code from a MATLAB Function Block” on page 27-4.

This is the generated VHDL code:

```
BEGIN
   --MATLAB Function 'Subsystem/eml_expr': '<S2>:1'
   -- fixpt arithmetic expression
   --'<S2>:1:4'
   mul_temp <= signed(a) * signed(b);
   sub_cast <= resize(mul_temp, 11);
   add_cast <= resize(signed(a & '0'), 7);
   add_cast_0 <= resize(signed(b), 7);
   add_temp <= add_cast + add_cast_0;
   sub_cast_0 <= resize(add_temp & '0' & '0', 11);
   expr <= sub_cast - sub_cast_0;
   -- cast the result to correct output type
   --'<S2>:1:7'

   y <= "0111111" WHEN ((expr(10) = '0') AND (expr(9 DOWNTO 7) /= "000"))
           OR ((expr(10) = '0') AND (expr(7 DOWNTO 1) = "0111111"))
          ELSE
         "1000000" WHEN (expr(10) = '1') AND (expr(9 DOWNTO 7) /= "111")
          ELSE
          std_logic_vector(expr(7 DOWNTO 1) + ("0" & expr(0)));
END fsm_SFHDL;
```

This is the generated Verilog code from the `eml_expr` MATLAB Function block:

```
//MATLAB Function 'Subsystem/eml_expr': '<S2>:1'
   // fixpt arithmetic expression
   //'<S2>:1:4'
   assign mul_temp = a * b;
   assign sub_cast = mul_temp;
   assign add_cast = {a[4], {a, 1'b0}};
   assign add_cast_0 = b;
   assign add_temp = add_cast + add_cast_0;
   assign sub_cast_0 = {{2{add_temp[6]}}, {add_temp, 2'b00}};
   assign expr = sub_cast - sub_cast_0;
   // cast the result to correct output type
   //'<S2>:1:7'
   assign y = (((expr[10] == 0) && (expr[9:7] != 0))
             || ((expr[10] == 0) && (expr[7:1] == 63)) ? 7'sb0111111 :
             ((expr[10] == 1) && (expr[9:7] != 7) ? 7'sb1000000 :
             expr[7:1] + $signed({1'b0, expr[0]})));
```

The generated HDL code shows the conversion of this expression with fixed-point operands. The default `fimath` specification for the blocks determines the behavior of arithmetic expressions using fixed-point operands inside the MATLAB Function block. You can see the `fimath` settings in the block comments of the `eml_expr` block.

Model State Using Persistent Variables

To model complex control logic, your model must be able to model registers. When generating code, you can represent state-holding elements as persistent variables. A persistent variable retains its value across function calls in the MATLAB code and across sample times steps during simulation. State-holding elements in hardware such as registers and flip-flops have similar behavior. Consequently, persistent variables in MATLAB Function blocks map to registers in hardware.

The `Delays` subsystem in the `em_hdl_design_patterns` library contains MATLAB Function blocks that show how you can use global and local reset conditions to change the values of persistent variables and how you can use vectors of persistent variables to model integer delay, tap delay, and tap delay vector blocks. You can use these design patterns to implement sequential algorithms that carry state between executions of the MATLAB Function block in a model.

Note that the MATLAB code must read the persistent variable before it is written in order for HDL Coder to map the persistent variable to a register in the HDL code. When generating code, HDL Coder displays a warning message if the MATLAB code does not follow this convention.

Model Control Logic and Finite State Machines

You can use conditional statements such as `switch`, `case`, `if`, and `else` together with fixed-point arithmetic operations and delay elements to model control logic. In the `em_hdl_design_patterns` library, the MATLAB Function blocks in the `FSMs` subsystems show how to use these conditional statements. The `mealy_fsm_blk` and `moore_fsm_blk blocks` provide example implementations of Mealy and Moore finite state machines in a MATLAB Function block.

The MATLAB Function block can model simple state machines and other control-based hardware algorithms, such as pattern matchers or synchronization-related controllers, by using control statements and persistent variables. For modeling more complex and hierarchical state machines with complicated temporal logic, use a Stateflow® chart to model the state machine.

Model Counters

The MATLAB Function blocks in the `Counters` subsystem show how to model state and quantize data elements in loops. You can use counters for global reset control of persistent state variables and local reset control. To implement arithmetic and control logic algorithms in MATLAB Function blocks intended for HDL code generation, ensure that:

- The top-level MATLAB Function block is called once per time step.
- It is possible to fully unroll program loops.
- Persistent variables with reset values and update logic are used to hold values across simulation time steps.
- Quantized data variables are used inside loops.

Model Bitwise and Bit Conversion Operations

The MATLAB Function block supports a variety of bitwise operations useful for hardware bit manipulation operations, such as bits-to-integer conversion, integer-to-bits conversion, bit concatenation, bit packing and unpacking, and pn-sequence generation and bit-scramblers.

These bitwise functions are supported by HDL Coder:

- bitget, bitsliceget, bitconcat, bitset, bitcmp
- bitand, bitor, bitxor
- bitandreduce, bitorreduce, bitxorreduce
- bitshift, bitsll, bitsrl, bitsra, bitrol, bitror

When modeling pure logic and no math operations in the MATLAB Function block, use:

- Unsigned instead of signed input operands.
- Non-saturating fimath options to generate less hardware.
- wrap as the OverflowMode setting.
- floor as the RoundMode setting.

In the em_hdl_design_patterns library, the MATLAB Function block hdl_bit_ops in the Bit Twiddlers subsystem and the signal_distance and nibble_swap_with_slice_concat MATLAB Function blocks in the Word Twiddlers subsystem show how to apply these settings.

For an example model that shows how you can perform a conversion from integer-to-bits and bits-to-integer, open the model hdlcoder_int2bits_bits2int, by entering:

```
open_system('hdlcoder_int2bits_bits2int')
```

You can also open the model by double-clicking on the black text box hdlcoder_int2bits_bits2int in the Word Twiddlers subsystem.

The hdlcoder_int2bits_bits2int model uses these MATLAB Function blocks from the Word Twiddlers subsystem in the em_hdl_design_patterns library:

- Bits2Int
- Int2Bits
- Integer to Bits
- Bits to Integer

Model Hardware Elements

You can use the MATLAB Function block to model various hardware elements like barrel shifters, rotators, and carry-save adders by using MATLAB scripts. In the em_hdl_design_patterns library, see the Shift Registers subsystem for MATLAB Function blocks that model various shift register hardware elements, such as 32-bit and 64-bit shift registers.

See Also

“Check for MATLAB Function block settings” on page 37-18

More About

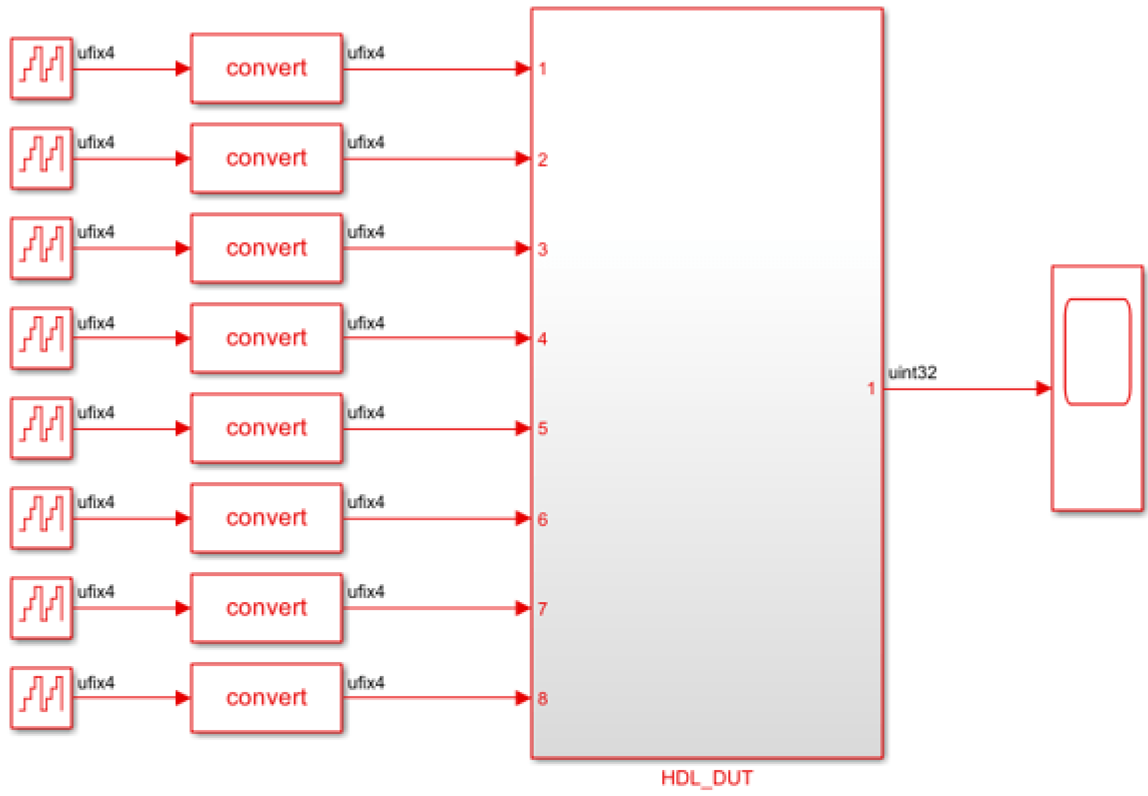
- “MATLAB Function Block Design Patterns for HDL” on page 27-14
- “Design Guidelines for the MATLAB Function Block” on page 27-19
- “Generate HDL Code from a MATLAB Function Block” on page 27-4

Use Distributed Pipelining Optimization in Models with MATLAB Function Blocks

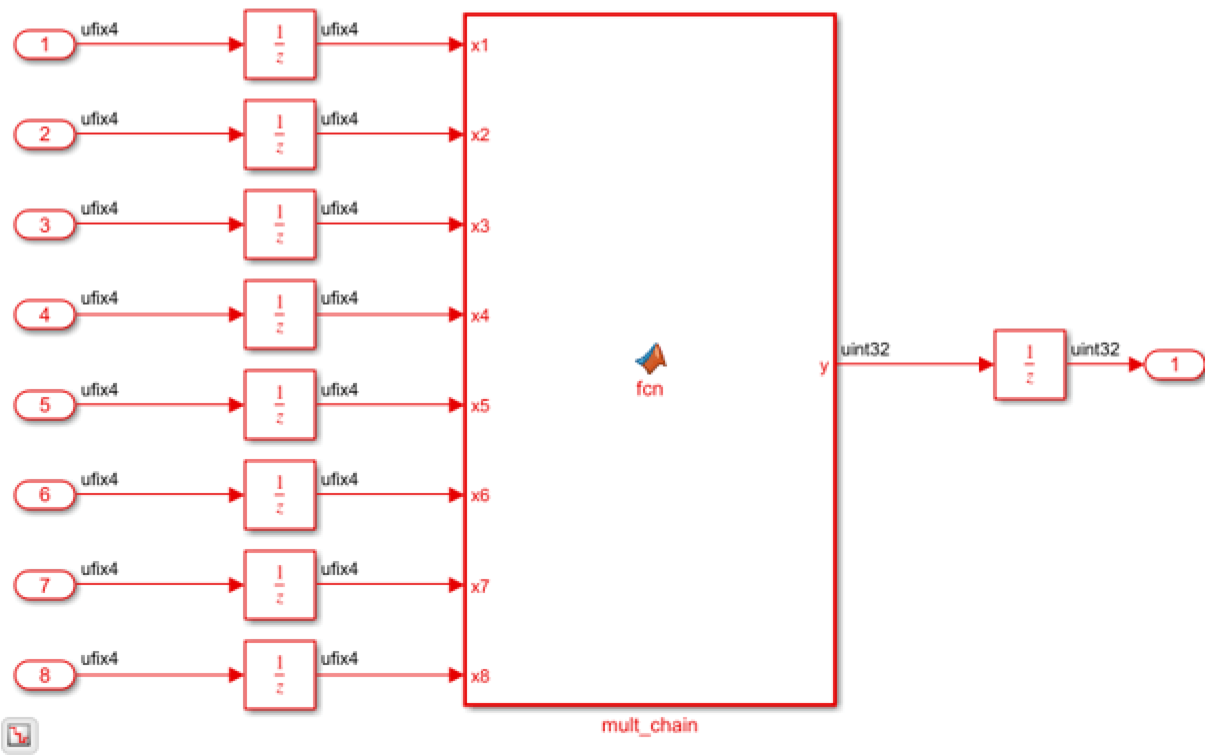
This example shows how to optimize the generated HDL code for MATLAB Function blocks by using the distributed pipelining optimization. Distributed pipelining is an HDL Coder™ optimization that improves the generated HDL code from MATLAB Function blocks, Simulink® models, or Stateflow® charts. By using distributed pipelining, your design achieves higher clock rates on the FPGA device. For more information, see “Distributed Pipelining” on page 21-130.

Open Multiplier Chain Model

This example shows how to distribute pipeline registers in a model that chains five multiplications.



The HDL_DUT subsystem is the DUT for which you want to generate HDL code. The subsystem drives a MATLAB Function block mult_chain.



To see the chain of multiplications, open the MATLAB Function block. The block contains this MATLAB code:

```
function y = fcn(x1,x2,x3,x4,x5,x6,x7,x8)
% A chained multiplication:
% y = (x1*x2)*(x3*x4)*(x5*x6)*(x7*x8)

y1 = x1 * x2;
y2 = x3 * x4;
y3 = x5 * x6;
y4 = x7 * x8;

y5 = y1 * y2;
y6 = y3 * y4;

y = y5 * y6;
```

Apply Distributed Pipelining Optimization

1. Specify generation of two pipeline stages for the MATLAB Function block. Right-click the `mult_chain` MATLAB Function block. Select **HDL Code** > **HDL Block Properties**. In the HDL Properties window, set **OutputPipeline** to 2.

2. Set **Architecture** to `MATLAB Datapath`. This architecture treats the MATLAB Function block like a regular subsystem. You can then apply various HDL Coder optimizations on MATLAB Function blocks along with other blocks in your Simulink model.

3. To enable the distributed pipelining optimization across hierarchies in your model, in the Configuration Parameters dialog box, click **HDL Code Generation** > **Optimization**. Click the **Pipelining** tab, then select the model configuration parameter **Distributed pipelining**.

4. To see the results of the optimization in the code generation report, click **HDL Code Generation** > **Report**. Then, enable the parameter **Generate optimization report**.

5. Optionally, when you use the MATLAB datapath architecture for a MATLAB Function block, you can enable synthesis timing estimates for distributed pipelining. Synthesis timing estimates calculate the propagation delays of the components in your design for distributed pipelining. Using this parameter, you can more accurately reflect how components function on hardware to better distribute pipelines in your design and maximize the clock frequency for your specific target device. For more information, see “Distributed Pipelining Using Synthesis Timing Estimates” on page 21-136.

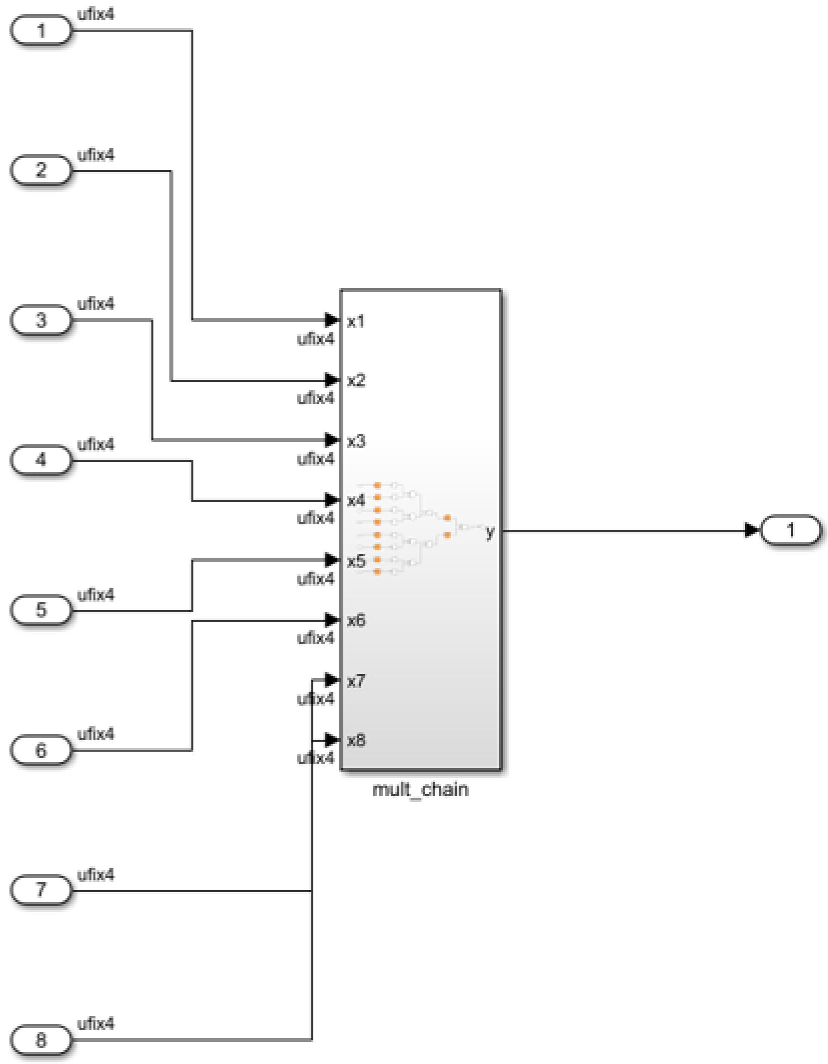
To enable synthesis timing estimates, click **HDL Code Generation** > **Optimization**, then select the **Pipelining** tab. Enable the **Use synthesis estimates for distributed pipelining** parameter.

6. Generate HDL Code for the `HDL_DUT` subsystem. In the **Apps** tab, click **HDL Coder**. In the **HDL Code** tab, click **Generate HDL Code**.

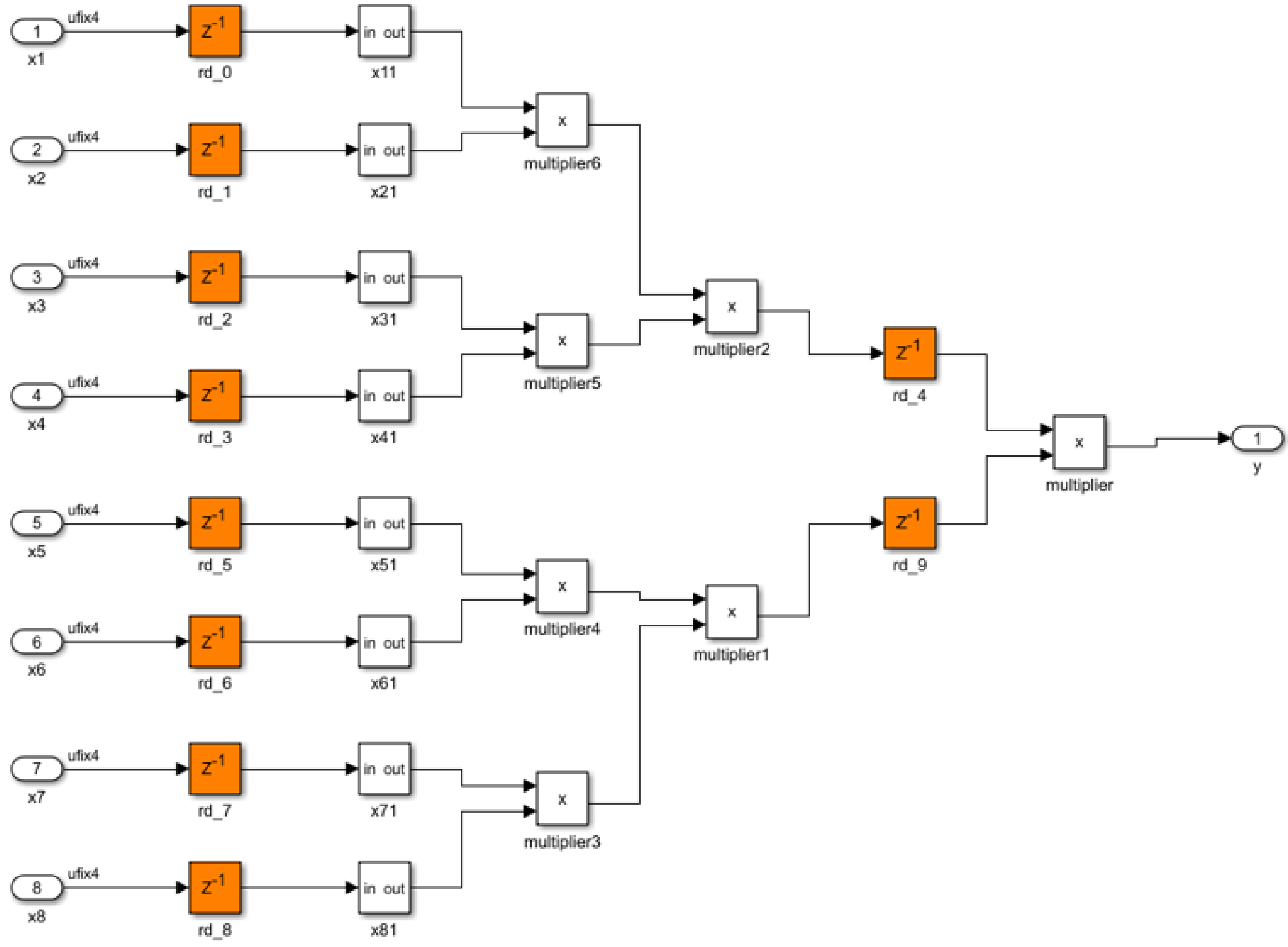
By default, HDL Coder generates VHDL code in the `hdlsrc` folder.

Analyze Results of Optimization

In the Code Generation Report window, in the **Distributed Pipelining** section, you see that the code generator moved the pipeline registers. To see the effects of the optimization, open the generated model `gm_hdlcoder_distpipe_multiplier_chain` and navigate to the `HDL_DUT` subsystem.



The MATLAB datapath architecture replaces the MATLAB Function block with a subsystem. The optimization can then distribute the pipeline registers and the unit delay inside the subsystem to optimize the multiplier chain and improve timing. Open the `mult_chain` subsystem to view this optimization.



See Also

Distributed pipelining | Generate optimization report | Use synthesis estimates for distributed pipelining

More About

- "Design Guidelines for the MATLAB Function Block" on page 27-19
- "Generate HDL Code from a MATLAB Function Block" on page 27-4
- "MATLAB Function Block Design Patterns for HDL" on page 27-14
- "Generate DUT Ports for Tunable Parameters" on page 14-18

Generating Scripts for HDL Simulators and Synthesis Tools

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 28-2
- “Structure of Generated Script Files” on page 28-3
- “Properties for Controlling Script Generation” on page 28-4
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 28-7
- “Add Synthesis Attributes” on page 28-17
- “Configure Synthesis Project Using Tcl Script” on page 28-18

Generate Scripts for Compilation, Simulation, and Synthesis

You can enable or disable script generation and customize the names and content of generated script files using either of the following methods:

- Use the `makehdl` or `makehdltb` functions, and pass in property name/property value arguments, as described in “Properties for Controlling Script Generation” on page 28-4.
- Set script generation options in the **HDL Code Generation > EDA Tool Scripts** pane of the Configuration Parameters dialog box, as described in “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 28-7.

Structure of Generated Script Files

A generated EDA script consists of an initialization phase, command-per-file phase, and a termination phase.

The sections are generated and executed in the following order:

1. An initialization (`Init`) phase. The `Init` phase performs the required setup actions, such as creating a design library or a project file. Some arguments to the `Init` phase are implicit, for example, the top-level entity or module name.
2. A command-per-file phase (`Cmd`). This phase of the script is called iteratively, once per generated HDL file or once per signal. On each call, a different file or signal name is passed in.
3. A termination phase (`Term`). This is the final execution phase of the script. One application of this phase is to execute a simulation of HDL code that was compiled in the `Cmd` phase. The `Term` phase does not take arguments.

The HDL Coder software generates scripts by passing format strings to the `fprintf` function. Using the GUI options (or `makehdl` and `makehdltb` properties) summarized in the following sections, you can pass in customized format names to the script generator. Some of these format names take arguments, such as the top-level entity or module name, or the names of the VHDL, Verilog or SystemVerilog files in the design.

You can use valid `fprintf` formatting characters. For example, `'\n'` inserts a newline into the script file.

Properties for Controlling Script Generation

Enable or disable script generation and customize the names and content of generated script files by using the `makehdl` or `makehdltb` functions.

Enabling and Disabling Script Generation

The `EDAScriptGeneration` property controls the generation of script files. By default, `EDAScriptGeneration` is set `on`. To disable script generation, set `EDAScriptGeneration` to `off`, as in the following example.

```
makehdl('sfir_fixed/symmetric_fir','EDAScriptGeneration','off')
```

Customizing Script Names

When you generate HDL code, HDL Coder appends a postfix string to the model or subsystem name *`system`* in the generated script name.

When you generate test bench code, HDL Coder appends a postfix string to the test bench name *`testbench_tb`*.

The postfix string depends on the type of script (compilation, simulation, or synthesis) being generated. The default postfix strings are shown in the following table. For each type of script, you can define your own postfix using the associated property.

Script Type	Property	Default Value
Compilation	`HDLCompileFilePostfix`	`_compile.do`
Simulation	`HDLSimFilePostfix`	`_sim.do`
Synthesis	`HDLSynthFilePostfix`	Depends on the selected synthesis tool. See Choose synthesis tool.

The following command generates VHDL code for the subsystem `system`, specifying a custom postfix for the compilation script. The name of the generated compilation script will be `system_test_compilation.do`.

```
makehdl('mymodel/system', 'HDLCompileFilePostfix', '_test_compilation.do')
```

Customizing Script Code

Using the property name/property value pairs summarized in the following table, you can pass in customized format names as character vectors to `makehdl` or `makehdltb`. The properties are named according to the following conventions:

- Properties that apply to the initialization (`Init`) phase are identified by the `Init` character vector in the property name.
- Properties that apply to the command-per-file phase (`Cmd`) are identified by the `Cmd` character vector in the property name.
- Properties that apply to the termination (`Term`) phase are identified by the `Term` character vector in the property name.

Property Name and Default	Description
Name: `HDLCompileInit` Default:`'vlib %s\n'`	Format name passed to `fprintf` to write the `Init` section of the compilation script. The implicit argument is the contents of the `VHDLLibraryName` property, which defaults to`'work'`. You can override the default `Init` string (`'vlib work\n'`) by changing the value of `VHDLLibraryName`.
Name: `HDLCompileVHDLCmd` Default: `'vcom %s %s\n'`	Format name passed to `fprintf` to write the `Cmd` section of the compilation script for VHDL files. The two implicit arguments are the contents of the `SimulatorFlags` property and the file name of the current entity or module. To omit the flags, set `SimulatorFlags` to `''` (the default).
Name: `HDLCompileVerilogCmd` Default: `'vlog %s %s\n'`	Format name passed to `fprintf` to write the `Cmd` section of the compilation script for Verilog files. The two implicit arguments are the contents of the `SimulatorFlags` property and the file name of the current entity or module. To omit the flags, set `SimulatorFlags` to `''` (the default).
Name:`HDLCompileTerm` Default:`''`	Format name passed to `fprintf` to write the termination portion of the compilation script.

Name: `HDLSimInit`
 	
Default:
 	
`['onbreak resume\n',...`
`'onerror resume\n']`
	Format name passed to `fprintf` to write the initialization section of the simulation script.

Name: `HDLSimCmd` Default: `'vsim -voptargs=+acc %s.%s\n'`	Format name passed to `fprintf` to write the simulation command. If your target language is VHDL, the first implicit argument is the value of the `VHDLLibraryName` property. If your target language is Verilog or SystemVerilog, the first implicit argument is `'work'`. The second implicit argument is the top-level module or entity name.
Name: `HDLSimViewWaveCmd` Default: `'add wave sim:%s\n'`	Format name passed to `fprintf` to write the simulation script waveform viewing command. The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.
Name: `HDLSimTerm` Default: `'run -all\n'`	Format name passed to `fprintf` to write the `Term` portion of the simulation script. The string is a synthesis project creation command. The content of the string is specific to the selected synthesis tool. See Choose synthesis tool.
Name: `HDLSynthInit`	Format name passed to `fprintf` to write the `Init` section of the synthesis script. The content of the format name is specific to the selected synthesis tool. See Choose synthesis tool.

Property Name and Default	Description
Name: `HDLSynthCmd`	Format name passed to `fprintf` to write the `Cmd` section of the synthesis script. The content of the format name is specific to the selected synthesis tool. See Choose synthesis tool.
Name: `HDLSynthTerm`	Format name passed to `fprintf` to write the `Term` section of the synthesis script. The content of the format name is specific to the selected synthesis tool. See Choose synthesis tool.

Examples

The following example specifies a custom VHDL library name for the Mentor Graphics ModelSim compilation script for code generated from the subsystem, `system`.

```
makehdl(system, 'VHDLLibraryName', 'mydesignlib')
```

The resultant script, `system_compile.do`, is:

```
vlib mydesignlib
vcom  system.vhd
```

The following example specifies that HDL Coder generate a Xilinx ISE synthesis file for the subsystem `sfir_fixed/symmetric_fir`.

```
 makehdl('sfir_fixed/symmetric_fir','HDLSynthTool', 'ISE')
```

The following listing shows the resultant script, `symmetric_fir_ise.tcl`.

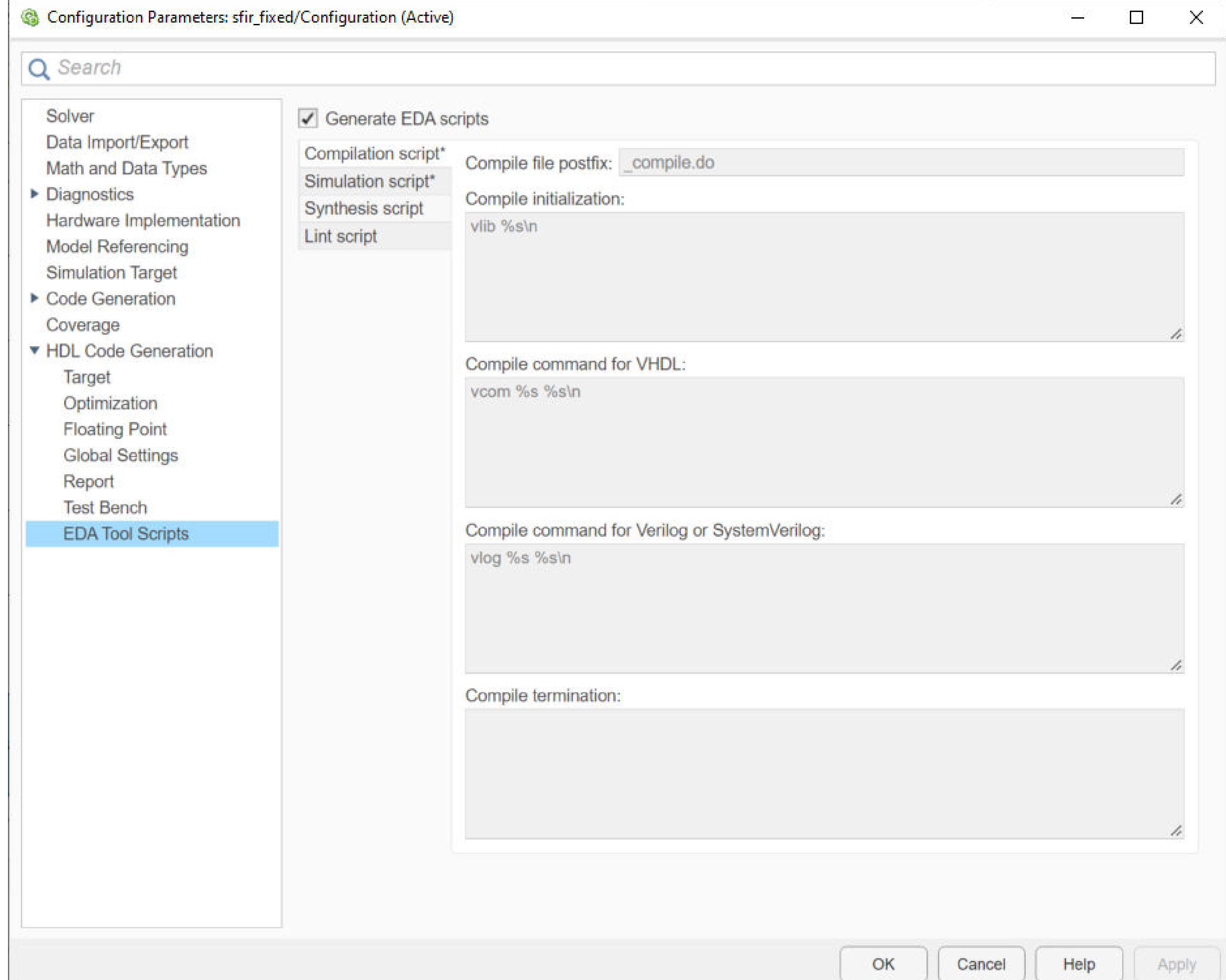
```
set src_dir "./hdlsrc"
set prj_dir "synprj"
file mkdir ../$prj_dir
cd ../$prj_dir
project new symmetric_fir.ise
xfile add ../$src_dir/symmetric_fir.vhd
project set family Virtex4
project set device xc4vsx35
project set package ff668
project set speed -10
process run "Synthesize - XST"
```

Configure Compilation, Simulation, Synthesis, and Lint Scripts

Configure the script file generation by using the **EDA Tool Scripts** pane. These options correspond to the properties described in “Properties for Controlling Script Generation” on page 28-4.

To view and set **EDA Tool Scripts** options:

1. Open the Configuration Parameters dialog box.
2. Select the **HDL Code Generation** > **EDA Tool Scripts** pane.



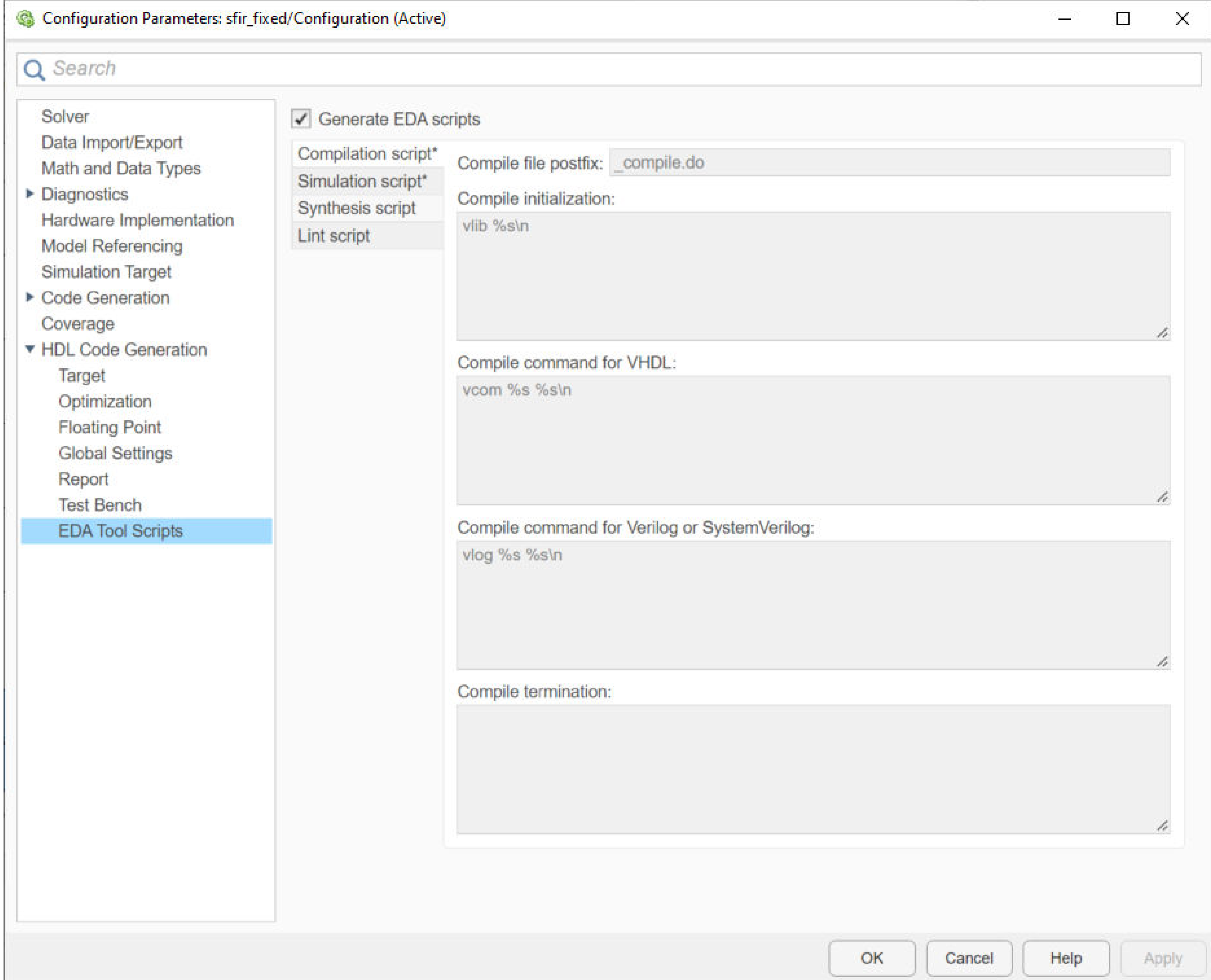
3. The **Generate EDA scripts** option controls the generation of script files. By default, this option is selected.

   If you want to disable script generation, clear this check box and click **Apply**.
4. The list on the left of the **EDA Tool Scripts** pane lets you select from several categories of options. Select a category and set the options as desired. The categories are:

- **Compilation script**: Options related to customizing scripts for compilation of generated VHDL, Verilog or SystemVerilog code. For more information, see “Compilation Script Options” on page 28-8.
- **Simulation script**: Options related to customizing scripts for HDL simulators. For more information, see “Simulation Script Options” on page 28-9.
- **Synthesis script**: Options related to customizing scripts for synthesis tools. For more information, see “Synthesis Script Options” on page 28-11.
- **Lint script**: Options related to customizing scripts for HDL lint tools. For more information, see “Lint Script Options” on page 28-14.

Compilation Script Options

The following figure shows the **Compilation script** pane, with options set to their default values.

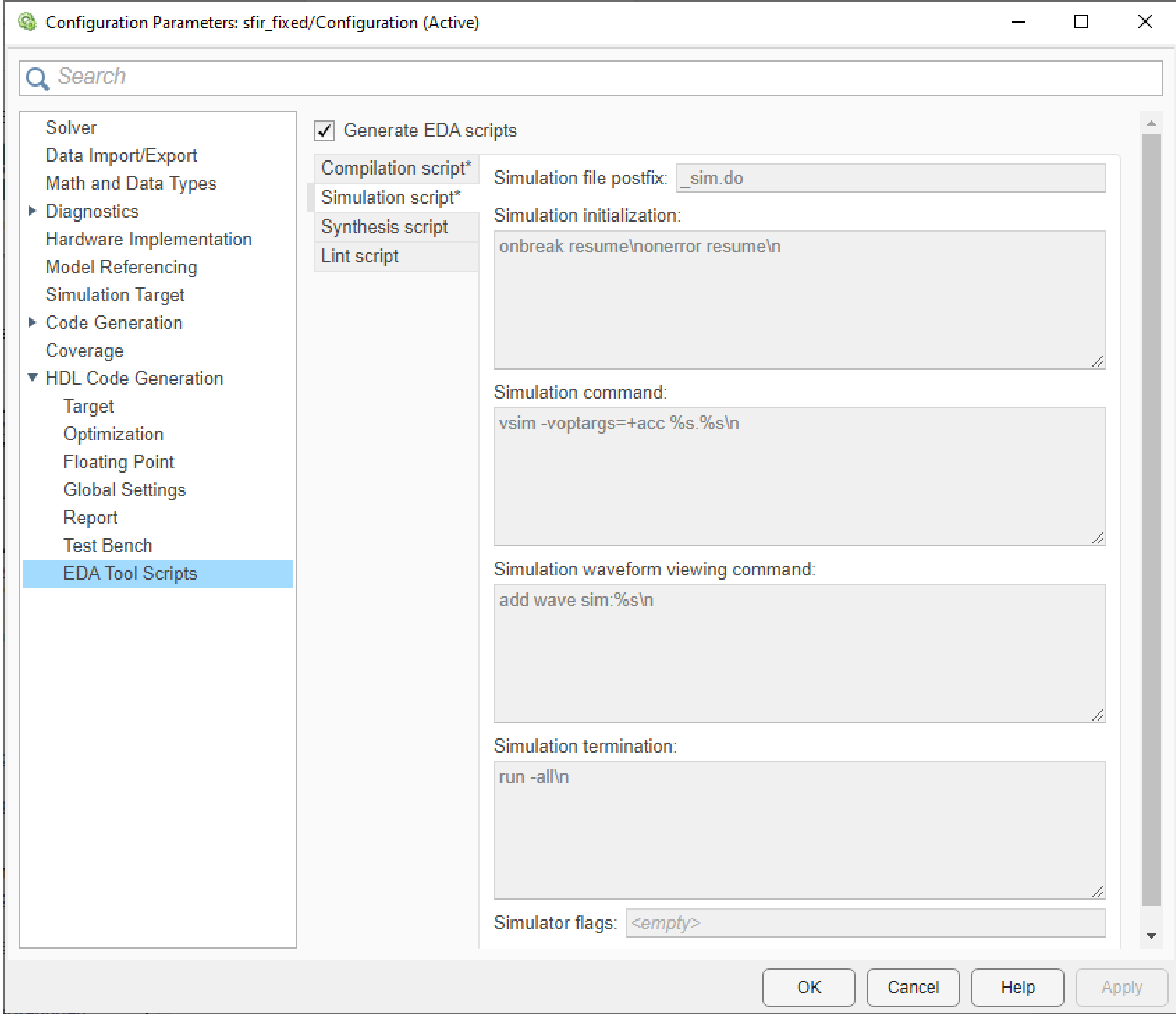


The following table summarizes the **Compilation script** options.

Option and Default	Description
Name: **Compile file postfix** Default: `'_compile.do'`	Postfix appended to the DUT name or test bench name to form the script file name.
Name: **Compile initialization** Default: `'vlib %s\n'`	Format name passed to `fprintf` to write the `Init` section of the compilation script. The argument is the contents of the `VHDLLibraryName` property, which defaults to`'work'`. You can override the default `Init 'vlib work\n'` by changing the value of `VHDLLibraryName`.
Name: **Compile command for VHDL** Default: `'vcom %s %s\n'`	Format name passed to `fprintf` to write the `Cmd` section of the compilation script for VHDL files. The two arguments are the contents of the `SimulatorFlags` property option and the filename of the current entity or module. To omit the flags, set `SimulatorFlags` to `''` (the default).
Name: **Compile command for Verilog or SystemVerilog** Default: `'vlog %s %s\n'`	Format name passed to `fprintf` to write the `Cmd` section of the compilation script for Verilog or SystemVerilog files. The two arguments are the contents of the `SimulatorFlags` property and the filename of the current entity or module. To omit the flags, set `SimulatorFlags` to `''` (the default).
Name: **Compile termination** Default:`''`	Format name passed to `fprintf` to write the termination portion of the compilation script.

Simulation Script Options

The following figure shows the **Simulation script** pane, with options set to their default values.



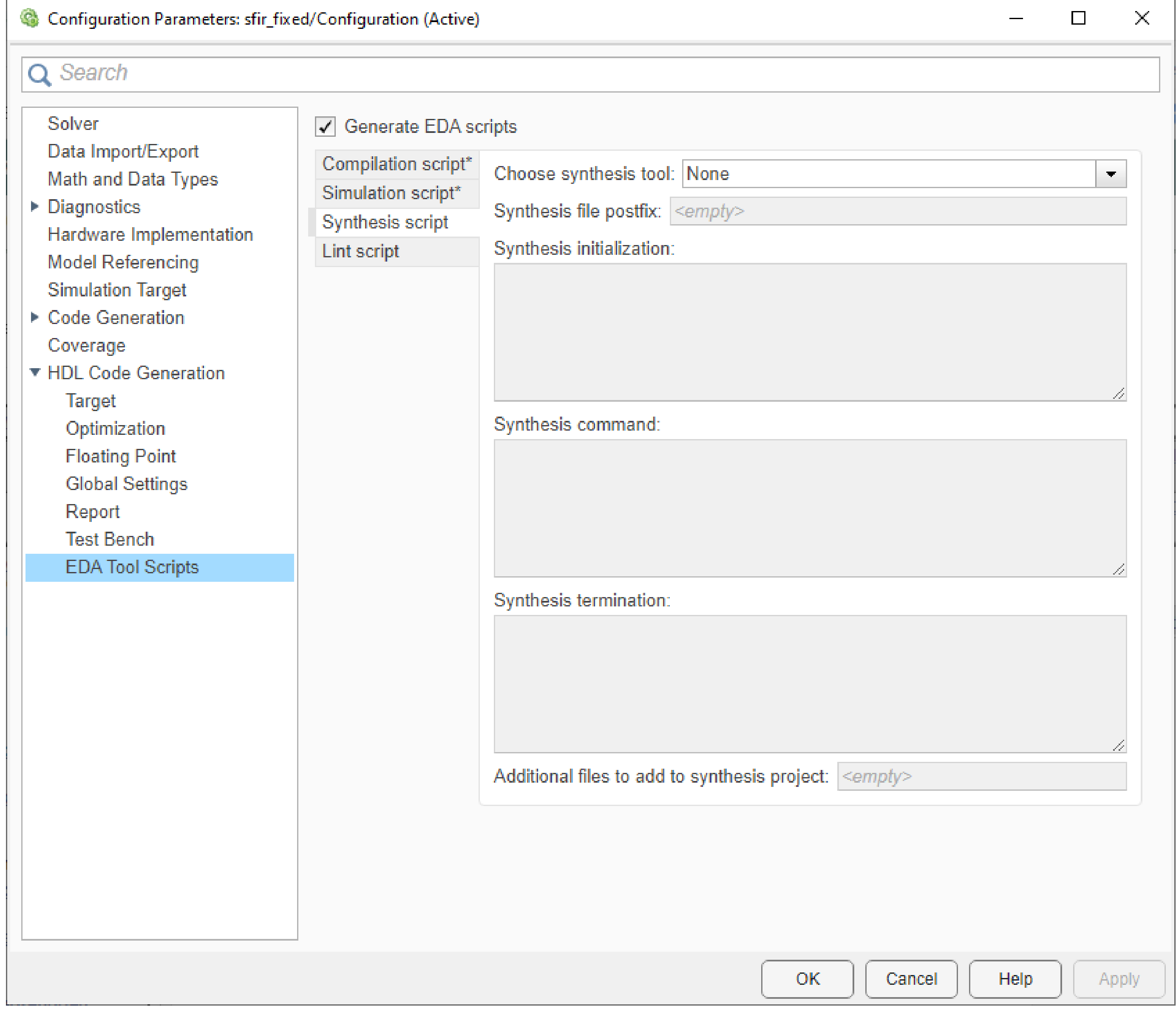
The following table summarizes the **Simulation script** options.

Option and Default	Description
Name: **Simulation file postfix** Default: `'_sim.do'`	Postfix appended to the model name or test bench name to form the simulation script file name.
Simulation initialization Default: `['onbreak resume\nonerror resume\n']`	Format name passed to `fprintf` to write the initialization section of the simulation script.

Option and Default	Description
Simulation command Default: `'vsim -voptargs=+acc %s.%s \n'`	Format name passed to `fprintf` to write the simulation command. If your `TargetLanguage` is `'VHDL'`, the first implicit argument is the value of `VHDLLibraryName`. If your `TargetLanguage` is `'Verilog'`or `'SystemVerilog'`, the first implicit argument is `'work'`. The second implicit argument is the top-level module or entity name.
Simulation waveform viewing command Default: `'add wave sim:%s\n'`	Format name passed to `fprintf` to write the simulation script waveform viewing command. The top-level module or entity signal names are implicit arguments.
Simulation termination Default: `'run -all\n'`	Format name passed to `fprintf` to write the `Term` portion of the simulation script.
Simulator flags Default: `<empty>`	Specify options that are specific to your application and the simulator you are using. For example, if you use the 1076-1993 VHDL compiler, specify the flag `-93`. The coder adds the flags you specify with this option to the compilation command in the generated EDA tool scripts.

Synthesis Script Options

The following figure shows the **Synthesis script** pane, with options set to their default values. The **Choose synthesis tool** property defaults to `None`, which disables generation of a synthesis script.

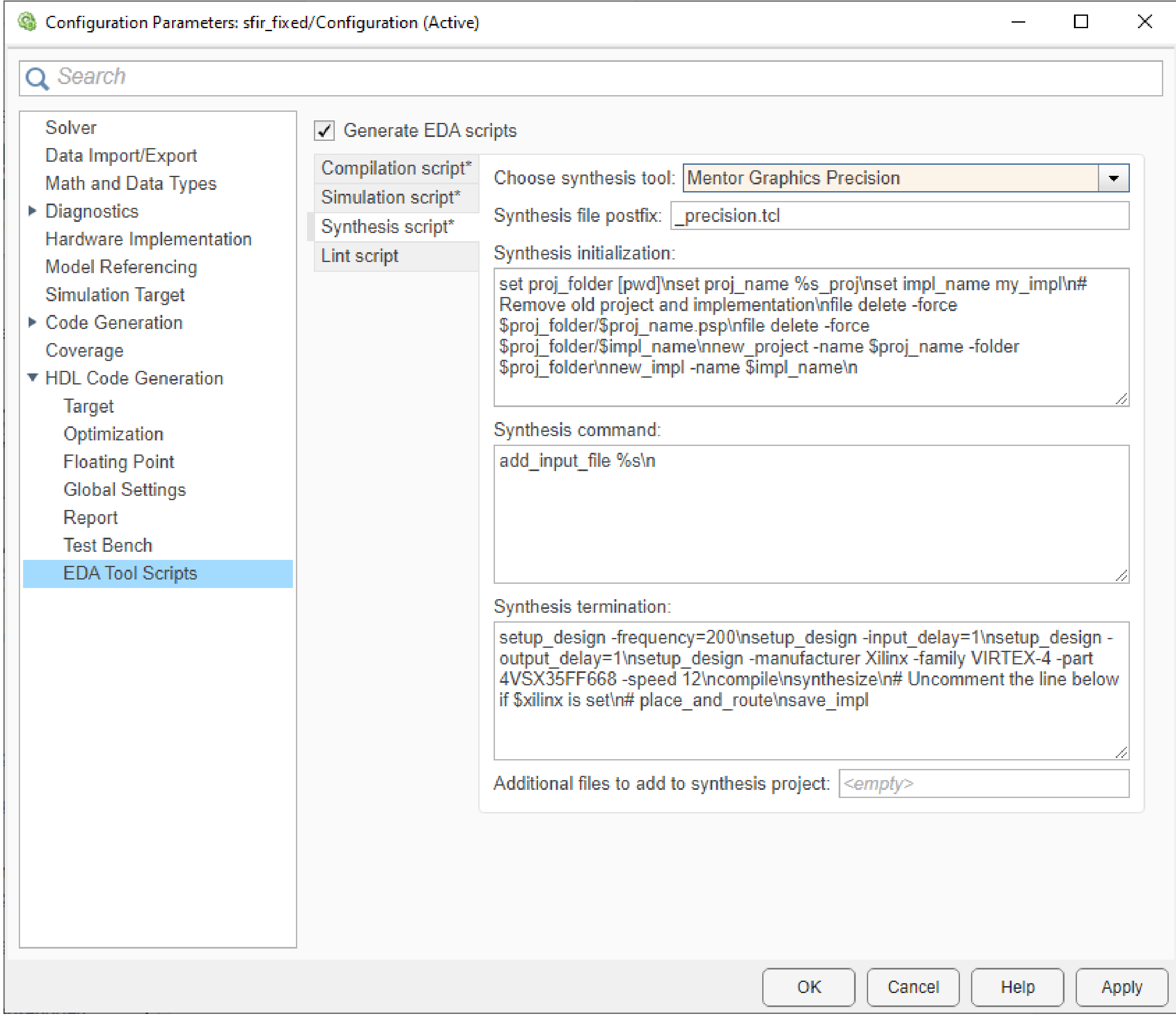


To enable synthesis script generation, select a synthesis tool from the **Choose synthesis tool** menu.

When you select a synthesis tool, HDL Coder:

- Enables synthesis script generation.
- Enters a file name postfix (specific to the chosen synthesis tool) into the **Synthesis file postfix** field.
- Enters strings (specific to the chosen synthesis tool) into the initialization, command, and termination fields.

The following figure shows the default option values entered for the Mentor Graphics Precision tool.



The following table summarizes the **Synthesis script** options.

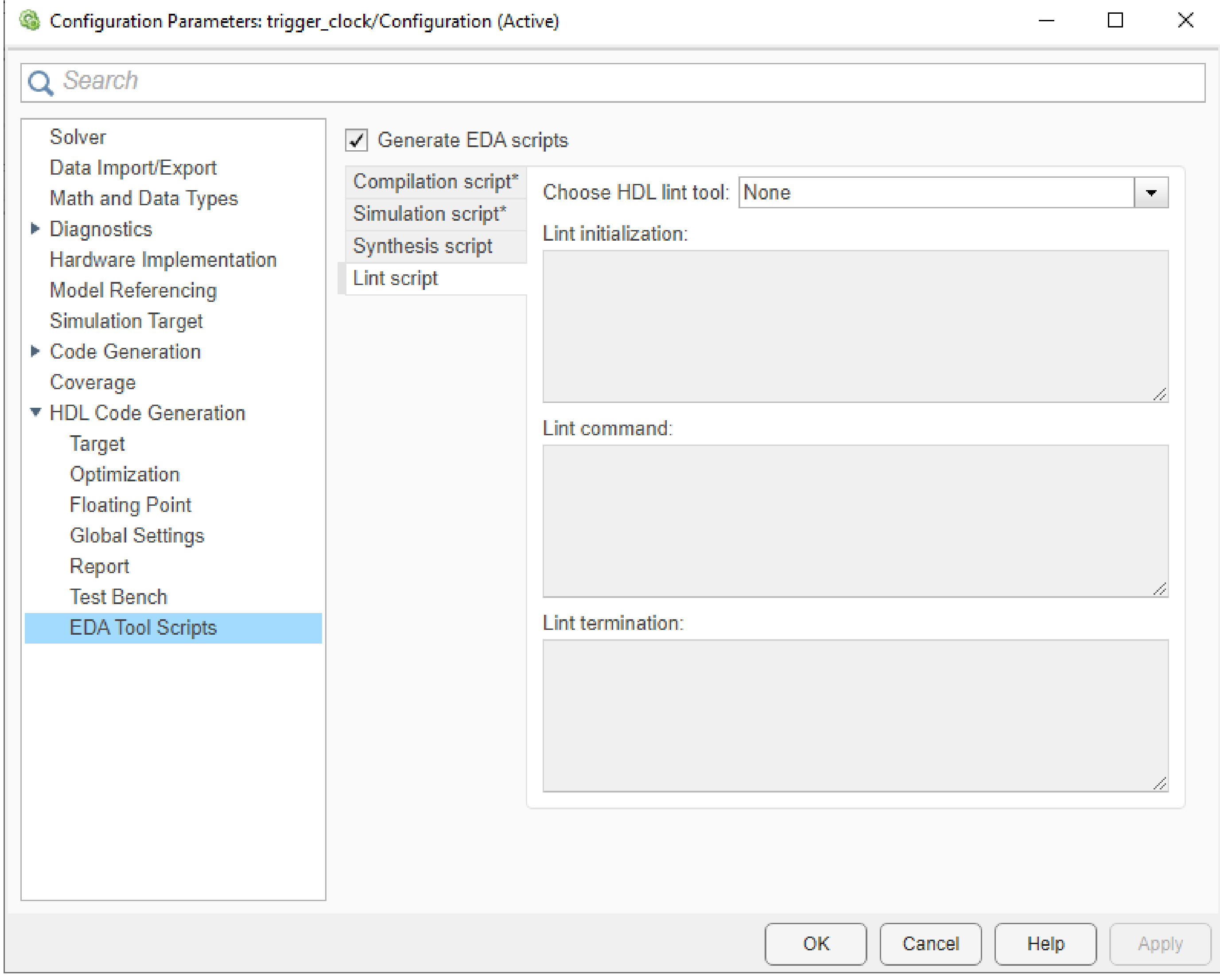
Option Name	Description
Choose synthesis tool	`None` (default): do not generate a synthesis script `Xilinx ISE`: generate a synthesis script for Xilinx ISE `Microchip Libero`: generate a synthesis script for Microchip Libero `Mentor Graphics Precision`: generate a synthesis script for Mentor Graphics Precision `Altera Quartus II`: generate a synthesis script for Altera Quartus II `Synopsys Synplify Pro`: generate a synthesis script for Synopsys Synplify Pro `Xilinx Vivado`: generate a synthesis script for Xilinx Vivado `Custom`: generate a custom synthesis script

Synthesis file postfix	Your choice of synthesis tool sets the postfix for generated synthesis file names to one of the following:
	`_ise.tcl`
	`_libero.tcl`
	`_precision.tcl`
	`_quartus.tcl`
	`_synplify.tcl`
	`_vivado.tcl`
	`_custom.tcl`

Synthesis initialization	Format name passed to `fprintf` to write the `Init` section of the synthesis script. The default string is a synthesis project creation command. The implicit argument is the top-level module or entity name. The content of the string is specific to the selected synthesis tool.
Synthesis command	Format name passed to `fprintf` to write the `Cmd` section of the synthesis script. The implicit argument is the file name of the entity or module. The content of the string is specific to the selected synthesis tool.
Synthesis termination	Format name passed to `fprintf` to write the `Term` section of the synthesis script. The content of the string is specific to the selected synthesis tool.

Lint Script Options

The following figure shows the **Lint script** pane, with options set to their default values.



The following table summarizes the **Lint script** options.

Option	Description
Choose HDL lint tool	Enable or disable generation of an HDL lint script, and select the HDL lint tool for which HDL Coder generates a script.
Lint initialization	Enter an initialization text for your HDL lint script.
Lint command	Enter the command for your HDL lint script.
Lint termination	Enter a termination character vector for your HDL lint script.

To enable lint script generation, select a HDL lint tool from the **Choose HDL lint tool** menu.

When you select the HDL lint tool, HDL Coder:

- Enables lint script generation.

- Enters strings (specific to the chosen HDL lint tool) into the initialization, command, and termination fields.

You can also generate the custom lint script. Select `Custom` from the **Choose HDL lint tool** and enter strings into the initialization, command, and termination fields.

For more information, see “Generate HDL Lint Tool Script” on page 24-49.

Add Synthesis Attributes

To learn how to add synthesis attributes in the generated HDL code for multiplier mapping, see “DSPStyle” on page 19-11.

Configure Synthesis Project Using Tcl Script

You can add a Tcl script that configures your synthesis project.

To configure your synthesis project using a Tcl script:

1. Create a Tcl script that contains commands to customize your synthesis project.

   For example, to specify the finite state machine style:

   - For Xilinx ISE, create a Tcl script that contains the following line:

     ```
     project set "FSM Encoding Algorithm" "Gray" -process "Synthesize - XST"
     ```
   - For Xilinx Vivado, create a Tcl script that contains the following line:

     ```
     set_property STEPS.SYNTH_DESIGN.ARGS.FSM_EXTRACTION gray [get_runs synth_1]
     ```
2. In the HDL Workflow Advisor, in the **FPGA Synthesis and Analysis > Create Project** task, in the **Additional source files** field, enter the full path to the Tcl file manually, or by using the **Add** button.

When HDL Coder creates the project, the Tcl script is executed to apply the synthesis project settings.

Using the HDL Workflow Advisor

- “Workflows in HDL Workflow Advisor” on page 29-2
- “Getting Started with the HDL Workflow Advisor” on page 29-5
- “Generate Code and Synthesize on FPGA Using HDL Workflow Advisor” on page 29-11
- “Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor” on page 29-16
- “Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries” on page 29-19
- “FPGA Floating-Point Library IP Mapping” on page 29-25
- “Customize Floating-Point IP Configuration” on page 29-36
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 29-41
- “Synthesis Objective to Tcl Command Mapping” on page 29-45
- “Run HDL Workflow with a Script” on page 29-47
- “Get Started with HDL Workflow Command-Line Interface” on page 29-56

Workflows in HDL Workflow Advisor

The HDL Workflow Advisor offers various workflows to check your algorithm for HDL compatibility, generate HDL code, verify the code, and then deploy the code to your target platform.

You can run the Workflow Advisor for your MATLAB algorithm or Simulink model. Before you deploy the code to a target hardware platform, install the synthesis tool and specify the path to that synthesis tool by using the `hdlsetuptoolpath` function. See “Tool Setup”.

HDL Workflow Advisor is not available in Simulink Online.

Set Up HDL Workflow Advisor in MATLAB

Before you specify the target workflow, when you run the Workflow Advisor from MATLAB, specify the design and test bench files, define the input types, and run fixed-point conversion.

To specify the target workflow for code generation:

1. On the MATLAB toolstrip, from the **Apps** tab, select the HDL Coder app.
2. Select the MATLAB design and test bench files and click the **Workflow Advisor** button.
3. In the **HDL Workflow Advisor**, select **Code Generation Workflow** as **MATLAB to HDL** or **MATLAB to SystemC**.
4. In the **Select Code Generation Target** task, select the **Workflow** for code generation.

Note The steps after code generation workflow selection change depending on your target workflow.

Set Up HDL Workflow Advisor in Simulink

When you run the Workflow Advisor from your Simulink model, irrespective of the target workflow, you run the steps to prepare the model for HDL code generation, and then generate code.

Open the Simulink model for which you want to run the workflow.

1. On the Simulink toolstrip, from the **Apps** tab, select the HDL Coder app.
2. On the **HDL Code** tab, click the **Workflow Advisor** button.
3. In the HDL Workflow Advisor, on the **Set Target Device and Synthesis Tool** task, select the **Target workflow**.

The steps in the Workflow Advisor change depending on the **Target workflow**, **Target platform**, and **Synthesis tool**.

Generic ASIC/FPGA

Generate HDL code from your Simulink model or MATLAB algorithm, verify the HDL code, and deploy the code to a generic ASIC or FPGA device. You can select from a family of devices that belong to these synthesis tools, as listed in “Generic ASIC/FPGA Hardware”.

By using this workflow, you can:

- Generate HDL code for your fixed-point MATLAB algorithm or your HDL-compatible Simulink model.
- Generate an HDL test bench and cosimulation test bench (requires HDL Verifier), and scripts to build and run the code and test bench. You can also generate a SystemVerilog DPI test benches and code coverage when running the Simulink HDL Workflow Advisor (requires HDL Verifier).
- Perform FPGA synthesis and timing analysis and rapidly prototype your design on generic FPGA platforms through integration with third-party synthesis tools.
- Back-annotate the model with critical path information and other information obtained during synthesis, and then optimize your design for area and speed.

Note If you select `Intel Quartus Pro` or `Microchip Libero SoC` as the **Synthesis tool**, the **Annotate Model with Synthesis Result** task is not available. To see the critical path, run the workflow to synthesis, and then open the timing reports.

To learn more, see:

- “HDL Code Generation and FPGA Synthesis from Simulink Model”
- “Basic HDL Code Generation and FPGA Synthesis from MATLAB”

IP Core Generation

Generate RTL code and a custom HDL IP core from your Simulink model or MATLAB algorithm. Before you run the workflow, partition your design into components that run on software and components that run on hardware. See “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9.

The IP core is a shareable and reusable HDL component that consists of IP core definition files, HDL code generated for your algorithm, C header file with the register address map, and the IP core report. See:

- “Custom IP Core Report” on page 39-20
- “Custom IP Core Generation” on page 39-17

You can select from one of these synthesis tools, as listed in “IP Core Generation Hardware”.

Use this workflow to:

- Generate a generic board-independent Xilinx or Intel HDL IP core.
- Integrate the IP core into a reference design to target standalone FPGA boards or SoC platforms with Xilinx Vivado IP integrator or Intel Qsys.
- Communicate with the generated HDL IP core by using embedded ARM processor or, from MATLAB, by using the HDL Verifier AXI Manager. See “Set Up AXI Manager” (HDL Verifier).

You can integrate the HDL IP core into HDL Coder provided reference designs such as the `default system reference design` or into a reference design that you created. To learn more, see:

- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270

Simulink Real-Time FPGA I/O

Generate HDL code from your Simulink model and deploy the code onto Speedgoat® FPGA I/O modules. This workflow requires Xilinx Vivado and uses the `IP Core Generation` workflow infrastructure, as mentioned in “Simulink Real-Time FPGA I/O: Speedgoat Target Computer”.

To run the `Simulink Real-Time FPGA I/O` workflow, install the Speedgoat I/O Blockset and the Speedgoat HDL Coder Integration Packages. After you install the integration packages, you can choose the **Target platform**, and then run the workflow to:

- Generate a reusable and shareable IP core.
- Integrate the IP core into the Speedgoat reference design.
- Generate an FPGA bitstream and download the bitstream to the target hardware.
- Generate a Simulink Real-Time™ model. The model is an interface subsystem model that contains the blocks to program the FPGA and communicate with the board during real-time execution.

For more information, see “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 40-145.

FPGA-in-the-Loop

Test your Simulink model or MATLAB algorithm on a target FPGA. This workflow requires HDL Verifier. You can select from one of these synthesis tools, as listed in “FPGA-in-the-Loop Hardware”.

Use this workflow to:

- Choose boards from the FPGA Board Manager that are **FIL Enabled** or create your own custom boards for verification. See “FPGA Board Customization” (HDL Verifier).
- Generate HDL code for your fixed-point MATLAB algorithm or your HDL-compatible Simulink model.
- Perform FPGA implementation and connect to the target FPGA board by using Ethernet, JTAG, or PCI Express for FIL simulation.

To learn more, see:

- “FIL Simulation with HDL Workflow Advisor for Simulink” (HDL Verifier)
- “FIL Simulation with HDL Workflow Advisor for MATLAB” (HDL Verifier)

See Also

`hdladvisor` | `makehdl`

More About

- “Getting Started with the HDL Workflow Advisor” on page 29-5
- “HDL Workflow Advisor Tasks” on page 36-2
- “Run HDL Workflow with a Script” on page 29-47
- “Get Started with HDL Workflow Command-Line Interface” on page 29-56

Getting Started with the HDL Workflow Advisor

In this section...

"Open the HDL Workflow Advisor" on page 29-5

"Run Tasks in the HDL Workflow Advisor" on page 29-6

"Fix HDL Workflow Advisor Warnings or Failures" on page 29-7

"Save and Restore the HDL Workflow Advisor State" on page 29-7

"View and Save HDL Workflow Advisor Reports" on page 29-8

The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

Open the HDL Workflow Advisor

To start the HDL Workflow Advisor from a Simulink model:

1. In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
2. Select the DUT Subsystem in your model, and make sure that this Subsystem name appears in the **Code for** option. To remember the selection, you can pin this option. Click **Workflow Advisor**.

To start the HDL Workflow Advisor for a model from the command line, enter `hdladvisor(`*`system`*`)`. *`system`* is a handle or name of the model or subsystem that you want to check. For more information, see the `hdladvisor` function reference page.

For how to use the HDL Workflow Advisor to generate HDL code from a MATLAB script, see "Basic HDL Code Generation and FPGA Synthesis from MATLAB".

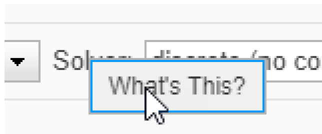
HDL Workflow Advisor is not available in Simulink Online.

In the HDL Workflow Advisor, the left pane lists the folders in the hierarchy. Each folder represents a group or category of related tasks. Expanding the folders shows available tasks in each folder. From the left pane, you can select a folder or an individual task. The HDL Workflow Advisor displays information about the selected folder or task in the right pane. The contents of the right pane depend on the selected folder or task. For some tasks, the right pane contains simple controls for running the task and a display area for status messages and other task results. For other tasks that involve setting code or test bench generation parameters, the right pane displays several parameter and option settings.

Run Tasks in the HDL Workflow Advisor

In the HDL Workflow Advisor window, you can run individual tasks, a group of tasks, or all the tasks in the workflow. To run a task, all tasks before it must have run successfully.

To learn more about each individual task, right-click that task, and select **What's This?**



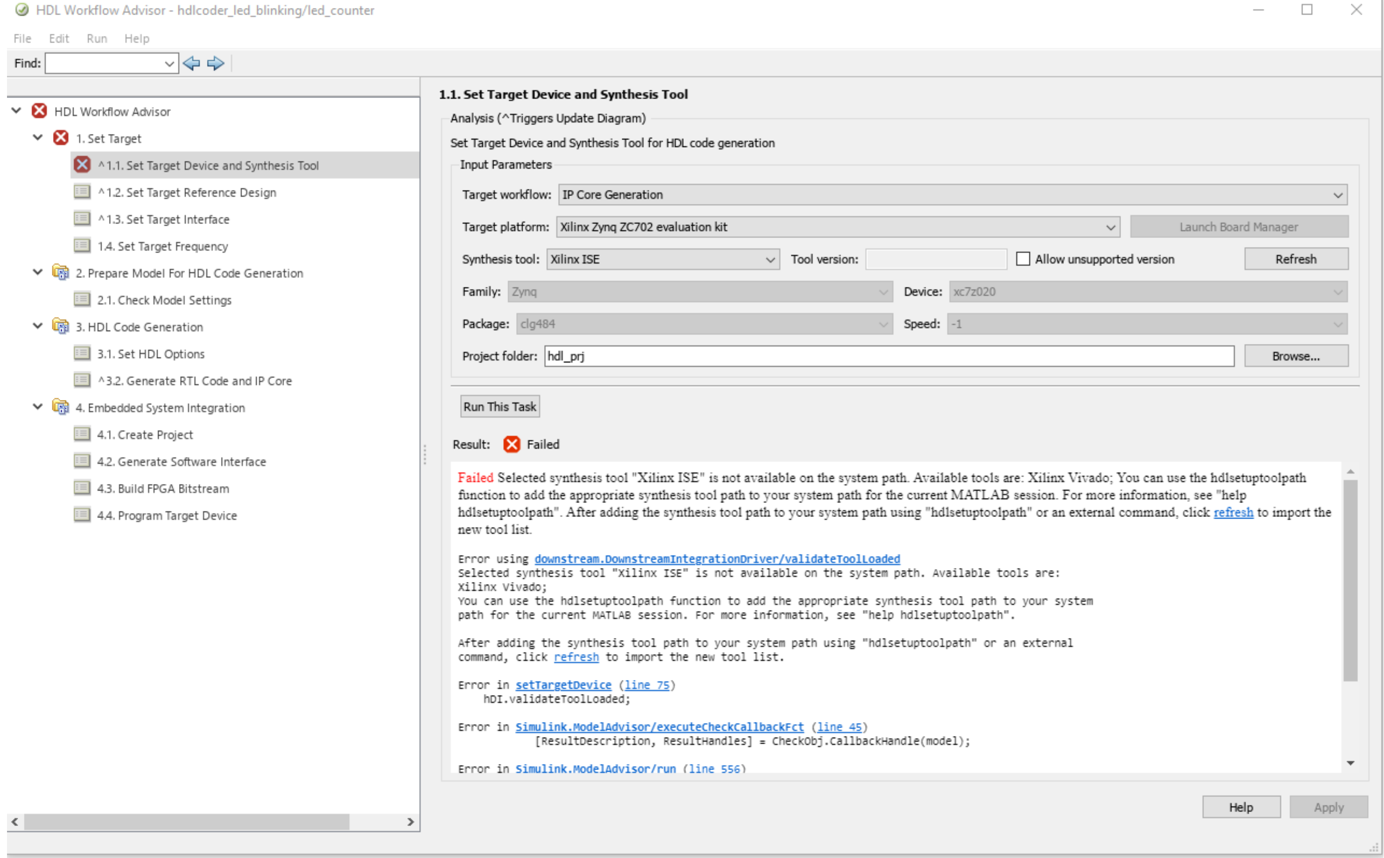
To generate HDL code, run the workflow to the **Generate RTL Code and Testbench** task. To run the workflow to a specific task inside a subfolder, expand that folder, and then right-click the task and select **Run To Selected Task**.

To rerun a task that you have already run, click **Reset This Task**. HDL Coder then resets the task and all tasks that follow it. For example, to customize the basic options for generating HDL code after running the **Generate RTL Code and Testbench** task, right-click the **Set Basic Options** task and select **Reset This Task**. You can then set the basic options on the model and click **Run This Task** to rerun the task.

To run all the tasks in the HDL Workflow Advisor with the default settings, in the HDL Workflow Advisor window, select **Run** > **Run All**. To run a group of tasks in a specific folder, select that folder and click **Run All**.

Fix HDL Workflow Advisor Warnings or Failures

In the HDL Workflow Advisor, if a task terminates due to a warning or failure condition, the right pane shows the warning or failure information in a **Result** subpane. The **Result** subpane displays model settings that you can use to fix the warnings. For some tasks, use the **Action** subpane to apply those recommended actions.



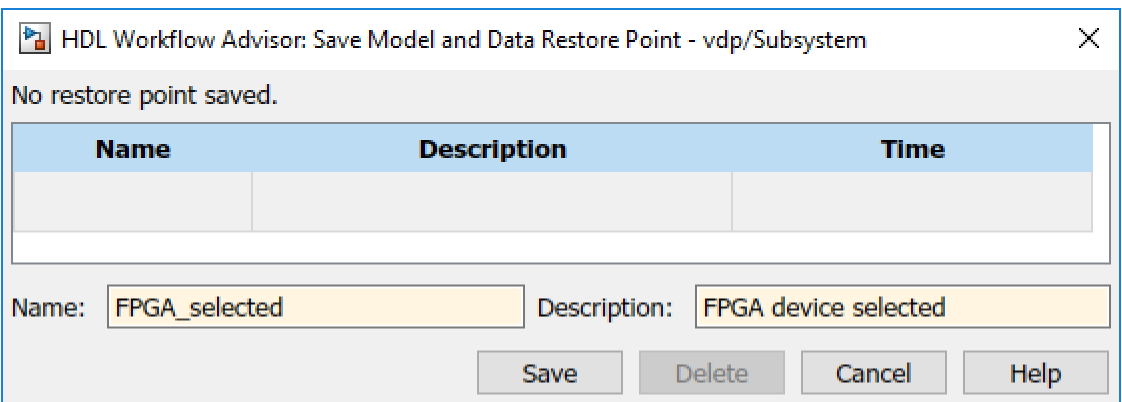
Save and Restore the HDL Workflow Advisor State

By default, the HDL Coder software saves the state of the most recent HDL Workflow Advisor session. The next time that you activate the HDL Workflow Advisor, it returns to that state. You can also save the current settings of the HDL Workflow Advisor to a named restore point. Later, you can restore the same settings by loading the restore point data into the HDL Workflow Advisor.

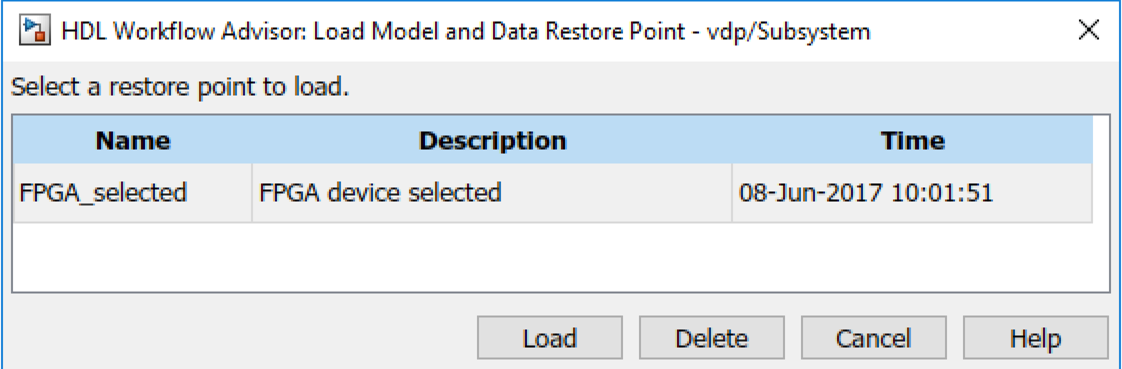
The save and restore process does not:

- Include operations that you perform outside the HDL Workflow Advisor.
- Save or restore the state of HDL Workflow Advisor tasks involving third-party tools.

To save the Workflow Advisor state, in the HDL Workflow Advisor Window, select **File** > **Save Restore Point As**. Enter a **Name** and **Description**, and then click **Save**. You can save more than one restore point.

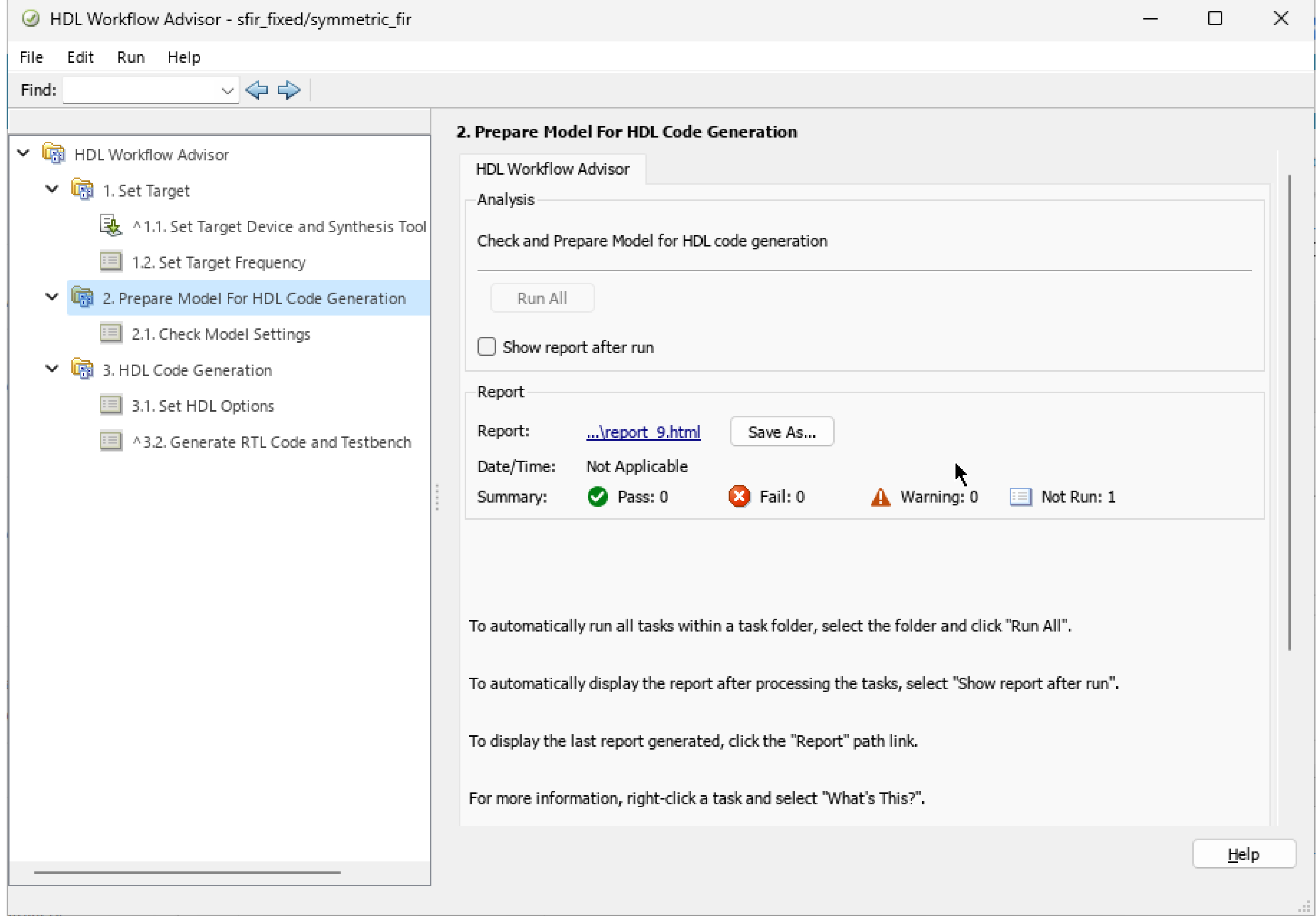


To restore a Workflow Advisor state, in the HDL Workflow Advisor, select **File** > **Load Restore Point**. Select the restore point that you want to load and click **Load**. When you load a restore point, the HDL Workflow Advisor warns that the restoration overwrites the current settings.

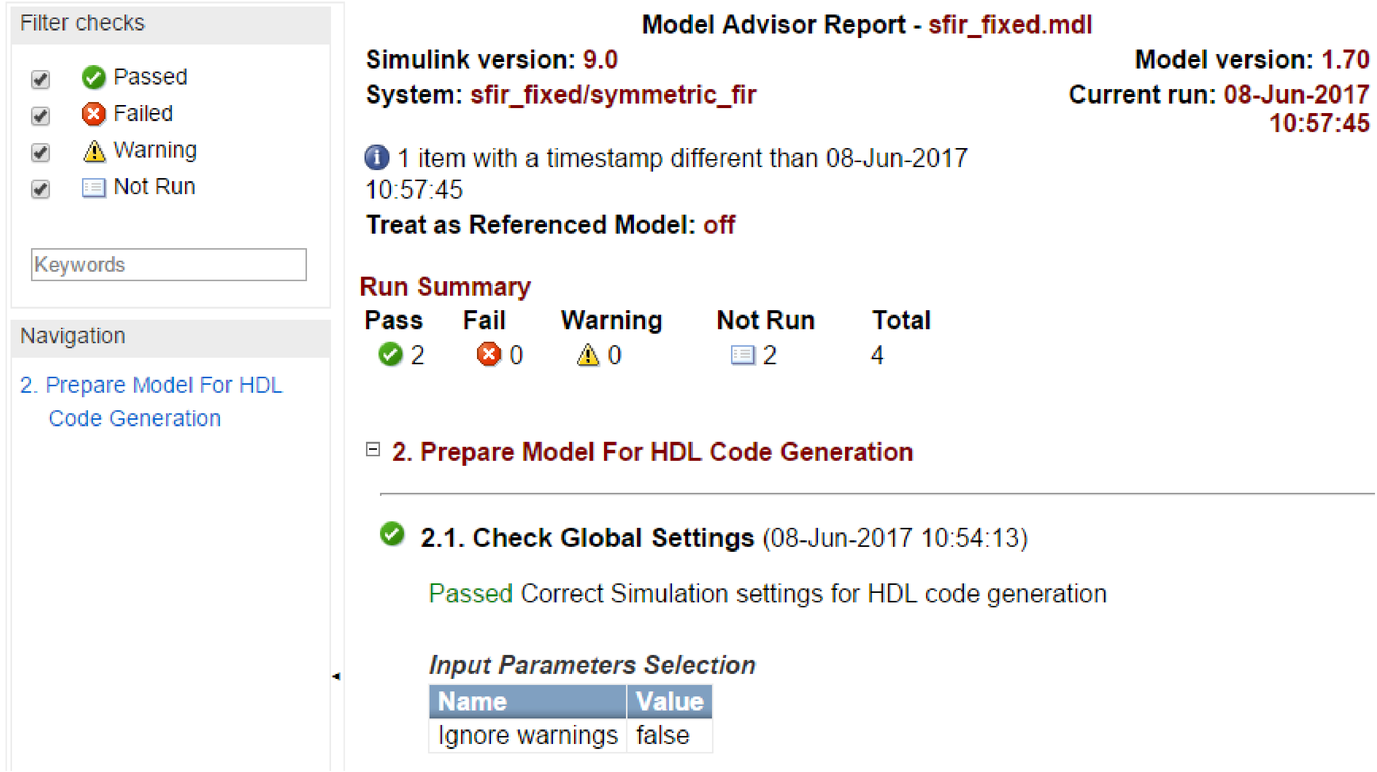


View and Save HDL Workflow Advisor Reports

When you run tasks in the HDL Workflow Advisor, HDL Coder generates an HTML report of the task results. Each folder in the HDL Workflow Advisor contains a report for the checks within that folder and its subfolders. To access reports, select a folder, such as **Prepare Model for HDL Code Generation**, and in the **Report** subpane, click **Save As**. If you rerun HDL Workflow Advisor, the report is updated in the working folder.



This report shows typical results after running the **Prepare Model For HDL Code Generation** tasks.



As you run checks, the HDL Workflow Advisor updates the reports with the latest information for each check in the folder. When you run the checks at different times, timestamps appear at the top right of the report to indicate when checks have been run. Checks that occurred during previous runs have a timestamp following the check name. You can filter checks in the report. For example, you can filter out tasks that are **Not Run** from the report, or you can filter the report to show tasks that **Passed**, and so on. To view the report for a folder each time the tasks in a folder are run, select **Show report after run**.

See Also

Functions
`hdlcoder.runWorkflow` | `setAllTasks` | `clearAllTasks`

Classes
`hdlcoder.WorkflowConfig`

Related Examples

- "HDL Workflow Advisor Tasks" on page 36-2
- "HDL Code Generation and FPGA Synthesis from Simulink Model"

Generate Code and Synthesize on FPGA Using HDL Workflow Advisor

The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

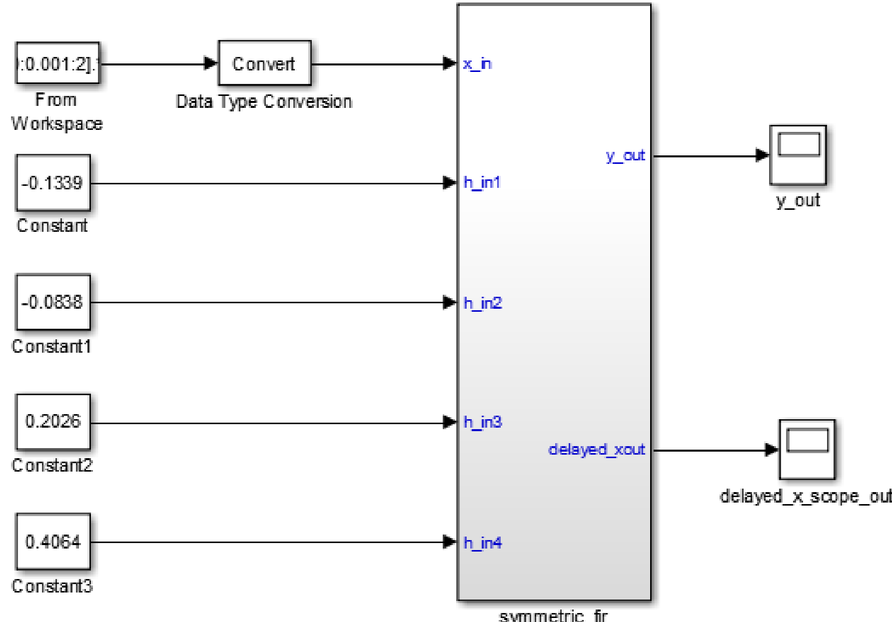
- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

FIR Filter Model

This example illustrates how you can generate HDL code for the FIR filter model and synthesize the design on an FPGA device. Before you generate HDL code, the model must be compatible for HDL code generation. To check and update your model for HDL compatibility, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 38-2. HDL Workflow Advisor is not available in Simulink Online.

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



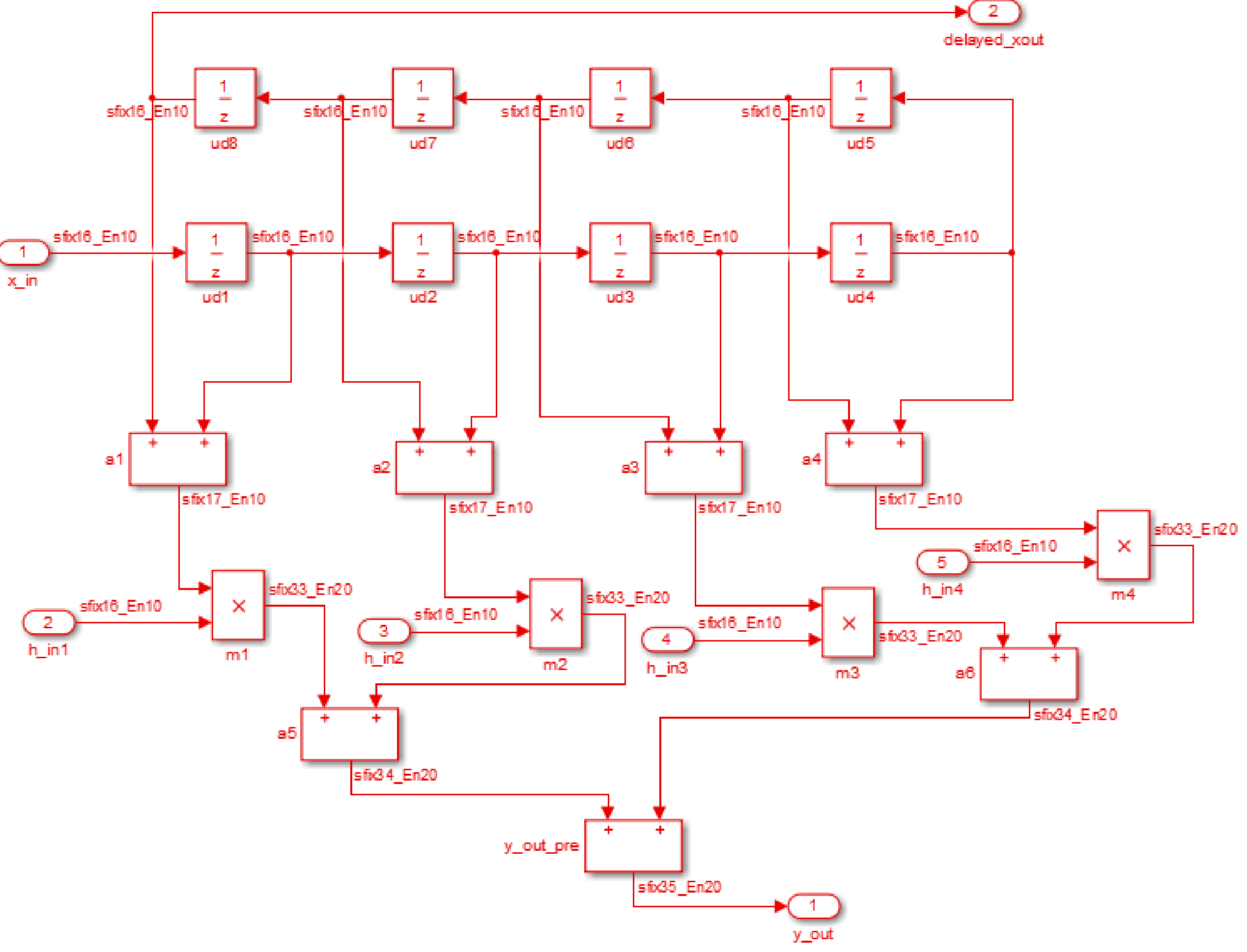
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



Create a Folder and Copy Relevant Files

In MATLAB:

1 Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

`sl_hdlcoder_work` stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

2 Make the `sl_hdlcoder_work` folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

3 Save a local copy of the `sfir_fixed` model to your current working folder. Leave the model open.

Set Up Tool Path

If you do not want to synthesize your design, but want to generate HDL code, you do not have to set the tool path. In the HDL Workflow Advisor, on the **Set Target** > **Set Target Device and Synthesis Tool** step, leave the **Synthesis tool** setting to the default `No Synthesis Tool Specified`, and then run the workflow.

If you want to synthesize your design on a target platform, before you open the HDL Workflow Advisor and run the workflow, set up the path to your synthesis tool. This example uses Xilinx Vivado, so you must have already installed Xilinx Vivado. To set the tool path, use the `hdlsetuptoolpath` function to point to an installed Xilinx Vivado 2019.2 executable. Optionally, you can use a different synthesis tool of your choice and follow this example. To set the path to that synthesis tool, use `hdlsetuptoolpath`. To learn about the latest supported tools, see “HDL Language Support and Supported Third-Party Tools and Hardware”.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
 'C:\Xilinx\Vivado\2019.1\bin\vivado.bat');
```

Open the HDL Workflow Advisor

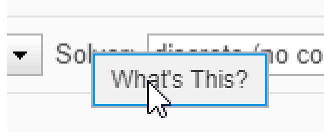
To start the HDL Workflow Advisor from a Simulink model,

1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
2 Select the DUT Subsystem in your model, and make sure that this Subsystem name appears in the **Code for** option. To remember the selection, you can pin this option. Click **Workflow Advisor**.

When you open the HDL Workflow Advisor, the code generator might warn that the project folder is incompatible. To open the Advisor, select **Remove slprj and continue**.

In the HDL Workflow Advisor, the left pane lists the folders in the hierarchy. Each folder represents a group or category of related tasks. From the left pane, you can select a folder or an individual task. The HDL Workflow Advisor displays information about the selected folder or task in the right pane.

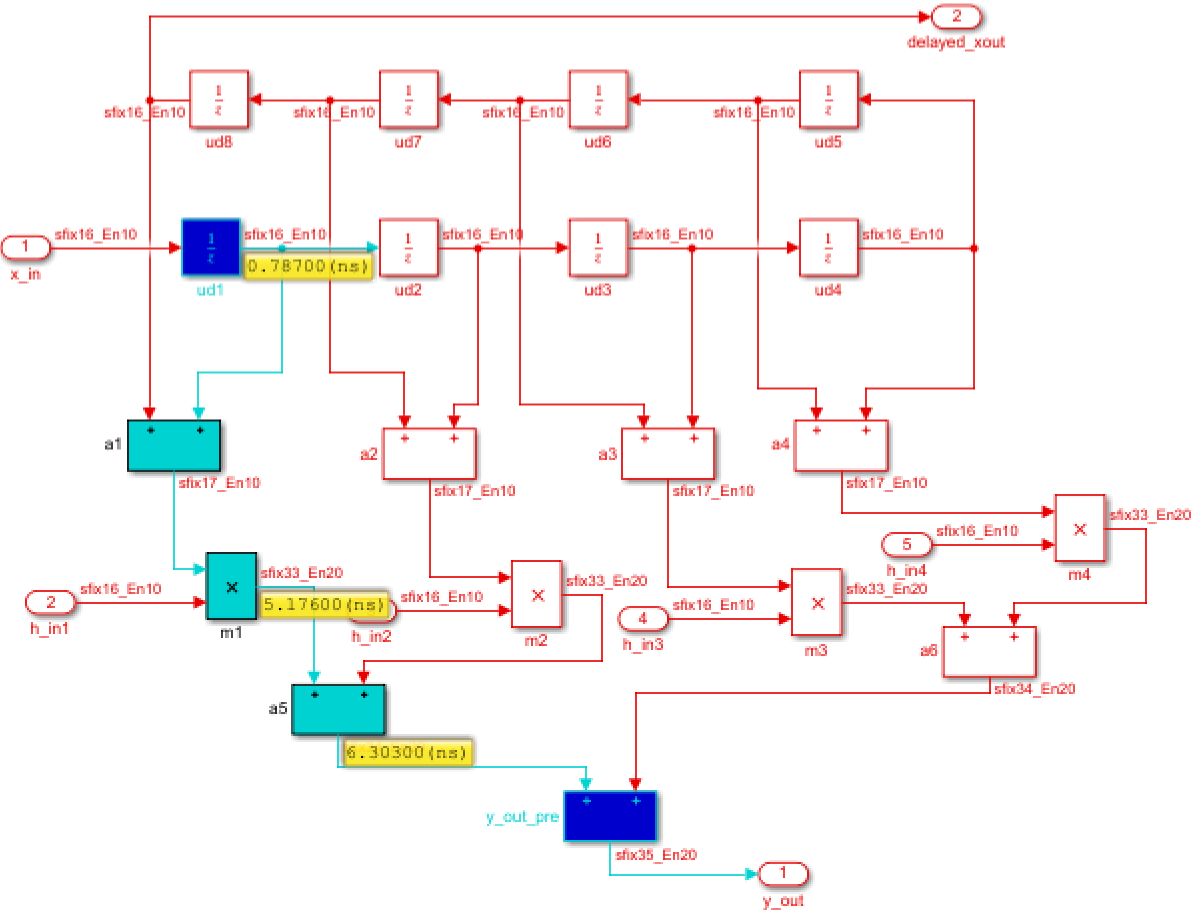
To learn more about each individual task, right-click that task, and select **What's This?**.



To learn more about the HDL Workflow Advisor window, see “Getting Started with the HDL Workflow Advisor” on page 29-5.

Generate HDL Code and Synthesize on FPGA

1. In the **Set Target** > **Set Target Device and Synthesis Tool** step, for **Synthesis tool**, select `Xilinx Vivado` and select **Run This Task**.
2. To generate code, right-click the **Generate RTL Code and Testbench** task, and select **Run to Selected Task**.
3. In the **FPGA Synthesis and Analysis** > **Perform Synthesis and P/R** > **Run Implementation** task, clear **Skip this task** and click **Apply**.
4. Right-click the **Annotate Model with Synthesis Result** and select **Run to Selected Task**.



Run Workflow at Command Line with a Script

To run the HDL workflow at a command line, you can export the Workflow Advisor settings to a script. To export to script, in the HDL Workflow Advisor window, select **File** > **Export to Script**. In the Export Workflow Configuration dialog box, enter a file name and save the script.

The script is a MATLAB file that you can run from the command line. You can modify the script directly or, import the script into the HDL Workflow Advisor, modify the tasks, and export the updated script. To learn more, see “Run HDL Workflow with a Script” on page 29-47.

See Also

`makehdl` | `hdladvisor`

More About

- “Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor” on page 29-16
- “Generate HDL Code from Simulink Model Using Configuration Parameters” on page 16-6
- “Generate HDL Code from Simulink Model from Command Line” on page 16-10

Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor

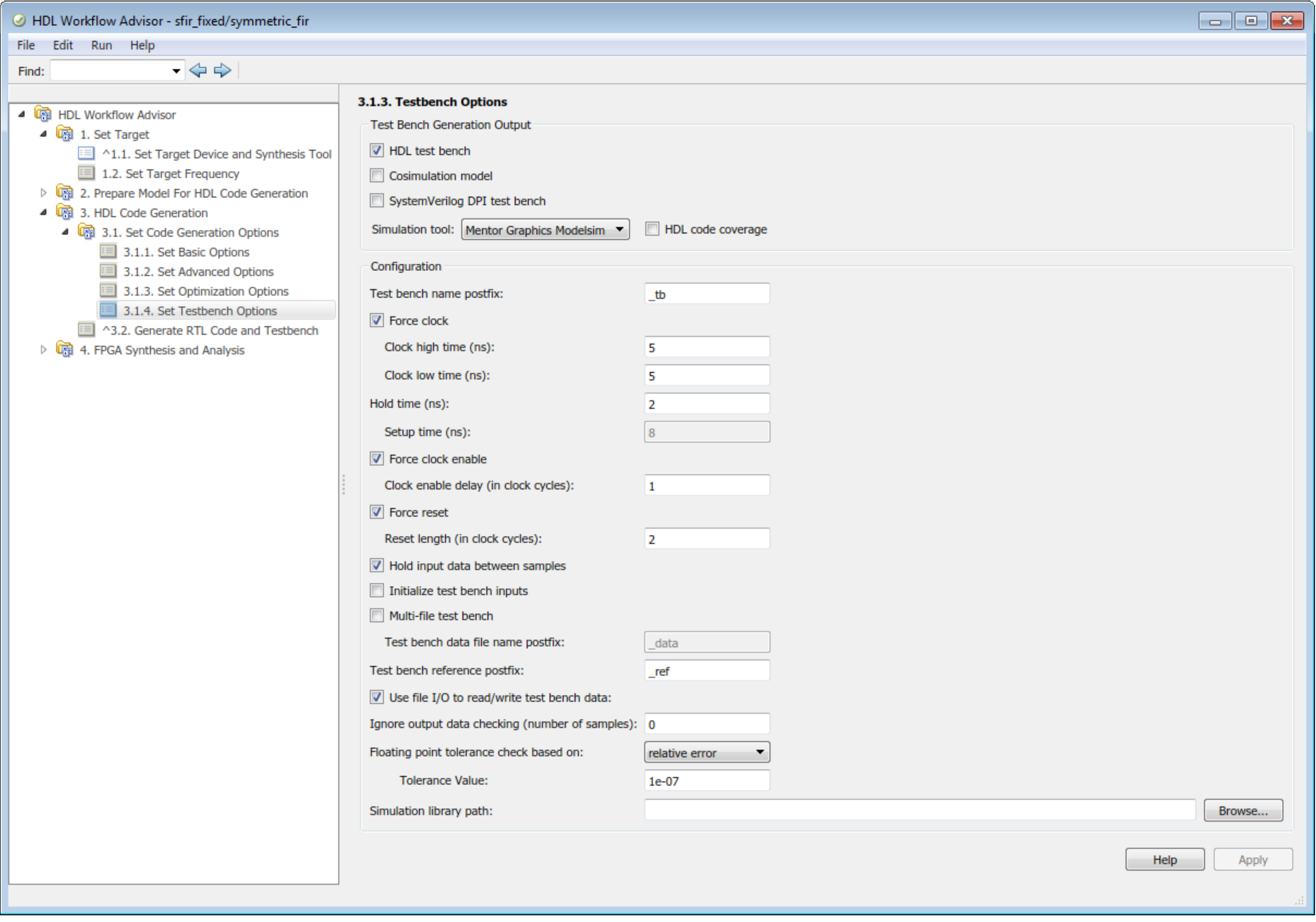
The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

HDL Workflow Advisor is not available in Simulink Online.

To select test bench and code coverage options for generating HDL code from a Simulink model using the HDL Workflow Advisor:

1. Perform the setup steps in “HDL Code Generation and FPGA Synthesis from Simulink Model”.
2. In Step 3.1.4 of the HDL Workflow Advisor, **Set Testbench Options**, select test bench and code coverage options from the **Test Bench Generation Output** section. The coder generates a build-and-run script for your test bench and the **Simulation tool** you specify. If you select multiple test bench options, the coder generates one test bench and script for each type of test bench selected. If you select **HDL code coverage**, the test bench scripts turn on code coverage for your generated HDL code. For more information about the different kinds of test benches, see “Choose a Test Bench for Generated HDL Code” on page 25-41. After you select your test bench options, click **Apply**.



3 In Step 3.2, **Generate RTL Code and Testbench**, select **Generate test bench**. Click **Apply**, and then click **Run This Task**. The coder generates HDL code for your subsystem, and the test benches and scripts you selected in step 3.1.3.

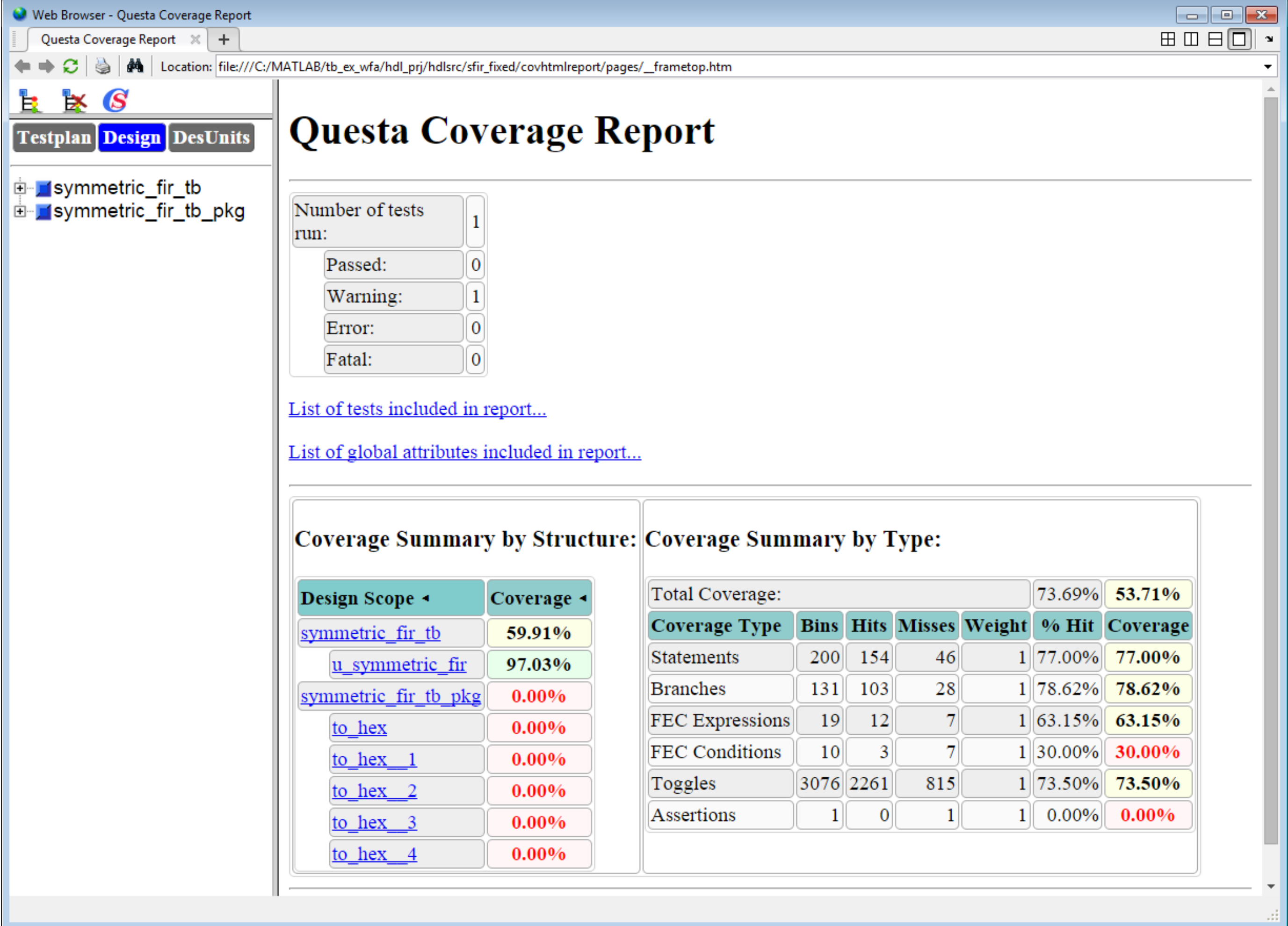
- If you selected **Cosimulation model**, then step 3.3, **Verify with HDL Cosimulation**, appears in the HDL Workflow Advisor. This step automatically runs the generated cosimulation model. The model compares the result of the HDL code running in your HDL simulator with the output of your Simulink subsystem.
- If you selected **HDL test bench**, the coder generates a compile script, `*subsystemname*_tb_compile`, and a run script, `*subsystemname*_tb_sim`. The script file extension depends on your selected simulator. For example, at the command line in the Mentor Graphics ModelSim simulator, change to the `hdl_prj/hdlsrc/*modelname*` folder and run these commands:

```
do symmetric_fir_compile.do
do symmetric_fir_tb_compile.do
do symmetric_fir_tb_sim.do
```

- If you selected **SystemVerilog DPI test bench**, the coder generates a script file, `*subsystemname*_dpi_tb`, that compiles the HDL code and runs the test bench simulation. The script file extension depends on your selected simulator. For example, at the command line in the Mentor Graphics ModelSim simulator, change to the `hdl_prj/hdlsrc/*modelname*` folder and run this command:

```
do symmetric_fir_dpi_tb.do
```

- If you selected **HDL code coverage**, the code coverage report from running any test bench, including the cosimulation model, is saved in `hdl_prj\hdlsrc\`*`modelname`*`\covhtmlreport`.



See Also

More About

- "Choose a Test Bench for Generated HDL Code" on page 25-41

Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries

In this section...

"Set Up Design for Mixed-Mode Mapping" on page 29-19
"Map to Native Floating-Point and FPGA Floating-Point Libraries" on page 29-20
"View Code Generation Reports of Floating-Point Library Mapping" on page 29-21
"Analyze Results of Floating-Point Library Mapping" on page 29-22

Mapping to a floating-point library enables you to synthesize your floating-point design without doing floating-point to fixed-point conversion. Eliminating the conversion step reduces the loss of data precision and enables you to model a wider dynamic range.

An FPGA floating-point library is a set of floating-point IP blocks that are optimized for synthesis on specific target hardware. These libraries are also referred to as vendor-specific floating point libraries because they target vendor-specific hardware. Altera Megafunctions and Xilinx LogiCORE IP are examples of these libraries. You can create a design and generate code that consists of HDL Coder native floating point (NFP) and vendor-specific FPGA point IP to more efficiently use resources on the FPGA, such as hardened DSP floating point adder or multiplier primitives, which allows you to fit a bigger design into the FPGA fabric. This mixed design is advantageous for large and complex models. Additionally, you can map blocks that are unsupported by the vendor-specific library to NFP and use the vendor library to map other blocks to vendor-specific floating point resources.

You can map your model to either the:

- Native floating-point library. This option is the default mapping.
- Native floating-point library and a vendor-specific floating point library. This is also called mixed-mode mapping.

When you map to both a native floating-point library and a vendor-specific floating-point library, HDL Coder maps to the vendor library IP wherever possible. The blocks that cannot be mapped to the vendor library then map to native floating-point library IP instead.

To see a list of HDL Coder blocks that support FPGA floating-point library mapping, see "HDL Coder Support for FPGA Floating-Point Library Mapping" on page 29-41.

Set Up Design for Mixed-Mode Mapping

To map your floating-point design to the native floating-point library and an Altera or Xilinx FPGA floating-point library:

1. Set up the path to your synthesis tool executable file by using `hdlsetuptoolpath`. For example, to set the path to the `Altera Quartus II` synthesis tool, enter:

   ```
   hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath',...
    'C:\altera\14.0\quartus\bin\quartus.exe');
   ```

   See "Synthesis Tool Path Setup".
2. Set the target device options for your Altera or Xilinx FPGA synthesis tool using `hdlset_param`. For example, to set the synthesis tool as `Altera Quartus II` and chip family as `Arria10`, enter:

```
hdlset_param(model,'SynthesisToolChipFamily','Arria10', ...
                   'SynthesisToolDeviceName','10AS066H2F34E1SG', ...
                   'SynthesisToolPackageName','', ...
                   'SynthesisToolSpeedValue','')
```

3 Set up your Altera or Xilinx FPGA floating-point simulation libraries. See "FPGA Simulation Library Setup".

Map to Native Floating-Point and FPGA Floating-Point Libraries

You can map your Simulink model to floating-point target libraries from the Configuration Parameters dialog box or from the command line.

Map to Floating-Point Target Libraries from the Configuration Parameters Dialog Box

To map to the native floating point library and FPGA floating-point library:

1 In the **Apps** tab, select **HDL Coder**. In the **HDL Code** tab, click **Settings**.
2 In the **HDL Code Generation** > **Floating Point** pane, select **Use Floating Point**.
3 The **Vendor Specific Floating Point Library** parameter displays options based the synthesis tool you chose for your design. To use the Xilinx LogiCORE® IP, set the **Vendor Specific Floating Point Library** to `XILINXLOGICORE`. For Altera megafunction IP, you can set **Vendor Specific Floating Point Library** to `ALTFP` or `ALTERAFPFUNCTIONS`.

Note When mapping to ALTERA FP FUNCTIONS, the target language must be VHDL.

4 You can customize the IP settings of the vendor-specific floating-point target library by using the `hdlcoder.FloatingPointTargetConfig` and the `hdlcoder.FloatingPointTargetConfig.IPConfig` objects. For more information, see "Customize Floating-Point IP Configuration" on page 29-36.
5 To share floating-point IP resources, in the **HDL Code Generation** > **Optimizations**pane, in the **Resource Sharing** tab, select **Floating-point IPs**. The number of floating-point IP blocks that get shared depends on the **SharingFactor** that you specify on the subsystem.
6 Click **OK**. In the **HDL Code** tab, click **Generate HDL Code**.

Map to Floating-Point Target Libraries from the Command Line

To generate HDL code from the command line, you can use the `hdlcoder.createFloatingPointTargetConfig` function to create a floating-point IP configuration.

1 Use the `hdlcoder.createFloatingPointTargetConfig` function to create a `hdlcoder.FloatingPointTargetConfig` object for the native floating-point library and a vendor-specific floating-point library. Then, use `hdlset_param` to save the configuration on the model.

   For example, to create a floating-point target configuration for the native floating point and `ALTERA FP FUNCTIONS` libraries with the default settings and set the configuration to the `sfir_single` model, use these commands:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",...
   VendorFloatingPointLibrary="ALTERAFPFUNCTIONS");
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
```

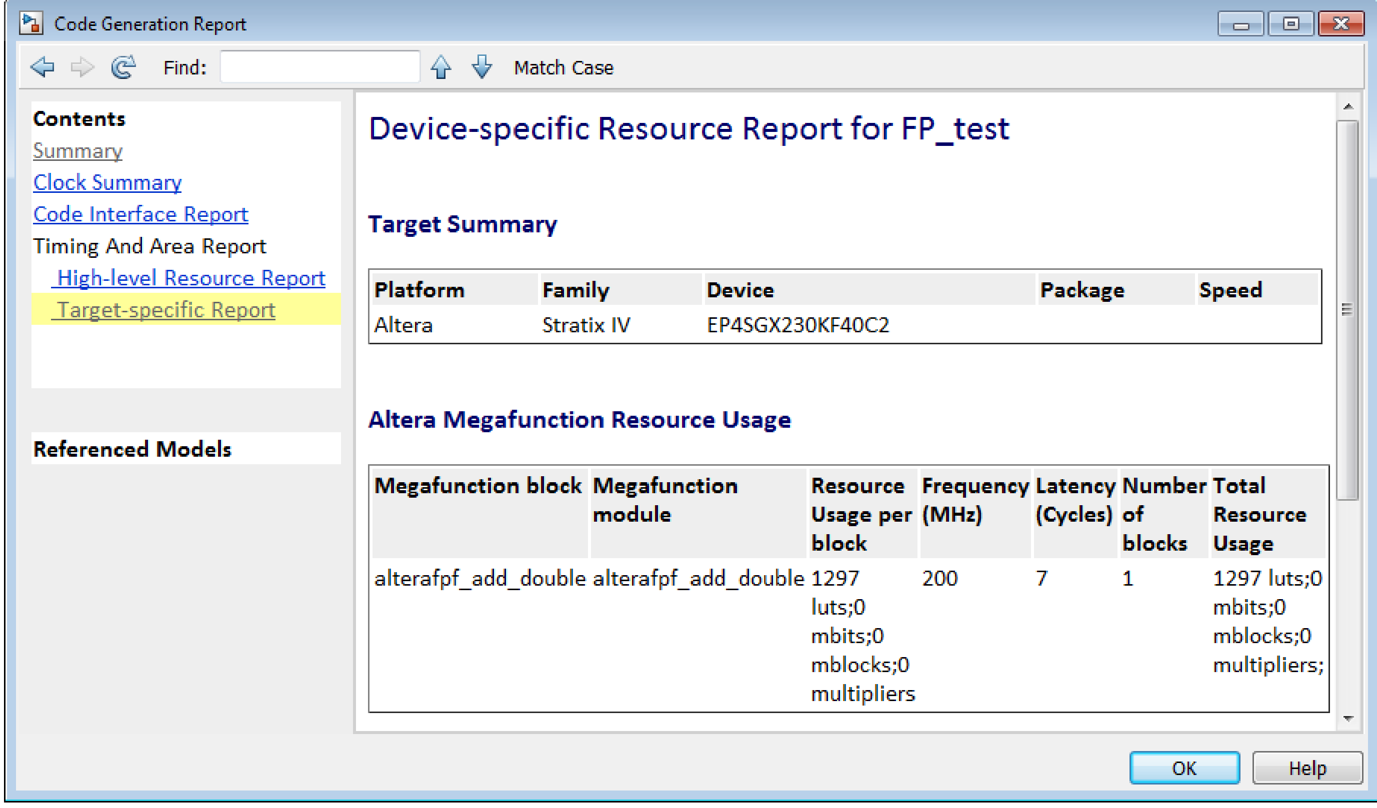
2. You can customize the IP settings based on the floating-point library that you specify. For more information, see “Customize Floating-Point IP Configuration” on page 29-36.
3. Use the `makehdl` function to generate HDL code from the DUT subsystem.

View Code Generation Reports of Floating-Point Library Mapping

To view information about the floating-point library mapping in the code generation report, before you begin code generation, enable the resource utilization report and optimization report. To learn how to generate these reports, see “Create and Use Code Generation Reports” on page 23-2.

Target-Specific Report

To see the target floating-point block your design mapped to, the latency, and number of target-specific hardware resources, in the code generation report, select **Target-specific Report**.



Target Code Generation Report

In the code generation report, the **Optimization Report** > **Target Code Generation** section shows the optimization settings applied to the model. This section shows whether HDL Coder successfully generated floating-point target code.

Code Generation Report

Find: Match Case

Contents

Summary
Clock Summary
Code Interface Report
Timing And Area Report
High-level Resource Report
Optimization Report
Distributed Pipelining
Streaming and Sharing
Delay Balancing
Adaptive Pipelining
Target Code Generation

Referenced Models

Target Code Generation Report for FP_Test

Target Summary

Platform	Family	Device	Package	Speed
Altera	Arria 10	10AS016C3U19E2LG		

Target Mapping Status

Successful.

Path Delay Summary

Port	Path Delay
Subsystem/ce_out	6
Subsystem/Out1	6

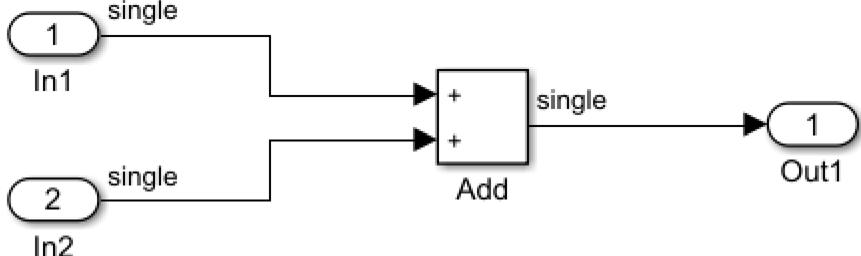
Generated Model

Generated model after the transformation:	gm_FP_Test

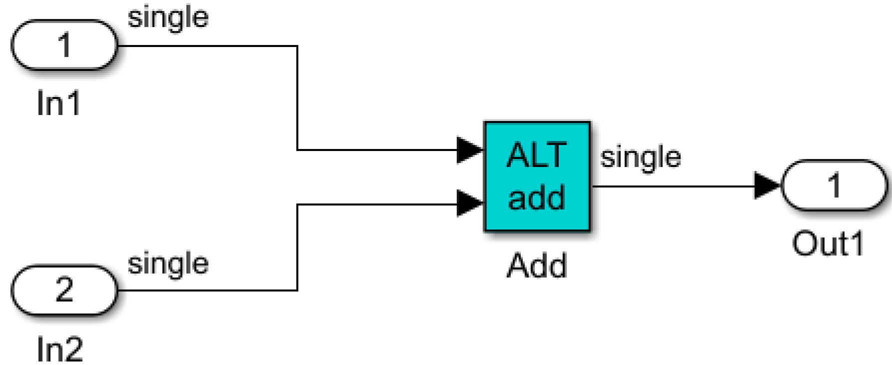
OK Help

Analyze Results of Floating-Point Library Mapping

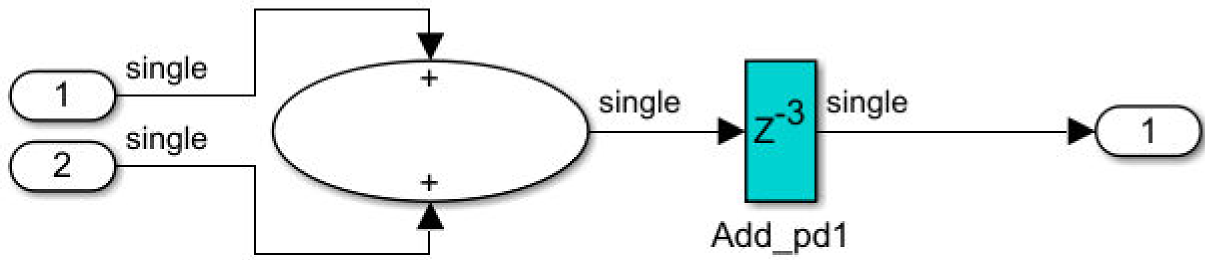
You can get the latency information of the floating-point target IP from the generated model after HDL code generation. For example, consider this Add block in Simulink that has inputs of single data type.



1. After HDL code generation, the optimization report displays a link to the generated model. To see the floating-point target library that your Simulink block mapped to, open the generated model and double-click the DUT subsystem. The blocks that map to native floating point IP are light blue and include `NFP` on the block mask. The blocks that map to the vendor-specific floating point IP are cyan and have initials associated with the vendor library on the block mask. In this example, the initials on the blocks that map to the `ALTERA FP FUNCTIONS` library display the initials `ALT`.

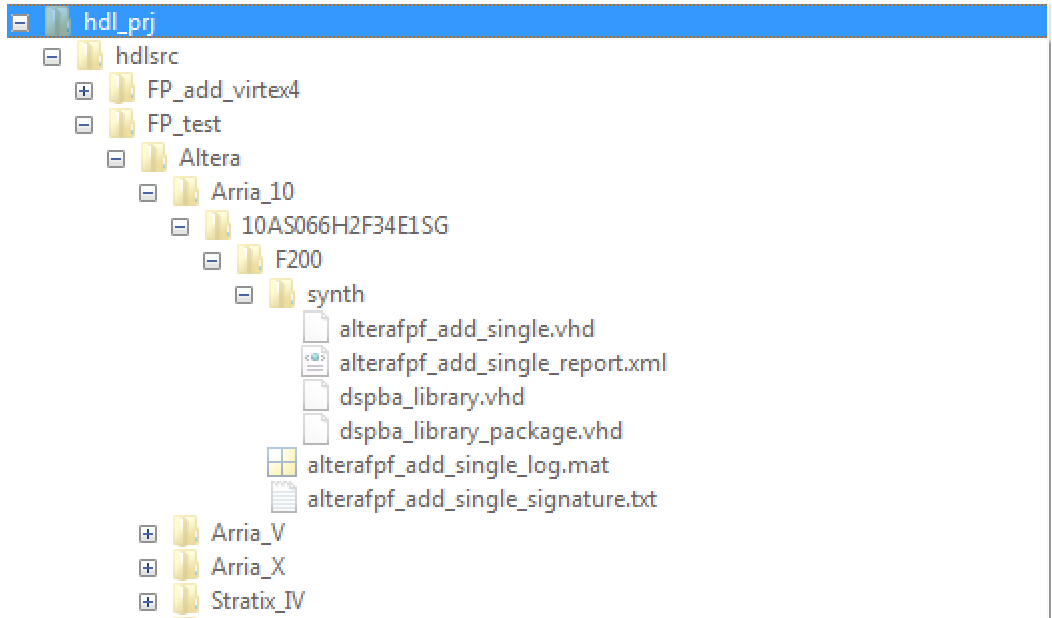


2. Double-click the Add block. The length of the delay block is the latency of the floating-point target IP.



To learn more about the generated model, see “Generated Model and Validation Model” on page 21-10.

To see your FPGA floating-point library mapping results, you can view the IP core files generated after HDL code generation.



HDL Coder checks and reuses existing generated IP core files, which takes less time when successively generating code for the same floating-point target IP.

See Also

`hdlcoder.FloatingPointTargetConfig` | `createFloatingPointTargetConfig` | `hdlcoder.FloatingPointTargetConfig.IPConfig` | `customize`

Related Examples

- "FPGA Floating-Point Library IP Mapping" on page 29-25
- "Design Model by Using HDL Coder Native Floating Point and Intel Hard Floating Point" on page 14-125

More About

- "Customize Floating-Point IP Configuration" on page 29-36
- "HDL Coder Support for FPGA Floating-Point Library Mapping" on page 29-41

FPGA Floating-Point Library IP Mapping

This example shows how to use vendor-specific floating-point IP libraries, such as Altera® and Xilinx®, to generate target specific HDL code. For more information on how to map designs to floating-point libraries, see “Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries” on page 29-19.

When you implement designs with floating-point arithmetic, you can model with higher precision and a wider dynamic range and you can avoid converting floating-point data to fixed-point data. Floating-point designs particularly beneficial for model-based design, where high-level algorithms are modeled with floating-point data and do not have implementation timing details, such as pipelining and timing constraints. However, timing details are necessary to map operations to floating-point IP modules. HDL Coder™ automatically optimizes and implements your designs with these timing details and provides interfaces for you to adjust them. HDL Coder then implements the floating-point math by integrating it with the floating-point IP modules from vendor libraries.

Open Model

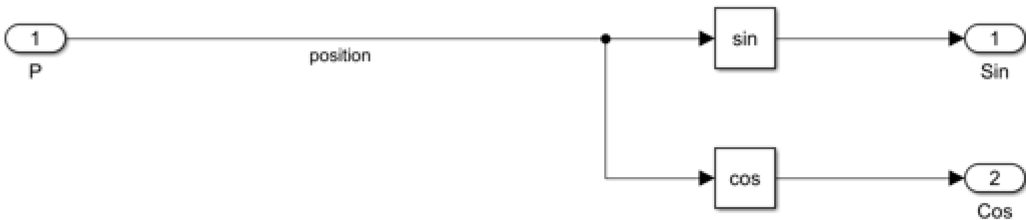
This field-oriented control (FOC) algorithm example demonstrates how to map designs to floating-point libraries. For more information about this algorithm, see “Field-Oriented Control of a Permanent Magnet Synchronous Machine” on page 14-66.

This model uses single-precision data types and contains blocks that perform basic math operators, such as adders, multipliers, comparators, and complex sine and cosine functions.

This model models the signal rates at 20 μs or 50 KHz. This model only contains the numerical implementation, and does not have FPGA implementation timing details, such as operation latencies. All numerical operations, including the sine and cosine functions, compute in a single sample time-step.

```
load_system('hdlcoderFocCurrentSingleTargetHdl');
open_system('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Sine_Cosine');
```

Sine Cosine Coefficients



Set Up FPGA for Floating-Point Mapping

To map to a vendor-specific floating-point library, set the FPGA device by choosing a synthesis tool and chip to target. Before setting the synthesis tool, use the `hdlsetuptoolpath` function to set your

synthesis tool path. This example uses both `Altera Quartus` and `XILINX ISE` as synthesis tools. `quartuspath` and `isepath` return synthesis tool path to the `Altera Quartus II` synthesis tool and the `Xilinx ISE` synthesis tool, respectively. For information on how to set up tools in your environment, see `hdlsetuptoolpath`.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II','ToolPath', quartuspath);
hdlsetuptoolpath('ToolName', 'XILINX ISE','ToolPath', isepath);
```

```
Prepending following Altera Quartus II path(s) to the system path:
D:\share\apps\HDLTools\Altera\22.1.1-mw-0\Windows\quartus\bin64
Setting XILINX environment variable to:
D:\share\apps\HDLTools\Xilinx_ISE\14.7-mw-0\Win\ISE_DS\ISE
Setting XILINX_EDK environment variable to:
D:\share\apps\HDLTools\Xilinx_ISE\14.7-mw-0\Win\ISE_DS\EDK
Setting XILINX_PLANAHEAD environment variable to:
D:\share\apps\HDLTools\Xilinx_ISE\14.7-mw-0\Win\ISE_DS\PlanAhead
Prepending following XILINX ISE path(s) to the system path:
D:\share\apps\HDLTools\Xilinx_ISE\14.7-mw-0\Win\ISE_DS\ISE\bin\nt64;D:\share\apps\HDLTools\Xilinx
```

Use the `hdlset_param` function to set the `SynthesisTool` and `SynthesisToolChipFamily` HDL model properties to `Altera Quartus II` and `Arria 10`, respectively.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'SynthesisTool', 'Altera Quartus II');
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'SynthesisToolChipFamily', 'Arria 10');
```

Choose FPGA IP Library and Create Floating Point Configuration Object

To map to the native floating point and a vendor-specific FPGA floating point library, choose a vendor library to target. For Xilinx devices, use the `XILINXLOGICORE` library. For Altera devices, use the `ALTERAFPFUNCTIONS` or `ALTFP` libraries. Check the library documentation for their supported devices.

Create a floating-point target configuration object for mixed native floating-point and ALTERAFPFUNCTIONS libraries by using the `createFloatingPointTargetConfig` function with the `VendorFloatingPointLibrary` name-value argument set to `ALTERAFPFUNCTIONS`. For more information on creating a floating-point target configuration object, see `createFloatingPointTargetConfig`.

```
fc = hdlcoder.createFloatingPointTargetConfig(VendorFloatingPointLibrary = 'ALTERAFPFUNCTIONS')
```

```
fc = 

  FloatingPointTargetConfig with properties:

                  Library: 'NATIVEFLOATINGPOINT'
          LibrarySettings: [1×1 fpconfig.NFPLatencyDrivenMode]
                 IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
            VendorLibrary: 'ALTERAFPFUNCTIONS'
    VendorLibrarySettings: [1×1 fpconfig.FrequencyDrivenMode]
           VendorIPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

Set the floating-point configuration object on the model to the configuration object `fc`.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'FloatingPointTargetConfiguration', fc);
```

To compile and simulate the generated code with QuestaSim, compile the Altera simulation library and set its path by using the `SimulationLibPath` parameter. See "Tool Setup" for more information. `alterasimulationlibpath` returns the path to the compiled Altera simulation library.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'SimulationLibPath', alterasimulationlibpath);
```

The Altera Megafunction (ALTERAFPFUNCTIONS) library allows you to generate IP modules for a given target frequency. In this example, set the target frequency to 250MHz.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'TargetFrequency', 250);
```

Generate Code and Adjust Model for IP Mapping

Generate HDL code for the DUT subsystem, `FOC_Current_Control`, by using the `makehdl` command.

```
try
    makehdl('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control');
catch me
    disp(me.message);
end
```

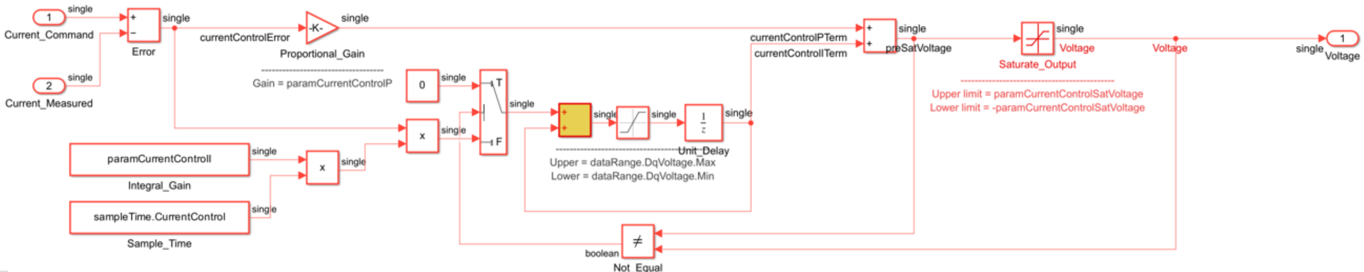
```
### Working on the model <a href="matlab:open_system('hdlcoderFocCurrentSingleTargetHdl')">hdlcod
### Generating HDL for <a href="matlab:open_system('hdlcoderFocCurrentSingleTargetHdl/FOC_Curren
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurre
### Running HDL checks on the model 'hdlcoderFocCurrentSingleTargetHdl'.
### Begin compilation of the model 'hdlcoderFocCurrentSingleTargetHdl'...
### Begin compilation of the model 'hdlcoderFocCurrentSingleTargetHdl'...
### Working on the model 'hdlcoderFocCurrentSingleTargetHdl'...
### Using D:\share\apps\HDLTools\Altera\22.1.1-mw-0\Windows\quartus\bin64\..\sopc_builder\bin\ip-
### Generating Altera(R) megafunction: alterafpf_add_single for target frequency of 250 MHz.
### alterafpf_add_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_mul_single for target frequency of 250 MHz.
### alterafpf_mul_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_lt_single_LT for target frequency of 250 MHz.
### alterafpf_lt_single_LT takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_gt_single_GT for target frequency of 250 MHz.
### alterafpf_gt_single_GT takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_sub_single for target frequency of 250 MHz.
### alterafpf_sub_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_neq_single_NEQ for target frequency of 250 MHz.
### alterafpf_neq_single_NEQ takes 0 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_le_single_LE for target frequency of 250 MHz.
### alterafpf_le_single_LE takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_ge_single_GE for target frequency of 250 MHz.
### alterafpf_ge_single_GE takes 1 cycles.
### Done.
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdls
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Creating HDL Code Generation Check Report file:///C:/Users/user/OneDrive%20-%20MathWorks/Docu
```

```
### HDL check for 'hdlcoderFocCurrentSingleTargetHdl' complete with 1 errors, 0 warnings, and 1 m
### HDL check for 'hdlcoderFocCurrentSingleTargetHdl' complete with 1 errors, 0 warnings, and 1 m

For the block 'hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/Q_Current
   Delay balancing unsuccessful. Cause:
   The input and output configurations of this atomic subsystem cause delay balancing to fail. Co
```

The error messages indicate that HDL Coder cannot replace operations in feedback loops with floating-point IP modules because these loops have fewer delays than the latency of the floating-point IP modules that HDL Coder tries to replace them with. HDL Coder implements floating-point IP modules as pipelined blocks. For some modules, there are minimum latency requirements. Because changing the latency of a feedback loop generates an incorrect implementation, HDL Coder prevents the addition of latency inside these feedback loops.

The error indicates that the adder inside the feedback loop requires multiple cycles of latency budget, but the loop has only one delay.

```
hilite_system('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Curren
```



To address these errors, you can use one of these approaches:

- Reducing the target frequency may lower the pipelining depth requirement. This change may also slow down all other IP modules in the design.
- Configuring the IP modules used in the loop with a smaller latency also slows down the operating frequency of the IP modules, but only for specified IP modules.
- Applying clock rate pipelining. When the data rate is slower than the FPGA clock rate, the FPGA has multiple cycles at clock rate to finish operations and still retain numerical consistency. For more information about clock rate pipelining, see “Clock-Rate Pipelining” on page 21-148.

Because the sample time is 20 μs and the FPGA target frequency is 250 MHz, or 4 ns, you can apply clock-rate pipelining to solve the feedback loop problem by setting `TreatRatesAsHardwareRates` to `on`. `TreatRatesAsHardwarRates` allows HDL Coder to determine an oversampling value automically for your design by treating your Simulink base rate as an actual hardware rate and calculating the oversampling value as the ratio between your target frequency and Simulink base rate. HDL Coder then uses the oversampling value for clock-rate pipelining. For more information, see Treat Simulink rates as actual hardware rates. For this example, the calculated oversampling value is 5000, which means that one unit delay with a sample time of 20 μs in the model is equivalent to 5000 clock-rate cycles at a sample time of 4 ns on the FPGA. This value is sufficient for floating-point IP modules in the loops.

Enable `TreatRatesAsHardwareRates` and enable the generation of the generated and validation model.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl','TreatRatesAsHardwareRates','on');
hdlset_param('hdlcoderFocCurrentSingleTargetHdl','GenerateModel','on');
hdlset_param('hdlcoderFocCurrentSingleTargetHdl','GenerateValidationModel','on');
```

Generate HDL code for the DUT subsystem, `FOC_Current_Control`, by using the `makehdl` command.

```
makehdl('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control');
```

```
### Working on the model <a href="matlab:open_system('hdlcoderFocCurrentSingleTargetHdl')">hdlcod
### Generating HDL for <a href="matlab:open_system('hdlcoderFocCurrentSingleTargetHdl/FOC_Curren
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurre
### Running HDL checks on the model 'hdlcoderFocCurrentSingleTargetHdl'.
### Begin compilation of the model 'hdlcoderFocCurrentSingleTargetHdl'...
### Begin compilation of the model 'hdlcoderFocCurrentSingleTargetHdl'...
### Working on the model 'hdlcoderFocCurrentSingleTargetHdl'...
### Using D:\share\apps\HDLTools\Altera\22.1.1-mw-0\Windows\quartus\bin64\..\sopc_builder\bin\ip-
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_lt_single_LT for target frequency of 250 MHz.
### alterafpf_lt_single_LT takes 1 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_gt_single_GT for target frequency of 250 MHz.
### alterafpf_gt_single_GT takes 1 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_mul_single for target frequency of 250 MHz.
### alterafpf_mul_single takes 3 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_le_single_LE for target frequency of 250 MHz.
### alterafpf_le_single_LE takes 1 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_ge_single_GE for target frequency of 250 MHz.
### alterafpf_ge_single_GE takes 1 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_sub_single for target frequency of 250 MHz.
### alterafpf_sub_single takes 3 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_add_single for target frequency of 250 MHz.
### alterafpf_add_single takes 3 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_neq_single_NEQ for target frequency of 250 MHz.
### alterafpf_neq_single_NEQ takes 0 cycles.
### Done.
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 2 cycles.
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocCurrentSingleTargetHdl', 'G
### Begin model generation 'gm_hdlcoderFocCurrentSingleTargetHdl'...
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoderFocCurrentSingleTargetHd
```

```
### Clock-rate pipelining obstacles can be diagnosed by running this script: <a href="matlab:run
### Delay absorption obstacles can be diagnosed by running this script: <a href="matlab:run('hdls
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: <a href="matlab:open_system('hdlsrc\hdlcoderFocCurrentSingle
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentSingleTargetHdl'.
### MESSAGE: The design requires 5000 times faster clock with respect to the base rate = 2e-05.
### Begin VHDL Code Generation for 'FOC_Current_Control_tc'.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Co
### Code Generation for 'FOC_Current_Control_tc' completed.
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocCurrentSingleTargetHdl', '
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Curren
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Curren
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Curren
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control as hdlsr
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Clarke_Transform as hdlsrc\h
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Sine_Cosine/nfp_sincos_sing
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Sine_Cosine as hdlsrc\hdlco
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Park_Transform as hdlsrc\hd
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Inverse_Park_Transform as h
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Inverse_Clarke_Transform/nf
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Inverse_Clarke_Transform as
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Space_Vector_Modulation as h
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control as hdlsrc\hdlcoderFocCurren
### Generating package file hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control_pkg.vhd
### Code Generation for 'hdlcoderFocCurrentSingleTargetHdl' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/Users/user/OneDrive%20-%20MathWorks/Doc
### HDL check for 'hdlcoderFocCurrentSingleTargetHdl' complete with 0 errors, 0 warnings, and 5 
### HDL code generation complete.
```

The design now maps to floating-point IP modules. The target code generation report summarizes the floating IP module usage. Open and view the generated model. For example, the subsystem `gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Add` that corresponds to the subsystem `hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Add` shows that this operation takes 3 cycles.

```
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Cur
get_param('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Current_
    'DelayLength')
```

```
ans =

    '3'
```

Because floating-point IP modules introduce latencies across the design, HDL Coder adds matching delays to maintain data synchronization. See “Delay Balancing” on page 21-81 for more details.

Share Floating-Point IPs

Floating-point IP modules are suitable to share, because they are usually identical for the same kind. Floating-point IPs are typically expensive operations and it is desirable to share these resources, if possible, to reduce the area footprint. HDL Coder shares resources in the same subsystem. In order to allow more resources to share, flatten the subsystem hierarchy and set resource sharing factor on the top network to 4.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control', 'FlattenHierarchy', 'on');
hdlset_param('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control', 'SharingFactor', 4);
```

Generate HDL code for the DUT subsystem, `FOC_Current_Control`, by using the `makehdl` command.

```
makehdl('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control');
```

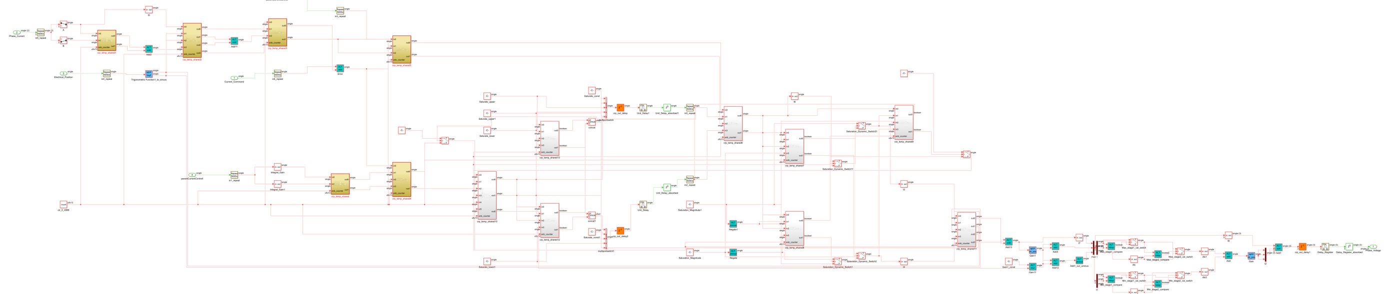
```
### Working on the model <a href="matlab:open_system('hdlcoderFocCurrentSingleTargetHdl')">hdlcod
### Generating HDL for <a href="matlab:open_system('hdlcoderFocCurrentSingleTargetHdl/FOC_Curren
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurre
### Running HDL checks on the model 'hdlcoderFocCurrentSingleTargetHdl'.
### Begin compilation of the model 'hdlcoderFocCurrentSingleTargetHdl'...
### Begin compilation of the model 'hdlcoderFocCurrentSingleTargetHdl'...
### Working on the model 'hdlcoderFocCurrentSingleTargetHdl'...
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_add_single for target frequency of 250 MHz.
### alterafpf_add_single takes 3 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_sub_single for target frequency of 250 MHz.
### alterafpf_sub_single takes 3 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_mul_single for target frequency of 250 MHz.
### alterafpf_mul_single takes 3 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_lt_single_LT for target frequency of 250 MHz.
### alterafpf_lt_single_LT takes 1 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_gt_single_GT for target frequency of 250 MHz.
### alterafpf_gt_single_GT takes 1 cycles.
### Done.
### Using D:\share\apps\HDLTools\Altera\22.1.1-mw-0\Windows\quartus\bin64\..\sopc_builder\bin\ip-
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_neq_single_NEQ for target frequency of 250 MHz.
### alterafpf_neq_single_NEQ takes 0 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_ge_single_GE for target frequency of 250 MHz.
### alterafpf_ge_single_GE takes 1 cycles.
### Done.
### Found an existing generated file in a previous session: (C:\Users\user\OneDrive - MathWorks\D
### Generating Altera(R) megafunction: alterafpf_le_single_LE for target frequency of 250 MHz.
### alterafpf_le_single_LE takes 1 cycles.
### Done.
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocCurrentSingleTargetHdl', 'G
### Begin model generation 'gm_hdlcoderFocCurrentSingleTargetHdl'...
```

```
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generated model saved at <a href="matlab:open_system('hdlsrc\hdlcoderFocCurrentSingleTargetH
### Generating new validation model: <a href="matlab:open_system('hdlsrc\hdlcoderFocCurrentSingle
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentSingleTargetHdl'.
### MESSAGE: The design requires 5000 times faster clock with respect to the base rate = 2e-05.
### Begin VHDL Code Generation for 'FOC_Current_Control_tc'.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Co
### Code Generation for 'FOC_Current_Control_tc' completed.
### Working on... <a href="matlab:configset.internal.open('hdlcoderFocCurrentSingleTargetHdl', '
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/nfp_gain_pow2_single as hdl
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/nfp_uminus_single as hdlsrc
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/nfp_sincos_single as hdlsrc
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control as hdlsrc\hdlcoderFocCurren
### Generating package file hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control_pkg.vhd
### Code Generation for 'hdlcoderFocCurrentSingleTargetHdl' completed.
### Generating HTML files for code generation report at <a href="matlab:hdlcoder.report.openDdg(
### Creating HDL Code Generation Check Report file:///C:/Users/user/OneDrive%20-%20MathWorks/Doc
### HDL check for 'hdlcoderFocCurrentSingleTargetHdl' complete with 0 errors, 0 warnings, and 3 
### HDL code generation complete.
```

The generated model infers fewer IP modules. You can see this change in the floating-point resource report.

Open the generated model to see the resource sharing results.

```
open_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control');
set_param('gm_hdlcoderFocCurrentSingleTargetHdl', 'SimulationCommand', 'update');
set_param('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control', 'ZoomFactor', 'FitSystem')
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared1');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared2');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared3');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared4');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared5');
```



Configure the IP Library

You can edit and control the customization configuration of the floating point IP libraries to meet your design requirements.

Create a mixed-mode floating-point target configuration object for the native floating-point and XILINX LOGICORE libraries.

```
fc = hdlcoder.createFloatingPointTargetConfig(VendorFloatingPointLibrary = 'XILINXLOGICORE');
```

In addition to the library name, library settings, and individual IP configuration settings for the native floating point library stored in the configuration object as `Library`, `LibrarySettings`, and `IPConfig`, respectively, the configuration object also has those properties for the XILINX LOGICORE library, specified as `VendorLibrary`, `VendorLibrarySettings`, and `VendorIPConfig`. View the configuration object, `fc`.

```
fc
```

```
fc =

  FloatingPointTargetConfig with properties:

                  Library: 'NATIVEFLOATINGPOINT'
          LibrarySettings: [1×1 fpconfig.NFPLatencyDrivenMode]
                 IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
            VendorLibrary: 'XILINXLOGICORE'
    VendorLibrarySettings: [1×1 fpconfig.LatencyDrivenMode]
           VendorIPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

The `VendorLibrarySettings` property contains library-wide settings for the vendor-specific floating-point library. Check the setting for XILINX LOGICORE library.

```
fc.VendorLibrarySettings
```

```
ans =

  LatencyDrivenMode with properties:

    LatencyStrategy: 'MIN'
          Objective: 'SPEED'
```

The `Objective` property specifies the `c_optimization` parameter to XILINX LOGICORE. Set the `Objective` property to `AREA`.

```
fc.VendorLibrarySettings.Objective = 'AREA';
fc.VendorLibrarySettings
```

```
ans =

  LatencyDrivenMode with properties:

    LatencyStrategy: 'MIN'
          Objective: 'AREA'
```

Vendor library settings are specific to the library. To see the settings for specific libraries, see `hdlcoder.FloatingPointTargetConfig`.

The `VendorIPConfig` property holds the `IPConfig` object for the vendor-specific library. This object has settings for individual IP modules that you can change by using the `customize` function, such as `Latency` and `ExtraArgs`.

Specify the latency for individual IP modules in the configuration object by using the `customize` function. HDL Coder uses the value of the `Latency` input argument that you set for code generation and optimizations.

```
fc.VendorIPConfig.customize('ADDSUB', 'SINGLE', 'Latency', 11);
fc.VendorIPConfig
```

```
ans =

      Name          DataType      MinLatency    MaxLatency    Latency     ExtraArgs
    __________    __________    __________    __________    _______    __________

    {'ADDSUB'}    {'DOUBLE'}        12            12           -1      {0×0 char}
    {'ADDSUB'}    {'SINGLE'}        12            12           11      {0×0 char}
    {'DIV'   }    {'DOUBLE'}        57            57           -1      {0×0 char}
    {'DIV'   }    {'SINGLE'}        28            28           -1      {0×0 char}
    {'MUL'   }    {'DOUBLE'}         9             9           -1      {0×0 char}
    {'MUL'   }    {'SINGLE'}         8             8           -1      {0×0 char}
    {'RELOP' }    {'DOUBLE'}         2             2           -1      {0×0 char}
    {'RELOP' }    {'SINGLE'}         2             2           -1      {0×0 char}
    {'SQRT'  }    {'DOUBLE'}        57            57           -1      {0×0 char}
    {'SQRT'  }    {'SINGLE'}        28            28           -1      {0×0 char}
```

The latency for the ADDSUB IP becomes 11 instead of the default value 12.

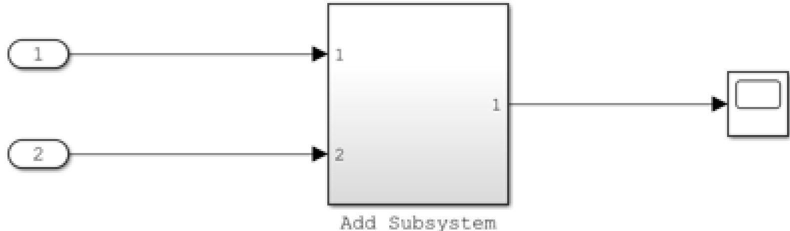
You can specify other IP-specific settings by setting the `ExtraArgs` property for individual IP modules. For example, HDL Coder calls XILINX LOGICORE to generate floating-point IP modules without using any DSP blocks by default. XILINX LOGICORE uses the `c_mult_usage` parameter to control DSP usage. In order to use DSP blocks, pass a different setting by using the `ExtraArgs` input argument to override the default behavior. Because the `ExtraArgs` string is appended to the default IP module generation parameters, it must comply with library setting syntax. Check the IP library documents for parameter usage and syntax.

```
fc.VendorIPConfig.customize('ADDSUB', 'SINGLE', 'ExtraArgs', 'CSET c_mult_usage=Full_Usage');
fc.VendorIPConfig
```

```
ans =

      Name          DataType      MinLatency    MaxLatency    Latency              ExtraArgs
    __________    __________    __________    __________    _______    ___________________________

    {'ADDSUB'}    {'DOUBLE'}        12            12           -1      {0×0 char
    {'ADDSUB'}    {'SINGLE'}        12            12           11      {'CSET c_mult_usage=Full_U
    {'DIV'   }    {'DOUBLE'}        57            57           -1      {0×0 char
    {'DIV'   }    {'SINGLE'}        28            28           -1      {0×0 char
    {'MUL'   }    {'DOUBLE'}         9             9           -1      {0×0 char
    {'MUL'   }    {'SINGLE'}         8             8           -1      {0×0 char
    {'RELOP' }    {'DOUBLE'}         2             2           -1      {0×0 char
    {'RELOP' }    {'SINGLE'}         2             2           -1      {0×0 char
    {'SQRT'  }    {'DOUBLE'}        57            57           -1      {0×0 char
    {'SQRT'  }    {'SINGLE'}        28            28           -1      {0×0 char
```

Open the model `hdlcoder_targetIP_configuration` and set the floating-point configuration object `fc` to the floating-point target configuration on the model.

```
open_system('hdlcoder_targetIP_configuration');
hdlset_param('hdlcoder_targetIP_configuration', 'FloatingPointTargetConfiguration', fc);
```



Run synthesis and mapping by using these commands to confirm the DSP block usage.

```
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx ISE', ...
    'TargetWorkflow','Generic ASIC/FPGA');
hWC.SkipPreRouteTimingAnalysis = true;
hWC.RunTaskAnnotateModelWithSynthesisResult = false;
hWC.GenerateRTLCode = true;
hWC.validate;
hdlcoder.runWorkflow('hdlcoder_targetIP_configuration/Add_Subsystem', hWC);
```

See Also

`hdlcoder.FloatingPointTargetConfig` | `createFloatingPointTargetConfig` | `hdlcoder.FloatingPointTargetConfig.IPConfig` | `customize`

Related Examples

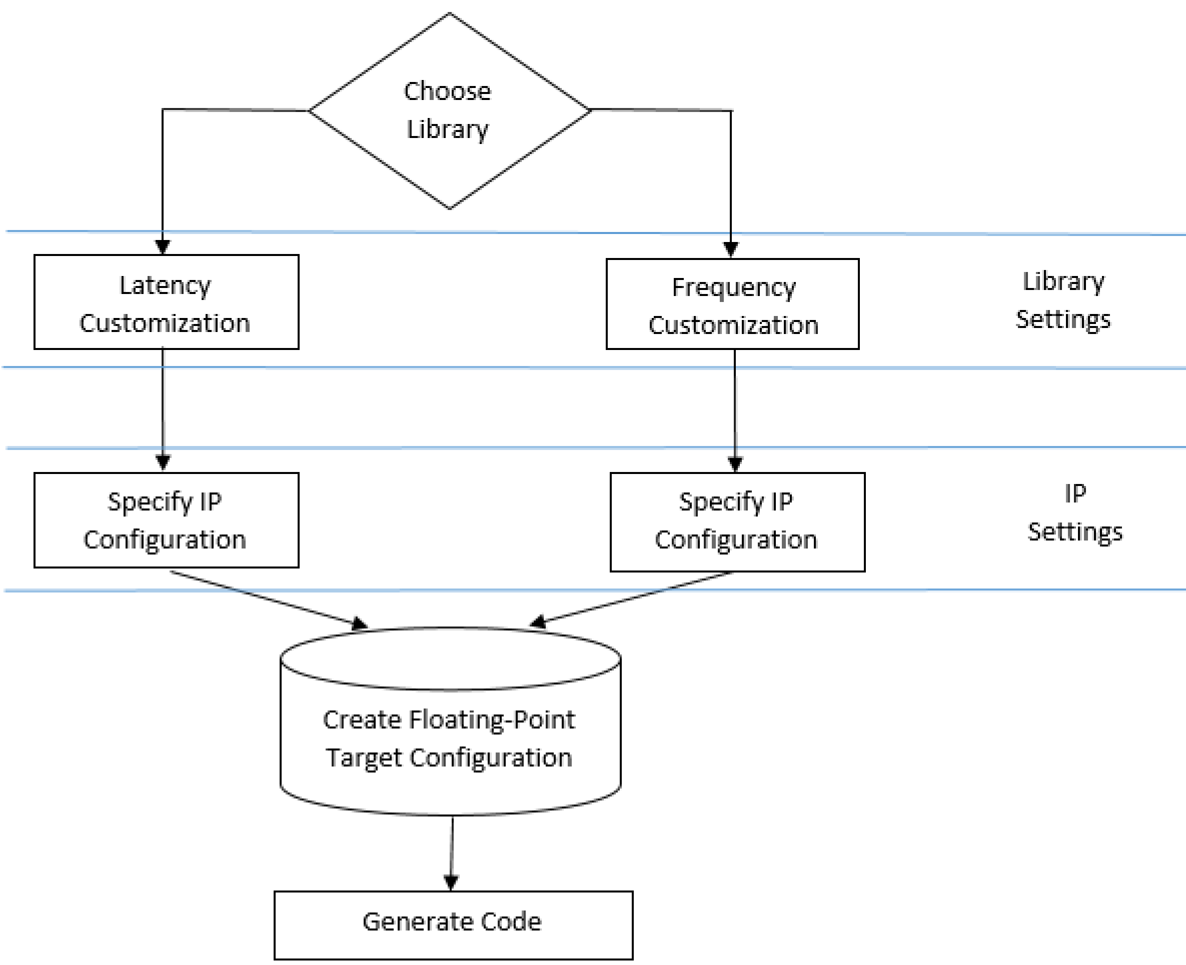
- “Design Model by Using HDL Coder Native Floating Point and Intel Hard Floating Point” on page 14-125

More About

- “Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries” on page 29-19
- “Customize Floating-Point IP Configuration” on page 29-36
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 29-41

Customize Floating-Point IP Configuration

When mapping your Simulink model to floating-point target libraries, you can create a floating-point target configuration with your own custom IP settings. To customize the IP settings, you can use an IP configuration table to choose from different combinations of IP names and data types. The table contains a list of IP types and additional columns that you can use to specify your own custom latency value and other IP settings.



The IP configuration depends on the library settings. The library settings are specific to the floating-point library that you choose. You can customize the IP latency by using the target frequency or the latency strategy setting.

Customize the IP Latency with Target Frequency

To specify the target frequency that you want the IP to achieve, use the `Altera Megafunctions (ALTERA FP Functions)` library. HDL Coder infers the latency of the IP based on the target frequency value. If you do not specify the target frequency, HDL Coder sets the target frequency to `200 MHz`.

You can customize the IP latency by using the **Target Frequency (MHz)** parameter in the Configuration Parameters dialog box or the `TargetFrequency` property from the command line.

Customize the IP Latency in the Configuration Parameters Window

To customize the IP latency by using the **Target Frequency (MHz)** parameter:

1. Specify the vendor-specific floating-point library.
   a. In the **Apps** tab, select **HDL Coder**. In the **HDL Code** tab, click **Settings**.
   b. In the **HDL Code Generation** > **Target** pane, set **Synthesis Tool** to the synthesis tool of your choice. To use the Altera Megafunctions (ALTERA FP Functions) library, set **Synthesis Tool** to `Altera Quartus II` or `Intel Quartus Pro`. For more information on synthesis tools, see "Tool Setup".
   c. In the **HDL Code Generation** > **Floating Point** pane, select **Use Floating Point**.
   d. Set **Vendor Specific Floating Point Library** to `ALTERAFPFUNCTIONS`.
2. Specify the target frequency. In the **Target** pane, set **Target Frequency (MHz)** to the target frequency that you want the floating-point IP to achieve. If you do not specify a target frequency, HDL Coder sets the target frequency to `200 MHz`. Click **OK**.
3. To specify the vendor specific library settings, get the model floating-point target configuration object, `hdlcoder.FloatingPointTargetConfig`, by using this command:

```
fpconfig = hdlget_param(gcs, 'FloatingPointTargetConfiguration')
```

4. Specify the vendor-specific floating-point library settings. Use the floating point target configuration object `fpconfig` to set the `InitializeIPPipelinesToZero` property which specifies whether to initialize pipeline registers in the IP to zero. To avoid potential numerical mismatches in the HDL simulation, it is recommended to leave the `InitializeIPPipelinesToZero` option set to `true`.
5. Specify the IP settings. Use the `VendorIPConfig` property of the `fpconfig` object to specify a custom latency and settings specific to the IP. To view the `VendorIPConfig` table, use this command:

```
fpconfig.VendorIPConfig
```

   - In the **Latency** column, the default latency value of `-1` means that the IP inherits the latency value from the target frequency. If you specify a latency value, HDL Coder tries to map your Simulink model to the IP at a target frequency corresponding to that latency value.
   - In the **ExtraArgs** column, you can specify additional settings specific to the IP.
6. Generate code. In Simulink, in the **HDL Coder** tab, click **Generate HDL Code**.

Customize the IP Latency Programmatically

To customize the IP latency from the command line:

1. Specify the library. Create a `hdlcoder.FloatingPointTargetConfig` object for the floating-point library by using the `hdlcoder.createFloatingPointTargetConfig` function. Then, use `hdlset_param` to save the configuration on the model.

   For example, to create a floating-point target configuration for the `Altera Megafunctions (ALTERA FP FUNCTIONS)` library with the default settings for the model `sfir_single`, enter:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",...
   VendorFloatingPointLibrary="ALTERAFPFUNCTIONS");
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
```

To see the default settings for the floating-point IP, enter `fpconfig`.

```
fpconfig = 

  FloatingPointTargetConfig with properties:

                 Library: 'NATIVEFLOATINGPOINT'
         LibrarySettings: [1×1 fpconfig.NFPLatencyDrivenMode]
                IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
           VendorLibrary: 'ALTERAFPFUNCTIONS'
   VendorLibrarySettings: [1×1 fpconfig.FrequencyDrivenMode]
          VendorIPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

2. Specify the target frequency. If you choose `ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)` as the vendor-specific floating-point library, you can create a floating-point configuration with a custom target frequency. To specify the target frequency for the IP to achieve, use the `TargetFrequency` property. For example:

   ```
   hdlset_param('sfir_single', 'TargetFrequency', 300);
   ```
3. Specify the vendor-specific floating-point library settings. Specify whether you want to initialize the pipeline registers in the IP to zero. Use the `InitializeIPPipelinesToZero` property of the `fpconfig.VendorLibrarySettings` property.

   For example, to set the `InitializeIPPipelinesToZero` property to false, enter:

   ```
   fpconfig.VendorLibrarySettings.InitializeIPPipelinesToZero = false;
   ```

   To see the library settings, enter `fpconfig.VendorLibrarySettings`.

   ```
   ans = 

     FrequencyDrivenMode with properties:

       InitializeIPPipelinesToZero: 0
   ```

   To avoid potential numerical mismatches in the HDL simulation, it is recommended to set `InitializeIPPipelinesToZero` to `true`.
4. Specify the IP settings. Use the `Latency` and `ExtraArgs` input arguments of the `VendorIPConfig.customize` method to customize the latency of the IP and specify additional settings specific to the IP.

   For example, when mapping to the `ADDSUB` IP with Xilinx LogiCORE libraries, to specify a custom latency of 8:

   ```
   fpconfig.VendorIPConfig.customize('ADDSUB', 'SINGLE', 'Latency', 8);
   ```

   To see the IP settings, enter `fpconfig.VendorIPConfig`.

   ```
   ans = 
   ```

```
    Name               DataType              Latency    ExtraArgs
  _________    ________________________    _______    _________

  'ABS'        'DOUBLE'                      -1          ''
  'ABS'        'SINGLE'                      -1          ''
  'ADDSUB'     'DOUBLE'                      -1          ''
  'ADDSUB'     'SINGLE'                       8          ''
  'CONVERT'    'DOUBLE_TO_NUMERICTYPE'       -1          ''
  'CONVERT'    'NUMERICTYPE_TO_DOUBLE'       -1          ''
  'CONVERT'    'NUMERICTYPE_TO_SINGLE'       -1          ''
  'CONVERT'    'SINGLE_TO_NUMERICTYPE'       -1          ''
```

5. Generate HDL code. To generate code from the subsystem, use `makehdl`.

Customize the IP Latency with Latency Strategy

To customize the IP latency with the latency strategy setting, use the `ALTERA MEGAFUNCTION (ALTFP)` or `XILINX LOGICORE` libraries. Specify whether to map your Simulink model to maximum or minimum latency. HDL Coder infers the latency of the IP from the latency strategy setting.

To customize the IP latency from the command line:

1. Specify the library. Create a `hdlcoder.FloatingPointTargetConfig` object for the floating-point library by using the `hdlcoder.createFloatingPointTargetConfig` function. Then, use `hdlset_param` to save the configuration on the model.

   For example, to create a floating-point target configuration for the `ALTERA MEGAFUNCTION (ALTFP)` library with the default settings for the model `sfir_single`, enter:

   ```
   fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",...
      VendorFloatingPointLibrary="ALTFP");
   hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
   ```

   By default, the library uses the minimum latency and speed objective for the floating-point IP.
2. Specify the library settings. Customize the library settings by setting the `Objective` and `LatencyStrategy` properties of the `fpconfig.VendorLibrarySettings` object.

   For example, to customize the `ALTERA MEGAFUNCTION (ALTFP)` library to use the maximum latency and objective as area, enter:

   ```
   fpconfig.VendorLibrarySettings.Objective = 'AREA';
   fpconfig.VendorLibrarySettings.LatencyStrategy = 'MAX';
   ```

   To see the library settings, enter `fpconfig.VendorLibrarySettings`.

   ```
   ans =

     LatencyDrivenMode with properties:

       LatencyStrategy: 'MAX'
             Objective: 'AREA'
   ```
3. Specify the IP settings. Use the `Latency` and `ExtraArgs` input arguments of the `VendorIPConfig.customize` method to customize the latency of the IP and specify additional settings specific to the IP.

For example, when mapping to the ADDSUB IP with Xilinx LogiCORE libraries, to use a custom latency of 8 and specify the DSP resource usage with the cmultusage parameter, enter:

```
fpconfig.VendorIPConfig.customize('ADDSUB', 'SINGLE',...
   'Latency', 8, 'ExtraArgs', 'CSET c_mult_usage=Full_usage');
```

To see the IP settings, enter `fpconfig.VendorIPConfig`.

```
ans =

      Name              DataType             MinLatency    MaxLatency    Latency            ExtraArgs
    _________    ______________________    __________    __________    _______    ______________________________

    'ADDSUB'     'DOUBLE'                    7             14            -1         ''
    'ADDSUB'     'SINGLE'                    7             14             8         'CSET c_mult_usage=Full_usage'
    'CONVERT'    'DOUBLE_TO_NUMERICTYPE'     6              6            -1         ''
    'CONVERT'    'NUMERICTYPE_TO_DOUBLE'     6              6            -1         ''
    'CONVERT'    'NUMERICTYPE_TO_SINGLE'     6              6            -1         ''
```

4 Generate HDL code. To generate code from the subsystem, use makehdl.

See Also

hdlcoder.FloatingPointTargetConfig | createFloatingPointTargetConfig | hdlcoder.FloatingPointTargetConfig.IPConfig | customize

Related Examples

- “FPGA Floating-Point Library IP Mapping” on page 29-25
- “Design Model by Using HDL Coder Native Floating Point and Intel Hard Floating Point” on page 14-125

More About

- “Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries” on page 29-19
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 29-41

HDL Coder Support for FPGA Floating-Point Library Mapping

In this section...

"Supported Blocks That Map to FPGA Floating-Point Target IP" on page 29-41
"Supported Blocks That Do Not Need to Map to FPGA Floating-Point Target IP" on page 29-43
"Limitations for FPGA Floating-Point Library Mapping" on page 29-44

In the HDL Coder block library, a subset of Simulink blocks support floating-point library mapping. The subset includes:

- Blocks that perform basic math operations such as addition, multiplication, and complex trigonometric sine and cosine functions. These blocks map to one or more floating-point IP units on the target FPGA device.
- Discrete blocks, blocks that perform signal routing, and blocks that perform math operations such as matrix concatenation. These blocks need not map to a floating-point IP unit on the target FPGA device.

Supported Blocks That Map to FPGA Floating-Point Target IP

The following table summarizes the Simulink blocks that can map to FPGA floating-point IP cores.

When mapping to floating-point IP cores, some blocks have mode restrictions.

Note Some blocks do not map to a floating-point IP core in the third-party hardware. For example, the Abs block maps to an Altera target IP core but not to a Xilinx target IP core.

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Abs	✓		—
Add	✓	✓	—
Bias	✓	✓	—
Compare To Constant	✓	✓	—
Compare To Zero	✓	✓	—

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Data Type Conversion	✓	✓	• Conversions between single and double data types are not supported. • **Integer rounding mode** attribute in the Block Parameters dialog box must be set to `Nearest`. • If you use Altera Megafunction IP for conversion between floating-point and fixed-point data types, the input bitwidth must be between `16` and `128` bits.
Decrement Real World	✓	✓	—
Discrete FIR Filter	✓	✓	—
Discrete Transfer Fcn	✓	✓	—
Discrete-Time Integrator	✓	✓	—
Divide	✓	✓	—
Dot Product	✓	✓	
Gain	✓	✓	—
Math Function	✓		• Set the **Function** attribute in Block Parameters dialog box to either `reciprocal`, `log` or `exp`.
MinMax	✓	✓	—
Multiply-Add	✓	✓	—
Product	✓	✓	• Product block with more than two inputs is not supported.
Product of Elements	✓	✓	• The **Architecture** in HDL Block Properties must be set to `Tree`.
Reciprocal Sqrt	✓		—
Relational Operator	✓	✓	—
Sqrt	✓	✓	—
Subtract	✓	✓	—
Sum	✓	✓	• Sum block with `-` ports is not supported. • The block cannot have more than two inputs.
Sum of Elements	✓	✓	• The **Architecture** in HDL Block Properties must be set to `Tree`.

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Trigonometric Function	✓		• Only single data types are supported for floating-point library mapping. • In the Block Parameters dialog box, **Function** must be set to either `sin` or `cos` and **Approximation method** must be set to `None`. • If you are using Altera Quartus 10.1 or 11.0, turn on the `AlteraBackward Incompatible SinCosPipeline` global property using `hdlset_param`.
Unary Minus	✓	✓	—

Supported Blocks That Do Not Need to Map to FPGA Floating-Point Target IP

Following are the Simulink blocks that generate HDL code but need not map to an FPGA floating-point IP core.

- Bus Assignment
- Bus Creator
- Bus Selector
- Constant
- Delay
- Demux
- Deserializer1D
- Downsample
- From
- Goto
- Index Vector
- Vector Concatenate, Matrix Concatenate
- Memory
- Model Info
- Multiport Switch
- Mux
- Rate Transition
- Reshape

- Serializer1D
- Subsystem, Atomic Subsystem, CodeReuse Subsystem
- Switch block with control input other than `u2 ~= 0`.
- Unit Delay
- Upsample
- Zero-Order Hold

Limitations for FPGA Floating-Point Library Mapping

- If your synthesis tool is Xilinx Vivado, you cannot use FPGA floating-point library mapping.
- Complex data types are not supported.
- The streaming optimization is not supported with floating-point library mapping.
- The resource sharing optimization is not supported with Unary Minus and Abs blocks.
- For IP Core Generation, and Simulink Real-Time FPGA I/O workflows, your DUT ports cannot use floating-point data types.

See Also

Related Examples

- “FPGA Floating-Point Library IP Mapping” on page 29-25

More About

- “Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries” on page 29-19
- “Customize Floating-Point IP Configuration” on page 29-36

Synthesis Objective to Tcl Command Mapping

In this section...

"Altera Quartus II" on page 29-45
"Xilinx Vivado 2014.4" on page 29-45
"Xilinx ISE 14.7 with PlanAhead" on page 29-46

The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

When you specify a synthesis objective in the HDL Workflow Advisor **Synthesis objective** field, or in the HDL Workflow CLI workflow `hdlcoder.Objective`, the HDL Coder software generates Tcl commands that are specific to your synthesis tool.

Altera Quartus II

Synthesis objective	Tcl Commands
Area Optimized	`set_global_assignment -name OPTIMIZATION_TECHNIQUE "Area"` `set_global_assignment -name FITTER_EFFORT "Standard Fit"`
Compile Optimized	`set_global_assignment -name OPTIMIZATION_TECHNIQUE "Balanced"` `set_global_assignment -name FITTER_EFFORT "Fast Fit"`
Speed Optimized	`set_global_assignment -name OPTIMIZATION_TECHNIQUE "Speed"` `set_global_assignment -name FITTER_EFFORT "Standard Fit"`

Xilinx Vivado 2014.4

If your tool version is different, the Tcl commands are slightly different.

Synthesis objective	Tcl Commands
Area Optimized	`set_property strategy {Vivado Synthesis Defaults} [get_runs synth_1]` `set_property strategy "Area_Explore" [get_runs impl_1]`

Synthesis objective	Tcl Commands
Compile Optimized	`set_property strategy "Flow_RuntimeOptimized" [get_runs synth1]` `set_property strategy "Flow_Quick" [get_runs impl_1]`
Speed Optimized	`set_property strategy {Vivado Synthesis Defaults} [get_runs synth_1]` `set_property strategy "Performance_Explore" [get_runs impl_1]`

Xilinx ISE 14.7 with PlanAhead

If your tool version is different, the Tcl commands are slightly different.

Synthesis objective	Tcl Commands
Area Optimized	`set_property strategy "AreaReduction" [get_runs synth_1]` `set_property strategy "MapCoverArea" [get_runs impl_1]`
Compile Optimized	`set_property strategy "{XST Defaults}" [get_runs synth_1]` `set_property strategy "{ISE Defaults}" [get_runs impl_1]`
Speed Optimized	`set_property strategy "TimingWithIOBPacking" [get_runs synth_1]` `set_property strategy "MapTiming" [get_runs impl_1]`

See Also

Related Examples

- “Getting Started with the HDL Workflow Advisor” on page 29-5
- “HDL Workflow Advisor Tasks” on page 36-2

Run HDL Workflow with a Script

In this section...

"Export an HDL Workflow Script" on page 29-48
"Specify Verbosity of Workflow Script" on page 29-48
"Enable or Disable Tasks in HDL Workflow Script" on page 29-48
"Run a Single Workflow Task" on page 29-48
"Import an HDL Workflow Script" on page 29-49
"Generic ASIC/FPGA Workflow Script Example" on page 29-49
"FPGA-in-the-Loop Script Example" on page 29-50
"IP Core Generation Workflow Script Example" on page 29-51
"Simulink Real-Time FPGA I/O Workflow Example" on page 29-53

The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

HDL Workflow Advisor is not available in Simulink Online.

To run the HDL workflow as a command-line script, configure and run the HDL Workflow Advisor with your Simulink design, then export a script. The script uses HDL Workflow CLI commands to perform the same tasks as the HDL Workflow Advisor, including FPGA bitstream or synthesis project generation.

You can export an HDL workflow script for these target workflows:

- `Generic ASIC/FPGA`
- `FPGA-in-the-Loop` (requires HDL Verifier license)
- `IP Core Generation`
- `Simulink Real-Time FPGA I/O` (requires Simulink Real-Time)

To update an existing script, import it into the HDL Workflow Advisor, modify the tasks, and export the updated script. Alternatively, you can manually edit the script.

Export an HDL Workflow Script

1. In the HDL Workflow Advisor, configure and run all the tasks.
2. Select **File** > **Export to Script**.
3. In the Export Workflow Configuration dialog box, enter a file name and save the script.

   The script is a MATLAB file that you can run from the command line.

Note When you export to script, default values such as `Asynchronous` value for **Reset type** are not exported. When you import from the script, if the model is unchanged, you do not see the default settings in the script.

Specify Verbosity of Workflow Script

You can use the `Verbosity` property of the `hdlcoder.runWorkflow` function to specify the level of detail for progress messages generated as code generation and deployment proceeds. To generate verbose messages while running the workflow for a `hdlcoder.WorkflowConfig` workflow configuration object, `hWC` and Simulink design, `model/DUTname`, set `Verbosity` to `on`.

```
hdlcoder.runWorkflow('model/DUTname', hWC, 'Verbosity', 'on');
```

Enable or Disable Tasks in HDL Workflow Script

To disable all workflow tasks, update the workflow configuration object with the `clearAllTasks` method.

To reenable all workflow tasks, update the workflow configuration object with the `setAllTasks` method.

Run a Single Workflow Task

To run a single workflow task without rerunning other workflow tasks:

1. Disable all tasks in the workflow configuration object by running the `clearAllTasks` method.
2. In the workflow configuration object, enable the task that you want to run.

For example, if you previously ran an HDL workflow script and generated a bitstream, you can program your target hardware without rerunning the other workflow tasks. To run the target device programming task for an `hdlcoder.WorkflowConfig` workflow configuration object, `hWC` and Simulink design, `model/DUTname`:

1. Run the `clearAllTasks` method.

   ```
   hWC.clearAllTasks;
   ```
2. Enable the target device programming task.

   ```
   hWC.RunTaskProgramTargetDevice = true;
   ```
3. Run the workflow.

   ```
   hdlcoder.runWorkflow('model/DUTname', hWC);
   ```

Import an HDL Workflow Script

1. In the HDL Workflow Advisor, select **File** > **Import from Script**.
2. In the Import Workflow Configuration dialog box, select the script file and click **Open**.

   The HDL Workflow Advisor updates the tasks with the imported script settings.

Note When you import a HDL Workflow Advisor script, make sure you use the same script that was exported from the HDL Workflow Advisor UI.

Generic ASIC/FPGA Workflow Script Example

This example shows how to configure and run an exported HDL workflow script.

This script is a generic ASIC/FPGA workflow script that targets a Xilinx Virtex 7 device. It uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 14:42:37 on 29/03/2018
% This script was generated using the following parameter values:
%     Filename  : 'S:\generic_workflow_example.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%--------------------------------------------------------------------------

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('sfir_fixed', 'GenerateHDLTestBench', 'off');
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7vx485t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg1761');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Generic ASIC/FPGA');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

%Set Properties related to synthesis tool version
hWC.AllowUnsupportedToolVersion  = true;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskRunSynthesis = true;
hWC.RunTaskRunImplementation = false;
hWC.RunTaskAnnotateModelWithSynthesisResult = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
```

```
hWC.GenerateRTLCode = true;
hWC.GenerateTestbench = false;
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set properties related to 'RunTaskRunSynthesis' Task
hWC.SkipPreRouteTimingAnalysis = false;

% Set properties related to 'RunTaskRunImplementation' Task
hWC.IgnorePlaceAndRouteErrors = false;

% Set properties related to 'RunTaskAnnotateModelWithSynthesisResult' Task
hWC.CriticalPathSource = 'pre-route';
hWC.CriticalPathNumber =  1;
hWC.AnnotateModel = 'original';
hWC.ShowAllPaths = false;
hWC.ShowDelayData = true;
hWC.ShowUniquePaths = false;
hWC.ShowEndsOnly = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `generic_workflow_example.m`, at the command line, enter:

```
generic_workflow_example
```

FPGA-in-the-Loop Script Example

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is an FPGA-in-the-Loop workflow script that targets a Xilinx Virtex 5 development board and uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
%--------------------------------------------------------------------------
% HDL Workflow Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 15:11:23 on 04/05/2018
% This script was generated using the following parameter values:
%     Filename  : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\hdlworkflow_FIL.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%--------------------------------------------------------------------------

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
```

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Kintex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7k325t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 25);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Xilinx Kintex-7 KC705 development board');
hdlset_param('sfir_fixed', 'Workflow', 'FPGA-in-the-Loop');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','FPGA-in-the-Loop');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = false;
hWC.RunTaskBuildFPGAInTheLoop = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
hWC.GenerateRTLCode = true;
hWC.GenerateTestbench = false;
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskBuildFPGAInTheLoop' Task
hWC.IPAddress = '192.168.0.2';
hWC.MACAddress = '00-0A-35-02-21-8A';
hWC.SourceFiles = '';
hWC.Connection = 'Ethernet';
hWC.RunExternalBuild = true;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
hdlcoder.runWorkflow('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `FIL_workflow_example.m`, at the command line, enter:

```
fil_workflow_example.m
```

IP Core Generation Workflow Script Example

This example shows how to configure and run an exported HDL workflow script.

This script is an IP core generation workflow script that targets the Altera Cyclone V SoC development kit. It uses the Altera Quartus II synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:42:16 on 08/07/2015
% Parameter Values:
%     Filename  : 'S:\ip_core_gen_workflow_example.m'
%     Overwrite : true
%     Comments  : true
```

```
%     Headers   : true
%     DUT       : 'hdlcoder_led_blinking/led_counter'

%% Load the Model
load_system('hdlcoder_led_blinking');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoder_led_blinking', ...
    'HDLSubsystem', 'hdlcoder_led_blinking/led_counter');
hdlset_param('hdlcoder_led_blinking', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_led_blinking', ...
    'ReferenceDesign', 'Default system (Qsys 14.0)');
hdlset_param('hdlcoder_led_blinking', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_led_blinking', 'ResourceReport', 'on');
hdlset_param('hdlcoder_led_blinking', 'SynthesisTool', 'Altera QUARTUS II');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolChipFamily', 'Cyclone V');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolDeviceName', '5CSXFC6D6F31C6');
hdlset_param('hdlcoder_led_blinking', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_led_blinking', ...
    'TargetPlatform', 'Altera Cyclone V SoC development kit - Rev.D');
hdlset_param('hdlcoder_led_blinking', 'Traceability', 'on');
hdlset_param('hdlcoder_led_blinking', 'Workflow', 'IP Core Generation');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter', ...
    'ProcessorFPGASynchronization', 'Free running');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterfaceMapping', 'x"100"');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterfaceOptions', {'RegisterInitialValue', 5});

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterfaceMapping', 'x"104"');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction, ...
    'IOInterfaceOptions', {'RegisterInitialValue', 1});

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface', 'External Port');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', 'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', ...
    'IOInterfaceMapping', 'x"108"');
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back, ...
    'IOInterfaceOptions', {'RegisterInitialValue', 3});

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Altera QUARTUS II', ...
    'TargetWorkflow','IP Core Generation');
```

```
% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

%Set Properties related to synthesis tool version
hWC.AllowUnsupportedToolVersion  = true;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskGenerateSoftwareInterface = false;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskProgramTargetDevice = false;

% Set Properties related to Generate RTL Code And IP Core Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.AreaOptimized;

% Set Properties related to Generate Software Interface Model Task
hWC.OperatingSystem = '';
hWC.AddLinuxDeviceDriver = false;

% Set Properties related to Build FPGA Bitstream Task
hWC.RunExternalBuild = true;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `ip_core_workflow_example.m`, at the command line, enter:

```
ip_core_gen_workflow_example
```

Simulink Real-Time FPGA I/O Workflow Example

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is a `Simulink Real-Time FPGA I/O` workflow script that targets the `Speedgoat IO333-325K` board that uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```
%--------------------------------------------------------------------------
% HDL Workflow Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 18:14:33 on 08/05/2018
% This script was generated using the following parameter values:
%     Filename  : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\hdlworkflow_IO333.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%--------------------------------------------------------------------------

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Kintex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7k325t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 100);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Speedgoat IO333-325K');
hdlset_param('sfir_fixed', 'Workflow', 'Simulink Real-Time FPGA I/O');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Simulink Real-Time FPGA I/O');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';
hWC.ReferenceDesignToolVersion = '2017.4';
hWC.IgnoreToolVersionMismatch = false;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndIPCore' Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;
hWC.GenerateIPCoreTestbench = false;
hWC.CustomIPTopHDLFile = '';
hWC.AXI4RegisterReadback = false;
hWC.IPDataCaptureBufferSize = '128';

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';
hWC.EnableIPCaching = true;

% Set properties related to 'RunTaskBuildFPGABitstream' Task
hWC.RunExternalBuild = false;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;
hWC.CustomBuildTclFile = '';
hWC.ReportTimingFailure = hdlcoder.ReportTiming.Error;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `slrt_workflow_example.m`, at the command line, enter:

```
slrt_workflow_example.m
```

See Also

Functions
`hdlcoder.runWorkflow` | `setAllTasks` | `clearAllTasks`

Classes
`hdlcoder.WorkflowConfig`

Related Examples

- "Getting Started with the HDL Workflow Advisor" on page 29-5
- "HDL Workflow Advisor Tasks" on page 36-2

Get Started with HDL Workflow Command-Line Interface

This example shows how to use the HDL Workflow Advisor to run HDL workflows from the command line and the **Export to script** option.

Introduction

This example is a step-by-step guide that helps introduce you to the HDL Workflow Command Line Interface.

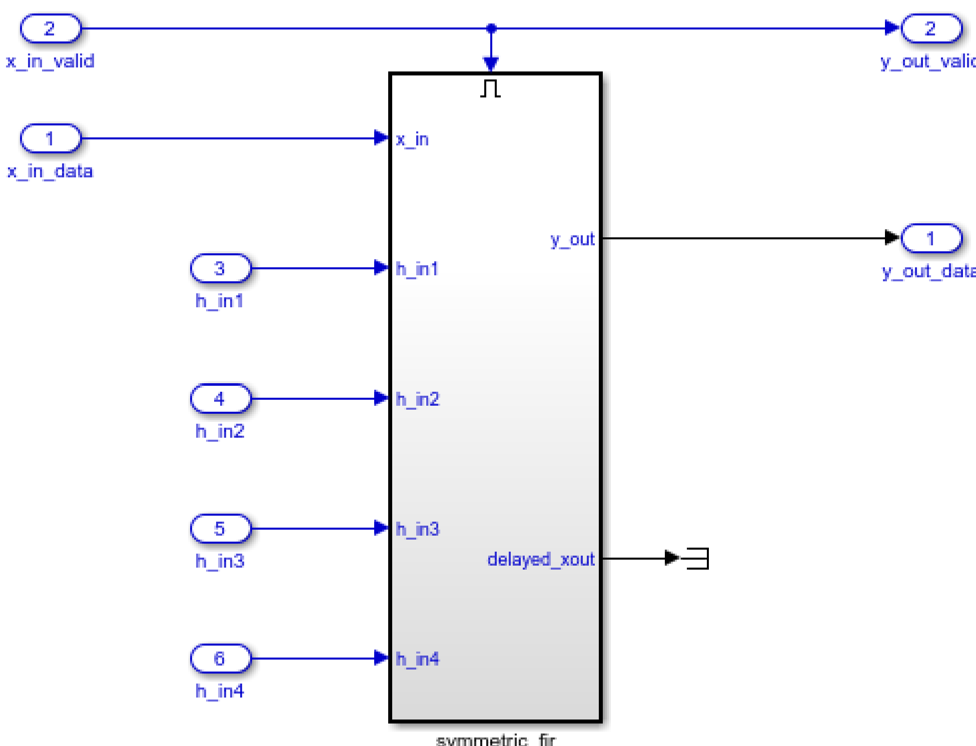
Using the HDL Workflow Command Line Interface, you can run the same sequence of steps and control the same configuration settings that are available in the HDL Workflow Advisor for the following workflows:

1. Generic ASIC/FPGA
2. IP Core Generation
3. Simulink® Real-Time™ FPGA I/O

Open the Model

In this example we will use the **hdlcoder_sfir_fixed_stream** model, but the HDL Workflow Command Line Interface can be used with any model that works with the workflows listed above.

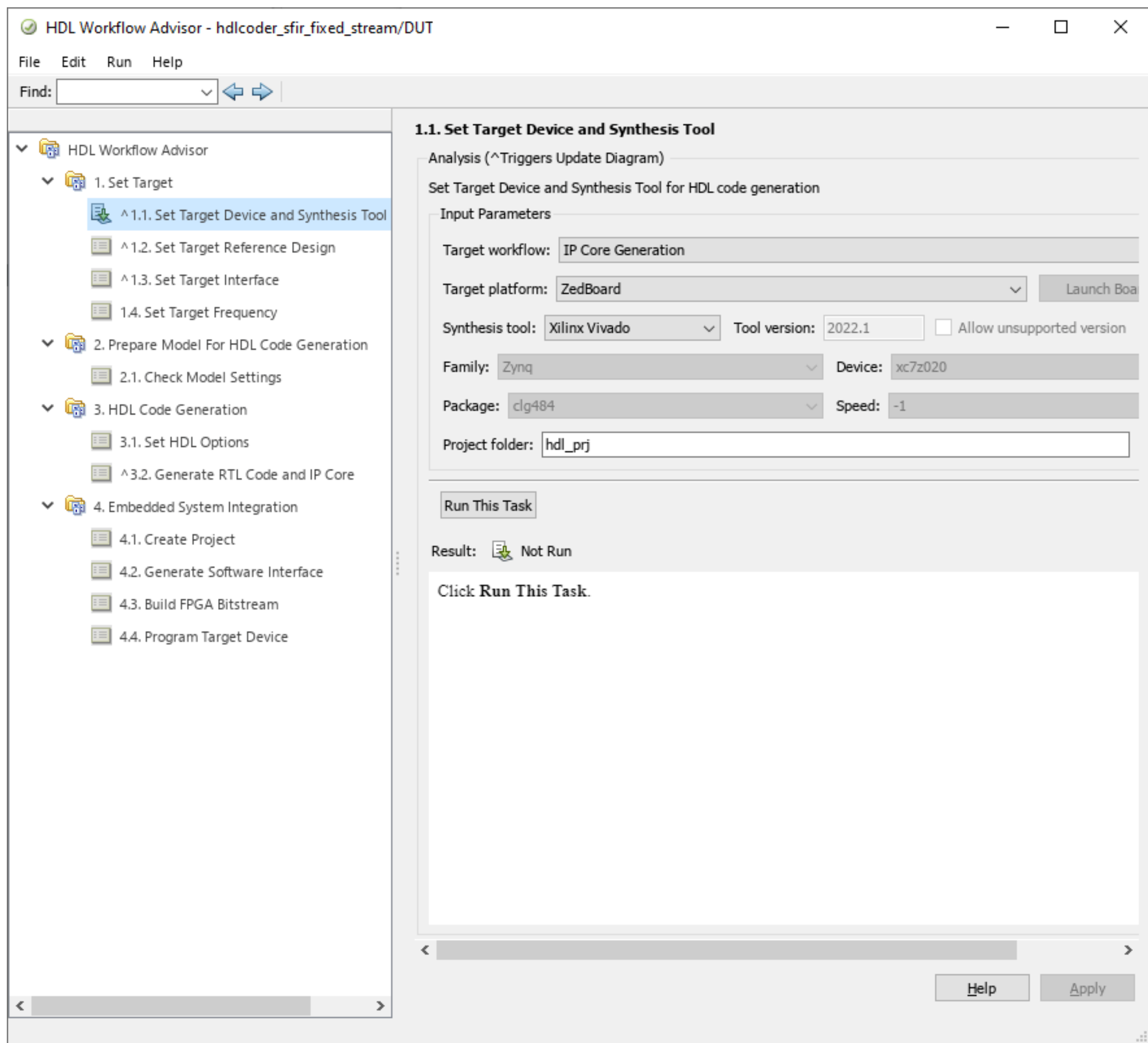
```
open_system('hdlcoder_sfir_fixed_stream')
```



Make sure that the desired third party tools are included on the path. For example, to include Vivado installed locally in its default windows location, use the following command:

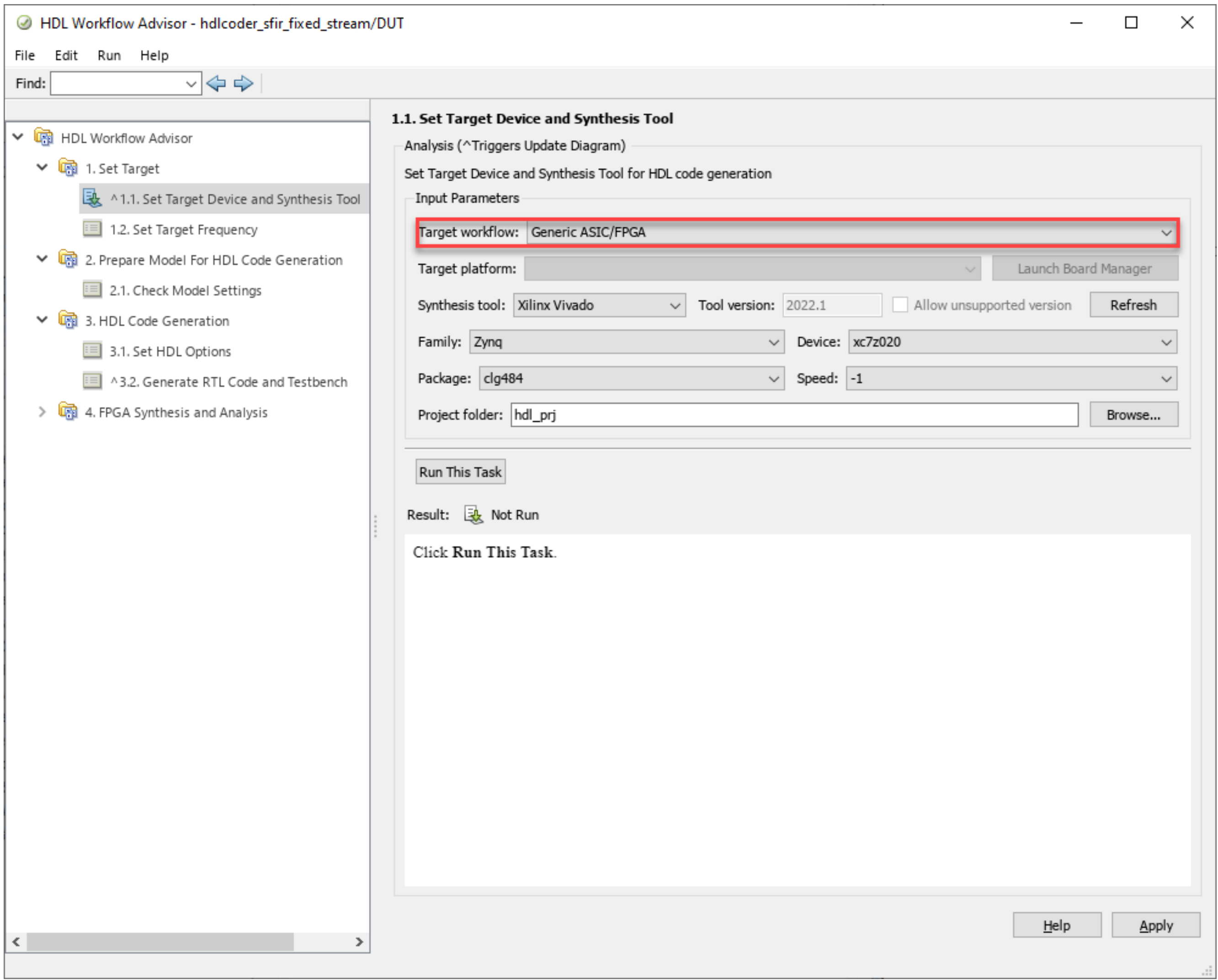
```
>> hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath','C:\Xilinx\Vivado\2022.1\bin\vivado.b
```

Next, launch the workflow advisor and select the appropriate subsystem as the DUT.

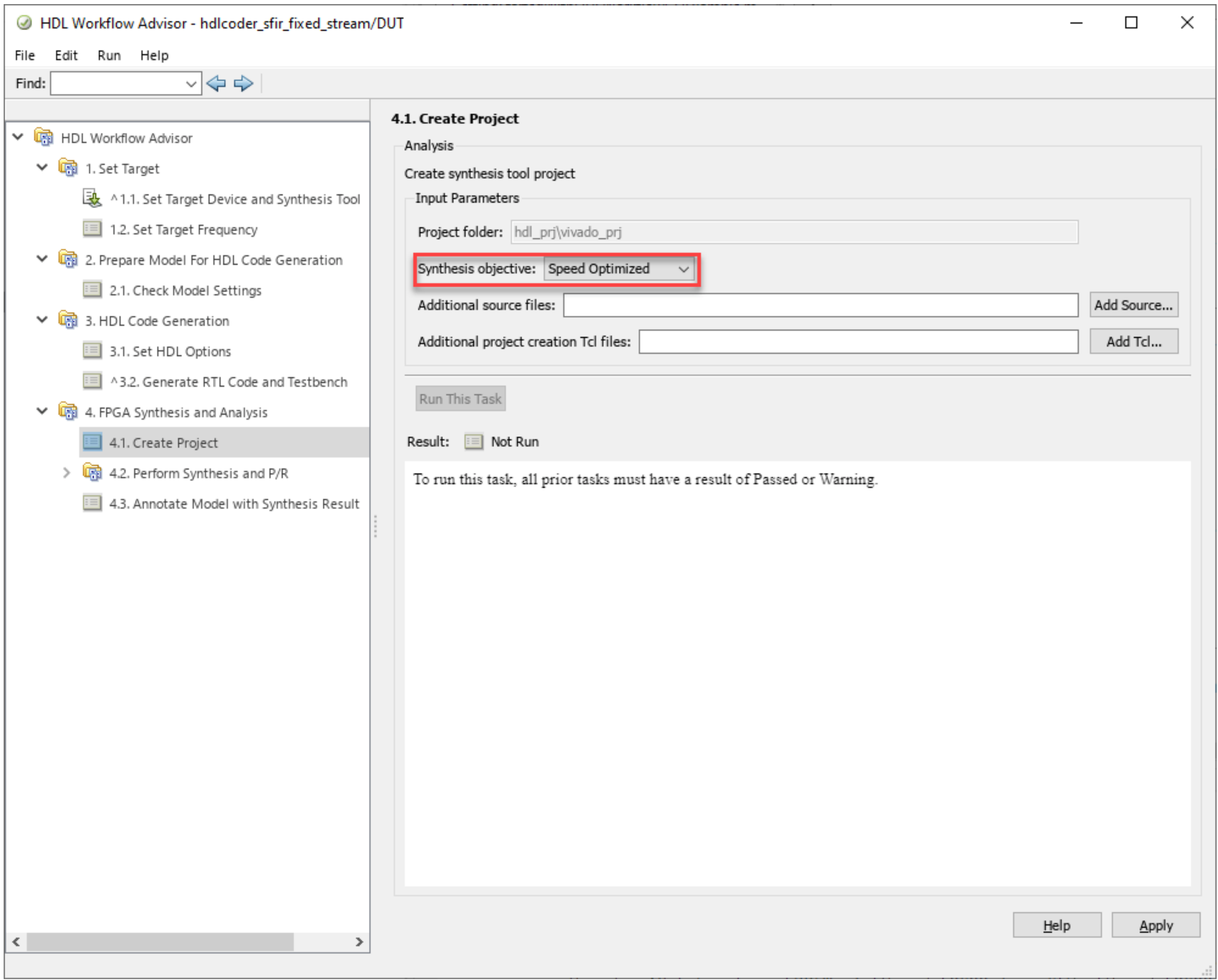


Setup the Workflow

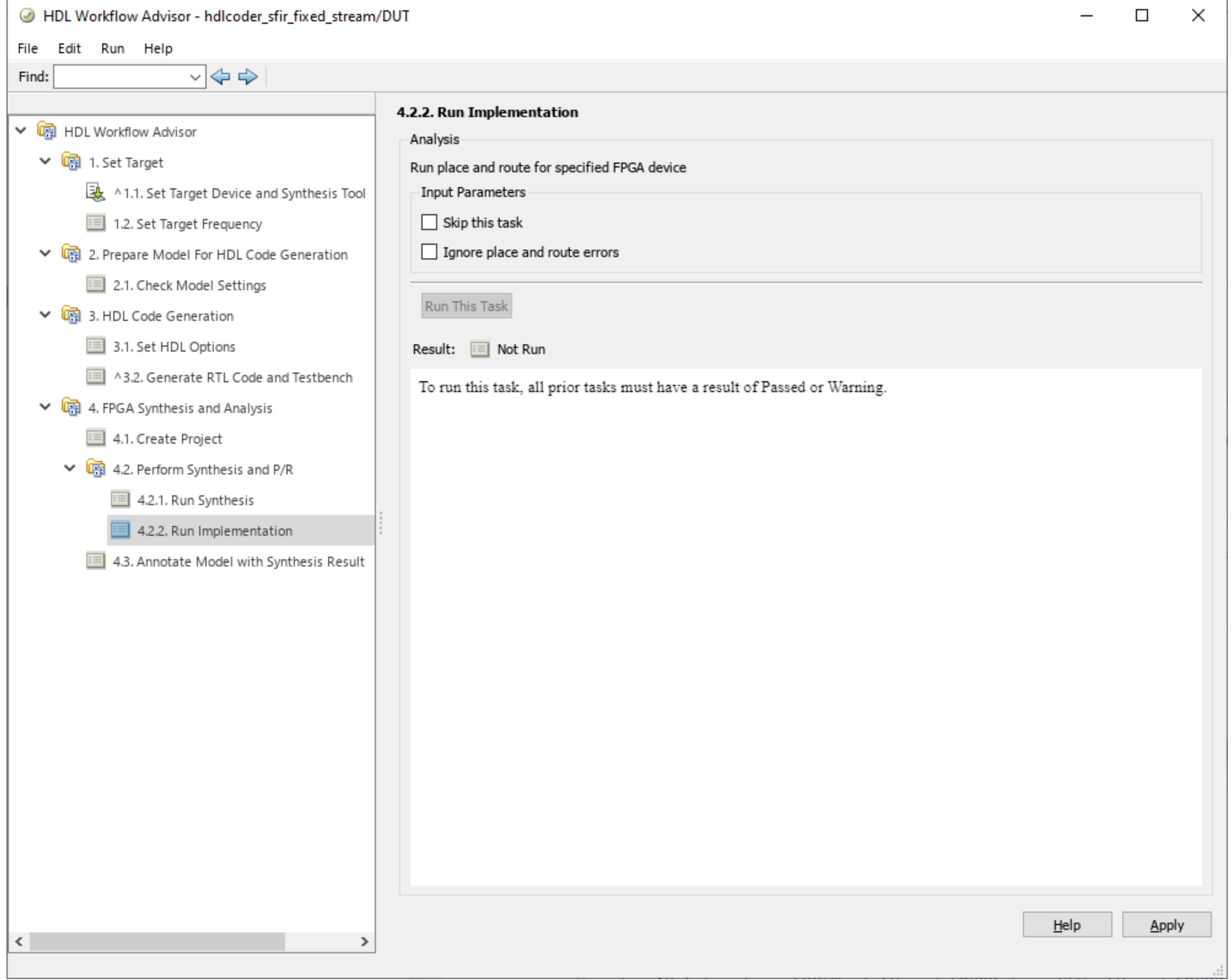
Use the HDL Workflow Advisor to setup your project with your desired settings, such as a Synthesis Tool and Device. Start by changing the workflow to "Generic ASIC/FPGA" so that we can Annotate the model with synthesis results.



You can also specify high level objectives for the synthesis tool. For example, try setting the tool to "Speed Optimized".

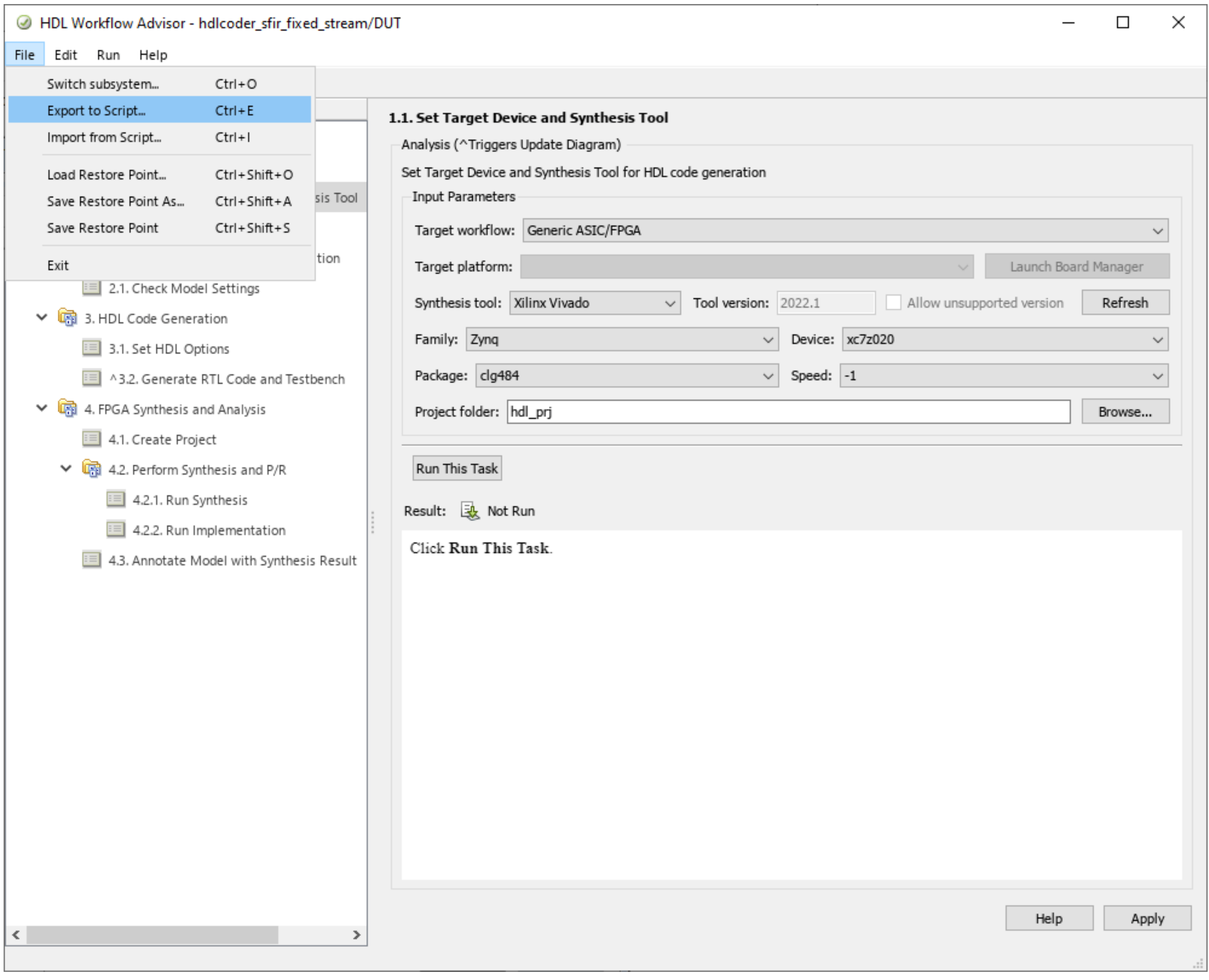


Also, change the "Skip this task" checkbox for Run Implementation so that the exported script runs this step as well.



Export to Script

After all the initial settings have been entered, export your workflow to a script which can be run directly from the command-line for faster design iterations.



Save the file as any name you like. The default will be "hdlworkflow.m". The exported script is shown below:

```
%--------------------------------------------------------------------------
% HDL Workflow Script
% Generated with MATLAB 9.0 (R2016b Prerelease) at 10:40:45 on 31/12/2015
% This script was generated using the following parameter values:
%     Filename  : '/mathworks/devel/sandbox/cberry/work/demo/hdlworkflow.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'hdlcoder_sfir_fixed_stream/DUT'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','hdlcoder_sfir_fixed_stream/DUT');
%--------------------------------------------------------------------------

%   Copyright 2018 The MathWorks, Inc.
```

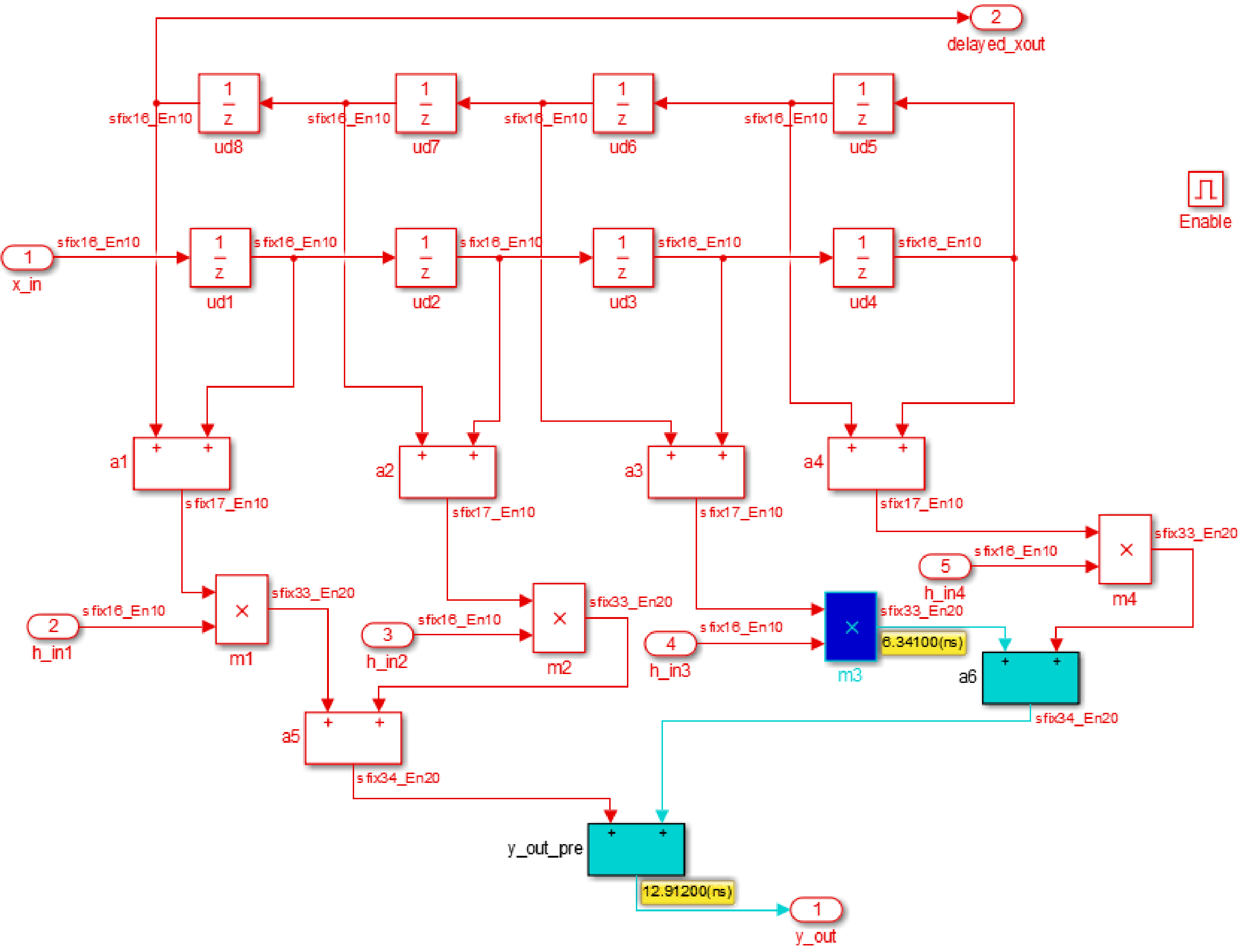
```
%% Load the Model
load_system('hdlcoder_sfir_fixed_stream');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('hdlcoder_sfir_fixed_stream/DUT');

%% Model HDL Parameters
%% Set Model 'hdlcoder_sfir_fixed_stream' HDL parameters
hdlset_param('hdlcoder_sfir_fixed_stream', 'HDLSubsystem', 'hdlcoder_sfir_fixed_stream/DUT');
hdlset_param('hdlcoder_sfir_fixed_stream', 'ReferenceDesign', 'Default system with AXI4-Stream in
hdlset_param('hdlcoder_sfir_fixed_stream', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisToolDeviceName', 'xc7z020');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisToolPackageName', 'clg484');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoder_sfir_fixed_stream', 'TargetDirectory', 'hdl_prj/hdlsrc');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Generic ASIC/FPGA

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskRunSynthesis = true;
hWC.RunTaskRunImplementation = false;
hWC.RunTaskAnnotateModelWithSynthesisResult = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
hWC.GenerateRTLCode = true;
hWC.GenerateRTLTestbench = false;
hWC.GenerateCosimulationModel = false;
hWC.CosimulationModelForUseWith ='Mentor Graphics ModelSim';
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles ='';

% Set properties related to 'RunTaskRunSynthesis' Task
hWC.SkipPreRouteTimingAnalysis = false;

% Set properties related to 'RunTaskRunImplementation' Task
hWC.IgnorePlaceAndRouteErrors = false;

% Set properties related to 'RunTaskAnnotateModelWithSynthesisResult' Task
hWC.CriticalPathSource ='pre-route';
hWC.CriticalPathNumber =  1;
hWC.AnnotateModel ='original';
hWC.ShowAllPaths = false;
hWC.ShowDelayData = true;
hWC.ShowUniquePaths = false;
hWC.ShowEndsOnly = false;
```

```
% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_sfir_fixed_stream/DUT', hWC);
```

Run workflow from Script

Running the script directly will execute your workflow and output a compact set of runtime messages to the cmd window. If you would like to see detailed synthesis tool output information, click on the relevant "Synthesis tool log: " hyperlink under the desired task header to open this file in the MATLAB® editor.

```
>> hdlworkflow
### Workflow begin.
### ++++++++++++++ Task Generate RTL Code and Testbench ++++++++++++++
### Generating HDL for 'hdlcoder_sfir_fixed_stream/DUT'.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_sfir_fixed_stream'.
### Working on hdlcoder_sfir_fixed_stream/DUT/symmetric_fir as hdl_prj\hdlsrc\hdlcoder_sfir_fixed
### Working on hdlcoder_sfir_fixed_stream/DUT as hdl_prj\hdlsrc\hdlcoder_sfir_fixed_stream\DUT.vh
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'hdlcoder_sfir_fixed_stream' complete with 0 errors, 0 warnings, and 0 messages
### HDL code generation complete.
### ++++++++++++++ Task Create Project ++++++++++++++
### Generating Xilinx Vivado 2014.4 project: hdl_prj\vivado_prj\DUT_vivado.xpr
### Synthesis tool log: hdl_prj\hdlsrc\hdlcoder_sfir_fixed_stream\workflow_task_CreateProject.log
### Task "Create Project" successful.
### ++++++++++++++ Task Run Synthesis ++++++++++++++
### Synthesis tool log: hdl_prj\hdlsrc\hdlcoder_sfir_fixed_stream\workflow_task_RunSynthesis.log
### Task "Run Synthesis" successful.
### ++++++++++++++ Task Run Implementation ++++++++++++++
### Synthesis tool log: hdl_prj\hdlsrc\hdlcoder_sfir_fixed_stream\workflow_task_RunImplementation
### Task "Run Implementation" successful.
### ++++++++++++++ Task Annotate Model with Synthesis Result ++++++++++++++
### Parsing the timing file...
### Matched Source = 'hdlcoder_sfir_fixed_stream/DUT/symmetric_fir/m3_out1'
### Matched Destination = 'hdlcoder_sfir_fixed_stream/DUT/y_out_data'
### Highlighting CP 1 from 'hdlcoder_sfir_fixed_stream/DUT/symmetric_fir/m3_out1' to 'hdlcoder_sf
### Click here to reset highlighting.
### Workflow complete.
```



Run workflow interactively

The HDL Workflow Command-Line interface can also be used interactively. For example, after either running the entire script or just the section "Workflow Configuration Settings", the WorkflowConfig object, hWC, will be populated in the workspace:

```
>> hWC =

GenericTurnkeyConfig with properties:

                             SynthesisTool: 'Xilinx Vivado'
                            TargetWorkflow: 'Generic ASIC/FPGA'
                             ProjectFolder: 'hdl_prj'

         RunTaskGenerateRTLCodeAndTestbench: true
                      RunTaskCreateProject: true
                       RunTaskRunSynthesis: true
                  RunTaskRunImplementation: true
    RunTaskAnnotateModelWithSynthesisResult: true
```

```
            TaskGenerateRTLCodeAndTestbench
                            GenerateRTLCode: true
                       GenerateRTLTestbench: false
                  GenerateCosimulationModel: false
                CosimulationModelForUseWith: 'Mentor Graphics ModelSim'
                    GenerateValidationModel: false

                          TaskCreateProject
                                  Objective: hdlcoder.Objective.SpeedOptimized
          AdditionalProjectCreationTclFiles: ''

                           TaskRunSynthesis
                 SkipPreRouteTimingAnalysis: false

                      TaskRunImplementation
                  IgnorePlaceAndRouteErrors: false

       TaskAnnotateModelWithSynthesisResult
                         CriticalPathSource: 'pre-route'
                         CriticalPathNumber: 1
                              AnnotateModel: 'original'
                               ShowAllPaths: false
                              ShowDelayData: true
                            ShowUniquePaths: false
                               ShowEndsOnly: false
```

You can edit this configuration object and then run the workflow with the modified settings. For example, since the task "Run Implementation" was enabled in the previous run, we can change the critical path source to "post-route" with and rerun just the Annotate model task:

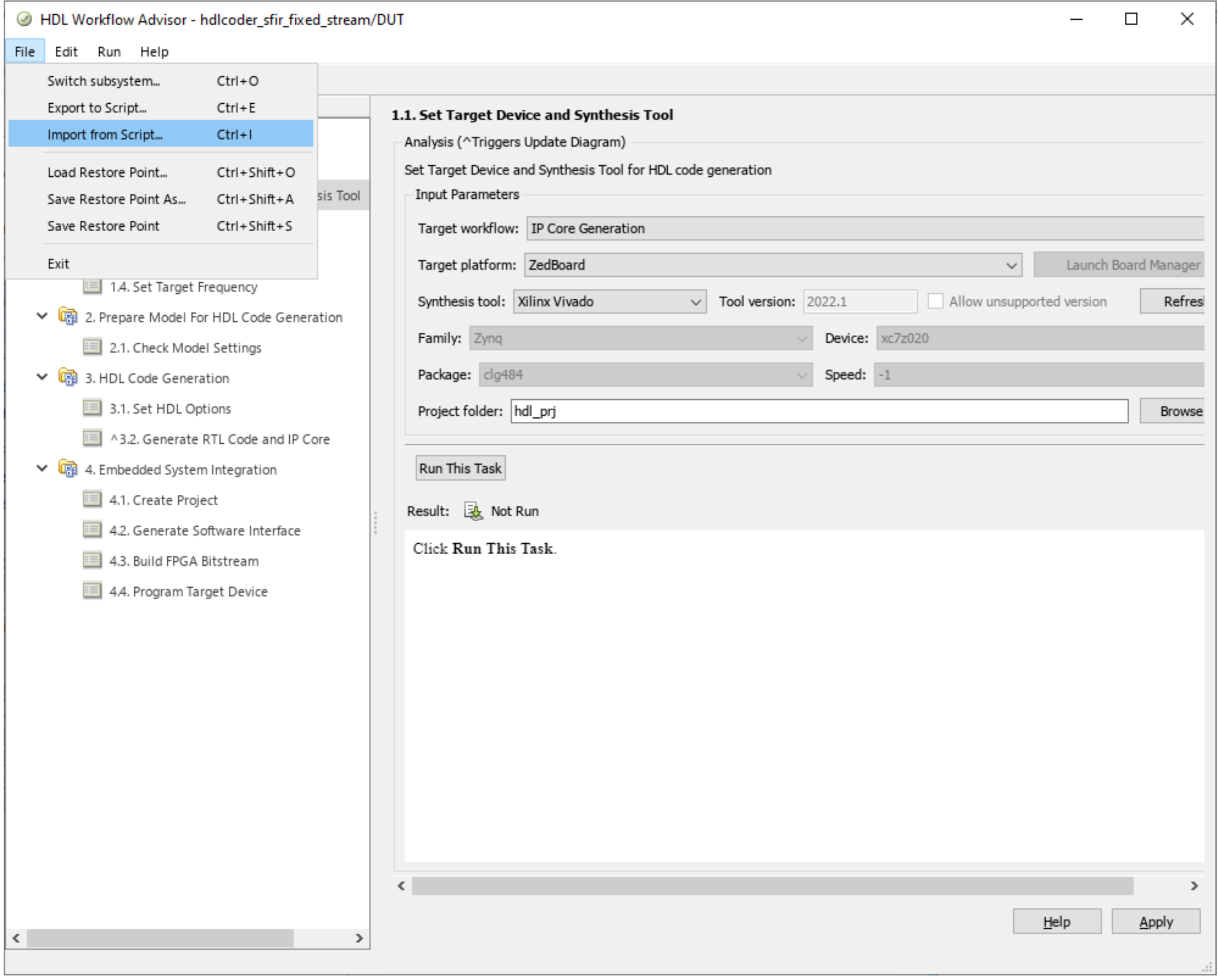
```
>> hWC.clearAllTasks;
>> hWC.RunTaskAnnotateModelWithSynthesisResult = true;
>> hWC.CriticalPathSource = 'post-route';
```

Then run the modified workflow configurations directly using the hdlcoder.runWorkflow command:

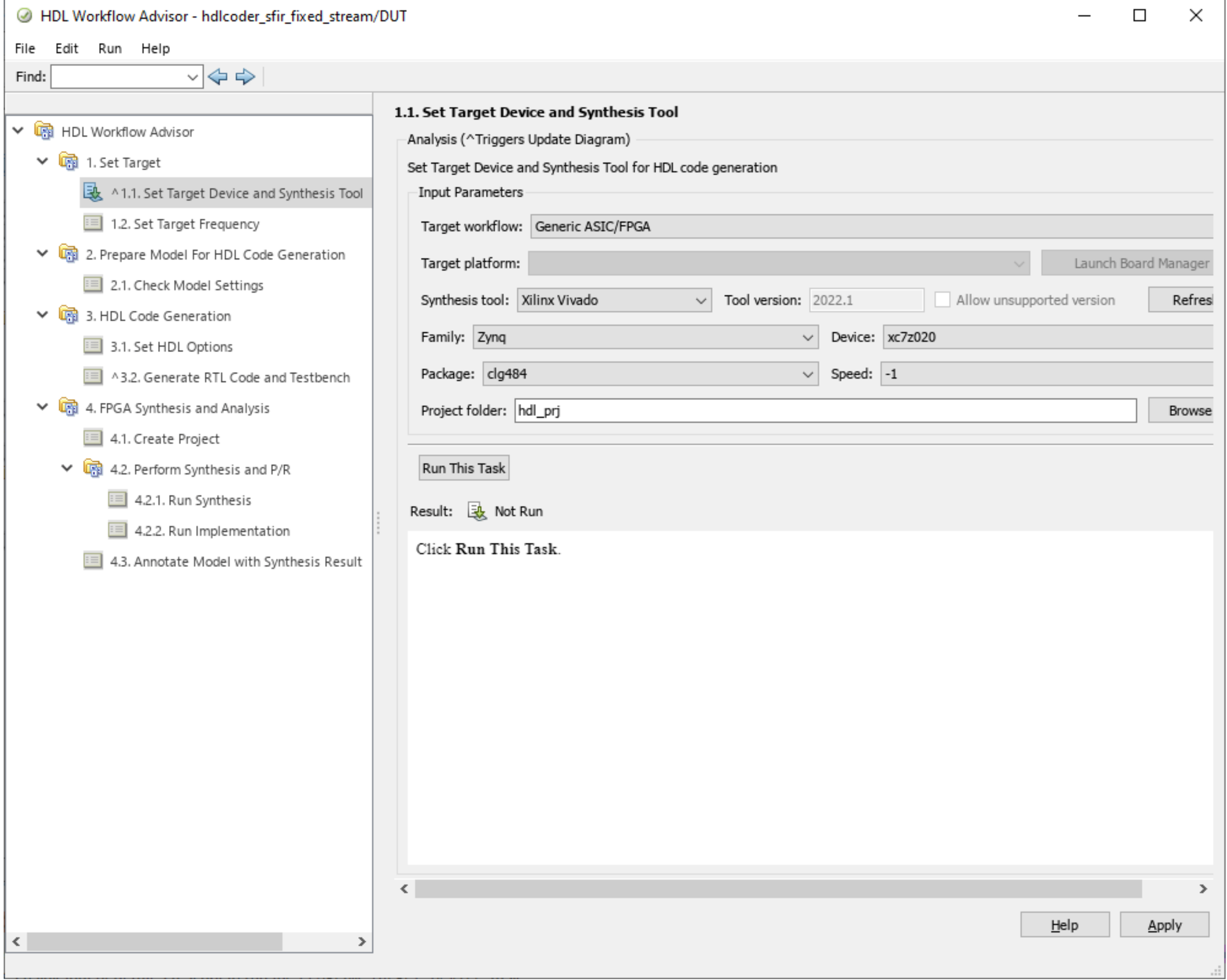
```
>> hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC)
```

Import script into HDL Workflow Advisor

Any changes you make to the exported script can also be imported back into the HDL Workflow Advisor at any time. To do this, make sure the model loaded is the same as the model used in the script, and select "Import from script" from the File menu.



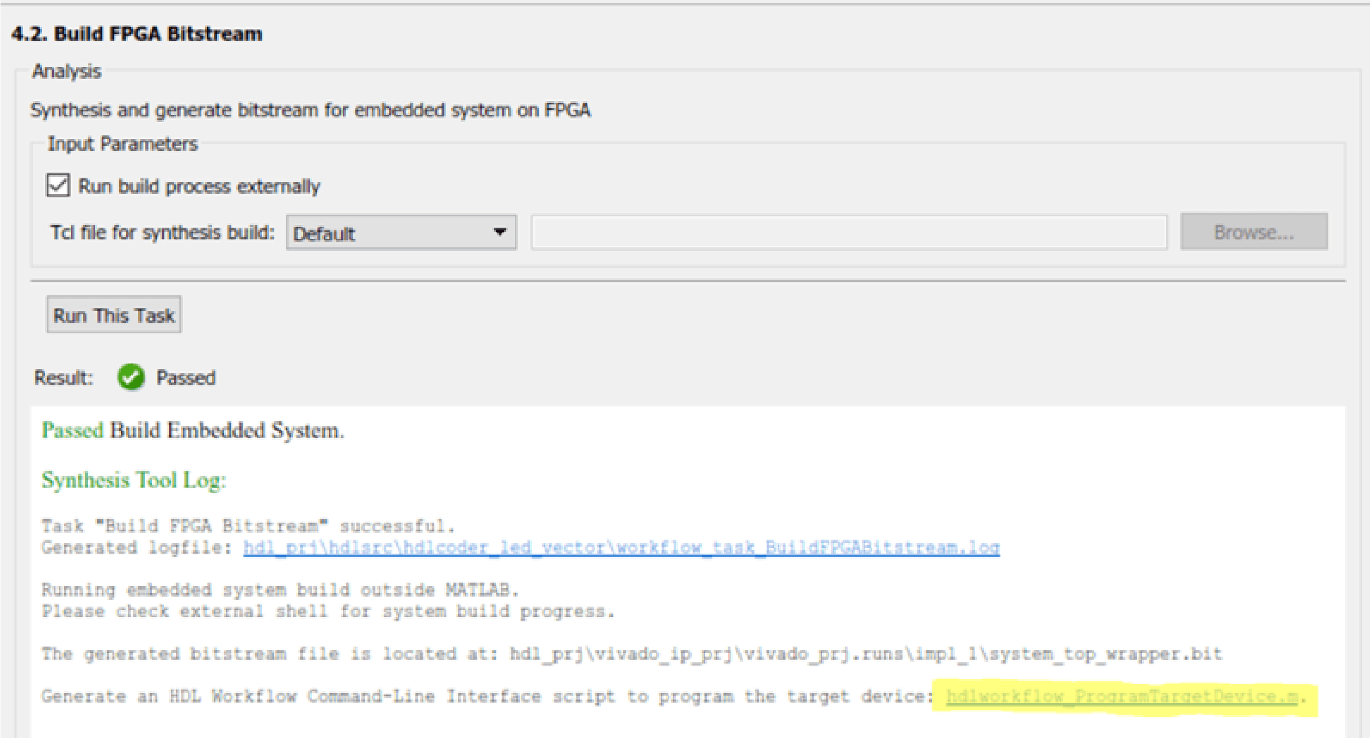
After importing, all of the script settings will be populated in the HDL Workflow Advisor.



Save the HDL Workflow Command-Line Interface programing script for later use

In certain cases, you can avoid re-running all the steps in the HDL Workflow Advisor just to perform specific tasks in the Advisor. For example, when running the IP Core Generation Workflow, after you generate the FPGA bitstream in the `Build FPGA Bitstream` Task, the Advisor provides a link that generates a script to run the `Program Target Device` Task.

Click on the link for `hdlworkflow_ProgramTargetDevice.m` in `Build FPGA Bitstream` Task to generate an HDL Workflow Command-Line Interface script which will execute only the `Program Target Device` Task from an existing hdl_prj directory. Therefore, you can avoid running all the steps in the workflow just to re-program the target device.



The generated script is a standard HDL Workflow CLI script. When running HDL Workflow in command-line interface, the `Program Target Device` Task can be run independent of previous tasks, as long as the FPGA bitstream is already generated.

As a related note, if you are using the `Download` programming method in the `Program Target Device` Task, the HDL Workflow Advisor copies the generated bitstream file onto the SD card on the Zynq or Intel SoC board, so you do not need to re-run the `Program Target Device` Task to download the bitstream. The FPGA bitstream will be reloaded from the SD card automatically during the Linux boot up process.

Summary

The HDL Workflow Command-Line Interface provides an easily scripted alternative to the graphical HDL Workflow Advisor. Workflows can be setup initially using the HDL Workflow Advisor and then exported to script for iterative or automated use.

Simscape to HDL Workflow

- “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 30-2
- “Modeling Guidelines for Simscape Subsystem Replacement” on page 30-9
- “Generate HDL Code for Simscape Models” on page 30-13
- “Generate Optimized HDL Implementation Model from Simscape” on page 30-20
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 30-28
- “Deploy Simscape Buck Converter Model to Speedgoat IO Module Using HDL Workflow Script” on page 30-36
- “Deploy Simscape Grid Tied Converter Model to Speedgoat IO Module Using HDL Workflow Script” on page 30-47
- “Partition Simscape Models Containing a Large Network into Multiple Smaller Networks” on page 30-61
- “Generate HDL Code for Simscape Models with Multiple Networks” on page 30-67
- “Replace Piecewise-Constant Resistor with Switched Linear Components” on page 30-74
- “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82
- “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89
- “Improve FPGA Sampling Frequency of HDL Implementation Model Generated from Simscape Algorithm” on page 30-96
- “Generate HDL Code for Nonlinear Simscape Models by Using Partitioning Solver” on page 30-100
- “Deploy Simscape DC Motor Model to Speedgoat FPGA IO Module” on page 30-106
- “Generate HDL Code for Simscape Three-Phase PMSM Drive Containing Averaged Switch” on page 30-118
- “Generate HDL Code for Two-Speed Transmission Model Containing Mode Charts” on page 30-125
- “Simscape Language Support” on page 30-131
- “Generate HDL Code for Simscape Models by Using Trapezoidal Rule Solver” on page 30-135
- “Generate HDL Code for Simscape Models by Using Linearized Switch Approximation” on page 30-146
- “Estimate Achievable Target Frequency Without Running Synthesis” on page 30-155
- “Generate FPGA Bitstream for Two-Phase DC-DC Converter with Tunable Run-Time Parameters” on page 30-156

Get Started with Simscape Hardware-in-the-Loop Workflow

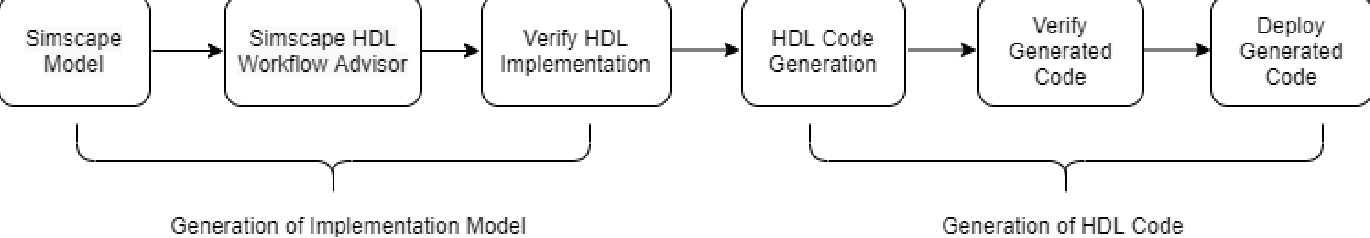
You can convert a Simscape model to HDL code for field-programmable gate array (FPGA) deployment, which requires HDL Coder. The simulation of physical systems is accelerated, enabling you to rapidly prototype models. The conversion also provides real-time simulation of your HDL implementation model by using hardware-in-the-loop (HIL) simulation.

You can perform HIL simulation with smaller time steps and increased accuracy by deploying the Simscape plant models to the generic FPGAs, system on a chip (SoCs), or Speedgoat FPGA I/O modules. Using this workflow, you can:

- Generate an HDL implementation model by using the Simscape HDL Workflow Advisor. The implementation model is a Simulink model that replaces the Simscape components with HDL-compatible Simulink blocks. For more information, see Generate HDL Implementation Model by Using the Simscape HDL Workflow Advisor on page 30-4.
- Generate HDL code for the implementation model, and then deploy the generated code to Speedgoat FPGA I/O modules by using the HDL Workflow Advisor. For more information, see HDL Code Generation and Deployment on page 30-7.

You can model and deploy complex physical systems in Simscape while converting your models into HDL code.

This workflow diagram shows the functionalities at various stages of the Simscape hardware-in-the-loop workflow.



Modeling Physical Systems in Simscape for HDL Compatibility

When you design your Simscape model for compatibility with the Simscape HDL Workflow Advisor, follow these guidelines. Before using this workflow, convert Simscape subsystems into state-space algorithms. For more information, see “Modeling Guidelines for Simscape Subsystem Replacement” on page 30-9.

You can create a Simscape model by using linear, switched linear, and nonlinear blocks.

- Linear blocks are blocks that are defined by a linear relationship such as resistors.
- Switched linear blocks are blocks such as diodes and switches. These blocks are also defined by a linear relationship, such as $V = IR$, where R can switch between two or more values depending on the state of the diodes or switches.
- Nonlinear blocks are blocks that are defined by nonlinear equations, such as nonlinear resistor, nonlinear inductor, DC Motor, and PMSM blocks.

You can choose a solver type (Backward Euler or Partitioning) based on the type of blocks in your model. If you have linear and switched linear blocks in your model, choose the Backward Euler solver

type. If you have nonlinear blocks in your model, choose the Partitioning solver type. For more information, see Solver Configuration.

Configure Solver Settings

To configure the solver settings for HDL code generation, open your model, for example:

```
openExample('plantdeployment/OpenTheSimscapeHDLWorkflowAdvisorExample','supportingFile','sschdlex
open_system('sschdlexBoostConverterExample/Simscape_system')
```

Double-click the Solver Configuration block.

Select Solver type

To choose a solver type for the block, double-click the Solver Configuration block.

1. In the Block Parameters dialog box, select **Use local solver**.
2. From the **Solver type** drop-down list, select `Backward Euler` for linear and switched linear blocks. For nonlinear blocks, select `Partitioning`. For boost converter, select `Backward Euler`.

Specify Sample Time

Similarly you can specify a discrete sample time.

1. In the Block Parameters dialog box, select **Use local solver**.
2. In the **Sample time** text box, specify a discrete sample time Ts.

To verify that the solver settings are specified correctly, run the “Check Solver Configuration” on page 31-3 in the Simscape HDL Workflow Advisor.

Run hdlsetup Function

After creating the model, configure the model for HDL code generation by running the `hdlsetup` function. The function `hdlsetup` sets your model configuration parameters to default values recommended for code generation. Open the model before you invoke the `hdlsetup` function.

To invoke `hdlsetup` function for your `current_model`, at the MATLAB command prompt, enter:

```
hdlsetup('current_model')
```

After running `hdlsetup` function, make sure your model compiles without any errors or warnings.

Simscape Example Models for HDL Code Generation

For HDL code generation, you can design your own Simscape algorithm or choose from a list of example models that are created in Simscape.

Example Models That Use Backward Euler Solver

- Boost converter on page 30-96
- Bridge rectifier on page 30-89
- Buck converter on page 30-9
- Half-wave rectifier on page 30-13

- Solar power inverter model on page 30-61
- Three-phase converter with grid on page 30-47
- Two-level converter ideal on page 30-28
- Variable resistor on page 30-74
- Vienna rectifier on page 30-20

Example Models That Use Partitioning Solver

- Permanent magnet synchronous motor (PMSM) on page 30-100
- DC motor on page 30-106
- Two-speed transmission on page 30-125
- Three-phase PMSM drive on page 30-118

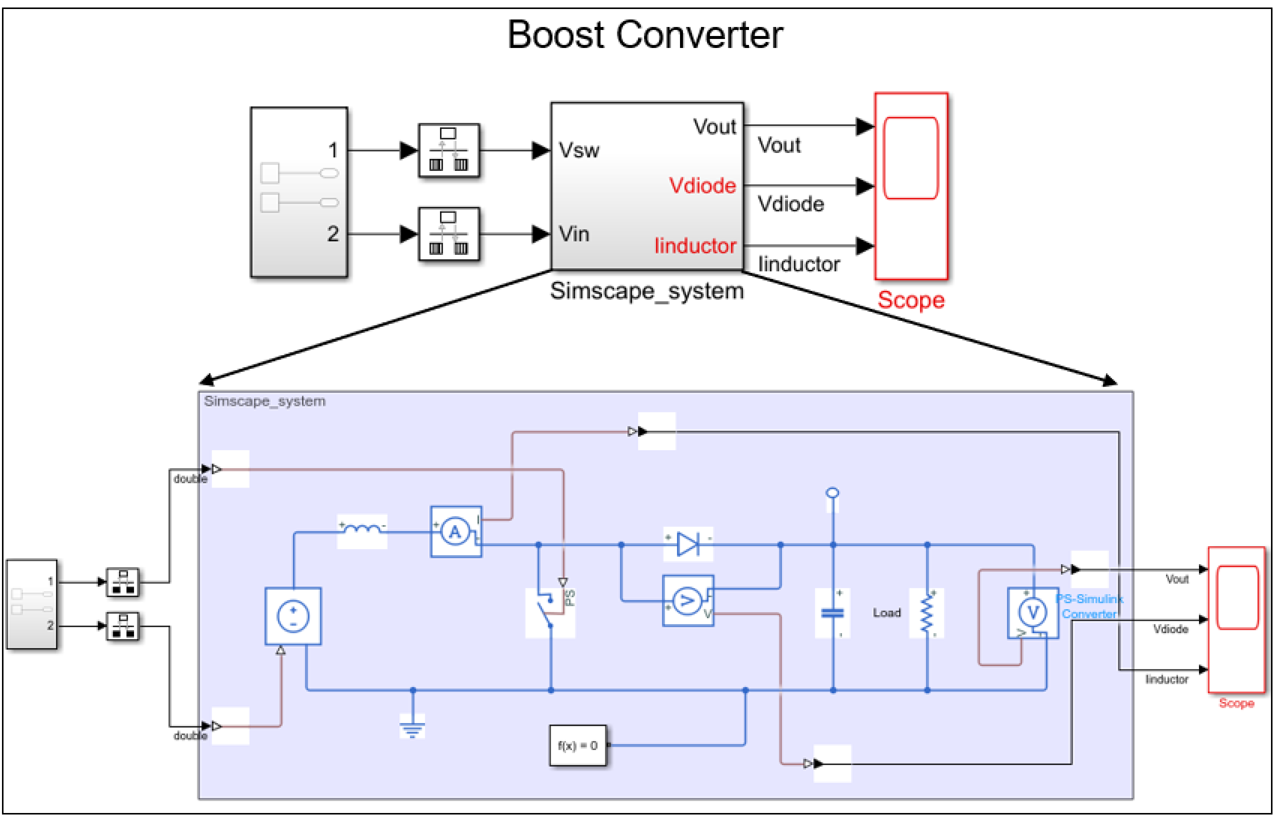
All examples are prefixed with `sschdlex` and postfixed with `Example`. For example, to open the boost converter model, at the MATLAB command prompt, enter:

```
openExample('plantdeployment/OpenTheSimscapeHDLWorkflowAdvisorExample','supportingFile','sschdlex
open_system('sschdlexBoostConverterExample/Simscape_system')
```

You can also find many other application-specific example models. To get a list of all the Simscape models designed for HDL code generation, type `sschdlex` in the MATLAB Command Window and press the **Tab** key.

Generate HDL Implementation Model by Using the Simscape HDL Workflow Advisor

You first generate an HDL implementation model from the Simscape model by using the Simscape HDL Workflow Advisor.



To generate an HDL implementation model, open the Simscape HDL Workflow Advisor.

At the MATLAB command prompt, run the `sschdladvisor` function:

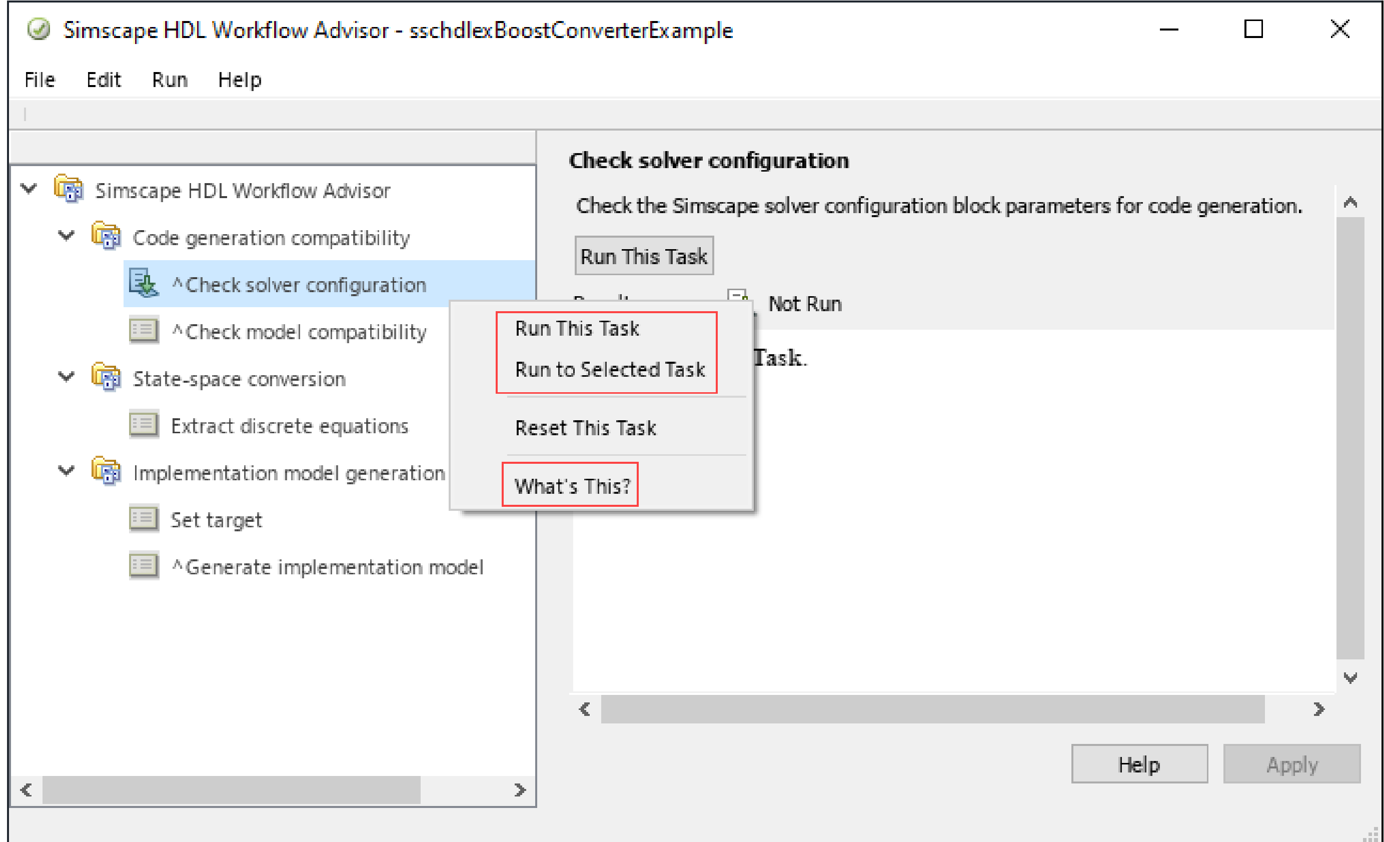
```
sschdladvisor('current_model')
```

This function opens the Simscape HDL Workflow Advisor for the `current_model`.

Run Tasks in the Simscape HDL Workflow Advisor

In the Simscape HDL Workflow Advisor, the left pane lists the folders containing a group or category of related tasks. Expanding the folders shows available tasks in each folder. From the left pane, you can select a folder or an individual task and see the related information in the right pane. For the tasks in the left pane, the right pane contains controls for running the task, a display area for status messages, and other task results. To learn more about each individual task, right-click that task, and select **What's This?** from the list.

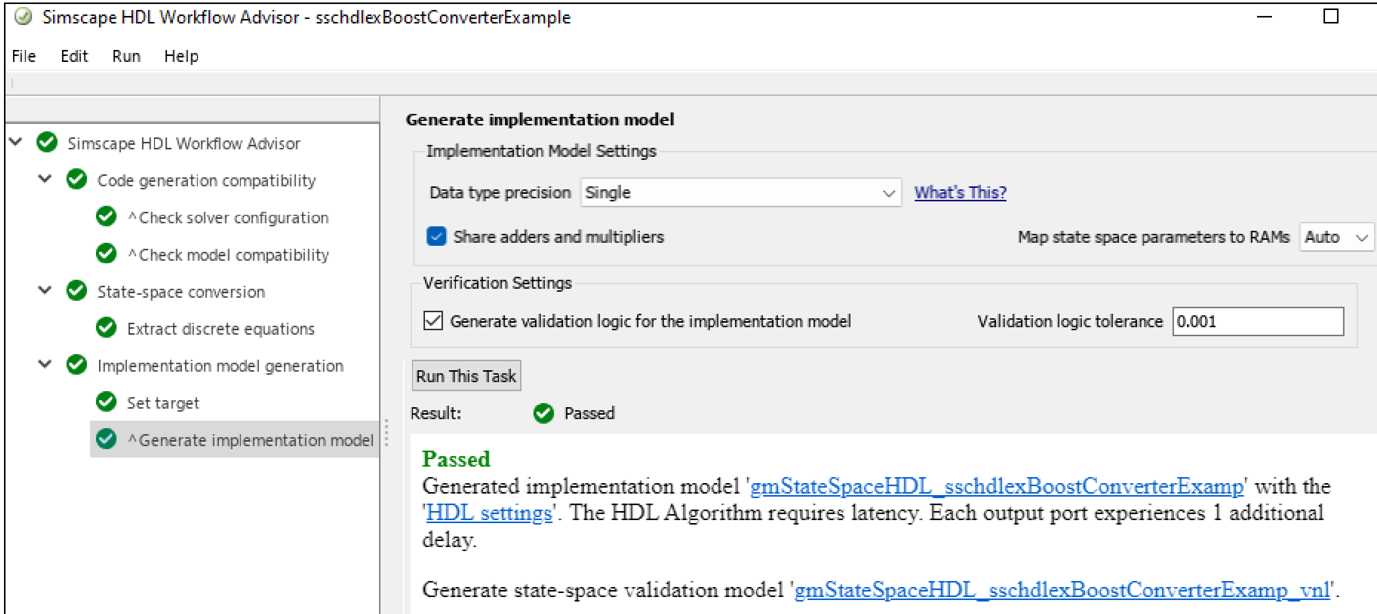
In the Simscape HDL Workflow Advisor window, you can run an individual task or all the tasks in the list. Select the task that you want to run and click **Run This Task**. To run a task, all tasks before it must have run successfully. To run the workflow to a specific task inside a subfolder, expand that folder, and then right-click the task and select **Run to Selected Task**.



To run the workflow and compare functionality of the HDL implementation model with the original Simscape algorithm:

1 Select the **Generate implementation model** task.
2 Under **Verification Settings**, select the **Generate validation logic for the implementation model** check box. This enables the **Validation logic tolerance** which is set with the default value of `0.001`.
3 Right-click the **Generate implementation model** task and select **Run to Selected Task** from the list.

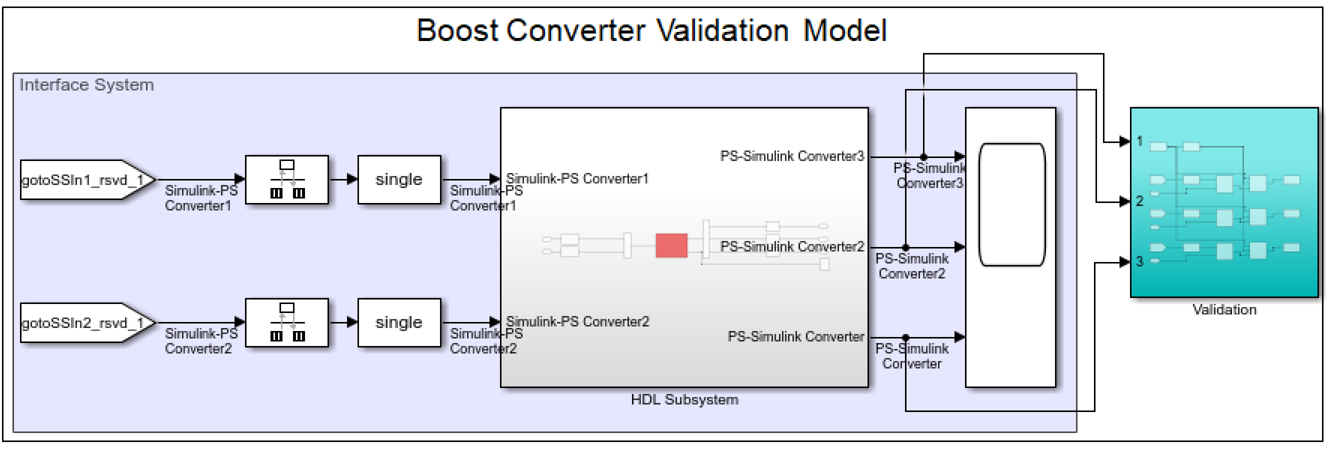
After the task passes, you see a link to the HDL implementation model and a state-space validation model. The implementation model has the same name as the original Simscape model and uses the prefix `gmStateSpaceHDL`. The state-space validation model has the same name as the implementation model and uses the postfix `_vnl`.



For more information on Simscape HDL Workflow Advisor tasks, see "Simscape HDL Workflow Advisor Tasks" on page 31-2.

HDL Code Generation and Deployment

For HDL code generation, you can verify the numeric results of the HDL implementation model and your original Simscape algorithm by using the generated state-space validation model.



Before you generate HDL code, validate the generated HDL implementation model.

Validate HDL Algorithm

You can compare the functionality of the HDL implementation model to the original Simscape algorithm. To validate the HDL implementation model, simulate the state-space validation model. If the simulation does not result in any assertion or warning, then it indicates that the output of the HDL implementation model matches the original Simscape algorithm within the specified tolerance.

For more information, see “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89.

Generate Code

Once you validate the HDL algorithm, you can then generate HDL code for the implementation model. At the MATLAB command prompt, enter:

```
makehdl('gmStateSpaceHDL_current_model/HDL Subsystem')
```

For more information, see “Generate HDL Code for Simscape Models” on page 30-13.

After generating the HDL code, you can deploy the real-time model onto your target hardware. For more information on how to deploy the code, see “Deploy Simscape Buck Converter Model to Speedgoat IO Module Using HDL Workflow Script” on page 30-36.

Restrictions for HDL Code Generation from Simscape Models

HDL Coder does not support code generation from Simscape networks that use:

- `events`.
- `delay`.
- Run-time parameters (Simscape).
- Simscape Multibody™ blocks.
- Simscape Electrical™ Specialized Power Systems blocks.
- Time-varying Simscape source blocks, such as the PS Counter, PS Random Number, PS Repeating Sequence, and PS Uniform Random Number blocks.
- Partitions that have nonlinear equation type. For more information, see “Understanding How the Partitioning Solver Works” (Simscape).
- GCC compiler `-ffast-math` option.

See Also

`makehdl` | `sschdladvisor` | `hdladvisor`

More About

- “Generate HDL Code for Simscape Models” on page 30-13
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 31-7
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 30-28

Modeling Guidelines for Simscape Subsystem Replacement

To generate HDL code for Simscape algorithms, you generate an HDL implementation model by using the Simscape HDL Workflow Advisor. If you follow certain guidelines when modeling the Simscape algorithm, the Simscape HDL Workflow Advisor replaces the Simscape subsystem with a corresponding `HDL Subsystem` block in the HDL implementation model. The `HDL Subsystem` block contains the state-space algorithm that uses HDL-compatible Simulink blocks instead of Simscape blocks. You can generate HDL code for the `HDL Subsystem` block and deploy the code onto FPGA target devices and FPGAs on board Speedgoat FPGA I/O modules. In this case, when you select the **Generate validation logic for the implementation model** check box in the **Generate implementation model** task of the Simscape HDL Workflow Advisor, the Advisor generates a separate state-space validation model. This model compares the outputs from the `HDL Subsystem` and the original Simscape subsystem to verify that they are functionally equivalent.

If you do not follow the guidelines, the Simscape HDL Workflow Advisor might not be able to perform this replacement. In that case, the HDL implementation model contains the state-space algorithm with the original Simscape subsystem beside it. Before generating code, you modify the implementation model and rearrange the blocks such that it replaces the Simscape subsystem with the state-space algorithm. In this case, when you select the **Generate validation logic for the implementation model** check box, the Advisor places a validation logic subsystem inside the implementation model to verify functional equivalence.

In addition to these guidelines, make sure that the Simscape model is configured for compatibility with Simscape HDL Workflow Advisor. See “Modeling Physical Systems in Simscape for HDL Compatibility” on page 30-2.

Enclose Simscape Blocks Inside a Subsystem

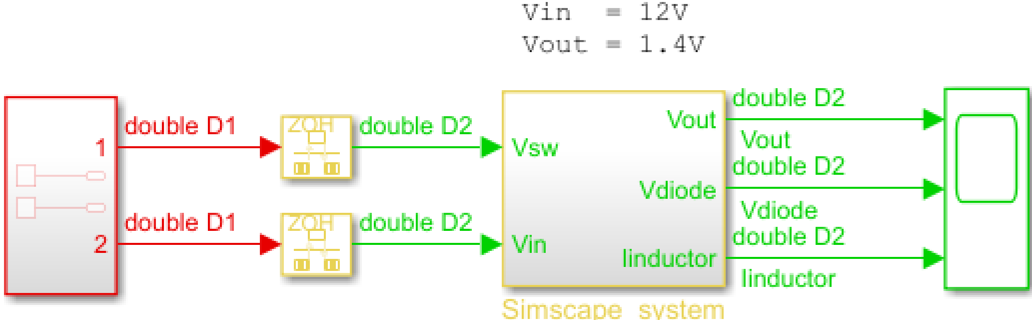
- Enclose the Simscape blocks for which you are generating an HDL implementation model inside a Subsystem block and provide the test inputs. Inside the Subsystem block, your model can have multiple hierarchies that use Simscape blocks.
- Do not use masked subsystems. The Simscape HDL Workflow Advisor cannot replace masked subsystems in the HDL implementation model. For automatic subsystem replacement, you can use masked subsystems that have cosmetic masks. Cosmetic masks are masks that only have an icon but don't have any parameters or initialization code.
- Inside the Subsystem block that contains Simscape blocks, at the input ports, add Simulink-PS Converter blocks. At the output ports of this subsystem, add PS-Simulink Converter blocks.

  - Use a meaningful name for the Simulink-PS Converter and PS-Simulink Converter blocks.

    The Simscape HDL Workflow Advisor uses the names of the Simulink-PS Converter and PS-Simulink Converter blocks for the input and output ports of the `HDL Subsystem` block. Using a meaningful name makes it easier to identify what the input and output ports in the HDL implementation model correspond to.
  - In the Block Parameters dialog box of the Simulink-PS Converter block, click **Input Handling** on the **Settings** tab, set the **Provided signals** to `Input only`, or change the **Filtering and derivatives** to `Filter input, derivatives calculated` or `Zero derivatives (piecewise constant)`.

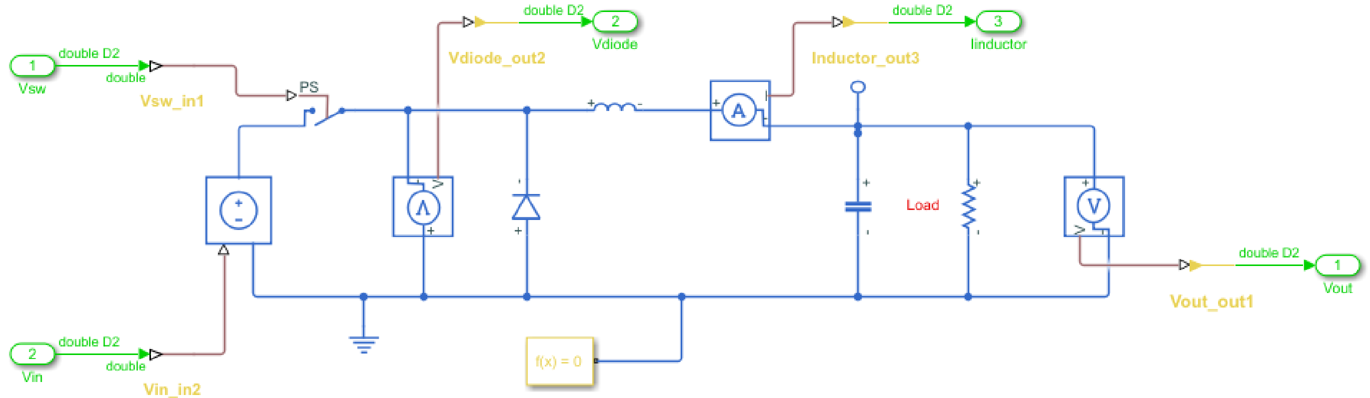
For example, open the buck converter model. The Simscape_system block contains Simscape blocks. Blocks outside this subsystem form the test environment.

```
openExample('plantdeployment/BuckConverterModelExample','supportingFile','sschdlexBuckConverterE
sim('sschdlexBuckConverterExample')
```



Inside the `Simscape_system` subsystem, the model uses Simscape blocks and physical signals. The model has Simulink-PS Converter and PS-Simulink Converter blocks at the interfaces. Provide unique names for these blocks such that they match the corresponding port names.

```
openExample('plantdeployment/BuckConverterModelExample','supportingFile','sschdlexBuckConverterE
open_system('sschdlexBuckConverterExample/Simscape_system')
```

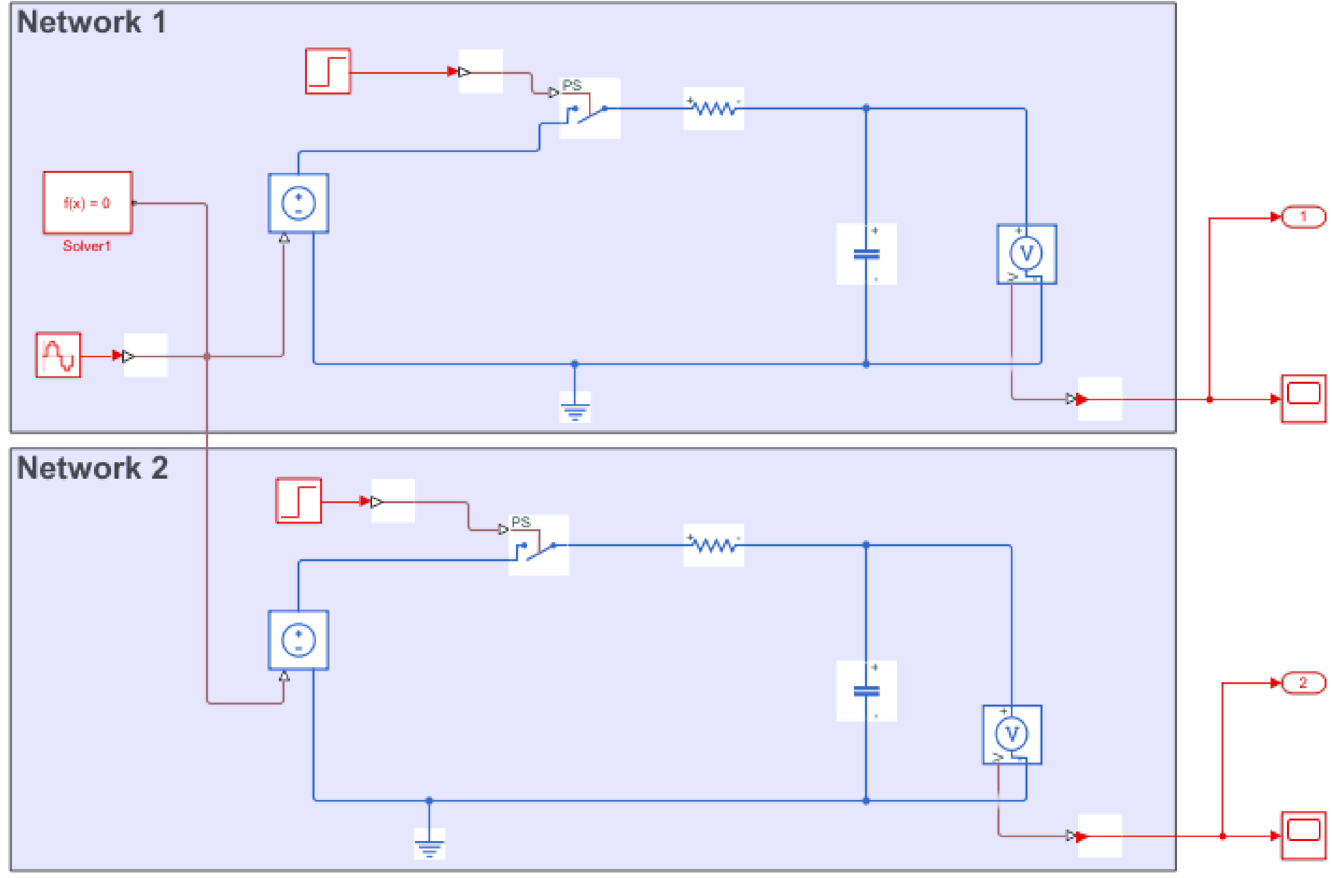


Multiple Simscape Network Considerations

If your Simscape model contains multiple networks:

- Enclose each network inside a subsystem. Add Simulink-PS Converter and PS-Simulink Converter blocks at the subsystem interface.
- Use a Solver Configuration block for each network. Use the same sample time across Solver Configuration blocks inside the different networks.

  For example, this model contains more Simscape networks than Solver Configuration blocks, the Simscape network is not replaced with the HDL subsystem.



The Simscape HDL Workflow Advisor then replaces each Simscape subsystem with the corresponding `HDL Subsystem`.

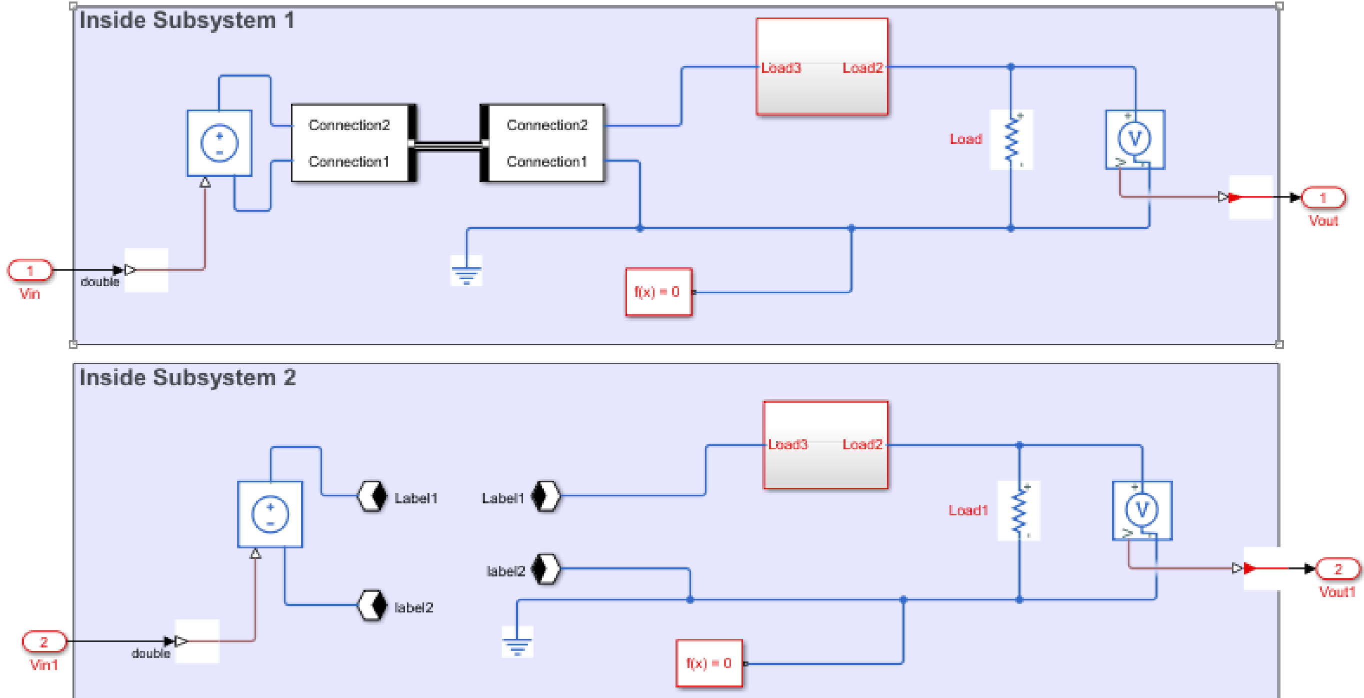
For an example that shows how to generate HDL code for a model that has multiple networks, see “Generate HDL Code for Simscape Models with Multiple Networks” on page 30-67.

Avoid Using Certain Blocks in Simscape Utilities Library

To generate an implementation model that replaces the Simscape subsystem with the state-space algorithm, in your original Simscape model, do not use these blocks from the **Simscape** > **Utilities** Library:

- Simscape Bus
- Connection Port
- Connection Label

For example, this model contains Connection Label and Simscape Bus blocks inside two different subsystems. The Simscape HDL Workflow Advisor cannot replace these subsystems with the state-space algorithm.



See Also

makehdl | sschdladvisor

More About

- "Get Started with Simscape Hardware-in-the-Loop Workflow" on page 30-2
- "Generate HDL Code for Simscape Models" on page 30-13
- "Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model" on page 30-28

Generate HDL Code for Simscape Models

This example shows how to generate HDL code for a half-wave rectifier model that uses Simscape™ blocks. Use the Simscape HDL Workflow Advisor to generate an HDL implementation model. You can then generate HDL code for the implementation model. See “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 30-2.

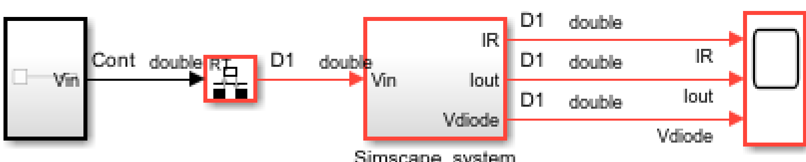
The Halfwave Rectifier Model

To open the half-wave rectifier model, at the MATLAB® command prompt, enter:

```
open_system('sschdlexHalfWaveRectifierModel')
```

Save this model locally as `HalfWaveRectifier_HDL` to run the workflow.

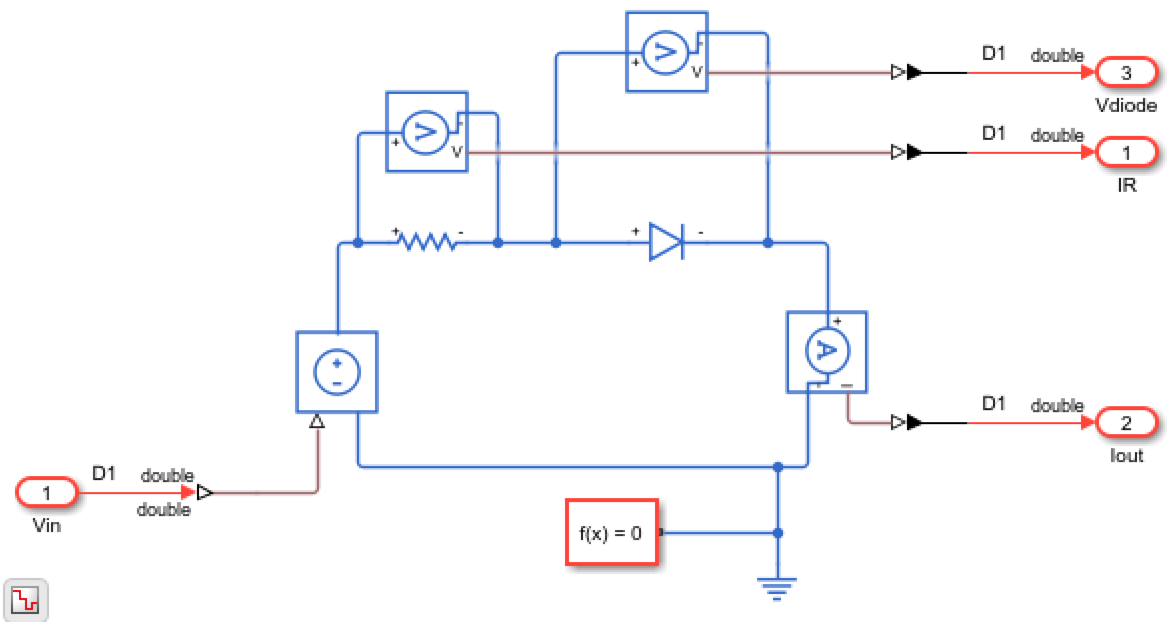
```
open_system('HalfWaveRectifier_HDL')
set_param('HalfWaveRectifier_HDL', 'SimulationCommand', 'Update')
```



Copyright 2020 The MathWorks, Inc.

At the top level of the model, a `Simscape_system` block models the half-wave rectifier algorithm. The model accepts a Sine Wave input, uses a Rate Transition block to discretize the continuous time input, and has a Scope block that calculates the output. To see the half-wave rectifier algorithm, double-click the `Simscape_system` subsystem.

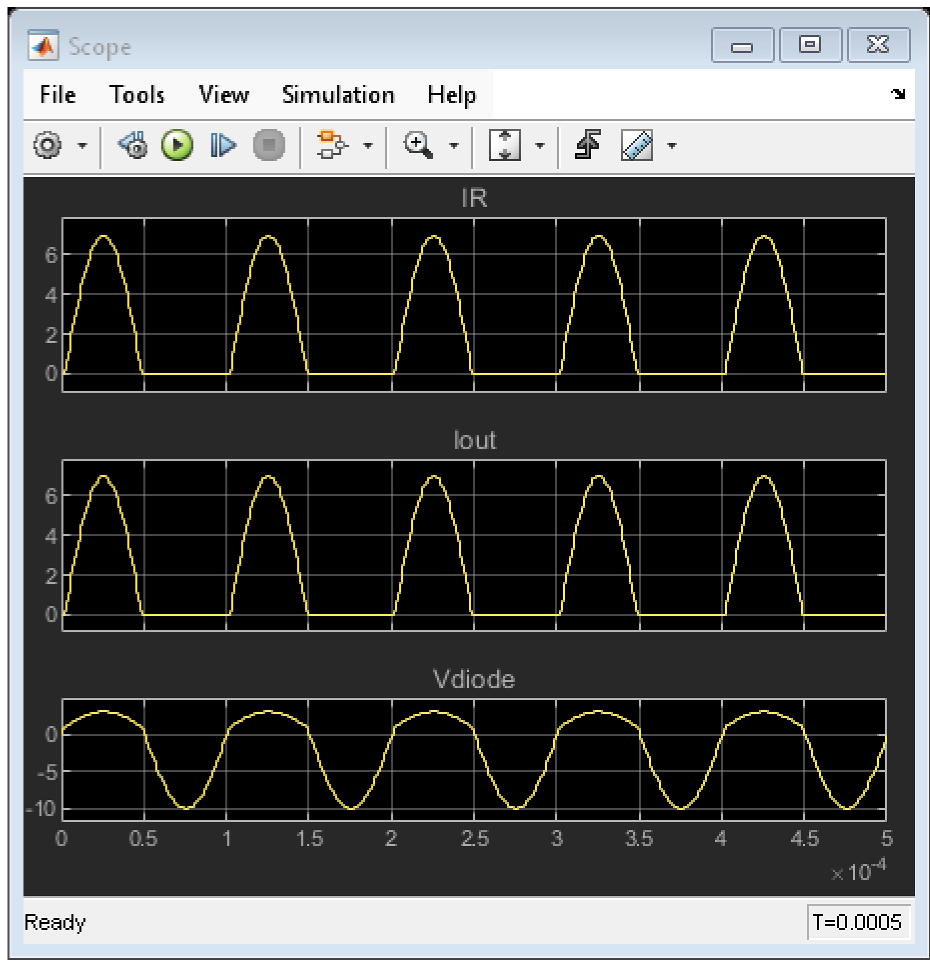
```
open_system('HalfWaveRectifier_HDL/Simscape_system')
```



The half-wave rectifier consists of a resistor, which is a linear block, and a diode, which is a switched linear block. The Simscape model is preconfigured for HDL compatibility. At the input and output port interfaces, the model has Simulink-PS Converter and PS-Simulink Converter blocks. The solver settings are configured for compatibility with Simscape HDL Workflow Advisor. If you open the Block Parameters dialog box for the Solver Configuration block, **Use local solver** is selected and `Backward Euler` is specified as the **Solver type**. See "Get Started with Simscape Hardware-in-the-Loop Workflow" on page 30-2.

To see the functionality, simulate the model and then open the Scope block.

```
sim('HalfWaveRectifier_HDL')
open_system('HalfWaveRectifier_HDL/Scope')
```



Run Simscape HDL Workflow Advisor

To generate an HDL implementation model from which you generate code, use the Simscape HDL Workflow Advisor. To open the Advisor, run this command:

```
sschdladvisor('HalfWaveRectifier_HDL')
```

This command updates the model advisor cache and opens the Simscape HDL Workflow Advisor. To learn more about the Simscape HDL Workflow Advisor and the various tasks, right-click that folder or task, and select **What's This?**. See also “Simscape HDL Workflow Advisor Tasks” on page 31-2.

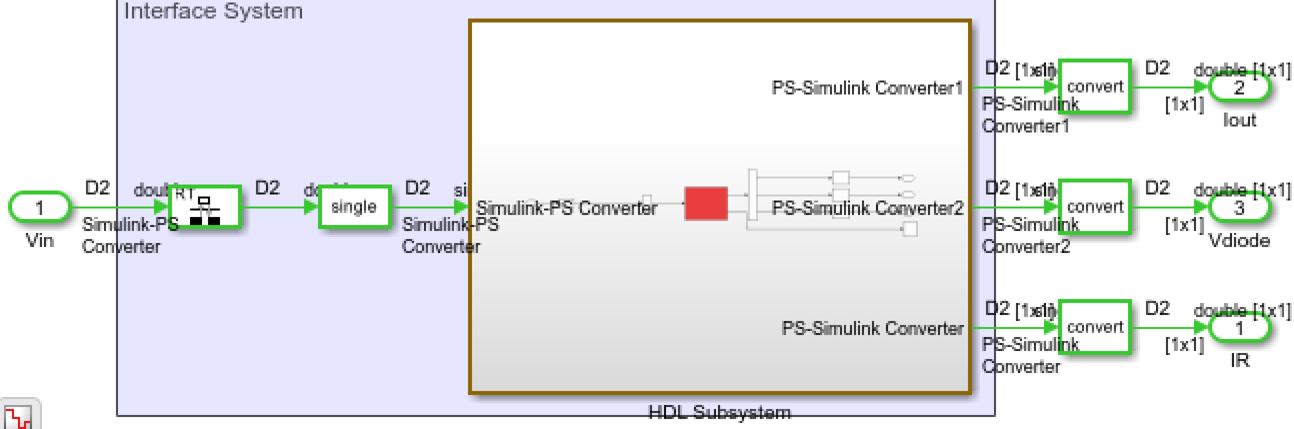
To run the workflow and compare functionality of the HDL implementation model with the original Simscape algorithm, select the **Generate implementation model** step, and then select the **Generate validation logic for the implementation model** check box. Use a **Validation logic tolerance** of `0.001`. Right-click the **Generate implementation model** step and select **Run to Selected Task**.

The Advisor generates an HDL implementation model and a state-space validation model. The implementation model has the same name as the original Simscape model and uses the prefix `gmStateSpaceHDL_`. The state-space validation model has the same name as the implementation model and uses the postfix `_vnl`.

Open and Examine HDL Implementation Model

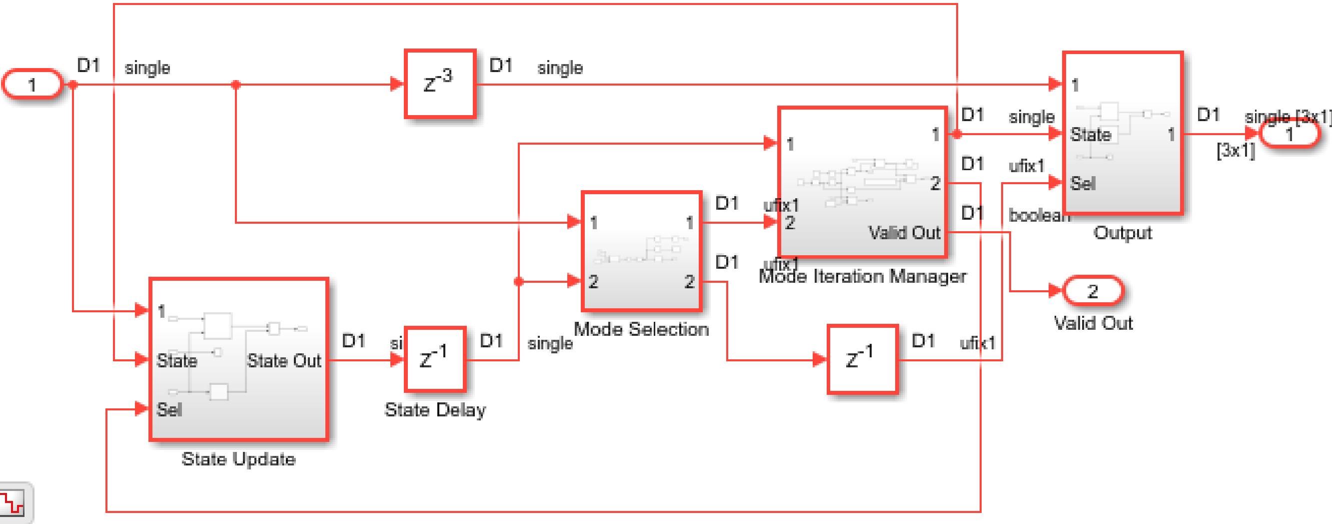
In the **Generate implementation model task**, click the link to open the implementation model. The model contains a `Simscape_system` subsystem that contains a `HDL Subsystem` block. The `HDL Subsystem` models the state-space representation that you generated from the Simscape model.

```
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL')
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL/Simscape_system')
set_param('gmStateSpaceHDL_HalfWaveRectifier_HDL','SimulationCommand','Update')
```



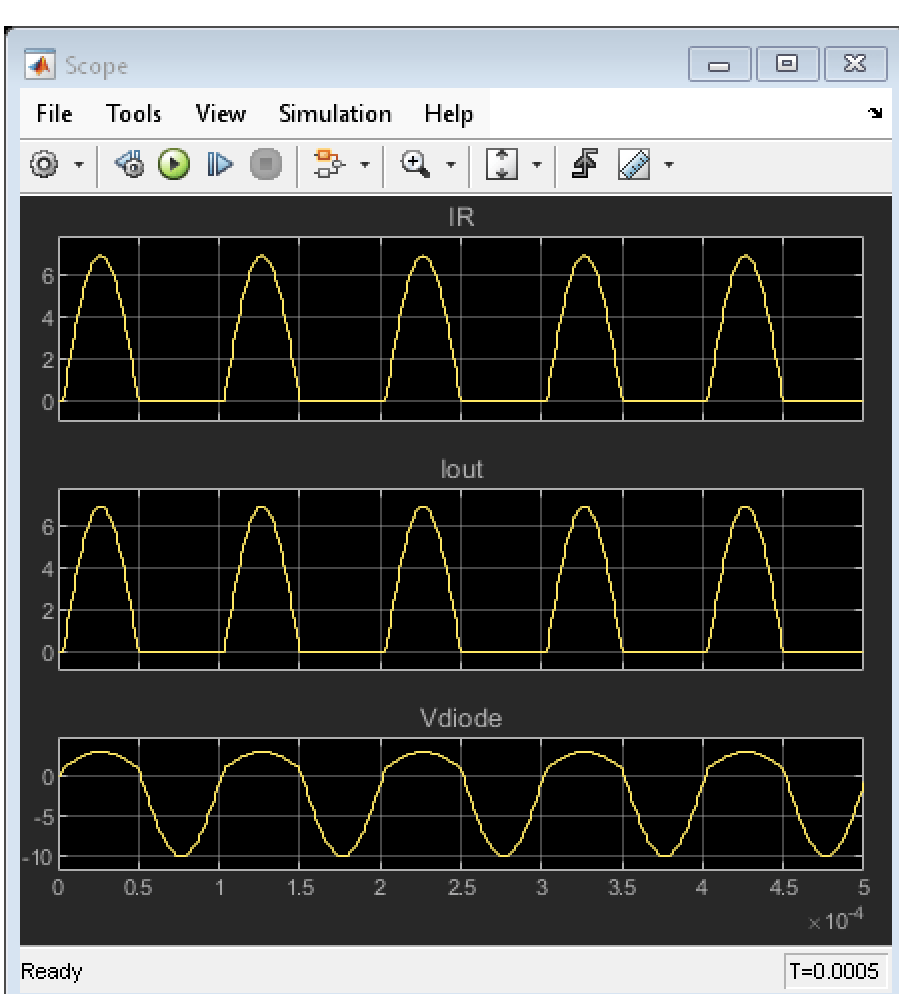
The ports of this subsystem use the same name as the Simulink-PS Converter and PS-Simulink Converter blocks in your original Simscape model. If you navigate inside this subsystem, you see several delays, adders, and Matrix Multiply blocks that model the state-space equations.

```
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL/Simscape_system/HDL Subsystem/HDL Algorithm')
```



To simulate the HDL Implementation model, enter these commands. Open the Scope block to view results.

```
sim('gmStateSpaceHDL_HalfWaveRectifier_HDL')
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL/Scope')
```

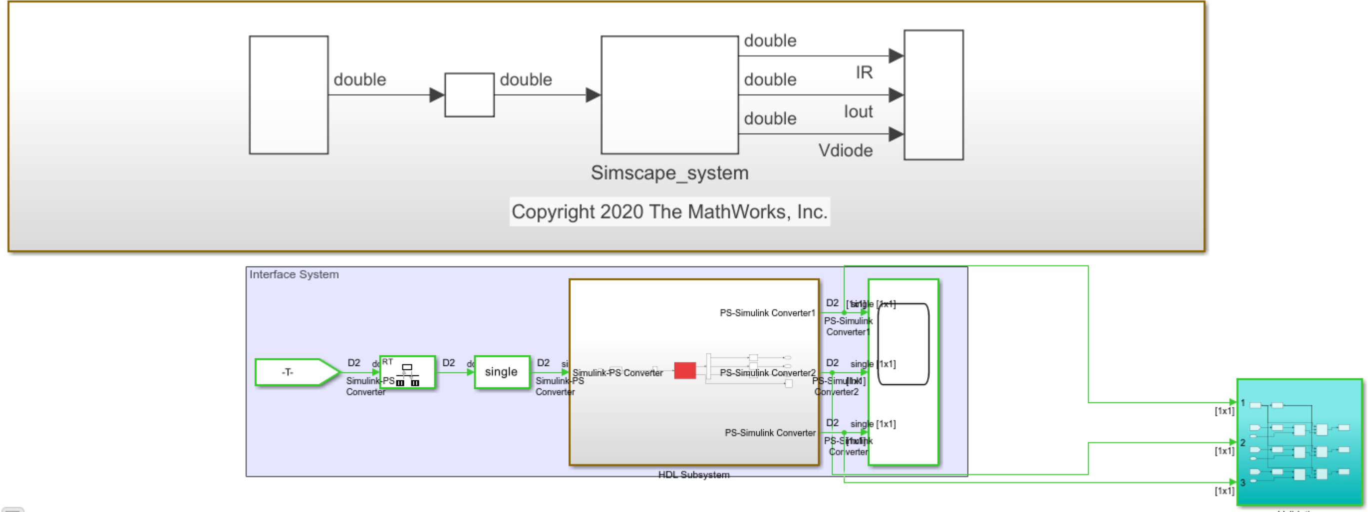


HDL code is generated for the `HDL Subsystem` block inside this model.

Validate HDL Algorithm

To compare functionality of the HDL implementation model with the original Simscape algorithm, open and simulate the state-space validation model.

```
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL_vnl')
sim('gmStateSpaceHDL_HalfWaveRectifier_HDL_vnl')
```



The output of this model matches the original Simscape model. The simulation does not generate assertions, which indicates that the outputs match. For a more systemic verification, see “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89.

Generate HDL Code and Validation Model

The HDL model and subsystem parameter settings are saved using this command:

```
hdlsaveparams('gmStateSpaceHDL_HalfWaveRectifier_HDL');
```

```
%% Set Model 'gmStateSpaceHDL_HalfWaveRectifier_HDL' HDL parameters
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'FPToleranceValue', 1.000000e-03);
fpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT' ...
, 'LatencyStrategy', 'Min' ...
);
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'FloatingPointTargetConfiguration', fpconf
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'HDLSubsystem', 'gmStateSpaceHDL_HalfWaveRe
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'MaskParameterAsGeneric', 'on');
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'Oversampling', 13);
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'UseFloatingPoint', 'on');

hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL/Simscape_system/HDL Subsystem/HDL Algorithm/
```

The model uses single data types and generates HDL code in `Native Floating Point` mode. Floating-point operators can introduce delays. Because the design contains feedback loops, to allocate sufficient delays for the operators inside the feedback loops, the model uses clock-rate pipelining.

For more information, see:

- “Clock-Rate Pipelining” on page 21-148
- Oversampling factor
- “Generate a Global Oversampling Clock” on page 20-9

Before you generate HDL code, enable generation of the validation model. The validation model compares the output of the generated model after code generation and the original model. To learn more, see “Generated Model and Validation Model” on page 21-10.

Run these commands to save validation model generation settings on your Simulink model:

```
HDLmodelname = 'gmStateSpaceHDL_HalfWaveRectifier_HDL';
hdlset_param(HDLmodelname, 'TargetDirectory', 'C:/Temp/hdlsrc');
hdlset_param(HDLmodelname, 'GenerateValidationModel', 'on');
```

To generate HDL code, run this command:

```
makehdl('gmStateSpaceHDL_HalfWaveRectifier_HDL/Simscape_system/HDL Subsystem');
```

The generated HDL code and validation model are saved in `C:/Temp/hdlsrc` directory. The generated code is saved as `HDL_Subsystem.vhd`. To open the validation model, click the link to `gm_gmStateSpaceHDL_HalfWaveRectifier_HDL_vnl.slx` in the code generation logs in the Command Window.

Open the `Compare` block at the output of `HDL Subsystem_vnl` subsystem of the validation model. To see the simulation results after HDL code generation, double-click the `Double click to turn 'on/off' all scopes` block. For each of the scopes (IR, Iout, and Vdiode), the first graph represents the output of the generated model, and the middle graph represents the output of the implementation model. The bottom graph calculates the difference between outputs of both models. As the outputs match, the error is zero.

See Also

Functions
`checkhdl` | `makehdl`

More About

- “Get Started with Simscape Electrical” (Simscape Electrical)
- “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 30-2
- “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89

Generate Optimized HDL Implementation Model from Simscape

This example shows how you can generate an optimized HDL implementation model for a Simscape™ vienna rectifier model by using optimizations such as resource sharing and RAM mapping.

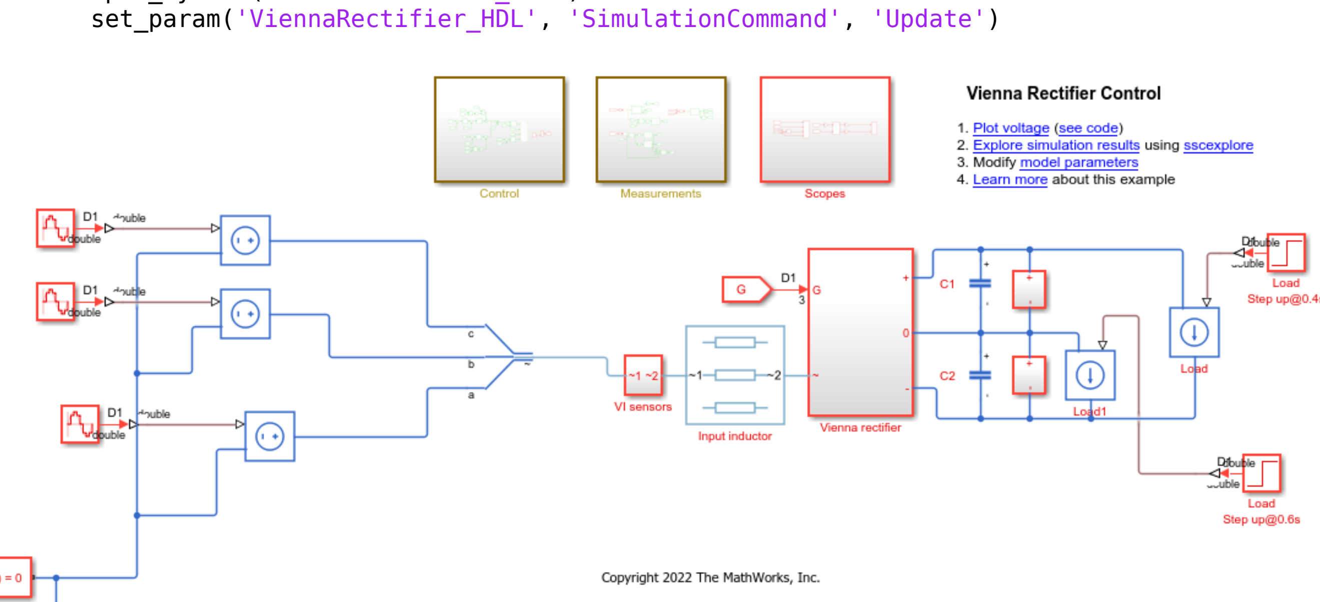
Why Optimize the HDL Implementation Model

For Simscape models that have many switching elements, the state-space representation contains a large number of configurations. The Simscape HDL Workflow Advisor simulates the Simscape model to calculate the number of relevant configurations. Certain Simscape models can have a large number of configurations that are relevant. The generated HDL implementation model for such a large design can consume a significantly large number of resources. Synthesizing the generated code can cause the design to occupy a large amount of resources on the FPGA device. In some cases, the design might not fit on the target FPGA device. To save resoures and make the design fit on the FPGA, the Simscape HDL Workflow Advisor uses HDL Coder™ optimizations such as clock-rate pipelining, resource sharing, and RAM mapping.

Vienna Rectifier Model

To open the model, at the MATLAB® command prompt, enter:

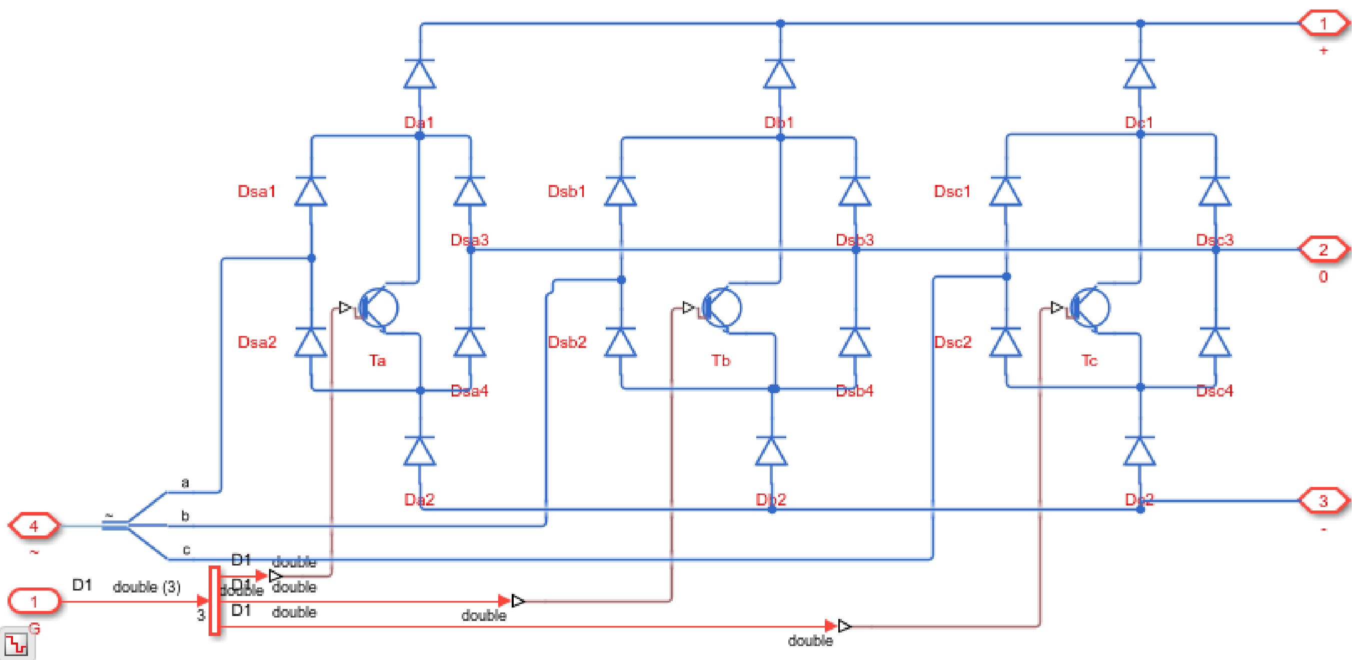
```
open_system('ViennaRectifier_HDL')
set_param('ViennaRectifier_HDL', 'SimulationCommand', 'Update')
```



The `Control` subsystem implements a closed-loop control strategy for the `Vienna rectifier` subsystem by using space-vector modulation. At simulation time `0.1s`, the vienna rectifier is engaged. At times `0.4s` and `0.6s`, the load steps up on the DC side.

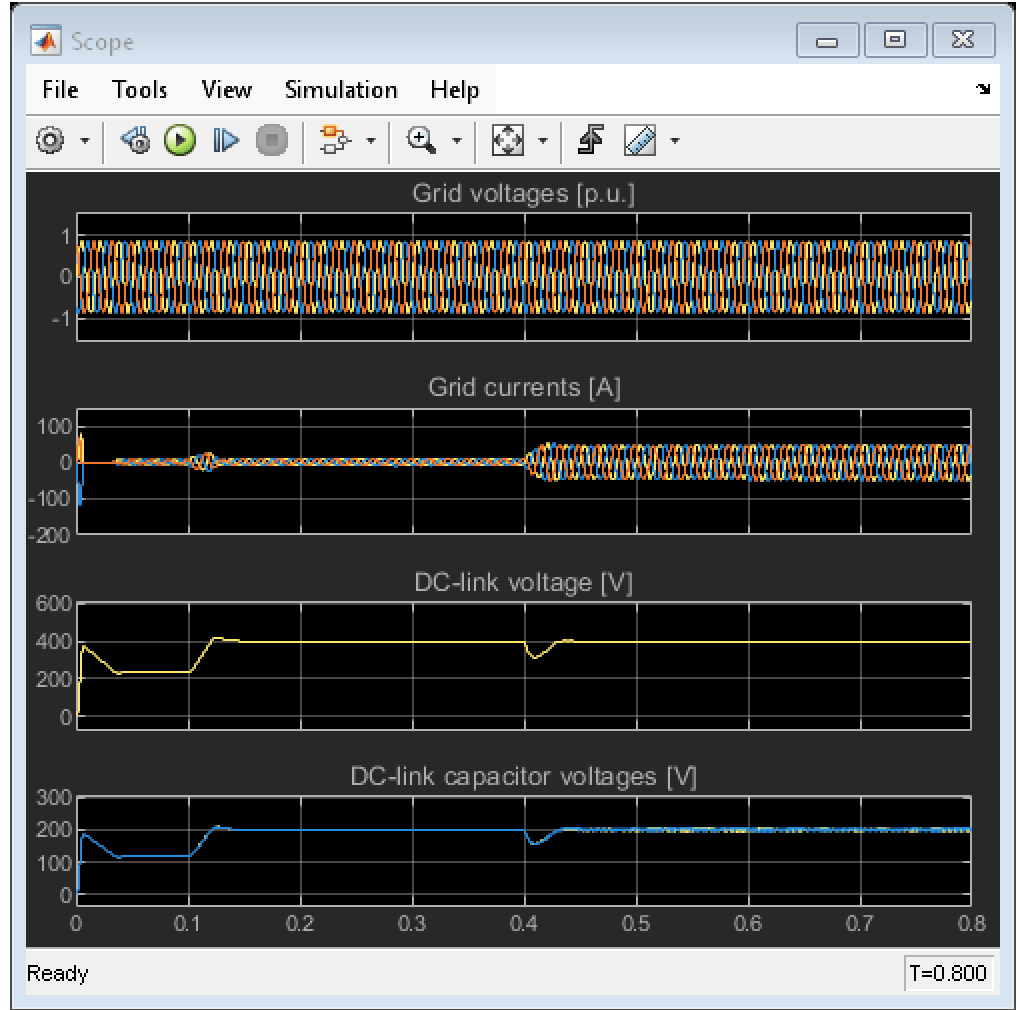
The `Vienna rectifier` subsystem consists of three-phase legs. Each leg has one power switch and six power diodes. See “Vienna Rectifier Control” (Simscape Electrical).

```
open_system('ViennaRectifier_HDL/Vienna rectifier')
```



Simulate the model. View the simulation results by double-clicking the Scope blocks inside the `Scopes` subsystem.

```
sim('ViennaRectifier_HDL')
open_system('ViennaRectifier_HDL/Scopes/Scope')
```



Generate HDL Implementation Model and Validate HDL Algorithm

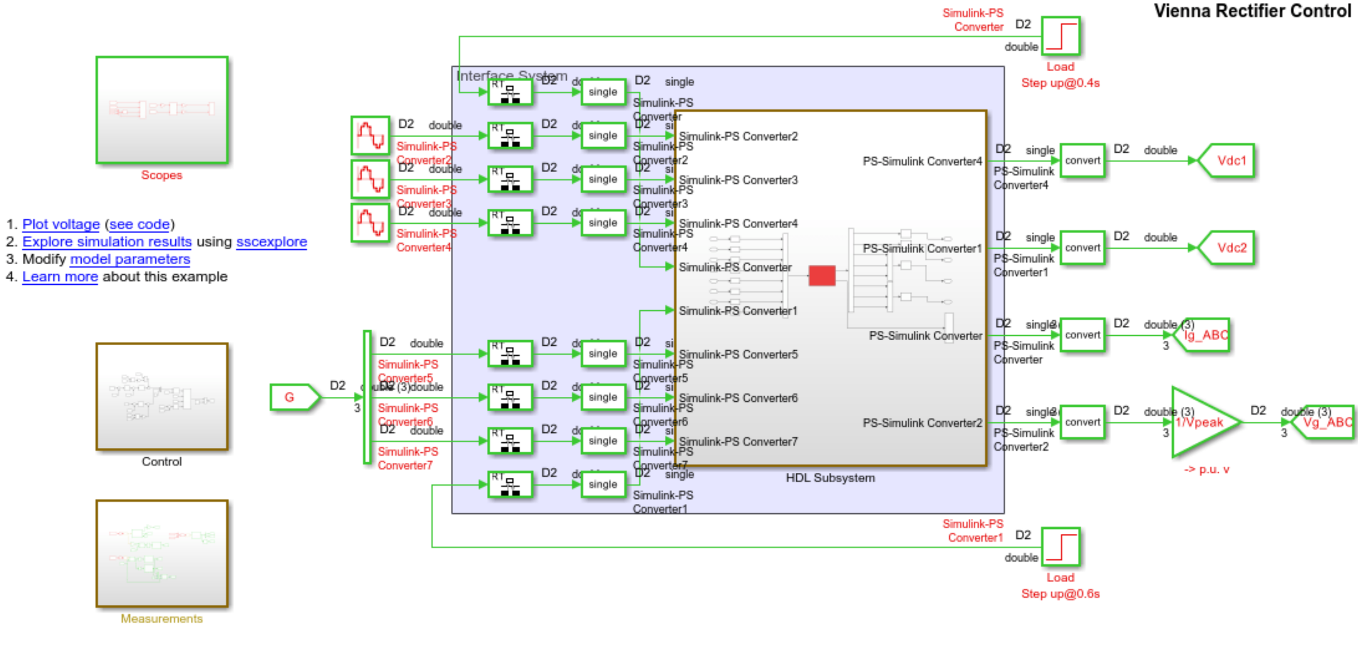
To generate an HDL implementation model, use the Simscape HDL Workflow Advisor. You can generate HDL code for the implementation model. To open the Advisor, run this command:

```
sschdladvisor('ViennaRectifier_HDL')
```

To generate the HDL implementation model, in the **Implementation model generation** subfolder, right-click the **Generate implementation model** task and select **Run to Selected Task**. After the task passes, you see a link to the HDL implementation model. To see the number of configurations, select the **Extract Equations** task under **State-space conversion** subfolder. When you click the task, you see that simulating the model reaches `397` modes. Such a large number of modes can increase resource consumption of the design on the FPGA.

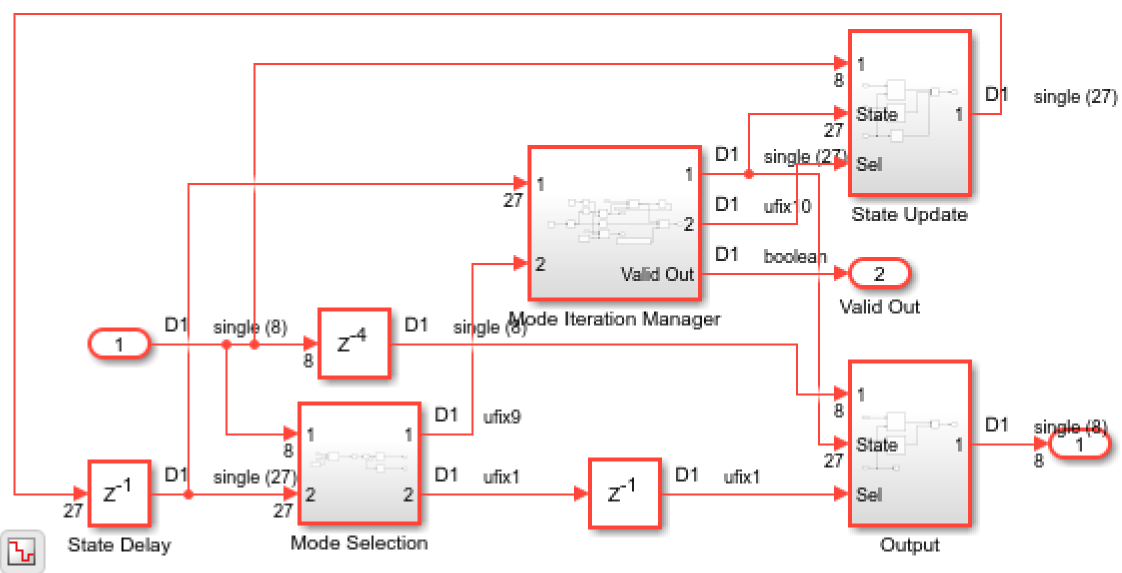
To open the HDL implementation model, enter these commands:

```
open_system('gmStateSpaceHDL_ViennaRectifier_HDL')
set_param('gmStateSpaceHDL_ViennaRectifier_HDL', 'SimulationCommand', 'Update')
```



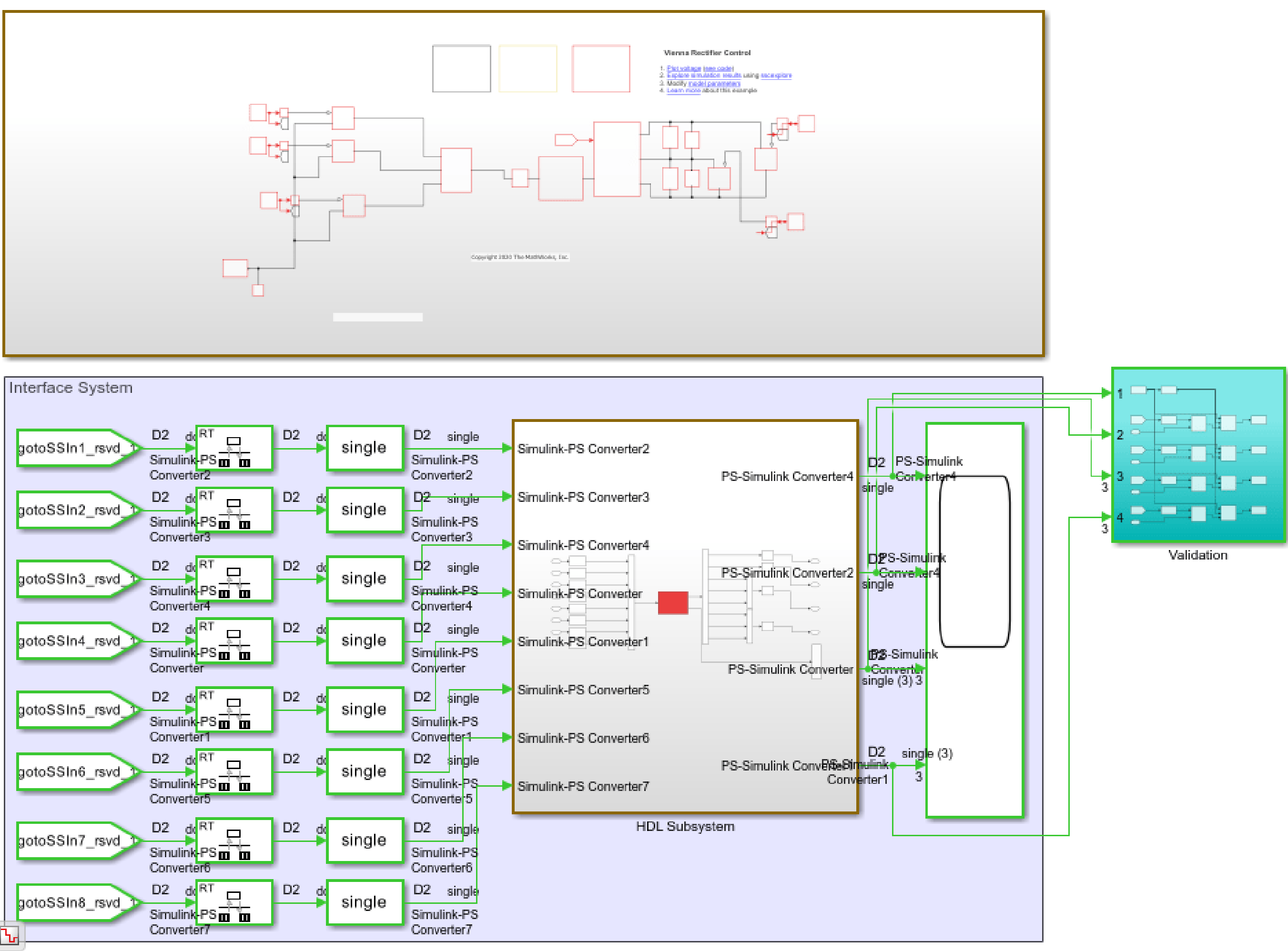
The ports of this subsystem use the same name as the Simulink-PS Converter and PS-Simulink Converter blocks in your original Simscape model. If you navigate inside this subsystem, you see several delays, adders, and Matrix Multiply blocks that model the state-space equations.

```
open_system('gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem/HDL Algorithm')
```



To validate the HDL algorithm, in the **Generate implementation model** task, select the **Generate validation logic for the implementation model** check box in the **Verification Settings**, set the **Validation logic tolerance** to `0.001`, and rerun this task. The task generates a state-space validation model that compares the implementation model and the original Simscape model.

```
open_system('gmStateSpaceHDL_ViennaRectifier_HDL_vnl')
set_param('gmStateSpaceHDL_ViennaRectifier_HDL_vnl', 'SimulationCommand', 'Update')
```



Simulating the model does not display assertions, which indicates that the HDL algorithm matches the original model.

```
sim('gmStateSpaceHDL_ViennaRectifier_HDL_vnl')
```

Map State-Space Parameters in Implementation Model to RAM

The HDL implementation model uses `single` data types and contains large Delay blocks that are inside a feedback loop in the `HDL Algorithm` subsystem. To accommodate the large delays and make the design run at a faster clock rate on the target FPGA, the model uses clock-rate pipelining in conjunction with a large value of **Oversampling factor**.

```
hdlsaveparams('gmStateSpaceHDL_ViennaRectifier_HDL')
```

For more information, see:

- “Clock-Rate Pipelining” on page 21-148
- Oversampling factor

- “Generate a Global Oversampling Clock” on page 20-9

In the **Generate implementation model** task, the **Map state space parameters to RAMs** setting uses the default value of `Auto`. This setting maps large state-space parameters in the HDL implementation model to RAMs when the number of modes exceed a threshold value of `200`. As the vienna rectifier model uses a large number of modes, the state-space parameters are mapped to RAMs. By mapping to RAMs, you save lookup table resources on the FPGA. To enable the RAM mapping, the “UseRAM” on page 19-25 parameter is enabled on the masked subsystem blocks that perform the state update and compute the output.

To map the parameters to RAMs irrespective of the threshold, set **Map state space parameters to RAMs** to `on`.

To see the effect of RAM mapping on the vienna rectifier model:

1. Verify the **UseRAM** parameter setting by running the `hdlget_param` function on the `Multiply Input` and `Multiply State` blocks.

```
Multiplysubsys1 = 'gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem/HDL Algorithm/State Update'
Multiplysubsys2 = 'gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem/HDL Algorithm/Output';
UseRAM1 = hdlget_param([Multiplysubsys1 '/Multiply Input'], 'UseRAM')
UseRAM2 = hdlget_param([Multiplysubsys1 '/Multiply State'], 'UseRAM')
```

```
UseRAM1 =

    'on'


UseRAM2 =

    'on'
```

2. Enable generation of the resource utilization report.

```
hdlset_param('gmStateSpaceHDL_ViennaRectifier_HDL', 'ResourceReport', 'on')
```

3. Generate HDL code for the implementation model.

```
makehdl('gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem');
```

When you generate code, HDL Coder opens a Code Generation report. The **High-level Resource Report** shows `136` RAMs utilized.

Multipliers	294
Adders/Subtractors	4894
Registers	28386
Total 1-Bit Registers	274486
RAMs	136
Multiplexers	44200
I/O Bits	516
Static Shift operators	0
Dynamic Shift operators	632

Resource Sharing of State Update and Output Computation Blocks

Before you generate HDL code for the HDL Subsystem, you can optimize the algorithm by using the resource sharing optimization in HDL Coder. Resource sharing is an area optimization that identifies multiple functionally equivalent resources and replaces them with a single, equivalent resource. The data is time-multiplexed over the shared resource to perform the same operations. See “Resource Sharing” on page 21-45.

In the HDL implementation model, you can share the masked subsystem blocks that perform state updates and compute the output.

To share these subsystems for the vienna rectifier and generate HDL code:

1. Specify a **SharingFactor** of 2 on the Multiply Input and Multiply State subsystems.

```
hdlset_param([Multiplysubsys1 '/Multiply Input'],'SharingFactor', 2)
hdlset_param([Multiplysubsys1 '/Multiply State'],'SharingFactor', 2)
hdlset_param([Multiplysubsys2 '/Multiply Input'],'SharingFactor', 2)
hdlset_param([Multiplysubsys2 '/Multiply State'],'SharingFactor', 2)
```

2. Enable generation of the optimization report

```
hdlset_param('gmStateSpaceHDL_ViennaRectifier_HDL', 'OptimizationReport', 'on')
```

3. Generate HDL code for the HDL Subsystem block in the implementation model.

```
makehdl('gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem');
```

When you generate code, HDL Coder opens a Code Generation report. To see the status of the resource sharing optimization, click the **Streaming and Sharing** section of the report. This sharing group shows the dot products that the optimization shared. When you click the **High-level Resource Report**, you see that the comsumption of adders, multipliers, and registers have decreased.

Subsystem: Multiply Input

SharingFactor: 2

Highlight shared resources and diagnostics

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1			2	dot_product_5	
2			2	dot_product_5	
3			2	dot_product_5	
4			2	dot_product_5	
5			2	dot_product_5	
6			2	dot_product_5	
7			2	dot_product_5	
8			2	dot_product_5	
9			2	dot_product_5	
10			2	dot_product_5	
11			2	dot_product_5	
12			2	dot_product_5	
13			2	dot_product_5	

See Also

Functions
`checkhdl` | `makehdl`

More About

- “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 30-2
- “Speed and Area Optimizations in HDL Coder” on page 21-3
- “Generate HDL Code for Simscape Models” on page 30-13

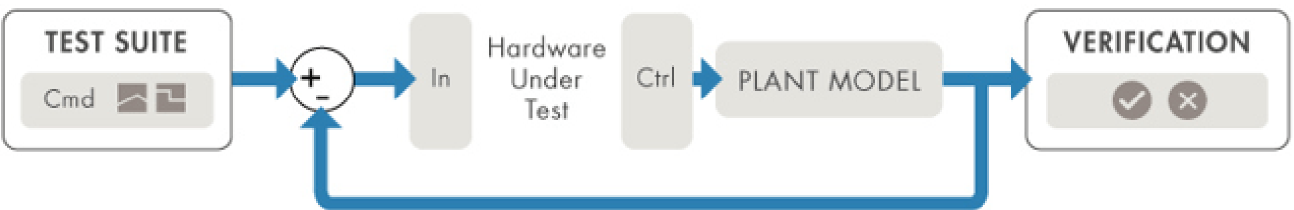
Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model

This example shows how to generate a Simulink® Real-Time Interface subsystem for a Simscape™ two-level converter plant model. You can then deploy the interface model on the Speedgoat® FPGA I/O module. This example uses the Speedgoat IO334-325k module.

Real-Time Simulation

Simulating the plant model on the FPGA provides:

- **Real-time Simulation**: Hardware-in-the-loop provides real-time simulation of your Simscape plant model on the target hardware.



- **Hardware Acceleration**: Accelerated simulation of complex physical systems on hardware while reconfigurable FPGAs provide rapid prototyping.

To use the workflow:

1. Develop the Simscape model and convert it into an implementation model by using the Simscape HDL Workflow Advisor.
2. Generate HDL code and deploy the code to the Speedgoat I/O module by using the HDL Workflow Advisor.

Setup and Configuration

Before deploying your algorithm on the Speedgoat I/O module:

1. Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat')
```

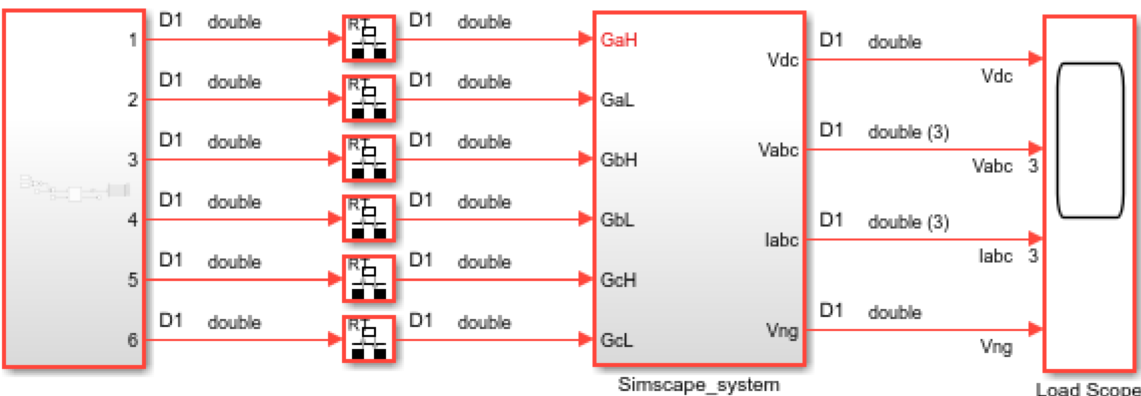
2. For real-time simulation, set up the development environment and target computer settings. See “Get Started with Simulink Real-Time” (Simulink Real-Time).

3. Install the Speedgoat I/O Blockset and the Speedgoat HDL Coder Integration packages. See Install Speedgoat HDL Coder Integration Packages.

Two-Level Converter Ideal Model

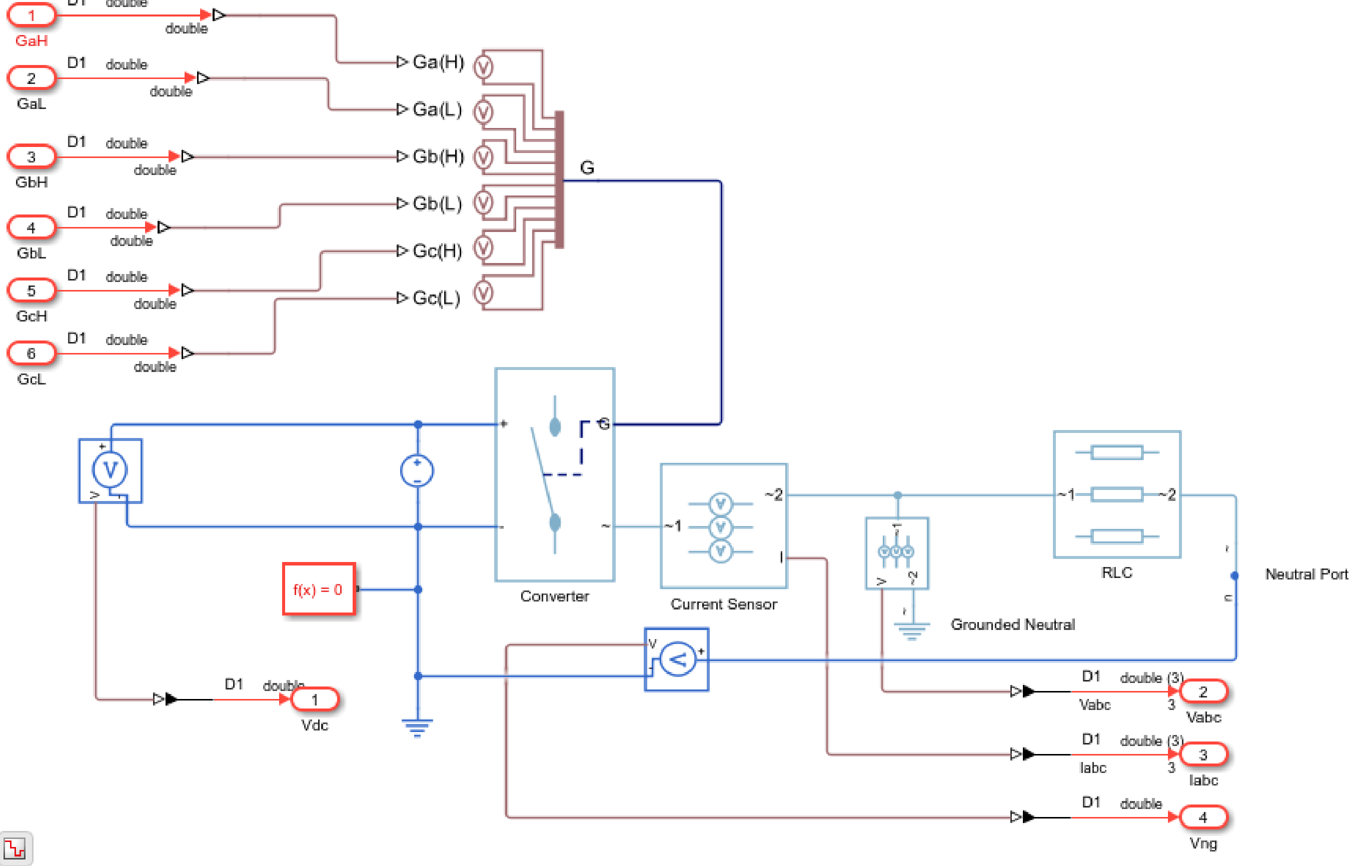
To open this model, at the MATLAB® command prompt, enter:

```
open_system('TwoLevelConverter_HDL')
set_param('TwoLevelConverter_HDL','SimulationCommand','update')
```



Copyright 2020 The MathWorks, Inc.

```
open_system('TwoLevelConverter_HDL/Simscape_system')
```

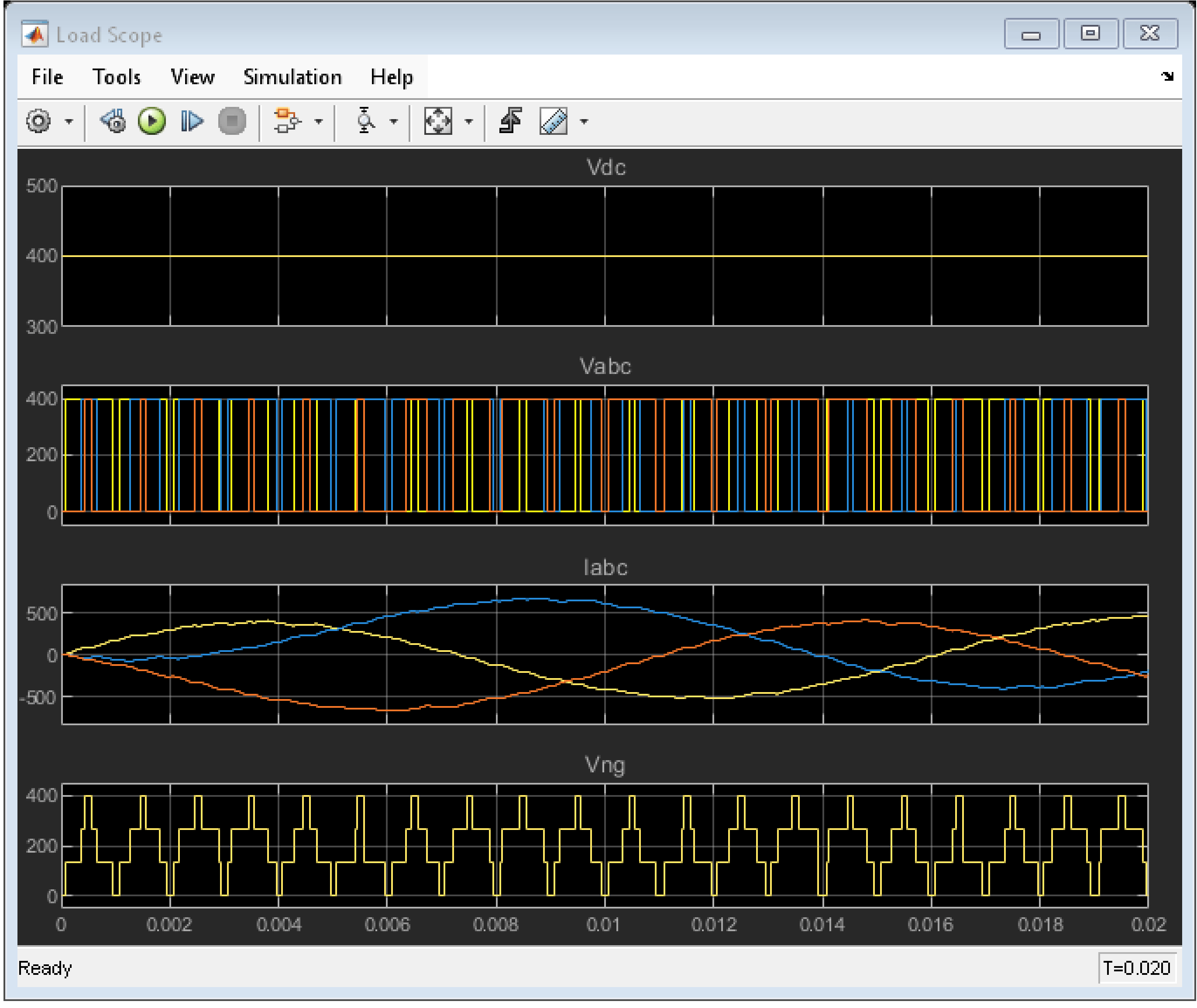


The Simscape subsystem receives six-switch controlling pulses as input. The Simscape subsystem acts as a generator that uses a two-level, carrier-based PWM method to:

1. Sample a reference wave.
2. Compare the sample to a triangular carrier wave.
3. Generate a switch-on pulse if a sample is higher than the carrier signal or a switch-off pulse if a sample is lower than the carrier wave.

Simulate the model.

```
sim('TwoLevelConverter_HDL')
open_system('TwoLevelConverter_HDL/Load Scope')
```



Generate HDL Implementation Model

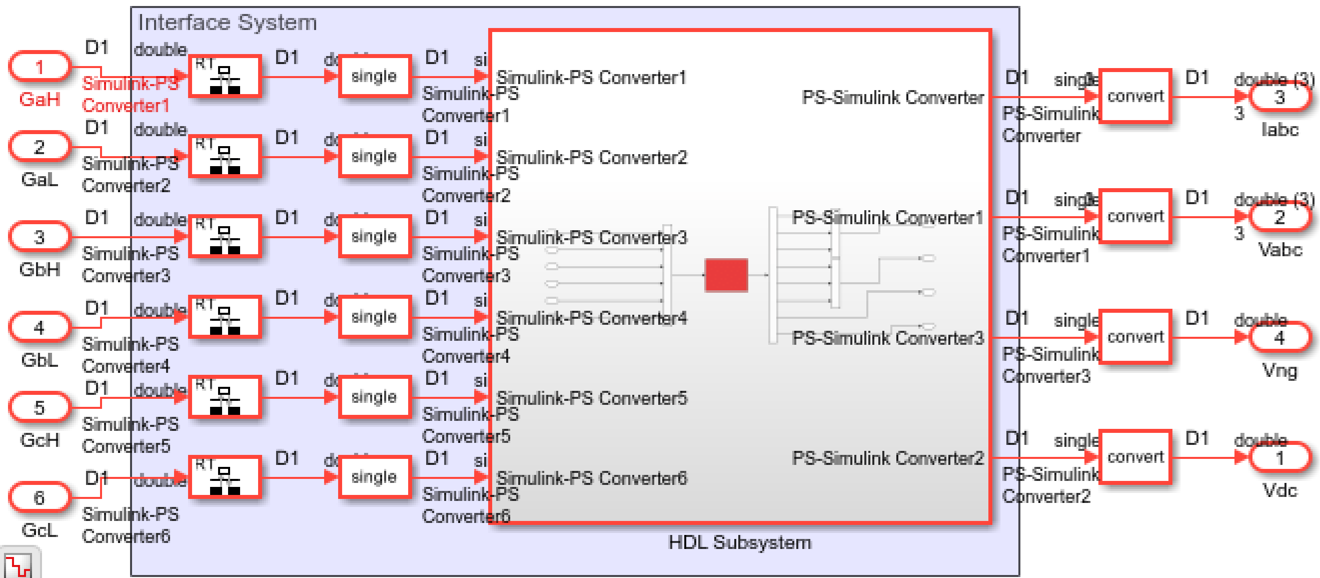
To generate an implementation model, use the Simscape HDL Workflow Advisor. Run the `sschdladvisor` function for your model:

```
sschdladvisor('TwoLevelConverter_HDL')
```

To generate the implementation model, in the Simscape HDL Workflow Advisor, keep the default settings for the tasks, and then run the tasks. You see a link to the model in the **Generate implementation model** task. This model has the same name as the original model prefixed with `gmStateSpaceHDL`.

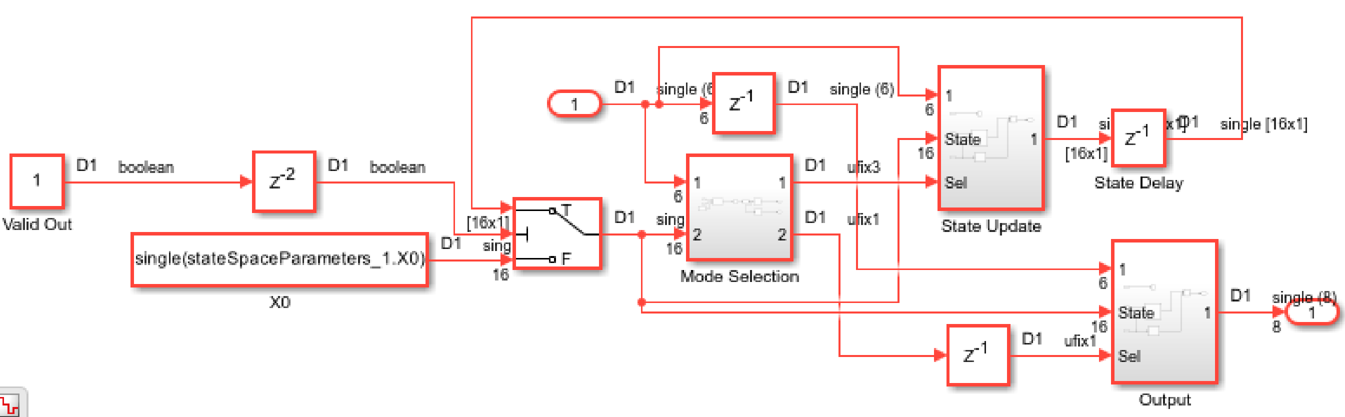
To open the implementation model, enter:

```
load_system('gmStateSpaceHDL_TwoLevelConverter_HDL')
open_system('gmStateSpaceHDL_TwoLevelConverter_HDL/Simscape_system')
set_param('gmStateSpaceHDL_TwoLevelConverter_HDL','SimulationCommand','update')
```



The implementation model replaces the Simscape subsystem with the HDL algorithm that performs the state-space computations. When you navigate inside this subsystem, you see several delays, adders, and Matrix Multiply blocks that model the state-space equations. From and Goto blocks inside this subsystem provide the same input as that of the original model to the `HDL Subsystem`.

```
open_system('gmStateSpaceHDL_TwoLevelConverter_HDL/Simscape_system/HDL Subsystem/HDL Algorithm')
```



HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.
- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis and timing analysis.
- Deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

To open the HDL Workflow Advisor for a subsystem inside the model, use the `hdladvisor` function.

```
load_system('sschdlexTwoLevelConverterIgbtExample')
hdladvisor('sschdlexTwoLevelConverterIgbtExample/Simscape_system')
```

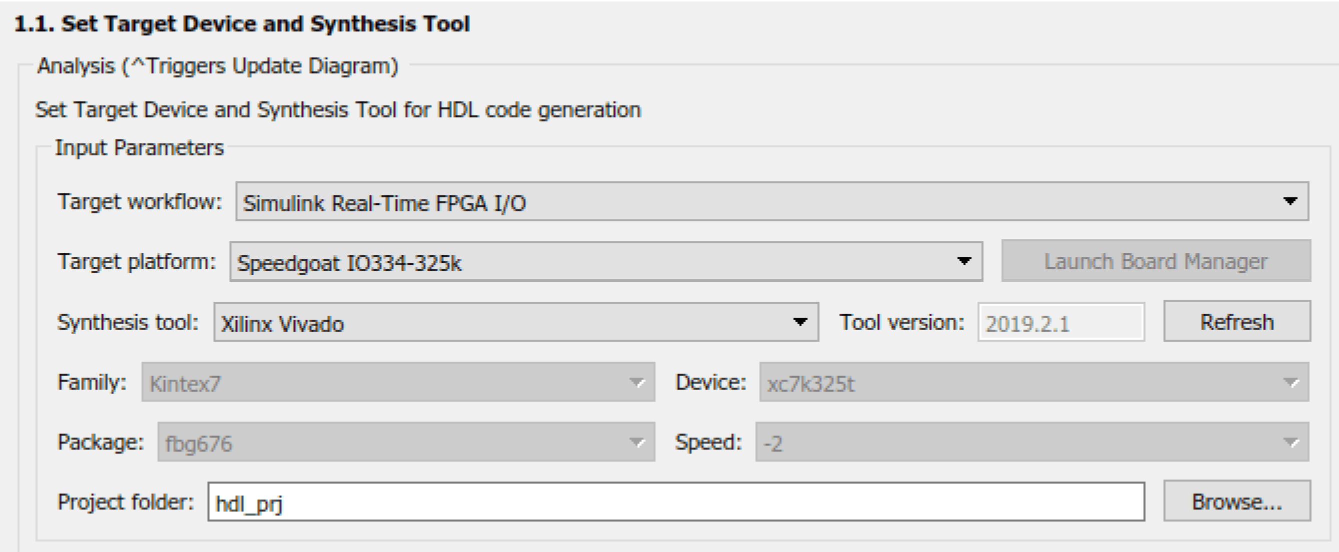
The left pane contains folders that represent a group of related tasks. Expanding the folders and selecting a task displays information about that task in the right pane. The right pane contains simple controls for running the task to advanced parameters and option settings that control code and test bench generation. To learn more about each task, right-click that task, and select **What's This?**. See "Getting Started with the HDL Workflow Advisor" on page 29-5.

Deploy Two Level Ideal Converter Model to Speedgoat IO334-325K Module

1. Open the HDL Workflow Advisor for the implementation model.

```
hdladvisor('gmStateSpaceHDL_TwoLevelConverter_HDL/Simscape_system/HDL Subsystem')
```

2. In **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Simulink Real-Time FPGA I/O` and **Target platform** as `Speedgoat IO334-325K`



3. Run the **Set Target Reference Design** task, select a value of `x4` for the parameter `PCIe lanes`, and select **Run This Task**.

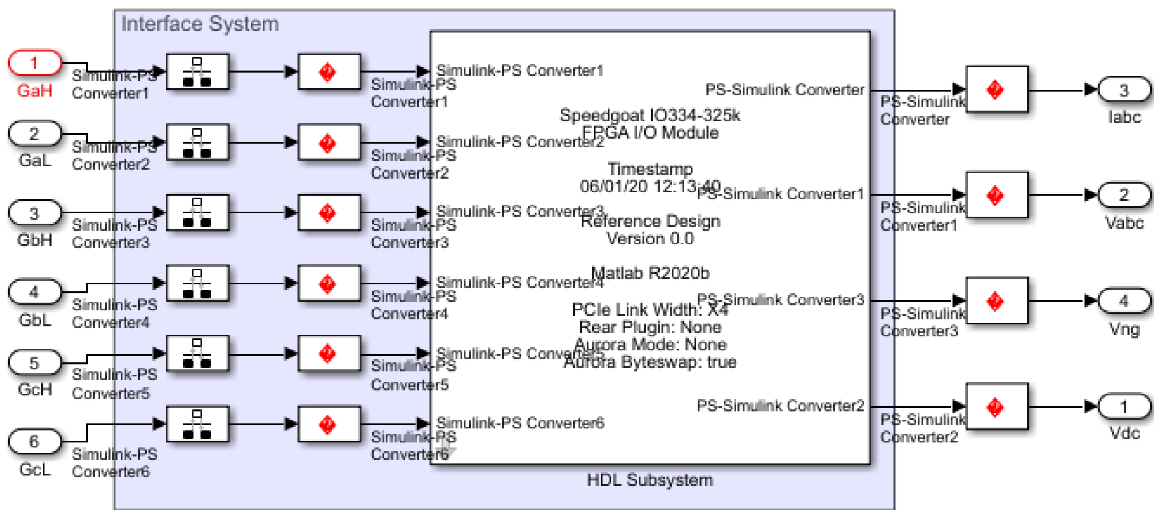
4. In **Set Target Interface** task, map the input and output `single` data type ports to `PCIe Interface` and select **Run This Task**.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
Simulink-PS Converte...	Inport	single	PCIe Interface	x"100"	Options...
Simulink-PS Converte...	Inport	single	PCIe Interface	x"104"	Options...
Simulink-PS Converte...	Inport	single	PCIe Interface	x"108"	Options...
Simulink-PS Converte...	Inport	single	PCIe Interface	x"10C"	Options...
Simulink-PS Converte...	Inport	single	PCIe Interface	x"110"	Options...
Simulink-PS Converte...	Inport	single	PCIe Interface	x"114"	Options...
PS-Simulink Converter	Outport	single (3)	PCIe Interface	x"120"	
PS-Simulink Converte...	Outport	single (3)	PCIe Interface	x"140"	
PS-Simulink Converte...	Outport	single	PCIe Interface	x"118"	
PS-Simulink Converte...	Outport	single	PCIe Interface	x"11C"	

5. Right-click **Generate RTL Code and IP Core** task and select **Run to Selected Task**. As the model uses vector data types, the **Generate RTL Code and IP Core** fails because the `ScalarizePorts` property must be set to `dutlevel`. Click the link to change this setting and rerun the task.

6. Run the workflow to the **Generate Simulink Real-Time interface** task. In **Create Project** task, you can open the Vivado project and see the implemented design. After the **Generate Simulink Real-Time interface** task passes, click the link to open the Simulink Real-Time Interface Model.



Export HDL Workflow to Script

For rapid prototyping, you can export the HDL Workflow Advisor settings to a script. The script is a MATLAB® file that you can run from the command line. You can then modify and run the script, or import the settings into the HDL Workflow Advisor User Interface.

To export an HDL Workflow script, after you run the tasks in the Advisor, select **File > Export to Script**. For this example, when you export to script, this file shows the settings you saved.

```
edit('hdlworkflow_slrt.m')
```

To import an HDL Workflow script, in the HDL Workflow Advisor, select **File > Import from Script**. Select the script file and click **Open**. The HDL Workflow Advisor updates the tasks with the imported script settings.

For an example that shows how to run the real-time application by deploying the FPGA bitstream, see “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82.

Deploy Simscape Buck Converter Model to Speedgoat IO Module Using HDL Workflow Script

This example shows how to deploy a Simscape™ buck converter model to a Speedgoat IO334 Simulink®-programmable I/O module and then run the model in real-time at a sample step size as small as 1 microsecond. The example uses a DCDC converter topology to show how to prepare your power electronic converter model for hardware in the loop (HIL) simulation on a Speedgoat real-time target machine.

To use this workflow:

1. Convert your model into an HDL-compatible implementation model by using the Simscape HDL Workflow Advisor
2. Generate HDL code and FPGA bitstream for the IO334 module by using the HDL Workflow Advisor.
3. Deploy the real-time model to the Speedgoat real-time target machine by using Simulink Real-Time.

The model runs at a sample time of 1us till HDL code generation and then runs at 50us on the CPU in real time. To generate HDL code and FPGA bitstream, the example shows how to run the HDL workflow script from the command line. For an example that shows how you can use the Workflow Advisor User Interface to run this workflow, see “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82.

Setup and Configuration

Before deploying your algorithm on the Speedgoat IO module:

1. Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat')
```

2. For real-time simulation, set up the development environment and target computer settings. See “Get Started with Simulink Real-Time” (Simulink Real-Time).

3. Install the Speedgoat Library and the Speedgoat HDL Coder Integration packages. See Install Speedgoat HDL Coder Integration Packages.

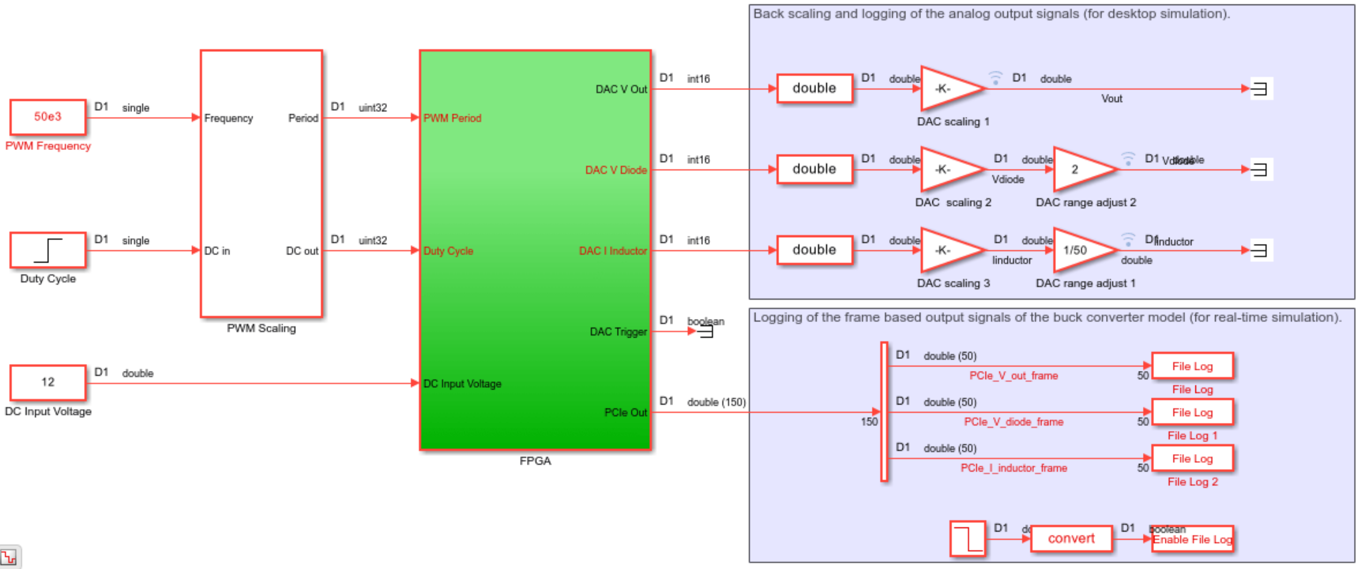
Buck Converter Model

To see the buck converter model, run this command:

```
open_system('sschdlexBuckConverterModel')
```

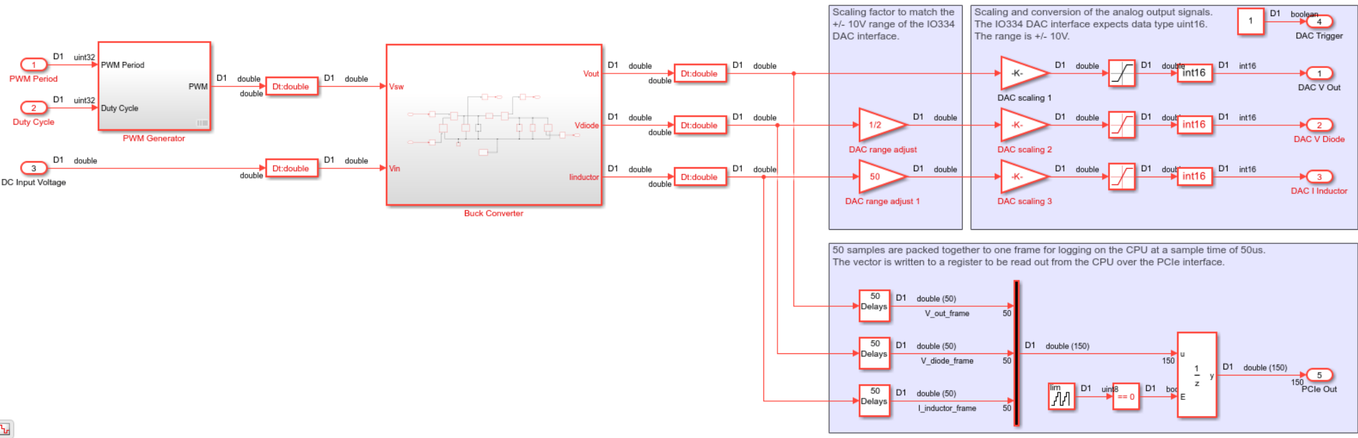
This model is modified for real-time deployment and saved as `sschdlex_IO334_BuckConverter`. The model has been partitioned into parts that run on the FPGA and parts that run on CPU. Parts inside the green `FPGA` subsystem run on the FPGA. Parts outside this subsystem run on the CPU in real time.

```
open_system('sschdlex_IO334_BuckConverter')
```



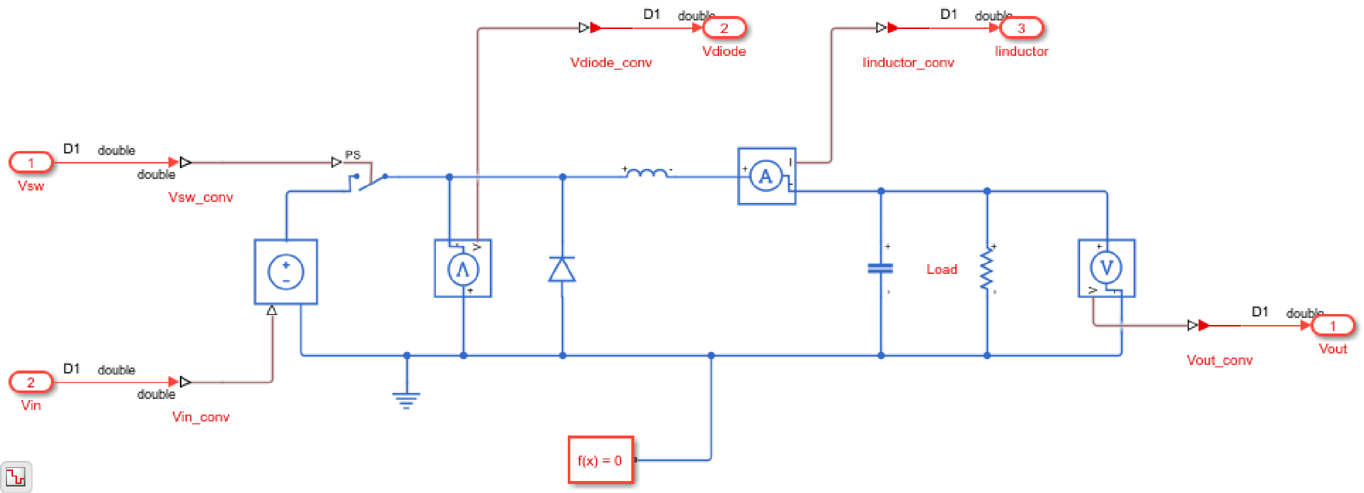
You generate VHDL code for blocks that are inside the green FPGA subsystem that contains the PWM generator and buck converter. The code is then deployed to the FPGA on board the IO334 module. The outputs of the subsystem are mapped to DAC interfaces. The output signals from the Buck Converter subsystem are scaled within a `10V` range and converted to use `uint16` data types. `50` samples are packed together to one frame to log the output signals on the CPU.

```
open_system('sschdlex_IO334_BuckConverter/FPGA')
```



To see the buck converter model, double-click the Buck Converter subsystem. The buck converter is a power converter model that steps down the input voltage at the output. The voltage at the output is stepped down by the duty cycle, D. The output voltage, `Vout`, is calculated as `Vin/D`

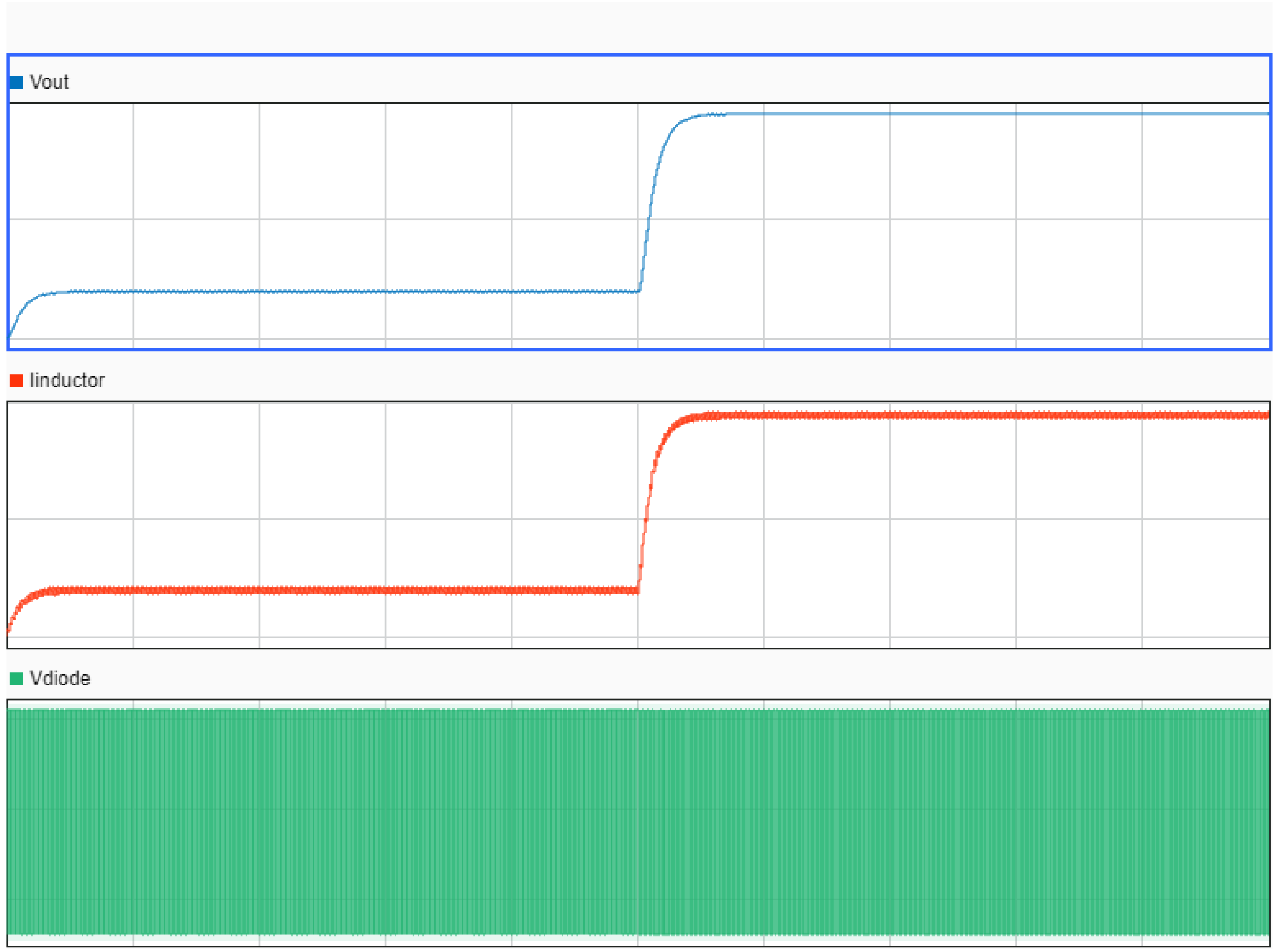
```
open_system('sschdlex_IO334_BuckConverter/FPGA/Buck Converter')
```



Run Desktop Simulation of Simscape model

The Simulation input is a duty cycle step wave from `0.2` to `0.8`. The input signals that include the DC input voltage, PWM frequency, and duty cycle are generated on the top level of the model. The sample time for the Simscape model is set to `1us`. Signal logging is enabled on the top level of the model.

```
sim('sschdlex_I0334_BuckConverter')
```



Generate HDL Implementation Model

For HDL code generation compatibility, you run the Simscape HDL Workflow Advisor to generate an HDL implementation model.

The Simscape solver is set to run for two iterations at each sample step. The Simscape HDL Workflow Advisor uses the solver settings in the next step for deterministic real-time behavior.

```
set_param('sschdlex_IO334_BuckConverter/FPGA/Buck Converter/Solver Configuration','DoFixedCost',
```

To open the Advisor, run the `sschdladvisor` function for your model:

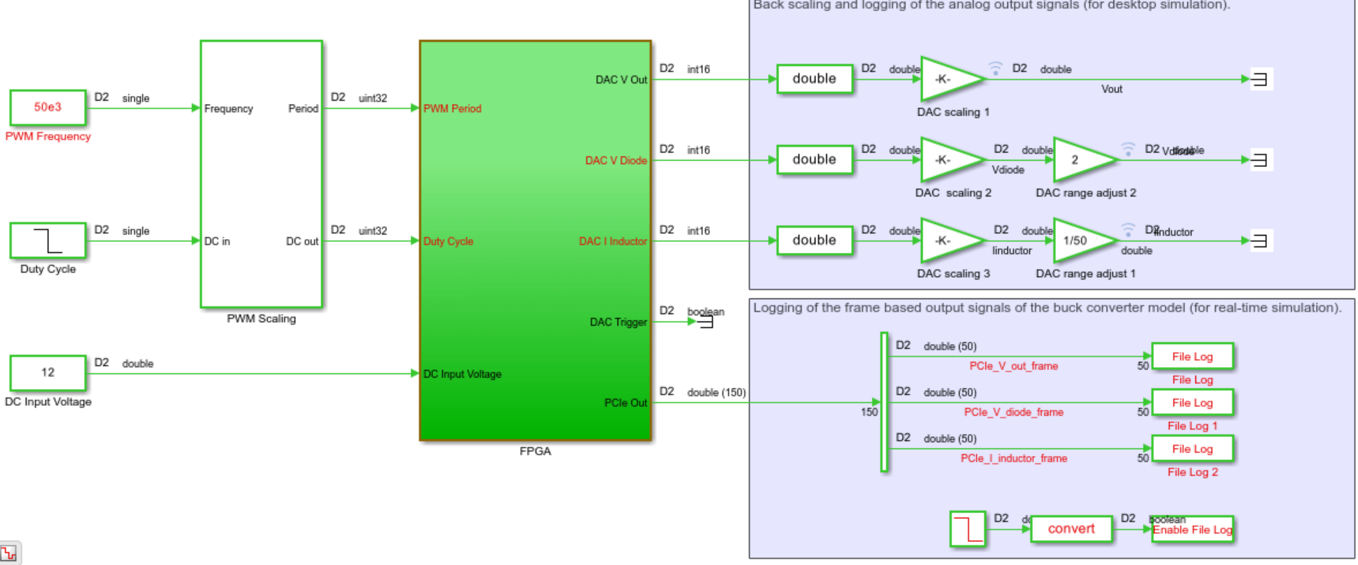
```
sschdladvisor('sschdlex_IO334_BuckConverter')
```

To generate the implementation model, in the Simscape HDL Workflow Advisor, keep the default settings for the tasks, and then run the tasks. Run the tasks in the Advisor by clicking the **Run all** button. You see a link to the model in the **Generate implementation model** task. This model has the same name as your original model with the prefix `gmStateSpaceHDL_`.

Prepare Implementation Model for HDL Code Generation

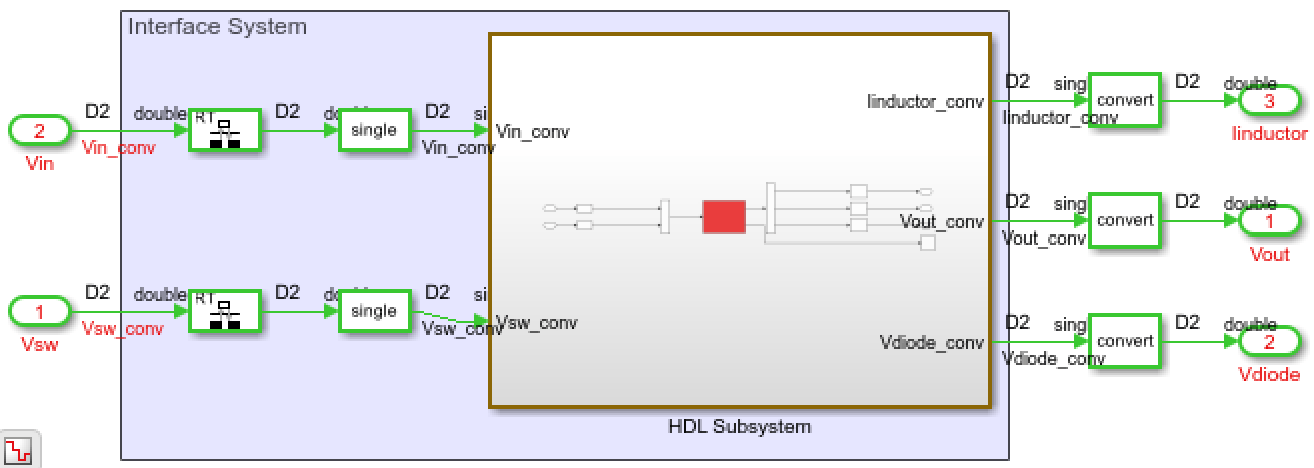
To open the implementation model, click the link in the **Generate implementation model** task.

```
open_system('gmStateSpaceHDL_sschdlex_I0334_BuckConverte');
```



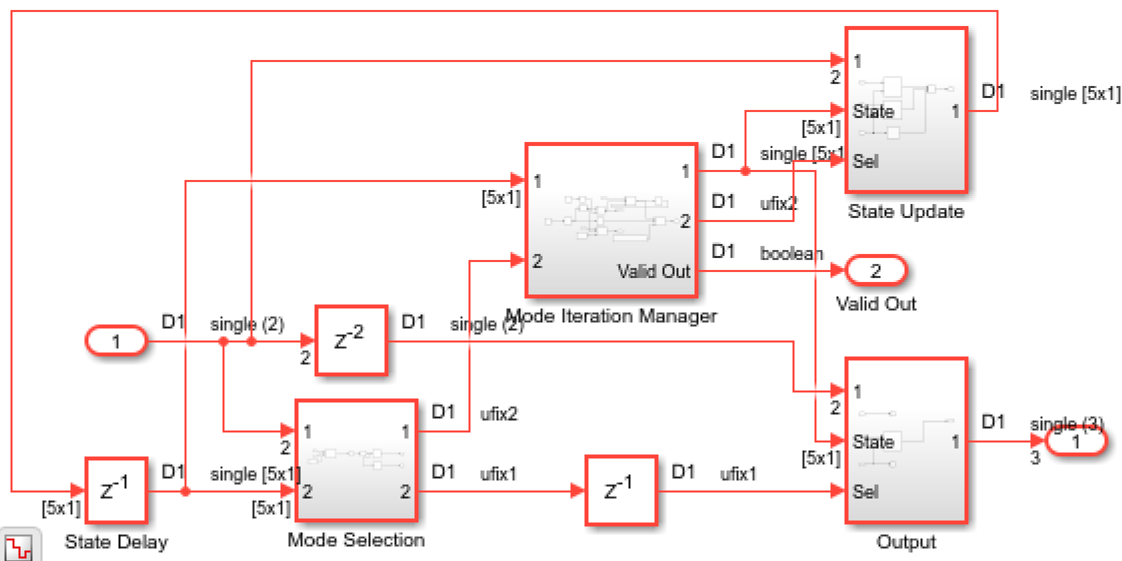
The model contains a switched linear Simulink replacement of the original buck converter model. You see that the Simscape model was replaced.

```
open_system('gmStateSpaceHDL_sschdlex_I0334_BuckConverte/FPGA/Buck Converter');
```



The implementation model replaces the Simscape subsystem with the HDL-compatible algorithm that performs the state-space computations. When you navigate inside this subsystem, you see several delays, adders, and Matrix Multiply blocks that model the state-space equations. From and Goto blocks inside this subsystem provide the same input as that of the original model to the `HDL Subsystem`.

```
open_system('gmStateSpaceHDL_sschdlex_I0334_BuckConverte/FPGA/Buck Converter/HDL Subsystem/HDL A
```



The data type of the buck converter output signals is set to single precision floating point for HDL code generation.

```
set_param('gmStateSpaceHDL_sschdlex_I0334_BuckConverte/FPGA/Signal Specification','OutDataTypeSt
```

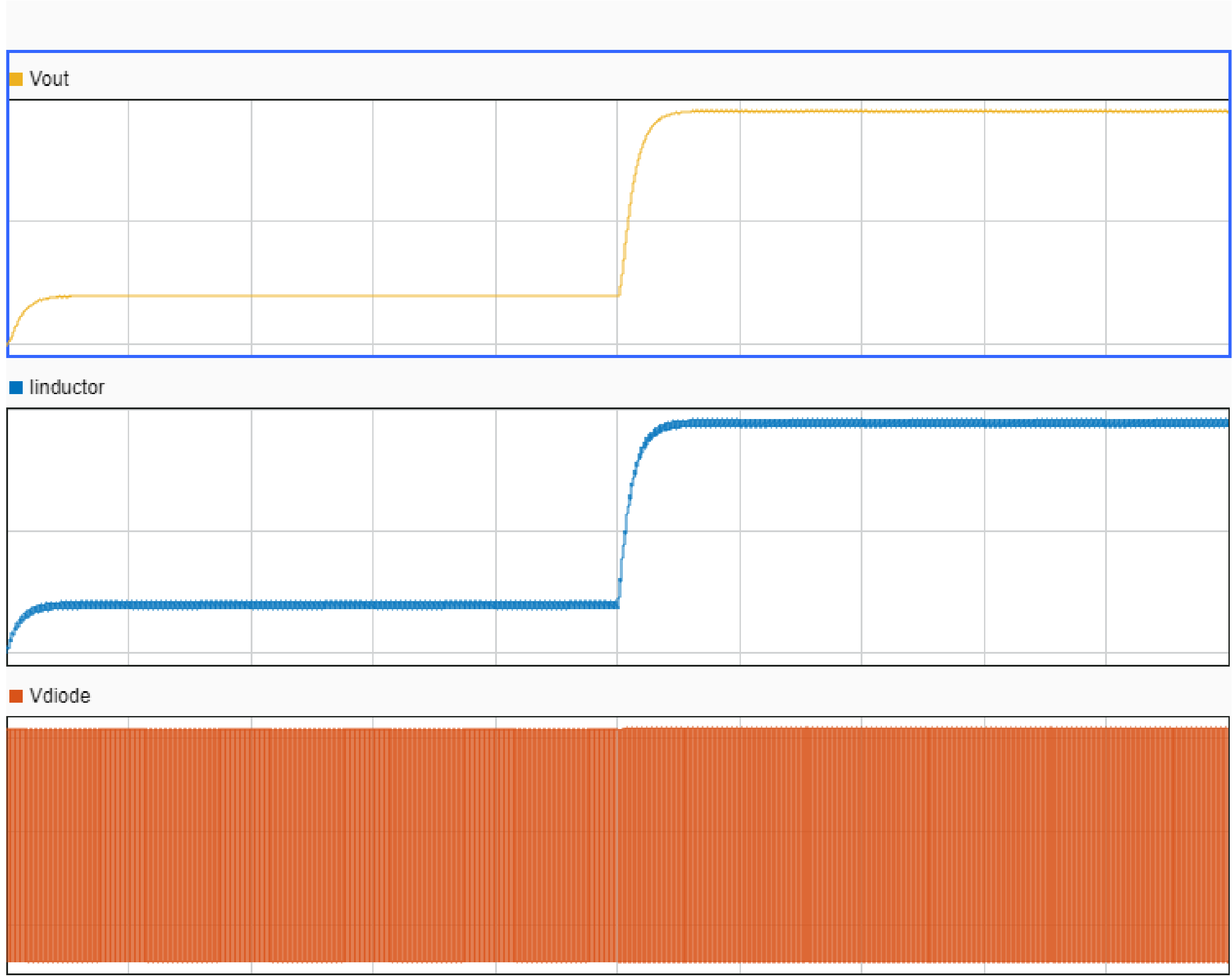
Run Desktop Simulation of HDL Implementation Model and Validate HDL Algorithm

You can simulate the switched linear state-space model of the buck converter in Simulink and display the signals in Simulation Data Inspector. The comparison of the runs show that the numeric results match.

Simulate the HDL implementation model.

```
sim('gmStateSpaceHDL_sschdlex_I0334_BuckConverte')
```

```
Warning: Unable to resolve the name
'CloneDetector.ExclusionEditorUIService.getInstance'.
```



To verify that the HDL implementation model matches the original Simscape model, generate a state-space validation model. In the **Generate implementation model** task, select the **Generate validation logic for the implementation model** check box and then run this task. Simulating the model does not display assertions, which indicates that the numeric results match. See “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89.

HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.
- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis, timing analysis, and deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

You run the Advisor for the FPGA subsystem in your model. To open the HDL Workflow Advisor for the subsystem inside the model, use the `hdladvisor` function. For example:

```
hdladvisor('gmStateSpaceHDL_sschdlex_IO334_BuckConverte/FPGA')
```

To learn about the tasks in the Advisor, right-click that task, and select **What's This?**. See “Getting Started with the HDL Workflow Advisor” on page 29-5.

Run Workflow Script to Generate Simulink Real-Time Interface Model

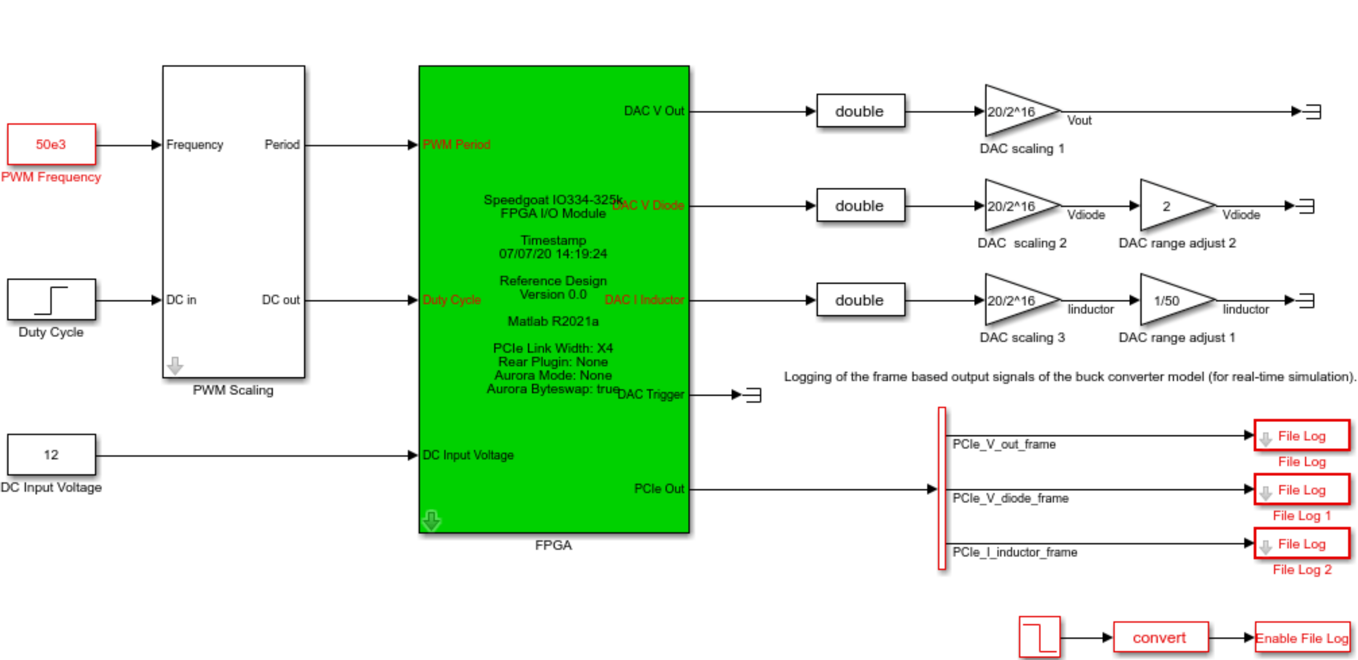
For rapid prototyping, export the HDL Workflow Advisor settings to a script. The script is a MATLAB® file that you run from the command line. You can modify and run the script, or import the settings into the HDL Workflow Advisor User Interface. See “Run HDL Workflow with a Script” on page 29-47.

This example shows how to run the HDL Workflow script. To generate a Simulink Real-Time Interface model, open and run this MATLAB script.

```
edit('hdlworkflow_buck_IO334')
```

Prepare Simulink Real-Time Interface Model for Real-Time Simulation

Running the workflow script generates RTL code and IP core, creates a Vivado project, builds the FPGA bitstream, and then generates the Simulink Real-Time Interface model.



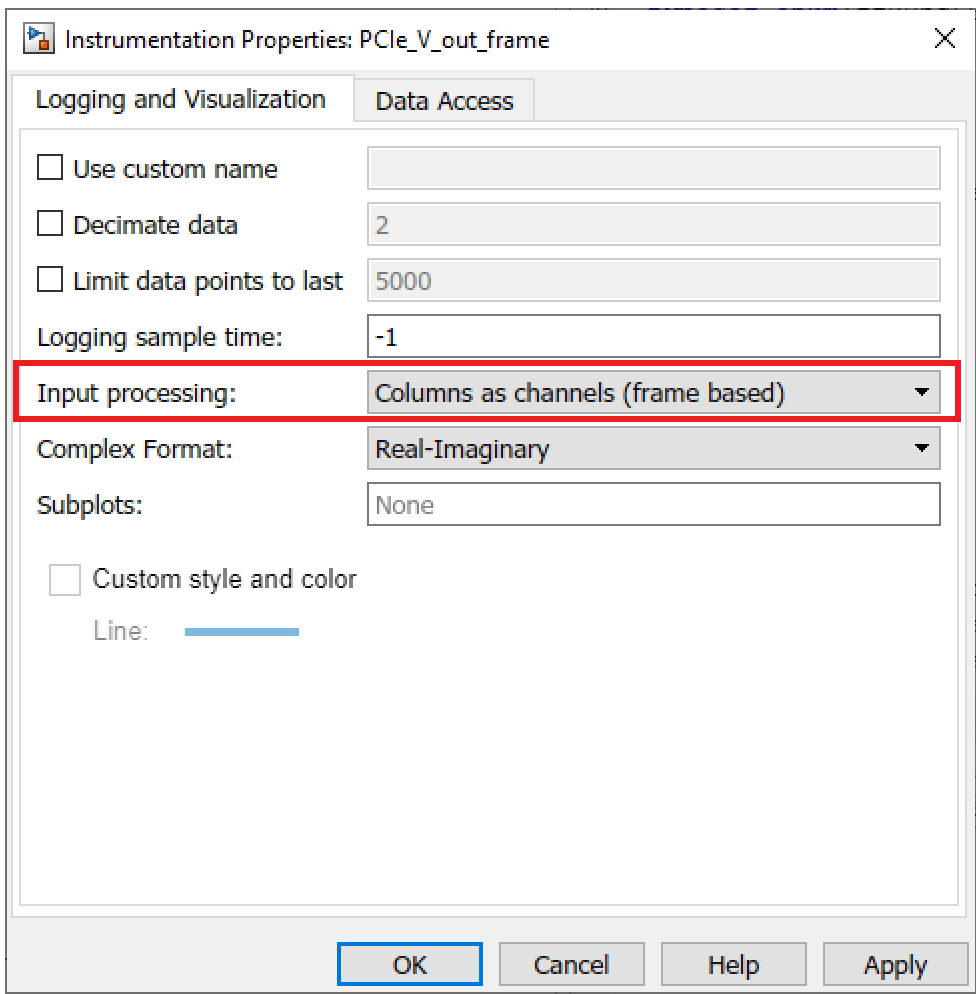
Before deploying the model to the Speedgoat real-time target machine:

1. Set the sample time for all blocks running on the CPU of the Speedgoat real-time target machine to 50us (including driver blocks for the FPGA).

```
generated_model = gcs;
```

2. Set the Simulation Data Inspector setting **Input processing** to `Columns as channels (frame based)` for the signals `PCIe_V_out_frame`, `PCIe_V_diode_frame` and `PCIe_I_inductor_frame`. Inside the mask of the `File Log` blocks, right-click the logging symbol and navigate to the Instrumentation Properties dialog box. To make the logging signal appear, you might have to update the model.

```
set_param(generated_model, 'SimulationCommand', 'update');
```



Alternatively, you can set signal logging to frame based mode by using these commands.

```
Simulink.sdi.setSignalInputProcessingMode([generated_model,'/File Log/Demux'], 1,'frame');
```

Connect to Target Machine and Run Real-Time Simulation

The model can now be deployed to the Speedgoat real-time target machine. The buck converter model is automatically loaded to the FPGA on the IO334.

Connect to the Speedgoat real-time target machine.

```
tg = slrealtime;
```

Build and download the model to the target machine.

```
rtwbuild(generated_model);
```

Start the model execution.

```
tg.start;
```

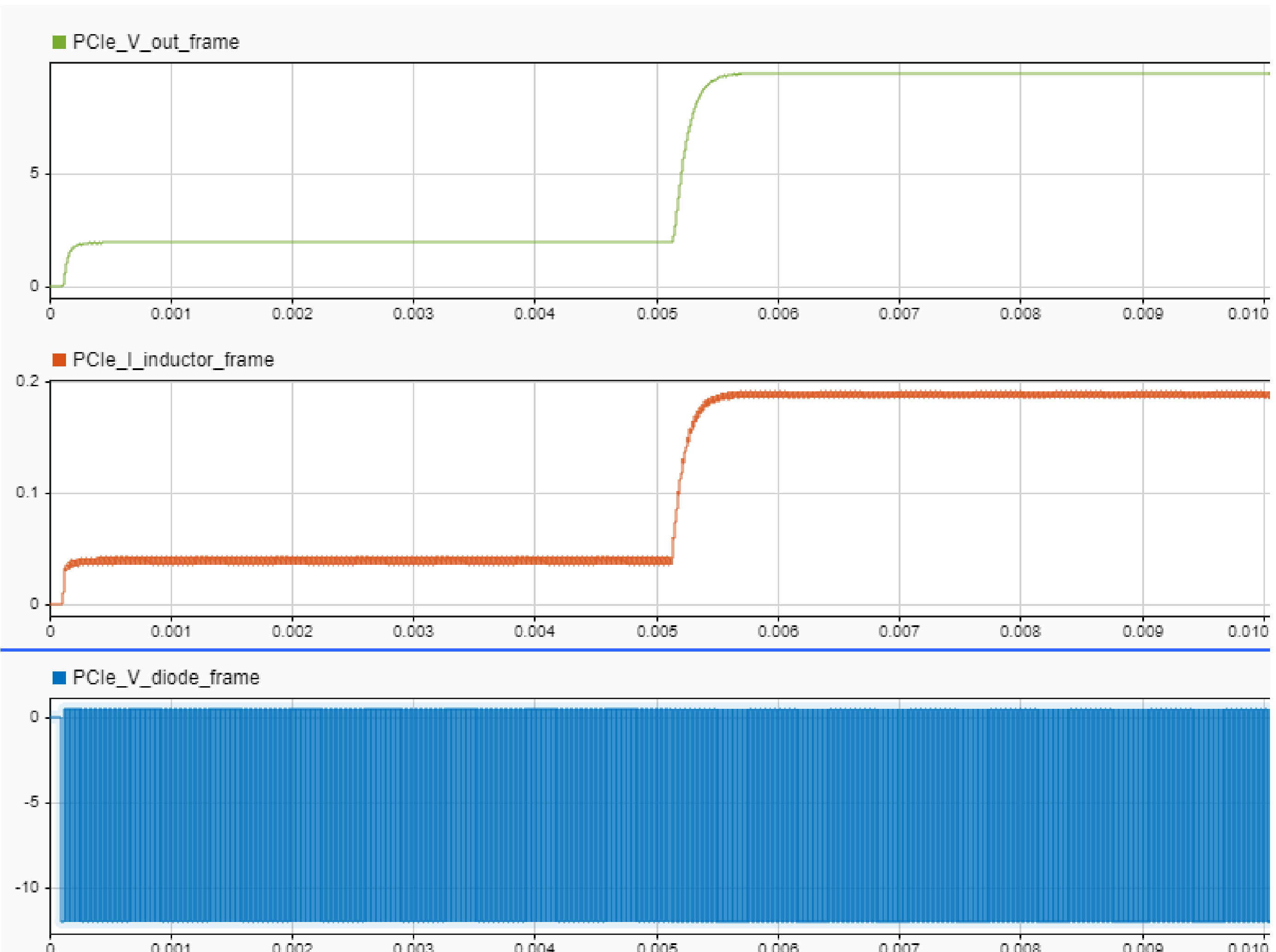
The file logging blocks store the signals on the SSD of the target-machine. The data is automatically uploaded to the host computer once the model is stopped. The data is visualized in Simulation Data Inspector. You can verify that the results of the real-time simulation matches the original Simscape model.

```
Simulink.sdi.setSubPlotLayout(3,1);
```

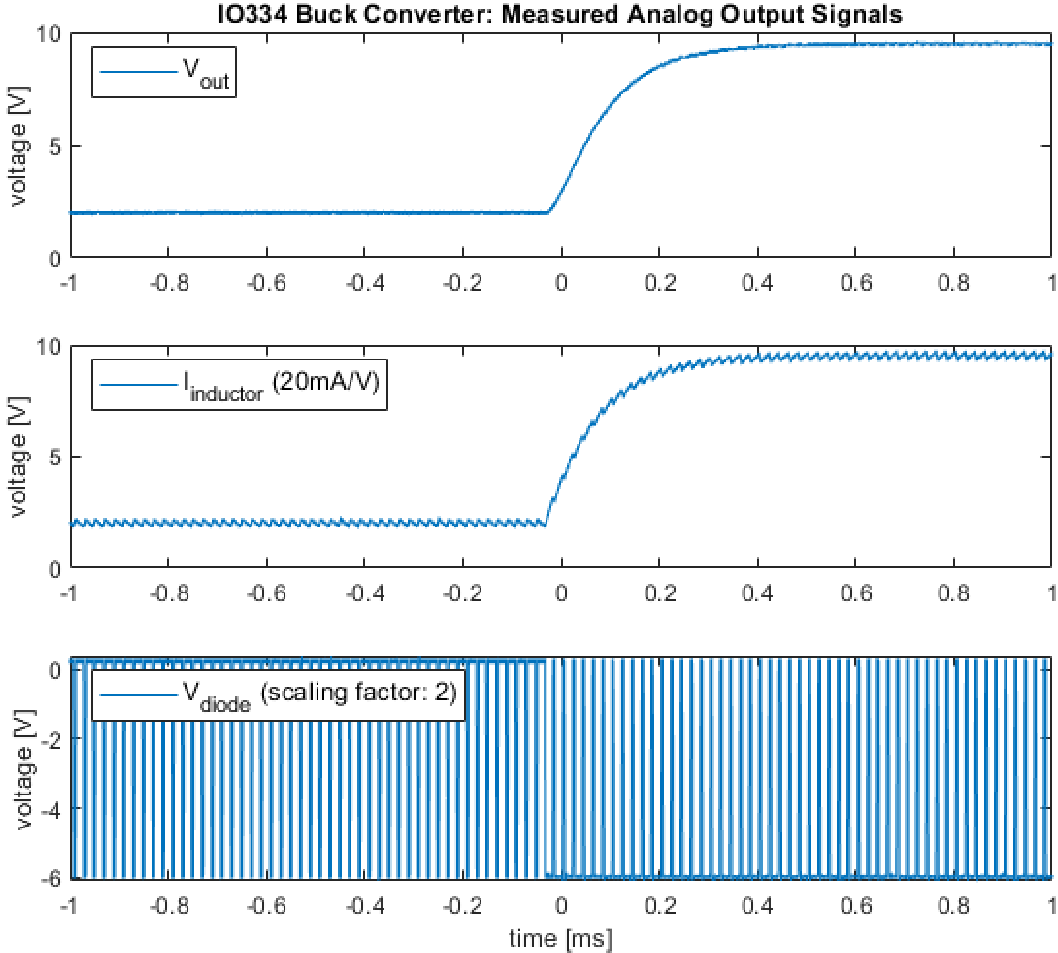
```
allIDs = Simulink.sdi.getAllRunIDs;
```

```
run.getAllSignals
```

```
Simulink.sdi.view
```



Alternatively, you can measure the signals at the analog output of the IO334. This figure shows a plot of the signals in MATLAB.



Deploy Simscape Grid Tied Converter Model to Speedgoat IO Module Using HDL Workflow Script

This example shows how to deploy a three-phase two-level voltage source converter connected to a low voltage grid modeled in Simscape™ to a Speedgoat® IO334 Simulink®-programmable I/O module to achieve a simulation time step of 1 microsecond (us).

The PWM signals can be captured with an increased resolution by modeling the converter using the sub-cycle averaging method. A maximum resolution of 4ns is possible, allowing to simulate state of the art power converters with switching frequencies above 100kHz. The switching frequency of this example is 20kHz.

This example demonstrates how to:

1. Convert your Simscape model into an HDL-compatible implementation model by using the Simscape HDL Workflow Advisor
2. Generate HDL code and FPGA bitstream for the IO334 module by using the HDL Workflow Advisor.
3. Deploy the real-time model to the Speedgoat real-time target machine by using Simulink Real-Time™.

The plant model runs eventually at a 1us time step on the FPGA and the controller is executed at a 50us time step on the CPU of the real-time system. To generate HDL code and FPGA bitstream, this example shows how to run the HDL workflow script from the command line. You can use the Workflow Advisor User Interface to run this workflow. For more information, see “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82.

Setup and Configuration

Before deploying your algorithm on the Speedgoat IO module:

1. Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.1\bin\vivado.bat')
```

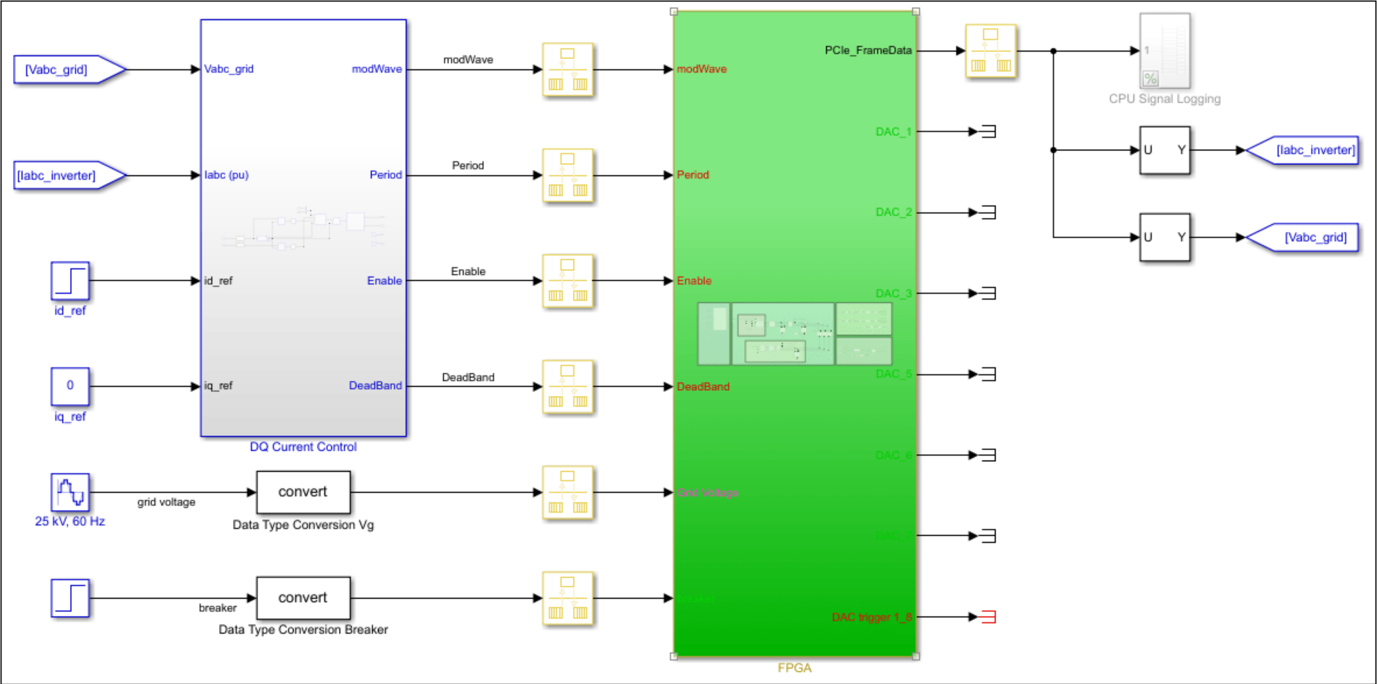
2. For real-time simulation, set up the development environment and target computer settings. See “Get Started with Simulink Real-Time” (Simulink Real-Time).

3. Install the Speedgoat Library and the Speedgoat HDL Coder Integration packages. See Install Speedgoat HDL Coder Integration Packages.

Three-Phase Two-Level Voltage Source Converter Model

To see the three-phase two-level converter model, run this command:

```
open_system('sschdlexThreePhaseConverterWithGridExample')
```



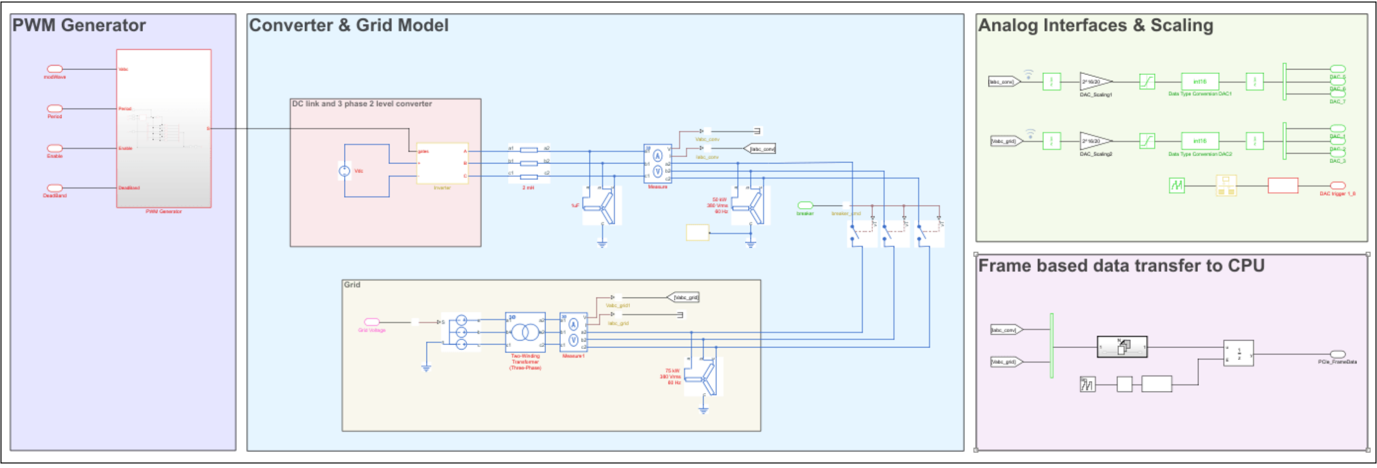
This example model consists of a three-phase two-level voltage source converter connected to a low voltage grid through an LC filter. A circuit breaker allows to disconnect and connect the converter with the grid. There is a low voltage load on the grid side and a transformer connecting to the medium voltage grid.

In the first step, the model is used to validate the closed-loop control by desktop simulation. The controller consists of a phase-locked loop for grid synchronization and PI current control in the synchronous reference frame.

In a second step, the model can be deployed to the real-time platform for HIL testing of the embedded controller.

The model has been partitioned into parts that run on the FPGA and parts that run on the CPU in real time. Parts inside the green FPGA subsystem run on the FPGA. Parts outside this subsystem run on the CPU in real time.

```
open_system('sschdlexThreePhaseConverterWithGridExample/FPGA')
```



Run Desktop Simulation of Simscape Model

The base sample rate is set to 1/(180e6) to match the clock ticks at the target frequency of 180 MHz on the hardware. To accelerate the desktop simulation, you can consider increasing the value of base sample rate.

```
Tbase = 1/(180e6);
```

Set the simulation and step time to close the circuit breaker

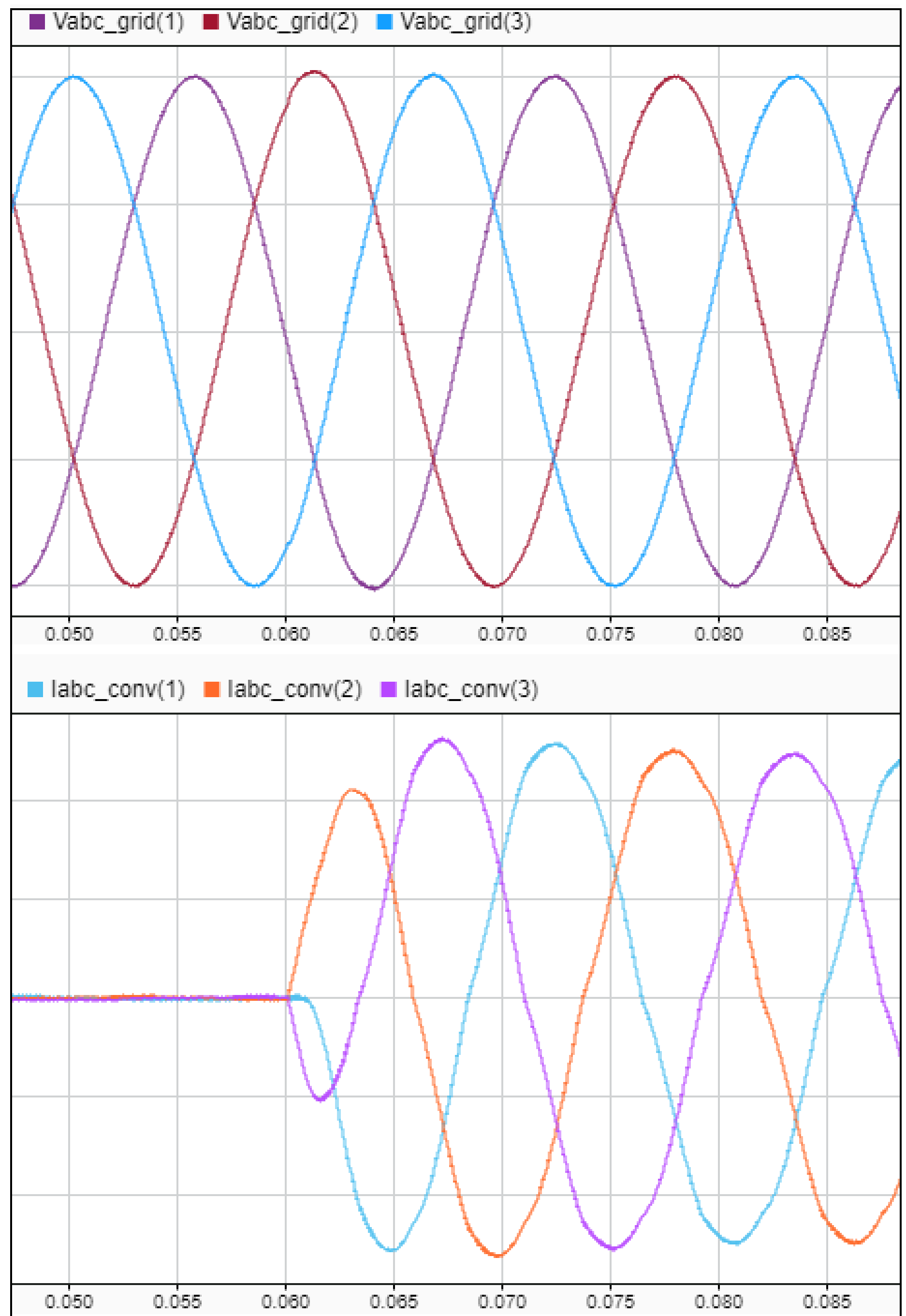
```
set_param('sschdlexThreePhaseConverterWithGridExample','StopTime','0.09');
set_param(['sschdlexThreePhaseConverterWithGridExample','/Step'],'Time','0.03');
```

Desktop simulation of the model

```
sim('sschdlexThreePhaseConverterWithGridExample')
```

Display output signals of three-phase two-level voltage source converter in Simulink Data Inspector

```
Simulink.sdi.clearAllSubPlots
Simulink.sdi.setSubPlotLayout(2,1);

allIDs = Simulink.sdi.getAllRunIDs;
runID1 = allIDs(end);
run1 = Simulink.sdi.getRun(runID1);
run1.name = 'Desktop Simulation';

Vabc_grid = run1.getSignalsByName('Vabc_grid');
Iabc_conv = run1.getSignalsByName('Iabc_conv');
plotOnSubPlot(Vabc_grid.Children(1),1,1,true);
plotOnSubPlot(Vabc_grid.Children(2),1,1,true);
plotOnSubPlot(Vabc_grid.Children(3),1,1,true);
plotOnSubPlot(Iabc_conv.Children(1),2,1,true);
plotOnSubPlot(Iabc_conv.Children(2),2,1,true);
plotOnSubPlot(Iabc_conv.Children(3),2,1,true);

Simulink.sdi.view;
```



Generate HDL Implementation Model

For HDL code generation compatibility, you run the Simscape HDL Workflow Advisor to generate an HDL implementation model.

The Simscape solver is set to run for one iteration at each sample step. The Simscape HDL Workflow Advisor uses the solver settings in the next step for deterministic real-time behavior.

The averaged switch model allows us to set the solver iterations to 1

```
set_param(['sschdlexThreePhaseConverterWithGridExample','/FPGA/Solver Configuration'],'DoFixedCos
set_param(['sschdlexThreePhaseConverterWithGridExample','/FPGA/Solver Configuration'],'MaxNonlin
```

To accelerate the `sschdladvisor` we reduce the sample time of the model. The time at which the circuit breaker closes is adapted to make sure that the workflow advisor captures all switching modes during simulation.

```
set_param('sschdlexThreePhaseConverterWithGridExample','StopTime','0.002');
set_param(['sschdlexThreePhaseConverterWithGridExample','/Step'],'Time','0.001');
```

To open the Advisor, run the `sschdladvisor` function for your model:

```
sschdladvisor('sschdlexThreePhaseConverterWithGridExample')
```

To generate the implementation model, in the Simscape HDL Workflow Advisor, keep the default settings for the tasks, and then run the tasks. Run the tasks in the Advisor by clicking the **Run all** button. You see a link to the generated model in the **Generate implementation model** task. This model has the same name as your original model with the prefix `gmStateSpaceHDL_`.

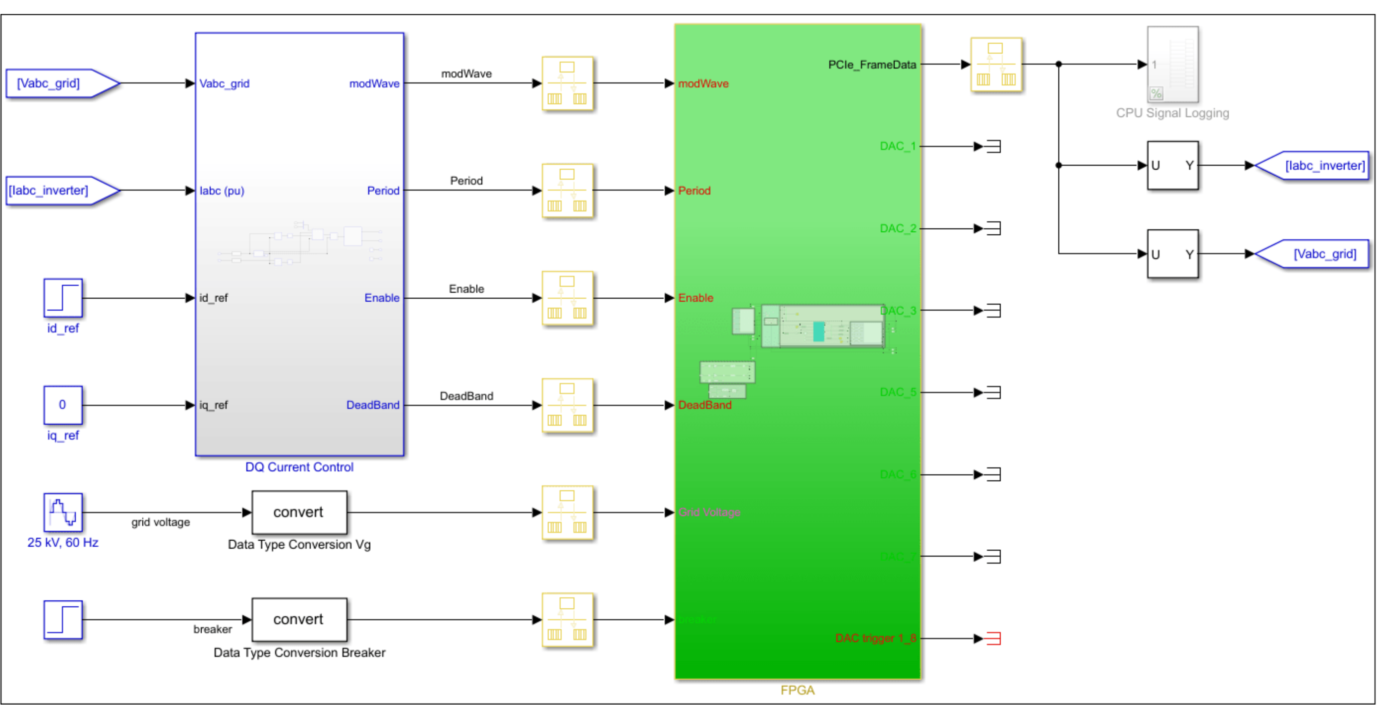
We set the simulation and step time to the original values

```
set_param('sschdlexThreePhaseConverterWithGridExample','StopTime','0.09');
set_param(['sschdlexThreePhaseConverterWithGridExample','/Step'],'Time','0.03');
```

Prepare Implementation Model for HDL Code Generation

To open the implementation model, click the link in the **Generate implementation model** task.

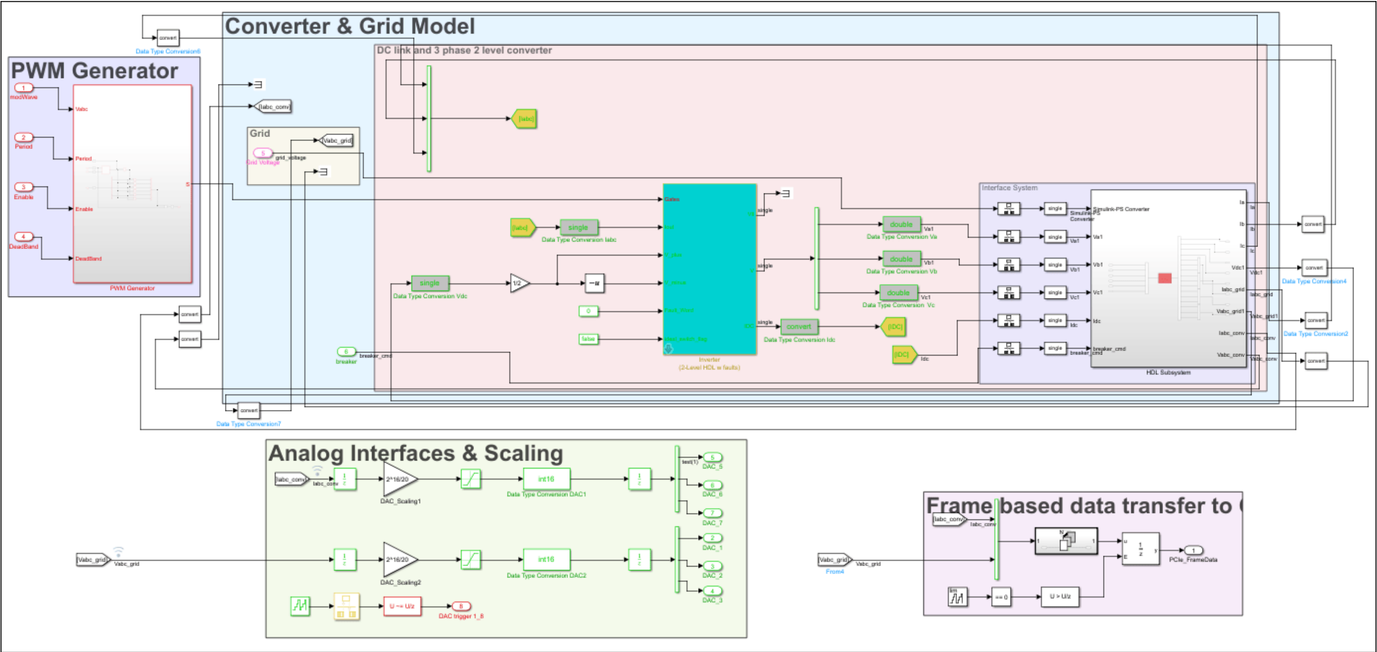
```
open_system('gmStateSpaceHDL_sschdlexThreePhaseConverter')
```




```
set_param('gmStateSpaceHDL_sschdlexThreePhaseConverter','StopTime','0.09');
set_param(['gmStateSpaceHDL_sschdlexThreePhaseConverter','/Step'],'Time','0.03');
```

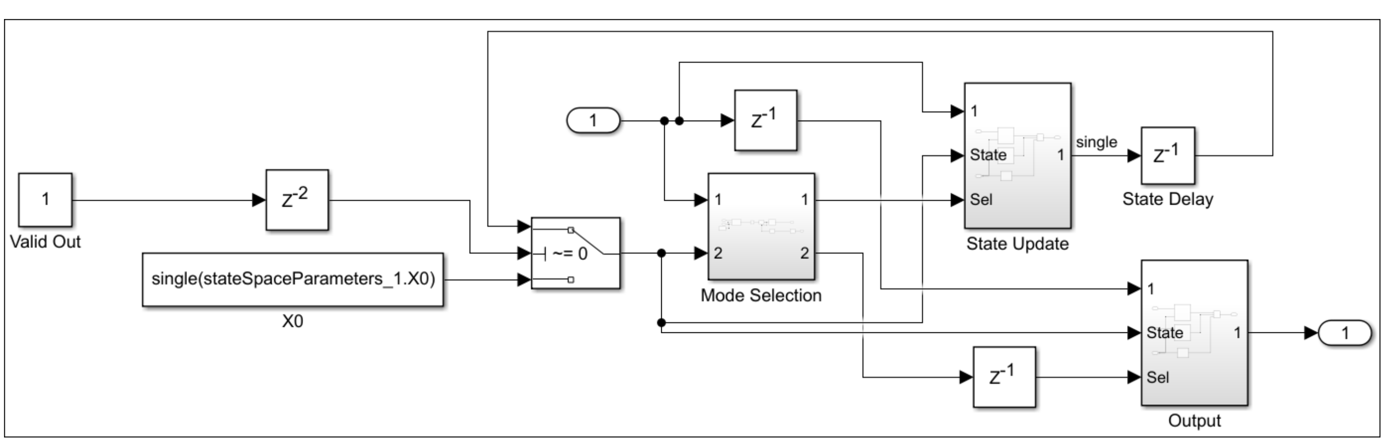
The model contains a switched linear Simulink replacement of the original three-phase two-level voltage source converter model. You see that the Simscape model was replaced.

```
open_system('gmStateSpaceHDL_sschdlexThreePhaseConverter/FPGA')
```



The implementation model replaces the Simscape subsystem with the HDL-compatible algorithm that performs the state-space computations. When you navigate inside this subsystem, you see several delays, adders, and Matrix Multiply blocks that model the state-space equations. From and Goto blocks inside this subsystem provide the same input as that of the original model to the `HDL Subsystem`.

```
open_system('gmStateSpaceHDL_sschdlexThreePhaseConverter/FPGA/HDL Subsystem/HDL Algorithm')
```



Run Desktop Simulation of HDL Implementation Model and Validate HDL Algorithm

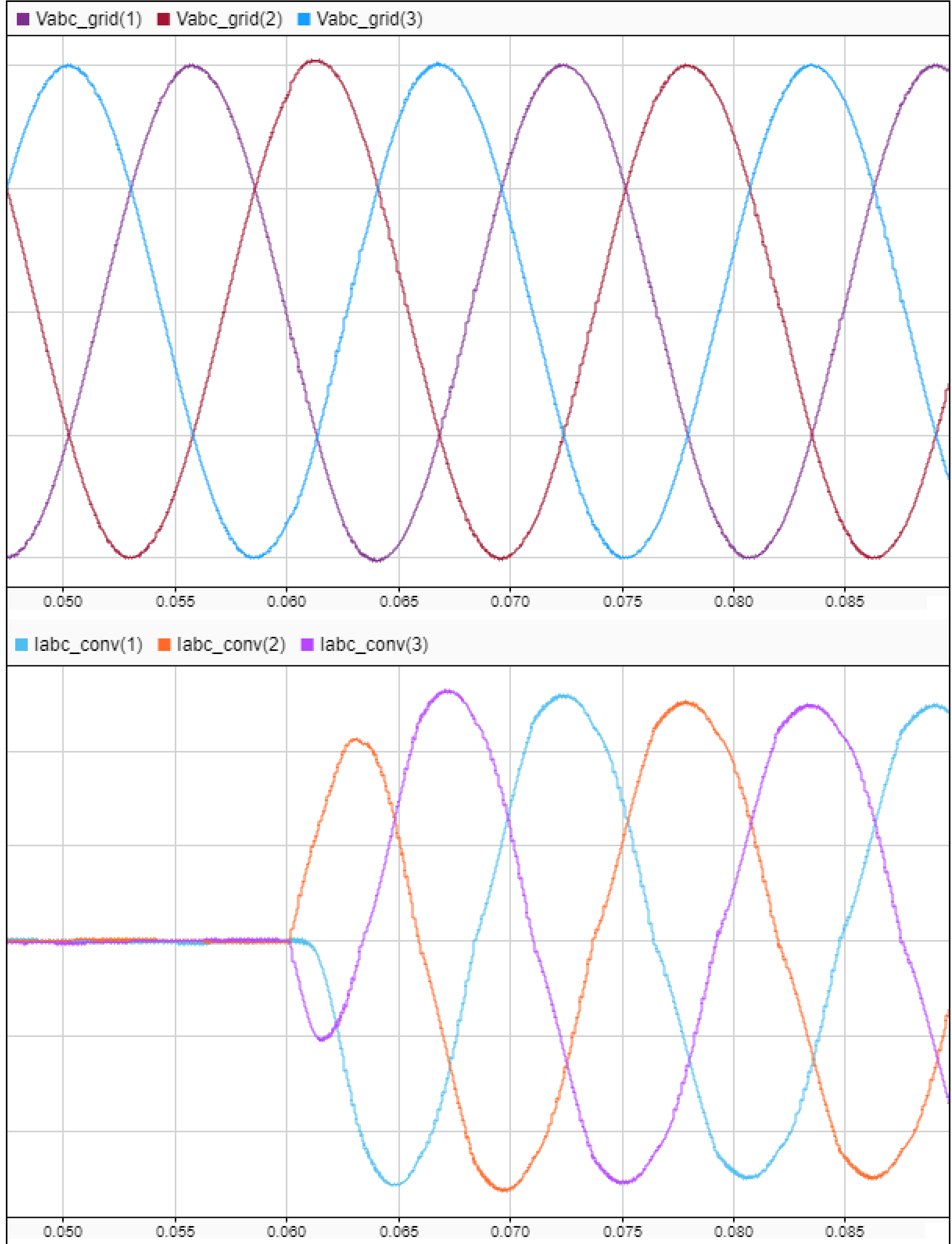
You can simulate the switched linear state-space model of the three-phase two-level voltage source converter in Simulink and display the signals in Simulation Data Inspector. The comparison of the runs show that the numeric results match with the original Simscape model.

Simulate the HDL implementation model.

```
sim('gmStateSpaceHDL_sschdlexThreePhaseConverter')
```

Display output signals of three-phase two-level voltage source converter Simscape model in Simulink Data Inspector

```
Simulink.sdi.clearAllSubPlots
Simulink.sdi.setSubPlotLayout(2,1);

allIDs = Simulink.sdi.getAllRunIDs;
runID1 = allIDs(end);
run2 = Simulink.sdi.getRun(runID1);
run2.name = 'HDL Model Simulation';

Vabc_grid = run2.getSignalsByName('Vabc_grid');
Iabc_conv = run2.getSignalsByName('Iabc_conv');
plotOnSubPlot(Vabc_grid.Children(1),1,1,true);
plotOnSubPlot(Vabc_grid.Children(2),1,1,true);
plotOnSubPlot(Vabc_grid.Children(3),1,1,true);
plotOnSubPlot(Iabc_conv.Children(1),2,1,true);
plotOnSubPlot(Iabc_conv.Children(2),2,1,true);
plotOnSubPlot(Iabc_conv.Children(3),2,1,true);

Simulink.sdi.view;
```



Vabc_grid(1)
Vabc_grid(2)
Vabc_grid(3)
0.050
0.055
0.060
0.065
0.070
0.075
0.080
0.085
Iabc_conv(1)
Iabc_conv(2)
Iabc_conv(3)
0.050
0.055
0.060
0.065
0.070
0.075
0.080
0.085

To verify that the HDL implementation model matches the original Simscape model, generate a state-space validation model. In the **Generate implementation model** task, select the **Generate validation logic for the implementation model** check box and then run this task. Simulating the model does not display assertions, which indicates that the numeric results match. See “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89.

HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.
- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis, timing analysis, and deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

You run the Advisor for the FPGA subsystem in your model. To open the HDL Workflow Advisor for the subsystem inside the model, use the `hdladvisor` function. For example:

```
hdladvisor('gmStateSpaceHDL_sschdlexThreePhaseConverter/FPGA')
```

To learn about the tasks in the Advisor, right-click that task, and select **What's This?**. See “Getting Started with the HDL Workflow Advisor” on page 29-5.

Run Workflow Script to Generate Simulink Real-Time Interface Model

You can export the HDL Workflow Advisor settings to a script to expedite and automate your workflow. The script is a MATLAB® file that you run from the command line. You can modify and run the script, or import the settings into the HDL Workflow Advisor User Interface. See “Run HDL Workflow with a Script” on page 29-47.

This example shows how to run the HDL Workflow script. To generate a Simulink Real-Time Interface model, open and run this MATLAB script.

```
edit('hdlworkflow_IO334')
```

```
%% ------------------------------------------------------------------------
% This script contains the model, target settings, interface mapping, and
% the Workflow Configuration settings for generating HDL code for the HDL
% implementation model generated for the three-phase two-level voltage source
% converter model, and for deploying the code to the FPGA on board the
% Speedgoat IO334-325K module.
%% -----------------------------------------------------------------------

% Make the model ready for HDL code generation

% Set all data type conversion blocks to type single (except the
% conversions for the analog output signals.
DataTypeConversionBlocks = Simulink.findBlocksOfType([HDLModelName,'/FPGA'],'DataTypeConversion'
for i = 1:length(DataTypeConversionBlocks)
    BlockName = get_param(DataTypeConversionBlocks(i),'Name');
    if ~any(strcmp(BlockName,{'Data Type Conversion DAC1','Data Type Conversion DAC2'}))
        set_param([HDLModelName,'/FPGA/',BlockName],'OutDataTypeStr','single');
    end
end
```

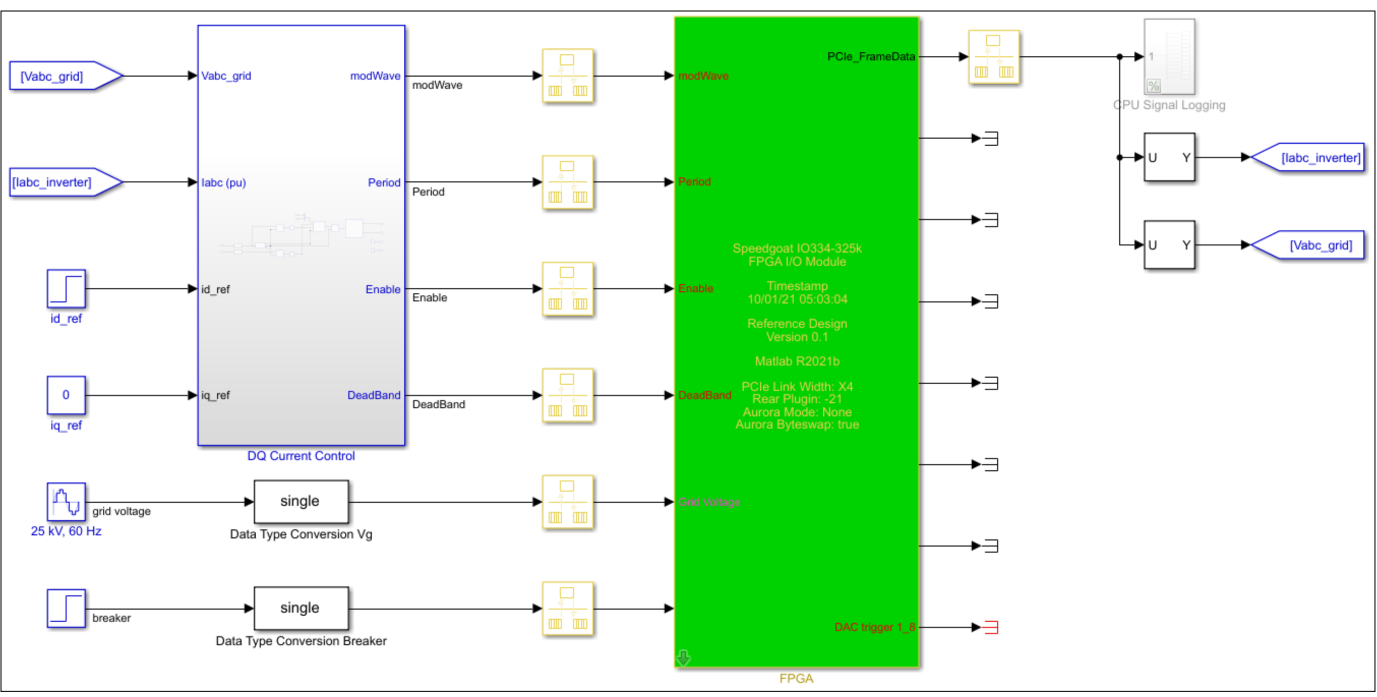
```
set_param([HDLModelName,'/Data Type Conversion Vg'],'OutDataTypeStr','single');
set_param([HDLModelName,'/Data Type Conversion Breaker'],'OutDataTypeStr','single');

% Set the base sample time to 1/(180e6) (equal to a clock tick at 180MHz target
% frequency). Note: The base rate determines the resolution of the PWM
% signal when running the model on hardware.
Tbase = 1/180e6;

% Model HDL parameters
% ----------------------------------------------------------------------------

% Set the model HDL parameters
hdlset_param(HDLModelName, 'FloatingPointTargetConfiguration', hdlcoder.createFloatingPointTarge
hdlset_param(HDLModelName, 'HDLSubsystem', [HDLModelName,'/FPGA']);
hdlset_param(HDLModelName, 'Oversampling',1);
hdlset_param(HDLModelName, 'ScalarizePorts', 'DUTLevel');
hdlset_param(HDLModelName, 'TargetFrequency', 180);
hdlset_param(HDLModelName, 'Workflow', 'Simulink Real-Time FPGA I/O');
hdlset_param(HDLModelName, 'TargetPlatform', 'Speedgoat IO334-325k');
hdlset_param(HDLModelName, 'TargetDirectory', 'c:\hdl_prj\hdlsrc');
hdlset_param(HDLModelName,'ResetType','Synchronous')
hdlset_param(HDLModelName, 'AdaptivePipelining', 'on');
hdlset_param(HDLModelName, 'ReferenceDesignParameter', {'PCIe_Link_Width','X4','RearPlugin','None

% Block ram options
hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm/State Update/Multiply Input'], 'Use
hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm/State Update/Multiply State'], 'Use

% Set input and output pipeline registers
hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm'], 'ConstrainedOutputPipeline', 2)
hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm'], 'InputPipeline', 2);
hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm'], 'OutputPipeline', 2);

hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm/Mode Selection'], 'ConstrainedOutpu
hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm/Mode Selection'], 'OutputPipeline',

hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm/Output'], 'ConstrainedOutputPipelin
hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm/Output'], 'OutputPipeline', 2);

hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm/State Update'], 'ConstrainedOutputP
hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm/State Update'], 'OutputPipeline', 2

% Enable flatten hyrarchy
hdlset_param([HDLModelName,'/FPGA'], 'FlattenHierarchy', 'on');

% Set the code generation options for the MATLAB function to "data path"
hdlset_param([HDLModelName,'/FPGA/HDL Subsystem/HDL Algorithm/Mode Selection/Generate Mode Vecto

% We define the hardware interfaces for all input and output ports to the
% FPGA subsystem

% Set inport HDL parameters
hdlset_param([HDLModelName,'/FPGA/modWave'], 'IOInterface', 'PCIe Interface');
hdlset_param([HDLModelName,'/FPGA/modWave'], 'IOInterfaceMapping', 'x"100"');

hdlset_param([HDLModelName,'/FPGA/Period'], 'IOInterface', 'PCIe Interface');
hdlset_param([HDLModelName,'/FPGA/Period'], 'IOInterfaceMapping', 'x"114"');
```

```
hdlset_param([HDLModelName,'/FPGA/Enable'], 'IOInterface', 'PCIe Interface');
hdlset_param([HDLModelName,'/FPGA/Enable'], 'IOInterfaceMapping', 'x"118"');

hdlset_param([HDLModelName,'/FPGA/DeadBand'], 'IOInterface', 'PCIe Interface');
hdlset_param([HDLModelName,'/FPGA/DeadBand'], 'IOInterfaceMapping', 'x"11C"');

hdlset_param([HDLModelName,'/FPGA/Grid Voltage'], 'IOInterface', 'PCIe Interface');
hdlset_param([HDLModelName,'/FPGA/Grid Voltage'], 'IOInterfaceMapping', 'x"120"');

hdlset_param([HDLModelName,'/FPGA/breaker'], 'IOInterface', 'PCIe Interface');
hdlset_param([HDLModelName,'/FPGA/breaker'], 'IOInterfaceMapping', 'x"134"');

% Set outport HDL parameters
hdlset_param([HDLModelName,'/FPGA/PCIe_FrameData'], 'IOInterface', 'PCIe Interface');
hdlset_param([HDLModelName,'/FPGA/PCIe_FrameData'], 'IOInterfaceMapping', 'x"1000"');

hdlset_param([HDLModelName,'/FPGA/DAC_1'], 'IOInterface', 'IO334 AO Data [0:15]');
hdlset_param([HDLModelName,'/FPGA/DAC_1'], 'IOInterfaceMapping', 'Channel 01');

hdlset_param([HDLModelName,'/FPGA/DAC_2'], 'IOInterface', 'IO334 AO Data [0:15]');
hdlset_param([HDLModelName,'/FPGA/DAC_2'], 'IOInterfaceMapping', 'Channel 02');

hdlset_param([HDLModelName,'/FPGA/DAC_3'], 'IOInterface', 'IO334 AO Data [0:15]');
hdlset_param([HDLModelName,'/FPGA/DAC_3'], 'IOInterfaceMapping', 'Channel 03');

hdlset_param([HDLModelName,'/FPGA/DAC_5'], 'IOInterface', 'IO334 AO Data [0:15]');
hdlset_param([HDLModelName,'/FPGA/DAC_5'], 'IOInterfaceMapping', 'Channel 05');

hdlset_param([HDLModelName,'/FPGA/DAC_6'], 'IOInterface', 'IO334 AO Data [0:15]');
hdlset_param([HDLModelName,'/FPGA/DAC_6'], 'IOInterfaceMapping', 'Channel 06');

hdlset_param([HDLModelName,'/FPGA/DAC_7'], 'IOInterface', 'IO334 AO Data [0:15]');
hdlset_param([HDLModelName,'/FPGA/DAC_7'], 'IOInterfaceMapping', 'Channel 07');

hdlset_param([HDLModelName,'/FPGA/DAC trigger 1_8'], 'IOInterface', 'IO334 AO Trigger [0:1]');
hdlset_param([HDLModelName,'/FPGA/DAC trigger 1_8'], 'IOInterfaceMapping', 'Channel 01 to 08');

%% Workflow Configuration Settings

% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Simulink Real-Ti

% Specify the top level project directory (make sure to define a short
% specific path)
hWC.ProjectFolder = 'c:/hdl_prj';
hWC.IgnoreToolVersionMismatch = true;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.SpeedOptimized;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

%% Run the workflow
```

```
hdlcoder.runWorkflow([HDLModelName,'/FPGA'], hWC);

%% Get the model ready for deployment on the Speedgoat real-time target machine

% Get the name of the generate real-time model
SLRTModelName = ['gm_',HDLModelName,'_slrt'];

% Comment through all rate transition blocks. The entire CPU model runs at
% a time step of 50us. The rate transitions between the FPGA and CPU model
% happen implicitly through the CPU read and write operations.
RateTransitionBlocks = Simulink.findBlocksOfType(SLRTModelName,'RateTransition',Simulink.FindOpt:
for i = 1:length(RateTransitionBlocks)
    BlockName = get_param(RateTransitionBlocks(i),'Name');
    if ~any(strcmp(BlockName,{'Data Type Conversion DAC1','Data Type Conversion DAC2'}))
        set_param([SLRTModelName,'/',BlockName],'Commented','through');
    end
end

% Uncomment the subsystem for data logging on the real-time CPU
set_param([SLRTModelName,'/CPU Signal Logging'],'Commented','off');
```

Prepare Simulink Real-Time Interface Model for Real-Time Simulation

Running the workflow script generates RTL code and IP core, creates a Vivado project, builds the FPGA bitstream, and then generates the Simulink Real-Time Interface model.



Before deploying the model to the Speedgoat real-time target machine, getting model ready for real time testing:

```
generated_model = get_param(gcs,'Name');
```

```
% comment through all rate transition blocks
RateTransitionBlocks = Simulink.findBlocksOfType(generated_model,'RateTransition',Simulink.FindOp
for i = 1:length(RateTransitionBlocks)
    BlockName = get_param(RateTransitionBlocks(i),'Name');
    if ~any(strcmp(BlockName,{'Data Type Conversion DAC1','Data Type Conversion DAC2'}))
        set_param([generated_model,'/',BlockName],'Commented','through');
    end
end


% uncomment the subsystem for logging
set_param([generated_model,'/CPU Signal Logging'],'Commented','off');
```

Connect to Target Machine and Run Real-Time Simulation

The model can now be deployed to the Speedgoat real-time target machine. The three-phase two-level voltage source converter model is automatically loaded to the FPGA on the IO334.

Connect to the Speedgoat real-time target machine.

```
tg = slrealtime;
connect(tg);
```

Build and download the model to the target machine.

```
model = 'generated_model';
open_system(model);
modelSTF = getSTFName(tg);
set_param(model,"SystemTargetFile",modelSTF)
set_param(model,'FixedStep','.01');
evalc('slbuild(model)');
```

Start the model execution.

```
load(tg,model);
start(tg);


while strcmp(tg.status,'running')
    pause(10);
end
```

The file logging blocks store the signals on the SSD of the target-machine. The data is automatically uploaded to the host computer once the model is stopped. The data is visualized in Simulation Data Inspector. You can verify that the results of the real-time simulation matches the original Simscape model.

```
% Display grid voltage and converter current in SDI
Simulink.sdi.clearAllSubPlots
Simulink.sdi.setSubPlotLayout(2,1);

allIDs = Simulink.sdi.getAllRunIDs;
runID1 = allIDs(end);
run3 = Simulink.sdi.getRun(runID1);
run3.name = 'Real-Time Simulation';

Vabc_grid = run3.getSignalsByName('Vabc_grid');
Iabc_conv = run3.getSignalsByName('Iabc_conv');
```

```
plotOnSubPlot(Vabc_grid.Children(1),1,1,true);
plotOnSubPlot(Vabc_grid.Children(2),1,1,true);
plotOnSubPlot(Vabc_grid.Children(3),1,1,true);
plotOnSubPlot(Iabc_conv.Children(1),2,1,true);
plotOnSubPlot(Iabc_conv.Children(2),2,1,true);
plotOnSubPlot(Iabc_conv.Children(3),2,1,true);

Simulink.sdi.view;
```

Alternatively, you can measure the signals at the analog output of the IO334.

See Also

Functions
`checkhdl` | `makehdl`

More About

- “Run HDL Workflow with a Script” on page 29-47
- “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 40-145
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 40-113
- “Simscape HDL Workflow Advisor Tasks” on page 31-2
- Speedgoat I/O Examples

Partition Simscape Models Containing a Large Network into Multiple Smaller Networks

This example shows how to partition a solar power inverter model that contains a single large Simscape™ network into multiple networks. After you partition the network, you can run the Simscape HDL Workflow Advisor to generate the HDL implementation model. To learn how to run the Advisor for the model, see “Generate HDL Code for Simscape Models with Multiple Networks” on page 30-67.

Why Partition a Simscape Network

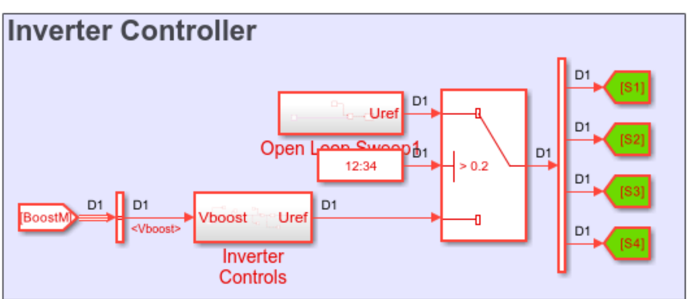
When your Simscape model contains many switching elements, the state-space representation can contain a large number of modes. The Simscape HDL Workflow Advisor simulates the Simscape model to calculate the number of modes that are relevant. Certain Simscape models can have a large number of modes that are relevant. The generated HDL implementation model for such a large design can consume a significant amount of resources, and the generated HDL implementation model may even fail to synthesize on the target FPGA device. To reduce the number of modes, you can partition the Simscape network in your model into multiple networks, and then run the Simscape HDL Workflow Advisor.

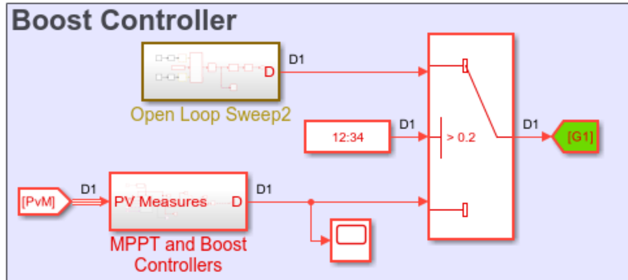
Open Solar Power Inverter Model with Single Network

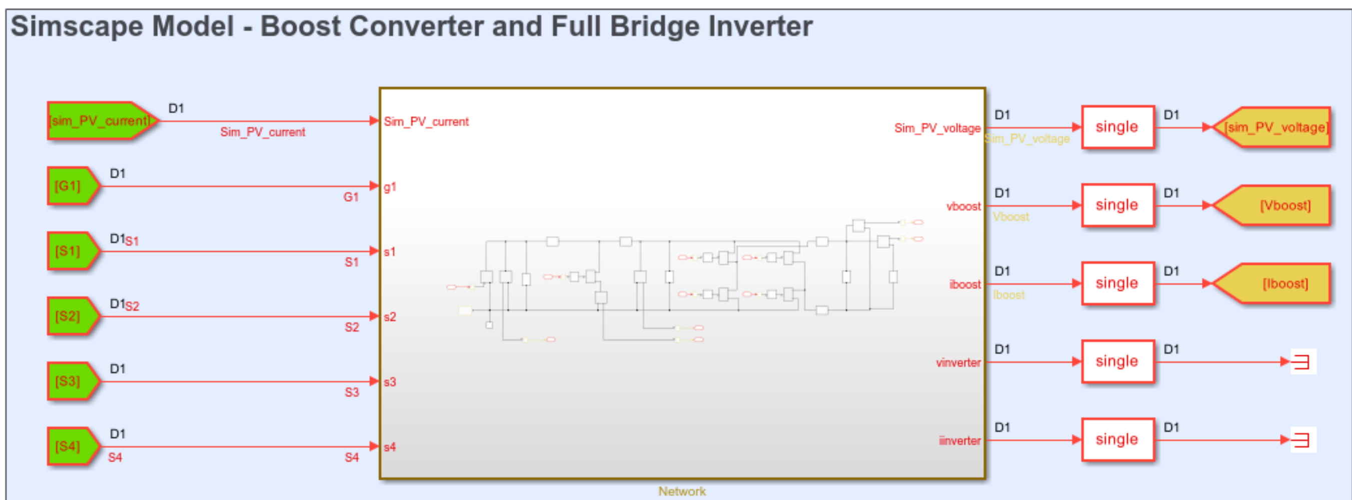
To open the solar power inverter example model, run:

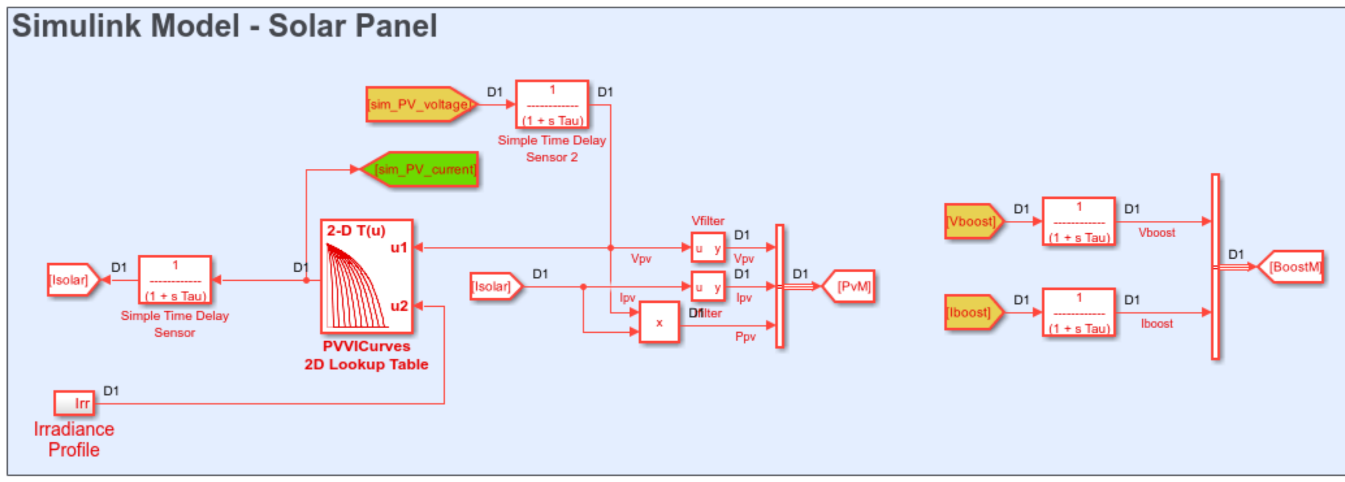
```
open_system('sschdlexSolarInverterSingleNetworkExample')
set_param('sschdlexSolarInverterSingleNetworkExample', 'SimulationCommand', 'Update')
```

Solar Panel and Inverter with Controls









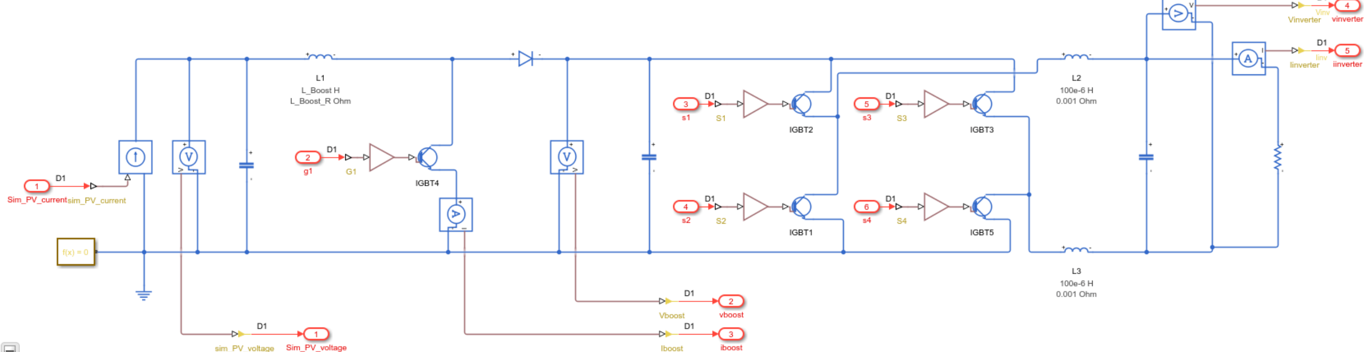
Copyright 2019-2023 The MathWorks, Inc.

The model consists of four parts: an inverter controller, a boost controller, a boost converter and full bridge inverter, and a solar panel. The solar panel is modeled in Simulink® by using lookup tables.

The boost controller and inverter controller provide the control signals for the boost converter and the full bridge inverter, which is an H-bridge.

To see the boost converter and full bridge inverter, open the Network subsystem.

```
open_system('sschdlexSolarInverterSingleNetworkExample/Network')
```



Run Simscape HDL Workflow Advisor

1. To open the Simscape HDL Workflow Advisor for the model, enter:

```
sschdladvisor('sschdlexSolarInverterSingleNetworkExample')
```

2. Run the workflow to the **Extract discrete equations** task. You see that the state-space representation uses 86 modes, which is a large number of modes.

Details related to the Simscape network sschdlexSolarInverterSingleNetworkExample/Network/Solver Configuration

- Number of states: 18
- Number of inputs: 6
- Number of outputs: 5
- Number of modes: 86
- Number of differential variables: 6

Summary of the state-space representation:

Parameter	Parameter size
A	18 x 18 x 86
B	18 x 6 x 86
F0	18 x 1 x 86
C	5 x 18 x 1
D	5 x 6 x 1
Y0	5 x 1 x 1

Such a large number of modes can consume a significant amount of hardware resources, and may even cause the DUT subsystem in the HDL implementation model to fail to synthesize on the target FPGA device.

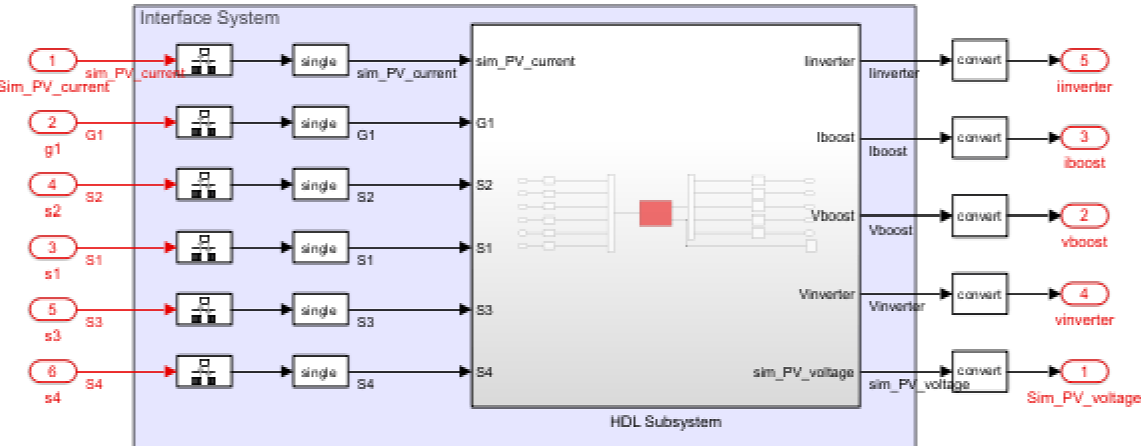
Generate HDL Implementation Model and View Resource Consumption

To see the resource consumption:

1. Run the **Generate implementation model** task. After the task passes, you see a link to the HDL implementation model `gmStateSpaceHDL_sschdlexSolarInverterSingle`. Click the link to open the HDL implementation model.

The model contains an HDL Subsystem block that models the state-space equations for the Simscape network. Open the generated HDL implementation model `gmStateSpaceHDL_sschdlexSolarInverterSingle`.

```
open_system('gmStateSpaceHDL_sschdlexSolarInverterSingle/Network')
```



2. Enable generation of the resource utilization report.

```
hdlset_param('gmStateSpaceHDL_sschdlexSolarInverterSingle', 'ResourceReport', 'on')
```

3. Generate code for the HDL Subsystem block.

```
makehdl('gmStateSpaceHDL_sschdlexSolarInverterSingle/Network/HDL Subsystem')
```

4. As you generate HDL code, open the Code Generation Report. The resource utilization report shows large numbers of multipliers, adders, and registers that might be used on the target FPGA device.

Summary

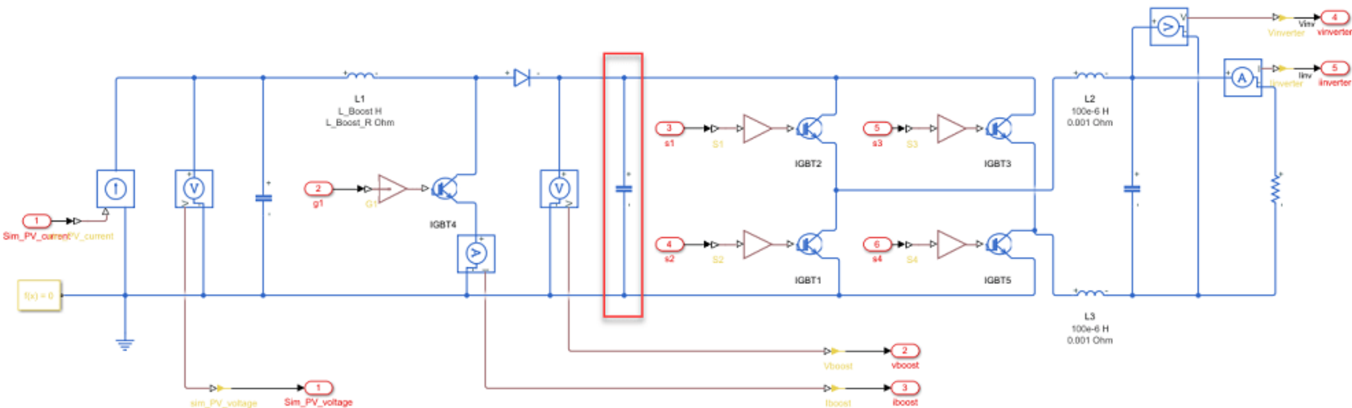
Multipliers	168
Adders/Subtractors	2820
Registers	15169
Total 1-Bit Registers	137384
RAMs	0
Multiplexers	23923
I/O Bits	356
Static Shift operators	503
Dynamic Shift operators	334

Partition Solar Inverter Network into Multiple Simscape Networks

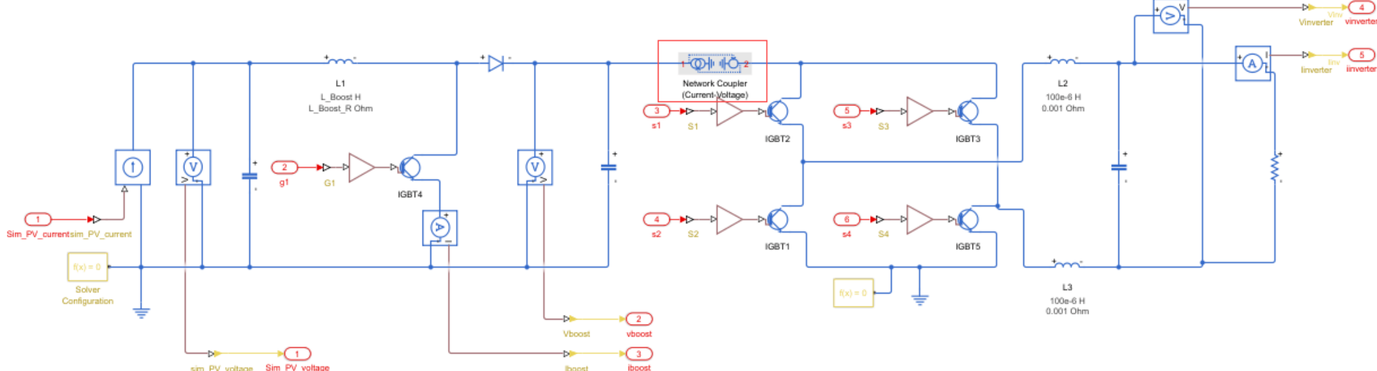
To reduce the number of modes, you can partition the Simscape network inside the Network subsystem into two Simscape networks. To partition the network into multiple networks you can use a Network Coupler (Current-Voltage) (Simscape) block. The Network Coupler (Current-Voltage) block provides a starting point for you to split a Simscape™ network into two coupled networks at an electrical connection.

1. Identify the boundary for partitioning the network into multiple networks. To produce a Simscape model that contains multiple networks and effectively reduces the number of modes in the state-space representation, choose a boundary that produces identical or near identical partitions. That is, the number of switching elements on either side of the boundary are identical or nearly identical.

For the solar power inverter, you can choose the DC link capacitor between the full bridge inverter and the boost converter as the boundary for partitioning the network.



Once you choose the boundary, connect the Network Coupler (Current-Voltage) for partitioning the network into two Simscape networks.



2. After you partition the network, prepare the modified Simscape model for compatibility with the Simscape HDL Workflow Advisor. Use a Solver Configuration block for each partition of the network as the Simscape HDL Workflow Advisor uses the Solver Configuration block to identify each unique network in your Simscape model.

Open Solar Power Inverter Model with Multiple Networks

The single network model is now partitioned into multiple networks. To open the model containing multiple networks, enter:

```
open_system('sschdlexSolarInverterPartitionedNetworkExample')
```

To learn how you run the Simscape HDL Workflow Advisor and generate HDL code for this model, see “Generate HDL Code for Simscape Models with Multiple Networks” on page 30-67.

See Also

Functions

`checkhdl` | `makehdl` | `sschdladvisor`

More About

- “Generate HDL Code for Simscape Models” on page 30-13
- “Simscape HDL Workflow Advisor Tasks” on page 31-2
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 31-7
- “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89

Generate HDL Code for Simscape Models with Multiple Networks

This example shows how you can run the Simscape HDL Workflow Advisor to generate the HDL implementation model for a Simscape™ model that contains multiple networks. You can also generate a validation logic that numerically compares each Simscape network with the corresponding state-space implementation in the HDL implementation model. The Simscape model in this example is a solar power inverter partitioned into two networks. To learn how this network is partitioned, see “Partition Simscape Models Containing a Large Network into Multiple Smaller Networks” on page 30-61.

Why Use a Simscape Model with Multiple Networks

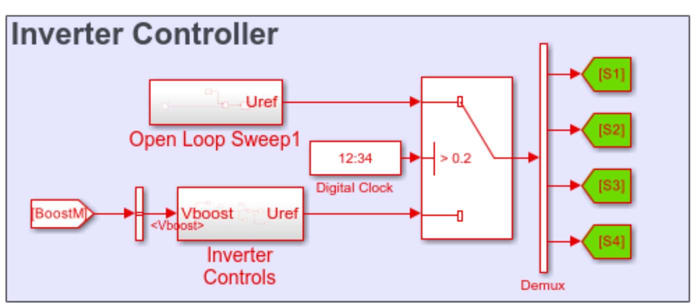
When your Simscape model contains many switching elements, the state-space representation can contain a large number of modes. The generated HDL implementation model for such a large design can consume a significantly large number of resources, and may even fail to synthesize on the target FPGA device. To reduce the number of modes, you can partition the Simscape network in your model into multiple networks, and then run the Simscape HDL Workflow Advisor.

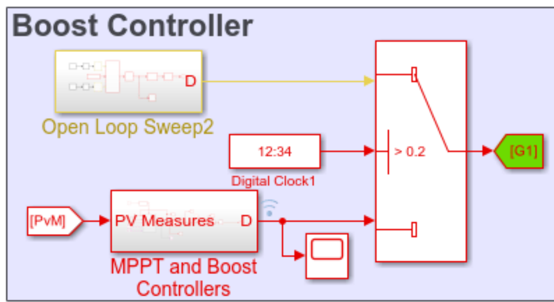
Solar Power Inverter Model with Multiple Networks

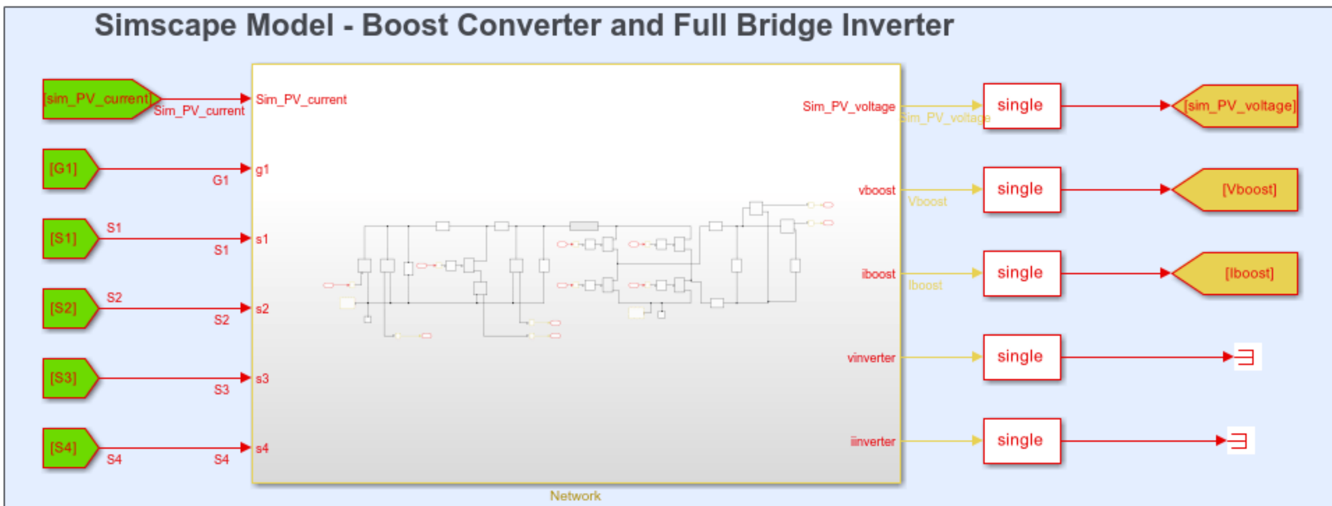
To open the model that contains multiple networks, run:

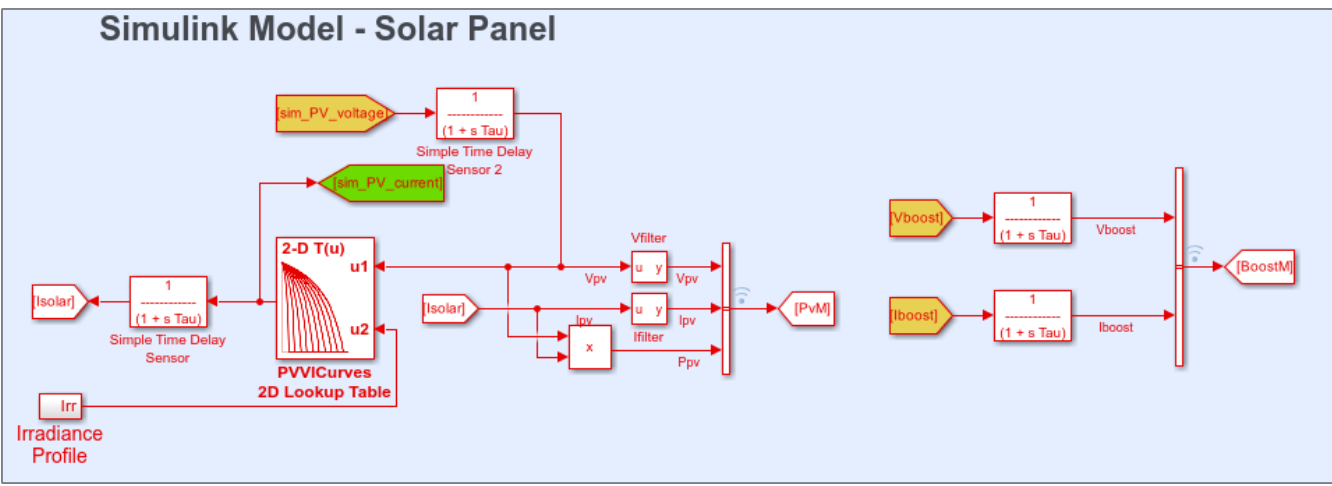
```
open_system('sschdlexSolarInverterPartitionedNetworkExample')
```

Solar Panel and Inverter with Controls









In this example, a Network coupler (Current-Voltage) is used at the interface of Boost Converter and Full Bridge Inverter to partition the single network into two networks.

Copyright 2023 The MathWorks, Inc.

The model consists of four parts: solar panel, boost controller, inverter controller, and a boost converter and full bridge inverter. The solar panel is modeled in Simulink® by using lookup tables.

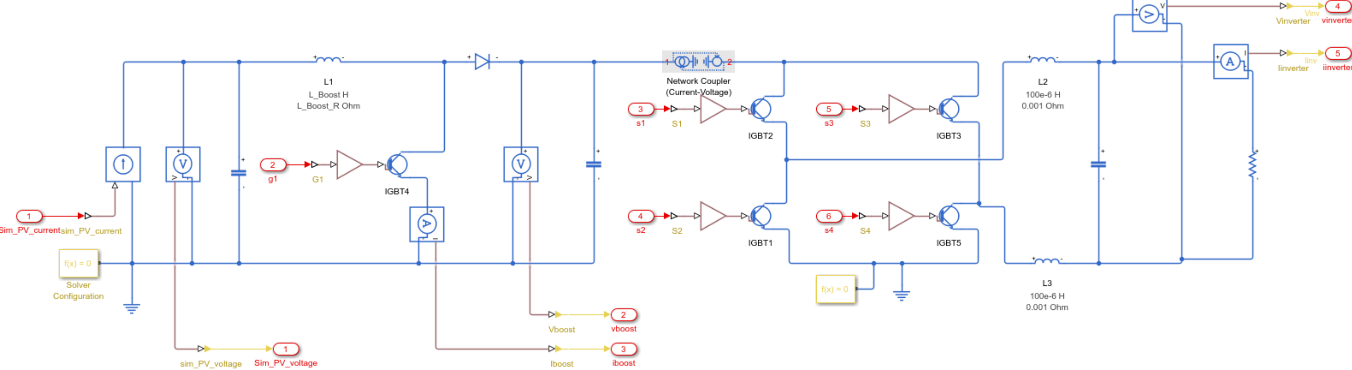
The boost controller and inverter controller provide the control signals for the boost converter and the full bridge inverter which is an H-bridge.

The original model contains the boost converter and full bridge inverter as a single network inside one subsystem. To see this model, enter:

```
open_system('sschdlexSolarInverterSingleNetworkExample')
```

The partitioned model contains the two networks inside a single subsystem. To see the partitioned network, open the Network subsystem.

```
open_system('sschdlexSolarInverterPartitionedNetworkExample/Network')
```



Run Simscape HDL Workflow Advisor for Model with Multiple Networks

1. To open the Simscape HDL Workflow Advisor for the model, enter:

```
sschdladvisor('sschdlexSolarInverterPartitionedNetworkExample')
```

2. Run the workflow to the **Check model compatibility** task.

The Simscape HDL Workflow Advisor lists the number of networks present in the model and the number of algebraic and differential variables for each network. The Advisor uses the Solver Configuration block to identify each unique network in your model.

Passed
Model 'sschdlexSolarInverterPartitionedNetworkExample' is switched linear.

Number of Simscape networks present in the model: 2

Details related to the Simscape network
sschdlexSolarInverterPartitionedNetworkExample/Network/Solver Configuration

Details
Number of Discrete Variables: 6

Number of Differential Variables: 3

Source	Value
Network.Capacitor.vc	Capacitor voltage
Network.Capacitor2.vc	Capacitor voltage
Network.L1.i_L	Inductor current

Number of Algebraic Variables: 3

Source	Value
Network.Diode.i	Current
Network.IGBT4.ideal_switch.i	i
Network.L1.v	Voltage

Details related to the Simscape network
sschdlexSolarInverterPartitionedNetworkExample/Network/Solver Configuration4

Details
Number of Discrete Variables: 12

3. Run the **Extract discrete equations** task.

This task displays Simulation stop time, Number of solver iterations, number of states, inputs, outputs, modes, differential variables, and state-space representation for each Simscape network.

Details related to the Simscape network
sschdlexSolarInverterPartitionedNetworkExample/Network/Solver Configuration

- Number of states: 6
- Number of inputs: 3
- Number of outputs: 4
- Number of modes: 5
- Number of differential variables: 3

Summary of the state-space representation:

Parameter	Parameter size
A	6 x 6 x 5
B	6 x 3 x 5
F0	6 x 1 x 5
C	4 x 6 x 1
D	4 x 3 x 1
Y0	4 x 1 x 1

Details related to the Simscape network
sschdlexSolarInverterPartitionedNetworkExample/Network/Solver Configuration4

- Number of states: 12
- Number of inputs: 5
- Number of outputs: 3
- Number of modes: 28
- Number of differential variables: 3

Summary of the state-space representation:

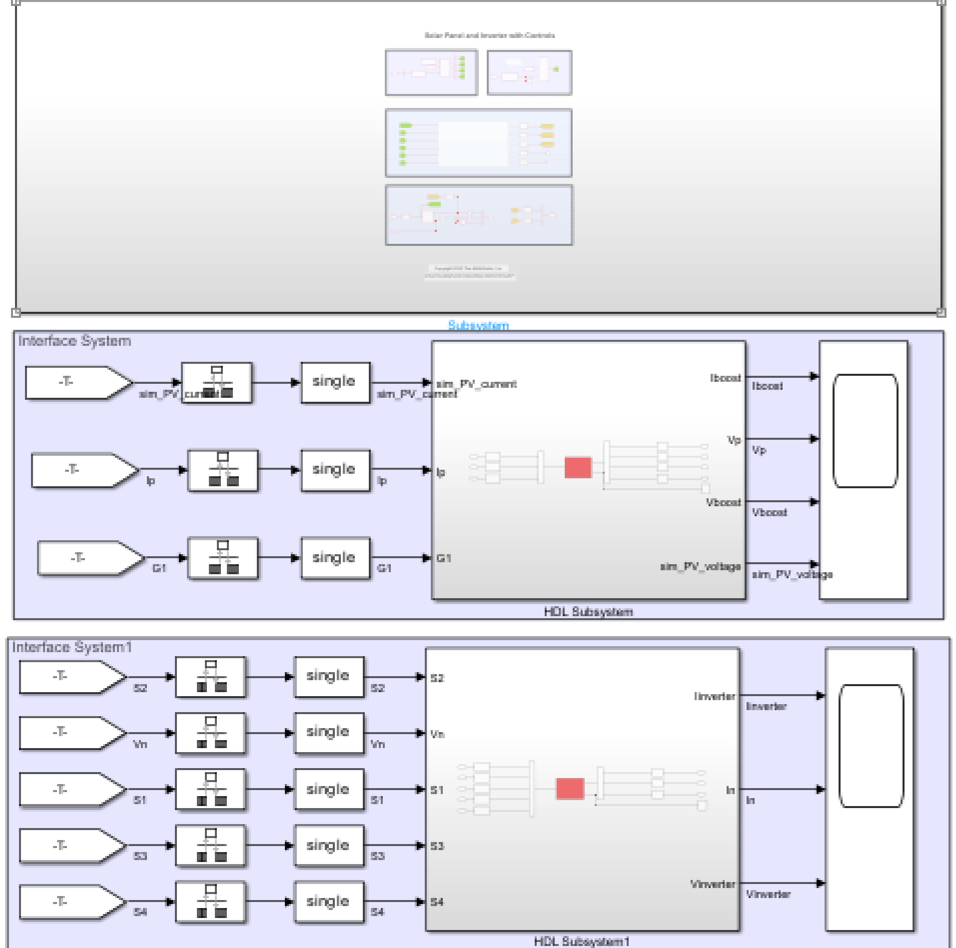
Parameter	Parameter size
A	12 x 12 x 28
B	12 x 5 x 28
F0	12 x 1 x 28
C	3 x 12 x 1
D	3 x 5 x 1
Y0	3 x 1 x 1

The state-space representation now uses fewer modes. The number of modes is 28 for the boost converter and 5 for the full bridge inverter, which results in a total number of 33 modes. The reduction in the number of modes saves area of the HDL implementation model on the target device.

Generate HDL Implementation Model and View Resource Consumption

1. Run the **Generate implementation model** task. After the task passes, you see a link to the HDL implementation model `gmStateSpaceHDL_sschdlexSolarInverterPartit`. Click the link to open the HDL implementation model or enter:

```
open_system('gmStateSpaceHDL_sschdlexSolarInverterPartit')
```



The model contains two HDL Subsystems. The HDL Subsystem block models the state-space equations for the boost converter. The HDL Subsystem1 block models the state-space equations for the full bridge inverter.

2. Enable generation of the resource utilization report.

```
hdlset_param('gmStateSpaceHDL_sschdlexSolarInverterPartit', 'ResourceReport', 'on')
```

3. Run the `makehdl` function to generate code. To generate HDL code for both HDL Subsystem blocks, you can place HDL Subsystem and HDL Subsystem1 blocks inside another top level subsystem and name this subsystem as HDL_DUT. Then generate HDL code.

```
makehdl('gmStateSpaceHDL_sschdlexSolarInverterPartit/HDL_DUT')
```

4. As you generate HDL code, open the Code Generation Report. The resource utilization report indicates the amount of adders, multipliers, and registers that might be consumed on the target FPGA device.

Summary

Multipliers	85
Adders/Subtractors	1519
Registers	8554
Total 1-Bit Registers	76790
RAMs	0
Multiplexers	13793
I/O Bits	484
Static Shift operators	267
Dynamic Shift operators	194

The overall resource consumption of the two networks is significantly less than the resource consumption of a single, large network. To learn about the resource consumption of the single solar power inverter network, see "Partition Simscape Models Containing a Large Network into Multiple Smaller Networks" on page 30-61.

See Also

Functions
`checkhdl` | `makehdl` | `sschdladvisor`

More About

- "Generate HDL Code for Simscape Models" on page 30-13
- "Simscape HDL Workflow Advisor Tasks" on page 31-2
- "Simscape HDL Workflow Advisor Tips and Guidelines" on page 31-7
- "Validate HDL Implementation Model to Simscape Algorithm" on page 30-89

Replace Piecewise-Constant Resistor with Switched Linear Components

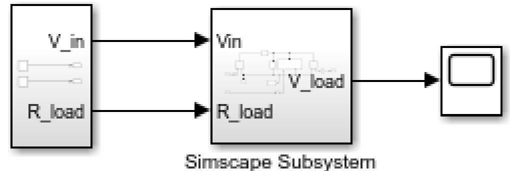
This example shows how to convert a Simscape™ model that has a `Piecewise-Constant Resistor` (nonlinear component) into a switched linear model, making it compatible with Simscape Hardware-in-the-Loop (HIL) workflow.

Introduction

Simscape HIL workflow supports conversion of Simscape models to functionally equivalent Simulink® models that are compatible for HDL code generation. In this example, the model uses a `Piecewise-Constant Resistor` that is event based. Events are not supported by the Simscape HIL workflow. You can convert such Simscape models to a switched linear model and make it compatible for HDL code generation. For more information, see “Generate HDL Code for Simscape Models” on page 30-13.

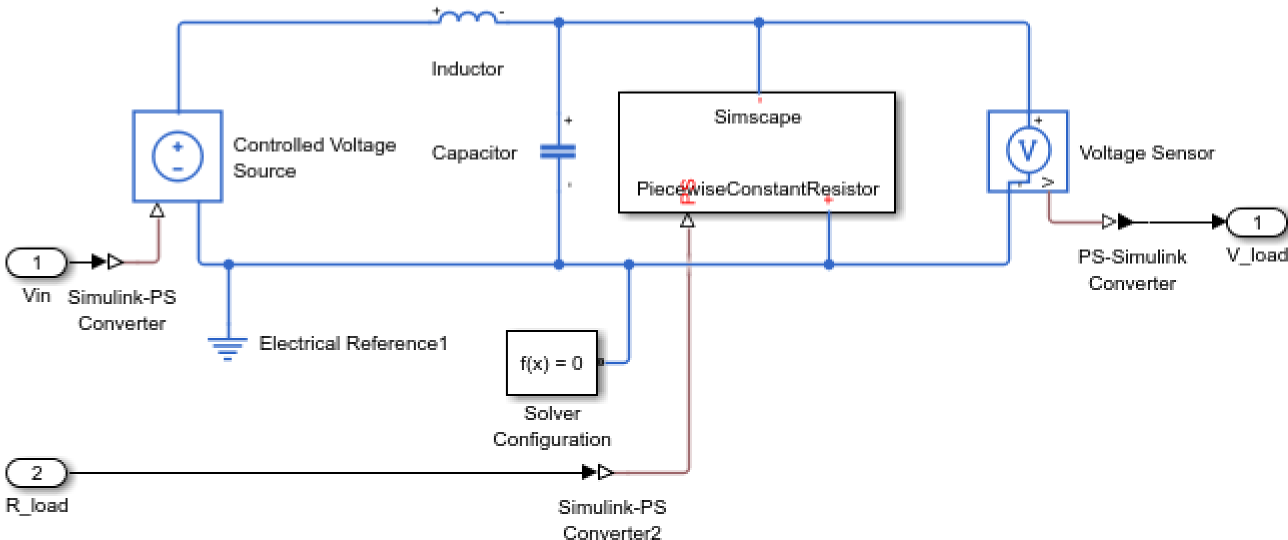
Open the Simscape model from the MATLAB® command prompt.

```
NonlinearModel = 'sschdlexVariableResistorExample';
open_system(NonlinearModel)
set_param(NonlinearModel, 'SimulationCommand', 'update');
```



Open the Simscape subsystem.

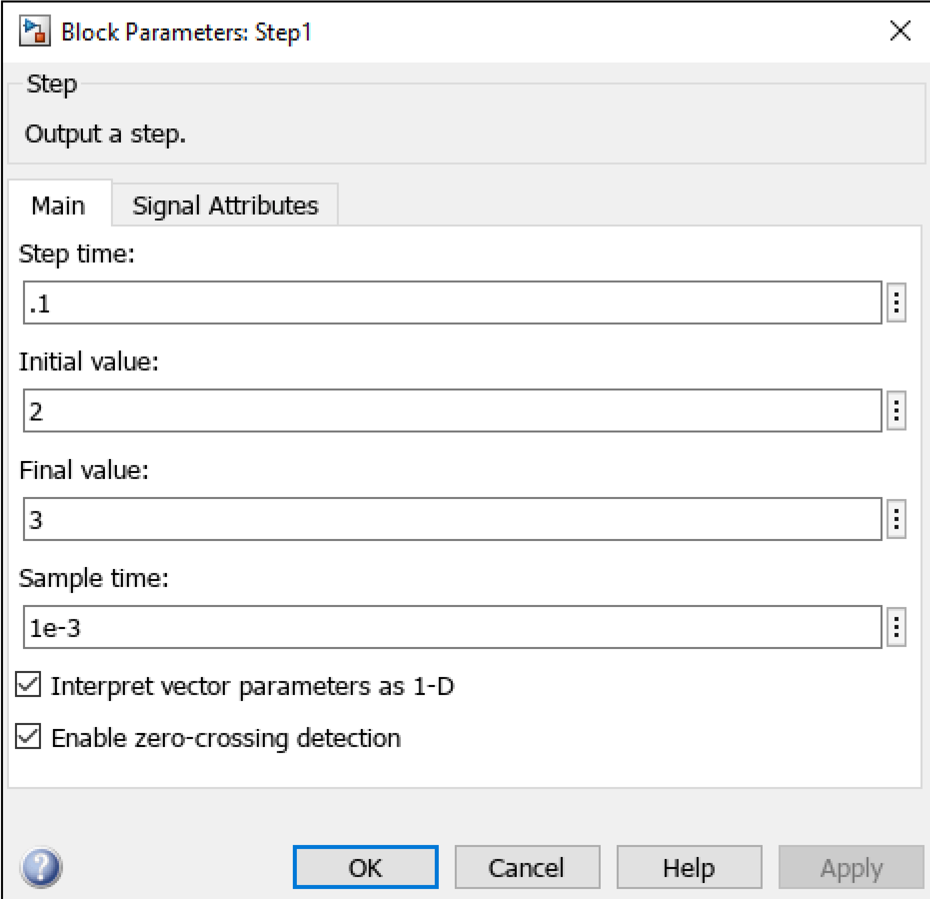
```
open_system([NonlinearModel,'/Simscape Subsystem'])
```



This model is an RLC circuit with a `Piecewise-Constant Resistor` acting as a "Load" resistance. For the `Piecewise-Constant Resistor`, the relationship between voltage V and current I is $V = I * R$, where R is the numerical value presented at the physical signal port R.

To ensure a positive value for the resistance, any value below `1e-6` is replaced by `1e-6`. This resistor is piecewise-constant because the resistance only changes when the input value differs from the current resistance value by more than the set tolerance. Thus, a continuously changing input is converted to a discrete set of resistances.

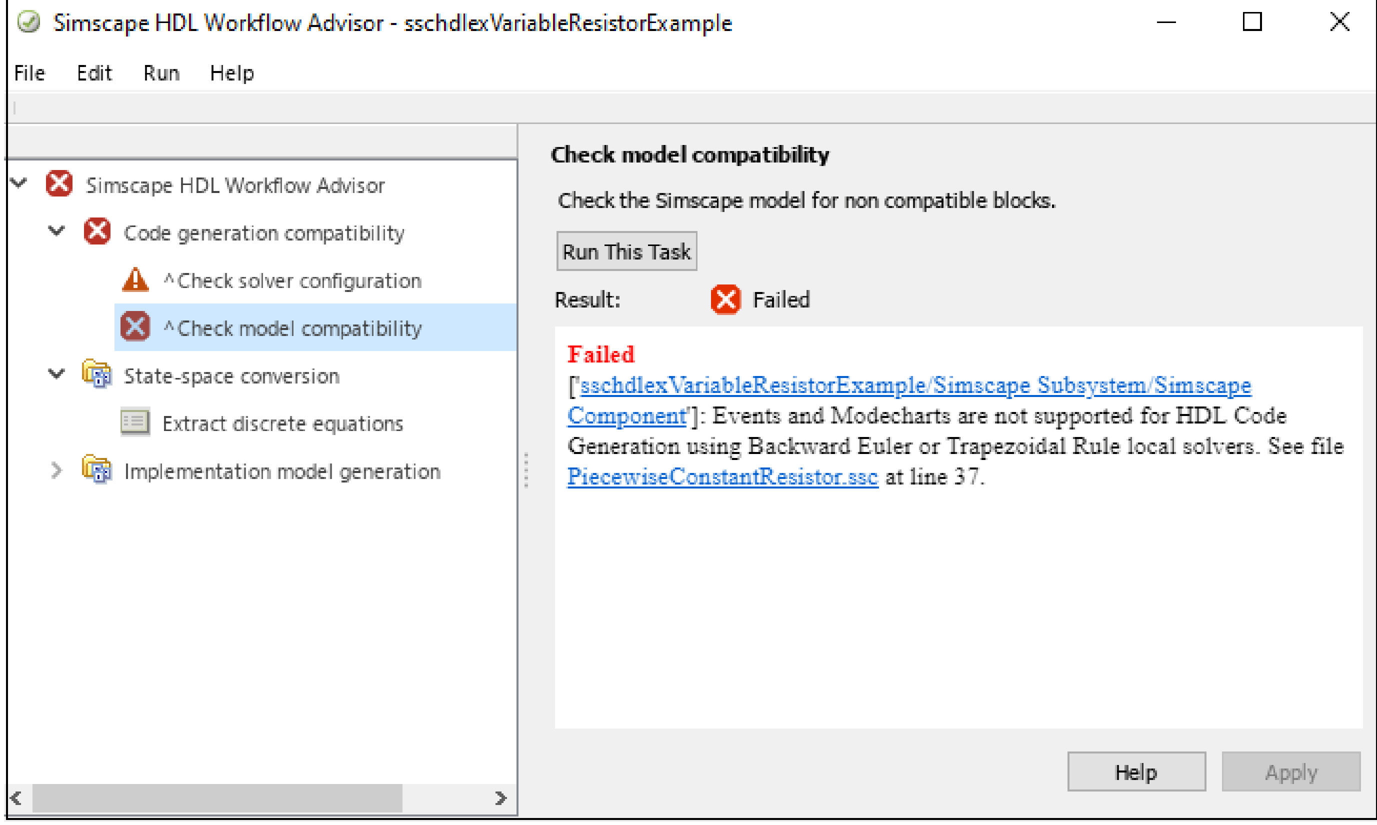
In this model, the signal going into the `Piecewise-Constant Resistor` is a step function that changes from 2 to 3 at `t = 0.1` thus changing the load resistance from 2 Ω to 3 Ω.



Open the Simscape HDL Workflow Advisor using `sschdladvisor` function.

```
sschdladvisor(NonlinearModel)
```

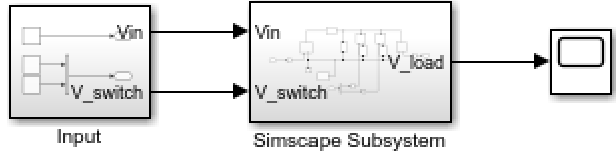
In the Simscape HDL Workflow Advisor, right-click the **Code generation compatibility** > **Check model compatibility** task and select **Run to Selected Task**. This task fails because of the presence of the `Piecewise-Constant Resistor`.



Replace Piecewise-Constant Resistor with Switches and Constant Resistors

To convert this model to an equivalent switched linear model, replace the Piecewise-Constant Resistor with a set of switches and resistors for each desired value. Open the switched linear version of the model.

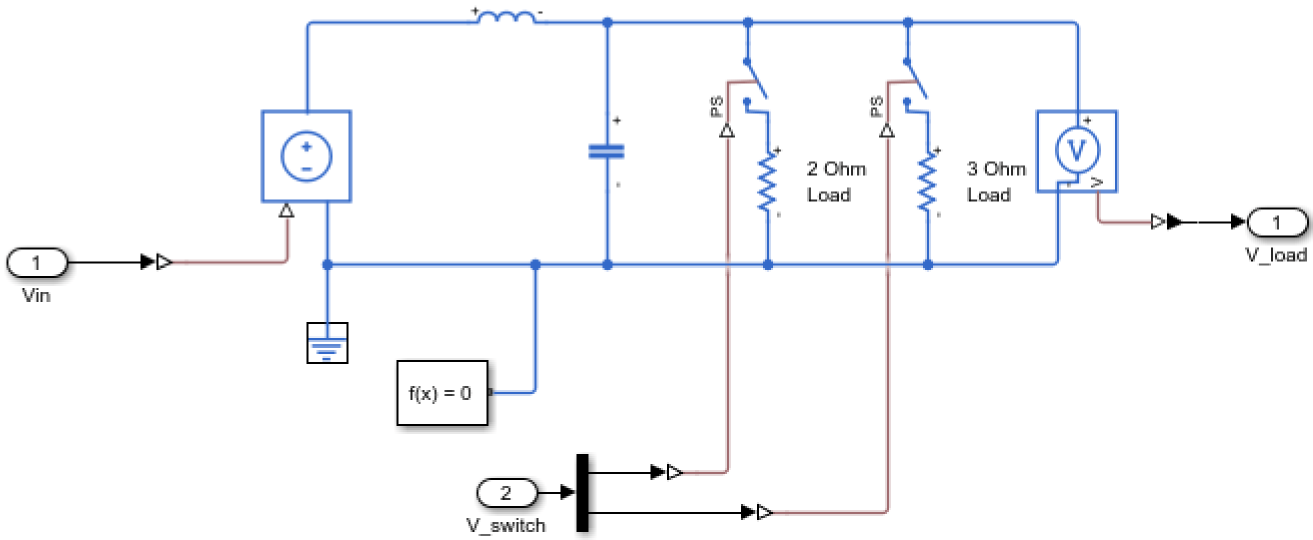
```
SwitchedLinearModel = 'sschdlexVariableResistorSwitchedLinearExample';
open_system(SwitchedLinearModel)
set_param(SwitchedLinearModel, 'SimulationCommand', 'update');
```



Copyright 2019-2022 The MathWorks, Inc.

Instead of a `Piecewise-Constant Resistor`, the model uses a resistor and a switch for each desired resistance. In particular, it uses a resistance of 2 Ω and the other with a resistance of 3 Ω. By closing and opening the switches at `t = 0.1`, the model changes the load resistance from 2 Ω to 3 Ω.

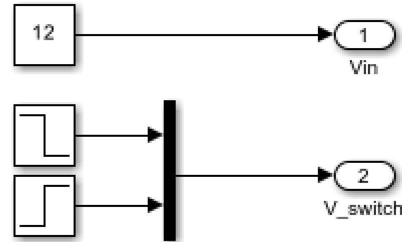
```
open_system([SwitchedLinearModel,'/Simscape Subsystem'])
```



Controlling the Switches

Open the control signals for the switches.

```
open_system([SwitchedLinearModel,'/Input'])
```



The input signal `V_switch` contains two step functions, one that opens the switch in series with the 2 Ω resistor at `t = 0.1` and one that closes the switch in series with the 3 Ω resistor at the same time.

Generate HDL Implementation Model

Use the default **Solver Configuration** settings for the switched linear model:

- Solver type: `Backward Euler`
- Sample time: `1e-3`

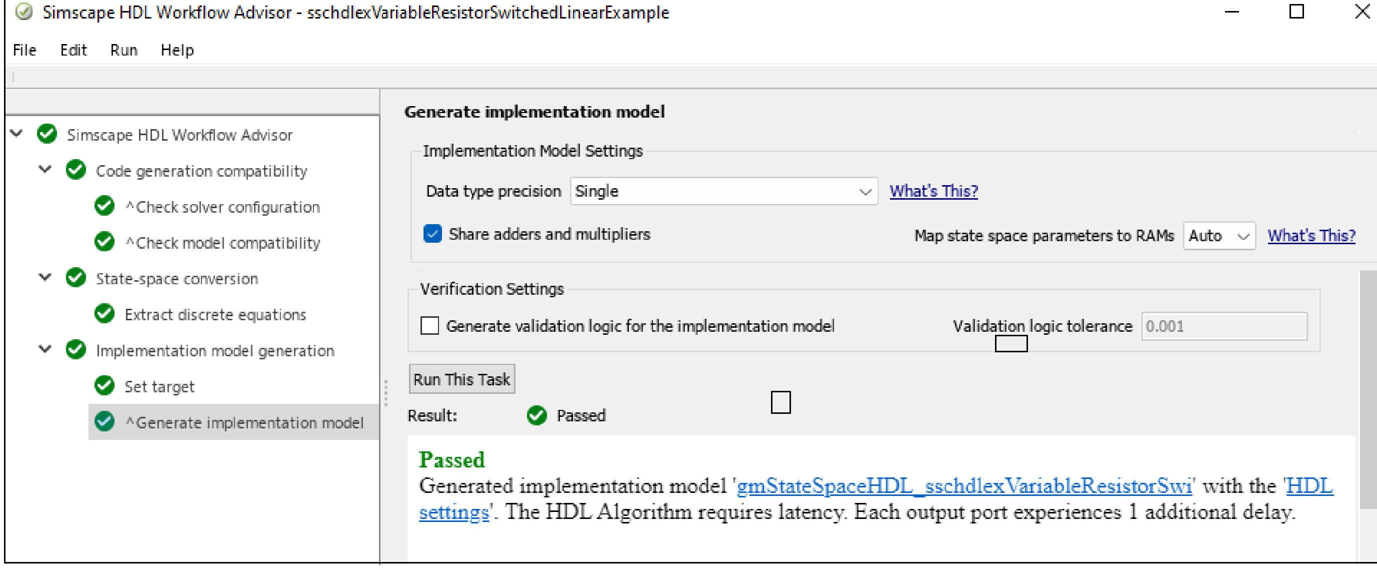
In the model window, you can view the model statistics. Select the **Debug** tab and click **Simscape > Statistics Viewer**. This opens the Simscape Statistics window for `sschdlexVariableResistorExample` model.

Open the Simscape HDL Workflow Advisor for the switched linear model.

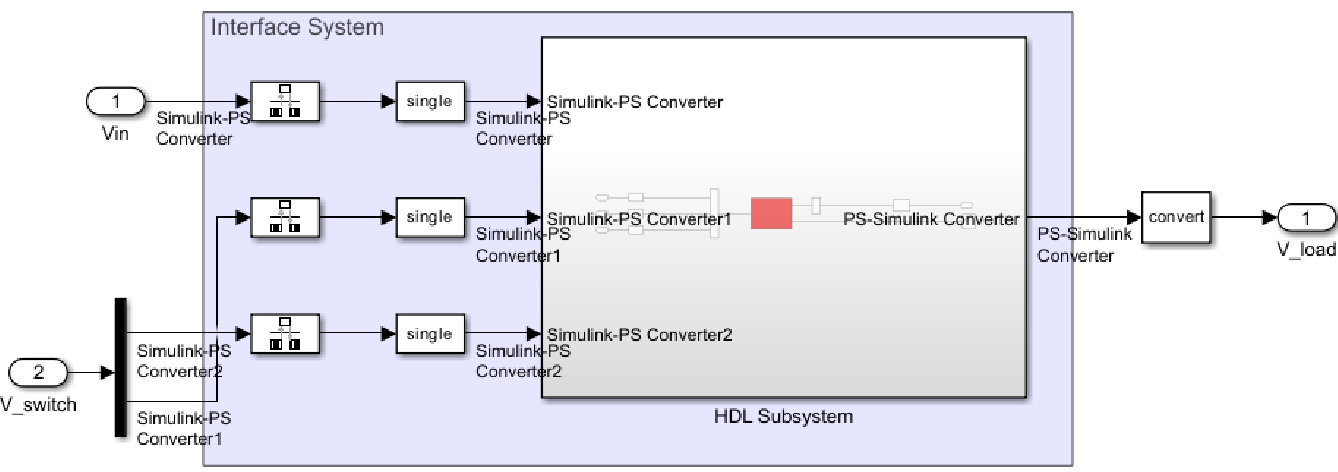
```
sschdladvisor(SwitchedLinearModel)
```

To generate the HDL implementation model, right-click the **Implementation model generation > Generate implementation model** task, and then select **Run to Selected Task**.

After the last task passes, it creates a link to the HDL implementation model `gmStateSpaceHDL_sschdlexVariableResistorSwi`.



Click on this link to open the generated implementation model.



In the right pane, the **Extract discrete equations** task output displays the number of states, inputs, outputs, modes, differential variables, and state-space representation for the Simscape network.

Details related to the Simscape network
sschdlexVariableResistorSwitchedLinearExample/Simscape Subsystem/Solver Configuration

- Number of states: 5
- Number of inputs: 3
- Number of outputs: 1
- Number of modes: 2
- Number of differential variables: 2

Summary of the state-space representation:

Parameter	Parameter size
A	5 x 5 x 2
B	5 x 3 x 2
F0	5 x 1 x 2
C	1 x 5 x 1
D	1 x 3 x 1
Y0	1 x 1 x 1

Generate HDL Code

To set up model parameters for HDL code generation, run the `hdlsetup` function.

```
hdlsetup('gmStateSpaceHDL_sschdlexVariableResistorSwi')
```

Save the parameters for the HDL implementation model.

```
hdlsaveparams('gmStateSpaceHDL_sschdlexVariableResistorSwi')
```

Enable generation of the resource utilization report.

```
hdlset_param('gmStateSpaceHDL_sschdlexVariableResistorSwi', 'ResourceReport', 'on')
hdlset_param('gmStateSpaceHDL_sschdlexVariableResistorSwi', 'MaskParameterAsGeneric', 'off');
```

Generate HDL code for the implementation model.

```
makehdl('gmStateSpaceHDL_sschdlexVariableResistorSwi/Simscape Subsystem/HDL Subsystem');
```

When you generate code, HDL Coder creates a code generation report. The resource utilization report in the **High-level Resource Report** indicates the number of adders, multipliers, and registers that might be consumed on the target FPGA device.

Summary

Multipliers	17
Adders/Subtractors	227
Registers	1237
Total 1-Bit Registers	10896
RAMs	0
Multiplexers	1777
I/O Bits	132
Static Shift operators	45
Dynamic Shift operators	22

By changing the `Piecewise-Constant Resistor` to resistors that switch on and off, you change the model to a form that is compatible with the Simscape to HDL workflow.

See Also

Functions
`sschdladvisor` | `makehdl` | `hdlsetup` | `hdlsaveparams`

More About

- "Get Started with Simscape Hardware-in-the-Loop Workflow" on page 30-2
- "Generate HDL Code for Simscape Models" on page 30-13
- "Simscape HDL Workflow Advisor Tasks" on page 31-2
- "Simscape HDL Workflow Advisor Tips and Guidelines" on page 31-7

Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules

This example shows how to synthesize and generate FPGA bitstream from a Simscape™ half-wave rectifier model and download the bitstream to a Speedgoat® FPGA I/O 334-325K target for Hardware-in-the-Loop (HIL) implementation.

Hardware-in-the-Loop Workflow

1. Generate a HDL implementation model from the Simscape model by using the Simscape HDL Workflow Advisor. The HDL implementation model is a Simulink® model that replaces the Simscape algorithm with HDL-compatible blocks
2. Generate FPGA bitstream for the HDL implementation model by using the HDL Workflow Advisor
3. Download the bitstream to the Speedgoat FPGA I/O module by using the Simulink Real-Time Explorer for Hardware-in-the-Loop Simulation.

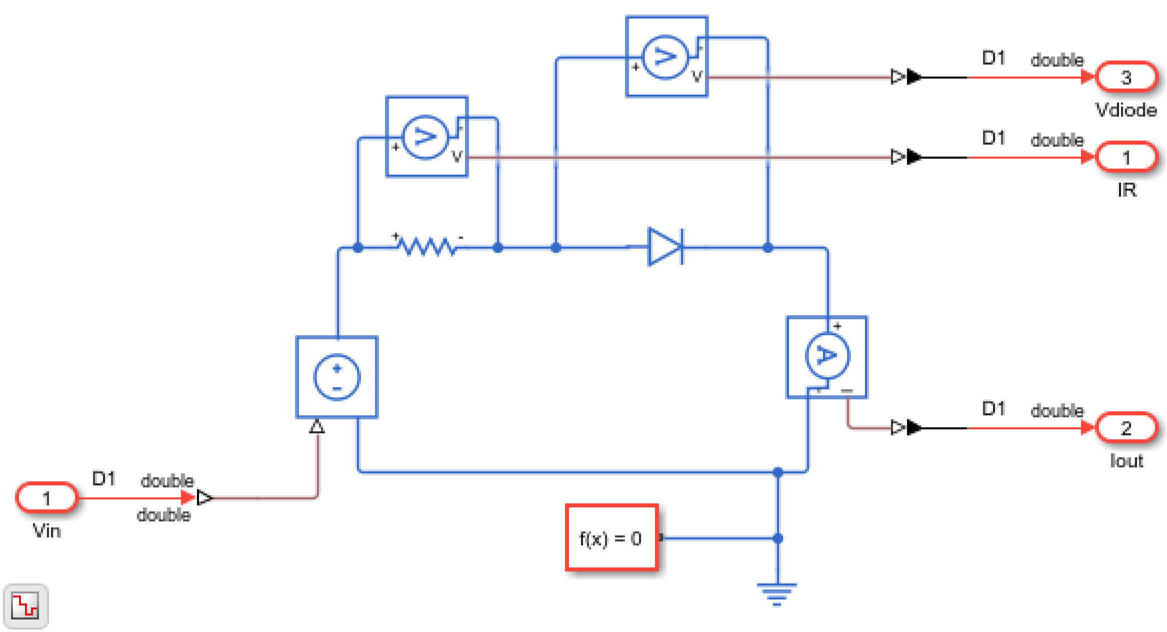
Half Wave Rectifier Model

Open the Simscape half wave rectifier model. In the MATLAB® command prompt, enter:

```
ModelName = 'sschdlexHalfWaveRectifierExample';
open_system(ModelName)
set_param(ModelName, 'SimulationCommand', 'update');
```



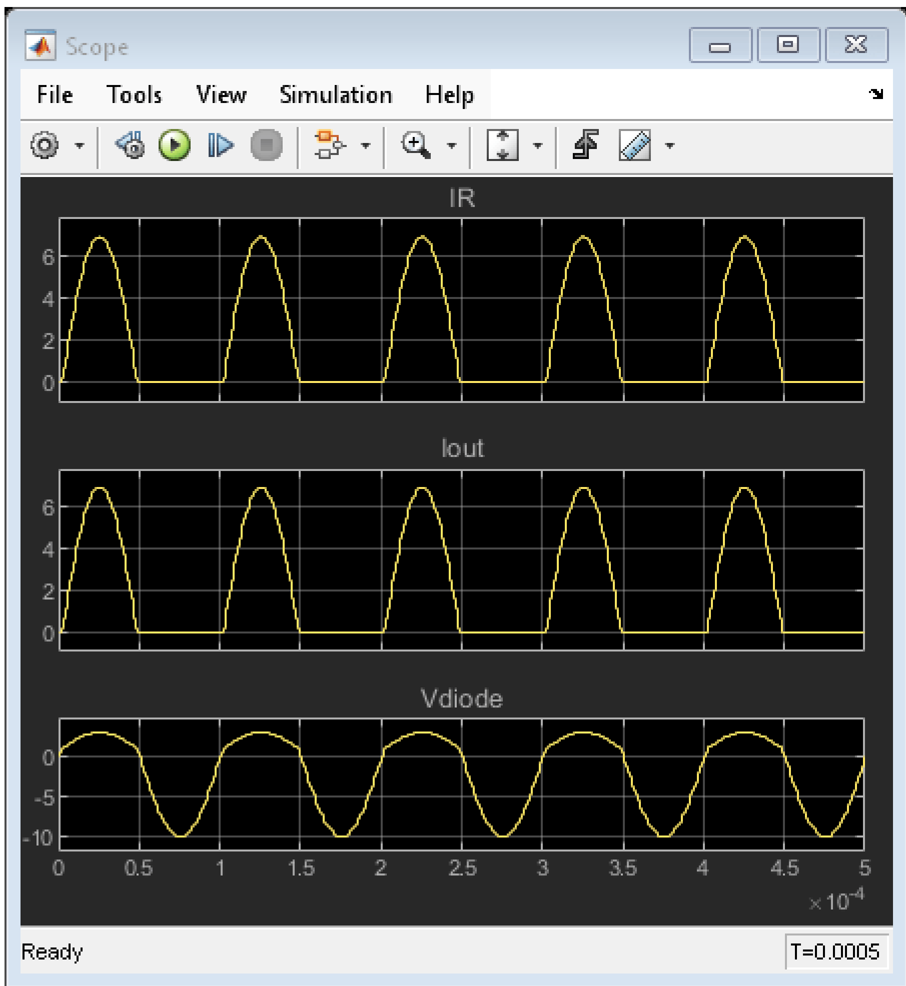
```
open_system([ModelName, '/Simscape_system'])
```



The half-wave rectifier consists of a Resistor, which is a linear block, and a Diode, which is a switched linear block. At the input and output port interfaces, the model has Simulink-PS Converter and PS-Simulink Converter blocks. The solver settings are configured for compatibility with Simscape HDL Workflow Advisor. If you open the Block Parameters dialog box for the Solver Configuration block, **Use local solver** is selected and `Backward Euler` is specified as the **Solver type**. See "Get Started with Simscape Hardware-in-the-Loop Workflow" on page 30-2.

To see the algorithm functionality, simulate the model.

```
sim(ModelName)
open_system([ModelName, '/Scope'])
```



2. Configure the Simscape Model for HDL compatibility by using the `hdlsetup` function:

```
hdlsetup('sschdlexHalfWaveRectifierExample')
```

Generate HDL Implementation Model

To generate the HDL implementation model:

1. Open the Simscape HDL Workflow Advisor:

```
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

2. To compare functionality of the HDL implementation model with the original Simscape algorithm, select the **Generate implementation model** step, and then select the **Generate validation logic for the implementation model** check box. Use a **Validation logic tolerance** of `0.001`. Right-click the **Generate implementation model** step and select **Run to Selected Task**.

The Advisor generates an HDL implementation model and a state-space validation model. To compare functionality of the HDL implementation model with the original Simscape algorithm, open and simulate the state-space validation model. The output of this model matches the original Simscape model. For a more systemic verification, see “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89.

See also “Simscape HDL Workflow Advisor Tasks” on page 31-2.

Setup and Configuration

The Speedgoat IO334-325K FPGA module uses Xilinx® Vivado® and *IP Core Generation* workflow infrastructure. Before you deploy the HDL implementation model on the Speedgoat I/O module:

1. **Install Xilinx Vivado and Setup Tool Path**

Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”. Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2022.1\bin\vivado.bat')
```

2. **Install Speedgoat I/O Blockset and Speedgoat - HDL Coder Integration Packages**

To install the Speedgoat I/O Blockset and the Speedgoat - HDL Coder Integration packages. Go to Speedgoat documentation online at www.speedgoat.com/knowledge-center.

3. **Setup I/O Module**

For real-time simulation, set up the I/O module. See Xilinx HDL Software for Speedgoat I/O Modules.

HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.
- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis and timing analysis.
- Deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

To open the HDL Workflow Advisor, use the `hdladvisor` function.

```
hdladvisor('gmStateSpaceHDL_sschdlexHalfWaveRectifierEx/Simscape_system/HDL Subsystem')
```

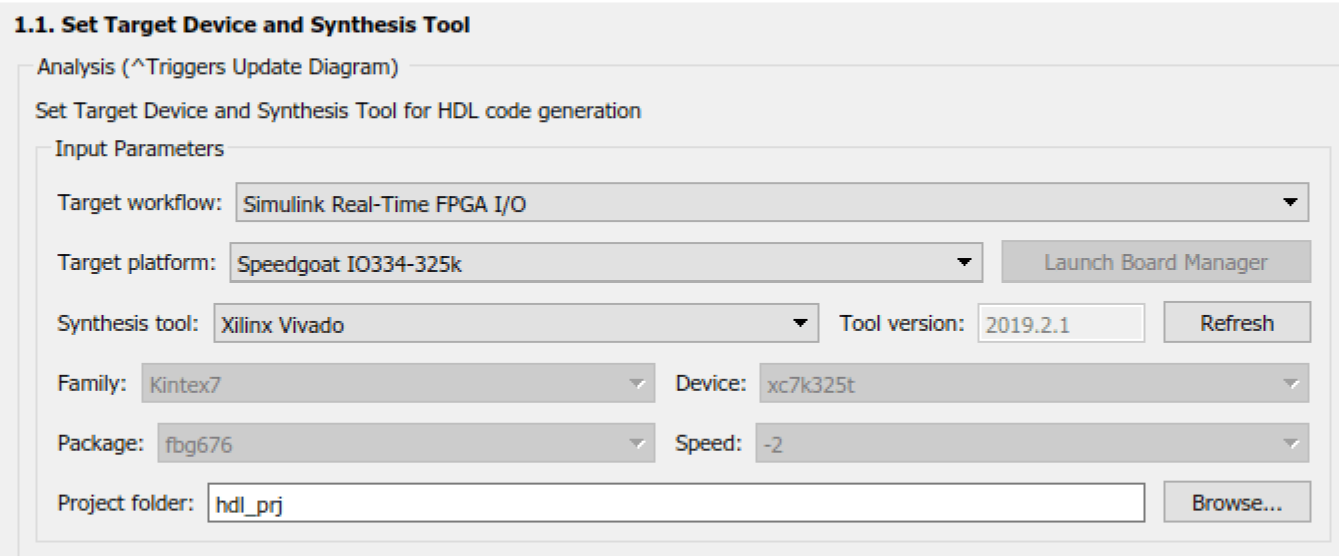
The left pane contains folders that represent a group of related tasks. Expanding the folders and selecting a task displays information about that task in the right pane. The right pane can contain simple controls for running the task to advanced parameters and option settings that control code and test bench generation. To learn more about each task, right-click that task, and select **What's This?**. See “Getting Started with the HDL Workflow Advisor” on page 29-5.

Generate FPGA Bitstream for Speedgoat Target Computer

1. Open the HDL implementation model, and then open the HDL Workflow Advisor for the implementation model.

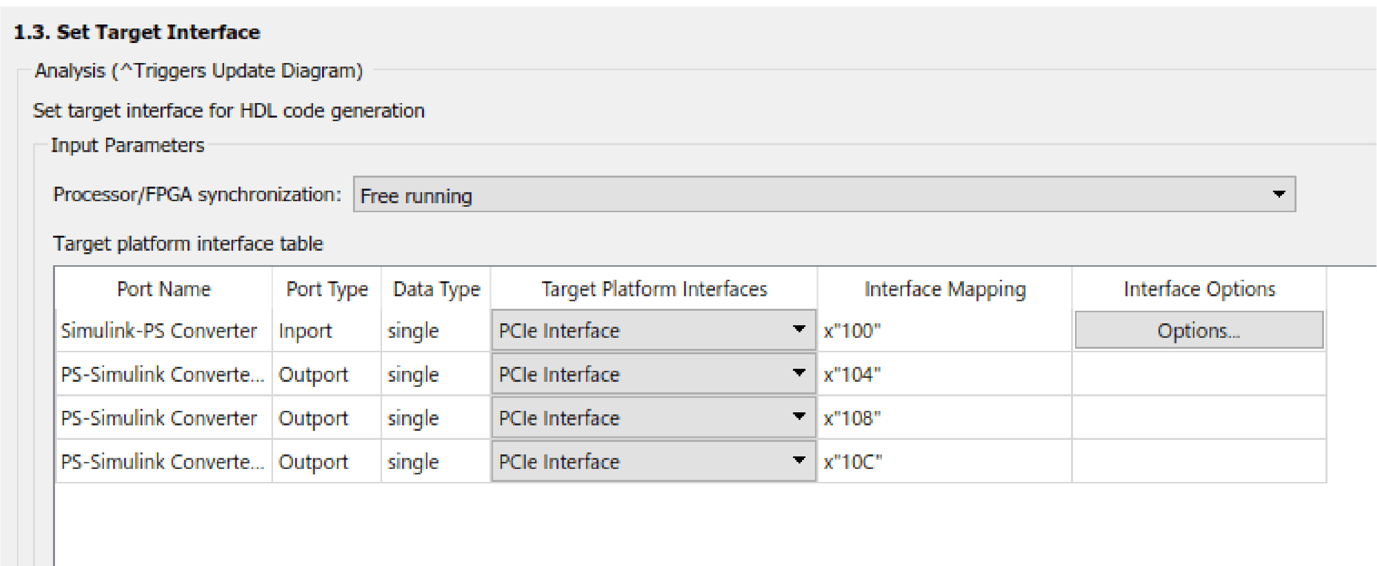
```
open_system('gmStateSpaceHDL_sschdlexHalfWaveRectifierEx')
hdladvisor('gmStateSpaceHDL_sschdlexHalfWaveRectifierEx/HDL Subsystem')
```

2. In **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Simulink Real-Time FPGA I/O` and **Target platform** as `Speedgoat IO334-325K`.



3. In the **Set Target Reference Design** task, select a value of `x4` for the parameter `PCIe lanes`, and select **Run This Task**.

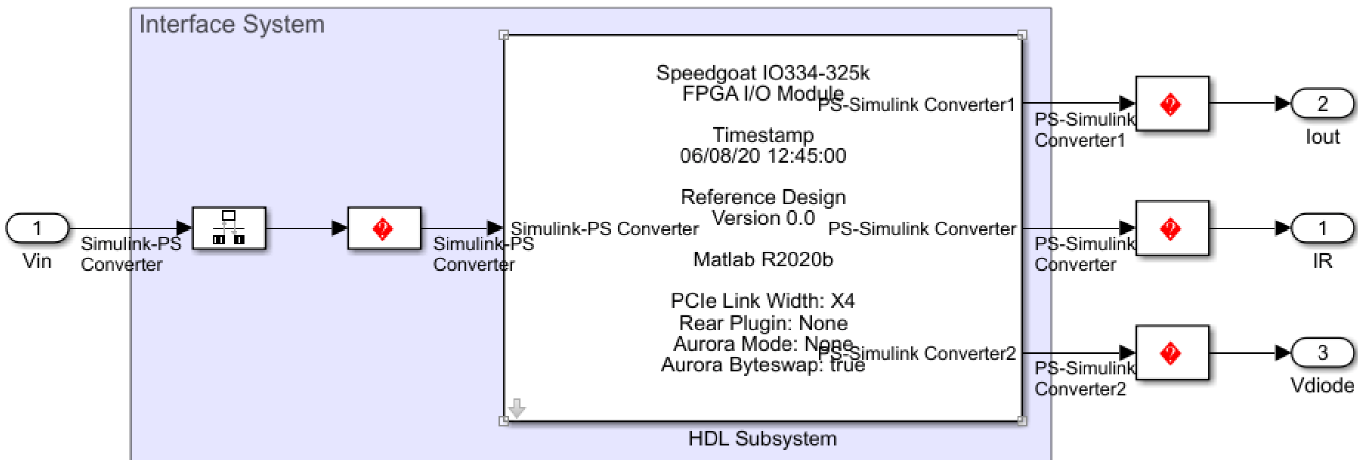
4. In **Set Target Interface** task, map the input and output `single` data type ports to `PCIe Interface` and select **Run This Task**.



5. In the **Set Target Frequency** task, set the **Target Frequency (MHz)** as `100`.

6. Right-click the **Generate Simulink Real-Time Interface** task and select **Run to Selected Task** to generate the HDL IP core, FPGA bitstream, and download the bitstream to the IO334 I/O module in the Speedgoat target computer.

A Simulink Real-Time Interface model is generated, and named as **gm_gmStateSpaceHDL_sschdlexHalfWaveRectifierEx_slrt**.



For rapid prototyping, you can export the Workflow Advisor settings to a script. The script is a MATLAB file that you run from the command line. You can modify and run the script, or import the settings into the HDL Workflow Advisor User Interface. To save the workflow, in the HDL Workflow Advisor User Interface, select **File > Export to Script**. Save the file as `hdlworkflow_slrt_IO334.m`.

To import this file, in the HDL Workflow Advisor User Interface, select **File > Import from Script**. In the Import Workflow Configuration dialog box, select the `hdlworkflow_slrt_IO334.m` file. The HDL Workflow Advisor updates the tasks according to the imported script. See “Run HDL Workflow with a Script” on page 29-47.

Deploy Bitstream to Speedgoat IO334-325k Target

1. **Connect Development Computer to Target**

Connect the development computer to the target by using a cross-over network cable. The default IP address for the Speedgoat target computer is `192.168.7.5`. Set the IP address of the communication link between the development computer and target computer to a value `192.168.7.2` because the communication link must be in the same network.

2. **Setup and Configure Simulink Real-Time Explorer**

You download the bitstream by using the Simulink Real-Time Explorer. To open the Simulink Real-Time Explorer, enter the command `slrtExplorer`. Alternatively, you can open the Explorer from the **Real-Time** tab of the Simulink Toolstrip.

```
slrtExplorer
```

The Simulink Editor displays the **Real-Time** tab for models that are configured for the `speedgoat.tlc` code generation target.

a. In Simulink Real-Time Explorer, on the Target Configuration tab, configure settings on the development computer:

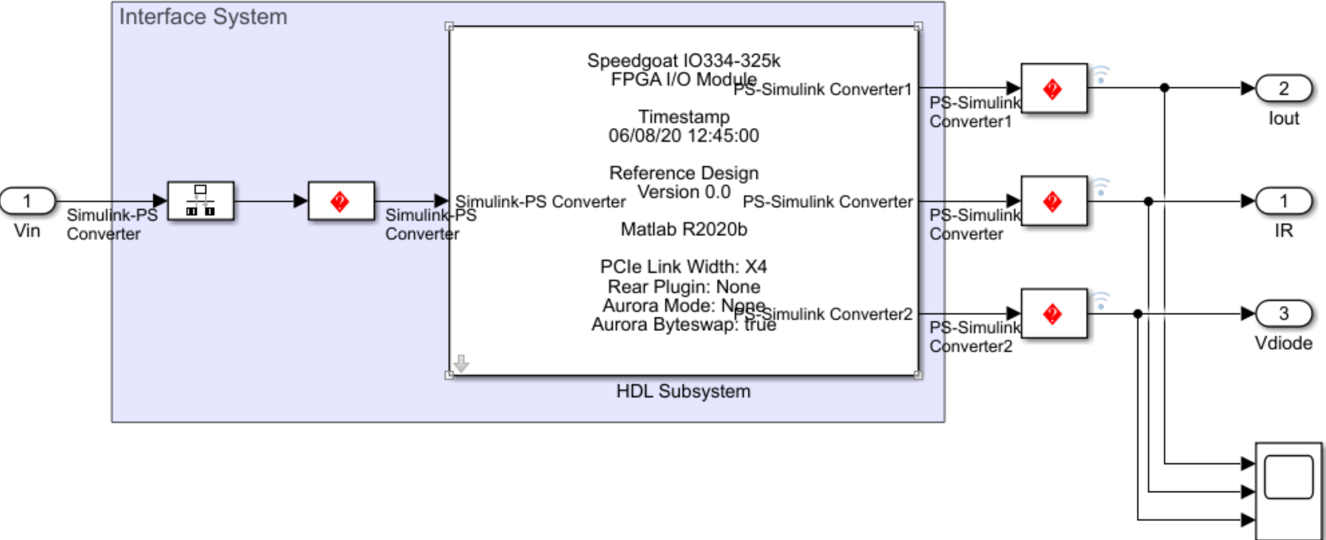
- Set **IP Address** as `192.168.7.5`, or set as needed for a custom target computer IP address.

- Set **Name** as `TargetPC1`, or set as needed for a custom target computer name.

b. If you change the settings on the development computer, click the `Change IP Address` button to apply corresponding changes on the target computer.

3. **Create Real-Time Application**

Open the Simulink Real-Time Interface model. Add a Scope block to the model and connect it to the outputs. Log the output signals to view the simulation results on the Simulation Data Inspector.



4. **Build and Run Real-Time Application**

Click the **Run on Target** button on the **Real-Time** tab to compile and download the model onto Speedgoat IO334-325k target.

Observe the output simulation results on the Simulation Data Inspector. The simulation results of the downloaded model match the original Simscape model simulation.

Validate HDL Implementation Model to Simscape Algorithm

If you design your algorithm by using Simscape switched linear blocks, you can run the Simscape HDL Workflow Advisor to generate an HDL implementation model. The HDL implementation model represents the Simscape algorithm by using Simulink blocks that are compatible for HDL code generation.

Before you prototype the implementation model on an FPGA or target Speedgoat FPGA I/O modules, you can verify the functionality of your design in the Simulink modeling environment. To verify the functionality, specify insertion of validation logic in the HDL implementation model when you run the Simscape HDL Workflow Advisor. This logic verifies whether the numeric results of the HDL implementation model match the original Simscape algorithm.

In some cases, there can be a mismatch in simulation results between the Simscape algorithm and the corresponding HDL implementation. Such mismatches generate warnings or assertions when you simulate the implementation model. To resolve the warnings, use a combination of various settings in the **Generate implementation model** task as illustrated below.

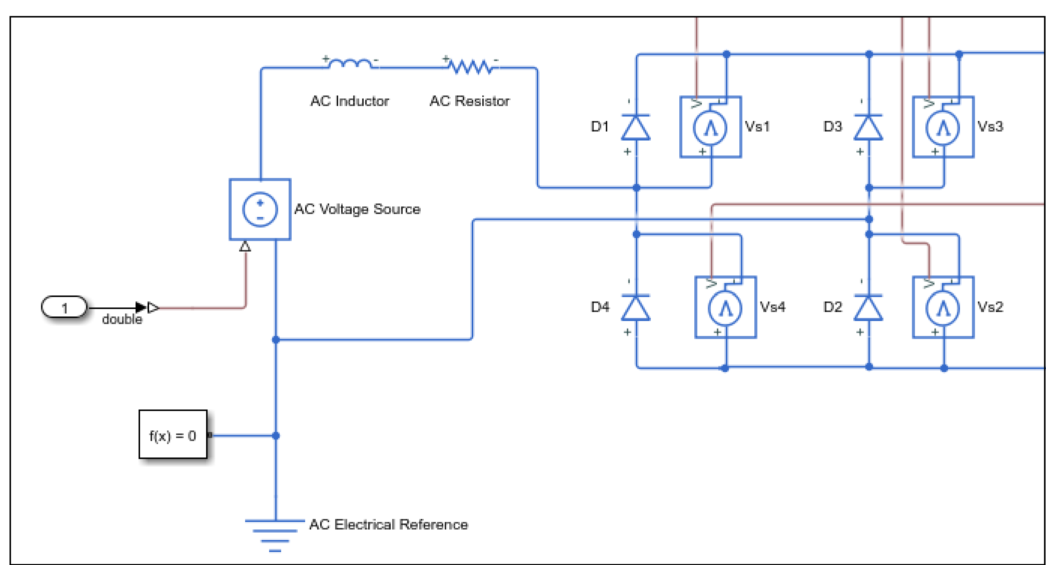
Bridge Rectifier Model

This example uses the bridge rectifier model to illustrate how to generate an implementation model with validation logic inserted in the model, and how you can resolve any assertions that may be generated when you simulate the implementation model.

1. Open the bridge rectifier model. In the MATLAB Command Window, enter:

```
openExample('plantdeployment/BridgeRectifierModelExample','supportingFile','sschdlexBridgeRec
open_system('sschdlexBridgeRectifierExample/Simscape_system')
```

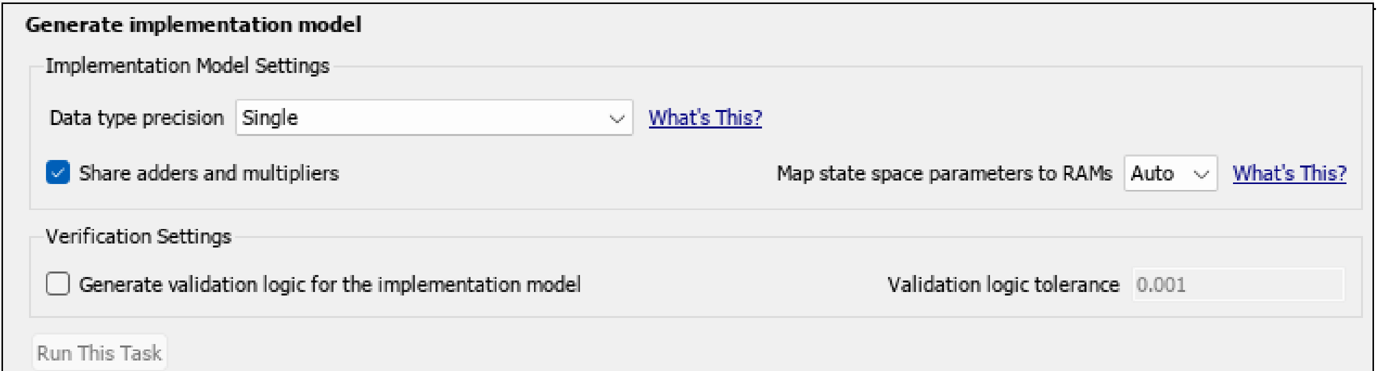
Inside the `Simscape_system`, you see four diodes arranged in a bridge configuration. For both positive and negative input values, this configuration provides a positive, rectified output.



2. Open the Simscape HDL Workflow Advisor for your model:

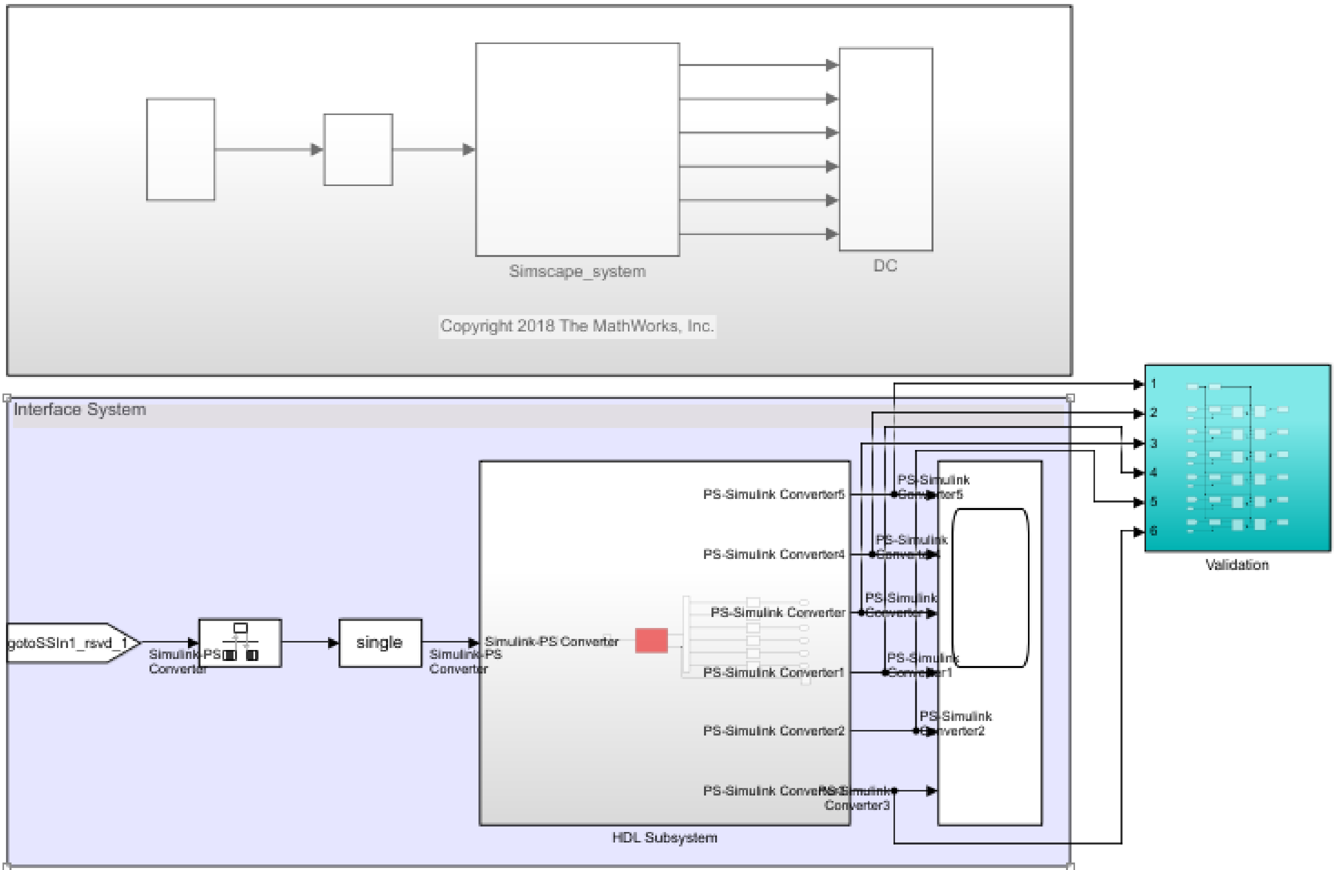
```
openExample('plantdeployment/BridgeRectifierModelExample','supportingFile','sschdlexBridgeRec
sschdladvisor('sschdlexBridgeRectifierExample')
```

3. Right-click the **State-space conversion** task and select **Run to Selected Task**.
4. In the **Generate implementation model** task, select the **Generate validation logic for the implementation model** check box. Leave the other options with their default values and select **Run This Task**.



After running this task, keep the UI window for this task open. If simulating the HDL implementation model generates warnings, you modify the settings in the **Generate implementation model** task and then rerun this task. You do not have to modify or rerun other tasks.

5. Click the link to open the HDL implementation model. You see a `Validation` Subsystem that compares the simulation results of the Simscape model to the HDL implementation model. Simulate the implementation model.



You see that simulating the model generates multiple assertions indicating a mismatch in the simulation results. If you open the Diagnostic Viewer, you see this message:

```
Assertion detected in 'gmStateSpaceHDL_BridgeRectifier_HDL_SimMismatch/
Validation/Check Static Range1' at time 0.04186 [4982 similar]
```

The message indicates that the Simscape algorithm does not match the equivalent HDL implementation. To resolve the validation mismatch, you can modify various settings in the **Generate implementation model** task until the HDL implementation model matches the Simscape algorithm. In most cases, to resolve the numeric mismatch, you may want to use a combination of these settings.

Increase Validation Logic Tolerance

Conversion of a Simscape algorithm to an equivalent HDL implementation leads to rounding errors. The default tolerance value is relatively small and can be difficult to achieve especially with single-precision data types in the HDL implementation model. To resolve the mismatch:

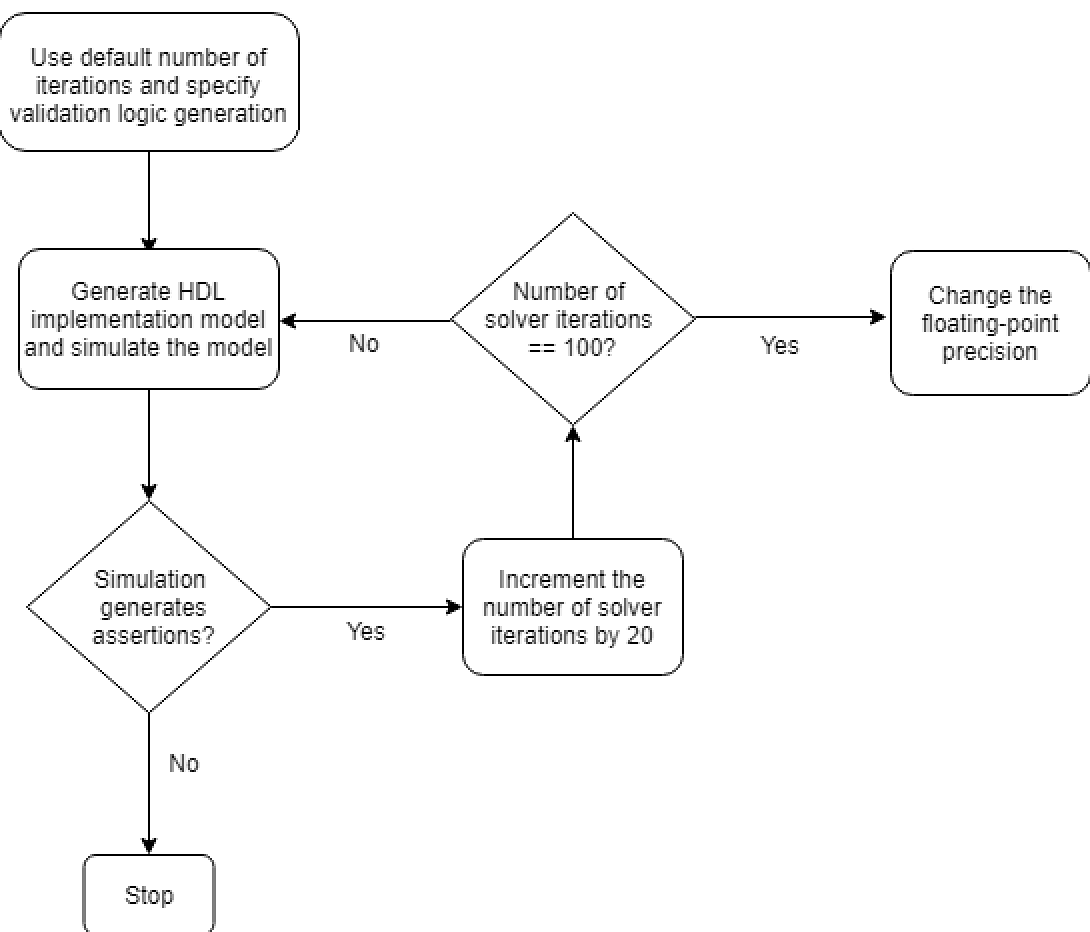
1. Start by increasing the **Validation logic tolerance** to an initial value such as `1e-4`.
2. Select **Generate validation logic for the implementation model** and run the task to generate the HDL implementation model that includes validation logic.
3. Simulate the model and check whether the simulation displays assertions in the Diagnostic Viewer. If the simulation results produce warnings, proceed to the next step to increase the number of solver iterations.

Increase Number of Solver Iterations

For each mode in the physical system, the switched linear workflow arrives at a state-space representation. The solver method is iterative and performs multiple computations to determine the correct mode for the next time step. After a certain number of iterations, the output value from the next time step becomes the same as the value from the previous time step. This consistency in the output value indicates the correct number of solver iterations.

The Advisor by default chooses an optimal value for the number of solver iterations. See “Using Number of Solver Iterations” on page 31-9. If increasing the tolerance value does not improve accuracy of the HDL implementation model, you can resolve the numeric mismatch by increasing the number of solver iterations.

When you increase the number of solver iterations, the code generator changes the sample time of the generated HDL implementation model. A large number of iterations can increase the simulation time significantly. See “Reducing Number of Solver Iterations” on page 30-98. This flowchart illustrates how to change the **Number of solver iterations**.

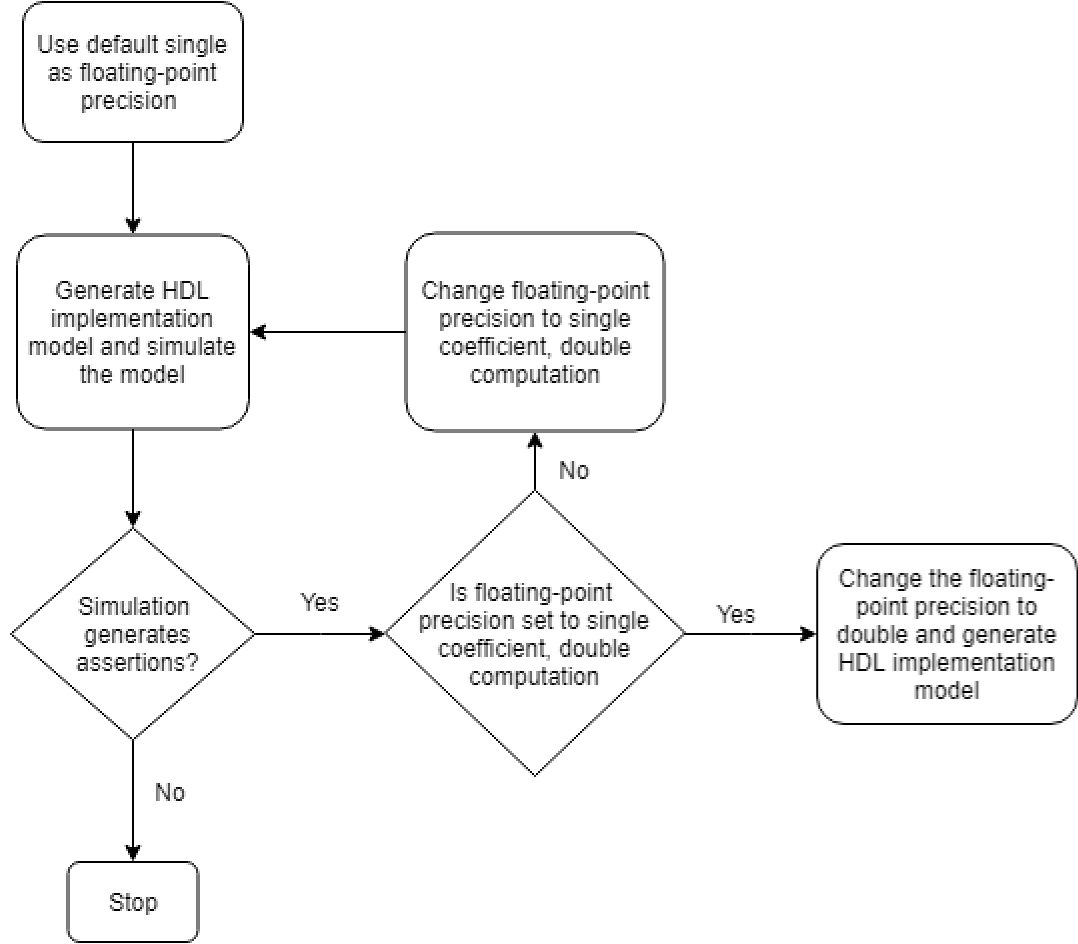


Use Larger Floating-Point Precision

You can use the **Data type precision** setting in the **Generate implementation model** task to specify the floating-point data type you want to use for the algorithm inside the `HDL Subsystem`. Specify whether you want to store the matrix coefficients in `single` or `double` data types and whether to use `single` or `double` when performing the computations.

Data type Precision	Description
`Double`	Using `double` floating-point precision increases the numerical accuracy of the generated model and the maximum achievable target frequency. However, the area consumption and pipeline latency are also increased.
`Single`	This is the default setting for data type precision.
`Single coefficient, double computation`	This mode offers a tradeoff between `Single` and `Double` modes of floating-point precision. To save memory usage, the coefficients that are stored in `single`. The matrix computations are then performed in `double` for improved accuracy.

This flowchart illustrates how to change the floating-point precision and improve the numeric accuracy of the generated HDL implementation model.



Note Double-precision operations have large latencies and require a large **Oversampling factor** to allocate sufficient delays for the floating-point operations, which reduces the sampling frequency. For a tradeoff between accuracy and precision, use `Single coefficient, double computation` as the **Data type precision**.

After specifying double data types, if the simulation results still produce warnings:

1. Proceed to the first step to further increase the validation logic tolerance. Use a tolerance value of `1e-03` and then simulate the model to see if the numeric accuracy requirements are met.
2. Increase the number of solver iterations if you still see warnings in the Diagnostic Viewer. Continue iterating between these steps till the HDL implementation model numerically matches the Simscape algorithm.

For the bridge rectifier model, to resolve the warnings, set the **Validation logic tolerance** to `1e-4` and specify the **Data type precision** as `double`. After you generate the implementation model with the validation logic, you see that simulating the model does not display warnings in the Diagnostic Viewer.

See Also

Functions
`sschdladvisor` | `simscape.findNonlinearBlocks`

More About

- “Generate HDL Code for Simscape Models” on page 30-13
- “Simscape HDL Workflow Advisor Tasks” on page 31-2
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 31-7
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 30-28

Improve FPGA Sampling Frequency of HDL Implementation Model Generated from Simscape Algorithm

If you design your algorithm by using Simscape blocks, you can run the Simscape HDL Workflow Advisor to generate an HDL implementation model. When you open the HDL implementation model, you see the HDL algorithm that models the state-space representation by using Simulink blocks that are compatible for HDL code generation. To learn more about the Simscape HDL Workflow Advisor, see “Simscape HDL Workflow Advisor Tasks” on page 31-2.

FPGA Sampling Frequency

FPGA sampling frequency is the frequency at which the models run on the hardware. When you generate HDL code and deploy the plant model onto an FPGA, you may want to improve the FPGA sampling frequency. The FPGA sampling frequency depends on these parameters:

- FPGA clock frequency
- Oversampling factor
- Number of solver iterations

$$FPGA\ sampling\ frequency = FPGA\ clock\ frequency / (Oversampling\ factor * Number\ of\ solver\ iterations)$$

FPGA sample time is the sample time at which the models run on the hardware. It is reciprocal of the FPGA sampling frequency.

$$FPGA\ sample\ time = \frac{1}{FPGA\ sampling\ frequency}$$

To improve the FPGA sampling frequency, you can find a tradeoff between improving the FPGA clock frequency, and balancing the oversampling factor and number of solver iterations. To learn more about how these parameters affect FPGA sampling frequency, see “Troubleshooting Real-Time Hardware Deployment Issues in Simscape Hardware-in-the-Loop Workflow” on page 32-2.

Simscape sampling frequency is the reciprocal of the Simscape sample time. In the Solver Configuration block dialog box, you can specify the value of Simscape sample time in the **Sample time** parameter text box under the **Use local solver** option.

Note To get accurate results on hardware, ensure that the Simscape sample time matches the FPGA sample time. For Simscape models containing a single Simscape network, the Simscape HDL Workflow Advisor calculates the target frequency to run the model on the hardware and automatically handles the oversampling factor. However, if the desired sample time is not achievable, then the Advisor displays a warning message providing details of estimated achievable frequency. To learn more, see “Estimate Achievable Target Frequency Without Running Synthesis” on page 30-155.

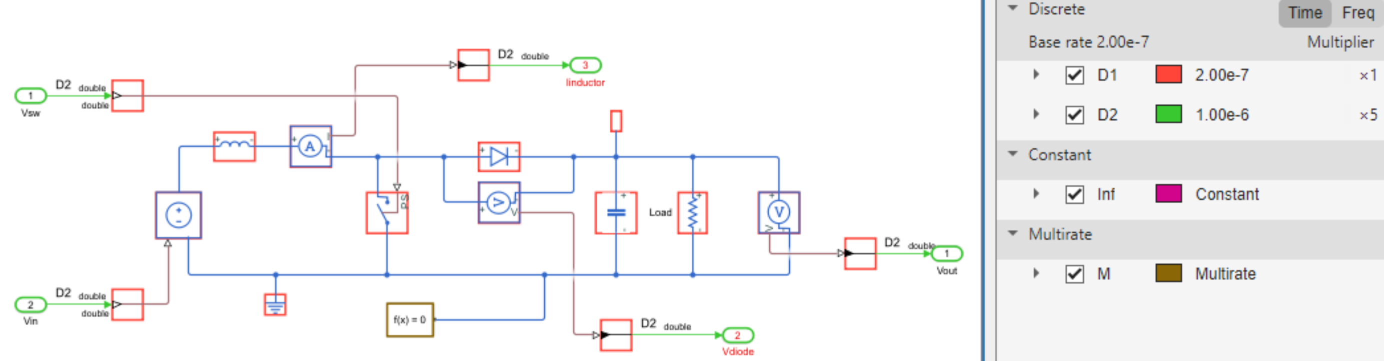
The preceding section uses the boost converter model as an example to illustrate how you can modify the oversampling factor and the number of solver iterations to improve the FPGA sample time.

Boost Converter Model

This example uses the boost converter model to illustrate the change in Simscape sample time in the generated HDL implementation model and the oversampling factor that is saved on the model.

1. Open the boost converter model. To learn how the boost converter is implemented, open the `Simscape_system` Subsystem. To open the boost converter model, in the MATLAB Command Window, enter:

```
openExample('plantdeployment/OpenTheSimscapeHDLWorkflowAdvisorExample','supportingFile','ssch
open_system('sschdlexBoostConverterExample/Simscape_system')
```



You see that the Simscape sample time for the model is `1e-6`. The sample time of `2.00e-7` corresponds to the sample time of the sources that drive the Simscape algorithm.

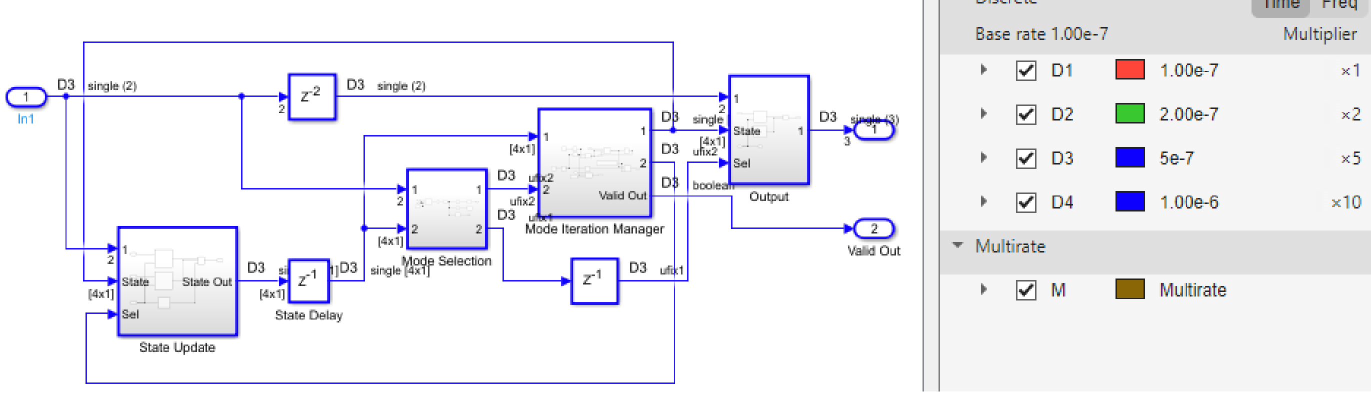
2. Open the Simscape HDL Workflow Advisor for your model:

```
sschdladvisor('sschdlexBoostConverterExample')
```

3. Run the workflow to the **Generate implementation model** task.

After running this task, you see a link to the generated HDL implementation model. Click the link to open the HDL implementation model.

4. Simulate the HDL implementation model. When you navigate the model to the `HDL Algorithm` Subsystem, you see that the model uses `single` data types and runs at a sample time `5.00e-7`, which is `5` times faster than the original Simscape model.



5. Run this command to see the HDL parameter settings that are saved on the model:

```
hdlsaveparams('gmStateSpaceHDL_sschdlexBoostConverterExamp')

%% Set Model 'gmStateSpaceHDL_sschdlexBoostConverterExamp' HDL parameters
hdlset_param('gmStateSpaceHDL_sschdlexBoostConverterExamp', 'AutoRoute', 'off');
hdlset_param('gmStateSpaceHDL_sschdlexBoostConverterExamp', 'FPToleranceValue', 1.000000e-03);
fpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT' ...
, 'LatencyStrategy', 'Min' ...
);
hdlset_param('gmStateSpaceHDL_sschdlexBoostConverterExamp', 'FloatingPointTargetConfiguration', fpconfig);
hdlset_param('gmStateSpaceHDL_sschdlexBoostConverterExamp', 'HDLSubsystem',...
 'gmStateSpaceHDL_sschdlexBoostConverterExamp/Simscape_system/HDL Subsystem');
hdlset_param('gmStateSpaceHDL_sschdlexBoostConverterExamp', 'MaskParameterAsGeneric', 'on');
hdlset_param('gmStateSpaceHDL_sschdlexBoostConverterExamp', 'Oversampling', 55);
hdlset_param('gmStateSpaceHDL_sschdlexBoostConverterExamp', 'UseFloatingPoint', 'on');

hdlset_param('gmStateSpaceHDL_sschdlexBoostConverterExamp/Simscape_system/HDL Subsystem/HDL Algorithm...
/Mode Selection/Generate Mode Vector', 'Architecture', 'MATLAB Datapath');
```

The HDL parameters that are saved indicate that the model has the native floating-point mode enabled and uses an **Oversampling factor** of `55` and has **Latency Strategy** set to `MIN`. These default values chosen for number of solver iterations and combination of HDL parameters offer an optimal tradeoff between oversampling factor and the target FPGA clock frequency and improves the FPGA sampling frequency. To further improve the FPGA sampling frequency, you can reduce the number of iterations and the oversampling factor as per the description here. Based on the suggested changes, you can verify if your model is compatible for these modifications.

Reducing Number of Solver Iterations

For each mode in the physical system, the workflow generates a state-space representation. The solver method is iterative and performs multiple computations to determine the correct mode for the next time step. After a certain number of iterations, the output value from the next time step becomes the same as the value from the previous time step. This consistency in the output value indicates the correct number of solver iterations.

The Advisor by default chooses an optimal value for the number of solver iterations. See “Using Number of Solver Iterations” on page 31-9. To improve the FPGA sampling frequency, reduce the number of solver iterations. The number of solver iterations depends on various factors such as the complexity of your design, the number of modes in the design that the workflow calculates, and so on.

In the Solver Configuration block, select the **Use fixed-cost runtime consistency iterations** check box and specify a custom value for the number of solver iterations in **Nonlinear iterations** text box. Start by reducing the number of solver iterations to a value such as `3`.

In the **Generate implementation model** task of the Simscape HDL Workflow Advisor, select **Generate validation logic for the implementation model**, and then generate the HDL implementation model. Simulate the HDL implementation model and open the **Diagnostic Viewer** to verify that the model does not display warnings or assertions.

If you see warnings or assertions, it indicates a simulation mismatch because the number of solver iterations that you specified is not adequate to compute the required number of modes in the state-space design. Resolve the mismatch by increasing the validation logic tolerance value or the number of solver iterations. Changing **Floating-point precision** to `double` is not recommended. Double-precision operations have large latencies and require a large **Oversampling factor** to allocate sufficient delays, which reduces the sampling frequency. See “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89.

Using Oversampling Factor and Latency Strategy

The **Oversampling factor** specifies the factor by which the FPGA clock rate is a multiple of the HDL implementation model base sample rate. The HDL implementation model contains feedback loops and performs multiplication of large matrices that have floating-point data types inside the feedback loops. To accommodate the large latency introduced by these floating-point operations inside the feedback loops, the code generator uses a large value of oversampling factor in conjunction with the clock-rate pipelining optimization on the model. For more information, see “Generate a Global Oversampling Clock” on page 20-9.

You vary the oversampling factor and latency strategy of the floating-point operator in conjunction. The default oversampling factor of `55` and minimum latency strategy gives an optimal sampling frequency. To achieve the maximum FPGA clock frequency, use the maximum latency strategy. When you specify this latency strategy, the floating-point operations introduce the maximum number of delays and higher FPGA clock frequency is achievable. To allocate these delays, increase the oversampling factor. If the increase in FPGA clock frequency outweighs the increase in oversampling factor, you achieve a higher sampling frequency.

To change the latency strategy and oversampling factor in conjunction from the Configuration Parameters dialog box:

1. On the **HDL Code Generation** > **Floating Point** pane, change the **Latency Strategy** to `Max`.
2. On the **HDL Code Generation** > **Global Settings** pane, increase the **Oversampling factor** to a value such as `100` depending on the complexity of your HDL design.

For the boost converter model, the default settings of **Number of solver iterations** set to `5`, **Oversampling factor** set to `55`, and **Latency Strategy** set to `Min` provides the optimal FPGA sampling frequency.

See Also

Functions
`sschdladvisor` | `simscape.findNonlinearBlocks`

More About

- “Solvers for Real-Time Simulation” (Simscape)
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 31-7
- “Latency Considerations with Native Floating Point” on page 14-104
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 30-28
- “Troubleshooting Real-Time Hardware Deployment Issues in Simscape Hardware-in-the-Loop Workflow” on page 32-2
- “Troubleshoot Validation Errors in Simscape Hardware-in-the-Loop Workflow” on page 32-9

Generate HDL Code for Nonlinear Simscape Models by Using Partitioning Solver

This example shows how to generate HDL code for a nonlinear Simscape™ model by using the Partitioning solver. You can then deploy the generated HDL code onto a Speedgoat® FPGA I/O module.

Introduction

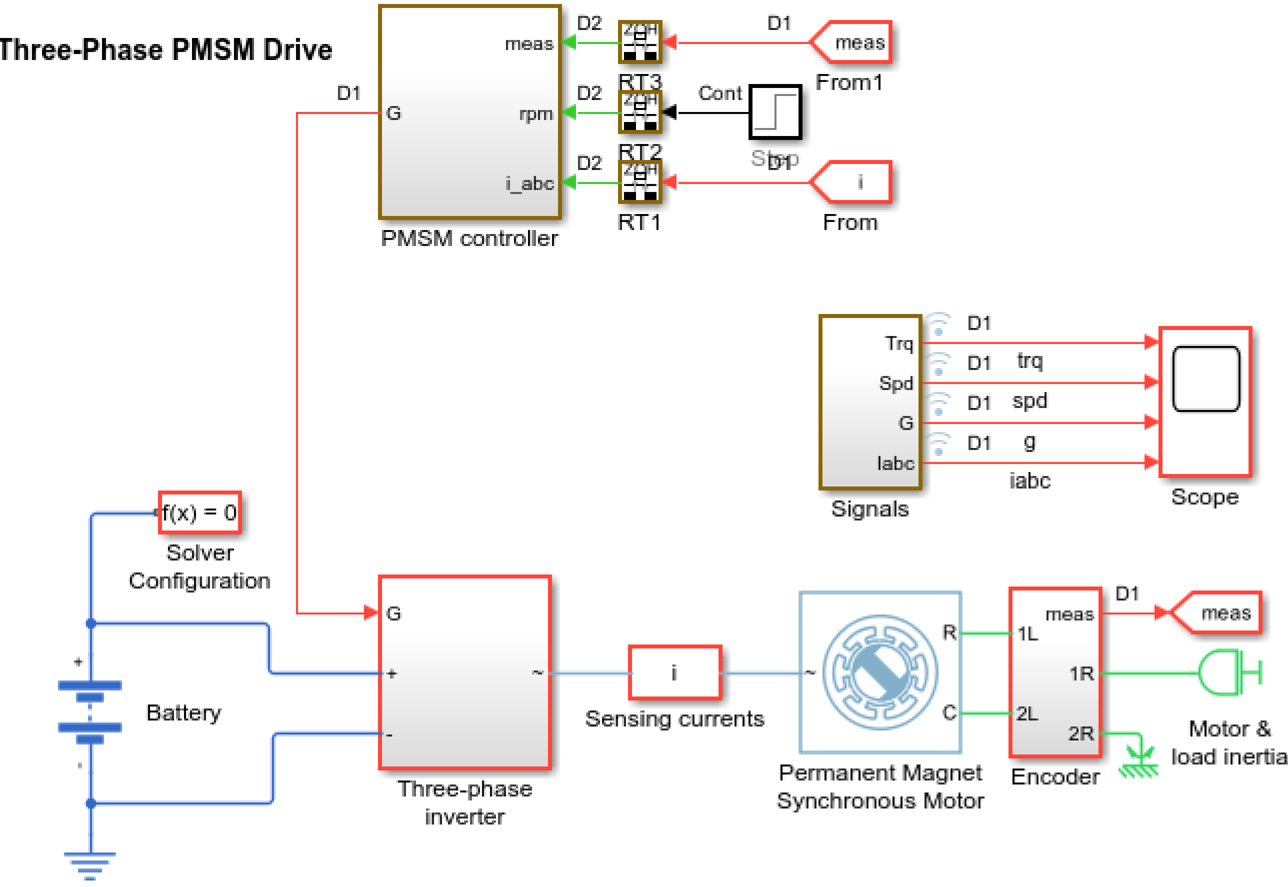
In this example, you learn how you can convert the Simscape™ three-phase permanent magnet synchronous motor (PMSM) model to an HDL implementation model by using the Simscape HDL Workflow Advisor. PMSM is a nonlinear block in the model that uses Partitioning solver. The Partitioning solver converts the entire system of equations for the Simscape network into several smaller sets of switched linear equations that are connected through nonlinear functions. By using this solver, you can run the Simscape HDL Workflow Advisor without having to remove nonlinear blocks in your model or replacing them with the corresponding Simulink blocks. For more information, see "Understanding How the Partitioning Solver Works" (Simscape). You can then generate HDL code and deploy the code to Speedgoat® FPGA I/O modules.

Permanent Magnet Synchronous Motor Model

The PMSM model is a physical system in Simscape™. This model uses field-oriented control (FOC) to control the speed of a three-phase permanent magnet synchronous motor (PMSM). The model contains a PMSM and a three-phase inverter that you can use in a typical hybrid vehicle. The inverter is connected to the battery.

To open the model, in the MATLAB® command prompt, enter:

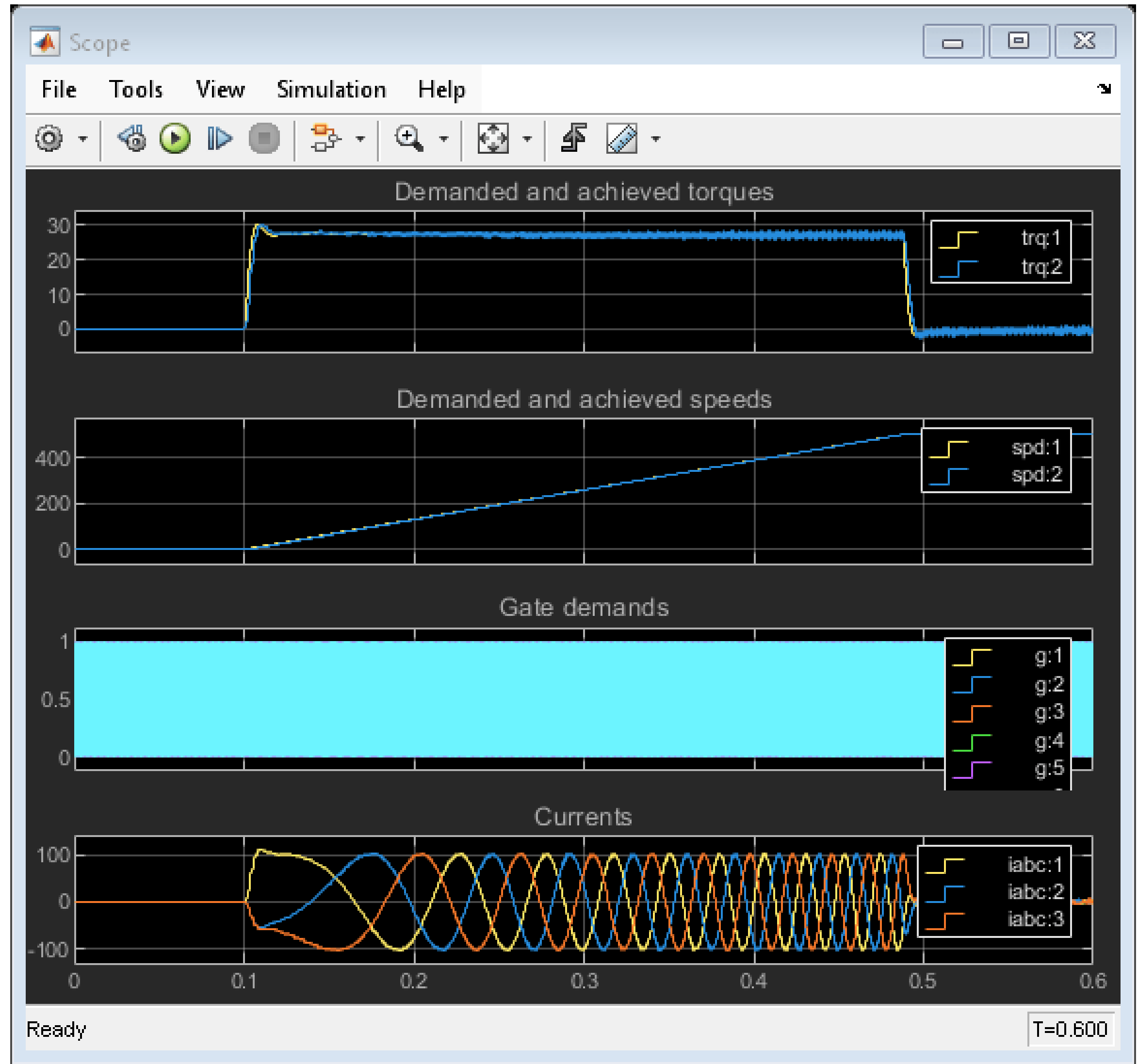
```
ModelName = 'sschdlexPMSMPartitioningSolver';
open_system(ModelName)
set_param(ModelName, 'SimulationCommand', 'update');
```



Copyright 2021-2023 The MathWorks, Inc.

To see how the model works, simulate the model.

```
sim(ModelName)
open_system([ModelName '/Scope'])
```



Configure the Simscape Model for HDL compatibility by using the `hdlsetup` function:

```
hdlsetup('sschdlexPMSMPartitioningSolver')
```

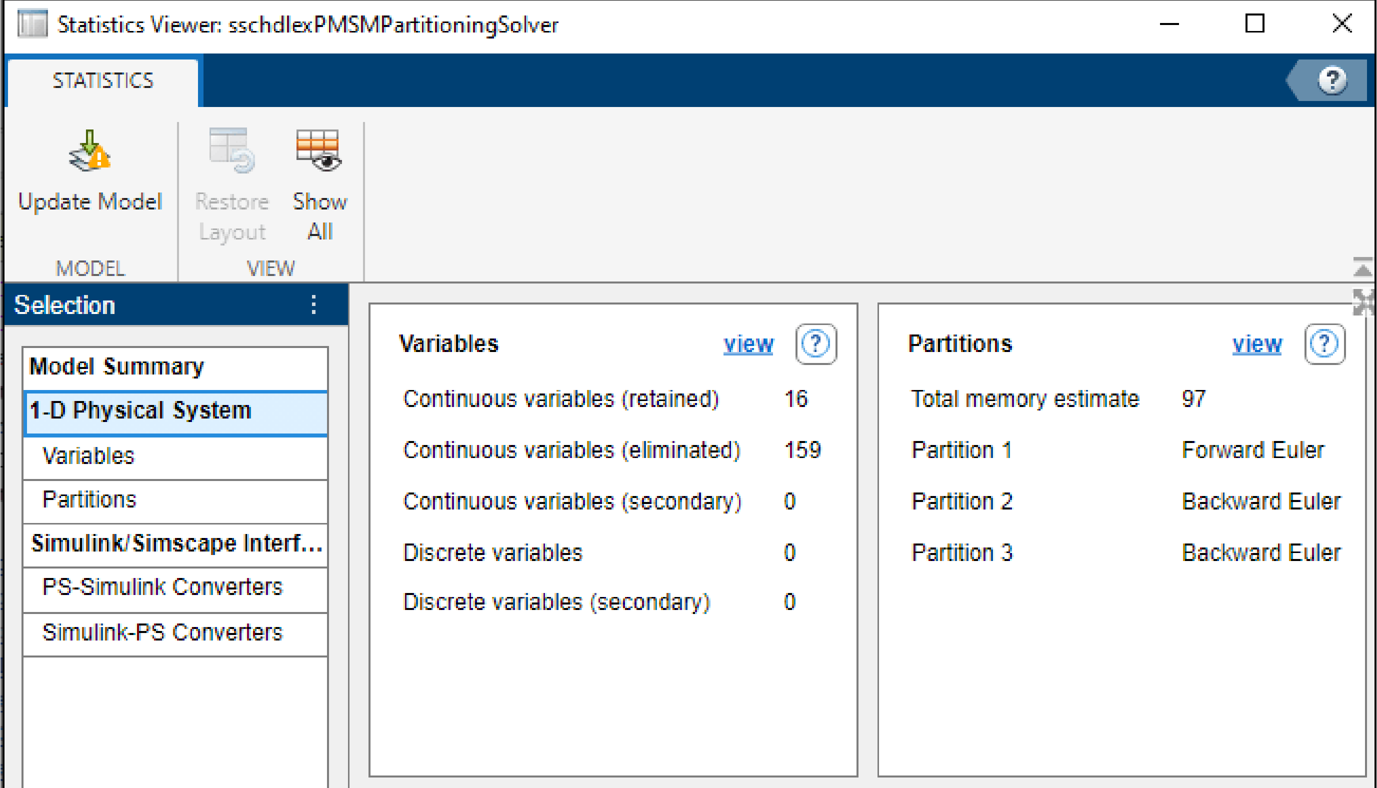
Setup and Configuration

To support the deployment of nonlinear models, choose the **Solver type** as `Partitioning` and a **Sample time** `Ts`. You can select a **Partition method** from the drop-down list. Select whether to prioritize speed or robustness when using the Partitioning solver. For more information, see Solver Configuration (Simscape). To show the functionality of the PMSM model, the default **Solver Configuration** settings are:

- Solver type: `Partitioning`
- Sample time: `Ts`
- Partitioning method: `Robust simulation`

In the model window, you can view the model statistics. Select the **Debug** tab and click **Simscape > Statistics Viewer**. This opens the Simscape Statistics window for

`sschdlexPMSMPartitioningSolver` model. Alternatively, you can click the `Model Statistics` link generated when the **Check model compatibility** task passes in the Simscape HDL Workflow Advisor. You can see that the solver divides the system into three partitions. The first partition is solved by using the Forward Euler method. The other two partitions are solved by using the Backward Euler method. Expand the **Number of partitions** node. Click each of the partitions to get a detailed description.



Generate HDL Implementation Model

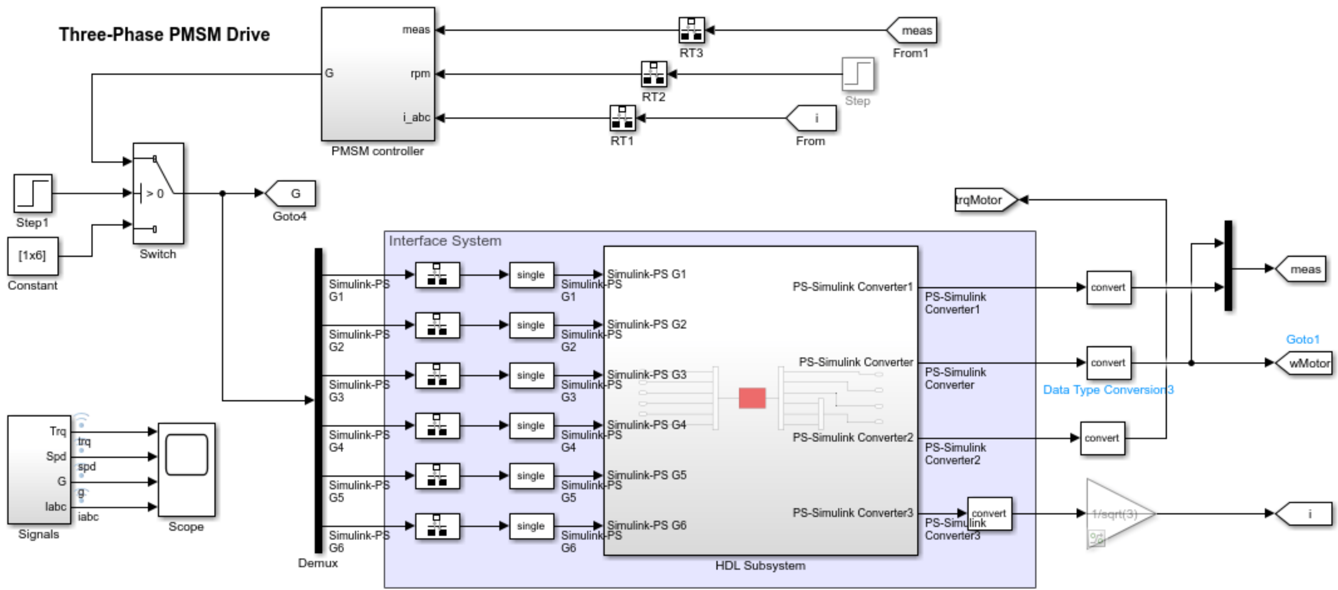
The Simscape HDL Workflow Advisor converts the Simscape plant model to an HDL-compatible implementation model from which you generate HDL code. The PMSM block is a nonlinear block for which you generate HDL-compatible implementation model. To generate the HDL implementation model:

1. Open the Simscape HDL Workflow Advisor:

```
sschdladvisor('sschdlexPMSMPartitioningSolver')
```

2. To generate the HDL implementation model, in the **Implementation model generation** task drop-down list, right-click the **Generate implementation model** task, and then select **Run to Selected Task** from the list.

After the task passes, you see a link to the HDL implementation model `gmStateSpaceHDL_sschdlexPMSMPartitioningSol`.



See also “Simscape HDL Workflow Advisor Tasks” on page 31-2.

When you run the **Extract discrete equations** task, the task displays the number of modes, states, clumps, and the state-space representation for the Simscape network in the right pane.

Details related to the Simscape network
sschdlexPMSMPartitioningSolver/Solver Configuration

- Number of States: 16
- Number of Modes: 6
- Number of Clumps: 3

Summary of the Partition Solver representation:

Clump	Parameter size of Ad	Parameter size of Bd
Differential Clump	2 x 2	
Algebraic Clump 1	4 x 4	4 x 4
Algebraic Clump 2	10 x 10 x 8	10 x 10 x 8

Generate HDL Code

Save the parameters for HDL implementation model.

```
hdlsaveparams('gmStateSpaceHDL_sschdlexPMSMPartitioningSol')
```

Enable generation of the resource utilization report.

```
hdlset_param('gmStateSpaceHDL_sschdlexPMSMPartitioningSol', 'ResourceReport', 'on')
```

Generate HDL code for the implementation model.

```
makehdl('gmStateSpaceHDL_sschdlexPMSMPartitioningSol/HDL Subsystem');
```

When you generate code, HDL Coder creates a code generation report. The resource utilization report in the **High-level Resource Report** indicates the amount of adders, multipliers, and registers that might be consumed on the target FPGA device.

Summary

Multipliers	180
Adders/Subtractors	1789
Registers	11242
Total 1-Bit Registers	120620
RAMs	0
Multiplexers	15132
I/O Bits	388
Static Shift operators	389
Dynamic Shift operators	202

Deploy Permanent Magnet Synchronous Motor to Speedgoat FPGA I/O Modules

In the HDL implementation model, the HDL Subsystem contains blocks you run on the FPGA. You can run the HDL Workflow Advisor on this Subsystem to deploy the HDL algorithm onto FPGA boards in Speedgoat target computers. For an example, see "Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules" on page 30-82.

See Also

Functions
`sschdladvisor` | `makehdl` | `hdlsetup` | `hdlsaveparams`

More About

- "Generate HDL Code for Simscape Models" on page 30-13
- "Simscape HDL Workflow Advisor Tasks" on page 31-2
- "Simscape HDL Workflow Advisor Tips and Guidelines" on page 31-7
- "Generate HDL Code for Simscape Three-Phase PMSM Drive Containing Averaged Switch" on page 30-118
- "Generate HDL Code for Simscape Models by Using Linearized Switch Approximation" on page 30-146

Deploy Simscape DC Motor Model to Speedgoat FPGA IO Module

This example shows how to generate an FPGA bitstream for a nonlinear Simscape™ model, such as a DC motor, and deploy it onto a Speedgoat® FPGA I/O module.

Introduction

In this example, you learn how to generate HDL code for Simscape models (both linear and nonlinear) and deploy the code onto FPGAs. You can generate an HDL implementation model by using the Simscape HDL Workflow Advisor. You then generate the HDL code and FPGA bitstream for the generated implementation model by using the HDL Workflow Advisor. You can simulate and validate your HDL compatible Simulink® model (HDL implementation model) against the Simscape model. Use the Simulink Real-Time™ development environment to generate, deploy, and run the real-time model run onto Speedgoat real-time target computers directly.

In this example, you learn how to:

1. Generate HDL code for a linear Simscape model by using the Backward Euler solver.
2. Generate HDL code for a nonlinear Simscape model by using the Partitioning solver.
3. Generate FPGA bitstream for the I/O 334-325K module by using the HDL Workflow Advisor.
4. Deploy the real-time model onto the Speedgoat real-time target machine by using Simulink Real-Time.

Setup and Configuration

1. Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

2. For real-time simulation, set up the development environment and target computer settings. See “Get Started with Simulink Real-Time” (Simulink Real-Time).

3. Install the Speedgoat Library and the Speedgoat HDL Coder Integration packages. See Install Speedgoat HDL Coder Integration Packages.

Buck Converter

A buck converter is a DC/DC power converter that steps down voltage from its input (source) to its output (load). In continuous conduction mode (current through the inductor never falls to zero), the theoretical transfer function of the buck converter is

$Vout/Vin = D$,

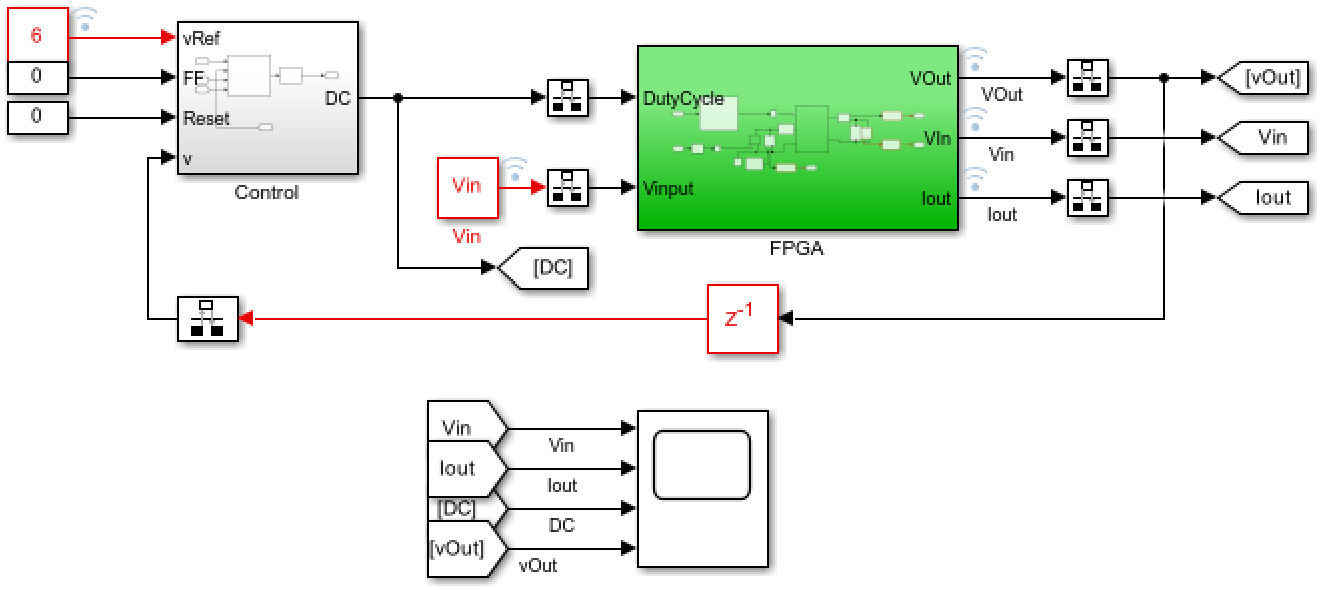
where D is the duty cycle. In this example, the converter feeds a resistor load from a 12 V source and the PWM frequency is set to 200 Hz. This example shows how to control the output voltage of a buck converter. To adjust the duty cycle, the Control subsystem uses a PI-based control algorithm.

Buck Converter with Fixed Resistor (Linear Simscape Network)

The model `ee_buck_converter_hdl.slx` consists of a buck converter with a fixed resistor as load. The fixed resistor is a linear Simscape element and therefore the model uses a Backward Euler solver.

To see the buck converter model, run this command.

```
open_system('ee_buck_converter_hdl')
```



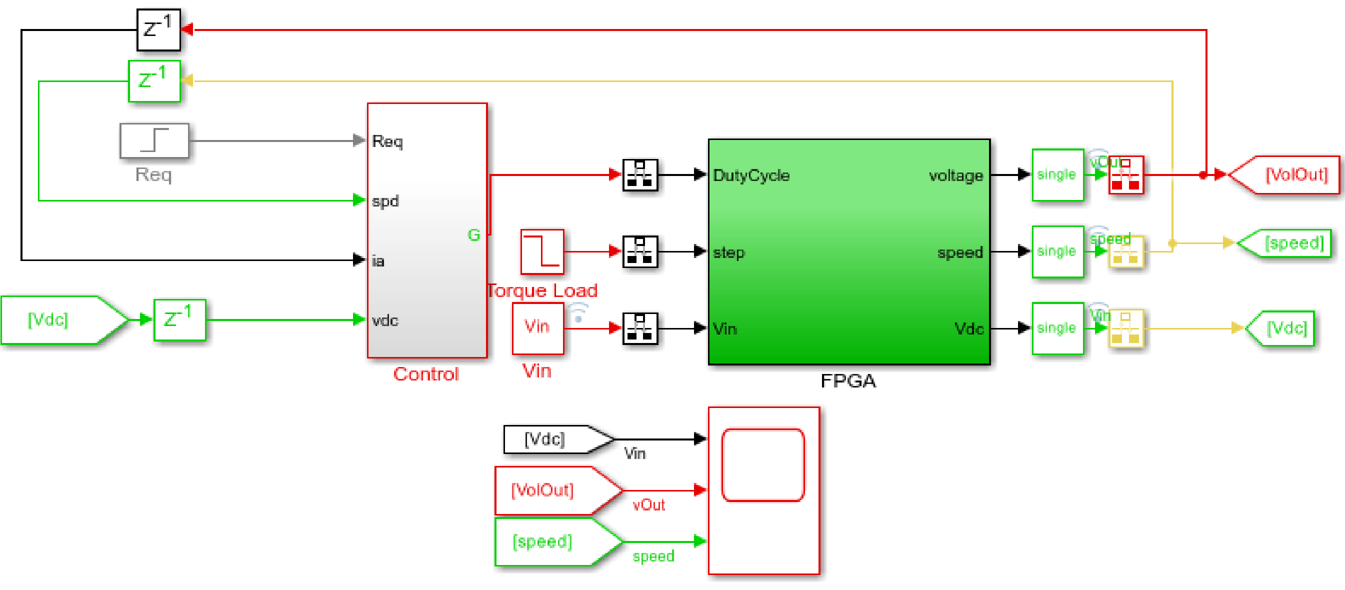
For details on steps of HDL code generation and deployment, see “Deploy Simscape Grid Tied Converter Model to Speedgoat IO Module Using HDL Workflow Script” on page 30-47.

Buck Converter with DC Motor (Nonlinear Simscape Network)

The model `ee_buck_converter_dc_motor_hdl.slx` consists of a buck converter with a DC motor as load. The DC motor is a nonlinear element, and this model therefore uses a Partitioning solver. For more information, see “Understanding How the Partitioning Solver Works” (Simscape).

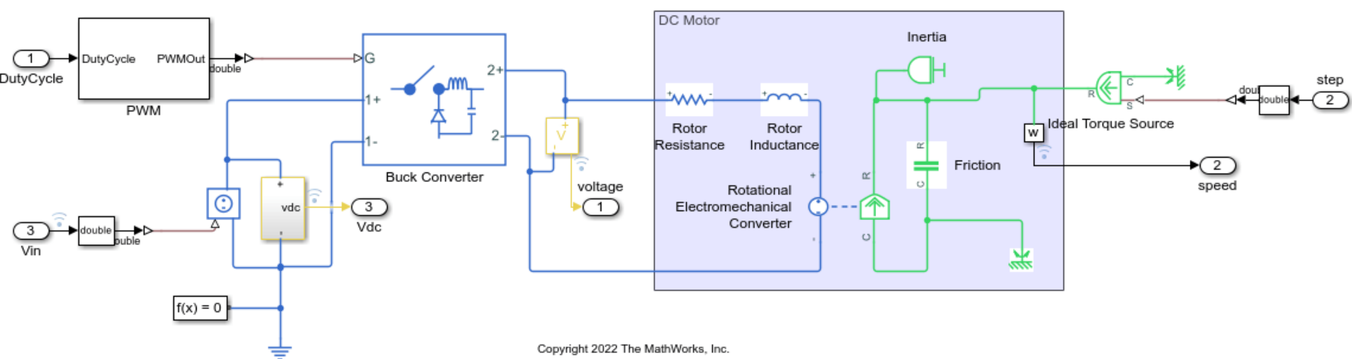
To see the buck converter model, run this command.

```
open_system('ee_buck_converter_dc_motor_hdl')
```



You generate VHDL® or Verilog® code for the blocks that are inside the green FPGA subsystem that contains the PWM generator and buck converter.

```
open_system('ee_buck_converter_dc_motor_hdl/FPGA')
```

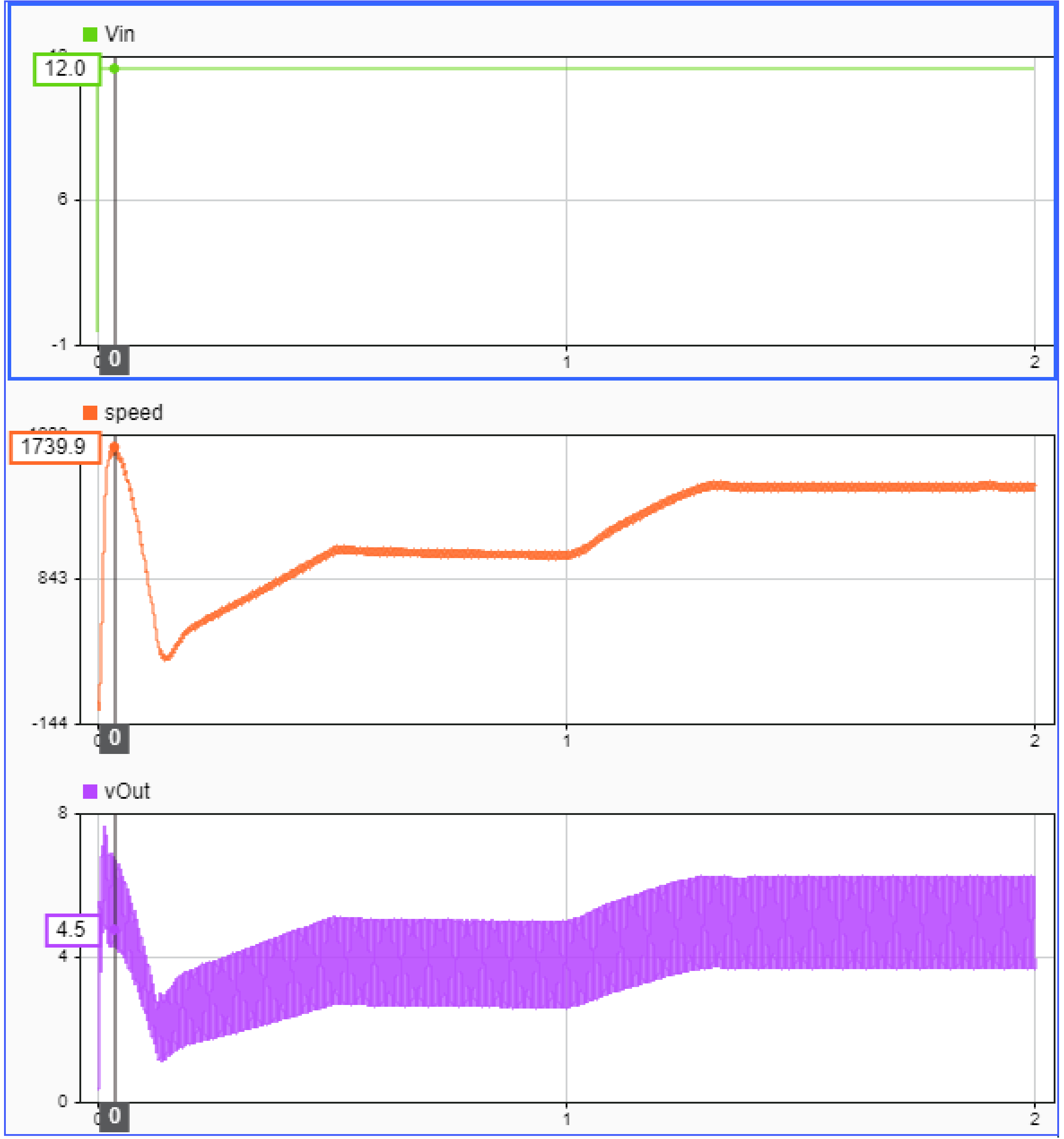


In the model, the Buck Converter block represents a converter that steps down DC voltage as driven by an attached controller and gate-signal generator. Buck converters are also known as step-down voltage regulators because they decrease voltage magnitude. The Rotational Electromechanical Converter block provides an interface between the electrical and mechanical rotational domains. It converts electrical energy into mechanical energy in the form of rotational motion, and can also convert the rotational motion into electrical energy.

Run Desktop Simulation of Buck Converter Model

The input signals that include the DC input voltage, PWM frequency, and duty cycle are generated on the top level of the model. The sample time for the Simscape model is set to 6 µs. Signal logging is enabled on the top level of the model.

```
sim('ee_buck_converter_dc_motor_hdl')

% Display the buck converter output signals in SDI
Simulink.sdi.clearAllSubPlots
Simulink.sdi.setSubPlotLayout(3,1);

allIDs2 = Simulink.sdi.getAllRunIDs;
runID2 = allIDs2(end);
run2 = Simulink.sdi.getRun(runID2);
run2.name = 'Simscape Desktop Simulation';

run2.getAllSignals;
plotOnSubPlot(run2.getSignalsByName('Vin'),1,1,true);
plotOnSubPlot(run2.getSignalsByName('speed'),2,1,true);
plotOnSubPlot(run2.getSignalsByName('vOut'),3,1,true);

Simulink.sdi.view;
```



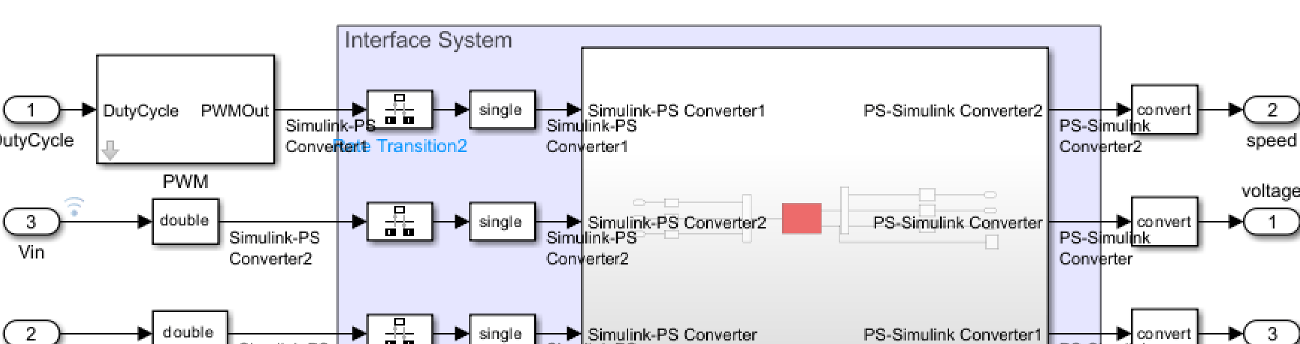
Generate HDL Implementation Model and Validate HDL Algorithm

The Simscape HDL Workflow Advisor converts the Simscape model to an HDL-compatible implementation model from which you generate HDL code. To open the Advisor, run the `sschdladvisor` function for your model.

```
sschdladvisor('ee_buck_converter_dc_motor_hdl')
```

To generate the HDL implementation model, in the **Implementation model generation** task drop-down list, right-click the **Generate implementation model** task, and then select **Run to Selected Task** from the list. After the task passes, you see a link to the HDL implementation model `gmStateSpaceHDL_ee_buck_converter_dc_motor_`.

```
open_system('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA')
```



To verify that the HDL implementation model matches the original Simscape model, generate a state-space validation model. In the **Generate implementation model** task, select the **Generate validation logic** for the implementation model check box and set the **Validation logic tolerance** to 1e-1. Then, run the task. See “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89.

HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.

```
hdlsetup('gmStateSpaceHDL_ee_buck_converter_dc_motor_')
```

- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis and timing analysis.
- Deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

Generate FPGA Bitstream for Speedgoat Target Computer

Use HDL Workflow Advisor to Generate Simulink Real-Time Interface Model

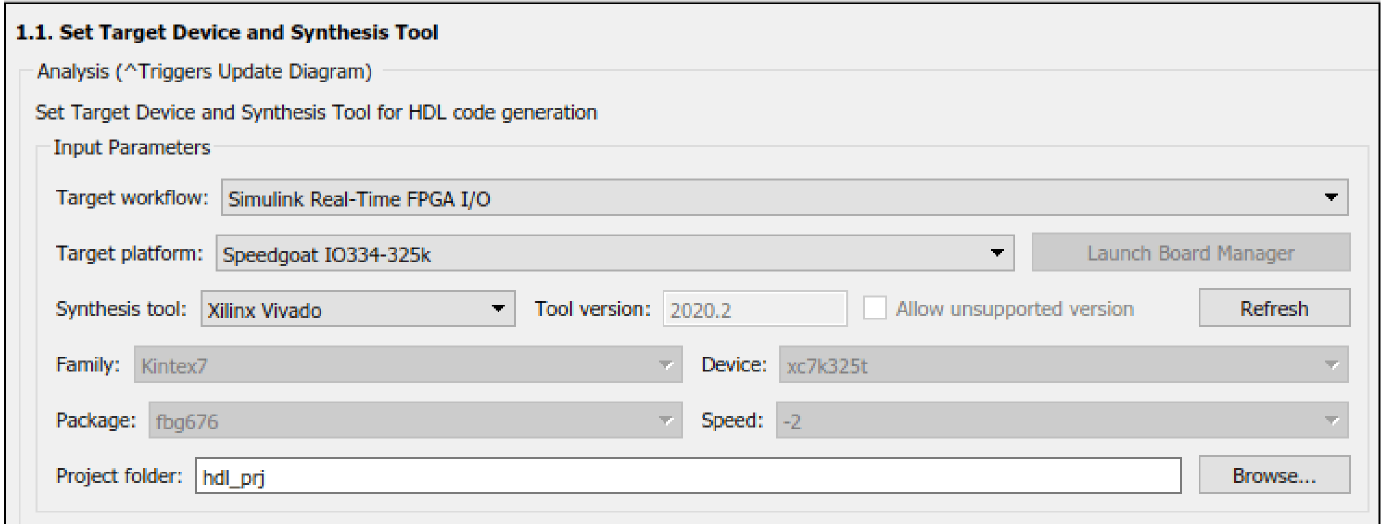
1. Open the HDL implementation model, and then open the HDL Workflow Advisor for the implementation model.

```
open_system('gmStateSpaceHDL_ee_buck_converter_dc_motor_')
```

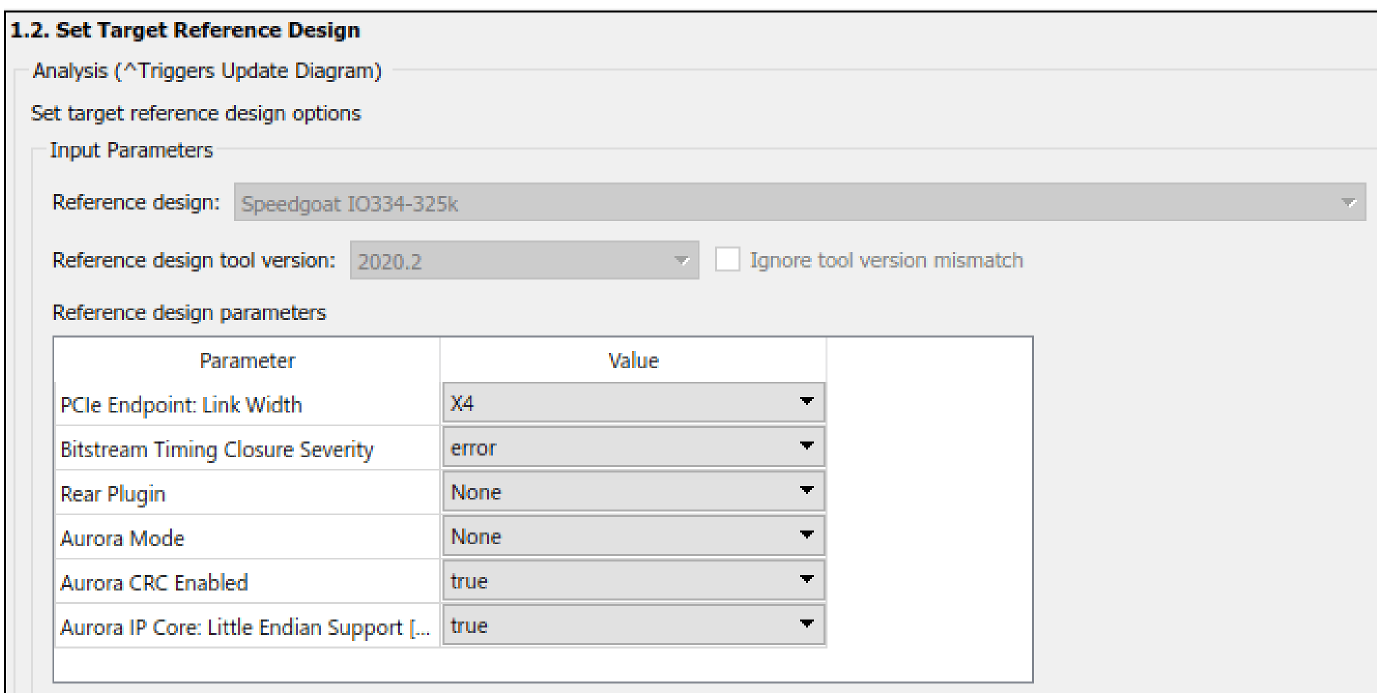
To open the HDL Workflow Advisor for a subsystem inside the model, use the `hdladvisor` function.

```
hdladvisor('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA')
```

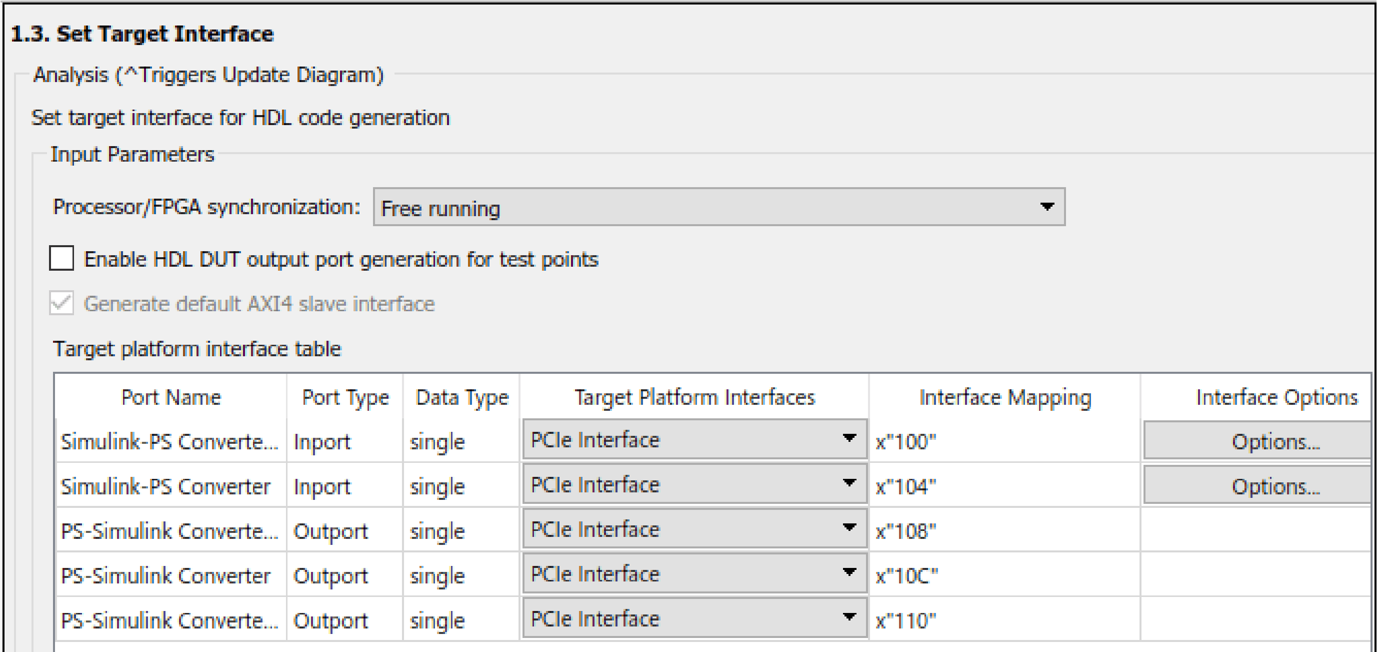
2. In the **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Simulink Real-Time FPGA I/O` and **Target platform** as `Speedgoat IO334-325K`.



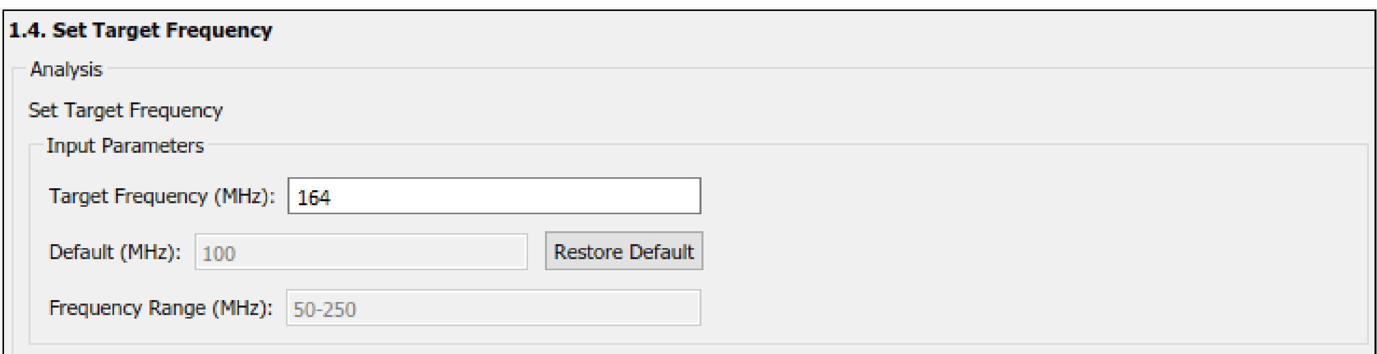
3. In the **Set Target Reference Design** task, select a value of X4 for the parameter `PCIe lanes`, and click the **Run This Task** button.



4. In **Set Target Interface** task, map the input and output `single` data type ports to `PCIe Interface` and click the **Run This Task** button.



5. In the **Set Target Frequency** task, the default value of **Target Frequency (MHz)** is set to 100. For this example model, this value is 164.



6. Right-click the **Generate Simulink Real-Time Interface** task, and select **Run to Selected Task** to generate the HDL IP core and FPGA bitstream.

Run Workflow Script to Generate Simulink Real-Time Interface Model

You can export the HDL Workflow Advisor settings to a script to expedite and automate your workflow. The script is a MATLAB® file that you run from the command line. You can modify and run the script, or import the settings into the HDL Workflow Advisor User Interface. See “Run HDL Workflow with a Script” on page 29-47.

This example shows how to run the HDL Workflow script. To generate a Simulink Real-Time Interface model, open and run this MATLAB script.

```
edit('hdlworkflow_dcmotor_IO334')


%--------------------------------------------------------------------------
```

```
% HDL Workflow Script
% This script contains the model, target settings, interface mapping, and
% the Workflow Configuration settings for generating HDL code for the HDL
% implementation model generated for the Buck Converter Model with DC Motor
% as load, and for deploying the code to the FPGA on board the Speedgoat
% IO334-325K module.
%------------------------------------------------------------------------

%% Load the Model
load_system('gmStateSpaceHDL_ee_buck_converter_dc_motor_');

%% Model HDL Parameters
%% Set Model 'gmStateSpaceHDL_ee_buck_converter_dc_motor_' HDL parameters
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'AdaptivePipelining', 'on');
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'FPToleranceValue', 1.0e-01);
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'FloatingPointTargetConfiguration', h
, 'LatencyStrategy', 'Max') ...
);
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'HDLSubsystem', 'gmStateSpaceHDL_ee_b
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'MaskParameterAsGeneric', 'on');
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'Oversampling', 328);
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'ReferenceDesign', 'Speedgoat IO334-3
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'SynthesisToolChipFamily', 'Kintex7')
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'SynthesisToolDeviceName', 'xc7k325t
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'SynthesisToolPackageName', 'fbg676')
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'SynthesisToolSpeedValue', '-2');
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'TargetDirectory', 'C:\BuckConverterD
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'TargetFrequency', 164);
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'TargetPlatform', 'Speedgoat IO334-32
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'Workflow', 'Simulink Real-Time FPGA

% Set SubSystem HDL parameters
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA', 'AXI4SlaveIDWidth', '14');
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA', 'ProcessorFPGASynchronization',

% Set Inport HDL parameters
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/DutyCycle', 'IOInterface', 'PCIe I
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/DutyCycle', 'IOInterfaceMapping',

% Set Inport HDL parameters
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/step', 'IOInterface', 'PCIe Inter
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/step', 'IOInterfaceMapping', 'x"10

% Set Inport HDL parameters
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/Vin', 'IOInterface', 'PCIe Interfa
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/Vin', 'IOInterfaceMapping', 'x"108

% Set Outport HDL parameters
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/voltage', 'IOInterface', 'PCIe Int
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/voltage', 'IOInterfaceMapping', 'x

% Set Outport HDL parameters
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/speed', 'IOInterface', 'PCIe Inter
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/speed', 'IOInterfaceMapping', 'x"1

% Set Outport HDL parameters
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/Vdc', 'IOInterface', 'PCIe Interfa
```

```
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/Vdc', 'IOInterfaceMapping', 'x"11

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Simulink Real-Ti

% Specify the top level project directory
hWC.ProjectFolder = 'C:\BuckConverterDCMotor\hdl_prj';
hWC.ReferenceDesignToolVersion = '2020.2';
hWC.IgnoreToolVersionMismatch = false;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndIPCore' Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';
hWC.EnableIPCaching = true;

% Set properties related to 'RunTaskBuildFPGABitstream' Task
hWC.RunExternalBuild = false;
hWC.EnableDesignCheckpoint = false;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;
hWC.CustomBuildTclFile = '';
hWC.DefaultCheckpointFile = 'Default';
hWC.RoutedDesignCheckpointFilePath = '';
hWC.MaxNumOfCoresForBuild = '';

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA', hWC);
```

Deploy Bitstream to Speedgoat IO334-325k Target

Prepare Simulink Real-Time Interface Model for Real-Time Simulation

Running the workflow script generates RTL code and IP core, creates a Vivado project, builds the FPGA bitstream, and then generates the Simulink Real-Time interface model.

The FPGA subsystem is automatically replaced in the real-time interface model with Speedgoat driver blocks to initialize the hardware and to interface with the FPGA during runtime.

Buck Converter Voltage Control with DC motor load

Copyright 2022 The MathWorks, Inc.

Make sure you comment out any remaining `Rate Transition` blocks in the model to obtain a single rate model. All the blocks in the generated Simulink Real-Time model are executed on the CPU of the real-time system. The buck converter and DC motor are programmed in the bitstream and run at the faster rate on the FPGA.

Connect to Target Machine and Run Real-Time Simulation

The model can now be deployed onto the Speedgoat real-time target machine. Make sure that the Speedgoat real-time target machine is connected to the host computer and powered on.

You download the bitstream by using the Simulink Real-Time Explorer. To open the Simulink Real-Time Explorer, enter the command `slrtExplorer`. Alternatively, you can open the Explorer from the **Real-Time** tab of the Simulink Toolstrip.

```
slrtExplorer
```

The Simulink Editor displays the **Real-Time** tab for models that are configured for the `speedgoat.tlc` code generation target. Click the **Connect to Target Computer** button in the Simulink **Real-Time** tab to connect to the machine. Once connected, click the **Run on Target** button to deploy the model.

Simulink Real-Time automatically generates C code from your model by using Simulink Coder. The generated code and the bitstream for the FPGA are loaded onto the target machine, and model execution starts automatically.

See Also

Functions
`sschdladvisor` | `hdladvisor` | `hdlsetup`

More About

- “Generate HDL Code for Simscape Models” on page 30-13
- “Simscape HDL Workflow Advisor Tasks” on page 31-2
- “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82
- “Deploy Simscape Grid Tied Converter Model to Speedgoat IO Module Using HDL Workflow Script” on page 30-47
- “Generate HDL Code for Nonlinear Simscape Models by Using Partitioning Solver” on page 30-100

Generate HDL Code for Simscape Three-Phase PMSM Drive Containing Averaged Switch

This example shows how to generate HDL code and synthesize the results for a three-phase PMSM Simscape™ model that has an averaged switch.

Simscape Models with Averaged Switch

You can generate HDL code for Simscape models with converter blocks that have averaged switches and deploy onto the FPGAs. This example uses a three-phase PMSM model with a six-pulse three-phase controlled converter block consisting of averaged switches. The switching frequency of this model is 50 kHz. For the system to be HDL compatible, the averaged switches use a piecewise constant approximation method for the gate input. Note that this model with the averaged switches might introduce instability during dead time (when all gate inputs are set to 0) with the Partitioning local solver.

Set Up Synthesis Tool Path

If you want to synthesize the generated HDL code, before you use HDL Coder™ to generate code, set up your synthesis tool path. For example, if your synthesis tool is Xilinx® Vivado®, then, install the latest version of Xilinx Vivado as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

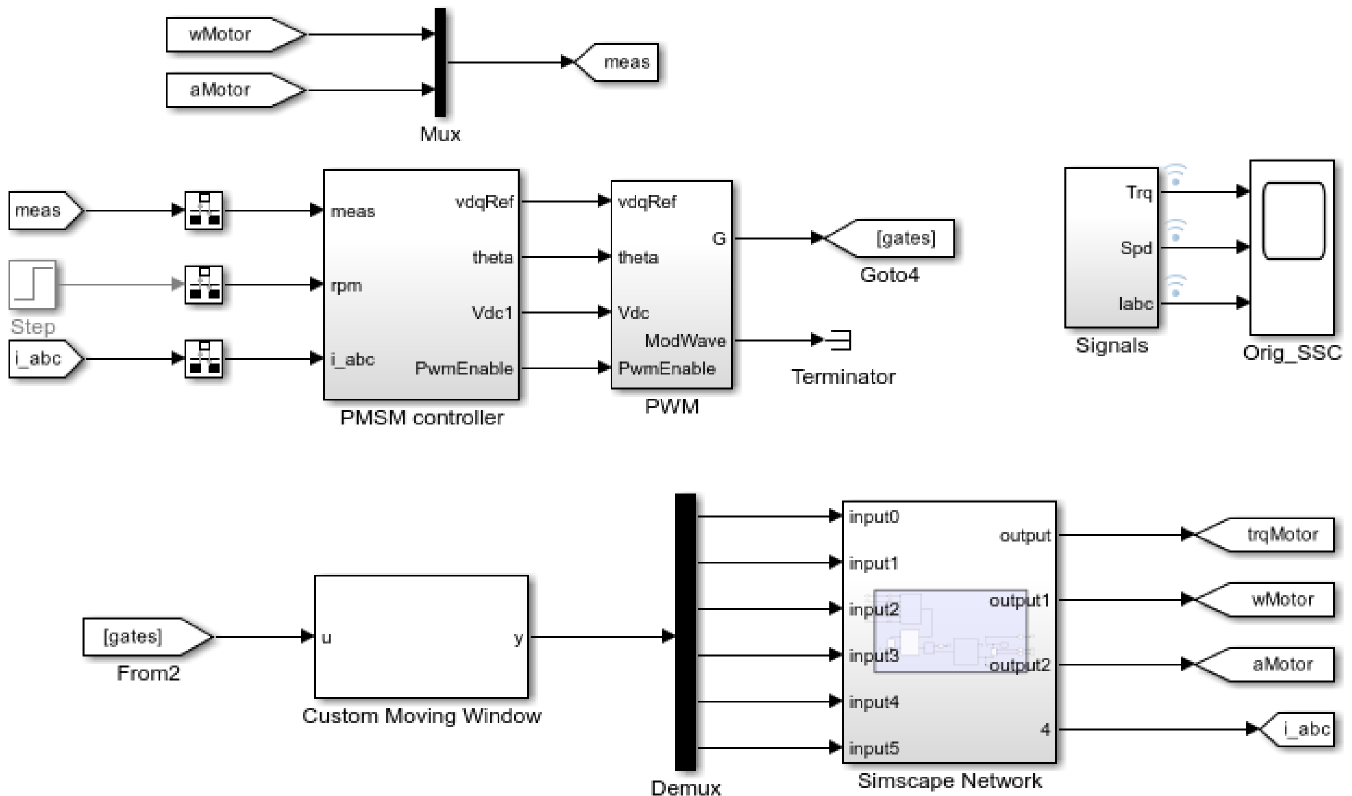
Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function:

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

Three-Phase PMSM Drive Model

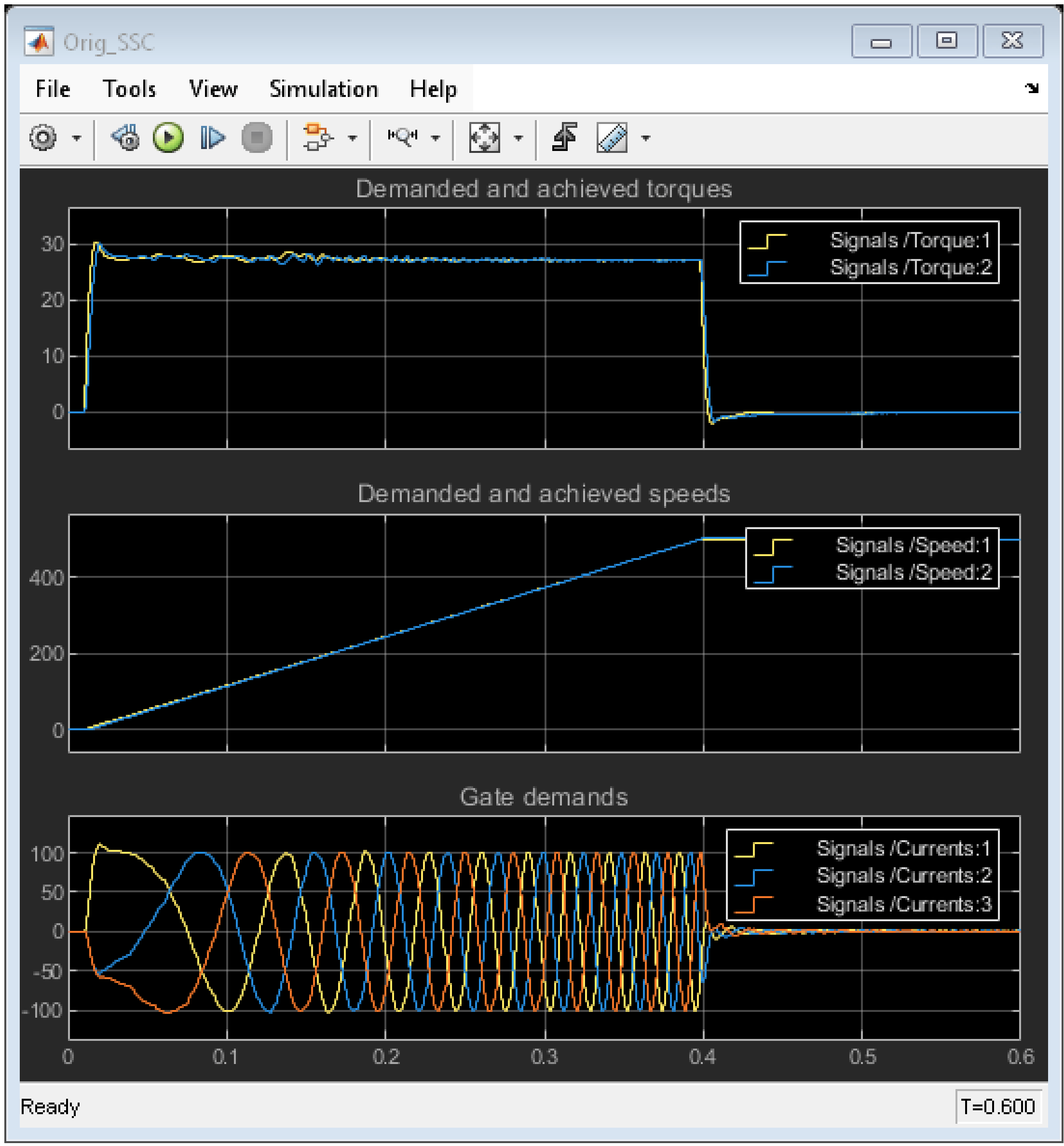
Open the model from the MATLAB® command prompt.

```
ModelName = 'ee_pmsm_drive_averaged_switch_hdl';
open_system(ModelName)
```



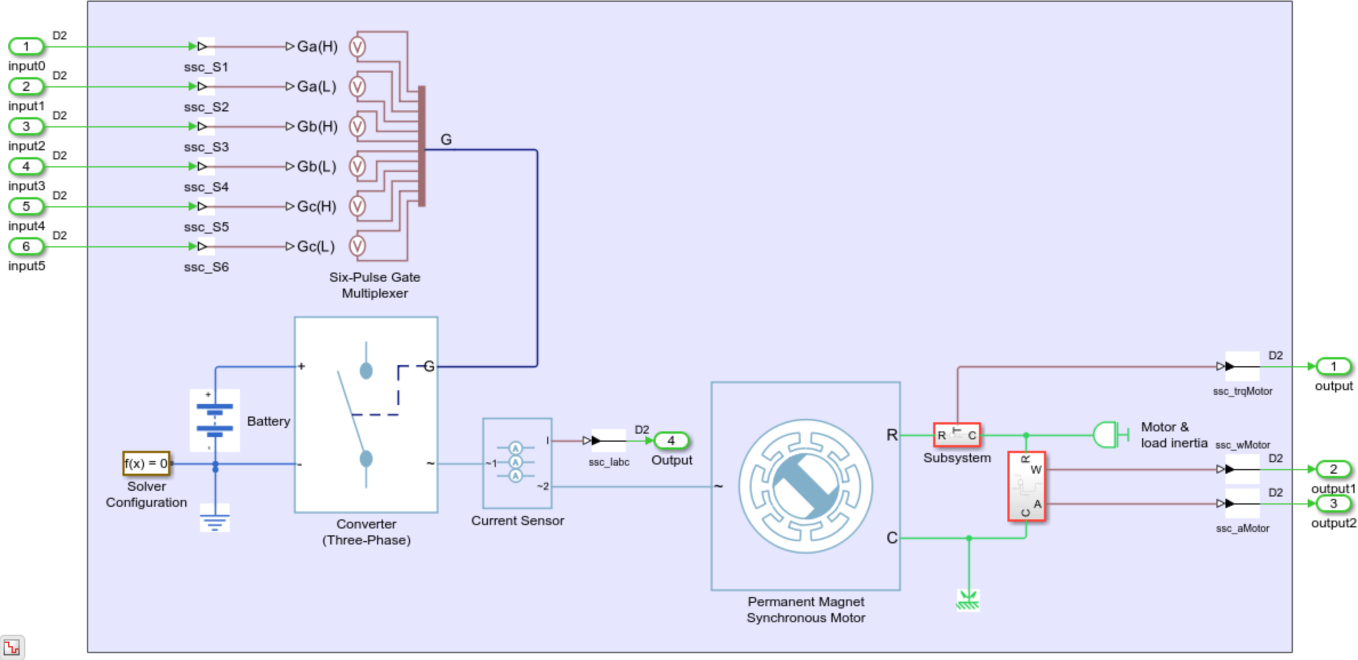
To see how the model works, simulate it.

```
sim(ModelName)
open_system([ModelName '/Orig_SSC'])
```



To see the components inside the Simscape Network subsystem, enter:

```
open_system('ee_pmsm_drive_averaged_switch_hdl/Simscape Network')
```



For HDL code generation, in the Converter(Three-Phase) block parameter **Settings** tab, the **Integer for piecewise constant approximation of gate input (0 for disabled)** parameter value is set to 10. The model is set up to use Partitioning solver by selecting the **Use local solver** check box in the Solver Configuration (Simscape) block.

Generate HDL Implementation Model

The Simscape HDL Workflow Advisor converts the Simscape plant model to an HDL-compatible implementation model from which you generate HDL code. To generate the HDL implementation model:

1. Open the Simscape HDL Workflow Advisor.

```
sschdladvisor('ee_pmsm_drive_averaged_switch_hdl')
```

2. In the **Implementation model generation** task drop-down list, right-click the **Generate implementation model** task, then select **Run to Selected Task** from the list. To get better resource utilization, in the **Generate implementation model** task window, set **Map state space parameter to RAM** to **On**.

After the task passes, you see a link to the HDL implementation model `gmStateSpaceHDL_ee_pmsm_drive_averaged_swit`.

Three-Phase PMSM Drive with Averaged Switch

Generate HDL Code

To modify the configuration parameter values for HDL code generation, navigate to `gmStateSpaceHDL_ee_pmsm_drive_averaged_swit/Simscape Network/HDL Subsystem`. Right-click the HDL Algorithm subsystem and select `HDL Block Properties` from the **HDL Code** list. In the HDL Properties dialog box, set the `DistributedPipelining` to `On`. Click **Apply** and then click **OK**. Then navigate to the `gmStateSpaceHDL_ee_pmsm_drive_averaged_swit/Simscape Network/HDL Subsystem/HDL Algorithm/State Update/Algebraic Clump2/Mode Vector To Index` block. Right-click the block and select `HDL Block Properties` from the **HDL Code** list. In the HDL Properties dialog box, set the `InputPipeline` and `OutputPipeline` parameter values to 2, click **Apply** and then click **OK**. Right-click the Multiply State and Multiply F blocks, set the `SharingFactor` to 8 for the two blocks. Save the model and proceed to the next step.

Set your model configuration parameters to default values recommended for HDL code generation.

```
hdlsetup('gmStateSpaceHDL_ee_pmsm_drive_averaged_swit')
```

HDL Workflow Advisor

The HDL Workflow Advisor guides you through the necessary tasks required for generating HDL code and an FPGA design process. It provides you with feedback on the results of each task. When you complete the tasks, you have a synthesis result report from one of the supported synthesis tools.

Open the subsystem `HDL Subsystem` within the HDL implementation model into the HDL Workflow Advisor.

```
hdladvisor('gmStateSpaceHDL_ee_pmsm_drive_averaged_swit/Simscape Network/HDL Subsystem')
```

You can also open the HDL Workflow Advisor from your model window. Right-click the `HDL Subsystem` and select `HDL Workflow Advisor` from the list.

Set Target Device and Synthesis Tool

Before you generate HDL code, if you want to deploy the code onto a target platform, specify the synthesis tool.

1. Open the HDL Workflow Advisor.
2. Under the **Set Target** task folder, in the **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Generic ASIC/FPGA` and **Synthesis tool** as `Xilinx Vivado`. The rest of the fields are auto-populated. Specify the **Family** as `Kintex7`, **Device** as `xc7k325t`, **Package** as `fbg676`, and **Speed** as -1.
3. In the **Set Target Frequency** window, specify the **Target Frequency** as 150.
4. Select the task that you want to run and click **Run This Task**.

HDL Code Generation

- In the **HDL Code Generation** task folder, click **Set HDL Options** and then click the **HDL Code Generation Settings** button. This opens the Configuration Parameters dialog box. Select `Adaptive pipelining` under **HDL Code Generation** > **Optimizations** > **Pipelining** and click **Apply**.
- Under the **HDL Code Generation** > **Global Settings** > **Clock Settings** section, set the `Oversampling factor` to 300. Click **Apply** then click **OK**. To generate HDL code, run the tasks under the **HDL Code Generation** task folder.

Synthesize Generated HDL Code

HDL Coder synthesizes the HDL code on the target platform and generates area and timing reports for your design based on the target device that you specify. You can run logic synthesis for a specified FPGA device and get the synthesis reports.

In the **FPGA Synthesis and Analysis** task folder:

- Create an FPGA synthesis project for your supported FPGA synthesis tool.
- Start supported FPGA synthesis tools to perform synthesis, mapping, and place/route tasks. To run FPGA synthesis, right-click the **Run Synthesis** task under the **Perform Synthesis and P/R** subtask folder. This starts Xilinx Vivado and executes the Vivado **Synthesis** step. You can annotate your original model with critical path information obtained from the synthesis tools.

Parsed resource report file: HDL_Subsystem_utilization_synth.rpt.

Resource summary			
Resource	Usage	Available	Utilization (%)
Slice LUTs	137096	203800	67.27
Slice Registers	106687	407600	26.17
DSPs	221	840	26.31
Block RAM Tile	115	445	25.84
URAM	0	0	

Parsed timing report file: timing_post_map.rpt.

Timing summary	
	Value
Requirement	6.6667 ns (150 MHz)
Data Path Delay	1.644 ns
Slack	1.8 ns
Clock Frequency	205.48 MHz

See Also

Functions
`sschdladvisor` | `hdladvisor` | `hdlsetup`

More About

- “Generate HDL Code for Simscape Models” on page 30-13
- “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82
- “Deploy Simscape Grid Tied Converter Model to Speedgoat IO Module Using HDL Workflow Script” on page 30-47
- “Generate HDL Code for Nonlinear Simscape Models by Using Partitioning Solver” on page 30-100
- “Generate HDL Code for Simscape Models by Using Linearized Switch Approximation” on page 30-146

Generate HDL Code for Two-Speed Transmission Model Containing Mode Charts

This example shows how to generate HDL code for a Simscape™ model containing mode charts.

Simscape Models with Mode Charts

You can use a mode chart implementation to model a Simscape component with multiple operating modes and transitions. Mode charts provide an intuitive way to model components characterized by a discrete set of distinct operating modes. A car clutch is a good example of such a component. To learn more about mode charts, see `modecharts` (Simscape) and "Mode Chart Modeling" (Simscape).

In this example, a two-speed transmission model with braking is designed for HDL code generation. This model uses the Partitioning solver as a local solver. For more information, see "Understanding How the Partitioning Solver Works" (Simscape). First, you generate an HDL implementation model by using the Simscape HDL Workflow Advisor. For this implementation model, you generate the HDL code and can synthesize the results by using the guided steps in the HDL Workflow Advisor.

Set Up Synthesis Tool Path

If you want to synthesize the generated HDL code, before you use HDL Coder™ to generate code, set up your synthesis tool path. For example, if your synthesis tool is Xilinx® Vivado®, install the latest version of Xilinx Vivado as listed in "HDL Language Support and Supported Third-Party Tools and Hardware".

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function:

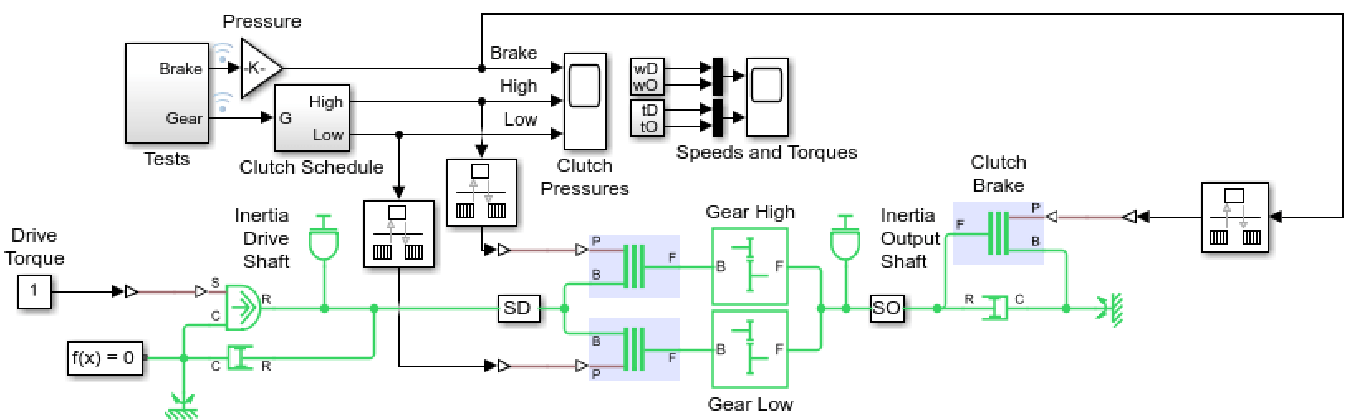
```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

Two-Speed Transmission Model with Braking

The two-speed transmission model is a Simscape driveline model. This transmission model couples the gears in a simple way, with each gear and the brake associated with its own clutch. Coupling one gear requires engaging and locking the corresponding clutch, while ensuring that the other two clutches are disengaged. The brake clutch is directly activated by its own switch. Two clutches control which gear is selected. A brake is connected to the output shaft and can be controlled independently. The highlighted blocks in the model contain mode charts.

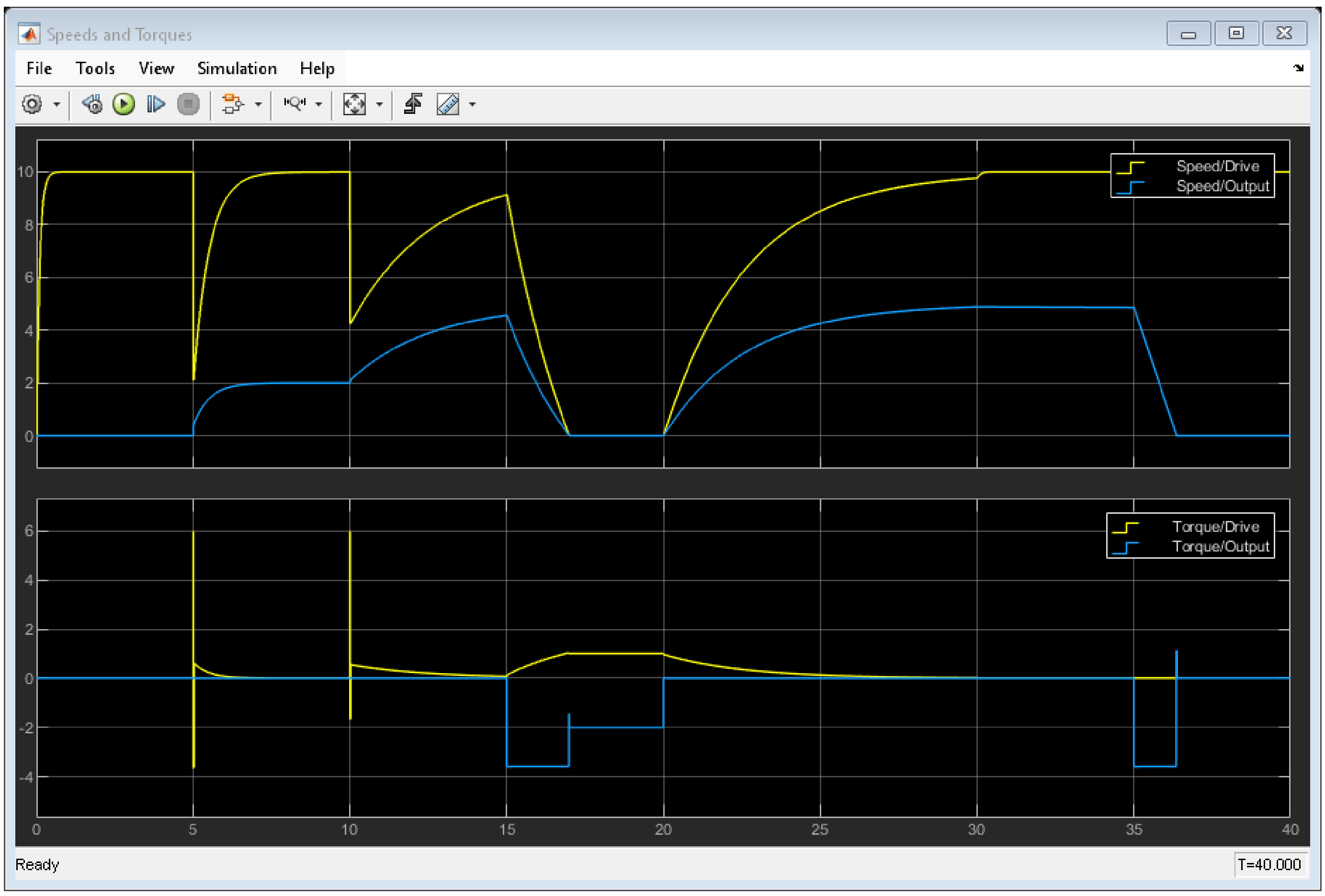
Open the model from the MATLAB® command prompt.

```
ModelName = 'sdl_transmission_2spd_hdl';
open_system(ModelName)
```



To see how the model works, simulate it

```
sim(ModelName)
open_system(['sdl_transmission_2spd_hdl' '/Speeds and Torques'])
```



Generate HDL Implementation Model

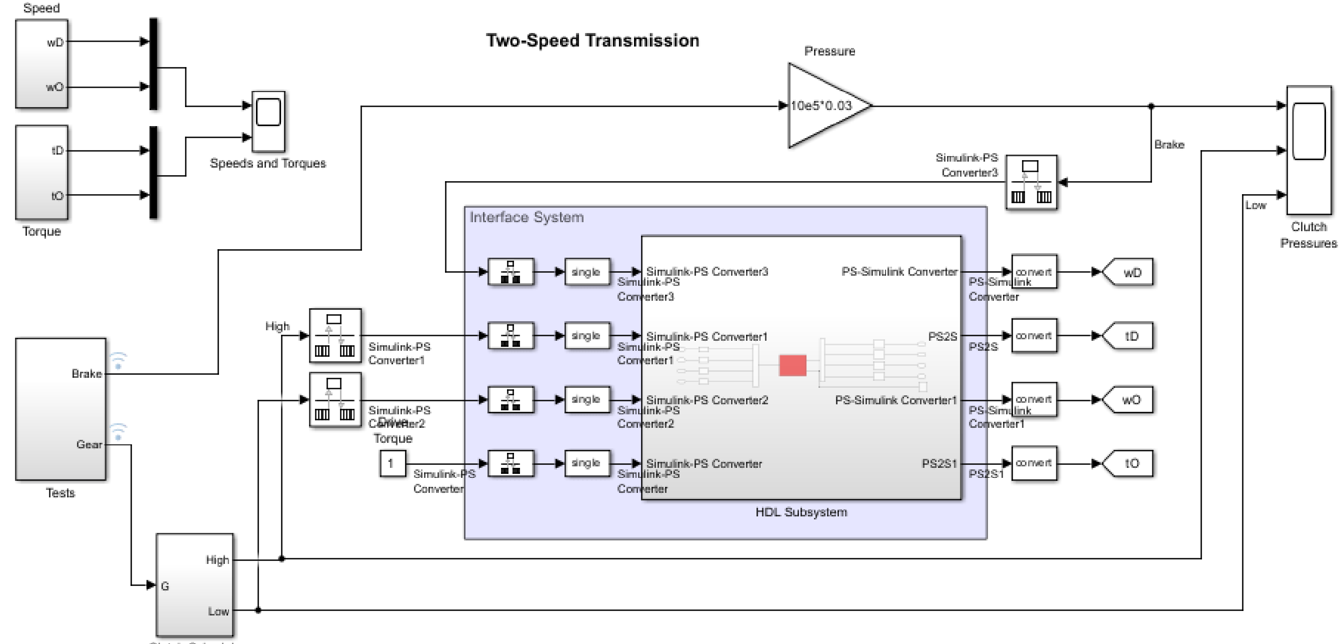
The Simscape HDL Workflow Advisor converts the Simscape plant model to an HDL-compatible implementation model from which you generate HDL code. To generate the HDL implementation model:

1. Open the Simscape HDL Workflow Advisor.

```
sschdladvisor('sdl_transmission_2spd_hdl')
```

2. In the **Implementation model generation** task drop-down list, right-click the **Generate implementation model** task, then select **Run to Selected Task** from the list. To get better resource utilization, in the **Generate implementation model** task window, set **Map state space parameter to RAM** to **On**.

After the task passes, you see a link to the HDL implementation model `gmStateSpaceHDL_sdl_transmission_2spd_hdl`.



Generate HDL Code

To modify the configuration parameter values for HDL code generation, run

```
hdlsetup('gmStateSpaceHDL_sdl_transmission_2spd_hdl')
```

HDL Workflow Advisor

The HDL Workflow Advisor guides you through the necessary tasks required for generating HDL code and an FPGA design process. It provides you with feedback on the results of each task. When you complete the tasks, you have a synthesis result report from one of the supported synthesis tools.

To open the subsystem `HDL Subsystem` within the HDL implementation model into the HDL Workflow Advisor, run

```
hdladvisor('gmStateSpaceHDL_sdl_transmission_2spd_hdl/HDL Subsystem')
```

You can also open the HDL Workflow Advisor from your model window. Right-click the `HDL Subsystem` and select `HDL Workflow Advisor` from the list.

Set Target Device and Synthesis Tool

Before you generate HDL code, if you want to deploy the code onto a target platform, specify the synthesis tool.

1. Open the HDL Workflow Advisor.
2. Under the **Set Target** task folder, in the **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Generic ASIC/FPGA` and **Synthesis tool** as `Xilinx Vivado`. The rest of

the fields are auto-populated. Specify the **Family** as `Kintex7`, **Device** as `xc7k325t`, **Package** as `fbg676`, and **Speed** as -1.

3. In the **Set Target Frequency** window, specify the **Target Frequency** as 100.
4. Select the task that you want to run and click **Run This Task**.

HDL Code Generation

- In the **HDL Code Generation** task folder, click **Set HDL Options** and then click the **HDL Code Generation Settings** button. This opens the Configuration Parameters dialog box. Select `Adaptive pipelining` under **HDL Code Generation** > **Optimizations** > **Pipelining** and click **Apply**.
- Under the **HDL Code Generation** > **Global Settings** > **Clock Settings** section, set the `Oversampling factor` to 500. Click **Apply** then click **OK**. To generate HDL code, run the tasks under the **HDL Code Generation** task folder.

Synthesize Generated HDL Code

HDL Coder synthesizes the HDL code on the target platform and generates area and timing reports for your design based on the target device that you specify. You can run logic synthesis for a specified FPGA device and get the synthesis reports.

In the **FPGA Synthesis and Analysis** task folder:

- Create an FPGA synthesis project for your supported FPGA synthesis tool.
- Start supported FPGA synthesis tools to perform synthesis, mapping, and place/route tasks. To run FPGA synthesis, right-click the **Run Synthesis** task under the **Perform Synthesis and P/R** subtask folder. This starts Xilinx Vivado and executes the Vivado **Synthesis** step. You can annotate your original model with critical path information obtained from the synthesis tools.

Passed Synthesis

Parsed resource report file: HDL_Subsystem_utilization_synth.rpt.

Resource summary			
Resource	Usage	Available	Utilization (%)
Slice LUTs	89105	203800	43.72
Slice Registers	84906	407600	20.83
DSPs	225	840	26.79
Block RAM Tile	38	445	8.54
URAM	0	0	

Parsed timing report file: timing_post_map.rpt.

Timing summary	
	Value
Requirement	10 ns (100 MHz)
Data Path Delay	5.558 ns
Slack	4.41 ns
Clock Frequency	178.89 MHz

See Also

Functions
`sschdladvisor` | `hdladvisor` | `hdlsetup`

More About

- “Generate HDL Code for Simscape Models” on page 30-13
- `modecharts` (Simscape)
- “Mode Chart Modeling” (Simscape)
- “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82
- “Deploy Simscape Grid Tied Converter Model to Speedgoat IO Module Using HDL Workflow Script” on page 30-47
- “Generate HDL Code for Nonlinear Simscape Models by Using Partitioning Solver” on page 30-100

Simscape Language Support

In this section...

"Domain and Component Declarations" on page 30-131

"Equations" on page 30-132

"Discrete Events and Mode Charts" on page 30-133

"Composite Components" on page 30-134

The Simscape language extends the Simscape modeling environment by enabling you to create new components that do not exist in the Foundation library or in any of the add-on products. It is a dedicated textual language for modeling physical systems. The Simscape language lets you define custom components as textual files, complete with parameterization, physical connections, and underlying equations. The components you create can reuse the physical domain definitions provided with Simscape to ensure that your components are compatible with the standard Simscape components. You can also add your own physical domains and provide complete component libraries for these domains. You can deploy the textual component files in block diagrams by converting them into custom Simscape blocks. For more information, see "What Is the Simscape Language?" (Simscape).

The tables list the Simscape Hardware-in-the-Loop (HIL) Workflow support for the Simscape language for different solvers.

Domain and Component Declarations

Keyword	Description	Backward Euler/ Trapezoidal Rule Solver	Partitioning Solver
`annotations`	Control appearance of Simscape block based on the component	✔	✔
`branches`	Establish relationship between component Through variables and nodes	✔	✔
`component`	Keyword to define component model classes	✔	✔
`domain`	Keyword to define domain model classes	✔	✔
`inputs`	Define component inputs, that is, Physical Signal input ports of block	✔	✔
`nodes`	Define component nodes, that is, conserving ports of block	✔	✔

Keyword	Description	Backward Euler/ Trapezoidal Rule Solver	Partitioning Solver
`outputs`	Define component outputs, that is, Physical Signal output ports of block	✔	✔
`parameters`	Declare domain or component parameters	✔	✔
`variables`	Declare domain or component variables	✔	✔

Equations

Keyword	Description	Backward Euler/ Trapezoidal Rule Solver	Partitioning Solver
`assert`	Program customized run-time errors and warnings	✔	✔
`delay`	Return past value of operand	╳	╳
`der`	Return time derivative of operand	✔	✔
`equations`	Define component or domain equations	✔	✔
`function`	Reuse expressions in component equations and in member declarations of domains and components	✔	✔
`integ`	Perform time integration of expression	╳	╳
`intermediates`	Define intermediate terms for use in equations	✔	✔
`scatteredlookup`	Return value based on interpolating unstructured set of data points	╳	╳
`tablelookup`	Return value based on interpolating set of data points	╳	✔
`time`	Access global simulation time	╳	✔

Keyword	Description	Backward Euler/ Trapezoidal Rule Solver	Partitioning Solver
value	Convert variable or parameter to unitless value with specified unit conversion	✔	✔

Discrete Events and Mode Charts

Discrete Variables and Events

Keyword	Description	Backward Euler/ Trapezoidal Rule Solver	Partitioning Solver
events	Model discrete events	╳	✔
edge	Trigger event	╳	✔
initialevent	Initialize event variables	╳	✔

Mode Charts

Keywords	Description	Backward Euler/ Trapezoidal Rule Solver	Partitioning Solver
modecharts	Declare mode charts that include operating modes and transitions	╳	✔
modes	Declare operating modes in mode chart	╳	✔
transitions	Define transitions between modes in mode chart	╳	✔
initial	Specify initial mode in mode chart	╳	✔
entry	Specify actions to be performed upon entering a mode	╳	✔

Composite Components

Language Syntax

Keywords	Description	Backward Euler/ Trapezoidal Rule Solver	Partitioning Solver
`components`	Declare member components included in composite component	✔	✔
`connect`	Connect two or more component ports of the same type	✔	✔
`connections`	Define connections for member component ports in composite component	✔	✔
`import`	Import model classes	✔	✔

See Also

"What Is the Simscape Language?" (Simscape) | "Simscape File Types and Structure" (Simscape) | "Understanding How the Partitioning Solver Works" (Simscape) | "Making Optimal Solver Choices for Physical Simulation" (Simscape)

Related Examples

- "Typical Simscape Language Tasks" (Simscape)

Generate HDL Code for Simscape Models by Using Trapezoidal Rule Solver

This example shows how to generate HDL code for a Simscape™ model by using the Trapezoidal Rule solver. You can then deploy the generated HDL code onto a Speedgoat® FPGA I/O module.

Setup and Configuration

1. Install the latest version of Xilinx® Vivado® as listed in "HDL Language Support and Supported Third-Party Tools and Hardware".

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

2. For real-time simulation, set up the development environment and target computer settings. See "Get Started with Simulink Real-Time" (Simulink Real-Time).

3. Install the Speedgoat Library and the Speedgoat HDL Coder Integration packages. See Install Speedgoat HDL Coder Integration Packages.

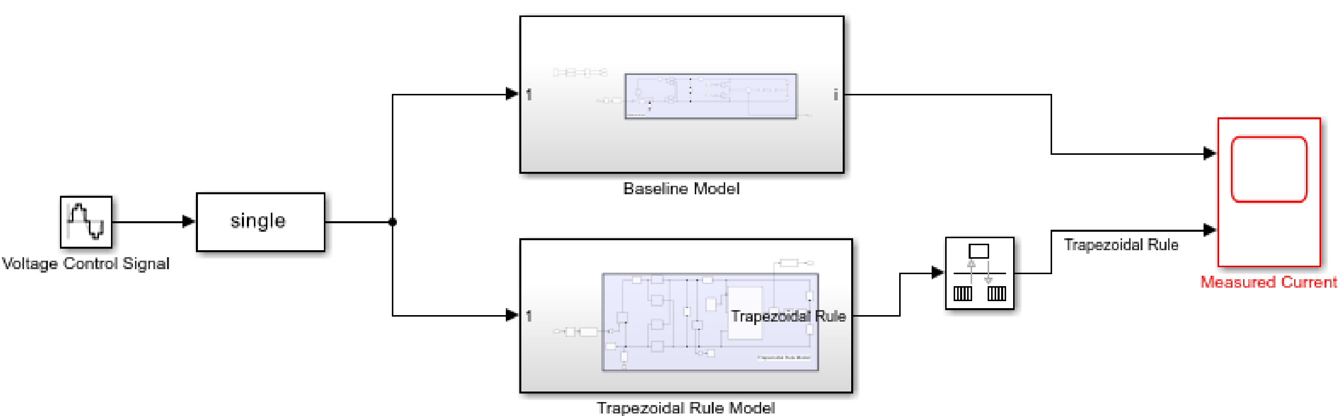
Use Trapezoidal Rule Solver to Simulate Large Time Steps

This example models an induction stovetop. The network model uses a half-bridge with subcycle averaging to generate alternating current to power induction coils. The model contains a network with fast RLC time constants. With the local solver set to `Trapezoidal Rule`, you can simulate the model with a larger time step and increased accuracy than with a Backward Euler local solver. Some electrical topologies such as wireless power transfer circuits cannot be accurately simulated with a Backward Euler solver with a time step larger than 100 ns. For real-time simulation, the Trapezoidal Rule solver lets you run these models with larger time steps (~1 µs) for specific hardware requirements.

The model in this example generates the necessary current to pass through the induction coils in an induction stovetop. The current is measured at the output.

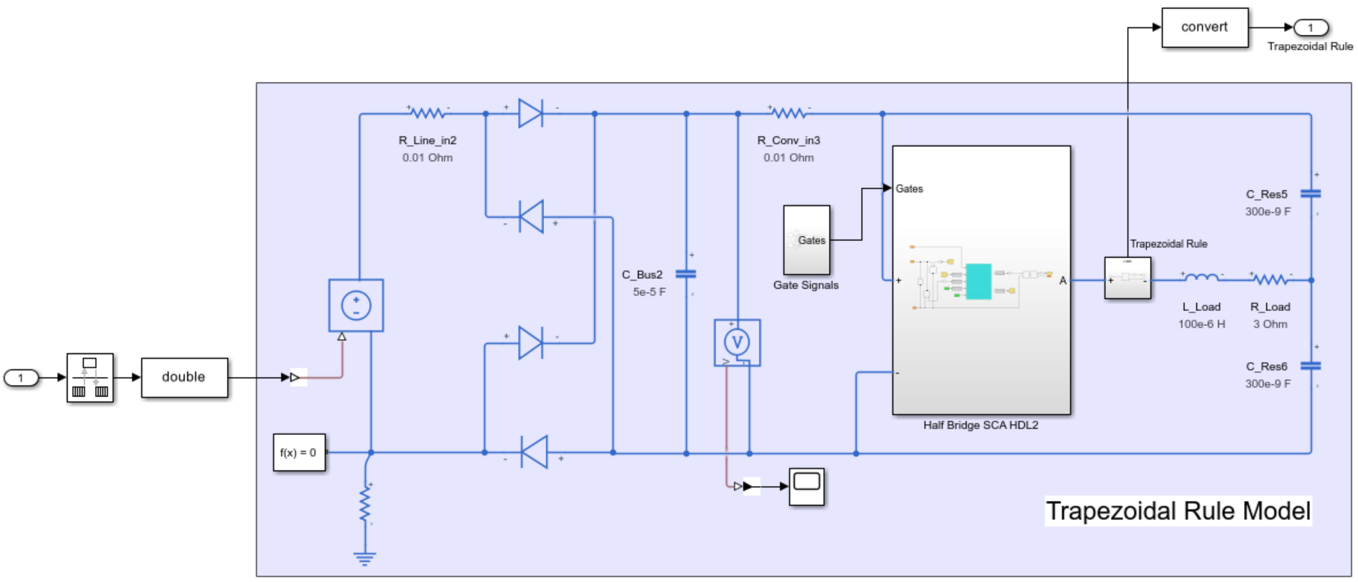
Open the model from the MATLAB® command prompt.

```
ModelName = 'sschdl_trapezoidal_example';
open_system(ModelName)
```



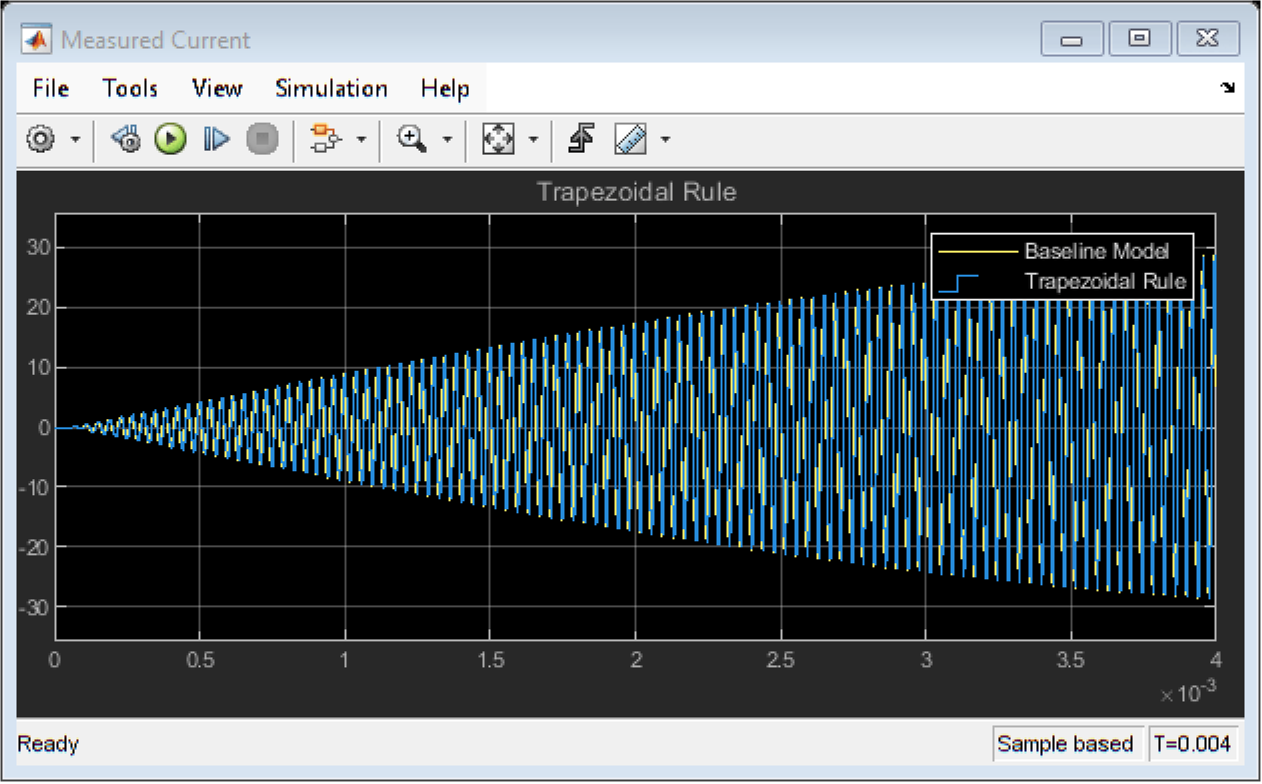
Open the Trapezoidal Rule Model subsystem

```
open_system('sschdl_trapezoidal_example/Trapezoidal Rule Model')
```



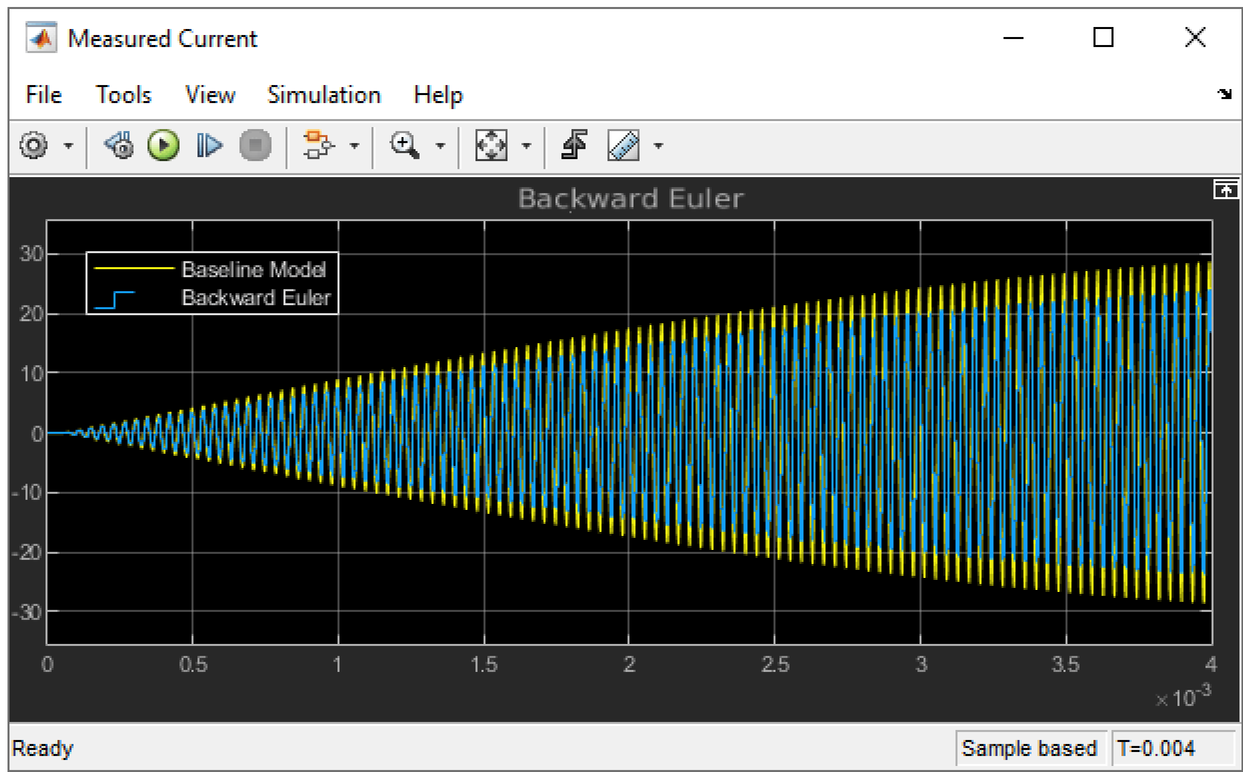
Simulate the model. To compare the results of Trapezoidal Rule local solver with the expected results from Baseline Model (using a variable step solver), open the Measured Current scope and observe the waveforms.

```
sim(ModelName);
open_system(['sschdl_trapezoidal_example' '/Measured Current'])
```



You can see that the two waveforms for the Baseline Model and Trapezoidal Rule Model overlap. There is no difference in the two results.

When you use the Backward Euler local solver with the same time step (1e-6 s) used with the Trapezoidal Rule local solver, the measured current is not correct. The Backward Euler local solver requires a smaller time step to provide accurate results for this model.



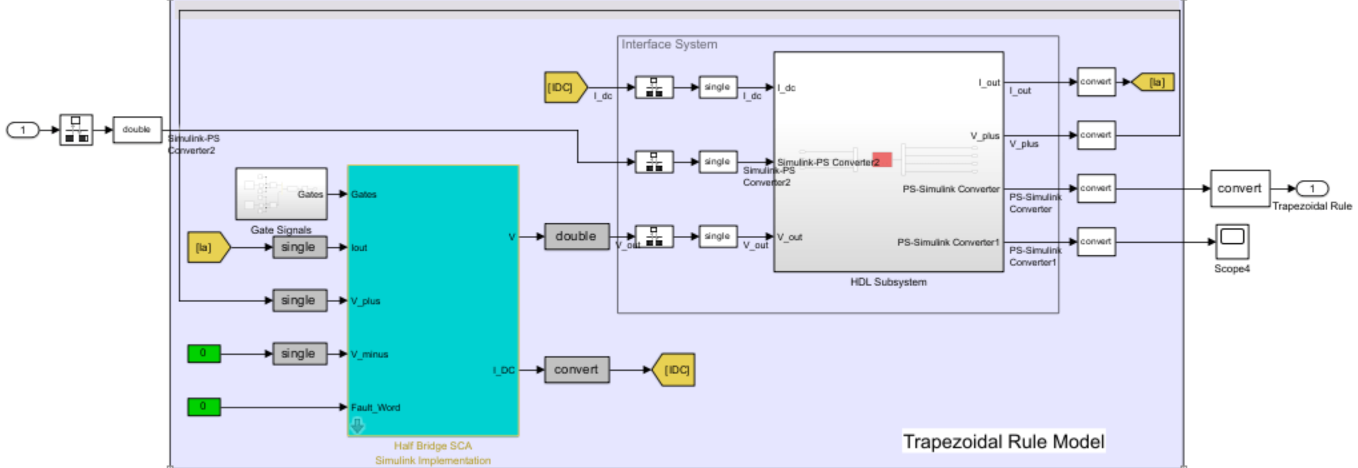
To continue with this example, right-click the Baseline Model, click **Comment Out** and save the model. On the **Modeling** tab, click **Model Settings** and select **Model Settings** from the drop-down menu. In the Solver window, specify the Solver selection options. From the **Type** list, select `Fixed-step`, and from the **Solver** list select `discrete (no continuous states)`. Save the model.

Generate HDL Implementation Model and Validate HDL Algorithm

The Simscape HDL Workflow Advisor converts the Simscape plant model to an HDL-compatible implementation model from which you generate HDL code. To open the Advisor, run the `sschdladvisor` function for your model.

```
sschdladvisor('sschdl_trapezoidal_example')
```

To generate the HDL implementation model, in the **Implementation model generation** task drop-down list, right-click the **Generate implementation model** task, and then select **Run to Selected Task** from the list. After the task passes, you see a link to the HDL implementation model `gmStateSpaceHDL_sschdl_trapezoidal_example`.



To verify that the HDL implementation model matches the original Simscape model, generate a state-space validation model. In the **Generate implementation model** task, select the **Generate validation logic** for the implementation model check box and set the **Validation logic tolerance** to 1e-1. Then, run the task. See “Validate HDL Implementation Model to Simscape Algorithm” on page 30-89.

To modify the configuration parameter values for HDL code generation, navigate to `gmStateSpaceHDL_sschdl_trapezoidal_example/Trapezoidal Rule Model/HDL Subsystem`. Right-click the HDL Algorithm subsystem and select `HDL Block Properties` from the **HDL Code** list. In the HDL Properties dialog box, set the `FlattenHierarchy` to `On`. Click **Apply** and then click **OK**.

HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.

```
hdlsetup('gmStateSpaceHDL_sschdl_trapezoidal_example')
```

- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis and timing analysis.
- Deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

Generate FPGA Bitstream for Speedgoat Target Computer

Use HDL Workflow Advisor to Generate Simulink Real-Time Interface Model

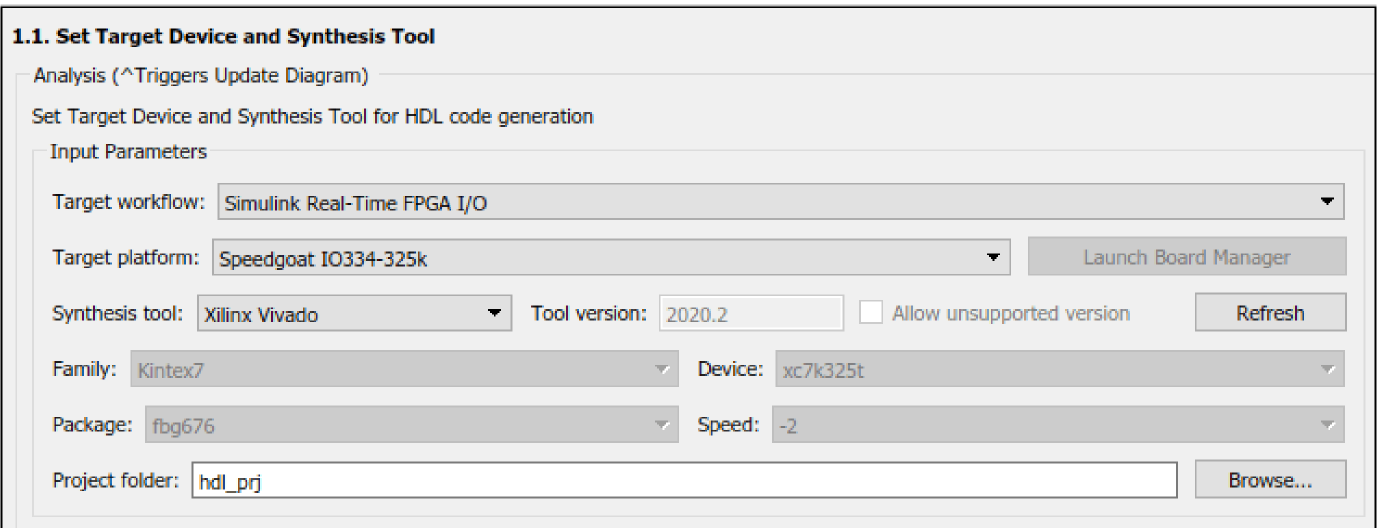
1. Open the HDL implementation model, and then open the HDL Workflow Advisor for the implementation model.

```
open_system('gmStateSpaceHDL_sschdl_trapezoidal_example')
```

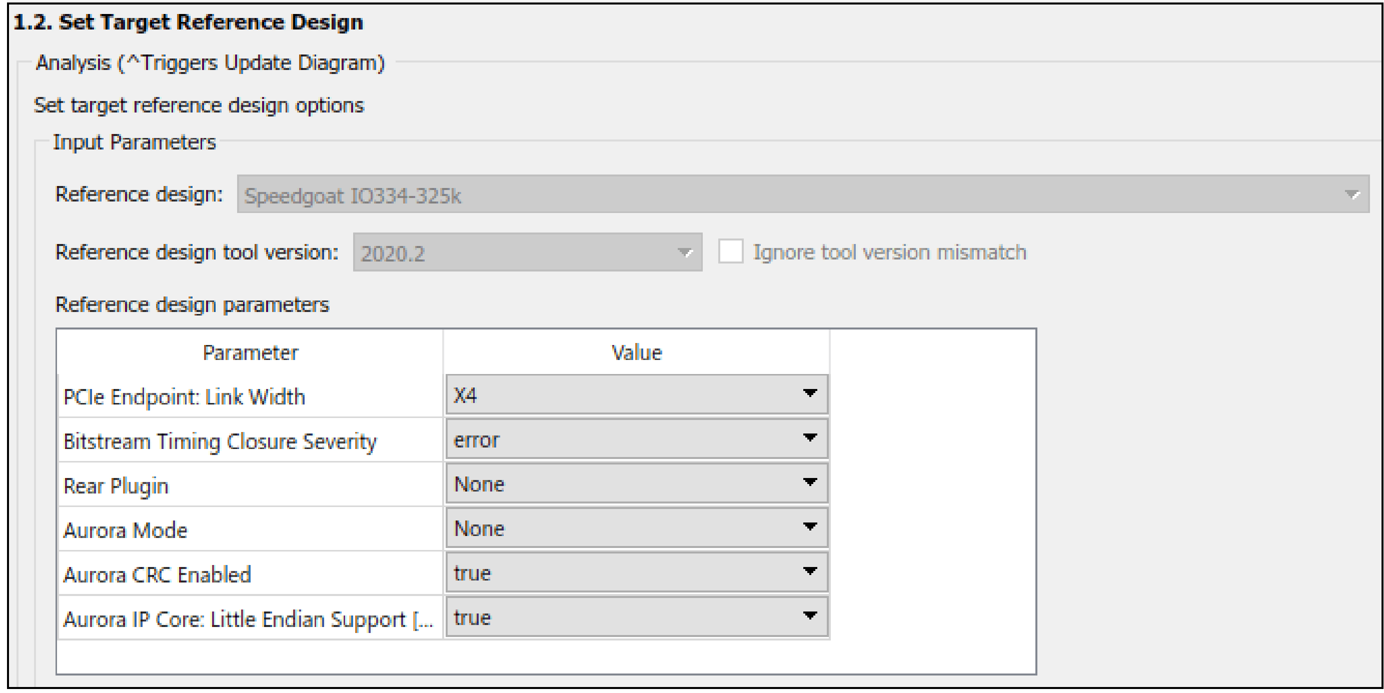
To open the HDL Workflow Advisor for a subsystem inside the model, use the `hdladvisor` function.

```
hdladvisor('gmStateSpaceHDL_sschdl_trapezoidal_example/Trapezoidal Rule Model')
```

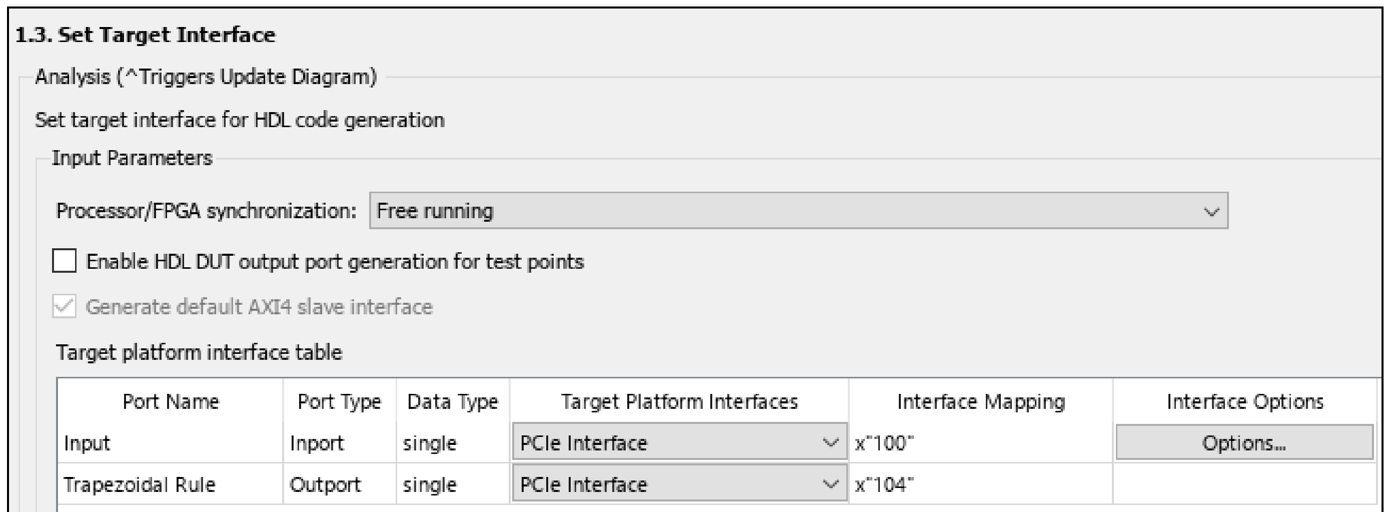
2. In the **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Simulink Real-Time FPGA I/O` and **Target platform** as `Speedgoat IO334-325k`.



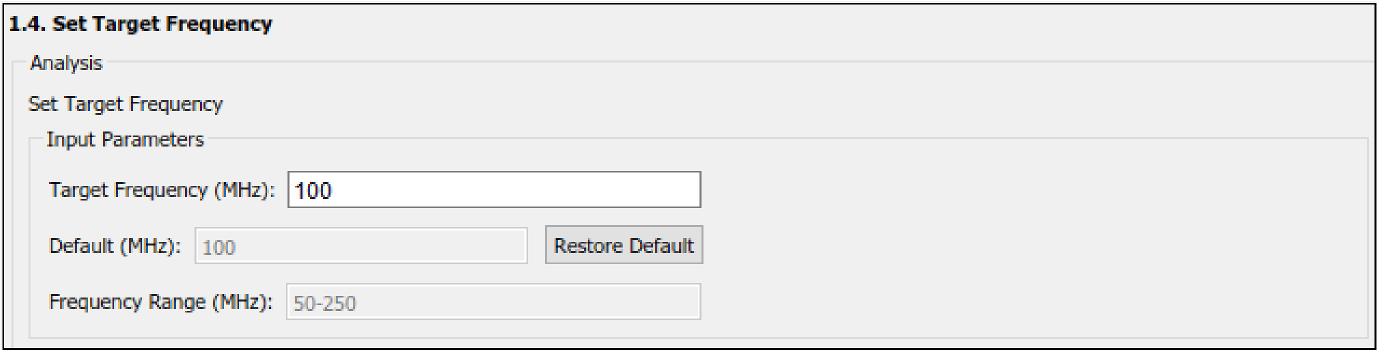
3. In the **Set Target Reference Design** task, select a value of X4 for the parameter `PCIe lanes`, and click the **Run This Task** button.



4. In the **Set Target Interface** task, map the input and output `single` data type ports to `PCIe Interface` and click the **Run This Task** button.



5. In the **Set Target Frequency** task, the default value of **Target Frequency (MHz)** is set to 100. For this example model, this value is same as default value.



6. In the **HDL Code Generation** task folder, click **Set HDL Options** then click the **HDL Code Generation Settings** button. This opens the Configuration Parameters dialog box. Under **HDL Code Generation** > **Global Settings** > **Clock Settings**, set the `Oversampling factor` to 1.

7. Right-click the **Generate Simulink Real-Time Interface** task, and select **Run to Selected Task** to generate the HDL IP core and FPGA bitstream.

Run Workflow Script to Generate Simulink Real-Time Interface Model

You can export the HDL Workflow Advisor settings to a script to expedite and automate your workflow. The script is a MATLAB® file that you run from the command line. You can modify and run the script, or import the settings into the HDL Workflow Advisor User Interface. See “Run HDL Workflow with a Script” on page 29-47.

This example shows how to run the HDL Workflow script. To generate a Simulink Real-Time Interface model, open and run this MATLAB script.

```
edit('hdlworkflow_trapezoidal_IO334')
```

```
%--------------------------------------------------------------------------
% HDL Workflow Script
```

```
% This script contains the model, target settings, interface mapping, and
% the Workflow Configuration settings for generating HDL code for the HDL
% implementation model generated for the Trapezoidal Example Model
% and for deploying the code to the FPGA on board the Speedgoat
% IO334-325K module.
%-------------------------------------------------------------------------

%% Load the Model
load_system('gmStateSpaceHDL_sschdl_trapezoidal_example');

%% Model HDL Parameters
%% Set Model 'gmStateSpaceHDL_ee_buck_converter_dc_motor_' HDL parameters
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'HDLSubsystem', 'gmStateSpaceHDL_ssch
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'Oversampling', 1);
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example/Trapezoidal Rule Model/HDL Subsystem/HDI
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'ReferenceDesign', 'Speedgoat IO334-32
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'SynthesisToolChipFamily', 'Kintex7')
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'SynthesisToolDeviceName', 'xc7k325t'
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'SynthesisToolPackageName', 'fbg676')
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'SynthesisToolSpeedValue', '-2');
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'TargetDirectory', 'C:\SscHdlTrapezoi
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'TargetFrequency', 100);
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'TargetPlatform', 'Speedgoat IO334-325
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example', 'Workflow', 'Simulink Real-Time FPGA I

% Set SubSystem HDL parameters
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example/Trapezoidal Rule Model', 'ProcessorFPGAS

% Set Inport HDL parameters
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example/Trapezoidal Rule Model/Input', 'IOInter
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example/Trapezoidal Rule Model/Input', 'IOInter

% Set Outport HDL parameters
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example/Trapezoidal Rule Model/Trapezoidal Rule
hdlset_param('gmStateSpaceHDL_sschdl_trapezoidal_example/Trapezoidal Rule Model/Trapezoidal Rule

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Simulink Real-Ti

% Specify the top level project directory
hWC.ProjectFolder = 'C:\SscHdlTrapezoidalExample\hdl_prj';
hWC.ReferenceDesignToolVersion = '2020.2';
hWC.IgnoreToolVersionMismatch = true;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndIPCore' Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
```

```
hWC.AdditionalProjectCreationTclFiles = '';
hWC.EnableIPCaching = true;

% Set properties related to 'RunTaskBuildFPGABitstream' Task
hWC.RunExternalBuild = false;
hWC.EnableDesignCheckpoint = false;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;
hWC.CustomBuildTclFile = '';
hWC.DefaultCheckpointFile = 'Default';
hWC.RoutedDesignCheckpointFilePath = '';
hWC.MaxNumOfCoresForBuild = '';

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('gmStateSpaceHDL_sschdl_trapezoidal_example/Trapezoidal Rule Model', hWC);

% Set the sample time in the final SLRT model
set_param('gm_gmStateSpaceHDL_sschdl_trapezoidal_example_slrt/Voltage Control Signal', 'SampleTi
```

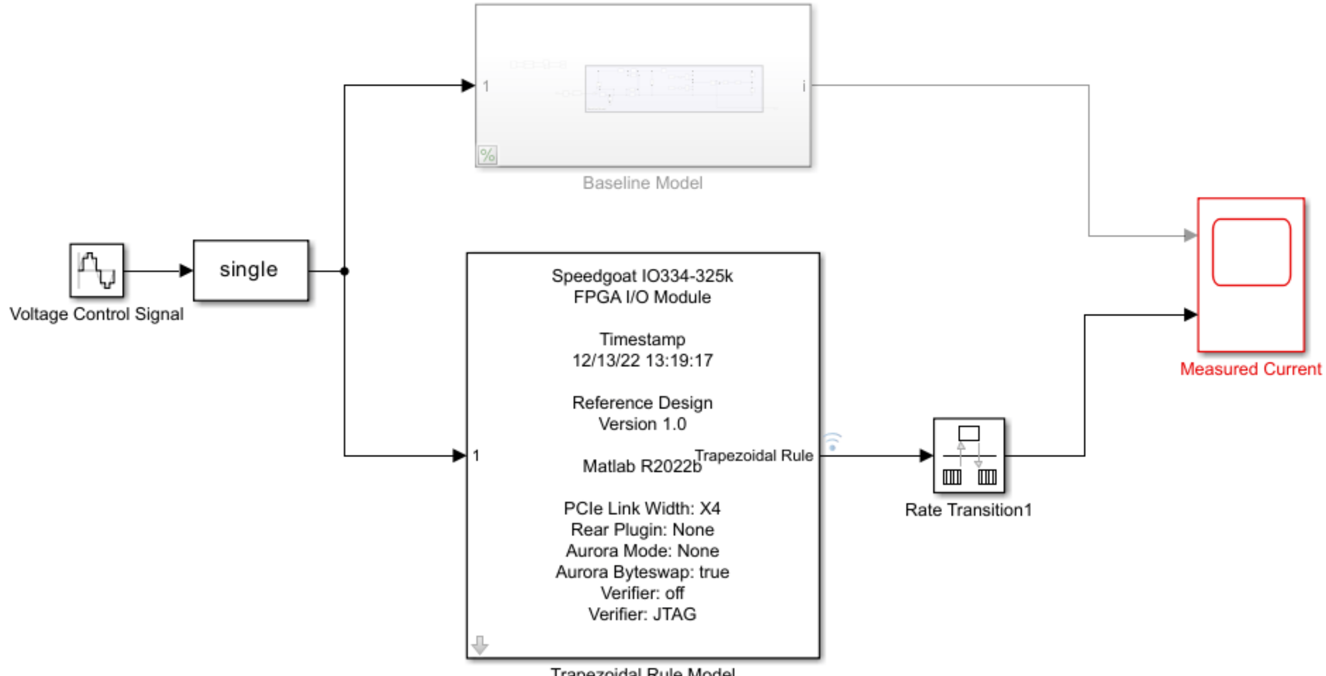
The last line of the script sets the sample time of the Voltage Control Signal block in the generated Simulink Real-Time Interface model to 4e-6.

Deploy Bitstream to Speedgoat IO334-325k Target

Prepare Simulink Real-Time Interface Model for Real-Time Simulation

Running the workflow script generates RTL code and IP core, creates a Vivado project, builds the FPGA bitstream, and then generates the Simulink Real-Time interface model.

The FPGA subsystem is automatically replaced in the real-time interface model with Speedgoat driver blocks to initialize the hardware and to interface with the FPGA during runtime.



Make sure you comment out any remaining `Rate Transition` blocks in the model to obtain a single rate model. All the blocks in the generated Simulink Real-Time model are executed on the CPU of the real-time system. The induction stovetop model is programmed in the bitstream and run at the faster rate on the FPGA.

Connect to Target Machine and Run Real-Time Simulation

The model can now be deployed onto the Speedgoat real-time target machine. Make sure that the Speedgoat real-time target machine is connected to the host computer and powered on.

You download the bitstream by using the Simulink Real-Time Explorer. To open the Simulink Real-Time Explorer, enter the command `slrtExplorer`. Alternatively, you can open the Explorer from the **Real-Time** tab of the Simulink Toolstrip.

```
slrtExplorer
```

The Simulink Editor displays the **Real-Time** tab for models that are configured for the `speedgoat.tlc` code generation target. Click the **Connect to Target Computer** button in the Simulink **Real-Time** tab to connect to the machine. Once connected, click the **Run on Target** button to deploy the model.

Simulink Real-Time automatically generates C code from your model by using Simulink Coder. The generated code and the bitstream for the FPGA are loaded onto the target machine, and model execution starts automatically.

See Also

Functions
`sschdladvisor` | `hdladvisor` | `hdlsetup`

More About

- "Generate HDL Code for Simscape Models" on page 30-13
- "Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules" on page 30-82
- "Deploy Simscape Grid Tied Converter Model to Speedgoat IO Module Using HDL Workflow Script" on page 30-47
- "Making Optimal Solver Choices for Physical Simulation" (Simscape)
- "Generate HDL Code for Nonlinear Simscape Models by Using Partitioning Solver" on page 30-100

Generate HDL Code for Simscape Models by Using Linearized Switch Approximation

This example shows how to generate HDL code for a Simscape™ model by using linearized switch approximation method.

Simscape Model with Linearized Switch Approximation

In this example, you learn how you can use the linearized switch approximation method to convert the Simscape™ three-phase permanent magnet synchronous motor (PMSM) model to an HDL implementation model for HDL code generation and synthesis. First, you replace the ideal insulated-gate bipolar transistor IGBT (Ideal, Switching) (Simscape Electrical) with linearized equivalents and generate an HDL implementation model by using the Simscape HDL Workflow Advisor. Then, for this implementation model, you generate the HDL code and synthesize the results by using the guided steps in the HDL Workflow Advisor. For more information, see “HDL Workflow Advisor Tasks” on page 36-2. You can then deploy the generated HDL code onto a Speedgoat® FPGA I/O module.

The linearized switch approximation method supports all local solvers. The three-phase inverter is modelled using linearized switch approximation method. The linearized switch approximation method provides an improved FPGA sample rate, reduced resource utilization, and dead time stability, and prevents validation errors. For more information on validation errors, see “Troubleshoot Validation Errors in Simscape Hardware-in-the-Loop Workflow” on page 32-9.

Set Up Synthesis Tool Path

To synthesize the generated HDL code, set up your synthesis tool path before you use HDL Coder™ to generate code. For example, if your synthesis tool is Xilinx® Vivado®, install the latest version of Xilinx Vivado as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function:

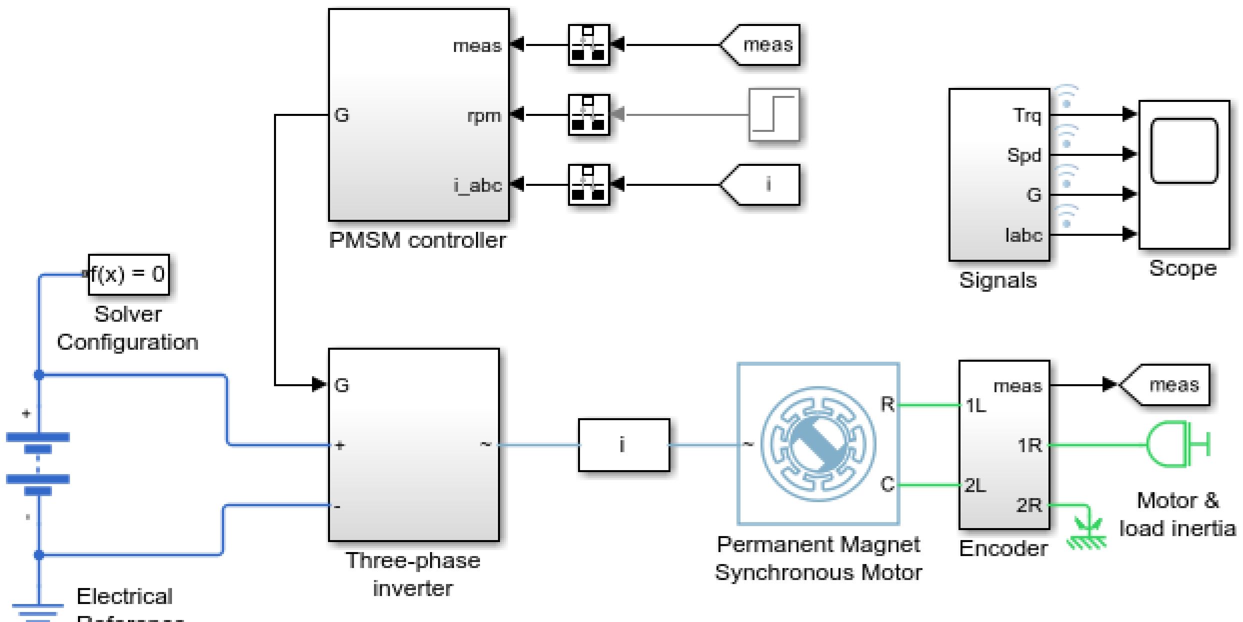
```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

Three-Phase PMSM Drive

The model used in this example contains a three-phase PMSM in wye-wound configuration and a three-phase inverter. This model uses field-oriented control (FOC) to control the speed of the three-phase PMSM. The PMSM is a nonlinear block in the model. The inverter is connected to the battery. The `Three-phase inverter` subsystem has the switching elements as IGBTs. This model uses the Partitioning solver as a local solver.

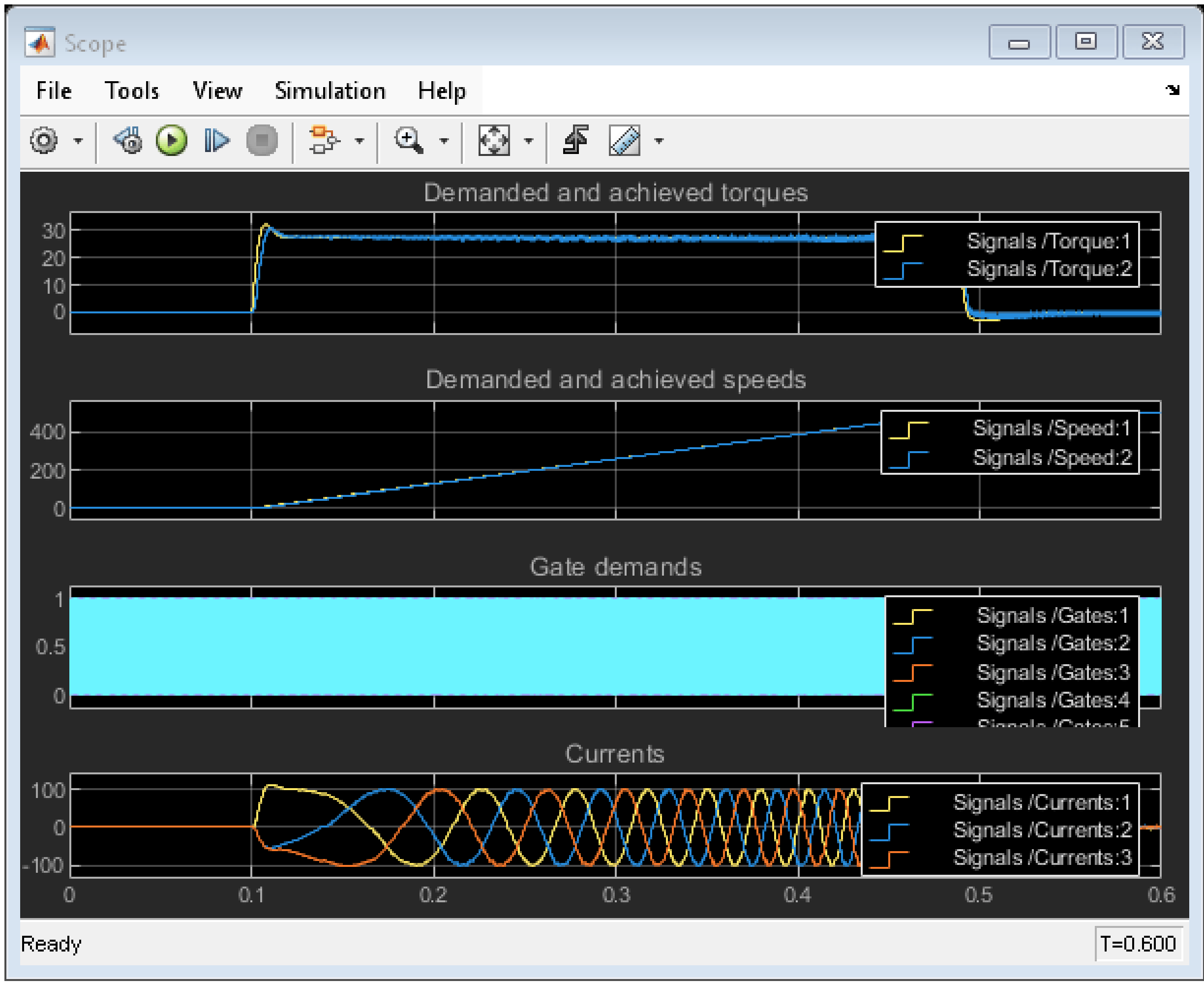
Open the model at the MATLAB® command prompt.

```
ModelName = 'sschdlexThreePhasePMSMDrive';
open_system(ModelName)
```



To see the waveforms, simulate the model.

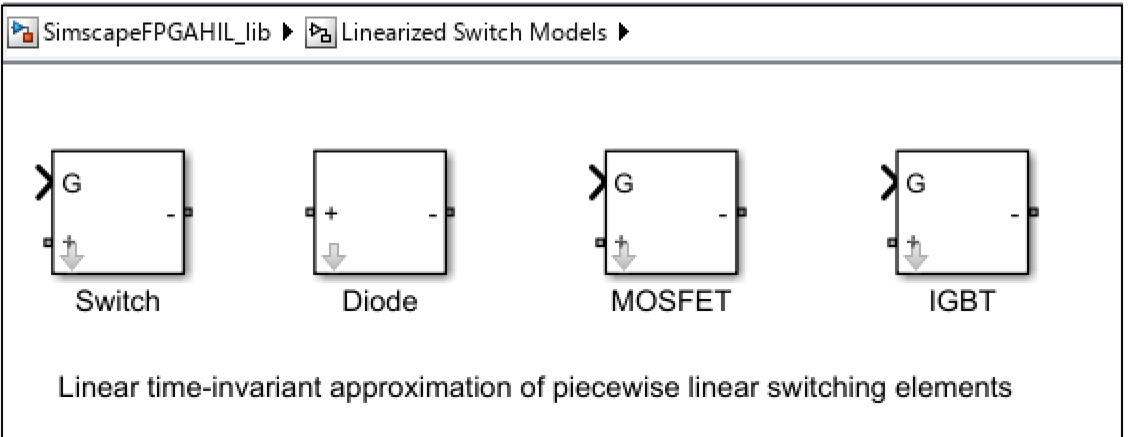
```
sim(ModelName)
open_system([ModelName '/Scope'])
```



PMSM Model Modified for Linearized Switch Approximation

You can modify this model for linearized switch approximation by replacing the switches (IGBTs) with linearized equivalents. To modify the model, double-click the `Three-phase inverter` subsystem and replace each IGBT with a subsystem containing an IGBT and a protection diode. The `SimscapeFPGAHIL_lib` library has the IGBTs and diodes compatible for HDL code generation (Simscape HIL simulation). You can open the library from MATLAB command window. To open this library, enter:

```
SimscapeFPGAHIL_lib
```



The library window contains the `Linearized Switch Models` subsystem. This subsystem contains linear time-invariant (LTI) approximations of the piecewise linear switching elements. Drag the IGBT block and the Diode block into the `Three-phase inverter` subsystem of your PMSM model and create a subsystem for each IGBT block and name the subsystems as `IGBT+Diode`.

The IGBT block consists of a resistor and a controlled current source. Double-click the IGBT block in the `IGBT+Diode1` subsystem to update the values of its parameters. The **Conductance (Gs)** is the conductance of the resistor in `Linearized Switch Models`. Gs is different from the off-state conductance ($Goff$) of an IGBT when it is used for switching applications. The **Time scaling** s scales the speed of the switching device response with respect to time. For the IGBT block,

- Set **Forward Voltage (V)** to 0.8.
- Set **Threshold Voltage (V)** to 0.5.
- Set **Conductance (1/Ohm)** Gs to 4.4526.
- Set **Time scaling** s to 2.8994.
- Set **Discrete sample time (s)** h to 2e-06.

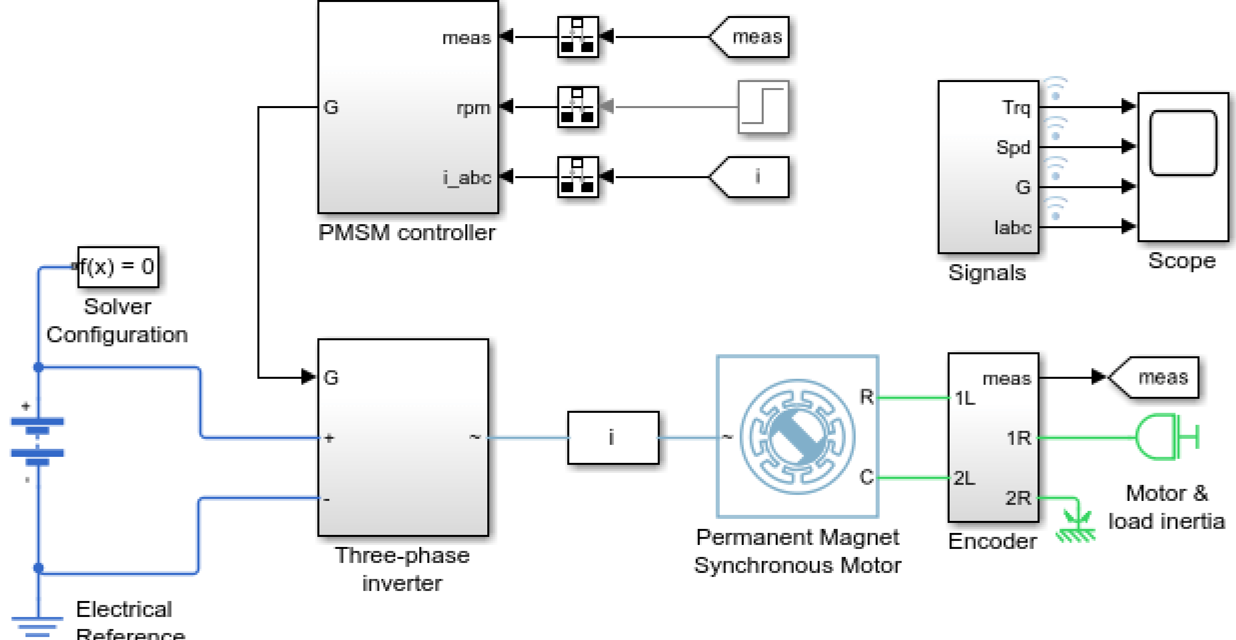
Then, double-click the Diode block in the `IGBT+Diode` subsystem to update the different parameter values. For the Diode block,

- Set **Forward Voltage (V)** to 0.8.
- Set **Conductance (1/Ohm)** Gs to 4.4526.
- Set **Time scaling** s to 2.8994.
- Set **Discrete sample time (s)** h to 2e-06.

Similarly, update the parameter values for the other IGBT+Diode subsystems. For this example, the values of Gs and s are calibrated to 4.4526 and 2.8994, respectively, by using the **Parameter Estimator** app. If you have a license for Simulink® Design Optimization™ toolbox, you can use the **Parameter Estimator** app to tune the parameter values. You can also calibrate these values manually.

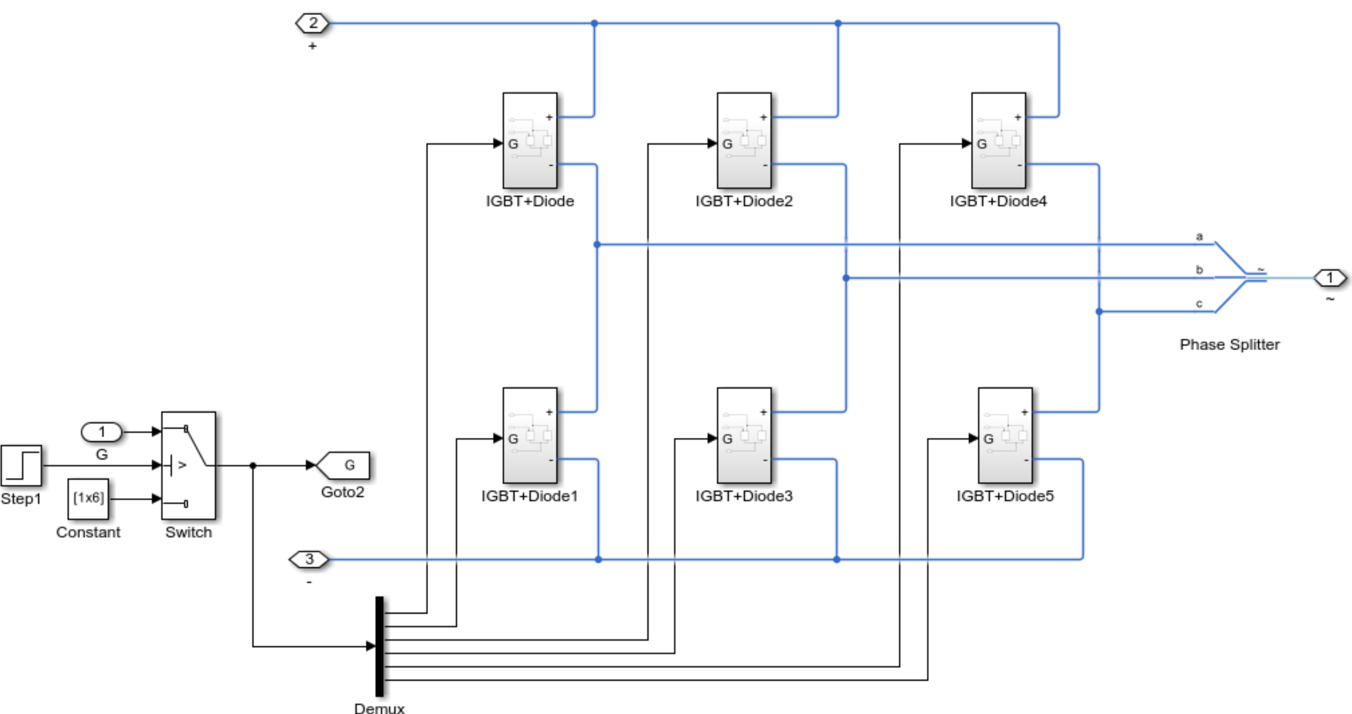
To open the model modified in this way, at the MATLAB command prompt, enter:

```
ModelNameLSA = 'sschdlexPMSMLinearizedSwitches';
open_system(ModelNameLSA)
```



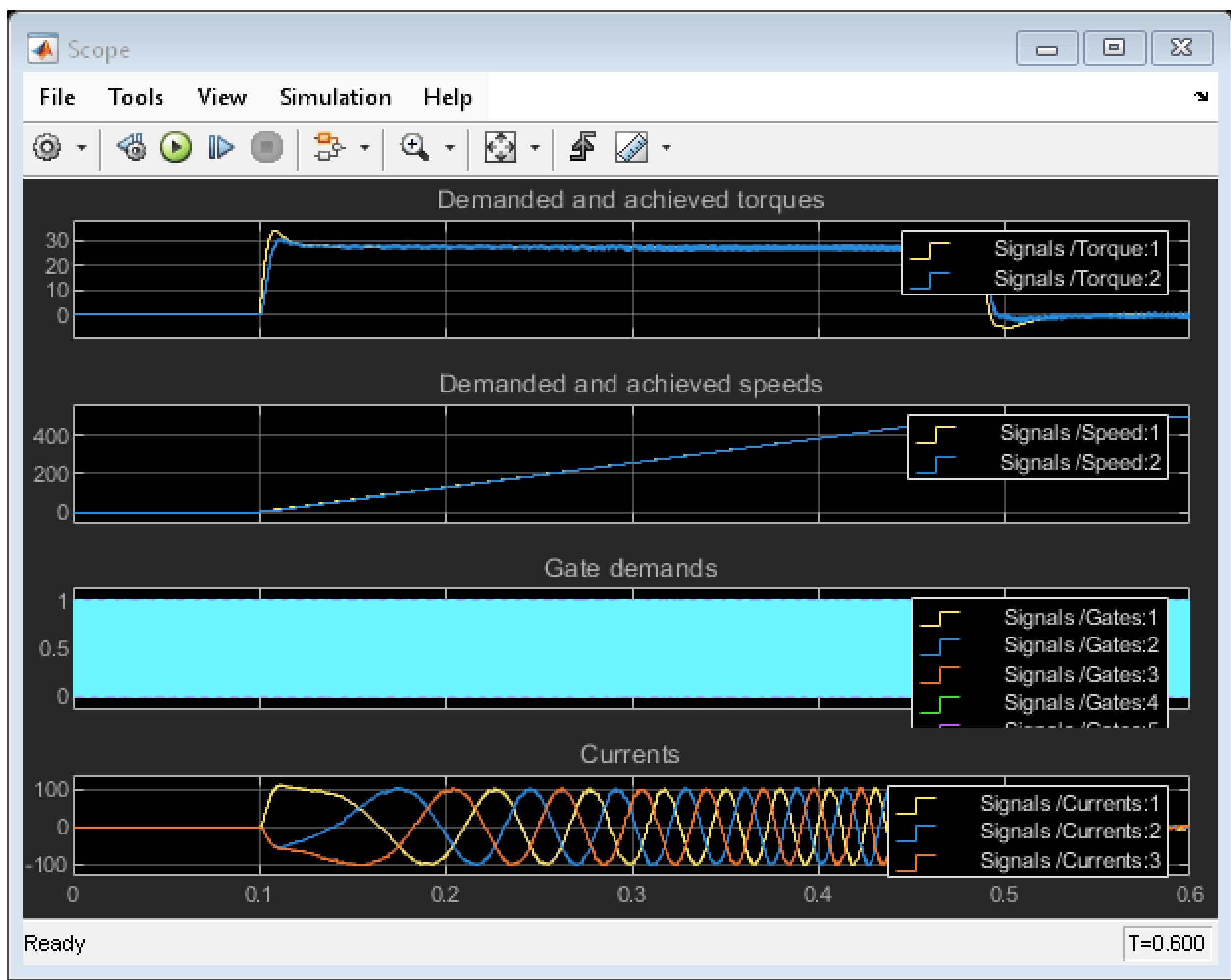
To see the modified subsystem, enter:

```
open_system([ModelNameLSA,'/Three-phase inverter'])
```



When you simulate this model, the results of this model and the original Simscape model are same. To see how the model works, simulate the model.

```
sim(ModelNameLSA)
open_system([ModelNameLSA '/Scope'])
```



Generate HDL Implementation Model

The Simscape HDL Workflow Advisor converts the Simscape plant model to an HDL-compatible implementation model from which you generate HDL code. To generate the HDL implementation model:

1. Open the Simscape HDL Workflow Advisor.

```
sschdladvisor('sschdlexPMSMLinearizedSwitches')
```

2. In the **Implementation model generation** task folder, right-click the **Generate implementation model** task, then select **Run to Selected Task**. To get better resource utilization, in the **Generate implementation model** task window, set **Map state space parameter to RAM** to `On`.

After the task passes, you see a link to the HDL implementation model `gmStateSpaceHDL_sschdlexPMSMLinearizedSwitc`.

Generate HDL Code from Implementation Model

To modify the configuration parameter values for HDL code generation, enter this command at the MATLAB command prompt.

```
hdlsetup('gmStateSpaceHDL_sschdlexPMSMLinearizedSwitc')
```

Open HDL Workflow Advisor

The HDL Workflow Advisor guides you through the tasks required for generating HDL code and an FPGA design process. It provides you with feedback on the results of each task. When you complete the tasks, you have a synthesis result report from one of the supported synthesis tools.

To open the subsystem `HDL Subsystem` in the HDL implementation model into the HDL Workflow Advisor, enter this command at the MATLAB command prompt.

```
hdladvisor('gmStateSpaceHDL_sschdlexPMSMLinearizedSwitc/HDL Subsystem')
```

You can also open the HDL Workflow Advisor from your model window. Right-click the subsystem `HDL Subsystem` and select `HDL Workflow Advisor` from the list.

Set Target Device and Synthesis Tool

Before you generate HDL code, if you want to deploy the code onto a target platform, specify the synthesis tool.

1. Open the HDL Workflow Advisor.
2. Under the **Set Target** task folder, in the **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Generic ASIC/FPGA` and **Synthesis tool** as `Xilinx Vivado`. The rest of the fields are auto-populated. Specify the **Family** as `Kintex7`, **Device** as `xc7k325t`, **Package** as `fbg676`, and **Speed** as -1.
3. In the **Set Target Frequency** task, specify the **Target Frequency** as 200.
4. Select the task that you want to run and click **Run This Task**.

Generate HDL Code

- In the **HDL Code Generation** task folder, select **Set HDL Options**, then click the **HDL Code Generation Settings** button to open the Configuration Parameters dialog box. Select `Adaptive pipelining` under **HDL Code Generation** > **Optimizations** > **Pipelining**. Click **Apply**.
- Under the **HDL Code Generation** > **Global Settings** > **Clock Settings** section, set the `Oversampling factor` to 273. Click **Apply**, then click **OK**. To generate HDL code, run the tasks under the **HDL Code Generation** task folder.

Synthesize Generated HDL Code

HDL Coder synthesizes the HDL code on the target platform and generates area and timing reports for your design based on the target device that you specify. You can run logic synthesis for a specified FPGA device and get the synthesis reports.

In the **FPGA Synthesis and Analysis** task folder:

- Create an FPGA synthesis project for your supported FPGA synthesis tool.
- Start supported FPGA synthesis tools to perform synthesis, mapping, and place/route tasks. To run FPGA synthesis, right-click the **Run Synthesis** task under the **Perform Synthesis and P/R**

subtask folder. This starts Xilinx Vivado and executes the Vivado **Synthesis** step. You can annotate your original model with critical path information obtained from the synthesis tools.

Passed Synthesis

Parsed resource report file: HDL_Subsystem_utilization_synth.rpt.

Resource summary			
Resource	Usage	Available	Utilization (%)
Slice LUTs	1.4654e+05	203800	71.90
Slice Registers	1.1248e+05	407600	27.59
DSPs	221	840	26.31
Block RAM Tile	92	445	20.67
URAM	0	0	

Parsed timing report file: timing_post_map.rpt.

Timing summary	
	Value
Requirement	5 ns (200 MHz)
Data Path Delay	4.884 ns
Slack	0.084 ns
Clock Frequency	203.42 MHz

Deploy Three-Phase PMSM Drive to Speedgoat FPGA I/O Modules

In the HDL implementation model, the `HDL Subsystem` contains blocks you run on the FPGA. You can run the HDL Workflow Advisor on this subsystem to deploy the HDL algorithm onto FPGA boards in Speedgoat target computers. For an example, see “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82.

References

[1] Pejovic, P., and D. Maksimovic. “A Method for Fast Time-Domain Simulation of Networks with Switches.” *IEEE Transactions on Power Electronics* 9, no. 4 (July 1994): 449–56, https://doi.org/10.1109/63.318904.

See Also

Functions
`sschdladvisor` | `hdladvisor` | `hdlsetup`

Related Examples

- “Generate HDL Code for Simscape Models” on page 30-13
- “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82
- “Generate HDL Code for Nonlinear Simscape Models by Using Partitioning Solver” on page 30-100

- “Generate HDL Code for Simscape Models by Using Trapezoidal Rule Solver” on page 30-135
- “Generate HDL Code for Simscape Three-Phase PMSM Drive Containing Averaged Switch” on page 30-118
- “Deploy Simscape DC Motor Model to Speedgoat FPGA IO Module” on page 30-106

Estimate Achievable Target Frequency Without Running Synthesis

By using the Simscape HDL Workflow Advisor, you can estimate the optimal frequency that you want your Simscape models to achieve on FPGA without running synthesis. This method of estimation helps you to run the model on the hardware without any mismatches and provides information about whether the Simscape model can achieve the required time step on FPGA without running synthesis.

The estimation is based on the **Sample time** parameter you specify in the Solver Configuration block of your model. If the calculated FPGA sample time matches the specified sample time, the Advisor automatically sets **Target Frequency (MHz)** in the Configuration Parameters dialog box and treats the model rates as hardware rates. However, if the FPGA sample time does not match the specified sample time, the Advisor displays a warning message for the generated HDL implementation model.

The estimation is limited to Simscape models containing a single Simscape network.

Example Models with Estimated Sample Time

The table lists the Simscape Hardware-in-the-Loop (HIL) example models with their respective estimated sample times on a Windows® 11 Intel Xeon® W-2133 CPU @3.60GHz test system with target device family set to Xilinx Vivado Virtex 7.

Example	Required Sample Time (μs)	Estimated Achievable Sample Time (μs)	True FPGA Sample Time (μs)
Half-wave rectifier on page 30-13	1	0.15	0.116
Buck converter on page 30-9	1	0.58	0.49
Boost converter on page 30-96	1	1.04	0.493
Bridge rectifier on page 30-89	10	0.694	0.38
DC motor on page 30-106	1000	1.011	0.9636

See Also

“Troubleshooting Real-Time Hardware Deployment Issues in Simscape Hardware-in-the-Loop Workflow” on page 32-2 | Optimal Sharing Factor Supported FPGA Device Families on page 31-11

Related Examples

- “Generate Optimized HDL Implementation Model from Simscape” on page 30-20
- “Deploy Simscape DC Motor Model to Speedgoat FPGA IO Module” on page 30-106

Generate FPGA Bitstream for Two-Phase DC-DC Converter with Tunable Run-Time Parameters

This example shows how to generate FPGA bitstream for a Simscape™ model and tune the Simscape run-time parameters without rerunning the synthesis.

Simscape Model with Tunable Run-Time Parameters

In this example, you learn how you can generate FPGA bitstream for Simscape two-phase interleaved bidirectional DC-DC converter model with tunable run-time parameters. First, you generate an HDL implementation model from a Simscape two-phase DC-DC converter model by using the Simscape HDL Workflow Advisor. Then, for this implementation model, you generate the HDL code and synthesize the results by using the guided steps in the HDL Workflow Advisor. For more information, see “HDL Workflow Advisor Tasks” on page 36-2. You can tune the run-time parameter values for the Simscape model without rerunning the synthesis.

The parameter tuning supports:

- Linear time-invariant models with local solver set to `Backward Euler`
- Simscape models containing only single Simscape network
- `Single` data type precision

Set Up Synthesis Tool Path

To synthesize the generated HDL code, before you use HDL Coder™ to generate code, set up your synthesis tool path. For example, if your synthesis tool is Xilinx® Vivado®, install the latest version of Xilinx Vivado as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function:

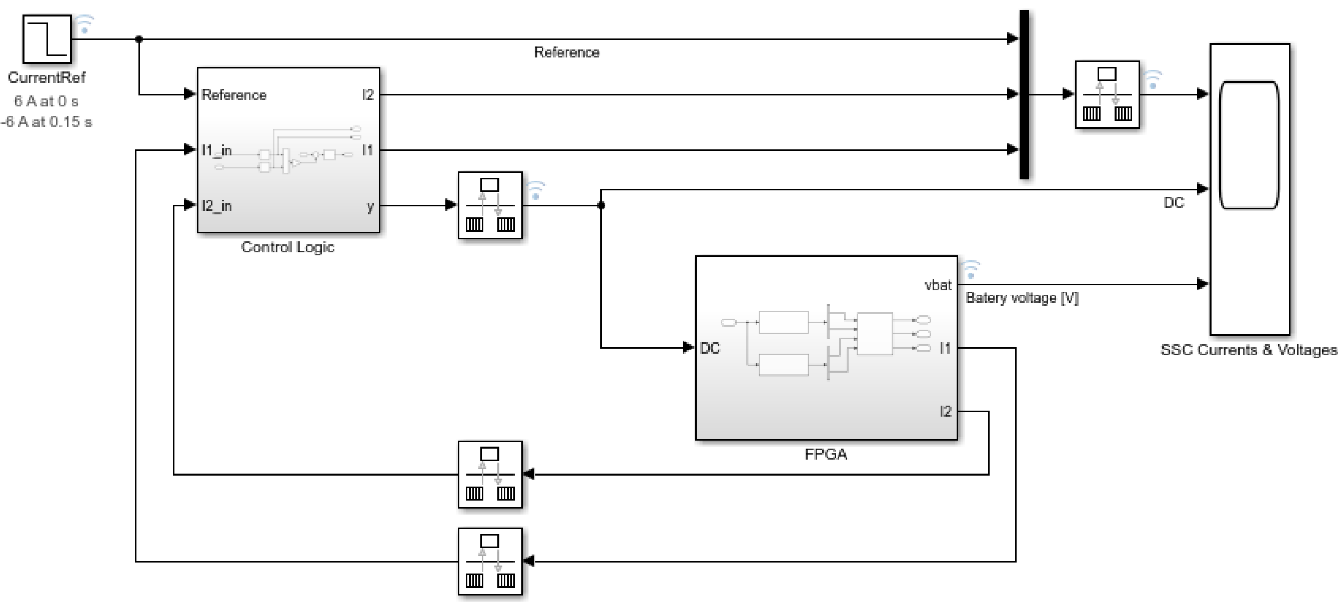
```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2022.1\bin\vivado.bat')
```

Two-Phase DC-DC Converter Current Control Model

The two-phase converter comprises two Bidirectional DC-DC Converters with ideal IGBTs. These configurations are a part of Simscape network inside the `FPGA` subsystem. To adjust the duty cycle, the `Control Logic` subsystem uses a PI-based control algorithm. To reduce the ripple at the output port of the converter, the two phases are switched with the same duty ratio but with a relative phase shift of 180 degrees. The `SSC Currents & Voltages` subsystem contains scopes that allow you to see the simulation results.

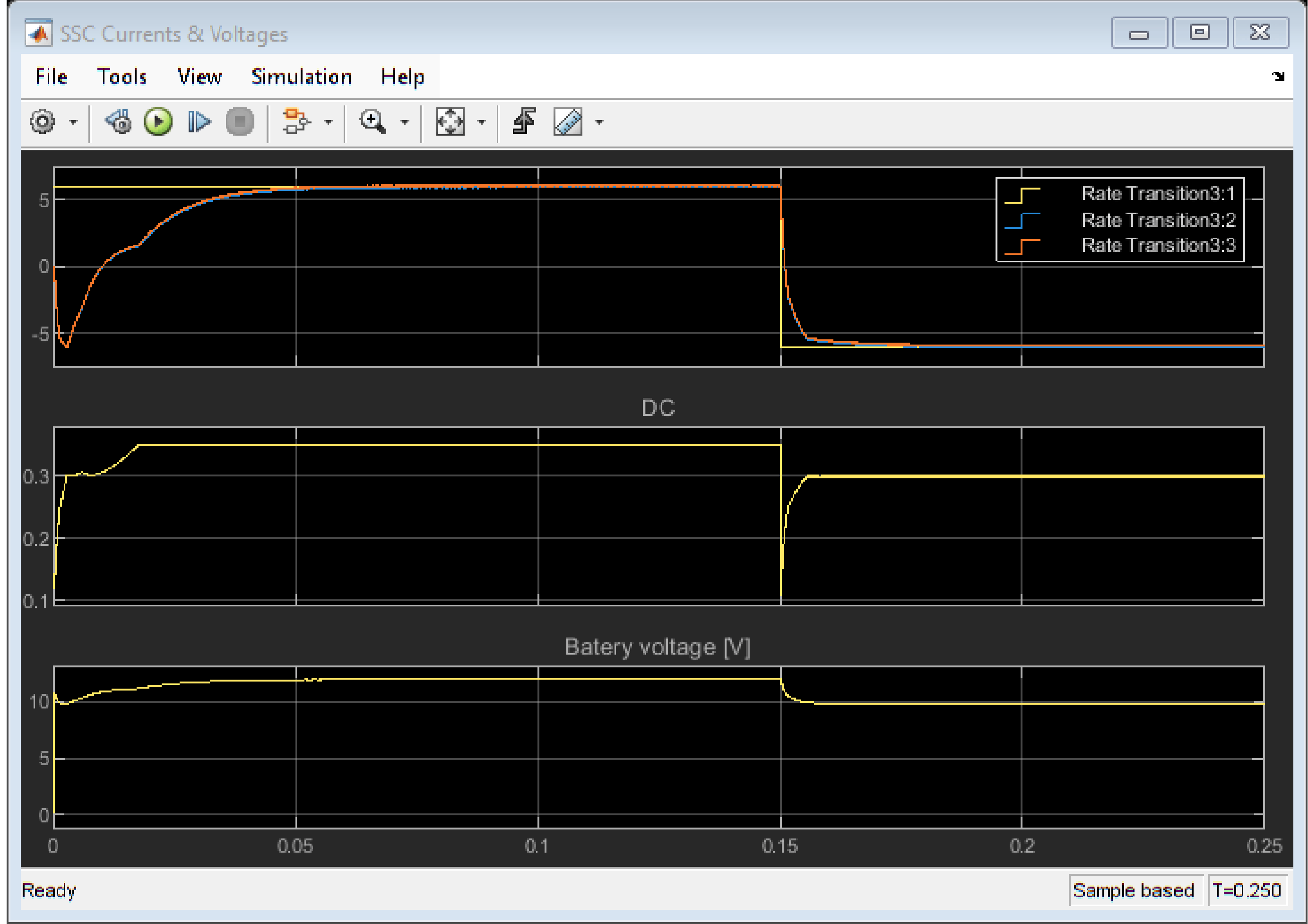
Open the model in the MATLAB® Command Window.

```
ModelName = 'sschdlexTwoPhaseDCDCConverterExample';
open_system(ModelName)
```



To see the waveforms, simulate the model.

```
sim(ModelName)
open_system([ModelName '/SSC Currents & Voltages'])
```



Set Up Tunable Parameters

You can set up a model for parameter tuning in two different ways.

- Make Simscape run-time parameters run-time configurable — To specify a Simscape block parameter as run-time configurable, change the **Configurability** setting in the block property inspector underneath the parameter name, from `Compile-time` to `Run-time`.
- Use the `Dynamic Switch Library` blocks — For the blocks in `Dynamic Switch Library`, the run-time configurability is not a mask parameter. You can set any mask parameter to be a `Simulink.Parameter` object by setting `obj.CoderInfo.StorageClass` equal to `"ExportedGlobal"`. This setup enables parameter tuning for the corresponding parameter. In the model, navigate to `FPGA` > `Simscape Network` > `Bidirectional DC-DC Converter` subsystem and double-click the Switch block. Here, `Gs` and `s` are `Simulink.Parameter` objects.

Once you set up the tunable parameter settings in the model, save the model for generating HDL implementation model from it.

Generate HDL Implementation Model

The Simscape HDL Workflow Advisor converts the Simscape plant model to an HDL-compatible implementation model from which you generate HDL code. To generate the HDL implementation model:

1. Open the Simscape HDL Workflow Advisor.

```
sschdladvisor('sschdlexTwoPhaseDCDCConverterExample')
```

2. For each task, right-click and select **Run This Task**. In the **Extract discrete equations**, the right pane displays a table for all tunable parameters.

3. In the **Implementation model generation** task folder, click **Generate implementation model** task and set **Map state space parameter to RAM** to `On` in the **Generate implementation model** task window to get better resource utilization. Then, right-click the **Generate implementation model** task and select **Run to Selected Task**.

After the task passes, you see a link to the HDL implementation model `gmStateSpaceHDL_sschdlexTwoPhaseDCDCConvert`.

Generate HDL Code from Implementation Model

To modify the configuration parameter values for HDL code generation, run this command:

```
hdlsetup('gmStateSpaceHDL_sschdlexTwoPhaseDCDCConvert')
```

Open HDL Workflow Advisor

The HDL Workflow Advisor guides you through the tasks required for generating HDL code and an FPGA design process. It provides you with feedback on the results of each task. When you complete the tasks, you have a synthesis result report from one of the supported synthesis tools.

To open the subsystem `HDL Subsystem1` in the HDL implementation model into the HDL Workflow Advisor, run this command:

```
hdladvisor('gmStateSpaceHDL_sschdlexTwoPhaseDCDCConvert/FPGA')
```

You can also open the HDL Workflow Advisor from your model window. Right-click the subsystem `HDL Subsystem1` and select **HDL Workflow Advisor**.

Set Target Device and Synthesis Tool

Before you generate HDL code, if you want to deploy the code onto a target platform, specify the synthesis tool.

1. Open the HDL Workflow Advisor.
2. Under the **Set Target** task folder, in the **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Generic ASIC/FPGA` and **Synthesis tool** as `Xilinx Vivado`. The rest of the fields are auto-populated. Specify the **Family** as `Kintex7`, **Device** as `xc7k325t`, **Package** as `fbg676`, and **Speed** as `-1`.
3. In the **Set Target Frequency** task, specify **Target Frequency** as 140.
4. Select the task that you want to run and click **Run This Task**.

Generate HDL Code

1. In the **HDL Code Generation** task folder, select **Set HDL Options** and then click the **HDL Code Generation Settings** button to open the Configuration Parameters dialog box. Select `Adaptive pipelining` under **HDL Code Generation** > **Optimizations** > **Pipelining**. Click **Apply**.
2. To modify the configuration parameter values for HDL code generation, navigate to `gmStateSpaceHDL_sschdlexTwoPhaseDCDCConvert/FPGA/Subsystem3/HDL Subsystem1/HDL Algorithm/State Update`. Right-click the `Multiply Input` block and select **HDL Code** > **HDL Block Properties** from the list. In the HDL Properties dialog box, set the value of **SharingFactor** to `20`. Click **Apply** and then click **OK**. For the `Multiply State` block, also set the value of **SharingFactor** to `20`. Similarly, in the `HDL Algorithm` subsystem, set the value of **SharingFactor** to `10` for the `Multiply State` and `Multiply Input` blocks of the `Output` subsystem.
3. To generate HDL code, run the tasks under the **HDL Code Generation** task folder.

Synthesize Generated HDL Code

HDL Coder synthesizes the HDL code on the target platform and generates area and timing reports for your design based on the target device that you specify. You can run logic synthesis for a specified FPGA device and get the synthesis reports.

In the **FPGA Synthesis and Analysis** task folder:

1. Create an FPGA synthesis project for your supported FPGA synthesis tool.
2. Start supported FPGA synthesis tools to perform synthesis, mapping, and place/route tasks. To run FPGA synthesis, right-click the **Run Synthesis** task under the **Perform Synthesis and P/R** subtask folder. This starts Xilinx Vivado and executes the Vivado **Synthesis** step. You can annotate your original model with critical path information obtained from the synthesis tools.

Passed Synthesis

Parsed resource report file: HDL_Subsystem1_utilization_synth.rpt.

Resource summary			
Resource	**Usage**	**Available**	**Utilization (%)**
Slice LUTs	1.2842e+05	203800	63.01
Slice Registers	1.3443e+05	407600	32.98
DSPs	77	840	9.17
Block RAM Tile	0	445	0.00
URAM	0	0	

Parsed timing report file: timing_post_map.rpt.

Timing summary	
	Value
Requirement	7.1429 ns (140 MHz)
Data Path Delay	1.644 ns
Slack	2.276 ns
Clock Frequency	205.47 MHz

Tune Run-Time Parameters

You can update the run-time parameter values of the generated HDL implementation model and fine-tune it. To generate and update tunable parameter data file for the implementation model:

1. Reopen the original Simscape model and update the parameter values to the desired configuration.

2. In MATLAB Command Window, run the `sschdl.updateRuntimeParameters` function:

```
sschdl.updateRuntimeParameters('sschdlexTwoPhaseDCDCConverterExample',...
'sschdl\sschdlexTwoPhaseDCDCConverterExample\stateSpaceParameters',...
'sschdl\sschdlexTwoPhaseDCDCConverterExample\tunableParameters',
'myTunableParams');
```

This generates a new data file with the specified name as `myTunableParams.mat` in the generated `sschdl` folder. Use this file in place of `tunableParameters.mat` file to update your Simulink or SLRT model parameter values. To update the desired file in the Simulink® or SLRT model, open the Simulink or SLRT model. On the **Modeling** tab, select **Model Settings > Model Properties**. On the **Callbacks** tab, in the **Model callback** pane, select `InitFcn` and load the `tunableParameter.mat` file with the newly generated `myTunableParams.mat` file.

Deploy Two-Phase DC-DC Converter to Speedgoat FPGA I/O Modules

Generate FPGA Bitstream for Speedgoat Target Computer

1. Open the HDL implementation model, and then open the HDL Workflow Advisor for the implementation model.

```
open_system('gmStateSpaceHDL_sschdlexTwoPhaseDCDCConvert')
```

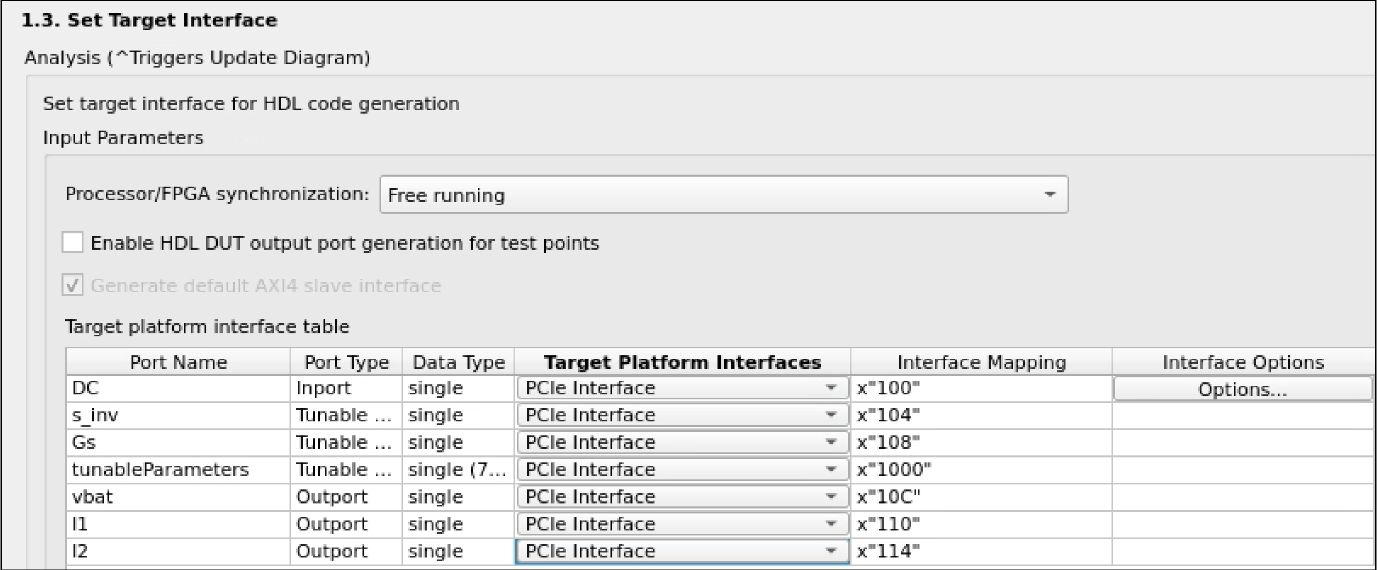
To open the HDL Workflow Advisor for a subsystem inside the model, use the `hdladvisor` function.

```
hdladvisor('gmStateSpaceHDL_sschdlexTwoPhaseDCDCConvert/FPGA')
```

2. In the **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Simulink Real-Time FPGA I/O` and **Target platform** as `Speedgoat IO334-325K`.

3. In the **Set Target Reference Design** task, select a value of `X4` for the parameter `PCIe lanes`, and click the **Run This Task** button.

4. In **Set Target Interface** task, map the input and output `single` data type ports to `PCIe Interface` and click the **Run This Task** button.



5. In the **Set Target Frequency** task, the default value of **Target Frequency (MHz)** is set to `100`. For this example model, this value is `140`.

6. Right-click the **Generate Simulink Real-Time Interface** task, and select **Run to Selected Task** to generate the HDL IP core and FPGA bitstream.

Deploy Bitstream to Speedgoat IO334-325k Target

The FPGA subsystem is automatically replaced in the real-time interface model with Speedgoat® driver blocks to initialize the hardware and to interface with the FPGA during run time.

Connect to Target Machine and Run Real-Time Simulation

The model can now be deployed onto the Speedgoat real-time target machine. Make sure that the Speedgoat real-time target machine is connected to the host computer and powered on.

You download the bitstream by using the Simulink Real-Time Explorer. To open the Simulink Real-Time Explorer, enter the command `slrtExplorer`. Alternatively, you can open the Explorer from the **Real-Time** tab of the Simulink Toolstrip.

The Simulink Editor displays the **Real-Time** tab for models that are configured for the `speedgoat.tlc` code generation target. Click the **Connect to Target Computer** button in the

Simulink **Real-Time** tab to connect to the machine. Once connected, click the **Run on Target** button to deploy the model.

Simulink Real-Time™ automatically generates C code from your model by using Simulink Coder™. The generated code and the bitstream for the FPGA are loaded onto the target machine, and model execution starts automatically.

See Also

`sschdladvisor` | `hdladvisor` | `hdlsetup`

Related Examples

- “Generate HDL Code for Simscape Models” on page 30-13
- “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82
- “Deploy Simscape DC Motor Model to Speedgoat FPGA IO Module” on page 30-106
- “Generate HDL Code for Simscape Three-Phase PMSM Drive Containing Averaged Switch” on page 30-118
- “Generate HDL Code for Simscape Models by Using Trapezoidal Rule Solver” on page 30-135

31

Simscape HDL Workflow Advisor Tasks

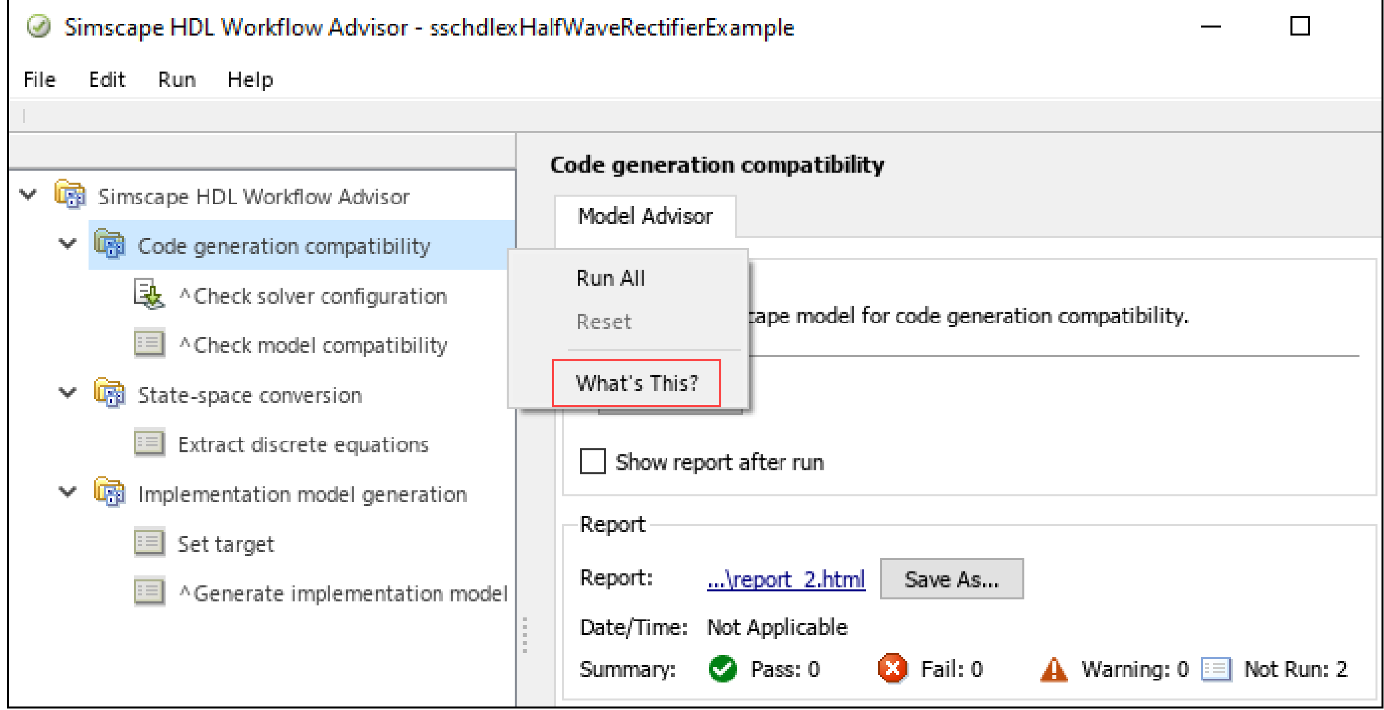
- “Simscape HDL Workflow Advisor Tasks” on page 31-2
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 31-7

Simscape HDL Workflow Advisor Tasks

By using the Simscape HDL Workflow Advisor, you can generate an HDL implementation model. You can then generate HDL code for the implementation model and deploy the code onto FPGA platforms. To open the Advisor, run the `sschdladvisor` function. For example:

```
openExample('hdlcoder/HILImplementationOfSimscapeModelOnSpeedgoatFPGAIOModulesExample',...
'supportingFile','sschdlexHalfWaveRectifierExample')
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

For summary information on each Simscape HDL Workflow folder or task, right-click the folder or task and select **What's This?**.



Simscape HDL Workflow Advisor

To convert your Simscape model to an HDL implementation model, the Simscape HDL Workflow Advisor contains various tasks that you can use. You can then generate code for the HDL Subsystem in the HDL implementation model. The tasks in the Simscape HDL Workflow Advisor are in these folders:

- The **Code generation compatibility** folder contains tasks that check whether the model uses the correct solver configuration settings and reports blocks that are not compatible for HDL code generation.
- The **State-space conversion** folder contains tasks that get the state-space parameters from your model for generating the implementation model.
- The **Implementation model generation** folder contains tasks that generate the HDL implementation model from the state-space parameters for the specified target hardware settings.

To learn more about each folder or task, right-click the folder or task and select **What's This?**.

Code Generation Compatibility

The tasks in the **Code generation compatibility** folder check whether:

- You have correctly specified the solver configuration settings and the settings are consistent across Solver Configuration blocks inside each network in your Simscape model.
- Your model uses HDL-compatible blocks.

Check Solver Configuration

The **Check solver configuration** task checks whether you have specified the correct settings and the settings are consistent across Solver Configuration blocks inside each network in your Simscape model.

The Simscape HDL Workflow Advisor checks whether you specified these settings for all Solver Configuration blocks:

- **Use local solver** is selected.
- **Solver type** is set to `Backward Euler`, `Partitioning`, or `Trapezoidal Rule`.
- A discrete sample time, `Ts`, is specified.
- **Use fixed-cost runtime consistency iterations** selection is same (either selected or cleared).
- The **Nonlinear iterations** value is same when **Use fixed-cost runtime consistency iterations** is selected.

If you did not specify these settings, the task provides a link to the Solver Configuration block in your model and the settings to modify.

Check Model Compatibility

The **Check model compatibility** task checks whether you use HDL-compatible blocks in your Simscape model.

If this task passes, it displays :

- A message indicating that the model is switched linear.
- The number of Simscape networks present in the model.
- The number of algebraic and differential variables for each Simscape network with links to the related blocks in your Simscape model.

  Differential variables consume a quadratic amount of multiplier resources on the target FPGA device. Algebraic variables consume a linear amount of multiplier resources. You can use this information to determine how many multiplier resources your Simscape design consumes on the FPGA device.
- A message with links to the Simulink-PS Converter and PS-Simulink Converter blocks in your model if you use the default names for these blocks.

The input and output ports of the `HDL Subsystem` in the implementation model use the names that you specify for the Simulink-PS Converter and PS-Simulink Converter blocks. To avoid this message, use a meaningful name for these blocks.

State-Space Conversion

Before you can generate the HDL implementation model, run the tasks in this folder to get the state-space parameters from your model. The tasks in this folder:

- Simulate the Simscape model to extract the differential algebraic equations and mode-switching function.
- Discretize the differential algebraic equations to generate an abstract state-space representation that represents the model in the form of linear modes.

Extract Discrete Equations

The **Extract discrete equations** task simulates your Simscape model and extracts the differential algebraic equations and the mode-switching function. It discretizes the differential algebraic equations and generates an abstract, discrete state-space representation of the model in the form of linear modes. Each mode is represented by a set of state-space matrices. This task gets the **Simulation stop time** value from the original Simscape model and displays the **Discrete sample time** for Simscape networks.

If this task passes, it displays the simulation stop time, number of solver iterations, number of states, inputs, outputs, modes, differential variables, and state-space representation for each Simscape network present in the model.

In the Solver Configuration block, select the **Use fixed-cost runtime consistency iterations** check box and specify a custom value for the number of solver iterations in the **Nonlinear iterations** text box. To learn more, see "Using Number of Solver Iterations" on page 31-9.

The number of modes is limited by the number of switches present in your Simscape model. The maximum number of modes possible is 2^n, where n is the number of switches. All the modes that the Simscape HDL Workflow Advisor generates are executed according to the input parameters by using a switching logic. The Advisor selects a valid number of modes depending on the design of your Simscape model.

Implementation Model Generation

The tasks in the **Implementation model generation** folder generate an HDL implementation model from the discrete state-space representation. The implementation model represents the Simscape algorithm by using Simulink blocks that are compatible for HDL code generation for the specified target hardware settings. If the task **Generate implementation model** in this folder passes, it provides a link to the generated HDL implementation model.

Set Target

In this task, you can provide the information required for target hardware settings before generating the HDL implementation model. This helps incorporate the HDL optimizations (such as setting optimal sharing factor value) required for hardware deployment. In the right pane, select a **Synthesis Tool** option and update **Family**, **Device**, **Package**, and **Speed** for the target hardware.

Generate Implementation Model

To generate an HDL implementation model from the discrete state-space representation, run the **Generate implementation model** task. The HDL implementation model contains an `HDL Subsystem` that models the state-space equations by using the state-space parameters you get by running the tasks in the **State-space conversion** folder. The `HDL Subsystem` block represents the DUT for which you can generate HDL code.

Before you run this task, you can:

- Use the **Data type precision** setting to specify whether the `HDL Subsystem` in the generated implementation model stores matrix types in `single`, `double`, or `fixed-point` and computes the results in the specified data type. You can select the data type from the **Data type precision** drop-down list. When you select `Fixed-point` data type, you can also specify the word length in the **Fixed-point word length** text box. To learn more, see Data Type Precision and Numerical Accuracy on page 31-10.
- Use the **Map state space parameters to RAMs** setting to map the state-space parameters onto the hardware resources such as RAMs or lookup tables (LUTs). To learn more, see “Map State Space Parameters to RAMs” on page 31-11.
- Select the **Generate validation logic for the implementation model** to generate the logic that verifies whether the generated HDL implementation model is functionally equivalent to the original Simscape model. This task generates the logic for each Simscape network present in the model. You can specify a tolerance for the numerical correctness by setting **Validation logic tolerance**. The **Validation logic tolerance** is an absolute value. For example, you can specify a tolerance value of `1e-12`.
- Select the **Share adders and multipliers** option to set an optimal value of the **SharingFactor** for the Matrix Multiply (Product) blocks in your HDL implementation model. The optimal sharing factor value for a model is calculated based on the target details provided in the **Set target** task in the **Implementation model generation** folder of the Simscape HDL Workflow Advisor. Setting this option reduces synthesis overhead and optimizes resource utilization before generating HDL code for your model. If you do not specify target hardware or you specify a target hardware that is not supported, by default the target platform Xilinx Vivado Kintex-7 `xc7k325t` part is used for calculating the optimal sharing factor. For more information on supported device families and parts, see Optimal Sharing Factor Supported FPGA Device Families on page 31-11.

If the task passes, you see a link to the generated HDL implementation model.

See Also

More About

- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 31-7
- “Generate HDL Code for Simscape Models” on page 30-13
- “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 30-2

- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 30-28

Simscape HDL Workflow Advisor Tips and Guidelines

By using the Simscape HDL Workflow Advisor, you can generate an HDL implementation model. You can generate HDL code for the implementation model and deploy the generated code onto FPGA platforms. To open the Advisor, run the `sschdladvisor` function. For example:

```
openExample('plantdeployment/OpenTheSimscapeHDLWorkflowAdvisorExample','supportingFile','sschdlex
sschdladvisor('sschdlexBoostConverterExample')
```

The Simscape HDL Workflow Advisor contains various tasks that convert your Simscape model to the HDL implementation model. When running various tasks in the Simscape HDL Workflow Advisor, you can follow certain tips and guidelines. You see these tips in the UI window of a particular task. For example, in the task that discretizes the equations to state-space parameters, the UI has a tip that suggests how to change the sample time. This section contains more information about each tip in the Simscape HDL Workflow Advisor UI.

Estimating Resource Consumption Using Algebraic and Differential Variables

After you run the **Check model compatibility** task, the task reports the number of differential and algebraic variables for each Simscape network present in the model. For example, this figure illustrates that there are two differential and two algebraic variables in the boost converter example model `sschdlexBoostConverterExample`.

```
openExample('plantdeployment/OpenTheSimscapeHDLWorkflowAdvisorExample','supportingFile','sschdlex
sschdladvisor('sschdlexBoostConverterExample')
```

Run the workflow to the **Check model compatibility** task.

Details

Number of Discrete Variables: 4

Number of Differential Variables: 2

Source	Value
Simscape_system.Capacitor.vc	Capacitor voltage
Simscape_system.Inductor.i_L	Inductor current

Number of Algebraic Variables: 2

Source	Value
Simscape_system.Capacitor.i	Current
Simscape_system.Inductor.v	Voltage

By viewing the number of algebraic and differential variables, you can determine how the design consumes resources on the FPGA device. If `Nd` is the number of differential variables and `Na` is the

number of algebraic variables, the resource usage on the target hardware varies according to the relation `Nd*(Nd+Na)`. Differential variables consume a quadratic amount of multiplier resources on the target FPGA device. Algebraic variables consume a linear amount of multiplier resources. You can use this information to determine how many multiplier resources your Simscape design consumes on the FPGA device and whether your design is ready for conversion to state-space representation.

Setting Simulation Stop Time for Extract Discrete Equations

Change Simulation Stop Time

When you run the **Extract discrete equations** task, the Simscape HDL Workflow Advisor reports the simulation stop time. The simulation stop time corresponds to the amount of time that the Advisor takes to run simulation on your Simscape model. The stop time must not be significantly large such that the Advisor takes a long time to run this task. Use a stop time that is sufficient to reach the required number of modes for your model. To change the stop time, navigate to the Simscape model, and then specify the **Stop Time**.

Simulation Stop Time and Number of Modes

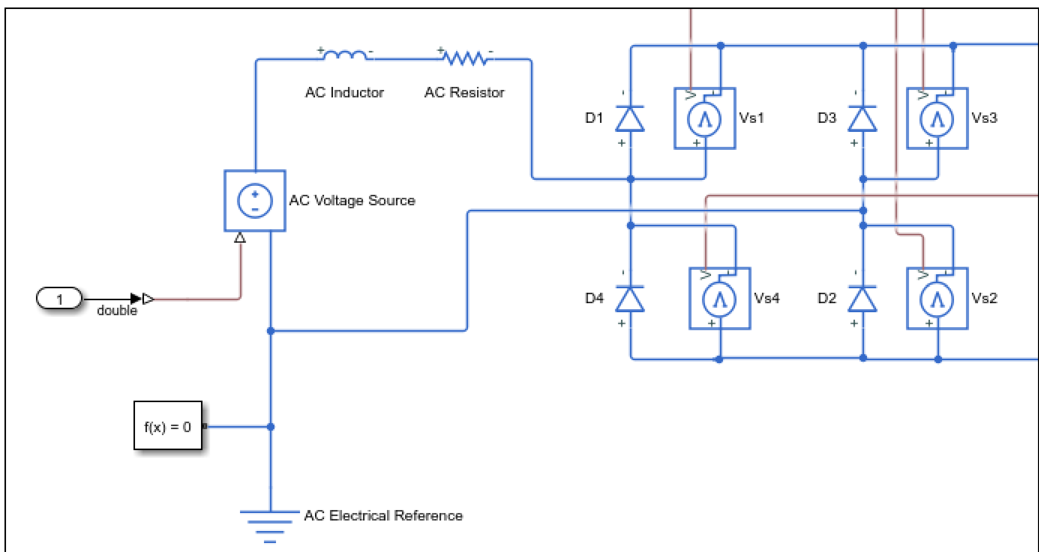
In the **Extract discrete equations** task, when you extract the differential algebraic equations, the Simscape HDL Workflow Advisor simulates the model to cover the nonlinear range of Simscape blocks. This task can take a long time depending on the number of switching elements in the Simscape model.

For a switched linear model, each switching element in the design has two modes. A switched linear model with `n` switching elements has `2^n` possible modes. For Simscape models with large number of switching elements, the number of modes can become significantly large. For example, the Vienna rectifier has `21` switching elements, which translates to `2^21` possible modes. The Simscape HDL Workflow Advisor can take a long time to simulate such a large model and cover such a large number of modes. In addition, the HDL implementation model that you generate for such a design can consume a large amount of resources or may not even fit on the target FPGA device.

In most cases, while simulating the model, the Advisor does not have to reach the entire `2^n` modes. For example, consider this bridge rectifier model. To open this model, enter:

```
openExample('plantdeployment/BridgeRectifierModelExample','supportingFile','sschdlexBridgeRectif
```

Inside the `Simscape_system` Subsystem, you see the four diodes arranged in a bridge configuration.



As each diode has two states, the Simscape design can have `2^4 = 16` possible states. In contrast, the bridge rectifier has only three modes. The modes are:

- Diodes `D1` and `D2` are `ON`, `D3` and `D4` are `OFF`
- Diodes `D1` and `D2` are `OFF`, `D3` and `D4` are `ON`
- Diodes `D1`, `D2`, `D3`, and `D4` are `OFF`

This example shows that, based on the Simscape algorithm and the input to the design, you can set the simulation stop time to a minimum value that covers the number of modes to be reached.

Changing Sample Time for Extract Discrete Equations

Change Sample Time

When you run the **Extract discrete equations** task, the Simscape HDL Workflow Advisor reports the discrete sample time. The discrete sample time corresponds to the sample time that the Advisor uses to discretize the differential algebraic equations to state-space parameters. To change the sample time, in your Simscape model, open the Block Parameters dialog box for the Solver Configuration block, and then specify the **Sample time**.

Sample Time and Discretizing Equations

In the **Extract discrete equations** task, the Simscape HDL Workflow Advisor discretizes the differential algebraic equations into state-space parameters. You extract the differential algebraic equations by simulating the Simscape model. The task obtains the sample time information from the sample time that you specify for the Solver Configuration block in your model. The Advisor then discretizes the equations to state-space parameters based on this sample time information.

Using Number of Solver Iterations

What is Number of Solver Iterations?

On the Solver Configuration block, you can specify a custom value for the number of solver iterations in **Nonlinear iterations** text box when you select the **Use fixed-cost runtime consistency iterations** check box. The number of solver iterations refer to the number of times the state-space model is executed per mode. The Simscape HDL Workflow Advisor generates the number of iterations that are required for executing the state-space model, automatically.

For each mode in the physical system, the switched linear workflow arrives at a state-space representation. The solver method is iterative and performs multiple computations to determine the correct mode for the next time step. After a certain number of iterations, the output value from the next time step becomes the same as the value from the previous time step. This consistency in the output value indicates the correct number of solver iterations.

By default, the **Number of Solver iterations** is 1 for linear models. For switched linear models, the **Number of solver iterations** depends on the number of mode iterations that Simscape uses during model simulation. This chosen value is optimal such that it causes the model to converge and avoids exceeding the threshold value for real-time deployment.

Using Fixed-Cost Runtime Consistency Iterations

On the Solver Configuration block, the **Use fixed-cost runtime consistency iterations** check box is cleared by default. If you select this check box, the **Nonlinear iterations** setting on the Solver Configuration block allows you to specify the Number of solver iterations.

To learn more about the **Use fixed-cost runtime consistency iterations** setting, see Solver Configuration. See also “Solvers for Real-Time Simulation” (Simscape).

Change Number of Solver Iterations

By default, you can change the number of solver iterations on this task. Increasing the number of solver iterations improves the numerical accuracy of generated HDL implementation model. To achieve higher sampling frequencies, reduce the number of solver iterations. Choose a value for number of solver iterations that trades off numerical accuracy and sampling frequency.

On the Solver Configuration block, if you select the **Use fixed-cost runtime consistency iterations** check box then you can change the number of solver iterations in the **Nonlinear iterations** parameter text box. Then, rerun the **Generate implementation model** task.

Trading off Numerical Accuracy and Sampling Frequency

To verify whether the numeric results of the HDL implementation model matches the original Simscape model, select **Generate validation logic for the implementation model**. If the numeric results from the HDL implementation model do not match, you can increase the number of solver iterations. To learn more, see “Increase Number of Solver Iterations” on page 30-92.

Changing the number of solver iterations trades off numerical accuracy for sampling frequency. Increasing the number of solver iterations increases the sample time of the HDL implementation model which can reduce the sampling frequency. See “Reducing Number of Solver Iterations” on page 30-98.

Data Type Precision and Numerical Accuracy

Use the **Data type precision** setting to specify whether you want the algorithm inside the HDL Subsystem in the generated implementation model to use single or double floating-point data type, or fixed-point data type when performing the matrix computations.

Data type Precision	Description
Double	Using double floating-point data type precision increases the numerical accuracy of the generated model and the maximum achievable target frequency. However, the area consumption and pipeline latency are also increased.
Single	This is the default setting for data type precision.
Single coefficient, double computation	This mode offers a tradeoff between Single and Double modes of floating-point data type precision. To save memory usage, the coefficients that are stored in single. The matrix computations are then performed in double for improved accuracy.
Fixed-point	This mode of data type precision determines the dynamic range of state-space matrices, and computes the appropriate fraction lengths and full precision integer rounding modes by using the specified word length. This reduces resource utilization and improves FPGA sampling frequency by reducing the oversampling factor. The fixed-point data type is supported for Simscape models with single Simscape network that use the Backward Euler or Trapezoidal Rule local solvers.

To learn more about the floating-point precision settings and tradeoffs, see “Use Larger Floating-Point Precision” on page 30-93.

Map State Space Parameters to RAMs

The **Map state space parameters to RAMs** setting enables you to map the state space parameters onto the hardware resources such as RAMs or lookup tables (LUTs).

Map State Space Parameters to RAMs	Description
Auto (default)	Maps the state space parameters to RAMs when the number of modes exceeds a set threshold value (200). Otherwise, it maps them to LUTs.
On	Maps state space parameters to RAMs.
Off	Maps state space parameters to LUTs.

Optimal Sharing Factor Supported FPGA Device Families

The Simscape HDL Workflow Advisor supports the optimal value of sharing factor for the following FPGA device families and parts listed in the table.

Device Family		Parts
XilinxVivado	Kintex-7	xc7k325t xc7k410t
	Kintex U	xcku115-flvb1760-1-c xcku085-flvb1760-1-c
	Artix-7	xc7a200t xc7a50t
	ZynqUltraScale+	xczu11eg-ffvc1760-2-e xczu17eg-ffvc1760-2-e xczu19eg-ffvc1760-2-e
Altera	MAX® 10	10M50DAF484C6GES

See Also

More About

- “Generate HDL Code for Simscape Models” on page 30-13
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 30-28

Troubleshooting

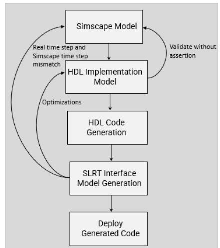
- “Troubleshooting Real-Time Hardware Deployment Issues in Simscape Hardware-in-the-Loop Workflow” on page 32-2
- “Troubleshoot Validation Errors in Simscape Hardware-in-the-Loop Workflow” on page 32-9

Troubleshooting Real-Time Hardware Deployment Issues in Simscape Hardware-in-the-Loop Workflow

Simscape lets you rapidly create models of physical systems within the Simulink environment. You model and simulate multidomain physical systems such as electric motors, bridge rectifiers, hydraulic actuators, and refrigeration systems by assembling fundamental components into a schematic. For more information, see “How Simscape Models Represent Physical Systems” (Simscape).

You can generate HDL code for the plant model you develop using Simscape blocks and then deploy the generated code to platforms such as standalone FPGA boards, Speedgoat FPGA I/O modules, and system on a chip (SoC) devices. By deploying the plant model to an FPGA board, you can accelerate the simulation of your plant model and simulate the model in real time by using hardware-in-the-loop (HIL) simulations. For more information, see “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 30-2.

This workflow diagram shows the different stages of the Simscape HIL workflow.



In this topic, you will learn how:

- Timing violations occur on hardware.
- To address timing violations on hardware.

Parameter Settings

You can generate HDL code for a wide variety of Simscape networks and simulate it. However, hardware deployment of the same models has certain requirements that need to be addressed early in the design process. For instance, the generated HDL code may consume a lot of hardware resources. Understanding how your design maps to your target device, is key to generating HDL code that will meet your needs when implemented downstream.

When you generate HDL code and deploy the plant model onto an FPGA, you may want to improve the sampling frequency.

- **Sampling Frequency:** The sampling frequency (FPGA sampling frequency) is the frequency at which the models run on the hardware. It is defined as,

$$\text{Sampling Frequency} = \frac{\text{FPGA clock frequency}}{(\text{Oversampling factor} \times \text{Number of solver iterations})}$$

  The sampling frequency depends on these parameters:

  - FPGA clock frequency
  - Oversampling factor
  - Number of solver iterations

  The real time step on FPGA, `Ts_fpga`, is the sample time at which the models run on the hardware. It is the reciprocal of the sampling frequency.

  The value of Simscape sample time is fixed for a Simscape model. Simscape sampling frequency is the reciprocal of the Simscape sample time. For the Simscape model simulation, you can specify the **Sample time** in the Solver Configuration block dialog box under the **Use local solver** parameter. To learn more about how this block is used in your model before running the Simscape HDL Workflow Advisor, see “Generate HDL Code for Simscape Models” on page 30-13.

- **Number of Solver Iterations:** The number of solver iterations refer to the number of times the state-space model is executed per time step. The Simscape HDL Workflow Advisor calculates the number of iterations that is required for executing the state-space model, automatically. On the Solver Configuration block, you can select the **Use fixed-cost runtime consistency iterations** check box and specify a custom value for the number of solver iterations in **Nonlinear iterations** text box.
- **Oversampling Factor:** The **Oversampling factor** specifies the factor by which the global clock signal is a multiple of the base rate at which the model operates. Generation of the global oversampling clock affects the generated HDL code. It does not affect the simulation behavior of your model. You can specify the value of oversampling factor in the **Clock Settings** section of the **HDL Code Generation** > **Global Settings** pane in the Configuration Parameters dialog box. For more information on oversampling factor, see Oversampling factor.
- **FPGA Clock Frequency:** The FPGA clock frequency refers to the clock rate for the FPGA implementation of your design. HDL Coder modifies the clock module setting in the reference design to produce the clock signal with the target frequency.

  The Simscape HDL Workflow Advisor calculates target frequency based on the details you specify for the **Synthesis Tool**. This calculation is limited to Simscape models containing a single Simscape network with a target frequency value that is within **Frequency Range (MHz)**.

  If required, you can change the target frequency by using the **Target Frequency (MHz)** setting in the **Set Target Frequency** task in the HDL Workflow Advisor. For more information, see “HDL Workflow Advisor Tasks” on page 36-2.

  The Speedgoat boards that are supported by `Xilinx Vivado` use the `IP Core Generation` workflow infrastructure. Enter a target frequency value that is within **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses **Default (MHz)** target frequency.

To get correct results, the Simscape time step and real time step on the FPGA need to match.

Critical Path Estimation

You can use critical path estimation to check if the model meets the timing requirement imposed by the target frequency. A critical path is a combinational path between an input and output that has the maximum timing delay. To make the critical path timing meet the target frequency that you want your design to achieve, break the critical path by adding delays. The additional delays increase the latency and register usage on the target FPGA. Critical path estimation speeds up the design iteration process. Critical path estimation is an alternative to annotating the critical path by performing **FPGA Synthesis and Analysis** with the HDL Workflow Advisor. For more information, see “Critical Path Estimation Without Running Synthesis” on page 21-192.

How Sample Rate Affects the Timing on the Hardware

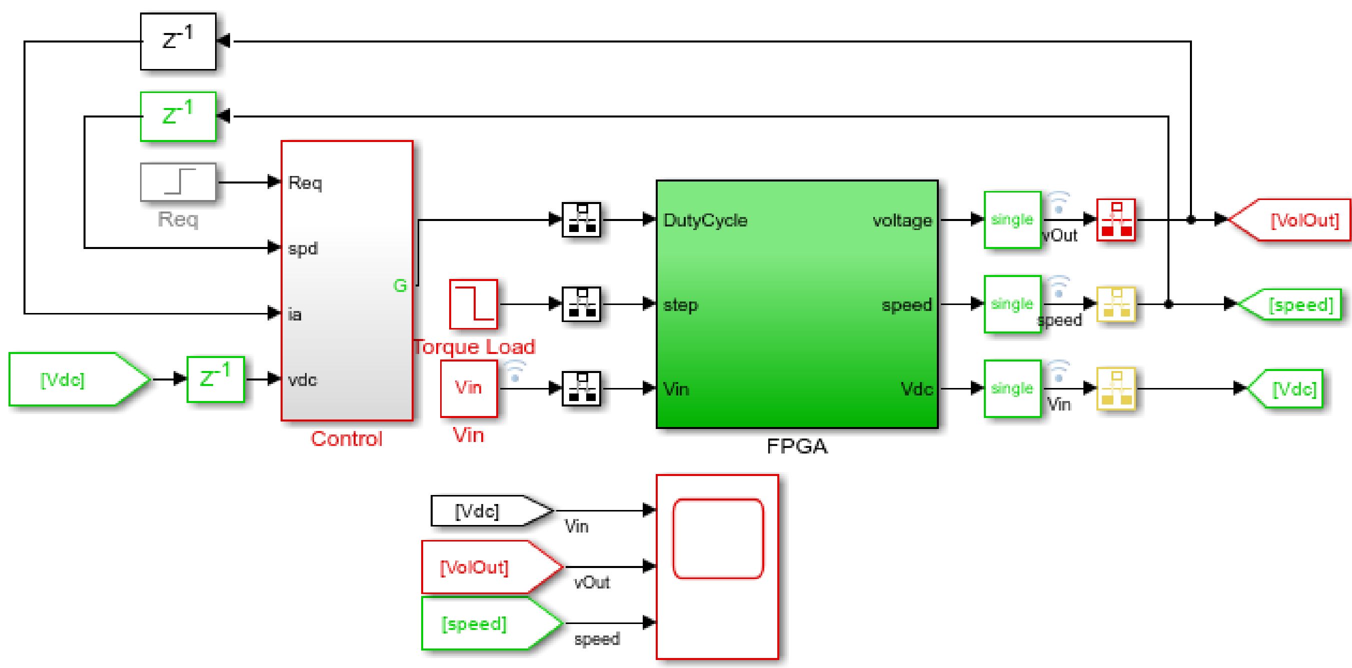
Variation of Sample Rate

In Simscape HIL workflow, the sample rate varies for the Simscape model, the HDL implementation model, and the Simulink Real-Time (SLRT) Interface model of the same design due to the dependency on other parameters such as oversampling factor and solver iterations.

Buck Converter with DC Motor

To understand the variation of sample rate, consider the model `ee_buck_converter_dc_motor_hdl.slx` that consists of a buck converter with a DC motor as load. To see the buck converter model, run this command.

```
openExample('plantdeployment/DeploySimscapeDCMotorToSpeedgoatFPGAIOModuleExample',...
'supportingFile','ee_buck_converter_dc_motor_hdl.slx')
```

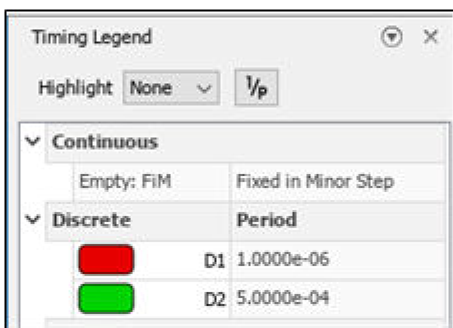


For the Simscape model, the Sample time, `Ts` is `6e-6`, which means that the sampling frequency of the Simscape model is 1/6 MHz. To understand the effect of different parameters on the hardware deployment, you can change the parameter values and see the effect on the output. For example, change the value of `Ts` to the default value `1e-6`. On the **Modeling** tab, select **Model Settings** > **Model Properties**. On the **Callbacks** tab, in the **Model callback** pane, select `PreLoadFcn` and change the value of `Ts` from `6e-6` to `1e-6`. Save and close the model. Reopen it for further steps.

To simulate the model, click the **Run** button or press **Ctrl + T**. You can view the Timing Legend for the Simscape model. The Timing Legend contains the sample time color, annotation, and value for each sample time in the model. To view the Timing Legend:

1 In the Simulink model window, on the **Modeling** tab, click **Update Model**.

2 On the **Debug** tab, select **Information Overlays** > **Timing Legend**, or press **Ctrl + J**.



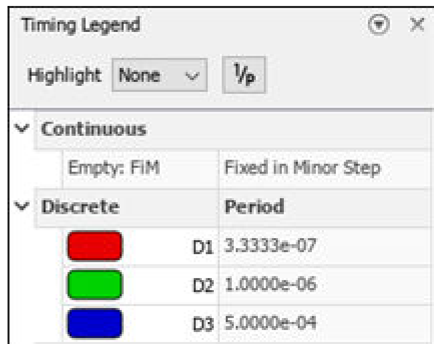
Generate the HDL implementation model by using Simscape HDL Workflow Advisor.

```
sschdladvisor('ee_buck_converter_dc_motor_hdl')
```

When you generate the HDL implementation model, the **Number of solver iterations** is set to an optimal value of 3 by default.

The sampling frequency of the generated HDL implementation model is 1 MHz. Because the HDL Algorithm in the HDL implementation model is iterative and with the value of **Number of solver iterations** (3 times), the HDL Algorithm subsystem runs at an updated sampling frequency of 3 MHz. The generated HDL implementation model uses Rate Transition blocks to handle the data transfer between the two subsystems (and blocks) operating at different rates.

You can now see the updated Timing Legend for the generated HDL implementation model.



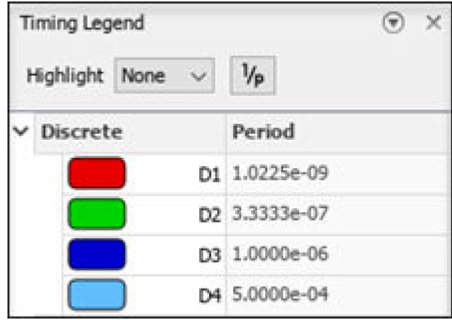
For the generated HDL implementation model, you can generate HDL code. To generate HDL code, use these commands:

```
open_system('gmStateSpaceHDL_ee_buck_converter_dc_motor_')
hdlsetup('gmStateSpaceHDL_ee_buck_converter_dc_motor_')
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'ResourceReport', 'on')
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'AdaptivePipelining', 'on');
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'FloatingPointTargetConfiguration',...
hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint', 'LatencyStrategy', 'Max'));
makehdl('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/HDL Subsystem');
```

When you run this code, you may encounter an error message that says to increase the oversampling factor to 326 to provide sufficient latency budget. The adaptive pipelining optimization and maximum latency strategy used for code generation increase the required value of the oversampling factor. Update the value of oversampling factor from 275 to 326 and generate the HDL code.

```
hdlset_param('gmStateSpaceHDL_ee_buck_converter_dc_motor_', 'Oversampling', 326);
makehdl('gmStateSpaceHDL_ee_buck_converter_dc_motor_/FPGA/HDL Subsystem');
```

When you run `makehdl`, it generates the HDL code and the generated model for `gmStateSpaceHDL_ee_buck_converter_dc_motor_` in the `hdlsrc` folder. Run the generated model `gm_gmStateSpaceHDL_ee_buck_converter_dc_motor_`. Observe the Timing Legend for this model.



Timing Legend

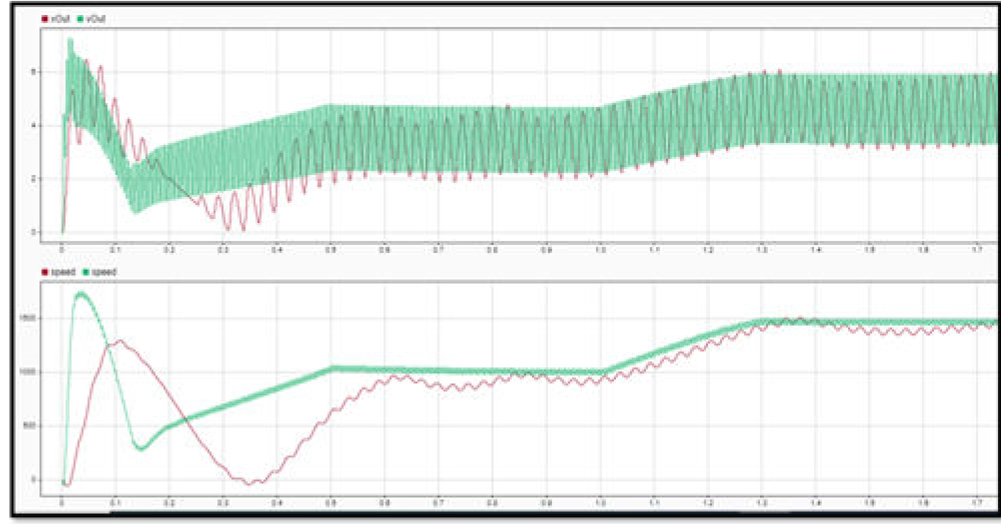
Highlight None

Discrete	Period
D1	1.0225e-09
D2	3.3333e-07
D3	1.0000e-06
D4	5.0000e-04

To generate a Simulink Real-Time (SLRT) Interface model, follow the steps from Simulink Real-Time FPGA I/O workflow by using the HDL Workflow Advisor. To learn more about each step, see “Deploy Simscape DC Motor Model to Speedgoat FPGA IO Module” on page 30-106.

With the oversampling factor set to the value 326, the FPGA clock frequency is 3*326 MHz, or 978 MHz. However, the allowable FPGA clock frequency range for the Speedgoat devices (IO334 module) is 50-250 MHz. This creates timing violation on the hardware.

As a result, the desktop simulation and the hardware results have a mismatch due to high FPGA clock frequency. The waveforms show the mismatch in desktop simulation (in green) and hardware results (in red) for the outputs `vOut` and `speed`.



How to Reduce the Sample Rate Variation

The Simscape sampling frequency is the rate at which the Simscape models run.

As mentioned in the previous section, the sampling frequency depends on FPGA clock frequency, the oversampling factor, and the number of solver iterations. To improve the sampling frequency, you can maximize the FPGA clock frequency, and minimize the oversampling factor and number of solver iterations. As you improve the sampling frequency, make sure that the updated sampling frequency is equivalent to the fixed sample time that you specify for your original Simscape model by using the Solver Configuration block.

You can synthesize your implementation model to find out the frequency at which the slack is positive. Then, using the number of solver iterations and the oversampling factor, calculate the correct Simscape time step.

Calculate the real time step on FPGA (`Ts_fpga`) by taking the reciprocal of the FPGA sampling frequency.

$$\text{Real time step on FPGA} = \frac{(\text{Oversampling factor} \times \text{Number of solver iterations})}{\text{FPGA clock frequency}}$$

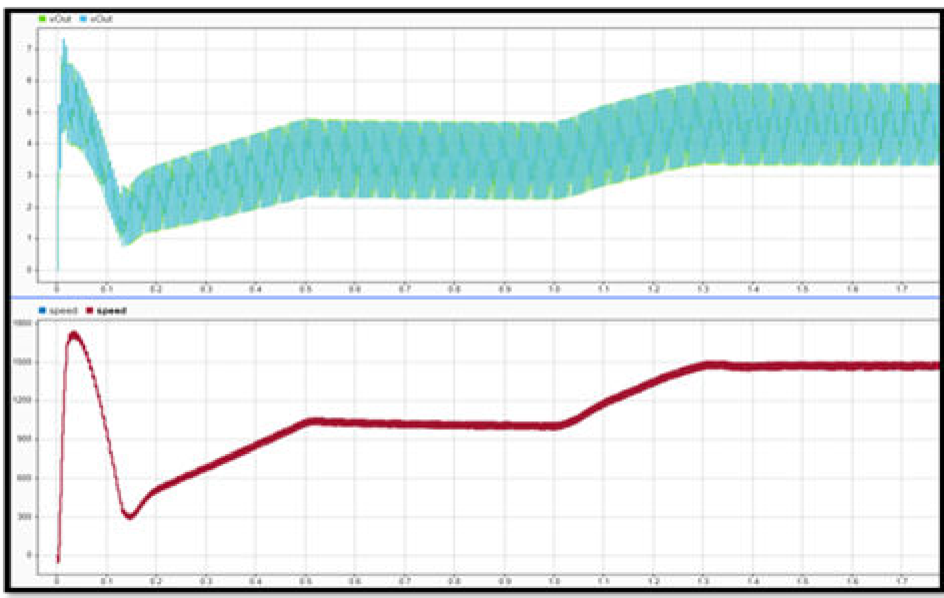
The target frequency of your current design is 164 MHz for which the design meets the timing constraints. Here, the oversampling factor is 326 and the number of solver iterations is 3. The updated real time step on FPGA is

```
Ts_fpga = (326*3)/164 = 5.963 ≈ 6 µs
```

To avoid timing violations on the hardware, use the calculated value of `Ts_fpga` as the Simscape sample time.

Change the sample time from `1e-6` to `6e-6` in the `PreLoadFcn` callback. Save the model, close, and reopen it.

Adjusting the parameter values reduces the mismatches in the results obtained from desktop simulation and the hardware results. The desktop simulation results align closely with the hardware results.



Limitations

- To estimate the correct target frequency, you need to perform model synthesis which might take some time (a few hours).
- Changing the sample time affects the discretization step in the Simscape HDL Workflow Advisor. As a result, the state-space parameters (A, B, C, D parameters) change, and you need to rerun the workflow from the beginning.

See Also

"Improve FPGA Sampling Frequency of HDL Implementation Model Generated from Simscape Algorithm" on page 30-96 | "Validate HDL Implementation Model to Simscape Algorithm" on page 30-89

Related Examples

- "Deploy Simscape DC Motor Model to Speedgoat FPGA IO Module" on page 30-106
- "Deploy Simscape Buck Converter Model to Speedgoat IO Module Using HDL Workflow Script" on page 30-36
- "Deploy Simscape Grid Tied Converter Model to Speedgoat IO Module Using HDL Workflow Script" on page 30-47

Troubleshoot Validation Errors in Simscape Hardware-in-the-Loop Workflow

You can generate an HDL implementation model from a Simscape model by using the Simscape HDL Workflow Advisor on page 31-2. The HDL implementation model represents the Simscape algorithm by using Simulink blocks that are compatible for HDL code generation for the specified target hardware settings. In the **State-space conversion** step, the Advisor extracts state-space data (the values of states or inputs) from the original Simscape model. In this step, the corresponding state-space matrices encountered during simulation are cached. The HDL implementation model uses the cached state-space data to calculate the updated state at each time step. If the HDL implementation model encounters states that are not cached during the simulation, a validation error occurs. Because of this, the values of states or inputs in the HDL implementation model differ from the original Simscape model.

In this topic, you will learn:

- What causes validation errors.
- How to address validation errors in HDL implementation models.

Consider, for example, an HDL implementation model that has four Boolean modes m1,m2, m3, and m4 with these configurations cached during simulation:

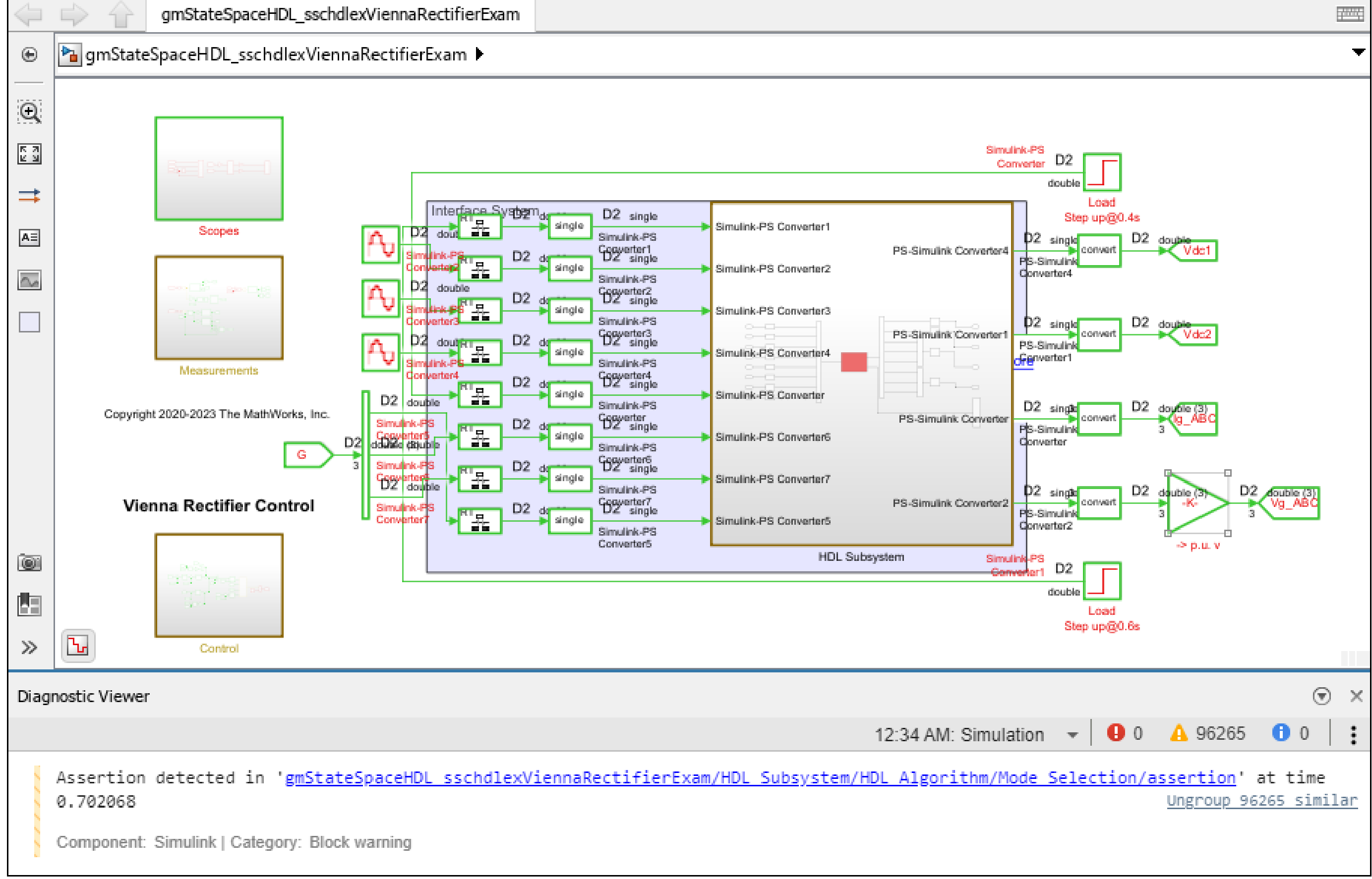
	m1	m2	m3	m4
Configuration1	0	0	0	0
Configuration2	0	1	1	0
Configuration3	1	1	1	1

Suppose that the HDL implementation model encounters a state with this mode configuration:

	m1	m2	m3	m4
ConfigurationX	0	1	0	0

Encountering this state may result in a validation mismatch because ConfigurationX was not cached during the simulation. As a result, the output of the HDL implementation model may not match the results from the original Simscape model. In some cases, it might lead to errors and the model might produce invalid results.

You can see the validation error in Vienna Rectifier Example on page 30-20 model when you generate the HDL implementation model with **Floating-point precision** selected as `Single`. The **Diagnostic Viewer** displays the assertion as warnings on simulating the HDL implementation model.



Causes of Validation Errors

When a validation error occurs in the HDL implementation model, the **Diagnostic Viewer** displays an assertion. These are some primary causes of validation errors.

Insufficient Precision

In the HDL Implementation model, you can select either single- or double-precision data type when performing matrix computations. When you select single precision for numerical calculations, precision losses can result. For sufficiently stiff models, even small variances caused by this precision loss can lead to validation errors if the HDL implementation model encounters new states that were not cached during Simscape simulation.

Numerical stiffness can make models prone to precision-related error. You can sometimes reduce the numerical stiffness by making changes to the model parameters (for example, the value of a capacitor). For more information, see "Reduce Numerical Stiffness" (Simscape).

You can resolve a validation error resulting from insufficient precision by generating an HDL implementation model with higher precision. To generate such a model by using the Simscape HDL Workflow Advisor, you can select either the `Single coefficient, double computation` or `Double` precision option under **Floating-point precision** in the **Generate implementation model** task window.

Note Models containing trigonometric functions are particularly prone to error due to precision loss. Models that use trigonometric functions may still experience validation errors even when using double precision.

Inputs to HDL Implementation Model Differ from Cached Data

The HDL implementation model relies on a set of cached state-space matrices encountered during the original Simscape model simulation. These matrices sometimes depend on the Simscape network input. When inputs to the HDL implementation model or HDL algorithm on the FPGA differ from the original desktop simulation data, the corresponding state-space data is not available and validation errors can occur. To cache the necessary data during desktop simulation in Simscape, provide the full range of expected inputs to the Simscape model.

To avoid this type of validation error, you can:

- Run the Simscape simulation up to the same **Simulation Stop Time** as is intended for the HDL implementation model. Do not extend the simulation run-time of the HDL implementation model significantly past the original Simscape run-time. This captures all input configurations during the **Extract discrete equations** stage of the workflow.
- Provide inputs to the Simscape model that cover the possible states for the model and the range of possible operating simulation inputs (for example, both 0 and 1 gate inputs, the full range of expected voltages for a controlled voltage source).

Inappropriate Solver Settings

The choice of solver settings can also lead to validation errors.

To match the behavior of a fixed-cost discrete HDL implementation model, in the Block Properties dialog box of the Solver Configuration block, select **Use fixed-cost runtime consistency iterations**. Selecting **Use local solver** automatically selects **Use fixed-cost runtime consistency iterations**, because these are the recommended settings for real-time and hardware-in-the-loop (HIL) simulations.

Make sure that you use enough solver iterations for convergence. If the Simscape model does not use enough iterations for convergence, it may produce inaccurate results. The `simscape.getLocalSolverFixedCostInfo` function reports the number of iterations required for fixed-step, fixed-cost simulation. The `MaxIterations` field of the function returns `-1` if the simulation fails to converge.

Data Transfer Issues

The Rate Transition block transfers data from the output of a block operating at one rate to the input of a block operating at a different rate. The behavior of the Rate Transition block depends on multiple model configuration parameters. If your model executes in multitasking mode with **Ensure data integrity during data transfer** disabled, then Rate Transition blocks introduce delays. Added delays at the input ports of the HDL algorithm can introduce validation errors for models with multiple solver iterations.

To prevent the error, you must use the single-tasking mode of execution for your model. To enable single-tasking mode, in the Configuration Parameters dialog box, under **Solvers > Solver details > Tasking and sample time options**, clear the **Treat each discrete rate as a separate task** check box. By default, the **Ensure data integrity during data transfer** check box is disabled for Simscape

HIL Workflow simulations. These settings ensure data integrity during data transfer while minimizing the latency in the HDL algorithm inside HDL implementation model.

New rates may be introduced in the HDL implementation models that are not a part of the original Simscape model. You can make all control systems or inputs to the Simscape plant behave equivalently across both models by manually implementing the rate transitions. To implement the rate transitions manually:

- Disable the model configuration parameter **Automatically handle rate transition for data transfer** and add Rate Transition blocks as needed. To learn more, see **Automatically handle rate transition for data transfer**.
- Enable the warning for single-task data transfer to identify any places that require manual rate transition handling. To enable the warning, in the Configuration Parameters dialog box, under **Diagnostics** > **Sample Time**, set the value of the **Single task data transfer** parameter to `warning`. For more information see, **Single task data transfer**.

These troubleshooting measures may involve significant tradeoff between accuracy and efficiency. Some of the techniques mentioned above may result in longer simulation run-times or take up more space on the FPGA (if you use double precision instead of single precision).

If your model achieves the desired accuracy without any of the suggested modifications, then you do not need to update your model with any of these settings. However, if your model fails to achieve acceptable accuracy or encounters validation errors, you might be able to improve your model using the suggested fixes.

See Also

Rate Transition | “Single task data transfer” | “Handle Rate Transitions” | “Multirate Model Requirements for HDL Code Generation” on page 20-7 | **Automatically handle rate transition for data transfer** | “Multitask data transfer”

Related Examples

- “Troubleshooting Real-Time Hardware Deployment Issues in Simscape Hardware-in-the-Loop Workflow” on page 32-2
- “Generate Optimized HDL Implementation Model from Simscape” on page 30-20

Model Protection in HDL Coder

- “Create Protected Models to Conceal Contents and Generate HDL Code” on page 33-2
- “Test Protected Models” on page 33-10
- “Package and Share Protected Models” on page 33-12
- “Obfuscate Generated HDL Code from Simulink Models” on page 33-15

Create Protected Models to Conceal Contents and Generate HDL Code

When you want to share a model with a third party without revealing intellectual property, protect the model. When you create a protected model, you conceal the implementation details of the original model by compiling it into a referenced model. The protected model includes derived files to support the optional functionalities that you specify, such as support for C code generation or HDL code generation.

If you have a HDL Coder license, you can create a protected model with simulation and HDL code generation support. The protected model user can then generate HDL code for models that reference the protected model that you created. To enable C code generation support or specify additional options such as code interface, you must have a Simulink Coder or Embedded Coder® license. To learn more about that workflow, see “Protect Models to Conceal Contents”.

How Model Protection Works

When you protect a model, you can allow the user of the protected model to:

- Simulate a model that includes the protected model in normal, accelerator, rapid accelerator, or external mode.
- Open a read-only web view of the model, including model contents and block parameters. Creating a read-only web view of the model requires a Simulink Report Generator license.
- Generate HDL code for a model that includes the protected model.
- Generate C and C++ code for a model that includes the protected model, if you have a Simulink Coder license.
- Generate code for the protected model through the standalone interface, if you have an Embedded Coder license and specify an ERT-based system target file for the model.

You can optionally password-protect each option. If you choose password protection for one of these options, the software protects the supporting files by using AES-256 encryption.

How to Create a Protected Model

Create a protected model by using one of these options:

- To create a protected model from a referenced model, select the Model block and on the Simulink Toolstrip **Model Block** tab, click the **Protect** button.
- To create a protected model from the current model:
  - On the Simulink Toolstrip **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
  - Select the Model block and on the **HDL Code > Share** tab, select **Generate Protected Model**.
- To programmatically create a protected model, use the `Simulink.ModelReference.protect` function.

When you create a protected model:

- Simulink creates and stores a protected version of the model in a file that has the same name as the source model, with an `.slxp` extension.

- The original model file, with the `.slx` extension, does not change. If you protect the model through a Model block, that Model block does not change.
- Optionally, Simulink creates a project archive (`.mlproj`) that contains the protected model, a harness model for the protected model, and additional supporting files.

If your protected model requires additional supporting files, such as base workspace definitions or a data dictionary, include these files with the model when you share the protected model. For more information, see "Package and Share Protected Models" on page 33-12.

General Protected Model Requirements and Limitations

When you create a protected model, consider these requirements:

- You must have a HDL Coder or Simulink Coder license to create a protected model.
- The model must be available on the MATLAB path.
- The model cannot have unsaved changes.
- The model uses the configuration that is active during protection. You cannot change the configuration of a protected model.
- If the model contains variants, the protected model includes only the variant that is active during protection.
- The protected model name must not be modified. Renaming the model or changing the suffix makes the model unusable until you restore its original name and suffix.

The model must also meet all requirements listed in "Model Reference Requirements and Limitations".

Protected Model Restrictions for HDL Code Generation

When you create a protected model that has HDL code generation support, these restrictions apply:

- The protected model must use the same configuration parameters as the top level that it is referenced from.

  In R2021b, HDL Coder enables a difference in the values of these parameters between the protected model and the top model so that you can simulate and generate HDL code for top model and protected models while retaining top model synthesis parameters settings.

  - Family
  - Device Name
  - Package Name
  - Speed value
  - Target frequency
- The solver settings that you specify in the **Solver** pane of the Configuration Parameters dialog box must be `Fixed-step` and `auto`.
- You must not enable these settings in the Configuration Parameters dialog box:
  - **Generate parameterized HDL code from masked subsystem**
  - **Module name prefix**

  - **Use trigger signal as clock**
  - **Minimize clock enables**
  - **Scalarize vector ports**
  - **Allow clock-rate pipelining at DUT output ports**
- Models with multiple clock signals or the **Clock inputs** set to `multiple` in the Configuration Parameters dialog box are not supported.
- Models that contain model arguments are not supported.
- HDL source code of the protected model cannot be obfuscated.
- Nested protected models are not supported.
- The protected model cannot have callbacks.

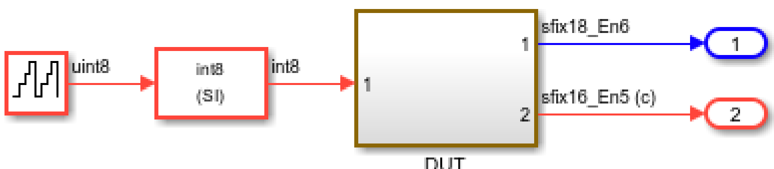
To learn more about limitations for C code generation, see “Code Generation Requirements and Limitations”.

Prepare the Parent Model

This example shows how you can protect a model that is referenced by a Model block in the parent model.

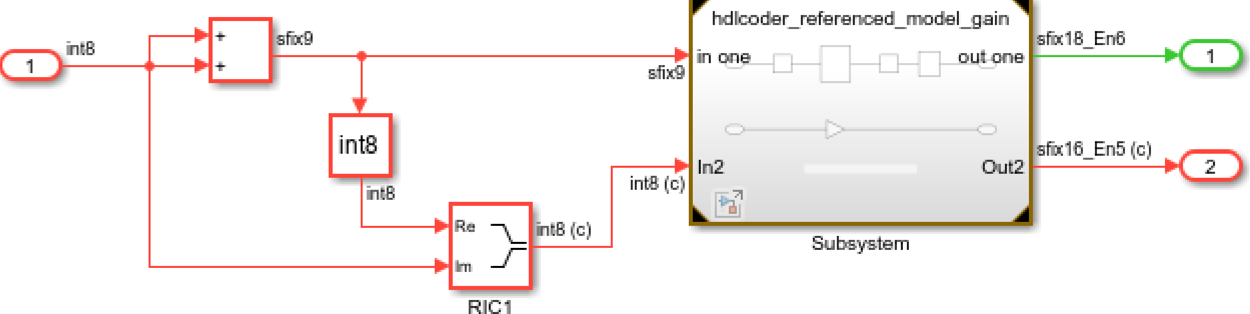
Open the parent model named `hdlcoder_protected_model_parent_harness`. To view the information overlays, in the Simulink Toolstrip, on the **Modeling** tab, click **Update Model**.

```
mdl = "hdlcoder_protected_model_parent_harness";
open_system(mdl)
set_param(mdl,SimulationCommand="Update")
```



To navigate to the Model block in your parent model, double-click the Subsystem block named `DUT`. Then, double-click the Subsystem block named `mynested`. In this subsystem, a Model block references the model named `hdlcoder_referenced_model_gain`.

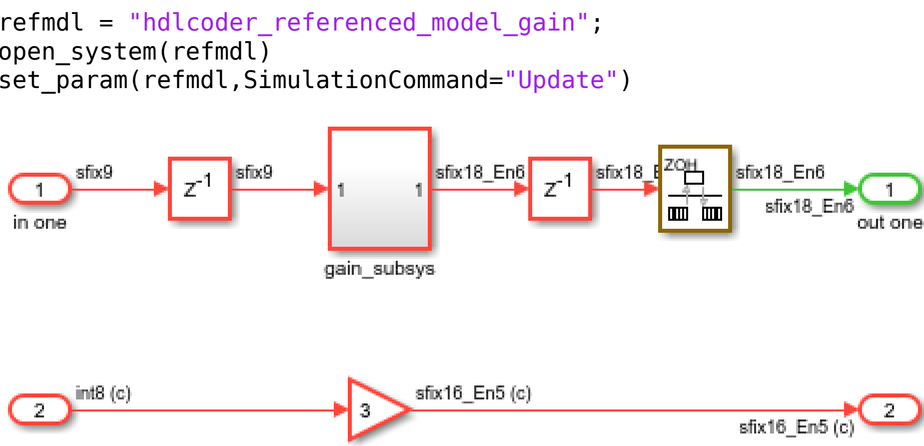
```
ss = "hdlcoder_protected_model_parent_harness/DUT/mynested";
open_system(ss)
```



Open the Property Inspector. Then, select the Model block. In this example, the **Model name** value includes the .slx extension. When both the referenced model and the protected model exist in the same folder, the parent model references the protected model unless the extension is specified.

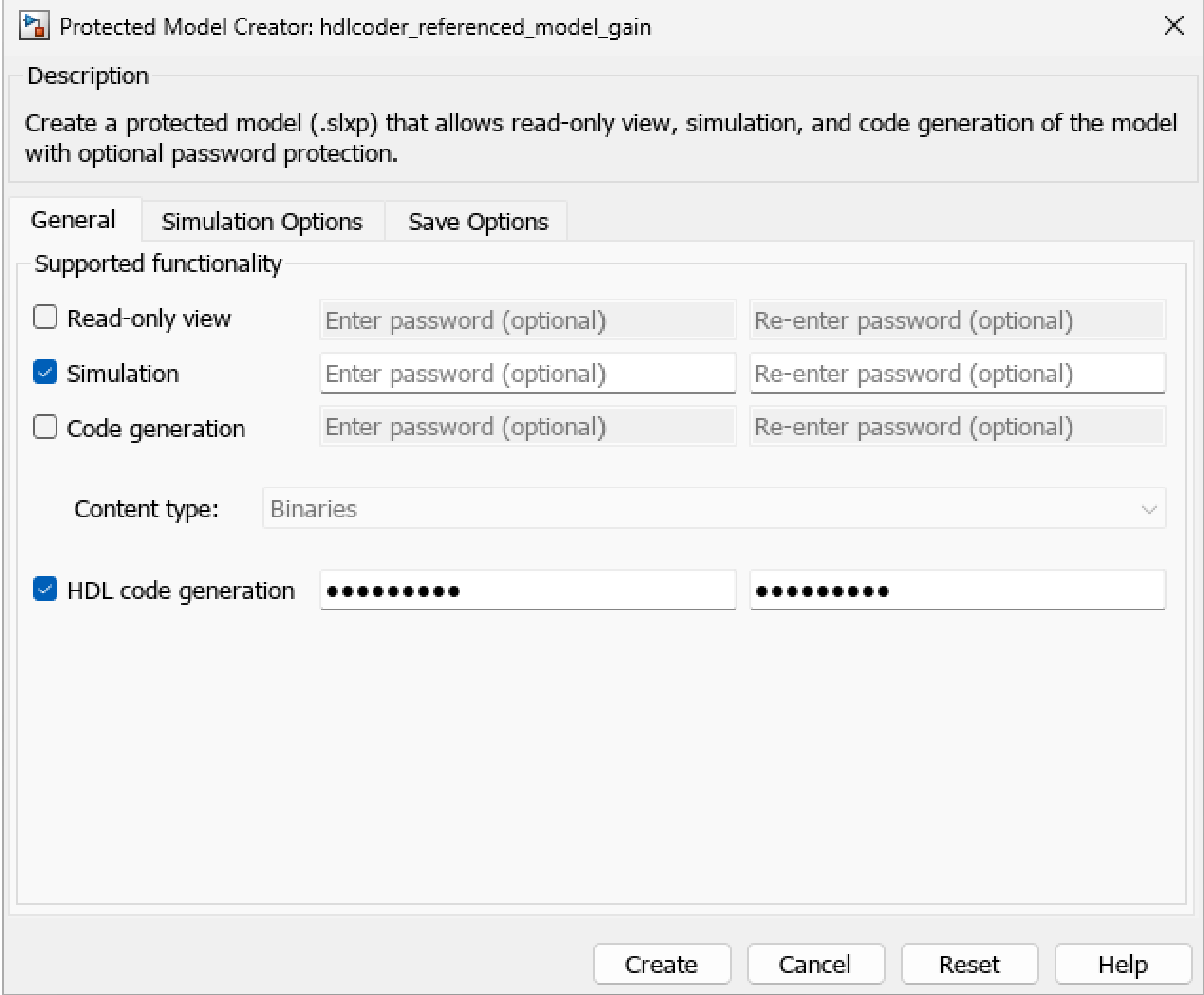
The Model block references the model named hdlcoder_referenced_model_gain.slx, which is the model that you want to protect. To view the referenced model, double-click the Model block or open the model named hdlcoder_referenced_model_gain in a separate window.

```
refmdl = "hdlcoder_referenced_model_gain";
open_system(refmdl)
set_param(refmdl,SimulationCommand="Update")
```



Protect the Referenced Model

1. Select the Model block.
2. On the Simulink Toolstrip **Model Block** tab, click **Protect**.



The Protected Model Creator opens.

3 In the Protected Model Creator, select **Simulation**. This option allows the protected model user to simulate the model that references the protected model.

Tip The Protected Model Creator caches your settings for a model during a MATLAB session. If you close and reopen the dialog box, the settings persist. Passwords and tunable parameter selections are not cached. To restore the default settings, click **Reset**.

---

4 If you have Simulink Coder or Embedded Coder, you can specify additional settings such as enabling code generation support with password protection by selecting **Code generation**, or specifying a **Code interface**. To learn more about these options, see “Protect Models to Conceal Contents” (Embedded Coder).

5 Select **HDL code generation** to generate HDL code for a model that references the protected model. If you want to password-protect this functionality of the protected model, you must specify a minimum of eight characters. You can specify a unique password for this option. You cannot obfuscate the HDL source code for a protected model.

6. In the **Destination folder** box, specify the folder path for the protected model. The default value is the current working folder.
7. To automatically collect, create, and package supporting files with the protected model, select **Package protected model with dependencies in a project**. For this example, clear this option.
8. To create a harness model for the protected model, select **Create harness model for protected model**. The harness model provides an isolated environment for the Model block that references the protected model. For this example, leave this option cleared.
9. Click **Create**.

   HDL Coder checks compatibility of the model for HDL code generation then generates code for the model. The generated code file contents are in the `hdlsrc` folder. To learn about the files that are generated, see “Package and Share Protected Models” on page 33-12.
10. To use the protected model in a model hierarchy, reference it through a Model block. The **Simulation mode** for Model blocks that reference a protected model is set to `Accelerator`. You cannot change the mode. For more information, see “Reference Protected Models from Third Parties”.

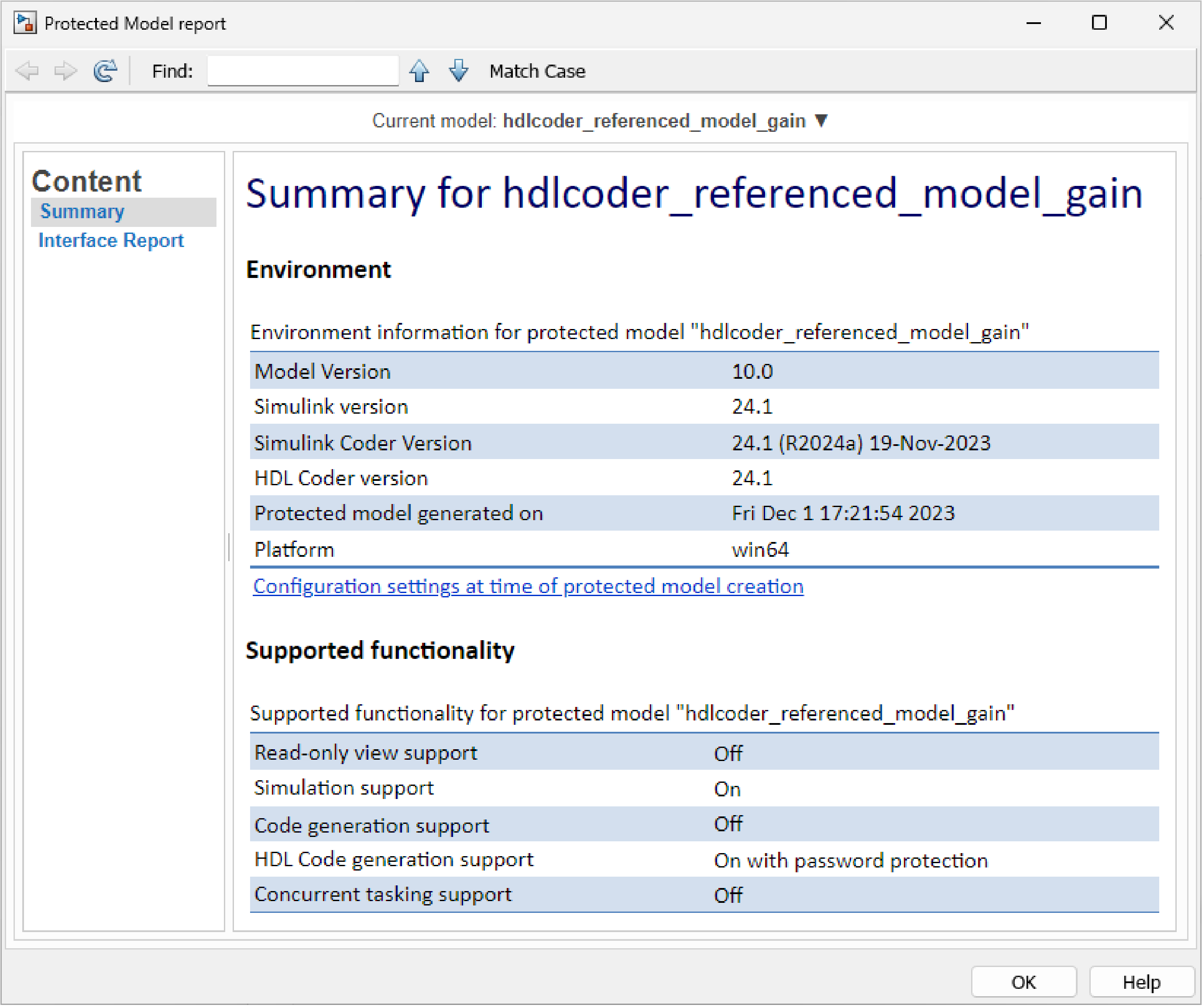
To learn more about the options, see Protected Model Creator.

To create a protected model when using the `Simulink.ModelReference.protect` function, set the `Mode` to `HDLCodeGeneration`. For example, run this command to protect the referenced model `hdlcoder_referenced_model_gain`:

```
Simulink.ModelReference.protect('hdlcoder_referenced_model_gain', ...
                                        'Mode','HDLCodeGeneration')
```

Protected Model Report

When you create the protected model from the Simulink Editor, a protected model report is generated and is included as part of the protected model. For this example, to view the protected model report, double-click the protected model or right-click the protected-model badge icon on the block in the harness model and select **Display Report**.



The report contains:

- A **Summary**, including the following tables:
  - **Environment**, providing the Simulink version, the Simulink Coder version, the HDL Coder version, and platform used to create the protected model.
  - **Supported functionality**, reporting `On`, `Off`, or `On with password protection` for each possible functionality that the protected model supports. If you configure your protected model for multiple targets, this table includes a list of supported targets.
- An **Interface Report**, including model interface information such as input and output specifications, exported function information, interface parameters, and data stores.

To generate a report when using the `Simulink.ModelReference.protect` function, set `Report` to `true`.

Generate HDL Code for Models Referencing Protected Model

If the protected model has simulation and HDL code generation support, the protected model user can simulate and generate HDL code from a model that references the protected model. You generate HDL code for a model that references a protected model in the same manner as how you would generate code for a regular model.

If the protected model is password-protected, before you generate code, right-click the protected model badge icon and select **Authorize**. You must then enter the password for each option. If the entered password matches the password that you specified when creating the protected model, the model is authorized. You can then generate HDL code for the model.

For example, to generate HDL code for the protected model `hdlcoder_referenced_model_gain.slxp` that is referenced by the `hdlcoder_protected_model_parent_harness` model:

1. Authorize the protected model `hdlcoder_referenced_model_gain.slxp` if you specified a password when creating the protected model.
2. Generate HDL code for the DUT Subsystem from the context menu or by using the `makehdl` function.

   ```
   makehdl('hdlcoder_protected_model_parent_harness/DUT')
   ```

See Also

Tools
Protected Model Creator

Functions
`Simulink.ModelReference.protect` | `Simulink.ModelReference.modifyProtectedModel`

More About

- “Test Protected Models” on page 33-10
- “Package and Share Protected Models” on page 33-12

Test Protected Models

To test a protected model that you created, compare the simulation results of the protected model to the output of the original model. As you supply the protected model from the original model, both the original and the protected model might exist on the MATLAB path.

In the parent model, if the Model block **Model name** parameter names the model without providing a suffix, the protected model takes precedence over the unprotected model. To override this default when testing the output, in the Model block **Model name** parameter, specify the file name with the extension of the unprotected model, `.slx`.

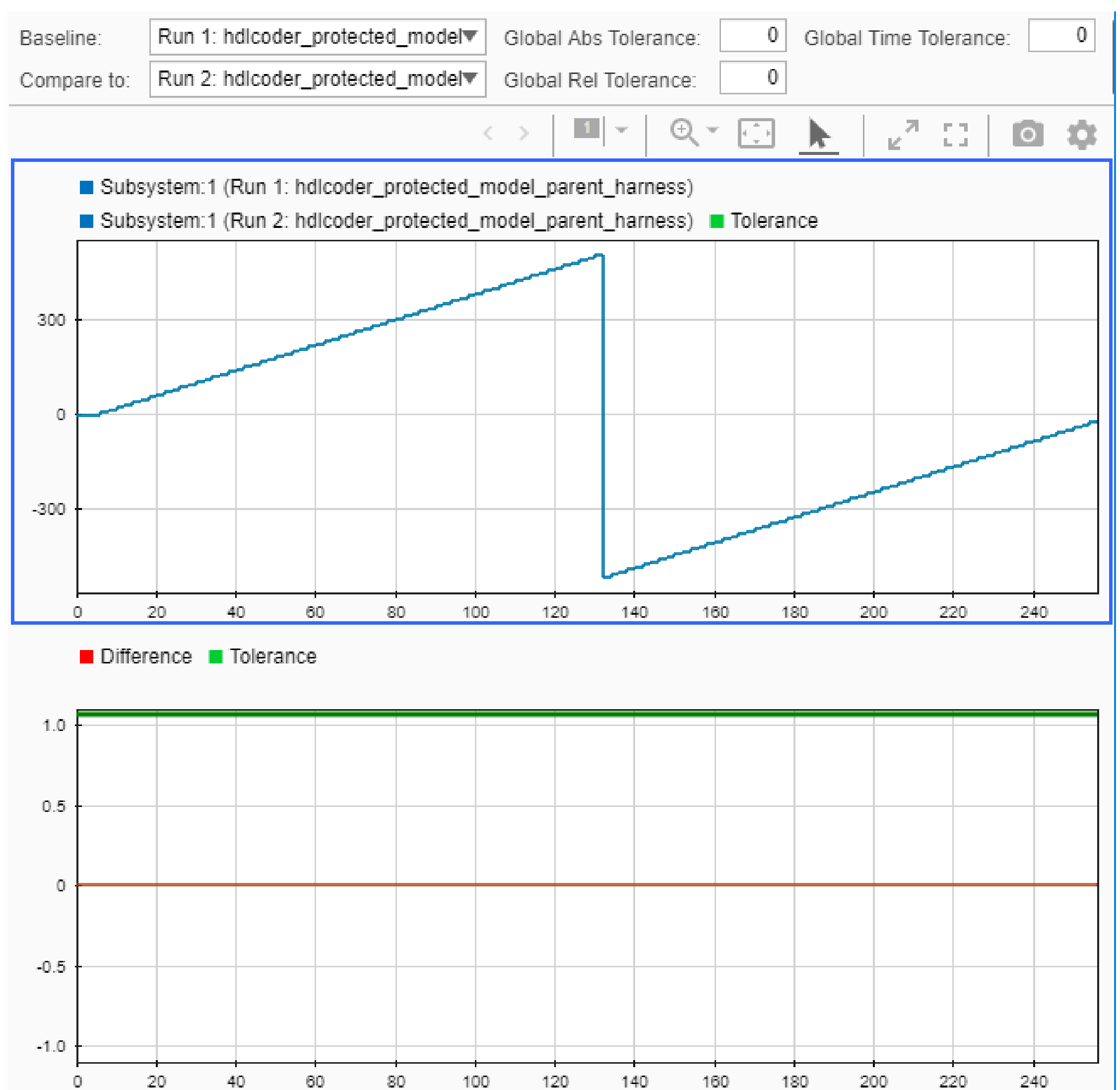
To compare the unprotected and protected versions of a Model block, you can use the Simulation Data Inspector. This example uses `hdlcoder_protected_model_parent_harness` and the protected model, `hdlcoder_referenced_model_gain.slxp`, which you created in “Create Protected Models to Conceal Contents and Generate HDL Code” on page 33-2.

1. If it is not already open, open the model `hdlcoder_protected_model_parent_harness`.

   ```
   open_system('hdlcoder_protected_model_parent_harness')
   ```
2. Make sure that the Model block in the `hdlcoder_protected_model_parent_harness/DUT/mynested` subsystem is referencing the original model `hdlcoder_referenced_model_gain.slx` and not the protected model.
3. Enable logging for the output signals of the Model block. Right-click the output signals and select **Log Selected Signals**.
4. Simulate the model and click the **Simulation Data Inspector**. In the Simulation Data Inspector, select the signals that you logged to see the simulation results. Save this simulation run with a name such as `original_model_run`.
5. Now, in the Block Parameters dialog box for the Model block, change the **Model name** to `hdlcoder_referenced_model_gain.slxp`.

   A badge icon appears on the Model block indicating that you are referencing the protected model. If you haven't already created the protected model, follow the steps mentioned in “Create Protected Models to Conceal Contents and Generate HDL Code” on page 33-2.
6. Simulate the model, which now refers to the protected model. When the simulation is complete, a new run appears in the Simulation Data Inspector. Save this run as `protected_model_run`.
7. In the Simulation Data Inspector, click the **Compare** tab. From the **Baseline** and **Compare To** lists, select the `original_model_run` and the `protected_model_run`. To compare the runs, click **Compare Runs**.

This figure displays the comparison between the **Baseline** and **Compare To** lists. You see that the simulation results match.



See Also

Functions

`Simulink.ModelReference.protect` | `Simulink.ModelReference.modifyProtectedModel`

More About

- "Save Signal Data Using Signal Logging"
- "Compare Simulation Data"
- "Package and Share Protected Models" on page 33-12

Package and Share Protected Models

When you protect a model, you can automatically create and package the following contents in a project archive (`.mlproj`) for easy sharing:

- Protected model file (`.slxp`)
- Harness model file
- MAT-file with base workspace definitions
- Data dictionary pruned to relevant definitions
- Other supporting files

In the Protected Model Creator, select **Package protected model with dependencies in a project**.

Note Before sharing the project, check whether the project contains the required supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify the missing files. Add the missing dependencies to the project and update the harness model if required.

Alternatively, you can use one of these options to deliver the protected model package:

- Create a project archive to share a project that contains the protected model file and supporting files. For more information, see “Create a Project from a Model” and “Share Projects”.
- Provide the protected model file and supporting files as separate files.
- Combine the files into a ZIP or other container file.
- Provide the files in some other standard or proprietary format specified by the receiver.

Whichever approach you use to deliver a protected model, include information on how to retrieve the original files.

Harness Model

You can create a harness model when you create your protected model. The harness model contains a Model block that references the protected model. A third party can use the Model block to reference your protected model. The harness model is set up for simulation of the protected model.

MAT-File with Base Workspace Definitions

Referenced models can use object definitions or tunable parameters that are defined in the MATLAB base workspace. These variables are not saved with the model. When you protect a model, you must obtain the definitions of required base workspace entities and ship them with the model.

For example, if the model uses the following base workspace variables, they must be saved to a MAT-file:

- Global tunable parameter
- Global data store
- The following objects used by a signal that connects to a root-level model Inport or Outport:

- `Simulink.Signal`
- `Simulink.Bus`
- `Simulink.Alias`
- `Simulink.NumericType` that is an alias

To determine the required base workspace definitions and save them to a MAT-file, see “Explore Protected Model Capabilities”. Before executing the protected model as a part of a third-party model, the receiver of the protected model must load the MAT-file.

Simulink Data Dictionary

Referenced models can use data definitions from a data dictionary, which are not saved with the model. When you protect a model that uses a data dictionary, package and ship the data dictionary with the protected model.

Protected Model File Contents

A protected model file (`.slxp`) consists of the derived files that support the options that you select when you create the protected model. The derived files are unpacked when you or a third party use the protected model in simulation or code generation.

The derived files that are unpacked depends on the support that you enabled when creating the protected model. The `slprj/sim/`*`model`*`/*` files are deleted after they are used.

This table provides the files that are unpacked depending on the options that you specify. If you specify the **Code generation** or **Code interface** options when you create the protected model, additional files are unpacked in the derived folder. To learn about these files, see “Protected Model File Contents”.

Supported Functionality	Derived Files
Simulation only and the referencing model simulates in normal mode	The *`model`*`.mexext` file is placed in the build folder.
Simulation only and the referencing model simulates in accelerator or rapid accelerator mode	These files are unpacked in the `slprj/sim/` folder: • `slprj/sim/`*`model`*`/*.h` • `slprj/sim/`*`model`*`/`*`model`*`lib.a` (or *`model`*`lib.lib`) • `slprj/sim/`*`model`*`/tmwinternal/*` • `slprj/sim/_sharedutils/*` For the protected model report, these additional files are unpacked (but not in the build folder): • `slprj/sim/`*`model`*`/html/*` • `slprj/sim/`*`model`*`/buildinfo.mat`

Supported Functionality	Derived Files
HDL code generation	These files are unpacked in the `hdlsrc` folder: (Additional files depend on whether you enabled support for other options such as code generation). • `hdlsrc/`*`model/model.vhd`* (*`model.v`* if you specified `Verilog` as the **Target language**.) • `hdlsrc/`*`model/Subsystem.vhd`* (*`Subsystem.v`* if you specified `Verilog` as the **Target language** of the model that you protected. The additional HDL files depend on how hierarchically the referenced model was designed.) • `hdlsrc/`*`model/model_pkg.vhd`* (This file is not generated if you specified `Verilog` as the **Target language** of the model that you protected.) • `hdlsrc/`*`model/model_report.html`* • `hdlsrc/`*`model/gm_model.slxp`* (This file is a generated protected model. If you use cosimulation, HDL Coder software instantiates this generated protected model.)

See Also

Functions

`Simulink.ModelReference.protect` | `Simulink.ModelReference.modifyProtectedModel`

More About

- “Create Protected Models to Conceal Contents and Generate HDL Code” on page 33-2
- “Test Protected Models” on page 33-10

Obfuscate Generated HDL Code from Simulink Models

To share HDL code with a third party without revealing the intellectual property, you can generate obfuscated HDL code from Simulink models. Obfuscation reduces readability of the code. The generated HDL code does not have any comments, newlines, or spaces, and replaces identifier names with random names.

How to Generate Obfuscated HDL Code

By default, the generated HDL code is not obfuscated. The HDL code contains newlines, comments, and is readable.

To generate obfuscated HDL code for the DUT subsystem in your model:

1. In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
2. Open the **HDL Code Generation** pane of the Configuration Parameters dialog box. In the **HDL Code** tab, select **Settings > HDL Code Generation Settings**.
3. Specify generation of obfuscated HDL code. In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Coding Style > RTL Style** section, select **Generate obfuscated HDL code**.
4. Generate HDL code. Select the DUT subsystem as the **Code for** subsystem, and then click the **Generate HDL Code** button.

To generate obfuscated HDL code from the command line, use the `ObfuscateGeneratedHDLCode` property with `hdlset_param` or `makehdl`. For example, to generate obfuscated HDL code for the `symmetric_fir` subsystem in the `sfir_fixed` model:

```
makehdl('sfir_fixed/symmetric_fir', 'ObfuscateGeneratedHDLCode', 'on')

% To generate obfuscated Verilog code, set 'Targetlanguage' to 'Verilog'
makehdl('sfir_fixed/symmetric_fir', 'TargetLanguage', 'Verilog', ...
                                       'ObfuscateGeneratedHDLCode', 'on')
```

Generated HDL Code with Obfuscation

By default, the generated HDL code is not obfuscated. For example, this code shows the generated VHDL code for the **Complex Multiplier** model template in Simulink. To learn more about this template, see “Use Simulink Templates for HDL Code Generation” on page 14-8.

```
...

-- -------------------------------------------------------------
--
-- Module: HDL_Complex_Multiplier
-- Source Path: untitled/HDL_Complex_Multiplier
-- Hierarchy Level: 0
--
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY HDL_Complex_Multiplier IS
  PORT( ...


        X_re            :   IN    std_logic_vector(17 DOWNTO 0);  -- sfix18_En17
```

```
        ... );
END HDL_Complex_Multiplier;

...
```

To generate obfuscated HDL code, enable HDL code obfuscation and then generate code. For example, this code shows entity names and port names that are obfuscated in the generated VHDL code.

```
LIBRARY IEEE; ... ENTITY Q1LNc1j7NFXR IS PORT(EEY54qLw4C0j9uD:IN std_logic_vector(17 DOWNTO 0); ...
```

Code Obfuscation Report

When you specify generation of obfuscated HDL code, and then generate code, HDL Coder produces a Code Obfuscation report. The Code Obfuscation report displays the status of HDL code obfuscation. It also displays whether the model uses configuration parameters that are incompatible with code obfuscation and provides a link to disable these parameters. These parameters are ignored during the obfuscation process.

HDL Model Parameters Incompatible with Code Obfuscation

HDL code obfuscation is not compatible with certain Configuration Parameters and ignores these parameters if they are enabled on the model. The parameters include:

- These parameters in the **HDL Code Generation** > **Global Settings** > **General** pane:
  - Enable prefix
  - Clocked process postfix
  - Timing controller postfix
  - Split entity file postfix
  - Split arch file postfix
  - Split entity and architecture
  - Complex real part postfix
  - Complex imaginary part postfix
  - Pipeline postfix
  - Instance prefix
  - Instance postfix
  - Entity conflict postfix
  - Package postfix
  - Reserved word postfix
  - Module name prefix
  - Pipeline postfix
  - Block generate label
  - Output generate label
  - Instance generate label

  - Vector prefix
  - Instance postfix
  - Instance prefix
  - Map file postfix
- These parameters in the **HDL Code Generation** > **Global Settings** > **Coding Standards** tab:
  - HDL coding standard
  - Show passing rules in coding standard report
  - Check for duplicate names
  - Check for HDL keywords in design names
  - Check module, instance, entity name length
  - Check signal, port, and parameter name length
  - Check for clock enable signals
  - Detect usage of reset signals
  - Detect usage of asynchronous reset signals
  - Check for conditional statements in processes
  - Check for assignments to the same variable in multiple cascaded control regions
  - Check if-else statement chain length
  - Check if-else statement nesting depth
  - Minimize use of variables
  - Check for initial statements that set RAM initial values
  - Check multiplier width
- These parameters in the **HDL Code Generation** > **Global Settings** > **Coding Style** tab:
  - Enable Comments
  - Include requirements in block comments
  - Emit time/date stamp in header
  - Custom File Header Comment
  - Custom File Footer Comment
- The Generate traceability report parameter in the **HDL Code Generation** > **Report** pane.

Code Obfuscation Considerations and Restrictions

- Synthesizing the obfuscated HDL code might produce different synthesis results from the synthesis results of the original HDL code. For best results, perform synthesis on the original code instead of the obfuscated code.
- HDL code obfuscation replaces only names corresponding to HDL files, signals, blocks, variable names, or ports with random names. Other identifier names are not replaced, such as names of vectors or enumerations.
- For some interfaces that you use in your Simulink model, the interface information such as the port names and interface names are preserved in the obfuscated HDL code. These names are not obfuscated. The interfaces include:

  - DUT
  - Model reference
  - Black box
  - Xilinx or Intel floating-point target
- You cannot obfuscate the HDL code generated for these blocks:
  - CIC Interpolation
  - FIR Decimation
  - FIR Interpolation

See Also

Functions
`makehdl` | `makehdltb`

Model Settings
Generate obfuscated HDL code

More About

- “Create HDL-Compatible Simulink Model”
- “Generate HDL Code from Simulink Model”

HDL Test Bench

- "Verify Generated Code Using HDL Test Bench from Configuration Parameters" on page 34-2
- "Verify Generated Code Using HDL Test Bench at Command Line" on page 34-9
- "Test Bench Generation" on page 34-15
- "Test Bench Block Restrictions" on page 34-17

Verify Generated Code Using HDL Test Bench from Configuration Parameters

In this section...

"FIR Filter Model" on page 34-2
"Create a Folder and Copy Relevant Files" on page 34-4
"What is a HDL Test Bench?" on page 34-5
"How to Verify the Generated Code" on page 34-5
"Generate HDL Test Bench" on page 34-5
"View HDL Test Bench Files" on page 34-6
"Run Simulation and Verify Generated HDL Code" on page 34-7

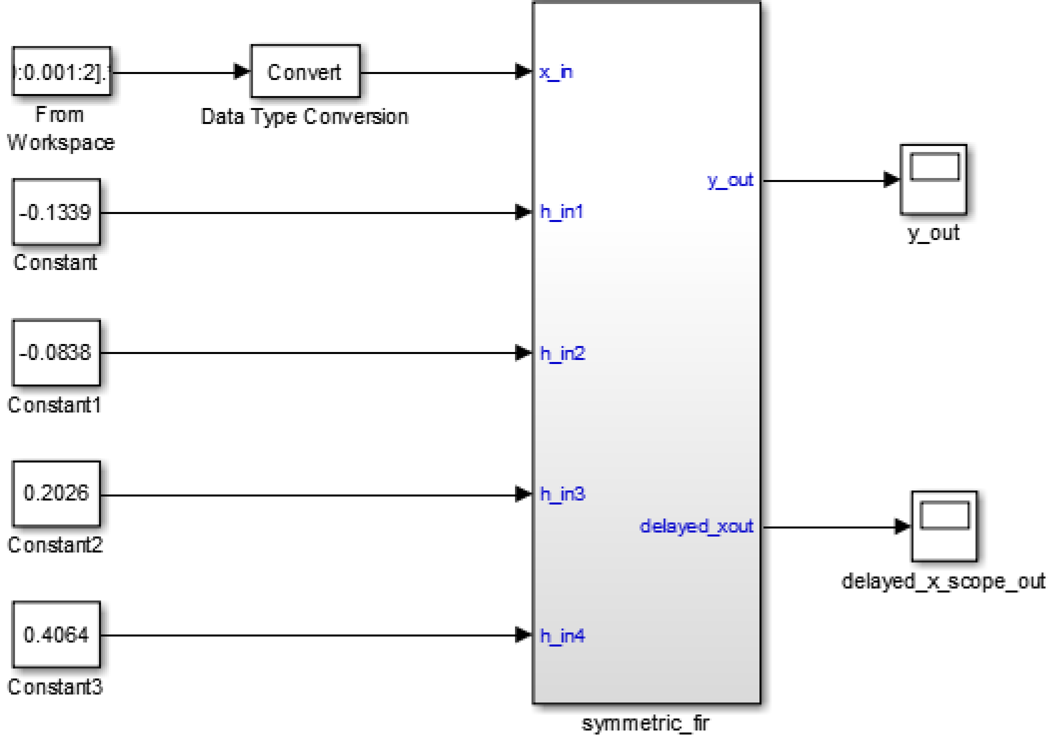
This example shows how to generate a HDL test bench and verify the generated code for your design. The example assumes that you have generated HDL code for your model. If you haven't generated HDL code, you can still open this model and generate the HDL test bench. Before generating the test bench, HDL Coder runs code generation to make sure that there is at least one successful code generation run before generating the testbench.

This example illustrates how to verify the generated code for the FIR filter model. To learn how to generate HDL code for this model, see "Generate HDL Code from Simulink Model Using Configuration Parameters" on page 16-6.

FIR Filter Model

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



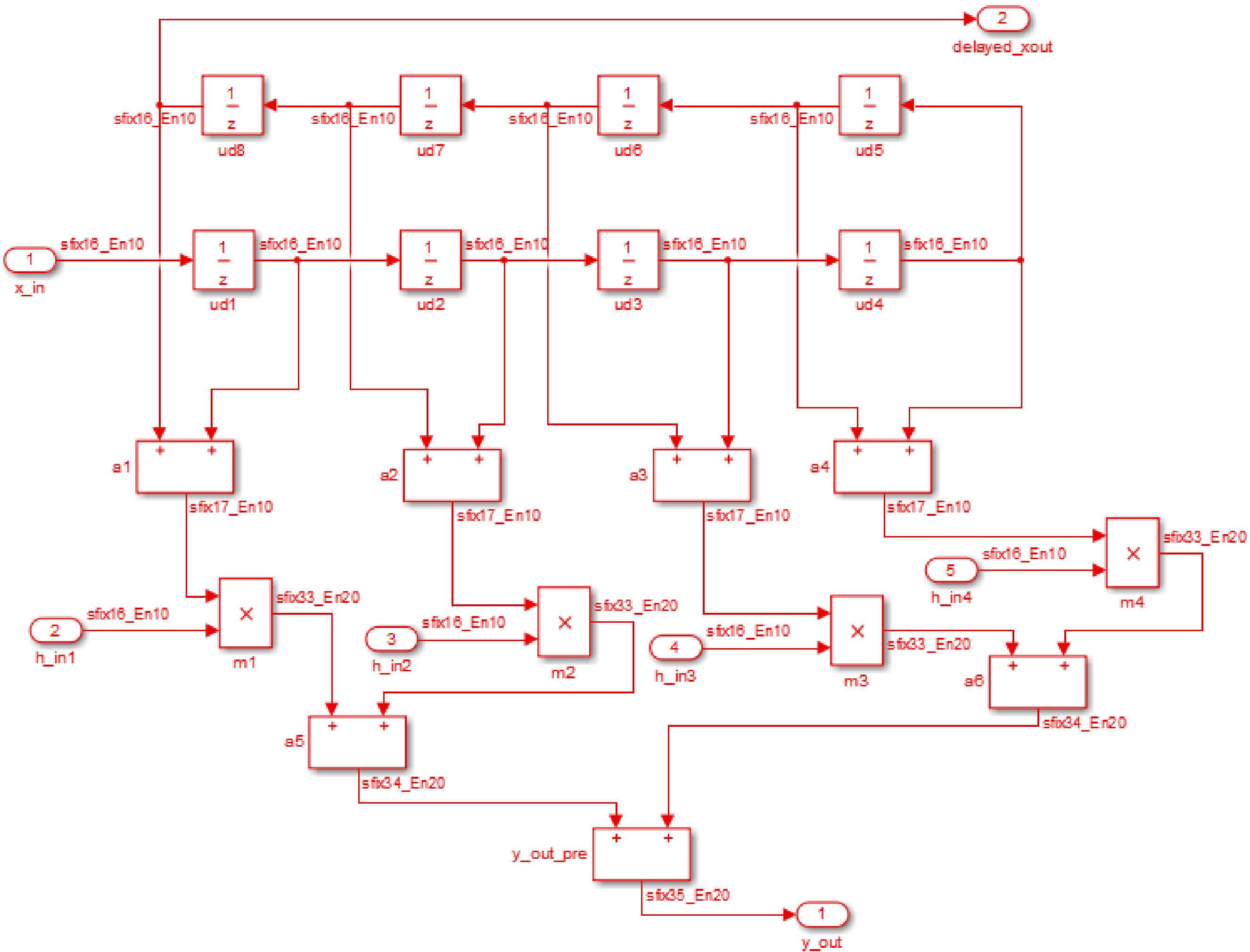
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



Create a Folder and Copy Relevant Files

In MATLAB:

1 Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

`sl_hdlcoder_work` stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

2 Make the `sl_hdlcoder_work` folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

3 Save a local copy of the `sfir_fixed` model to your current working folder. Leave the model open.

What is a HDL Test Bench?

To verify the functionality of the HDL code that you generated for the DUT, generate a HDL test bench. A test bench includes:

- Stimulus data generated by signal sources connected to the entity under test.
- Output data generated by the entity under test. During a test bench run, this data is compared to the outputs of the VHDL code, for verification purposes.
- Clock, reset, and clock enable inputs to drive the entity under test.
- A component instantiation of the entity under test.
- Code to drive the entity under test and compare its outputs to the expected data.

You can simulate the generated test bench and script files with the Mentor Graphics ModelSim simulator.

How to Verify the Generated Code

This example illustrates how to generate a HDL test bench to simulate and verify the generated HDL code for your design. You can also verify the generated HDL code from your model using these methods:

Verification Method	For More Information
Validation Model	"Generated Model and Validation Model" on page 21-10
HDL Cosimulation (requires HDL Verifier)	"Cosimulation"
SystemVerilog DPI Test Bench (requires HDL Verifier)	"SystemVerilog DPI Test Bench"
FPGA-in-the-Loop (requires HDL Verifier)	"FPGA-in-the-Loop"

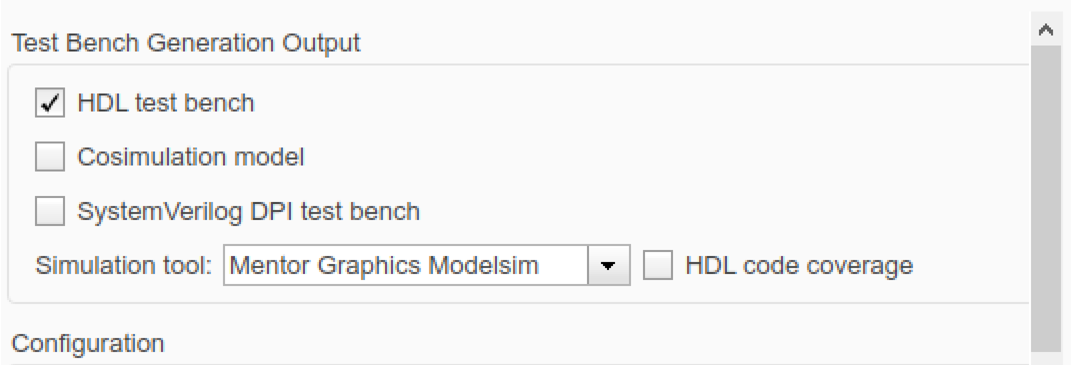
Generate HDL Test Bench

Depending on whether you generated VHDL, Verilog or SystemVerilog code, generate VHDL, Verilog or SystemVerilog test bench code. The test bench code drives the HDL code that you generated for the DUT. By default, the HDL code and the test bench code are written to the same target folder `hdlsrc` relative to the current folder.

For the FIR filter, the **symmetric_fir** subsystem is the DUT. To generate the testbench, select this subsystem. You cannot generate a HDL testbench for an entire model.

1. In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
2. Select the DUT Subsystem in your model, and make sure that this Subsystem name appears in the **Code for** option. To remember the selection, you can pin this option. Click **Generate Testbench**.

By default, HDL Coder generates VHDL testbench code in the target `hdlsrc` folder.



Generate Verilog or SystemVerilog Test Bench Code

If you want to generate Verilog or SystemVerilog test bench code, you can specify this setting in the **HDL Code Generation** pane of the Configuration Parameters dialog box.

To generate Verilog testbench code for the counter model:

1. In the **HDL Code** tab, click **Settings**.
2. In the **HDL Code Generation** pane, for **Language**, select `Verilog` or `SystemVerilog`. Leave other settings to the default.
3. In the **HDL Code Generation** > **Test Bench**pane, click **Generate Test Bench**.

If you haven't already generated code for your model, HDL Coder compiles the model and generates HDL code before generating the test bench. Depending on model display options such as port data types, the model can change in appearance after code generation.

As test bench generation proceeds, HDL Coder displays progress messages. The process should complete with the message

```
### HDL TestBench Generation Complete.
```

After generating the test bench, you see the generated files in the `hdlsrc` folder.

View HDL Test Bench Files

- `symmetric_fir_tb.vhd`: VHDL test bench code, with generated test and output data. If you generated Verilog or SystemVerilog test bench code, the generated file is `symmetric_fir_tb.v` or `symmetric_fir_tb.sv`.
- `symmetric_fir_tb_pkg.vhd`: Package file for VHDL test bench code. If you generated SystemVerilog test bench code, the generated file is `symmetric_fir_tb_pkg.sv`. This file is not generated if you specified Verilog as the target language.
- `symmetric_fir_tb_compile.vhd`: Compilation script (vcom commands). This script compiles and loads the entity to be tested (symmetric_fir.vhd) and the test bench code (symmetric_fir_tb.vhd)..
- `symmetric_fir_tb_sim.do`: Mentor Graphics ModelSim script to initialize the simulator, set up **wave** window signal displays, and run a simulation.

To view the generated test bench code in the MATLAB Editor, double-click the `symmetric_fir_tb.vhd` or `symmetric_fir_tb.v` file in the current folder.

Run Simulation and Verify Generated HDL Code

To verify the simulation results, you can use the Mentor Graphics ModelSim simulator. Make sure that you have already installed Mentor Graphics ModelSim.

To launch the simulator, use the `vsim` (HDL Verifier) function. This command shows how to open the simulator by specifying the path to the executable:

```
vsim('vsimdir','C:\Program Files\ModelSim\questasim\10.6b\win64\vsim.exe')
```

To compile and run a simulation of the generated model and test bench code, use the scripts that are generated by HDL Coder. Following example illustrates the commands that compile and simulate the generated test bench for the `sfir_fixed/symmetric_fir` subsystem.

1 Open the Mentor Graphics ModelSim software and navigate to the folder that has the previously generated code files and the scripts.

```
QuestaSim>cd C:/work/sl_hdlcoder_work/hdlsrc
```

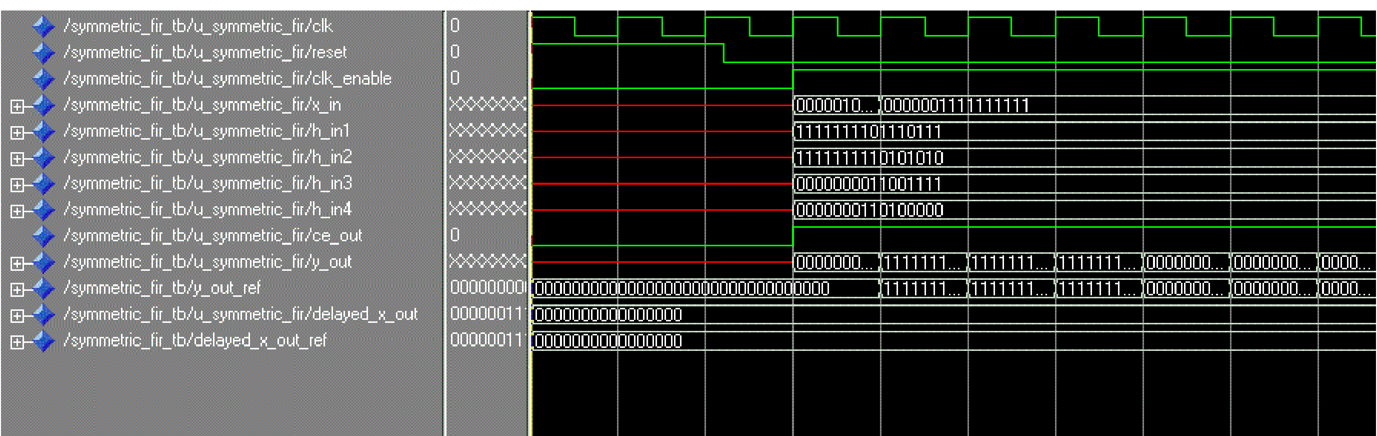
2 Use the generated compilation script to compile and load the generated model and text bench code. Run this command to compile the generated code.

```
QuestaSim>do symmetric_fir_tb_compile.do
```

3 Use the generated simulation script to execute the simulation. The following listing shows the command and responses. You can ignore any warning messages. The test bench termination message indicates that the simulation has run to completion without comparison errors. Run this command to simulate the generated code.

```
QuestaSim>do symmetric_fir_tb_sim.do
```

The simulator optimizes your design and displays the results of simulating your HDL design in a **wave** window. if you don't see the simulation results, open the **wave** window. The simulation script displays inputs and outputs in the model including the clock , reset, and clock enable signals in the **wave** window.



You can now view the signals and verify that the simulation results match the functionality of your original design. After verifying, close the Mentor Graphics ModelSim simulator, and then close the files that you have opened in the MATLAB Editor.

See Also

Functions
`makehdl` | `makehdltb`

Model Settings
HDL test bench | Cosimulation model | SystemVerilog DPI test bench | Simulation tool | HDL code coverage

More About

- "HDL Test Bench"
- "HDL Code Generation and FPGA Synthesis from Simulink Model"

Verify Generated Code Using HDL Test Bench at Command Line

In this section...

"FIR Filter Model" on page 34-9
"Create a Folder and Copy Relevant Files" on page 34-11
"What is a HDL Test Bench?" on page 34-12
"How to Verify the Generated Code" on page 34-12
"Generate HDL Test Bench" on page 34-12
"View HDL Test Bench Files" on page 34-13
"Run Simulation and Verify Generated HDL Code" on page 34-13

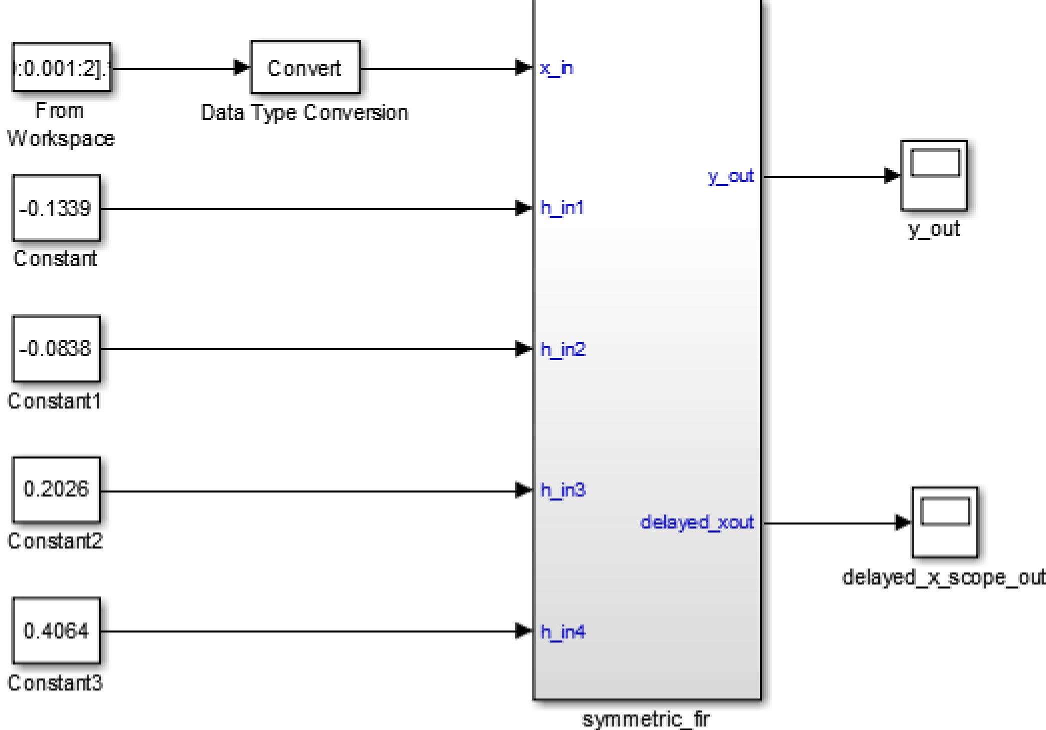
This example shows how to generate a HDL test bench and verify the generated code for your design. The example assumes that you have already generated HDL code for your model. If you haven't already generated HDL code, you can still open this model and generate the HDL test bench. Before generating the test bench, HDL Coder runs code generation to make sure that there is at least one successful code generation run before generating the testbench.

This example illustrates how to verify the generated code for the FIR filter model. To learn how to generate HDL code, see "Generate HDL Code from Simulink Model from Command Line" on page 16-10.

FIR Filter Model

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



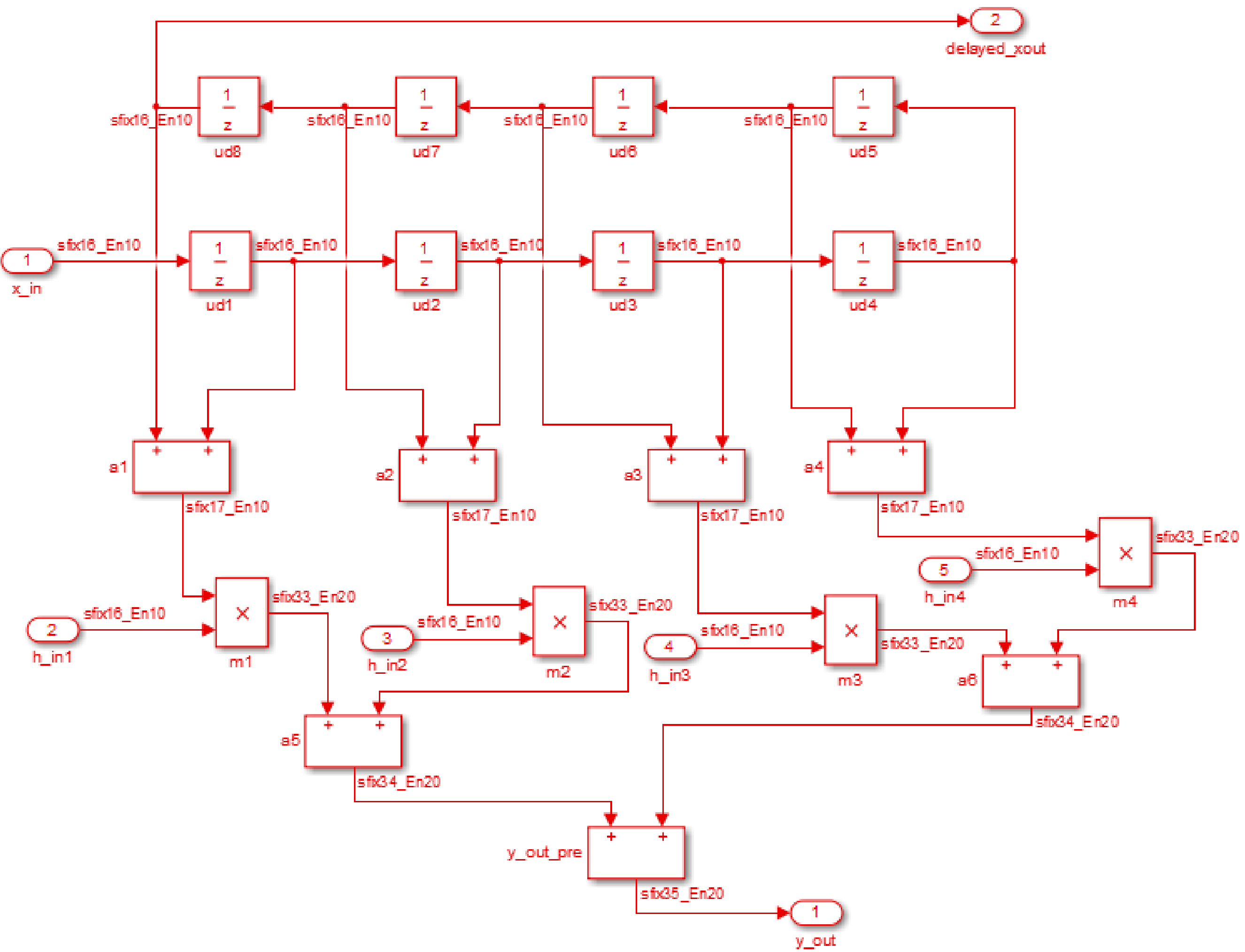
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



Create a Folder and Copy Relevant Files

In MATLAB:

1. Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

   `sl_hdlcoder_work` stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

2. Make the `sl_hdlcoder_work` folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

3. Save a local copy of the `sfir_fixed` model to your current working folder. Leave the model open.

What is a HDL Test Bench?

To verify the functionality of the HDL code that you generated for the DUT, generate a HDL test bench. A test bench includes:

- Stimulus data generated by signal sources connected to the entity under test.
- Output data generated by the entity under test. During a test bench run, this data is compared to the outputs of the VHDL code, for verification purposes.
- Clock, reset, and clock enable inputs to drive the entity under test.
- A component instantiation of the entity under test.
- Code to drive the entity under test and compare its outputs to the expected data.

You can simulate the generated test bench and script files with the Mentor Graphics ModelSim simulator.

How to Verify the Generated Code

This example illustrates how to generate a HDL test bench to simulate and verify the generated HDL code for your design. You can also verify the generated HDL code from your model using these methods:

Verification Method	For More Information
Validation Model	"Generated Model and Validation Model" on page 21-10
HDL Cosimulation (requires HDL Verifier)	"Cosimulation"
SystemVerilog DPI Test Bench (requires HDL Verifier)	"SystemVerilog DPI Test Bench"
FPGA-in-the-Loop (requires HDL Verifier)	"FPGA-in-the-Loop"

Generate HDL Test Bench

Depending on whether you generated VHDL, Verilog or SystemVerilog code, generate VHDL, Verilog or SystemVerilog test bench code. The test bench code drives the HDL code that you generated for the DUT. By default, the HDL code and the test bench code are written to the same target folder `hdlsrc` relative to the current folder.

To generate test bench code and the scripts for compilation and simulation, use the `makehdltb` function. At the command line, enter:

```
makehdltb('sfir_fixed/symmetric_fir')
```

To specify the customizations before you generate testbench code, use the `hdlset_param` function. You can also specify various name-value pair arguments with the `makehdltb` function to customize HDL code generation options while generating HDL code. For example, to generate Verilog test bench code, use the `TargetLanguage` property.

```
makehdltb('sfir_fixed/symmetric_fir', 'TargetLanguage', 'Verilog')
```

Alternatively, if you are using `hdlset_param`, set this parameter on the model and then run the `makehdltb` function.

```
hdlset_param('sfir_fixed', 'TargetLanguage', 'Verilog')
makehdltb('sfir_fixed/symmetric_fir')
```

If you haven't already generated code for your model, HDL Coder compiles the model and generates HDL code before generating the test bench. Depending on model display options such as port data types, the model can change in appearance after code generation.

As test bench generation proceeds, HDL Coder displays progress messages. The process should complete with the message

```
### HDL TestBench Generation Complete.
```

After generating the test bench, you see the generated files in the `hdlsrc` folder.

View HDL Test Bench Files

- `symmetric_fir_tb.vhd`: VHDL test bench code, with generated test and output data. If you generated Verilog or SystemVerilog test bench code, the generated files are `symmetric_fir_tb.v` or `symmetric_fir_tb.sv`.
- `symmetric_fir_tb_pkg.vhd`: Package file for VHDL test bench code. If you generated SystemVerilog test bench code, the generated file is `symmetric_fir_tb_pkg.sv`. This file is not generated if you specified Verilog as the target language.
- `symmetric_fir_tb_compile.vhd`: Compilation script (`vcom` commands). This script compiles and loads the entity to be tested (`symmetric_fir.vhd`) and the test bench code (`symmetric_fir_tb.vhd`).
- `symmetric_fir_tb_sim.do`: Mentor Graphics ModelSim script to initialize the simulator, set up **wave** window signal displays, and run a simulation.

To view the generated test bench code in the MATLAB Editor, double-click the `symmetric_fir_tb.vhd`, `symmetric_fir_tb.v` or `symmetric_fir_tb.sv` file in the current folder.

Run Simulation and Verify Generated HDL Code

To verify the simulation results, you can use the Mentor Graphics ModelSim simulator. Make sure that you have already installed Mentor Graphics ModelSim.

To launch the simulator, use the `vsim` (HDL Verifier) function. This command shows how to open the simulator by specifying the path to the executable:

```
vsim('vsimdir','C:\Program Files\ModelSim\questasim\10.6b\win64\vsim.exe')
```

To compile and run a simulation of the generated model and test bench code, use the scripts that are generated by HDL Coder. Following example illustrates the commands that compile and simulate the generated test bench for the `sfir_fixed/symmetric_fir` subsystem.

1. Open the Mentor Graphics ModelSim software and navigate to the folder that has the previously generated code files and the scripts.

   ```
   QuestaSim>cd C:/work/sl_hdlcoder_work/hdlsrc
   ```

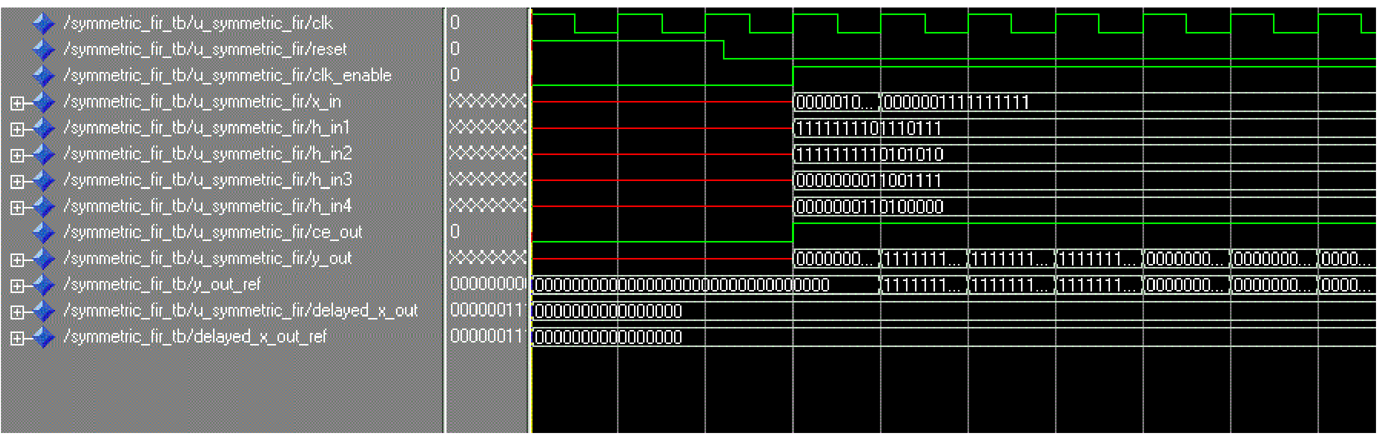
2. Use the generated compilation script to compile and load the generated model and text bench code. Run this command to compile the generated code.

```
QuestaSim>do symmetric_fir_tb_compile.do
```

3. Use the generated simulation script to execute the simulation. The following listing shows the command and responses. You can ignore any warning messages. The test bench termination message indicates that the simulation has run to completion without comparison errors. Run this command to simulate the generated code.

```
QuestaSim>do symmetric_fir_tb_sim.do
```

The simulator optimizes your design and displays the results of simulating your HDL design in a **wave** window. if you don't see the simulation results, open the **wave** window. The simulation script displays inputs and outputs in the model including the clock , reset, and clock enable signals in the **wave** window.



You can now view the signals and verify that the simulation results match the functionality of your original design. After verifying, close the Mentor Graphics ModelSim simulator, and then close the files that you have opened in the MATLAB Editor.

See Also

Functions
`makehdl` | `makehdltb`

Model Settings
HDL test bench | **Cosimulation model** | **SystemVerilog DPI test bench** | **Simulation tool** | **HDL code coverage**

More About

- “HDL Test Bench”
- “HDL Code Generation and FPGA Synthesis from Simulink Model”

Test Bench Generation

You can generate a HDL Testbench for a subsystem or model reference that you specify in your Simulink model. The coder generates an HDL test bench by running a Simulink simulation to capture input vectors and expected output data for your DUT.

How Test Bench Generation Works

HDL Coder writes the DUT stimulus and reference data from your MATLAB or Simulink simulation to data files (`.dat`).

During HDL simulation, the HDL test bench reads the saved stimulus from the `.dat` files. The test bench compares the actual DUT output with the expected output, which is also saved in .dat files. After you generate code, the message window displays links to the test bench data files.

Reference data is delayed by one clock cycle in the waveform viewer compared to default test bench generation due to the delay in reading data from files.

Test Bench Data Files

The coder saves stimulus and reference data for each DUT input and output in a separate test bench data file (`.dat`), with the following exceptions:

- Two files are generated for the real and imaginary parts of complex data.
- Constant DUT input data is written to the test bench as constants.

Vector input or output data is saved as a single file.

Test Bench Data Type Limitations

If you have double, single, or enumeration data types at the DUT inputs and outputs, the simulation data is generated as constants in the test bench code, instead of writing the simulation data to files.

Use Constants Instead of File I/O

You can generate test bench stimulus and reference data as constants in the test bench code instead of using file I/O. Simulating a long running test bench that uses constants requires more memory than a test bench that uses file I/O.

If your DUT inputs or outputs use data types that are not supported for file I/O, test bench generation automatically generates data as constants. For details, see “Test Bench Data Type Limitations” on page 34-15.

Using the HDL Workflow Advisor

To generate a test bench that uses constants:

1. In the **HDL Code Generation > Set Code Generation Options > Set Testbench Options** task, clear **Use file I/O to read/write test bench data** and click **Apply**.
2. In the **HDL Code Generation > Generate RTL Code and Testbench** task, select **Generate RTL testbench** and click **Apply**.

Using the Command Line

To generate a test bench that uses constants, use the `UseFileIOInTestBench` parameter with `makehdltb`.

For example, to generate a Verilog test bench by using constants for a DUT subsystem, `sfir_fixed/symmetric_fir`, enter:

```
makehdltb('sfir_fixed/symmetric_fir','TargetLanguage','Verilog',...
          'UseFileIOInTestBench','off');
```

See Also

`makehdltb`

More About

- “Test Bench Block Restrictions” on page 34-17
- “Choose a Test Bench for Generated HDL Code” on page 25-41

Test Bench Block Restrictions

Blocks that belong to the blocksets and toolboxes in the following list should not be directly connected to the DUT. Instead, place them in a subsystem, and connect the subsystem to the DUT. This restriction applies to all blocks in the following products:

- RF Blockset™
- Simscape Driveline™
- SimEvents®
- Simscape Multibody
- Simscape Electrical Power Systems
- Simscape

FPGA Board Customization

- “FPGA Board Customization” on page 35-2
- “Create Custom FPGA Board Definition” on page 35-6
- “Create Xilinx KC705 Evaluation Board Definition File” on page 35-7
- “FPGA Board Manager” on page 35-18
- “New FPGA Board Wizard” on page 35-22
- “FPGA Board Editor” on page 35-33

FPGA Board Customization

In this section...
"Feature Description" on page 35-2
"Custom Board Management" on page 35-2
"FPGA Board Requirements" on page 35-2

Feature Description

Both HDL Coder and HDL Verifier software include a set of predefined FPGA boards you can use with the Turnkey or FPGA-in-the-loop (FIL) workflows. You can view the lists of these supported boards in the HDL Workflow Advisor or in the FIL wizard. With the FPGA Board Manager, you can add additional boards to use either of these workflows. To add a board, you need the relevant information from the board specification documentation.

The FPGA Board Manager is the hub for accessing wizards and dialog boxes that take you through the steps necessary to create a custom board configuration. You can also access options for:

- Importing a custom board
- Copying a board definition file for further modification
- Verifying a new board

Custom Board Management

You manage FPGA custom boards through the following user interfaces:

- "FPGA Board Manager" on page 35-18: portal to adding, importing, deleting, and otherwise managing board definition files.
- "New FPGA Board Wizard" on page 35-22: This wizard guides you through creating a custom board definition file with information you obtain from the board specification documentation.
- "FPGA Board Editor" on page 35-33: user interface for viewing or editing board information.

To begin, review the "FPGA Board Requirements" on page 35-2 and then follow the steps described in "Create Custom FPGA Board Definition" on page 35-6.

FPGA Board Requirements

- "FPGA Device" on page 35-2
- "FPGA Design Software" on page 35-3
- "General Hardware Requirements" on page 35-3
- "Ethernet Connection Requirements for FPGA-in-the-Loop" on page 35-3
- "JTAG Connection Requirements for FPGA-in-the-Loop" on page 35-5

FPGA Device

To view a current list of supported FPGA device families with FPGA-in-the-loop (FIL), see "Supported FPGA Device Families for Board Customization" (HDL Verifier).

FPGA Design Software

Altera Quartus II or Xilinx ISE is required. See product documentation for HDL Coder or HDL Verifier for the specific software versions required.

The following MathWorks tools are required to use FIL.

Workflow	Required Tools
FPGA-in-the-loop	• HDL Verifier • Fixed-Point Designer

General Hardware Requirements

To use an FPGA development board, make sure that you have the following FPGA resources:

- **Clock**: An external clock connected to the FPGA is required. The clock can be differential or single-ended. The accepted clock frequency is from 5 MHz to 300 MHz. When used with FIL, there are additional requirements to the clock frequency (see “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 35-3).
- **Reset**: An external reset signal connected to the FPGA is optional. When supplied, this signal functions as the global reset to the FPGA design.
- **JTAG download cable**: A JTAG download cable that connects host computer and FPGA board is required for the FPGA programming. The FPGA must be programmable using Xilinx iMPACT or Altera Quartus II.

Ethernet Connection Requirements for FPGA-in-the-Loop

- “Supported Ethernet PHY Device” on page 35-3
- “Ethernet PHY Interface” on page 35-4
- “Special Timing Considerations for RGMII” on page 35-4
- “Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface” on page 35-4

Supported Ethernet PHY Device

On the FPGA board, the Ethernet MAC is implemented in FPGA. An Ethernet PHY chip is required to be on the FPGA board to connect the physical medium to the Media Access (MAC) layer in the FPGA.

Note When programming the FPGA, HDL Verifier assumes that there is only one download cable connected to the Host computer. It also assumes that the FPGA programming software automatically recognizes the cable. If not, use FPGA programming software to program your FPGA with the correct options.

The FIL feature is tested with the following Ethernet PHY chips and may not work with other Ethernet PHY devices.

Ethernet PHY Chip	Test
Marvell® Alaska 88E1111	For GMII, RGMII, SGMII, and 100 Base-T MII interfaces
National Semiconductor DP83848C	For 100 Base-T MII interface only

Ethernet PHY Interface

The Ethernet PHY chip must be connected to the FPGA using one of the following interfaces:

Interface	Note
Gigabit Media Independent Interface (GMII)	Only 1000 Mbits/s speed is supported using this interface.
Reduced Gigabit Media Independent Interface (RGMII)	Only 1000 Mbits/s speed is supported using this interface.
Serial Gigabit Media Independent Interface (SGMII)	Only 1000 Mbits/s speed is supported using this interface.
Media Independent Interface (MII)	Only 100 Mbits/s speed is supported using this interface.

Note For GMII, the TXCLK (clock signal for 10/100 Mbits signal) signal is not required because only 1000 Mbits/s speed is supported.

In addition to the standard GMII/RGMII/SGMII/MII interface signals, FPGA-in-the-loop also requires an Ethernet PHY chip reset signal (ETH_RESET_n). This active-low reset signal performs the PHY hardware reset by FPGA. It is active-low.

Special Timing Considerations for RGMII

When the RGMII interface is used, the MAC on the FPGA assumes that the data are aligned with the edges of reference clock as specified in the original RGMII v1.3 standard. In this case, PC board designs provide additional trace delay for clock signals.

The RGMII v2.0 standard allows the transmitter to integrate this delay so that PC board delay is not required. Marvell Alaska 88E1111 has internal registers to add internal delays to RX and TX clocks. The internal delays are not added by default, which means that you must use the MDIO module to configure Marvell 88E1111 to add internal delays. For more information on the MDIO module, see “FIL I/O” on page 35-26.

Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface

When GMII/RGMII/SGMII interfaces are used, the FPGA requires an exact 125 MHz clock to drive the 1000 Mbits/s communication. This clock is derived from the user supplied external clock using the clock module or PLL.

Not all external clock frequencies can derive an exact 125 MHz clock frequency. The acceptable clock frequencies vary depending on the FPGA device family. The recommended clock frequencies are 50, 100, 125, and 200 MHz.

JTAG Connection Requirements for FPGA-in-the-Loop

Vendor	Required Hardware	Required Software
Intel	USB Blaster I or USB Blaster II download cable	• USB Blaster I or II driver • For Windows operating systems: Quartus Prime executable directory must be on system path. • For Linux® operating systems: versions below Quartus II 13.1 are not supported. Quartus II 14.1 is not supported. Only 64-bit Quartus is supported. Quartus library directory must be on `LD_LIBRARY_PATH` *before* starting MATLAB. Prepend the Linux distribution library path before the Quartus library on `LD_LIBRARY_PATH`. For example, `/lib/x86_64-linux-gnu:$QUARTUS_PATH`.
Xilinx	Digilent® download cable • If your board has an onboard Digilent USB-JTAG module, use a USB cable • If your board has a standard Xilinx 14 pin JTAG connector, use with HS2 or HS3 cable from Digilent	• For Windows operating systems: Xilinx Vivado executable directory must be on system path. • For Linux operating systems: Digilent Adept 2. For the installation steps, see “Install Digilent Adept 2 Runtime” (HDL Verifier).
	FTDI USB-JTAG cable • Supported for boards with onboard FT4232H, FT232H, or FT2232H devices implementing USB-to JTAG	Install these D2XX drivers. • For Windows operating systems: 2.12.36.4 (64 bit) • For Linux operating systems: 1.4.27 (64 bit) For the installation guide, see D2XX Drivers from the FTDI Chip website.
Microchip	JTAG connection not supported	

Create Custom FPGA Board Definition

1. Be ready with the following:

   a. Board specification document. Any format you are comfortable with is fine. However, if you have it in an electronic version, you can search for the information as it is required.

   b. If you plan to validate (test) your board definition file, set up FPGA design software tools:

      For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

2. Open the FPGA Board Manager by typing `fpgaBoardManager` in the MATLAB command window. Alternatively, if you are using the HDL Workflow Advisor, you can click **Launch Board Manager** at Step 1.1.
3. Open the New FPGA Board wizard by clicking **Create Custom Board**. For a description of all the tasks you can perform with the FPGA Board Manager, see “FPGA Board Manager” on page 35-18.
4. The wizard guides you through entering all board information. At each page, fill in the required fields. For assistance in entering board information, see “New FPGA Board Wizard” on page 35-22.
5. Save the board definition file. This step is the last and is automatically instigated when you click **Finish** in the New FPGA Board wizard. See “Save Board Definition File” on page 35-13.

Your custom board definition now appears in the list of available FPGA Boards in the FPGA Board Manager. If you are using HDL Workflow Advisor, it also shows in the **Target platform** list.

Follow the example “Create Xilinx KC705 Evaluation Board Definition File” on page 35-7 for a demonstration of adding a custom FPGA board with the New FPGA Board Manager.

Create Xilinx KC705 Evaluation Board Definition File

In this section...

“Overview” on page 35-7
“What You Need to Know Before Starting” on page 35-7
“Start New FPGA Board Wizard” on page 35-7
“Provide Basic Board Information” on page 35-8
“Specify FPGA Interface Information” on page 35-9
“Enter FPGA Pin Numbers” on page 35-10
“Run Optional Validation Tests” on page 35-12
“Save Board Definition File” on page 35-13
“Use New FPGA Board” on page 35-14

Overview

For FPGA-in-the-loop, you can use your own qualified FPGA board, even if it is not in the pre-registered FPGA board list supplied by MathWorks. Using the New FPGA Board wizard, you can create a board definition file that describes your custom FPGA board.

In this example, you can follow the workflow of creating a board definition file for the Xilinx KC705 evaluation board to use with FIL simulation.

What You Need to Know Before Starting

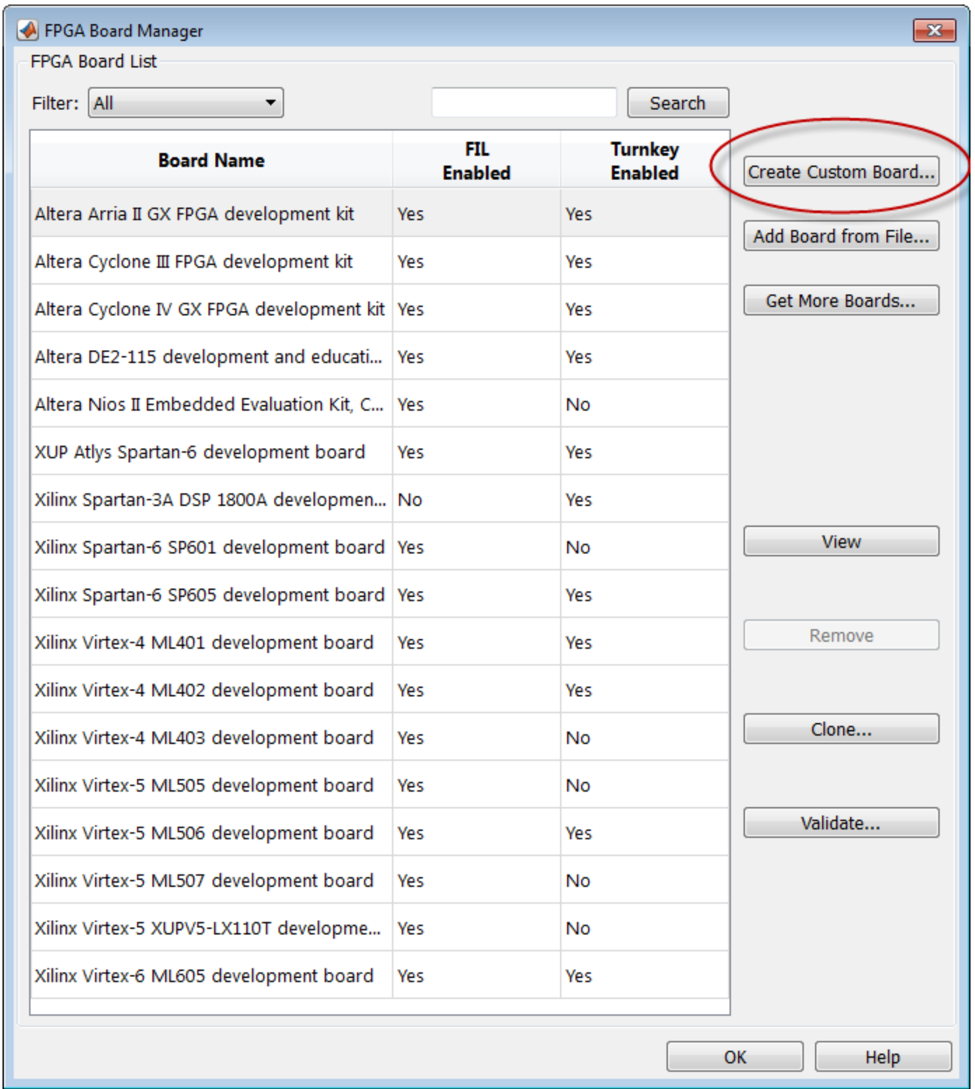
- Check the board specification so that you have the following information ready:
  - FPGA interface to the Ethernet PHY chip
  - Clock pins names and numbers
  - Reset pins names and numbers

  In this example, the required information is supplied to you. In general, you can find this type of information in the board specification file. This example uses the KC705 Evaluation Board for the Kintex-7 FPGA User Guide, published by Xilinx.
- For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.
- To verify programming the FPGA board after you add its definition file, attach the custom board to your computer. However, having the board connected is not necessary for creating the board definition file.

Start New FPGA Board Wizard

1. Start the FPGA Board Manager by entering the following command at the MATLAB prompt:

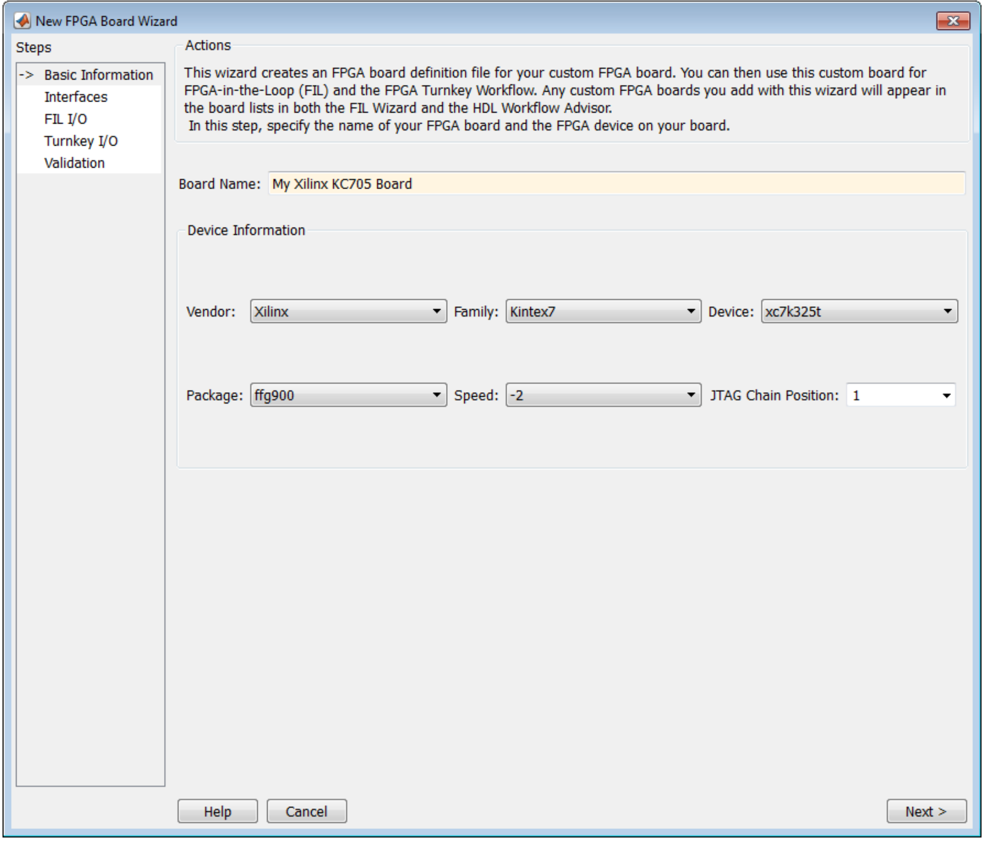
   ```
   >>fpgaBoardManager
   ```
2. Click **Create Custom Board** to open the New FPGA Board wizard.



Provide Basic Board Information

1 In the Basic Information pane, enter the following information:

- **Board Name:** Enter "My Xilinx KC705 Board"
- **Vendor:** Select `Xilinx`
- **Family:** Select `Kintex7`
- **Device:** Select `xc7k325t`
- **Package:** Select `ffg900`
- **Speed:** Select `-2`
- **JTAG Chain Position:** Select `1`



The information you just entered can be found in the KC705 Evaluation Board for the Kintex-7 FPGA User Guide.

2 Click **Next**.

Specify FPGA Interface Information

1 In the Interfaces pane, perform the following tasks.

a Select **FIL Interface**. This option is required for using your board with FPGA-in-the-loop.

b Select **GMII** in the PHY Interface Type. This option indicates that the onboard FPGA is connected to the Ethernet PHY chip via a GMII interface.

c Leave the **User-defined I/O** option in the FPGA Turnkey Interface section cleared. FPGA Turnkey workflow is not the focus of this example.

d **Clock Frequency:** Enter 200. This Xilinx KC705 board has multiple clock sources. The 200 MHz clock is one of the recommended clock frequencies for use with Ethernet interface (50, 100, 125, and 200 MHz).

e **Clock Type:** Select `Differential`.

f **Clock_P Pin Number:** Enter AD12.

g **Clock_N Pin Number:** Enter AD11.

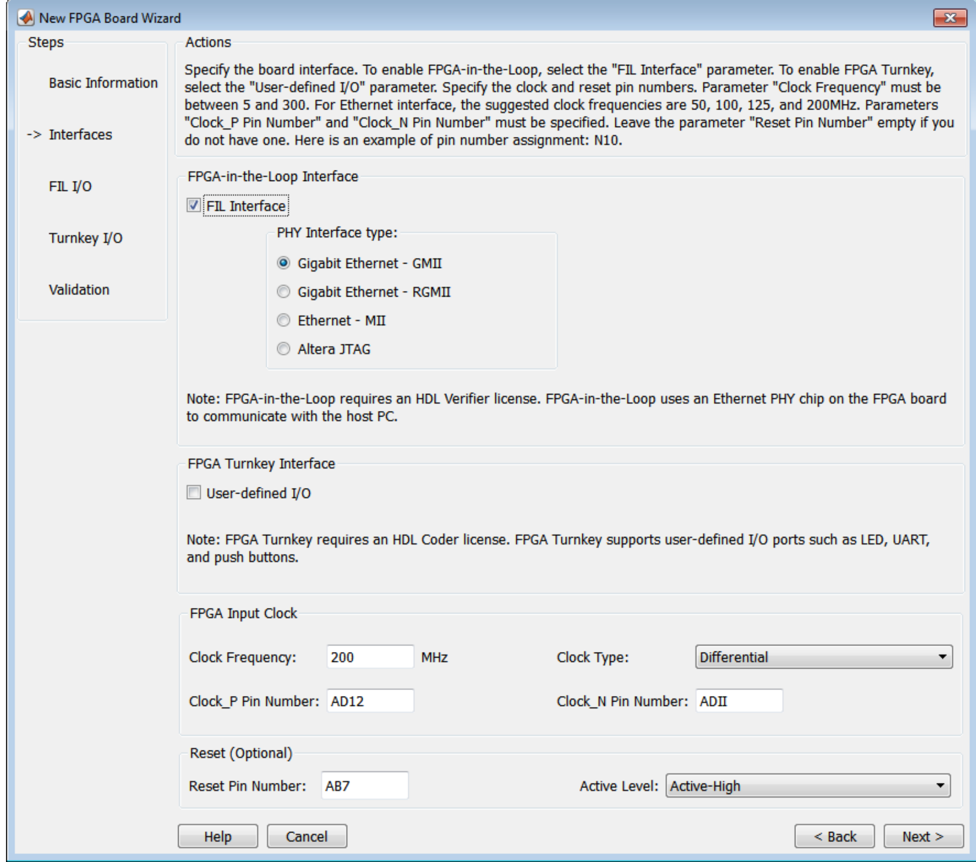
h **Clock IO Standard** — Leave blank.

i **Reset Pin Number:** Enter AB7. This value supplies a global reset to the FPGA.

j **Active Level:** Select `Active-High`.

k **Reset IO Standard** — Leave blank.

You can obtain all necessary information from the board design specification.



2 Click **Next**.

Enter FPGA Pin Numbers

1 In the FILI/O pane, enter the numbers for each FPGA pin. This information is required.

Pin numbers for RXD and TXD signals are entered from the least significant digit (LSD) to the most significant digit (MSB), separated by a comma.

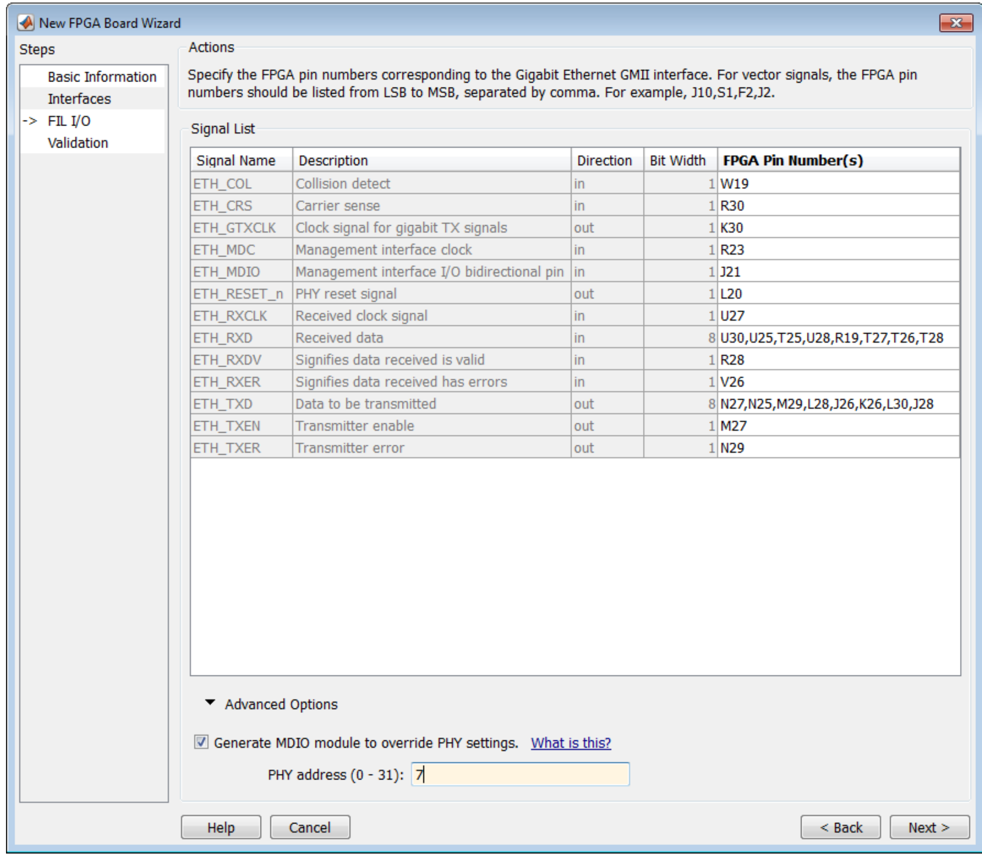
For signal name...	Enter FPGA pin number...
ETH_COL	W19
ETH_CRS	R30
ETH_GTXCLK	K30
ETH_MDC	R23
ETH_MDIO	J21
ETH_RESET_n	L20

For signal name...	Enter FPGA pin number...
ETH_RXCLK	U27
ETH_RXD	U30,U25,T25,U28,R19,T27,T26,T28
ETH_RXDV	R28
ETH_RXER	V26
ETH_TXD	N27,N25,M29,L28,J26,K26,L30,J28
ETH_TXEN	M27
ETH_TXER	N29

2. Click Advanced Options to expand the section.
3. Check the **Generate MDIO module to override PHY settings** option.

   This option is selected for the following reasons:

   - There are jumpers on the Xilinx KC705 board that configure the Ethernet PHY device to MII, GMII, RGMII, or SGMII mode. Since this example uses the GMII interfaces, the FPGA board does not work if the PHY devices are set to the wrong mode. When the **Generate MDIO module to override PHY settings** option is selected, the FPGA uses the Management Data Input/Output (MDIO) bus to override the jumper settings and configure the PHY chip to the correct GMII mode.
   - This option currently only applies to Marvell Alaska PHY device 88E1111 and this KC705 board is using the Marvel device.
4. **PHY address (0 – 31):** Enter 7.



5 Click **Next**.

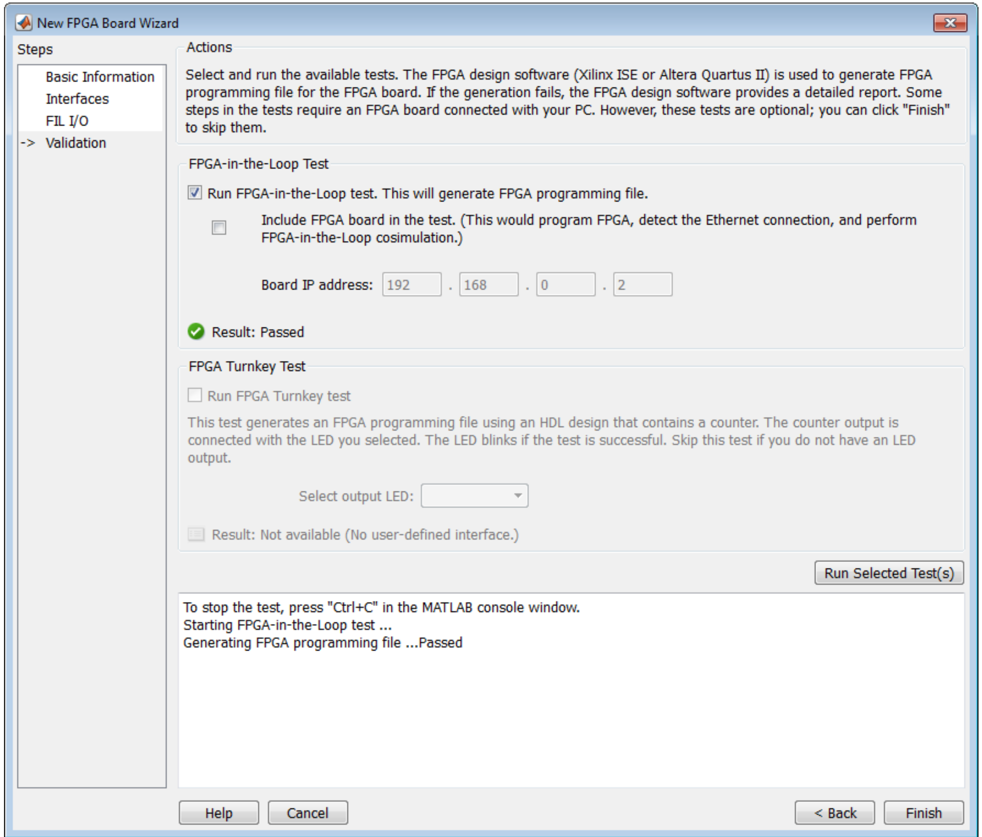
Run Optional Validation Tests

This step provides a validation test for you to verify if the entered information is correct by performing FPGA-in-the-loop cosimulation. You need Xilinx ISE 13.4 or higher versions installed on the same computer. This step is optional and you can skip it, if you prefer.

Note For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

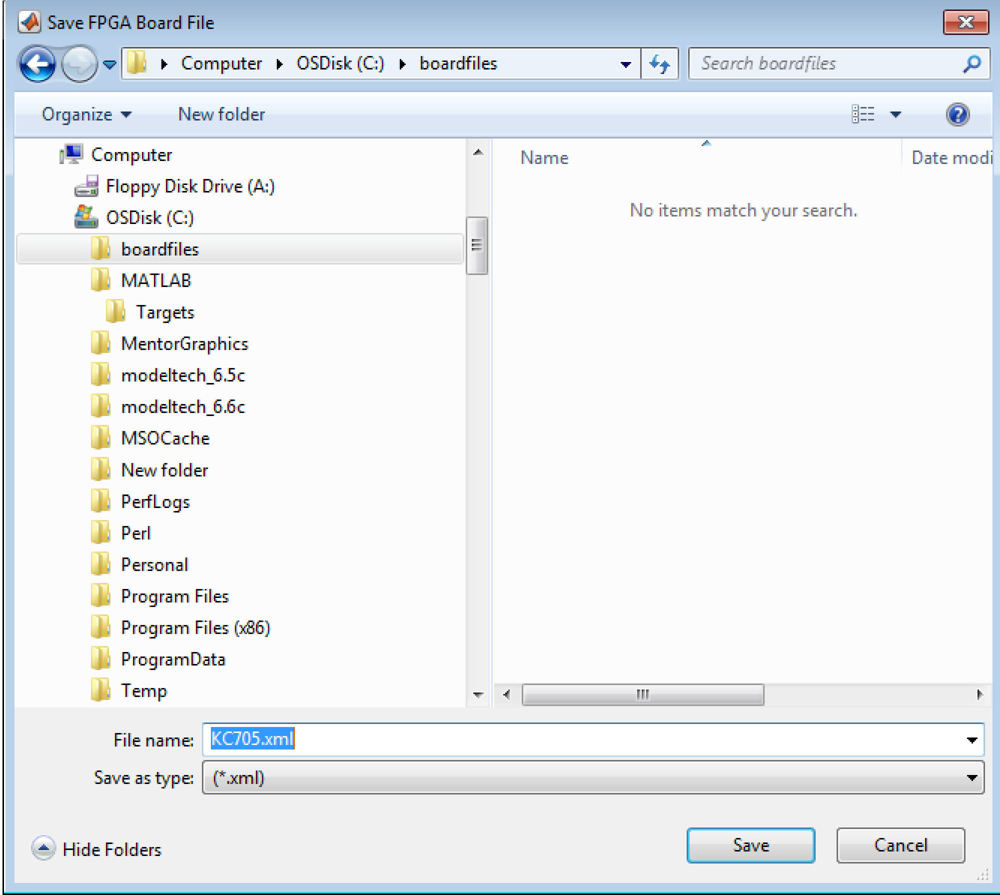
To run this test, perform the following actions.

1. Check the **Run FPGA-in-the-Loop test** option.
2. If you have the board attached, check the **Include FPGA board in the test** option. You need to supply the IP address of the FPGA Board. This example assumes that the Xilinx KC705 board is attached to your host computer and it has an IP address of 192.168.0.2.
3. Click **Run Selected Test(s)**. The tests take about 10 minutes to complete.



Save Board Definition File

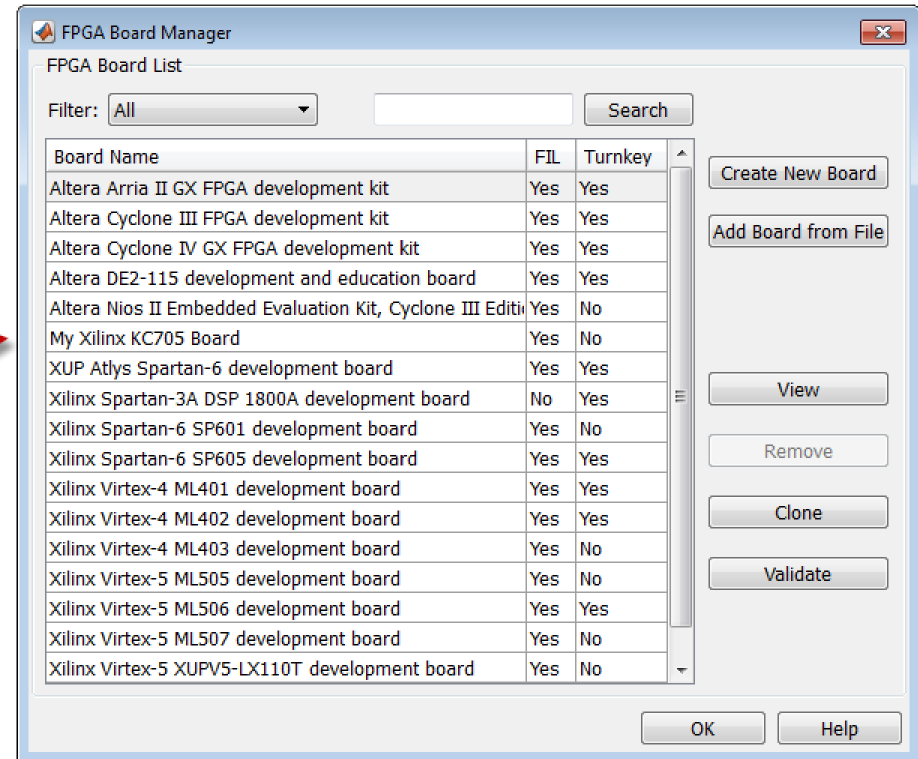
1 Click **Finish** to exit the New FPGA Board wizard. A **Save As** dialog box pops up and asks for the location of the FPGA board definition file. For this example, save as `C:\boardfiles\KC705.xml`.



2 Click **Save** to save the file and exit.

Use New FPGA Board

1 After you save the board definition file, you are returned to the FPGA Board Manager. In the FPGA Board List, you can now see the new board you defined.



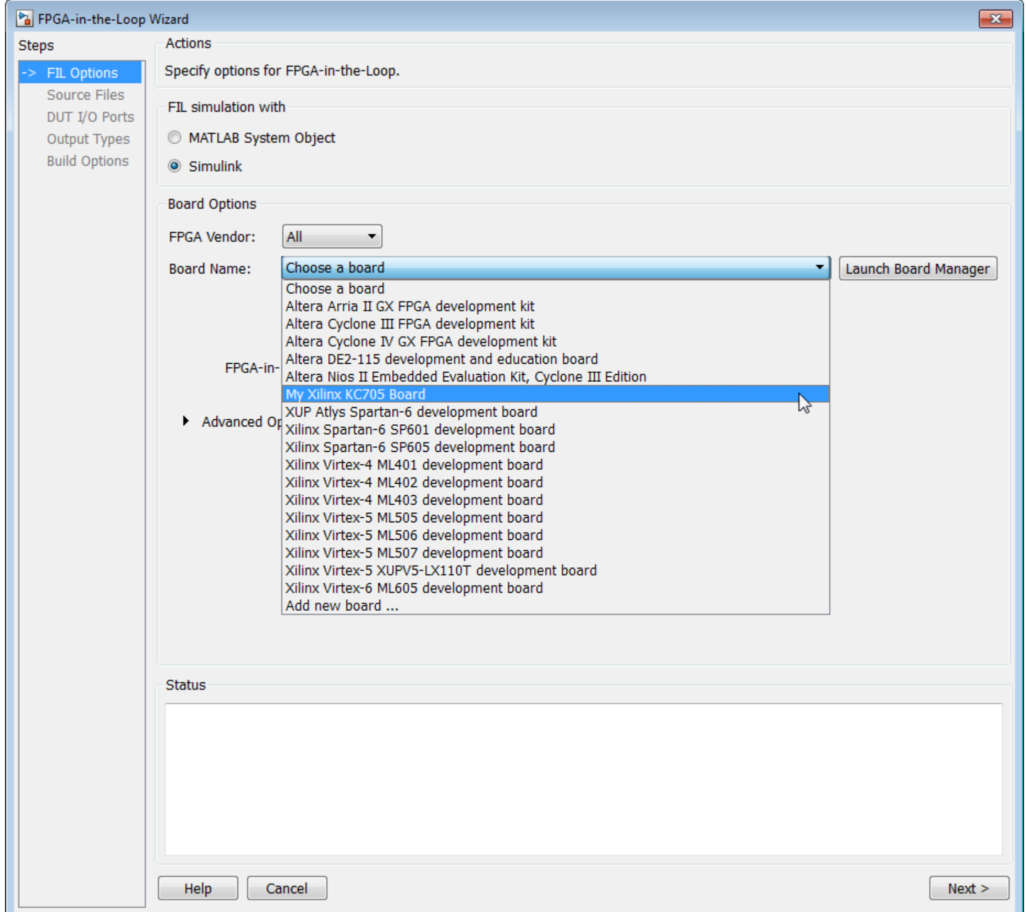
Click **OK** to close the FPGA Board Manager.

2 You can view the new board in the board list from either the FIL wizard or the HDL Workflow Advisor.

a Start the FIL wizard from the MATLAB prompt.

```
>>filWizard
```

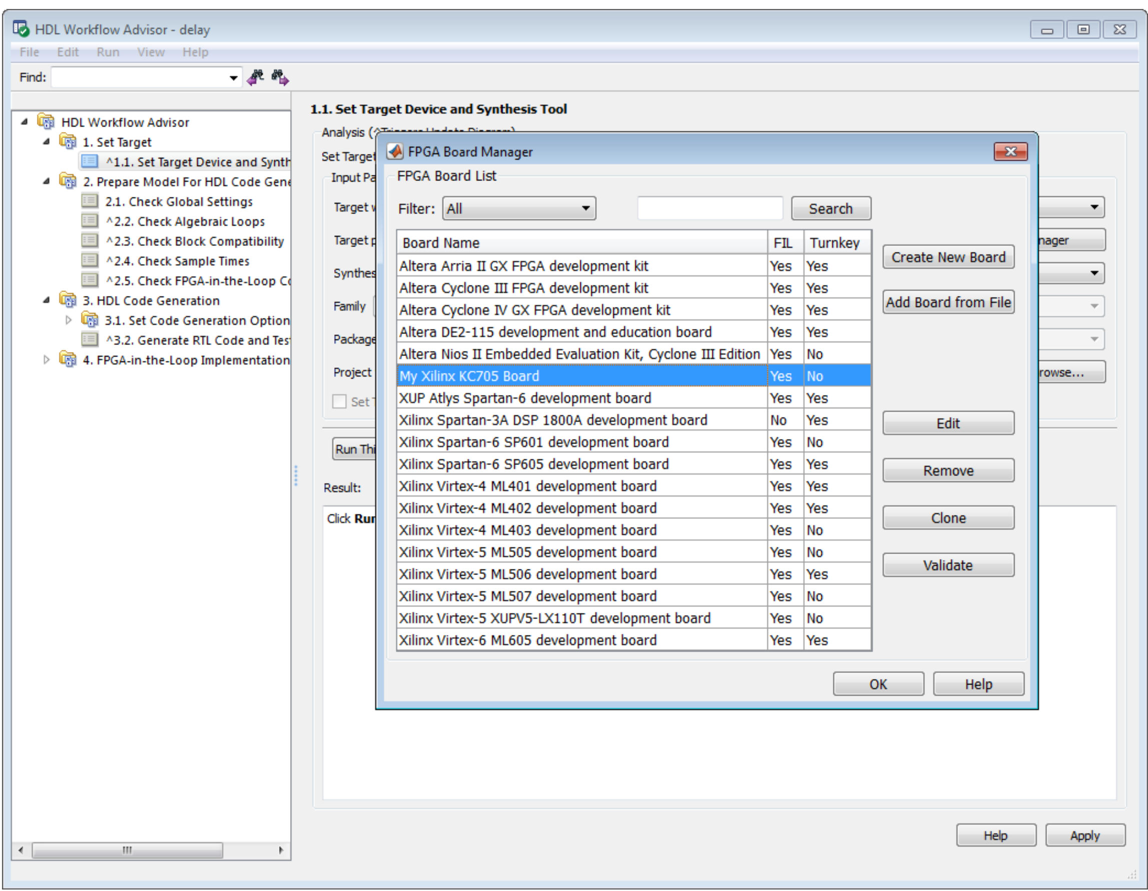
The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.



b Start HDL Workflow Advisor.

In step 1.1, select `FPGA-in-the-Loop` and click **Launch Board Manager**.

The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.


HDL Workflow Advisor - delay
File Edit Run View Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synth
2. Prepare Model For HDL Code Gene
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
^2.5. Check FPGA-in-the-Loop C
3. HDL Code Generation
3.1. Set Code Generation Option
^3.2. Generate RTL Code and Tes
4. FPGA-in-the-Loop Implementation
1.1. Set Target Device and Synthesis Tool
FPGA Board Manager
FPGA Board List
Filter: All
Search
Board Name
FIL
Turnkey
Altera Arria II GX FPGA development kit
Altera Cyclone III FPGA development kit
Altera Cyclone IV GX FPGA development kit
Altera DE2-115 development and education board
Altera Nios II Embedded Evaluation Kit, Cyclone III Edition
My Xilinx KC705 Board
XUP Atlys Spartan-6 development board
Xilinx Spartan-3A DSP 1800A development board
Xilinx Spartan-6 SP601 development board
Xilinx Spartan-6 SP605 development board
Xilinx Virtex-4 ML401 development board
Xilinx Virtex-4 ML402 development board
Xilinx Virtex-4 ML403 development board
Xilinx Virtex-5 ML505 development board
Xilinx Virtex-5 ML506 development board
Xilinx Virtex-5 ML507 development board
Xilinx Virtex-5 XUPV5-LX110T development board
Xilinx Virtex-6 ML605 development board
Create New Board
Add Board from File
Edit
Remove
Clone
Validate
OK
Help
Result:
Help
Apply

FPGA Board Manager

In this section...

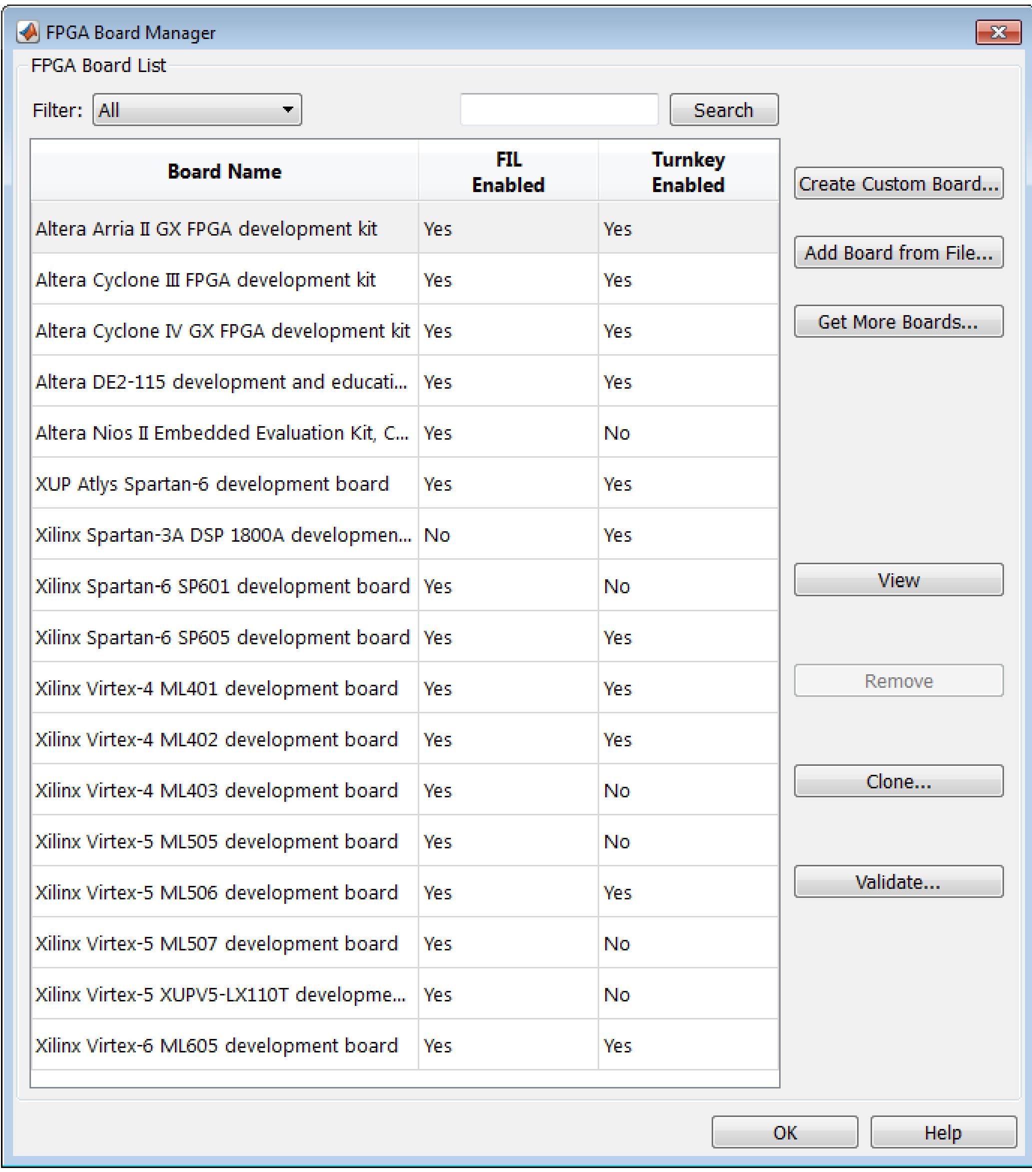
"Introduction" on page 35-18
"Filter" on page 35-19
"Search" on page 35-19
"FIL Enabled/Turnkey Enabled" on page 35-20
"Create Custom Board" on page 35-20
"Add Board from File" on page 35-20
"Get More Boards" on page 35-20
"View/Edit" on page 35-20
"Remove" on page 35-20
"Clone" on page 35-20
"Validate" on page 35-20

Introduction

The FPGA Board Manager is the portal to managing custom FPGA boards. You can create a board definition file or edit an existing one. You can even import a custom board from an existing board definition file.

You start the FPGA Board Manager by one of the following methods:

- By typing `fpgaBoardManager` in the MATLAB command window
- From the FIL wizard by clicking **Launch Board Manager** on the first page
- From the HDL Workflow Advisor (when using HDL Coder) at Step 1.1



Filter

Choose one of the following views:

- All boards
- Only those boards that were preinstalled with HDL Verifier or HDL Coder
- Only custom boards

Search

Find a specific board in the list or those boards that fully or partially match your search string.

FIL Enabled/Turnkey Enabled

These columns indicate whether the specified board is supported for FIL or Turnkey operations.

Create Custom Board

Start New FPGA Board wizard. See “New FPGA Board Wizard” on page 35-22. You can find the process for creating a board definition file in “Create Custom FPGA Board Definition” on page 35-6.

Add Board from File

Import a board definition file (.xml).

Get More Boards

Download FPGA board support packages for use with FIL

1. Click **Get more boards**.
2. Follow the prompts in the Support Package Installer to download an FPGA board support package.
3. When the download is complete, you can see the new boards in the board list in the FPGA Board Manager.

Offline Support Package Installation You can install an FPGA board support package without an internet connection. See “Install Support Package Offline” (HDL Verifier).

View/Edit

View board configurations and modify the information. You can view a read-only file but not edit it. See “FPGA Board Editor” on page 35-33.

Remove

Remove custom board from the list. This action does not delete the board definition XML file.

Clone

Makes a copy of an existing custom board for further modification.

Validate

Runs the validation tests for FIL See “Run Optional Validation Tests” on page 35-12.

See Also

Related Examples

- “Create Custom FPGA Board Definition” (HDL Verifier)
- “Create Xilinx KC705 Evaluation Board Definition File” (HDL Verifier)

New FPGA Board Wizard

Using the New FPGA Board wizard, you can enter all the required information to add a board to the FPGA board list. This list applies to both FIL and Turnkey workflows. Review “FPGA Board Requirements” on page 35-2 before adding an FPGA board to make sure that it is compatible with the workflow for which you want to use it.

Several buttons in the New FPGA Board wizard help with navigation:

- **Back**: Go to a previous page to review or edit data already entered.
- **Next**: Go to next page when all requirements of current page have been satisfied.
- **Help**: Open Doc Center, and display this topic.
- **Cancel**: Exit New FPGA Board wizard. You can exit with or without saving the information from your session.

Adding Boards Once for Multiple Users To add new boards globally, follow these instructions. To access a board added globally, all users must be using the same MATLAB installation.

1. Create the following folder:

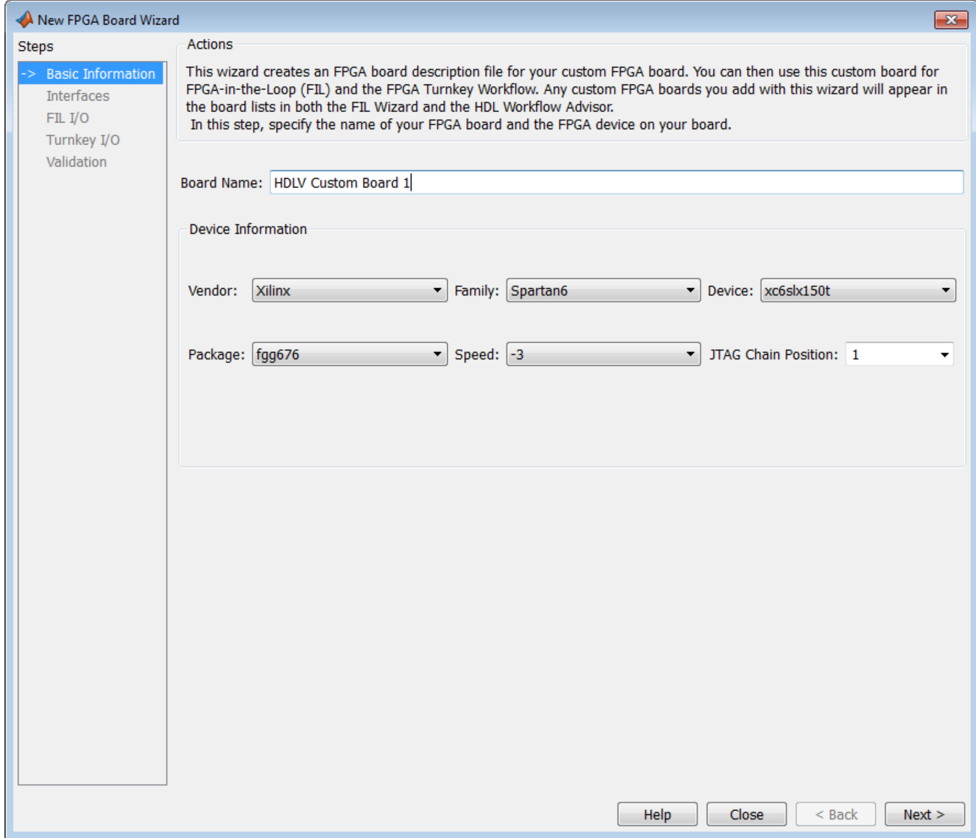
   `*matlabroot*/toolbox/shared/eda/board/boardfiles`
2. Copy the board description XML file to the `boardfiles` folder.
3. After copying the XML file, restart MATLAB. The new board appears in the FPGA board list for either or both the FIL and Turnkey workflows.

All boards under this folder show-up in the FPGA board list automatically for users with the same MATLAB installation. You do not need to use FPGA Board Manager to add these boards again.

The workflow for adding an FPGA board contains these steps:

In this section...
“Basic Information” on page 35-23
“Interfaces” on page 35-23
“FIL I/O” on page 35-26
“Turnkey I/O” on page 35-28
“Validation” on page 35-31
“Finish” on page 35-32

Basic Information



Board Name: Enter a unique board name.

Device Information:

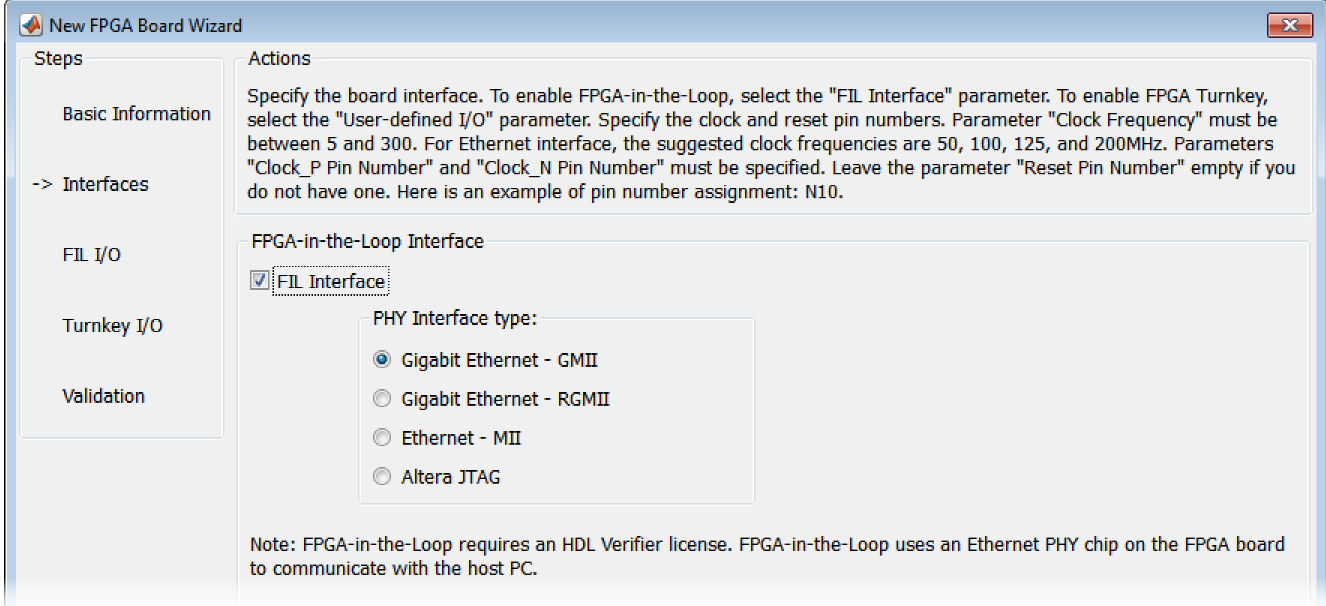
- **Vendor**: `Xilinx` or `Altera`
- **Family**: Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device**: Use the board specification file to select the correct device.
- For Xilinx boards only:
  - **Package**: Use the board specification file to select the correct package.
  - **Speed**: Use the board specification file to select the correct speed.
  - **JTAG Chain Position**: Value indicates the starting position for JTAG chain. Consult the board specification file for this information.

Interfaces

- "FIL Interface for Altera Boards" on page 35-24
- "FIL Interface for Xilinx Boards" on page 35-25
- "FPGA Turnkey Interface" on page 35-25

- “FPGA Input Clock and Reset” on page 35-26

FIL Interface for Altera Boards

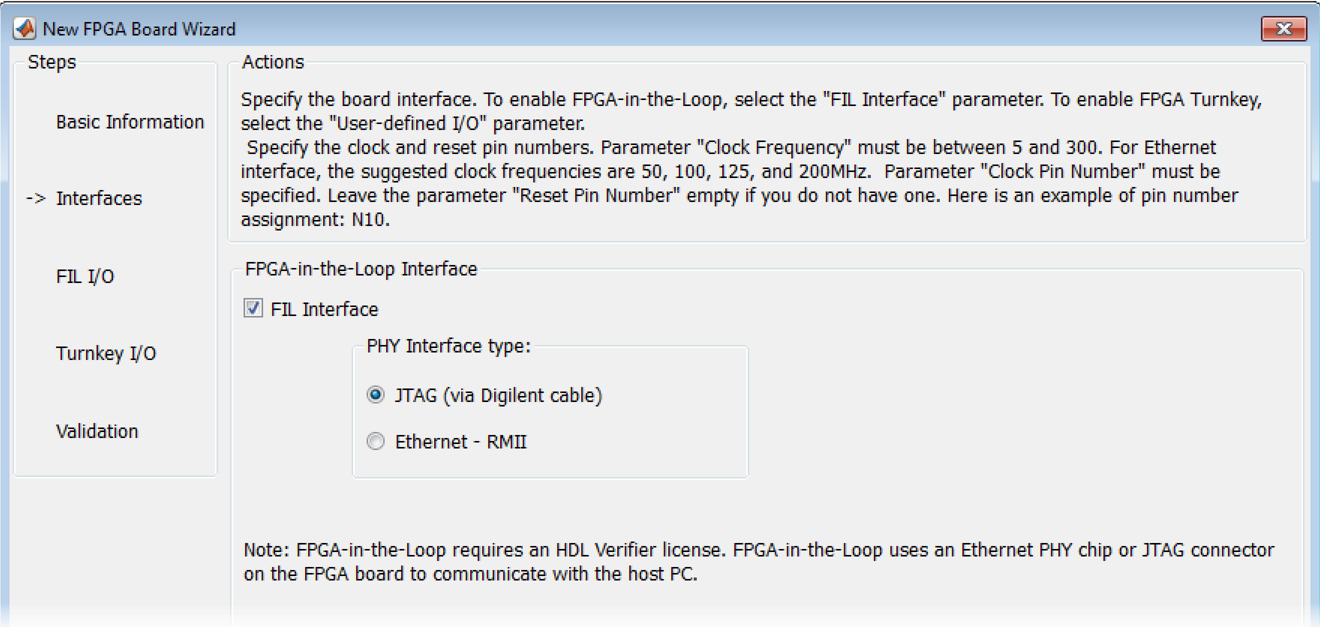


1. **FPGA-in-the-Loop**: To use this board with FIL, select **FIL Interface**.
2. Select one of the following **PHY Interface type**s:

   - **Gigabit Ethernet — GMII**
   - **Gigabit Ethernet — RGMII**
   - **Gigabit Ethernet — SGMII** (the SGMII option appears if you select a board from the Stratix V or Stratix IV device families)
   - **Ethernet — MII**
   - **Altera JTAG** (Altera boards only)

Note Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

FIL Interface for Xilinx Boards



1 **FPGA-in-the-Loop Interface**: To use this board with FIL, select **FIL Interface**.
2 Select one of the following **PHY Interface type**s:

- **JTAG (via Digilent cable)** (Xilinx boards only)
- **Ethernet — RMII**

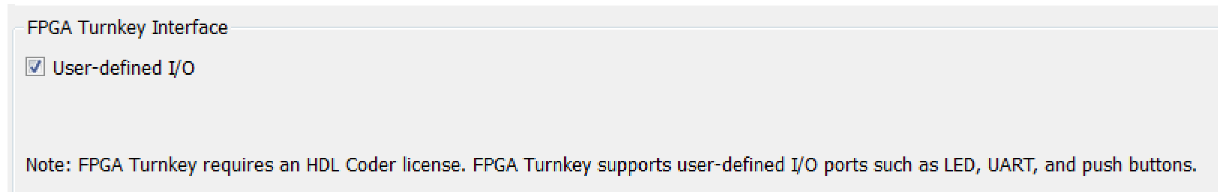
Note Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

For more information on how to set up the JTAG connection for Xilinx boards, see “JTAG with Digilent Cable Setup” on page 35-35.

Limitations

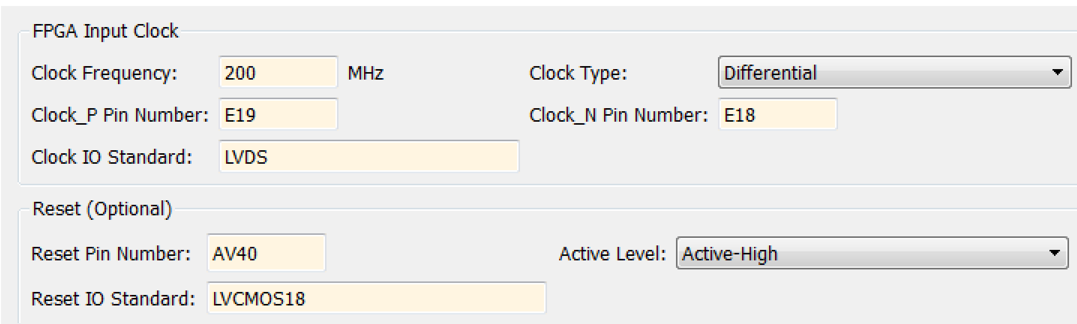
When you simulate your FPGA design through a Digilent JTAG cable, you cannot use any other debugging feature that requires access to the JTAG; for example, the Vivado Logic Analyzer.

FPGA Turnkey Interface



FPGA Turnkey Interface: If you want to use with board with the HDL Coder FPGA Turnkey workflow, select **User-defined I/O**.

FPGA Input Clock and Reset



1 **FPGA Input Clock** — Clock details are required for both workflows. You can find all necessary information in the board specification file.

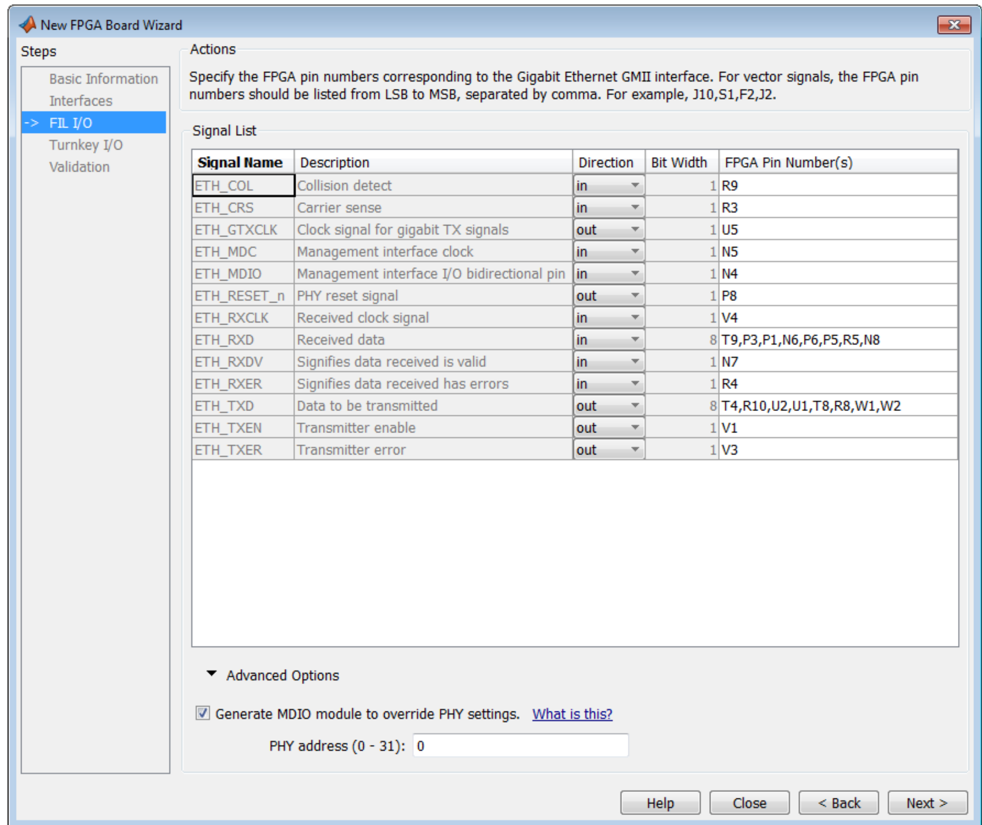
- **Clock Frequency** — Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
- **Clock Type** — `Single_Ended` or `Differential`.
- **Clock Pin Number** (Single_Ended) — Must be specified. Example: `N10`.
- **Clock_P Pin Number** (Differential) — Must be specified. Example: `E19`.
- **Clock_N Pin Number** (Differential) — Must be specified. Example: `E18`.
- **Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVDS`.

2 **Reset (Optional)** — If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.

- **Reset Pin Number** — Leave empty if you do not have one.
- **Active Level** — `Active-Low` or `Active-High`.
- **Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVCMOS33`.

FIL I/O

When you select an Ethernet connection to your board, you must specify pins for the Ethernet signals on the FPGA.



Signal List: Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas.

Note If your PHY chip does not have the optional TX_ER pin, tie ETH_TXER to one of the unused pins on the FPGA.

Generate MDIO module to override PHY settings: See the next section on FPGA Board Management Data Input/Output Bus (MDIO) to determine when to use this feature. If you do select this option, enter the PHY address.

What Is the Management Data Input/Output Bus?

Management Data Input/Output (MDIO) is a serial bus, defined in the IEEE 802.3 standard, that connects MAC devices and Ethernet PHY devices. The FPGA MAC uses the MDIO bus to set control registers in the Ethernet PHY device on the board.

Currently only the Marvell 88E1111 PHY chip is supported by this MDIO module implementation. Do not select this check box if you are not using Marvell 88E1111.

The generated MDIO module is used to perform the following operations:

- **GMII mode**: The PHY device can start up using other modes, such as RGMII/SGMII. The generated MDIO module sets the PHY chip in GMII mode.

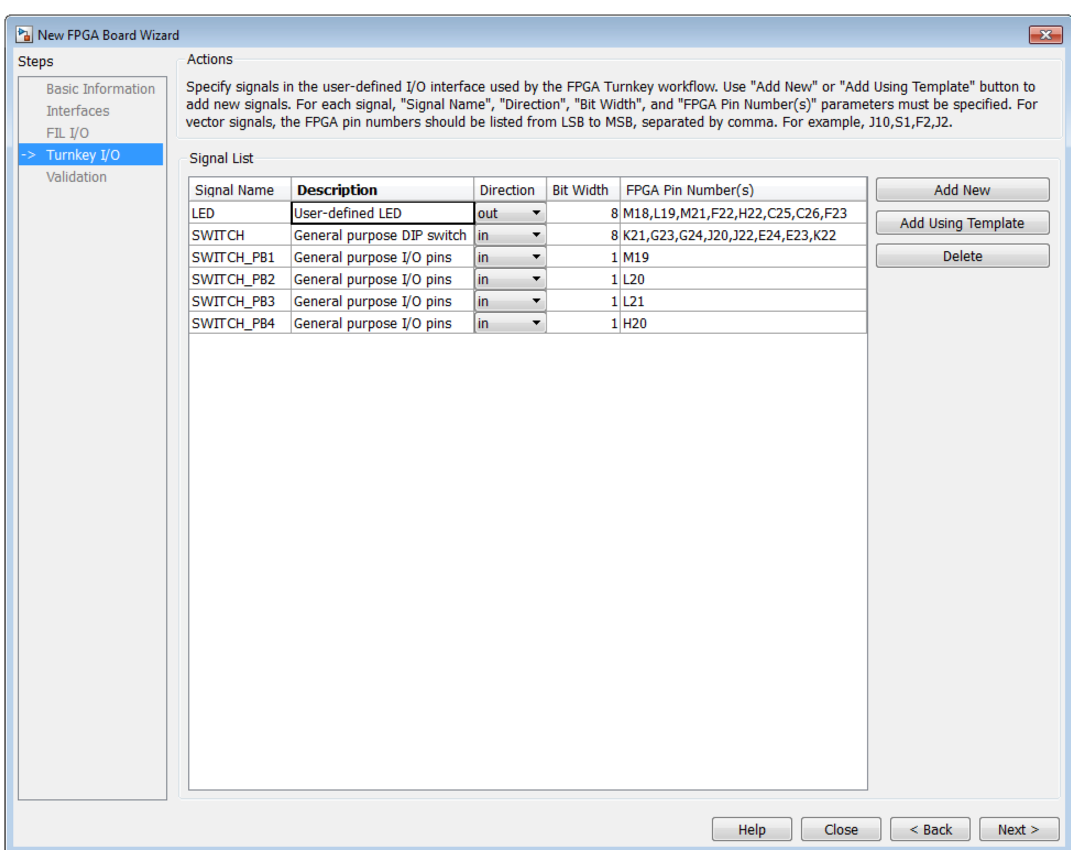
- **RGMII mode**: The PHY device can start up using other modes, such as GMII/SGMII. The generated MDIO module sets the PHY device in RGMII mode. In addition, the module sets the PHY chip to add internal delay for RX and TX clocks.
- **SGMII mode**: The PHY device can start up using other modes, such as RGMII/GMII. The generated MDIO module sets the PHY chip in SGMII mode.
- **MII mode**: The generated MDIO module sets the PHY device in GMII compatible mode. The module also sets the autonegotiation register to remove the 1000 Base-T capability advertisement. This reset ensures that the autonegotiation process does not select 1000 Mbits/s speed, which is not supported in MII mode.

When To Select MDIO: Select the **Generate MDIO module to override PHY settings** option when both the following conditions are met:

- The onboard Ethernet PHY device is Marvell 88E1111.
- The PHY device startup settings are not compatible with the FPGA MAC. The MDIO modules for different PHY modes must override these settings, as previously described.

Specifying the PHY Address: The PHY address is a 5-bit integer. The value is determined by the CONFIG[0] and CONFIG[1] pin on Marvell 88E1111 PHY device. See the board manual for this value.

Turnkey I/O



Note Provide FIL I/O for an Ethernet connection only. Define at least one output port for the Turnkey I/O interface.

Signal List: Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas. The number of pin numbers must match the bit width of the corresponding signal.

Add New: You are prompted to enter all entries in the signal list manually.

Add Using Template: The wizard prepopulates a new signal entry for UART, LED, GPIO, or DIP Switch signals with the following:

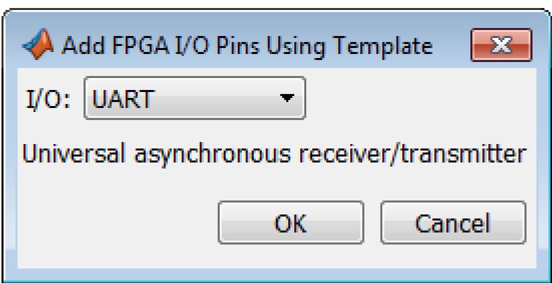
- A generic signal name
- Description
- Direction
- Bit width

You can change the values in any of these prepopulated fields.

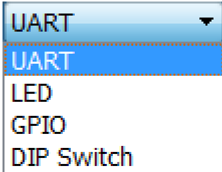
Delete: Delete the selected signal from list.

The following example demonstrates using the **Add Using Template** feature.

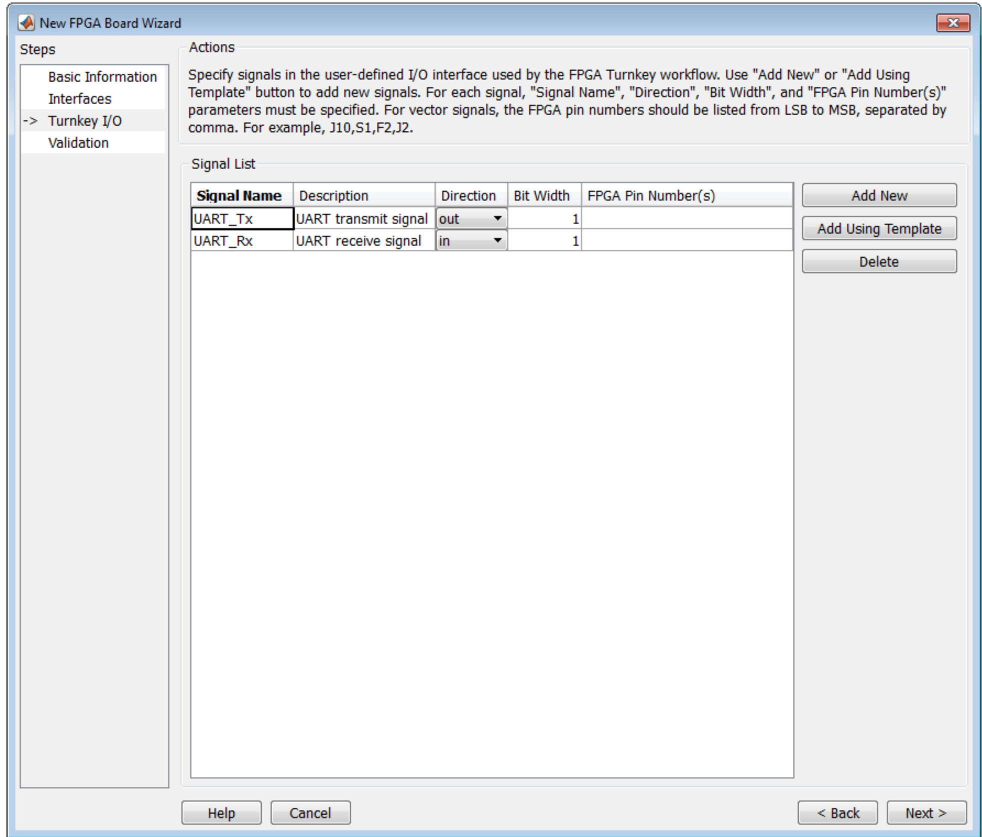
1 In the Turnkey I/O dialog box, click **Add Using Template**.
2 You can now view the template dialog box.



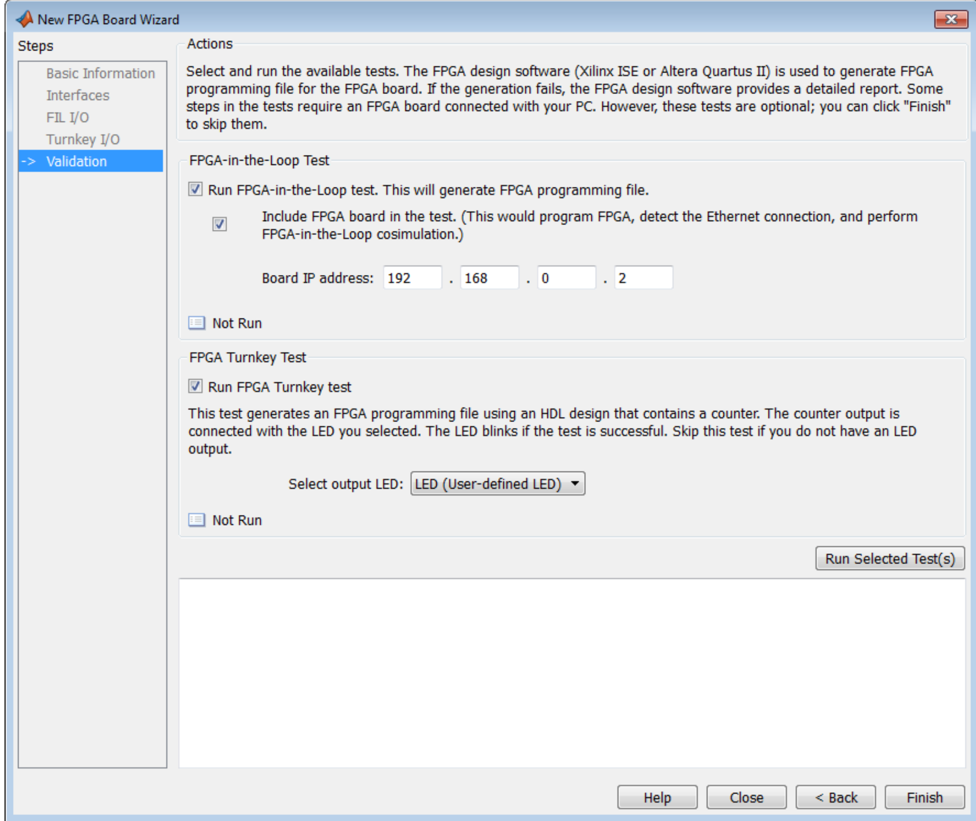
3 Pull down the I/O list and select from the following options:



4 Click **OK**.
5 The wizard adds the specified signal (or signals) to the I/O list.


New FPGA Board Wizard
Steps
Basic Information
Interfaces
-> Turnkey I/O
Validation
Actions
Specify signals in the user-defined I/O interface used by the FPGA Turnkey workflow. Use "Add New" or "Add Using Template" button to add new signals. For each signal, "Signal Name", "Direction", "Bit Width", and "FPGA Pin Number(s)" parameters must be specified. For vector signals, the FPGA pin numbers should be listed from LSB to MSB, separated by comma. For example, J10,S1,F2,J2.
Signal List
Signal Name
Description
Direction
Bit Width
FPGA Pin Number(s)
UART_Tx
UART transmit signal
out
1
UART_Rx
UART receive signal
in
1
Add New
Add Using Template
Delete
Help
Cancel
< Back
Next >

Validation



FPGA-in-the-Loop Test

- **Run FPGA-in-the-Loop test**: Select to generate an FPGA programming file.
  - **Include FPGA board in the test**: (Optional) This selection program the FPGA with the generated programming file, detects the Ethernet connection (if selected), and performs FPGA-in-the-loop simulation.
  - **Board IP address**: (Ethernet connection only) Use this option for setting the board IP address if it is not the default IP address (192.168.0.2).

    If necessary, change the computer IP address to a different subnet from 192.168.0.x when you set up the network adapter. If the default board IP address 192.168.0.2 is in use by another device, change the Board IP address according to the following guidelines:

    - The subnet address, typically the first 3 bytes of board IP address, must be the same as the host IP address.
    - The last byte of the board IP address must be different from the host IP address.
    - The board IP address must not conflict with the IP addresses of other computers.

      For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.

FPGA Turnkey Test

- **Run FPGA Turnkey test**: Select to generate an FPGA programming file using an HDL design that contains a counter. You must have a board attached.
- **Select output LED**: The counter's output is connected with the LED you select. Skip this test if you do not have an LED output.

Finish

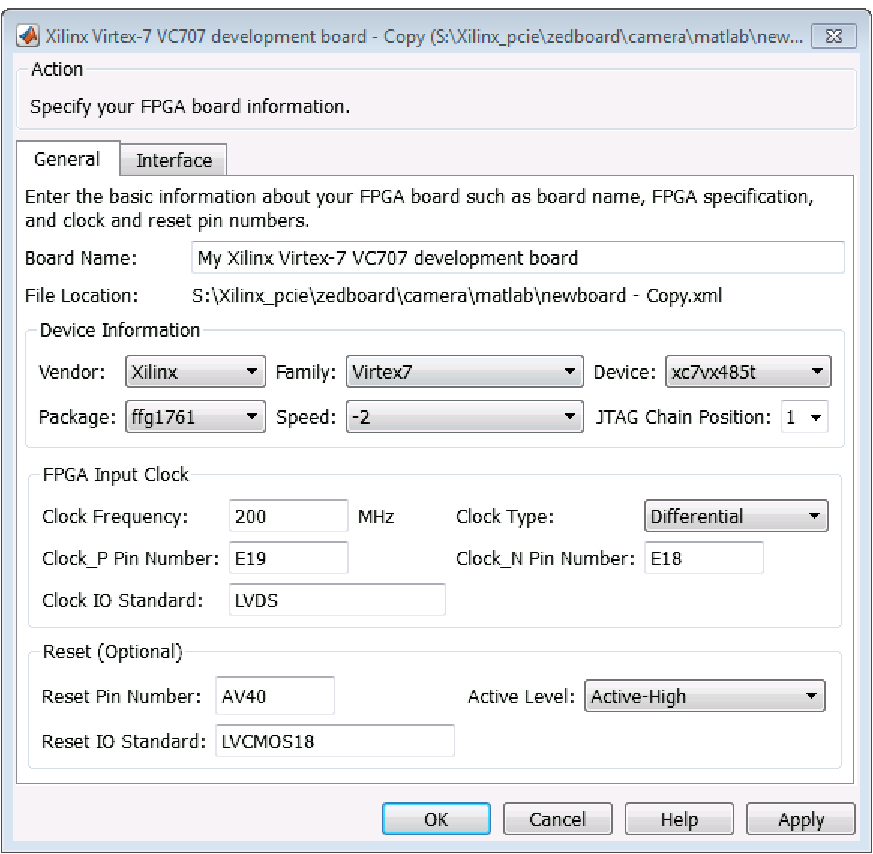
When you have completed validation, click **Finish**. See "Save Board Definition File" on page 35-13.

FPGA Board Editor

In this section...
"General Tab" on page 35-33
"Interface Tab" on page 35-35

To edit a board definition XML file, first make it writeable. If the file is read-only, the FPGA Board Editor only lets you view the board configuration information. You cannot modify that information.

General Tab



Board Name: Unique board name

Device Information:

- **Vendor**: `Xilinx` or `Altera`
- **Family**: Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device**: Device depends on the specified vendor and family. See the board specification file for applicable settings.
- For Xilinx boards only:

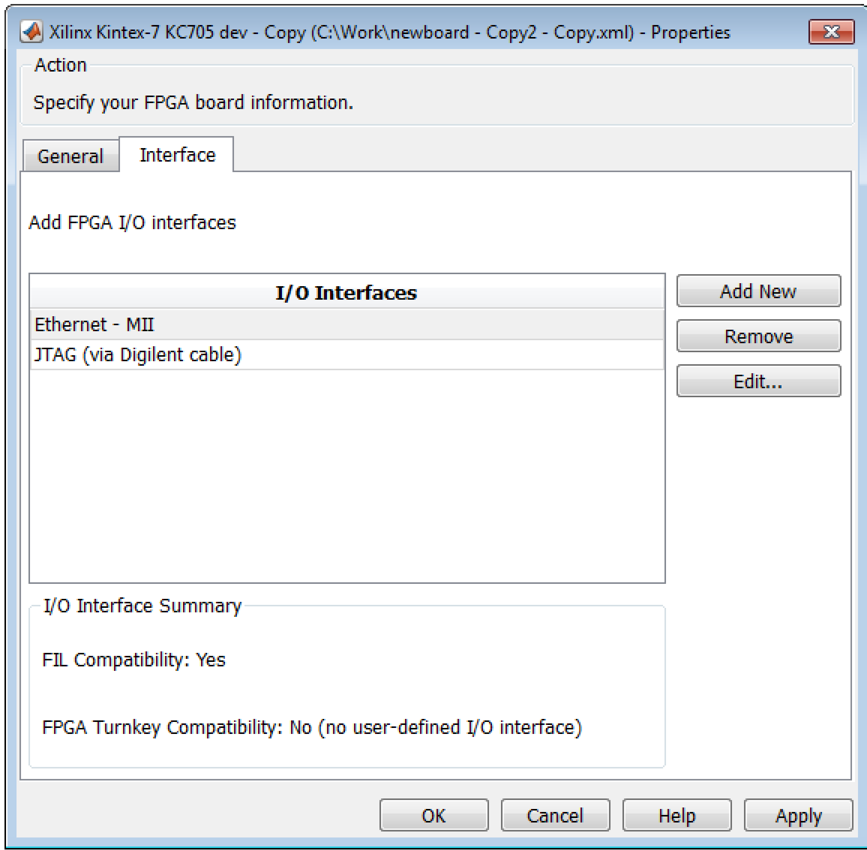
    - **Package**: Package depends on specified vendor, family, and device. See the board specification file for applicable settings.
    - **Speed**: Speed depends on package. See the board specification file for applicable settings.
    - **JTAG Chain Position**: Value indicates the starting position for JTAG chain. Consult the board specification file for this information.

- **FPGA Input Clock**. Clock details are required for both the FIL and Turnkey workflows. You can find all necessary information in the board specification file.

    - **Clock Frequency**. Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
    - **Clock Type**: `Single_Ended` or `Differential`.
    - **Clock Pin Number** (Single_Ended) — Must be specified. Example: `N10`.
    - **Clock_P Pin Number** (Differential) — Must be specified. Example: `E19`.
    - **Clock_N Pin Number** (Differential) — Must be specified. Example: `E18`.
    - **Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVDS`.
- **Reset (Optional)**. If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.

    - **Reset Pin Number**. Leave empty if you do not have one.
    - **Active Level** : `Active-Low` or `Active-High`.
    - **Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVCMOS33`.

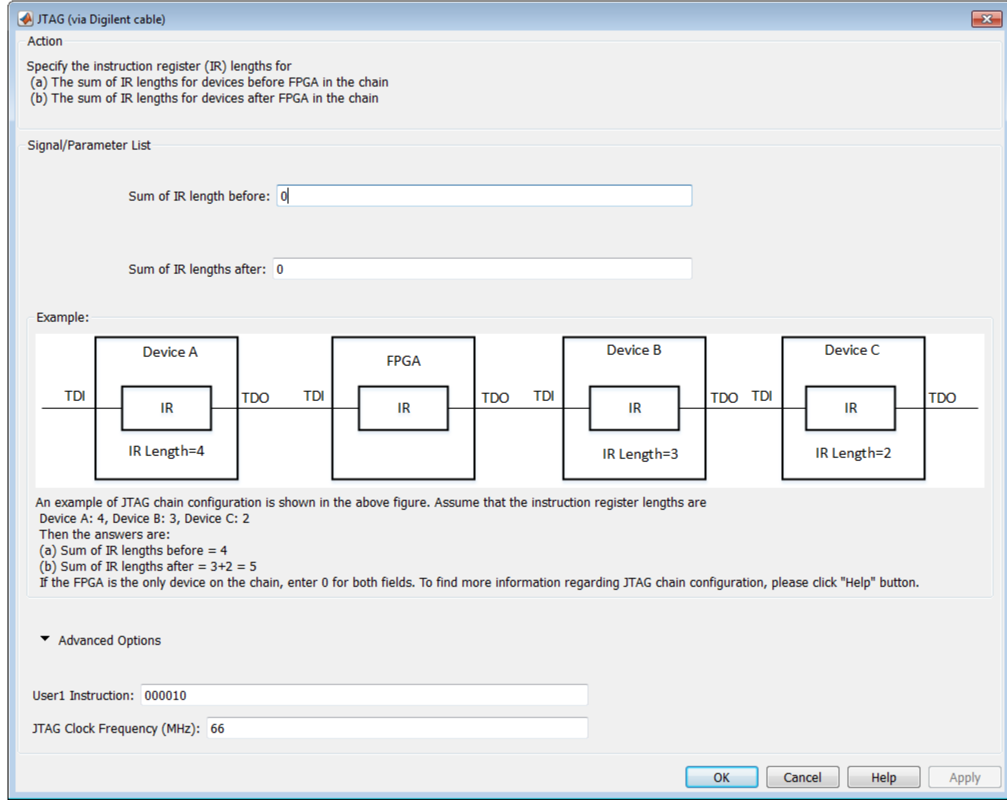
Interface Tab



The Interface page describes the supported FPGA I/O Interfaces. Select any listed interface and click **View** to see the **Signal List**. If the board definition file has write permission, you can also **Add New** interface, **Edit** the interface, or **Remove** an interface.

JTAG with Digilent Cable Setup

Note Enter information for the JTAG cable setup carefully. If the settings are incorrect, the simulation errors out and does not work. If you are still unsure about how to setup your JTAG cable after reading these instructions, contact MathWorks technical support with detailed information about your board.



1 **Signal/Parameter List** — Provide the sum of the lengths of the instruction registers (IR) for all devices before and after the FPGA in the chain.

- If the FPGA is the only item in the device chain, use zeros in both **Sum of IR length before** and **Sum of IR length after**.
- If you are using a Zynq device, and it is the only item in the device chain, enter `4` in **Sum of IR length before** and `0` in **Sum of IR length after**.

If your board does not meet either of those conditions, follow these instructions to obtain the IR lengths:

a. Connect the FPGA board to your computer using the JTAG cable. Turn on the board.
b. Make sure that you installed the cable drivers during Vivado installation.
c. Open Vivado Hardware Manager and select **Open a new hardware target**. In the dialog box is a summary of the IR lengths for all devices for that target.
d. Sum the IR lengths before the FPGA and enter the total in **Sum of IR length before**. Sum the IR lengths after the FPGA and enter the total in **Sum of IR length after**.

Vivado Hardware Manager cannot recognize the IR length of less common devices. For these devices, consult the device manual for instruction register length.

2 **Advanced Options** — If the default values are not the same as the most common settings for many devices, set the **User1 Instruction** and **JTAG Clock Frequency (MHz)** parameters. The most common settings are `000010` and `66`, respectively.

- **User1 Instruction** — The JTAG USER1 Instruction defined in the Xilinx Bscane2 primitive. This binary instruction number, defined by Xilinx, varies from device to device. For most of the

7-series devices, this instruction is `000010`. If your device has a different value, enter it in this parameter.

To find this value, look at the `bsd` file for your specific device, found in your Vivado installation. For example, for the XA7A32T-CPG236 device, the `bsd` file is located in `Vivado\2020.2\data\parts\xilinx\artix7\public\bsdl\xc7a35t_cpg236.bsd`.

Open this file. The `USER1` value is `000010`. Enter this value at **User1 Instruction**.

```
"USER1       (000010),"
```

- **JTAG Clock Frequency (MHz)** — Clock frequency used by the JTAG circuit. This value varies by device. You can find this value in the same `bsd` file described under **User1 Instruction**. For example, the JTAG clock frequency is 66 MHz for device XA7A32T-CPG236:

```
attribute TAP_SCAN_CLOCK of TCK : signal is (66.0e6, BOTH);
```

HDL Workflow Advisor Tasks

HDL Workflow Advisor Tasks

In this section...

“HDL Workflow Advisor Tasks Overview” on page 36-3
“Set Target Overview” on page 36-3
“Set Target Device and Synthesis Tool” on page 36-4
“Set Target Reference Design” on page 36-5
“Set Target Interface” on page 36-6
“Set Target Interface” on page 36-7
“Set Target Frequency” on page 36-7
“Prepare Model for HDL Code Generation Overview” on page 36-8
“Check Model Settings” on page 36-9
“Check FPGA-in-the-Loop Compatibility” on page 36-9
“HDL Code Generation Overview” on page 36-9
“Set HDL Options” on page 36-10
“Generate RTL Code and Testbench” on page 36-10
“Verify with HDL Cosimulation” on page 36-11
“Generate RTL Code and IP Core” on page 36-11
“FPGA Synthesis and Analysis Overview” on page 36-13
“Create Project” on page 36-14
“Perform Synthesis and P/R Overview” on page 36-15
“Perform Logic Synthesis” on page 36-15
“Perform Mapping” on page 36-15
“Perform Place and Route” on page 36-16
“Run Synthesis” on page 36-16
“Run Implementation” on page 36-17
“Annotate Model with Synthesis Result” on page 36-17
“Download to Target Overview” on page 36-18
“Generate Programming File” on page 36-18
“Program Target Device” on page 36-18
“Generate Simulink Real-Time Interface” on page 36-19
“Save and Restore HDL Workflow Advisor State” on page 36-19
“FPGA-in-the-Loop (FIL) Implementation” on page 36-19
“Set FPGA-in-the-Loop Options” on page 36-19
“Build FPGA-in-the-Loop” on page 36-20
“Embedded System Integration” on page 36-20
“Create Project” on page 36-20
“Generate Software Interface” on page 36-21
“Build FPGA Bitstream” on page 36-22

In this section...
“Program Target Device” on page 36-22

HDL Workflow Advisor Tasks Overview

The HDL Workflow Advisor is a tool that supports a suite of tasks covering the stages of the FPGA design process. Some tasks perform model validation or checking. Other tasks run the HDL code generator or third-party tools. Each folder at the top level of the HDL Workflow Advisor contains a group of related tasks that you can select and run.

HDL Workflow Advisor is not available in Simulink Online.

For summary information on each HDL Workflow Advisor folder or task, select the folder or task icon, and then click the HDL Workflow Advisor **Help** button.

- **Set Target**: The tasks in this category enable you to select the target device and map its I/O interface to the inputs and outputs of your model.
- **Prepare Model For HDL Code Generation**: The tasks in this category check your model for HDL code generation compatibility. The tasks also report on model settings, blocks, or other conditions (such as algebraic loops) that impede code generation, and provide advice on how to fix such problems.
- **HDL Code Generation**: This category supports all HDL-related options in the Configuration Parameters dialog box, including setting HDL code and test bench generation parameters, and generating code, test bench, or a cosimulation model.
- **FPGA Synthesis and Analysis**: The tasks in this category support:
  - Synthesis and timing analysis through integration with third-party synthesis tools
  - Back annotation of the model with critical path and other information obtained during synthesis
- **FPGA-in-the-Loop Implementation**: This category implements the phases of FIL, including providing block generation, synthesis, logical mapping, PAR (place-and-route), programming file generation, and a communications channel. These capabilities are designed for a particular board and tailored to your register transfer level (RTL) code. HDL Verifier is required for FIL.
- **Download to Target**: The tasks in this category depend on the selected target device and potentially include:
  - Generation of a target-specific FPGA programming file
  - Programming the target device
  - Generation of a model that contains a Simulink Real-Time interface subsystem

See Also

“Getting Started with the HDL Workflow Advisor” on page 29-5

Set Target Overview

In the **Set Target** folder, you can select a target FPGA device and define the interface generated for the device.

- **Set Target Device and Synthesis Tool**: Select a target FPGA device and synthesis tools.
- **Set Target Reference Design**: For `IP Core Generation` workflow, select a reference design for your target device.
- **Set Target Interface**: For `IP Core Generation`, and `Simulink Real Time FPGA I/O` workflows, use the target platform interface table to assign each port on your DUT to an I/O resource on the target device. Use **Enable HDL DUT port generation for test points** to create DUT output ports for the test point signals in the generated HDL code.
- **Set Target Frequency**: Select the target clock rate for the FPGA implementation of your design.

For more information on each **Set Target** task, select the task icon, and then click the HDL Workflow Advisor **Help** button.

See Also

“Getting Started with the HDL Workflow Advisor” on page 29-5

Set Target Device and Synthesis Tool

The **Set Target Device and Synthesis Tool** task enables you to select an FPGA target device and an associated synthesis tool from a context menu that lists the devices that HDL Workflow Advisor supports.

Description

This task displays these options:

- **Target Workflow**: A context menu that lists the possible workflows that HDL Workflow Advisor supports. Choose from:
  - Generic ASIC/FPGA
  - FPGA-in-the-loop
  - Simulink Real-Time FPGA I/O
  - IP Core Generation
  - Customization for the USRP device
  - Software Defined Radio
- **Target platform**: A context menu that lists the devices the HDL Workflow Advisor supports. Not available for the Generic ASIC/FPGA workflow.
- **Synthesis tool**: Select a synthesis tool, then select the **Family**, **Device**, **Package**, and **Speed** for your synthesis target.

  If your synthesis tool is not one of the **Synthesis tool** options, see “Synthesis Tool Path Setup”. After you set up your synthesis tool path, click **Refresh** to make the tool available in the HDL Workflow Advisor.
- **Project folder**: Specify the project folder name.
- **Tool version**: This box displays the current synthesis tool version.
- **Allow unsupported version**: When you are using an unsupported synthesis tool version, selecting this check box. You can continue to create a project with unsupported synthesis tool

version. If you clear this check box, HDL Coder generates an error when you run this task. This option is unavailable when you are using the supported synthesis tool version.

It is not recommended to use the unsupported tool version because it can potentially cause synthesis failure. For more information on the list of supported tools, see "HDL Language Support and Supported Third-Party Tools and Hardware"

Note If you select `Intel Quartus Pro` or `Microchip Libero SoC` as the **Synthesis tool**, you can run only the `Generic ASIC/FPGA` workflow. When you use these tools, the **Annotate Model with Synthesis Result** task is not available. In this case, you can run the workflow for synthesis, and then view the timing reports to see the critical path.

Set Target Reference Design

The **Set Target Reference Design** task displays when **IP Core Generation** is selected as the **Target Workflow** and the **Target Platform** is not generic. This task displays the reference design input parameters and the tool version. A **Reference design parameters** section displays any custom parameters that you specify for the reference design.

Description

The task displays the following options:

- **Reference design**: A context menu that lists the reference designs that HDL Coder supports and any custom reference designs that you specify. To learn more about creating a custom board and reference design, see "Board and Reference Design Registration System" on page 40-89.
- **Reference design tool version**: A text box that displays the current reference design tool version. It is recommended to use a reference design tool version that is compatible with the supported tool version. If there is a tool version mismatch, HDL Coder generates an error when you run this task. The tool version mismatch can potentially cause the **Create Project** task to fail.

  If you select the **Ignore tool version mismatch** check box, HDL Coder generates a warning instead of an error. You can attempt to continue with creating the reference design project.
- **Reference design parameters**: A context menu that lists the parameters of the reference design. These parameters can be parameters available with the default reference designs that HDL Coder supports or parameters that you define for your custom reference design. For more information, see "Define Custom Parameters and Callback Functions for Custom Reference Design" on page 40-98.

  - **FPGA Data Capture (HDL Verifier required)**: Generate and integrate the data capture IP into your reference design. Use FPGA data capture to observe signals from your design while the design is running on the FPGA. This feature captures a window of signal data from the FPGA and returns the data to MATLAB or Simulink over a JTAG or Ethernet connection. To capture data over a JTAG connection, set this parameter to `JTAG`. To capture data over an Ethernet connection, set this parameter to `Ethernet`. Then, map each signal that you want to capture to the `FPGA Data Capture` interface in the **Set Target Interface** task.

    Note

- FPGA data capture support for JTAG connections is available for Intel and Xilinx boards. Support for Ethernet connections is available for Xilinx boards only.
- FPGA data capture in the HDL Workflow Advisor supports programmable logic (PL) Ethernet only. The processing system (PS) Ethernet is not supported.
- By default, the `Ethernet` option is available for the Artix-7 35T Arty and Kintex-7 KC705 boards. To enable this option for other Xilinx boards that have the Ethernet physical layer (PHY), manually add the Ethernet media access controller (MAC) Hub IP in the `plugin_board` file using the `addEthernetMACInterface` method before you launch the HDL Workflow Advisor.
- FPGA data capture in the HDL Workflow Advisor does not support SGMII interface.

To use this capability, you must have the HDL Verifier hardware support packages installed and downloaded. See "Download FPGA Board Support Package" (HDL Verifier).

- **Board IP Address**: Specify the IP address of the Ethernet port on the target board as a dotted-quad value. The target IP address must be a set of four numbers consisting of integers in the range [0, 255] that are separated by three dots. The default value is `192.168.0.2`.

  To enable this parameter, set **FPGA Data Capture (HDL Verifier required)** to `Ethernet`.
- **Insert AXI Manager (HDL Verifier required)**:

  By default, HDL Coder adds the **Insert AXI Manager (HDL Verifier required)** parameter to all reference designs. When you set this parameter to `JTAG`, the code generator inserts the JTAG AXI Manager IP into your reference design. When you set this parameter to `Ethernet`, the code generator inserts the UDP AXI Manager IP into your reference design.

  Note By default, the `Ethernet` option is available for only the Artix-7 35T Arty, Kintex-7 KC705, and Virtex-7 VC707 boards. To enable this option for other Xilinx boards that have the Ethernet physical layer (PHY), manually add the Ethernet media access controller (MAC) Hub IP in the `plugin_board` file using the `addEthernetMACInterface` method before you launch the HDL Workflow Advisor tool.

  By using the AXI manager IP, you can easily access the AXI registers in the generated DUT IP core on a hardware board from MATLAB or Simulink through the JTAG or Ethernet connection. See also "Set Up AXI Manager" (HDL Verifier).

Set Target Interface

The **Set Target Interface** task displays properties of input and output ports on your DUT and enables you to map these ports to I/O resources on the target device.

Description

Set Target Interface displays the Target Platform Interface Table, which shows:

- The name, port type (inputs and outputs), and data type for each port on your DUT.
- A context menu listing the available I/O resources for the target device.

These resources are device-specific. For detailed information on each resource, see the documentation for your FPGA development board.

Set Target Interface

The **Set Target Interface** task that displays when **Simulink Real-Time FPGA I/O** or **IP Core Generation** is selected as the **Target Workflow**. Select a processor-FPGA synchronization mode and map your DUT input ports, output ports, and test points to I/O resources on the target device.

Description

Coprocessing mode is not supported for the **Simulink Real-Time FPGA I/O** workflow. For **Processor/FPGA synchronization**, select:

- **Free running** if you do not want your processor and FPGA to be automatically synchronized.
- **Coprocessing – blocking** if you want HDL Coder to generate synchronization logic for the FPGA automatically, so that the processor and FPGA run in tandem. Select this mode when FPGA execution time is short relative to the processor sample time and you want the FPGA to complete the synchronization before the processor continues.

This setting is saved with the model as the `ProcessorFPGASynchronization` HDL block property for the DUT block.

Selecting the **Enable HDL DUT port generation for testpoints**:

- Marks test point signals for code generation. See “Model and Debug Test Point Signals with HDL Coder” on page 14-71.
- Enables the Enable HDL DUT output port generation for test points configuration set option.
- Refreshes the target interface table to display the test point output ports in the interface table.

The Target Platform Interface Table displays:

- The name, port type (input, output, and test point), and data type for each port on your DUT.
- A context menu listing the available I/O resources for the target device.

  These resources are device-specific. For detailed information on each resource, see the documentation for your FPGA development board.

See Also

- “Processor and FPGA Synchronization” on page 39-38
- “Custom IP Core Generation” on page 39-17
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 40-113
- Enable HDL DUT output port generation for test points

Set Target Frequency

Specify the target frequency for these workflows:

- `Generic ASIC/FPGA`: Specify the target frequency that you want your design to achieve. HDL Coder generates a timing constraint file for that clock frequency and adds the constraint to the FPGA synthesis tool project that you create in the **Create Project** task. If the target frequency is not achievable, the synthesis tool generates an error.
- `IP Core Generation`: Specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.
- `Simulink Real-Time FPGA I/O`: For Speedgoat boards that are supported by `Xilinx ISE`, specify the target frequency to generate the clock module to produce the clock signal with that frequency.

  The Speedgoat boards that are supported by `Xilinx Vivado` use the `IP Core Generation` workflow infrastructure. Specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.

See Also

Target Frequency

Prepare Model for HDL Code Generation Overview

The tasks in the **Prepare Model For HDL Code Generation** folder check the model for compatibility with HDL code generation. If a check encounters a condition that raises a code generation warning or error, the right pane of the HDL Workflow Advisor displays information about the condition and how to fix it. The **Prepare Model For HDL Code Generation** folder contains these checks:

- **Check Model Settings** check expedites model checks by removing redundant checks. The check provides you the option to open the **HDL Code Advisor** checks in a separate window and run those checks.
- **Check FPGA-in-the-Loop Compatibility**: Check model compatibility with FPGA-in-the-loop, specifically:
  - Not allowed: sink/source subsystems, single/double data types, zero sample time
  - Must be present: HDL Verifier

  This option is available only if you select `FPGA-in-the-Loop` for Target workflow.
- **Check USRP Compatibility**: The model must have two input ports and two output ports of signed 16-bit signals.

  This option is available only if you select `Customization for the USRP Device` for Target workflow.

For summary information on each **Prepare Model For HDL Code Generation** task, select the task icon, and then click the HDL Workflow Advisor **Help** button.

See Also

"Getting Started with the HDL Workflow Advisor" on page 29-5

Check Model Settings

Check Model Settings checks model-wide parameter settings for HDL code generation compatibility of the model.

Description

This check examines the model parameters for compatibility with HDL code generation and flags conditions that raise an error or a warning during code generation. The HDL Workflow Advisor displays a table with the following information about each condition detected:

- Block: Hyperlink to the model configuration dialog box page that contains the error or warning condition.
- Settings: Name of the model parameter that caused the error or warning condition.
- Current: Current value of the setting.
- Recommended: Recommended value of the setting.
- Severity: Severity level of the warning or error condition. Minimally, fix settings that are tagged as `error`.

This check provides a button to open the **HDL Code Advisor** checks in a separate window. Clicking **Run This Task** does not open the **HDL Code Advisor** checks. HDL Code Advisor can run additional HDL code generation compatibility checks not covered in this task. For more info, see "Model configuration checks" on page 38-10.

Tip To set reported settings to their recommended values, click the **Modify All** button. You can then rerun the check again and proceed to the next check.

Check FPGA-in-the-Loop Compatibility

HDL Verifier checks model for compatibility with FPGA-in-the-loop processing.

See Also

"Prepare DUT For FIL Interface Generation" (HDL Verifier).

HDL Code Generation Overview

The tasks in the **HDL Code Generation** folder enable you to:

- Set and validate HDL code and test bench generation parameters. Most parameters on the **HDL Code Generation** pane in the Configuration Parameters dialog box and the Model Explorer are supported.

- Generate any or all of:
  - RTL code
  - RTL test bench
  - Cosimulation model
  - SystemVerilog DPI test bench

To run the tasks in the **HDL Code Generation** folder automatically, select the folder and click **Run All**.

Tip After each task in this folder runs, HDL Coder updates the Configuration Parameters dialog box and the Model Explorer.

Set HDL Options

Optional task to open the HDL Coder Configuration Parameters dialog box.

Description

The **Set HDL Options** is an optional task. This task provides you with the option to open the HDL Coder Configuration Parameters dialog box in a separate window. Changes to the configuration parameters are used in the next HDL Coder Workflow Advisor task.

Note Before doing this task, close the HDL Coder Configuration Parameters dialog box.

Limitations

When the Workflow Advisor window is open for the current design under test (DUT), these configuration parameters are disabled for editing:

- Name of the DUT model or subsystem.
- Name and path of the target code generation folder.
- Name of the synthesis tool.
- Device family selection.
- Device selection.
- Device package selection.
- Device speed selection.
- Target frequency.

If you make any changes to these configuration parameters, rerun all the previous Workflow Advisor tasks. If you change the name of the DUT model or subsystem, restart the Workflow Advisor and rerun all the Workflow Advisor tasks.

Generate RTL Code and Testbench

Select and initiate generation of RTL code, RTL test bench, and cosimulation model.

Description

The **Generate RTL Code and Testbench** task enables choosing what type of code or model that you want to generate. You can select any combination of the following:

- **Generate RTL code**: Generate RTL code in the target language.
- **Generate test bench**: Generate the test bench if **HDL Test Bench** is selected in HDL Coder Configuration Parameters > **Test Bench** > **Test Bench Generation Output**.
- **Generate validation model**: Generate a validation model that highlights generated delays and other differences between your original model and the generated cosimulation model. With a validation model, you can observe the effects of streaming, resource sharing, and delay balancing.

  The validation model contains the DUT from the original model and the DUT from the generated cosimulation model. Using the validation model, you can verify that the output of the optimized DUT is bit-true to the results produced by the original DUT.

See Also

"Generating a Simulink Model for Cosimulation with an HDL Simulator" (Filter Design HDL Coder).

Verify with HDL Cosimulation

Run this step to verify the generated HDL code using cosimulation between the HDL Simulator and the Simulink test bench. This step shows as a Workflow Advisor Task only if you:

- Select **Generate test bench** in **Generate RTL Code and Testbench**.
- Select **Cosimulation model** and specify a **Simulation Tool** in HDL Coder Configuration Parameters > **Test Bench** > **Test Bench Generation Output**.

Generate RTL Code and IP Core

Select and initiate generation of RTL code and a custom IP core.

Description

In the **Generate RTL Code and IP Core** task, specify characteristics of the generated IP core:

- **IP core name**: Enter the IP core name.

  This setting is saved with the model as the `IPCoreName` HDL block property for the DUT block.
- **IP core version**: Enter the IP core version number. HDL Coder appends the version number to the IP core name to generate the output folder name.

  This setting is saved with the model as the `IPCoreVersion` HDL block property for the DUT block.
- **IP core folder** (not editable): HDL Coder generates the IP core files in the output folder shown, including the HTML documentation.

- **IP repository**: If you have an IP repository folder, enter its path manually or by using the **Browse** button. The coder copies the generated IP core into the IP repository folder.
- **Additional source files**: If you are using a black box interface in your design to include existing Verilog, SystemVerilog or VHDL code, enter the file names. Enter each file name manually, separated with a semicolon (;) or by using the **Add** button.

  This setting is saved with the model as the `IPCoreAdditionalFiles` HDL block property for the DUT block.
- **FPGA Data Capture buffer size**: Specify the size of the memory in the generated IP core. The width of the memory is the total bit width of the data signals. The buffer size uses values equal to $128*(2^n)$, where n is an integer. By default, the buffer size is `128` (n = 0). The maximum value of n is `13`, which means that the maximum value for buffer size is $128*(2^{13})$ = 1048576.

  This setting is saved with the model as the `IPDataCaptureBufferSize` HDL block property for the DUT block.
- **FPGA Data Capture maximum sequence depth**: Specify the maximum sequence depth to capture data from an FPGA in one or more trigger stages. The maximum sequence depth is an integer that ranges from 1 to 10. By default, the maximum sequence depth is `1`. To capture the specified data by providing a set of trigger conditions in multiple stages, set the maximum sequence depth to a value greater than 1.

  This setting is saved with the model as the `IPDataCaptureSequenceDepth` HDL block property for the DUT block.
- **Include capture condition logic in FPGA Data Capture**: Select this parameter to include capture condition logic in the IP core. Include capture condition logic to use a capture condition to control which data to capture from the FPGA. The IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition. For more information on capture conditions, see “Capture Conditions” (HDL Verifier).

  Set up a capture condition in the FPGA Data Capture tool or the `hdlverifier.FPGADataReader` System object.

  This setting is saved with the model as the `IncludeDataCaptureControlLogicEnable` HDL block property for the DUT block.
- When you select **Insert AXI Manager**, AXI4-Slave ID Width value is adjusted for the DUT IP core that HDL Coder generates. When you select **Insert AXI Manager** and add custom IPs, you must specify the new adjusted slave ID width value. For example, when you select **Insert AXI Manager** and add custom IPs, if you get this error message during the **Create Project** task:

```
2020.12.07.16:29:08 Info: Adding audio_pll_0 [altera_up_avalon_audio_pll 18.0]
2020.12.07.16:29:08 Info: Parameterizing module audio_pll_0
2020.12.07.16:29:08 Info: Adding hps_0 [altera_hps 18.1]
2020.12.07.16:29:08 Info: Parameterizing module hps_0
2020.12.07.16:29:08 Info: Adding led_count_ip_0 [led_count_ip 1.0]
2020.12.07.16:29:08 Info: Parameterizing module led_count_ip_0
2020.12.07.16:29:08 Info: Adding pll_0 [altera_pll 18.1]
2020.12.07.16:29:08 Info: Parameterizing module pll_0
2020.12.07.16:29:08 Info: Building connections
2020.12.07.16:29:08 Info: Parameterizing connections
2020.12.07.16:29:08 Info: Validating
2020.12.07.16:29:16 Info: Done reading input file
2020.12.07.16:29:20 Info: system_soc.hps_0: HPS Main PLL counter settings: n = 0  m = 47
2020.12.07.16:29:20 Info: system_soc.hps_0: HPS peripherial PLL counter settings: n = 0  m = 39
2020.12.07.16:29:20 Info: system_soc.pll_0: The legal reference clock frequency is 5.0 MHz..650.0 MHz
2020.12.07.16:29:20 Info: system_soc.pll_0: Able to implement PLL with user settings
2020.12.07.16:29:20 Error: system_soc.I2C_SSM2603_0.s_axi: Width of slave ID signals (12) must be at least 13. Increase slave ID width or reduce ID widths for any connected AXI masters.
2020.12.07.16:29:20 Error: system_soc.I2S_SSM2603_0.s_axi: Width of slave ID signals (12) must be at least 13. Increase slave ID width or reduce ID widths for any connected AXI masters.
```

  Regenerate the IP cores by setting the target platform as either Generic Intel or Generic Xilinx. Then map one of the DUT ports to the AXI4 interface, calculate the new width by using `New width = Base Width + log2(Number of AXI Masters +1)`, and enter the calculated width value in **AXI4 Slave ID width**. Add the regenerated IP cores with the new width values to the

reference design folder. Right-click **Create Project** and select **Run to Selected Task**. In this example, the two custom IPs are I2C and I2s. The new calculated width is 13.

- **Generate IP core report**: Leave this option selected to generate HTML documentation for the IP core.
- **Enable readback on AXI4 slave write registers**: Select this option if you want to read back the value that is written to the AXI4 slave registers by using the AXI4 slave interface. When you run this task, the code generator adds a mux for each AXI4 register in the address decoder logic. This mux compares the address that the data is written to when reading the values. If you are reading from multiple AXI4 slave registers, the readback logic becomes a long mux chain that can affect synthesis frequency.

  This setting is saved with the model as the `AXI4RegisterReadback` HDL block property for the DUT block.
- **Enable clock domain crossing on AXI4-Lite registers**: Select this option if you want to run the register interface at a slower clock frequency than your high frequency design under test (DUT) algorithm. To use this option, set your **Synthesis Tool** to `Xilinx Vivado`.This option can be used only for AXI4-Lite interfaces.
- **Generate default AXI4 slave interface**: Leave this option selected if you want to generate an HDL IP core with the AXI4 slave interface for signals such as clock, reset, ready, timestamp, and so on. If you want to generate a generic HDL IP core without any AXI4 slave interfaces, clear this check box. Make sure that you do not map any of the DUT ports to AXI4 or AXI4-Lite interfaces. You can map the ports to only External or Internal IO interfaces or to AXI4-Stream interface with TLAST mapping.

  This setting is saved with the model as the `GenerateDefaultAXI4Slave` HDL block property for the DUT block.
- **Expose DUT clock enable input port**: Select this option if you want to expose the DUT clock enable input port. Trigger the DUT from upstream IPs by using the clock enable input port. If your design has ports mapped to the AXI4 Slave interface, this option is disabled.
- **Expose DUT clock enable output port**: Select this option if you want to expose the clock enable output port to downstream IPs. Drive or synchronize downstream custom IPs by using the clock enable output port.
- **AXI4 slave port to pipeline register ratio**: Specify the number of AXI4 subordinate ports for which you want a pipeline register to be inserted by using the **AXI4 Slave port to pipeline register ratio** setting in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see “Model Design for AXI4 Slave Interface Generation” on page 40-3.

  Values: `'off'` (default) | `'on''10''20''35''50'`

See Also

- “Custom IP Core Generation” on page 39-17
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 39-33
- “Custom IP Core Report” on page 39-20

FPGA Synthesis and Analysis Overview

Create projects for supported FPGA synthesis tools, perform FPGA synthesis, mapping, and place/route tasks, and annotate critical paths in the original model.

Description

The tasks in the **FPGA Synthesis and Analysis** folder enable you to:

- Create FPGA synthesis projects for supported FPGA synthesis tools.
- Start supported FPGA synthesis tools, using the project files to perform synthesis, mapping, and place/route tasks.
- Annotate your original model with critical path information obtained from the synthesis tools.

For a list of supported third-party synthesis tools, see “Third-Party Synthesis Tools and Version Support”.

The tasks in the folder are:

- **Create Project**
- **Perform Synthesis and P/R**
- **Annotate Model with Synthesis Result**

See Also

“HDL Code Generation and FPGA Synthesis from Simulink Model”

Create Project

Create an FPGA synthesis project for a supported FPGA synthesis tool.

Description

This task creates a synthesis project for the selected synthesis tool and loads the project with the HDL code generated for your model.

When the project creation is complete, the HDL Workflow Advisor displays a link to the project in the right pane. Click this link to view the project in the synthesis tool project window.

Synthesis objective

Select a synthesis objective to generate tool-specific optimization Tcl commands for your project. If you specify `None`, no Tcl commands are generated.

See “Synthesis Objective to Tcl Command Mapping” on page 29-45.

Additional source files

Enter additional HDL source files that you want included in your synthesis project. Enter each file name manually, separated with a semicolon (;) or by using the **Add Source** button.

For example, you can include HDL source files (`.vhd` or `.v`) or a constraint file (`.ucf` or `.sdc`).

Additional project creation Tcl files

Enter additional project creation Tcl files that you want to include in your synthesis project. Enter each file name manually, separated with a semicolon (;) or by using the **Add Tcl** button.

For example, you can include a Tcl script (`.tcl`) to execute after creating the project.

See Also

- “Third-Party Synthesis Tools and Version Support”
- “HDL Code Generation and FPGA Synthesis from Simulink Model”
- “Synthesis Objective to Tcl Command Mapping” on page 29-45

Perform Synthesis and P/R Overview

Start supported FPGA synthesis tools to perform synthesis, mapping, and place/route tasks.

Description

The tasks in the **Perform Synthesis and P/R** folder enable you to start supported FPGA synthesis tool and:

- Synthesize the generated HDL code.
- Perform mapping and timing analysis.
- Perform place and route functions.

For a list of supported third-party synthesis tools, see “Third-Party Synthesis Tools and Version Support”.

See Also

“HDL Code Generation and FPGA Synthesis from Simulink Model”

Perform Logic Synthesis

Start supported FPGA synthesis tool and synthesize the generated HDL code.

Description

The **Perform Logic Synthesis** task:

- Starts the synthesis tool in the background.
- Opens the previously generated synthesis project, compiles HDL code, synthesizes the design, and emits netlists and related files.
- Displays a synthesis log in the **Result** subpane.

See Also

“HDL Code Generation and FPGA Synthesis from Simulink Model”

Perform Mapping

Starts supported FPGA synthesis tool and maps the synthesized logic design to the target FPGA.

Description

The **Perform Mapping** task:

- Starts the synthesis tool in the background.
- Runs a mapping process that maps the synthesized logic design to the target FPGA.
- Emits a circuit description file for use in the place and route phase.
- Emits pre-routing timing information for use in critical path analysis and back annotation of your source model.
- Displays a log in the **Result** subpane.

Enable **Skip pre-route timing analysis** if your tool does not support early timing estimation. When this option is enabled, the **Annotate Model with Synthesis Result** task sets **Critical path source** to **post-route**.

See Also

"HDL Code Generation and FPGA Synthesis from Simulink Model"

Perform Place and Route

Starts the synthesis tool in the background and runs a Place and Route process.

Description

The **Perform Place and Route** task:

- Starts the synthesis tool in the background.
- Runs a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.
- Starts post-routing timing information for use in critical path analysis and back annotation of your source model.
- Displays a log in the **Result** subpane.

If you select **Skip this task**, the HDL Workflow Advisor executes the workflow, but omits the **Perform Place and Route** task, marking it `Passed`. If you prefer to do place and route work manually, you might want to select **Skip this task**.

If **Perform Place and Route** fails, but you want to use the post-mapping timing results to find critical paths in your model, you can select **Ignore place and route errors** and continue to the **Annotate Model with Synthesis Result** task.

See Also

"HDL Code Generation and FPGA Synthesis from Simulink Model"

Run Synthesis

Starts Xilinx Vivado and executes the Vivado **Synthesis** step.

If you do not want to do early timing estimation, enable**Skip pre-route timing analysis**.

Run Implementation

Starts Xilinx Vivado and executes the Vivado **Implementation** step.

If you select **Skip this task** , the HDL Workflow Advisor omits the **Run Implementation** task, marking it `Passed`. If you prefer to do place and route work manually, select **Skip this task**.

If **Run Implementation** fails, you can select **Ignore place and route errors** and continue to the **Annotate Model with Synthesis Result** task.

Check Timing Report

If there are timing failures during this task, the task does not fail. You must check the timing report for timing failures.

Annotate Model with Synthesis Result

Analyzes pre- or post-routing timing information and visually highlights critical paths in your model.

Description

The **Annotate Model with Synthesis Result** task helps you to identify critical paths in your model. Depending on your option selection, the task analyzes pre- or post-routing timing information produced by the **Perform Synthesis and P/R** task group and visually highlights one or more critical paths in your model.

Note If you select `Intel Quartus Pro` or `Microchip Libero SoC` as the **Synthesis tool**, the **Annotate Model with Synthesis Result** task is not available. Run the workflow to synthesis, and then view the timing reports to see the critical path.

If you select **Generate FPGA top level wrapper** in the **Generate RTL Code and Testbench** task, **Annotate Model with Synthesis Result** is not available. To perform back-annotation analysis, clear the check box for **Generate FPGA top level wrapper**.

Input Parameters

Critical path source

Select **pre-route** or **post-route**.

The **pre-route** option is unavailable when **Skip pre-route timing analysis** is enabled in the previous task group.

Critical path number

You can annotate up to three critical paths. Select the number of paths that you want to annotate.

Choose Model to Annotate

You can perform the annotation on the original as well as generated model. Select the `original` or `generated` model that you want to annotate. For more information on generated model, see “Generated Model and Validation Model” on page 21-10.

Show all paths

Show critical paths, including duplicate paths.

Show unique paths

Show only the first instance of a path that is duplicated.

Show delay data

Annotate the cumulative timing delay on each path.

Show ends only

Show the endpoints of each path, but omit the connecting signal lines.

Results and Recommended Actions

When the **Annotate Model with Synthesis Result** task runs to completion, HDL Coder displays the DUT with critical path information highlighted.

See Also

“HDL Code Generation and FPGA Synthesis from Simulink Model”

Download to Target Overview

The **Download to Target** folder supports the following tasks:

- **Generate Programming File**: Generate an FPGA programming file.
- **Program Target Device**: Download generated programming file to the target development board.
- **Generate Simulink Real-Time Interface** (for Speedgoat target devices only): Generate a model that contains a Simulink Real-Time interface subsystem.

For summary information on each **Download to Target** task, select the task icon, and then click the HDL Workflow Advisor **Help** button.

See Also

“Getting Started with the HDL Workflow Advisor” on page 29-5

Generate Programming File

The **Generate Programming File** task generates an FPGA programming file that is compatible with the selected target device.

Program Target Device

The **Program Target Device** task downloads the generated FPGA programming file to the selected target device.

Before executing the **Program Target Device** task, make sure that your host PC is properly connected to the target development board using the required programming cable.

Generate Simulink Real-Time Interface

The **Generate Simulink Real-Time Interface** task generates a model containing an interface subsystem that you can plug into a Simulink Real-Time model.

The naming convention for the generated model is:

```
gm_fpgamodelname_slrt
```

where `fpgamodelname` is the name of the original model.

Save and Restore HDL Workflow Advisor State

You can save the current settings of the HDL Workflow Advisor to a named restore point. Later, you can restore the same settings by loading the restore point data into the HDL Workflow Advisor.

See Also

"Getting Started with the HDL Workflow Advisor" on page 29-5.

FPGA-in-the-Loop (FIL) Implementation

Set FIL options and run FIL processing.

Set FPGA-in-the-Loop Options

Set connection type, board IP, and MAC addresses and select additional files, if required.

Connection

Select either `JTAG` (Altera boards only) or `Ethernet`.

Board IP Address

Set the IP address of the board if it is not the default IP address (192.168.0.2).

Board MAC Address

Under most circumstances, you do not need to change the Board MAC address. If you connect more than one FPGA development board to a single computer (for which you must have a separate NIC for each board), you must change the Board MAC address. You must change the Board MAC address for additional boards so that each address is unique.

Additional Source Files

Select additional source files for the HDL design that is to be verified on the FPGA board, if required. HDL Workflow Advisor attempts to identify the file type. Change the file type in the **File Type** column if it is incorrect.

Build FPGA-in-the-Loop

During the build process:

- FPGA-in-the-loop generates a FIL block named after the top-level module and places it in a new model.
- After new model generation, FIL opens a Command Prompt. In this Command Prompt, the FPGA design software performs synthesis, fit, place-and-route, timing analysis, and FPGA programming file generation. When the process is complete, a message in the Command Prompt prompts you to close the window.
- FPGA-in-the-loop builds a test bench model around the generated FIL block.

Embedded System Integration

Tasks in this folder integrate your generated HDL IP core into the embedded processor.

Create Project

Create project for embedded system tool.

In the message window, after the project is generated, you can click the project link to open the generated embedded system tool project.

Embedded system tool
: Embedded design tool.

Project folder
: Folder where your generated project files are saved.

Synthesis objective
: Select a synthesis objective to generate tool-specific optimization Tcl commands for your project. If you specify `None`, no Tcl commands are generated.

  To learn how the synthesis objectives map to Tcl commands, see "Synthesis Objective to Tcl Command Mapping" on page 29-45.

Enable IP caching
: Create IP cache to reduce reference design synthesis time. When you enable IP caching, the code generator creates an IP cache. The `IP Core Generation` workflow uses an out-of-context (OOC) workflow. This workflow synthesizes the IP in the reference design out of context from the top-level design. You can reuse this cache in subsequent project runs, which reduces reference design synthesis time. To learn more, see "IP Caching for Faster Reference Design Synthesis" on page 39-44.

Generate Software Interface

To generate the embedded C code, generate a Simulink software interface model with the IP core driver blocks. To verify the IP core functionality and connect to the onboard memory locations, generate a host interface model, host interface script, or both with the AXI Manager.

When you clear the **Generate Simulink software interface model**, **Generate host interface model**, and **Generate host interface script** check boxes, the HDL Workflow Advisor skips this task.

Description

In the **Generate Software Interface** task, specify a software interface that you want to generate for the IP core.

- **Generate Simulink software interface model** — Select this parameter to generate a Simulink software interface model for an SoC device. The software interface model is your original model with the AXI driver blocks replacing the parts you want to run on hardware. This parameter is not available for standalone FPGA boards.

  After you generate the Simulink software interface model, you can generate C code from it by using Embedded Coder. If you do not have the Embedded Coder hardware support package for the target board installed, this parameter is not available. For example, if the target hardware board is a Zynq device, you must have the Embedded Coder Support Package for Xilinx Zynq Platform installed.
- **Operating system** — Select your target operating system.
- **Host target interface** — Select an interface that communicates between your host machine and the target hardware. Use one of these options.

  - `JTAG AXI Manager (HDL Verifier)` — Use the JTAG interface to access AXI4 and AXI4-Lite registers on the target hardware. To enable this option, in the **Set Target Reference Design** task, set **Insert AXI Manager (HDL Verifier required)** to `JTAG` and in the **Set Target Interface** task, map each DUT signal that you want to capture to the `AXI4` or `AXI4-Lite` interface.
  - `Ethernet AXI Manager (HDL Verifier)` — Use the Ethernet interface to access AXI4 and AXI4-Lite registers on the target hardware. To enable this option, in the **Set Target Reference Design** task, set **Insert AXI Manager (HDL Verifier required)** to `Ethernet` and in the **Set Target Interface** task, map each DUT signal that you want to capture to the `AXI4` or `AXI4-Lite` interface.
  - `Ethernet` — Use the Ethernet interface to access the generated IP core deployed on your target hardware. This option is not available for standalone FPGA boards.
- **Generate host interface model** — Select this parameter to generate a host interface model. The host interface model enables you to write to or read from the memory-mapped locations on the target hardware over a JTAG or Ethernet cable by using the AXI Manager Write and AXI Manager Read blocks.

  To enable this parameter, set **Host target interface** to `JTAG AXI Manager (HDL Verifier)` or `Ethernet AXI Manager (HDL Verifier)`.
- **Generate host interface script** — Select this parameter to generate a host interface script. The host interface script contains commands that enable you to connect to the target hardware and to write to or read from the generated IP core by using the AXI driver blocks or the AXI Manager.

Build FPGA Bitstream

Generate bitstream for embedded system.

Run build process externally

Enable this option to run the build process in parallel with MATLAB. If this option is disabled, you cannot use MATLAB until the build is finished. This option is valid only when you use the `IP Core Generation` workflow.

Tcl file for synthesis build

To customize your synthesis build, save your custom Tcl commands in a file and select `Custom`. Enter the file path manually or find the path by using the **Browse** button. The contents of your custom Tcl file are inserted between the Tcl commands that open and close the project.

If you select `Custom` and want to generate a bitstream, the bitstream generation Tcl command must refer to the top file wrapper name and location either directly or implicitly. For example, the following Xilinx Vivado Tcl command generates a bitstream and implicitly refers to the top file name and location:

```
launch_runs impl_1 -to_step write_bitstream
```

Enable routed design checkpoint for build

Select this option to expedite bitstream generation time by using the design checkpoint from the previous build. This option is available only when using the Xilinx Vivado synthesis tool.

Routed design checkpoint file for build

To use this option, select **Enable routed design checkpoint for build**. To use the default routed design checkpoint file, select `Default`. The default file location is `hdl_prj\checkpoint\system_routed.dcp`. To use a custom routed design checkpoint file, select `Custom` and provide the file path to your custom file location. This option is available only when using the Xilinx Vivado synthesis tool.

Routed design checkpoint file

To use this option, select `Custom` for **Routed design checkpoint file for build**. Use this option to point to your custom routed design checkpoint file. After bitstream generation is completed the new routed checkpoint design file is written to the location specified in **Routed design checkpoint file**. This option is available only when using the Xilinx Vivado synthesis tool.

Maximum number of cores for build

Reduce bitstream generation times by using multiple logical cores of the PC. Use this option to choose the maximum number of PC cores to use. Selecting `synthesis tool default` selects the maximum number of cores set in the synthesis tool. To manually select the maximum number of cores, select between `2` and `32`.

Program Target Device

Program the connected target SoC device. Specify the **Programming method** for the target device:

- `JTAG`: Uses a JTAG cable to program the target SoC device.
- `Download`: This is the default **Programming method**. Copies the generated FPGA bitstream, device tree, and system initialization scripts to the SD card on the Zynq board and keeps the

bitstream on the SD card persistently. To use this programming method, you do not need Embedded Coder. You can create an SSH object by specifying the **IP Address**, **SSH Username**, and **SSH Password**. HDL Coder uses the SSH object to copy the bitstream to the SD card and reprogram the board.

To define your own function to program the target device in your custom reference design, you can use the `Custom` **Programming method**. To use the custom programming, register the function handle of the custom programming function by using the `CallbackCustomProgrammingMethod` method of the `hdlcoder.ReferenceDesign` class. For example:

```
hRD.CallbackCustomProgrammingMethod = ...
    @parameter_callback.callback_CustomProgrammingMethod;
```

For more information, see “Program Target FPGA Boards or SoC Devices” on page 39-64.

HDL Code Advisor

- “HDL Coder Checks in Model Advisor / HDL Code Advisor Overview” on page 37-3
- “Model configuration checks overview” on page 37-4
- “Check for model parameters suited for HDL code generation” on page 37-5
- “Check for global reset setting for Xilinx and Altera devices” on page 37-7
- “Check inline configurations setting” on page 37-8
- “Check algebraic loops” on page 37-9
- “Check for visualization settings” on page 37-10
- “Check delay balancing setting” on page 37-11
- “Check for ports and subsystems overview” on page 37-12
- “Check for invalid top level subsystem” on page 37-13
- “Check for blocks and block settings overview” on page 37-14
- “Check for infinite and continuous sample time sources” on page 37-15
- “Check for unsupported blocks” on page 37-16
- “Check for large matrix operations” on page 37-17
- “Check for MATLAB Function block settings” on page 37-18
- “Check for Stateflow chart settings” on page 37-19
- “Check for obsolete Unit Delay Enabled/Resettable Blocks” on page 37-20
- “Check for blocks that have nonzero output latency” on page 37-21
- “Check for unsupported storage class for signal objects” on page 37-22
- “Check for HDL Reciprocal block usage” on page 37-23
- “Check for Trigonometric Function block for LUT-based approximation method” on page 37-24
- “Native Floating Point Checks Overview” on page 37-25
- “Check for single datatypes in the model” on page 37-26
- “Check for double data types in the model” on page 37-27
- “Check for Data Type Conversion blocks with incompatible settings” on page 37-28
- “Check for HDL Reciprocal block usage” on page 37-29
- “Check for Relational Operator block usage” on page 37-30
- “Check for unsupported blocks with Native Floating Point” on page 37-31
- “Check blocks with nonzero ULP error” on page 37-32
- “Industry standard checks overview” on page 37-33
- “Check file extension” on page 37-34
- “Check naming conventions” on page 37-35
- “Check top-level subsystem/port names” on page 37-36
- “Check module/entity names” on page 37-37
- “Check signal and port names” on page 37-38

- “Check package file names” on page 37-39
- “Check generics” on page 37-40
- “Check clock, reset, and enable signals” on page 37-41
- “Check architecture name” on page 37-42
- “Check entity and architecture” on page 37-43
- “Check clock settings” on page 37-44

HDL Coder Checks in Model Advisor / HDL Code Advisor Overview

The **HDL Coder** checks in the Model Advisor or the HDL Code Advisor verify and update your Simulink model or subsystem for compatibility with HDL code generation. Running the checks produces a report that lists suboptimal conditions or settings, and then proposes better model configuration settings.

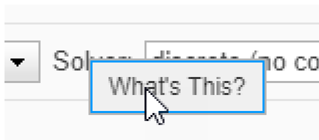
To learn about:

- HDL Code Advisor UI, see "Check HDL Compatibility of Simulink Model Using HDL Code Advisor" on page 38-2.
- Model Advisor, see "Run Model Advisor Checks for HDL Coder" on page 38-6.

The left pane displays folders that perform various checks:

- **Model configuration checks**: Prepare your model for compatibility with HDL code generation. This folder contains checks that verify whether model parameters are HDL-compatible, whether your design contains algebraic loops, and so on.
- **Checks for ports and subsystems**: Verify whether ports and subsystems in your model have settings that are compatible for HDL code generation. The checks include whether you have a valid top-level DUT Subsystem and whether you have specified an initial condition for Enabled Subsystem and Triggered Subsystem blocks.
- **Checks for blocks and block settings**: Verify whether blocks in your model are supported for HDL code generation, and whether the supported blocks have HDL-compatible settings. The checks include whether source blocks in your model have a continuous sample time and whether Stateflow Charts and MATLAB Function blocks have HDL-compatible settings, and so on.
- **Native Floating Point checks**: Verify whether the block is compatible for HDL code generation in `Native Floating Point` mode. The checks include whether the blocks in your Simulink model are supported for HDL code generation with `Native Floating Point`, and whether the model uses single data types, and so on. Native floating-point support in HDL Coder generates target-independent HDL code from your single-precision floating-point model. For more information, see "Generate Target-Independent HDL Code with Native Floating-Point" on page 14-111.
- **industry-standard checks**: Verify whether your Simulink model conforms to the industry-standard rules. Industry-standard rules recommend using certain HDL coding guidelines. When generating code, HDL Coder displays an HDL coding standard report that shows how well the generated code adheres to the industry-standard guidelines. For more information, see "HDL Coding Standards" on page 24-4.

To learn more about each individual check, right-click that check, and select **What's This?**.



See also "HDL Code Advisor Checks" on page 38-9.

Model configuration checks overview

Use the checks in this folder to prepare your model for compatibility with HDL code generation. This folder contains checks that verify whether:

- The model parameters and visualization settings are compatible with HDL code generation.
- Your model uses foreign characters that are incompatible with the current encoding.
- The global reset setting is asynchronous for Altera devices and synchronous for Xilinx devices.
- The `InlineConfigurations` setting is enabled on the model.
- The design contains algebraic loops.

Check for model parameters suited for HDL code generation

Check ID: com.mathworks.HDL.ModelChecker.runModelParamsChecks

Check ID: com.mathworks.HDL.ModelAdvisor.runModelParamsChecks

Check for model parameters set up for HDL code generation.

Description

This check verifies whether the model parameters that you specify are compatible for HDL code generation. This check ensures that you use these settings in the Configuration Parameters dialog box.

Command-Line Parameter Setting	Configuration Parameter Setting
Set Solver to FixedStepDiscrete.	Set **Type** to Fixed-step and **Solver** to Discrete (no continuous states).
Set FixedStep to auto.	Set **Fixed-step size (fundamental sample time)** to auto.
Set EnableMultiTasking to off.	Disable the **Treat each discrete rate as a separate task** check box.
Set AlgebraicLoopMsg to error	Set **Algebraic loop** to error.
Set SingleTaskRateTransMsg to error.	Set **Single task data transfer** to error.
Set MultiTaskRateTransMsg to error.	Set **Multitask data transfer** to error.
Set BlockReduction to off	Disable the **Block Reduction** check box.
Set ConditionallyExecuteInputs to off.	Disable **Conditional input branch execution**.
Set DefaultParameterBehaviour to Inlined. You can set this parameter at the command line by using set_param or hdlsetup.	Set **Default parameter behavior** to Inlined. If you want to set this parameter in the Configuration Parameters dialog box, you must have Simulink Coder. **Note** Enabling this parameter is the same as setting the InlineParams property to on. Setting InlineParams to off changes DefaultParameterBehavior value to Tunable.
Set DataTypeOverride to off.	No dialog box prompt.
Set ProdHWDeviceType to ASIC/FPGA->ASIC/FPGA.	Set **Device vendor** to ASIC/FPGA.
Set ShowLineDimensions and ShowPortDataTypes to on.	In the **Debug** tab, on the **Information Overlays**, select **Base data types** and **Signal Dimensions**.
Set SampleTimeColors to on.	In the Simulink Editor, in the **Debug** tab, select **Information Overlays > Colors**.
Set SignalLoggingSaveFormat to Dataset.	No dialog box prompt.

Command-Line Parameter Setting	Configuration Parameter Setting
Set `UnconnectedInputMsg` to `error` or `warning`.	Set **Unconnected block input ports** to `error` or `warning`.
Set `UnconnectedOutputMsg` to `error` or `warning`.	Set **Unconnected block output ports** to `error` or `warning`.
Set `UnconnectedLineMsg` to `error` or `warning`.	Set **Unconnected lines** to `error` or `warning`.

If there are incompatible model parameters, HDL Coder displays a warning and lists the model parameters that have to be fixed.

Results and Recommended Actions

To fix this warning, click **Modify Settings**, and the code generator runs the `hdlsetup` command to set up the model parameters for HDL code generation. You can then rerun the check.

See Also

- `hdlsetup`
- "Create HDL-Compatible Simulink Model"

Check for global reset setting for Xilinx and Altera devices

Check ID: `com.mathworks.HDL.ModelChecker.runGlobalResetChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runGlobalResetChecks`

Check asynchronous reset setting for Altera devices and synchronous reset setting for Xilinx devices.

Description

This check verifies whether you use a global synchronous reset for a Xilinx device or a global asynchronous reset for an Altera device. You can improve the performance of your design by adhering to this recommended global reset setting depending on whether you target a Xilinx device or an Altera device.

Note If you do not specify a target device, this check passes successfully.

Results and Recommended Actions

To fix this warning, click **Modify Settings**, and the code generator updates the **Reset type** setting to `Synchronous` if you use a Xilinx device and `Asynchronous` if you use an Altera device. You can then rerun the check.

See Also

Reset type

Check inline configurations setting

Check ID: `com.mathworks.HDL.ModelChecker.runInlineConfigurationsChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runInlineConfigurationsChecks`

Check `InlineConfigurations` is enabled.

Description

This check verifies whether you have the `InlineConfigurations` property enabled. By default, this property is enabled, and HDL Coder includes VHDL configurations for the model inline with the rest of the VHDL code. If you disable this property, the code generator displays a warning.

Results and Recommended Actions

To fix this warning, click **Modify Settings**, and the code generator updates the `InlineConfigurations` setting to `on`.

Check algebraic loops

Check ID: `com.mathworks.HDL.ModelChecker.runAlgebraicLoopChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runAlgebraicLoopChecks`

Check model for algebraic loops.

Description

HDL Coder does not support code generation for models in which algebraic loop conditions exist. This check examines the model and fails the check if it detects an algebraic loop.

Results and Recommended Actions

To fix this warning, eliminate algebraic loops from your model and then run this check again.

See Also

"Algebraic Loop Concepts"

Check for visualization settings

Check ID: `com.mathworks.HDL.ModelChecker.runVisualizationChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runVisualizationChecks`

Check for display settings: port data types and sample time color coding.

Description

This check determines whether you have:

- Enabled the **Port Data Types** setting on the model.
- Set the **Sample Time** to **Colors** on the model.

This check displays a message if your Simulink model doesn't have either or both of these settings.

Results and Recommended Actions

Click **Modify Settings**, and the code generator:

- Enables the **Port Data Types** setting on the model.
- Sets the **Sample Time** to **Colors**.

You can then rerun the check.

See Also

- “Display Port Data Types”
- “View Sample Time Information”

Check delay balancing setting

Check ID: `com.mathworks.HDL.ModelChecker.runBalanceDelaysChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runBalanceDelaysChecks`

Check Balance Delays is enabled.

Description

This check reports a warning if the **Balance delays** setting is disabled on the model. When you generate HDL code, certain optimizations or block implementations can introduce delays along some signal paths in your model. If **Balance delays** is disabled, the code generator does not introduce equivalent delays on other parallel signal paths, which can result in a numerical mismatch between the original model and the generated model.

Results and Recommended Actions

To fix this warning, click **Modify Settings**, and the code generator enables the **Balance delays** setting on the model. You can then rerun the check.

See Also

- “Delay Balancing” on page 21-81
- Balance delays

Check for ports and subsystems overview

This folder contains checks that verify whether ports and subsystems in your model have settings that are compatible for HDL code generation. You can verify whether:

- The subsystem in your Simulink model is a valid top level subsystem.
- The initial condition of Enabled Subsystem or Triggered Subsystem blocks in your model is zero.

Check for invalid top level subsystem

Check ID: `com.mathworks.HDL.ModelChecker.runInvalidDUTChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runInvalidDUTChecks`

Check for subsystems that cannot be at the top level for HDL code generation.

Description

This check verifies whether your Subsystem is a valid DUT for generating HDL code. For example, if you use an invalid DUT, such as an Enabled Subsystem, For Each Subsystem or a BlackBox Subsystem, this check displays a failed message and provides a link to the Subsystem.

Results and Recommended Actions

To fix this failed check, place this Subsystem inside another Subsystem, and then use that Subsystem as the DUT. You can then rerun the check.

Check for blocks and block settings overview

These checks verify whether blocks in your model are supported for HDL code generation, and whether the supported blocks have HDL-compatible settings. You can verify whether:

- There are source blocks with infinite sample time in your model.
- The blocks in your Simulink model are compatible for HDL code generation.
- There are unconnected lines, input ports, or output ports in your model.
- There are unresolved or disabled library links.
- MATLAB Function and Stateflow Chart blocks in your model have HDL-compatible settings.
- There are Delay, Unit Delay, and Zero-Order Hold blocks in the model that perform rate transition and replace them with Rate Transition blocks.

Check for infinite and continuous sample time sources

Check ID: `com.mathworks.HDL.ModelChecker.runSampleTimeChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runSampleTimeChecks`

Check source blocks with infinite or continuous sample time.

Description

By default, the **Sample time** parameter of a Constant block is `inf`. During HDL code generation, HDL Coder does not resolve `inf` sample times of Constant blocks if the sample times propagate to the device under test (DUT) output. For more details, see the “HDL Code Generation” section of the Constant block page.

This check reports a warning if your design contains source blocks that have an infinite or continuous sample time.

Results and Recommended Actions

To fix this warning, click **Modify Settings** to update the **Sample time** of these source blocks to inherit through back propagation. That is, HDL Coder sets **Sample time** of these blocks to `-1`. You can then rerun the check.

See Also

- “What Is Sample Time?”
- “Usage of Rate Change and Constant Blocks” on page 18-100

Check for unsupported blocks

Check ID: `com.mathworks.HDL.ModelChecker.runBlockSupportChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runBlockSupportChecks`

Check for unsupported blocks for HDL code generation.

Description

This check displays a failed message if your model uses blocks that are not supported for HDL code generation.

Results and Recommended Actions

To fix this failed check, update your design to use blocks that are supported with HDL Coder. You can then rerun the check.

Check for large matrix operations

Check ID: `com.mathworks.HDL.ModelChecker.runMatrixSizesChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runMatrixSizesChecks`

Check for large matrix operations.

Description

This check displays a warning if your design contains:

- Signals with matrix types that have more than two dimensions. HDL code generation supports matrix types that have a maximum of two dimensions.
- Large matrix operations with Add, Sum, or Product blocks that result in a matrix output with more than ten elements. Synthesizing the generated HDL code from such a design can result in the utilization of large number of resources on the target FPGA. For more details, see the "HDL Code Generation" sections of the block reference pages.

Results and Recommended Actions

To fix this warning, update your design so that there are no matrix types with more than two dimensions and the result of a matrix operation does not produce an output with more than ten elements. To verify that the check passes, compile the design and rerun the check.

See Also

- Product
- “Signal and Data Type Support” on page 14-3

Check for MATLAB Function block settings

Check ID: `com.mathworks.HDL.ModelChecker.runMLFcnBlkChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runMLFcnBlkChecks`

Check HDL compatible settings for MATLAB Function blocks.

Description

This check displays a warning if your model uses MATLAB Function blocks that have settings not recommended for HDL code generation. The settings include checking whether the MATLAB Function block has:

- `fimath` settings defined as per `hdlfimath`. The `hdlfimath` function uses `fimath` settings that are compatible for HDL code generation.
- **Saturate on integer overflow** check box cleared.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the MATLAB Function block settings to be compatible with HDL code generation.

See Also

"Design Guidelines for the MATLAB Function Block" on page 27-19

Check for Stateflow chart settings

Check ID: `com.mathworks.HDL.ModelChecker.runStateflowChartSettingsChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runStateflowChartSettingsChecks`

Check HDL-compatible settings for Stateflow Chart blocks.

Description

This check displays a failed message if your model uses Stateflow Chart blocks that have settings incompatible for HDL code generation. The check passes when you select these settings for the Stateflow charts:

- **Execute (enter) chart at initialization** property is selected.
- **Action Language** is set to `MATLAB`.
- **Support variable-size arrays** property is disabled.

For more information, see “Specify Properties for Stateflow Charts” (Stateflow).

Results and Recommended Actions

To fix this failed check, click **Modify Settings**. The code generator updates the Stateflow Chart settings to be compatible with HDL code generation.

See Also

Chart

Check for obsolete Unit Delay Enabled/Resettable Blocks

Check ID: `com.mathworks.HDL.ModelChecker.runObsoleteDelaysChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runObsoleteDelaysChecks`

Check if the DUT contains obsolete Unit Delay Enabled/Resettable blocks

Description

This check displays a warning if the DUT Subsystem contains any of these blocks:

- Unit Delay Enabled
- Unit Delay Resettable
- Unit Delay Enabled Resettable

These blocks have been obsoleted. The code generator does not recommend usage of these blocks in your Simulink model.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator replaces these blocks with the corresponding synchronous counterparts:

- Unit Delay Enabled is replaced by Unit Delay Enabled Synchronous.
- Unit Delay Resettable is replaced by Unit Delay Resettable Synchronous.
- Unit Delay Enabled Resettable is replaced by Unit Delay Enabled Resettable Synchronous.

These blocks are recommended because they use the State Control block for synchronous simulation behavior and generate hardware-friendly HDL code. For more information, see “Synchronous Subsystem Behavior with the State Control Block” on page 25-75.

Check for blocks that have nonzero output latency

Check ID: `com.mathworks.HDL.ModelChecker.runNFPLatencyChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runNFPLatencyChecks`

Check for blocks that introduce latency in the generated code but do not simulate with latency in original model

Description

Native floating-point operators and certain fixed-point blocks introduce latency in the generated HDL code. This check detects blocks in your Simulink model that introduce latency in the generated HDL code when you use fixed point and floating-point types. If your model uses floating-point types, select **Use Floating Point**.

When you run the check, the **Result** subpane displays hyperlinks to the blocks that have a nonzero output latency, and the latency value. When you generate code, HDL Coder figures out this latency.

Results and Recommended Actions

By using the latency information reported in the **Result** subpane, you can add the corresponding number of delays adjacent to those blocks in your original model, and therefore simulate the original model with latency. The code generator absorbs the delays you added to your model, and does not have to introduce additional latency in the generated model.

For blocks with nonzero latency that are reported by this check, consider the effect this latency has in the validation model. In the generated model and validation model, you see the additional delays that the code generator adds to account for the latency.

See Also

"Latency Considerations with Native Floating Point" on page 14-104

Check for unsupported storage class for signal objects

Check ID: `com.mathworks.HDL.ModelChecker.runSignalObjectStorageClassChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runSignalObjectStorageClassChecks`

Check whether signal object storage class is `'ExportedGlobal'` or `'ImportedExtern'` or `'ImportedExternPointer'`

Description

This check displays a warning if your model contains signals that have the signal object storage class set to `'ExportedGlobal'`, `'ImportedExtern'`, or `'ImportedExternPointer'`. The warning message also provides links to those signals that have the signal object storage class set to one of these signal object storage class specifications.

HDL code generation ignores these storage class specifications that you specify in your design, which may sometimes result in conflicting signal names. When you simulate the validation model, HDL Coder may generate errors.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator replaces those signals that have the signal object storage class specified as `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` to `Auto`.

See Also

`coder.storageClass`

More About

- “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder)
- “Storage Classes for Code Generation from MATLAB Code” (Embedded Coder)

Check for HDL Reciprocal block usage

Check ID: `com.mathworks.HDL.ModelChecker.runHDLRecipChecks`

Check ID:`com.mathworks.HDL.ModelAdvisor.runHDLRecipChecks`

Check if the model uses HDL Reciprocal blocks

Description

This check displays a warning if your Simulink model contains HDL Reciprocal blocks.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator replaces the HDL Reciprocal blocks with Math Reciprocal blocks.

See Also

HDL Reciprocal

Check for Trigonometric Function block for LUT-based approximation method

Check ID: `com.mathworks.HDL.ModelChecker.runUnsupportedLUTTrigFunChecks`

Check ID:`com.mathworks.HDL.ModelAdvisor.runUnsupportedLUTTrigFunChecks`

Check Trigonometric Function blocks in a model that use the look up table (LUT) based approximation method

Description

This check displays a warning if your Simulink model contains Trigonometric Function blocks such as `sin`, `cos`, `sincos`, `atan2`, and `cos+jsin` that use the approximation method as `Lookup`.

This check requires Simulink and HDL Coder.

Results and Recommended Actions

To fix this warning for `sin`, `cos`, `sincos`, and `cos+jsin` blocks that have the LUT-based approximation method, use Sine HDL Optimized or Cosine HDL Optimized blocks instead.

Capabilities and Limitations

- Provides the list of the unsupported Trigonometric Function blocks that have the LUT-based approximation method in your Simulink model.
- HDL code generation is not supported for the `atan2` block in the LUT-based approximation method.

See Also

Sine HDL Optimized, Cosine HDL Optimized

Native Floating Point Checks Overview

These checks verify whether the model is compatible for HDL code generation in `Native Floating Point` mode. Native floating-point support in HDL Coder generates target-independent HDL code from your single-precision floating-point model. For more information, see “Generate Target-Independent HDL Code with Native Floating-Point” on page 14-111.

Use the checks in this folder to verify whether:

- You use the `Native Floating Point` mode when your design contains `single` data types.
- Your model uses `double` data types in `Native Floating Point` mode.
- Blocks in your model are supported for HDL code generation in `Native Floating Point` mode.
- Blocks in your model have a nonzero ulp error and a nonzero output latency.

Check for single datatypes in the model

Check ID: `com.mathworks.HDL.ModelChecker.runNFPSuggestionChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runNFPSuggestionChecks`

Check for `single` data types in the model.

Description

This check detects whether your Simulink model uses single data types and displays a warning if **Use Floating Point** is not selected.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator selects **Use Floating Point**. You can then rerun the check.

See Also

“Getting Started with HDL Coder Native Floating-Point Support” on page 14-88

“Check for double data types in the model” on page 37-27

Check for double data types in the model

Check ID: `com.mathworks.HDL.ModelChecker.runDoubleDatatypeChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runDoubleDatatypeChecks`

Check for `double` data types in the model.

Description

This check detects whether your Simulink model uses double data types and displays a warning if **Use Floating Point** is not selected.

Results and Recommended Actions

To fix this warning, enable **Use Floating Point** configuration setting for your model. You can then rerun the check.

See Also

“Getting Started with HDL Coder Native Floating-Point Support” on page 14-88

“Check for single datatypes in the model” on page 37-26

Check for Data Type Conversion blocks with incompatible settings

Check ID: `com.mathworks.HDL.ModelChecker.runNFPDTCChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runNFPDTCChecks`

Check conversion mode of Data Type Conversion blocks.

Description

This check displays a warning when Data Type Conversion blocks in your model convert from a floating-point data type to a fixed-point data type or vice-versa, and has **Input and output to have equal** parameter set to `Stored Integer (SI)`.

HDL Coder does not support Data Type Conversion blocks that use the `Stored Integer (SI)` conversion mode and convert between floating point and fixed point data types. During this conversion, the `Stored Integer (SI)` mode does not preserve the underlying stored integer bits of the floating point input signal.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator replaces Data Type Conversion blocks in `Stored Integer (SI)` mode with Float Typecast blocks.

Using the Float Typecast block, you can access the stored integer bits of a floating point input signal when converting between floating point and fixed point data types. The block works similar to the `typecast` function.

See Also

"Getting Started with HDL Coder Native Floating-Point Support" on page 14-88

Check for HDL Reciprocal block usage

Check ID: `com.mathworks.HDL.ModelChecker.runNFPHDLRecipChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runNFPHDLRecipChecks`

Check HDL Reciprocal blocks are not using floating point types

Description

This check displays a warning if your Simulink model contains HDL Reciprocal blocks that use floating-point data types. HDL Reciprocal blocks with floating point data types can have potential numerical mismatches with the Simulink simulation results, and can use more resources on the target hardware.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator replaces the HDL Reciprocal blocks with Math Reciprocal blocks.

See Also

HDL Reciprocal

Check for Relational Operator block usage

Check ID: `com.mathworks.HDL.ModelChecker.runNFPRelopChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runNFPRelopChecks`

Check Relational Operator blocks which use floating point types have boolean outputs.

Description

This check displays a warning if your Simulink model contains Relational Operator blocks that compare floating-point data types and produce a nonboolean output. By default, the **Output data type** of the block is `boolean`. If you specify a nonboolean output, the block implementation after HDL code generation is not optimal, and can increase the resource usage on the target hardware device.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator changes the **Output data type** of the block to `boolean`.

See Also

Relational Operator

Check for unsupported blocks with Native Floating Point

Check ID: `com.mathworks.HDL.ModelChecker.runNFPSupportedBlocksChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runNFPSupportedBlocksChecks`

Check for unsupported blocks with Native Floating Point.

Description

This check displays a failed message if your Simulink model contains blocks that use `single` data types, and are not supported for HDL code generation in the native floating-point mode.

Results and Recommended Actions

To fix this failed check, make sure that your design uses blocks that are supported for HDL code generation with `Native Floating Point`.

See Also

"Simulink Blocks Supported by Using Native Floating Point" on page 14-137

Check blocks with nonzero ULP error

Check ID: `com.mathworks.HDL.ModelChecker.runNFPULPErrorChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runNFPULPErrorChecks`

Check for blocks that have nonzero ULP error with Native Floating Point.

Description

This check detects blocks in your Simulink model that have a nonzero ULP error in native floating-point mode. When you run the check, the **Result** subpane displays hyperlinks to the blocks that have nonzero ULP error, and the ULP values.

Results and Recommended Actions

To fix this warning, look for instances of blocks that have nonzero ULP error and specify a **Tolerance Value** by setting **Floating point tolerance check based on** the `ULP error`. You can then rerun the check.

Note Fixing warnings that are reported by this check does not guarantee that your Simulink model has a zero ULP error. Make sure that you verify the ULP of your design by using multiple methods, such as by generating HDL code and test benches.

See Also

- “Numeric Considerations for Native Floating-Point” on page 14-92
- “ULP Considerations of Native Floating-Point Operators” on page 14-95

Industry standard checks overview

These checks verify whether your Simulink model conforms to the industry-standard rules. industry-standard rules recommend using certain HDL coding guidelines. When generating code, HDL Coder displays an HDL coding standard report that shows how well the generated code adheres to the industry-standard guidelines. For more information, see “HDL Coding Standards” on page 24-4

Use the checks in this folder to verify whether:

- Names in your design adhere to the standard naming conventions.
- Subsystem names, top-level subsystem and port names, and signal and port names have the recommended number of characters in length.
- The generated VHDL code from your design follows recommended guidelines. The guidelines recommend that the file name extension is `.vhd`, the architecture name is `rtl`, the package file postfix is `_pkg`, and that the generated code does not use generics at the top level.
- Clock settings adhere to the industry-standard guidelines, and whether clock, reset, and enable signals adhere to the naming conventions.

Check file extension

Check ID: `com.mathworks.HDL.ModelChecker.runFileExtensionChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runFileExtensionChecks`

Check file extensions based on the target language.

Description

This check detects whether the file extension is:

- `.vhd` when you generate code with VHDL as the target language.
- `.v` when you generate code with Verilog as the target language.
- `.sv` when you generate code with SystemVerilog as the target language.

This check corresponds to rule `1.A.A.1` of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the file name extension.

See Also

Rule 1.A.A.1 of “Basic Coding Practices” on page 24-7.

Check naming conventions

Check ID: `com.mathworks.HDL.ModelChecker.runNameConventionChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runNameConventionChecks`

Check standard keywords used by EDA tools.

Description

This check detects whether names in the design are adhering to the standard naming convention and not using names starting with `VDD`, `VSS`, and so on. This check corresponds to rule 1.A.A.4 of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the names by replacing the reserved keywords with `rsvd` to adhere to the industry-standard rule.

See Also

Rule 1.A.A.4 of “Basic Coding Practices” on page 24-7.

Check top-level subsystem/port names

Check ID: `com.mathworks.HDL.ModelChecker.runToplevelNameChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runToplevelNameChecks`

Check top-level module/entity and port names.

Description

This check verifies whether top-level module/entity and port names are of the same case and less than or equal to 16 characters in length. This check corresponds to rule 1.A.A.9 of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the top-level module/entity and port names to be of the same case and less than or equal to 16 characters. Names longer than 16 characters are truncated to fit within 16 characters in length.

See Also

Rule 1.A.A.9 of "Basic Coding Practices" on page 24-7.

Check module/entity names

Check ID: `com.mathworks.HDL.ModelChecker.runSubsystemNameChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runSubsystemNameChecks`

Check module/entity names.

Description

This check verifies whether subsystems in your model have names between 2 and 32 characters in length. This check corresponds to rule 1.A.B.1 of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the Subsystem names such that the module/entity and port names are between 2 and 32 characters.

See Also

Rule 1.A.B.1 of "Basic Coding Practices" on page 24-7.

Check signal and port names

Check ID: `com.mathworks.HDL.ModelChecker.runPortSignalNameChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runPortSignalNameChecks`

Check signal and port name lengths.

Description

This check verifies whether ports and signals from the blocks have names between `2` and `40` characters in length. This check corresponds to rule 1.A.C.3 of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the signal and port names to be between `2` and `40` characters in length.

See Also

Rule 1.A.C.3 of “Basic Coding Practices” on page 24-7.

Check package file names

Check ID: `com.mathworks.HDL.ModelChecker.runPackageNameChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runPackageNameChecks`

Check file name containing packages.

Description

This check verifies whether the package file postfix is `_pac` when you generate code with VHDL as the target language. By default, the generated package file postfix is `_pkg`. This check corresponds to rule 1.A.D.1 of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the package file name to `_pac`.

See Also

Rule 1.A.D.1 of “Basic Coding Practices” on page 24-7.

Check generics

Check ID: `com.mathworks.HDL.ModelChecker.runGenericChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runGenericChecks`

Check generics at top level subsystem.

Description

This check verifies whether you have generics at the top-level subsystem. If your design uses mask parameters and has the `MaskParameterAsGeneric` setting enabled, then the top-level subsystem can have generics. This check corresponds to rule 1.A.D.9 of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator disables the `MaskParameterAsGeneric` setting so that there are no generics at the top-level subsystem.

See Also

Rule 1.A.D.9 of “Basic Coding Practices” on page 24-7.

Check clock, reset, and enable signals

Check ID: `com.mathworks.HDL.ModelChecker.runClockResetEnableChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runClockResetEnableChecks`

Check naming convention for clock, reset, and enable signals.

Description

This check verifies whether clock, reset, and clock enable signals follow the recommended naming convention. Clock signal names must contain `clk` or `ck`, reset signal names must contain `rstx`, `resetx`, `rst_n`, `reset_n`, `rst_x`, or `reset_x`, and clock enable signal names must contain `en`. This check corresponds to rule 1.A.E.2 of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the clock, reset, and enable signals to adhere to the naming conventions.

See Also

Rule 1.A.E.2 of “Basic Coding Practices” on page 24-7.

Check architecture name

Check ID: `com.mathworks.HDL.ModelChecker.runArchitectureNameChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runArchitectureNameChecks`

Check VHDL architecture name in the generated HDL code.

Description

This check verifies whether the architecture name is `rtl` when you generate code with VHDL as the target language. This check corresponds to rule 1.A.F.1 of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the `VHDLArchitectureName` setting to `rtl` to adhere to the industry-standard rule.

See Also

Rule 1.A.F.1 of “Basic Coding Practices” on page 24-7.

Check entity and architecture

Check ID: `com.mathworks.HDL.ModelChecker.runSplitEntityArchitectureChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runSplitEntityArchitectureChecks`

Check whether the VHDL entity and architecture are described in the same file.

Description

This check detects when you have the entity and architecture descriptions in separate files when you generate code with VHDL as the target language. The entity and architecture descriptions can be in separate files if you enable the `SplitEntityArch` setting. This check corresponds to rule 1.A.F.4 of the industry-standard rules.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator disables the `SplitEntityArch` setting so that the entity and architecture descriptions are in the same file.

See Also

Rule 1.A.F.4 of "Basic Coding Practices" on page 24-7.

Check clock settings

Check ID: `com.mathworks.HDL.ModelChecker.runClockChecks`

Check ID: `com.mathworks.HDL.ModelAdvisor.runClockChecks`

Check constraints on clock signals.

Description

This check detects multiple constraints on clock signals that correspond to these industry-standard rules:

- Rule 1.B.A.1: Design should have only a single clock and use only one edge of the clock. This rule may be violated if you have the `ClockInputs` property set to `Multiple`.
- Rule 1.D.C.6: Do not use flip-flops with inverted edges.
- Rule 1.D.D.2: One hierarchical level should have a single clock only. This rule can be violated if you set `ClockInputs` to `Multiple`, or your design uses trigger signals and enabling `TriggerAsClock` can result in clock signals at various levels in the hierarchy.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator:

- Updates the `ClockInputs` property to `Single`.
- Disables the `TriggerAsClock` setting.

See Also

Rules 1.B.A.1, 1.D.C.6, and 1.D.D.2 of “Basic Coding Practices” on page 24-7.

Using the HDL Code Advisor

- “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 38-2
- “Run Model Advisor Checks for HDL Coder” on page 38-6
- “HDL Code Advisor Checks” on page 38-9

Check HDL Compatibility of Simulink Model Using HDL Code Advisor

In this section...

"Open the HDL Code Advisor" on page 38-2
"Run Checks In the HDL Code Advisor" on page 38-3
"Fix HDL Code Advisor Warnings or Failures" on page 38-3
"View and Save HDL Code Advisor Reports" on page 38-4

The HDL Code Advisor verifies and updates your Simulink model or subsystem for compatibility with HDL code generation. The Model Checker checks for model configuration settings, ports and subsystem settings, block settings, support for native floating point, and conformance to the industry-standard rules. The Code Advisor produces a report that lists suboptimal conditions or settings, and then proposes better model configuration settings.

The HDL Code Advisor has these caveats:

- If you reference one model in another by using a Model block, the HDL Code Advisor checks the model configurations or settings of the parent model. To check whether the referenced model is compatible with HDL code generation, open the HDL Code Advisor for the referenced model, and then run the checks.
- If you run the checks on masked library blocks in your Simulink model, the Code Advisor cannot verify whether the blocks inside the library blocks have HDL-compatible settings.
- When you apply Model Advisor checks to your model, it increases the likelihood that your model does not violate certain modeling standards or guidelines. However, it does not guarantee that the design is ready for HDL code generation. Make sure that you verify the design by using multiple methods for HDL code generation readiness.

Open the HDL Code Advisor

To open the HDL Code Advisor:

- From the UI, in the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the DUT Subsystem and then click **HDL Code Advisor**.
- To run the checks for the Subsystem you want to analyze, right-click that Subsystem, and in the context menu, select **HDL Code > HDL Code Advisor**.
- At the command line, enter `hdlcodeadvisor('system')`. *system* is a handle or name of the model or subsystem that you want to check. For more information, see `hdlcodeadvisor`.

In the HDL Code Advisor, the left pane lists the folders in the hierarchy. Each folder represents a group or category of related checks. Expanding the folders shows available checks in each folder. From the left pane, you can select a folder or an individual check. The HDL Code Advisor displays information about the selected folder or check in the right pane. The content of the right pane depends on the selected folder or check. The right pane has a **Result** subpane that contains a display area for status messages and other task results.

To learn more about each individual check, right-click that check, and select **What's This?**.

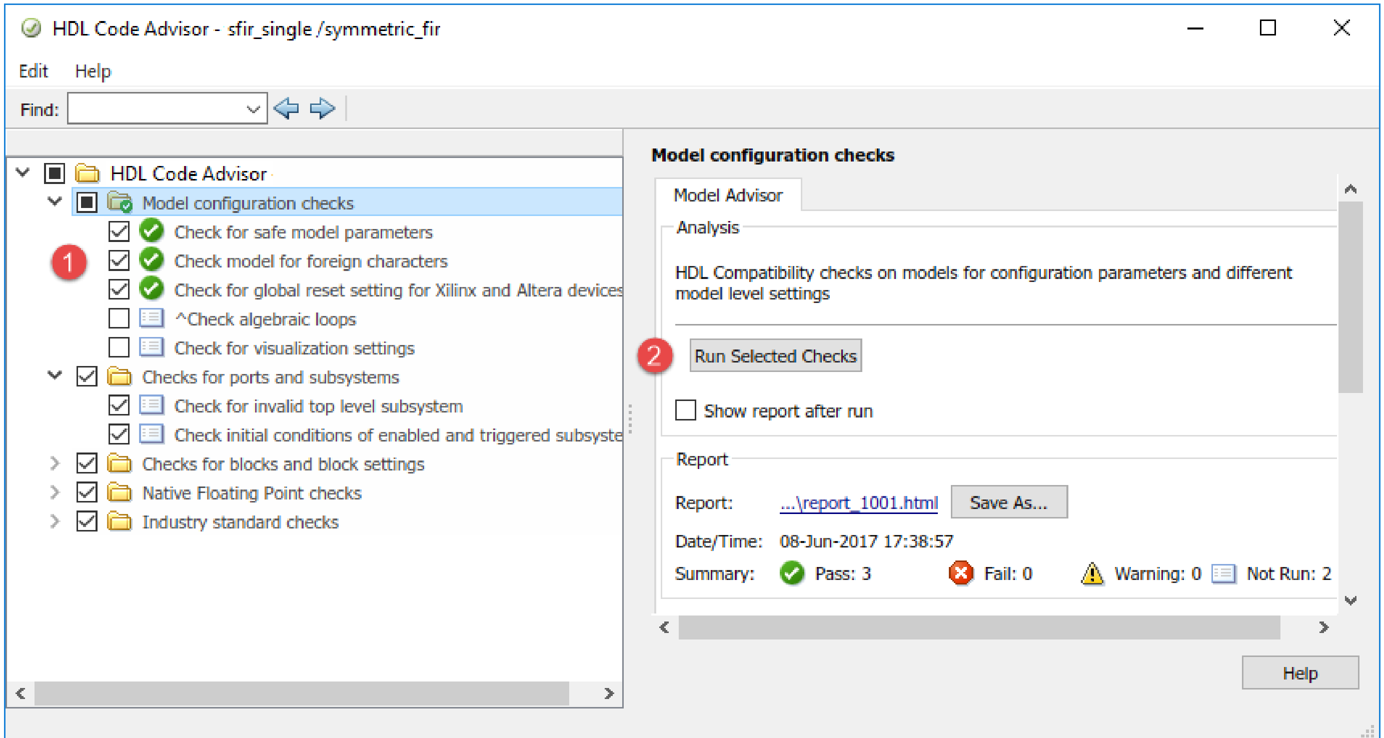
Run Checks In the HDL Code Advisor

In the HDL Code Advisor window, you can run individual checks or a group of checks. To run a check, **Select** that check and then click **Run This Check**. For example, to run the **Check for safe model parameters**, select the check box, and then click **Run This Check**.

In the HDL Code Advisor window, you can run a group of checks within a folder.

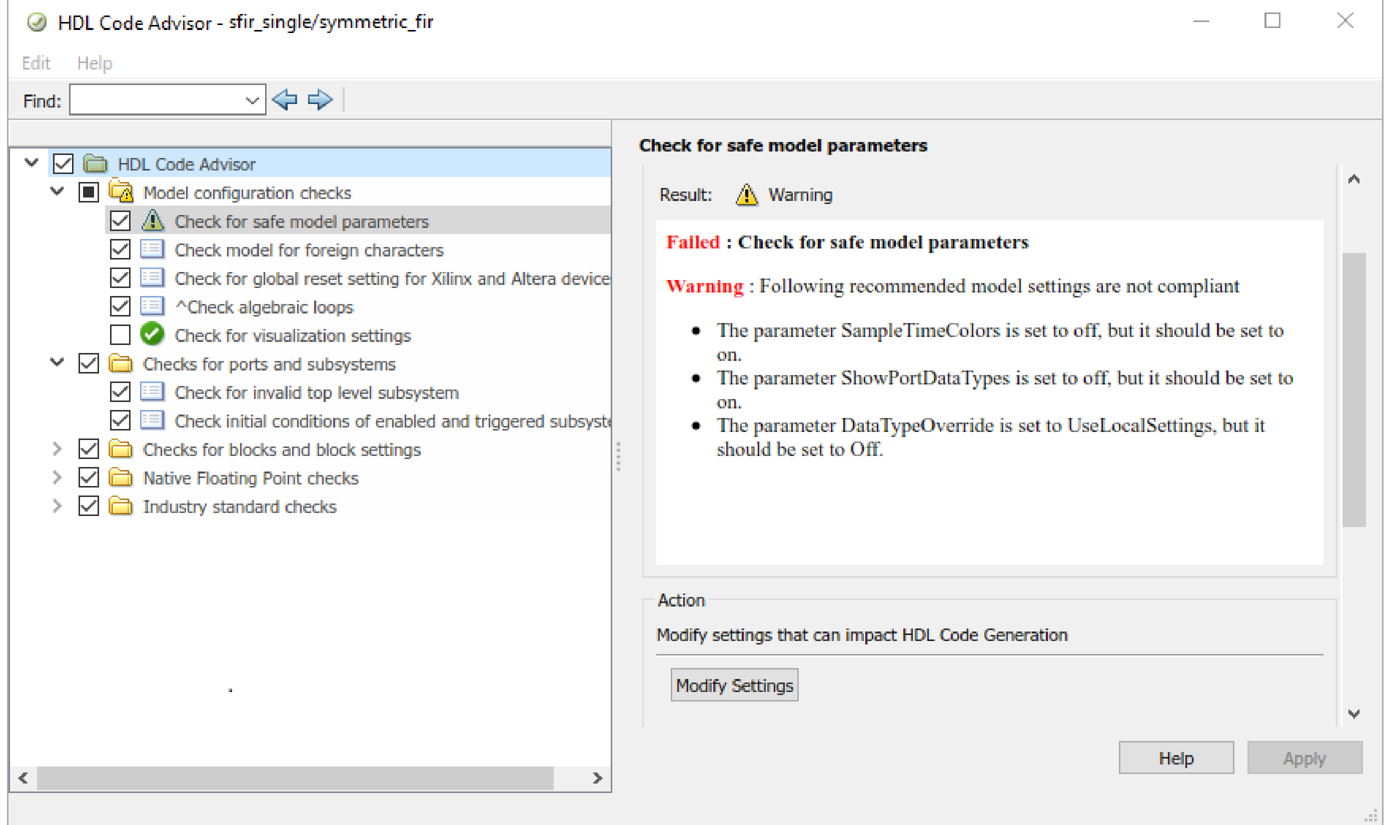
1. Select the checks that you want to run.
2. Select the folder that contains these checks and then click **Run Selected Checks**.

This example shows how to run selected checks in the **Model configuration checks** folder.



Fix HDL Code Advisor Warnings or Failures

In the HDL Code Advisor, if a check fails, the right pane shows the warning or failure information in a **Result** subpane. The **Result** subpane displays model settings that are not compliant. For some tasks, use the **Action** subpane to apply the Code Advisor recommended settings. This example displays the incorrect model settings that caused the **Check for safe model parameters** to fail.

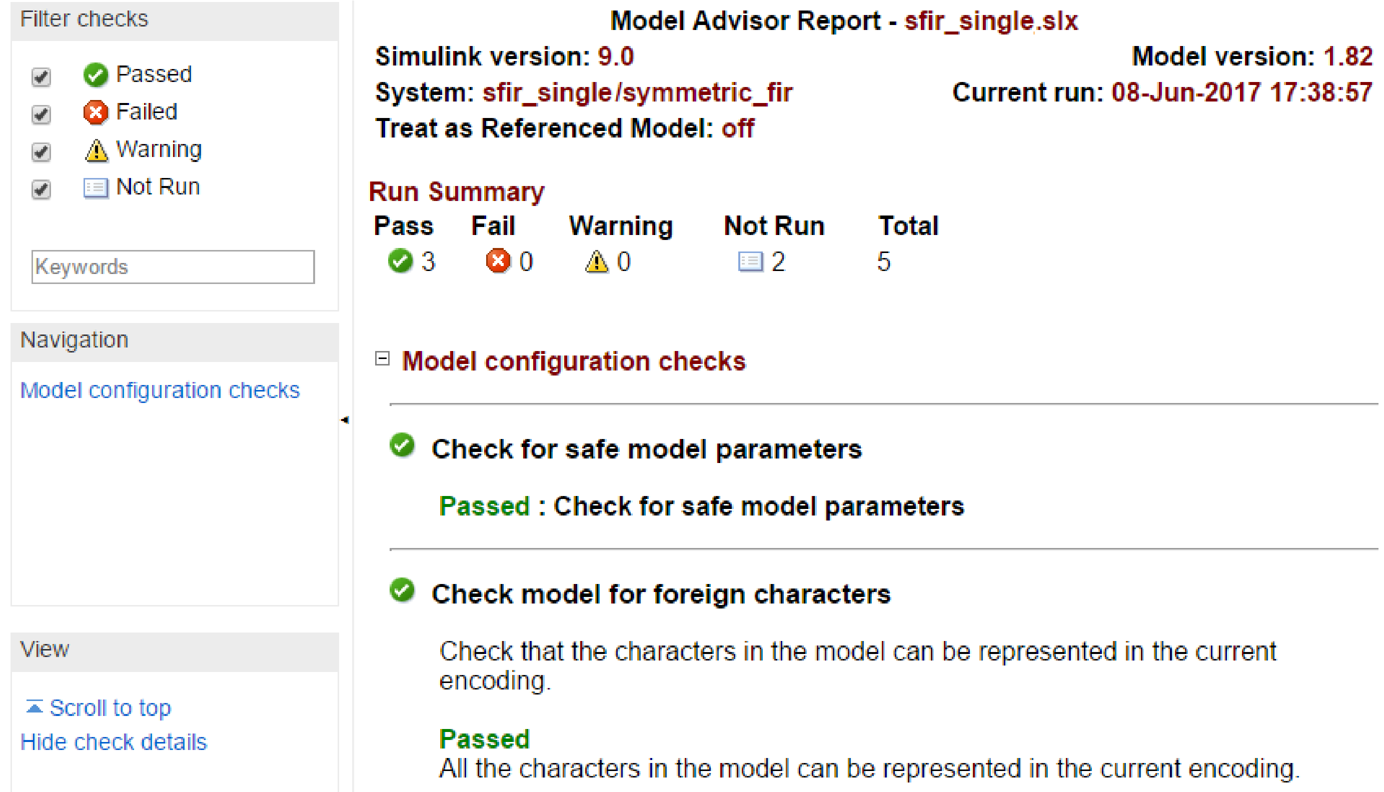


To apply the corresponding model configuration settings that the code generator reported in the **Result** subpane, click the **Modify Settings** button. After you click **Modify Settings**, the **Result** subpane reports the changes that were applied. You can now run this check.

View and Save HDL Code Advisor Reports

When you run checks in the HDL Code Advisor, HDL Coder generates an HTML report of the check results. Each folder in the HDL Code Advisor contains a report for the checks within that folder and its subfolders. To access reports, select a folder such as **Model configuration checks**, and in the **Report** subpane, click **Save As**. If you rerun the HDL Code Advisor, the report is updated in the working folder, not in the save location.

This report shows typical results for a run of the **Model configuration checks** folder.



As you run the checks, the HDL Code Advisor updates the reports with the latest information for each check in the folder. When you run the checks at different times, timestamps appear at the top right of the report to indicate when checks have been run. Checks that occurred during previous runs have a timestamp following the check name. You can filter checks in the report such that tasks that are **Not Run** do not appear or show tasks that **Passed**, and so on. To view the report for a folder each time the tasks for the folder have been run, select **Show report after run**.

See Also

More About

- “HDL Code Advisor Checks” on page 38-9

Run Model Advisor Checks for HDL Coder

The Model Advisor checks a model or subsystem for conditions and configuration settings that can result in inaccurate or inefficient simulation. The Model Advisor produces a report that lists the suboptimal conditions or settings that it finds, and proposes better model configuration settings where possible. HDL Coder integrates the checks in the HDL Code Advisor with the Model Advisor.

Open the Model Advisor Checks

You can open the Model Advisor in either of these ways:

- In the **Modeling** tab, select **Model Advisor**. In the System Selector dialog box, select the model or Subsystem that you want to analyze, and click **OK**.
- To run the model advisor checks for the Subsystem that you want to analyze, right-click that Subsystem, and select **Model Advisor > Open Model Advisor**.
- At the command line, enter `modeladvisor('system')`. *system* is the name of the model or Subsystem that you want to analyze. For more information, see `modeladvisor`.

When you open the Model Advisor in Simulink, you see the checks in the **HDL Coder** subfolder of the **By Product** folder. Each subfolder in the **HDL Coder** folder represents a group or category of related checks. Expanding the folders display available checks in each folder. From the left pane, you can select a folder or an individual check. The Model Advisor displays information about the selected folder or check in the right pane. The content of the right pane depends on the selected folder or check. The right pane has a **Result** subpane that contains a display area for status messages and other task results.

To learn more about each individual check, right-click that check, and select **What's This?**.

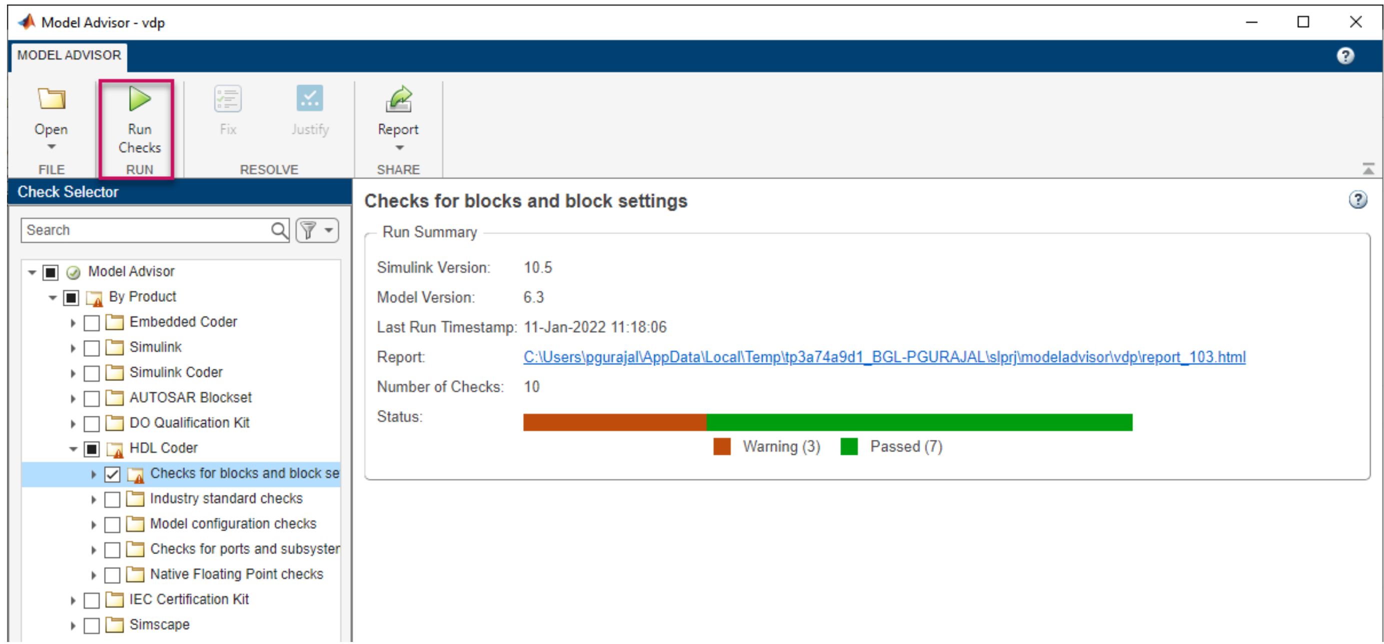
Run Checks in the Model Advisor

In the Model Advisor window, you can run individual checks or a group of checks. To run a check, **Select** that check, and then click **Run This Check**.

To run a group of checks within a folder:

1. Select the checks that you want to run.
2. Select the folder that contains these checks and then click **Run Checks**

For example, to run the checks in the **Checks for blocks and block settings** folder, select the folder, and then click **Run Checks**.

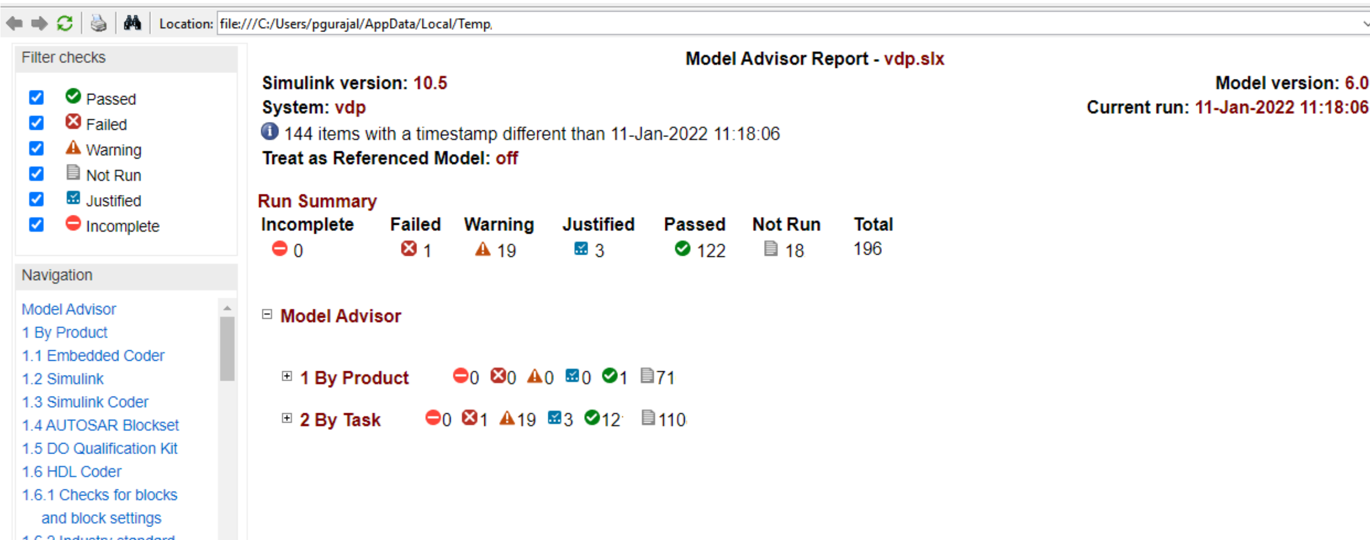


You can also right-click the folder **Checks for blocks and block settings**, and select **Run Selected Checks** in the Model Advisor.

Display Check Results in the Model Advisor Report

To display an HTML report of the check results, click **Report** from the toolstrip. You can use the **Report** drop-down to change the report format to **PDF** or **WORD**.

This report shows typical results for a run of the **Checks for blocks and block settings** folder.



The report displays a run summary of the checks in the folder that you generated the report for. As you run the checks, the Model Advisor updates the reports with the latest information for each check in the folder. When you run the checks at different times, timestamps appear at the top right of the

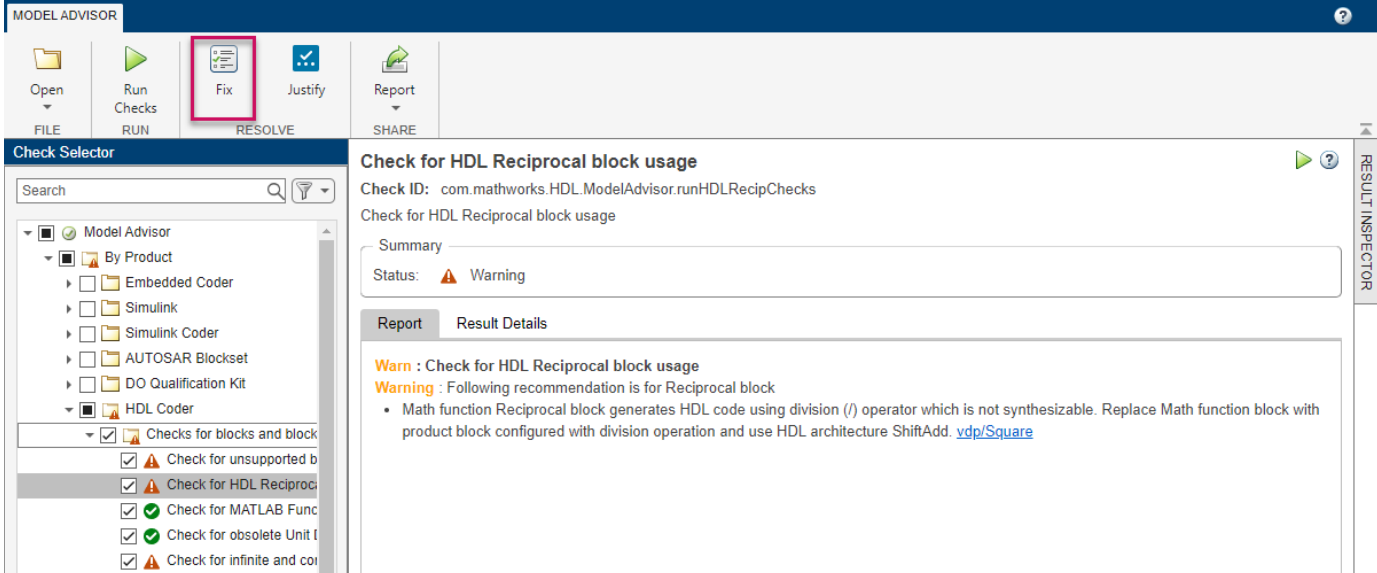
report to indicate when checks have been run. Checks that occurred during previous runs have a timestamp following the check name. You can filter checks in the report to show checks that display a **Warning**, or show checks that **Passed**, and so on.

Fix Warnings or Failures

When a model or referenced model has a suboptimal condition, checks can fail. After you run a Model Advisor analysis, indicates checks that have warnings. A warning result is informational. You can fix the reported issue or move on to the next task.

To fix warnings or failures, in the **Result** sub-pane, review the recommended actions to make changes to your model. When you fix a warning or failure, to verify that the check passes, re-run the check.

Some checks can use the **Fix** option from the toolstrip. Note that this option is available for selected checks that display violations with input parameters. This example displays the incorrect Reciprocal block settings that caused the check to display a warning.



When you select **Fix**, the **Action Report** window shows the changes that were applied. To verify that the check passes, rerun the check. If you use the Model Advisor dashboard, you see that the analysis is faster when you rerun the check because the Model Advisor does not reload the checks before executing them.

See Also

More About

- “HDL Code Advisor Checks” on page 38-9
- “Run Model Advisor Checks”

HDL Code Advisor Checks

In this section...

"Model configuration checks" on page 38-10

"Checks for ports and subsystems" on page 38-10

"Checks for blocks and block settings" on page 38-10

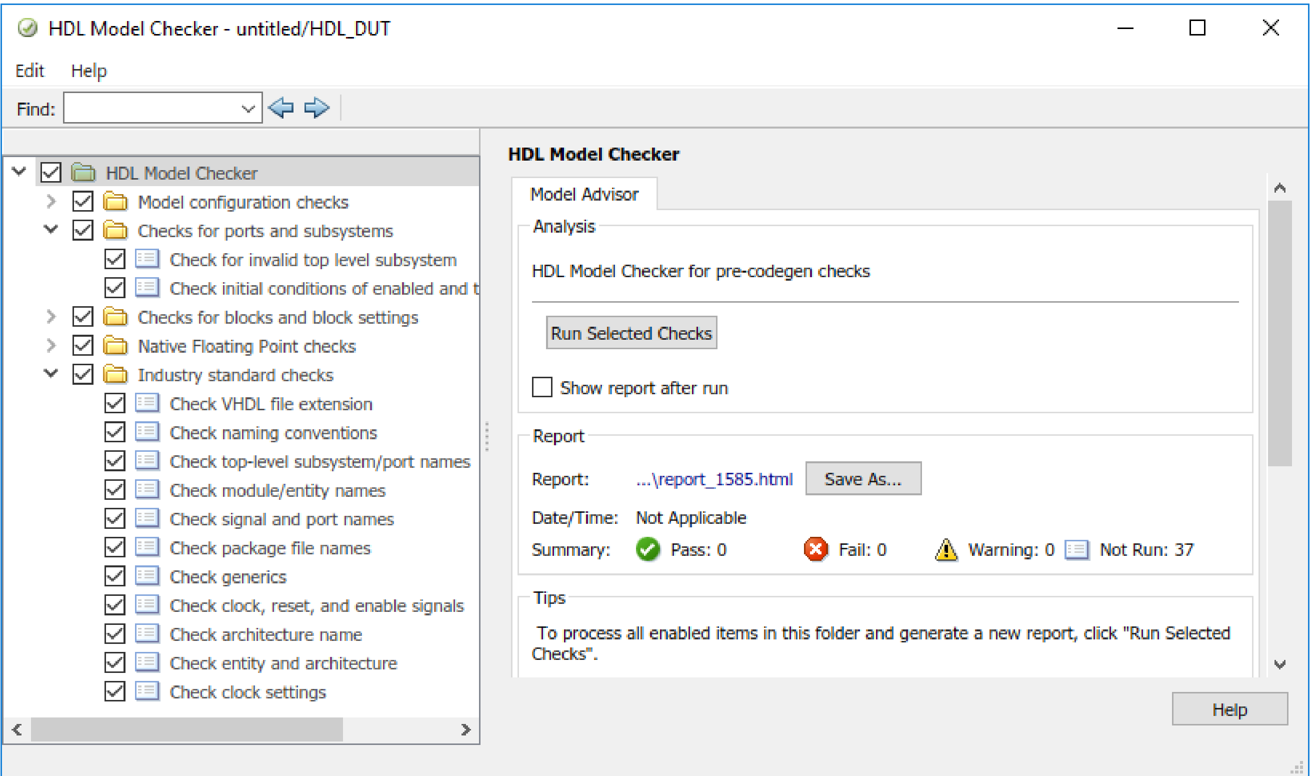
"Native Floating Point checks" on page 38-11

"Industry standard checks" on page 38-12

The HDL Code Advisor and the Model Advisor checks in HDL Coder verify and update your Simulink model or subsystem for compatibility with HDL code generation. The Code Advisor has checks for:

- Model configuration settings
- Ports and Subsystem settings
- Blocks and block settings
- Native Floating Point support
- Industry standard guidelines

When you run a check, the Code Advisor displays the result as a pass, a warning or a failure. You can fix warnings or failures by using the Model Advisor recommended settings.



Model configuration checks

Use the checks in this folder to prepare your model for compatibility with HDL code generation. This folder contains checks that verify whether model parameters are HDL-compatible, whether your design contains algebraic loops, and so on.

Check Name	Description
"Check for model parameters suited for HDL code generation" on page 37-5	Check for model parameters set up for HDL code generation.
"Check for global reset setting for Xilinx and Altera devices" on page 37-7	Check asynchronous reset setting for Altera devices and synchronous reset setting for Xilinx devices.
"Check inline configurations setting" on page 37-8	Check whether you have `InlineConfigurations` enabled.
"Check algebraic loops" on page 37-9	Check model for algebraic loops.
"Check for visualization settings" on page 37-10	Check model for display settings: port data types and sample time color coding.
"Check delay balancing setting" on page 37-11	Check Balance Delays is enabled.

Checks for ports and subsystems

This folder contains checks that verify whether ports and subsystems in your model have settings that are compatible for HDL code generation. The checks include whether you have a valid top-level DUT Subsystem and whether you have specified an initial condition for Enabled Subsystem and Triggered Subsystem blocks.

Check Name	Description
"Check for invalid top level subsystem" on page 37-13	Check for subsystems that cannot be at the top level for HDL code generation.

Checks for blocks and block settings

These checks verify whether blocks in your model are supported for HDL code generation, and whether the supported blocks have HDL-compatible settings. The checks include whether source blocks in your model have a continuous sample time and whether Stateflow Charts and MATLAB Function blocks have HDL-compatible settings, and so on.

Check Name	Description
"Check for infinite and continuous sample time sources" on page 37-15	Check source blocks with continuous sample time.
"Check for unsupported blocks" on page 37-16	Check for unsupported blocks for HDL code generation.
"Check for large matrix operations" on page 37-17	Check for large matrix operations.
"Identify unconnected lines, input ports, and output ports"	Check for unconnected lines or ports.

Check Name	Description
"Check for MATLAB Function block settings" on page 37-18	Check HDL compatible settings for MATLAB Function blocks.
"Check for Stateflow chart settings" on page 37-19	Check HDL compatible settings for Stateflow Chart blocks.
"Check for blocks that have nonzero output latency" on page 37-21	Check for blocks that have nonzero output latency with fixed point and native floating point.
"Check for unsupported storage class for signal objects" on page 37-22	Check whether signal object storage class is `'ExportedGlobal'` or `'ImportedExtern'` or `'ImportedExternPointer'`.
"Check for Trigonometric Function block for LUT-based approximation method" on page 37-24	Check for Trigonometric Function blocks in a model that use the look up table (LUT) based approximation method.
"Check for HDL Reciprocal block usage" on page 37-23	Check if the model uses HDL Reciprocal blocks.

Note If you use the Model Advisor, you see the "Identify unconnected lines, input ports, and output ports" in the **Simulink** folder.

Native Floating Point checks

These checks verify whether the model is compatible for HDL code generation in `Native Floating Point` mode. The checks include whether the blocks in your Simulink model are supported for HDL code generation with `Native Floating Point`, and whether the model uses single data types, and so on. Native floating-point support in HDL Coder generates target-independent HDL code from your single-precision floating-point model. For more information, see "Generate Target-Independent HDL Code with Native Floating-Point" on page 14-111.

Check Name	Description
"Check for single datatypes in the model" on page 37-26	Check for `single` data types in the model.
"Check for double data types in the model" on page 37-27	Check for `double` data types in the model.
"Check for Data Type Conversion blocks with incompatible settings" on page 37-28	Check conversion mode of Data Type Conversion blocks.
"Check for HDL Reciprocal block usage" on page 37-29	Check HDL Reciprocal blocks are not using floating point types.
"Check for Relational Operator block usage" on page 37-30	Check Relational Operator blocks which use floating point types have boolean outputs.
"Check for unsupported blocks with Native Floating Point" on page 37-31	Check for unsupported blocks with native floating-point.
"Check blocks with nonzero ULP error" on page 37-32	Check for blocks that have nonzero ulp error with native floating-point.

Industry standard checks

These checks verify whether your Simulink model conforms to the industry-standard rules. industry-standard rules recommend using certain HDL coding guidelines. When generating code, HDL Coder displays an HDL coding standard report that shows how well the generated code adheres to the industry-standard guidelines.

Check Name	Description
"Check file extension" on page 37-34	Check file extensions of VHDL files containing entities.
"Check naming conventions" on page 37-35	Check standard keywords used by EDA tools.
"Check top-level subsystem/port names" on page 37-36	Check top-level module/entity and port names.
"Check module/entity names" on page 37-37	Check module/entity names.
"Check signal and port names" on page 37-38	Check signal and port name lengths.
"Check package file names" on page 37-39	Check file name containing packages.
"Check generics" on page 37-40	Check generics at top-level subsystem.
"Check clock, reset, and enable signals" on page 37-41	Check naming convention for clock, reset, and enable signals.
"Check architecture name" on page 37-42	Check VHDL architecture name in the generated HDL code.
"Check entity and architecture" on page 37-43	Check whether the VHDL entity and architecture are described in the same file.
"Check clock settings" on page 37-44	Check constraints on clock signals.

For more information, see:

- "HDL Coding Standards" on page 24-4
- "Basic Coding Practices" on page 24-7
- "RTL Description Rules and Checks" on page 24-18
- "RTL Design Methodology Guidelines" on page 24-44

See Also

`hdlcodeadvisor`

More About

- "Check HDL Compatibility of Simulink Model Using HDL Code Advisor" on page 38-2
- "Run Model Advisor Checks for HDL Coder" on page 38-6

Hardware-Software Codesign

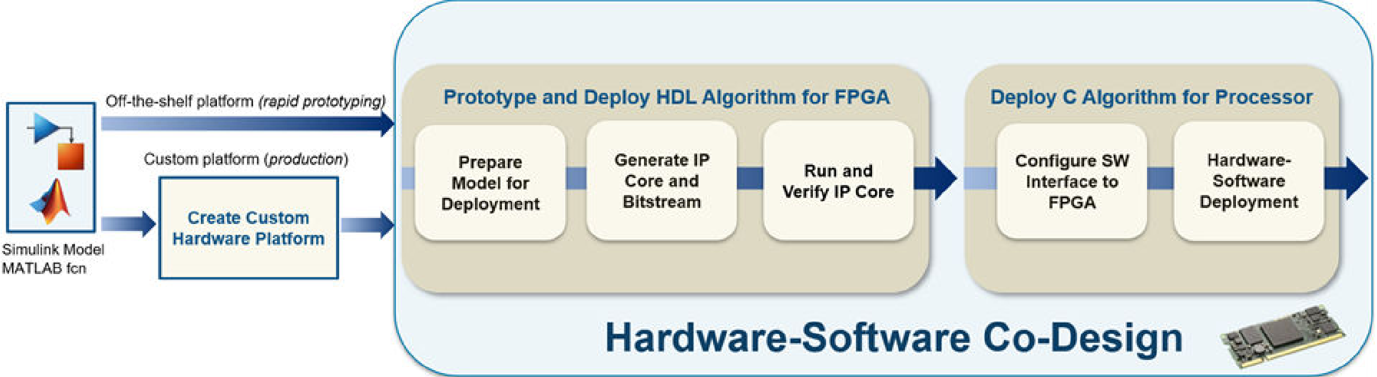
Hardware-Software Co-Design Basics

- “Targeting FPGA & SoC Hardware Overview” on page 39-3
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- “Speedgoat FPGA Support with HDL Workflow Advisor” on page 39-15
- “Custom IP Core Generation” on page 39-17
- “Custom IP Core Report” on page 39-20
- “Comparison of IP Core Generation Techniques” on page 39-27
- “Comparison of IP Core Deployment and Verification Techniques” on page 39-30
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 39-33
- “Processor and FPGA Synchronization” on page 39-38
- “Synchronization of Global Reset Signal to IP Core Clock Domain” on page 39-40
- “IP Caching for Faster Reference Design Synthesis” on page 39-44
- “Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows” on page 39-49
- “Define Multiple AXI Master Interfaces in Reference Designs to Access DUT AXI4 Slave Interface” on page 39-60
- “Program Target FPGA Boards or SoC Devices” on page 39-64
- “Generate and Manage FPGA I/O Host Interface Scripts” on page 39-68
- “Choose a Method to Interact with IP Cores on Target Hardware” on page 39-74
- “Generate Software Interface Model to Probe and Rapidly Prototype HDL IP Core” on page 39-84
- “Use FPGA I/O to Rapidly Prototype HDL IP Core” on page 39-90
- “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98
- “Getting Started with Targeting Zynq UltraScale+ MPSoC Platform” on page 39-112
- “Getting Started with Targeting Intel SoC Devices” on page 39-132
- “Getting Started with Targeting Intel Quartus Pro Based Devices” on page 39-147
- “Integrate HDL IP Core with Microchip PolarFire SoC Icicle Kit Reference Design” on page 39-161
- “Save Target Hardware Settings in Model” on page 39-177
- “Generate IP Core from MATLAB for Blinking LEDs on FPGA Board” on page 39-180
- “Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow” on page 39-190
- “Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow” on page 39-201
- “IP Core Generation Workflow with a MicroBlaze processor: Xilinx Kintex-7 KC705” on page 39-211
- “Map Bus Data Types to AXI4 Slave Interfaces” on page 39-234
- “Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands” on page 39-241

- “Author a Xilinx Zynq Linux Image for a Custom Zynq Board by Using MathWorks Buildroot” on page 39-253
- “Generate Board-Independent HDL IP Core for Microchip Platforms” on page 39-260
- “Getting Started with Targeting Xilinx Versal Adaptive SoC Platform” on page 39-265

Targeting FPGA & SoC Hardware Overview

Target an FPGA or SoC device by following the steps that design and deploy your algorithm specifically on hardware. You start with a Simulink model or MATLAB function, choose a hardware platform to target, and follow the Hardware-Software Co-Design workflow. Explore the best ways to partition and deploy your design by iterating through these steps.



Choose your target platform. After modeling your design in Simulink or as a MATLAB function, decide what target platform you want to deploy your design to. Consider:

- Which type of device do you want to target: standalone FPGA, SoC device, or a platform that has a separate FPGA and processor?

Device Type	Recommended Workflow
Standalone FPGA	If you are deploying only to an FPGA, use only the first part of the Hardware-Software Co-Design Workflow: Prototype and Deploy HDL Algorithm for FPGA.
SoC Device	To deploy a model to an SoC device, you need to partition the design to contain a hardware portion for the FPGA, and an embedded software portion for the processor, referred to as a hardware-software model. The FPGA requires a bitstream to alter its physical connections, the hardware part of the model, and the processor requires a new set of instructions in the form of an executable, the software part of the model.
Simulink Real-Time FPGA I/O Modules	This type of platform refers to a Speedgoat target machine, which acts as a processor, and an FPGA IO module. You can program both the FPGA and processor using the targeting FPGA & SoC hardware steps. For more information, see “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 40-145.

- What stage of development are you in: rapid prototyping or production?

  - For rapid prototyping, save development time by selecting a pre-existing target device that HDL Coder supports. You can then immediately get started using the Hardware-Software Co-Design workflow. See “HDL Coder Supported Hardware”.
  - For production, or rapid prototyping with an unsupported device, create a custom hardware platform first, and then follow the Hardware-Software Co-Design workflow. See “Create Custom Hardware Platform” on page 39-7.

Hardware-Software Co-Design

The Hardware-Software Co-Design workflow is broken into two stages:

- Prototype and Deploy HDL Algorithm for an FPGA

  Prepare your model for deployment on target hardware. HDL Coder generates an IP core and bitstream containing the generated HDL Code from your design to deploy to the FPGA of your device. You can then run and verify that the IP core is working on your target hardware. This stage can be used to target a standalone FPGA device, an SoC device, or platform that has a separate FPGA and processor.
- Deploy C Algorithm for Processor

  Configure the hardware-software interface by configuring the connection between the FPGA and processor generating C code for your processor.

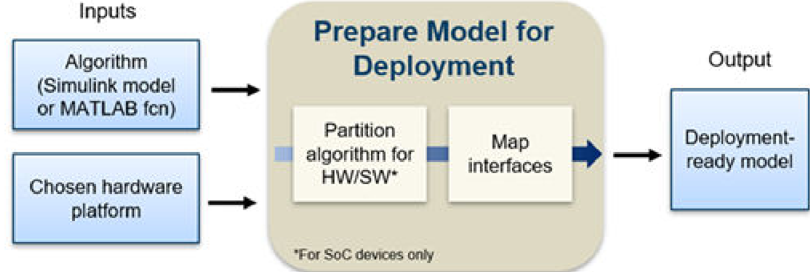
  Deploy your design that has been partitioned into hardware and embedded software components to run on a platform containing an FPGA and an embedded processor, such as an SoC device. The design consists of the DUT algorithm for which you generate an IP core and a bitstream containing your HDL code, and software components for which you generate embedded code to run on the processor. You use the generated software interface model and IP core that includes hardware interface components, such as AXI interfaces, to interface between the hardware and software components.

Prototype and Deploy HDL Algorithm for an FPGA

Prepare Model for Deployment

For the model and design of your algorithm to be deployable on your target device, these are the high-level tasks:

1. Partition your design for hardware and software components of your target hardware, if you are using a platform containing a processor and an FPGA, such as an SoC device. You can partition your design to generate the hardware that targets the FPGA and the software that runs on the embedded processor.
2. Map the inputs and outputs of your model to hardware interfaces, such as AXI4 interfaces, push buttons, and LEDs. The generated design can then communicate with the rest of the hardware system when it is deployed. For more information, see “Target Platform Interfaces” on page 39-17.

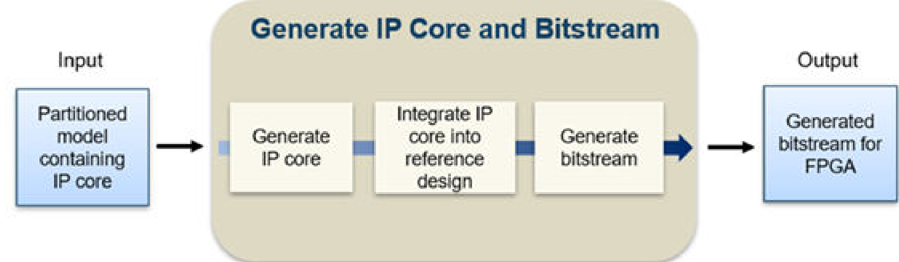


For an example on preparing a model for deployment, see “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98.

Generate IP Core and Bitstream

Once your design is prepared for deployment, you can use the HDL Workflow Advisor to:

- Generate a generic board-independent Xilinx or Intel HDL IP core. The IP core is a shareable and reusable HDL component that implements a specific function, typically an algorithm. An IP core consists of IP core definition files, HDL code generated for your algorithm, a C header file containing the register address map, and the IP core report. For an example, see “Generate Board-Independent HDL IP Core from Simulink Model” on page 39-33.
- Integrate the IP core into a reference design to target standalone FPGA boards or SoC platforms by using Xilinx Vivado IP integrator or Intel Qsys. The reference design contains elements that the Intel or Xilinx software requires to deploy your design to the SoC platform, except for the custom IP core and embedded software that you generate from the hardware-software model. The reference design acts as a platform to build your algorithm on top of and abstracts your hardware platform so you can focus on your algorithm development. Once you have completed your algorithm development, HDL Coder enables you to package the algorithm as an IP core and fit it into the reference design.
- Generate a bitstream that contains your IP core for deployment on your target FPGA.



For an example on generating an IP core and bitstream, see “Generate IP Core from MATLAB for Blinking LEDs on FPGA Board” on page 39-180.

Run and Verify IP Core on Target Hardware

Run and verify the generated bitstream from your IP core design on your target hardware:

1. Prepare your target hardware by connecting it to the host machine. If you are using a platform containing an FPGA and processor, such as an SoC device, download a Linux image for the processor. For more information, see “Guided Hardware Setup” for Xilinx Zynq Platforms or “Guided Hardware Setup” for Intel SoC Devices.
2. Establish a JTAG or Ethernet connection from your host machine to the target hardware.
3. Program your target device.
4. Prototype, debug, and verify your design on your target hardware by using one of these prototyping approaches:

   - FPGA I/O. You can use the host interface model or host interface script generated by the HDL Workflow Advisor to communicate between your host PC, running MATLAB and Simulink, and the generated IP core deployed on your FPGA.
   - AXI Manager
   - FPGA Data Capture

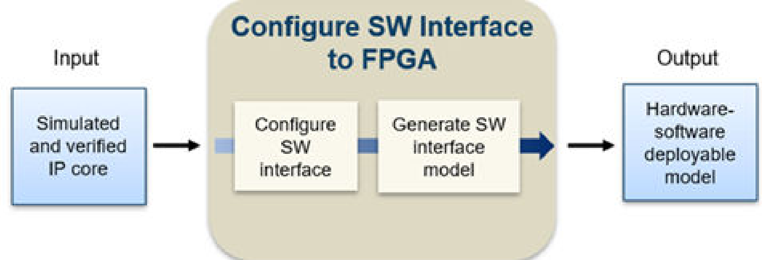
- FPGA-in-the-Loop (FIL)
- Software Interface Model Generation, see “Configure Software Interface to FPGA”.

For an example, see “Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands” on page 39-241.

Deploy C Algorithm for Processor

Configure Software Interface to FPGA

To target a complete hardware-software system, HDL Workflow Advisor generates a software interface model from your original Simulink model. The software interface model can be used for deployment to the on-board processor of your target platform. This model contains properly configured device drivers for the processor on-board your SoC device or target platform. If you are targeting an SoC device, this model is the second model required for the hardware-software design. The first model is the original model with your HDL algorithm to target the on-board FPGA. This software interface model targets your processor and communicates between the software deployed to the processor and the generated IP core deployed to the FPGA.

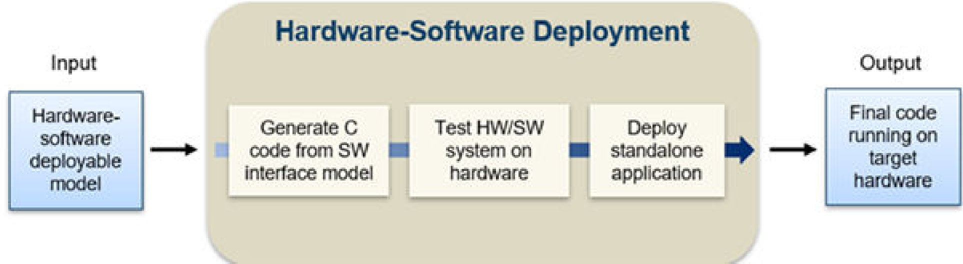


For more information, see “Generate Software Interface Model to Probe and Rapidly Prototype HDL IP Core” on page 39-84.

Hardware-Software Deployment

Generate C code (requires Embedded Coder) from your software interface model and use external mode or processor-in-the-loop (PIL) mode to deploy and run your hardware-software model on target hardware. The deployable design for your SoC device consists of:

- A generated bitstream from your original model containing your HDL IP core.
- A software interface model containing configured device drivers to allow communication between your processor and FPGA and C code generation for embedded processor.

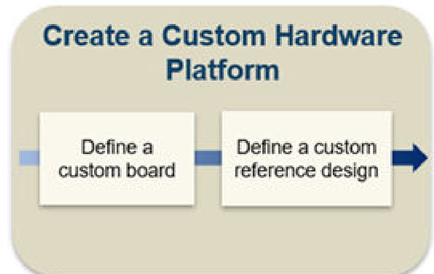


The deployment of your complete partitioned design into generated C code for the processor, from the software interface model, and a bitstream for the on-board FPGA, from your original model containing your IP core, enables you to run your algorithm on board an SoC or a Simulink Real-Time target machine that has FPGA I/O boards.

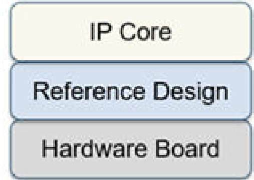
For an example, see “Debug a Zynq Design Using HDL Coder and Embedded Coder” on page 40-346.

Create Custom Hardware Platform

If there are no platforms already available for your hardware, you can define a custom hardware platform to use with the Hardware-Software Codesign workflow. A custom hardware platform creates flexibility to define your hardware system and deploy your design. For example, you can define a custom platform when deploying to production hardware. Creating a hardware platform consists of creating a new board definition to describe your hardware and creating one or more new reference designs for your board. You can also create new reference designs for an existing board.



A reference design provides a level of abstraction between your algorithmic IP core and the hardware board, making it easier and more efficient to design an algorithm to deploy on hardware. The reference design acts as the layer between your algorithm and the hardware platform, which you can use to leverage resources within your hardware without having to model these specific hardware resources in your design. You can focus on your algorithm design without implementing low-level details of the hardware. You can also retarget your algorithm to different hardware. For example, you can retarget your algorithm design when you transition from an evaluation board used for prototyping to production hardware.



For more information, see “Board and Reference Design Registration System” on page 40-89.

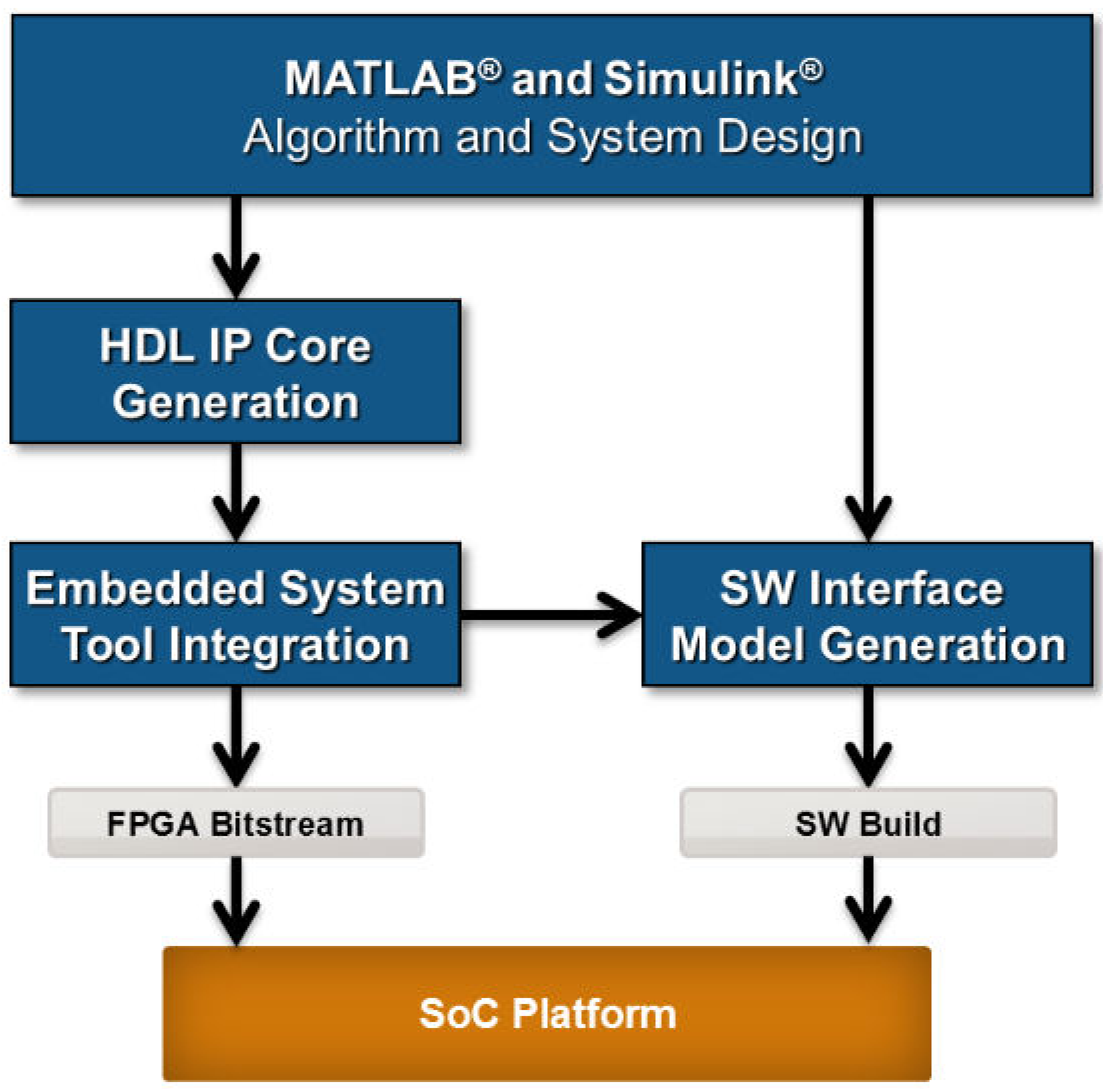
See Also

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- “Targeting FPGA & SoC Hardware”
- “HDL Coder Supported Hardware”

Hardware-Software Co-Design Workflow for SoC Platforms

The HDL Coder hardware-software co-design workflow helps automate the deployment of your MATLAB and Simulink design to a Zynq-7000 platform or Intel SoC platform. You can explore the best ways to partition and deploy your design by iterating through the following workflow.



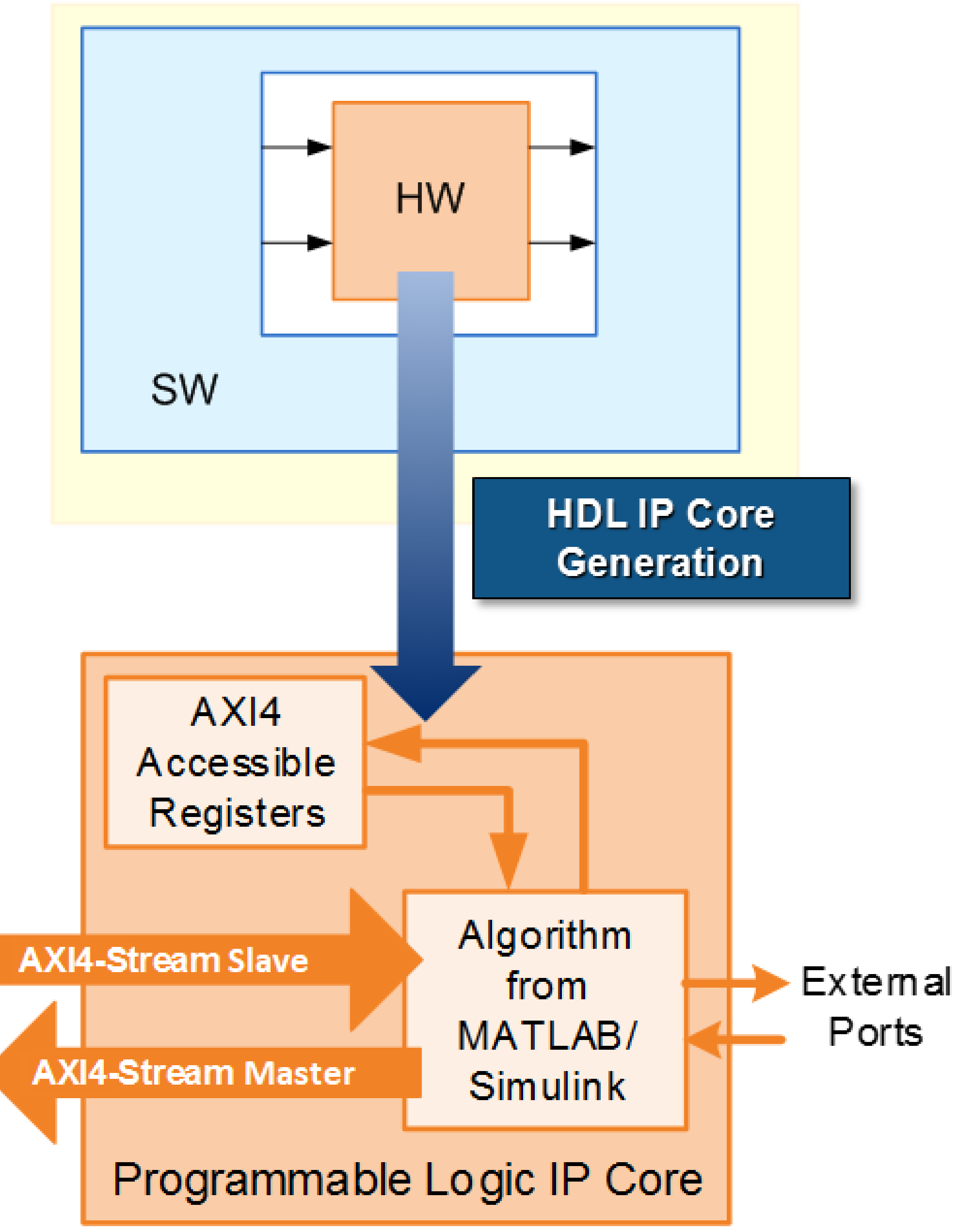
1. *MATLAB and Simulink Algorithm and System Design*: You begin by implementing your design in MATLAB or Simulink. When the design behavior meets your requirements, decide how to partition your design: which parts you want to run in hardware, and which parts you want to run in embedded software.

The part of the design that you want to run in hardware must use MATLAB syntax or Simulink blocks that are supported and configured for HDL code generation. See:

- “MATLAB Algorithm Design”
- “Model and Architecture Design”

2. *HDL IP Core Generation*: Enclose the hardware part of your design in an atomic Subsystem block or MATLAB function, and use the HDL Workflow Advisor to define and generate an HDL IP core.

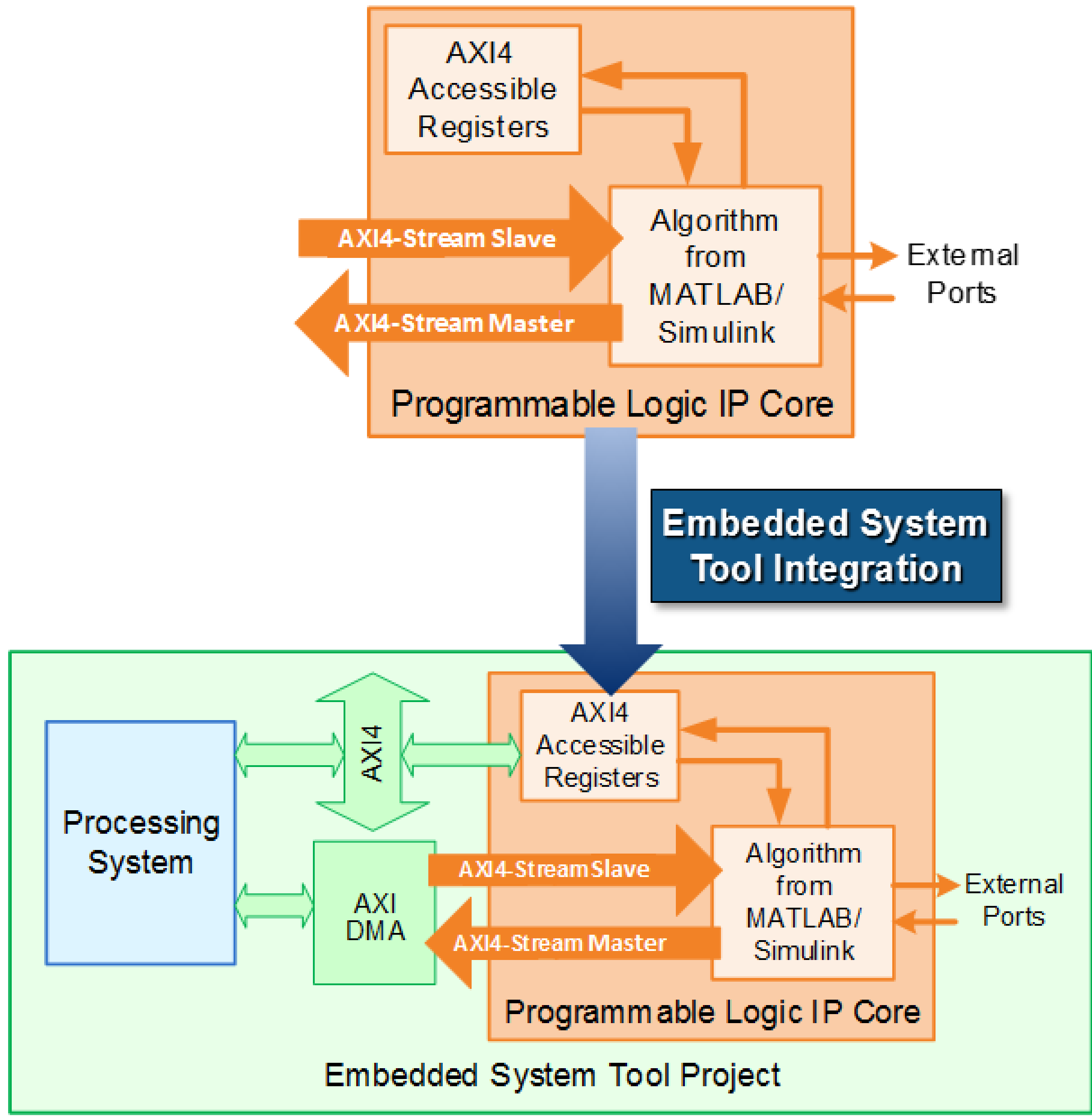
   The following diagram shows a design that has been partitioned into a hardware part, in orange, and software part, in blue. HDL IP core generation creates an IP core from the hardware part of the model. The IP core includes hardware interface components such as AXI4 accessible registers, AXI4 or AXI4-Lite interfaces, AXI4-Stream or AXI4-Stream Video interfaces, AXI4 Master interfaces, and external ports.



3 *Embedded System Tool Integration*: As part of the HDL Workflow Advisor IP core generation workflow, you insert your generated IP core into a *reference design*, and generate an FPGA bitstream for the SoC hardware.

The *reference design* is a predefined embedded system integration project. It contains all elements the Intel or Xilinx software needs to deploy your design to the SoC platform, except for the custom IP core and embedded software that you generate.

The following diagram illustrates the relationship between the reference design, in green, and the generated IP core, in orange.

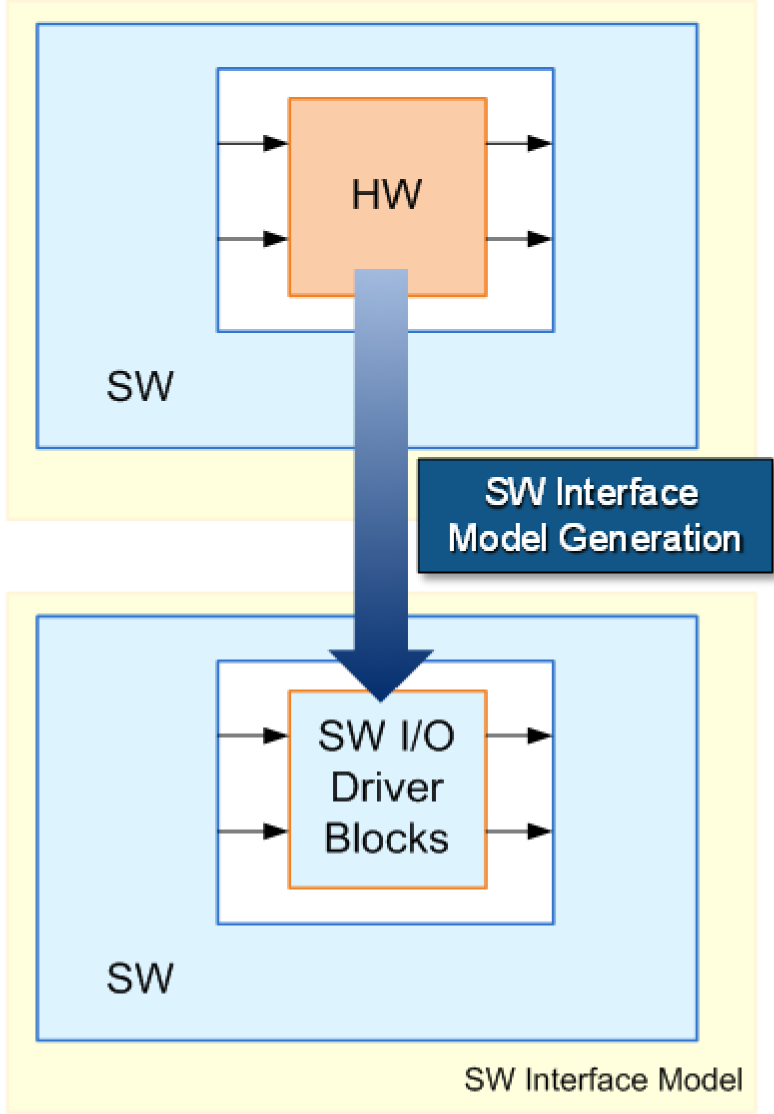


4 *SW Interface Generation* (requires a Simulink license and Embedded Coder license): In the HDL Workflow Advisor, after you generate the IP core and insert it into the reference design, you can optionally generate a software interface model, host interface model, and host interface script. The software interface model is your original model with AXI driver blocks replacing the hardware part. The host interface model enables you to write to or read from the memory-mapped locations on the target hardware over a JTAG or Ethernet cable by using the AXI Manager Write and AXI Manager Read blocks. The host interface script is a MATLAB file that is generated based on the reference design and Target platform interface table settings. It contains commands that enable you to connect to the target hardware, and to write to or read from the generated IP core by using the AXI driver blocks or the AXI Manager.

If you have an Embedded Coder license, you can automatically generate the software interface model and host interface script, generate embedded code from it, and build and run the executable on the Linux kernel on the ARM® processor. The generated embedded software includes AXI driver code generated from the AXI driver blocks that controls the HDL IP core.

If you do not have an Embedded Coder license or Simulink license, you can write the embedded software and manually build it for the ARM processor. See "Generate and Manage FPGA I/O Host Interface Scripts" on page 39-68

The following diagram shows the difference between the original model and the software interface model.



5 *SoC Platform* and *External Mode PIL*: Using the HDL Workflow Advisor, you program your FPGA bitstream to the SoC platform. You can then run the software interface model in external mode, or processor-in-the-loop (PIL) mode, to test your deployed design.

If your deployed design does not meet your design requirements, you can repeat the workflow with a modified model, or a different hardware-software partition.

See Also

"Targeting FPGA & SoC Hardware Overview" on page 39-3

Related Examples

- "Xilinx FPGA and SoC Devices"
- "Intel FPGA and SoC Devices"

Speedgoat FPGA Support with HDL Workflow Advisor

Use Simulink Real-Time and HDL Coder to implement Simulink algorithms and configure I/O functionality on Speedgoat Simulink-Programmable I/O modules. For an example that shows the development workflow for FPGA I/O modules, see “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 40-113.

When you open the HDL Workflow Advisor in HDL Coder and run the `Simulink Real-Time FPGA I/O` workflow, you generate a Simulink Real-Time interface subsystem. The subsystem mask controls the block parameters. Do not edit the parameters directly. The FPGA I/O board block descriptions are for informational purposes only.

Speedgoat Simulink-Programmable I/O Module Support

Speedgoat Simulink-Programmable I/O modules are part of Speedgoat target computer systems. To run the `Simulink Real-Time FPGA I/O` workflow, install the Speedgoat I/O Blockset and the Speedgoat HDL Coder Integration Packages. You can then choose the **Target platform** and run the workflow to generate a Simulink Real-Time interface subsystem. To see the documentation for the integration packages, enter this command at the MATLAB command prompt.

```
speedgoat.hdlc.doc
```

To learn about	See links
The integration packages and how you can install them.	See Speedgoat - HDL Coder Integration Packages documentation online at www.speedgoat.com/knowledge-center.
Speedgoat I/O modules that are supported with the HDL Workflow Advisor.	See Speedgoat Real-Time FPGA Application Support from HDL Coder.
Applications and use cases	See Common Use Cases under Speedgoat - HDL Coder Integration Packages documentation online at www.speedgoat.com/knowledge-center.
Supported interfaces for various types of I/O connectivity and protocols as well as fundamental functionality such as PCIe read/write and DMA.	See Interfaces under Speedgoat - HDL Coder Integration Packages documentation online at www.speedgoat.com/knowledge-center.
Provided examples for all supported I/O modules and functionality	See Examples under Speedgoat - HDL Coder Integration Packages documentation online at www.speedgoat.com/knowledge-center.

Prepare for FPGA Workflow

To work with FPGAs in the Simulink Real-Time environment, install:

- HDL Coder and Simulink Real-Time.
- Xilinx design tools with specific tool and version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”. You must also set up the path to the tool by using the `hdlsetuptoolpath` function.
- Speedgoat I/O Blockset and the Speedgoat HDL Coder Integration Packages.
- Speedgoat FPGA I/O module in the Speedgoat target machine.

You can use the workflow in HDL Coder to generate HDL code for your FPGA target device.

See Also

Related Examples

- "FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules" on page 40-113

More About

- "HDL Language Support and Supported Third-Party Tools and Hardware"
- "Tool Setup"

External Websites

- www.speedgoat.com

Custom IP Core Generation

In this section...
"Custom IP Core Architectures" on page 39-17
"Target Platform Interfaces" on page 39-17
"Processor and FPGA Synchronization" on page 39-18
"Custom IP Core Generated Files" on page 39-18
"Restrictions" on page 39-19

Using the HDL Workflow Advisor, you can generate a custom IP core from a model or algorithm. The generated IP core is sharable and reusable. You can integrate it with a larger design by adding it in an embedded system integration environment, such as Intel Qsys, Xilinx EDK, or Xilinx IP Integrator.

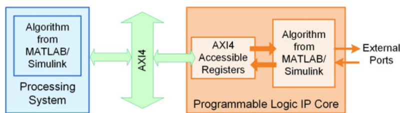
To learn how to generate a custom IP core, see:

- "Generate Board-Independent HDL IP Core from Simulink Model" on page 39-33
- "Generate Board-Independent IP Core from MATLAB Algorithm" on page 5-38

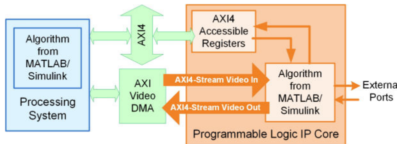
Custom IP Core Architectures

You can generate an IP core:

- With an AXI4 or AXI4-Lite interface.



- With an AXI4 or AXI4-Lite interface and AXI4-Stream Video interfaces.



- Without any AXI4 or AXI4-Lite interfaces. To learn more, see "Generate Board-Independent HDL IP Core from Simulink Model" on page 39-33.

The Algorithm from MATLAB and Simulink block represents your DUT. HDL Coder generates the rest of the IP core based on your target platform interface settings and processor or FPGA synchronization mode.

Target Platform Interfaces

You can map each port in your DUT to one of these target platform interfaces in the IP core:

- AXI4-Lite: Use this interface to access control registers or for lightweight data transfer. HDL Coder generates memory-mapped registers and allocates address offsets for the ports. See "Model Design for AXI4 Slave Interface Generation" on page 40-3.

- AXI4: Use this interface to connect to components that support burst data transmission. HDL Coder generates memory-mapped registers and allocates address offsets for the ports. In the generated HDL IP core, you can have either AXI4 or AXI4-Lite interface but not both interfaces. See “Model Design for AXI4 Slave Interface Generation” on page 40-3.
- AXI4-Master: Use this interface for designs that require accessing large data sets from an external memory. For more information, see “Model Design for AXI4 Master Interface Generation” on page 40-128.
- AXI4-stream: Use this interface for designs that require high speed data transfers. See “Model Design for AXI4-Stream Interface Generation” on page 40-14.
- AXI4-Stream Video: Use this interface to send or receive a 32-bit scalar video data stream. See “Model Design for AXI4-Stream Video Interface Generation” on page 40-118.
- External ports: Use external ports to connect to FPGA external IO pins or to other IP cores with external ports.
- FPGA Data Capture: Use FPGA Data Capture over the JTAG or Ethernet interface to observe test point signals and signals at the DUT output ports while your design runs on the FPGA. For an example of marking internal signals as test points, see “Debug IP Core Using FPGA Data Capture” on page 40-351. For more information on capturing data, see “Data Capture Workflow” (HDL Verifier).

Note To use this interface, you must download a hardware support package for your FPGA board. See “Download FPGA Board Support Package” (HDL Verifier).

To learn more about the AXI4, AXI4-Lite, and AXI4-Stream Video protocols, refer to your target hardware documentation. To add multiple AXI4-Stream and AXI4-Master interfaces and generate an IP core with multiple interfaces, see “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 40-33.

Processor and FPGA Synchronization

HDL Coder generates synchronization logic in the IP core based on the processor and FPGA synchronization mode that you choose.

When generating a custom IP core, these processor and FPGA synchronization options are available:

- `Free running` (default)
- `Coprocessing – blocking`

To learn more, see “Processor and FPGA Synchronization” on page 39-38.

Custom IP Core Generated Files

After you generate a custom IP core, the IP core files are in the `ipcore` folder within your project folder. In the HDL Workflow Advisor, you can view the IP core folder name in the **IP core folder** field of the **HDL Code Generation > Generate RTL Code and IP Core** task.

The IP core folder contains:

- IP core definition files.
- HDL source files (`.vhd` or `.v`).

- A C header file with the register address map.
- (Optional) An HTML report with instructions for using the core and integrating the IP core in your embedded system project.
- When you use the multicycle path constraints to meet the timing requirements, HDL Coder generates the constraints file of XDC format (`.xdc`) for Xilinx workflow and SDC format (`.sdc`) for Intel workflow.

Restrictions

The IP Core Generation workflow does not support **RAM Architecture** set to `Generic RAM without clock enable`.

When you disable clock domain crossing (CDC) you cannot use different clocks for the IP core and the AXI interface. The `IPCore_Clk` and `AXILite_ACLK` must be synchronous and connected to the same clock source. The `IPCore_RESETN` and `AXILite_ARESETN` must be connected to the same reset source. See “Synchronization of Global Reset Signal to IP Core Clock Domain” on page 39-40 When you enable CDC you can connect different clocks for the IP core and the AXI interface. To learn more about CDC, see “Enable Clock Domain Crossing on AXI4-Lite Interfaces” on page 40-439.

See Also

Related Examples

- “Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow” on page 39-190
- “Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow” on page 39-201

More About

- “Multirate IP Core Generation” on page 40-85

Custom IP Core Report

In this section...
"Summary" on page 39-20
"Target Interface Configuration" on page 39-20
"Register Address Mapping" on page 39-21
"Bit Packing Order" on page 39-22
"IP Core User Guide" on page 39-23
"IP Core File List" on page 39-26

You generate an HTML custom IP core report by default when you generate a custom IP core. The report describes the behavior and contents of the generated custom IP core.

Summary

The Summary section shows your coder settings when you generated the custom IP core.

The following figure is an example of a Summary section.

Summary

IP core name	DUT_ip
IP core version	1.0
IP core folder	hdl_prj\ipcore\DUT_ip_v1_0
Target platform	Arrow SoCKit development board
Target tool	Altera QUARTUS II
Target language	Verilog
Reference Design	Default system
Model	axi4_vec
Model version	1.91
HDL Coder version	3.10
IP core generated on	10-Dec-2016 21:06:26
IP core generated for	DUT

Target Interface Configuration

The Target Interface Configuration section shows how your DUT ports map to the target hardware interface and the processor/FPGA synchronization mode.

The following figure is an example of a Target Interface Configuration section.

Target Interface Configuration

You chose the following target interface configuration for axi4_vec :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
In1	Inport	uint8 (3)	AXI4	x"100"
In2	Inport	uint16 (100)	AXI4	x"200"
In3	Inport	single	AXI4	x"114"
In4	Inport	uint32	AXI4	x"118"
Out1	Outport	uint8 (3)	AXI4	x"160"
Out2	Outport	uint16 (100)	AXI4	x"A00"
Out3	Outport	single	AXI4	x"11C"
Out4	Outport	uint32	AXI4	x"120"

To learn more about processor/FPGA synchronization modes, see “Processor and FPGA Synchronization” on page 39-38.

To learn more about target platform interfaces, see “Custom IP Core Generation” on page 39-17.

Register Address Mapping

The Register Address Mapping section shows the address offsets for AXI4-Lite bus accessible registers in your custom IP core, and the name of the C header file that contains the same address offsets.

The following figure is an example of a Register Address Mapping section.

Register Address Mapping

The following AXI4 bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
In1_Data	0x100	data register for Inport In1, vector with 3 elements, address ends at 0x108
In1_Strobe	0x110	strobe register for port In1
In3_Data	0x114	data register for Inport In3
In4_Data	0x118	data register for Inport In4
Out3_Data	0x11C	data register for Outport Out3
Out4_Data	0x120	data register for Outport Out4
Out1_Data	0x160	data register for Outport Out1, vector with 3 elements, address ends at 0x168
Out1_Strobe	0x170	strobe register for port Out1
In2_Data	0x200	data register for Inport In2, vector with 100 elements, address ends at 0x38C
In2_Strobe	0x400	strobe register for port In2
Out2_Data	0xA00	data register for Outport Out2, vector with 100 elements, address ends at 0xB8C
Out2_Strobe	0xC00	strobe register for port Out2

The register address mapping is also in the following C header file for you to use when programming the processor:
include\DUT_ip_addr.h
The IP core name is appended to the register names to avoid name conflicts.

Bit Packing Order

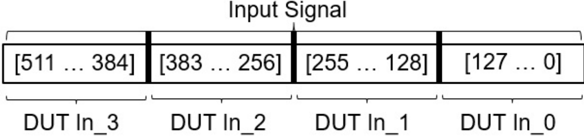
The Bit Packing Order section gives a high-level overview of the data packing order for vector inputs and outputs for Internal IO, External IO, and External port interfaces. This section only appears when the port width for Internal IO, External IO, and External ports are set to greater than 128 bits wide.

This image is an example of the Bit packing Order section:

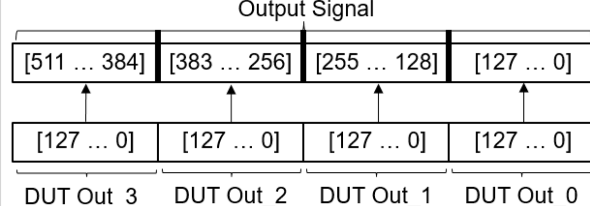
Bit Packing Order

Following is the general representation of data packing order and data unpacking order for Vector Input and output cases for Internal IO, External IO and External port interfaces. If it is assumed that an interface is mapped to one input(or output) port of the model which has a port width of 128 and port dimension of 4, then:

Following is the bit packing order to the DUT IP for Input vector case.



Following is the bit unpacking order from the DUT IP for Vector Output case.



It should be noted that the above instances are just for demonstration purpose and may not represent the actual mapped port width and port dimension.

IP Core User Guide

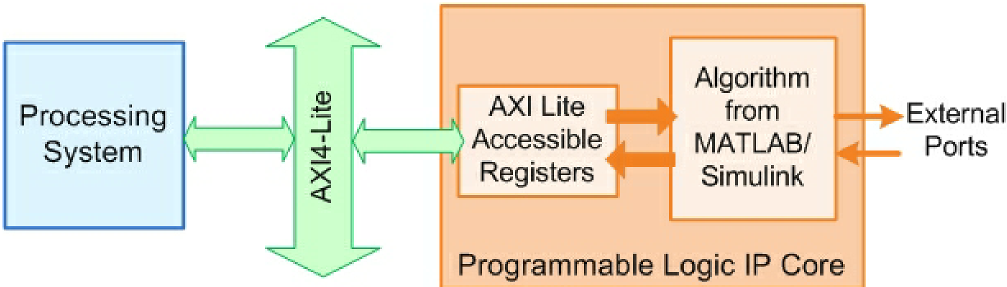
The IP Core User Guide section gives a high-level overview of the system architecture, describes the processor and FPGA synchronization mode, and gives instructions for integrating the IP core in your embedded system integration environment.

The following figure is an example of an IP Core User Guide system architecture description.

Theory of Operation

This IP core is designed to be connected to an embedded processor with an **AXI4-Lite bus.** The processor acts as bus master, and the IP core acts as slave. By accessing the generated registers via the AXI4-Lite bus, the processor can control the IP core, and read and write data from and to the IP core.

For example, to reset the IP core, write 0x1 to the bit 0 of IPCore_Reset register. To enable or disable the IP core, write 0x1 or 0x0 to the IPCore_Enable register. To access the data ports of the MATLAB/Simulink algorithm, read or write to the associated data registers.

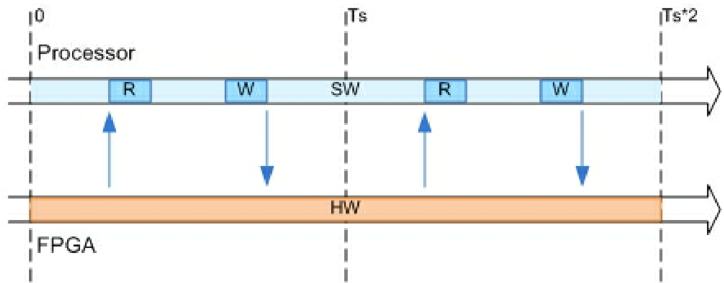


This IP core also support the **External Port** interface. To connect the external ports to the FPGA external IO pins, add FPGA pin assignment constraints in the Xilinx EDK environment.

The following figure is an example of a processor/FPGA synchronization description.

Processor/FPGA Synchronization

The **Free running** mode means there is no explicit synchronization between embedded processor software execution (SW) and the IP core (HW). SW and HW runs independently. The data written from the processor to IP core takes effect immediately, and the data read from the IP core is the latest data available on the IP core output ports.

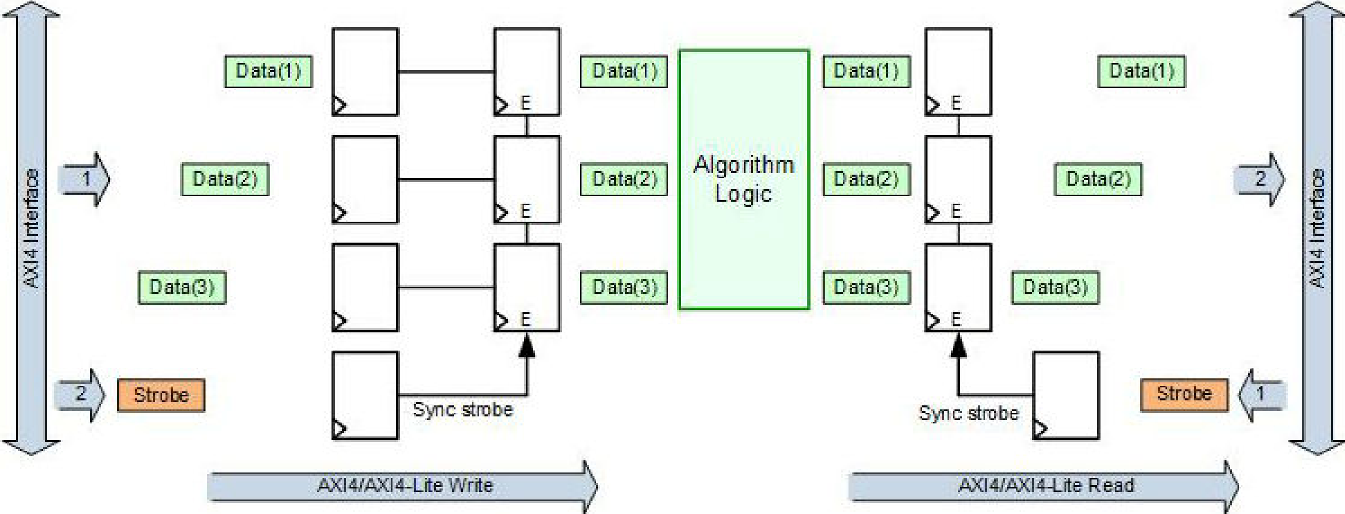


If you use vector data signals at the DUT interface, the IP core report displays this section that shows how the code generator synchronizes vector data across the AXI4 interface.

Vector Data Read/Write with Strobe Synchronization

All the elements of vector data are treated as synchronous to the IP core algorithm logic. Additional strobe registers added for each vector input and output port maintain this synchronization across multiple sequential AXI4 reads/writes. For input ports, the strobe register controls the enables on a set of shadow registers, allowing the IP core logic to see all the updated vector elements simultaneously. For output ports, the strobe register controls the synchronous capturing of vector data to be read.

To read a vector data port, first write the strobe address with 0x1, then read each desired data element from corresponding address range. To write a vector data port, first write each desired data element, then write 0x1 to the strobe address to complete the transaction.



The following figure is an example of instructions for integrating the IP core into your embedded system integration environment on the Xilinx platform. If you are targeting an Altera platform, the report displays similar instructions for integrating the IP core into the Altera Qsys environment.

EDK Environment Integration

This IP Core is generated for the Xilinx EDK environment. The following steps are an example showing how to add the IP core into the EDK environment:

1. Copy the IP core folder into the "pcores" folder in your Xilinx Platform Studio (XPS) project. This step adds the IP core into the XPS project user library.
2. In the XPS project, find the IP core in the user library and add the IP core to the design.
3. Connect the S_AXI port of the IP core to the embedded processor's AXI master port.
4. Connect the clock and reset ports of the IP core to the global clock and reset signals.
5. Assign a base address for the IP core.
6. Connect external ports and add FPGA pin assignment constraints.
7. Generate FPGA bitstream and download the bitstream to target device.

IP Core File List

The IP Core File List section lists the files and file folders that comprise your custom IP core.

The following figure is an example of an IP core file list.

IP Core File List

The IP core folder is located at:
hdl_prj\ipcore\hdlcoder_led_blinking_led_counter_pcore_v1_00_a
Following files are generated under this folder:

IP core definition files
data\hdlcoder_led_blinking_led_counter_pcore_v2_1_0.mpd
data\hdlcoder_led_blinking_led_counter_pcore_v2_1_0.pao

IP core report
doc\hdlcoder_led_blinking_ip_core_report.html

IP core HDL source files
hdl\vhdl\led_counter_pkg.vhd
hdl\vhdl\led_counter.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_dut.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_axi_lite_module.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_addr_decoder.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_axi_lite.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore.vhd

IP core C header file
include\hdlcoder_led_blinking_led_counter_pcore_addr.h

See Also

More About

- "Custom IP Core Generation" on page 39-17
- "Hardware-Software Co-Design Workflow for SoC Platforms" on page 39-9
- "Multirate IP Core Generation" on page 40-85

Comparison of IP Core Generation Techniques

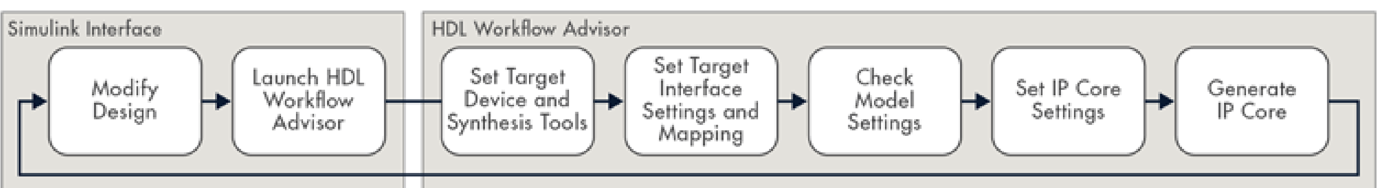
To generate an IP core interactively from a Simulink design, you can use:

- The HDL Workflow Advisor. In releases prior to R2023b, this is the only method to generate an IP core interactively from a Simulink model. See “Using the HDL Workflow Advisor” on page 5-28.
- The Simulink Toolstrip. You can use the **HDL Code** tab to generate an IP core and use the IP Core editor and the Configuration Parameters dialog box to configure the HDL and IP core settings. See “Create Custom Simulink Toolstrip Tabs”.

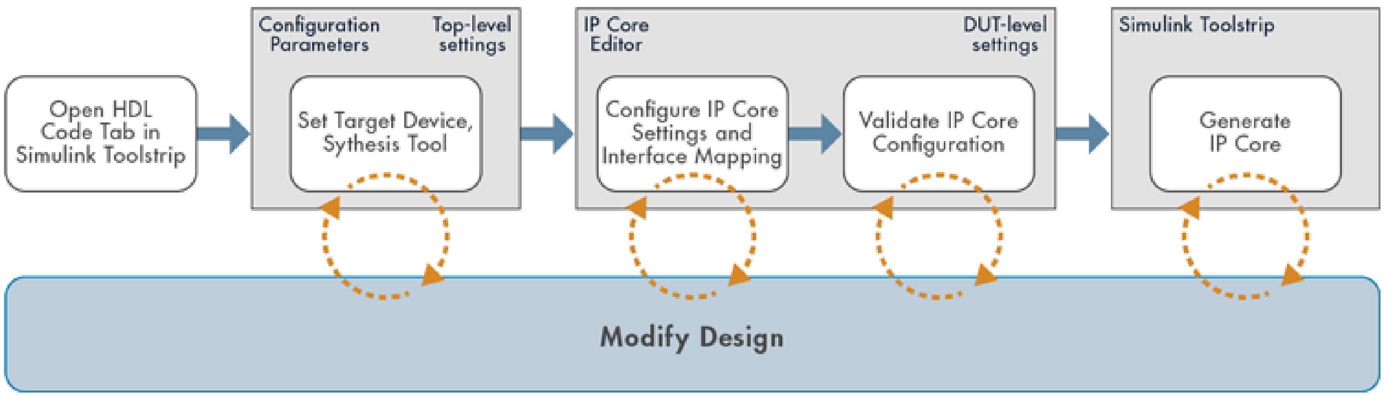
While both the HDL Workflow Advisor and the Simulink Toolstrip can generate an IP core, each IP core generation method offers different advantages. The HDL Workflow Advisor provides step-by-step instructions that you can apply to new models that are not yet configured for IP core generation. You can also use the HDL Workflow Advisor as a guided workflow if you are new to the IP core generation process. After you generate an IP core and you want to iterate on your design, you can use the Simulink Toolstrip to rapidly prototype and generate an updated IP core with more flexibility, efficiency, and scalability.

IP Core Process Comparison

This diagram shows the sequential steps when using the HDL Workflow Advisor to generate an IP core:



This diagram shows how to rapidly prototype, generate an IP core, and iterate on your design using the Simulink Toolstrip. The general order of steps follows the left-to-right order of the **HDL Code** tab in the Simulink Toolstrip, but with flexibility to modify your design at any point in the process.



When rapidly prototyping a design, you need to generate an IP core quickly with multiple design iterations. Using the Simulink Toolstrip allows you to quickly to generate an IP core. You can stay inside the Simulink environment, make changes to your design, and update your IP core without needing to re-run any previous options and tasks that you already completed. Using the Simulink

Toolstrip gives you greater flexibility to design, configure, and generate an IP core in the order and iteration that best suits your workflow and design process.

This table compares the two processes:

Workflow Advisor Task	Simulink Toolstrip Step
Open the HDL Workflow Advisor	Open the **HDL Coder** app from the **Apps** tab on the Simulink Toolstrip. In the **HDL Code** tab, select **IP Core** from the drop-down button in the **Output** section.
1.1 Set Target Device and Synthesis Tools	In the Configuration Parameters dialog box, use the **HDL Code Generation > Target** pane to set the top-level model settings, such as the target device and synthesis tool settings.
1.2 Set Target Reference Design	In the Configuration Parameters dialog box, use the **HDL Code Generation > Target** pane to set the reference design parameters. Note that the **Reference design parameters** table requires a string input typed directly into the **Value** box.
1.3 Set Target Interface	Use the IP Core editor to set DUT-level settings. In the IP Core editor, use the **Interface Mapping** tab to set the values in the target platform interface table. Use the **Interface Settings** tab to configure interface-related settings for the IP core, such as the register interface and FPGA data capture properties. Use the Reload IP core settings button to compile your model and repopulate the DUT ports and their data types in the target platform interface table. Use the Validate IP core settings button to verify the interface table and IP core settings.
1.4 Set Target Frequency	In the Configuration Parameter dialog box, use the **HDL Code Generation > Target** pane to set the **Target Frequency** parameter.
2.1 Check Model Settings	On the **HDL Code** tab of the Simulink Toolstrip, use the **HDL Code Advisor** button to check your model settings. For more information, see “HDL Code Advisor Checks” on page 38-9.
3.1 Set HDL Options	Use the Configuration Parameters dialog box to set HDL code and testbench generation options.

Workflow Advisor Task	Simulink Toolstrip Step
3.2 Generate RTL Code and IP Core	In the IP Core editor, use the **General** and **Clock Settings** tabs to configure general IP core and clock-related settings, respectively. On the **HDL Code** tab of the Simulink Toolstrip, use the **Generate IP Core** button to generate an IP core.

For an example on IP core generation using the Simulink Toolstrip, see “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98. For an example on IP core generation using the HDL Workflow Advisor, see “Getting Started with Targeting Zynq UltraScale+ MPSoC Platform” on page 39-112.

Limitations

When generating an IP core with either the HDL Workflow Advisor or the Simulink Toolstrip:

- You can specify the parameters and settings for an IP core by using the HDL Workflow Advisor, IP Core editor, or Configuration Parameters dialog box. Updating a parameter using one of these options also updates the parameter in the other locations.
- When the HDL Workflow Advisor is open, you cannot configure IP core generation settings and options using the Configuration Parameter dialog box or the IP Core editor. To use the Simulink Toolstrip to update an IP core and its settings, first close the HDL Workflow Advisor.

See Also

IP Core Editor

Related Examples

- “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98

Comparison of IP Core Deployment and Verification Techniques

When prototyping a design on hardware, you must generate a bitstream and program your target device over multiple design iterations. Using the Simulink Toolstrip to generate a bitstream allows you to rapidly deploy the IP core to your target device. You can stay in the Simulink environment, while you make changes to your design, re-generate the bitstream, or program your device without having to repeat previously completed tasks.

Prior to R2023b, you completed the sequential tasks in HDL Workflow Advisor to generate an IP core project to enable the host interface script and software interface generation. Starting in R2023b, you can generate the software interface model and the host interface script from the Simulink Toolstrip directly by using the IP core editor to configure the IP Core settings and interface mapping. Use the Simulink Toolstrip to prototype your design and generate a bitstream, host interface script, and software interface model with higher flexibility, efficiency, and scalability.

Embedded System Integration Process Comparison

To compare the IP core generation processes using HDL Workflow Advisor and the Simulink Toolstrip, see “Comparison of IP Core Generation Techniques” on page 39-27. This table compares the process to generate a bitstream by using the HDL Workflow Advisor and the Simulink Toolstrip:

Workflow Advisor Task	Simulink Toolstrip Step
Open the HDL Workflow Advisor	Open the **HDL Coder** app from the **Apps** tab on the Simulink Toolstrip. In the **HDL Code** tab, in the **Output** selection, select **IP Core** from the drop-down button.
Task 4.1 Create Project	In the **HDL Code** tab, select **Build Bitstream > Create IP Core Project**. To specify a custom project name and other options, in the **HDL Code** tab, select **Build Bitstream > Deployment Settings** and specify the project-related settings under **Create IP Project Settings**.

Workflow Advisor Task	Simulink Toolstrip Step
Task 4.2 Generate Software Interface	To generate the software interface model, in the **HDL Code** tab, select **Build Bitstream > Software Interface Model**. To specify the operating system for the software model, in the **HDL Code** tab, select **Build Bitstream > Deployment Settings**, and specify the project-related settings under **Generate Software Interface**. You can generate the software interface model without generating the IP core. To generate the host interface script, in the **HDL Code** tab, select **Host Interface Script > Host Interface Script**. To specify the target interface for the host interface script, in the **HDL Code** tab, select **Host Interface Script > Host target interface** and select either `Ethernet` or `JTAG` from the drop-down menu options. You can generate the host interface script without generating the IP core.
Task 4.3 Build FPGA Bitstream	In the **HDL Code** tab, click the **Build Bitstream** button. To specify the options related to bitstream build, in the **HDL Code** tab, select **Build Bitstream > Deployment Settings**, and specify the bitstream-related settings under **Build Bitstream Settings**.
Task 4.4 Program Target Device	In the **HDL Code** tab, select **Build Bitstream > Program Target Device**. To specify the options related to the bitstream build, in the **HDL Code** tab, select **Build Bitstream > Deployment Settings**, and specify the program target device settings under **Program Device Settings**.

Limitations

When deploying and verifying your IP core using either the HDL Workflow Advisor or the Simulink Toolstrip:

- You can specify the parameters and settings for a the embedded system integration tasks by using the HDL Workflow Advisor or you can specify the IP core deployment settings on the **HDL Code** tab by clicking **Build Bitstream** > **Deployment Settings**. Updating a parameter using one of these options also updates the parameter in the other locations.
- When the HDL Workflow Advisor is open, you cannot perform any IP core generation, deployment, or verification actions from the Simulink Toolstrip. To use the Simulink Toolstrip to deploy and validate an IP core, first close the HDL Workflow Advisor.
- When the **Target Device** parameter is set to a generic FPGA device, the **Build Bitstream** and **Host Interface Script** buttons are dimmed.
- You cannot generate a host interface model from the Simulink Toolstrip.

See Also

Related Examples

- “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98
- “Comparison of IP Core Generation Techniques” on page 39-27

Generate Board-Independent HDL IP Core from Simulink Model

In this section...

"Generate Board-Independent IP Core" on page 39-33
"Generate Tool-Independent IP Core" on page 39-35
"IP Core Without AXI4 Slave Interfaces" on page 39-36
"Requirements and Limitations for IP Core Generation" on page 39-36

When you open the HDL Workflow Advisor and run the `IP Core Generation` workflow for your Simulink model, you can specify a generic Xilinx platform, a generic Intel platform, generic Microchip platform or generic platform. The workflow then generates a generic IP core that you can integrate into any target platform of your choice. For IP core integration, define and register a custom reference design for your target board by using the `hdlcoder.ReferenceDesign` class. To learn more, see:

- "Define Custom Board and Reference Design for Zynq Workflow" on page 40-252
- "Define Custom Board and Reference Design for Intel SoC Workflow" on page 40-270
- "Define Custom Board and Reference Design for Microchip Workflow" on page 40-285

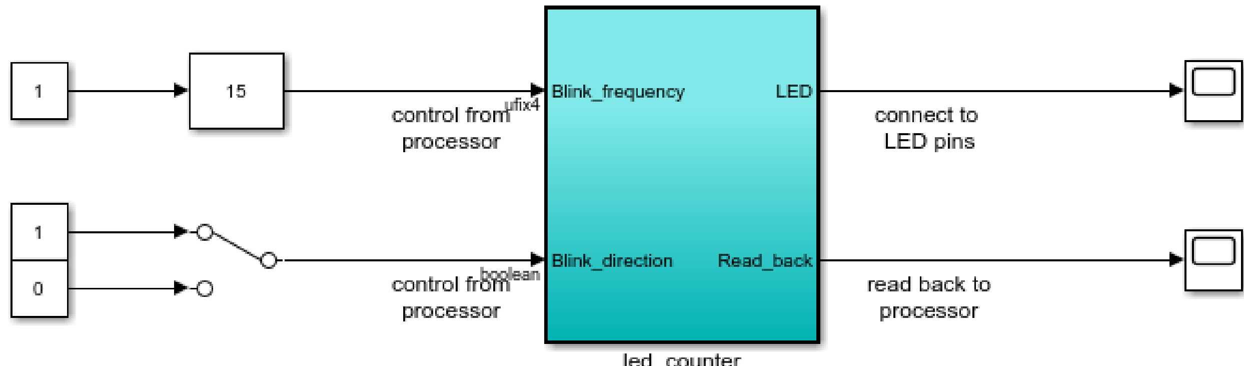
Generate Board-Independent IP Core

To generate a board-independent custom IP core to use in an embedded system integration environment, such as Intel Qsys, Xilinx EDK, or Xilinx IP Integrator:

1. Select DUT in Simulink model and open the HDL Workflow Advisor. For example, open the model `hdlcoder_led_blinking`.

```
open_system('hdlcoder_led_blinking')
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
hdladvisor('hdlcoder_led_blinking/led_counter')

Launch HDL Workflow Advisor

Run Demo

Copyright 2012 The MathWorks, Inc.

2 Set the path to the installed synthesis tool for the target device by using the `hdlsetuptoolpath` function. For example, if Xilinx Vivado is the synthesis tool, use the command:

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
             'C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

See “HDL Language Support and Supported Third-Party Tools and Hardware” for latest supported version of the synthesis tool.

3 Open the HDL Workflow Advisor for the DUT Subsystem. For the LED blinking model, the `led_counter` Subsystem is the DUT. In the **Set Target** > **Set Target Device and Synthesis Tool** task:

- For **Target workflow**, select `IP Core Generation`.
- For **Target platform**, depending on the synthesis tool and device that you are targeting, select `Generic Altera Platform` or `Generic Xilinx Platform`. Click **Run This Task**.

4 In the **Set Target** > **Set Target Interface** task, select a **Target Platform Interface** for each port, and then click **Apply**. You can map each DUT port to one of these interfaces: `AXI4-Lite`, `AXI4`, `AXI4-Stream`, `AXI4-Stream Video`, `External Port`, or `FPGA Data Capture`. For more information about these interfaces, see “Target Platform Interfaces” on page 39-17.

You can also map the ports to multiple target platform interfaces. To learn more, see “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 40-33.

If you do not want to map the DUT ports to AXI4 slave interfaces, you can map them to `External Port` interfaces.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
Blink_frequency	Inport	ufix4	External Port	
Blink_direction	Inport	boolean	External Port	
LED	Outport	uint8	External Port	
Read_back	Outport	uint8	External Port	

5 Expand the **Set Code Generation Options** task. Right-click the **Set Optimization Options** task and select **Run to Selected Task**.

6 In the **HDL Code Generation > Generate RTL Code and IP Core** task, you can specify:

- Whether you want to connect the DUT IP core to multiple AXI Master interfaces. By default, the **AXI4 Slave ID Width** value is `12`, which enables you to connect the HDL IP core to one AXI Master interface. To connect the DUT IP core to multiple AXI Master interfaces, you may want to increase the **AXI4 Slave ID Width**. When you run this task, this setting is saved on the DUT as the HDL block property **AXI4SlaveIDWidth**.

  To learn more, see "Define Multiple AXI Master Interfaces in Reference Designs to Access DUT AXI4 Slave Interface" on page 39-60.
- Whether you want to generate the default AXI4 slave interface. By default, HDL Coder generates AXI4 slave interfaces for signals such as clock, reset, ready, timestamp, and so on. If you do not want to generate any AXI4 slave interfaces, clear the **Generate default AXI4 slave interface** check box.

Note If you mapped any of the DUT ports to AXI4 slave interfaces in the **Set Target Interface** task, the code generator maps the ports to AXI4 slave interfaces, whether or not the **Generate default AXI4 slave interface** check box is cleared.

Click **Run This Task**. When you clear the check box and run the task, the code generator saves this setting on the DUT Subsystem as the HDL block property **GenerateDefaultAXI4Slave**.

7 After running the task, HDL Coder generates the IP core files in the output folder shown the **IP core folder** field, including the HTML documentation. To view the IP core report, click the link in the message window.

Generate Tool-Independent IP Core

Using `Generic Platform`, you can generate IP core and RTL code without setting the third-party synthesis tool. To generate IP core using `Generic Platform`, follow these steps:

1 In step **1.1. Set Target Device and Synthesis Tool**, set **Target workflow** to `IP Core Generation`. Then, set the **Target platform** to `Generic Platform`. Synthesis tool and target device information are not required when you set this target platform.

2 In step **1.2. Set Target Interface**, set **Target Platform Interfaces** for the input and output ports. You can choose `AXI4` as well as `external port` as the target interface for the ports.

3 Perform each task in the HDL Workflow Advisor until step **3.1. Set HDL Options**.

4 In step **3.2. Generate RTL Code and IP Core**, run this task to generate RTL code and IP core for your model. The generated code can be integrated on the hardware.

IP Core Without AXI4 Slave Interfaces

When you run the `IP Core Generation` workflow, you can also generate an HDL IP core without any AXI4 slave interfaces in your reference design.

To run this workflow, open the HDL Workflow Advisor, specify `Generic Xilinx Platform` or `Generic Altera Platform` as the target platform, and map the DUT ports to only External Port, or AXI4-Stream interface with TLAST mapping. In addition, when you generate the HDL IP core, in the **Generate RTL Code and IP Core** task, clear the **Generate default AXI4 slave interface** check box, and then select **Run This Task**.

Use this capability when:

- You do not want to tune the IP core parameters by using the AXI4 slave interfaces.
- You want to create a custom reference design without AXI4 slave interfaces, such as standalone FPGA boards.

In addition, avoiding generation of the AXI4 slave interfaces in such cases reduces hardware resource usage and design complexity.

Note External IO and internal IO interfaces connect your HDL IP core to other existing IPs in your custom reference design. To define these interfaces, you use the `addInternalIOInterface` and `addExternalIOInterface` methods of the `hdlcoder.ReferenceDesign` class.

To integrate the HDL IP core, you can create a custom reference design without AXI4 slave interfaces. In the custom reference design, you can only use External IO, Internal IO or AXI4-Stream interface with TLAST mapping. For examples, see:

- “Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow” on page 39-190
- “Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow” on page 39-201

When you generate an HDL IP core without AXI4 slave interfaces, certain restrictions apply. See “IP Core Without AXI4 Slave Interface Restrictions” on page 39-37.

Requirements and Limitations for IP Core Generation

Custom IP Core Generation Limitations

- The DUT must be an atomic system.
- The same IP core cannot use both an AXI4 interface and an AXI4-Lite interface.
- The DUT cannot contain Xilinx System Generator blocks or Intel DSP Builder Advanced blocks.
- If your target language is VHDL, and your synthesis tool is Xilinx ISE or Intel Quartus Prime, the DUT cannot contain a model reference.

AXI4-Lite Interface Restrictions

- The input and output ports must have a bit width less than or equal to 32 bits.
- The input and output ports must be scalar.

AXI4-Stream Video Interface Restrictions

- Ports must have a 32-bit width.
- Ports must be scalar.
- You can have a maximum of one input video port and one output video port.
- The AXI4-Stream Video interface is not supported in `Coprocessing – blocking` mode. **Processor/FPGA synchronization** must be set to `Free running` mode.

IP Core Without AXI4 Slave Interface Restrictions

- You can only map the ports to External/Internal IO interfaces, or AXI4-Stream interface with TLAST mapping. Other interfaces that require AXI4 slave interfaces such as AXI4 Master, AXI4-Stream, and AXI4-Stream Video are not supported.
- You must use the `Free running` mode for **Processor/FPGA synchronization**. `Coprocessing – blocking` mode is not supported.

See Also

Classes
`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

More About

- “Custom IP Core Generation” on page 39-17
- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-38
- “Board and Reference Design Registration System” on page 40-89
- “Generate Board-Independent HDL IP Core for Microchip Platforms” on page 39-260

Processor and FPGA Synchronization

In the HDL Workflow Advisor, you can choose a **Processor/FPGA synchronization mode** for your processor and FPGA when you generate a custom IP core to use in an embedded system integration project. The following synchronization modes are available:

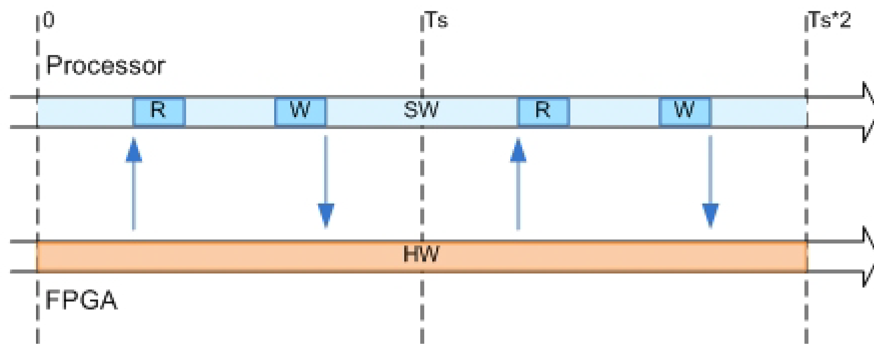
- `Free running` (default)
- `Coprocessing – blocking`
- `Coprocessing – nonblocking with delay`

Free Running Mode

In free running mode, the processor and FPGA each run nonsynchronized, continuously, and in parallel.

Select **Free running** as the **Processor/FPGA synchronization mode** when you do not want your processor and FPGA to be automatically synchronized.

The following diagram shows how the processor and FPGA can communicate in free running mode. The shaded areas indicate that the processor and FPGA are running continuously.

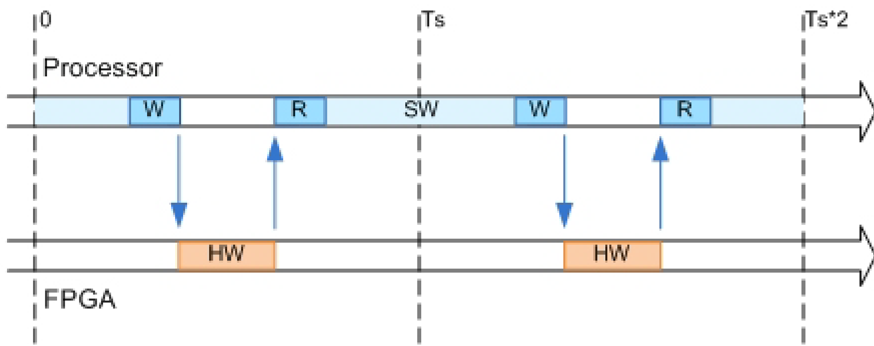


Coprocessing – Blocking Mode

In blocking coprocessor mode, HDL Coder automatically generates synchronization logic for the FPGA so that the processor and FPGA run in tandem.

Select **Coprocessing – blocking** as the **Processor/FPGA synchronization mode** when FPGA execution time is short relative to the processor sample time, and you want the FPGA to complete before the processor continues.

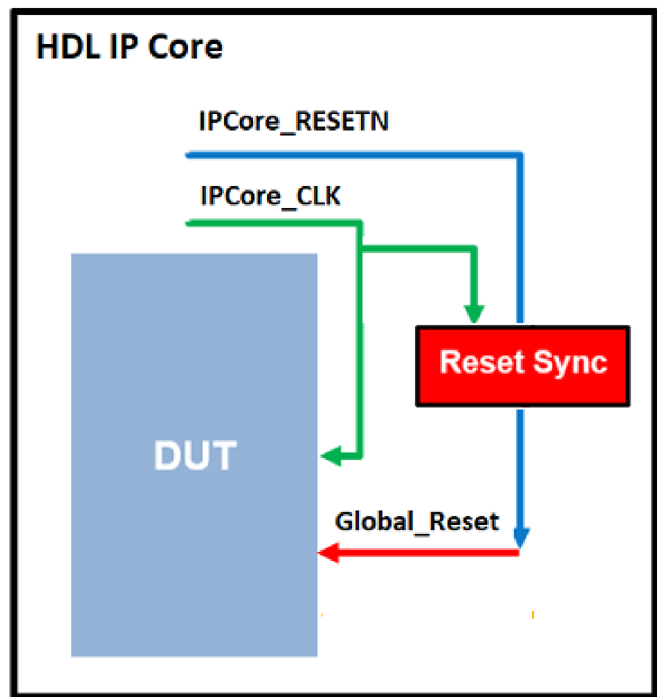
The following diagram shows how the processor and FPGA run in blocking coprocessing mode.



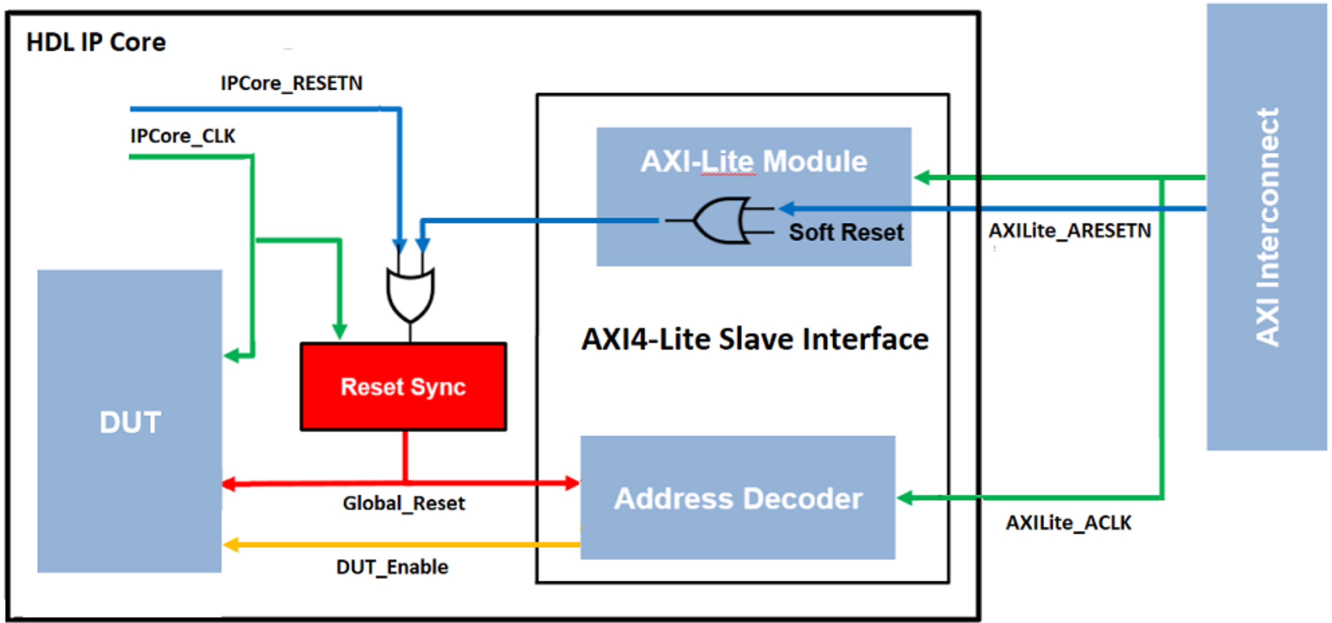
The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor writes to the FPGA, then stops and waits for an indication that the FPGA has finished processing before continuing to run. Each time the FPGA runs, it executes the logic generated for one DUT subsystem sample time.

Synchronization of Global Reset Signal to IP Core Clock Domain

The HDL DUT IP core and the Address Decoder logic in the AXI4 Slave interface wrapper of the HDL IP core are driven by a global reset signal. If you generate an HDL IP core without any AXI4 slave interfaces, HDL Coder does not generate the AXI4 slave interface wrapper. The global reset signal becomes the same as the IP core reset signal and drives the HDL IP core for the DUT. To learn how you can generate an IP core without AXI4 slave interfaces, see “Generate Board-Independent HDL IP Core from Simulink Model” on page 39-33.



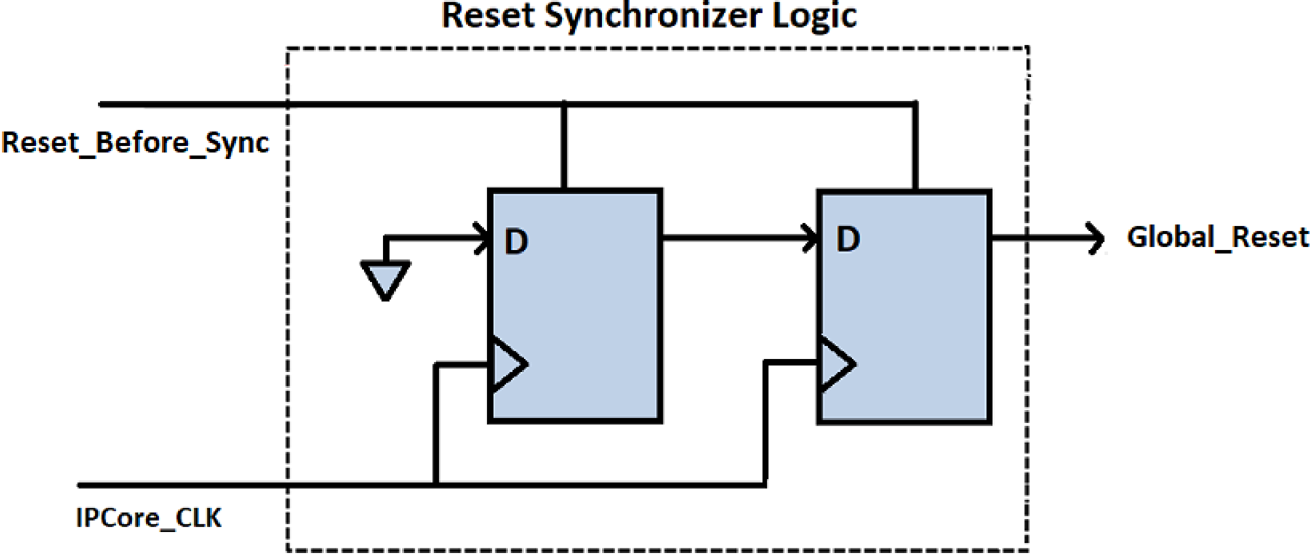
When you generate the AXI4 slave interfaces in the HDL IP core, the global reset signal is driven by three reset signals: the IP core external reset, the reset signal of the AXI interconnect, and the soft reset which is asserted when you write to the `IPCore_Reset` AXI register. For more information, see “Custom IP Core Report” on page 39-20. The global reset signal in this case drives the HDL IP core for the DUT and the Address Decoder logic in the AXI4 slave wrapper.



The `IPCore_Clk` and `AXILite_ACLK` must be connected to the same clock source. The `IPCore_RESETN` and `AXILite_ARESETN` must be connected to the same reset source.

These reset signals can be either synchronous or asynchronous. Using asynchronous reset signals can be problematic and result in potential metastability issues in flipflops when the reset de-asserts within the latching window of the clock. To avoid generation of possible metastable values when combining the reset signals, HDL Coder automatically inserts a reset synchronization logic, as indicated by the `Reset Sync` block. The reset synchronization logic synchronizes the global reset signal to the IP core clock domain. This logic is inserted when you open the HDL Workflow Advisor and run the **Generate RTL Code and IP Core** task of the `IP Core Generation` workflow.

The reset synchronization logic contains two back-to-back flipflops that are synchronous to the `IPCore_CLK` signal. The flipflops make sure that de-assertion of the reset signal occurs after two clock cycles of when the `IPCore_CLK` signal becomes high. This synchronous de-assertion avoids generation of a global reset signal that has possible metastable values.



The logic works differently depending on whether you specify the **Reset type** as `Synchronous` or `Asynchronous` on the model. If your **Reset type** is `Asynchronous`, the synchronization logic asserts the reset signal asynchronously and de-asserts the reset signal synchronously. For example, this code illustrates the generated Verilog code for the reset synchronization logic when you generate the IP core with asynchronous reset.

```
...
...

reg_reset_pipe_process : PROCESS (clk, reset_in)
  BEGIN
    IF reset_in = '1' THEN
      reset_pipe <= '1';
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        reset_pipe <= const_0;
      END IF;
    END IF;
  END PROCESS reg_reset_pipe_process;


  reg_reset_delay_process : PROCESS (clk, reset_in)
  BEGIN
    IF reset_in = '1' THEN
      reset_out <= '1';
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        reset_out <= reset_pipe;
      END IF;
    END IF;
  END PROCESS reg_reset_delay_process;
```

```
END rtl;
```

If your **Reset type** is `Synchronous`, the synchronization logic asserts and de-asserts the reset signal synchronously. For example, this code illustrates the generated Verilog code for the reset synchronization logic when you generate the IP core with synchronous reset.

```
...
...

reg_reset_pipe_process : PROCESS (clk)
  BEGIN
    IF clk'EVENT AND clk = '1' THEN
      IF reset_in = '1' THEN
        reset_pipe <= '1';
      ELSIF enb = '1' THEN
        reset_pipe <= const_0;
      END IF;
    END IF;
  END PROCESS reg_reset_pipe_process;


  reg_reset_delay_process : PROCESS (clk)
  BEGIN
    IF clk'EVENT AND clk = '1' THEN
      IF reset_in = '1' THEN
        reset_out <= '1';
      ELSIF enb = '1' THEN
        reset_out <= reset_pipe;
      END IF;
    END IF;
  END PROCESS reg_reset_delay_process;


END rtl;
```

See Also

More About

- “Custom IP Core Generation” on page 39-17
- “Custom IP Core Report” on page 39-20
- “Processor and FPGA Synchronization” on page 39-38

IP Caching for Faster Reference Design Synthesis

In this section...

"Requirements for Using IP Caching" on page 39-44
"What Is an IP Cache?" on page 39-44
"How IP Caching Works" on page 39-45
"Enable IP Caching" on page 39-45
"IP Caching in HDL Coder Reference Designs" on page 39-46
"IP Caching in Custom Reference Designs" on page 39-47

For target platforms that support the `IP Core Generation` workflow with Xilinx Vivado, you can use IP caching. IP caching reduces the synthesis time of reference designs that have many IP modules or that have IP modules with a significant synthesis run time. When you enable IP caching, the Vivado project uses an out-of-context (OOC) workflow. This workflow synthesizes the IP in the reference design out of context from the top-level design. The OOC workflow accelerates project runs because the synthesis tool reuses the IP cache, and does not have to resynthesize the IP when you run the workflow.

If you do not enable IP caching, by default, the Vivado project uses the global synthesis flow. This flow synthesizes the IP modules in the reference design along with the top-level design. In subsequent project runs, this workflow resynthesizes the IP modules in the reference design.

Requirements for Using IP Caching

- **Target workflow**:
  - `IP Core Generation`
  - `Simulink Real-Time FPGA I/O` for Speedgoat boards that use Xilinx Vivado
- **Synthesis tool**: Xilinx Vivado

What Is an IP Cache?

An IP cache is a folder that consists of subfolders corresponding to IP modules in the reference design. Each subfolder is organized by a hash index that corresponds to the file name. For each IP module, the subfolder consists of Xilinx Core Instance (XCI) files, Design Checkpoint (DCP) files, and synthesis log files. The DCP is a container file that contains synthesized netlists, black box HDL stub files, and the output clock constraints.

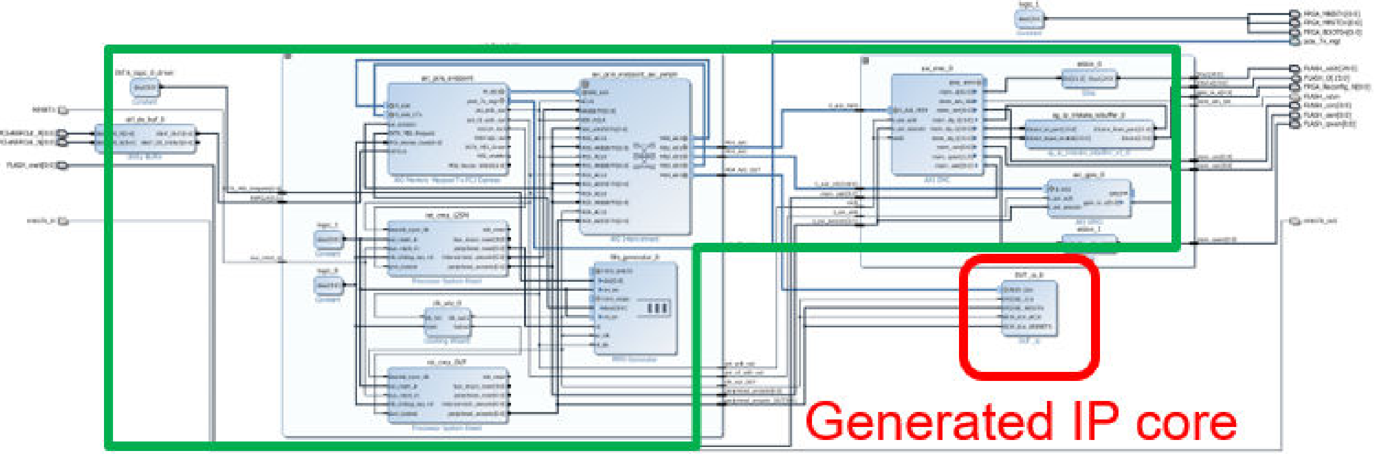
To reuse the IP cache when you run the workflow, the IP synthesis has to match the hash index in the IP cache. The hash index match corresponds to a hit in the IP cache. To hit the IP cache in subsequent runs, use the same:

- Part, language, and target platform settings
- Reference design version
- Target frequency
- `hdl_prj` folder when you created the IP cache

How IP Caching Works

When you enable IP caching, the Xilinx Vivado project uses an out-of-context (OOC) workflow. The OOC design flow is a bottom-up workflow that:

1. Synthesizes the IP modules in the reference design separately from the top-level design. The synthesis output is the Design Checkpoint (DCP) file.
2. Synthesizes your top-level design while treating the IP in the reference design as a black box by using the HDL stub files provided with the DCP.
3. Implements your design on the target device by linking the netlists from the IP design checkpoint files with your top-level netlist.

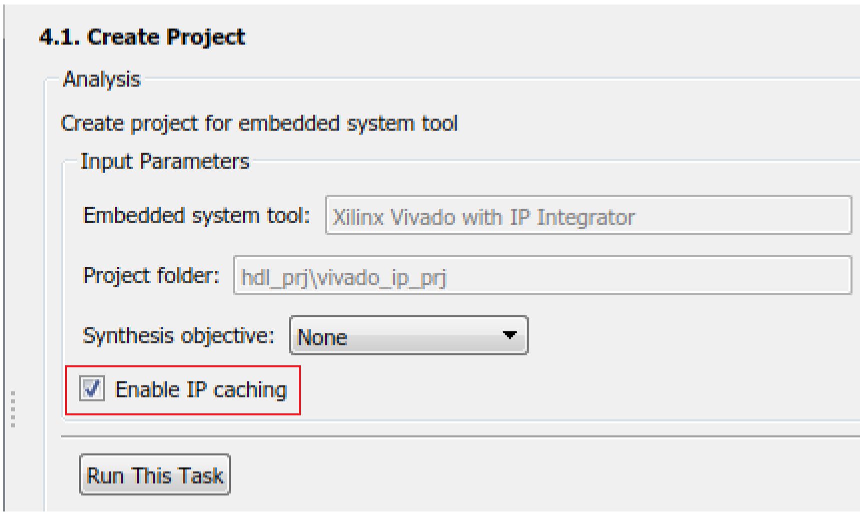


For large reference designs, the OOC flow improves synthesis run time, because you do not have to resynthesize the IP when you modify your design and run the workflow. To learn more about the OOC workflow and IP synthesis options, refer to the Xilinx documentation.

Enable IP Caching

Before you enable IP caching, specify `IP Core Generation` as the target workflow, and then specify the target platform settings. To enable IP caching:

- From the HDL Workflow Advisor, in the **Create Project** task, select the **Enable IP caching** check box.



- From the command line, use the `EnableIPCaching` property of the `hdlcoder.WorkflowConfig` class. To use this property, create an object of the `hdlcoder.WorkflowConfig` class, or export the HDL Workflow Advisor settings to a script.

```
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','IP Core Generation');
% ...
% ...
hWC.EnableIPCaching = true;
```

IP Caching in HDL Coder Reference Designs

Use IP caching for large reference designs that have a significant synthesis time. For example, the HDL Coder reference design `Default video system (requires HDMI FMC module)` is a potential candidate for IP caching.

Note The `Speedgoat IO333-325K` board that you use with the `Simulink Real-Time FPGA I/O` workflow comes with an IP cache. The first time that you run the workflow, the code generator reuses this IP cache, which improves reference design synthesis time.

To enable IP caching, in the HDL Workflow Advisor, specify `IP Core Generation` as the target workflow, and then specify the target platform settings. Before you run the workflow for the first time:

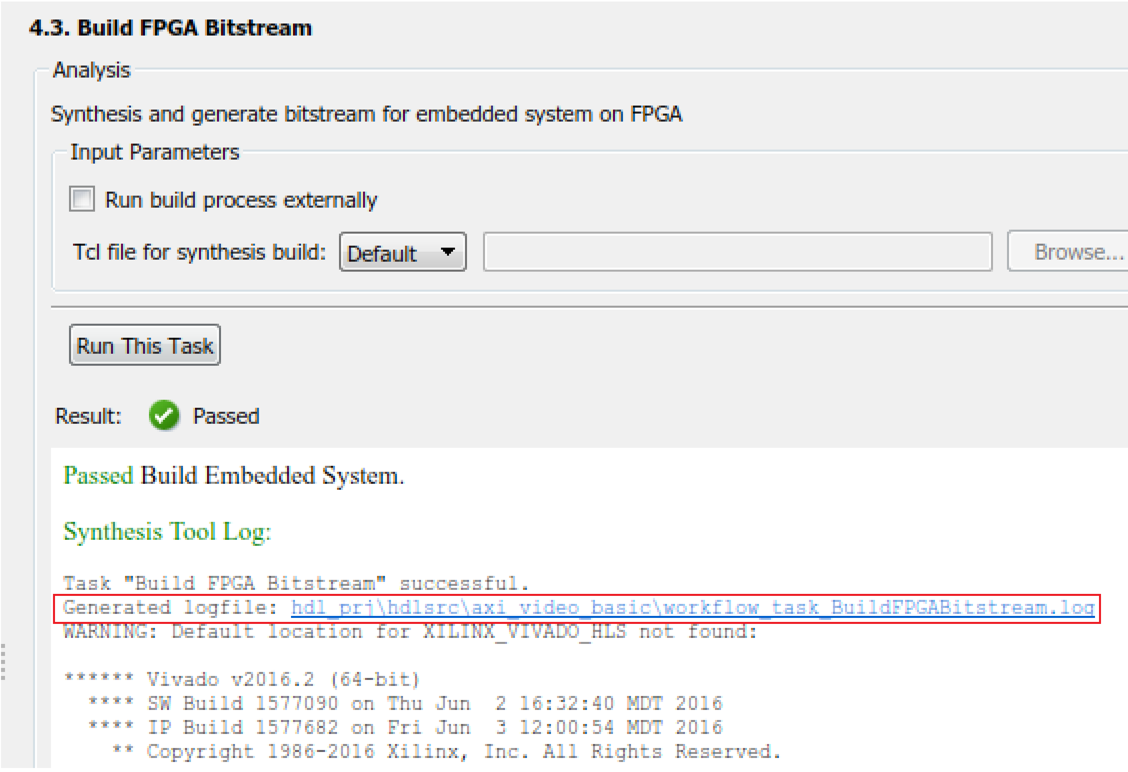
1. In the **Create Project** task, select the **Enable IP caching** check box.

   When you run this task, the workflow creates an empty IP cache folder. You can see the `ipcache` folder in the `hdl_prj/vivado_ip_prj` path.
2. Run the **Build FPGA Bitstream** task.

   This task populates the IP cache folder with synthesis logs and design checkpoint files generated for the HDL IP core and other IP blocks in the reference design. When this task has run successfully, you can see the generated files in the `ipcache` folder.

When you run the `IP Core Generation` workflow a second time, in the **Build FPGA Bitstream** task, you can see an improvement in the task run time. Make sure that you use the same IP settings

and `hdl_prj` folder as the first time that you ran the workflow. When this task has run successfully, to see if your workflow reused the IP cache, open the `workflow_task_buildFPGABitstream.log` file.



This code snippet shows that the Vivado project launches a maximum number of jobs to synthesize the design and reuse the IP modules in the IP cache folder. You can see that the `cacheID` of the IP modules match the file names of the subfolders in the `ipcache` folder.

```
...
# reset_run impl_1
# reset_run synth_1
# launch_runs -jobs 4 synth_1
...
...
...
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_RGBtoYCbCr_0_0, cacheID = 3575924730488800
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_YCbCrtoRGB_0_0, cacheID = e71459f41e26e141
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_xbar_0, cacheID = d0f0971cb77bcaed
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_axis2hdmi_0_0, cacheID = 7601a322f9fd0ec4
...
```

IP Caching in Custom Reference Designs

If you are using your own custom reference design, IP caching can accelerate reference design synthesis when you run the workflow for the first time. To reuse the IP cache, create an IP cache zip file, and then make sure that the reference design definition file points to this zip file.

To create an IP cache zip file:

1. Open the HDL Workflow Advisor for any Simulink model that has a DUT subsystem, and then run the **IP Core Generation** workflow to the **Generate RTL Code and IP Core** task.

2. In the **Create Project** task, select the **Enable IP caching** check box, and then click **Run This Task**. This task creates an empty IP cache folder.
3. Run the workflow to the **Build FPGA Bitstream** task. This task populates the IP cache with the HDL IP core and the reference design IP modules.
4. In the IP cache folder, delete the IP core files generated for the DUT. Extract the remaining files from this folder into a zip file, name it `ipcache.zip`, and then save the file in the reference design folder.

To reuse the IP cache, in the reference design definition file `plugin_rd.m`, use the `IPCacheZipFile` property of the `hdlcoder.ReferenceDesign` class. By using that property, you add the `ipcache.zip` file to the Xilinx Vivado project.

```
function hRD = plugin_rd()
% Reference design definition

hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
% ...
% ...
hRD.IPCacheZipFile = 'ipcache.zip';
```

When you use the workflow to target your custom reference design, the code generator selects the **Enable IP caching** check box. To see the improvement in synthesis time, run the **Build FPGA Bitstream** task.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

More About

- “Custom IP Core Generation” on page 39-17
- “Custom IP Core Report” on page 39-20
- “Board and Reference Design Registration System” on page 40-89
- “Run HDL Workflow with a Script” on page 29-47

Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows

In this section...

"Step 1: Identify the Timing Failure" on page 39-49
"Step 2: Find the Critical Path" on page 39-52
"Step 3: Resolve Timing Failures" on page 39-56

Synchronous circuits require that data propagates from a source register to a destination register within one clock cycle. For the synthesis tools, the Delay blocks that you add to your Simulink model run at the clock rate. The tools require data to travel between the blocks within one clock cycle. If the tool is unable to propagate the data between the registers for one or more signal paths in your model within one clock cycle, a timing failure occurs.

The tools report a slack information for each signal path, which corresponds to the difference between the required time and the arrival time. Required time is the expected time at which a signal must arrive at the destination register. Arrival time is the time elapsed for a signal to arrive at that point. A positive slack indicates that the signal arrived much faster than the required time, and the path passes the timing requirement. A negative slack indicates that the signal path is slower than the required time, and the path fails the timing requirement. To make sure that your design meets the timing requirements, speed up all signal paths that have a negative slack.

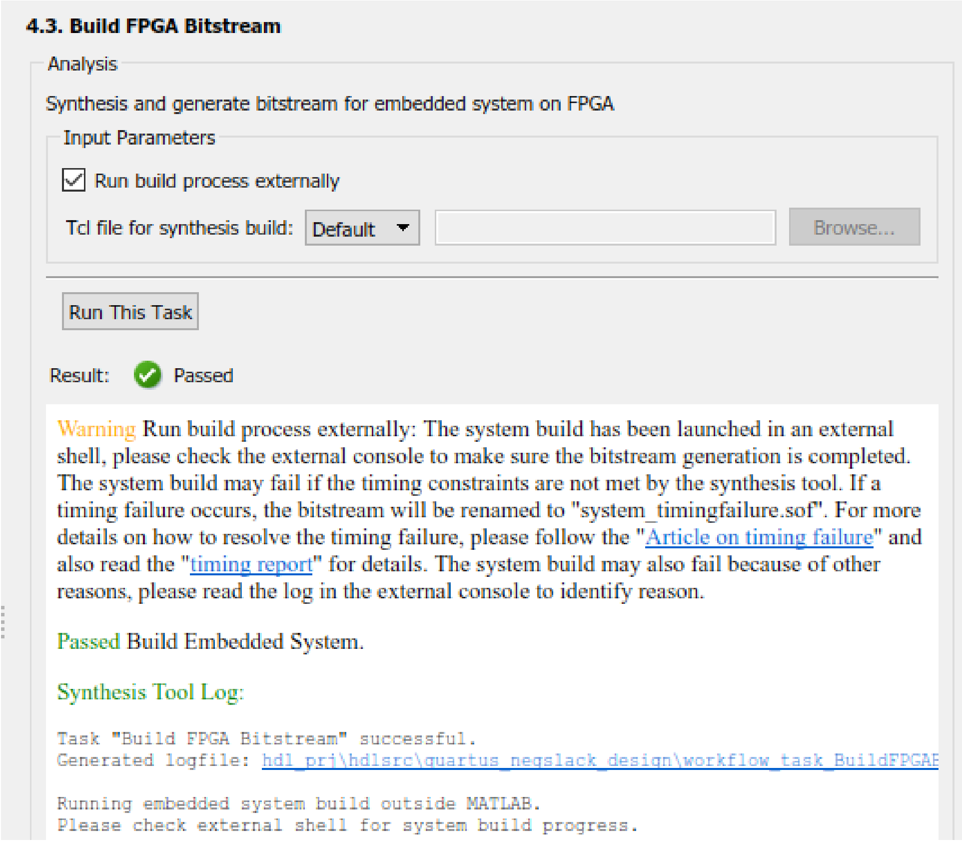
To identify if your design meets the timing requirements and how you can resolve timing failures, perform these steps.

Step 1: Identify the Timing Failure

When you run the `IP Core Generation` workflow or the `Simulink Real-Time FPGA I/O` workflow for Vivado-based boards, if your Simulink model does not meet the timing requirements, HDL Coder generates an error in the **Build FPGA Bitstream** step of the workflow. See:

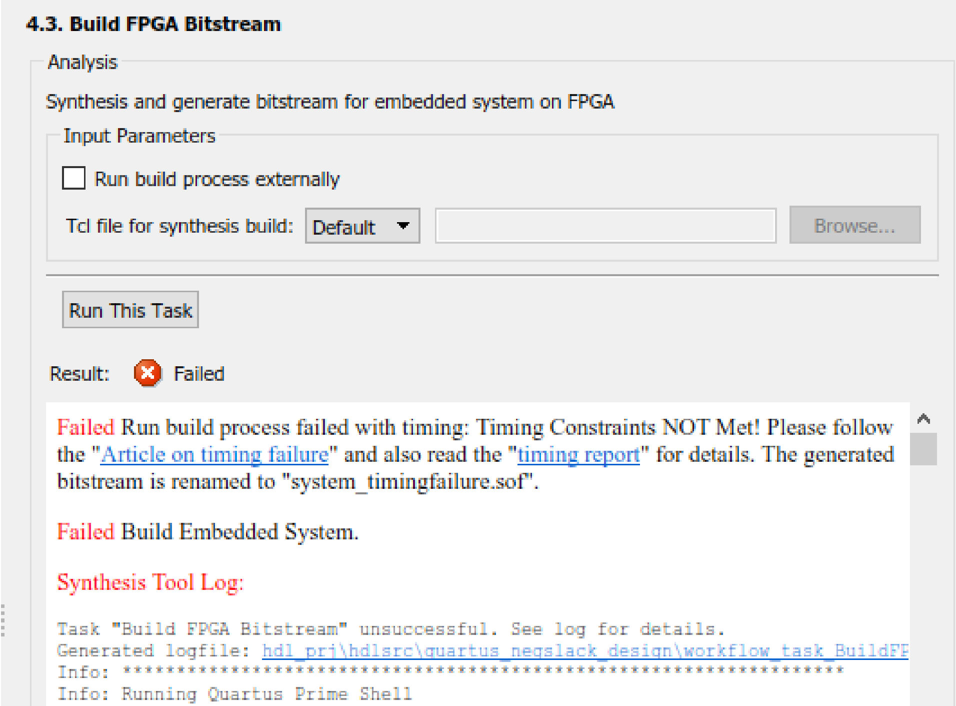
- "Getting Started with Targeting Xilinx Zynq Platform" on page 39-98 for information about how to run the `IP Core Generation` workflow.
- "FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules" on page 40-113 for information about how to run the `Simulink Real-Time FPGA I/O` workflow. When you run this workflow, use a Speedgoat board that supports Xilinx Vivado as the synthesis tool.

When you run the **Build FPGA Bitstream** task, if you left the **Run build process externally** check box selected by default, whether or not there is a timing failure, HDL Coder displays the results as **Passed** and provides warning messages. View the build log in the external console to identify if there is a potential timing failure.



In the external console, if there is a timing failure, you see the worst slack and this message: `Timing constraints NOT met!`

When you clear the **Run build process externally** check box, and then run the **Build FPGA Bitstream** task, if a timing failure occurs, the task fails, and you see these messages in the **Result** subpane.



In both cases, when there is a timing failure, the code generator replaces the previous bitstream with a bitstream that has the same name and the postfix `_timingfailure.bit` or `_timingfailure.sof` depending on whether you created a project by using Vivado or Quartus. For example, if the previous generated bitstream was called `system_top_wrapper.bit`, and if there is a timing failure, HDL Coder renames this bitstream to `system_top_wrapper_timingfailure.bit`.

If you run the **Program Target Device** task, the task fails.

Report Timing Failures as Warnings

If you have already implemented the custom logic to resolve the timing failures, you can specify the timing failures to be reported as warnings instead of errors. You can then continue the workflow and generate the FPGA bitstream. Before programming the target SoC device, it is recommended that you have resolved the timing failures.

After you have resolved the timing failures, to verify that the failures have been resolved, you can use the HDL Coder software. Change the timing failures to be reported as errors and then rerun the `IP Core Generation` workflow to ensure that the **Build FPGA Bitstream** task passes. If the **Build FPGA Bitstream** task still fails, perform the steps in the preceding sections to identify and resolve the timing failures.

To specify timing failures to be reported as warnings:

- After you run the **Build FPGA Bitstream** task, export the HDL Workflow Advisor to a script. In the script, to report timing failures as warnings, use the `ReportTimingFailure` property of the `hdlcoder.WorkflowConfig` class. You can then run the script or import the script to the HDL Workflow Advisor and then run the workflow.

  ```
  hWC.ReportTimingFailure = hdlcoder.ReportTiming.Warning;
  ```

- If you are targeting a custom reference design that you have already defined for the board, to report timing failures as warnings, use the `ReportTimingFailure` property of the `hdlcoder.ReferenceDesign` class.

  ```
  hRD.ReportTimingFailure = hdlcoder.ReportTiming.Warning;
  ```

To learn how you can identify the critical path and resolve the timing failures, perform the steps in the preceding sections.

Step 2: Find the Critical Path

Critical path is a combinational path between the input and the output that has the maximum delay. This path corresponds to the signal path that has the worst negative slack. By identifying and optimizing the critical path, you can resolve timing failures and improve the timing of your design. You can identify the critical path in your design by using either of these strategies.

Strategy 1: Check the Timing Report

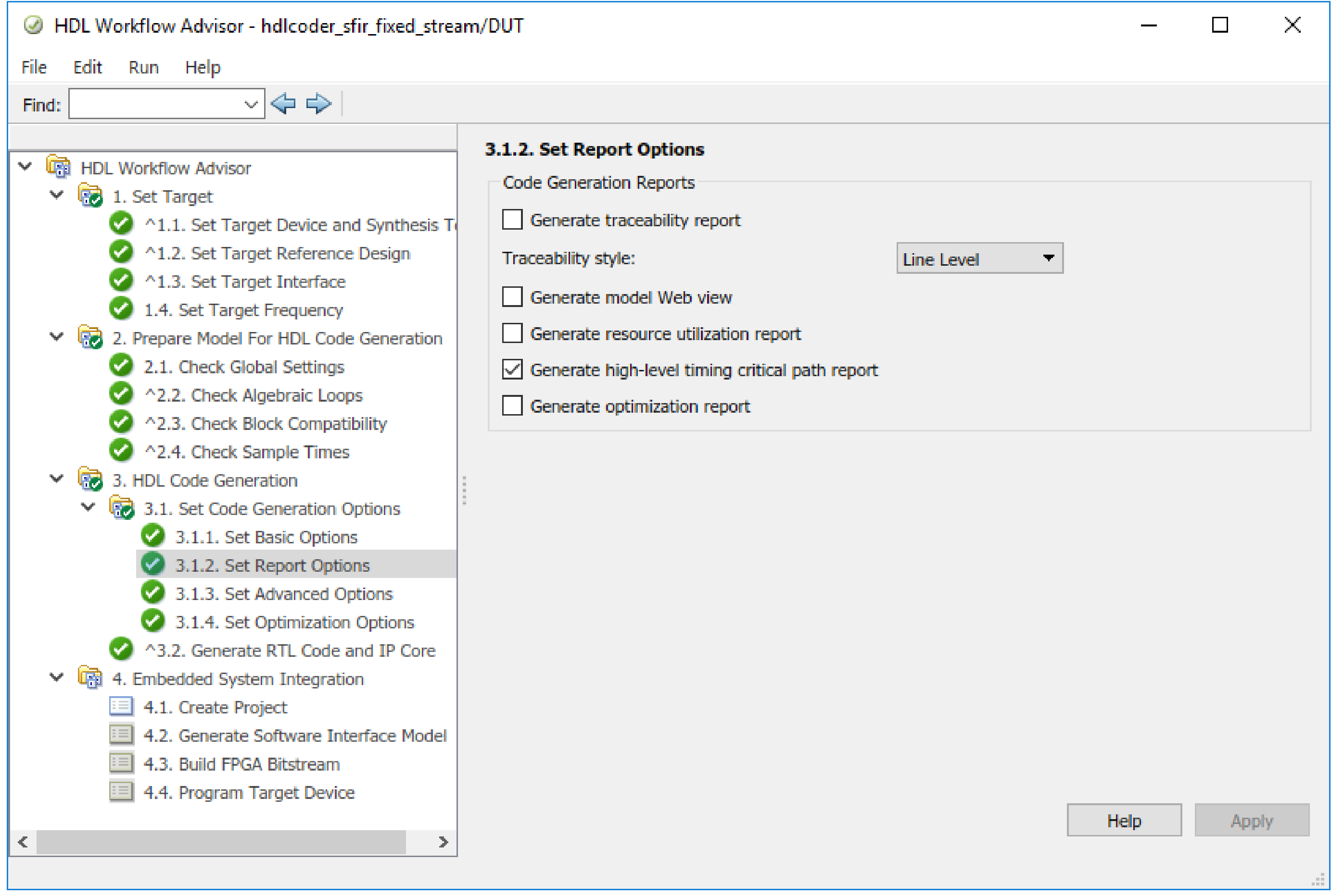
In the **Result** subpane, to open the timing report that is generated by the synthesis tool, select the **timing_report** link. You can use the report to identify the critical path in your design. In the report, if you search for `Worst Slack`, you can identify the worst setup slack. Then, use the `Source` and `Destination` points to identify the critical path. For example, this report for the LED blinking model `hdlcoder_led_blinking` shows that the critical path is inside the HDL Counter block.

```
---------------------------------------------------------------------------------------
From Clock:  clk_out1_system_top_clk_wiz_0_0
  To Clock:  clk_out1_system_top_clk_wiz_0_0

Setup :   1193  Failing Endpoints,  Worst Slack   -2.478ns,  Total Violation  -1226.784ns
Hold  :      0  Failing Endpoints,  Worst Slack    0.034ns,  Total Violation      0.000ns
PW    :      2  Failing Endpoints,  Worst Slack   -0.576ns,  Total Violation     -0.731ns
---------------------------------------------------------------------------------------

Max Delay Paths
---------------------------------------------------------------------------------------
Slack (VIOLATED) : -2.478ns  (required time - arrival time)
  Source:     system_top_i/led_count_ip_0/U0/u_led_count_ip_dut_inst/
                   u_led_count_ip_src_led_counter/HDL_Counter1_out1_reg[0]/C
              (rising edge-triggered cell FDRE clocked by clk_out1_system_top_clk_wiz_0_0
                                         {rise@0.000ns fall@1.000ns period=2.000ns})
 Destination:  system_top_i/led_count_ip_0/U0/u_led_count_ip_dut_inst/
                    u_led_count_ip_src_led_counter/HDL_Counter1_out1_reg[20]/R
               (rising edge-triggered cell FDRE clocked by clk_out1_system_top_clk_wiz_0_0
                                          {rise@0.000ns fall@1.000ns period=2.000ns})
  Path Group:       clk_out1_system_top_clk_wiz_0_0
  Path Type:        Setup (Max at Slow Process Corner)
  Requirement:      2.000ns  (clk_out1_system_top_clk_wiz_0_0 rise@2.000ns -
                                      clk_out1_system_top_clk_wiz_0_0 rise@0.000ns)
  Data Path Delay:  3.899ns  (logic 1.412ns (36.211%)  route 2.487ns (63.789%))
```

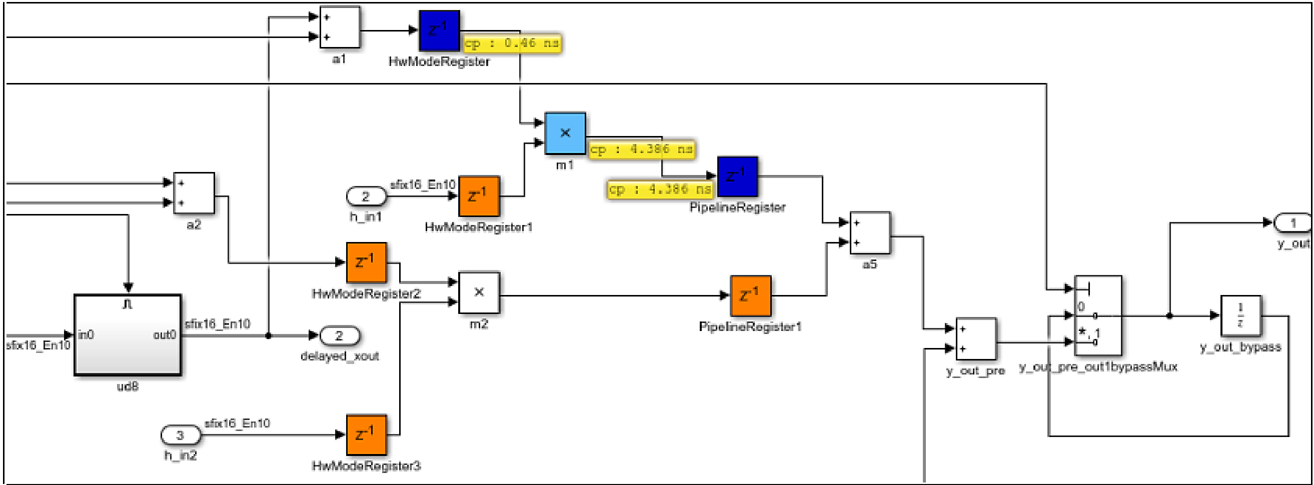
Strategy 2: Estimate Critical Path Without Running Synthesis

Use HDL Coder to estimate and highlight the critical path in your model without synthesizing your design. Critical path estimation identifies the critical path by performing static timing analysis with timing data from target-specific databases. Estimating the critical path without using synthesis tools can lead to inaccurate timing results. Critical path estimation speeds up the process of identifying and optimizing the critical path in your design. It is an alternative to performing **FPGA Synthesis and Analysis** with the HDL Workflow Advisor. To learn more, see “Critical Path Estimation Without Running Synthesis” on page 21-192.

To estimate the critical path, in the **Set Report Options** task, select the **Generate high-level timing critical path report** check box. Run the workflow to the **Generate RTL Code and IP Core** task.



HDL Coder generates a critical path estimation section in the Code Generation Report. On this section, when you select the link to the `criticalpathestimated` script, the code generator highlights the critical path in the generated model. This figure shows a section of the `hdlcoder_sfir_fixed_stream` model with the critical path highlighted.



Strategy 3: Annotate Critical Path By Using Backannotation

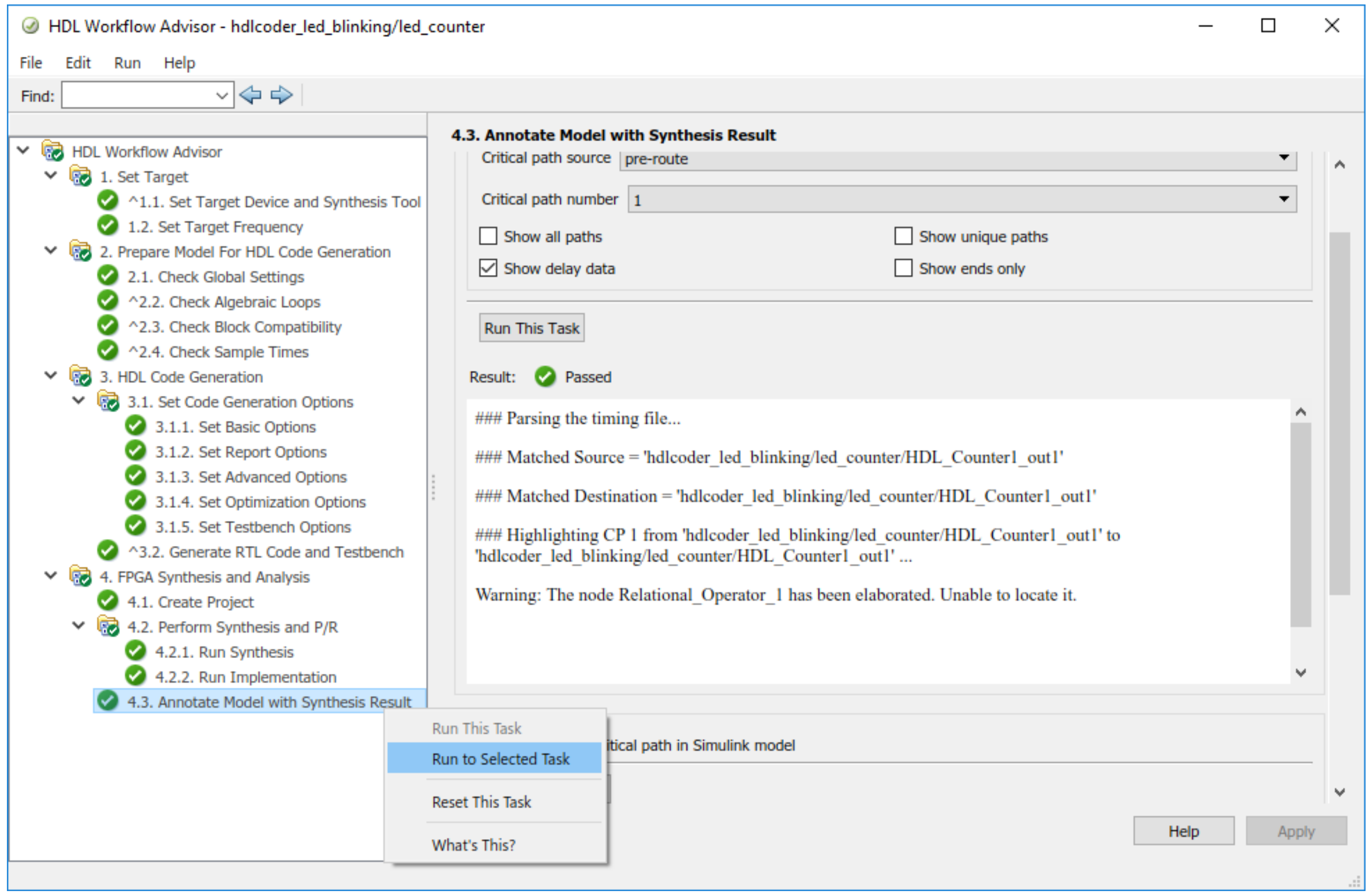
For more accurate critical path information and highlighting of critical path in your design, use backannotation. To use backannotation, you have to leave the current Workflow Advisor session, and then run the `Generic ASIC/FPGA` workflow to annotate the model with the synthesis results.

Before you use backannotation, it is recommended that you export the current HDL Workflow Advisor settings to a script. By exporting the settings to a script, you can iterate on the critical path and customize various settings to optimize your design until you meet the timing requirements. You can import the Workflow Advisor script to the HDL Workflow Advisor and then run the workflow. See also “Run HDL Workflow with a Script” on page 29-47.

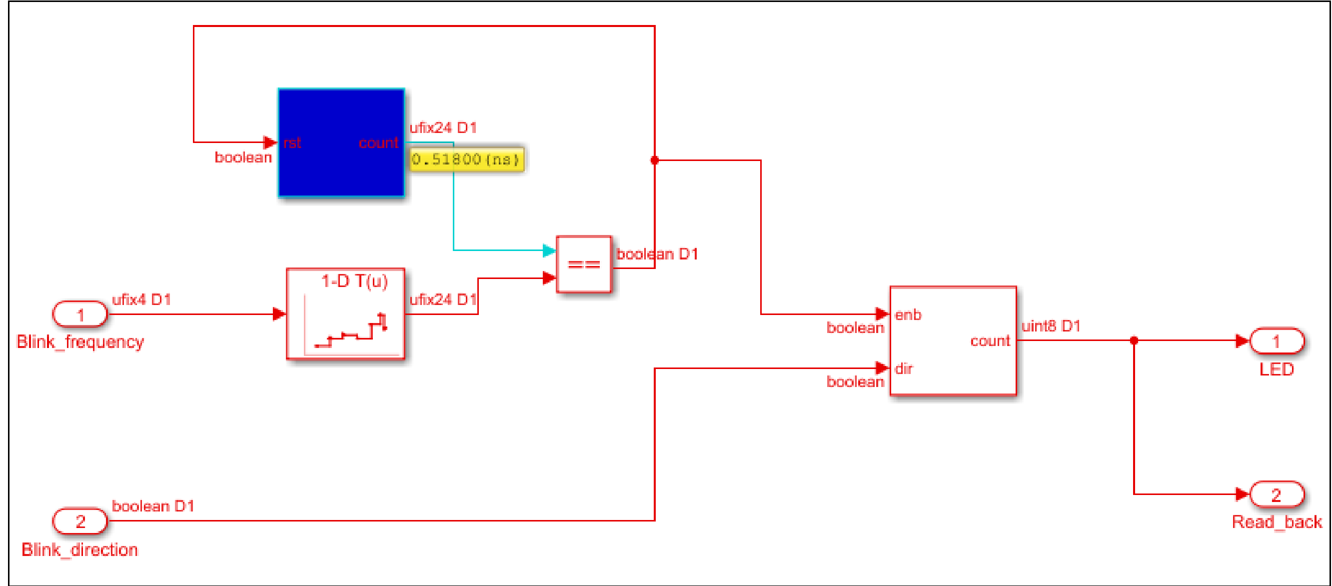
To use backannotation:

1. In the **Set Target Device and Synthesis Tool** task, select `Generic ASIC/FPGA` as the **Target workflow**. For **Synthesis tool**, specify the same tool that you used to run the `IP Core Generation` workflow.

   When you specify these settings, HDL Coder resets this task and all tasks that follow it.
2. Select **Run This Task**.
3. In the **Set Target Frequency** task, specify the same target frequency that you used to run the `IP Core Generation` workflow. Select **Run This Task**.
4. Right-click the **Annotate Model with Synthesis Result** task and select **Run to Selected Task**.



When you run the link to the **Annotate Model with Synthesis Result** task, the code generator highlights the critical path in the generated model. This figure shows that the critical path in the `hdlcoder_led_blinking` model is inside the HDL Counter block.



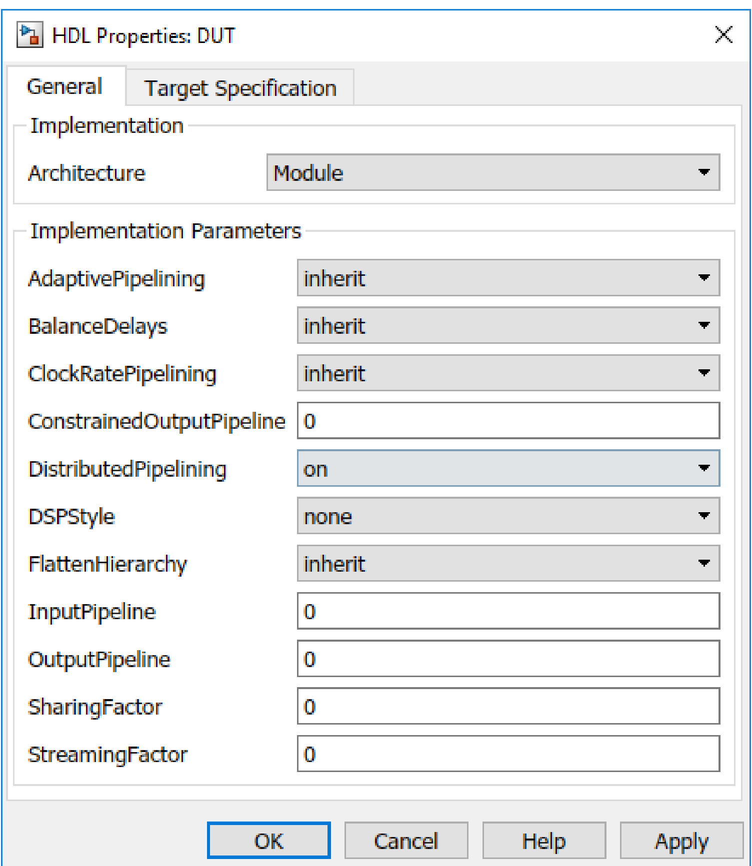
Step 3: Resolve Timing Failures

To resolve timing failures, you can use any of these recommendations or a combination of the recommendations until your design meets the target frequency.

Recommendation 1: Use Speed Optimizations

You can use speed optimizations such as distributed pipelining and clock-rate pipelining to break the critical path by adding pipeline registers while preserving the functional behavior. By reducing the critical path, you can achieve higher clock frequencies and increase the arrival time of the signal until it equals the required time and the slack becomes zero.

Distributed pipelining optimization move the existing delays you have in your design across the subsystem hierarchy. To enable distributed pipelining on a subsystem, set **DistributedPipelining** as `on`. If your design does not meet the timing requirements, you can add more pipelines by using **InputPipeline** or **OutputPipeline** block properties. You can specify these properties in the HDL Block Properties dialog box of the Subsystem.

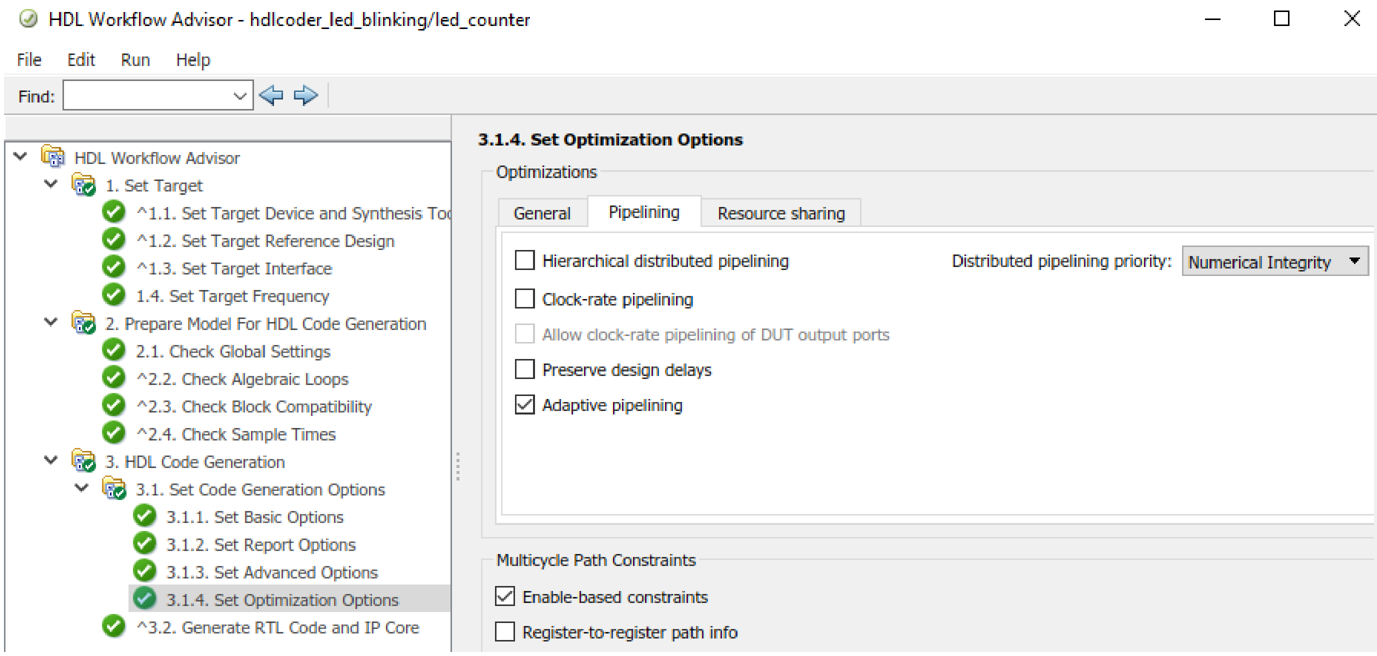


If distributed pipelining is unable to move the registers, you can add Delay blocks to your model, and then disable the **Allow design delay distribution** setting. Reset the **Check Global Settings** task and run the workflow to the **Build FPGA Bitstream** task. You can iterate on this approach and use other optimizations such as clock-rate pipelining with a large value for the **Oversampling factor** if the design does not meet the timing requirements. To specify these settings, use the **Pipelining** tab of the **Set Optimization Options** task in the HDL Workflow Advisor. For more information, see “Speed Optimization”.

Recommendation 2: Specify Enable-Based Multicycle Path Constraints

If your design contains multiple sample rates or uses certain HDL block implementations or speed optimizations that insert pipeline registers at a faster rate after code generation, your design can have multicycle paths. By default, HDL Coder uses a single clock mode that generates a primary clock at the fastest sample rate and creates a timing controller entity. The timing controller generates a set of clock enables with the required rate and phase information to sample the clock signal for blocks that operate at a slower sample time.

If your critical path is on a slower signal rate, synthesis tools can fail to meet the timing requirements. A timing failure occurs because the tools cannot infer the sample rates from the generated HDL code and assume that these paths have to run at the fastest rate. You can use enable-based multicycle path constraints to generate a constraints file that enables the synthesis tool to ease the clock constraint on the multicycle paths. To specify generation of multicycle path constraints, in the **Set Optimization Options** task, select the **Enable-based constraints** check box. Run the workflow to the **Build FPGA Bitstream** task. For an example, see “Use Multicycle Path Constraints to Meet Timing for Slow Paths” on page 20-38.



Recommendation 3: Reduce the Target Frequency

Use the **Target Frequency (MHz)** setting to specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. See also Target Frequency.

To resolve timing failures, reduce the **Target Frequency (MHz)** setting so that the synthesis tool can meet the timing constraint. To see what target frequency you can specify, use the slack and the critical path information from the synthesis tool timing report. Because slack is equal to the difference between the required time and arrival time, you can add the slack to the required time, and then use this sum as the `New required time`. Use the reciprocal of this `New required time` as the target frequency value to meet the timing requirements because the `New required time` equals the arrival time. To compute the target frequency, in the MATLAB Command Window, run this script.

```
% Specify the required time (ns) and slack (ns) using timing report
required_time = 2;
slack = 2.2;

% Slack is difference between required_time and arrival_time
% By adding slack to required_time you can resolve
New_required_time = required_time + slack;
Target_frequency = 1 / (New_required_time);

% Since we computed the new time in nanoseconds
Target_frequency_MHz = Target_frequency * 10^3;
```

In the **Set Target Frequency** task, specify this value for **Target Frequency (MHz)**, and then run the workflow to the **Build FPGA Bitstream** task. If you see a timing failure, you can use this approach to iterate on the target frequency value until your design meets the timing requirements and the slack becomes zero.

You can also export the HDL Worflow Advisor settings to a script and keep iterating on the target frequency value by specifying `Target_frequency_MHz` as the value for the `TargetFrequency` property. Then, run the script.

```
% Set this frequency as the new target frequency
hdlset_param('hdlcoder_led_blinking', 'TargetFrequency', Target_frequency_MHz);
```

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

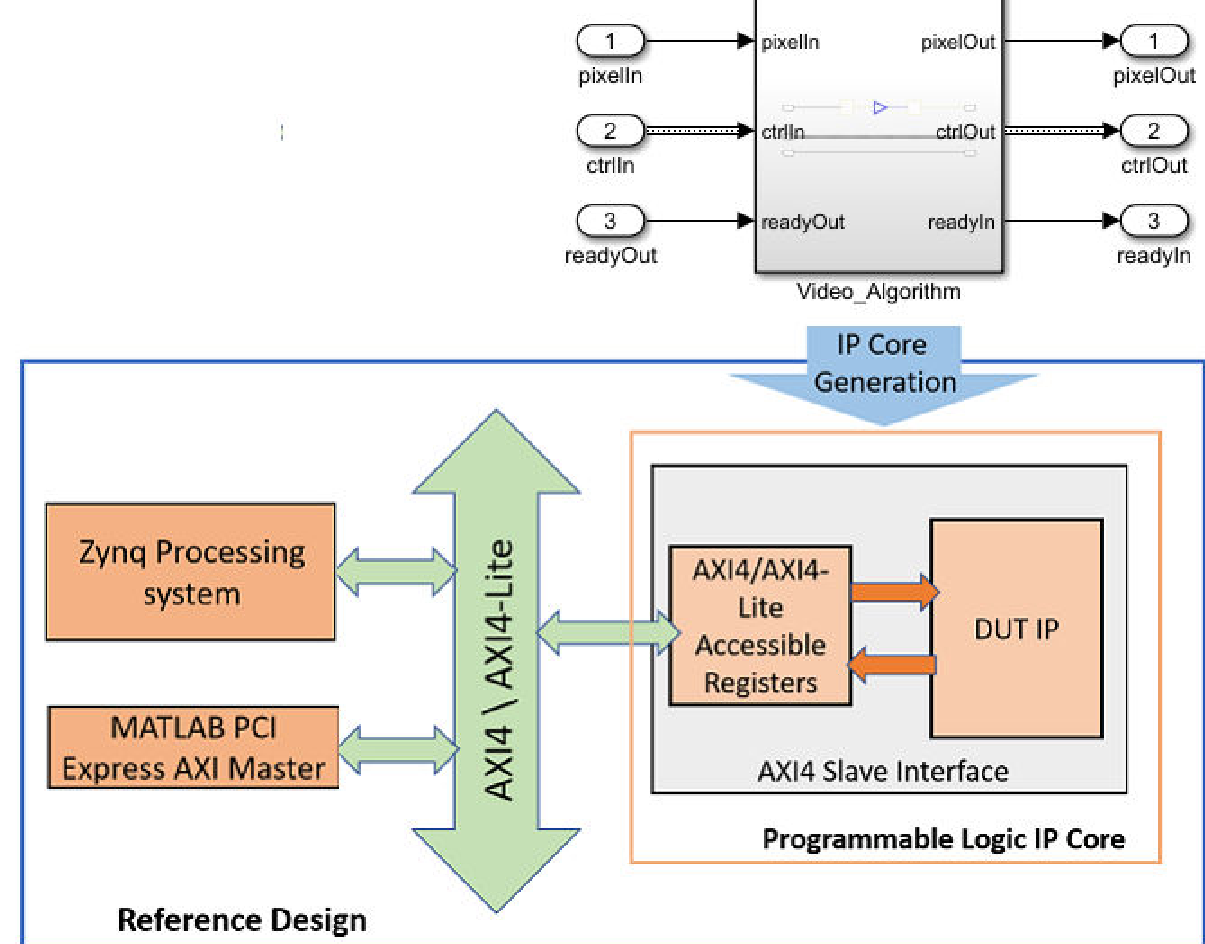
More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- “Custom IP Core Generation” on page 39-17
- “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 40-145
- “Program Target FPGA Boards or SoC Devices” on page 39-64

Define Multiple AXI Master Interfaces in Reference Designs to Access DUT AXI4 Slave Interface

In this section...
“Vivado-Based Reference Designs” on page 39-61
“Qsys-Based Reference Designs” on page 39-62

You can define multiple AXI Master interfaces in your custom reference design and access the AXI4 slave interfaces in the generated HDL DUT IP core for the DUT. This capability enables you to simultaneously connect the HDL DUT IP core to two or more AXI Master IP in the reference design, such as the HDL Verifier JTAG AXI Manager IP and the ARM processor in the Zynq processing system.



Vivado-Based Reference Designs

To define multiple AXI Master interfaces, you specify the `BaseAddressSpace` and `MasterAddressSpace` for each AXI Master instance, and also the `IDWidth` property.

`IDWidth` is the width of all ID signals, such as `AWID`, `WID`, `ARID`, and `RID`, specified as a positive integer. By default, the `IDWidth` is `12`, which enables you to specify one AXI Master interface connection to the DUT IP core. To connect the DUT IP core to multiple AXI Master interfaces, you may have to increase the `IDWidth`. The `IDWidth` value is tool-specific. To see the value that you must use when specifying more than one AXI Master interface, refer to the documentation for that tool. If you use an incorrect ID width, the synthesis tool generates an error, and reports the correct `IDWidth` that you must use.

This code is the syntax for the `MasterAddressSpace` field when specifying multiple AXI Master interfaces in Vivado-based reference designs:

```
'MasterAddressSpace', ...
   {'AXI Master Instance Name1/Address Space of Instance Name1', ...
    'AXI Master Instance Name2/Address_Space of Instance Name2',...};
```

For example, this code illustrates how you can modify the `plugin_rd` file to define two AXI Master interfaces.

```
% ...

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'xilinx.com:zc706:part0:1.0');


% ...
% ...

% The DUT IP core in this reference design is connected
% to both Zynq Processing System and the AXI Manager IP.
% Because of 2 AXI Master, ID width has to be increased
% from 12 to 13.
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_interconnect_0/M00_AXI', ...
    'BaseAddress',         {'0x40010000', '0x40010000'}, ...
    'MasterAddressSpace',  {'processing_system7_0/Data', 'hdlverifier_axi_manager_0/axi4m'}, ...
    'IDWidth',             13);

% ...
```

In this example, the two AXI Master IP are the HDL Verifier AXI Manager IP and the ARM processor. Based on the syntax of the `MasterAddressSpace`, for the HDL Verifier AXI Manager IP, the `AXI Master Instance Name` is `hdlverifier_axi_manager_0` and the `Address_Space of Instance Name` is `axi4m`.

The AXI4 slave interfaces in the HDL DUT IP core connect to the Xilinx AXI Interconnect IP that is defined by the `InterfaceConnection` property of the `addAXI4SlaveInterface` method. The AXI4 slave interfaces have a `BaseAddress`. This `BaseAddress` must map to the `MasterAddressSpace` for the two AXI Master IP, which is specified as a cell array of character vectors.

You must make sure that the AXI Master IPs have already been included in the Vivado reference design project. `system_top.tcl` is the TCL file that is defined by the `CustomBlockDesignTcl`

property of the `addCustomVivadoDesign` method. In this TCL file, you must make sure that the two AXI Master IP are connected to the same Xilinx AXI Interconnect IP. The interconnects then connect the AXI Master IPs to the AXI4 slave interfaces in the HDL IP core.

After you run the `IP Core Generation` workflow and create the Vivado project, open the project. In the Vivado project, if you open the block design, you see the two AXI Master IP connected to the HDL DUT IP core. If you select the **Address Editor** tab, you see the AXI Master instance names and the corresponding address spaces.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
hdlverifier_axi_master_0					
axi4m (32 address bits : 4G)					
led_count_ip_0	AXI4_Lite	reg0	0x4001_0000	64K	0x4001_FFFF
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
led_count_ip_0	AXI4_Lite	reg0	0x4001_0000	64K	0x4001_FFFF

Qsys-Based Reference Designs

To define multiple AXI Master interfaces, you specify the `InterfaceConnection` and `BaseAddressSpace` for each AXI Master instance, and also the `IDWidth` property. This code is the syntax for the `InterfaceConnection` field when specifying multiple AXI Master interfaces in Qsys-based reference designs:

```
'InterfaceConnection', ...
   {'AXI Master Instance Name1/Port name of Instance Name1', ...
    'AXI Master Instance Name2/Port name of Instance Name1', ...};
```

For example, this code illustrates how you can modify the `plugin_rd` file to define three AXI Master interfaces.

```
% ...

%% Add custom design files
% add custom Qsys design
hRD.addCustomQsysDesign('CustomQsysPrjFile', 'system_soc.qsys');
hRD.CustomConstraints = {'system_soc.sdc','system_setup.tcl'};


% ...

% add AXI4 slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', {'hps_0.h2f_axi_master','master_0.master','AXI_Manager_0.axm_m0'}, ...
    'BaseAddress',         {'0x0000_0000','0x0000_0000','0x0000_0000'},...
    'InterfaceType',       'AXI4'...
    'IDWidth',             14);

% ...
```

Based on the syntax of the `InterfaceConnection` option, for the HDL Verifier AXI Manager IP, the `AXI Master Instance Name` is `AXI_Manager_0` and the `Port name` is `axm_m0`. For each AXI Master IP, the `BaseAddress` of the HDL IP core and `InterfaceConnection` must be specified as a cell array of character vectors.

You must make sure that the AXI Master IPs have already been included in the Qsys reference design project. `system_soc.qsys` is the file that is defined by the `CustomQsysPrjFile` property of the `addCustomQsysDesign` method. In this file, you must make sure that the two AXI Master IP are connected to the same Qsys AXI Interconnect IP.

The interconnects then connect the AXI Master IPs to the AXI4 slave interfaces in the HDL IP core.

After you run the `IP Core Generation` workflow and create the Quartus project, open the project. In the Quartus project, you see the three AXI Master IP and the AXI Master interfaces connected to the HDL IP core for the DUT. If you select the **Address Map** tab, you see the AXI Master instance names, the port names, and the corresponding address spaces.

System: system_soc **Path:** hps_0

	MATLAB_as_AXI_Master_0.axm_m0	hps_0.h2f_axi_master	hps_0.h2f_lw_axi_master	master_0.master
hps_0.f2h_axi_slave				
led_count_ip_0.s_axi	0x0000_0000 - 0x0000_ffff	0x0000_0000 - 0x0000_ffff		0x0000_0000 - 0x0000_ffff

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270

More About

- “Board and Reference Design Registration System” on page 40-89
- “Register a Custom Board” on page 40-92
- “Register a Custom Reference Design” on page 40-95

Program Target FPGA Boards or SoC Devices

In this section...
“How to Program Target Device” on page 39-64
“Programming Methods” on page 39-65

To configure or program the connected target SoC device or FPGA board, use the `IP Core Generation` workflow in the HDL Workflow Advisor.

How to Program Target Device

Using the Workflow Advisor UI

1. Open the HDL Workflow Advisor. Right-click the DUT Subsystem that contains the algorithm to be deployed on the target FPGA, and select **HDL Code > HDL Workflow Advisor**.
2. In the **Set Target Device and Synthesis Tool** task, specify `IP Core Generation` as the **Target workflow**, and specify a **Target platform** other than the `Generic Xilinx Platform` or `Generic Altera Platform`.
3. Right-click the **Program Target Device** task and select **Run to Selected Task**.
4. In the **Program Target Device** task, specify the **Programming method**, and run this task.

To learn more about the HDL Workflow Advisor, see “Getting Started with the HDL Workflow Advisor” on page 29-5.

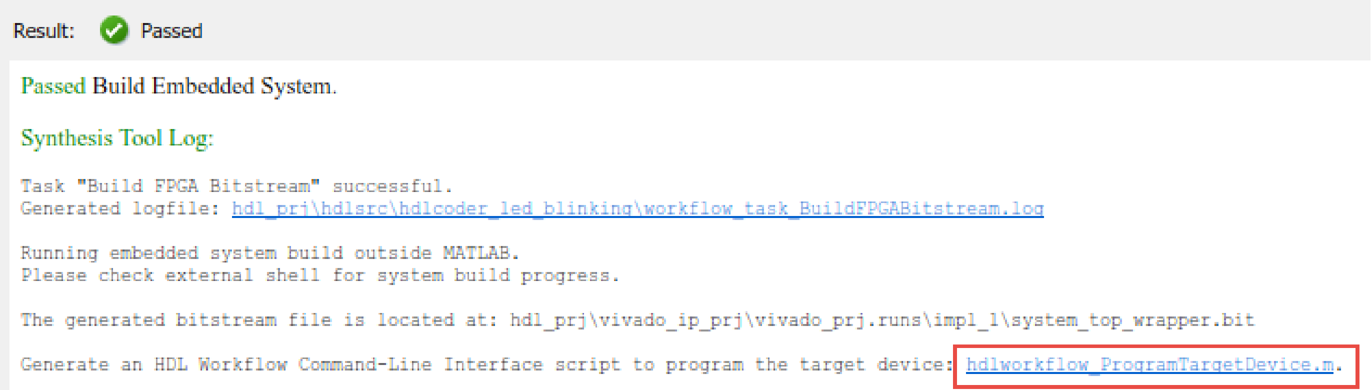
Using the Workflow Advisor Script

To program the target device at the command line, after you specify the target workflow and target platform in the **Set Target Device and Synthesis Tool** task, export the HDL Workflow Advisor settings to a script. In the HDL Workflow Advisor window, select **File** > **Export to Script**. The script creates and configures an `hdlcoder.WorkflowConfig` object that is denoted by `hWC`.

Before you run the script, you can specify how to program the target hardware by using the `ProgrammingMethod` property of the `WorkflowConfig` object. This code snippet shows an example script that is exported from the HDL Workflow Advisor when you use the default `Download` **Programming method**. To run the **Program Target Device** task, customize this script by setting the `RunTaskProgramTargetDevice` attribute of the `WorkflowConfig` object to `true`.

```
% This script was generated using the following parameter values:

% ...

% Set properties related to 'RunTaskProgramTargetDevice' Task
hWC.RunTaskProgramTargetDevice = true;
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.Download;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC);
```

After you run the **Build FPGA Bitstream** task, the Workflow Advisor provides you a link to generate a Workflow script that programs the target device without rerunning the previous tasks in the Workflow Advisor.



Click the link to open the script in the MATLAB Editor. This code snippet shows an example script that is generated by the HDL Workflow Advisor. If close the HDL Workflow Advisor, you can run the script to program the target hardware without running other workflow tasks.

```
% Load the Model

% ...

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = false;
hWC.RunTaskCreateProject = false;
hWC.RunTaskGenerateSoftwareInterface = false;
hWC.RunTaskBuildFPGABitstream = false;
hWC.RunTaskProgramTargetDevice = true;

% Set properties related to 'RunTaskProgramTargetDevice' Task
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.Download;

% Validate the Workflow Configuration Object
hWC.validate;
```

To learn more about the script-based workflow, see “Run HDL Workflow with a Script” on page 29-47.

Programming Methods

Download

If you use the reference designs for Zynq and Intel SoC hardware platforms, the code generator uses `Download` as the default **Programming method**.

When you use `Download` as the **Programming method** and run the **Program Target Device** task, HDL Coder copies the generated FPGA bitstream, Linux devicetree, and system initialization scripts to the SD card on the target board, and then keeps the bitstream on the SD card persistently. To use this programming method, you do not require an Embedded Coder license. You can create an SSH object by specifying the **IP Address**, **SSH Username**, and **SSH Password**. HDL Coder uses the SSH object to copy the bitstream to the SD card and reprogram the board.

It is recommended that you use the `Download` method, because this method programs the FPGA bitstream and loads the corresponding Linux devicetree before booting the Linux system. Therefore, the `Download` method is more robust and does not result in a kernel panic or kernel hang on boot up. During the Linux reboot, the FPGA bitstream is reprogrammed from the SD card automatically. To specify `Download` as the **Programming method** when you run the workflow using the Workflow Advisor script, before you run the script, make sure that the `ProgrammingMethod` property of the WorkflowConfig object, `hWC`, is set to `Download`.

```
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.Download;
```

JTAG

When you specify `JTAG` as the **Programming method** and run the **Program Target Device** task, HDL Coder uses a JTAG cable to program the target SoC device. Use this method to program Intel and Xilinx SoC devices and standalone FPGA boards.

For programming SoC devices, it is recommended that you use the `Download` method. The `JTAG` mode does not involve the ARM processor and programs the onboard FPGA directly. This mode does not update the Linux devicetree, and can crash the Linux system or result in a kernel panic when the bitstream does not match the devicetree. To avoid this situation, use the `Download` method to program the SoC device.

The standalone Intel and Xilinx FPGA boards do not have an embedded ARM processor. `JTAG` is the default method that you use to program the FPGA boards.

To specify `JTAG` as the **Programming method** when you run the workflow using the Workflow Advisor script, before you run the script, change the `ProgrammingMethod` property of the WorkflowConfig object, `hWC`, to `JTAG`.

```
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.JTAG;
```

Custom

To program the target device, you can specify a custom programming method when you create your own custom reference design. Use the `CallbackCustomProgrammingMethod` of the `hdlcoder.ReferenceDesign` class to register a function handle for the callback function that gets executed when running the **Program Target Device** task. To define your callback function, create a file that defines a MATLAB function and add the file to your MATLAB path.

This example code snippet shows a reference design definition file that uses a custom programming method. To learn more, see `CallbackCustomProgrammingMethod`.

```
unction hRD = plugin_rd()
% Reference design definition

% ...

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Parameter Callback Custom';
hRD.BoardName = 'ZedBoard';

% Tool information
hRD.SupportedToolVersion = {'2020.2'};
```

```
% ...

hRD.CallbackCustomProgrammingMethod =
            @my_reference_design.callback_CustomProgrammingMethod;
```

See Also

Related Examples

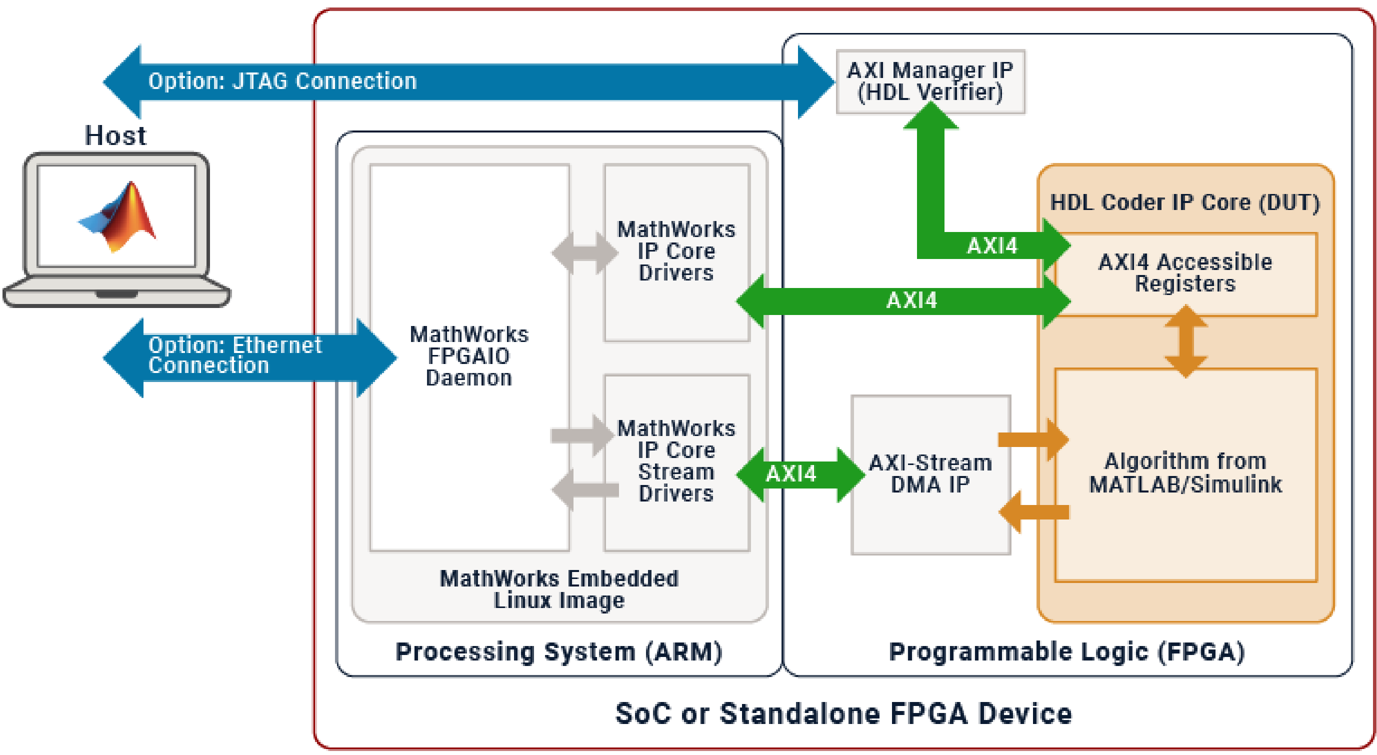
- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270

More About

- “Program Target Device” on page 36-22
- “IP Core Generation Workflow for Standalone FPGA Devices” on page 40-141
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9

Generate and Manage FPGA I/O Host Interface Scripts

You can use host interface scripts to prototype your algorithms on hardware from MATLAB. Host interface scripts contains simple MATLAB commands that you can use to connect to your hardware and interact with your algorithm while it runs on hardware. For example, you can tune parameters or capture data for further analysis.



Prerequisites

- To run the generated host interface script, you need a hardware board. For instructions on how to set up your hardware board, see “Guided Hardware Setup”.
- You must have the latest version of the third-party synthesis tool, such as Xilinx Vivado. See “HDL Language Support and Supported Third-Party Tools and Hardware”. In your MATLAB session, set the path to the installed synthesis tool by using the `hdlsetuptoolpath` function.

Generate Host Interface Scripts

You can generate a host interface script when you generate an IP core using HDL Coder. During this process, you must select a host-target interface, which specifies how MATLAB communicates to your hardware board. To specify the target interface for the host interface script, in the **HDL Code** tab, select **Host Interface Script > Host target interface** and select either `Ethernet` or `JTAG` from the drop-down menu options. The host-target interface options are:

- Ethernet — To use the Ethernet option, your target board must have an embedded ARM processor.
- JTAG AXI Manager — Use this option for standalone FPGA boards that do not have an embedded ARM processor.

To generate a host interface script by using the Simulink Toolstrip:

1. Open the Simulink model and select the design under test (DUT) for which you want to generate an IP core.
2. Open the **HDL Coder** tab. In the **Apps** tab, click **HDL Coder**.
3. To use a JTAG interface, in the **HDL Code** tab, click **Settings > Target Settings**. Set the **Insert AXI Manager (HDL Verifier required)** to `JTAG`.
4. To generate the host interface script, in the **HDL Code** tab, click **Host Interface Script > Host Interface Script**. You can generate the host interface script without generating the IP core. To configure the target interface that the host interface script uses to access the generated IP core, select the **Host Interface Script** drop-down arrow and set **Host target interface** to either `JTAG` or `Ethernet`. To use the JTAG interface, you must have a HDL Verifier license.

To generate a host interface script by using the HDL Workflow Advisor:

1. Open the Simulink model and select the design under test (DUT) for which you want to generate an IP core.
2. Open the **HDL Coder** App.
3. In the HDL Workflow Advisor, click **Task 1.1 Set Target device and Synthesis Tool** and configure the target device and synthesis tool.
4. In **1.2 Set Target Reference Design** set the **Reference design** property. To use the JTAG interface in **Task 1.2 Set Target Reference Design**, set **Insert AXI Manager (HDL Verifier required)** to `JTAG`.
5. In **1.3 Set Target Interface** map your DUT ports to various interfaces. HDL Coder uses these interface settings during host interface script generation.
6. Click **4.2. Generate Software Interface** and select **Generate host interface script**. Right-click **4.2. Generate Software Interface** and select **Run to Selected Task**.

Host Interface Script Files

When you generate a host interface script file, you create two MATLAB files based on the reference design and target interface table mapping that was configured for your IP core:

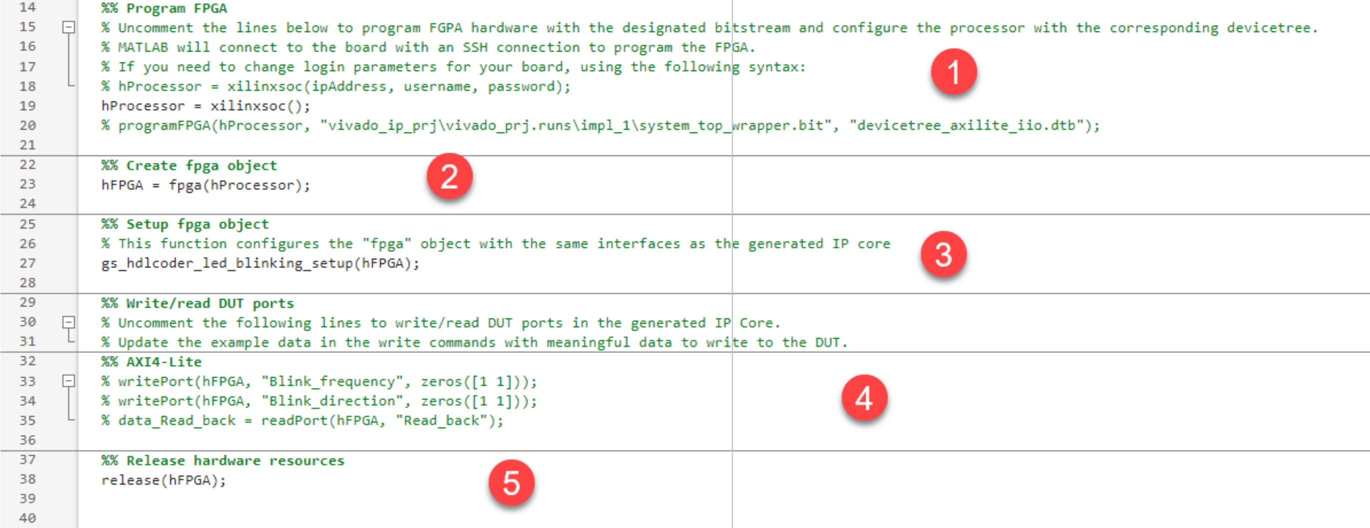
- Interface script (`gs_<modelName>_interface.m`: This script creates an `fpga` hardware object for interfacing with your FPGA from MATLAB. The interface script contains MATLAB commands that connect to your hardware and program the FPGA, and examples of how to exchange data with your algorithm as it runs on hardware.
- Interface setup function (`gs_<modelName>_setup.m`: This function configures the `fpga` object with the hardware interfaces and ports from your DUT algorithm. The setup function contains DUT port objects that have the port name, direction, data type, and interface mapping information. The function maps these DUT ports to the corresponding interfaces.

Interface Script File

The interface script file creates a connection to your FPGA hardware for reading and writing data. The script file:

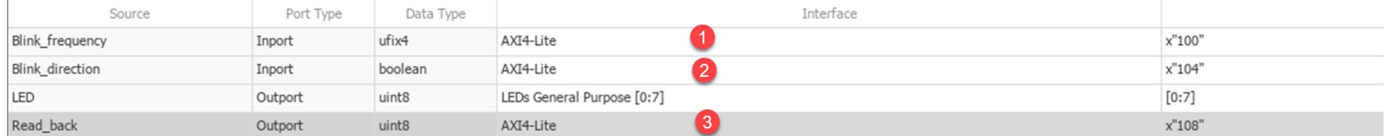
1. Creates an `intelsoc` or `xilinsoc` object, which represents a connection to the processor on your hardware board. The script file contains the `programFPGA` command that programs the FPGA with the generated bitstream and corresponding device tree. This section is present only when you generate an Ethernet host target interface.

2. Creates an `fpga` hardware object that represents a connection to the FPGA on your hardware board.
3. Configures the `fpga` object with the desired hardware interfaces and ports from your DUT algorithm.
4. Contains commands that read or write data to DUT ports, which you can use to exercise the algorithm running on the hardware. These commands serve only as an example. Update them before running the script to exercise your algorithm.
5. Releases any hardware resources used by the `fpga` object to clean up the connection.



Setup Function File

The setup function file configures your `fpga` object with the same interfaces as your generated IP core. The setup script is a reusable file. When you make changes to the IP core, you must update or re-generate the setup script. This image shows the ports and interface mapping of the DUT and the setup script and how they are related.



Source	Port Type	Data Type	Interface	
Blink_frequency	Inport	ufix4	AXI4-Lite 1	x"100"
Blink_direction	Inport	boolean	AXI4-Lite 2	x"104"
LED	Outport	uint8	LEDs General Purpose [0:7]	[0:7]
Read_back	Outport	uint8	AXI4-Lite 3	x"108"

```
11  %% AXI4-Lite
12  addAXI4SlaveInterface(hFPGA, ...
13      "InterfaceID", "AXI4-Lite", ...
14      "BaseAddress", 0x40010000, ...
15      "AddressRange", 0x10000);
16
17  DUTPort_Blink_frequency = hdlcoder.DUTPort("Blink_frequency", ...
18      "Direction", "IN", ...
19      "DataType", numerictype(0,4,0), ...
20      "IsComplex", false, ...
21      "Dimension", [1 1], ...
22      "IOInterface", "AXI4-Lite", ...
23      "IOInterfaceMapping", "0x100");
24
25  DUTPort_Blink_direction = hdlcoder.DUTPort("Blink_direction", ...
26      "Direction", "IN", ...
27      "DataType", "logical", ...
28      "IsComplex", false, ...
29      "Dimension", [1 1], ...
30      "IOInterface", "AXI4-Lite", ...
31      "IOInterfaceMapping", "0x104");
32
33  DUTPort_Read_back = hdlcoder.DUTPort("Read_back", ...
34      "Direction", "OUT", ...
35      "DataType", "uint8", ...
36      "IsComplex", false, ...
37      "Dimension", [1 1], ...
38      "IOInterface", "AXI4-Lite", ...
39      "IOInterfaceMapping", "0x108");
40
41  mapPort(hFPGA, [DUTPort_Blink_frequency, DUTPort_Blink_direction, DUTPort_Read_back]);
42
```

1

2

3

The setup function file changes based on whether the target interface is Ethernet or JTAG. When you set the target interface to JTAG, HDL Coder uses the `aximanager` hardware object as the read and write driver for communicating with AXI registers in your design. This image shows a comparison between the setup function for the Ethernet and JTAG interfaces.

```
%% AXI4-Lite
addAXI4SlaveInterface(hFPGA, ...
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0x400D0000, ...
    "AddressRange", 0x10000, ...
    "WriteDeviceName", "mwipcore0:mmwr0", ...
    "ReadDeviceName", "mwipcore0:mmrd0");
```

Ethernet

```
%% AXI4-Lite
hAXIMDriver = aximanager("Xilinx");
addAXI4SlaveInterface(hFPGA, ...
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0x400D0000, ...
    "AddressRange", 0x10000, ...
    "WriteDriver", hAXIMDriver, ...
    "ReadDriver", hAXIMDriver, ...
    "DriverAddressMode", "Full");
```

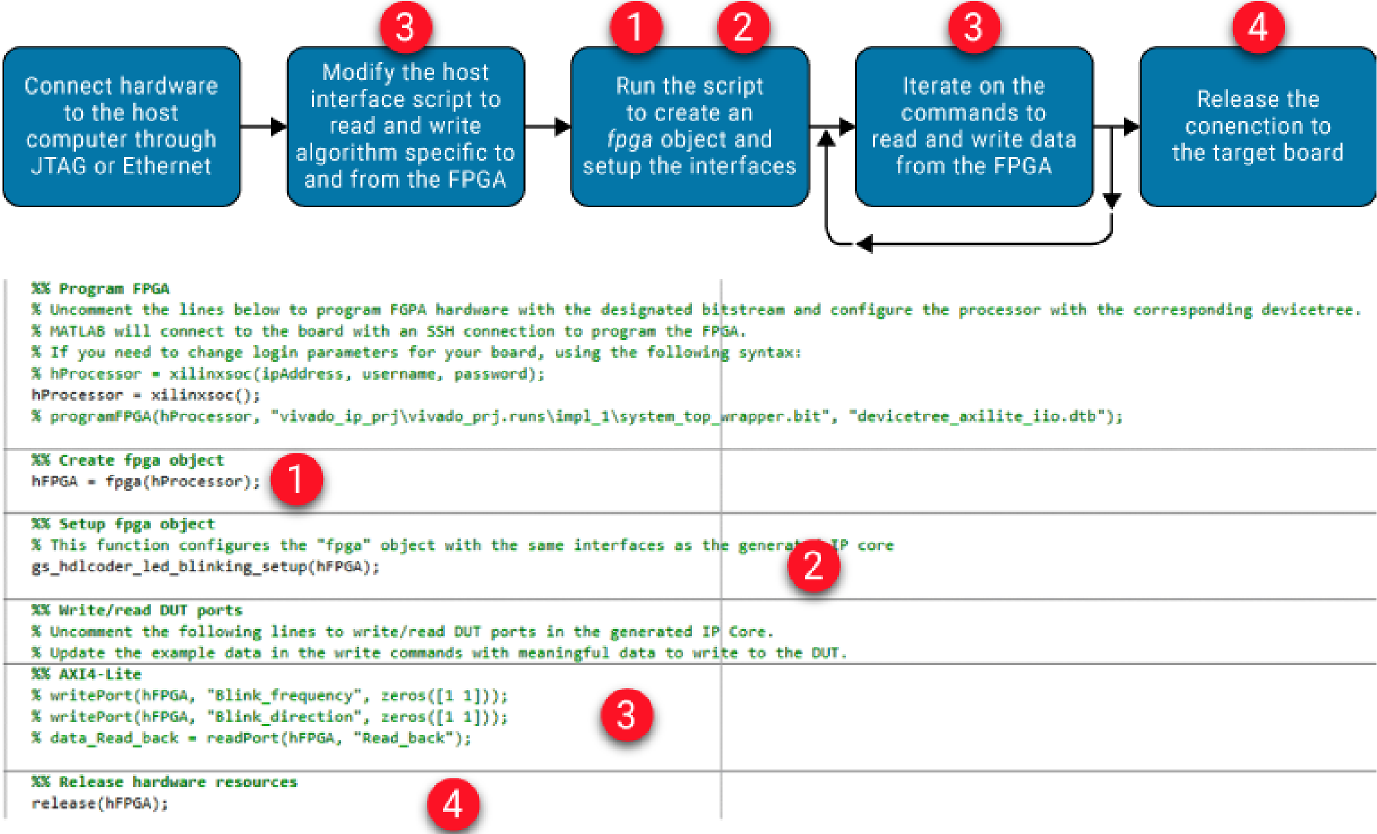
JTAG

Use the Host Interface Script with Hardware

For rapid prototyping, customize the host interface script based on how you modify your original design. After you generate the host interface scripts:

1 Connect your host computer to the target hardware board using either Ethernet or JTAG.
2 Modify the read and write commands in the interface script file to match your data requirements. Use the modified script interface with your deployed DUT IP core or algorithm running on the target board.
3 Run the sections of the host interface script to connect to your hardware board, program the FPGA, create an `fpga` object, and set up the interfaces. The interfaces are set up based on the interface setup function file. You need to run these sections only once.
4 Run the modified read and write commands and keep iterating on these commands. You can use the MATLAB command line to run specific commands.
5 Release the connection to the hardware board after completing your prototyping.

This image shows how to use the host interface script and the relation between the steps and the interface script file.



After you have iterated and tested the host interface script, you can :

- Integrate the script into a testing or verification workflow.
- Create a live script and interactive prototype your design. For example, see “Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands” on page 39-241.

Manage Host Interface Scripts

You can manage your host interface script files by either updating or re-generating the files. Update host interface script files when you make minor changes such as modifying existing parameters, making minor error fixes, and so on. Re-generate host interface script files when making major changes such as modifying the DUT port mappings, changing the target hardware device vendor, changing the target software tool, and so on. When you re-generate the host interface script files, HDL Coder displays a warning about overwriting the existing files. You can rename your existing files to prevent them from being overwritten.

See Also

Objects
`hdlcoder.WorkflowConfig` | `fpga` | `intelsoc` | `xilinxsoc`

Functions
`hdlcoder.runWorkflow`

More About

- "Model Design for AXI4-Stream Interface Generation" on page 40-14
- "Model Design for AXI4 Slave Interface Generation" on page 40-3
- "Use FPGA I/O to Rapidly Prototype HDL IP Core" on page 39-90

Choose a Method to Interact with IP Cores on Target Hardware

You can interact with your generated IP core on your targeted hardware by using MATLAB host interface scripts, Simulink host interface models, Simulink software interface models, FPGA data capture, or generic software interfaces. Depending on the method you choose, you can interact with your IP core at different stages of the hardware-software co-design process, such as when you run and verify the IP core on hardware, configure software interfaces to the FPGA, or deploy the complete design to hardware and software. To determine the most appropriate way to interact with your IP core, evaluate the design stage, required tools, and the associated strengths and limitations of each method.

Overview of Interface Methods

Key aspects of the available ways to interact with your IP core are highlighted in the following table. For more information about the stages of the hardware-software co-design process, see “Targeting FPGA & SoC Hardware Overview” on page 39-3.

Interface Method	When to Use Method	Supported Host-Target Connections	Required Tools	Strengths	Limitations
“FPGA I/O” on page 39-75	• Run and Verify IP Core	• Ethernet • JTAG* *HDL Verifier is required to insert the AXI Manager IP.	• HDL Coder	Leverage MATLAB scripting capability to rapidly prototype and eliminates the need to generate embedded software and drivers.	Requires a MATLAB connection. You can only use this method to create prototypes. You cannot generate code.
“Simulink Host Interface Model” on page 39-77	• Run and Verify IP Core	• JTAG* *HDL Verifier is required to insert the AXI Manager IP.	• Simulink • HDL Coder • HDL Verifier	Leverage the built-in data logging and visualization capabilities in Simulink to prototype the design.	You can only use this method to prototype the design. You cannot generate code.
“FPGA Data Capture” on page 39-78	• Run and Verify IP Core	• Ethernet • JTAG	• Simulink • HDL Coder • HDL Verifier	Observe the raw signal data in the IP core on an FPGA without using interface protocol modeling.	You must regenerate the IP core to accommodate FPGA data capture interface mappings, such as testpoints.

Interface Method	When to Use Method	Supported Host-Target Connections	Required Tools	Strengths	Limitations
“Simulink Software Interface Model in External Mode” on page 39-79	• Run and Verify IP Core • Configure Software Interface to FPGA	• Ethernet	• Simulink • HDL Coder • Embedded Coder • Simulink Coder	Use Simulink environment to monitor and tune IP core. Additionally, develop and deploy software model directly to the processor.	Requires a connection to Simulink. This method is unsuitable for production-level designs.
“Simulink Software Interface Model for Standalone Applications” on page 39-80	• Configure Software Interface to FPGA • Hardware-Software Deployment	• No host required. • UDP connections optional.	• Simulink • HDL Coder • Embedded Coder • Simulink Coder	Build and deploy an executable to a processor that does not require a connection to MATLAB or Simulink.	You must manually implement external communication channels.
“Generic Software Interface” on page 39-81	• Hardware-Software Deployment	• No host required.	• None	You have complete control over software.	Requires proficiency in embedded software design and SoC architecture.

Prerequisites

Before choosing an interface method, configure your hardware board and host environment to interact with your design on hardware:

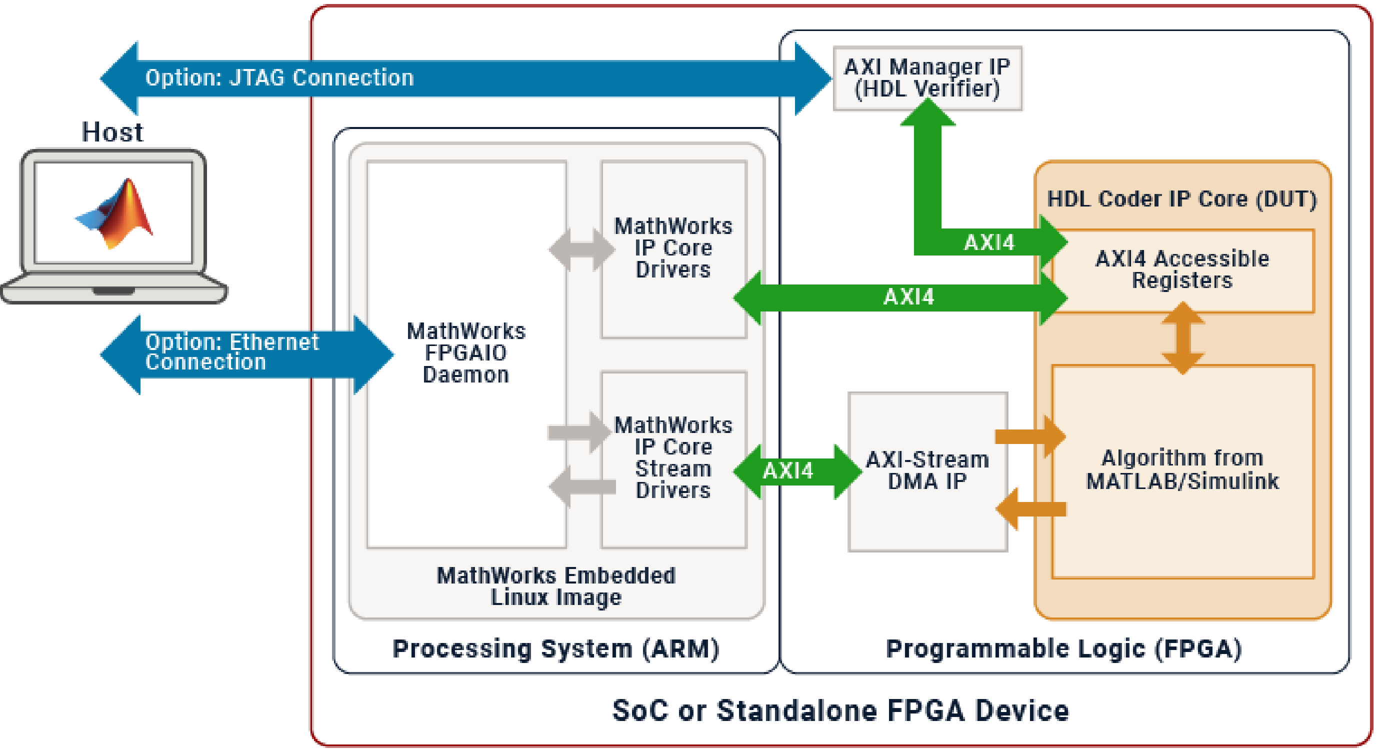
- Set up your hardware board. For more information, see Guided SD Card Setup.
- Install a synthesis tool that is compatible with your hardware. For a list of supported versions, see HDL Language Support and Supported Third-Party Tools and Hardware.
- Compile design to run on hardware. One approach is to complete through "Embedded System Tool Integration" in “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9.

FPGA I/O

You can use MATLAB FPGA I/O host interface scripts when you generate IP cores in MATLAB or Simulink. If you use Simulink, you can automatically generate these scripts to match your desired connection method to the host, such as Ethernet or JTAG. If you use MATLAB, you must manually construct the FPGA I/O host interface scripts. Use the scripts to enable MATLAB to communicate with the FPGA using a MathWorks Linux SoC environment installed on the board, if the board is connected to the host via Ethernet, or directly with the FPGA if the board is connected to the host via JTAG using the AXI Manager IP.

When connecting to the host via JTAG, you can bypass the processor and directly perform read and write operations on the inputs and outputs of the IP core, which is useful for targeting standalone FPGA boards. Host interface scripts eliminate the need to deploy embedded software or drivers on the processor to run and verify your IP core.

You can use MATLAB host interface scripts in the "Run and Verify IP Core on Target Hardware" on page 39-5 stage of the hardware-software co-design process.



For an example about generating and using host interface scripts in Simulink, see Prototype FPGA Design on Hardware with Live Data by Using MATLAB Commands on page 39-241.

For an example that uses host interface scripts in MATLAB, see Generate IP Core from MATLAB for Blinking LEDs on FPGA Board on page 39-180.

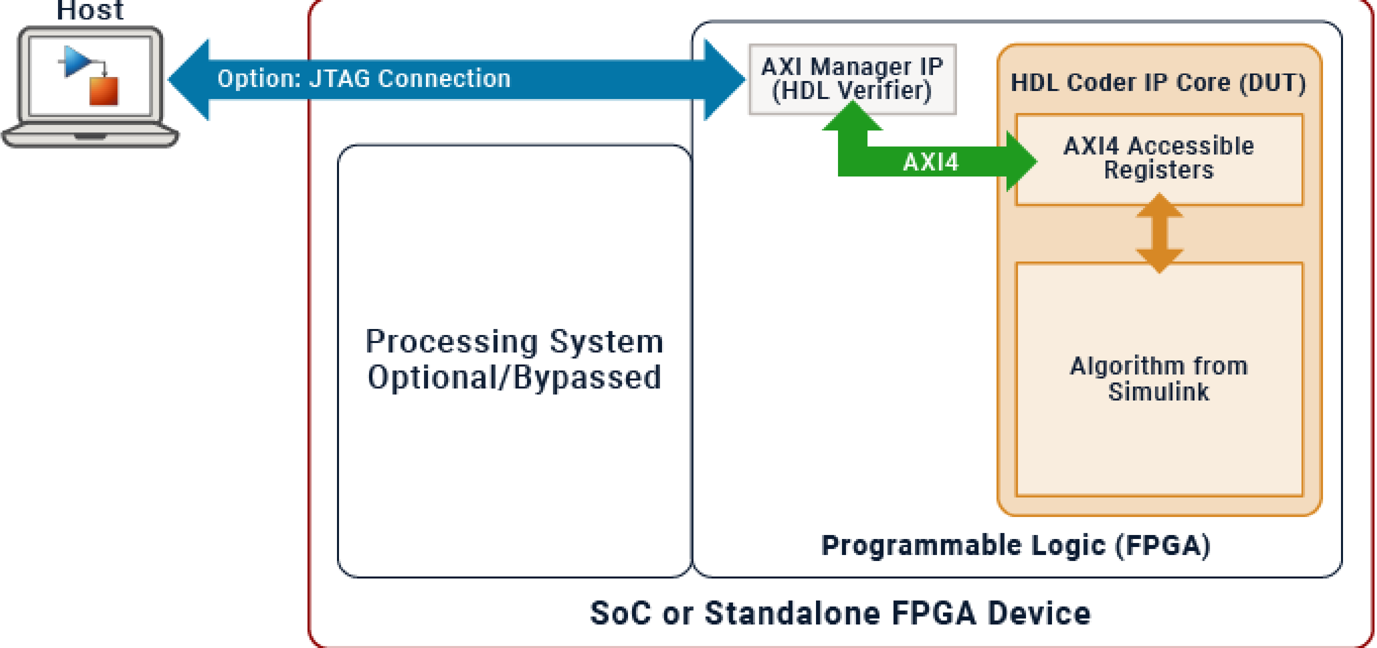
To learn how to generate and manage host interface scripts, see "Generate and Manage FPGA I/O Host Interface Scripts" on page 39-68.

Strengths	Limitations
• Provides a simple solution for rapid prototyping. • Leverages the scripting capabilities of MATLAB. • Provides greater flexibility and control over low-level operations, which makes it simpler to interface with hardware. • Eliminates the need for embedded software and driver generation and deployment. • Can be used in Ethernet or JTAG connections to the host. • Supports AXI-Stream for fast data transfer.	• Requires a connection to MATLAB, which makes this method unsuitable for production-level designs. • The latency and throughput may not be suitable for real-time operations that require high-bandwidth data capture or fast control loops.

Simulink Host Interface Model

If you are using Simulink to develop your IP core, you can prototype your design on an FPGA platform by using the Simulink host interface model. This method takes advantage of the JTAG AXI Manager IP, which is beneficial when targeting standalone FPGA boards or SoCs that lack a compatible MathWorks Linux image. This method leverages the Simulink environment to interact with the FPGA. By using a Simulink host interface model, you can bypass the processor and directly perform read and write operations on the inputs and outputs of the IP core mapped to the AXI4 registers.

You can use Simulink host interface models in the "Run and Verify IP Core on Target Hardware" on page 39-5 stage of the hardware-software co-design process.



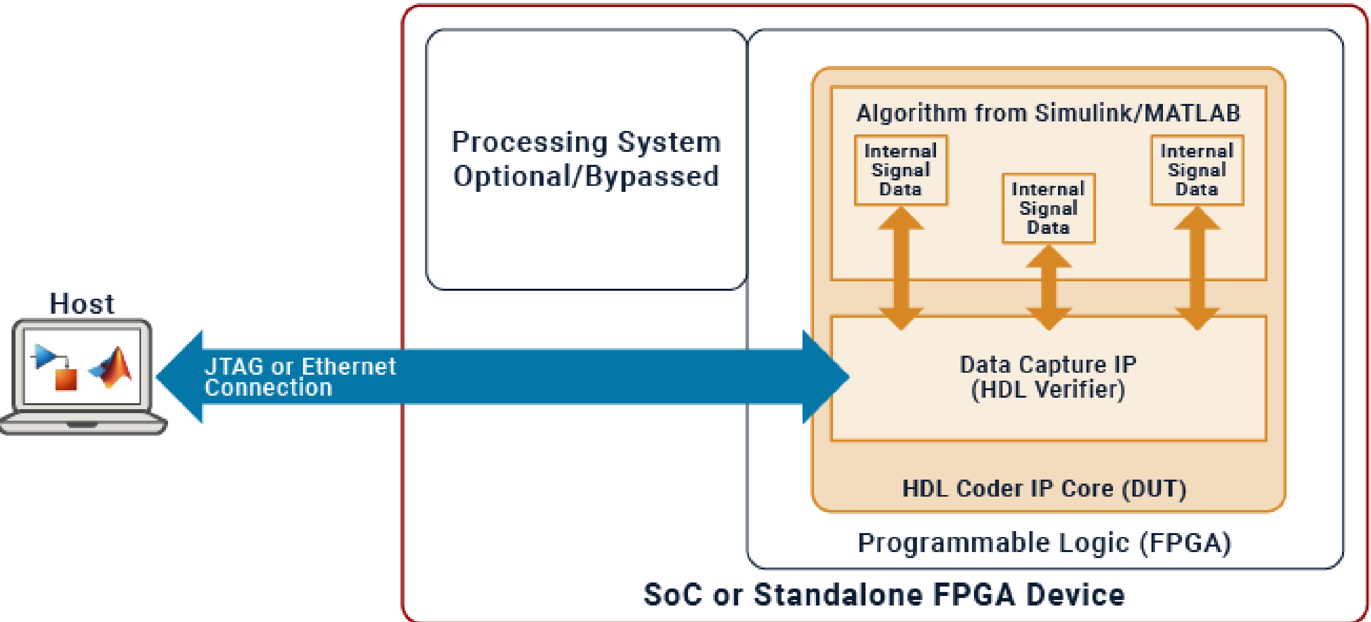
To learn more about creating and using a Simulink host interface model, see Use JTAG AXI Manager to Control HDL Coder Generated IP Core on page 40-333.

Strengths	Limitations
• Leverages the built-in data logging and visualization capabilities in Simulink to run and verify the IP core. • Useful when using Simulink and targeting standalone FPGAs and SoCs that do not have a compatible MathWorks Linux image.	• Sequential operations can be challenging to implement in a Simulink environment. • You can only interact with the IP core on the FPGA and cannot run and verify operations on the processor due to the JTAG connection. • Does not support the AXI-Stream interface. • The AXI Manager IP consumes FPGA resources, which you must take into account when designing the overall FPGA system.

FPGA Data Capture

To interface with internal signals within your IP core, you can utilize FPGA data capture. When you develop a model using Simulink, you can automatically integrate this method with your IP core. This method usually involves establishing a JTAG connection to the host. You can employ test points to monitor the internal signals of the IP core while your hardware-software design runs on the hardware. FPGA Data Capture is useful when debugging and analyzing the design of the IP core because it eliminates the need for interface protocol modeling.

You can use FPGA data capture in the “Run and Verify IP Core on Target Hardware” on page 39-5 stage of the hardware-software co-design process.



To learn more about using FPGA Data Capture, see Debug IP Core Using FPGA Data Capture on page 40-351.

Strengths	Limitations
• Allows for real-time debugging, which enables you to quickly identify design issues. • Enables you to debug your IP core while it is running on hardware by using conditional logic triggers to perform event-based capture. • Enables you to observe raw signal data at the FPGA level without the need for interface protocol modeling.	• You must integrate the FPGA data capture IP into the design. The data capture IP can be automatically integrated into the design if you use Simulink. If you are not using Simulink, you must manually integrate the data capture IP. • Consumes FPGA resources, which can potentially limit the available resources for other aspects of the FPGA design. • Requires that you understand the internal workings of the FPGA design, including the signal functionality and expected behavior.

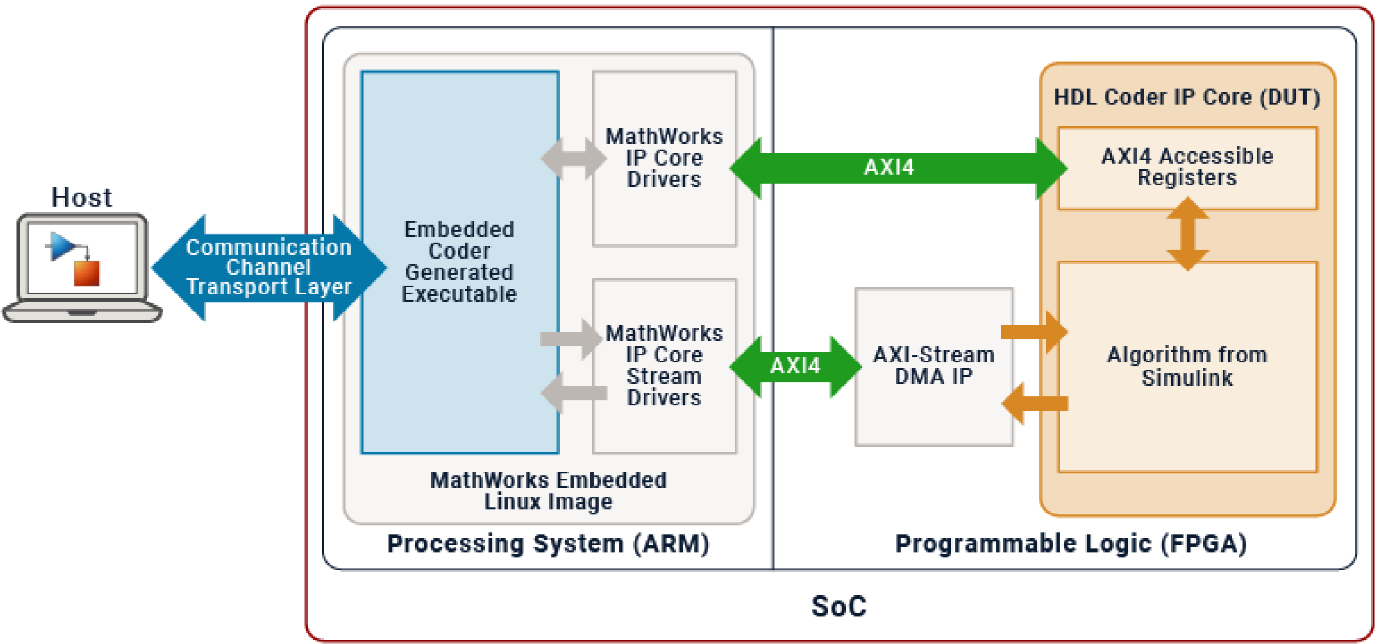
Simulink Software Interface Model in External Mode

If you are using Simulink to develop your IP core, you can monitor and tune your design by creating a Simulink software interface model and running it in external mode. The Simulink software interface model includes the software portion of the design, which consists of all blocks outside of the HDL DUT subsystem. Software interface models replace the DUT algorithm from your original model with AXI driver blocks based on the target platform interface table and reference design settings. The software interface model interacts with the deployed IP core on the FPGA by using the AXI driver blocks.

Additionally, you can use Embedded Coder to generate C code from your software interface model, build it, and deploy it to the processor. Running your software interface model in external mode establishes automatic communication channels between the host PC and the hardware, which enables parameter tuning and data visualization on the host PC. This functionality allows you to not only run and verify your IP core but also configure the software interface to the FPGA.

You can use Simulink software interface models in these stages of the hardware-software co-design process:

- “Run and Verify IP Core on Target Hardware” on page 39-5
- “Configure Software Interface to FPGA” on page 39-6



To learn more about monitoring and tuning your design using a Simulink software interface model, see "Debug a Zynq Design Using HDL Coder and Embedded Coder" on page 40-346.

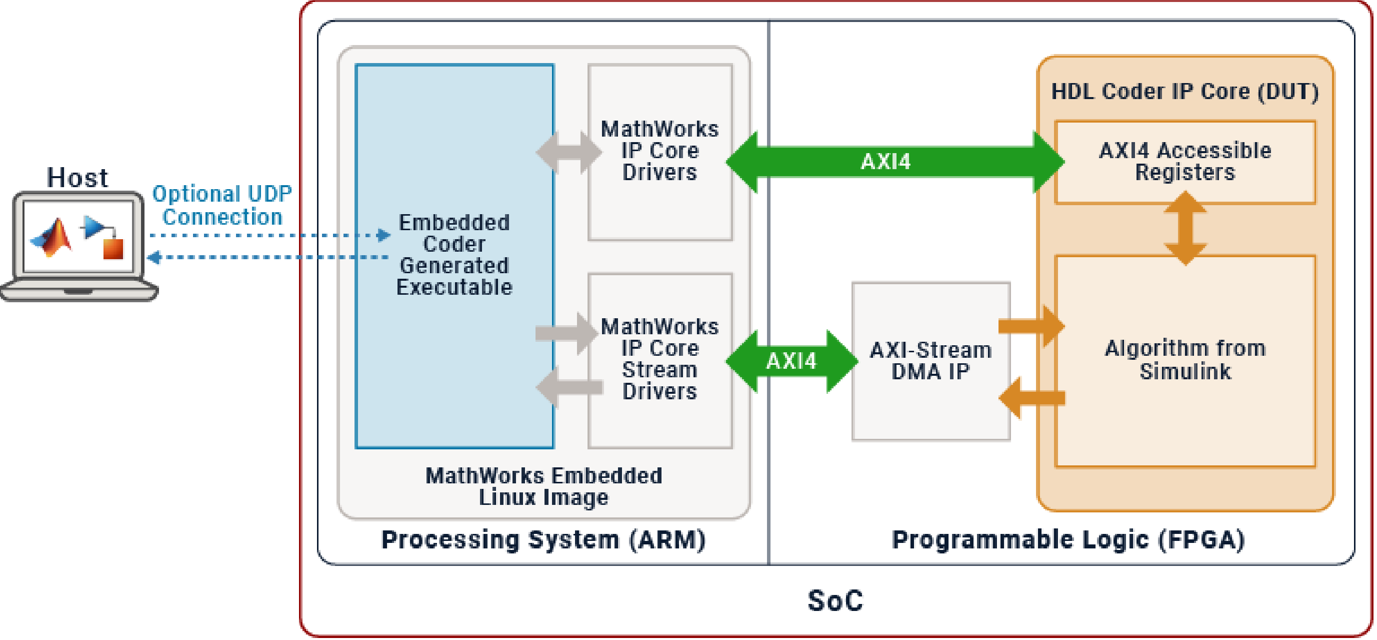
Strengths	Limitations
• Allows you to easily develop the software side of the design using Simulink. • Enables to you to deploy the software side of the design to processor by using the Simulink interfaces for running and verifying the design. • Supports AXI-Stream for fast data transfer.	• Implementing a software interface in Simulink can be more time consuming and less intuitive compared to MATLAB. • Interacting with hardware in Simulink can lead to complex block diagrams to represent the entire system. • Relying solely on Simulink may not be suitable for production-level designs.

Simulink Software Interface Model for Standalone Applications

If you are using Simulink to develop your IP core, you can create a standalone application intended for production-level designs. This approach involves integrating both the HDL DUT subsystem and the software portion of your design in the model. If needed, you can also use UDP Send and UDP Receive blocks to enable external communication. You can use Embedded Coder to compile and deploy a standalone executable of the model onto the processor that is completely independent of any connection to Simulink.

You can use Simulink software interface models for Simulink standalone applications in these stages of the hardware-software co-design process:

• "Configure Software Interface to FPGA" on page 39-6
• "Hardware-Software Deployment" on page 39-6



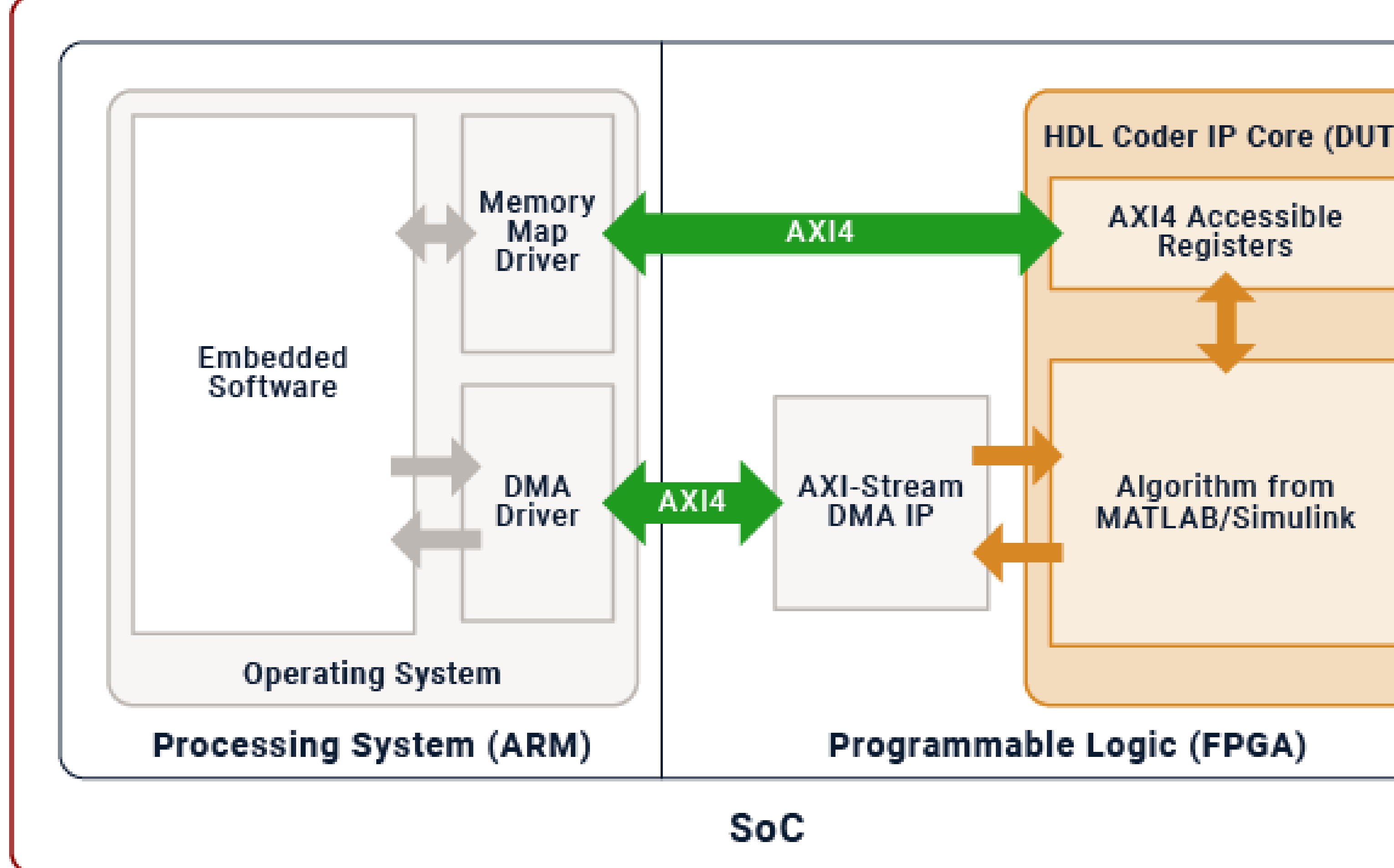
For more information about generating a Simulink standalone application, see Build and Run Executable on Xilinx Zynq Platform (Embedded Coder).

Strengths	Limitations
• Enables you to build and deploy standalone executables that are independent of MATLAB or Simulink and that are suitable for production-level designs. • Supports AXI-Stream for fast data transfer.	• Manually implementing external communication channels can result in increased development time and additional overhead for debugging.

Generic Software Interface

If you have a production-level design, you can use a generic software interface to manually implement embedded software and drivers on the processor using third-party tools. For example, you can write C code to run in bare metal, a real-time operating system (RTOS) or an embedded Linux environment. You can use these software implementations to interact with and control your IP core on the FPGA. Additionally, generic software interfaces are compatible with IP cores generated and deployed with MATLAB or Simulink.

You can use generic software interfaces in the “Hardware-Software Deployment” on page 39-6 stage of the hardware-software co-design process.



To learn more about files generated by HDL Coder during the IP core process, see "Custom IP Core Generated Files" on page 39-18.

Strengths	Limitations
• Complete control over software application. • Supports AXI-Stream for fast data transfer.	• Requires expertise in embedded software design, driver creation, and SoC architectures.

See Also

More About

- "Generate and Manage FPGA I/O Host Interface Scripts" on page 39-68
- "Generate Software Interface Model to Probe and Rapidly Prototype HDL IP Core" on page 39-84
- "Communicate with the Programmable Logic IP Core on Xilinx Zynq Platform Using AXI4-Lite Protocol" (Embedded Coder)
- "Communicate with Xilinx Zynq Platform Using UDP Protocol" (Embedded Coder)

- “Model Design for AXI4-Stream Interface Generation” on page 40-14

Generate Software Interface Model to Probe and Rapidly Prototype HDL IP Core

When you run the hardware-software co-design workflow for SoC platforms, you generate an HDL IP core for the DUT algorithm, and then integrate the IP core into the reference design. See “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9.

To test the HDL IP core on the target hardware, generate a software interface model. The software interface model uses AXI driver blocks to test the HDL IP core functionality in external mode simulation.

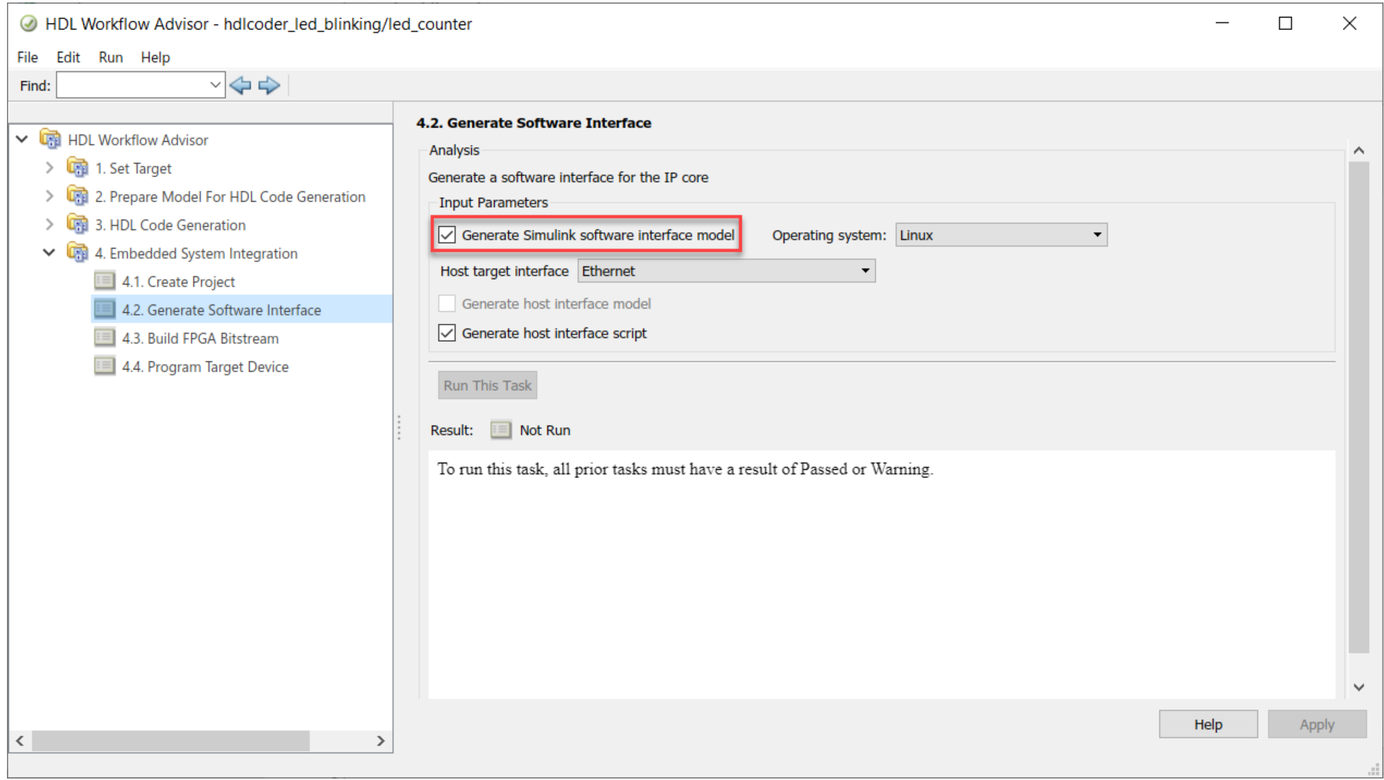
Prerequisites

- A target platform, such as ZedBoard, where you want to deploy your software interface model.
- The latest version of the third-party synthesis tool, such as Xilinx Vivado. See “HDL Language Support and Supported Third-Party Tools and Hardware”. In your MATLAB session, set the path to that installed synthesis tool by using the `hdlsetuptoolpath` function.
- If you are generating the software interface model, install Embedded Coder and Simulink Coder.
- Install the HDL Coder and Embedded Coder support packages for the target platform. On the MATLAB Toolstrip, click **Home** > **Add-Ons** > **Get Add-Ons** button. See “Get and Manage Add-Ons”.

Generate Software Interface

When running the IP core generation workflow, you can generate a host interface script and software interface model from the HDL Workflow Advisor UI or at the command line.

From the UI, in the **Embedded System Integration** > **Generate Software Interface** task, select the **Generate Simulink software interface model** check box.



If you are targeting standalone FPGA boards, you cannot generate a software interface model. Instead, you can generate a host interface script and test the IP core by using the AXI Manager driver.

1. In the **Set Target Reference Design** task, set **Insert AXI Manager (HDL Verifier required)** to `JTAG` or `Ethernet` based on the interface that communicates between your host machine and the FPGA board. For Ethernet, specify the IP address of the FPGA board using the **Board IP Address** parameter.

   Note By default, the `Ethernet` option is available for only the Artix-7 35T Arty, Kintex-7 KC705, and Virtex-7 VC707 boards. To enable this option for other Xilinx boards that have the Ethernet physical layer (PHY), manually add the Ethernet media access controller (MAC) Hub IP in the `plugin_board` file using the `addEthernetMACInterface` method before you launch the HDL Workflow Advisor tool.

   Run the workflow to the **Generate Software Interface** task.
2. In the **Generate Software Interface** task, select the **Generate host interface script** check box and run this task.

At the command line, export the HDL Workflow Advisor settings to a script, and then use these properties with the Workflow Configuration object. This script specifies running the software interface task by generating the model and script. If you skip the task by setting RunTaskGenerateSoftwareInterface to false, then the model and script are not generated. See “Configure and Run IP Core Generation Workflow with a Script”.

```
% Export Workflow Configuration Script

% ...

%% Load the Model
load_system('hdlcoder_led_blinking');

%% Model HDL Parameters
% Set Model HDL parameters

% ...

hdlset_param('hdlcoder_led_blinking', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_led_blinking', 'Workflow', 'IP Core Generation');

% ...

% Set Workflow tasks to run
hWC.RunTaskGenerateSoftwareInterface = true;
hWC.GenerateSoftwareInterfaceModel = true;
hWC.GenerateHostInterfaceScript = true;

% ...

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC);
```

Software Interface Model

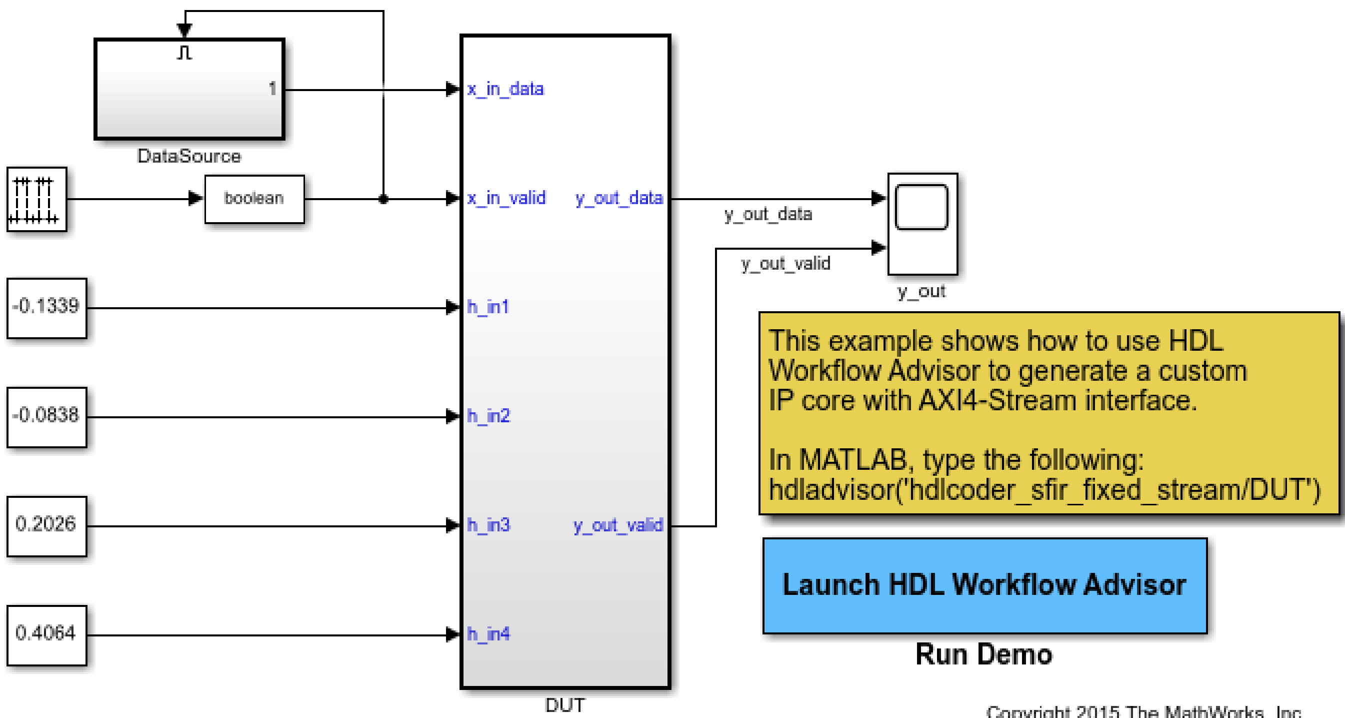
When you run the workflow for SoC platforms, a software interface model is generated to test the HDL IP core functionality. If you have Embedded Coder and Simulink Coder installed, you can generate embedded code from the model and build and run the executable on the ARM processor. When you target standalone FPGA boards, you cannot generate a software interface model because the boards do not have an embedded ARM processor. Instead, generate a host interface script to test the IP core by using the AXI Manager.

The generated software interface model replaces the DUT algorithm in your original model with AXI driver blocks based on the target platform interface table and reference design settings. To test the HDL IP core functionality, simulate the model in external mode to run on the target hardware. The model has concurrent execution enabled by default, which means that multiple tasks are executed concurrently by the processor on board the SoC platform.

The software interface model has the same name as your original model with the prefix `gm_` and the postfix `_interface`. The generated model from HDL code generation has the prefix `gm_`. To indicate that you are using this model as the software interface model, change the prefix to `sm_`.

Open the model `hdlcoder_sfir_fixed_stream`. This model maps the `sfir_fixed` model to the simplified AXI4-Stream protocol by inserting the `Valid` signal as an input control port.

```
openExample('hdlcoder_sfir_fixed_stream')
```



1. Open the HDL Workflow Advisor for the DUT subsystem. In the **Set Target Device and Synthesis Tool** task, specify `IP Core Generation` as **Target workflow** and ZedBoard as **Target platform**. Click **Run this task**.
2. In the **Set Target Reference Design** task, specify Default system with AXI4-Stream interface as the **Target Reference Design**. Click **Run this task**.
3. In the **Set Target Interface** task, map the DUT ports to target interfaces in the Target platform interface table. Click **Run this task**.

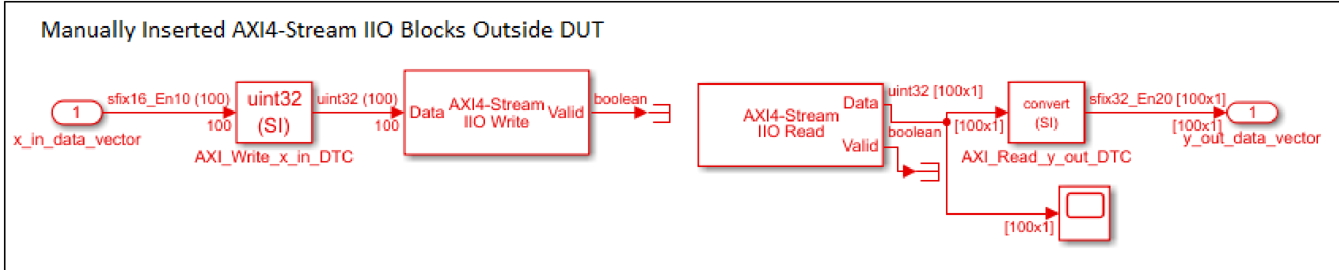
Target platform interface table

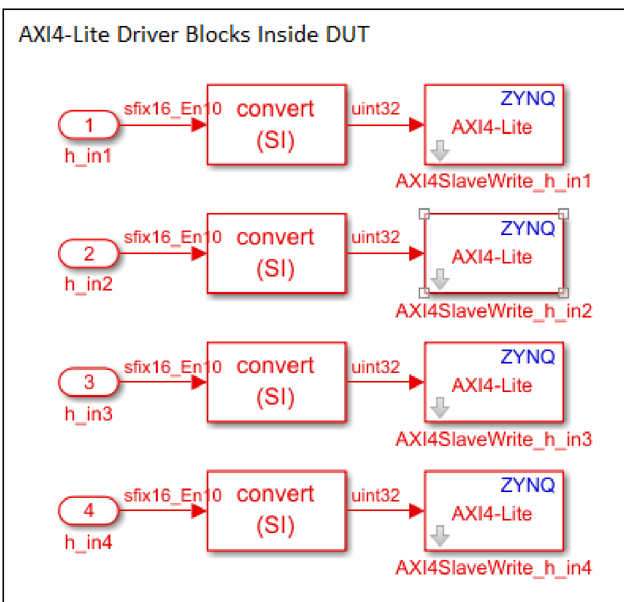
Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
x_in_data	Inport	sfix16_En...	AXI4-Stream Slave	Data	
x_in_valid	Inport	boolean	AXI4-Stream Slave	Valid	
h_in1	Inport	sfix16_En...	AXI4-Lite	x"100"	Options...
h_in2	Inport	sfix16_En...	AXI4-Lite	x"104"	Options...
h_in3	Inport	sfix16_En...	AXI4-Lite	x"108"	Options...
h_in4	Inport	sfix16_En...	AXI4-Lite	x"10C"	Options...
y_out_data	Outport	sfix32_En...	AXI4-Stream Master	Data	
y_out_valid	Outport	boolean	AXI4-Stream Master	Valid	

4 Right-click the **Generate Software Interface** task and select **Run to selected task**. Run the workflow to generate the software interface model.

The generated software interface model has the name `gm_hdlcoder_sfir_fixed_stream_interface.slx`. Because your original model uses scalar ports for the data ports `x_in_data` and `y_out_data`, the **Generate software interface** task displays a warning that AXI4-Stream IIO driver blocks are not automatically generated in the software interface model. You can either insert the driver blocks from the Embedded Coder Support Package Library for the target platform in the Simulink Library Browser or rerun the workflow by mapping the data ports to vector signals. To see a software interface model that has the AXI4-Stream IIO driver blocks inserted in it, open `hdlcoder_sfir_fixed_stream_sw`. The model uses AXI4-Stream IIO Write and AXI4-Stream IIO Read blocks for the data ports. The filter ports are mapped to AXI4-Lite driver blocks.

```
openExample('hdlcoder_sfir_fixed_stream_sw')
```





Configure the model with a stop time of `inf`. On the **Hardware** tab, the hardware settings specified on the model. You can then connect, and build and run the application on the target platform to verify the HDL IP core functionality. See “Deploy Model with AXI-Stream Interface in Zynq Workflow” on page 40-192.

See Also

Objects
`hdlcoder.WorkflowConfig` | `fpga` | `xilinxsoc` | `intelsoc` | `programFPGA`

Functions
`hdlcoder.runWorkflow`

More About

- “Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands” on page 39-241
- “Model Design for AXI4-Stream Interface Generation” on page 40-14
- “Model Design for AXI4 Slave Interface Generation” on page 40-3
- “Use FPGA I/O to Rapidly Prototype HDL IP Core” on page 39-90

Use FPGA I/O to Rapidly Prototype HDL IP Core

Rapidly prototype the HDL IP core by interfacing with your target board over Ethernet or JTAG. Use an Ethernet connection for boards that have an ARM processor. Use a JTAG connection for boards that do not have an ARM processor.

Prerequisites

- A target platform, such as ZedBoard™, where you deploy your software interface model
- The latest version of the third-party synthesis tool, such as Xilinx Vivado. See "HDL Language Support and Supported Third-Party Tools and Hardware". In your MATLAB session, set the path to that installed synthesis tool by using the `hdlsetuptoolpath` function.

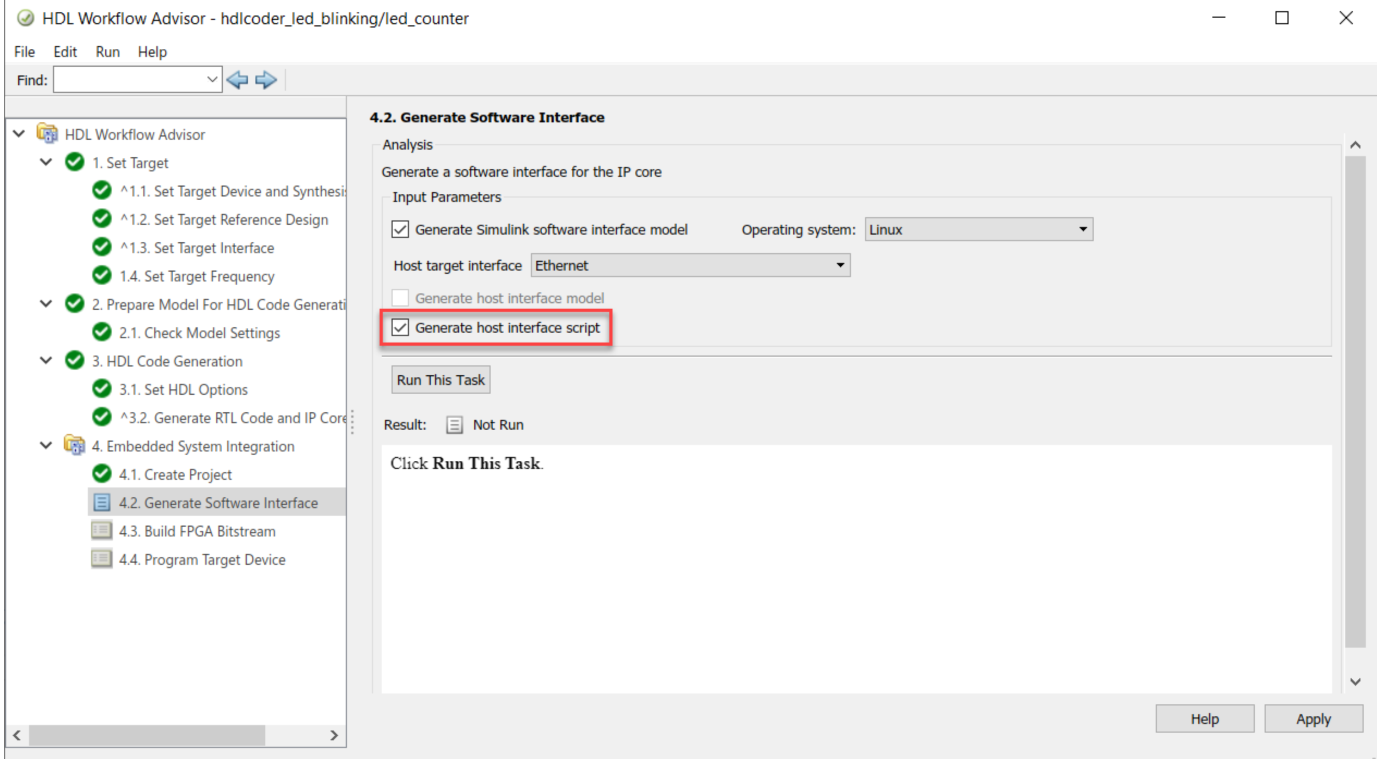
Ethernet-Based Interface

To use an Ethernet-based connection to your target hardware boards that have an embedded ARM processor, you can either generate a host interface script or create a custom software script. Before setting up the Ethernet-based interface, set up the board's SD card with the MathWorks firmware image. To set up the firmware image for your target board:

- For Intel related boards, see "Guided Hardware Setup"
- For Xilinx related boards, see "Guided Hardware Setup".

Host Interface Script

Set the **Host target interface** parameter to `Ethernet` and select the **Generate host interface script** check box. Then, run the **Generate Software Interface** task.



The generated MATLAB files are:

- `gs_modelName_setup.m`, which is a setup script that adds the AXI4 slave and AXI4-Stream interfaces. The script also contains DUT port objects that have the port name, direction, data type, and interface mapping information. The script then maps the DUT ports to the corresponding interfaces.
- `gs_modelName_interface.m`, which creates a target object, instantiates the setup script `gs_modelName_setup.m`, and then connects to the target hardware. The script then sends read and write commands to the generated HDL IP core.

See “Generate and Manage FPGA I/O Host Interface Scripts” on page 39-68.

Customize the Host Interface Script

For rapid prototyping, customize the host interface script or create your own script based on how you modify your original design. Customize the script to specify:

- A target object for a different FPGA vendor.
- Additional interfaces or configure existing interfaces based on modifications to your original design. HDL Coder uses this information to create the IIO drivers to access the HDL IP core.
- Additional DUT port objects or remove existing objects based on how you modify your design, and then change the mapping information accordingly.
- Input data to write to the DUT ports and output data to read from the ports.

Develop Host Interface Script

You can customize the generated host interface script or create your own host interface script. To create a custom host interface script:

1. Create an `fpga` object for the target device and store in `hFPGA`.

   ```
   hFPGA = fpga("Xilinx")
   ```

   ```
   hFPGA = 

      fpga with properties:

            Vendor: "Xilinx"
        Interfaces: [0×0 fpgaio.interface.InterfaceBase]
   ```

   To use an Intel target:

   ```
   hFPGA = fpga("Intel")
   ```

   ```
   hFPGA = 

      fpga with properties:

            Vendor: "Intel"
        Interfaces: [0×0 fpgaio.interface.InterfaceBase]
   ```

2. Configure the AXI interfaces to map the DUT ports in the generated HDL IP core. You can add AXI4 slave and AXI4-Stream interfaces. To add AXI4 slave interfaces, use the `addAXI4SlaveInterface` function.

```
addAXI4SlaveInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0xA0000000, ...
    "AddressRange", 0x10000, ...
    ... % Driver properties
    "WriteDeviceName", "mwipcore0:mmwr0", ...
    "ReadDeviceName", "mwipcore0:mmrd0");
```

To add AXI4-Stream interfaces, use the `addAXI4StreamInterface` function.

```
addAXI4StreamInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID", "AXI4-Stream", ...
    "WriteEnable", true, ...
    "ReadEnable", true, ...
    "WriteFrameLength", 1024, ...
    "ReadFrameLength", 1024, ...
    ... % Driver properties
    "WriteDeviceName", "mwipcore0:mm2s0", ...
    "ReadDeviceName", "mwipcore0:s2mm0");
```

The interface mapping information that you specified is saved as a property on the `fpga` object, `hFPGA`.

```
hFPGA

hFPGA = 

   fpga with properties:

         Vendor: "Xilinx"
     Interfaces: [1×2 fpgaio.interface.InterfaceBase]
```

For standalone FPGA boards that do not have an embedded ARM processor, you can create an object, and then use the `aximanager` object. Then use this object as the driver for the `addAXI4SlaveInterface` function. The `aximanager` object requires the HDL Verifier support package for the Intel or Xilinx FPGA boards.

```
% Create an "aximanager" object
hAXIMDriver = aximanager("Xilinx");

% Pass it into the addInterface command
addAXI4SlaveInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID",  "AXI4-Lite", ...
    "BaseAddress",  0xB0000000, ...
    "AddressRange", 0x10000, ...
    ... % Driver properties
    "WriteDriver", hAXIMDriver, ...
    "ReadDriver", hAXIMDriver, ...
    "DriverAddressMode", "Full");
```

3 Specify information about the DUT ports in the generated HDL IP core as a port object array by using the `hdlcoder.DUTPort` object. The object represents the ports of your DUT on the target hardware.

```
hPort_h_in1 = hdlcoder.DUTPort("h_in1", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x100")
```

```
hPort_h_in1 = 

   DUTPort with properties:

                   Name: "h_in1"
              Direction: IN
               DataType: [1×1 embedded.numerictype]
              Dimension: [1 1]
            IOInterface: "AXI4-Lite"
     IOInterfaceMapping: "0x100"
```

To write to or read from the DUT ports in the generated HDL IP core, map the ports to the AXI interface by using the `mapPort` function. After you map the ports to the interfaces, this information is saved on the `fpga` object as the `Interfaces` property.

```
mapPort(hFPGA, hPort_h_in1);
hFPGA.Interfaces
```

```
ans = 

  AXI4Slave with properties:

      InterfaceID: "AXI4-Lite"
      BaseAddress: "0xA0000000"
     AddressRange: "0x10000"
      WriteDriver: [1×1 fpgaio.driver.AXIMemoryMappedIIOWrite]
       ReadDriver: [1×1 fpgaio.driver.AXIMemoryMappedIIORead]
       InputPorts: "h_in1"
      OutputPorts: [0×0 string]
```

You can also specify this information for ports mapped to AXI4-Stream interfaces.

```
hPort_x_in_data = hdlcoder.DUTPort("x_in_data", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");

hPort_y_out_data = hdlcoder.DUTPort("y_out_data", ...
    "Direction", "OUT", ...
    "DataType", numerictype(1,32,20), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");
```

To write to or read from the DUT ports in the generated HDL IP core, map the ports to the AXI interface by using the `mapPort` function.

```
mapPort(hFPGA, [hPort_x_in_data, hPort_y_out_data]);
```

After you map the ports to the interfaces, this information is saved on the `fpga` object as the `Interfaces` property.

```
hFPGA

hFPGA = 

   fpga with properties:

         Vendor: "Xilinx"
     Interfaces: [1×2 fpgaio.interface.InterfaceBase]

hFPGA.Interfaces


ans = 

   AXI4Slave with properties:

      InterfaceID: "AXI4-Lite"
      BaseAddress: "0xA0000000"
     AddressRange: "0x10000"
      WriteDriver: [1×1 fpgaio.driver.AXIMemoryMappedIIOWrite]
       ReadDriver: [1×1 fpgaio.driver.AXIMemoryMappedIIORead]
       InputPorts: "h_in1"
      OutputPorts: [0×0 string]

   AXI4Stream with properties:

          InterfaceID: "AXI4-Stream"
          WriteEnable: 1
           ReadEnable: 1
     WriteFrameLength: 1024
      ReadFrameLength: 1024
          WriteDriver: [1×1 fpgaio.driver.AXIStreamIIOWrite]
           ReadDriver: [1×1 fpgaio.driver.AXIStreamIIORead]
           InputPorts: "x_in_data"
          OutputPorts: "y_out_data"
```

4 To test the HDL IP core functionality, use the `readPort` and `writePort` functions to write data to or read data from these ports.

```
writePort(hFPGA, "h_in1", 5);

writePort(hFPGA, "x_in", sin(linspace(0, 2*pi, 1024)));
data = readPort(hFPGA, "y_out");
```

5 After you have tested the HDL IP core, you can release the hardware resource associated with the `fpga` object by using the `release` function.

```
release(hFPGA)
```

6 For an example on creating a custom interface script and prototyping your design on a target FPGA board over an Ethernet connection, see “Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands” on page 39-241.

JTAG-Based Interface

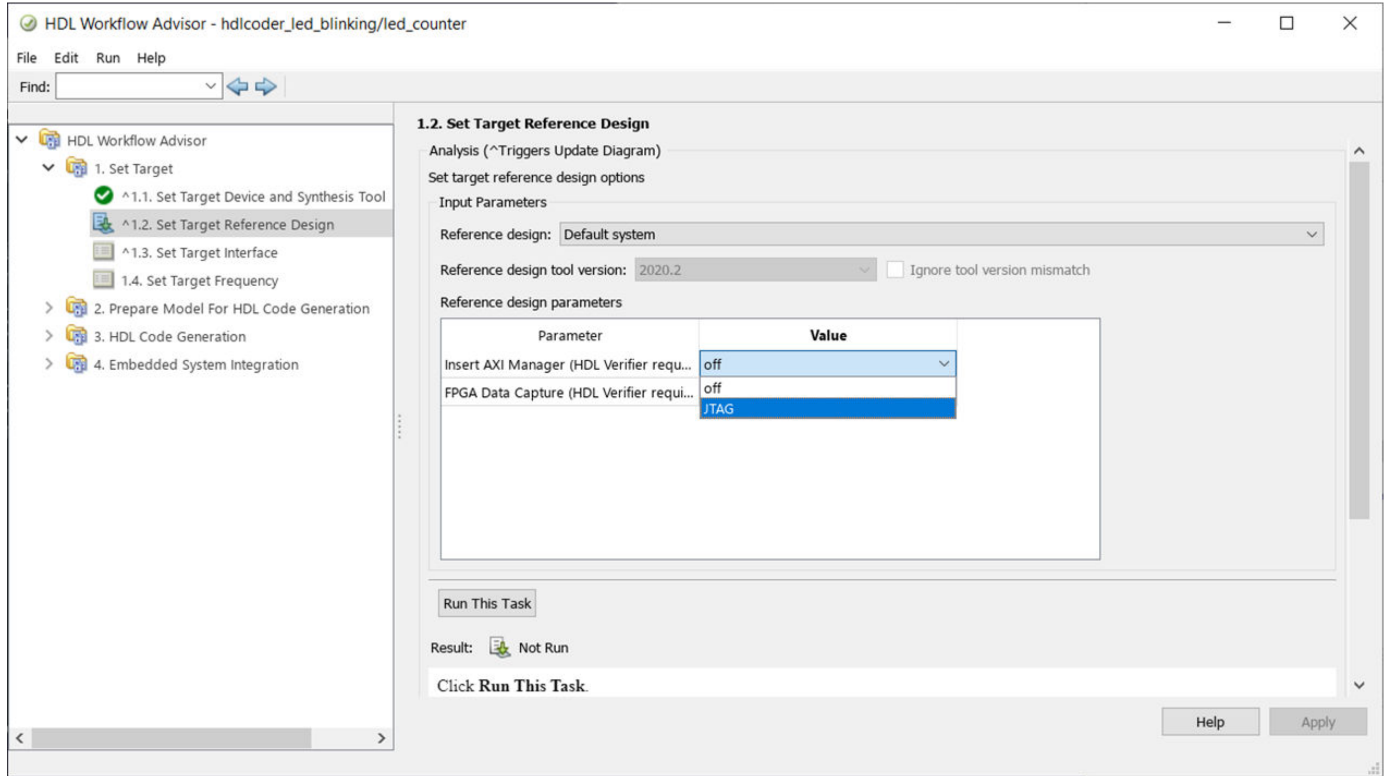
For standalone FPGA boards that do not have an embedded ARM processor, you can insert the JTAG AXI Manager IP into your reference design. Create a script that uses the `aximanager` object that connects to the IP over a physical JTAG cable. This script enables read and write commands to slave memory locations from the MATLAB command line.

To use AXI Manager:

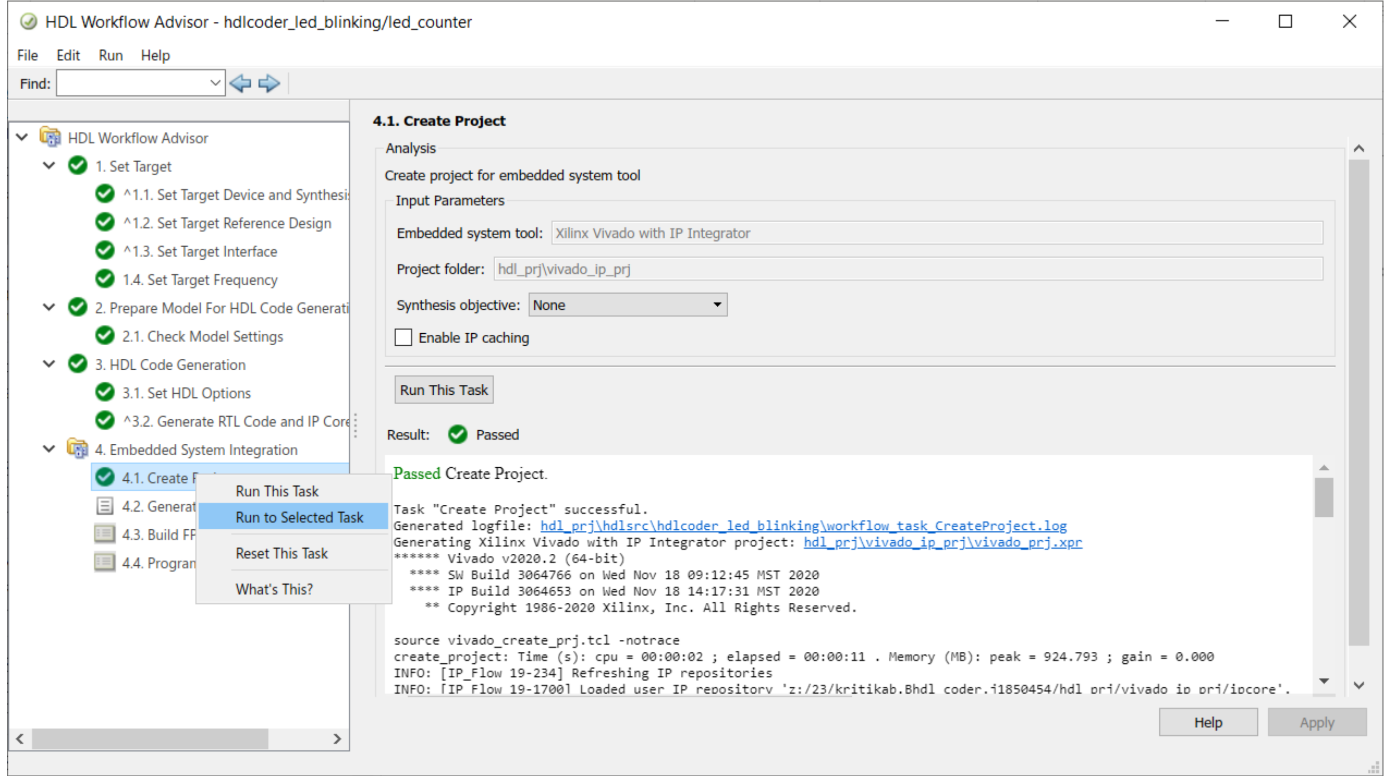
- Install the HDL Verifier hardware support packages.
- Do not target standalone boards that do not have the `hRD.AXI4SlaveInterface` functionality or boards that are based on Xilinx ISE.

Insert JTAG AXI Manager

When you run the IP Core Generation workflow, in the **Set Target > Set Target Reference Design** task, choose `Default system` for **Reference Design** and set **Insert AXI Manager (HDL Verifier required)** to `JTAG`.



To generate the IP Core and create a project with AXI Manager IP, right-click the **Create Project** task and select **Run to Selected Task**.



Control the HDL Coder IP Core at the MATLAB Command Line

You can now interact with your target FPGA board by using the JTAG AXI Manager feature. Create an object by using the `aximanager` object. Use the created object as the driver for the `addAXI4SlaveInterface` function.

```
% Create an "aximanager" object
hAXIMDriver = aximanager("Xilinx");

% Pass it into the addInterface command
addAXI4SlaveInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID",  "AXI4-Lite", ...
    "BaseAddress",  0xB0000000, ...
    "AddressRange", 0x10000, ...
    ... % Driver properties
    "WriteDriver", hAXIMDriver, ...
    "ReadDriver", hAXIMDriver, ...
    "DriverAddressMode", "Full");
```

For an example on how to interface with a target board over JTAG, see “Use JTAG AXI Manager to Control HDL Coder Generated IP Core” on page 40-333.

See Also

More About

- “Generate and Manage FPGA I/O Host Interface Scripts” on page 39-68
- “Generate Software Interface Model to Probe and Rapidly Prototype HDL IP Core” on page 39-84
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9

Getting Started with Targeting Xilinx Zynq Platform

This example shows how to use the hardware-software co-design workflow to blink LEDs at various frequencies on the Xilinx® Zynq® ZC702 evaluation kit.

Introduction

This example is a step-by-step guide that helps you use HDL Coder™ to generate a custom HDL IP core which blinks LEDs on the Xilinx Zynq ZC702 evaluation kit, and shows how to use Embedded Coder® to generate C code that runs on the ARM® processor to control the LED blink frequency.

You can use MATLAB® and Simulink® to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your design for hardware and software implementation on the Zynq-7000 AP SoC by deciding which system elements will be performed by the programmable logic, and which system elements will run on the ARM Cortex-A9.

Using the guided workflow shown in this example, you automatically generate HDL code for the programmable logic using HDL Coder, generate C code for the ARM using Embedded Coder, and implement the design on the Xilinx Zynq Platform.

In this workflow, you perform the following steps:

1. Set up your Zynq hardware and tools.
2. Partition your design for hardware and software implementation.
3. Generate an HDL IP core using HDL Workflow Advisor.
4. Integrate the IP core into a Xilinx Vivado project and program the Zynq hardware.
5. Generate a software interface model.
6. Generate C code from the software interface model and run it on the ARM Cortex-A9 processor.
7. Tune parameters and capture results from the Zynq hardware using External Mode.

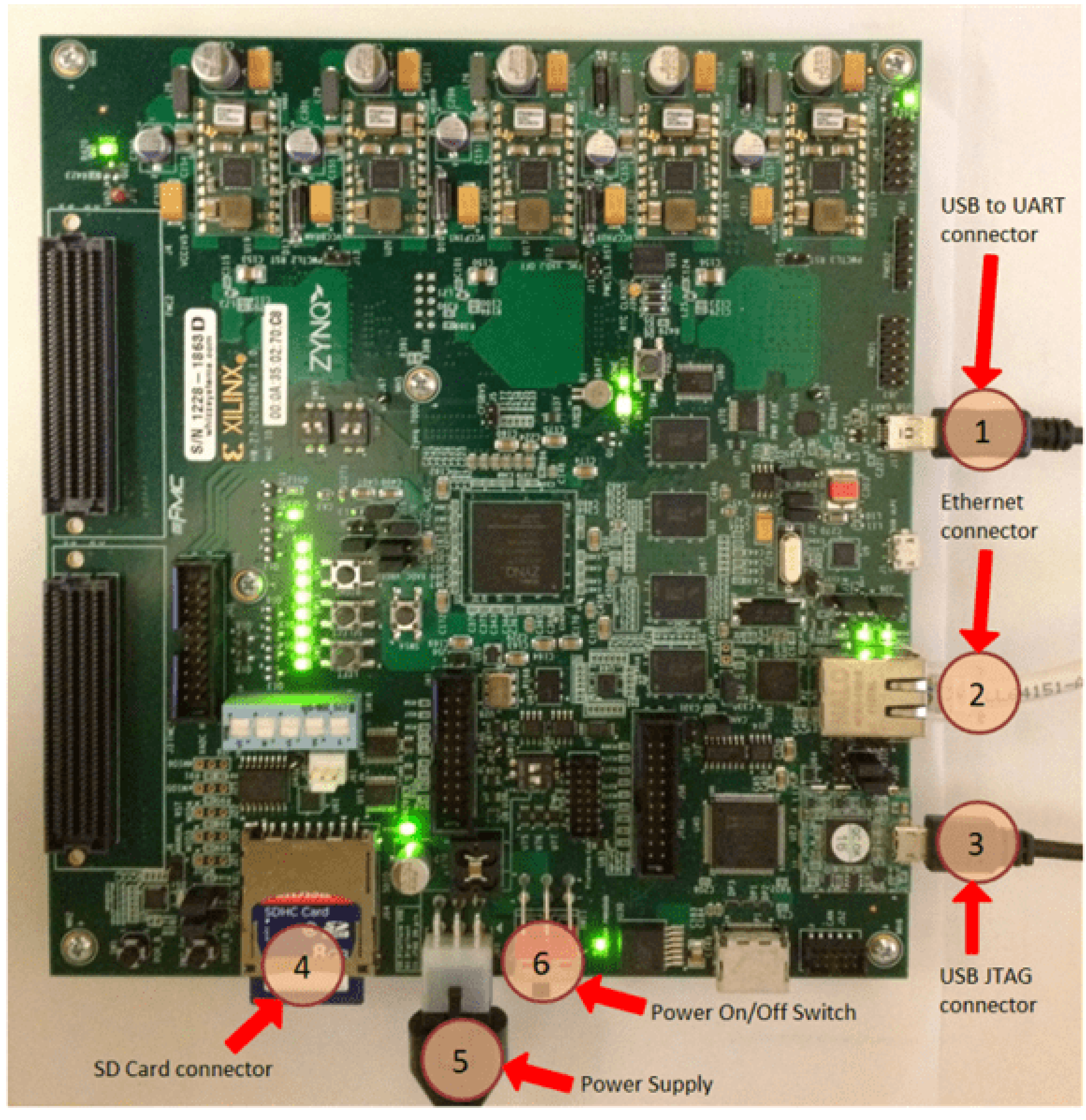
For more information, refer to other more advanced examples, and the HDL Coder and Embedded Coder documentation.

Requirements

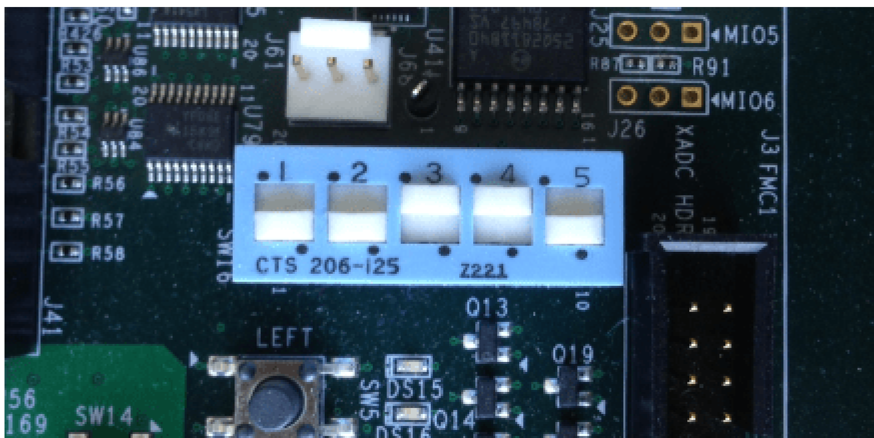
1. Xilinx Vivado Design Suite, with supported version listed in the “HDL Language Support and Supported Third-Party Tools and Hardware”.
2. Xilinx Zynq-7000 SoC ZC702 Evaluation Kit
3. HDL Coder Support Package for Xilinx FPGA and SoC Devices

Set Up Zynq Hardware and Tools

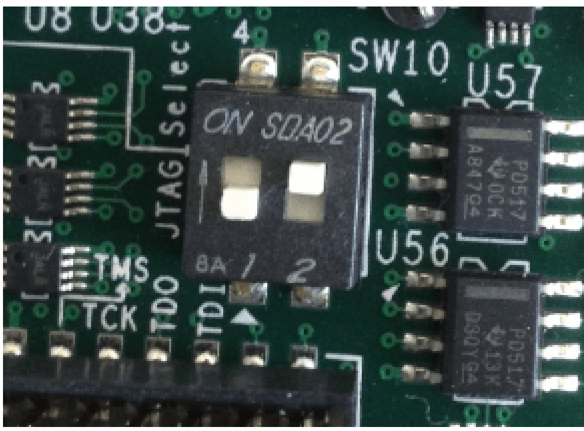
1. Set up the Xilinx Zynq ZC702 evaluation kit as shown in the figure below. To learn more about the ZC702 hardware setup, please refer to documentation.



2. Make sure the SW16 switch is set as shown in the figure below, so you can boot Linux from the SD card.



3. Make sure the `SW10` switch (JTAG chain input select two-position DIP switch) is set as shown in the figure below, so you can use the Digilent USB-to-JTAG interface (U23). Position 1 is off and Position 2 is on.



4. Connect your computer to the USB UART connector using a Micro-USB cable. Make sure your USB device drivers, such as for the Silicon Labs CP210x USB to UART Bridge, are installed correctly. If not, search for the drivers online and install them.

5. Connect your computer and the Zynq board using an Ethernet cable.

6. Install the HDL Coder Support Package for Xilinx FPGA and SoC Devices if you haven't already. To start the installer, go to the MATLAB toolstrip and click **Add-Ons** > **Get Hardware Support Packages**. For more information, see “Install Support for Xilinx Zynq Platform” (Embedded Coder).

7. Make sure you are using the SD card image provided by the HDL Coder Support Package for Xilinx FPGA and SoC Devices. If you need to update your SD card image, see the Hardware Setup section of “Install Support for Xilinx Zynq Platform” (Embedded Coder).

8. Set up the Zynq hardware connection by entering the following command in the MATLAB command window:

```
h = zynq
```

The `zynq` function logs in to the hardware via COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

9. You can optionally test the serial connection using the following configuration using a program such as PuTTY™. Baud rate: `115200`; Data bits: `8`; Stop bits: `1`; Parity: `None`; Flow control: `None`. You should be able to observe Linux booting log on the serial console when you power cycle the Zynq board. You must close this serial connection before using the `zynq` function again.

10. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
```

Partition Design for Hardware and Software Implementation

The first step of the Zynq hardware-software co-design workflow is to decide which parts of your design to implement on the programmable logic, and which parts to run on the ARM processor.

Group all the blocks you want to implement on programmable logic into an atomic subsystem. This atomic subsystem is the boundary of your hardware-software partition. All the blocks inside this subsystem are implemented on programmable logic, and all the blocks outside this subsystem run on the ARM processor.

In this example, the subsystem **led_counter** is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem **led_counter** are for software implementation.

In Simulink, you can use the **Slider Gain** or **Manual Switch** block to adjust the input values of the hardware subsystem. In the embedded software, this means the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. The output port, **Read_Back**, can be used to read data back to the processor.

```
open_system('hdlcoder_led_blinking');
```

Generate HDL IP Core

You can generate a reusable IP core module from a Simulink model using HDL Coder. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic that connects the IP core to the embedded processor. HDL Coder packages the generated files into a folder you specify. You can then integrate the generated IP core with a larger FPGA embedded design in the Xilinx Vivado environment.

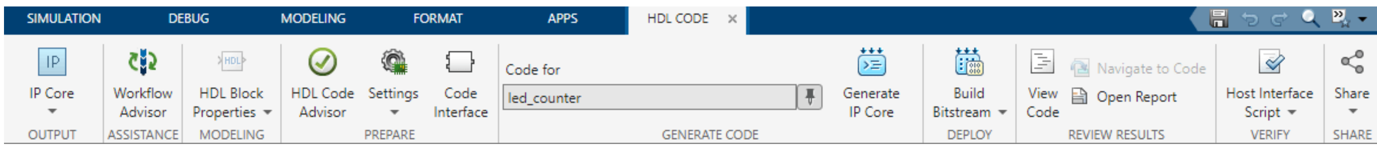
For an overview of how to generate an IP core for a specific hardware platform, see “Targeting FPGA & SoC Hardware Overview” on page 39-3. For more information on different ways to generate an IP core using HDL Coder, see “Comparison of IP Core Generation Techniques” on page 39-27.

Prepare Model for IP Core Generation

To generate an IP core from the `hdlcoder_led_blinking/led_counter` subsystem, prepare your model by using the configuration parameters, configure your design by using the IP Core editor, and generate the IP core by using the **HDL Code** tab of the Simulink Toolstrip.

1. In the **Apps** tab, click **HDL Coder**. In the **HDL Code** tab, in the **Output** section, set the drop-down button to **IP Core**.

2. Select the `led_counter` subsystem which is the device under test (DUT) for this example. Make sure that **Code for** is set to this subsystem. To remember the selection, you can pin this option.



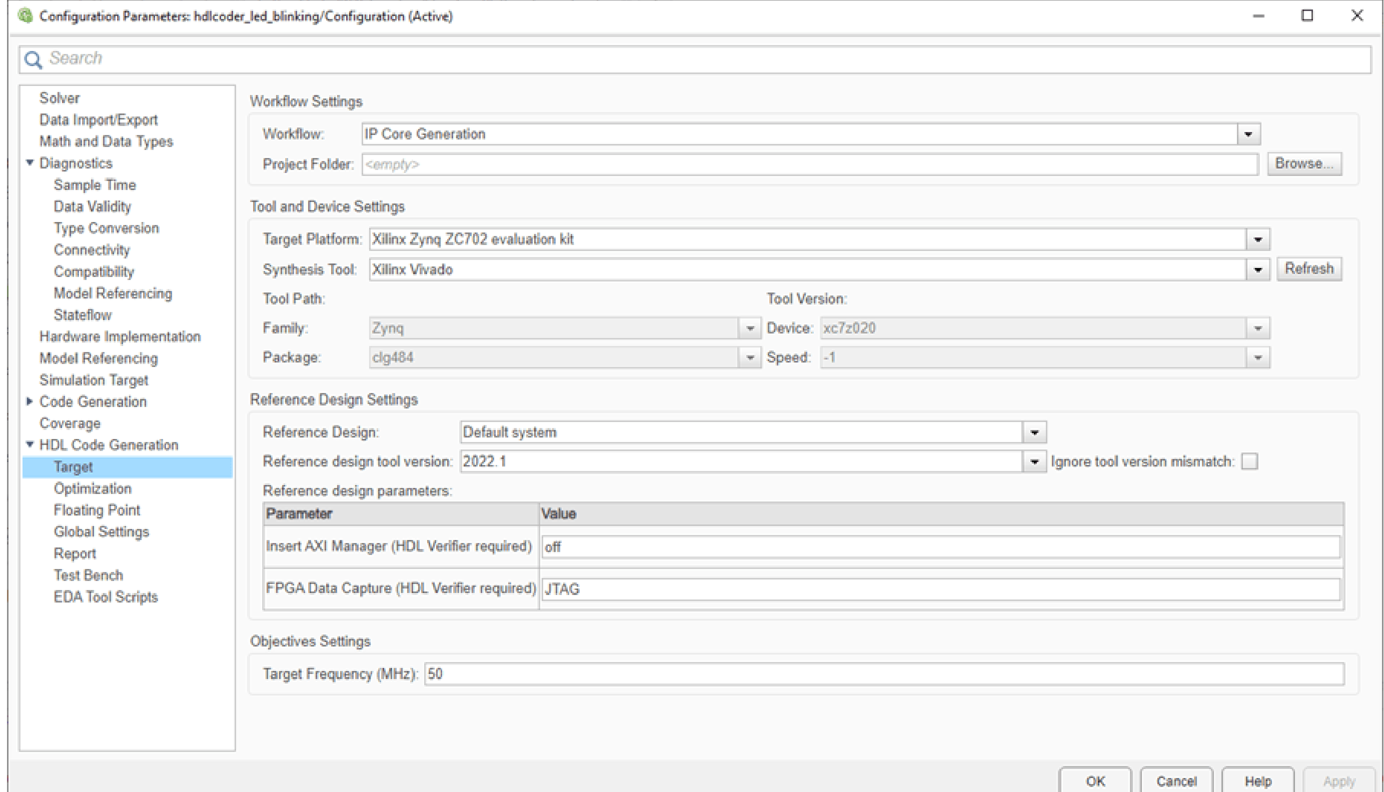
3. Open the **HDL Code Generation > Target** tab of Configuration Parameters dialog box by clicking the **Settings** button.

4. Set the **Target Platform** parameter to `Xilinx Zynq ZC702 evaluation kit`. If this option does not appear, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Xilinx FPGA and SoC Devices and follow the instructions to complete the installation. Ensure the **Synthesis Tool** is set to `Xilinx Vivado`.

5. Ensure that the **Reference Design** parameter is set to `Default system`.

6. Set the **Target Frequency** to `50` MHz.

7. Click **OK** to save your updated settings.

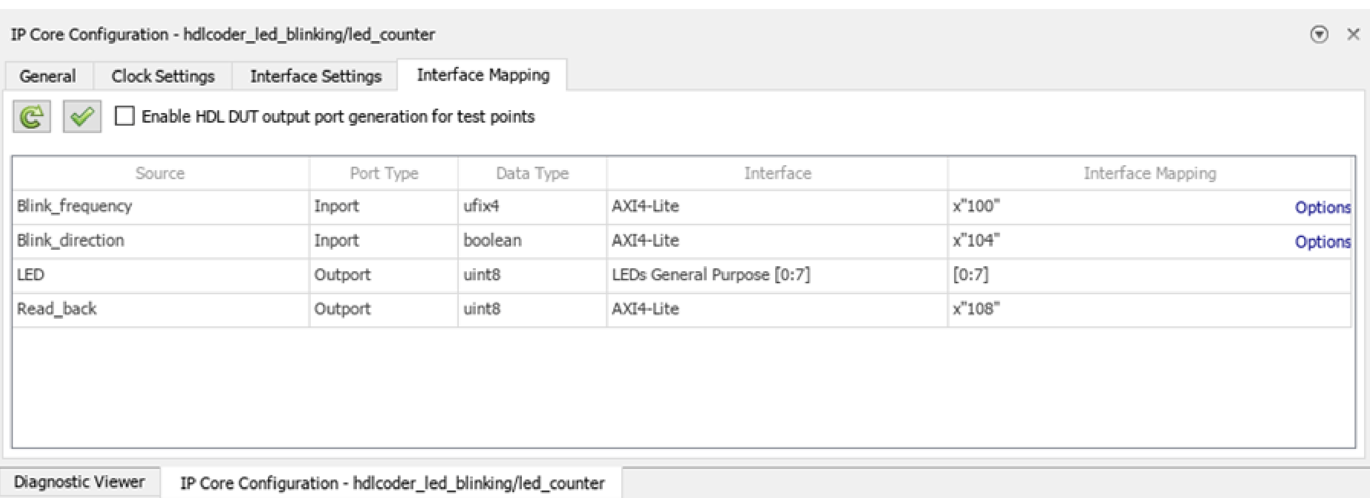


Configure Design and Target Interface

Configure your design to map to the target hardware by mapping the DUT ports to IP core target hardware and setting DUT-level IP core options. In this example, the input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4-Lite interface, so HDL Coder generates AXI interface

accessible registers for them. The **LED** output port is mapped to an external interface, `LEDs General Purpose [0:7]`, which connects to the LED hardware on the Zynq board.

1. In Simulink, in the **HDL Code** tab, click **Target Interface** to open the IP Core editor.
2. Select the **Interface Mapping** tab to map each DUT port to one of the IP core target interfaces. The generated design can then communicate with the rest of the hardware system when it is deployed. If no mapping table appears, click the Reload IP core settings button to compile the model and repopulate the DUT ports and their data types.
3. For the DUT ports **Blink_frequency**, **Blink_direction**, and **Read_back**, set the cells in the **Interface** column to `AXI4-Lite`.
4. For the **LED** output port, set the cell in the **Interface** column to `LEDs General Purpose [0:7]`.
5. Validate your settings by clicking the Validate IP core settings button.



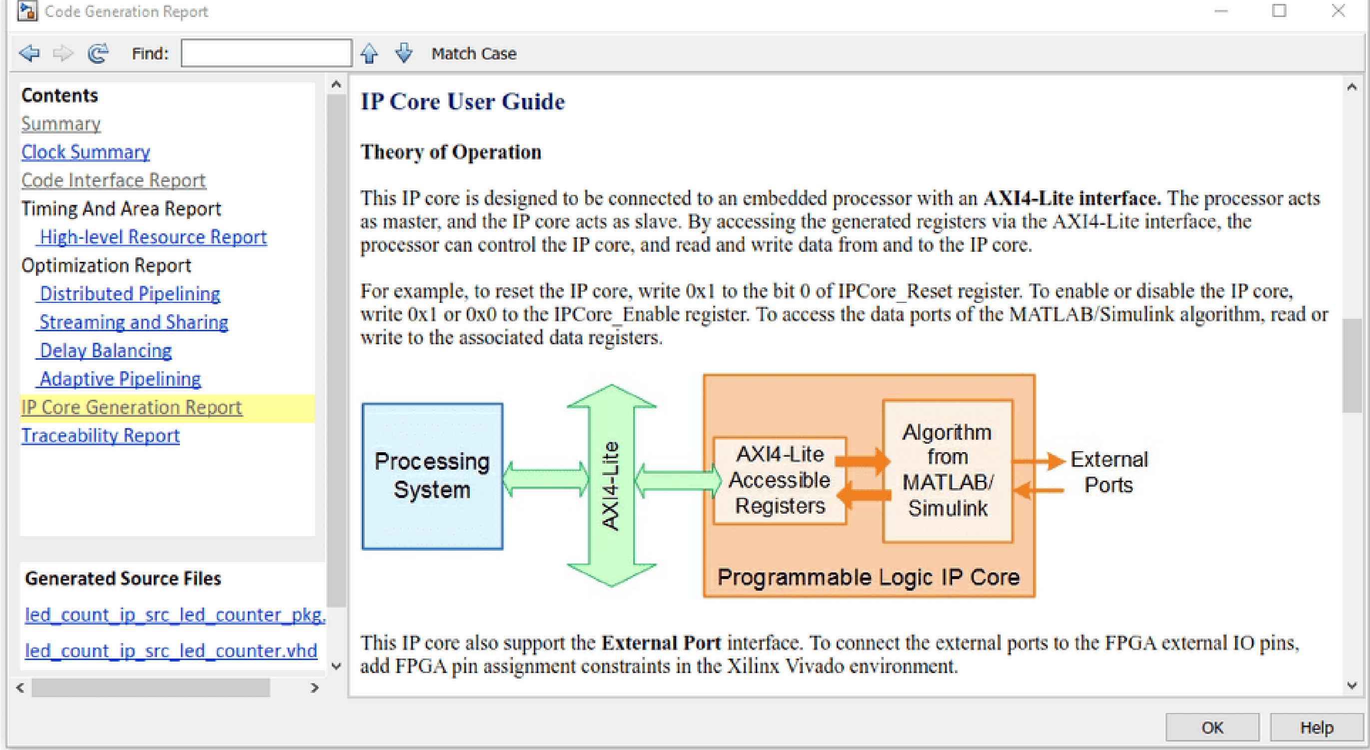
In the IP Core editor, you can optionally adjust the DUT-level IP core settings for your target hardware by:

- Using the **General** tab to configure top-level settings, such as the name of the IP core and whether to generate an IP core report.
- Using the **Clock Settings** tab to configure clock-related settings.
- Using the **Interface Settings** tab to configure interface-related settings, such as the register interface and FPGA data capture properties.

Generate IP Core

After you configure the IP core settings and mappings for your design, you can generate an IP core. In the Simulink Toolstrip, in the **HDL Code** tab, click **Generate IP Core**. After you generate the custom IP core, the IP core files are in the `ipcore` folder in your current directory. To specify a top-level project folder for the `ipcore` folder to be stored along with all other generated files, in the Configuration Parameters dialog box, use the **Project Folder** parameter in the **HDL Code Generation > Target** tab. If the **Project Folder** parameter is empty, HDL Coder saves the generated files in the current directory.

Generating an IP core also generates the code generation report. In the Code Generation Report window, in the left pane, click the **IP Core Generation Report**. The report describes the behavior and contents of the generated custom IP core.



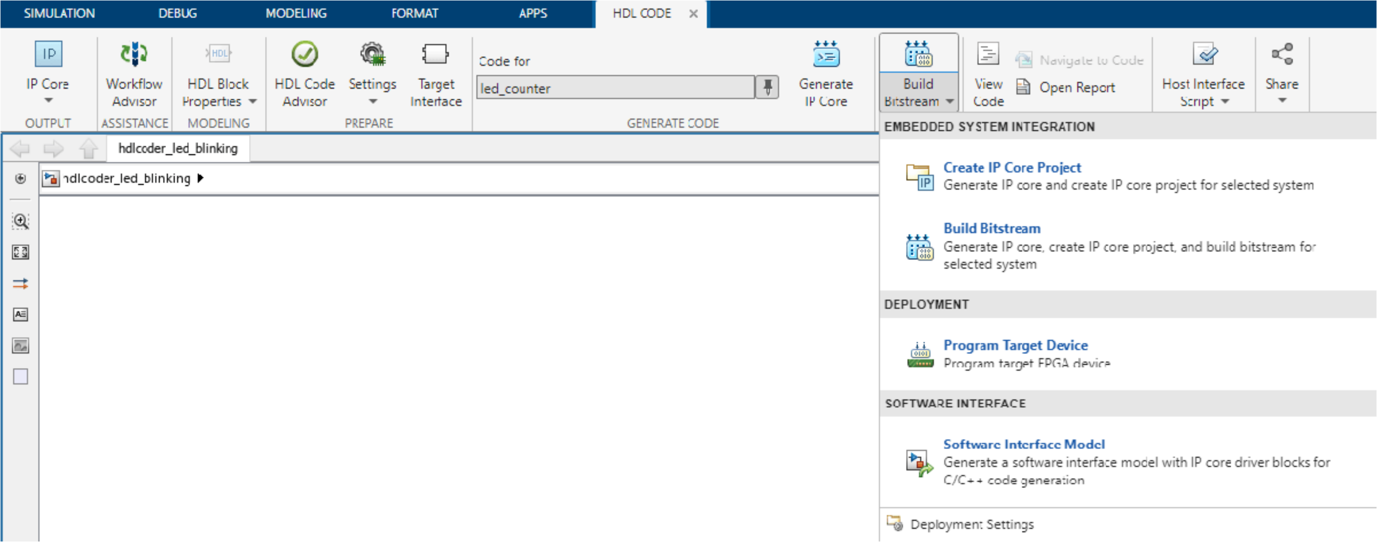
Integrate IP Core with Xilinx Vivado Environment

Next, insert your generated IP core into a embedded system reference design by creating a project, generating an FPGA bitstream, and downloading the bitstream to the Zynq hardware.

The reference design is a predefined Xilinx Vivado project that contains all the elements the Xilinx software needs to deploy your design to the Zynq platform, except for the custom IP core and embedded software.

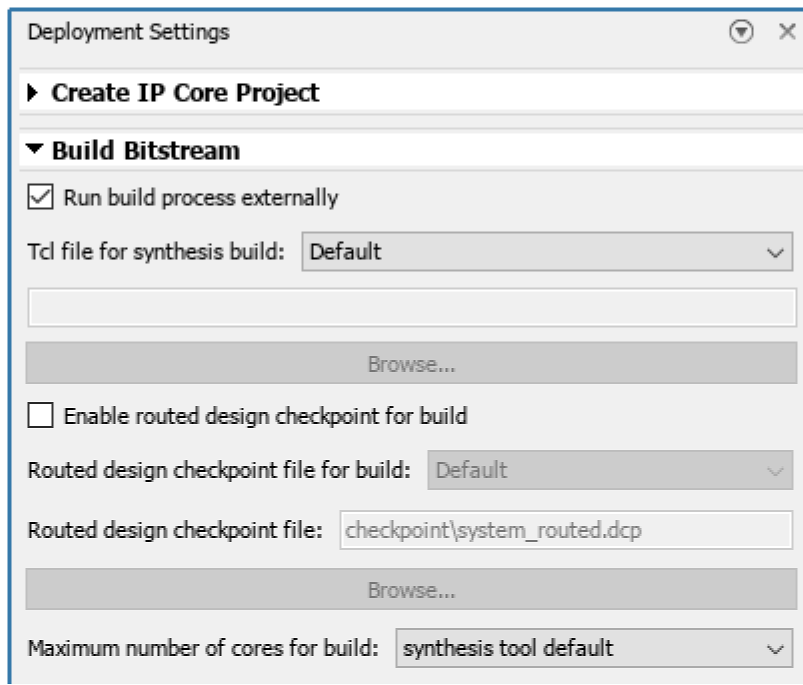
Create IP Core Project

Integrate your generated IP core into the Xilinx platform by creating a Vivado™ project that organizes and maintains the files associated with the IP core. To create a Vivado project in the Simulink Toolstrip, in the **HDL Code** tab, select **Build Bitstream > Create IP Core Project**. HDL Coder generates an IP integrator embedded design and links it in the Diagnostic Viewer.



Configure Deployment Settings

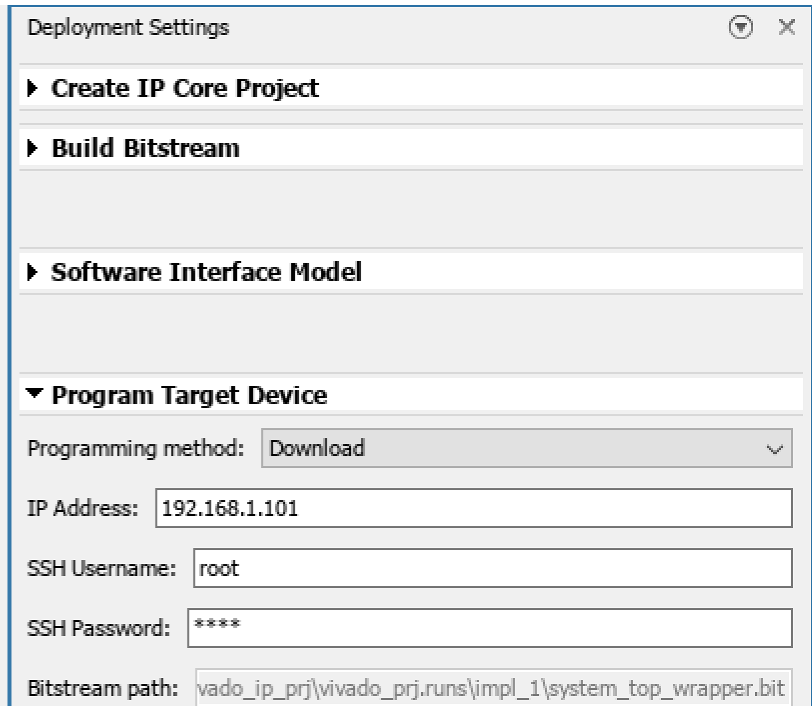
Next, configure the build bitstream settings. In the Simulink Toolstrip in the **HDL Code** tab**,** select **Build Bitstream > Deployment Settings**. In the Deployment Settings window under the **Build Bitstream** section, select **Run build process externally** to run the Xilinx synthesis tool in a separate window than the current MATLAB session.



In the Deployment Settings window, under the **Program Target Device** section:

- Set **Programming method to** `Download` to download the bitstream onto your target Zynq board SD card and load your design when the board power cycles.
- Set **IP Address** to your target board IP address.

- Set **SSH username** and **SSH Password** to your target board settings.

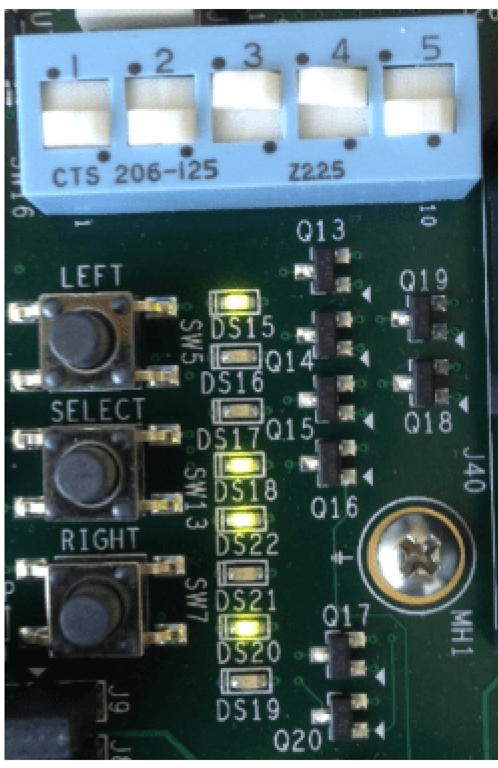


Generate Bitstream and Program Target Device

To generate the bitstream file, in the Simulink Toolstrip, in the **HDL Code** tab, click **Build Bitstream** and wait until the synthesis tool runs in the external window.

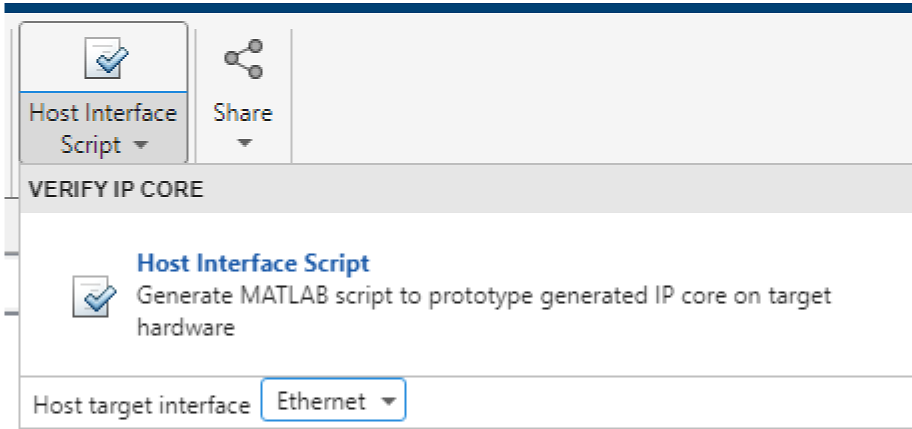
To download the bitstream, in the Simulink Toolstrip, in the **HDL Code** tab**,** select **Build Bitstream > Program Target Device**.

After you program the FPGA hardware, the LED starts blinking on your Zynq board.



Prototype and Verify IP Core on Hardware

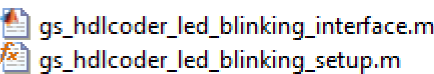
Verify your generated IP core on the hardware by using MATLAB to generate a host interface script. This script contains MATLAB commands that connect your hardware and interact with your IP core.. To generate a host interface script file, in the Simulink Toolstrip, in the **HDL Code** tab, select **Host Interface Script > Host Interface Script**.



The Simulink Diagnostic Viewer displays a link to the generated host interface script file.

```
Host Interface Script will be generated based on last stored Interface Mapping.

### Generate host interface script

Generating new Xilinx Host Interface script: gs_hdlcoder_led_blinking_interface.m
Xilinx Host Interface script generation complete.
No driver was generated for port(s) "LED" mapped to interface "LEDs General Purpose" in the host interface script.
```

Two MATLAB files are generated in your current folder and enable you to prototype your generated IP core directly from MATLAB.



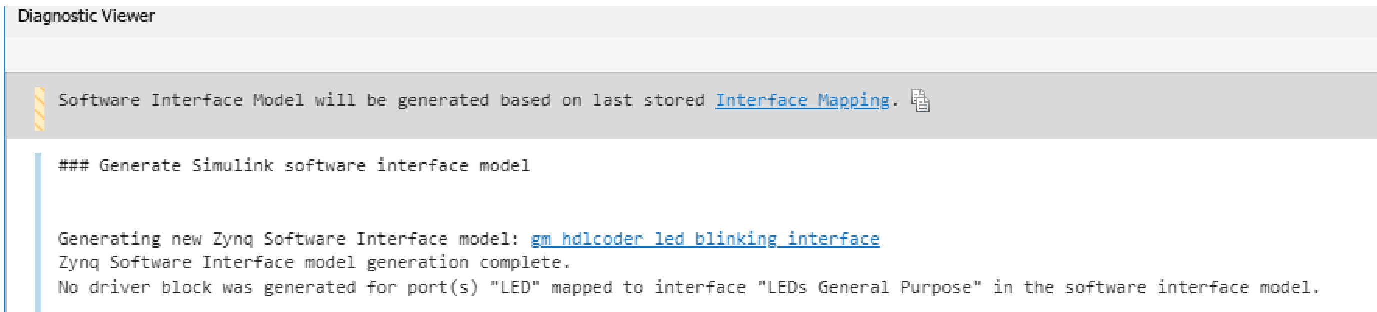
Open the generated interface script file by clicking the link in the Simulink Diagnostic Viewer. This file creates a connection to your FPGA hardware that reads and writes data and configures the `fpga` hardware object with the same ports and interfaces that were mapped in the **Target Interface** table. This function can be reused in your own scripts to recreate this configuration. For example, uncomment line 33 in the `gs_hdlcoder_led_blinking_interface.m` function and modify it to change the LED blink frequency. Run the modified script and observe the LED blink frequency changing on the hardware. In addition, uncomment line 35 in the `gs_hdlcoder_led_blinking_interface.m` function and observe the change in values by reading back the data values.

```
%% AXI4-Lite
% writePort(hFPGA, "Blink_frequency", zeros([1 1]));
% writePort(hFPGA, "Blink_direction", zeros([1 1]));
% data_Read_back = readPort(hFPGA, "Read_back");
```

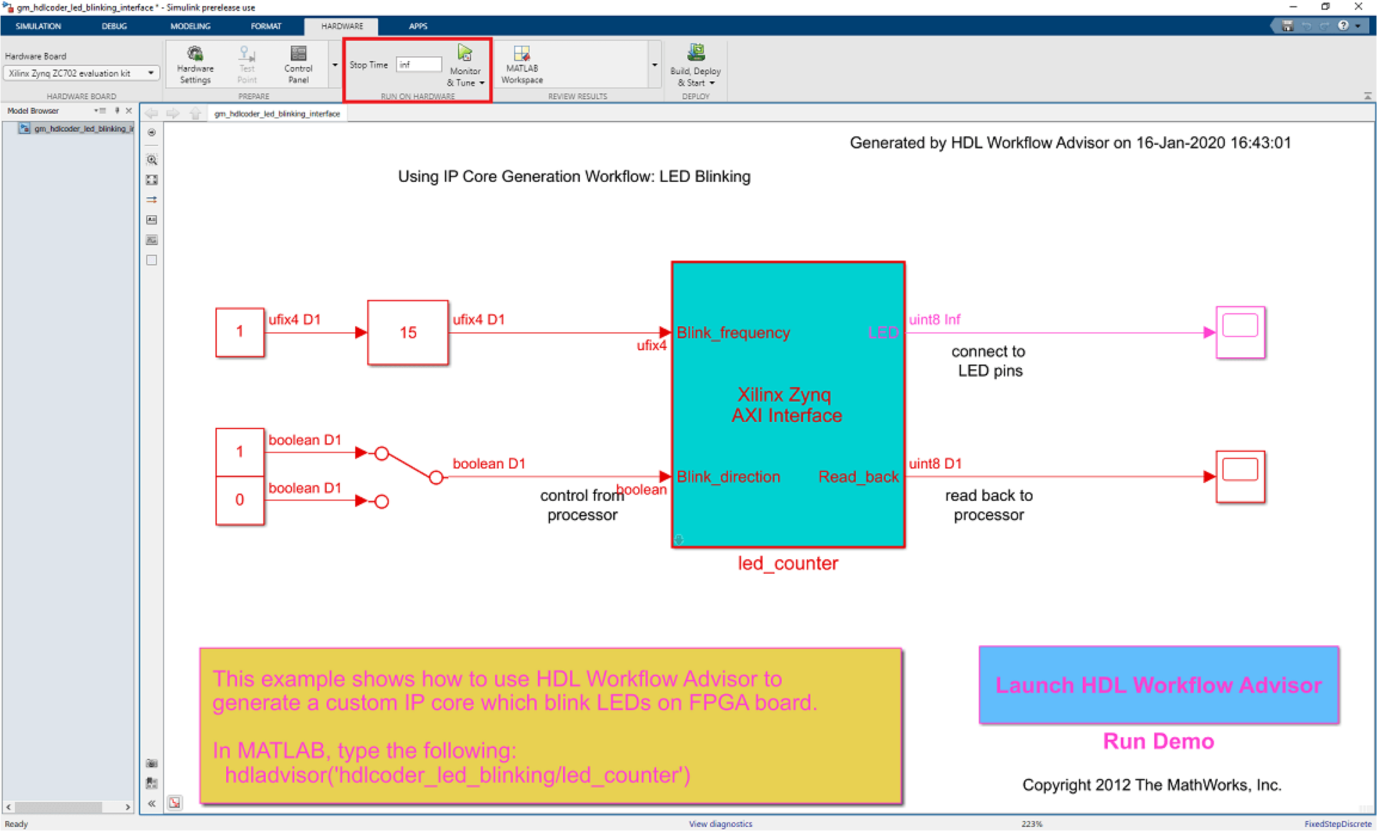
Deploy to Processor Using Software Interface Model

To target a portion of your design for the ARM processor, generate a software interface model. The software interface model contains the part of your design that runs in software. It includes all the blocks outside of the HDL subsystem, and replaces the HDL subsystem with AXI driver blocks. If you have an Embedded Coder license, you can automatically generate embedded code from the software interface model, build it, and run the executable on Linux on the ARM processor. The generated embedded software includes the AXI driver code, generated from the AXI driver blocks, that controls the HDL IP core. You can generate the software interface model at any stage of the IP core generation and IP core integration process.

To generate the software interface model, in the Simulink Toolstrip, in the **HDL Code** tab, select **Build Bitstream > Software Interface Model.** The Simulink Diagnostic Viewer displays a link to the generated software interface model.



In the generated software interface model, the `led_counter` subsystem is replaced with the AXI driver blocks that generate the interface logic between the ARM processor and FPGA.

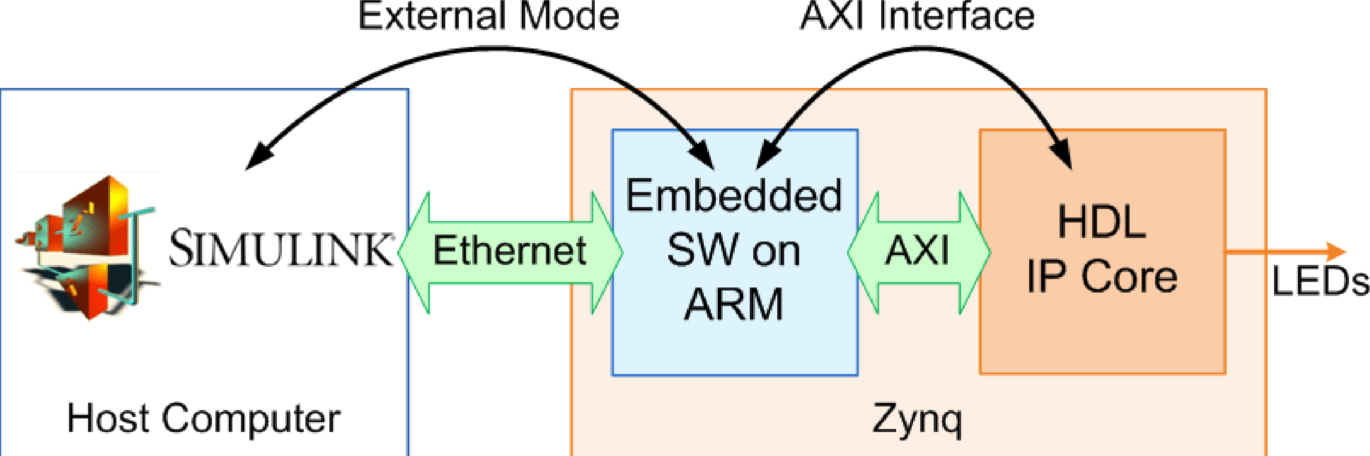


Run Software Interface Model on Zynq ZC702 Hardware

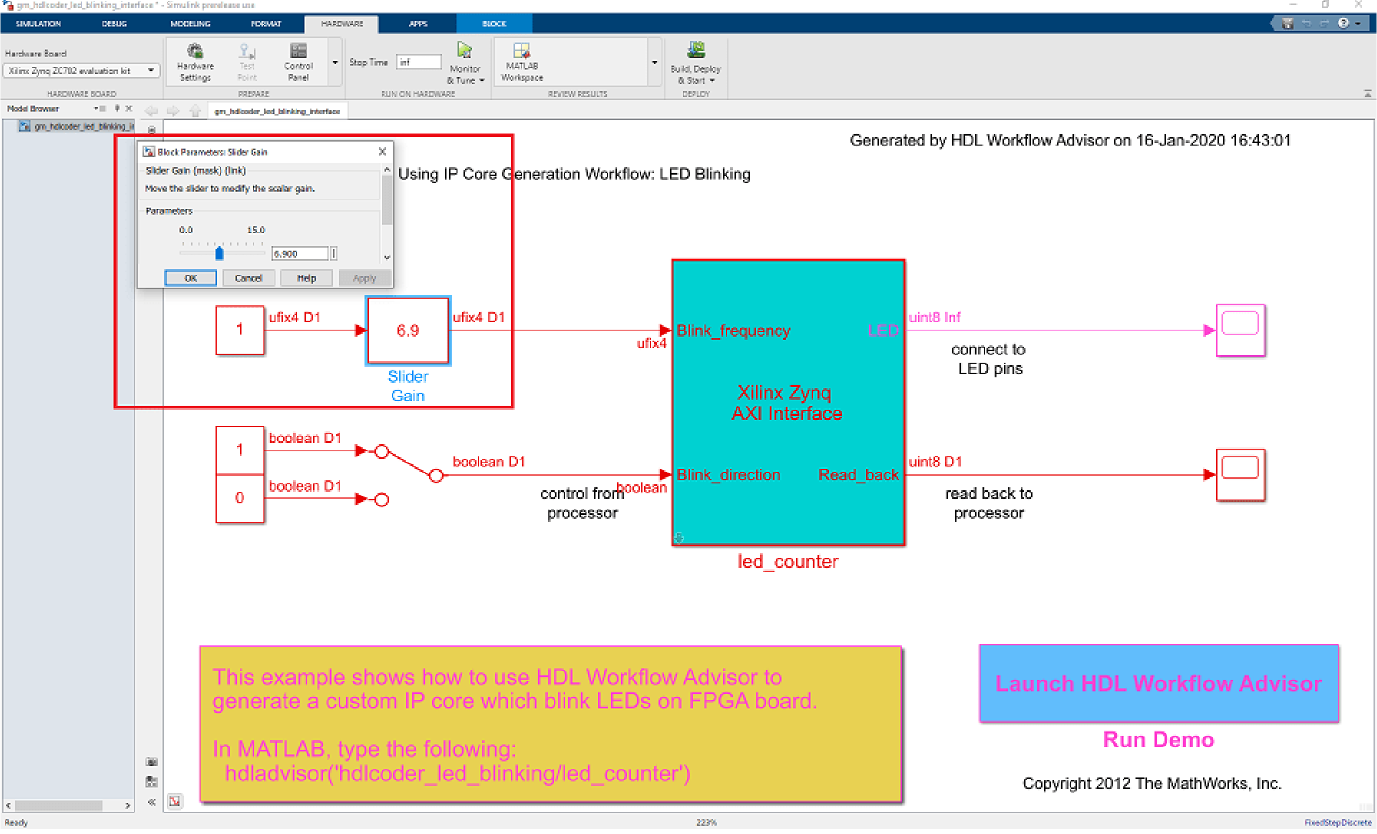
Next, configure the generated software interface model, generate the embedded C code, and run your model on the ARM processor in the Zynq hardware in external mode.

When you prototype and develop an algorithm, it is useful to monitor and tune the algorithm while it runs on hardware. You can use external mode to deploy your algorithm to the ARM processor in the Zynq hardware, and then link the algorithm with the Simulink model on the host computer through an Ethernet connection.

The Simulink model tunes and monitors the algorithm running on the hardware. Because the ARM processor is connected to the HDL IP core through the AXI interface, you can use external mode to tune parameters, and capture data from the FPGA.



1. In the generated model,in the **Hardware** tab, click **Hardware settings** to open **Configuration Parameter** dialog box.
2. In the **Solver** pane, set **Stop time** to `inf`. Click **OK**.
3. In the **Hardware** tab, click **Monitor & Tune** to run your model on the ARM processor in the Zynq ZC702 hardware in external mode. Embedded Coder builds the model, downloads the ARM executable to the Zynq ZC702 hardware, executes it, and connects the model to the executable.
4. Double-click the Slider Gain block. Change the slider and observe the change in frequency of the LED array blinking on the Zynq ZC702 hardware. Double-click the Manual Switch block to switch the direction of the blinking LEDs.
5. Double-click the Scope block connected to the **Read_back** output port. The Scope block captures the output data of the FPGA IP core.
6. When you are done changing model parameters, click the **Stop** button in the toolstrip, then close the system command window.



See Also

More About

- “Generate and Manage FPGA I/O Host Interface Scripts” on page 39-68
- “Generate Software Interface Model to Probe and Rapidly Prototype HDL IP Core” on page 39-84
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- “Comparison of IP Core Generation Techniques” on page 39-27
- “Command-Line Session for Xilinx SoC Devices”

Getting Started with Targeting Zynq UltraScale+ MPSoC Platform

This example shows how to use the hardware-software co-design workflow to blink LEDs at various frequencies on the Xilinx® Zynq® UltraScale+ MPSoC.

Introduction

This example is a step-by-step guide that helps you use the HDL Coder™ software to generate a custom HDL IP core which blinks LEDs on the Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation kit, and shows how to use Embedded Coder® to generate C code that runs on the ARM® processor to control the LED blink frequency.

You can use MATLAB® and Simulink® to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your design for hardware and software implementation on the Xilinx Zynq UltraScale+ MPSoC by deciding which system elements will be performed by the programmable logic, and which system elements will run on the ARM Cortex-A53.

Using the guided workflow shown in this example, you automatically generate HDL code for the programmable logic using HDL Coder, generate C code for the ARM processor using Embedded Coder, and implement the design on the Xilinx Zynq UltraScale+ MPSoC Platform.

In this workflow, you perform the following steps:

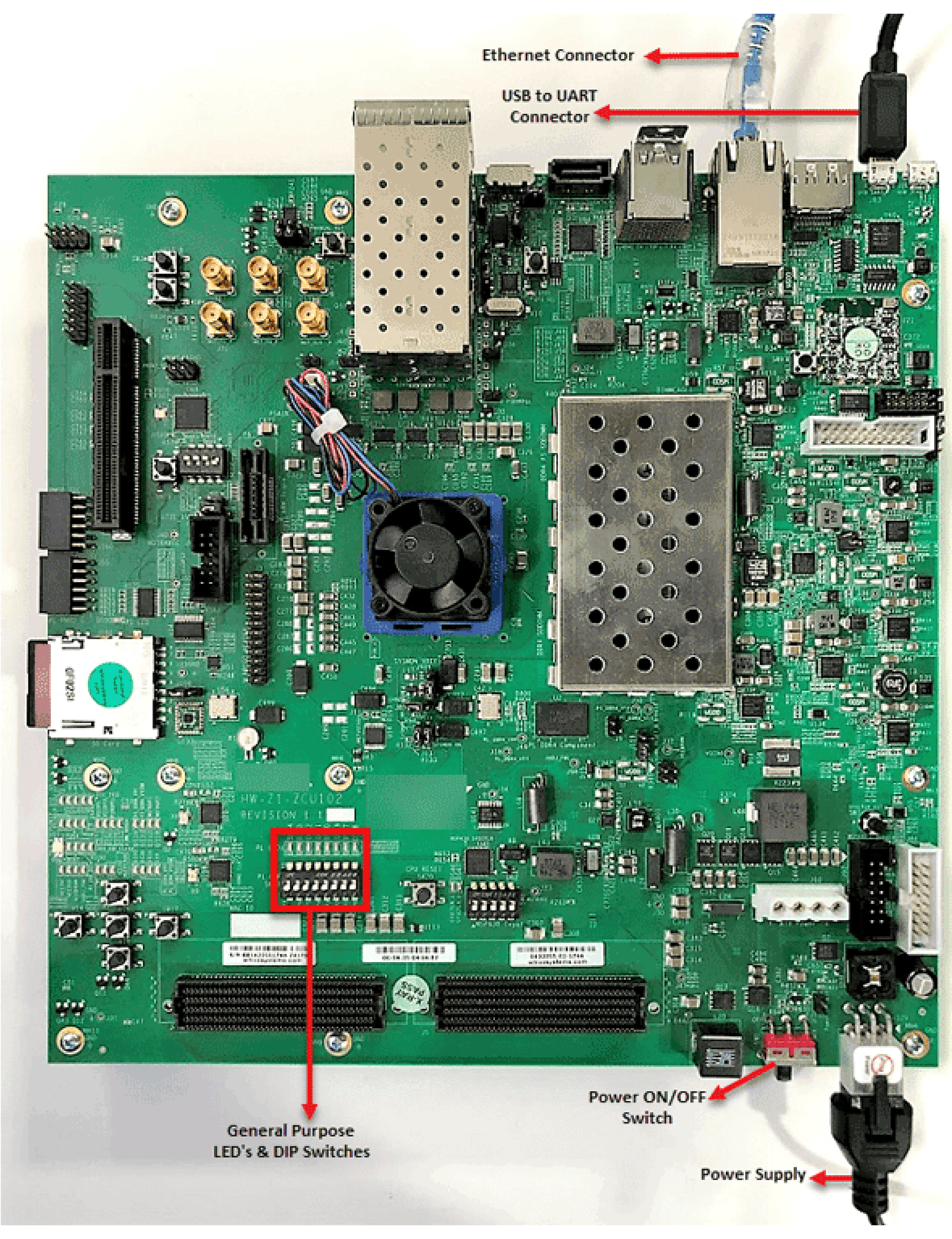
1. Set up your Xilinx Zynq UltraScale+ MPSoC ZCU102 hardware and tools.
2. Partition your design for hardware and software implementation.
3. Generate an HDL IP core using HDL Workflow Advisor.
4. Integrate the IP core into a Xilinx Vivado project and program the Xilinx Zynq UltraScale+ MPSoC hardware.
5. Generate a software interface model.
6. Generate C code from the software interface model and run it on the ARM Cortex-A53 processor.
7. Tune parameters and capture results from the Zynq hardware using External Mode.

Requirements

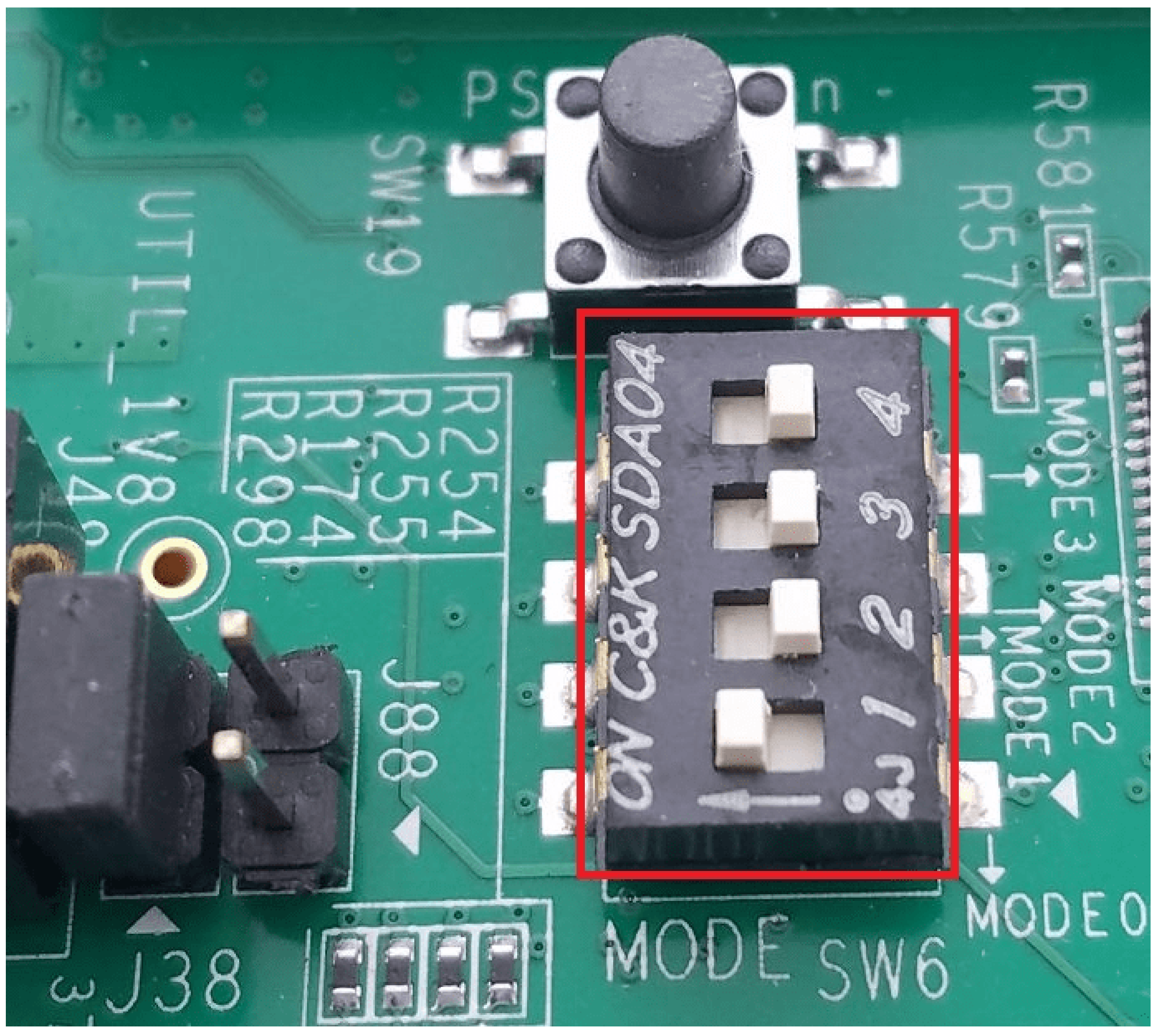
1. Xilinx Vivado Design Suite, with supported version listed in the “HDL Language Support and Supported Third-Party Tools and Hardware”.
2. Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit.
3. HDL Coder Support Package for Xilinx FPGA and SoC Devices.
4. Embedded Coder Support Package for Xilinx Zynq Platform.

Set Up Your Xilinx Zynq UltraScale+ MPSoC Hardware and Tools

1. Set up the Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation kit as shown in the figure below. To learn more about the ZCU102 hardware setup, please refer to Xilinx documentation.



1.1. Configure the ZCU102 board to boot in SD-boot mode by setting switch SW6 to **1-ON**, **2-OFF**, **3-OFF**, and **4-OFF**, as shown in figure below.



1.2 Connect your computer to the USB UART connector of ZCU102 using a Micro-USB cable. Make sure your USB device drivers, such as for the Silicon Labs CP210x USB to UART Bridge, are installed correctly. If not, search for the drivers online and install them.

1.3 Connect Xilinx Zynq UltraScale+ MPSoC board to your computer using an Ethernet cable.

2. Install the HDL Coder Support Package for Xilinx FPGA and SoC Devices and Embedded Coder Support Packages for Xilinx Zynq Platform if you haven't already.

2.1 On the MATLAB **Home** tab in the **Environment** section, Click Add-Ons > Manage Add-Ons.

2.2 In the Add-On Manager, start the hardware setup process by clicking the setup button for Embedded Coder Support Package for Xilinx Zynq Platform.

3. Make sure you are using the SD card image provided by the Embedded Coder Support Package for Xilinx Zynq Platform.

4. Set up the Zynq hardware connection by entering the following command in the MATLAB command window:

```
h = zynq
```

The `zynq` function logs in to the hardware via COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

5. You can optionally test the serial connection using the following configuration using a program such as PuTTY™. Baud rate: `115200`; Data bits: `8`; Stop bits: `1`; Parity: `None`; Flow control: `None`. You should be able to observe Linux booting log on the serial console when you power cycle the MPSoC board. You must close this serial connection before using the `zynq` function again.

6. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.ba
```

Partition Your Design for Hardware and Software Implementation

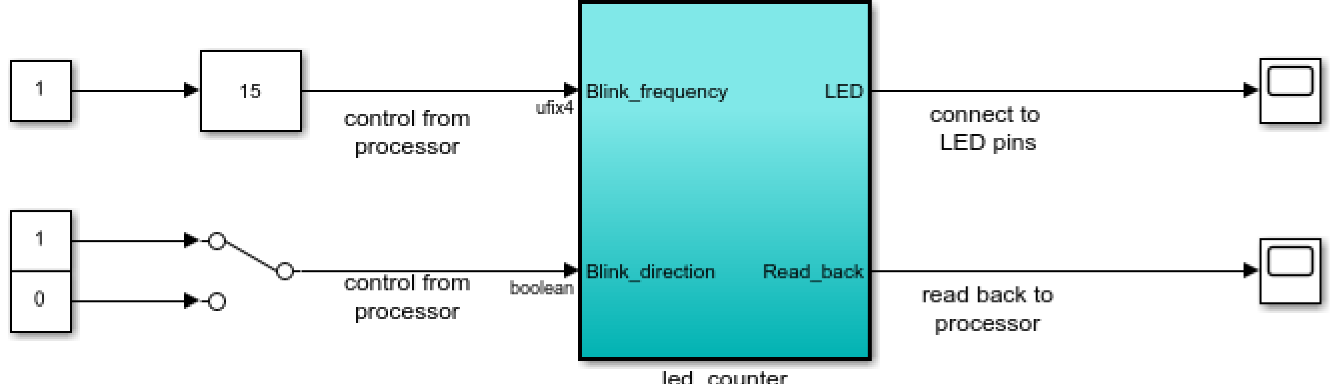
The first step of the Zynq hardware-software co-design workflow is to decide which parts of your design to implement on the programmable logic, and which parts to run on the ARM processor.

Group all the blocks you want to implement on programmable logic into an Atomic Subsystem. This Atomic Subsystem is the boundary of your hardware-software partition. All the blocks inside this subsystem will be implemented on programmable logic, and all the blocks outside this subsystem will run on the ARM processor.

In this example, the subsystem **led_counter** is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem **led_counter** are used for software implementation.

In Simulink, you can use the **Slider Gain** or **Manual Switch** block to adjust the input values of the hardware subsystem. In the embedded software, this means the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. The output port, **Read_Back**, can be used to read data back to the processor.

```
open_system('hdlcoder_led_blinking');
```



Copyright 2014-2023 The MathWorks, Inc.

Generate an HDL IP Core Using the HDL Workflow Advisor

Using the IP Core Generation workflow in the HDL Workflow Advisor enables you to automatically generate a shareable and reusable IP core module from a Simulink model. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic connecting the IP core to the embedded processor. HDL Coder packages all the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Xilinx Vivado environment.

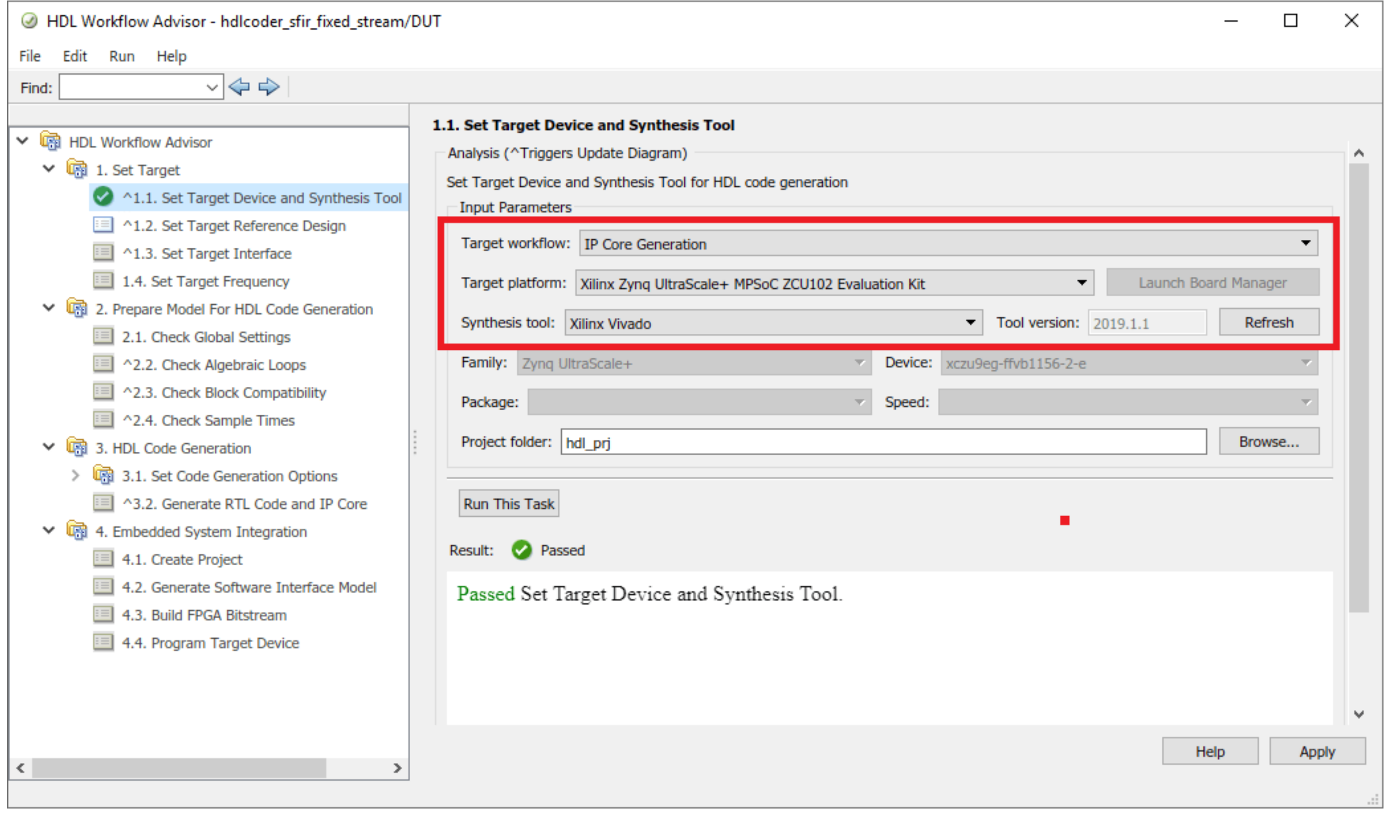
1. Start the IP core generation workflow.

1.1. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem, and choosing **HDL Code** > **HDL Workflow Advisor**.

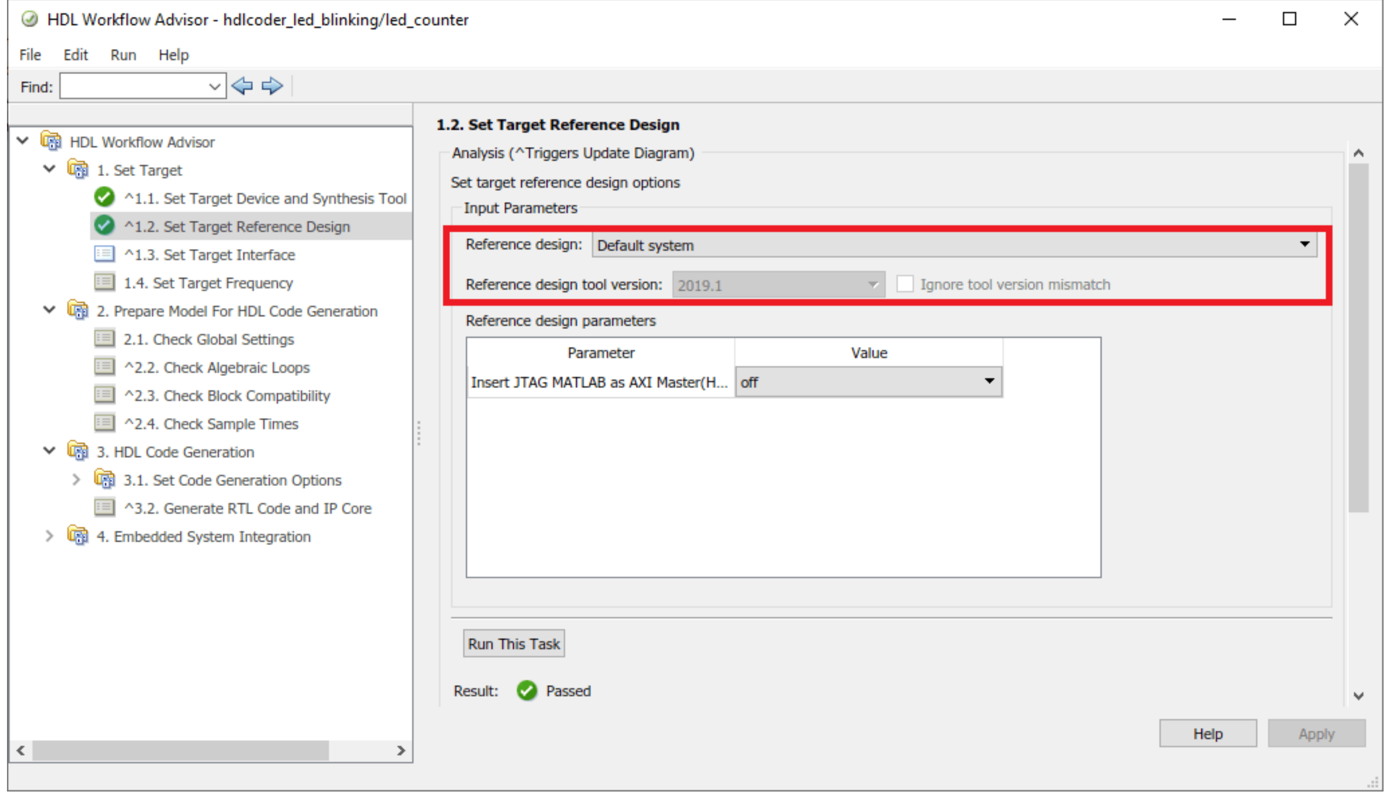
1.2. In the **Set Target** > **Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.

1.3. For **Target platform**, select **Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit**. If you don't have this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Xilinx Zynq Platform and follow the instructions provided by the Support Package Installer to complete the installation.

1.4. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.



1.5 In the **Set Target > Set Target Reference Design** task, choose **Default system**.



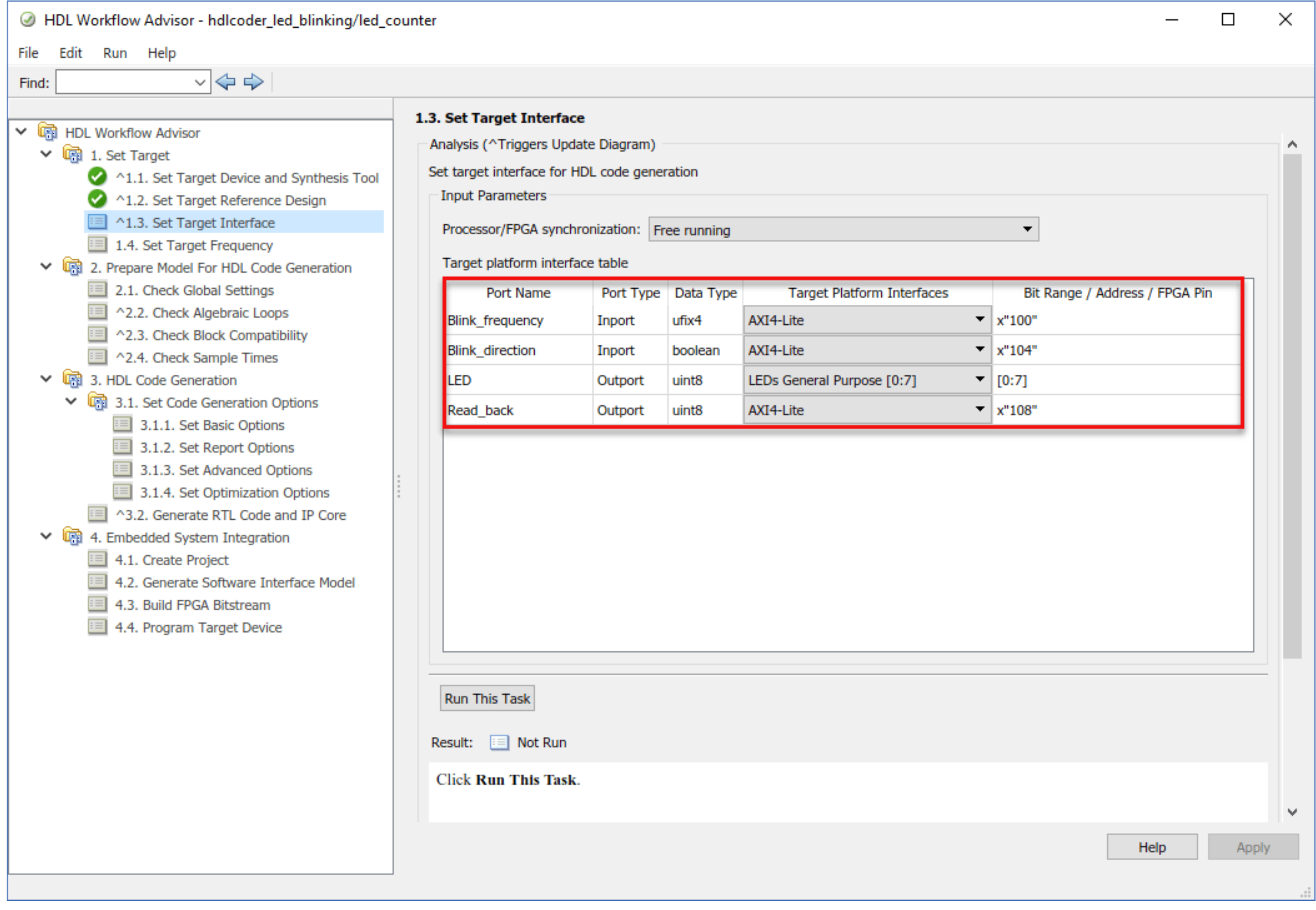
1.6. Click **Run This Task** to run the **Set Target Reference Design** task.

2. Configure the target interface.

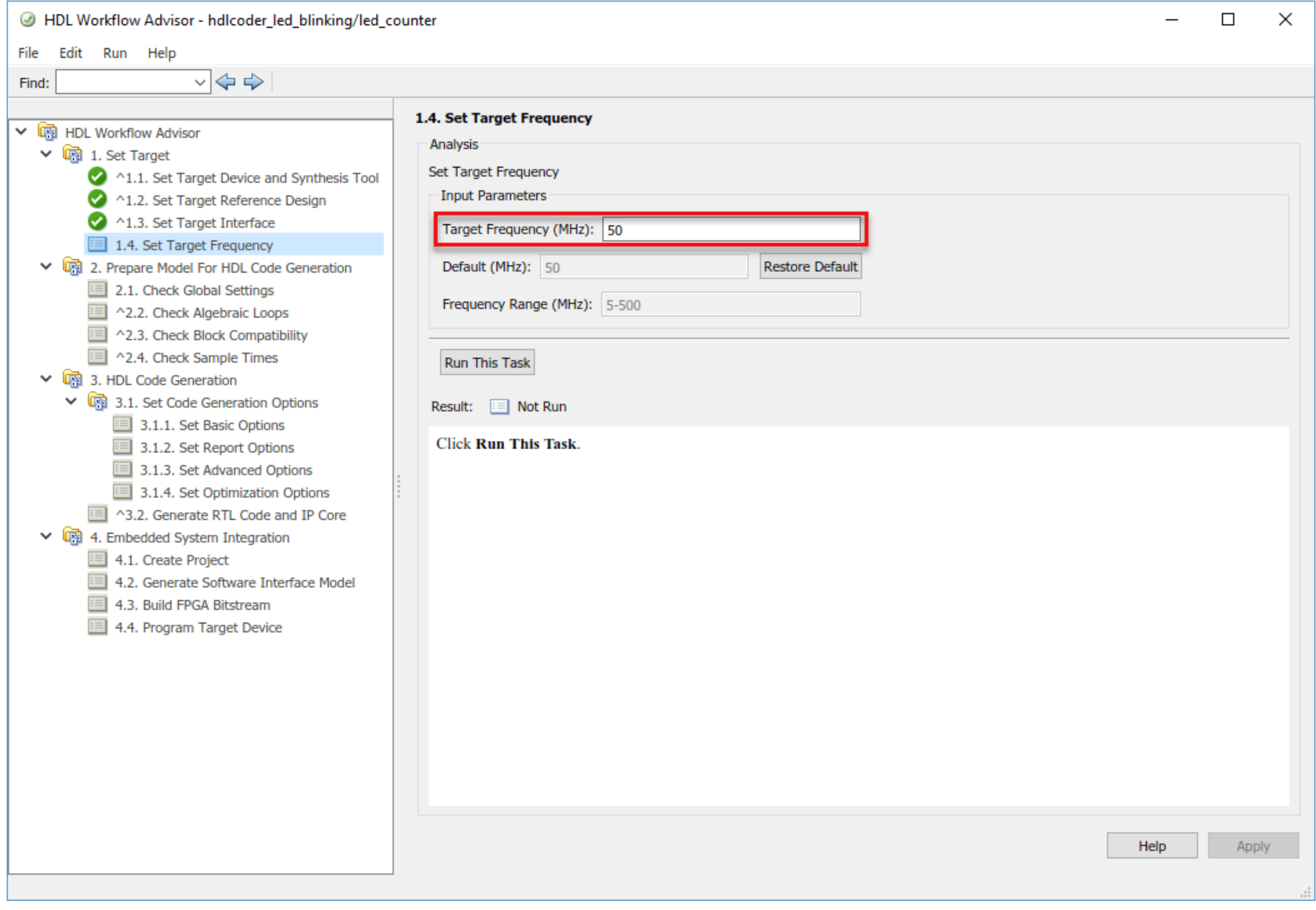
Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4-Lite interface, so HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:7]**, which connects to the LED hardware on the Zynq board.

2.1 In the **Set Target > Set Target Interface** task, choose **AXI4-Lite** for **Blink_frequency**, **Blink_direction**, and **Read_back**.

2.2 Choose **LEDs General Purpose [0:7]** for **LED**.

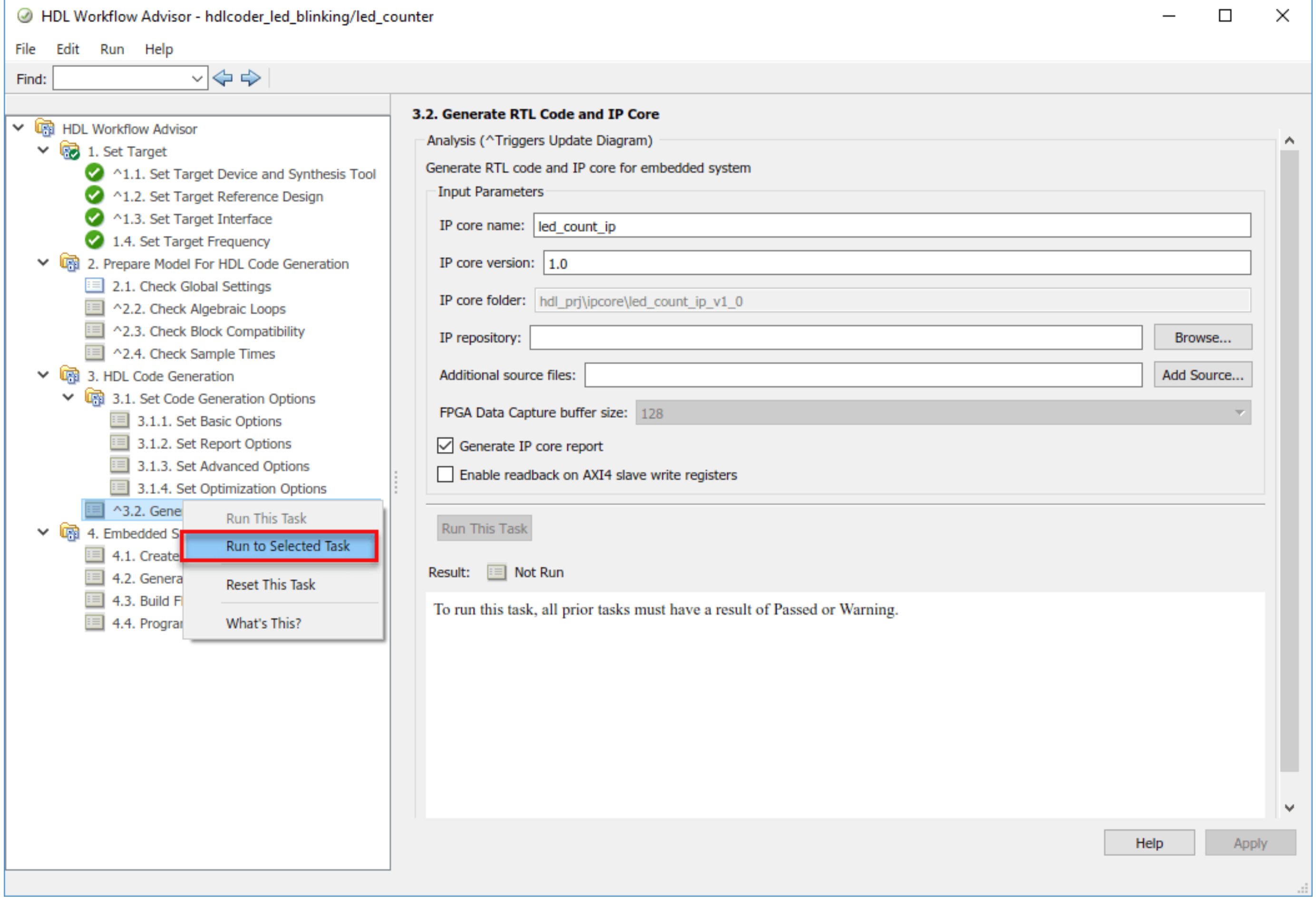


2.3 In the **Set Target > Set Target Frequency** task, choose **Target Frequency as 50 MHz**.



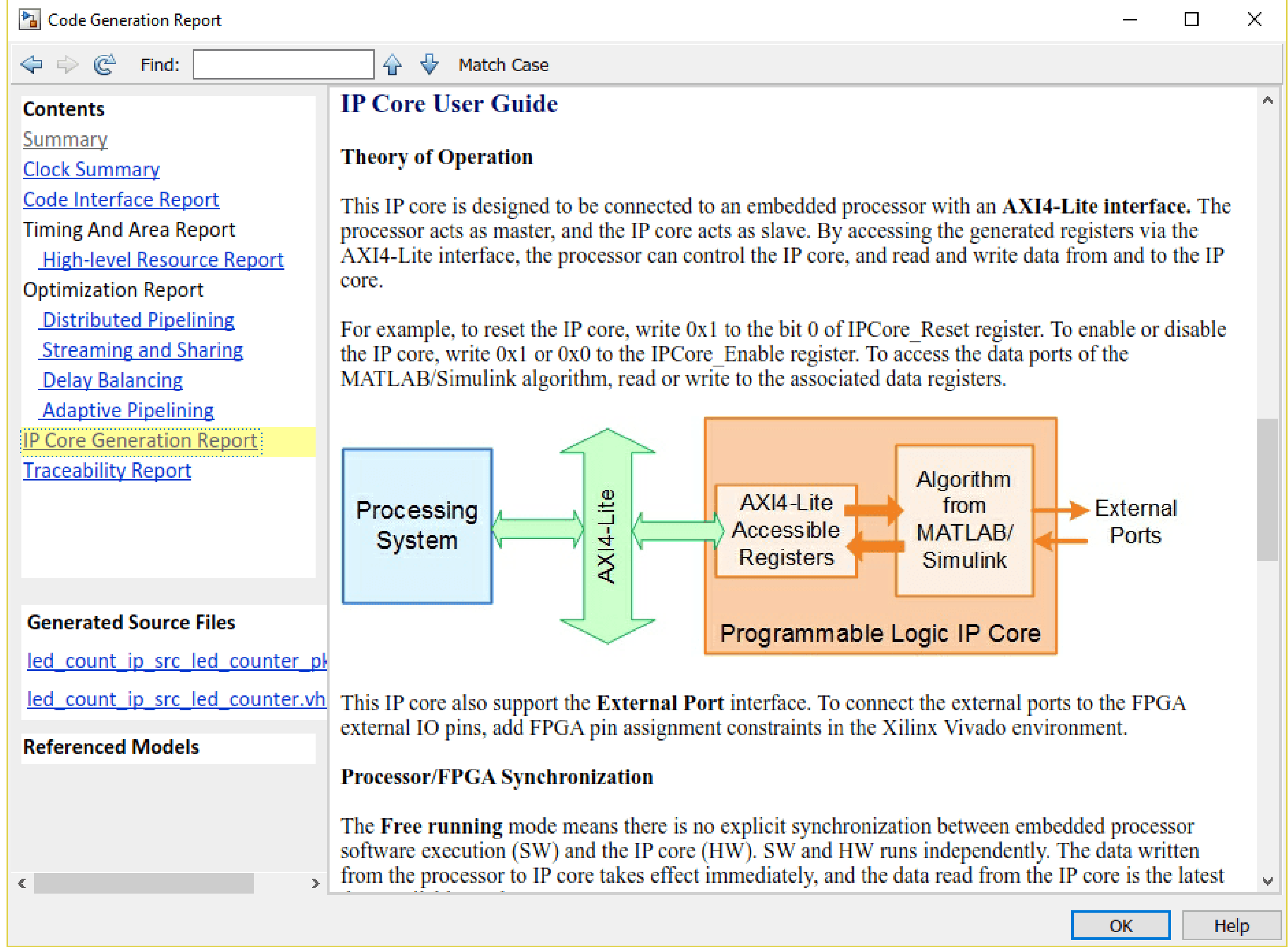
3. Generate the IP Core.

To generate the IP core, right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.



4. Generate and view the IP core report.

After you generate the custom IP core, the IP core files are in the **ipcore** folder within your project folder. An HTML custom IP core report is generated together with the custom IP core. The report describes the behavior and contents of the generated custom IP core.

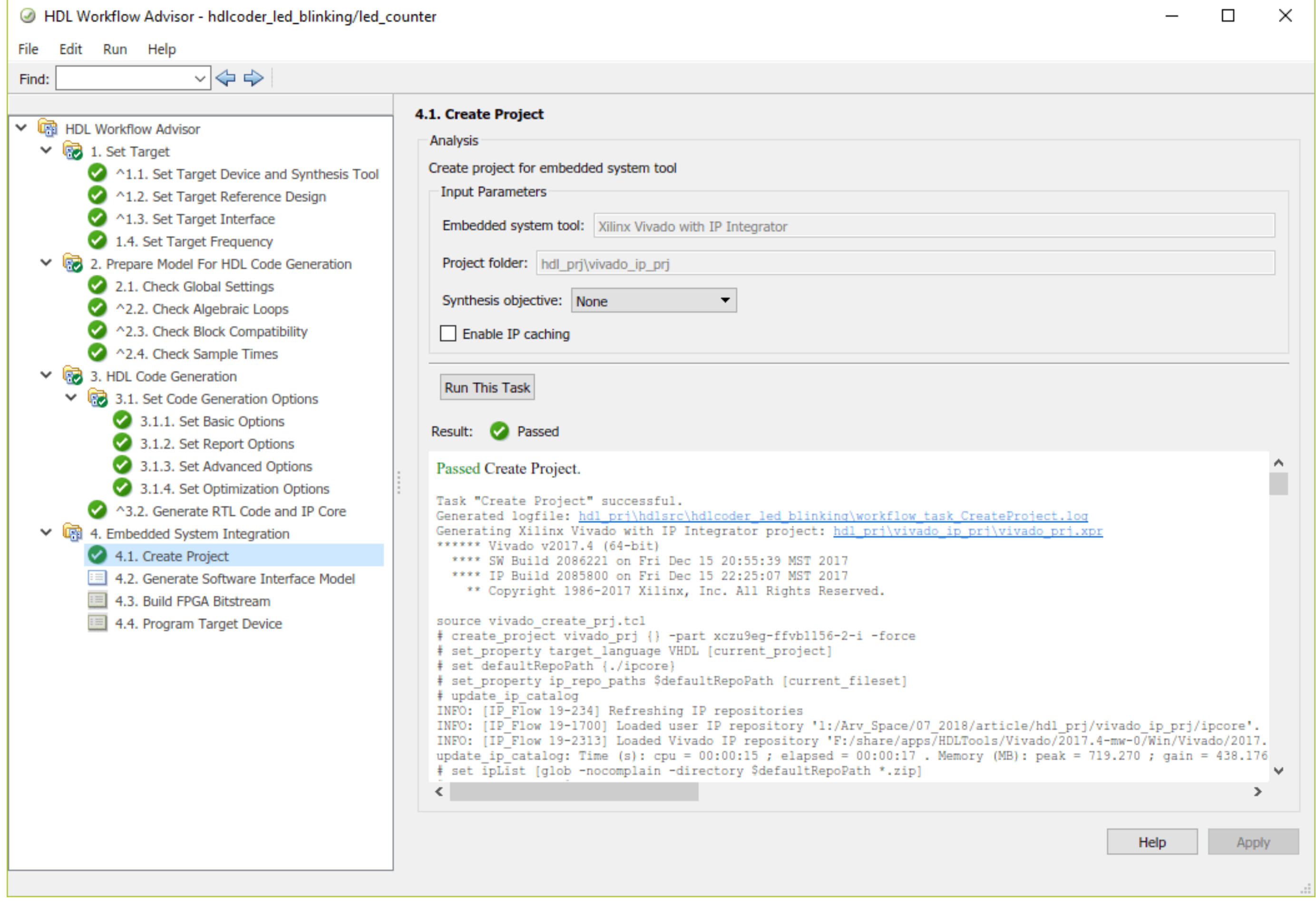


Integrate the IP Core with the Xilinx Vivado Environment

In this part of the workflow, you insert your generated IP core into a embedded system reference design, generate an FPGA bitstream, and download the bitstream to the Zynq hardware.

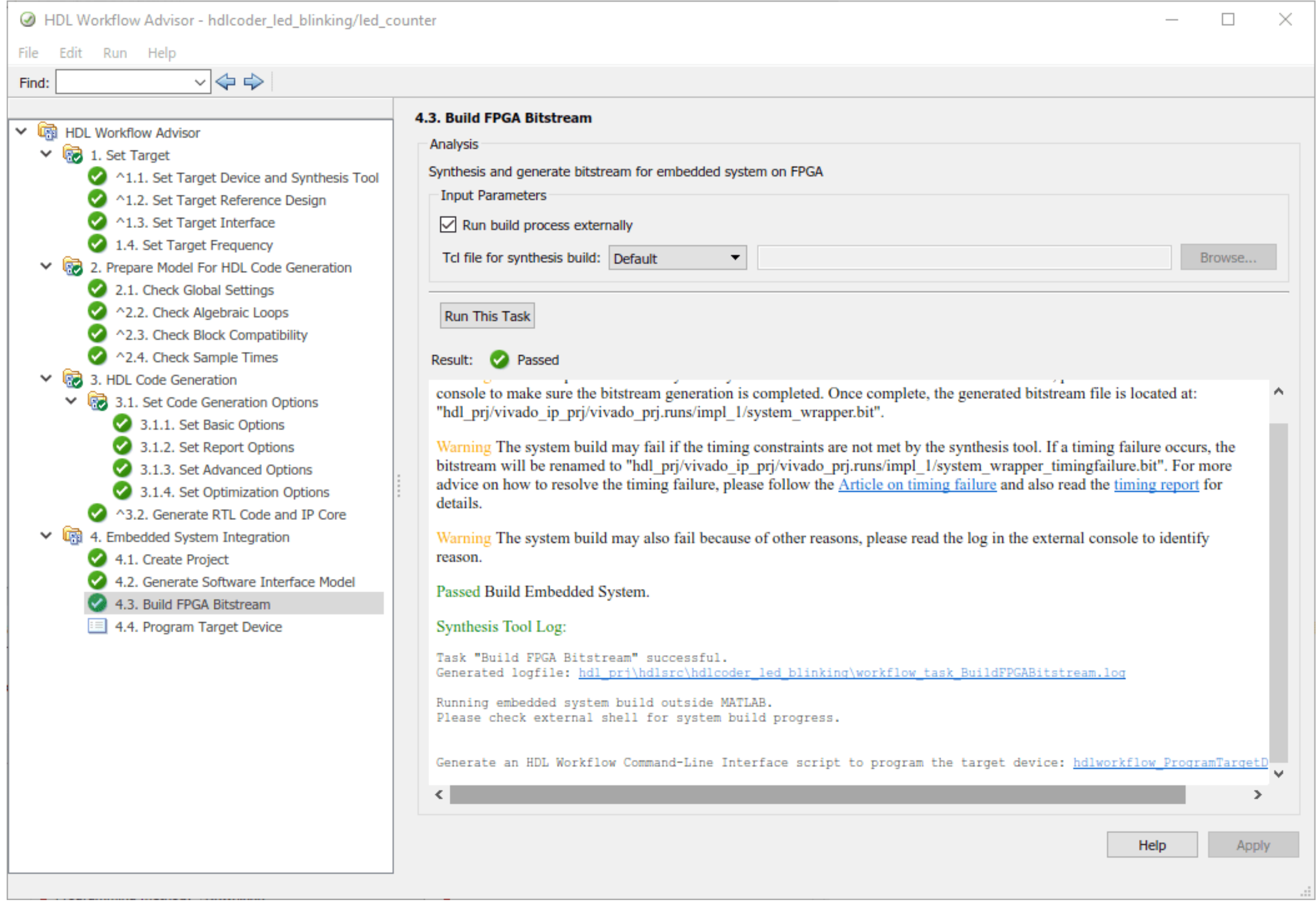
The reference design is a predefined Xilinx Vivado project. It contains all the elements the Xilinx software needs to deploy your design to the Zynq platform, except for the custom IP core and embedded software that you generate.

1. To integrate with the Xilinx Vivado environment, select the **Create Project** task under **Embedded System Integration**, and click **Run This Task**. A Xilinx Vivado project with IP Integrator embedded design is generated, and a link to the project is provided in the dialog window. You can optionally open up the project to take a look.

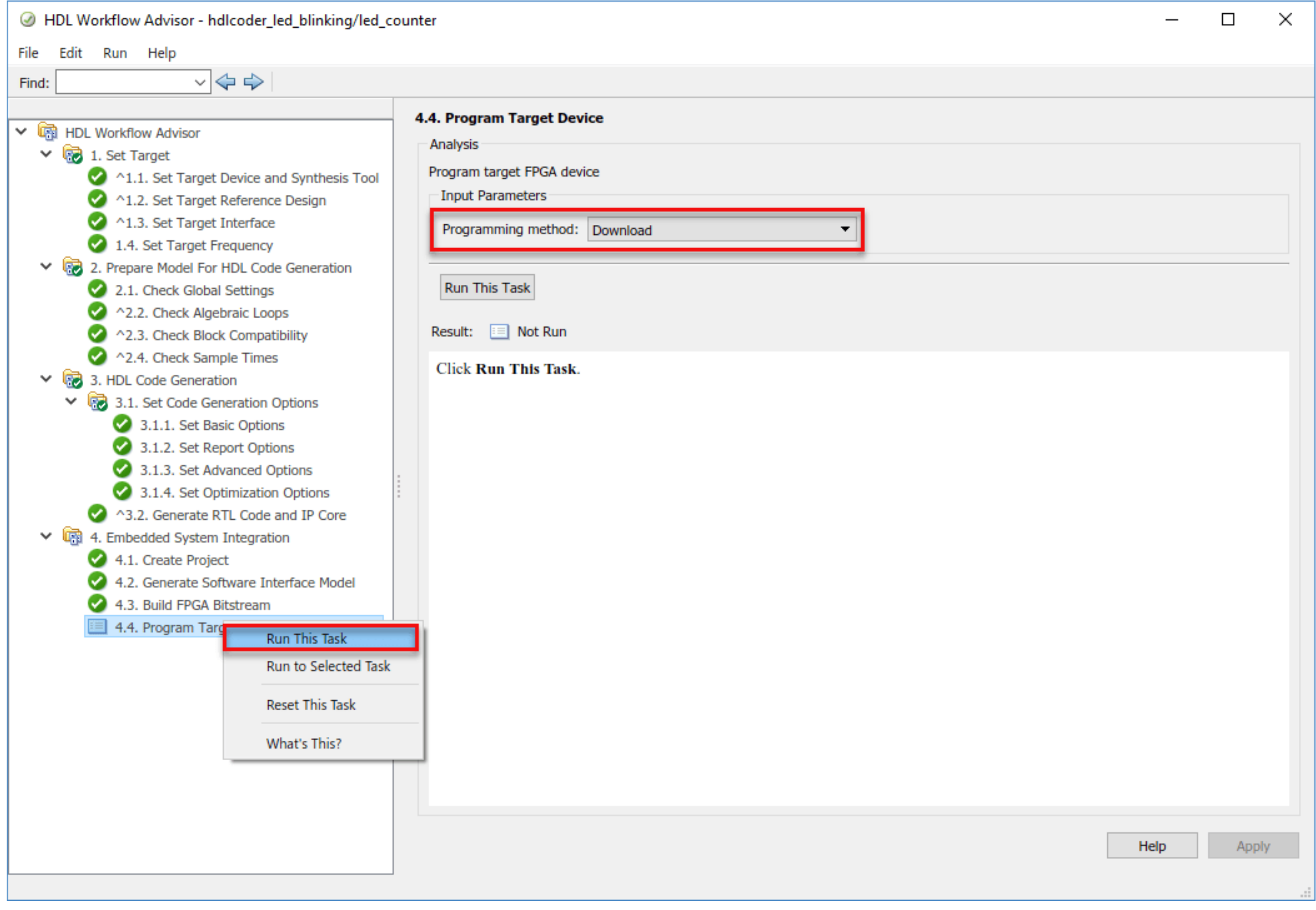


2. If you have an Embedded Coder license, you can generate a software interface model in the next task, **Generate Software Interface Model**. The details of the software interface model are explained in the next section of this example, "Generate a software interface model".

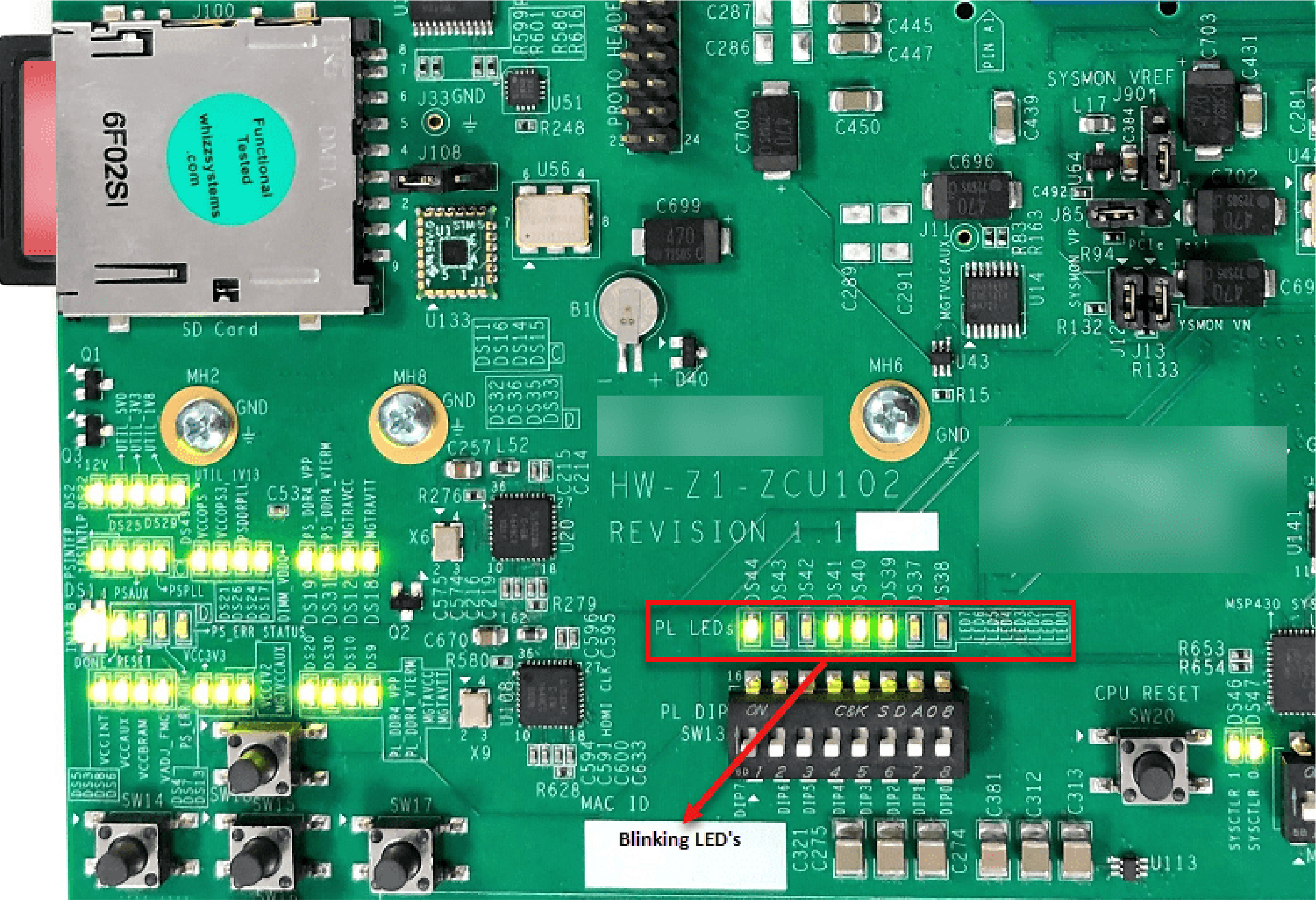
3. Build the FPGA bitstream in the **Build FPGA Bitstream** task. Make sure the **Run build process externally** option is checked, so the Xilinx synthesis tool will run in a separate process from MATLAB. Wait for the synthesis tool process to finish running in the external command window.



4. After the bitstream is generated, select the **Program Target Device** task. Choose **Download** for **Programming method** to download the FPGA bitstream onto the SD card on the Xilinx Zynq UltraScale+ MPSoC board, so your design will be automatically reloaded when you power cycle the Zynq board. click **Run This Task** to program the Zynq hardware.



After you program the FPGA hardware, the LED starts blinking on your Xilinx Zynq UltraScale+ MPSoC ZCU102 board.



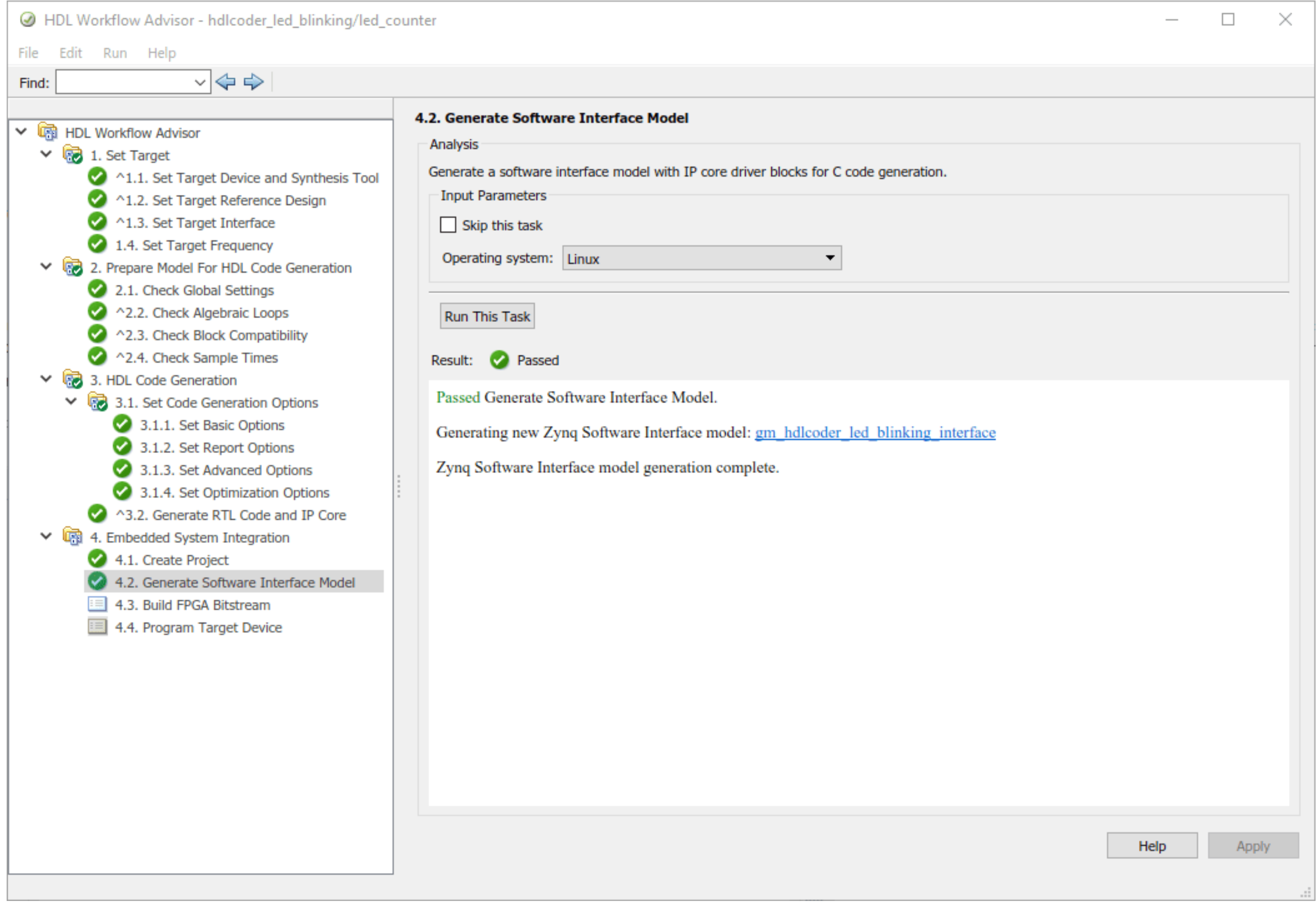
Next, you will generate C code to run on the ARM processor to control the LED blink frequency and direction.

Generate a Software Interface Model

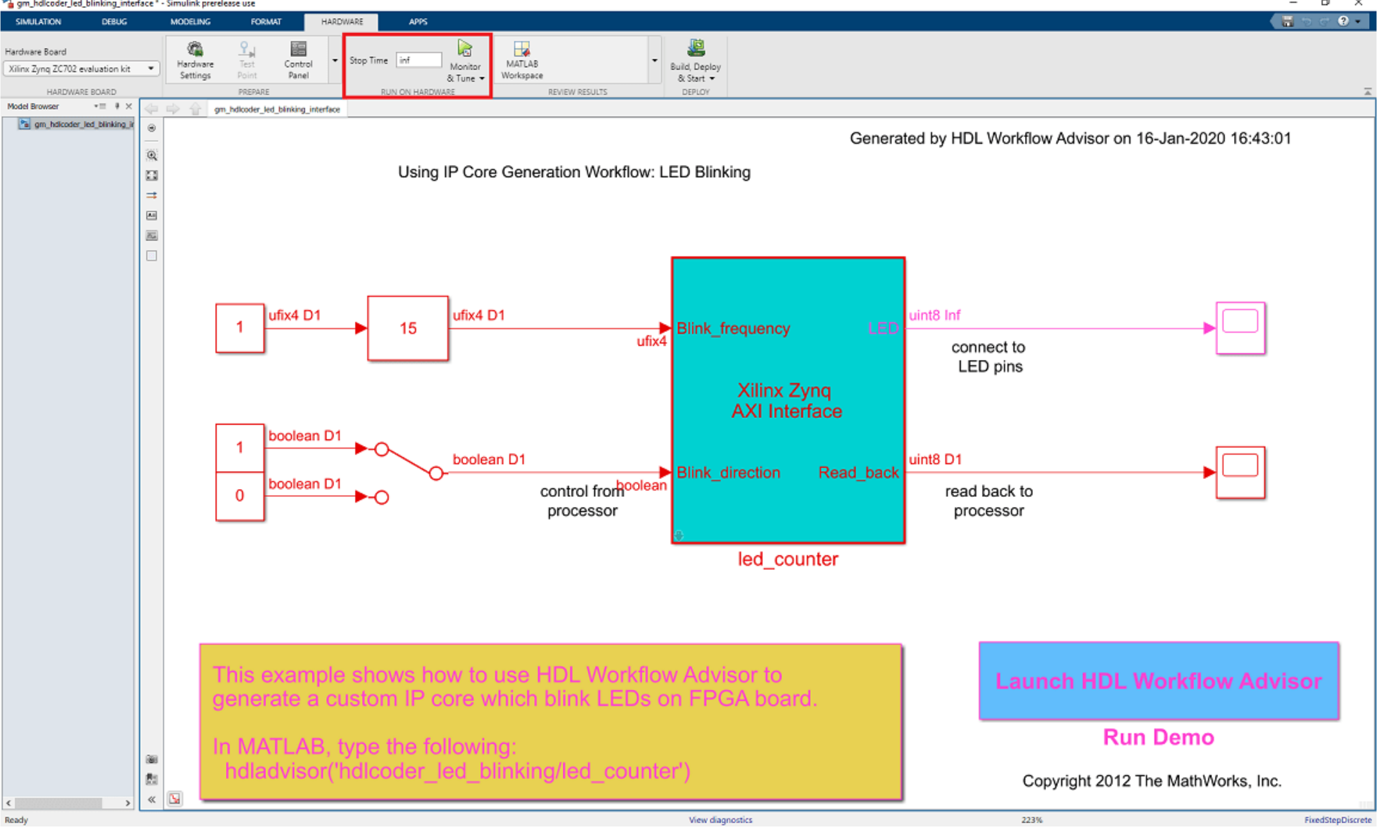
In the HDL Workflow Advisor, after you generate the IP core, you can create a vivado project in step 4.1, you can optionally generate a software interface model in the **Embedded System Integration** > **Generate Software Interface Model** task.

The software interface model contains the part of your design that runs in software. It includes all the blocks outside of the HDL subsystem, and replaces the HDL subsystem with AXI driver blocks. If you have an Embedded Coder license, you can automatically generate embedded C code from the software interface model, build it, and run the executable on Linux on the ARM processor. The generated embedded software includes AXI driver code, generated from the AXI driver blocks, that controls the HDL IP core.

Run the **Generate Software Interface Model** task and see that a new model is generated. The task dialog shows a link to the model.



In the generated software interface model, the `led_counter` subsystem is replaced with the AXI driver blocks which generates the interface logic between the ARM processor and FPGA.

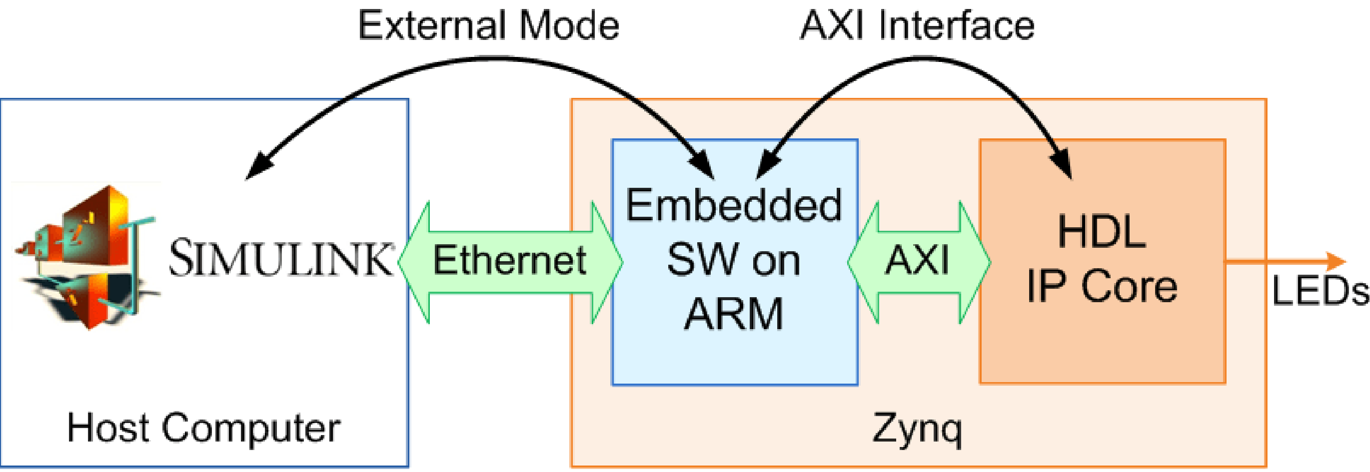


Run the Software Interface Model on Zynq ZCU102 Hardware

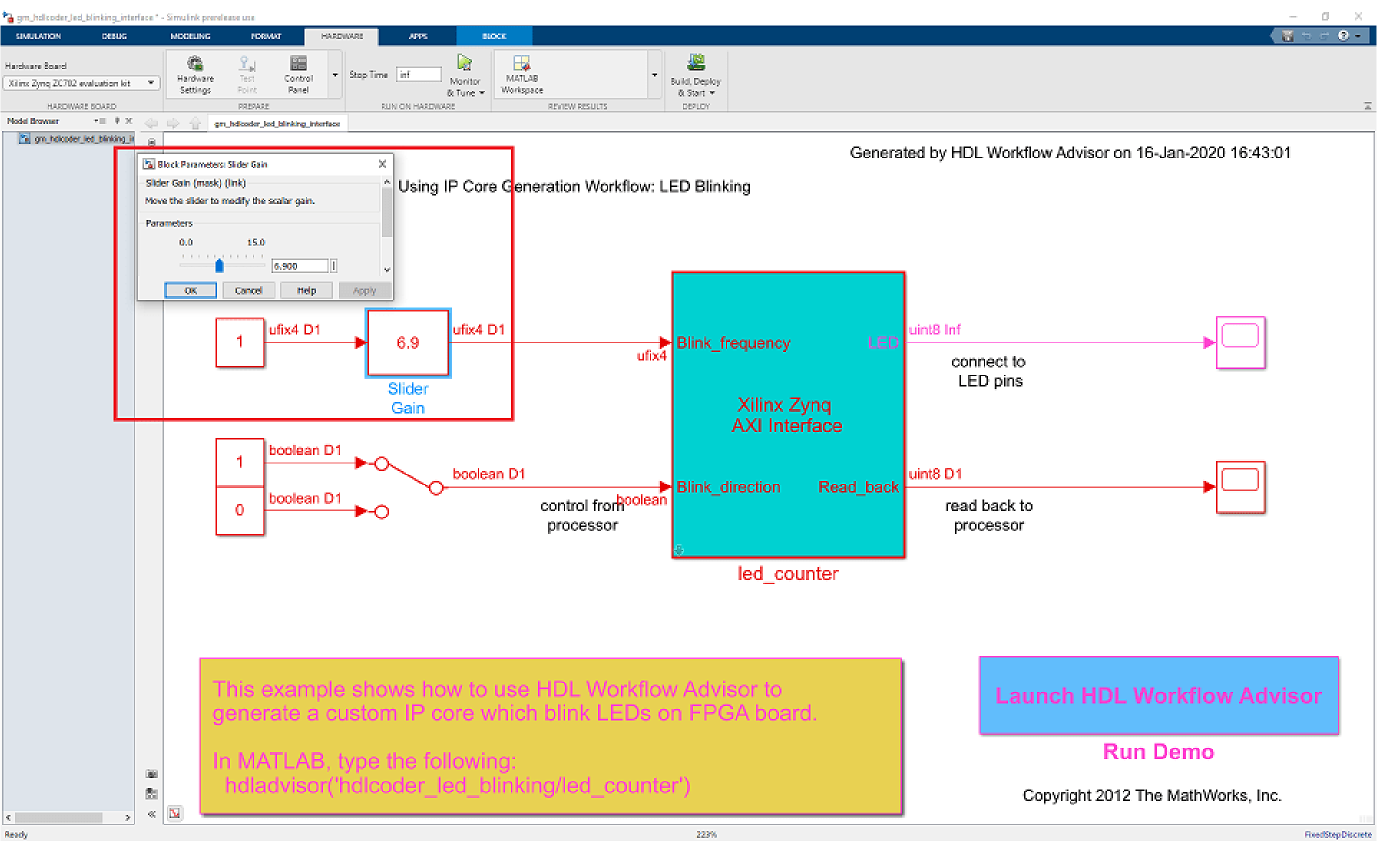
In this part of the workflow, you configure the generated software interface model, automatically generate embedded C code, and run your model on the ARM processor in the Zynq hardware in External mode.

When you are prototyping and developing an algorithm, it is useful to monitor and tune the algorithm while it runs on hardware. The External mode feature in Simulink enables this capability. In this mode, your algorithm is first deployed to the ARM processor in the Zynq hardware, and then linked with the Simulink model on the host computer through an Ethernet connection.

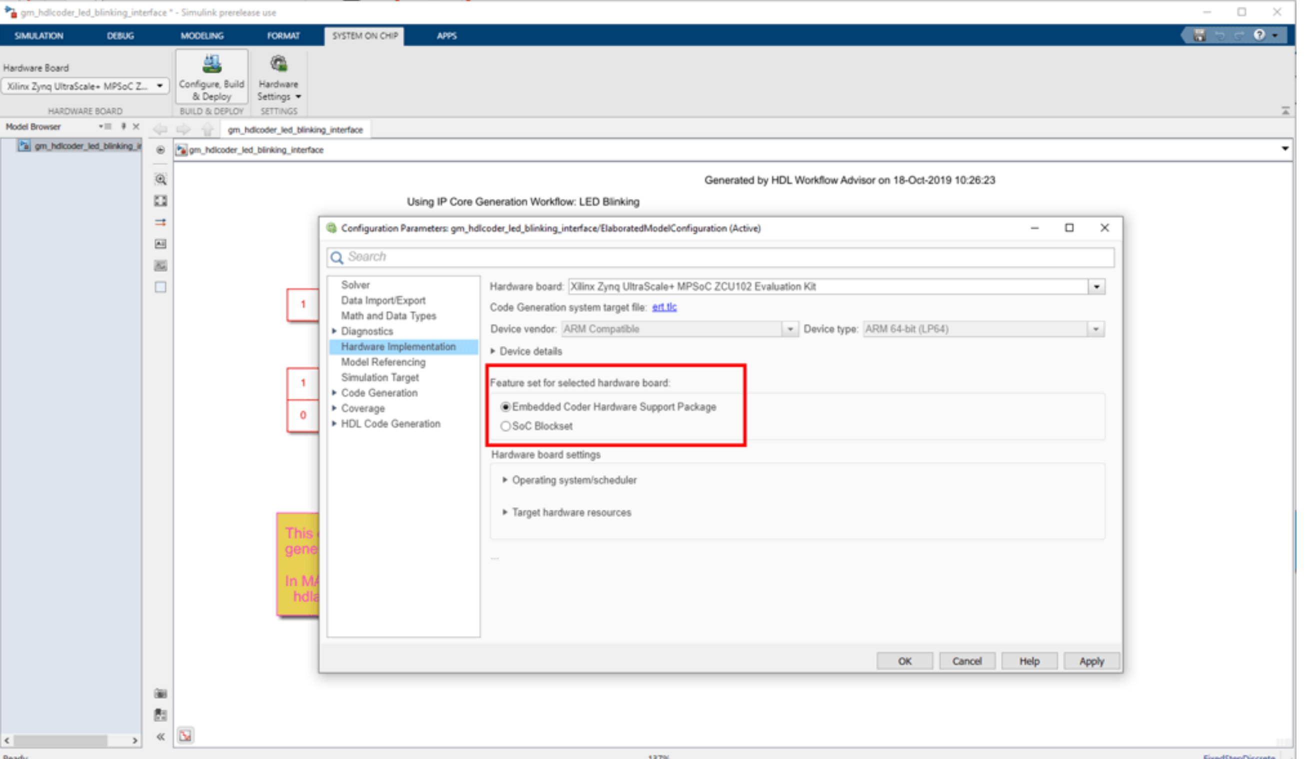
The main role of the Simulink model is to tune and monitor the algorithm running on the hardware. Because the ARM processor is connected to the HDL IP core through the AXI interface, you can use External mode to tune parameters, and capture data from the FPGA.



1. In the generated model,click on Hardware pane and go to **Hardware settings** to open **Configuration Parameter** dialog box.
2. Select **Solver** and set **Stop Time** to **inf**.
3. Select **Hardware Implementation** and set **Feature set for selected hardware board** to **Embedded Coder Hardware Support Package**.
4. From the **HARDWARE** menu, click the **Monitor & Tune** button on the model toolstrip to run your model on the ARM processor in the Zynq UltraScale+ MPSoC ZCU102 hardware in External mode. Embedded Coder builds the model, downloads the ARM executable to the Xilinx Zynq UltraScale+ MPSoC ZCU102 hardware, executes it, and connects the model to the executable running on the Zynq hardware.
5. Double-click the **Slider Gain** block. Change the Slider Gain value and observe the change in frequency of the LED array blinking on the Zynq hardware. Double-click the **Manual Switch** block to switch the direction of the blinking LEDs.
6. Double-click the scope connected to the **Read_back** output port and observe that the output data of the FPGA IP core is captured and sent back to the Simulink scope.
7. When you are done changing model parameters, click the **Stop** button on the model.



Generated by HDL Workflow Advisor on 16-Jan-2020 16:43:01
Using IP Core Generation Workflow: LED Blinking
Block Parameters: Slider Gain
Slider Gain (mask) (link)
Move the slider to modify the scalar gain.
Parameters
0.0
15.0
6.900
OK
Cancel
Help
Apply
1
ufix4 D1
6.9
ufix4 D1
Slider Gain
ufix4
Blink_frequency
LED
uint8 Inf
connect to LED pins
Xilinx Zynq AXI Interface
1
boolean D1
0
boolean D1
boolean D1
control from processor
boolean
Blink_direction
Read_back
uint8 D1
read back to processor
led_counter
This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.
In MATLAB, type the following:
hdladvisor('hdlcoder_led_blinking/led_counter')
Launch HDL Workflow Advisor
Run Demo
Copyright 2012 The MathWorks, Inc.

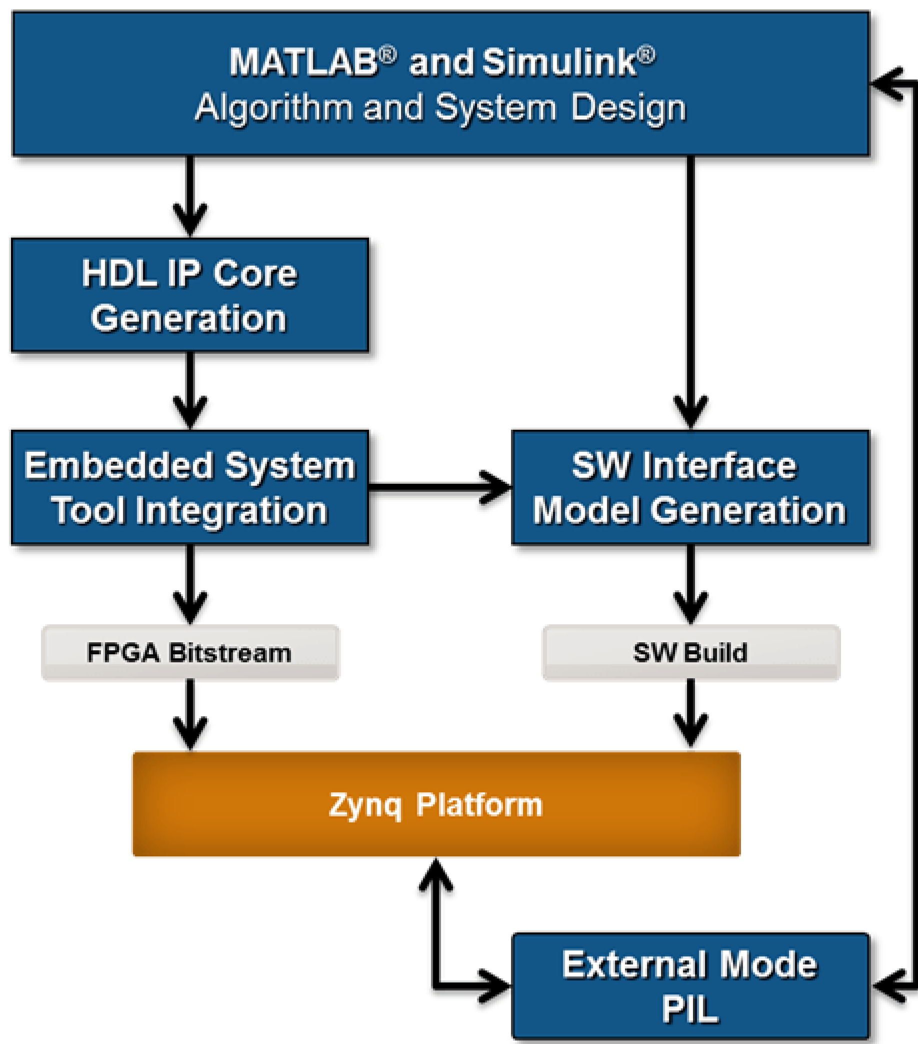


Generated by HDL Workflow Advisor on 18-Oct-2019 10:26:23
Using IP Core Generation Workflow: LED Blinking
Configuration Parameters: gm_hdlcoder_led_blinking_interface/ElaboratedModelConfiguration (Active)
Search
Solver
Data Import/Export
Math and Data Types
Diagnostics
Hardware Implementation
Model Referencing
Simulation Target
Code Generation
Coverage
HDL Code Generation
Hardware board: Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
Code Generation system target file: ert.tlc
Device vendor: ARM Compatible
Device type: ARM 64-bit (LP64)
Device details
Feature set for selected hardware board:
Embedded Coder Hardware Support Package
SoC Blockset
Hardware board settings
Operating system/scheduler
Target hardware resources
OK
Cancel
Help
Apply

Summary

This example shows how the hardware-software co-design workflow helps automate the deployment of your MATLAB and Simulink design to a Xilinx Zynq Ultrascale+ MPSoC. You can explore the best ways to partition and deploy your design by iterating through the workflow.

The following diagram shows the high-level picture of the workflow you went through in this example. To learn more about the hardware and software co-design workflow, see “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9.



Getting Started with Targeting Intel SoC Devices

This example shows how to use the Simulink Toolstrip and Simulink Interface along with the hardware-software co-design workflow to blink LEDs at various frequencies on the Arrow® SoCKit® evaluation kit.

Introduction

This example is a step-by-step guide that helps you use the HDL Coder™ software to generate a custom HDL IP core which blinks LEDs on the Arrow SoCKit evaluation kit, and shows how to use Embedded Coder® to generate C code that runs on the ARM® processor to control the LED blink frequency.

You can use MATLAB® and Simulink® to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your design for hardware and software implementation on the Altera Cyclone V SoC by deciding which system elements will be performed by the programmable logic, and which system elements will run on the ARM Cortex-A9.

Using the guided workflow shown in this example, you automatically generate HDL code for the programmable logic using HDL Coder, generate C code for the ARM using Embedded Coder, and implement the design on the Intel SoC devices.

In this workflow, you perform the following steps:

1. Set up your Intel SoC hardware and tools.
2. Partition your design for hardware and software implementation.
3. Generate an HDL IP core using HDL Workflow Advisor.
4. Integrate the IP core into a Intel Qsys project and program the Intel SoC hardware.
5. Generate a software interface model.
6. Generate C code from the software interface model and run it on the ARM Cortex-A9 processor.
7. Tune parameters and capture results from the Intel SoC hardware using External Mode.

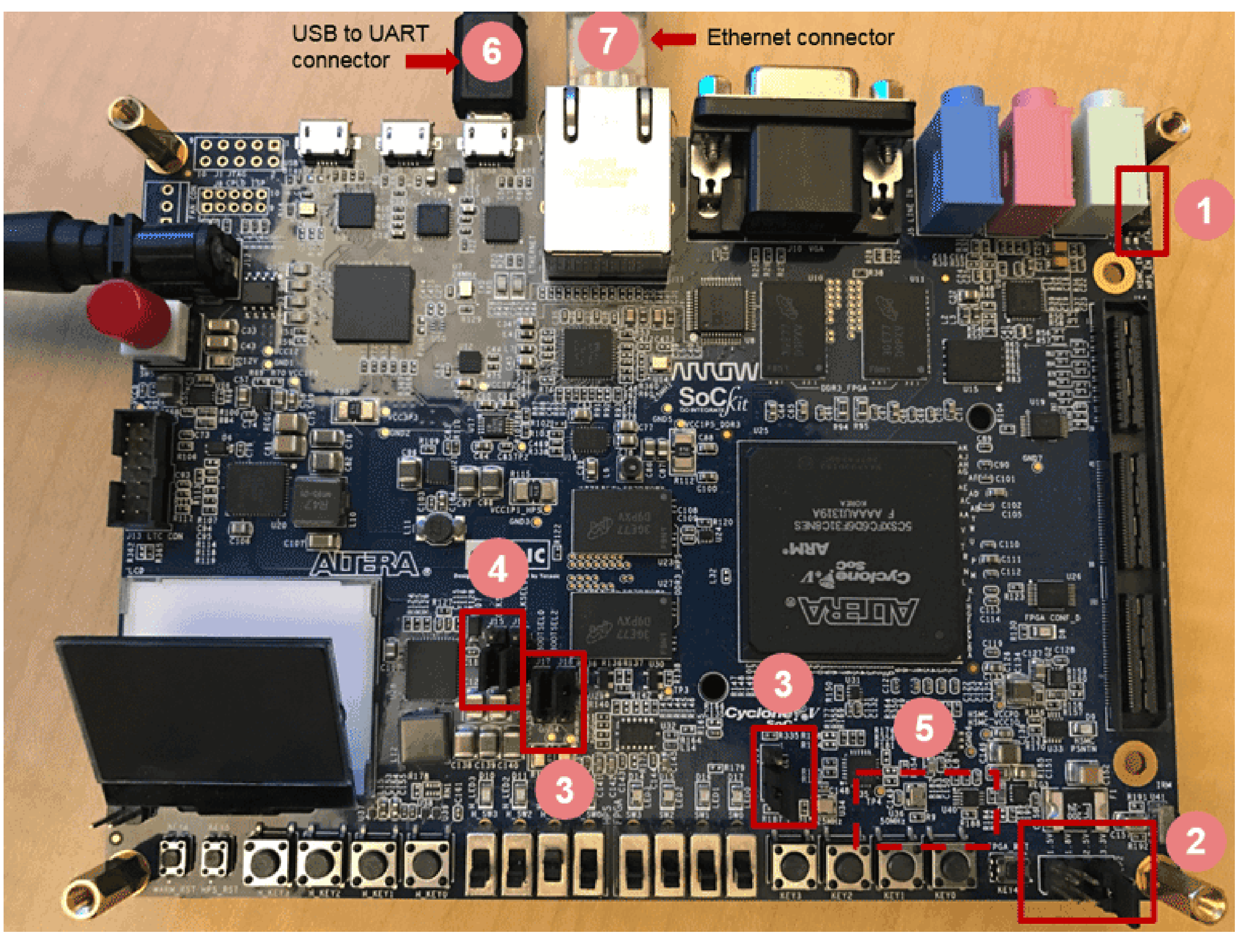
For more information, refer to other more advanced examples, and the HDL Coder and Embedded Coder documentation.

Requirements

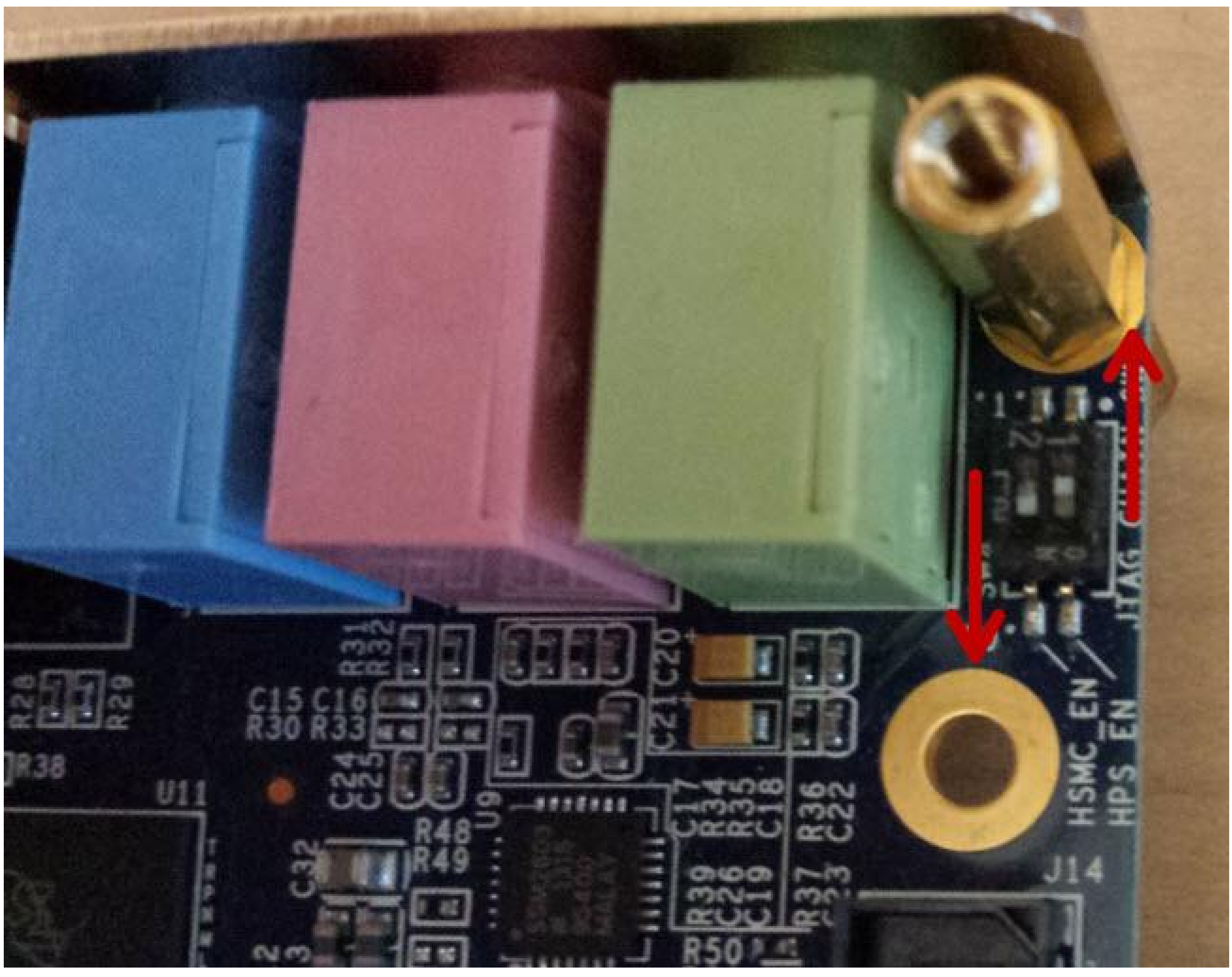
1. Intel Quartus Prime, with supported version listed in the “HDL Language Support and Supported Third-Party Tools and Hardware”.
2. Intel SoC Embedded Design Suite
3. Arrow SoCKit Cyclone V SoC evaluation kit
4. HDL Coder Support Package for Intel FPGA and SoC Devices
5. Embedded Coder Support Package for Intel SoC Devices

Set Up Intel SoC Hardware and Tools

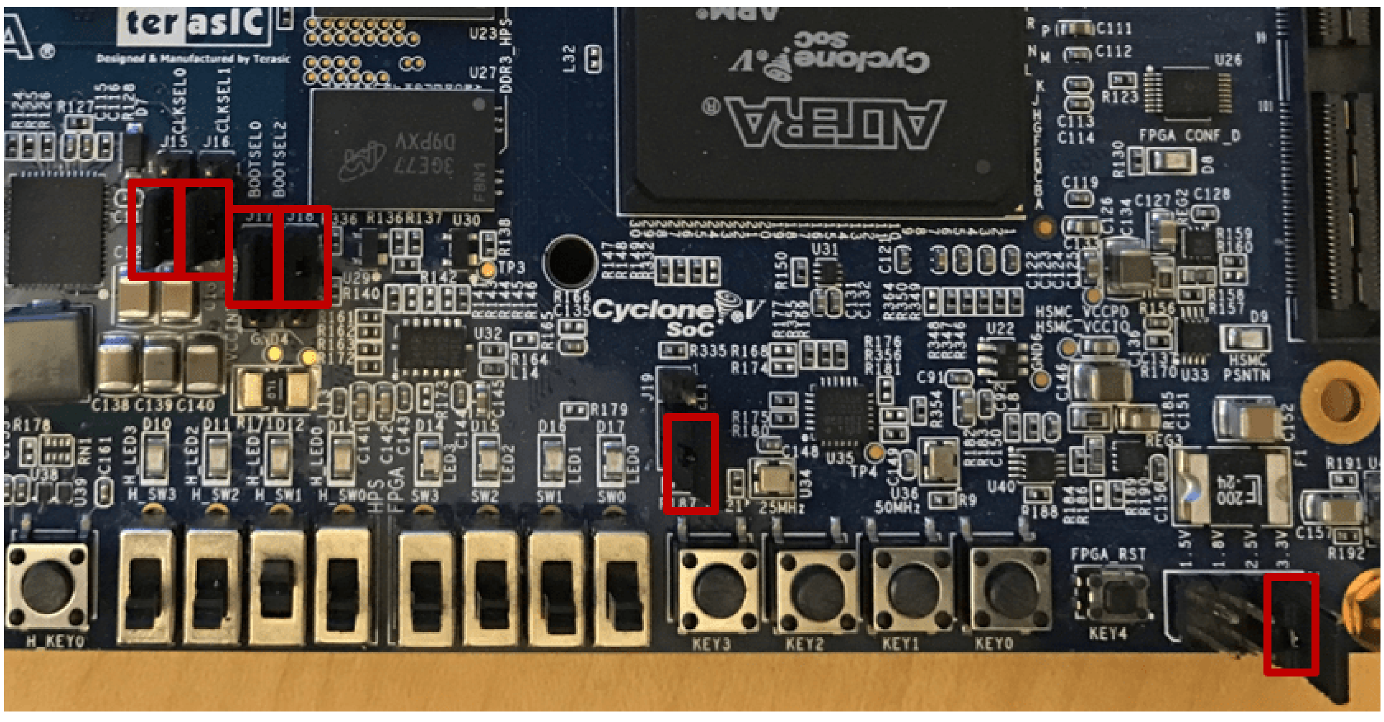
1. Set up the Arrow SoCKit evaluation kit as shown in the figure below. To learn more about the Arrow SoCKit hardware setup, please refer to the board documentation.



2. Set up SW4 switch (JTAG chain select) as shown in the figure below. Position 1 : OFF; Position 2 : ON. This configuration includes HPS in JTAG chain, and bypasses HSMC.



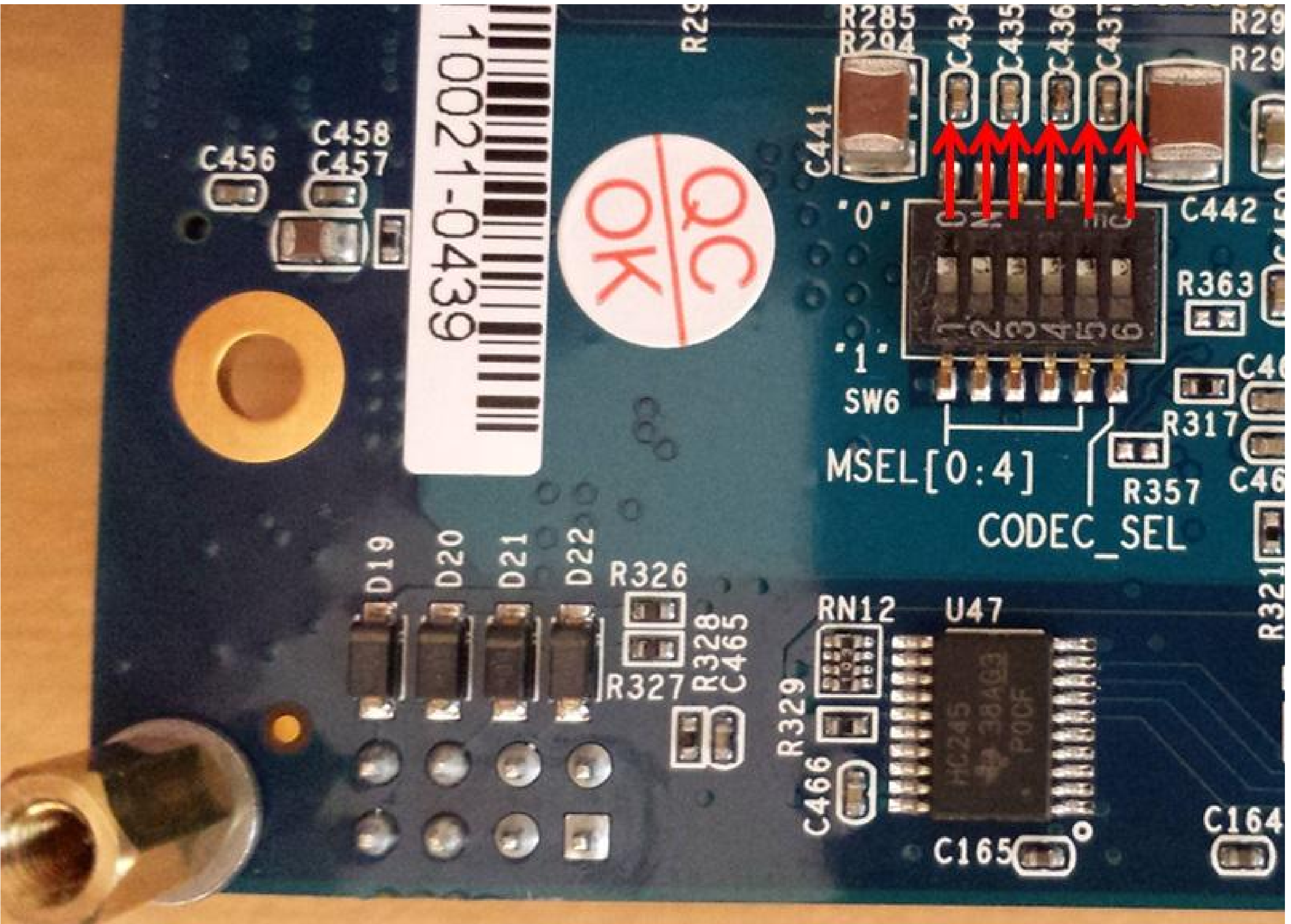
3. Set up JP2 as shown in the figure below to adjust the I/O Standard of the FPGA/HSMC pins. Short Pin 5 and 6 to set the I/O voltage to 2.5V.



4. Set up J17 - J19 as shown in the figure above to boot HPS from SD card. J17: Short Pin 1 and 2; J18: Short Pin 1 and 2; J19: Short Pin 2 and 3.

5. Set up J15 - J16 as shown in the figure above for HPS clock setting. J15: Short Pin 2 and 3; J16: Short Pin 2 and 3.

6. Set up SW6 on the back side of the board as shown in the figure below. This switch set the FPGA configuration mode. Set all 6 positions to ON.



7. Connect your computer to the USB UART connector using a Micro-USB cable. Make sure your USB device drivers, such as for the FTDI USB to UART, are installed correctly. If not, search for the drivers online and install them.

8. Connect your computer and the Arrow SoCKit board using an Ethernet cable.

9. Install the HDL Coder Support Package for Intel FPGA and SoC Devices and Embedded Coder Support Package for Intel SoC Devices if you haven't already. To start the installer, go to the MATLAB toolstrip and click **Add-Ons > Get Hardware Support Packages**.

10. Make sure you are using the SD card image provided by the Embedded Coder Support Package for Intel SoC Devices. If you need to update your SD card image, refer to the “Add Support for Intel SoC Platform” (Embedded Coder) of this document.

11. Set up the Arrow SoCKit hardware connection by entering the following command in the MATLAB command window:

```
h = alterasoc
```

The `alterasoc` function logs in to the hardware via COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

12. You can optionally test the serial connection using the following configuration using a program such as PuTTY™. Baud rate: `115200`; Data bits: `8`; Stop bits: `1`; Parity: `None`; Flow control: `None`. You should be able to observe Linux booting log on the serial console when you power cycle the Arrow SoCKit board. You must close this serial connection before using the `alterasoc` function again.

13. Set up the Intel Quartus synthesis tool path using the following command in the MATLAB command window. Use your own Quartus installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', 'ToolPath', 'C:\intelFPGA\18.0\quartus\bin64\qu
```

Partition Design for Hardware and Software Implementation

The first step of the Intel SoC hardware-software co-design workflow is to decide which parts of your design to implement on the programmable logic, and which parts to run on the ARM processor.

Group all the blocks you want to implement on programmable logic into an atomic subsystem. This atomic subsystem is the boundary of your hardware-software partition. All the blocks inside this subsystem will be implemented on programmable logic, and all the blocks outside this subsystem will run on the ARM processor.

In this example, the subsystem **led_counter** is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem **led_counter** are for software implementation.

In Simulink, you can use the **Slider Gain** or **Manual Switch** block to adjust the input values of the hardware subsystem. In the embedded software, this means the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. The output port, **Read_Back**, can be used to read data back to the processor.

```
open_system('hdlcoder_led_blinking_4bit');
```

Generate HDL IP Core

Since R2023b

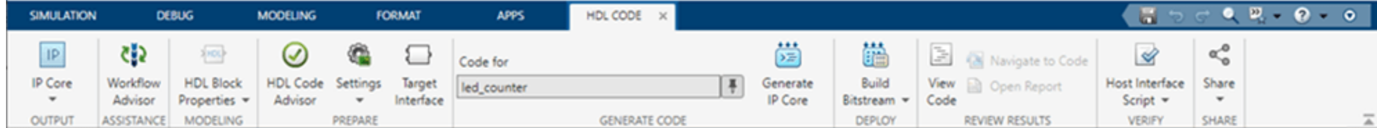
Next, configure your model for IP core generation, configure your design and target interface, and generate the IP core. This example uses the **HDL Code** tab in the Simulink Toolstrip to generate an IP core. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic that connects the IP core to the embedded processor. HDL Coder packages the generated files into a folder you specify. You can then integrate the generated IP core with a larger FPGA embedded design in the Intel Qsys environment.

For an overview of how to generate an IP core for a specific hardware platform, see “Targeting FPGA & SoC Hardware Overview” on page 39-3. For more information on different ways to generate an IP core using HDL Coder, see “Comparison of IP Core Generation Techniques” on page 39-27.

Prepare Model for IP Core Generation

To generate an IP core from the `hdlcoder_led_blinking_4bit/led_counter` subsystem, prepare your model by using the configuration parameters, configure your design by using the IP Core editor, and generate the IP core by using the **HDL Code** tab of the Simulink Toolstrip.

1. In the **Apps** tab, click **HDL Coder**. In the **HDL Code** tab, in the **Output** section, set the drop-down button to **IP Core**.
2. Select the `led_counter` subsystem which is the device under test (DUT) for this example. Make sure that **Code for** is set to this subsystem. To remember the selection, you can click the pin button .



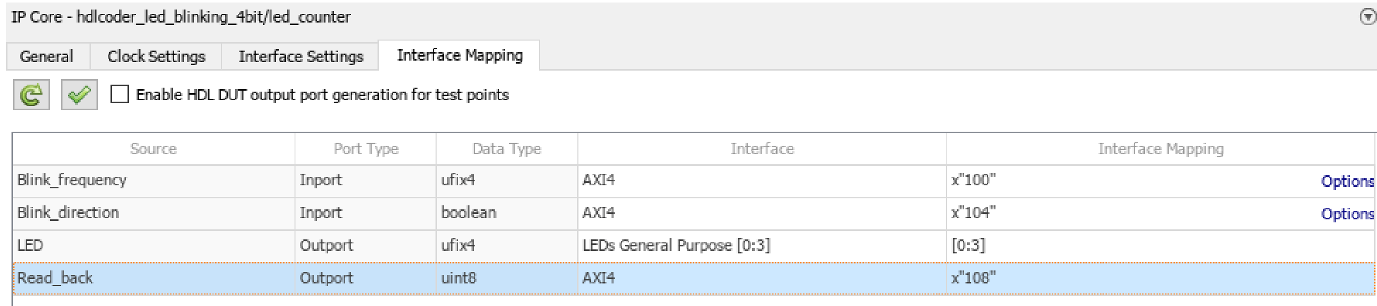
3. Click **Setting** to open the **HDL Code Generation > Target** pane of the Configuration Parameters dialog box.
4. Set the **Target Platform** parameter to `Arrow SoCKit development board`. If this option does not appear, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Intel FPGA and SoC Devices and follow the instructions to complete the installation. Ensure the **Synthesis Tool** is set to `Altera QUARTUS II`.
5. Ensure that the **Reference Design** parameter is set to `Default system`.
6. Set the **Target Frequency** to `50` MHz.
7. Click **OK** to save your updated settings.

Configure Design and Target Interface

Configure your design to map to the target hardware by mapping the DUT ports to IP core target hardware and setting DUT-level IP core options. In this example, the input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4 interface, so HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, `LEDs General Purpose [0:3]`, which connects to the LED hardware on the Intel SoC board.

1. In Simulink, in the **HDL Code** tab, click **Target Interface** to open the IP Core editor.
2. Select the **Interface Mapping** tab to map each DUT port to one of the IP core target interfaces.

   If no mapping table appears, click the Reload IP core settings button to compile the model and repopulate the DUT ports and their data types.
3. For the DUT ports **Blink_frequency**, **Blink_direction**, and **Read_back**, set the cells in the **Interface** column to `AXI4`.
4. For the **LED** output port, set the cell in the **Interface** column to `LEDs General Purpose [0:3]`.



Source	Port Type	Data Type	Interface	Interface Mapping	
Blink_frequency	Inport	ufix4	AXI4	x"100"	Options
Blink_direction	Inport	boolean	AXI4	x"104"	Options
LED	Outport	ufix4	LEDs General Purpose [0:3]	[0:3]	
Read_back	Outport	uint8	AXI4	x"108"	

5 Validate your settings by clicking the Validate IP core settings button.

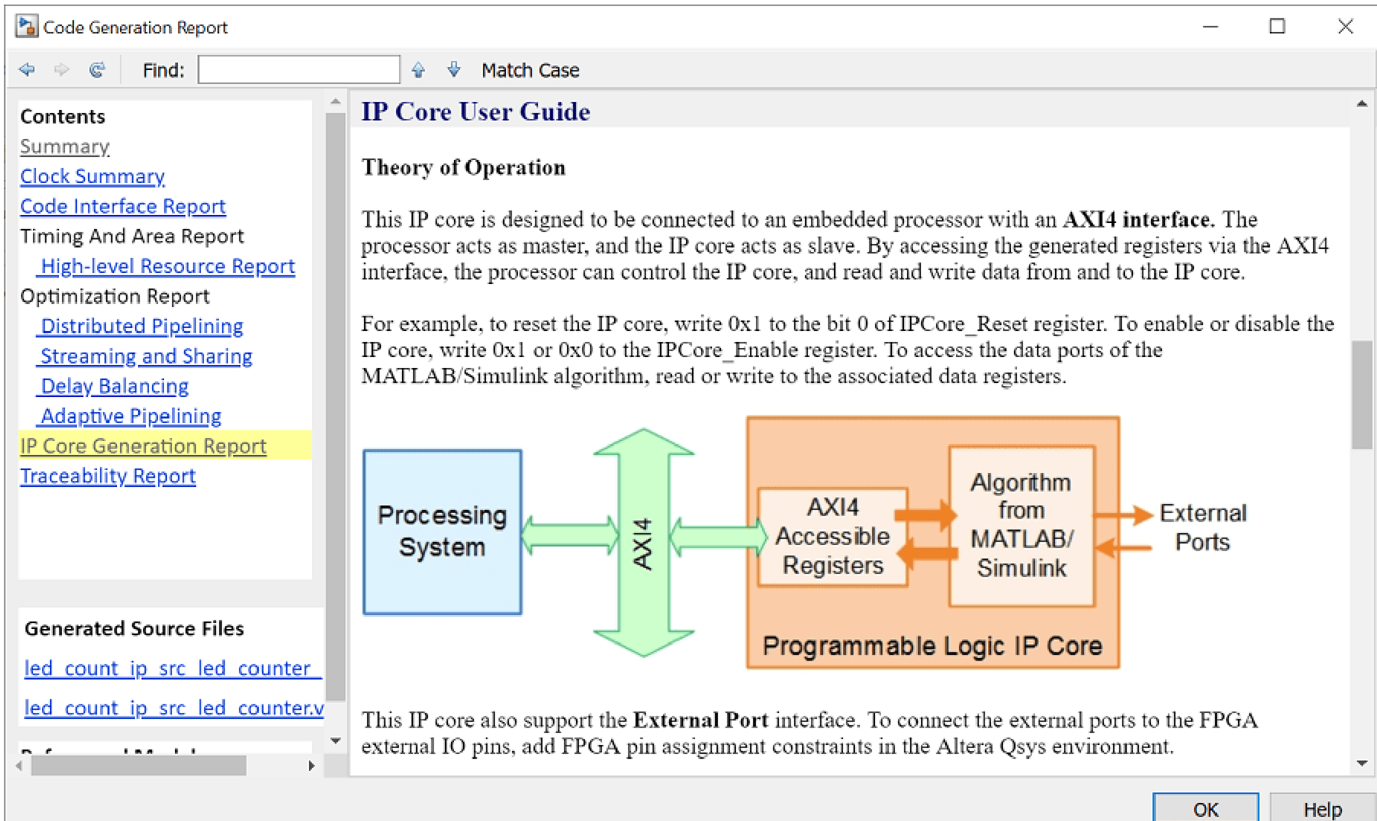
In the IP Core editor, you can optionally adjust the DUT-level IP core settings for your target hardware by:

- Using the **General** tab to configure top-level settings, such as the name of the IP core and whether to generate an IP core report.
- Using the **Clock Settings** tab to configure clock-related settings.
- Using the **Interface Settings** tab to configure interface-related settings, such as the register interface and FPGA data capture properties.

Generate IP Core

Next generate the IP core. In the Simulink Toolstrip, in the **HDL Code** tab, click **Generate IP Core**. After you generate the custom IP core, the IP core files are in the `ipcore` folder in your current directory. To specify a top-level project folder for the `ipcore` folder to be stored along with all other generated files, in the Configuration Parameters dialog box, use the **Project Folder** parameter in the **HDL Code Generation > Target** tab. If the **Project Folder** parameter is empty, HDL Coder saves the generated files in the current directory.

Generating an IP core also generates the code generation report. In the Code Generation Report window, in the left pane, click the **IP Core Generation Report**. The report describes the behavior and contents of the generated custom IP core.



Integrate IP Core with Intel Qsys

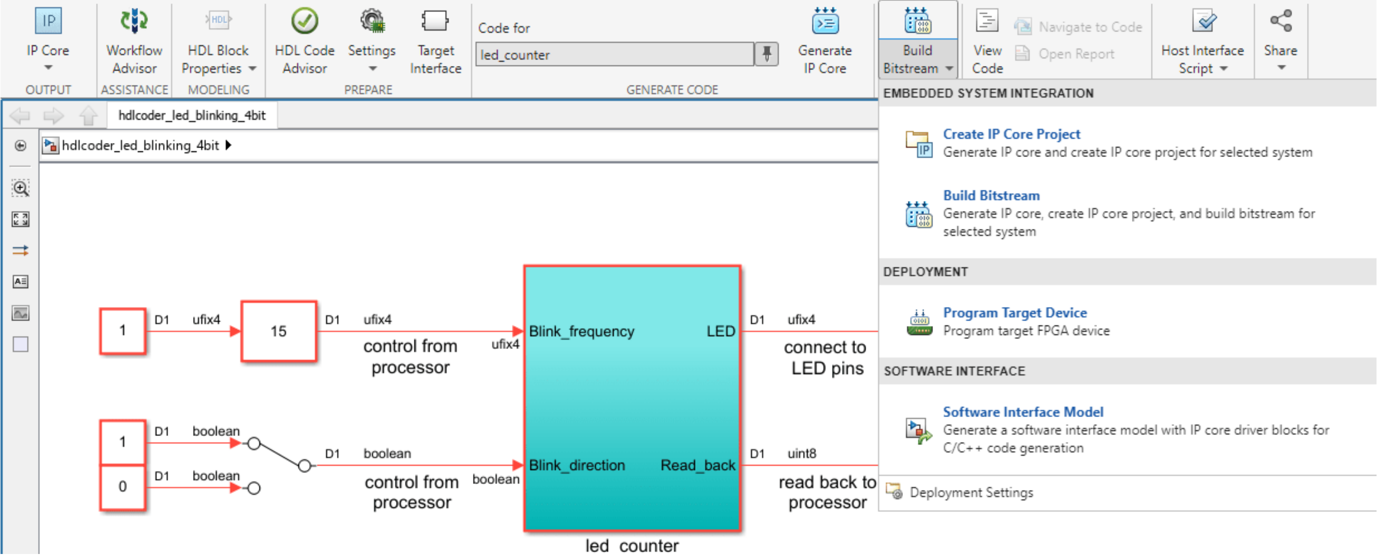
Since R2023b

Next, insert your generated IP core into an embedded system reference design by creating a project, generating an FPGA bitstream, and downloading the bitstream to the Intel SoC hardware.

The reference design is a predefined Intel Qsys project that contains the elements the Intel software needs to deploy your design to the Intel SoC device, except for the custom IP core and embedded software that you generate. This example uses the **HDL Code** tab in the Simulink Toolstrip to deploy and verify an IP core. For more information on how to deploy and verify an IP Core using the HDL workflow advisor, see “Comparison of IP Core Deployment and Verification Techniques” on page 39-30.

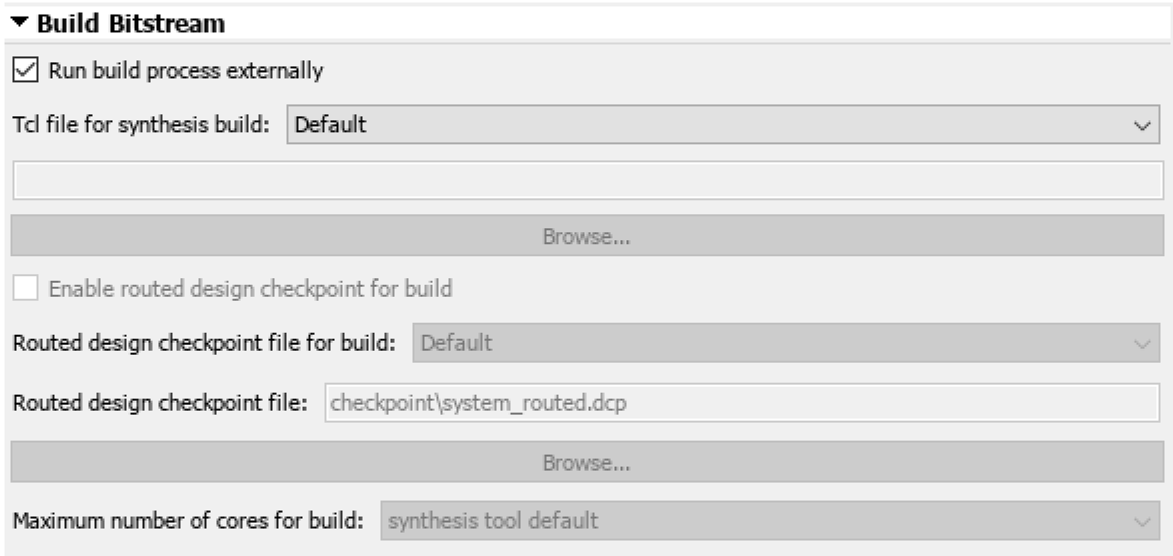
Create IP Core Project

Integrate your generated IP core into the Intel platform by creating a Qsys project that organizes and maintains the files associated with the IP core. To create a Qsys project in the Simulink Toolstrip, in the **HDL Code** tab**,** select **Build Bitstream > Create IP Core Project**. HDL Coder generates an IP integrator embedded design and displays the link to it in the Diagnostic Viewer.



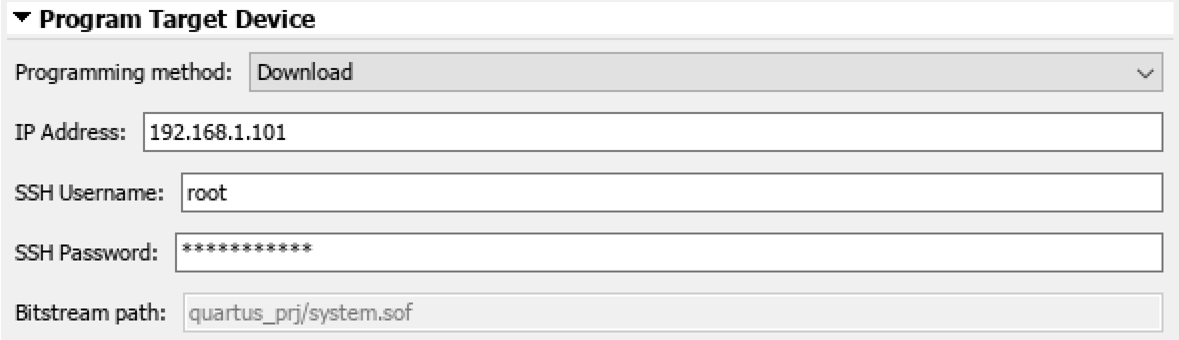
Configure Deployment Settings

Next, configure the build bitstream settings. In the Simulink Toolstri,p in the **HDL Code** tab**,** select **Build Bitstream > Deployment Settings**. In the Deployment Settings window, under the **Build Bitstream** section, select **Run build process externally** to run the Intel synthesis tool in a separate window than the current MATLAB session.



In the **Program Target Device** section:

- Set **Programming method to** `Download` to download the bitstream to your target Intel SoC board SD card and load your design when the board power cycles.
- Set **IP Address** to your target board IP address.
- Set **SSH username** and **SSH Password** to your target board settings.

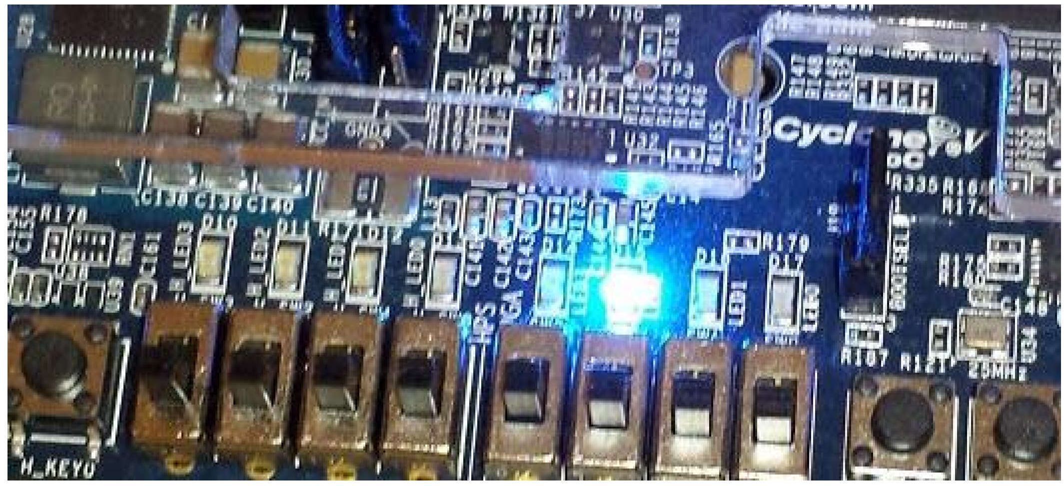


Generate Bitstream and Program Target Device

To generate the bitstream file, in the Simulink Toolstrip, in the **HDL Code** tab, click **Build Bitstream** and wait until the synthesis tool runs in the external window.

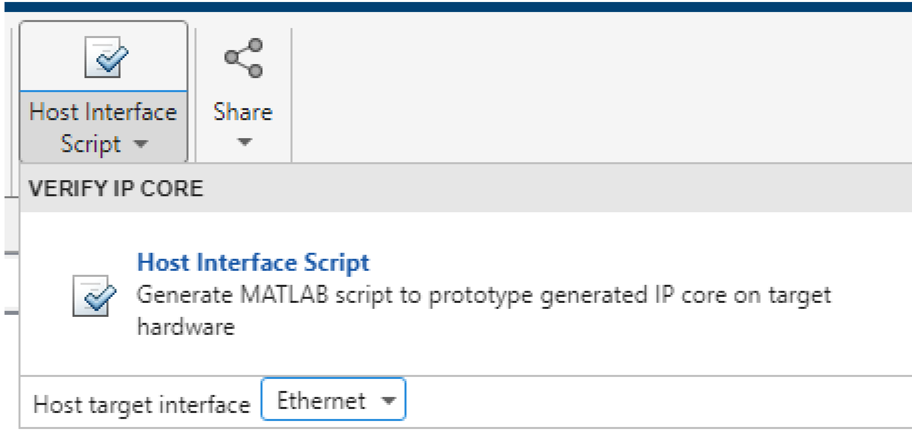
To download the bitstream, in the Simulink Toolstrip, in the **HDL Code** tab**,** select **Build Bitstream > Program Target Device**.

After you program the FPGA hardware, the LED starts blinking on your Intel SoC board.



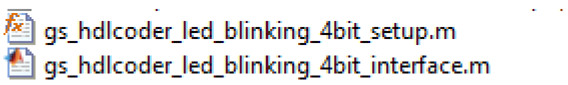
Prototype and Verify IP Core on Hardware

Verify your generated IP core on the hardware by using MATLAB to generate a host interface script. This script contains MATLAB commands that connect your hardware and interact with your IP core. To generate a host interface script file, in the Simulink Toolstrip, in the **HDL Code** tab, select **Host Interface Script > Host Interface Script**.




```
Host Interface Script will be generated based on last stored Interface Mapping.

### Generate host interface script


Generating new Intel Host Interface script: gs_hdlcoder_led_blinking_4bit_interface.m
Intel Host Interface script generation complete.
No driver was generated for port(s) "LED" mapped to interface "LEDs General Purpose" in the host interface script.
```

HDL Coder generates two MATLAB files in your current folder that enable you to prototype your generated IP core directly from MATLAB.



Open the generated interface script file by clicking the link in the Simulink Diagnostic Viewer. This file creates a connection to your FPGA hardware that reads and writes data and configures the `fpga` hardware object with the same ports and interfaces that you mapped in the **Target Interface** table. You can reuse this function in your own scripts to recreate this configuration. For example, uncomment the lines with the `writePort` function in the `gs_hdlcoder_led_blinking_4bit_interface.m` function and modify it to change the LED blink frequency. Run the modified script and observe that the LED blink frequency changes on the hardware. In addition, uncomment lines with the `readPort` function in the `gs_hdlcoder_led_blinking_interface.m` function and observe the change in values by reading back the data values.

```
%% AXI4-Lite
% writePort(hFPGA, "Blink_frequency", zeros([1 1]));
% writePort(hFPGA, "Blink_direction", zeros([1 1]));
% data_Read_back = readPort(hFPGA, "Read_back");
```

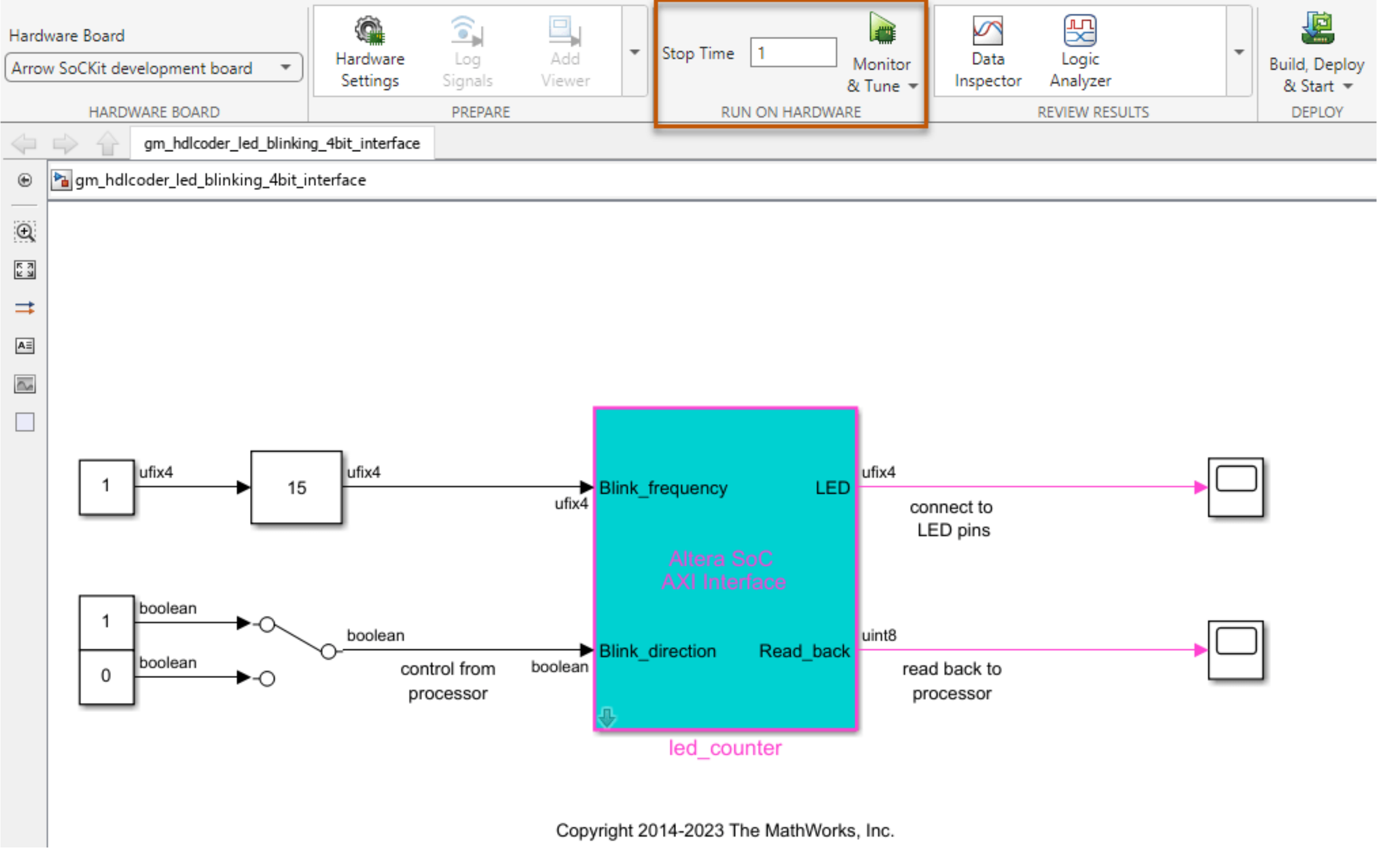
Deploy to Processor Using Software Interface Model

To target a portion of your design for the ARM processor, generate a software interface model. The software interface model contains the part of your design that runs in software. It includes the blocks outside of the HDL subsystem, and replaces the HDL subsystem with AXI driver blocks. If you have an Embedded Coder license, you can automatically generate embedded code from the software interface model, build it, and run the executable on Linux on the ARM processor. The generated embedded software generates AXI driver code from the AXI driver blocks and uses the code to control the HDL IP core. You can generate the software interface model at any stage of the IP core generation and IP core integration process.

To generate the software interface model, in the Simulink Toolstrip, in the **HDL Code** tab, select **Build Bitstream > Software Interface Model.** The Simulink Diagnostic Viewer displays a link to the generated software interface model.

```
Software Interface Model will be generated based on last stored Interface Mapping.

### Generate Simulink software interface model


Generating new Altera SoC Software Interface model: gm_hdlcoder_led_blinking_4bit_interface
Altera SoC Software Interface model generation complete.
No driver block was generated for port(s) "LED" mapped to interface "LEDs General Purpose" in the software interface model.
```

The generated software interface model replaces the `led_counter` subsystem with the AXI driver blocks that generate the interface logic between the ARM processor and FPGA.

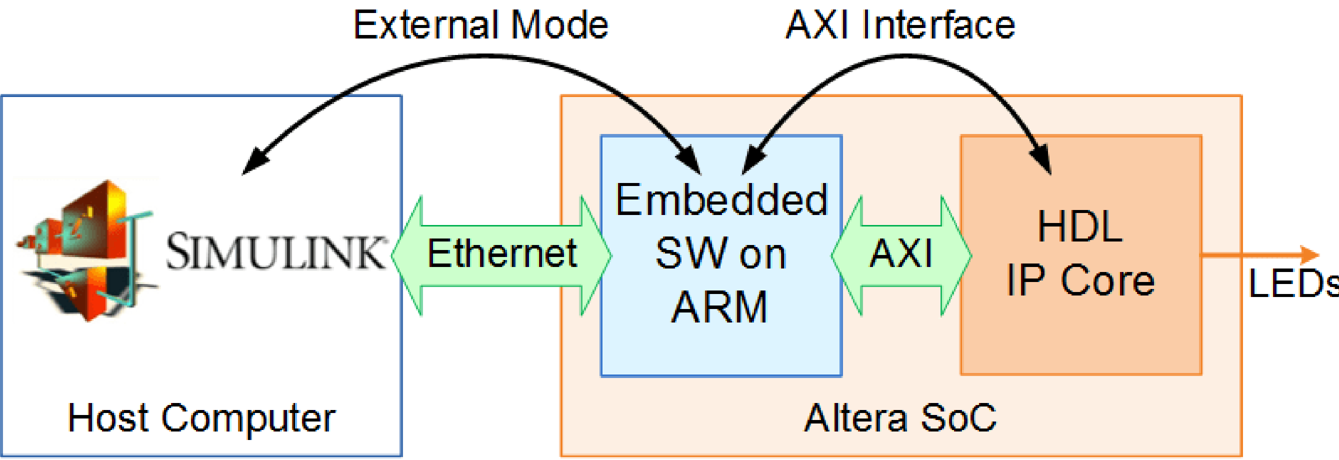


Run Software Interface Model on Intel SoC Hardware

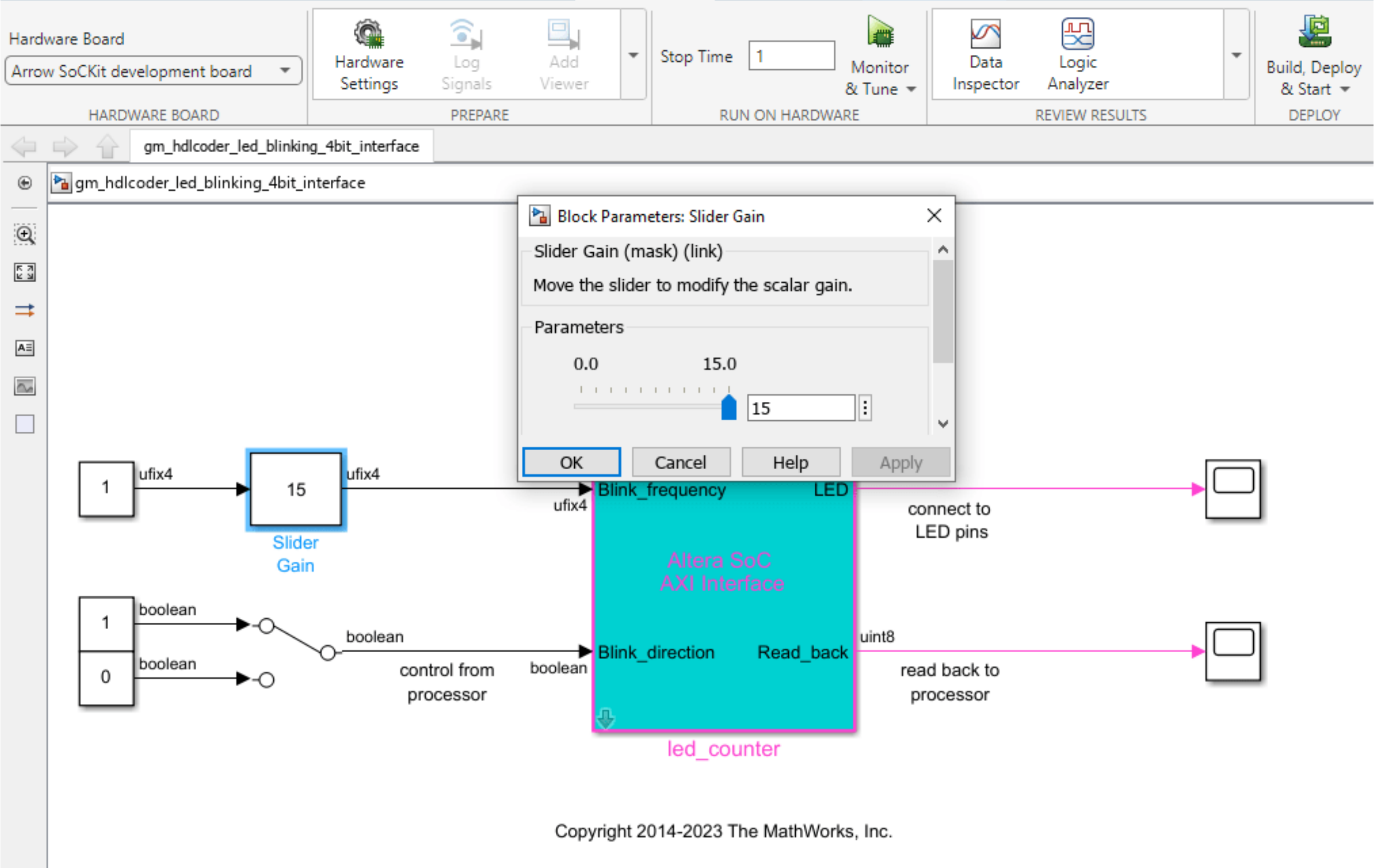
Next, configure the generated software interface model, generate the embedded C code, and run your model on the ARM processor in the Intel SoC hardware in external mode.

When you prototype and develop an algorithm, it is useful to monitor and tune the algorithm while it runs on hardware. You can use external mode to deploy your algorithm to the ARM processor in the Intel SoC hardware, and then link the algorithm with the Simulink model on the host computer through an Ethernet connection.

The Simulink model tunes and monitors the algorithm running on the hardware. Because the ARM processor is connected to the HDL IP core through the AXI interface, you can use external mode to tune parameters, and capture data from the FPGA.



1. In the generated model, in the **Hardware** tab, in the **Prepare** section, click **Hardware settings** to open the Configuration Parameters dialog box.
2. In the **Run On Hardware** section, set **Stop time** to `inf`.
3. Click **Monitor & Tune** to run your model on the ARM processor in the Intel SoC hardware in external mode. Embedded Coder builds the model, downloads the ARM executable to the Intel SoC hardware, executes it, and connects the model to the executable.
4. Double-click the Slider Gain block. Change the slider and observe the change in frequency of the LED array blinking on the Intel SoC hardware. Double-click the Manual Switch block to switch the direction of the blinking LEDs.
5. Double-click the Scope block connected to the **Read_back** output port. The Scope block captures the output data of the FPGA IP core.
6. When you are done changing model parameters, click the **Stop** button in the toolstrip, then close the system command window.



See Also

More About

- "Generate and Manage FPGA I/O Host Interface Scripts" on page 39-68
- "Comparison of IP Core Generation Techniques" on page 39-27
- "Comparison of IP Core Deployment and Verification Techniques" on page 39-30
- "Hardware-Software Co-Design Workflow for SoC Platforms" on page 39-9

Getting Started with Targeting Intel Quartus Pro Based Devices

This example shows how to define and register the board and reference design for the Intel® Arria10 SoC development kit. In this example, you use a partitioned hardware-software co-design implementation to blink LEDs at various frequencies. In this example, you register the Arria 10 SoC development kit and the reference design in the HDL Workflow Advisor. The reference design also shows the *Early I/O (Split bitstream)* feature supported by Intel Arria 10 SoC in HDL Workflow Advisor. This example uses Embedded Coder(R) to generate C code that runs on the ARM(R) processor to control the LED blink frequency. You can prepare your design for hardware and software implementation on the Intel Arria 10 SoC by deciding which system elements are performed by the programmable logic, and which system elements will run on the ARM Cortex-A9.

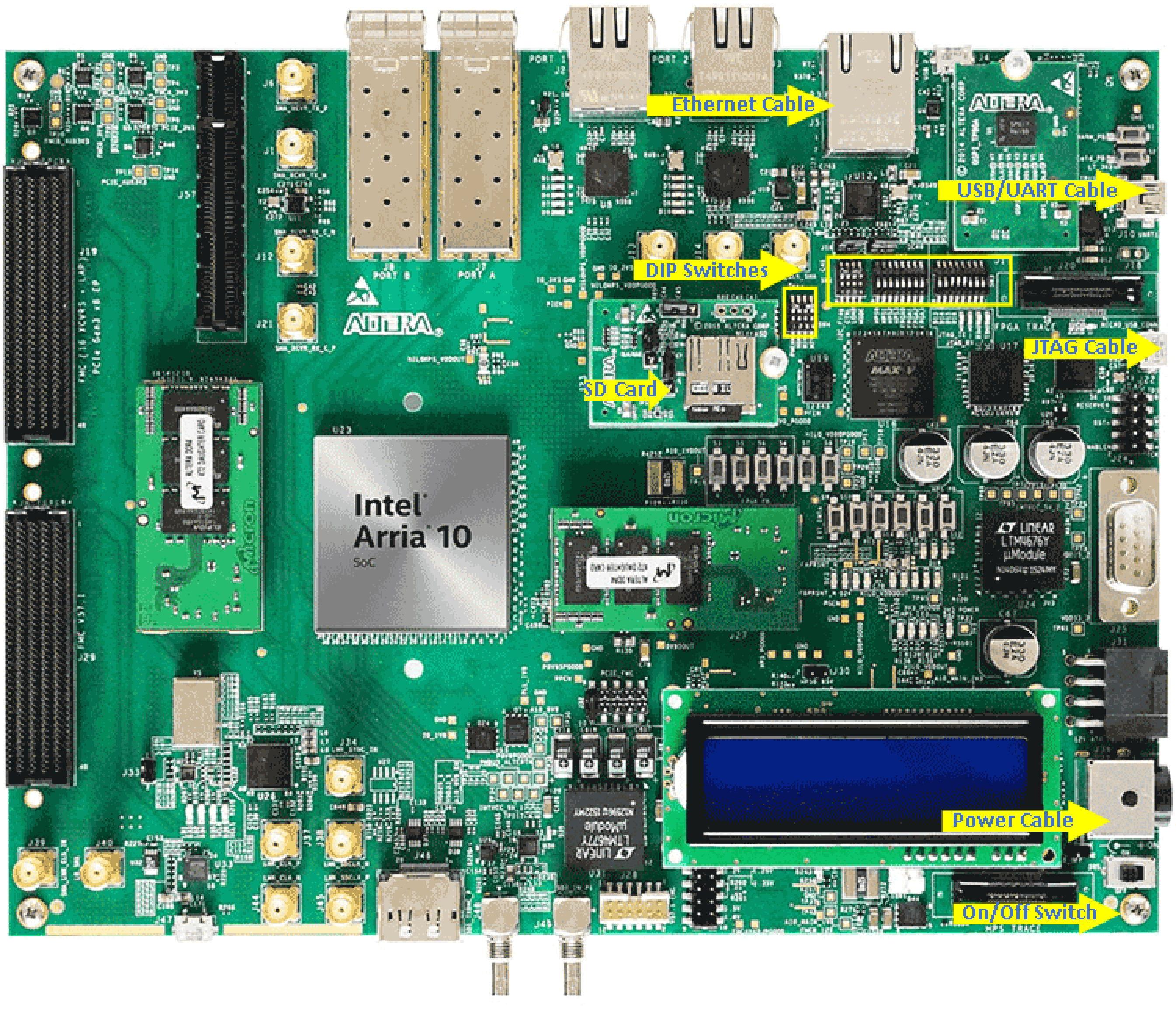
Requirements

This example requires:

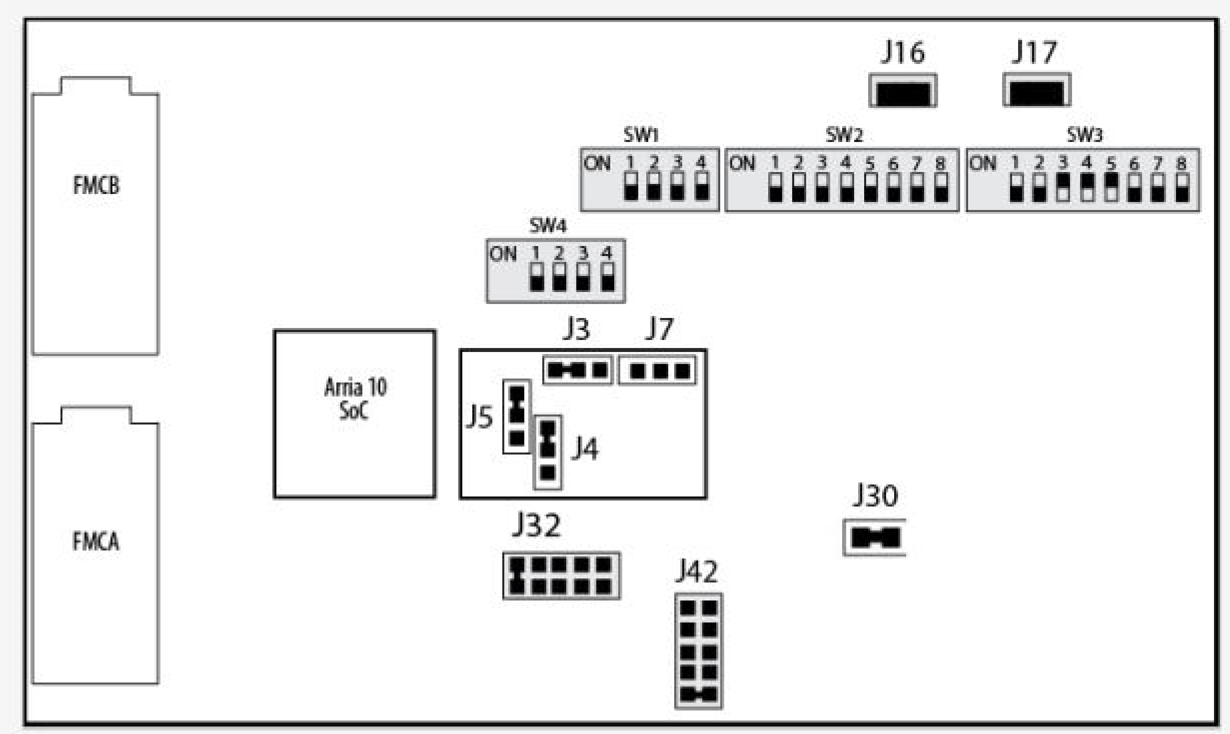
1. Supported version of Intel Quartus® Pro (or Intel QUARTUS II). For more information, see "HDL Language Support and Supported Third-Party Tools and Hardware".
2. Intel SoC Embedded Design Suite
3. Intel Arria 10 SoC development kit
4. HDL Coder Support Package for Intel FPGA and SoC Devices
5. Embedded Coder Support Package for Intel SoC Devices

Set Up Intel SoC Hardware and Tools

1. Set up the Arria 10 SoC as shown in the figure below. To learn more about the Arria 10 SoC hardware setup, refer to the board documentation.



1.1 Set up the DIP switches and jumper settings as shown in the figure below.



1.2 Connect the USB UART using a Micro-USB cable to your computer. Ensure your USB device drivers, such as for the FTDI USB to UART, are installed correctly.

1.3 Connect the Arria10 SoC kit to your computer using an Ethernet cable.

2. Install the HDL Coder Support Package for Intel FPGA and SoC Devices and Embedded Coder Support Packages for Intel SoC Devices. To start the installer, open MATLAB and , in the **Home** tab click **Add-Ons > Get Hardware Support Packages**.

3. Ensure you are using the SD card image provided by the Embedded Coder Support Package for Intel SoC Devices. If you need to update your SD card image, refer to the Hardware Setup section of "Add Support for Intel SoC Platform" (Embedded Coder).

4. Set up the Arria10 SoC hardware connection by entering:

```
h = alterasoc
```

The `alterasoc` function logs in to the hardware via the COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

5. You can optionally test the serial connection by using the following configuration and a program such as PuTTY™.

*Baud rate: `115200` *Data bits: `8` *Stop bits: `1` *Parity: `None` *Flow control: `None`. You should be able to observe Linux booting log on the serial console when you power cycle the Arria10 SoCKit board. You must close this serial connection before using the `alterasoc` function again.

6. Set up the Intel Quartus Pro synthesis tool path by entering this command. Use your own Quartus installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Intel Quartus Pro', 'ToolPath', 'C:\intelFPGA\19.4\quartus\bin64\qp
```

If you are using an Intel QUARTUS II, use the follow command:

```
hdlsetuptoolpath('ToolName', 'Intel QUARTUS II', 'ToolPath', 'C:\intelFPGA\18.1\quartus\bin64\qua
```

Partition your design for hardware and software implementation

The first step of the Intel SoC hardware-software co-design workflow is to decide which parts of your design to implement on the programmable logic, and which parts to run on the ARM processor.

Group all the blocks you want to implement on programmable logic into an atomic subsystem. This atomic subsystem is the boundary of your hardware-software partition. All the blocks inside this subsystem are implemented on programmable logic, and all the blocks outside this subsystem will run on the ARM processor.

In this example, the subsystem **led_counter** is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem **led_counter** are for software implementation.

In Simulink, you can use the **Slider Gain** or **Manual Switch** block to adjust the input values of the hardware subsystem. In the embedded software, this means the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. The output port, **Read_Back**, can be used to read data back to the processor.

```
open_system('hdlcoder_led_blinking_4bit');
```

Generate an HDL IP core using the HDL Workflow Advisor

Using the `IP Core Generation` workflow in the HDL Workflow Advisor enables you to automatically generate a sharable and reusable IP core module from a Simulink model. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic connecting the IP core to the embedded processor. HDL Coder packages all the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Intel Qsys environment.

1. Start the IP core generation workflow.

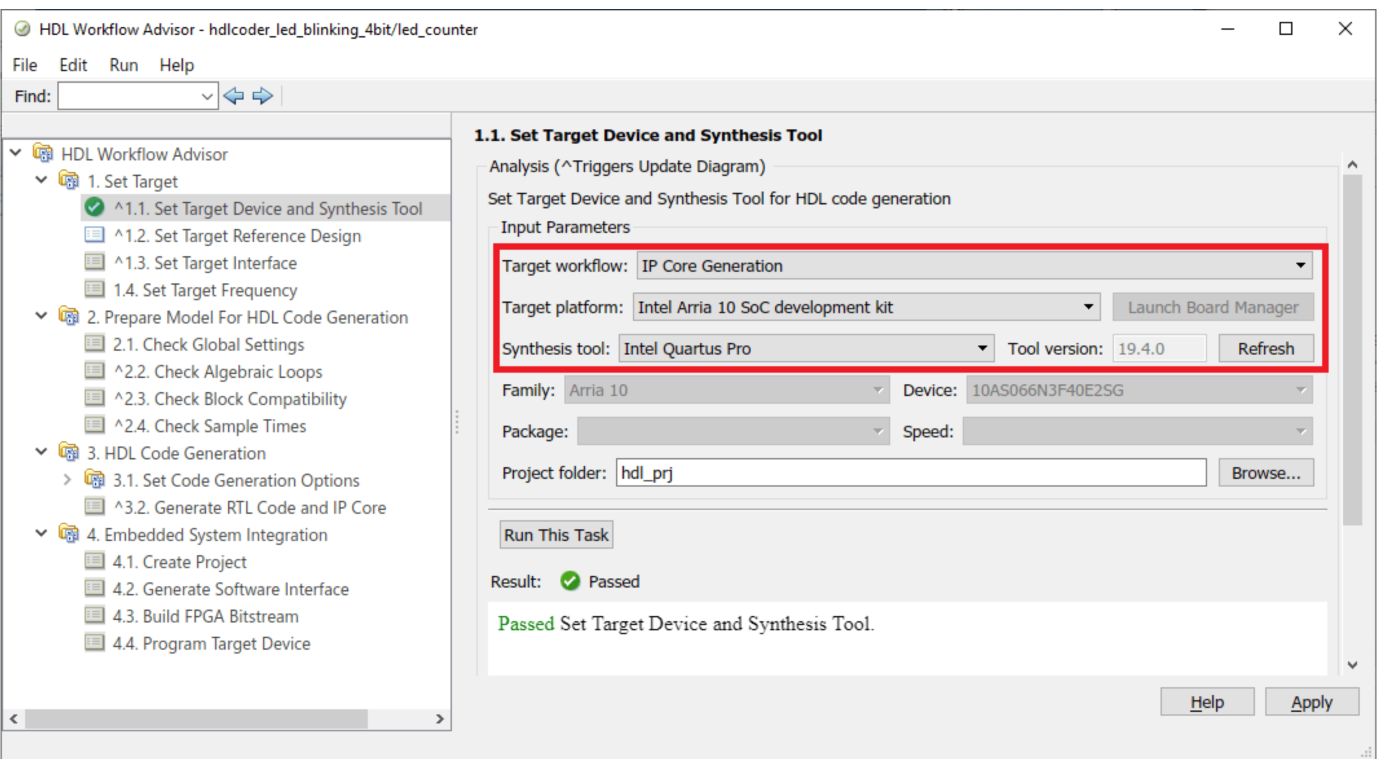
1.1. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking_4bit/led_counter` subsystem by right-clicking the `led_counter` subsystem, and choosing **HDL Code** > **HDL Workflow Advisor**.

1.2. In the **Set Target** > **Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.

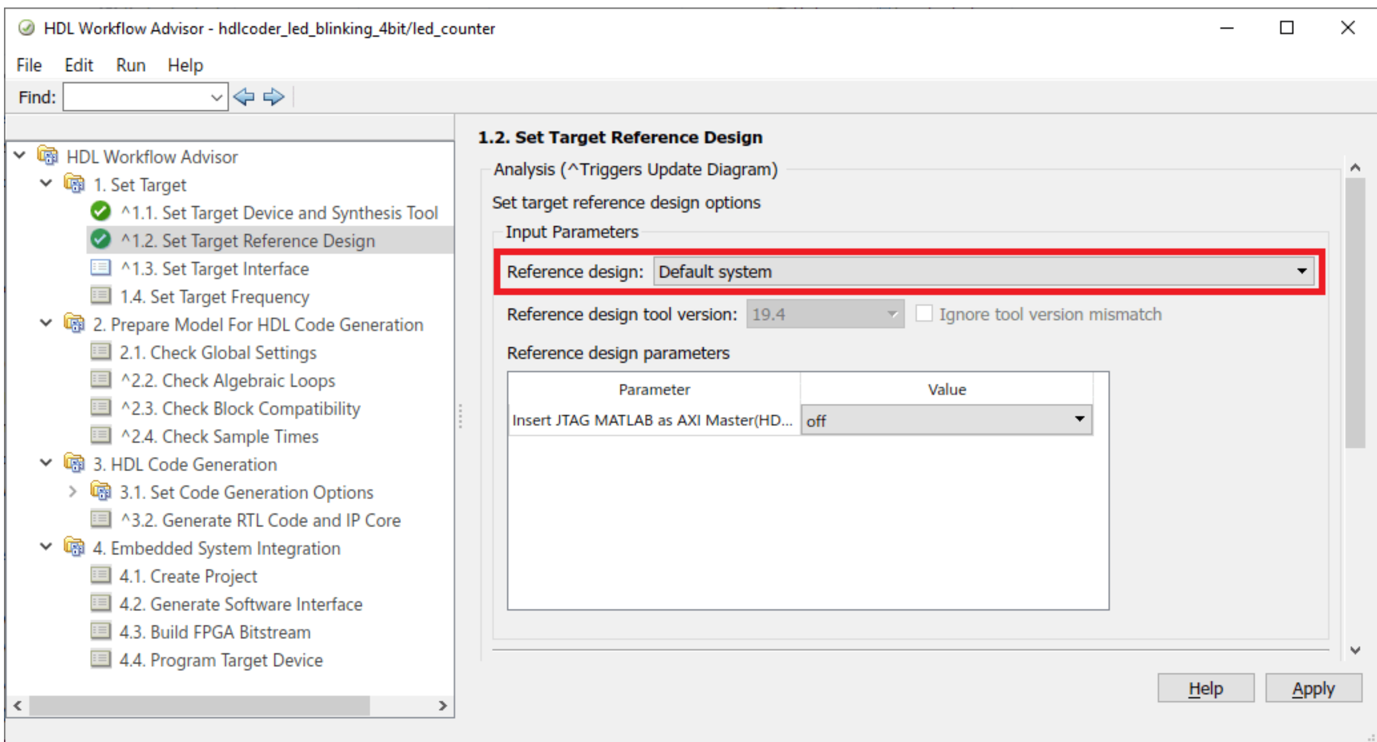
1.3. For **Target platform**, select **Intel Arria 10 SoC development kit**. If you don't have this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Intel FPGA and SoC Devices and follow the instructions provided by the Support Package Installer to complete the installation.

1.4. Select the **Synthesis tool** as **Intel Quartus Pro** (or **Altera QUARTUS II**)

1.5. Click **Run This Task** to run the **Set Target Device and Synthesis Tool task**.



1.6. In the **Set Target** > **Set Target Reference Design** task, choose **Default system**. For this example, it is selected by default.



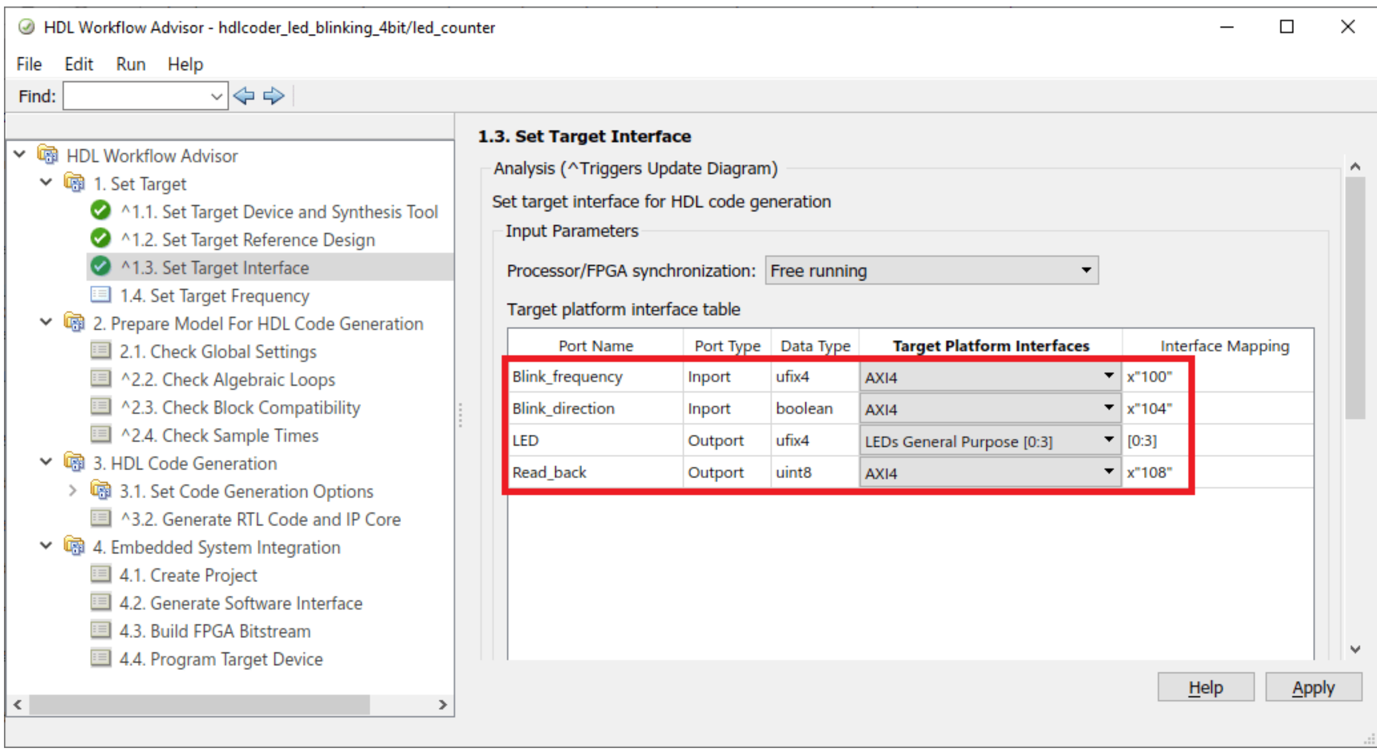
1.7. Click **Run This Task** to run the **Set Target Reference Design** task.

2. Configure the target interface.

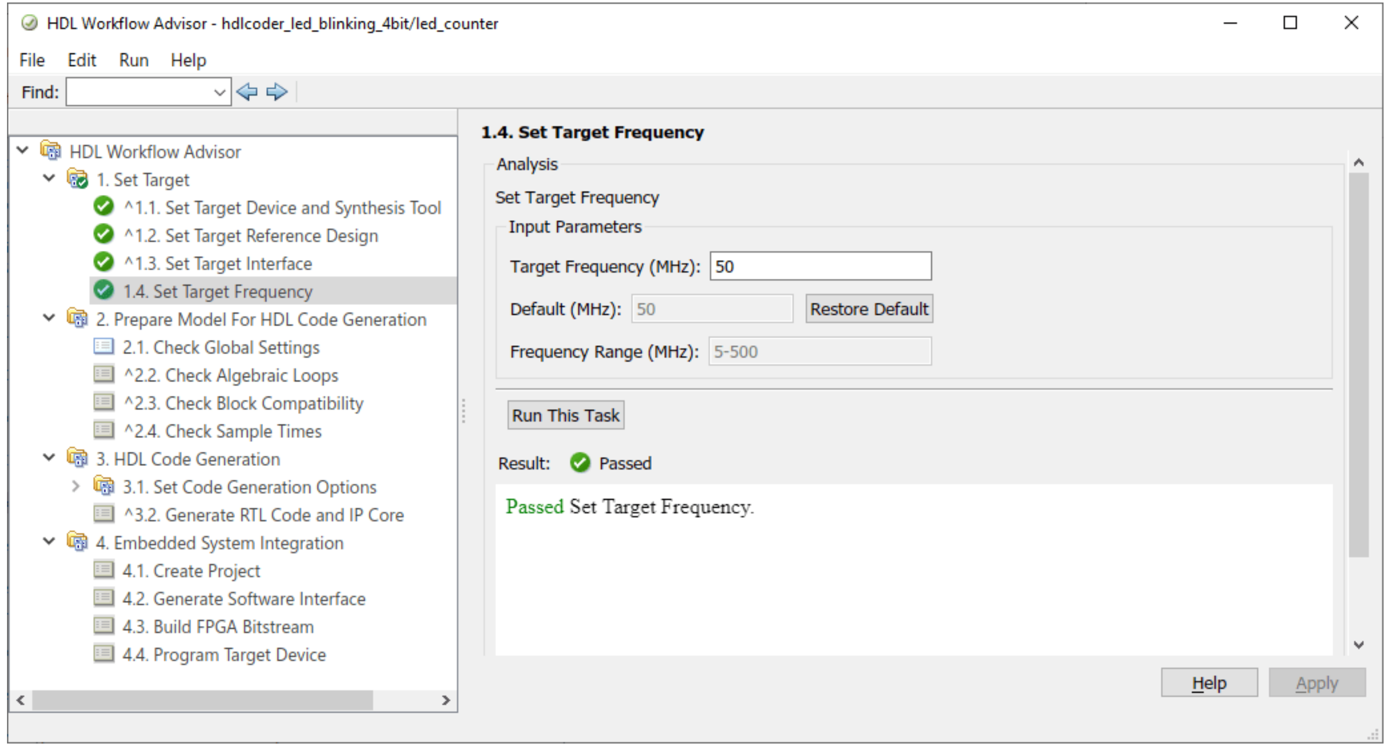
Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4 interface, so HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:3]**, which connects to the LED hardware on the Intel SoC board.

2.1 In the **Set Target > Set Target Interface** task, choose **AXI4** for **Blink_frequency**, **Blink_direction**, and **Read_back**.

2.2 Choose **LEDs General Purpose [0:3]** for **LED**.

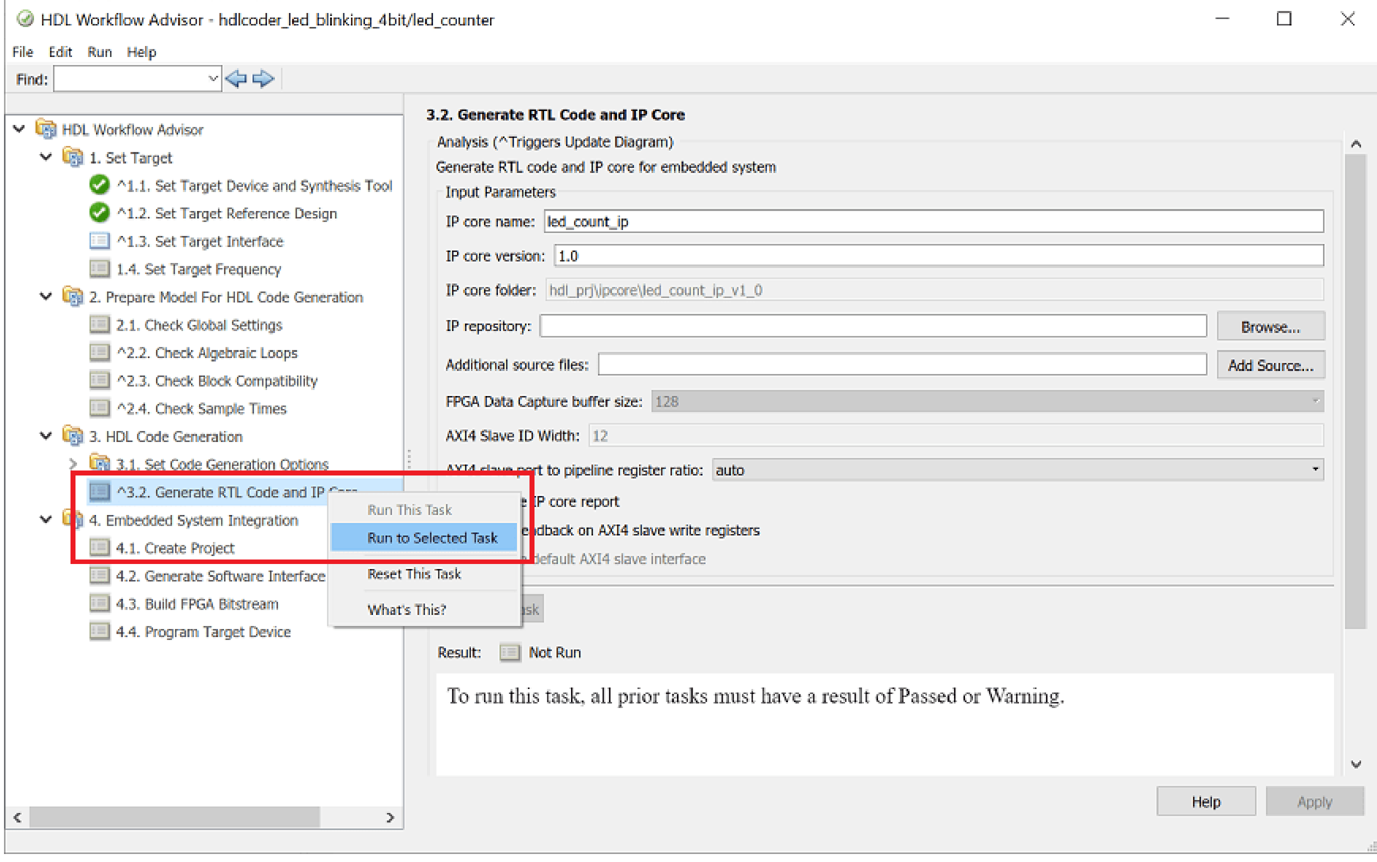


2.3 In the **Set Target > Set Target Frequency** task, choose **Target Frequency as 50 MHz**.



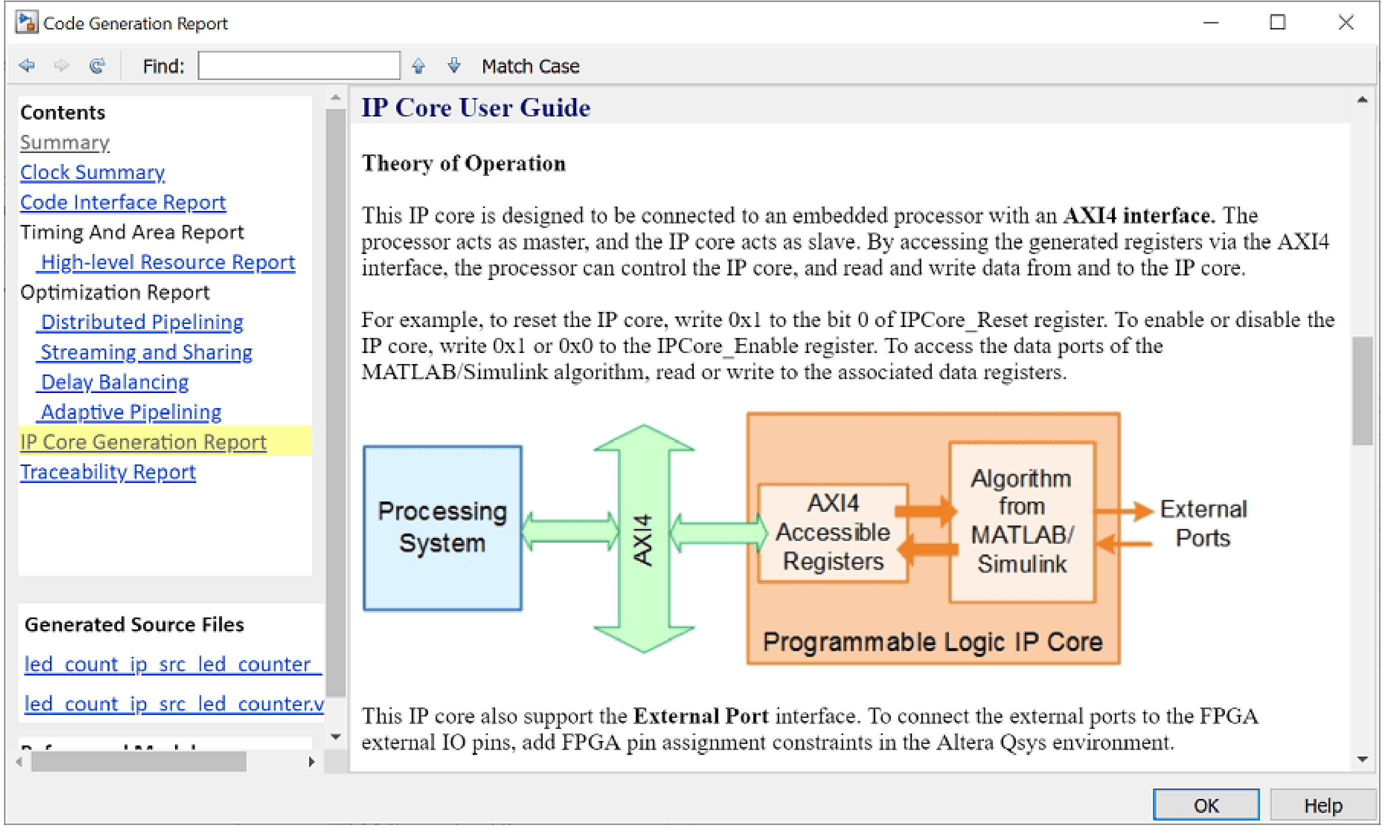
3. Generate the IP Core.

To generate the IP core, right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.



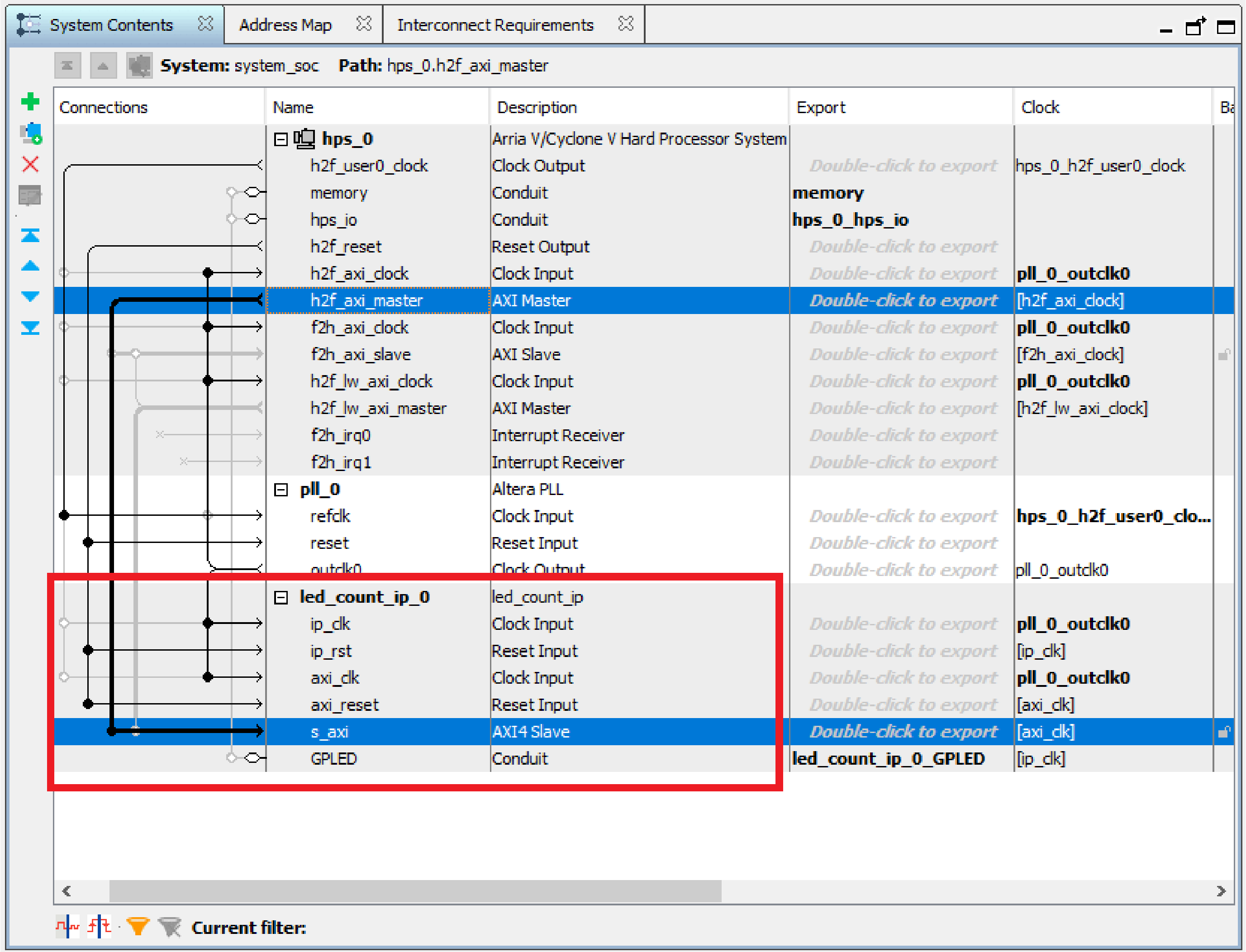
4. Generate and view the IP core report.

After you generate the custom IP core, the IP core files are in the **ipcore** folder within your project folder. An HTML custom IP core report is generated together with the custom IP core. The report describes the behavior and contents of the generated custom IP core.



5. Follow step 1 of **Integrate the IP core with the Intel Qsys environment** section of “Getting Started with Targeting Intel SoC Devices” on page 39-132 example to integrate the IP core in the reference design and create the Qsys project.

6. Now let us examine the Intel Qsys project created by the SoC workflow after completing the Create Project task under Embedded System Integration. The following figure shows the SoC project where we have highlighted the HDL IP Core. It is instructive to compare this project with the previous project used in the custom reference design plugin for a deeper understanding of the relationship between a custom reference design and an HDL IP Core.



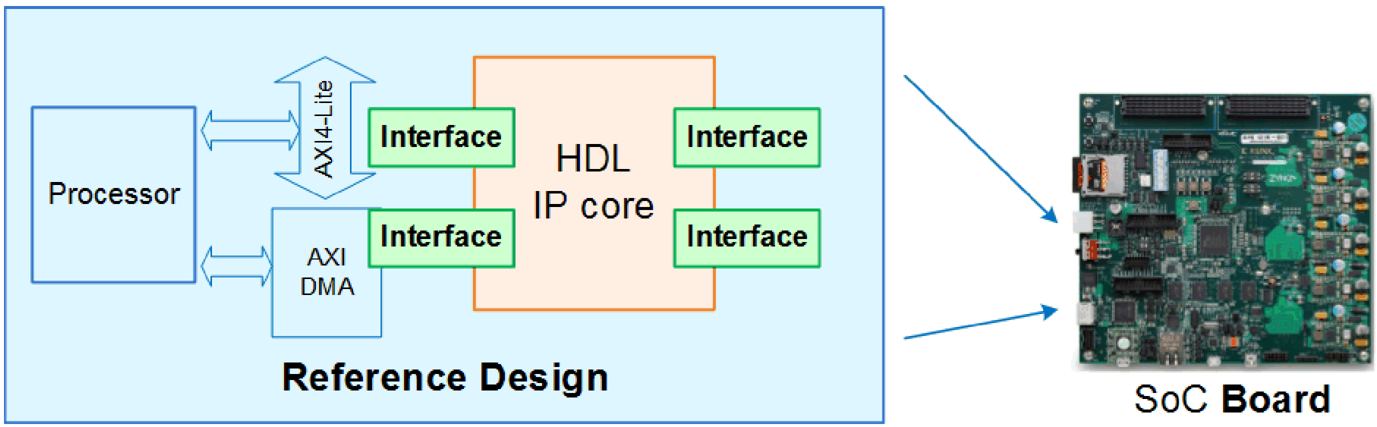
7. Follow the steps 2, 3 and 4 of **Integrate the IP core with the Intel Qsys environment** section of “Getting Started with Targeting Intel SoC Devices” on page 39-132 example to generate software interface model, generate FPGA bitstream and program target device respectively.

8. The LEDs on the Arria 10 SoC will start blinking after loading the bitstream. In addition, you can control the LED blink frequency and direction by executing the software interface model. Refer to the example “Getting Started with Targeting Intel SoC Devices” on page 39-132 example to control the LED blink frequency and direction from the generated software interface model.

If you want to create a reference design, see Create a reference design using Intel Quartus Pro on page 39-156. To register the custom reference design, see Register the custom reference design in the HDL Workflow Advisor on page 39-159.

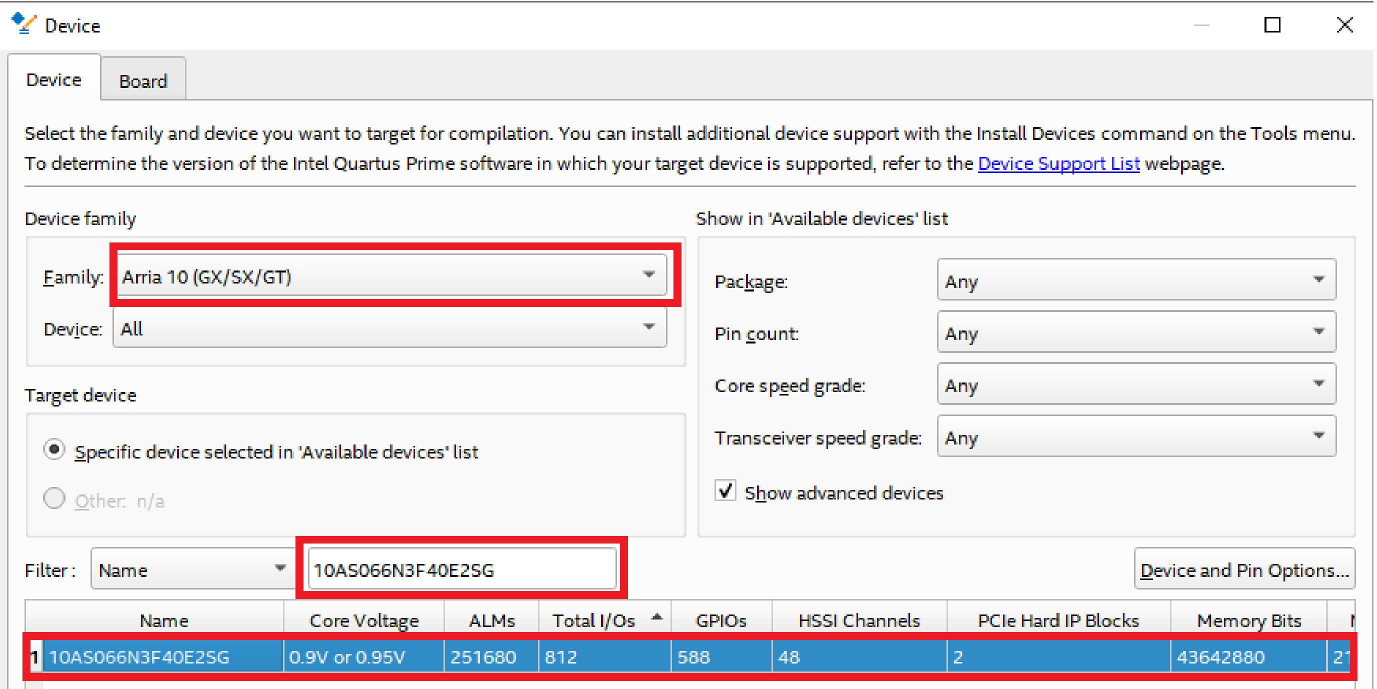
Create a Reference Design Using Intel Quartus Pro

A reference design captures the complete structure of an SoC design and defines the different components and their interconnections. HDL Coder generates an IP core that integrates with the reference design, and is then used to program an SoC board. The following figure describes the relationship between a reference design, an HDL IP core, and an SoC board.



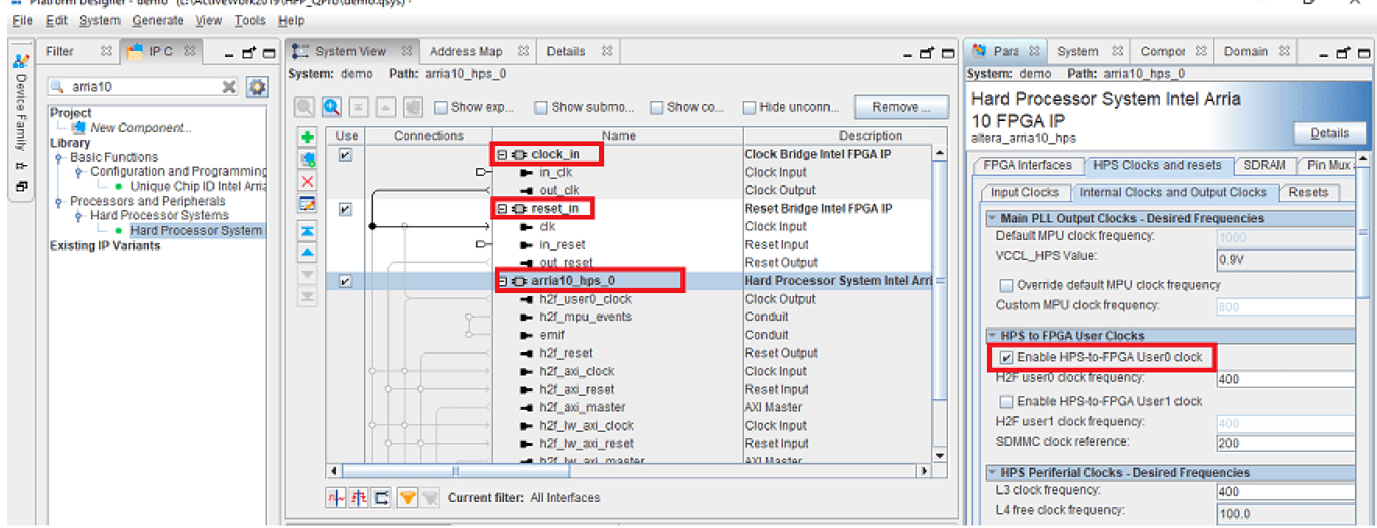
Follow these steps to create and export a simple reference design using the Intel Quartus Pro and Platform Designer (Qsys) environment. For more information about the QSys system integration tool, refer to Intel documentation.

1. Create an empty Quartus project using the New Project wizard. Enter the device part number as shown in this figure:

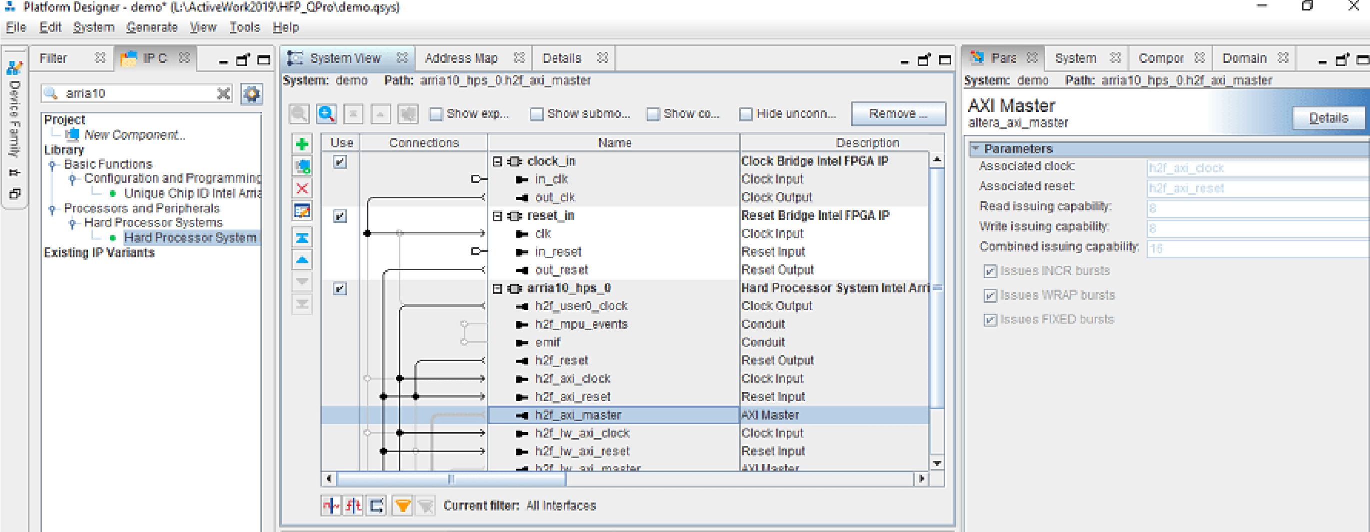


2. Initialize the Platform Designer(Qsys) in Quartus Pro by navigating to **Tools --> Platform Designer** as shown in the following figure.

3. Select Hard Processor System Intel Arria 10 FPGA IP(HPS), clock and reset IPs from IP catalog to the created Platform Designer project. Connect the required clocks and resets to the Arria 10 HPS IP as shown in the following figure. complete the other settings required for Arria 10 Hard Processor System such as Peripheral pin set and mode settings.



4. Keep h2f_axi_master port connection open in order to connect to DUT IP during the process of workflow IP integration. Complete the rest of the connections between Altera PLL IP and HPS IP as shown in the following figure.



5. Save the Qsys file. This file is used when you create reference design plugin.

Register the Intel Arria 10 SoC development kit

In this section, You register the Intel Arria 10 SoC development kit in HDL Workflow Advisor.

1. Create a board registration file with the name `hdlcoder_board_customization.m` and add it to the MATLAB path.

For more details on creating a board registration file, refer to "Define Custom Board and Reference Design for Intel SoC Workflow" on page 40-270.

2. Create the board definition file.

A board definition file contains information about the SoC board.

For more details on creating the board definition file, refer to "Define Custom Board and Reference Design for Intel SoC Workflow" on page 40-270.

Register the custom reference design in HDL Workflow Advisor

In this section, You register the custom reference design in HDL Workflow Advisor.

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` containing a list of reference design plugins associated with an SoC board.

For more details on creating a custom reference design, refer to "Define Custom Board and Reference Design for Intel SoC Workflow" on page 40-270.

2. Create the reference design definition file.

A reference design definition file defines the interfaces between the custom reference design and the HDL IP core that is generated by the HDL Coder SoC workflow. For more details on creating the

reference design definition file, refer to “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270.

Early I/O for Arria 10: The Intel Arria 10 SoC FPGA device supports Early I/O Release.

Early IO release allows you to enable DDR functioning prior programming the core raw binary file (RBF) for speeding up the boot time. In this flow, the shared I/O and hard memory controller I/O are configured and released allowing HPS immediate access to them.

This feature splits the FPGA configuration sequence into two parts. The first part configures the FPGA I/O, the Shared I/O and also enables the HPS External Memory Interface (EMIF) if present. The second part of the sequence configures the core FPGA fabric. By splitting the configuration sequence, the Arria10 Hard Processing System now has access to Shared I/O and EMIF before the FPGA fabric is configured. This allows more flexibility for designs that need faster boot times or alternate boot sources. In this Early I/O, two Raw Binary Format (.rbf) files are generated: (1) **peripheral.rbf** file. (2) **core.rbf** file. Together, these configuration files contain the same data as a combined configuration .rbf file that is generated when the Early I/O Release feature is not used. The **peripheral.rbf** file is loaded first and configures the FPGA I/O, Shared I/O and HPS EMIF. The **core.rbf** is loaded next and completes the FPGA configuration sequence by configuring the FPGA fabric. After the **peripheral.rbf** is successfully loaded, the Intel Arria 10 SoC FPGA HPS EMIF pins are released and the interface begins calibration.

To use this feature, you need to enable a reference design parameter `hRD.GenerateSplitBitstream = true;` as shown in the below plugin_rd file. Accordingly if this reference design parameter is made true, it generates two .rbf files for configuring the FPGA as mentioned above.

The contents of this reference design definition file `plugin_rd.m` is similar to the Intel Quartus standard version, the differences for the Intel Quartus Pro are listed below.

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Intel Quartus Pro');

%% Add custom design files
% add custom Qsys design
hRD.addCustomQsysDesign( ...
    'CustomQsysPrjFile', 'system_soc.qsys');

% split the full rbf file into core and peripheral rbf files for Early I/O feature
hRD.GenerateSplitBitstream = true;
```

Integrate HDL IP Core with Microchip PolarFire SoC Icicle Kit Reference Design

This example shows how to use the hardware-software co-design workflow to blink LEDs at various frequencies on the Microchip PolarFire® SoC Icicle kit.

Introduction

This example is a step-by-step guide that helps you use HDL Coder™ to generate a custom HDL IP core that blinks LEDs on the Microchip PolarFire SoC Icicle kit.

You can use MATLAB® and Simulink® to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your design for hardware and software implementation on the PolarFire SoC Icicle Kit by deciding which system elements are performed by the programmable logic, and which system elements are run on the RISC-V processor.

Using the guided workflow shown in this example, you automatically generate HDL code for the programmable logic by using HDL Coder and implement the design on the Microchip PolarFire SoC Icicle Kit.

In this workflow, you perform these tasks:

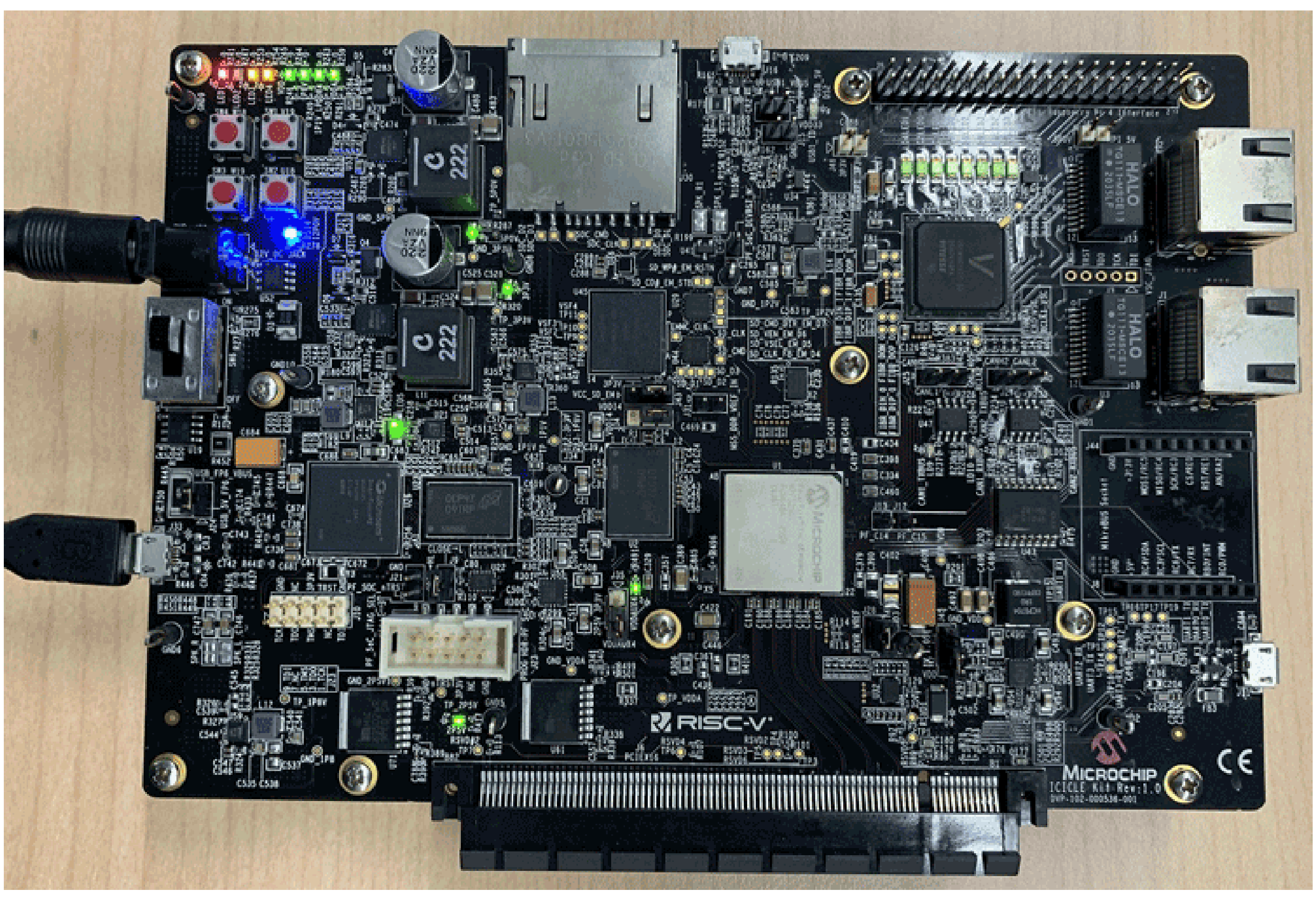
1. Set up your PolarFire SoC hardware and tools.
2. Partition your design for hardware and software implementation.
3. Generate an HDL IP core by using HDL Workflow Advisor.
4. Integrate the IP core into a Microchip Libero project and program the PolarFireSoC hardware.

Requirements

1. Microchip Libero Design Suite, with supported version listed in the “HDL Language Support and Supported Third-Party Tools and Hardware”.
2. Microchip PolarFire SoC Icicle Kit.
3. HDL Coder Support Package for Microchip FPGA and SoC Devices.

Set Up PolarFire SoC Hardware and Tools

1. Set up the Microchip PolarFire SoC Icicle Kit as shown in the figure. To learn more about the PolarFire SoC hardware setup, see PolarFire SoC Icicle Kit Quickstart Card.



Connect your computer to the USB UART connector by using a Micro-USB cable. Make sure that your USB device drivers, such as for the Silicon Labs CP210x USB to UART Bridge, are installed correctly. If not, search for the drivers online and install them.

2. Connect your computer and the PolarFire SoC board using an Ethernet cable.

3. Install the HDL Coder Support Packages for Microchip FPGA and SoC Devices if you have not already done so. To start the installer, on the MATLAB toolstrip, click **Add-Ons > Get Hardware Support Packages**. Search for **HDL Coder Support Packages for Microchip FPGA and SoC Devices** and install them.

4. Make sure that you are using the SD card image provided by Microchip from Github Link.

5. Set up the PolarFire SoC hardware connection in this MATLAB Command Window.

6. You can optionally test the serial connection by using the configuration that uses a program such as PuTTY™. Baud rate: `115200`; Data bits: `8`; Stop bits: `1`; Parity: `None`; Flow control: `None`. You can see Linux booting log on the serial console when you Power Cycle the PolarFire SoC board.

7. Set up the Microchip Libero SoC synthesis tool path in the MATLAB Command Window. When you run the command, use own Microchip Libero installation path.

```
hdlsetuptoolpath('ToolName', 'Microchip Libero SoC', 'ToolPath', 'C:\Microsemi\Libero_SoC_v12.6\D
```

Partition your Design for Hardware and Software Implementation

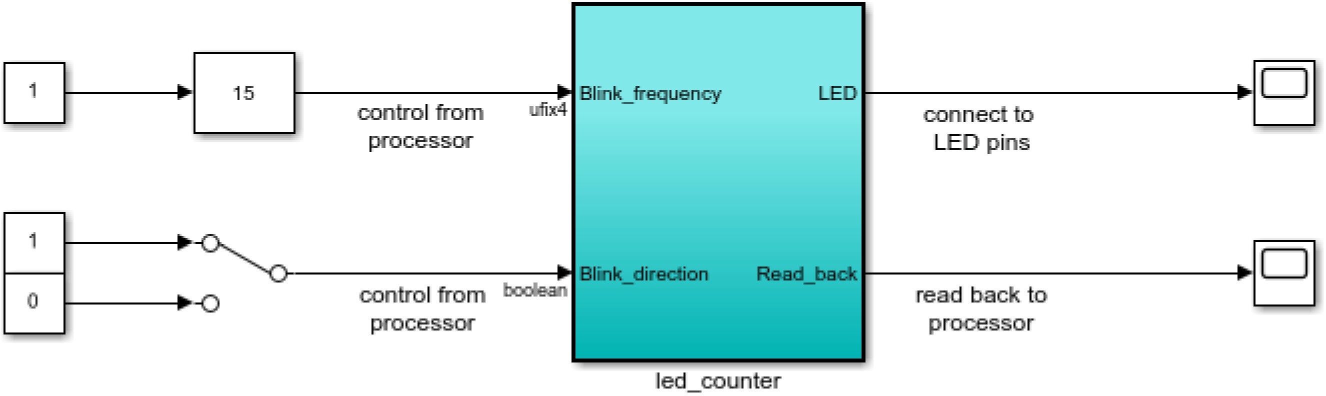
For the PolarFire SoC hardware-software co-design workflow, decide which parts of your design to implement on the programmable logic and which parts to run on the RISC-V processor.

Group all the blocks that you want to implement on programmable logic into an atomic subsystem. This atomic subsystem is the boundary of your hardware-software partition. The blocks inside this subsystem are implemented on programmable logic and the blocks outside this subsystem run on the RISC-V processor.

In this example, the subsystem **led_counter** is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are control ports that determine the LED blink frequency and direction. The blocks outside of the subsystem **led_counter** are for software implementation.

In Simulink, you can use the **Slider Gain** or **Manual Switch** block to adjust the input values of the hardware subsystem. In the embedded software, this means the RISC-V processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. You can use the output port, **Read_Back**, to read data back to the processor.

```
open_system('hdlcoder_led_blinking_4bit');
```



Generate HDL IP Core by using the HDL Workflow Advisor

You can generate a sharable and reusable IP core module from a Simulink model. The generated IP core connects to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks and generates HDL code for the AXI interface logic connecting the IP core to the embedded processor. HDL Coder packages the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Microchip Libero environment.

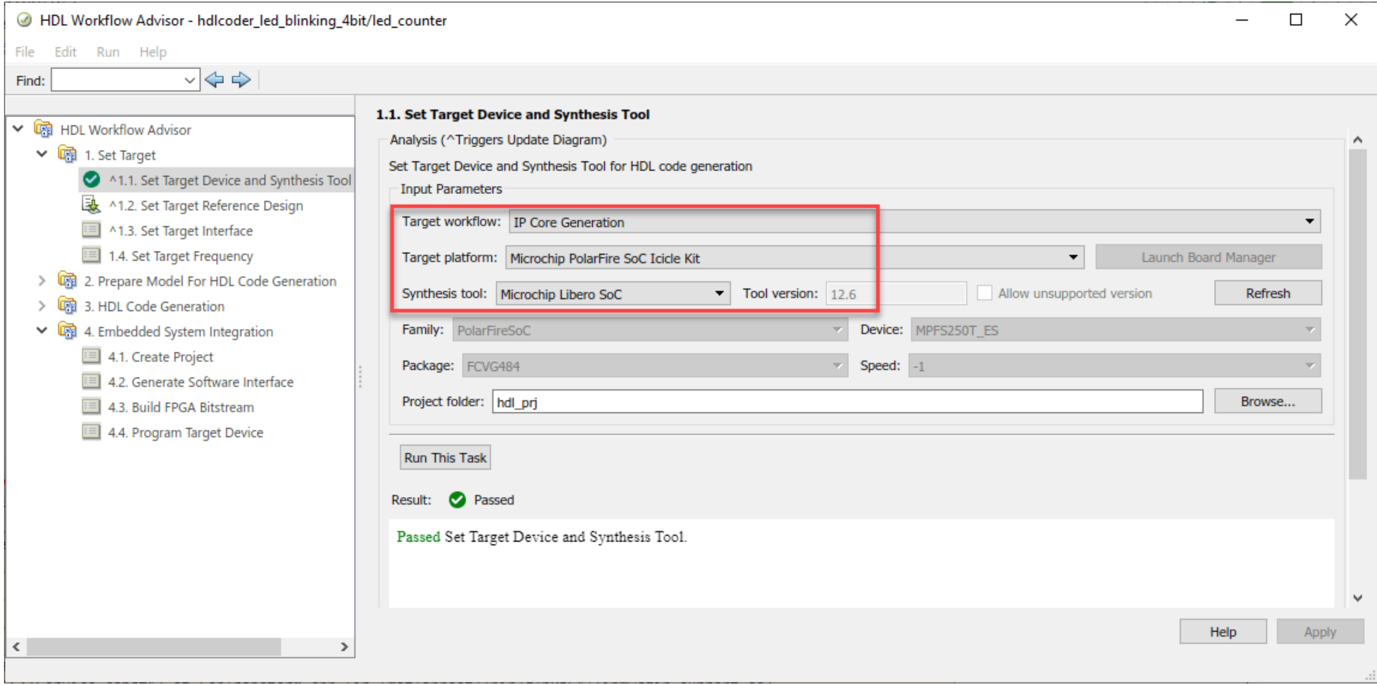
Start IP Core Generation Workflow.

1. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem, and choosing **HDL Code** > **HDL Workflow Advisor**.

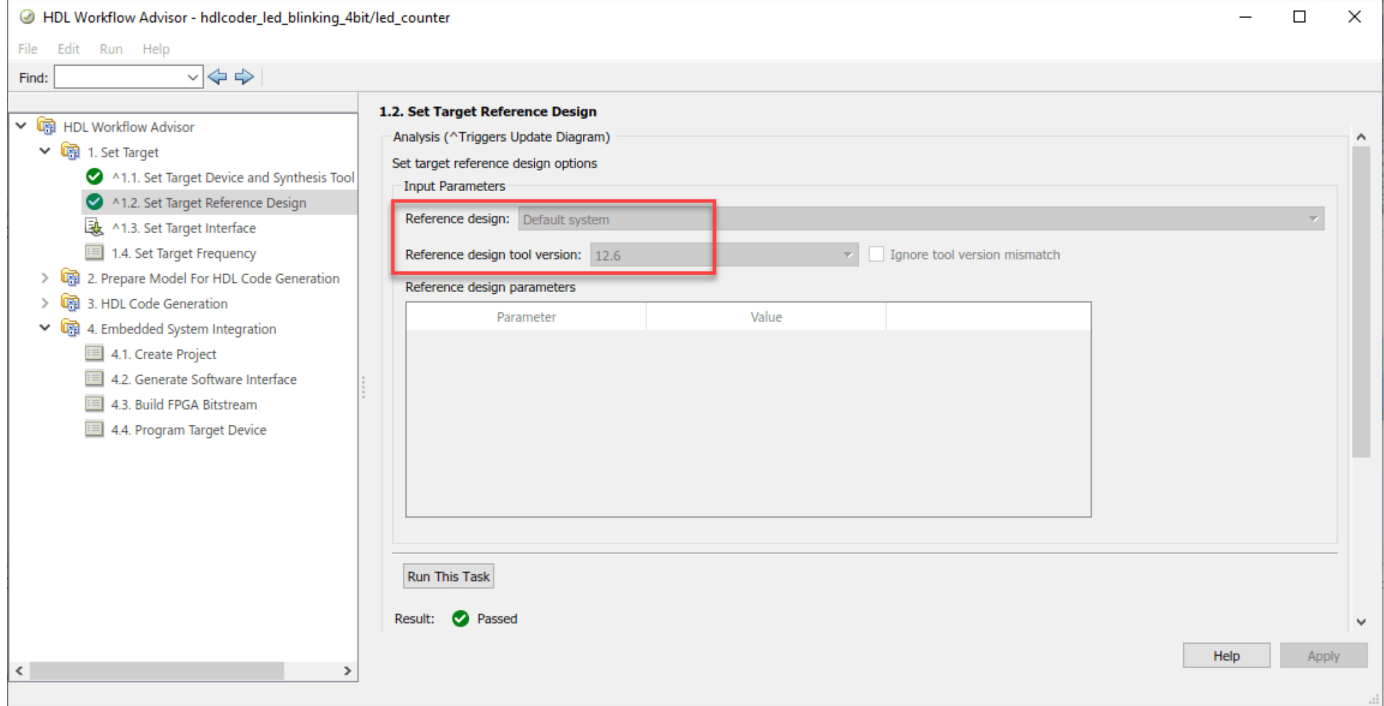
2. In the **Set Target** > **Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.

3. For **Target platform**, select **Microchip PolarFire SoC Icicle kit**. If you don't have this option, select **Get more** to open the Support Package Installer.

4. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.



5. In the **Set Target** > **Set Target Reference Design** task, Reference Design **Default system** is selected by default.



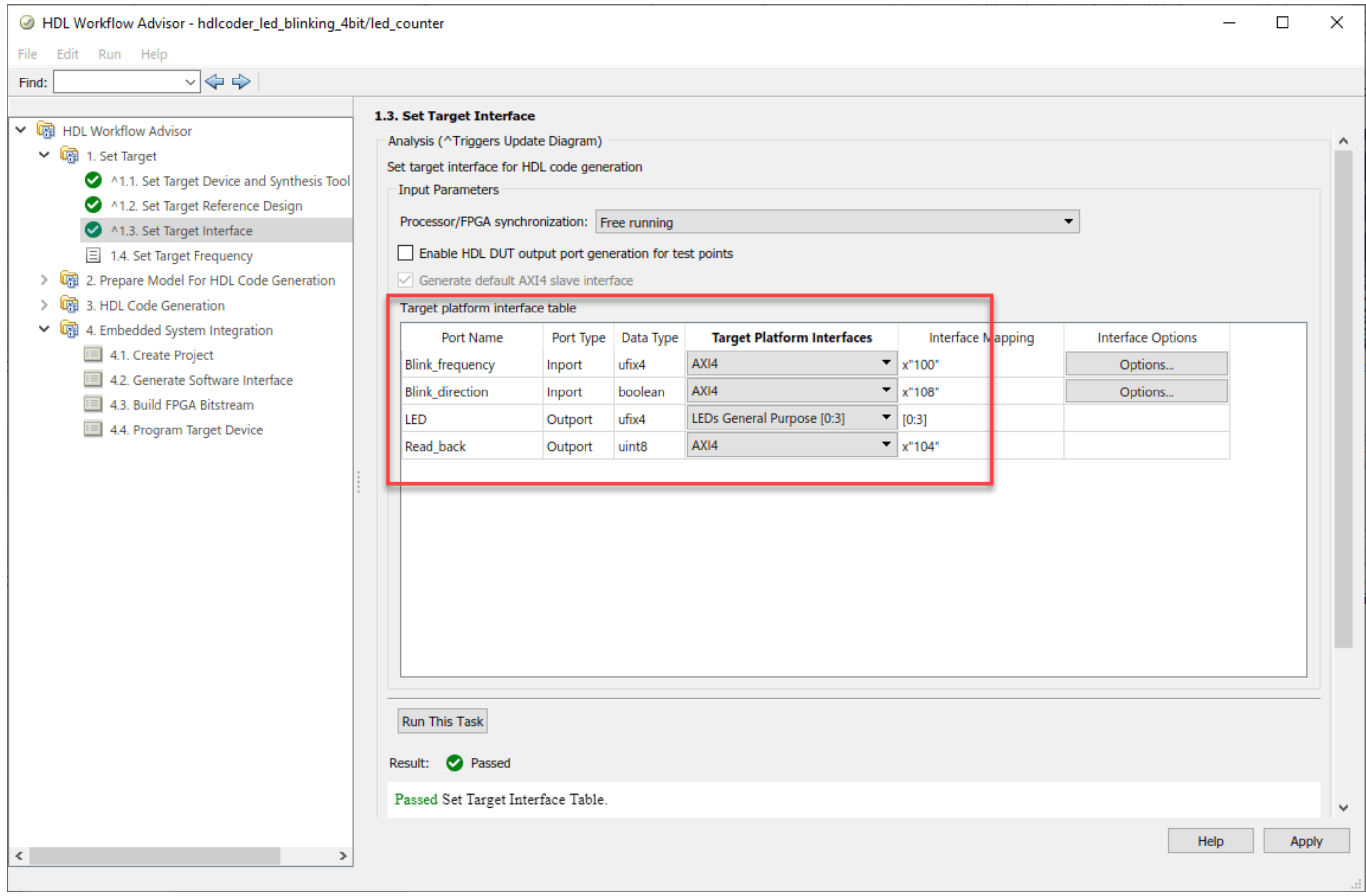
6. Click **Run This Task** to run the **Set Target Reference Design** task.

Configure Target Interface.

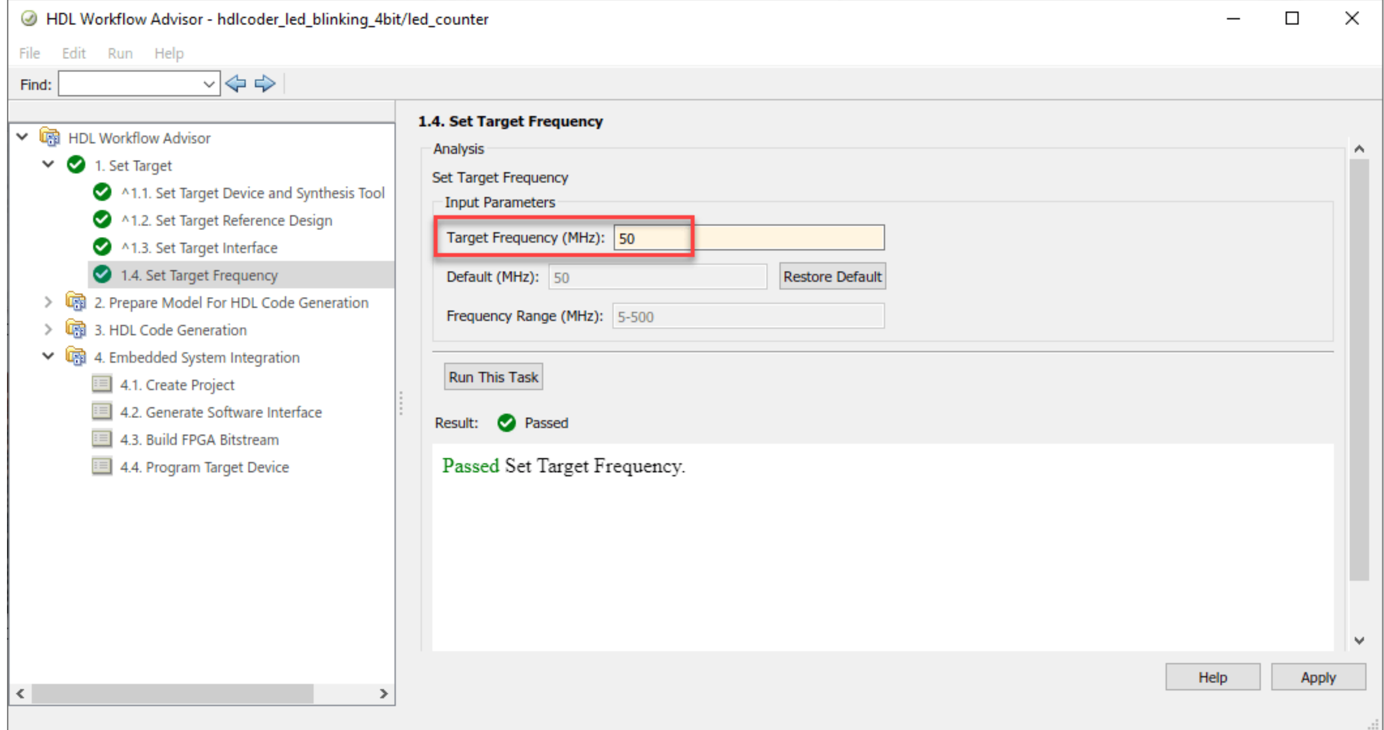
Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4 interface. HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:3]**, which connects to the LED hardware on the PolarFire SoC board.

1. In the **Set Target > Set Target Interface** task, choose **AXI4** for **Blink_frequency**, **Blink_direction**, and **Read_back**.

2. Choose **LEDs General Purpose [0:3]** for **LED**.

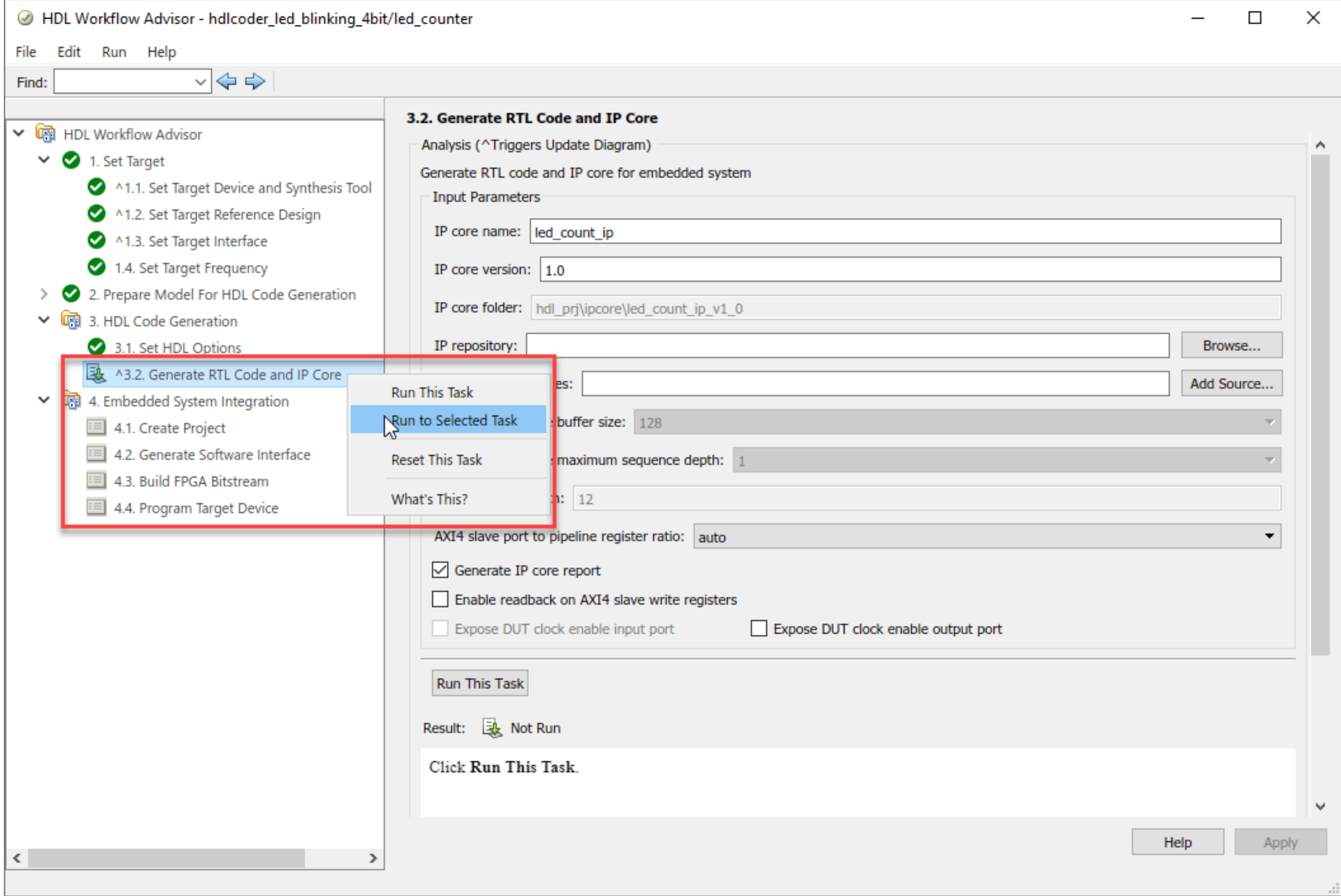


3. In the **Set Target > Set Target Frequency** task, choose **Target Frequency as 50 MHz**.



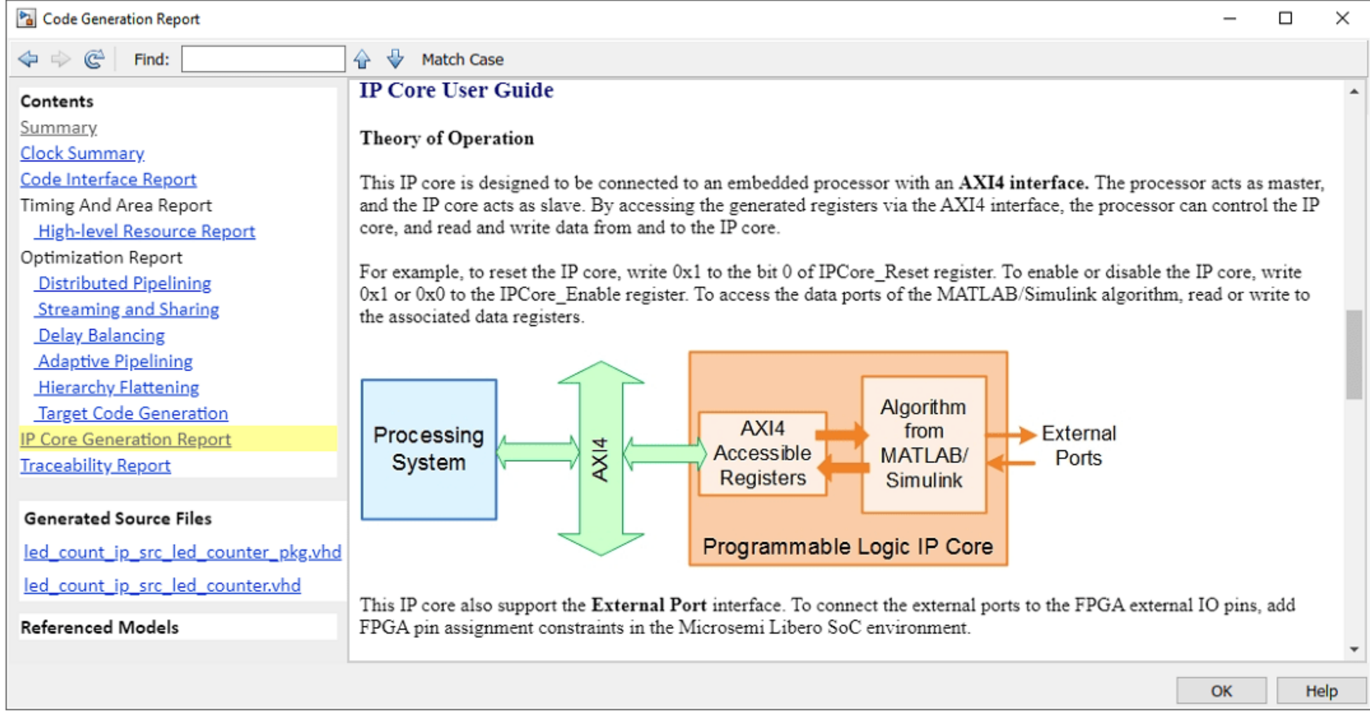
Generate IP Core

1. Right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.



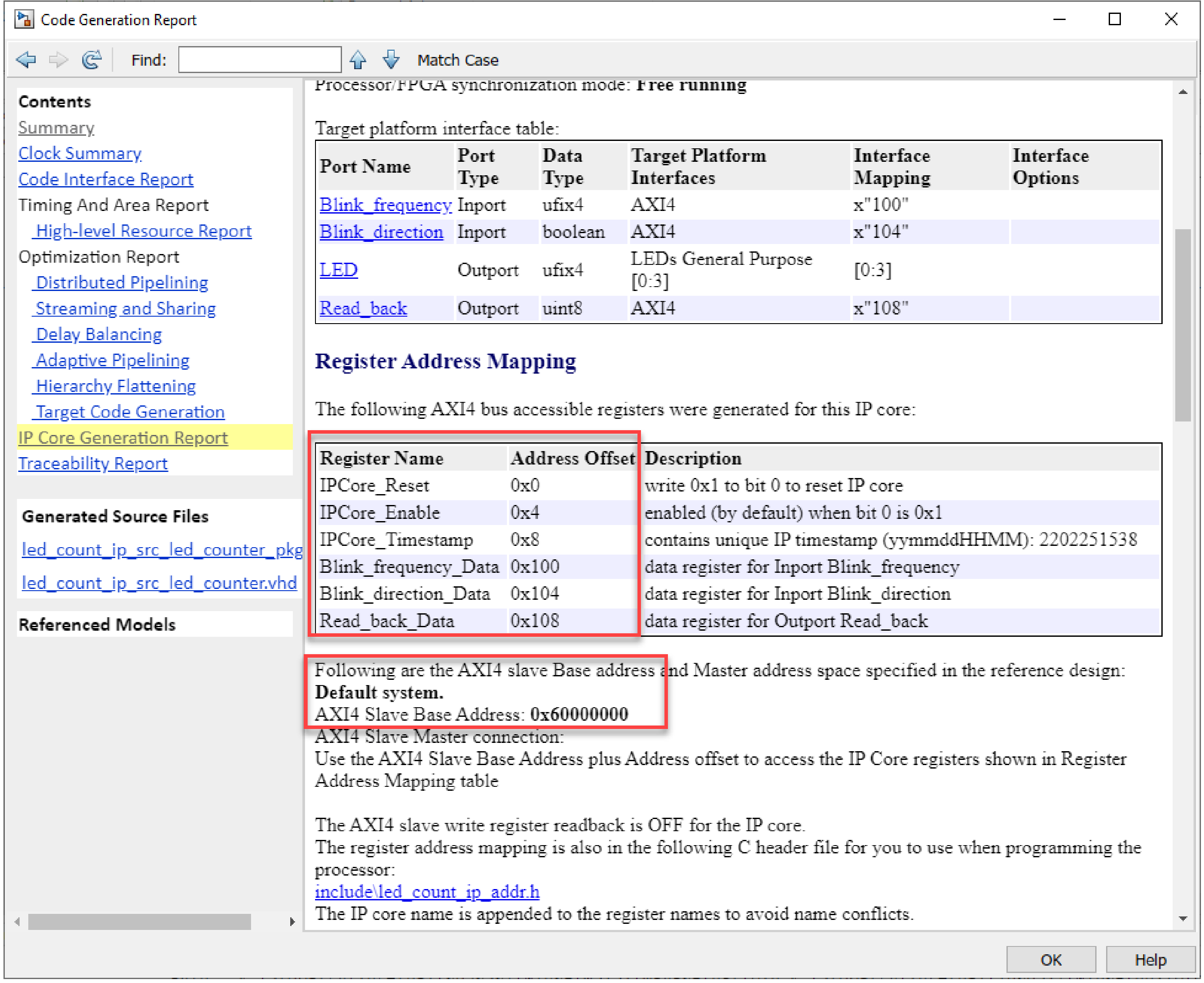
2. Generate and view the IP core report.

After you generate the custom IP core, the IP core files are in the **ipcore** folder within your project folder. An HTML custom IP core report is generated with the custom IP core. The report describes the behavior and contents of the generated custom IP core.



3. Register Address Mapping.

IP Core Generation Report also contains the details about Target Platform Interface of your model. The figure shows the AXI4 interface mapped to the hdlcoder_led_blinking_4bit model ports. The table in IP Core Generation Report shows the interface mapping address of each AXI4 slave register. These addresses are used to read the AXI4 slave input registers.



Integrate IP Core with Microchip Libero SoC Environment

Insert your generated IP core into a embedded system reference design, generate an FPGA bitstream, and download the bitstream to the PolarFire SoC hardware.

The reference design is a predefined Microchip Libero project. It contains the elements the Libero software requires to deploy your design to the PolarFire SoC board, except for the custom IP core.

1. To integrate with the Microchip Libero environment, select the **Create Project** task under **Embedded System Integration**. Click **Run This Task**. A Microchip Libero project with the IP Integrator embedded design is generated.

HDL Workflow Advisor - hdlcoder_led_blinking_4bit/led_counter

File Edit Run Help

Find:

HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Reference Design
^1.3. Set Target Interface
1.4. Set Target Frequency
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
3.1. Set HDL Options
^3.2. Generate RTL Code and IP Core
4. Embedded System Integration
4.1. Create Project
4.2. Generate Software Interface
4.3. Build FPGA Bitstream
4.4. Program Target Device

4.1. Create Project

Analysis

Create project for embedded system tool

Input Parameters

Embedded system tool:

Project folder: hdl_prj\libero_ip_prj

Synthesis objective: None

Enable IP caching

Run This Task

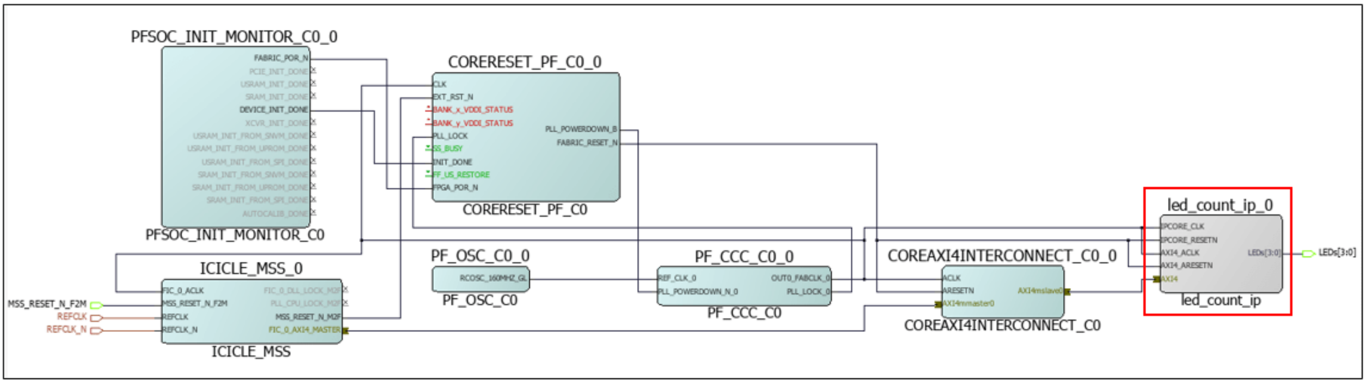
Result: Passed

Passed Create Project.

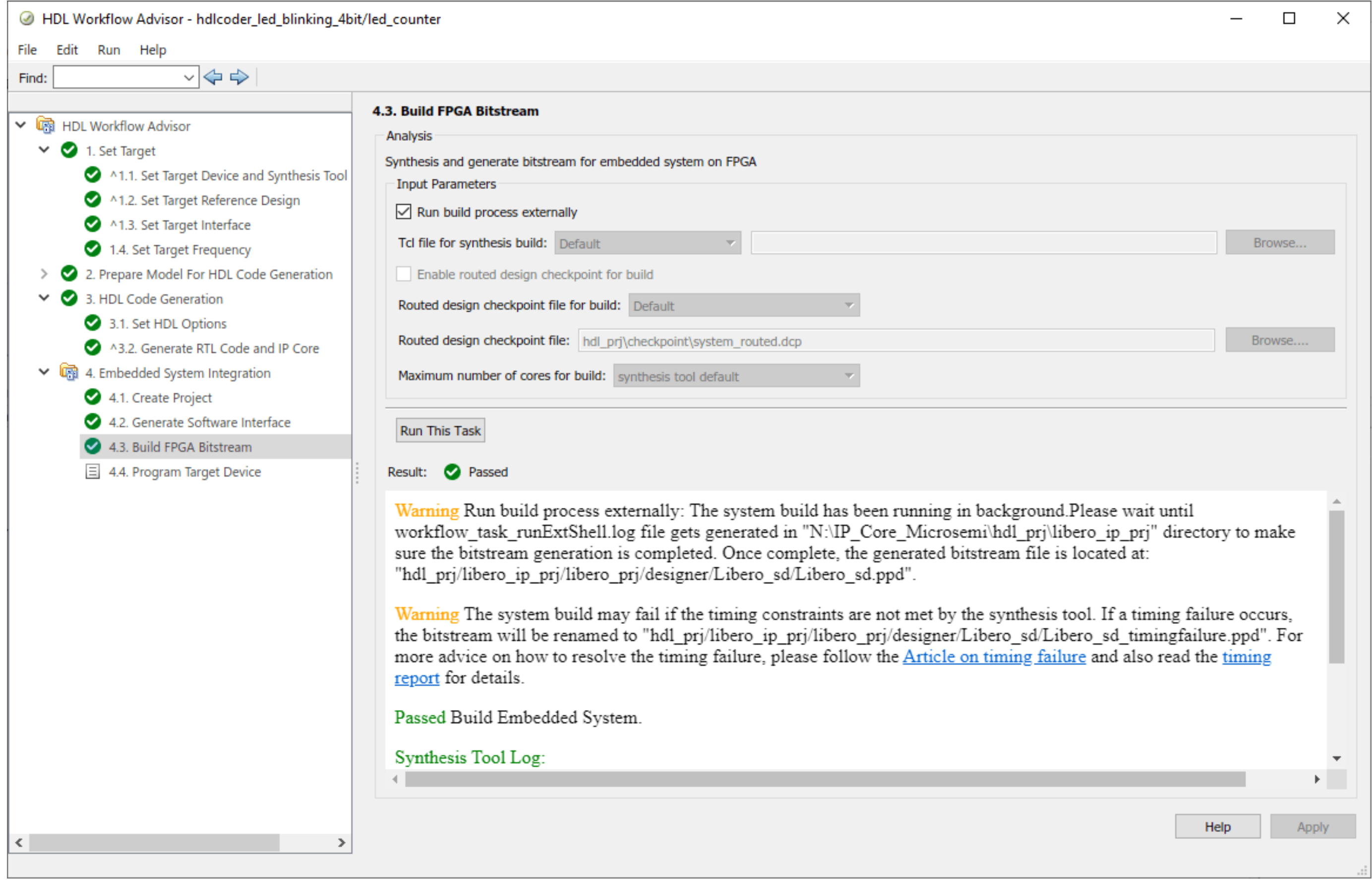
```
Task "Create Project" successful.
Generated logfile: hdl_prj\hdlsrc\hdlcoder_led_blinking_4bit\workflow_task_CreateProject.log

Generated Microchip Libero SoC project:
hdl_prj\libero_ip_prj\libero_prj.prjx
Console Mode = Info: This version of Libero supports only the enhanced constraint flow.
The libero_prj project was created.
Downloading Actel:DirectCore:CORERESET_PF:2.3.100...

Downloading Actel:SgCore:PF_CCC:2.2.100...

Downloading Actel:SgCore:PF_OSC:1.0.102...
```

Help Apply

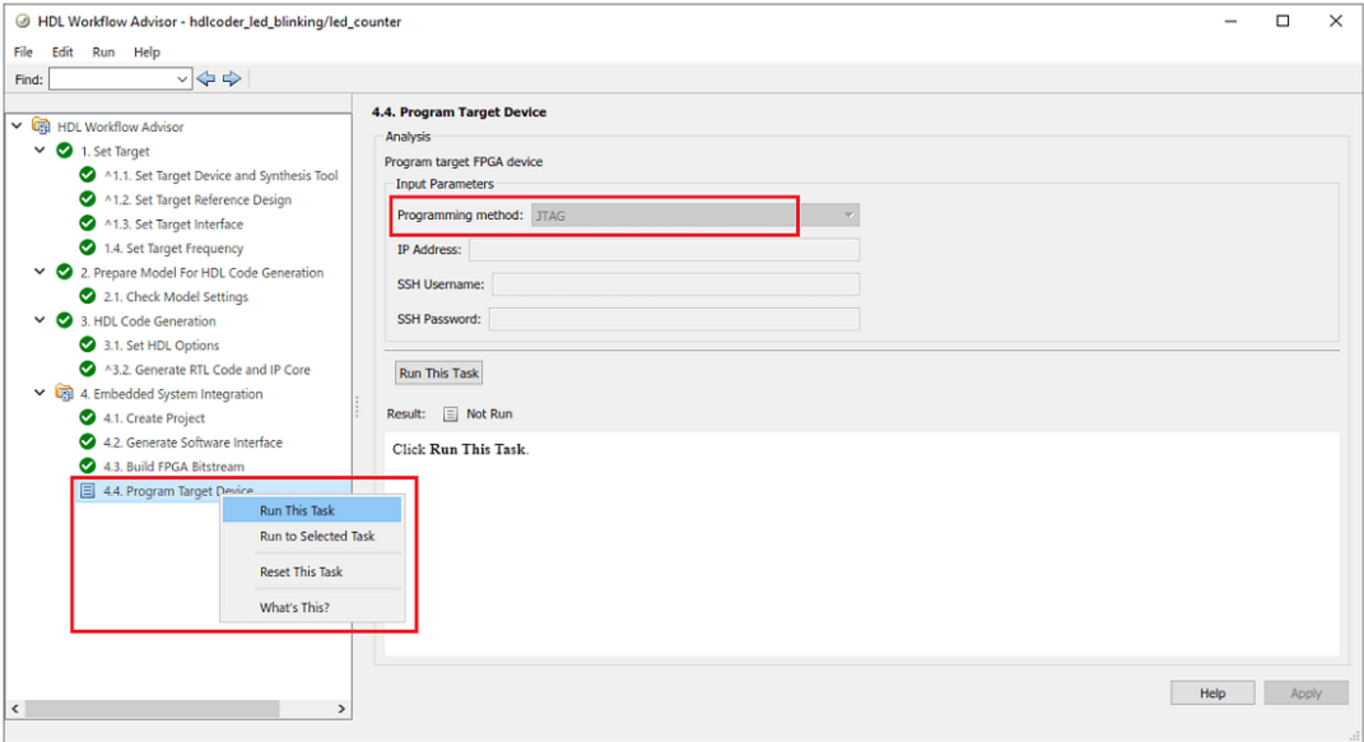
A link to the project is provided in the dialog box. You can optionally open up the project to take a look. From the block diagram in Libero tool, you can see the HDL Coder generated IP core led_count_ip_0 is connected to the Microprocessor Subsystem through the AXI interface.



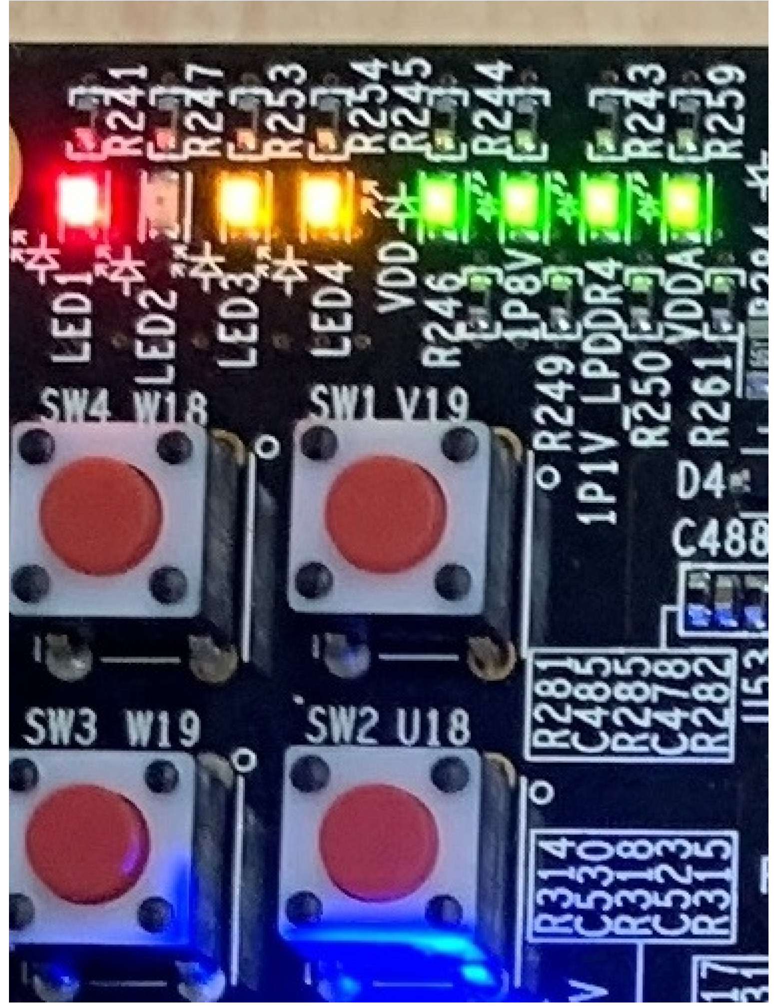
2. Build the FPGA bitstream in the **Build FPGA Bitstream** task. Make sure that you select **Run build process externally** option so that Libero synthesis tool will runs in a separate process from MATLAB. Wait until workflow_task_runExtShell.log file gets generated in libero_ip_prj folder.



3. After the bitstream is generated, select the **Program Target Device** task. **JTAG** option for **Programming method** will be selected automatically to download the FPGA bitstream onto the PolarFire SoC board. Your design will be automatically reloaded when you power cycle the PolarFire SoC board. click **Run This Task** to program the PolarFire SoC hardware.



After you program the FPGA hardware, the LED starts blinking on your PolarFire SoC board.



Read Values of the AXI4 Slave input registers

The AXI4 slave base address is used together with interface mapping address (from IP Core Generation report) to read the value of AXI4 slave input registers. These address mapping details are used to read the AXI4 slave registers. You can perform the readback of input registers in these ways:

- Using the Devmem command

Readback of AXI4 Slave Input Registers Using Devmem Command

Devmem command can be used in Putty or Hyper Terminal. Once the bitstream is programmed into the target device, open Putty or hyper terminal using serial interface. From **Register Address Mapping** , AXI4 Slave Base Address is 0x60000000. You can perform read and write operation on AXI4 Slave registers using Base Address and Address Offset. To use devmem command for writing and reading value of **Blink_frequency_Data** register, use these steps.

1. Read the data from address '60000100':

```
devmem2 0x60000100
```

You get the value as 0x00000000.

2. Write some value in address '60000100':

```
devmem2 0x60000100 w 0x1
```

3. Reread the address '60000100':

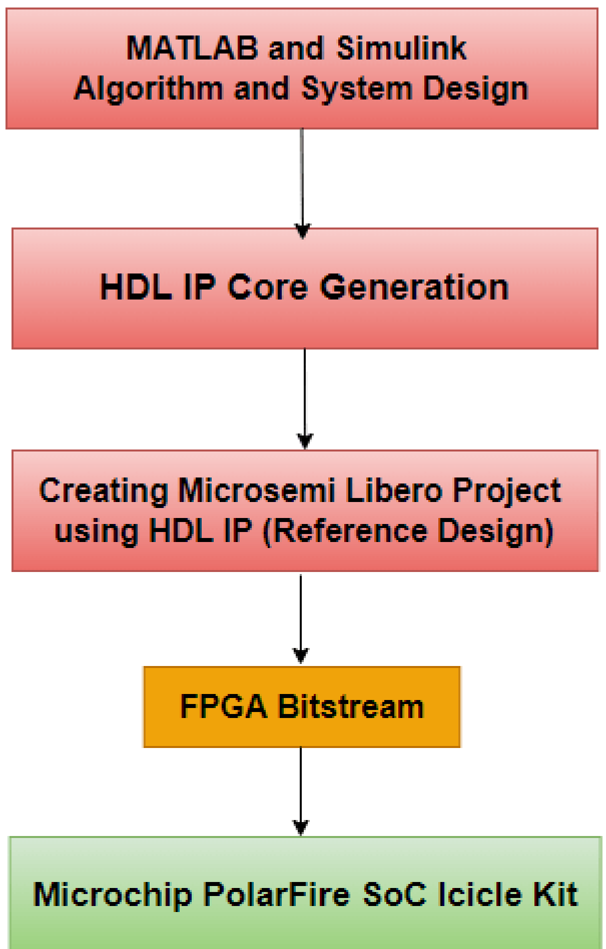
```
devmem2 0x60000100
```

You get the value as 0x00000001.

Summary

This example shows how the hardware and software co-design workflow helps automate the deployment of your MATLAB and Simulink design to a Microchip PolarFire SoC Icicle Kit. You can explore the best ways to partition and deploy your design by iterating through the workflow.

The following diagram shows high-level picture of the workflow you went through in this example. To learn more about the hardware and software co-design workflow, see “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9.


MATLAB and Simulink
Algorithm and System Design
HDL IP Core Generation
Creating Microsemi Libero Project
using HDL IP (Reference Design)
FPGA Bitstream
Microchip PolarFire SoC Icicle Kit

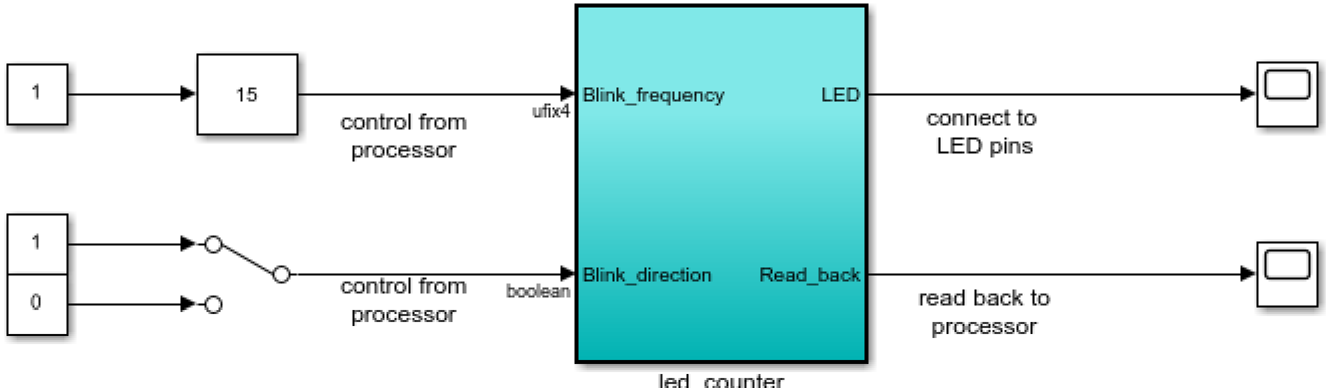
Save Target Hardware Settings in Model

This example shows how to save your target hardware settings in a Simulink® model.

This example also shows different ways that you can export, modify, and import target hardware settings. This example uses the Xilinx Zynq platform, but in the same way, you can save target hardware settings in models that target the Intel SoC devices, and Simulink Real-Time FPGA I/O boards.

Open the Model

```
open_system('hdlcoder_led_blinking');
```



Configure the Target Hardware Settings

When you configure the target hardware settings, you modify the model. If you save the model, the target hardware settings are saved as part of the model.

You can configure the target hardware settings in three ways:

- HDL Workflow Advisor
- HDL Block Properties dialog box for Inport or Outport
- `hdlset_param`

Since the HDL Workflow Advisor provides a dropdown menu for each target hardware option, it is best to use the HDL Workflow Advisor when you configure the target hardware settings for the first time. After you save the model with a valid configuration, you can view, modify, and apply settings from the command line.

Use HDL Workflow Advisor to configure model or port hardware settings

Open the HDL Workflow Advisor to generate an IP core from the subsystem `hdlcoder_led_blinking/led_counter` by using the **Launch HDL Workflow Advisor** button in `hdlcoder_led_blinking`. Specify your target hardware settings in task 1 **Set Target**.

In the **Set Target** > **Set Target Device and Synthesis Tool** task:

- For **Target workflow**, select `IP Core Generation`.
- For **Target platform**, select `Xilinx Zynq ZC702 evaluation kit`.

Right-click task 1.2 **Set Target Reference Design** and select **Run to Selected Task**.

In the **Set Target** > **Set Target Interface** task, map the ports to interfaces as follows:

- For **Blink_frequency** and **Blink_direction** input ports, select the `AXI4-Lite` interface.
- For the **LED** output port, select `External Port`.
- For the **Read_back** output port, select the `AXI4-Lite` interface.

Specify the HDL IP core name and version in task 3.2 **Generate RTL Code and IP Core**. For details, see "Getting Started with Targeting Xilinx Zynq Platform" on page 39-98.

Use HDL Block Properties dialog box to map DUT ports to target interface

You can specify target interface settings for the DUT interface by using the HDL Block Properties dialog box for any Inport or Outport. You can also specify the HDL IP core settings by using the HDL Block Properties dialog box for the DUT subsystem. However, you can use the HDL Block Properties dialog box to configure only the DUT target interface and HDL IP core settings. Set other target hardware settings from the HDL Workflow Advisor, or by using `hdlset_param` at the command line.

For example, you can change the bit range of the **Blink_frequency** Inport to x"120" and remap the **LED** Outport to LEDs General Purpose [0:7]:

1. From the subsystem `hdlcoder_led_blinking/led_counter`, right-click the **Blink_frequency** Inport, and select **HDL Code** > **HDL Block Properties** to open the HDL block properties dialog box. Click the **Target Specification** tab. For **IOInterfaceMapping**, enter `x"120"`.
2. Open the HDL block properties dialog for the **LED** Outport. Click the **Target Specification** tab. For **IOInterface**, enter `LEDs General Purpose [0:7]`.

Navigate to the model level `hdlcoder_led_blinking` and right-click the subsystem `hdlcoder_led_blinking/led_counter`. Select **HDL Code** > **HDL Block Properties**. In the **Target Specification** tab, you can change IP core name and version by using the **IPCoreName** and **IPCoreVersion** parameters respectively.

The target interface and HDL IP core settings you specify using the HDL Block Properties dialog box are validated when you open the HDL Workflow Advisor.

Use hdlset_param to configure model or DUT port hardware settings

To configure target hardware settings for your model or DUT ports, you can use `hdlset_param`.

For example, to change the **TargetPlatform** to Xilinx Zynq ZC706 evaluation kit, enter:

```
hdlset_param('hdlcoder_led_blinking', 'TargetPlatform', 'Xilinx Zynq ZC706 evaluation kit');
```

To set the Bit Range of **Blink_frequency** Inport to `x"120"`; and set the **LED** Outport to `LEDs General Purpose [0:7]`, enter:

```
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', 'IOInterfaceMapping', 'x"120"')
hdlset_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface', 'LEDs General Purpose [0:7]
```

To set the IP core name and version, enter:

```
hdlset_param('hdlcoder_led_blinking/led_counter', 'IPCoreName', 'my_ipcore');
hdlset_param('hdlcoder_led_blinking/led_counter', 'IPCoreVersion', '2.0');
```

Export and Import Target Hardware Settings

To export all non-default HDL code generation options in your model, including the target hardware settings, you can use **hdlsaveparams** and **hdlrestoreparams**. You can modify the model settings in the saved MATLAB file, and apply the settings to the same model or to a different model.

For example, to export the settings from the hdlcoder_led_blinking model to a MATLAB file, `targetSetting.m`, enter:

```
hdlsaveparams('hdlcoder_led_blinking/led_counter','targetSetting.m')
```

You can modify the settings in `targetSetting.m` as desired, then enter the following command to apply the settings to the model:

```
hdlrestoreparams('hdlcoder_led_blinking/led_counter','targetSetting.m')
```

Save and Reopen the Model

1. Save the model `hdlcoder_led_blinking` as `hdlcoder_led_blinking_saved`.
2. Open the saved model, `hdlcoder_led_blinking_saved`.
3. Open the HDL Workflow Advisor by using the **Launch HDL Workflow Advisor** button in `hdlcoder_led_blinking_saved`.

Notice that the modified settings are automatically loaded to task 1 **Set Target** in the HDL Workflow Advisor.

Generate IP Core from MATLAB for Blinking LEDs on FPGA Board

This example shows how to use the MATLAB® HDL Workflow Advisor to generate a customized IP core that blinks LEDs on an FPGA board. This example deploys the generated IP core on the ZedBoard hardware, but you can use any Xilinx® Zynq® platform or Xilinx FPGA with a MicroBlaze processor.

Introduction

You can use MATLAB to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your application design for hardware and software implementation on your FPGA board and decide which elements are performed by the programmable logic and which elements run on the processor.

In this example, you can:

1. Set up your Zynq hardware and tools.
2. Review your design for hardware and software implementation.
3. Convert your MATLAB algorithm to HDL code using HDL Coder.
4. Generate an HDL IP core by using the MATLAB HDL Workflow Advisor.
5. Deploy the IP core to Zynq hardware.
6. Prototype the Zynq hardware using included MATLAB scripts.

Additionally, you can use this IP core in other designs or projects to enhance and build upon its functionality.

Prerequisites

- Install Xilinx Vivado®. For a list of supported versions, see “HDL Language Support and Supported Third-Party Tools and Hardware”.
- Set the path to the installed synthesis tool for the target device by using the `hdlsetuptoolpath` function. In the MATLAB Command Window, use this command, making sure to replace the path with your installation path.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

- Set up the ZedBoard. Refer to the "Set up Zynq Hardware and Tools" section in “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98 for more information.
- In this example, you implement the `mlhdlc_ip_core_led_blinking` function on the ZedBoard. Open and save a local copy of the function:

```
open mlhdlc_ip_core_led_blinking.m
```

- Open and save a local copy of the MATLAB test bench.

```
open mlhdlc_ip_core_led_blinking_tb.m
```

Review and Partition Design for Hardware and Software Implementation

To generate and deploy an IP core from a MATLAB algorithm, you need three components:

1. A MATLAB function that runs on the hardware.
2. A MATLAB test bench that exercises the function design in MATLAB.
3. MATLAB scripts that prototype the deployed MATLAB function on the hardware. Prototyping using MATLAB is optional, but provides an additional means of IP core verification.

To effectively implement your design, you must:

- Distinguish between the parts of the algorithm that are suitable for programmable logic and the parts that are suitable for the ARM processor.
- Group the algorithm parts for the programmable logic into a MATLAB function.
- Use the MATLAB function as the boundary for your hardware-software partition.
- Use HDL Coder to implement the MATLAB code in this function on programmable logic.

Open the `mlhdlc_ip_core_led_blinking` function to examine how it is partitioned.

```
open mlhdlc_ip_core_led_blinking.m
```

This function models a counter that blinks LEDs on an FPGA board. It has two inputs, `Blink_frequency` and `Blink_direction`, which determine the LED blink frequency and direction. The output, `LED`, connects to the LED hardware found on the board and the output, `Read_back` is set to the count value and can be read back to the processor.

Open the `mlhdlc_ip_core_led_blinking_tb` test bench to examine how it exercises the function design.

```
open mlhdlc_ip_core_led_blinking_tb.m
```

The test bench provides a range of input conditions to the `mlhdlc_ip_core_led_blinking` function.

Open the included MATLAB scripts `matlab_hdlcoder_led_blinking_interface.m` and `matlab_hdlcoder_led_blinking_setup.m`.

```
open matlab_hdlcoder_led_blinking_interface.m
open matlab_hdlcoder_led_blinking_setup.m
```

These scripts set up and interface with the generated and deployed IP core by writing to the AXI accessible registers.

Create MATLAB HDL Coder Project

Create an HDL Coder project by entering this command in the MATLAB Command Window.

```
coder -hdlcoder -new blinkingLEDmatlab
```

For a more comprehensive tutorial on creating and populating MATLAB HDL Coder projects, see “Get Started with MATLAB to HDL Workflow”.

Alternatively, in the **Apps** tab, click **HDL Coder**. In the **HDL Code Generation** pane:

1. Set the **MATLAB Function** to the locally saved MATLAB function, `mlhdlc_ip_core_led_blinking`.
2. Set the **MATLAB Test Bench** to the locally saved MATLAB test bench, `mlhdlc_ip_core_led_blinking_tb`.

3. In the **MATLAB Function** section, click **Autodefine types** to define the input types automatically.
4. Click **Workflow Advisor**.

Using the **Workflow Advisor**, perform a fixed-point conversion:

1. In the **Fixed-Point Conversion** task, click **Advanced** and ensure **Safety margin for sim min/max (%)** is 0.
2. In the bottom pane, in the **Variables** tab, in the **Proposed Type** column, set the proposed type of the `freqCounter` variable to unsigned 24-bit integer by entering `numerictype(0, 24, 0)`.
3. Right-click the **Fixed-Point Conversion** task and select **Run This Task**.

For an overview on fixed-point conversion, see "Floating-Point to Fixed-Point Conversion" on page 4-51.

Define Code Generation Target

You can generate a shareable and reusable IP core using the MATLAB HDL Workflow Advisor. HDL Coder integration with the Xilinx Vivado IDE facilitates the incorporation of the generated IP core into the FPGA design.

For an overview of how to generate an IP core for a specific hardware platform, see "Targeting FPGA & SoC Hardware Overview" on page 39-3.

Perform IP Core Generation using the MATLAB HDL Workflow Advisor:

1. In the MATLAB HDL Workflow Advisor, click **Select Code Generation Target**.
2. Set **Workflow** to `IP Core Generation`.

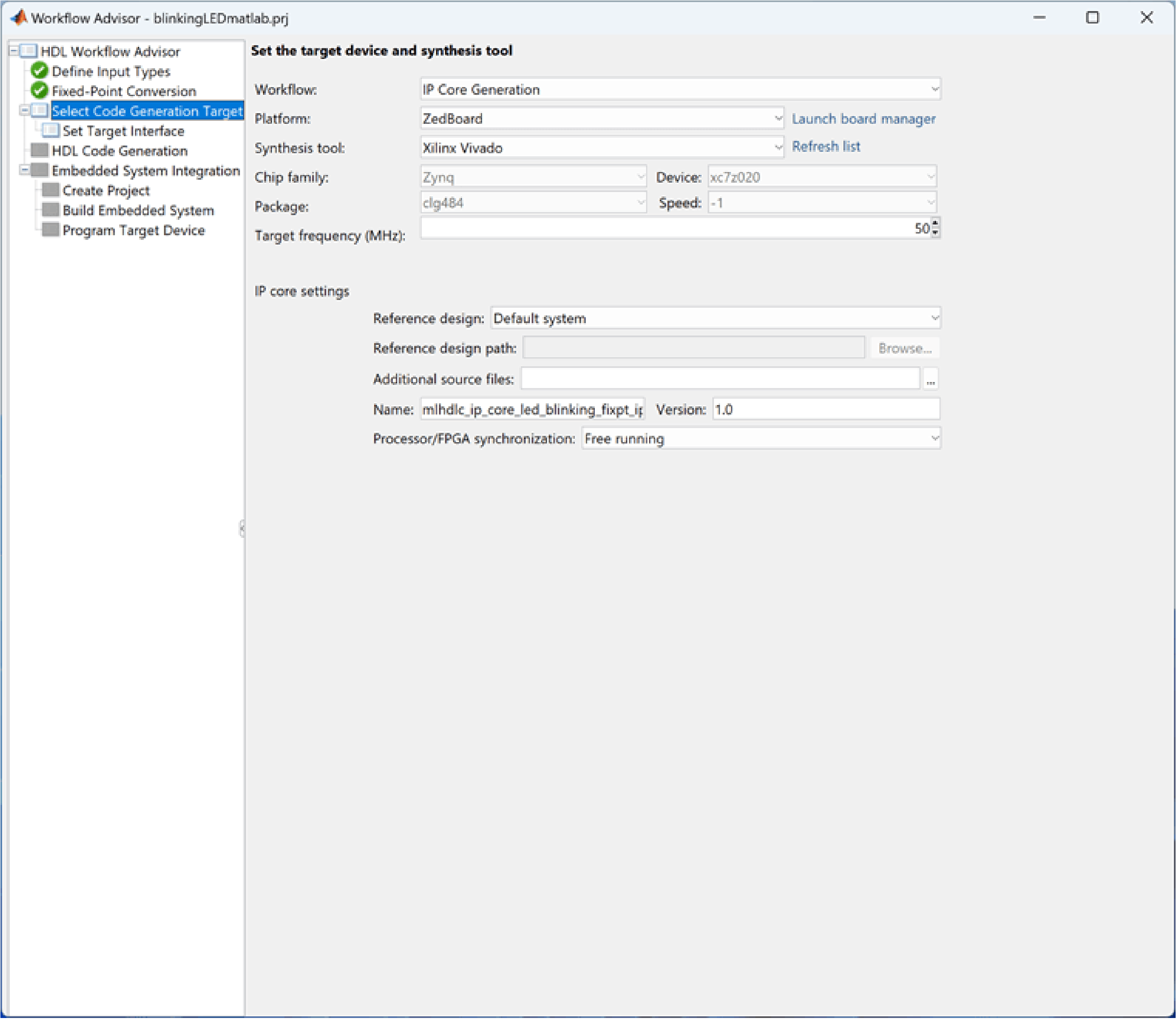
Specify Platform

In order to integrate the generated IP core in a reference design, set **Platform** to a target platform:

- Generic Xilinx Platform: This is a board-independent option that generates a generic Xilinx IP core. However, you must manually integrate the IP core into an existing Xilinx Vivado project.
- Board-specific platforms: These options are specific to Xilinx boards and integrate the IP core into a predefined reference design. This integration uses the Xilinx Vivado with IP Integrator embedded system tool.

In this example, you target a ZedBoard platform. Set these parameters:

1. Set **Platform** to `ZedBoard`.
2. Set **Synthesis tool** to `Xilinx Vivado`.
3. Set **Target Frequency (MHz)** to `50`.
4. Under **IP core settings**, set **Reference design** to `Default system`.



Configure Target Interfaces

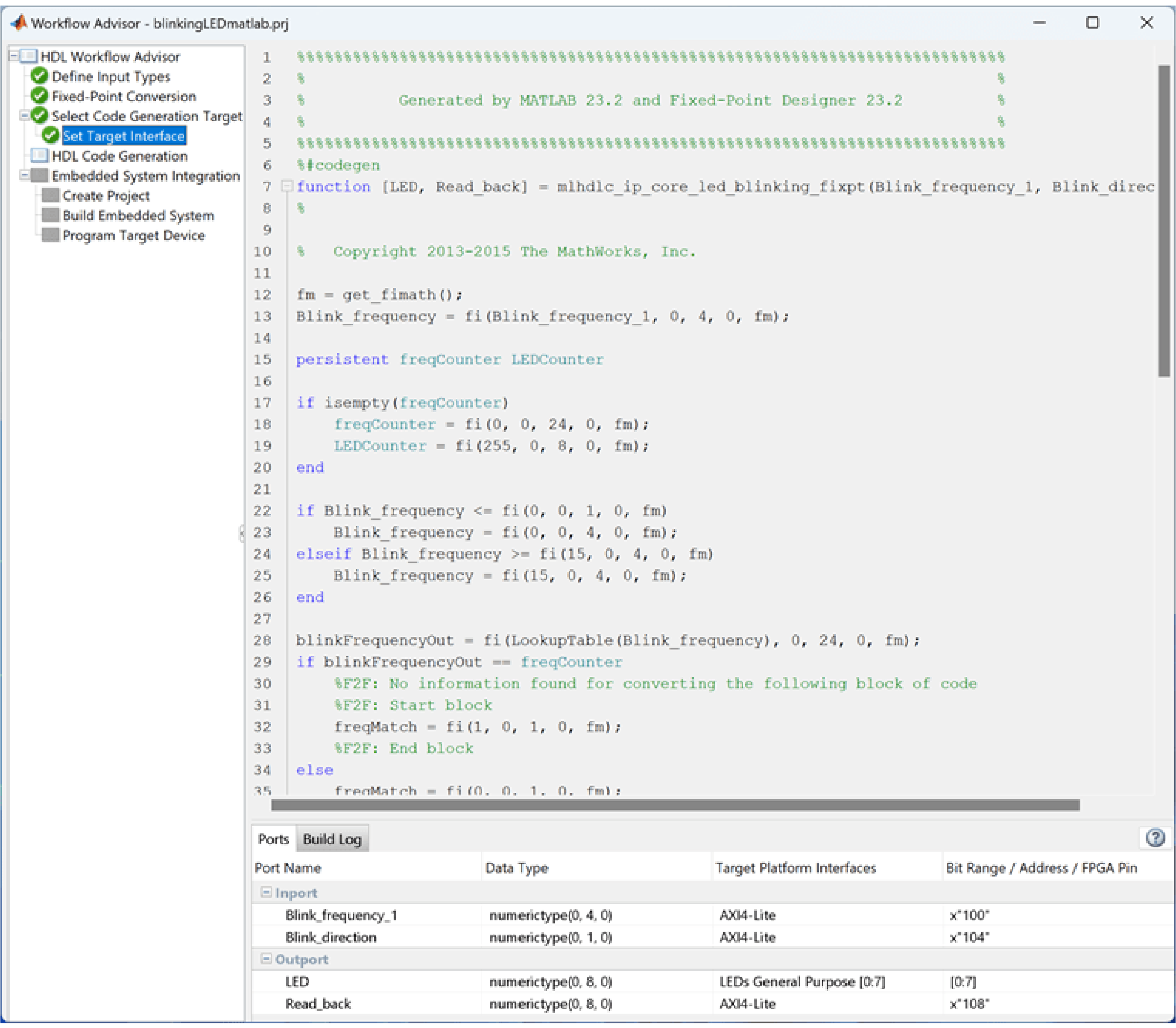
Next, you use the **Set Target Interface** subtask to map each input and output in the MATLAB design function to one of the IP core target interfaces.

In this example, map inputs `Blink_frequency` and `Blink_direction`, along with the output `Read_back` to the `AXI4-Lite` interface. HDL Coder generates AXI accessible registers for them. Map the `LED` output to an external interface, `LEDs General Purpose [0:7]`, which connects to the LED hardware on the ZedBoard.

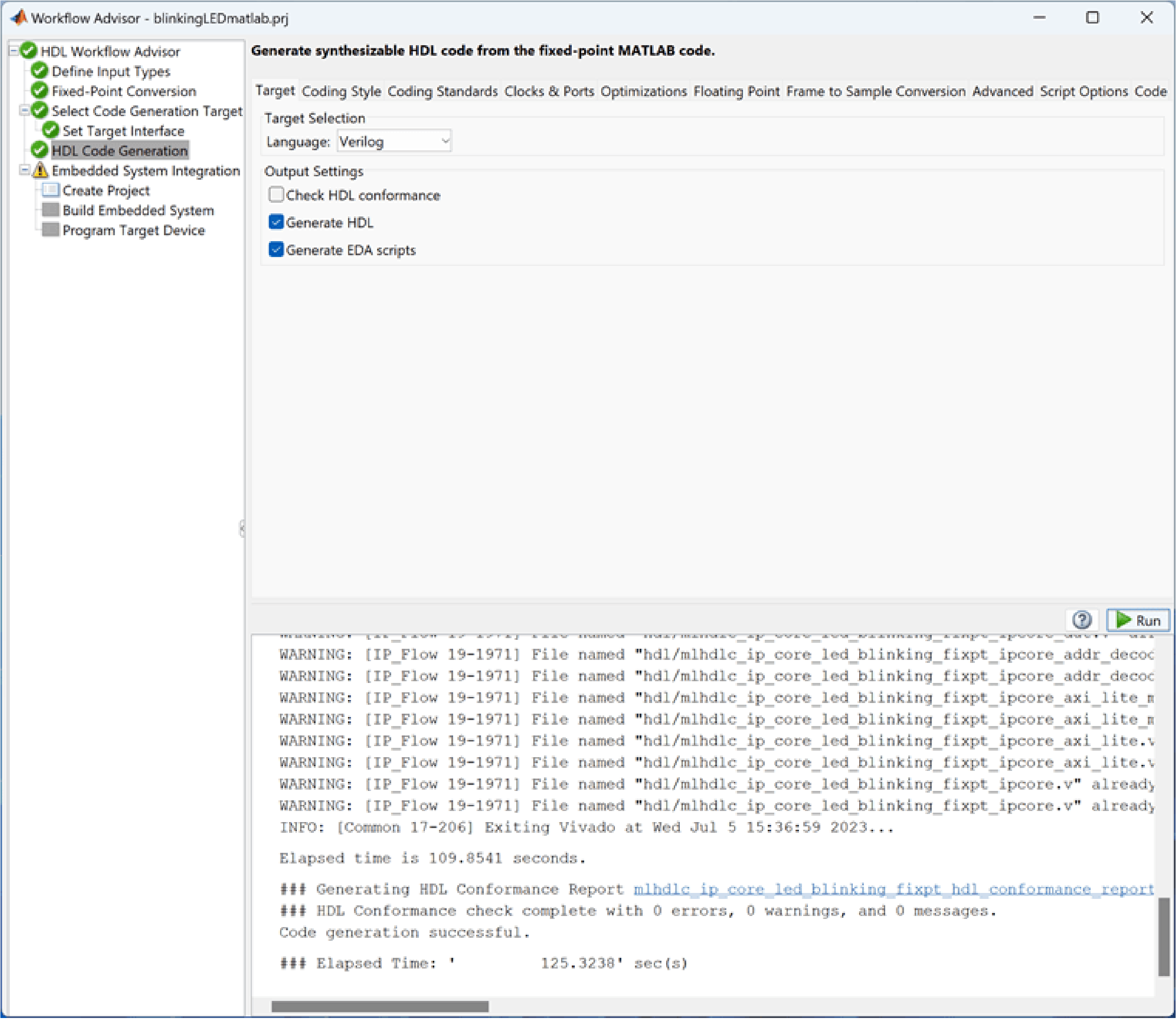
1 In the bottom pane, in the **Ports** tab, in the **Target Platform Interfaces** column, set the target platform interface for the `Blink_frequency`, `Blink_direction`, and `Read_back` variables to the `AXI4-Lite` interface.

2 In the bottom pane, in the **Ports** tab, in the **Target Platform Interfaces** column, set the target platform interface for the `LED` variable to the `LEDs General Purpose [0:7]` external interface.

3 Right-click the **Set Target Interface** subtask and select **Run to Selected Task**.



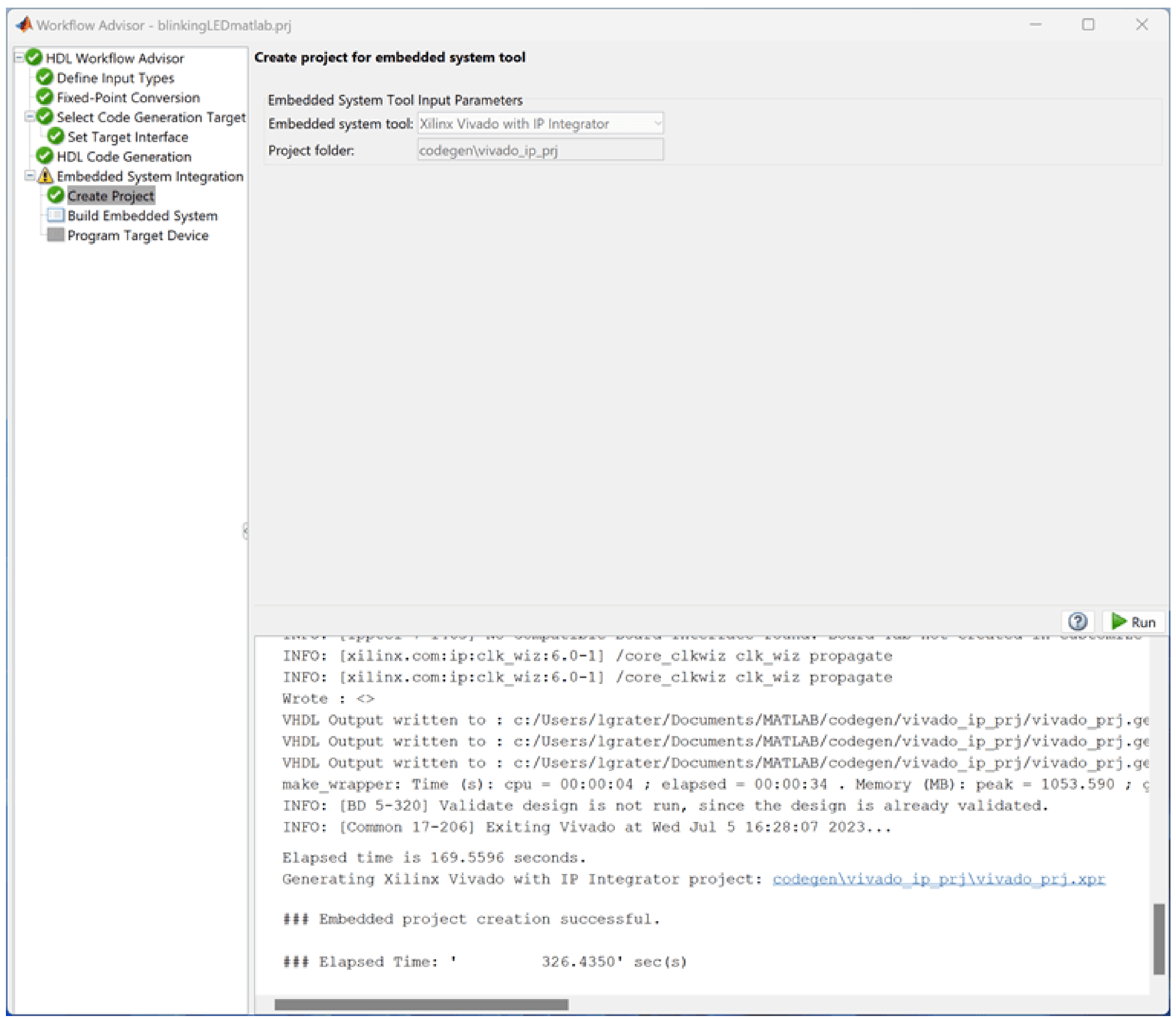
To generate the IP core and IP core report, right-click the **HDL Code Generation** task and select **Run this task**.



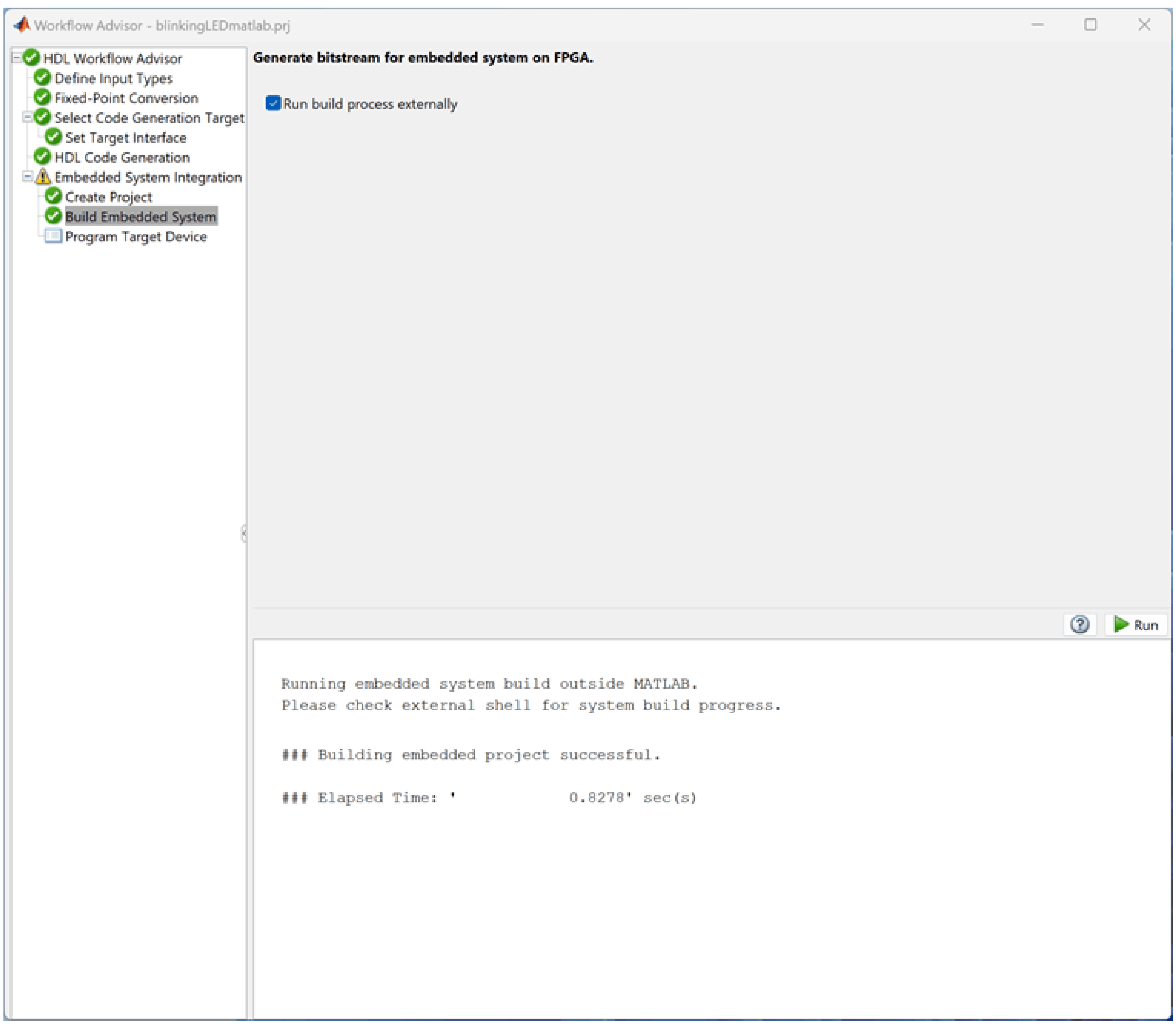
Integrate Generated IP Core with Xilinx Vivado Environment

After generating the IP core, you integrate it with a reference design, then synthesize and download the bitstream to the FPGA within the MATLAB HDL Workflow Advisor.

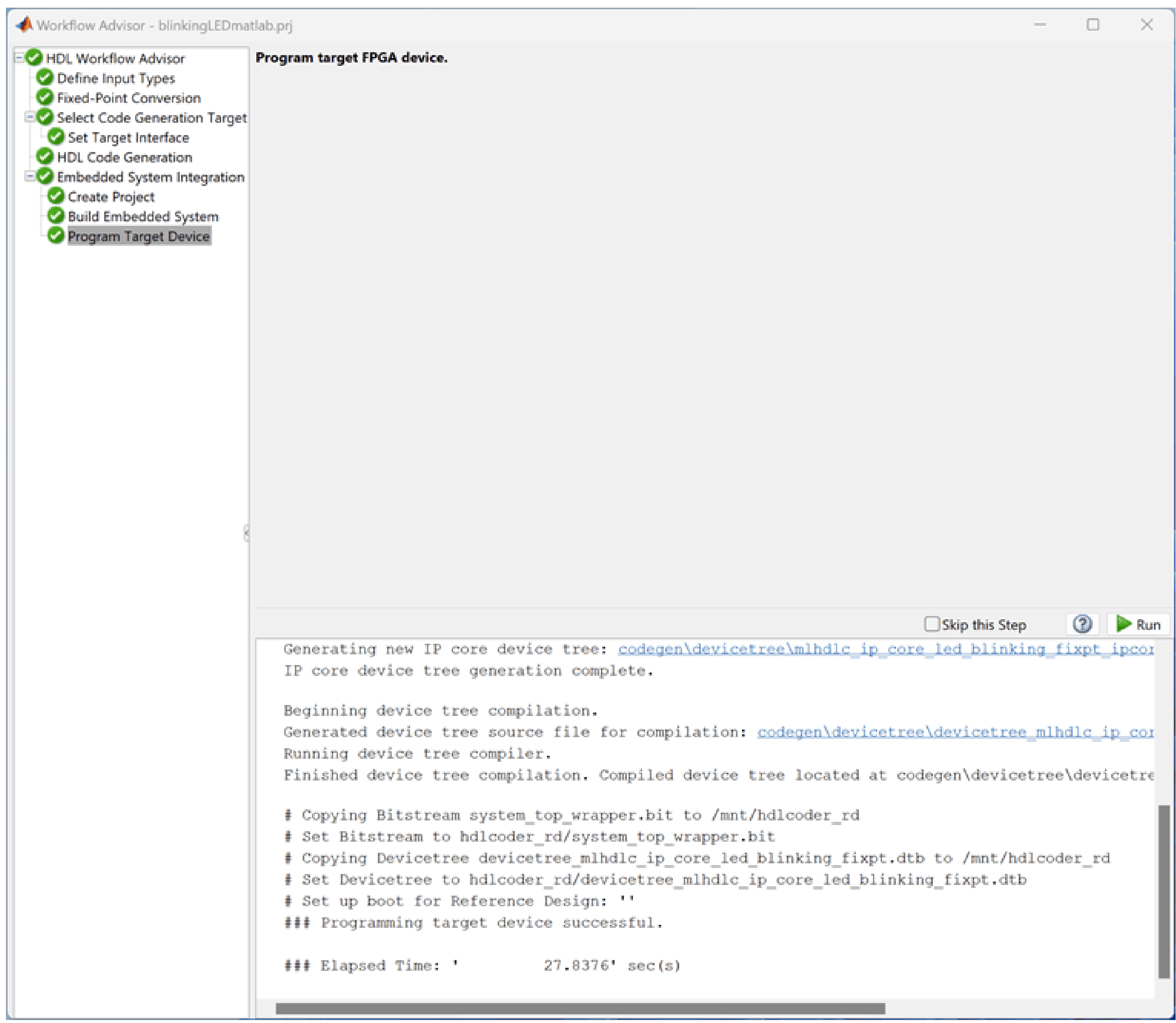
1. Under the **Embedded System Integration** task, right-click the **Create Project** subtask and select **Run This Task**. A Xilinx Vivado project is generated that integrates the IP core in a predefined reference design by leveraging the IP Integrator embedded system tool. The dialog box includes a link to open the project in Vivado.



2. In the **Build Embedded System** subtask, select **Run build process externally**, then click **Run**. The Xilinx synthesis tool runs as a separate process outside of MATLAB. Wait for the synthesis tool to complete.



3. After generating the bitstream, right-click the **Program Target Device** subtask and select **Run This Task** to program the ZedBoard.



4. After programming the FPGA hardware, the LEDs start blinking on the ZedBoard.

Prototype and Verify IP Core on Hardware

To verify your generated IP core on the ZedBoard hardware, use the included handwritten MATLAB files, `matlab_hdlcoder_led_blinking_interface.m` and `matlab_hdlcoder_led_blinking_setup.m`. These files contain MATLAB commands that connect your hardware and interact with your IP core.

Open the interface and setup files and save local copies:

```
open matlab_hdlcoder_led_blinking_interface.m
open matlab_hdlcoder_led_blinking_setup.m
```

The `matlab_hdlcoder_led_blinking_interface.m` script establishes a connection to your FPGA hardware and uses the `matlab_hdlcoder_led_blinking_setup` function to enable data reading and writing by configuring the `fpga` hardware object with the mapped ports and interfaces from the **Set Target Interface** subtask.

You can reuse this function in your own scripts to replicate the same configuration. For example, in the `matlab_hdlcoder_led_blinking_interface.m` script uncomment and modify this line to change the LED blink frequency.

```
% writePort(testFPGA, "Blink_frequency_1", 0);
```

Run the modified script and observe the LED blink frequency changing on the hardware. Additionally, in the `matlab_hdlcoder_led_blinking_interface.m` script uncomment this line and run the script to observe the changes in the `data_Read_back` variable in the MATLAB workspace.

```
% data_Read_back = readPort(testFPGA, "Read_back");
```

See Also

More About

- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Generate and Manage FPGA I/O Host Interface Scripts” on page 39-68

Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow

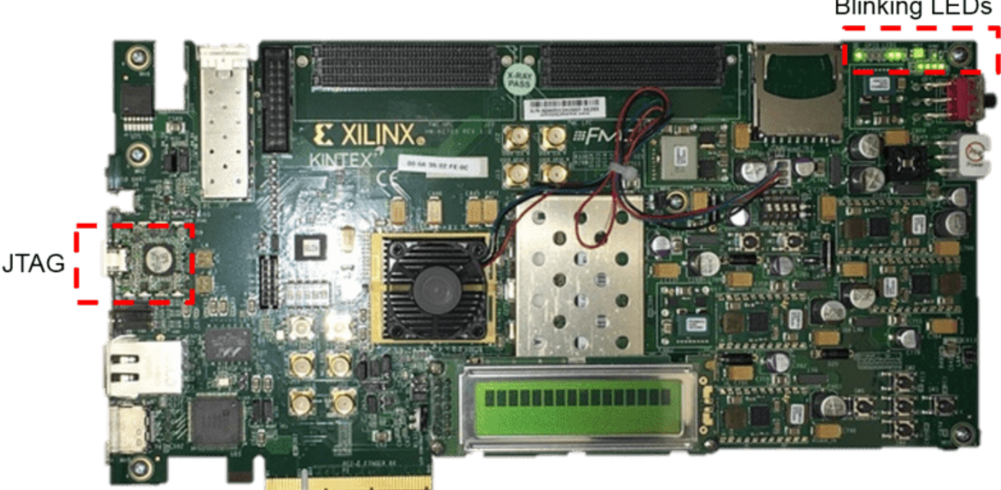
This example shows how to use the HDL Coder™ IP core generation workflow to develop reference designs for Xilinx® parts that do not use an embedded ARM® processor present but that still utilize the HDL Coder generated AXI interface to control the design under test (DUT). This example uses the HDL Verifier™ AXI Manager IP to access the HDL Coder generated DUT registers from MATLAB®. Alternatively, you can use the Xilinx JTAG AXI Master to access the DUT registers using Vivado® Tcl console by writing Tcl commands. For the Xilinx JTAG AXI Master, you must create a custom reference design. The FPGA design is implemented on the Xilinx Kintex®-7 KC705 board.

Requirements

- Xilinx Vivado Design Suite, with a supported version listed in "HDL Language Support and Supported Third-Party Tools and Hardware"
- Xilinx Kintex-7 KC705 development board
- HDL Coder Support Package for Xilinx FPGA and SoC Devices
- HDL Verifier Support Package for Xilinx FPGA Boards

Xilinx Kintex-7 KC705 Development Board

This figure shows the Xilinx Kintex-7 KC705 development board.



Example Reference Designs

Designs that can benefit from using the HDL Coder IP core generation workflow without using either an embedded ARM processor or an Embedded Coder™ support package but still leverage the HDL Coder generated AXI4-Lite registers can include one of these IP sets.

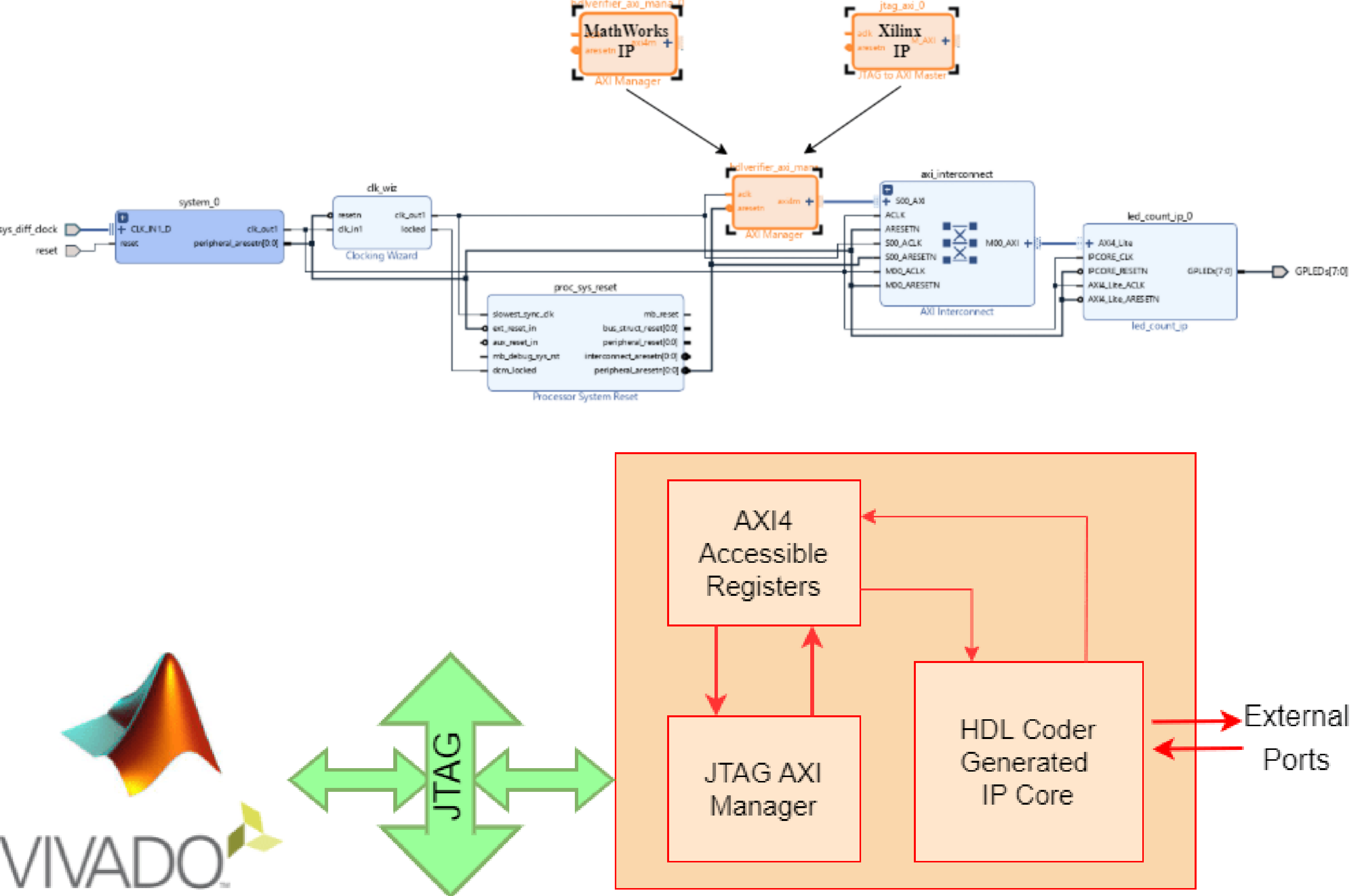
- HDL Verifier AXI Manager + HDL Coder IP Core
- Xilinx JTAG Master + HDL Coder IP Core
- MicroBlaze™ + HDL Coder IP Core

- PCIe Endpoint + HDL Coder IP Core

This example includes two reference designs.

- The Default System reference design uses MathWorks® IP and a MATLAB command line interface for issuing read and write commands. To use this design, you must have the HDL Verifier product.
- The Xilinx JTAG to AXI Master reference design uses Vivado IP for the JTAG to AXI Master and requires using the Vivado Tcl console to issue read and write commands.

The two reference designs differ by only the JTAG manager IP that they use, as this figure shows.



HDL Verifier AXI Manager Reference Design

In the IP core generation workflow of the HDL Workflow Advisor, in the **Set Target Reference Design** step, set the **Insert AXI Manager (HDL Verifier required)** parameter to an interface that communicates between your host machine and the target hardware. This option adds AXI manager IP for your interface automatically into the reference design and connects the added IP to the DUT IP using the AXI4-slave interface. The next section details the steps to auto-insert the JTAG AXI Manager IP in the reference design.

Execute IP Core Workflow

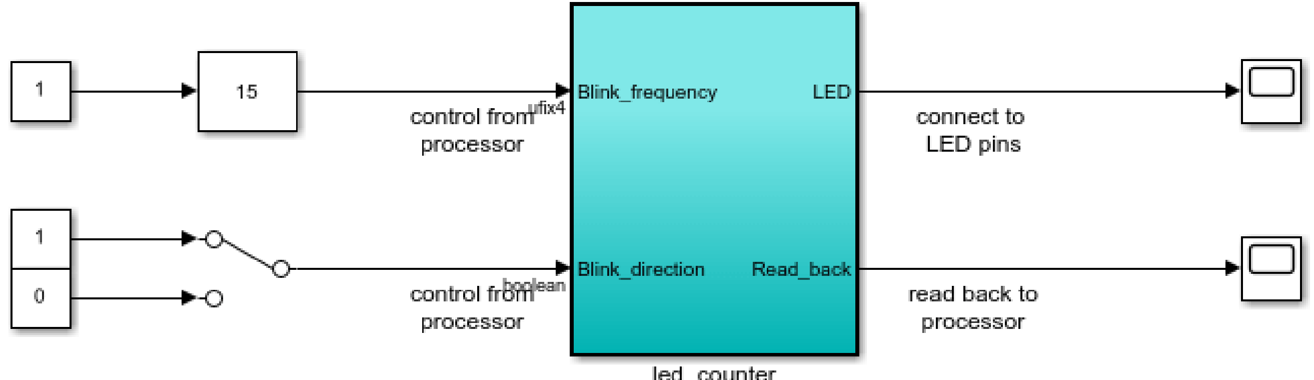
Follow these steps to execute the IP core workflow for the Default System reference design, which uses JTAG AXI Manager IP. Using this reference design, you can generate an HDL IP core that blinks LEDs on the KC705 board. To generate an HDL IP core, follow these steps.

1. Set up the Xilinx Vivado tool path by executing this command in MATLAB. Use your own Xilinx Vivado installation path when executing the command.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath', ...
                 'C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

2. Open the Simulink model that implements LED blinking by executing this command in MATLAB.

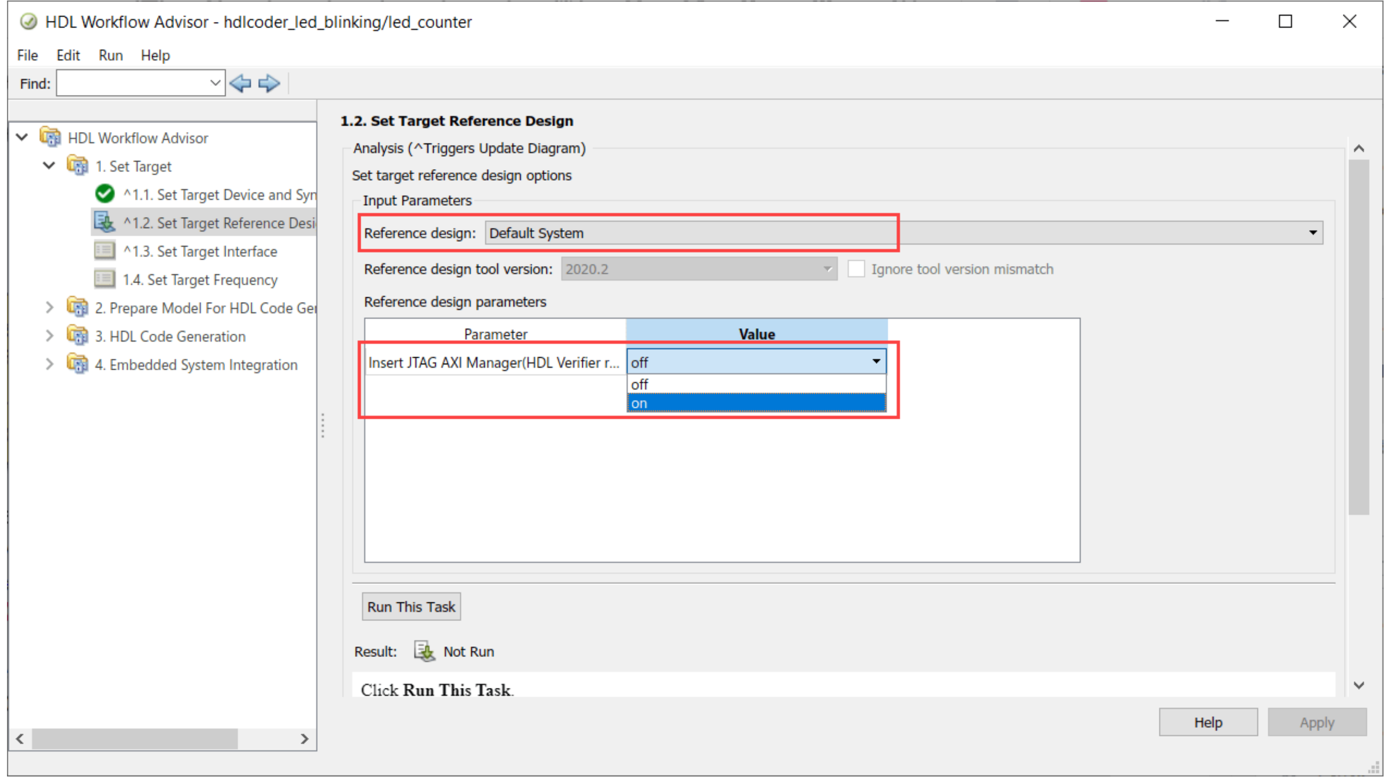
```
open_system('hdlcoder_led_blinking')
```



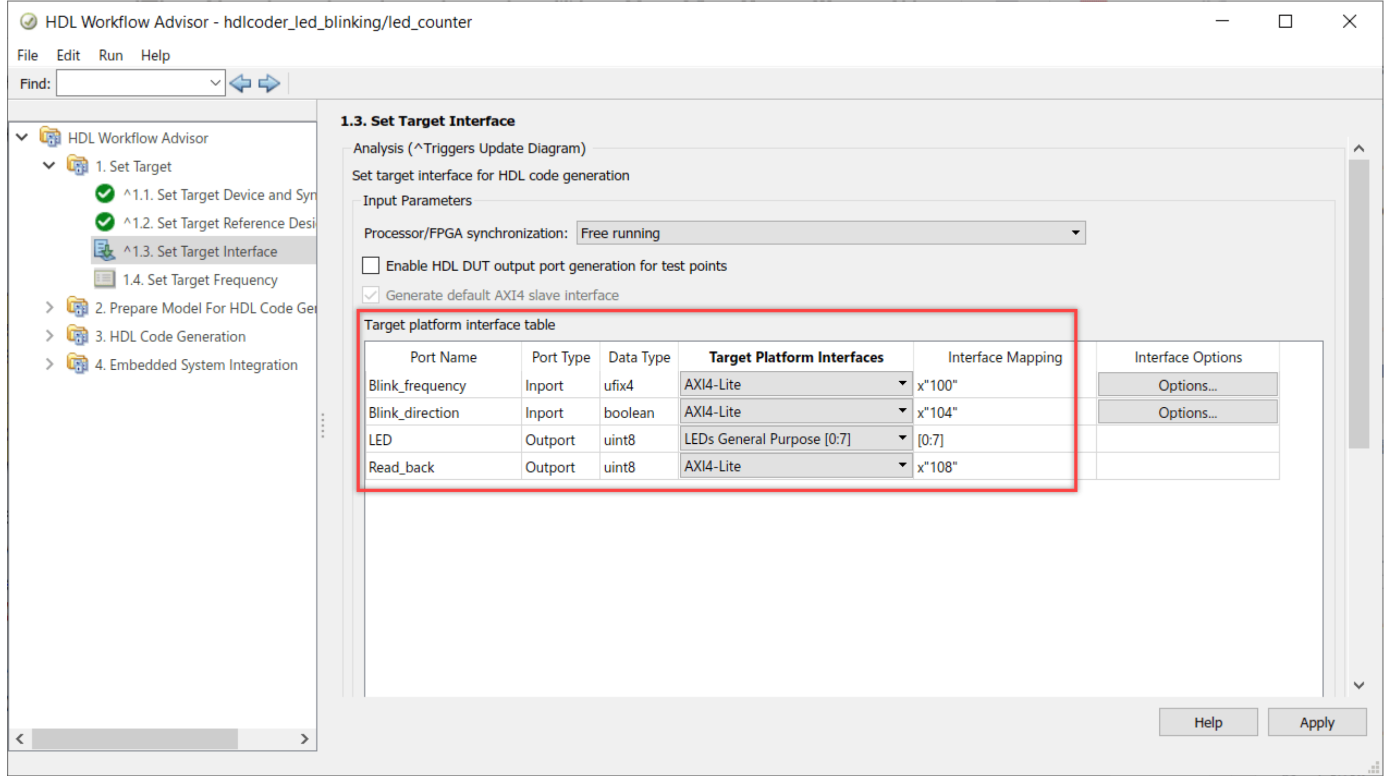
3. Launch HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem and selecting **HDL Code** followed by **HDL Workflow Advisor**.

4. In step 1.1, set **Target workflow** to `IP Core Generation` and **Target platform** to `Xilinx Kintex-7 KC705 development board`. Click **Run This Task**.

5. In step 1.2, set **Reference design** to `Default System`. Under **Reference design parameters**, set **Insert AXI Manager (HDL Verifier required)** to `JTAG`. Click **Run This Task**.

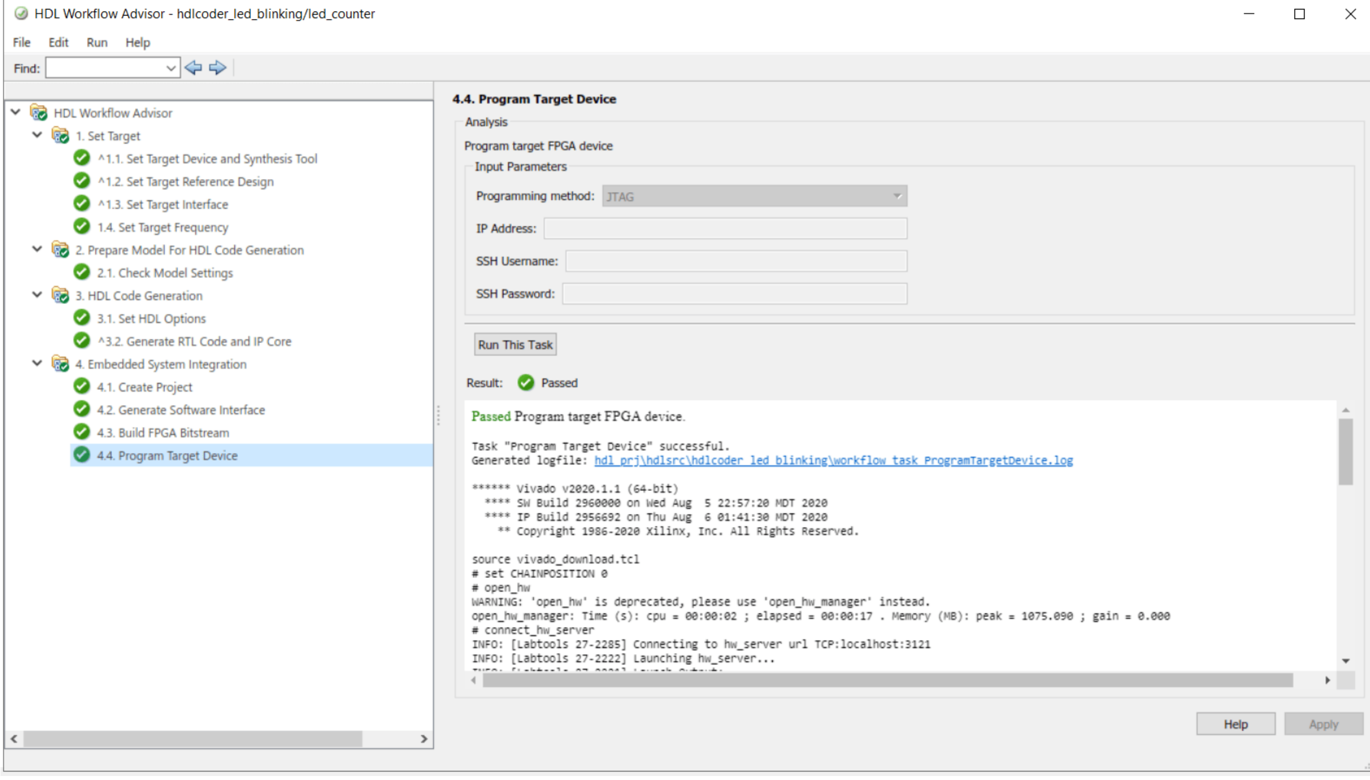


6. In step 1.3, set the interface of the **Blink_frequency**, **Blink_direction**, and **Read_back** ports to `AXI4-Lite`. Set the interface of the **LED** port to `LEDs General Purpose [0:7]`.



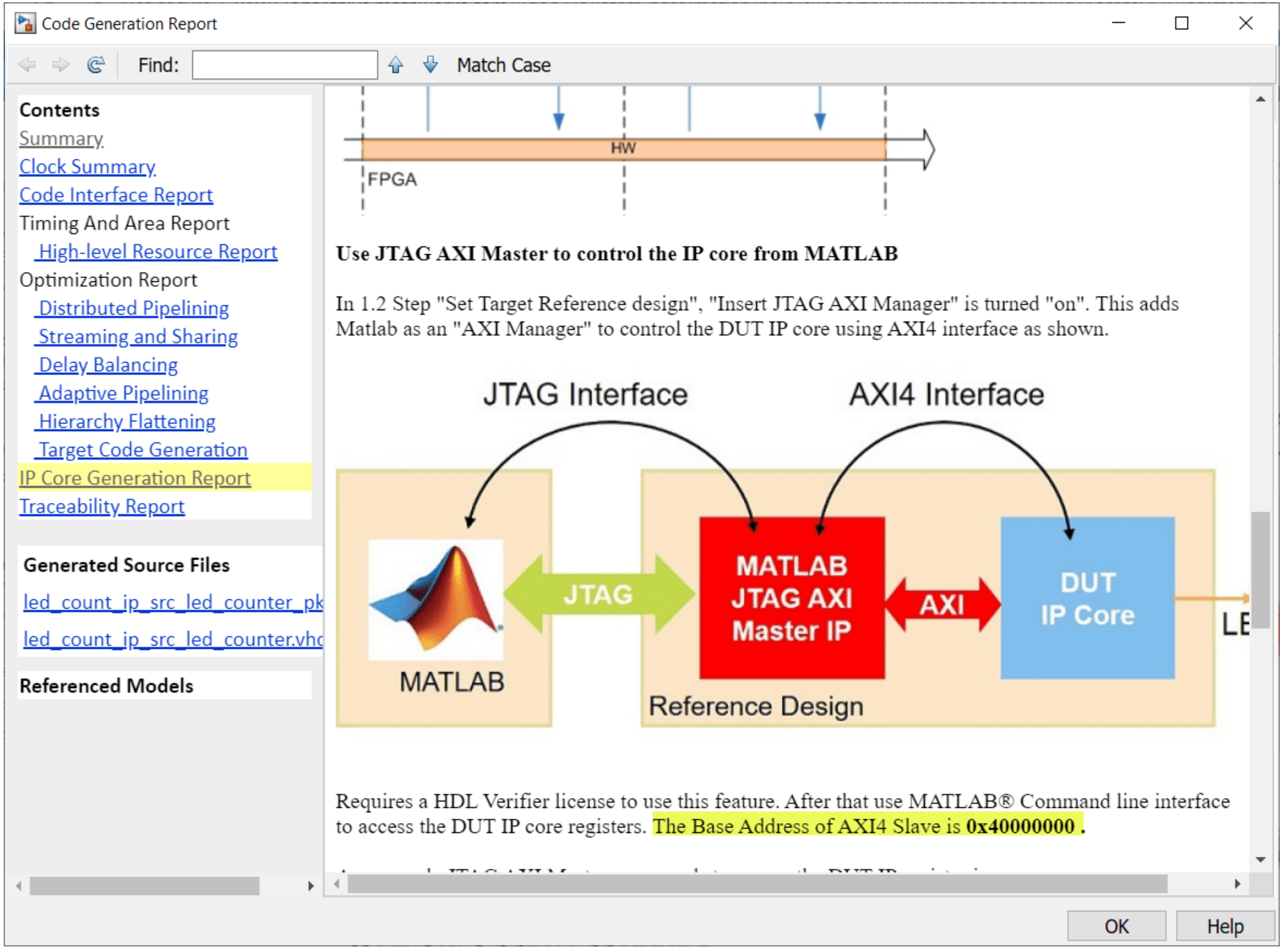
7. Run the remaining steps in the workflow to generate a bitstream and program the target device.

Unlike the Zynq-based reference design, a **Generate Software Interface Model** task does not exist, as this figure shows.

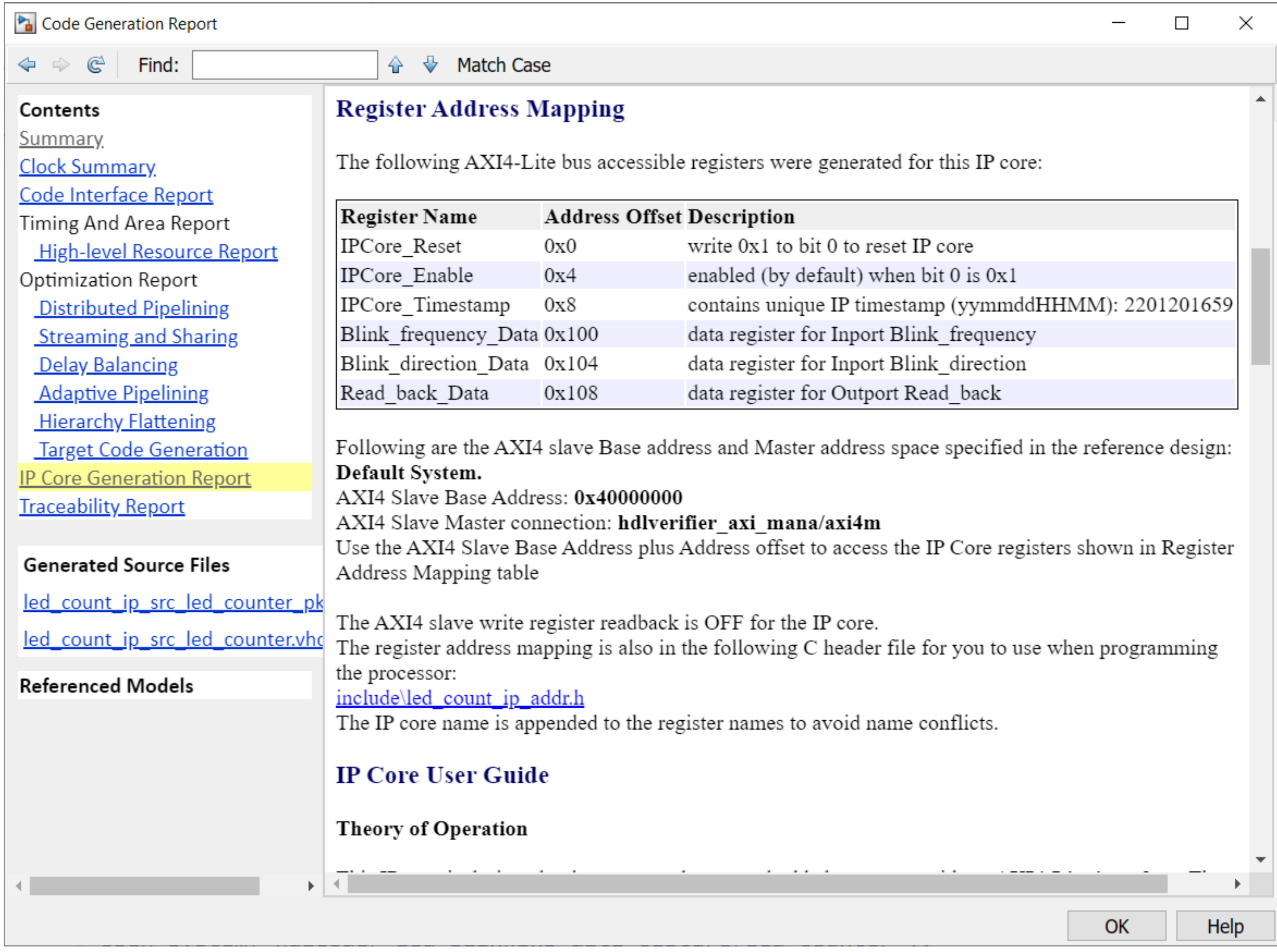


Determine Addresses from IP Core Report

The base address for an HDL Coder IP core is defined as `0x40000000` for the Default System reference design, which uses the AXI Manager IP. You can see address setting in the generated IP core report as shown in this figure.

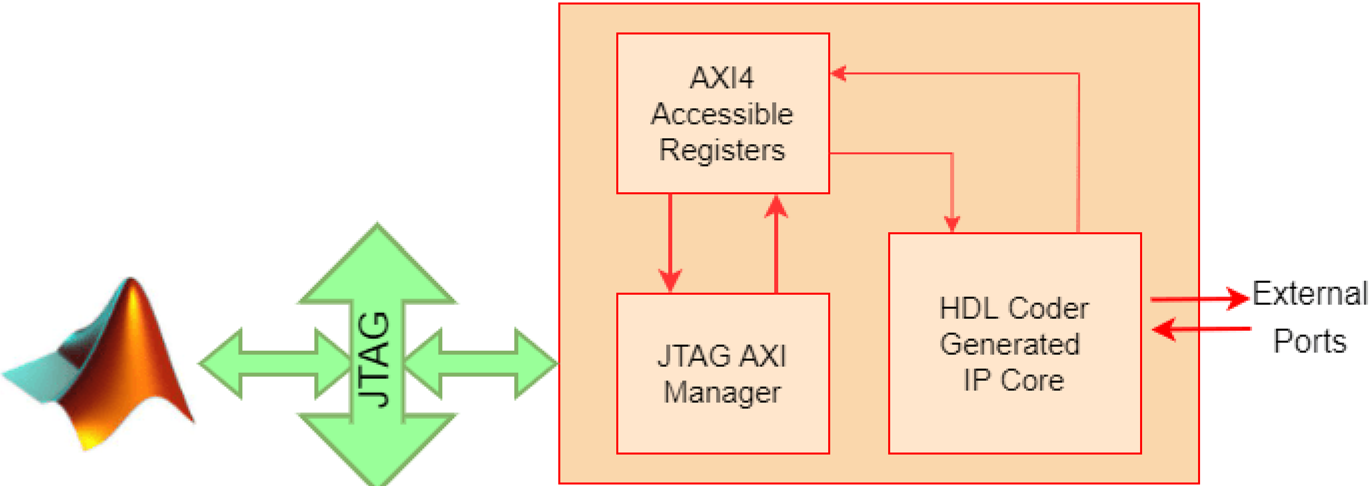


The IP core report register address mapping table shows the offsets.



HDL Verifier Command Line Interface

If you have the HDL Verifier support package for Xilinx FPGA boards, select the AXI Manager reference design, then you can use MATLAB command line interface to access the IP core that is generated by the HDL Coder product.



To write and read from the DDR memory, follow these steps.

1. Create an AXI manager object.

```
h = aximanager('Xilinx')
```

2. Issue a write command. For example, disable the DUT.

```
h.writememory('40000004',0)
```

3. Re-enable the DUT.

```
h.writememory('40000004',1)
```

4. Issue a read command. For example, read the current counter value.

```
h.readmemory('40000108',1)
```

5. Delete the object to free up the JTAG resource. If you do not delete the object, other JTAG operations, such as programming the FPGA, fail.

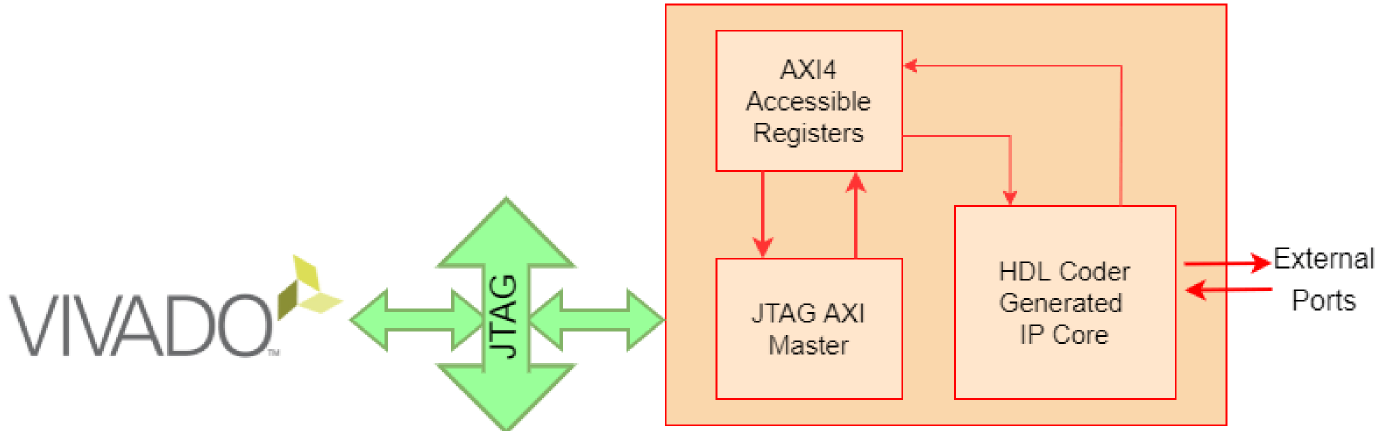
```
delete(h)
```

Xilinx JTAG to AXI Master Reference Design

Create a custom reference design to use the Xilinx JTAG to AXI Master IP in the reference design, and then add the reference design files to the MATLAB path using the `addpath` command.

Access the HDL Coder IP core registers using the Xilinx JTAG to AXI Master IP by using the base address that is defined in reference design plugin file.

Vivado Tcl Commands for AXI Read and Write



This example uses the standalone Vivado Tcl console for basic commands to issue reads and writes. You can use these commands to open the JTAG device and set up an "enable" and "disable" write to the DUT. You can enter these commands directly into the Vivado Tcl console or save them in a Tcl file and source them later. For simplicity, copy these Tcl commands into a file `open_jtag.tcl`.

```
# Open connection to the JTAG Master
open_hw
connect_hw_server
open_hw_target
refresh_hw_device [lindex [get_hw_devices] 0]

# Create some reads/writes
create_hw_axi_txn wr_enable [get_hw_axis hw_axi_1] ...
 -address 44a0_0004 -data 0000_0001 -type write
create_hw_axi_txn wr_disable [get_hw_axis hw_axi_1] ...
 -address 44a0_0004 -data 0000_0000 -type write
```

Launch the Vivado Tcl console, sourcing the file you just created.

```
system('vivado -mode tcl -source open_jtag.tcl&')
```

```
C:\windows\system32\cmd.exe - vivado -mode tcl
WARNING: Default location for XILINX_VIVADO_HLS not found:

****** Vivado v2015.4 (64-bit)
  **** SW Build 1412921 on Wed Nov 18 09:43:45 MST 2015
  **** IP Build 1412160 on Tue Nov 17 13:47:24 MST 2015
    ** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Vivado%
```

When you are done using the JTAG Master, close the connection by using these Tcl commands.

```
# Close and disconnect from the JTAG Master
close_hw_target;
disconnect_hw_server;
```

Summary

You can use the JTAG AXI Manager IP to interface with HDL Coder IP core registers in systems that do not have an embedded ARM processor, such as the Kintex-7. You can use this IP as a first step to debug standalone HDL Coder IP cores, prior to hand coding software for soft processors, (such as MicroBlaze), or as a way to tune parameters on a running system.

Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow

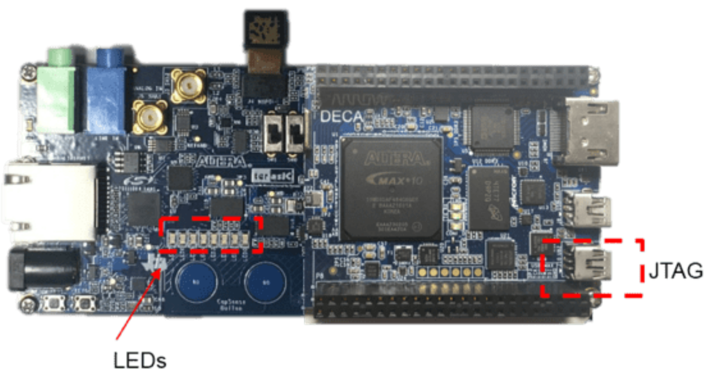
This example shows how to use the HDL Coder™ IP core generation workflow to develop reference designs for Intel® parts that do not use an embedded ARM® processor present but that still utilize the HDL Coder generated AXI interface to control the design under test (DUT). This example uses the HDL Verifier™ AXI Manager IP to access the HDL Coder generated DUT registers from MATLAB®. Alternatively, you can use the Intel Qsys JTAG to Avalon® Master Bridge IP to access the FPGA registers using Tcl commands in the Qsys system console. For the Intel JTAG AXI Master, you must create a custom reference design. The FPGA design is implemented on the Arrow® DECA MAX® 10 FPGA evaluation kit.

Requirements

- Intel Quartus® Prime Standard, with a supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- Arrow DECA MAX 10 FPGA evaluation kit
- HDL Coder Support Package for Intel FPGA and SoC Devices
- HDL Verifier Support Package for Intel FPGA Boards

Arrow DECA MAX 10 FPGA Evaluation Kit

This figure shows the Arrow DECA MAX 10 FPGA evaluation kit.



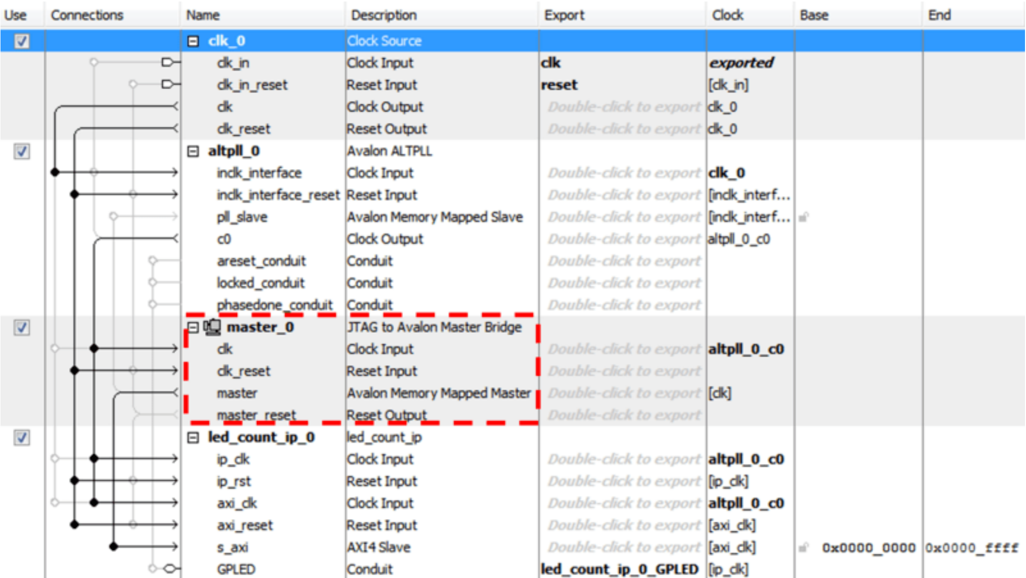
Example Reference Designs

Designs that can benefit from using the HDL Coder IP core generation workflow without using either an embedded ARM processor or an Embedded Coder™ support package but still leverage the HDL Coder generated AXI4 registers can include one of these IP sets.

- HDL Verifier AXI Manager + HDL Coder IP Core
- JTAG Master + HDL Coder IP Core
- Nios® II + HDL Coder IP Core
- PCIe® Endpoint + HDL Coder IP Core

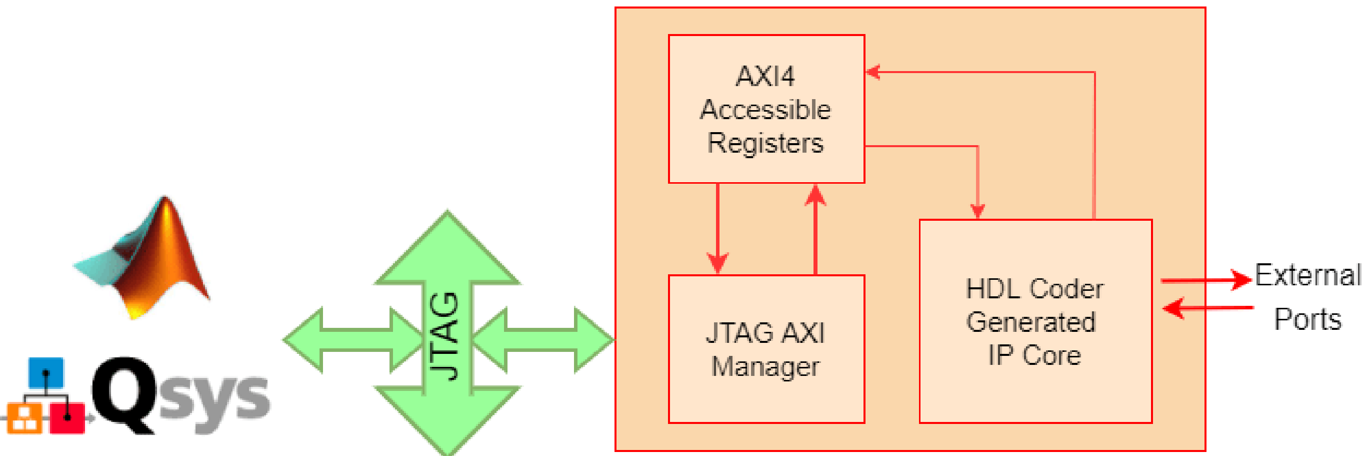
This example includes two reference designs.

- The Default System reference design uses MathWorks® IP and a MATLAB command line interface for issuing read and write commands. To use this design, you must have the HDL Verifier product.
- The Intel JTAG to AXI Master reference design uses Quartus IP for the JTAG to AXI Master and requires using the Quartus Tcl console to issue read and write commands.



Use	Connections	Name	Description	Export	Clock	Base	End
✓		clk_0	Clock Source				
		clk_in	Clock Input	clk	exported		
		clk_in_reset	Reset Input	reset	[clk_in]		
		clk	Clock Output	Double-click to export	clk_0		
		clk_reset	Reset Output	Double-click to export	clk_0		
✓		altpll_0	Avalon ALTPLL				
		inclk_interface	Clock Input	Double-click to export	clk_0		
		inclk_interface_reset	Reset Input	Double-click to export	[inclk_interf...		
		pll_slave	Avalon Memory Mapped Slave	Double-click to export	[inclk_interf...		
		c0	Clock Output	Double-click to export	altpll_0_c0		
		areset_conduit	Conduit	Double-click to export			
		locked_conduit	Conduit	Double-click to export			
		phasedone_conduit	Conduit	Double-click to export			
✓		master_0	JTAG to Avalon Master Bridge				
		clk	Clock Input	Double-click to export	altpll_0_c0		
		clk_reset	Reset Input	Double-click to export			
		master	Avalon Memory Mapped Master	Double-click to export	[clk]		
		master_reset	Reset Output	Double-click to export			
✓		led_count_ip_0	led_count_ip				
		ip_clk	Clock Input	Double-click to export	altpll_0_c0		
		ip_rst	Reset Input	Double-click to export	[ip_clk]		
		axi_clk	Clock Input	Double-click to export	altpll_0_c0		
		axi_reset	Reset Input	Double-click to export	[axi_clk]		
		s_axi	AXI4 Slave	Double-click to export	[axi_clk]	0x0000_0000	0x0000_ffff
		GPLED	Conduit	led_count_ip_0_GPLED	[ip_clk]		

The two reference designs differ by only the JTAG manager IP that they use.



HDL Verifier AXI Manager Reference Design

In the IP core generation workflow of the HDL Workflow Advisor, in the **Set Target Reference Design** step, enable the **Insert AXI Manager (HDL Verifier required)** parameter. This option adds AXI Manager IP automatically into the reference design and connects the added IP to the DUT IP using the AXI4-slave interface. The next section details the steps to auto-insert the JTAG AXI Manager IP in the reference design.

Execute IP Core Workflow

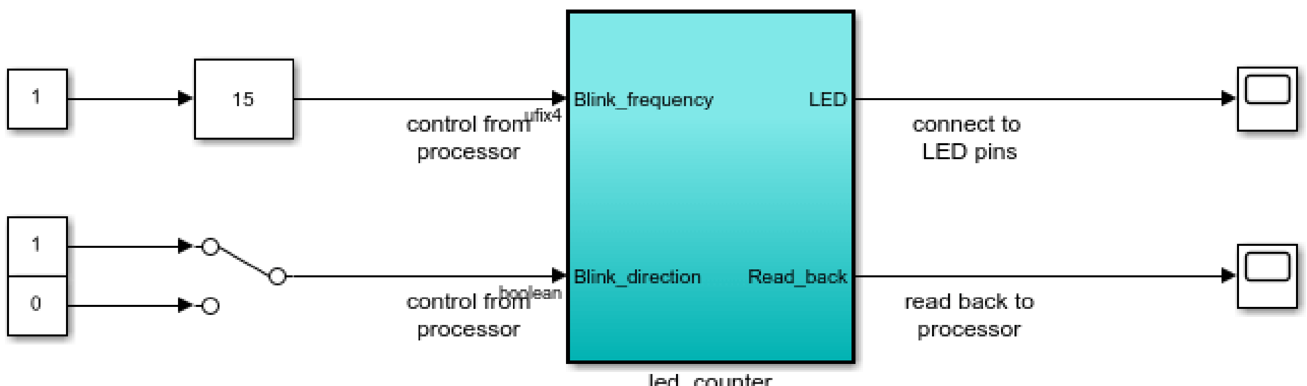
Follow these steps to execute the IP core workflow for the Default System reference design, which uses JTAG AXI Manager IP. Using this reference design, you can generate an HDL IP core that blinks LEDs on the DECA board. To generate the HDL IP core, follow these steps.

1. Set up the Intel Quartus® tool path. Use your own Intel Quartus installation path by executing this command in MATLAB.

```
hdlsetuptoolpath('ToolName','Altera QUARTUS II', ...
            'ToolPath','C:\intelFPGA\20.1\quartus\bin64\quartus.exe');
```

2. Open the Simulink model that implements LED blinking by executing this command in MATLAB.

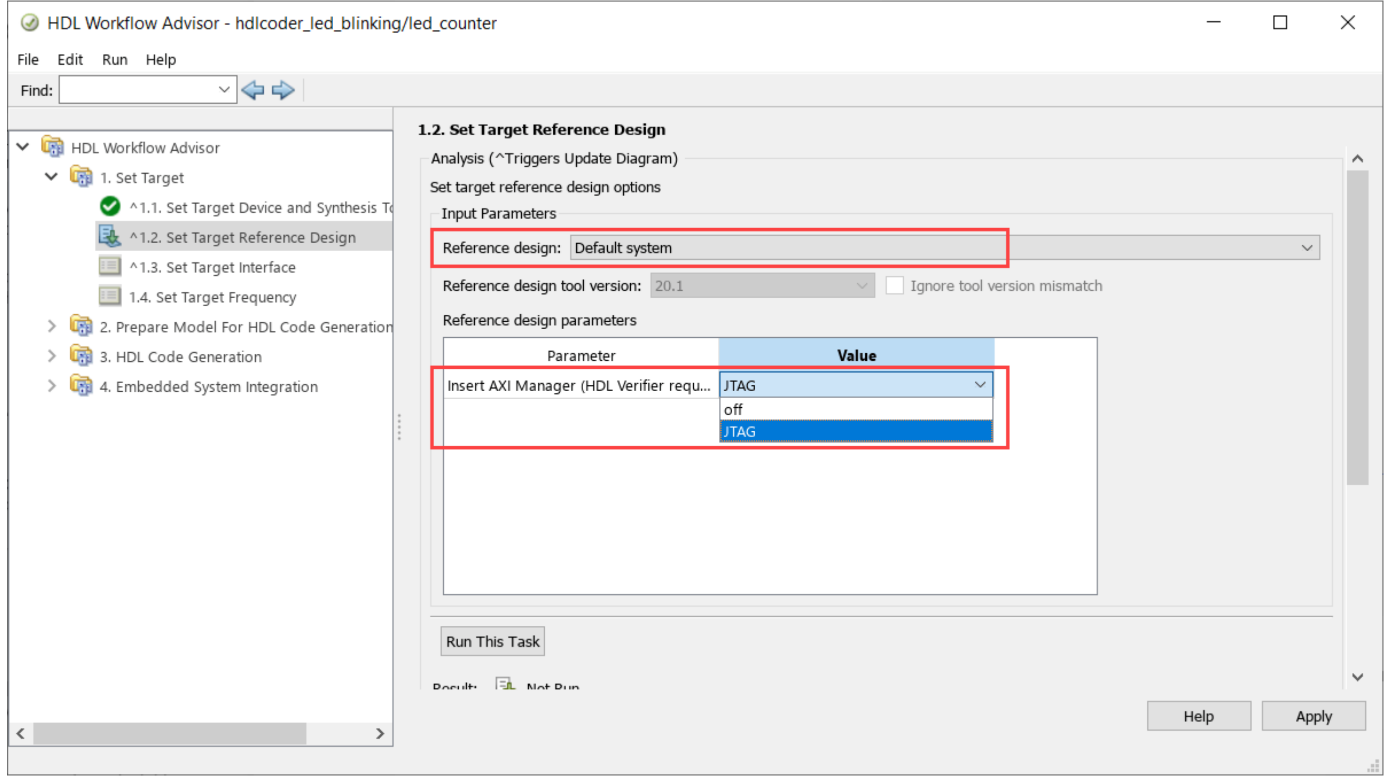
```
open_system('hdlcoder_led_blinking')
```



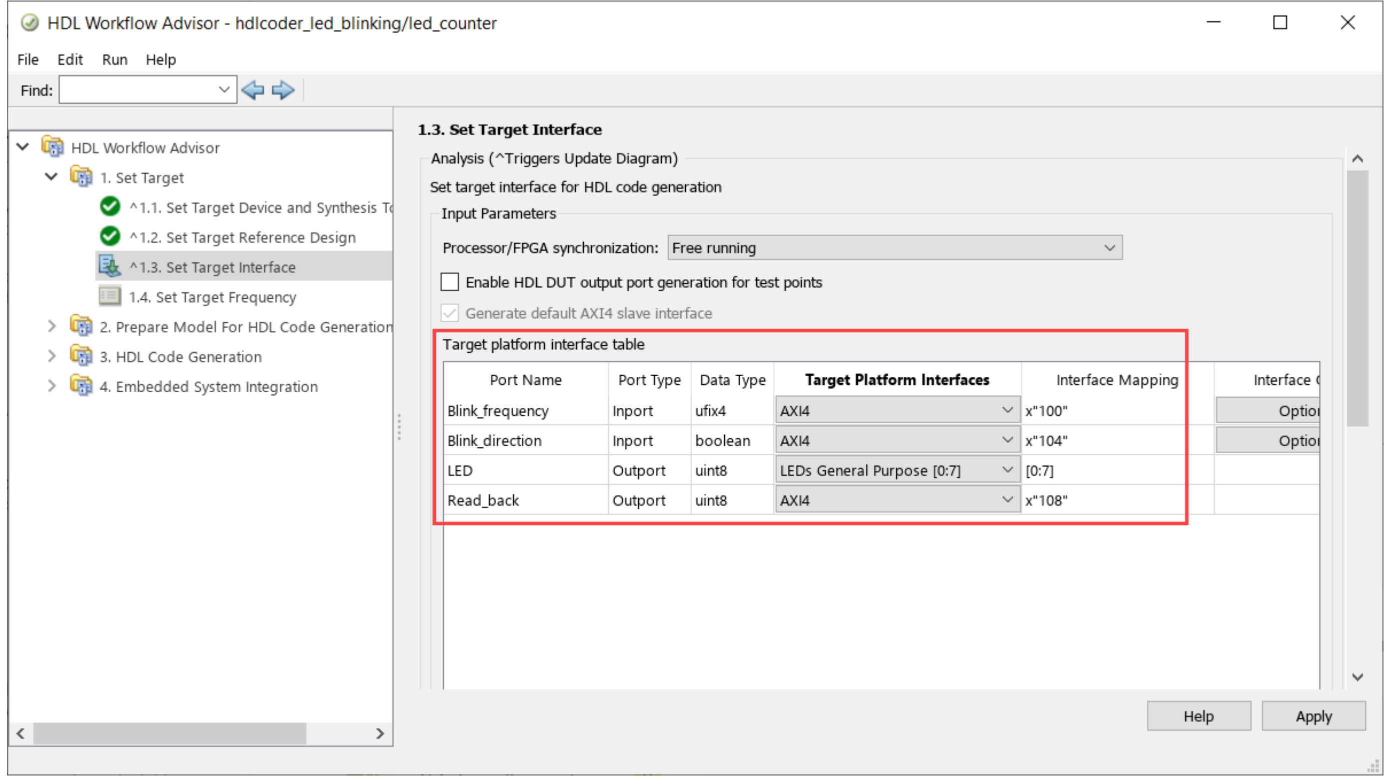
3. Launch HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem and selecting **HDL Code** followed by **HDL Workflow Advisor**.

4. In step 1.1, set **Target workflow** to `IP Core Generation` and **Target platform** to `Arrow DECA MAX 10 FPGA evaluation kit`. Click **Run This Task**.

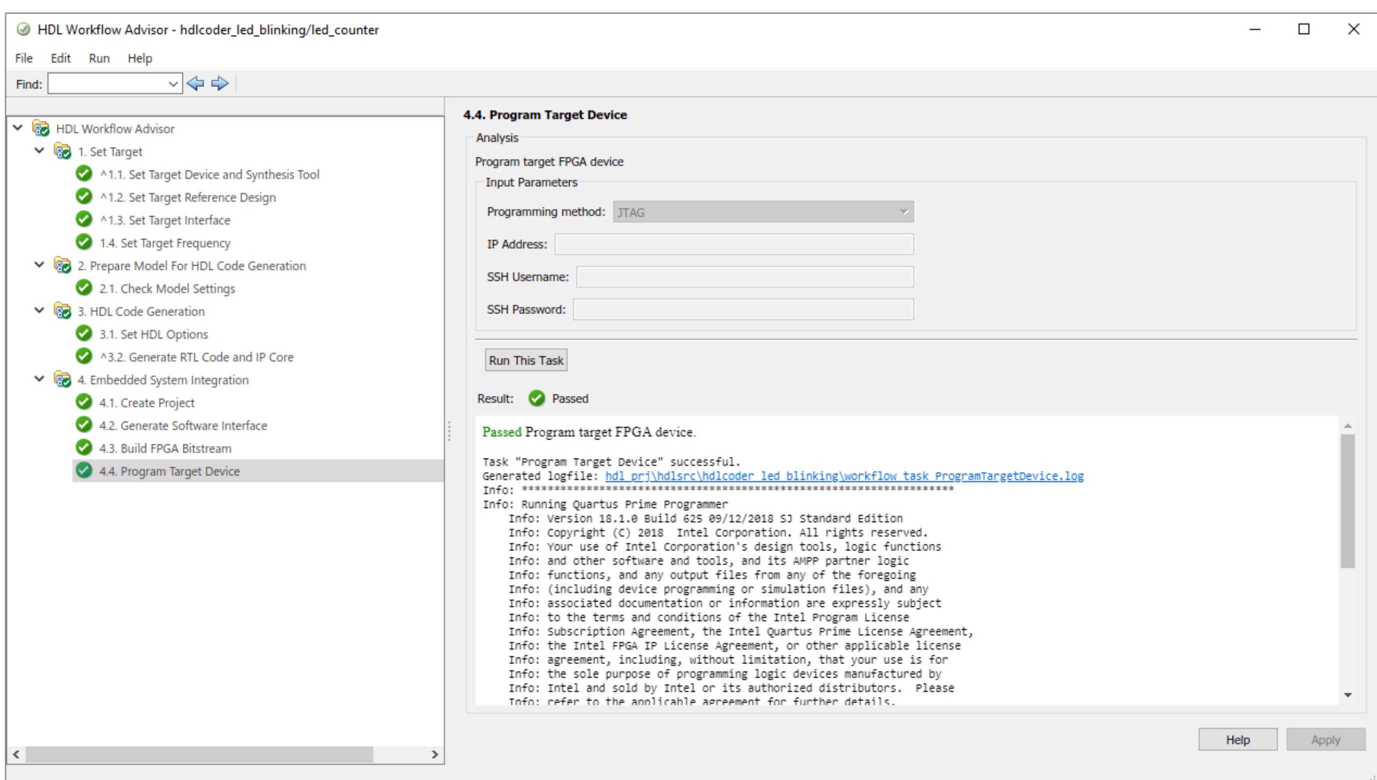
5. In step 1.2, set **Reference design** to `Default system`. Under **Reference design parameters**, set **Insert AXI Manager (HDL Verifier required)** to `JTAG`.



6. In step 1.3, set the interface of the **Blink_frequency**, **Blink_direction**, and **Read_back** ports to `AXI4`. Set the interface of the **LED** port to `LEDs General Purpose [0:7]`.

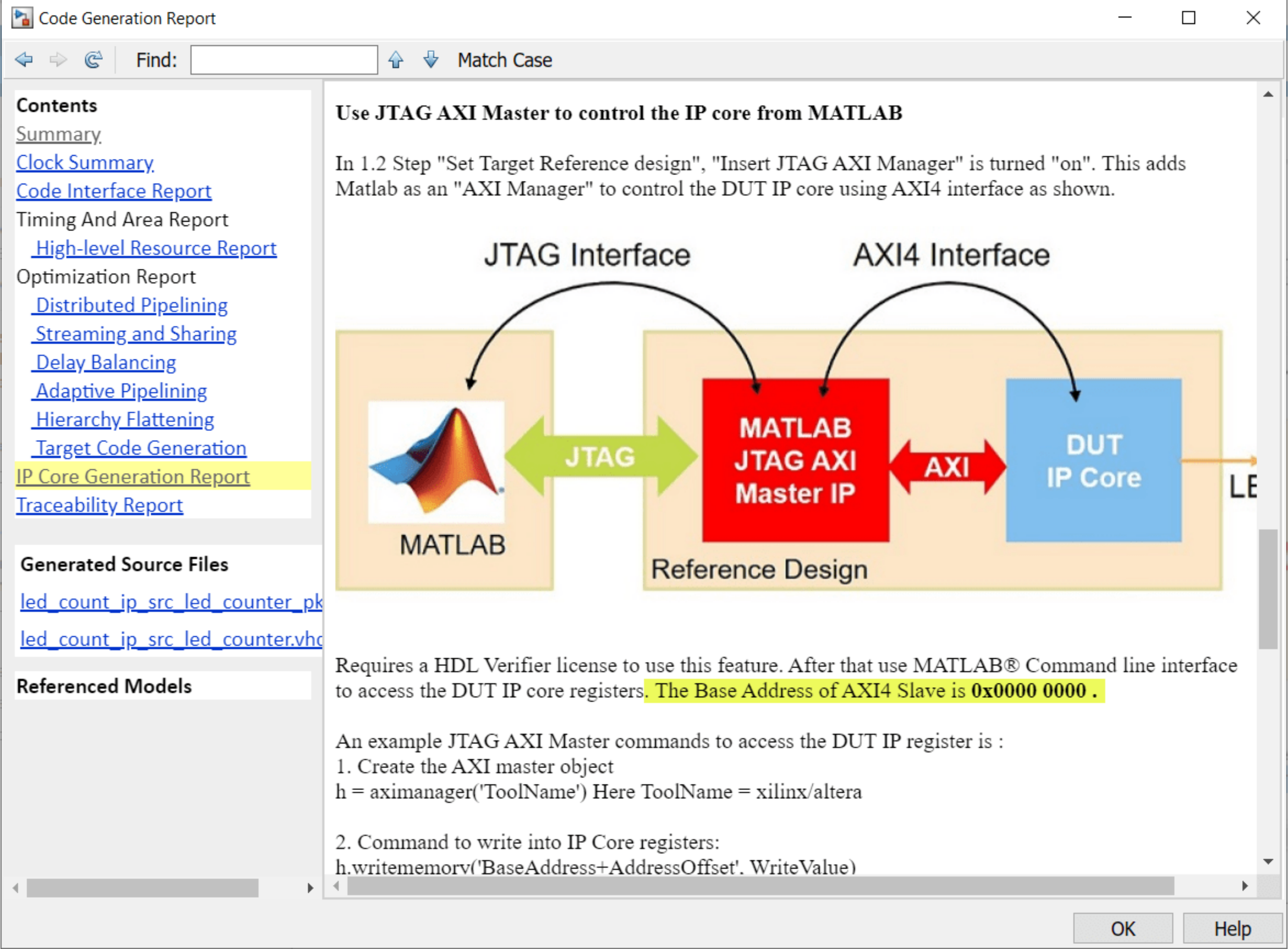


7. Run the remaining steps in the workflow to generate a bitstream and program the target device.

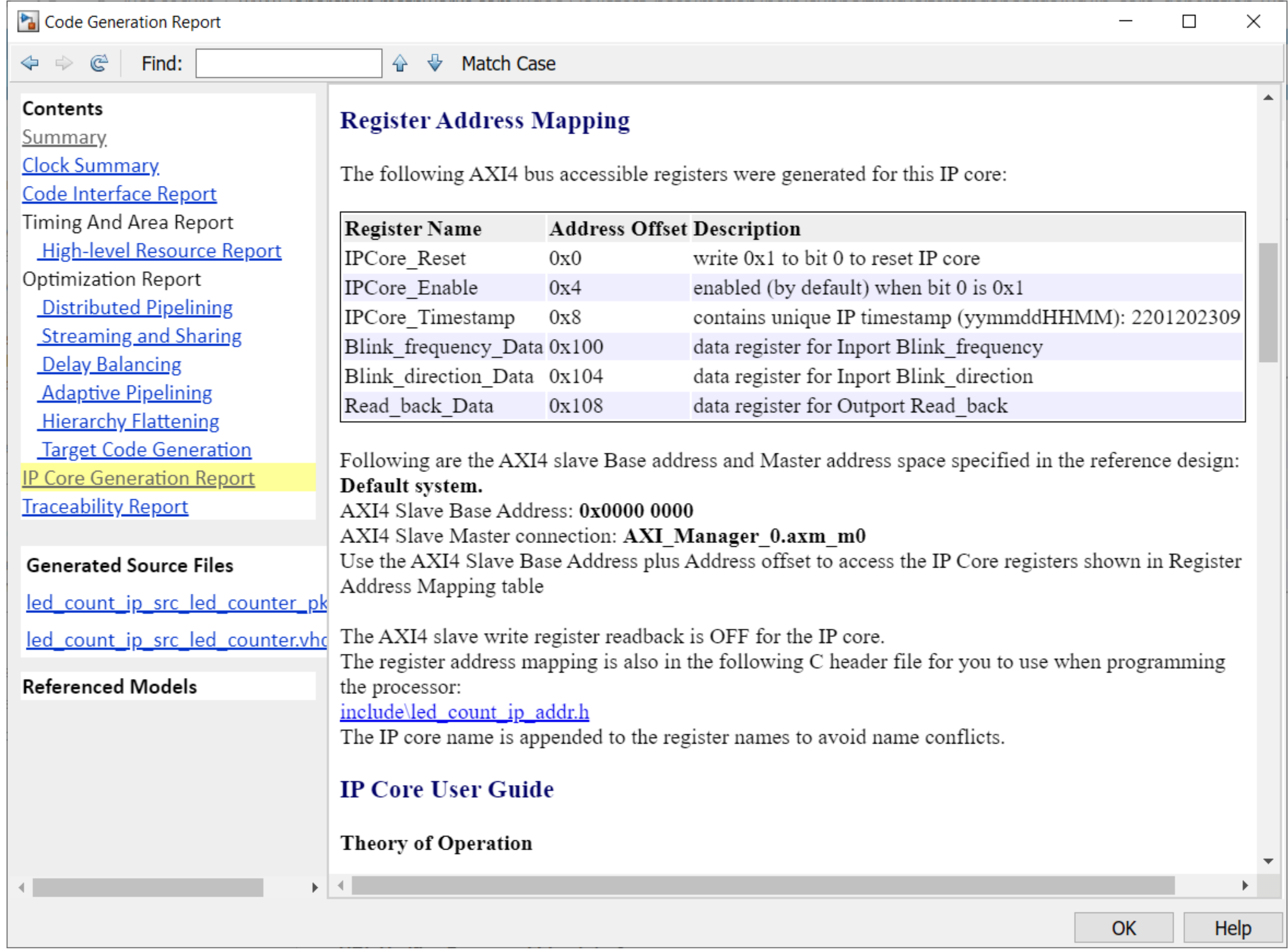


Determine Addresses from IP Core Report

The base address for an HDL Coder IP core is defined as `0x00000000` for the Default System reference design, which uses the AXI Manager IP. You can see address setting in the generated IP core report as shown this figure.

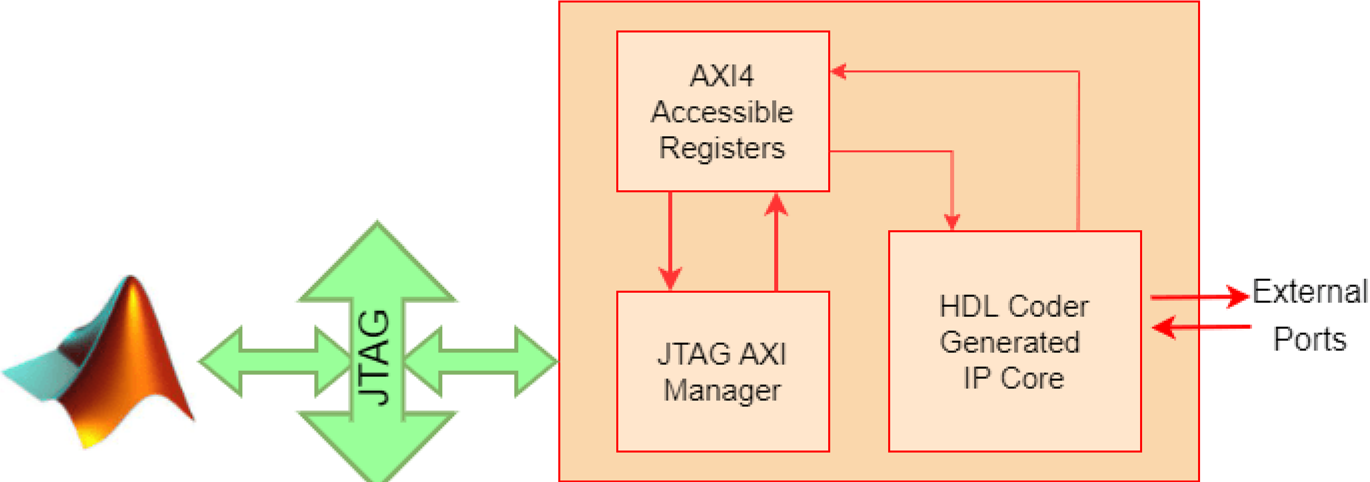


The IP core report register address mapping table shows the offsets.



HDL Verifier Command Line Interface

If you have the HDL Verifier support package for Intel FPGA boards and select the AXI Manager reference design, then you can use the MATLAB command line interface to access the IP core that is generated by the HDL Coder product.



To write and read from the DDR memory, follow these steps.

1. Create an AXI manager object.

```
h = aximanager('Altera')
```

2. Issue a write commands. For example, disable the DUT.

```
h.writememory('4',0)
```

3. Re-enable the DUT.

```
h.writememory('4',1)
```

4. Read the current counter value.

```
h.readmemory('108',1)
```

5. Delete the object to free up the JTAG resource. If you do not delete the object, other JTAG operations, such as programming the FPGA, fail.

```
delete(h)
```

Intel JTAG to AXI Master Reference Design

Create a custom reference design to use the Intel JTAG to AXI Master IP in the reference design, and then add reference design files to the MATLAB path using the `addpath` command.

Access the HDL Coder IP core registers using the Intel JTAG to AXI Master IP by using the base address that is defined in reference design plugin file.

Qsys System Console Tcl Commands for AXI Read and Write

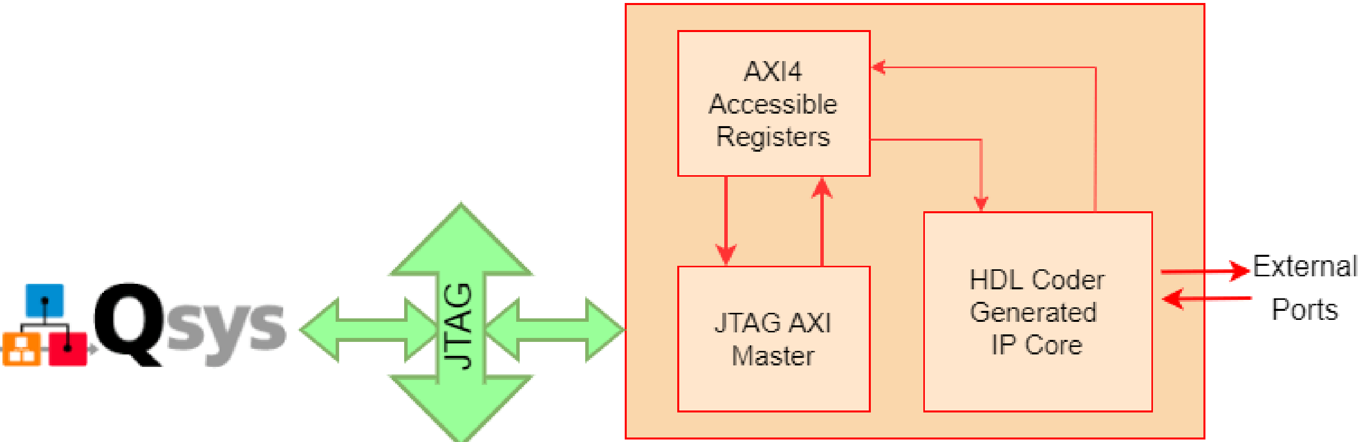
Before opening a system console, choose the appropriate Qsys read and write commands. For this example, because all of the HDL Coder generated IP core registers are currently 32 bits, use these read and write commands.

```
% master_write_32 <service-path> <start-address> <list-of-32-bit-values>
% master_read_32  <service-path> <start-address> <size-in-multiples-of-32-bits>
```

For example, write the 32 bit hex value 0x12345678 to the IP core register defined by offset 0x100 using a previously defined service path stored in the variable `$jtag`.

```
% master_write_32 $jtag 0x100 0x12345678
```

Before you can generate reads and writes, you must first launch a system console and open a connection to the JTAG Master that issues the register reads and writes.



To open a connection to the JTAG Master, first set a variable that stores the service path (in this case, only one master exists).

```
% set jtag [lindex [get_service_paths master] 0]
```

Use the variable to open the JTAG Master in master mode.

```
% open_service master $jtag
```

Launch the Altera® system console and enter the commands to open the JTAG Master.

```
system('C:\intelFPGA\17.1\quartus\sopc_builder\bin\system-console&')
```

Tcl Console

```
%

% set jtag [lindex [get_service_paths master] 0]
/devices/10M50DA(.|ES)|10M50DC@1#USB-1#Arrow MAX 10 DECA/(link)/JTAG/(110:132 v1 #0)/phy_0/master
% open_service master $jtag

% master_write_32 $jtag 0x04 0x00

% master_write_32 $jtag 0x04 0x01

% master_read_32 $jtag 0x108 1
0x000000f0
% close_service master $jtag

%
```

When you are done using the JTAG Master, close the connection by using this Tcl command.

```
close_service master $jtag
```

Summary

You can use the JTAG AXI Manager IP to interface with HDL Coder IP core registers in systems that do not have an embedded ARM processor, such as the MAX 10. You can use this IP as a first step to debug standalone HDL Coder IP cores, prior to hand coding software for soft processors, (such as Nios II), or as a way to tune parameters on a running system.

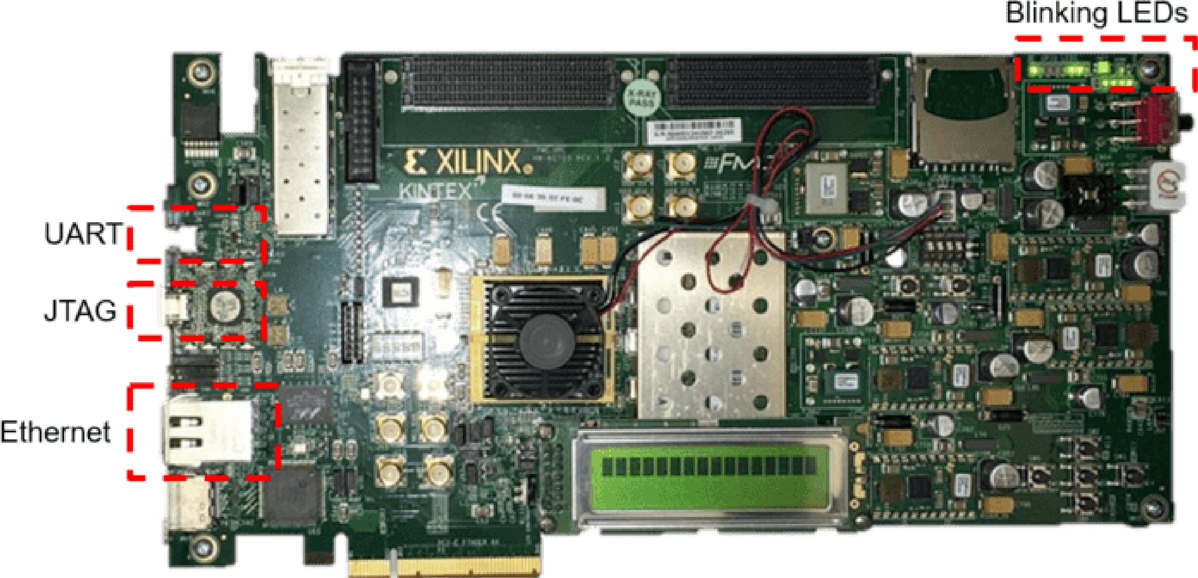
IP Core Generation Workflow with a MicroBlaze processor: Xilinx Kintex-7 KC705

This example shows how to use the HDL Coder™ IP Core Generation Workflow to develop reference designs for Xilinx® parts without an embedded ARM® processor present, but which still utilize the HDL Coder generated AXI interface to control the DUT. Specifically, this example will use the Xilinx Kintex-7 KC705 board and a MicroBlaze™ soft processor running a LightWeightIP (lwIP) based TCP/IP firmware server in the reference design to access the HDL Coder generated DUT registers from anywhere on the connected network. Further, this example will also highlight the difference between accessing data from a collection of registers implemented as multiple scalar ports and a collection of registers implemented as a single vector port.

Requirements

- Xilinx Vivado Design Suite, with the supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.
- Xilinx Kintex-7 KC705 development board
- HDL Coder support package for Xilinx FPGA and SoC Devices
- Ethernet connection

Xilinx Kintex-7 KC705 development board



Example Reference Designs

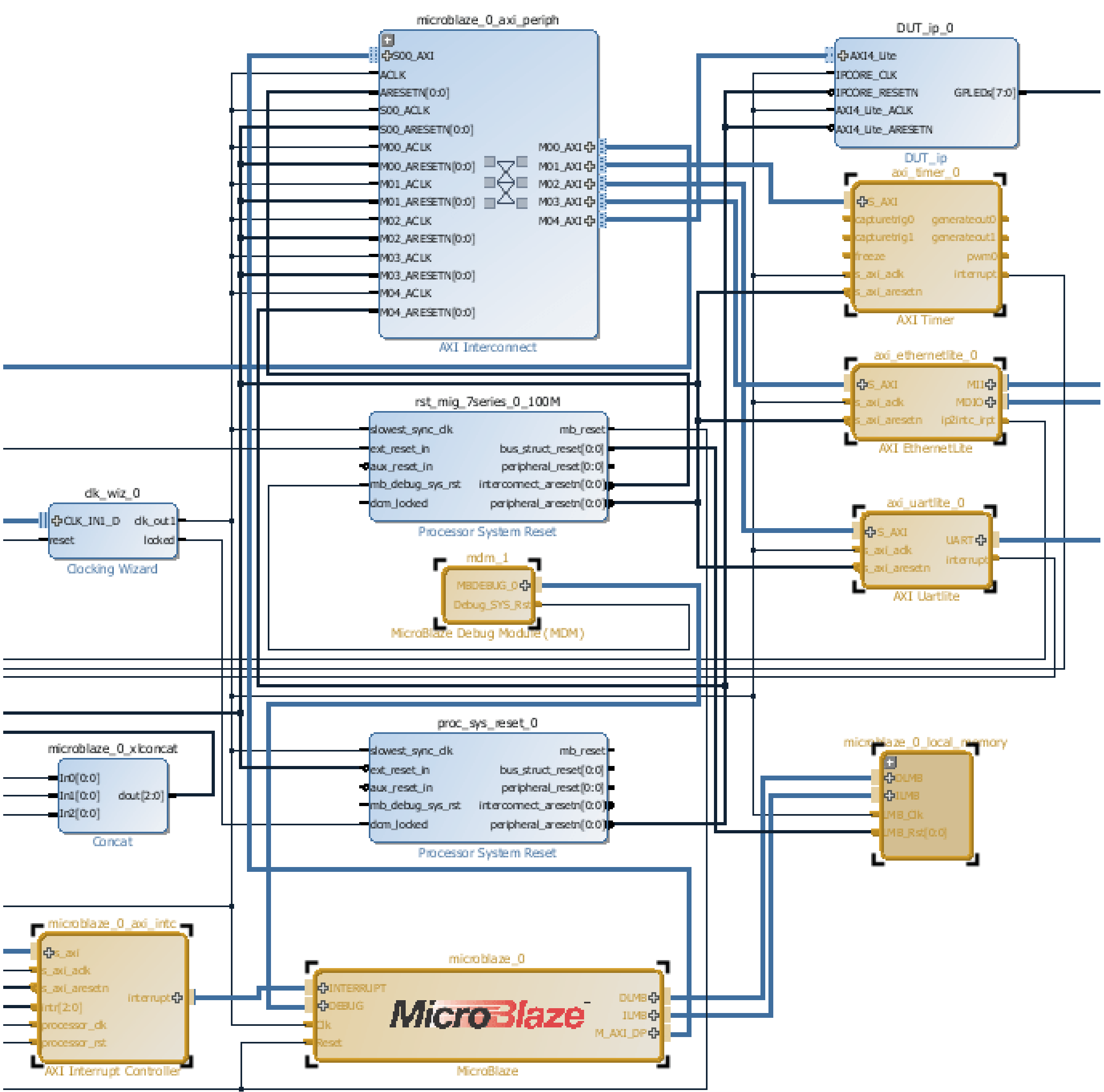
MicroBlaze is a simple, versatile soft-core processor that can be used in Xilinx FPGA only platforms, such as the Kintex-7, to perform the functionality a full-fledged processor, or as a flexible, programmable IP. When programs are small, the ELF can sit in BRAM and the design becomes

completely self contained in the FPGA. There are many applications well suited to being implemented on a MicroBlaze. Here we list just a few:

1 Remote networked control of a deployed IP Core algorithm
2 Embedded web server for control and data display
3 Integration of existing software algorithms to hardware-only platforms

The following is a system level diagram for this MicroBlaze system:

The reference design, "Xilinx MicroBlaze TCP/IP to AXI4-Lite Master", uses Vivado™ MicroBlaze IP to translate TCP/IP packets into AXI4-Lite reads and writes. Below is a block diagram of the complete system, including all the peripherals required to operate the TCP/IP server and debug via the UART serial console.

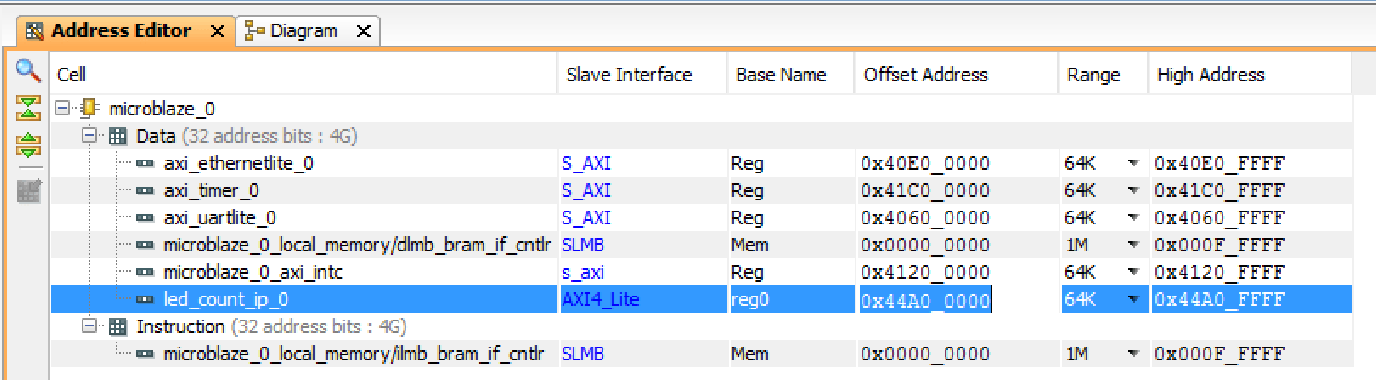


MicroBlaze Setup

In order to operate the TCP/IP server, the MicroBlaze IP needs a few basic peripherals:

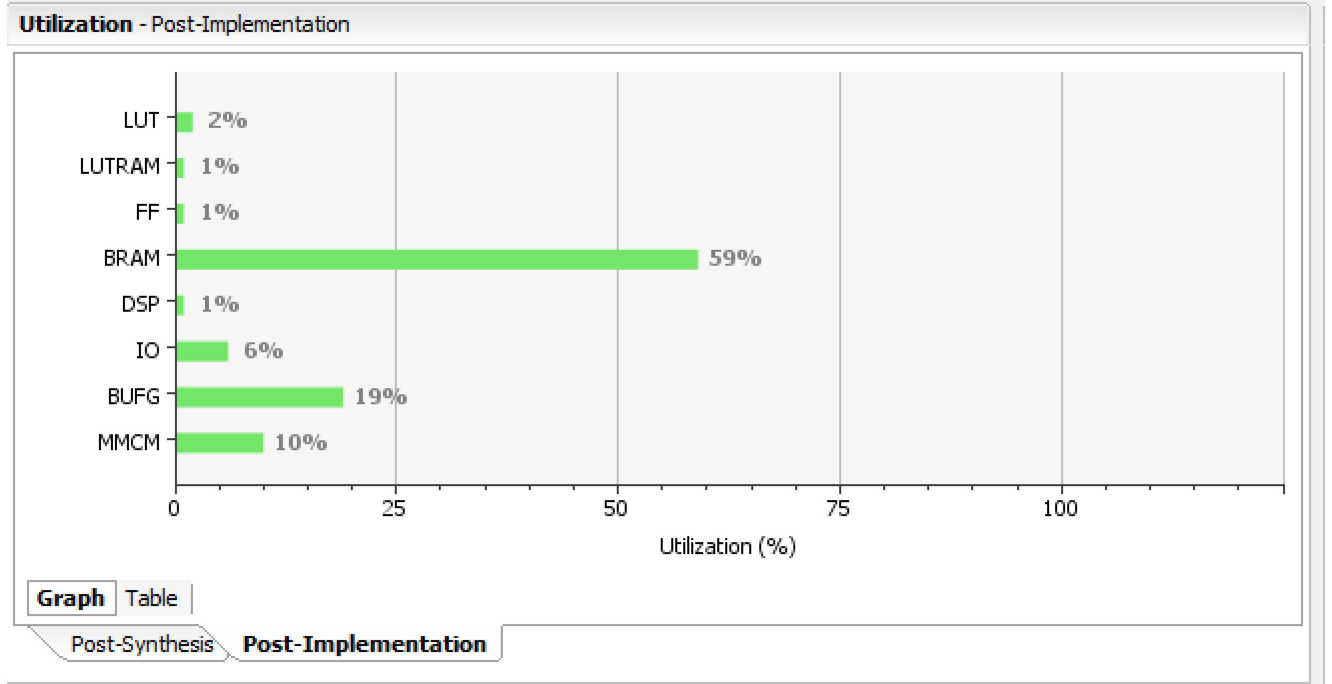
- local memory (BRAM) for data/instructions
- Ethernet core for transmitting and receiving frames
- UART core for sending debug messages
- Timer core for generating timeout interrupts
- Interrupt controller for handling interrupts from all these peripherals.

As can be seen in the address editor below, all these peripherals are connected to the MicroBlaze via AXI4 interfaces.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
axi_ethernetlite_0	S_AXI	Reg	0x40E0_0000	64K	0x40E0_FFFF
axi_timer_0	S_AXI	Reg	0x41C0_0000	64K	0x41C0_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	1M	0x000F_FFFF
microblaze_0_axi_intc	s_axi	Reg	0x4120_0000	64K	0x4120_FFFF
led_count_ip_0	AXI4_Lite	reg0	0x44A0_0000	64K	0x44A0_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	1M	0x000F_FFFF

The amount of local memory allocated using BRAM is 1MB. This amount is needed to run the lwIP stack. The benefit of specifying BRAM for local memory is that the executable ELF can be included in the bitstream, which simplifies programming and enables targeting existing FPGA boards which may not have external DRAM memory. However, this convenience comes at a cost of increased utilization:



If BRAM utilization is a concern and DRAM resources are available, you may opt to replace local BRAM memory with external DRAM memory. See Xilinx app note "xapp1026" for more details on alternate configurations and application information.

Example Reference Design plugin_rd.m

The plugin_rd.m for this reference design is shown below:

```
function hRD = plugin_rd()
% Reference design definition


%   Copyright 2014-2018 The MathWorks, Inc.


% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Xilinx MicroBlaze TCP/IP to AXI4-Lite Master';
hRD.BoardName = 'Xilinx Kintex-7 KC705 development board';


% Tool information
hRD.SupportedToolVersion = {'2017.2','2017.4'};


%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
  'CustomBlockDesignTcl', 'system_top.tcl',...
  'VivadoBoardPart',      'xilinx.com:kc705:part0:1.1');


% add custom files, use relative path
hRD.CustomFiles         = {'mw_lwip_tcpip_axi4.elf'};


%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
  'ClockConnection',      'clk_wiz_0/clk_out1', ...
  'ResetConnection',      'proc_sys_reset_0/peripheral_aresetn',...
  'DefaultFrequencyMHz',  100,...
  'MinFrequencyMHz',      100,...
  'MaxFrequencyMHz',      100,...
  'ClockNumber',          1,...
  'ClockModuleInstance',  'clk_wiz_0');


% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
  'InterfaceConnection', 'microblaze_0_axi_periph/M04_AXI', ...
  'BaseAddress',         '0x44A00000',...
  'MasterAddressSpace',  'microblaze_0/Data',...
  'InterfaceType',       'AXI4-Lite',...
  'InterfaceID',         'MicroBlaze AXI4-Lite Interface');

hRD.HasProcessingSystem = false; % No hard processing system
```

Note that the reference design includes the MicroBlaze executable `mw_lwip_tcpip_axi4.elf` in the reference design property `CustomFiles`. This will copy the executable from the reference design to the Vivado project so that it can be associated with the MicroBlaze IP.

Additional code in 'system_top.tcl' to attach ELF to uBlaze

The 'system_top.tcl' file included in most reference designs is used to create the top level Vivado IP Integrator block diagram containing most of the reference design IP. Here we are adding some additional Tcl code to this file after the block diagram has been created to associate the standalone MicroBlaze ELF executable with the MicroBlaze IP.

```
import_files -norecurse mw_lwip_tcpip_axi4.elf
generate_target all [get_files system_top.bd]
set_property SCOPED_TO_REF system_top [get_files -all -of_objects [get_fileset sources_1] {mw_
set_property SCOPED_TO_CELLS { microblaze_0 } [get_files -all -of_objects [get_fileset source
```

Doing this allows the ELF to be packaged with the bitstream and programmed into the MicroBlaze BRAM memory at the same time as the FPGA.

Execute the IP Core Workflow

Using the above reference design you will generate an HDL IP Core that blinks LEDs on the KC705 board. You will then use `tcpclient` to send/receive formatted packets to the MicroBlaze to issue reads/writes over the AXI4-Lite interface to the generated HDL IP Core.

1. Add the MicroBlaze reference design files to the MATLAB path using the command:

```
example_root = (hdlcoder_amd_examples_root)
cd (example_root)
addpath(genpath('KC705'));
```

2. Set up the Xilinx Vivado tool path by using the following command:

```
>> hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado
```

Use your own Xilinx Vivado installation path when executing the command.

3. Open the Simulink model that implements LED blinking, as well as vector output ports for comparison to scalar ports, using the command:

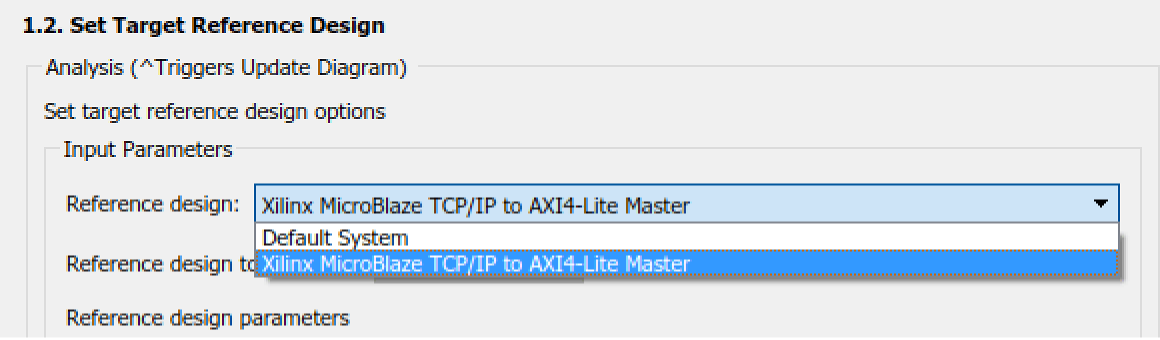
```
open_system('hdlcoder_led_vector')
```

LED
Read_back
1
15
ufix4
Blink_frequency
count1
count2
count3
count4
1
0
count5
boolean
Blink_direction
count6
vec_count
DUT

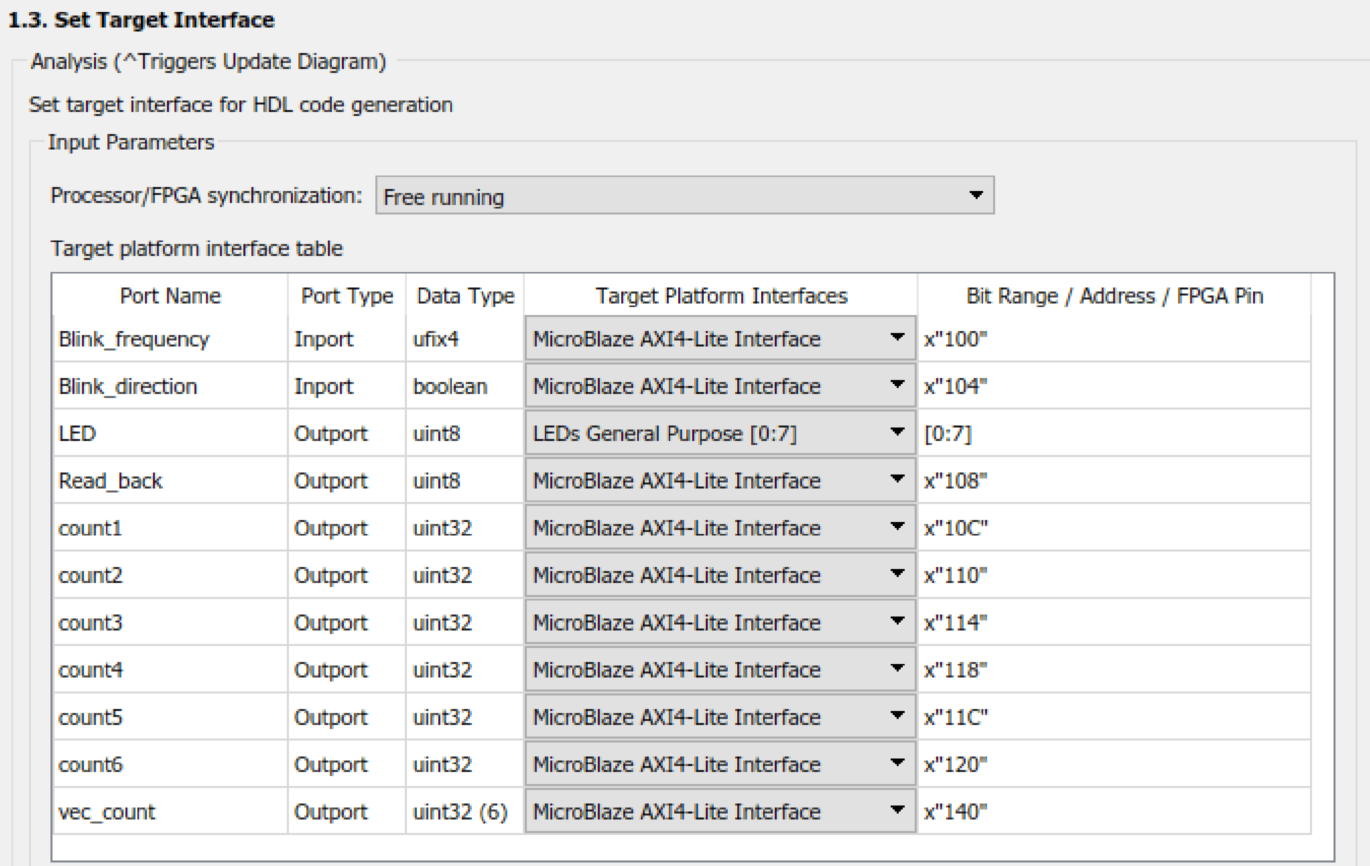
Copyright 2016 The MathWorks, Inc.

4. Launch HDL Workflow Advisor from the `hdlcoder_led_vector/DUT` subsystem by right-clicking the `DUT` subsystem, and selecting **HDL Code > HDL Workflow Advisor**.

5. Select reference design from the drop down in step 1.2



6. Assign register ports to the "MicroBlaze AXI4-Lite Interface". These will then be accessible at the hex offset shown in the table.



Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
Blink_frequency	Inport	ufix4	MicroBlaze AXI4-Lite Interface	x"100"
Blink_direction	Inport	boolean	MicroBlaze AXI4-Lite Interface	x"104"
LED	Outport	uint8	LEDs General Purpose [0:7]	[0:7]
Read_back	Outport	uint8	MicroBlaze AXI4-Lite Interface	x"108"
count1	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"10C"
count2	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"110"
count3	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"114"
count4	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"118"
count5	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"11C"
count6	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"120"
vec_count	Outport	uint32 (6)	MicroBlaze AXI4-Lite Interface	x"140"

7. Run the remaining steps in the workflow to generate a bitstream and program the target device.

Determining Addresses from the IP Core Report

The Base Address for an HDL Coder IP Core is defined in the reference design plugin_rd.m with the following command:

```
% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
  'InterfaceConnection', 'microblaze_0_axi_periph/M04_AXI', ...
  'BaseAddress',         '0x44A00000',...
  'MasterAddressSpace',  'microblaze_0/Data',...
  'InterfaceType',       'AXI4-Lite',...
  'InterfaceID',         'MicroBlaze AXI4-Lite Interface');
```

For this design, the base address is `0x44A0_0000`. The offsets can be found in the IP Core Report Register Address Mapping table:

Register Address Mapping

The following AXI4-Lite bus accessible registers were generated for this IP core:

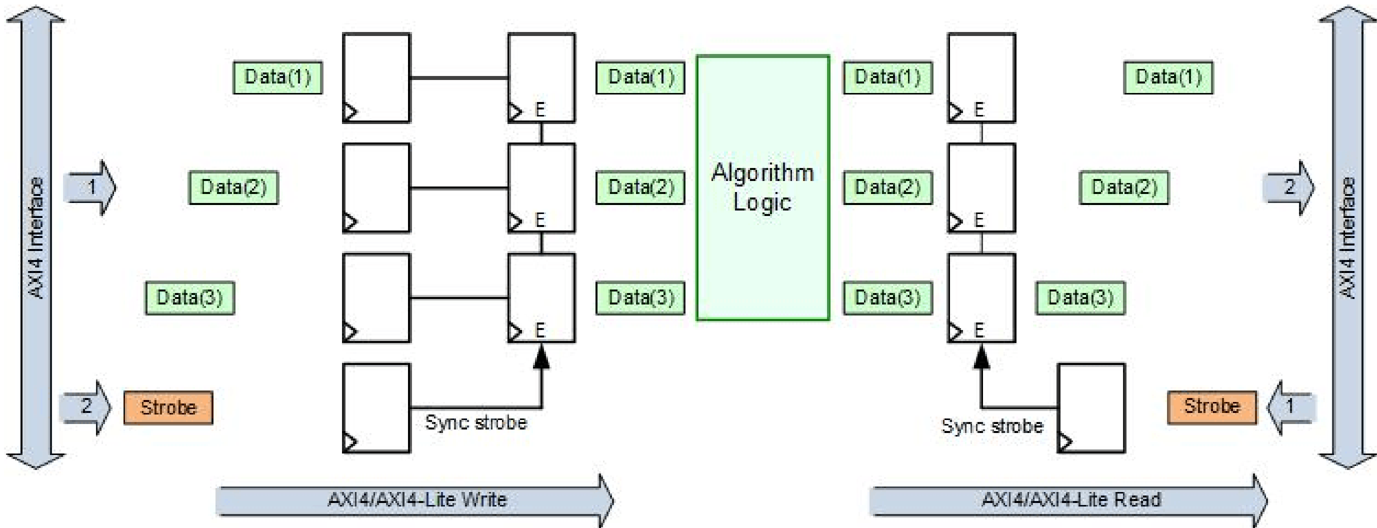
Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
Blink_frequency_Data	0x100	data register for Inport Blink_frequency
Blink_direction_Data	0x104	data register for Inport Blink_direction
Read_back_Data	0x108	data register for Outport Read_back
count1_Data	0x10C	data register for Outport count1
count2_Data	0x110	data register for Outport count2
count3_Data	0x114	data register for Outport count3
count4_Data	0x118	data register for Outport count4
count5_Data	0x11C	data register for Outport count5
count6_Data	0x120	data register for Outport count6
vec_count_Data	0x140	data register for Outport vec_count, vector with 6 elements, address ends at 0x154
vec_count_Strobe	0x160	strobe register for port vec_count

The register address mapping is also in the following C header file for you to use when programming the processor:
include\DUT_ip_addr.h
The IP core name is appended to the register names to avoid name conflicts.

Vector Data Read/Write with Strobe Synchronization

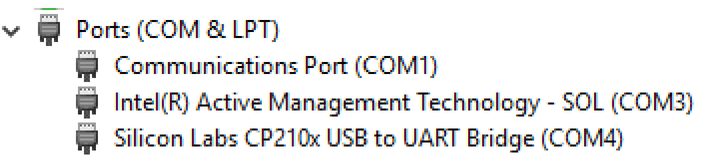
Vector data is supported on the AXI4/AXI4-Lite interfaces as of R2017a. Unlike a collection of scalar ports, all the elements of vector data are treated as synchronous to the IP Core algorithm logic. Additional strobe registers added for each vector input and output port maintain this synchronization across multiple sequential AXI4 reads/writes.

For input ports, the strobe register controls the enables on a set of shadow registers, allowing the IP core logic to see all the updated vector elements simultaneously. For output ports, the strobe register controls the synchronous capturing of vector data to be read. Below is a diagram of the synchronization logic generated with vector data:

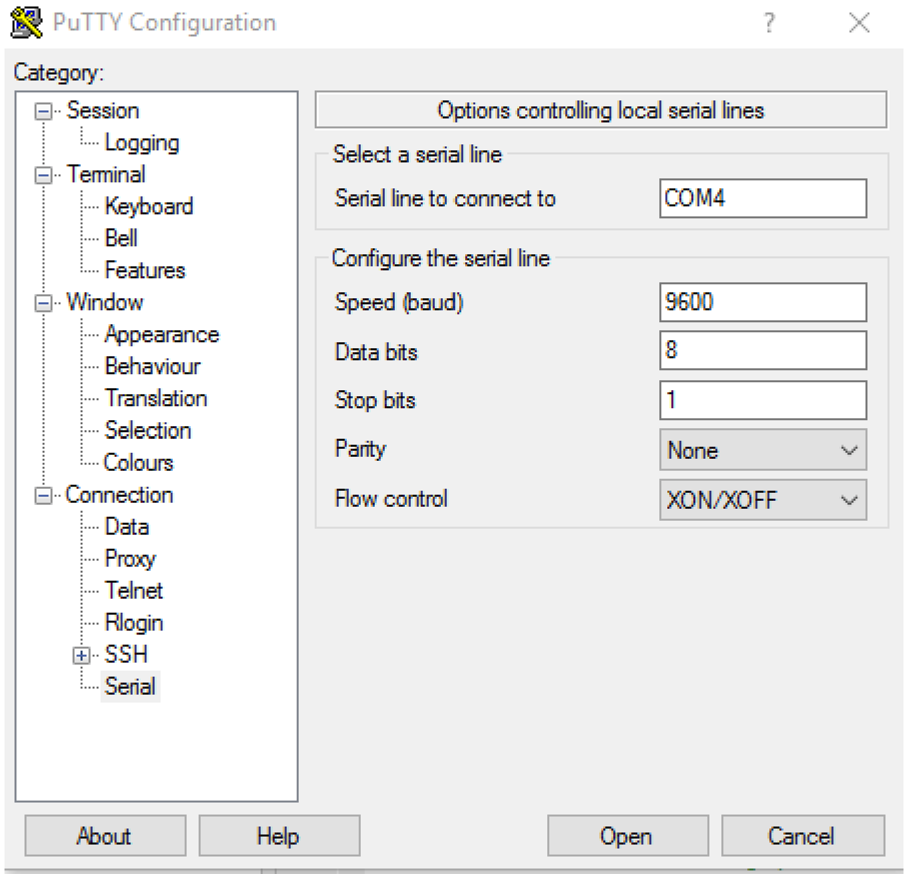


Connect to the TCP/IP server

To start interacting with the TCP/IP server running on the MicroBlaze, first connect the UART serial console to view debug messages, which will help ensure things are working as expected. First find the serial port that is connected to the UART on the board:



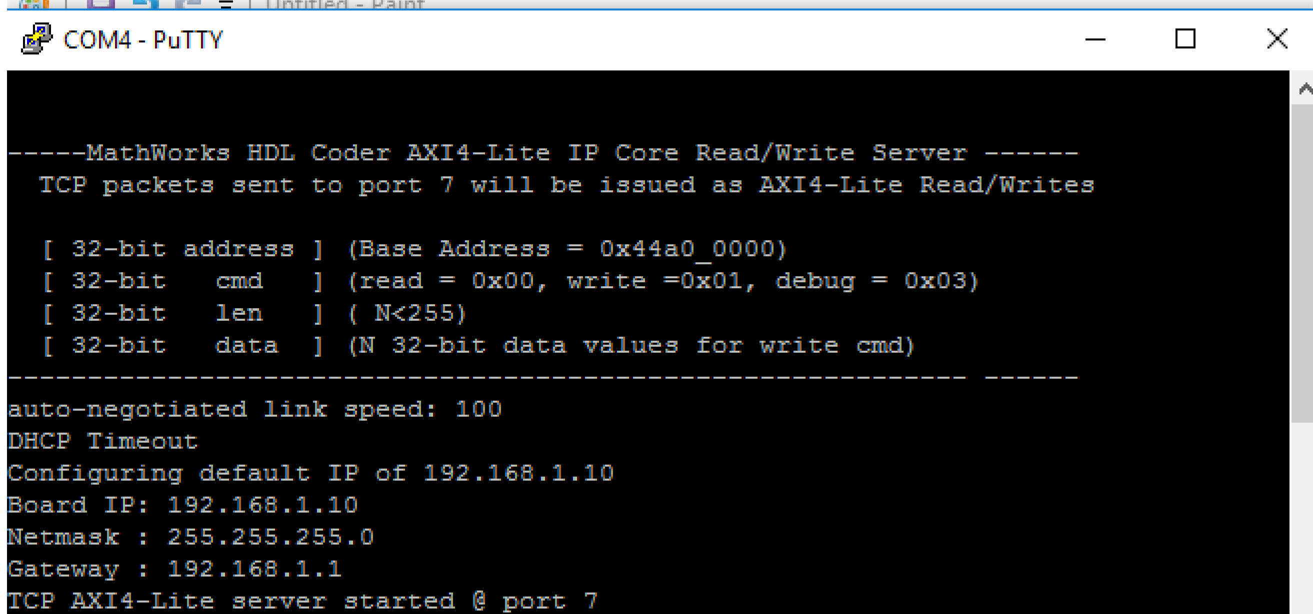
Then use this port to connect using a program such as PuTTY™:



Once connected to the UART serial console, run the `hdlworkflow_ProgramTargetDevice.m` script to reprogram the board.

```
>>hdlworkflow_ProgramTargetDevice
### Workflow begin.
### Loading settings from model.
### ++++++++++++++ Task Program Target Device ++++++++++++++
### Generated logfile: hdl_prj\hdlsrc\hdlcoder_led_vector\workflow_task_ProgramTargetDevice.lo
### Task "Program Target Device" successful.
### Workflow complete.
```

In the console window, you should see the following header, displaying the IP address and port number the server is connected to.



NOTE: If the board is connected to a network with a DHCP server enabled, the IP information will be different than shown above. In this case, you will need to modify line 43 of the `read_write_test.m` script to connect to the correct IP address of the board:

```
t = tcpclient('192.168.1.10',7);
```

Sending AX4-Lite transactions to the MicroBlaze from MATLAB using `tcpipclient`

In order to issue reads and writes to the IP Core via TCP/IP and AXI4-Lite, the address,data and command to be performed must be encoded in the packet sent to the TCP/IP server. For this example, we use the following packet format:

```
[--Address--]  32-bits
[----Cmd----]  lower byte of 32-bit word (READ = 0, WRITE = 1, DEBUG = 2)
[---Length---]  lower byte of 32-bit work (N<255)
[----Data---]  32-bits, used only for WRITE cmd
```

For example, a packet issuing a read of 3 consecutive values starting at address 0x44a0010c:

```
[44 a0 01 0c]
[00 00 00 00]
[00 00 00 03]
```

This will return data at offsets 0x10c,0x110,0x114.

For example, a packet issuing a write of 0x0 to offset 0x04, would be:

```
[44 a0 00 04]
[00 00 00 01]
[00 00 00 01]
[00 00 00 00]
```

To change the debug level used to print to the console, send a debug cmd packet:

```
[xx xx xx xx]
[00 00 00 03]
[00 00 00 01] %0 = no msg, 1 = READ|WRITE, 2 = full pkt
```

Run the read_write_test.m script

This example includes a script which will setup a connection to the TCP/IP server running on the MicroBlaze, create commands to enable/disable the DUT, read 6 scalar ports and the same data as a vector port and compare the results.

1. To run this script, first copy it to your local directory

```
>> copyfile(fullfile('ublaze_lwip_read_write_vector_test.m'),'ublaze_test.m');
```

and open the script in the editor:

```
>> edit('ublaze_test.m');
```

The script has three sections. The first section, connects to the board and sets up the commands that will be used. If required, update the IP address of the board on line 41.

2. Execute section 1. You will have generated the following commands as arrays of uint32 types:

```
read6_cmd      = uint32([hex2dec('44a0010c') 0 6]);   %read 6 32-bit regs
read_vec_cmd   = uint32([hex2dec('44a00140') 0 6]);   %read 6 elements of vec
strobe_vec_cmd = uint32([hex2dec('44a00160') 1 1 1]); %write strobe for vec
enable_cmd     = uint32([hex2dec('44a00004') 1 1 1]); %enable ip core
disable_cmd    = uint32([hex2dec('44a00004') 1 1 0]); %disable ip core
debug0_cmd     = uint32([hex2dec('00000000') 3 0]);   %disable all debug printfs
debug1_cmd     = uint32([hex2dec('00000000') 3 1]);   %enable READ|WRITE printfs
debug2_cmd     = uint32([hex2dec('00000000') 3 2]);   %enable pkt printfs
```

NOTE: these arrays store data in the endian format used by the local machine, which for many x86 systems is the little endian format. However, the TCP/IP server expects values in the big endian format (network byte order). As a result, if the system you are on is little endian, the bytes in each element must be swapped using `swapbytes`.

3. Execute section 2 to disable the DUT logic and read a single counter value connected to the 6 scalar ports as well as all 6 elements in the vector port. Notice that all the counter values match. This is because the same data is driven to all the ports and the DUT is disabled, so the asynchronous access across the AXI4 interface is not apparent.

```
Scalar port (top) vs vector port (bottom) access with DUT disabled:
   7e8aec14   7e8aec14   7e8aec14   7e8aec14   7e8aec14   7e8aec14

   7e8aec14   7e8aec14   7e8aec14   7e8aec14   7e8aec14   7e8aec14
```

4. Execute section 3 to re-enable the DUT logic and read the same counter values back. Notice that the 6 scalar ports all show different values, while the 6 elements of the vector port are all the same. This is due to the sequential access that must occur across the AXI4 interface and the lack of synchronization register in the scalar port case and the presence of an explicit synchronization register in the vector port case.

```
Scalar port (top) vs vector port (bottom) access with DUT enabled:
   7f7796dc   7fc70e4b   8016860a   8065fdce   80b5758d   8104ed4d

   815964dd   815964dd   815964dd   815964dd   815964dd   815964dd
```

The corresponding debug output on the serial console will be:

```
DEBUG | packet payload:
44 A0 00 04
00 00 00 01
00 00 00 01
00 00 00 00
WRITE | address: 0x44A00004, data[0]: 0x00000000
DEBUG | packet payload:
44 A0 01 0C
00 00 00 00
00 00 00 06
READ  | address: 0x44A0010C, data[0]: 0x7E8AEC14
READ  | address: 0x44A00110, data[1]: 0x7E8AEC14
READ  | address: 0x44A00114, data[2]: 0x7E8AEC14
READ  | address: 0x44A00118, data[3]: 0x7E8AEC14
READ  | address: 0x44A0011C, data[4]: 0x7E8AEC14
READ  | address: 0x44A00120, data[5]: 0x7E8AEC14
DEBUG | packet payload:
00 00 00 00
00 00 00 03
00 00 00 00
Debug level set to : 0x00
Debug level set to : 0x01
READ  | address: 0x44A0010C, data[0]: 0x7F7796DC
READ  | address: 0x44A00110, data[1]: 0x7FC70E4B
READ  | address: 0x44A00114, data[2]: 0x8016860A
READ  | address: 0x44A00118, data[3]: 0x8065FDCE
READ  | address: 0x44A0011C, data[4]: 0x80B5758D
READ  | address: 0x44A00120, data[5]: 0x8104ED4D
WRITE | address: 0x44A00160, data[0]: 0x00000001
READ  | address: 0x44A00140, data[0]: 0x815964DD
READ  | address: 0x44A00144, data[1]: 0x815964DD
READ  | address: 0x44A00148, data[2]: 0x815964DD
READ  | address: 0x44A0014C, data[3]: 0x815964DD
READ  | address: 0x44A00150, data[4]: 0x815964DD
READ  | address: 0x44A00154, data[5]: 0x815964DD
```

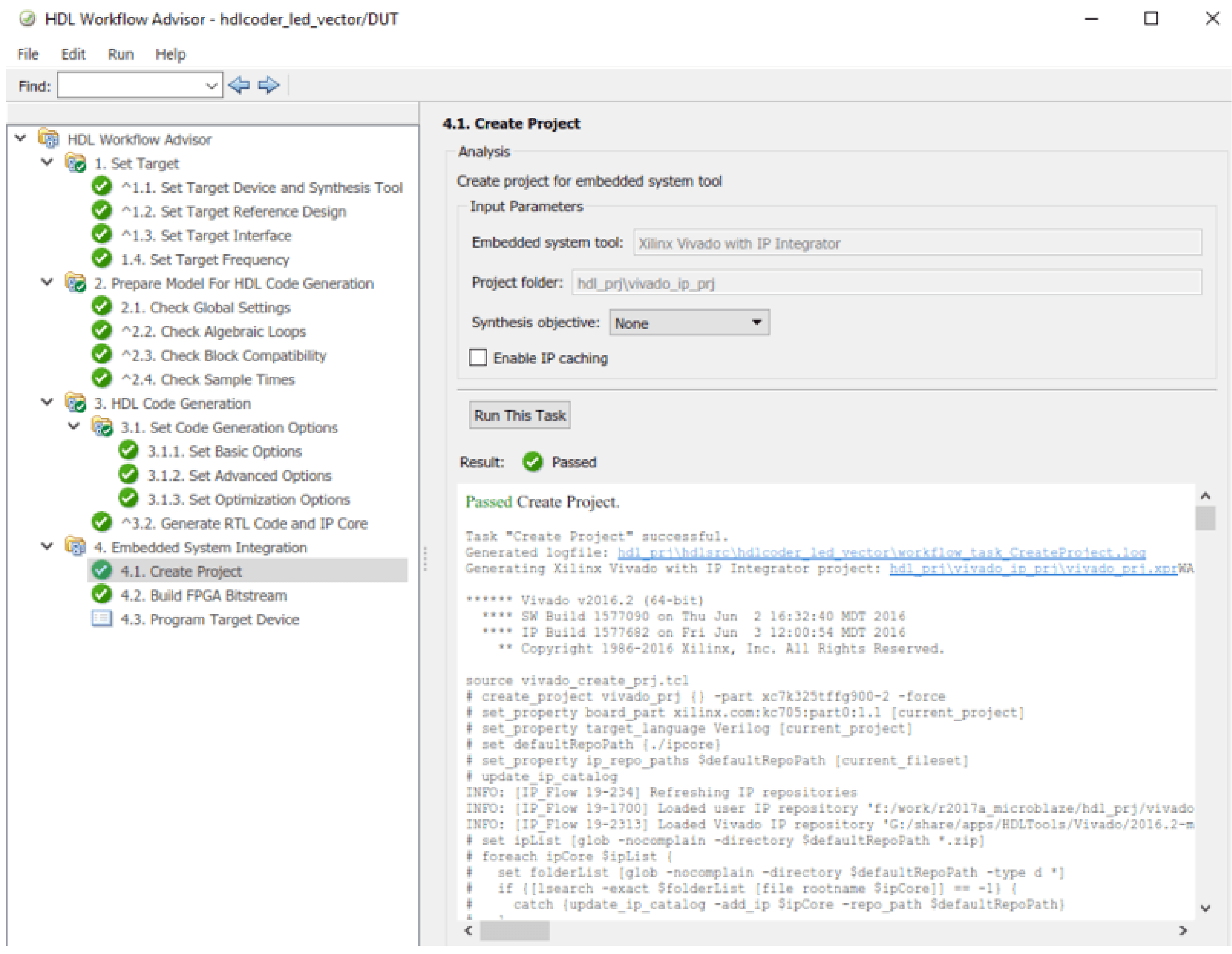
Summary

This demo highlighted the use of a MicroBlaze soft-core processor in FPGA only designs. The MicroBlaze is well suited to function as a full-fledged processor or as a flexible IP running legacy C code as a firmware application. This demo also showed the difference between a collection of scalar ports and a vector port in regards to data synchronization across the AXI4 interface.

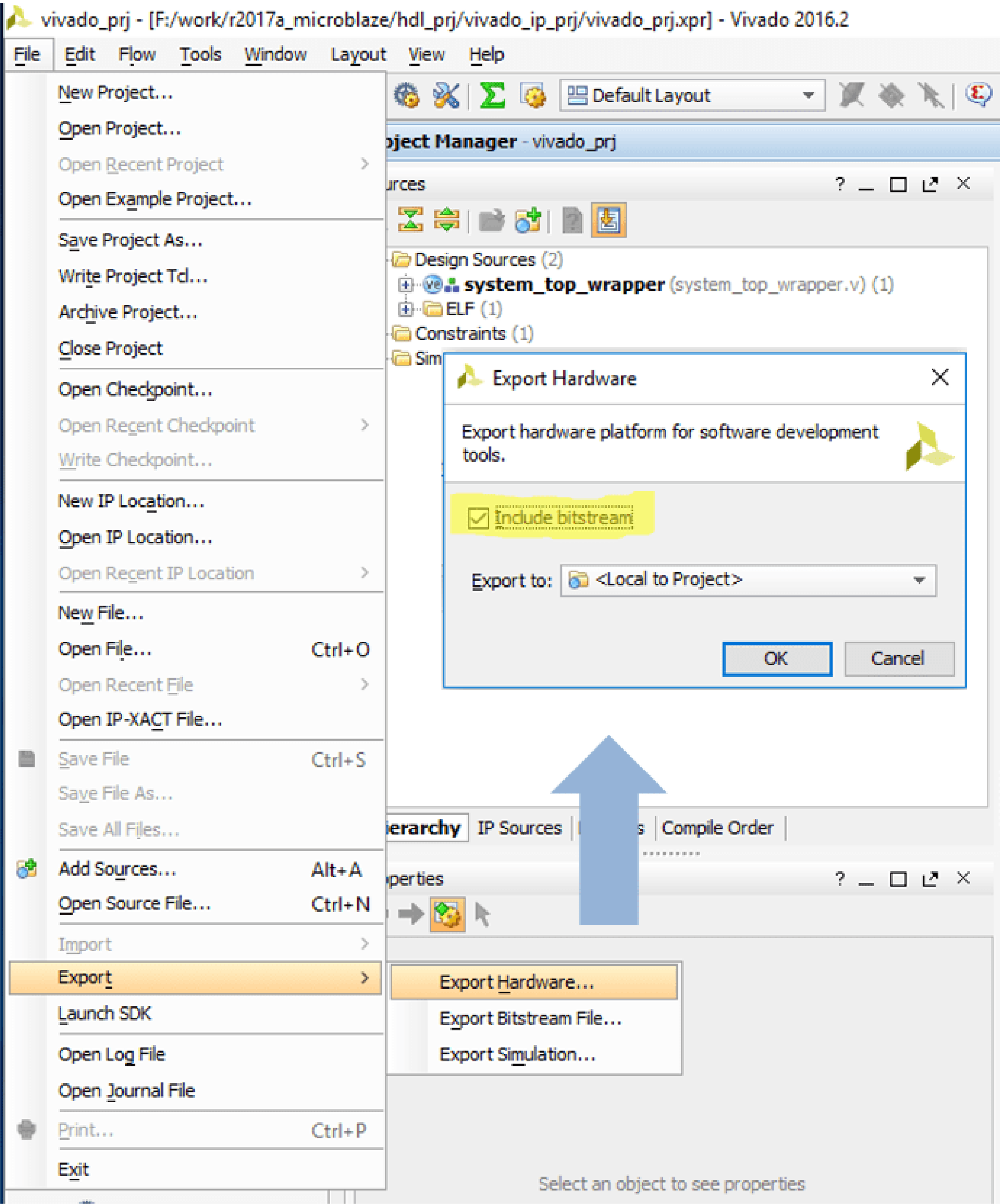
Appendix A: Creating and editing a Xilinx SDK application

This section will show how to create a new Xilinx SDK project and incorporate the code from this example to then modify or extend.

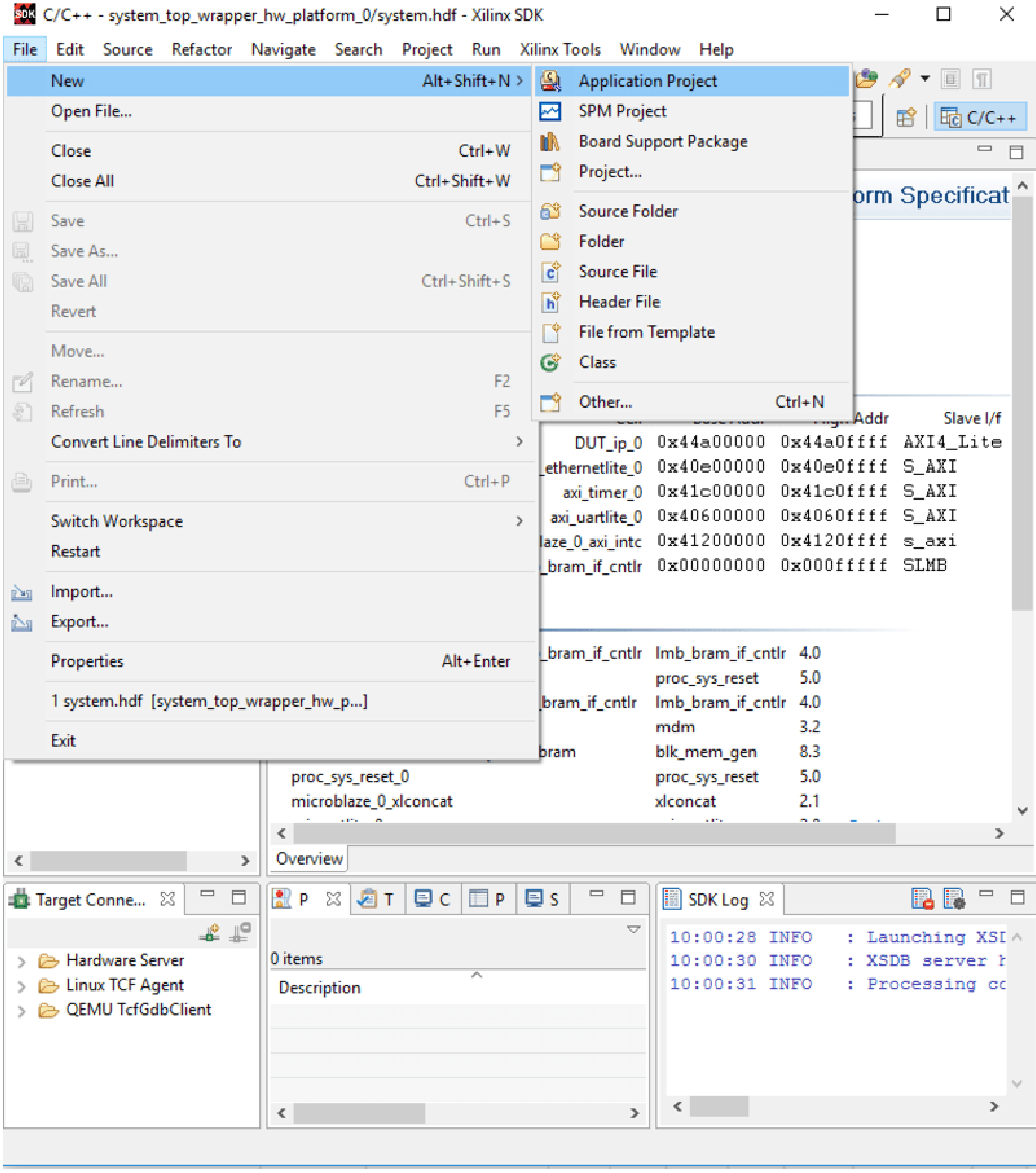
1. Open the Xilinx Vivado project by clicking the link in HDL Workflow Advisor step 4.1 "Create Project" :



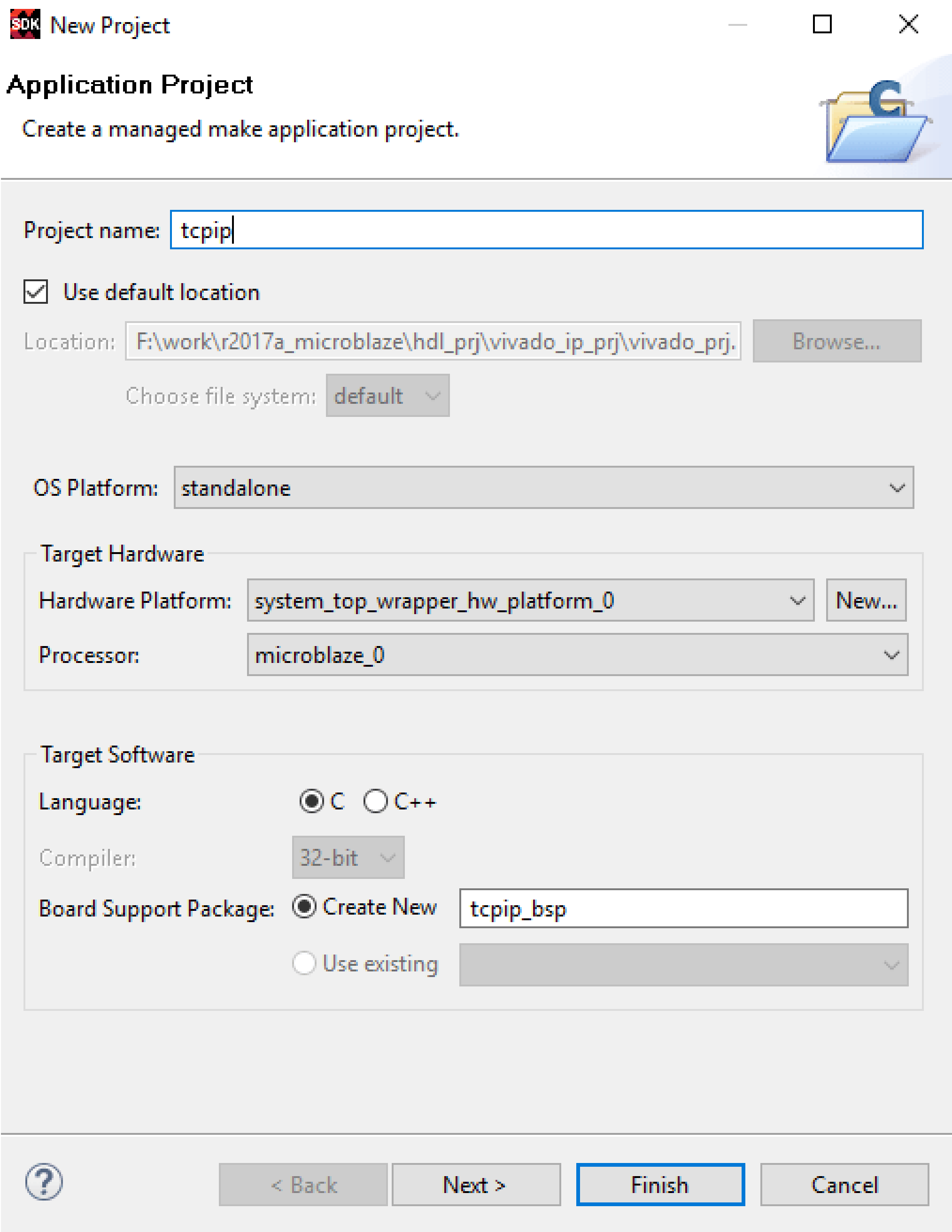
2. Export the existing design, including the generated bitstream, to a local folder. Once this is done, go ahead and "Launch SDK" as well.


vivado_prj - [F:/work/r2017a_microblaze/hdl_prj/vivado_ip_prj/vivado_prj.xpr] - Vivado 2016.2
File
Edit
Flow
Tools
Window
Layout
View
Help
New Project...
Open Project...
Open Recent Project
Open Example Project...
Save Project As...
Write Project Tcl...
Archive Project...
Close Project
Open Checkpoint...
Open Recent Checkpoint
Write Checkpoint...
New IP Location...
Open IP Location...
Open Recent IP Location
New File...
Open File... Ctrl+O
Open Recent File
Open IP-XACT File...
Save File Ctrl+S
Save File As...
Save All Files...
Add Sources... Alt+A
Open Source File... Ctrl+N
Import
Export
Launch SDK
Open Log File
Open Journal File
Print... Ctrl+P
Exit
Default Layout
Design Sources (2)
system_top_wrapper (system_top_wrapper.v) (1)
ELF (1)
Constraints (1)
Export Hardware
Export hardware platform for software development tools.
Include bitstream
Export to: <Local to Project>
OK
Cancel
IP Sources
Compile Order
Export Hardware...
Export Bitstream File...
Export Simulation...
Select an object to see properties

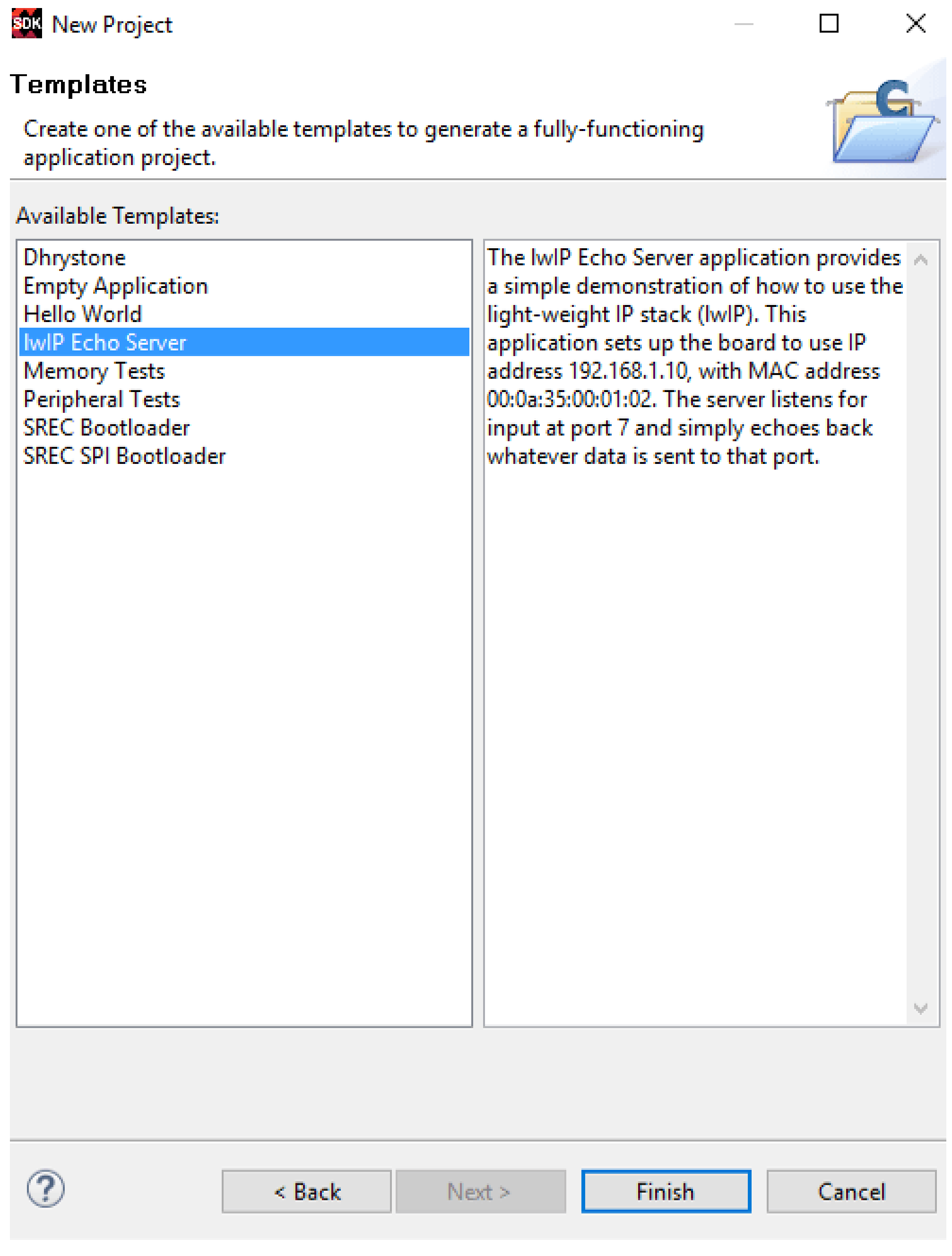
3. From within the SDK, create a new application project



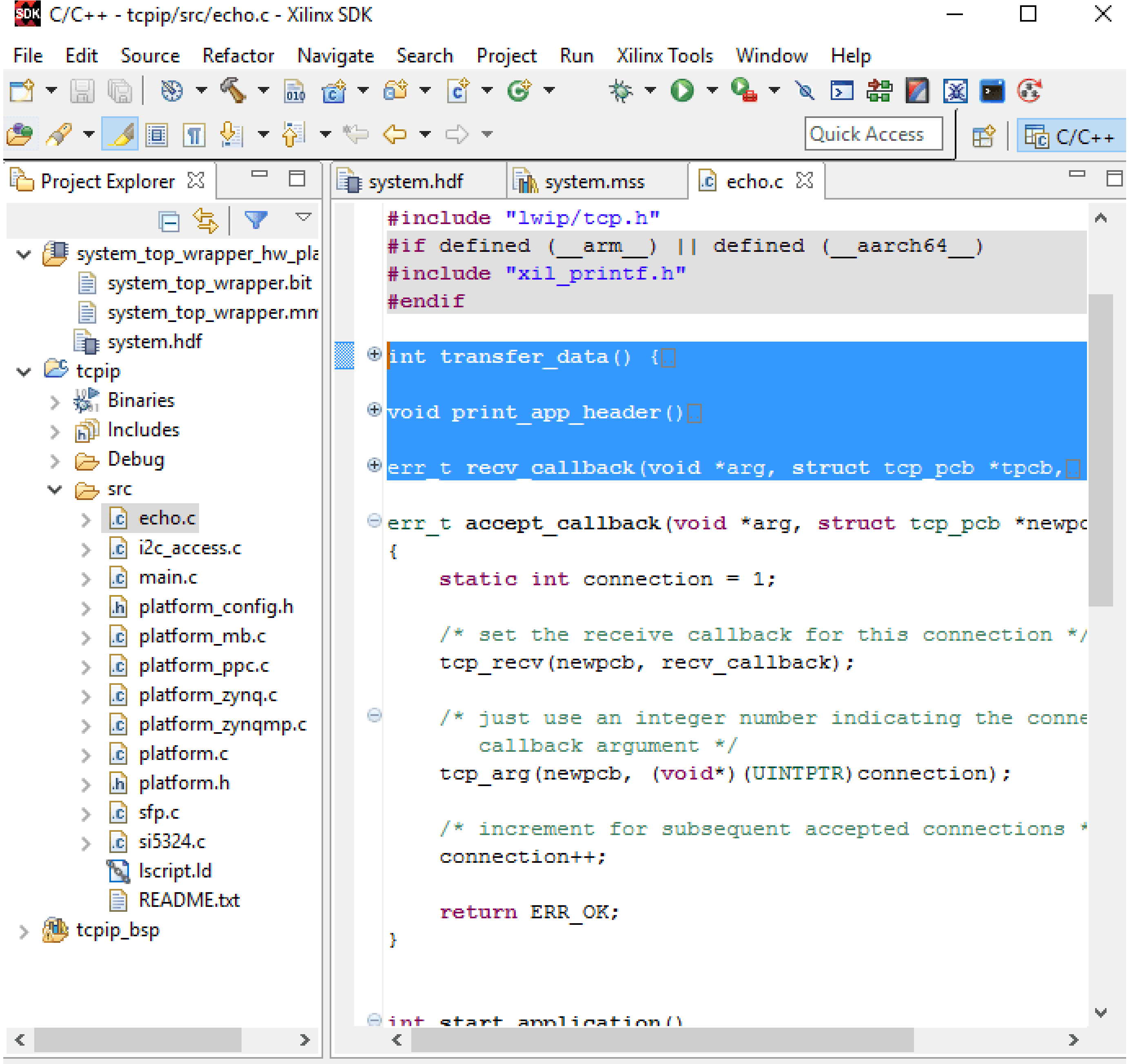
4. You will then have the option of naming the project and creating a new bsp



Next you can choose from a few pre-configured example/template projects to get started. This example is built off of the "lwIP Echo Server" project, so select that now.



5. Using the echo server as a template, you can replace the following 3 methods with the code snippet below to modify the behavior of the server



Appendix B: Copy C file contents to project

```
/* Copyright 2016-2020 The MathWorks, Inc. */
#define IPCOREBASE 0x44a00000
#define WRITE  0x01
#define READ   0x00
#define DEBUG  0x03

int transfer_data() {
    return 0;
```

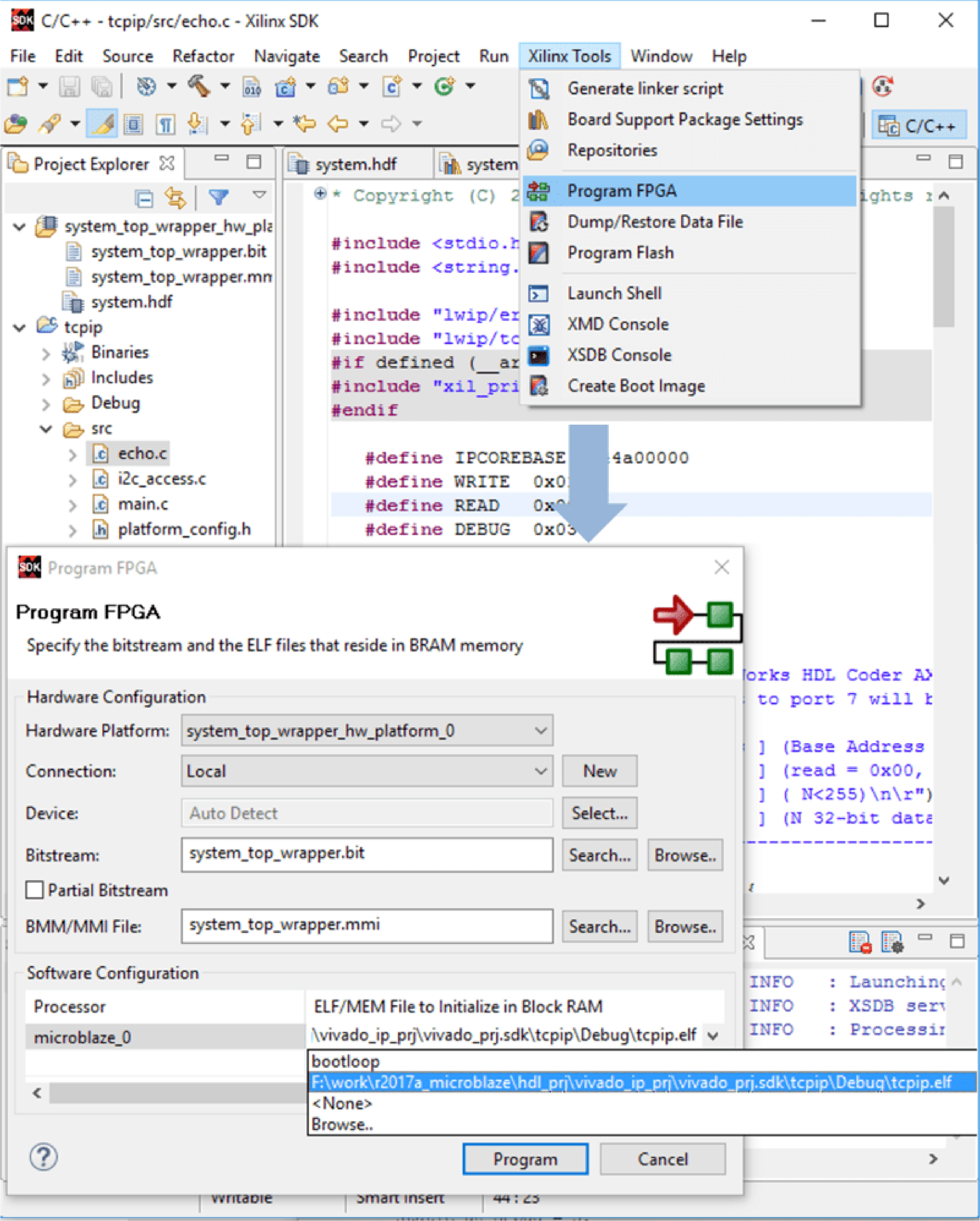
```
}

void print_app_header()
{
    xil_printf("\n\r\n\r-----MathWorks HDL Coder AXI4-Lite IP Core Read/Write Server ------\n\r")
    xil_printf("  TCP packets sent to port 7 will be issued as AXI4-Lite Read/Writes\n\r");
    xil_printf("\n\r");
    xil_printf("  [ 32-bit address ] (Base Address = 0x44a0_0000)\n\r");
    xil_printf("  [ 32-bit   cmd   ] (read = 0x00, write =0x01, debug = 0x03)\n\r");
    xil_printf("  [ 32-bit   len   ] ( N<255)\n\r");
    xil_printf("  [ 32-bit   data  ] (N 32-bit data values for write cmd)\n\r");
    xil_printf("------------------------------------------------------------ ------\n\r");

}

void print_packet(struct pbuf *p) {
    u16 ii;
    u8 *pktPtr;

    pktPtr = p->payload;
    xil_printf("DEBUG | packet payload:\r\n");
    for (ii=0;ii<p->len;ii+=4) {
        xil_printf("%02x %02x %02x %02x\r\n",*(pktPtr+ii),*(pktPtr+ii+1),*(pktPtr+ii+2),*(pktPtr+
    }


}
err_t recv_callback(void *arg, struct tcp_pcb *tpcb,
                               struct pbuf *p, err_t err)
{
    u8 *pktPtr,*pktEnd;
    volatile u32 *addr;
    u32 data[255],cmd;
    u16 len;
    int ii;
    static u8 debug = 3;

    /* do not read the packet if we are not in ESTABLISHED state */
    if (!p) {
        tcp_close(tpcb);
        tcp_recv(tpcb, NULL);
        return ERR_OK;
    }

    /* indicate that the packet has been received */
    tcp_recved(tpcb, p->len);
    if (debug > 1) print_packet(p);

    //[ 32 bits address ]
    //[ 32 bits read = 0x00, write =0x01]
    //[ 32 bits length ]
    //[ 32 bits write data]
    pktPtr = p->payload;
    pktEnd = pktPtr+p->len;


    /* could be multiple commands per packet */
    while ( pktPtr < pktEnd) {
```

```
        addr = (u32*) (pktPtr[0]<<24 | pktPtr[1]<<16 | pktPtr[2]<<8 | pktPtr[3]);
        cmd  = (u32) pktPtr[7];      // cmd is 32 bits, but only 1st byte used, ignore rest
        pktPtr += 8;

        switch(cmd) {
        case WRITE :
            len  = (u32) pktPtr[3];  // len is 32 bits, but only 1st byte used, ignore rest
            pktPtr += 4;
            for (ii=0;ii<len; ii++) {
                data[0] = (u32) (pktPtr[0]<<24 | pktPtr[1]<<16 | pktPtr[2]<<8 | pktPtr[3]);
                *addr = data[0];
                if (debug > 0) xil_printf("WRITE | address: 0x%08x, data[0]: 0x%08x\r\n",addr,da
                addr++;
                pktPtr += 4;
            }
            break;
        case READ :
            len  = (u32)   pktPtr[3]; // len is 32 bits, but only 1st byte used, ignore rest
            pktPtr += 4;
            for (ii=0;ii<len; ii++) {
                data[ii] = *addr;
                if (debug > 0) xil_printf("READ  | address: 0x%08x, data[%d]: 0x%08x\r\n",addr,i
                addr++;
            }
            /* send the packet back */
            if (tcp_sndbuf(tpcb) > p->len)
                err = tcp_write(tpcb, data, 4*len, 1);
            else
                xil_printf("no space in tcp_sndbuf\n\r");
            break;
        case DEBUG:
            debug = pktPtr[3]; // only need the low byte
            pktPtr += 4;
            xil_printf("Debug level set to : 0x%02x\r\n",debug);
            break;
        default :
            xil_printf("INVALID | cmd: 0x%08x\r\n",cmd);

        }
    }
        /* free the received pbuf */
        pbuf_free(p);

        return ERR_OK;
}
```

6. Save the modified echo.c file and the application will be rebuilt.

Appendix C: Program the FPGA with ELF and bitstream

Now, you can program the FPGA using the exported bitstream and the newly created ELF file.


C/C++ - tcpip/src/echo.c - Xilinx SDK
File
Edit
Source
Refactor
Navigate
Search
Project
Run
Xilinx Tools
Window
Help
Generate linker script
Board Support Package Settings
Repositories
Program FPGA
Dump/Restore Data File
Program Flash
Launch Shell
XMD Console
XSDB Console
Create Boot Image
Project Explorer
system_top_wrapper.bit
system.hdf
tcpip
Binaries
Includes
Debug
src
echo.c
i2c_access.c
main.c
platform_config.h
Program FPGA
Specify the bitstream and the ELF files that reside in BRAM memory
Hardware Configuration
Hardware Platform:
system_top_wrapper_hw_platform_0
Connection:
Local
New
Device:
Auto Detect
Select...
Bitstream:
system_top_wrapper.bit
Search...
Browse..
Partial Bitstream
BMM/MMI File:
system_top_wrapper.mmi
Software Configuration
Processor
ELF/MEM File to Initialize in Block RAM
microblaze_0
\vivado_ip_prj\vivado_prj.sdk\tcpip\Debug\tcpip.elf
bootloop
F:\work\r2017a_microblaze\hdl_prj\vivado_ip_prj\vivado_prj.sdk\tcpip\Debug\tcpip.elf
<None>
Browse..
Program
Cancel

Map Bus Data Types to AXI4 Slave Interfaces

This example shows how to map bus data types to an AXI4 slave interface, generate an HDL IP core with a AXI4 Master interface, perform matrix multiplication in an HDL IP core, and write the output result to DDR memory. In this example, you:

1. Create a bus element by using bus creator blocks and map the bus element to an AXI4 slave interface.
2. Generate an HDL IP core with AXI4 Master interface.
3. Access large matrices from the external DDR4 memory on the Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit board using the AXI4 Master interface.
4. Perform matrix vector multiplication in the HDL IP core and write the output result back to the DDR memory using the AXI4 Master interface.

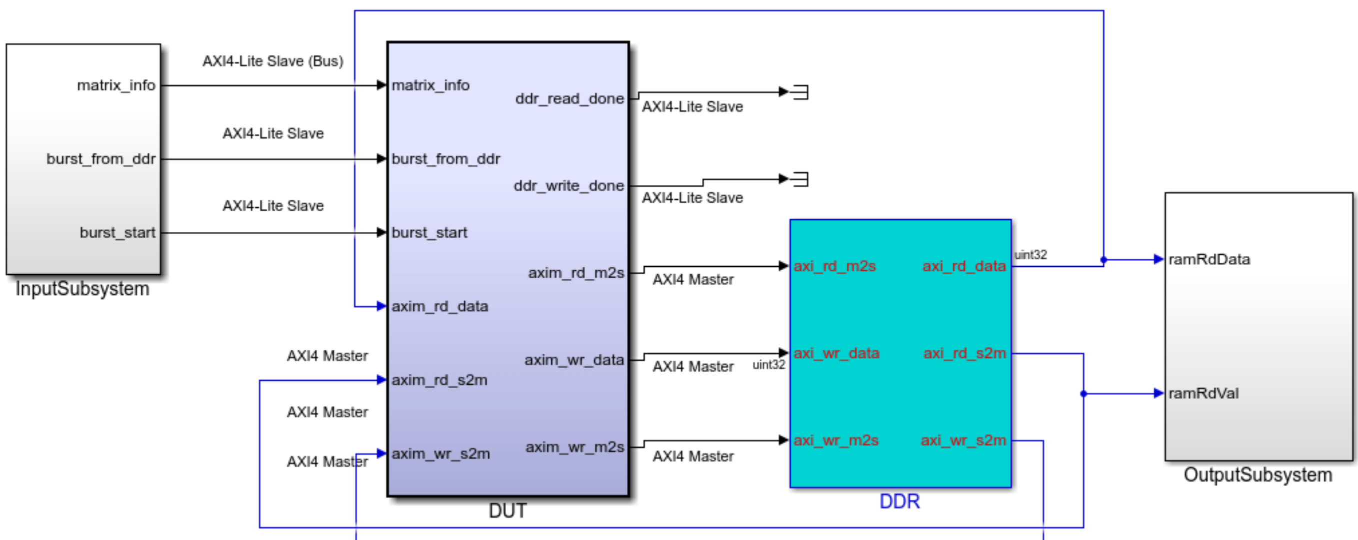
Requirements

For this example, you must have the following software and hardware installed and set up:

- Xilinx Vivado Design Suite. To see the supported versions, see "HDL Language Support and Supported Third-Party Tools and Hardware"
- Xilinx Zynq Ultrascale+ MPSoC ZCU102 evaluation kit.
- HDL Coder Support Package for Xilinx FPGA and SoC Devices.
- HDL Verifier Support Package for Xilinx FPGA Boards.

Open the Model

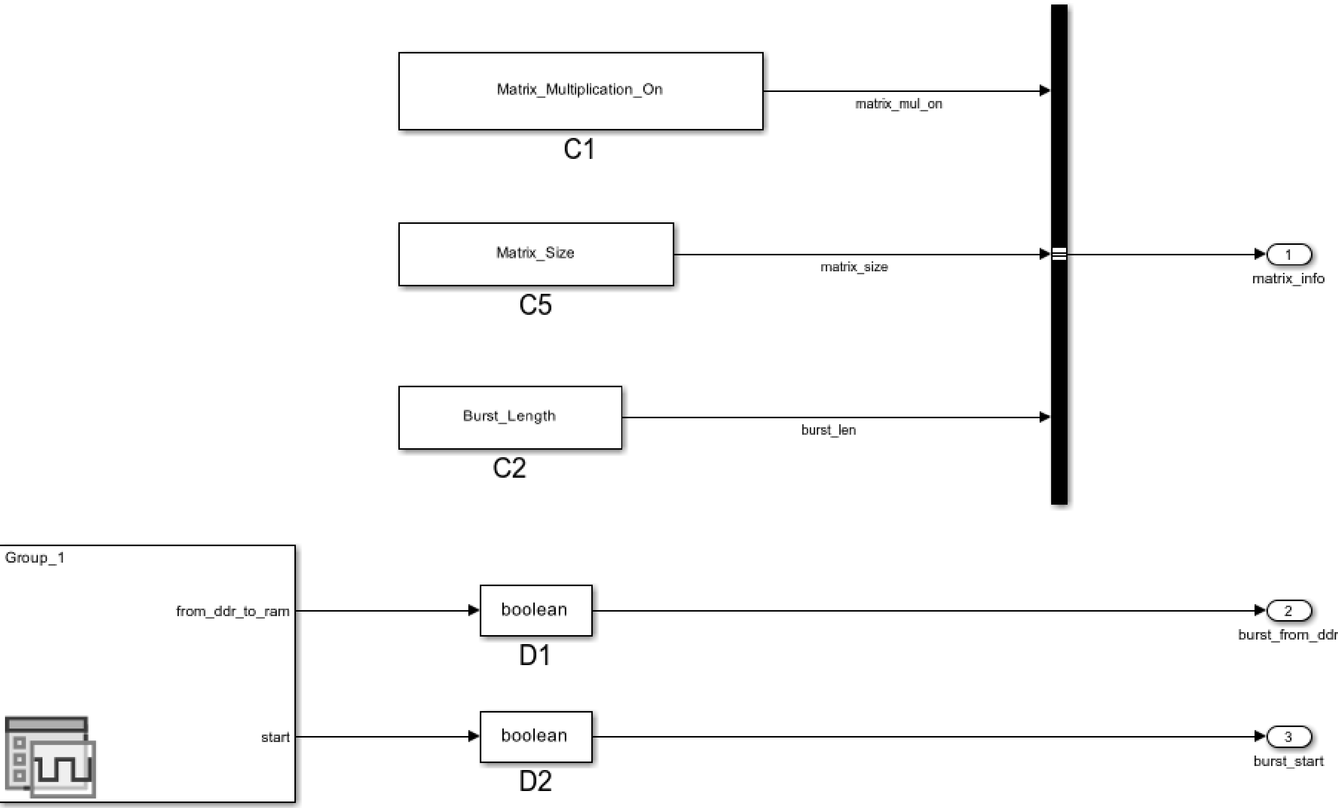
```
%
% This example models a matrix vector multiplication algorithm and
% implements it on the Xilinx Zynq FPGA board. Large matrices
% might not map efficiently to Block RAMs on the FPGA fabric. Instead, store
% the matrices in the external DDR memory on the FPGA board. The
% AXI4 Master interface can access the data by communicating with
% vendor-provided memory interface IP cores that interface with the DDR
% memory. This capability enables you to model algorithms that involve
% large data processing and require high-throughput DDR access, such as
% matrix operations, computer vision algorithms, and so on.
%
% The matrix vector multiplication module supports fixed-point matrix
% vector multiplication with a configurable matrix size of 2 to 4000.
% The size of the matrix is run-time configurable through the AXI4 accessible
% register.

modelname = 'hdlcoder_axi_slave_bus_data_type';
open_system(modelname);

%
% Use HDL Coder to generate a custom IP core that performs large matrix
% operations on FPGAs by using external memory. In MATLAB, type:
%   hdladvisor('hdlcoder_axi_slave_bus_data_type/DUT')
```



Create Bus Element

Use a Bus Creator block to combine the `Matrix_Multiplication_On`, `Matrix_Size`, and `Burst_Length` signals into a bus. For more information on the Bus Creator block, see Bus Creator

```
open_system('hdlcoder_axi_slave_bus_data_type/InputSubsystem')
```



Verify Simulation Results

You can simulate this example model and verify the simulation results by running this script in MATLAB:

```
hdlcoder_axi_slave_bus_data_type_simulation;
```

This script first initializes the parameters like `Matrix_Size`. By default, the `Matrix_Size` is 64, which generates a 64-by-64 matrix. This default value is kept small to ensure the model simulates faster. After the DUT is implemented on the FPGA board, the script uses a larger value for `Matrix_Size` because the FPGA calculation is much faster. You can also adjust these parameters in the script.

The script then simulates the model and verifies the result by comparing the logged simulation result to the expected value.

```
>> hdlcoder_axi_slave_bus_data_type_simulation
PASSED: DDR initialization data matches.
PASSED: Matrix vector multiplication output matches with the expected data
>>
```

By default, the `Matrix_Multiplication_On` signal is true. The script verifies the matrix vector multiplication result.

When `Matrix_Multiplication_On` is false, the script verifies the loop back mode, which means that the DUT reads `Burst_Length` amount of data from DDR, and then writes the data back to DDR.

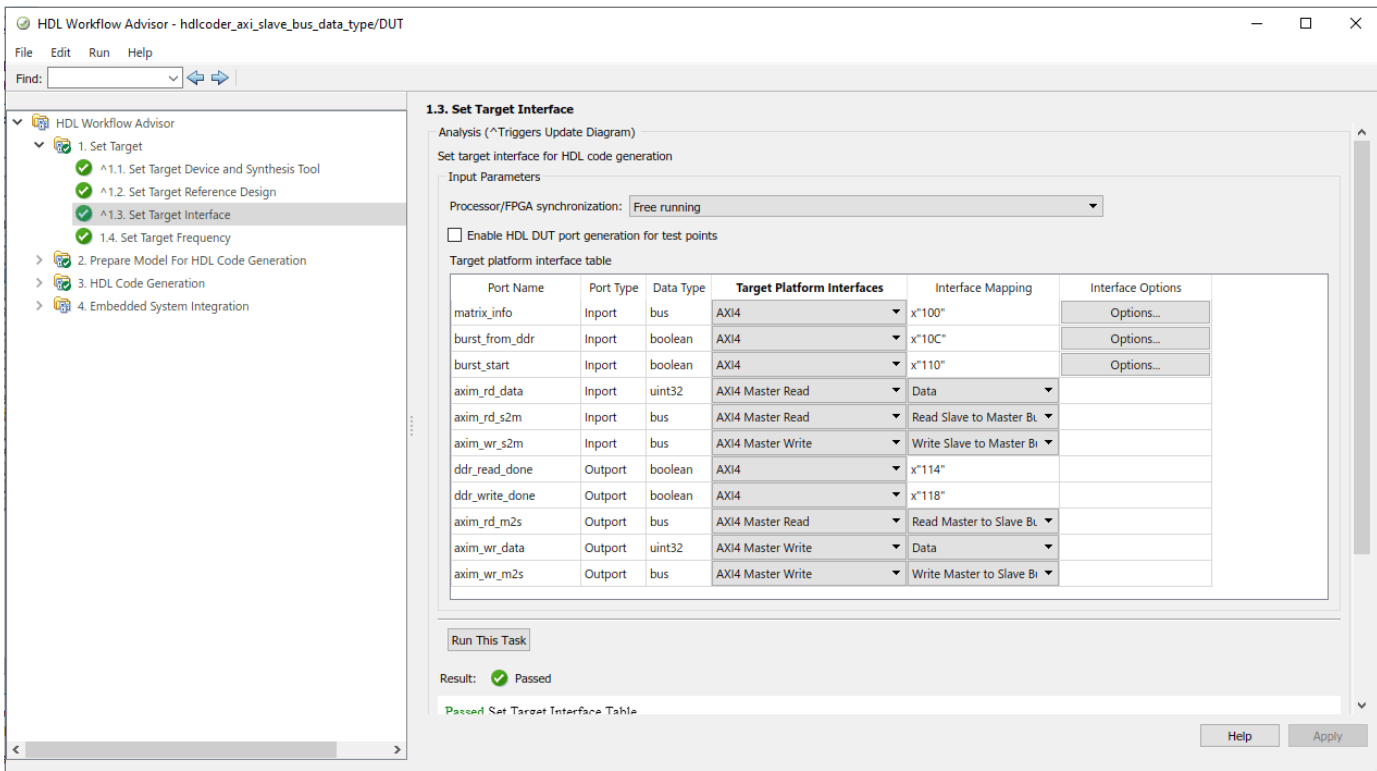
Generate HDL IP Core with AXI4 Master Interface

Next, start the HDL Workflow Advisor and deploy the DUT on the Zynq hardware. For a more detailed step-by-step guide, see “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98 example.

1. Set up the Xilinx Vivado synthesis tool path. Use your own Vivado installation path when you run the command. In the MATLAB Command Window, enter:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.1\bin\vivado.ba
```

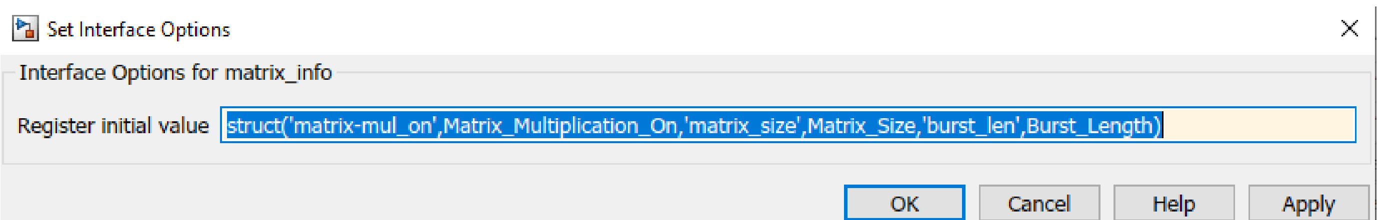
2. Start the HDL Workflow Advisor from the DUT subsystem `hdlcoder_axi_slave_bus_data_type/DUT`. The target interface settings are saved in the model. The **Target workflow** parameter is `IP Core Generation`, and the **Target platform** parameter is `Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit`. The **Reference Design** parameter is |Default System with External DDR4 memory access. This image shows the **Target platform interface table** settings.



In this example, the bus data type ports such as `matrix_info` and input parameter ports such as `burst_from_ddr` and `burst start` are mapped to the AXI4 interface. HDL Coder generates the AXI4 interface-accessible registers for these ports. Later, you can use MATLAB to tune these parameters at run time when the design is running on the FPGA board.

You can specify the bus data type initial values in the **Target platform interface table** by specifying either the initial values or by creating a variable to store the initial values in a structure and using the variable name in the Set Interface Options window. To specify the initial value, click `Options` under the **Interface Options** column in the **Target platform interface table**.

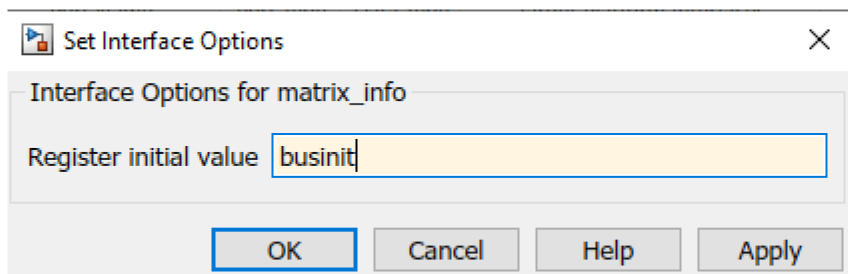
Alternatively, specify the initial values in the Set Interface Options window.



To store the bus data type initial values in a variable called `businit`, run this code:

```
businit = struct('matrix_mul_on',Matrix_Multiplication_On,'matrix_size',Matrix_Size,'burst_len',
```

Then specify the `businit` variable in the Set Interface Options window.



The AXI4 Master interface has separate read and write channels. The read channel ports such as `axim_rd_data`, `axim_rd_s2m`, `axim_rd_m2s` are mapped to the AXI4 Master Read interface. The write channel ports such as `axim_wr_data`, `axim_wr_s2m`, `axim_wr_m2s` are mapped to the AXI4 Master Write interface.

3. Right-click **Generate RTL Code and IP Core** and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP core report.

The report shows the individual bus elements and their address mappings.

Target Interface Configuration

You chose the following target interface configuration for hdlcoder_axi_slave_bus_data_type :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
matrix_info	Inport	bus	AXI4		RegisterInitialValue 0
> matrix_info.matrix_mul_on	Bus element	boolean	AXI4	x"100"	
> matrix_info.matrix_size	Bus element	uint32	AXI4	x"104"	
> matrix_info.burst_len	Bus element	uint32	AXI4	x"108"	
burst_from_ddr	Inport	boolean	AXI4	x"10C"	RegisterInitialValue 0
burst_start	Inport	boolean	AXI4	x"110"	
axim_rd_data	Inport	uint32	AXI4 Master Read	Data	
axim_rd_s2m	Inport	bus	AXI4 Master Read	Read Slave to Master Bus	
axim_wr_s2m	Inport	bus	AXI4 Master Write	Write Slave to Master Bus	
ddr_read_done	Outport	boolean	AXI4	x"114"	
ddr_write_done	Outport	boolean	AXI4	x"118"	
axim_rd_m2s	Outport	bus	AXI4 Master Read	Read Master to Slave Bus	
axim_wr_data	Outport	uint32	AXI4 Master Write	Data	
axim_wr_m2s	Outport	bus	AXI4 Master Write	Write Master to Slave Bus	

If you specify the initial value for the bus element using the `businit` variable, the generated IP core report shows the bus element initial values.

Target Interface Configuration

You chose the following target interface configuration for hdlcoder_axi_slave_bus_data_type :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
matrix_info	Inport	bus	AXI4		RegisterInitialValue businit
> matrix_info.matrix_mul_on	Bus element	boolean	AXI4	x"100"	RegisterInitialValue 1
> matrix_info.matrix_size	Bus element	uint32	AXI4	x"104"	RegisterInitialValue 64
> matrix_info.burst_len	Bus element	uint32	AXI4	x"108"	RegisterInitialValue 4160
burst_from_ddr	Inport	boolean	AXI4	x"10C"	
burst_start	Inport	boolean	AXI4	x"110"	
axim_rd_data	Inport	uint32	AXI4 Master Read	Data	
axim_rd_s2m	Inport	bus	AXI4 Master Read	Read Slave to Master Bus	
axim_wr_s2m	Inport	bus	AXI4 Master Write	Write Slave to Master Bus	
ddr_read_done	Outport	boolean	AXI4	x"114"	
ddr_write_done	Outport	boolean	AXI4	x"118"	
axim_rd_m2s	Outport	bus	AXI4 Master Read	Read Master to Slave Bus	
axim_wr_data	Outport	uint32	AXI4 Master Write	Data	
axim_wr_m2s	Outport	bus	AXI4 Master Write	Write Master to Slave Bus	

4. Right-click **Build FPGA Bitstream** and select **Run to Selected Task** to generate the Vivado project. Build the FPGA bitstream.

During the project creation, the generated DUT IP core is integrated into the `Default System with External DDR4 Memory Access` reference design. This reference design comprises of a Xilinx Memory Interface Generator IP that communicates with the onboard external DDR4 memory on the ZCU102 platform. The AXI Manager IP is also added to enable MATLAB to control the DUT IP and to initialize and verify the DDR memory content.

Run FPGA Implementation on Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit

After the FPGA bitstream is generated, run the **Program Target Device** task to program the FPGA board through the JTAG cable.

You can then run the FPGA implementation and verify the hardware result by running this script in MATLAB:

```
hdlcoder_axi_slave_bus_data_type_hw_run_ZCU102.m
```

This script first initializes the `Matrix_Size` variable to 2000, which generates a 2000-by-2000 matrix. You can adjust the `Matrix_Size` up to 4000.

The script then configures the AXI4 Master read and write channel base addresses. These addresses define the base address that the DUT reads from the external DDR memory, and then writes to external DDR memory. In this script, the DUT reads from base address '80000000' and writes to the base address '90000000'.

The AXI Manager initializes the external DDR4 memory with input vector and matrix data and clears the output DDR memory location.

The DUT calculation starts by controlling the AXI4 accessible registers. The DUT IP core reads input data from the DDR memory, performs the matrix vector multiplication, and then writes the result back to the DDR memory.

The output result is read back to MATLAB and compared to the expected value. The hardware results are verified in MATLAB.

```
>> hdlcoder_axi_slave_bus_data_type_hw_run_ZCU102
Initializing external DDR4 memory (data size 4002000) ...
Starting DUT IP core processing ...
Verifying result ...
PASSED: Matrix vector multiplication output matches with the expected data.
```

See Also

More About

- “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98
- “Map Bus Data Types to AXI4 Slave Interface” on page 40-8

Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands

This example shows how to use MATLAB® to prototype an algorithm running on FPGA hardware from your host computer. The example demonstrates the workflow on the Versal AI Core Series VCK190 Evaluation Kit, but can be run on any Xilinx Zynq, Zynq UltraScale+, or Versal board supported by HDL Coder.

Introduction

At many stages in the design process it can be useful to interact with an FPGA design that is running directly on hardware. Working with hardware enables you to rapidly prototype designs, verify functionality, tune key parameters, connect to real-world signals, collect data for analysis, and much more.

This example shows you how to connect MATLAB on your host computer to your FPGA hardware. Use MATLAB to:

- Write input signals to your FPGA algorithm.
- Capture output signals from your FPGA for analysis.
- Read from and write to registers in your FPGA design.

As part of this example, you:

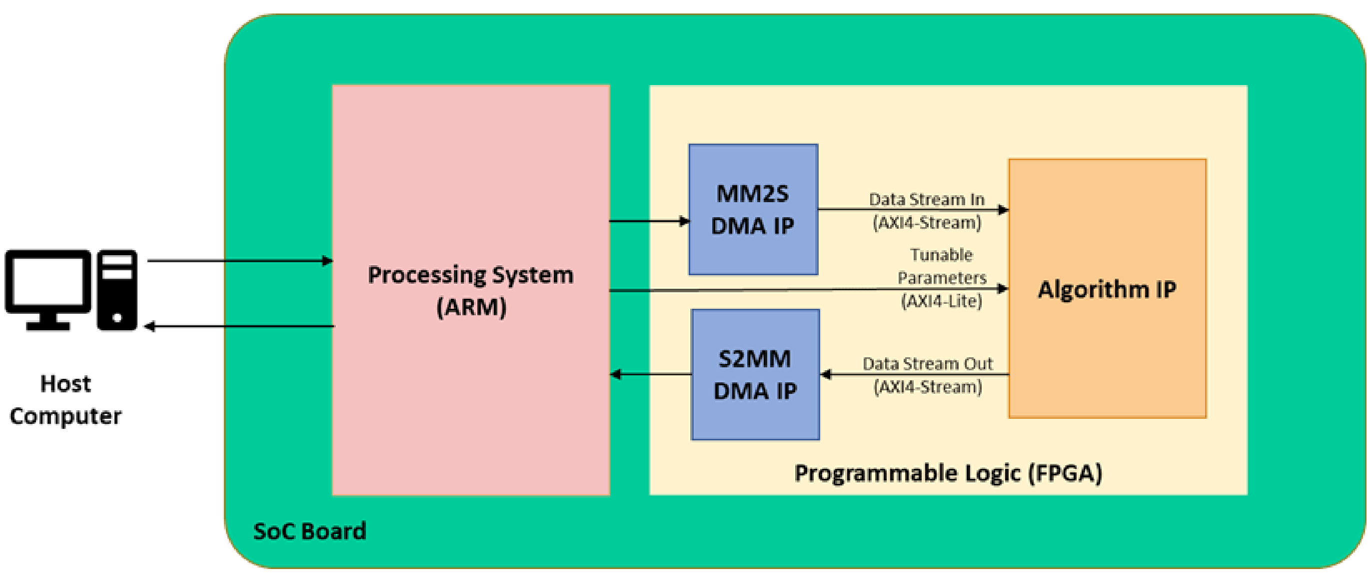
1. Generate and deploy a simple algorithm on hardware.
2. Create a hardware object to establish a connection to your FPGA.
3. Use a simple script to prototype the design running on hardware with live data.

Before You Begin

To run this example, install and set up:

- HDL Coder™ Support Package for Xilinx® FPGA and SoC Devices
- Xilinx Vivado® (version indicated in “HDL Language Support and Supported Third-Party Tools and Hardware”)
- Any Xilinx SoC board supported by HDL Coder. This example uses the Versal AI Core Series VCK190 Evaluation Kit.
- To setup the Versal, see Set Up Xilinx Versal ACAP Hardware and Tools in “Getting Started with Targeting Xilinx Versal Adaptive SoC Platform” on page 39-265.
- MathWorks® firmware image on the board’s SD card. For help with SD card setup, see “Guided Hardware Setup” (HDL Coder Support Package for Xilinx FPGA and SoC Devices).

System Architecture



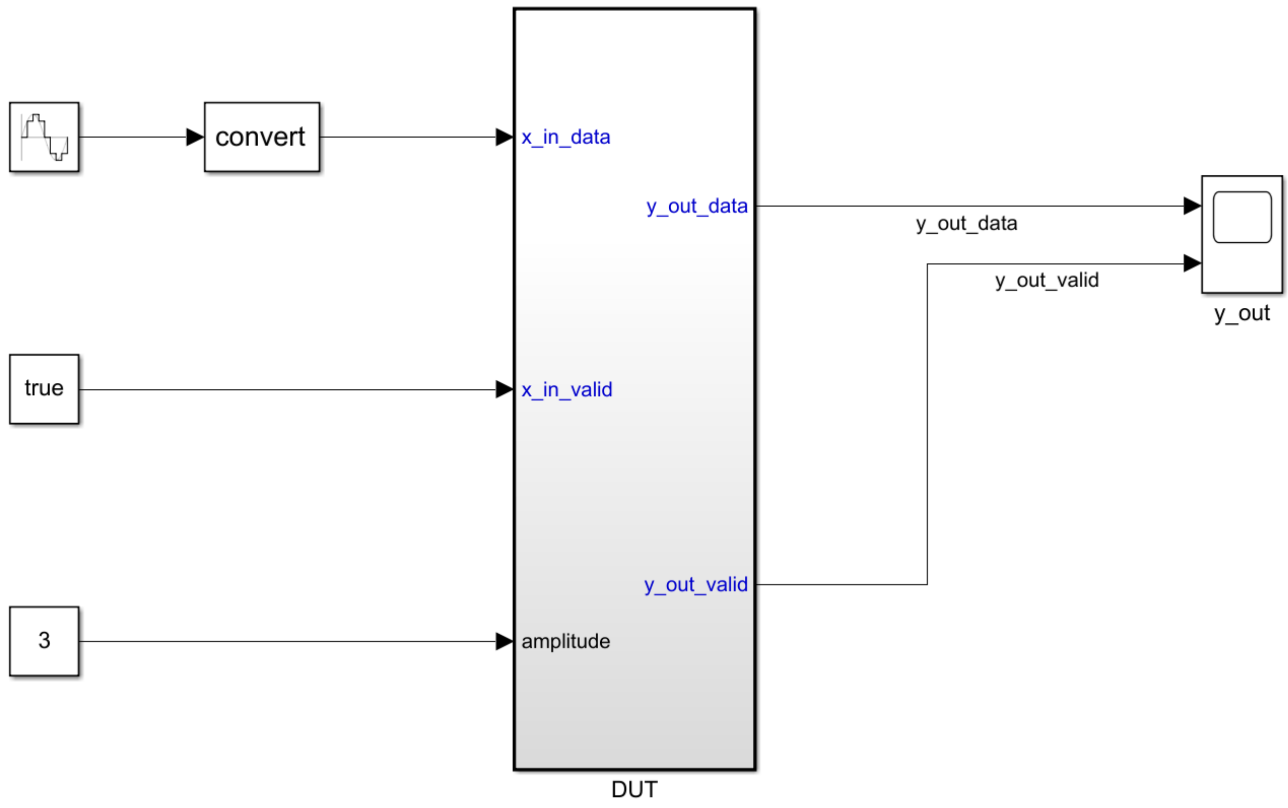
The preceding image shows the high-level architecture of the system. The host computer communicates to the FPGA through the processing system on the System on Chip (SoC) board. The host computer can send and receive frames of data, which are translated to and from streaming data by the Direct Memory Access (DMA) IPs. The host computer can also tune parameters by writing to AXI4-Lite registers within the algorithm IP core.

FPGA Algorithm

The algorithm deployed to the FPGA is a simple streaming algorithm that scales the amplitude of the input signal by a constant. The streaming data is modeled with data and valid signals. The amplitude signal is modeled as a constant.

Open the model. The model consists of the design under test (DUT) and the testbench. The DUT contains the algorithm that is deployed to the FPGA. The testbench exercises the DUT during simulation by providing inputs and capturing outputs for display.

```
% open_system hdlcoder_scale_amplitude.slx
```



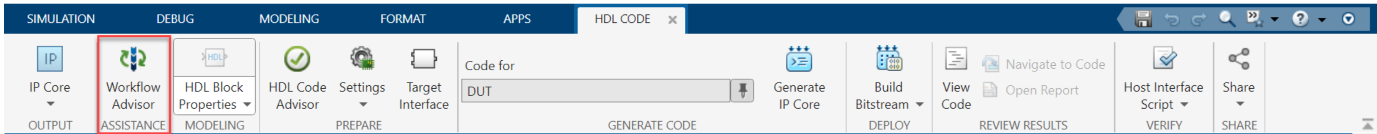
Generate HDL IP Core

To generate an IP core from the DUT by using the HDL Workflow Advisor:

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB Command Window. Use your own Vivado installation path when you run the command.

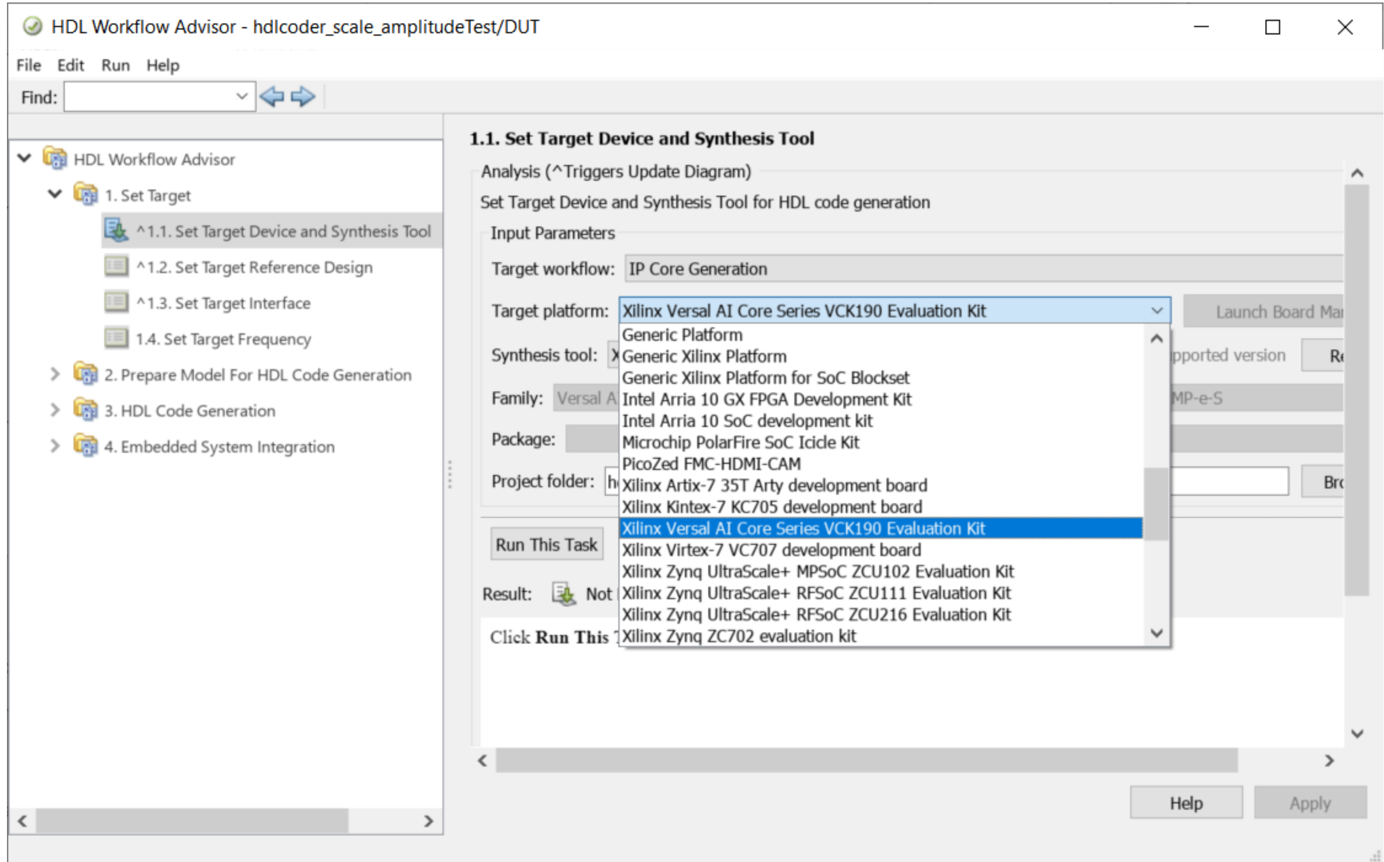
```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

2. Open the HDL Coder toolstrip app from **Apps > HDL Coder**. Click the toolstrip icon to open the Workflow Advisor.



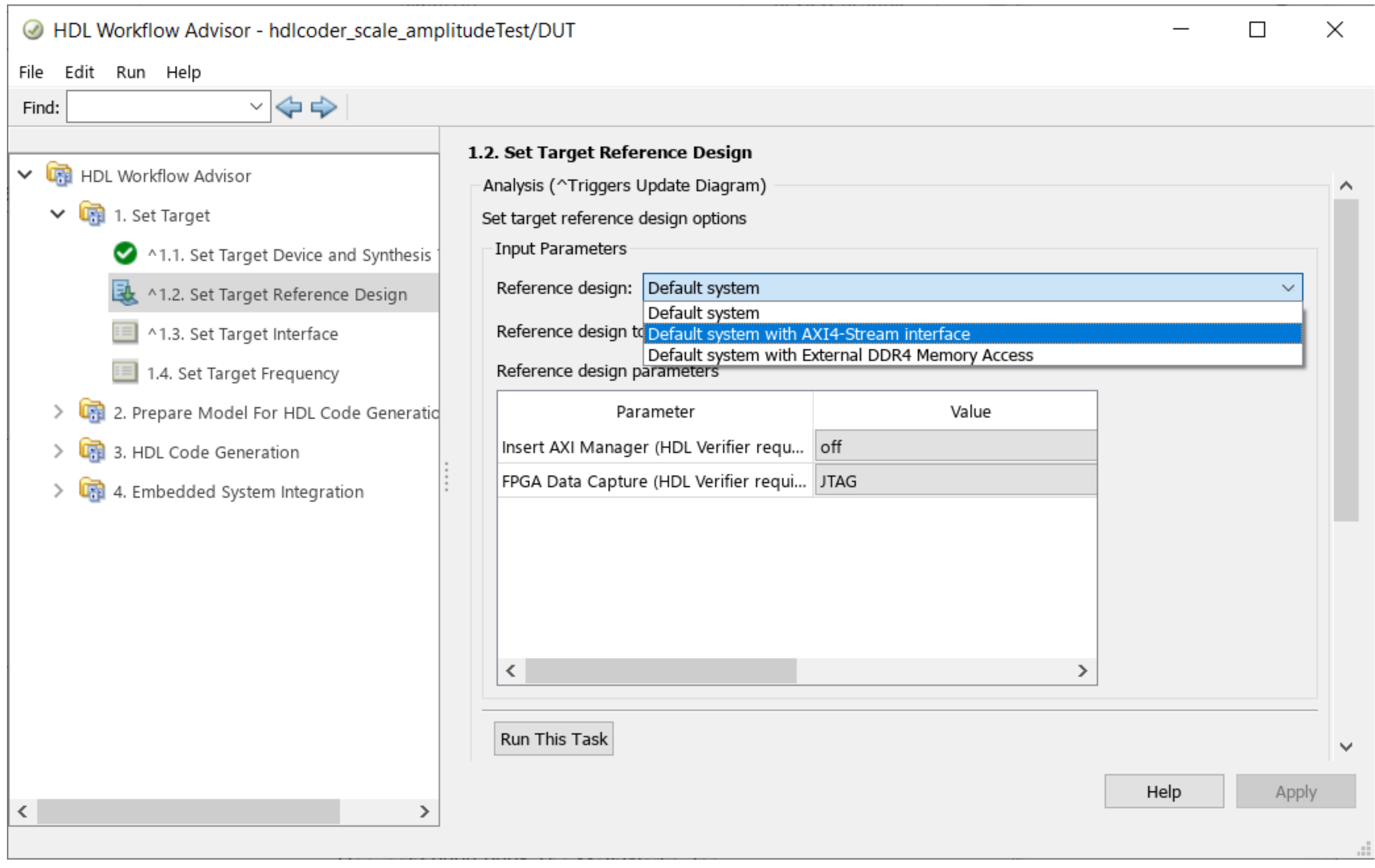
3. In the **Set Target Device and Synthesis Tool** task, select **IP Core Generation** for **Target workflow** and **Xilinx Versal AI Core Series VCK190 Evaluation Kit** for **Target platform**. If you are targeting a different Xilinx SoC, choose your board from the Target platform context menu.

Click **Run This Task**.



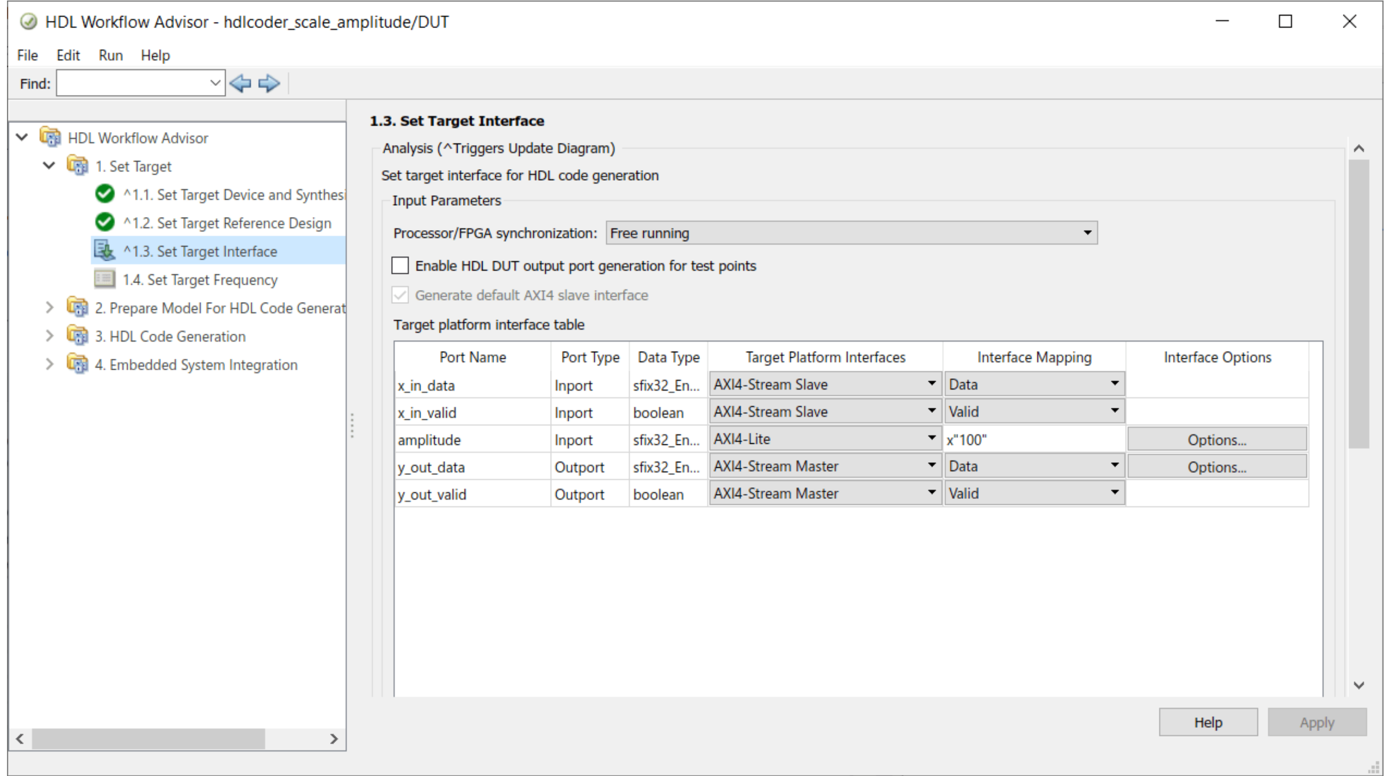
4. In the **Set Target Reference Design** task, select **Default system with AXI4-Stream interface** for **Reference design**.

Click **Run This Task.**

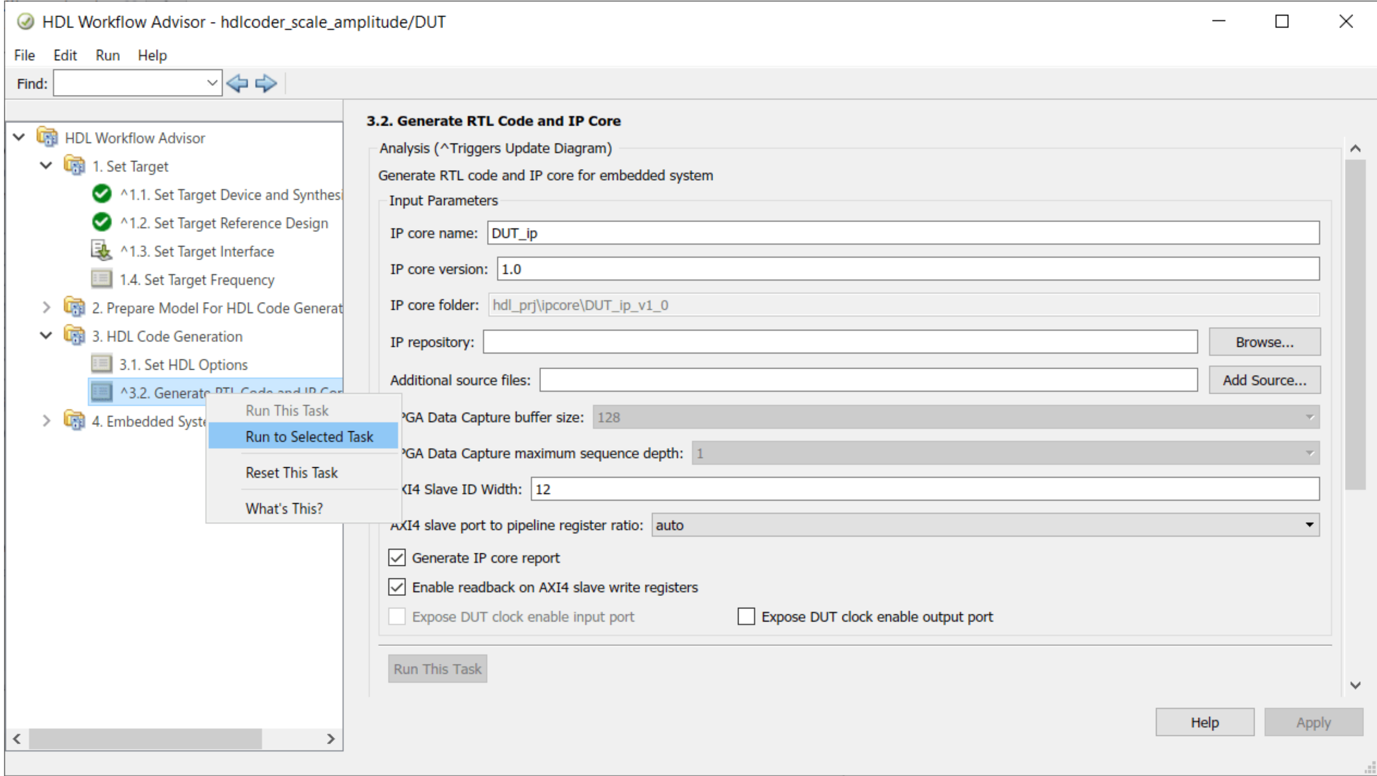


5. In **Set Target Interface** task, the ports of the **DUT** subsystem are mapped to IP Core interfaces. The input data and valid ports are mapped to **AXI4-Stream Slave**. The output data and valid ports are mapped to **AXI4-Stream Master**. The amplitude signal is mapped to **AXI4-Lite**.

Click **Run This Task**.



6) Right-click the **Generate RTL Code and IP Core** task, and select **Run to Selected Task** to generate the IP core.

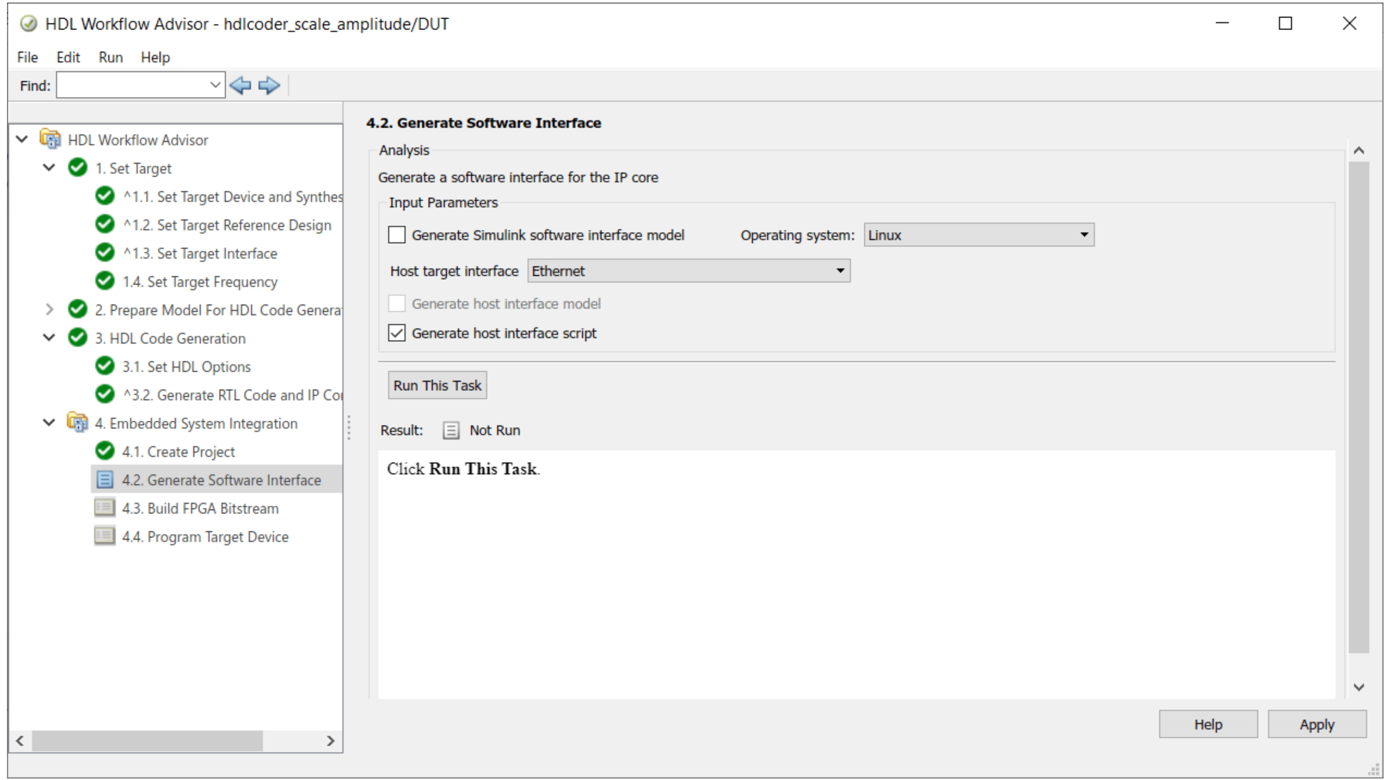


Generate Interface Between Host Computer and IP Core

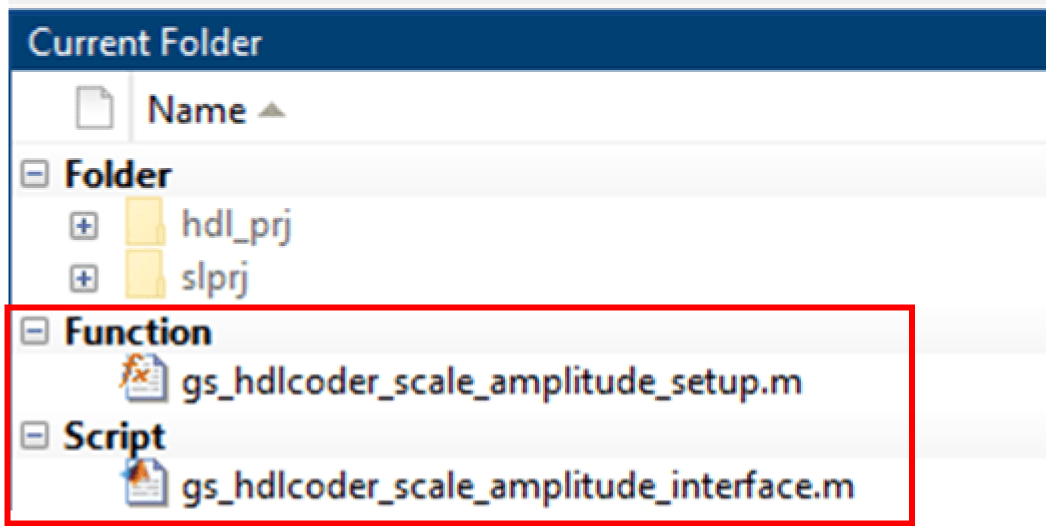
To generate a host computer interface to the IP core and deploy the design to the target hardware board:

1. Run the **Create Project** task. This task inserts the generated IP core for the FPGA algorithm into the reference design to create the system shown in the System Architecture diagram.

2. In the **Generate Software Interface** task, select the box for **Generate host interface script** then run this task.

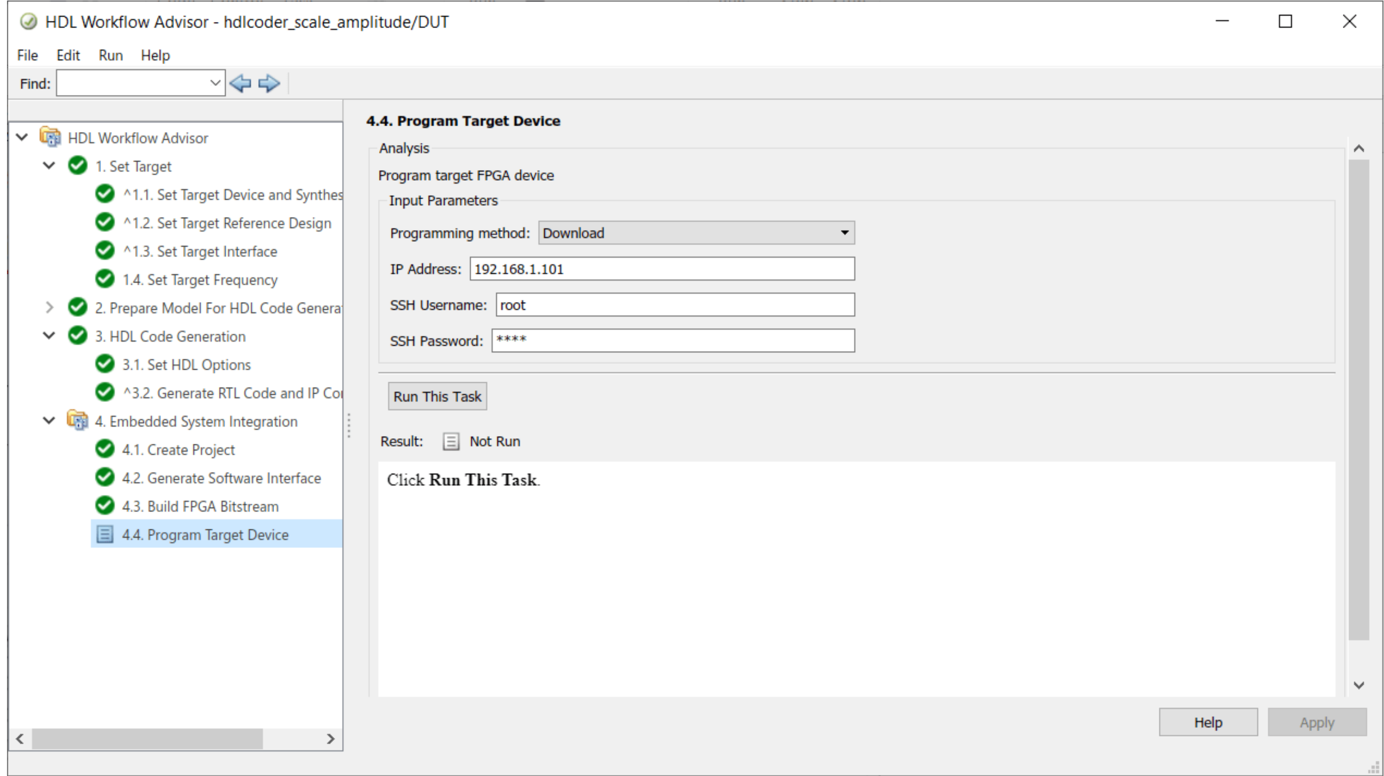


3. Two MATLAB files are generated in your current folder that enable you to prototype your generated IP core directly from MATLAB.



4. Inspect these generated files in Interact with FPGA Design from Host Computer. on page 39-249 First, complete the remaining Workflow Advisor tasks. Run the **Build FPGA Bitstream** task, which might take some time to finish.

5. Run the **Program Target Device** task to program the FPGA algorithm onto the board. Choose the **Download** programming method, which downloads the FPGA bitstream onto the SD card and configures the ARM processing system to start up properly.



Interact with FPGA Design from Host Computer

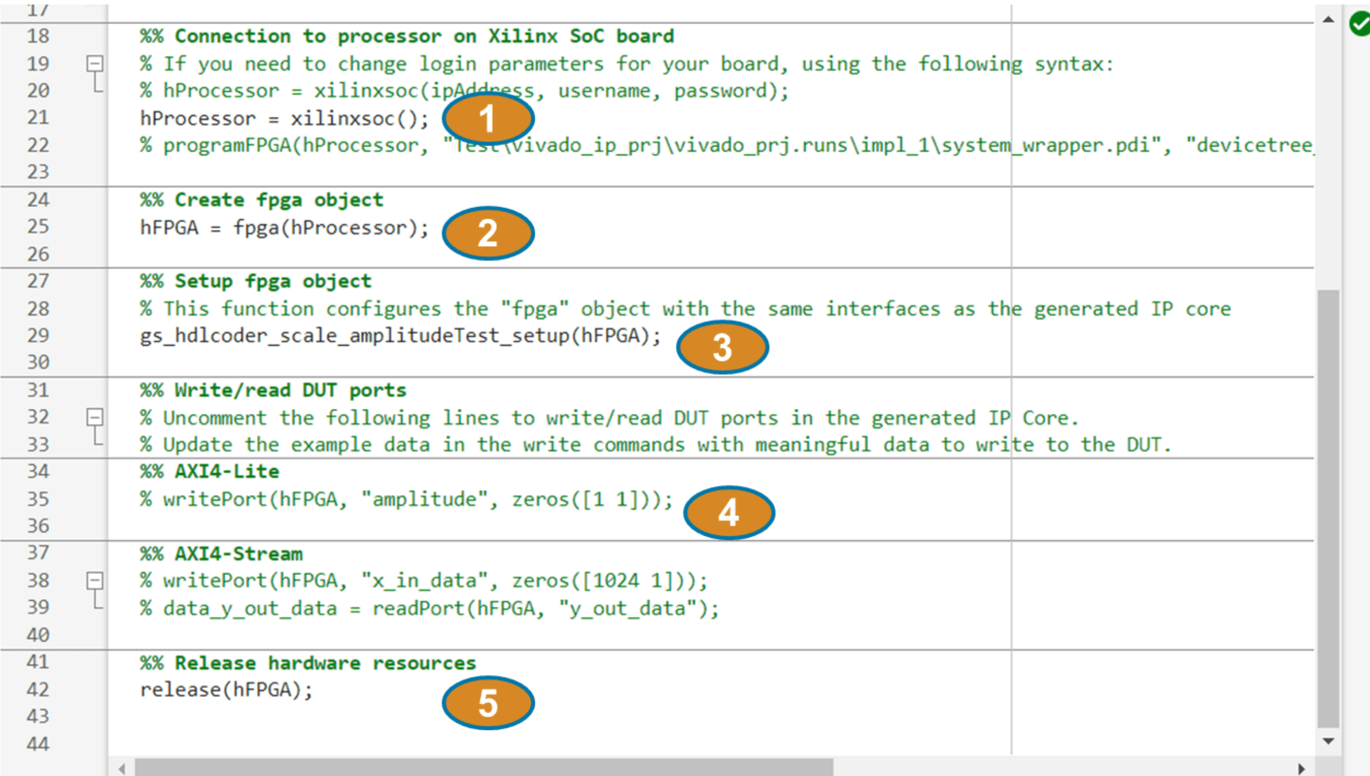
Interact with the FPGA design by reading and writing data from MATLAB on the host computer.

Open the generated script file:

```
open gs_hdlcoder_scale_amplitude_interface.m
```

This file creates a connection to your FPGA hardware for reading and writing data.

1. Creates a Processor hardware object, which represents a connection from MATLAB to the processor on your Xilinx SoC board.
2. Creates an “fpga” hardware object, which represents a connection to the FPGA through the processor on your hardware board.
3. Configures the “fpga” object with the desired hardware interfaces and ports from your DUT algorithm.
4. Reads and writes data to DUT ports to exercise your algorithm running on hardware.
5. Releases any hardware resources used by the `fpga` object to clean up the connection.



Open the generated setup function:

```
open gs_hdlcoder_scale_amplitude_setup.m
```

This function configures the `fpga` hardware object with the same ports and interfaces that were mapped in the **Set Target Interface** task. You can reuse this function can be reused in your own scripts to recreate this configuration.

```
function gs_hdlcoder_scale_amplitudeTest_setup(hFPGA)
%--------------------------------------------------------------------------
% Host Interface Script Setup
%
% Generated with MATLAB 24.1 (R2024a) at 19:42:47 on 05/07/2023.
% This function was created for the IP Core generated from design 'hdlcoder_scale_amplitudeTest'.
%
% Run this function on an "fpga" object to configure it with the same interfaces as the generated IP core.
%--------------------------------------------------------------------------

%% AXI4-Lite
addAXI4SlaveInterface(hFPGA, ...
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0xA4000000, ...
    "AddressRange", 0x10000);

DUTPort_amplitude = hdlcoder.DUTPort("amplitude", ...
    "Direction", "INOUT", ...
    "DataType", numerictype(1,32,16), ...
    "IsComplex", false, ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x100");

mapPort(hFPGA, [DUTPort_amplitude]);

%% AXI4-Stream
addAXI4StreamInterface(hFPGA, ...
    "InterfaceID", "AXI4-Stream", ...
    "WriteEnable", true, ...
    "WriteDataWidth", 32, ...
    "WriteFrameLength", 1024, ...
    "ReadEnable", true, ...
    "ReadDataWidth", 32, ...
    "ReadFrameLength", 1024);

DUTPort_x_in_data = hdlcoder.DUTPort("x_in_data", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,32,16), ...
    "IsComplex", false, ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");

DUTPort_y_out_data = hdlcoder.DUTPort("y_out_data", ...
    "Direction", "OUT", ...
    "DataType", numerictype(1,32,16), ...
    "IsComplex", false, ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");

mapPort(hFPGA, [DUTPort_x_in_data, DUTPort_y_out_data]);

end
```

DUT input port “amplitude” is Mapped to an AXI register

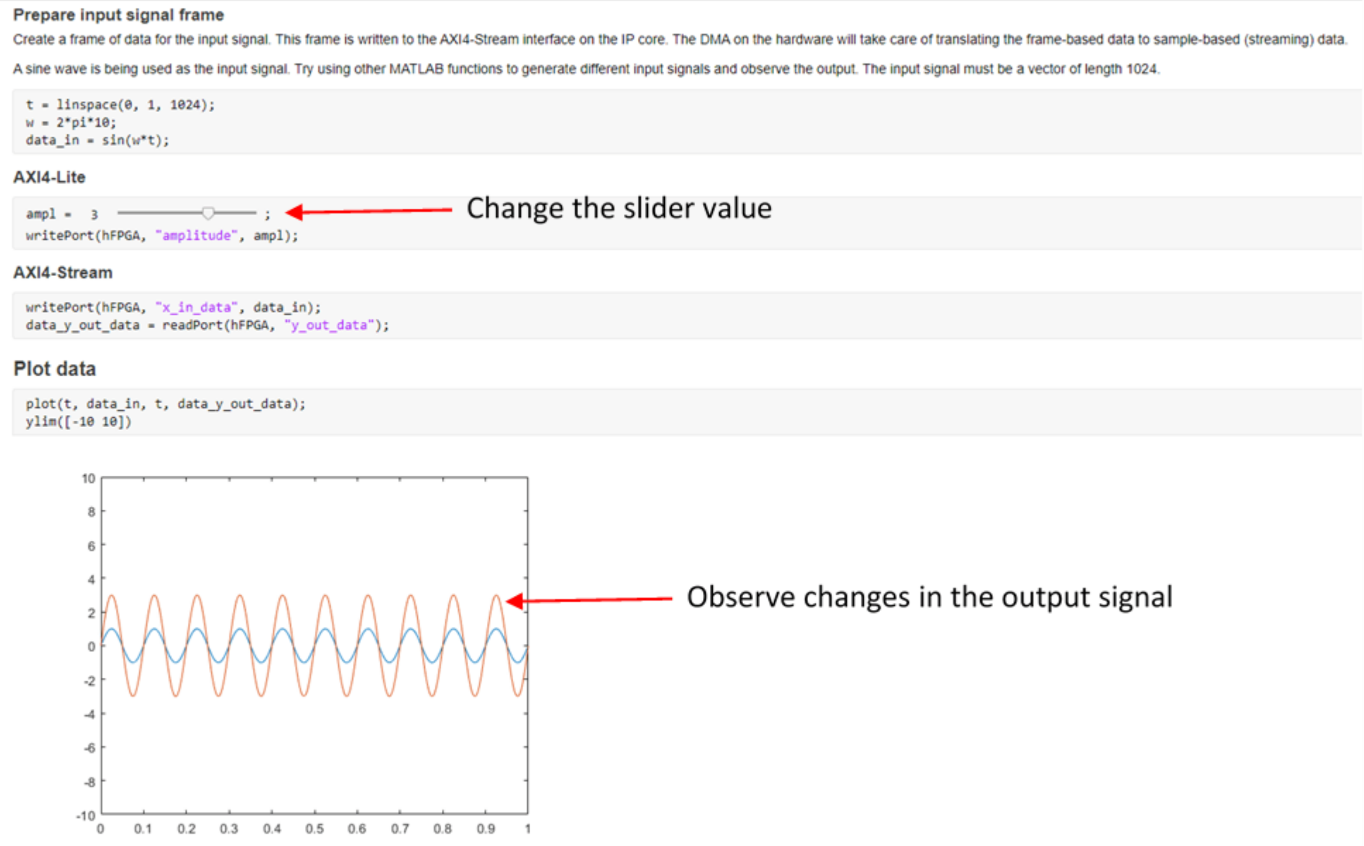
DUT input and output streaming ports

You can modify the generated script file to exercise the algorithm running on hardware. A live script has been prepared, which you can open by running this command:

```
open hdlcoder_scale_amplitude_script.mlx
```

Change the slider value and observe how the output data (orange) changes in the graph below it. As the slider moves, the code below it is executed. Each execution of the code:

- Writes the new amplitude value from the slider to an AXI register in the IP core.
- Writes one frame of the input signal.
- Reads one frame of output signal.
- Plots the input and output signals on the same graph to show the difference in amplitude.



When finished, run the last line of the script to release any hardware resources used by the `fpga` object for clean up:

```
release(hFPGA);
```

Next Steps

Experiment further with the generated script:

- Change the MATLAB function used to produce the input signal data on the line 7 of the live script. Some other functions to try are `cos`, `square` (Signal Processing Toolbox), and `sawtooth` (Signal Processing Toolbox).
- Use “Develop Apps Using App Designer” to create a custom app with the commands from this script. Add User Interface (UI) components for interacting with the algorithm as it runs on the hardware.
- Follow the steps from this example with your own model. Use the generated script to prototype your algorithm on hardware.

Author a Xilinx Zynq Linux Image for a Custom Zynq Board by Using MathWorks Buildroot

This example shows how to create a Zynq® Linux® image for starting a Zynq® custom board with the Linux® operating system in a Zynq® workflow by using MathWorks® buildroot system.

Introduction

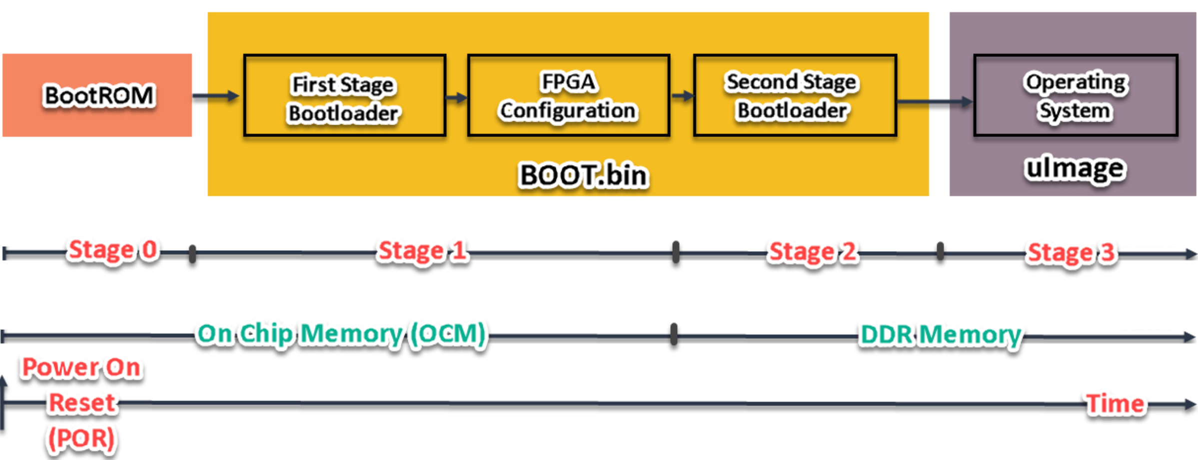
You create a Zynq® Linux® Image for a Digilent™ Zybo Z7-10 Zynq® development board for the custom reference design in the HDL workflow advisor. This example uses a Zybo Z7-10 board. You can build a Linux® image for other Zynq® platforms by using the MathWorks® build system.

Requirements

- Mathworks build system - Buildroot
- Linux® operating system
- Xilinx® Vivado® Design Suite, the supported version listed in the HDL Coder™ documentation
- Digilent® Zybo Z7-10 Zynq® development board with the accessory kit

Zynq Configuration

The Zynq® boot process involves the processor system (PS), loading and executing Zynq® boot image which consists of a First stage bootloader (FSBL), a bitstream to configure the programmable logic (PL), and user applications.



Boot Process

Stage 0

After the board is switched `ON` or the Zynq® processor is `reset`. BootROM, a read-only code executes from (CPU0) one of the processor from processing system (PS). Based on the boot mode pin

settings (JP5) in Zybo Z7-10 from available boot mode options, such as QSPI Flash, NAND Flash, SD Card, it copies the BootROM to on-chip memory (OCM). It executes the BootROM. Post-execution, the BootROM transfers execution to FSBL in the OCM.

Stage 1

The FSBL enables by configuring PS components, such as DDR memory controller and I/O components and looks for the bitstream file in the boot device. The FSBL loads application binaries and data files into memory until the complete image has been read from the boot device then FSBL initializes the external RAM and loads the second stage bootloader or stand alone application.

Stage 2

The user application loaded in the previous stage will be executed like U-Boot. It runs in CPU0 to initialize and setup the environment in which OS will boot. Initialization includes configuring the memory management unit.

Stage 3

The bootloader then fetches the kernal image and boots the embedded Linux operating system. In case of Linux, the OS detects and enables the second processor core CPU1, configures and activates the MMU and data caches and performs other actions to make a complete system available to applications.

Mathworks Build System

Mathworks Buildroot is a tool you use to generate embedded Linux® systems through cross-compilation. The MathWorks® build system wraps Buildroot in pre and post-processing scripts to automate the generation of system images for various platforms. The build system is based on scripts that take as input the target board, platform or image description file and output a complete system image, including bootloaders, kernel, and user space. This repository hosts the buildroot framework that creates the Altera® SoC and Xilinx® Zynq® platform SD Card images for use with MathWorks® tools.

Build a Zynq Linux image by Using the Mathworks Build System

1. Setup Cross compiler

Set up the cross compiler by Installing the Linaro ARM toolchain on your Linux machine using the following command. Below command works in `Debian® 10` Linux Machines.

```
>> sudo apt-get install gcc-arm-linux-gnueabihf
```

2. Sync MathWorks Buildroot from Git Hub

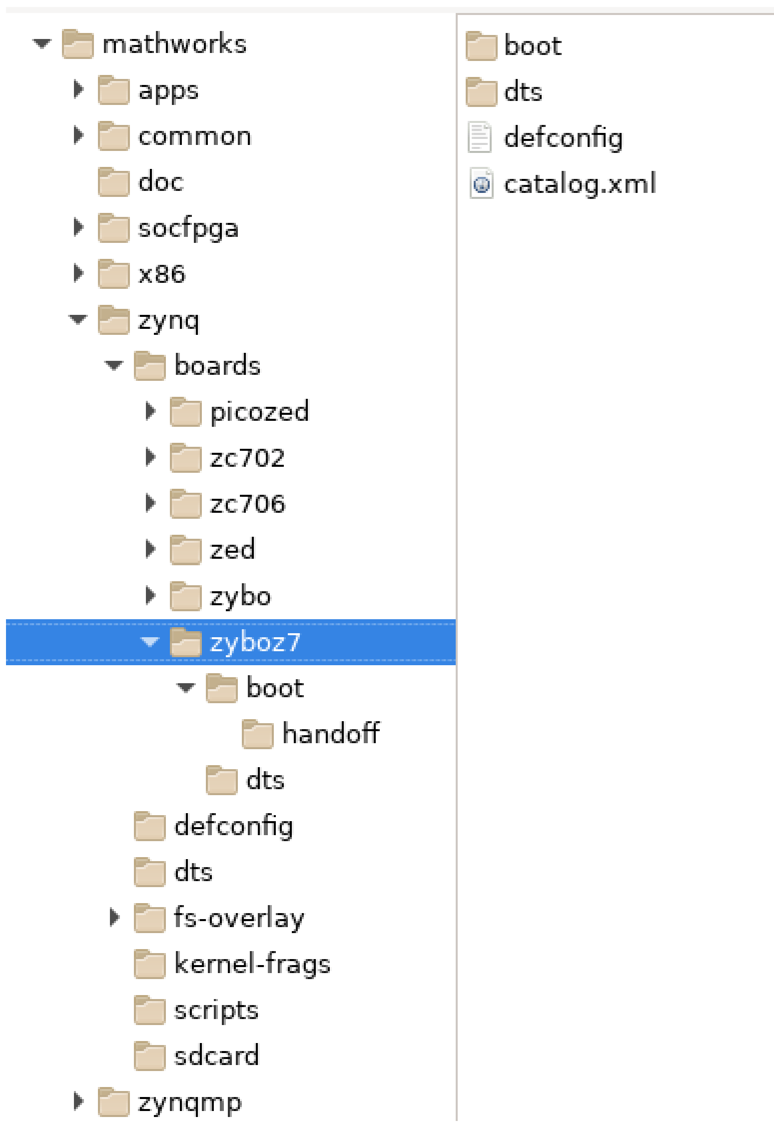
Get the buildroot from the git clone GitHub - MathWorks Buildroot by using the latest release tag.

```
>> git checkout mathworks_zynq_R22.1.0
```

The release tags are labeled as mathworks_zynq_R2x.y.z, where the "R2x" denotes the year of the MATLAB release and "y" denotes the 'a' or 'b' release. The "z" is used for updates to the image within a particular MATLAB release. For example, the tag "R22.1.0" is interpreted as MATLAB R2022a where "R22" represents R2022 and "1.0" is for release 'a'. If there will be any updates to the image within the R2022a release, the versions would be "R22.1.1", "R22.1.2", etc.

After syncing the Mathworks buildroot, build the image by using the following files. Navigate to the file folder and place the required files in it. This image shows the folder hierarchy and contents inside it that you use for building the Linux image for a specific board (ZyboZ7).

/buildroot/board/mathworks/zynq/boards/zyboz7/



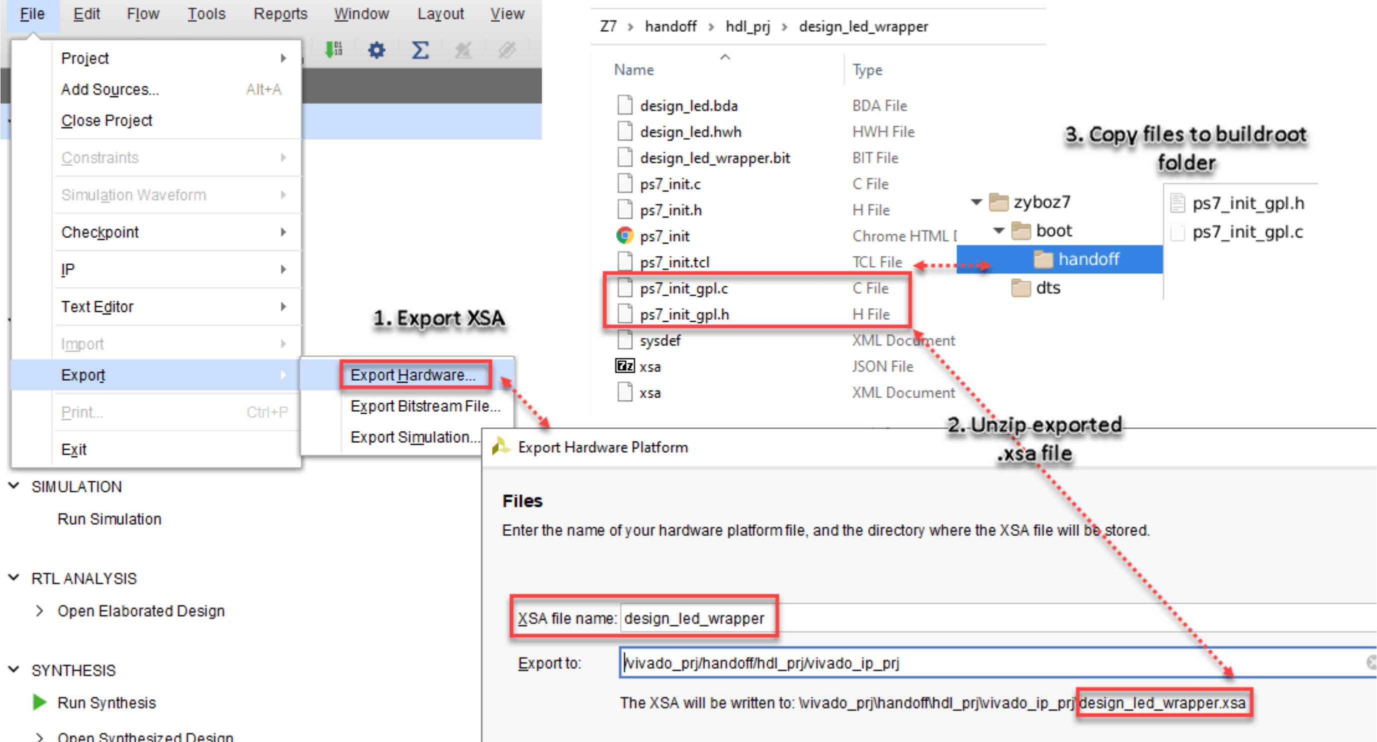
Required files for Building Zynq Linux Image

- Exported Handoff files
- Device Tree Source file
- Zynq Defconfig file

Handoff Files

The settings applied in Zynq processing system IP such as address mapping of memory and peripherals, are exported from the Xilinx Vivado project. Bitstream file, system hardware project file (.hdf) are exported after custom reference design creation. For more information, see the Create and

Export a Custom Reference Design by Using Xilinx Vivado section in “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252. Export the hardware platform from the vivado block design project and unzip the .xsa file to get ps7_init_gpl.c, ps7_init_gpl.h and place these files into handoff folder in the zyboz7 buildroot directory as shown in the below image.



Device Tree Source

Device tree is a data structure that describes the hardware which is readable by an operating system like Linux so that it doesn't need to hard code any details of the machine. A device tree source (DTS) file contains plain text data that describes a list of nodes,properties and child nodes for platform identification, run-time configuration like bootargs. DTS files are compiled using device tree compiler into device tree blob (DTB) which is a binary file that the hardware uses during the boot process.

This code is the sample code of the Zybo Z7-10 Device tree source file.

```
#include "zynq-7000.dtsi"
#include <dt-bindings/gpio/gpio.h>

/ {
    model = "Digilent Zybo Z7 board";
    compatible = "digilent,zynq-zybo-z7", "xlnx,zynq-7000";


    aliases {
        ethernet0 = &gem0;
        serial0 = &uart1;
    };


    memory@0 {
```

```
        device_type = "memory";
        reg = <0x0 0x40000000>;
    };


    chosen {
        bootargs = "";
        stdout-path = "serial0:115200n8";
    };


    gpio-leds {
        compatible = "gpio-leds";


        ld4 {
            label = "zynq-zybo-z7:green:ld4";
            gpios = <&gpio0 7 GPIO_ACTIVE_HIGH>;
        };
    };


    usb_phy0: phy0 {
        #phy-cells = <0>;
        compatible = "usb-nop-xceiv";
        reset-gpios = <&gpio0 46 GPIO_ACTIVE_LOW>;
    };
};


&clkc {
    ps-clk-frequency = <33333333>;
};


&gem0 {
    status = "okay";
    phy-mode = "rgmii-id";
    phy-handle = <&ethernet_phy>;


    ethernet_phy: ethernet-phy@0 {
        reg = <0>;
        device_type = "ethernet-phy";
    };
};


&sdhci0 {
    status = "okay";
};


&uart1 {
    status = "okay";
};
```

```
&usb0 {
    status = "okay";
    dr_mode = "host";
    usb-phy = <&usb_phy0>;
};
```

3. Zynq Defconfig

The platform defconfig contains the Linux kconfig settings required to properly configure the kernel build. This file is the zynq_zybo_z7_defconfig file used for building the Z7-10 Image.

```
###########################################

# Zybo Options

###########################################

BR2_TARGET_UBOOT_BOARD_DEFCONFIG="zynq_zyboz7"

BR2_PACKAGE_XILINX_BOOTLOADER_UBOOT_TARGET_DIR="board/xilinx/zynq/zynq-zyboz7"
```

Build Script

Using the following build script we can initiate the linux SD image build for the specific board.

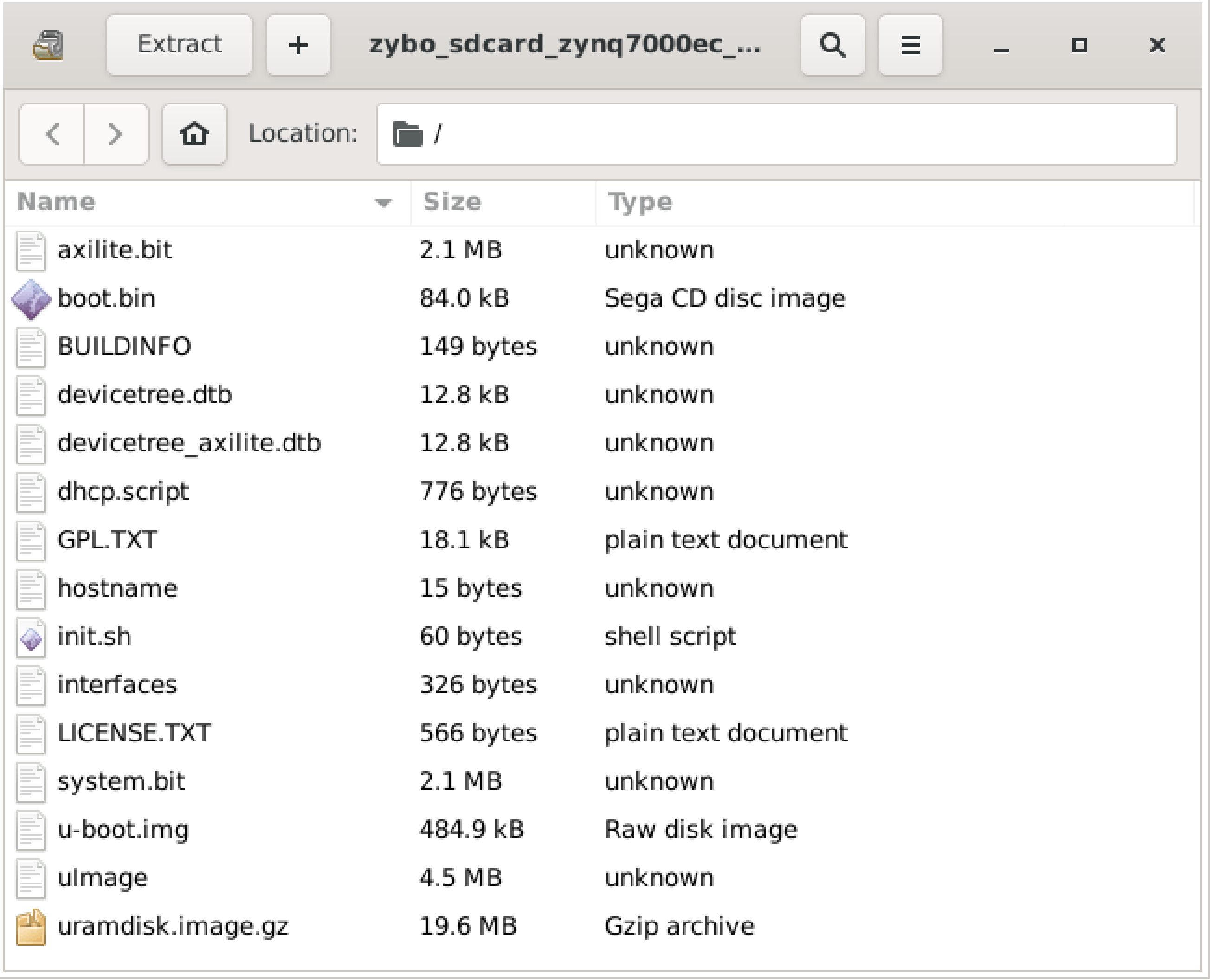
```
>> build.py -c board/mathworks/zynq/boards/zyboz7/catalog.xml
```

Otherwise use this command to provide the different toolchain path manually to build.

```
>> ./build.py -p zynq -b zyboz7 --brconfig BR2_TOOLCHAIN_EXTERNAL_PATH='/opt/ARM_CortexA/LinaroToolchain/6.3.1-2017.02/Linux/aarch32'
```

List of files after build

After a build, these files are in the output folder **/buildroot/output/zybo_linux_linaro/images/** as a zip file. Unzip the files into the SD card and insert the SD card on Zybo Z7-10 board. To learn more about board connections, see “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252.



Summary

This example shows step by step process of building a Linux image by using the Mathworks Buildroot system. You can build a Linux image for other Zynq platforms by using the MathWorks build system. Using this example you can create Xilinx Zynq Linux image for the custom Zynq boards by including the required files in the Mathworks buildroot folder and setting up the linaro cross-compiler.

Generate Board-Independent HDL IP Core for Microchip Platforms

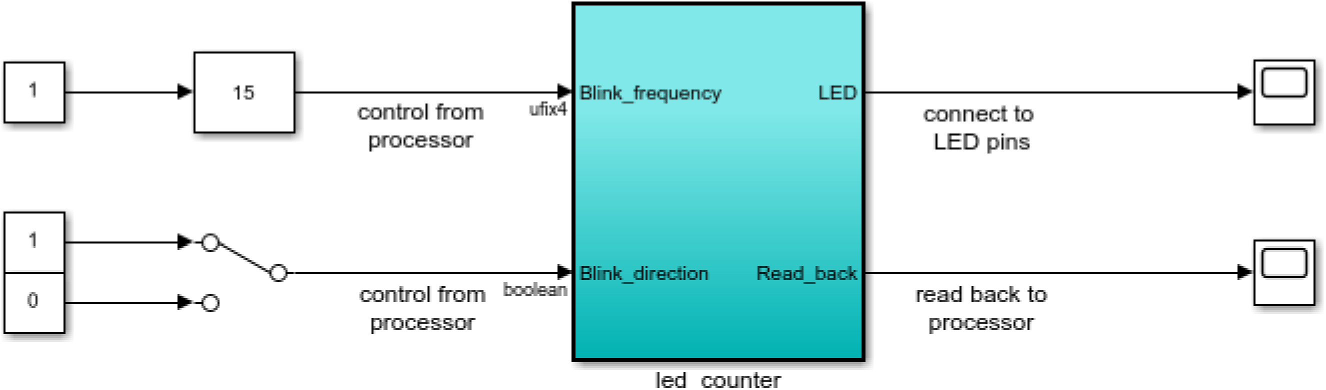
This example shows how to generate an HDL IP core for generic Microchip platform. You generate an HDL IP core for an LED blinking model, then integrate this HDL IP core into a target Microchip platform of your choice. You can generate a board-independent custom IP core to use in an embedded system integration environment, such as Microchip Libero SoC tool.

Generate Board-Independent IP Core

To generate a board-independent custom IP core for an LED blinking model, follow these steps:

1. Open the `hdlcoder_led_blinking_4bit` model and click the `led_counter` subsystem.

```
open_system('hdlcoder_led_blinking_4bit');
```



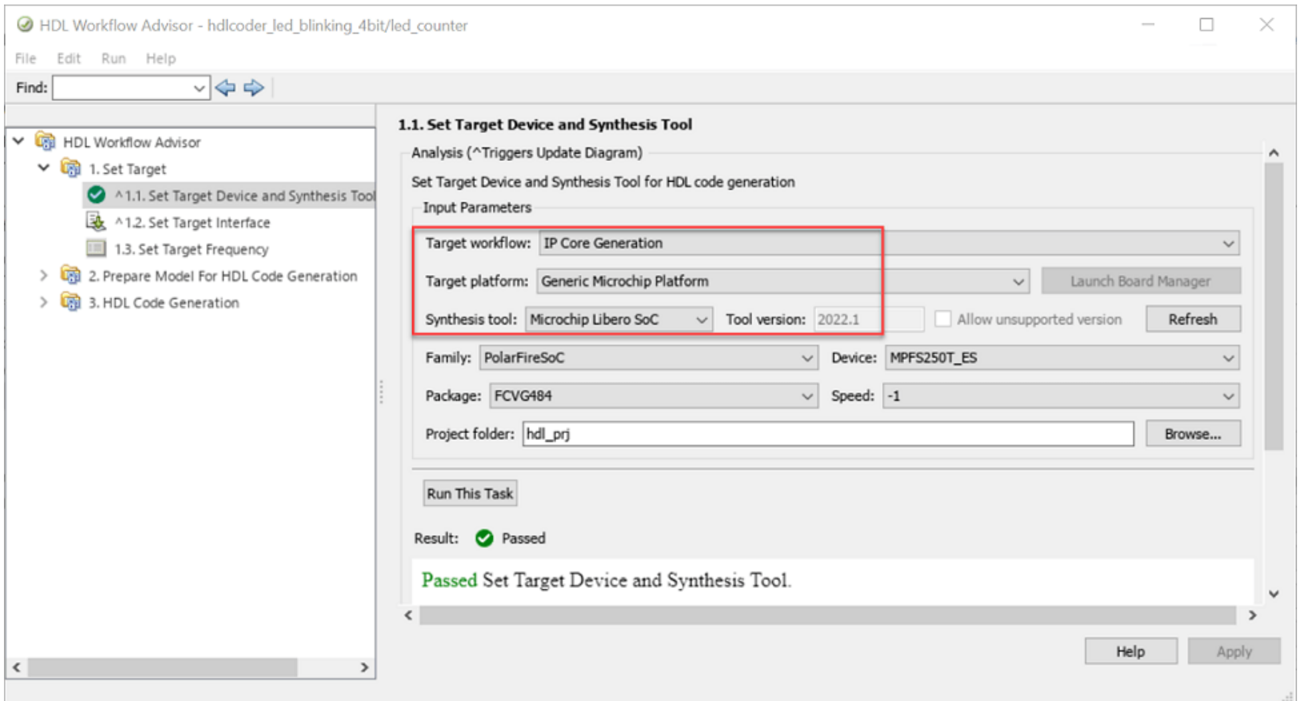
2. Set the path to the Microchip Libero SoC synthesis tool by using the `hdlsetuptoolpath` function:

```
hdlsetuptoolpath('ToolName','Microchip Libero SoC', ...
    'ToolPath','C:\Microsemi\Libero_SoC_v2022.1\Designer\bin\libero.exe')
```

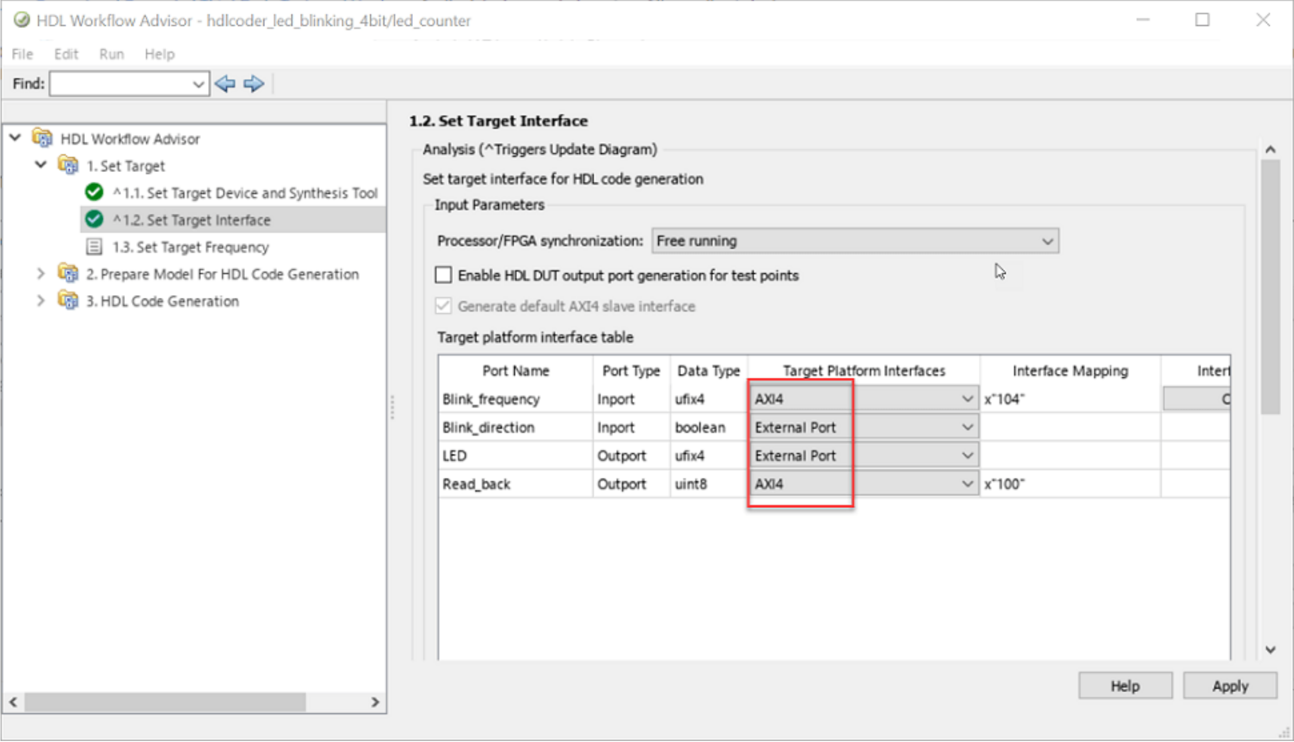
See “HDL Language Support and Supported Third-Party Tools and Hardware” for the latest supported version of the synthesis tool.

3. Open the HDL Workflow Advisor for the subsystem representing the device under test (DUT). For the LED blinking model, the `led_counter` subsystem represents the DUT. In the **Set Target > Set Target Device and Synthesis Tool** task:

- Set **Target workflow** to `IP Core Generation`.
- Set **Target platform** to `Generic Microchip Platform`.
- Click **Run This Task**.



4. In the **Set Target > Set Target Interface** task, select a **Target Platform Interface** value for each port, and then click **Apply**. You can map each DUT port to the AXI4-Lite or AXI4 interface. For more information about these interfaces, see "Target Platform Interfaces" on page 39-17.



If you do not want to map the DUT ports to AXI4 slave interfaces, map them to external Port interfaces.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
Blink_frequency	Inport	ufix4	External Port		
Blink_direction	Inport	boolean	External Port		
LED	Outport	ufix4	External Port		
Read_back	Outport	uint8	External Port		

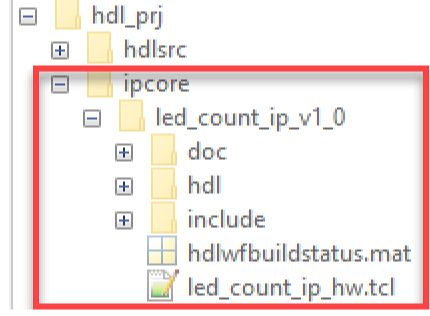
5. In the **HDL Code Generation > Set HDL Options**, click **HDL Code Generation Settings** to set HDL configuration parameters.

6. In the **HDL Code Generation > Generate RTL Code and IP Core** task, you can specify these options:

- Option to connect the DUT IP core to multiple AXI Master interfaces. By default, the **AXI4 Slave ID** Width value is 12, which indicates that you can connect the HDL IP core to one AXI Master interface. To connect the DUT IP core to multiple AXI Master interfaces, to increase the **AXI4 Slave ID Width** parameter value. When you run this task, this setting is saved on the DUT as the HDL block property **AXI4SlaveIDWidth**.

- Option to generate the default AXI4 slave interface. By default, HDL Coder™ generates AXI4 slave interfaces for signals such as the clock, reset, ready, and timestamp. If you do not want to generate AXI4 slave interfaces, clear the **Generate default AXI4 slave interface** check box. Click **Run This Task**. When you clear the check box and run the task, the code generator saves this setting in the DUT subsystem as the HDL block property **GenerateDefaultAXI4Slave**.

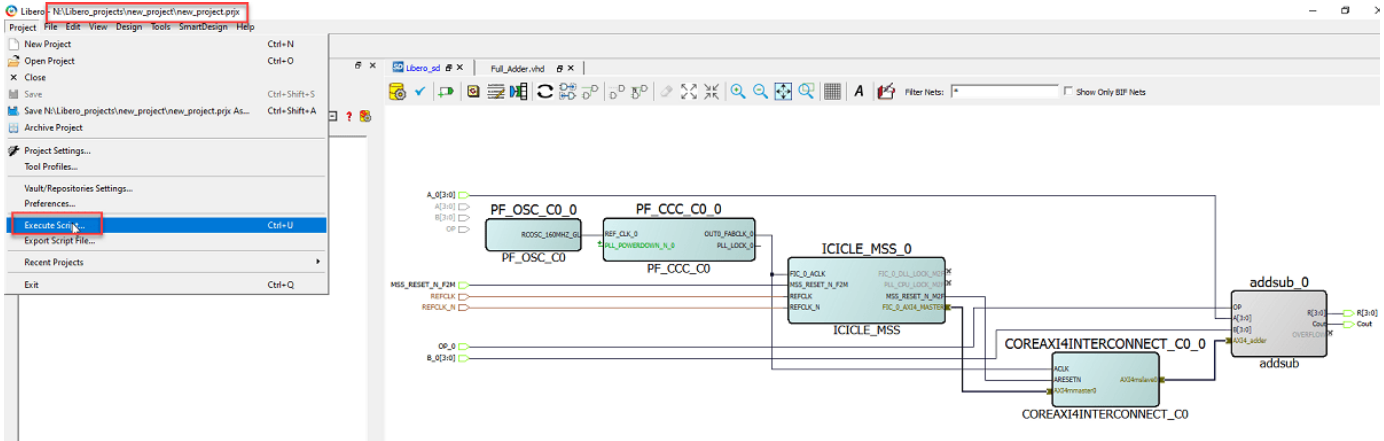
After running the task, HDL Coder generates the IP core files in the output folder in the **IP core folder** field, including the HTML documentation. To view the IP core report, click the link in the message window.



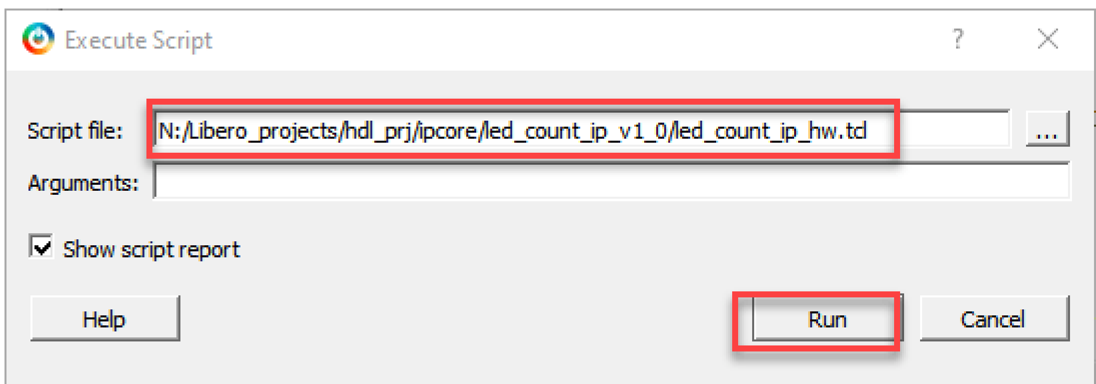
Insert HDL IP Core in Microchip Libero SoC SmartDesign Project

To import generated HDL IP Core into Libero SmartDesign Project, follow the below process.

1. Open Microchip Libero Project in which HDL Coder IP needs to be imported. In the **Project** pane, click **Execute Script**.

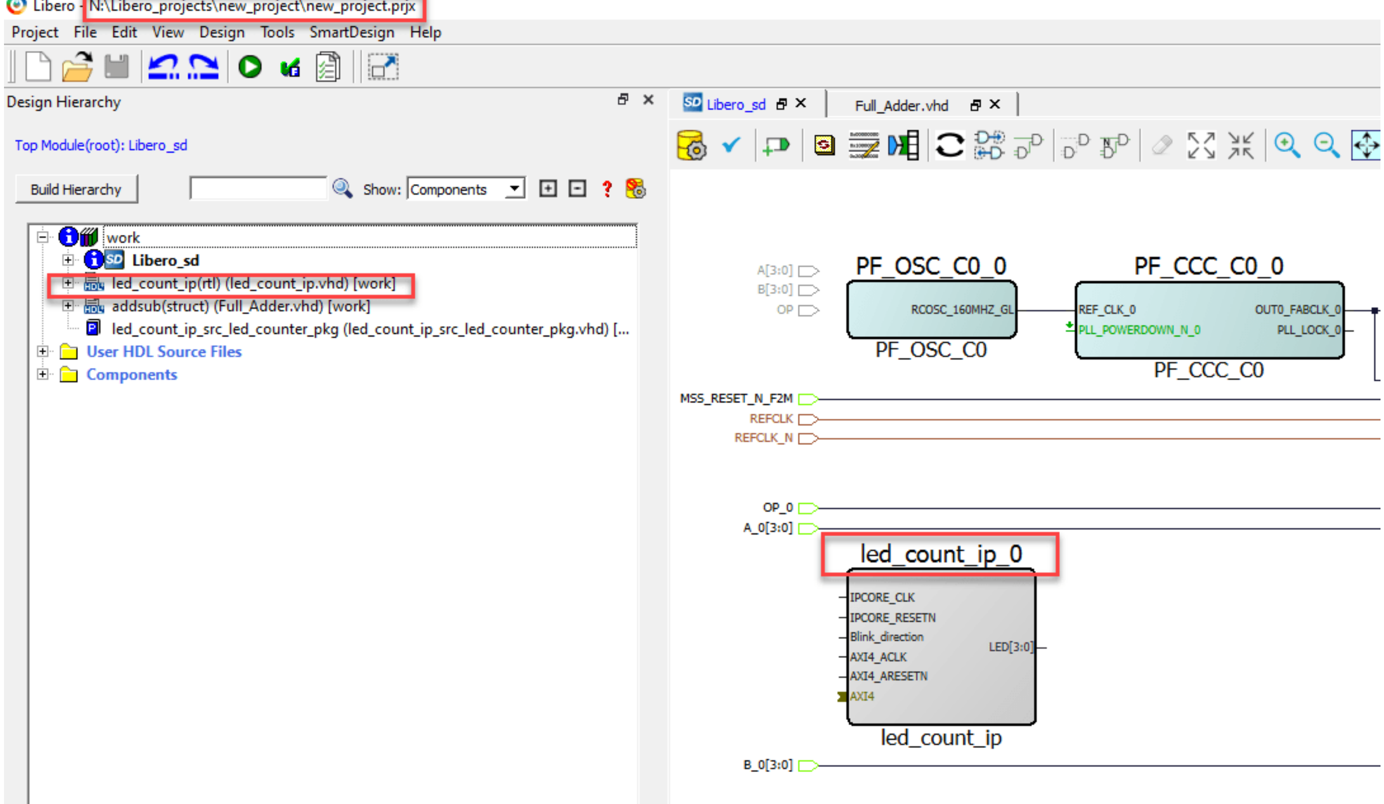


2. In Script file, choose the `led_count_hw_ip.tcl` file from `N:/Libero_Projects/hdl_prj/ipcore/led_count_ip_v1_0` folder. Click **Run**.



3. After running `led_count_hw_ip.tcl` script, HDL Coder IP gets imported into Microchip Libero Project.

4. Instantiate HDL Coder IP by drag and drop of `led_count_ip` module into SmartDesign.



5. Connect `led_count_ip` Clock, Reset ports and Interfaces to other IP's Ports and interface as per requirement. This will form a complete Reference Design.

6. Now, You can follow next steps of implementation in Microchip Libero Tool.

Conclusion and Further Exploration

This example shows how to generate an HDL IP core using a generic microchip platform and integrate the generated IP core into Libero SmartDesign. You can integrate other IP cores into Libero SmartDesign and use these cores to create a complete reference design.

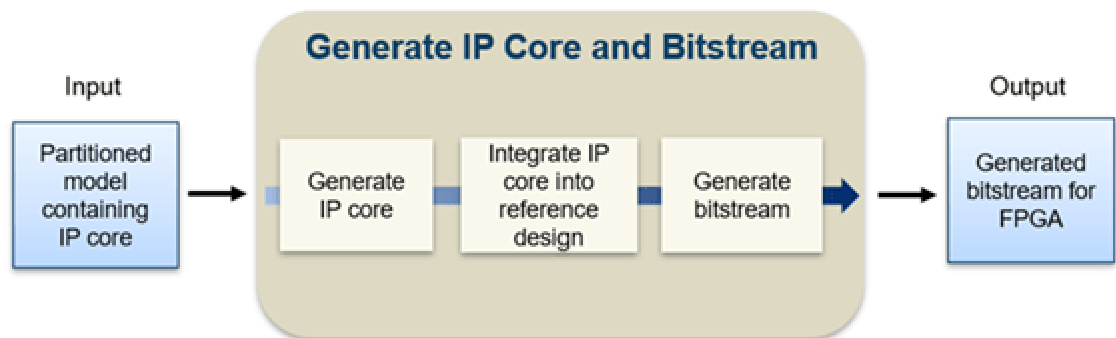
See Also

- "Board and Reference Design Registration System" on page 40-89
- "Custom IP Core Generation" on page 39-17

Getting Started with Targeting Xilinx Versal Adaptive SoC Platform

This example shows how to use HDL Coder™ and the hardware-software co-design workflow to blink LEDs at various frequencies on the Xilinx® Versal® Adaptive SoC platform.

You can use the hardware-software co-design workflow to automate the deployment of your MATLAB® and Simulink® design to a Xilinx Versal Adaptive SoC platform device. You can use this workflow to experiment with ways to partition and deploy your design. This diagram shows the high-level process of generating an IP core and bitstream. For more information, see “Targeting FPGA & SoC Hardware Overview” on page 39-3.



In this example, you:

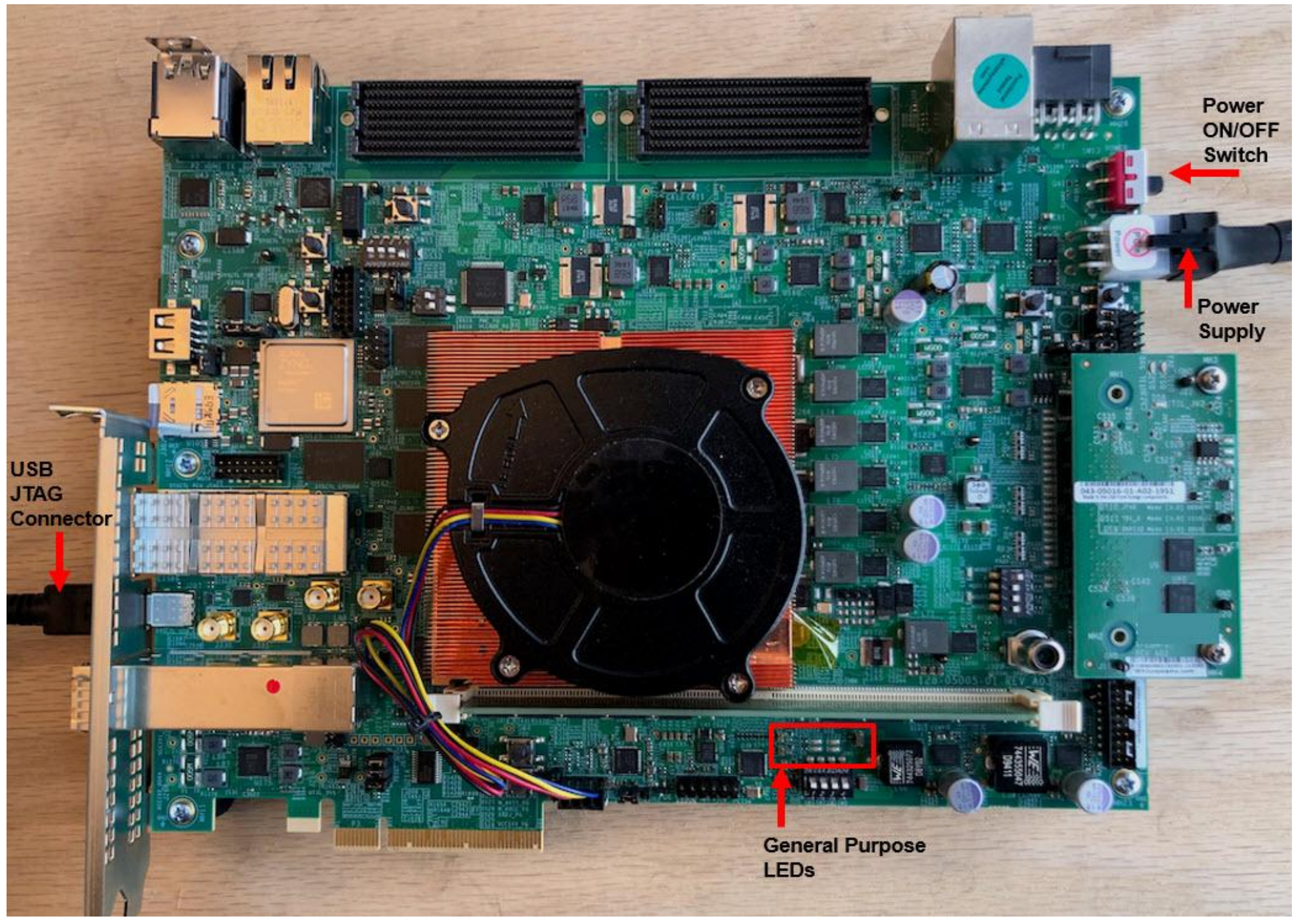
- Set up your Xilinx Versal VCK190 hardware and tools.
- Generate an HDL IP Core using HDL Workflow Advisor.
- Integrate the IP core into a Xilinx Vivado project and program the Xilinx Versal hardware.
- Run FPGA I/O commands to access AXI registers on the FPGA from the MATLAB command prompt.
- Run a software algorithm in External Mode on the ARM processor from Simulink.

Requirements

- Xilinx Vivado Design Suite. To see the supported versions, see “HDL Language Support and Supported Third-Party Tools and Hardware”
- Xilinx Versal AI Core Series VCK190 Evaluation Kit.
- HDL Coder Support Package for Xilinx FPGA and SoC Devices.

Set Up Xilinx Versal Adaptive SoC Hardware and Tools

1. Set up the Xilinx Versal AI Core Series VCK190 evaluation kit as shown in this image. To learn more about the VCK190 hardware setup, refer to the Xilinx VCK190 board user guide.



2. Connect your computer to the USB JTAG/UART connector of the VCK190 using a USB-C cable.

3. Connect your computer to the top Ethernet connector of the VCK190 using an Ethernet cable.

4. Install the HDL Coder Support Packages for Xilinx FPGA and SoC Devices.

5. Set up the Xilinx Vivado synthesis tool path by entering this command in the MATLAB Command Window. Use your own Vivado installation path.

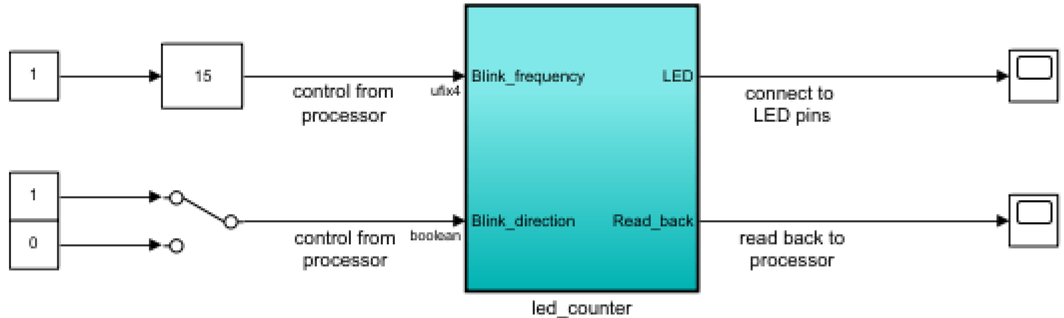
```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2022.1\bin\vivado.ba
```

Generate HDL IP Core Using HDL Workflow Advisor

You can use the HDL Workflow Advisor to automatically generate a shareable and reusable IP core module from a Simulink model. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic that connects the IP core to the embedded processor. HDL Coder packages the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Xilinx Vivado environment.

In this example, the subsystem `led_counter` is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. The two input ports, `Blink_frequency` and `Blink_direction`, are control ports that determine the LED blink frequency and direction.

```
open_system('hdlcoder_led_blinking_4bit');
```



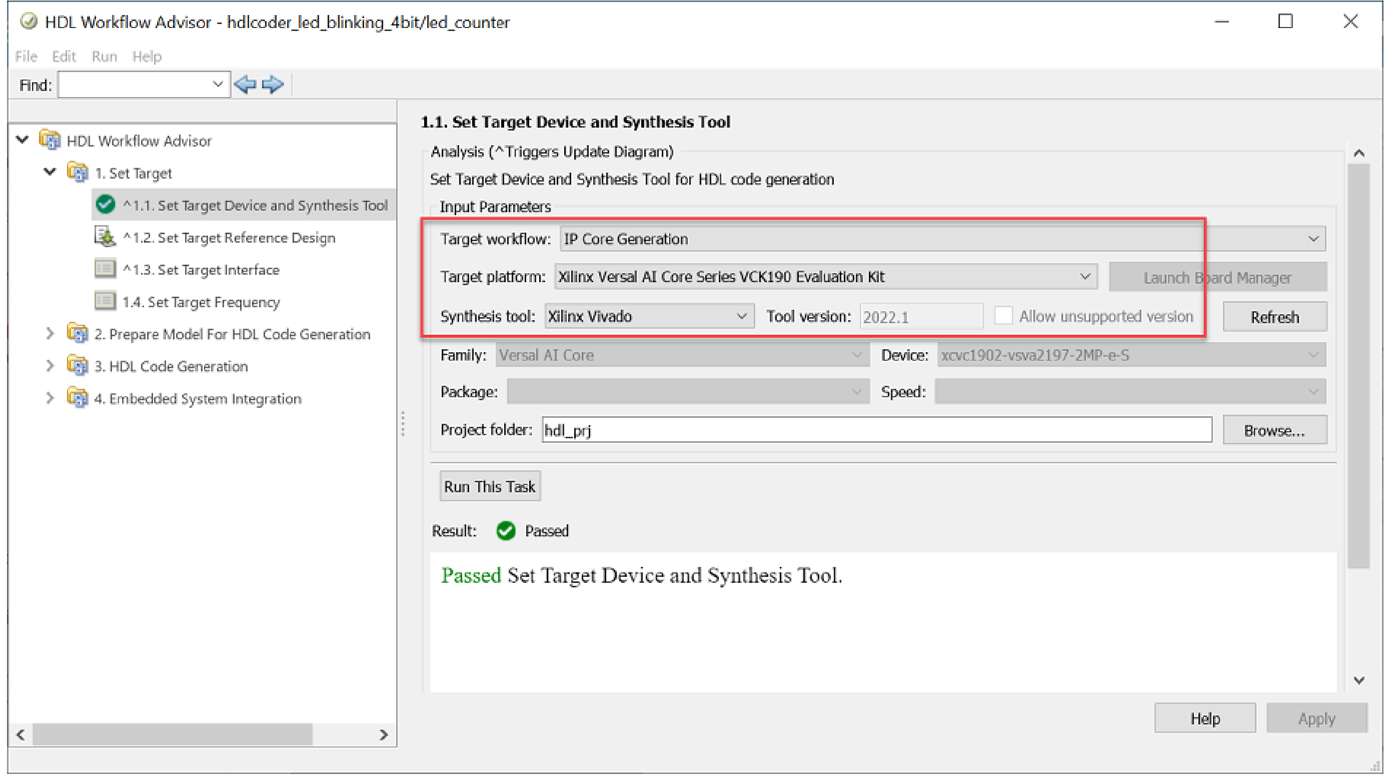
Start the IP core generation workflow

1. Open the HDL Workflow Advisor from the `led_counter` subsystem by right-clicking the `led_counter` subsystem and choosing **HDL Code** > **HDL Workflow Advisor**.

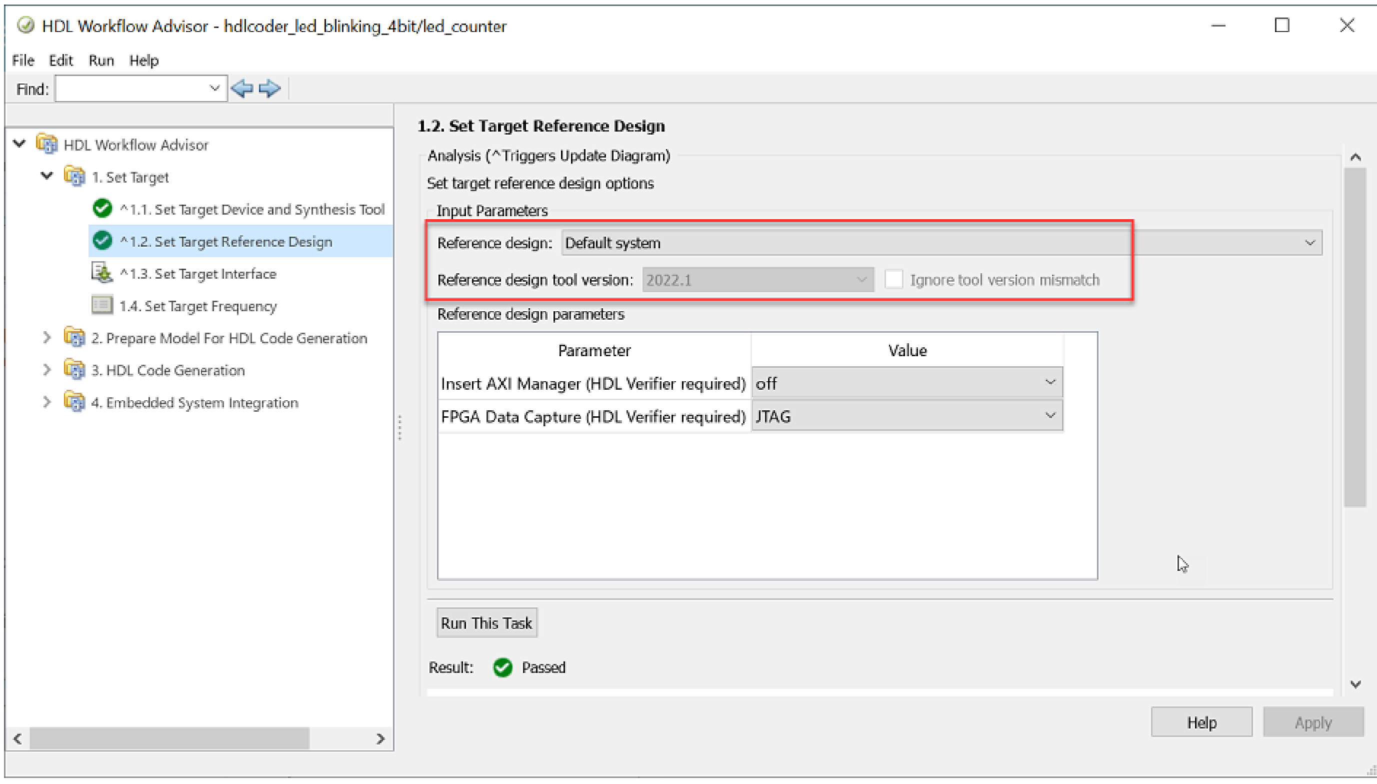
2. In the **Set Target** > **Set Target Device and Synthesis Tool** task, set **Target workflow** to `IP Core Generation`.

3. Set **Target platform** to `Xilinx Versal AI Core Series VCK190 Evaluation Kit`. If you do not have this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Xilinx Versal Platform and follow the instructions to complete the installation.

4. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.



5 In the **Set Target > 1.2. Set Target Reference Design** task, set **Reference Design** to `Default system`.

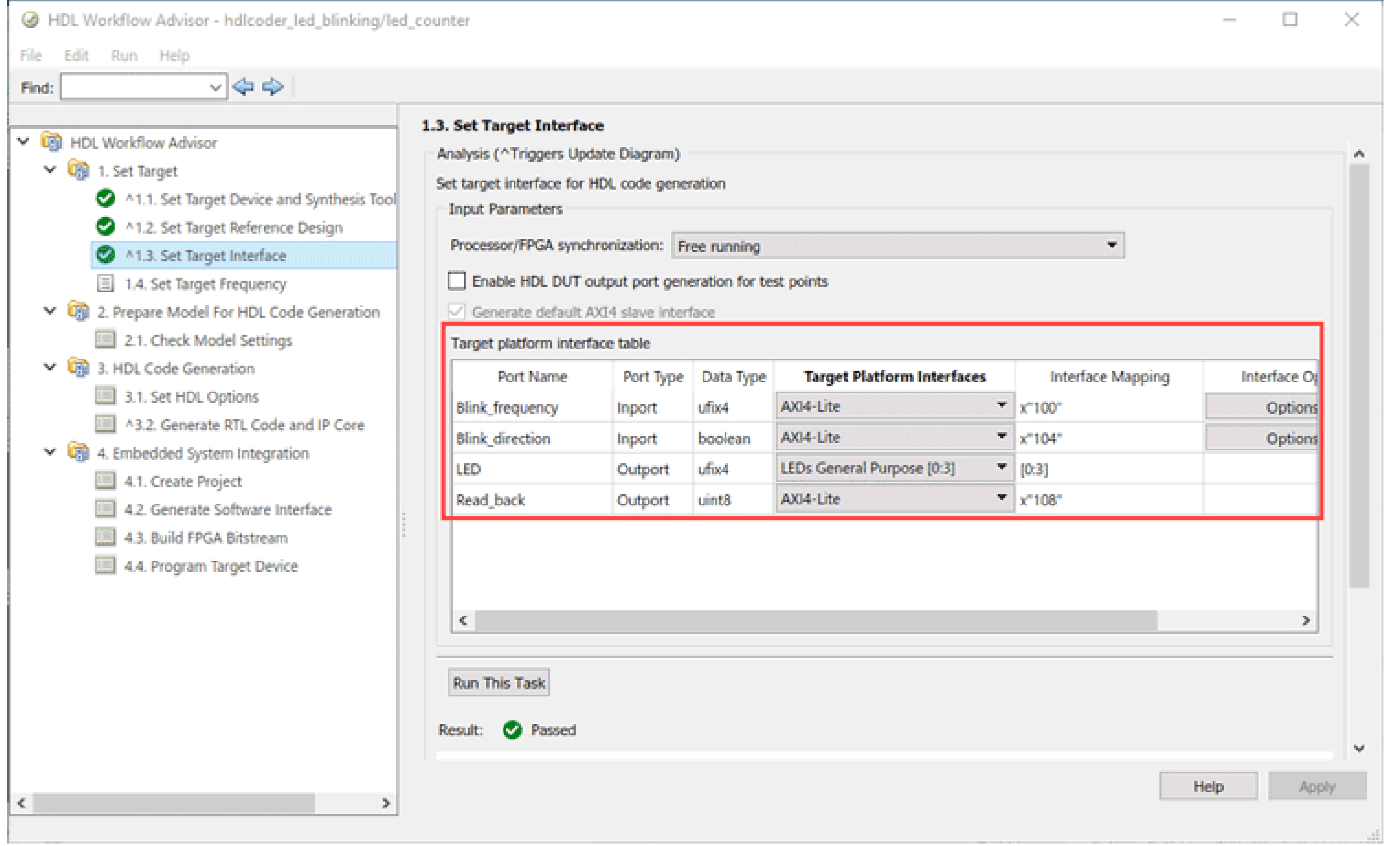


6. Click **Run This Task** to run the **Set Target Reference Design** task.

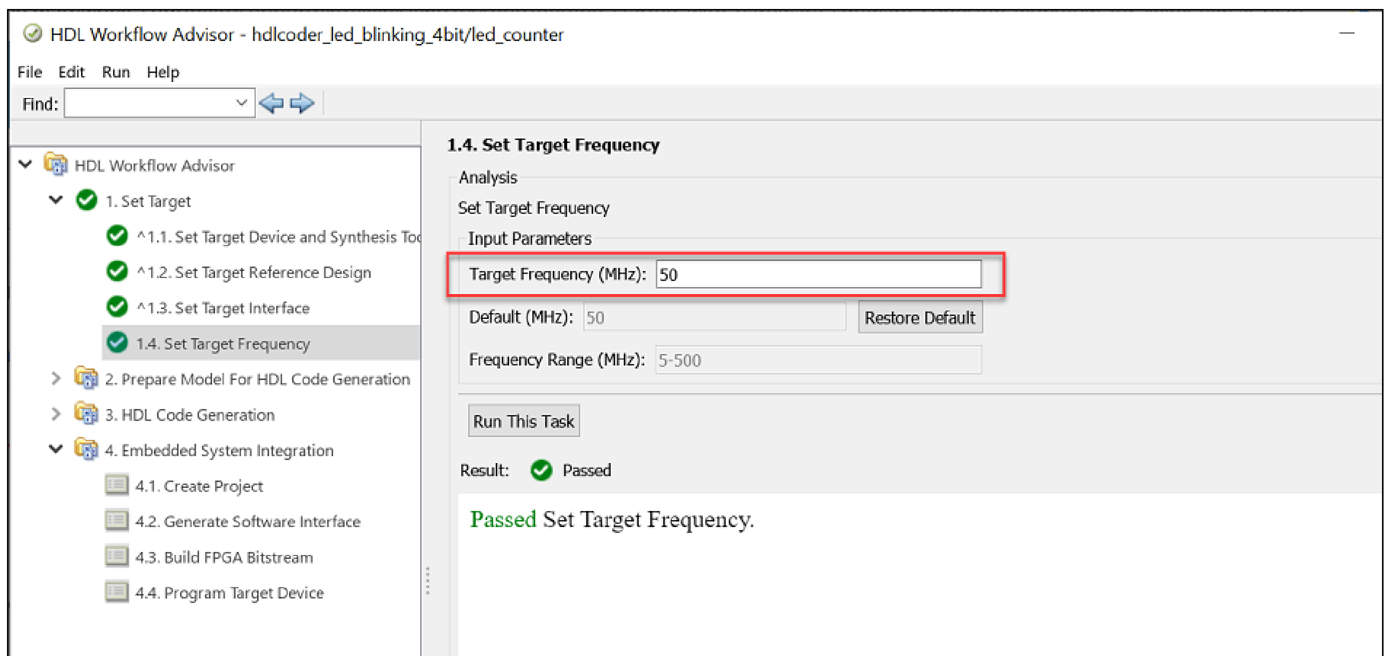
Next, configure your design to map to the target hardware by mapping the DUT ports to IP core target hardware and setting DUT-level IP core options. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4-Lite interface, so HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:3]**, which connects to the LED hardware on the Versal board.

1. In the **Set Target** > **1.3. Set Target Interface** task, set the **Target Platform Interfaces** cell to `AXI4-Lite` for `Blink_frequency`, `Blink_direction`, and `Read_back`.

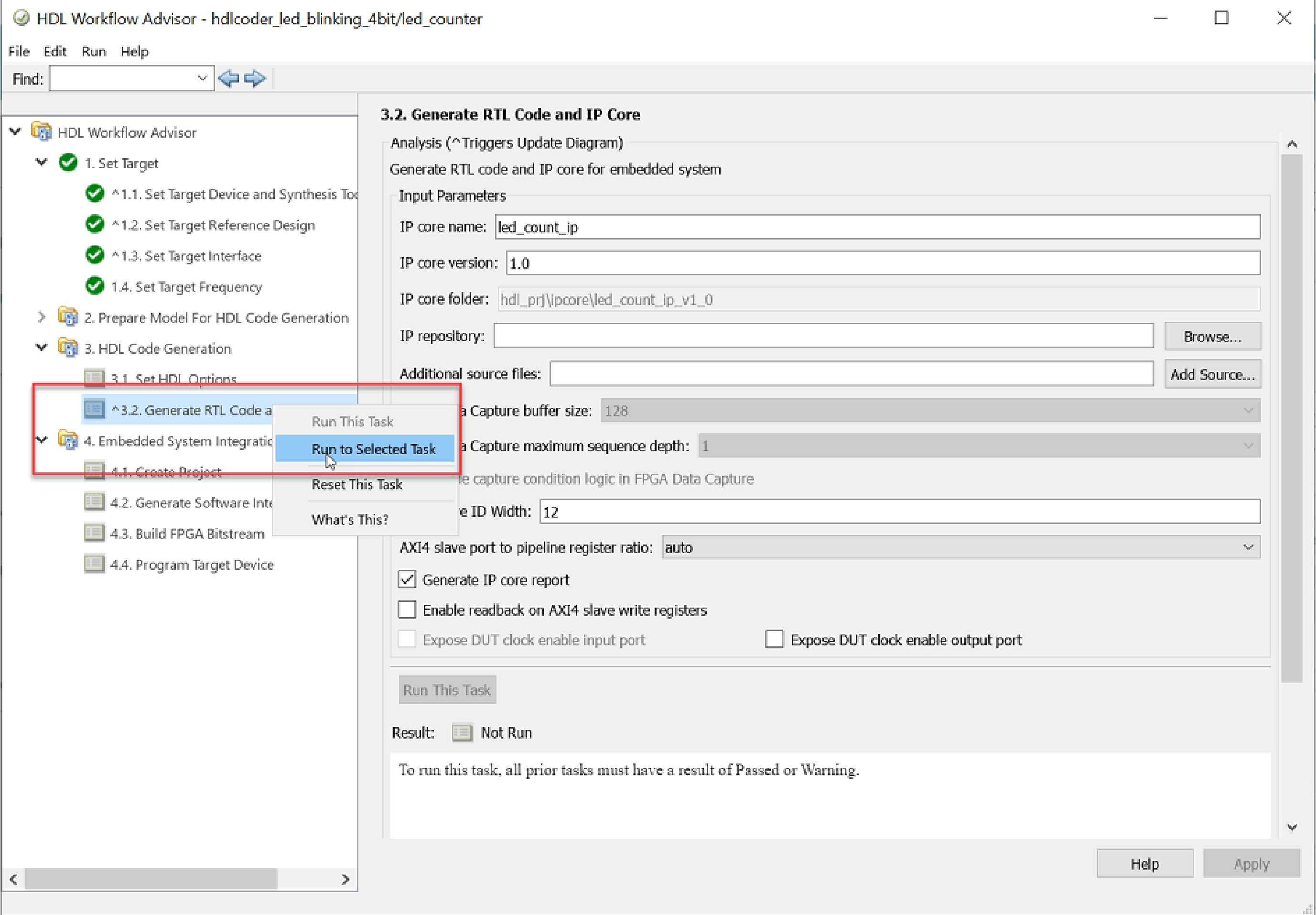
2 . Set the **Target Platform Interfaces** cell to `LEDs General Purpose [0:3]` for `LED`.



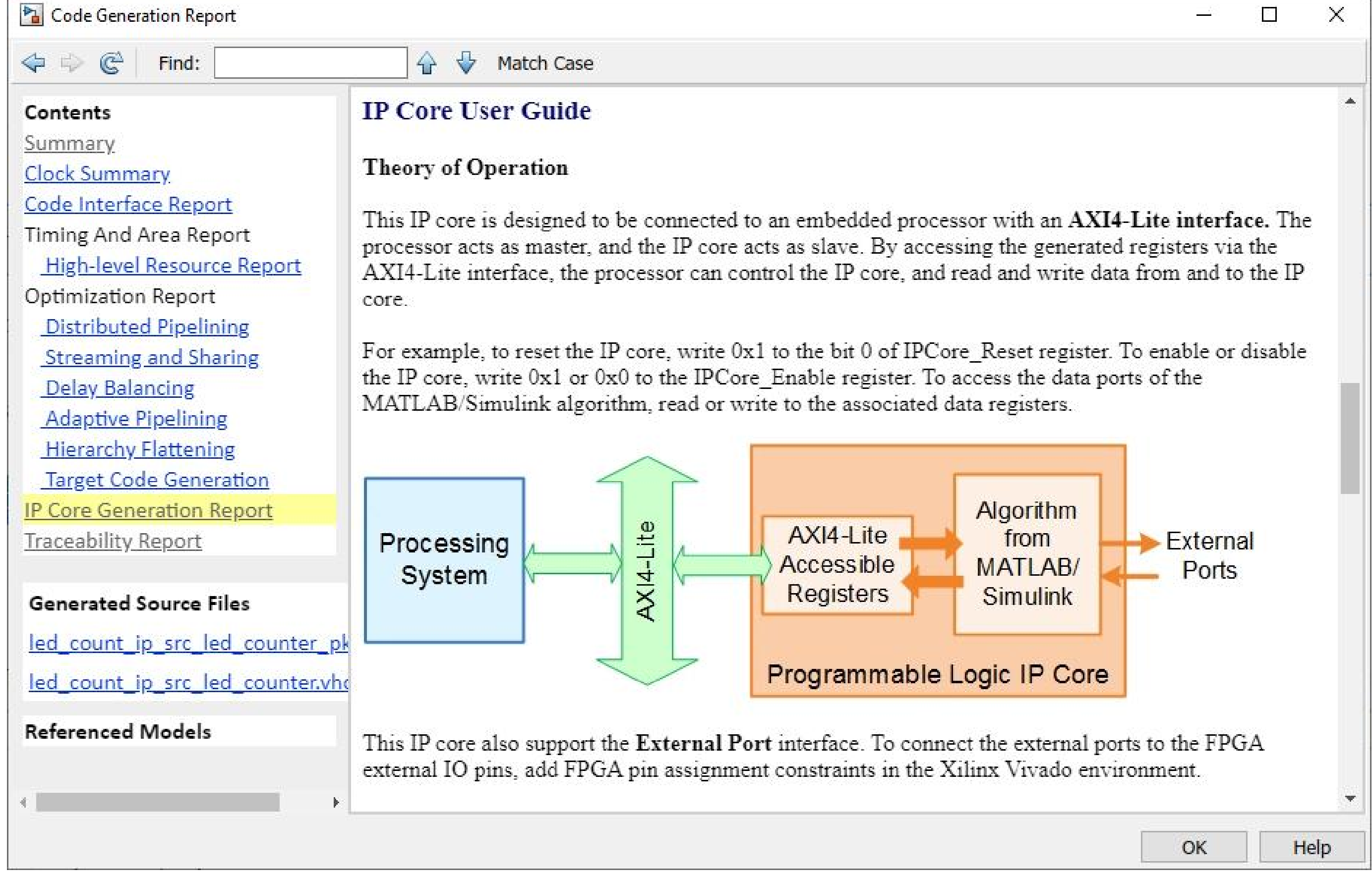
3. In the **Set Target > 1.4. Set Target Frequency** task, set **Target Frequency** to 50.



Next, generate the IP core for your DUT algorithm by right-clicking the **Generate RTL Code and IP Core** task and selecting **Run to Selected Task**.



Finally, generate and view the IP core report. After you generate the custom IP core, the IP core files and an HTML custom IP core report are in the `ipcore` folder in your project folder. The report describes the behavior and contents of the generated custom IP core.

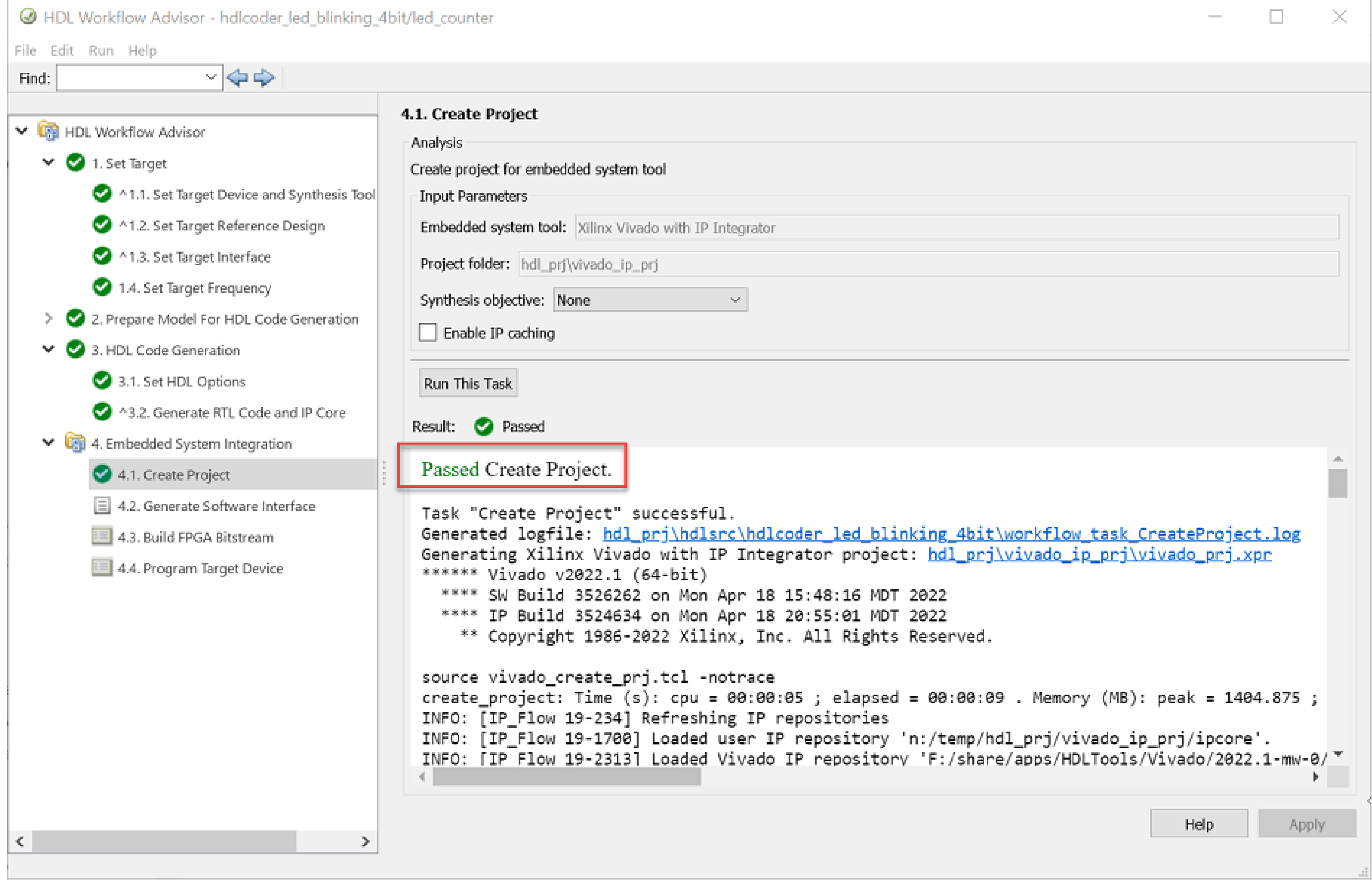


Integrate IP Core with Xilinx Vivado Environment

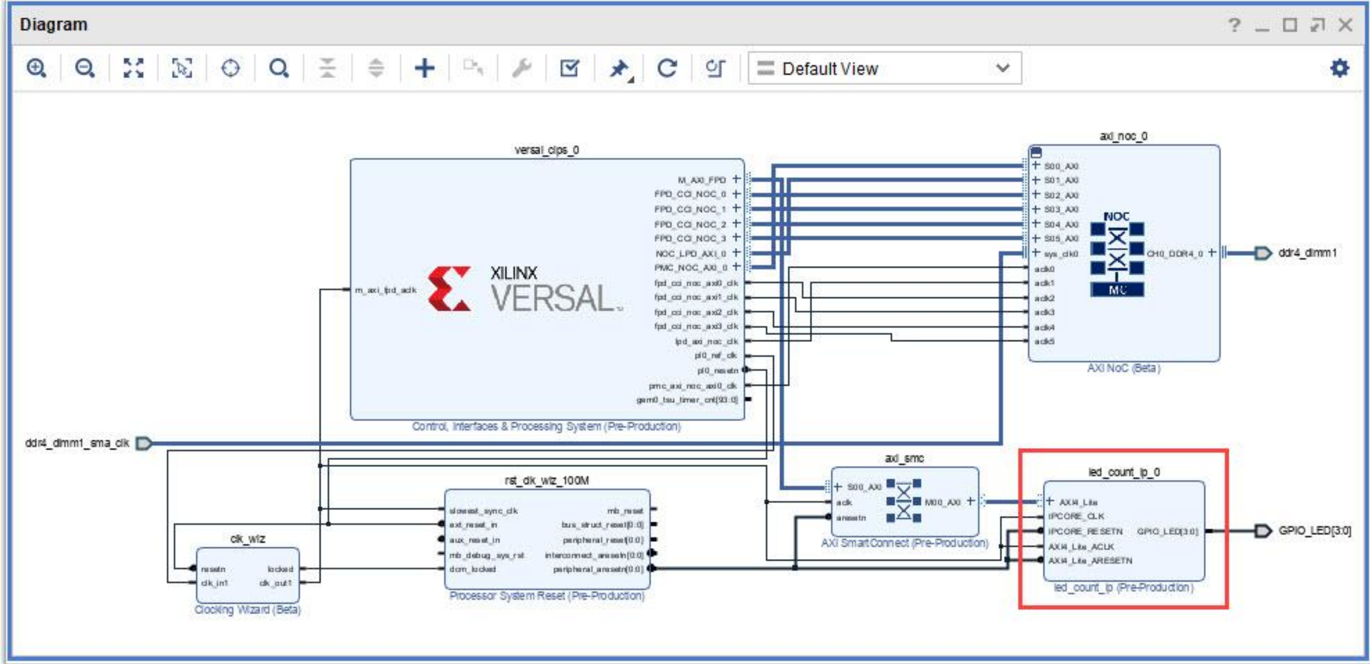
Next, insert your generated IP core into a embedded system reference design by creating a project, generating an FPGA bitstream, and downloading the bitstream to the Versal hardware.

The reference design is a predefined Xilinx Vivado project. It contains all the elements the Xilinx software needs to deploy your design to the Versal platform, except for the custom IP core and embedded software that you generate.

1. To integrate with the Xilinx Vivado environment, select the **Embedded System Integration > 4.1 Create Project** task and click **Run This Task**. A Xilinx Vivado project with an IP Integrator embedded design is generated.



The dialog window includes a link to the project. Optionally, click the link to view the project. From the block diagram in Vivado tool, you can see the HDL Coder generated IP core `led_count_ip_0` is connected to the processing system through the AXI interface.

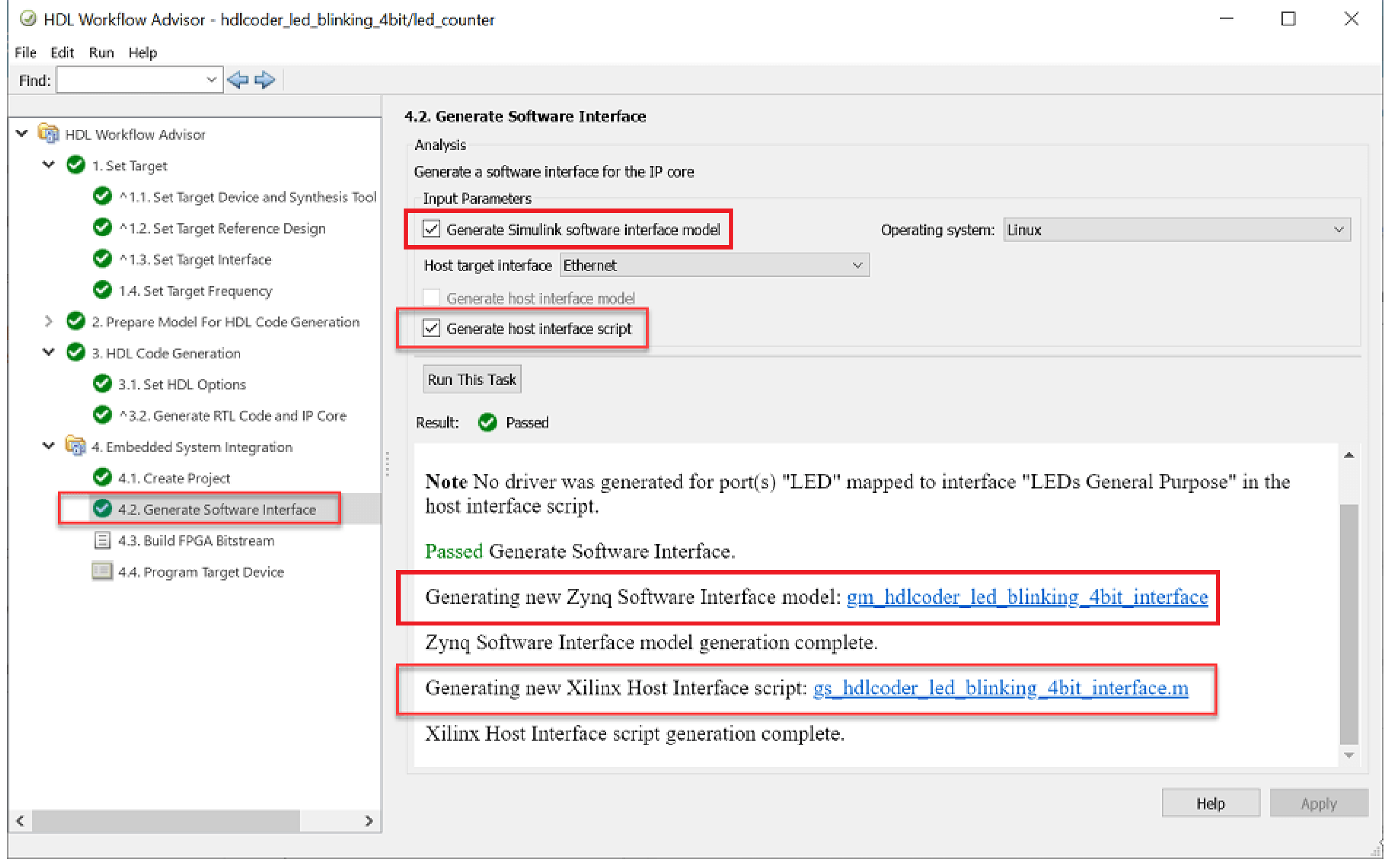


2. Next, select task **4.2 Generate Software Interface**.

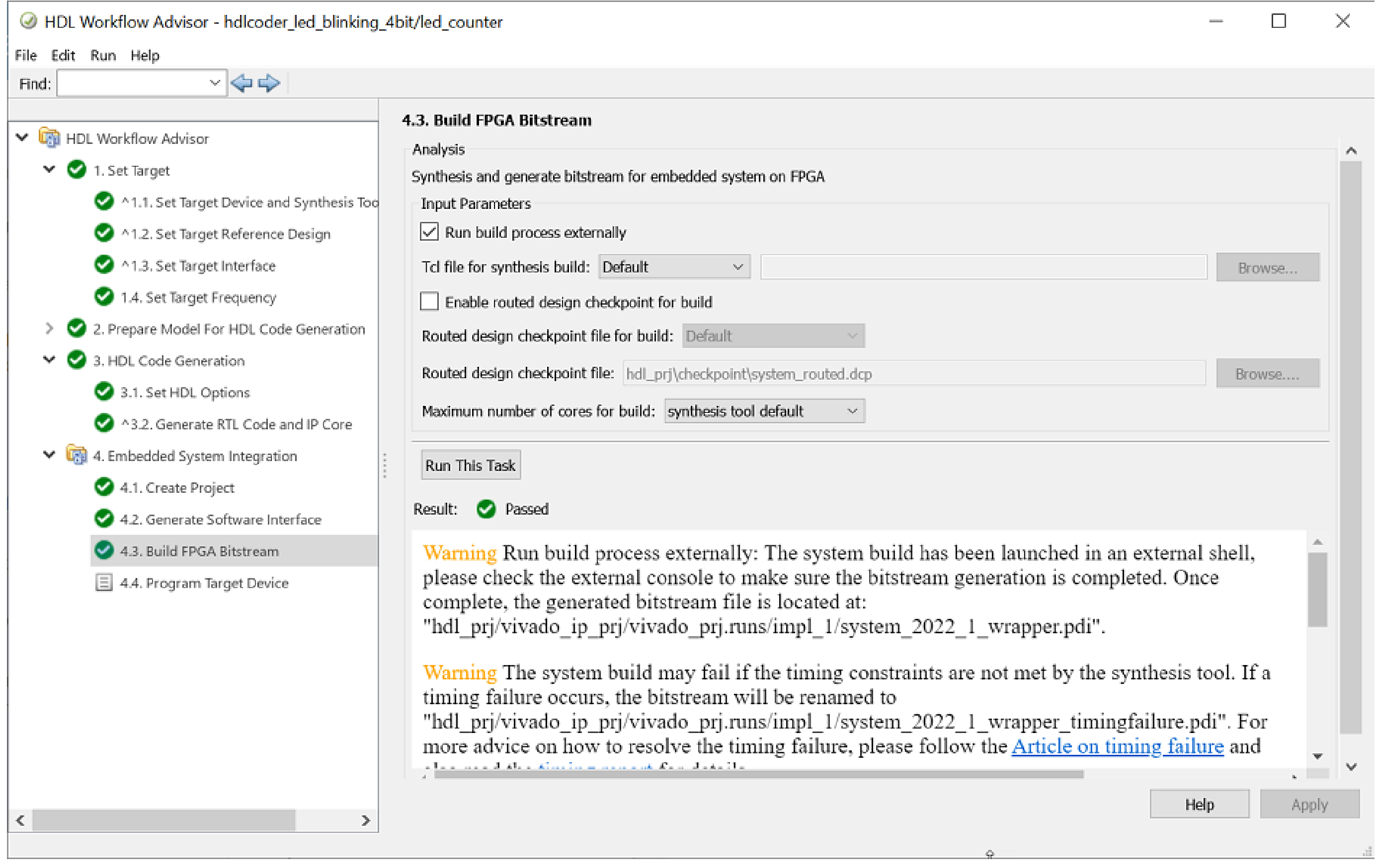
Enable the **Generate host interface script** checkbox. The host interface script contains commands that enable you to connect to the target hardware and to write to or read from the generated IP core over the AXI interface. This script will be used below in Run FPGA I/O Commands to Access AXI Registers on FPGA on page 39-278.

Enable **Generate Simulink software interface model** checkbox. The software interface model contains driver blocks that generate embedded C code and can run on the ARM processor. This model will be used below in Run Software Interface Model in External Mode on ARM Processor on page 39-279.

Click **Run This Task.**

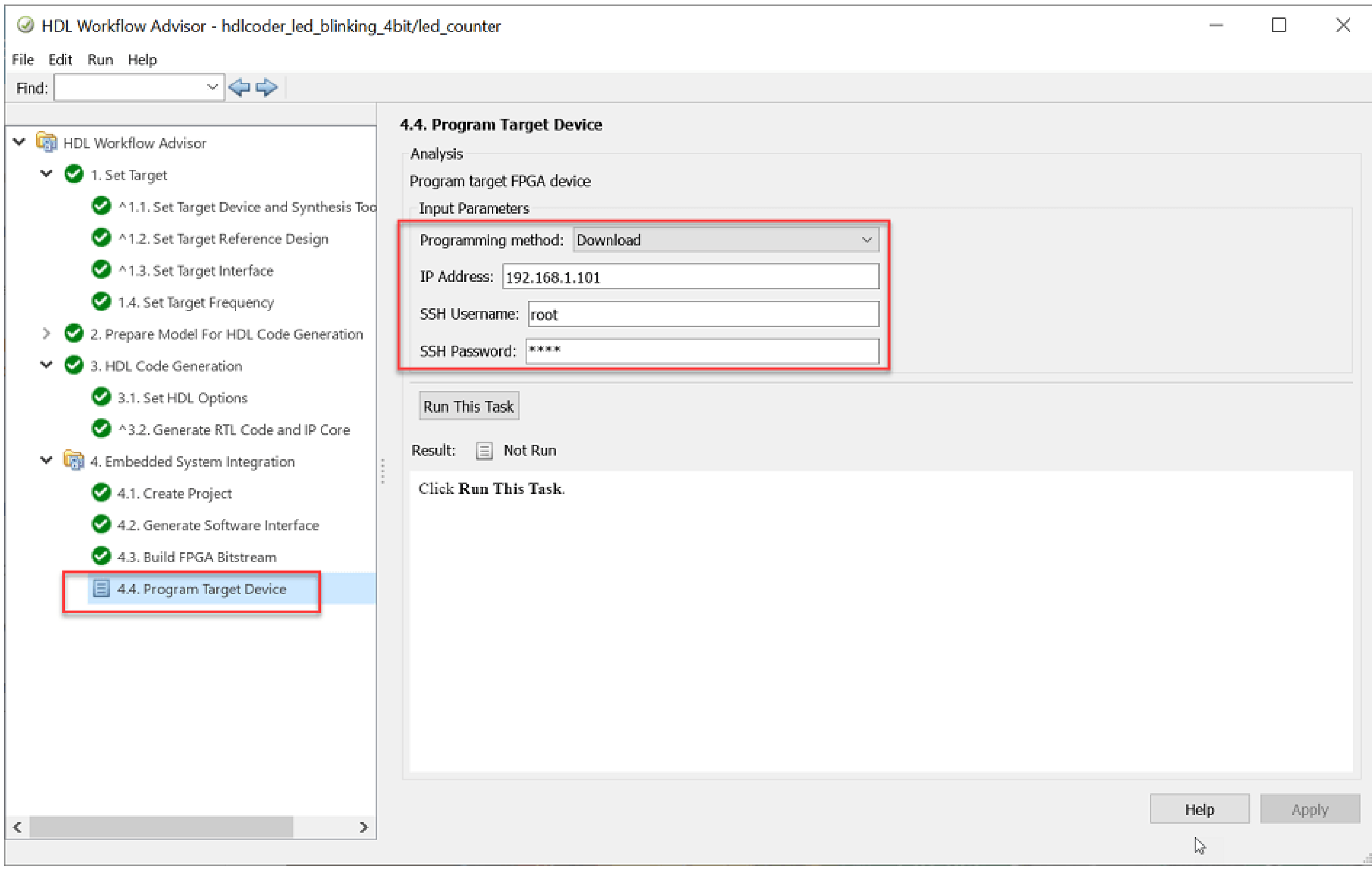


3. Build the FPGA bitstream in the **Build FPGA Bitstream** task. Make sure the **Run build process externally** check box is selected, which runs the Xilinx synthesis tool in a separate process from MATLAB. Wait for the synthesis tool process to finish running



.

4. After the bitstream is generated, select the **4.4 Program Target Device** task. Set **Programming method** to `Download`, to program the FPGA bitstream through download mode to the Xilinx Versal VCK190 board. Click **Run This Task** to program the Versal hardware. HDL Coder copies the generated bitstream to the SD card connected to the Versal hardware.



After you program the FPGA hardware, the LED starts blinking on your Xilinx Versal VCK190 board.

Run FPGA I/O Commands to Access AXI Registers on FPGA

When you select the **Generate host interface script** option, HDL Coder generates two MATLAB files, `gs_hdlcoder_led_blinking_4bit_interface.m` and `gs_hdlcoder_led_blinking_4bit_setup.m`.

- `gs_hdlcoder_led_blinking_4bit_interface.m` - This interface script creates a target object, instantiates the setup script and connects to the target hardware. The script sends read and write commands to the generated HDL IP core.
- `gs_hdlcoder_led_blinking_4bit_setup.m` - This setup function adds the AXI4 slave interface. The script also contains DUT port objects that have the port name, direction, data type, and interface mapping information. The script maps the DUT ports to the corresponding interfaces.

Follow these steps to access data from the AXI Register by using the `readPort` and `writePort` commands:

1. Set up the hardware connection by entering this command in the MATLAB Command Window.

```
hProcessor = xilinxsoc
```

The `xilinxsoc` function logs in to the hardware over SSH. You can also specify the board's IP address as part of this command.

2. Create a FPGA object by running this command:

```
hFPGA = fpga(hProcessor)
```

3. Run the `gs_hdlcoder_led_blinking_4bit_setup(hFPGA)` setup function using the generated FPGA object as the input argument. This function configures the FPGA object with the same interfaces as the generated IP core.

4. Run the `writePort` command to change the blinking frequency of LEDs:

```
writePort(hFPGA, "Blink_frequency", 6)
```

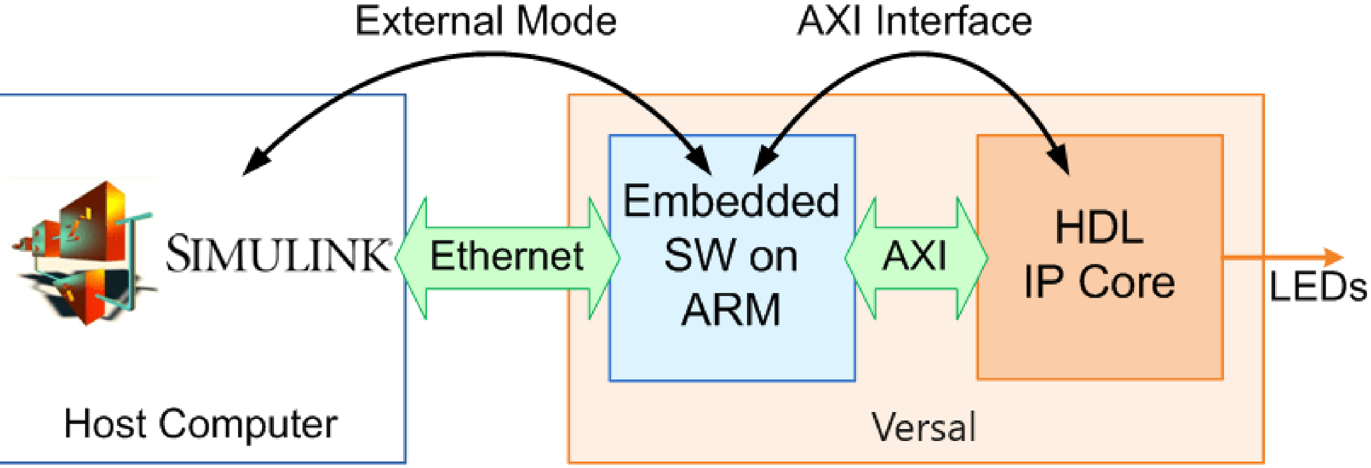
5. Use the `readPort` command to read the values from the read back port Read_back.

```
data_Read_back = readPort(hFPGA, "Read_back")
```

Run Software Interface Model in External Mode on ARM Processor

When you select the **Generate software interface model** option, HDL Coder generates a new Simulink model `gm_hdlcoder_led_blinking_4bit_interface.slx`. This software interface model is used to deploy the software portion of your algorithm. It contains driver blocks which communicate with the algorithm running on the FPGA through the AXI interface. This model generates embedded C code and runs on the ARM processor in the Versal hardware.

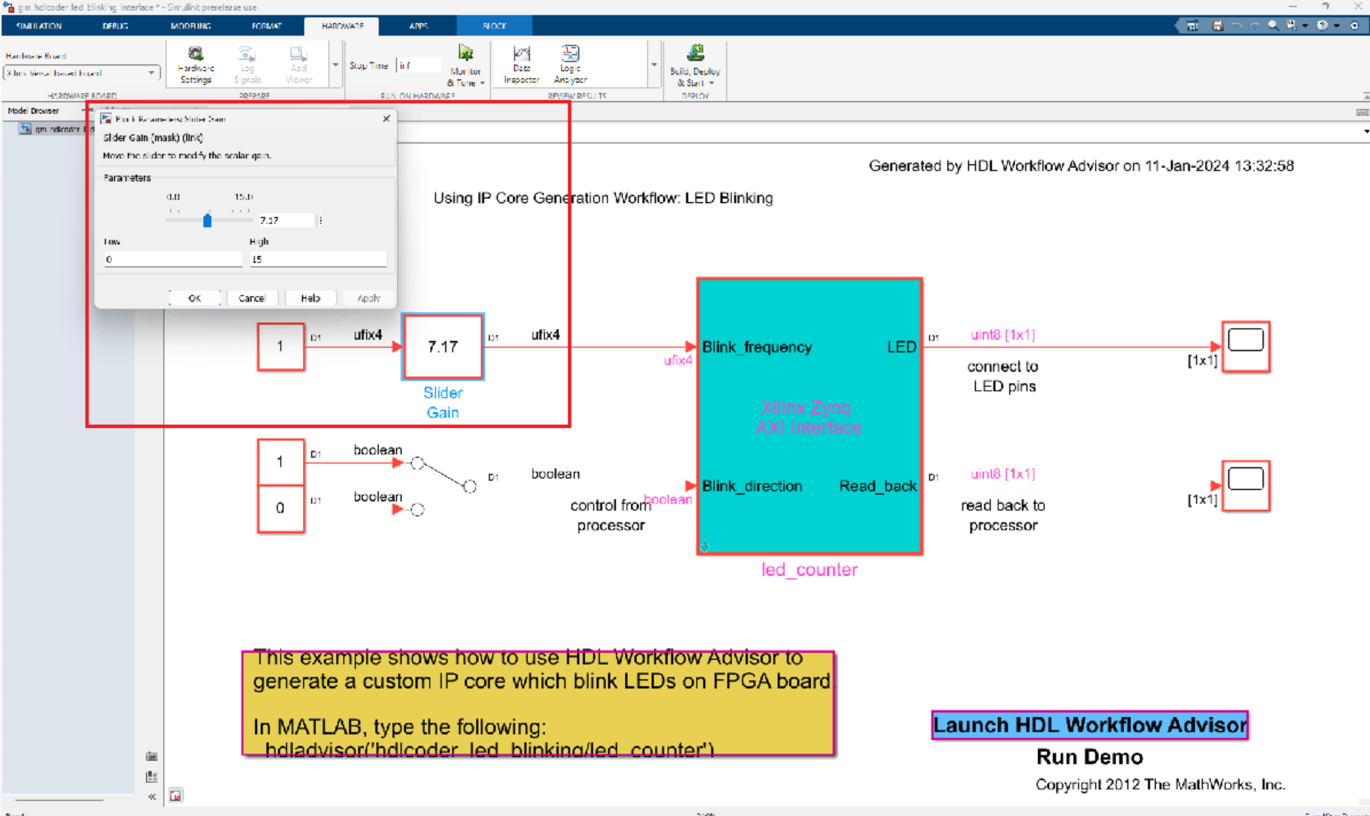
When you are prototyping and developing an algorithm, it is useful to monitor and tune the algorithm while it runs on hardware. The External mode feature in Simulink enables this capability. In this mode, the software portion of your algorithm is first deployed to the ARM processor in the Versal hardware, and then linked with the Simulink model on the host computer through an Ethernet connection. Because the ARM processor is connected to the HDL IP core through the AXI interface, you can use External mode to tune parameters, and capture data from the FPGA.



1. In the generated software interface model, click on the **HARDWARE** tab in the Simulink toolstrip.
2. Set **Stop Time** to **inf** on the toolstrip.
3. Click the **Monitor & Tune** button on the toolstrip to run your model on the ARM processor in External mode. Embedded Coder builds the model, downloads the ARM executable to the Versal hardware, executes it, and connects the model to the executable running on the hardware.
4. Double-click the **Slider Gain** block. Change the Slider Gain value and observe the change in frequency of the LED array blinking on the Versal hardware. Double-click the **Manual Switch** block to switch the direction of the blinking LEDs.

5 Double-click the scope connected to the **Read_back** output port and observe that the output data of the FPGA IP core is captured and sent back to the Simulink scope.

6 When you are done changing model parameters, click the **Stop** button on the model.



Summary

This example shows how the hardware-software co-design workflow helps automate the deployment of your MATLAB and Simulink design to a Xilinx Versal AI Core Series VCK190 Evaluation Kit. You can explore the best ways to partition and deploy your design by iterating through the workflow.

To learn more about the hardware and software co-design workflow, see “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9.

Target SoC Platforms and Speedgoat Boards

- “Model Design for AXI4 Slave Interface Generation” on page 40-3
- “Model Design for AXI4-Stream Interface Generation” on page 40-14
- “Model Design for Frame-Based IP Core Generation” on page 40-29
- “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 40-33
- “Running Audio Filter with Multiple AXI4-Stream Channels on ZedBoard” on page 40-38
- “Inspect the Written Values of AXI4 Slave Registers by Using the Readback Methods” on page 40-51
- “Use MATLAB FPGA I/O Host Interface to Communicate with FPGA on Zynq-Based Radio” on page 40-67
- “Multirate IP Core Generation” on page 40-85
- “Board and Reference Design Registration System” on page 40-89
- “Register a Custom Board” on page 40-92
- “Register a Custom Reference Design” on page 40-95
- “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 40-98
- “Customize Reference Design Dynamically Based on Reference Design Parameters” on page 40-104
- “Define and Add IP Repository to Custom Reference Design” on page 40-109
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 40-113
- “Model Design for AXI4-Stream Video Interface Generation” on page 40-118
- “Model Design for AXI4 Master Interface Generation” on page 40-128
- “IP Core Generation Workflow for Standalone FPGA Devices” on page 40-141
- “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 40-145
- “Map Bus Data Types to PCIe Interface” on page 40-148
- “IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip” on page 40-150
- “Running an Audio Filter on Live Audio Input Using Intel Board” on page 40-170
- “Running an Audio Filter on Live Audio Input Using a Zynq Board” on page 40-181
- “Deploy Model with AXI-Stream Interface in Zynq Workflow” on page 40-192
- “Deploy Model with AXI4-Stream Video Interface on Zynq Hardware” on page 40-207
- “Perform Matrix Operation Using External Memory” on page 40-218
- “Authoring a Reference Design for Audio System on a Zynq Board” on page 40-226
- “Authoring a Reference Design for Audio System on a ZYBO Board” on page 40-236
- “Authoring a Reference Design for Audio System on Intel Board” on page 40-242

- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270
- “Define Custom Board and Reference Design for Microchip Workflow” on page 40-285
- “Define Custom Board and Reference Design for Microchip Pure FPGA Platforms” on page 40-305
- “Dynamically Create Reference Design with Master Only or Slave Only AXI4-Stream Interface” on page 40-320
- “Use JTAG AXI Manager to Control HDL Coder Generated IP Core” on page 40-333
- “Debug a Zynq Design Using HDL Coder and Embedded Coder” on page 40-346
- “Debug IP Core Using FPGA Data Capture” on page 40-351
- “Field-Oriented Control of a Permanent Magnet Synchronous Machine on a Xilinx Zynq Platform” on page 40-361
- “Deploy a Frame-Based Model with AXI4-Stream Interfaces” on page 40-378
- “Deploy Frame-Based Models with AXI4-Stream Video Interfaces in Zynq-Based Hardware” on page 40-386
- “DAC and ADC Loopback Data Capture” on page 40-392
- “IQ Mixer Mode Capture” on page 40-404
- “PL-DDR4 ADC Data Capture” on page 40-410
- “DAC PL-DDR4 Transmit” on page 40-419
- “Polyphase Channelizer” on page 40-426
- “Multi-Tile Synchronization” on page 40-430
- “Enable Clock Domain Crossing on AXI4-Lite Interfaces” on page 40-439
- “Use Clock Domain Crossing to Run DUT Algorithm and AXI4-Lite Interface at Different Frequencies” on page 40-442

Model Design for AXI4 Slave Interface Generation

In this section...

"Considerations" on page 40-3
"Map Scalar Ports to AXI4 Slave Interface" on page 40-3
"Map Vector Ports to AXI4 Slave Interface" on page 40-4
"Map Double Data Types and Data Larger than 32 bits to AXI4-Slave Interfaces" on page 40-5
"Map Bus Data Types to AXI4 Slave Interface" on page 40-8
"Specify Initial Value of AXI4 Slave Registers" on page 40-9
"Read Back Value of AXI4 Slave Interfaces" on page 40-10
"Optimize AXI4 Slave Read Logic" on page 40-13

To perform lightweight data transfer or to access control registers, use AXI4 slave interfaces. The AXI4 slave interfaces include the AXI4 and AXI4-Lite interfaces. With the HDL Coder software, you don't have to implement AXI4 or AXI4-Lite protocol in your model. The software generates AXI4 or AXI4-Lite interfaces in the HDL IP core.

When you model your design, specify the data ports, you want to map to the AXI4 slave interfaces. HDL Coder then maps the data ports to memory-mapped registers and allocates address offsets for the ports.

Considerations

When you map your DUT ports to AXI4 or AXI4-Lite interfaces:

- You can map all scalar, vector, or bus ports in your design to either AXI4 or AXI4-Lite interfaces.
- You cannot map some DUT ports to AXI4 interfaces and other DUT ports to AXI4-Lite interfaces for the same design.

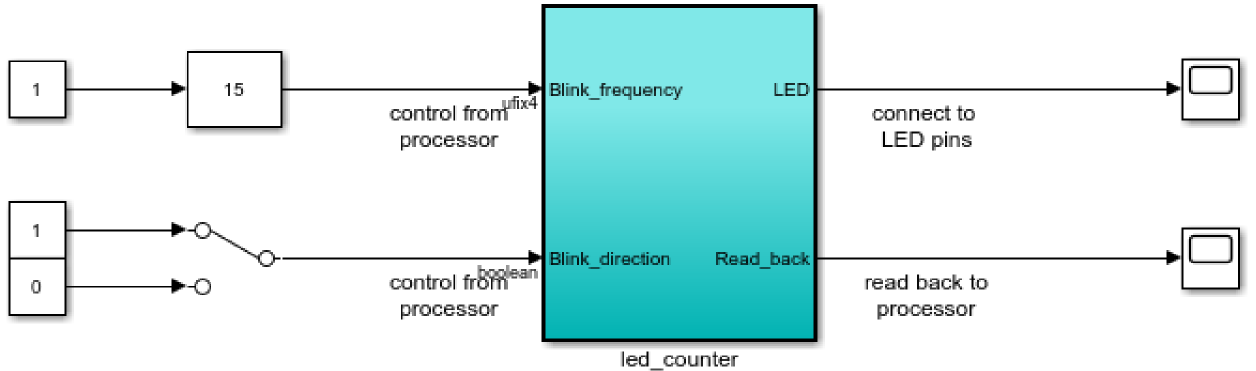
Map Scalar Ports to AXI4 Slave Interface

When you use scalar data types at the DUT interface ports, you can directly map the interface ports to AXI4 or AXI4-Lite interfaces. The code generator assigns a unique address to each data port that you want to map to the AXI4 interface.

For an example that shows how to map scalar ports to AXI4-Lite interfaces, open the model `hdlcoder_led_blinking`.

```
openExample('hdlcoder/IPCoreGenWorkflowWithAMicroBlazeProcessorKC705Example',...)
'supportingFile','hdlcoder_led_vector.slx')
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
hdladvisor('hdlcoder_led_blinking/led_counter')

Launch HDL Workflow Advisor

Run Demo

Copyright 2012 The MathWorks, Inc.

In this model, the subsystem `led_counter` is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, `Blink_frequency` and `Blink_direction`, are control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem `led_counter` are for software implementation.

In Simulink, you can use the Slider Gain block or the Manual Switch block to adjust the hardware subsystem's input values. The ARM processor controls the generated IP core by writing to the AXI interface accessible registers in the embedded software. The output port of the hardware subsystem connects to the LED hardware. You can use the output port `Read_back` to read data back to the processor.

When you run the `IP Core Generation` workflow, in the **Set Target Interface** task, you see that the ports `Blink_frequency`, `Blink_direction`, and `Read_back` map to AXI4-Lite interfaces.

To learn more about this example, see:

- "Getting Started with Targeting Xilinx Zynq Platform" on page 39-98
- "Getting Started with Targeting Intel SoC Devices" on page 39-132

Map Vector Ports to AXI4 Slave Interface

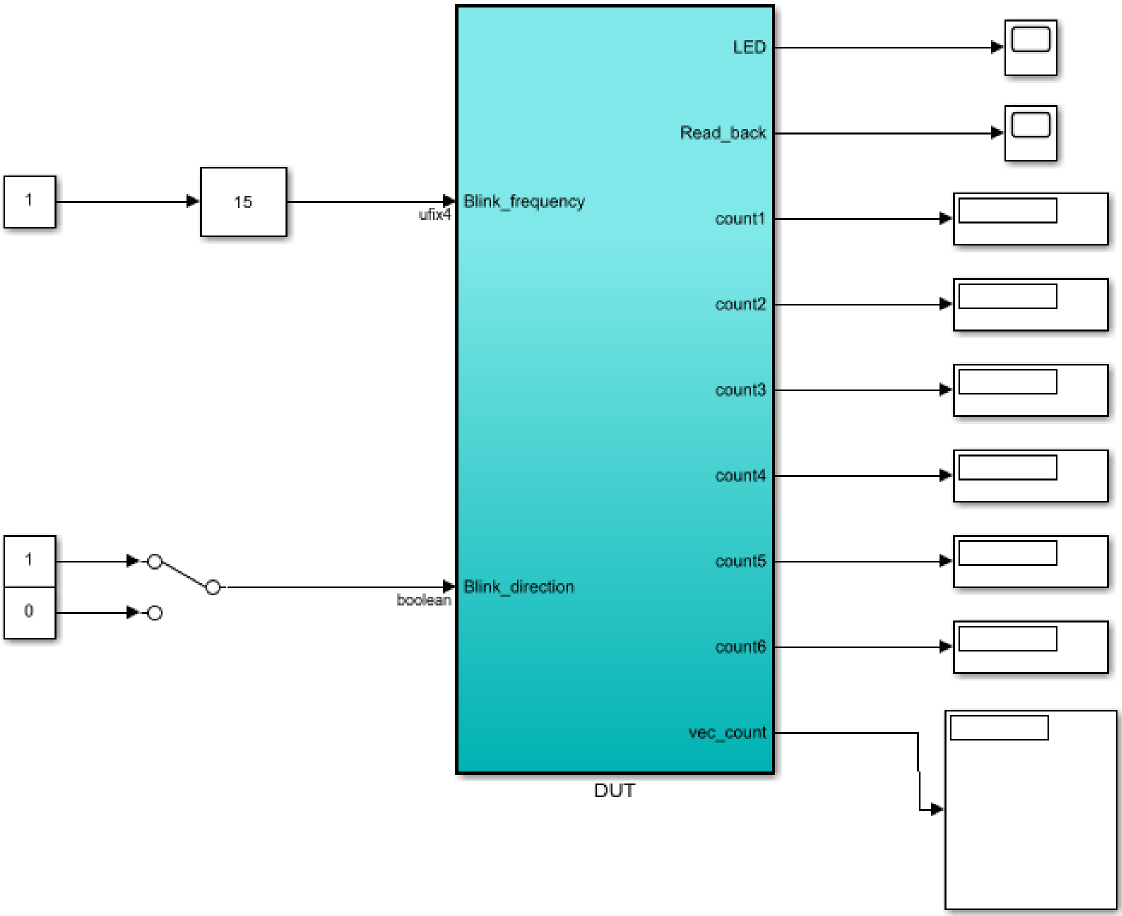
When you use vector data types at the DUT interface ports, you can directly map the interface ports to AXI4 or AXI4-Lite interfaces. The code generator assigns a unique address for each data port that you want to map to the AXI4 interface.

When you map vector ports, HDL Coder uses additional strobe registers for each port to maintain the synchronization with the IP core algorithm logic. For input ports, the strobe registers control the

enable signals for a set of shadow registers, making the IP core algorithm logic see the updated vector elements simultaneously. For output ports, the strobe registers make sure that the vector data to be read is captured synchronously.

For an example that shows how to map vector ports to AXI4-Lite interfaces, open the model `hdlcoder_led_vector`.

```
open_system('hdlcoder_led_vector')
```



In this model, the subsystem `DUT` implements the LED blinking algorithm and has vector output ports. When you run the `IP Core Generation` workflow, you see that the input ports and output ports map to AXI4-Lite interfaces in the **Set Target Interface** task.

To learn more, see “IP Core Generation Workflow with a MicroBlaze processor: Xilinx Kintex-7 KC705” on page 39-211.

Map Double Data Types and Data Larger than 32 bits to AXI4-Slave Interfaces

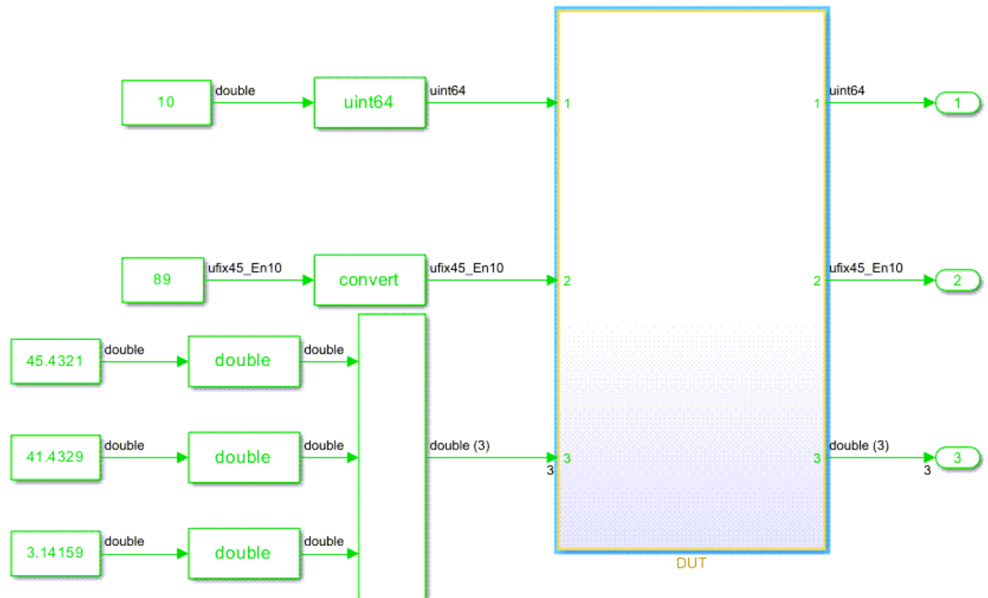
When your DUT port has double data types or data types with width greater than 32 bits, HDL Coder:

- Splits the data into individual 32 bit words to match the register width of the AXI4 or AXI4-Lite interface.
- Assigns each 32 bit word to an individual address. You can find the resulting start and end addresses in the IP Core Generation report under the Register Address Mapping section.
- Creates an additional strobe register to synchronize the data at the DUT boundary. The strobe logic behaves the same as the strobe register for vector ports. For more information on how vector ports are handled, see “Map Vector Ports to AXI4 Slave Interface” on page 40-4.

Learn how HDL Coder splits and maps data with bit widths greater than 32 bits to individual addresses. In this example model there are three input ports and three output ports:

- Input and output port one: Scalar with `uint64` datatype.
- Input and output port two: Scalar with `ufix45_En10` datatype.
- Input and output port three: Vector of three elements with 64 bit datatype.

This image shows the example DUT with the corresponding datatypes on the input and output ports. Each port is mapped to the AXI4 interface.



When HDL Coder processes the DUT it converts the data into 32 bit words and maps each 32 bit word to a specific address. This image shows the target interface configuration and register address mapping sections of the generated IP core report. The register address mapping section mentions the start address, number of 32 bit words and the final register address for data with bit widths greater than 32 bits.

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
In1	Inport	uint64	AXI4	x"100"	
In2	Inport	ufix45_En10	AXI4	x"110"	
In3	Inport	double (3)	AXI4	x"120"	
Out1	Outport	uint64	AXI4	x"148"	
Out2	Outport	ufix45_En10	AXI4	x"158"	
Out3	Outport	double (3)	AXI4	x"180"	

Register Address Mapping

The following AXI4 bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
IPCore_Timestamp	0x8	contains unique IP timestamp (yymmddHHMM): 2301051005
In1_Data	0x100	data register for Inport In1. Data width is wider than the register width, so data is split into 2 32-bit sections.. Register is split across a total of 2 addresses, last address is 0x104.
In1_Strobe	0x108	strobe register for port In1
In2_Data	0x110	data register for Inport In2. Data width is wider than the register width, so data is split into 2 32-bit sections.. Register is split across a total of 2 addresses, last address is 0x114.
In2_Strobe	0x118	strobe register for port In2
In3_Data	0x120	data register for Inport In3. Vector with 3 elements. Data width is wider than the register width, so data is split into 2 32-bit sections.. Register is split across a total of 6 addresses, last address is 0x134.
In3_Strobe	0x140	strobe register for port In3
Out1_Data	0x148	data register for Outport Out1. Data width is wider than the register width, so data is split into 2 32-bit sections.. Register is split across a total of 2 addresses, last address is 0x14C.
Out1_Strobe	0x150	strobe register for port Out1
Out2_Data	0x158	data register for Outport Out2. Data width is wider than the register width, so data is split into 2 32-bit sections.. Register is split across a total of 2 addresses, last address is 0x15C.
Out2_Strobe	0x160	strobe register for port Out2
Out3_Data	0x180	data register for Outport Out3. Vector with 3 elements. Data width is wider than the register width, so data is split into 2 32-bit sections.. Register is split across a total of 6 addresses, last address is 0x194.
Out3_Strobe	0x1A0	strobe register for port Out3

For input port one the 64 bit data is split into two 32 bit words. The two words are mapped to addresses `0x100` and `0x104`. Bits zero through 31 are mapped to `0x100` and bits 32 through 63 are mapped to `0x104`. Output port one is split across addresses `0x148` and `0x14C`.

For input port two the 45 bit data is split into two 32 bit words. The two words are mapped to addresses `0x110` and `0x114`. Bits zero through 31 are mapped to `0x110` and bits 32 through 63 are mapped to `0x114`. Output port two is split across addresses `0x158` and `0x15C`.

For input port three, each vector element is split into two 32 bit words. This results in a total of six 32 bit words for the entire vector. The six words are mapped to the addresses 0x120, 0x124, 0x128, 0x12C, 0x130, and 0x134.

- Bits zero through 31 of the first vector element are mapped to 0x120.
- Bits 32 through 63 of the first vector element are mapped to 0x124.
- Bits zero through 31 of the second vector element are mapped to 0x128.
- Bits 32 through 63 of the second vector element are mapped to 0x12C.
- Bits zero through 31 of the third vector element are mapped to 0x130.
- Bits 32 through 63 of the third vector element are mapped to 0x134.

Output port three is split into 6 32 bit words. The six words are mapped to addresses 0x180, 0x184, 0x188, 0x18C, 0x190, 0x194.

Limitations

HDL Coder does not support mapping::

- Bit widths greater than 128 bits to AXI4-Slave ports
- Complex number data to AXI4-Slave ports
- Shift register data to AXI4-Slave ports

Map Bus Data Types to AXI4 Slave Interface

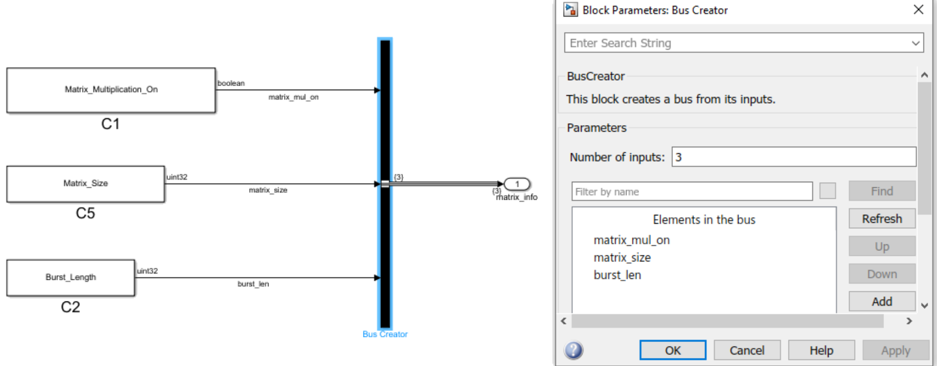
When you use bus data types at the DUT interface ports, you can directly map the interface ports to AXI4 or AXI4-Lite interfaces.

When you map a port with bus data types to an AXI slave Interface, HDL Coder assigns a unique address for each bus element. HDL Coder treats bus ports as a group of independent scalar and vector ports. When HDL Coder assigns an address to bus elements, they are treated as separate registers and the addresses are not contiguous. When you change the address of a bus port in the target interface table, only the address of the first element changes.

Model Bus Element

Model a bus element by using a bus creator block or bus element block to create a bus port.

Model a bus element by using a bus creator block.



Model a bus element by using bus element blocks:

For more information, see “Map Bus Data Types to AXI4 Slave Interfaces” on page 39-234.

Specify Initial Value of AXI4 Slave Registers

When you run the `IP Core Generation` workflow or the `Simulink Real-Time FPGA I/O` workflow, you can specify an initial value for input ports mapped to the AXI4 slave registers. You can specify an initial value when mapping to these target interfaces:

- AXI4
- AXI4-Lite
- PCIe

By default, the initial value is zero. To specify a nonzero value:

1. In the target platform interface table, when you map an input DUT port to an AXI4 slave interface, an **Options** button appears in the **Interface Options** column.
2. Click the **Options** button, and then specify the **RegisterInitialValue**.

The specified value is saved on the DUT Inport blocks as the HDL block property **IOInterfaceOptions** in the **Target Specification** tab. For example, if you map a DUT input port to AXI4-Lite interface, set **RegisterInitialValue** to `5`, and then run the **Set Target Interface** task. The **IOInterfaceOptions** property of that input port is saved with the value `{'RegisterInitialValue','5'}`.

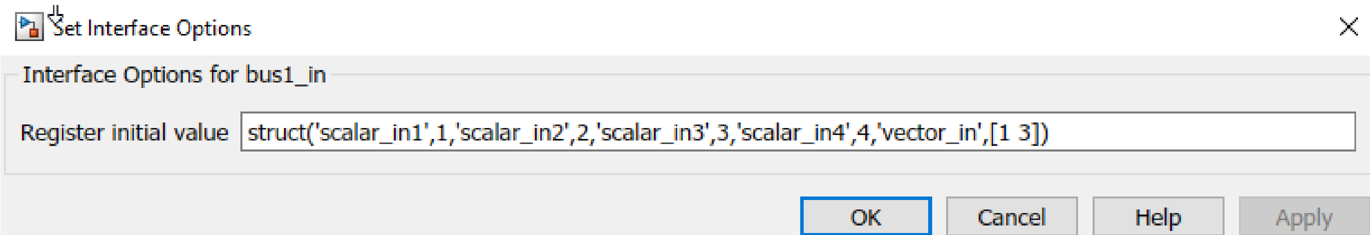
To view the `IOInterfaceOptions` value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

```
hdlget_param('hdlcoder_led_blinking/led_counter/LED',...
             'IOInterfaceOptions')
```

Specify the initial value for scalar ports.

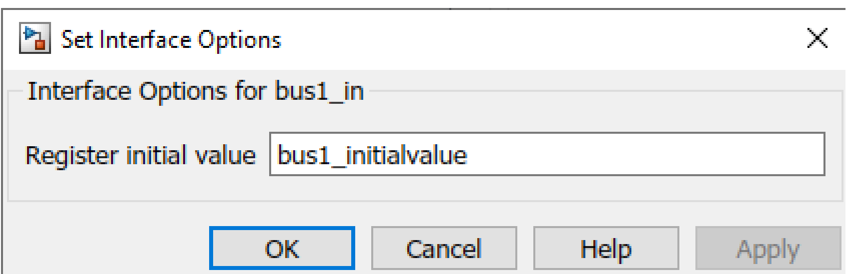
```
hdlset_param('hdlcoder_led_blinking/led_counter/LED', ...
     'IOInterfaceOptions', {'RegisterInitialValue','5'});
```

To set the initial value for a bus, specify a struct whose field names match the bus element names. For example,



You can also specify the initial value by using a variable defined in the MATLAB workspace. For example:

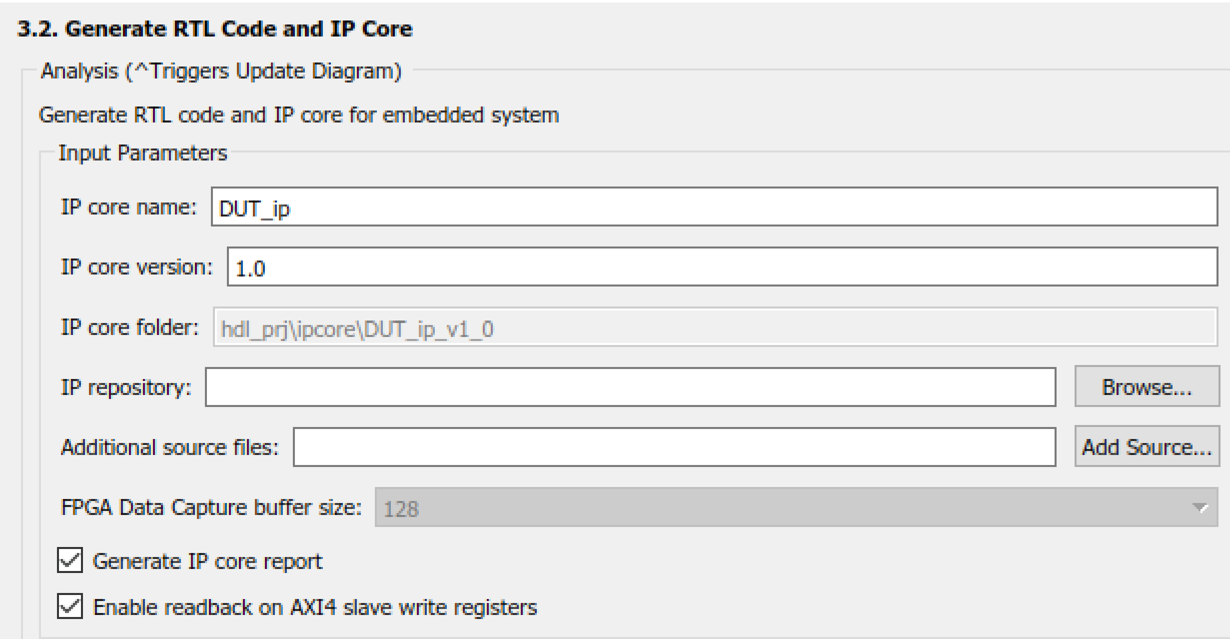
```
bus1_initialvalue = struct('scalar_in1',1,'scalar_in2',2,'scalar_in3',3,'scalar_in4',4,'vector_in',[1 3])
```



Read Back Value of AXI4 Slave Interfaces

When you run the `IP Core Generation` workflow, you can read back the value that is written to the AXI4 slave registers by using the AXI4 slave interface. For example, you can read back the values that are written to the AXI4 slave registers by using the `devmem` command in the Linux console of the ARM processor. If you have HDL Verifier installed, you can use the AXI Manager IP to read back the values.

To use this capability, in the **Generate RTL Code and IP Core** task of the `IP Core Generation` workflow, select the **Enable read back on AXI4 slave write registers** check box, and then run the **Generate RTL Code and IP Core** task



When you run this task, HDL Coder saves the read back setting that you enabled on the model. In the HDL Block Properties of the DUT Subsystem, on the **IP Core Parameter** section of the **Target Specification** tab, you see a parameter **AXI4RegisterReadback** set to `on`. If you export the HDL Workflow Advisor run to a script, you see this setting saved on the model by using `hdlset_param`.

```
hdlset_param('hdlcoder_led_vector/DUT', 'AXI4RegisterReadback', 'on');
```

These examples show how you can read back values by using the `devmem` command in the Linux console with a program such as PuTTy.

To read back values when mapping scalar ports to AXI4 interfaces, you first write values to the AXI4 registers, and then read back the values. You can see the memory address of the AXI4 registers in the IP Core Generation report.

COM7 - PuTTY

```
zynq>
zynq>
zynq>
zynq> devmem 0x400d0100
0x00000000
zynq> devmem 0x400d0100 w 0x1
zynq> devmem 0x400d0100
0x00000001
zynq> devmem 0x400d0100 w 0x2
zynq> devmem 0x400d0100
0x00000002
zynq>
```

To read back values when mapping vector ports to AXI4 interfaces, you first write to the AXI4 registers, then write the strobe register address with `0x1`, and then read back the values. You can see the memory address of the AXI4 registers and the strobe register in the IP Core Generation report.

COM7 - PuTTY

```
zynq>
zynq>
zynq>
zynq> devmem 0x400d0100
0x00000000
zynq> devmem 0x400d0104
0x00000000
zynq> devmem 0x400d0100 w 0x2
zynq> devmem 0x400d0110 w 0x1
zynq> devmem 0x400d0100
0x00000002
zynq> devmem 0x400d0104 w 0x3
zynq> devmem 0x400d0110 w 0x1
zynq> devmem 0x400d0104
0x00000003
zynq>
```

Optimize AXI4 Slave Read Logic

When your model contains several output registers and you want to read back data from multiple AXI4 slave registers, the read back logic becomes a long mux chain that can reduce the synthesis frequency. If you select the **Enable readback on AXI4 slave write registers** setting in the **Generate RTL Code and IP Core** task, HDL Coder adds a mux for each AXI4 register in the Address Decoder logic. As the number of AXI4 slave registers increases, the mux chain becomes longer, which further reduces the synthesis frequency.

You can optimize the readback logic and achieve the target frequency that you want. When you run the `IP Core Generation` workflow, in the **Generate RTL Code and IP Core** task, you see a setting **AX4 slave port to pipeline register ratio**. The default value of this setting is `auto`. This setting indicates how many AXI4 slave registers a pipeline register is inserted for. For example, an **AX4 slave port to pipeline register ratio** of `20` means that one pipeline register is inserted for every `20` AXI slave registers. The `auto` setting means that the code generator inserts a certain number of pipelines for the AXI4 slave ports depending on the number of ports and the synthesis tool that you specify. You can disable this setting or specify one of these valid values, off, 10, 20, 35, 50.

When you run this task, HDL Coder saves the value that you specified for the setting on the model. In the HDL Block Properties of the DUT Subsystem, on the **IP Core Parameter** section of the **Target Specification** tab, you see a parameter **AX4SlavePortToPipelineRegisterRatio** set to the value that you specified. If you export the HDL Workflow Advisor run to a script, you see this setting saved on the model by using `hdlset_param`.

```
hdlset_param('hdlcoder_led_vector/DUT', ...
                'AXI4SlavePortToPipelineRegisterRatio', '20');
```

See Also

Related Examples

- “Use Clock Domain Crossing to Run DUT Algorithm and AXI4-Lite Interface at Different Frequencies” on page 40-442

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- “Custom IP Core Generation” on page 39-17
- “Enable Clock Domain Crossing on AXI4-Lite Interfaces” on page 40-439

Model Design for AXI4-Stream Interface Generation

In this section...

"Sample-Based Modeling" on page 40-14
"Frame-Based Modeling" on page 40-21
"Legacy Frame-Based Modeling" on page 40-21
"Map Vector Ports to AXI4-Stream Interfaces " on page 40-21
"Model Designs with Multiple Streaming Channels" on page 40-25
"Model Designs That Have Multiple Sample Rates" on page 40-25
"Interface Options for AXI4-Stream Data" on page 40-26
"Restrictions" on page 40-28

For designs that require high speed data transfers use AXI4-Stream interfaces. You can implement a simplified, streaming protocol in your model by using HDL Coder. The software generates AXI4-Stream interfaces in the IP core.

Choose from these three modeling styles based on how your algorithm operates:

- **Sample-Based Modeling** — Use these guidelines when your algorithm operates on a stream of samples.
- **Frame-Based Modeling** — Use these guidelines when your algorithm operates on a complete frame of data. The data signals at the design under test (DUT) boundary can be vectors or matrices. Do not use this mode if you want to model the Valid and Ready signals.
- **Legacy Frame-Based Modeling** — Use these guidelines when your algorithm operates on a stream of samples and you want to simulate the data signal as a frame on the boundary of the design under test (DUT).

Note The **Legacy Frame-Based Modeling** style will be deprecated in a future release. If you want to model the Valid and Ready signals, use the sample-based modeling style.

Sample-Based Modeling

When you want to simulate the data signal as a stream of samples on the DUT boundary, model in sample-based mode. You can model the data signal as either a scalar or a vector. If you model the data signal as a vector, in the HDL Coder Workflow Advisor **Task 1.2. Set Target Interface > Interface Options** set the **Sample Packing Dimension** to `All`. HDL Coder packs the vector elements together and treats the vector as a single sample. You can specify how HDL Coder packs the data by using the "Packing Mode" on page 40-26. See "Sample Packing Dimension" on page 40-26.

Simplified Streaming Protocol

To map the design under test (DUT) ports to AXI4-Stream interfaces, use the simplified AXI4-Stream protocol. You do not have to model the actual AXI4-Stream protocol and instead can use the simplified protocol. When you run the `IP Core Generation` workflow, the generated HDL code contains wrapper logic that translates between the simplified protocol and the actual AXI4-Stream protocol. The simplified protocol requires fewer protocol signals, eases the handshaking mechanism between valid and ready signals, and supports bursts of arbitrary lengths.

Use the simplified AXI4-Stream protocol for write and read transactions. When you want to generate an AXI4-Stream interface in your IP core, in your DUT interface, implement these signals:

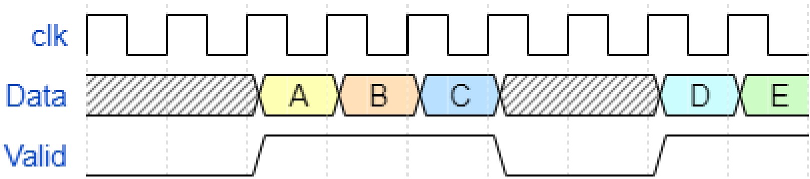
- `Data`
- `Valid`

Optionally, when you map scalar DUT ports to an AXI4-Stream interface, you can model these signals:

- `Ready`
- Other protocol signals, such as:
  - `TSRTB`
  - `TKEEP`
  - `TLAST`
  - `TID`
  - `TDEST`
  - `TUSER`

Data and Valid Signals

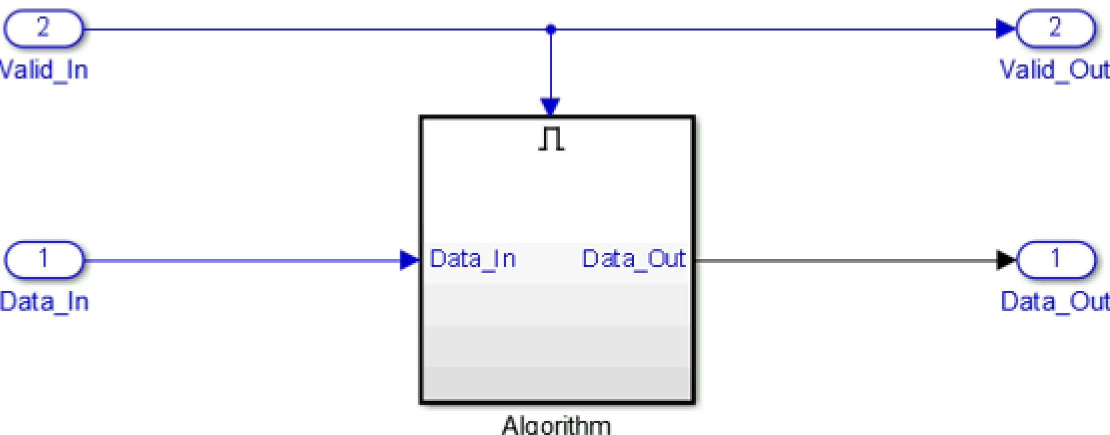
When the `Data` signal is valid, the `Valid` signal is asserted. This diagram illustrates the `Data` and `Valid` signal relationship according to the simplified streaming protocol. When you run the IP core generation workflow, HDL Coder adds a streaming interface module in the HDL IP core that translates the simplified protocol to the full AXI4-stream protocol. In this image, the clock signal is `clk`.



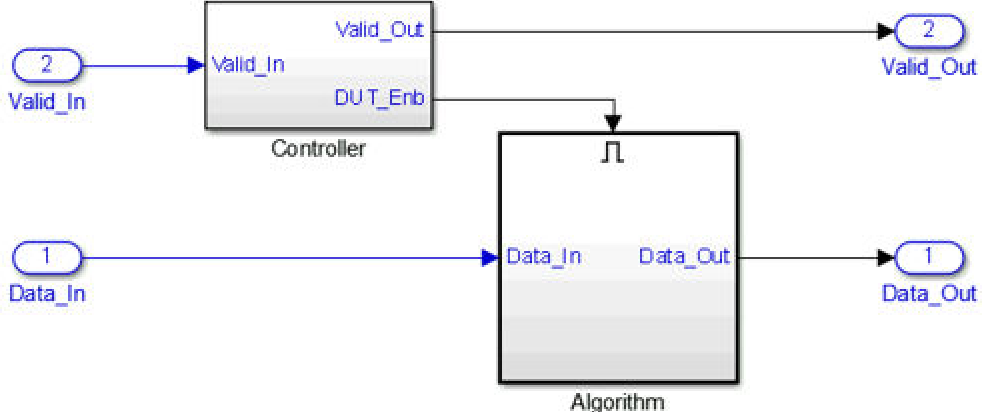
Model Data and Valid Signals in Simulink

1. Enclose the algorithm that processes the `Data` signal by using an enabled subsystem.
2. Control the enable port of the enabled subsystem by using the `Valid` signal.

For example, you can directly connect the `Valid` signal to the enable port.



You can also use a controller in your DUT that generates an enable signal for the enabled subsystem.

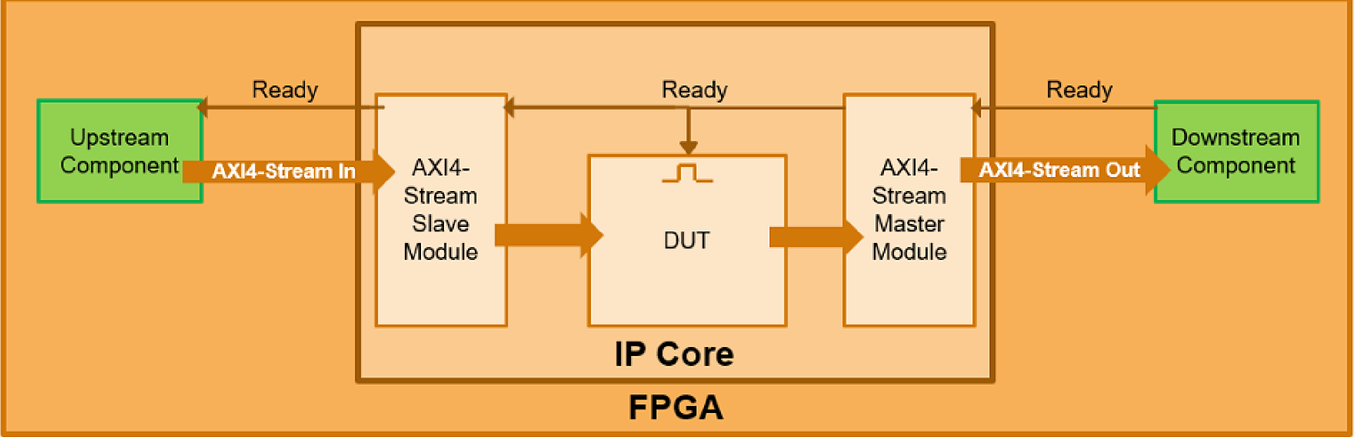


Ready Signal (Optional)

Downstream components use back pressure to tell upstream components they are not ready to receive data. The AXI4-Stream interfaces in your DUT can optionally include a Ready signal. Use the Ready signal to:

- Apply back pressure in an AXI4-Stream slave interface. For example, drop the Ready signal when the downstream component is not ready to receive data.
- Respond to back pressure in an AXI4-Stream master interface. For example, stop sending data when the downstream component Ready signal is low.

When you use a single streaming channel, by default, HDL Coder generates the Ready signal and the logic to handle the back pressure. The back pressure logic ties the Ready signal to the DUT Enable signal. When the input master Ready signal is low, the DUT is disabled, and the output slave Ready signal is driven low. Because HDL Coder generates the back pressure logic and Ready signal, when you use a single streaming channel, the Ready signal is optional and you do not have to model this signal at the DUT port.



When you use multiple streaming channels, HDL Coder generates a ready signal and does not generate the back pressure logic. In a DUT that has multiple streaming channels:

- The master channel ignores the `Ready` signal from downstream components.
- The slave channel `Ready` signal is high, which causes upstream components to continue sending data.

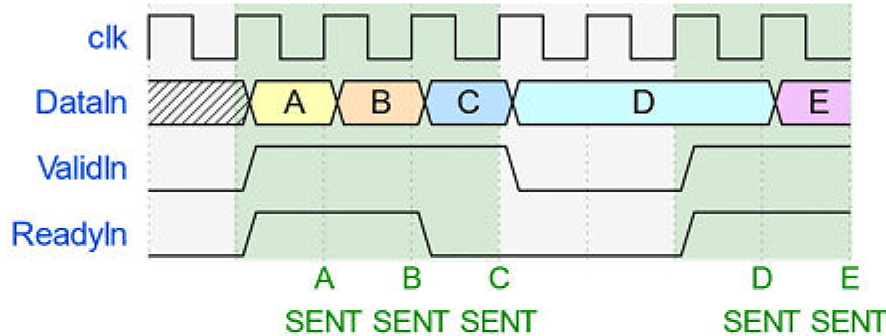
The absence of a back pressure logic might result in data being dropped. To avoid data loss and to apply back pressure on the slave interface or respond to back pressure from the master interface in your design:

- Model the Ready signal for each additional stream interface.
- Map the modeled Ready signal to a DUT port for the additional interface.

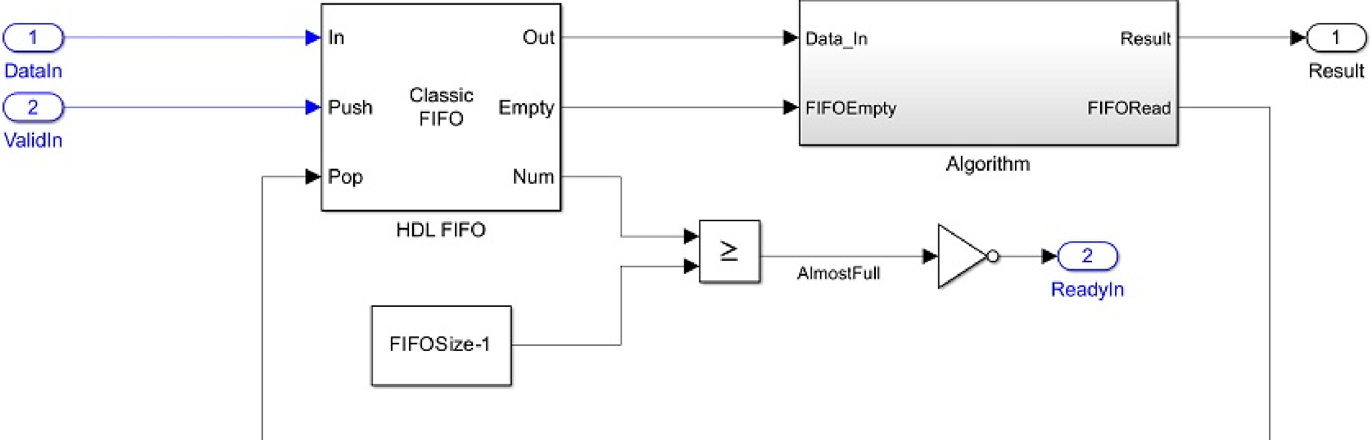
When you do not model the Ready signal, the **Set Target Interface** task displays a warning that provides names of interfaces that require a Ready port. If your design does not require applying or responding to back pressure, ignore this warning.

Simplified Streaming Protocol Input

This image illustrates the timing relationship between the `Data`, `Valid`, and `Ready` signals according to the simplified streaming protocol. In this image clock is `clk`. The AXI4-Stream Slave module sends the `DataIn` and `ValidIn` signals after asserting the `ReadyIn` signal from the DUT. This is represented by data packets A,B,D, and E in the image. When you drop the `ReadyIn` signal, the module always sends one more `DataIn` and `ValidIn` signal. This is represented by data packet C in the image. When you model the `ReadyIn` signal, the DUT must be able to accept one more value after de-asserting the ready signal.

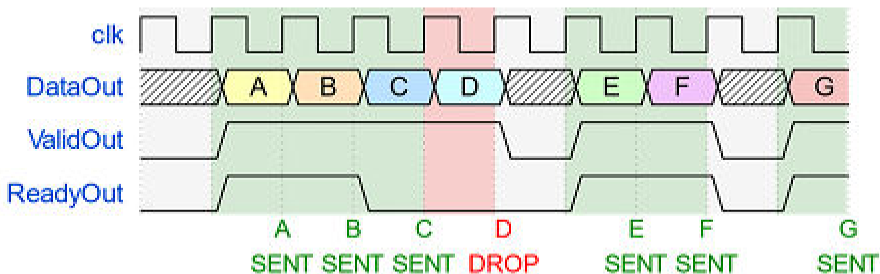


For example, if you have a first in first out (FIFO) in your DUT to store a frame of data, to apply backpressure to the upstream component, model the `Ready` signal based on the FIFO almost full signal.



Simplified Streaming Protocol Output

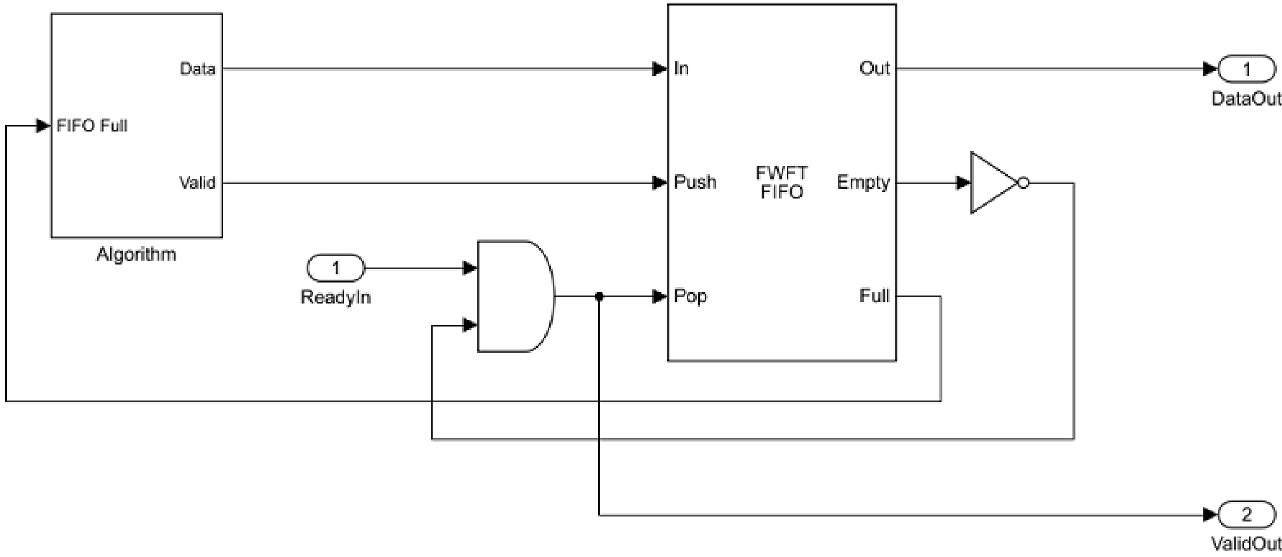
This image illustrates the timing relationship between the `Data`, `Valid`, and `Ready` signals according to the simplified streaming protocol. In this image clock is `clk`. You can send `DataOut` and `ValidOut` signals to the AXI4-Stream Master module after you assert the `ReadyOut` signal. This is represented by data packets A,B,E,F,and G in the image. You can optionally send one more `DataOut` and `ValidOut` signal after the `ReadyOut` signal drops. This is represented by data packet C in the image. You can only send one additional packet after the `ReadyOut` signal drops, subsequent data packets will be dropped until the `Ready` signal is asserted again. This is represented by data packet D in the image.



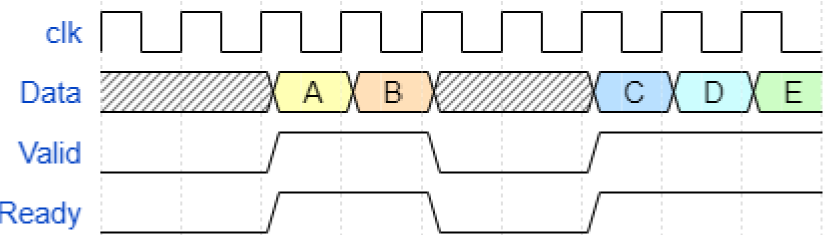
The optional one cycle latency between the `Valid` and `Ready` signals of the simplified streaming protocol allows you to use a classic or first word fall through (FWFT) FIFO to handle the backpressure from downstream components.

Downstream backpressure handling with FWFT FIFO

You can use a FWFT FIFO to store a frame of data and handle back pressure from downstream components by modeling the `ValidOut` signal as `ReadyOut` and FIFO not empty.

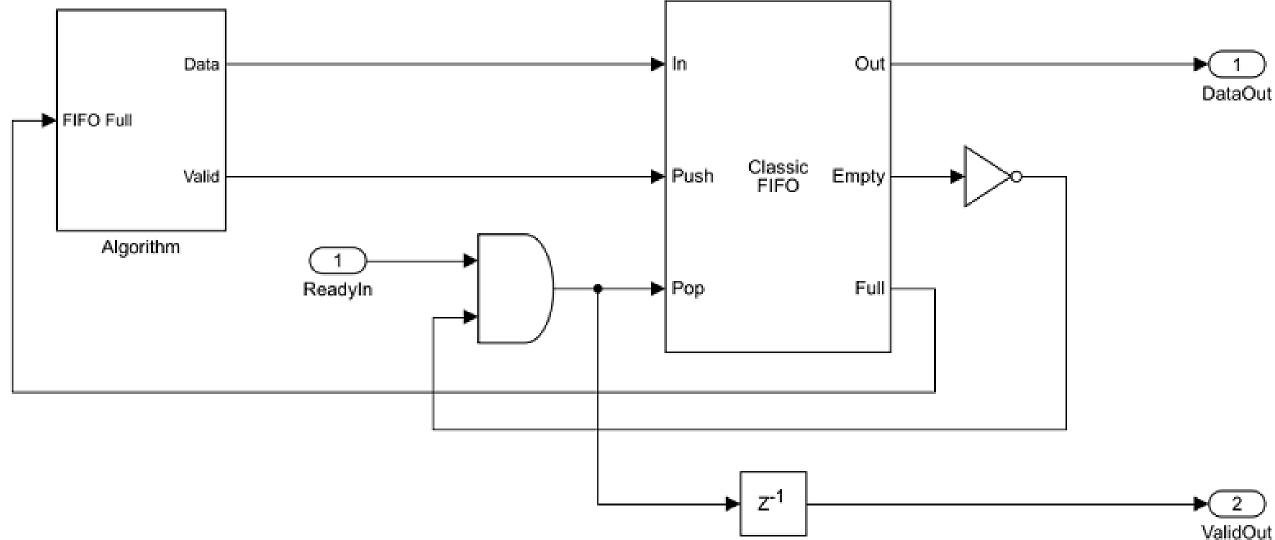


For a DUT using an FWFT FIFO there is zero latency for the `ReadyOut` signal between upstream and downstream components. This image shows the timing relationship between the `DataOut`, `ValidOut`, and `ReadyOut` signals. The clock signal is `clk`.

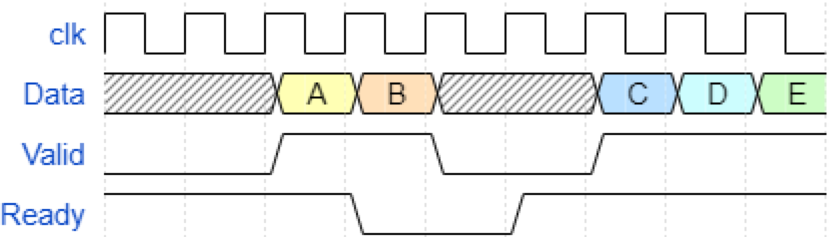


Downstream backpressure handling with Classic FIFO

If your DUT uses a classic FIFO to store a frame of data, model the `ValidOut` signal as `ReadyOut` and FIFO not empty.



This image shows the timing relationship between the `DataOut`, `ValidOut`, and `ReadyOut` signals. The clock signal is `clk`.



If you do not model the `Ready` signal, HDL Coder generates the signal and the associated back pressure logic. When you generate the IP core, HDL Coder adds a streaming interface module in the HDL IP core that translates the simplified protocol to the full AXI4-Stream protocol.

Note If you enable delay balancing, the coder inserts one or more delays on the Ready signal. Disable delay balancing for the Ready signal path.

TLAST Signal (optional)

The AXI4-Stream interface on your DUT can optionally model a TLAST signal, which is used to indicate the end of a frame of data. If you do not model this signal, HDL Coder generates it for you. On the AXI4-Stream Slave interface, the incoming TLAST signal is ignored. On the AXI4-Stream Master interface, the autogenerated TLAST signal is asserted when the number of valid samples counts up to the default frame length value. The default frame length value can be set by using the AXI4-Stream interface options in the Target Interface Table. See “Interface Options for AXI4-Stream Data” on page 40-26.

When the IP core has an AXI4 Slave interface, the default frame length value is stored in a programmable register in the IP core. You can change the default frame length during run time.

When the default frame length register is changed in the middle of a frame, the TLAST counter state is reset to zero and the TLAST signal is asserted early. You can find the address for the programmable TLAST register in the IP core generation report.

Frame-Based Modeling

You can design your DUT to operate on frames of data and map the data ports to a streaming interface. To map frame ports (vectors, matrices, and complex matrices) to an AXI4-Stream interface use the frame-to-sample optimization. For more information, see “Model Design for Frame-Based IP Core Generation” on page 40-29.

Legacy Frame-Based Modeling

Design your algorithm to operate on a stream of samples and model the data signal as a vector. To operate in this mode in the HDL Coder Workflow Advisor **Task 1.2.Set Target Interface > Interface Options** set the **Sample Packing Dimension** to `None`.

Note This modeling style will be deprecated in a future release.

Map Vector Ports to AXI4-Stream Interfaces

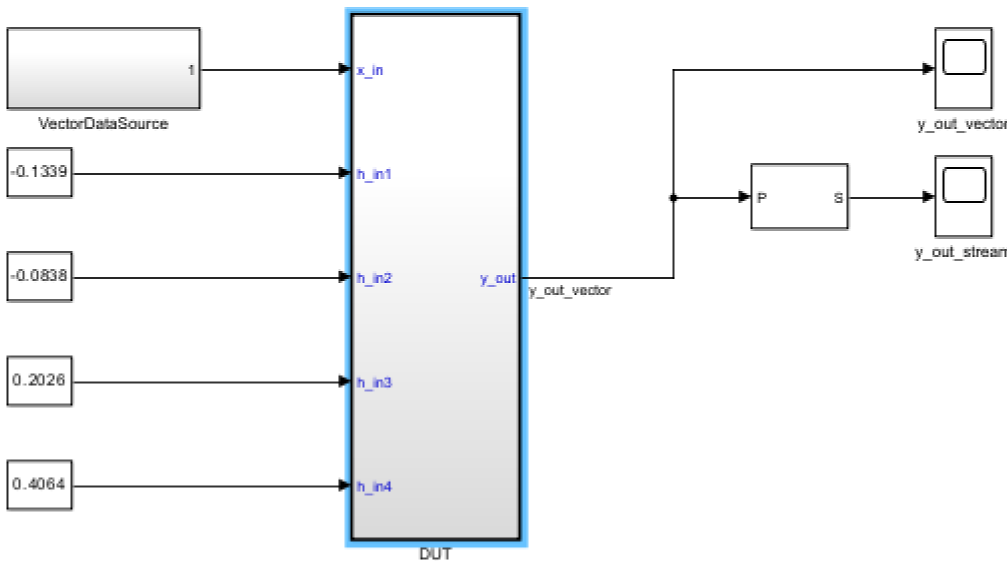
This example shows how to map vector data types to AXI4-Stream interfaces and generate an IP core. This example uses the legacy frame-based modeling where you design your algorithm to operate on a stream of samples and map the data ports to a streaming interface.

Open the Model

To open the model, enter:

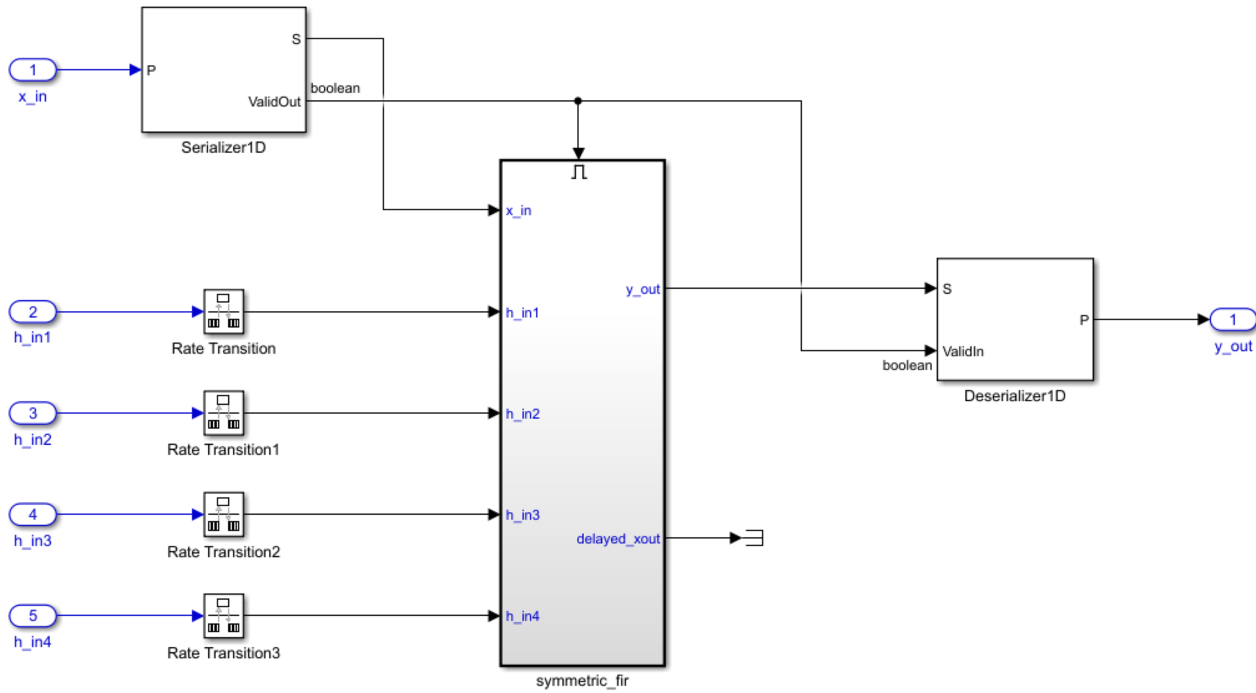
```
open_system('hdlcoder_sfir_fixed_vector');
```

The model consists of a top-level subsystem block name `DUT`, that accepts one vector input and three scalar inputs and returns a streaming vector output.



To map vector ports to AXI4-Stream interfaces:

- Connect each DUT input vector data port to a Serializer1D block. The serializer block must have a **ValidOut** port and the **Ratio** parameter must be set to the vector bit width.
- Connect each DUT output vector data port to a Deserializer1D block. The deserializer block must have a **ValidOut** port and the block **Ratio** parameter must be set to the vector bit width.
- Connect each scalar port that maps to an AXI4-Lite interface to a Rate Transition block.
- Each scalar port that maps to an external port must have the same sample time as the streaming algorithm subsystem.

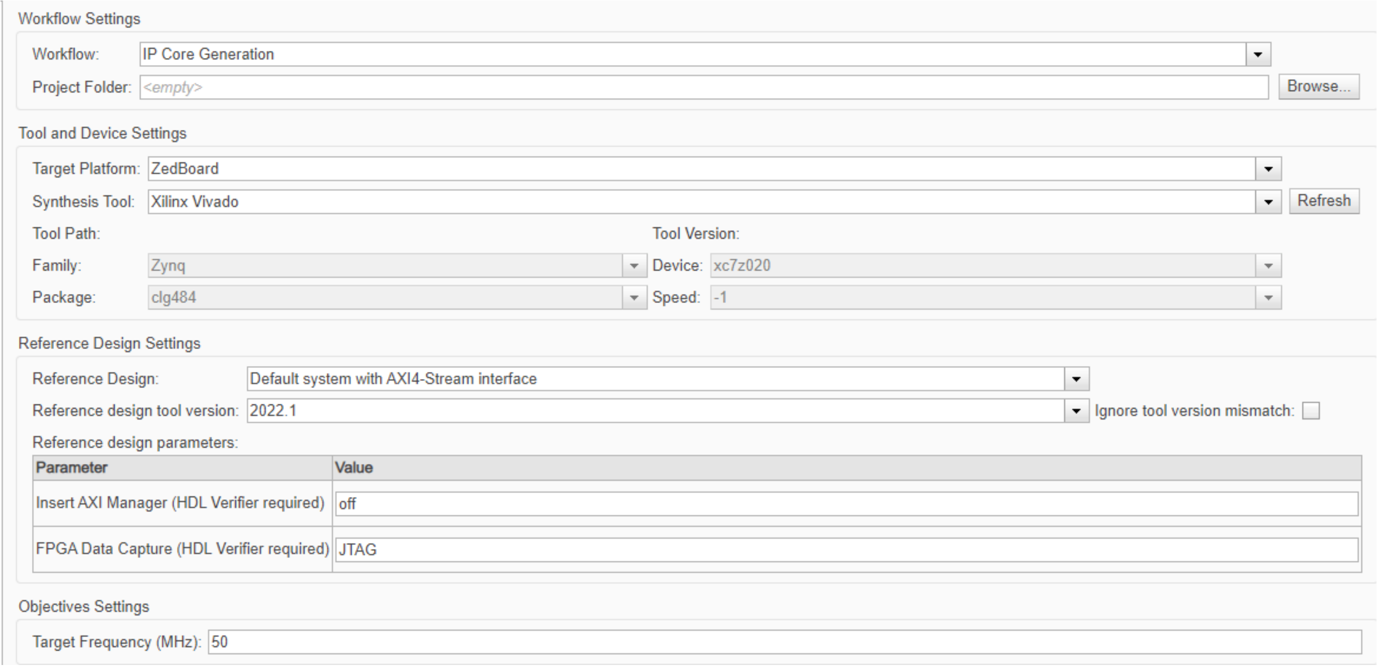


Generate HDL IP Core Using Simulink Interface

To generate an IP core using the Simulink interface, prepare your model by using the configuration parameters, configure your design by using the IP Core editor, and generate the IP core by using the Simulink toolstrip.

Prepare Model for IP Core Generation

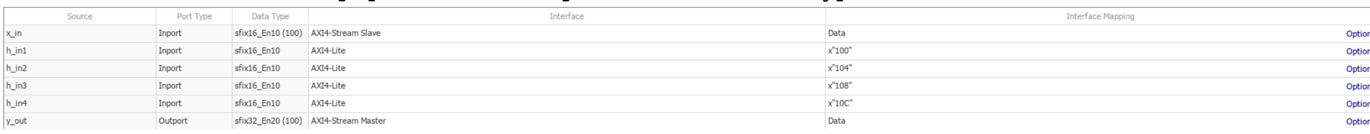
1. In the Apps gallery, click **HDL Coder**. In the **HDL Code** tab, in the **Output** section, set the drop-down button to **IP Core**. Alternatively, in the **HDL Code** tab, click **Settings** to open the Configuration Parameters dialog box. In the **HDL Code Generation > Target** tab, set the **Workflow** parameter to `IP Core Generation`.
2. In the Configuration Parameters dialog box, set the **Target Platform to** `ZedBoard` and **Synthesis Tool** to `Xilinx Vivado`. Optionally, you can use the **Project Folder** parameter to specify a top-level folder for any generated folders and files. If **Project Folder** is empty, HDL Coder saves the generated files in the current directory.
3. In the Configuration Parameters dialog box, set **Reference Design** to `Default system with AXI4-Stream interface`.
4. Click **Apply** to save your updated settings.



Configure Design

Configure your design to map to the target hardware by mapping the DUT ports to IP core target hardware and setting DUT-level IP core options. In this example, the input port **x_in** is mapped to an AXI4-Stream Slave interface and input ports **h_in1**, **h_in2**, **h_in3**, and **h_in4** are mapped to the AXI4-Lite interface, so HDL Coder generates AXI interface accessible registers for them.

1. In Simulink, in the **HDL Code** tab, click **Target Interface** to open the IP Core editor.
2. Select the **Interface Mapping** tab to map each DUT port to one of the IP core target interfaces. The generated design can then communicate with the rest of the hardware system when it is deployed. If no mapping table appears, click the Reload IP core settings button to compile the model and repopulate the DUT ports and their data types.



Source	Port Type	Data Type	Interface	Interface Mapping	
x_in	Inport	sfix16_En10 (100)	AXI4-Stream Slave	Data	Options
h_in1	Inport	sfix16_En10	AXI4-Lite	x"100"	Options
h_in2	Inport	sfix16_En10	AXI4-Lite	x"104"	Options
h_in3	Inport	sfix16_En10	AXI4-Lite	x"108"	Options
h_in4	Inport	sfix16_En10	AXI4-Lite	x"10C"	Options
y_out	Outport	sfix32_En20 (100)	AXI4-Stream Master	Data	Options

3. Validate your settings by clicking the Validate IP core settings button .

In the IP Core editor, you can adjust the IP core settings and configure the interface mappings for your target hardware:

- Use the **General** tab to configure top-level IP core settings such as the name of the IP core and whether to generate an IP core report.
- Use the **Clock Settings** tab to configure clock-related settings for the IP core.

.Generate IP Core

After you configure the IP core settings and mappings for your design, you can generate an IP core. In the Simulink Toolstrip, in the **HDL Code** tab, click **Generate IP Core**. After you generate the

custom IP core, the IP core files are in the `ipcore` folder in your current directory. To specify a top-level project folder for the `ipcore` folder to be stored along with all other generated files, in the Configuration Parameters dialog box, use the **Project Folder** parameter in the **HDL Code Generation > Target** tab. If the **Project Folder** parameter is empty, HDL Coder saves the generated files in the current directory.

Generating an IP core also generates the code generation report. In the Code Generation Report window, in the left pane, click the **IP Core Generation Report**. The report describes the behavior and contents of the generated custom IP core.

Model Designs with Multiple Streaming Channels

When you run the `IP Core Generation` workflow, you can map multiple scalar DUT ports to AXI4-Stream Master and AXI4-Stream Slave channels. In legacy frame-based mode, you can use at most one AXI4-Stream Master channel and one AXI4-Stream Slave channel.

Note In the sample-based mode when you use multiple streaming channels, HDL Coder generates the Ready signal but does not generate the back pressure logic. If you want your design to handle back pressure, model the Ready signal in your design.

To learn more, see “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 40-33.

When you model your DUT using the frame-based mode you can map multiple frame DUT ports to AXI4-Stream Master and AXI4-Stream Slave channels. When you use the frame-based mode HDL Coder generates the ready and valid signals for all the streaming ports.

Model Designs That Have Multiple Sample Rates

When you run the IP Core Generation workflow, use the HDL Coder software for designs that have multiple sample rates. When you map the interface ports to AXI4-Stream Master or AXI4-Stream Slave interfaces, to use multiple sample rates, map the DUT ports that map to these AXI4 interfaces to run at the fastest rate of the design or at rates slower than the design rate.

HDL Coder runs the DUT ports mapped to AXI4-Stream master and slave interfaces at rates slower than the model design rate by:

- Setting the AXI4-Stream master channel valid signal to high at the first cycle every N clock cycles. For example, if the design rate is eight times faster than the slow rate DUT ports, the valid signal is high for the first clock cycle every eight clock cycles.
- Asserting back pressure on the AXI4-Stream slave interface to make sure that the incoming data is streamed at the rate of one data frame every N clock cycles. For example, if the design rate is eight times faster than the slow rate DUT ports, the first frame is streamed at clock cycle one, the second frame at clock cycle nine, and so on.

When you map the AXI4-Stream Interface DUT port to the fastest rate in the design, the valid signal is high, making sure there is no back pressure on the AXI4-Stream slave interface.

When designing models that have multiple sample rates, for each AXI4-Stream interface, AXI4-Stream interface signals, such as data signals, valid signals and all the optional signals, need to be mapped at same rate.

To learn more, see “Multirate IP Core Generation” on page 40-85.

Interface Options for AXI4-Stream Data

When you run the `IP Core Generation` workflow on a model that has vector data, you can specify how the vector data is treated as a sample or as a frame by using the **Sample Packing Dimension**. When the vector data is treated as a sample, you can specify how the vector elements are packed together by using the **Packing mode** option.

When you run the `IP Core Generation` workflow on a model that has an AXI4- Stream interface with none of the signals mapped to `TLAST`, you can specify the `TLAST` register value by using the **DefaultFrameLength** option.

Default Frame Length

When you do not model the TLAST signal, specify the default frame length (TLAST) value for the AXI4 Stream Master interface. The TLAST signal is created for you In the generated IP core and the signal is asserted when the number of valid samples counts up to the value in the default frame length counter. When the generated IP core has an AXI4 Slave interface, HDL Coder generates the default frame length as a programmable register. When the default frame length register is changed in the middle of a frame, the TLAST counter state is reset to zero and the TLAST signal is asserted early. For more information, see “TLAST Signal (optional)” on page 40-20. When you do not select the **Generate default AXI4 slave interface** the default frame length is generated as a constant value and not as a programmable register.

Sample Packing Dimension

Specify if the vector data is treated as a sample or as a frame:

- None. This value is the default value. When you specify None, vectors are treated as frames and vector elements are streamed one after the other. For example, when the input is a six-by-one vector in the first clock cycle, the first vector element is streamed, the second vector element, in the second clock cycle, and so on. To use this mode, the model must contain a Serializer block for inputs and a Deserializer block for the outputs. The **Packing mode** is not available when the **Sample Packing Dimension** is set to None.
- All. When you specify All, the vectors are packed together and streamed in a single clock cycle. For example, when the input is a six-by-one vector, all vector elements are packed together and streamed in a single clock cycle. In this case, you can specify how the vector elements are packed by using the **Packing mode** option.

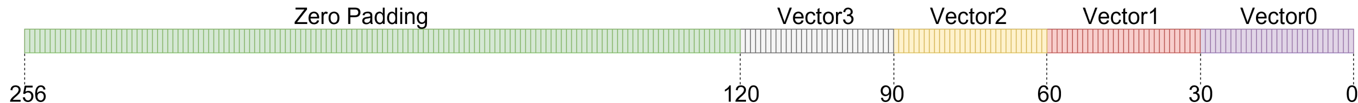
Packing Mode

When you set **Sample Packing Dimension** to `All`, you can specify how HDL Coder packs vector elements, complex data, and complex vectors by specifying the **Packing Mode** parameter to either `Bit Aligned` or `Power of 2 Aligned`. This setting applies to the AXI4-Stream master and slave channels.

Non-Complex Vector Data Packing

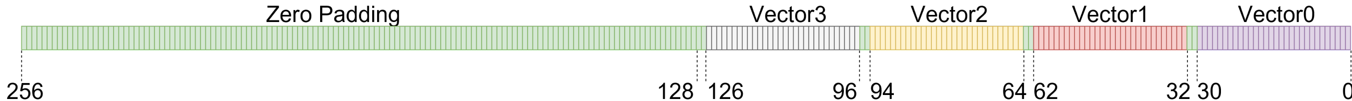
- `Bit Aligned`. When you set the **Packing Mode** to this setting, HDL Coder packs the vector elements directly next to each other. If the packed bit width is less than the AXI4-Stream channel width, then pad the packed data with zeros to match the channel width.

For example, if the AXI4-Stream channel width is 256 bits, there are four vector elements, and the vector elements are 30 bits long, the total data width is 120 bits. When the packing mode is set to `Bit Aligned`, HDL Coder packs the AXI4-Stream data as shown in this diagram.



- `Power of 2 Aligned`. In this mode, the vector elements are first padded with zeros to the closest power of two boundary. Then, the padded elements are packed together. If the packed vector bit width is less than the AXI4-Stream channel width, then the packed data is padded with zeros to match the channel width.

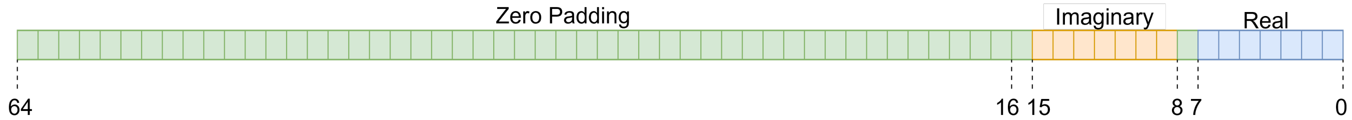
  For example, if the AXI4-Stream channel width is 256 bits, there are four vector elements, and the vector elements are 30 bits long, the total data width is 120 bits. When the packing mode is set to `Bit Aligned`, HDL Coder packs the AXI4-Stream data as shown in this diagram. When the packing mode is set to `Power of 2 Aligned`, HDL Coder packs the AXI4-Stream data as shown in this diagram.



  Each vector element of bit width 30 is padded with zeroes of bit width two to extend the vector element width to 32, the nearest power of two boundary.

Complex Data Packing

When the input data is a complex number, HDL Coder pads the real and imaginary parts with zeroes to the closest power of two boundary. HDL Coder then packs the data together and pads the data with zeroes to match the channel width. For example, if the AXI4-Stream channel width is 64-bits and the complex data type is `ufix7`, HDL Coder packs the stream data as shown in this image:



HDL Coder pads the complex data with zeroes of bit width one to extend the real and imaginary data packets to a size eight, and then pads the data with zeroes of width 48 to extend the complex data frame size to 64, the AXI4-Stream channel width.

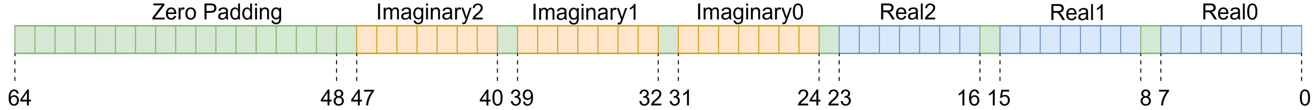
When you use the generated software interface model to stream complex data to the target board, you must pack the complex data as shown in the example images.

Complex Vector Data Packing

To model complex data signals as frames on the DUT boundary follow the frame-based modeling requirements. In this case, HDL Coder packs the complex data elements of the frame as described in the Complex Data Packing section.

Alternatively, if you want to treat the entire complex vector as a sample, follow the guidelines in the Sample Based Modeling section. HDL Coder sets the **Sample Packing Dimension** parameter to `All` and the **Packing Mode** parameter to `Power of 2 Aligned`.

For example, if the AXI4-Stream channel width is 64-bits, the complex data type is`ufix7`, and the complex vector has three elements, HDL Coder packs the data as shown in this image:



HDL Coder pads the complex data with:

- Zeroes of bit width one after every real and imaginary vector element to extend the data size to eight, the nearest power of two boundary.
- Zeroes of bit width 16 at the end to extend the frame size to 64, the width of the AXI4-Stream channel.

When you use the generated software interface model to stream complex data to the target board, you must pack the complex data in the data packing format as shown in the example images.

Restrictions

When you map scalar or vector DUT ports to AXI4-Stream interfaces:

- Xilinx Zynq-7000 or Intel Quartus Prime must be your target platform.
- Xilinx Vivado or Intel Quartus Prime must be your synthesis tool.
- **Processor/FPGA synchronization** must be Free Running.

See Also

Related Examples

- “Deploy Model with AXI-Stream Interface in Zynq Workflow” on page 40-192
- “Offload Large Delays from Frame-Based Models to External Memory” on page 22-32
- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270
- “Deploy a Frame-Based Model with AXI4-Stream Interfaces” on page 40-378

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 39-33

Model Design for Frame-Based IP Core Generation

In this section...

"Frame-Based Modeling for AXI4-Stream Interfaces" on page 40-29
"Frame-Based Modeling for AXI4-Stream Video Interfaces" on page 40-30
"Enable the Optimization" on page 40-31
"Modeling Requirements" on page 40-32

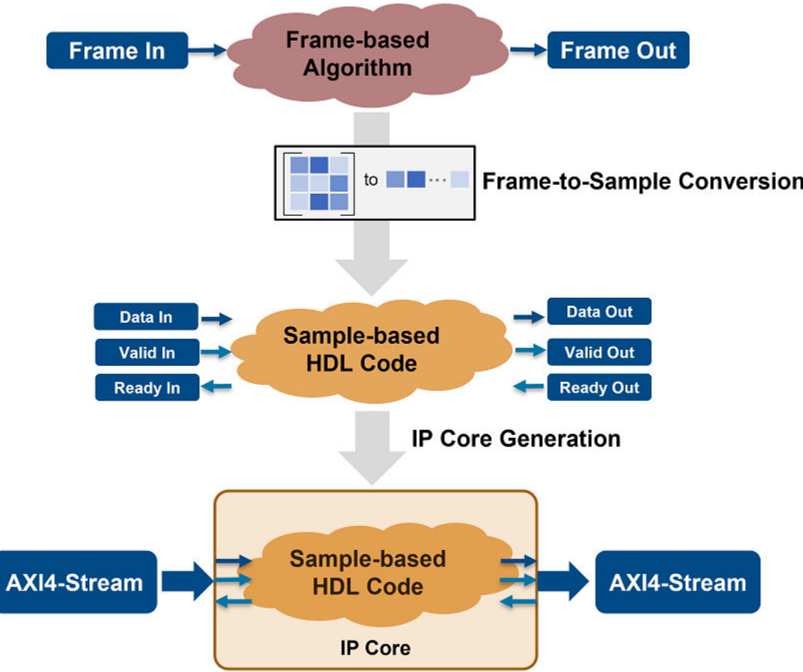
You can use the frame-to-sample optimization to generate IP cores for frame-based models. The frame-to-sample optimization maps matrices and vectors to the AXI4-Stream ports and matrices to AXI4-Stream Video interfaces, and then creates the necessary logic to handle the streamed data in the frame-based design.

Frame-Based Modeling for AXI4-Stream Interfaces

If your model includes streamed data, but does not process videos, you can use the frame-to-sample optimization to generate an IP core that operates on frames of data, then map the data ports to the streaming interface. You can then translate and implement your frame-based models on pixel-based hardware. See "HDL Code Generation from Frame-Based Algorithms" on page 22-2.

When you generate the IP core, you map the frame data ports at the DUT boundary to AXI4-Stream interfaces. HDL Coder generates the Valid and Ready signals for each port. This image shows a top-level overview of the frame-to-sample optimization and IP core generation.

Map vectors, complex vectors, matrices, and complex matrix data to AXI4-Stream interfaces by using frame-to-sample conversion optimization. The TLAST signal is created in the generated IP core and the signal is asserted when the number of valid samples counts to the frame size of the data port.



For an example on modeling a frame-based model with AXI4-Stream interfaces, see “Deploy a Frame-Based Model with AXI4-Stream Interfaces” on page 40-378.

To model your algorithm using a simplified streaming protocol and a sample-based DUT, see “Model Design for AXI4-Stream Interface Generation” on page 40-14.

Frame-Based Modeling for AXI4-Stream Video Interfaces

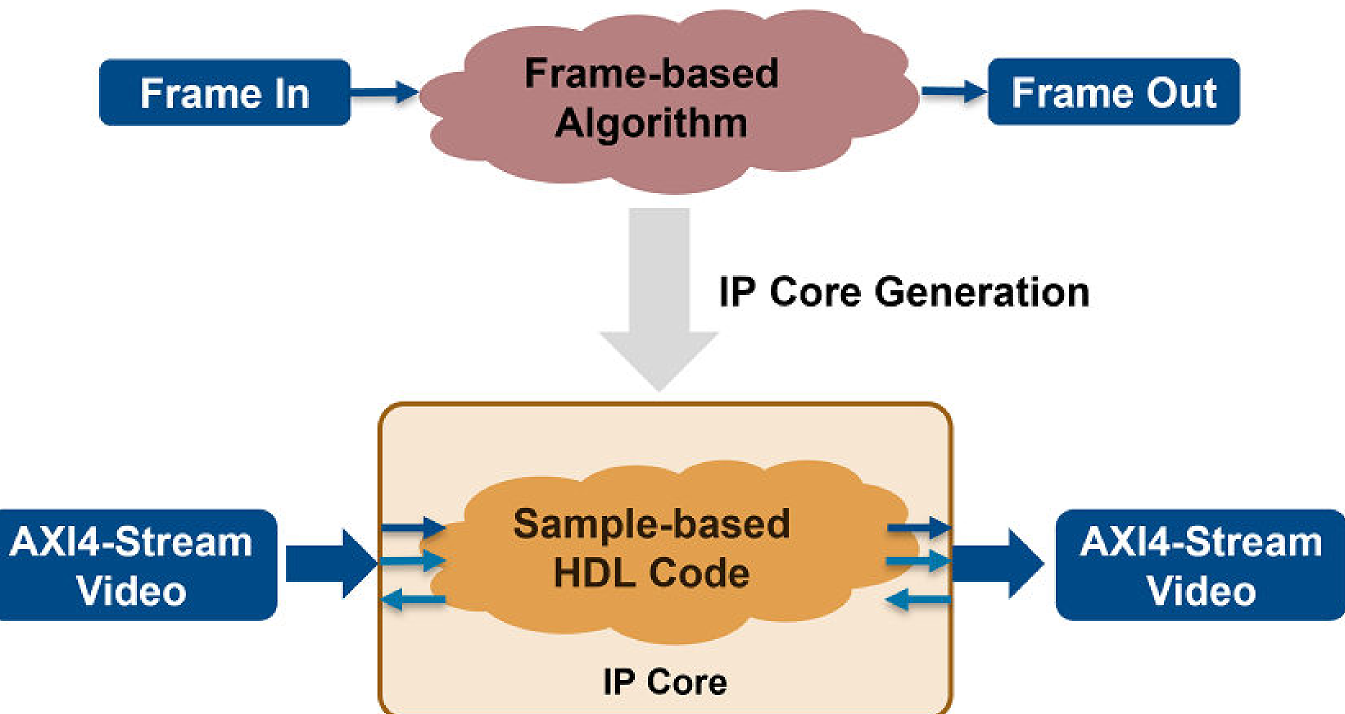
If your model includes streamed video ports, you can use the frame-to-sample optimization to map the two-dimensional matrix ports of your DUT to an AXI4-Stream Video interface. You can then prototype your algorithm in Simulink using frame-based modeling and test the functionality on live video inputs and outputs.

When you use the frame-to-sample optimization, HDL Coder:

- Converts the interface and generates the Data, Valid, and Ready signals for each port.
- Inserts the video porch and handles the start of frame (SOF) signal.

This image shows a top-level overview of the frame-to-sample optimization and IP core generation.

Map matrix ports to an AXI4-Stream Video interface by using the frame-to-sample optimization. The TUSER and TLAST signals are created during IP core generation. TUSER is asserted at the start of every frame and TLAST is asserted at the end of each line.



Video Porch Insertion Logic

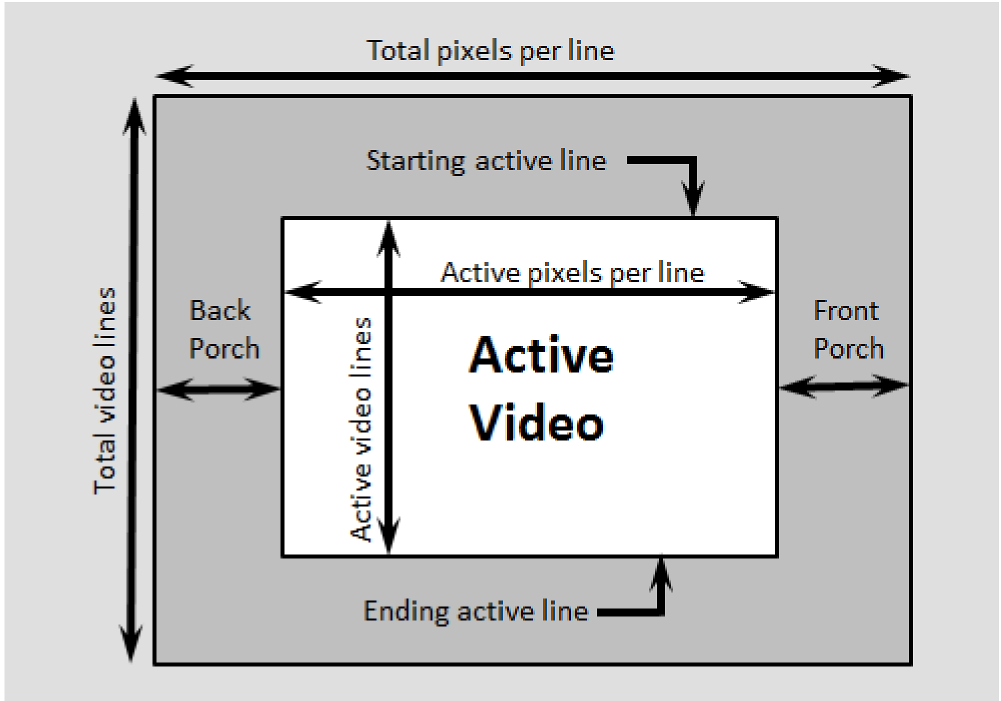
Video capture systems scan video signals from left to right and from top to bottom. As these systems scan, they generate inactive intervals between lines and frames of active video. This inactive interval is called a video porch. When you generate an IP core using frame-to-sample conversion, the active

pixels per line and the active lines in each frame are defined by the frame size of the frame-based model and they are not configurable at runtime. However, you can configure the horizontal and vertical porch. HDL Coder inserts vertical and horizontal porch to the pixel stream based on the AXI4-Lite registers in the generated IP core. You can customize these porch parameters for each video frame:

- Horizontal porch length (Default: 280)
- Vertical porch length (Default: 45)

The default values correspond to the porch values of a 1080p video frame.

This figure shows a video frame with the horizontal porch split into a front and a back porch.



To model your algorithm using a sample-based DUT and the streaming pixel protocol, see “Model Design for AXI4-Stream Video Interface Generation” on page 40-118.

For an example on modeling a frame-based model with AXI4-Stream Video interfaces, see “Deploy Frame-Based Models with AXI4-Stream Video Interfaces in Zynq-Based Hardware” on page 40-386.

Enable the Optimization

To use the frame-to-sample optimization:

- Enable the frame-to-sample optimization. See, Enable frame to sample conversion.
- Enable frame-to-sample optimization on the input data signal on the DUT that maps to the AXI-4 Stream or AXI4-Stream Video interfaces.

Modeling Requirements

- Model only the Data signals. The frame-to-sample conversion optimization generates the Ready and Valid signals.
- Set the `SamplesPerCycle` parameter of the frame-to-sample optimization to 1. See Samples per cycle.

See Also

More About

- “Model Design for AXI4-Stream Interface Generation” on page 40-14
- “Streaming Pixel Interface” (Vision HDL Toolbox)
- “Model Design for AXI4-Stream Interface Generation” on page 40-14
- “HDL Code Generation from Frame-Based Algorithms” on page 22-2

See Also

Related Examples

- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270
- “Deploy a Frame-Based Model with AXI4-Stream Interfaces” on page 40-378
- “Deploy Frame-Based Models with AXI4-Stream Video Interfaces in Zynq-Based Hardware” on page 40-386

Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces

When you run the generic IP Core Generation workflow for your Simulink model or target your own custom reference design that you authored, you can generate an HDL IP core with multiple AXI4-Stream interfaces, AXI4-Stream Video interfaces, or AXI4 Master interfaces. To learn about these interfaces, see “Target Platform Interfaces” on page 39-17.

Why Use Multiple AXI4 Interfaces

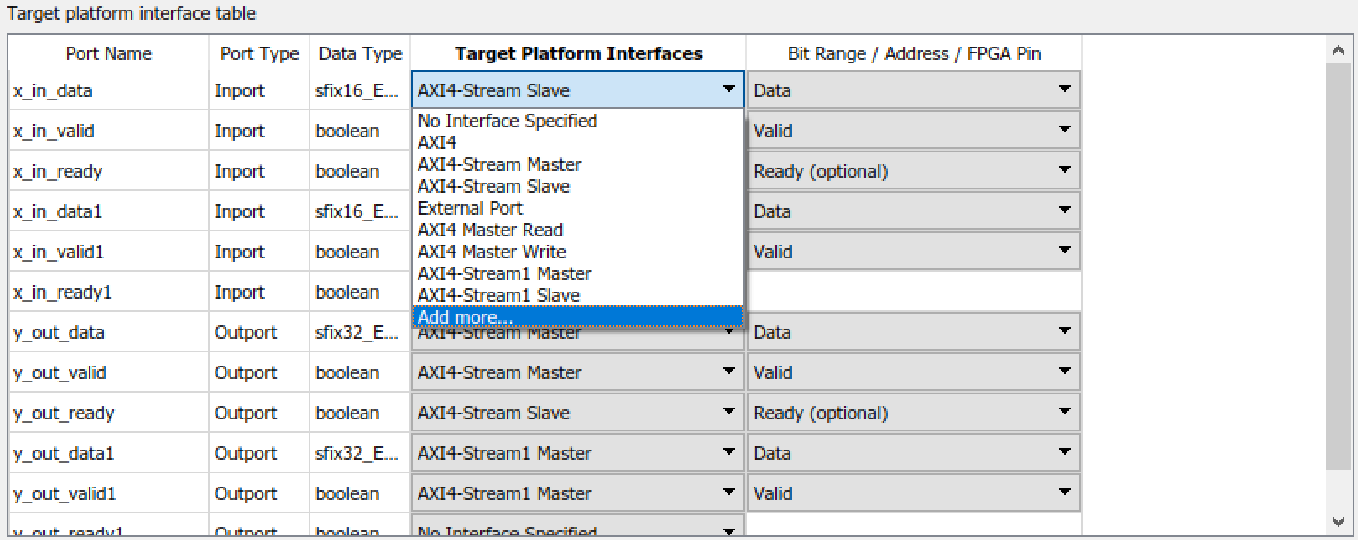
You can use multiple streaming interfaces to facilitate high-speed data transfer in various applications such as:

- Transferring data between A/D and D/A converters
- Software-defined radio algorithms that process multiple transceiver channels
- Vision algorithms that perform image annotation or object detection

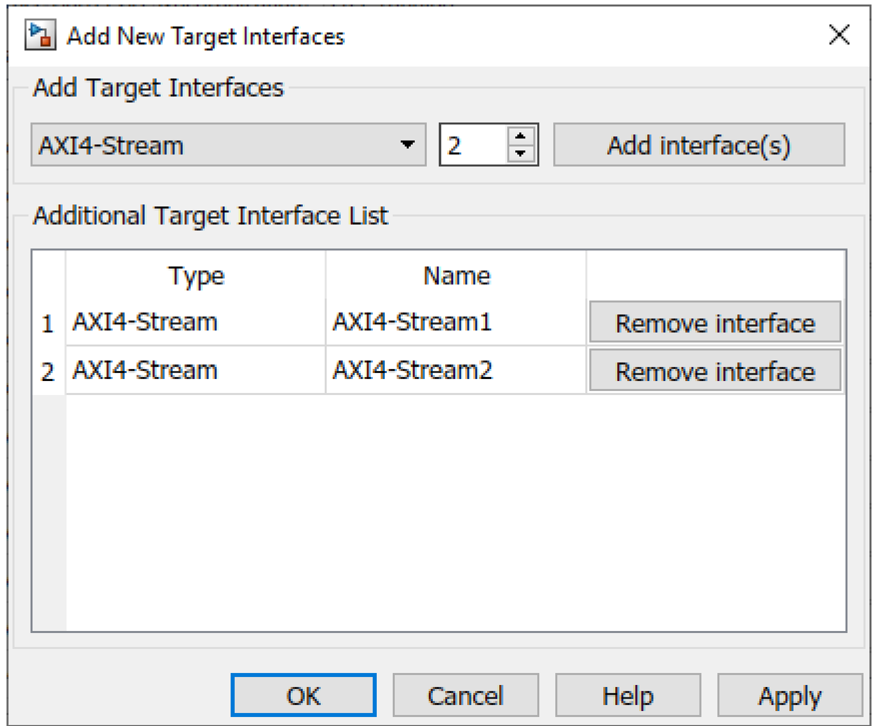
Specify Multiple AXI4 Interfaces in Generic IP Core Generation Workflow

To specify more than one AXI4-Stream, AXI4-Stream Video, or AXI4 Master channel:

1. In the **Set Target Device and Synthesis Tool** task, select `IP Core Generation` as the **Target workflow** and `Generic Xilinx Platform` or `Generic Altera Platform` as the **Target platform**. Run this task.
2. To add multiple target interfaces, in the **Set Target Interface** task, on the **Target Platform Interfaces** section of the Target platform interface table, select **Add more ...**.



3. You can then add more interfaces in the Add New Target Interfaces dialog box. Specify the type of interface you want to add, the number of interfaces, and a custom name for each additional interface.



After you apply the settings, the interfaces you created appear in the Target platform interface table. After you run this task, the additional interfaces specified are saved on the DUT subsystem as the HDL block property **AdditionalTargetInterfaces**.

If you modify the additional interfaces that were already mapped to DUT ports such as deleting or renaming an interface that was already mapped, the previous interface mapping information might be lost. The ports then become unmapped to interfaces and the **Target platform interfaces** section displays `No interface specified`. Therefore, if you make changes to the additional target interfaces, verify that the DUT ports are mapped to the correct target interfaces.

Specify Multiple AXI4 Interfaces in Custom Reference Designs

When you create your own custom reference design, you can add multiple AXI4-Stream, AXI4-Stream Video, and AXI4 Master interfaces. Depending on the interface type you want to add, specify additional interfaces by using the `addAXI4StreamInterface`, `addAXI4StreamVideoInterface`, or `addAXI4MasterInterface` methods of the `hdlcoder.ReferenceDesign` class.

To add more interfaces, in the `plugin_rd` file, call the interface method each time you want to add more interfaces. This example shows how to add two AXI4-Stream interfaces.

```
function hRD = plugin_rd()
% Reference design definition

%   Copyright 2017-2019 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Multiple Interface Reference Design';
hRD.BoardName = 'ZedBoard';
```

```
% Tool information
hRD.SupportedToolVersion = {'2019.1'};

% ...
% ...

% Add AXI4-Stream interface 1
hRD.addAXI4StreamInterface (...
    'MasterChannelEnable',      true, ...
    'SlaveChannelEnable',       true, ...
    'MasterChannelConnection',  'axi_dma_s2mm/S_AXIS_S2MM', ...
    'SlaveChannelConnection',   'axi_dma_mm2s/M_AXIS_MM2S', ...
    'MasterChannelDataWidth',   32, ...
    'SlaveChannelDataWidth',    32, ...
    'InterfaceID',              'AXI4-Stream1');

% Add AXI4-Stream interface 2
hRD.addAXI4StreamInterface (...
    'MasterChannelEnable',      true, ...
    'SlaveChannelEnable',       true, ...
    'MasterChannelConnection',  'ADC/S_AXIS_S2MM', ...
    'SlaveChannelConnection',   'DAC/M_AXIS_MM2S', ...
    'MasterChannelDataWidth',   32, ...
    'SlaveChannelDataWidth',    32, ...
    'InterfaceID',              'AXI4-Stream2');



% ...
% ...
```

When you run the `IP Core Generation` workflow and target the custom reference design `Multiple Interface Reference Design`, in the **Set Target Interface** task, you can map the DUT ports to AXI4-Stream1 Master and Slave channels and AXI4-Stream2 Master and Slave channels.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
x_in_data	Inport	sfix16_E...	AXI4-Stream1 Slave	Data
x_in_valid	Inport	boolean	AXI4-Stream1 Slave	Valid
x_in_ready	Inport	boolean	AXI4-Stream1 Master	Ready (optional)
h_in1	Inport	sfix16_E...	AXI4-Lite	x"100"
h_in2	Inport	sfix16_E...	AXI4-Lite	x"104"
h_in3	Inport	sfix16_E...	AXI4-Lite	x"108"
h_in4	Inport	sfix16_E...	AXI4-Lite	x"10C"
x_in_data1	Inport	sfix16_E...	AXI4-Stream2 Slave	Data
x_in_valid1	Inport	boolean	AXI4-Stream2 Slave	Valid
x_in_ready1	Inport	boolean	AXI4-Stream2 Master	Ready (optional)
h_in5	Inport	sfix16_E...	AXI4-Lite	x"110"
h_in6	Inport	sfix16_E...	AXI4-Lite	x"114"

Note When you target your own custom reference design and map the additional interfaces to DUT ports in the **Set Target Interfaces** task, the additional interfaces are not saved on the model as the **AdditionalTargetInterfaces** HDL block property. Instead, the additional interfaces are saved on the custom reference design in the `plugin_rd.m` file.

You can also dynamically customize the reference design to specify the number of interfaces you want to add and the interface properties.

1. In the `plugin_rd` file, create a reference design parameter for the number of additional interfaces you want to add.
2. Create a callback function that has different choices for the number of interfaces you want to add and then reference the function in the `plugin_rd` file by using the `CustomizeReferenceDesignFcn` method of the `hdlcoder.ReferenceDesign` class.

To learn more, see "Customize Reference Design Dynamically Based on Reference Design Parameters" on page 40-104.

Ready Signal Mapping for Multiple Streaming Interfaces

When you use a single streaming channel, HDL Coder automatically generates the Ready signal and the associated back pressure logic.

If you use multiple streaming channels, HDL Coder does not automatically generate the back pressure logic. In this case, the Ready signal is generated but the master Ready signal at the input is ignored and the slave Ready signal at the output is tied to high value. The absence of a back pressure logic can result in samples being dropped. If you want your design to apply back pressure on the Slave interface or respond to back pressure from the Master interface, you must model the Ready signal for each additional interface and then map the port to the Ready signal for that interface. When you do not model, the **Set Target Interface** task displays a warning that provides names of interfaces that require a Ready port. If your design does not need to apply or respond to back pressure, you can ignore this warning and you do not have to model the Ready signal.

When using multiple AXI4-Stream interfaces, if you want your design to apply back pressure on the Slave interface or respond to back pressure from the Master interface, you must model the Ready signal for each additional interface and then map the port to the Ready signal for that interface. To learn how the back pressure logic is generated for a single streaming channel and how to model the Ready signal, see "Ready Signal (Optional)" on page 40-16.

Restrictions

- When you run the generic IP Core Generation workflow, you can specify the interface type and a custom interface ID for each additional interface. Other interface properties such as the data width cannot be customized and use default values. When you create your own custom reference design, you can customize the interface name and interface properties.
- **Processor/FPGA synchronization** must be `Free running`.

See Also

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 39-33

See Also

Related Examples

- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270
- “Inspect the Written Values of AXI4 Slave Registers by Using the Readback Methods” on page 40-51

Running Audio Filter with Multiple AXI4-Stream Channels on ZedBoard

This example shows how to model an audio system with multiple AXI4-Stream channels and deploy it on a ZedBoard™ by using an audio reference design.

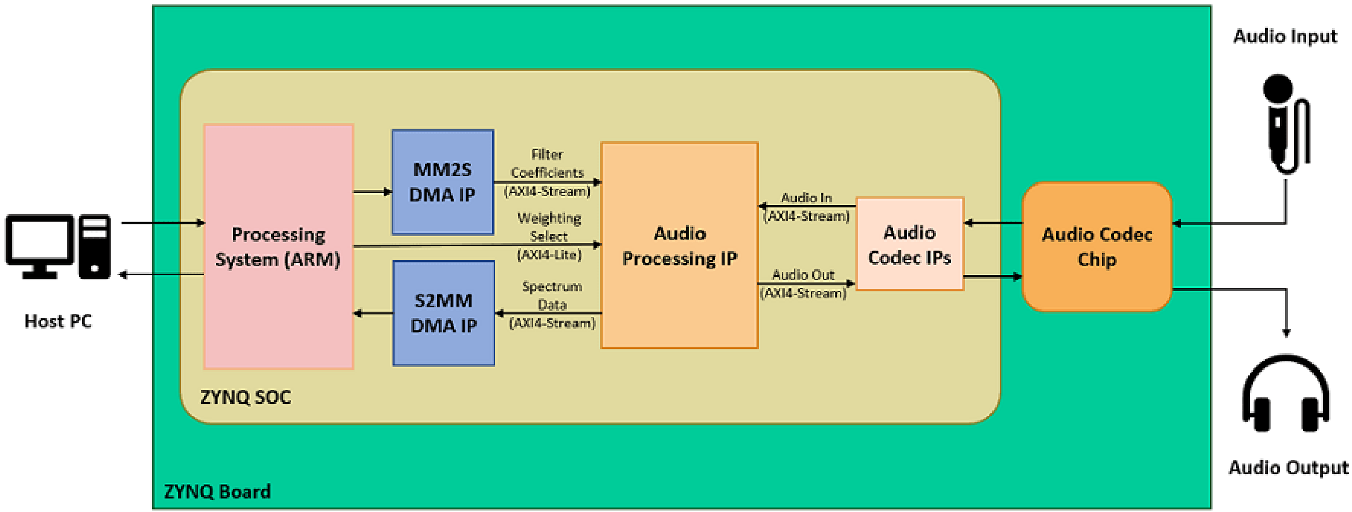
Introduction

In this example, you model a programmable audio filter with spectrogram using multiple AXI4-Stream channels and advanced AXI4-Stream signals `Ready` and `TLAST`. One AXI4-Stream channel transfers data between the filter and the audio codec. The other AXI4-Stream channel interfaces with the Processing System to program filter coefficients and transmit spectrogram data to the host computer for analysis.

You can then run the `IP Core Generation` workflow to generate an HDL IP core and deploy the algorithm on a ZedBoard by using an audio reference design.

System Architecture

This figure shows the high-level architecture of the system.



The **Audio Codec IPs** configure the audio codec and transfer audio data between the ZedBoard and audio codec. The Audio Processing IP generated by HDL Coder™ performs filtering and spectrum analysis. The DMA IPs transfer AXI4-Stream data between the Processing System and the FPGA. The stream data transmitted from the Processing System through the **MM2S DMA IP** programs the filter coefficients on the FPGA. The stream data received by the Processing System through the **S2MM DMA IP** contains the spectrogram data computed on the FPGA. The Processing System also configures the weighting curve for spectrum analysis using an AXI4-Lite interface.

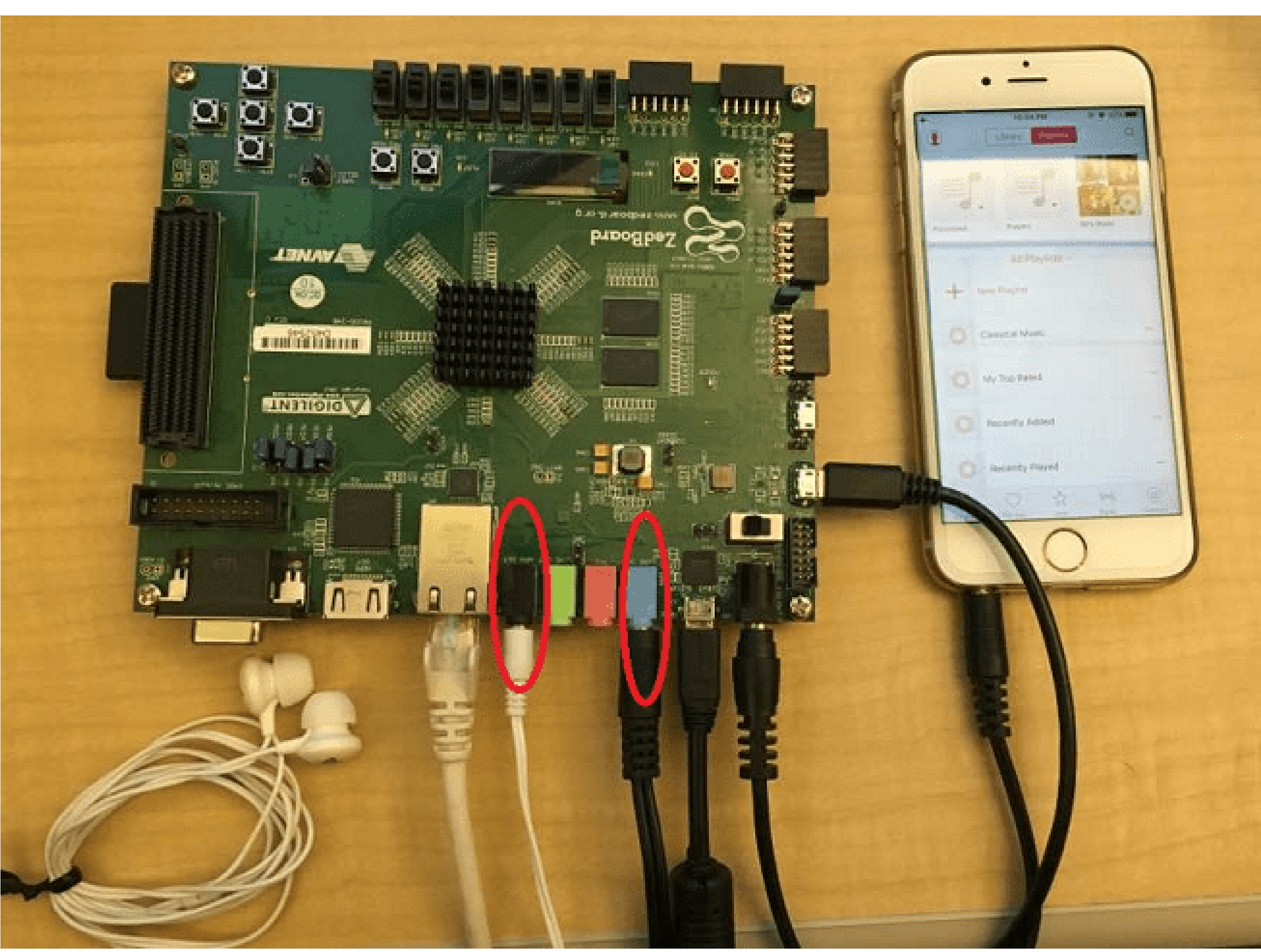
Prerequisites

This example extends the audio filter on live input example to use multiple streaming channels. To learn about the example that uses a single streaming channel, see “Running an Audio Filter on Live Audio Input Using a Zynq Board” on page 40-181.

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder Support Package for Xilinx® FPGA and SoC Devices
- Embedded Coder® Support Package for Xilinx Zynq Platform
- Xilinx Vivado® Design Suite latest version, as mentioned in “HDL Language Support and Supported Third-Party Tools and Hardware”
- ZedBoard

To setup the ZedBoard, refer to the *Set up Zynq hardware and tools* section in the “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98 example. Connect an audio input from a mobile or an MP3 player to the **LINE IN** jack and either earphones or speakers to the **HPH OUT** jack on the ZedBoard as shown below.



Model Audio Processing Algorithm

Open the model `hdlcoder_audio_filter_multistream`.

```
open_system('hdlcoder_audio_filter_multistream')
set_param('hdlcoder_audio_filter_multistream', 'SimulationCommand', 'Update')
```

The model contains the DUT subsystem for audio processing, source and sink blocks for simulating the audio, and plant models for DMAs that transfer stream data between the Processing System and FPGA.

Rate Considerations

For audio applications running on the FPGA, the FPGA clock rate is several times faster than the audio sample rate. The ratio of the FPGA clock rate to the audio data sample rate is the **Oversampling factor**. In this example, the **Oversampling factor** is modeled by using Repeat and Upsample blocks.

Modeling your design at the FPGA clock rate allows you to optimize resource usage on the target hardware platform by leveraging idle clock cycles and reusing various components. The audio application illustrated in this example uses an audio sample rate of `48kHz` and an FPGA clock rate of `96MHz`. The **Oversampling factor** in this case is `2000`. Such a large value of **Oversampling factor** slows down the Simulink® simulation significantly.

To reduce the simulation time, instead of using the **Oversampling factor** setting, you can model your design at the minimum **Oversampling factor** that is required by the design. The minimum required **Oversampling factor** for the design can be determined by the length of the audio filter, which is `1001`. This value reduces the simulation time by half and provides sufficient idle cycles between the data samples for the serial filter logic.

Audio Filter

Inside the DUT subsystem, the FIR filter processes data from the audio codec AXI4-Stream channel. The filter coefficients are generated in MATLAB® and programmed by using the second AXI4-Stream interface that interfaces with the Processing System. The filtered audio output is streamed back to the audio codec.

The audio filter is a fully serial implementation of an FIR filter. This filter structure is best suited for audio applications that require large **Oversampling factor** because the filter uses a multiply accumulate (MAC) operation for each channel. The filter also uses RAM blocks to implement the data delay line and the coefficient source. This implementation saves area by avoiding the high slice logic usage of high-order filters.

Spectrum Analyzer

The audio signal is fed into a spectrum analyzer after passing through the FIR filter. The spectrum analyzer computes the FFT of the filtered signal, applies a weighting function, and converts the result to dBm. You can program the type of weighting to be performed by using the AXI4-Lite interface as either No-weighting, A-weighting, C-weighting, or K-weighting. The actual weighting functions are implemented using lookup tables that have been generated by using Audio Toolbox™ function `weightingFilter`.

Model AXI4-Stream Interfaces

The model contains two AXI4-Stream interfaces. One AXI4-Stream interface communicates with the audio codec. The other AXI4-Stream interface communicates with the Processing System through the DMAs. The audio codec interface only requires the `Data` and `Valid` signals. The DMA interface, on the other hand, additionally uses the `Ready` and `TLAST` signals of the AXI4-Stream protocol.

To learn more about the signals used in AXI4-Stream modeling, see “Model Design for AXI4-Stream Interface Generation” on page 40-14.

Ready Signal

In an AXI4-Stream interface, you use the `Ready` signal to apply or respond to back pressure. The model uses the `Ready` signal on the AXI4-Stream Master channel from the FPGA to the Processing System to respond to back pressure from the DMA. When the downstream DMA cannot receive more spectrogram samples, it de-asserts the input `Ready` signal on the AXI4-Stream Master channel. To ensure that the spectrogram samples are not dropped, the model buffers the data in a FIFO until the `Ready` signal is asserted, indicating that the DMA is ready to receive samples again.

The AXI4-Stream Slave channel from the Processing System to the FPGA does not have to apply back pressure, and hence its `Ready` signal is always asserted. The audio codec does not process back pressure and does not use its `Ready` signal on either channel. To learn more about the `Ready` signal in AXI4-Stream modeling, see “Ready Signal (Optional)” on page 40-16.

TLAST Signal

The `TLAST` signal is used to indicate the last sample of a frame. The model uses the `TLAST` signal on the AXI4-Stream Slave channel as an indicator that it has received a full set of filter coefficients. On the AXI4-Stream Master channel, the `TLAST` signal is used to indicate the end of a spectrum analyzer frame. To learn more about the `TLAST` signal in AXI4-Stream modeling, see “TLAST Signal (optional)” on page 40-20.

Customize the Model for ZedBoard

To implement this model on the ZedBoard, you must first have a reference design in Vivado that receives audio input on the ZedBoard and transmits the processed audio data out of the ZedBoard. For details on how to create a reference design which interfaces with the audio codec on the ZedBoard, see “Authoring a Reference Design for Audio System on a Zynq Board” on page 40-226. This example extends the reference design in that example by adding DMA IPs for communication with the Processing System.

In the reference design, left and right channel audio data are combined to form a single channel such that the lower 24 bits form the left channel and upper 24 bits form the right channel. In the Simulink model shown above, `CODEC_data_in` is split into left and right channels. Filtering is done on each channel individually. The channels are then concatenated to form a single channel for `CODEC_data_out`.

Generate HDL IP Core with AXI4-Stream Interfaces

Next, you can start the HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy this design on the Zynq hardware. For a more detailed step-by-step guide, you can refer to the Getting Started with HW/SW Co-design Workflow for Xilinx Zynq Platform example.

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', ...
        'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.bat');
```

2. Add both the IP repository folder and the ZedBoard registration file to the MATLAB path using following commands:

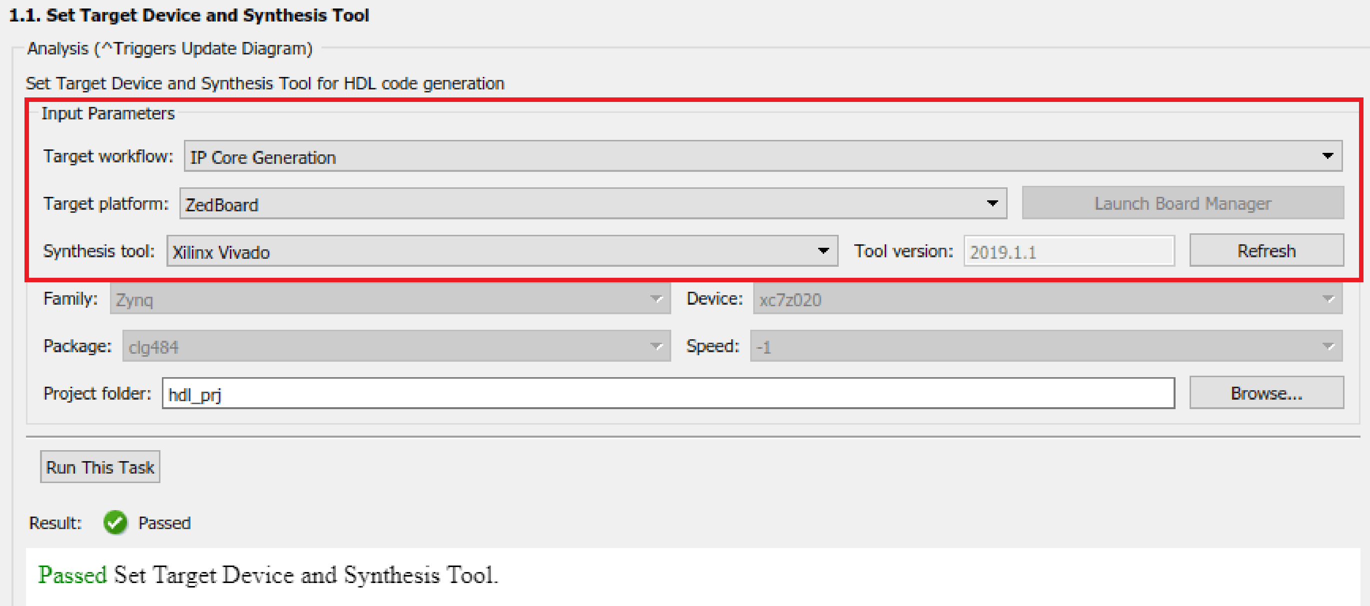
```
example_root = (hdlcoder_amd_examples_root)
cd (example_root)
addpath(genpath('ipcore'));
addpath(genpath('ZedBoard'));
```

3. Open the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_audio_filter_multistream/DUT` or double-click the **Launch HDL Workflow Advisor** box in the model.

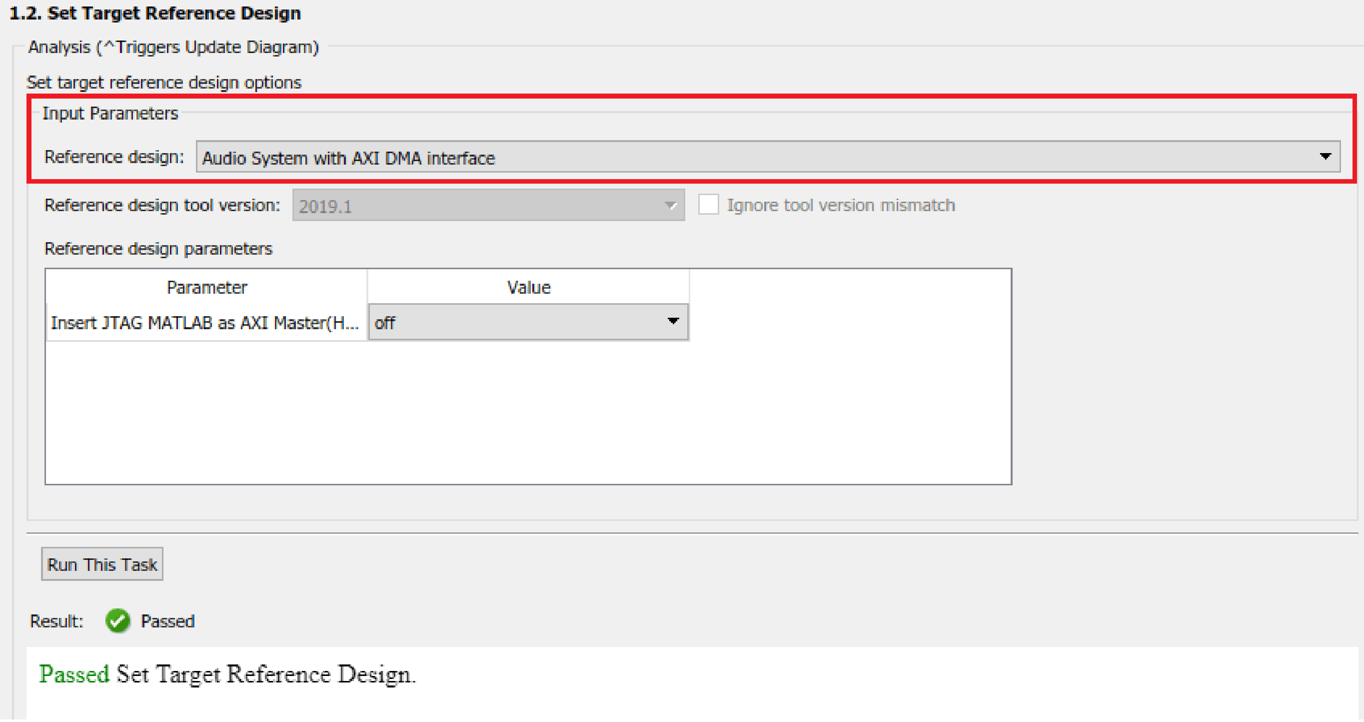
The target interface settings are already saved for ZedBoard in this example model, so the settings in tasks **1.1** to **1.3** are automatically loaded. To learn more about saving target interface settings in the model, you can refer to the Save Target Hardware Settings in Model example.

4. Run the **Set Target Device and Synthesis Tool** task.

In this task, `IP Core Generation` is selected for **Target workflow**, and `ZedBoard` is selected for **Target platform**.



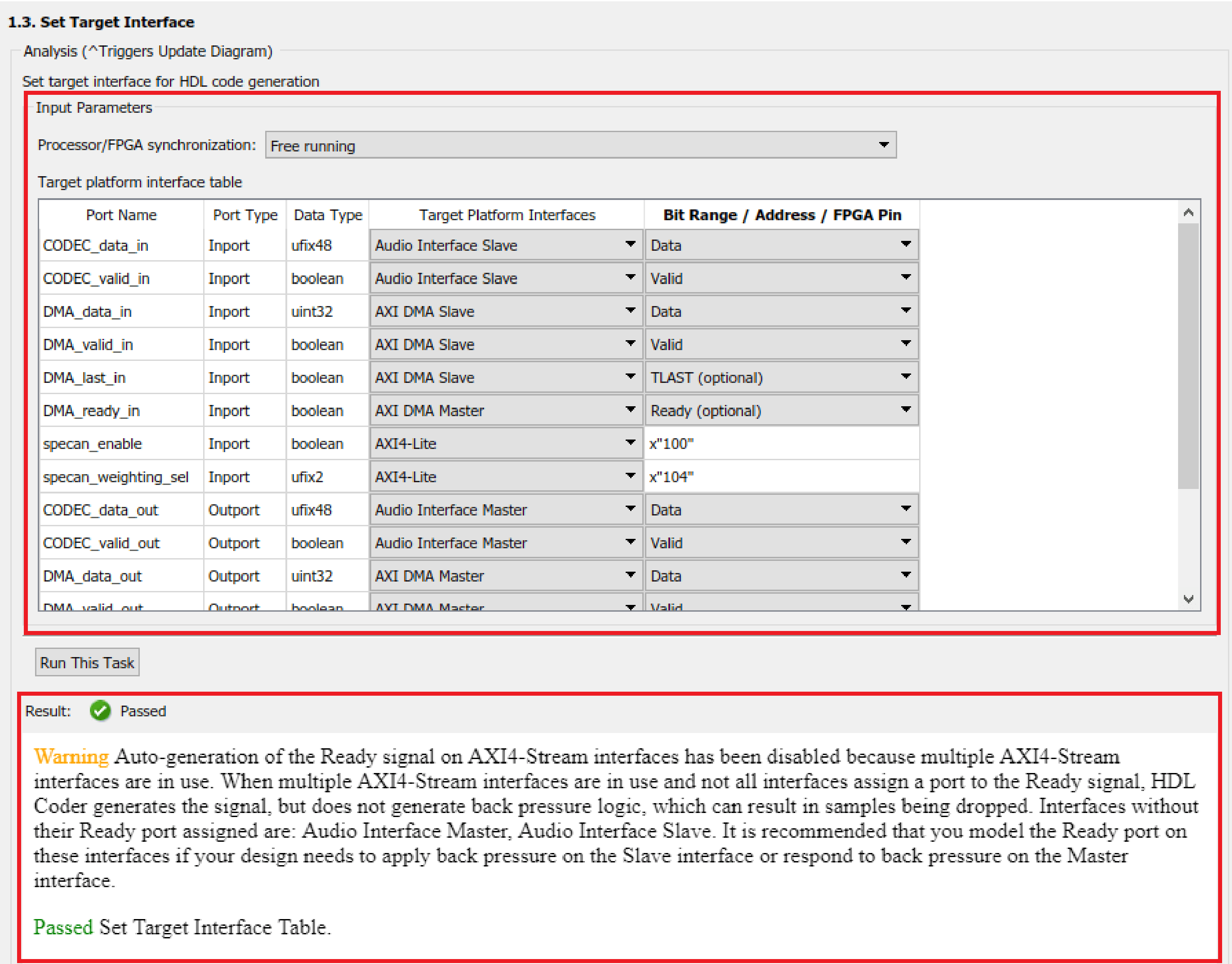
5. Run the **Set Target Reference Design** task. `Audio system with DMA Interface` is selected as the **Reference Design**.



6. Run the **Set Target Interface** task.

In this task, the ports of the DUT subsystem are mapped to the IP Core interfaces. The audio codec ports are mapped to the Audio Interface and the DMA ports are mapped to the AXI DMA interface. These are both AXI4-Stream interfaces. The AXI4-Stream interface communicates in master/slave

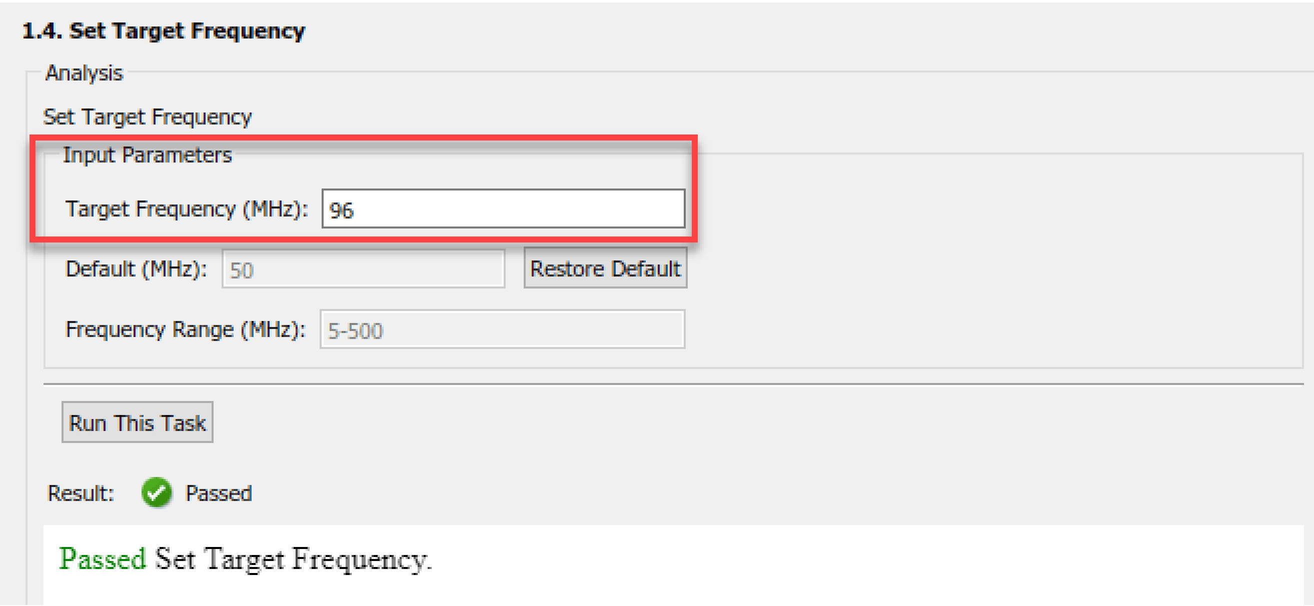
mode, where the master device sends data to the slave device. Therefore, if a data port is an input port, it is assigned to an `AXI4-Stream Slave` interface, and if a data port is output port, it is assigned to an `AXI4-Stream Master` interface. The exception to this is the **Ready** signal. The `AXI4-Stream Master Ready` signal is an input to the model, and the `AXI4-Stream Slave Ready` signal is an output of the model. The spectrum analyzer control ports are mapped to `AXI4-Lite`.



Running this task issues a warning that auto-generation of the **Ready** signal is disabled, and that the **Audio Interface** does not assign a **Ready** port. You can ignore the warning for this design, because back pressure has already been accounted for. Namely, the design addressed back pressure on the DMA interface by using a FIFO. On the audio codec interface, back pressure cannot be applied, so no **Ready** signal logic is needed.

7. In the **Set Target Frequency** task, set the **Target Frequency (MHz)** to `96`. Run this task.

This target frequency value makes the **Oversampling factor** an even integer relative to the audio sample rate of `48kHz`.



8. Right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.

You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

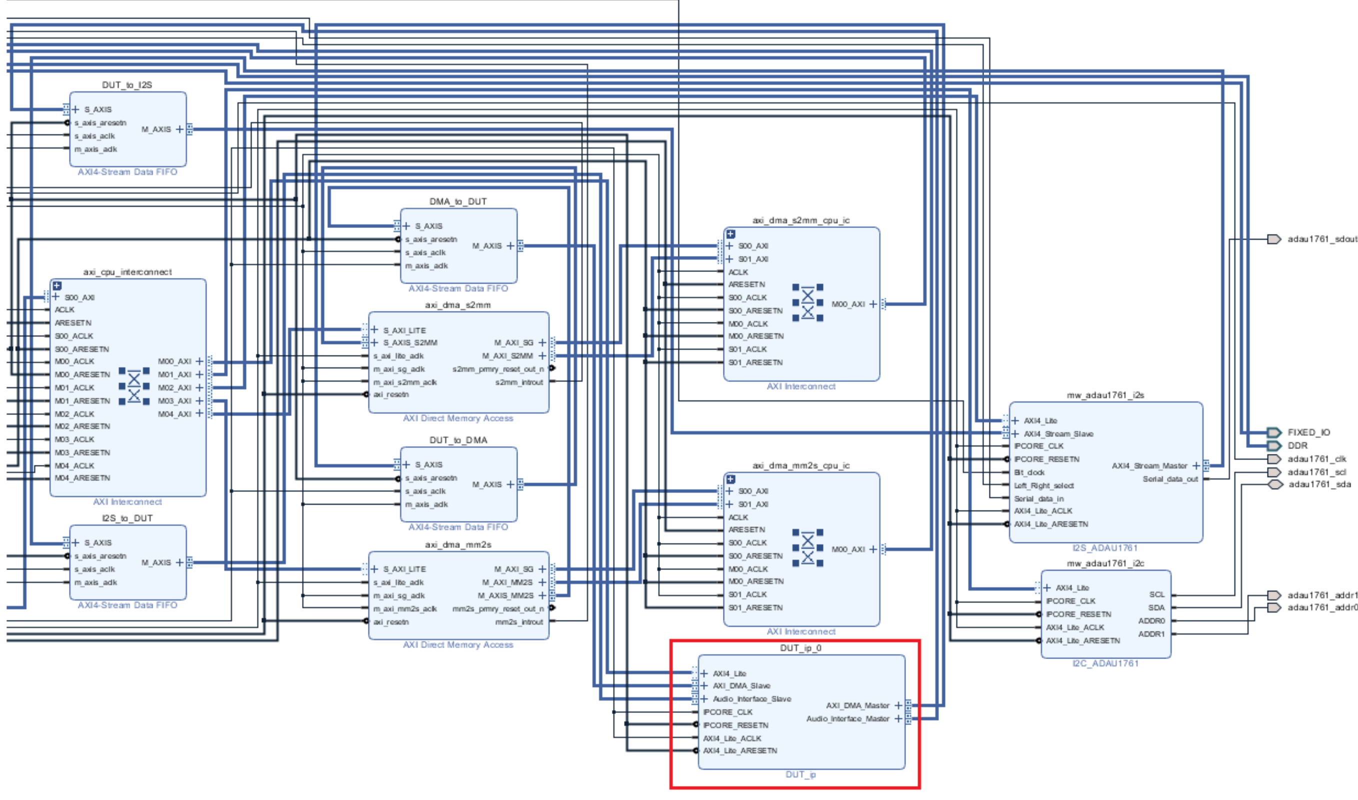
Integrate IP into AXI4-Stream Audio-Compatible Reference Design

Next, in the HDL Workflow Advisor, you run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Zynq hardware.

1. Run the **Create Project** task.

This task inserts the generated IP core into the `Audio System with AXI DMA Interface` reference design. As shown in the first diagram, this reference design contains the IPs to handle streaming audio data in and out of ZedBoard, and for streaming data in and out of the Processing System. The generated project is a complete ZedBoard design. It includes the algorithm part, which is the generated DUT algorithm IP, and the platform part, which is the reference design.

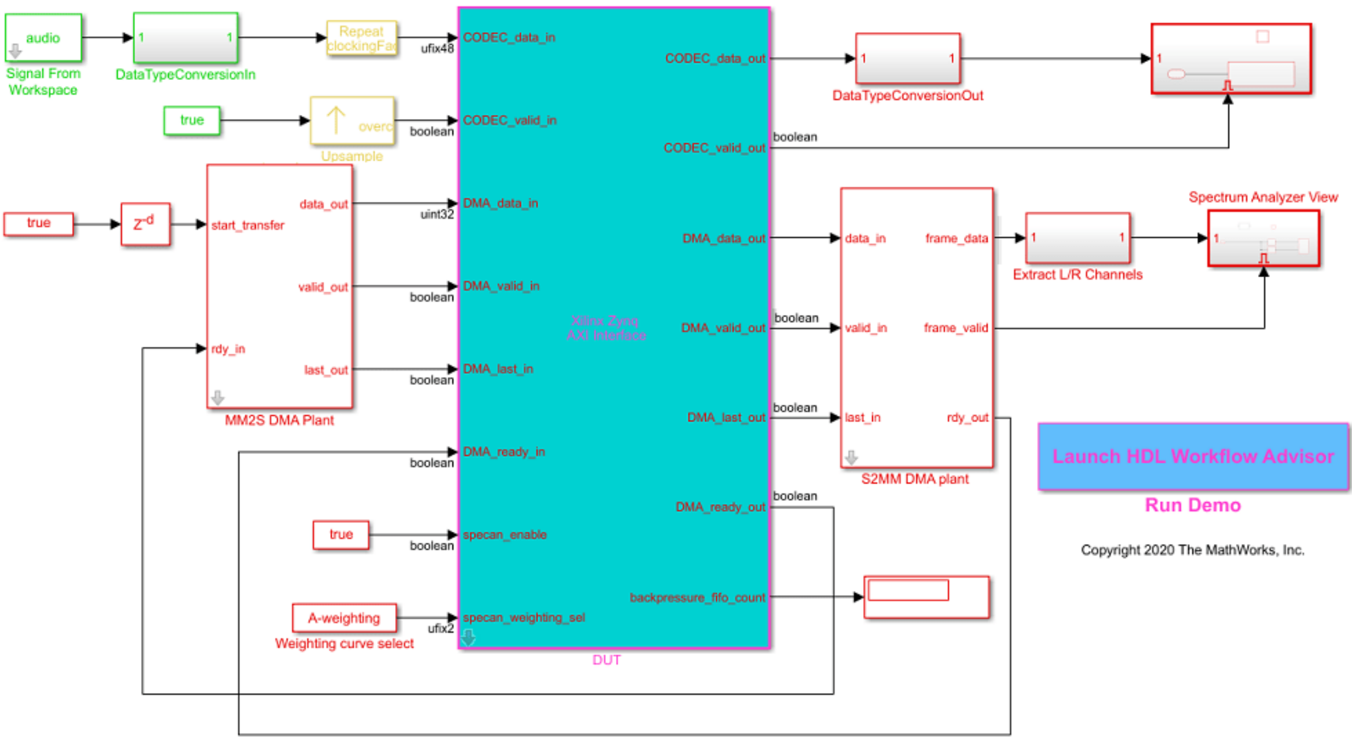
2. Click the link in the **Result** pane to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram, which includes the generated HDL IP core, other audio processing IPs and the Zynq processor.



3. In the HDL Workflow Advisor, run the remaining tasks to generate the software interface model, and build and download the FPGA bitstream. Choose `Download` programming method in the task **Program Target Device** to download the FPGA bitstream onto the SD card on the Zynq board. Your design is then automatically reloaded when you power cycle the Zynq board.

Generate ARM Executable to Tune Parameters on FPGA Fabric

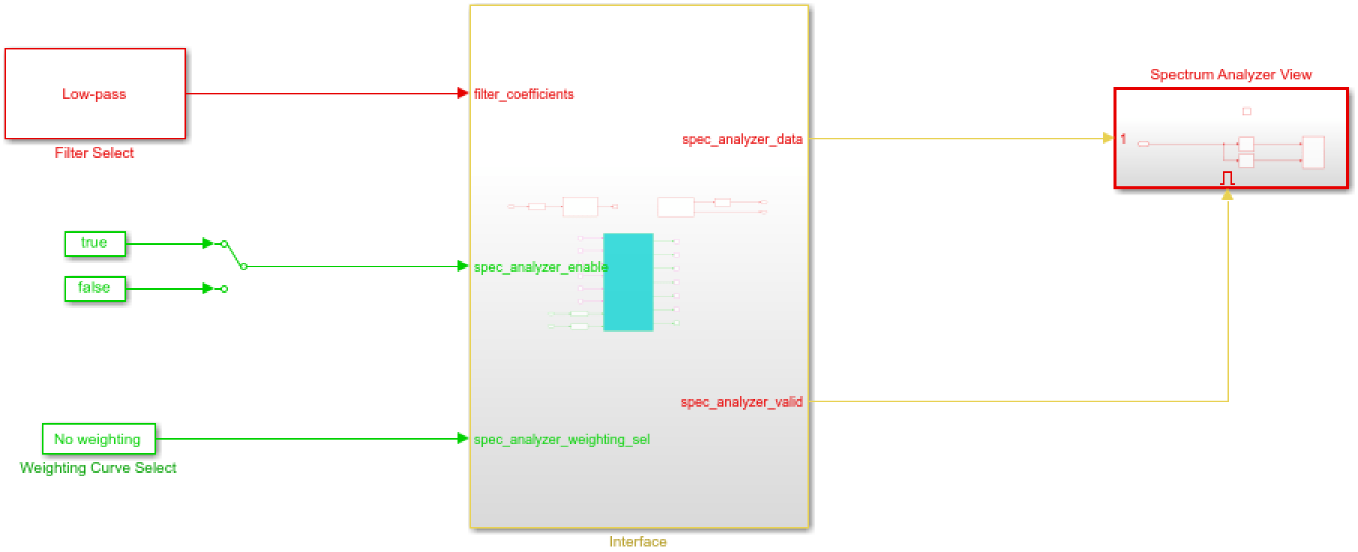
In task **Generate Software Interface Model**, a software interface model is generated.



In the generated model, AXI4-Lite driver blocks have been automatically added. However, AXI4-Stream driver blocks cannot be automatically generated, because the driver blocks expect vector inputs on the software side, but the DMA DUT ports are scalar ports. For details on how to update the software interface model with the correct driver blocks, refer to “Deploy Model with AXI-Stream Interface in Zynq Workflow” on page 40-192.

For this example, you use an updated software interface model. To open this model, run:

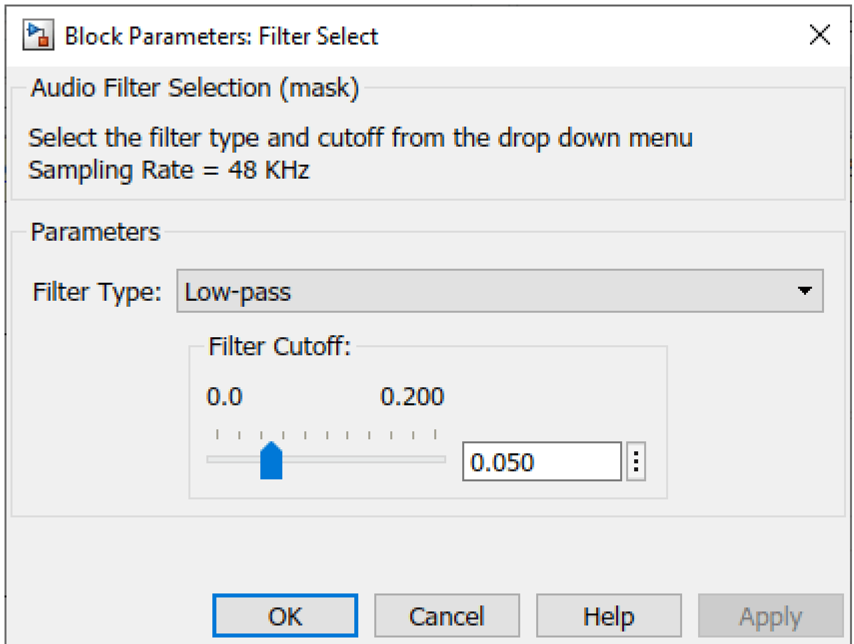
```
open_system('hdlcoder_audio_filter_multistream_sw');
```



To tune the parameters:

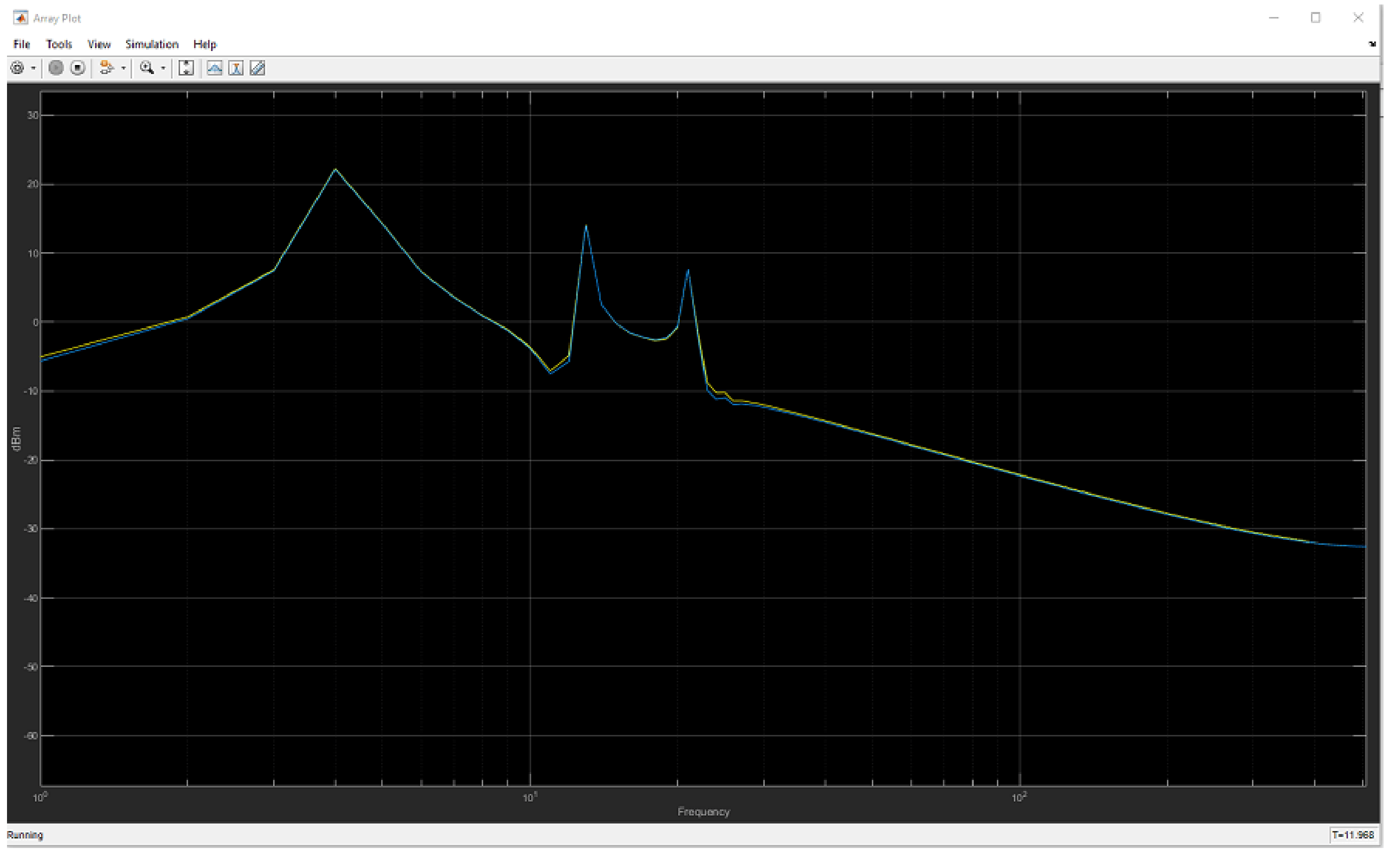
1. Click the **Monitor & Tune** button on the **Hardware** tab of model toolstrip. Embedded Coder builds the model, downloads the ARM® executable to the ZedBoard hardware, executes it, and connects the model to the running executable. While the model is running, different parameters can be tuned.

2. You can select the type of filter by using the **Filter Type** block parameter of the Filter Select block. The filter coefficients are calculated in this block using the `fir1` function from Signal Processing Toolbox™. The coefficients are sent from the Processing System to the FPGA by using the AXI4-Stream IIO Write block, which communicates through the **MM2S DMA IP**.



3. The weighting curve used by the spectrum analyzer can be selected using the **Curve** block parameter of the Weighting Curve Select block. The selection is sent from the Processing System to the FPGA using the AXI4-Lite interface.

4. The spectrum analyzer output can be viewed in the Array Plot. Select a different filter type or modify the weighting curve and observe how the spectrum data changes.



The filtered audio output can be heard by plugging earphones or speakers to **HPH OUT** jack on the ZedBoard.

See Also

More About

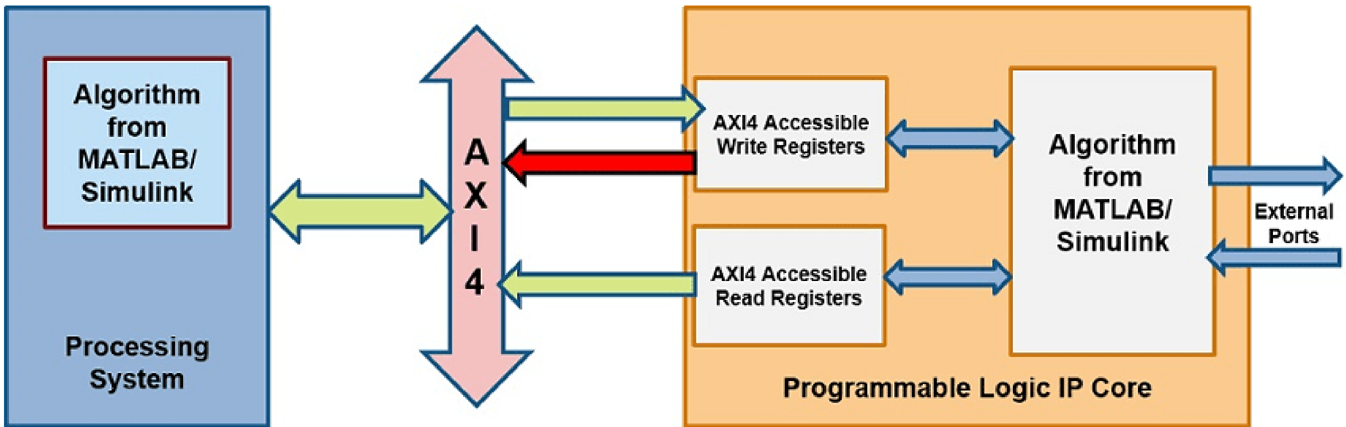
- "Hardware-Software Co-Design Workflow for SoC Platforms" on page 39-9
- "Custom IP Core Generation" on page 39-17
- "Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces" on page 40-33

Inspect the Written Values of AXI4 Slave Registers by Using the Readback Methods

This example describes the different techniques to read the AXI4 slave input registers in your design. It shows the process of how to enable readback on AXI4 slave input registers and read the values of AXI4 slave input registers for your design.

Introduction

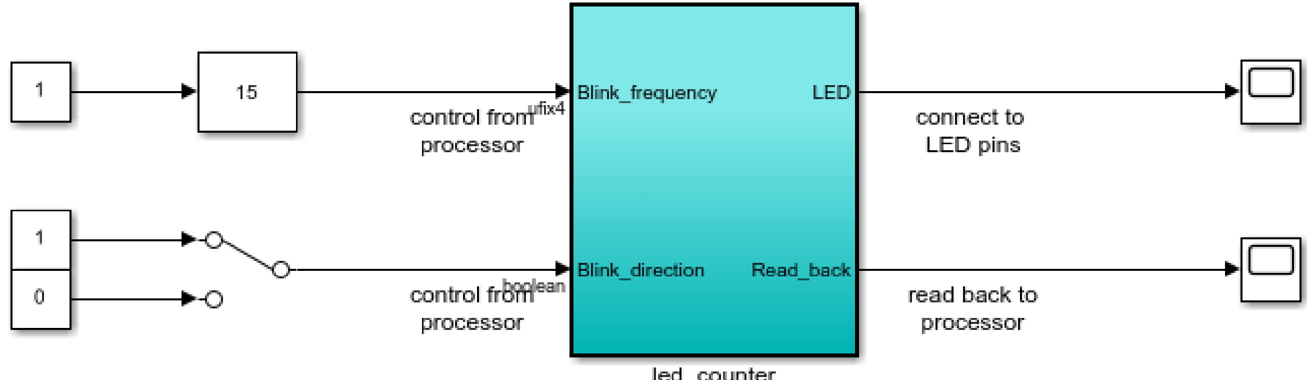
You can select the AXI4 or AXI4-Lite interface in the IP core generation or the Simulink® Real-Time FPGA I/O workflow in HDL Workflow Advisor. By default, your IP is generated without the readback capability of the AXI4 or AXI4-Lite input registers. When the readback feature is enabled from HDL Workflow Advisor or from command-line interface, you can read the values of input registers. This technique is useful for debugging the input values, which are written into AXI4 or AXI4-Lite interface. The figure shows the AXI4 slave interface and its registers.



Enabling AXI4 Slave Input Register Readback

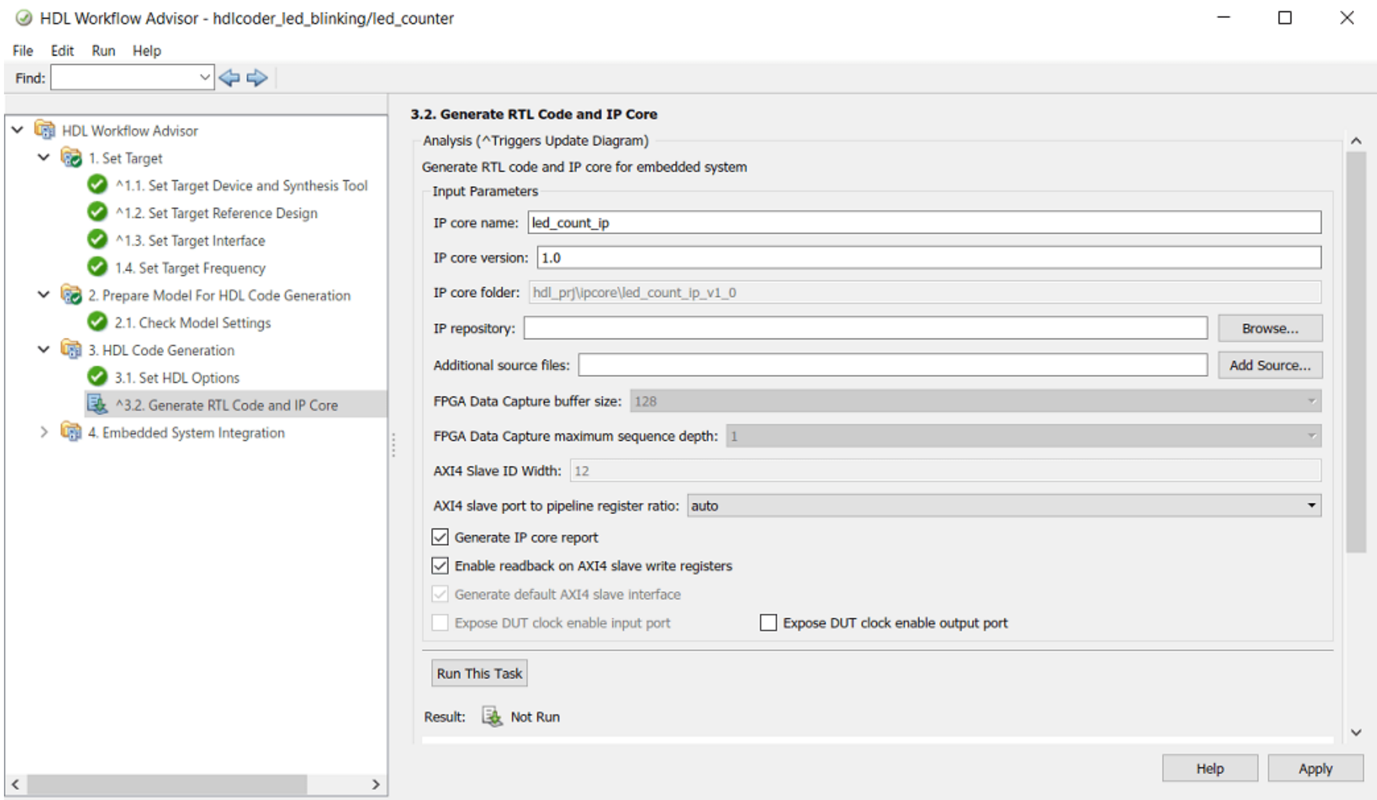
1. When your design contains the AXI4 or AXI4 Lite interface, you can perform the readback of AXI4 registers. You can use any model which has these interfaces. Use **hdlcoder_led_blinking** or **hdlcoder_led_blinking_4bit** model. Open the Simulink model that implements LED blinking by entering this command:

```
open_system('hdlcoder_led_blinking');
```



2. Run all the tasks up until **Set HDL Options**.

3. You can turn on the readback on AXI4 slave input registers by using the HDL workflow Advisor or by using the command-line interface. Generate RTL code and IP Core, by default, the readback on AXI4 slave input register is turned off. To enable this option, select **Enable readback on AXI4 slave write registers** under the **Generate RTL Code and IP Core** task of HDL Workflow Advisor.

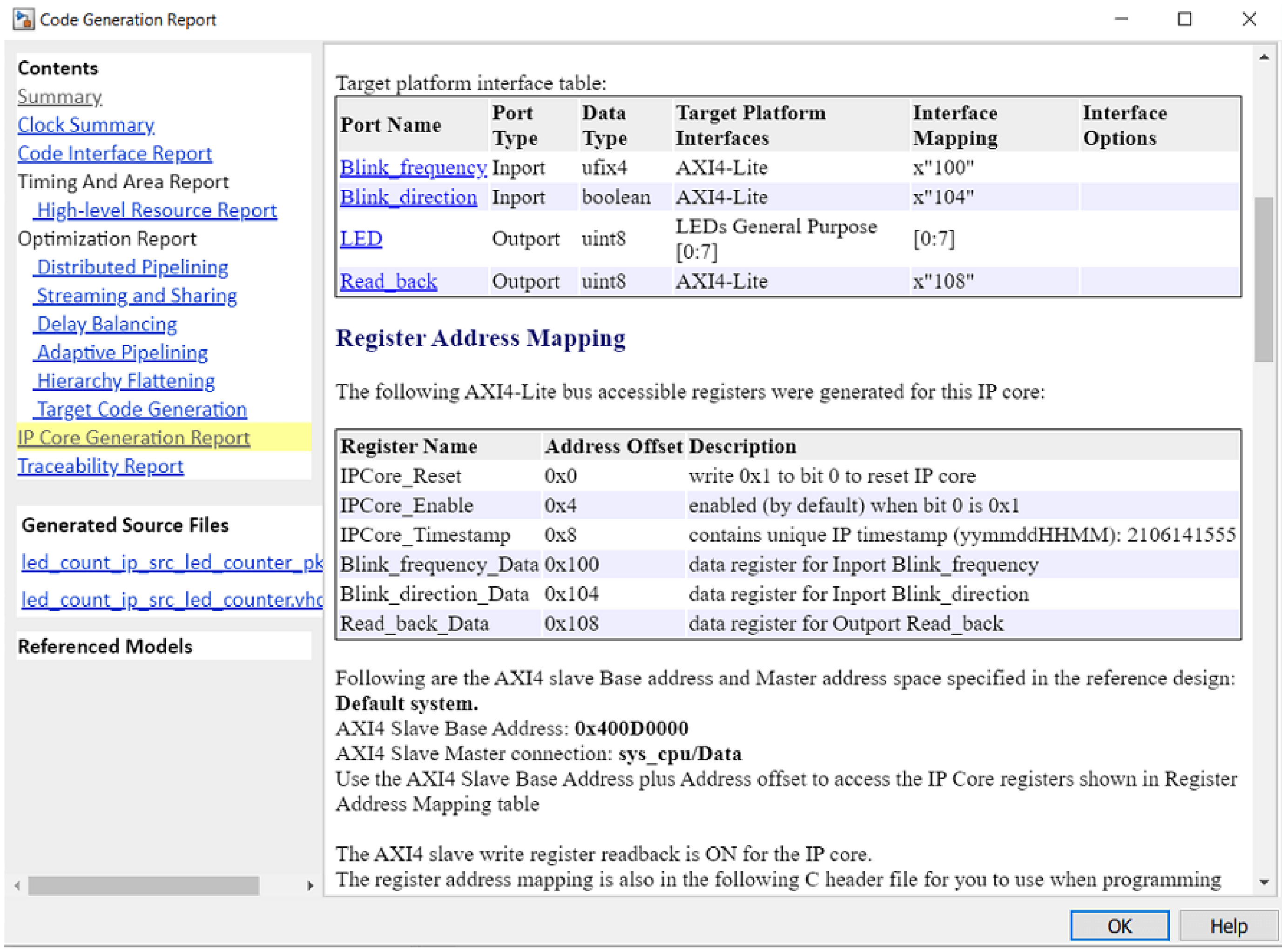


You can also enable the readback option at the MATLAB command line by using the `hdlset_param` function.

```
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter', 'AXI4RegisterReadback', 'on');
hdlset_param('hdlcoder_led_blinking/led_counter', 'AXI4SlaveIDWidth', '12');
hdlset_param('hdlcoder_led_blinking/led_counter', 'ProcessorFPGASynchronization',
```

Read Values of the AXI4 input registers

After setting the readback option for your model, generate the RTL code and the IP Core in the HDL Workflow Advisor or through the command-line interface (CLI). Once the IP core is generated, the IP Core Generation Report is generated in the code generation report. The IP Core Generation Report contains the details about the Target Platform Interface of your model. The figure shows the AXI-Lite interface mapped to the hdlcoder_led_blinking model ports. The table in the IP Core Generation Report shows the interface mapping address of each AXI4 slave register. These addresses are used to read the AXI4 slave input registers.



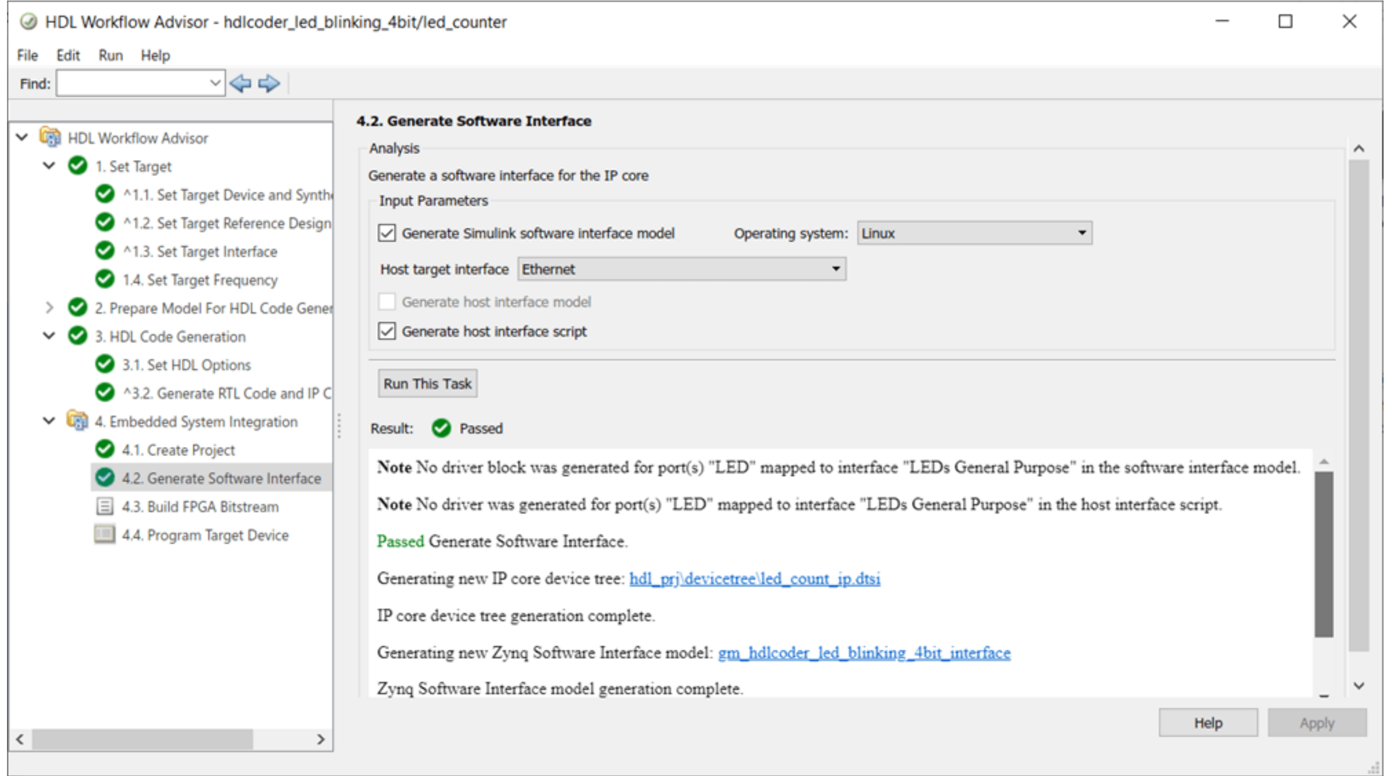
The AXI4 slave base address is used together with interface mapping address to read the value of AXI4 slave input registers. These address mapping details are used to read the AXI4 slave registers. You can perform the readback of input registers in these ways:

- Using the MATLAB FPGA Prototyping API Script
- Using the JTAG AXI Manager
- Using the Devmem command

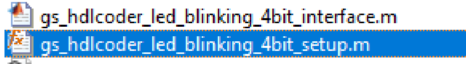
To read the value, you can use any of the preceding techniques. Then you must complete all the steps in HDL Workflow Advisor. Once the bitstream is generated and programmed into the target device, perform readback of input registers.

Readback of AXI4 Slave Input Registers by Using MATLAB FPGA Prototyping API Script

You can read AXI4 Slave input registers by using MATLAB® FPGA prototyping API scripts. Once the RTL code and IP core are generated, host interface scripts can be generated by using the **Generate Software Interface** task from the HDL Workflow Advisor. The option **Generate host interface script** generates the MATLAB prototyping API scripts.



This option generates two scripts that are interface and setup scripts, shown here for the `hdlcoder_led_blinking_4bit` model.



The setup function contains commands for the AXI4 slave interfaces that HDL Coder uses to control the DUT ports in the generated HDL IP core, which are mapped to their corresponding interfaces.

```
function gs_hdlcoder_led_blinking_4bit_setup(hFPGA)
%--------------------------------------------------------------------------
% Host Interface Script Setup
%
% Generated with MATLAB 9.12 (R2022a) at 16:26:53 on 10/01/2022.
% This function was created for the IP Core generated from design 'hdlcoder_led_blinking_4bit'.
%
% Run this function on an "fpga" object to configure it with the same interfaces as the generated IP core.
%--------------------------------------------------------------------------

%% AXI4-Lite
addAXI4SlaveInterface(hFPGA, ...
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0x400D0000, ...
    "AddressRange", 0x10000, ...
    "WriteDeviceName", "mwipcore0:mmwr0", ...
    "ReadDeviceName", "mwipcore0:mmrd0");

DUTPort_Blink_frequency = hdlcoder.DUTPort("Blink_frequency", ...
    "Direction", "IN", ...
    "DataType", numerictype(0,4,0), ...
    "IsComplex", false, ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x100");

DUTPort_Blink_direction = hdlcoder.DUTPort("Blink_direction", ...
    "Direction", "IN", ...
    "DataType", "logical", ...
    "IsComplex", false, ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x104");

DUTPort_Read_back = hdlcoder.DUTPort("Read_back", ...
    "Direction", "OUT", ...
    "DataType", "uint8", ...
    "IsComplex", false, ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x108");

mapPort(hFPGA, [DUTPort_Blink_frequency, DUTPort_Blink_direction, DUTPort_Read_back]);

end
```

The host interface script instantiates this setup function to connect to the target and send read or write commands. You can uncomment and send meaningful data by using the inputs to the DUT in your original model. After interfacing with the hardware, the script disconnects from the hardware resource associated with the FPGA object.

```
% Use this script to access DUT ports in the design that were mapped to compatible IP core interfaces.
% You can write to input ports in the design and read from output ports directly from MATLAB.
% To write to input ports, use the "writePort" command and specify the port name and input data. The input data will be cast to the DUT port's data type before writing.
% To read from output ports, use the "readPort" command and specify the port name. The output data will be returned with the same data type as the DUT port.
% Use the "release" command to release MATLAB's control of the hardware resources.
%---------------------------------------------------------------------

%% Create fpga object
hFPGA = fpga("Xilinx");

%% Setup fpga object
% This function configures the "fpga" object with the same interfaces as the generated IP core
gs_hdlcoder_led_blinking_4bit_setup(hFPGA);

%% Write/read DUT ports
% Uncomment the following lines to write/read DUT ports in the generated IP Core.
% Update the example data in the write commands with meaningful data to write to the DUT.
%% AXI4-Lite
% writePort(hFPGA, "Blink_frequency", zeros([1 1]));
% writePort(hFPGA, "Blink_direction", zeros([1 1]));
% data_Read_back = readPort(hFPGA, "Read_back");

%% Release hardware resources
release(hFPGA);
```

Generate the bitstream and program the target device. To write and read the data for blink_frequency, modify the interface script and run the script. The value 10 is written to the blink_frequency and axi4read shows the readback of blink_frequency.

```
18  % This function configures the "fpga" object with the same interfaces as the generated IP core
19  gs_hdlcoder_led_blinking_4bit_setup(hFPGA);
20
21  %% Write/read DUT ports
22  % Uncomment the following lines to write/read DUT ports in the generated IP Core.
23  % Update the example data in the write commands with meaningful data to write to the DUT.
24  %% AXI4-Lite
25  writePort(hFPGA, "Blink_frequency", 10);
26  axi4read = readPort(hFPGA,"Blink_frequency");
27  % writePort(hFPGA, "Blink_direction", zeros([1 1]));
28  % data_Read_back = readPort(hFPGA, "Read_back");
29
30  %% Release hardware resources
31  release(hFPGA);
32
```

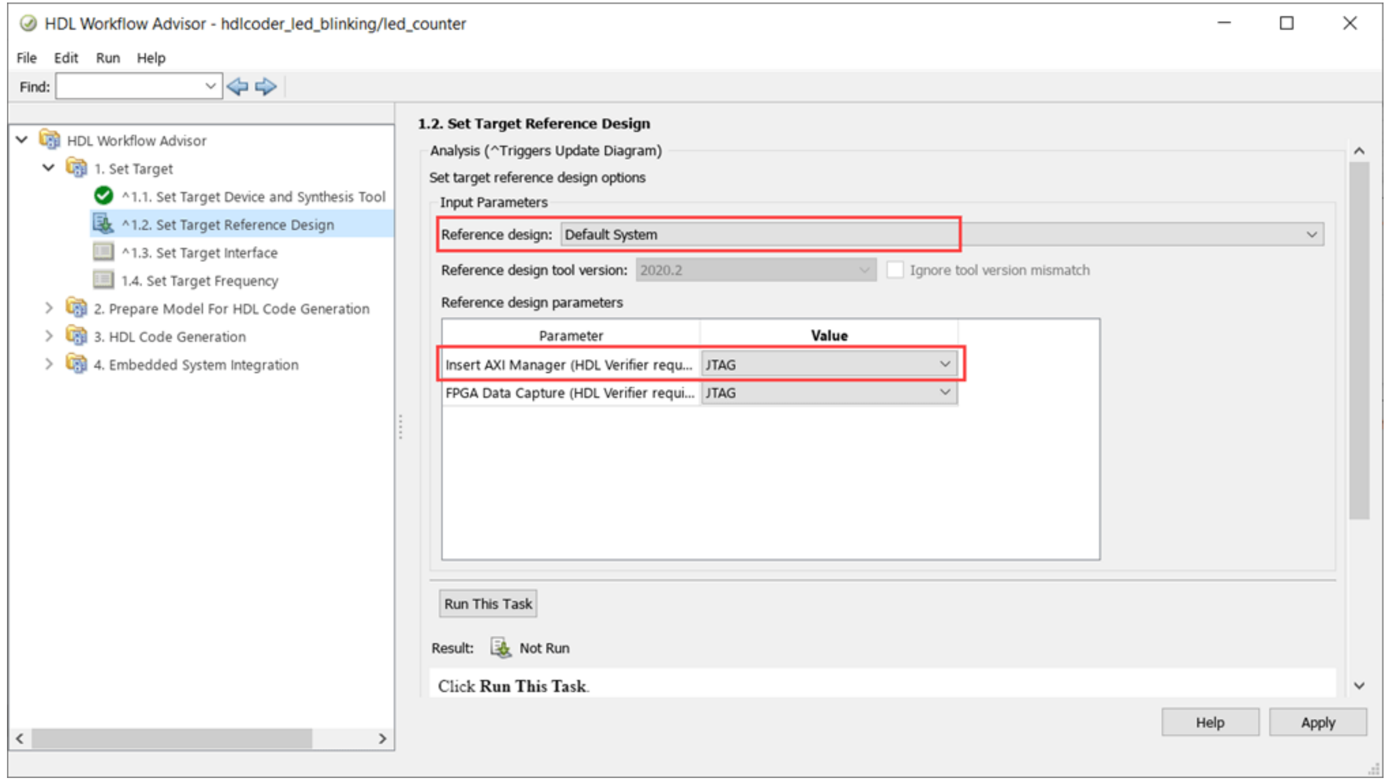
Command Window

```
axi4read =

    10

      DataTypeMode: Fixed-point: binary point scaling
        Signedness: Unsigned
        WordLength: 4
    FractionLength: 0
```

MATLAB® prototyping API uses JTAG AXI Manager for readback of AXI4 slave input registers. For more information, see “Generate and Manage FPGA I/O Host Interface Scripts” on page 39-68.

Readback of AXI4 Slave Input Registers by Using JTAG AXI Manager

You can also use separately the JTAG AXI Manager for readback of AXI4 slave input registers. To use JTAG AXI Manager, you must insert the JTAG AXI Manager IP into reference design.



Set the necessary options for enabling readback as mentioned in earlier section and follow the example “Use JTAG AXI Manager to Control HDL Coder Generated IP Core” on page 40-333 for more details.

Readback of AXI4 Slave Input Registers by Using Devmem Command (Probe Registers from Target)

You can use the devmem command in Putty or a hyper terminal. Once you program the bitstream into the target device, open Putty or hyper terminal by using the serial interface. To use devmem command for the hdlcoder_led_blinking model:

1. Read the data from address '400D0100':

```
devmem 0x400D0100
```

You get the value as 0x00000000.

2. Write some value in address '400D0100':

```
devmem 0x400D0100 w 0x1
```

3. Reread the address '400D0100':

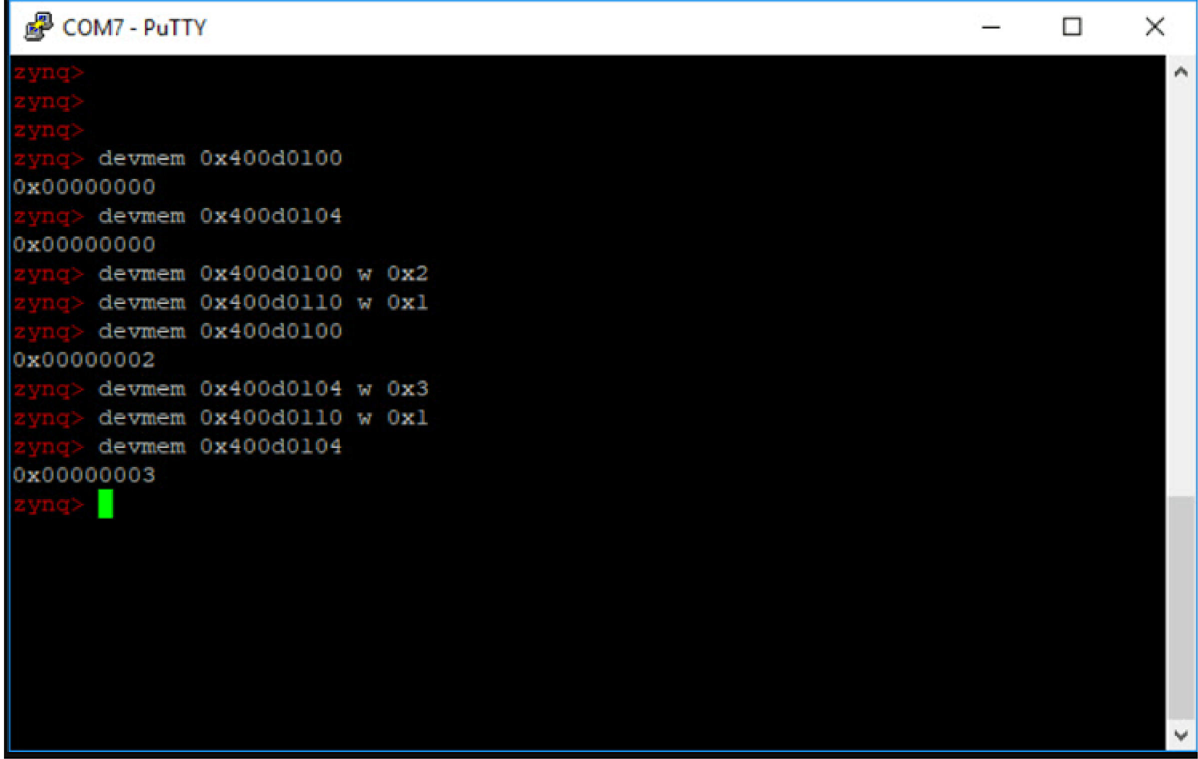
```
devmem 0x400D0100
```

You get the value as 0x00000001.

COM7 - PuTTY

```
zynq>
zynq>
zynq>
zynq> devmem 0x400d0100
0x00000000
zynq> devmem 0x400d0100 w 0x1
zynq> devmem 0x400d0100
0x00000001
zynq> devmem 0x400d0100 w 0x2
zynq> devmem 0x400d0100
0x00000002
zynq>
```

In this way scalars can be read. In the vector data type, you first need to write the data on the register, and then write 0x1 to the strobe address as in vector mode strobe synchronization done for writing. After writing on strobe address, the customer reads the data on the register by using same address. The figure shows the writing and reading of the values in the vector data type by using devmem.



1. Try to read the address 0x400D0100 and 0x400D0104, which shows as 0.

2. Try writing 0x2 on address 0x400D0100, and then write 0x1 on strobe address 0x400D0110.

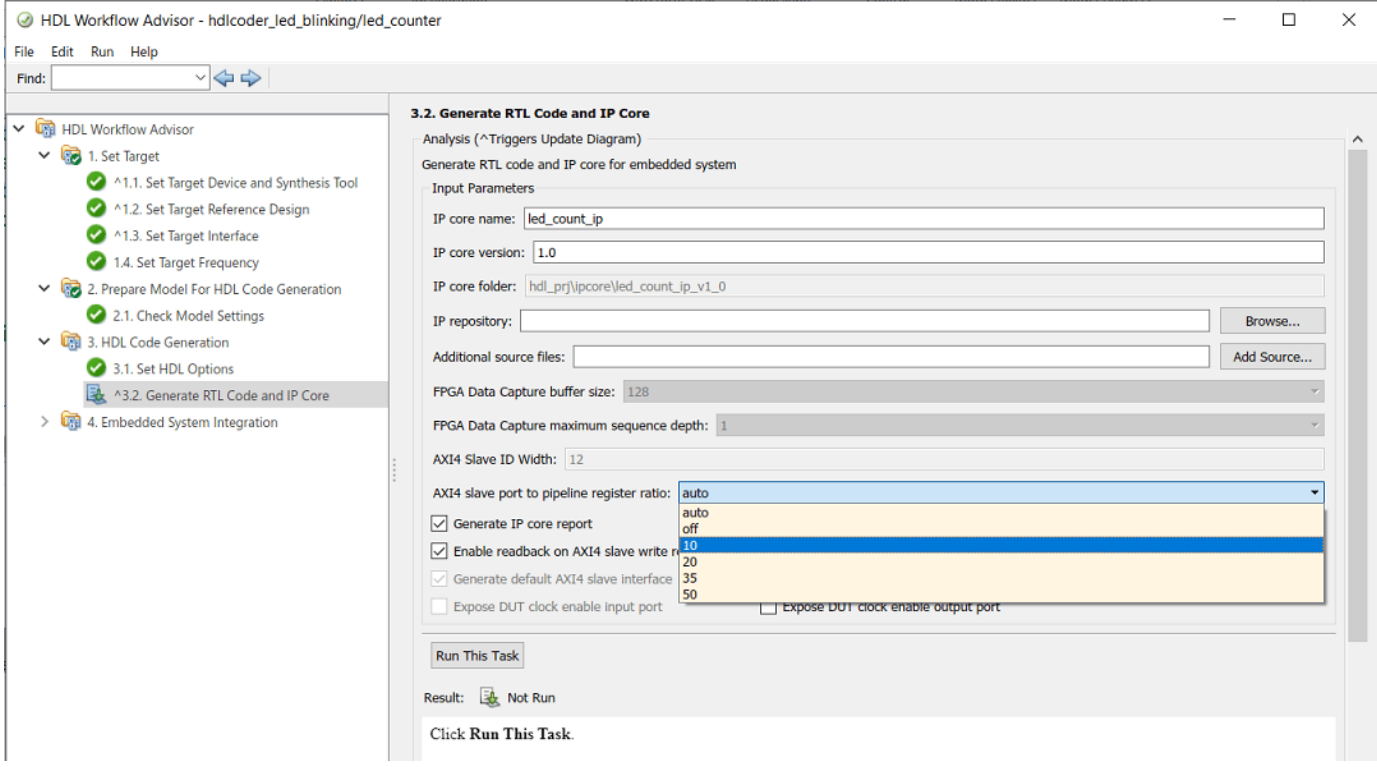
3. Read the data on address 0x400D0100. It shows as 0x00000002.

4. Write 0x3 on address 0x400D0104, and then write 0x1 on strobe address 0x400D0110.

5. Read the data on address 0x400D0104. It shows 0x00000003.

AXI4 Slave Port to Pipeline Register Ratio

This option is an additional option to optimize your design to meet frequency that you might require. When you have a considerable number of AXI4 slave input ports in the design and AXI4 slave readback of input register is turned on, the design becomes complex and you might not get timing requirements. To optimize the readback capability of the AXI4 slave input registers, you can insert pipelined registers for a number of AXI4 slave ports. You can set the **AXI4 slave port to pipeline register ratio** option from HDL Workflow Advisor, the CLI, or from the HDL Block properties of the DUT subsystem. The figure shows the HDL Workflow Advisor to set the **AXI4 slave port to pipeline register ratio** option.



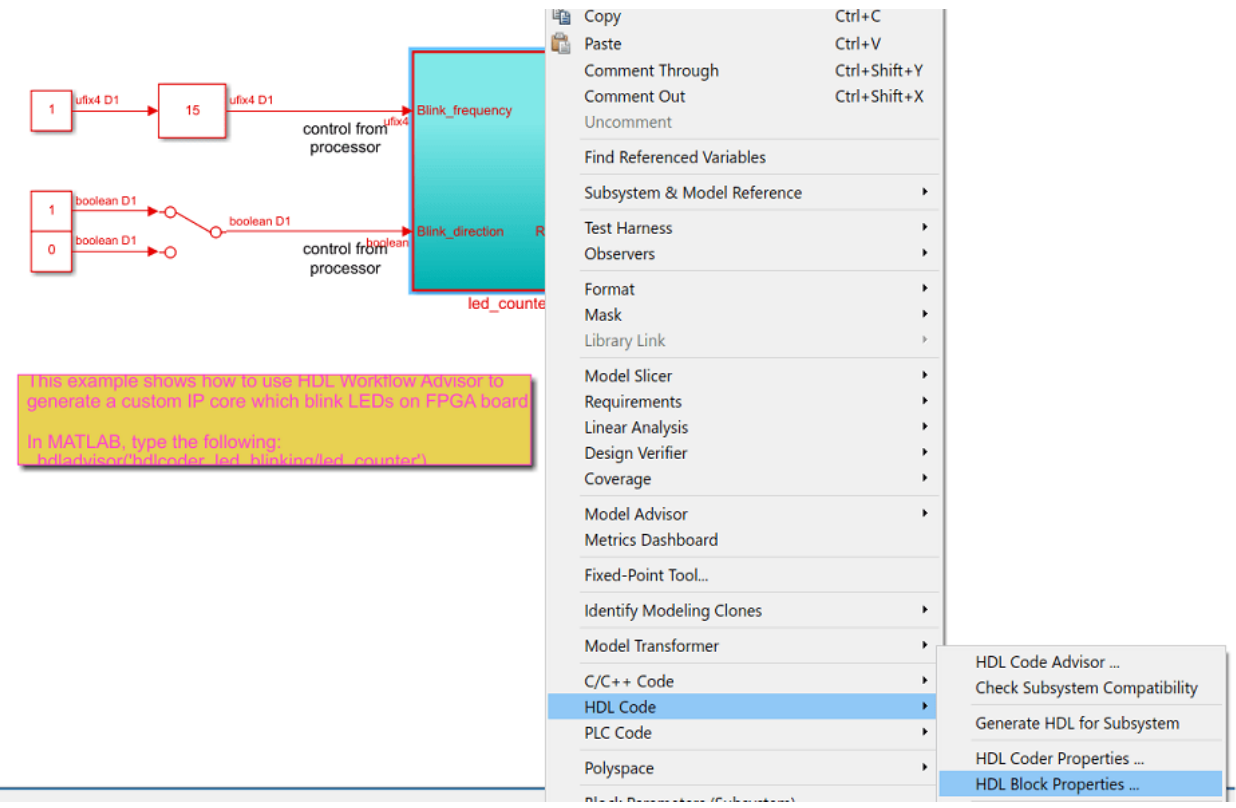
The table shows the drop-down options available in **AXI4 slave port to pipeline ratio.**

Sr. No	Dropdown Item	Description
1	Off	Pipelining Register insertion is set to off and no register will be inserted in the mux chain of port
2	Auto	One pipelining register will be inserted per 35 AXI4 ports. It is default option
3	10	One pipelining register will be inserted per 10 AXI4 ports
4	20	One pipelining register will be inserted per 20 AXI4 ports
5	35	One pipelining register will be inserted per 35 AXI4 ports
6	50	One pipelining register will be inserted per 50 AXI4 ports

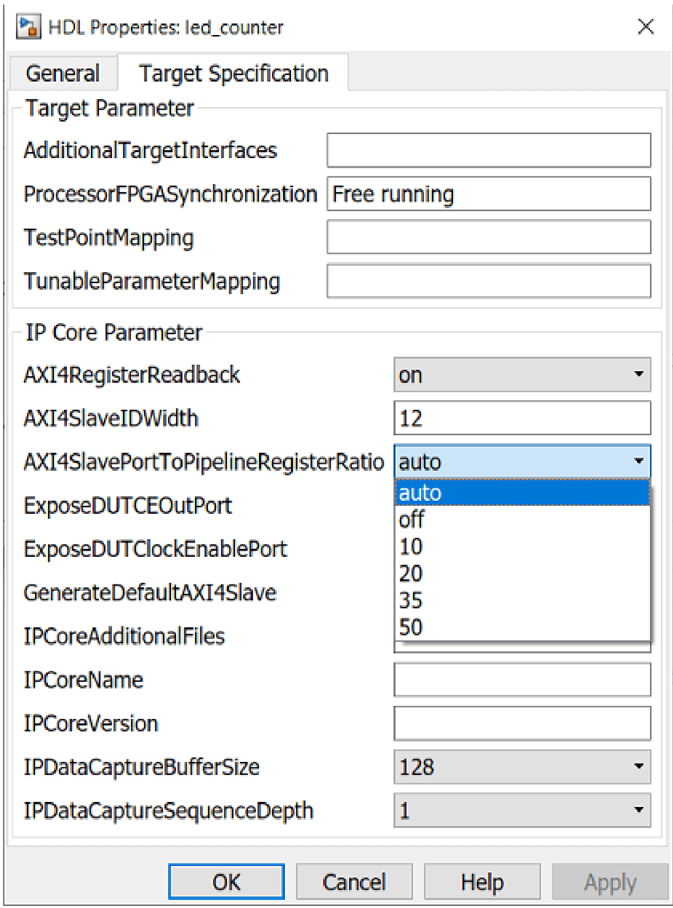
You can also set this option at the MATLAB command line by using the `hdlset_param` function.

```
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter', 'AXI4RegisterReadback', 'on');
hdlset_param('hdlcoder_led_blinking/led_counter', 'AXI4SlaveIDWidth', '12');
hdlset_param('hdlcoder_led_blinking/led_counter', 'AXI4SlavePortToPipelineRegisterRatio', '10');
hdlset_param('hdlcoder_led_blinking/led_counter', 'ProcessorFPGASynchronization', 'Free running');
```

To select this option by using HDL Block properties, right-click DUT subsystem and select **HDL Code > HDL Block Properties**.



In HDL Block Properties, on the **Target Specification** tab and set **AXI4SlavePortToPipelineRegisterRatio** in the **IP Core Parameter** section.



Once you set the **AXI4 slave port to pipeline register ratio** by using one of the receding options, follow the steps for the readback as described in the section **Read Values of the AXI4 Slave input registers**.

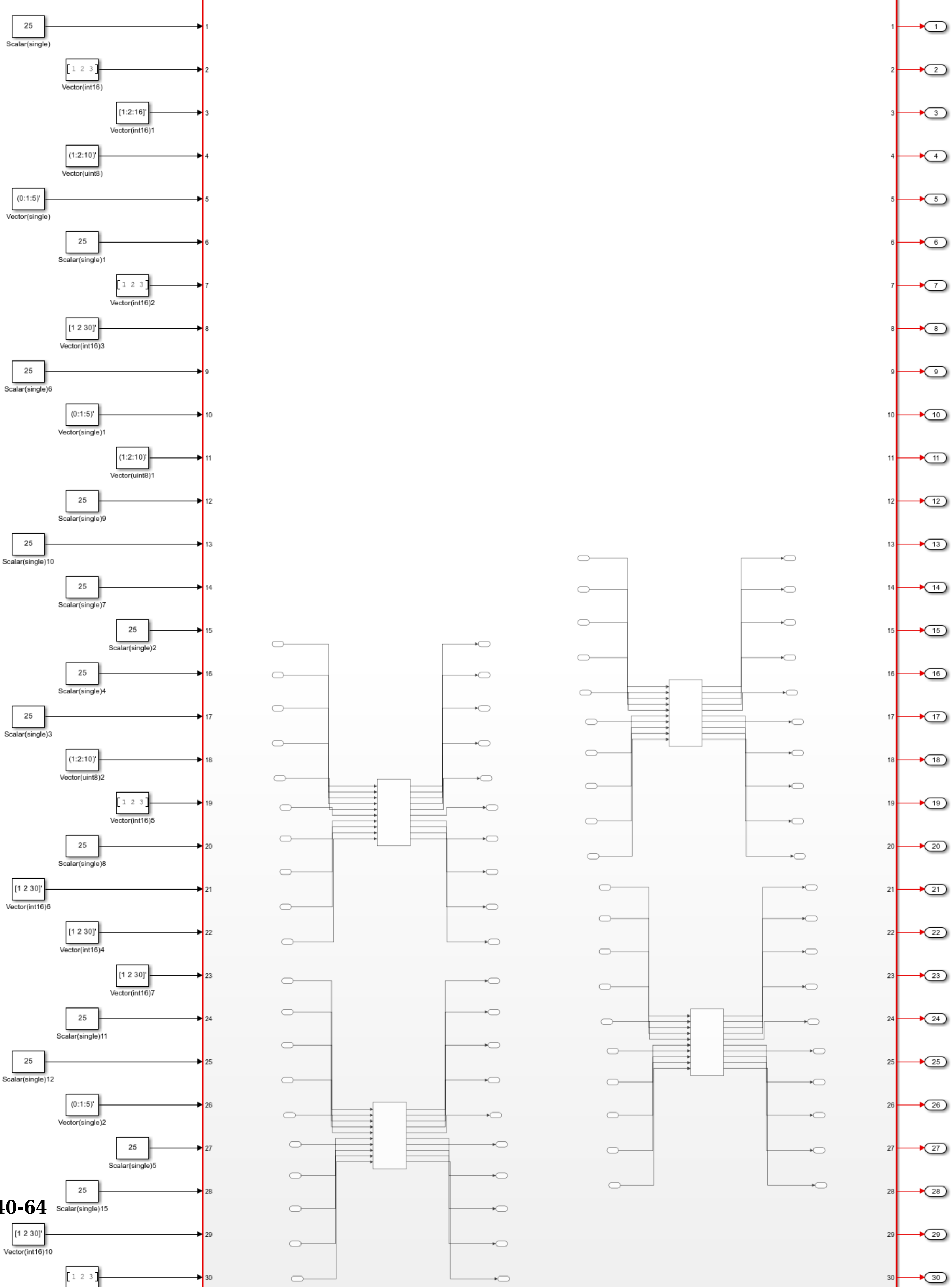
Enabling AXI4 Slave Input Register Readback for individual ports

If your design contains a large number of input ports and you enable readback on all of the input ports, your design can consume lot of resources and can have timing violations.

Instead of enabling the global readback on all the input ports, You can enable the readback for individual ports based on your requirement.

1. Below is the example Simulink model which contains large number of input ports.

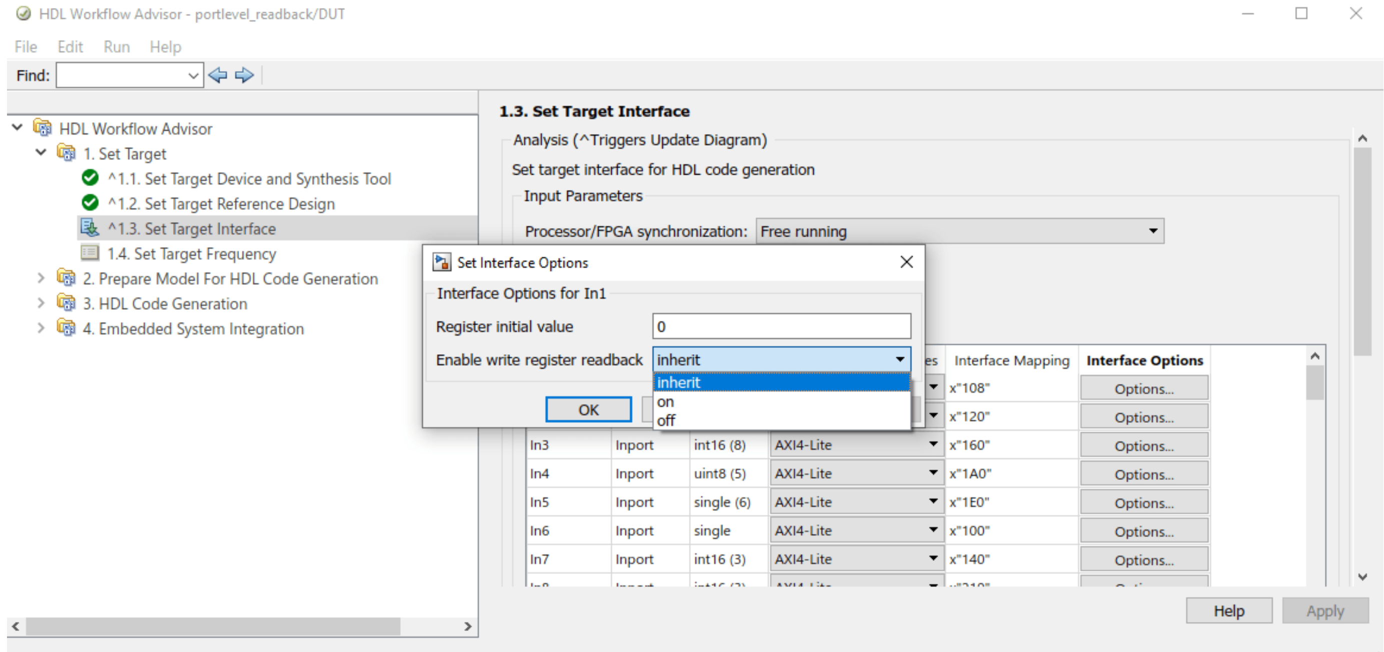
```
open_system('hdlcoder_portlevel_readback');
```


25
Scalar(single)
Vector(int16)
[1:2:16]'
Vector(int16)1
(1:2:10)'
Vector(uint8)
(0:1:5)'
Vector(single)
Scalar(single)1
Vector(int16)2
[1 2 30]'
Vector(int16)3
Scalar(single)6
Vector(single)1
Vector(uint8)1
Scalar(single)9
Scalar(single)10
Scalar(single)7
Scalar(single)2
Scalar(single)4
Scalar(single)3
Vector(uint8)2
Vector(int16)5
Scalar(single)8
Vector(int16)6
Vector(int16)4
Vector(int16)7
Scalar(single)11
Scalar(single)12
Vector(single)2
Scalar(single)5
Scalar(single)15
Vector(int16)10

2. Run all the tasks up until **Set Target Interface** under **Set Target** of the HDL Workflow Advisor

You can turn on the readback for any of the individual ports by clicking the Options under Interface Options.

In the **Set Interface Options** window, you can set the **Enable write register readback** option to `on`, `off` or `inherit`. By default, the **Enable write register readback** option is set to `inherit`, where the readback on that port will inherit from the global readback option.



You can also turn on the readback for any individual port at the MATLAB command line by using `hdlset_param` function in the exported script.

hdlset_param('portlevel_readback/DUT/In1','IOInterfaceOptions',
{'RegisterInitialValue','0','EnableReadback','on'});

Utilization Summary for port level readback and global readback

The table below shows the resource utilization for different cases of port level readback and global readback for the above Simulink model portlevel_readback.

SNO.	Resource	Available	Case:1 Global=OFF, Portlevel=INHERIT	Case:2 Global=OFF, Portlevel=ON (20 ports)	Case:3 Global=ON, Portlevel=INHERIT
1	LUT	218600	6458	9124	14538
2	LUTRAM	70400	2554	3869	7191
3	FF	437200	16152	18227	22582

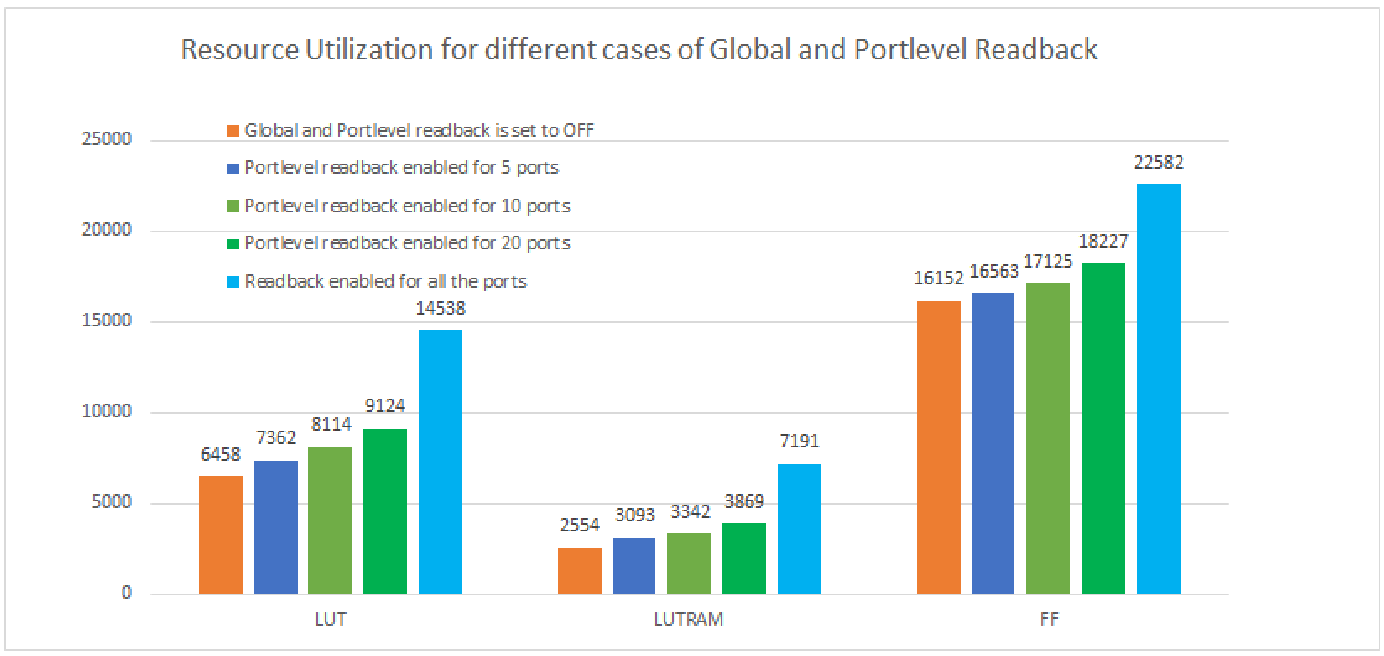
The above table shows the utilization summary for three different cases of portlevel and global readback as described below:

Case1: When global readback is off and portlevel readback is also off. In this case, the global and portlevel readback is set to off for all the ports for the above simulink model.

Case2: When global readback is off and portlevel readback is on. In this case, the portlevel readback is enabled for 20 ports for the above simulink model.

Case3: When global readback is on and portlevel readback is off. In this case, the readback is enabled for all the input ports for the above simulink model.

The below chart shows the utilization summary for different cases of port level readback enabled



Use MATLAB FPGA I/O Host Interface to Communicate with FPGA on Zynq-Based Radio

This example shows how to prototype an FPGA design on a Xilinx Zynq-based radio and communicate with hardware by using MATLAB as the host computer. In this example, you can deploy a waveform transmitter and receiver algorithm on hardware, and then transmit and receive the stream signals through the MATLAB host.

In this example, you to interact with an FPGA design that is running on the hardware. Working with hardware helps you to rapidly prototype designs, verify functionality, tune key parameters, connect to real-world signals, collect data for analysis, and more.

This example highlights how to:

- Generate simple algorithm that runs on hardware.
- Generate HDL IP core and host interface script for your design
- Use host interface script to establish connection with the hardware
- Deploy bitstream and integrate HDL IP core on hardware
- Write input stream signal to your FPGA design
- Read output stream signal from your FPGA design for analysis

Prerequisites

Install and configure these support packages and third-party tools.

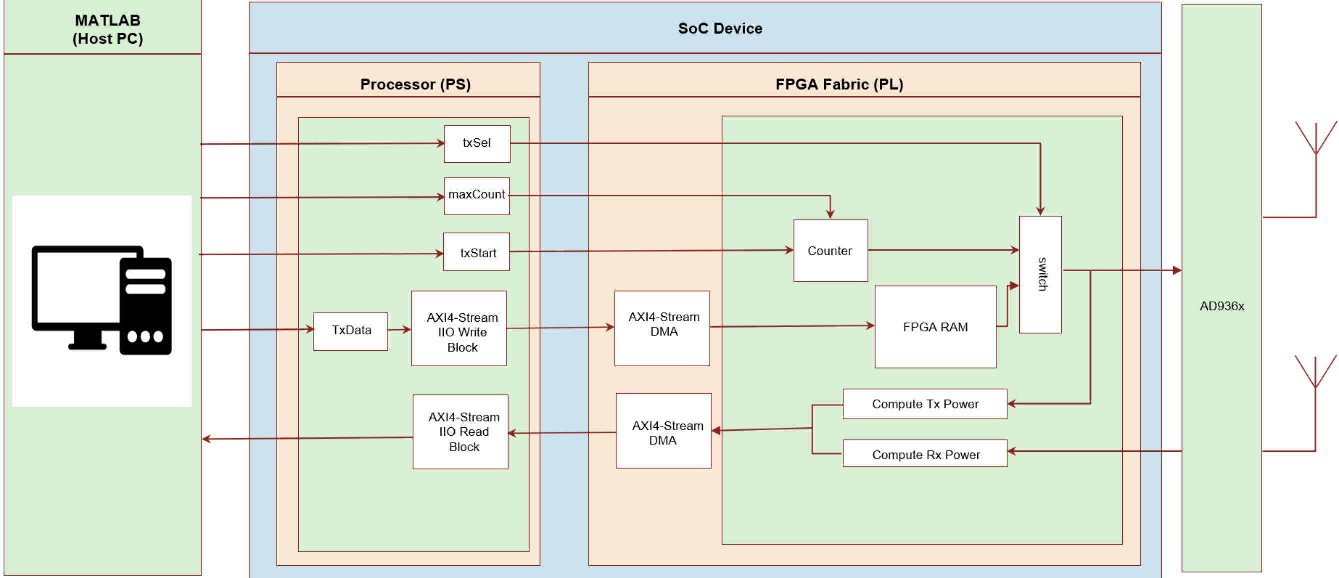
- HDL Coder™ Support Package for Xilinx® FPGA and SoC Devices
- Xilinx Vivado® (version indicated in "HDL Language Support and Supported Third-Party Tools and Hardware".
- MathWorks® firmware image on the board's SD card.

After installing HDL Coder Support Package for Xilinx FPGA and SoC Devices, obtain the Linux image from the install directory. To get required path, enter the command:

```
>> fullfile(matlabshared.supportpkg.getSupportPackageRoot,'3P.instrset','xilinxlinuxbinaries.ins
```

System Architecture

This figure shows the high-level architecture of the system in this example.The host computer communicates to the FPGA through the processing system on the System on Chip (SoC) board. The host computer can also tune parameters by writing to AXI4-Lite registers in the algorithm IP core.



The HDL IP core on the Zynq® programmable logic (PL) combines:

- Waveform transmission from the FPGA RAM to the radio front end
- Power signal computation of both the transmitted and received waveforms
- Triggering logic

The Zynq® processing system (PS) performs these tasks:

- Loading the waveform from the ARM processor to the FPGA RAM for transmission onto SDR
- Receiving the computed power signals from the PL

Unlike the code generation and deployment on the PS, which happens while using software interface model, when you use the `fpga()` function in the MATLAB host, the IIO drivers in the PS transmit and receive the signals.

The waveform transmission and reception is trigger-based and the algorithm computes the power signals of the transmitted and received waveforms. Because the algorithm is trigger-based and deals with packetized data, the example uses AXI4-Stream interface instead of the I/Q stream to manage the data transmission between the ARM processor and the FPGA.

The HDL IP core has these port interfaces:

- AXI4-Stream input ports for writing the waveform into FPGA RAM
- AXI4-Stream output ports for reading the computed power of transmitted and received signals into the PS
- AXI4-Lite input ports for real-time control and adjustment of the design
- AXI DMA ports for data movement of the I/Q samples between the HDL IP core and the PS
- The transmitter and receiver base-band ports for data movement of the I/Q samples between the HDL IP core and the AD936x IP

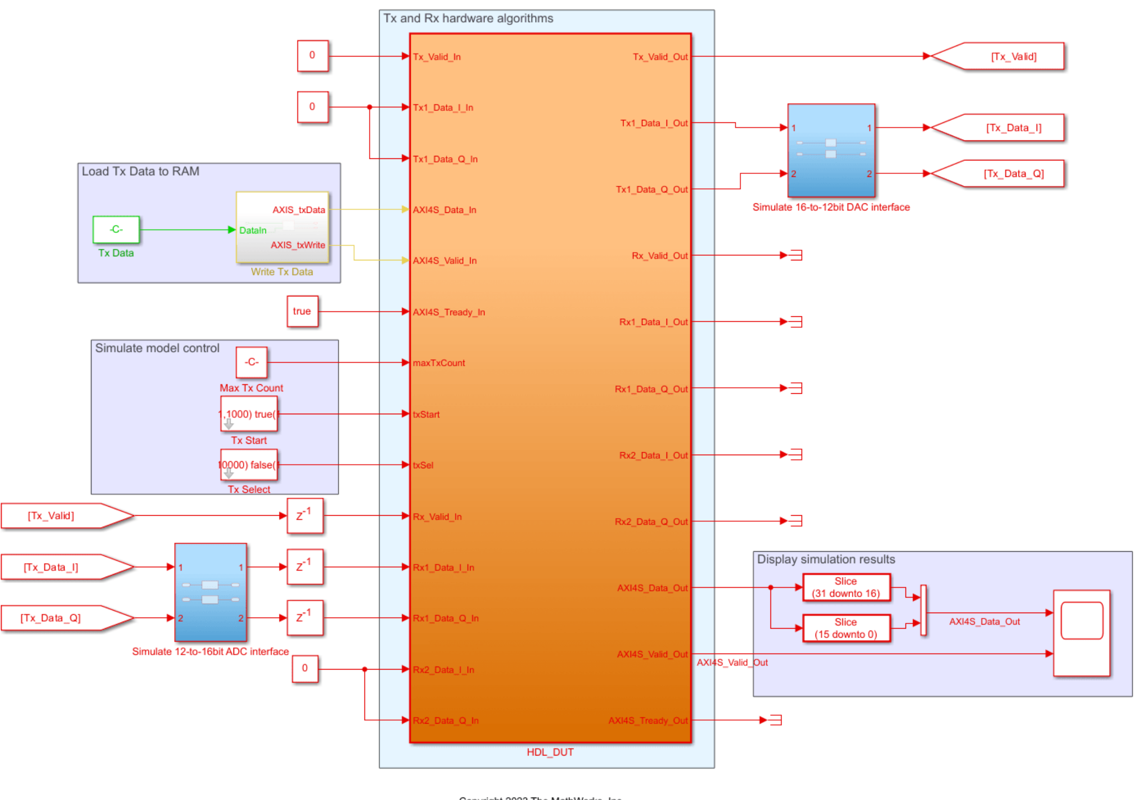
Open the Model

The `hdlcoder_radio_loopback` model is compatible with hardware generation. You can use this model to generate HDL code for the PL and generate a host interface script by using the HDL Workflow Advisor. Using the host interface script, you can connect to the design running on the hardware.

The `HDL_DUT` subsystem models the functionality to be implemented on the PL. The highlighted blocks outside the subsystem represent the control functionality to be implemented on the PS.

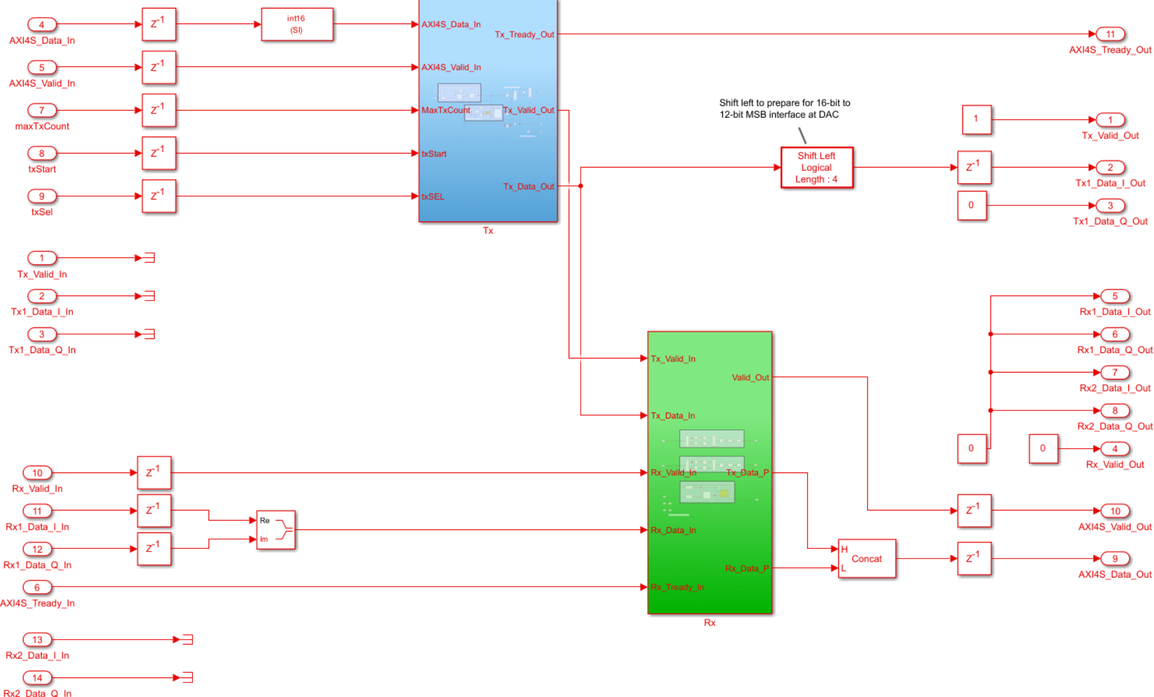
```
modelname = 'hdlcoder_radio_loopback';
load_system(modelname);
open_system(modelname);
```

HW/SW Co-Design AXI4-Stream Radio Loopback Hardware Generation Model



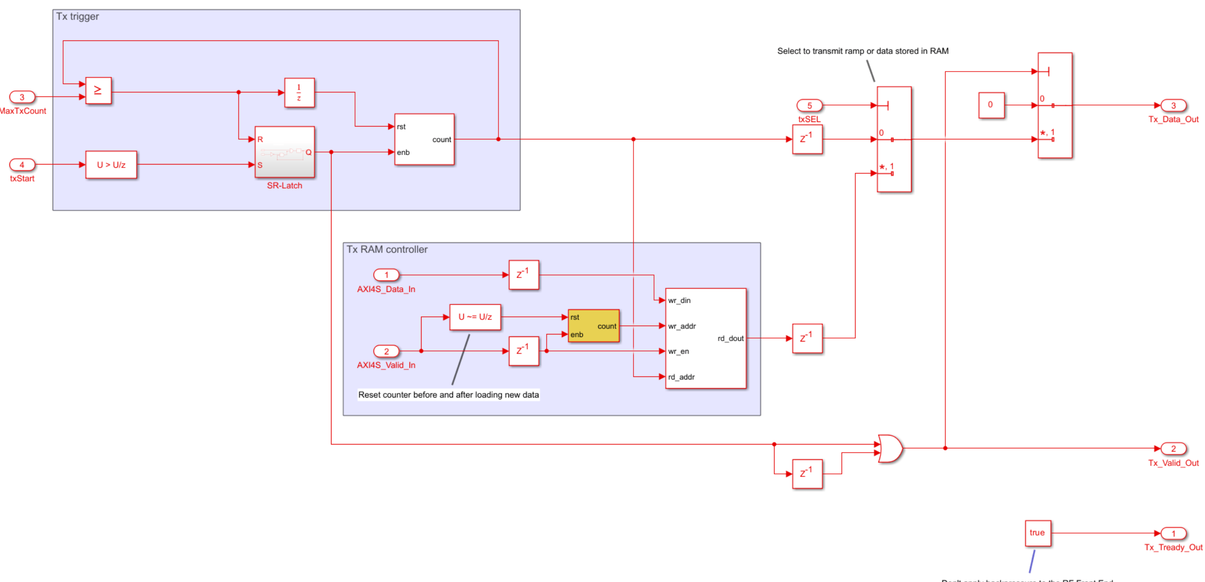
The `Tx` and `Rx` subsystems model the transmitter and receiver logic of the PL algorithm, respectively.

```
currentSubSys = 'HDL_DUT';
open_system([modelname '/' currentSubSys]);
```



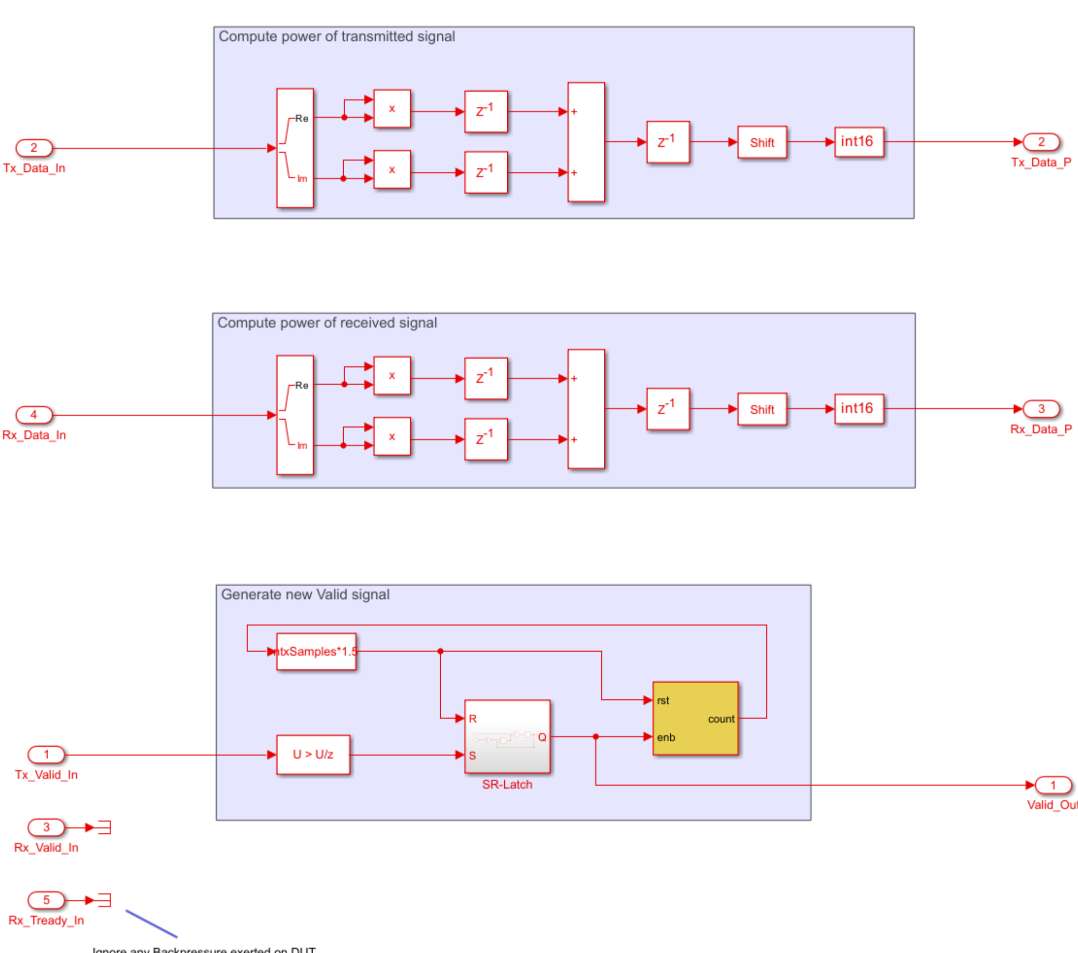
In the Tx subsystem, a valid input frame of AXI4-S data writes to the dual-port block RAM. The maxTxCount and txSel input ports control the transmission of the data in the Tx subsystem, and the txStart input port triggers the transmission of data.

```
currentSubSys = 'HDL_DUT/Tx';
open_system([modelname '/' currentSubSys]);
```



The Rx subsystem computes the power for both the transmitted and received signals as I/Q squared magnitude. To provide padding, the Valid Out Output port outputs a valid period that corresponds to 1.5 times the frame size of the transmitted signal. This padding accounts for the time taken for the transmitted waveform to appear at the receiver.

```
currentSubSys = 'HDL_DUT/Tx';
open_system([modelname '/' currentSubSys]);
```



The HDL_DUT subsystem has designated ports that model AXI4 data paths between the PS and PL.

The AXI4-Lite registers are:

- maxTxCount, which defines the maximum number of transmit signal samples (up to 2047)
- txStart, which triggers a single transmission of maxTxCount samples when set to true
- txSel, which switches between transmitting data stored in the FPGA RAM to the PS or to the ramp signal

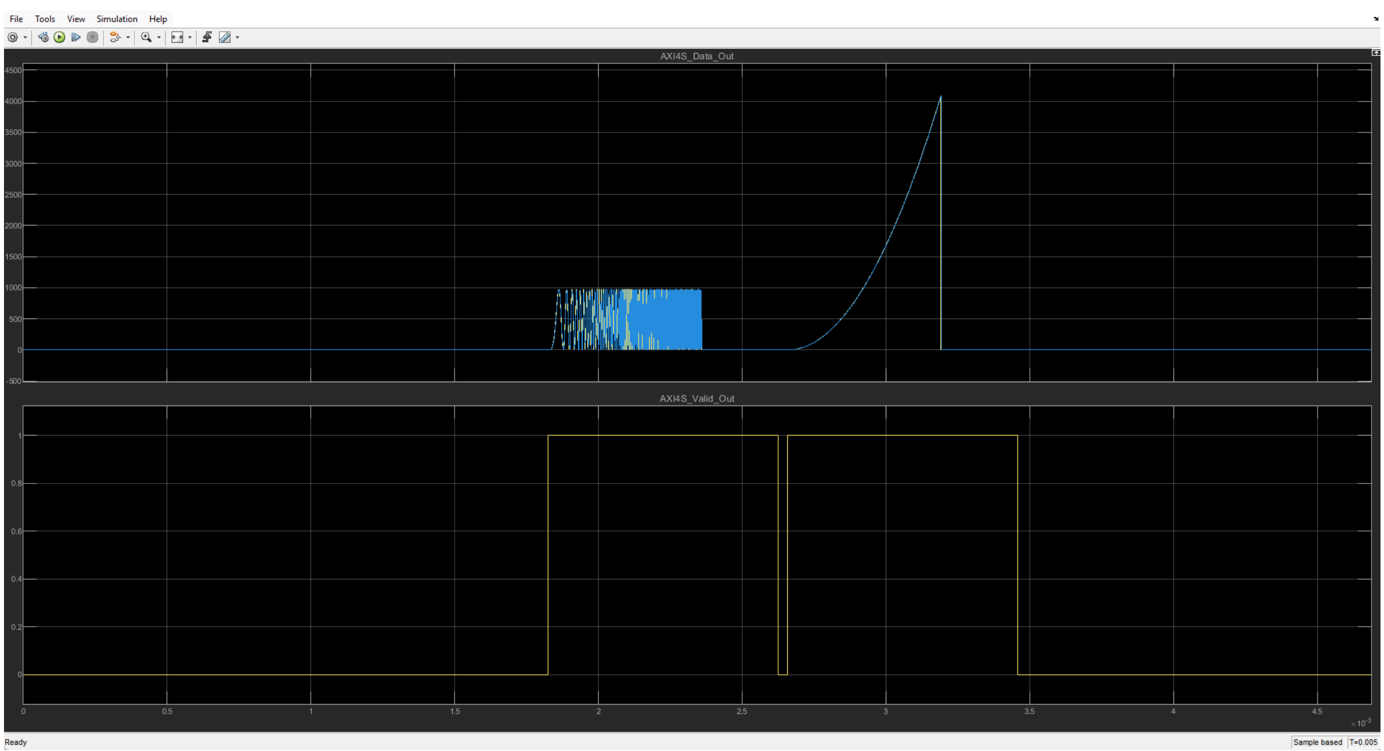
The AXI4-Stream interfaces are:

- `AXI4S_Data_In`, which loads transmission data from the PS to the FPGA RAM
- `AXI4S_Data_Out`, which retrieves computed power of Tx and Rx signals to the PS

Simulate the Model

Simulate this model to confirm its operation. A chirp signal is defined in workspace variable `inpSig`. The Tx Data block reads the signal into the model and scales it for representation by the `int16` fixed-point data type. The `Write Tx Data` block generates AXI4-Stream Data and Valid signals, which writes the signal samples to RAM in the `HDL_DUT` subsystem. The `Max Tx Count`, `Tx Start` and `Tx Select` blocks model the AXI4-Lite control registers.. The values of these blocks trigger a single transmission of the chirp signal, followed by a single transmission of the ramp signal for 2047 samples which loop back over the I/Q interfaces. This transmission triggers a receive at the same time which captures both the transmitted and received signal for a certain length and calculates the power of both signals before transferring them over the AXI4-Stream Data and Valid signals.

The image shows the power output of the Tx and Rx data frames in the Scope block.



After the simulation completes, you can start the process of generating the HDL IP Core, integrating the core with the SDR reference design, and generating the host interface script.

Generate the IP Core

First set up the Xilinx tool chain by invoking the `hdlsetuptoolpath` function. For example:

```
>> hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2023.1\bin\vivado.ba
```

Register the custom boards and compile ADI IP path to MATLAB path for creating a Vivado project by entering these commands:

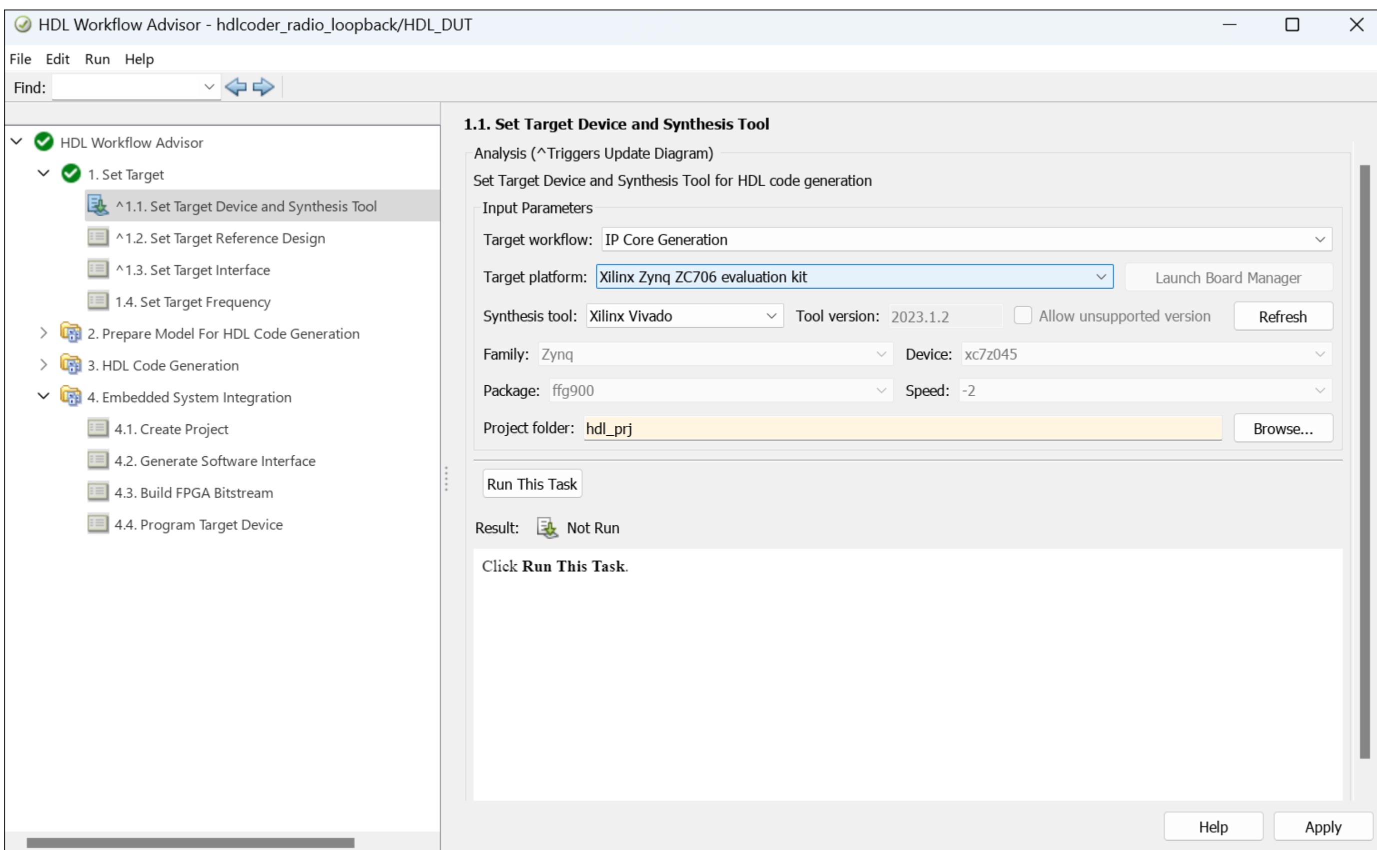
```
hdlcoder_amd_examples_root
addpath(fullfile(hdlcoder_amd_examples_root,'ZC706'))
```

```
addpath(fullfile(hdlcoder_amd_examples_root,'ipcore'))
hdlcxilinx.internal.hwsetup.compileADIIP
```

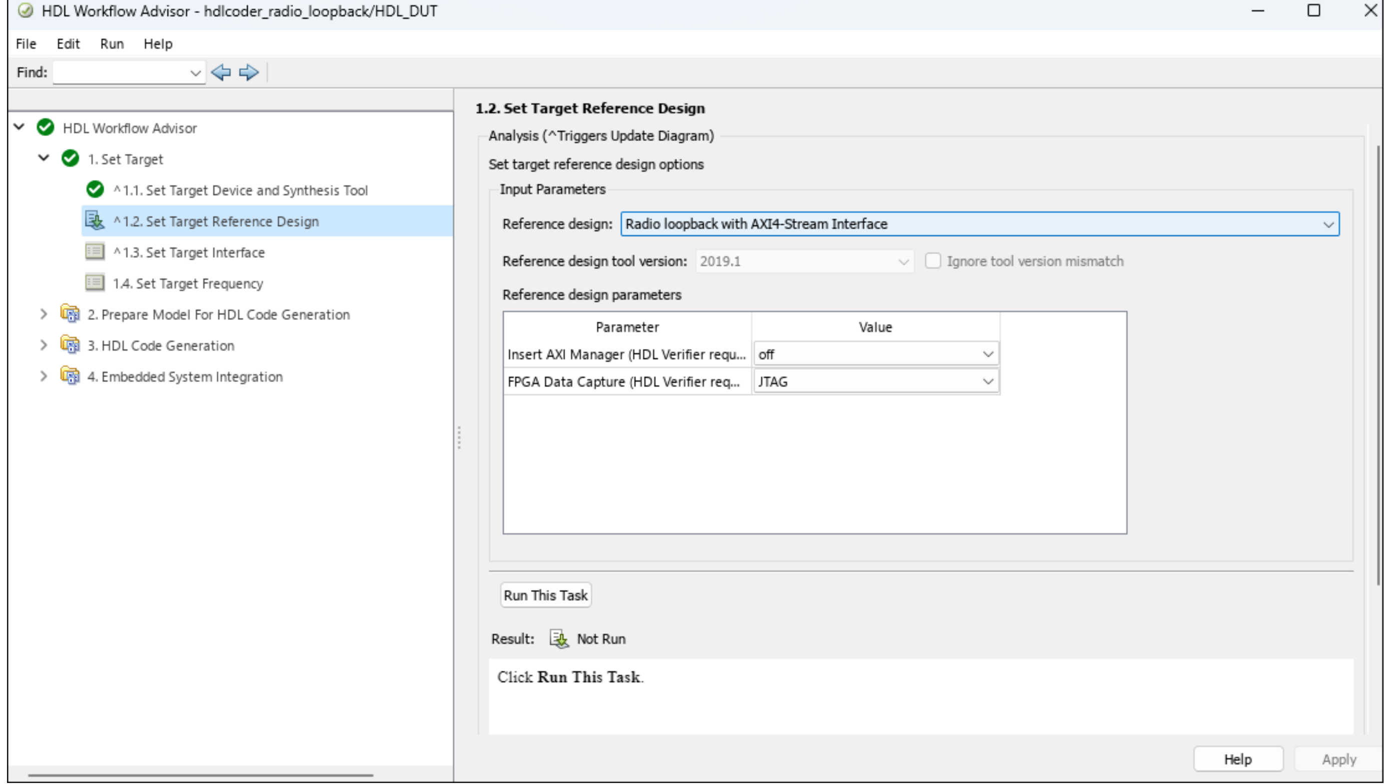
Running the compilation for first time in MATLAB takes several minutes to complete.

Right-click the `HDL_DUT` subsystem and select **HDL Code > HDL Workflow Advisor**.

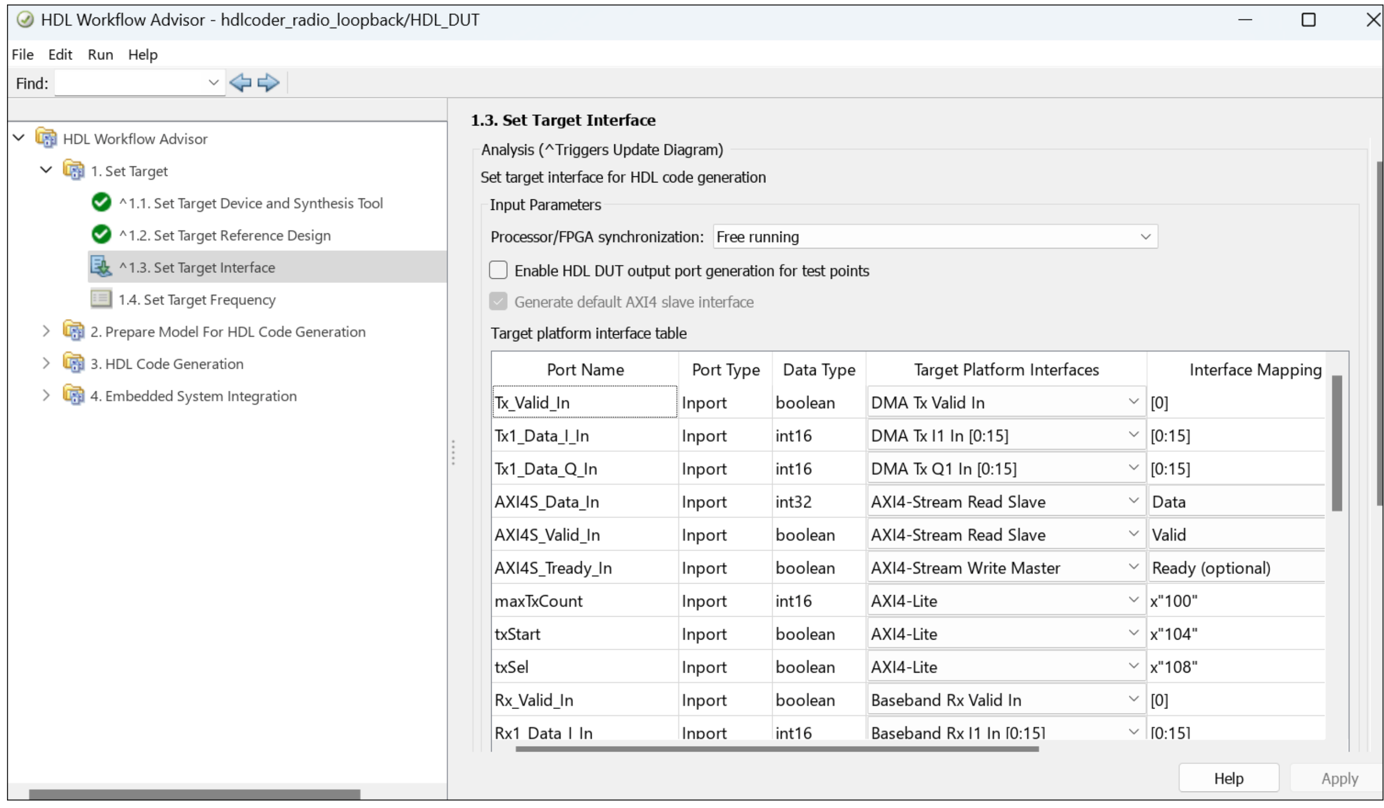
1. In step 1.1, set **Target workflow** to `IP Core Generation` and **Target platform** to `Xilinx Zynq ZC706 evaluation kit`.

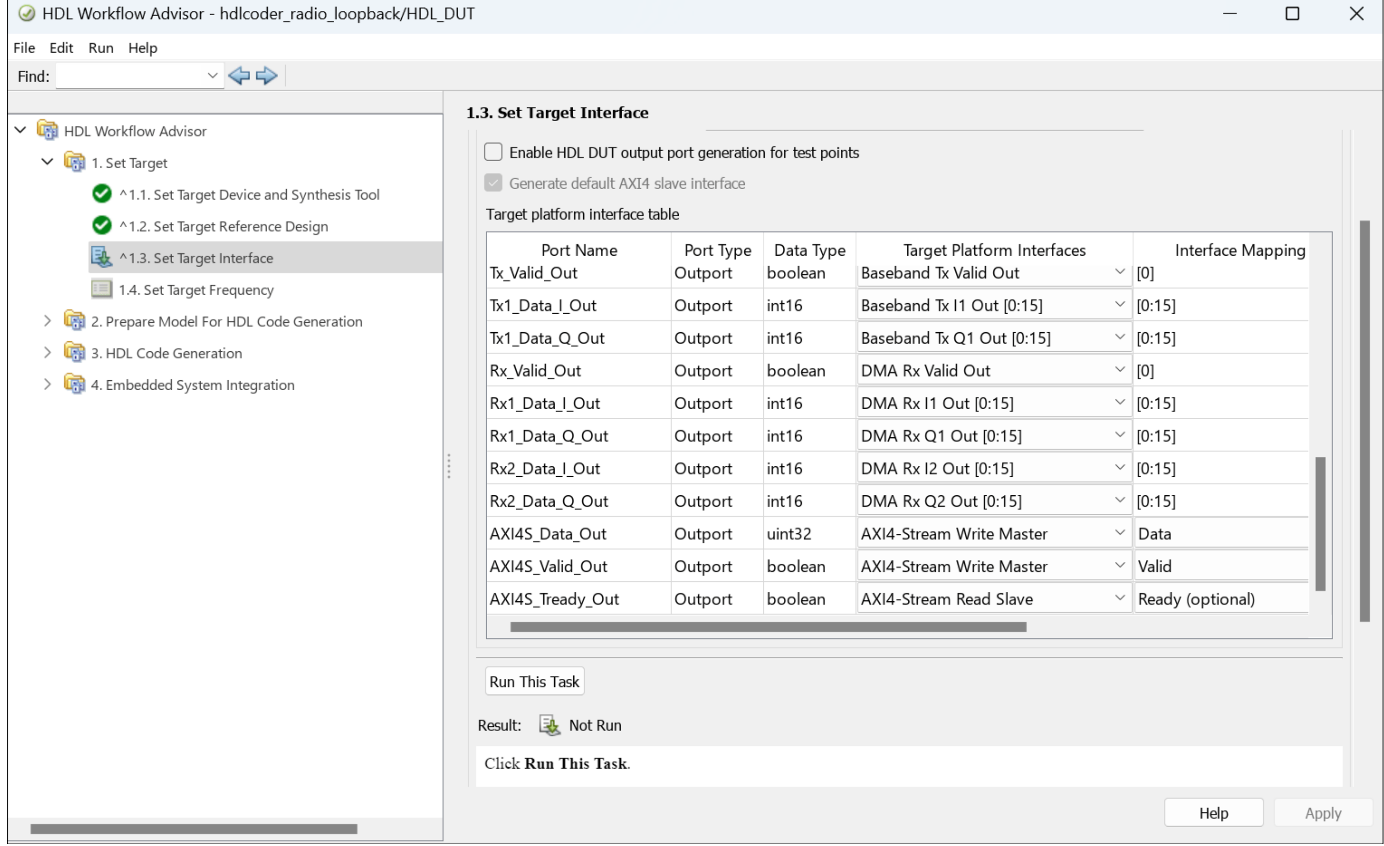


2. In step 1.2, set Reference design to `Radio loopback with AXI4-Stream Interface`. For this example, you can use the default reference design parameters.



3. In step 1.3, review the data in the **Target platform interface table** section. The Workflow Advisor automatically maps the DUT signals to the interface signals in the reference design.

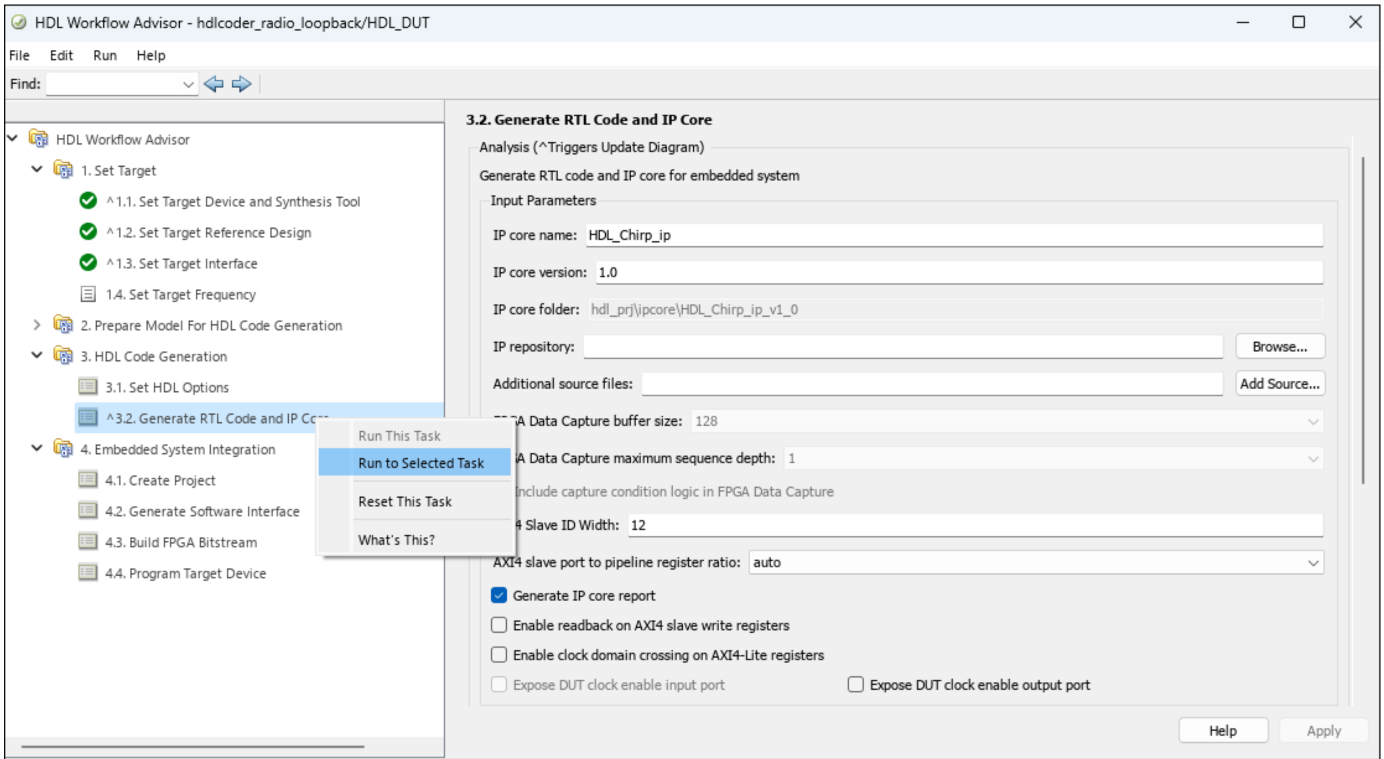

HDL Workflow Advisor - hdlcoder_radio_loopback/HDL_DUT
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Reference Design
^1.3. Set Target Interface
1.4. Set Target Frequency
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. Embedded System Integration
1.3. Set Target Interface
Analysis (^Triggers Update Diagram)
Set target interface for HDL code generation
Input Parameters
Processor/FPGA synchronization: Free running
Enable HDL DUT output port generation for test points
Generate default AXI4 slave interface
Target platform interface table
Port Name | Port Type | Data Type | Target Platform Interfaces | Interface Mapping
Tx_Valid_In | Inport | boolean | DMA Tx Valid In | [0]
Tx1_Data_I_In | Inport | int16 | DMA Tx I1 In [0:15] | [0:15]
Tx1_Data_Q_In | Inport | int16 | DMA Tx Q1 In [0:15] | [0:15]
AXI4S_Data_In | Inport | int32 | AXI4-Stream Read Slave | Data
AXI4S_Valid_In | Inport | boolean | AXI4-Stream Read Slave | Valid
AXI4S_Tready_In | Inport | boolean | AXI4-Stream Write Master | Ready (optional)
maxTxCount | Inport | int16 | AXI4-Lite | x"100"
txStart | Inport | boolean | AXI4-Lite | x"104"
txSel | Inport | boolean | AXI4-Lite | x"108"
Rx_Valid_In | Inport | boolean | Baseband Rx Valid In | [0]
Rx1_Data_I_In | Inport | int16 | Baseband Rx I1 In [0:15] | [0:15]
Help
Apply



4. In step 1.4, set the T**arget Frequency (MHz)** parameter to the DUT synthesis frequency.. The DUT synthesis frequency depends on the baseband sampling rate of the system. In this example, the sample rate is 3.84 MHz, so a synthesis frequency of at least 4 MHz is sufficient.

5. Right-click step 2 and click **Run to Selected Task** to perform the design checks.

6. Right-click step 3 and click **Run to Selected Task** to generate the HDL code for the IP core.



Generate Host Interface Script

Next, you use the generated host interface script and modify it to:

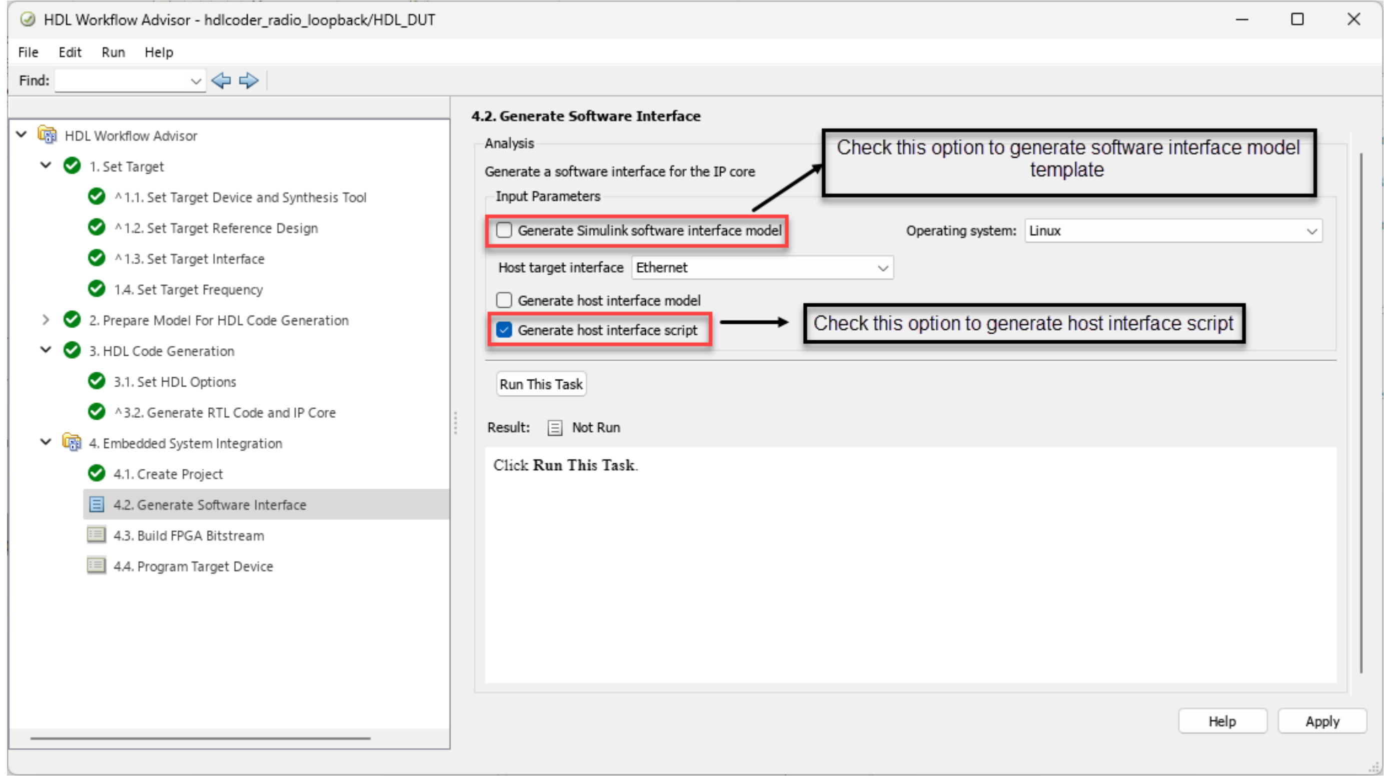
- Create a hardware object to establish a connection to your FPGA.
- Deploy the bitstream on hardware.
- Write input control signals and stream signals to the PS.
- Read the output stream signals from the PS.

After you use the fpga function to read the stream signal from the PS to the MATLAB host, you unpack the data to visualize the behaviour of the design.

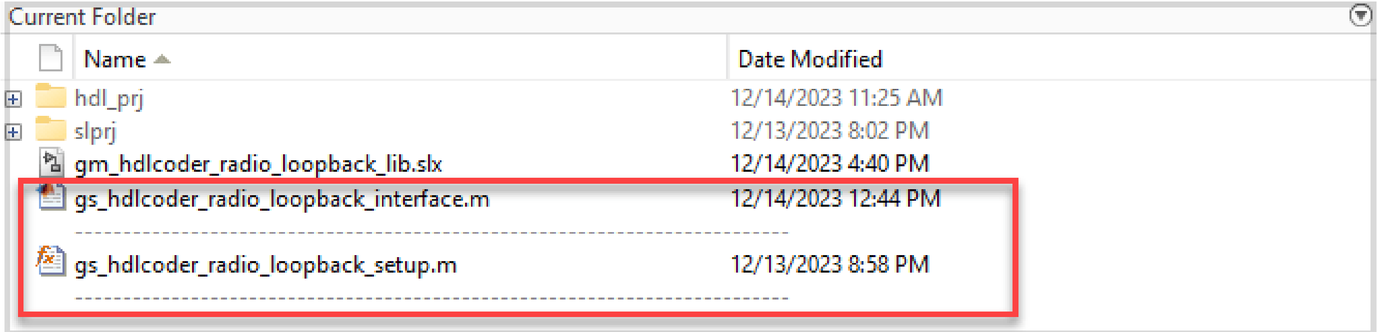
To generate a host computer interface to the IP core and deploy the design to the target hardware board:

1. In step 4.1, click **Run This Task.**. This task inserts the generated IP core for the FPGA algorithm into the reference design and creates the system shown in the system architecture diagram.

2. In step 4.2, select **Generate host interface script** and clear the **Generate Simulink software interface model**. Then, click **Run this task**.

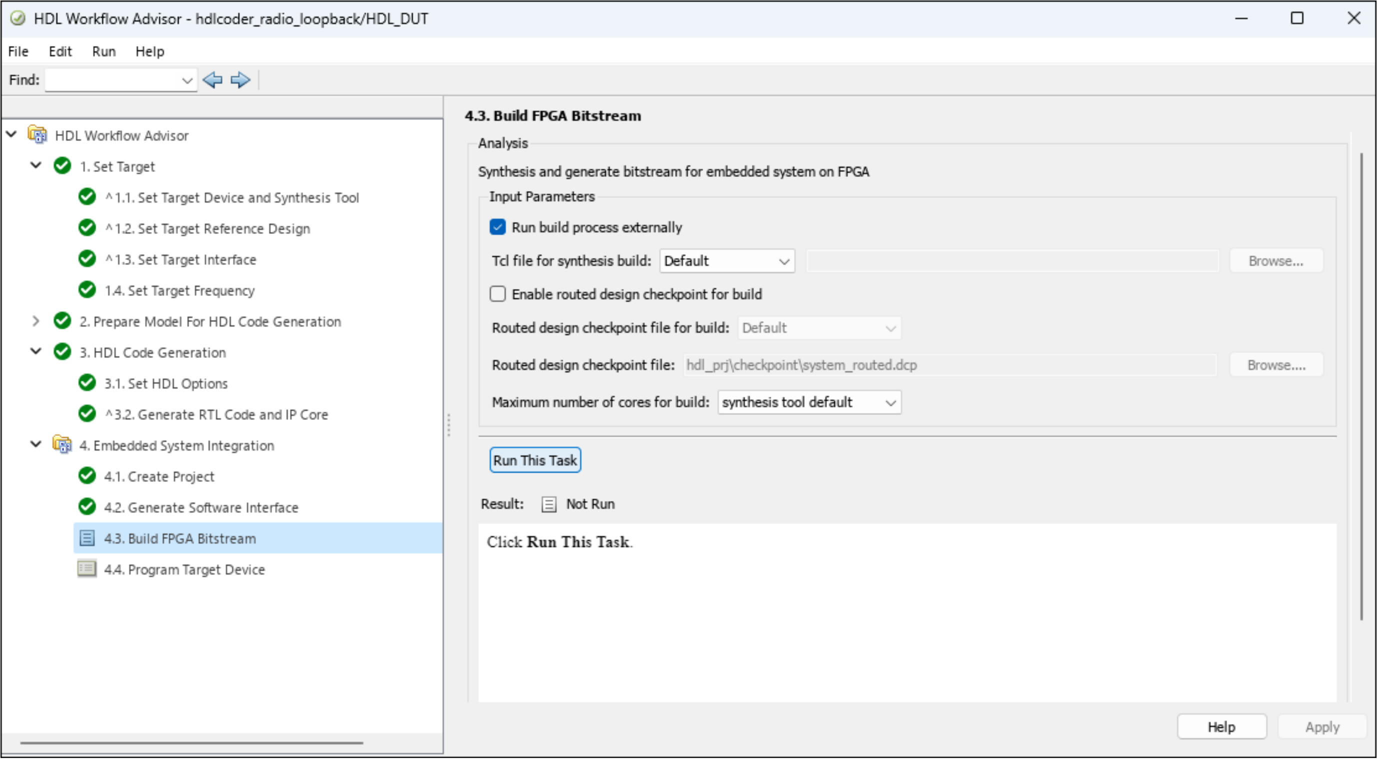


The HDL Workflow Advisor generates two MATLAB files in your current folder. You can use these files to prototype your generated IP core directly from MATLAB.



Generate Bitstream

Step 4.3 generates a bitstream for the PL. You can execute this step in an external shell by selecting **Run build process externally**. This selection allows you to continue using MATLAB while building the FPGA image. Next, click **Run This Task**. After the HDL Workflow Advisor completes the product checks, it marks step 4.3 with a green check mark. You must wait until the external shell displays a successful bitstream build before continuing on to the next step.



When the bistream is ready, you can load it on the hardware board by using the host interface script.

Interact with the FPGA Design from the Host Computer

Interact with the FPGA design by reading and writing data through MATLAB on the host computer.

Open the generated script file by entering:

```
open gs_hdlcoder_radio_loopback_interface.m
```

Using host interface script, you can:

- Program the FPGA hardware with the generated bitstream and configure the processor with the device tree, which requires board IP address as well.
- Create an `fpga` hardware object, which represents a connection to the FPGA on your hardware board.
- Configure the `fpga` object with the desired hardware interfaces and ports from your DUT algorithm.
- Read and write data to the DUT to exercise your algorithm on the hardware.
- Release the hardware board from the current MATLAB session.

```
%-----------------------------------------------------------------------
% Host Interface Script
%
% Generated with MATLAB 24.1 (R2024a) at 16:54:44 on 13/12/2023.
% This script was created for the IP Core generated from design 'hdlcoder_radio_loopback'.
%
% Use this script to access DUT ports in the design that were mapped to compatible IP core interfaces.
% You can write to input ports in the design and read from output ports directly from MATLAB.
% To write to input ports, use the "writePort" command and specify the port name and input data. The input data will be cast to the DUT port's data type before writing.
% To read from output ports, use the "readPort" command and specify the port name. The output data will be returned with the same data type as the DUT port.
% Use the "release" command to release MATLAB's control of the hardware resources.
%-----------------------------------------------------------------------

%% Program FPGA
% Uncomment the lines below to program FPGA hardware with the designated bitstream and configure the processor with the corresponding devicetree.
% MATLAB will connect to the board with an SSH connection to program the FPGA.
% If you need to change login parameters for your board, using the following syntax:
% hProcessor = xilinxsoc(ipAddress, username, password);
hProcessor = xilinxsoc();
% programFPGA(hProcessor, "hdl_prj\vivado_ip_prj\vivado_prj.runs\impl_1\system_top.bit", "devicetree_fmcomms2_axis.dtb");

%% Create fpga object
hFPGA = fpga(hProcessor);

%% Setup fpga object
% This function configures the "fpga" object with the same interfaces as the generated IP core
gs_hdlcoder_radio_loopback_setup(hFPGA);

%% Write/read DUT ports
% Uncomment the following lines to write/read DUT ports in the generated IP Core.
% Update the example data in the write commands with meaningful data to write to the DUT.
%% AXI4-Lite
% writePort(hFPGA, "maxTxCount", zeros([1 1]));
% writePort(hFPGA, "txStart", zeros([1 1]));
% writePort(hFPGA, "txSel", zeros([1 1]));

%% AXI4-Stream Write
% data_AXI4S_Data_Out = readPort(hFPGA, "AXI4S_Data_Out");

%% AXI4-Stream Read
% writePort(hFPGA, "AXI4S_Data_In", zeros([1024 1]));

%% Release hardware resources
release(hFPGA);

```

1 2 3 4 5

You can modify the interface script depending on the algorithm implemented in your Simulink model.

Open the generated setup function by entering:

open `gs_hdlcoder_radio_loopback_setup.m`

This function configures the `fpga` hardware object with the same ports and interfaces specified in task 1.3. **Set Target Interface**.

- Under **AXI4-Lite** section, DUT input ports such as `maxTCount,txStart,` and `txSel` are mapped to AXI registers because the table in step 1.3 declared these ports as AXI4-Lite.
- Similarly, the DUT input port `AXI4S_Data_In` and DUT output port `AXI4S_Data_Out` are mapped to AXI4-Stream Read and AXI4-Stream Write, respectively.

```
1  function gs_hdlcoder_radio_loopback_setup(hFPGA)
2  %--------------------------------------------------------------------------
3  % Host Interface Script Setup
4  %
5  % Generated with MATLAB 24.1 (R2024a) at 20:23:38 on 13/12/2023.
6  % This function was created for the IP Core generated from design 'hdlcoder_radio_loopback'.
7  %
8  % Run this function on an "fpga" object to configure it with the same interfaces as the generated IP core.
9  %--------------------------------------------------------------------------
10 %% AXI4-Lite
11 addAXI4SlaveInterface(hFPGA, ...
12     "InterfaceID", "AXI4-Lite", ...
13     "BaseAddress", 0x43C00000, ...
14     "AddressRange", 0x10000);
15
16 DUTPort_maxTxCount = hdlcoder.DUTPort("maxTxCount", ...
17     "Direction", "IN", ...
18     "DataType", "int16", ...
19     "IsComplex", false, ...
20     "Dimension", [1 1], ...
21     "IOInterface", "AXI4-Lite", ...
22     "IOInterfaceMapping", "0x100");
23
24 DUTPort_txStart = hdlcoder.DUTPort("txStart", ...
25     "Direction", "IN", ...
26     "DataType", "logical", ...
27     "IsComplex", false, ...
28     "Dimension", [1 1], ...
29     "IOInterface", "AXI4-Lite", ...
30     "IOInterfaceMapping", "0x104");
31
32 DUTPort_txSel = hdlcoder.DUTPort("txSel", ...
33     "Direction", "IN", ...
34     "DataType", "logical", ...
35     "IsComplex", false, ...
36     "Dimension", [1 1], ...
37     "IOInterface", "AXI4-Lite", ...
38     "IOInterfaceMapping", "0x108");
39
40 mapPort(hFPGA, [DUTPort_maxTxCount, DUTPort_txStart, DUTPort_txSel]);
41
42 %% AXI4-Stream Write
43 addAXI4StreamInterface(hFPGA, ...
44     "InterfaceID", "AXI4-Stream Write", ...
45     "WriteEnable", false, ...
46     "ReadEnable", true, ...
47     "ReadDataWidth", 32, ...
48     "ReadFrameLength", 1024);
49
50 DUTPort_AXI4S_Data_Out = hdlcoder.DUTPort("AXI4S_Data_Out", ...
51     "Direction", "OUT", ...
52     "DataType", "uint32", ...
53     "IsComplex", false, ...
54     "Dimension", [1 1], ...
55     "IOInterface", "AXI4-Stream Write");
56
57 mapPort(hFPGA, [DUTPort_AXI4S_Data_Out]);
58
59 %% AXI4-Stream Read
60 addAXI4StreamInterface(hFPGA, ...
61     "InterfaceID", "AXI4-Stream Read", ...
62     "WriteEnable", true, ...
63     "WriteDataWidth", 32, ...
64     "WriteFrameLength", 1024, ...
65     "ReadEnable", false);
66
67 DUTPort_AXI4S_Data_In = hdlcoder.DUTPort("AXI4S_Data_In", ...
68     "Direction", "IN", ...
69     "DataType", "int32", ...
70     "IsComplex", false, ...
71     "Dimension", [1 1], ...
72     "IOInterface", "AXI4-Stream Read");
73
74 mapPort(hFPGA, [DUTPort_AXI4S_Data_In]);
75
76 end
```

DUT input ports "maxTxCount", "txStart", "txSel" mapped to AXI4-Lite register

DUT output stream port "AXI4S_Data_Out" mapped to AXI4-Stream write

DUT input stream port "AXI4S_Data_In" mapped to AXI4-Stream Read

]

You can also modify the setup script depending on the algorithm you implement in your Simulink model. You can alter parameters, such as `WriteFrameLength` and `ReadFrameLength`, as shown in this example. In this setup script, the values for `WriteFrameLength` and `ReadFrameLength` have been adjusted from 1024 to 2048. However, you cannot change parameters like `IOInterface` because they have been defined to work as intended.

Configure the Software Defined Radio (SDR)

This example uses the System Object `hdlcoder.sdr` to configure the AD9361/AD9364 transmitter and receiver, respectively. The transmitter and receiver do not send data from the PS to the PL but configuring them initializes RF parameters and enables transmit and receive data paths.

```
%% Configure SDR
%
SDRTxRx = hdlcoder.sdr('AD936x', ...
    IPAddress = IPAddress,...
    CenterFrequency = Fc,...
    ChannelMapping = [1 2],...
    BasebandSampleRate = Fs,...
    TxGain = txGain,...
    RxGain = rxGain,...
    SamplesPerFrame = FrameSize);
setup(SDRTxRx);
```

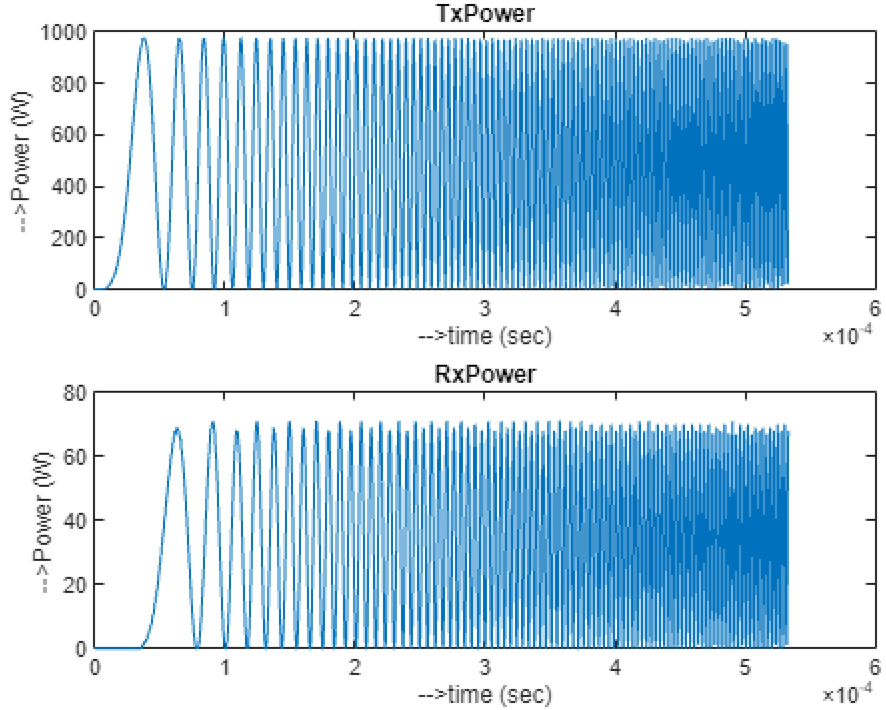
Before performing any write or read operations using host interface script, the configure the SDR. If not, the SDR can take erroneous values and exhibit unexpected behaviour.

Run Host Interface Script on the Target

Modify the generated host script as shown in this code. First, initialize all the variables in the script and then load the generated bitstream on the hardware board. The `sel` variable selects between the internally generated ramp signal and externally passing signal. After the bitstream loads, you can verify the SDR connection to hardware.

```
%% Variables Initialization

IPAddress = '192.168.3.2'; %Hardware board IP address
Fc = 2.4e9;    %Center Frequency
Fs = 3840000;  % Sampling Frequency
nsamp = 2048;  %Number of time samples
ntxSampleS = 2048;     %Number of transmit signal samples
txGain = -10;  %SDR transmitter gain
rxGain = 1;    %SDR receiver gain
FrameSize = 2048;
t = (0:nsamp)/Fs;
sel = input('Enter 0 to see the internal ramp and 1 for externally passing signal:  ') %#ok<NOPTS

sel = 1

hProcessor = xilinxsoc(IPAddress,'root','root');
programFPGA(hProcessor, "hdl_prj\vivado_ip_prj\vivado_prj.runs\impl_1\system_top.bit", "devicetre

### Programming FPGA device on Xilinx Zynq ZC706 hardware board at 192.168.3.2...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream system_top.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/system_top.bit
# Copying Devicetree devicetree_fmcomms2_axis.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_fmcomms2_axis.dtb
# Set up boot for Reference Design: ''
### Rebooting Xilinx Zynq ZC706 at 192.168.3.2...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.3.2...
### Connection successful

%% Create fpga object

hFPGA = fpga(hProcessor);

%% Configure SDR

SDRTxRx = hdlcoder.sdr('AD936x', ...
    IPAddress = IPAddress,...
    CenterFrequency = Fc,...
    ChannelMapping = [1 2],...
    BasebandSampleRate = Fs,...
    TxGain = txGain,...
    RxGain = rxGain,...
    SamplesPerFrame = FrameSize);
setup(SDRTxRx);

## Establishing connection to hardware. This process can take several seconds.
## Establishing connection to hardware. This process can take several seconds.
```

```
%% Setup fpga object
% This function configures the "fpga" object with the same interfaces as the generated IP core

gs_hdlcoder_radio_loopback_setup(hFPGA);

%% AXI4-Stream Read

%writePort(hFPGA, "AXI4S_Data_In",int32(1000*sin(2*pi*20e3*(0:2047)/Fs)));
writePort(hFPGA, "AXI4S_Data_In",int32(1000*chirp(t(1:2048),0,t(2048),Fs/20,[],90)));
%writePort(hFPGA, "AXI4S_Data_In",int32(1000*sawtooth(2*pi*20e3*(0:2047)/Fs)));

%% AXI4 Lite

if(sel == 0)
 writePort(hFPGA, "txSel", zeros([1 1]));
else
 writePort(hFPGA, "txSel", ones([1 1]));
end
 writePort(hFPGA, "maxTxCount", int16(ntxSampleS-1));
 writePort(hFPGA, "txStart", ones([1 1]));

%% AXI4-Stream Write

data_AXI4S_Data_Out = readPort(hFPGA, "AXI4S_Data_Out");

%% Data Unpacking

dataReceived = typecast(data_AXI4S_Data_Out,'int16');
rxPower = dataReceived(1:2:4096);
txPower = dataReceived(2:2:4096);
subplot(2,1,1)
%figure
plot(t(1:end-1),txPower)
title('TxPower');
xlabel('-->time (sec)')
ylabel('-->Power (W)')
subplot(2,1,2)
%figure
plot(t(1:end-1),rxPower)
title('RxPower');
xlabel('-->time (sec)')
ylabel('-->Power (W)')
```




```
release(hFPGA);
delete(hFPGA);
```

After the SDR configuration completes, the `writePort` function passes input stream signal and control signals and the data coming from the DUT output port `AXI4S_Data_Out` is read into `data_AXI4S_Data_Out` through `readPort.`

The data type of the `AXI4S_Data_Out` is `int32. The` MSB 16 bits contains the information pertaining to TxPower. LSB 16 bits contains the information of RxPower. Hence data unpacking is required to separate TxPower and RxPower.

Finally, the `release` command releases the `fpga` handle from the MATLAB session and the `delete` command deletes the variable completely from the MATLAB workspace.

See Also

`xilinxsoc` | `readPort` | `writePort`

Related Examples

- "Inspect the Written Values of AXI4 Slave Registers by Using the Readback Methods" on page 40-51
- "Getting Started with Targeting Xilinx Versal Adaptive SoC Platform" on page 39-265
- "Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands" on page 39-241

Multirate IP Core Generation

This example shows HDL Coder™ supports designs with multiple sample rates when you run the `IP Core Generation` workflow.

If you are only using AXI4 slave interfaces such as AXI4 or AXI4-Lite, and when you use `Free running` for **Processor/FPGA Synchronization**, you can use multiple sample rates in your design without restrictions.

When you map the interface ports to AXI4-Stream, AXI4-Stream Video, or AXI4 Master interfaces, to use multiple sample rates, make sure that the DUT ports that map to the AXI4 interfaces run at the fastest rate of the design after HDL code generation.

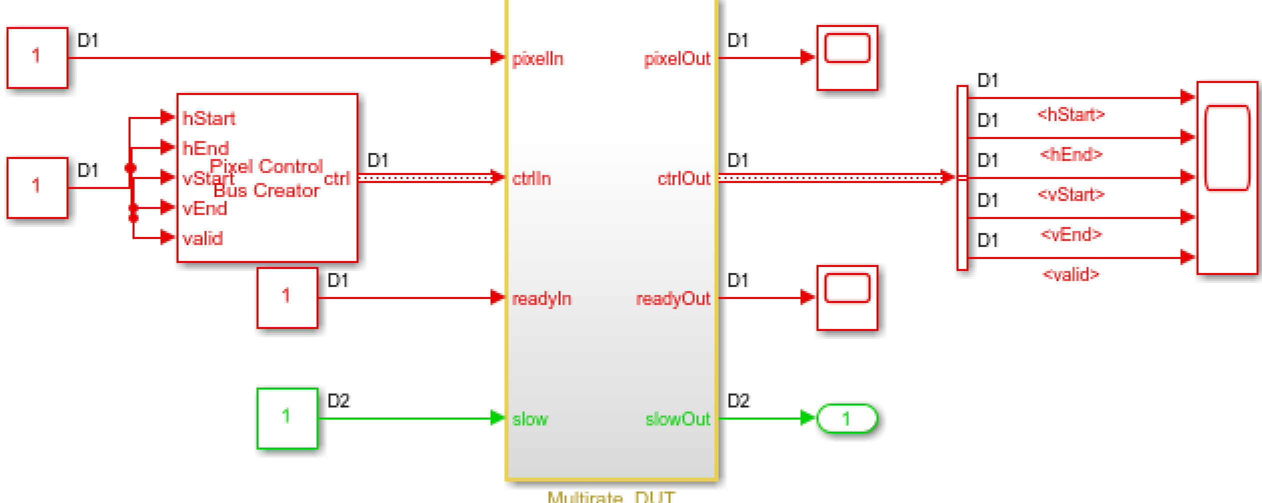
These examples illustrate how you can model your design with multiple sample rates when using AXI4-Stream, AXI4-Stream Video, or AXI4-Master Master interfaces.

Run Part of Design at Slower Rate

You can run part of the design at a slower rate while making sure that the DUT ports that map to the interface run at the fastest rate. This example illustrates mapping to AXI4-Stream Video interfaces but you can map to AXI4-Stream or AXI4 Master interfaces by using this approach.

For an example, open the model `hdlcoder_axi_video_multirate`.

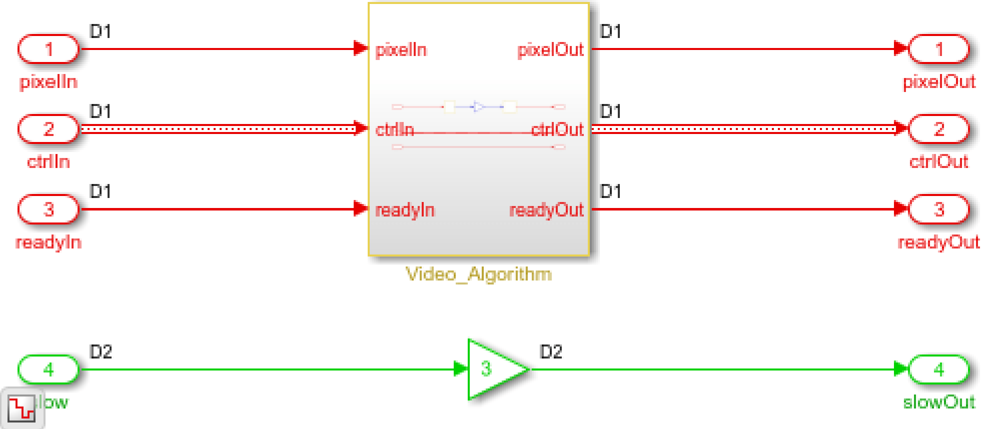
```
load_system('hdlcoder_axi_video_multirate')
set_param('hdlcoder_axi_video_multirate','SimulationCommand','update')
open_system('hdlcoder_axi_video_multirate')
```



Copyright 2018-2021 The MathWorks, Inc.

In this model, the DUT ports corresponding to inputs and outputs of the `Video_Algorithm` run at the fastest rate.

```
open_system('hdlcoder_axi_video_multirate/Multirate_DUT')
```



These ports can therefore map to AXI4-Stream Video interfaces. Part of the design running outside this algorithm corresponding to input `slow` and output `slowOut` running at a slower rate can map to AXI4 or AXI4-Lite interfaces. This figure shows an example of the target platform interface mapping for this model.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
pixelIn	Inport	uint16	AXI4-Stream Video Slave	Pixel Data
ctrlIn	Inport	bus	AXI4-Stream Video Slave	Pixel Control Bus
readyIn	Inport	boolean	AXI4-Stream Video Master	Ready (optional)
slow	Inport	uint16	AXI4-Lite	x"100"
pixelOut	Outport	uint16	AXI4-Stream Video Master	Pixel Data
ctrlOut	Outport	bus	AXI4-Stream Video Master	Pixel Control Bus
readyOut	Outport	boolean	AXI4-Stream Video Slave	Ready (optional)
slowOut	Outport	uint16	AXI4-Lite	x"104"

Note: To use the Pixel Control Bus Creator and Pixel Control Bus Selector blocks, you must have Vision HDL Toolbox™ installed. If you do not have Vision HDL Toolbox, use Bus Creator and Bus Selector blocks instead.

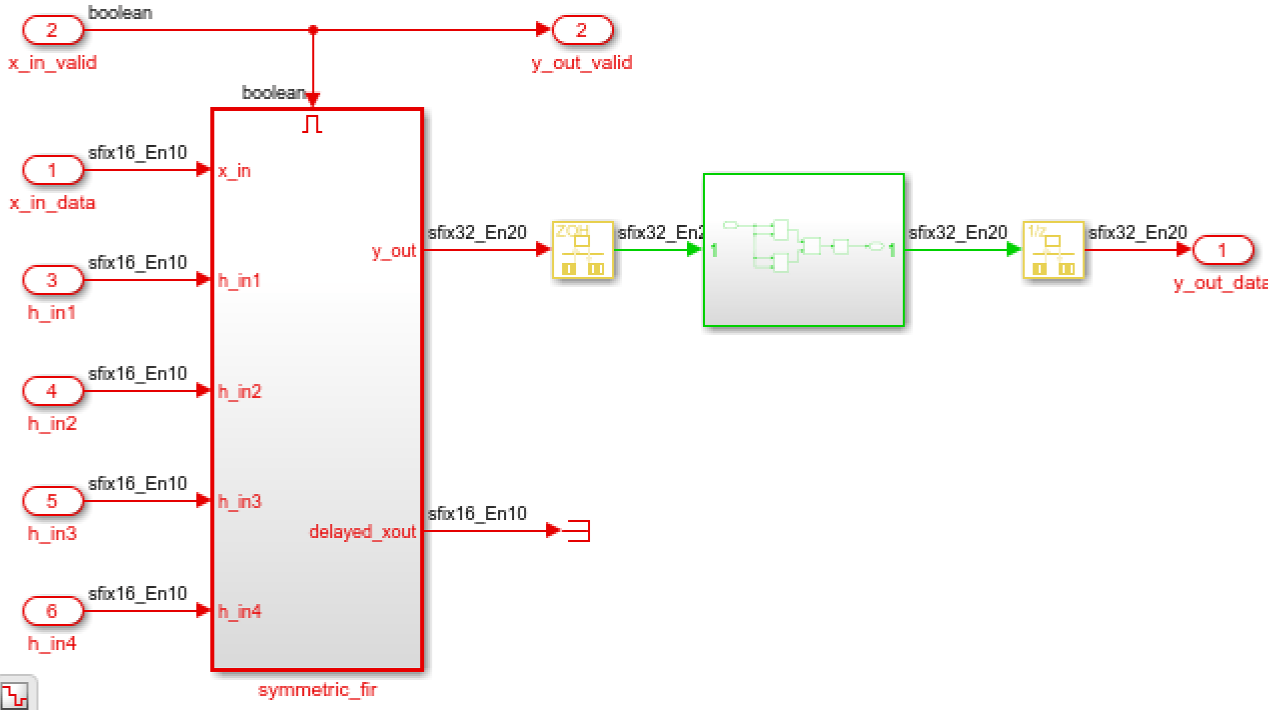
See also “Model Design for AXI4-Stream Video Interface Generation” on page 40-118.

Apply Optimizations to Part of Design Running at Slow Rate

With multirate support, you can apply optimizations such as resource sharing to a part of the design running at a slower rate. Make sure that the optimizations do not introduce a faster rate in your Simulink™ model. This example illustrates mapping to AXI4-Stream interfaces but you can map to AXI4-Stream Video or AXI4 Master interfaces by using this approach.

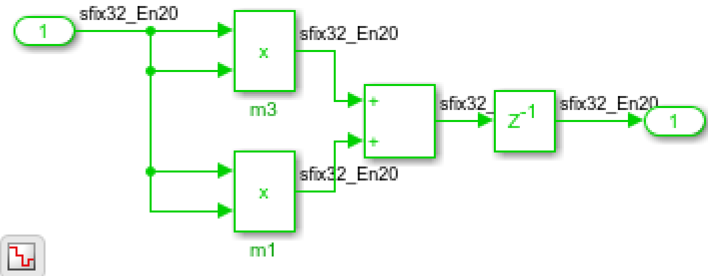
For an example, open the model `hdlcoder_axi_multirate_sharing`

```
load_system('hdlcoder_axi_multirate_sharing')
set_param('hdlcoder_axi_multirate_sharing','SimulationCommand','update')
open_system('hdlcoder_axi_multirate_sharing/DUT')
```



In this model, the Subsystem contains a simple multiply-add algorithm running at a slower rate.

```
open_system('hdlcoder_axi_multirate_sharing/DUT/Subsystem')
```



Resource sharing can be applied to this part of the design. To see the parameters saved on this Subsystem, run `hdlsaveparams`.

```
hdlsaveparams('hdlcoder_axi_multirate_sharing/DUT/Subsystem')

%% Set Model 'hdlcoder_axi_multirate_sharing' HDL parameters
hdlset_param('hdlcoder_axi_multirate_sharing', 'HDLSubsystem', 'hdlcoder_axi_multirate_sharing/DU
hdlset_param('hdlcoder_axi_multirate_sharing', 'ReferenceDesign', 'Default system with AXI4-Strea
hdlset_param('hdlcoder_axi_multirate_sharing', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisToolDeviceName', 'xc7z020');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisToolPackageName', 'clg484');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoder_axi_multirate_sharing', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_axi_multirate_sharing', 'TargetFrequency', 50);
hdlset_param('hdlcoder_axi_multirate_sharing', 'TargetPlatform', 'ZedBoard');
hdlset_param('hdlcoder_axi_multirate_sharing', 'Workflow', 'IP Core Generation');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_axi_multirate_sharing/DUT/Subsystem', 'SharingFactor', 3);
```

You can map the DUT interface ports to AXI4-Stream Master or AXI4-Stream Slave interfaces. This figure shows an example of the target platform interface mapping for this model.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
x_in_data	Inport	sfix16_E...	AXI4-Stream Slave	Data
x_in_valid	Inport	boolean	AXI4-Stream Slave	Valid
h_in1	Inport	sfix16_E...	AXI4-Lite	x"100"
h_in2	Inport	sfix16_E...	AXI4-Lite	x"104"
h_in3	Inport	sfix16_E...	AXI4-Lite	x"108"
h_in4	Inport	sfix16_E...	AXI4-Lite	x"10C"
y_out_data	Outport	sfix32_E...	AXI4-Stream Master	Data
y_out_valid	Outport	boolean	AXI4-Stream Master	Valid

See also “Model Design for AXI4-Stream Interface Generation” on page 40-14.

See Also

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- “Custom IP Core Generation” on page 39-17

Board and Reference Design Registration System

In this section...

"Board, IP Core, and Reference Design Definitions" on page 40-89
"Board Registration Files" on page 40-89
"Reference Design Registration Files" on page 40-90
"Predefined Board and Reference Design Examples" on page 40-91

You can define custom boards and custom reference designs so that they are available as target hardware options in the SoC workflow. Custom boards and custom reference designs use the same system that HDL Coder uses for predefined board and reference design targets.

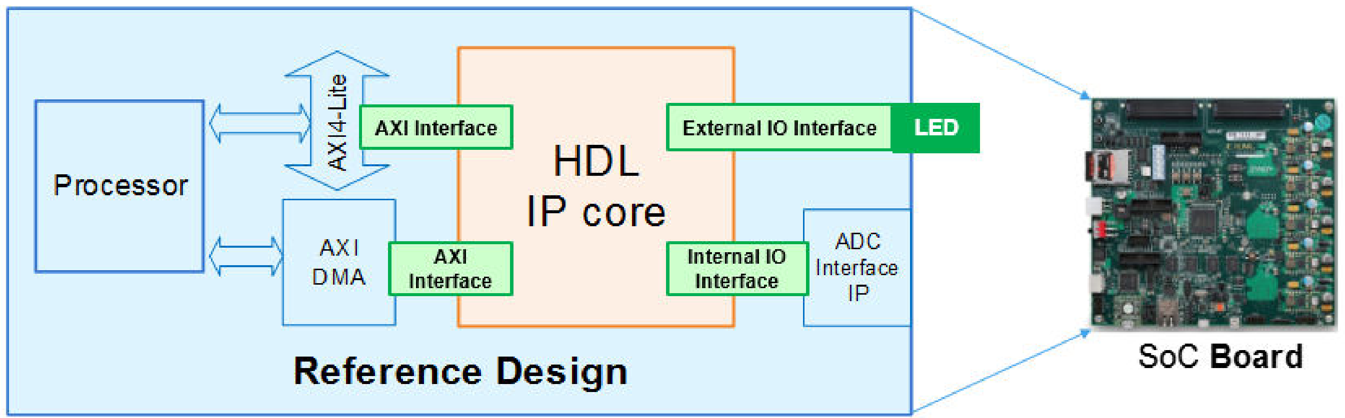
Board, IP Core, and Reference Design Definitions

A reference design is the embedded system design that your generated IP core integrates with. The board is the SoC platform.

For a custom board or custom reference design, you can define different kinds of interfaces:

- *AXI interface*: an interface between your generated IP core and an AXI4 or AXI4-Lite interface.
- *External IO interface*: an interface between your generated IP core and an external interface.
- *Internal IO interface*: an interface between your generated IP core and another IP core in the reference design.

After you integrate your reference design and IP core in an embedded system design project, you can program the board with the embedded system design.



Board Registration Files

To define and register a board, you must have a board definition, a board plugin, and a board registration file.

Board Definition

A board definition is a file that defines the characteristics of a board. You can define more than one custom board.

Board Plugin

A board plugin is a package folder that contains:

- The board definition.
- All reference design plugins that are associated with the board.

A board plugin has one board definition, but can have multiple reference designs.

Board Registration File

A board registration file is always named `hdlcoder_board_customization.m`, and contains a list of board plugins. There can be multiple board registration files on your MATLAB path, but a board plugin cannot be listed in more than one board registration file.

When the HDL Workflow Advisor opens, it searches the MATLAB path for files named `hdlcoder_board_customization.m`, and uses the information to populate the target board options. Interfaces you add and define for the board appear as options in the **Target Platform Interface** drop-down list.

Reference Design Registration Files

To define and register a reference design, you must have a reference design definition, a reference design plugin, and a reference design registration file.

Reference Design Definition

A reference design definition is a file that defines the characteristics of a reference design, including its associated board and interfaces. You can define multiple custom reference designs per board.

Reference Design Plugin

A reference design plugin is a package folder that contains:

- The reference design definition.
- Files that are part of the embedded system design project, and are specific to your third-party synthesis tool, including Tcl, project, and design files.

A reference design plugin has one reference design definition and is associated with one board.

Reference Design Registration File

A reference design registration file is always named `hdlcoder_ref_design_customization.m`, and contains a list of reference design plugins for a specific board. There can be multiple reference design registration files for a specific board on your MATLAB path, but a reference design plugin cannot be listed in more than one reference design plugin registration file.

When the HDL Workflow Advisor opens, it searches the MATLAB path for files named `hdlcoder_ref_design_customization.m`, and uses the information to populate the reference

design options for each board. Interfaces you add and define for the reference design appear as options in the **Target Platform Interface** drop-down list.

Predefined Board and Reference Design Examples

For examples of working board and reference design definitions, refer to the predefined Altera SoC, Xilinx Zynq and Microchip SoC board plug-ins that include predefined reference design plug-ins:

- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/zynq7000/+VCK190/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/zynq7000/+ZCU102/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/zynq7000/+ZedBoard/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/zynq7000/+ZynqZC702/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/zynq7000/+ZynqZC706/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/alterasoc/+AlteraCycloneV/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/alterasoc/+ArrowSoCKit/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/alterasoc/+IntelArria10SoC/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/microchip/+PolarFireSoC/`

To get to the root folder of support packages, use the `matlabshared.supportpkg.getSupportPackageRoot` function.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Register a Custom Board” on page 40-92
- “Register a Custom Reference Design” on page 40-95
- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270

Register a Custom Board

In this section...

"Define a Board" on page 40-92
"Create a Board Plugin" on page 40-93
"Define a Board Registration Function" on page 40-93

To register a custom board, you must:

1. Define a board.
2. Create a board plugin.
3. Define a board registration function, or add the new board plugin to an existing board registration function.

Define a Board

Before you begin, have the board documentation at hand so you can refer to the details of the board.

Requirements

A board definition must be:

- A MATLAB function that returns an `hdlcoder.Board` object.

  The board definition function can have any name.
- In its board plugin folder.

How To Define A Board

1. Create a new file that defines a MATLAB function with any name.
2. In the MATLAB function, create an `hdlcoder.Board` object and specify its properties and interfaces according the characteristics of your custom board.
3. Optionally, to check that the definition is complete, run the `validateBoard` method.

For example, the following code defines a board:

```
function hB = plugin_board()
% Board definition

% Construct board object
hB = hdlcoder.Board;

hB.BoardName    = 'Digilent Zynq ZyBo';

% FPGA device information
hB.FPGAVendor   = 'Xilinx';
hB.FPGAFamily   = 'Zynq';
hB.FPGADevice   = 'xc7z010';
hB.FPGAPackage  = 'clg400';
hB.FPGASpeed    = '-2';

% Tool information
```

```
hB.SupportedTool = {'Xilinx Vivado'};

% FPGA JTAG chain position
hB.JTAGChainPosition = 2;

%% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});
```

Create a Board Plugin

Requirements

A board plugin:

- Must be a package folder that contains the board definition file.

  A package folder has a + prefix before the folder name. For example, the board plugin can be a folder named `+ZedBoard`.
- Must be on the MATLAB path.
- Can contain one or more reference design plugins.

How To Create a Board Plugin

1. Create a folder that has a name with a + prefix.
2. Save your board definition file to the folder.
3. Add the folder to your MATLAB path.

Define a Board Registration Function

Requirements

A board registration function:

- Must be named `hdlcoder_board_customization.m`.
- Returns a list of board plugins, specified as a cell array of character vectors.
- Must be on the MATLAB path.

How To Define a Board Registration Function

1. Create a file named `hdlcoder_board_customization.m` and save it anywhere on the MATLAB path.
2. In `hdlcoder_board_customization.m`, define a function that returns a list of board plugins as a cell array of character vectors.

   For example, the following code defines a board registration function.

   ```
   function r = hdlcoder_board_customization
   % Board plugin registration files
   % Format: % board_folder.board_definition_function
   ```

```
r = {'ZyboRegistration.plugin_board'};

end
```

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Register a Custom Reference Design” on page 40-95
- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270
- “Define Custom Board and Reference Design for Microchip Workflow” on page 40-285

More About

- “Board and Reference Design Registration System” on page 40-89

Register a Custom Reference Design

In this section...

"Define a Reference Design" on page 40-95
"Create a Reference Design Plugin" on page 40-96
"Define a Reference Design Registration Function" on page 40-96

To register a custom reference design:

1. Define a reference design.
2. Create a reference design plugin.
3. Define a reference design registration function, or add the new reference design plugin to an existing reference design registration function.

Define a Reference Design

A reference design definition must be a MATLAB function that returns an `hdlcoder.ReferenceDesign` object. Create the reference design definition function in the reference design plugin folder. You can use any name for the reference design definition function.

To create a reference design definition:

1. Create a new file that defines a MATLAB function with any name.
2. In the MATLAB function, create an `hdlcoder.ReferenceDesign` object and specify its properties and interfaces according to the characteristics of your embedded system design.
3. If you want to check that the definition is complete, run the `validateReferenceDesign` method.

This MATLAB function defines a custom reference design:

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Demo system';
hRD.BoardName = 'Digilent Zynq ZyBo';

% Tool information
% It is recommended to use a tool version that is compatible with the supported tool
% version. If you choose a different tool version, it is possible that HDL Coder is
% unable to create the reference design project for IP core integration.
hRD.SupportedToolVersion = {'2020.2'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'design_led.tcl');

hRD.CustomFiles = {'ZYBO_zynq_def.xml'};
%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',   'clk_wiz_0/clk_out1', ...
    'ResetConnection',   'proc_sys_reset_0/peripheral_aresetn');

% add AXI4 and AXI4-Lite slave interfaces
```

```
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_interconnect_0/M00_AXI', ...
    'BaseAddress',         '0x40010000', ...
    'MasterAddressSpace',  'processing_system7_0/Data');
```

By default, HDL Coder generates an IP core with the default settings and integrates it into the reference design project. To customize these default settings, use the properties in the `hdlcoder.ReferenceDesign` object to define custom parameters and to register the function handle of the custom callback functions. For more information, see “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 40-98.

Create a Reference Design Plugin

A reference design plugin is a package folder that you define on the MATLAB path. The folder contains the board definition file and any custom callback functions.

To create a reference design plugin:

1. In the board plugin folder for the associated board, create a new folder that has a name with a + prefix.

   For example, the reference design plugin can be a folder named `+vivado_base_ref_design`.
2. In the new folder, save your reference design definition file and any custom callback functions that you create.
3. In the new folder, save any files that are required by the embedded system design project, and are specific to your third-party synthesis tool, including Tcl, project, and design files.
4. Add the folder to your MATLAB path.

Define a Reference Design Registration Function

A reference design registration function contains a list of reference design functions and the associated board name. You must name the function `hdlcoder_ref_design_customization.m`. When the HDL Workflow Advisor opens, it searches the MATLAB path for files named `hdlcoder_ref_design_customization.m`, and uses the information to populate the reference design options for each board.

To define a reference design registration function:

1. Create a file named `hdlcoder_ref_design_customization.m` and save it anywhere on the MATLAB path.
2. In `hdlcoder_board_customization.m`, define a function that returns the associated board name, specified as a character vector, and a list of reference design plugins, specified as a cell array of character vectors.

   For example, the following code defines a reference design registration function.

   ```
   function [rd, boardName] = hdlcoder_ref_design_customization
   % Reference design plugin registration file

   rd = {'ZyBoRegistration.Vivado2018_2.plugin_rd', ...
        };

   boardName = 'Digilent Zynq ZyBo';
   ```

```
    end
```

The reference design registration function returns the associated board name, specified as a character vector, and a list of reference design plugins, specified as a cell array of character vectors.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- "Register a Custom Board" on page 40-92
- "Define Custom Parameters and Callback Functions for Custom Reference Design" on page 40-98
- "Define Custom Board and Reference Design for Zynq Workflow" on page 40-252
- "Define Custom Board and Reference Design for Intel SoC Workflow" on page 40-270

More About

- "Board and Reference Design Registration System" on page 40-89

Define Custom Parameters and Callback Functions for Custom Reference Design

In this section...
"Define Custom Parameters and Register Callback Function Handle" on page 40-98
"Define Custom Callback Functions" on page 40-102

When you define your custom reference design, you can optionally use the properties in the `hdlcoder.ReferenceDesign` object to define custom parameters and callback functions.

Define Custom Parameters and Register Callback Function Handle

This MATLAB code shows how to define custom parameters and register the function handle of the custom callback functions in the reference design definition function.

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.ReferenceDesignName = 'My Reference Design';
hRD.BoardName = 'ZedBoard';

% Tool information
hRD.SupportedToolVersion = {'2020.2'};

%% Add custom design files
% ...
% ...

%% Add optional custom parameters by using addParameter property.
% Specify custom 'DUT path' and 'Channel Mapping' parameters.
% The parameters get populated in the 'Set Target Reference Design'
% task of the HDL Workflow Advisor.
hRD.addParameter( ...
    'ParameterID',   'DutPath', ...
    'DisplayName',   'Dut Path', ...
    'DefaultValue',  'Rx', ...
    'ParameterType',  hdlcoder.ParameterType.Dropdown, ...
    'Choice',       {'Rx', 'Tx'});
hRD.addParameter( ...
    'ParameterID',   'ChannelMapping', ...
    'DisplayName',   'Channel Mapping', ...
    'DefaultValue',  '1');

%% Enable AXI manager IP insertion for the JTAG connection. The IP
% insertion setting is visible in the 'Set Target Reference Design'
% task of the HDL Workflow Advisor. By default, the
% AddMATLABAXIManagerParameter property is set to 'true'.
hRD.AddMATLABAXIManagerParameter = 'true';
hRD.MATLABAXIManagerDefaultValue = 'JTAG';

%% Enable AXI manager IP insertion for the Ethernet connection. The IP
% insertion setting is visible in the 'Set Target Reference Design'
% task of the HDL Workflow Advisor. By default, the Ethernet option
% for the AXI manager IP insertion setting is available for the
% Artix-7 35T Arty, Kintex-7 KC705, and Virtex-7 VC707 boards.
% Enable this option for other Xilinx boards that have the Ethernet
% physical layer (PHY) by adding the Ethernet media access
% controller (MAC) Hub IP in the plugin_board file before launching
% the HDL Workflow Advisor. To add the Ethernet MAC Hub IP, use
% the addEthernetMACInterface method.
hRD.AddMATLABAXIManagerParameter = 'true';
hRD.MATLABAXIManagerDefaultValue = 'Ethernet';
```

```
%% Add custom callback functions. These are optional.
% With the callback functions, you can enable custom
% validations, customize the project creation, software
% interface model generation, and the bistream build.
% Register the function handle of these callback functions.

% Specify an optional callback for 'Set Target Reference Design'
% task in Workflow Advisor. Use property name
% 'PostTargetReferenceDesignFcn'.
hRD.PostTargetReferenceDesignFcn = ...
    @my_reference_design.callback_PostTargetReferenceDesign;

% Specify an optional callback for 'Set Target Interface' task in Workflow Advisor.
% Use the property name 'PostTargetInterfaceFcn'.
hRD.PostTargetInterfaceFcn     = ...
    @my_reference_design.callback_PostTargetInterface;

% Specify an optional callback for 'Generate IP Core' task
hRD.PostGenerateIPCoreFcn =...
 @my_reference_design.callback_PostGenerateIPCoreFcn;

% Specify an optional callback for 'Create Project' task
% Use the property name 'PostCreateProjectFcn' for the ref design object.
hRD.PostCreateProjectFcn       = ...
    @my_reference_design.callback_PostCreateProject;

% Specify an optional callback for 'Generate Software Interface Model' task
% Use the property name 'PostSWInterfaceFcn' for the ref design object.
hRD.PostSWInterfaceFcn         = ...
    @my_reference_design.callback_PostSWInterface;

% Specify an optional callback for 'Build FPGA Bitstream' task
% Use the property name 'PostBuildBitstreamFcn' for the ref design object.
hRD.PostBuildBitstreamFcn      = ...
    @my_reference_design.callback_PostBuildBitstream;

% Specify an optional callback for 'Program Target Device'
% task to use a custom programming method.
hRD.CallbackCustomProgrammingMethod = ...
    @my_reference_design.callback_CustomProgrammingMethod;

%% Add interfaces
% ...
% ...
```

Define Custom Parameters

With the `addParameter` method of the `hdlcoder.ReferenceDesign` class, you can define custom parameters. In the preceding example code, the reference design defines two custom parameters, `DUT Path` and `Channel Mapping`. To learn more about the `addParameter` method, see `addParameter`.

Specify Insertion of AXI Manager IP

By default, HDL Coder adds the **Insert AXI Manager (HDL Verifier required)** parameter to all reference designs. When you set this parameter to `JTAG`, the code generator inserts the JTAG AXI Manager IP into your reference design. When you set this parameter to `Ethernet`, the code generator inserts the UDP AXI Manager IP into your reference design.

Note By default, the `Ethernet` option is available for only the Artix-7 35T Arty, Kintex-7 KC705, and Virtex-7 VC707 boards. To enable this option for other Xilinx boards that have the Ethernet physical layer (PHY), manually add the Ethernet media access controller (MAC) Hub IP in the `plugin_board`

file using the `addEthernetMACInterface` method before you launch the HDL Workflow Advisor tool.

By using the AXI manager IP, you can easily access the AXI registers in the generated DUT IP core on a hardware board from MATLAB or Simulink through the JTAG or Ethernet connection. See also “Set Up AXI Manager” (HDL Verifier).

To use this capability, you must have the HDL Verifier hardware support packages installed and downloaded. See “Download FPGA Board Support Package” (HDL Verifier).

The code generator adjusts the **AXI4 Slave ID Width** to accommodate the AXI manager IP connection. After you generate the HDL IP core and create the reference design project, you can open the Vivado block design to see the AXI manager IP inserted in the reference design.

In the previous example code, the reference design defines the `AddMATLABAXIManagerParameter` and `MATLABAXIManagerDefaultValue` properties of the `hdlcoder.ReferenceDesign` class. These properties control the default behavior of the **Insert AXI Manager (HDL Verifier required)** setting in the **Set Target Reference Design** task of the HDL Workflow Advisor. If you do not specify any of these properties in the `hdlcoder.ReferenceDesign` class, the **Insert AXI Manager (HDL Verifier required)** parameter is visible in the **Set Target Reference Design** task and the value is set to `off`. This example code illustrates the default behavior.

This property controls visibility of the **Insert AXI Manager (HDL Verifier required)** parameter in the **Set Target Reference Design** task of the HDL Workflow Advisor. By default, the property value is `'true'`, which means that the parameter is visible in the task. To disable the parameter, set this value to `'false'`.

```
hRD.AddMATLABAXIManagerParameter = 'true';
```

This property controls the value of the **Insert AXI Manager (HDL Verifier required)** parameter in the **Set Target Reference Design** task. By default, the property value is `'off'`, which means that the parameter is visible in the task and the value is `off`. To enable automatic insertion of AXI manager IP in the reference design, set this value to `'JTAG'` or `'Ethernet'`. In that case, you must set the `AddMATLABAXIManagerParameter` to `'true'`.

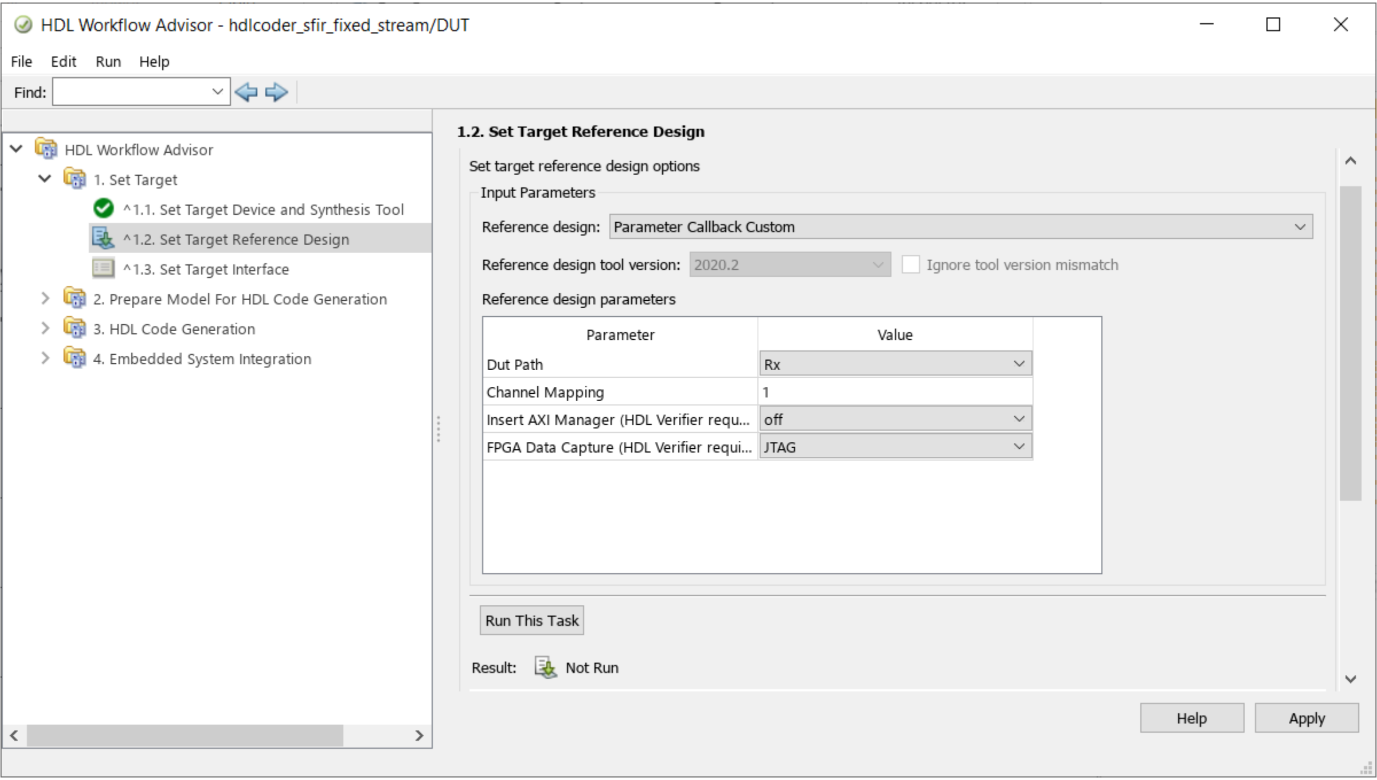
```
hRD.MATLABAXIManagerDefaultValue = 'off';
```

For examples, see:

- Using JTAG MATLAB as AXI Master to control the HDL Coder IP Core on page 40-333
- “Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow” on page 39-190

Run IP Core Generation Workflow

When you open the HDL Workflow Advisor, HDL Coder populates the **Set Target Reference Design** task with the reference design name, tool version, custom parameters that you specified, and the **Insert AXI Manager (HDL Verifier required)** option set to `JTAG`.



HDL Coder then passes these parameter values to the callback functions in the input structure.

If your synthesis tool is Xilinx Vivado, HDL Coder sets the reference design parameter values to variables. The variables are then input to the block design Tcl file. This code snippet is an example from the reference design project creation Tcl file.

```
update_ip_catalog
set DutPath {Rx}
set ChannelMapping {1}
source vivado_custom_block_design.tcl
```

The code shows how HDL Coder sets the reference design parameters before sourcing the custom block design Tcl file.

Register Callback Function Handles

In the reference design definition, you can register the function handle to reference the custom callback functions. You then can:

- Enable custom validations.
- Customize the reference design dynamically.
- Customize the reference design project creation settings.
- Change the generated software interface model.
- Customize the FPGA bitstream build process.
- Specify custom FPGA programming method.

With the `hdlcoder.ReferenceDesign` class, you can define callback property names. The callback properties have a naming convention. The callback functions can have any name. In the HDL Workflow Advisor, you can define callback functions to customize these tasks.

Workflow Advisor Task	Callback Property Name	Functionality
Set Target Reference Design	• `CustomizeReferenceDesignFcn` • `PostTargetReferenceDesignFcn`	• `CustomizeReferenceDesignFcn` enables customization of the reference design dynamically. By using this callback function, you can customize the block design Tcl file, reference design interfaces, reference design interface properties, and IP repositories in your reference design. See "Customize Reference Design Dynamically Based on Reference Design Parameters" on page 40-104. • `PostTargetReferenceDesignFcn` enables custom validations. For an example that shows how you can validate that the **Reset type** is `Synchronous`, see `PostTargetReferenceDesignFcn`.
Set Target Interface	`PostTargetInterfaceFcn`	Enable custom validations. For an example that shows how you can validate not choosing a certain interface for a certain custom parameter setting, see `PostTargetInterfaceFcn`.
Generate IP Core	`PostGenerateIPCoreFcn`	Specify custom tasks to run after HDL Coder generates the IP core. For an example, see `PostGenerateIPCoreFcn`.
Create Project	`PostCreateProjectFcn`	Specify custom settings when HDL Coder creates the project. For an example, see `PostCreateProjectFcn`.
Generate Software Interface	`PostSWInterfaceFcn`	Change the generated software interface model. For an example, see `PostSWInterfaceFcn`.
Build FPGA Bitstream	`PostBuildBitstreamFcn`	Specify custom settings when you build the FPGA bitstream. When you use this function, the build process cannot be run externally. You must run the build process within the HDL Workflow Advisor by clearing the **Run build process externally** check box in the **Build FPGA Bitstream** task. For an example, see `PostBuildBitstreamFcn`.
Program Target Device	`CallbackCustomProgrammingMethod`	Specify a custom FPGA programming method. For an example, see `CallbackCustomProgrammingMethod`.

Define Custom Callback Functions

1. For each of the callback function that you want HDL Coder to execute after running a task, create a file that defines a MATLAB function with any name.
2. Make sure that the callback function has the documented input and output arguments.
3. Verify that the functions are accessible from the MATLAB path.
4. Register the function handle of the callback functions in the reference design definition function.

5 Follow the naming conventions for the callback property names.

To learn more about these callback functions, see `hdlcoder.ReferenceDesign`.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Register a Custom Board” on page 40-92
- “Register a Custom Reference Design” on page 40-95
- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270
- “Define Custom Board and Reference Design for Microchip Workflow” on page 40-285

More About

- “Board and Reference Design Registration System” on page 40-89

Customize Reference Design Dynamically Based on Reference Design Parameters

In this section...

"Why Customize the Reference Design" on page 40-104

"How Reference Design Customization Works" on page 40-104

"Customizable Reference Design Parameters" on page 40-105

"Example: Create Master Only or Slave Only or Both Slave and Master Reference Designs" on page 40-106

When you define your own custom reference design, you can dynamically customize the reference design by using the `CustomizeReferenceDesignFcn` method of the `hdlcoder.ReferenceDesign` class.

Why Customize the Reference Design

By customizing the reference design parameters, instead of maintaining separate reference designs that have different interface choices, data widths, or I/O plugins, create one reference design that has different interface choices or data widths as parameters. You can then use the `CustomizeReferenceDesignFcn` method to reference the callback function that has different choices for interfaces or data widths.

For example, instead of creating separate reference designs that have different data widths for the interfaces, you can parameterize the data width and then create a reference design parameter. When you run the `IP Core Generation` workflow, you can use the parameter to select the data width that you want to use. Similarly, instead of using multiple reference designs, you can create one reference design that has only AXI4-Stream Master or only AXI4-Stream Slave or both AXI4-Stream Master and AXI4-Stream Slave interfaces.

How Reference Design Customization Works

To define the callback function:

1. In the `plugin_rd` file, define the reference design parameters you want to customize by using the `addParameter` method.
2. Create a MATLAB file that defines the callback function. You can use any arbitrary name for the callback function.
3. Save the callback function in the same folder as the `plugin_rd.m` file.
4. Register the function handle of the callback function in the reference design definition `plugin_rd` file by using the `CustomizeReferenceDesignFcn` method.

To use the different reference design customizations:

1. Open the HDL Workflow Advisor. In the **Set Target Device and Synthesis Tool** task, select `IP Core Generation` as the **Target workflow** and then select the target board for which you created your own custom reference design as the **Target platform**.
2. In the **Set Target Reference Design** task, when you select the custom reference design that you want to customize for the target board, HDL Coder populates the reference design

parameters. Depending on the parameter choices such as the interface types you specify, the callback function is evaluated. Run this task.

3 Select the **Set Target Interface** task. Depending on the parameter you selected in the previous step, the target interface selection is populated in the Target platform interface table.

Customizable Reference Design Parameters

In the callback function, you can customize these reference design parameters. Do not specify these parameters in the `plugin_rd` file.

- Block design Tcl file

```
% ...

% if ~isempty(ParamValue)
    hRD.addCustomVivadoDesign( ...
            'CustomBlockDesignTcl', 'system_top.tcl', ...
            'VivadoBoardPart',      'xilinx.com:zc706:part0:1.0'};

% ...
```

- Reference design interfaces and reference design interface properties

  For example, you can parameterize the data width of the AXI4-Stream Master Channel. In this case, use the `addAXI4StreamInterface` method in the callback function instead of the `plugin_rd` file.

```
% ...

% Add AXI4-Stream interface by parameterizing data width
DataWidth = hRD.getParamValue(paramValue)

if ~isempty(DataWidth)
   hRD.addAXI4StreamInterface(
    'MasterChannelEnable',     'true', ...
    'SlaveChannelEnable',      'true', ...
    'MasterChannelConnection', 'ByPass_0.AXI4_Stream_Slave', ...
    'SlaveChannelConnection',  'ByPass_0.AXI4_Stream_Master', ...
    'MasterChannelDataWidth',   DataWidth, ...
    'SlaveChannelDataWidth',    DataWidth);
end

% ...
```

- IP repositories

  In the callback function, you must specify the block design Tcl file when you add IP repositories.

```
% ...

%% Add IP Repository
hRD.addIPRepository(...
    'IPListFunction', 'mathworks.hdlcoder.vivado.hdlcoder_video_iplist',
         'NotExistMessage', 'IP repository not found');

%% Add custom design files
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'em.avnet.com:zed:part0:1.0');


% ...
```

Note You cannot modify the reference design name, board name, and supported tool versions in the callback function. These parameters are used in the **Set Target Device and Synthesis Tool** task

before the callback function is evaluated when the target reference design is selected in the **Set Target Reference Design** task.

Example: Create Master Only or Slave Only or Both Slave and Master Reference Designs

Instead of using multiple reference designs, you can create one reference design that has only AXI4-Stream Master or only AXI4-Stream Slave or both AXI4-Stream Master and AXI4-Stream Slave interfaces. This example shows how you can customize the AXI4-Stream interface channels you want to use when targeting your own reference design for the `Xilinx Zynq ZC706 evaluation kit`.

This code shows the reference design parameter and interface choices for the reference design `my_reference_design` specified by using the `addParameter` method in the `plugin_rd` file. The `CustomizeReferenceDesignFcn` method references a callback function that has the name `customcallback_axistreamchannel`.

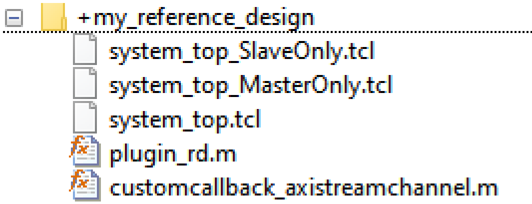
```
function hRD = plugin_rd()
% Reference design definition

%   Copyright 2017-2019 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Vivado Custom Reference Design';
hRD.BoardName = 'Xilinx Zynq ZC706 evaluation kit';

% Tool information
hRD.SupportedToolVersion = {'2019.1'};

% ...
% ...

% Parameter For  calling AXI4 Master
interface from Callback function
hRD.addParameter ...
('ParameterID'    ,  'stream_channel', ...
 'DisplayName'    ,  'Stream Channel', ...
 'DefaultValue'   ,  'Both Master and Slave',...
 'ParameterType'  ,   hdlcoder.ParameterType.Dropdown, ...
 'Choice'         ,   {'Both Master and Slave','Master Only','Slave Only'});

% Reference the callback function.
hRD.CustomizeReferenceDesignFcn = @my_reference_design.customcallback_axistreamchannel;

% ...
```

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. This code shows the callback function `customcallback_axistreamchannel` that has the AXI4-Stream Master or Slave Channels or both channels specified by using the `addAXI4StreamInterface` method.

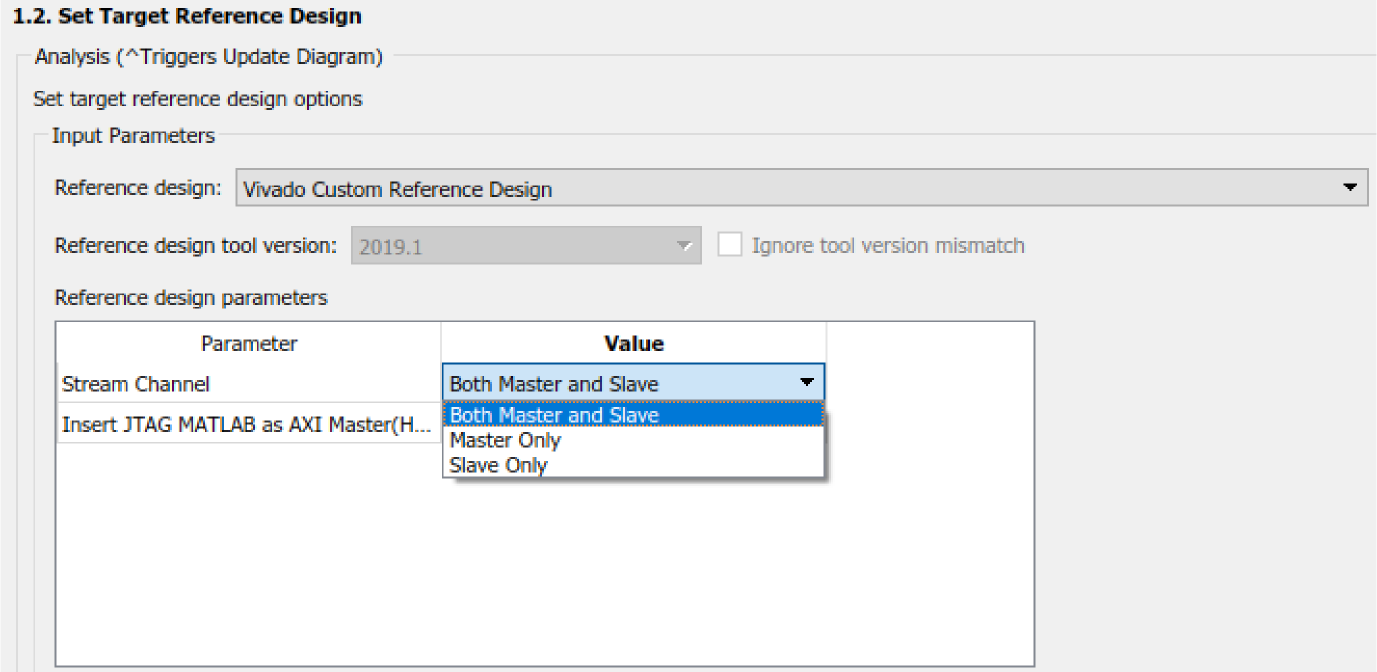
```
% Control AXI Master or Slave channel selection by using callback function

function customcallback_axistreamchannel(infoStruct)
%% Reference design callback run at the end of the task Set Target Reference Design
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in struct format
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
```

```
% infoStruct.ReferenceDesignToolVersion: Reference design Tool Version set in 1.2 Task

paramStruct = infoStruct.ParameterStruct;

if ~isempty(paramStruct)
    paramIDCell = fieldnames(paramStruct);
    paramValue = '';
    for ii = 1:length(paramIDCell)
        paramID = paramIDCell(ii);
        if strcmp(paramID,'Both Master and Slave')
            paramValue = paramStruct.paramID;
            break;
        elseif strcmp(paramID,'Master Only')
            paramValue = paramStruct.paramID;
            break;
        elseif strcmp(paramID,'Slave Only')
            paramValue = paramStruct.paramID;
            break;
        end
    end
end
interface_type = str2double(paramValue);

if ~isempty(interface_type)

    if strcmp(interface_type, 'Both Master and Slave')

        % add custom vivado design
        hRD.addCustomVivadoDesign( ...
            'CustomBlockDesignTcl', 'system_top.tcl', ...
            'VivadoBoardPart',      'xilinx.com:zc706:part0:1.0');

        hRD.addAXI4StreamInterface( ...
            'MasterChannelEnable',      'true', ...
            'SlaveChannelEnable',       'true', ...
            'MasterChannelConnection',  'axi_dma_s2mm/S_AXIS_S2MM', ...
            'SlaveChannelConnection',   'axi_dma_mm2s/M_AXIS_MM2S', ...
            'MasterChannelDataWidth',   32, ...
            'SlaveChannelDataWidth',    32);

    elseif strcmp(interface_type, 'Master Only')

        % add custom vivado design
        hRD.addCustomVivadoDesign( ...
            'CustomBlockDesignTcl', 'system_top.tcl', ...
            'VivadoBoardPart',      'xilinx.com:zc706:part0:1.0');

        hRD.addAXI4StreamInterface( ...
            'MasterChannelEnable',      true, ...
            'MasterChannelConnection',  'axi_dma_s2mm/S_AXIS_S2MM', ...
            'MasterChannelDataWidth',   32);


    % ...
```

Save the callback function in the same folder as the `plugin_rd` file.



When you run the `IP Core Generation` workflow with `Xilinx Zynq ZC706 evaluation kit` as the **Target platform**, you see the parameter **Stream Channel** displayed in the **Set Target Reference Design** task.



You can specify the reference design type you want to use and then run the workflow to specify the target platform interfaces and then generate the HDL IP core and then integrate the IP core into the reference design.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Register a Custom Board” on page 40-92
- “Register a Custom Reference Design” on page 40-95
- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270

More About

- “Board and Reference Design Registration System” on page 40-89

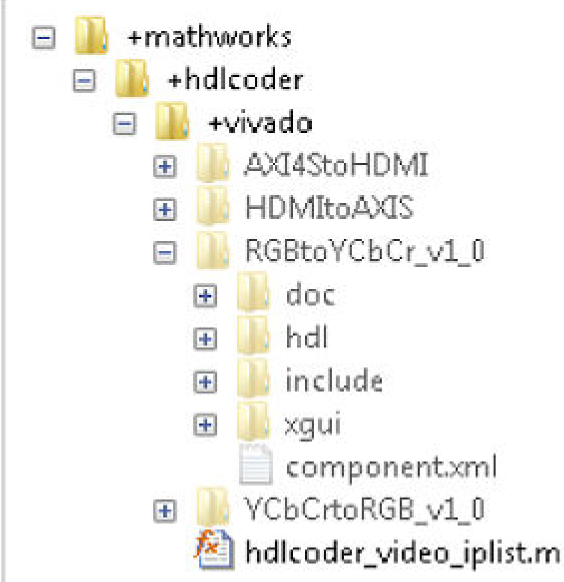
Define and Add IP Repository to Custom Reference Design

In this section...
“Create an IP Repository Folder Structure” on page 40-109
“Define IP List Function” on page 40-110
“Add IP List Function to Reference Design Project” on page 40-111

When you create your custom reference design, you might require custom IP modules that do not come with Altera Qsys or Xilinx Vivado. To use custom IP modules, create your own IP repository folder that contains IP module subfolders. The `IP Core Generation` workflow then uses these custom IP modules when creating the reference design project. You can create multiple IP repositories and add all or some of the IP modules in each repository to your custom reference design project. You can also reuse and share IP repositories across multiple reference designs.

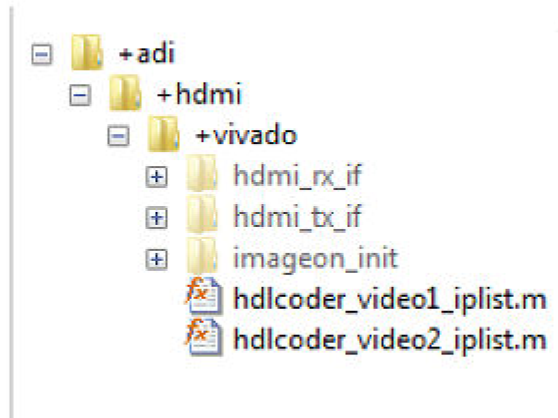
Create an IP Repository Folder Structure

Create an IP repository folder anywhere on the MATLAB path. This figure shows a typical IP repository folder structure.



When you create the folder structure, use the naming convention `+(company)/+(product)/+(tool)`. In this example, the folder structure is `+(mathworks)/+(hdlcoder)/+(vivado)`. The folder `+vivado` acts as the IP repository. This folder contains subfolders corresponding to IP modules such as `AXI4StoHDMI` and `HDMItoAXIS`. The folder also contains a `hdlcoder_video_iplist` MATLAB function. Using this function, specify the IP modules to add to the reference design.

The same repository folder can have multiple MATLAB functions. This figure shows two MATLAB functions, `hdlcoder_video1_iplist` and `hdlcoder_video2_list`, in the `+vivado` folder. The functions can share the same IP modules or point to different IP modules in the repository.



Note If your synthesis tool is Xilinx Vivado, the IP modules in the repository folder can be in zip file format.

Define IP List Function

Create a MATLAB function that specifies the IP modules to add to the reference design. Save this function in the IP repository folder. For the function name, use the naming convention `hdlcoder_<specific_use>_iplist`. This example uses `hdlcoder_video_iplist` as the function name because it targets video applications. Using this function, specify whether you want to add all or some of the IP modules in the repository to the reference design project. To add all IP modules, use an empty cell array for `ipList`. This MATLAB code shows how to add all IP modules in the repository to the reference design.

```
function [ipList] = hdlcoder_video_iplist(  )
% All IP modules in the repository folder.

ipList = {};
```

You can specify the root directory as an optional second output argument to the IP list function. In this case, the IP modules do not have to be located in a path relative to the IP list function. The IP repository can also be located outside the MATLAB path.

If you do not specify the root directory, the function searches for IP modules relative to its location.

```
function [ipList, rootDir] = hdlcoder_video_iplist( )
% All IP modules with a root directory.

ipList = {};
```

To add some of the IP modules that are in the folder, specify the IP modules as a cell array of character vectors. This MATLAB code specifies the `AXI4StoHDMI` IP and the `HDMItoAXIS` IP as the IP modules to add to your custom reference design.

```
function [ipList] = hdlcoder_video_iplist(  )
% AXI4StoHDMI and HDMItoAXIS IP in the repository folder.

ipList = {'AXI4StoHDMI','HDMItoAXIS'};
```

Add IP List Function to Reference Design Project

Using the `addIPRepository` method of the `hdlcoder.ReferenceDesign` class, add the IP list function to your custom reference design. This example reference design adds `hdlcoder_video_iplist` to the custom reference design `My Reference Design`.

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.ReferenceDesignName = 'My Reference Design';
hRD.BoardName = 'ZedBoard'

% Tool information
hRD.SupportedToolVersion = {'2020.2'};

%% Add custom design files
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'em.avnet.com:zed:part0:1.0');

% Add IP Repository
hRD.addIPRepository(...
    'IPListFunction', 'mathworks.hdlcoder.vivado.hdlcoder_video_iplist',
         'NotExistMessage', 'IP repository not found');

% ...
% ...
```

To use the IP modules when the code generator creates the project, open the HDL Workflow Advisor, and run the `IP Core Generation` workflow to the **Create Project** task. After running this task, you can see the IP module subfolders in the repository copied over to the `ipcore` folder of the project. The `CustomBlockDesignTcl` can then use these IP modules.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270

More About

- “Board and Reference Design Registration System” on page 40-89

- “Register a Custom Reference Design” on page 40-95
- “Customize Reference Design Dynamically Based on Reference Design Parameters” on page 40-104

FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules

This example shows how to implement a Simulink® algorithm on a Speedgoat® Simulink-programmable I/O module by using the HDL Workflow Advisor. You run the `Simulink Real-Time FPGA IO` workflow to:

1. Specify an FPGA I/O module and its interfaces.
2. Synthesize the Simulink algorithm for FPGA programming.
3. Generate a Simulink® Real-Time™ interface subsystem model.

The interface subsystem model contains blocks to program the FPGA and communicate with the FPGA module through the PCIe bus during real-time application execution. You add the generated subsystem to your Simulink Real-Time domain model.

This example uses the Speedgoat IO397-50k module. See “Speedgoat FPGA Support with HDL Workflow Advisor” on page 39-15.

Setup and Configuration

Before deploying your algorithm on the Speedgoat IO module:

1. Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat')
```

2. For real-time simulation, set up the development environment and target computer settings. See “Get Started with Simulink Real-Time” (Simulink Real-Time).

3. Install the Speedgoat Library and the Speedgoat HDL Coder Integration packages. See Install Speedgoat HDL Coder Integration Packages.

HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.
- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis and timing analysis.
- Deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

To open the HDL Workflow Advisor for a subsystem inside the model, use the `hdladvisor` function.

```
load_system('sschdlexTwoLevelConverterIgbtExample')
hdladvisor('sschdlexTwoLevelConverterIgbtExample/Simscape_system')
```

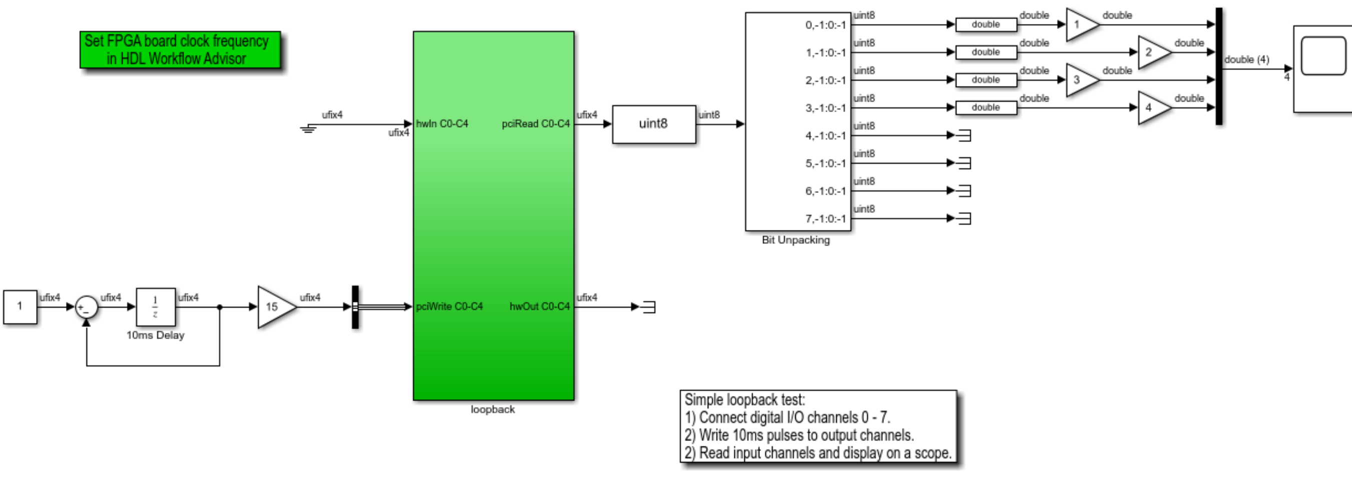
The left pane of the Advisor contains folders that represent a group of related tasks. Expanding the folders and selecting a task displays information about that task in the right pane. The right pane

contains simple controls for running the task to advanced parameters and option settings that control HDL code and test bench generation. To learn more about each task, right-click that task, and select **What's This?**. See "Getting Started with the HDL Workflow Advisor" on page 29-5.

Simulink Loopback Domain Model

This model is your FPGA domain model. It represents the simulation sample rate of the clock on your FPGA board. The `loopback` subsystem contains the algorithm to load on the FPGA. The data type and the number of input and output lines of the model are configured to fit the `Speedgoat IO397-50k` platform.

```
open_system('hdlcoder_slrt_loopback')
set_param('hdlcoder_slrt_loopback', 'SimulationCommand', 'Update')
```

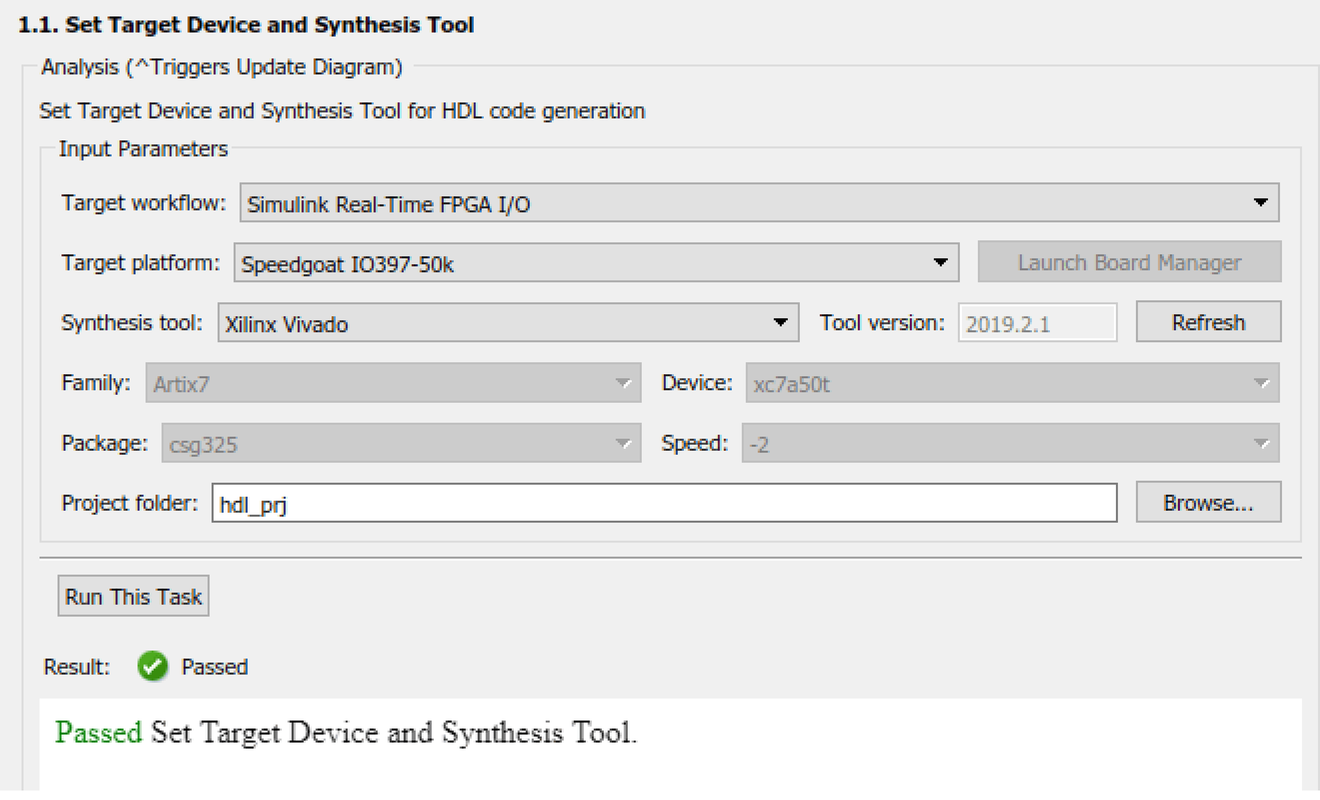


Generate Simulink Real-Time Interface Model for Speedgoat IO397 Platform

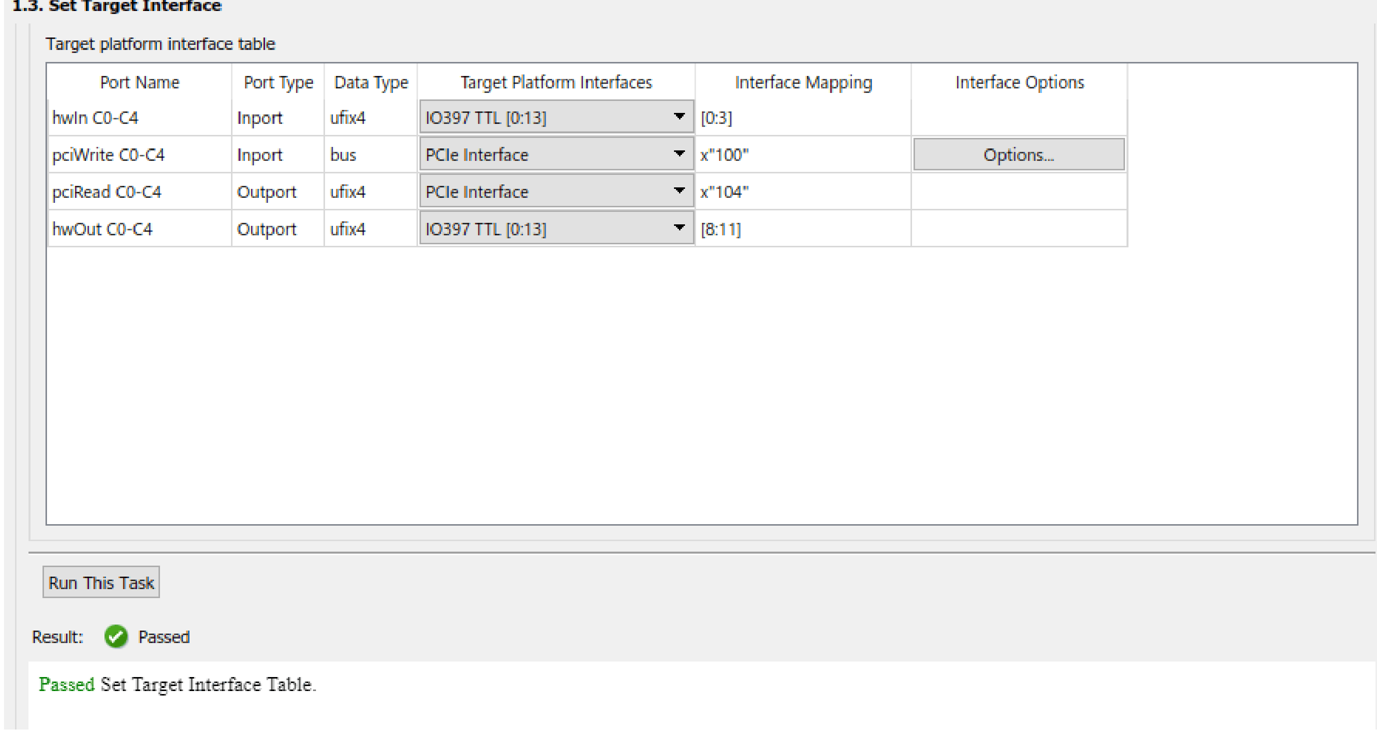
1. Open the HDL Workflow Advisor for the `loopback` subsystem. This subsystem is loaded on the FPGA.

```
hdladvisor('hdlcoder_slrt_loopback/loopback')
```

2. Expand the **Set Target** folder. In the **Set Target Device and Synthesis Tool** task, specify **Target workflow** as `Simulink Real-Time FPGA I/O` and **Target platform** as `Speedgoat IO397-50k`. Right-click the **Set Target Reference Design** task and select **Run to Selected Task**.



3. In the **Set Target Interface** task, map ports `hwIn` and `hwOut` to `IO397_TTL [0:13]` and `pciRead C0-C4` and `pciWrite C0-C4` to `PCIe interface`. Click **Run This Task**.



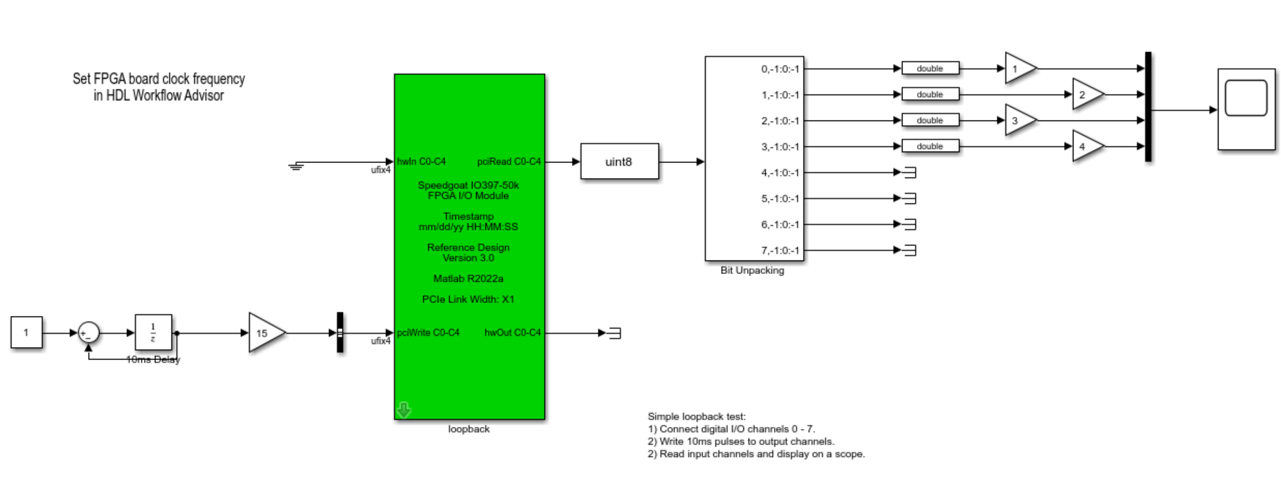
4. Run the **Set Target Frequency** task with the default value set for **Target Frequency (MHz)**. The target frequency must be in the range **Frequency Range (MHz)**.

5. Expand the **Download to Target** task. Right-click the **Generate Simulink Real-Time interface** task and select **Run to Selected Task**.

This task generates RTL code and IP core, FPGA bitstream, and the Simulink Real-Time Interface model. In the **Create Project** task, open the Vivado project to see the implemented block design.

Real-Time Subsystem Integration and Execution

After the **Generate Simulink Real-Time interface** task passes, click the link to open the Simulink Real-Time interface model.



The Simulink-Real Time Interface model contains a masked subsystem that has the same name as the subsystem in the Simulink FPGA domain model. This subsystem is the Simulink Real-Time Interface subsystem that contains the algorithm which is loaded onto the FPGA. Use the generated Simulink Real-Time Interface model or create a Simulink Real-Time Domain model and copy the Simulink Real-Time Interface subsystem into that model to simulate your FPGA algorithm on the Speedgoat target machine.

In the Simulink Real-Time interface subsystem mask, set three parameters:

- Device index
- PCI slot
- Sample time

When the target has a single FPGA I/O board, leave the device index to the default value. For multiple FPGA I/O boards, specify a unique device index. If two or more boards are of the same type, specify the PCI slot for each board.

For real-time testing, you can log the signals and view the simulation results on the Simulation Data Inspector.

1. On the **REAL-TIME** tab, open the Simulink Real-Time Explorer and specify the target interface connection settings. For an example, see “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 30-82.
2. On the **REAL-TIME** tab, click **Run on Target** to build and download the Simulink Real-Time application. The real-time application loads onto the Speedgoat target machine and the FPGA algorithm bitstream loads onto the FPGA.

You can then view the simulation results on the Simulation Data Inspector.

See Also

Related Examples

- “Processor and FPGA Synchronization” on page 39-38
- “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 40-145
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- Speedgoat I/O Examples

Model Design for AXI4-Stream Video Interface Generation

In this section...

"Sample Based Modeling" on page 40-118
"Protocol Signals and Timing Diagrams" on page 40-119
"Model Data and Control Bus Signals" on page 40-120
"Map DUT Ports to Multiple Channels" on page 40-124
"Model Designs with Multiple Sample Rates" on page 40-124
"Video Porch Insertion Logic" on page 40-124
"Default Video System Reference Design" on page 40-125
"Restrictions" on page 40-126
"Frame-Based Modeling" on page 40-126

For designs that require high speed video streaming use AXI4-Stream Video interfaces. Choose from these two modeling styles based on how your algorithm operates:

- **Sample-Based Modeling** — Use these guidelines when your algorithm operates on a stream of samples.
- **Frame-Based Modeling** — Use these guidelines when your algorithm operates on a complete frame of data. The data signals at the design under test (DUT) boundary must be matrices. Do not use this mode if you want to model the streaming pixel protocol.

With the HDL Coder software, you can implement a simplified, streaming pixel protocol in your model. The software generates an HDL IP core with AXI4-Stream Video interfaces.

Sample Based Modeling

With the HDL Coder software, you can implement a simplified, streaming pixel protocol in your model. The software generates an HDL IP core with AXI4-Stream Video interfaces. You can use the streaming pixel protocol for AXI4-Stream Video interface mapping. Video algorithms process data serially and generate video data as a serial stream of pixel data and control signals. To learn about the streaming pixel protocol, see "Streaming Pixel Interface" (Vision HDL Toolbox).

To generate an IP core with AXI4-Stream Video interfaces, in your DUT interface, implement these signals:

- Pixel Data
- Pixel Control Bus

The **Pixel Control Bus** is a bus that has these signals:

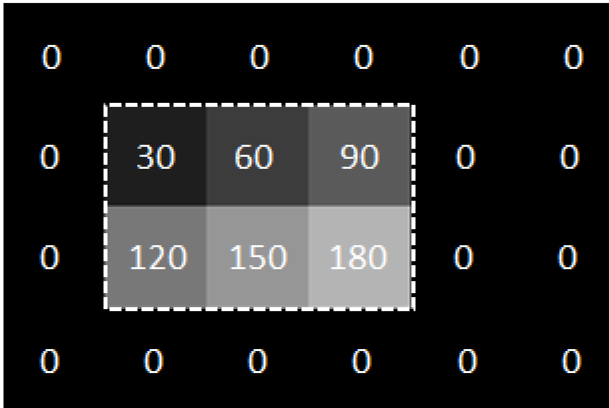
- `hStart`
- `hEnd`
- `vStart`
- `vEnd`
- `valid`

The signals **hStart** and **hEnd** represent the start of an active line and the end of an active line respectively. The signals **vStart** and **vEnd** represent the start of a frame and the end of a frame.

You can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

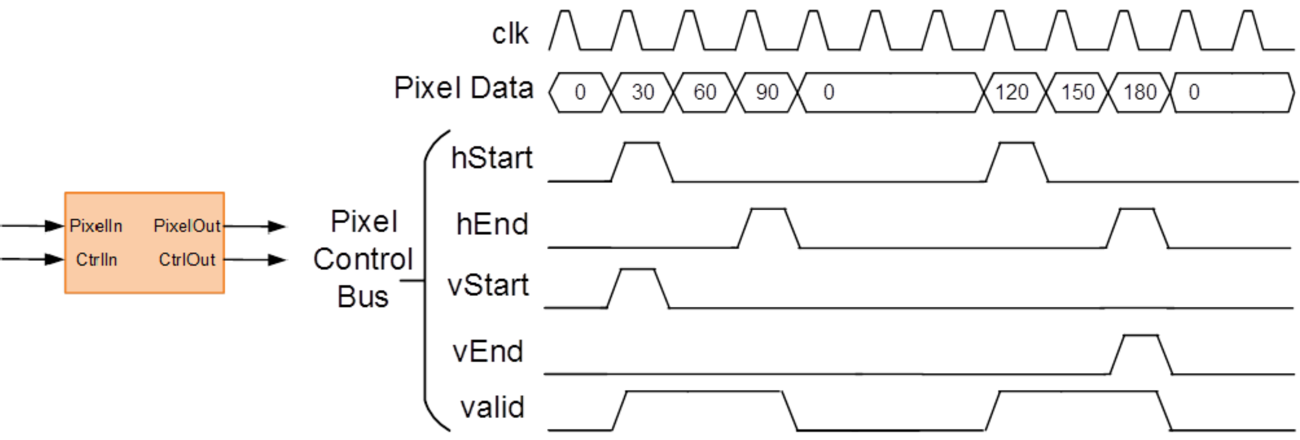
Protocol Signals and Timing Diagrams

This figure is a 2-by-3 pixel image. The active image area is the rectangle with a dashed line around it and the inactive pixels that surround it. The pixels are labeled with their grayscale values.



Pixel Data and Pixel Control Bus

This figure shows the timing diagram for the **Pixel Data** and **Pixel Control Bus** signals that you model at the DUT interface.



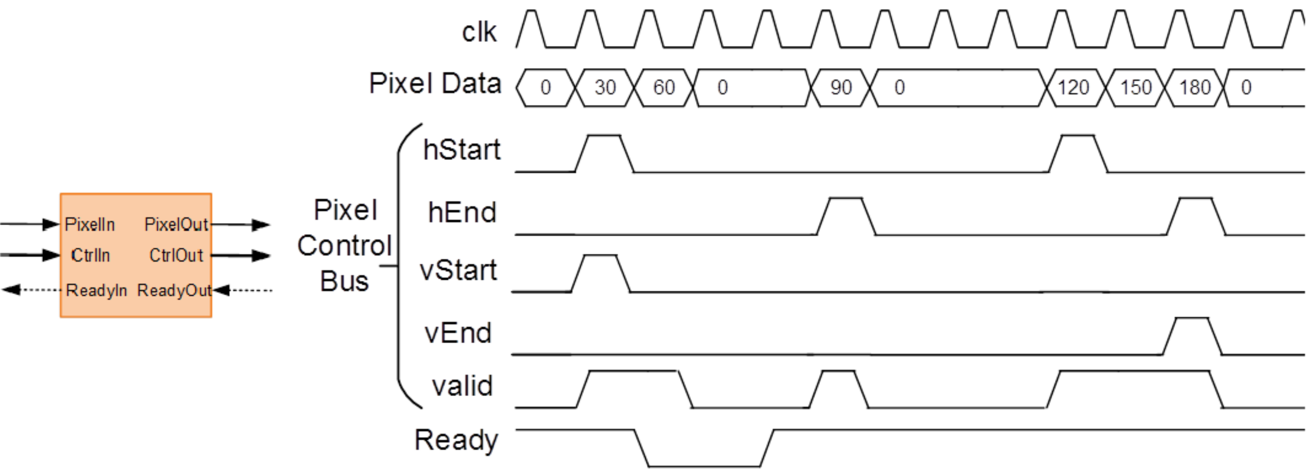
The **Pixel Data** signal is the primary video signal that is transferred across the AXI4-Stream Video interface. When the **Pixel Data** signal is valid, the **valid** signal is asserted.

The **hStart** signal becomes high at the start of the active lines. The **hEnd** signal becomes high at the end of the active lines.

The **vStart** signal becomes high at the start of the active frame in the second line. The **vEnd** signal becomes high at the end of the active frame in the third line.

Optional Ready Signal

This figure shows the timing diagram for the **Pixel Data**, the **Pixel Control Bus**, and the **Ready** signal that you model at the DUT interface.



When you map the DUT ports to an AXI4-Stream Video interface, you can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

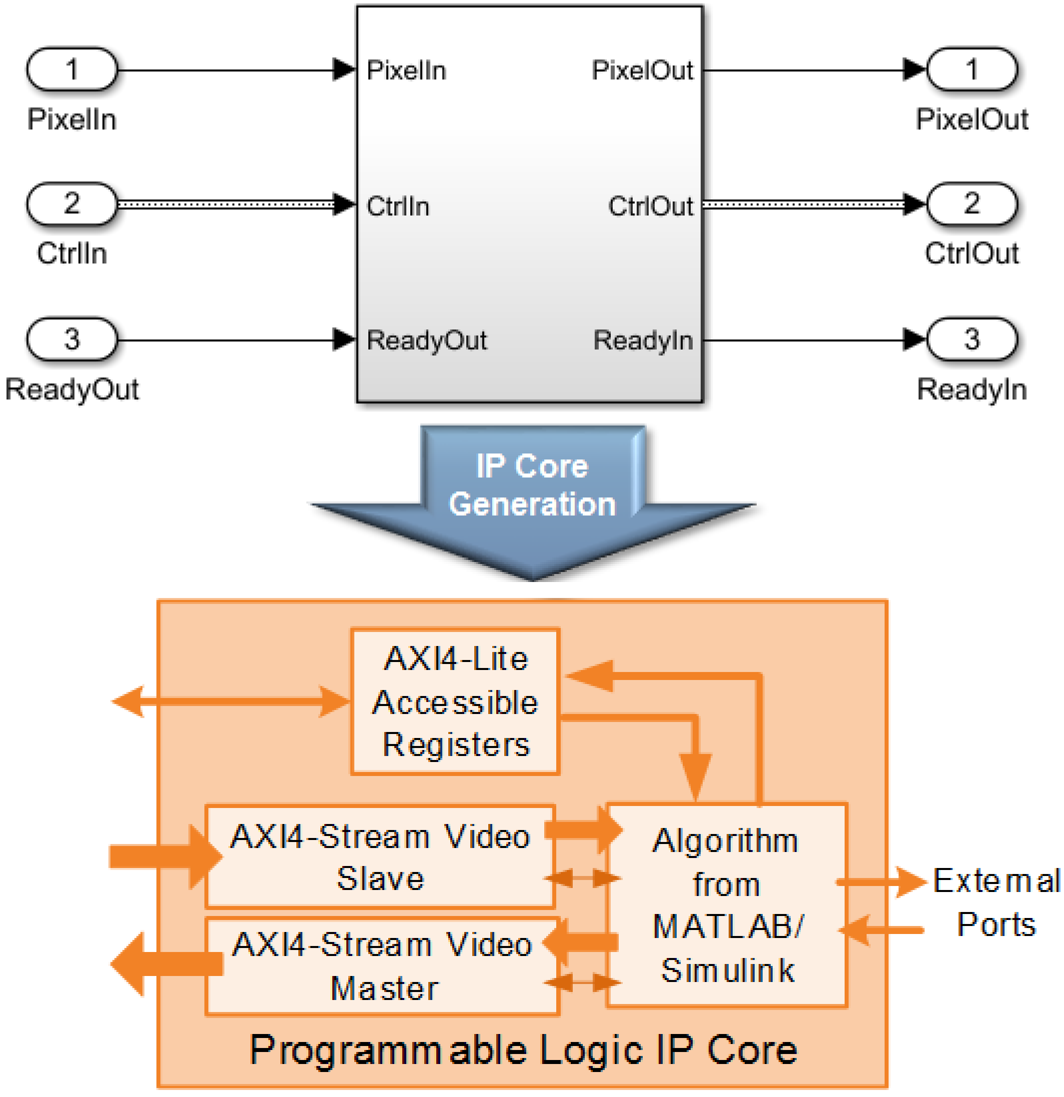
In a Slave interface, with the **Ready** signal, you can apply back pressure. In a Master interface, with the **Ready** signal, you can respond to back pressure.

If you model the **Ready** signal in your AXI4-Stream Video interfaces, your Master interface must deassert its **valid** signal one cycle after the **Ready** signal is deasserted.

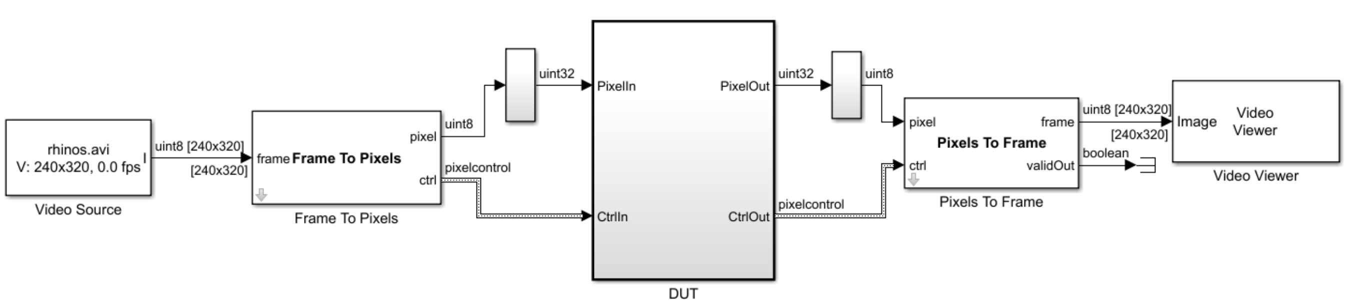
If you do not model the **Ready** signal, HDL Coder generates the associated backpressure logic.

Model Data and Control Bus Signals

You can model your video algorithm with **Pixel Data** and **Pixel Control Bus** signals at the DUT ports and map the signals to AXI4-Stream Video interfaces. You can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.



This figure shows an example of a top-level Simulink model with a Video Source input.

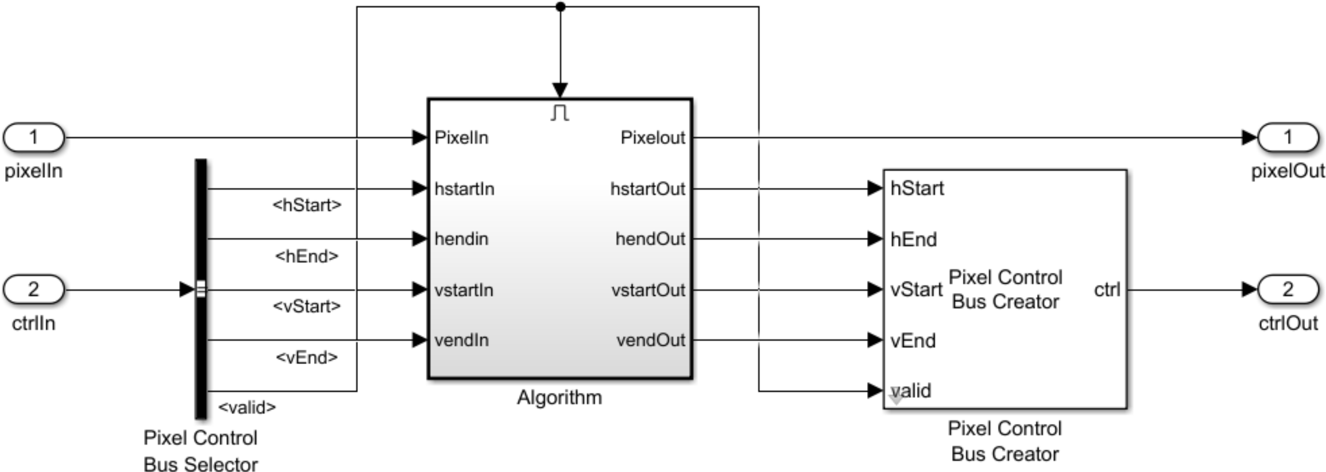


The Frame To Pixels and Pixels To Frame blocks perform the conversion between the video frames and the **Pixel Data** and **Pixel Control Bus** at the DUT interface. To use these blocks, you must have the Vision HDL Toolbox installed.

See also Frame To Pixels and Pixels To Frame.

Pixel Data and Pixel Control Bus Modeling

This figure shows how to model the **Pixel Data** and **Pixel Control Bus** signals inside the **DUT** subsystem.



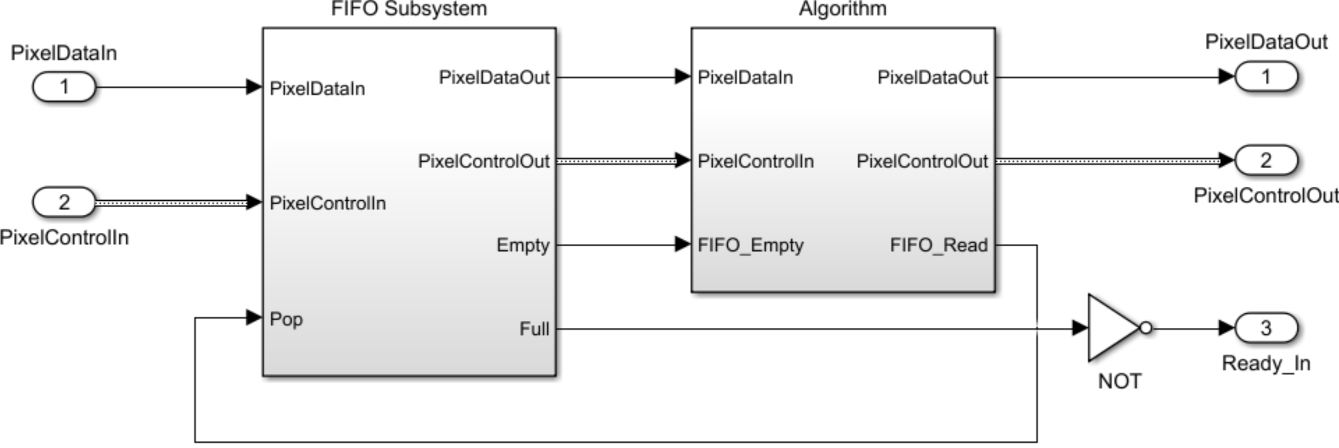
You can directly connect the **valid** signal from the **Pixel Control Bus** to the Enable port. If you do not have the Vision HDL Toolbox software, replace the Pixel Control Bus Selector and Pixel Control Bus Creator blocks with the Bus Selector and Bus Creator blocks respectively.

Ready Signal Modeling

The AXI4-Stream Video interfaces in your DUT can optionally include a **Ready** signal.

For example, you can have a FIFO in your DUT to store some video data before processing the signals. Use a **FIFO Subsystem** that contains HDL FIFO blocks to store the **Pixel Data** and the **Pixel Control Bus** signals. To apply the backpressure to the upstream component, model the **Ready** signal based on the FIFO Full signal.

This figure shows how to model the **Ready** signal inside the **DUT** subsystem.



The **FIFO Subsystem** block uses HDL FIFO blocks for the **Pixel Data** and for the **Pixel Control Bus** signals.

Disable delay balancing for the **Ready** signal path. If you enable delay balancing, the coder can insert one or more delays on the **Ready** signal.

Map DUT Ports to Multiple Channels

When you run the `IP Core Generation` workflow, you can map multiple DUT ports to AXI4-Stream Video Master and AXI4-Stream Video Slave channels. The DUT ports mapped to multiple interface channels must use scalar data type. When you use vector ports, you can map the ports to at most one AXI4-Stream Video Master channel and one AXI4-Stream Video Slave channel.

To learn more, see “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 40-33.

Model Designs with Multiple Sample Rates

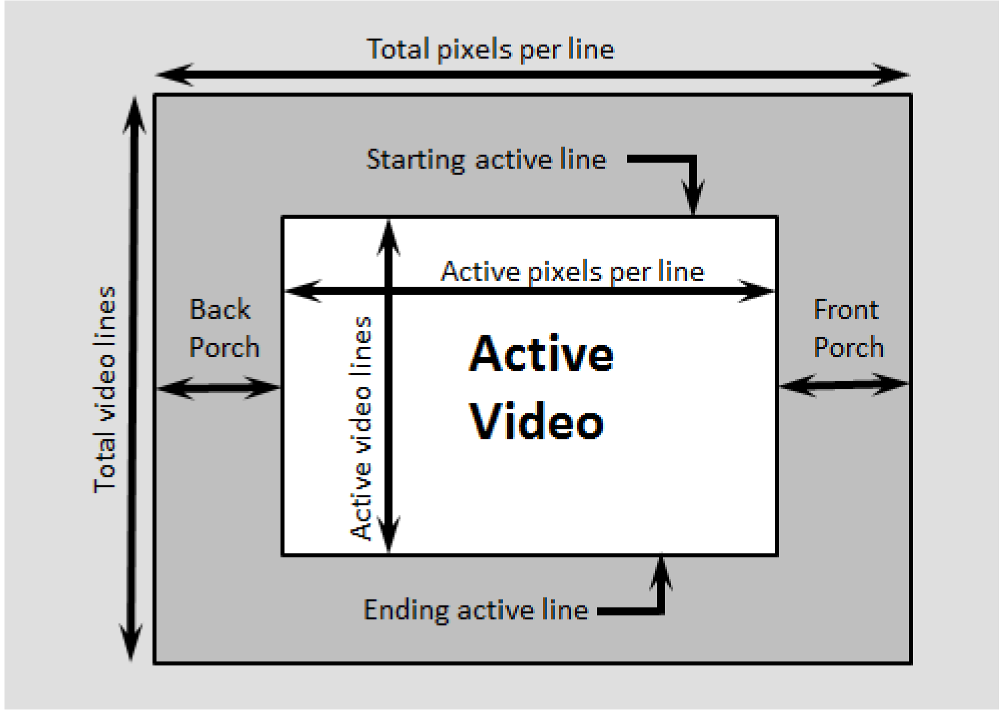
The HDL Coder software supports designs with multiple sample rates when you run the IP Core Generation workflow. When you map the interface ports to AXI4-Stream Video Master or AXI4-Stream Video Slave interfaces, to use multiple sample rates, ensure that the DUT ports that map to these AXI4 interfaces run at the fastest rate of the design after HDL code generation.

To learn more, see “Multirate IP Core Generation” on page 40-85.

Video Porch Insertion Logic

Video capture systems scan video signals from left to right and from top to bottom. As these systems scan, they generate inactive intervals between lines and frames of active video. This inactive interval is called a video porch. The horizontal porch consists of inactive cycles between the end of one line and the beginning of next line. The vertical porch consists of inactive cycles between the ending active line of one frame and the starting active line of next frame.

This figure shows a video frame with the horizontal porch split into a front and a back porch.



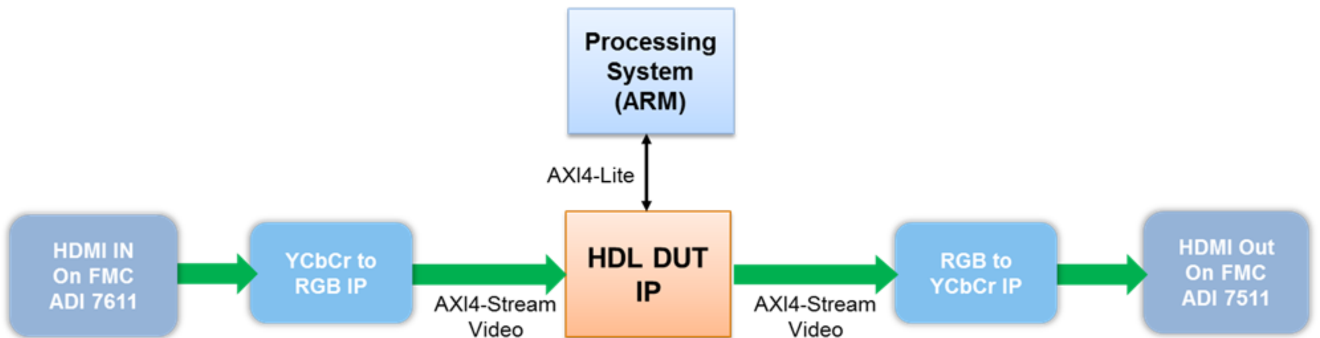
The AXI4-Stream Video interface does not require a video porch, but Vision HDL Toolbox algorithms require a porch for processing video streams. If the incoming pixel stream does not have a sufficient porch, HDL Coder inserts the required amount of porch to the pixel stream. By using the AXI4-Lite registers in the generated IP core, you can customize these porch parameters for each video frame:

- Active pixels per line (Default: 1920)
- Active video lines: (Default: 1080)
- Horizontal porch length (Default: 280)
- Vertical porch length (Default: 45)

Default Video System Reference Design

You can integrate the generated HDL IP core with AXI4-Stream Video interfaces into the `Default video system` reference design.

This figure is a block diagram of the `Default video system` reference design architecture.



You can use this `Default video system` reference design architecture with these target platforms:

- `Xilinx Zynq ZC702 evaluation kit`
- `Xilinx Zynq ZC706 evaluation kit`
- `ZedBoard`

To use the `Default video system` reference design, you must install the Support Package for SoC Blockset Support Package for Xilinx Devices.

Restrictions

When you map the DUT ports to AXI4-Stream Video interfaces:

- The DUT port mapped to the **Pixel Data** signal must use a scalar data type.
- Xilinx Zynq-7000 must be your target platform.
- You must use Xilinx Vivado as your synthesis tool.
- **Processor/FPGA synchronization** must be `Free running`.

Frame-Based Modeling

Model your DUT algorithm to operate on frames of data and map your matrix port to an AXI4-Stream Video interface by using the frame-to-sample optimization. For more information, see “Model Design for Frame-Based IP Core Generation” on page 40-29.

See Also

More About

- “Model Design for AXI4-Stream Interface Generation” on page 40-14
- “Streaming Pixel Interface” (Vision HDL Toolbox)

See Also

Related Examples

- “Deploy Model with AXI4-Stream Video Interface on Zynq Hardware” on page 40-207

Model Design for AXI4 Master Interface Generation

In this section...

“Simplified AXI4 Master Protocol - Write Channel” on page 40-128
“Simplified AXI4 Master Protocol - Read Channel” on page 40-130
“Base Address Register Calculation” on page 40-132
“Specify Initial Value of AXI4 Master Read and Write Base Address” on page 40-132
“Modeling for AXI4 Master Interfaces” on page 40-132
“Map Vector Ports to AXI4 Master Interfaces” on page 40-134
“Model ID Signals to Reduce the Number of AXI-4 Master Interfaces” on page 40-136
“Model Designs with Multiple Sample Rates” on page 40-138
“Reference Designs for IP Core Integration” on page 40-138
“Restrictions” on page 40-139

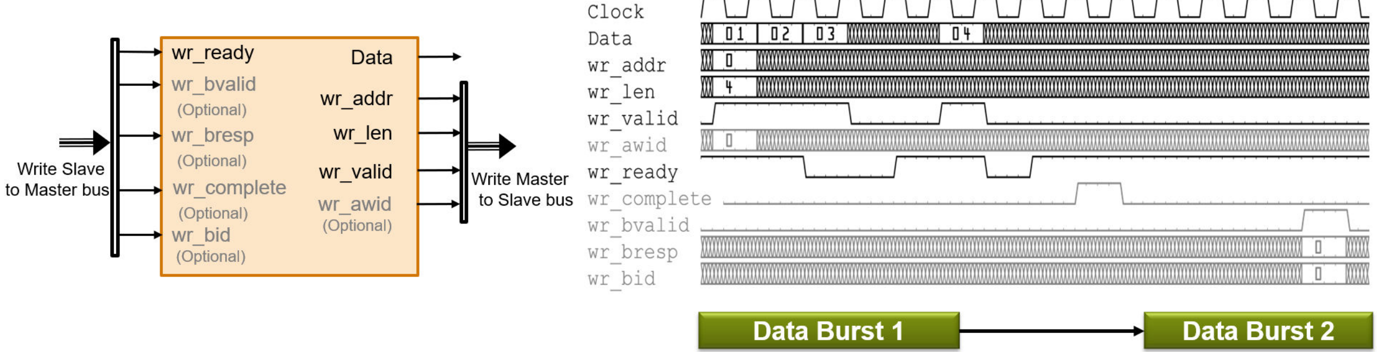
For designs that require accessing large data sets from an external memory, model your algorithm with a simplified AXI4 Master protocol. When you run the `IP Core Generation` workflow, HDL Coder generates an IP core with AXI4 Master interfaces. The AXI4 Master interface can communicate between your design and the external memory controller IP by using the AXI4 Master protocol. Use the AXI4 Master interface when your:

- Design targets multi-frame video processing applications. You can store the image data in external memory, such as a DDR3 memory on board, and then read or write the images to your design in a burst fashion for high-speed processing.
- Algorithm must access memory data in a non-streaming arbitrary pattern.
- DUT IP core must control other IPs with the AXI4 slave interface in the system. This capability is especially useful in standalone FPGA devices.

Simplified AXI4 Master Protocol - Write Channel

To map the DUT ports to AXI4 Master interfaces, use the simplified AXI4 Master protocol. You do not have to model the actual AXI4 Master protocol. Instead, you can use the simplified protocol. When you run the `IP Core Generation` workflow, the generated HDL code contains wrapper logic that translates between the simplified protocol and the actual AXI4 Master protocol. The simplified protocol requires fewer protocol signals, eases the handshaking mechanism between valid and ready signals, and supports bursts of arbitrary lengths.

For a write transaction, use the simplified AXI4 Master write protocol. For a read transaction, use the simplified AXI4 Master read protocol. This figure shows the timing diagram for the signals that you model at the DUT input and output interfaces for an AXI4 Master write transaction.



The DUT waits for `wr_ready` to become high to initiate a write request. When `wr_ready` becomes high, the DUT can send the write request. The write request consists of the `Data` and `Write Master to Slave bus` signals. This bus consists of `wr_len`, `wr_addr`, and `wr_valid`. The `wr_addr` signal specifies the starting address that the DUT writes to. The `wr_len` signal corresponds to the number of data elements in this write transaction. `Data` can be sent when `wr_valid` is high. When `wr_ready` becomes low, the DUT must stop sending data within one clock cycle, and the `Data` signal becomes invalid. If the DUT continues to send data after one clock cycle, the data is ignored.

The simplified AXI4 Master Protocol supports pipelined requests. The protocol is not required to wait for the `wr_complete` signal to be high before issuing a subsequent write request. The interface supports up to 16 transactions (or 16 data words) before the pipeline stalls and the `wr_ready` signal goes low.

Output Signals

Model the `Data` and `Write Master to Slave bus` signals at the DUT output interface.

- `Data`: The data that you want to transfer, valid each cycle of the transaction.
- `Write Master to Slave bus` that consists of:

  - `wr_addr`: Starting address of the write transaction that is sampled at the first cycle of the transaction. The address is specified in bytes.
  - `wr_len`: The number of data values that you want to transfer, sampled at the first cycle of the transaction. The `wr_len` signal is specified in words, meaning that each unit of `wr_len` is a complete data element. For example, when `wr_len` is `2`, and the bit width of data is `128` bit, two `128`-bit data elements are written.
  - `wr_valid`: When this control signal becomes high, it indicates that the `Data` signal sampled at the output is valid.
  - `wr_awid` (optional signal): This signal is the address ID that identifies multiple streams over a single channel. When you do not map `wr_awid`, the generated IP core sets `wr_awid` to a constant 0.

Input Signals

Model the `Write Slave to Master bus` that consists of:

- `wr_complete` (optional signal): Control signal that when high for one clock cycle, indicates that the write transaction has completed. The next burst of data can be sent after `wr_complete` asserts. The early assertion of `wr_complete` makes the average latency nearly `3` clock cycles

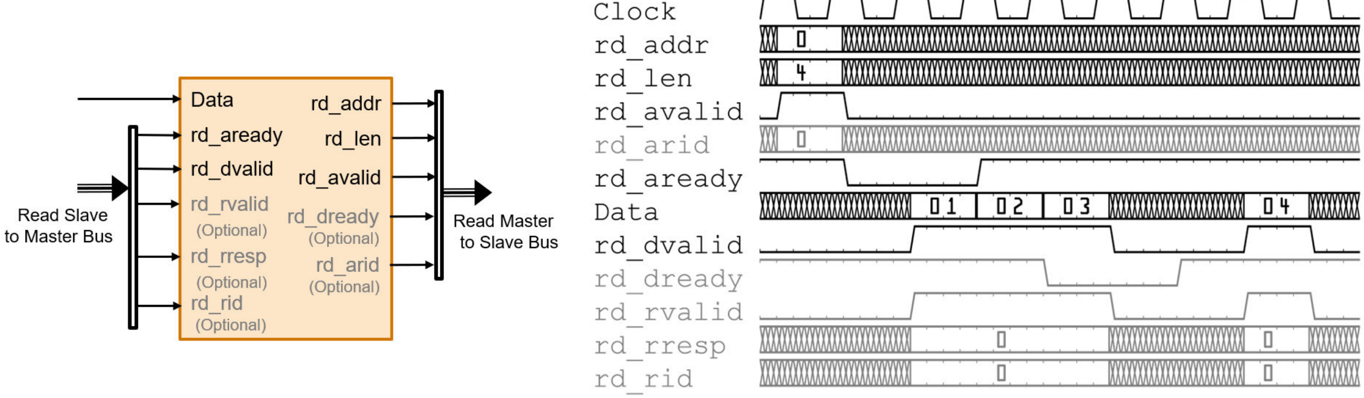
between two bursts, which makes the write operation pipelined and improves the write throughput.

- wr_ready: This signal corresponds to the back pressure from the slave IP core or external memory. When this control signal goes high, it indicates that data can be sent. When wr_ready is low, the DUT must stop sending data within one clock cycle. You can also use the wr_ready signal to determine whether the DUT can send a second burst signal immediately after the first burst signal has been sent. Multiple burst signals are supported, which means that the wr_ready signal remains high to accept the second burst immediately after the last element of the first burst has been accepted. Using wr_ready to determine when to start the next burst can reduce the average latency between two bursts to less than 3 clock cycles.
- wr_bvalid (optional signal): Response signal from the slave IP core that you can use for diagnosis purposes. The wr_bvalid signal becomes high after the AXI4 interconnect accepts each burst transaction. If wr_len is greater than 256, the AXI4 Master write module splits the large burst signal into 256-sized bursts. The wr_bvalid signal becomes high for each 256-sized burst.
- wr_bresp (optional signal): Response signal from the slave IP core that you can use for diagnosis purposes. Use this signal with the wr_bvalid signal.
- wr_bid (optional signal): This signal is the write response ID that identifies multiple streams over a single channel. When you do not map wr_bid, the generated IP core terminates wr_bid.

The AXI4 Master protocol supports a maximum burst size of 256. When you have a large burst size of greater than 256, the AXI Master interface in the generated HDL IP core divides the large burst into multiple smaller bursts with size 256. Even for large bursts of data, you see an improved write throughput.

Simplified AXI4 Master Protocol - Read Channel

This figure shows the timing diagram for the signals that you model at the DUT input and output interfaces for an AXI4 Master read transaction. These signals include the Data, Read Master to Slave Bus, and Read Slave to Master Bus.



The DUT waits for rd_aready to become high to initiate a read request. When rd_aready is high, the DUT can send out the read request. The read request consists of the rd_addr, rd_len, and rd_avalid signals of the Read Master to Slave bus. The slave IP or the external memory responds to the read request by sending the Data at each clock cycle. The rd_len signal

corresponds to the number of data values to read. The DUT can receive `Data` as long as `rd_dvalid` is high.

Read Request

To model a read request, at the DUT output interface, model the `Read Master to Slave bus` that consists of:

- `rd_addr`: Starting address for the read transaction that is sampled at the first cycle of the transaction. The address is specified in bytes.
- `rd_len`: The number of data values that you want to read, sampled at the first cycle of the transaction. The `rd_len` signal is specified in words, meaning each unit of `rd_len` is a complete data element. For example, when `rd_len` is `2`, and the bit width of data is `128` bit, two `128`-bit data elements are read.
- `rd_avalid`: Control signal that specifies whether the read request is valid.
- `rd_arid` (optional signal): This signal is the address ID that identifies multiple streams over a single channel. When you do not map `rd_arid`, the generated IP core sets `rd_arid` to a constant 0.

At the DUT input interface, you can model the ready signal `rd_aready` to indicate when to send a new read request. The `rd_aready` signal indicates whether the DUT can send consecutive multiple burst requests. The DUT can send a new request one clock cycle after the ready signal is asserted. After the ready signal is de-asserted, the DUT can send one more read request. The DUT cannot send a new read request and the request is ignored one clock cycle after the ready signal is de-asserted.

The simplified AXI4 Master Protocol supports pipelined requests, so it is not required to wait for the read response to complete before issuing a subsequent read request. The interface supports up to four read transactions before the pipeline stalls and the `rd_aready` signal goes low.

Read Response

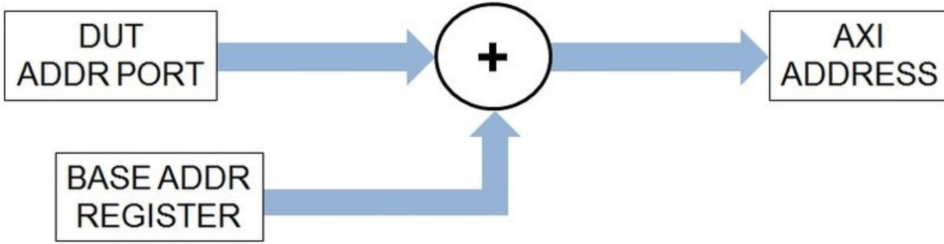
At the DUT input interface, model the `Data` and `Read Slave to Master bus` signals.

- `Data`: The data that is returned from the read request.
- `Read Slave to Master bus` that consists of:
  - `rd_dvalid`: Control signal, which indicates that the `Data` returned from the read request is valid.
  - `rd_rvalid` (optional signal): Response signal from the slave IP core that you can use for diagnosis purposes.
  - `rd_rresp` (optional signal): Response signal from the slave IP core that indicates the status of the read transaction.
  - `rd_rid` (optional signal): This signal is the data ID that identifies multiple streams over a single channel. When you do not map `rd_rid`, the generated IP core terminates `rd_rid`.

At the DUT output interface, you can optionally implement the `rd_dready` signal. This signal is part of the `Read Master to Slave bus` and indicates when the DUT can start accepting data. By default, if you do not map this signal to the AXI4 Master read interface, the generated HDL IP core ties the `rd_dready` signal to logic high.

Base Address Register Calculation

For IP cores that you generate, HDL Coder includes a base address register to support driver authoring for both the AXI4 Master read and write channels. The base address register is added to the address that is specified by the `DUT ADDR` port to form the AXI4 Master address. This capability enables the driver to use an addressing mode that programs a fixed register address with the base address of a buffer. The programmed address together with the `DUT ADDR` port is used to index the buffer. By default, the registers take a value of zero, if you do not use them.



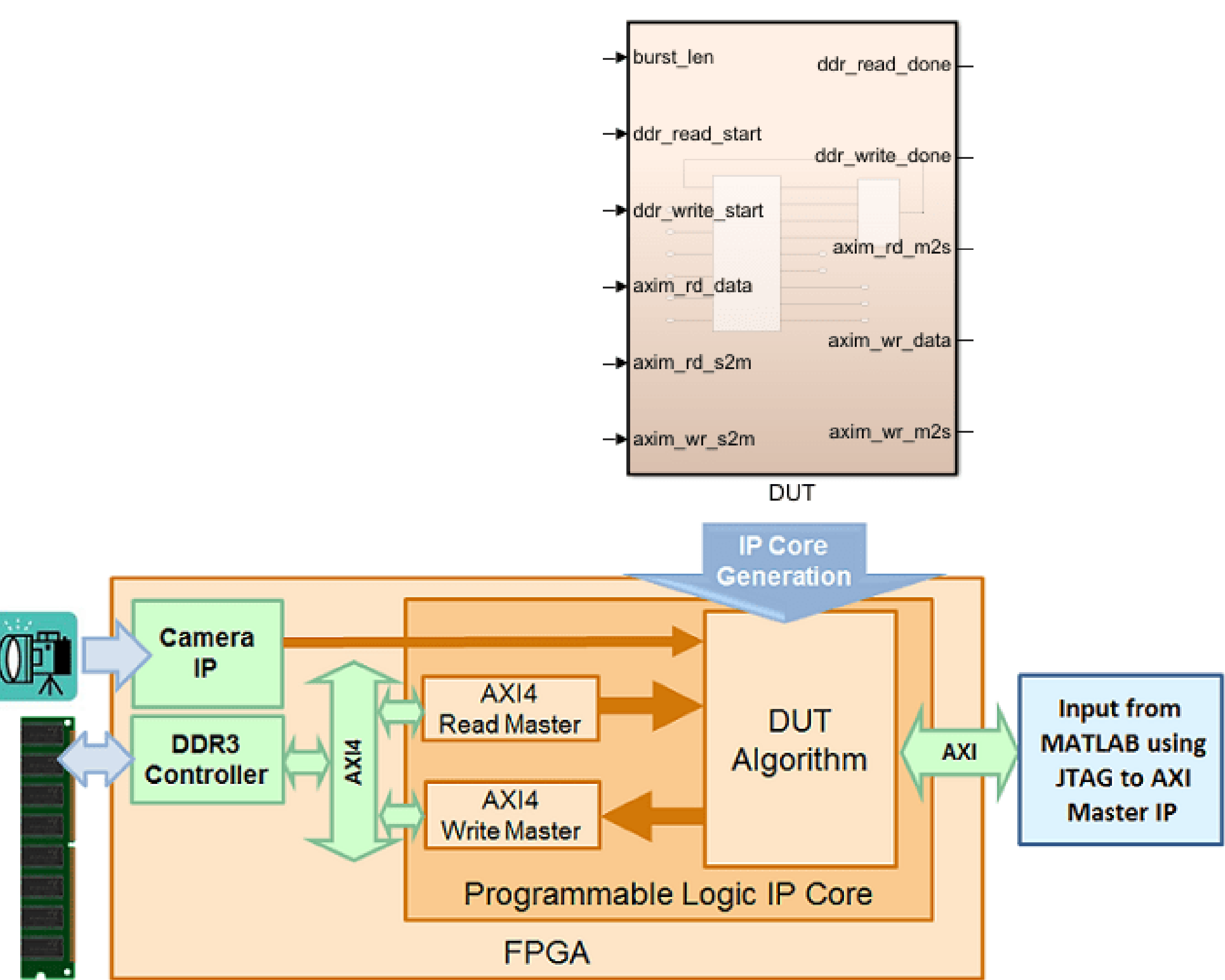
Specify Initial Value of AXI4 Master Read and Write Base Address

When you run the `IP Core Generation` workflow or the `Simulink Real-Time FPGA I/O` workflow, you can specify an initial value for the AXI4 master data read and write base address registers. By default, the initial value is zero. To specify a nonzero value:

1. In the target platform interface table, when you map an input DUT port to an AXI4 Master Read data port, or an output DUT port to an AXI4 Master Write data port interface, an **Options** button appears in the **Interface Options** column.
2. Click the **Options** button, and then specify the **DefaultReadBaseAddress** or **DefaultWriteBaseAddress**.

Modeling for AXI4 Master Interfaces

You can model your algorithm with Data and AXI4 Master protocol signals at the DUT ports and then map the signals to AXI4 Master interfaces.



To learn how to model your DUT algorithm for AXI4 Master interface mapping, open this Simulink® model. The DUT Subsystem contains a simple algorithm that reads data from the DDR and writes the data back to a different address in the DDR memory.

```
open_system('hdlcoder_axi_master')
sim ('hdlcoder_axi_master')
```

Double-click the DUT Subsystem. The DDR_Access_Controller Subsystem models the AXI Master read and write channels and has a Simple Dual Port RAM that calculates the wr_data signal. If you double-click the DDR_Access_Controller Subsystem, you see two Edge Detection Subsystem blocks that generate the two start pulses as input to each MATLAB Function block. One Edge Detection Subsystem and DDR Read Controller MATLAB Function models the read transaction. The other Edge Detection Subsystem and DDR Write Controller MATLAB Function models the write transaction. You can modify this design to model only the write transaction or the read transaction by using one Edge Detection Subsystem and the corresponding MATLAB Function block.

Read Channel

The DDR Read Controller is modeled as a state machine with four states: INIT, IDLE, READ_BURST_START, and DATA_COUNT. The INIT state initializes the read signals and the RAM input signals. When the start signal goes high, the state machine switches to the IDLE state, and then waits for the `rd_aready` signal to become high. When `rd_aready` becomes high, the state machine transitions to the READ_BURST_START state and the DUT starts reading data. The state machine then unconditionally switches to the DATA_COUNT state and continues to read data till `rd_avalid` goes low.

Write Channel

The DDR Write Controller is modeled similar to the Read channel as a state machine with four states: IDLE, WRITE_BURST_START, DATA_COUNT, and ACK_WAIT. The DUT is in the IDLE state and then switches to the WRITE_BURST_START state where it waits for the `wr_ready` signal. When `wr_ready` becomes high, the state machine switches to the DATA_COUNT state and starts writing data. The data is valid when `wr_valid` is high. The DUT continues to write data when `wr_ready` is high. As `wr_ready` becomes low, the state machine switches to the ACK_WAIT state and then waits for the ready signal to initiate the next write transaction.

To see the simplified AXI4 Master protocol in effect, simulate the model. If you have DSP System Toolbox™ installed, you can view and analyze the results in the Logic Analyzer.

<<../axi_master_read_waveform.png>>

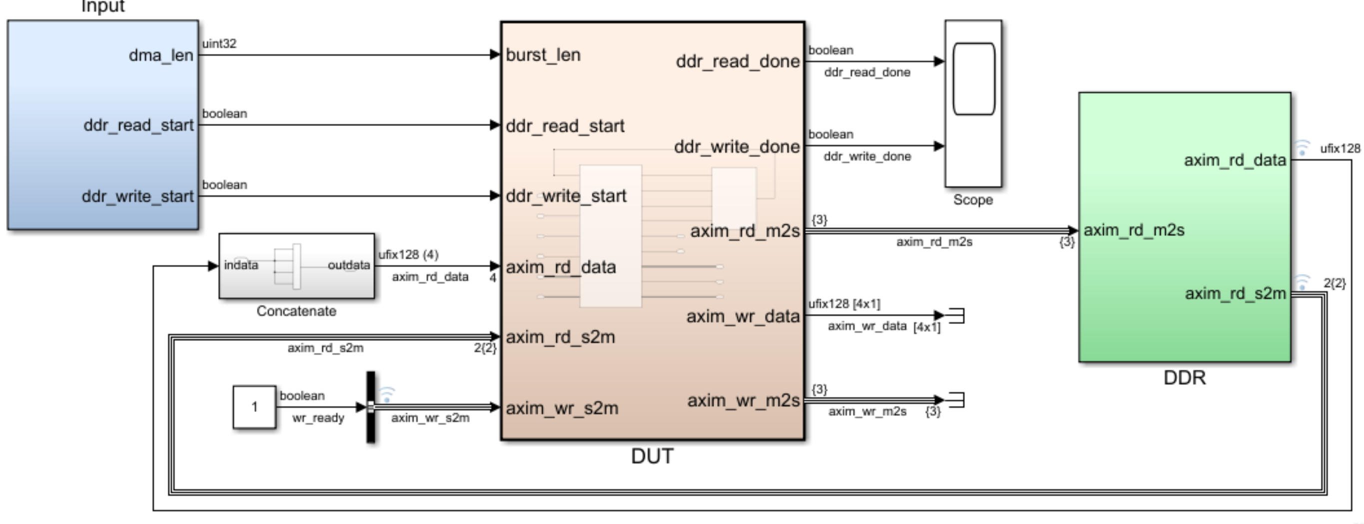
You can use the IP Core Generation workflow to generate an HDL IP core with the AXI4 Master interface. If you have HDL Verifier™ installed, and you use the Xilinx® Zynq® ZC706 board, then you can integrate the IP core into the Default System with External DDR3 memory access reference design.

Map Vector Ports to AXI4 Master Interfaces

To integrate your HDL IP core into larger reference designs, and to achieve higher throughput when you use the AXI4 Master port to access external DDR memory, you may want to use larger bit widths on the `Data` port. The AXI4 Master interface bus supports a maximum bit width of `1024` bits.

Simulink supports fixed-point data types that have word length of up to `128` bits. To model your DUT ports with word lengths greater than `128` bits, use vector data types. If you use a vector port such that the combined bit width of all the elements in the vector is greater than `1024` bits, the **Set Target Interface** task displays an error.

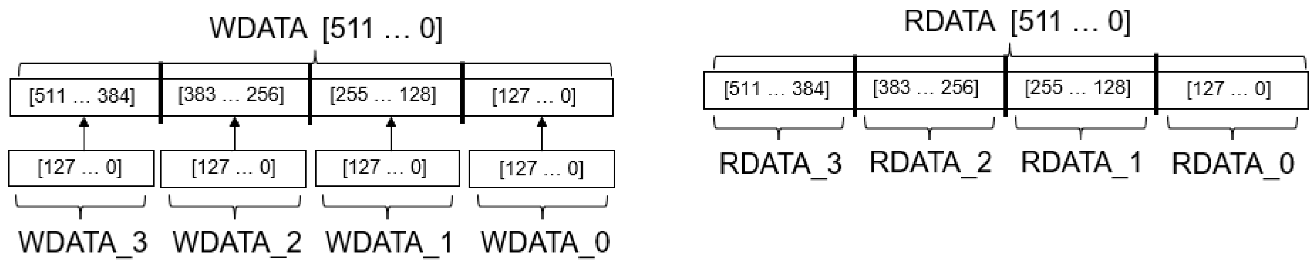
For example, in the `hdlcoder_axi_master` model, to expand the bit width of the `axim_rd_data` port to `512` bits, change the `ddr_data` parameter inside the DDR to `fi(([40:-1:1]),0,128,0)` and then concatenate the 128-bit input four times to generate an output of 512 bits. You can use a Vector Concatenate block to output a combined bit width of 512 bits. To simulate the model, replace the Simple Dual Port RAM block inside the DUT subsystem with a Simple Dual port RAM System.



You can then map these DUT Data ports to `AXI4 Master Read` or `AXI Master Write` ports in the Target platform interface table, generate the HDL IP core, and integrate the IP core into your Vivado or Qsys reference designs. In the generated HDL code for the DUT IP core, the `Data` ports are mapped to `512`-bit interfaces. Multiple FIFO blocks are generated corresponding to each element of the vector input.

```
ENTITY DUT_ip IS
  PORT( IPCORE_CLK            :   IN    std_logic;  -- ufix1
        IPCORE_RESETN         :   IN    std_logic;  -- ufix1
        AXI4_Master_Rd_RDATA  :   IN    std_logic_vector(511 DOWNTO 0);  -- ufix256

        ...
        ...

        AXI4_Master_Wr_WDATA  :   OUT   std_logic_vector(511 DOWNTO 0);  -- ufix256

        ...

        );
END DUT_ip;
```

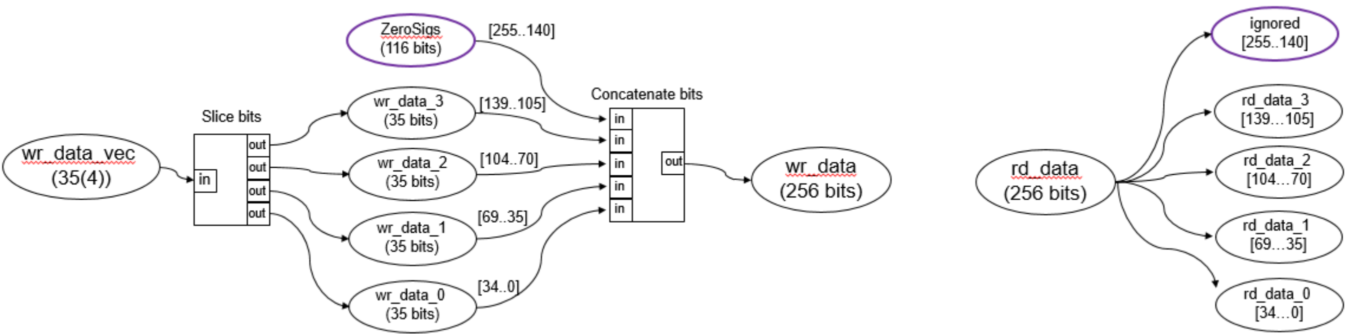
This figure illustrates the order in which the vector data is written to and read form.



In the HDL code for the DUT IP core, you can see how the `AXI4_Master_Rd_RDATA` and `AXI4_master_Wr_WDATA` interfaces are mapped to the DUT ports and the order in which data is written to the AXI4 Master interface and then read back.

```
...
...
--------------------------------------------------------------------
AXI4 Master Read Sequence
--------------------------------------------------------------------
AXI4_Master_Rd_RDATA_0 <= AXI4_Master_Rd_RDATA_unsigned(127 DOWNTO 0);

AXI4_Master_Rd_RDATA_1 <= AXI4_Master_Rd_RDATA_unsigned_1(255 DOWNTO 128);

AXI4_Master_Rd_RDATA_2 <= AXI4_Master_Rd_RDATA_unsigned_7(383 DOWNTO 256);

AXI4_Master_Rd_RDATA_3 <= AXI4_Master_Rd_RDATA_unsigned_7(511 DOWNTO 384);

--------------------------------------------------------------------
AXI4 Master Write Sequence
--------------------------------------------------------------------
AXI4_Master_Wr_WDATA_tmp <= unsigned(AXI4_Master_Wr_WDATA_Vec_3) &
                            unsigned(AXI4_Master_Wr_WDATA_Vec_2) &
                            unsigned(AXI4_Master_Wr_WDATA_Vec_1) &
                            unsigned(AXI4_Master_Wr_WDATA_Vec_0);


AXI4_Master_Wr_WDATA <= std_logic_vector(AXI4_Master_Wr_WDATA_tmp);

...
...
```

If you use a nonstandard bit width for the AXI4 Master `Data` port, the `Data` port is upgraded to a standard bit width container that has a bigger size. Standard bit widths include `32`, `64`, `128`, `256`, `512`, and `1024` bits. For example, if you use a vector that has four `35`-bit elements, the resulting bit width of `140` bits (`35x4`) is mapped to a `256`-bit AXI4 Master interface. At the Write channel `Data` port, bits `255` to `141` are padded with zeroes. At the Read channel `Data` port, bits `255` to `141` are ignored.



Using nonstandard bit widths can have a performance impact because the entire bandwidth of the AXI4 Master interface is not used. To avoid performance hits, use standard AXI bit widths.

Model ID Signals to Reduce the Number of AXI-4 Master Interfaces

You can model ID signals for the Simplified AXI4 Master Protocol in the IP core generation workflow. Use ID signals to:

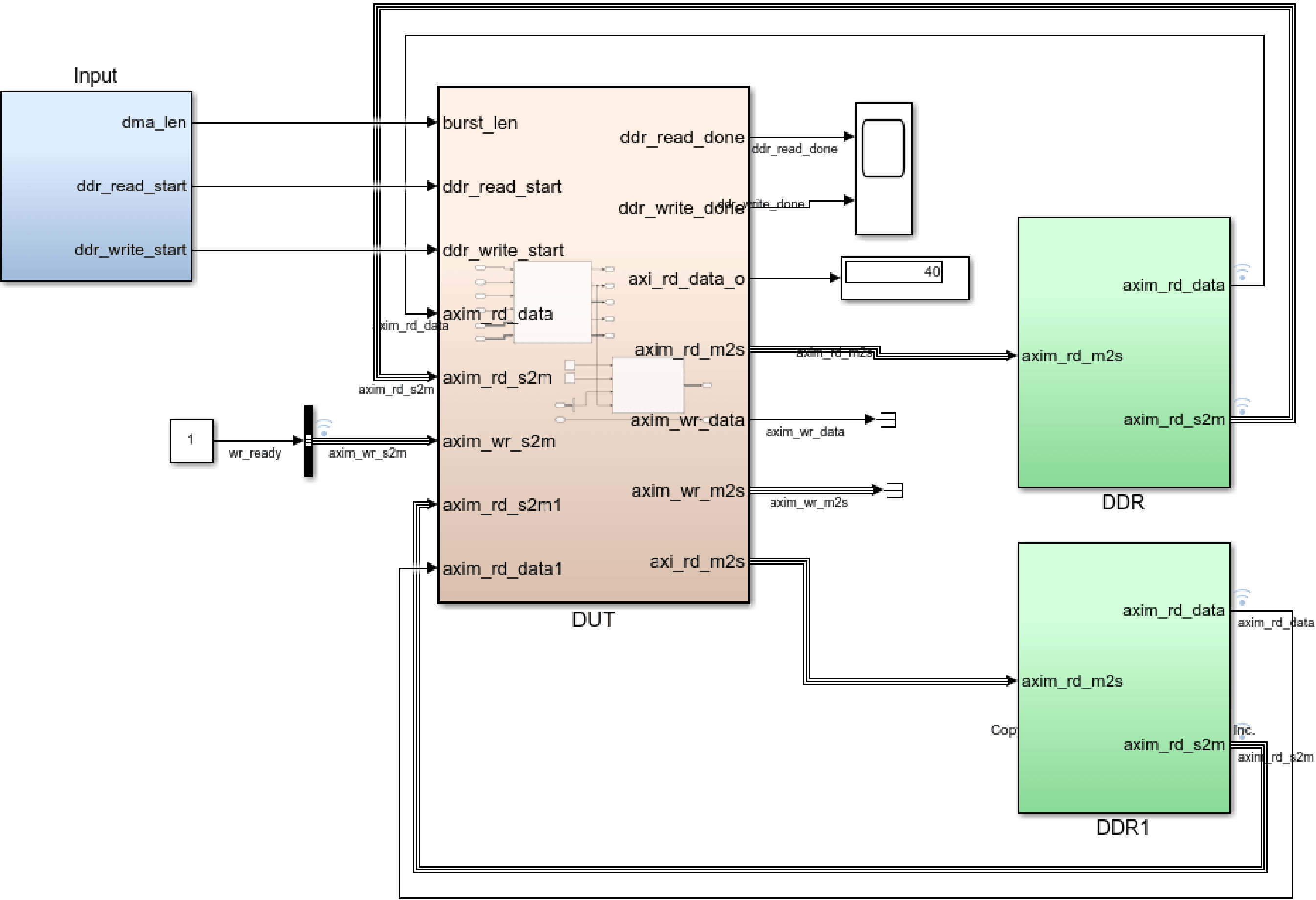
- Allow multiple modules to share the same AXI Master Interface.
- Enable out-of-order requests for data. ID signals allow an AXI Master to issue requests without waiting for a prior request to finish.

- Reduce the amount of AXI4 Master Interfaces needed for a Simulink model.

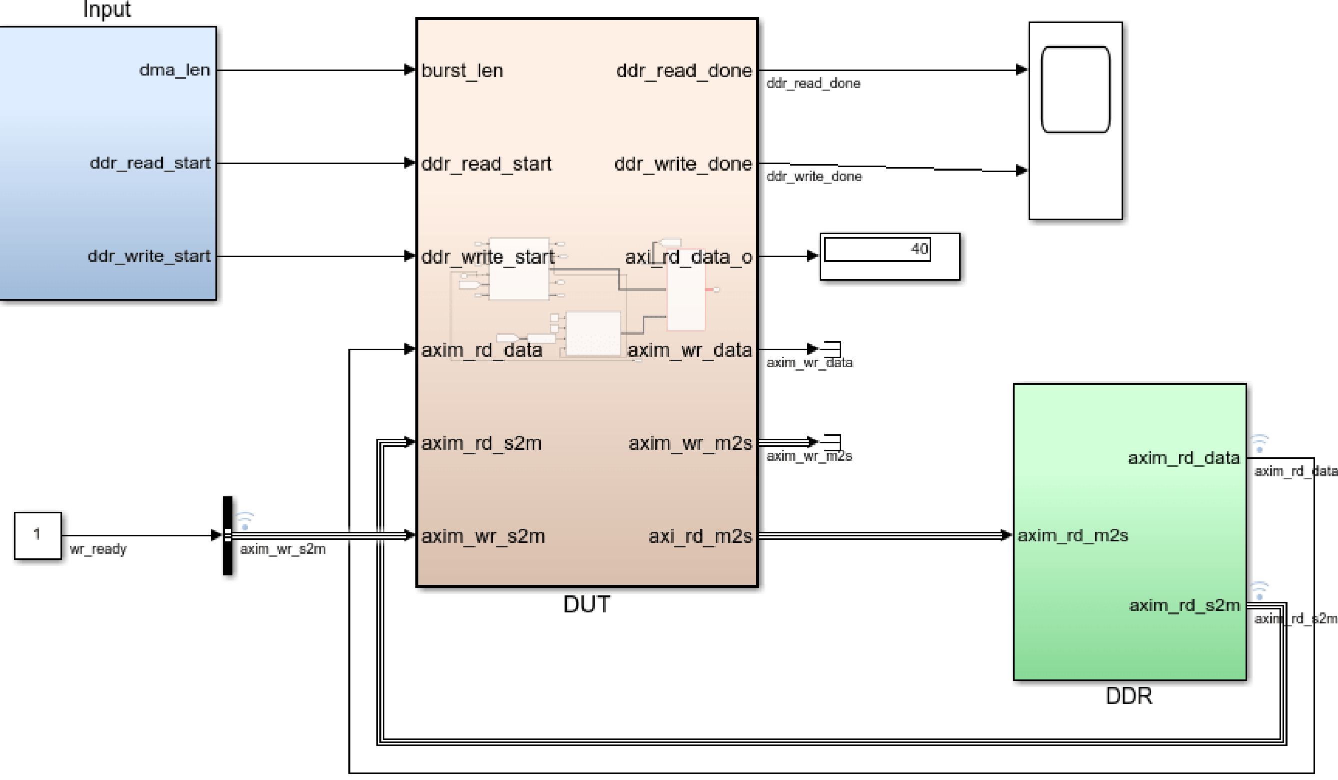
You can model ID signals in a write channel by using the optional signals `wr_awid` and `wr_bid`. You can model ID signals in a read channel by using the optional signals `rd_arid` and `rd_rid`. ID signal widths have these limitations:

- The ID widths must be an unsigned integer having a maximum of 32 bits.
- The ID widths for the read and write channel of the same AXI4 Master interface must match. Different AXI Master interfaces can have different ID widths.

When there are multiple kernels in the DUT that want to access external memory, during IP core generation, you can generate the read and write AXI masters for each kernel interface inside the DUT. These AXI masters consume hardware resources and add critical delays to the entire IP core when the number of AXI masters increase.



To allow multiple modules to share the same AXI Master Interface to reduce hardware consumption and critical delays, you can use ID signals. Besides modeling the physical ID signals, model an arbitrator inside the DUT. Based on your design needs, the arbitrator can be extendable as the number of processing kernels increases.



To see the simple read arbitrator modeled in this design, open the `multi_masters_with_ID.slx` model and double-click the `DUT` subsystem.

Model Designs with Multiple Sample Rates

The HDL Coder software supports designs with multiple sample rates when you run the IP Core Generation workflow. When you map the interface ports to AXI4 Master interfaces, to use multiple sample rates, ensure that the DUT ports that map to these AXI4 interfaces run at the fastest rate of the design after HDL code generation.

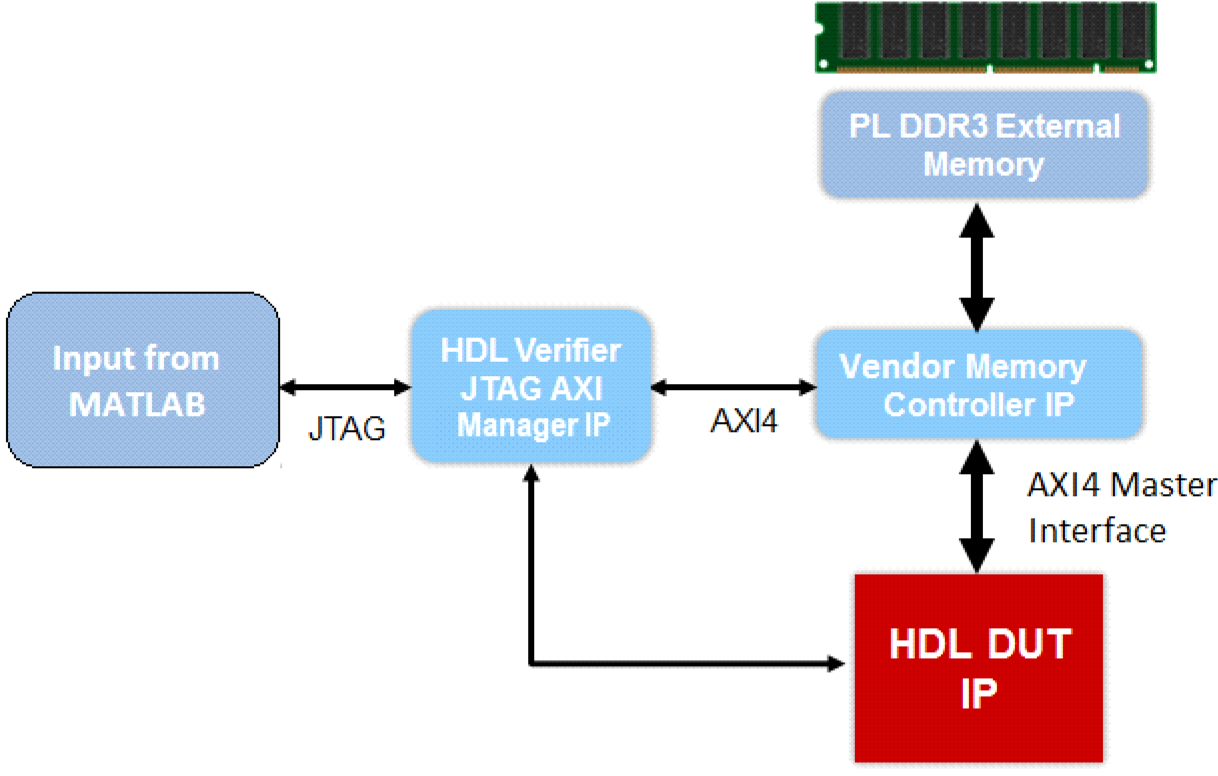
To learn more, see “Multirate IP Core Generation” on page 40-85.

Reference Designs for IP Core Integration

You can integrate the generated HDL IP core with AXI4 Master interfaces into these HDL Coder reference designs:

- `Default System with External DDR3 Memory Access`: When your target platform is `Xilinx Zynq ZC706 evaluation kit`.
- `Default System with External DDR4 Memory Access`: When your target platform is `Intel Arria10 SoC development kit`.

To use these reference designs, you must have HDL Verifier installed. This figure shows a high level block diagram of the reference design architecture.



In this architecture, the `HDL DUT IP` block corresponds to the IP core that is generated from the `IP Core Generation` workflow. Other blocks in the architecture represent the predefined reference design, that consists of a MATLAB based JTAG AXI Manager IP that is provided by HDL Verifier. After you run the FPGA design on the board, using the JTAG AXI Manager IP, you can use the input data in MATLAB to initialize the onboard DDR3 external memory. The `HDL DUT IP` core reads the input data from the external memory via the AXI4 Master interface. The IP core then performs the algorithm computation and writes the result to DDR3 memory via the AXI4 Master interface. The JTAG AXI Manager IP can read the result from DDR3 memory and then verify the result in MATLAB.

Using the `addAXI4MasterInterface` method of the `hdlcoder.ReferenceDesign` class, you can integrate the IP core with AXI4 Master Interface into your own custom reference design.

Restrictions

- **Synthesis tool**: Must be `Xilinx Vivado` or `Altera QUARTUS II`. `Xilinx ISE` is not supported.
- **Target workflow**: Use the `IP Core Generation` workflow. To run the workflow, open the HDL Workflow Advisor from your DUT algorithm in Simulink. MATLAB to HDL workflow is not supported.
- **Processor/FPGA synchronization**: Must be `Free running` mode.

See Also

Related Examples

- “Perform Matrix Operation Using External Memory” on page 40-218

More About

- “Model Design for AXI4-Stream Interface Generation” on page 40-14
- “Streaming Pixel Interface” (Vision HDL Toolbox)

IP Core Generation Workflow for Standalone FPGA Devices

In this section...

"Targeting FPGA Reference Designs with AXI4 Interface" on page 40-142
"Targeting FPGA Reference Designs Without AXI4 Interface" on page 40-144
"Board Support" on page 40-144
"Restrictions" on page 40-144

You can generate a reusable HDL IP core for any supported Xilinx or Altera FPGA device. The workflow produces an IP core report that displays the target interface configuration and the coder settings that you specify. See "Custom IP Core Generation" on page 39-17.

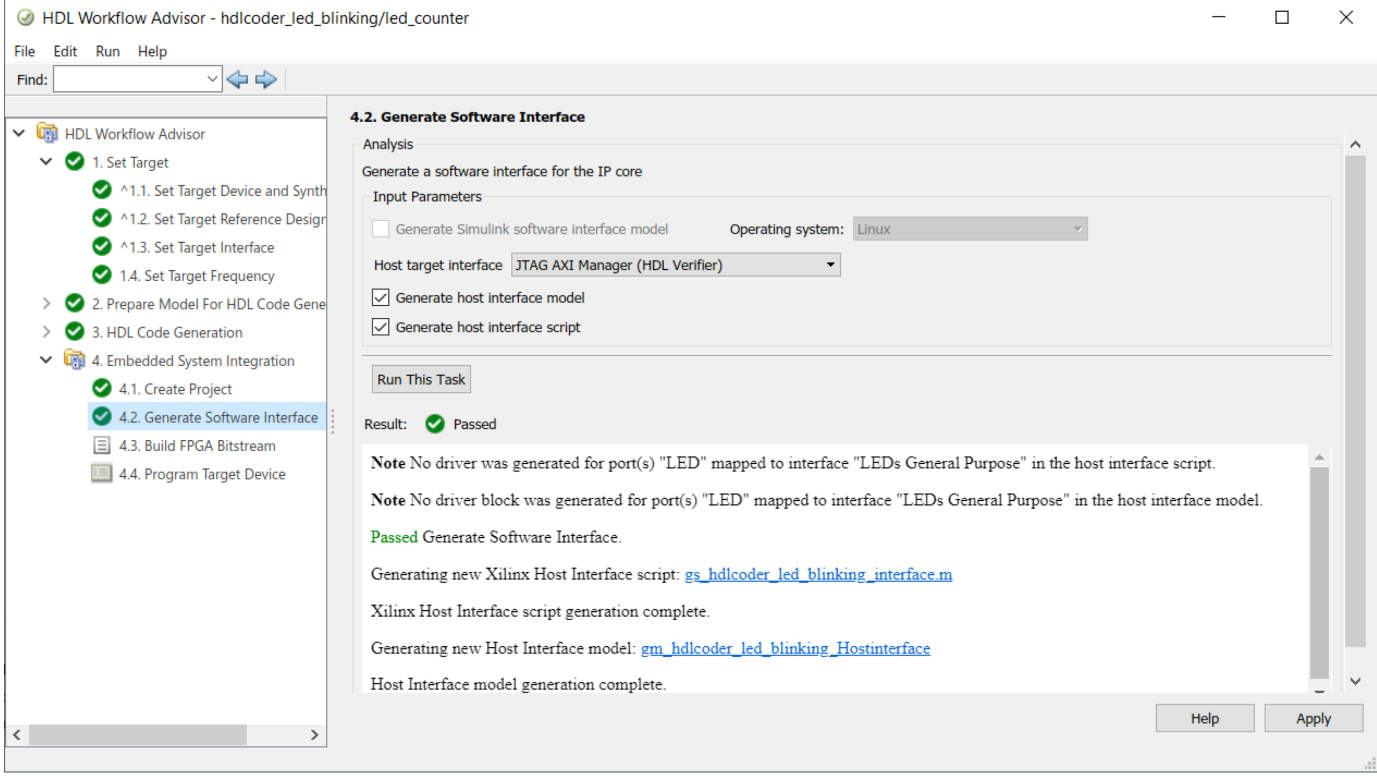
You can optionally build your own custom reference designs and integrate the generated IP core into the reference design. The workflow does not require the Embedded Coder software, because you need not generate the embedded code that is run on the processor. This means that the workflow has a **Generate Software Interface** task, but you cannot generate a software interface model.

If you have HDL Verifier installed, on the **Set Target Reference Design** task, set **Insert AXI Manager (HDL Verifier required)** to `JTAG` or `Ethernet` based on the interface that communicates between your host machine and the target board. For Ethernet, specify the IP address of the target board using the **Board IP Address** parameter.

Note By default, the `Ethernet` option is available for only the Artix-7 35T Arty, Kintex-7 KC705, and Virtex-7 VC707 boards. To enable this option for other Xilinx boards that have the Ethernet physical layer (PHY), manually add the Ethernet media access controller (MAC) Hub IP in the `plugin_board` file using the `addEthernetMACInterface` method before you launch the HDL Workflow Advisor tool.

You can then generate a host interface model, host interface script, or both in the **Generate Software Interface** task to rapidly prototype and test the HDL IP core functionality by using the AXI Manager. See "Generate and Manage FPGA I/O Host Interface Scripts" on page 39-68.

Note To generate a host interface model, you must map each DUT signal that you want to capture to the `AXI4` or `AXI4-Lite` interfaces in the **Set Target Interface** task.



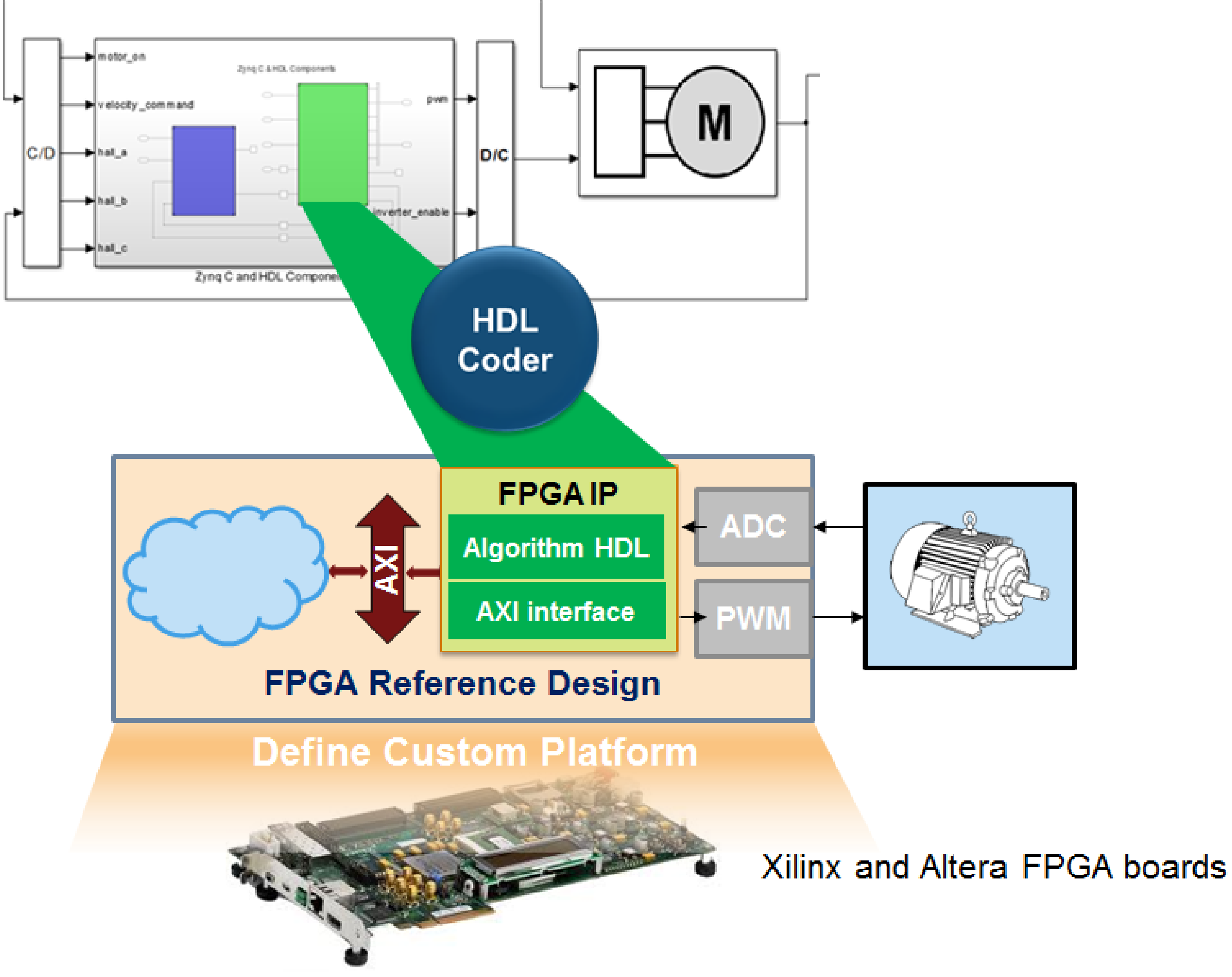
The workflow for the FPGA boards has these features:

- **Set Target Reference Design** task. Populates the reference design, its tool version, and the parameters that you specify.
- **Set Target Interface** task. Map your DUT ports to the interfaces on the target platform.
- **Set Target Frequency** task. Specifies the **Target Frequency (MHz)** to modify the clock module in the reference design to produce a clock signal with that frequency.
- **Generate RTL Code and IP Core** task. Generates a reusable and sharable IP core. The IP core packages the RTL code, a C header file, and the IP core definition files.
- **Create Project** task. Creates a project for integrating the IP core into the predefined reference designs.

You can generate an IP core with an optional AXI4 or an AXI4-Lite interface.

Targeting FPGA Reference Designs with AXI4 Interface

This figure shows how HDL Coder generates an IP core with an AXI4 interface and integrates the IP core into the FPGA reference design. See “Board and Reference Design Registration System” on page 40-89.



Use the HDL Coder generated AXI4-Lite interface to connect the IP core with an AXI4 or AXI4-Lite Master device such as:

- MicroBlaze processor.
- Nios II processor.
- PCIe Endpoint that connects to an external processor.
- JTAG Master.

When you connect the HDL IP core to a processor such as the MicroBlaze, you must integrate the handwritten C code to run on the processor. The generated IP core report displays the register address mapping information. To find the register offsets in the IP core register space, use this mapping information. To get the memory address of each register, add the register offset to the base address that you specify in your reference design. You can also find the register offsets in the C header file in the generated IP core folder.

Targeting FPGA Reference Designs Without AXI4 Interface

In the reference design definition function, you can create your own custom reference designs without the AXI4 slave interface. See also `addAXI4SlaveInterface`.

When creating a custom reference design, to target a standalone FPGA board, use the `EmbeddedCoderSupportPackage` method of the `hdlcoder.ReferenceDesign` class:

```
hRD.EmbeddedCoderSupportPackage = ...
                hdlcoder.EmbeddedCoderSupportPackage.None;
```

See `EmbeddedCoderSupportPackage`.

Board Support

HDL Coder supports these FPGA boards with the `IP Core Generation` workflow:

- `Xilinx Kintex-7 KC705 development board`
- `Arrow DECA MAX 10 FPGA evaluation kit`

Using these boards, you can integrate the generated IP core into the `default system` reference design. By default, this reference design does not have an AXI4 slave interface. Optionally, you can add the interface in the reference design definition function.

Restrictions

`IP Core Generation` workflow does not support :

- **RAM Architecture** set to `Generic RAM without clock enable`.
- Using different clocks for the IP core and the AXI interface. The `IPCore_Clk` and `AXILite_ACLK` must be synchronous and connected to the same clock source. The `IPCore_RESETN` and `AXILite_ARESETN` must be connected to the same reset source. See “Synchronization of Global Reset Signal to IP Core Clock Domain” on page 39-40.

See Also

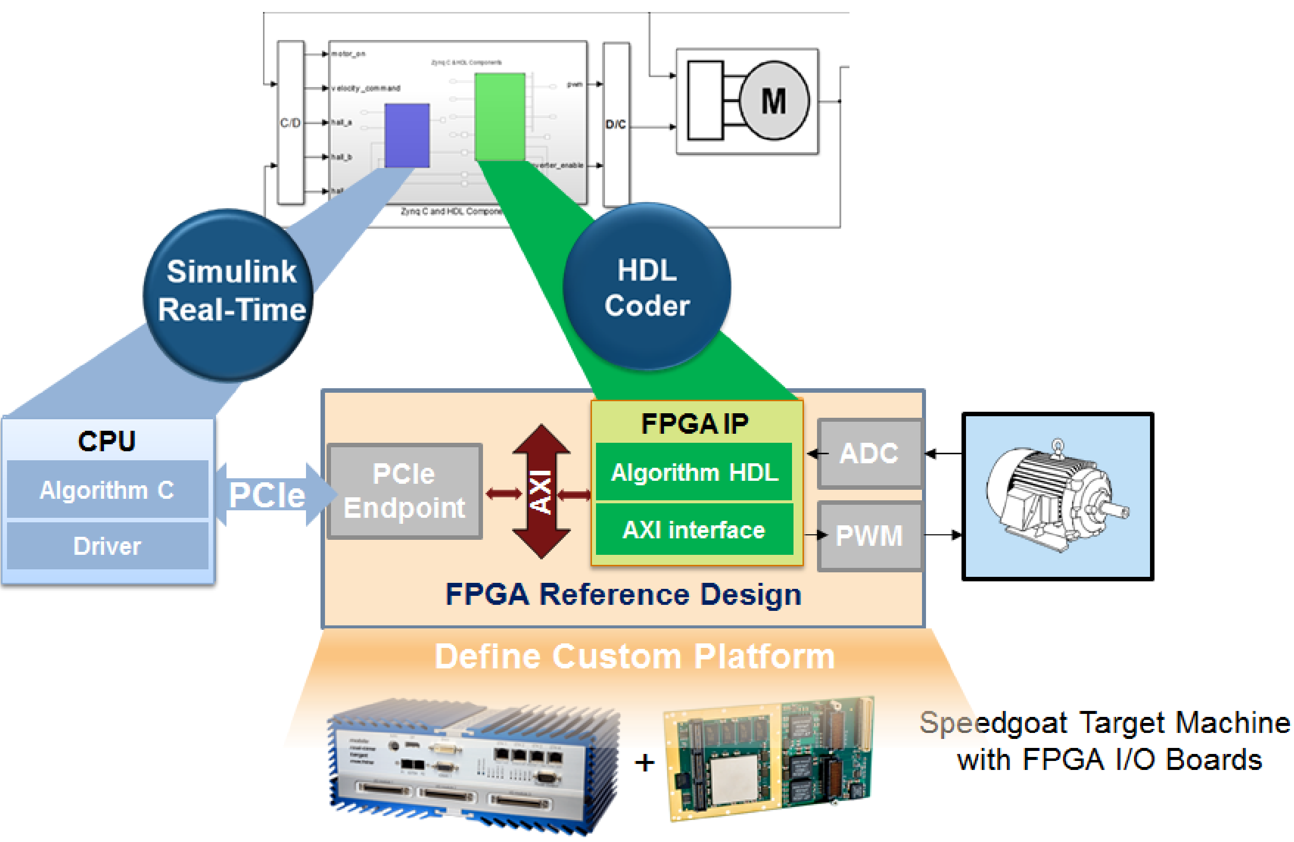
Related Examples

- “Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow” on page 39-190

IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules

HDL Coder uses the `IP Core Generation` workflow infrastructure to generate a reusable HDL IP core for the Speedgoat Simulink-Programmable I/O modules that support Xilinx Vivado. The workflow produces an IP core report that displays the target interface configuration and the code generator settings that you specify. You can integrate the IP core into a larger design by adding it in an embedded system integration environment. See “Custom IP Core Generation” on page 39-17.

This figure shows how the software generates an IP core with an AXI interface and integrates the IP core into the FPGA reference design.



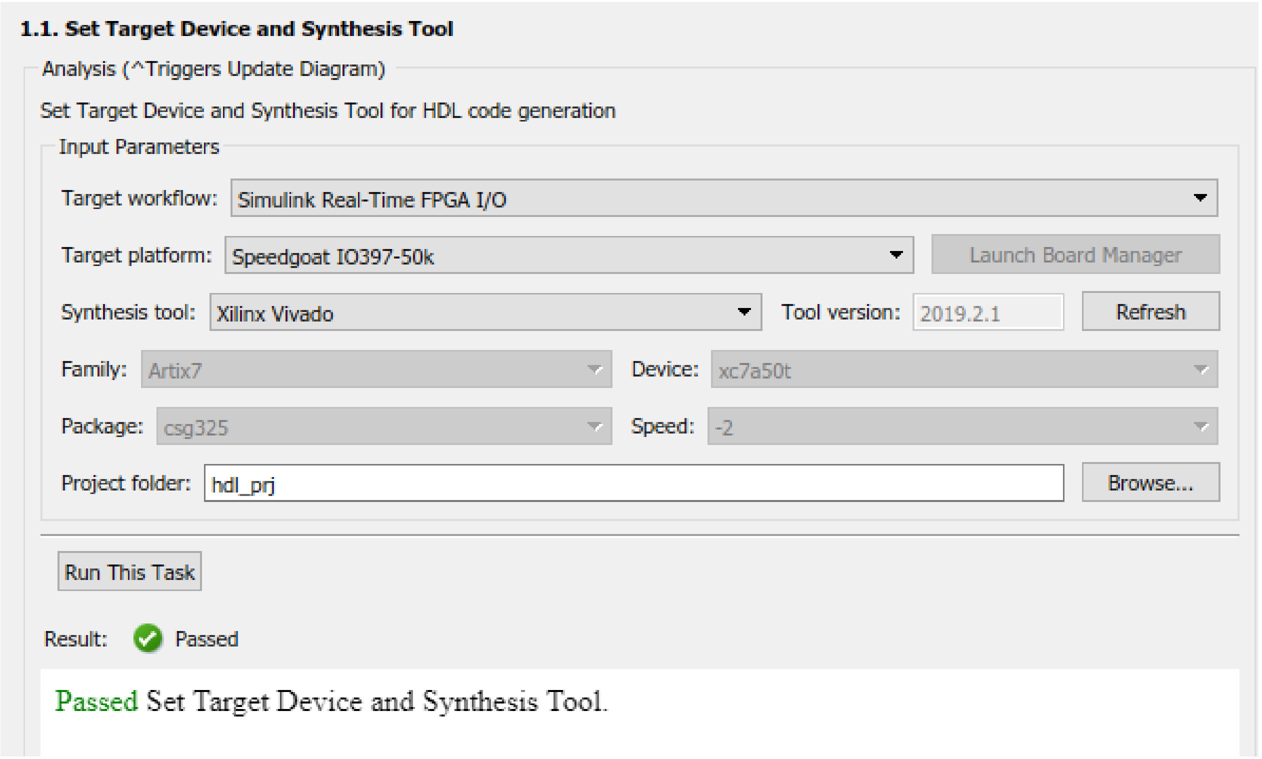
Supported I/O Modules

To learn about I/O modules that HDL Coder supports with the `Simulink Real-Time FPGA I/O` workflow, see “Speedgoat FPGA Support with HDL Workflow Advisor” on page 39-15.

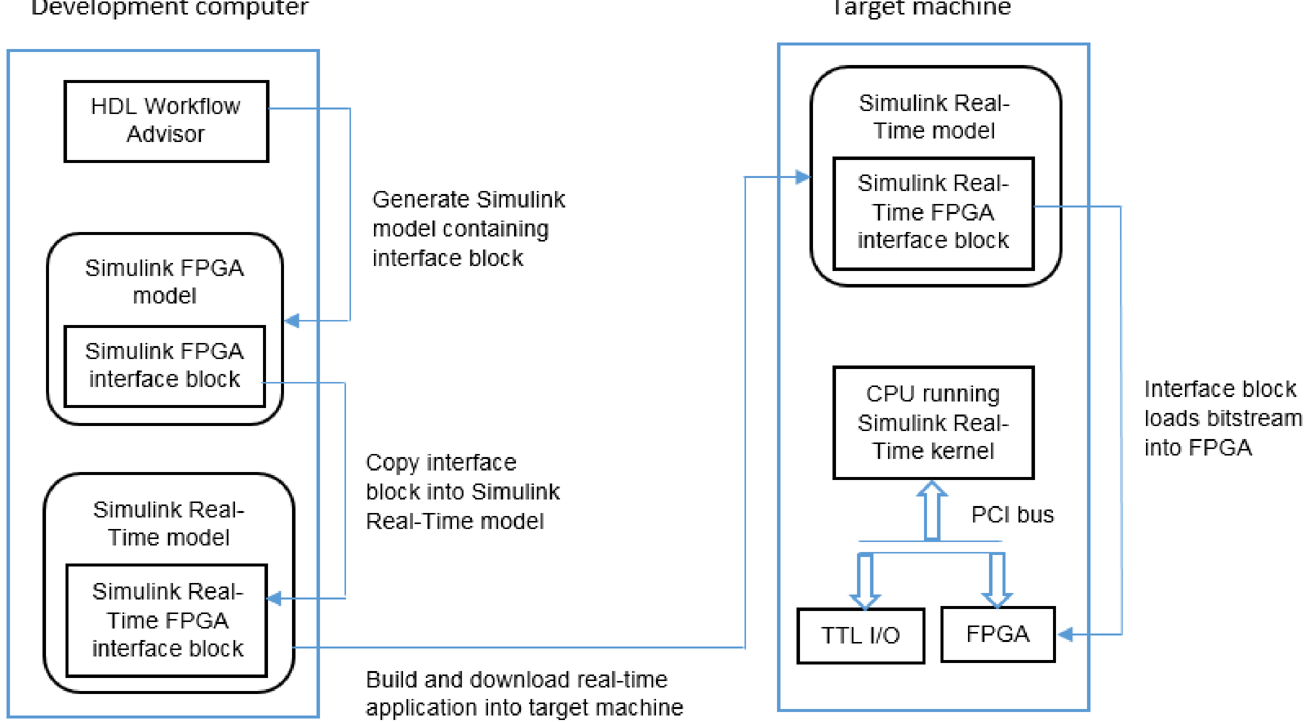
IP Core Generation Workflow

This workflow has these key features:

- Uses Xilinx Vivado as the synthesis tool.
- Generates a reusable and sharable IP core. The IP core packages the RTL code, a C header file, and the IP core definition files.
- Creates a project for integrating the IP core into the Speedgoat reference design.
- Generates an FPGA bitstream and downloads the bitstream to the target hardware.



After building the FPGA bitstream, the workflow generates a Simulink Real-Time model. The model is an interface subsystem model that contains the blocks to program the FPGA and communicate with the I/O module through the PCIe bus during real-time execution.



Restrictions

`IP Core Generation` workflow does not support :

- **RAM Architecture** set to `Generic RAM without clock enable`.
- Using different clocks for the IP core and the AXI interface. The `IPCore_Clk` and `AXILite_ACLK` must be synchronous and connected to the same clock source. The `IPCore_RESETN` and `AXILite_ARESETN` must be connected to the same reset source. See “Synchronization of Global Reset Signal to IP Core Clock Domain” on page 39-40.

See Also

More About

- “Speedgoat FPGA Support with HDL Workflow Advisor” on page 39-15
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 40-113
- “Custom IP Core Generation” on page 39-17

External Websites

- https://www.speedgoat.com/knowledge-center

Map Bus Data Types to PCIe Interface

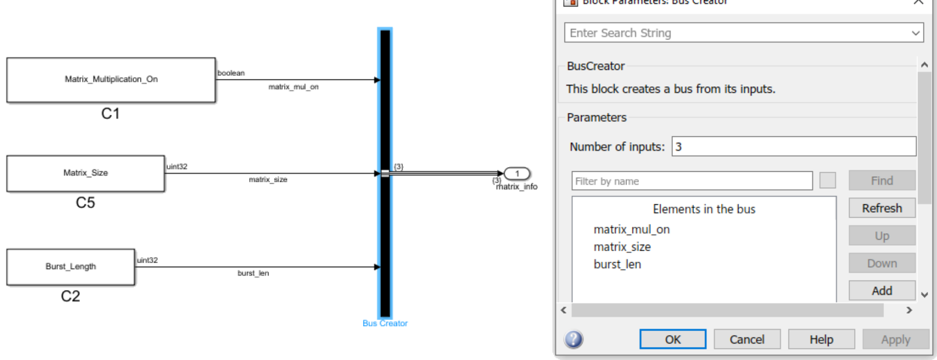
When you use bus data types at the DUT interface ports, you can directly map the interface ports to PCIe interfaces.

When you map bus data types, HDL Coder assigns a unique address for each data port that you want to map to the PCIe interface. The top-level and sub-level buses do not have a register offset address. The address mapping for separate scalar or vector bus elements is not contiguous.

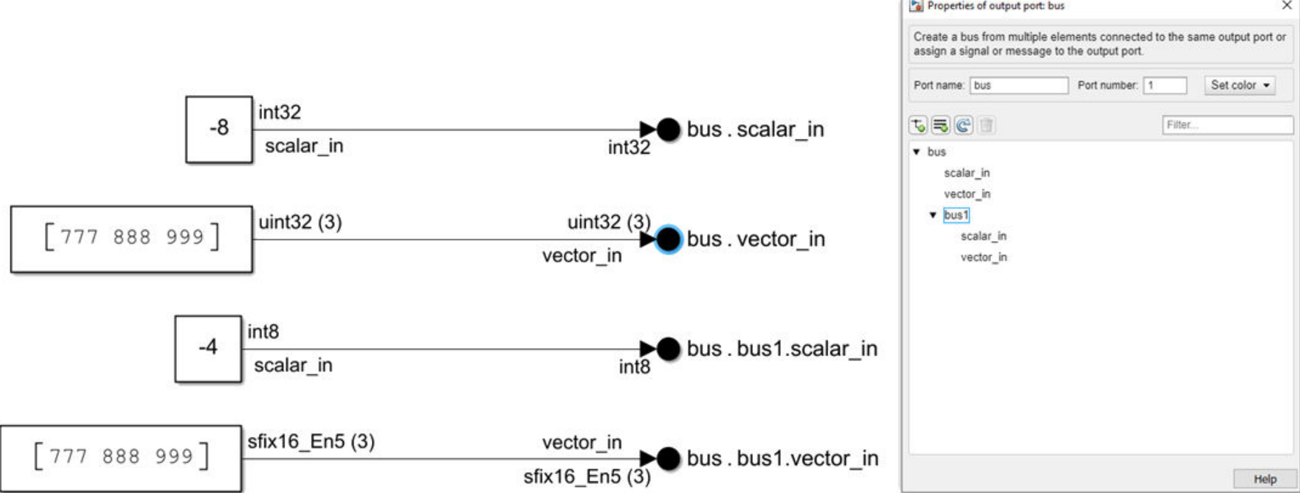
Model Bus Element

Model a bus element by using a bus creator block or bus element block to create a bus port.

Model a bus element by using a bus creator block.



Model a bus element by using bus element blocks:



For an example that maps bus data type to PCIe Interface, see “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 40-113.

IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip

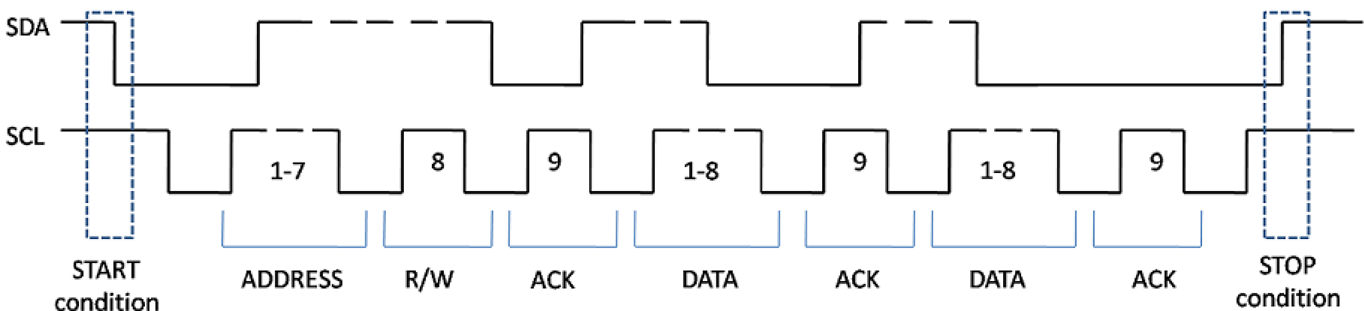
This example illustrates how to model an I2C controller using an I2C Master controller modeled using Stateflow® blocks for configuring the audio codec chip.

In this example, you:

1. Model the I2C Master Controller using Stateflow® blocks in Simulink®
2. Model the I2C Controller using the I2C Master Controller block for configuring the Audio Codec Chip
3. Use the blackbox subsystem and bidirectional port features to handle tri-state logic in I2C IP core
4. Use the IP Core Generation workflow to generate an IP core for the I2C Controller

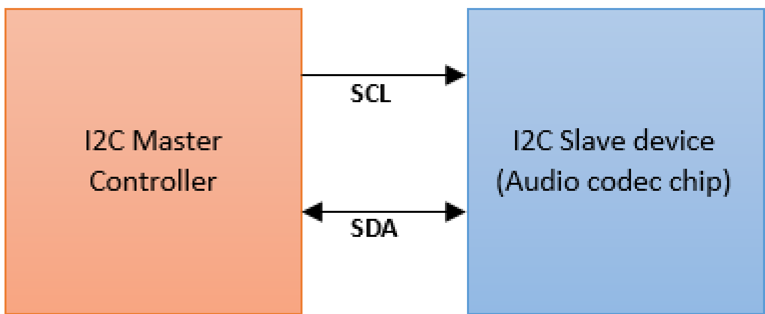
1. Overview of I2C protocol

I2C bus, also called Inter-IC bus, is a simple, multi-master, multi-slave, bidirectional two-wire bus, that consists of serial data (SDA) and serial clock (SCL) lines. Each device connected to the bus is software addressable by a unique 7-bit or 10-bit address, and maintains a simple master-slave relationship. Serial, 8-bit oriented, bi-directional data transfers can be made at up to 100 kbit/s in the Standard mode, up to 400 kbit/s in the Fast-mode,or up to 3.4 Mbit/s in the High-speed mode. I2C bus has two nodes: master node and slave node. The master node generates clock and initiates communication with the slave. The slave node which is addressed by the master receives clock and responds to the master during acknowledgment. There are four modes of operation which are master transmit, master receive, slave transmit and slave receive. The master starts the communication by sending start bit followed by 7 or 10 bit address of the slave followed by read(1) or write(0) bit. If the slave corresponding to that address is present, then it responds with ACK bit. The master continues communication in transmit or receive mode based on the read or write bit. Similarly, the slave continues its operation based on the read or write instruction from the master. The figure below shows the timing diagram of I2C protocol.

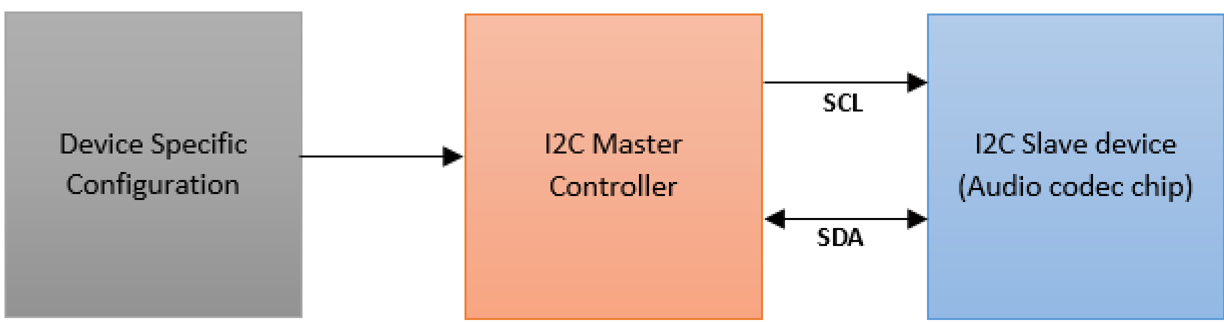


2. Modeling Generalized I2C Master Controller in Simulink Using Stateflow Blocks

Configuring multiple peripherals in the design can be a cumbersome and tedious process. Instead, create a generic I2C Master Controller that you can directly use to configure the audio codec chips. The figure below shows the architecture of the Generalized I2C Master Controller which is implemented using Stateflow blocks in Simulink.

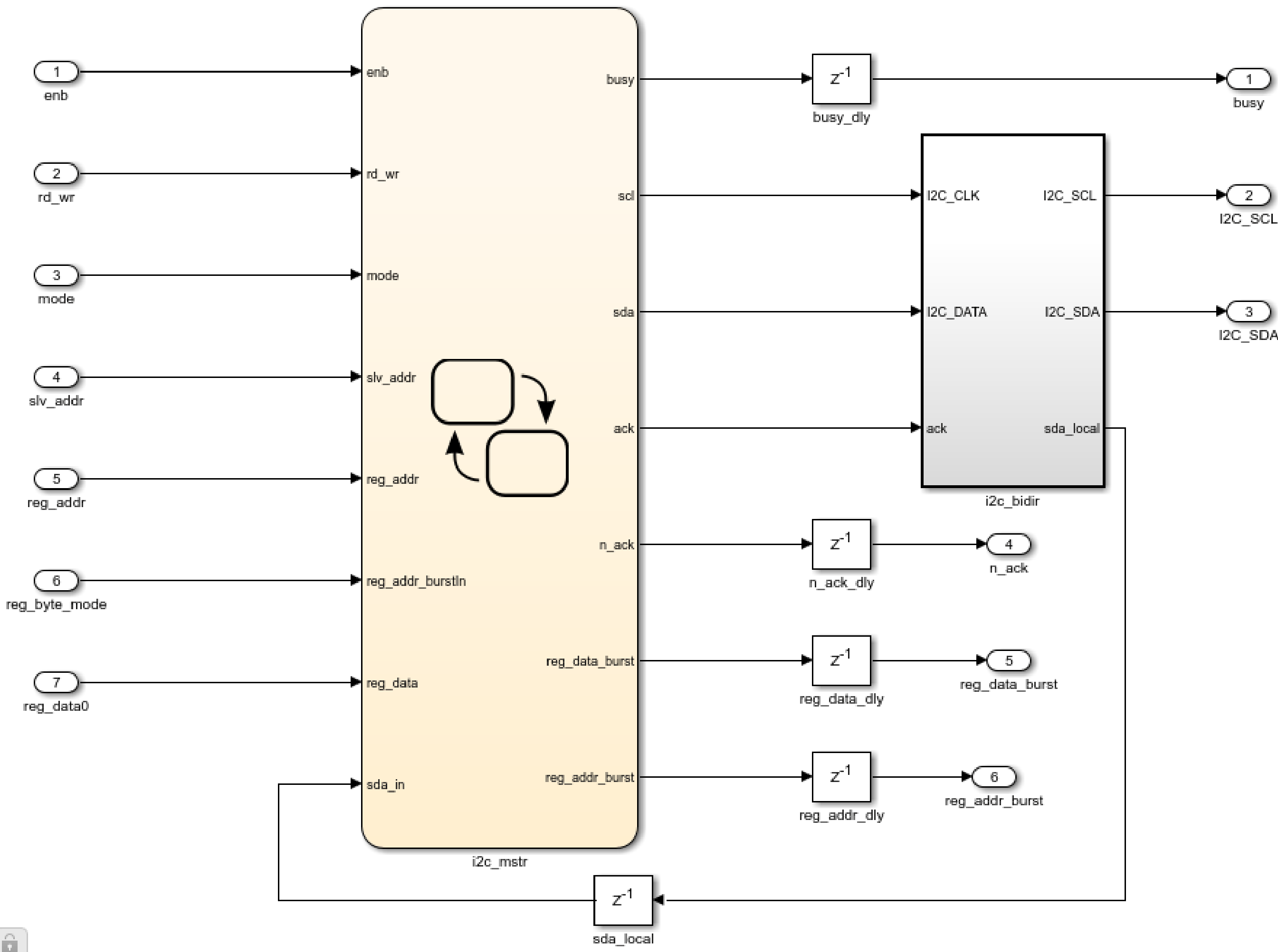


The above part shows the I2C Master Controller block. To configure the peripheral, you must provide device-specific configuration to the I2C Master Controller block. The block diagram to configure the audio codec chip using I2C Master Controller is as shown below.



The below model shows the I2C Master Controller which is modeled in Simulink using Stateflow blocks.

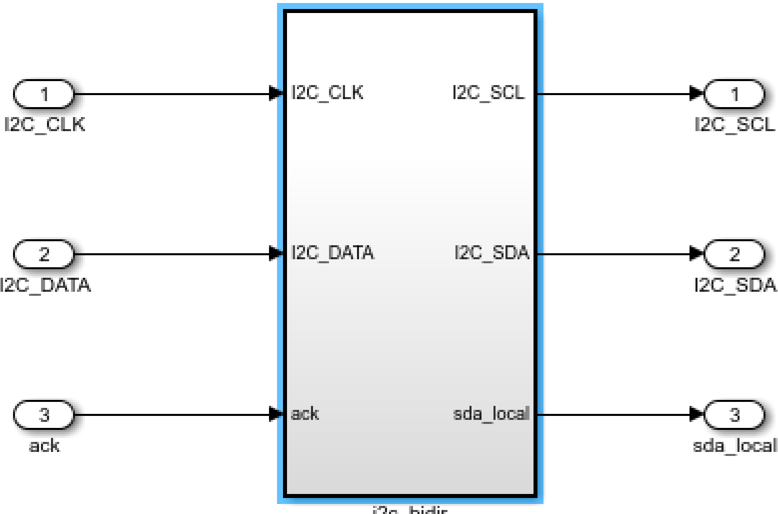
```
modelname = 'hdlcoder_I2C_master_controller';
open_system(modelname);
open_system('hdlcoder_I2C_master_controller/I2C_MasterController');
```



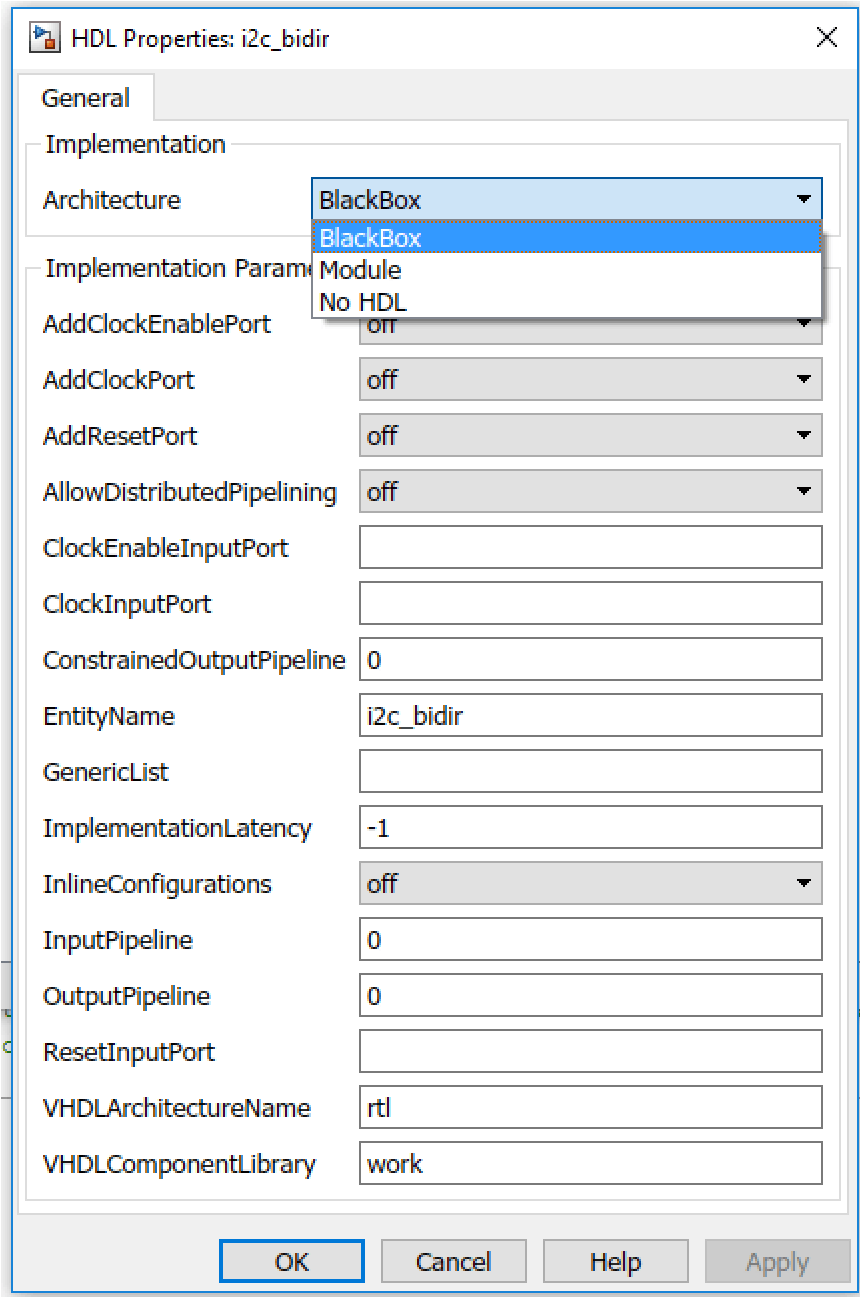
The I2C Master Controller only supports I2C write and doesn't support I2C readback currently. I2C Master Controller consists of two parts, I2C Master Controller chart and tristate buffer blackbox. I2C Master Controller chart provides serial data, SDA and serial clock, SCL to slave device through the tristate buffer blackbox. Tristate buffer blackbox uses the handwritten VHDL code and used for the bidirectional functionality of the SDA port. Tristate buffer blackbox is added in the model as Simulink doesn't support bidirectional port modeling.

To create a blackbox use the following steps.

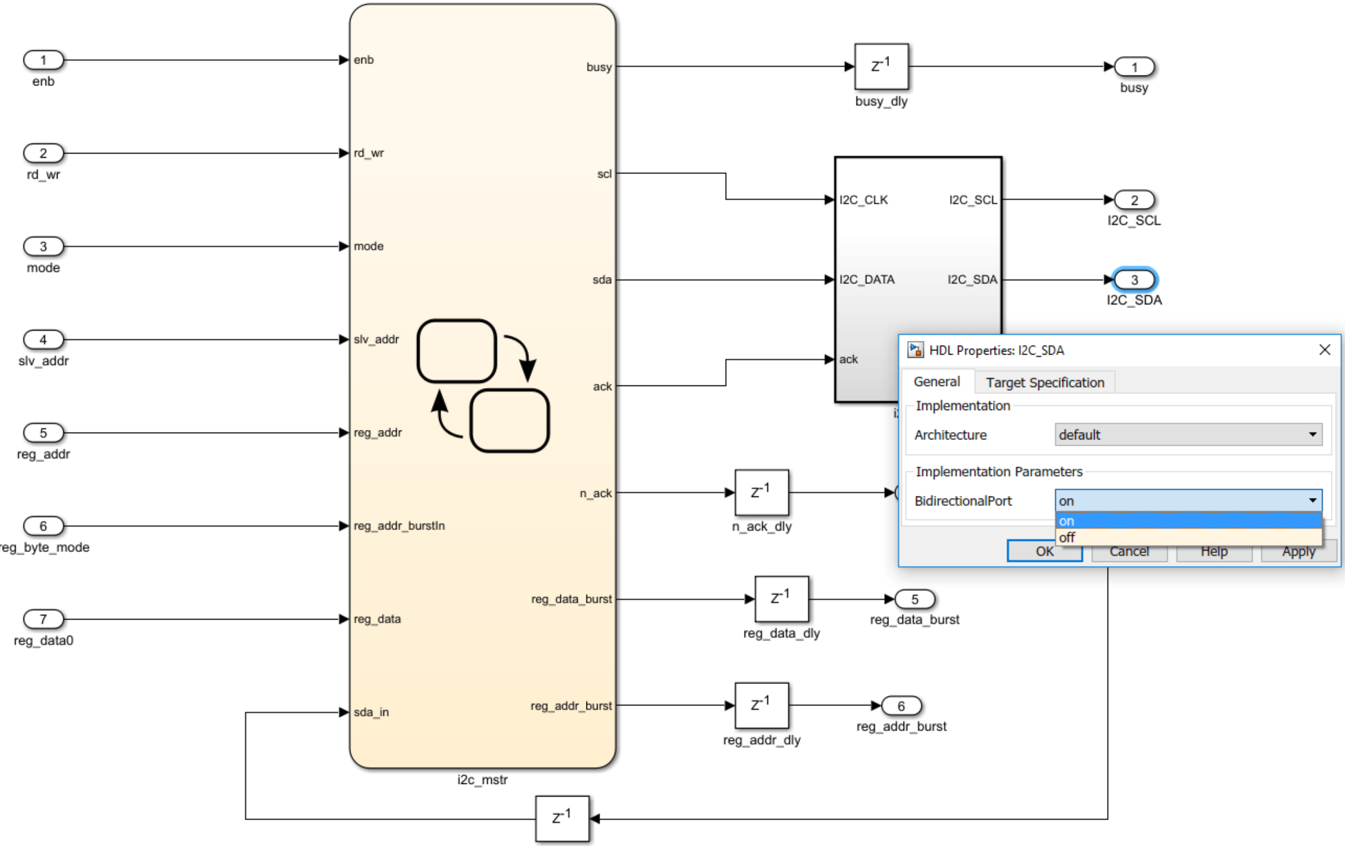
1. Make a subsystem which contains input and output ports of the HDL source code which you want to import for the blackbox creation. The I2C tristate buffer blackbox is as shown below.



2. To specify your subsystem as a black box interface, right click on the subsystem and select **HDL Code > HDL Block Properties** and set the Architecture to Blackbox as shown in the following figure.



3. The data port **I2C_SDA** of I2C Master Controller is bidirectional. To set the port as bidirectional, right click on the **I2C_SDA** port and click on HDL block properties and set the BidirectionalPort on as shown below.



4. During simulation, the actual content inside of the blackbox subsystem will be used for simulation.

During code generation, HDL Coder will not generate the code under the blackbox subsystem. Instead, the code generator integrates your hand-written HDL code into the IP core. Inputs to the I2C Master Controller block can be provided by adding a device specific configuration chart at the input. This chart contains details about the registers that need to be configured for your slave device. More about device configuration is covered in the section of Zedboard, Zybo board and Arrow SoC Development Kit Audio Codec configuration using I2C Master Controller.

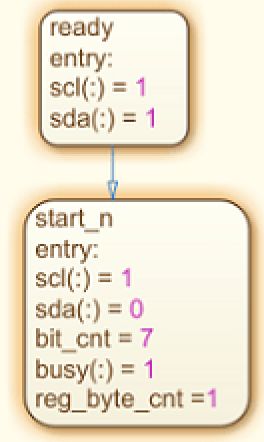
2.1 I/O Description of I2C Master Controller block

The following figure gives the details about input and output ports of the I2C Master Controller block.

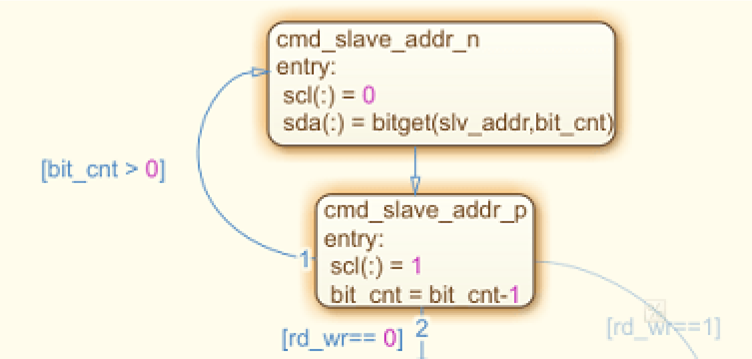
Name	Type	Width (in bits)	Description
enb	Input	1	I2C master controller enable
rd_wr	Input	1	Read/Write cycle
mode	Input	1	Mode of operation Normal = 0 Burst = 1
slv_addr	Input	8	Device address
reg_addr	Input	8	Device register address
reg_byte_mode	Input	1	Register address mode Byte = 0 Burst = 1
reg_data0	Input	8	Device register data
busy	Output	1	Status signal of the master controller chart
I2C_SCL	Output	1	Serial clock to slave device
I2C_SDA	Output	1	Serial data to slave device
n_ack	Output	1	No acknowledgment
reg_data_burst	Output	1	Burst data enable signal
Reg_addr_burst	Output	1	Burst address enable signal

2.2 Description of I2C Master Controller Stateflow chart

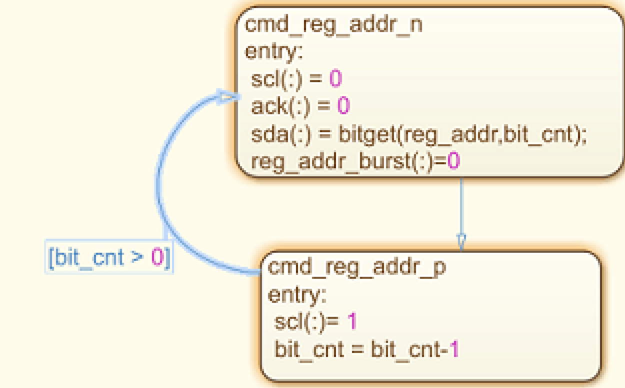
I2C Master Controller chart is made in such a way that in all the states required clock (SCL) is generated and data is provided as per the I2C protocol through serial data (SDA) port. Following states shows the generation of the clock and start bit.



The following states are used to send 7-bit address of the slave device.



The function **bitget** is used to send bits serially to SDA port. It allows user to get bit value at specified position of the integer mentioned in its argument list. The transition from one state to other state depends on execution order specified for the transition conditions. As shown in the above figure the transition from **cmd_slave_addr_p** to the **cmd_slave_addr_n** state occurs based on the transition condition(bit_cnt > 0). The value of bit_cnt keeps decrementing until the transition condition satisfies. The value of bit_cnt is initialized to '7' and its value decrements till it becomes '1' which is used to send 7-bit address of slave device on SDA port. For HDL code generation, supported data types must be used. Colon(:) operator as shown in the states which is a typecasting operator used in cmd_slave_addr_n state(scl(:)= 0) converts a value of type 'double' to a type 'logical' (SCL is the logical datatype in the states shown). The states shown below are used to send register address to the slave device.



The states shown below are used to send register data to the slave device.

cmd_reg_data_p
entry:
scl(:) = 1
bit_cnt = bit_cnt-1

[bit_cnt > 0]

cmd_reg_data_n
entry:
scl(:) = 0
sda(:) = bitget(reg_data,bit_cnt);
ack(:) = 0
reg_data_burst(:) = 0

Following states shows the stop bit generation.

stop_ext
entry:
scl (:) = 1
sda(:) = 1
ack(:) = 0

stop
entry:
sda(:) = 0

3. Configuring audio codec ADAU1761 on Zedboard using I2C Master Controller library block

This section shows how to:

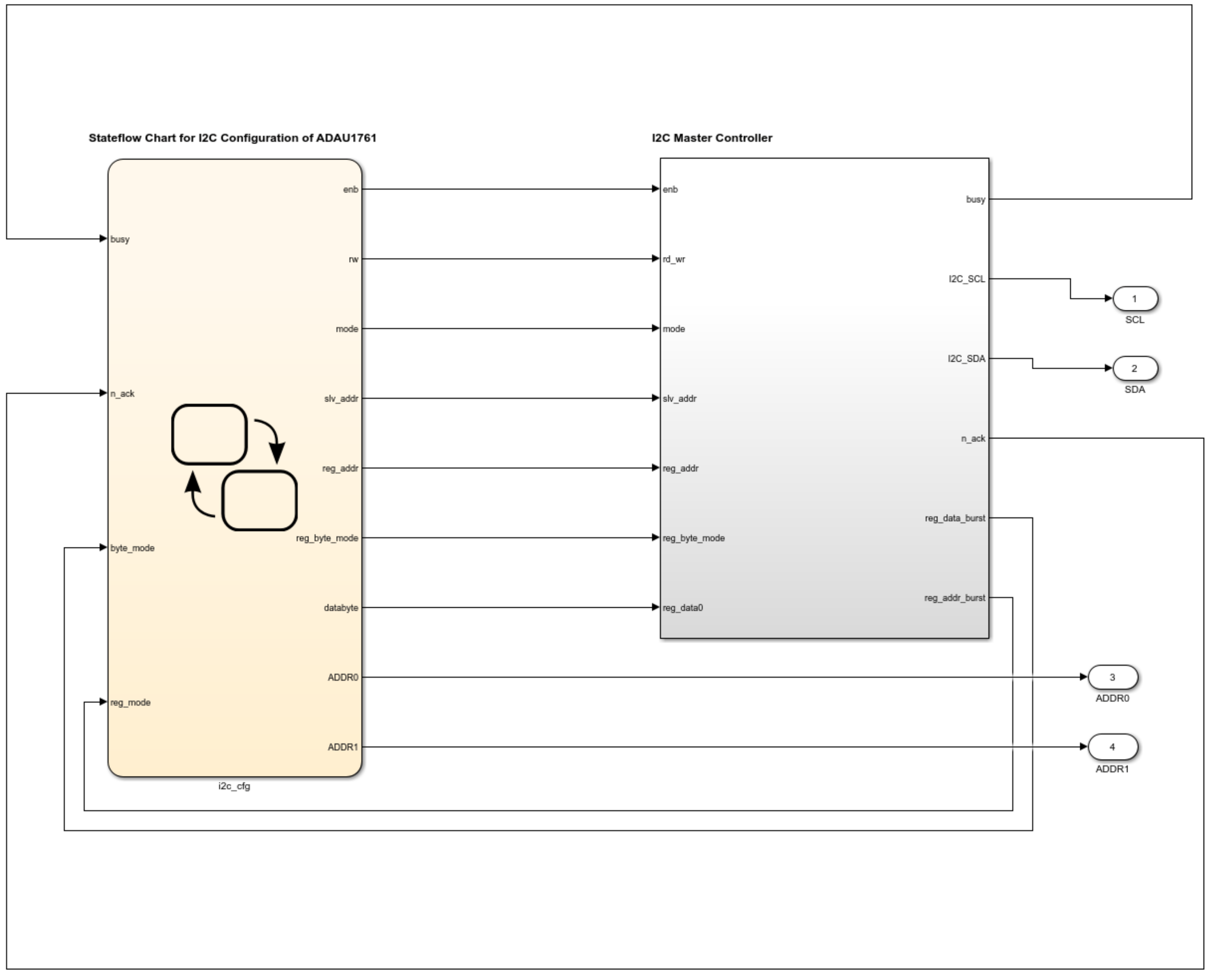
1. Model audio codec ADAU1761 device configuration chart using Stateflow blocks in Simulink.
2. Use I2C Master controller library block to configure audio codec ADAU1761.
3. Perform simulation of created model.

As mentioned earlier to configure the audio codec ADAU1761 on Zedboard, device configuration chart for ADAU1761 need to be created. This chart should be connected to the I2C Master controller library block created earlier.

Note: You have to create device configuration chart for your own device. This example is to show how the I2C Master Controller library block can be used to configure audio codec devices. The device configuration chart used for ADAU1761 is specific to this device and can't be used to configure other devices.

The configuration model created for ADAU1761 is as shown below.

```
modelname = 'hdlcoder_I2C_adau1761';
open_system(modelname);
```



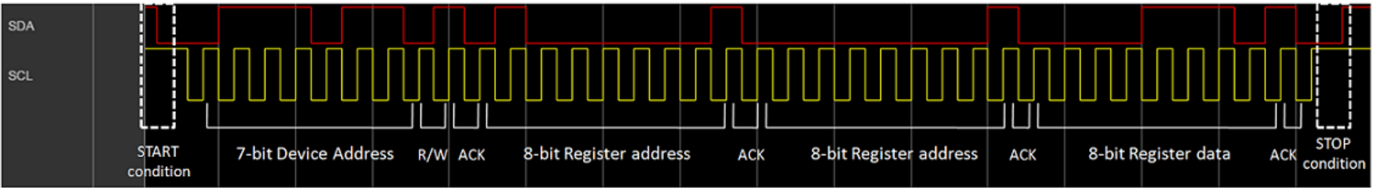
3.1 Simulating audio codec ADAU1761 configuration model

For Audio codec chip ADAU1761, 20 registers need to be configured. Few of them have to be written by I2C Master Controller in burst mode and few in byte mode. The first register is written in byte mode, second in burst mode of length 6-bytes. Remaining 18 registers are written in byte mode.

Simulation waveform for configuration of audio codec ADAU1761 is as shown below.

Byte mode transfer between I2C Master Controller and audio codec chip ADAU1761 is shown in the following figure.

Below simulation shows sending of start bit, followed by 7-bit address of the slave device(0x3B), followed by write(0) bit, followed by 16-bit register address(0x4000), followed by 8-bit register data(0x0E) and acknowledgements from the slave device.



3.2 IP Core generation workflow

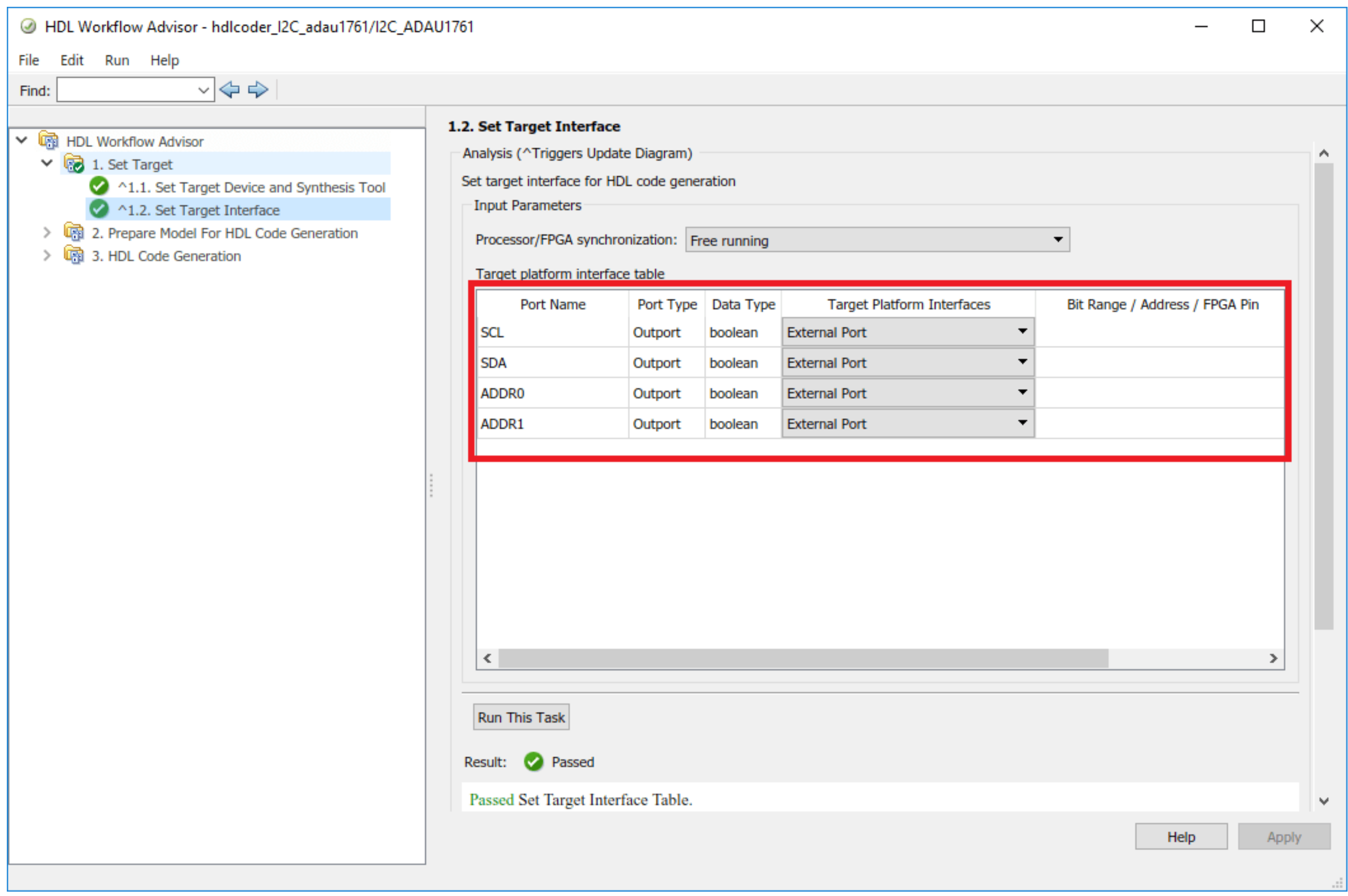
To generate the audio codec ADAU1761 configuration HDL IP core, follow the steps given below.

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command

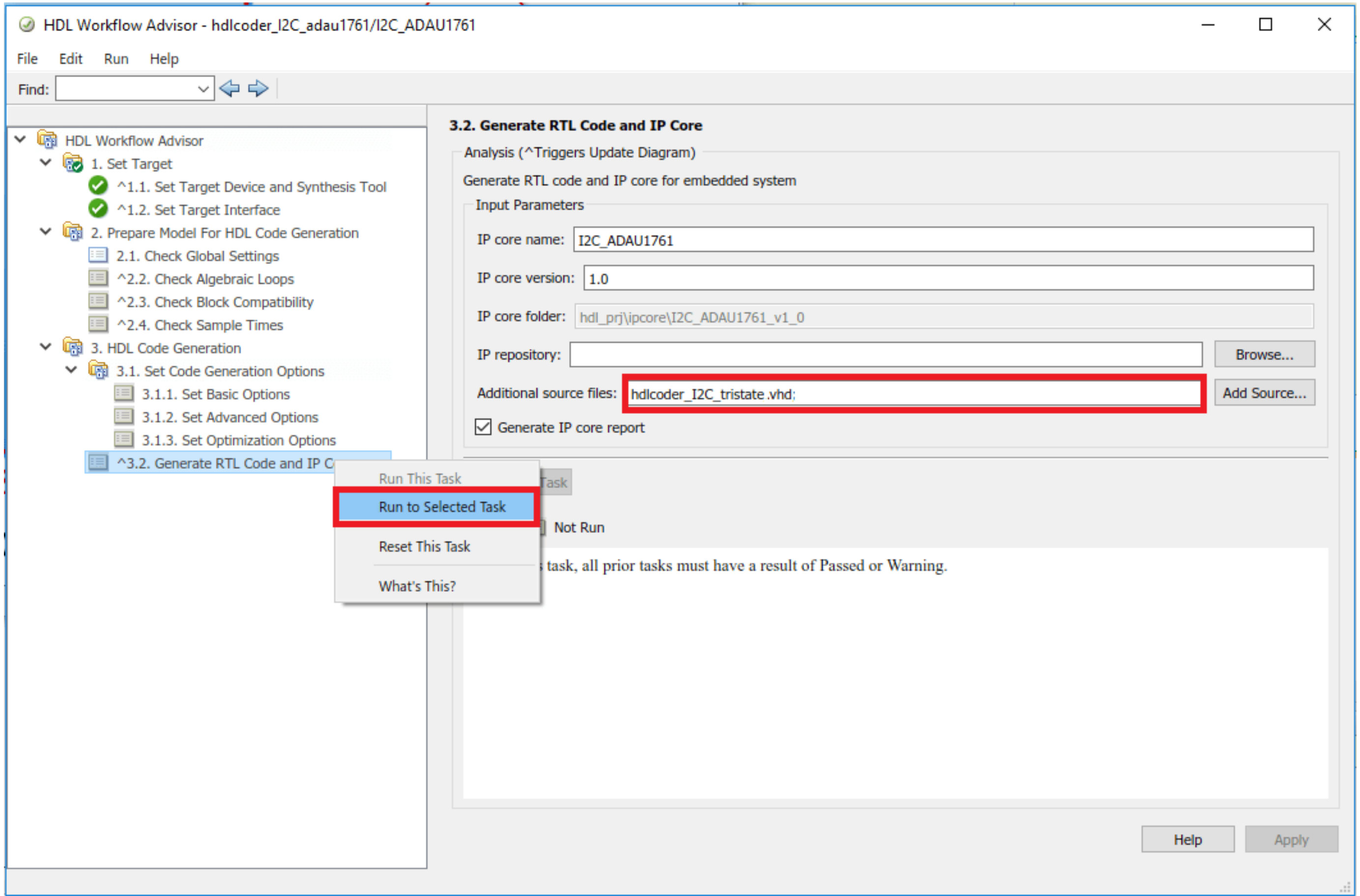
```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2017.4\bin\vivado.ba
```

2. In the ADAU1761 configuration model, select I2C_IP subsystem and by right clicking open HDL workflow advisor. In Task 1.1, Select **IP Core Generation** for Target workflow, **Generic Xilinx Platform** for Target platform and **Xilinx Vivado** for Synthesis Tool. Also select family, device, package and speed as shown in the figure below.

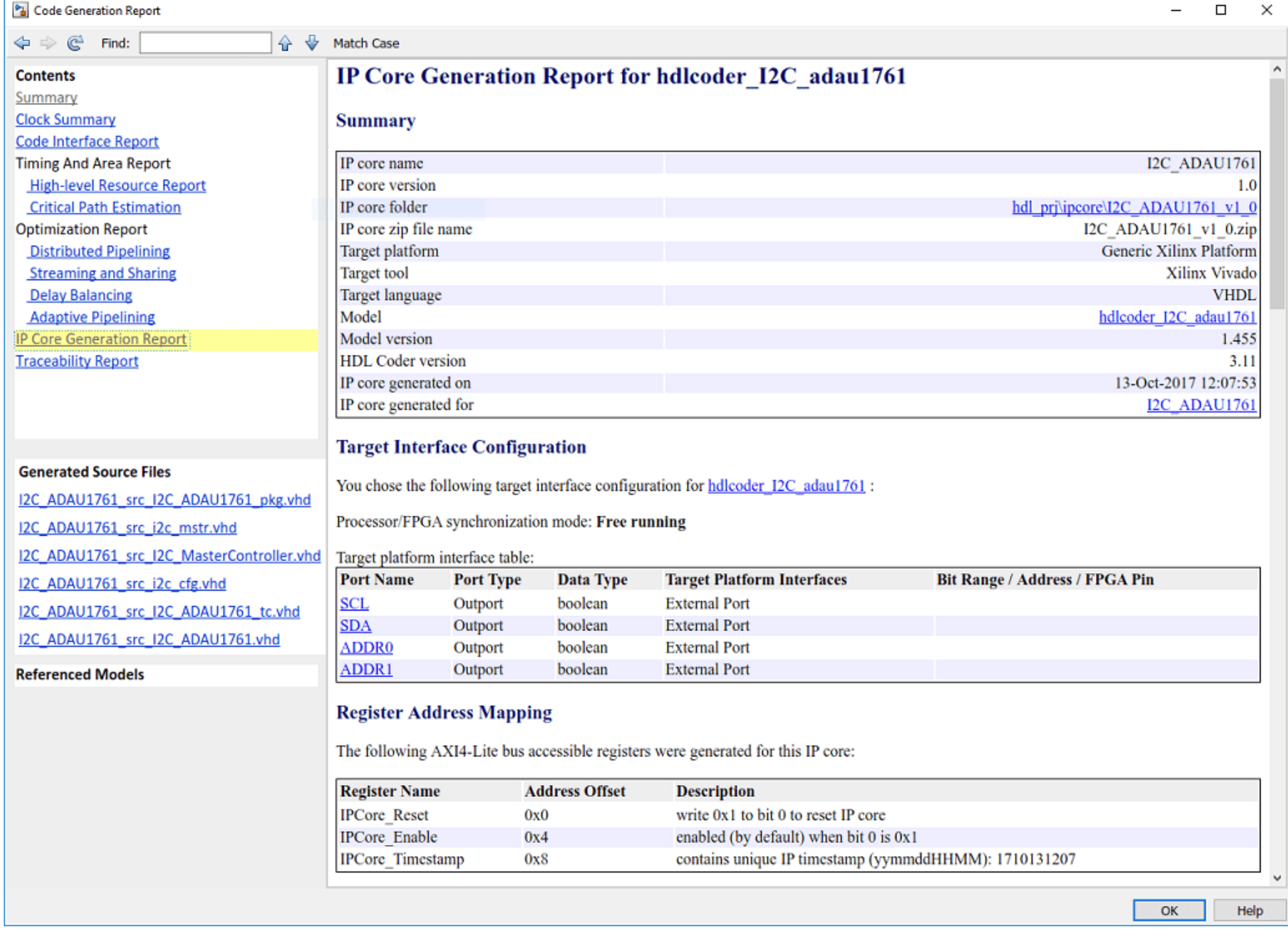
3. In Task 1.2, set the Target Platform Interfaces to "External Port" for all the ports.



4. In Task 3.2, add the tristate buffer VHDL file in the additional source files. Then right click on Generate RTL code and IP Core and click on Run to Selected Task.



I2C IP Core for configuration of ADAU1761 will be generated. Below figure shows the IP Core generation report.


Code Generation Report

Find: Match Case

Contents
Summary
Clock Summary
Code Interface Report
Timing And Area Report
High-level Resource Report
Critical Path Estimation
Optimization Report
Distributed Pipelining
Streaming and Sharing
Delay Balancing
Adaptive Pipelining
IP Core Generation Report
Traceability Report

Generated Source Files
I2C_ADAU1761_src_I2C_ADAU1761_pkg.vhd
I2C_ADAU1761_src_i2c_mstr.vhd
I2C_ADAU1761_src_I2C_MasterController.vhd
I2C_ADAU1761_src_i2c_cfg.vhd
I2C_ADAU1761_src_I2C_ADAU1761_tc.vhd
I2C_ADAU1761_src_I2C_ADAU1761.vhd

Referenced Models

IP Core Generation Report for hdlcoder_I2C_adau1761

Summary

IP core name	I2C_ADAU1761
IP core version	1.0
IP core folder	hdl_prj\ipcore\I2C_ADAU1761_v1_0
IP core zip file name	I2C_ADAU1761_v1_0.zip
Target platform	Generic Xilinx Platform
Target tool	Xilinx Vivado
Target language	VHDL
Model	hdlcoder_I2C_adau1761
Model version	1.455
HDL Coder version	3.11
IP core generated on	13-Oct-2017 12:07:53
IP core generated for	I2C_ADAU1761

Target Interface Configuration

You chose the following target interface configuration for hdlcoder_I2C_adau1761 :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
SCL	Outport	boolean	External Port	
SDA	Outport	boolean	External Port	
ADDR0	Outport	boolean	External Port	
ADDR1	Outport	boolean	External Port	

Register Address Mapping

The following AXI4-Lite bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
IPCore_Timestamp	0x8	contains unique IP timestamp (yymmddHHMM): 1710131207

OK Help

Generated IP core can be used in user reference designs. For creating the reference design, refer to “Authoring a Reference Design for Audio System on a Zynq Board” on page 40-226.

4. Configuring audio codec SSM2603 on Zybo board using I2C Master Controller library block

This section shows how to:

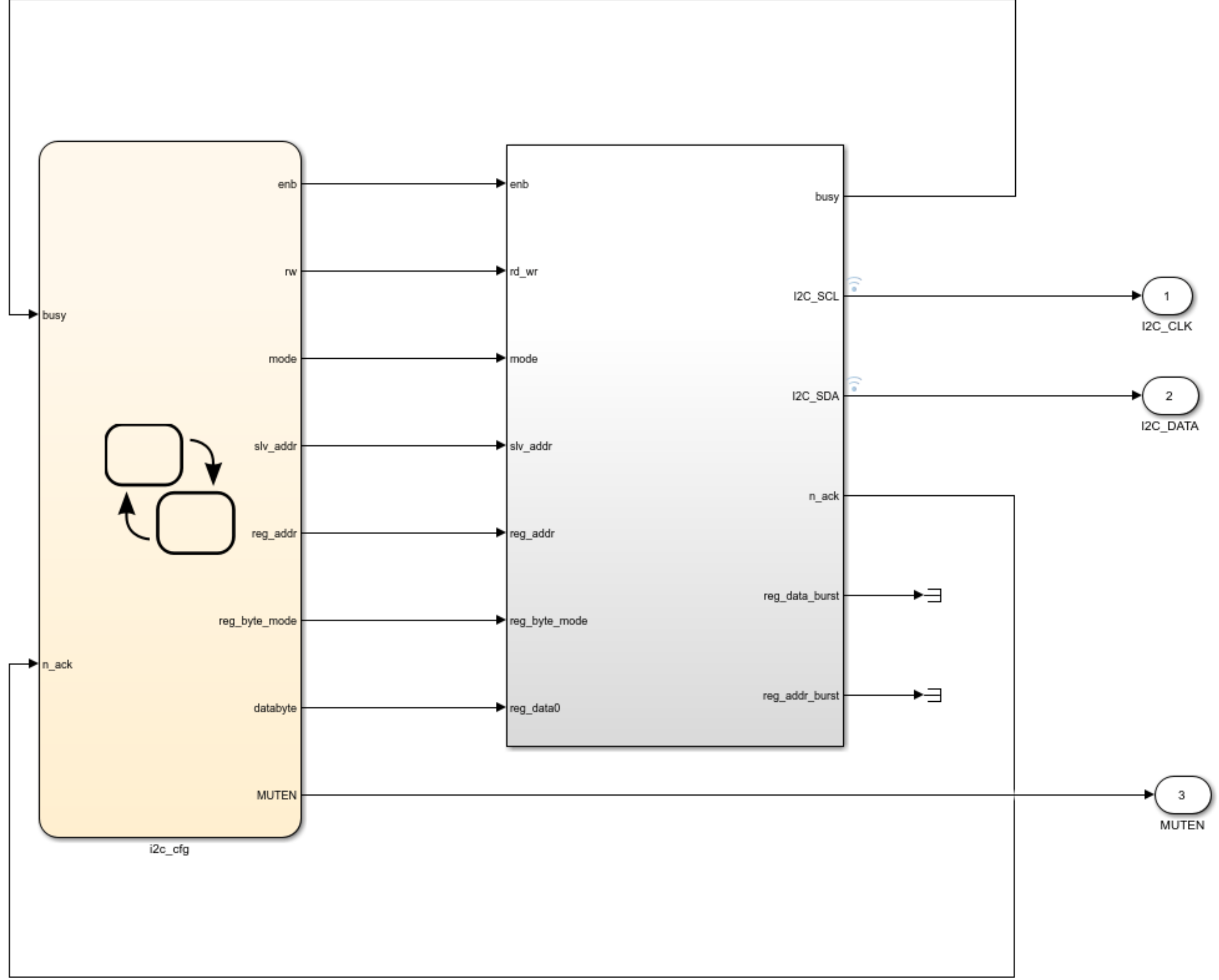
1. Model audio codec SSM2603 device configuration chart using Stateflow blocks in Simulink.
2. Use I2C Master controller library block to configure audio codec SSM2603.
3. Perform simulation of created model.

To configure the audio codec SSM2603 on Zybo board, device configuration chart for SSM2603 need to be created. This chart should be connected to the I2C Master controller library block created earlier.

Note: The device configuration chart used for SSM2603 is specific to this device and can't be used to configure other devices.

The configuration model created for SSM2603 is as shown below.

```
modelname = 'hdlcoder_I2C_ssm2603';
open_system(modelname);
```

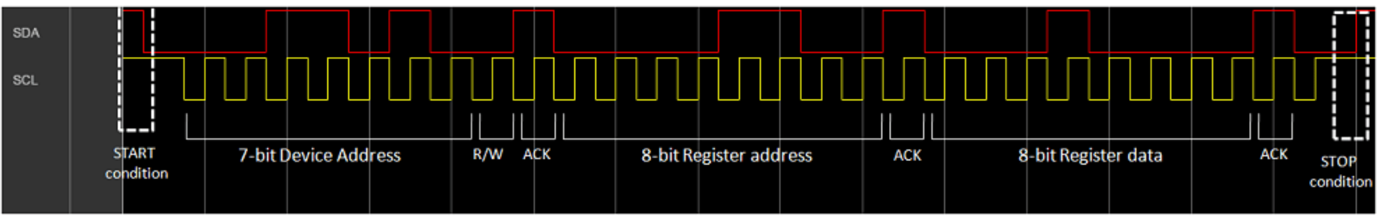


4.1 Simulating audio codec SSM2603 configuration model

For Audio codec chip SSM2603, 11 registers need to be configured. All the to be written in byte mode.

Simulation waveform for configuration of audio codec SSM2603 is as shown below.

Below simulation shows sending of start bit, followed by 7-bit address of the slave device(0x1A), followed by write(0) bit, followed by 8-bit register address(0x0C), followed by 8-bit register data(0x10) and acknowledgments from the slave device.



4.2 IP Core generation workflow

IP Core generation steps for SSM2603 configuration model are same as the steps mentioned above in section 3.2, IP Core generation workflow. Generated IP core can be used in user reference designs. For creating the reference design, refer to "Authoring a Reference Design for Audio System on a ZYBO Board" on page 40-236.

5 Configuring audio codec SSM2603 on Arrow SoC Development Kit using I2C Master Controller library block

This section shows how to:

1. Model audio codec SSM2603 device configuration chart using Stateflow blocks in simulink.
2. Use I2C Master controller library block to configure audio codec SSM2603.
3. Perform simulation of created model.

To configure the audio codec SSM2603 on Arrow SoC Development Kit, device configuration chart for SSM2603 need to be created. This chart should be connected to the I2C Master controller library block created earlier.

Note: The device configuration chart used for SSM2603 is specific to this device and can't be used to configure other devices.

The configuration model for SSM2603 on Arrow SoC Development Kit is same as configuration model for SSM2603 on Zybo board. Please refer to section 4 of this article for SSM2603 configuration model.

5.1 Simulating audio codec SSM2603 configuration model

The audio codec chip SSM2603 configuration on Arrow SoC Development Kit is same as audio codec chip SSM2603 configuration on Zybo board. Please refer to section 4.1 of this article for simulation.

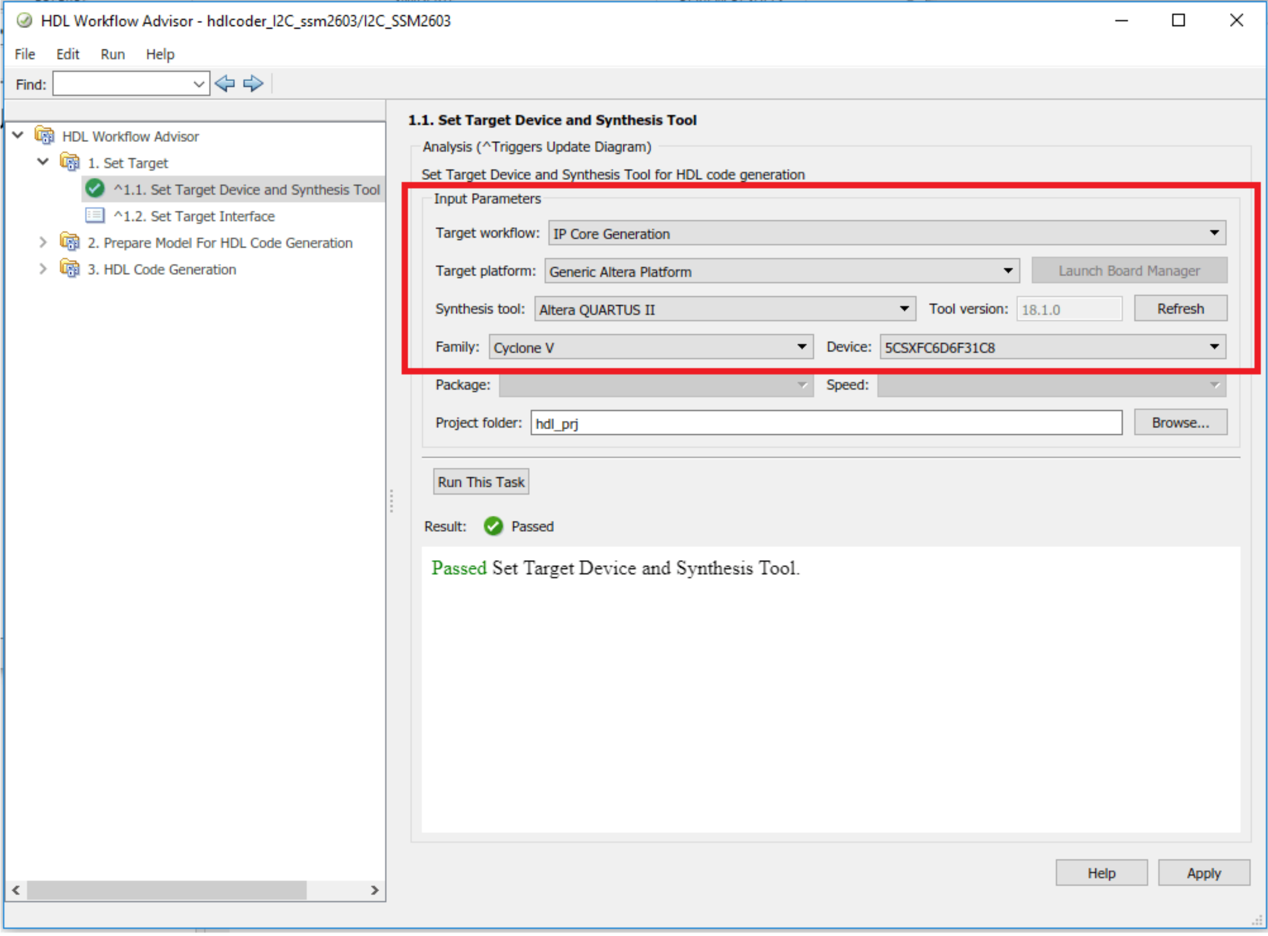
5.2 IP Core generation workflow

To generate the audio codec SSM2603 configuration HDL IP core, follow the steps given below.

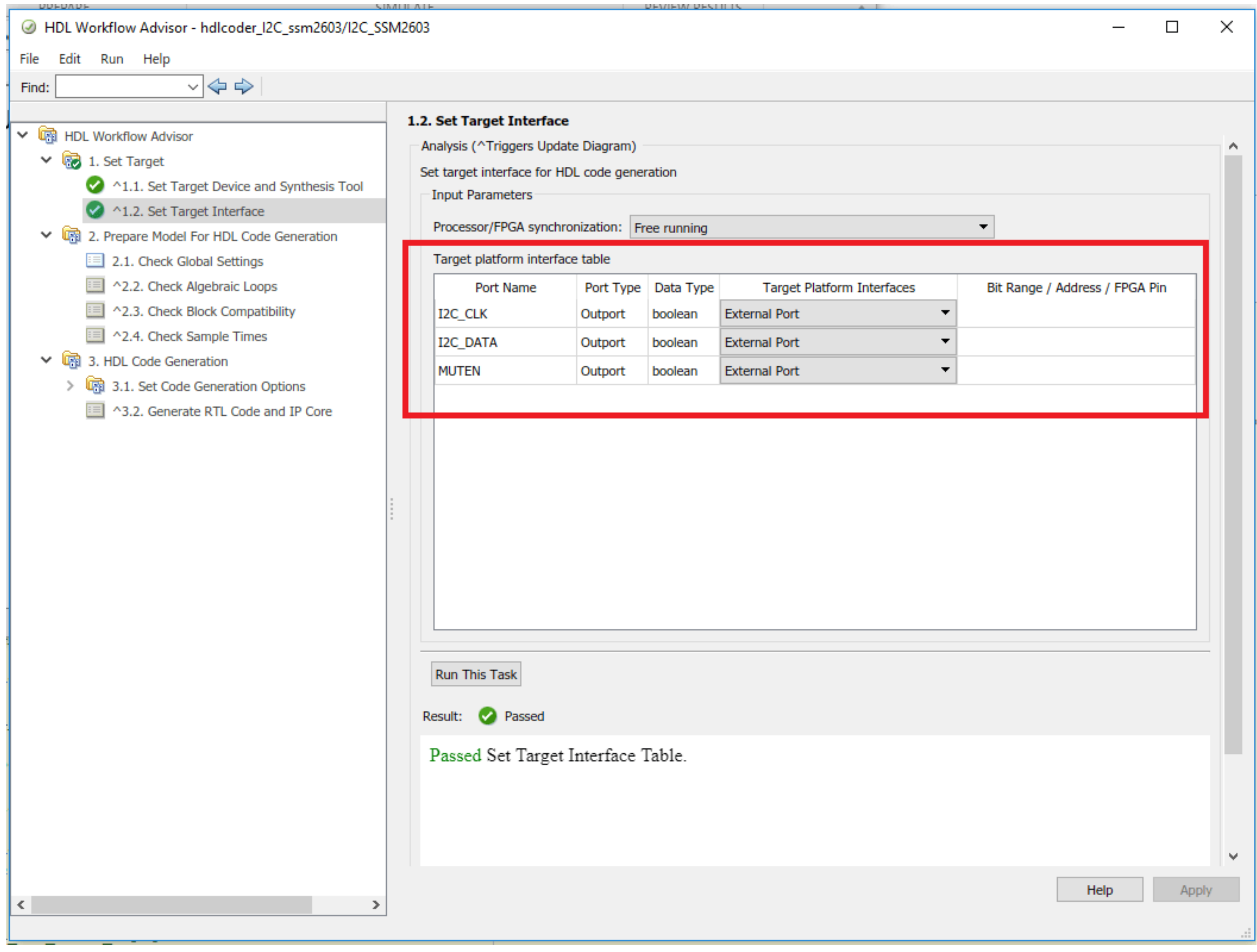
1. Set up the Intel Quartus tool path using the following command in the MATLAB command window. Use your own Quartus installation path when you run the command

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', 'ToolPath', 'C:\intelFPGA\18.1\quartus\bin64\qu
```

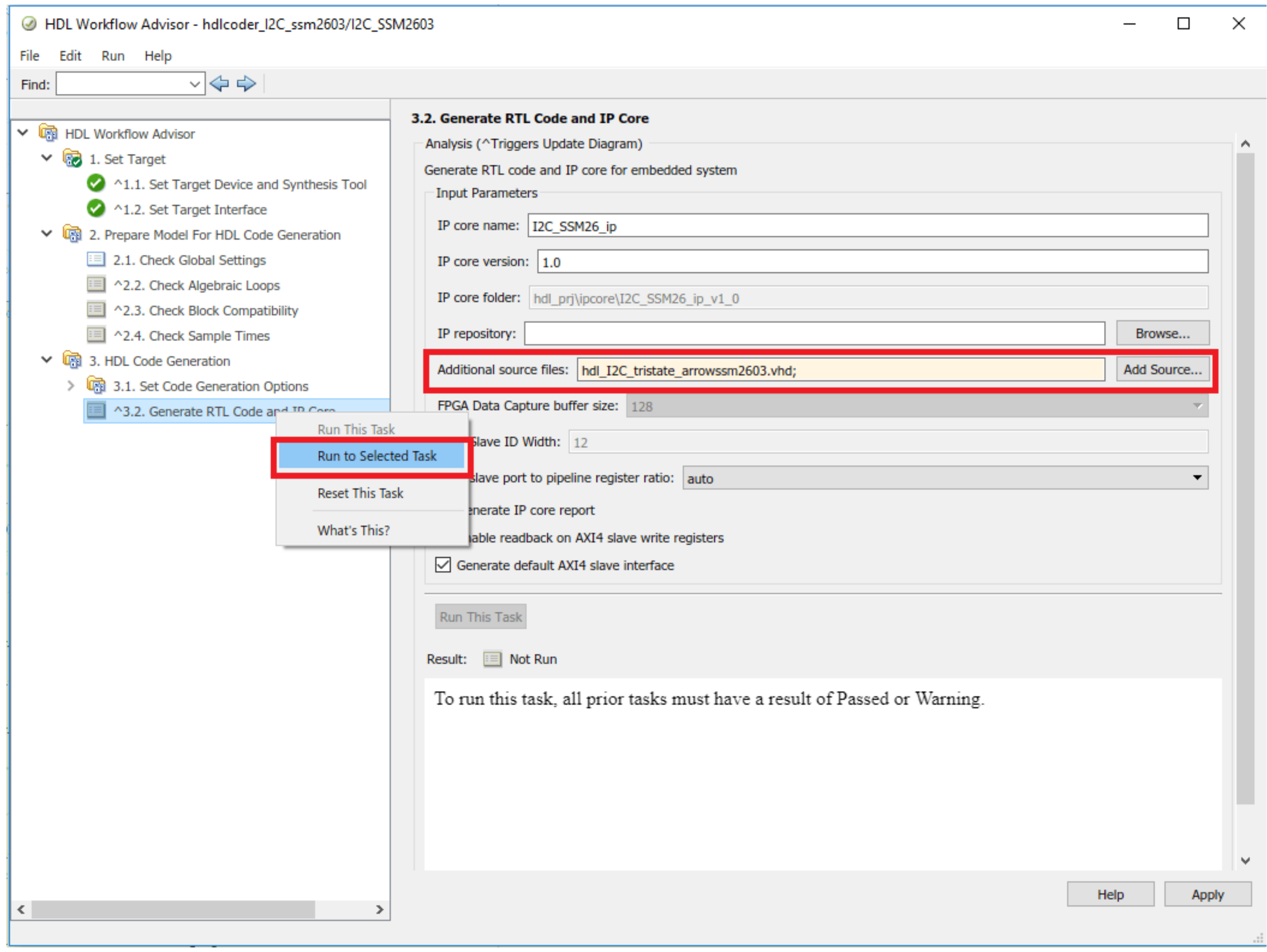
2. In the SSM2603 configuration model, select I2C_SSM2603 subsystem and by right clicking open HDL workflow advisor. In Task 1.1, Select **IP Core Generation** for Target workflow, **Generic Altera Platform** for Target platform and **Altera QUARTUS II** for Synthesis Tool. Also select family, device, package and speed as shown in the figure below.



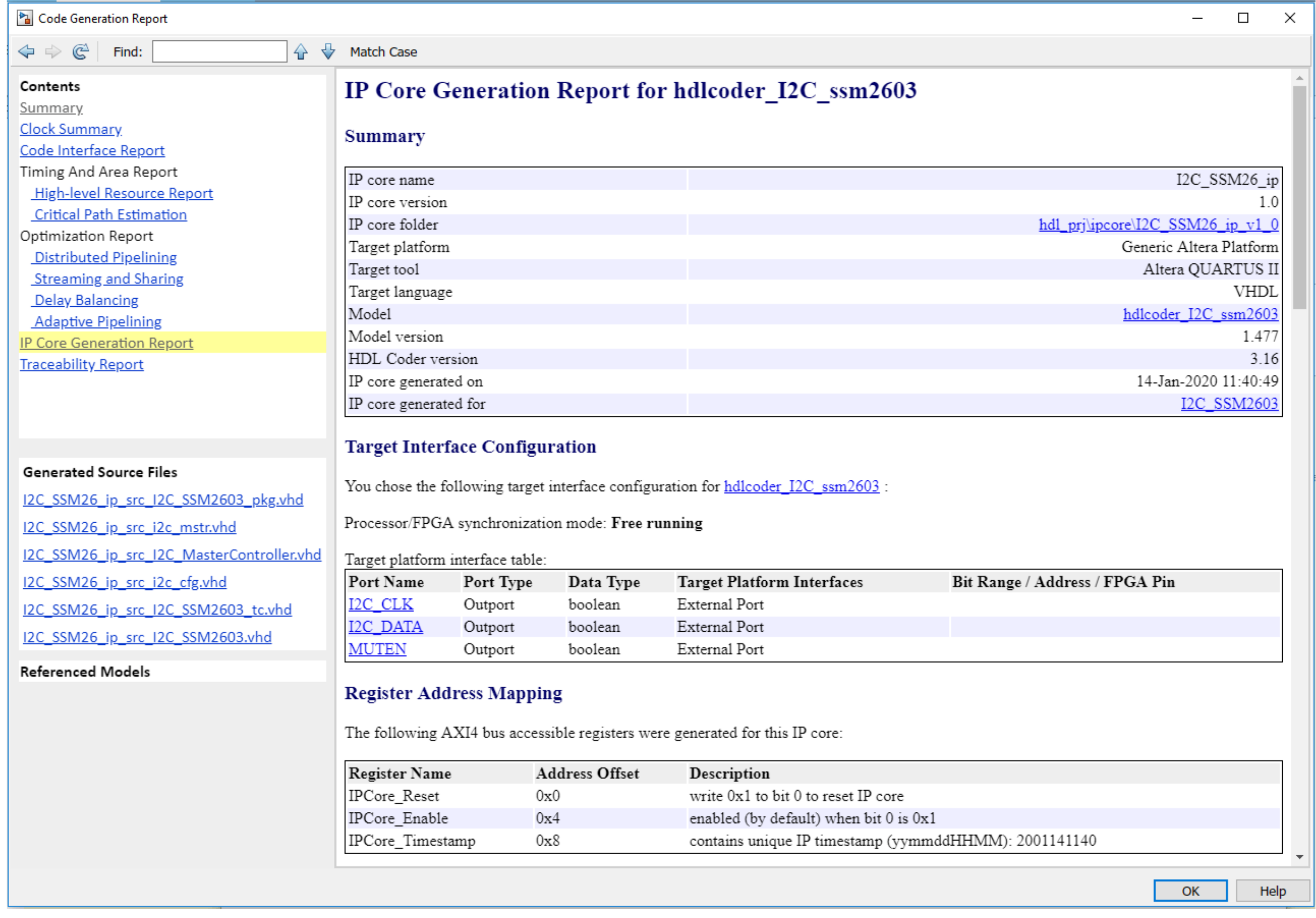
3. In Task 1.2, set the Target Platform Interfaces to "External Port" for all the ports.



4. In Task 3.2, add the tristate buffer Verilog file in the additional source files. Then right click on Generate RTL code and IP Core and click on Run to Selected Task.



I2C IP Core for configuration of SSM2603 will be generated. Below figure shows the IP Core generation report.

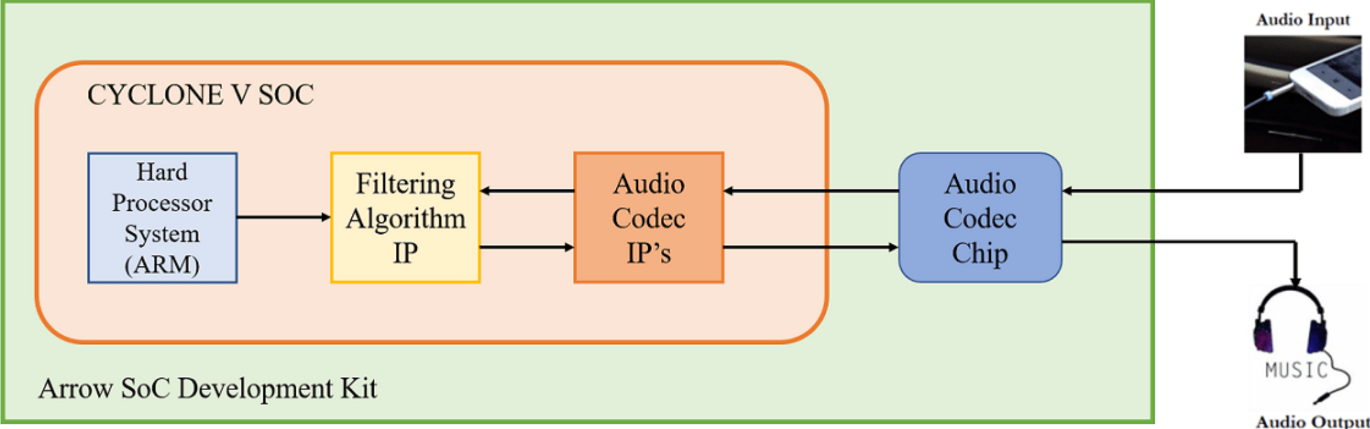


Generated IP core can be used in user reference designs. For creating the reference design, refer to “Authoring a Reference Design for Audio System on Intel Board” on page 40-242.

Running an Audio Filter on Live Audio Input Using Intel Board

In this example, we illustrate how to:

1. Model an audio system with Low pass and Band pass filters
2. Implement it on a Intel® board using an audio reference design



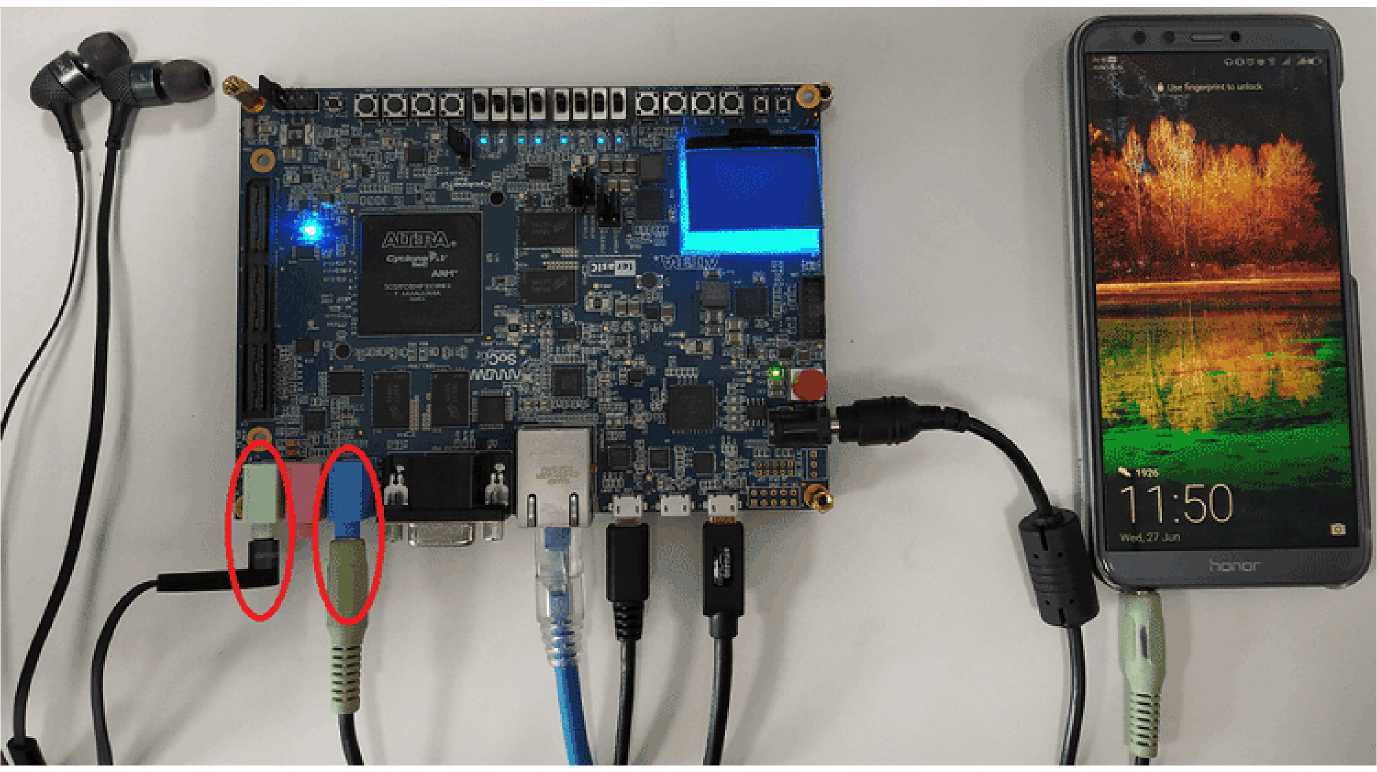
The objective of this example is to receive audio input through Arrow SoC Development Kit's line input, process it on the FPGA and transmit the processed audio to a speaker. The above figure shows the high-level architecture of such a system. It uses an audio codec to interface to the peripherals and to convert analog to digital signals and vice-versa. The Audio Codec IPs are used to configure the audio codec and for transferring audio data between Intel SoC and audio codec. The Filter IP is used for audio processing. ARM processor is used to control the type of filter to be used i.e. low pass or band pass.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder™ Support Package for Intel FPGA and SoC Devices
- Embedded Coder® Support Package for Intel SoC Devices
- Intel Quartus® Prime Standard, with supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- Intel SoC Embedded Design Suite
- Arrow SoC Development Kit

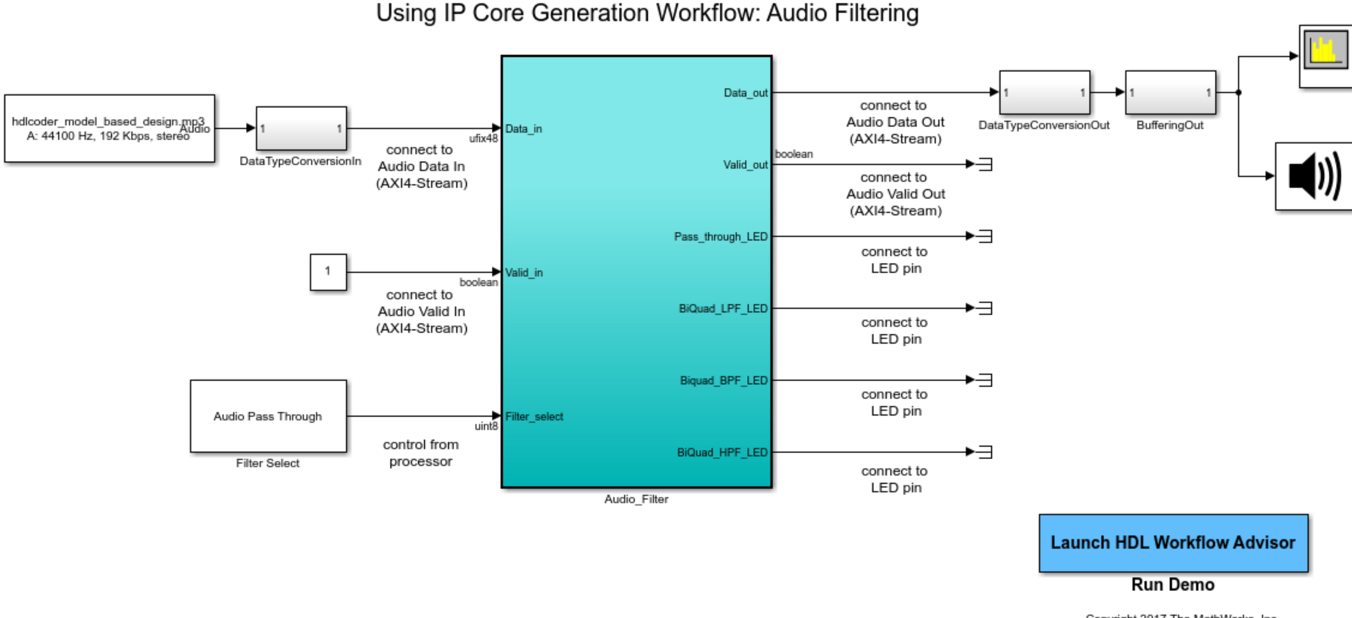
To setup the Arrow SoC Development Kit, refer to the Set up Intel SoC hardware and tools section in the “Getting Started with Targeting Intel SoC Devices” on page 39-132 example. Connect an audio input from a mobile or an MP3 player to **LINE IN** jack and either earphones or speakers to **LINE OUT** jack on the Arrow SoC as shown below.



Introduction

In the following model, an audio file is used as input to the DUT subsystem, **Audio_filter**. On simulating this model in Simulink, the processed audio effect can be heard through the **Audio Device Writer** block and **Spectrum Analyzer** block displays the spectrogram of the filtered audio output.

```
modelname = 'hdlcoder_audio_filter_biquad';
open_system(modelname);
```



Model a System with Low pass and Band pass Filters

Filter coefficients may be generated using a MATLAB function or in Simulink. In this model, `filterDesigner` (DSP System Toolbox) (DSP System Toolbox) tool is used to generate the filter coefficients for each type of filter. Then these filter coefficients are exported and stored as a matlab file. These coefficients will be used to design the filters in Simulink. In this model, discrete IIR filter blocks from Simulink® are used as Biquad low pass or band pass filters depending on the corresponding filter coefficients.

You can test this model by simulating the model in Simulink. The range of frequencies seen on the Spectrum Analyzer and the audio effect heard through the Audio Device Writer block should vary depending on the type of filter selected. **Filter Select** block is used to select the type of filtering to be done on the audio input.

Customize the Model for Arrow SoC Development Kit

In order to implement this model on Arrow SoC, you must first create a reference design in Qsys which receives audio input on Arrow SoC and transmits the processed audio data out of Arrow SoC. For details on how to create a reference design which integrates the audio filter model, refer to “Authoring a Reference Design for Audio System on Intel Board” on page 40-242 example.

In the reference design, left and right channel audio data are combined to form a single channel. They are concatenated such that lower 24 bits is the left channel and upper 24 bits is the right channel. In the Simulink model shown above, Data_in is split into 2 channels i.e. left and right accordingly. Their magnitude is divided by 2 and the 2 channels are added together to form a single channel. Filtering is done on this channel.

Data_in and **Valid_in** are the AXI4-Stream signals. **Data_in** contains the audio data to be processed and **Valid_in** acts as the enable signal. Each filter is mapped to an LED on Arrow SoC to visually indicate whether the filter is on or off.

FilterSelect input is controlled via AXI4 interface.

Generate HDL IP core with AXI4-Stream Interface

Next, you can start the HDL Workflow Advisor and use the Intel® hardware-software co-design workflow to deploy this design on the Intel hardware. For a more detailed step-by-step guide, you can refer to the "Getting Started with Targeting Intel SoC Devices" on page 39-132 example.

1. Set up the Intel Quartus synthesis tool path using the following command in the MATLAB command window. Use your own Intel Quartus installation path when you run the command. For example:

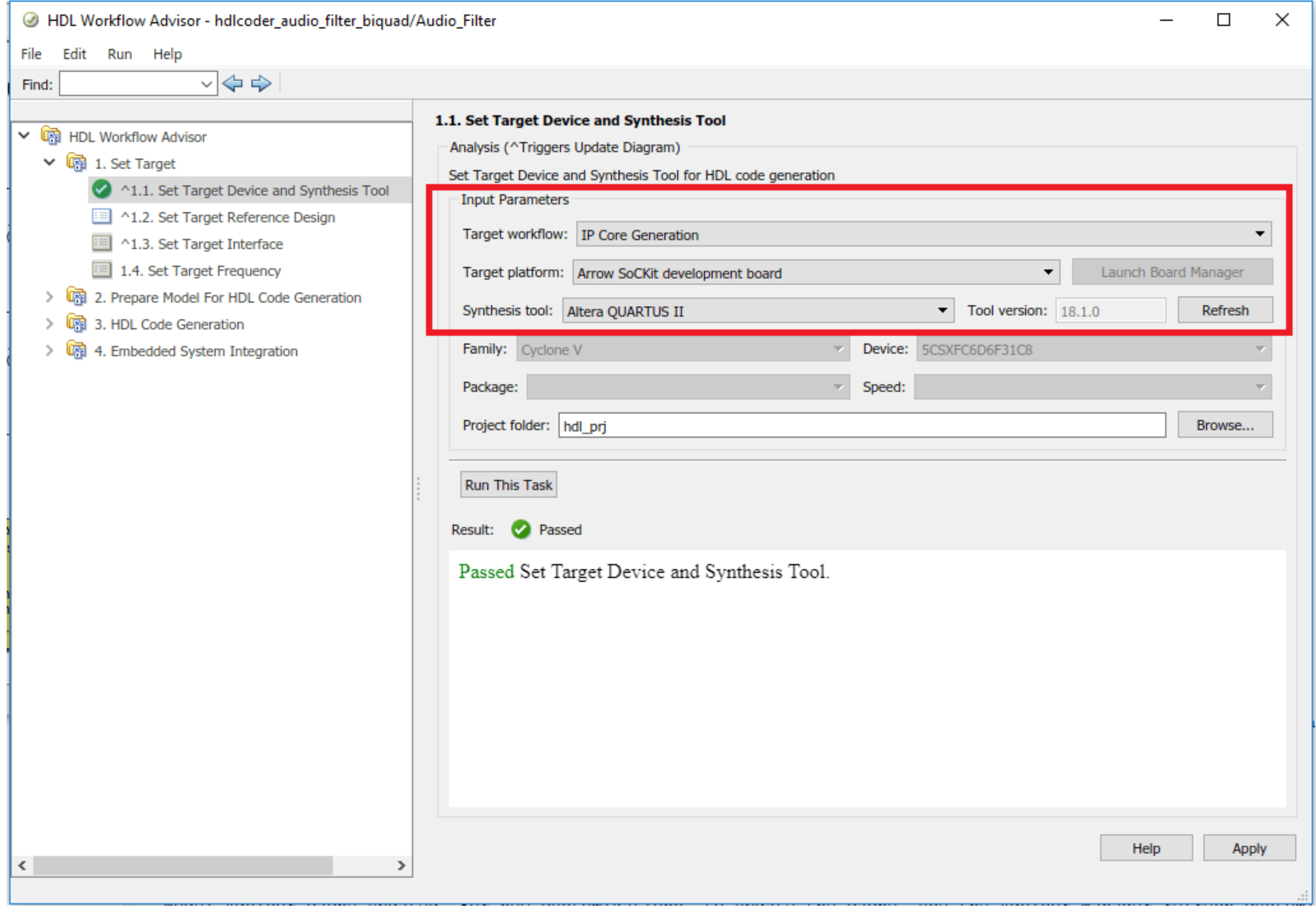
```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', 'ToolPath', 'C:\intelFPGA\18.1\quartus\bin64\qu
```

2. Add both the IP repository folder and the Arrow SoC Development Kit registration file to the MATLAB path using following commands:

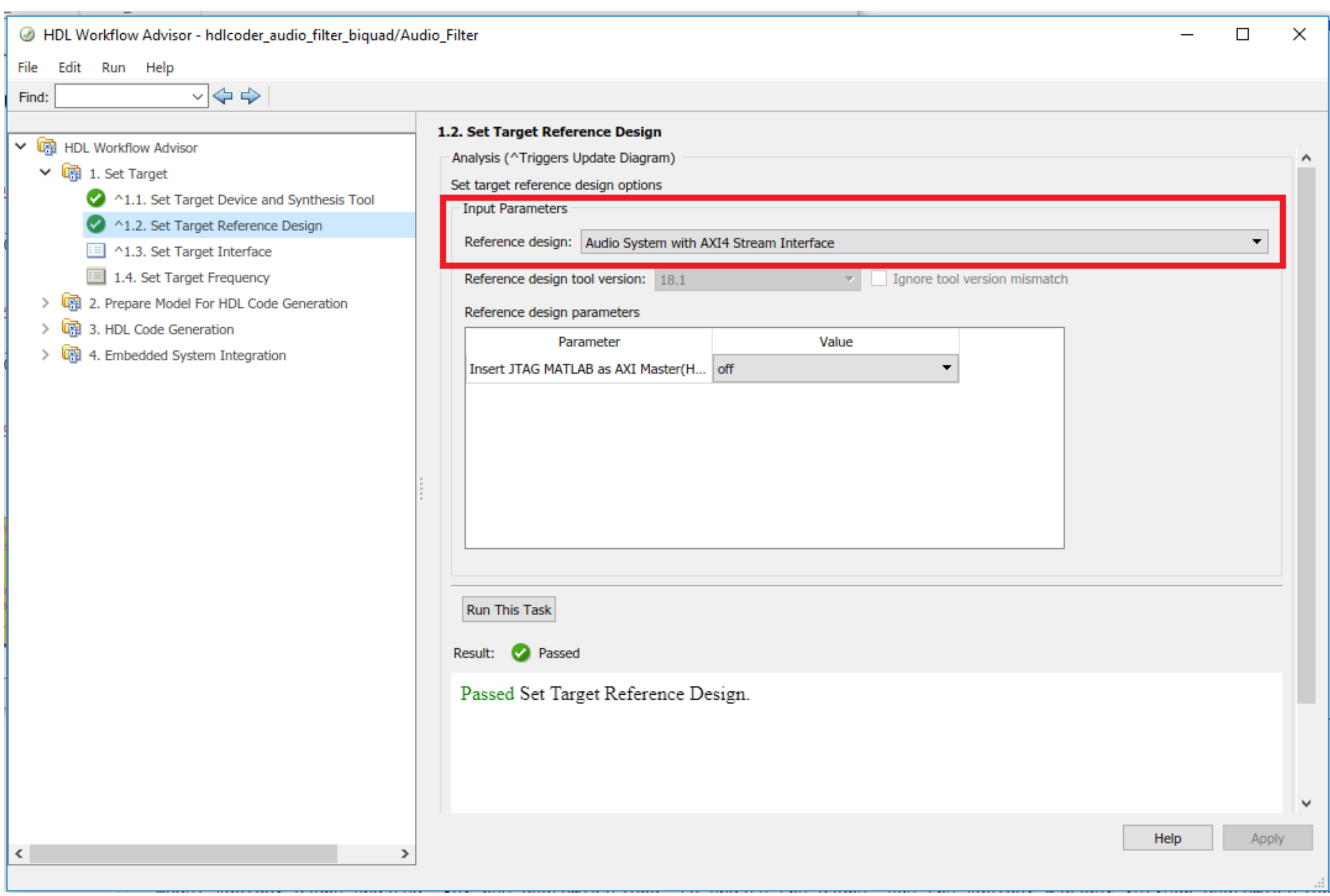
```
example_root = (hdlcoder_intel_examples_root)
cd (example_root)
addpath(genpath('ipcore'));
addpath(genpath('ArrowSoC'));
```

3. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_audio_filter_biquad/Audio_filter` or by double-clicking the Launch HDL Workflow Advisor box in the model.

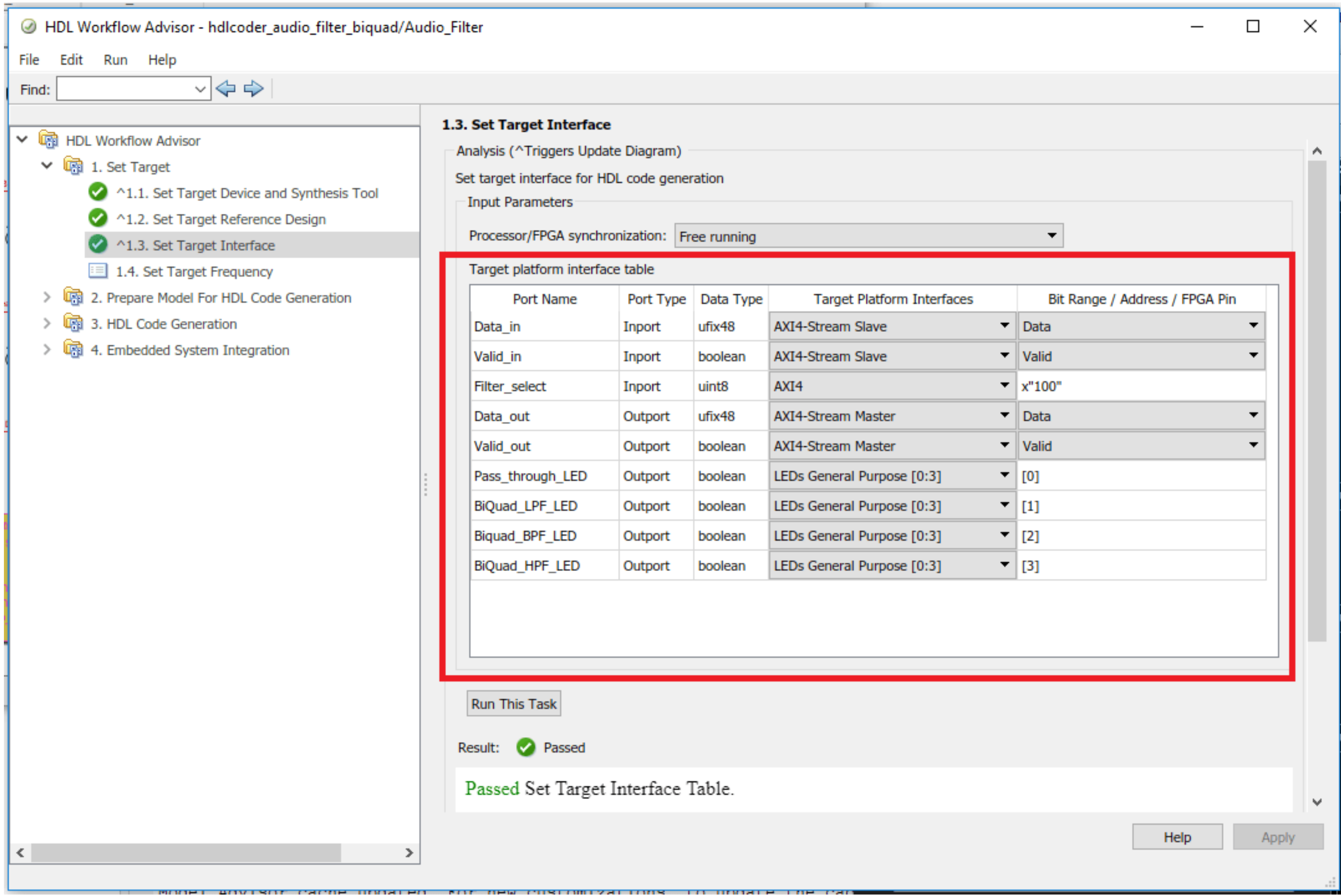
In Task 1.1, select **IP Core Generation** for **Target workflow**, and select `Arrow SoC Development Kit` for **Target platform**.



In Task 1.2, `Audio System with AXI4 Stream Interface` is selected for **Reference Design**.



The AXI4-Stream interface is used for transferring audio data between the reference design and the filtering algorithm IP. The AXI4-Stream interface contains data (**Data**) and control signals such as data valid (**Valid**), back pressure (**Ready**), and data boundary (**TLAST**). At least **Data** and **Valid** signals are required for AXI4-Stream IP core generation. In Task 1.3, the **Target platform interface table** is loaded as shown in the following picture. The audio data stream ports, **Valid_in**, **Data_in**, **Valid_out** and **Data_out**,are mapped to the AXI4-Stream interfaces, **Pass_through_LED, BiQuad_LPF_LED, BiQuad_BPF_LED** are mapped to the LEDs on Arrow SoC and the control parameter port **Filter_select** is mapped to the AXI4 interface.



The AXI4-Stream interface communicates in master/slave mode, where the master device sends data to the slave device. Therefore, if a data port is an input port, assign it to an **AXI4-Stream Slave** interface, and if a data port is output port, assign it to an **AXI4-Stream Master** interface.

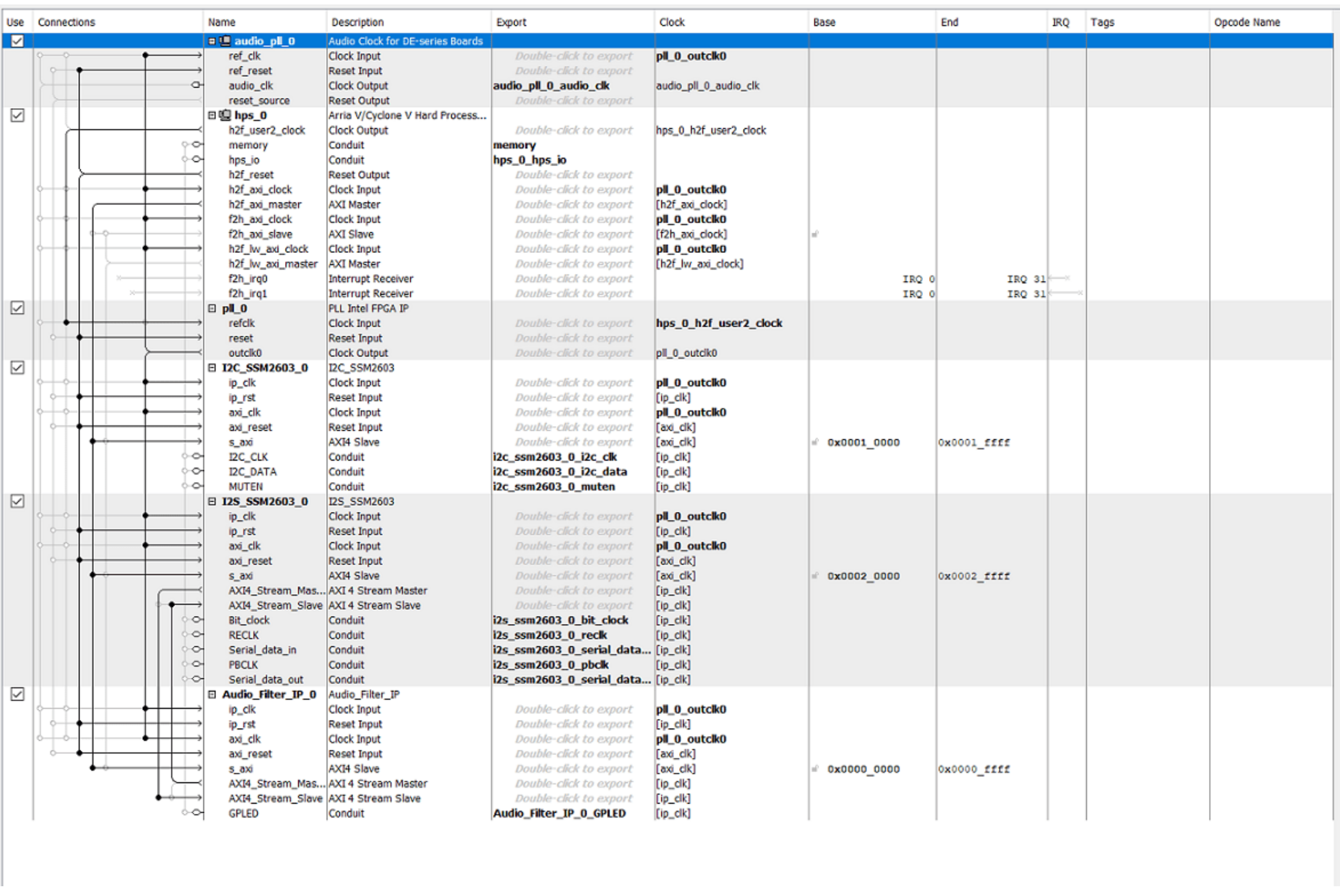
3. Right-click Task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

Integrate IP Into AXI4-Stream Audio Compatible Reference Design

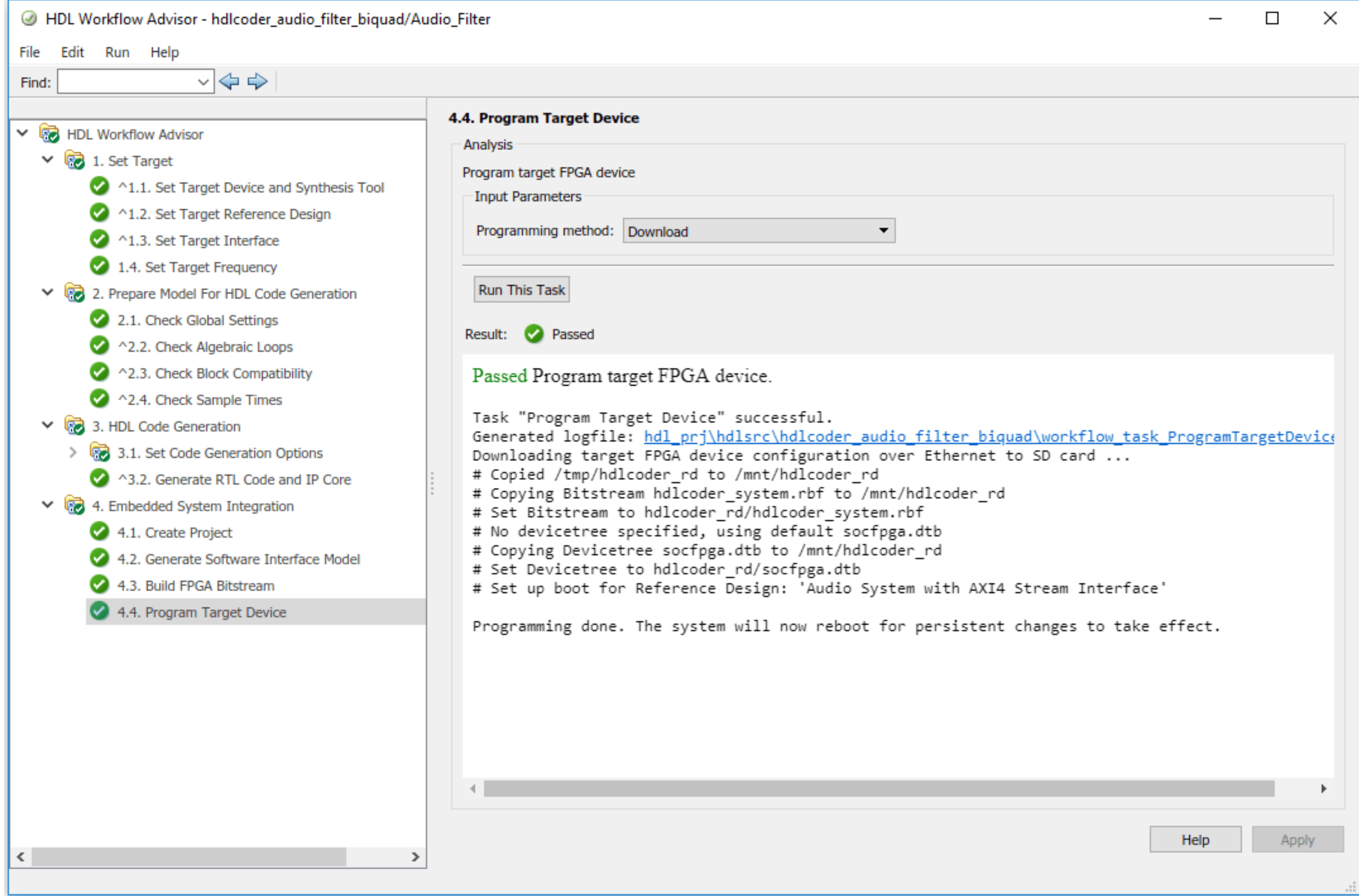
Next, in the HDL Workflow Advisor, you run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Intel SoC.

1. Run Task 4.1, **Create Project**. This task inserts the generated IP core into the **Audio System with AXI4 Stream Interface** reference design. As shown in the first diagram, this reference design contains the IPs to handle audio data in and out of Arrow SOC. The generated project is a complete design, including the algorithm part (the generated DUT algorithm IP), and the platform part (the reference design). For details on how to create a reference design which integrates the audio filter model, refer to “Authoring a Reference Design for Audio System on Intel Board” on page 40-242 example.

2. Click on the **Generated Altera Qsys project** link in the Result pane to open the generated platform designer Qsys project.



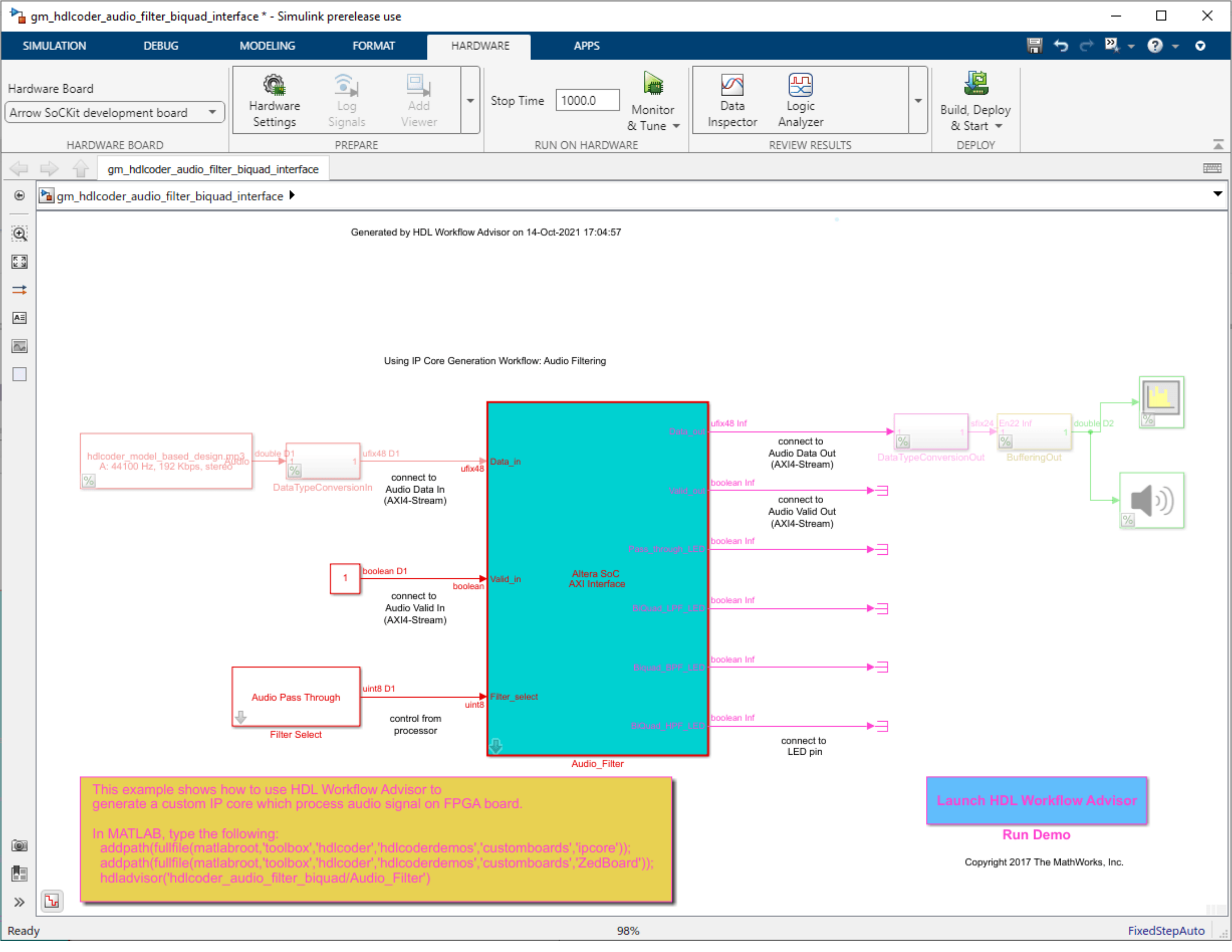
3. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model, and build and download the FPGA bitstream. Choose **Download** programming method in the task **Program Target Device** to download the FPGA bitstream onto the SD card on the Intel board, so your design will be automatically reloaded when you power cycle the Intel board.



Generate ARM executable to Tune Parameters on the FPGA Fabric

A software interface model is generated in Task 4.2, **Generate Software Interface Model**.

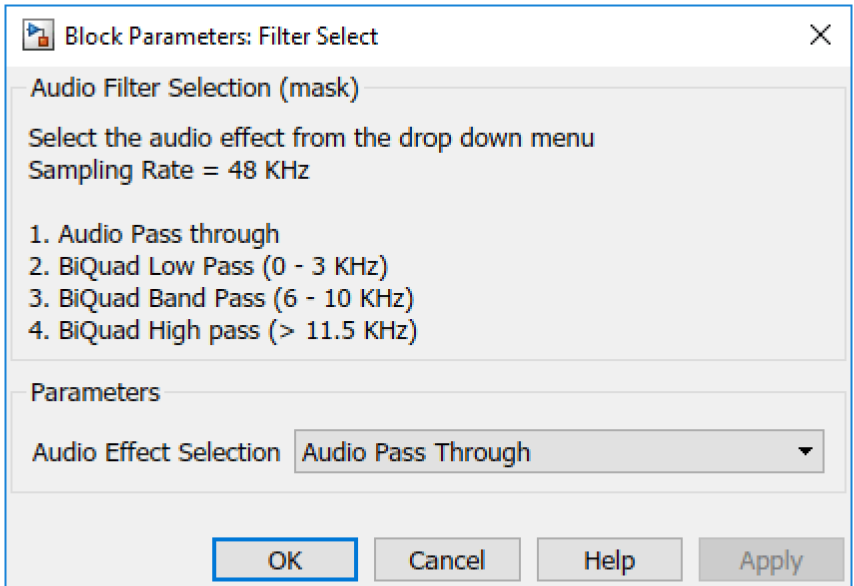
1. Before you generate code from the software interface model, comment out the audio input source and audio output sink i.e. From Multimedia File, Data Type Conversion, Buffer, Audio Device Writer and Spectrum Analyzer Blocks. These blocks do not need to be run on the ARM processor. The Audio_filter IP is running as `Filtering_Algorithm` on FPGA fabric. The ARM processor is using AXI4 interface for selecting the filter type i.e. Biquad Low pass, Band pass or Pass Through.



1. In the generated model, go to **Hardware** pane and then open the Configuration Parameters dialog box by clicking on **Hardware Settings**.

1. Select **Solver** and set **Stop Time** to **inf**.
2. Then in the **Hardware Pane**, Click on **Monitor & Tune** button. Embedded Coder builds the model, downloads the ARM executable to the Intel board hardware, executes it, and connects the model to the executable running on the Intel board hardware. Then you can tune model parameters.

The type of filter to be used can be selected using the drop down options in **Filter Select** block



The filtered audio output can be heard by plugging earphones or speakers to **LINE OUT** jack on the Arrow SoC. Depending on the filter selected, the corresponding LED on the Arrow SoC turns on. In this example, LD0 turns on when Pass through (No filter used) option is selected, LD1 turns on when Biquad Low pass filter is selected, LD2 turns on when Biquad Band pass filter is selected.

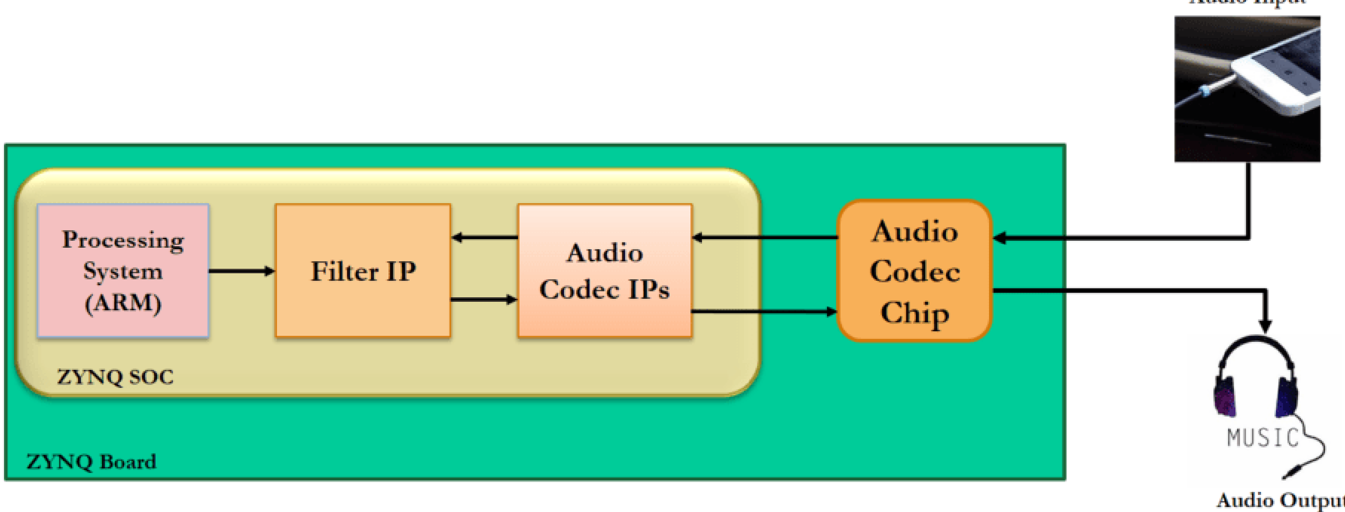
Running an Audio Filter on Live Audio Input Using a Zynq Board

This example shows how to model an audio system and implement it on a Zynq® board using an audio reference design.

Introduction

In this example, you:

1. Model an audio system with Low pass, Band pass and High pass filters.
2. Implement it on a Zynq board using an audio reference design.



The objective of this example is to receive audio input through Zedboard or Zybo board's line input, process it on the FPGA and transmit the processed audio to a speaker. The above figure shows the high-level architecture of such a system. It uses an audio codec to interface to the peripherals and to convert analog to digital signals and vice-versa. The Audio Codec IPs are used to configure the audio codec and for transferring audio data between Zynq Soc and audio codec. The Filter IP is used for audio processing. ARM processor is used to control the type of filter to be used i.e. low pass, band pass or high pass.

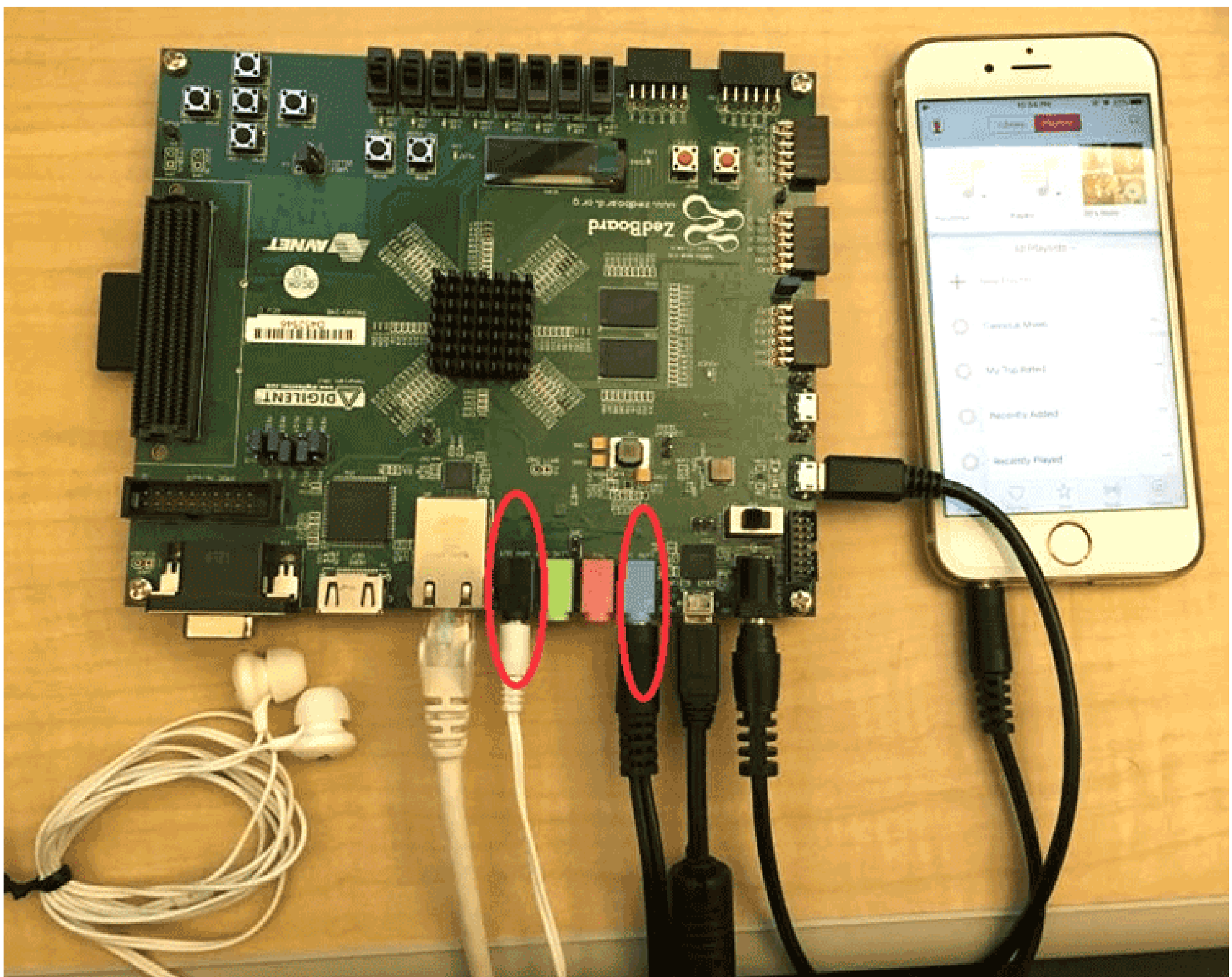
Before You Begin

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder™ Support Package for Xilinx® FPGA and SoC Devices
- Xilinx Vivado® version 2019.1
- ZedBoard™ or Zybo Board

To setup the Zedboard board, refer to the *Set up Zynq hardware and tools* section in the “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98 example. Connect an audio input from a mobile or an MP3 player to **LINE IN** jack and either earphones or speakers to **HPH OUT** jack on the

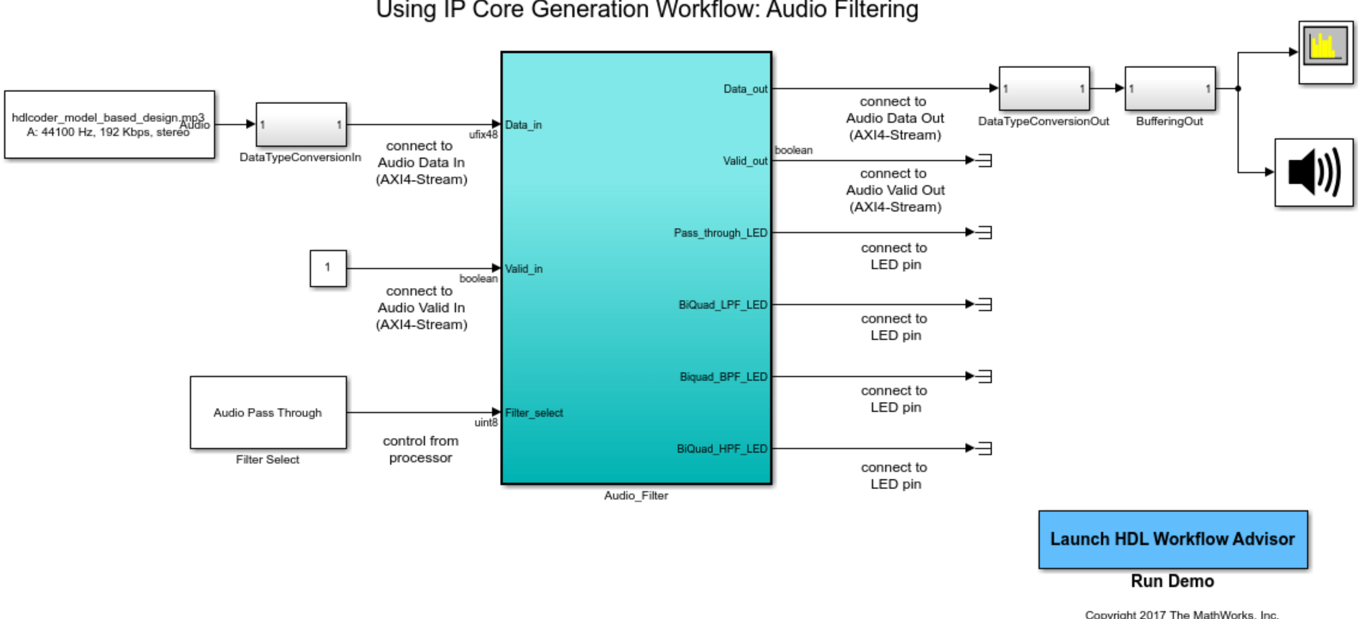
Zedboard as shown below. Similar setup can be done on Zybo board. For Zybo board setup, refer to *Set up the Zybo board* section in the “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252 example.



Introduction

In the following model, an audio file is used as input to the DUT subsystem, **Audio_filter**. On simulating this model in Simulink, the processed audio effect can be heard through the **Audio Device Writer** block and **Spectrum Analyzer** block displays the spectrogram of the filtered audio output.

```
modelname = 'hdlcoder_audio_filter_biquad';
open_system(modelname);
```



Model a System with Low Pass, Band Pass and High Pass Filters

Filter coefficients may be generated using a MATLAB® function or in Simulink®. In this model, filterDesigner tool is used to generate the filter coefficients for each type of filter. Then these filter coefficients are exported and stored as a MATLAB file. These coefficients will be used to design the filters in Simulink. In this model, discrete IIR filter blocks from Simulink are used as Biquad low pass, band pass or high pass filters depending on the corresponding filter coefficients.

You can test this model by simulating the model in Simulink. The range of frequencies seen on the Spectrum Analyzer and the audio effect heard through the Audio Device Writer block should vary depending on the type of filter selected. **Filter Select** block is used to select the type of filtering to be done on the audio input.

Customize the Model for Zynq Board

In order to implement this model on Zedboard, you must first create a reference design in Vivado which receives audio input on Zedboard and transmits the processed audio data out of Zedboard. For details on how to create a reference design which integrates the audio filter model, refer to “Authoring a Reference Design for Audio System on a Zynq Board” on page 40-226 example.

For Zybo board, refer to “Authoring a Reference Design for Audio System on a ZYBO Board” on page 40-236.

In the reference design, left and right channel audio data are combined together to form a single channel. They are concatenated such that lower 24 bits is the left channel and upper 24 bits is the right channel. In the Simulink model shown above, Data_in is split into 2 channels i.e. left and right accordingly. Their magnitude is divided by 2 and the 2 channels are added together to form a single channel. Filtering is done on this channel.

Data_in and **Valid_in** are the AXI4-Stream signals. To understand how AXI4-stream interface is used, refer to *Model Streaming Algorithm with Simplified Streaming Protocol* section in “Deploy Model with AXI-Stream Interface in Zynq Workflow” on page 40-192 example. **Data_in** contains the audio

data to be processed and **Valid_in** acts as the enable signal. Each filter is mapped to an LED on Zedboard or Zybo board to visually indicate whether the filter is on or off.

FilterSelect input is controlled via AXI4 LITE interface.

Generate HDL IP Core with AXI4-Stream Interface

Next, you can start the HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy this design on the Zynq hardware. For a more detailed step-by-step guide, you can refer to the “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98 example.

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.ba
```

2. Add both the IP repository folder and the Zedboard registration file to the MATLAB path using following commands:

```
example_root = (hdlcoder_amd_examples_root)
cd (example_root)
addpath(genpath('ipcore'));
addpath(genpath('ZedBoard'));
```

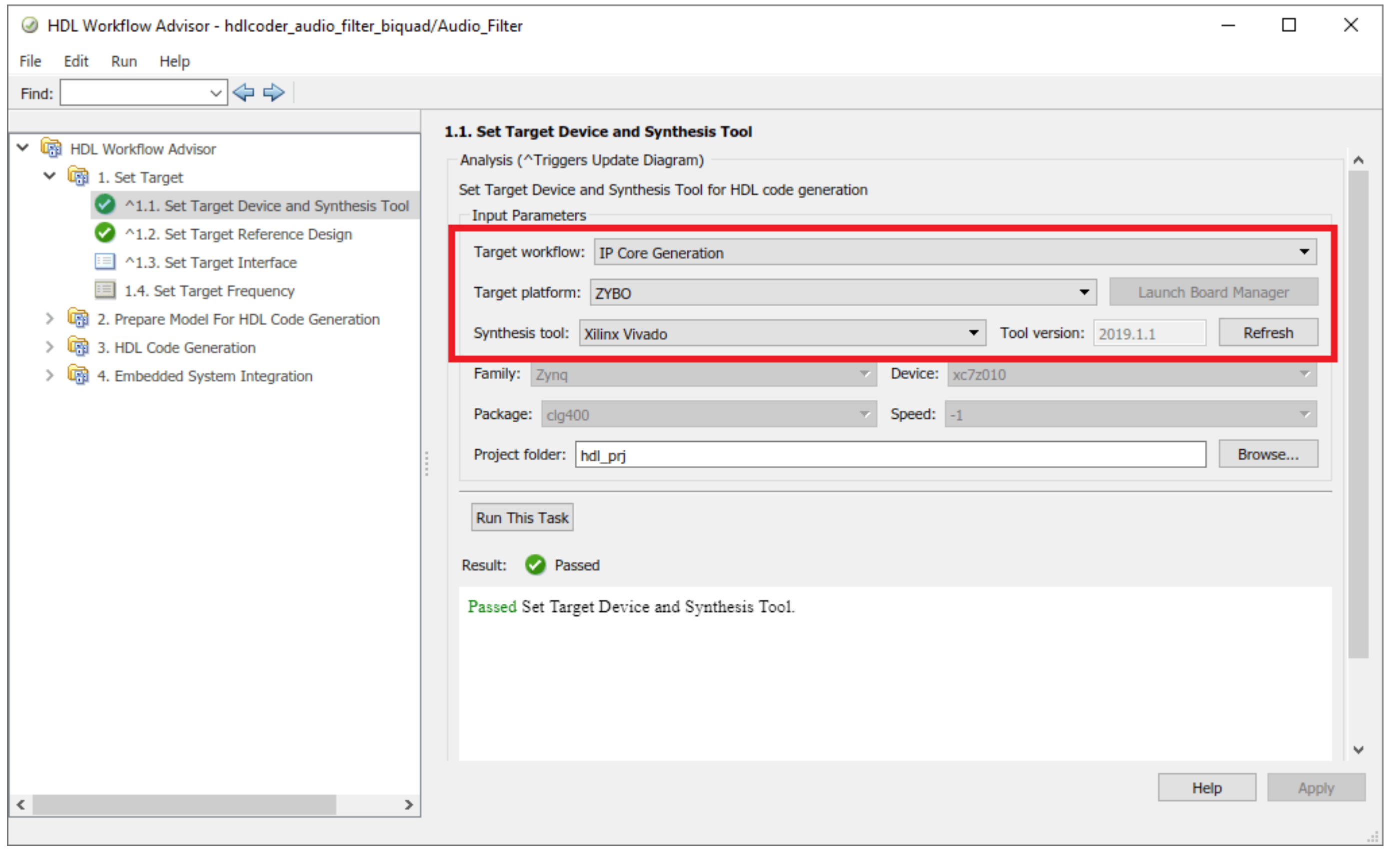
For Zybo board use the following commands.

```
example_root = (hdlcoder_amd_examples_root)
cd (example_root)
addpath(genpath('ipcore'));
addpath(genpath('ZYBO'));
```

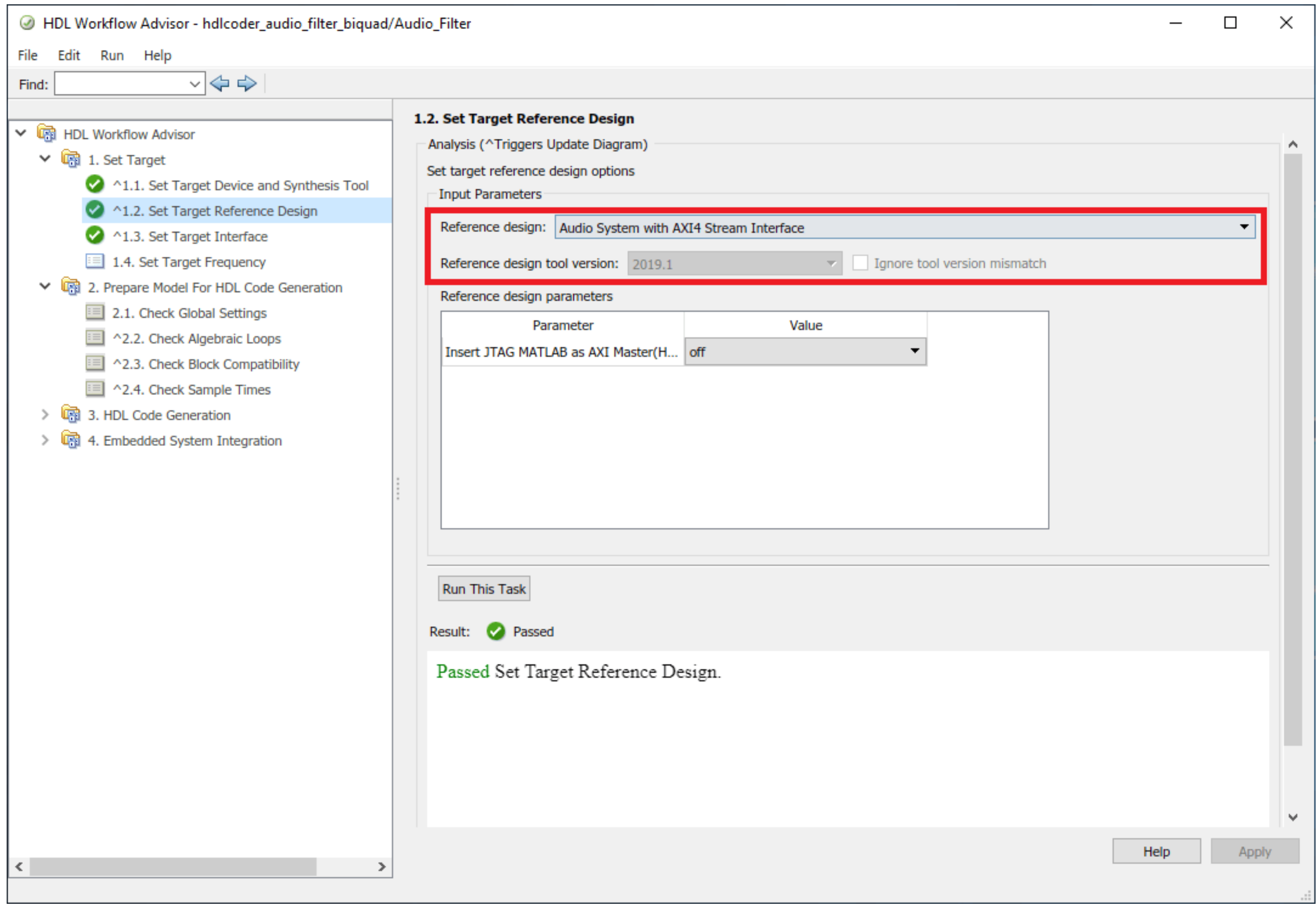
3. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_audio_filter_biquad/Audio_filter` or by double clicking on the Launch HDL Workflow Advisor box in the model.

The target interface settings are already saved for Zedboard in this example model, so the settings in Task 1.1 to 1.3 are automatically loaded. To learn more about saving target interface settings in the model, you can refer to the “Save Target Hardware Settings in Model” on page 39-177 example.

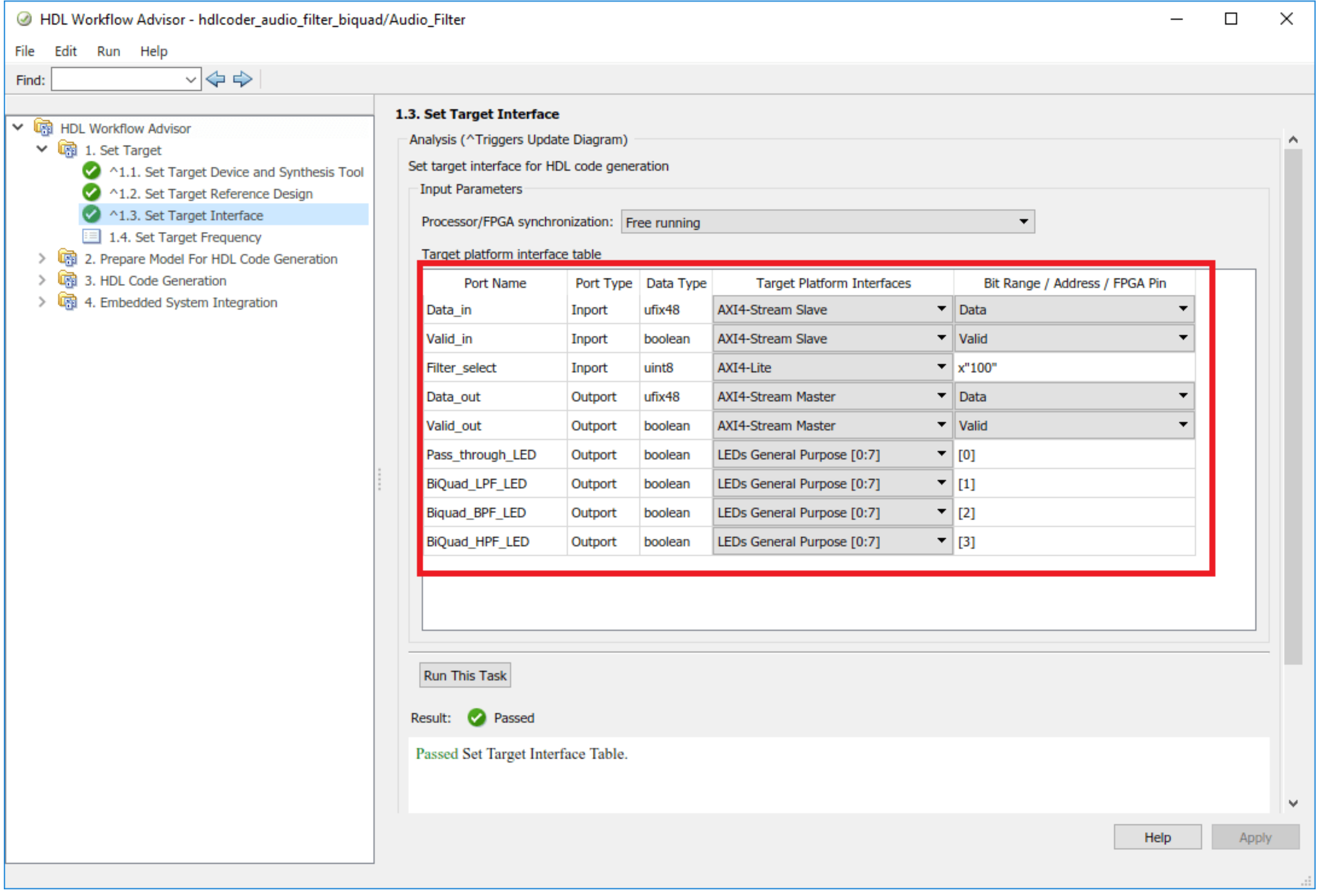
In Task 1.1, **IP Core Generation** is selected for **Target workflow**, and **ZedBoard** is selected for **Target platform**. If you are using Zybo board then select **ZYBO** as **Target Platform** instead of Zedboard.



In Task 1.2, **Audio System with AXI4 Stream Interface** is selected for **Reference Design**.



The AXI4-Stream interface is used for transferring audio data between the reference design and the filtering algorithm IP. The AXI4-Stream interface contains data (**Data**) and control signals such as data valid (**Valid**), back pressure (**Ready**), and data boundary (**TLAST**). At least **Data** and **Valid** signals are required for AXI4-Stream IP core generation. In Task 1.3, the **Target platform interface table** is loaded as shown in the following picture. The audio data stream ports, **Valid_in**, **Data_in**, **Valid_out** and **Data_out**,are mapped to the AXI4-Stream interfaces, **Pass_through_LED, BiQuad_LPF_LED, BiQuad_BPF_LED,BiQuad_HPF_LED** are mapped to the LEDs on Zedboard and the control parameter port **Filter_select** is mapped to the AXI4-Lite interface. If you are using Zybo board then map the LEDs to the filter manually by selecting **LEDs General Purpose[0:4]**.



The AXI4-Stream interface communicates in master/slave mode, where the master device sends data to the slave device. Therefore, if a data port is an input port, assign it to an **AXI4-Stream Slave** interface, and if a data port is output port, assign it to an **AXI4-Stream Master** interface.

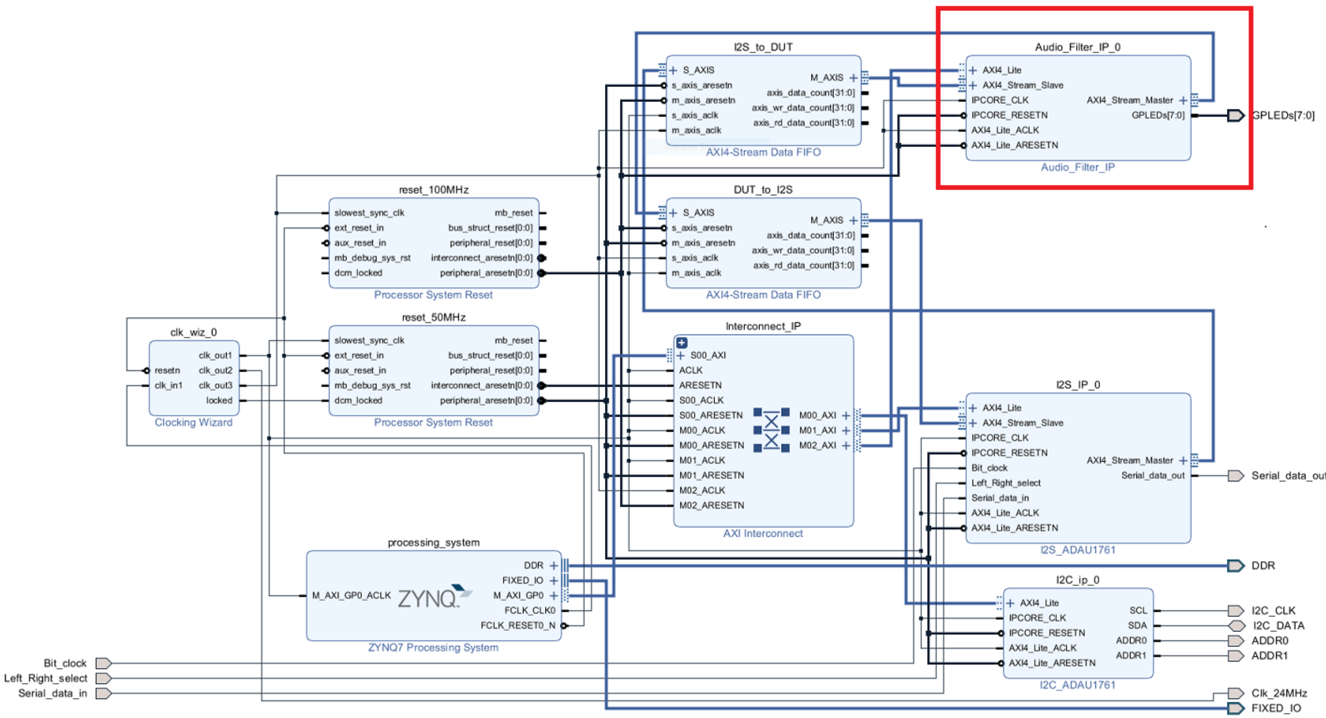
3. Right-click Task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

Integrate IP into AXI4-Stream Audio Compatible Reference Design

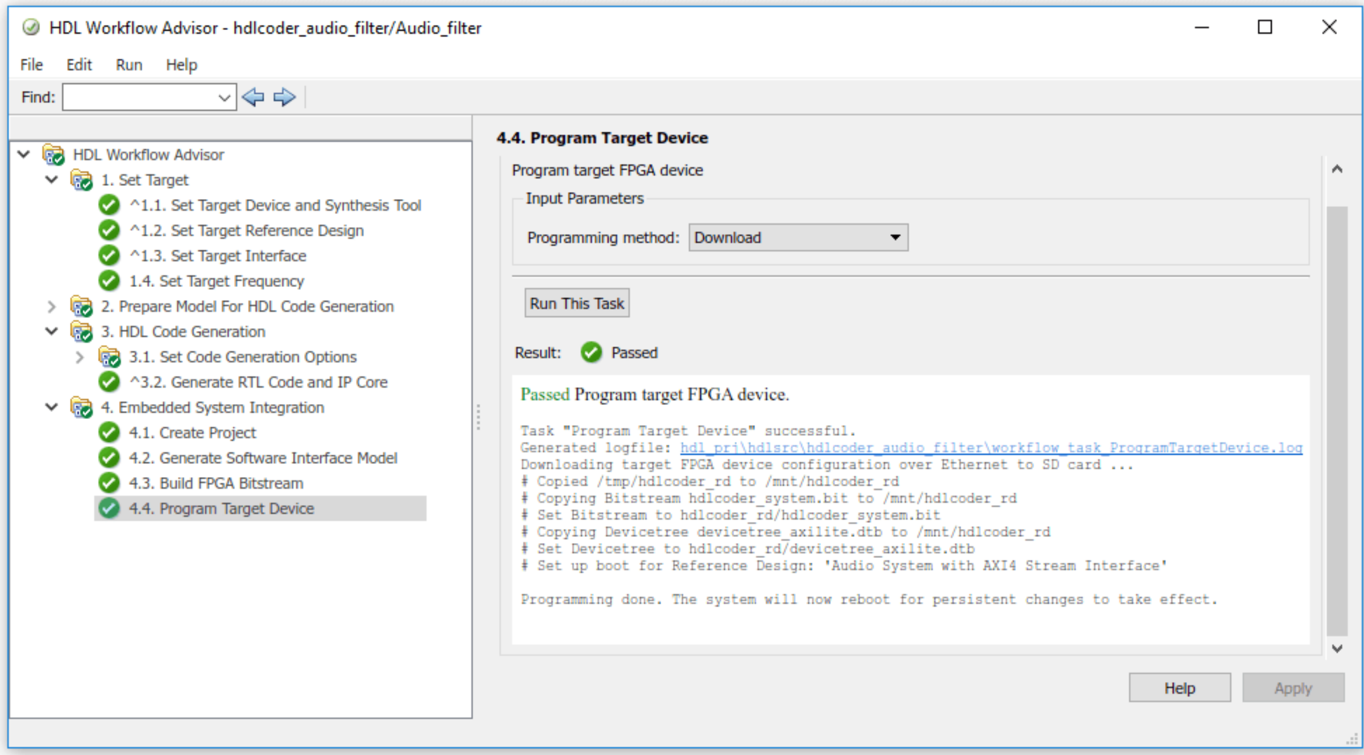
Next, in the HDL Workflow Advisor, you run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Zynq hardware.

1. Run Task 4.1, **Create Project**. This task inserts the generated IP core into the **Audio System with AXI4 Stream Interface** reference design. As shown in the first diagram, this reference design contains the IPs to handle audio data in and out of Zedboard. The generated project is a complete Zynq design, including the algorithm part (the generated DUT algorithm IP), and the platform part (the reference design). For details on how to create a reference design which integrates the audio filter model, refer to “Authoring a Reference Design for Audio System on a Zynq Board” on page 40-226 or “Authoring a Reference Design for Audio System on a ZYBO Board” on page 40-236 example.

2. Click the link in the Result pane to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram, which includes the generated HDL IP core, other audio processing IPs and the Zynq processor.



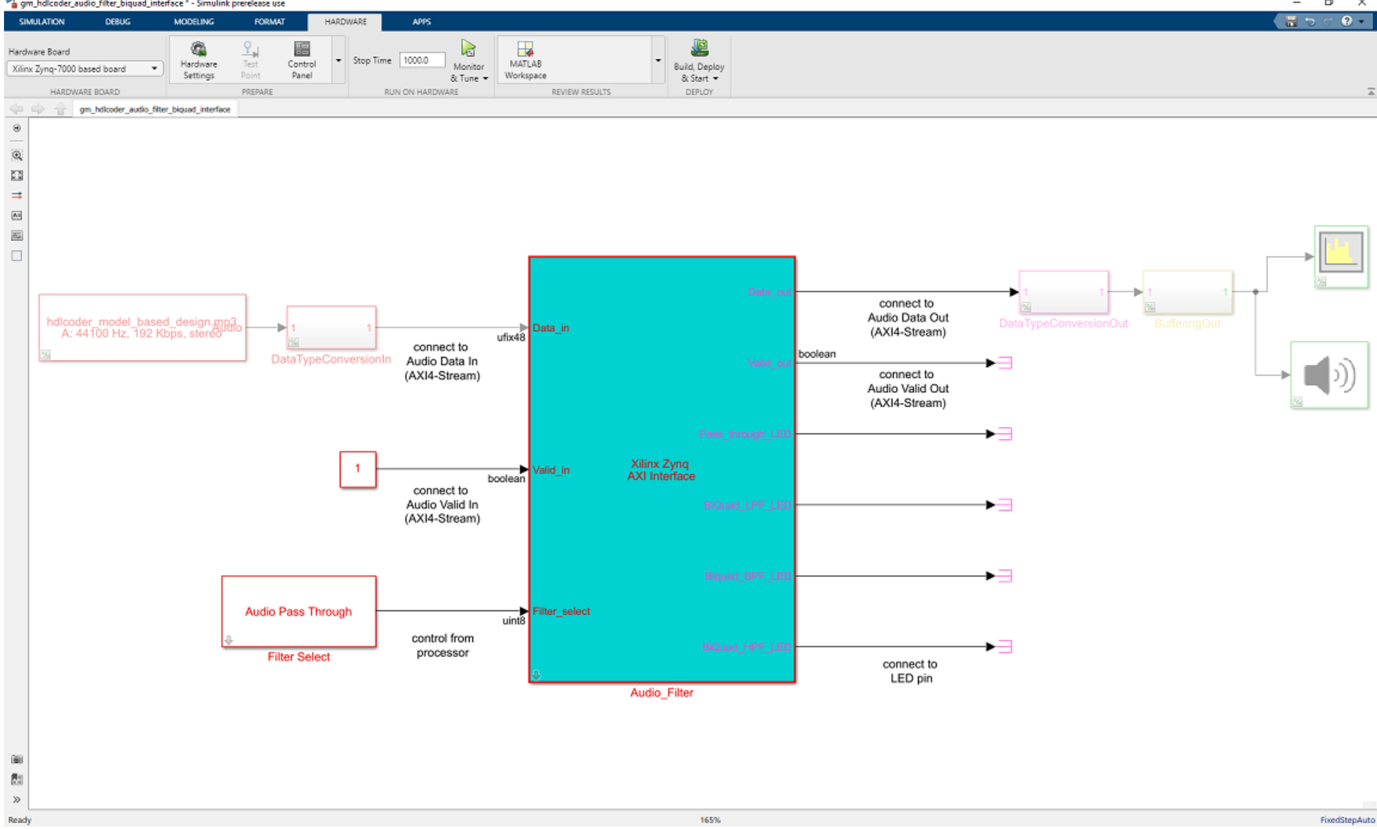
3. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model, and build and download the FPGA bitstream. Choose **Download** programming method in the task **Program Target Device** to download the FPGA bitstream onto the SD card on the Zynq board, so your design will be automatically reloaded when you power cycle the Zynq board.



Generate ARM Executable to Tune Parameters on the FPGA Fabric

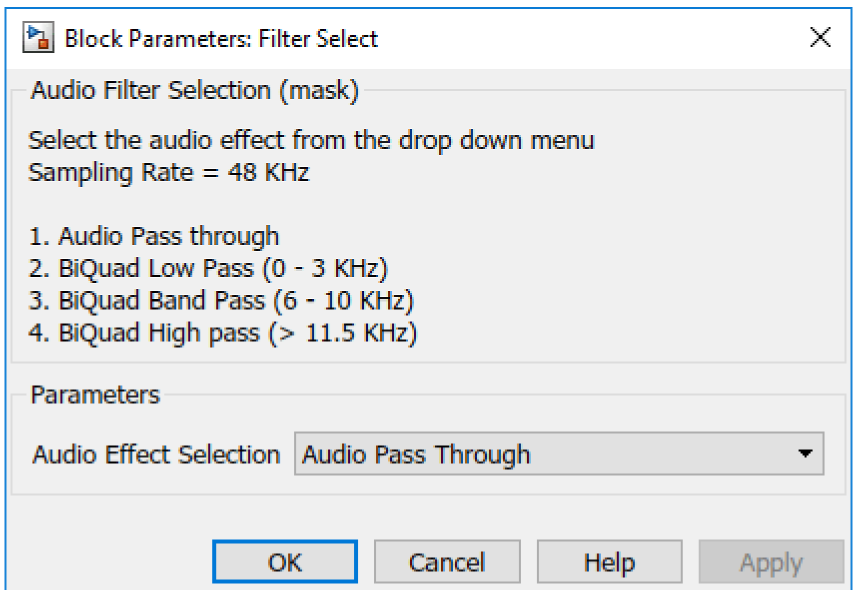
A software interface model is generated in Task 4.2, **Generate Software Interface Model**.

Before you generate code from the software interface model, comment out the audio input source and audio output sink i.e. From Multimedia File, Data Type Conversion, Buffer, Audio Device Writer and Spectrum Analyzer Blocks. These blocks do not need to be run on the ARM processor. The Audio_filter IP is running as *Filtering_Algorithm" on FPGA fabric. The ARM processor is using AXI4-Lite interface for selecting the filter type i.e. Biquad Low pass, band pass, High pass or Pass Through.



1. In the generated model,click on Hardware pane and go to **Hardware settings** to open **Configuration Parameter** dialog box.
2. Select **Solver** and set "Stop Time" to "inf" and click **ok**.
3. From the Hardware pane, click the **Monitor and Tune** button.
4. Click the **Run** button on the model toolstrip. Embedded Coder builds the model, downloads the ARM executable to the Zynq board hardware, executes it, and connects the model to the executable running on the Zynq board hardware.

The type of filter to be used can be selected using the drop down options in **Filter Select** block:



The filtered audio output can be heard by plugging earphones or speakers to **HPH OUT** jack on the Zynq board. Depending on the filter selected, the corresponding LED on the Zynq board turns on. In this example, LD0 turns on when Pass through (No filter used) option is selected, LD1 turns on when Biquad Low pass filter is selected, LD2 turns on when Biquad Band pass filter is selected and LD3 turns on when Biquad High pass filter is selected.

Deploy Model with AXI-Stream Interface in Zynq Workflow

This example shows how to use the AXI4-Stream interface to enable high speed data transfer between the processor and FPGA on Zynq® hardware.

Before You Begin

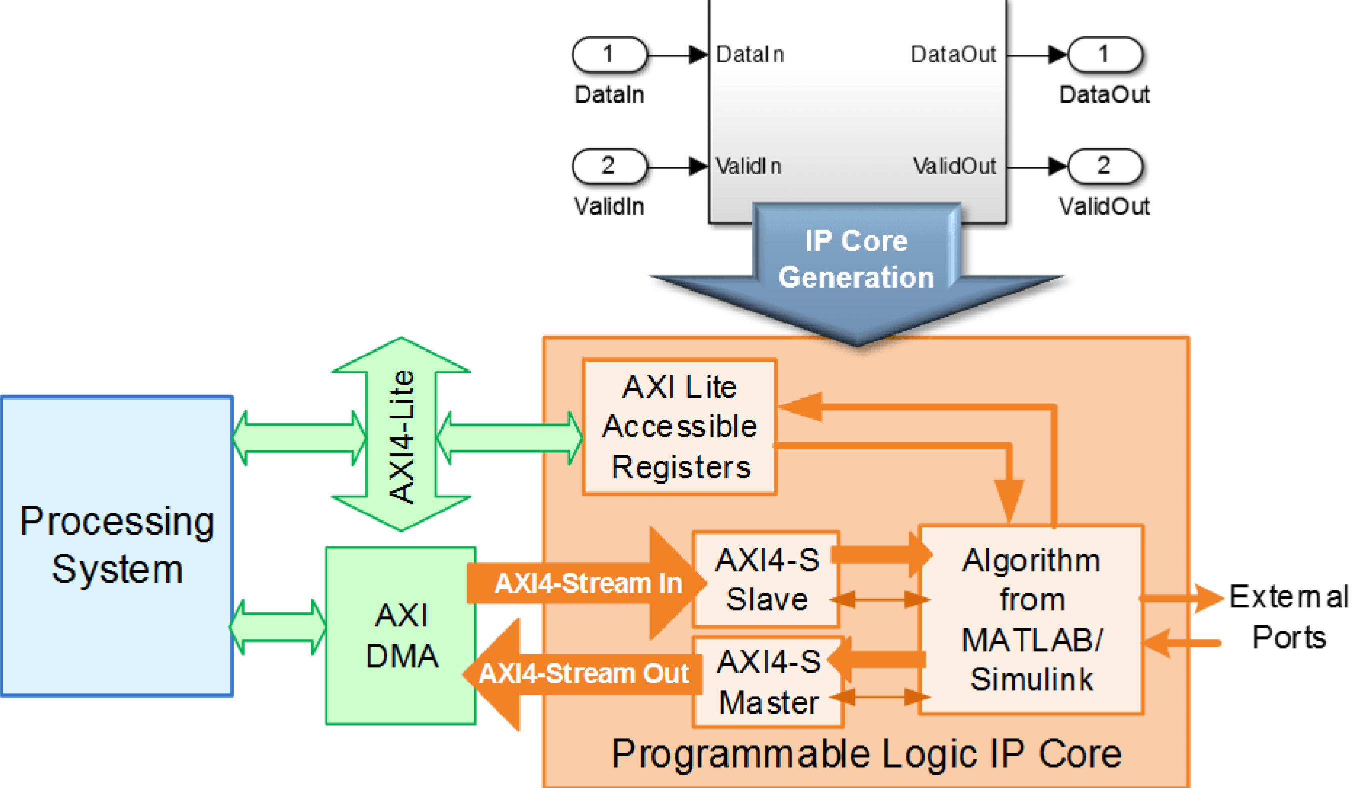
To run this example, you must have the following software and hardware installed and set up:

- Xilinx Vivado® Design Suite, with supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- Zedboard (To setup the Zedboard, refer to the *Set up Zynq hardware and tools* section in the example “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98.)

Introduction

This example shows how to:

1. Model a streaming algorithm using a simplified streaming protocol.
2. Generate an HDL IP core with AXI4-Stream interface.
3. Integrate the generated IP core into a Zedboard reference design with DMA controller.
4. Use the AXI4-Stream driver block to generate C code that runs on an ARM processor.



The figure above is a high level architecture diagram that shows a streaming data transfer between the processor and FPGA fabric on Zynq platform. Typically, the AXI4-Stream interface is used

together with a Direct Memory Access (DMA) controller to transfer a large chunk of data from the processor to FPGA. The data is usually represented as vector data in the software. The DMA controller reads the vector data from memory, and streams it to the FPGA IP through the AXI4-Stream interface. The streaming process sends one data element per sample, which means the data path of the streaming algorithm in the FPGA IP is using a scalar data type.

The FPGA IP can also include an AXI4-Lite interface for control signals or parameter tuning. Compared to the AXI4-Lite interface, the AXI4-Stream interface transfers data much faster, making it more suitable for the data path of an algorithm.

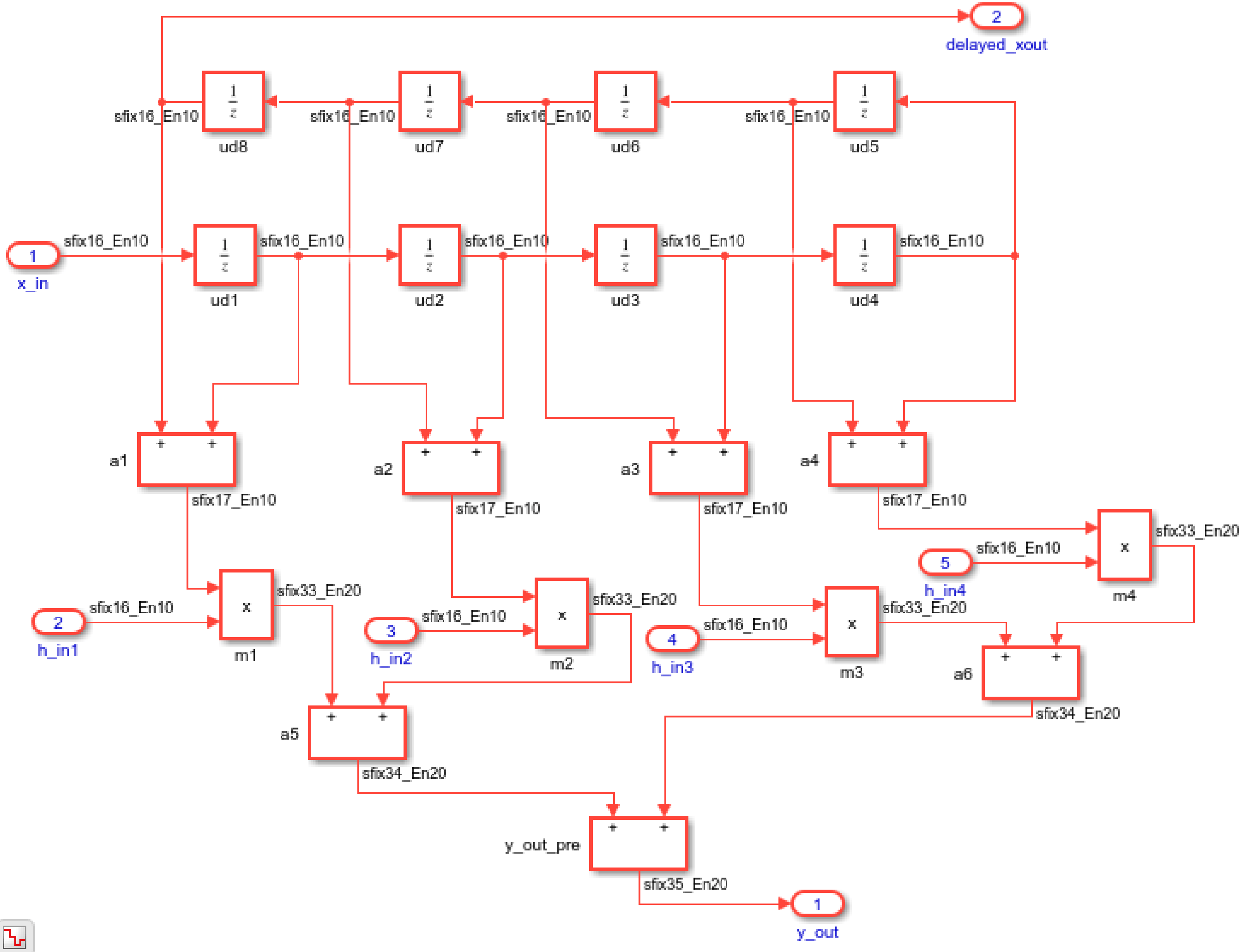
Other than connecting to processor, the FPGA IP with AXI4-Stream interface can also be connected with other IPs with AXI4-Stream interface to transfer data inside of FPGA.

Model Streaming Algorithm with Simplified Streaming Protocol

To deploy a simple symmetric FIR filter on Zynq hardware, implement the filter on FPGA. The ARM processor generates the source data to stream it to FPGA through the AXI4-Stream interface.

Consider the `sfir_fixed` model.

```
open_system('sfir_fixed')
set_param('sfir_fixed', 'SimulationCommand', 'Update')
```

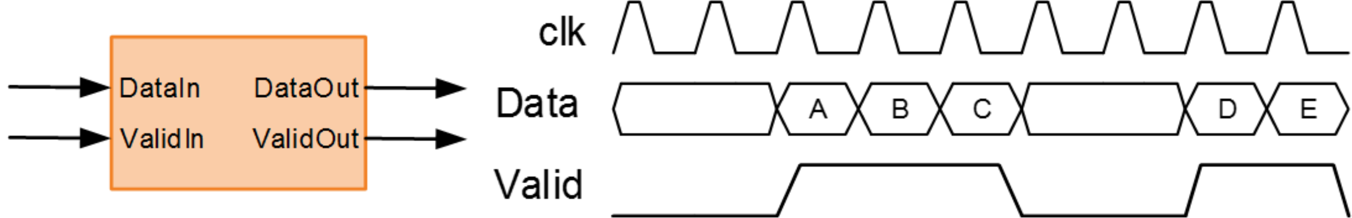


Note that the data path of this model (from `x_in` to `y_out`) is processing scalar input data, which is suitable for a streaming interface.

In order to enable data transfer from the software to the filter algorithm, we need to map the data path ports to the AXI4-Stream interface. The AXI4-Stream interface contains data (**Data**) and control signals such as data valid (**Valid**), back pressure (**Ready**), and data boundary (**TLAST**).

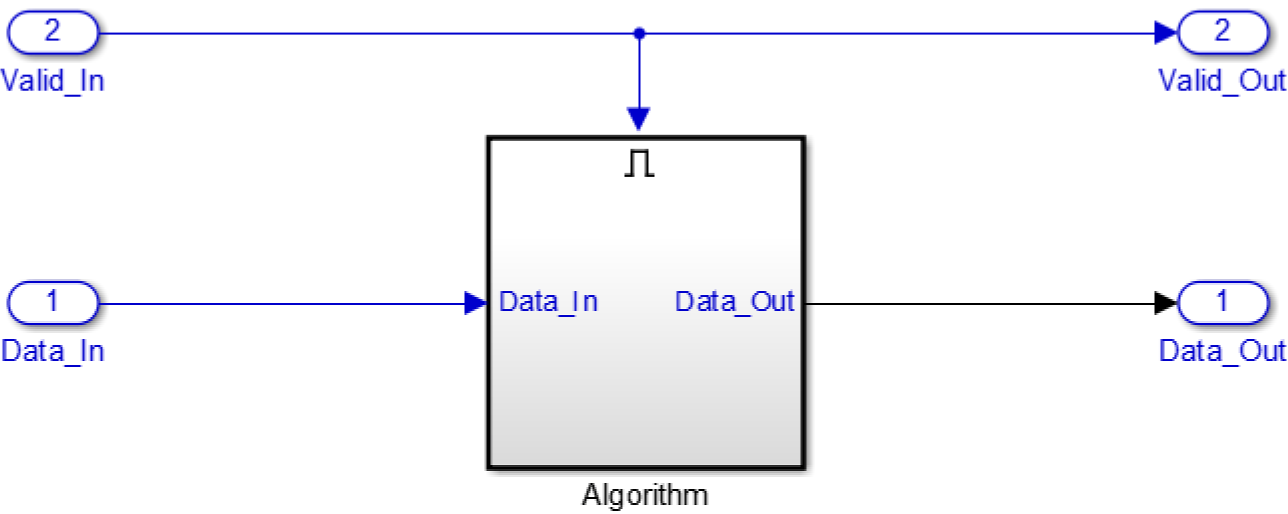
The AXI4-Stream IP core generation feature requires at least the **Data** and **Valid** signals to be modeled in the DUT. The **Data** signal is the primary payload to send across the interface. The **Valid** signal indicates when the **Data** signal is valid. Other control signals are optional.

Note: To generate IP Core, **Data** and **Valid** follow a simplified streaming protocol. You don't need to model the full AXI4-Stream protocol. HDL Coder automatically generates a streaming interface module in the HDL IP core to translate the simplified streaming protocol into the full AXI4-Stream protocol. As shown in the figure below, the protocol is that whenever the **Data** signal is valid, the **Valid** signal must also be asserted.



To map `sfir_fixed` algorithm to the simplified streaming protocol, a **Valid** signal needs to be added. To add the **Valid** signal to your model, use this modeling pattern:

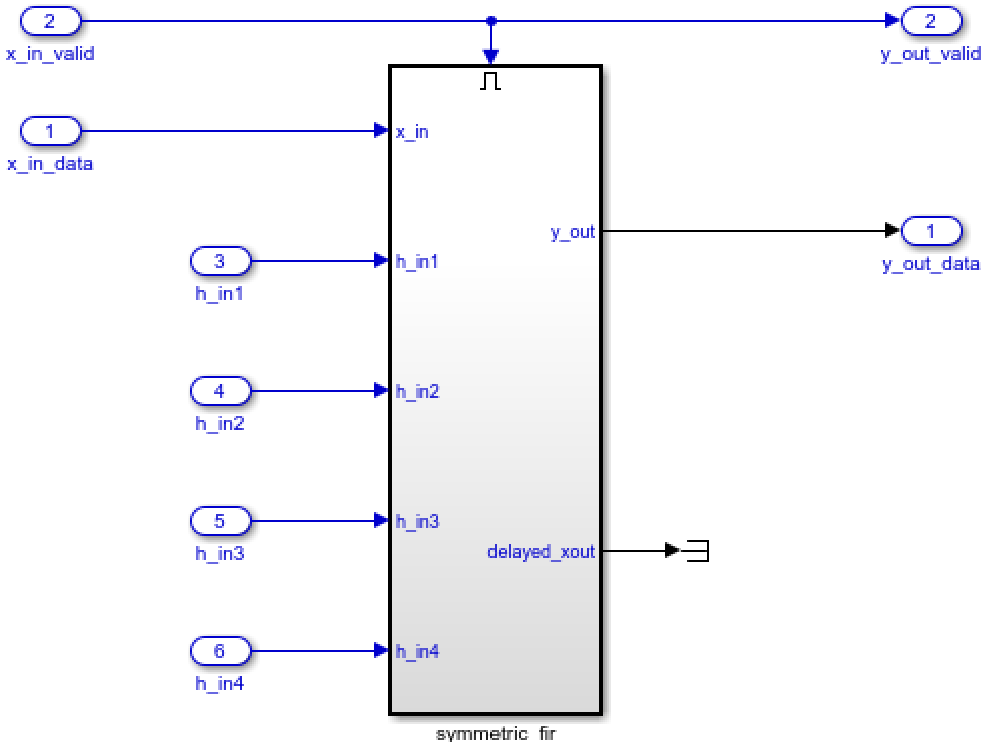
1. Convert the algorithm subsystem into an enabled subsystem.
2. Add an input control port, `Valid_In`, and output control port, `Valid_Out`.
3. Use `Valid_In` to drive the algorithm subsystem's enable port and `Valid_Out`.



In this pattern, both the input streaming channel and output streaming channel follow the simplified streaming protocol.

Open the example model.

```
open_system('hdlcoder_sfir_fixed_stream');
```

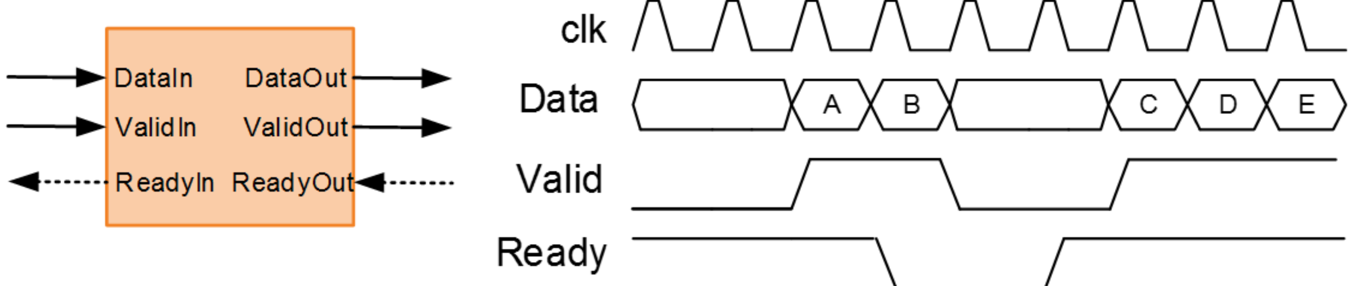


The subsystem DUT is the hardware subsystem targeting the FPGA fabric. Inside this subsystem, the symmetric_fir subsystem represents the filter algorithm. The input ports, x_in_data and x_in_valid, and output ports, y_out_data and y_out_valid, are the data path ports of the filter. The other input ports, such as h_in1, are control ports that tune the filter parameters.

The model follows the modeling pattern for simplified streaming protocol. The symmetric_fir subsystem is an enabled subsystem. The input control signal, x_in_valid, controls the symmetric_fir subsystem's enable port and also drives the output control signal, y_out_valid.

With AXI4-Stream IP core generation, you can optionally model other streaming control signals. For example, you can model the back pressure signal, **Ready**. The AXI4-Stream interface communicates in master/slave mode, where the master device sends data to the slave device. The **Ready** signal is a back pressure signal from the slave device to master device that indicates whether the slave device can accept new data. As shown in following figure, the **Ready** signal is asserted when the slave device can accept new data. When the slave device can no longer accept new data, it needs to de-assert the **Ready** signal. When the master device sees that the **Ready** signal is deasserted, it stops the data transfer at most one sample later. This one sample allowance is built into the protocol.

Note: This figure illustrates the relationship between the **Data**, **Valid**, and **Ready** signals according to the simplified streaming protocol. When you run the IP Core Generation workflow, the code generator adds a streaming interface module in the HDL IP core that translates the simplified protocol to the full streaming protocol.



You can use the **Ready** signal when you use a FIFO block to collect a frame of incoming streaming data, which is then processed with your algorithm. During data processing, you deassert the **Ready** signal to prevent further incoming data.

Generate HDL IP core with AXI4-Stream Interface

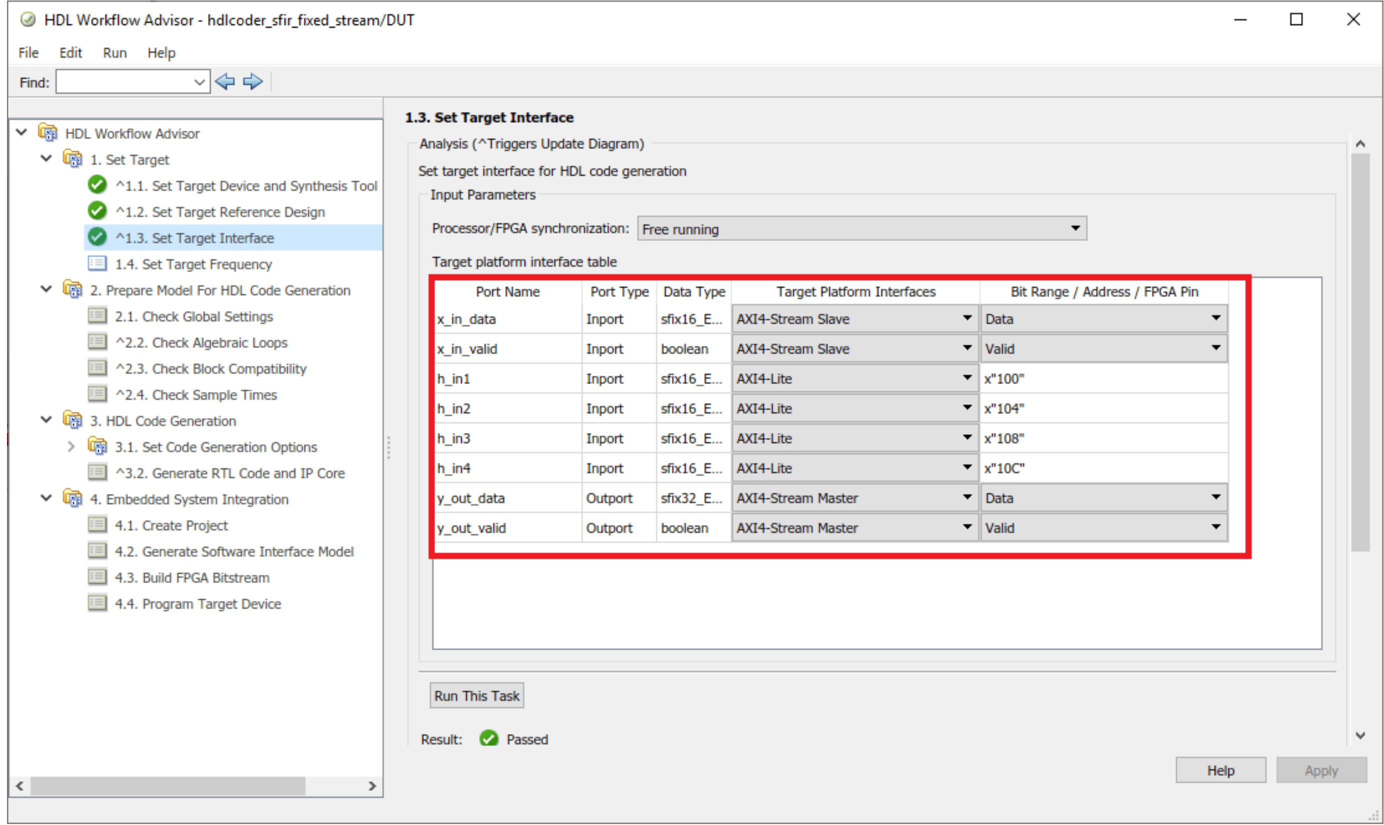
Next, we start the HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy this design on the Zynq hardware. For a more detailed step-by-step guide, you can refer to the "Getting Started with Targeting Xilinx Zynq Platform" on page 39-98 example.

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.
```

2. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_sfir_fixed_stream/DUT`. The target interface settings are already saved in this example model, so the settings in Task 1.1 and 1.2 are automatically loaded. To learn more about saving target interface settings in the model, you can refer to the "Save Target Hardware Settings in Model" on page 39-177 example.

In Task 1.1, **IP Core Generation** is selected for **Target workflow**, and **Zedboard** is selected for **Target platform**. In Task 1.2, **Default system with AXI4-Stream interface** is selected for **Reference Design**, and the **Target platform interface table** is loaded as shown in the following picture. The data path ports, **x_in_data**, **x_in_valid**, **y_out_data**, and **y_out_valid**, are mapped to the AXI4-Stream interfaces, and the control parameter ports, such as **h_in1**, are mapped to the AXI4-Lite interface.



The AXI4-Stream interface communicates in master/slave mode, where the master device sends data to the slave device. Therefore, if a data port is an input port, assign it to an **AXI4-Stream Slave** interface, and if a data port is output port, assign it to an **AXI4-Stream Master** interface.

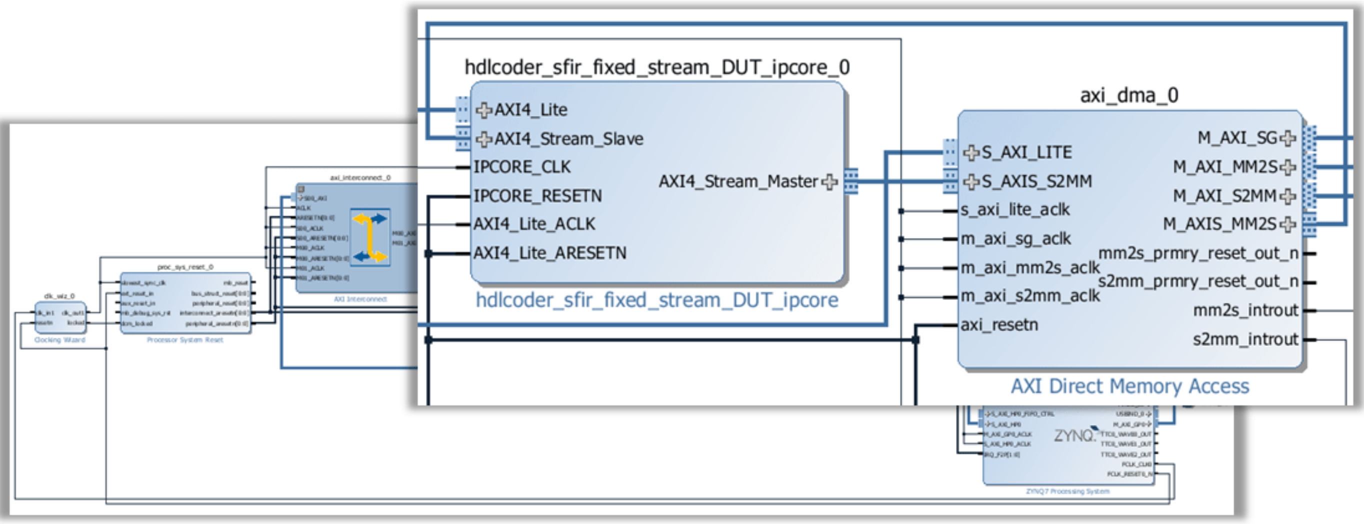
3. Right-click Task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

Integrate IP into AXI4-Stream Compatible Reference Design

Next, in the HDL Workflow Advisor, we run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Zynq hardware.

1. Run Task 4.1, **Create Project**. This task inserts the generated IP core into the **Default system with AXI4-Stream interface** reference design. This reference design contains Xilinx AXI DMA IP to handle the processor to FPGA fabric data streaming. As shown in the first diagram, or in the IP core report, the data is sent from the ARM processing system, through the DMA controller and AXI4-Stream interface, to the generated HDL FIR filter IP core. The output of the filter IP core is then sent back to the processing system.

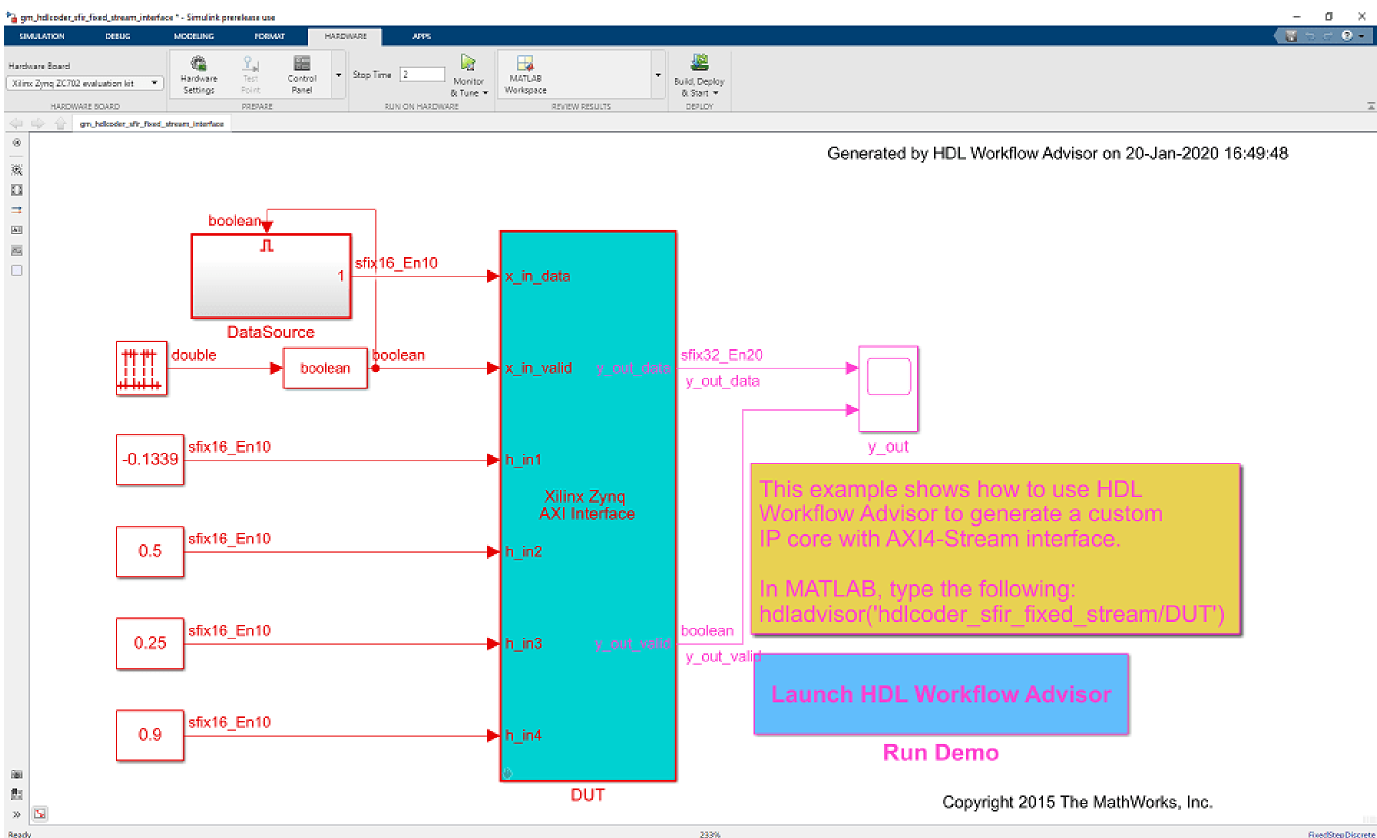
2. Optionally click the link in the Result pane to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram, which includes the generated HDL IP core, AXI DMA controller and the processor.



3. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model, and build and download the FPGA bitstream.

Generate ARM Executable Using AXI4-Stream Driver Block

A software interface model is generated in Task 4.2, **Generate Software Interface Model**, as shown in the following picture.



Although the AXI4-Lite driver is automatically generated in the software interface model, the AXI4-Stream driver block cannot be automatically generated. The reason is that the AXI4-Stream driver block expects to be connected to a vector port on the software side, but the **x_in_data** DUT port is a scalar port.

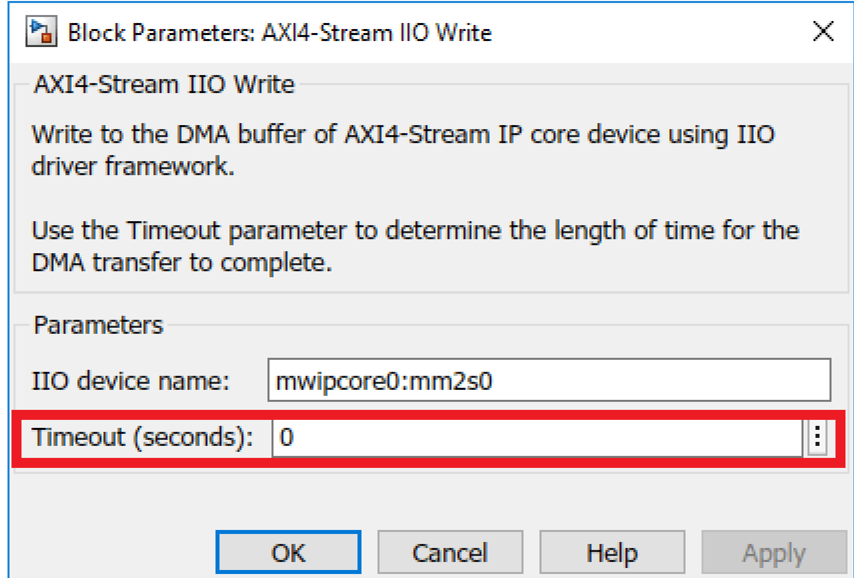
1. Before you generate code from the software interface model:

a. Add the **AXI4-Stream IIO Read** and **AXI4-Stream IIO Write** driver blocks from **Simulink Library Browser -> Embedded Coder Support Package for Xilinx Zynq Platform** library.

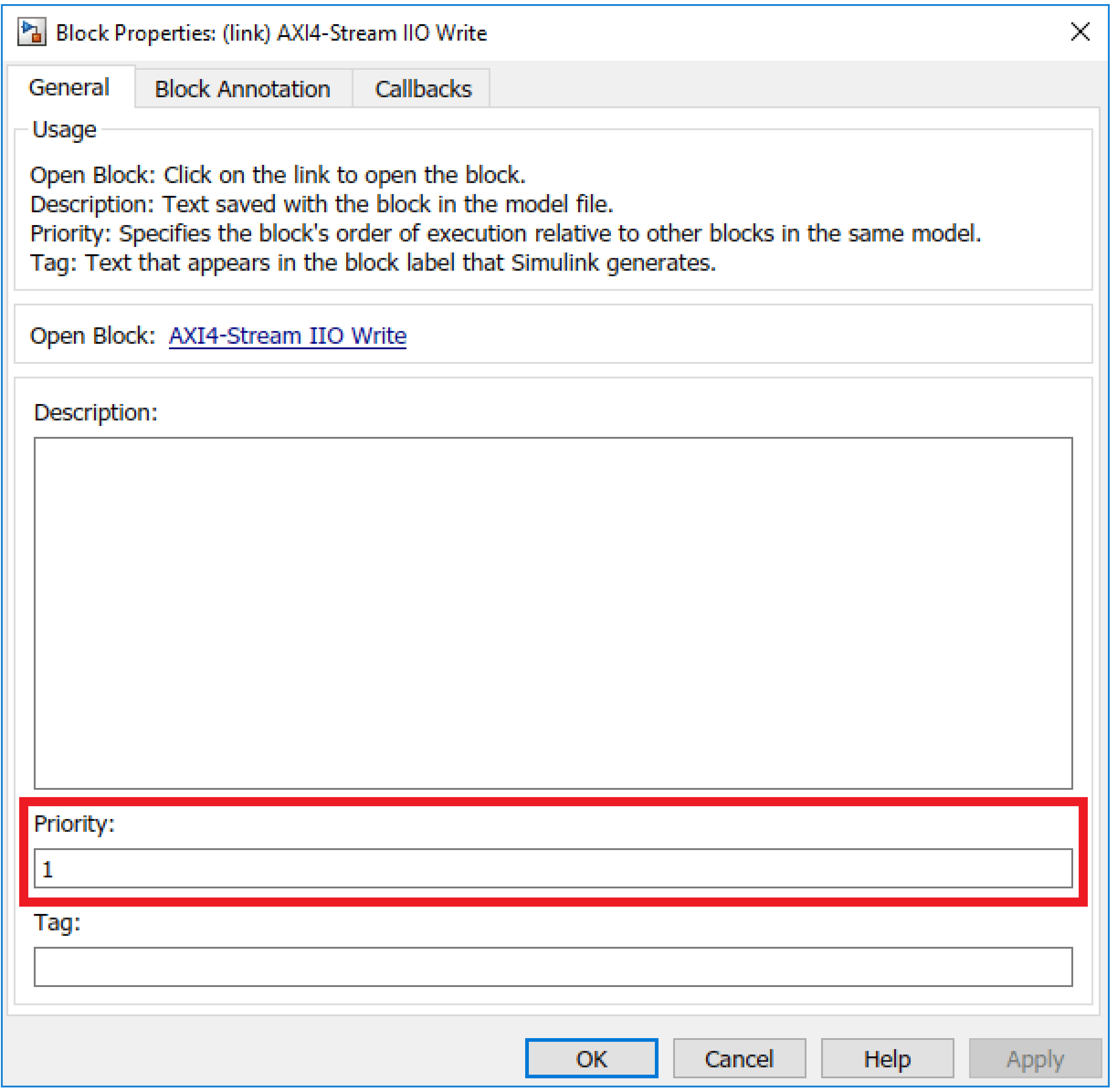
b. Use a vector data source to drive the **x_in_data** port.

c. Connect the **x_in_data** port to the driver block.

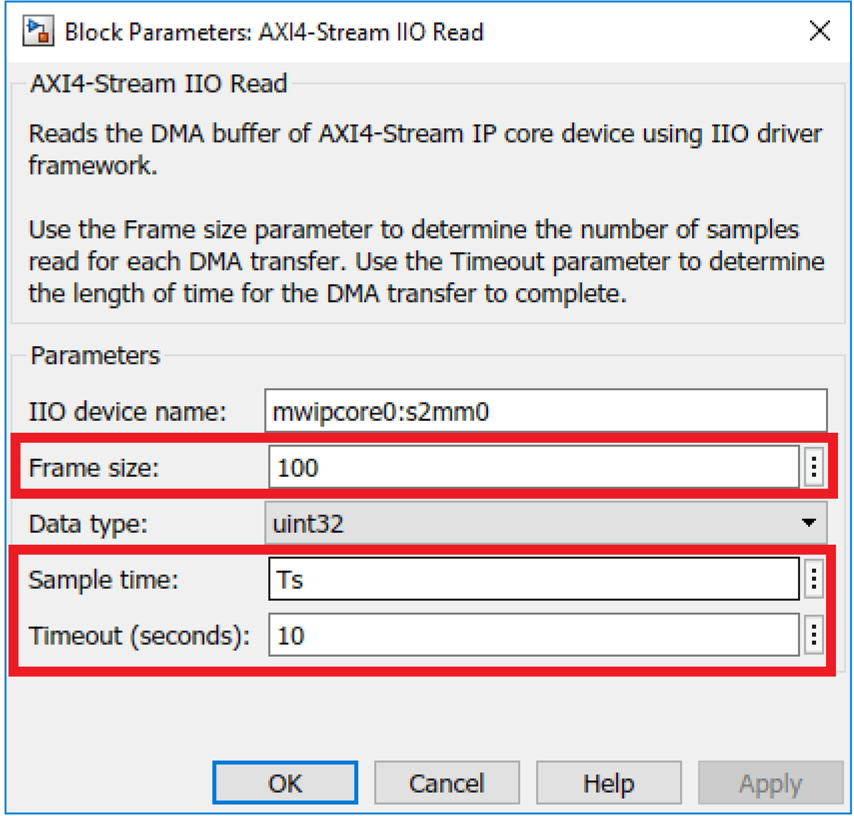
d. Double click on the **AXI4-Stream IIO Write** block and set the Timeout to 0 instead of inf. This is as shown below.



e. Set the priority of the **AXI4-Stream IIO Write** block to 1 to make sure that write happens before read. To set the priority, right click on the block and open properties, set the priority to 1. This is as shown below.

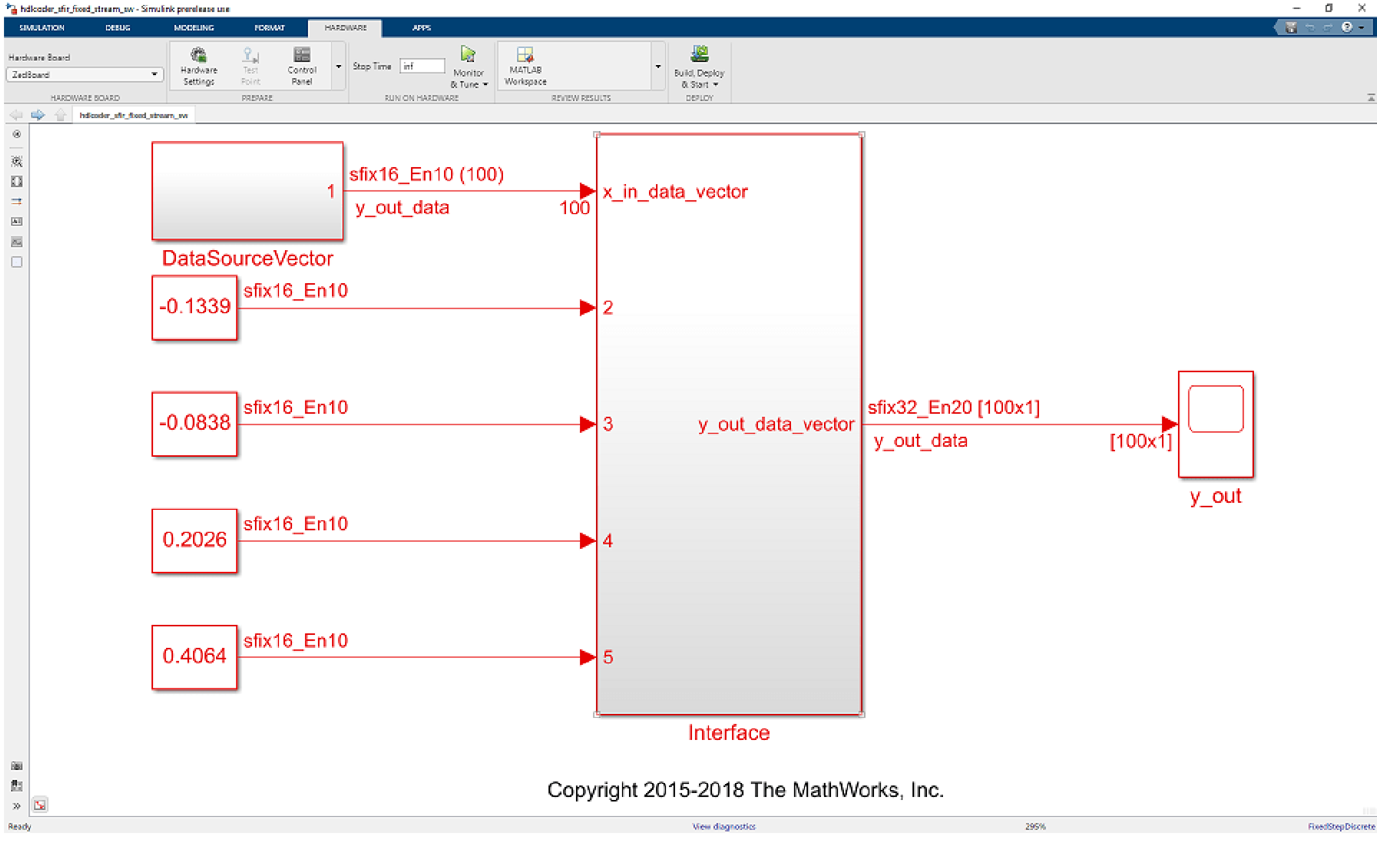


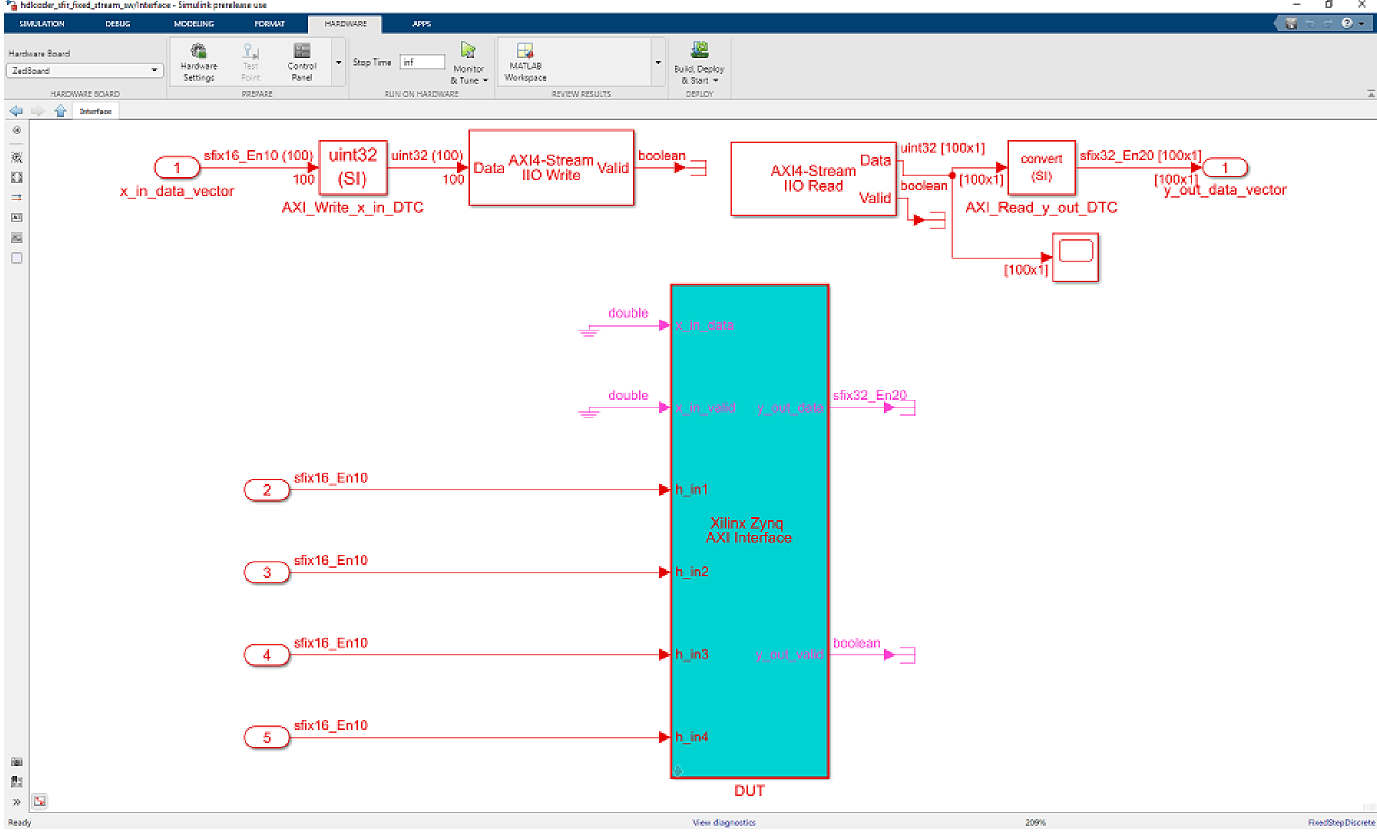
f. Now double click on the **AXI4-Stream IIO Read** block and set the frame size to 100, Sample time to Ts and Timeout to 10. This is as shown below.



g. The priority of the **AXI4-Stream IIO Read** block need not to be set. Setting the priority for write block to 1 alone already ensure that write happens before read.

For this example, the updated software interface model is provided: `hdlcoder_sfir_fixed_stream_sw`. A vector data source with 100 data elements is used in this model, and is connected to the AXI4-Stream DMA driver block. This means that for each processor sample time, the DMA controller will stream 100 32-bit data samples to the HDL IP core via the AXI4-Stream interface, and receive 100 32-bit streaming data samples.


sfix16_En10 (100)
1
y_out_data
100
x_in_data_vector
DataSourceVector
-0.1339
sfix16_En10
2
-0.0838
sfix16_En10
3
y_out_data_vector
sfix32_En20 [100x1]
y_out_data
[100x1]
y_out
0.2026
sfix16_En10
4
0.4064
sfix16_En10
5
Interface
Copyright 2015-2018 The MathWorks, Inc.



2. Configure and build the software interface model for external mode:

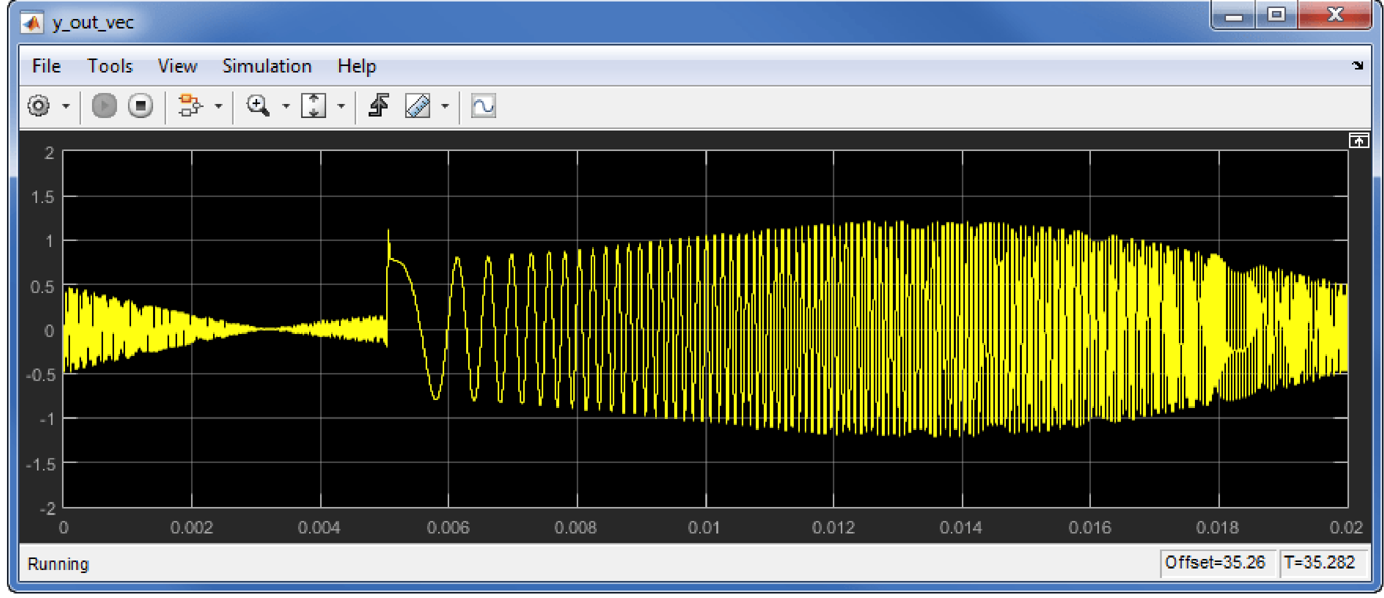
a. In the generated model,click on Hardware pane and go to **Hardware settings** to open **Configuration Parameter** dialog box.

b. Select **Solver** and set "Stop Time" to "inf".

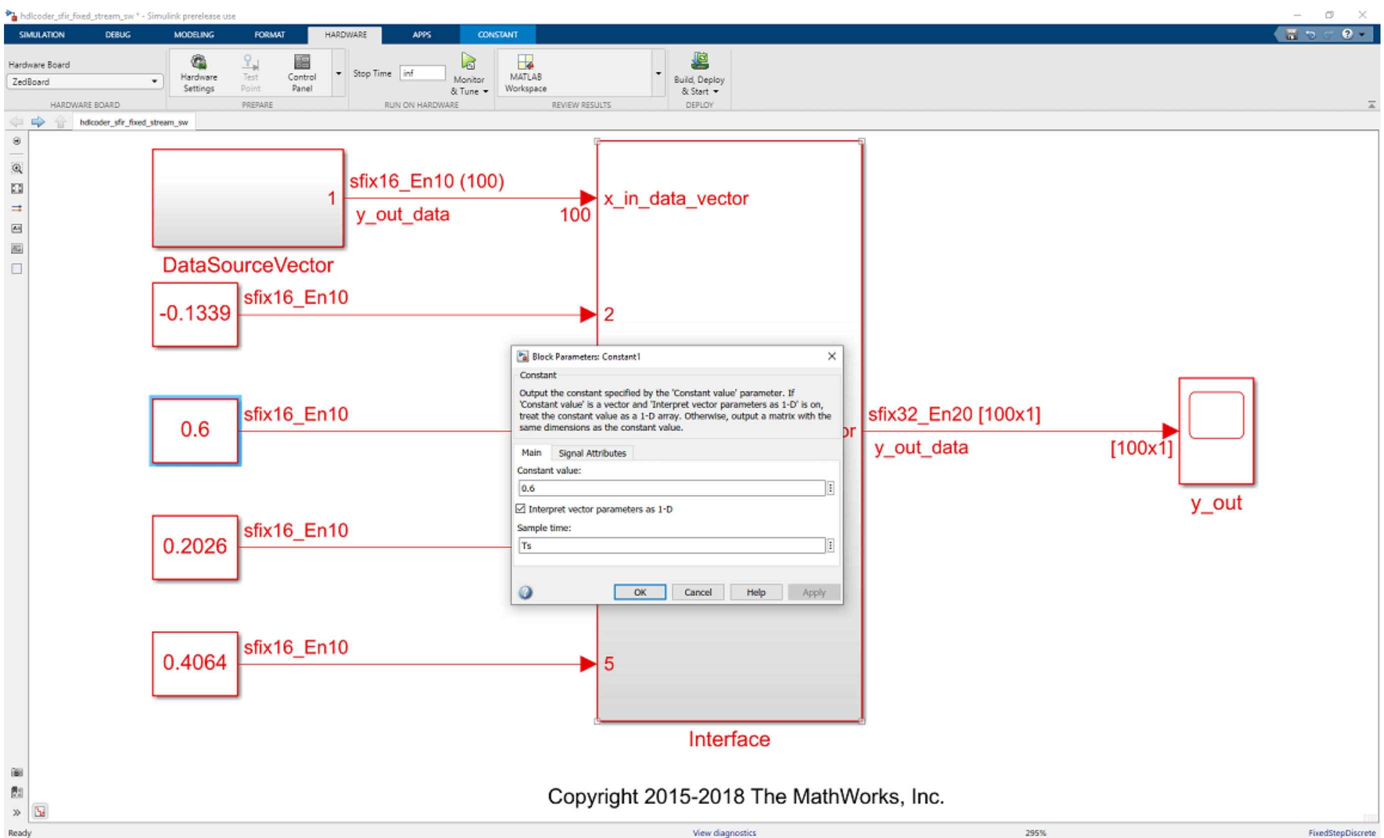
c. From the Hardware pane, click the **Monitor and Tune** button.

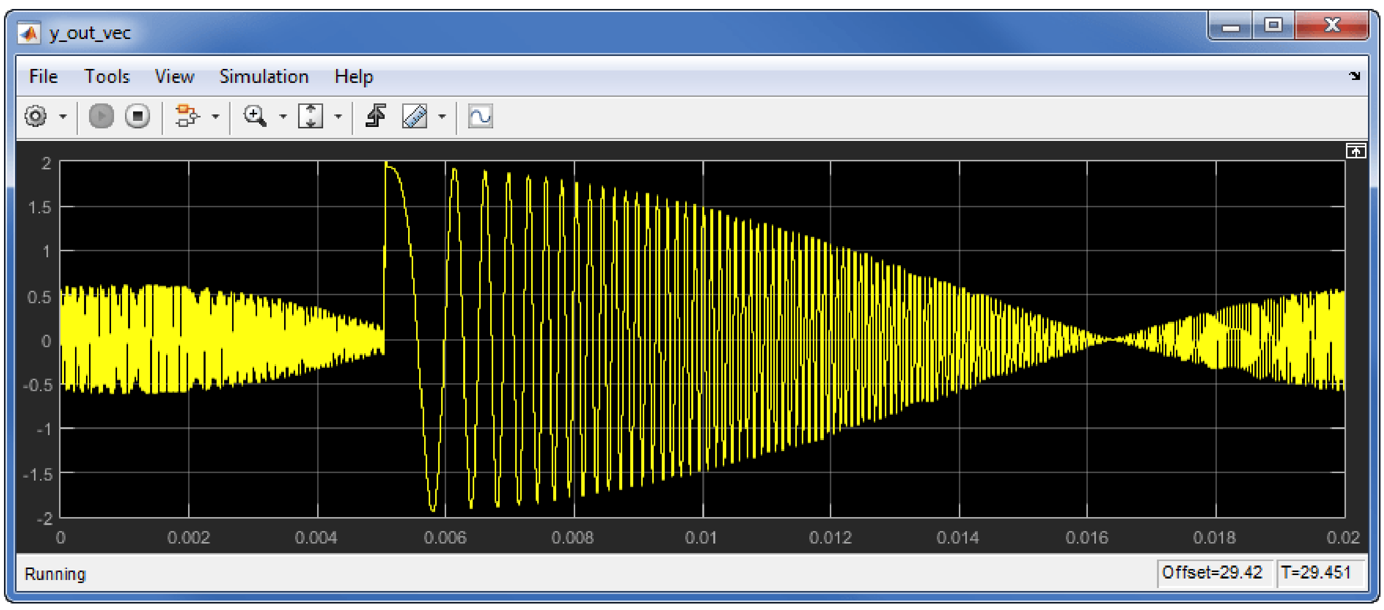
d. Click the **Run** button on the model toolstrip. Embedded Coder builds the model, downloads the ARM executable to the Zedboard hardware, executes it, and connects the model to the executable running on the Zedboard hardware.

3. Now, both the hardware and software parts of the design are running on Zynq hardware. The ARM processor sends the source data to the FPGA IP, through the DMA controller and the AXI4-Stream interface. The ARM processor receives the filter result data from the FPGA IP, and sends the result data to Simulink via external mode. Observe the output of the FIR filter IP core from the Zynq hardware on the Time Scope **y_out**.



4. Tune the FIR filter parameters in the software interface model and observe how the output of the FIR filter changes as you tune the parameters. The parameter values are sent to the Zynq hardware via external mode and the AXI4-Lite interface.




y_out_vec
File Tools View Simulation Help
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0 0.002 0.004 0.006 0.008 0.01 0.012 0.014 0.016 0.018 0.02
Running
Offset=29.42
T=29.451

Deploy Model with AXI4-Stream Video Interface on Zynq Hardware

This example shows how to use the AXI4-Stream Video interface to enable high speed video streaming on the generated HDL IP core.

Requirements

To run this example, you must have the following software and hardware installed and set up:

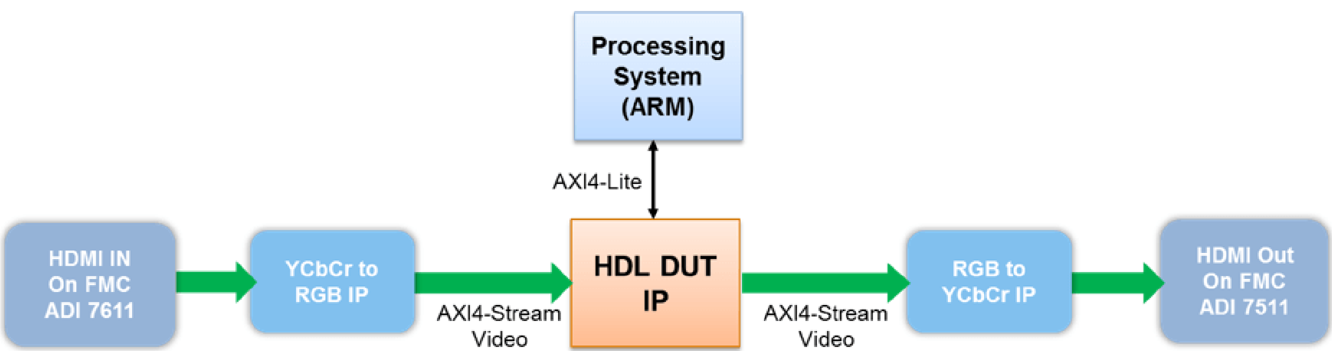
- Xilinx® Vivado® Design Suite, with supported version listed in "HDL Language Support and Supported Third-Party Tools and Hardware"
- ZedBoard™
- FMC HDMI I/O card (FMC-HDMI-CAM or FMC-IMAGEON)

To setup the Zedboard, refer to the *Set up Zynq hardware and tools* section in the example "Getting Started with Targeting Xilinx Zynq Platform" on page 39-98.

Introduction

This example shows how to:

1. Model a video streaming algorithm using the streaming pixel protocol.
2. Generate an HDL IP core with AXI4-Stream Video interface.
3. Integrate the generated IP core into a ZedBoard video reference design with access to HDMI interfaces.
4. Use the ARM® processor to tune the parameters on the FPGA fabric to change the live video output.
5. Create your own custom video reference design.



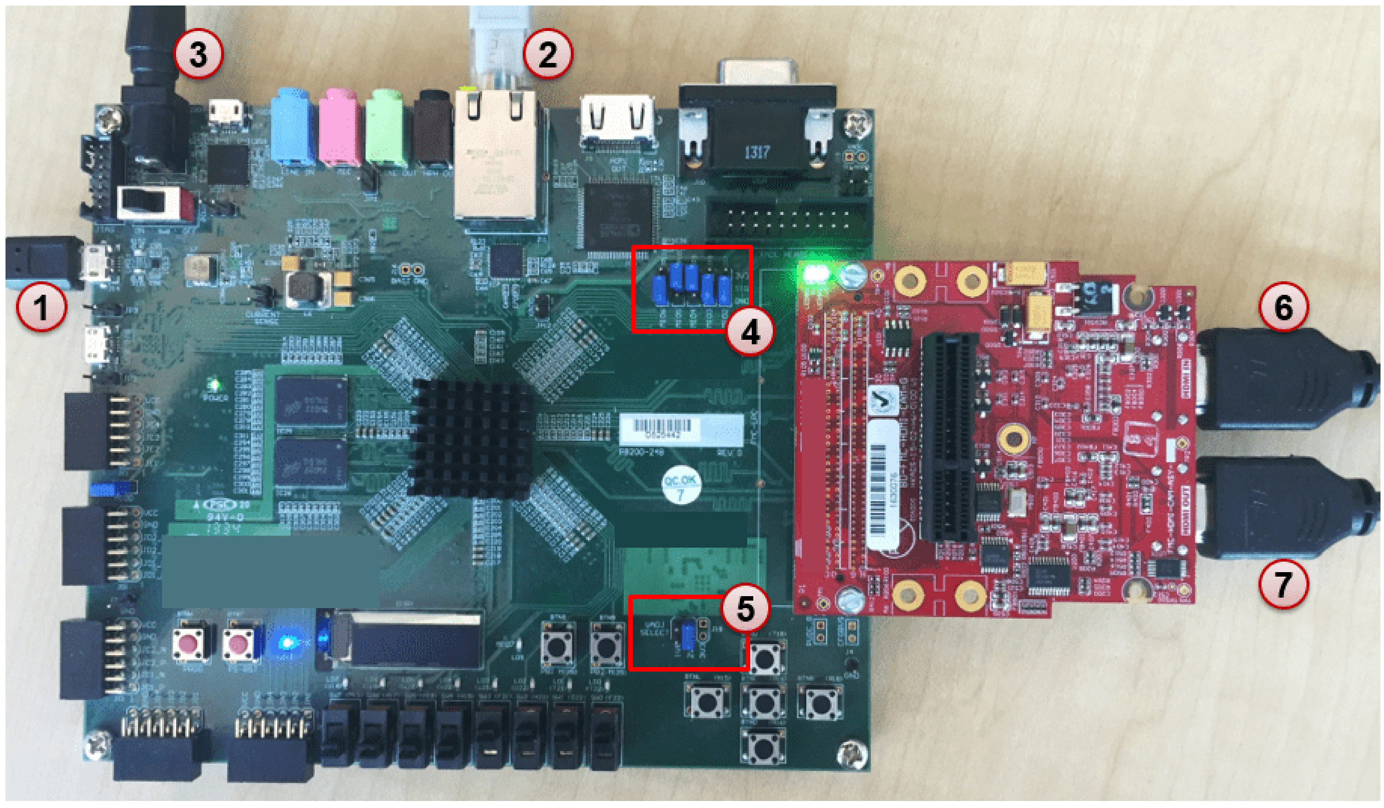
The picture above is a high level architecture diagram that shows how the generated HDL DUT IP core works in a pre-defined video reference design. In this diagram, the **HDL DUT IP** block is the IP core that is generated from the IP core generation workflow. The rest of the diagram represents the pre-defined video reference design, which contains other IPs to handle the HDMI input and output interfaces.

The **HDL DUT IP** processes a video stream coming from the HDMI input IP, generates an output video stream, and sends it to the HDMI output IP. All of these video streams are transferred in AXI4-Stream Video interface.

The **HDL DUT IP** can also include an AXI4-Lite interface for parameter tuning. Compared to the AXI4-Lite interface, the AXI4-Stream Video interface transfers data much faster, making it more suitable for the data path of the video algorithm.

Set Up Zynq Hardware and Tools

1. Set up the ZedBoard and the FMC HDMI I/O card as shown in the figure below. To learn more about the ZedBoard hardware setup, please refer to the board documentation.



2. Connect the USB UART cable, the Ethernet cable and the power cable as shown in the figure above (marker 1 to 3).

3. Make sure the `JP7` to `JP11` jumpers are set as shown in the figure above (marker 4), so you can boot Linux from the SD card. `JP7`: down; `JP8`: down; `JP9`: up; `JP10`: up; `JP11`: down.

4. Make sure the `J18` jumper are set on 2V5 as shown in the figure above (marker 5).

5. Connect HDMI video source to the FMC HDMI I/O card as shown in the figure above (marker 6). The video source must be able to provide 1080p video output, for example, it could be a video camera, smart phone, tablet, or your computer's HDMI output.

6. Connect a monitor to the FMC HDMI I/O card as shown in the figure above (marker 7). The monitor must be able to support 1080p display.

7. If you haven't already, install the HDL Coder™ Support Package for Xilinx FPGA and SoC Devices, and SoC Blockset™ Support Package for Xilinx Devices. To install the support package, go to the MATLAB® Toolstrip and click **Add-Ons > Get Hardware Support Packages**.

8. Make sure you are using the SD card image provided by the HDL Coder Support Package for Xilinx FPGA and SoC Devices. If you need to update your SD card image, see the Hardware Setup section of “Install Support for Xilinx Zynq Platform” (Embedded Coder).

9. Set up the Zynq® hardware connection by entering the following command in the MATLAB command window:

```
h = zynq
```

The `zynq` function logs in to the hardware via COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

10. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
  hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.b
```

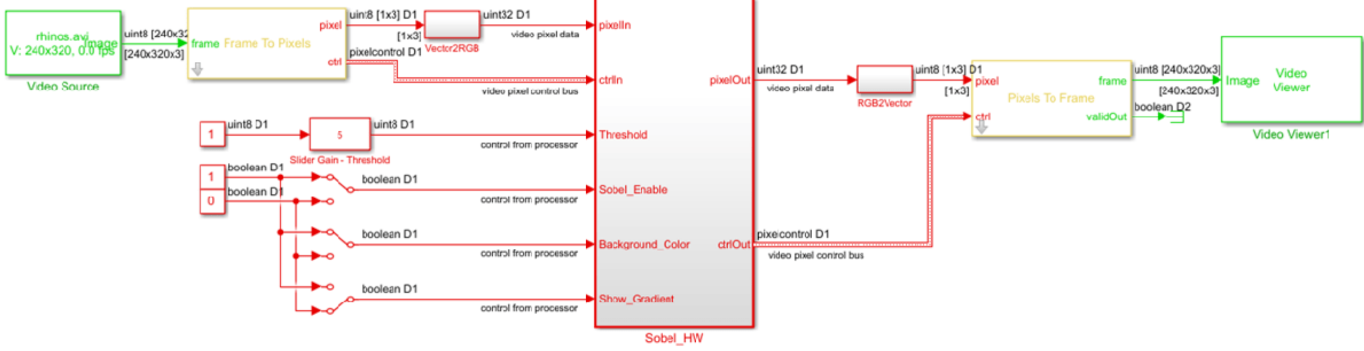
Model Video Streaming Algorithm Using the Streaming Pixel Protocol

To deploy a simple Sobel edge detection algorithm on Zynq hardware, the first step is to determine which part of the design to be run on FPGA, and which part of the design to be run on the ARM processor. This example implements the edge detector on FPGA and processes the incoming video stream in AXI4-Stream Video protocol. The ARM processor tunes the parameters on FPGA to change the live video output.

In the example model, the DUT subsystem, `Sobel_HW`, uses a edge detector block to implement the Sobel edge detection algorithm. The video data and control signals are modeled in the video streaming pixel protocol, which is used by all the blocks in Vision HDL Toolbox™. `pixelIn` and `pixelOut` are data ports for video streams. `ctrlIn` and `ctrlOut` are control ports for video streams. They are modeled using a bus data type (`Pixel Control Bus`) which contains following signals: `hStart`, `hEnd`, `vStart`, `vEnd`, `valid`.

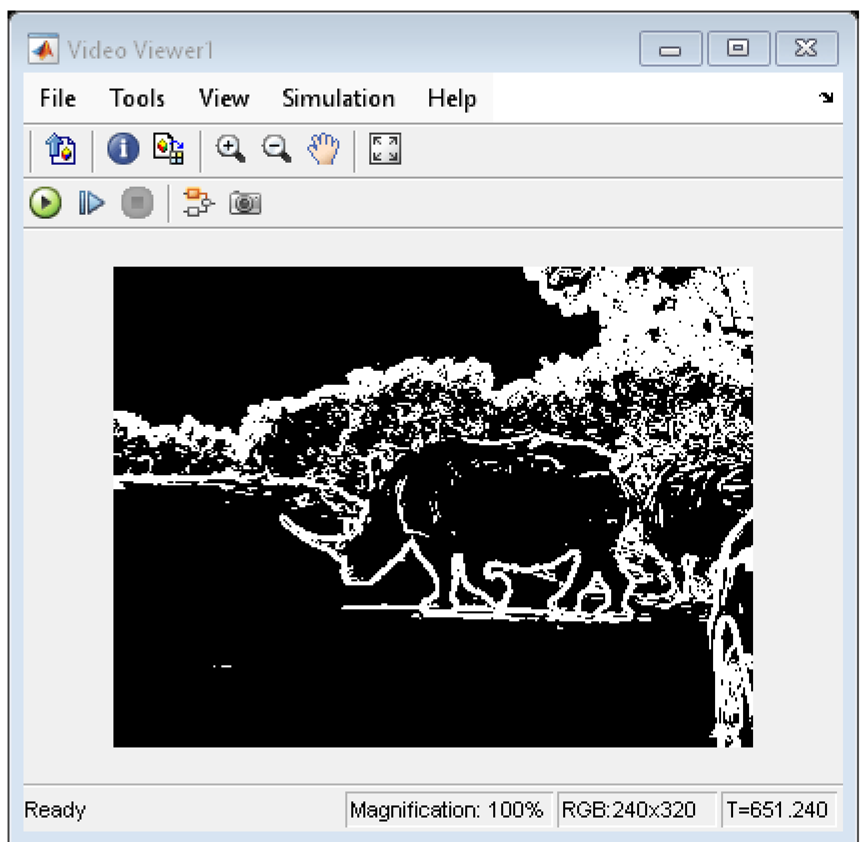
Four input ports, `Threshold`, `Sobel_Enable`, `Background_Color` and `Show_Gradient`, are control ports to adjust the parameters the Sobel edge detection algorithms. You can use the Slider Gain or Manual Switch block to adjust the input values of these ports. After mapping these ports to AXI4-Lite interface, the ARM processor can control the generated IP core by writing to the generated AXI interface accessible registers.

```
modelname = 'hdlcoder_sobel_video_stream';
open_system(modelname);
```



Copyright 2016-2023 The MathWorks, Inc.

```
sim(modelname);
```



Generate HDL IP Core with AXI4-Stream Video Interface

Since R2023b

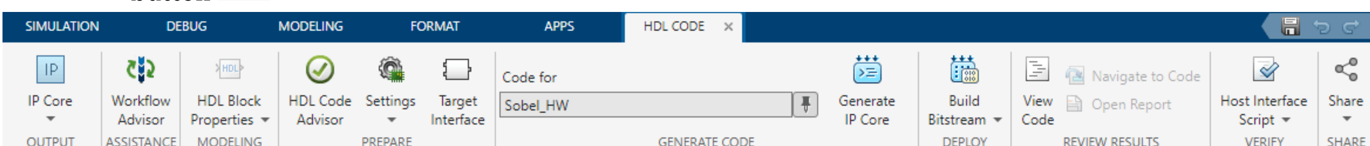
Next, configure your model for IP core generation, configure your design and target interface, and generate the IP core. This example uses the **HDL Code** tab in the Simulink® Toolstrip to generate an IP core. To generate an IP core using the HDL Workflow Advisor, see “Comparison of IP Core

Generation Techniques” on page 39-27. The generated IP core can then be deployed to the Zynq hardware and connected to the embedded processor. For a more detailed step-by-step guide, see “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98. For an overview of how to generate an IP core for a specific hardware platform, see “Targeting FPGA & SoC Hardware Overview” on page 39-3.

Prepare Model for IP Core Generation

Prepare your model by using the configuration parameters, configure your design by using the IP Core editor, and generate the IP core by using the **HDL Code** tab of the Simulink Toolstrip. In this example, the target interface settings are already applied to the model. To learn more about saving target interface settings in the model, see “Save Target Hardware Settings in Model” on page 39-177.

1. In the **Apps** tab, click **HDL Coder**. In the **HDL Code** tab, in the **Output** section, ensure the drop-down button is set to **IP Core**.
2. Select the `Sobel_HW` subsystem, which is the device under test (DUT) for this example. Make sure that **Code for** is set to this subsystem. To remember the selection, you can click the pin button



3. Click **Settings** to open the **HDL Code Generation > Target** pane of the Configuration Parameters dialog box.
4. Set the **Target Platform** parameter to `ZedBoard`. If this option does not appear, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Xilinx FPGA and SoC Devices and follow the instructions to complete the installation. Ensure the **Synthesis Tool** is set to `Xilinx Vivado`.
5. Ensure that the **Reference Design** parameter is set to `Default video system (requires HDMI FMC module)`.
6. Click **OK** to save your updated settings.

Configure Design and Target Interface

Configure your design to map to the target hardware by mapping the DUT ports to IP core target hardware and setting DUT-level IP core options. In this example, the AXI4-Stream Video interface communicates in master/slave mode, where the master device sends data to the slave device. Therefore, input data ports are mapped to an `AXI4-Stream Video Slave` interface, and output data ports are mapped to an `AXI4-Stream Video Master` interface.

1. In Simulink, in the **HDL Code** tab, click **Target Interface** to open the IP Core editor.
2. Select the **Interface Mapping** tab to map each DUT port to one of the IP core target interfaces.

   If no mapping table appears, click the Reload IP core settings button to compile the model and repopulate the DUT ports and their data types.
3. Map the `pixelIn`, `ctrlIn`, `pixelOut`, and `ctrlOut` ports to the AXI4-Stream video interfaces. Map the control parameter ports, such as the `Threshold` port, to the AXI4-Lite interface. Ensure

the DUT ports are mapped to the interfaces specified in the image below.

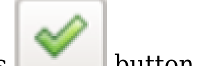
IP Core - hdlcoder_sobel_video_stream/Sobel_HW

General | Clock Settings | Interface Settings | Interface Mapping

☐ Enable HDL DUT output port generation for test points

Source	Port Type	Data Type	Interface	Interface Mapping
pixelIn	Inport	uint32	AXI4-Stream Video Slave	Pixel Data
ctrlIn	Inport	bus	AXI4-Stream Video Slave	Pixel Control Bus
Threshold	Inport	uint8	AXI4-Lite	x"100" Options
Sobel_Enable	Inport	boolean	AXI4-Lite	x"104" Options
Background_Color	Inport	boolean	AXI4-Lite	x"108" Options
Show_Gradient	Inport	boolean	AXI4-Lite	x"10C" Options
pixelOut	Outport	uint32	AXI4-Stream Video Master	Pixel Data
ctrlOut	Outport	bus	AXI4-Stream Video Master	Pixel Control Bus

4

Validate your settings by clicking the Validate IP core settings  button.

In the IP Core editor, you can optionally adjust the DUT-level IP core settings for your target hardware by:

- Using the **General** tab to configure top-level settings, such as the name of the IP core and whether to generate an IP core report.
- Using the **Clock Settings** tab to configure clock-related settings.
- Using the **Interface Settings** tab to configure interface-related settings, such as the register interface and FPGA data capture properties.

Generate IP Core

Next, generate the IP core. In the Simulink Toolstrip, in the **HDL Code** tab, click **Generate IP Core**.

Generating an IP core also generates the code generation report. In the Code Generation Report window, in the left pane, click the **IP Core Generation Report**. The report describes the behavior and contents of the generated custom IP core, such as the register address mapping for the IP core.

Integrate IP into AXI4-Stream Video-Compatible Reference Design

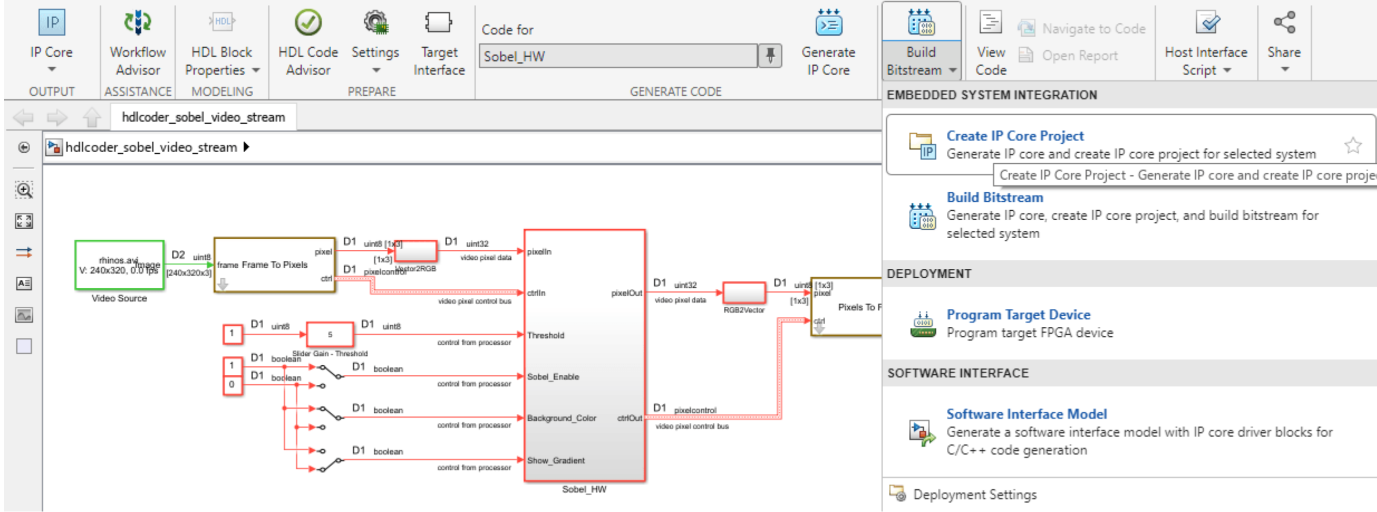
Since R2023b

Next, insert your generated IP core into a embedded system reference design by creating a project, generating an FPGA bitstream, and downloading the bitstream to the Zynq hardware.

The reference design is a predefined Xilinx Vivado project that contains all the elements the Xilinx software needs to deploy your design to the Zynq platform, except for the custom IP core and embedded software. This example uses the **HDL Code** tab in the Simulink Toolstrip to deploy and verify an IP core. For more information on how to deploy and validate an IP core using the HDL Workflow Advisor, see “Comparison of IP Core Deployment and Verification Techniques” on page 39-30.

Create IP Core Project

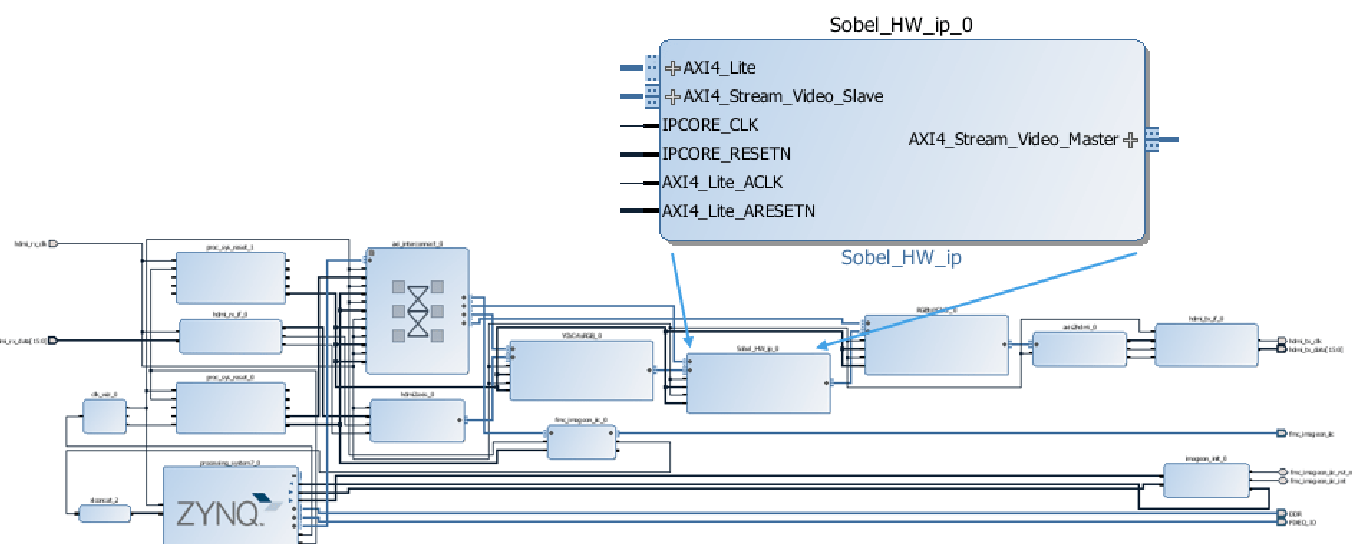
Integrate your generated IP core into the Xilinx platform by creating a Vivado project that organizes and maintains the files associated with the IP core. To create a Vivado project in the Simulink Toolstrip, in the **HDL Code** tab**,** select **Build Bitstream > Create IP Core Project**. HDL Coder generates an IP integrator embedded design and displays a link to it in the Diagnostic Viewer.



Click the link in the Simulink Diagnostic Viewer to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram, which includes the generated HDL IP core, other video pipelining IPs, and the Zynq processor. As shown in this image, the reference design contains the IPs to handle the:

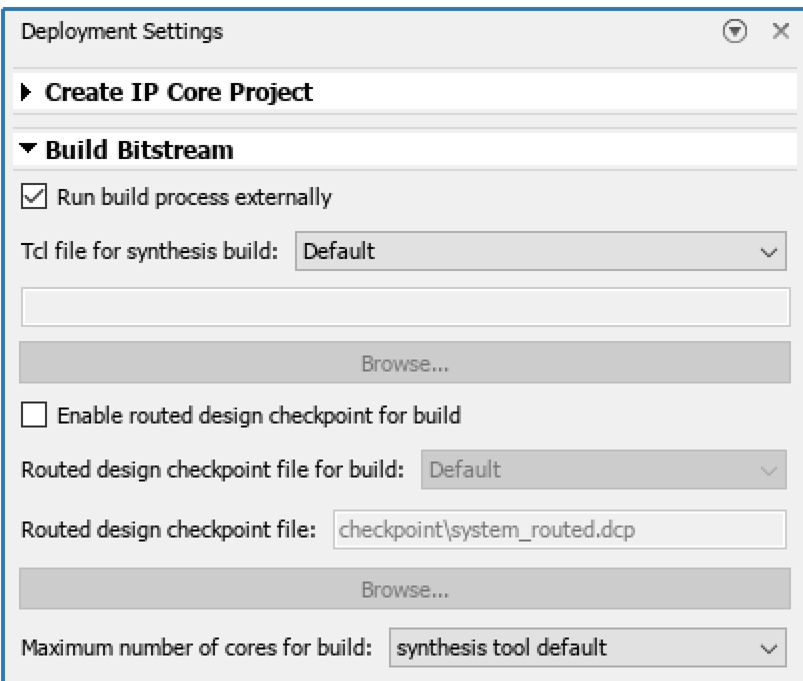
- HDMI input and output interfaces
- Color space conversion from YCbCr to RGB

The generated project is a complete Zynq design, including the generated DUT algorithm IP, and the reference design.



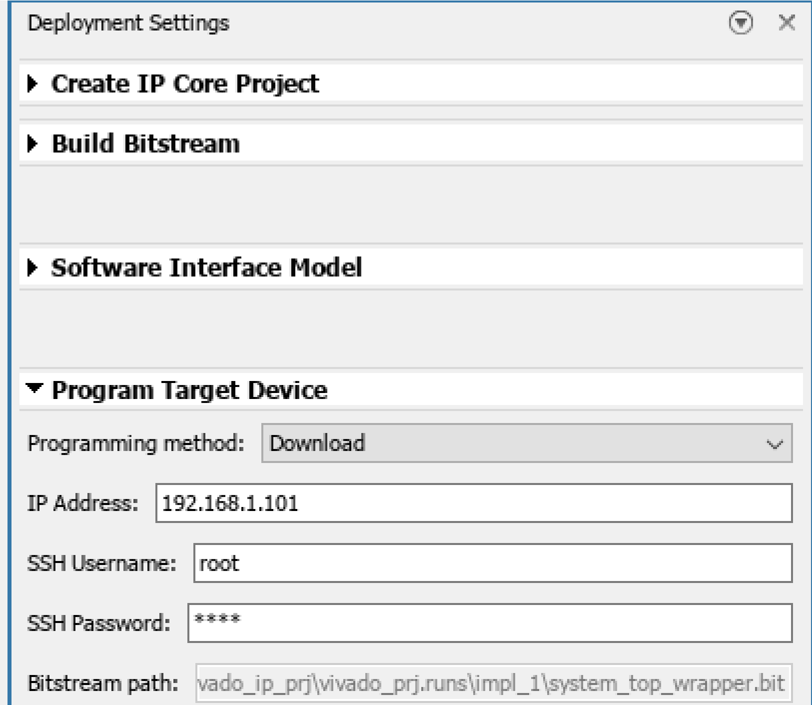
Configure Deployment Settings

Next, configure the build bitstream settings. In the Simulink Toolstrip, in the **HDL Code** tab**,** select **Build Bitstream > Deployment Settings**. In the Deployment Settings window, under the **Build Bitstream** section, select **Run build process externally** to run the Xilinx synthesis tool in a separate window than the current MATLAB session.



Then, under the **Program Target Device** section:

- Set **Programming method** to `Download` to download the FPGA bitstream onto the SD card on your target Zynq board and automatically reload the design when the board power cycles.
- Set **IP Address** to your target board IP address.
- Set **SSH username** and **SSH Password** to your target board settings.



Generate Bitstream and Program Target Device

To generate the bitstream file, in the Simulink Toolstrip, in the **HDL Code** tab, click **Build Bitstream** and wait until the synthesis tool runs in the external window.

To download the bitstream, in the **HDL Code** tab, select **Build Bitstream > Program Target Device**.

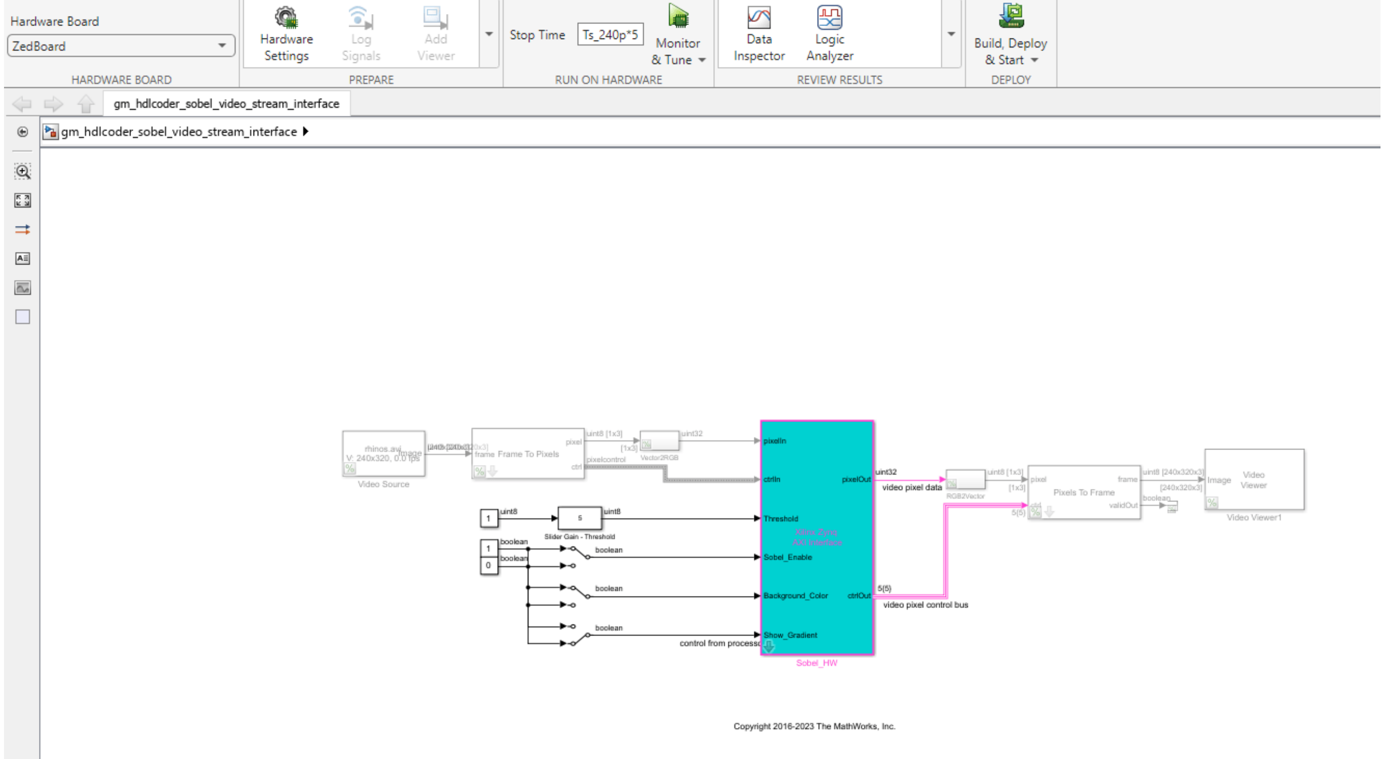
Deploy to Processor Using Software Interface Model

To target a portion of your design for the ARM processor, generate a software interface model. The software interface model contains the part of your design that runs in software. It includes all the blocks outside of the HDL subsystem, and replaces the HDL subsystem with AXI driver blocks. If you have an Embedded Coder license, you can automatically generate embedded code from the software interface model, build it, and run the executable on Linux on the ARM processor. The generated embedded software generates AXI driver code from the AXI driver blocks and uses the code to control the HDL Coder generated IP core. You can generate the software interface model at any stage of the IP core generation and IP core integration process.

To generate the software interface model, in the Simulink Toolstrip, in the **HDL Code** tab, select **Build Bitstream > Software Interface Model.** The Simulink Diagnostic Viewer displays a link to the generated software interface model.

```
Software Interface Model will be generated based on last stored Interface Mapping.

### Generate Simulink software interface model

Generating new Zynq Software Interface model: gm_hdlcoder_sobel_video_stream_interface
Zynq Software Interface model generation complete.
No driver block was generated for port(s) "pixelIn, ctrlIn, pixelOut, ctrlOut" mapped to interface "AXI4-Stream Video" in the software interface model.
```

Before you generate code from the software interface model, comment out the Video Source and Video Viewer in the generated model, as shown in this picture. These blocks do not need to run on the ARM processor. Because the ARM processor uses an AXI4-Lite interface to control the FPGA fabric, the actual video source and display interface run on the FPGA fabric. The video source comes from the HDMI input, and the video output is sent to the monitor connected to the HDMI output.



Run Software Interface Model on ZedBoard Hardware

Next, configure the generated software interface model, generate the embedded C code, and run your model on the ARM processor in the Zynq hardware in external mode.

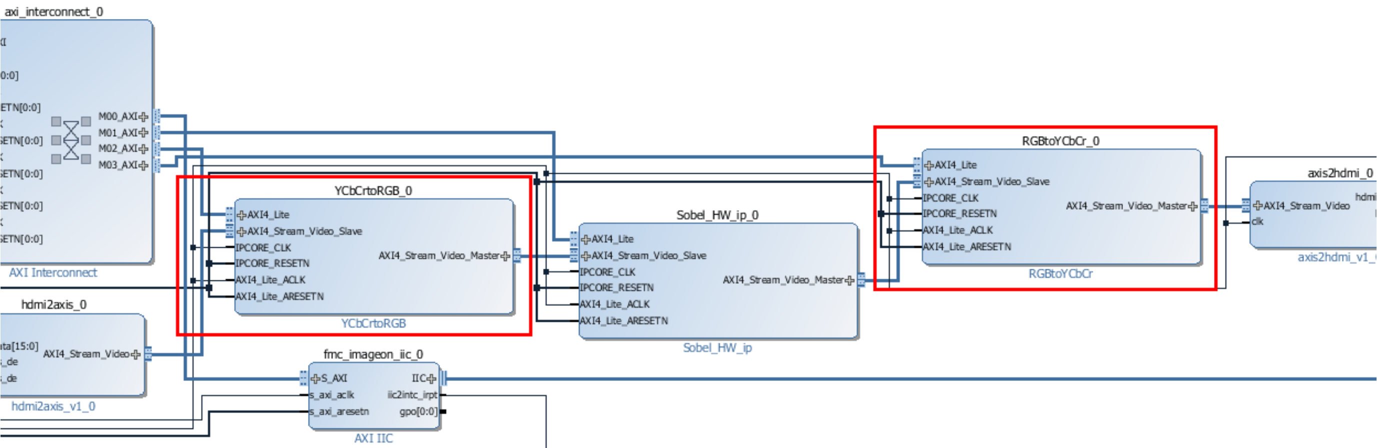
When you prototype and develop an algorithm, it is useful to monitor and tune the algorithm while it runs on hardware. You can use external mode to deploy your algorithm to the ARM processor in the Zynq hardware, and then link the algorithm with the Simulink model on the host computer through an Ethernet connection.

1. In the generated model, in the **Hardware** tab, in the **Prepare** section, click **Hardware settings** to open the Configuration Parameters dialog box.
2. In the **Run On Hardware** section, set **Stop time** to `inf`. Click **OK**.
3. In the **Hardware** tab, **Run On Hardware** section, click **Monitor & Tune**. Embedded Coder builds the model, downloads the ARM executable to the Zynq hardware, executes it, and connects the model to the executable running on the Zynq hardware.
4. Use the input switch to the **Sobel_Enable** port to observe that the live video output switches between the edge detector output and the original video. Use the switch inputs to the **Threshold** or **Background_Color** ports to see the different edge detection effects on the live video output. The model sends the parameter values to the Zynq hardware via external mode and the AXI4-Lite interface.
5. When you are done changing model parameters, click the **Stop** button in the Simulink Toolstrip, then close the system command window.

Customize the Video Reference Design

Suppose that you want to extend the existing `Default video system` reference design to add additional pre-processing or post-processing camera pipelining IPs, or that you want to use a different SoC hardware or video camera interface. You can use the `Default Video System` reference design as a starting point to create your own custom reference design.

For example, the `Default video system` reference design contains two IP cores that do color space conversion from YCbCr to RGB, as shown in this picture. These two IP cores are generated by HDL Coder when you generate an IP Core. You can generate other pre-processing or post-processing camera pipelining IP cores and add them into a custom reference design to extend your video platform.



For more details on creating your own custom reference design, see the “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252 example.

See Also

More About

- “Comparison of IP Core Generation Techniques” on page 39-27
- “Comparison of IP Core Deployment and Verification Techniques” on page 39-30
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9

Perform Matrix Operation Using External Memory

This example shows how to generate an HDL IP core with AXI4 Master interface, perform matrix multiplication in HDL IP core, and write output result to DDR memory.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

- Xilinx® Vivado® Design Suite, with supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.
- Xilinx Zynq® ZC706 Evaluation Kit.
- HDL Coder™ Support Package for Xilinx FPGA and SoC Devices.
- HDL Verifier™ Support Package for Xilinx FPGA Boards.
- This example can also be run on a Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit.

Introduction

In this example, you:

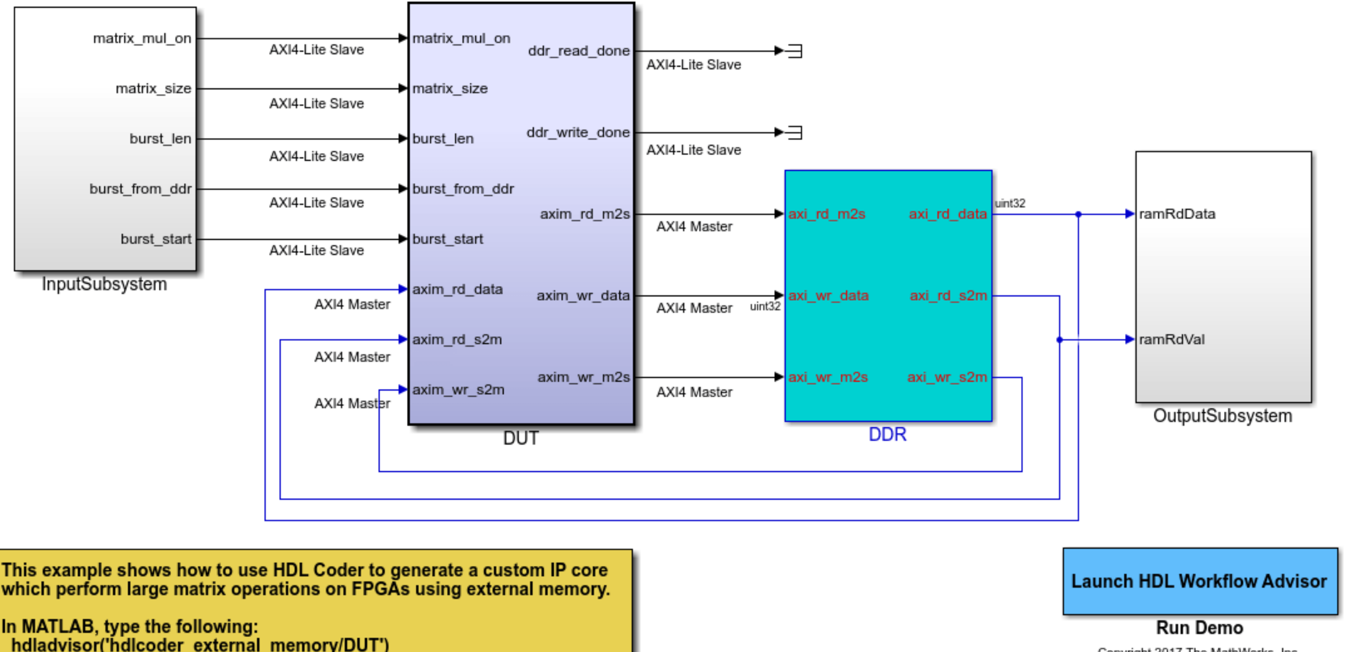
1. Generate an HDL IP core with AXI4 Master interface.
2. Access large matrices from the external DDR3 memory on the Xilinx Zynq ZC706 board using the AXI4 Master interface.
3. Perform matrix vector multiplication in the HDL IP core and write the output result back to the DDR memory using the AXI4 Master interface.

This example can also be run on a Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit, to access the external DDR4 memory.

This example models a matrix vector multiplication algorithm and implements the algorithm on the Xilinx Zynq FPGA board. Large matrices may not map efficiently to Block RAMs on the FPGA fabric. Instead, we can store the matrices in the external DDR memory on the FPGA board. The AXI4 Master interface can access the data by communicating with vendor-provided memory interface IP cores that interface with the DDR memory. This capability enables you to model algorithms that involve large data processing and requires high-throughput DDR access, such as matrix operations, computer vision algorithms, and so on.

The matrix vector multiplication module supports fixed point matrix vector multiplication, with a configurable matrix size ranging from 2 to 4000. The size of the matrix is run-time configurable through AXI4 accessible register.

```
modelname = 'hdlcoder_external_memory';
open_system(modelname);
```



Model Algorithm Using AXI4 Master Protocol

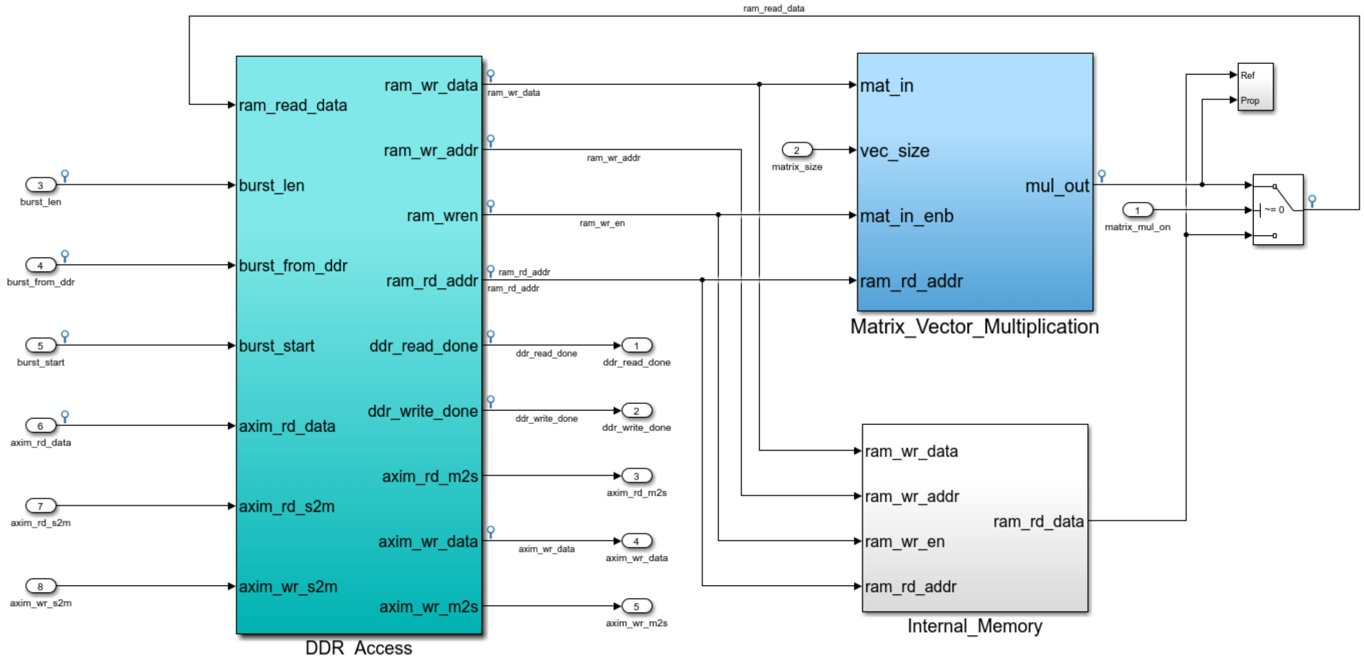
This example model includes the FPGA implementable `DUT` (Design under test) block, the `DDR` functional behavior block and the test environment to drive inputs and verify the expected outputs.

The `DUT` subsystem contains the AXI4 Master read/write controller along with the matrix vector multiplication module. Using the AXI4 Master interface, the `DUT` subsystem reads data from the external DDR memory, feed the data into the `Matrix_Vector_Multiplication` module, and then write the output data to the external DDR memory using AXI4 Master interface. The `DUT` module also has several parameter ports. These ports will be mapped to AXI4-Lite accessible registers, so you can adjust these parameters from MATLAB®, even after you implement the design onto the FPGA.

The `DDR` module represents the external DDR memory in simulation environment. The interface between the `DUT` and `DDR` modules are the simplified AXI4 Master protocol.

One of the parameter port `matrix_mul_on` controls whether to run the `Matrix_Vector_Multiplication` module. When the input to `matrix_mul_on` is true, the `DUT` subsystem performs matrix vector multiplication as describe above. When the input to `matrix_mul_on` is false, the `DUT` subsystem perform a data loop back mode. In this mode, the `DUT` subsystem read data from the external DDR memory, write it into the `Internal_Memory` module, and then write the same data back to the external DDR memory. The data loop back mode is a simple way to verify the functionality of the AXI4 Master external DDR memory access.

```
open_system('hdlcoder_external_memory/DUT');
```



Inside the DUT subsystem, the `DDR_Access` module models the simplified AXI4 Master protocol, and use it to read and writes data on DDR. During the IP Core Generation workflow, HDL Coder will then generate the translator between the simplified AXI4 Master protocol and the actual AXI4 Master protocol in the generated HDL IP core. For more information on the simplified AXI4 Master protocol, see “Model Design for AXI4 Master Interface Generation” on page 40-128.

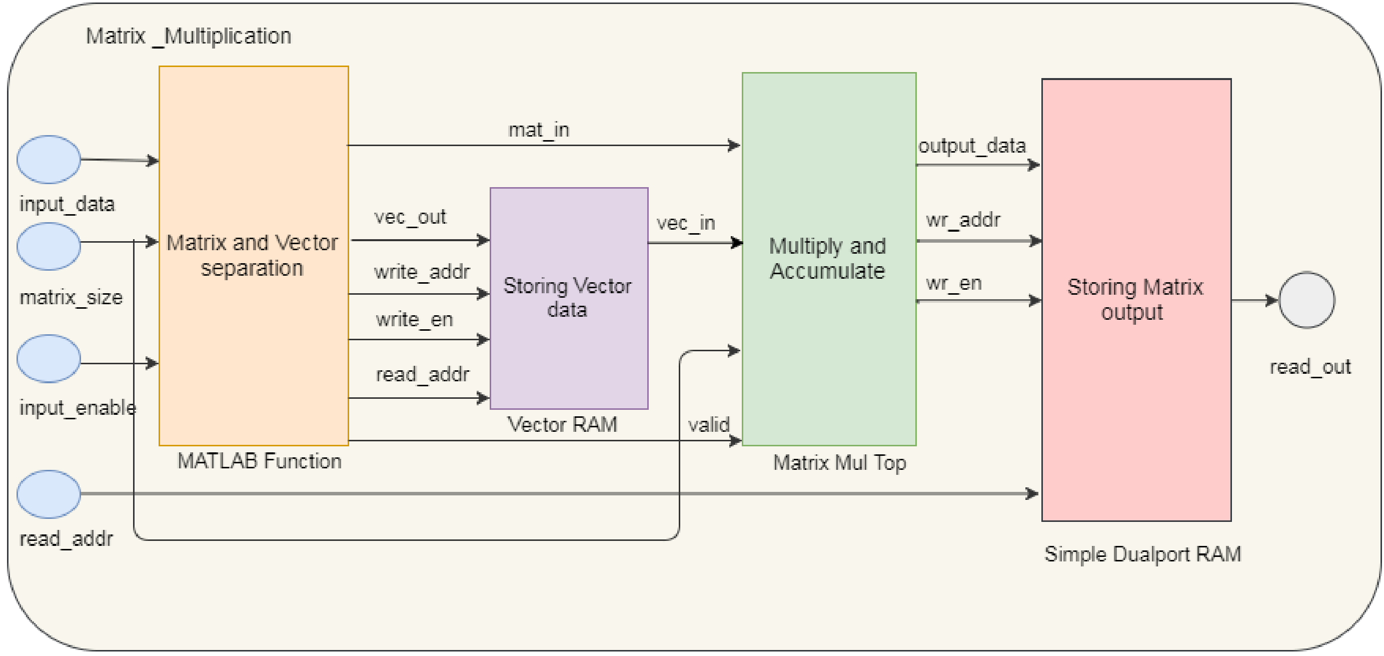
Also inside the DUT subsystem, the `Matrix_Vector_Multiplication` module uses a multiply-add block to implement a streaming dot-product computation for the inner-product of the matrix vector multiplication.

Lets say, A be a matrix of size NxN and B is a vector of size Nx1.

Then, matrix vector multiplication output will be: Z = A * |B|, of size Nx1.

The first N values from the DDR are treated as the Nx1 size vector, followed by NxN size matrix data. First N values (vector data) are stored into a RAM. From N+1 values onwards, data is directly streamed as matrix data. Vector data will be read from the `Vector_RAM` in parallel. Both matrix and vector inputs are fed into the `Matrix_mul_top` subsystem. The first matrix output is available after N clock cycles and will be stored into output RAM. Again, vector RAM read address is reinitialized to 0 and starts reading same vector data corresponding to new matrix stream. This operation is repeated for all the rows of the matrix.

The follow diagram shows the architecture of the `Matrix_Vector_Multiplication` module.



Functional Simulation in Simulink

You can simulate this example model, and verify the simulation result by running following script in MATLAB:

```
hdlcoder_external_memory_simulation;
```

```
PASSED: DDR initialization data matches.
PASSED: Matrix vector multiplication output matches with the expected data
```

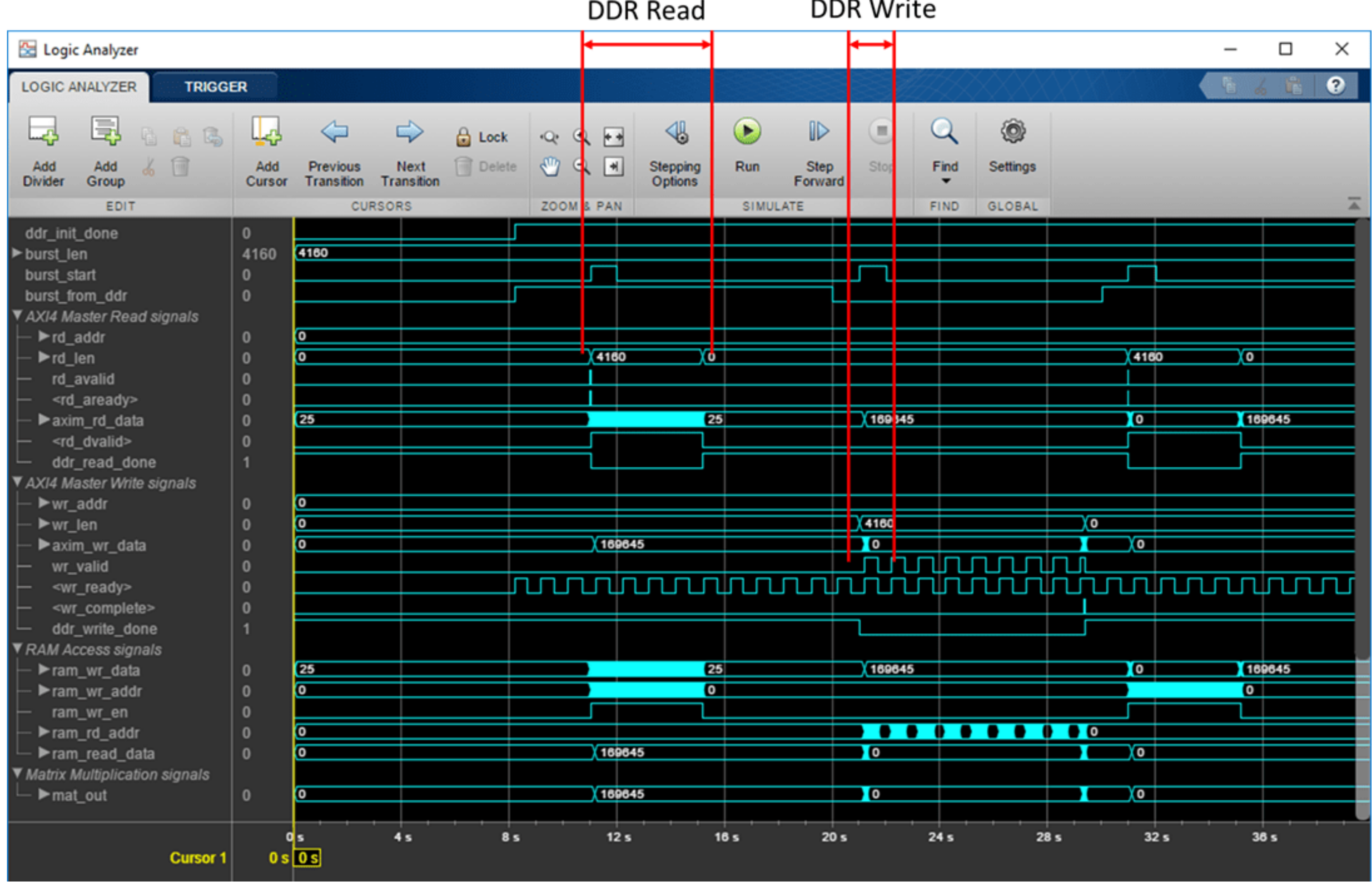
This script first initializes the parameters like `Matrix_Size`. By default the `Matrix_Size` is 64, which means a 64x64 matrix. The default `Matrix_Size` is kept small so the simulation is faster. After the DUT is implemented onto the FPGA board, larger `Matrix_Size` then can be used as the FPGA calculation is much faster. You can also adjust these parameters in the script.

The script then simulates the model, and verifies the result by comparing the logged simulation result with the expected value.

By default, the Matrix_Multiplication_On is true, the script verifies the matrix vector multiplication result.

When the Matrix_Multiplication_On is false, the script verifies the loop back mode, which means the DUT read `Burst_Length` amount of data from DDR, and write the data back to DDR.

If you have a DSP System Toolbox license, you can view the model signals over time using the Logic Analyzer.



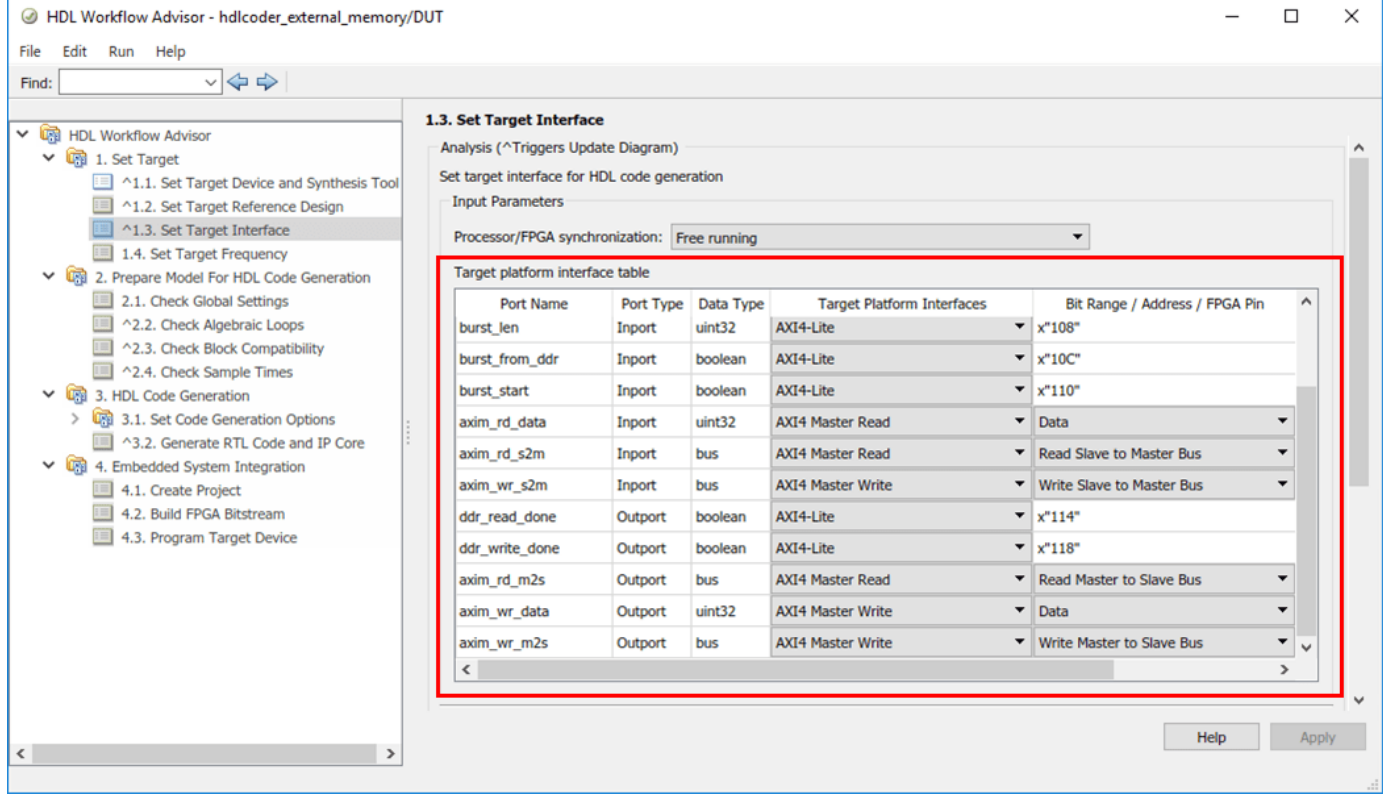
Generate HDL IP core with AXI4 Master Interface

Next, we start the HDL Workflow Advisor and use the IP Core Generation workflow to deploy this design on the Zynq hardware. For a more detailed step-by-step guide, see "Getting Started with Targeting Xilinx Zynq Platform" on page 39-98.

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2018.2\bin\vivado.
```

2. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_external_memory/DUT`. The target interface settings are saved on the model. Notice that **Target workflow** is `IP Core Generation`, **Target platform** is `Xilinx Zynq ZC706 evaluation kit`, **Reference Design** is `Default System with External DDR3 memory access`, and **Target platform interface table** settings are as shown below.



In this example, the input parameter ports like `matrix_mul_on`, `matrix_size`, `burst_len`, `burst_from_ddr` and `burst start` are mapped to the `AXI4-Lite` interface. HDL Coder will generate AXI4 interface accessible registers for these ports. Later, you can use MATLAB to tune these parameters at run-time when the design is running on FPGA board.

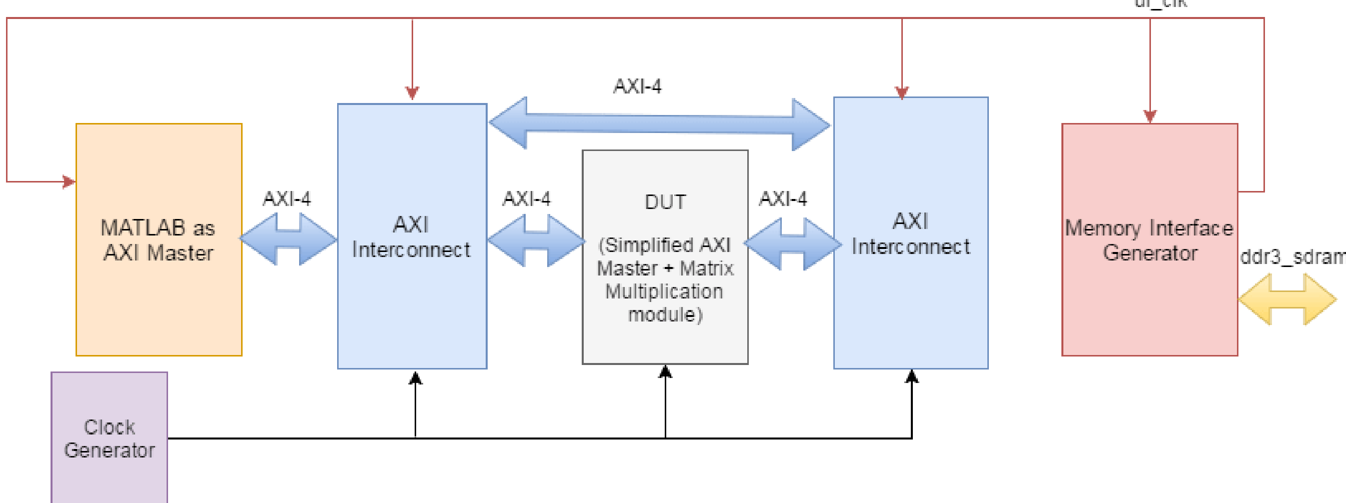
The AXI4 Master interface has separate Read and Write channels. The read channel ports like `axim_rd_data`, `axim_rd_s2m`, `axim_rd_m2s` are mapped to `AXI4 Master Read` interface. The write channel ports like `axim_wr_data`, `axim_wr_s2m`, `axim_wr_m2s` are mapped to `AXI4 Master Write` interface.

3. Right-click Task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

4. Now Right-click Task 4.2 **Build FPGA Bitstream**, and select **Run to Selected Task** to generate the Vivado project, and then build the FPGA bitstream.

During the project creation, the generated DUT IP core is integrated into the `Default System with External DDR3 Memory Access` reference design. This reference design comprises of a Xilinx Memory Interface Generator IP to communicate with the on-board external DDR3 memory on ZC706 platform. The AXI Manager IP is also added to enable MATLAB to control the DUT IP, and to initialize and verify the DDR memory content.

You can click the link in the result window in Task 4.1 "Create Project" to view the generate Vivado project. If you open the Vivado block design, the generated reference design project looks similar to this architecture diagram.



Run FPGA Implementation on Xilinx Zynq ZC706 Evaluation Kit

After the FPGA bitstream is generated, you can run Task 4.3 **Program Target Device** to program the FPGA board through JTAG cable.

You can then run the FPGA implementation, and verify the hardware result by running following script in MATLAB:

```
hdlcoder_external_memory_hw_run
```

This script first initializes the `Matrix_Size` to 500, which means a 500x500 matrix. You can adjust the `Matrix_Size` up to 4000.

The AXI4 Master Read and Write channel base addresses are then configured. These addresses defines the base address that DUT reads from, and writes to external DDR memory. In this script, the DUT is reading from base address '80000000', and write to base address '81000000'.

Then the AXI Manager feature is used to initialize the external DDR3 memory with input vector and matrix data, and also clear the output DDR memory location.

Then the DUT calculation is started by controlling the AXI4-Lite accessible registers. The DUT IP core first read input data from the DDR memory, perform the matrix vector multiplication, and then write the result back to the DDR memory.

Finally, the output result is read back to MATLAB, and compared with the expected value. In this way, the hardware results are verified in MATLAB.

```
>> hdlcoder_external_memory_hw_run
Initializing external DDR3 memory (data size 250500) ...
Starting DUT IP core processing ...
Verifying result ...
PASSED: Matrix vector multiplication output matches with the expected data.
```

Accessing External DDR4 Memory on Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit

1. Use the same model `hdlcoder_external_memory` to access external DDR4 memory on ZCU102 using HDL Coder IP core generation workflow.

2. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_external_memory/DUT`. In Task 1.1 Set **Target platform** as `Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit` and in Task 1.2 set **Reference Design** as `Default System with External DDR4 Memory Access`.

3. Now Right-click Task 4.2 **Build FPGA Bitstream**, and select **Run to Selected Task** to generate the Vivado project, and then build the FPGA bitstream.

4. You can run Task 4.3 **Program Target Device** to program the device and verify the hardware result by running following script in MATLAB:

```
hdlcoder_external_memory_hw_run_ZCU102
```

This script first initializes the `Matrix_Size` to 2000, which means a 2000x2000 matrix. In this script, the DUT is reading from base address '80000000', and write to base address '90000000'.

Finally, the output result is read back to MATLAB, and compared with the expected value. In this way, the hardware results are verified in MATLAB.

```
>> hdlcoder_external_memory_hw_run_ZCU102
Initializing external DDR4 memory (data size 4002000) ...
Starting DUT IP core processing ...
Verifying result ...
PASSED: Matrix vector multiplication output matches with the expected data.
```

Authoring a Reference Design for Audio System on a Zynq Board

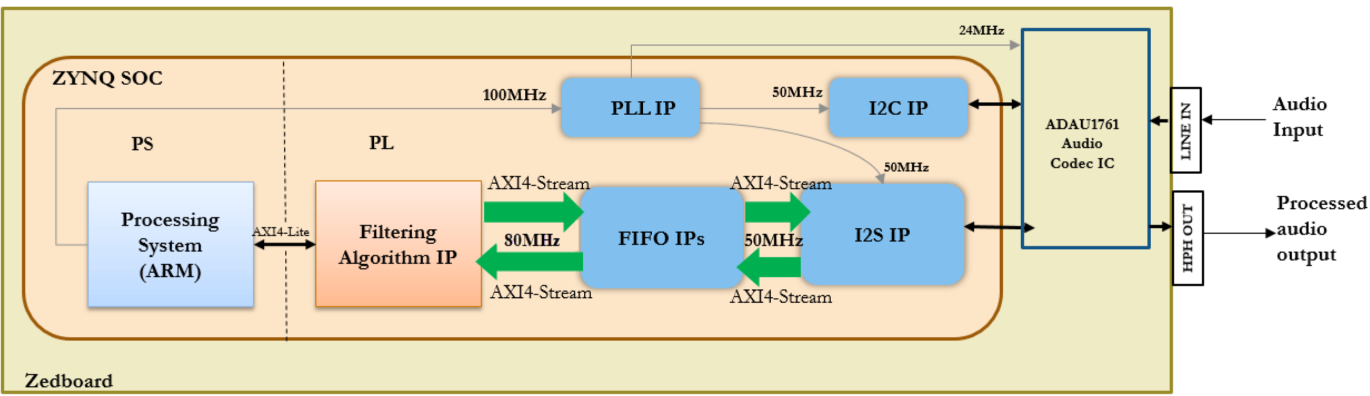
This example shows how to build a reference design to run an audio algorithm and access audio input and output on a Zynq® board.

Introduction

In this example you will create a reference design which receives audio input from Zedboard, performs some processing on it and transmits the processed audio data out of Zedboard. You also generate IP cores for peripheral interfaces using **HDL Workflow Advisor**.

To perform audio processing on Zedboard, you need the following 2 protocols:

1. I2C to configure the ADAU1761 audio codec chip on Zedboard.
2. I2S to stream the digitized audio data between the codec chip and Zynq fabric.



The above figure is a high level architecture diagram that shows how the reference design is used by the Filtering Algorithm IP. I2S IP operates at 50MHz frequency whereas one may want to run the Filtering Algorithm IP at a higher frequency. This frequency is controlled in Step **1.4** in **HDL Workflow Advisor**. In this example, assume that the filter operates at 80MHz. Since I2S and Filtering Algorithm IPs operate at different frequencies, we need FIFOs to handle clock domain crossing. Depending on the type of filter selected, Filtering Algorithm IP filters a range of frequencies from the incoming audio data and passes the filtered audio data out. In the above figure, the Filtering Algorithm IP is our main algorithm that we are modeling in Simulink. While FIFO IP is provided in Vivado, the I2C and I2S IPs need to be created. Here, the user has 3 choices: (a) use pre-packaged IP, e.g., if one exists, (b) model it in Simulink and generate an IP core using IP core generation workflow or (c) use legacy HDL code. To create an IP out of legacy HDL code, use a Simulink model to black box the HDL code and generate IP core from it. FIFO IPs handle the clock domain crossing between incoming audio data at 50MHz and the filter IP running at 80MHZ. I2C, I2S, PLL, FIFO IPs and Processing System form a part of the reference design.

The following steps are used to create the reference design described above:

1. Generate IP Cores for peripheral interfaces
2. Create a custom audio codec reference design in Vivado
3. Create the reference design definition file

4 Verify the reference design

1. Generate IP Cores for peripheral interfaces using HDL Workflow Advisor

In this example,

1. I2C IP is developed by modeling it using Stateflow blocks, and also using legacy VHDL code for tristate buffer.
2. I2S IP is developed by modeling it in Simulink.

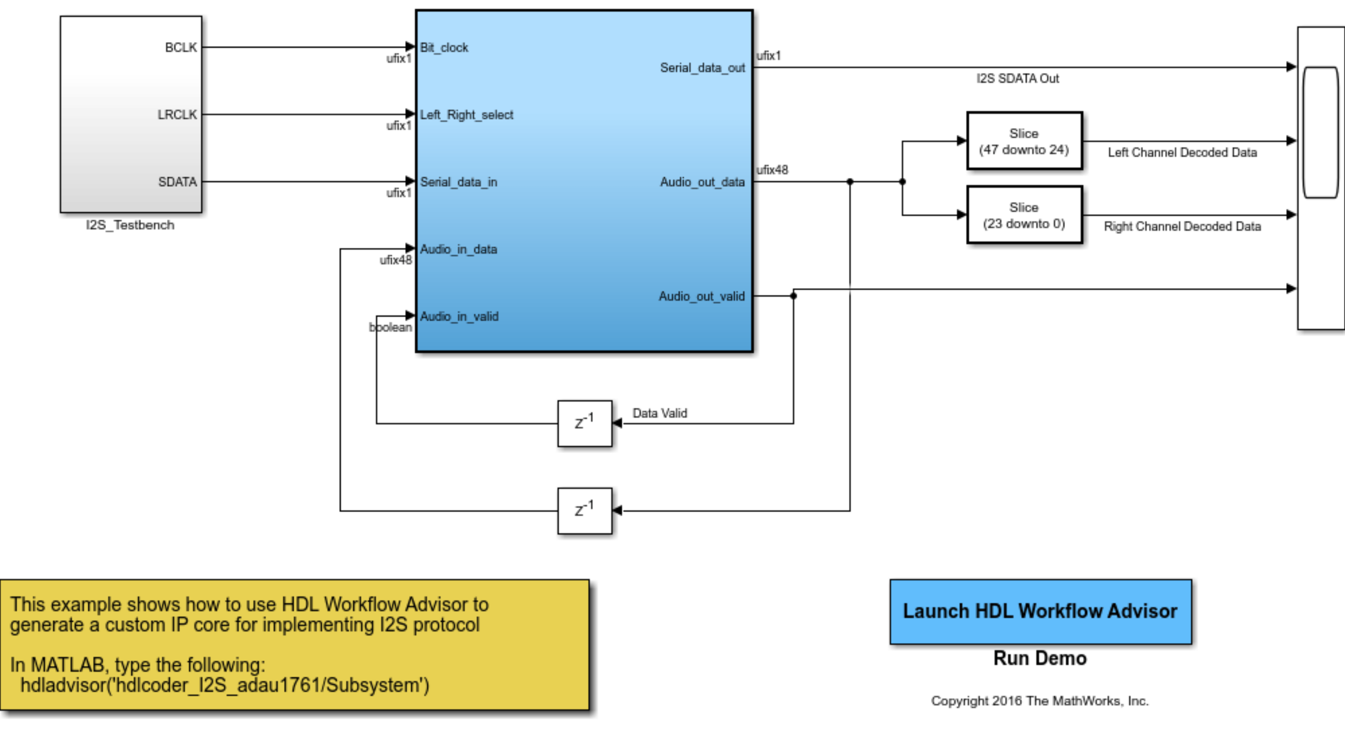
1.1 Creation of I2C IP

For creation of I2C IP to configure Audio Codec ADAU1761, refer to “IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip” on page 40-150 article.

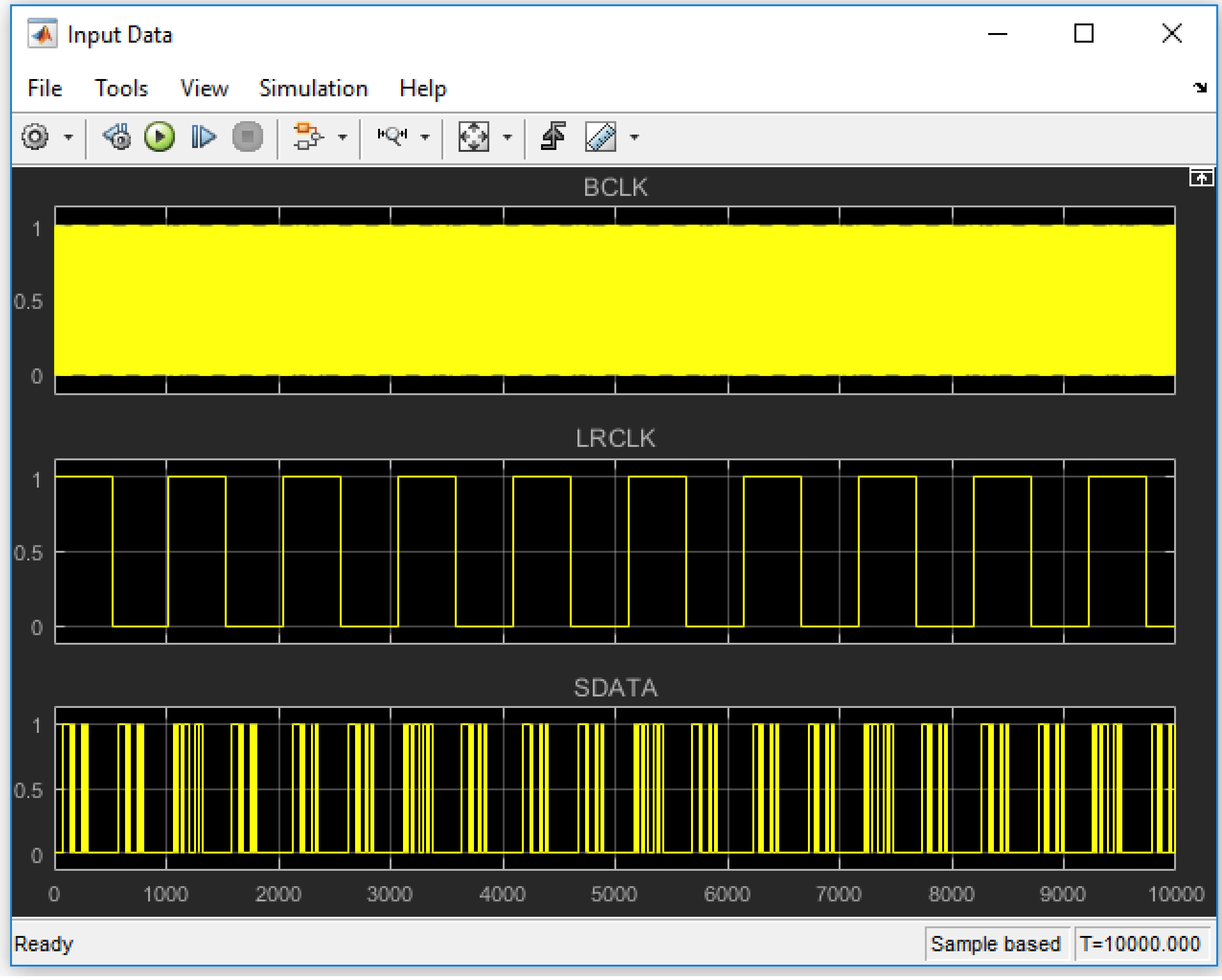
1.2 Creation of I2S IP

Design a model in Simulink with a MATLAB function which implements the I2S protocol.

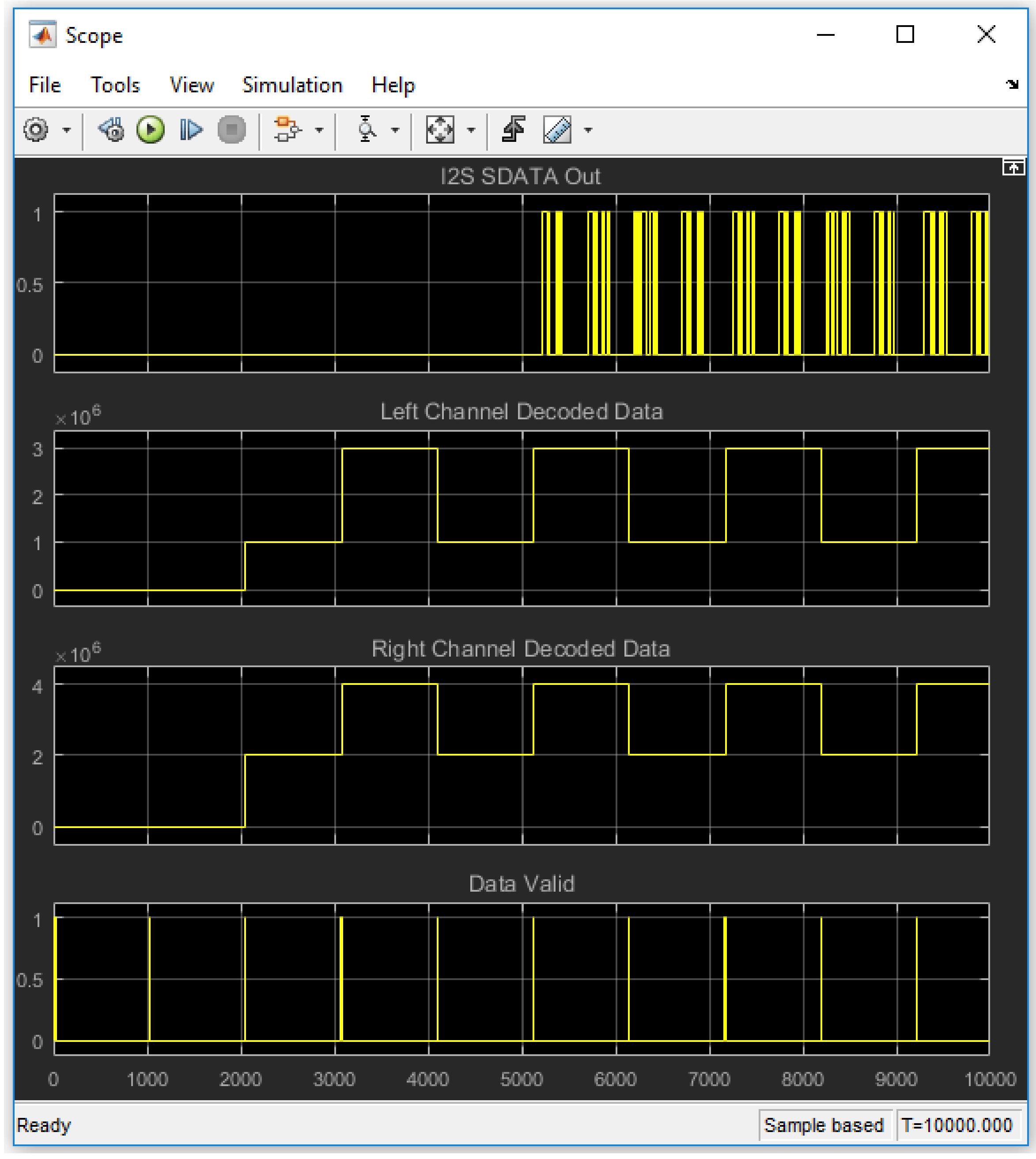
```
modelname = 'hdlcoder_I2S_adau1761';
open_system(modelname);
```



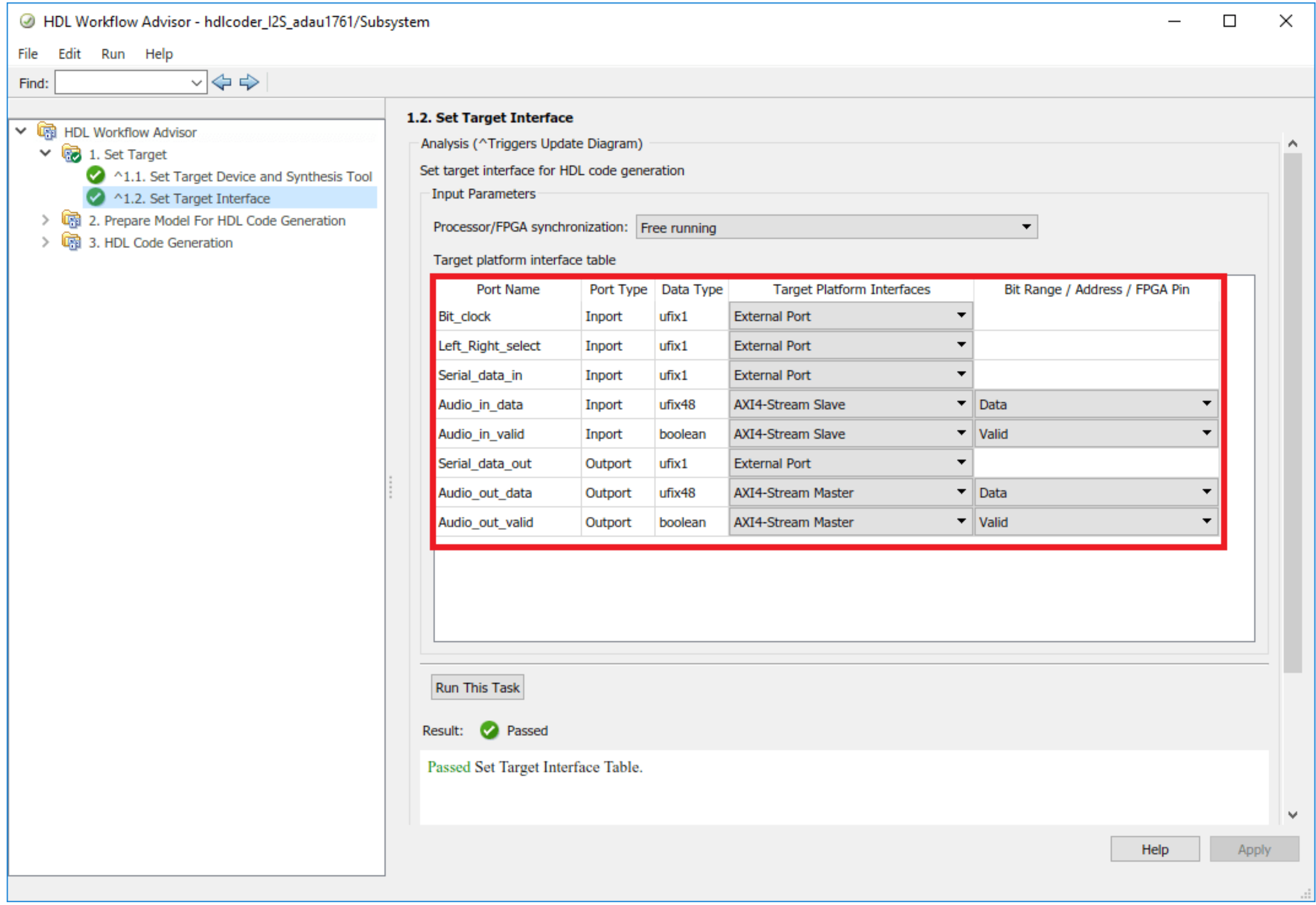
Create a test bench in the model to mimic the incoming audio data from the codec.



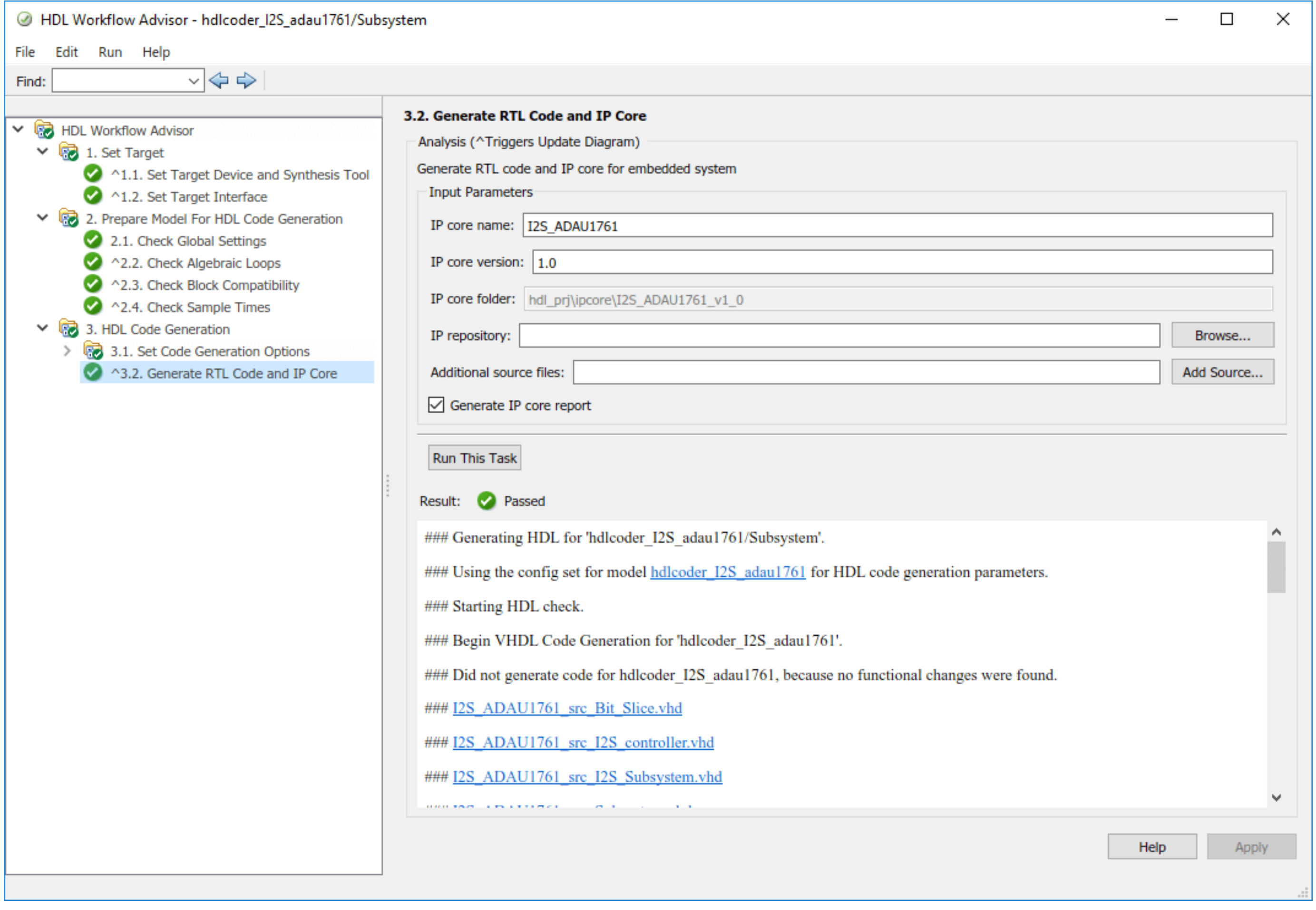
Feed this data to the Subsystem block which does the I2S operation. Verify the output of the Subsystem on a Scope.



Start the HDL Workflow Advisor from the DUT subsystem. In Task 1.1, keep the same settings as those of I2C IP generated earlier. in Task 1.2, set the Target Platform Interfaces as shown below:



Run Task 3.2 and generate the ip core.



2. Create a custom audio codec reference design in Vivado

I2C, I2S and FIFO IPs are incorporated in the custom reference design. To create a custom reference design, refer to the "Create and export a custom reference design using Xilinx Vivado" section in “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252.

Key points to be noted while creating this custom reference design:

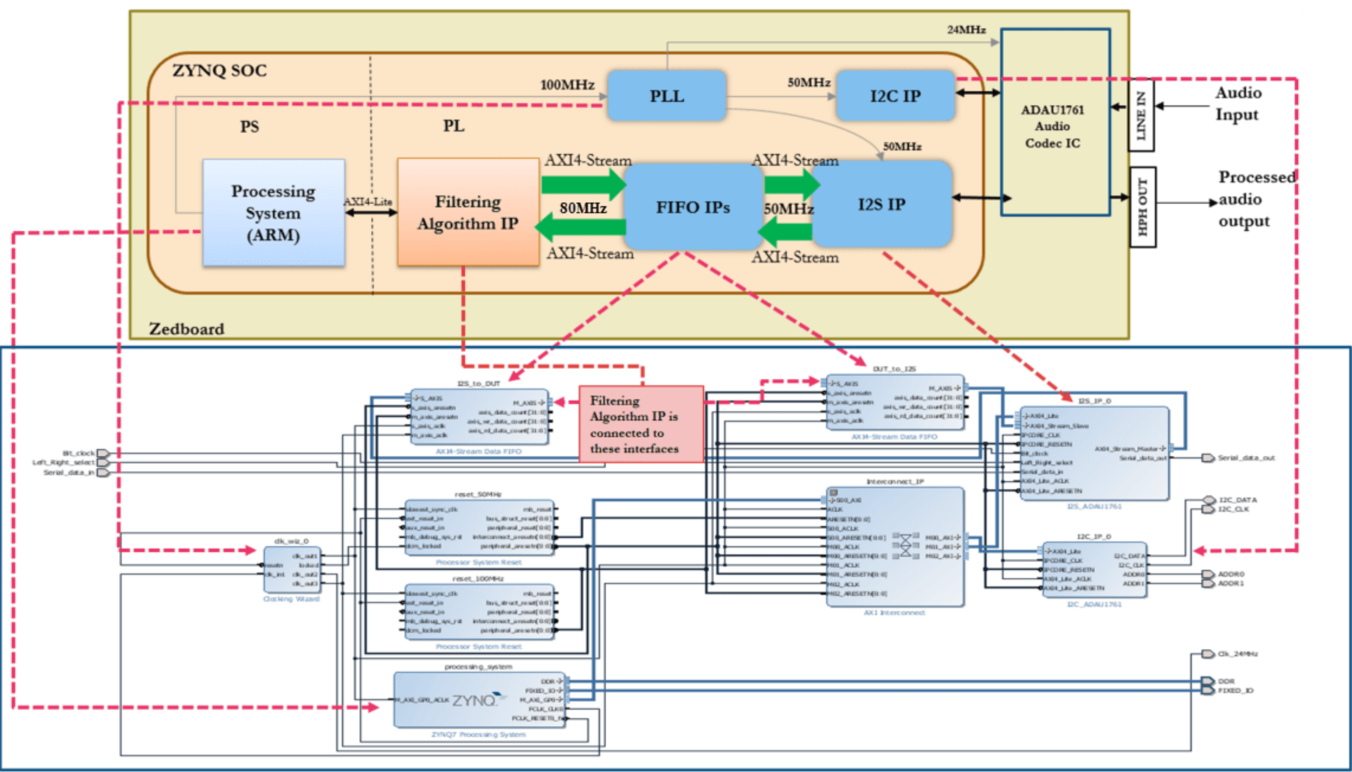
1. We must understand the theory of operation of the audio codec chip on the Zedboard.
2. The FIFOs are set to default values for their configuration.
3. For the IP cores generated using HDL Workflow Advisor, **IPCORE_CLK** and **AXI4_Lite_ACLK** should be connected to the same clock source.
4. On validating the block design in Vivado, there should be no critical warnings except for the unconnected ports.
5. In this reference design, the audio codec is configured to operate in the Master mode.

The following signals run between the reference design on Zynq Soc and the audio codec on Zedboard:

1. **Bit_clock** is the product of the sampling frequency, the number of bits per channel and the number of channels. It is driven by the audio codec in master mode. In this example, Sampling frequency is 48KHz, No of channels is 2, Number of bits per channel is 24.

2 **Left_right_select** is to distinguish between left audio channel data and right audio channel data. It is in sync with the Bit clock.
3 **Serial_data_in** is the analog to digital converted audio data from the codec.
4 **Serial_data_out** is the digital audio data going to codec to be converted into analog form.
5 **I2C_CLK** and **I2C_DATA** are standard I2C signals
6 **ADDR0** and **ADDR1** are the I2C Address Bits.
7 **Clk_24MHz** is the 24MHz clock signal required by the codec.

The custom audio codec reference design created for this example is shown below:



3. Create the reference design definition file

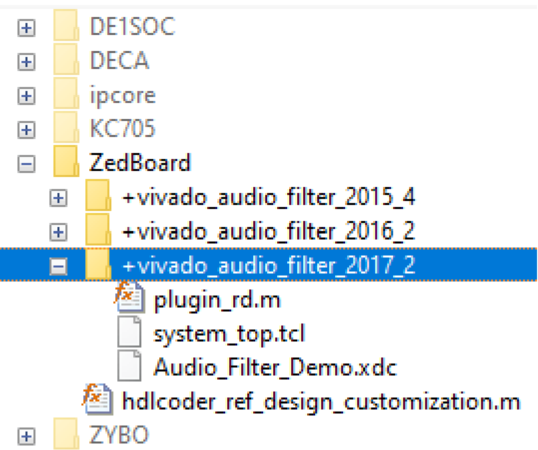
The following code describes the contents of the Zedboard reference design definition file **plugin_rd.m** for the above reference design. For more details on how to define and register custom board, refer to "Define Custom Board and Reference Design for Zynq Workflow" on page 40-252 example.

```
function hRD = plugin_rd()
% Reference design definition

%   Copyright 2016-2018 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Audio System with AXI4 Stream Interface';
hRD.BoardName = 'ZedBoard';

%% Tool information
hRD.SupportedToolVersion = {'2017.2','2017.4'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',       'em.avnet.com:zed:part0:1.0');

hRD.addIPRepository(...
    'IPListFunction', 'mathworks.hdlcoderdemo.vivado.hdlcoderdemo_adau1761_iplist');

% Add constraint files
hRD.CustomConstraints = {'Audio_Filter_Demo.xdc'};

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',      'clk_wiz_0/clk_out3', ...
    'ResetConnection',      'reset_100MHz/peripheral_aresetn',...
    'DefaultFrequencyMHz', 80,...
    'MinFrequencyMHz',      5,...
    'MaxFrequencyMHz',      500,...
    'ClockModuleInstance', 'clk_wiz_0',...
    'ClockNumber',          3);


% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'Interconnect_IP/M02_AXI', ...
    'BaseAddress',          '0x400D0000', ...
    'MasterAddressSpace',   'processing_system/Data');

% add AXI4-Stream interface
hRD.addAXI4StreamInterface( ...
    'MasterChannelNumber',       1, ...
    'SlaveChannelNumber',        1, ...
    'MasterChannelConnection', 'DUT_to_I2S/S_AXIS', ...
    'SlaveChannelConnection',  'I2S_to_DUT/M_AXIS', ...
    'MasterChannelDataWidth',   48, ...
    'SlaveChannelDataWidth',    48 ...
```

Add the **Zedboard** folder to MATLAB path using the following command:

```
example_root = (hdlcoder_amd_examples_root)
cd (example_root)
addpath(genpath('ZedBoard'));
```

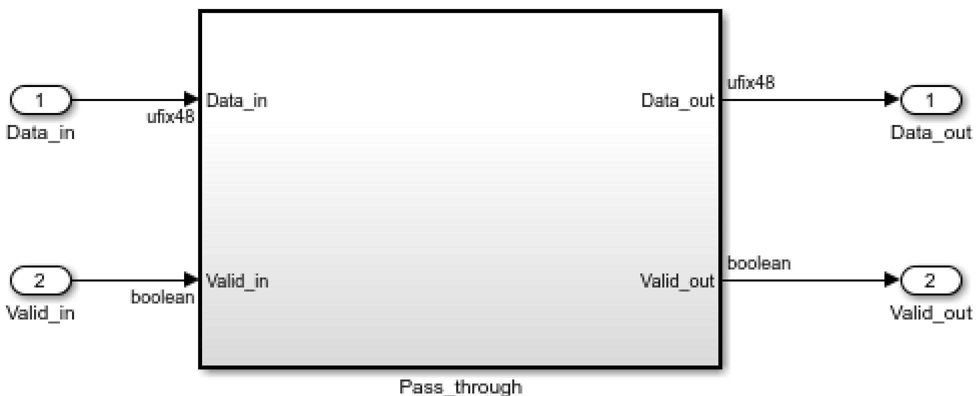
All files that are required for the reference design such as IP core files, XDC files, plugin_rd file etc should be added to the MATLAB path, inside **Zedboard** folder using the hierarchy shown below. The user generated IP core files should be in **+vivado** folder. plugin_rd.m, tcl files and xdc files should be in the reference design plugin folder, for example, **+vivado_audio_filter_2017_2** folder.



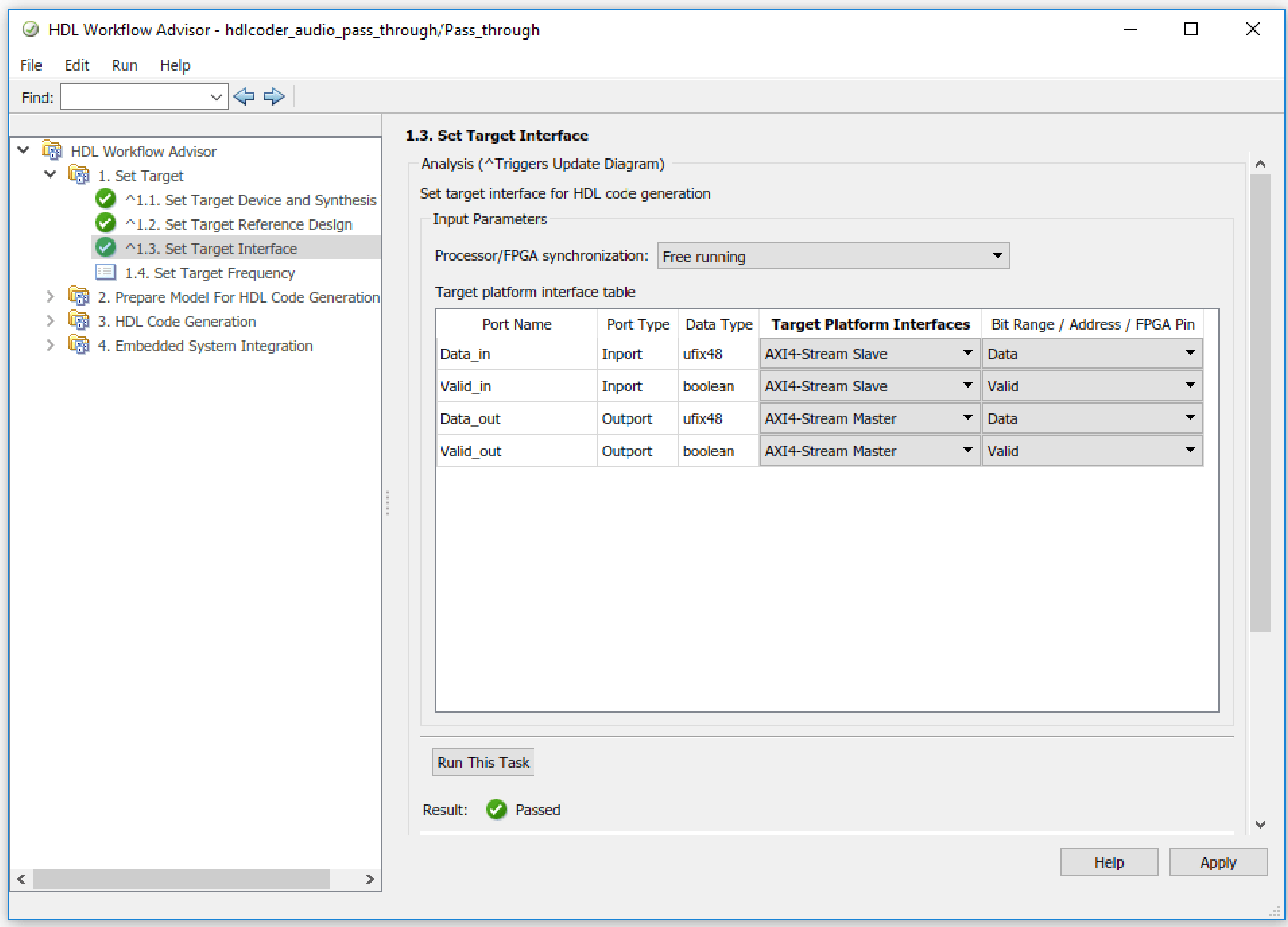
4. Verify the reference design

In order to ensure that the reference design and the interfaces in the reference design work as expected, design a Simulink model which just sends the audio through the Algorithm IP, integrate it with the reference design and test it on Zedboard. The user should be able to hear the audio loop back.

```
modelname = 'hdlcoder_audio_pass_through';
open_system(modelname);
```



The interfaces in the model should be selected as shown below:



To generate IP core from a model and integrate it with the audio codec reference design, refer to "Running an Audio Filter on Live Audio Input Using a Zynq Board" on page 40-181 example.

Authoring a Reference Design for Audio System on a ZYBO Board

This example shows how to build a reference design to run an audio algorithm and access audio input and output on ZYBO™ Z7-10 board.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder™ Support Package for Xilinx® FPGA and SoC Devices
- Xilinx Vivado®, with latest version mentioned in the documentation
- Digilent® Zybo Z7-10 Zynq development board with the accessory kit

This example can be used as reference for creating custom reference design for Digilent® ZYBO Z7-20 board.

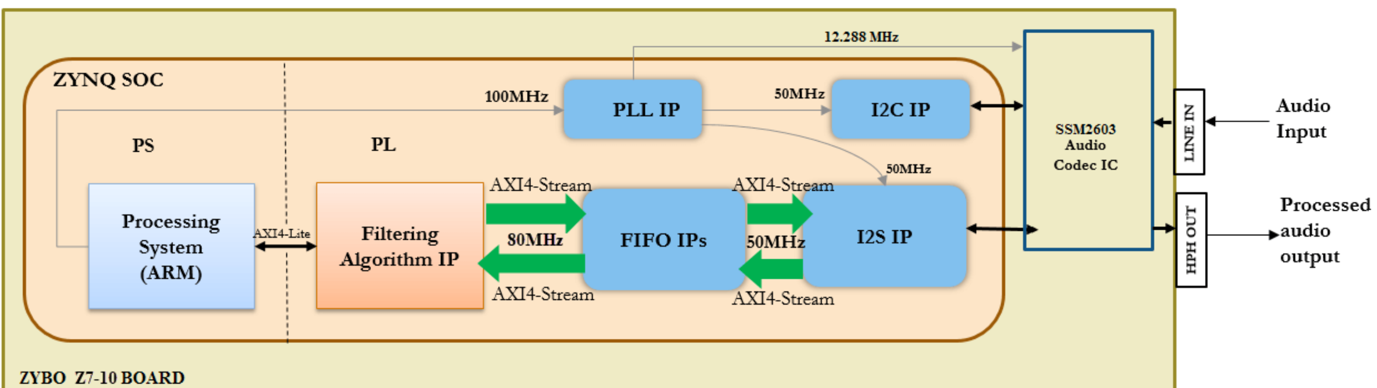
To setup the ZYBO Z7-10 board, refer to the *Set up the Zybo board* section in the "Define Custom Board and Reference Design for Zynq Workflow" on page 40-252 article.

Introduction

In this example you will create a reference design which receives audio input from ZYBO Z7-10 board, performs some processing on it and transmits the processed audio data out of ZYBO Z7-10 board. You also generate IP cores for peripheral interfaces using **HDL Workflow Advisor**.

To perform audio processing on ZYBO Z7-10 board, following 2 protocols are needed:

1. I2C to configure the SSM2603 audio codec chip on ZYBO Z7-10 board.
2. I2S to stream the digitized audio data between the codec chip and zynq fabric.



The above figure is a high level architecture diagram that shows how the reference design is used by the Filtering Algorithm IP on ZYBO Z7-10 board. This example is similar to the audio system reference design for Zedboard except that the ZYBO Z7-10 board uses a SSM2603 audio codec chip whereas the Zedboard uses ADAU1761 audio codec chip. Rest of the operating parameters are same as the Audio System reference design for Zedboard. For more details, please refer to "Authoring a Reference Design for Audio System on a Zynq Board" on page 40-226 example.

The following steps are used to create the reference design described above:

1. Generate IP Cores for peripheral interfaces
2. Create a custom audio codec reference design in Vivado
3. Create the reference design definition file
4. Verify the reference design

1. Generate IP Cores for peripheral interfaces using HDL Workflow Advisor

In this example,

1. I2C IP is developed using Stateflow® blocks & legacy VHDL code for tristate buffer.
2. I2S IP is developed by modelling it in Simulink.

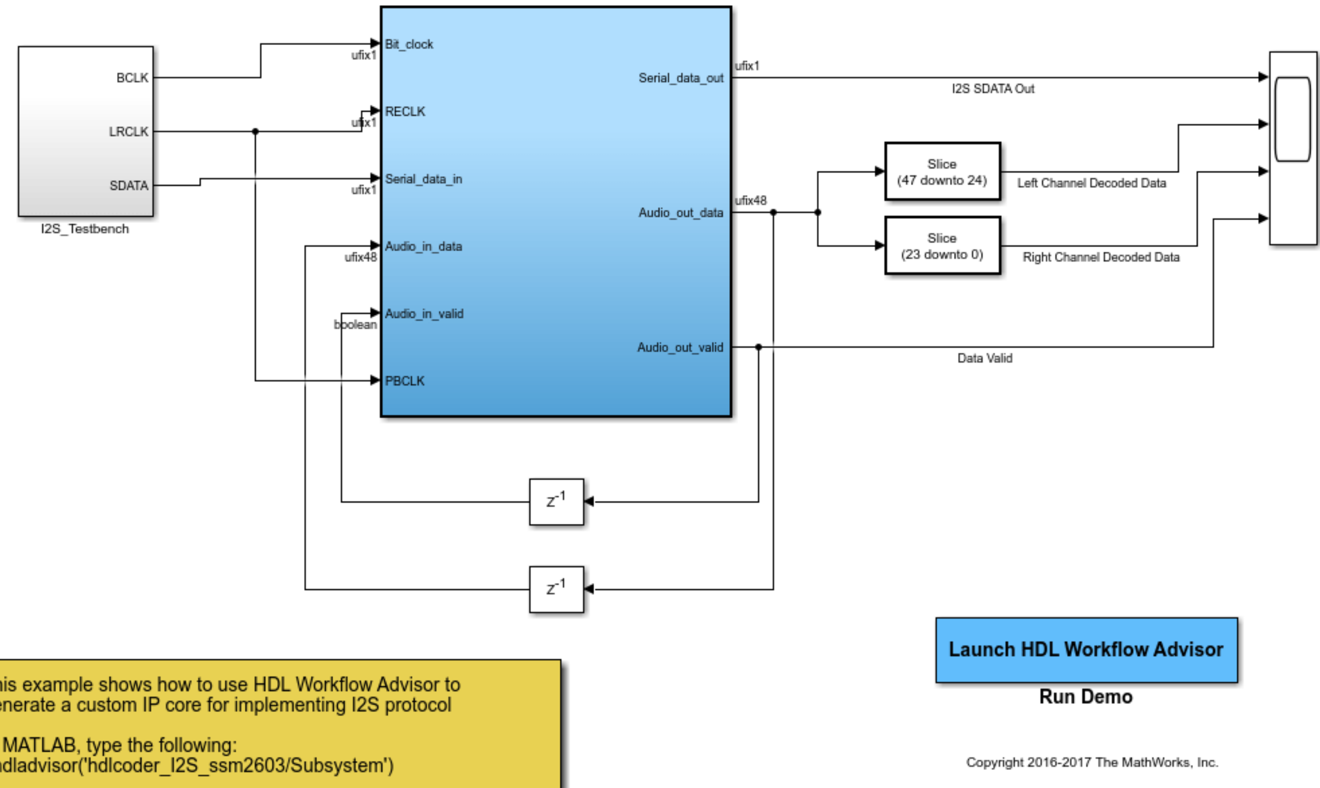
1.1 Creation of I2C IP

For creation of I2C IP to configure Audio Codec SSM2603, refer to “IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip” on page 40-150 article.

1.2 Creation of I2S IP

Design a model in Simulink® with a MATLAB® function which implements the I2S protocol.

```
modelname = 'hdlcoder_I2S_ssm2603';
open_system(modelname);
```



Testing and IP core generation steps are same as Zedboard I2S model. For generation of I2S IP, see “Authoring a Reference Design for Audio System on a Zynq Board” on page 40-226 example.

2. Create a custom audio codec reference design in Vivado

I2C, I2S and FIFO IPs are incorporated in the custom reference design. To create a custom reference design, refer to the "Create and export a custom reference design using Xilinx Vivado" section in “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252.

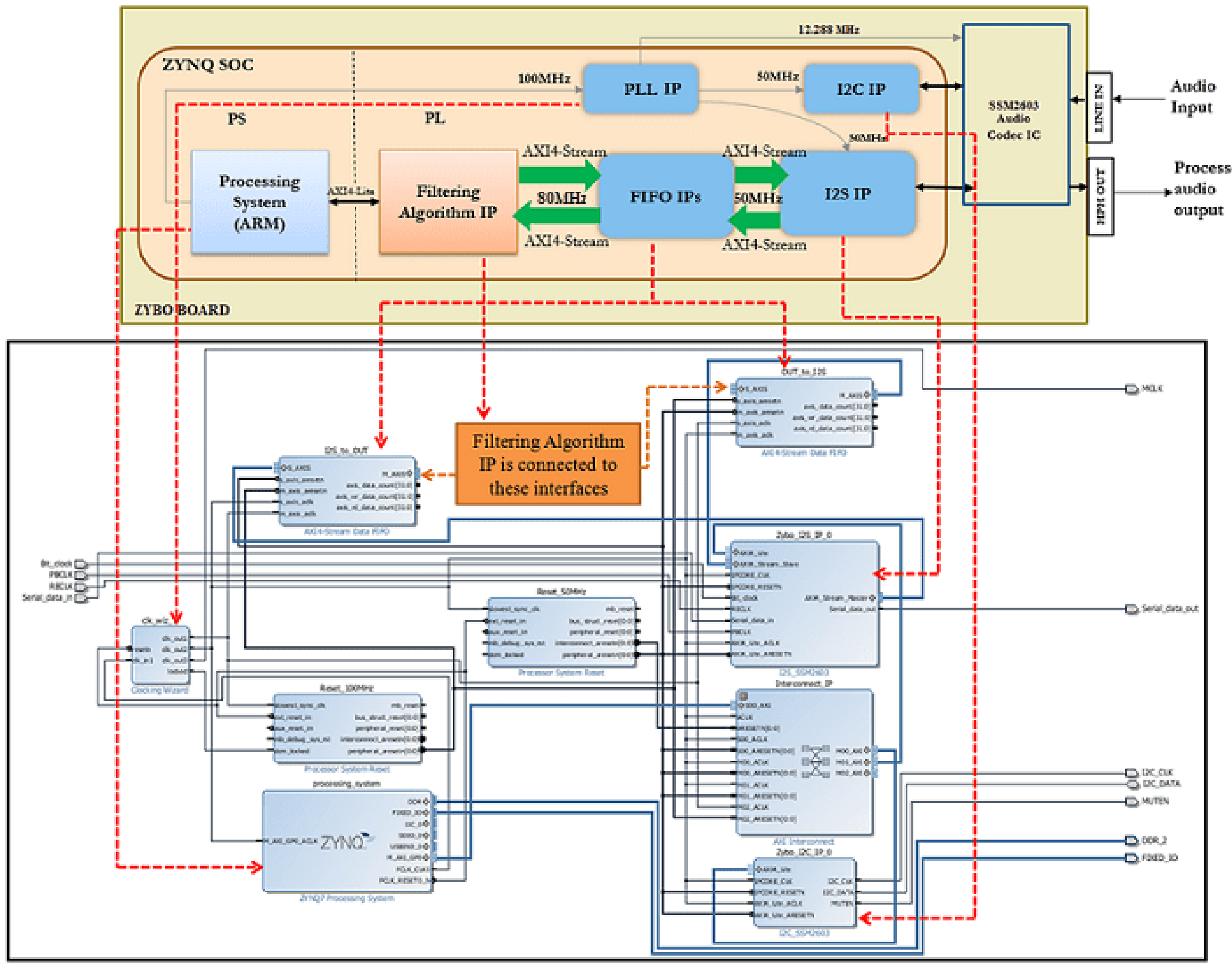
Key points to be noted while creating this custom reference design:

1. We must understand the theory of operation of the audio codec chip on the ZYBO board.
2. The FIFOs are set to default values for their configuration.
3. For the IP cores generated using HDL Workflow Advisor, **IPCORE_CLK** and **AXI4_Lite_ACLK** should be connected to the same clock source.
4. On validating the block design in Vivado, there should be no critical warnings except for the unconnected ports.
5. In this reference design, the audio codec is configured to operate in the Master mode.

The following signals run between the reference design on Zynq Soc and the audio codec on ZYBO board:

1. **Bit_clock** is the product of the sampling frequency, the number of bits per channel and the number of channels. It is driven by the audio codec in master mode. In this example, Sampling frequency is 48KHz, No of channels is 2, Number of bits per channel is 24.
2. **Serial_data_in** is the analog to digital converted audio data from the codec.
3. **Serial_data_out** is the digital audio data going to codec to be converted into analog form.
4. **I2C_CLK** and **I2C_DATA** are standard I2C signals
5. **MUTEN** is the Hardware mute pin connected to audio codec SSM2603.
6. **MCLK** is the 12.288MHz clock signal required by the codec.

The custom audio codec reference design created for this example is shown below:



3. Create the reference design definition file

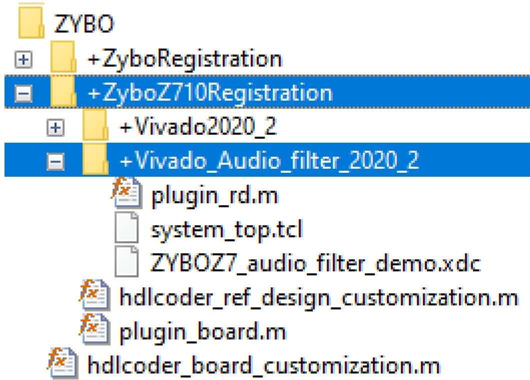
The following code describes the contents of the ZYBO board reference design definition file **plugin_rd.m** for the above reference design. For more details on how to define and register custom board, refer to “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252 example.

```
function hRD = plugin_rd()
% Reference design definition

%   Copyright 2021 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Audio System with AXI4 Stream Interface';
hRD.BoardName = 'ZYBO Z7-10';

% Tool information
hRD.SupportedToolVersion = {'2020.2'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'digilentinc.com:zybo-z7-10:part0:1.0');

hRD.addIPRepository(...
    'IPListFunction', 'mathworks.hdlcoderdemo.vivado.hdlcoderdemo_ssm2603_iplist');

% Add constraint files
hRD.CustomConstraints = {'ZYBOZ7_audio_filter_demo.xdc'};

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',     'clk_wiz_0/clk_out1', ...
    'ResetConnection',     'reset_100MHz/peripheral_aresetn',...
    'DefaultFrequencyMHz', 80,...
    'MinFrequencyMHz',     5,...
    'MaxFrequencyMHz',     500,...
    'ClockModuleInstance', 'clk_wiz_0',...
    'ClockNumber',         1);

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'Interconnect_IP/M02_AXI', ...
    'BaseAddress',         '0x40010000', ...
    'MasterAddressSpace',  'processing_system/Data');

% add AXI4-Stream interface
hRD.addAXI4StreamInterface( ...
    'MasterChannelEnable',      true, ...
    'SlaveChannelEnable',       true, ...
    'MasterChannelConnection', 'DUT_to_I2S/S_AXIS', ...
    'SlaveChannelConnection',  'I2S_to_DUT/M_AXIS', ...
    'MasterChannelDataWidth',   48, ...
    'SlaveChannelDataWidth',    48 ...
    );

hRD.DeviceTreeName = 'devicetree_axilite.dtb';
```

Add the **ZYBO** folder to the MATLAB path using the following command:

```
example_root = (hdlcoder_amd_examples_root)
cd (example_root)
addpath(genpath('ZYBO'));
```

All files that are required for the reference design such as IP core files, XDC files, plugin_rd file etc should be added to the MATLAB path, inside **ZYBO** folder using the hierarchy shown below. The user generated IP core files should be in **+vivado** folder. plugin_rd.m, tcl files and xdc files should be in **+vivado_audio_filter_2020_2** folder.



4. Verify the reference design

To verify the reference design, to generate Audio Filter IP core from a model and integrate it with the audio codec reference design, refer to “Running an Audio Filter on Live Audio Input Using a Zynq Board” on page 40-181 example.

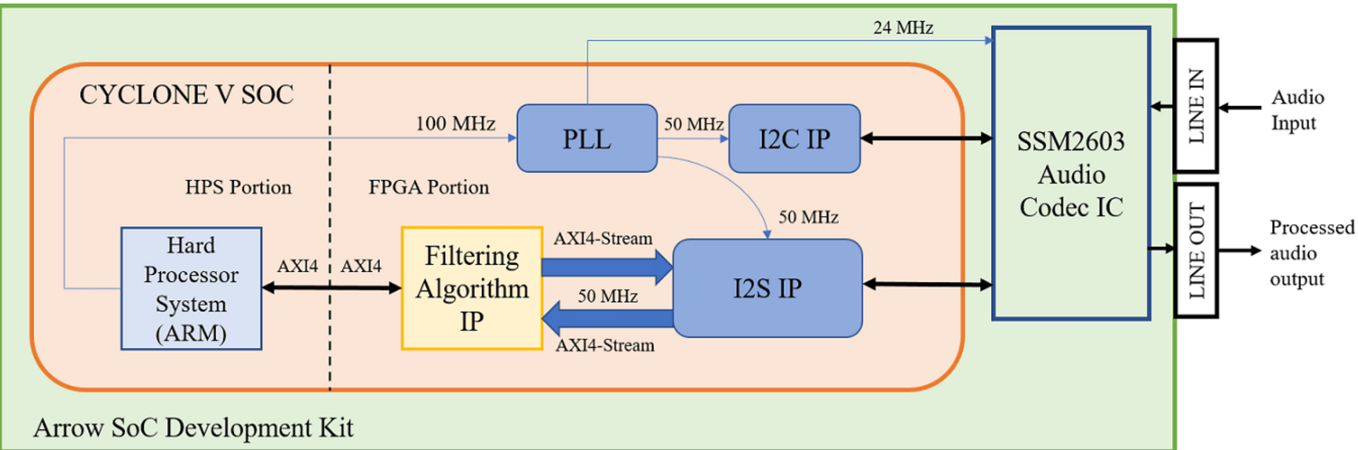
Authoring a Reference Design for Audio System on Intel Board

This example shows how to build a reference design to run an audio algorithm and access audio input and output on an Intel® Arrrow® SoC board.

Introduction

In this example, you create a reference design which receives audio input from Intel Arrow SoC Development Kit, performs some processing on it and transmits the processed audio data out of Arrow SoC Development Kit. To perform audio processing on Arrow SoC, you need the following 2 protocols:

1. I2C to configure the SSM2603 audio codec chip on Arrow SoC.
2. I2S to stream the digitized audio data between the codec chip and Cyclone V FPGA.



The above figure is a high level architecture diagram that shows how the reference design is used by the Filtering Algorithm IP. I2S IP operates at 50MHz frequency whereas one may want to run the Filtering Algorithm IP at a higher frequency. This frequency is controlled in Step **1.4** in **HDL Workflow Advisor**. In this example, assume that the filter operates at 50MHz. Depending on the type of filter selected, Filtering Algorithm IP filters a range of frequencies from the incoming audio data and passes the filtered audio data out. In the above figure, the Filtering Algorithm IP is our main algorithm that we are modeling in Simulink. While the I2C and I2S IPs need to be created. Here, you has 3 choices:

- Use pre-packaged IP, e.g., if one exists
- Model it in Simulink and generate an IP core using IP core generation workflow or
- Use legacy HDL code. To create an IP out of legacy HDL code, use a Simulink model to black box the HDL code and generate IP core from it. I2C, I2S, PLL IPs and Cyclone V Hard Processor System(HPS) form a part of the reference design.

The following steps are used to create the reference design described above:

1. Generate IP Cores for peripheral interfaces
2. Create a custom audio codec reference design in Qsys
3. Create the reference design definition file

4. Verify the reference design

1. Generate IP Cores for peripheral interfaces using HDL Workflow Advisor

In this example,

1. I2C IP is developed by modeling it using Stateflow blocks, and also using legacy VHDL code for tristate buffer.
2. I2S IP is developed by modeling it in Simulink.

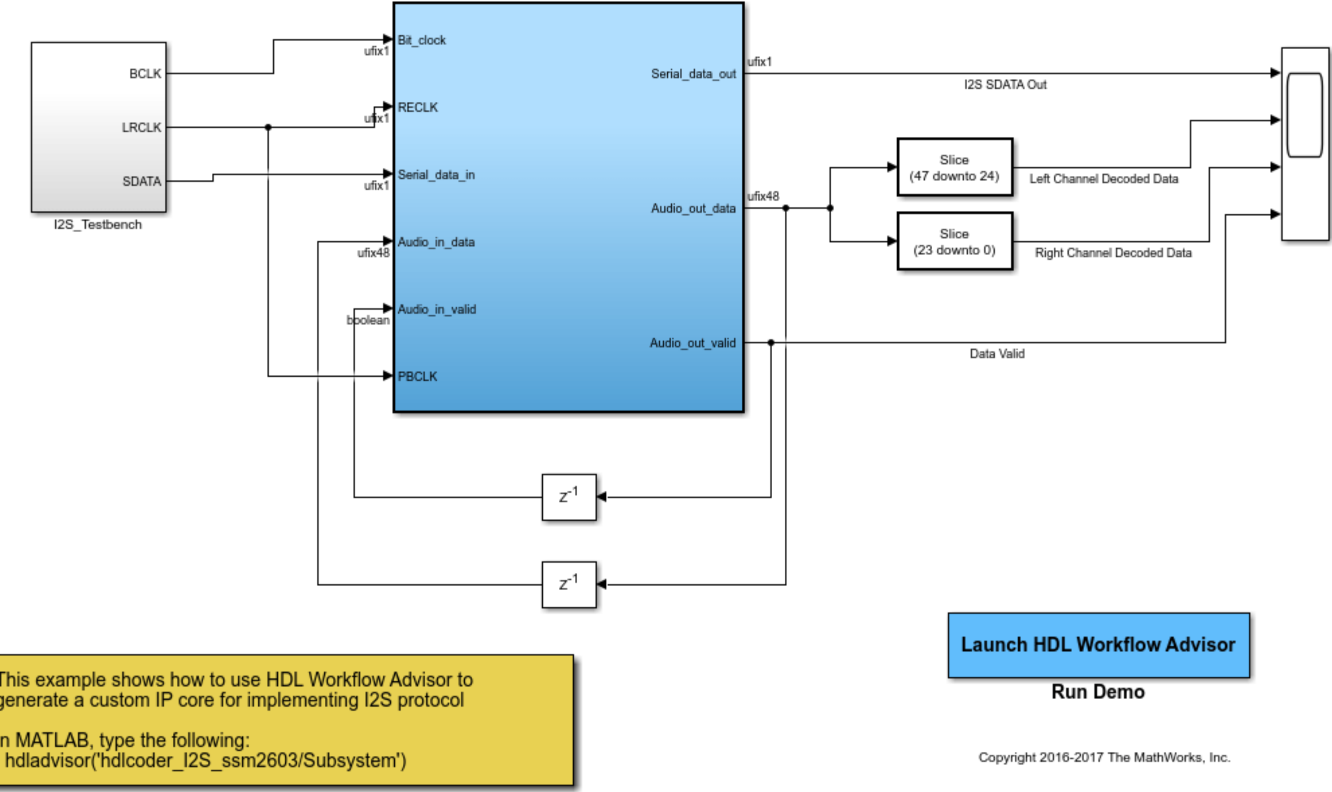
1.1 Creation of I2C IP

For creation of I2C IP to configure Audio Codec SSM2603, refer to “IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip” on page 40-150.

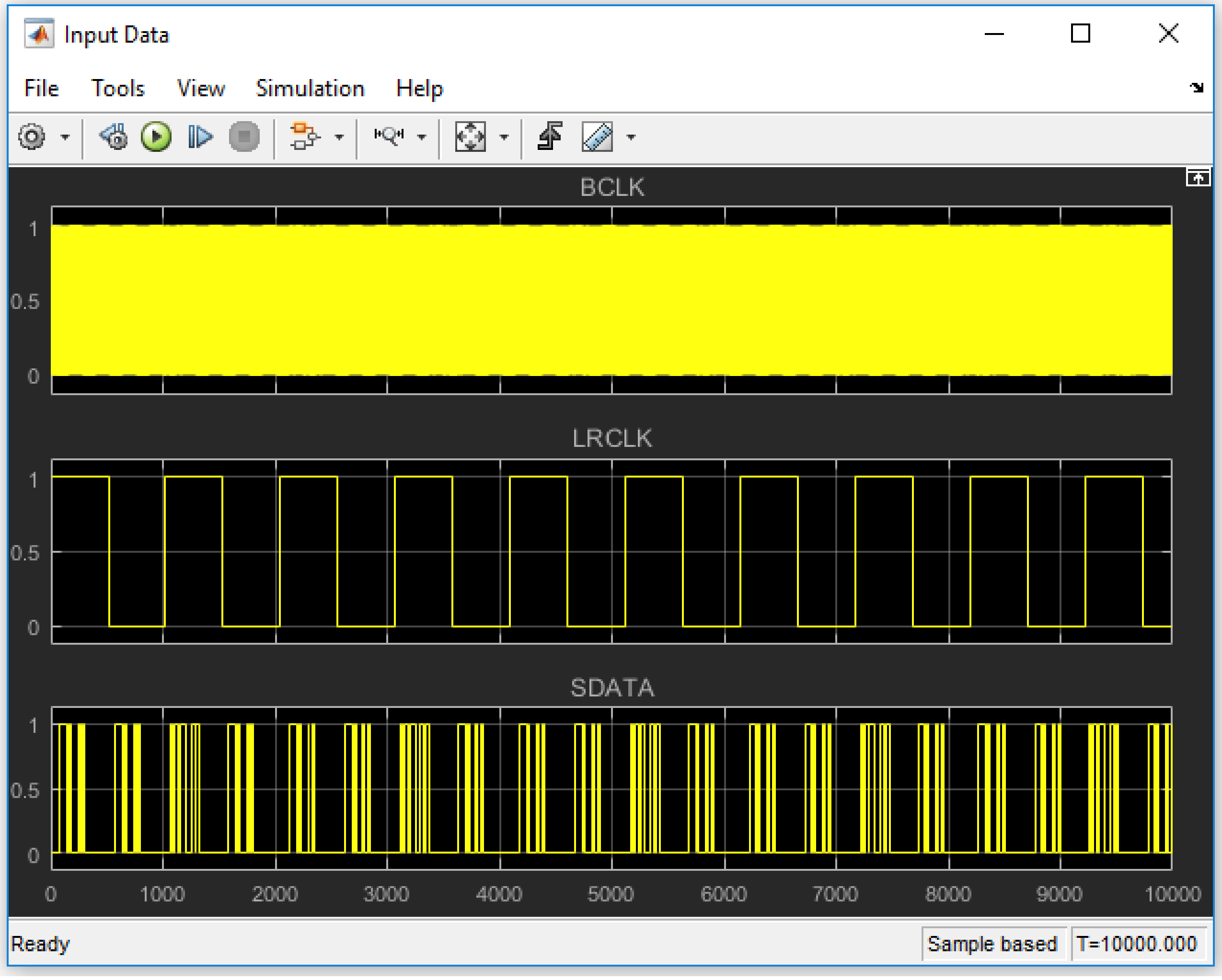
1.2 Creation of I2S IP

Design a model in Simulink with a MATLAB function which implements the I2S protocol.

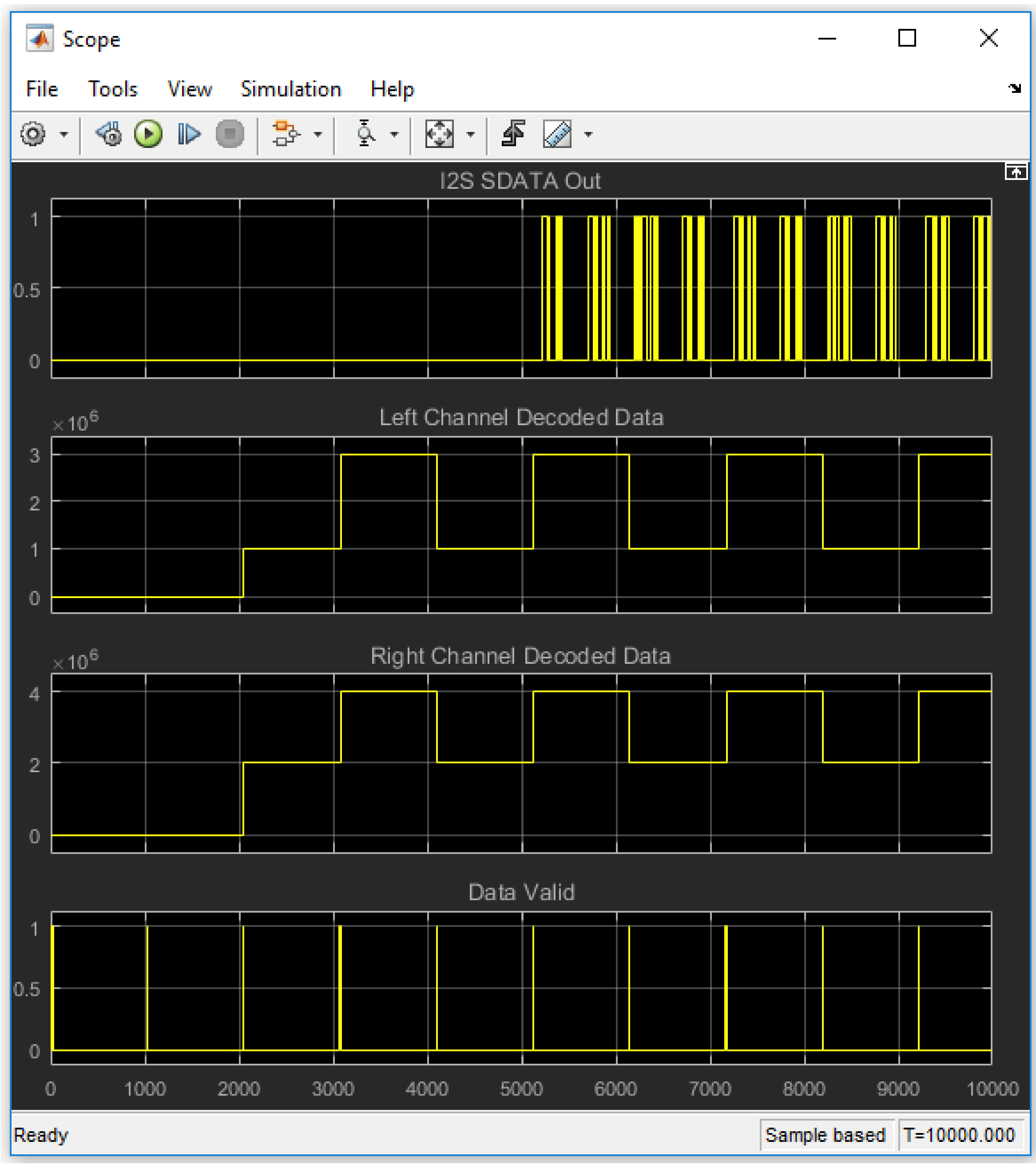
```
modelname = 'hdlcoder_I2S_ssm2603';
open_system(modelname);
```



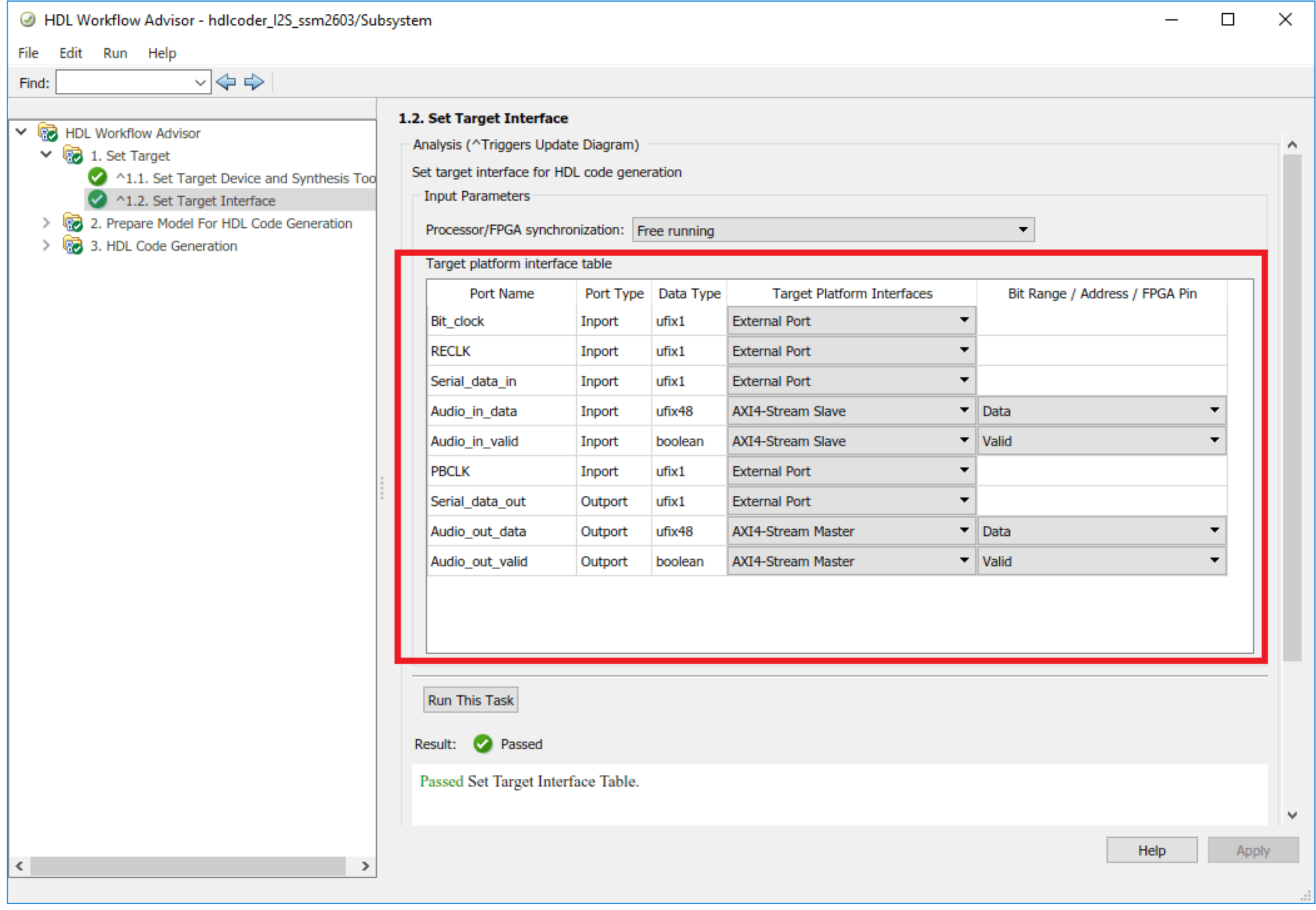
Create a test bench in the model to mimic the incoming audio data from the codec.



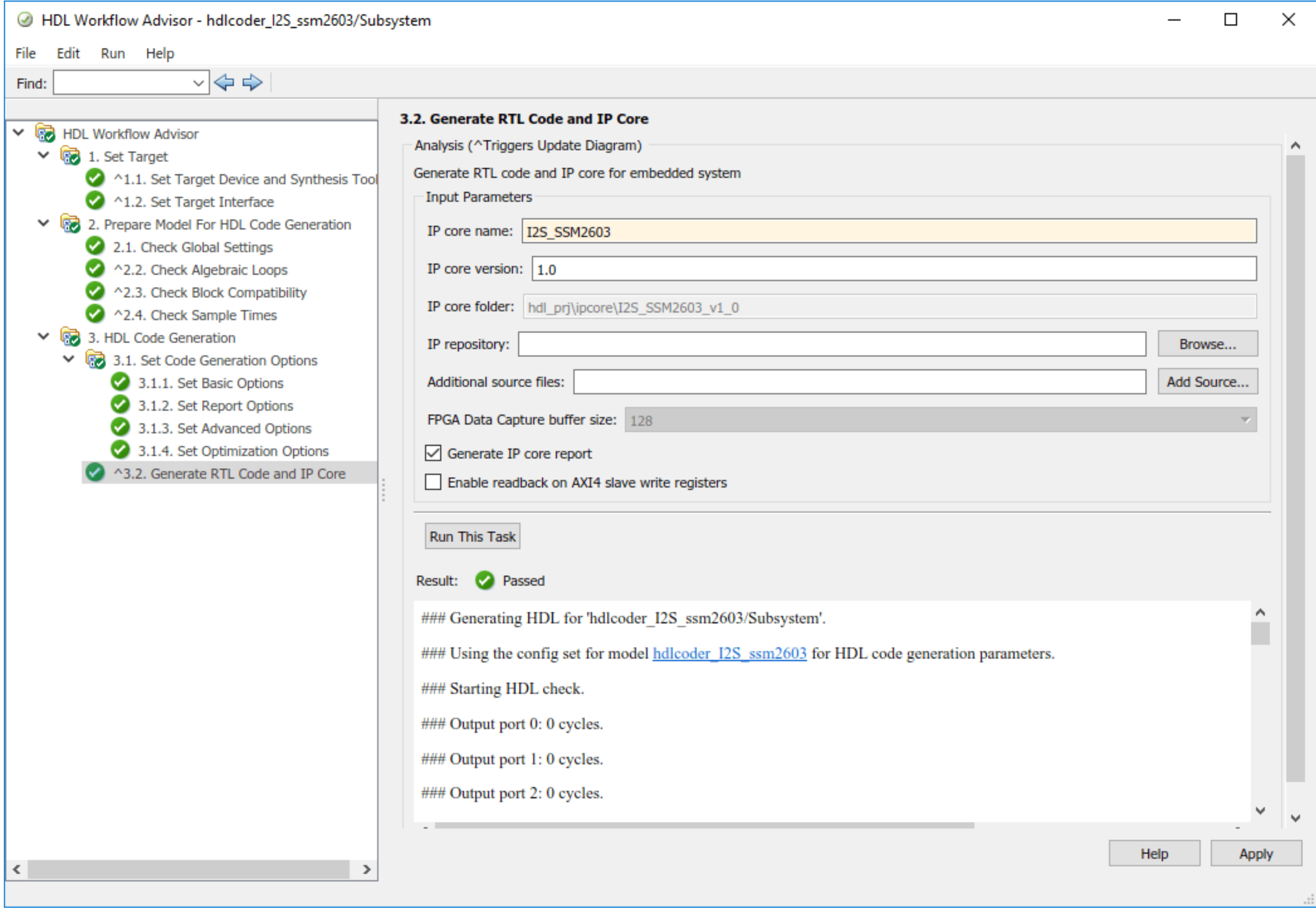
Feed this data to the Subsystem block which does the I2S operation. Verify the output of the Subsystem on a Scope.



Start the HDL Workflow Advisor from the DUT subsystem. In Task 1.1, keep the same settings as those of I2C IP generated earlier. in Task 1.2, set the Target Platform Interfaces as shown below:



Run Task 3.2 and generate the ip core.



2. Create a custom audio codec reference design in Qsys

I2C and I2S IPs are incorporated in the custom reference design. To create a custom reference design, refer to the **Reference Design creation using Intel Quartus Prime** section in "Define Custom Board and Reference Design for Intel SoC Workflow" on page 40-270.

Key points to be noted while creating this custom reference design:

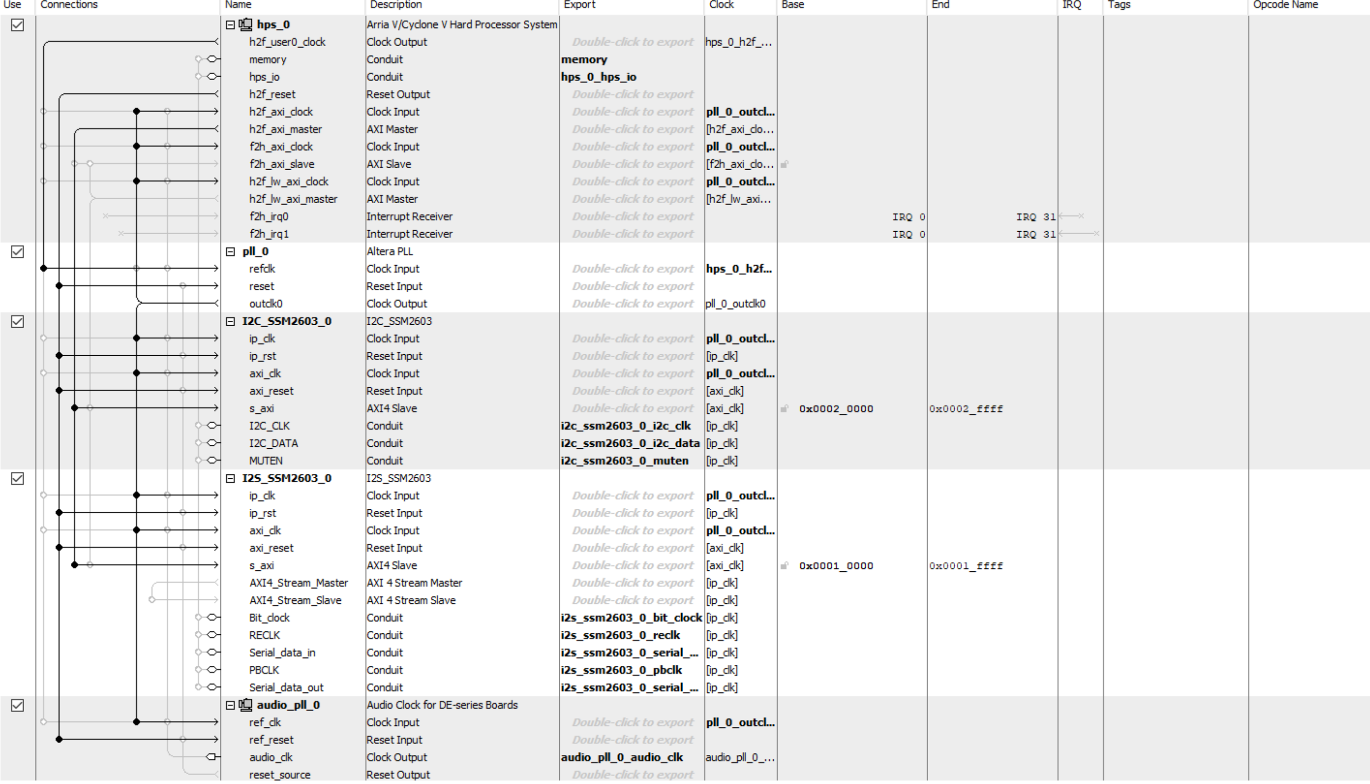
1. We must understand the theory of operation of the audio codec chip on the Arrow SoC.
2. For the IP cores generated using HDL Workflow Advisor, **IPCORE_CLK** and **AXI4_ACLK** should be connected to the same clock source.
3. In this reference design, the audio codec is configured to operate in the Master mode.

The following signals run between the reference design on Intel SoC and the audio codec on Arrow SoC:

1. **Bit_clock** is the product of the sampling frequency, the number of bits per channel and the number of channels. It is driven by the audio codec in master mode. In this example, Sampling frequency is 48KHz, No of channels is 2, Number of bits per channel is 24.
2. **Left_right_select** is to distinguish between left audio channel data and right audio channel data. It is in sync with the Bit clock.

3 **Serial_data_in** is the analog to digital converted audio data from the codec.
4 **Serial_data_out** is the digital audio data going to codec to be converted into analog form.
5 **I2C_CLK** and **I2C_DATA** are standard I2C signals
6 **ADDR0** and **ADDR1** are the I2C Address Bits.
7 **Clk_24MHz** is the 24MHz clock signal required by the codec.

The custom audio codec reference design created for this example is shown below:



3. Create the reference design definition file

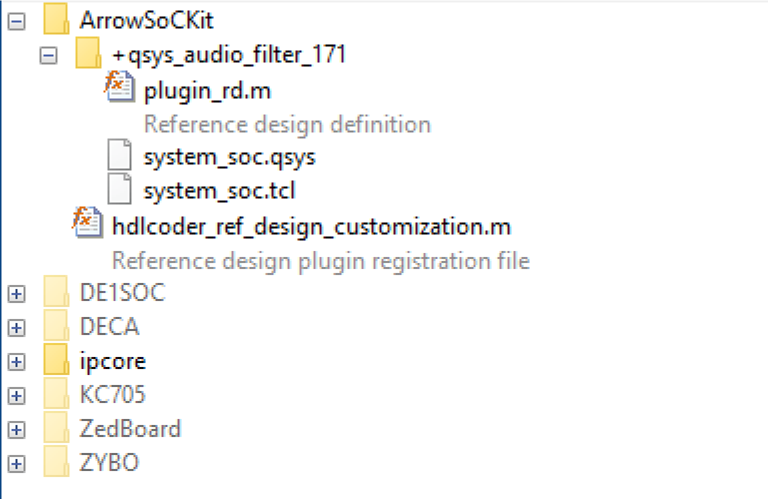
The following code describes the contents of the Arrow SoC Development Kit reference design definition file **plugin_rd.m** for the above reference design. For more details on how to define and register custom board, refer to “Define Custom Board and Reference Design for Intel SoC Workflow” on page 40-270.

```
function hRD = plugin_rd()
% Reference design definition

%   Copyright 2012-2019 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Altera QUARTUS II');

hRD.ReferenceDesignName = 'Audio System with AXI4 Stream Interface';

hRD.BoardName = 'Arrow SoCKit development board';

% Tool information
hRD.SupportedToolVersion = {'18.1'};


%% Add custom design files
% add custom Qsys design
hRD.addCustomQsysDesign( ...
    'CustomQsysPrjFile', 'system_soc.qsys');

hRD.addIPRepository(...
    'IPListFunction', 'mathworks.hdlcoderdemo.quartus.hdlcoderdemo_ssm2603_quartus_iplist');

% Add constraint files
hRD.CustomConstraints = {'system_soc.tcl'};



%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',      'pll_0.outclk0', ...
    'ResetConnection',      'hps_0.h2f_reset',...
    'DefaultFrequencyMHz', 50,...
    'MinFrequencyMHz',      5,...
    'MaxFrequencyMHz',      500,...
    'ClockModuleInstance', 'pll_0',...
    'ClockNumber',          0);

% add AXI4 slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'hps_0.h2f_axi_master', ...
    'BaseAddress',          '0x0000');

% add AXI4-Stream interface
hRD.addAXI4StreamInterface( ...
    'MasterChannelEnable',      true, ...
    'SlaveChannelEnable',       true, ...
    'MasterChannelConnection', 'I2S_SSM2603_0.AXI4_Stream_Slave', ...
    'SlaveChannelConnection',  'I2S_SSM2603_0.AXI4_Stream_Master', ...
    'MasterChannelDataWidth',   48, ...
    'SlaveChannelDataWidth',    48 ...
    );
```

Add to MATLAB path the **ArrowSoC** folder using the following commands:

```
example_root = (hdlcoder_intel_examples_root)
cd (example_root)
addpath(genpath('ArrowSoC'));
```

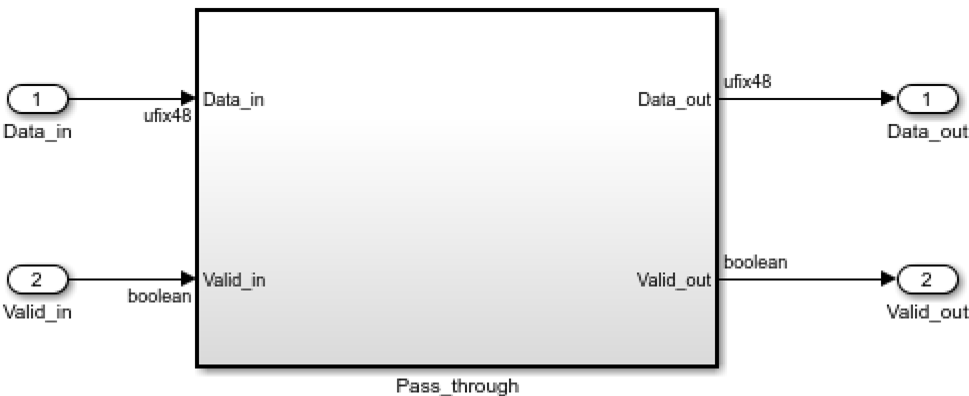
All files that are required for the reference design such as IP core files, qsys file, tcl file, plugin_rd file etc should be added to the MATLAB path, inside **ArrowSoC** folder using the hierarchy shown below. The user generated IP core files should be in **+quartus** folder. plugin_rd.m, tcl files and qsys files should be in the reference design plugin folder, for example, **+qsys_audio_filter_18_1** folder.



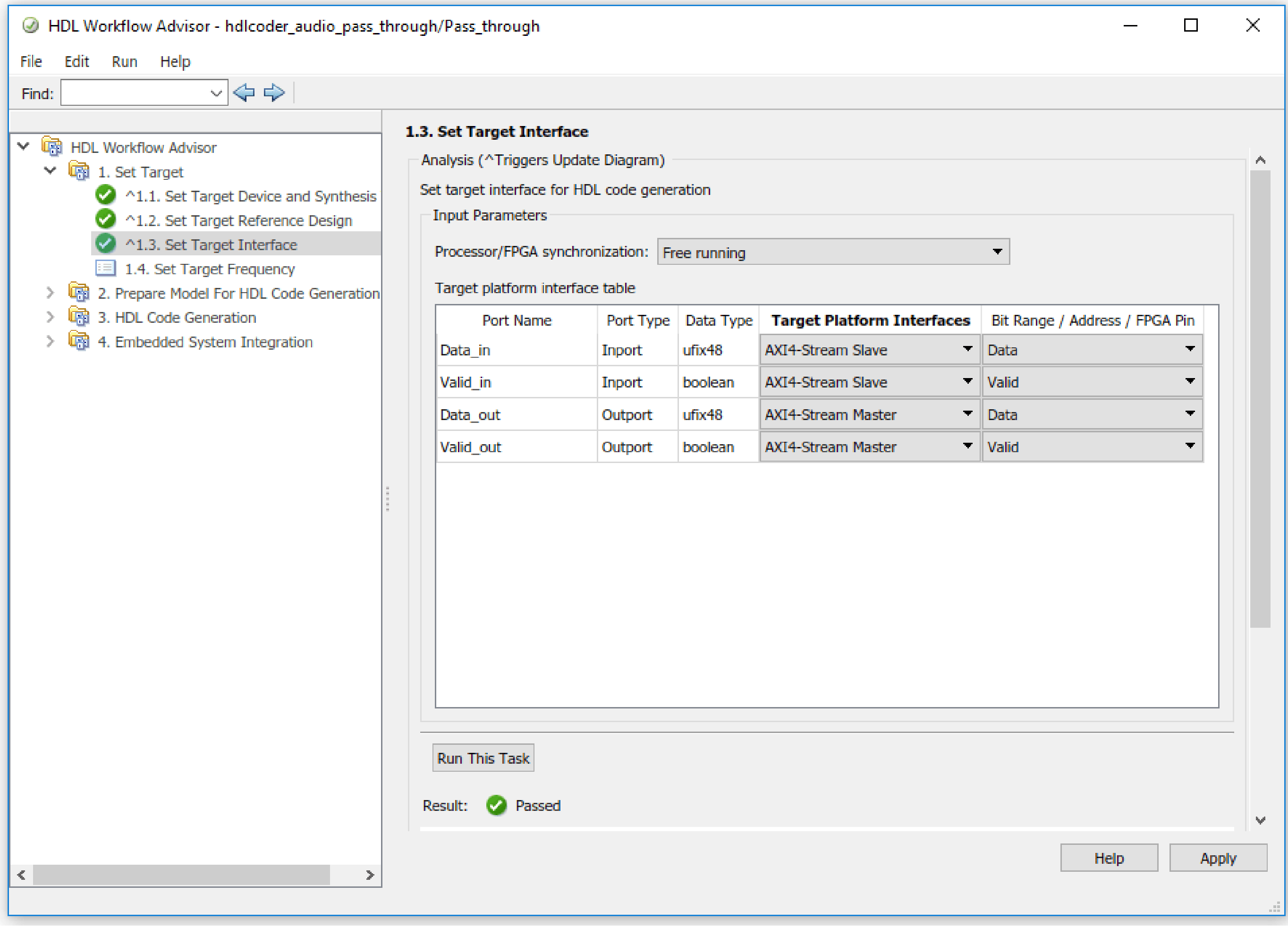
4. Verify the reference design

In order to ensure that the reference design and the interfaces in the reference design work as expected, design a Simulink model which just sends the audio through the Algorithm IP, integrate it with the reference design and test it on Arrow SoC. You should be able to hear the audio loop back.

```
modelname = 'hdlcoder_audio_pass_through';
open_system(modelname);
```



The interfaces in the model should be selected as shown below:



To generate IP core from a model and integrate it with the audio codec reference design, refer to “Running an Audio Filter on Live Audio Input Using Intel Board” on page 40-170.

Define Custom Board and Reference Design for Zynq Workflow

This example shows how to define and register a custom board and reference design in the Zynq® workflow.

Introduction

You can register the Digilent® Zybo Z7-10 Zynq development board and a custom reference design in the HDL Workflow Advisor for the Zynq workflow.

This example uses a Zybo Z7-10 Zynq board, but you can define and register a custom board or a custom reference design for other Zynq platforms.

Requirements

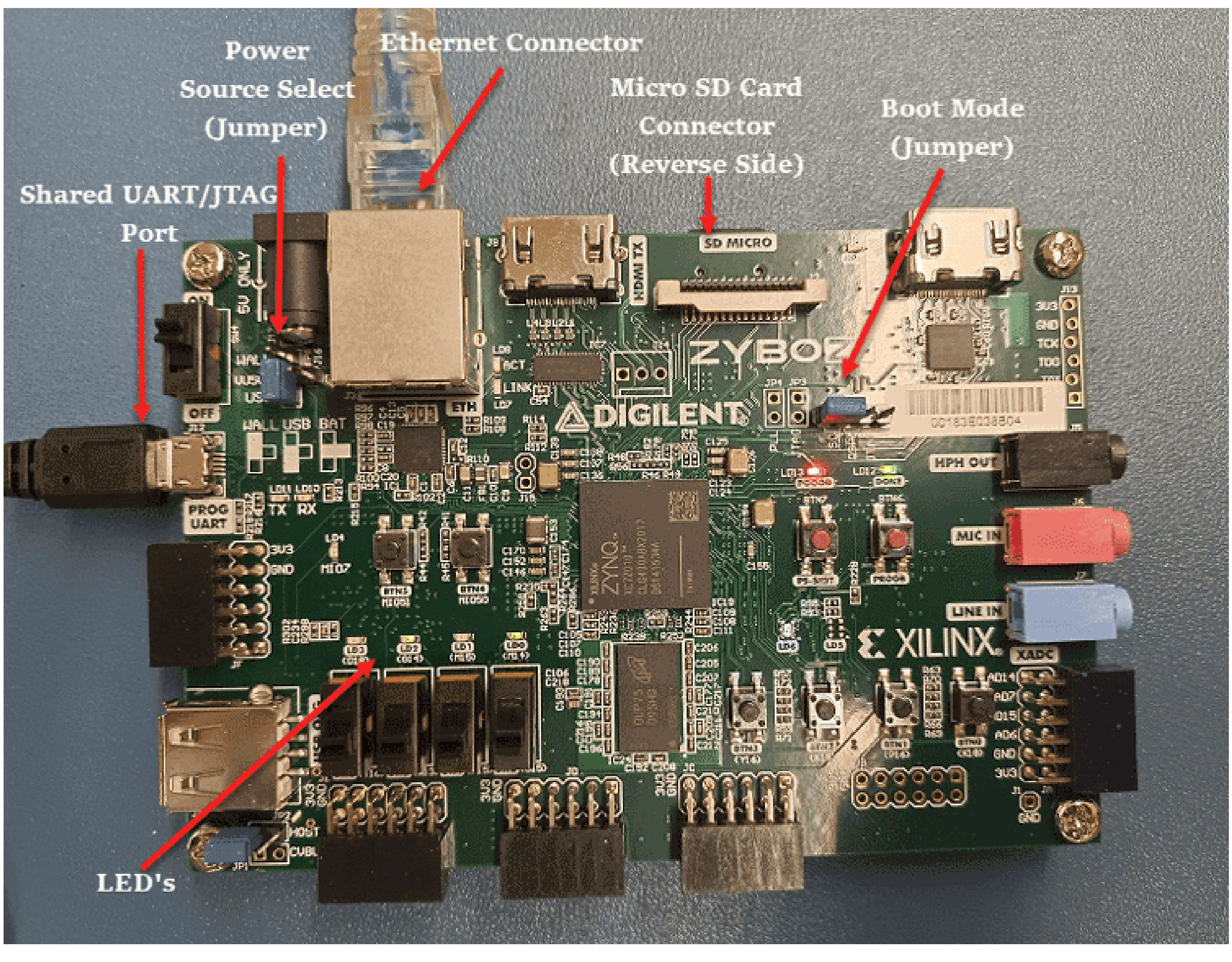
- Xilinx Vivado Design Suite, with the supported version listed in "HDL Language Support and Supported Third-Party Tools and Hardware"
- Digilent® Zybo Z7-10 Zynq™ development board with the accessory kit
- HDL Coder Support Package for Xilinx FPGA and SoC Devices

Note: This example uses Digilent® Zybo Z7-10 Zynq-7000 ARM/FPGA SoC trainer board. This example can be used as reference to create Zybo Z7-20 Custom Board and Reference Design.

Set Up the Zybo board

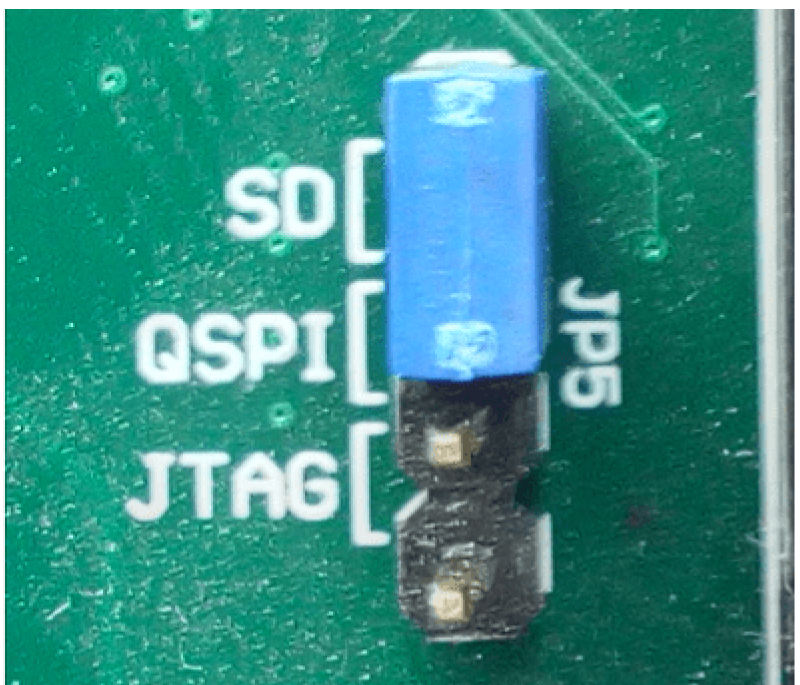
To become familiar with the features of the Zybo board, see the Zybo Z7-10 Board Reference Manual.

1. Set up the Zybo Z7-10 board.

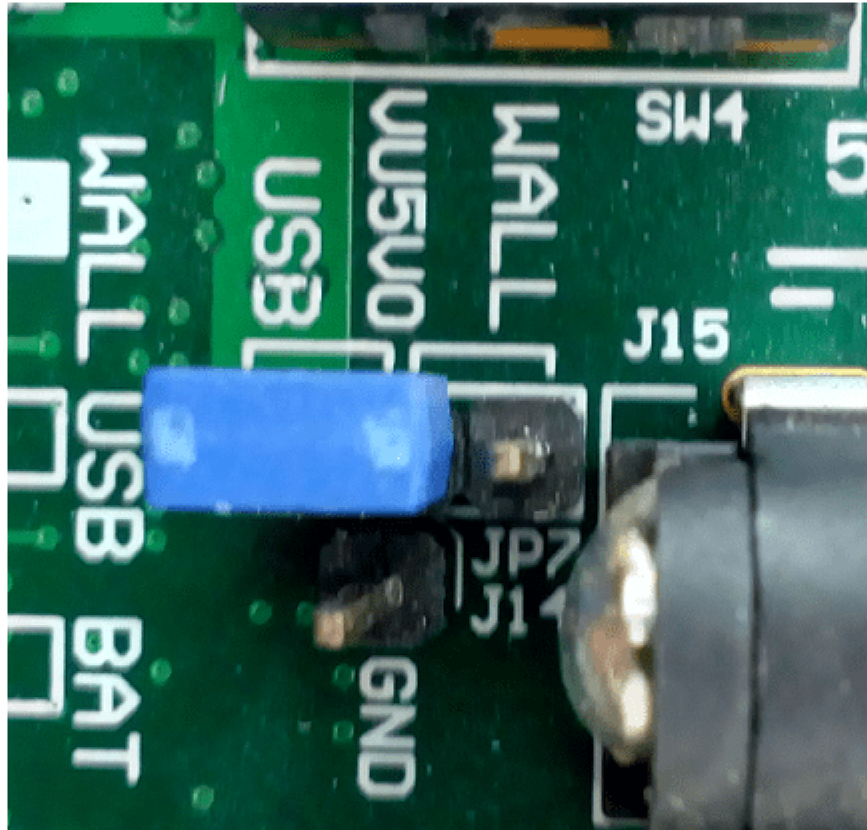


2. Ensure that you have properly installed the USB COM port device drivers on your computer.

3. Configure the `JP5` boot mode jumper to enable the loading of a Zynq Linux image from a microSD card connected to connector `J4`.



4. Configure the `JP7` power source select jumper to use USB as the power source.



5. Connect the shared UART/JTAG USB port on the Zybo board to your computer.

6. Connect the Zybo Z7-10 board to your computer by using an Ethernet cable. The default Zybo IP address is `192.168.1.110`.

7. Download the Zybo Zynq Linux image, extract the Zip archive, and copy the contents to the microSD card. Insert the microSD card in connector `J4`. You can also build the Linux image, by following “Author a Xilinx Zynq Linux Image for a Custom Zynq Board by Using MathWorks Buildroot” on page 39-253.

8. Set up the Xilinx Vivado tool path by using this command:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
```

Use your own Xilinx Vivado installation path when executing the command.

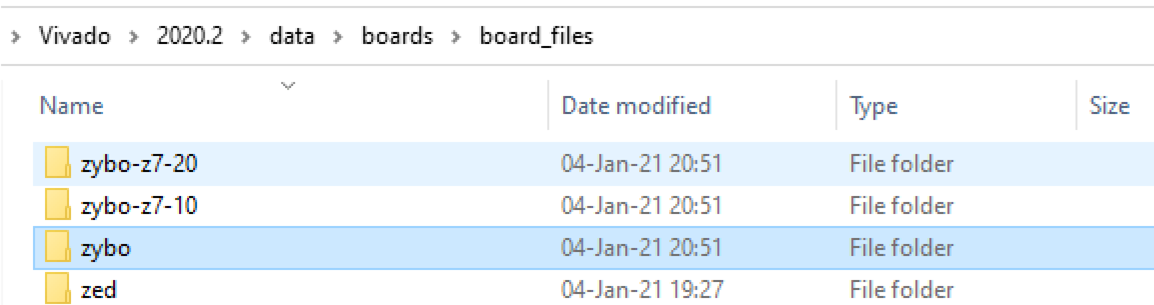
9. Set up the Zynq hardware connection by using this command:

```
h = zynq();
```

Register the Zybo Board in Xilinx Vivado tool

By default, Vivado 2020.2 tool does not have the Zybo board files pre-installed. Download these board files from the Xilinx Board Store GitHub. Unzip the content and navigate to the installation folder of Vivado. Copy the updated Zybo board files to Xilinx Vivado tools.

```
C:\Xilinx\Vivado\2020.2\data\boards\board_files
```

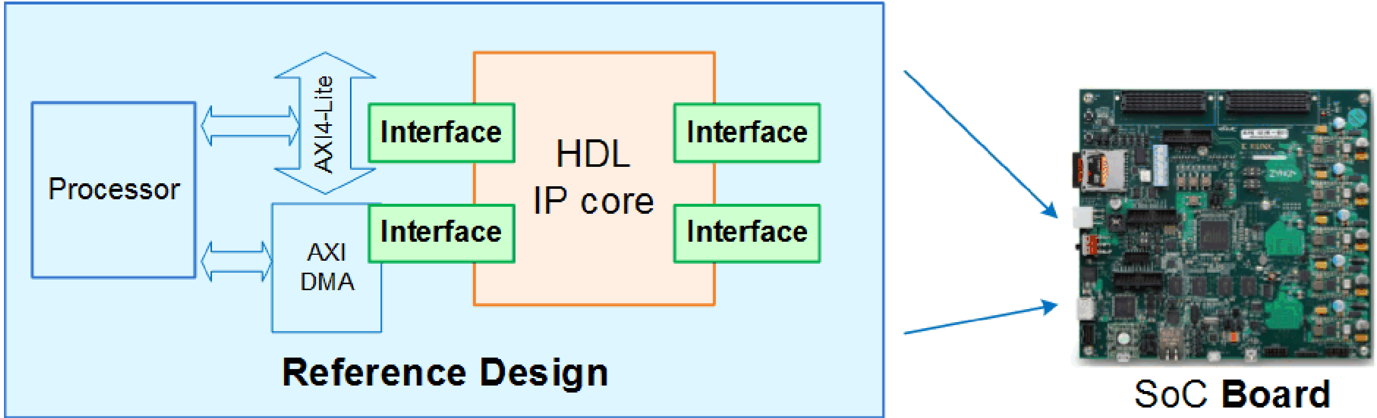


The Zybo board part is added to list of development boards while creating Vivado project specific to board.

Note: If the preceding link is unavailable, get the Zybo board files from the Digilent website.

Create and Export a Custom Reference Design by Using Xilinx Vivado

A reference design captures the complete structure of an SoC design, defining the different components and their interconnections. The HDL Coder SoC workflow generates an IP core that integrates with the reference design, and is then used to program an SoC board. This figure shows the relationship between a reference design, an HDL IP core, and an SoC board.



To create and export a simple reference design by using the Xilinx Vivado IP Integrator environment, follow these steps. For more information about the IP Integrator tool, refer to Xilinx documentation.

1. Create an empty Xilinx Vivado RTL project by using board part **`Zybo Z7-10`** as the default board.

New Project

Default Part

Choose a default Xilinx part or board for your project.

Parts | **Boards**

Reset All Filters

Install/Update Boards

Vendor: All

Name: All

Board Rev: Latest

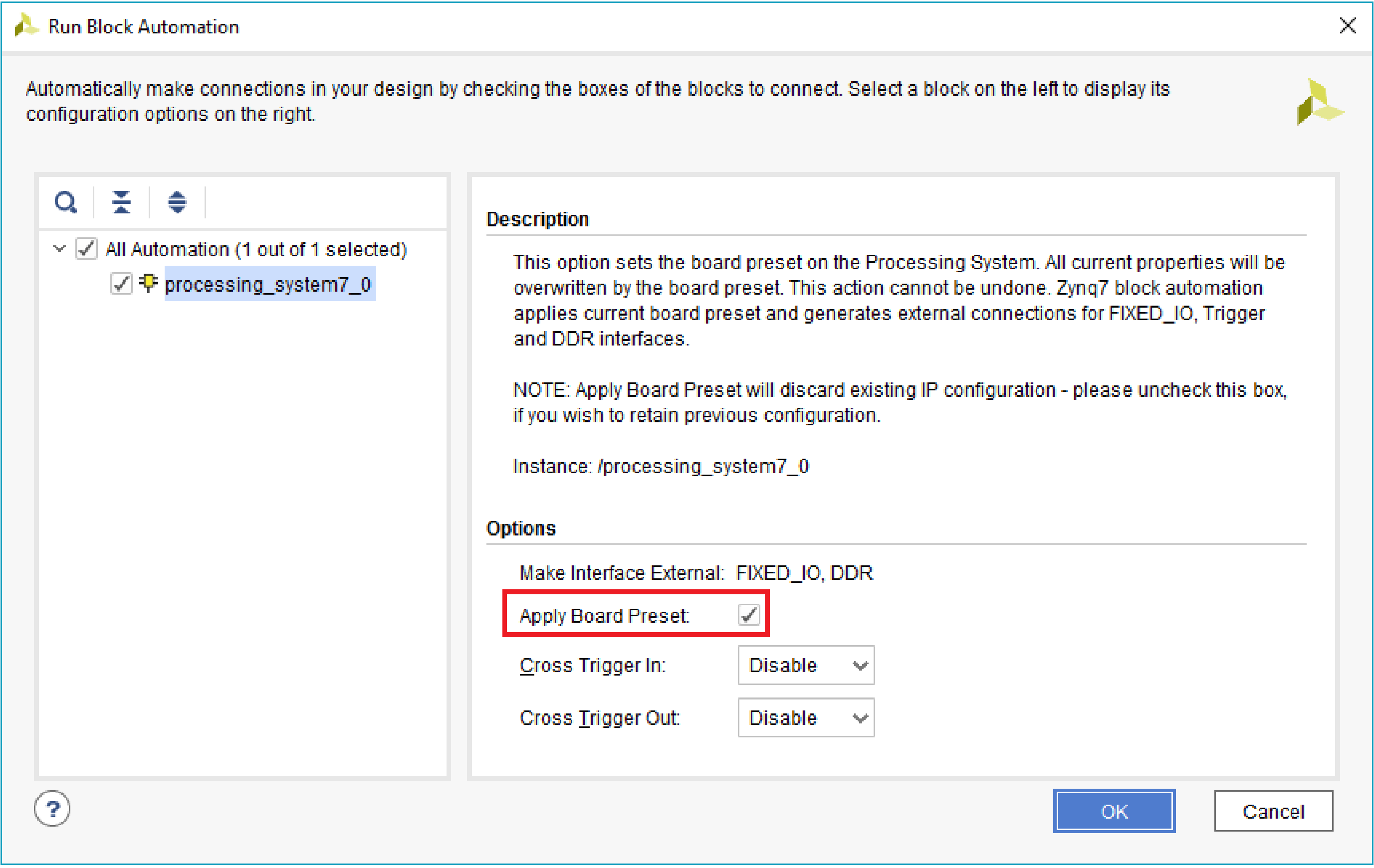
Search:

Display Name	Preview	Vendor	File Version	Part
Zybo Z7-10		digilentinc.com	1.0	xc7z010clg400-1
Zybo Z7-20		digilentinc.com	1.0	xc7z020clg400-1
Zybo		digilentinc.com	2.0	xc7z010clg400-1
ZedBoard Zynq Evaluation and Development Kit Add Companion Card Connections		em.avnet.com	1.4	xc7z020clg484-1
Artix-7 AC701 Evaluation Platform				

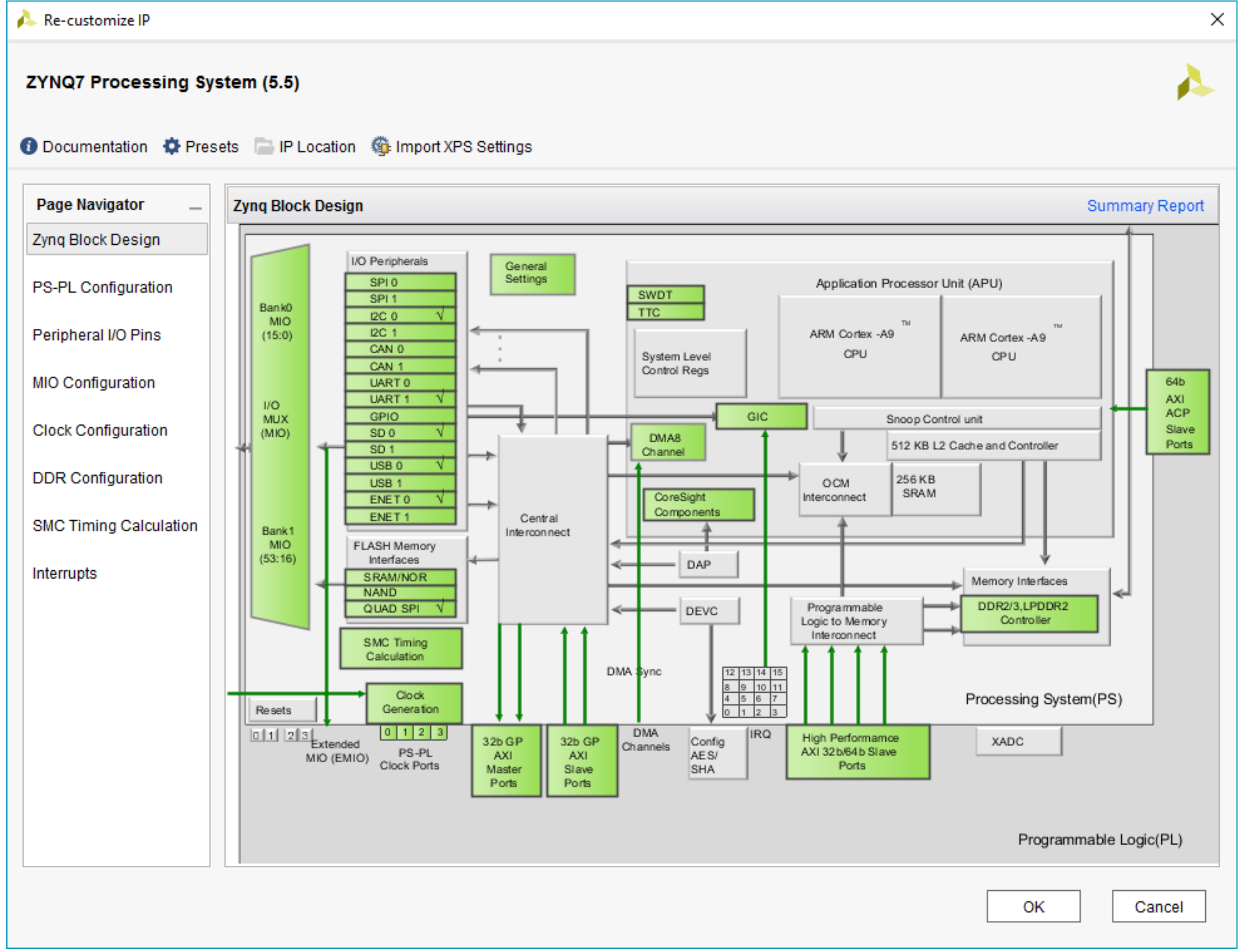
?

< Back Next > Finish Cancel

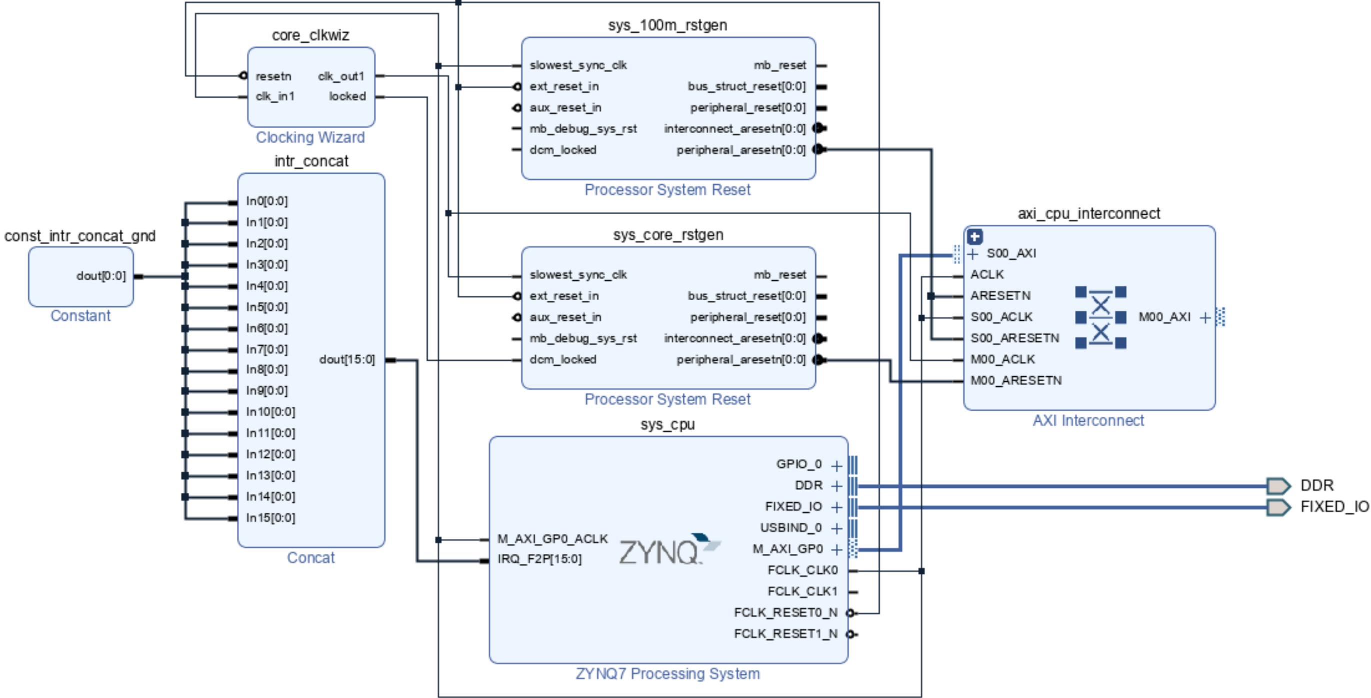
2. Create an empty block design and add the **`ZYNQ7 Processing System`** IP block. Run block automation to set a board preset for Zybo Z7-10, which contains the parameters for ZYNQ7 Processing system IP related to MIO Configuration, Clock configuration, and DDR Configuration.



The MIO peripherals are been marked in accordance with the Zybo board definition as the result of applying a board preset.

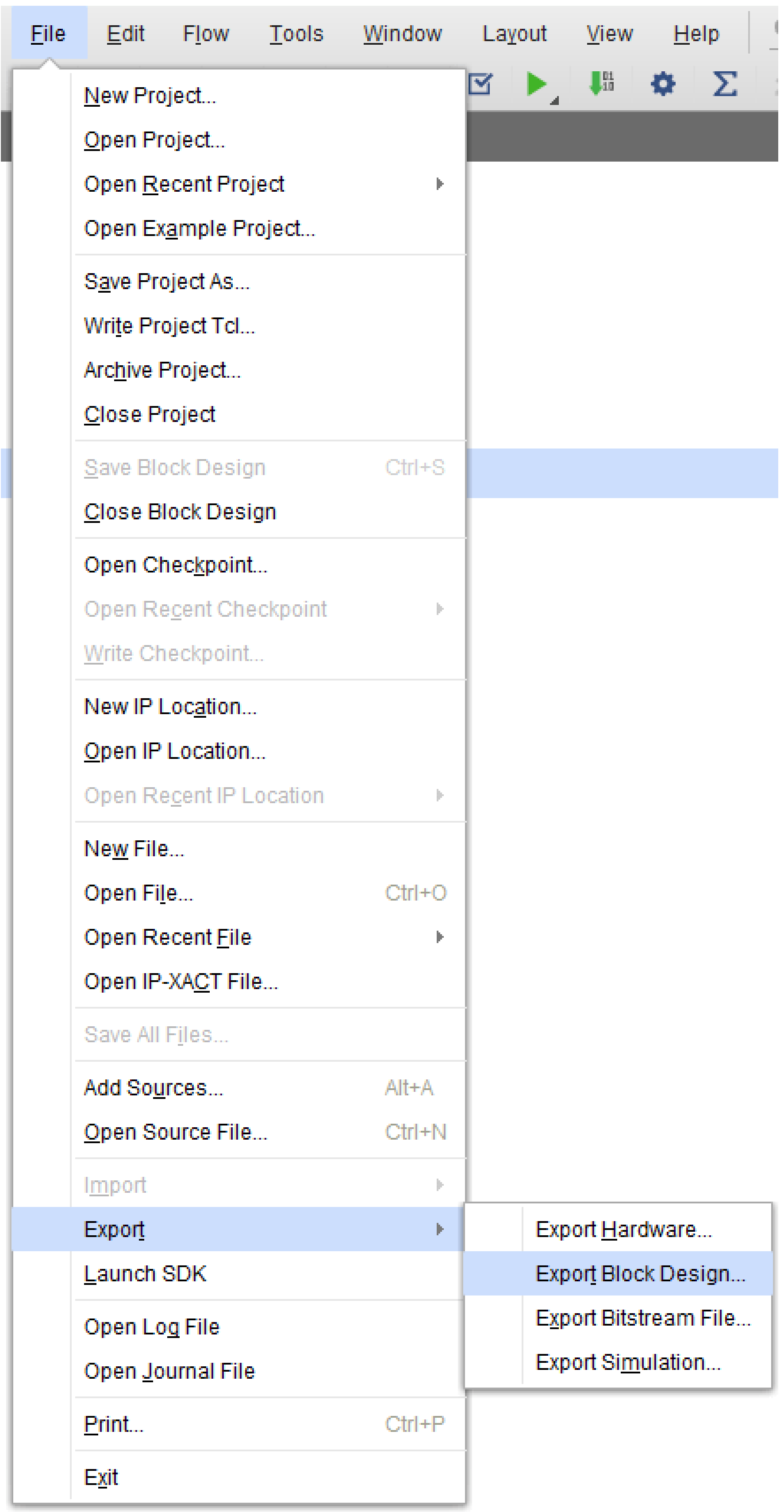


3. Complete the block design.



Notice that the block design does not contain any information about the HDL IP core.

4. Export the completed block design as a Tcl script `design_led.tcl` as shown in the following figure:


File
Edit
Flow
Tools
Window
Layout
View
Help
New Project...
Open Project...
Open Recent Project
Open Example Project...
Save Project As...
Write Project Tcl...
Archive Project...
Close Project
Save Block Design Ctrl+S
Close Block Design
Open Checkpoint...
Open Recent Checkpoint
Write Checkpoint...
New IP Location...
Open IP Location...
Open Recent IP Location
New File...
Open File... Ctrl+O
Open Recent File
Open IP-XACT File...
Save All Files...
Add Sources... Alt+A
Open Source File... Ctrl+N
Import
Export
Launch SDK
Open Log File
Open Journal File
Print... Ctrl+P
Exit
Export Hardware...
Export Block Design...
Export Bitstream File...
Export Simulation...

The exported Tcl script (`design_led.tcl`) constitutes the custom reference design. The Tcl script is used in the HDL Coder SoC workflow to recreate the block design and integrate the generated HDL IP core with the block design in a Xilinx Vivado project.

Register the Zybo Z7-10 Board in HDL Workflow Advisor

Register the Zybo Z7-10 board in HDL Workflow Advisor.

1. Create a board registration file with the name `hdlcoder_board_customization.m` and add it to the MATLAB path.

A board registration file contains a list of board plugins. A board plugin is a MATLAB package folder containing a board definition file and all reference design plugins associated with the board.

This code describes the contents of a board registration file that contains the board plugin `ZyboZ710Registration` to register the Zybo board in HDL Workflow Advisor.

```
function r = hdlcoder_board_customization
% Board plugin registration file
% 1. Any registration file with this name on MATLAB path will be picked up
% 2. Registration file returns a cell array pointing to the location of
%    the board plugins
% 3. Board plugin must be a package folder accessible from MATLAB path,
%    and contains a board definition file

r = { ...
      'ZyboZ710Registration.plugin_board', ...
     };
end
```

2. Create the board definition file.

A board definition file contains information about the SoC board.

This code describes the contents of the Zybo board definition file `plugin_board.m` that resides inside the board plugin `ZyboZ710Registration`.

Information about the FPGA I/O pin locations (`'FPGAPin'`) and standards (`'IOSTANDARD'`) is obtained from the Zybo master constraints file from the Digilent website.

The property `BoardName` defines the name of the Zybo Z7-10 board as `ZYBO Z7-10` in HDL Workflow Advisor.

```
function hB = plugin_board()
% Board definition


% Construct board object
hB = hdlcoder.Board;

hB.BoardName    = 'ZYBO Z7-10';


% FPGA device information
hB.FPGAVendor   = 'Xilinx';
hB.FPGAFamily   = 'Zynq';
```

```
hB.FPGADevice   = 'xc7z010';
hB.FPGAPackage  = 'clg400';
hB.FPGASpeed    = '-1';

% Tool information
hB.SupportedTool = {'Xilinx Vivado'};

% FPGA JTAG chain position
hB.JTAGChainPosition = 2;

%% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

% Custom board external I/O interface
hB.addExternalIOInterface( ...
  'InterfaceID',    'LEDs General Purpose', ...
  'InterfaceType',  'OUT', ...
  'PortName',       'LEDs', ...
  'PortWidth',      4, ...
  'FPGAPin',        {'M14', 'M15', 'G14', 'D18'}, ...
  'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
   'InterfaceID',    'Push Buttons', ...
   'InterfaceType',  'IN', ...
   'PortName',       'PushButtons', ...
   'PortWidth',      4, ...
   'FPGAPin',        {'K18', 'P16', 'K19', 'Y16'}, ...
   'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
   'InterfaceID',    'Slide switches ', ...
   'InterfaceType',  'IN', ...
   'PortName',       'SlideSwitches', ...
   'PortWidth',      4, ...
   'FPGAPin',        {'G15', 'P15', 'W13', 'T16'}, ...
   'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});
```

Register the Custom Reference Design in HDL Workflow Advisor

Register the custom reference design in HDL Workflow Advisor.

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` containing a list of reference design plugins associated with an SoC board.

A reference design plugin is a MATLAB package folder containing the reference design definition file and all files associated with the SoC design project. A reference design registration file must also contain the name of the associated board.

This code describes the contents of a Zybo reference design registration file containing the reference design plugin `ZyboZ710Registration.Vivado2020_2` associated with the board `ZYBO Z7-10`.

```
function [rd, boardName] = hdlcoder_ref_design_customization
```

```
% Reference design plugin registration file
% 1. The registration file with this name inside of a board plugin folder
%    will be picked up
% 2. Any registration file with this name on MATLAB path will also be picked up
% 3. The registration file returns a cell array pointing to the location of
%    the reference design plugins
% 4. The registration file also returns its associated board name
% 5. Reference design plugin must be a package folder accessible from
%    MATLAB path, and contains a reference design definition file

rd = {'ZyboZ710Registration.Vivado2020_2.plugin_rd', ...
     };

boardName = 'ZYBO Z7-10';
end
```

2. Create the reference design definition file.

A reference design definition file defines the interfaces between the custom reference design and the HDL IP core that is generated by the HDL Coder SoC workflow.

This code describes the contents of the Zybo Z7-10 reference design definition file `plugin_rd.m` associated with the board `ZYBO Z7-10` that resides inside the reference design plugin `ZyboZ710Registration.Vivado2020_2`. The property `ReferenceDesignName` defines the name of the reference design as `Demo system` in HDL Workflow Advisor.

```
function hRD = plugin_rd()
% Reference design definition


% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Demo system';
hRD.BoardName = 'ZYBO Z7-10';


% Tool information
hRD.SupportedToolVersion = {'2020.2'};


% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'design_led.tcl', ...
    'VivadoBoardPart','digilentinc.com:zybo-z7-10:part0:1.0');


%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
  'ClockConnection',   'core_clkwiz/clk_out1', ...
  'ResetConnection',   'sys_core_rstgen/peripheral_aresetn');


% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_cpu_interconnect/M00_AXI', ...
```

```
    'BaseAddress',          '0x40010000', ...
    'MasterAddressSpace',   'sys_cpu/Data');
```

```
hRD.DeviceTreeName = 'devicetree_axilite.dtb';
```

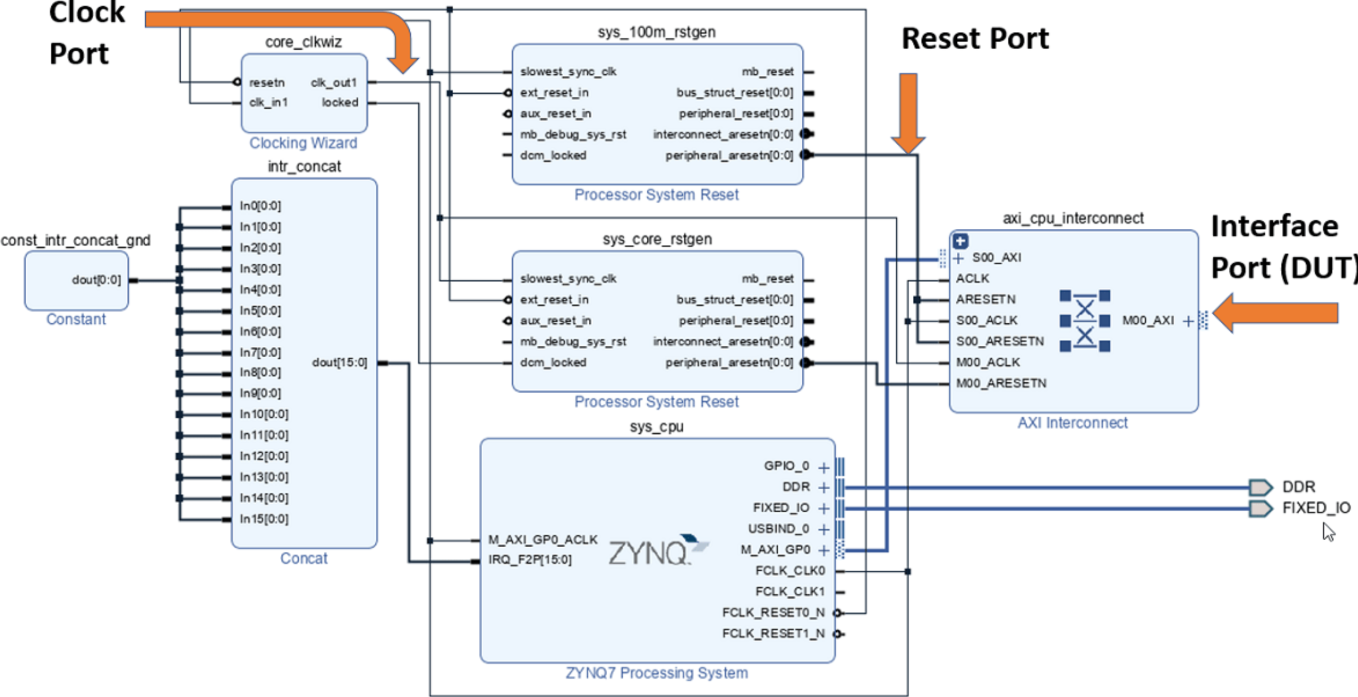
A reference design plugin must also contain the SoC design project files.

The Zybo reference design plugin folder `ZyboZ710Registration.Vivado2020_2` must contain the Tcl script `design_led.tcl` exported previously from the Xilinx Vivado project. The Zybo reference design definition file `plugin_rd.m` identifies the SoC design project file through this statement:

```
hRD.addCustomVivadoDesign('CustomBlockDesignTcl', 'design_led.tcl');
```

`plugin_rd.m` also defines the interface connections between the custom reference design and the HDL IP core indicated by these statements:

```
hRD.addClockInterface( ...
  'ClockConnection',   'core_clkwiz/clk_out1', ...
  'ResetConnection',   'sys_core_rstgen/peripheral_aresetn');
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_cpu_interconnect/M00_AXI', ...
    'BaseAddress',          '0x40010000', ...
    'MasterAddressSpace',   'sys_cpu/Data');
```



Note The `'BaseAddress'` of the AXI4 interface must be a valid address in the `'MasterAddressSpace'`. The address must not create any address conflict with other address-based peripherals in the custom reference design.

Execute the SoC Workflow for the Zybo Board

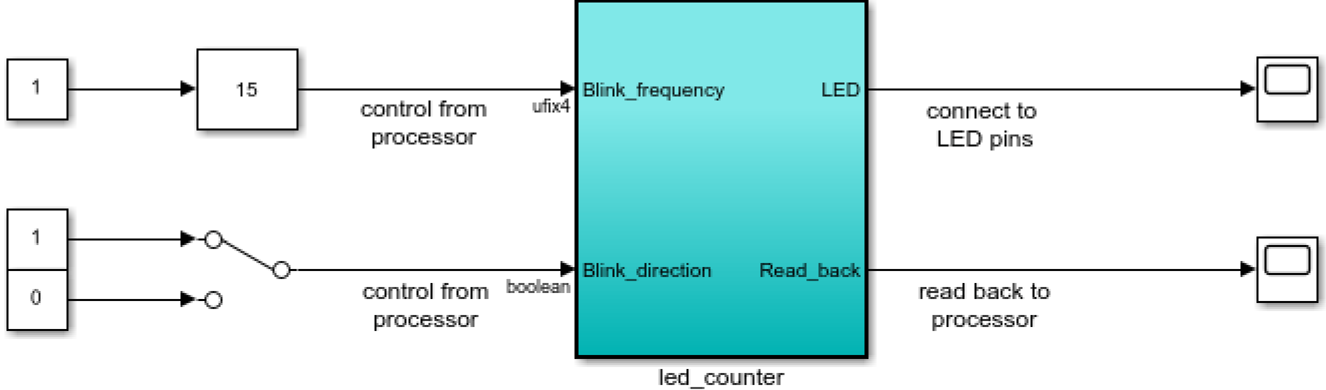
Use the custom board and reference design registration system to generate an HDL IP core that blinks LEDs on the Zybo Z7-10 board. The required files used are located in the ZYBO folder.

1. Add the Zybo board registration file to the MATLAB path.

```
example_root = (hdlcoder_amd_examples_root)
cd (example_root)
addpath(genpath('ZYBO'));
```

2. Open the Simulink model that implements LED blinking.

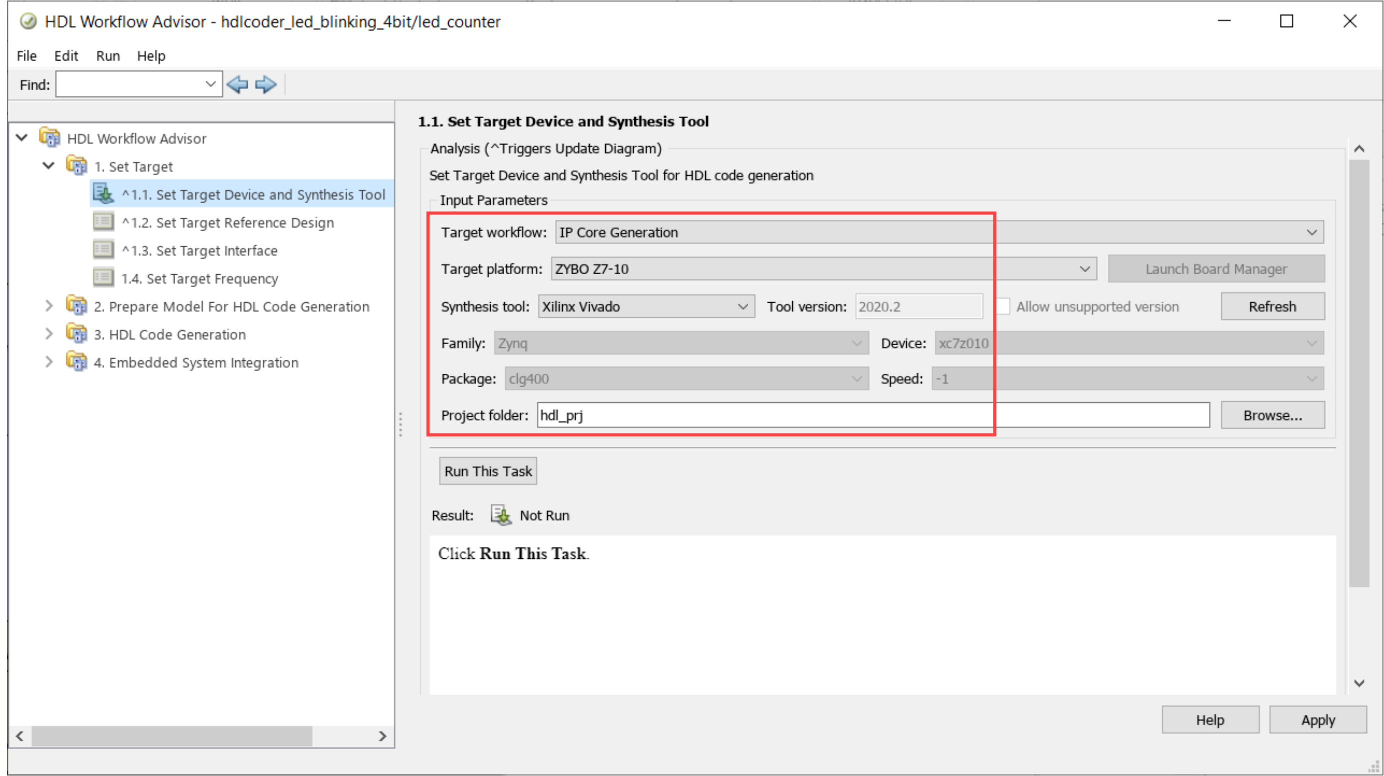
```
open_system('hdlcoder_led_blinking_4bit');
```



Copyright 2014-2023 The MathWorks, Inc.

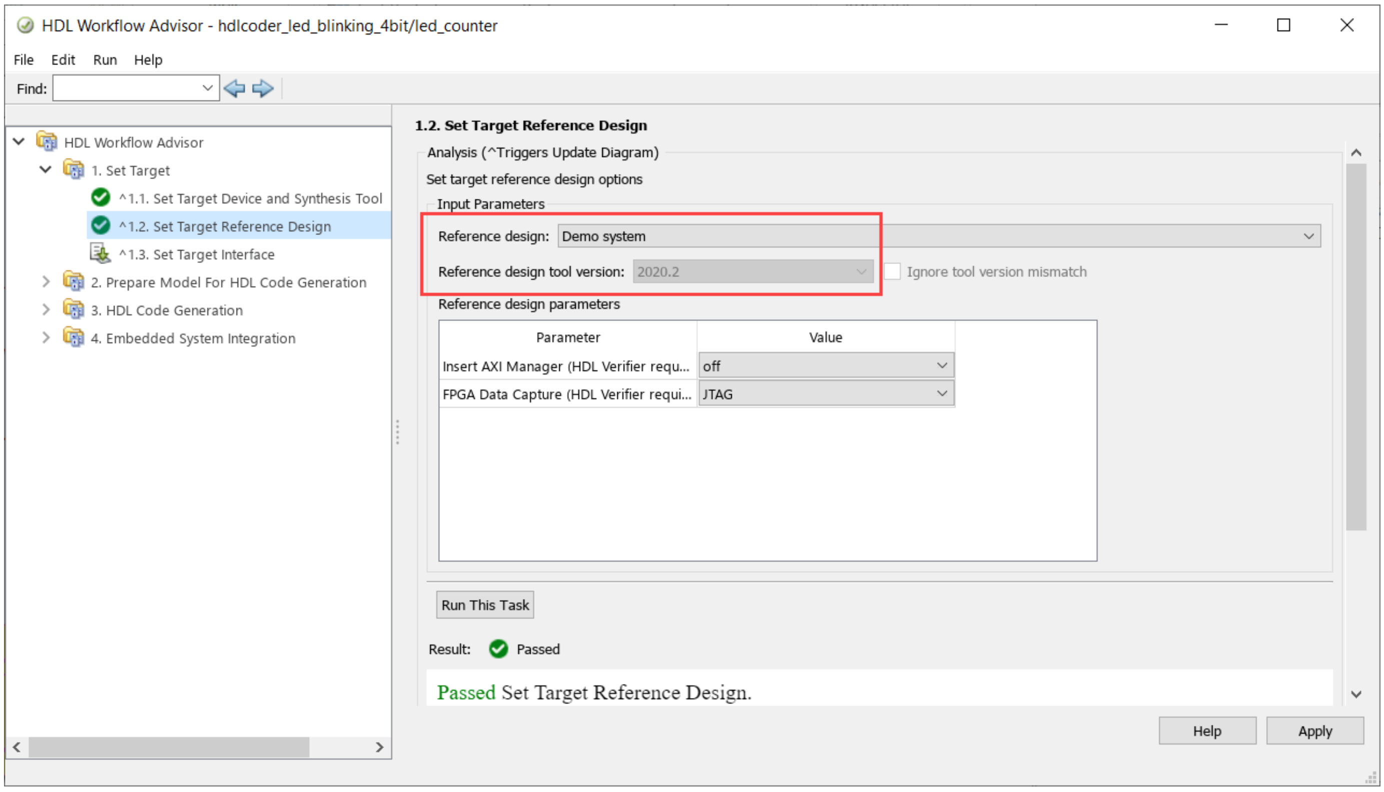
3. Launch HDL Workflow Advisor from the `hdlcoder_led_blinking_4bit/led_counter` subsystem by right-clicking the `led_counter` subsystem, and selecting **HDL Code** > **HDL Workflow Advisor**. Or, you can click the **Launch HDL Workflow Advisor** box in the model.

In the **Set Target** > **Set Target Device and Synthesis Tool** task, select **IP Core Generation** for the **Target workflow**. `ZYBO Z7-10` now appears in the drop-down list **Target Platform**. Select `ZYBO Z7-10` as a **Target Platform**.

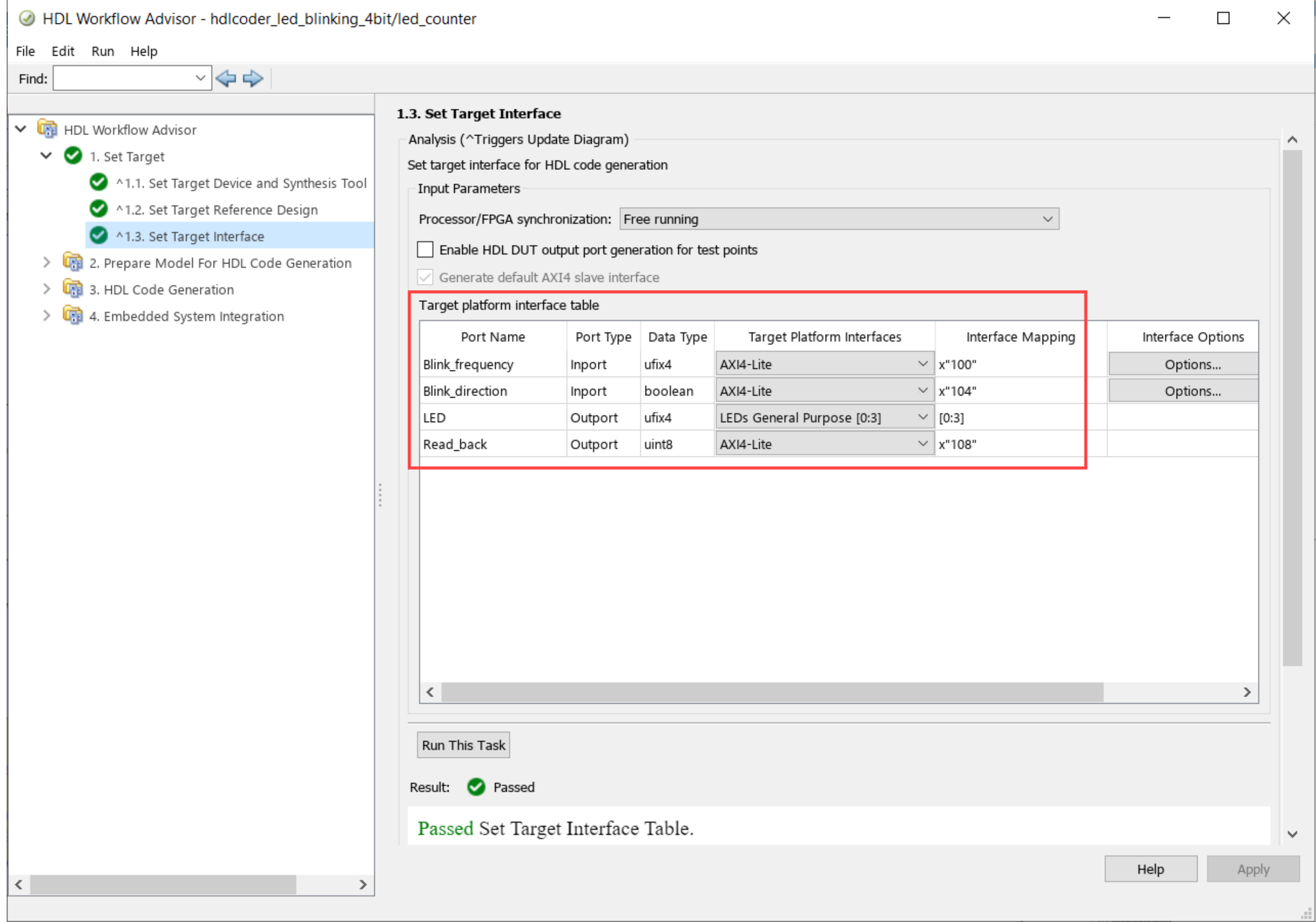


4. Click **Run This Task** to complete the **Set Target Device and Synthesis Tool** task.

5. In the **Set Target** > **Set Target Reference Design** task, the custom reference design `Demo system` now appears in the **Reference design** field. Select `Demo system` and click **Run This Task**.



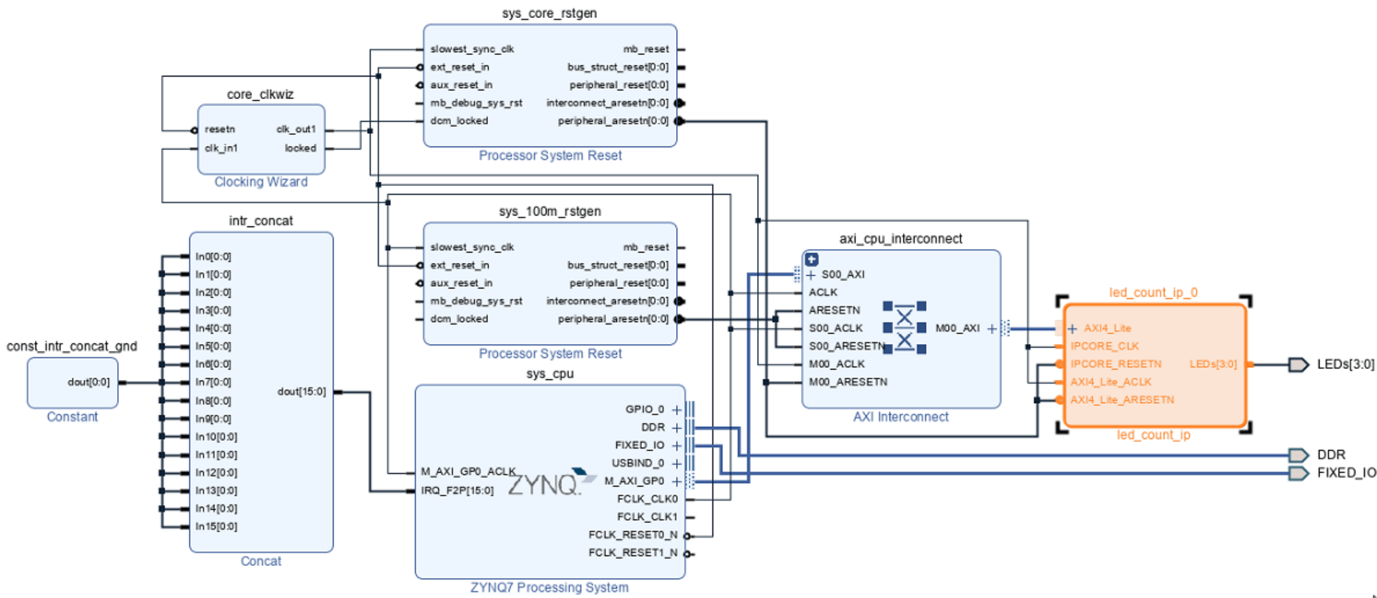
6. In Task 1.3 **Set Target Interface**, select the connections, and then click on **Run This Task**.



7. To generate IP core and view the IP core generation report, follow step 3 and step 4 of **Generate an HDL IP core using the HDL Workflow Advisor** section of “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98.

8. To integrate the IP core in the reference design and create the vivado project, follow step 1 of **Integrate the IP core with the Xilinx Vivado environment** section of “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98.

9. After completing the **Create Project** task under **Embedded System Integration**, examine the Xilinx Vivado project created by the SoC workflow. This figure shows the block design of the SoC project where you have highlighted the HDL IP Core. Compare this block design with the previous block design used to export the custom reference design for a deeper understanding of the relationship between a custom reference design and an HDL IP Core.



10. Follow the steps 2, 3 and 4 of **Integrate the IP core with the Xilinx Vivado environment** section of “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98 example to generate the software interface model, generate the FPGA bitstream, and program the target device respectively.

11. After loading the bitstream, the LEDs on the Zybo Z7-10 board now start blinking. You can control the LED blink frequency and direction by executing the software interface model on the Zynq ARM processor. Refer to **Generate a software interface model** section of “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98 example to control the LED blink frequency and direction from the generated software interface model.

Define Custom Board and Reference Design for Intel SoC Workflow

This example shows how to define and register a custom board and reference design in the HDL Coder™ Intel® SoC workflow.

Introduction

Using this example, you will be able to register the Terasic DE1-SoC development kit and a custom reference design in the HDL Workflow Advisor for the Intel SoC workflow.

This example uses Terasic DE-1 SoC, but in the same way, you can define and register a custom board or a custom reference design for other Intel SoC devices.

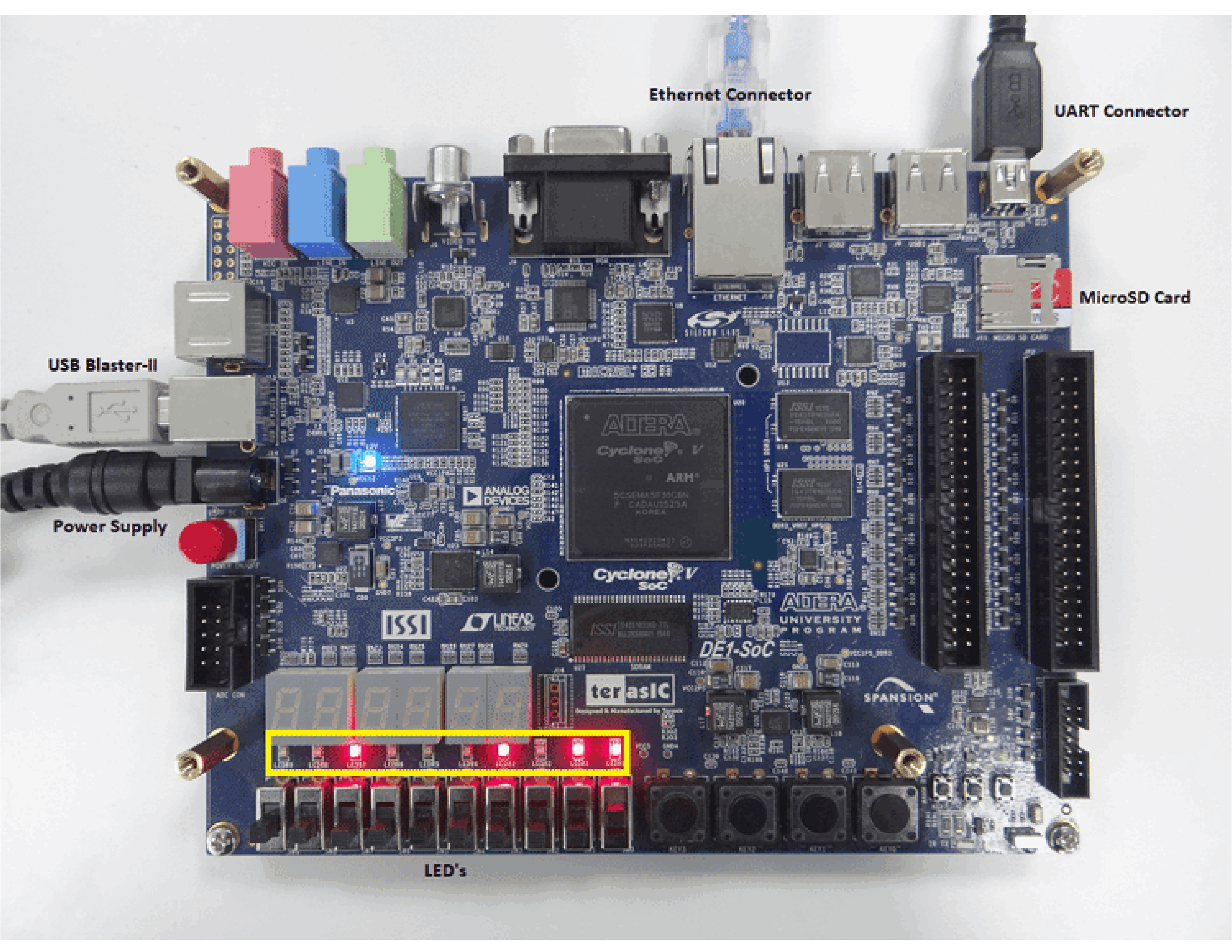
Requirements

1. Intel Quartus® Prime, with supported version listed in the “HDL Language Support and Supported Third-Party Tools and Hardware”
2. Intel SoC Embedded Design Suite
3. Terasic DE1-SoC development Kit
4. HDL Coder Support Package for Intel FPGA and SoC Devices
5. Embedded Coder® Support Package for Intel SoC Devices

Set up Intel SoC hardware and tools

1. Understand the features available on the Terasic DE1-SoC by reading the board reference manual.

2. Set up the Terasic DE1-SoC as shown in the following figure:



3. Ensure that you have properly installed the USB COM port device drivers on your computer.

4. Connect the UART and USB blaster port on the Terasic DE1-SoC to your computer.

5. Connect the Terasic DE1-SoC to your computer using an Ethernet cable. The default Terasic DE1-SoC IP address is `192.168.1.101`.

6. Download the Terasic DE1-SoC Linux image file, extract the GZ archive, and then write the raw disc image file to the microSD card. Insert the microSD card in connector `J11`.

7. Set up the Intel Quartus tool path by using the following command:

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', 'ToolPath', 'C:\intelFPGA\20.1.1\quartus\b
```

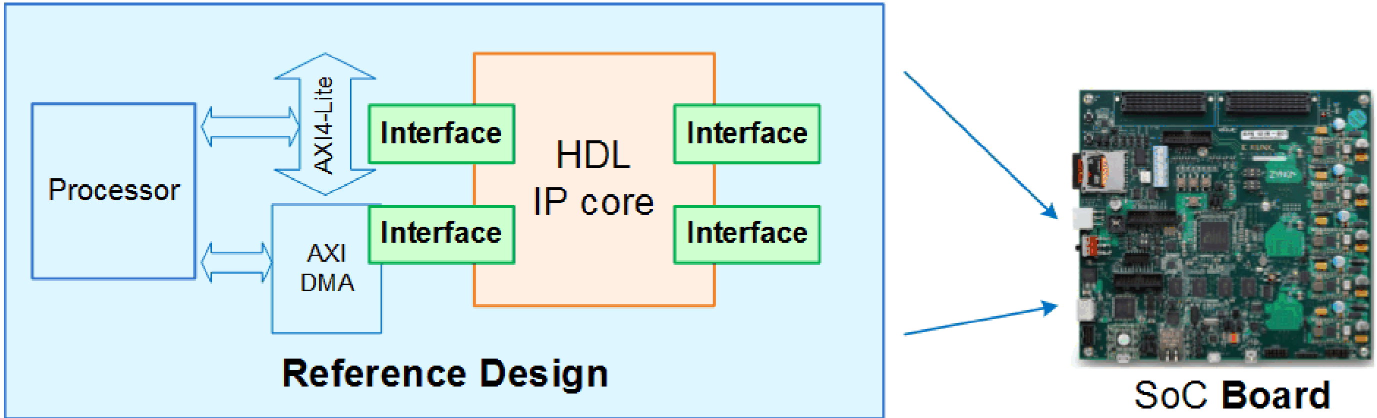
Use your own Intel Quartus installation path when executing the command.

8. Set up the Terasic DE1-SoC hardware connection by using the following command:

```
h = alterasoc('192.168.1.101','root','cyclonevsoc');
```

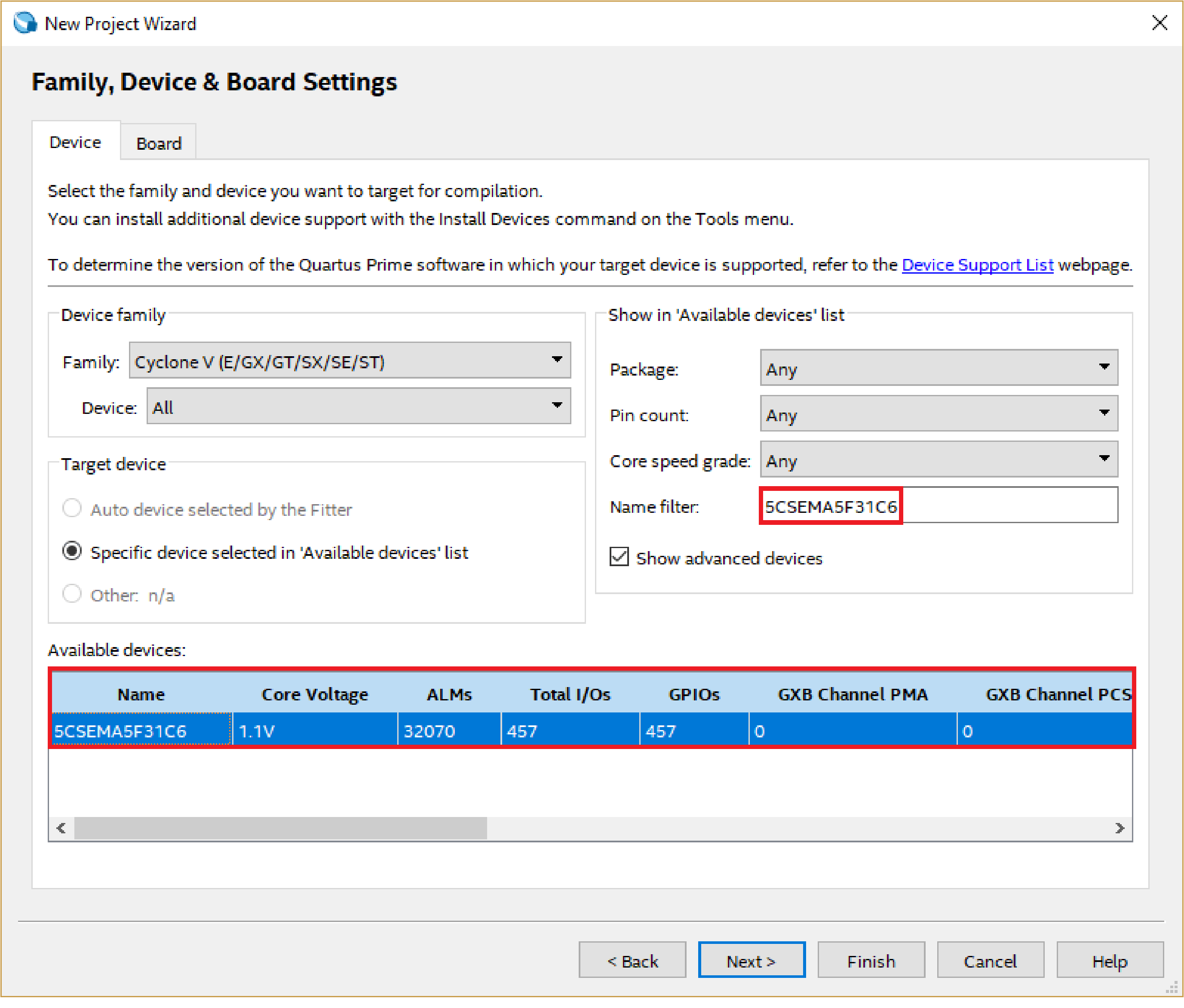
Reference Design creation using Intel Quartus Prime

A reference design captures the complete structure of an SoC design, defining the different components and their interconnections. The HDL Coder SoC workflow generates an IP core that integrates with the reference design, and is then used to program an SoC board. The following figure describes the relationship between a reference design, an HDL IP core and an SoC board

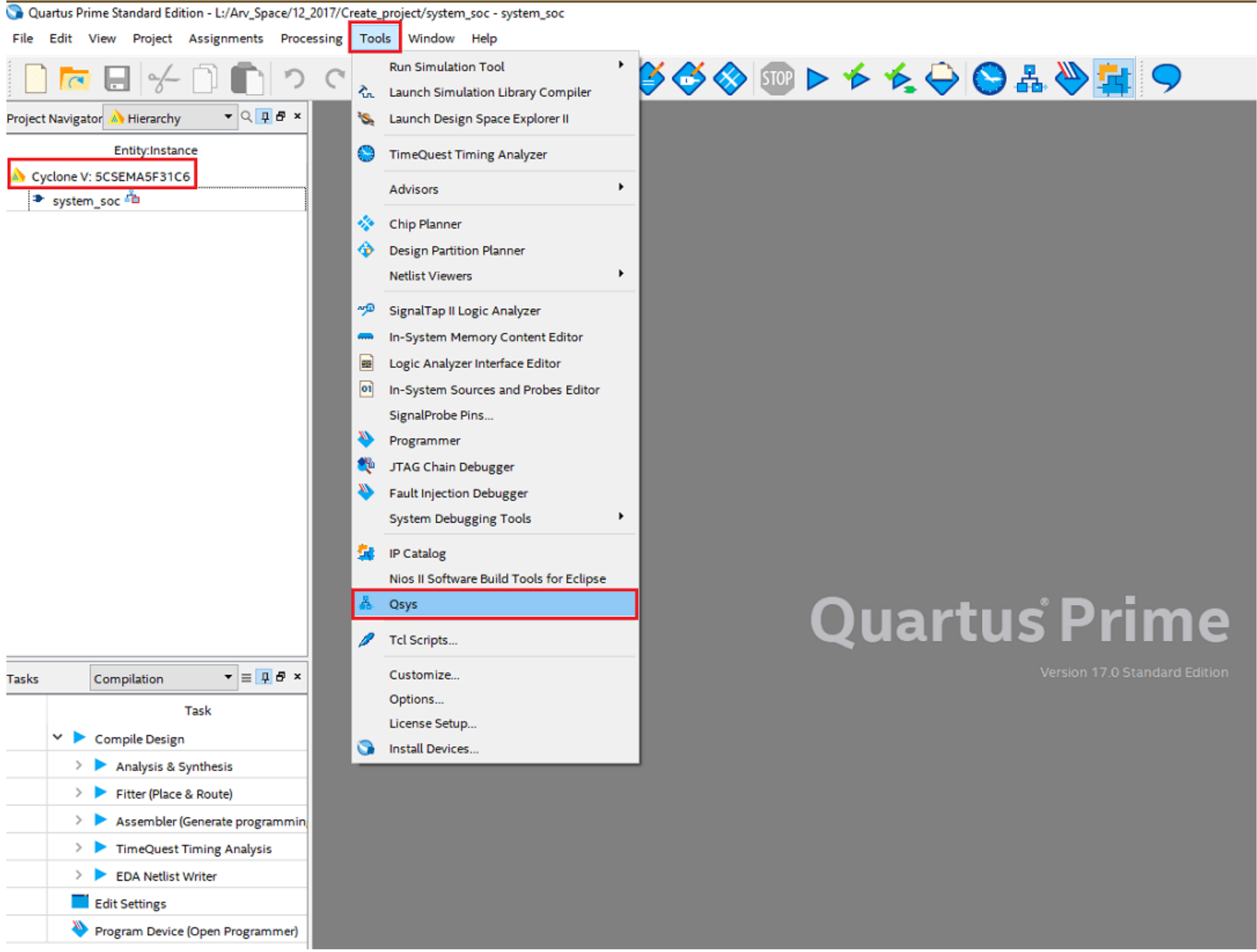


In this section, we outline the basic steps necessary to create and export a simple reference design using the Intel Quartus and QSys environment. For more information about the QSys system integration tool, refer to Altera®/Intel documentation.

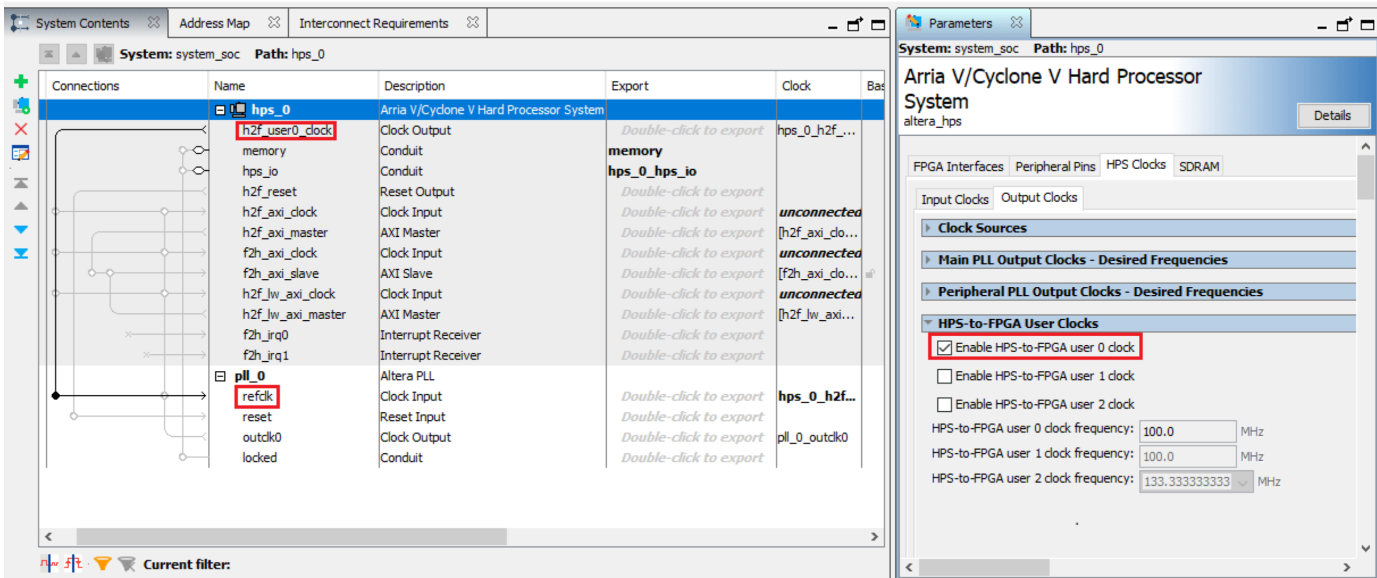
1. Create an empty Quartus project using the New project wizard with device part number as shown in the following figure



2. Initialize the Qsys in Quartus by navigating **Tools --> Qsys** as shown in the following figure

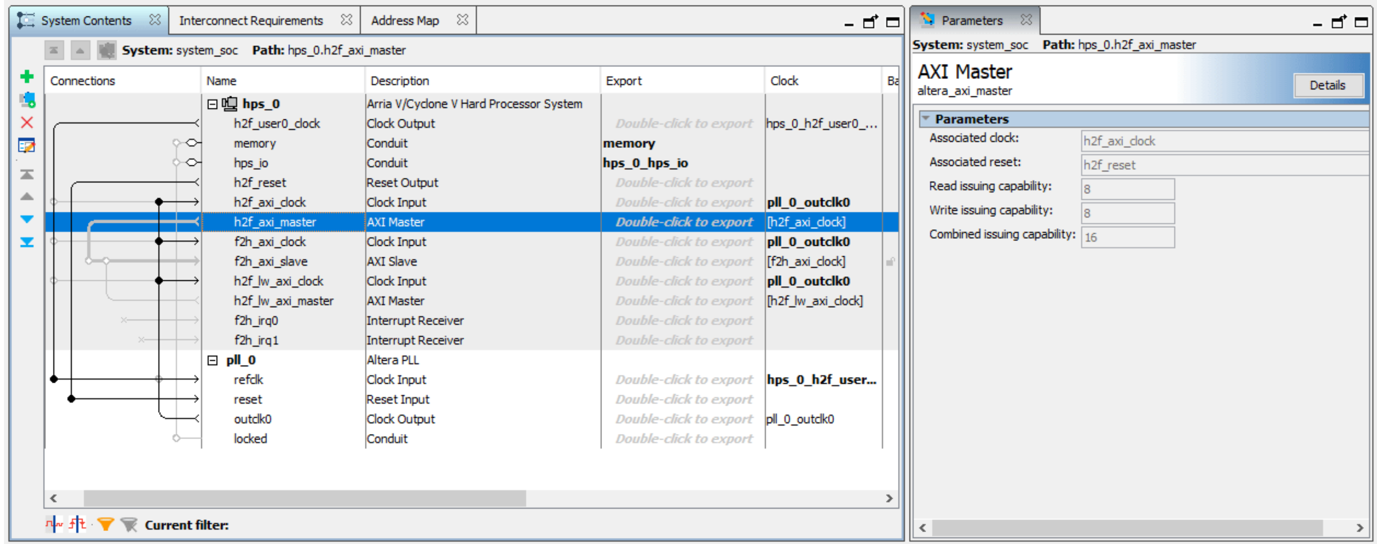


3. Select **Cyclone-V Hard Processor System(HPS)** & **Altera PLL** IP's from IP catalog to the created Qsys project. Enable HPS-to-FPGA user 0 clock (`h2f_user0_clock`) and connect that to `refclk` of Altera PLL as shown in the following figure



complete the other settings required for Hard Processor System such as Peripheral pin set and mode settings.

4. keep `h2f_axi_master` port connection open in order to connect to DUT IP during the process of workflow IP integration. Complete the rest of the connections between Altera PLL IP and HPS IP as shown in the following figure



5. Save the Qsys file. This file will be used while you create reference design plugin.

Register the DE1-SoC board in HDL Workflow Advisor

In this section, we outline the steps necessary to register the Terasic DE1-SoC development kit in HDL Workflow Advisor.

1. Create a board registration file with the name `hdlcoder_board_customization.m` and add it to the MATLAB® path.

A board registration file contains a list of board plug-ins. A board plugin is a MATLAB package folder containing a board definition file and all reference design plug-ins associated with the board.

The following code describes the contents of a board registration file that contains the board plugin `DE1SoCRegistration` to register the Terasic DE1-SoC development kit in HDL Workflow Advisor.

```
function r = hdlcoder_board_customization
% Board plugin registration file
% 1. Any registration file with this name on MATLAB path will be picked up
% 2. Registration file returns a cell array pointing to the location of
%    the board plugin
% 3. Board plugin must be a package folder accessible from MATLAB path,
%    and contains a board definition file

r = { ...
    'DE1SoCRegistration.plugin_board', ...
    };
end
```

2. Create the board definition file.

A board definition file contains information about the SoC board.

The following code describes the contents of the DE1-SoC board definition file `plugin_board.m` that resides inside the board plugin `DE1SoCRegistration`.

Information about the FPGA I/O pin locations (`'FPGAPin'`) and standards (`'IOSTANDARD'`) is obtained from the Pin Planner of Intel Quartus-II.

The property `BoardName` defines the name of the DE-1 SoC board as `Terasic DE1-SoC development Kit` in HDL Workflow Advisor.

```
function hB = plugin_board()
% Board definition


% Construct board object
hB = hdlcoder.Board;

hB.BoardName    = 'Terasic DE1-SoC development Kit';


% FPGA device information
hB.FPGAVendor   = 'Altera';
hB.FPGAFamily   = 'Cyclone V';
hB.FPGADevice   = '5CSEMA5F31C6';
hB.FPGAPackage  = '';
hB.FPGASpeed    = '';


% Tool information
hB.SupportedTool = {'Altera QUARTUS II'};


% FPGA JTAG chain position
hB.JTAGChainPosition = 2;
```

```
%% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IO_STANDARD "2.5V"'});


% Custom board external I/O interface
hB.addExternalIOInterface( ...
  'InterfaceID',    'LEDs General Purpose', ...
  'InterfaceType',  'OUT', ...
  'PortName',       'GPLED', ...
  'PortWidth',      10, ...
  'FPGAPin',        {'V16', 'W16', 'V17', 'V18', 'W17', 'W19', 'Y19', 'W20', 'W21', 'Y21'}, ...
  'IOPadConstraint', {'IO_STANDARD "3.3-V LVTTL"'});

hB.addExternalIOInterface( ...
   'InterfaceID',    'Switches', ...
   'InterfaceType',  'IN', ...
   'PortName',       'SW', ...
   'PortWidth',      10, ...
   'FPGAPin',        {'AB12', 'AC12', 'AF9', 'AF10','AD11', 'AD12', 'AE11', 'AC9', 'AD10', 'AE12
   'IOPadConstraint', {'IO_STANDARD "3.3-V LVTTL"'});

hB.addExternalIOInterface( ...
   'InterfaceID',    'Push Buttons', ...
   'InterfaceType',  'IN', ...
   'PortName',       'KEY', ...
   'PortWidth',      4, ...
   'FPGAPin',        {'AA14', 'AA15', 'W15', 'Y16'}, ...
   'IOPadConstraint', {'IO_STANDARD "3.3-V LVTTL"'});
```

Register the custom reference design in HDL Workflow Advisor

In this section, we outline the steps necessary to register the custom reference design in HDL Workflow Advisor.

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` containing a list of reference design plugins associated with an SoC board.

A reference design plugin is a MATLAB package folder containing the reference design definition file and all files associated with the SoC design project. A reference design registration file must also contain the name of the associated board.

The following code describes the contents of a DE1-SoC reference design registration file containing the reference design plugin `DE1SoCRegistration.qsys_base_170` associated with the board `Terasic DE1-SoC development Kit`.

```
function [rd, boardName] = hdlcoder_ref_design_customization
% Reference design plugin registration file
% 1. The registration file with this name inside of a board plugin folder
%    will be picked up
% 2. Any registration file with this name on MATLAB path will also be picked up
% 3. The registration file returns a cell array pointing to the location of
%    the reference design plugins
% 4. The registration file also returns its associated board name
```

```
% 5. Reference design plugin must be a package folder accessible from
%    MATLAB path, and contains a reference design definition file

rd = {'DE1SoCRegistration.qsys_base_170.plugin_rd', ...
     };

boardName = 'Terasic DE1-SoC development Kit';
end
```

2. Create the reference design definition file.

A reference design definition file defines the interfaces between the custom reference design and the HDL IP core that will be generated by the HDL Coder SoC workflow.

The following code describes the contents of the DE1-SoC reference design definition file `plugin_rd.m` associated with the board `Terasic DE1-SoC development Kit` that resides inside the reference design plugin `DE1SoCRegistration.qsys_base_170`. The property `ReferenceDesignName` defines the name of the reference design as `Demo system` in HDL Workflow Advisor.

```
function hRD = plugin_rd()
% Reference design definition


% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Altera QUARTUS II');

hRD.ReferenceDesignName = 'Demo system)';
hRD.BoardName = 'Terasic DE1-SoC development Kit';


% Tool information
hRD.SupportedToolVersion = {'17.0','17.1'};


%% Add custom design files
% add custom Qsys design
hRD.addCustomQsysDesign( ...
    'CustomQsysPrjFile', 'system_soc.qsys');


%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',     'pll_0.outclk0', ...
    'ResetConnection',     'hps_0.h2f_reset',...
    'DefaultFrequencyMHz', 50,...
    'MinFrequencyMHz',     5,...
    'MaxFrequencyMHz',     500,...
    'ClockModuleInstance', 'pll_0',...
    'ClockNumber',         0);


% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'hps_0.h2f_axi_master', ...
    'BaseAddress',         '0x0000');
```

The DE1-SoC reference design plugin folder `DE1SoCRegistration.qsys_base_170` must contain the Qsys file `system_soc.qsys` saved previously from the Intel Quartus Prime project. The DE1-SoC reference design definition file `plugin_rd.m` identifies the SoC design project file via the following statement:

```
hRD.addCustomQsysDesign('CustomQsysPrjFile', 'system_soc.qsys');
```

In addition to the SoC design project files, `plugin_rd.m` also defines the interface connections between the custom reference design and the HDL IP core indicated in the following figure via the statements:

```
hRD.addClockInterface( ...
    'ClockConnection',     'pll_0.outclk0', ...
    'ResetConnection',     'hps_0.h2f_reset',...
    'DefaultFrequencyMHz', 50,...
    'MinFrequencyMHz',     5,...
    'MaxFrequencyMHz',     500,...
    'ClockModuleInstance', 'pll_0',...
    'ClockNumber',         0);
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'hps_0.h2f_axi_master', ...
    'BaseAddress',         '0x0000');
```

Execute the SoC workflow for the Terasic DE1-SoC

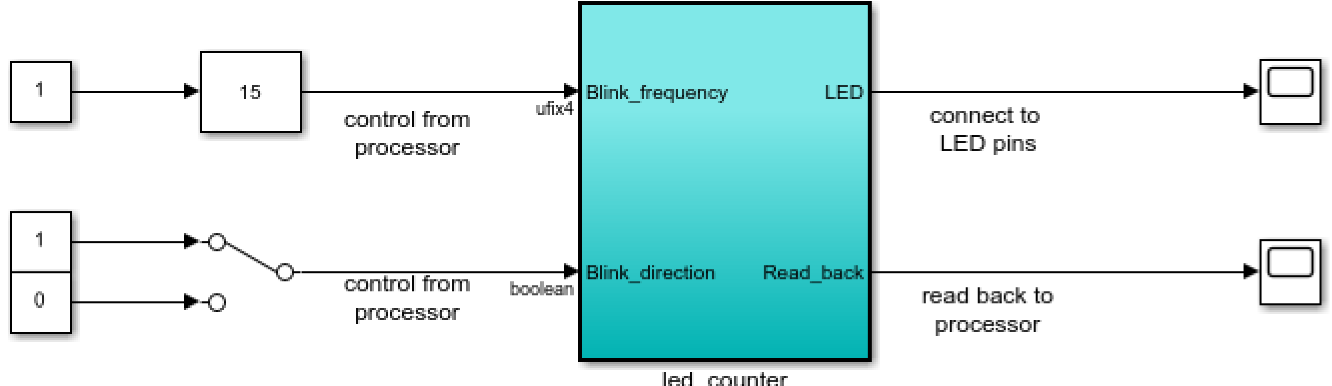
The preceding sections discussed the steps to define and register the Terasic DE1-SoC and a custom reference design in the HDL Workflow Advisor for the SoC workflow. In this section, we use the custom board and reference design registration system to generate an HDL IP core that blinks LEDs on the Terasic DE1-SoC.

1. Add the Terasic DE1-SoC registration file to the MATLAB path using following commands.

```
example_root = (hdlcoder_intel_examples_root)
cd (example_root)
addpath(genpath('DE1SOC'));
```

2. Open the Simulink model that implements LED blinking using the command,

```
open_system('hdlcoder_led_blinking');
```



Generate an HDL IP core using the HDL Workflow Advisor

1. Using the IP Core Generation workflow in the HDL Workflow Advisor enables you to automatically generate a sharable and reusable IP core module from a Simulink model. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic connecting the IP core to the embedded processor. HDL Coder packages all the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Intel Qsys environment.

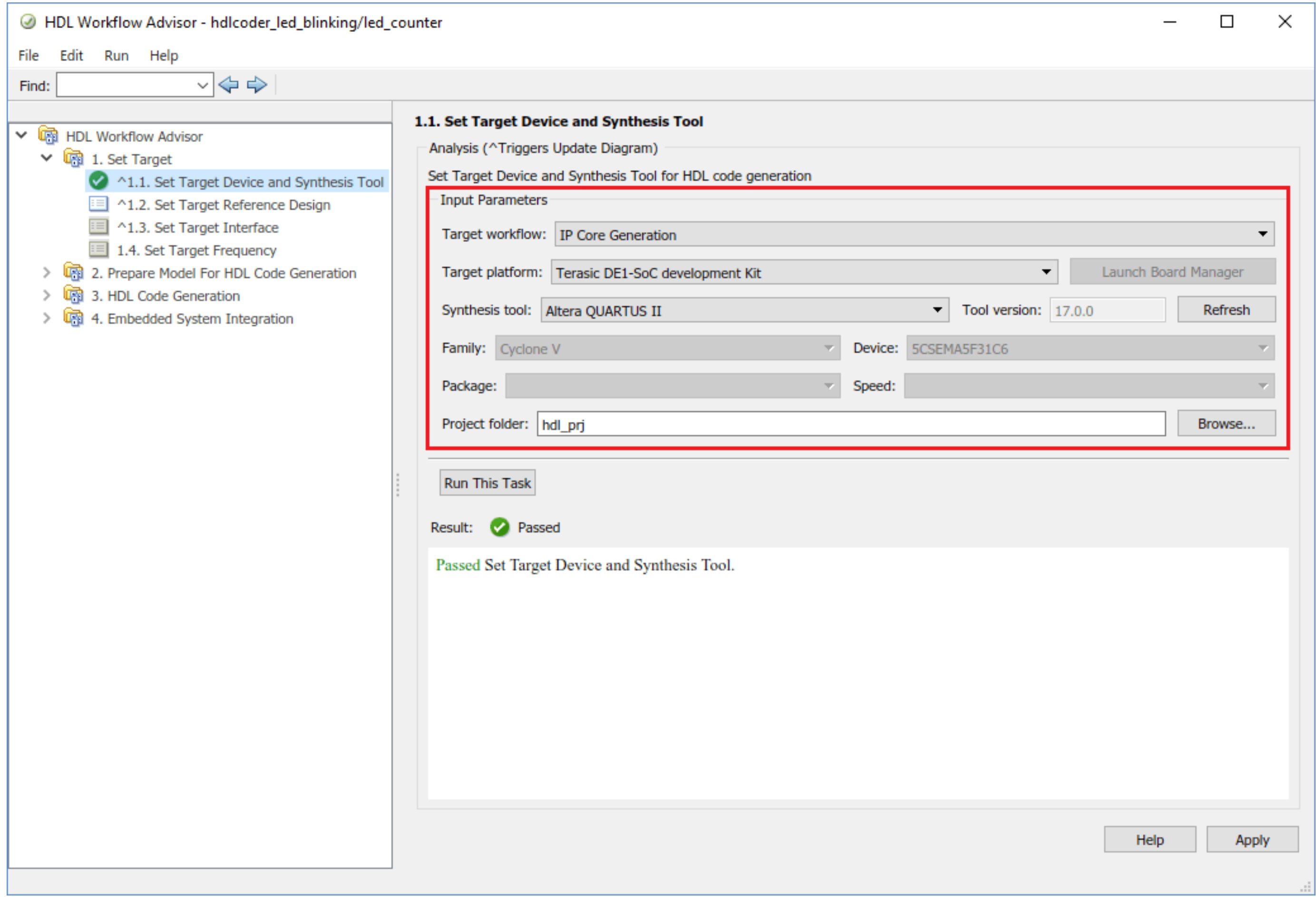
2. Start the IP core generation workflow.

2.1. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem, and choosing **HDL Code** > **HDL Workflow Advisor**.

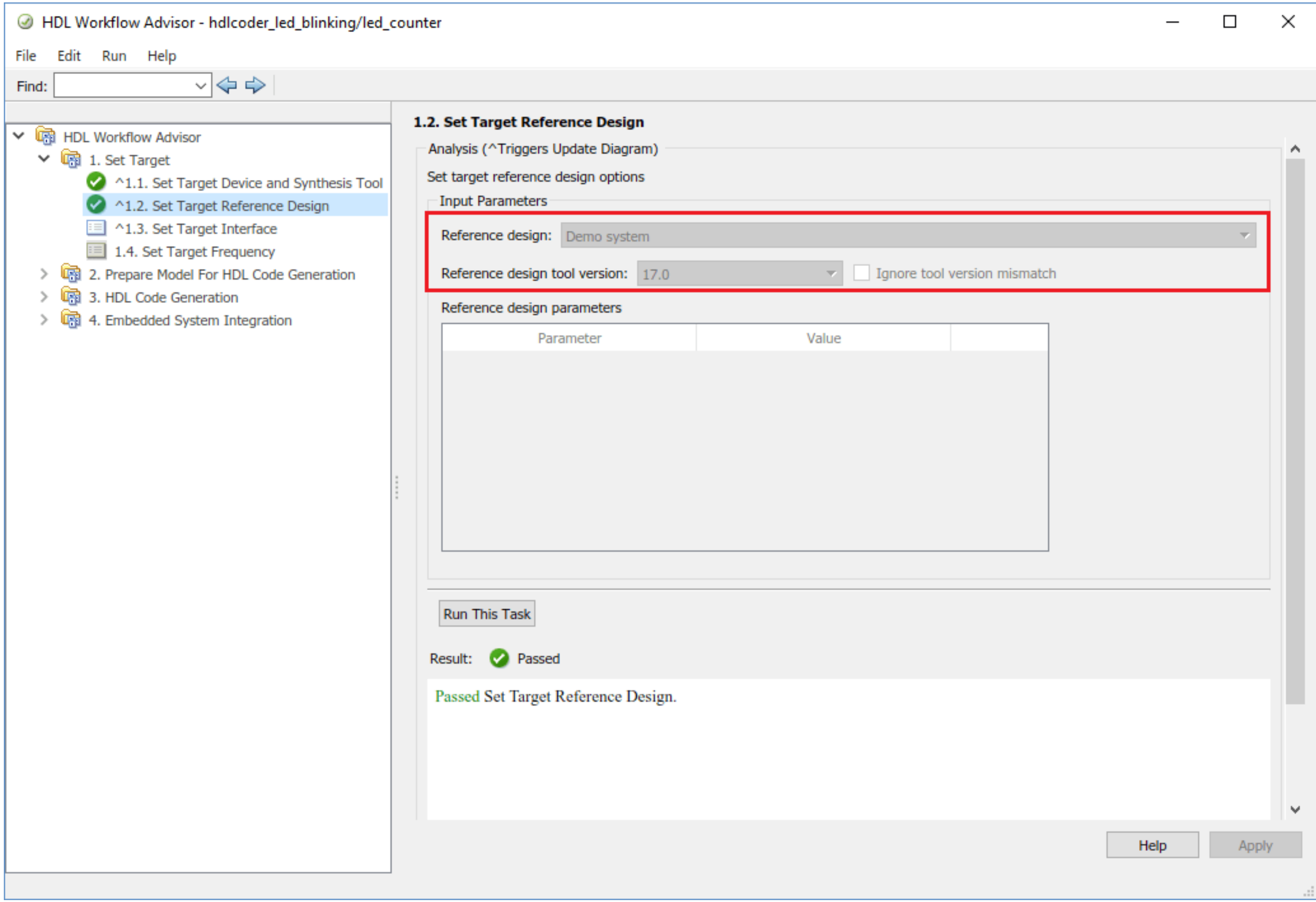
2.2. In the **Set Target** > **Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.

2.3. For **Target platform**, select **Terasic DE1-SoC development Kit**.

2.4. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.

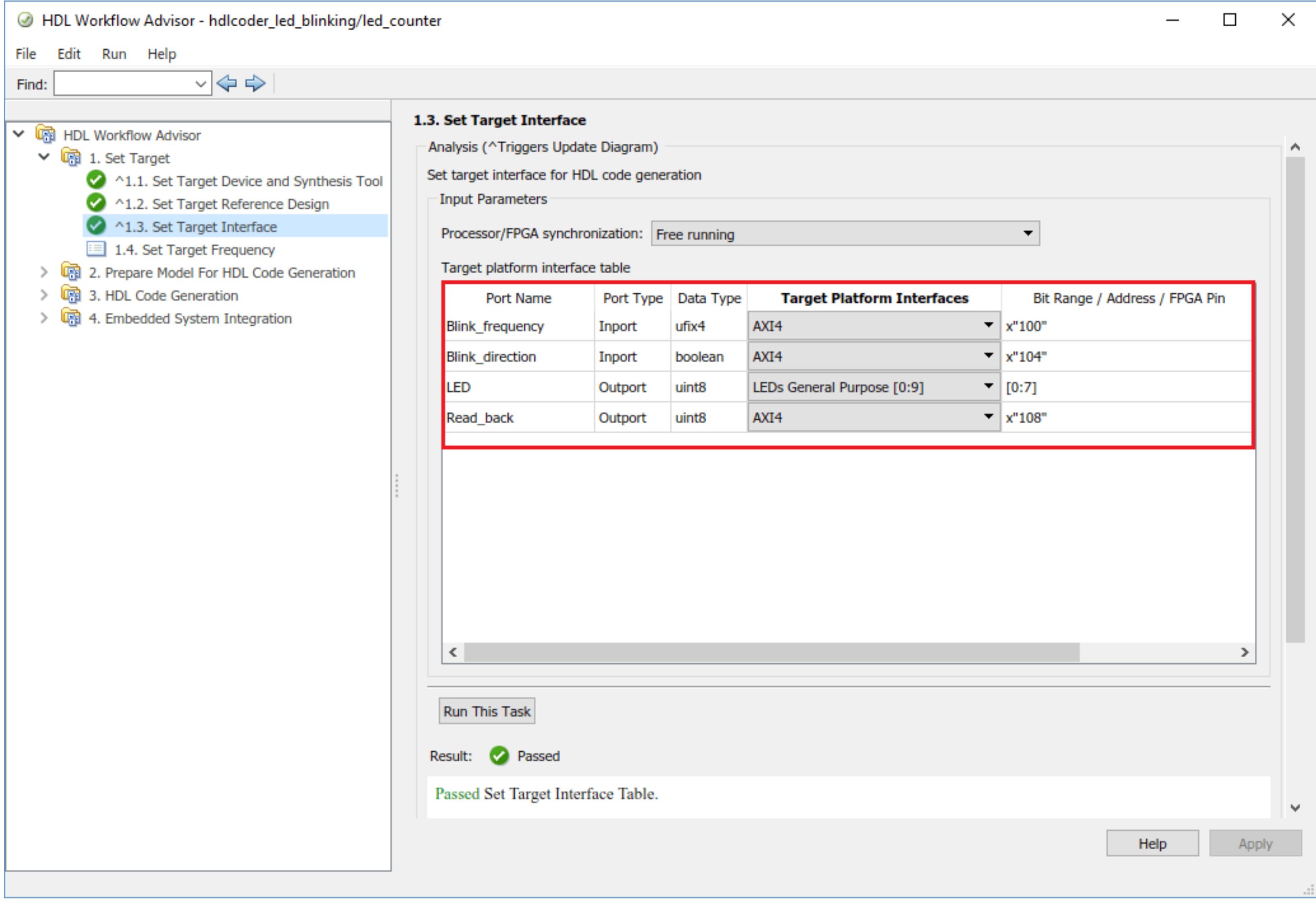


3. In the task 1.2, set target reference design default system is selected. click on **Run This Task**.



4. Configure the Target Interface.

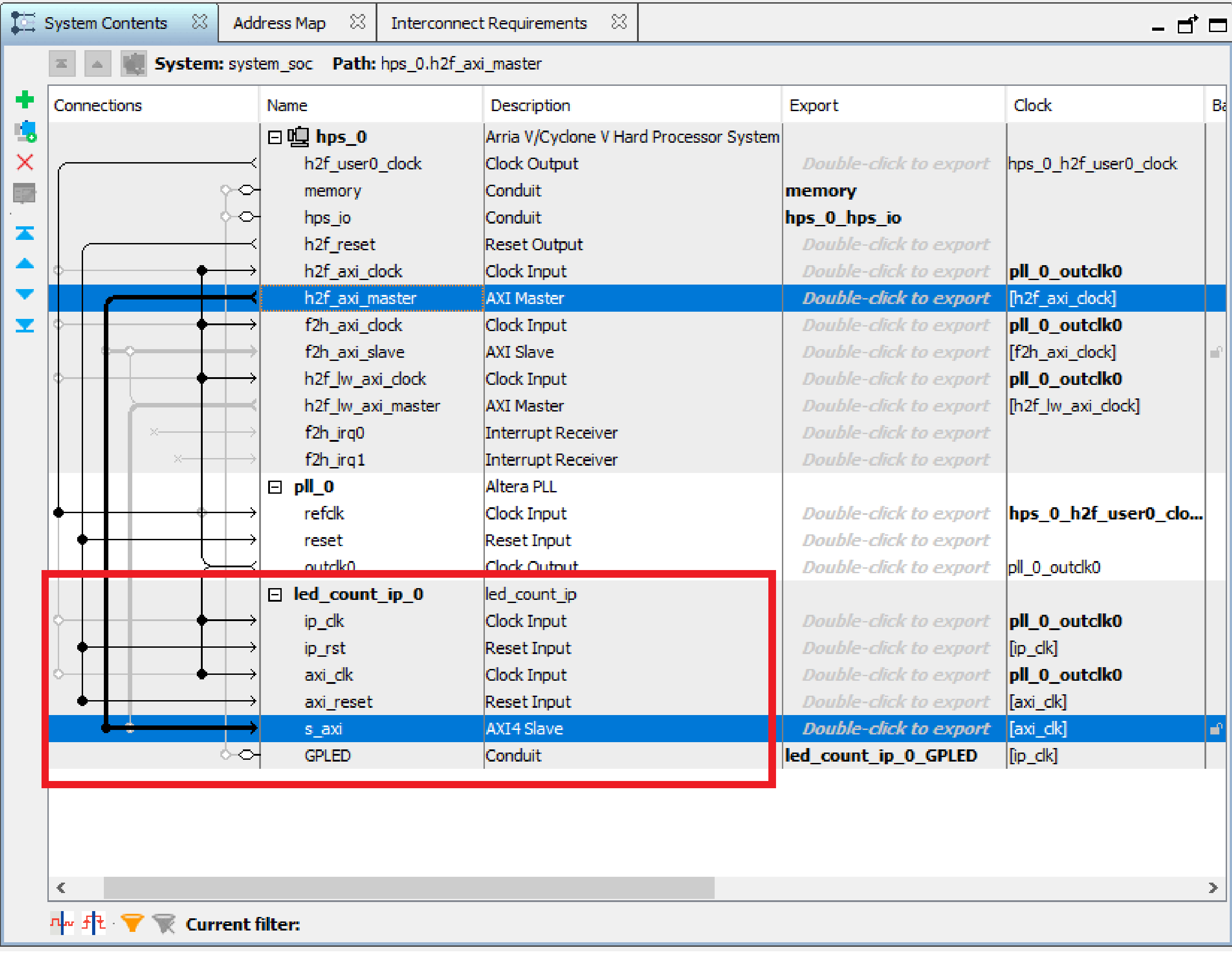
Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to **AXI4**. You can also select AXI4-lite as a target platform interface. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:9]**, which connects to the LED hardware on the Terasic DE1-SoC development Kit.



5. Follow step 3 and step 4 of **Generate an HDL IP core using the HDL Workflow Advisor** section of “Getting Started with Targeting Intel SoC Devices” on page 39-132 example to generate IP core and view the IP core generation report.

6. Follow step 1 of **Integrate the IP core with the Intel Qsys environment** section of “Getting Started with Targeting Intel SoC Devices” on page 39-132 example to integrate the IP core in the reference design and create the Qsys project.

7. Now let us examine the Intel Qsys project created by the SoC workflow after completing the **Create Project** task under **Embedded System Integration**. The following figure shows the SoC project where we have highlighted the HDL IP Core. It is instructive to compare this project with the previous project used in the custom reference design plugin for a deeper understanding of the relationship between a custom reference design and an HDL IP Core.



8. Follow the steps 2, 3 and 4 of **Integrate the IP core with the Intel Qsys environment** section of "Getting Started with Targeting Intel SoC Devices" on page 39-132 example to generate software interface model, generate FPGA bitstream and program target device respectively.

9. The LEDs on the Terasic DE1-SoC will start blinking after loading the bitstream. In addition, you will be able to control the LED blink frequency and direction by executing the software interface model. Refer to **Generate a software interface model** section of "Getting Started with Targeting Intel SoC Devices" on page 39-132 example to control the LED blink frequency and direction from the generated software interface model.

Define Custom Board and Reference Design for Microchip Workflow

This example shows how to define and register a custom board and reference design for a blinking LED model in the Microchip workflow of the HDL Workflow Advisor. You use a SmartFusion2® board, but you can define and register a custom board or a custom reference design for other Microchip platforms.

Requirements

To run this example, you need access to:

- A Microchip Libero SoC Design Suite. For a list of supported versions, see “HDL Language Support and Supported Third-Party Tools and Hardware”.
- A Microchip SmartFusion2 development board with the accessory kit.
- HDL Coder™ Support Package for Microchip FPGA and SoC Devices.

Set Up SmartFusion2 Board

To familiarize yourself with the features of the SmartFusion2 board, see the SmartFusion2 Board Reference Manual. To set up your SmartFusion2 board, follow these steps.

1. Install the USB COM port device drivers on your computer.

2. Connect the shared UART/JTAG USB port on the SmartFusion2 board to your computer.

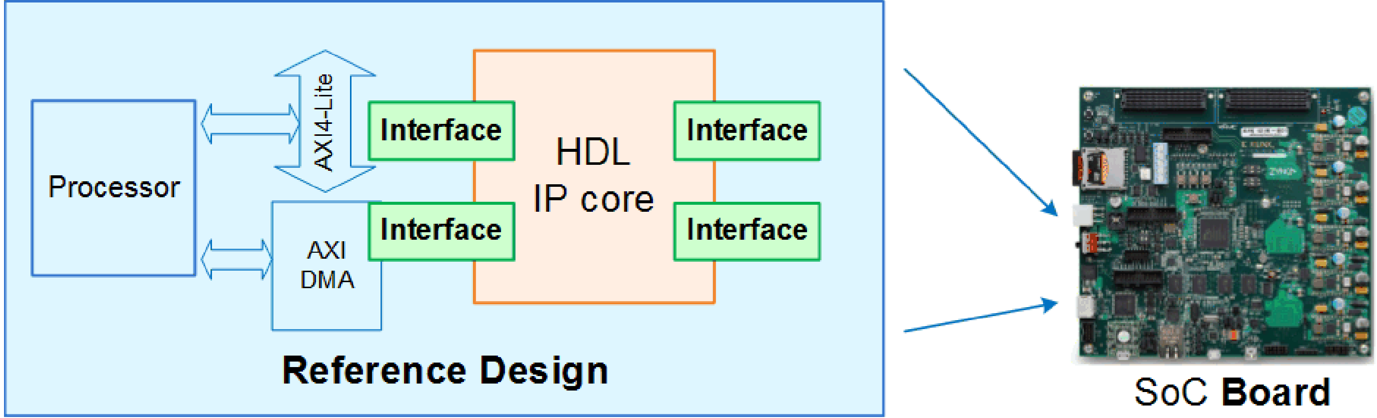
3. Download and install **HDL Coder Support Package for Microchip FPGA and SoC Devices**. To access the hardware support package, in the **Home** tab of your MATLAB session, click **Add-Ons** and then click **Get Hardware Support Packages**.

4. Set up the Microchip Libero tool path by using the `hdlsetuptoolpath` function. For example, to set up the Microchip Libero SoC tool, specify `ToolName` as `Microchip Libero SoC` and `ToolPath` as your installed Libero executable path.

```
hdlsetuptoolpath('ToolName', 'Microchip Libero SoC', ...
    'ToolPath', 'C:\Microsemi\Libero_SoC_v2022.1\Designer\bin\libero.exe');
```

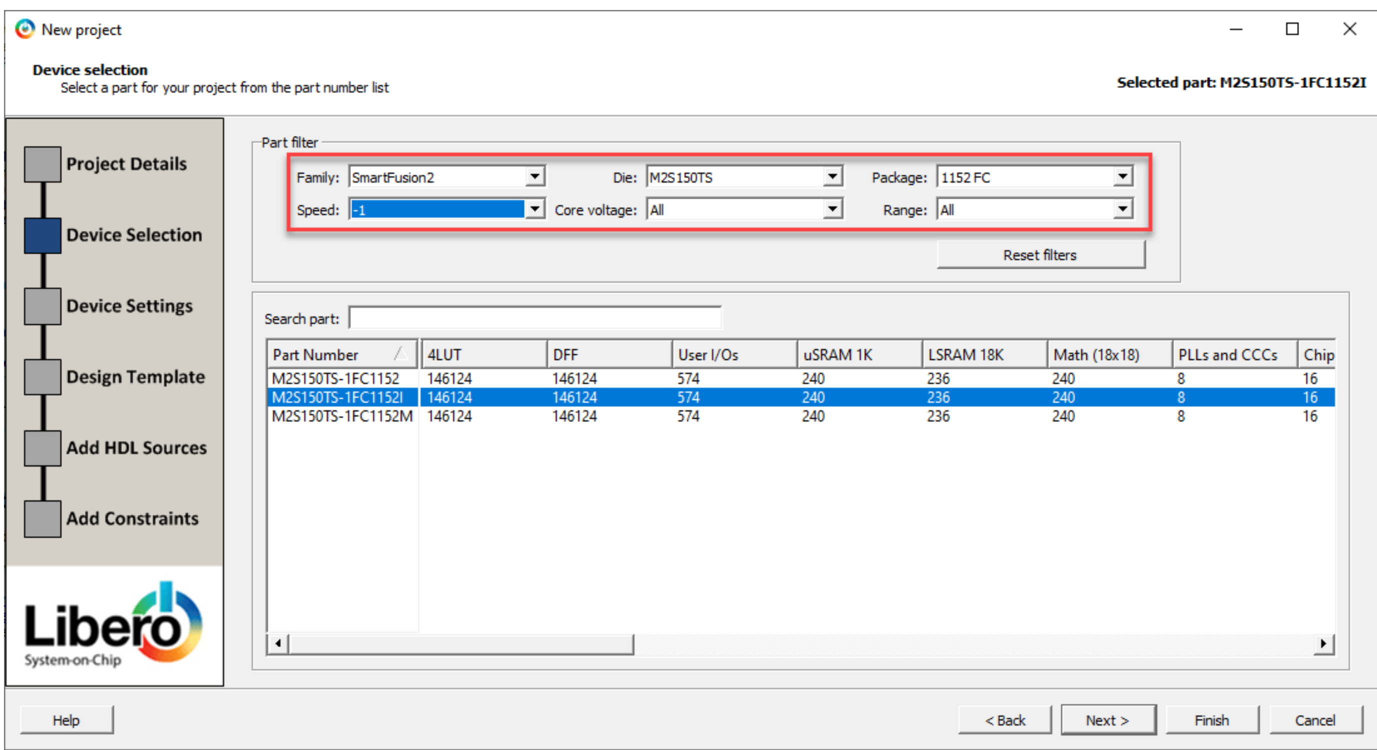
Create and Export Custom Reference Design by Using Microchip Libero SoC

A reference design captures the complete structure of an SoC design, defining the different components and their interconnections. Use the HDL Coder SoC workflow in the HDL Workflow Advisor to generate an IP core that integrates with the reference design and program an SoC board. This figure shows the relationship between a reference design, an HDL IP core, and an SoC board.

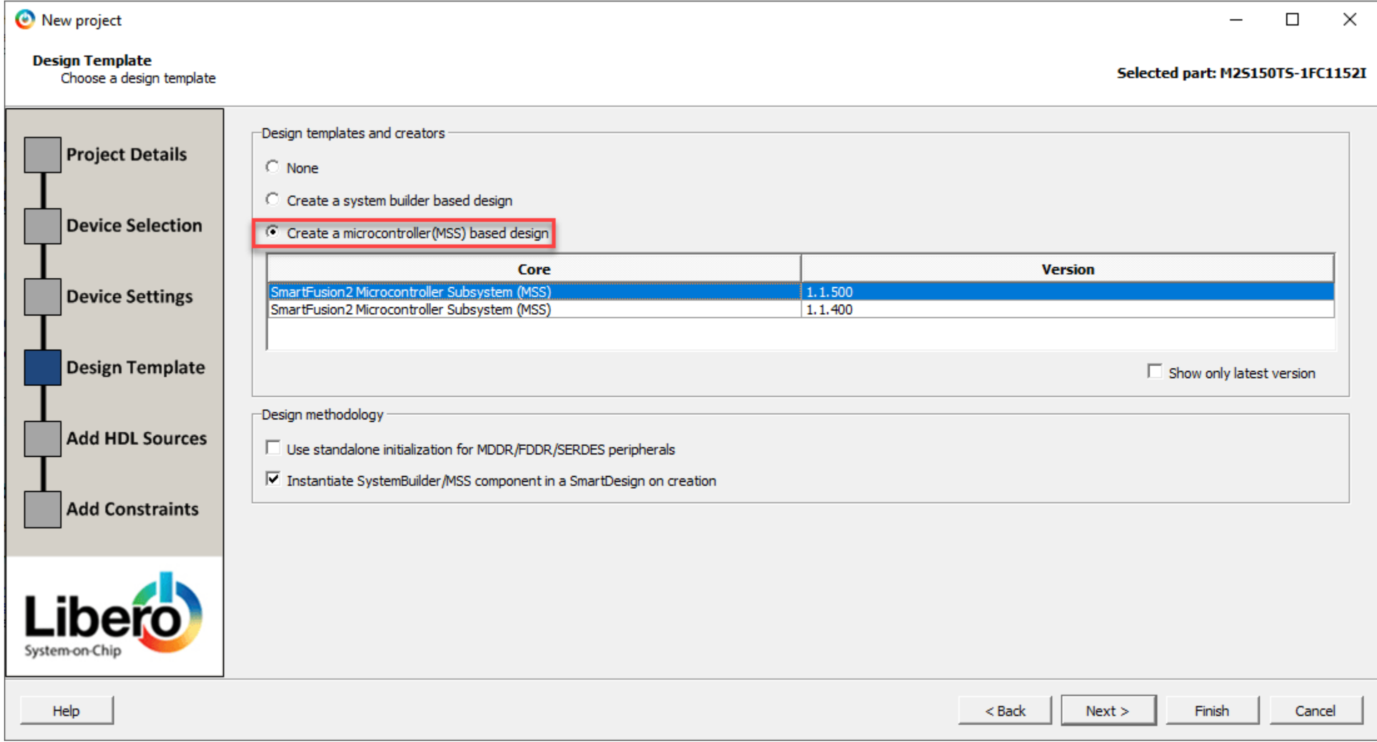


To create and export a reference design by using the Microchip Libero Tool environment, follow these steps.

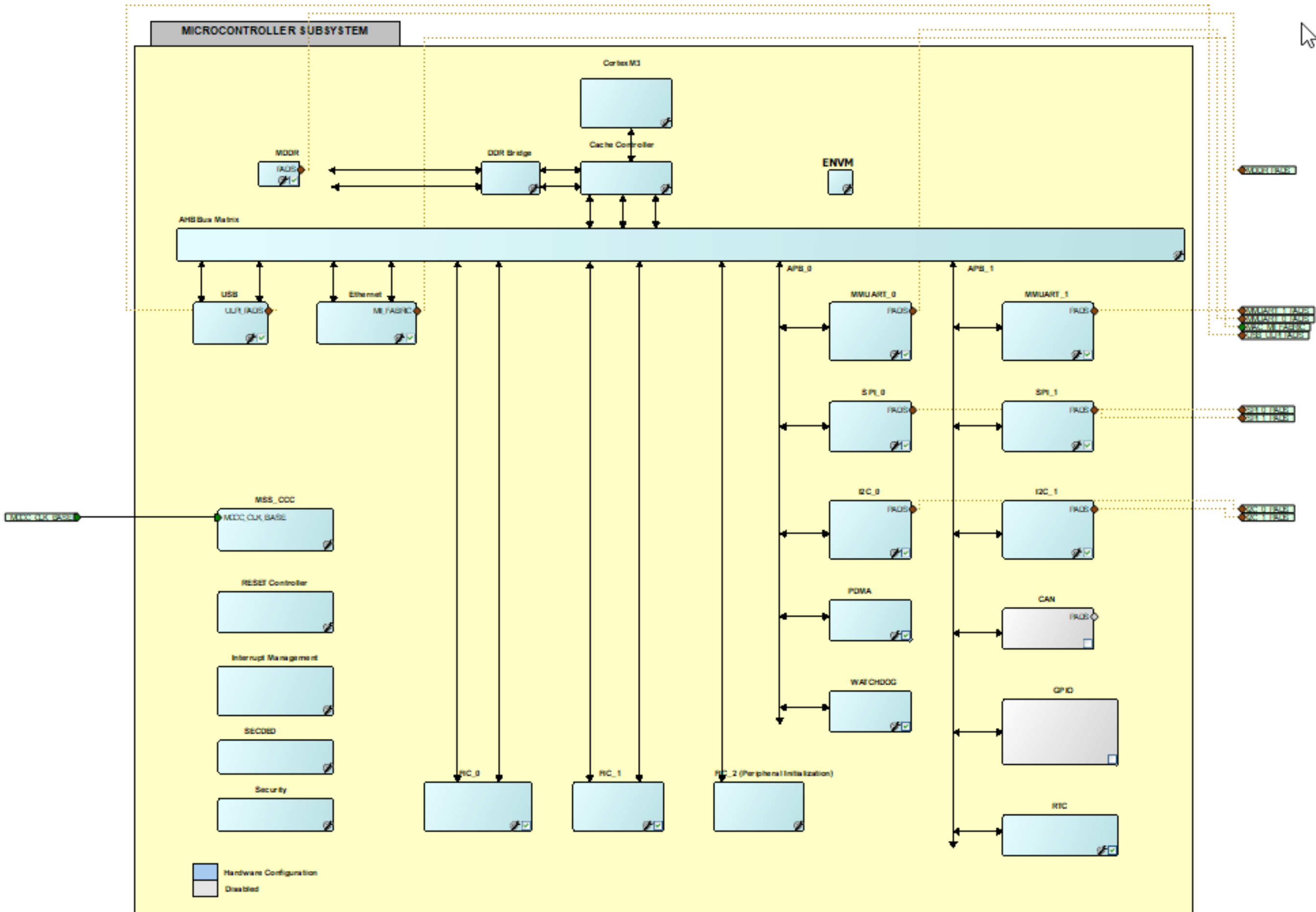
1. Create a microcontroller subsystem (MSS) based Microchip Libero SoC RTL project by using the SmartFusion2 board settings as specified in this figure and then click **Next**. This Libero project is required only to generate SmartFusion2 configuration files.



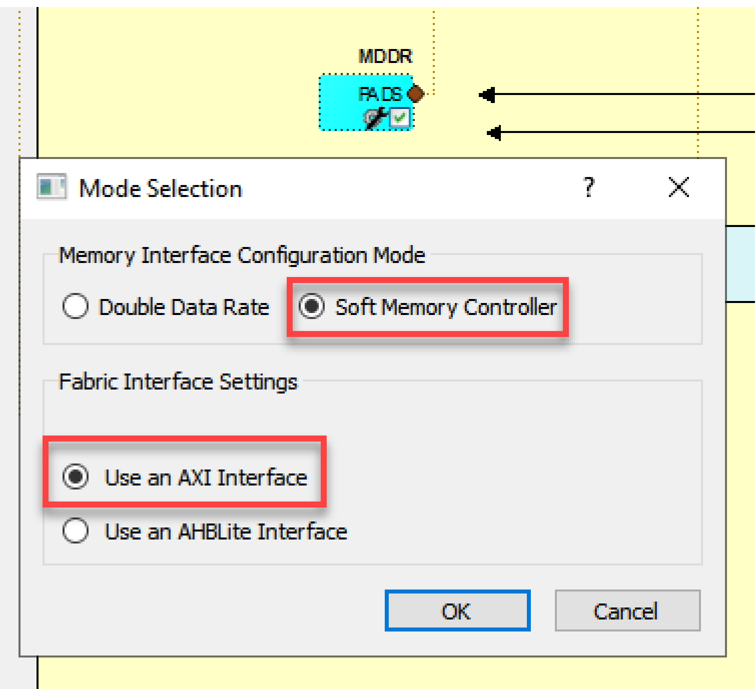
2. In the Design Template window, select **Create a microcontroller (MSS) based design** and then click **Next**.



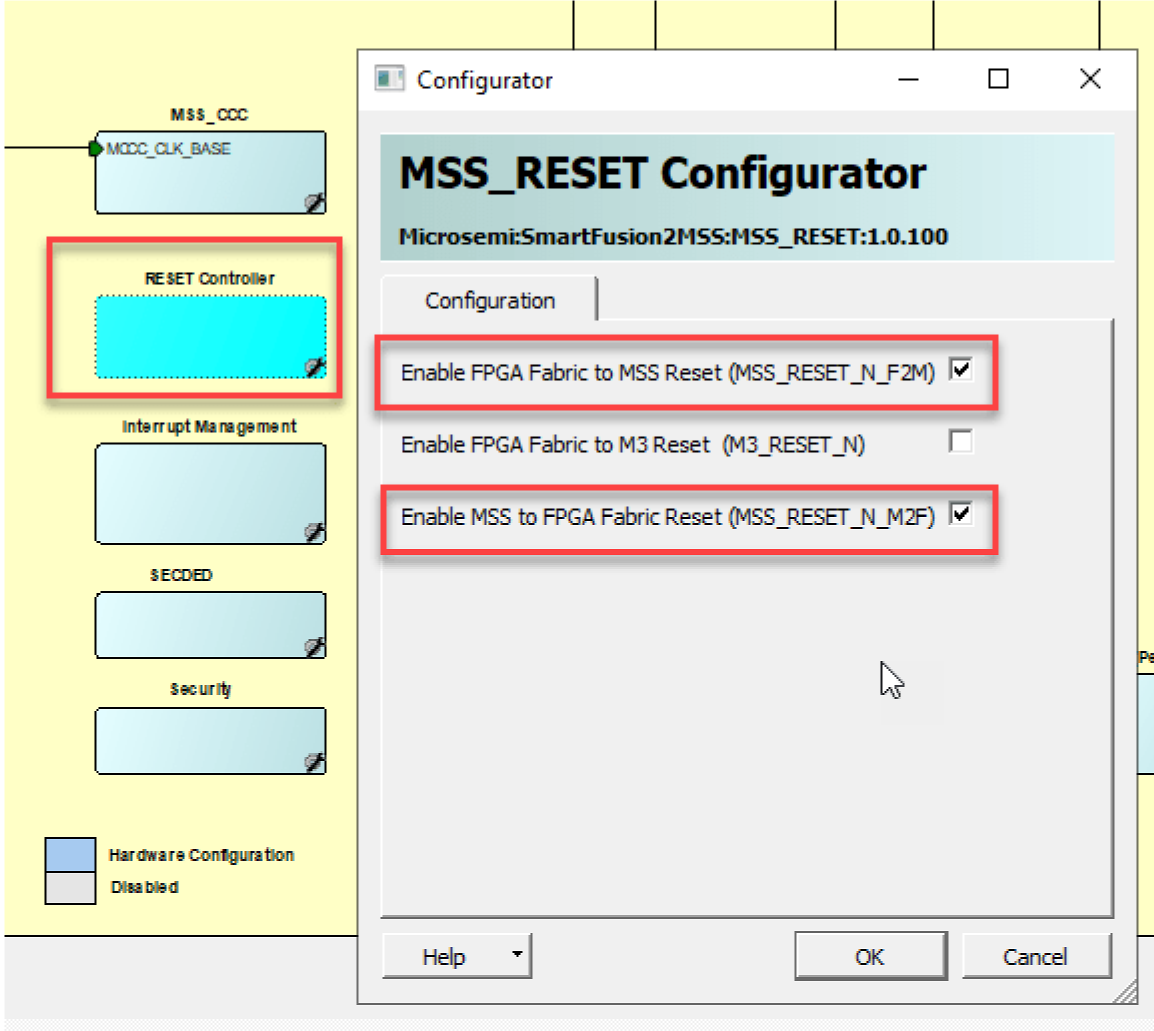
3. Click **Finish** to create a smart design containing the SmartFusion2 MSS component. To configure the MSS, double-click **SmartFusion2_MSS_MSS** IP.



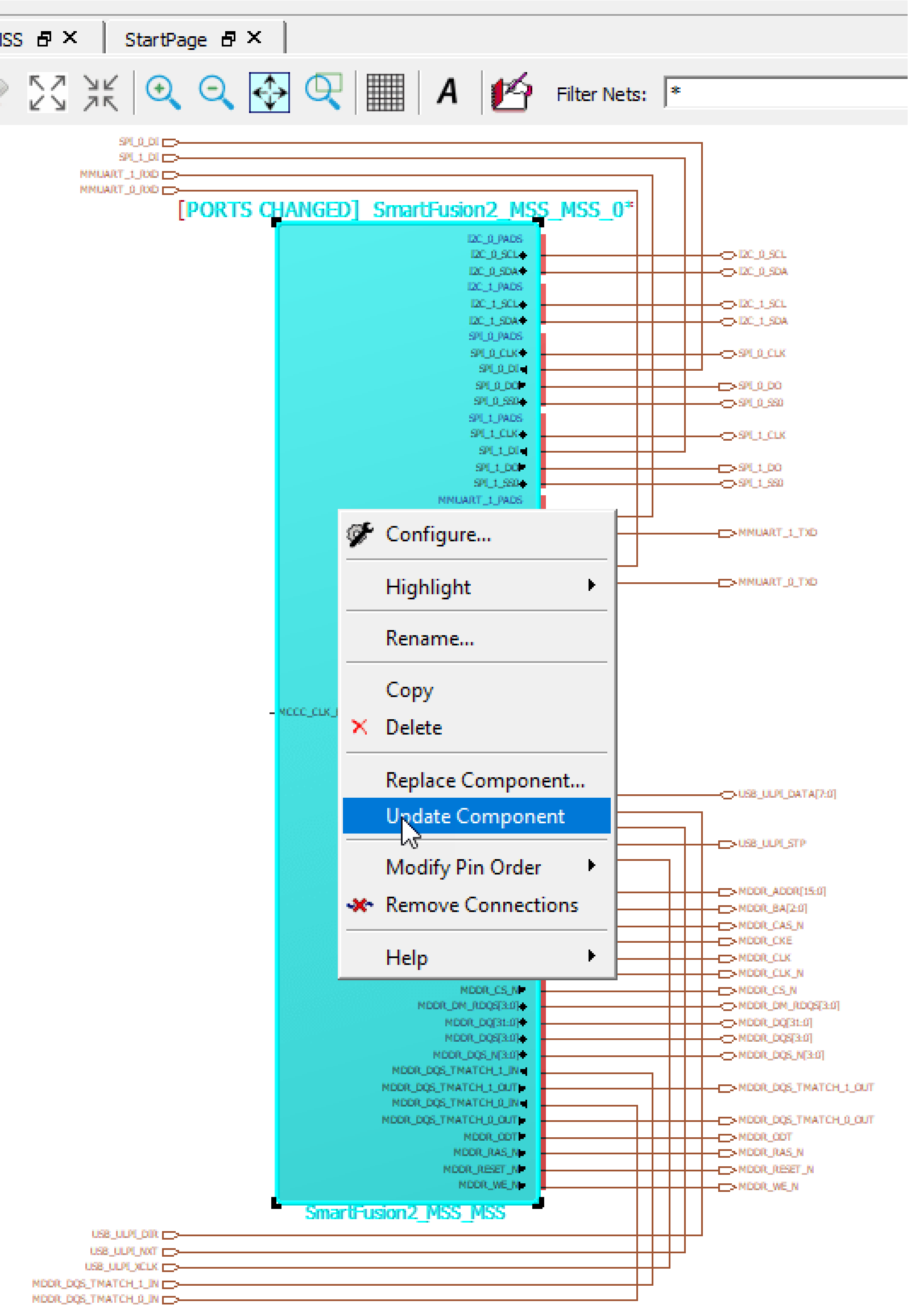
4. Clear the check boxes from the peripherals that you do not want to use. Double-click **MDDR** and select this configuration for the AXI Master interface.



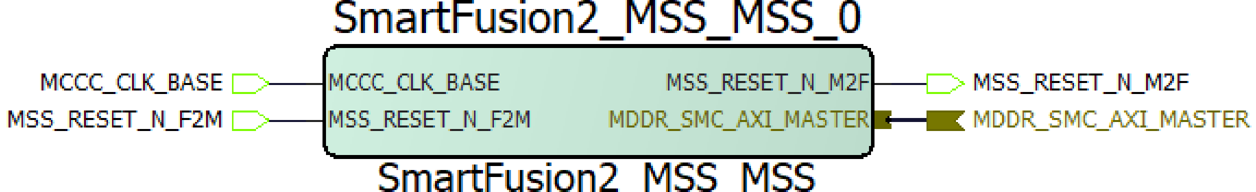
5. Double-click **Reset Controller** to configure the reset controller, which enables the **MSS_RESET_N_M2F** and **MSS_RESET_N_M2F** parameters.



6. Save the **SmartFusion2_MSS_MSS** smart design. Right-click the **SmartFusion2_MSS_MSS** IP and select **Update Component**.

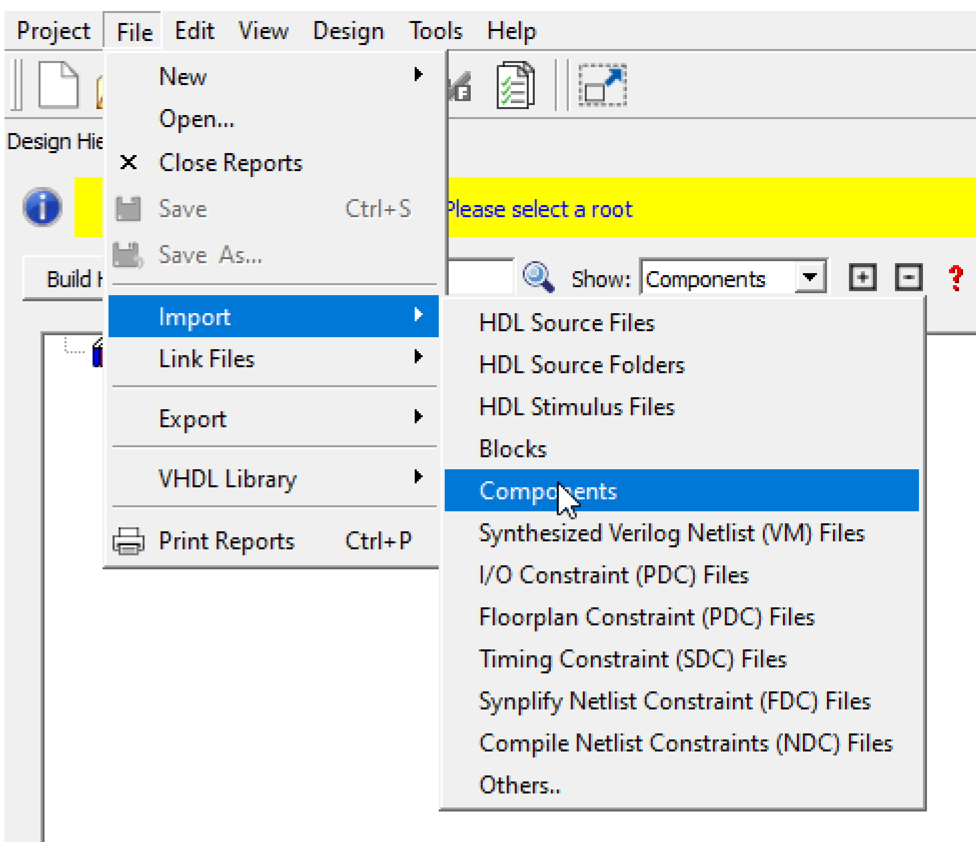


7. MSS IP is configured with AXI Master interface and reset connections. Promote all pins to the top level of your design and click **Generate Component**.

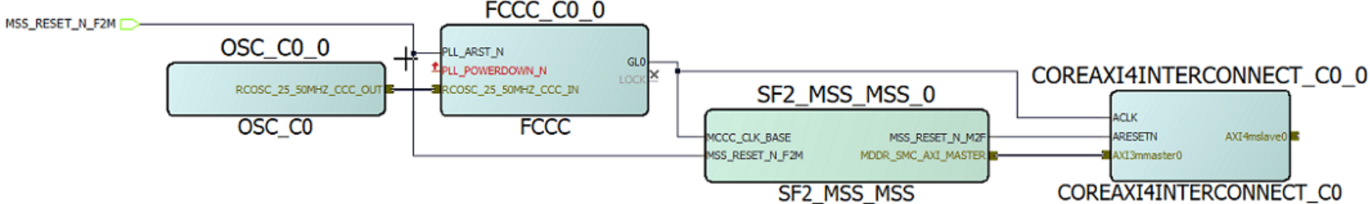


8. Generate the **SmartFusion2_MSS_MSS** component files in the Libero project folder `SmartFusion2_MSS\component\work\SmartFusion2_MSS_MSS`. Save the generated `SmartFusion2_MSS_MSS.cxf` and `SmartFusion2_MSS_MSS.sdb` files in the same folder as the Simulink® blinking LED model is saved.

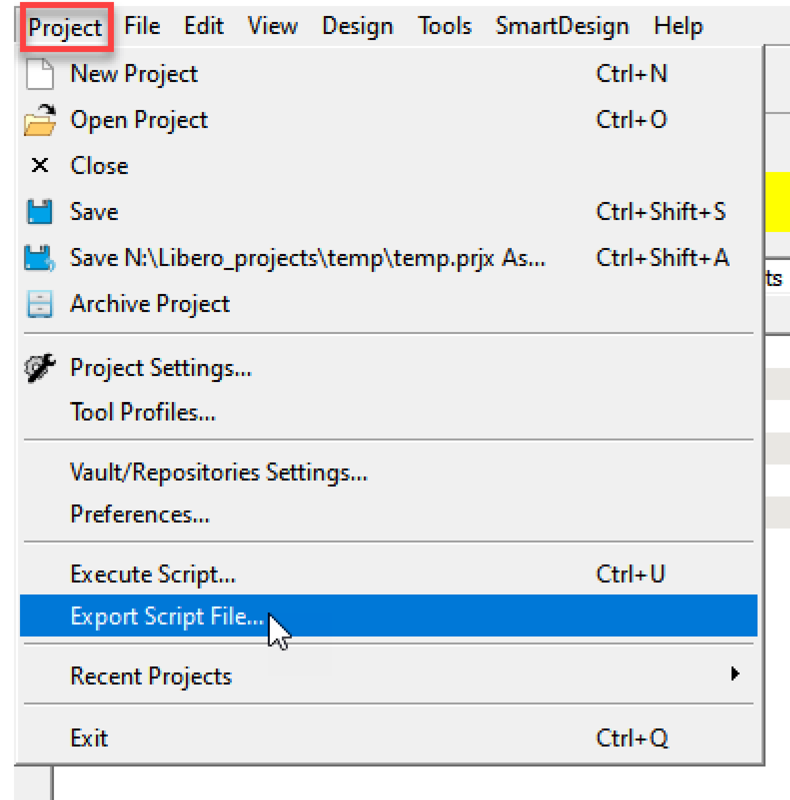
9. Create a new empty Libero project with the same SmartFusion2 device settings and import Microcontroller SubSystem (MSS) processor. Click **File**, hover on **Import**, then click **Components**. Specify the path of the `SmartFusion2_MSS_MSS.cxf` file that you generate in step 8. Create a new smart design with the name `Libero_sd` and instantiate `SmartFusion2_MSS_MSS` IP in `Libero_sd`. Generate a component for the `SmartFusion2_MSS_MSS` IP by promoting all the pins to the top level.



10. Add `OSC_C0`, `FCCC_C0` and `COREAXI4INTERCONNECT_C0` IPs to the smart design. Connect the on-chip oscillator clock to clock conditioning circuitry (CCC) and use the output of the CCC to drive whole circuit clock (GL0).



11. Export the completed block design as a Tcl script with the name `design1_led.tcl`.



The exported Tcl script (`design1_led.tcl`) constitutes the custom reference design. You use this reference design in the HDL Coder SoC workflow of the HDL Workflow Advisor to recreate the block design and integrate the generated HDL IP core with the block design in a Microchip Libero project.

Register SmartFusion2 Board in HDL Workflow Advisor

To register the SmartFusion2 board in the HDL Workflow Advisor, follow these steps.

1. Create a board registration file with the name `hdlcoder_board_customization` and add it to the MATLAB path.

A board registration file contains a list of board plugins. A board plugin is a MATLAB package folder containing a board definition file and all the reference design plugins associated with the board.

This code describes the contents of a board registration file that contains `SmartFusion2Registration` board plugin to register the SmartFusion2 board in HDL Workflow Advisor. The function finds any registration file with the specified name on the MATLAB path and returns a cell array with the locations of the board plugins. The board plugin must be a package folder that is accessible from your MATLAB path containing a board definition file.

```
function r = hdlcoder_board_customization
% Board plugin registration file

r = { ...
      'SmartFusion2Registration.plugin_board', ...
     };
end
```

2. Create the board definition file.

A board definition file contains information about the SoC board.

Create a SmartFusion2 board definition file named `plugin_board.m` that resides inside the board plugin `SmartFusion2Registration`.

For information about the FPGA I/O pin locations (`'FPGAPin'`) and standards (`'IOSTANDARD'`), see the SmartFusion2 constraints file from the Microchip website.

The property `BoardName` defines the name of the SmartFusion2 board in the HDL Workflow Advisor.

```
function hB = plugin_board()
% Board definition


% Construct board object
hB = hdlcoder.Board;

hB.BoardName = 'Microsemi SmartFusion2 SoC FPGA Advanced Developement Kit';


% FPGA device information
hB.FPGAVendor = 'Microchip';
hB.FPGAFamily = 'SmartFusion2';
hB.FPGADevice = 'M2S150TS';
hB.FPGAPackage = '1152 FC';
hB.FPGASpeed  = '-1';


% Tool information
hB.SupportedTool = {'Microchip Libero SoC'};


% FPGA JTAG chain position
hB.JTAGChainPosition = 2;


%% Add interfaces
```

```
% Standard "External Port" interface
hB.addExternalPortInterface( ...
   'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});


% Custom board external I/O interface
hB.addExternalIOInterface( ...
   'InterfaceID', 'LEDs General Purpose', ...
   'InterfaceType', 'OUT', ...
   'PortName', 'LEDs', ...
   'PortWidth', 8, ...
   'FPGAPin', {'D26', 'F26', 'A27', 'C26', 'C28', 'B27', 'C27', 'E26'}, ...
   'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
   'InterfaceID', 'Push Buttons', ...
   'InterfaceType', 'IN', ...
   'PortName', 'PushButtons', ...
   'PortWidth', 4, ...
   'FPGAPin', {'J25', 'H25', 'J24', 'H23'}, ...
   'IOPadConstraint', {'IOSTANDARD = LVCMOS25'});

hB.addExternalIOInterface( ...
  'InterfaceID', 'Slide switches ', ...
  'InterfaceType', 'IN', ...
  'PortName', 'SlideSwitches', ...
  'PortWidth', 8, ...
  'FPGAPin', {'F25', 'G25', 'J23', 'J22', 'G27' , 'H27' , 'F23' , 'G23'}, ...
  'IOPadConstraint', {'IOSTANDARD = LVCMOS25'});
```

Register Custom Reference Design in HDL Workflow Advisor

To register the custom reference design in HDL Workflow Advisor, follow these steps.

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` containing a list of reference design plugins associated with an SoC board.

A reference design plugin is a MATLAB package folder containing the reference design definition file and all the files associated with the SoC design project. A reference design registration file must also contain the name of the associated board.

This code describes the contents of a SmartFusion2 reference design registration file containing the reference design plugin `SmartFusion2Registration.Libero_12_6` associated with the `Microsemi SmartFusion2 SoC FPGA Advanced Developement Kit` board. The registration file finds files with the specified name inside a board plugin folder or on the MATLAB path. The function returns a cell array containing the locations of the reference design plugins and a character vector containing the associated board name. The reference design plugin must be a package folder that is accessible from the MATLAB path and must contain a reference design definition file.

```
function [rd,boardName] = hdlcoder_ref_design_customization
% Reference design plugin registration file

rd = {'SmartFusion2Registration.Libero_12_6.plugin_rd', ...
     };

boardName = 'Microsemi SmartFusion2 SoC FPGA Advanced Developement Kit';
end
```

2. Create the reference design definition file.

A reference design definition file defines the interfaces between the custom reference design and the HDL IP core that you generate.

Create a SmartFusion2 reference design definition file `plugin_rd.m` to associate with the `Microsemi SmartFusion2 SoC FPGA Advanced Developement Kit` board that resides inside the reference design plugin `SmartFusion2Registration.Libero_12_6`. The `ReferenceDesignName` property defines the name of the reference design as `Default system` in HDL Workflow Advisor.

```
function hRD = plugin_rd()
% Reference design definition


% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Microchip Libero SoC');

hRD.ReferenceDesignName = 'Default system';
hRD.BoardName = 'Microsemi SmartFusion2 SoC FPGA Advanced Developement Kit';


% Tool information
hRD.SupportedToolVersion = {'12.0','12.6','2022.1'};


%% Add custom design files
% Add custom Libero  design
hRD.addCustomLiberoDesign( ...
    'CustomBlockDesignTcl', 'design1_led.tcl');


% Add custom MSS Config
hRD.addCustomComponentFiles(...
    'CustomMSSCxfFile','SF2_MSS_MSS.cxf',...
    'CustomMSSSdbFile','SF2_MSS_MSS.sdb');


%% Add interfaces
% Add clock interface
hRD.addClockInterface( ...
    'ClockConnection', 'FCCC_C0_0/GL0', ...
    'ResetConnection', 'SF2_MSS_MSS_0/MSS_RESET_N_M2F',...
    'DefaultFrequencyMHz', 50,...
    'MinFrequencyMHz', 5,...
    'MaxFrequencyMHz', 500,...
    'ClockModuleInstance', 'FCCC_C0_0',...
    'ClockNumber', 1);


% Add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'COREAXI4INTERCONNECT_C0_0/AXI4mslave0', ...
    'BaseAddress', '0xA0000000');


% Disable 'Generate Software Interface Model' task
hRD.HasProcessingSystem = false;
```

A reference design plugin must also contain the SoC design project files.

The SmartFusion2 reference design plugin folder `SmartFusion2Registration.Libero_12_6` must contain the Tcl script `design1_led.tcl` exported from the Microchip Libero project. The SmartFusion2 reference design definition file `plugin_rd.m` identifies the SoC design project file by using the `addCustomLiberoDesign` function.

```
hRD.addCustomLiberoDesign('CustomBlockDesignTcl', 'design1_led.tcl');
```

The reference design definition file `plugin_rd.m` also defines the interface connections between the custom reference design and the HDL IP core by using the `addClockInterface` and `addAXI4SlaveInterface` functions:

```
hRD.addClockInterface( ...
     'ClockConnection', 'FCCC_C0_0/GL0', ...
     'ResetConnection', 'SF2_MSS_MSS_0/MSS_RESET_N_M2F',...
hRD.addAXI4SlaveInterface( ...
     'InterfaceConnection', 'COREAXI4INTERCONNECT_C0_0/AXI4mslave0', ...
     'BaseAddress', '0xA0000000');
```

Generate HDL IP core for SmartFusion2 Board

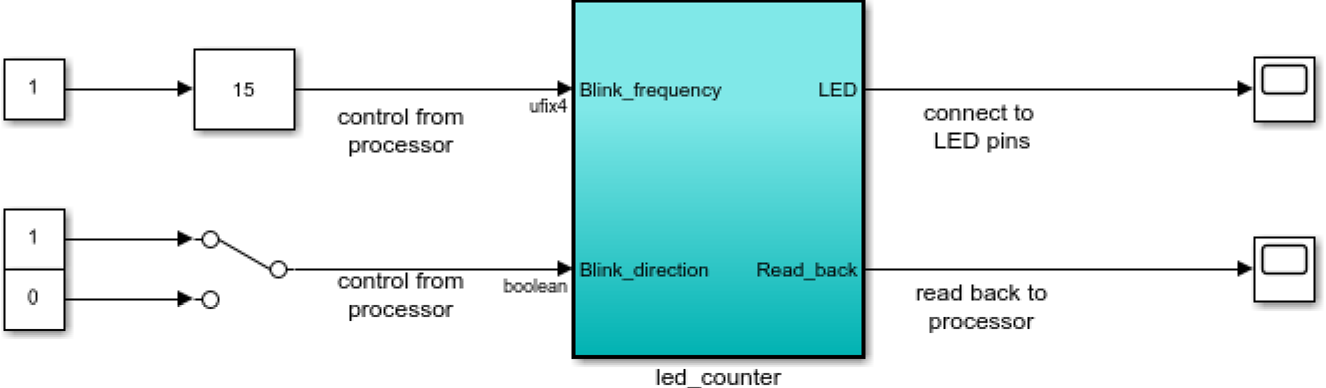
Use the custom board and reference design registration system to generate an HDL IP core that blinks LEDs on the SmartFusion2 board.

1. Add the SmartFusion2 board registration files to the MATLAB path using these commands.

```
hdlcoder_microchip_examples_root;
addpath(fullfile(hdlcoder_microchip_examples_root,'SmartFusion2'));
```

2. Open the Simulink LED blinking model.

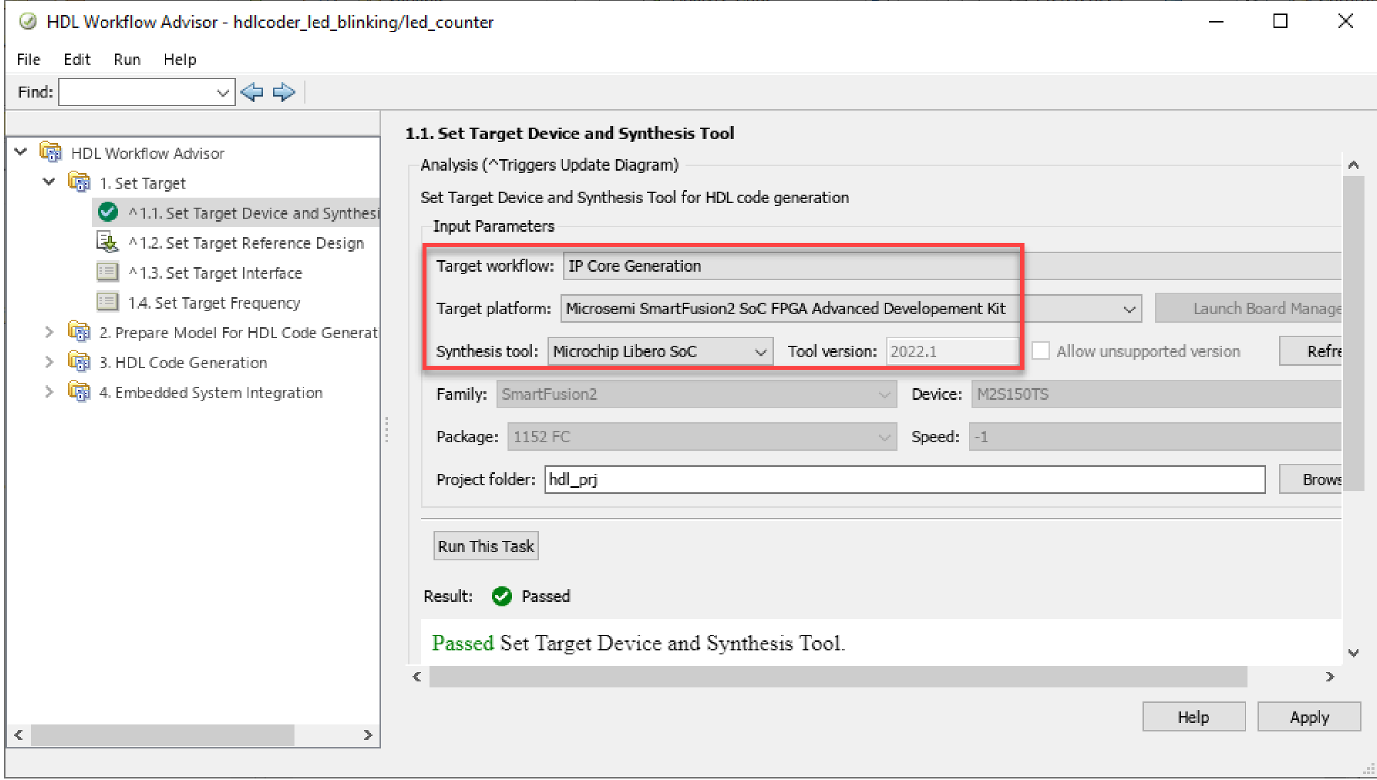
```
open_system('hdlcoder_led_blinking');
```



Copyright 2014-2023 The MathWorks, Inc.

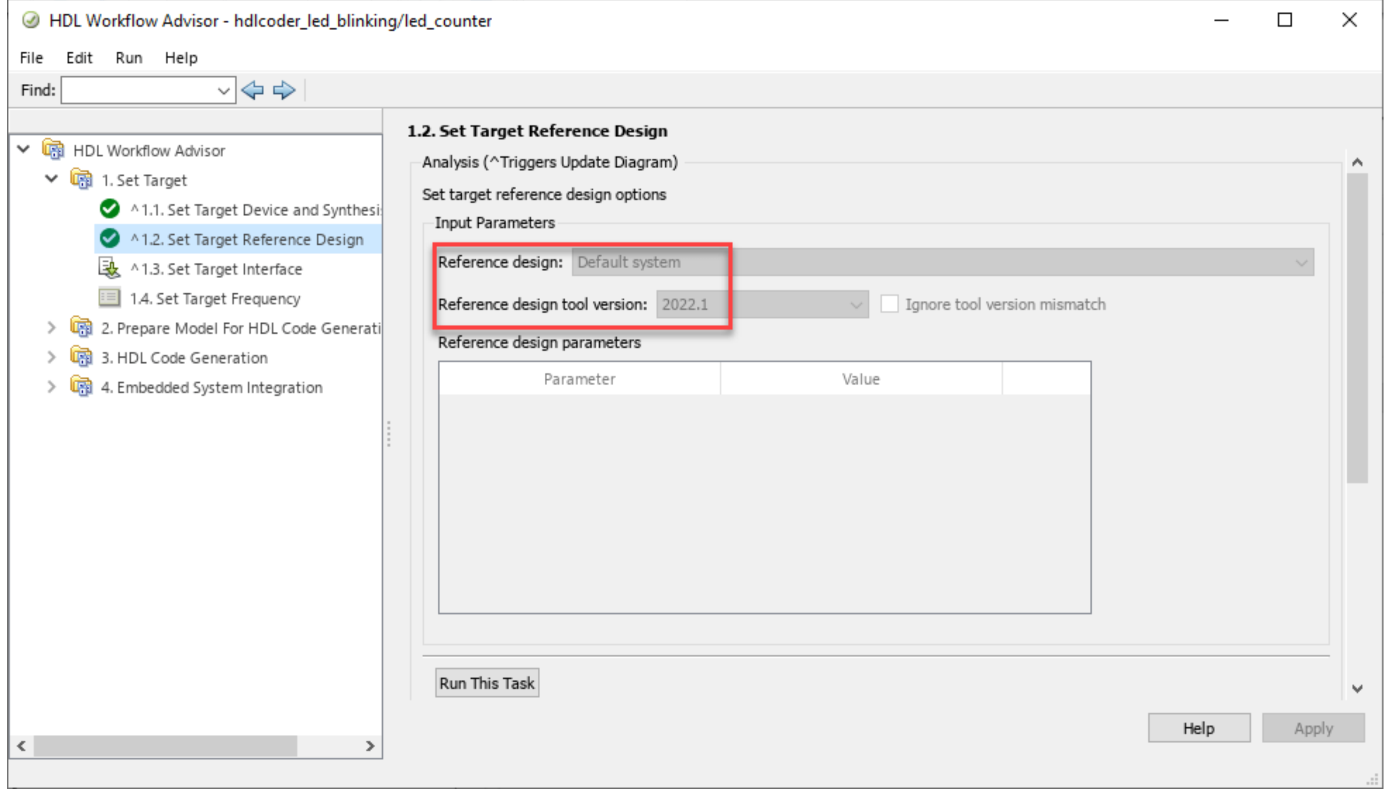
3. Launch the HDL Workflow Advisor from the `hdlcoder_led_blinking_4bit/led_counter` subsystem by right-clicking the `led_counter` subsystem, and hovering over **HDL Code**, and then clicking **HDL Workflow Advisor**. Alternatively, click the **Launch HDL Workflow Advisor** box in the model.

In the **Set Target** > **Set Target Device and Synthesis Tool** task, set **Target workflow** to `IP Core Generation`. In the **Target Platform** dropdown menu, select `Microsemi SmartFusion2 SoC FPGA Advanced Developement Kit`.

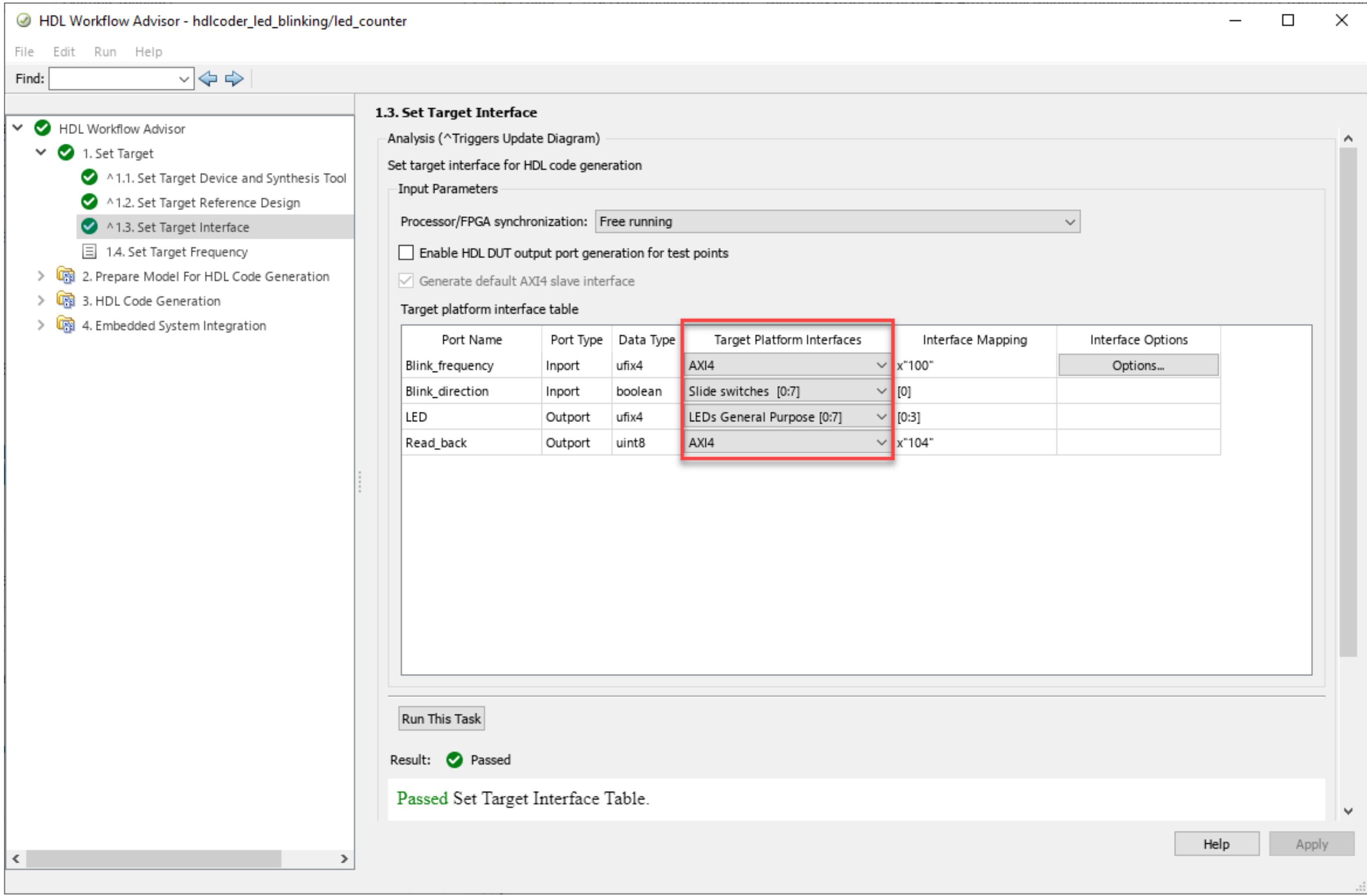


5. Click **Run This Task** to complete the **Set Target Device and Synthesis Tool** task.

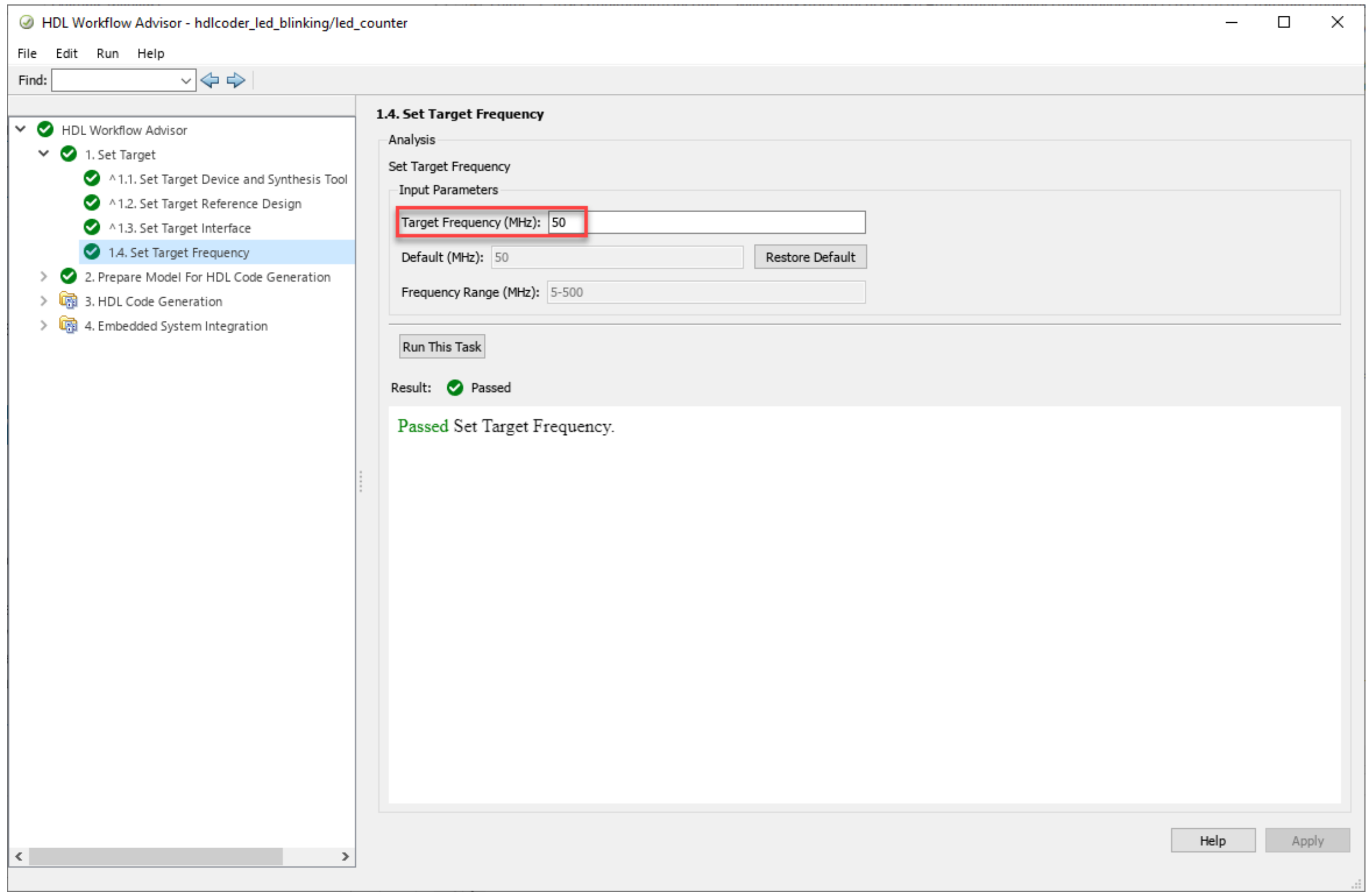
6. In the **Set Target** > **Set Target Reference Design** task, set the **Reference design** field to `Default system` and click **Run This Task**.



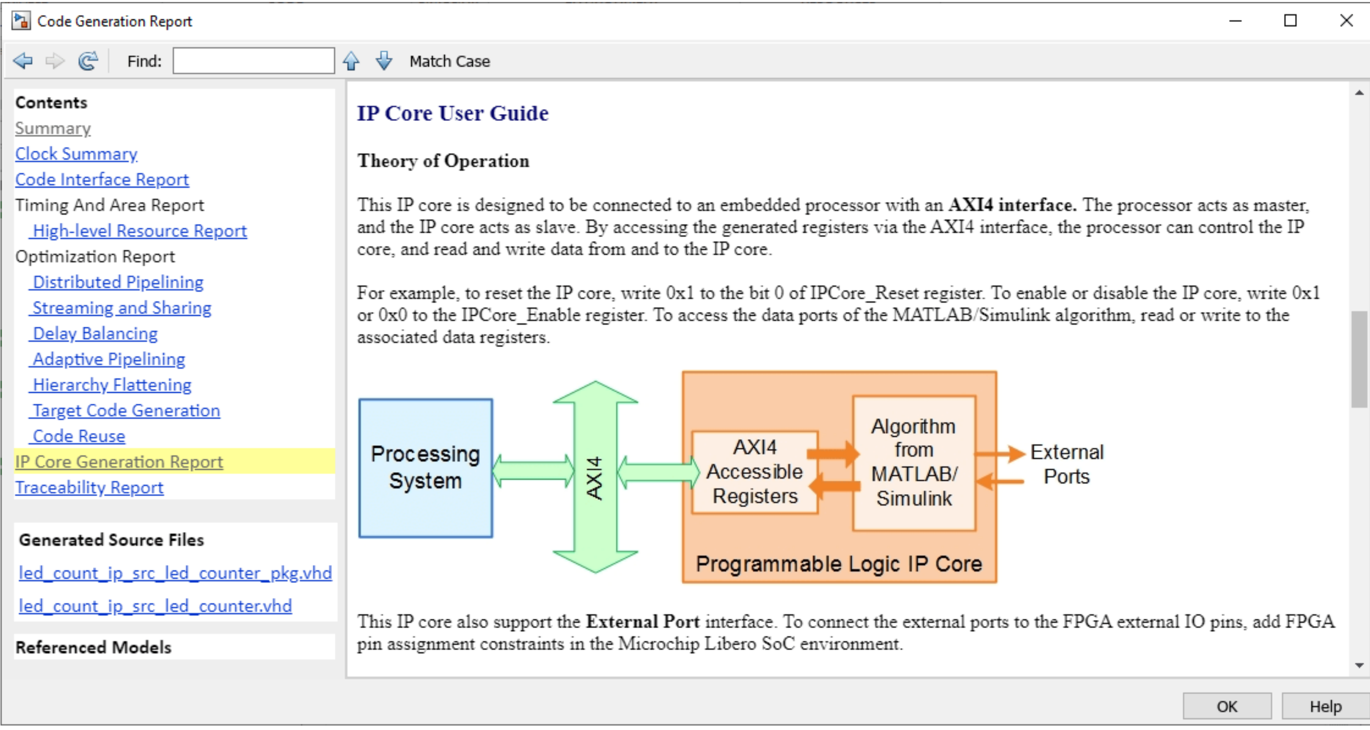
7. In the **Set Target Interface** task, set the Target Platform Interface options and click **Run This Task**.



8. In the **Set Target Frequency** task, set **Target Frequency** to 50 MHz.

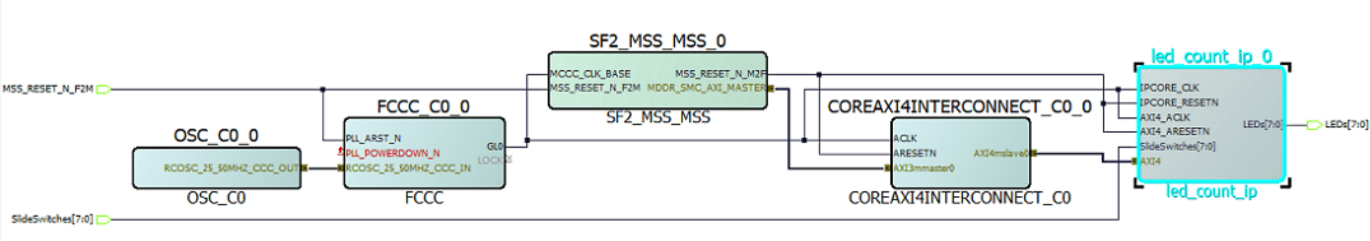


9. To generate the IP core and view the IP core generation report, right-click **Generate RTL Code and IP Core** and select **Run to Selected Task**. After you generate the custom IP core, the IP core files are in the `ipcore` folder within your project folder. An HTML custom IP core report is generated with the custom IP core. The report describes the behavior and contents of the generated custom IP core.


Code Generation Report
Find:
Match Case
Contents
Summary
Clock Summary
Code Interface Report
Timing And Area Report
High-level Resource Report
Optimization Report
Distributed Pipelining
Streaming and Sharing
Delay Balancing
Adaptive Pipelining
Hierarchy Flattening
Target Code Generation
Code Reuse
IP Core Generation Report
Traceability Report
Generated Source Files
led_count_ip_src_led_counter_pkg.vhd
led_count_ip_src_led_counter.vhd
Referenced Models
IP Core User Guide
Theory of Operation
This IP core is designed to be connected to an embedded processor with an AXI4 interface. The processor acts as master, and the IP core acts as slave. By accessing the generated registers via the AXI4 interface, the processor can control the IP core, and read and write data from and to the IP core.
For example, to reset the IP core, write 0x1 to the bit 0 of IPCore_Reset register. To enable or disable the IP core, write 0x1 or 0x0 to the IPCore_Enable register. To access the data ports of the MATLAB/Simulink algorithm, read or write to the associated data registers.
Processing System
AXI4
AXI4 Accessible Registers
Algorithm from MATLAB/ Simulink
External Ports
Programmable Logic IP Core
This IP core also support the External Port interface. To connect the external ports to the FPGA external IO pins, add FPGA pin assignment constraints in the Microchip Libero SoC environment.
OK
Help

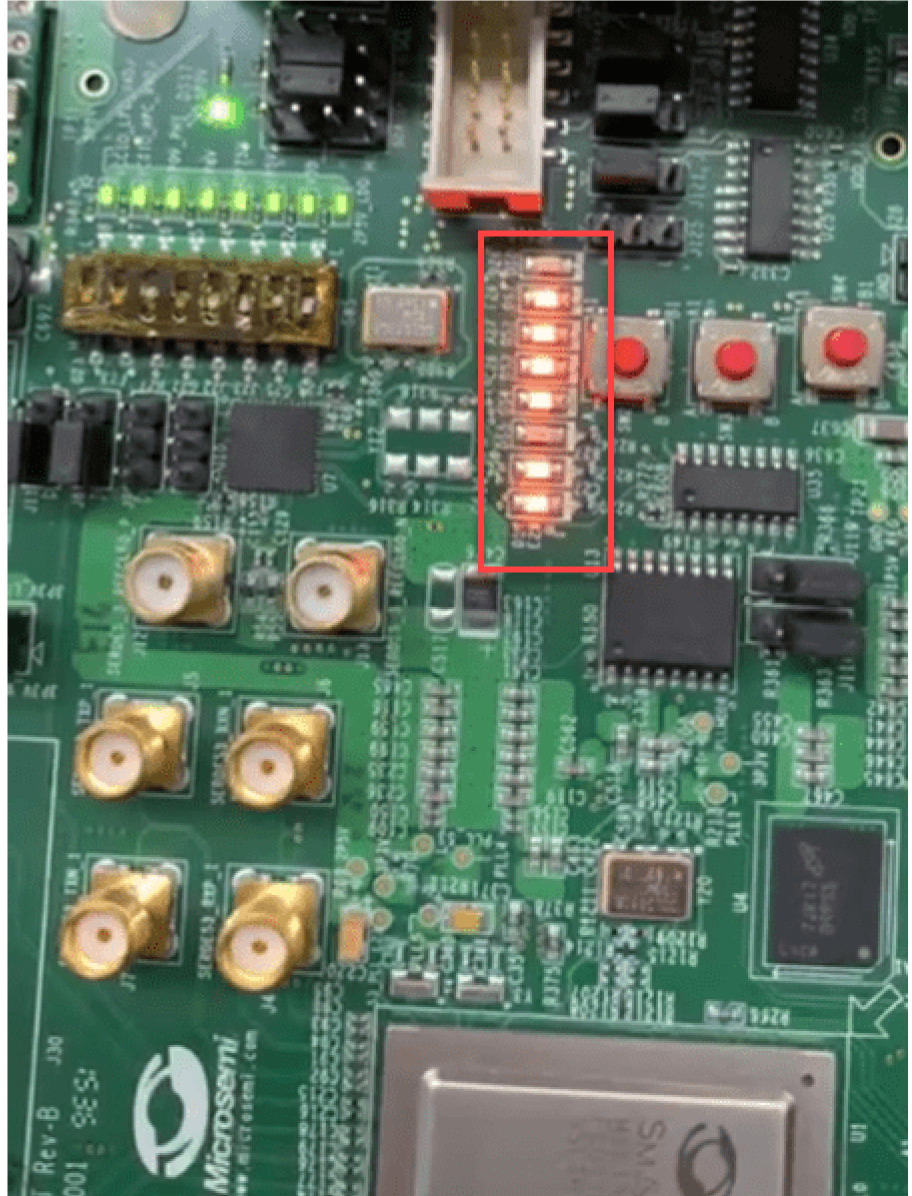
10. To integrate the IP core into the reference design and create the Libero project, follow step 1 of the **Integrate IP Core with Microsemi Libero SoC Environment** section in “Integrate HDL IP Core with Microchip PolarFire SoC Icicle Kit Reference Design” on page 39-161.

11. After completing the **Create Project** task under **Embedded System Integration**, examine the Microchip Libero SoC project. This figure shows the block design of the SoC project when you highlight the HDL IP Core. Compare this block design with the previous block design used to export the custom reference design for a deeper understanding of the relationship between a custom reference design and an HDL IP Core.



12. Follow steps 2 and 3 of the **Integrate IP Core with Microsemi Libero SoC Environment** section of “Integrate HDL IP Core with Microchip PolarFire SoC Icicle Kit Reference Design” on page 39-161 to generate the FPGA bitstream and program the target device, respectively.

13. After you load the bitstream, the LEDs on the SmartFusion2 board start blinking. You can change the direction of LED blinking by pressing **Slide Switch[0]**.



See Also

- `hdlsetuptoolpath`
- `hdlcoder.Board`
- `hdlcoder.ReferenceDesign`

Define Custom Board and Reference Design for Microchip Pure FPGA Platforms

This example shows how to define and register a Pure FPGA boards and reference design for a LED blinking model in the IP Core Generation workflow of the HDL Workflow Advisor. You use a Microchip PolarFire® board, but you can define and register a custom board or a custom reference design for other Microchip platforms.

Requirements

To run this example, you need access to:

- A Microchip Libero SoC Design Suite. For a list of supported versions, see “HDL Language Support and Supported Third-Party Tools and Hardware”.
- A Microchip PolarFire Splash Kit.
- HDL Coder™ Support Packages for Microchip FPGA and SoC Devices.

Set Up PolarFire Splash board

To familiarize yourself with the features of the PolarFire Splash kit, see the PolarFire Splash Kit Reference Manual. To set up your PolarFire Splash kit, follow these steps.

1. Install the USB COM port device drivers on your computer.

2. Connect the shared UART/JTAG USB port on the PolarFire Splash kit to your computer.

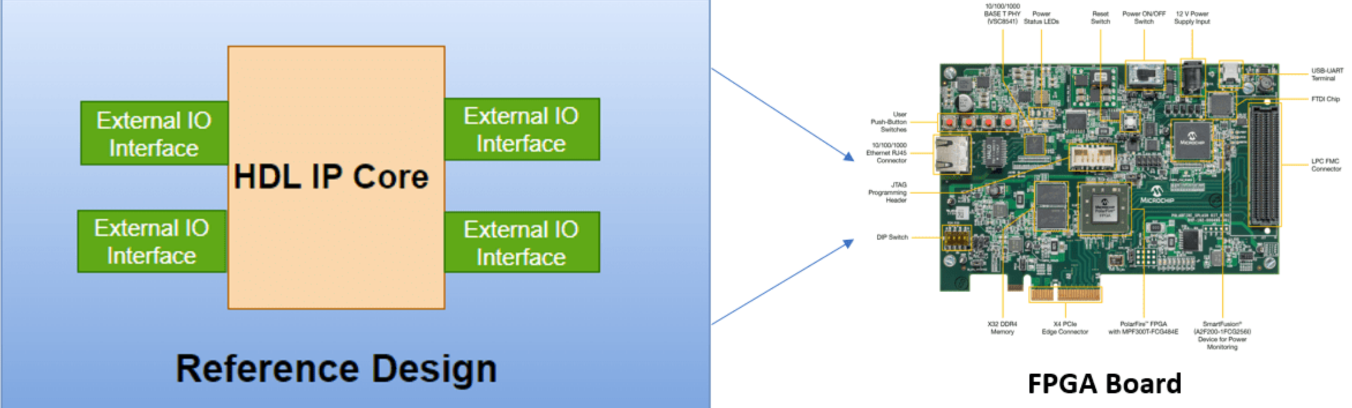
3. Download and install **HDL Coder Support Package for Microchip and SoC devices**. To access the hardware support package , in the **Home** tab of your MATLAB session, click **Add-Ons** and then click **Get Hardware Support Packages**.

4. Set up the Microchip Libero tool path by using the `hdlsetuptoolpath` function. For example, to set up the Microchip Libero SoC tool, specify `ToolName` as `Microchip Libero SoC` and `ToolPath` as your installed Libero executable path.

```
hdlsetuptoolpath('ToolName', 'Microchip Libero SoC', ...
    'ToolPath', 'C:\Microsemi\Libero_SoC_v2022.1\Designer\bin\libero.exe');
```

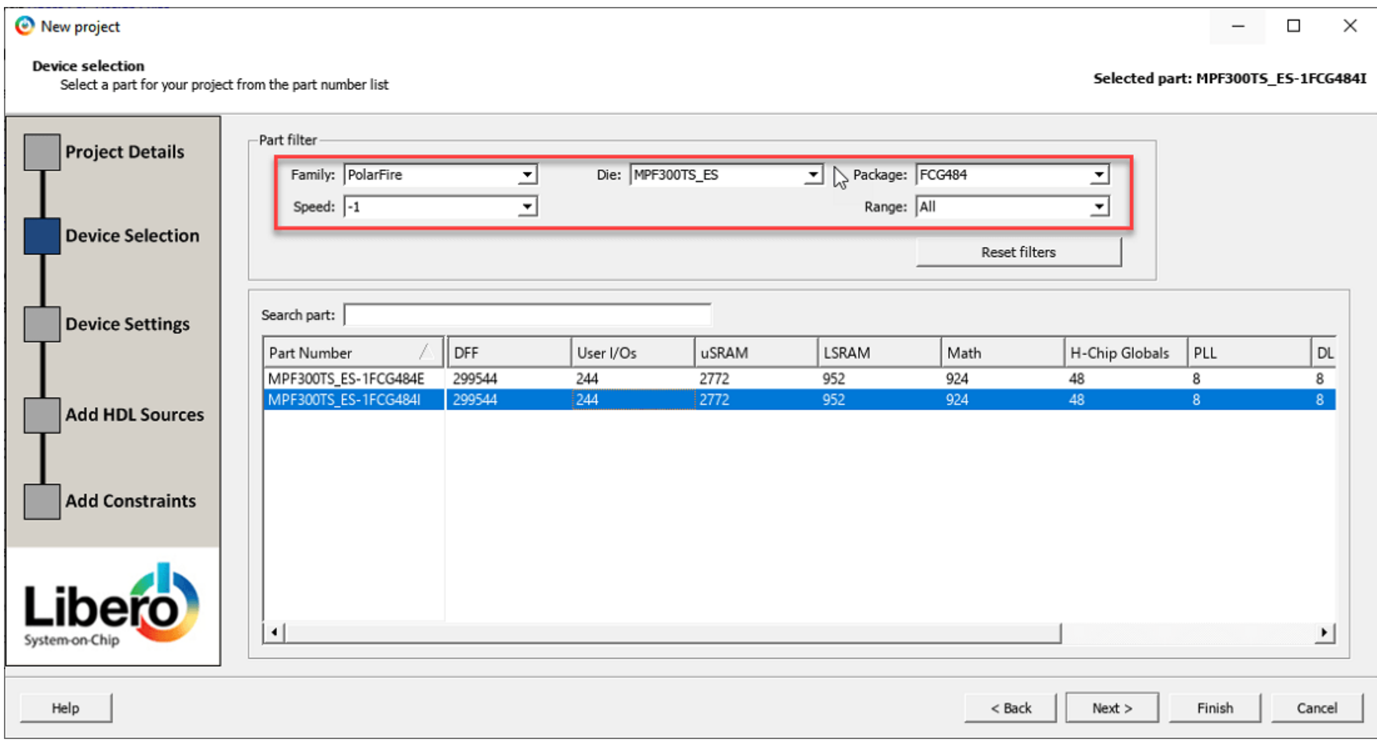
Create and Export Custom Reference Design by Using Microchip Libero SoC

A reference design captures the complete structure of an SoC design, defining the different components and their interconnections. Use the HDL Coder Reference Design workflow in the HDL Workflow Advisor to generate an IP core that integrates with the reference design and program FPGA board. This figure shows the relationship between a reference design, an HDL IP core, and FPGA board.

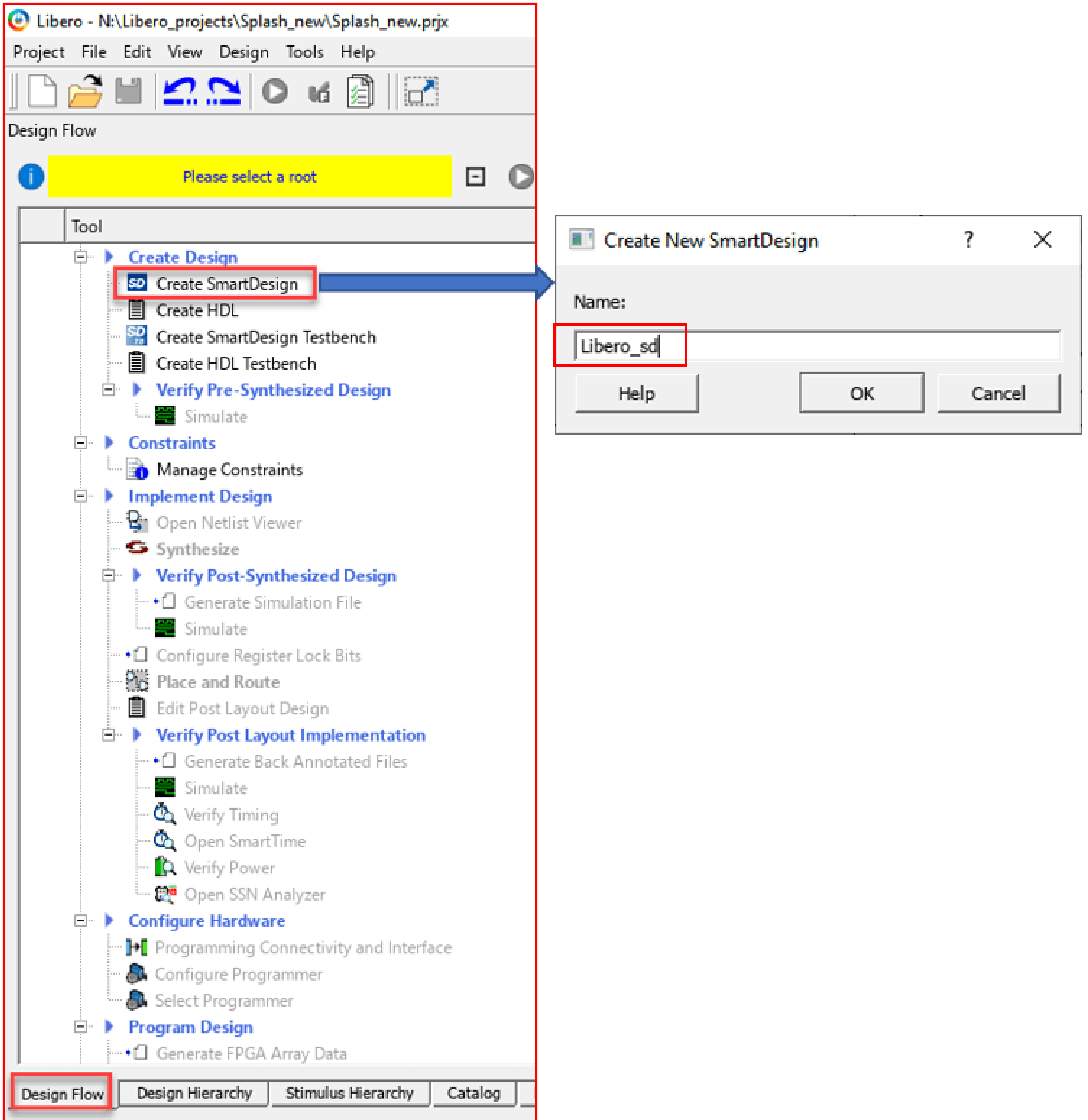


To create and export a reference design by using the Microchip Libero Tool environment, follow these steps.

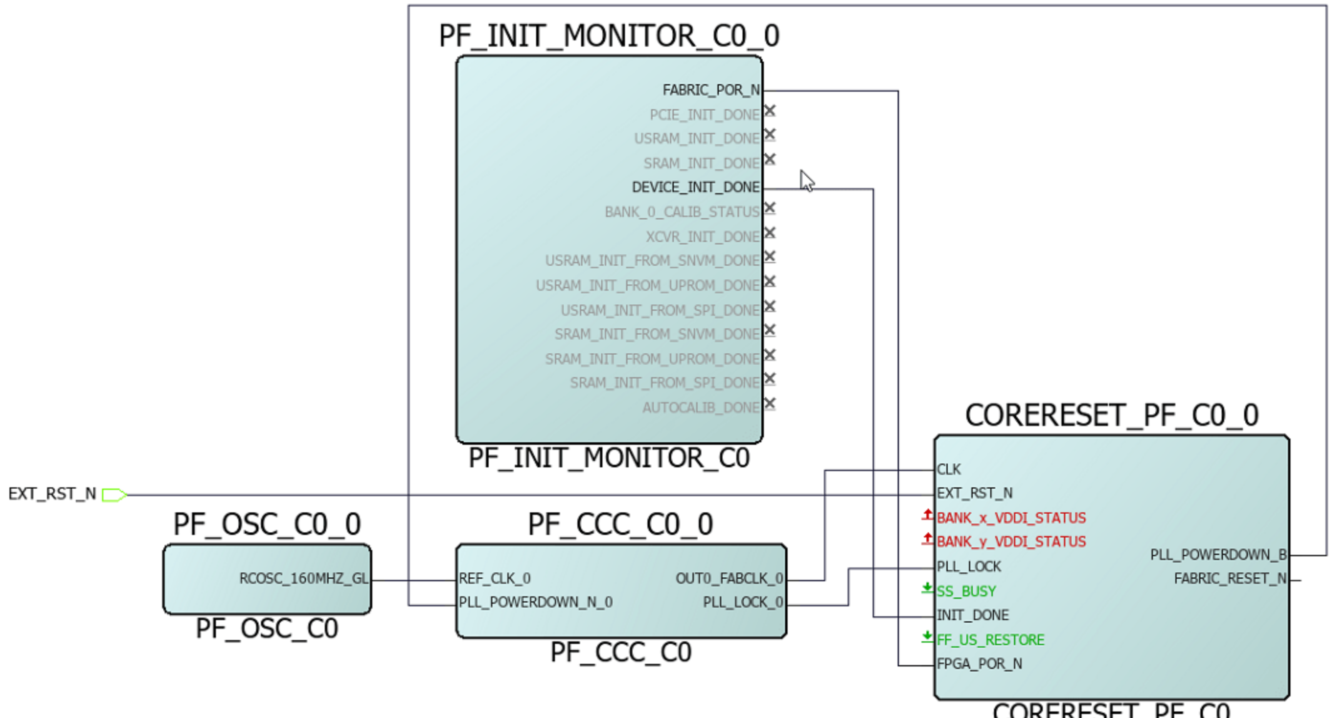
1. Create an empty Microchip Libero project by using below settings for PolarFire family and Click **Next**. Click **Finish** to create a new Libero project.



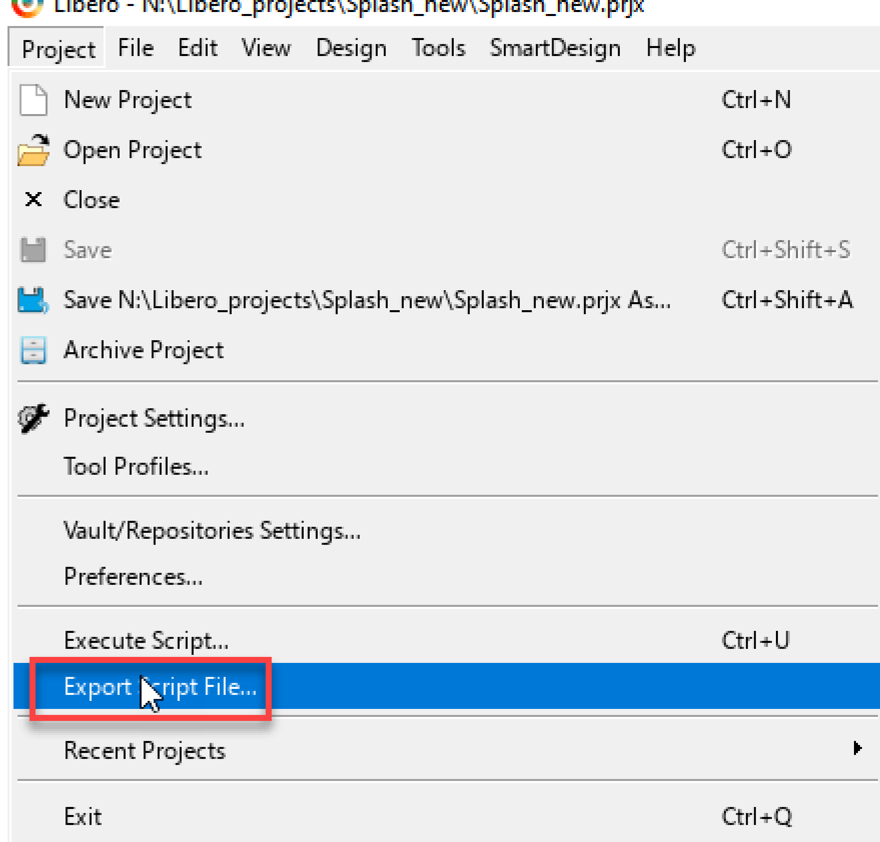
2. Create new Smartdesign from **Design Flow** tab.



3. Add **PF_OSC_CO**, **PF_CCC_C0**, **CORERESET_PF_C0** and **PF_INIT_MONITOR_C0** IP's from IP Catalog to **Libero_sd** and make connections as shown below.



4. Export the completed block design as a Tcl script with the name `design1_led.tcl`.



The exported Tcl script (`design1_led.tcl`) constitutes the custom reference design. You use this reference design in the HDL Coder Reference Design workflow of the HDL Workflow Advisor to recreate the block design and integrate the generated HDL IP core with the block design in a Microchip Libero project.

Register PolarFire Splash Board in HDL Workflow Advisor

To register the PolarFire Splash Kit in the HDL Workflow Advisor, follow these steps.

1. Create a board registration file with the name `hdlcoder_board_customization` and add it to the MATLAB path.

A board registration file contains a list of board plugins. A board plugin is a MATLAB package folder containing a board definition file and all the reference design plugins associated with the board.

This code describes the contents of a board registration file that contains `PolarFireRegistration` board plugin to register the PolarFire Splash Kit in HDL Workflow Advisor. The function finds any registration file with the specified name on the MATLAB path and returns a cell array with the locations of the board plugins. The board plugin must be a package folder that is accessible from your MATLAB path containing a board definition file.

```
function r = hdlcoder_board_customization
% Board plugin registration file

r = { ...
      'PolarFireRegistration.plugin_board', ...
```

```
    };
end
```

2. Create the board definition file.

A board definition file contains information about the FPGA board.

Create a PolarFire Splash board definition file named `plugin_board.m` that resides inside the board plugin `PolarFireRegistration`.

For information about the FPGA I/O pin locations (`'FPGAPin'`) and standards (`'IOSTANDARD'`), see the PolarFire Splash Kit constraints file from the Microchip website.

The property `BoardName` defines the name of the PolarFire board which will appear in the HDL Workflow Advisor.

```
function hB = plugin_board()
% Board definition


% Construct board object
hB = hdlcoder.Board;

hB.BoardName    = 'Microchip PolarFire Splash Kit (ES)';


% FPGA device information
hB.FPGAVendor   = 'Microchip';
hB.FPGAFamily   = 'PolarFire';
hB.FPGADevice   = 'MPF300TS_ES';
hB.FPGAPackage  = 'FCG484';
hB.FPGASpeed    = '-1';


% Tool information
hB.SupportedTool = {'Microchip Libero SoC'};


% FPGA JTAG chain position
hB.JTAGChainPosition = 2;


% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
   'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});


% Custom board external I/O interface
hB.addExternalIOInterface( ...
   'InterfaceID',    'LEDs General Purpose', ...
   'InterfaceType',  'OUT', ...
   'PortName',       'LEDs', ...
   'PortWidth',      8, ...
   'FPGAPin',        {'P7', 'P8', 'N7', 'N8', 'N6', 'N5', 'M8', 'M9'}, ...
   'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
   'InterfaceID',    'Push Buttons', ...
```

```
   'InterfaceType',  'IN', ...
   'PortName',       'PushButtons', ...
   'PortWidth',      4, ...
   'FPGAPin',        {'L6', 'M7', 'K5', 'K4'}, ...
   'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
  'InterfaceID',    'Slide switches ', ...
  'InterfaceType',  'IN', ...
  'PortName',       'SlideSwitches', ...
  'PortWidth',      4, ...
  'FPGAPin',        {'L3', 'M4', 'J6', 'K6'}, ...
  'IOPadConstraint', {'IOSTANDARD = LVCMOS25'});
```

Register Custom Reference Design in HDL Workflow Advisor

To register the custom reference design in HDL Workflow Advisor, follow these steps.

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` containing a list of reference design plugins associated with an SoC board.

A reference design plugin is a MATLAB package folder containing the reference design definition file and all the files associated with the FPGA design project. A reference design registration file must also contain the name of the associated board.

This code describes the contents of a PolarFire Splash reference design registration file containing the reference design plugin `PolarFireRegistration.Libero_12_6` associated with the `Microchip PolarFire Splash Kit (ES)` board. The registration file finds files with the specified name inside a board plugin folder or on the MATLAB path. The function returns a cell array containing the locations of the reference design plugins and a character vector containing the associated board name. The reference design plugin must be a package folder that is accessible from the MATLAB path and must contain a reference design definition file.

```
function [rd,boardName] = hdlcoder_ref_design_customization
% Reference design plugin registration file

rd = {'PolarFireRegistration.Libero_12_6.plugin_rd', ...
     };

boardName = 'Microchip PolarFire Splash Kit (ES)';
end
```

2. Create the reference design definition file.

A reference design definition file defines the interfaces between the custom reference design and the HDL IP core that you generate.

Create a PolarFire reference design definition file `plugin_rd.m` to associate with the `Microchip PolarFire Splash Kit (ES)` board that resides inside the reference design plugin `PolarFireRegistration.Libero_12_6`. The `ReferenceDesignName` property defines the name of the reference design as `Default system` in HDL Workflow Advisor.

```
function hRD = plugin_rd()
% Reference design definition
```

```
% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Microchip Libero SoC');

hRD.ReferenceDesignName = 'Default system';
hRD.BoardName = 'Microchip PolarFire Splash Kit (ES)';


% Tool information
hRD.SupportedToolVersion = {'12.0','12.6','2022.1'};


% Add custom design files
% add custom Libero  design
hRD.addCustomLiberoDesign( ...
   'CustomBlockDesignTcl', 'design1_led.tcl');


% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',     'PF_CCC_C0_0/OUT0_FABCLK_0', ...
    'ResetConnection',     'CORERESET_PF_C0_0/FABRIC_RESET_N',...
    'DefaultFrequencyMHz', 50,...
    'MinFrequencyMHz',     5,...
    'MaxFrequencyMHz',     500,...
    'ClockModuleInstance', 'PF_CCC_C0_0',...
    'ClockModuleComponent','PF_CCC_C0',...
    'ClockNumber',         1);


% disable 'Generate Software Interface Model' task
hRD.HasProcessingSystem = false;
```

A reference design plugin must also contain the FPGA design project files.

The PolarFire reference design plugin folder `PolarFireRegistration.Libero_12_6` must contain the Tcl script `design1_led.tcl` exported from the Microchip Libero project. The PolarFire reference design definition file `plugin_rd.m` identifies the FPGA design project file by using the `addCustomLiberoDesign` function.

```
hRD.addCustomLiberoDesign('CustomBlockDesignTcl', 'design1_led.tcl');
```

The reference design definition file `plugin_rd.m` also defines the interface connections between the custom reference design and the HDL IP core by using the `addClockInterface` function:

```
hRD.addClockInterface( ...
   'ClockConnection',     'PF_CCC_C0_0/OUT0_FABCLK_0', ...
   'ResetConnection',     'CORERESET_PF_C0_0/FABRIC_RESET_N',...
```

Generate HDL IP core for PolarFire Splash Kit

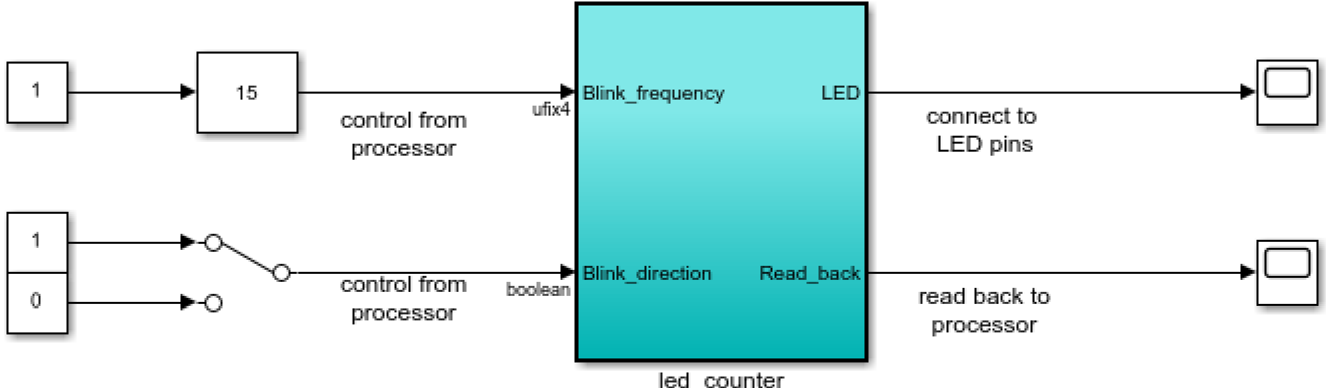
Use the custom board and reference design registration system to generate an HDL IP core that blinks LEDs on the PolarFire Splash Kit.

1. Add the PolarFire board registration files to the MATLAB path using these commands.

```
hdlcoder_microchip_examples_root;
addpath(fullfile(hdlcoder_microchip_examples_root,'PolarFire'));
```

2. Open the Simulink LED blinking model.

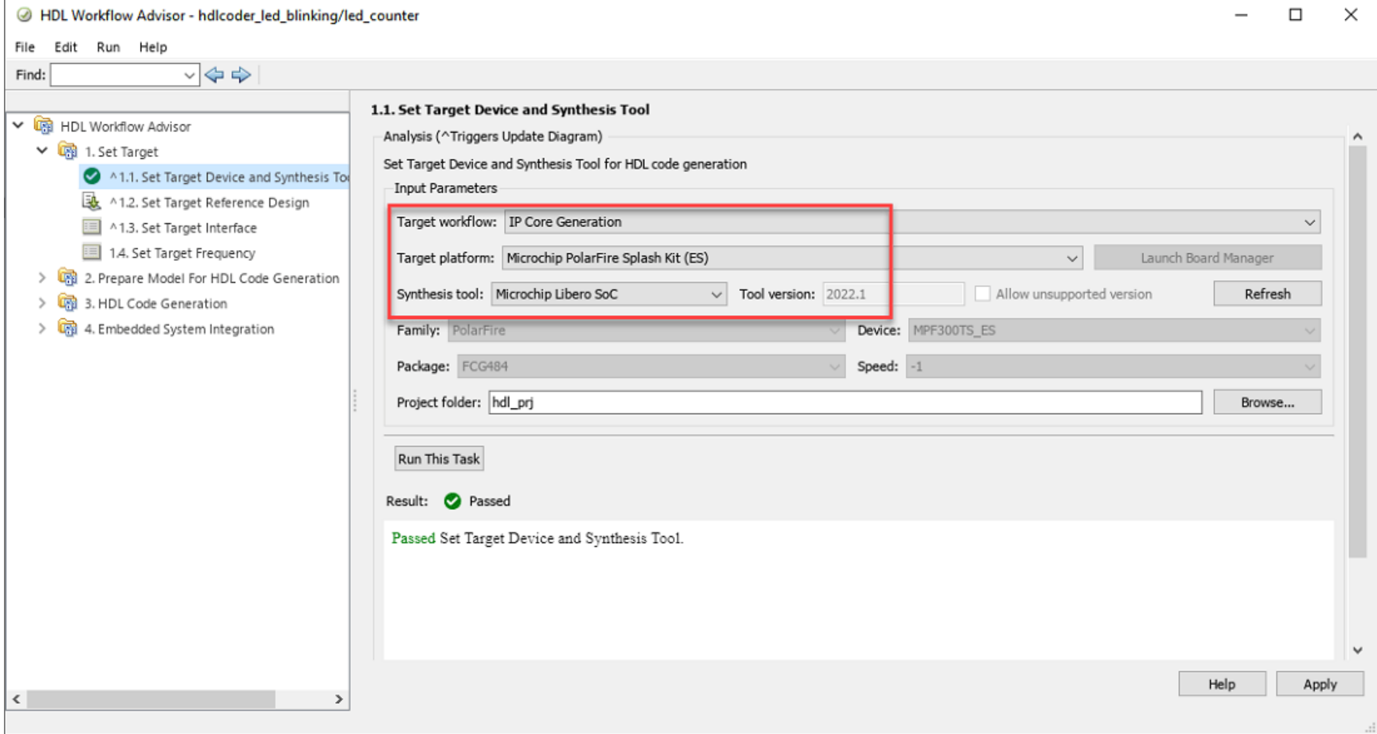
```
open_system('hdlcoder_led_blinking');
```



Copyright 2014-2023 The MathWorks, Inc.

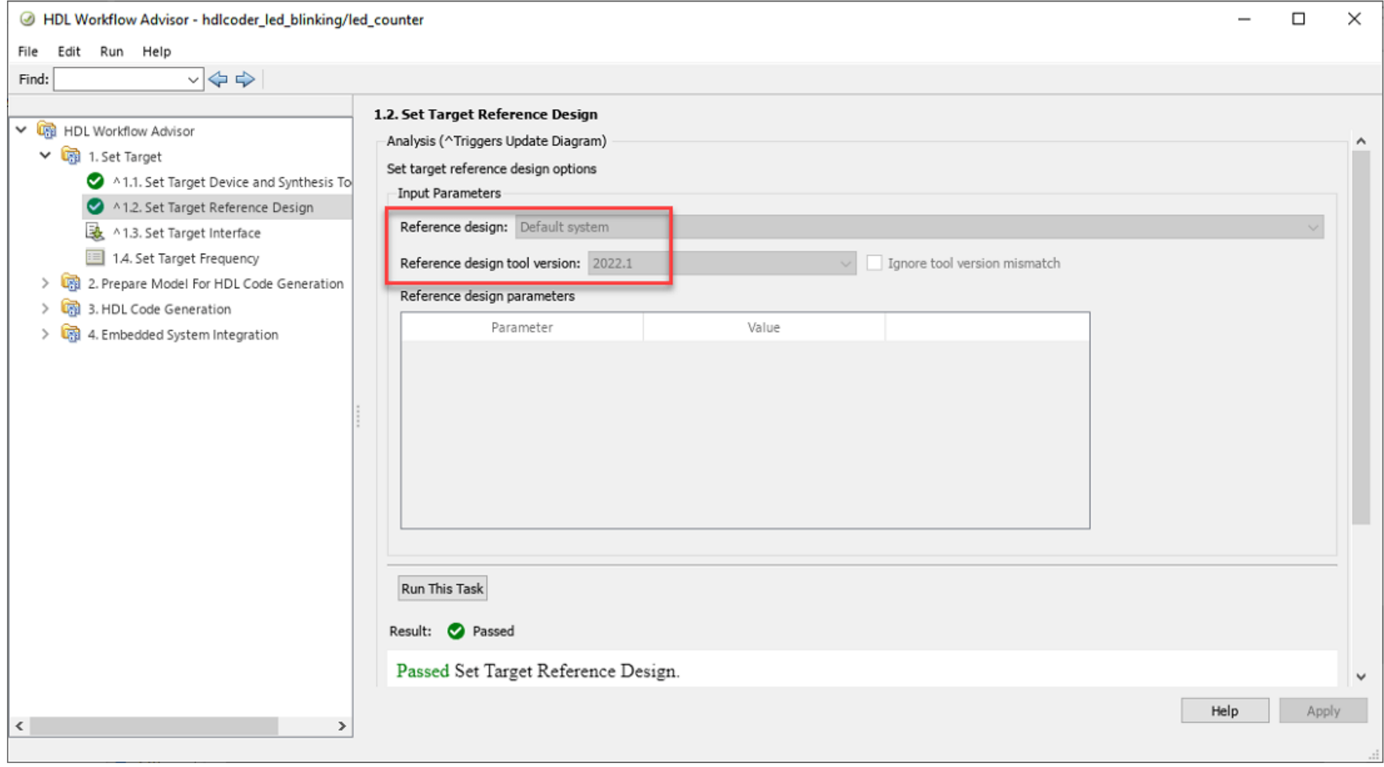
3. Launch the HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem, and hovering over **HDL Code**, and then clicking **HDL Workflow Advisor**. Alternatively, click the **Launch HDL Workflow Advisor** box in the model.

In the **Set Target** > **Set Target Device and Synthesis Tool** task, set **Target workflow** to `IP Core Generation`. In the **Target Platform** dropdown menu, select `Microchip PolarFire Splash Kit (ES)`.

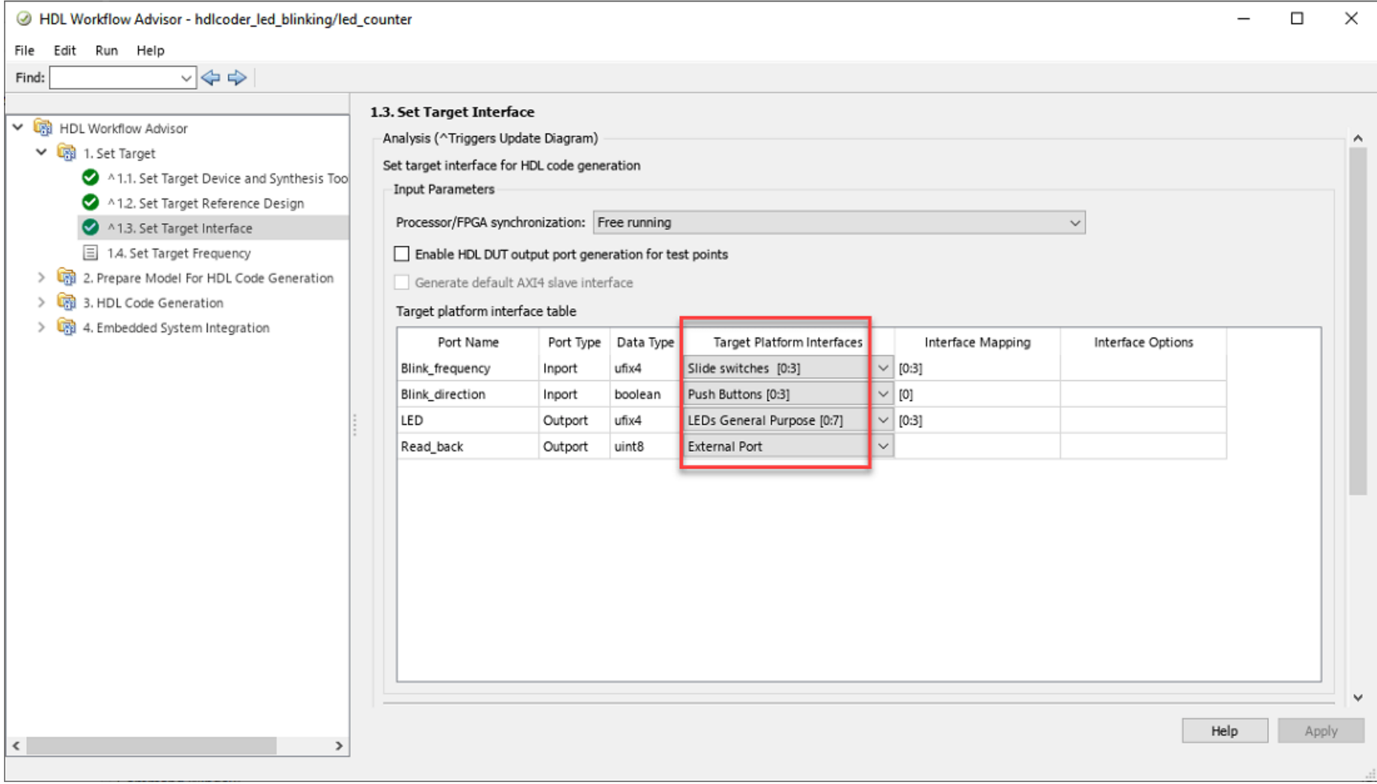


5. Click **Run This Task** to complete the **Set Target Device and Synthesis Tool** task.

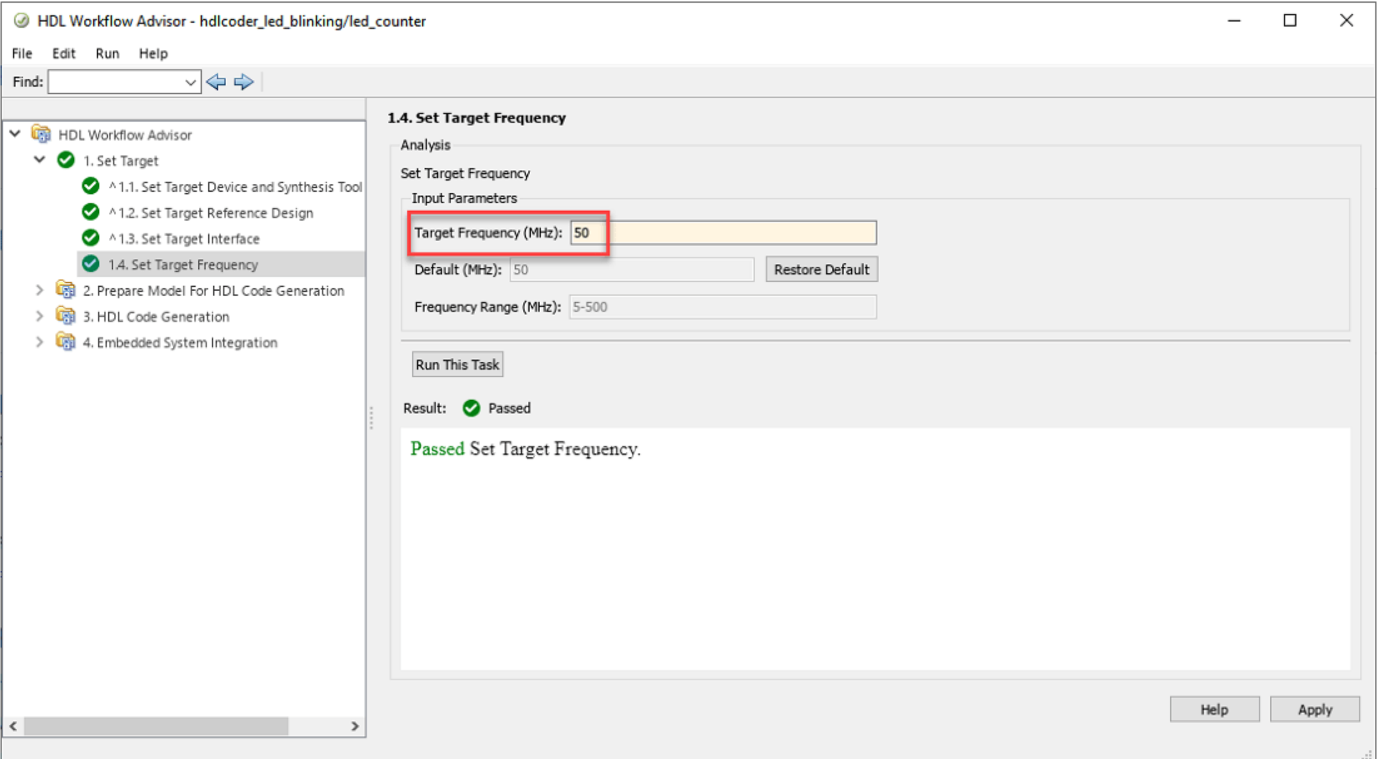
6. In the **Set Target > Set Target Reference Design** task, set the **Reference design** field to `Default system` and click **Run This Task**.



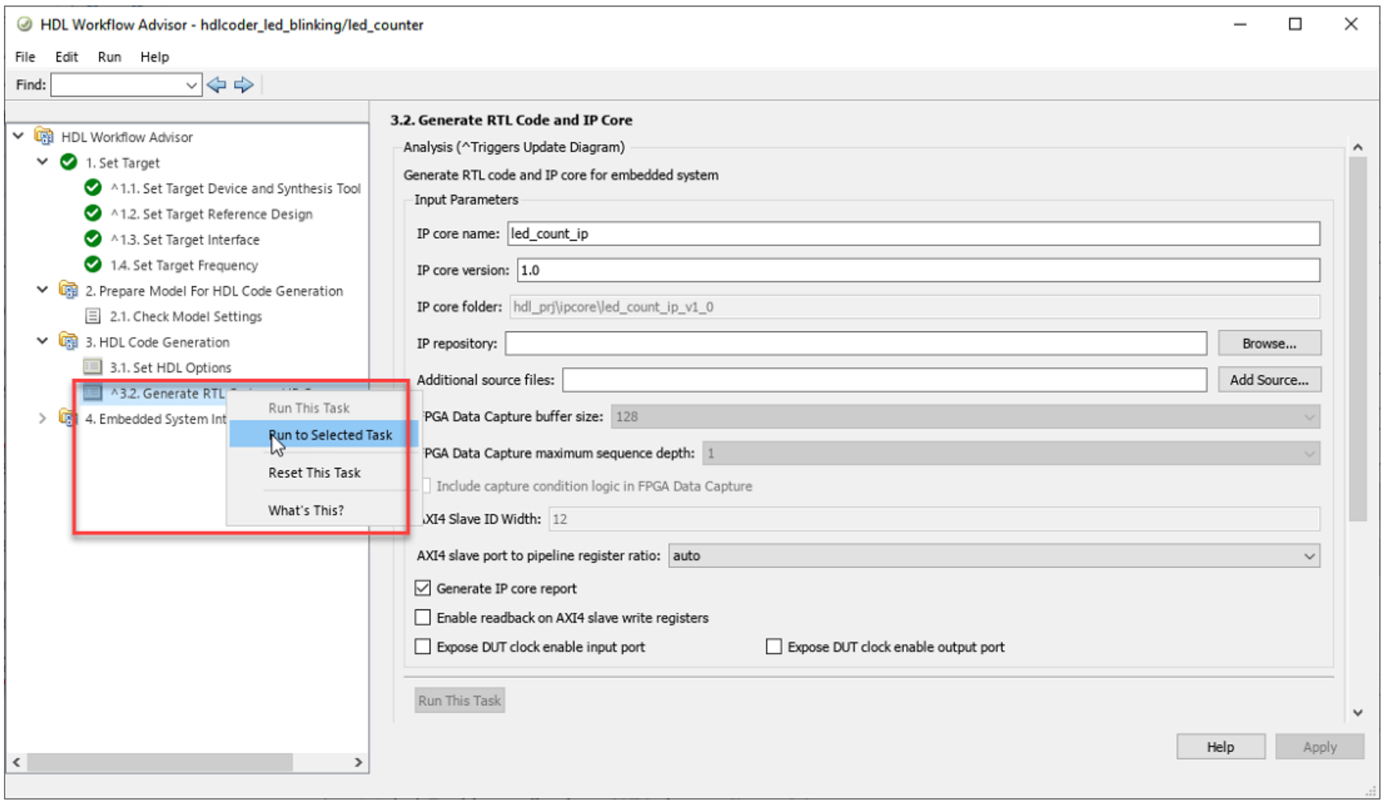
7. In the **Set Target Interface** task, set the Target Platform Interface options and click **Run This Task**.

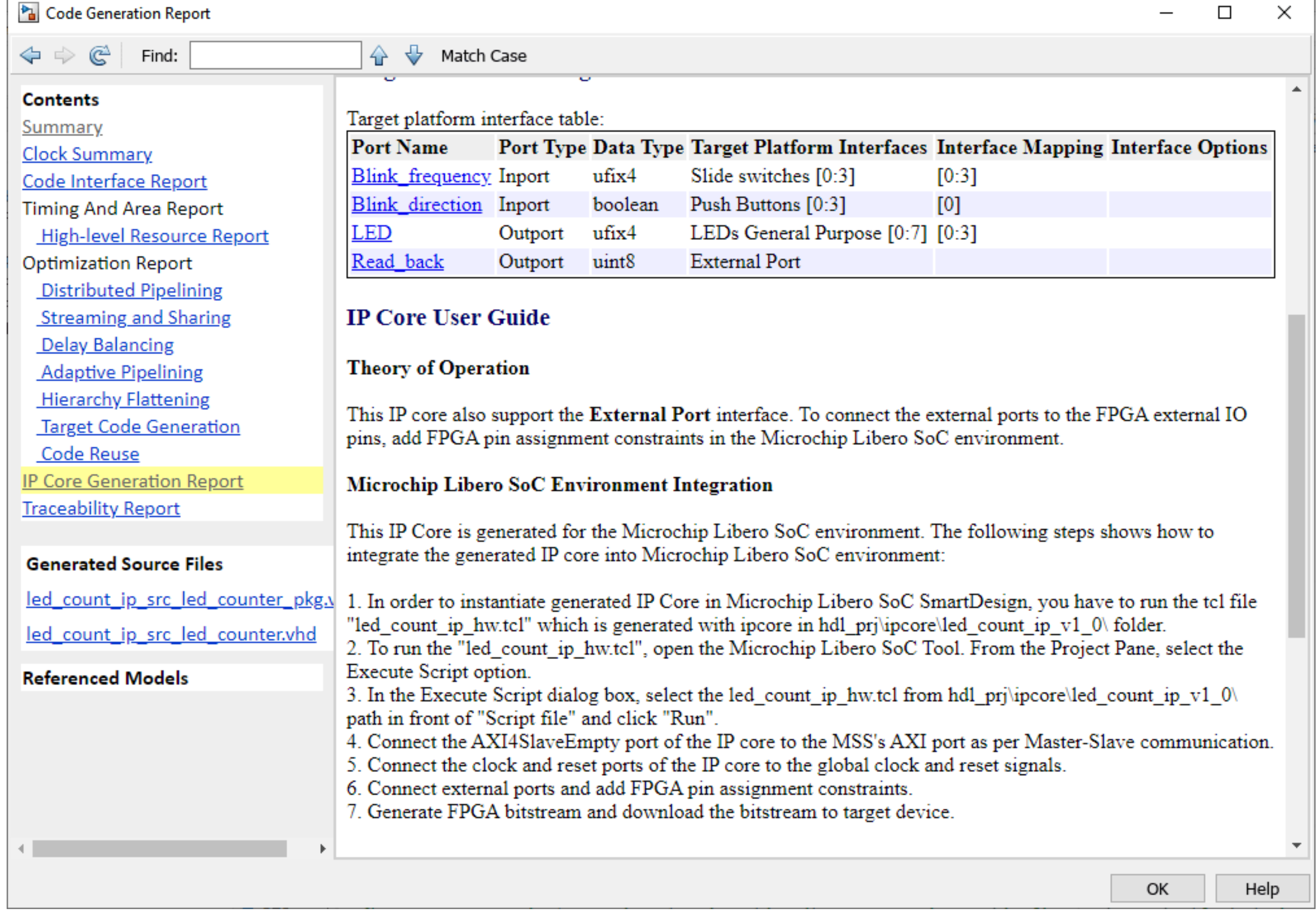


8. In the **Set Target Frequency** task, set **Target Frequency** to 50 MHz.



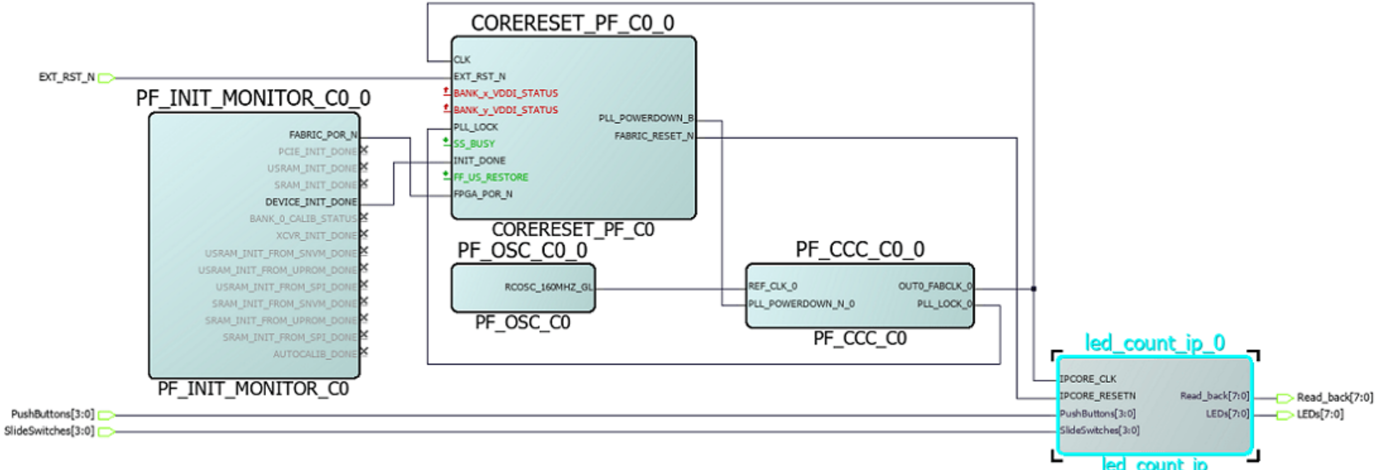
9. To generate the IP core and view the IP core generation report, right-click **Generate RTL Code and IP Core** and select **Run to Selected Task**. After you generate the custom IP core, the IP core files are in the `ipcore` folder within your project folder. An HTML custom IP core report is generated with the custom IP core. The report describes the behavior and contents of the generated custom IP core.





10. To integrate the IP core into the reference design and create the Libero project, follow step 1 of the **Integrate IP Core with Microsemi Libero SoC Environment** section in “Integrate HDL IP Core with Microchip PolarFire SoC Icicle Kit Reference Design” on page 39-161.

11. After completing the **Create Project** task under **Embedded System Integration**, examine the Microchip Libero SoC project. This figure shows the block design of the FPGA project when you highlight the HDL IP Core. Compare this block design with the previous block design used to export the custom reference design for a deeper understanding of the relationship between a custom reference design and an HDL IP Core.



12. Follow steps 2 and 3 of the **Integrate IP Core with Microsemi Libero SoC Environment** section of “Integrate HDL IP Core with Microchip PolarFire SoC Icicle Kit Reference Design” on page 39-161 to generate the FPGA bitstream and program the target device, respectively.

13. After you load the bitstream, the LEDs on the PolarFire Splash Kit start blinking. You can change the direction of LED blinking by pressing **Push Buttons[0]**.

See Also

- `hdlsetuptoolpath`
- `hdlcoder.Board`
- `hdlcoder.ReferenceDesign`

Dynamically Create Reference Design with Master Only or Slave Only AXI4-Stream Interface

This example illustrates how to customize the reference design dynamically by using a callback function based on the reference design parameter options. Also, this example shows how you can customize the number of AXI4-Stream interface channels in your reference design to be Only AXI4-Stream Master interface, Only AXI4-Stream Slave interface or both the interfaces.

Prerequisites

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder™ Support Package for Xilinx® FPGA and SoC Devices
- Embedded Coder® Support Package for Xilinx® Zynq® Platform
- Xilinx® Vivado® Design Suite, with supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- Xilinx® Zynq®-7000 SoC ZC706 Evaluation Kit
- Latest SD image from ECoder support package setup wizard

Introduction

Instead of creating multiple reference designs that have different interface choices, data widths, or I/O plugins now you have an option of creating one reference design that has different interface choices or data widths as parameters. You can use the `CustomizeReferenceDesignFcn` method to reference the callback function that has different choices for interfaces or data widths in your reference design. For example, you can create one reference design and select the reference design parameter choice that has only AXI4-Stream Master or only AXI4-Stream Slave or both AXI4-Stream Master and AXI4-Stream Slave interfaces instead of creating three separate reference designs.

Create Reference Design Definition File and Callback Function

For this demo, you can consider Xilinx® Zynq® ZC706 AXI4-Stream reference design which consists of two Xilinx® AXI DMAs to handle the data transfer from Processor to FPGA and vice versa using AXI4-Stream Master and Slave interfaces. In this example you customize the number of AXI4-Stream interface channels with a single reference design by creating a callback function. The different interface channels in the callback code shown below uses the different device tree. Similarly, you can modify your created reference design definition file to select different reference design parameter choices using the callback function.

The picture below shows the **Stream_Channel** reference design parameter and interface choices for the **AXI4-Stream interface with Stream channel Selection** reference design specified by using the `addParameter` method. The `CustomizeReferenceDesignFcn` method references a callback function that has the name callback_CustomizeReferenceDesign.

```
function hRD = plugin_rd()
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.ReferenceDesignName = 'AXI4-Stream interface with Stream channel Selection';
hRD.BoardName = 'Xilinx Zynq ZC706 evaluation kit';
% Tool information
hRD.SupportedToolVersion = {'2018.2','2018.3','2019.1','2019.2'};
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',      'core_clkwiz/clk_out1', ...
    'ResetConnection',      'sys_core_rstgen/peripheral_aresetn',...
    'DefaultFrequencyMHz', 50,...
    'MinFrequencyMHz',      5,...
    'MaxFrequencyMHz',      500,...
    'ClockModuleInstance', 'core_clkwiz',...
    'ClockNumber',          1);
% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_cpu_interconnect/M00_AXI', ...
    'BaseAddress',          '0x40010000', ...
    'MasterAddressSpace',  'sys_cpu/Data');
hRD.addParameter( ...
    'ParameterID',    'StreamChanel', ...
    'DisplayName',    'Stream Channel', ...
    'DefaultValue',   'Both Master & Slave', ...
    'ParameterType',   hdlcoder.ParameterType.Dropdown, ...
    'Choice',         {'Master Only','Slave Only','Both Master & Slave'});
hRD.CustomizeReferenceDesignFcn  = ...
    @Stream_ChannelSelection.callback_CustomizeReferenceDesign;
```

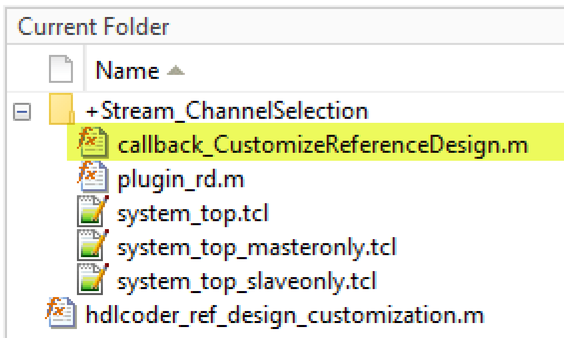
The code below shows the callback function `callback_CustomizeReferenceDesign` that has the AXI4-Stream Master or Slave Channels or both channels specified by using the `addAXI4StreamInterface` method. The `DeviceTreeName` method shown below in the callback is to specify the device tree file, which is different for different stream channels.

```
function callback_CustomizeReferenceDesign(infoStruct)
% Reference design callback run at the end of the task Set Target Reference Design


% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in struct format
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.ReferenceDesignToolVersion: Reference design Tool Version set in 1.2 Task

hRD = infoStruct.ReferenceDesignObject;
paramStruct = infoStruct.ParameterStruct;
```

```
% get the reference design parameter value
ParamValue = paramStruct.StreamChanel;

% Add the reference design interface into interface list table baed on the
% reference design Parameter value

if strcmp(ParamValue, 'Both Master & Slave')
 % add custom Vivado design
 hRD.addCustomVivadoDesign( ...
     'CustomBlockDesignTcl', 'system_top.tcl', ...
     'VivadoBoardPart',      'xilinx.com:zc706:part0:1.0');
 hRD.addAXI4StreamInterface( ...
     'MasterChannelEnable',      true, ...
     'SlaveChannelEnable',       true, ...
     'MasterChannelConnection', 'axi_dma_s2mm/S_AXIS_S2MM', ...
     'SlaveChannelConnection',  'axi_dma_mm2s/M_AXIS_MM2S', ...
     'MasterChannelDataWidth',   32, ...
     'SlaveChannelDataWidth',    32, ...
     'HasDMAConnection',         true);
 hRD.DeviceTreeName = 'devicetree_axistream_iio.dtb';

elseif strcmp(ParamValue, 'Master Only')
 % Block design TCl for Master Only reference design
 hRD.addCustomVivadoDesign( ...
     'CustomBlockDesignTcl', 'system_top_masteronly.tcl', ...
     'VivadoBoardPart',      'xilinx.com:zc706:part0:1.0');

  hRD.addAXI4StreamInterface( ...
      'MasterChannelEnable',      true, ...
      'SlaveChannelEnable',       false, ...
      'MasterChannelConnection', 'axi_dma_s2mm/S_AXIS_S2MM', ...
      'MasterChannelDataWidth',   32, ...
      'HasDMAConnection',         true);
  hRD.DeviceTreeName = 'devicetree_axistream_MasterOnly_iio.dtb';

elseif strcmp(ParamValue, 'Slave Only')
 % Block design TCl for Slave Only reference design
 hRD.addCustomVivadoDesign( ...
     'CustomBlockDesignTcl', 'system_top_slaveonly.tcl', ...
     'VivadoBoardPart',      'xilinx.com:zc706:part0:1.0');

  % add AXI4-Stream Slave only interface
  hRD.addAXI4StreamInterface( ...
      'MasterChannelEnable',      false, ...
      'SlaveChannelEnable',       true, ...
      'SlaveChannelConnection',  'axi_dma_mm2s/M_AXIS_MM2S', ...
      'SlaveChannelDataWidth',    32, ...
      'HasDMAConnection',         true);
  hRD.DeviceTreeName = 'devicetree_axistream_SlaveOnly_iio.dtb';
  end
 end
```

So, create the callback function like as shown above and pass the infoStruct argument to the callback function. The argument contains reference design customization information in a structure format.

Save the created callback function in the reference design folder like as shown below or you can save anywhere and add the file to MATLAB path.



Generate HDL IP core with Only AXI4-Stream Master/Only AXI4-Stream Slave Interface

1.Set up the Xilinx® Vivado® synthesis tool path using the following command in the MATLAB(r) command window. Use your own Vivado® installation path when you run the command.

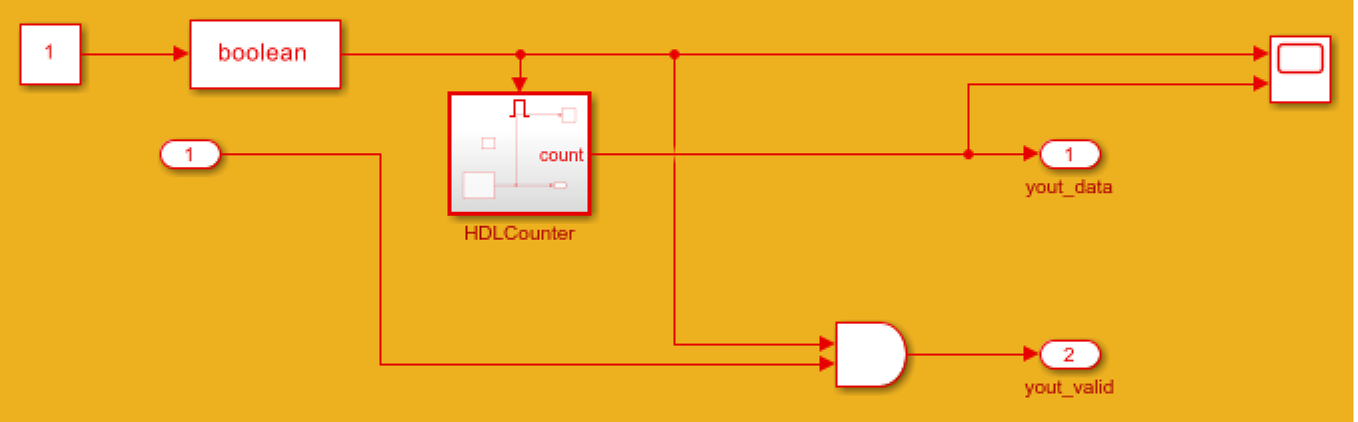
```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.ba
```

2. Add the demo reference design folder to the MATLAB path using following command:

```
example_root = (hdlcoder_amd_examples_root)
cd (example_root)
addpath(genpath('ZC706'));
```

3. Open AXI4-Stream Master only model using following command:

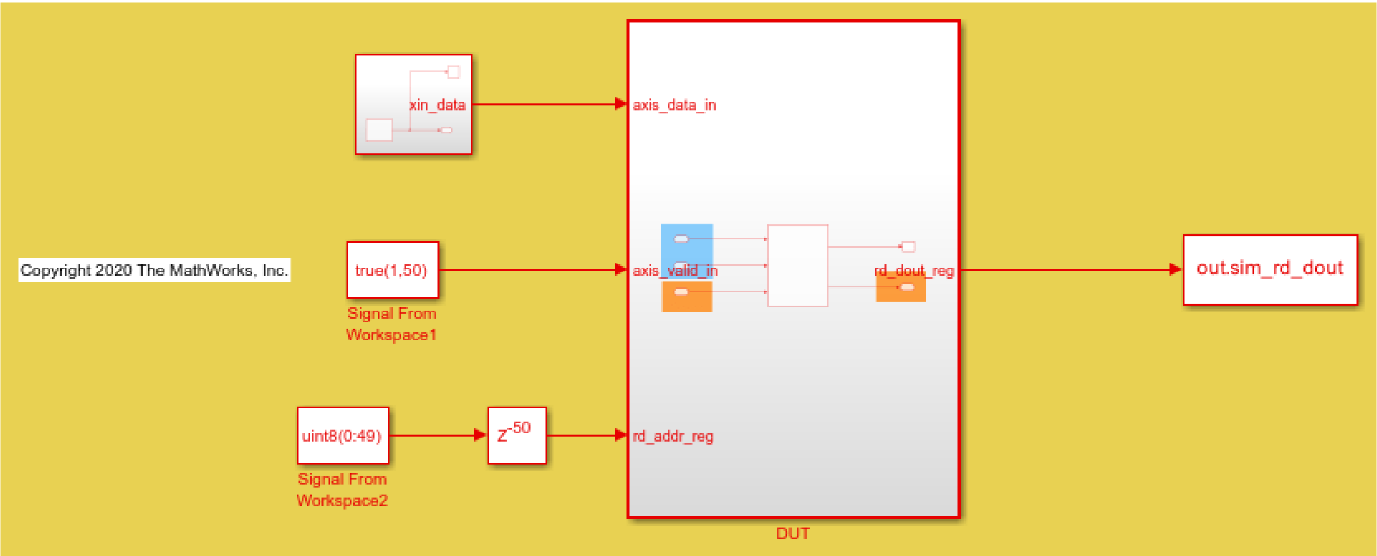
```
open_system('hdlcoder_AXI4StreamMaster');
```



The **DUT** is the hardware subsystem targeting the FPGA fabric. Inside this DUT, the **HDL Counter** subsystem acts as master. This counter counts from 1 to 50 and is connected to **yout_data** output signal which is mapped to AXI4-Stream master interface.

You can also use AXI4 Slave only model. Use the following command to open the model:

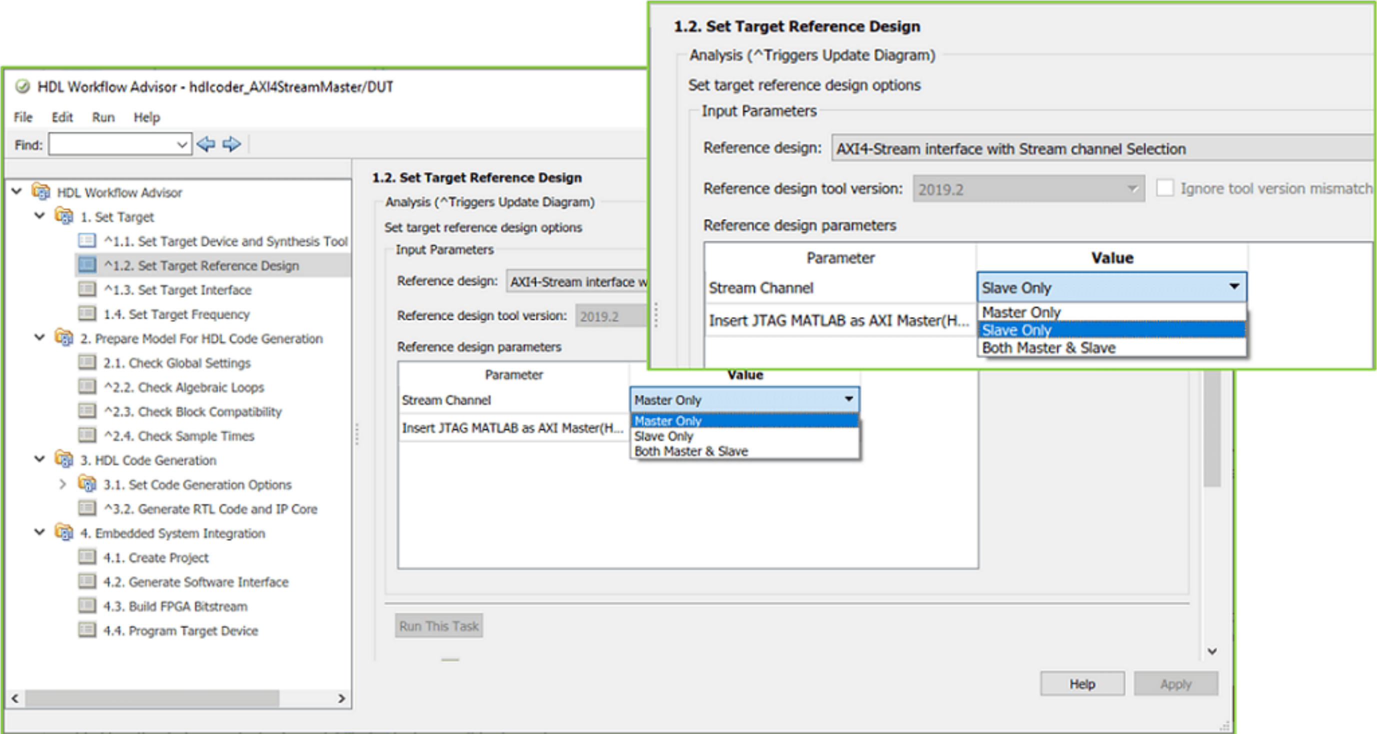
```
open_system('hdlcoder_AXI4StreamSlave');
```



As shown above **DUT** has a **Dual Port RAM** which acts as slave and receives data using AXI4-Stream Slave interface through **axis_data_in** input signal.

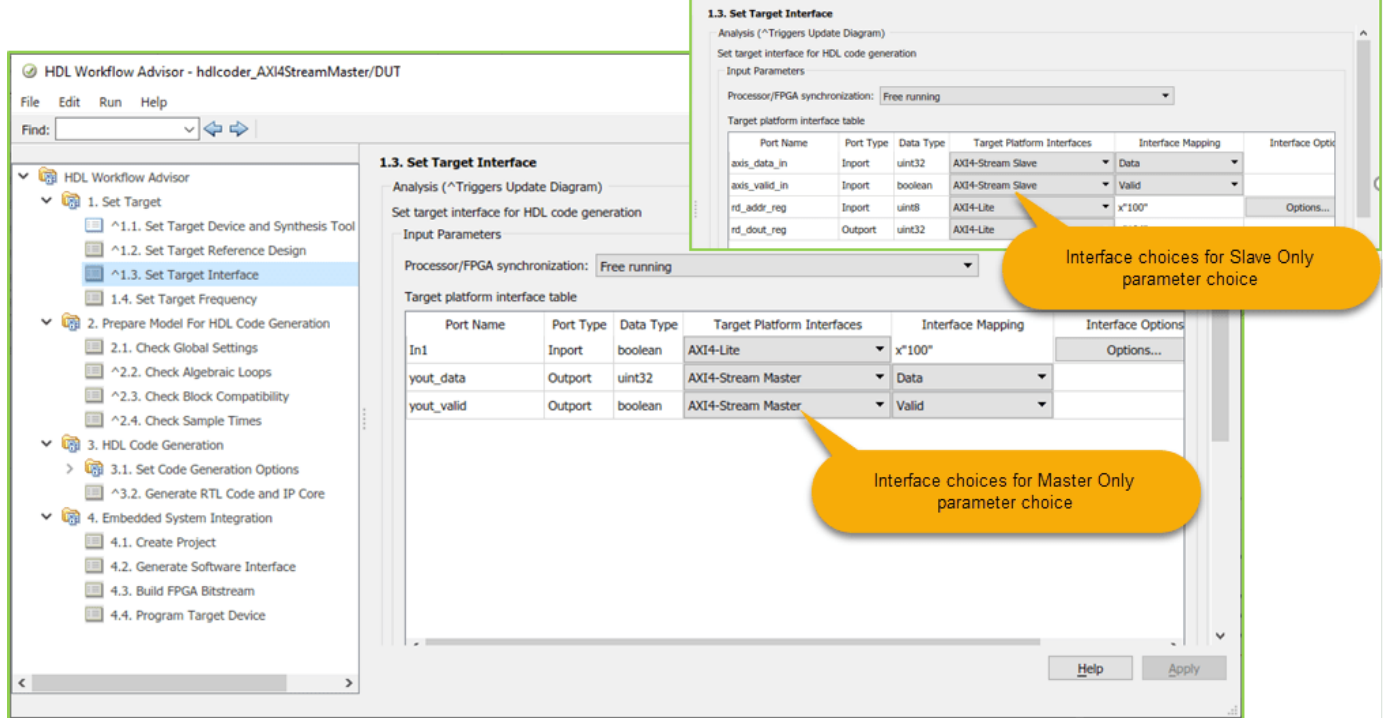
4. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_AXI4StreamMaster/DUT` for AXI4-Stream Master only demo. Similarly Open HDL Workflow Advisor from the DUT subsystem, `hdlcoder_AXI4StreamSlave/DUT` for AXI4-Stream Slave only demo.

The target interface settings are already saved for **ZC706** in these models, so the settings in Task 1.1 to 1.3 are automatically loaded. In Task 1.1, **IP Core Generation** is selected for Target workflow, and **Xilinx Zynq ZC706 evaluation kit** is selected for Target platform. In task 1.2, **AXI4-Stream interface with Stream channel Selection** is selected for reference design. Select **Stream Channel** reference design parameter choice as `Master Only` for AXI4-Stream Master only demo and choose `Slave Only` as parameter choice for AXI4-Stream Slave only demo.



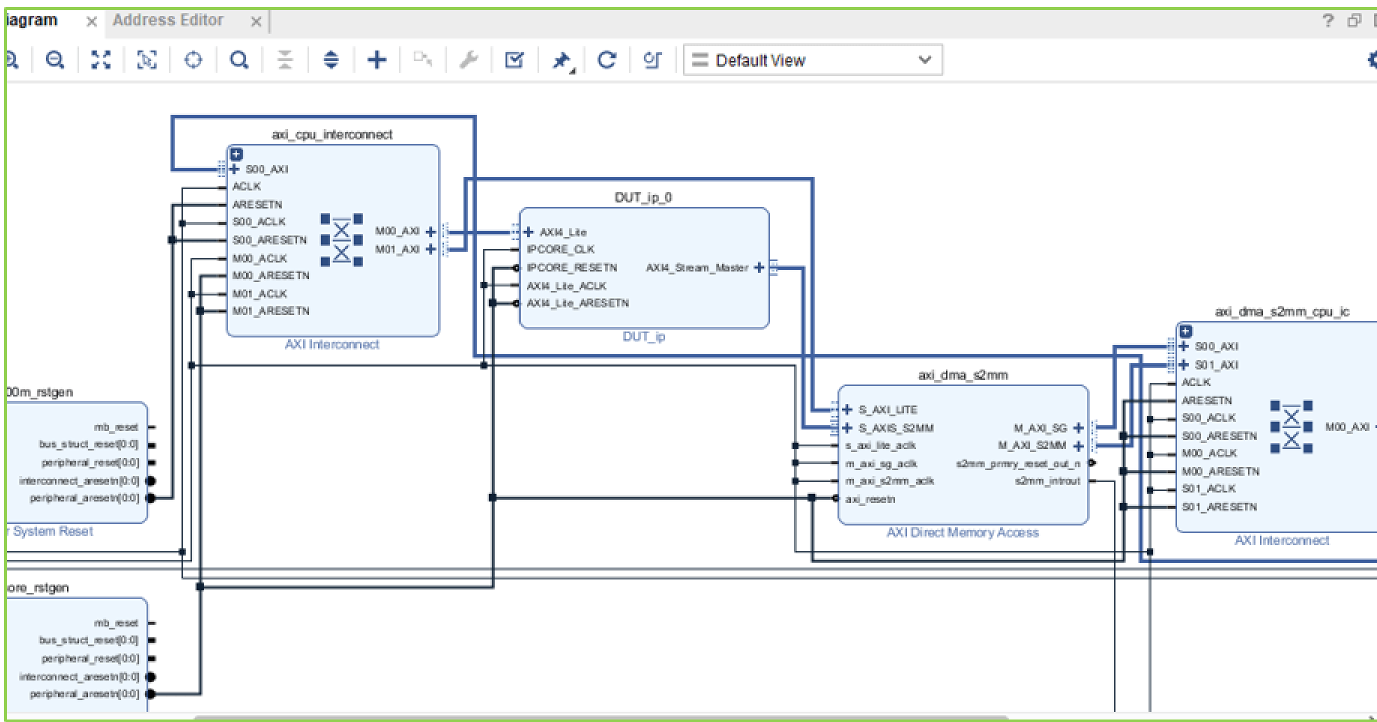
As shown in the figure above you can customize the reference design to Only AXI4-Stream Master or Only Slave or both AXI4-Stream Master and Slave by selecting the **Stream Channel** reference design parameter choice. Callback function with corresponding Tcl gets evaluated at the end of the **Set Target Reference Design** task.

5. If the parameter choice selected as `Master Only`, then the interface choice in task 1.3 shows as `AXI4 Stream Master`. Here, the AXI4-Stream interface communicates in master mode and sends data to AXI4_Stream Slave IP through **yout_data** signal. Similarly, If `Slave Only` parameter choice is selected, then the interface choice shows as `AXI4 Stream Slave`. Where, the AXI4-Stream interface communicates in slave mode and receives data through **axis_data_in** signal as shown below.

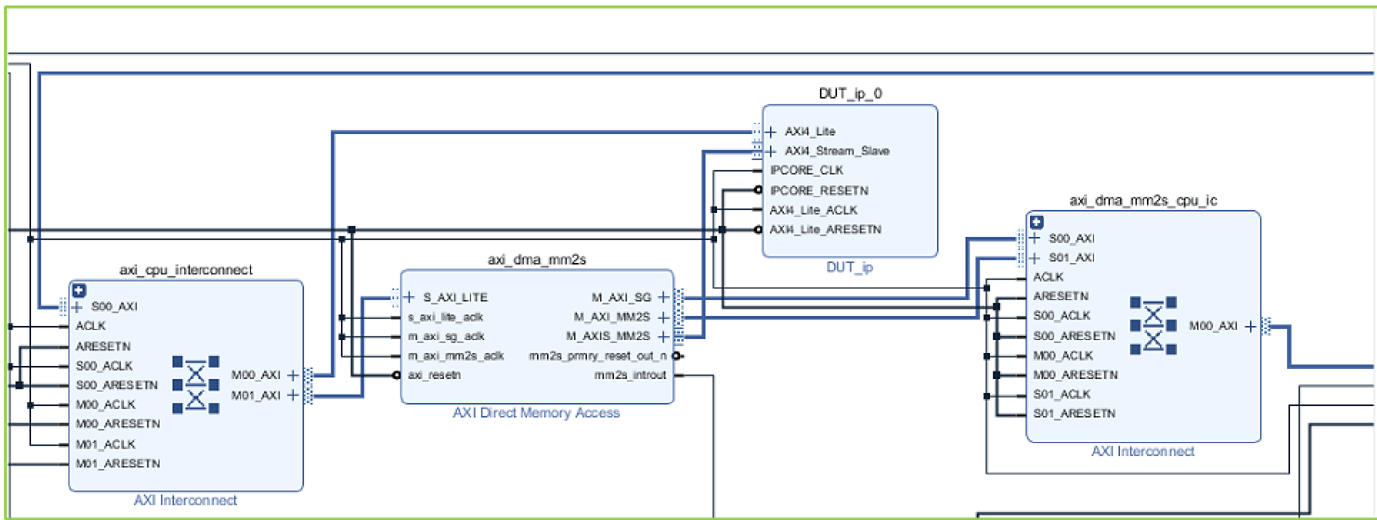


6. Right-click task 4.1, **Create Project**, and select **Run to Selected Task** to insert the generated IP core into the **AXI4-Stream interface with Stream channel Selection** reference design. The reference design contains Xilinx AXI DMA IP to handle the data streaming between FPGA fabric and processor or vice versa based on the reference design interface either Only AXI4-Stream master or only AXI4-Stream Slave.

Following diagram shows the generated vivado project with **AXI4-Stream Master only** interface choice, and you can see the connection between HDL coder generated DUT IP and slave to memory mapped Xilinx AXI DMA IP. In this reference design, DMA Controller reads the data from FPGA IP.



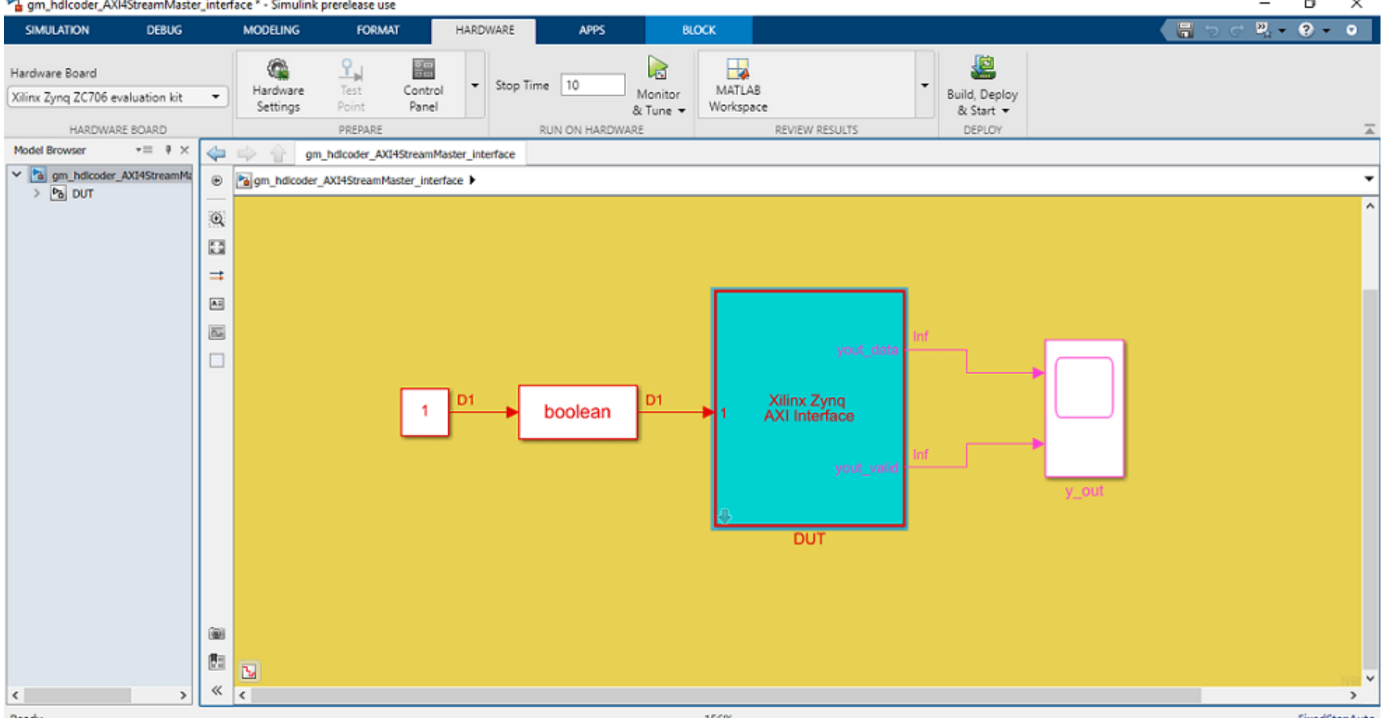
Similarly, following diagram shows the generated vivado project with **AXI4-Stream Slave only** interface choice, where the FPGA IP receives streaming data from DMA Controller.



7. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model, and build and download the FPGA bitstream.

Generate ARM executable Using AXI4-Stream Driver Block for AXI4-Stream Master only reference design

A software interface model is generated in Task 4.2, **Generate Software Interface Model**, as shown in the following picture.

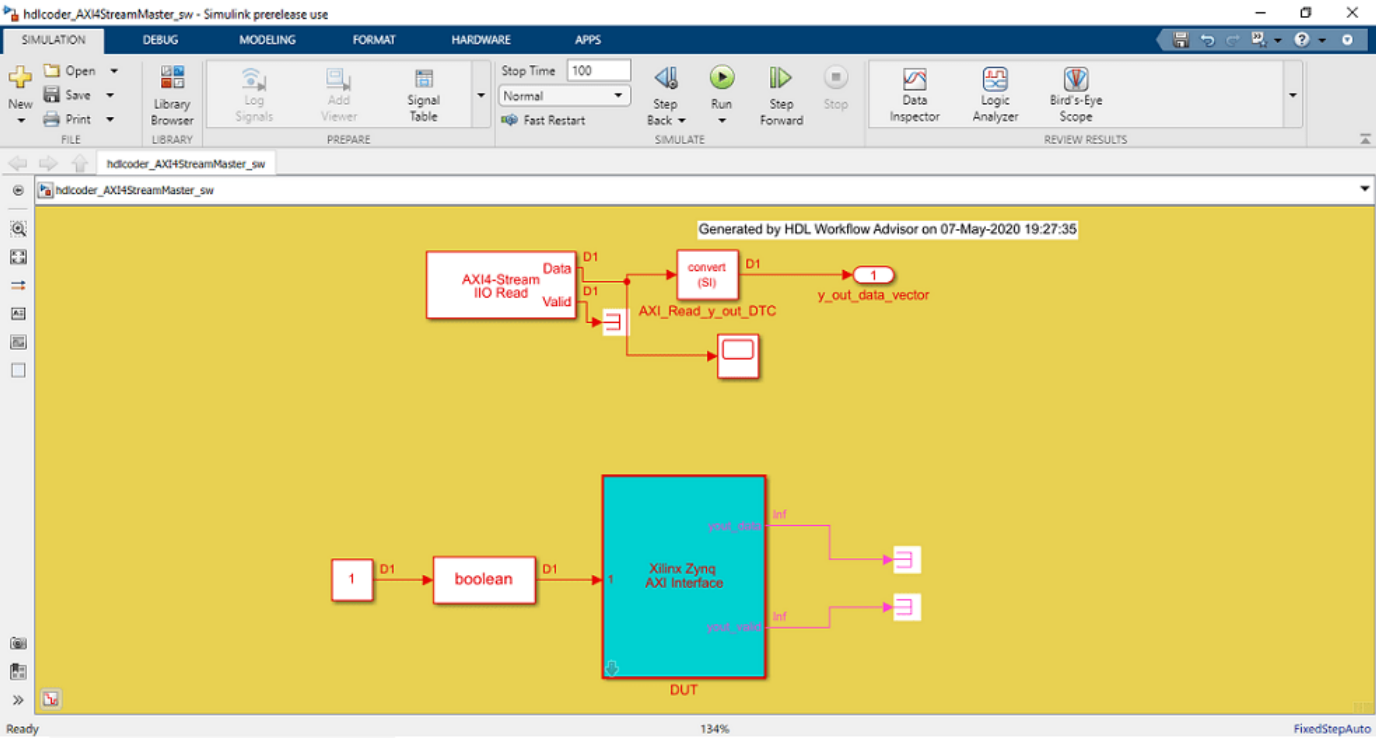


The AXI4-Stream IIO driver block cannot be automatically generated in the software interface model when a scalar port **yout_data** is mapped to AXI4-Stream interface "AXI4-Stream Master". Before you generate code from the software interface model, add the AXI4-Stream IIO Read driver block from the **Embedded Coder Support Package for Xilinx Zynq Platform** Library in the Simulink Library Browser.

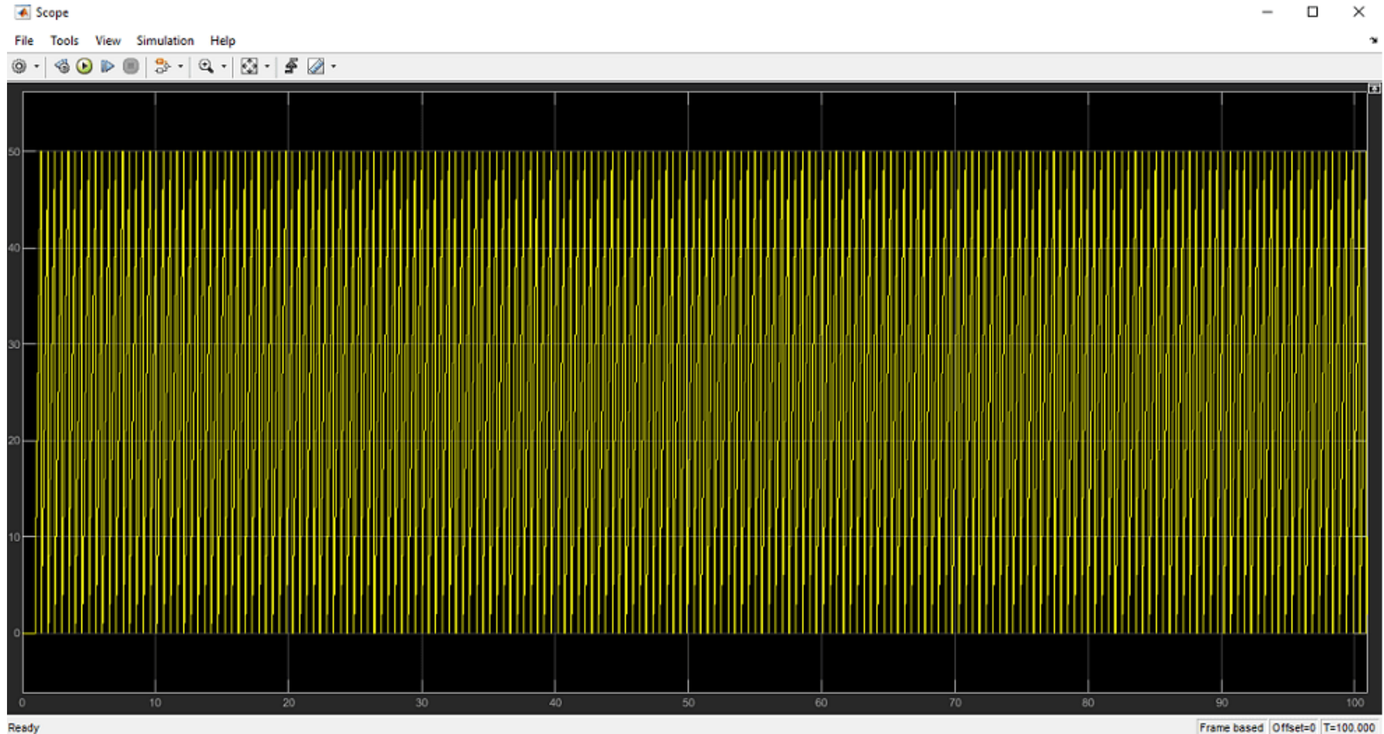
The updated software interface model for AXI4-Stream Master only reference designs are provided:

```
open_system('hdlcoder_AXI4StreamMaster_sw.slx')
```

Continuous data of count 0 to 50 is used in this model, and is connected to AXI4 Stream DMA driver block.



Click the **Monitor & Tune button** on the **Hardware** tab of Simulink Toolstrip. Embedded Coder builds the model, downloads the ARM executable to the Zc706 hardware. Now, both the hardware and software parts of the design are running on Zynq hardware. The FPGA IP sends the source data through the DMA controller and the AXI4-Stream interface. The ARM processor receives the data from the FPGA IP, and sends the result data to Simulink via external mode. Observe the output from the Zynq hardware on the time Scope y_out.

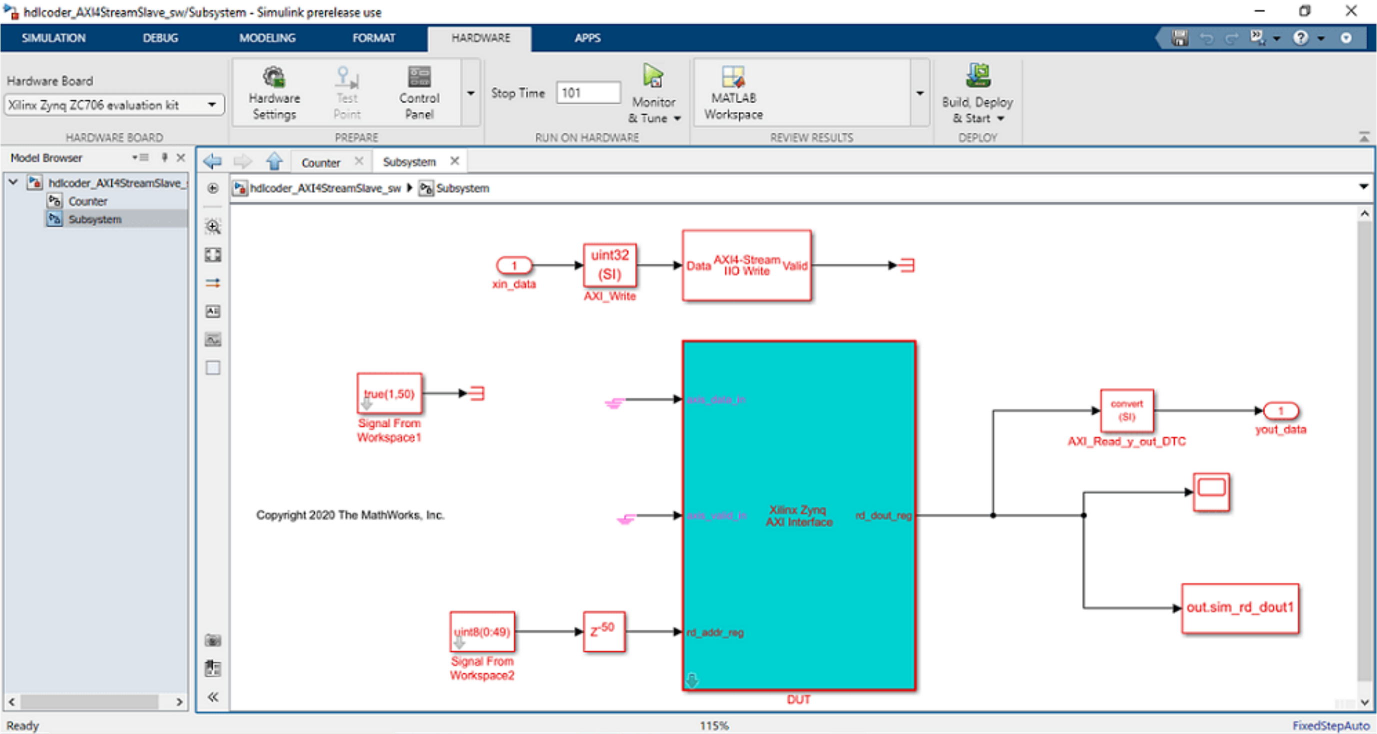


Generate ARM executable Using AXI4-Stream Driver Block for AXI4-Stream Slave only reference design

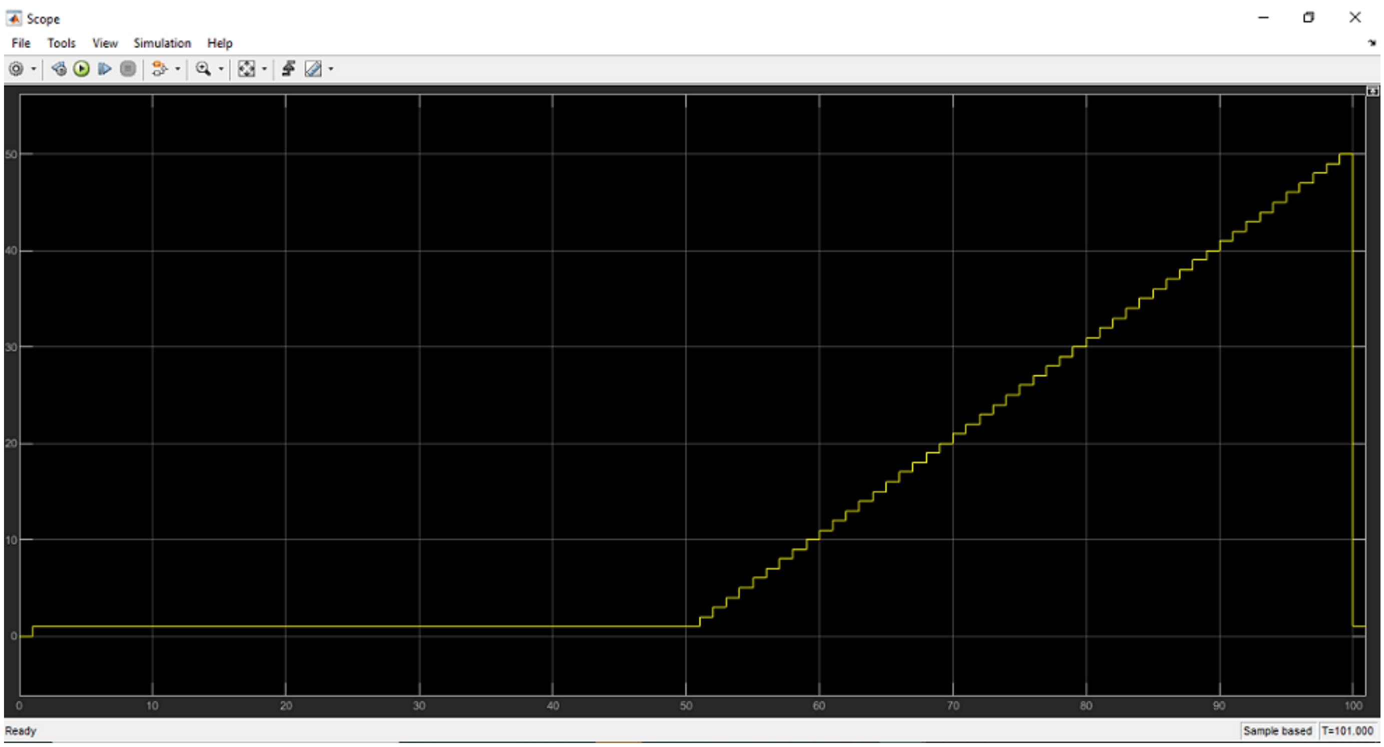
The updated software interface model for only AXI4-Stream Slave reference design is provided:

```
open_system('hdlcoder_AXI4StreamSlave_sw.slx')
```

In this model, a counter block which generates 1 to 50 incremental data is connected to AXI4-Stream Write DMA driver block. This means the DMA controller will stream count data samples to the HDL IP core via the AXI4-Stream Slave interface.



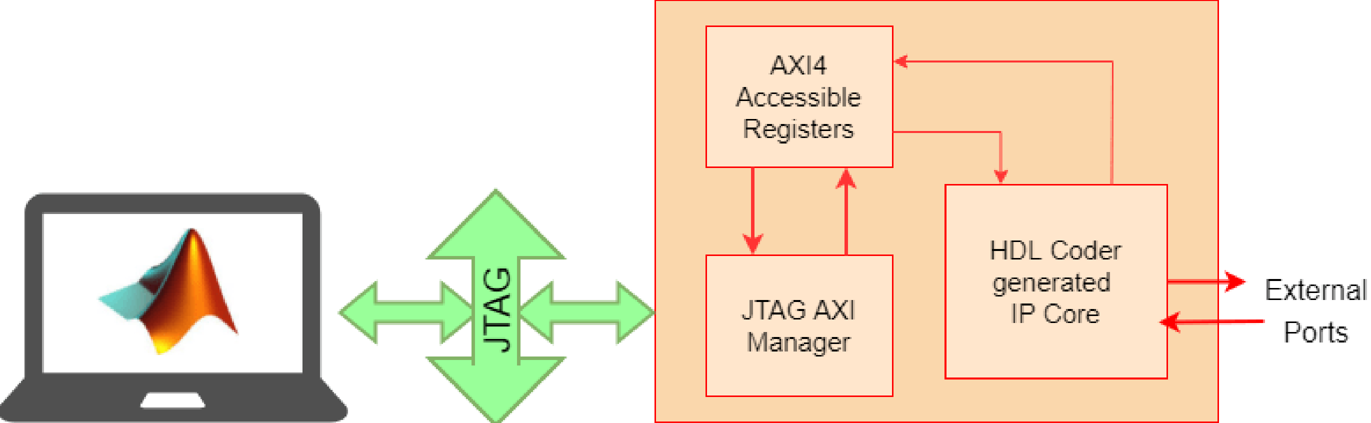
Click the **Monitor & Tune button** on the Hardware tab of model toolstrip. Embedded Coder builds the model, downloads the ARM executable to the Zc706 hardware. Now, both the hardware and software parts of the design are running on Zynq hardware. The ARM processor sends the source data to the FPGA IP, through the DMA controller and the AXI4-Stream interface. Observe the output of the IP core from the Zynq hardware on the Time Scope y_out.


Scope
File Tools View Simulation Help
0
10
20
30
40
50
60
70
80
90
100
Ready
Sample based
T=101.000

Use JTAG AXI Manager to Control HDL Coder Generated IP Core

This example shows how to specify automatic insertion of the HDL Verifier™ AXI Manager IP into a reference design. This example also explains how to use MATLAB® or Simulink® to configure the HDL Coder™ generated FPGA IP core.

To access onboard memory locations and quickly probe or control the FPGA logic from MATLAB or Simulink, use the JTAG AXI Manager IP. The `aximanager` System object™ you create in MATLAB connects to the AXI Manager IP over a physical JTAG cable and allows you to run read and write commands to subordinate memory locations from the MATLAB command line. In Simulink, the AXI Manager Read (HDL Verifier) and AXI Manager Write (HDL Verifier) blocks drive AXI Manager IP to perform read and write operations on output and input ports of the HDL Coder generated device under test (DUT) over the JTAG interface.



This example uses ZedBoard™ to implement the FPGA design.

Requirements

- HDL Verifier™ Support Packages for Xilinx FPGA Boards
- HDL Coder™ Support Package for Xilinx FPGA and SoC Devices
- Xilinx® Vivado™ Design Suite, with the supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- ZedBoard

Generate Reference Design with AXI Manager IP

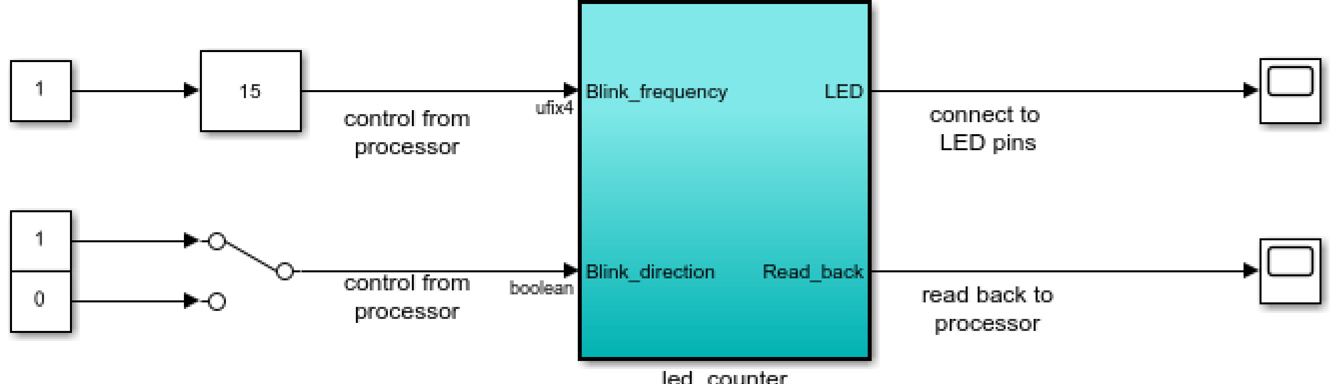
Set up the Xilinx Vivado tool path. Use your own Xilinx Vivado installation path when executing the command.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath', ...
                 'C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

Enable Insertion of JTAG AXI Manager

1. Open the `hdlcoder_led_blinking` model by running this command at the MATLAB command prompt.

```
open_system('hdlcoder_led_blinking')
```



Copyright 2014-2023 The MathWorks, Inc.

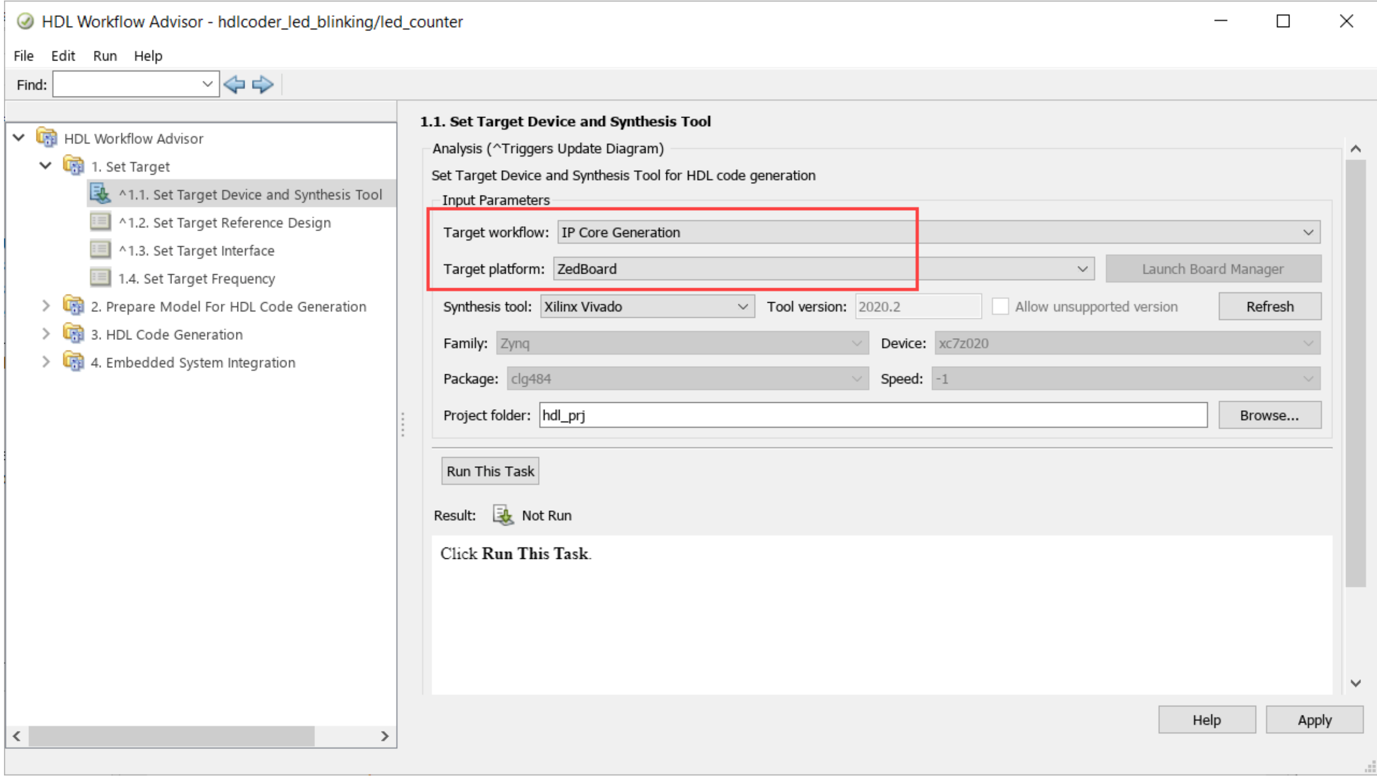
This example implements the `led_counter` subsystem on the hardware. This subsystem models a counter that causes LEDs to blink on the hardware. Two input ports, `Blink_frequency` and `Blink_direction`, are control ports that determine the LED blink frequency and direction, respectively. The output port `LED` connects to the LEDs on the hardware. You can use the output port `Read_back` to read data back to MATLAB.

2. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem and choosing **HDL Code > HDL Workflow Advisor**.

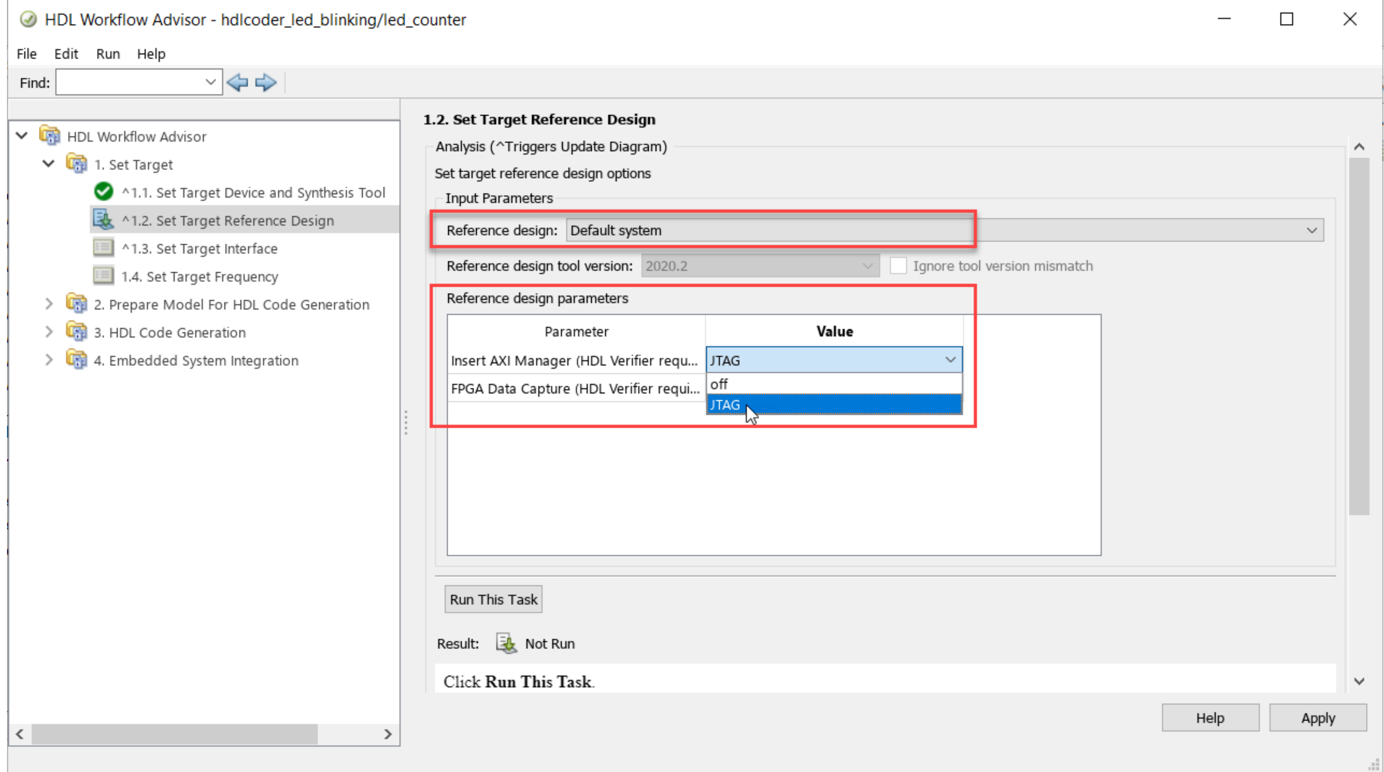
3. In the **Set Target > Set Target Device and Synthesis Tool** task, set **Target workflow** to `IP Core Generation`.

4. Set **Target platform** to `ZedBoard`. If you do not see this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select `Xilinx Zynq Platform` and follow the instructions provided by the Support Package Installer to complete the installation.

5. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.



6. In the **Set Target > Set Target Reference Design** task, choose **Default system** and set the **Insert AXI Manager (HDL Verifier required)** reference design parameter to JTAG.



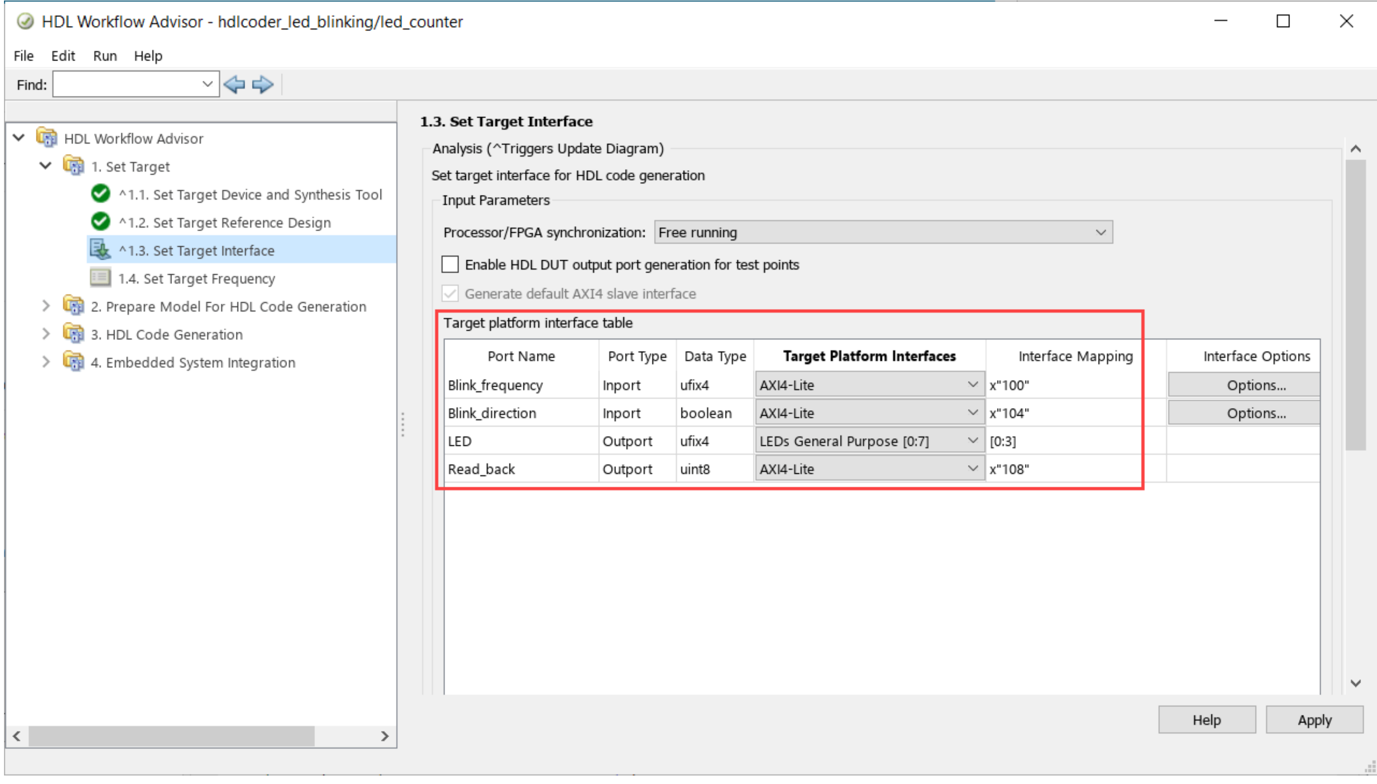
7. Click **Run This Task** to run the **Set Target Reference Design** task.

Generate HDL IP Core and Create Project with AXI Manager IP

Map each port in your DUT to one of the IP core target interfaces. This example maps the **Blink_frequency** and **Blink_direction** input ports to the `AXI4-Lite` interface, so the HDL Coder generates registers that can be accessed through the AXI interface for these ports. This example maps the **LED** output port to an external interface, `LEDs General Purpose [0:7]`, which connects to the LED hardware on the ZedBoard.

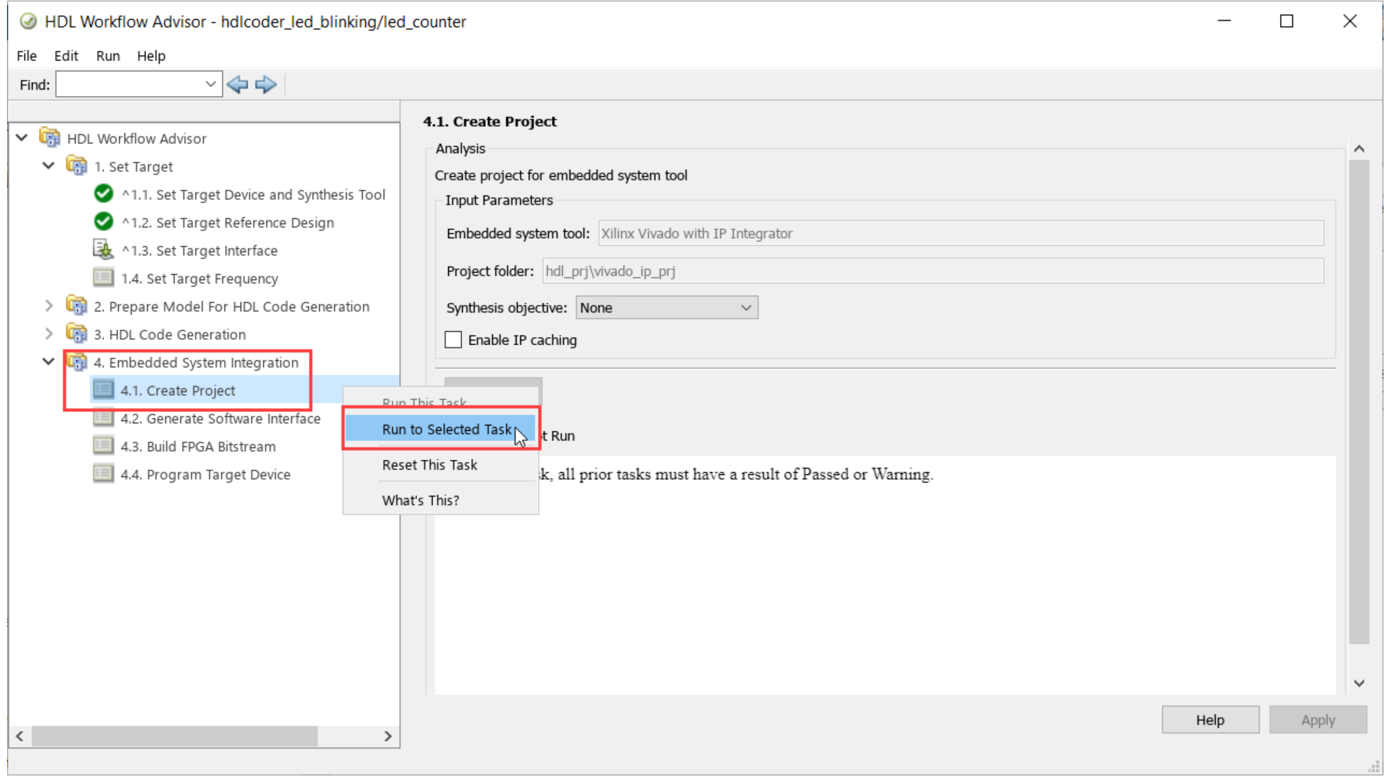
1. In the **Set Target** > **Set Target Interface** task, choose `AXI4-Lite` for **Blink_frequency**, **Blink_direction**, and **Read_back**.

2. Choose `LEDs General Purpose [0:7]` for **LED**.

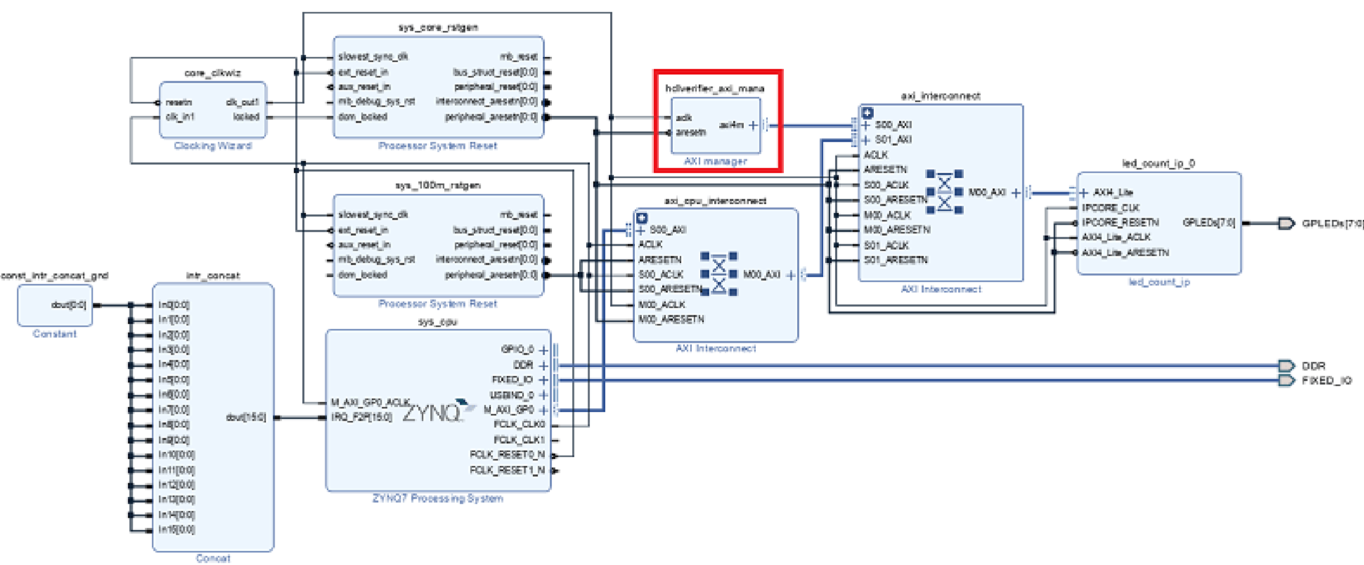


3. Create the reference design project, which includes JTAG AXI Manager.

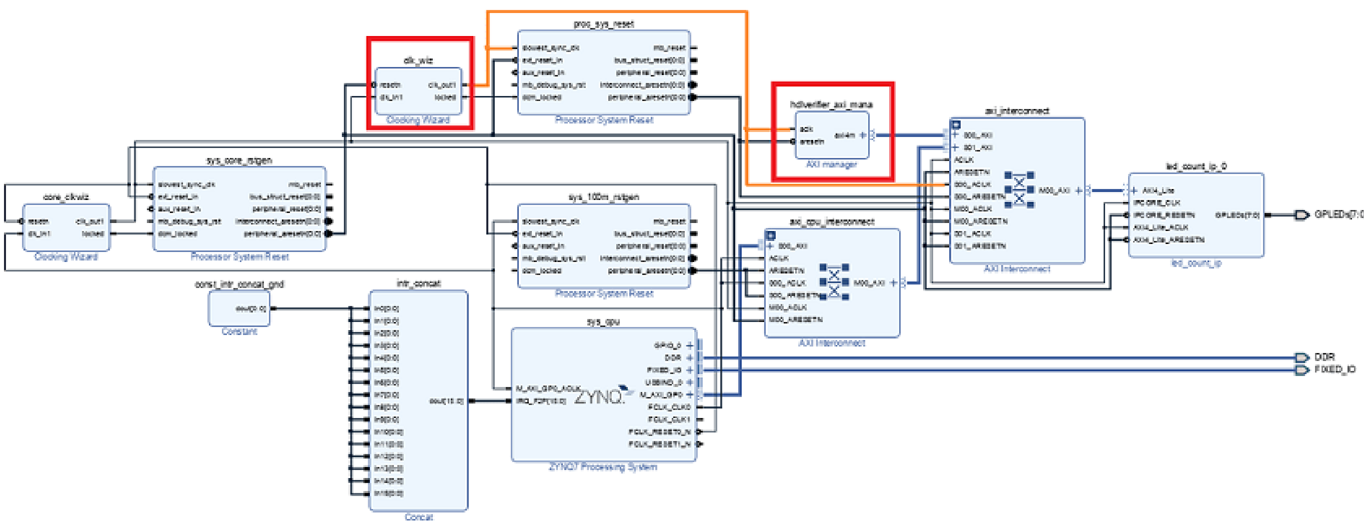
To create the project, right-click the **Create Project** task and select **Run to Selected Task**.



In the Vivado project, you can see the JTAG AXI Manager IP inserted in the reference design. If the target frequency in the **Set Target Frequency** task is less than or equal to 100 MHz, the software uses this reference design and the JTAG AXI Manager IP is driven by the DUT clock.



If the target frequency in the **Set Target Frequency** task is greater than 100 MHz, the software uses this reference design and the JTAG AXI Manager IP is driven by the 50 MHz fixed clock generated by clk_wiz_IO to avoid any timing failures.



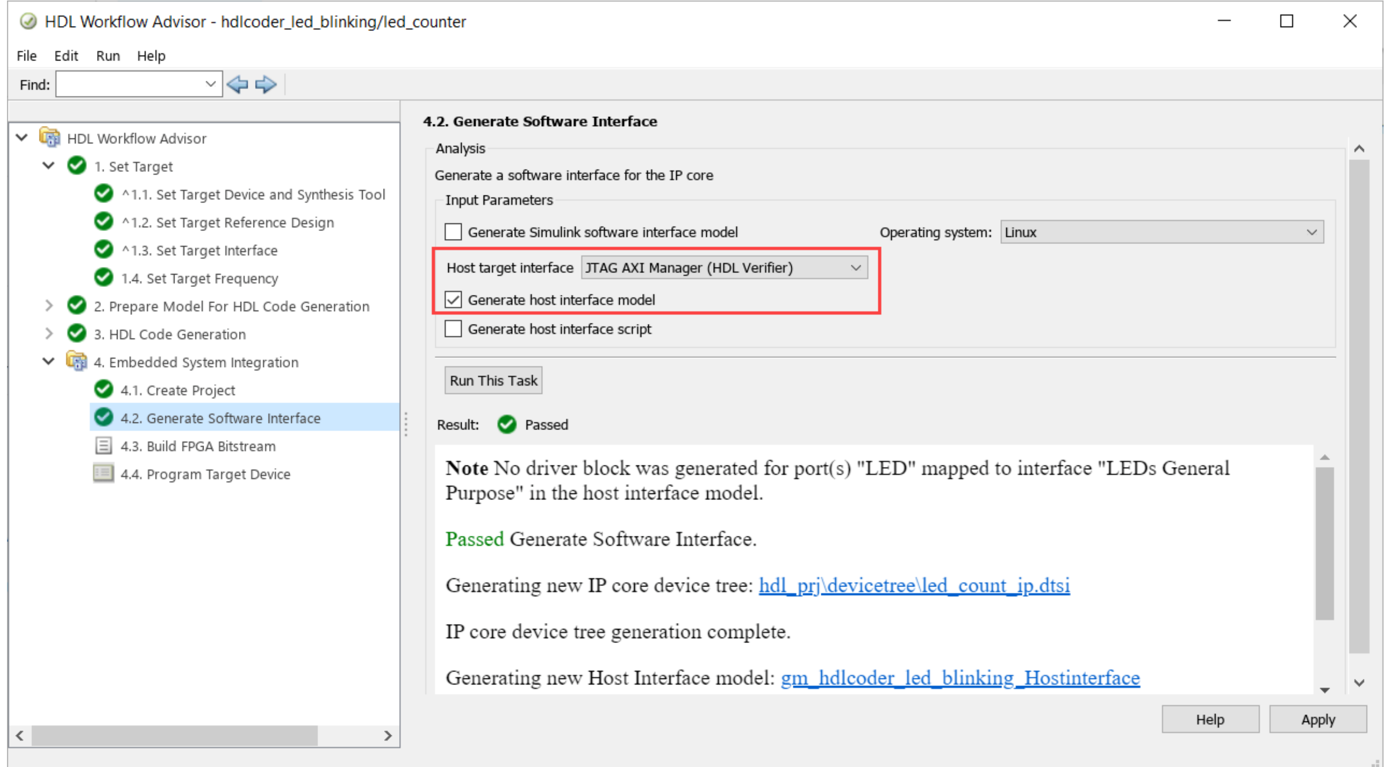
Generate Host Interface Model

Generate the host interface model by replacing the DUT with the AXI Manager Read (HDL Verifier) and AXI Manager Write (HDL Verifier) blocks in the Simulink model. Follow these steps to generate the host interface model.

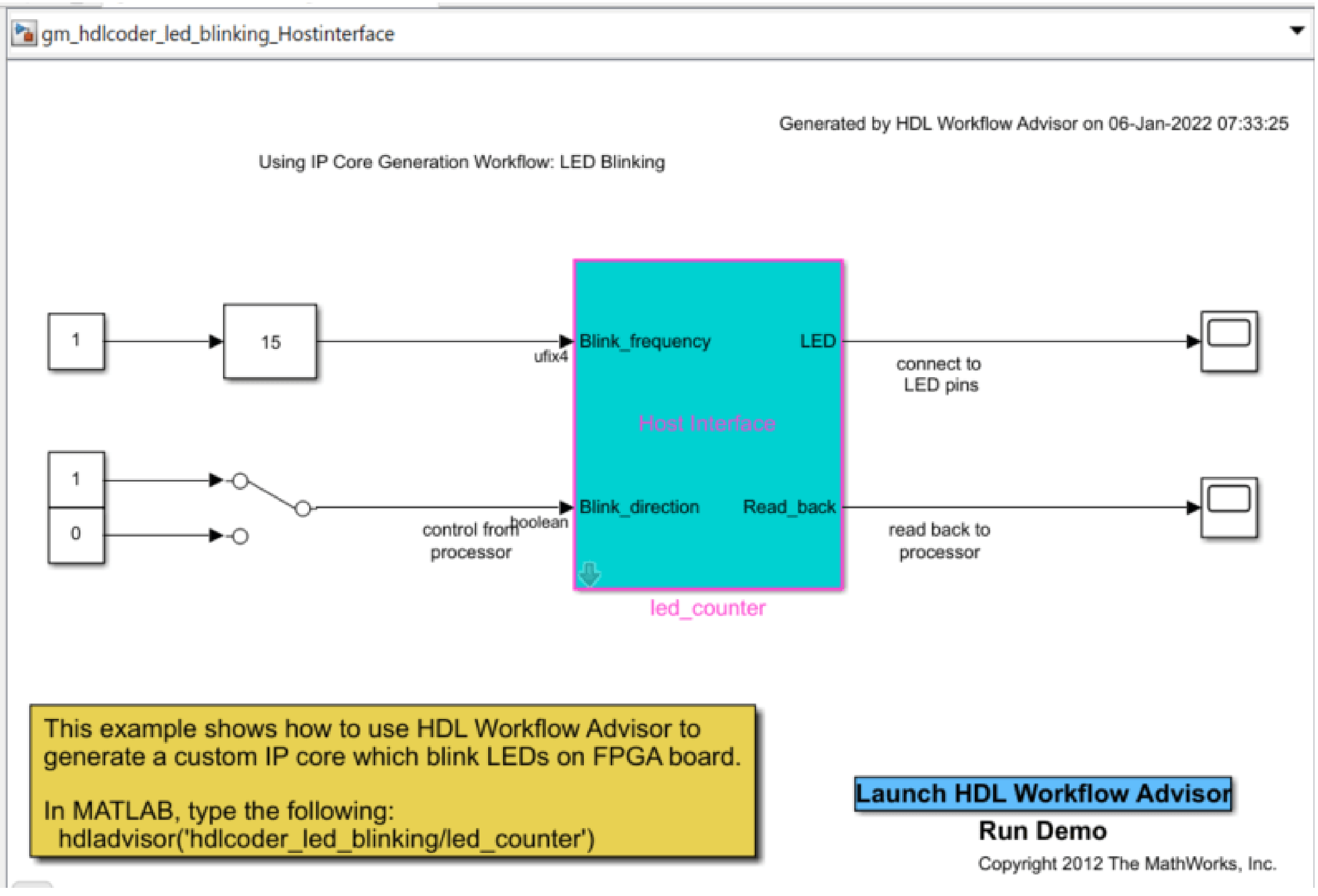
1. In the **Embedded System Integration** > **Generate Software Interface** task, set **Host target interface** to `JTAG AXI Manager (HDL Verifier)`.

2. Select **Generate host interface model**.

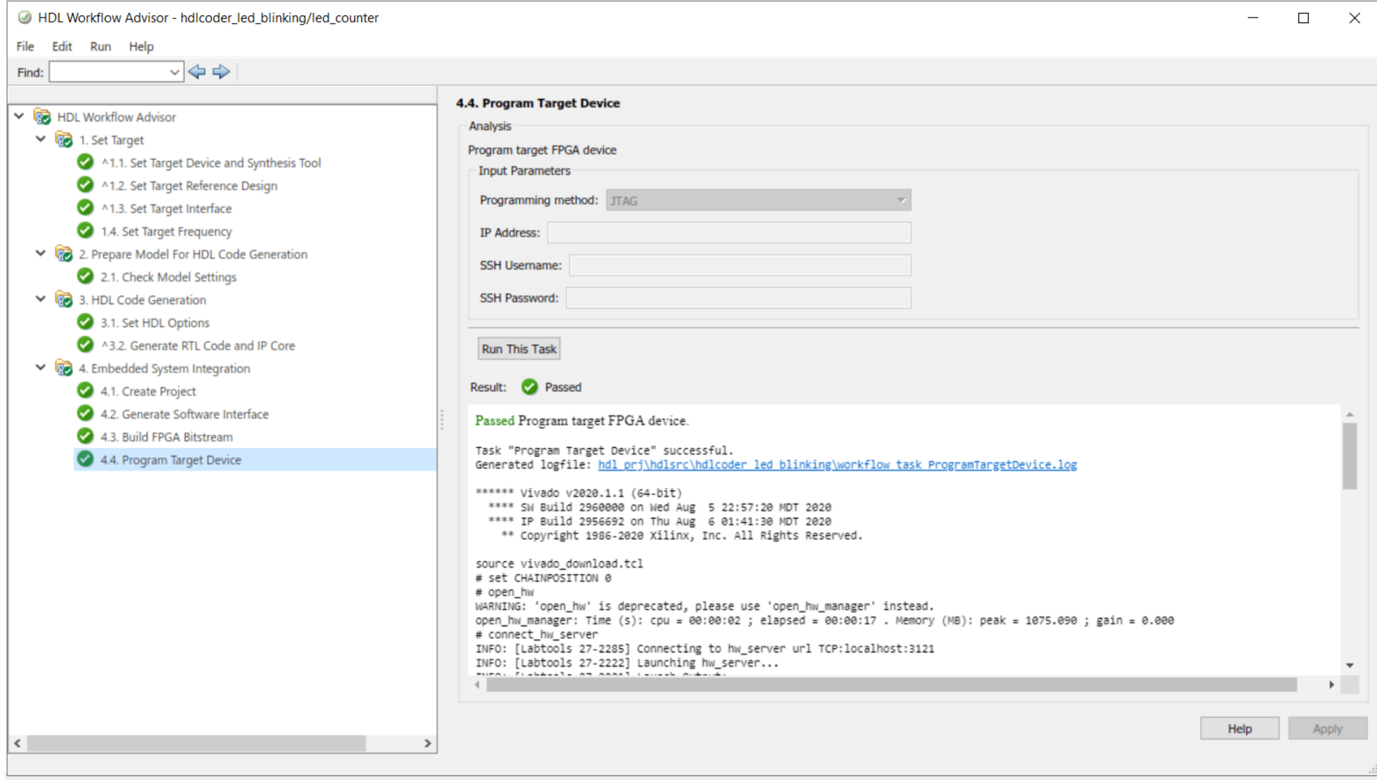
3. Click **Run This Task**.



The following screen shot shows the `gm_hdlcoder_led_blinking_Hostinterface.slx` host interface model. You can use the host interface model as a Simulink test bench to tune DUT registers.



Run the remaining steps in the workflow to generate a bitstream and program the target device.



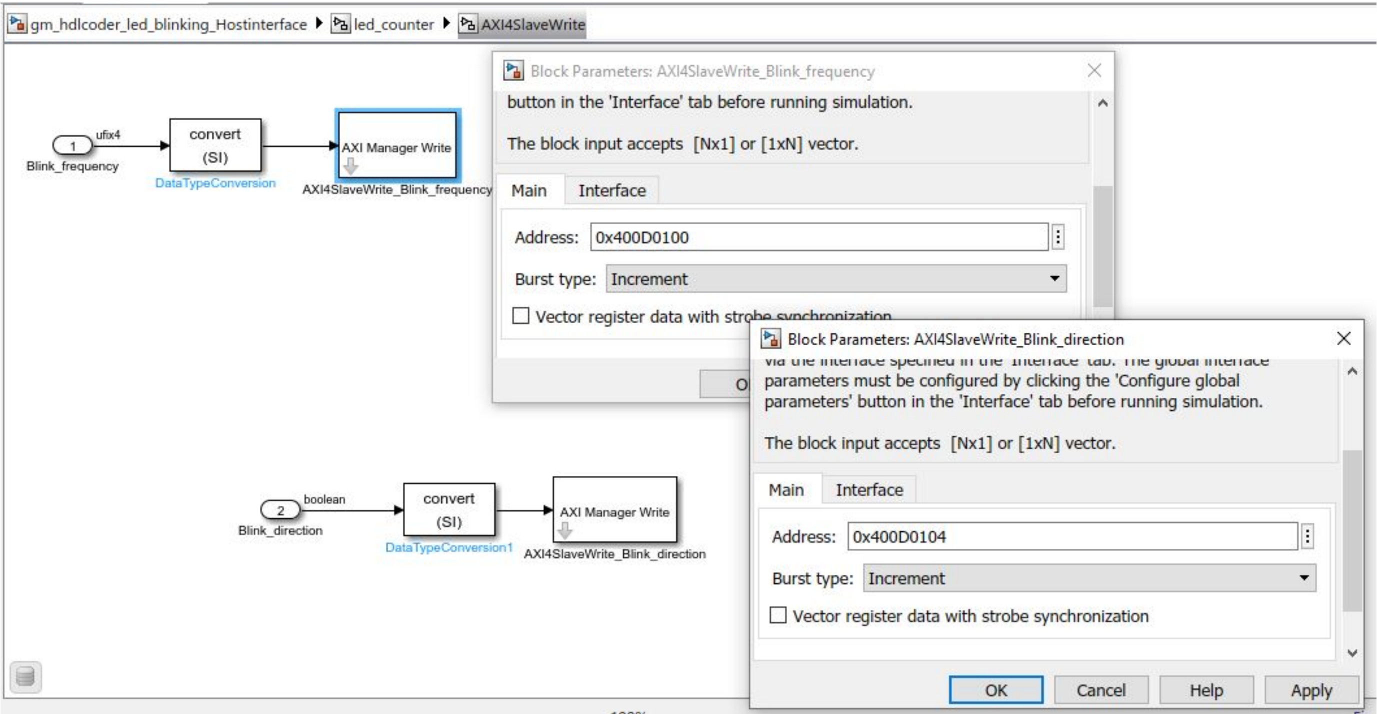
Control HDL Coder IP Core Using JTAG AXI Manager

Simulink Model to Control HDL Coder IP Core

1. Open the `gm_hdlcoder_led_blinking_Hostinterface` model.

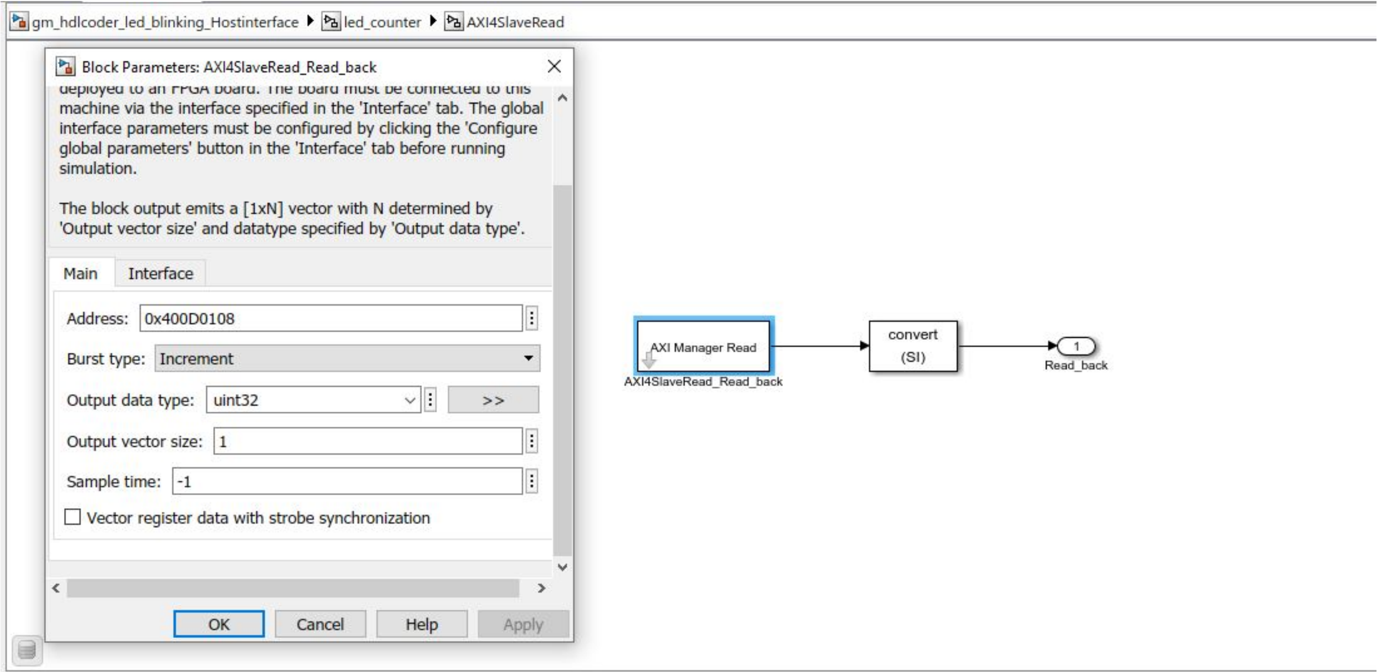
2. Go to `gm_hdlcoder_led_blinking_Hostinterface` > `led_counter` > `AXI4SlaveWrite` subsystem. Configure the AXI Manager Write (HDL Verifier) block.

This figure shows the default configuration of the AXI Manager Write block for the current HDL Workflow Advisor configuration.



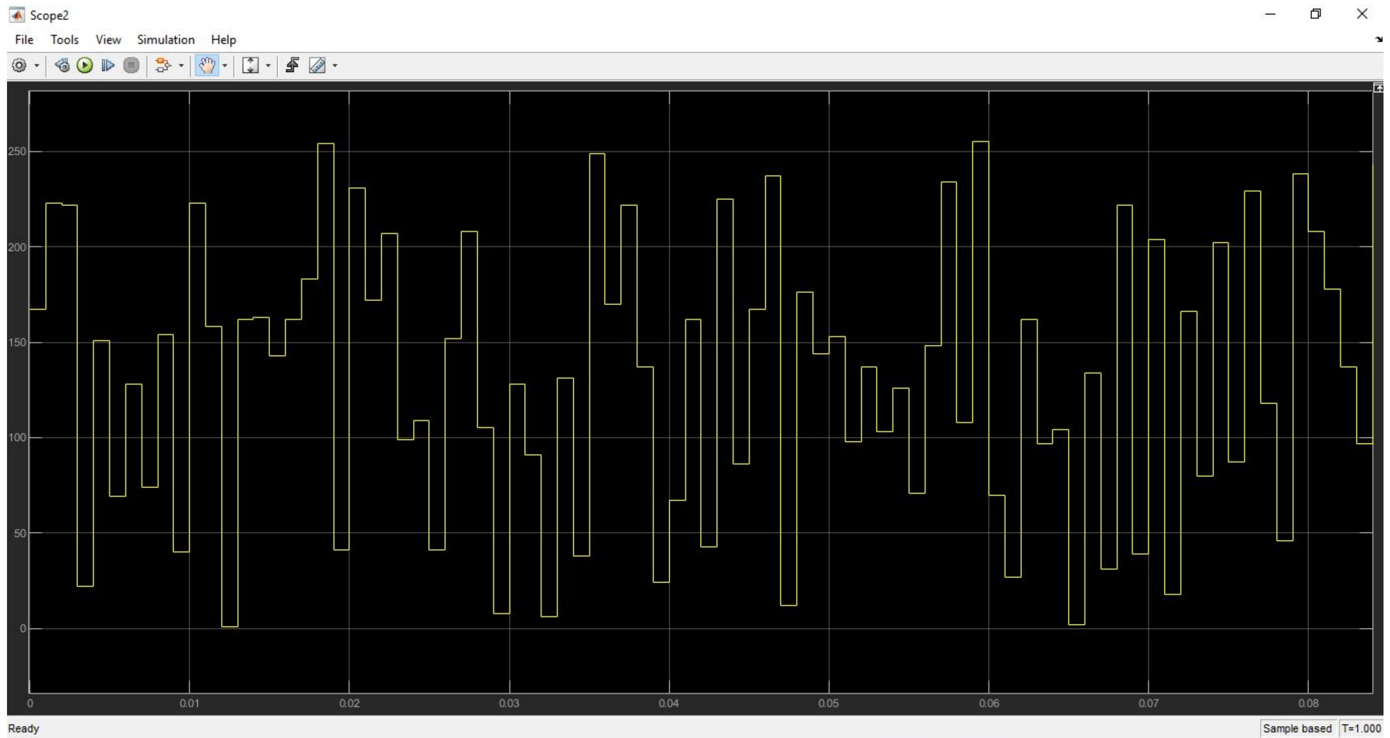
3. Go to the `gm_hdlcoder_led_blinking_Hostinterface` > `led_counter` > `AXI4SlaveRead` subsystem. Configure the AXI Manager Read (HDL Verifier) block.

This figure shows default configuration of the AXI Manager Read block for the current HDL Workflow Advisor configuration.



Run the `gm_hdlcoder_led_blinking_Hostinterface` model to perform read and write operations from Simulink.

The figure shows **Read_back** register (LED output) samples over time.



MATLAB Command Line Interface

You can also control the DUT from the MATLAB command line by creating an object for AXI Manager IP.

1. Create the AXI manager object.

```
h = aximanager('Xilinx')
```

2. Change the LED blinking frequency.

```
h.writememory('400D0100',0)
```

Observe that the LED blinking frequency is low. Change the frequency from 0 to 15 to increase the LED blinking frequency.

```
h.writememory('400D0100',15)
```

3. Read the current counter value.

```
h.readmemory('400D0108',1)
```

4. Delete the object when done to free up the JTAG resource. If you do not delete the object, any other JTAG operation fails.

```
delete(h)
```

JTAG AXI Manager in Custom Reference Designs

The **Insert AXI Manager (HDL Verifier required)** reference design parameter is by default added to your custom reference design. The default value for the parameter is `off`. In custom reference design, you can control this parameter by using the following properties.

- “AddMATLABAXIManagerParameter” - Control visibility of the Insert AXI Manager parameter.
- “MATLABAXIManagerDefaultValue” - Specify whether to insert the AXI manager IP.

You can set the **Insert AXI Manager (HDL Verifier required)** parameter to display in **Set Target Reference Design** task and assign the default value as `JTAG` by adding the properties in the **plugin_rd.m** file of the custom reference design workflow shown in this figure.

plugin_rd.m

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.ReferenceDesignName = 'HDL Verifier AXIM and Data Capture Design';
hRD.BoardName = 'ZedBoard';

% Tool information
hRD.SupportedToolVersion = {'2019.2', '2020.1', '2020.2'};

% Enable AXI Manager
hRD.AddMATLABAXIManagerParameter = true;
% set default value as 'JTAG'
hRD.MATLABAXIManagerDefaultValue = 'JTAG';
```

Debug a Zynq Design Using HDL Coder and Embedded Coder

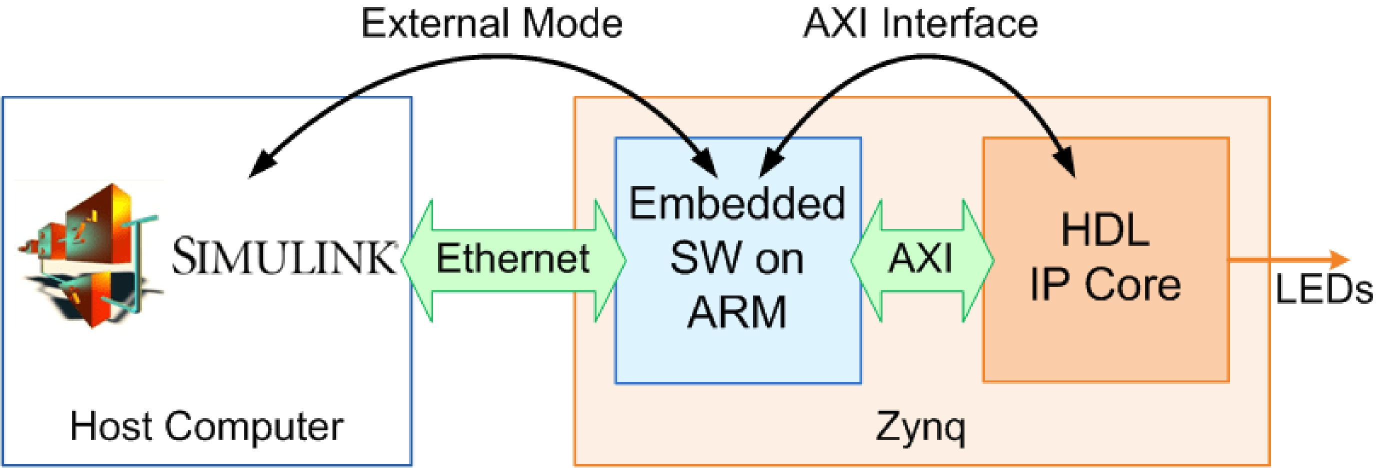
This example shows how to debug a Zynq® design using HDL Coder™ and Embedded Coder® features.

Requirements

- Xilinx® Zynq-7000 SoC ZC702 Evaluation Kit
- HDL Coder Support Package for Xilinx FPGA and SoC Devices
- Follow the Set up Zynq hardware and tools section in “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98 to setup ZC702 hardware.

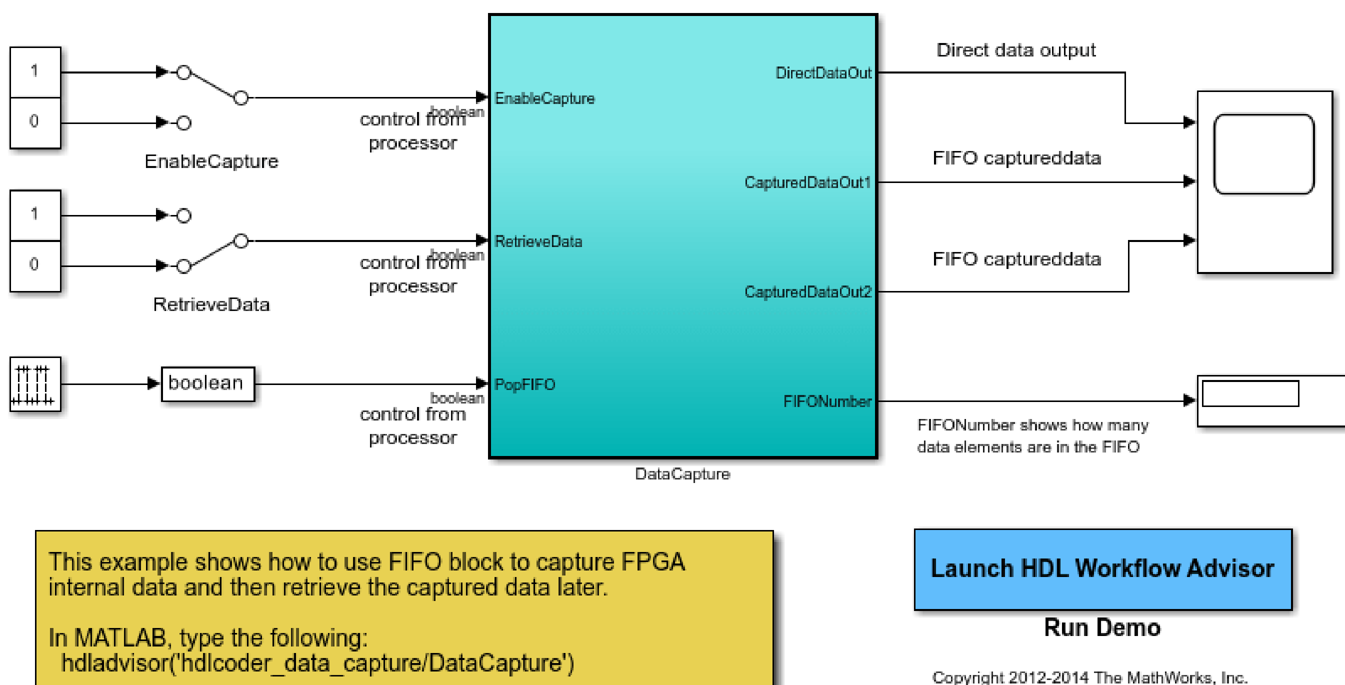
Introduction

When you are prototyping and developing an algorithm for Zynq platform, it is useful to monitor, tune, and debug the algorithm while it runs on hardware. This example shows how to use features like external mode, AXI interface and HDL FIFO blocks to probe into the Zynq design. Using the external mode feature, you can probe the internal data in the software running on the ARM® processor. And because the ARM processor is connected to the FPGA through AXI interface, you can monitor and tune the parameters on FPGA as well. Together with HDL FIFO blocks, you can capture fast FPGA data and retrieve it back to Simulink® for analysis.



Start by opening the example model.

```
open_system('hdlcoder_data_capture');
```



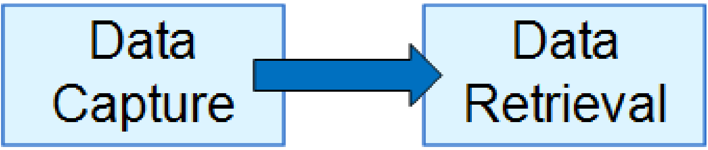
The subsystem **DataCapture** is the hardware subsystem targeting the FPGA fabric. Inside this subsystem, the **OriginalDUT** subsystem contains a **Trigonometric Function** block, which generates fast sine and cosine data streams. The **OriginalDUT** subsystem represents our algorithm design. If we want to debug this design, how do we capture and monitor this fast data stream?

The FPGA runs at a much faster clock frequency than the software code on the ARM processor. External mode can be used with the software running on the ARM processor to monitor slow-changing status parameters, such as FIFO status, but the sample rate of the software code, for example, 1KHz, is not fast enough to capture the fast-changing data in the FPGA, for example, 50MHz.

This example shows how to use a FIFO block to capture the fast FPGA data, and then use the software on the ARM processor to retrieve the captured data through the AXI interface and external mode.

For debugging purposes, we add the subsystem **Debug_FIFOs** to the DUT. This subsystem uses two HDL FIFO blocks to capture the fast data streams for future retrieval. The control signal inputs to the **Debug_FIFOs** subsystem are connected to the DUT interface, and are connected to the ARM processor via the AXI interface.

At the top level of the example model, when the **EnableCapture** switch is turned on, and **RetrieveData** switch is turned off, the **Debug_FIFOs** module will capture 1000 data samples into the **HDL FIFO** blocks. This is the data capture phase. Then, when the **EnableCapture** switch is kept on, and the **RetrieveData** switch is turned back on, the **Debug_FIFOs** module will transfer the captured data back to the ARM processor. This is the data retrieval phase. You can use the manual switches to switch between these two phases to capture and monitor the internal FPGA data.



Thus, for every signal you want to monitor, you can insert more **Debug_FIFO** modules to your design to capture and retrieve the data back to Simulink. You can also use your own control signals, or extend this example with your own triggers, or qualifiers.

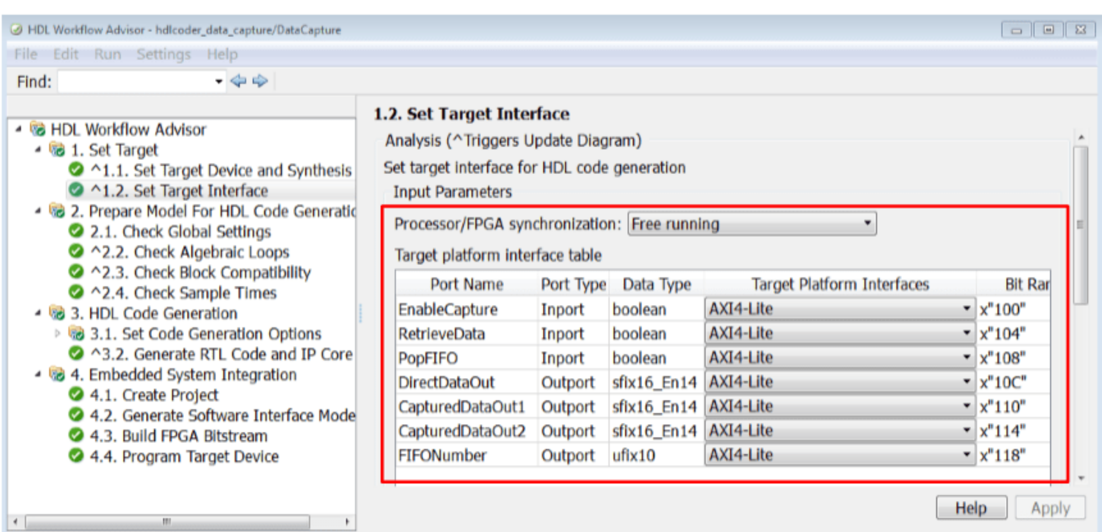
The output port of the hardware subsystem, **DirectDataOut**, outputs data directly to the AXI interface. In contrast, the output ports **CapturedDataOut1** and **CapturedDataOut2** outputs captured data from the FIFOs. We will compare the results of these two outputs in the last section.

Deploy the design on Zynq hardware

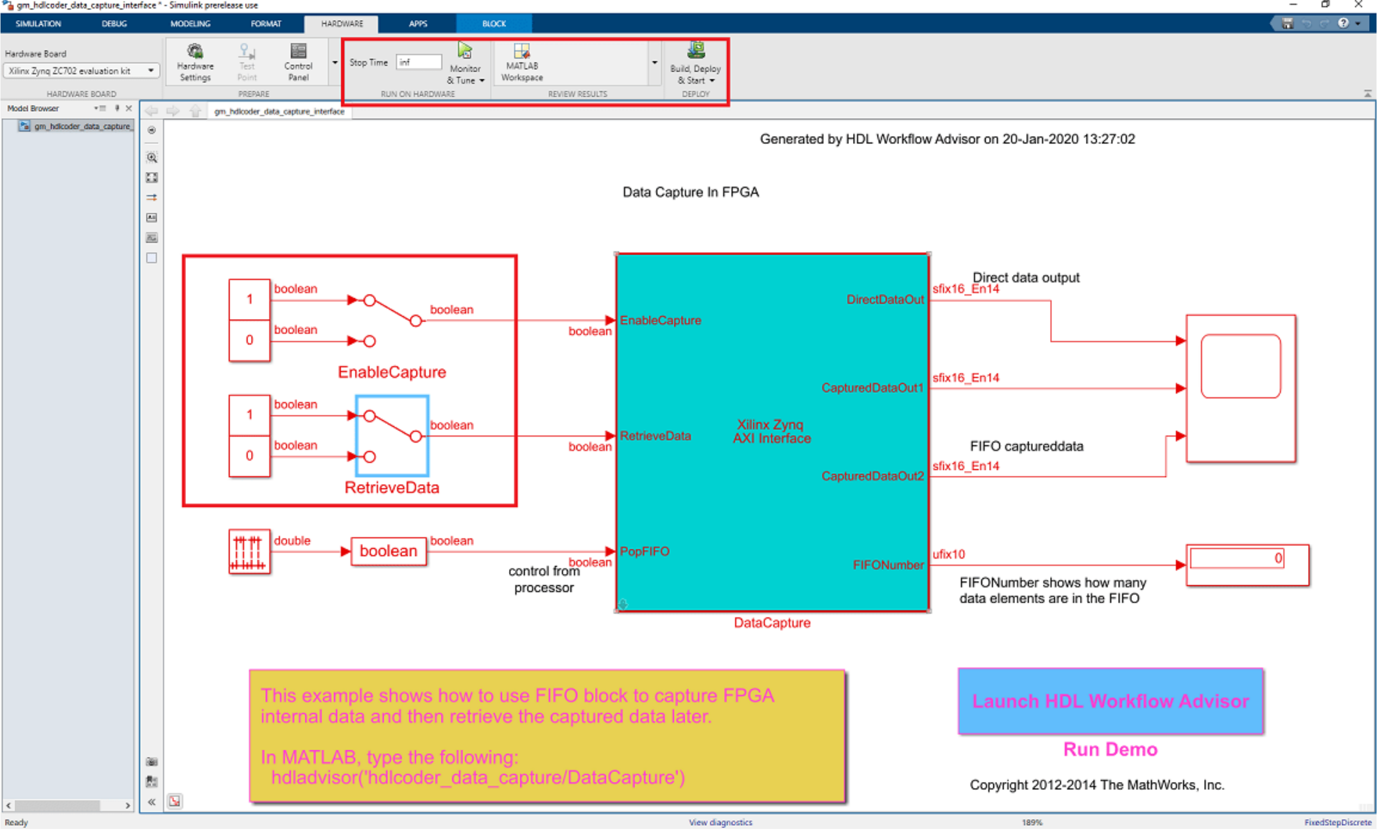
Next, we will start HDL Workflow Advisor from the model and run through the Zynq HW/SW co-design workflow to deploy this design on Zynq hardware. For a detailed step by step guide, refer to the example "Getting Started with Targeting Xilinx Zynq Platform" on page 39-98.

1. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**. For **Target platform**, select **Xilinx Zynq ZC702 evaluation kit**. Run this task.

2. In the **Set Target > Set Target Interface** task, choose **AXI4-Lite** for all the input and output ports.



3. Then run through all the workflow steps to generate HDL IP, create an EDK project, generate the software interface model, and build and download the FPGA bitstream. The generated software interface model is shown in following picture:



4. Configure and build the software interface model for external mode:

1. In the generated model, click on Hardware pane and go to **Hardware settings** to open **Configuration Parameter** dialog box.
2. Select **Solver** and set **Stop Time** to **inf**.
3. From the Hardware pane, click the **Monitor and Tune** button.
4. Click the **Run** button on the model toolstrip. Embedded Coder builds the model, downloads the ARM executable to the Zynq ZC702 hardware, executes it, and connects the model to the executable running on the Zynq ZC702 hardware.

Capture and display data from Zynq hardware

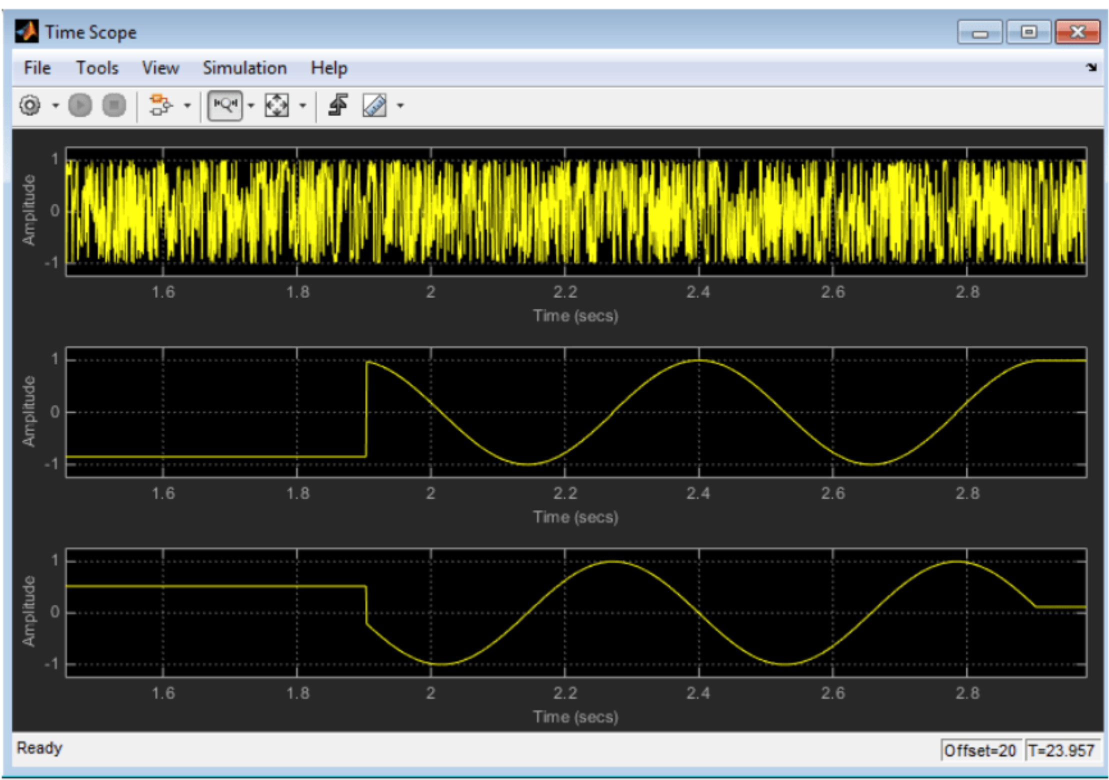
Now both the hardware and software parts of the design are running on Zynq hardware, the next step is to capture and retrieve data from the Zynq board.

Once the external mode is connected, make sure the **EnableCapture** switch is in the **1** position, and the **RetrieveData** switch is in the **0** position. Notice the **FIFONumber** display box increases to 1000 almost immediately. This means the FIFO inside the FPGA fabric started capturing data and was quickly filled with 1000 data samples.

Open the **Time Scope** block and observe the **DirectDataOut** output in the first row. Notice the received data is a seemingly random waveform between -1 and 1. This is because the FPGA is running at a much faster frequency than software is running on the ARM processor. Directly using external mode to monitor fast FPGA data means sampling a fast sine waveform at a very slow rate, which generates a random waveform between -1 and 1.

Now double-click the **RetrieveData** switch to enable data readout. The **EnableCapture** switch needs to be kept on. When the **RetrieveData** switch is turned on, the internal logic modeled in this example sends out the captured data samples one by one from FIFO to the ARM processor, through the AXI interface. These data samples are then sent from ARM processor to Simulink via external mode. Notice the **FIFONumber** display box decreases to 0.

Open the **Time Scope** block, the second and third row of the scope now shows the sine and cosine wave we captured in the FIFO. Following picture shows the scope waveforms.



Summary

This example shows how to use features like external mode, AXI interface and HDL FIFO blocks to probe into the Zynq design, capture fast FPGA data, and retrieve it back to Simulink for analysis. You can also do similar monitoring with FPGA vendor tools like ChipScope™ or SignalTap™. But the strength of this approach is that you can get all the visualization benefits of Simulink, like various scope blocks, and you can also build your own custom controls, triggers, or qualifiers in Simulink.

Debug IP Core Using FPGA Data Capture

This example shows how to debug an IP core you generate in HDL Coder™ using only FPGA Data Capture as well as both AXI Manager and FPGA Data Capture together.

Requirements

- Xilinx® Zynq® ZC702 evaluation kit. For information on setting up the ZC702 hardware, see "Getting Started with Targeting Xilinx Zynq Platform" on page 39-98.
- HDL Coder Support Package for Xilinx FPGA and SoC Devices
- HDL Verifier™ Support Package for Xilinx FPGA Boards
- Xilinx Vivado™ Design Suite, with a supported version listed in the "HDL Language Support and Supported Third-Party Tools and Hardware"
- (Optional) DSP System Toolbox

Introduction

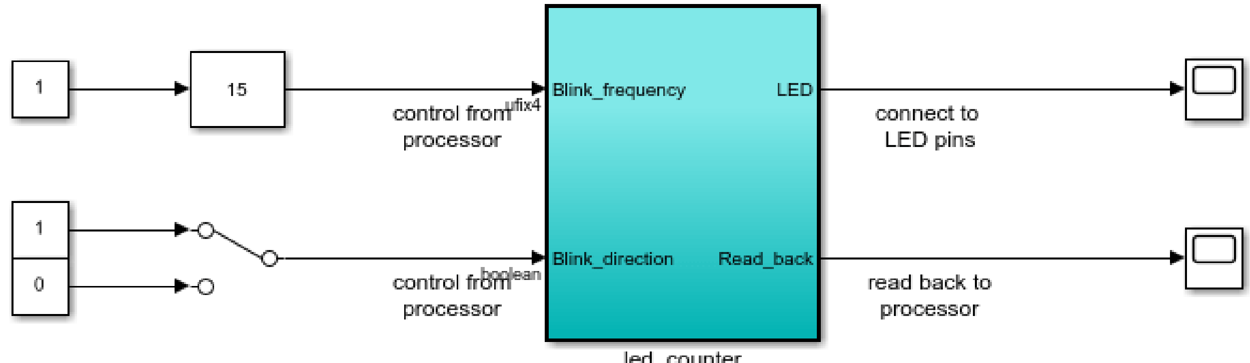
Monitoring IP core internal signals while a design is running on real hardware is useful because you can debug and analyze the design. This example illustrates how to use the FPGA Data Capture to capture internal signals of the generated IP core in MATLAB®.

This example shows how to use FPGA Data Capture in blocking and nonblocking modes. For more information on capture modes, see "CaptureMode" (HDL Verifier). Nonblocking mode allows simultaneous use of AXI Manager and FPGA Data Capture over a JTAG interface. Blocking mode suspends MATLAB execution, so you cannot use other applications when you use this mode. By contrast, nonblocking mode allows you to use AXI Manager to configure the IP core while FPGA Data Capture captures data from the hardware.

Open the model.

```
open_system('hdlcoder_led_blinking_data_capture');
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
hdladvisor('hdlcoder_led_blinking_data_capture/led_counter')

Launch HDL Workflow Advisor

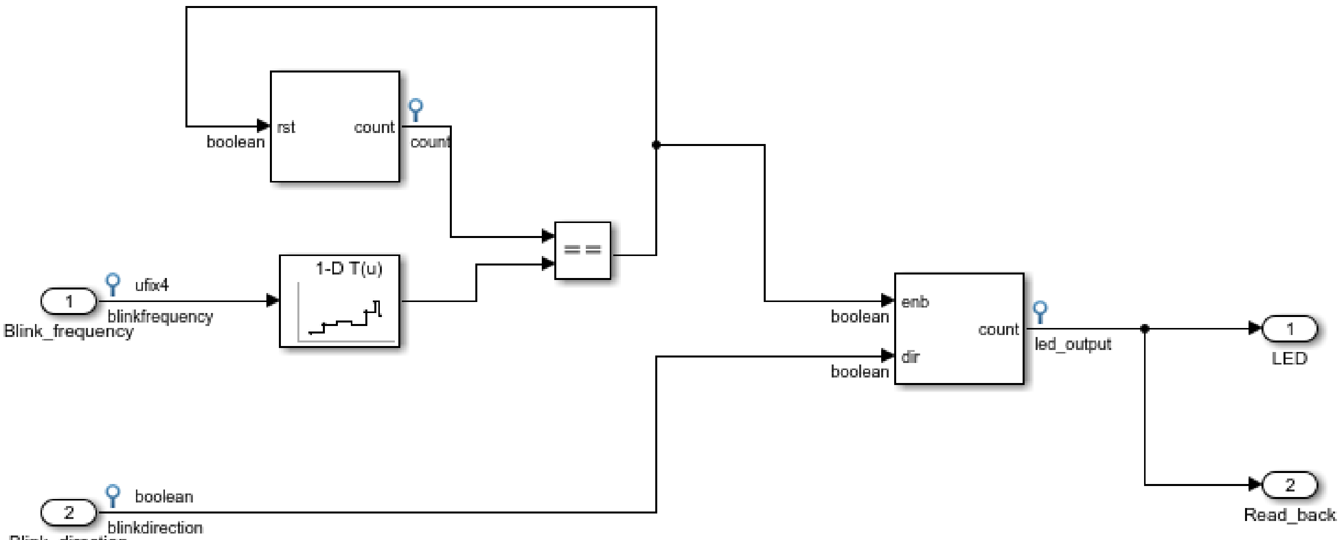
Run Demo

Copyright 2018 The MathWorks, Inc.

This example implements the `led_counter` subsystem on the hardware. This subsystem models a counter that causes LEDs to blink on the hardware. Two input ports, `Blink_frequency` and `Blink_direction`, are control ports that determine the LED blink frequency and direction, respectively. The output port `LED` connects to the LEDs on the hardware. You can use the output port `Read_back` to read data back to MATLAB.

In the `led_counter` subsystem, several internal signals are test points. HDL Coder routes those internal signals out of the DUT and into the IP core wrapper so that the signals can be connected to the FPGA Data Capture HDL IP.

```
open_system('hdlcoder_led_blinking_data_capture/led_counter');
```

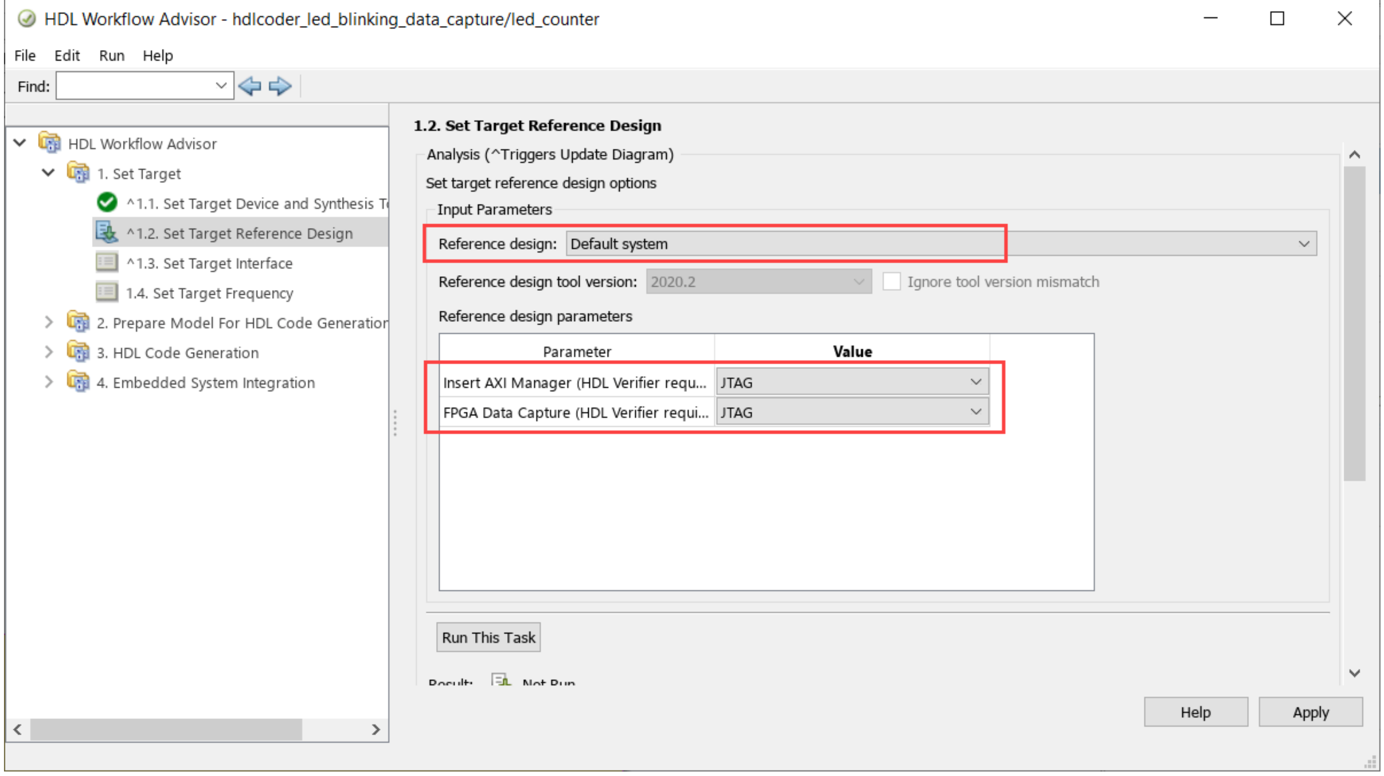


Generate HDL IP Core

Start HDL Workflow Advisor from the model and run through the IP Core Generation workflow. For a step-by-step guide, see "Getting Started with Targeting Xilinx Zynq Platform" on page 39-98.

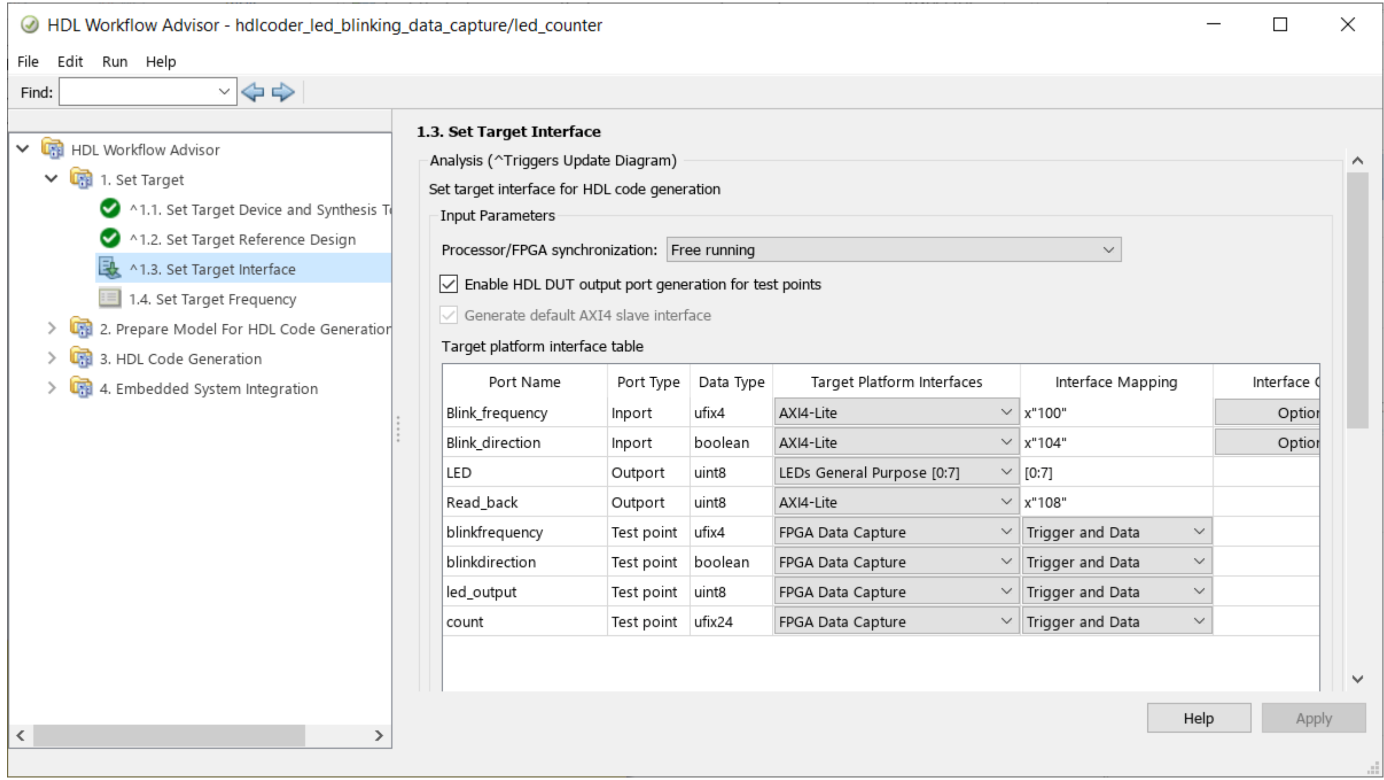
1. In step 1.1, set **Target workflow** to `IP Core Generation` and **Target platform** to `Xilinx Zynq ZC702 evaluation kit`. Click **Run This Task**.

2. In step 1.2, set **Reference design** to `Default system`. Set **Insert AXI Manager (HDL Verifier required)** and **FPGA Data Capture (HDL Verifier required)** to `JTAG`. Click **Run This Task**.

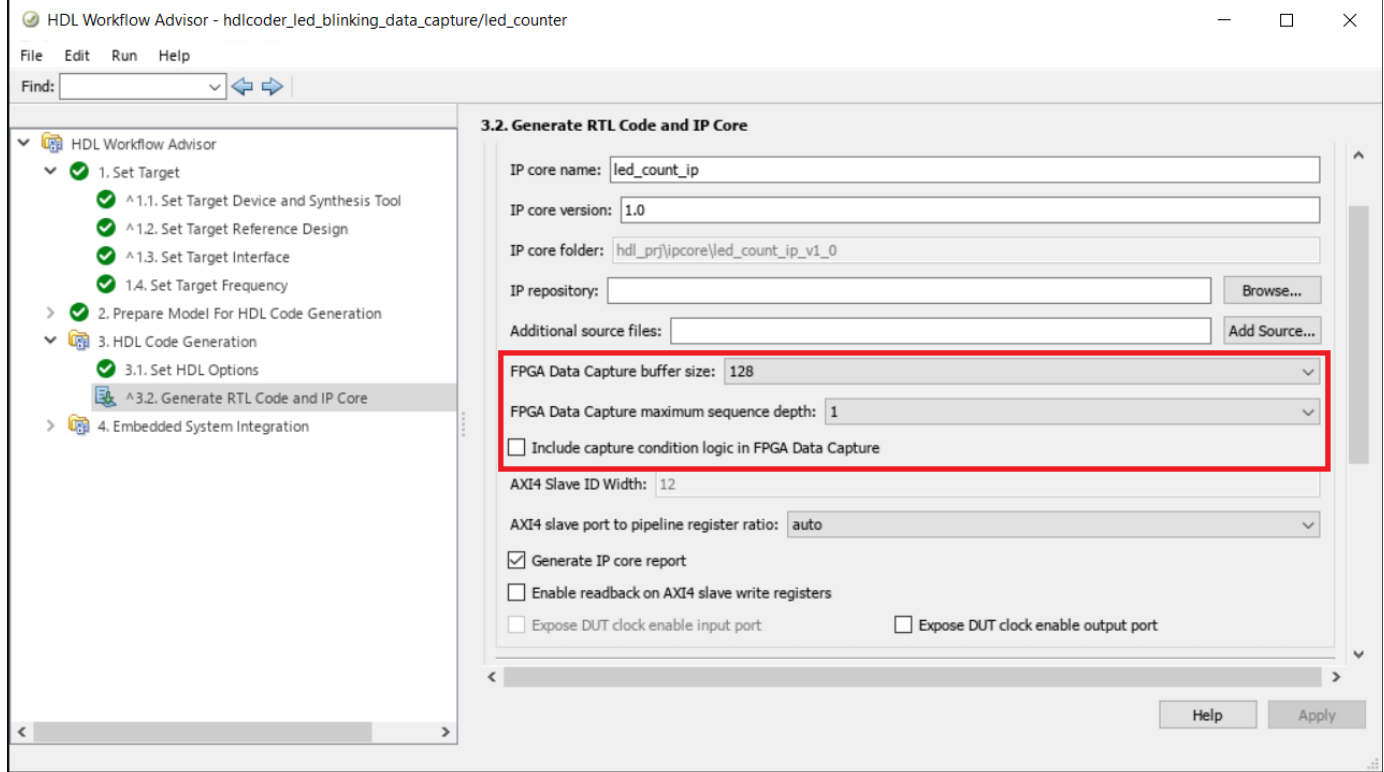


3. In step 1.3, select **Enable HDL DUT output port generation for test points**.

4. In step 1.3, set the interface of the **blinkfrequency**, **blinkdirection**, **led_output**, and **count** ports to `FPGA Data Capture`. Click **Run This Task**.



5. Run through the remaining workflow steps till step 3.1. In step 3.2, you can configure these FPGA Data Capture IP parameters: **FPGA Data Capture buffer size**, **FPGA Data Capture maximum sequence depth**, and **Include capture condition logic in FPGA Data Capture**. This example uses default values of these parameters. Click **Run This Task**.



6. Run through the remaining workflow steps to generate the HDL IP and program the target device.

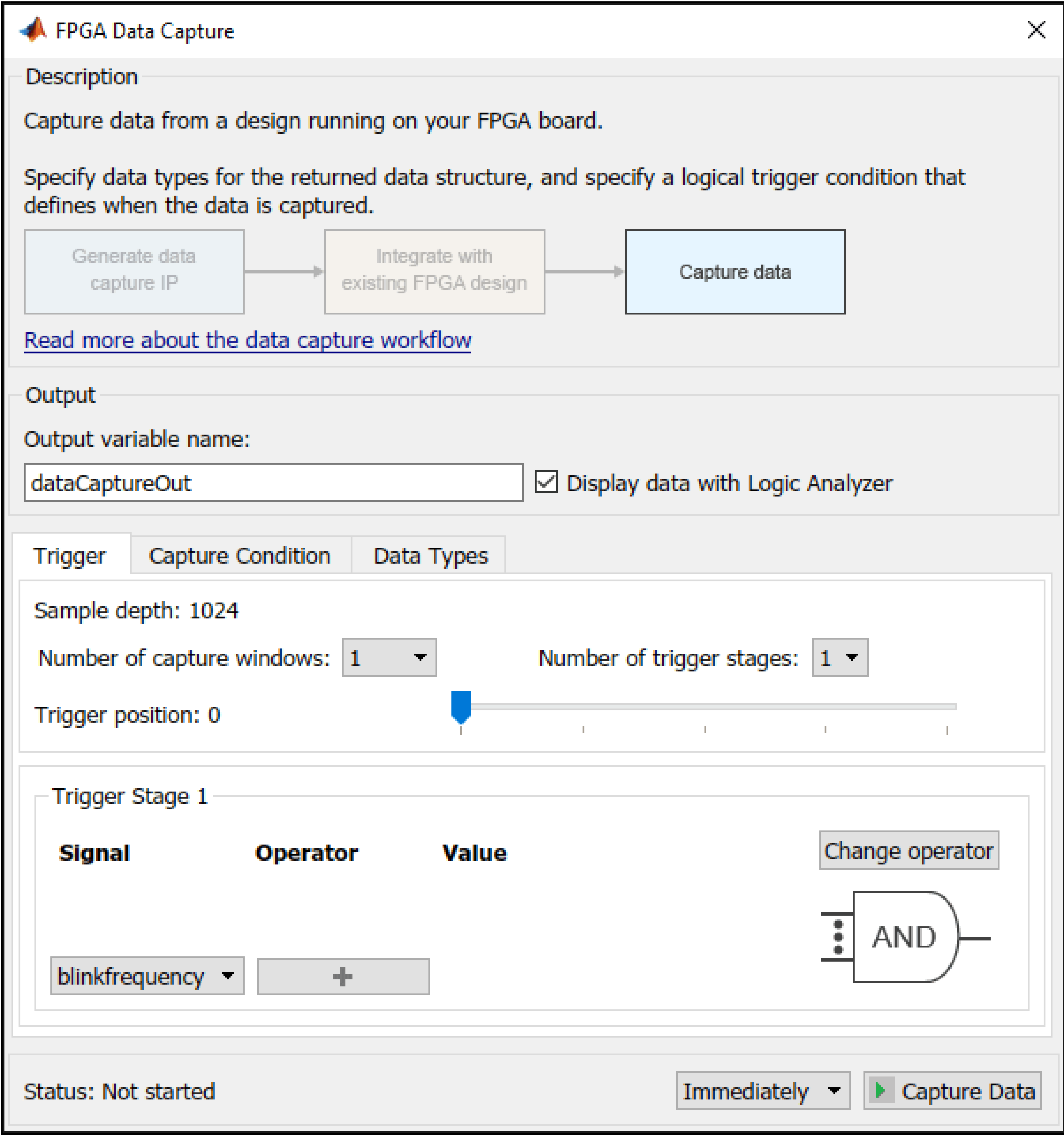
Capture and Display Data from IP Core

Next, capture data from the Zynq board.

Locate the FPGA Data Capture launch script. For this example, the script is in your HDL code generation directory: `hdl_prj/ip_core/led_count_ip_v1_0/fpga_data_capture/launchDataCaptureApp.m`. You can also locate this script in the code generation report.

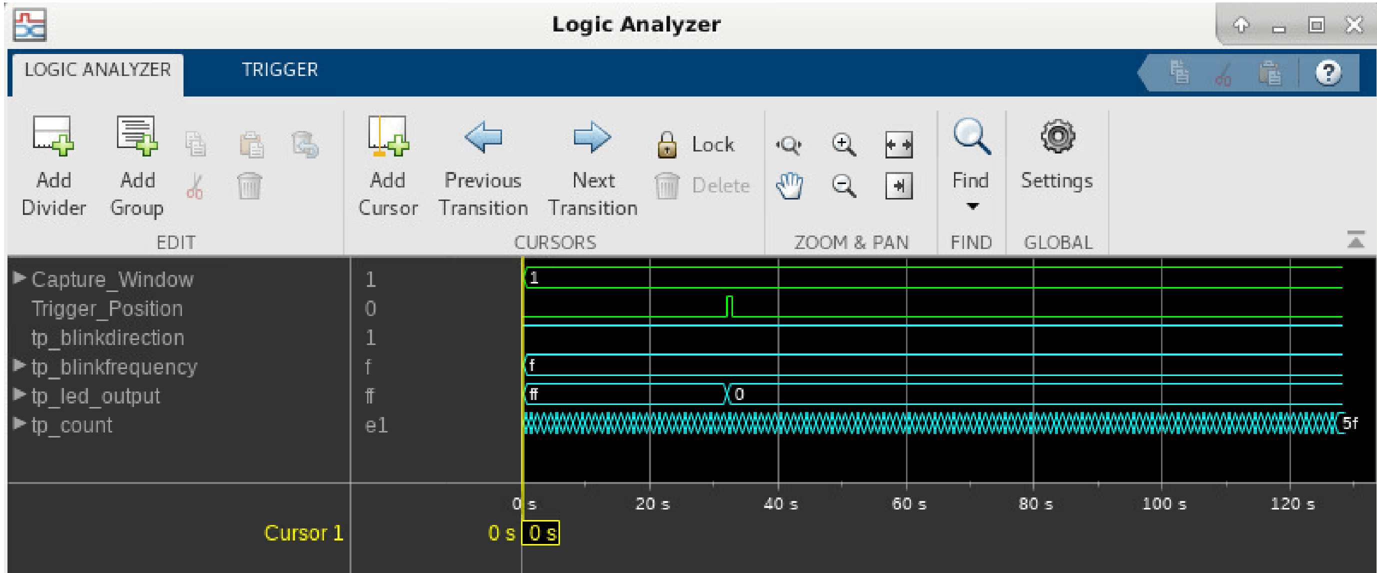
Run the `launchDataCaptureApp` script in MATLAB. Add the script directory to the MATLAB path or change the current working folder.

Capture Data in Blocking Mode



Execute the script to launch the **FPGA Data Capture** tool. By default, FPGA Data Capture works in blocking mode. To capture data from the FPGA without setting a trigger condition, click **Capture Data**.

Alternatively, you can capture data with a trigger condition. For example, set the trigger condition `led_counter == 0` and the trigger position 32. Then, click **Capture Data** again.

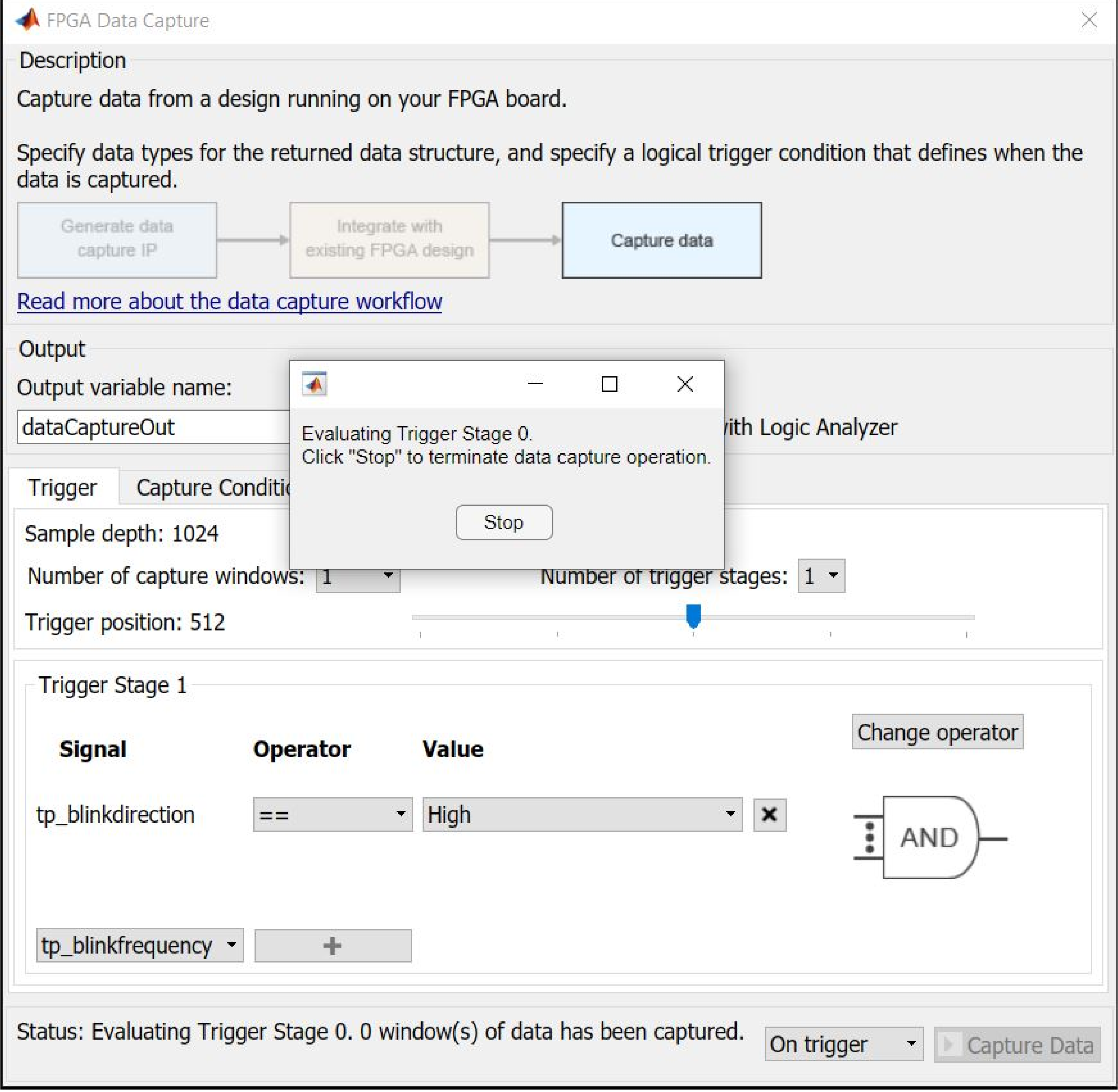


Capture Data in Nonblocking Mode

Execute the script to launch the **FPGA Data Capture** tool and create the `fpgadc_obj` object in the workspace. Change the capture mode to nonblocking by executing the following command at the MATLAB command prompt.

```
fpgadc_obj.CaptureMode = 'nonblocking';
```

FPGA Data Capture in nonblocking mode allows simultaneous use of FPGA Data Capture and AXI Manager. For more information, see “Simultaneous Use of FPGA Data Capture and AXI Manager” (HDL Verifier). Now set a trigger condition where `led_blink_direction == 1` and set a trigger position of 512. Then, click **Capture Data**.



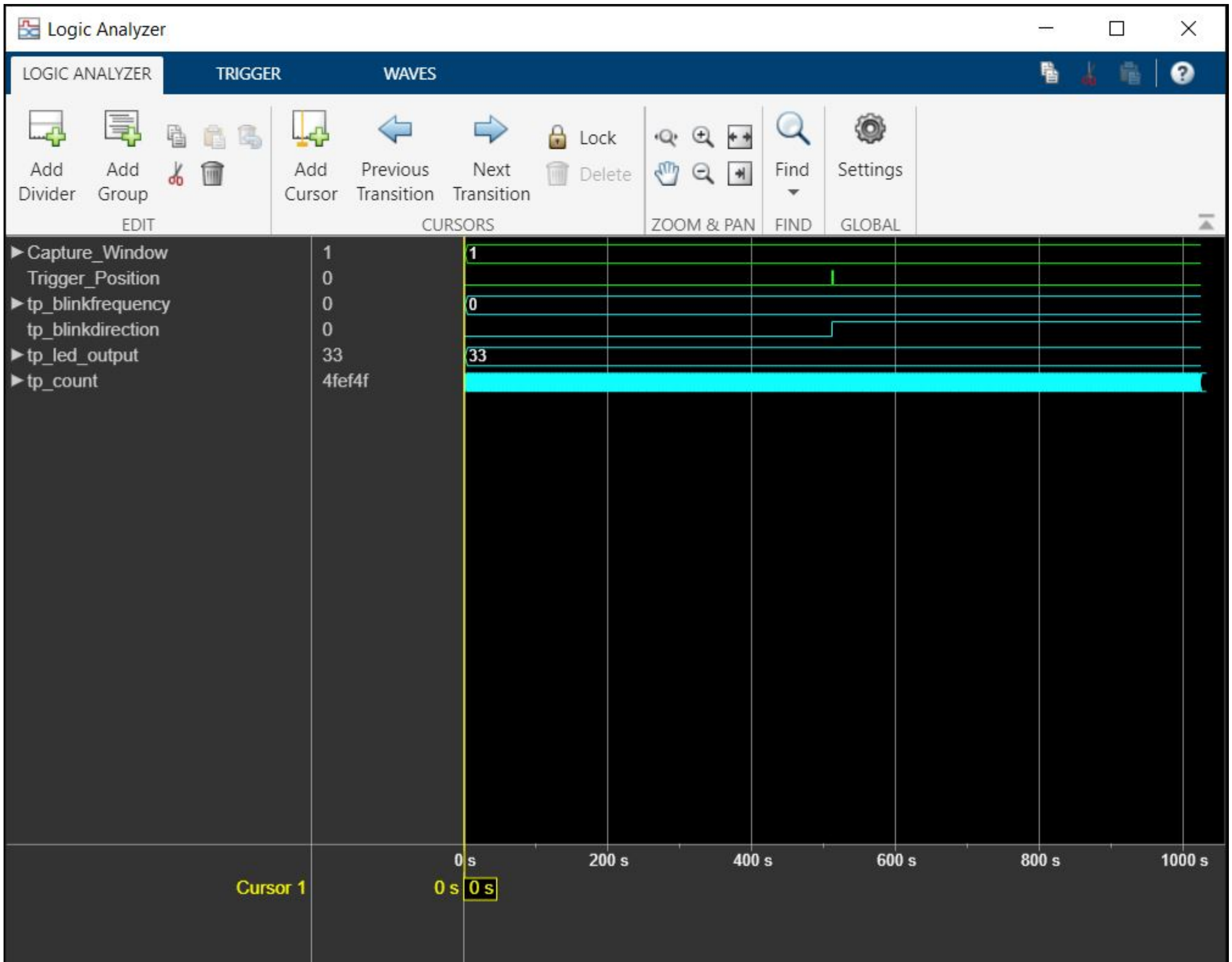
FPGA Data Capture waits for the trigger condition. As FPGA Data Capture allows AXI Manager to perform read and write operations in nonblocking mode.

Next, in FPGA Data Capture, set **tp_blinkdirection** to `High`. Then, create an AXI Manager object in MATLAB.

```
axi_manager_obj = aximanager('Xilinx');
```

```
axi_manager_obj.writememory('0x40010104',1);
```

FPGA Data Capture captures the data in the **Logic Analyzer**. The figure shows the change in the blink direction.



Field-Oriented Control of a Permanent Magnet Synchronous Machine on a Xilinx Zynq Platform

This example models a field-oriented controller (FOC) for a permanent magnet synchronous machine (PMSM) on the Xilinx® Zynq®- UltraScale+™ MPSoC target. This example uses the Trenz Electronic™ Motor Control Development Kit TE0820 revision 4. If you do not have the required hardware, you can use this example to help you develop a controller for your own hardware configuration.

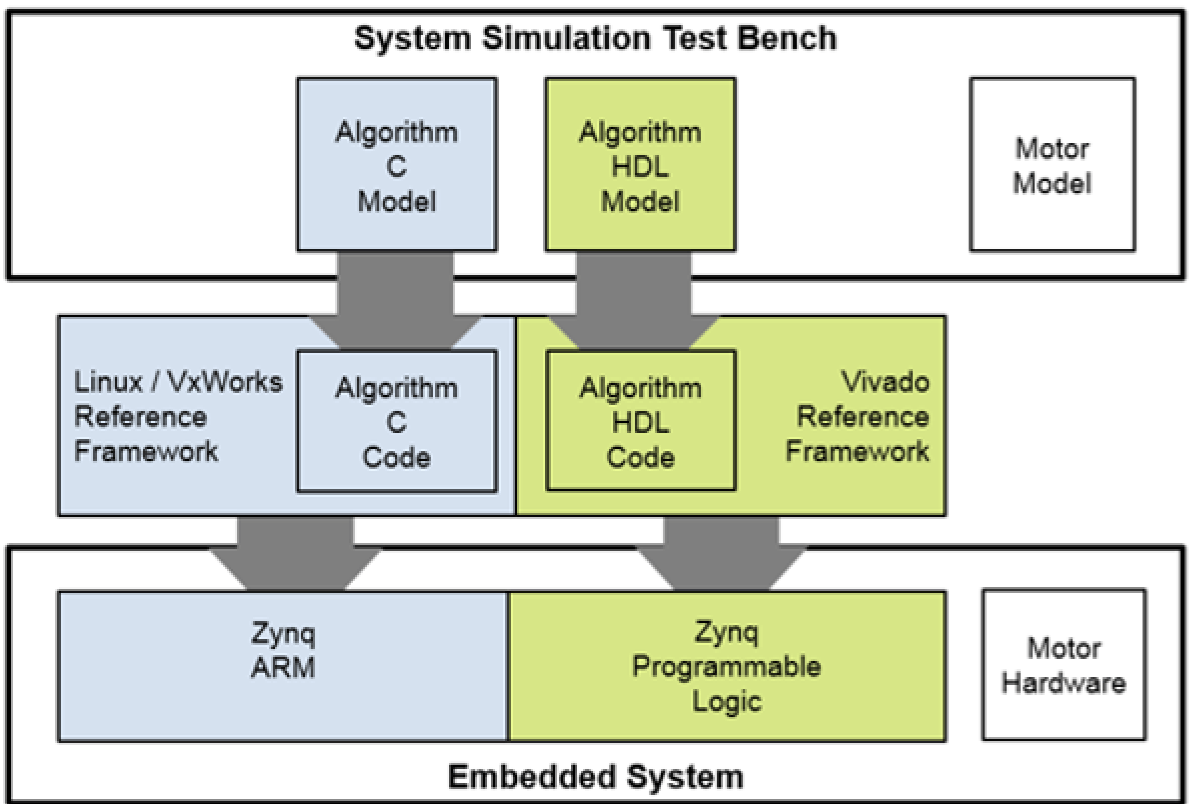
Requirements

- “Install Support for Xilinx Zynq Platform” (Embedded Coder)
- Trenz Electronic™ Motor Control Development Kit TE0820 revision 4

For more information about the hardware, see Trenz Electronic Motor Control Development kit support from Simulink.

Introduction

This image shows the general process of creating, simulating, and deploying a design of a controller algorithm onto an embedded hardware board.



You simulate a system test bench to gain insight into the behavior of the controller algorithm design. Explore the design to see how the algorithm is partitioned. The high-rate portion of the algorithm is

partitioned into a model that is configured for HDL code generation. The low-rate portion of the algorithm is partitioned into a model that is configured for C code generation. Generate C and HDL code from these models and learn how you can integrate this code into your design.

You can automate deployment of the algorithmic code into reference frameworks for the processor and programmable logic. Then, execute a test on the deployed application, log the results, and compare them to the simulation results.

Set up the Xilinx Zynq hardware board before starting the example. This example deploys a bitstream and ARM® executable to a Xilinx Zynq device. To ensure the correct setup of your environment, complete the “Getting Started with Targeting Xilinx Zynq Platform” on page 39-98 example with your hardware configuration before starting this example.

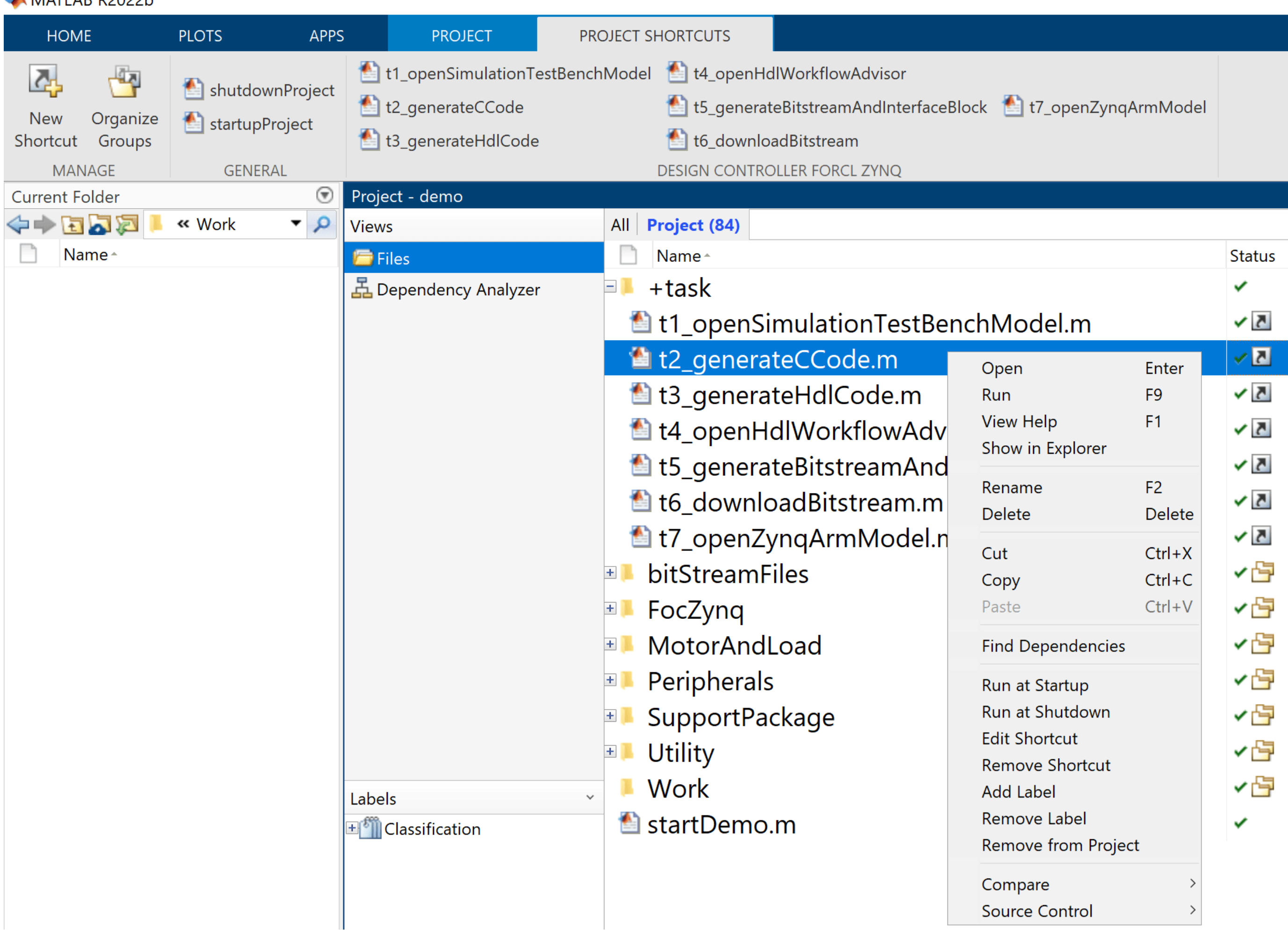
Simulate Algorithm Behavior

First. open the example project, inspect the controller models, and verify the controller behavior through simulation. This example is packaged as a project. For more information on Simulink® projects, see “What Are Projects?”

1. Open the example project by entering this command at the MATLAB® command prompt.

```
zynqPMSMFOCStart
```

2. Click the **Project Shortcuts** tab to view the shortcuts to the files and folders that this example uses.



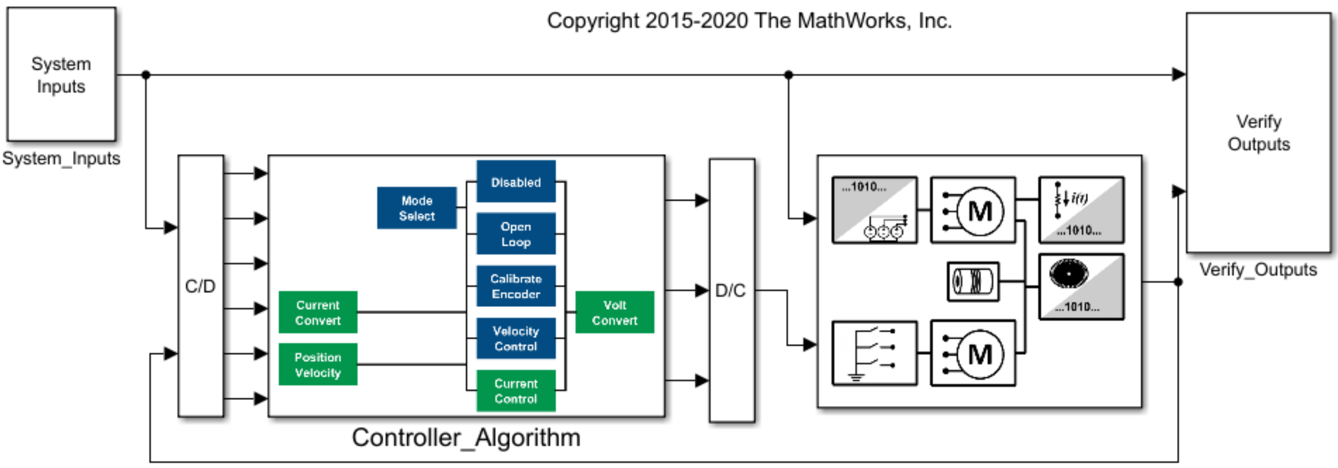
3. Run `task.t1_openSimulationTestBenchModel` to open the `focZynqTestBench` model.

```
task.t1_openSimulationTestBenchModel
```

The `Motor_And_Load` subsystem consists of a mathematical model of a surface PMSM, motor load, encoder, and current sensor. The `Controller_Algorithm` subsystem includes:

- I/O engineering unit conversion
- Electrical position calculation
- Rotor velocity calculation
- Mode scheduler, and four control modes: disabled, open loop velocity control, encoder calibration, and closed loop velocity control.

The `C/D` and `D/C` subsystems convert the data from continuous- and variable-time step solvers and floating-point data types to discrete- and fixed-time step solvers and fixed-point data types.

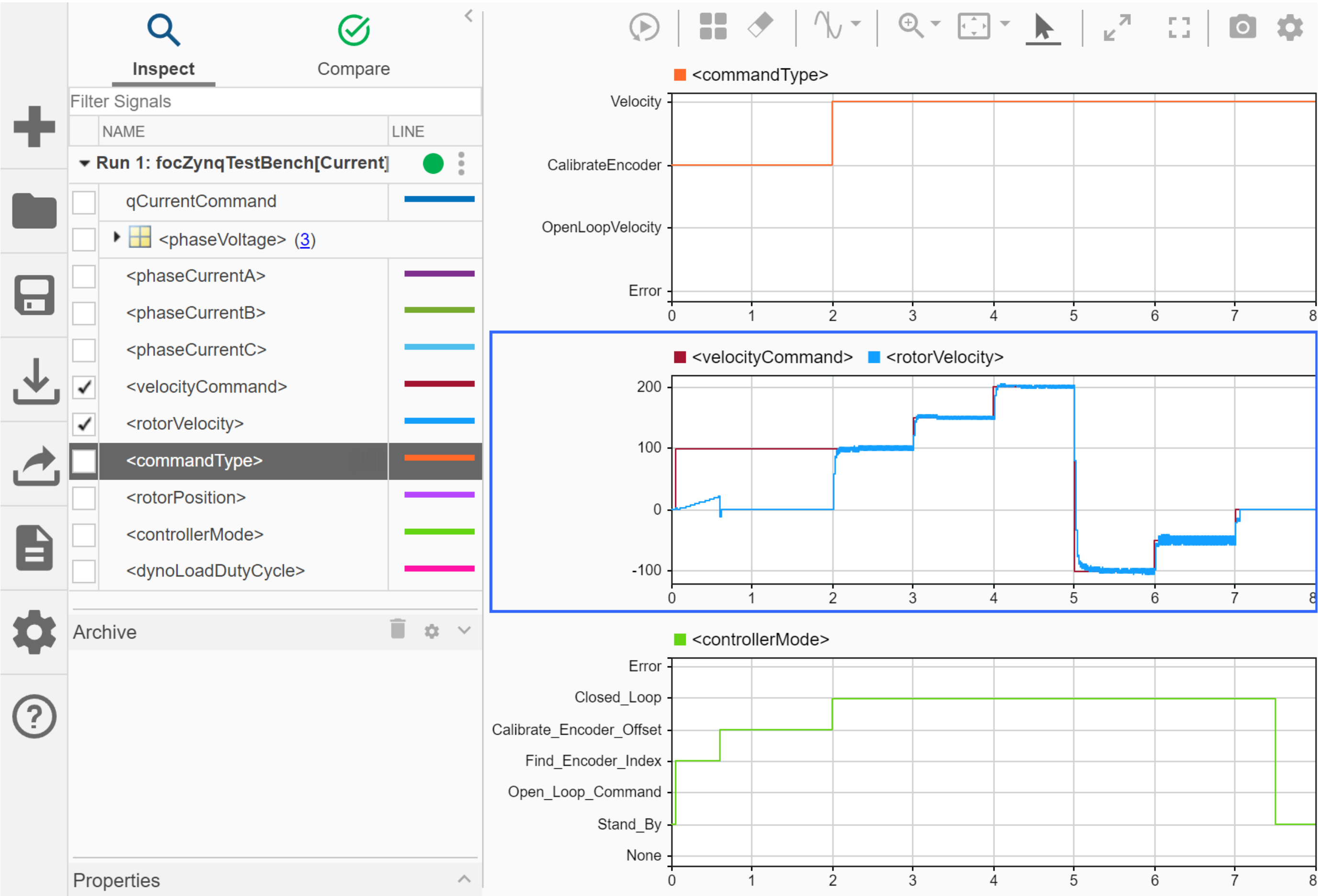


If you do not have the Specialized Power Systems Library in Simscape™ Electrical™ installed, the `Motor_And_Load` subsystem contains a block that enables you to simulate the model with default motor and load parameters. In this case, you are not able to explore or modify parameters.

4. On the **Simulation** tab click **Run** to simulate the model.

5. After simulation completes, open the **Simulation Data Inspector**. On the **Simulation** tab, click **Data Inspector**. For more information on the Simulation Data Inspector, see Simulation Data Inspector.

6. In the **Simulation Data Inspector**, select the **<commandType>**, **<velocityCommand>**, **<rotorVelocity>**, and **<controllerMode>** signals.



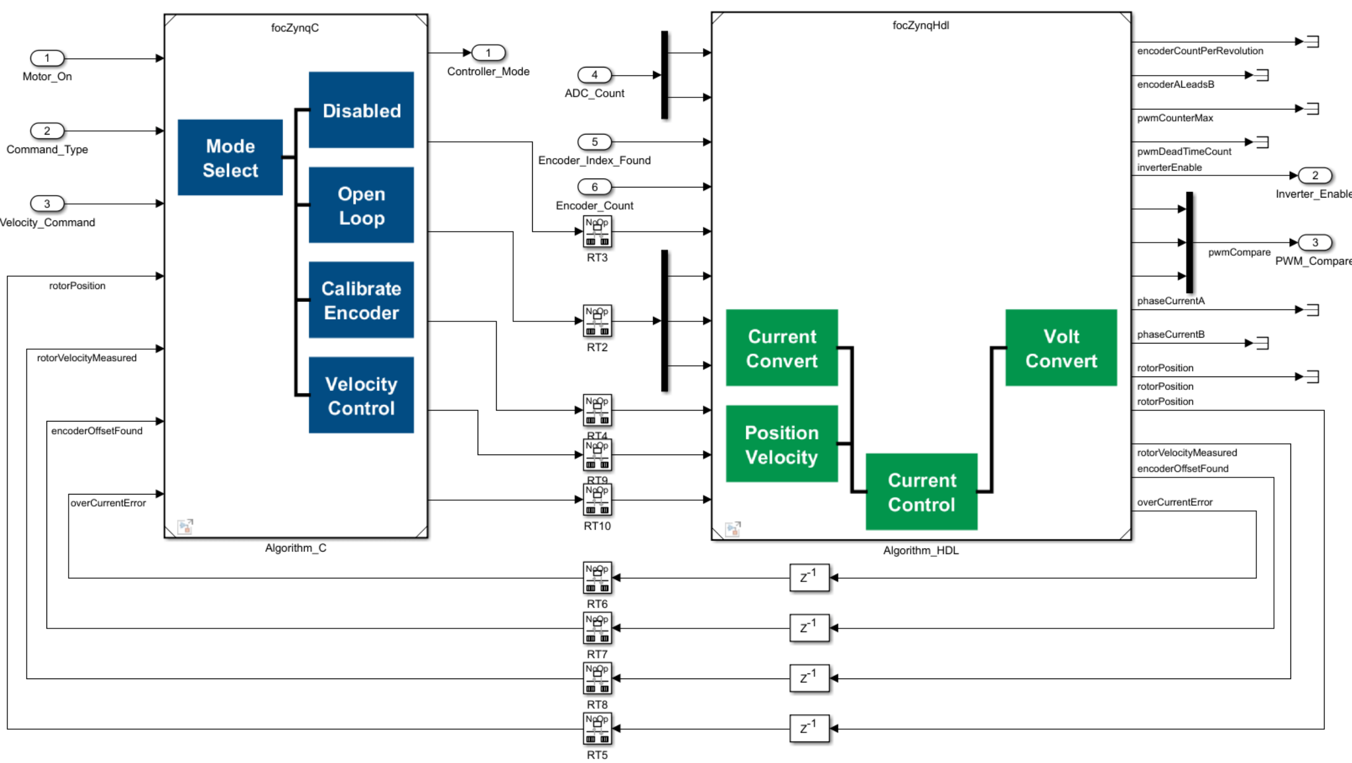
For the first two seconds, the controller receives commands to calibrate the encoder position sensor. The encoder position sensor must be calibrated before the controller can achieve closed-loop control. During the first portion of position calibration, the motor accelerates using open-loop control to identify the index pulse of the encoder. After the index is found, the controller commands and holds a zero position until the encoder offset is identified. During this period, the velocity is zero. After two seconds, the controller changes into closed-loop control and follows the commanded velocity profile. During closed-loop velocity control, the FOC regulates phase current in the PMSM.

Partition Controller Algorithm and Generate Code

Next you partition the controller algorithm into complementary software and hardware implementations by generating C and HDL code for the software and hardware implementations, respectively. The reports created during code generation show how you can integrate this code into your own embedded design.

1. In the `focZynqTestBench` model, open the `Controller_Algorithm` subsystem. The controller algorithm contains the `Algorithm_C` and `Algorithm_HDL` blocks, which reference the `focZynqC` and `focZynqHDL` models, respectively. The `focZynqC` model contains the portion of the algorithm to be implemented in software. The `focZynqHDL` model contains the portion of the algorithm to be implemented on the hardware.

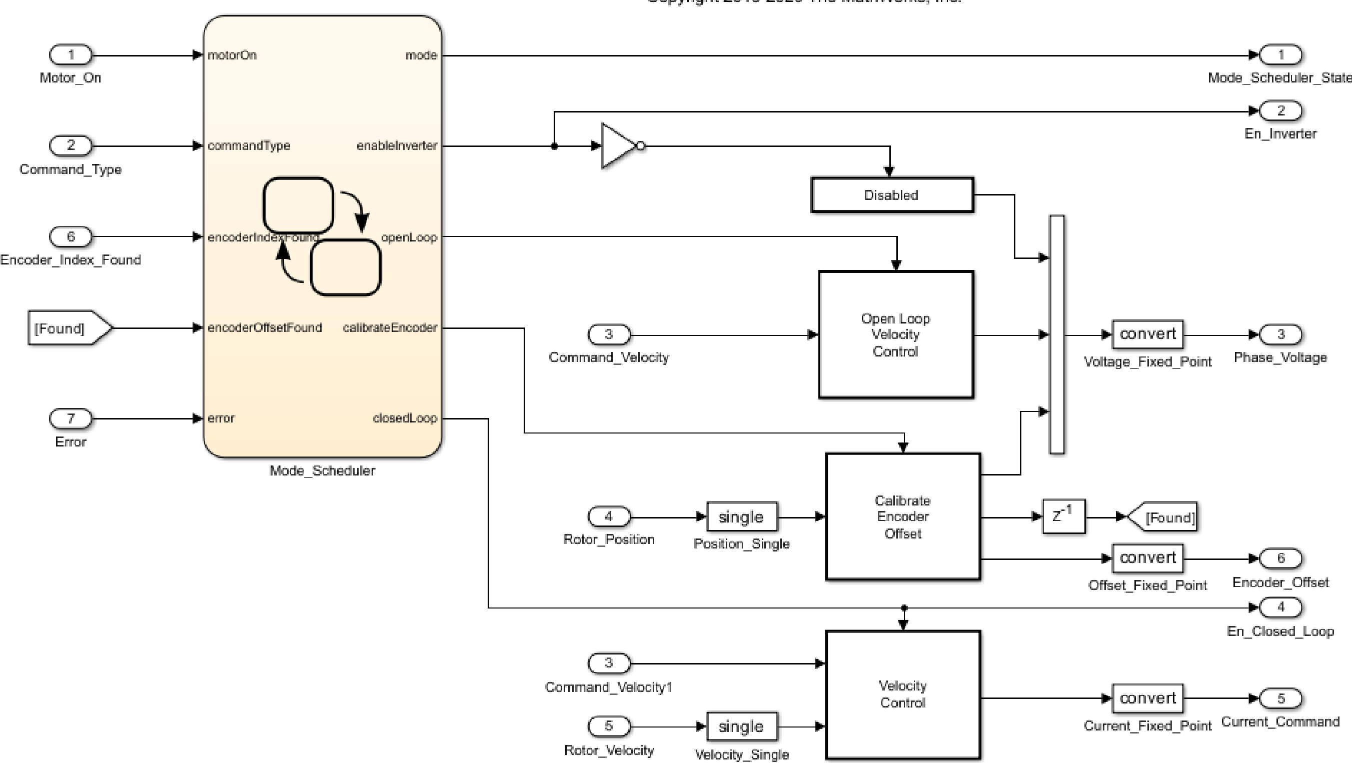
Controller Algorithm



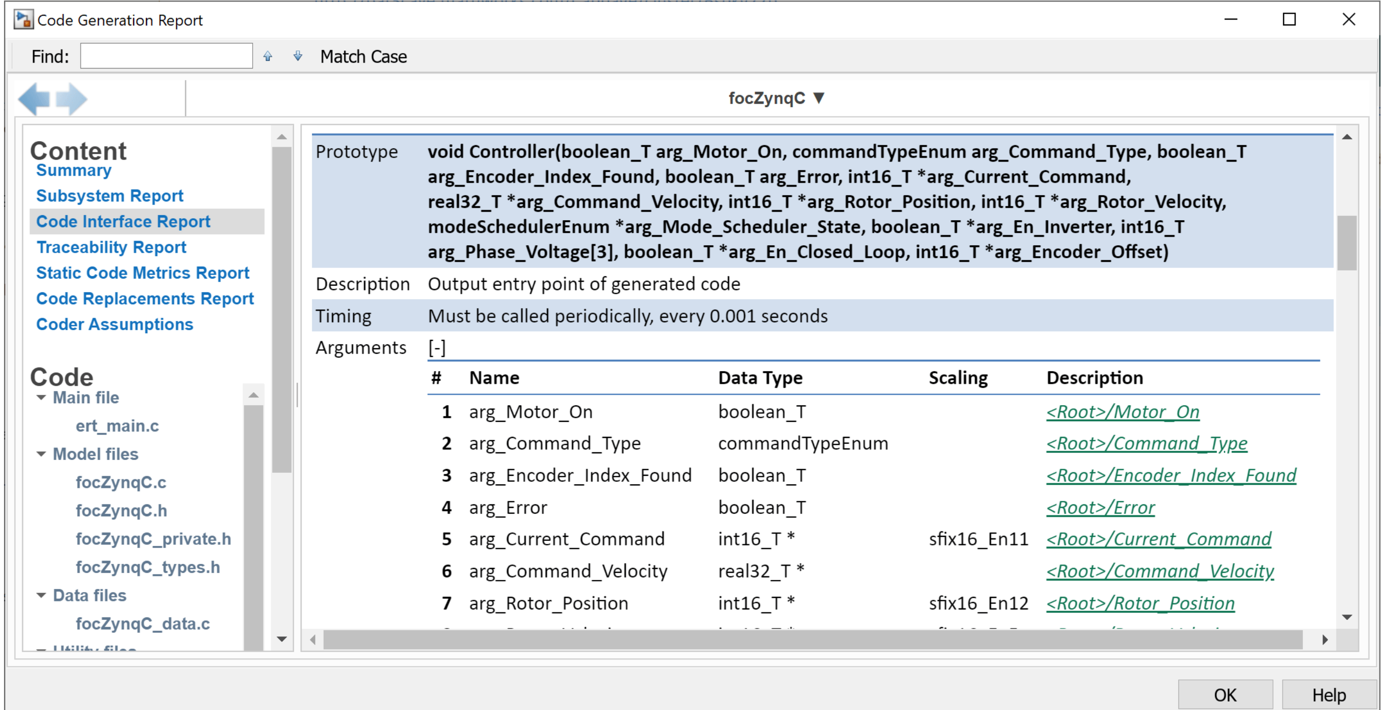
2. Run the `task.t2_generateCCode` function to open the `focZynqC` model, generate C code, and generate a report.

```
task.t2_generateCCode
```

The `focZynqC` model contains the mode scheduler, velocity control loop, open-loop velocity controller, and a routine that automatically calibrates the encoder offset.



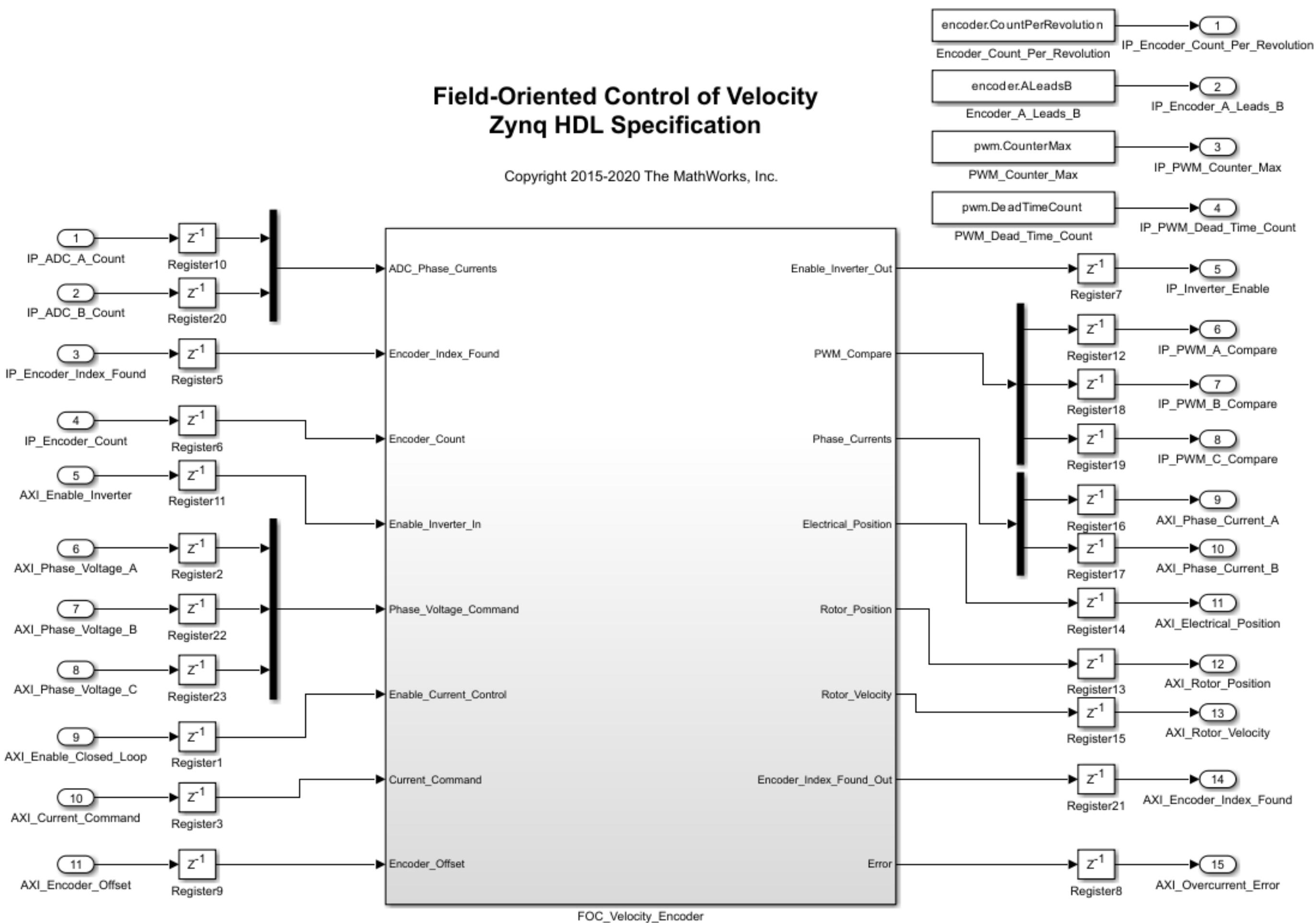
3. The code generation report shows how the generated code corresponds to the model. Click **Code Interface Report** to view the function interface of the code. The C code is portable and can integrate with any floating-point embedded processor that uses ANSI-C compiler. For more information on the Code Generation Report, see “Reports for Code Generation” (Embedded Coder).



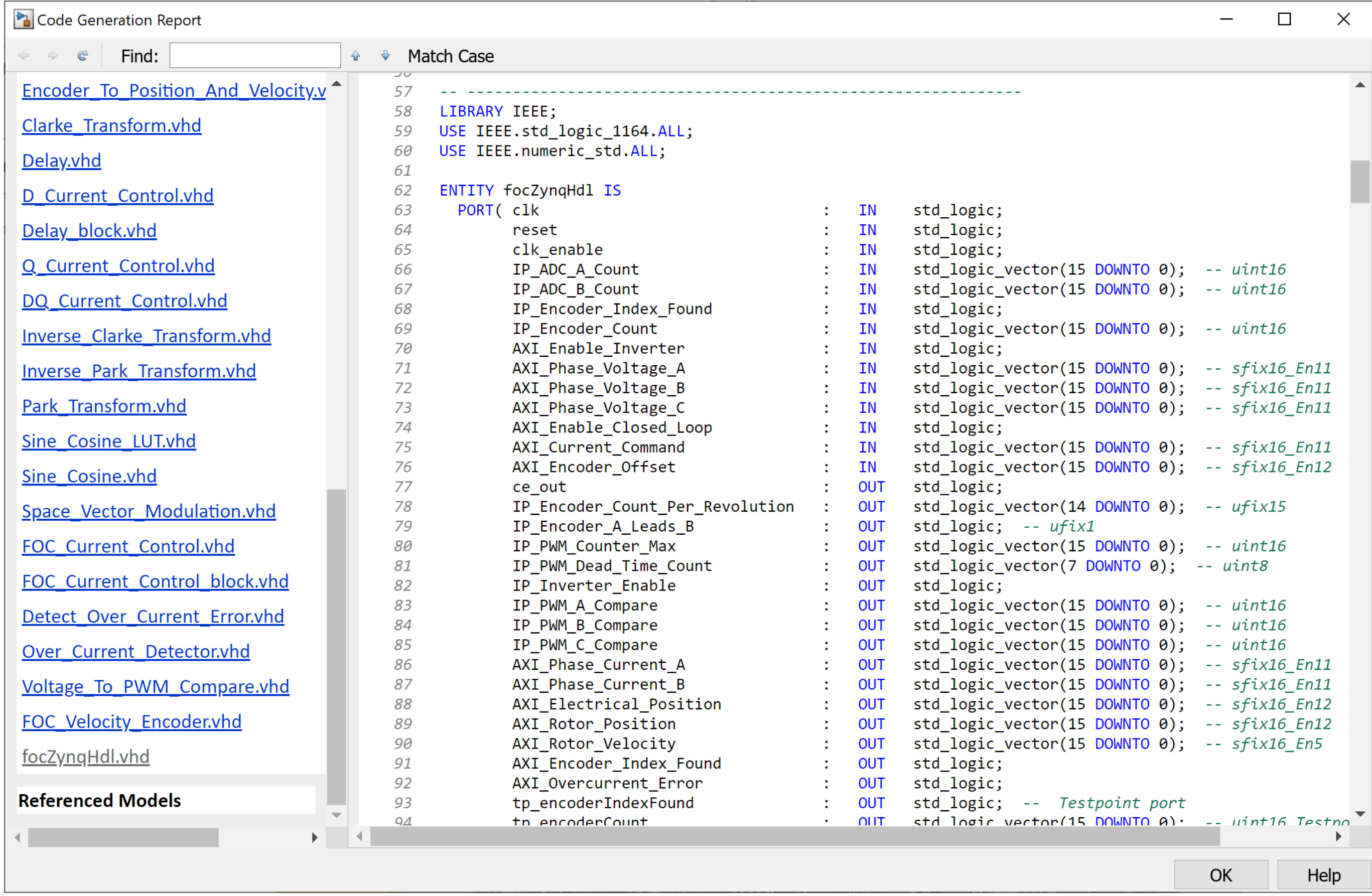
4. Run the `task.t3_generateHdlCode` function to open the `focZynqHdl` model, generate HDL code, and generate a report.

```
task.t3_generateHdlCode
```

The `focZynqHdl` model contains the electrical position calculation, rotor velocity calculation, overcurrent checks, and the field-oriented controller.



5. The code generation report shows how the HDL code corresponds to the model. Under **Generated Source Files** select the `focZynqHdl.vhd` file, which contains the entity specification. The HDL code for the algorithm is portable and can integrate with any FPGA that supports VHDL® code.

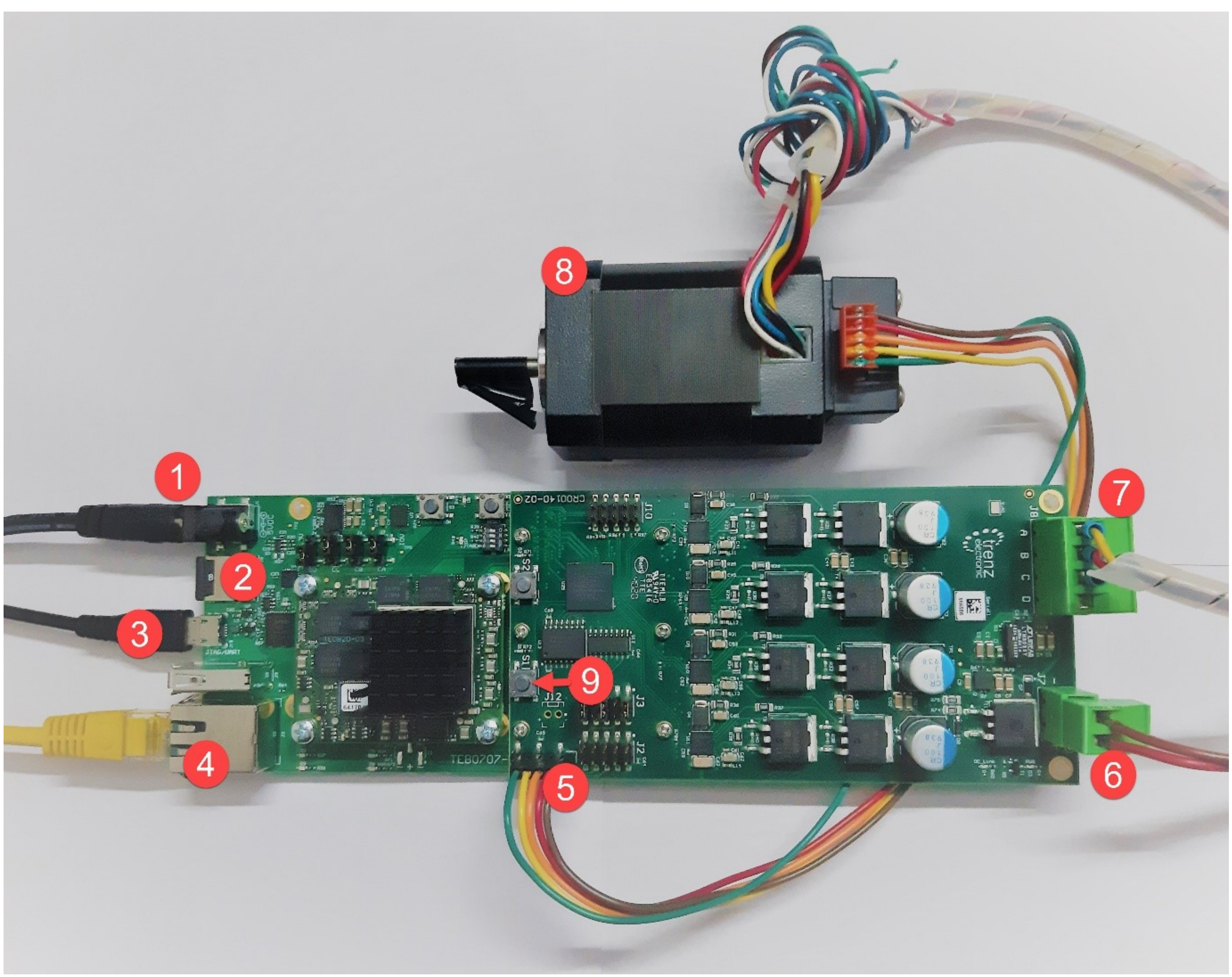


Set Up Xilinx Zynq Platform and Motor Boards

Next, set up and connect the hardware boards.

1. Run the hardware setup for the Xilinx Zynq platform. For information on the hardware setup, see “Install Support for Xilinx Zynq Platform” (Embedded Coder). You must install both Embedded Coder® Support Package for Xilinx Zynq Platform and HDL Coder™ Support Package for Xilinx FPGA and SoC Devices.

2. Connect the Trenz board as shown in the image.



The image shows these components:

1 5V DC power supply

2 SD card

3 Micro USB cable for UART and JTAG

4 Ethernet connector

5 Encoder connector

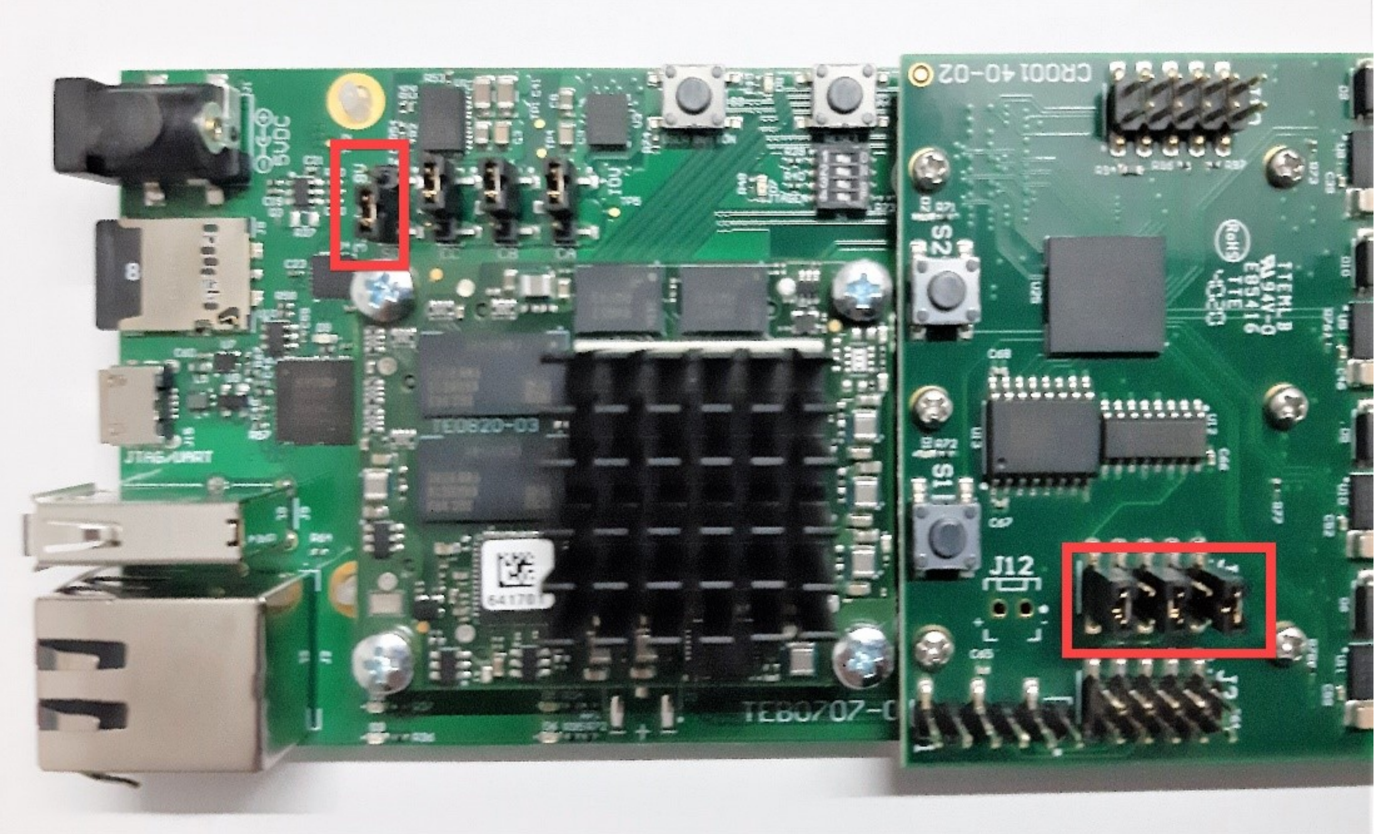
6 24 V DC power supply

7 Motor power cable (A, B, C)

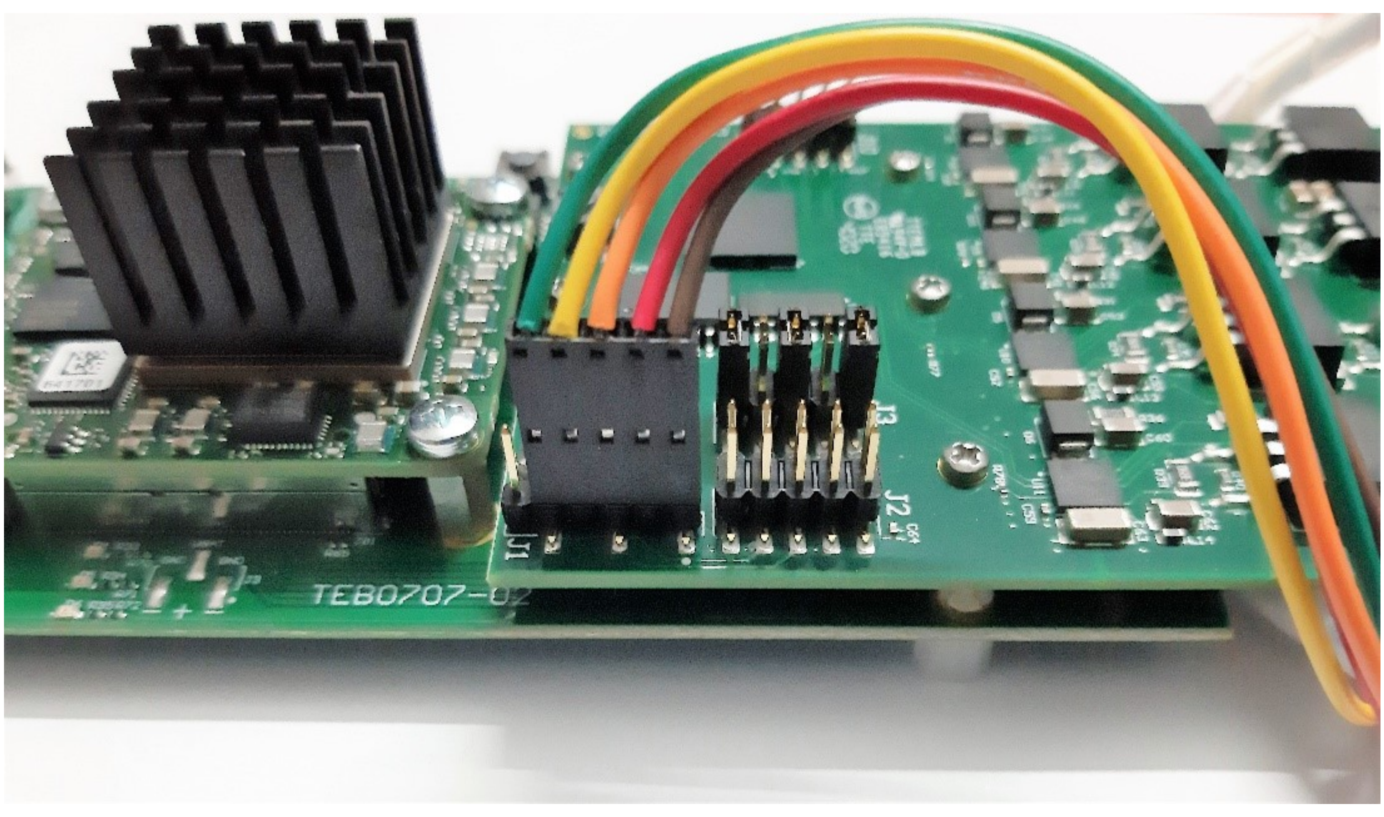
8 24 V brushless DC motor

9 Switch 1 (S1), which controls power to the driver board

3. Ensure that jumper J4 on the carrier module is set to SD and J3 on the motor driver card is set up for a single ended encoder.



4. Insert the encoder cable according to the picture below. For more information on the jumper settings, visit the Trenz Electronic website.



5. Download the Trenz TE0820 Linux® Image, extract the ZIP archive, and copy the contents to the microSD card. Insert the microSD card in connector J8.

6. After you program FPGA bitstream, press the S1 switch located on the motor driver card once to connect the 24V to the MOSFETS. In case of any unexpected behavior from the device, use the S1 switch to disconnect the power.

Deploy Bitstream to Programmable Logic

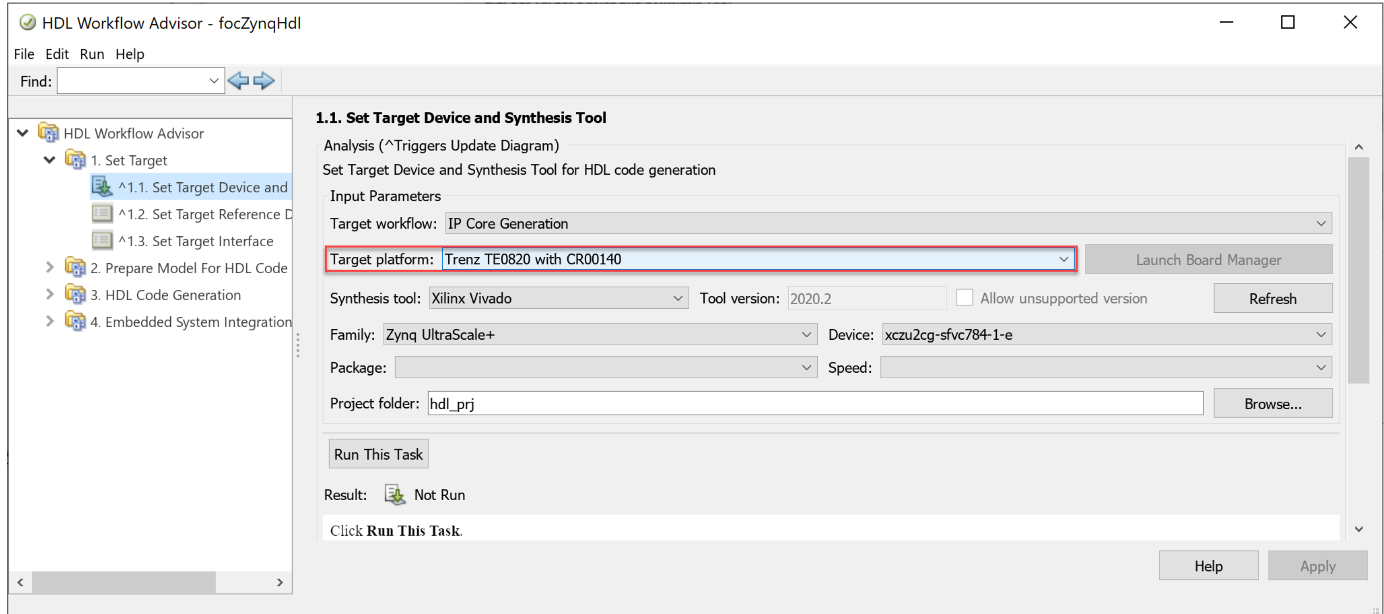
Next, use the HDL Workflow Advisor to generate HDL code for the algorithm, package the HDL into an IP core, integrate the IP core into a Xilinx reference design, and create a bitstream. The path to the reference design used in this example is added automatically when you open the project if the `HDL Coder Support Package for Xilinx FPGA and SoC Devices` was already installed. Alternatively use this command to add the reference design files to your path:

```
addpath(genpath(fullfile(hdlcoder_amd_examples_root, 'TE0820FOC')));
```

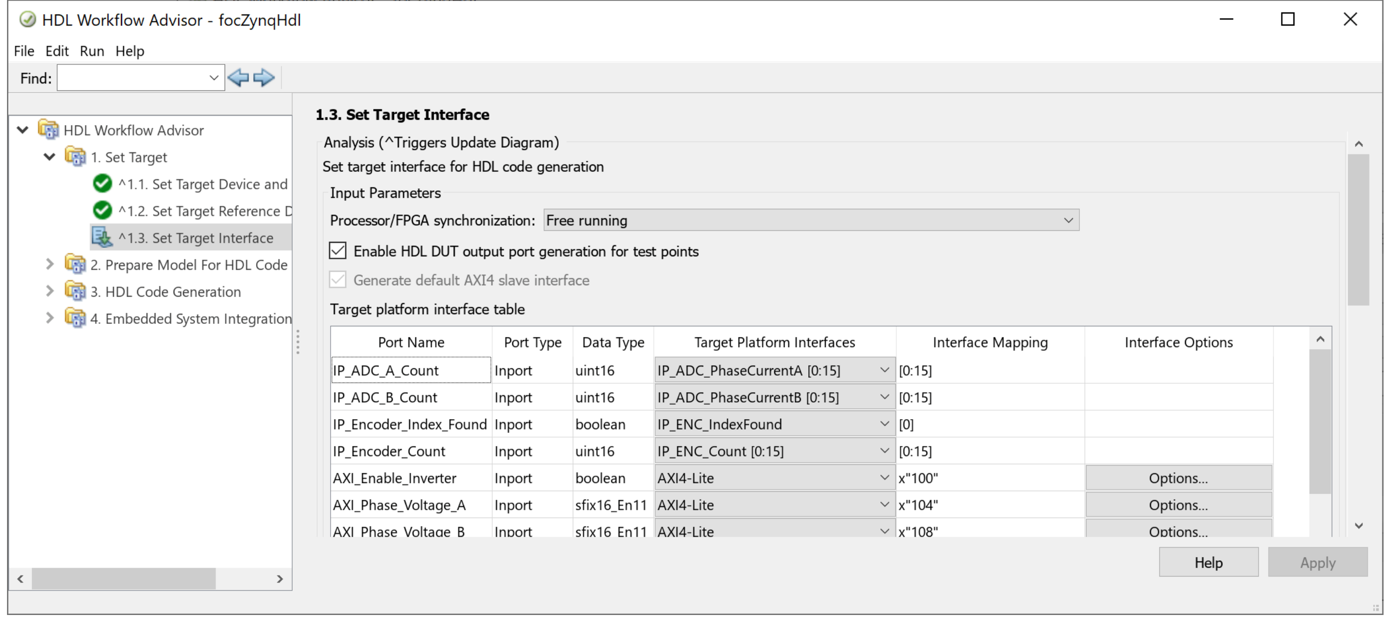
1. Run the `task.t4_openHdlWorkflowAdvisor` function to open HDL Workflow Advisor.

```
task.t4_openHdlWorkflowAdvisor
```

2. In the **HDL Workflow Advisor** in the **1. Set Target > 1.1 Set Target Device and Synthesis Tool** group, set the **Target platform** to `Trenz TE0820 with CR00140`. `Trenz TE0820 with CR00140` is a Vivado® reference design that contains the ADC, encoder, and PWM components. For information on this reference design, see “Define Custom Board and Reference Design for Zynq Workflow” on page 40-252.

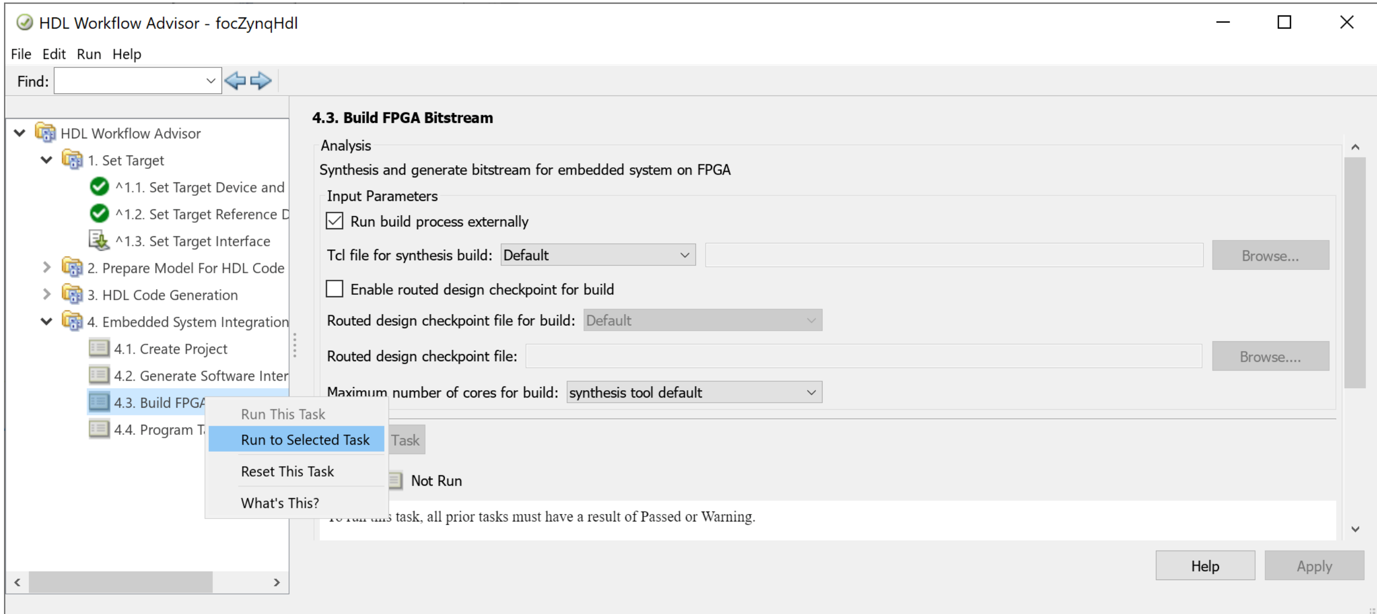


3. Select **1.2. Set Target Interface** to identify the ports. The entries in the **Target Platform Interfaces** column of the table that have the prefix IP refer to connections that are registered with the Trenz motor control reference design.

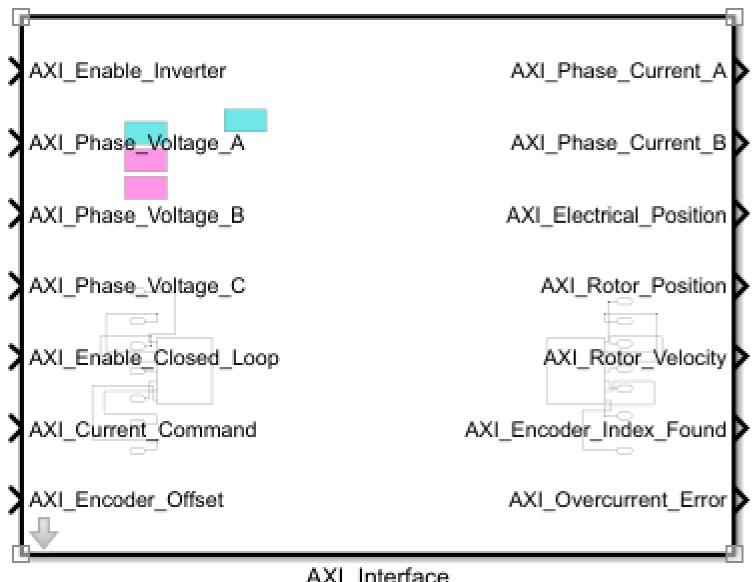


4. Select **4.3 Build FPGA Bitstream > Run to Selected Task** or run the `task.t5_generateBitstreamAndInterfaceBlock` function from the project to generate the HDL code for the algorithm and create the FPGA bitstream from the Xilinx reference design.

```
task.t5_generateBitstreamAndInterfaceBlock
```



5. Follow the progress of bitstream generation in the DOS Command Prompt window. In addition to generating a bitstream the customized target generates the `focZynqHdlAxiInterfaceLib` software interface library. The library contains an AXI_Interface block. The AXI_Interface block, which contains the AXI4-Lite interface components, provides connectivity from the model deployed on the ARM processor to the model deployed on the programmable logic.



6. Run task **4.4 Program Target Device** or run the `task.t6_downloadBitstream` function from the project to program the FPGA:

```
task.t6_downloadBitstream
```

Deploy Executable to ARM processor

Next, generate C code for the controller and integrate this code with a Linux® reference framework to build, deploy, and run the model as an executable to the ARM processor. Data logged from the model running on the processor can then be compared to the results of the simulation.

1. Run the `task.t7_openZynqArmModel` function to open the `focZynqArmDeployment` model.

```
task.t7_openZynqArmModel
```

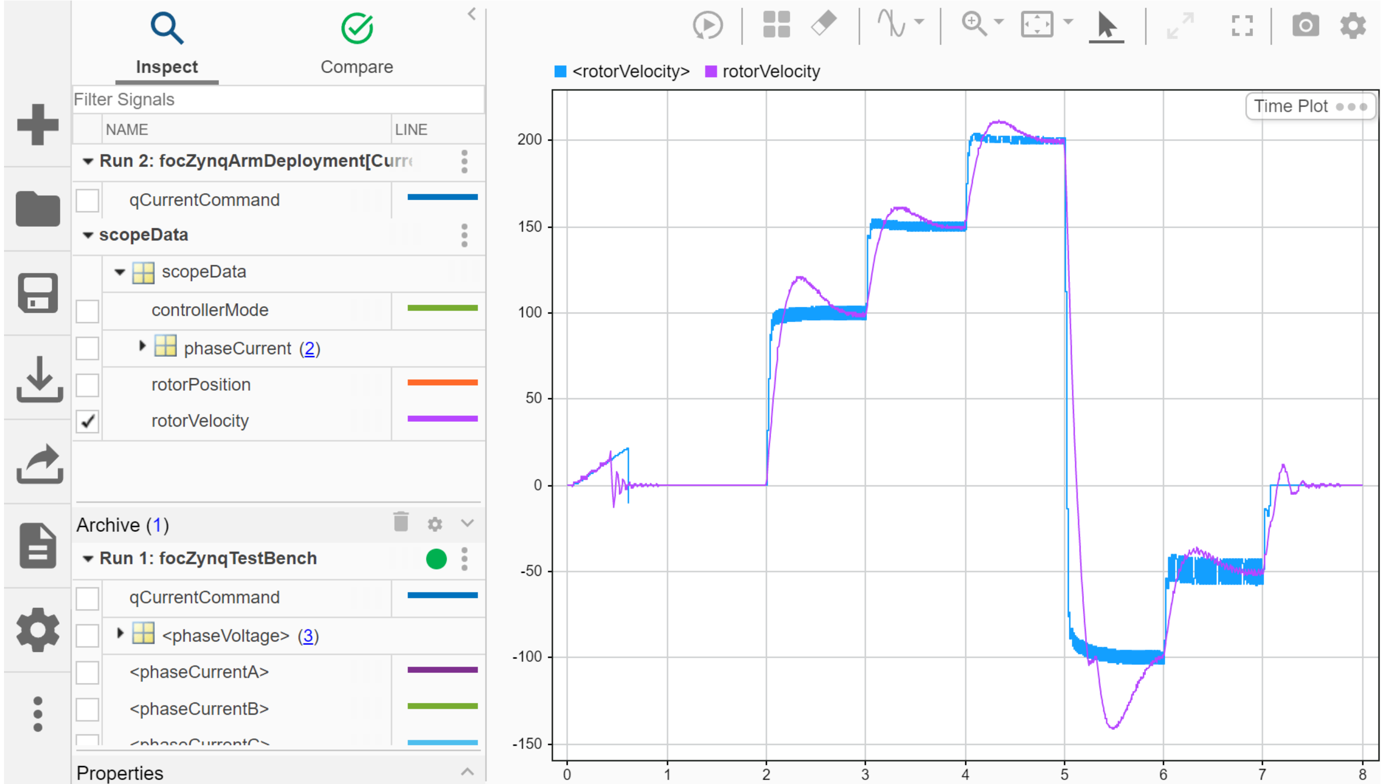
The `focZynqArmDeployment` model can generate C code, automate the integration with a Linux ARM reference framework, and deploy the executable to the ARM processor on the Xilinx Zynq platform. The deployment model references the original controller model and contains the test stimulus, scope, and AXI_Interface library block created in the Deploy Bitstream to Programmable Logic on page 40-373 section.

Field-Oriented Control of Velocity
Zynq ARM Deployment for Trenz Motor Control Kit

2. On the **Hardware** tab, click **Monitor & Tune** to build, deploy, and run the model as an executable on the ARM processor. The generated code is compiled against the reference framework to create an executable. While executing, Simulink monitors the signals and shows them in the Scope block.

3. Open the **Simulation Data Inspector** to view the logged signals and compare them to the signals logged previously from `focZynqTestBench` model. On the **Simulation** tab, click **Data Inspector**.

4. In the **Simulation Data Inspector**, select the **rotorVelocity** signal. During the encoder calibration mode, the signals initially differ because the simulated and real motors started with different rotor positions. In contrast, the closed-loop velocity control from the simulation and hardware are very similar. Differences occur because the simulation model of the motor and sensors use data sheet values and do not explicitly account for the manufacturing tolerances of the physical motor.



See Also

Related Examples

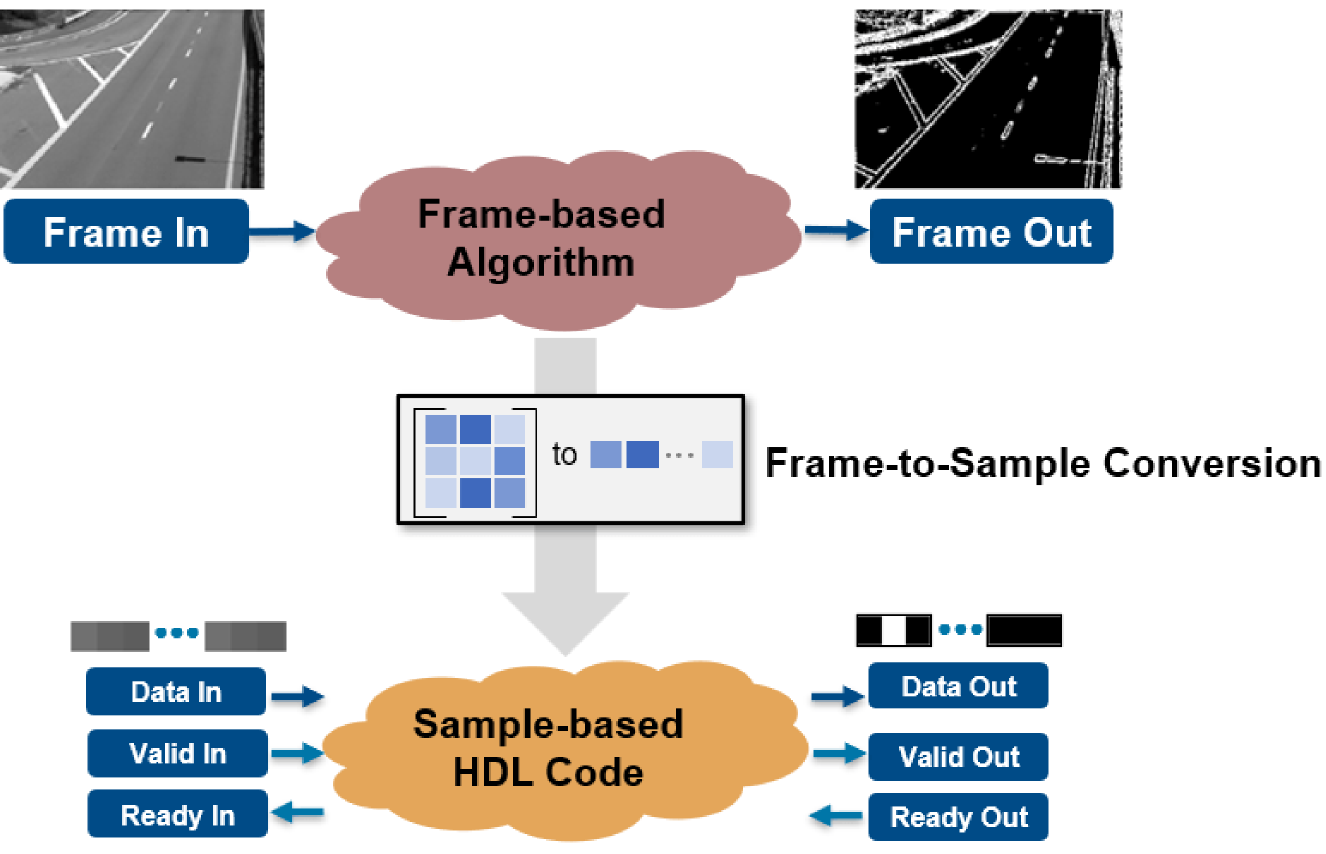
- “FOC of PMSM Using FPGA-Based Motor Control Development Kit” (Motor Control Blockset)

Deploy a Frame-Based Model with AXI4-Stream Interfaces

This example shows how to leverage the frame to sample optimization from HDL Coder™ to generate a sample-based IP core with AXI4-Stream interfaces from a frame-based Simulink® model. The generated IP core can then be deployed to hardware and verified using live streaming data from MATLAB®.

It is common to model frame-based algorithms in MATLAB and Simulink. This modeling style allows intuitive algorithm designs that process data as entire frames using frame-based operations. However, these operations are not efficient to implement on FPGA/ASIC devices, which typically process large datasets as samples. Thus, frame-based algorithms need manual conversion to their sample-based counterparts before deployment, adding time-consuming and error-prone work to the design process.

Using the frame to sample optimization, HDL Coder automates the frame to sample conversion process and generates sample-based HDL code for frame-based algorithms modeled using element-wise operations, neighborhood idioms, iterator, and reduction operations. When you use the frame to sample conversion, HDL Coder automatically transforms your frame-based algorithm into a sample-based model with valid and ready control signals and the logic to handle and align the data streams as shown below:



In this example, you:

1. Model a frame-based edge detection algorithm by using neighborhood processing functions.
2. Generate an HDL IP core with an AXI4-Stream interface.
3. Integrate the generated IP core into a reference design with a DMA controller.
4. Use a simple script to prototype the design running on hardware with live data.

Prerequisites

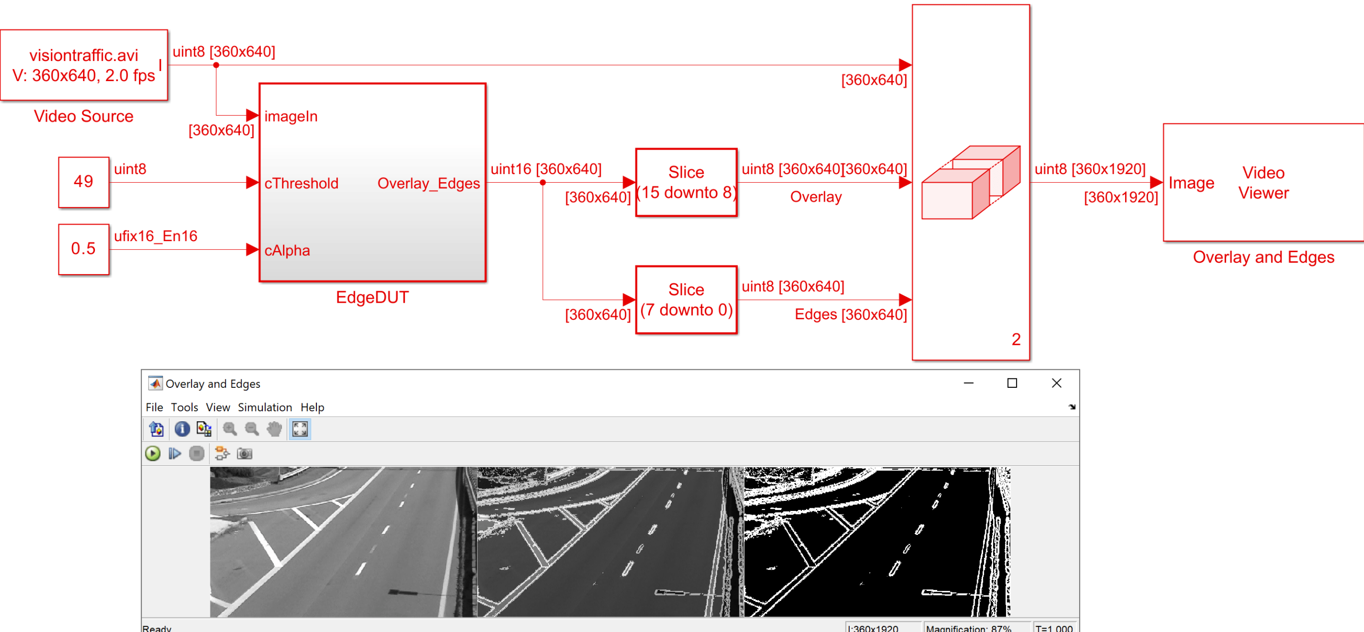
To run this example, you must have the following software installed and hardware boards setup:

- HDL Coder Support Package for Xilinx® FPGA and SoC Devices
- Xilinx Vivado® (To view the supported versions, see “HDL Language Support and Supported Third-Party Tools and Hardware”)
- Xilinx SoC board. This example uses the Xilinx Zynq ZC706 evaluation kit.
- MathWorks® firmware image on the SD card of the board. For help with SD card setup, see “Guided Hardware Setup”.

Model an Edge Detection Algorithm Using Neighborhood Processing Functions

Open the frame-based edge detection model.

```
open_system('hdlcoder_tunable_edge_detection')
set_param('hdlcoder_tunable_edge_detection', 'SimulationCommand', 'Update')
```



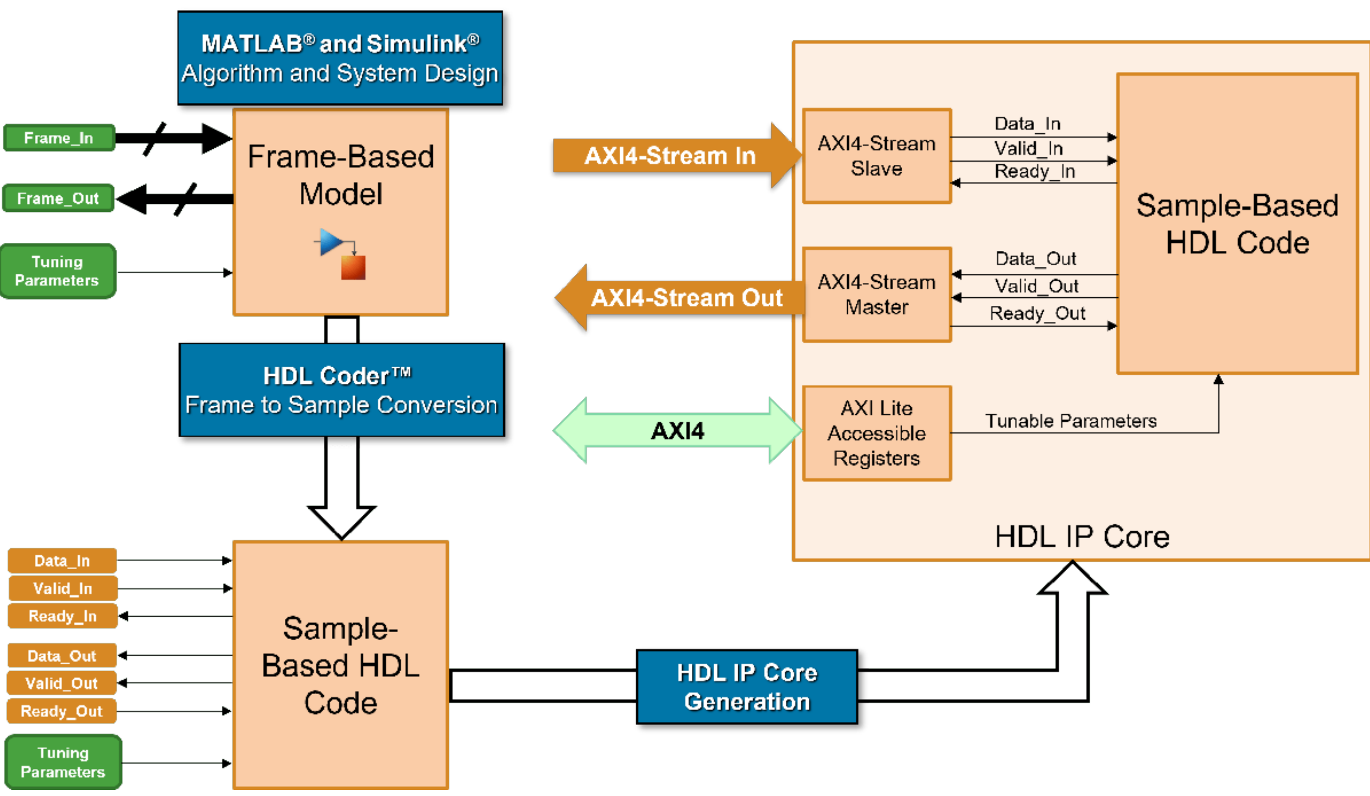
The model consists of the design under test (DUT) and the test bench. The DUT contains a MATLAB® Function block that models a Sobel edge detection algorithm by using the `hdl.npufun` function. The edge calculation uses scalar input parameters, `cThreshold` and `cAlpha`, with the frame input to compute the edge overlay on the original frame input. For more information, see `hdl.npufun`.

```
function [O,E]  = edgeDetectionAndOverlay(I, cThreshold, cAlpha)
```

```
E = hdl.npufun(@sobel_kernel, [3 3], I,'KernelArg', cThreshold);
O = hdl.npufun(@mix_kernel, [1 1], E, I,'KernelArg', cAlpha);

end

function e = sobel_kernel(in,cThreshold)

u = fi(in);
hGrad = u(1) + fi(2)*u(2) + u(3) - (u(7) + fi(2)*u(8) + u(9));
vGrad = u(1) + fi(2)*u(4) + u(7) - (u(3) + fi(2)*u(6) + u(9));

hGrad = bitshift(hGrad, -3); % Divide by 8
vGrad = bitshift(vGrad, -3); % Divide by 8

thresholdValueSq = fi(cThreshold); % Edge threshold
e = (hGrad*hGrad + vGrad*vGrad) > thresholdValueSq;

end

function O = mix_kernel(E, I, cAlpha)

alpha = fi(cAlpha); % Parameter for combining images
scaleE = E*fi(255,0,8,0);
O = scaleE * (fi(1)-alpha) + I*alpha;

end
```

You can use the `hdl.npufun` function for the frame-to-sample conversion to translate the frame-based algorithm into a sample-based algorithm. You can connect the streaming I/O of your sample-based algorithm to a streaming interface. Additionally, you can map the tunable parameters to AXI4-Lite or External Ports.

When you convert this model by using the HDL Coder Workflow Advisor, you generate an AXI4-Stream interface that contains the signals in this image:



Generate HDL IP Core

Generate an IP core from the frame-based DUT by using the HDL Workflow Advisor.

1. Enable the frame-to-sample conversion by entering this command:

```
hdlset_param('hdlcoder_tunable_edge_detection','FrameToSampleConversion','on');
```

2. Enable the HDL block property `ConvertToSamples` for the input image to be streamed (`imageIn`) using this command:

```
hdlset_param('hdlcoder_tunable_edge_detection/EdgeDUT/imageIn','ConvertToSamples','on');
```

3. Set up the Xilinx Vivado synthesis tool path by using the `hdlsetuptoolpath` command. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

4. Open the HDL Coder app. Click the **Apps** tab, then open **HDL Coder**. Click the **Workflow Advisor** button to open the HDL Workflow Advisor.

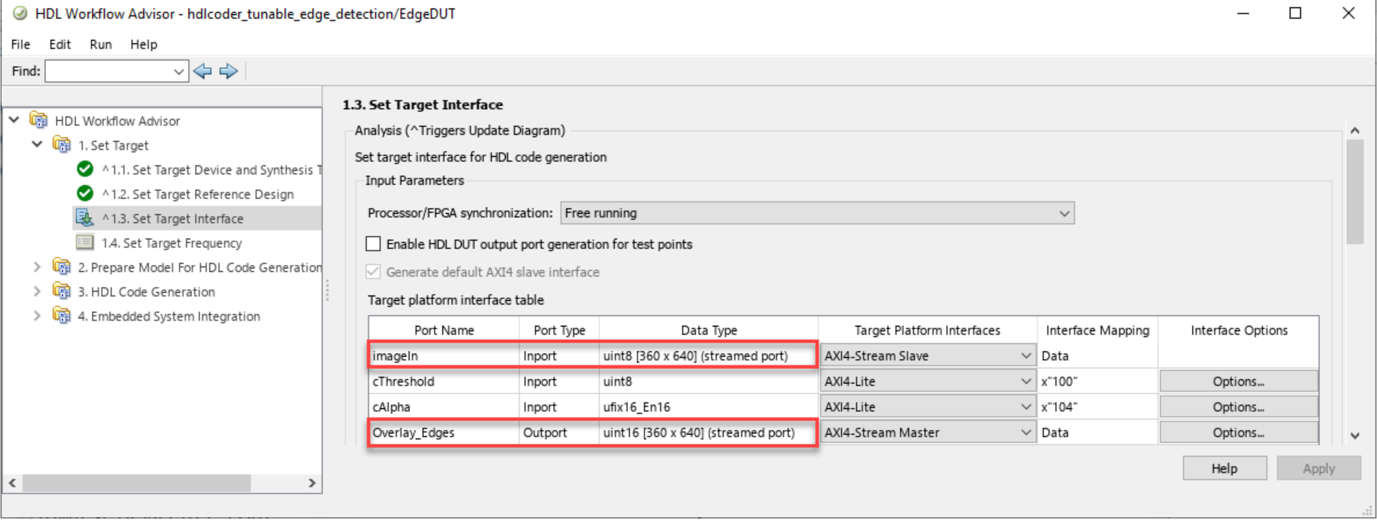
5. Click **1. Set Target > 1.1. Set Target device and Synthesis Tool**, then set **Target Workflow** to `IP Core Generation` and **Target Platform** to `Xilinx Zynq ZC706 evaluation kit`. If you are targeting a different Xilinx SoC, choose your board from the **Target Platform** context menu.

6. Click **1.2.Set Target Reference Design**, then set **Reference design** to `Default system with AXI4-Stream interface`.

7. Click **1.3.Set Target Interface**. The ports of the DUT subsystem are mapped to IP core interfaces. The AXI4-Stream interface communicates in master/slave mode, where the master device sends data

to the slave device. For this example, for the input port, **imageIn**, set **Target Platform Interfaces** to `AXI4-Stream Slave` and set the output port, **Overlay_Edges**, to `AXI4-Stream Master`. The cells in **Data Type** column that include `streamed port` indicate the ports you can map to stream interfaces. Finally, to control the tunable parameters at runtime, set **Target Platform Interfaces** to `AXI4-Lite` for the non-streamed kernel inputs, **cAlpha** and **cThreshold.**

Click **Run This Task**.



The AXI4-Stream interface contains data (**Data_In, Data_Out**) and control signals such as data valid (**Valid_In, Valid_Out**), back pressure (**Ready_In, Ready_Out**), and data boundary (**TLAST**). It is only required to map your frame inputs and outputs as **Data**. The **Valid**, **Ready**, and **TLAST** control signals are automatically generated. Refer to “Model Design for AXI4-Stream Interface Generation” on page 40-14 for a detailed description of the protocol signals.

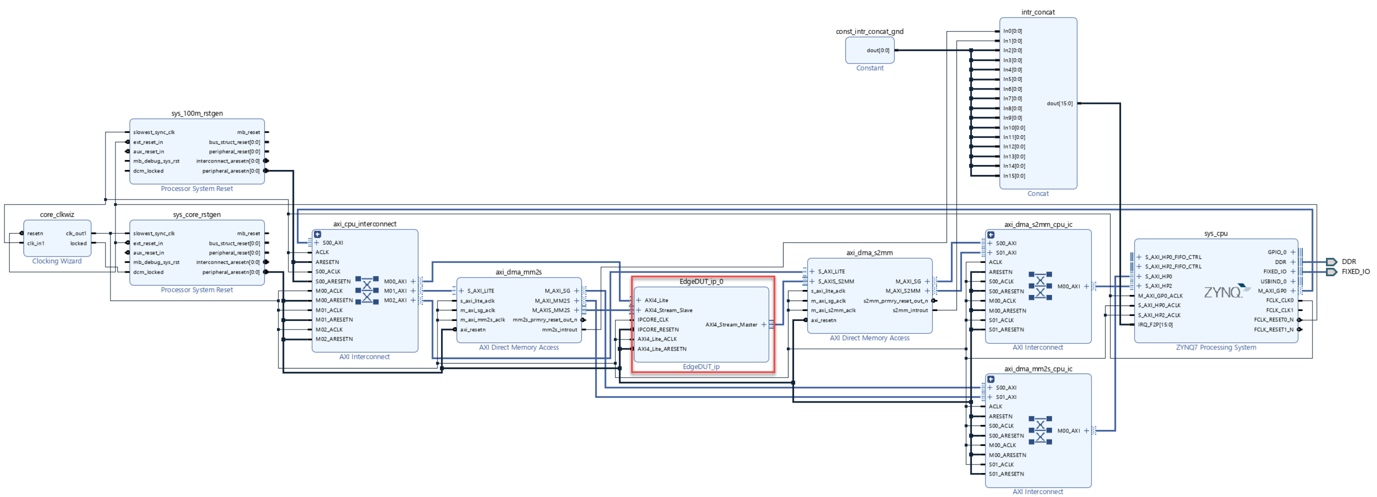
8. Right-click **3.2**, **Generate RTL Code and IP Core** and select **Run to Selected Task** to generate the IP core.

Integrate IP Into Reference Design Compatible with AXI4-Stream

In the HDL Workflow Advisor, run the tasks under **4.Embedded System Integration** to deploy the generated HDL IP core on Zynq hardware.

1. Click **4.1 Create Project**, then click **Run This Task**. This task inserts the generated IP core into the **Default system with AXI4-Stream interface** reference design. As shown in the IP core report, data flows from the ARM processing system through the DMA controller and the AXI4-Stream interface to the HDL Coder generated edge detection IP core. The processing system receives the processed output from the edge detection IP core.

2. Optionally, click the link in the Result pane to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram.



Generate Interface Between Host Computer and IP Core

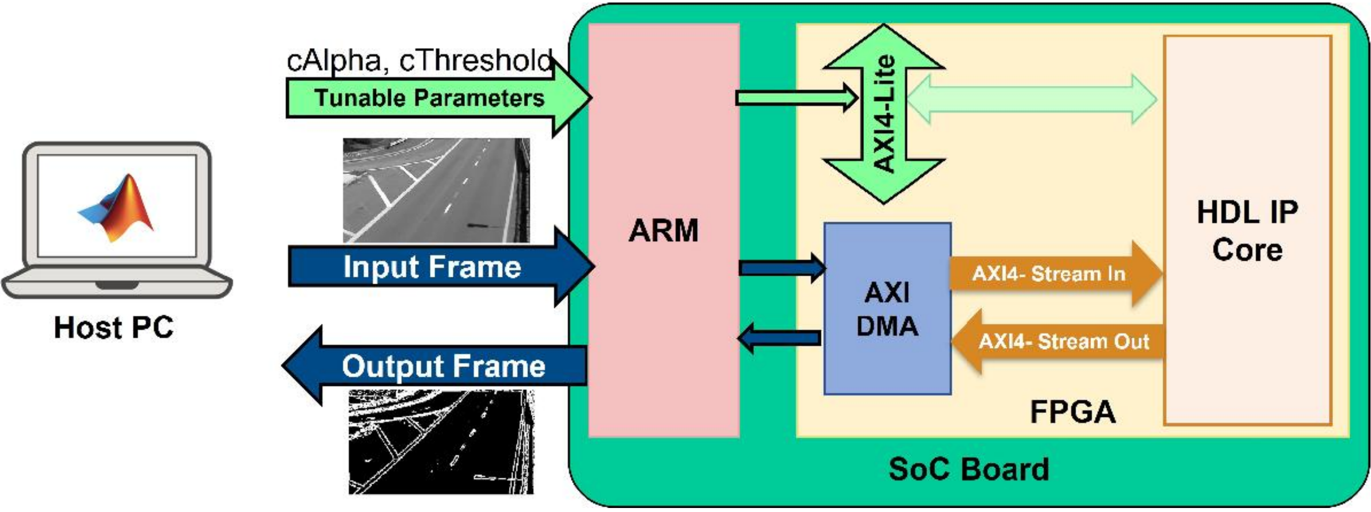
1. In the **Generate Software Interface**, select **Generate host interface script** and click **Run this Task**. The HDL Coder Workflow advisor generates two MATLAB files in your current folder that you can use to prototype your generated IP core directly from MATLAB:

- `gs_hdlcoder_tunable_edge_detection_setup.m` **:** This function configures the `fpga` hardware object with the same ports and interfaces that you mapped in the **1.3 Set Target Interface** task.
- `gs_hdlcoder_tunable_edge_detection_interface.m`**:** This file creates a connection to your FPGA hardware for reading and writing data.

2. In the HDL Workflow Advisor, run the rest of the tasks to build and download the FPGA bitstream.

Live Frame-Based Model Running on FPGA

You can interact with the FPGA design by reading and writing data from MATLAB on the host computer as described in the **Interact with FPGA Design from Host Computer** section of the “Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands” on page 39-241. The host computer sends and receive frames of data from the System on Chip (SoC) board as shown below in the high level architecture of the system:



This live script uses the generated script file as a starting point to test the frame-based model deployed on the FPGA.

```
open hdlcoder_frame_edge_detection_script.mlx
```

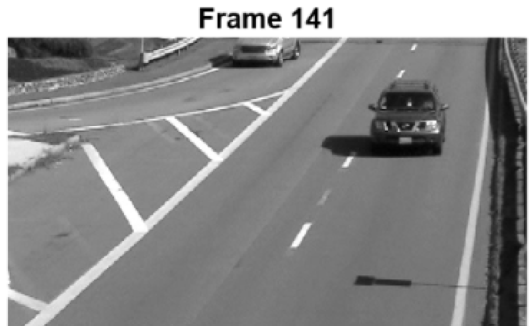
In the Write/Read DUT ports section of the `hdlcoder_frame_edge_detection_script` script, the code performs edge detection for every 20th frame for the first 200 frames of the `visiontraffic.avi` video and varies the cAlpha and cThreshold parameters as shown below

```
vidObj = VideoReader('visiontraffic.avi');
h = figure();
set(h, 'Position', [0 0 800 800])
for ii=1:20:200
    % Modify Parameters
    cThreshold = randi([20,90],1); % Vary cthreshold from 20 to 90
    % Write cThreshold value to "cThreshold" port in the DUT (AXI4-Lite)
    writePort(hFPGA, "cThreshold", cThreshold);
    cAlpha = rand(); % Vary cAlpha from 0 to 1
    % Write cAlpha value to "cAlpha" port in the DUT (AXI4-Lite)
    writePort(hFPGA, "cAlpha", cAlpha);
    % Read the iith frame of vidObj
    vidFrame = read(vidObj,ii);
    vidFrameGr = rgb2gray(vidFrame);
    % Write to inputImage DUT port (AXI4-Stream)
    wrValid = writePort(hFPGA, "imageIn", vidFrameGr);
    if wrValid
        subplot(2,2,1), imagesc(vidFrameGr); axis image; axis off; colormap(gray);
        title(sprintf('Frame %d',ii))
    end
    % Read from inputImage DUT port (AXI4-Stream)
    [outputFrame, rdValid] = readPort(hFPGA,"Overlay_Edges");
    if rdValid
        % Display the data read from the DUT
        subplot(2,2,2), imagesc(uint8(bitsliceget(fi(outputFrame),16,9))); axis image; axis off;
        title('Overlay')
        subplot(2,2,3), imagesc(uint8(bitsliceget(fi(outputFrame),8,1))); axis image; axis off;
        title('Edges')
        sgtitle(sprintf('threshold %0.2f, alpha %0.2f',cThreshold, cAlpha))
        drawnow
```

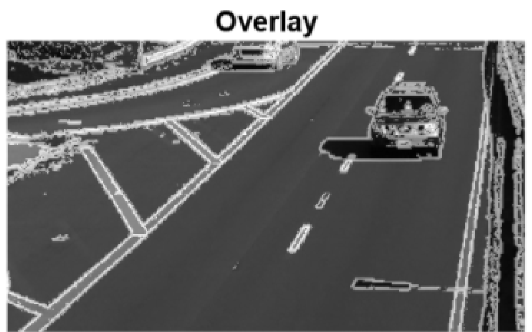
```
    else
        continue;
    end
end
```

threshold 49.00, alpha 0.54

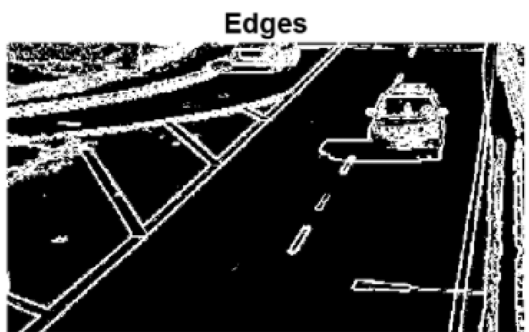
Frame 141



Overlay



Edges

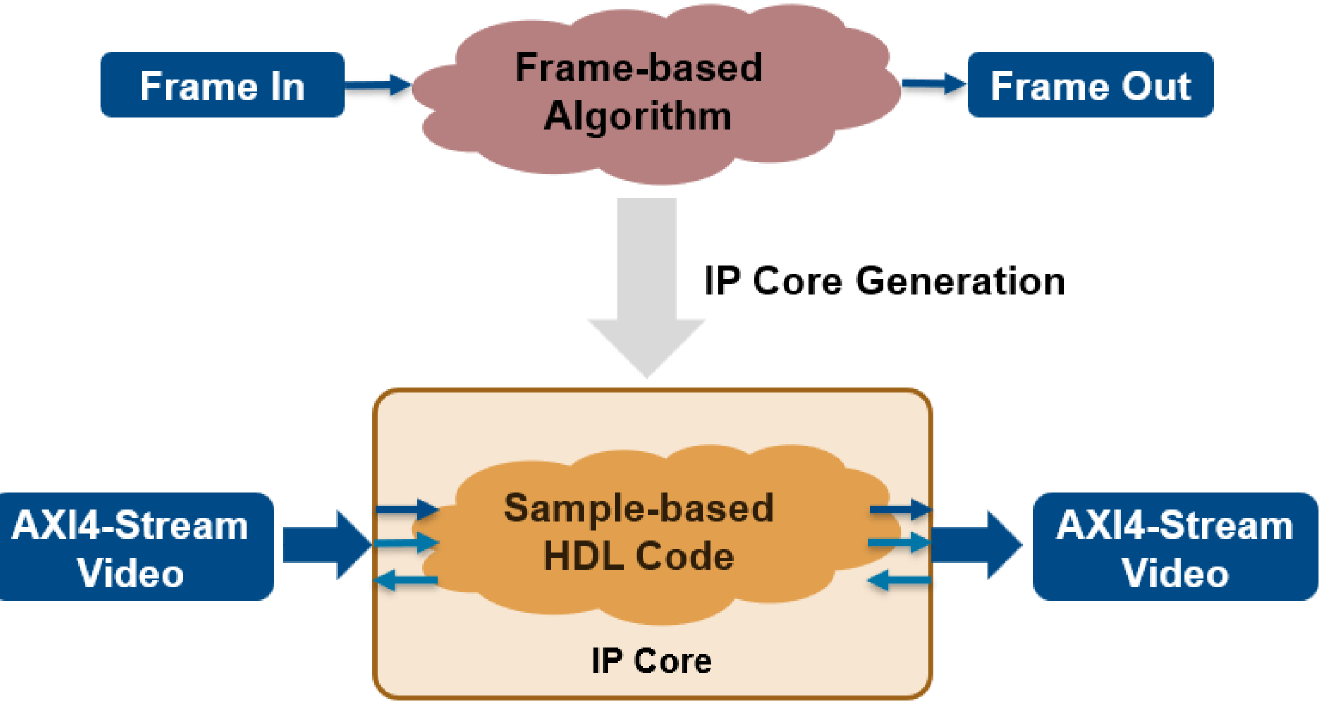


When you finish the example, run the last line of the script to release any hardware resources used by the `fpga` object:

```
release(hFPGA);
```

Deploy Frame-Based Models with AXI4-Stream Video Interfaces in Zynq-Based Hardware

You can use the HDL Coder™ frame-to-sample optimization to design and deploy frame-based algorithms that use AXI4-Stream Video interfaces to enable high-speed video streaming.



In this example, you:

1. Model a frame-based edge detection algorithm by using neighborhood processing functions.
2. Generate an HDL IP core with an AXI4-Stream Video interface.
3. Integrate the generated IP core into a Xilinx® Zynq® ZC706 reference design with access to HDMI interfaces.

Prerequisites

To run this example, you must have the following software and hardware board setup:

- Xilinx Vivado® Design Suite
- Xilinx Zynq ZC706 evaluation kit with FMC HDMI I/O card (FMC-HDMI-CAM or FMC-IMAGEON)

Set Up Hardware

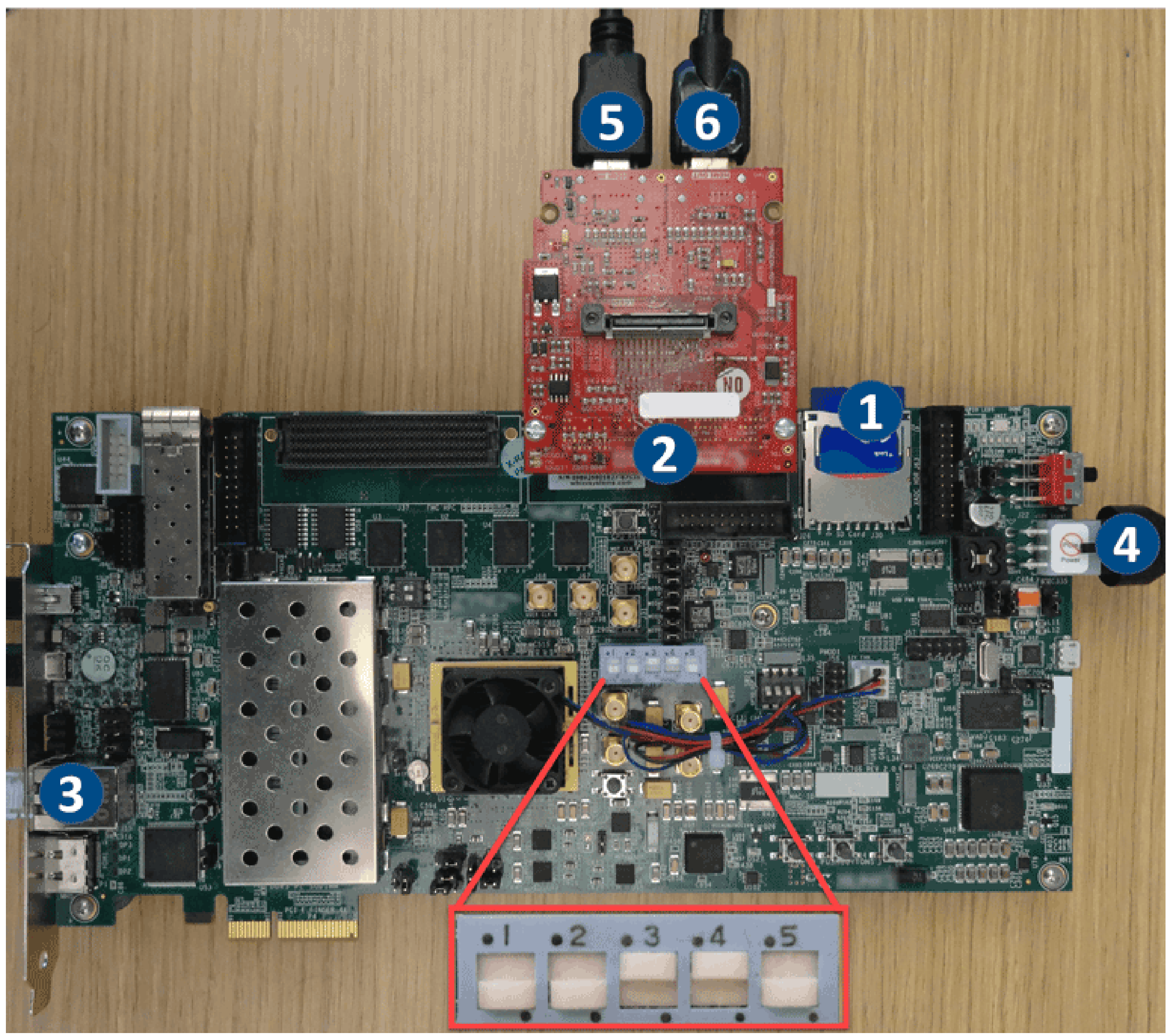
1.Install the HDL Coder Support Package for Xilinx FPGA and SoC Devices, and SoC Blockset™ Support Package for Xilinx Devices. To install the support packages, in MATLAB®, in the **Home** tab, click **Add-Ons > Get Hardware Support Packages**.

2. Use the SD card image provided by the HDL Coder Support Package for Xilinx FPGA and SoC Devices.

3. Set up the Xilinx Zynq ZC706 board and the FMC HDMI I/O card as shown in the figure below. Follow these instructions for connecting the hardware:

1. Remove the SD card from the host computer and insert it into the Zynq board.
2. Plug the HDMI FMC card into the FMC connector on the Zynq board.
3. Connect an Ethernet cable to the board.
4. Connect the power cable.
5. Connect an HDMI video source to the FMC HDMI I/O card. The video source must be able to provide 1080p video output, such as a video camera, smart phone, tablet, or the HDMI output of your computer.
6. Connect a monitor to the FMC HDMI I/O card. The monitor must be able to support 1080p display.

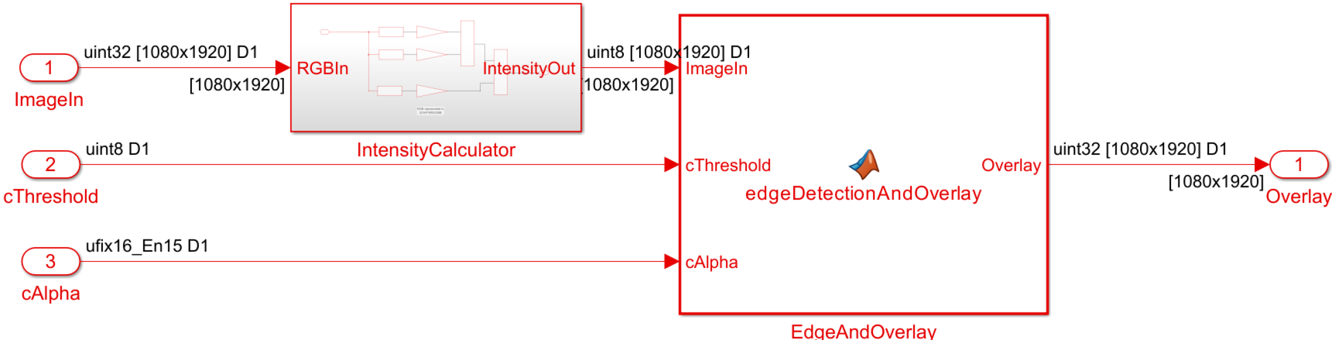
To learn more about the ZC706 hardware setup, see Xilinx ZC706 Evaluation Board User Guide.



Model a Frame-Based Algorithm to Integrate with the Default Video System Reference Design

Open the frame-based model, `hdlcoder_frame_video_edge`,which contains an edge detection and image overlay algorithm in the device under test (DUT), `EdgeDUT`.

```
load_system('hdlcoder_frame_video_edge')
open_system('hdlcoder_frame_video_edge/EdgeDUT')
```



In the example model, the DUT subsystem, `hdlcoder_frame_video_edge/EdgeDUT`, uses the `hdl.npufun` function to implement the Sobel edge detection algorithm and the `code.hdl.pipeline` function to pipeline the design to achieve the frequency needed to perform edge detection on live video. For more information on modeling the algorithm, see "Synthesize Code for Frame-Based Model" on page 22-26. When using the frame-to-sample optimization, you do not need to model additional control signals. The ports `ImageIn` and `Overlay` are the pixel data ports for video streams. The additional input ports, `cThreshold` and `cAlpha`, are control ports that adjust the parameters of the Sobel edge detection algorithm. You can enable the frame-to-sample conversion for this model using these commands:

```
hdlset_param('hdlcoder_frame_video_edge','FrameToSampleConversion','on');
hdlset_param('hdlcoder_frame_video_edge/EdgeDUT/ImageIn','ConvertToSamples','on');
```

Generate HDL IP Core with AXI4-Stream Video Interface

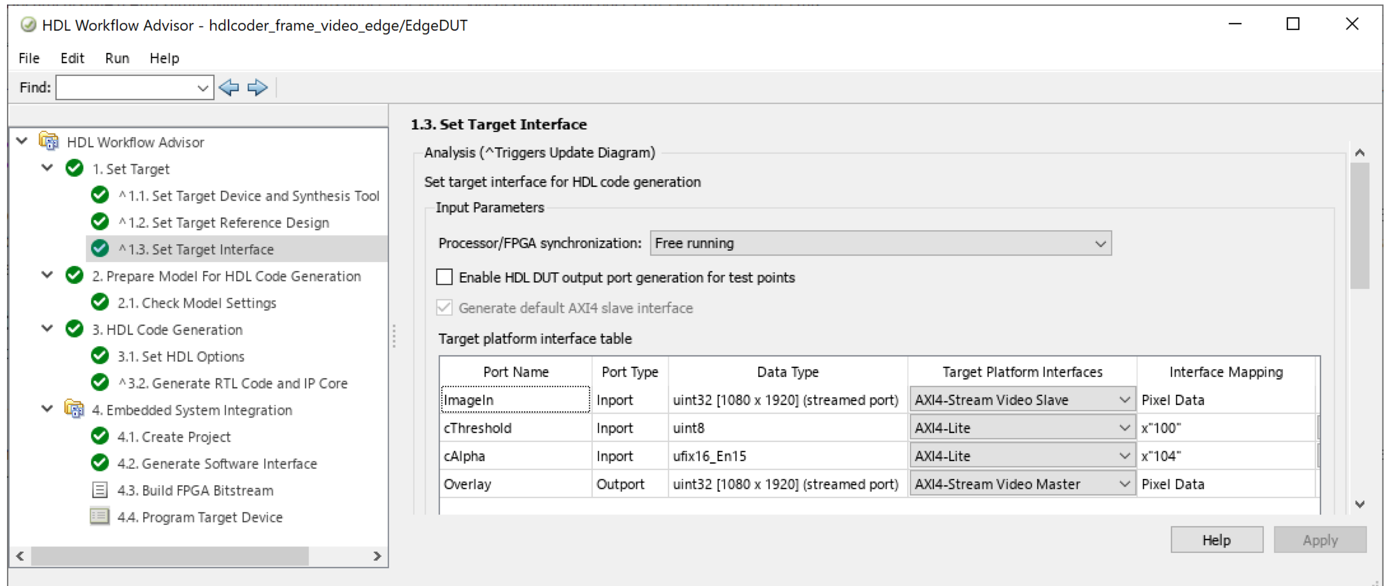
When you generate an IP core that is integrated into the `Default Video System Reference Design`, the video stream signals are encoded into a 32-bit representation 32'hFFRRGGBB. To account for the data format, the subsystem `hdlcoder_frame_video_edge/EdgeDUT/IntensityCalculator` decodes the RGB 32-bit representation of the video stream to get the intensity of the frame. Additionally, the subsystem encodes the output into the same format when it calculates the overlay. For more information on the architecture of the reference design see "Default Video System Reference Design".

To generate an IP core with the AXI4-Stream Video interface from a frame-based model:

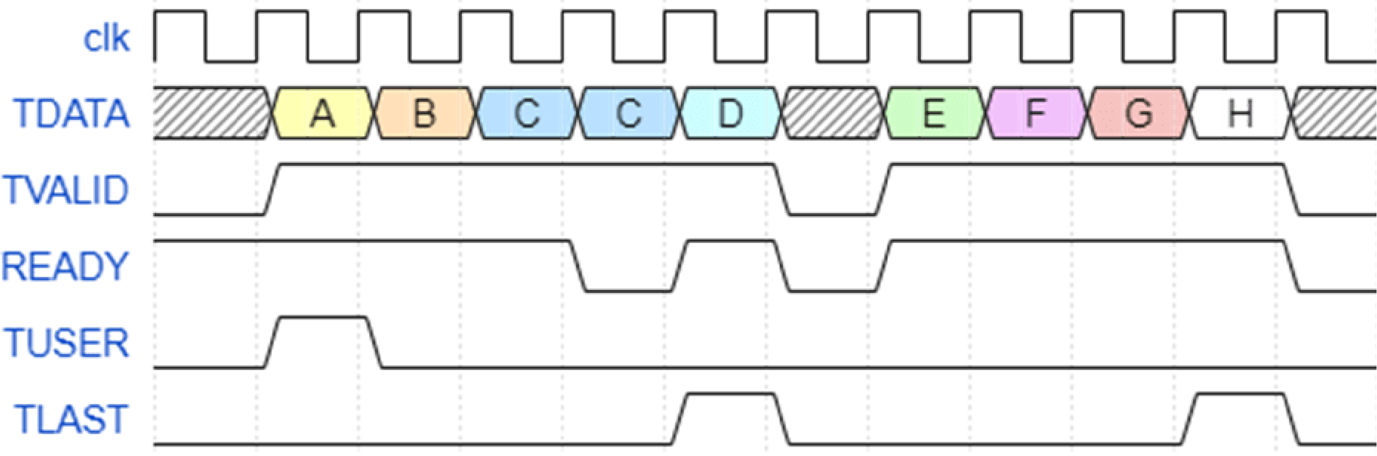
1. Set up the Xilinx Vivado synthesis tool path by using the `hdlsetuptoolpath` command. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2022.1\bin\vivado.bat')
```

2. In the **1.1 Set Target Device and Synthesis Tool Task,** set **Target workflow** to `IP Core Generation` and set **Target platform** to `Xilinx Zynq ZC706 evaluation kit`. In task 1.2, set the **Reference Design** to `Default video system (requires HDMI FMC module)`. In task 1.3, **Target Platform Interface Table** section, set the cells to these settings:



The AXI4-Stream video interface contains data (`TDATA`) and control signals such as TVALID, back pressure (`READY`), and data boundary (`TDATA` and `TLAST`). `TDATA` indicates the start of a video frame and `TLAST` indicates the end of a line. The timing diagram shows the timing for a 2x4 frame.



To use the frame-to-sample optimization, you must map your frame inputs and outputs as `Pixel Data`. The `TVALID`, `READY`, and `TLAST` control signals are automatically generated. In this example, the frame ports `ImageIn` and `Overlay` are mapped to the AXI4-Stream Video interface. The scalar ports `cAlpha` and `cThreshold` are mapped to the AXI4-Lite interface. HDL Coder generates AXI4 interface accessible registers for these ports.

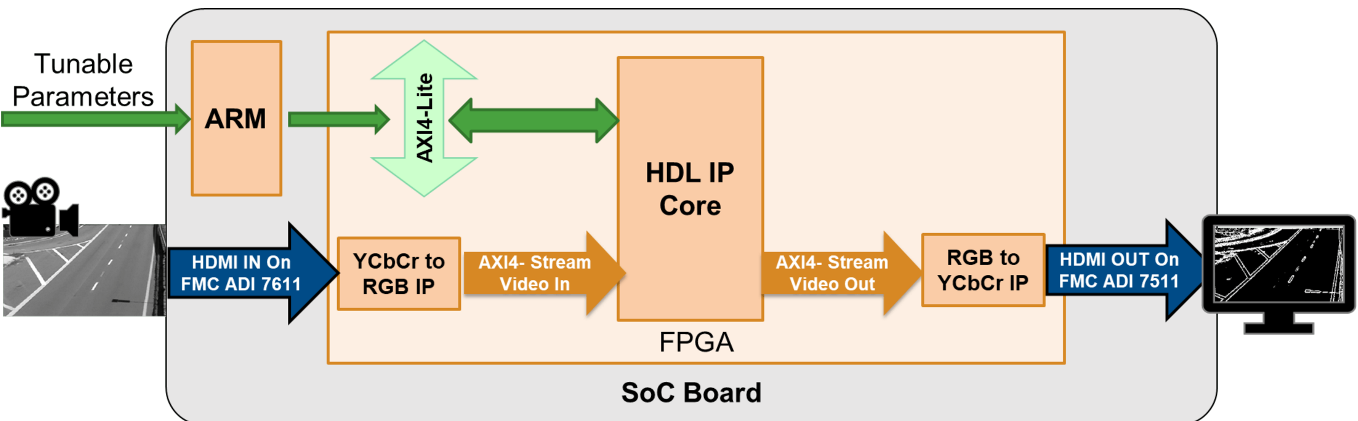
3. Right-click task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other information for the IP core in the generated IP Core Report.

4. Right-click task 4.2 **Build FPGA Bitstream**, and select **Run to Selected Task** to generate the Vivado project and build the FPGA bitstream.

5. Run task 4.3, **Program Target Device,** to program the FPGA board.

System Architecture with the Generated IP Core

The high-level architecture of the complete Zynq design is shown below.



The generated DUT IP core is integrated into the `Default Video System` reference design. The AXI4-Stream Video interface processes the video stream from the HDMI input and sends the output video stream to the HDMI output. The host computer can tune parameters by writing to AXI4-Lite registers within the algorithm IP core.

See Also

Related Examples

- “Model Design for AXI4-Stream Video Interface Generation” on page 40-118

More About

- “HDL Code Generation from Frame-Based Algorithms” on page 22-2

DAC and ADC Loopback Data Capture

This example shows how to capture raw analog-to-digital converter (ADC) data using the FPGA I/O API from the Xilinx® Zynq® UltraScale+™; ZCU111 evaluation kit or Xilinx Zynq UltraScale+ ZCU216 evaluation kit. This example uses the HDL Workflow Advisor to configure an SoC model for HDL code generation. In this example, you generate the HDL code for your algorithm, build and deploy the HDL design on an RFSoC device, and run a MATLAB® script to interactively capture data from the deployed HDL design.

Requirements

- Vivado® Design Suite with a supported version listed in “Supported EDA Tools and Hardware”
- Xilinx Zynq UltraScale+ ZCU111 evaluation kit or Xilinx Zynq UltraScale+ ZCU216 evaluation kit
- HDL Coder™
- HDL Coder™ Support Package for Xilinx® FPGA and SoC Devices
- To set up the board connection for DAC and ADC loopback data capture, follow the hardware configuration settings for RFSoC boards in “Board-Specific Setup Information”.

Introduction

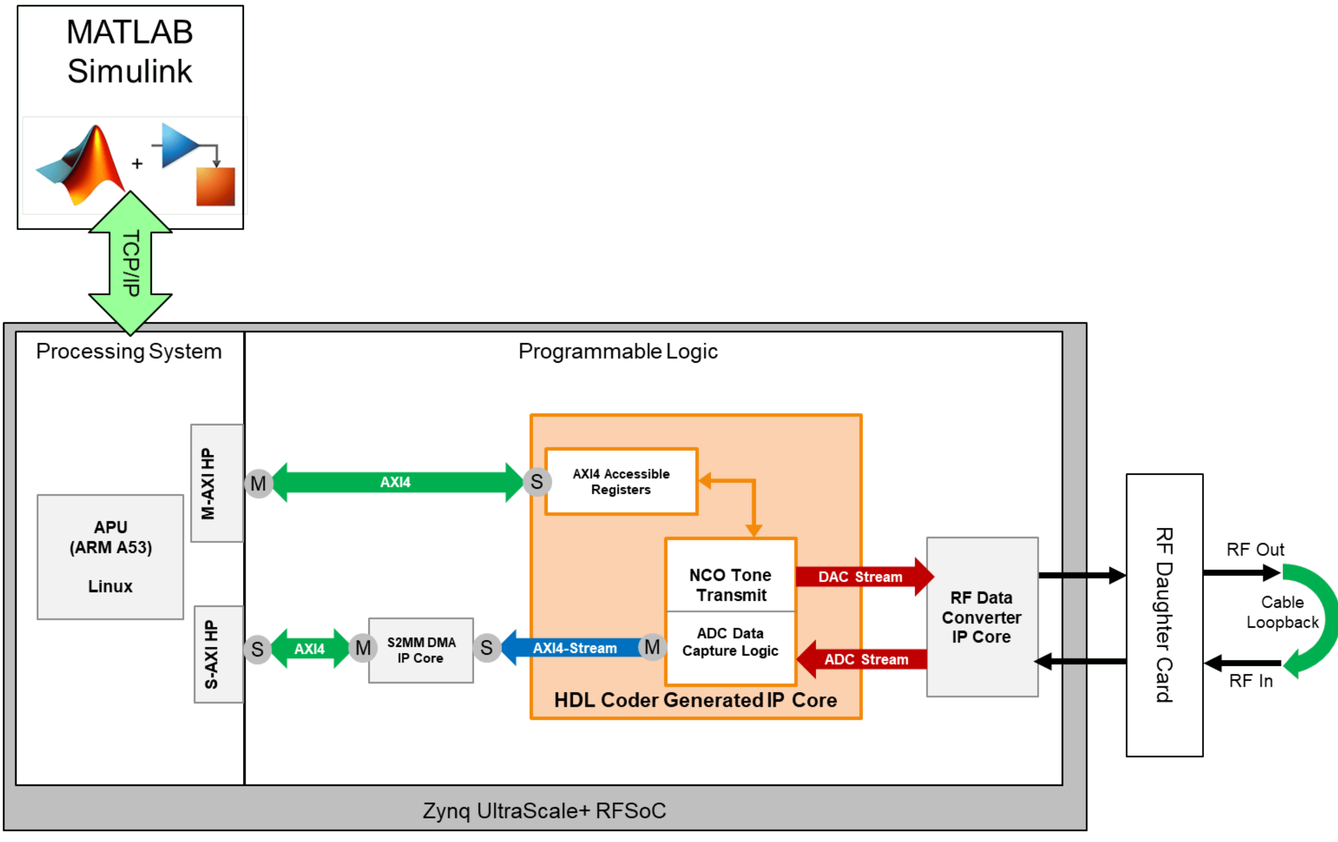
Open the example project and copy the example files to a temporary directory

1. Navigate to the RFSoC root example directory of HDL Coder™ Support Package for Xilinx® FPGA and SoC Devices by entering these commands at the MATLAB command prompt.

```
example_root = (hdlcoder_rfsoc_examples_root)
cd (example_root)
```

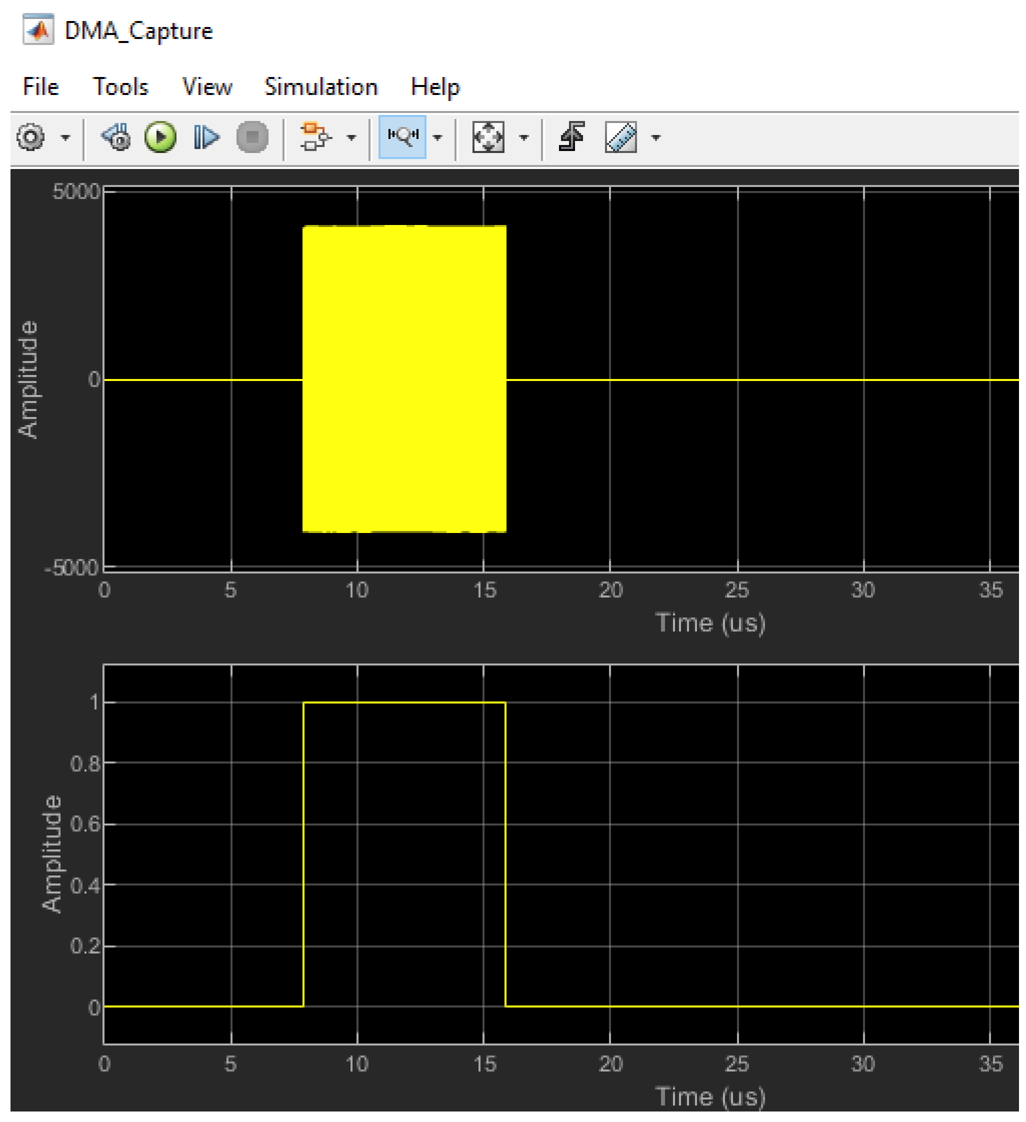
2. Copy all of the example files in the `ADCDataCapture` folder to a temporary directory.

This diagram depicts how the HDL design is used for this ADC capture example. The HDL Coder IP design transmits a numerically-controlled oscillator (NCO) waveform tone out of the digital-to-analog converter (DAC), which is then subsequently received by the ADC in the loopback configuration. AXI4-Stream transfers facilitate data movement back to the ARM processor in the Zynq RFSoC, which is then sent back to MATLAB over a TCP/IP connection.

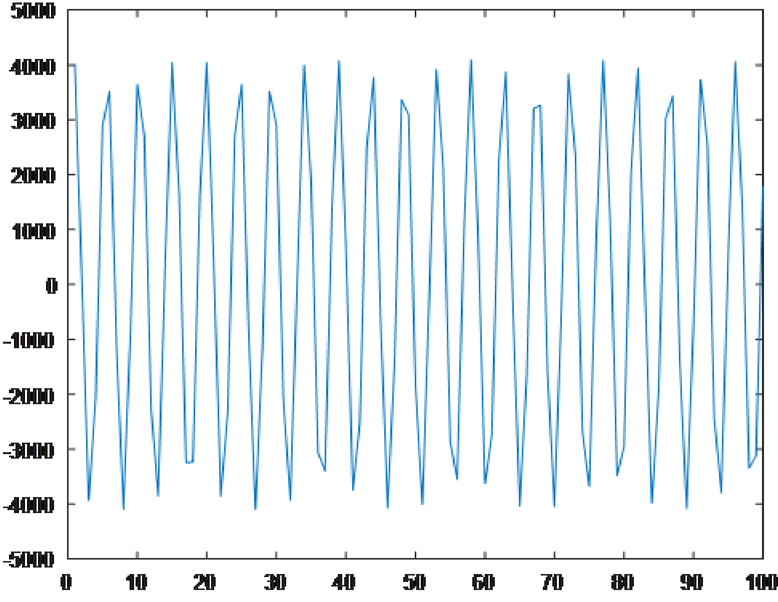


Capture Logic Simulation

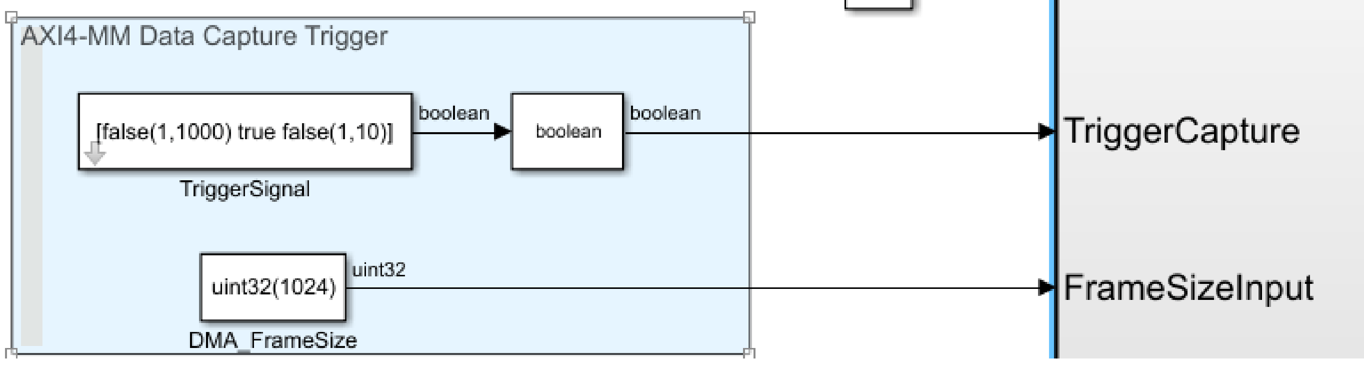
Simulate the model until the simulation terminates at the stop time. To examine the simulated captured data of the NCO waveform, open the DMA_Capture time scope plot, which this figure shows.



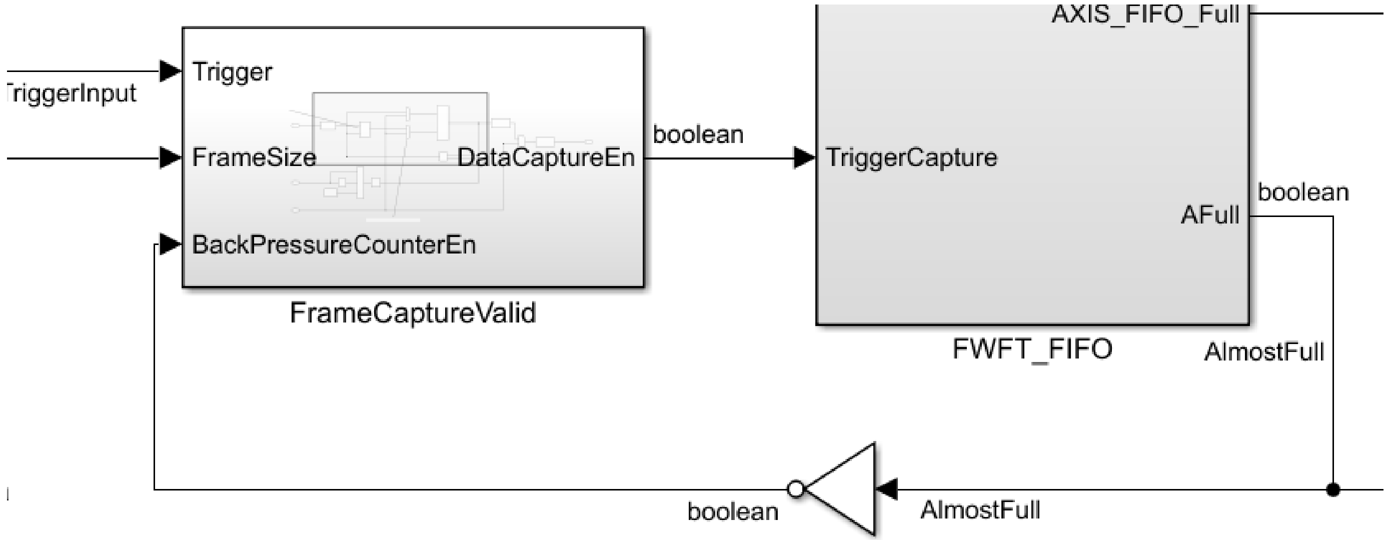
You can also view the captured data in MATLAB. The script `rfsocADCCaptureSimulationPlot.m` displays the data but uses the valid signal to extract the sample. This trimmed-down plot is what the direct memory access (DMA) transfers.



The simulation is set up such that the data movement logic moves 1024 samples of ADC samples when the trigger input is rising edge. This movement is done by using these two AXI4 register inputs.



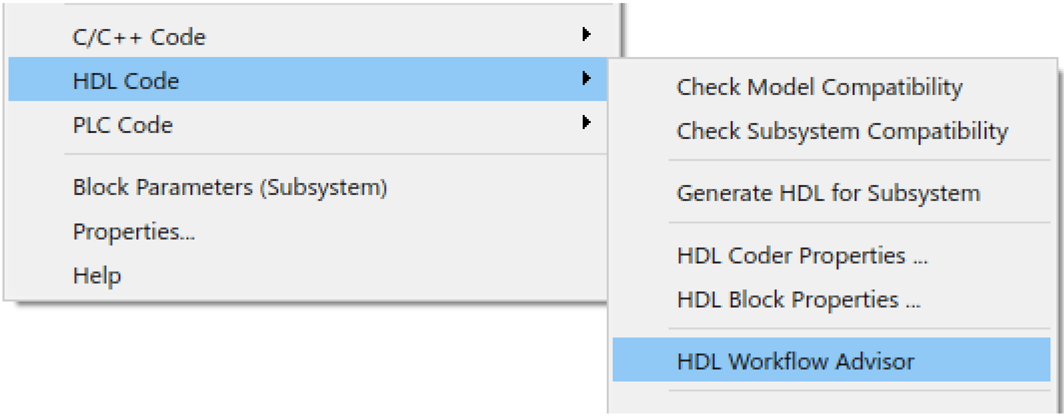
The register for the DMA frame size and the amount of data to transfer in the DMA must be equal. To examine the counter logic on how the capture is done, see this subsystem: `rfsocADCCapture/ADC_Data_Capture/FrameCaptureValid`.



Generate HDL and Synthesize Bitstream

If you are running the example on a Windows® platform, copy and paste the example into a new folder with a shorter path before running this example. There are path character-length limitations that cause Vivado to error. This issue is not present on Linux® platforms.

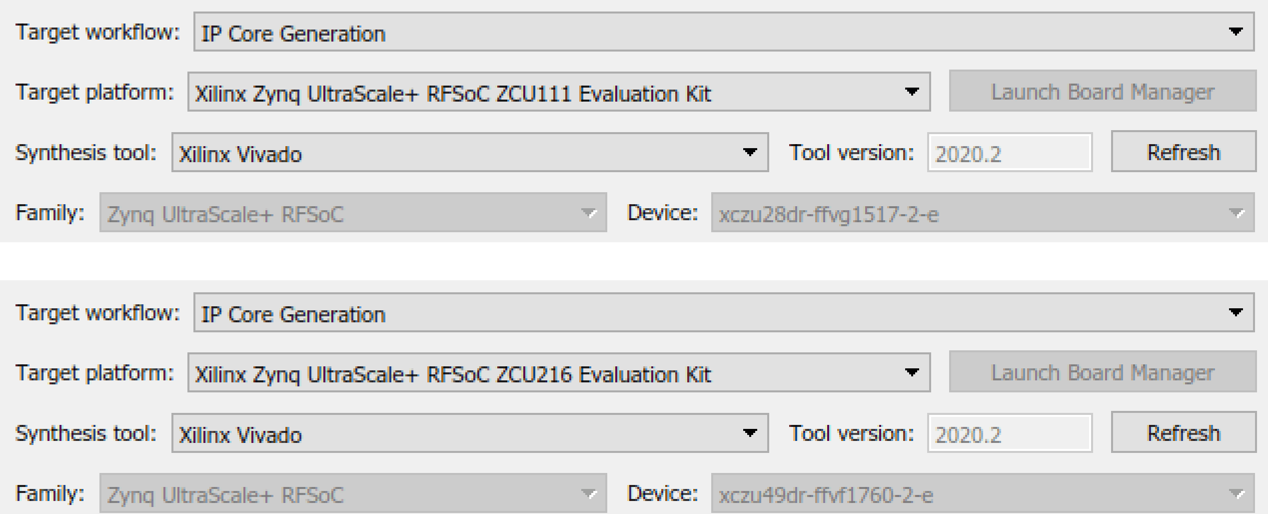
Open the model `rfsocADCCapture.slx`, and then right-click the `ADC_Data_Capture` subsystem. Select **HDL Code**, then click **HDL Workflow Advisor**.



Follow these steps in the HDL Workflow Advisor.

Step 1: Select Target Board

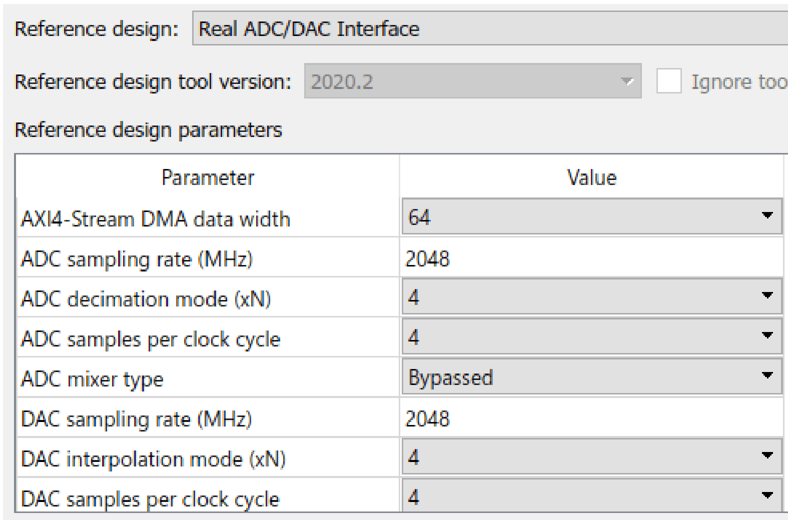
In step 1.1, select **Target platform** as `Xilinx Zynq Ultrascale+ RFSoC ZCU111 Evaluation Kit` or `Xilinx Zynq Ultrascale+ RFSoC ZCU216 Evaluation Kit`.



Right-click the design under test (DUT) subsystem, and then click **Run This Task**.

Step 2: Set Target Reference Design

In step 1.2, set the **Reference design** parameter and the parameters in the **Reference design parameters** section to the indicated values in this figure.



These settings give an effective clock rate of 128 MHz for the HDL Coder generated IP core. This value is the result of using a decimation of 4x and four samples per clock at 2.048 giga samples per second (GSPS). This table provides the reference design parameters for the ZCU111 and ZCU216 boards. Before proceeding to the next step, set these reference design parameters to the indicated values.

- **AXI4-Stream DMA data width** to 64
- **ADC sampling rate (MHz)** to 2048
- **ADC decimation mode (xN)** to 4
- **ADC samples per clock cycle** to 4

- **ADC mixer type** to `Bypassed`
- **DAC sampling rate (MHz)** to `2048`
- **DAC interpolation mode (xN)** to `4`
- **DAC samples per clock cycle** to `4`
- **DAC mixer type** to `Bypassed`
- **ADC/DAC NCO mixer LO (GHz)** to `Disabled`
- **Enable multi-tile sync** to `false`

If you are using a ZCU216 board, additionally set the **DAC DUC mode** parameter to `Full DUC Nyquist (0-Fs/2)`.

Click **Run This Task**.

Step 3: Set Target Interface

The target interface settings are saved in the example model. In step 1.3, these target interface settings load automatically after you successfully run the **Set Target Reference Design** task.

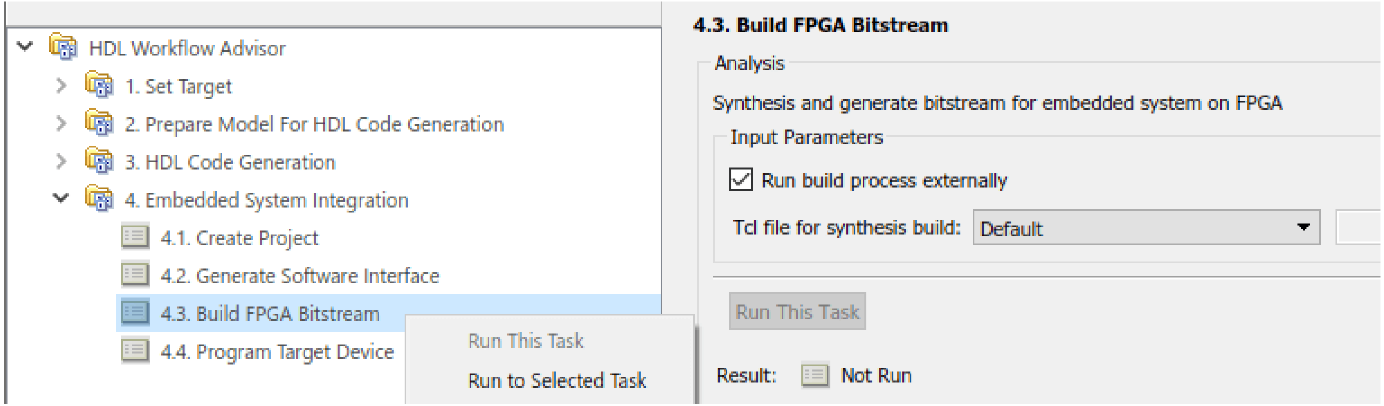
Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping
MM2S_Valid	Inport	boolean	AXI4-Stream DMA Slave	Valid
MM2S_Data	Inport	uint64	AXI4-Stream DMA Slave	Data
S2MM_Tready	Inport	boolean	AXI4-Stream DMA Master	Ready (optional)
Tile0 ADC Ch0 Data	Inport	int16 (4)	Tile0 ADC Ch0 Data [0:63]	[0:63]
Tile0 ADC Ch0 Valid	Inport	boolean	Tile0 ADC Ch0 Valid	[0]
TriggerCapture	Inport	boolean	AXI4	x"100"
FrameSizeInput	Inport	uint32	AXI4	x"104"
NCO_incr_AXI	Inport	uint16	AXI4	x"114"
NCO_DAC_Ch0_AXI	Inport	ufix16_En...	AXI4	x"11C"
S2MM_Data	Outport	uint64	AXI4-Stream DMA Master	Data
S2MM_Valid	Outport	boolean	AXI4-Stream DMA Master	Valid
MM2S_Ready	Outport	boolean	AXI4-Stream DMA Slave	Ready (optional)

The port interface table designates the Simulink® subsystem input and outputs to an IP interface type. Options include AXI4-Stream DMA interfaces, ADC/DAC connections, and AXI4 slave registers. For more information on port interfacing, see “Set Target Interface” on page 36-6.

Step 4: Build FPGA Bitstream

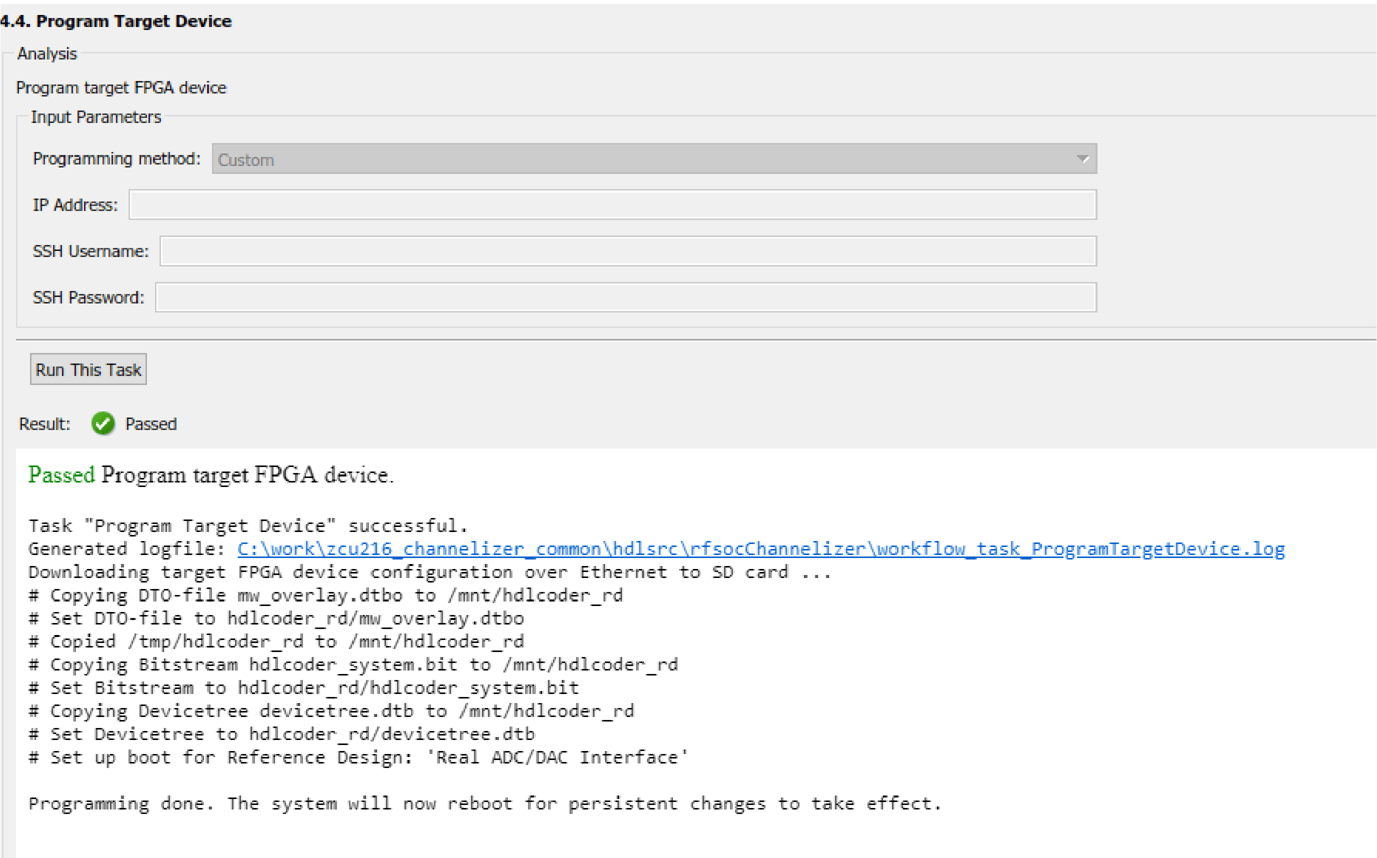
To generate a Vivado project for synthesis and bitstream creation, see “Build FPGA Bitstream” on page 36-22. A command prompt window opens to display synthesis and place and route progress. Bitstream generation takes about 20 to 30 minutes to build depending on the specifications of your computer.



In step 4.2, to generate a script that provides MATLAB connectivity to the board for interactive testing and live I/O interfacing, select **Generate host interface script**.

Program FPGA Bitstream

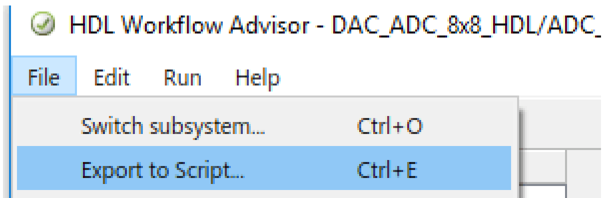
When the Vivado terminal reports that the FPGA bitstream has finished building, you can program the FPGA bitstream. Prior to programming the FPGA bitstream, you must ensure that the board has the correct SD card and is powered on. After Vivado programs the bit file, you see the message, which this figure shows.



The board restarts and programs the bit file to the FPGA during initialization. On the next power or boot cycle, the board continues using this bitstream repeatedly until it is reprogrammed with a new bit file or until the SD card is reformatted. You do not need to repeatedly program the same bit file because it remains persistent until a new one needs to be loaded.

If you need to swap bitstreams for different designs, programmatically script this swap out rather than using the HDL Workflow Advisor again, which can be time consuming. You can use a script that can be generated in the workflow advisor.

Select **File**, then click **Export to Script**.



After the script is generated, edit the script to download the bitstream by setting the appropriate options to `false`, as this figure shows.

```
290       % Set Workflow tasks to run
291 -     hWC.RunTaskGenerateRTLCodeAndIPCore = false;
292 -     hWC.RunTaskCreateProject = false;
293 -     hWC.RunTaskGenerateSoftwareInterfaceModel = false;
294 -     hWC.RunTaskBuildFPGABitstream = false;
295 -     hWC.RunTaskProgramTargetDevice = true;
```

Set the last task, `RunTaskProgramTargetDevice`, to `true`.

Now, run this script to program the FPGA. If successful, the MATLAB output window displays a message indicating success.

You can reuse the preceding script for generating a Vivado project again. If you make a change in the Simulink HDL design, you must recompile the Vivado design.

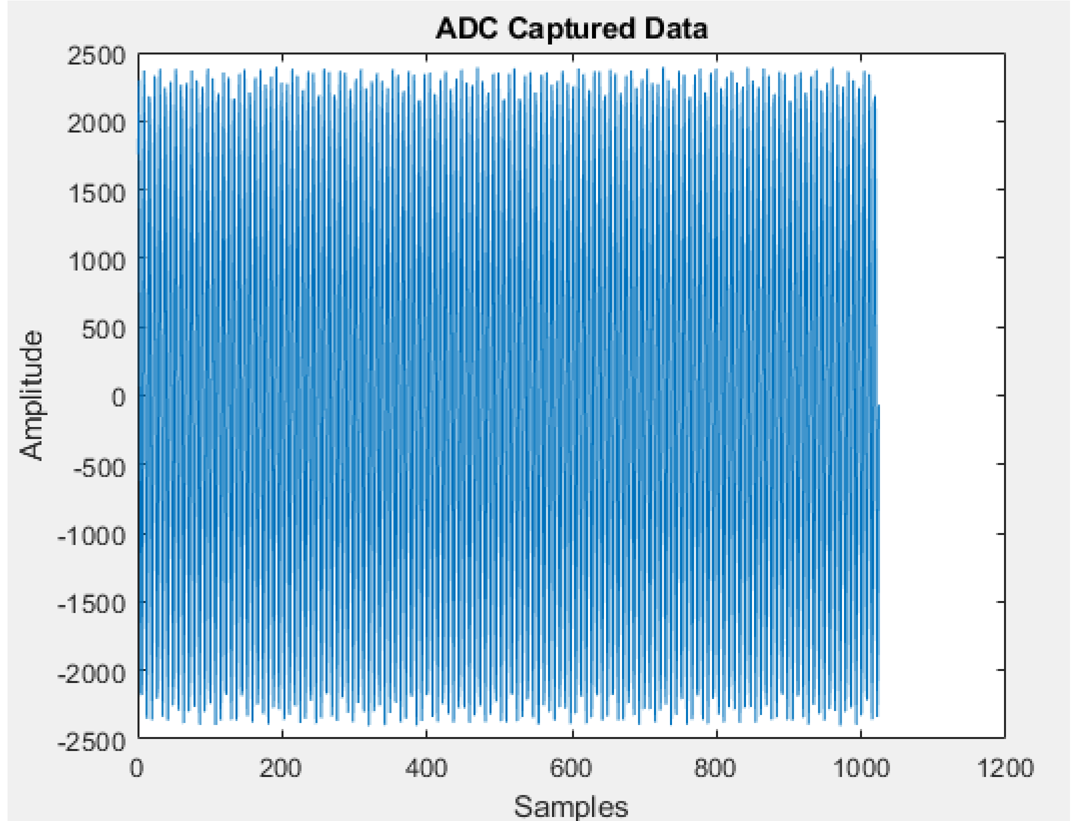
ADC Data Capture

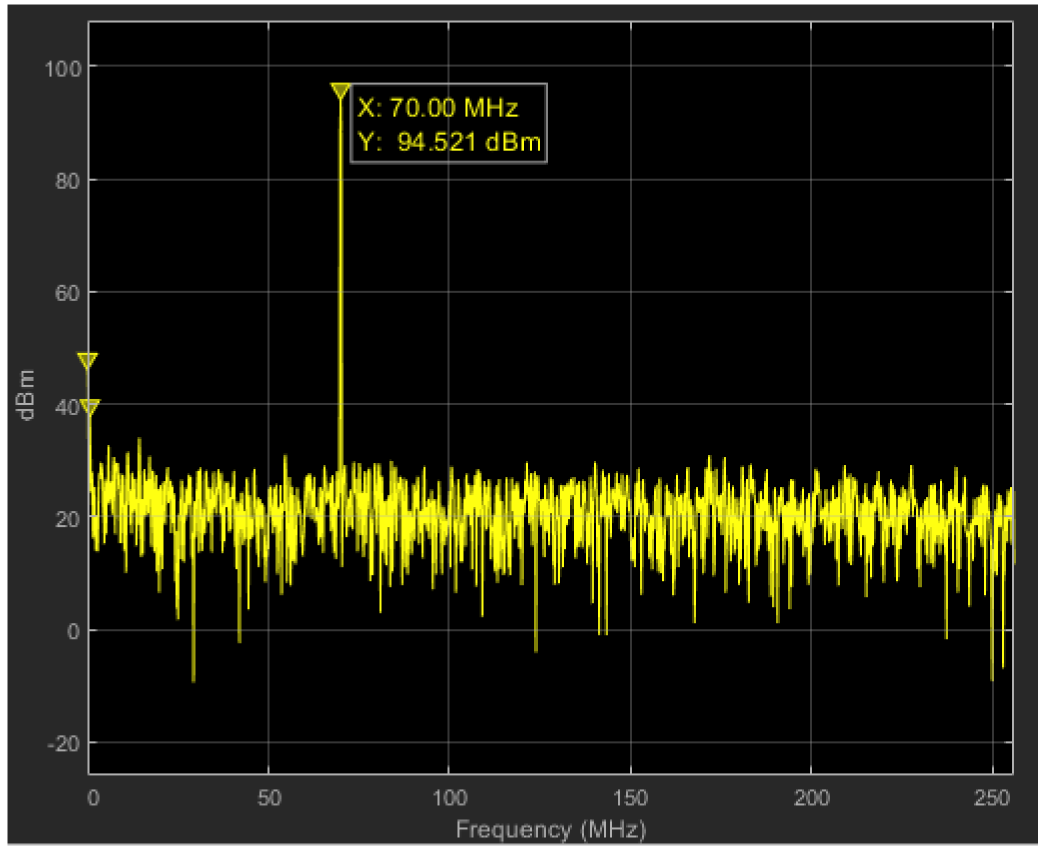
The HDL Coder Workflow Advisor generates scripts that you can use to help with communicating to the IP design deployed to the RFSoC at run time. These scripts include AXI4-Registers and AXI4-Streaming interfaces that are specified in the previous section in the workflow advisor. For more information, see the `fpga` object.

In this example, only the stream to memory-map (S2MM) interface is used to move data from the FPGA to the ARM processor by using the DMA. Several AXI4 registers are used to control the NCO tone, which transmits to the DAC.

The generated scripts are meant to be templates, which you can then customize to fit the intended application. To perform the data captures, this example provides MATLAB script files `HostIO_rfsocADCCapture_setup.m` and `HostIO_rfsocADCCapture_interface.m`. When the board is powered on after programming, the FPGA design runs the capture script. This script displays the ADC captured data plots.

```
HostIO_rfsocADCCapture_interface
```





These time domain and spectrum plots show the resulting data captures. The script captures data in each iteration of a `while` loop while also changing the location of the NCO tone. To terminate the capture of data, close the ADC Captured Data figure. A register that is triggered by the software captures the data, and the rising edge detect logic that initiates data capture counters to move data into the AXI4-Stream S2MM.

Change Converter Settings

The RFSoC data converter contains settings and configuration options, some of which you can modify at run time. When the board is running and Linux starts, default settings are applied to the RF data converters that match the original design from the HDL Coder Workflow Advisor. This matching is accomplished with a CFG file that is uploaded to the board, and then read in during boot initialization.

To accommodate the modification of these parameters at run time, this example provides a programmatic MATLAB approach. Create the `soc.RFDataConverter` System object™ by using these commands in MATLAB.

```
rfobj = soc.RFDataConverter('ZU28DR','192.168.1.101'); % For ZCU111 board
```

```
rfobj = soc.RFDataConverter('ZU49DR','192.168.1.101'); % For ZCU216 board
```

This object requires the device and IP address as input arguments. Currently, RF data converter object supports these devices: ZU28DR (ZCU111) and ZU49DR (ZCU216). After creation, `rfobj` is an object in the MATLAB workspace. You can access this object further by using various object functions or properties to alter device settings.

This example provides a convenience script that gives an example of how to change the RF converter parameters. This script is autogenerated during the HDL Workflow Advisor workflow. In this ADC capture example, see the MATLAB script `ADC_Data_Capture_setup_rfsoc.m`.

```
%% Instantiate object and basic settings
IPAddr = '192.168.1.101';
rfobj = soc.RFDataConverter('ZU49DR',IPAddr);



PLLSrc = 'Internal';
ReferenceClock = 245.76; % MHz
ADCSamplingRate = 2048; % MHz
DACSamplingRate = 2048; % MHz
DecimationFactor = 4;
InterpolationFactor = 4;


%% User FPGA-logic settings
rfobj.FPGASamplesPerClock = 4;
rfobj.ConverterClockRatio = 1;

% Check if FPGA clock-rate exceeds timing used during synthesis
FPGAClockRate = ADCSamplingRate/DecimationFactor/rfobj.FPGASamplesPerC
if FPGAClockRate > 128
    warning('Selected FPGA rate %3.3f MHz exceeds the timing that was
            FPGAClockRate,128);
end

%% Establish TCP/IP connection
setup(rfobj)
```

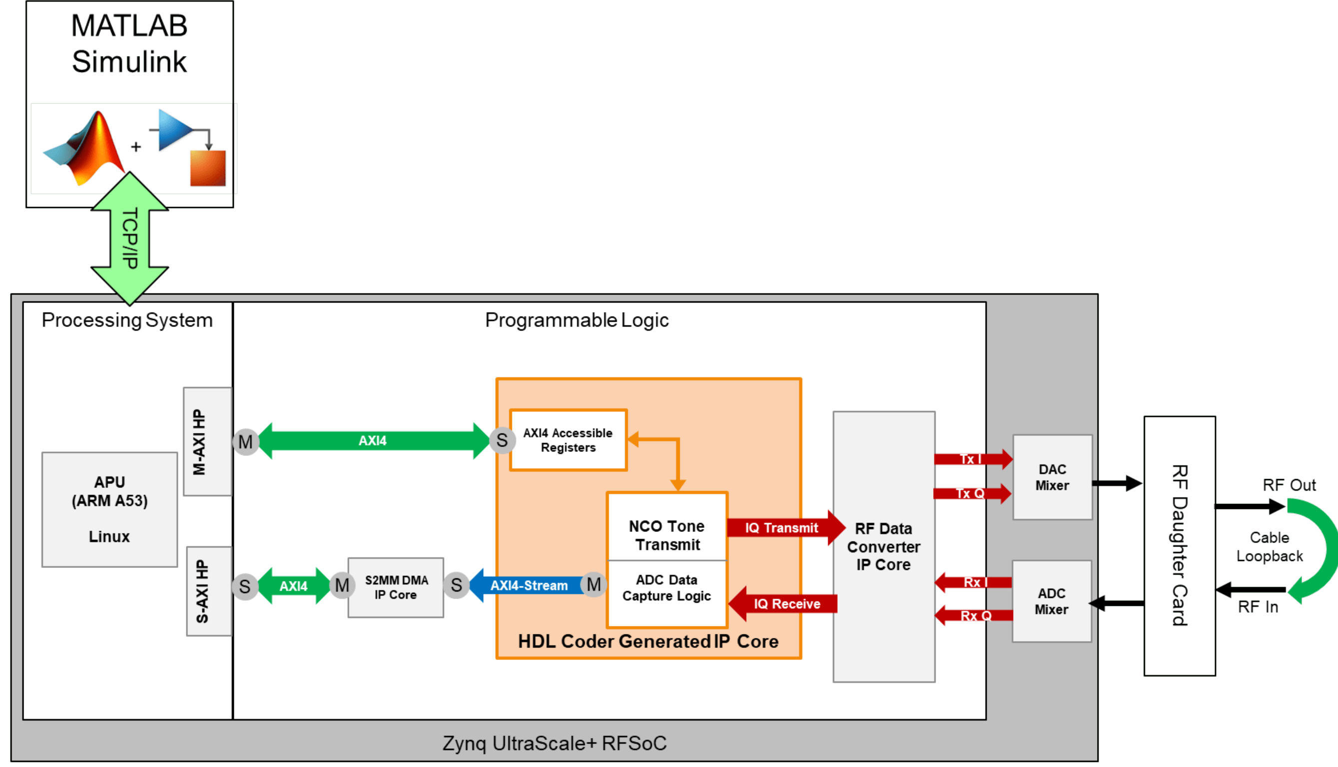
If your system is connected to the board, then running this script shows the MATLAB prompt update with messages indicating that parameters are adjusted. For this example, the sampling rate is set for 2.048 GSPS with 4x interpolation and decimation for all ADC and DAC tiles. Because you can dynamically control the converter with this script, you can rerun the script with different values, such as changing the sampling rates or interpolation modes.

For example, if you change the decimator and interpolator to 8 instead of 4, your sampling rate is cut by half. You can transmit only the DAC NCO tone to 128 MHz (Fs as 1024 and Nyquist as Fs/2) rather than 256 MHz. This example also provides a log file that shows the TCP/IP activity between host and target for more details on what was sent back and forth.

IQ Mixer Mode Capture

This example shows how to enable the RFSoC built-in numerically-controlled oscillator (NCO) mixer. The mixer design uses a different data format that, instead of providing real signals, provides a complex in-phase and quadrature (IQ) signal to a digital-to-analog converter (DAC) and an analog-to-digital converter (ADC). There is more data to transfer to direct memory access (DMA) and back to MATLAB® given that you need to transfer I and Q samples back. You can accommodate this larger data width by increasing the DMA data width to 128 bits.

You must configure settings such as the local oscillator (LO) mixer frequency and enable other mixer properties that are relevant to the design. To change these properties at run time, you can use the `soc.RFDataConverter` System object™. The HDL Workflow Advisor autogenerates a script to assist you with changing properties and uploads a CFG file to the board.



Requirements

- Vivado® Design Suite with a supported version listed in “Supported EDA Tools and Hardware”
- Xilinx® Zynq® UltraScale+™ ZCU111 evaluation kit or Xilinx Zynq UltraScale+ ZCU216 evaluation kit
- HDL Coder™
- HDL Coder Support Package for Xilinx FPGA and SoC Devices

Open Example

Open the example project and copy the example files to a temporary directory.

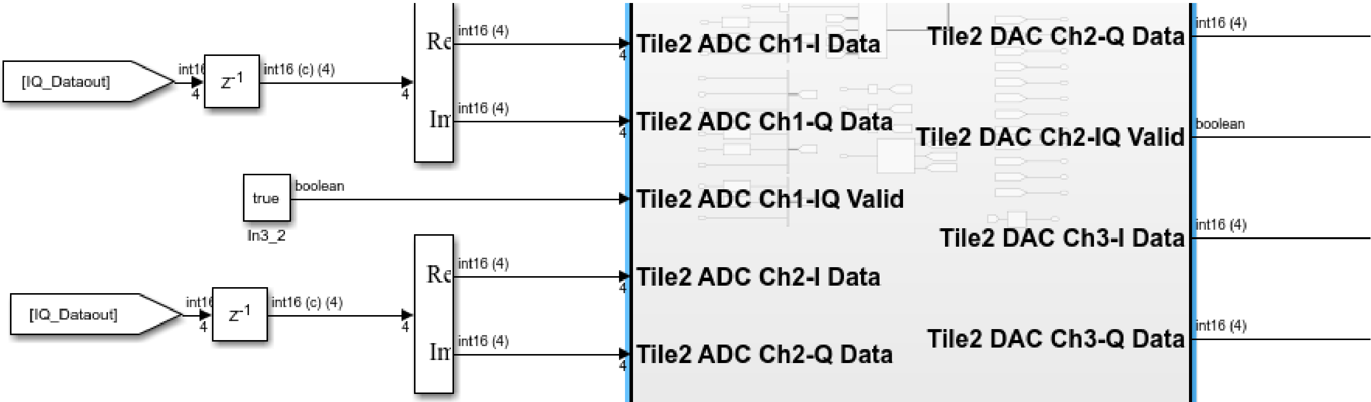
1. Navigate to the RFSoC root example directory of HDL Coder Support Package for Xilinx FPGA and SoC Devices by entering these commands at the MATLAB command prompt.

```
example_root = (hdlcoder_rfsoc_examples_root)
cd (example_root)
```

2. Copy all of the example files in the `IQDataCapture` folder to a temporary directory.

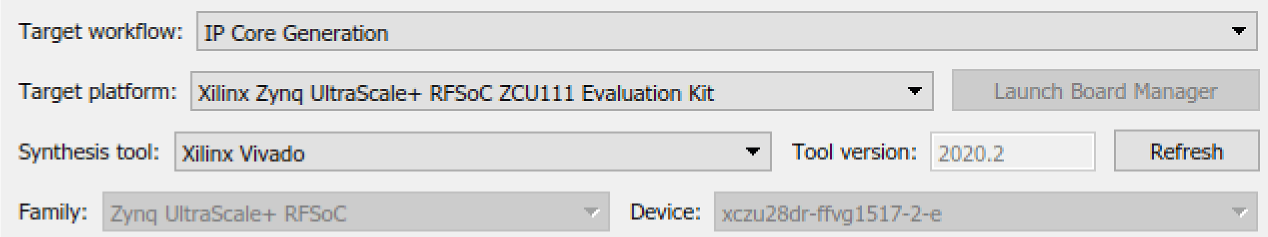
Generate HDL and Synthesize Bitstream

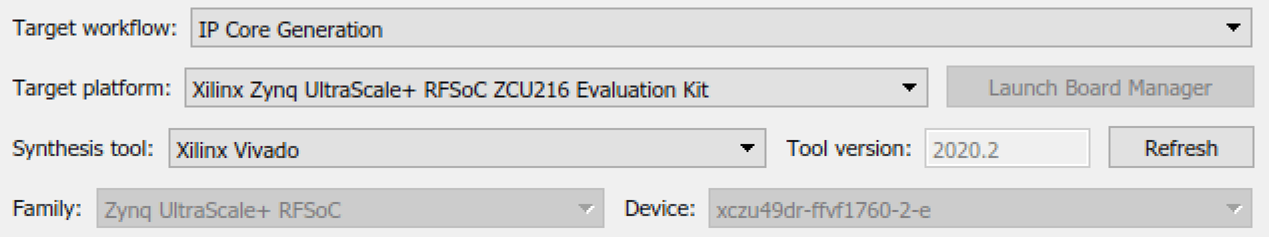
When you enable the mixer in the RFSoC, adhere to the required data formats that the IP expects. For the ADC, the IP core separates the I and Q channels. For the DAC , the IP core concatenates both I and Q channels together. For convenience, this bit concatenation is handled outside of the generated HDL and in Vivado. You need to model the I and Q channels separately for only the DAC and ADC.



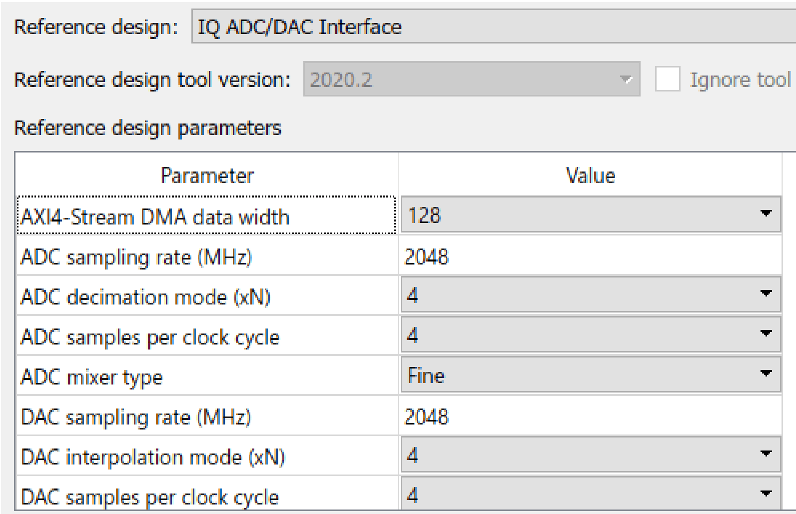
This example uses four complex samples per clock cycle at 512 mega samples per second (MSPS). Each I and Q sample is 16 bits, which results in a total of 128 bits for channel I and a total of 128 bits for channel Q. From the Simulink® modeling perspective, two parts (I and Q) that exist, each with four samples per clock cycle.

To proceed with the HDL code generation, right-click the subsystem. Select **HDL Code**, then click **HDL Workflow Advisor**. In step 1.1 of the HDL Workflow Advisor, select **Target platform** as `Xilinx Zynq Ultrascale+ RFSoC ZCU111 Evaluation Kit` or `Xilinx Zynq Ultrascale+ RFSoC ZCU216 Evaluation Kit`.





In step 1.2, select **Reference design** as `IQ ADC/DAC Interface`.

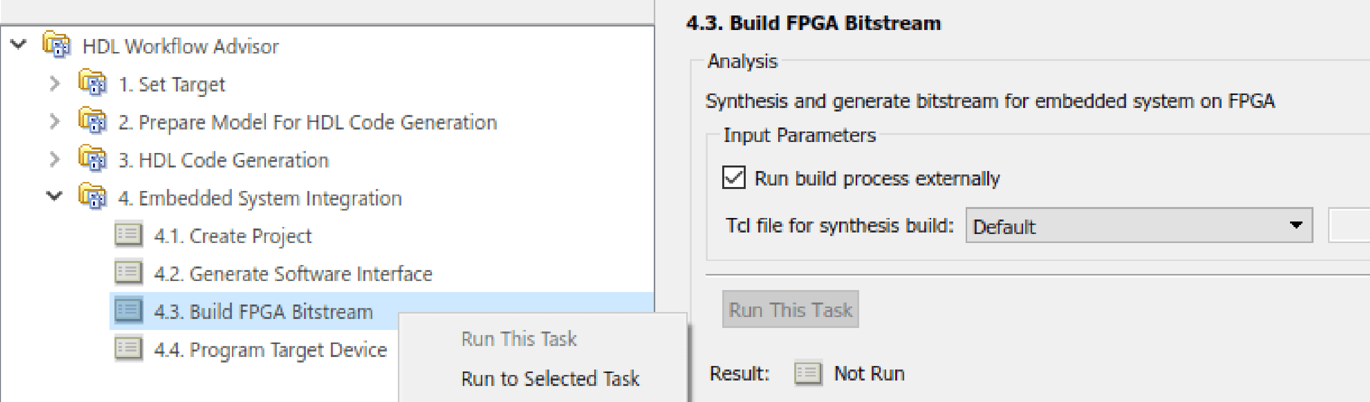


You can also specify a LO setting, which is used at board startup. The LO setting can be run-time adjustable. Before proceeding to the next step, set these reference design parameters to the indicated values.

- **AXI4-Stream DMA data width** to `128`
- **ADC sampling rate (MHz)** to `2048`
- **ADC decimation mode (xN)** to `4`
- **ADC samples per clock cycle** to `4`
- **ADC mixer type** to `Fine`
- **DAC sampling rate (MHz)** to `2048`
- **DAC interpolation mode (xN)** to `4`
- **DAC samples per clock cycle** to `4`
- **DAC mixer type** to `Fine`
- **ADC/DAC NCO mixer LO (GHz)** to `0.5`
- **Enable multi-tile sync** to `false`

If you are using a ZCU216 board, additionally set the **DAC DUC mode** parameter to `Full DUC Nyquist (0-Fs/2)`.

To build the FPGA bitstream design, right-click **Build FPGA Bitstream** and click **Run to Selected Task**.

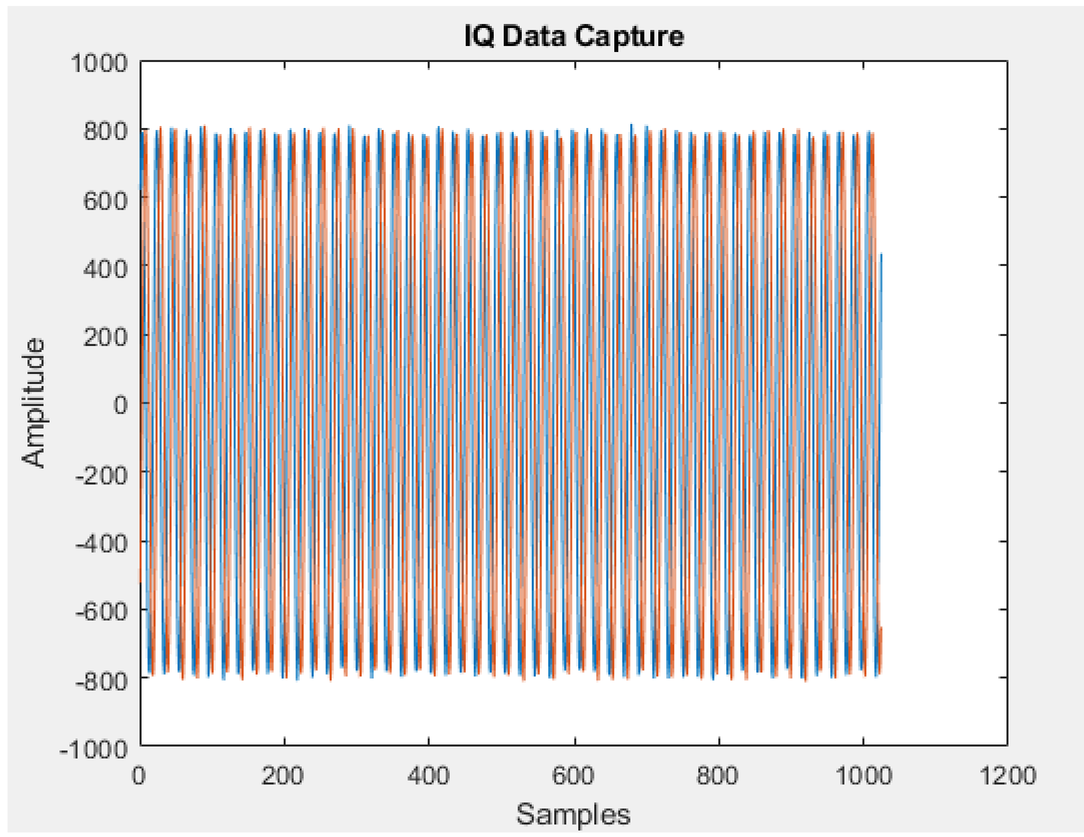


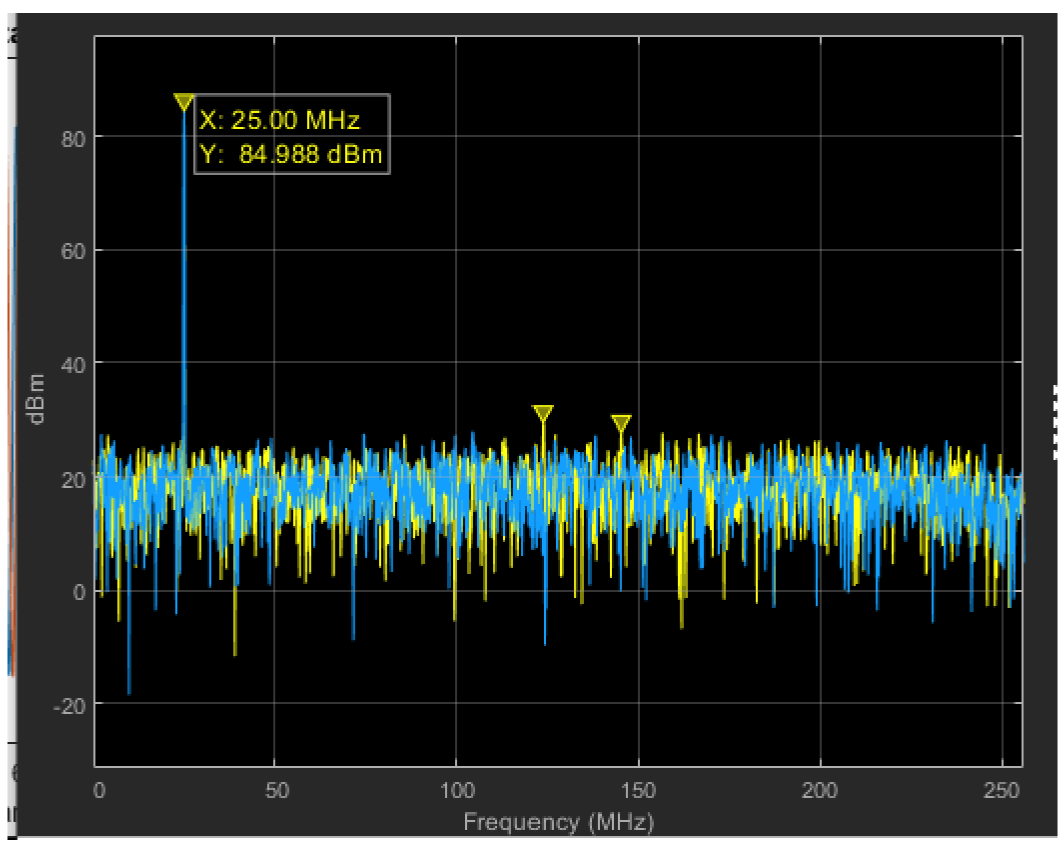
IQ Data Capture

To capture IQ data in this loopback test, perform a full HDL Workflow Advisor build and wait until the FPGA bitstream has been compiled. Afterward, program the board and run this capture script.

```
HostIO_rfsocIQDataCapture_interface.m
```

The capture loop triggers a register to initiate data capture logic to move IQ data into the DMA. The loop then reads the frame of IQ data and formats it to display these plots. To stop the capture, close the MATLAB plot.





Change Converter Settings: LO Mixer

You can set the LO from MATLAB by using the System object script at run time. This example provides a script to show an example of how you can use the RFSoC System to alter the mixer tone of the DAC and ADC. Open and examine this MATLAB script file: `HostIO_ChangeLO_rfsocIQDataCapture.m`.

In this example, this script shifts the LO tone such that the DAC experiences a shift of 10 MHz relative to the ADC LO, with a center frequency of 800 MHz.

Connect to the target board.

```
rfobj = soc.RFDataConverter('zu49dr','192.168.1.101');
setup(rfobj)
```

Specify LO Value in MHz. All tiles and channels use this LO Value.

```
LO_VALUE = 800;
```

Create a shift of 10 MHz in the DAC RF NCO.

```
rfobj.configureDACLocalOscillator(1,3,LO_VALUE+10);
```

Configure frequency of the ADC RF NCO.

```
rfobj.configureADCLocalOscillator(0,0,-LO_VALUE);
```

Disconnect the communication channel created between the `soc.RFDataConverter` System object and the `rftool` running on the SoC device.

```
release(rfobj)
```

After running this script, run the capture script again and examine the spectrum plot.

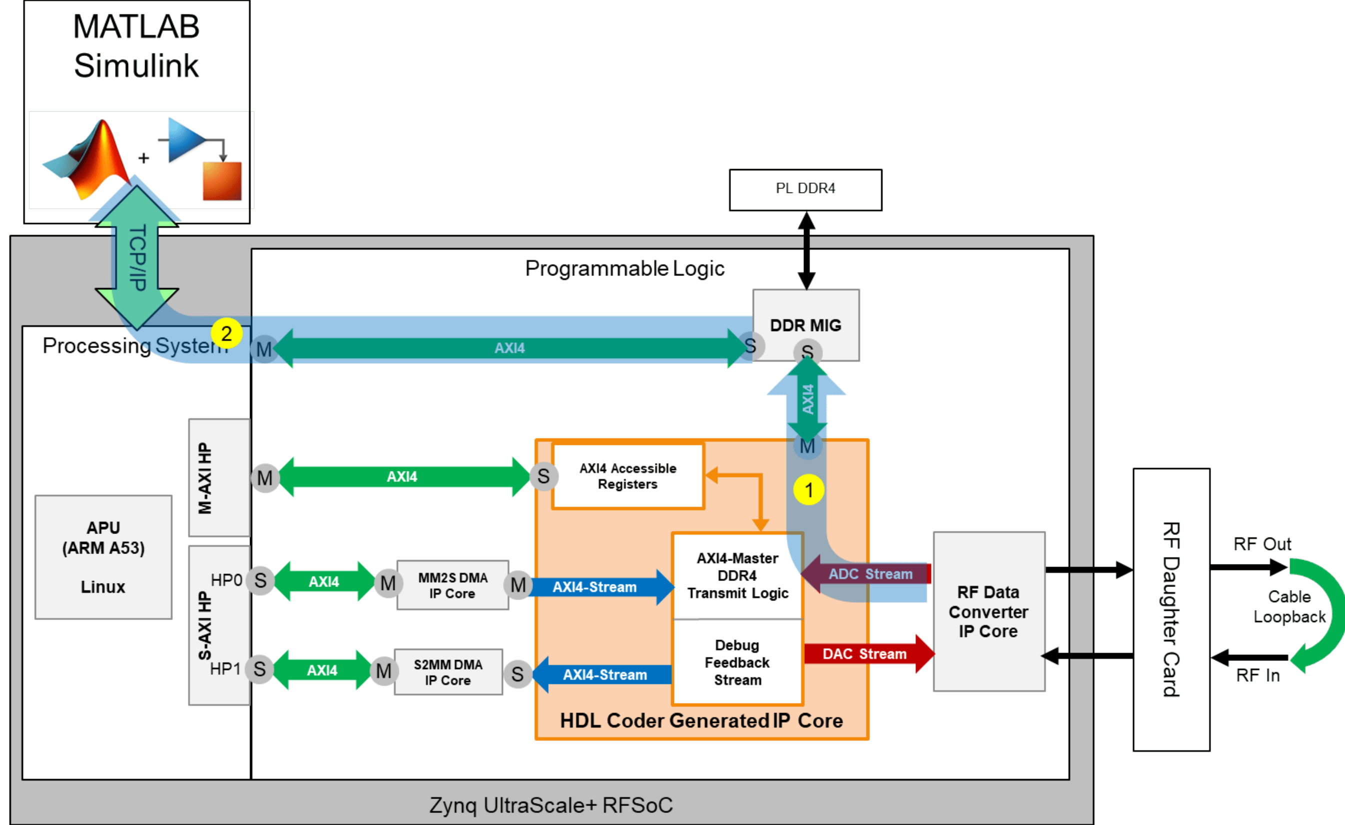
```
HostIO_ChangeLO_rfsocIQDataCapture
```

The tone, which was previously at 25 MHz, is now shifted by 10 MHz to 35 MHz.

PL-DDR4 ADC Data Capture

This example shows how to perform analog-to-digital converter (ADC) data captures with programmable logic (PL) double data rate 4 (DDR4) memory.

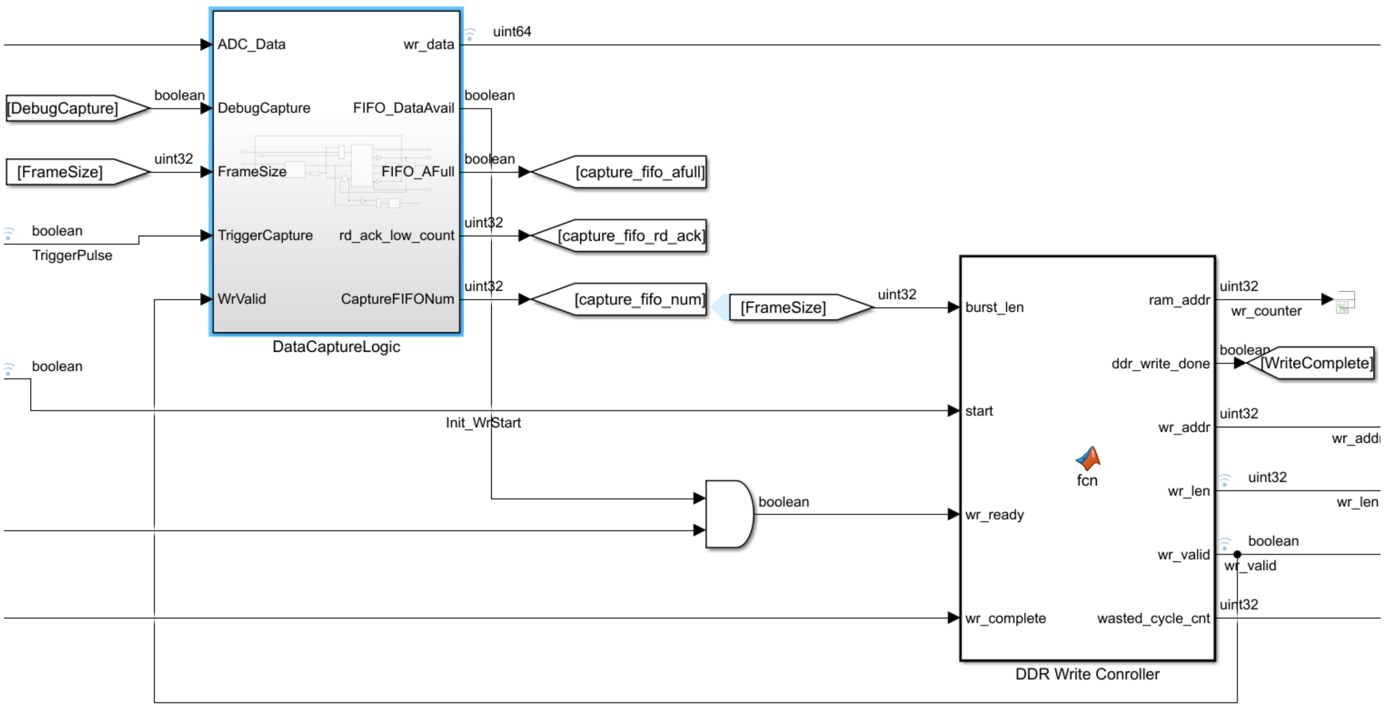
Storing data into PL-DDR4 memory can be advantageous because of the large amount of space available to read and write to. A total of 4 gigabytes is available to access from the FPGA. The HDL Coder™ reference designs provide a means for your IP design to connect to this memory interface by using AXI4-Master. AXI4-Master requires state-machine logic to perform reading and writing using multiple control signal lines.



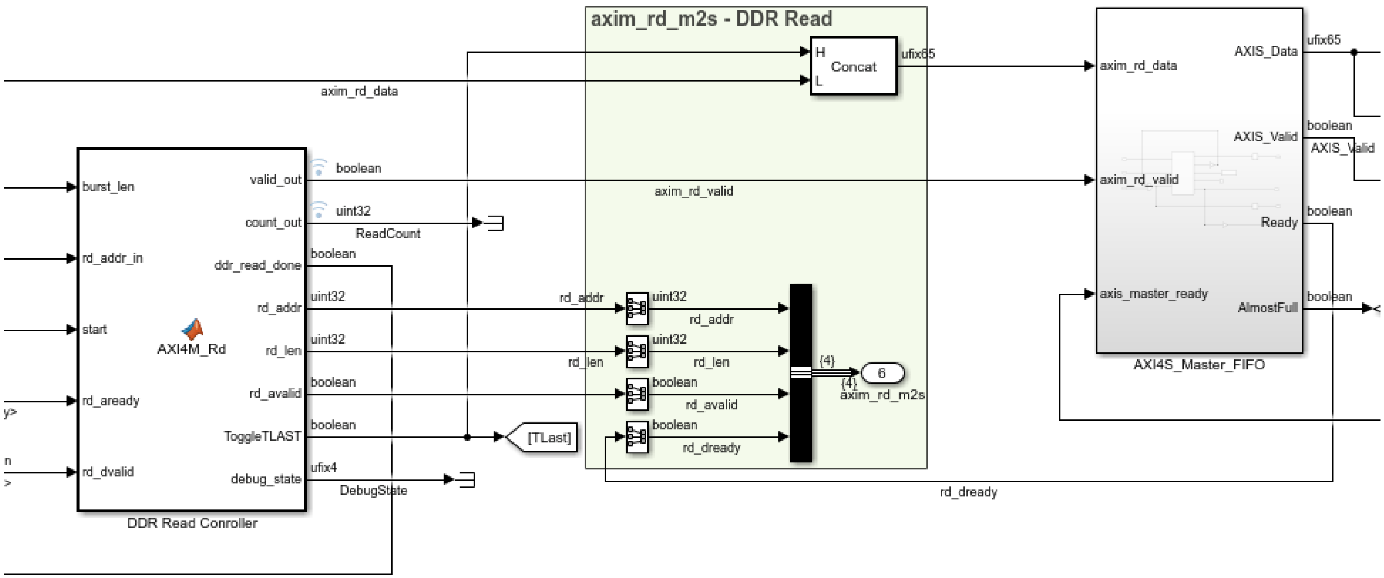
State-machine logic in the IP design helps coordinate this transfer by issuing commands originating from MATLAB® as AXI4 register writes. These register writes command the state machine to be configured in different modes. For this data capture example, these modes are broken down into two main operations.

The first arrow (labeled 1) in the first figure shows a data path connecting the ADC input stream directly into AXI4-Master to write to the memory interface generator (MIG). From MATLAB, an AXI4 register is written to the state machine to issue a command that data is ready to be written. The state machine then waits for acknowledgement from the DDR4 slave device that it is ready to accept samples. Afterward, data can flow through to the MIG. Because there is a bit of wait time on issuing burst write commands, a back pressure first in, first out (FIFO) is used to help temporarily store data during these periods. In some cases, the amount of back pressure experienced can exceed the FIFO, causing data to be lost. This example inserts several registers to help determine if this situation

arises. Typically, this issue happens if the requested data capture size is large enough to induce enough back pressure events to exceed the FIFO capacity. Constant writing to DDR4 memory can be sustained to only a certain frame length before back pressure occurs. Increasing the length of the FIFO can help alleviate this issue at the cost of more FPGA resources. This figure shows the section of the model that is responsible for writing to the digital-to-analog converter (DAC).



The second arrow (labeled 2) in the first figure shows the data path of reading from DDR4 memory and returning the frame back to MATLAB or Simulink® over the Ethernet network. Similar to the data path labeled 1, the data path labeled 2 is also controlled with a state machine that is adjustable with AXI4 registers. For each read command issued, an offset address is used to specify which spot in PL-DDR4 memory to read from. This figure shows the read controller logic.



Requirements

- Vivado® Design Suite with a supported version listed in “Supported EDA Tools and Hardware”
- Xilinx® Zynq® UltraScale+™ ZCU111 evaluation kit or Xilinx Zynq UltraScale+ ZCU216 evaluation kit
- HDL Coder
- HDL Coder Support Package for Xilinx FPGA and SoC Devices

Open Example

Open the example project and copy the example files to a temporary directory.

1. Navigate to the RFSoC root example directory of HDL Coder Support Package for Xilinx FPGA and SoC Devices by entering these commands at the MATLAB command prompt.

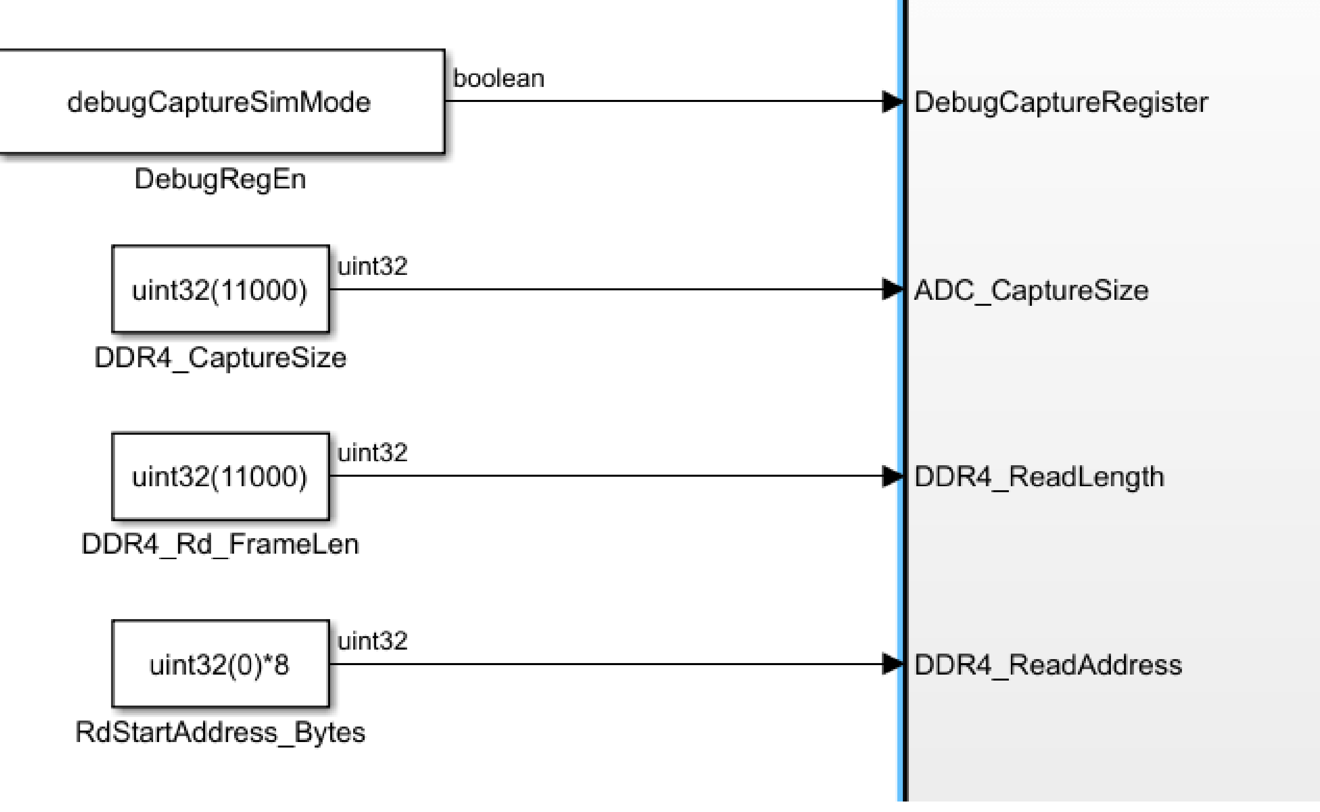
```
example_root = (hdlcoder_rfsoc_examples_root)
cd (example_root)
```

2. Copy all of the example files in the `DDR4_ADCCapture` folder to a temporary directory.

Simulate PL-DDR4 ADC Capture Model

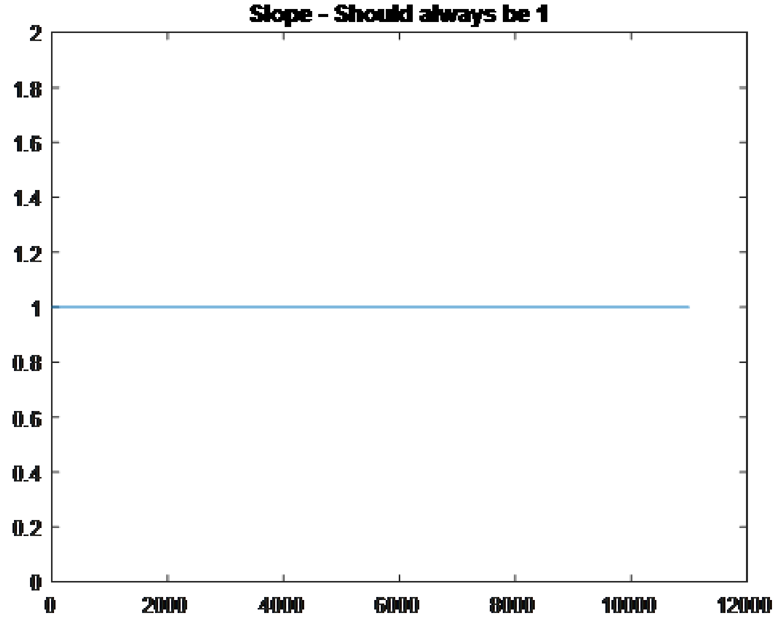
To examine how these operations take place, open the model `rfsocADCDDR4Capture.slx` and simulate the design.

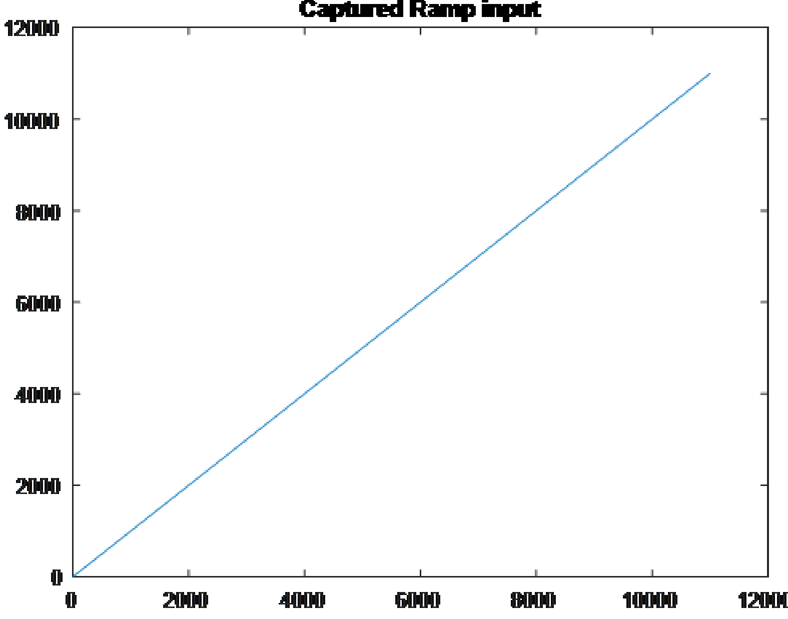
By default, the simulation uses the `debugCaptureSimMode` set to 1. With this mode, the capture logic captures counter data instead of ADC data. The goal is to validate that the counter data written to PL-DDR4 memory is the same data read out when issuing a read command later.



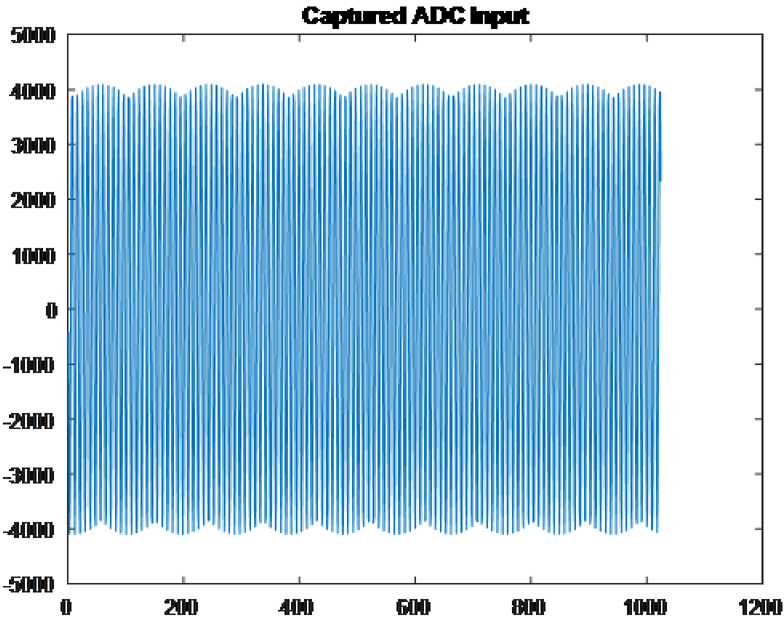
The other input AXI4 registers specify the amount of data to capture (`ADC_CaptureSize`), the amount of data to write back to the DMA AXI4-Stream per read (`DDR4_Rd_FrameLen`), and the offset address of where to read from in PL-DDR4 memory (`DDR4_ReadAddress`).

The model stops simulating at the end of the last sample of the frame of data that is sent over the DMA. Afterward, validate that the data written into PL-DDR4 memory is the same as the data that was read out by running the MATLAB script `simValidateADCDDR4Capture.m`.





The counter value matches expectations when read out of the DDR4 memory and accessed via the AXI4-Stream DMA. Next, disable debug mode and configure the simulation to capture ADC data instead. Open the script `rfsocADCDDR4CaptureInit.m`, and then set `debugCaptureSimMode` to `false`. Rerun the simulation, and then plot the data by running the script `simValidateADCDDR4Capture.m`.



Generate HDL and Synthesize Bitstream

Open the model `rfsocADCDDR4Capture.slx`, and then right-click the `ADC_DDR4_Data_Capture` subsystem. Select **HDL Code**, then click **HDL Workflow Advisor**.

In step 1.1 of the HDL Workflow Advisor, select **Target platform** as `Xilinx Zynq Ultrascale+ RFSoC ZCU111 Evaluation Kit` or `Xilinx Zynq Ultrascale+ RFSoC ZCU216 Evaluation Kit`.

In step 1.2, select **Reference design** as `Real ADC/DAC Interface with PL-DDR4`.

Before proceeding to the next step, set these reference design parameters to the indicated values.

- **AXI4-Stream DMA data width** to `64`
- **ADC sampling rate (MHz)** to `2048`
- **ADC decimation mode (xN)** to `4`
- **ADC samples per clock cycle** to `4`
- **ADC mixer type** to `Bypassed`
- **DAC sampling rate (MHz)** to `2048`
- **DAC interpolation mode (xN)** to `4`
- **DAC samples per clock cycle** to `4`
- **DAC mixer type** to `Bypassed`
- **ADC/DAC NCO mixer LO (GHz)** to `Disabled`
- **Enable multi-tile sync** to `false`

If you are using a ZCU216 board, additionally set the **DAC DUC mode** parameter to `Full DUC Nyquist (0-Fs/2)`.

Initiate the bitstream compilation. After the compilation is complete, use a programming script to program the FPGA bit file.

Collect Captured PL-DDR4 ADC Data

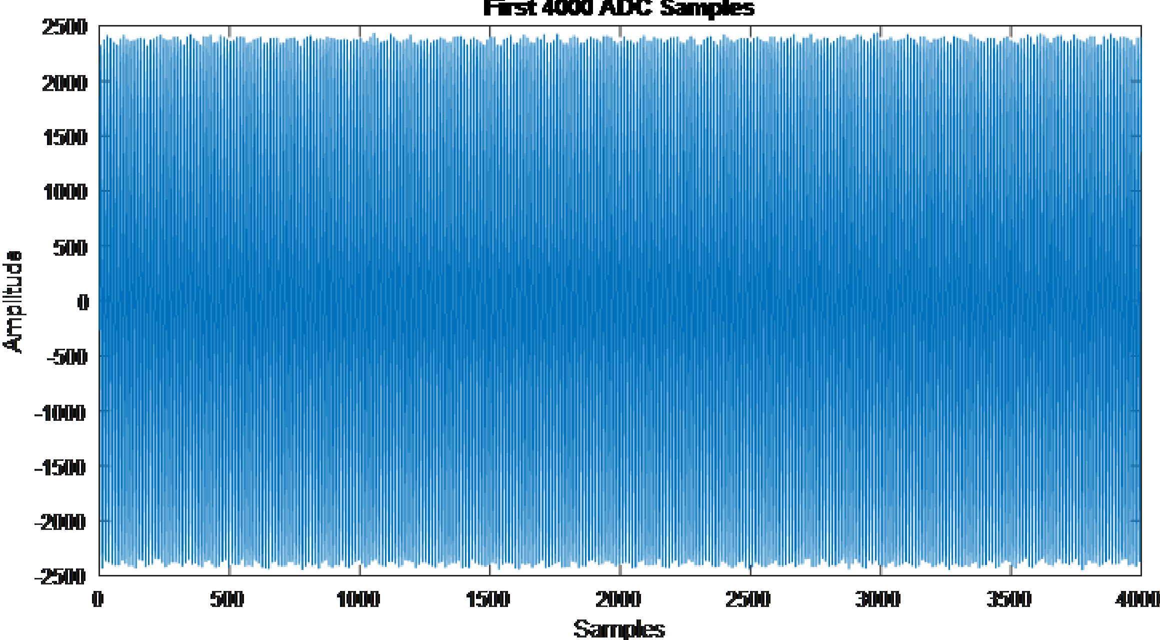
After you create and program the FPGA bit file onto the board, you can capture data.

In this capture scenario, the goal is to capture 4 million data points of ADC samples. Because the design is operating at four samples per clock, the data width of each data point written to DDR4 memory is 64 bits. Each data point has an 8 byte offset. To retrieve the samples, use the shared-memory FPGA I/O API over a TCP/IP connection.

The script performs these steps.

1. Initiate a data capture and store 4 million samples into DDR4 memory.
2. Check to see if samples were dropped in the data collection due to back pressure.
3. Use shared memory to directly read samples from DDR4 memory from ARM®-Linux® and transfer samples back to MATLAB.

This plot shows the first 4000 samples when the capture loop execution finishes.



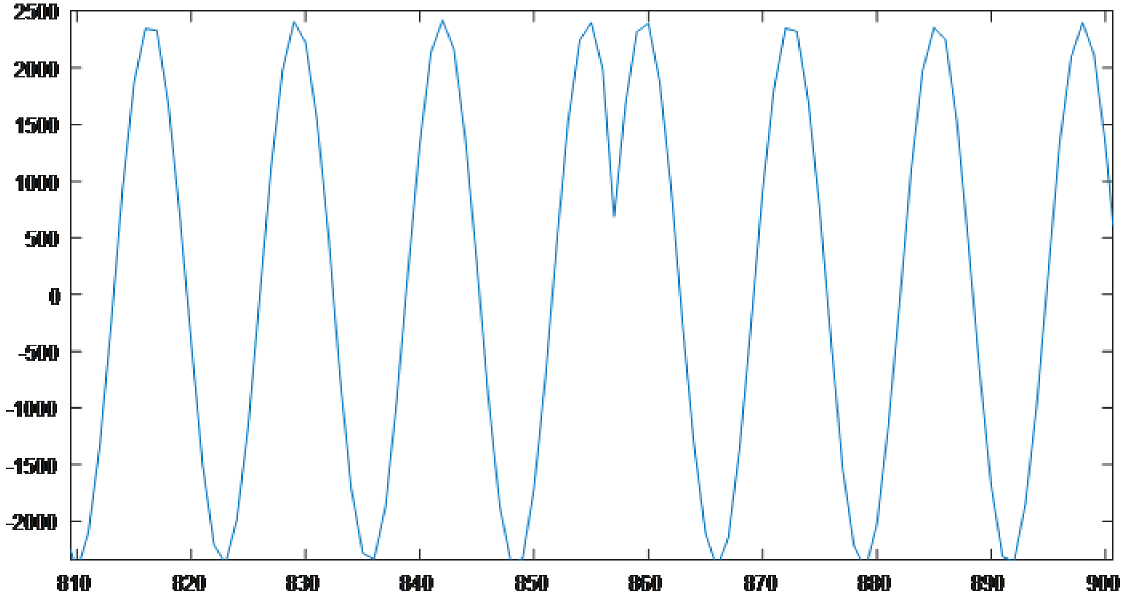
In this example, a capture size of 4 million is possible. Increasing the capture size can yield results where data is missing. For instance, setting the capture length to 8 million (by changing the frame size to 2e6 rather than 1e6).

```
% %% Parameters
% DebugMode = 0;
% incrScale = 2^14/512e6; % Used to adjust NCO frequency
% Fs = 512e6;
% CaptureSize = 2e6;      % Total amount of four-vector contigious samples to store in PL-DDR4
% DDR4_ReadLen = 100e3;   % Amount of four-vector samples to retrieve per read from PL-DDR4
```

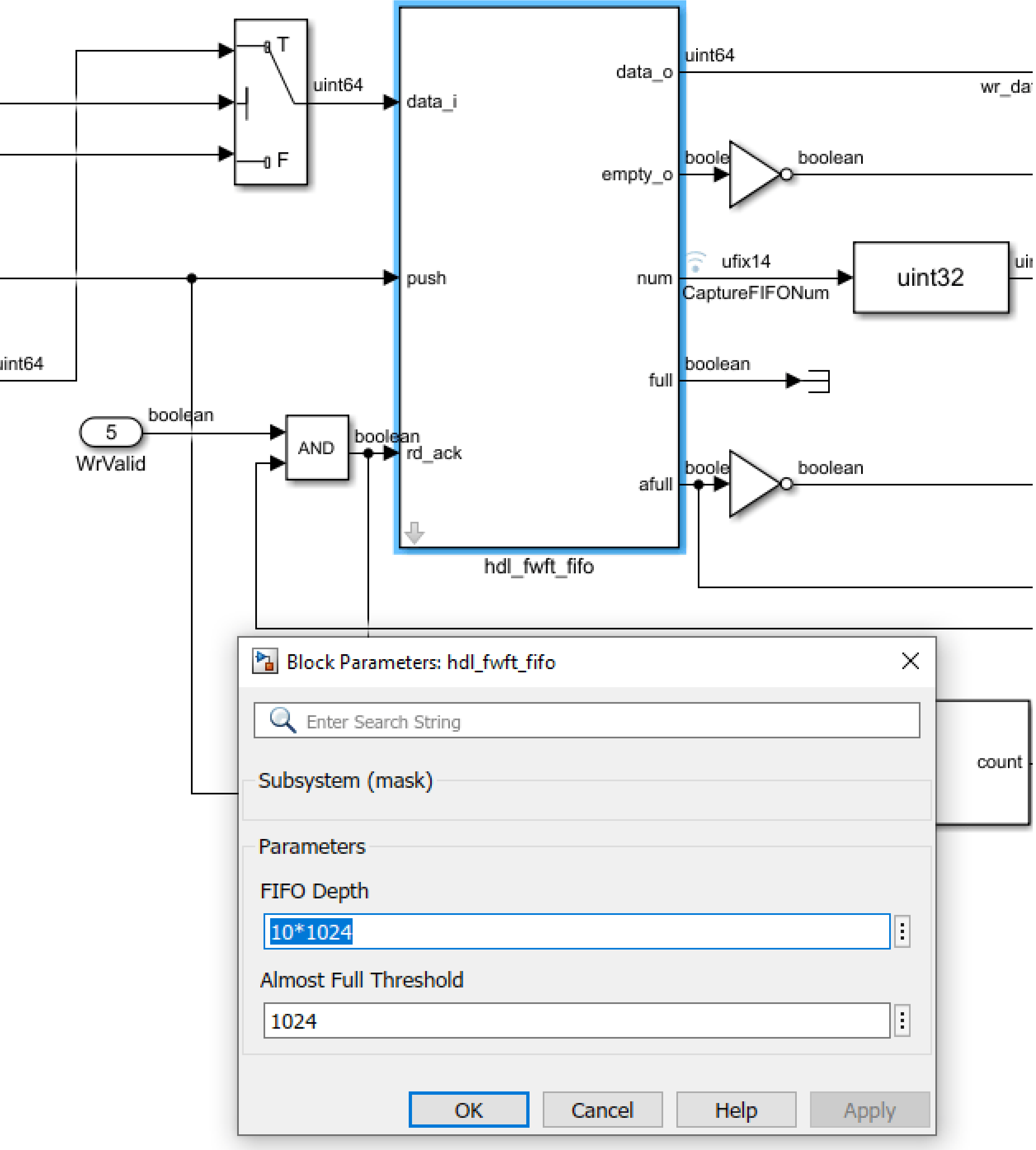
Run this script again.

```
HostIOs_rfsocADCDDR4Capture_interface
```

From these 8 million samples captured, 6329 four-vector samples were dropped, (that is, 25,316 ADC samples are missing). In plots of the captures, a missing sample is displayed as a discontinuity, which you can view by using the zoom feature of the plots.



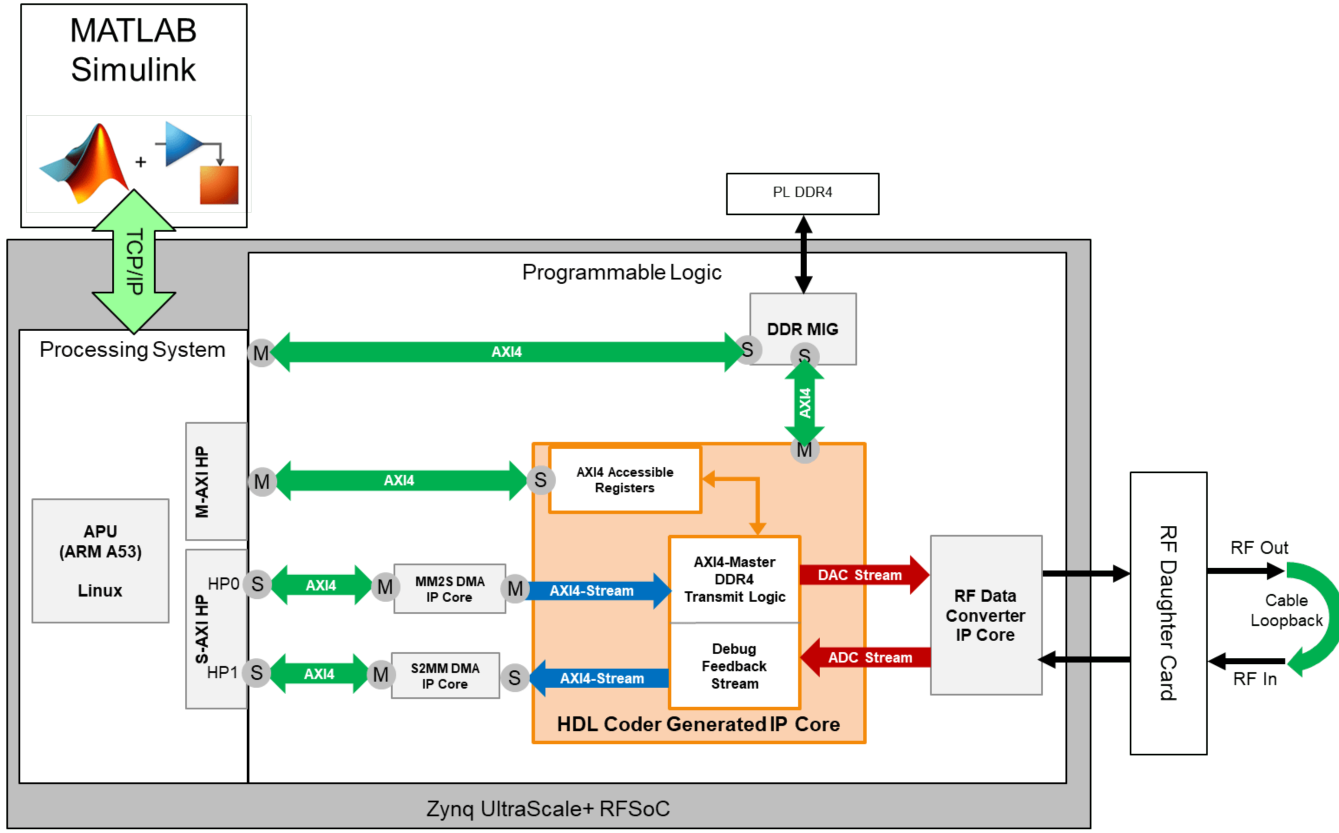
The spectrum might not look as severely affected because many samples exist in the frame, but this many samples can still cause issues depending on your end applications. Choosing a back-pressure FIFO depth that is optimal to your design specifications is one workaround to help mitigate missing samples. For more details on where to change this FIFO depth, see this part of the model: `rfsocADCDDR4Capture/ADC_DDR4_Data_Capture/DDR_Capture_Logic/DataCaptureLogic`.



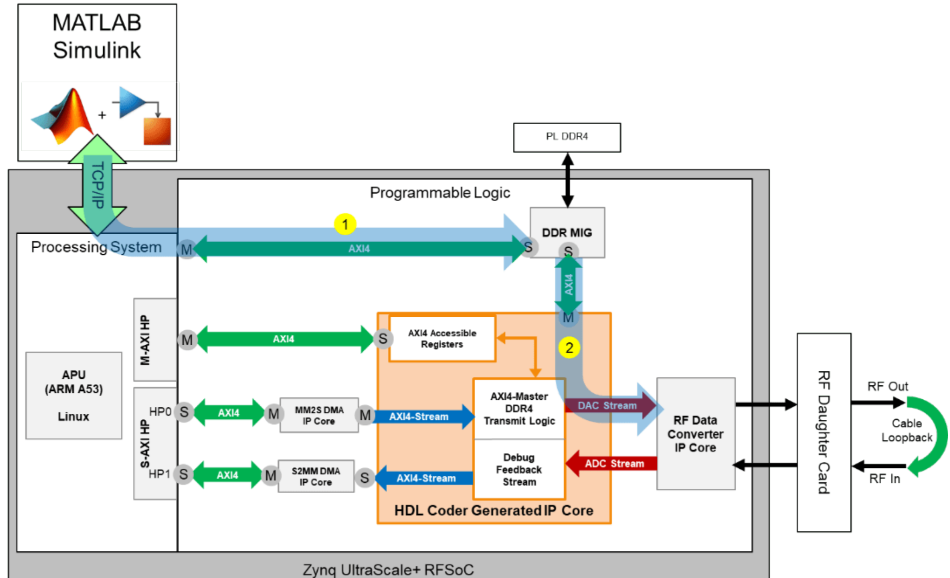
T
F
uint64
data_i
data_o
uint64
wr_da
boole
empty_o
boolean
push
num
ufix14
CaptureFIFONum
uint32
uint64
full
boolean
5
WrValid
boolean
AND
boolean
rd_ack
afull
boole
boolean
hdl_fwft_fifo
Block Parameters: hdl_fwft_fifo
Enter Search String
Subsystem (mask)
Parameters
FIFO Depth
10*1024
Almost Full Threshold
1024
OK
Cancel
Help
Apply
count

DAC PL-DDR4 Transmit

This example shows how to transmit waveforms out of the digital-to-analog converter (DAC) that is read from programmable logic (PL) double data rate 4 (DDR4) memory. Using PL-DDR4 memory offers advantages such as the ability to access 4 gigabytes of memory space. This amount of memory can help facilitate applications requiring large waveform transmissions that are not normally feasible with FPGA block random access memory (BRAM). This figure shows the relevant intellectual property (IP) interfaces and connectivity that are used in this design.



This IP design uses a shared-memory region between the Linux® operating system and PL-DDR4 memory. The memory interface uses the processing system (PS) AXI4 master high performance (HP) lines to directly access PL-DDR4 memory such that it can be accessed with reads and writes. A MATLAB® interface enables access to this shared memory region. This figure shows the flow of the data.



In this figure, the first arrow (labeled 1) shows the moving data over the Ethernet network to the PL-DDR4 memory. The second arrow (labeled 2) shows data that is read from the PL-DDR4 memory to the DAC. The PL design waits on an AXI4 register trigger from MATLAB to initiate data movement and read data out of memory to the DAC.

Requirements

- Vivado® Design Suite with a supported version listed in “Supported EDA Tools and Hardware”
- Xilinx® Zynq® UltraScale+™ ZCU111 evaluation kit or Xilinx Zynq UltraScale+ ZCU216 evaluation kit
- HDL Coder
- HDL Coder Support Package for Xilinx FPGA and SoC Devices

Open Example

Open the example project and copy the example files to a temporary directory.

1. Navigate to the RFSoC root example directory of HDL Coder Support Package for Xilinx FPGA and SoC Devices by entering these commands at the MATLAB command prompt.

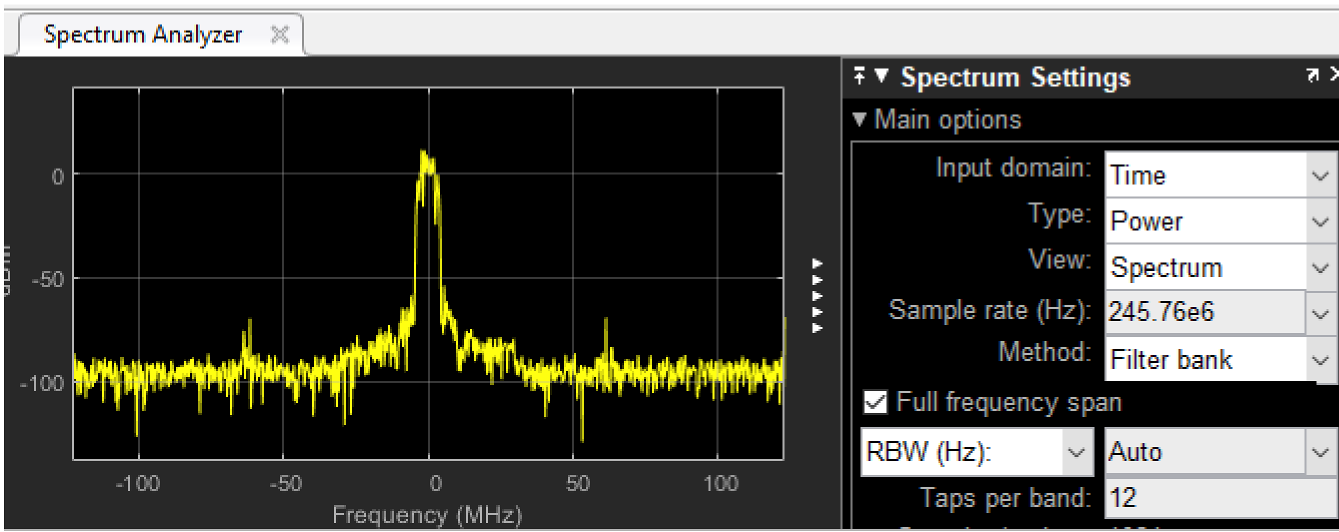
```
example_root = (hdlcoder_rfsoc_examples_root)
cd (example_root)
```

2. Copy all of the example files in the `DDR4_DACWrite` folder to a temporary directory.

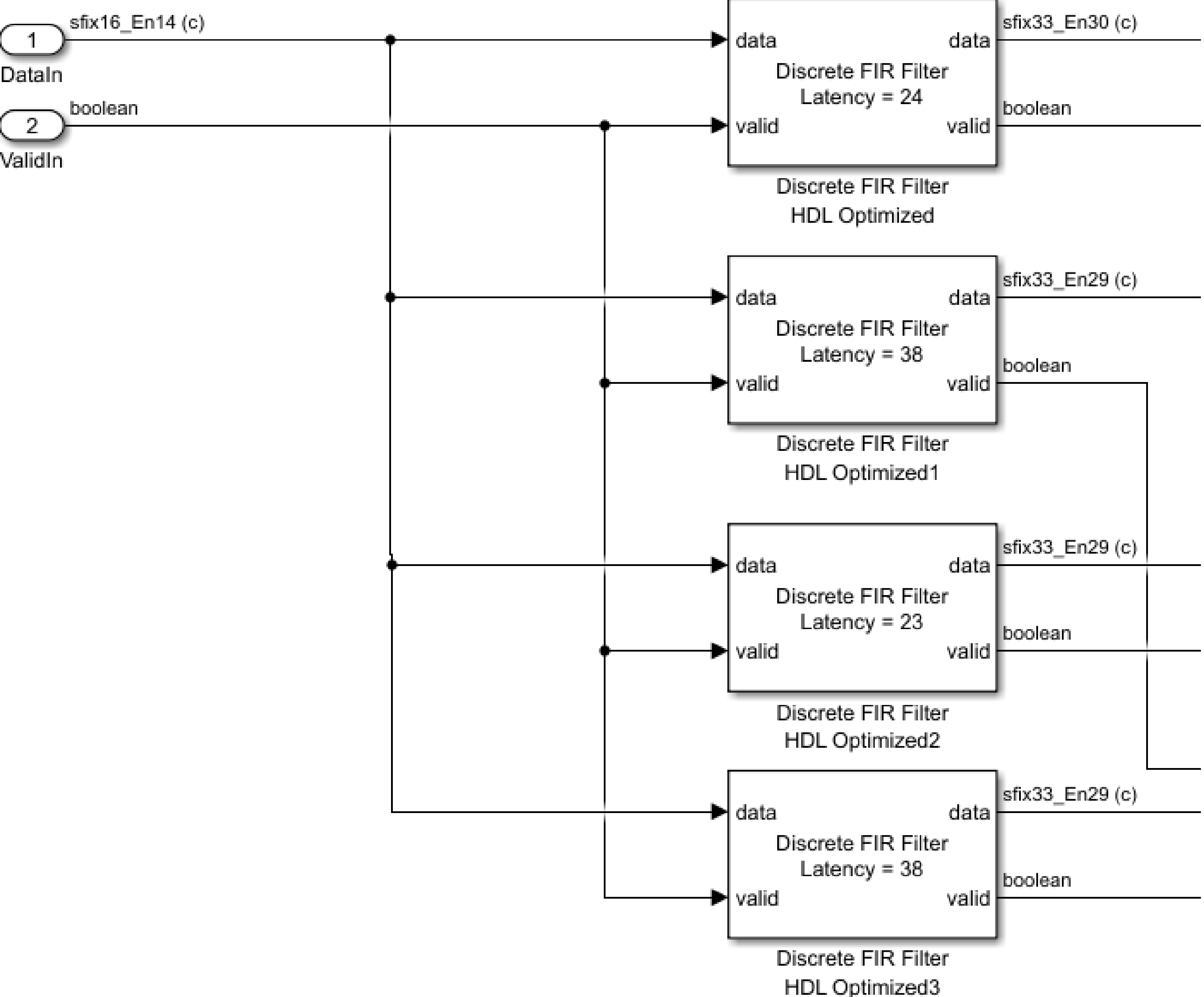
Simulate PL-DDR4 DAC Transmit Model

The HDL model for DDR4 DAC transmit is `waveformWriteDAC_DDR4.slx`. This model simulates the data transfer of using AXI4 master. The waveform that is transmitted can be either an LTE or single-tone waveform, depending on your selection. See the MATLAB script `model_init.m`.

This figure shows an LTE-transmitted waveform. To synthesize this waveform, you must have the LTE Toolbox™ product. If you do not have this toolbox the waveform that is loaded into memory for the simulation is a 15 MHz complex tone instead.



The waveform in the design is upconverted using interpolation by 4x. The waveform that is stored in memory is 61.44 MHz. Upconvert the data rate to 1966.08 MHz by using a 4x interpolator in the PL device under test (DUT) and the 8x interpolator on the RFSoC. This figure shows a vector interpolator to execute samples and produce a vector of four from a single scalar input.



See the subsystem: `waveformWriteDAC_DDR4/DDR4_DAC_Transmit/Interpolation_4x/FIR_Vector_Interp`. The design also implements the capture logic to collect analog-to-digital converter (ADC) samples using an AXI4-register capture.

Generate HDL and Synthesize Bitstream

Open the model `waveformWriteDAC_DDR4.slx`, and then right-click the `DDR4_DAC_Transmit` subsystem. Select **HDL Code**, then click **HDL Workflow Advisor**.

In step 1.1 of the HDL Workflow Advisor, select **Target platform** as `Xilinx Zynq Ultrascale+ RFSoC ZCU111 Evaluation Kit` or `Xilinx Zynq Ultrascale+ RFSoC ZCU216 Evaluation Kit`.

In step 1.2, select **Reference design** as `IQ ADC/DAC Interface with PL-DDR4`.

- **AXI4-Stream DMA data width** to 128
- **ADC sampling rate (MHz)** to 1966.08
- **ADC decimation mode (xN)** to 8
- **ADC samples per clock cycle** to 4
- **ADC mixer type** to Fine
- **DAC sampling rate (MHz)** to 1966.08
- **DAC interpolation mode (xN)** to 8
- **DAC samples per clock cycle** to 4
- **DAC mixer type** to Fine
- **ADC/DAC NCO mixer LO (GHz)** to 0.5
- **Enable multi-tile sync** to true

You can set the numerically-controlled oscillator (NCO) mixer frequency to any value and adjust it at run time from MATLAB. If you are using a ZCU216 board, additionally set the **DAC DUC mode** parameter to Full DUC Nyquist (0-Fs/2).

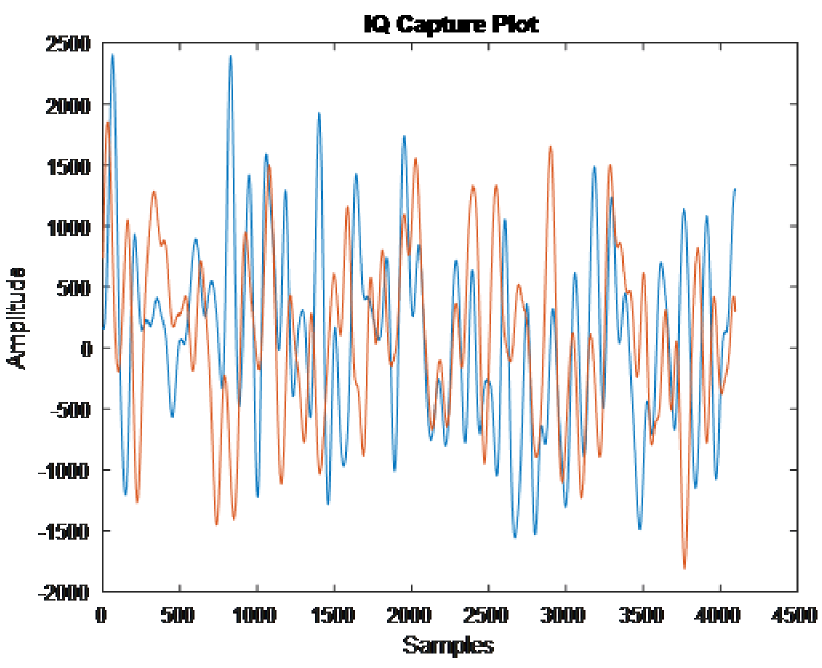
Transmit PL-DDR4 DAC Data

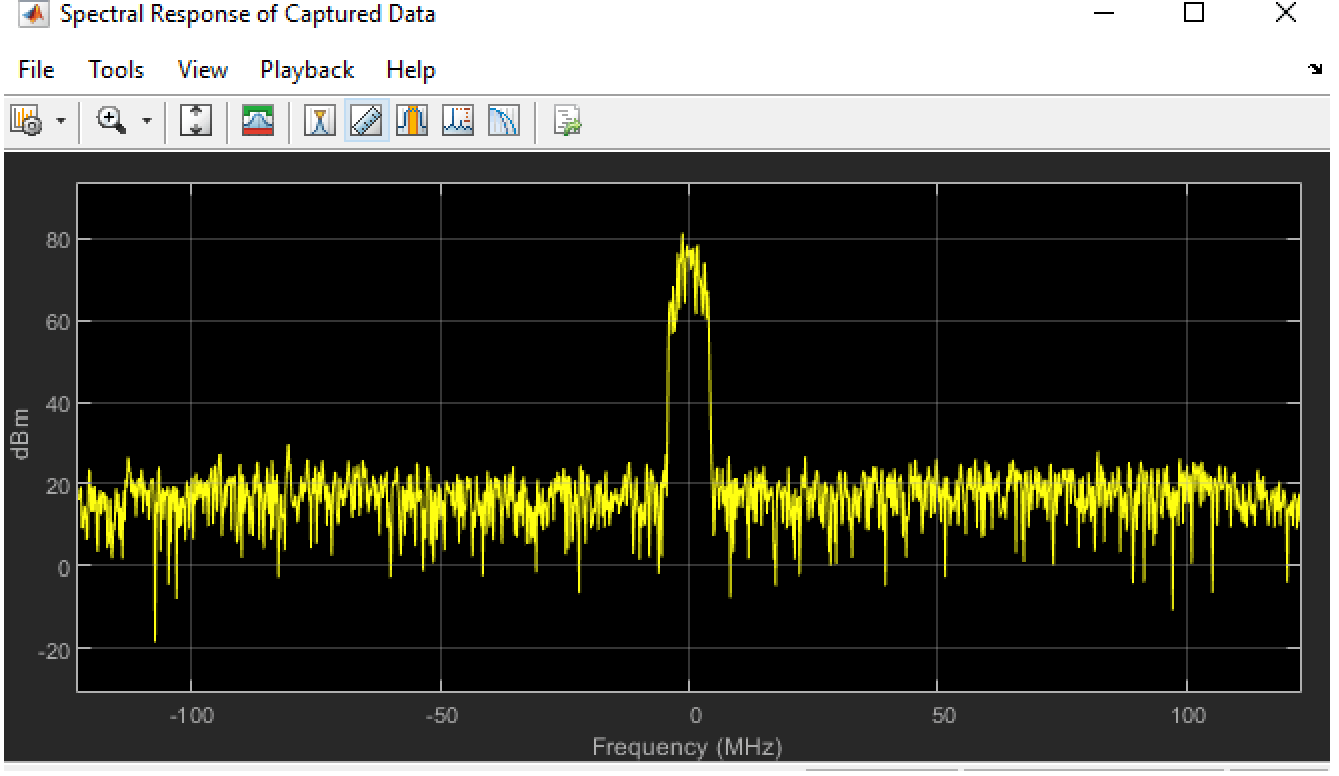
After you program the FPGA bitstream, run the script fpgaio_TxWaveformAndCapture.m. The script performs these steps.

1. Create a complex waveform of size 30.72e5 samples and package the waveform into a data format of int32.
2. Write the waveform into the PL-DDR4 memory and command the DUT to begin transmitting data over AXI4.
3. Trigger a data capture of 4096 samples to move ADC data into direct memory access (DMA).
4. Display the results of the captured ADC data in MATLAB by using this command.

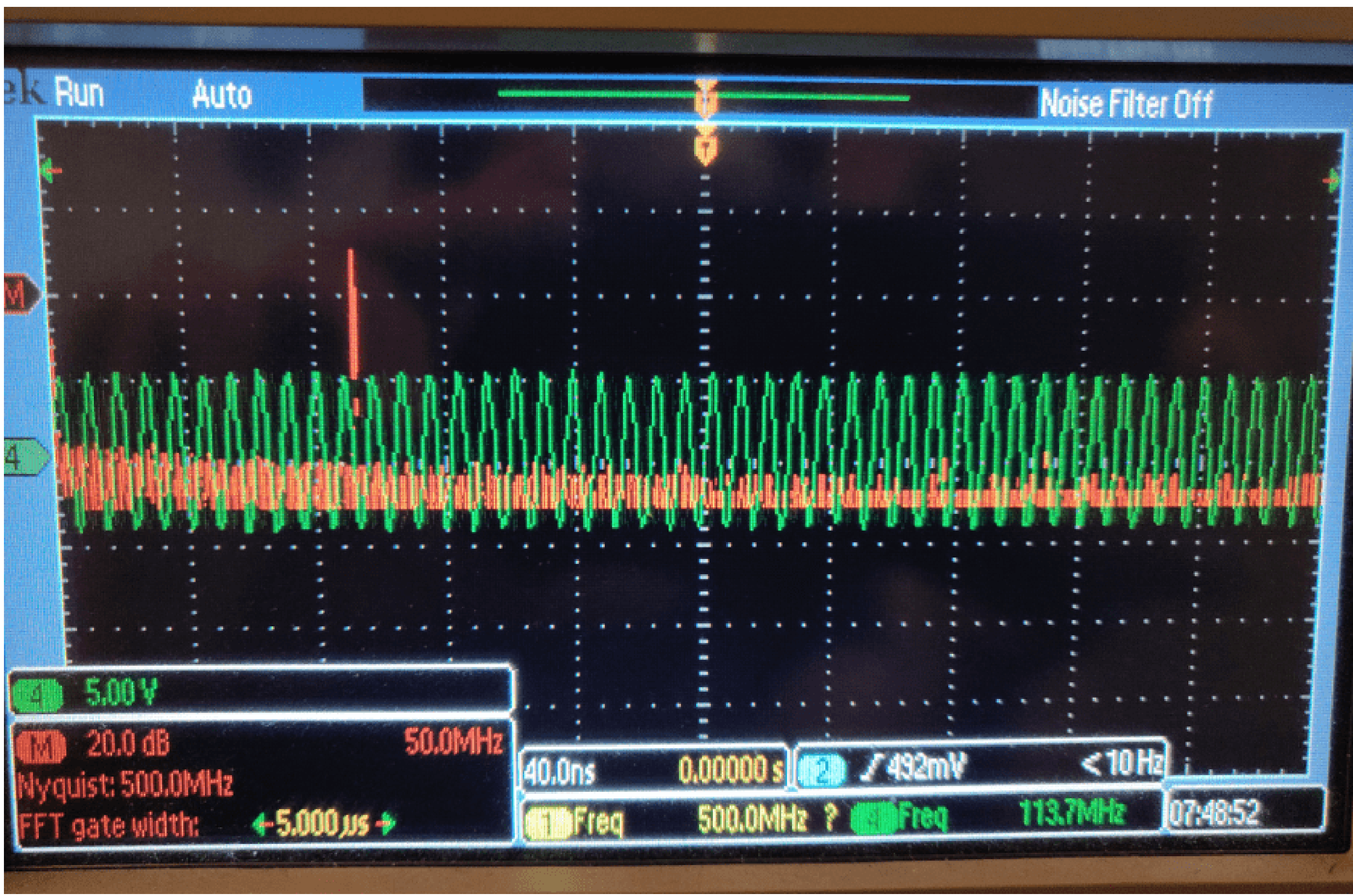
```
fpgaio_TxWaveformAndCapture
```

Verify that the data looks as expected by checking an oscilloscope.





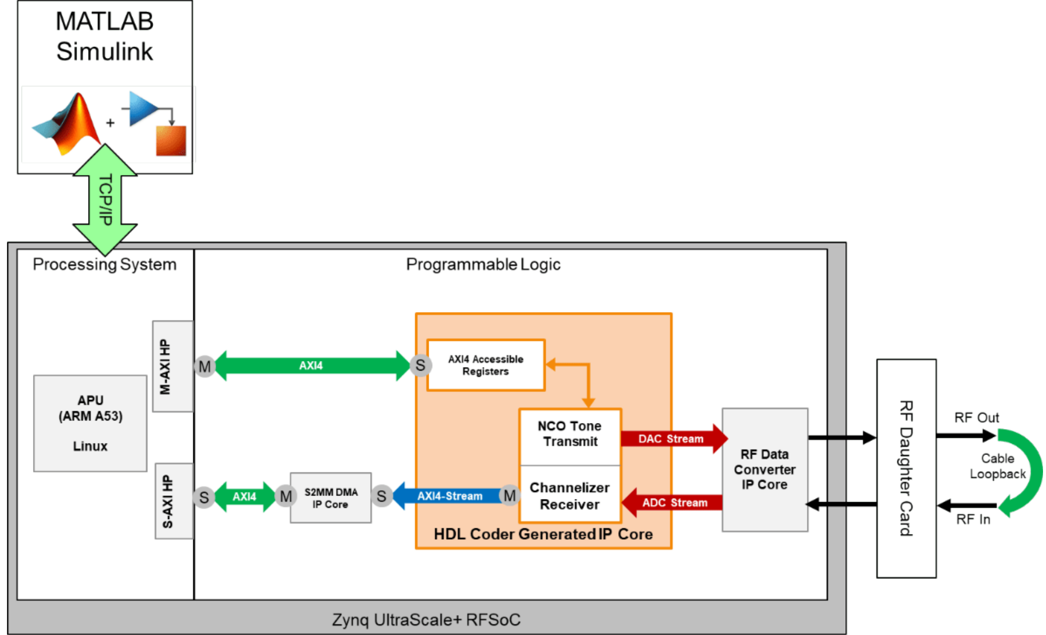
Using an oscilloscope, you can confirm the presence of the waveform by connecting to the correct channel on your RFSoC device and examining the output. For more details on which port to use, check the user guide on SMA cable connectivity. This figure shows an oscilloscope capture of a single tone value that is transmitted out of the DAC channel.



You can choose between the LTE waveform or single NCO tone by adjusting the parameters in the script `model_init.m`.

Polyphase Channelizer

This example shows how to use the HDL-optimized Channelizer block to process incoming analog-to-digital converter (ADC) samples and produce a spectrum that has 512 MHz of bandwidth. MATLAB® displays the resulting spectrum plot by using FPGA API functions over a TCP/IP connection. The channelizer data sent back is in limited bursts, which are triggered by an AXI4 register in a capture loop. The model also contains an interface to the digital-to-analog converter (DAC) by using a numerically-controlled oscillator (NCO) to drive a tone commanded at some frequency over AXI4.



Requirements

- Vivado® Design Suite with a supported version listed in “Supported EDA Tools and Hardware”
- Xilinx® Zynq® UltraScale+™ ZCU111 evaluation kit or Xilinx Zynq UltraScale+ ZCU216 evaluation kit
- HDL Coder™
- HDL Coder Support Package for Xilinx FPGA and SoC Devices

Open Example

Open the example project and copy the example files to a temporary directory.

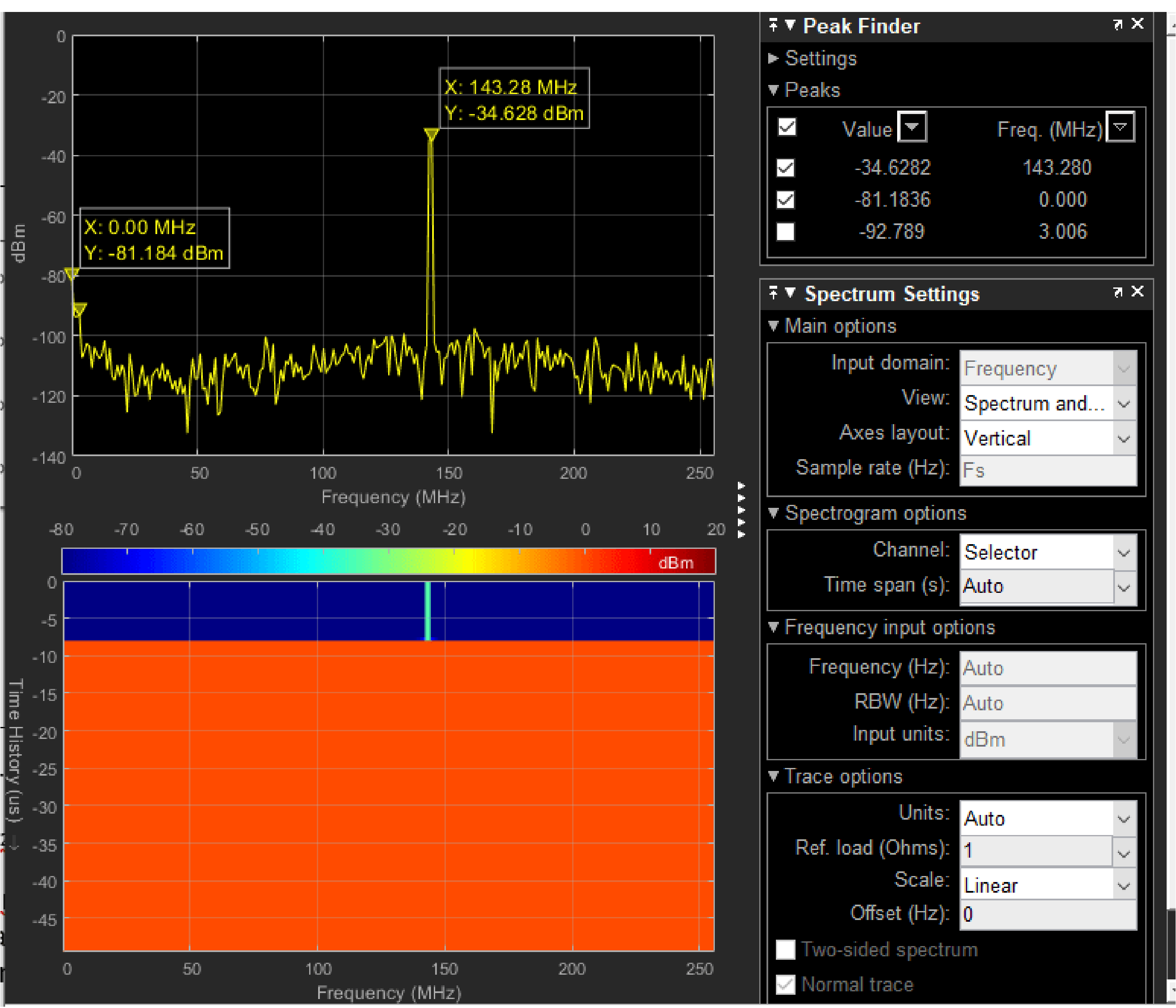
1. Navigate to the RFSoC root example directory of HDL Coder Support Package for Xilinx FPGA and SoC Devices by entering these commands at the MATLAB command prompt.

```
example_root = (hdlcoder_rfsoc_examples_root)
cd (example_root)
```

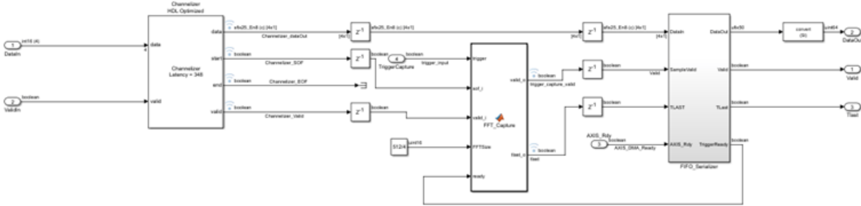
2. Copy all of the example files in the `PolyphaseChannelizer` folder to a temporary directory.

Simulate Channelizer Model

To understand the channelizer model in more detail, run the simulation of `rfsocChannelizer.slx` and view the power spectrum view and spectrogram.

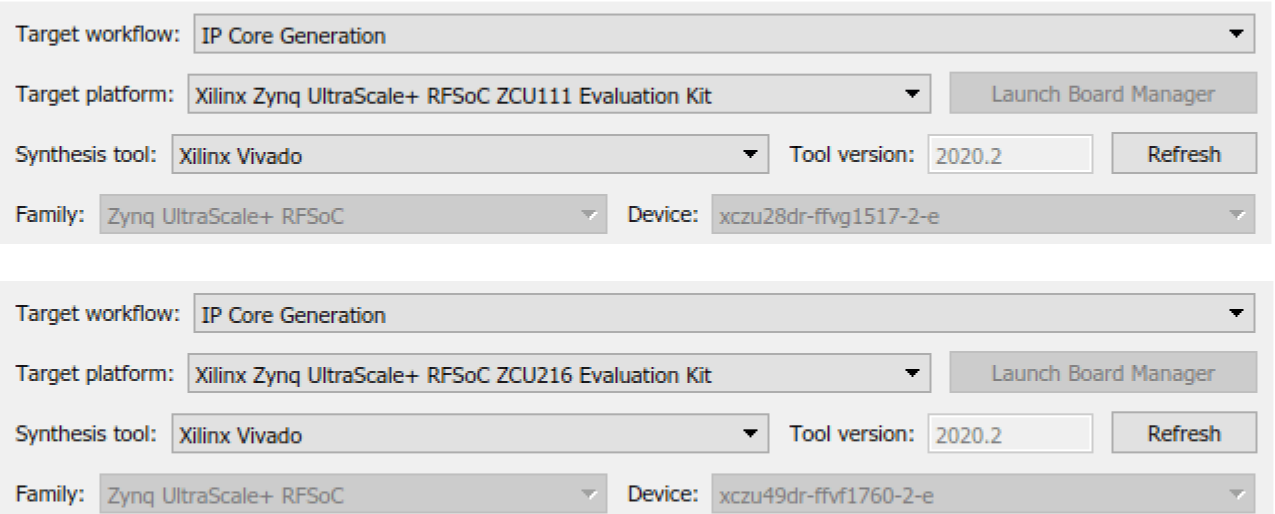


You can see the channelizer processing inside this block: `rfsocChannelizer/PolyphaseChannelizer/Channelizer Receive Processing`. The FFT_Capture logic captures data based off of a trigger condition called **TriggerCapture**, which is driven by an AXI4 register. The state machine does not wait for a trigger and processes the next available frame from the channelizer to the direct memory access (DMA). The channelizer block `Start-Of-Frame` and `Valid` control lines are used to help determine when to latch onto the frame.



Generate HDL and Synthesize Bitstream

Right-click the channelizer subsystem device under test (DUT). Select **HDL Code**, then click **HDL Workflow Advisor**. In step 1.1 of the HDL Workflow Advisor, select **Target platform** as `Xilinx Zynq Ultrascale+ RFSoC ZCU111 Evaluation Kit` or `Xilinx Zynq Ultrascale+ RFSoC ZCU216 Evaluation Kit`.



Before proceeding to the next step, set these reference design parameters to the indicated values.

- **AXI4-Stream DMA data width** to `64`
- **ADC sampling rate (MHz)** to `2048`
- **ADC decimation mode (xN)** to `4`
- **ADC samples per clock cycle** to `4`
- **ADC mixer type** to `Bypassed`
- **DAC sampling rate (MHz)** to `2048`
- **DAC interpolation mode (xN)** to `4`
- **DAC samples per clock cycle** to `4`
- **DAC mixer type** to `Bypassed`
- **ADC/DAC NCO mixer LO (GHz)** to `Disabled`
- **Enable multi-tile sync** to `false`

If you are using a ZCU216 board, additionally set the **DAC DUC mode** parameter to `Full DUC Nyquist (0-Fs/2)`.

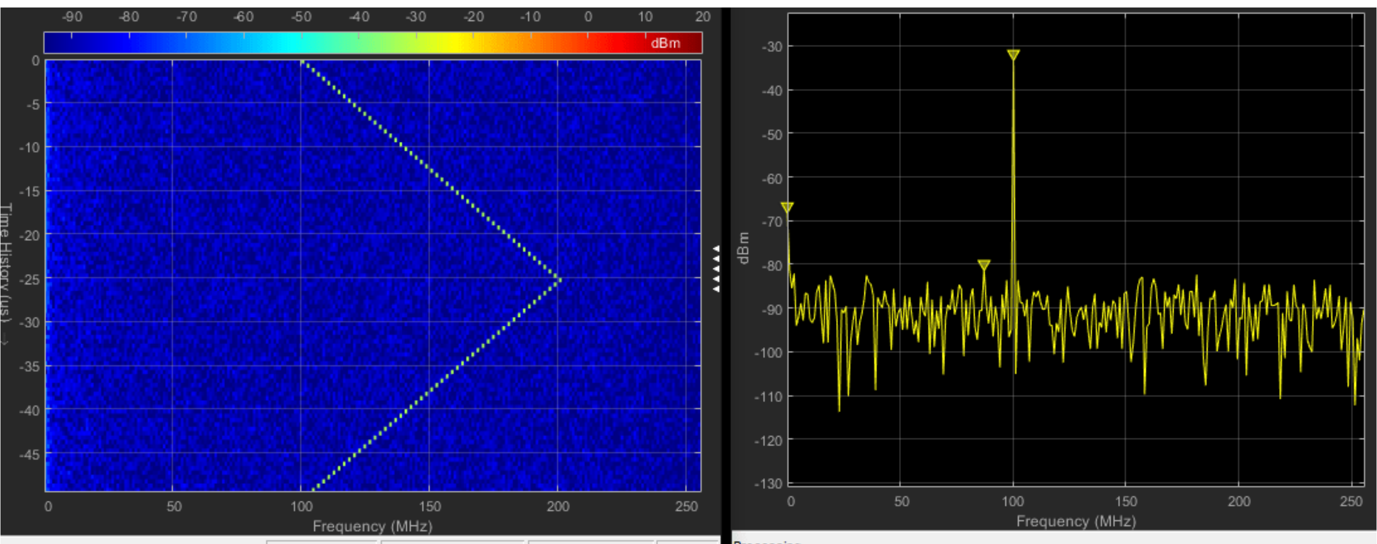
After the bitstream is done compiling, program the FPGA bit file if you are using an HDL Workflow Advisor script.

Channelizer Spectrum Data Capture

The FPGA returns the output of the channelizer, which is frequency domain data of the spectrum. The capture logic fetches a frame upon receiving the signal condition over AXI4. With the board programmed, run the script `hostIO_rfsocChannelizer_interface.m`.

```
% %%Data capture loop
% for ii=1:150
%    writePort(hFPGA,"TriggerCapture",false);
%    writePort(hFPGA,"TriggerCapture",true);
%    writePort(hFPGA,"TriggerCapture",false);
%
%    rd_data = readPort(hFPGA,"S2MM_Data");
%
%    idx = mod(ii,length(FrequencyArr)) + 1;
%    nco_tone = FrequencyArr(idx);
%    fprintf('Changing NCO Tx tone to %d MHz \n',nco_tone);
%    writePort(hFPGA,"AXI4_NCO_incr",uint16(incrScale*nco_tone*1e6));
%
%    fft_frame = reinterp_stream_data(rd_data);
%    spectrum = 20*log10(abs(fft_frame(1:256)));
%
%    scopeSpectrum(spectrum);
%    scopeSpectrogram(spectrum);
% end
%
% disp('Done frequency sweep');
```

Spectrogram and spectrum plots are displayed for 150 captures. The NCO is commanded with different tones from an array that is declared before the capture.



Multi-Tile Synchronization

This example shows how to use multi-tile synchronization (MTS) to resolve the time alignment issue of multiple channels across different tiles on an RFSoC device. The RFSoC has built-in features that enforce the time alignment for samples of multiple channels across different tiles. Channels in a tile alone are aligned in time but a guarantee of alignment with another channel from a different tile does not exist. You can enable multi-tile synchronization (MTS) to correct for this issue by first measuring latency across different tiles and then applying sample delays to ensure samples align correctly.

MTS for Xilinx® Zynq® UltraScale+™ RFSoC ZCU111 and Xilinx Zynq UltraScale+ RFSoC ZCU216 evaluation kits requires that you chose specific sample rates that are governed by SYSREF signals from an external clock. According to *Xilinx datasheet PG269*, the SYSREF frequency must meet these requirements.

- If synchronizing RF-ADC and RF-DAC tiles with the different sample frequencies, the frequency must be an integer submultiple of: GCD(DAC_Sample_Rate/16, ADC_Sample_Rate/16).
- SYSREF must also be an integer submultiple of all PL clocks that sample it. This is to ensure the periodic SYSREF is always sampled synchronously.
- Less than 10 MHz.

In the context of the ZCU111 and ZCU216 boards, the reference clock must be an integer multiple of the SYSREF frequency. For the ZCU111 board, the default SYSREF frequency produced by the LMK is 7.68 MHz. To meet the requirements, choose a sampling rate from the available provided frequencies from the LMK that is a multiple of 7.68 MHz.

In many designs, this reference clock is chosen in such a way to satisfy this requirement. For example, 245.76 MHz is a common choice when you use a ZCU216 board. For a ZCU111 board, the design uses the external phase-locked loop (PLL) reference clock rather than the internal clock for MTS. The configuration files and System object™ scripts that are generated during the HDL Workflow Advisor step complete this process.

Requirements

- Vivado® Design Suite with a supported version listed in “Supported EDA Tools and Hardware”
- Xilinx Zynq UltraScale+ ZCU111 evaluation kit or Xilinx Zynq UltraScale+ ZCU216 evaluation kit
- HDL Coder
- HDL Coder Support Package for Xilinx FPGA and SoC Devices

Open Example

Open the example project and copy the example files to a temporary directory.

1. Navigate to the RFSoC root example directory of HDL Coder Support Package for Xilinx FPGA and SoC Devices by entering these commands at the MATLAB® command prompt.

```
example_root = (hdlcoder_rfsoc_examples_root)
cd (example_root)
```

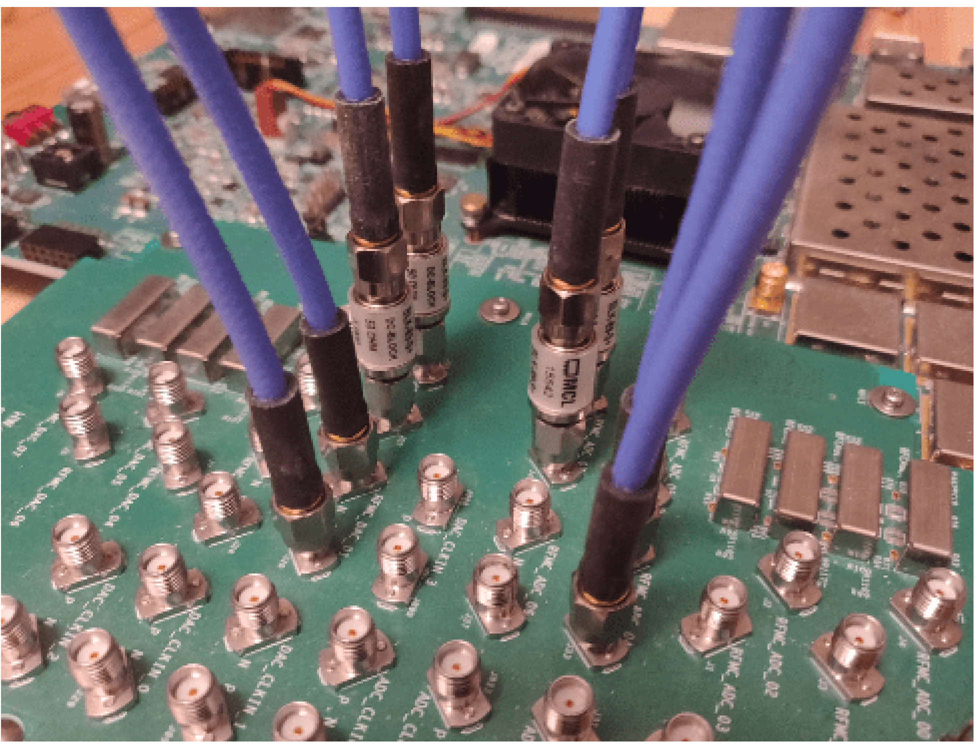
2. Copy all of the example files in the `MTS` folder to a temporary directory.

MTS Cable Setup

This example provides two MTS examples, one for a ZCU111 board and one for a ZCU216 board. These examples show that analog-to-digital converter (ADC) channel samples from different tiles are aligned after you apply MTS. By comparing one channel with the other, visual inspection can be performed.

When using Multi-Tile Synchronization, use ADC and DAC channels that are connected to the differential SMA ports on the XM500. The single-ended baluns introduce amplitude and phase offset between channels even when MTS is enabled.

For comparing channels, the ZCU111 example cable setup for the XM500 balun card is configured so that it compares two channels from differing tiles. These two figures show the cable setup.



In these figures:

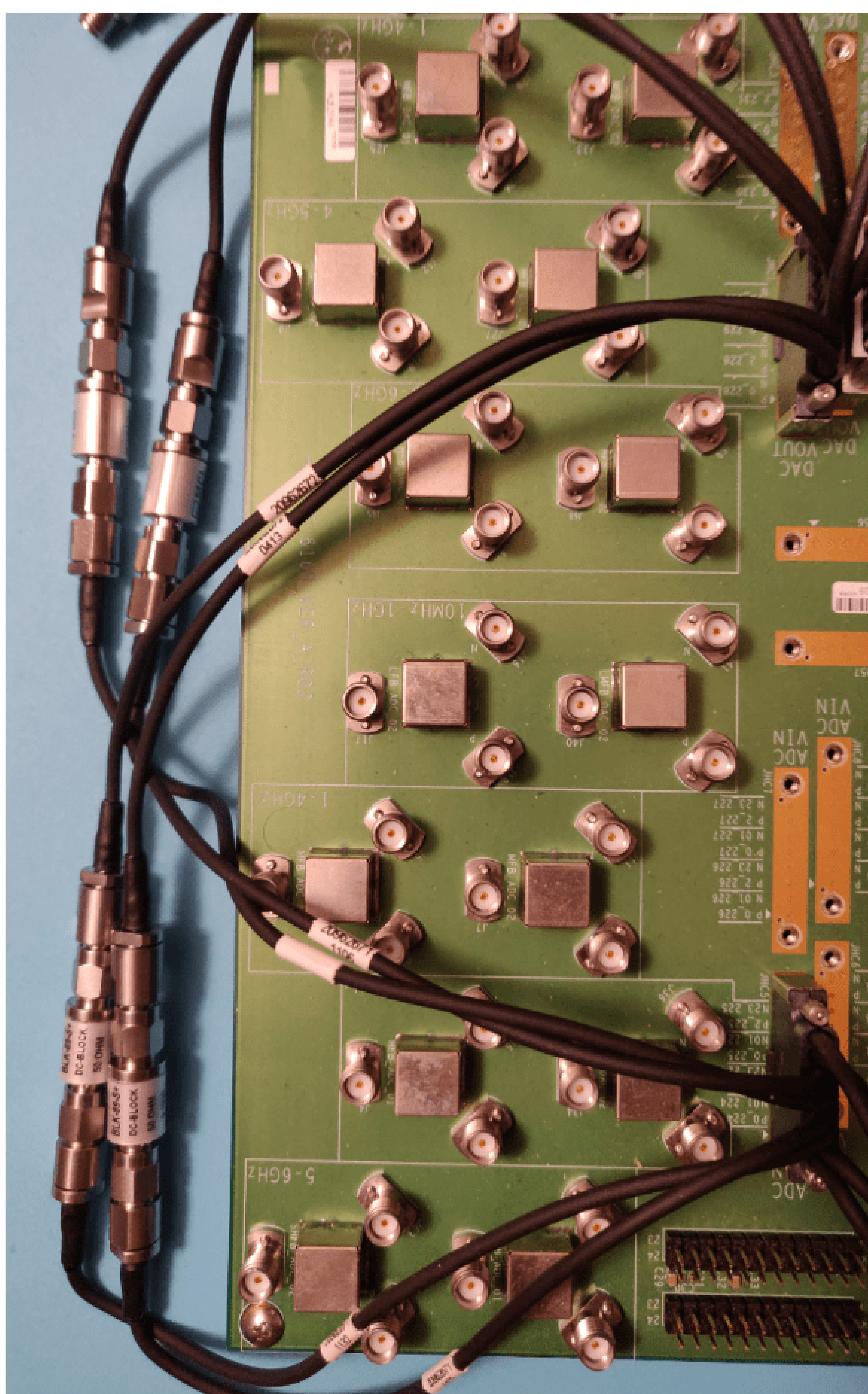
- DAC 00 connects to ADC 04.
- DAC 01 connects to ADC 07.

In terms of tile connections, the setup that these figures show represents 0-based indexing.

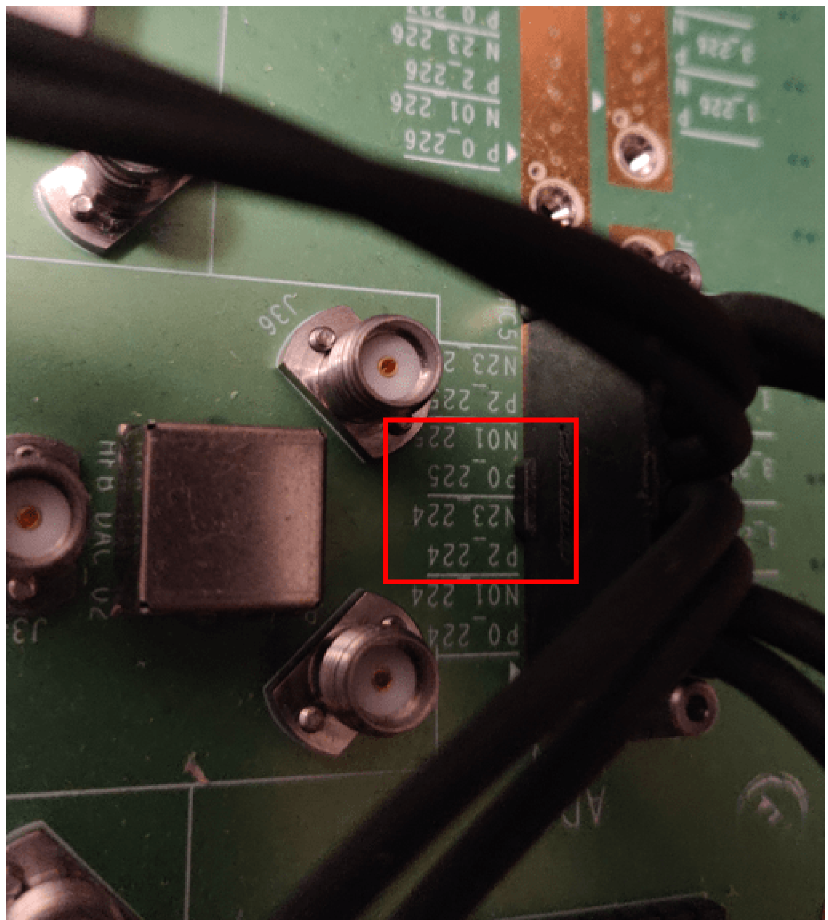
- DAC Tile 0 Channel 0 connects to ADC Tile 2 Channel 0.
- DAC Tile 0 Channel 1 connects to ADC Tile 3 Channel 2.

Differential cables that have DC blockers are used to make use of the differential ports. The cables use a data path that does not have an analog RF cage filter, which can impose phase delays across different channels. Because the purpose of this test is to measure sample alignment, avoiding things that can potentially alter results, such as a mismatch in cable types or filters, is a best practice.

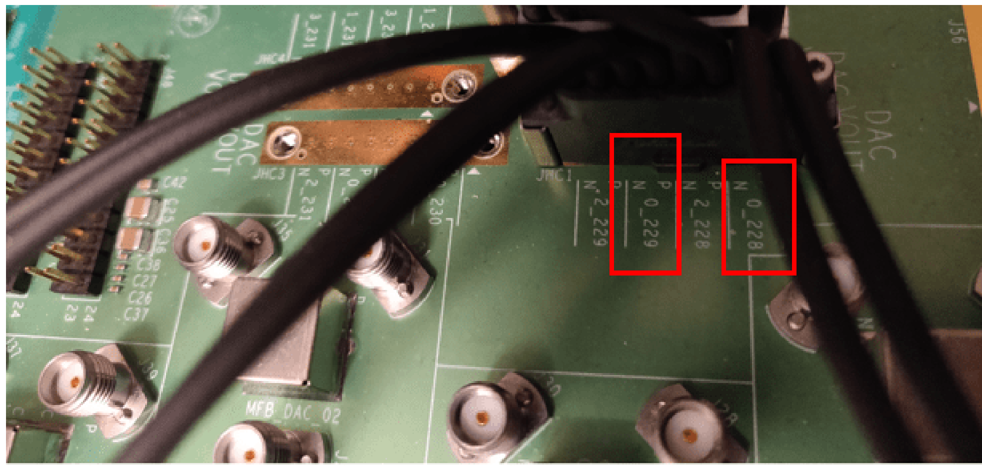
For the ZCU216 board, a similar setup is used with differential SMA connections by using the XM655 balun card. This figure shows the XM655 board with a differential cable.



ADC Setup



DAC Setup

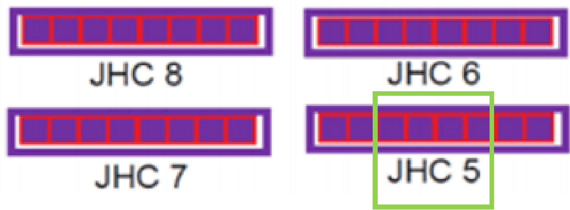


In the DAC setup figure:

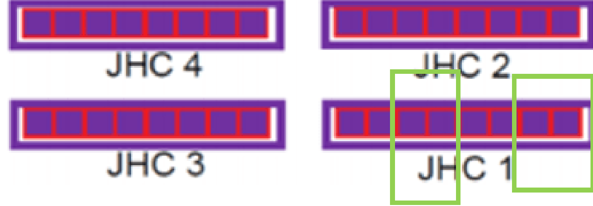
- DAC P/N 0_228 connects to ADC P/N 02_224.
- DAC P/N 0_229 connects to ADC P/N 00_225.

The next two figures show a schematic that indicates which differential connectors this example uses.

ADC



DAC



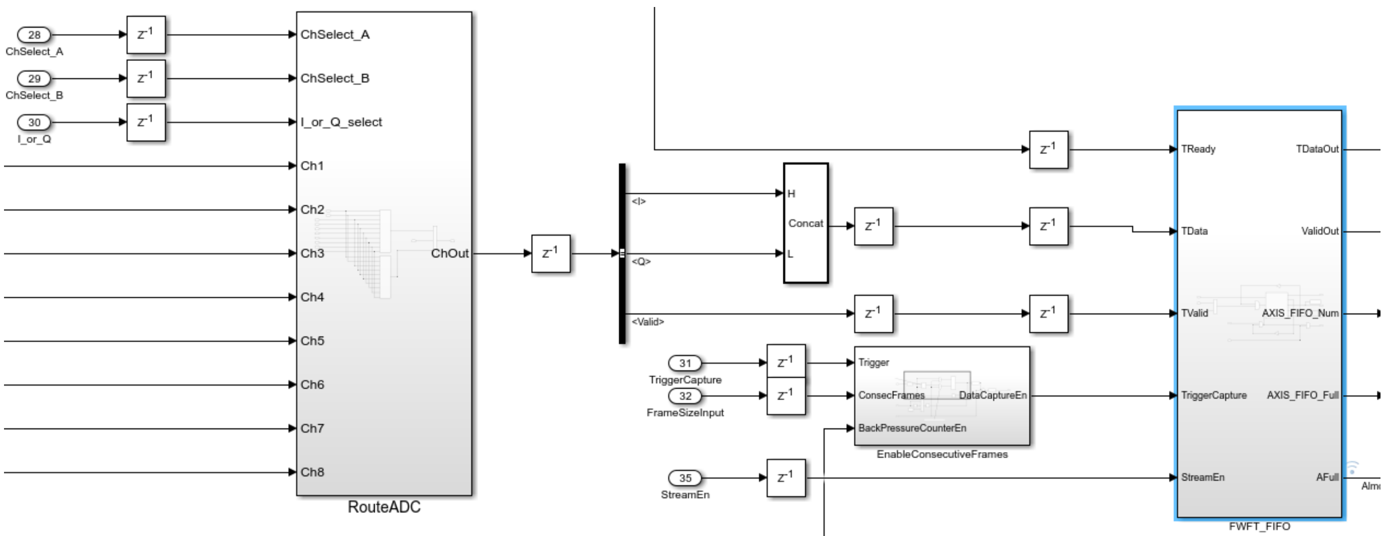
In terms of tile connections, the setup that these figures show represents 0-based indexing.

- DAC Tile 0 Channel 0 connects to ADC Tile 0 Channel 2.
- DAC Tile 1 Channel 0 connects to ADC Tile 1 Channel 2.

For more information on cable setups, see the Xilinx documentation.

HDL Model and Bitstream Synthesis

Two HDL models (`rfsoc_zcu216_MTS_iq_HDL.slx` and `rfsoc_zcu111_MTS_iq_HDL.slx` located in the example root) are provided for the ZCU216 and ZCU111 boards. The models take in two channels for data capture selected by an AXI4 register for routing. Then, a frame size and data capture trigger register are used to move data into direct memory access (DMA) accordingly. An additional mux is added to pick between inphase (I) or quadrature (Q) when comparing the channels. Because the design runs at four samples per clock for in-phase and quadrature (IQ), a limited amount of data width is available for moving data across.



To synthesize HDL, right-click the subsystem. Select **HDL Code**, then click **HDL Workflow Advisor**. In step 1.1 of the HDL Workflow Advisor, select **Target platform** as `Xilinx Zynq Ultrascale+ RFSoC ZCU111 Evaluation Kit` or `Xilinx Zynq Ultrascale+ RFSoC ZCU216 Evaluation Kit`.

In step 1.2, set these reference design parameters to the indicated values.

- **AXI4-Stream DMA data width** to `128`
- **ADC sampling rate (MHz)** to `1966.08`
- **ADC decimation mode (xN)** to `4`
- **ADC samples per clock cycle** to `4`
- **ADC mixer type** to `Fine`
- **DAC sampling rate (MHz)** to `1966.08`
- **DAC interpolation mode (xN)** to `4`
- **DAC samples per clock cycle** to `4`
- **DAC mixer type** to `Fine`
- **ADC/DAC NCO mixer LO (GHz)** to `0.5`
- **Enable multi-tile sync** to `true`

If you are using a ZCU216 board, additionally set the **DAC DUC mode** parameter to `Full DUC Nyquist (0-Fs/2)`.

Afterward, build the bitstream and then program the board.

Run-Time Testing of MTS Channel Alignment

After you program the board, it reboots and initializes with MTS applied when Linux® loads. To configure the RFSoC with various properties and settings, use a configuration CFG file.

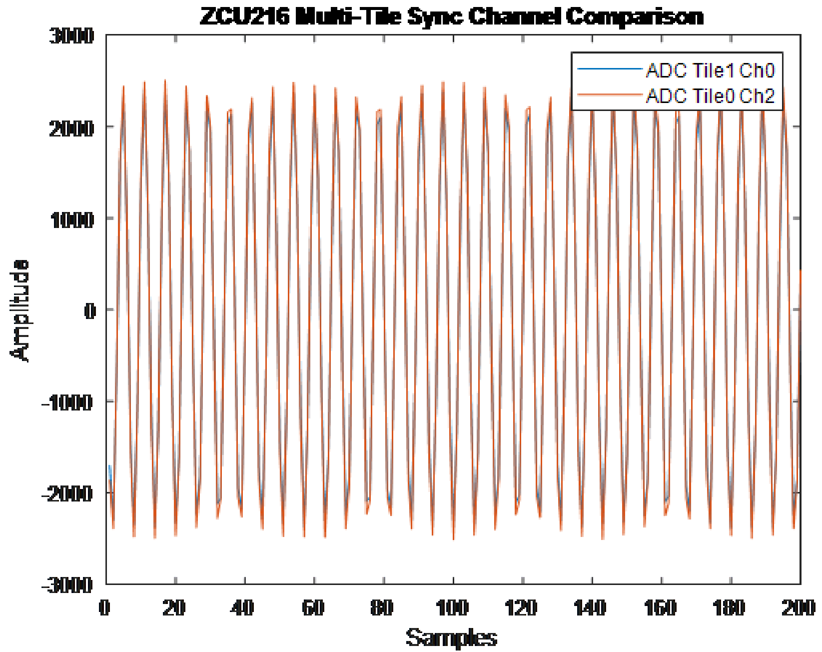
To check channel alignment, data capture scripts are provided for both ZCU216 and ZCU111 boards. If the SMA attachment cards match the setup described in the previous sections of this example, run the script. The results show near-perfect alignment of the channels.

Run whichever script matches the board that you are testing against.

```
HostIO_rfsoc_zcu216_MTS_iq_HDL_interface.m
```

```
HostIO_rfsoc_zcu111_MTS_iq_HDL_interface.m
```

A single plot shows the result of the data capture of two channels.

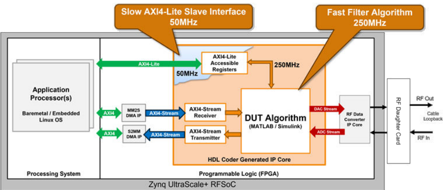


When you use MTS, avoid changing the digital local oscillator (LO) of the RFSoC during MTS. The LO for each channel might not be aligned in time, which can impact alignment. The RFSoC provides ways of dealing with this issue by synchronizing the reset condition on all channels based on tile events. By setting tile events to listen to a SYSREF signal, alignment can be achieved when you use the mixer during an MTS routine. The SYSREF capture must be disabled first, then the change to the LO is applied, and then an MTS calibration is done again. To see an example of this process, run the script `ZCU216_ChangeLO.m` or `ZCU111_ChangeLO.m`.

Enable Clock Domain Crossing on AXI4-Lite Interfaces

When you generate an IP Core Generation, if you want to run the register interface at a slower clock frequency than your high-frequency design under test (DUT) algorithm, you can select the **Enable clock domain crossing on AXI4-Lite registers** setting in the **Generate RTL Code and IP Core** task in the HDL Workflow Advisor.

When you enable clock domain crossing (CDC), HDL Coder inserts CDC and back pressure logic that crosses the signals from the AXI4-Lite clock domain to the IP core clock domain, or from the IP core clock domain to the AXI4-Lite clock domain.



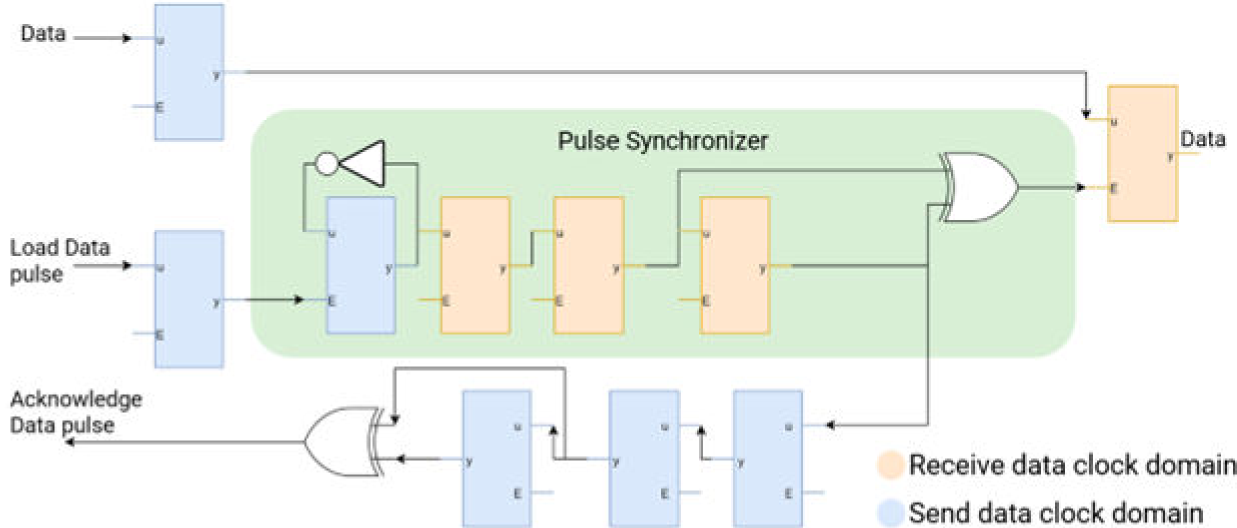
This image shows the generated IP core. When you enable CDC, you can connect different clock signals to the IP core clock `IPCORE_CLK` and AXI4-Lite interface clock `AXI4_Lite_ACLK`. When you disable CDC, you must connect the same clock signal to the IP core clock and AXI4-Lite interface clock. For more information on how to connect the clock signals, see “Custom IP Core Generation” on page 39-17.

AXI4_Lite
IPCORE_CLK
IPCORE_RESETN
AXI4_Lite_ACLK
AXI4_Lite_ARESETN
GPIO_LED[7:0]

When you export the HDL Workflow Advisor run to a script, you can observe the saved CDC setting as a `hdlset_param` command. For example, if you enable the setting on the DUT subsystem in the `hdlcoder_led_vector` model, you can access the CDC setting by entering:

```
hdlset_param('hdlcoder_led_vector/DUT',...
ClockDomainCrossingOnregisterInterface,'on');
```

When you enable CDC, the multi-bit data synchronization is achieved by using a pulse synchronizer and two multi-bit registers. The address decoder repeatedly uses this implementation to cross the signals to and from each clock domain. This image shows the architecture that the CDC implementation uses.



The additional CDC logic introduces latency and back pressure on the AXI4-Lite interface due to the added latency to the read and write operations. However, because the ARM processor that accesses the AXI registers on the SoC boards has a larger latency, the introduced latency has minimal impact. In addition, there are no dependencies between the clock frequency for the DUT IP core and the AXI4-Lite clock frequency.

The synthesis tool requires timing constraints to analyze the paths in the CDC logic. HDL Coder generates these constraints and outputs them as a Xilinx Design Constraints (XDC) file and packages the file with the generated IP core. This code snippet presents the XDC file generated for use with the Xilinx Vivado environment.

```
# AXI4-Lite CDC constraints:
set falsepathregcells [get_cells -hier -filter {cdc_info=="synchronizer_false_path"}]
set falsepathregpins [get_pins -of_objects $falsepathregcells -filter {REF_PIN_NAME=="D"}]
set_false_path -to $falsepathregpins
set startreglist_axitoip [get_cells -hier -filter {cdc_info=="max_delay_startpoint_axi_to_ip"}]
set endreglist_axitoip [get_cells -hier -filter {cdc_info=="max_delay_endpoint_axi_to_ip"}]
set reg_axitoip [get_cells -hier -filter {cdc_info=="max_delay_clk_axi_to_ip"}]
set clk_axitoip [get_clocks -of $reg_axitoip]
set clk_period_axitoip [get_property PERIOD $clk_axitoip]
set_max_delay -from $startreglist_axitoip -to $endreglist_axitoip -datapath_only $clk_period_axitoip -quiet
set startreglist_iptoaxi [get_cells -hier -filter {cdc_info=="max_delay_startpoint_ip_to_axi"}]
set endreglist_iptoaxi [get_cells -hier -filter {cdc_info=="max_delay_endpoint_ip_to_axi"}]
set reg_iptoaxi [get_cells -hier -filter {cdc_info=="max_delay_clk_ip_to_axi"}]
set clk_iptoaxi [get_clocks -of $reg_iptoaxi]
set clk_period_iptoaxi [get_property PERIOD $clk_iptoaxi]
set_max_delay -from $startreglist_iptoaxi -to $endreglist_iptoaxi -datapath_only $clk_period_iptoaxi -quiet
```

The HDL Coder generated constraints add maximum delay and false path constraints to sets of registers that are part of the CDC synchronizing circuits. HDL Coder finds these registers by searching the synthesis netlist for the `cdc_info ATTRIBUTE` keyword. This approach is more reliable than just looking for patterns in module or signal names. This code snippet shows an example of the use of the `cdc_info` keyword in one of the address decoder modules netlists:

```
...
ATTRIBUTE cdc_info : string;
…
ATTRIBUTE cdc_info OF s_level_2FlopSync1 : SIGNAL IS "synchronizer_false_path";
...
```

You must set your **Synthesis Tool** to `Xilinx Vivado` to use CDC for the AXI4-Lite interface.

See Also

Related Examples

- “Use Clock Domain Crossing to Run DUT Algorithm and AXI4-Lite Interface at Different Frequencies” on page 40-442

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 39-9
- “Custom IP Core Generation” on page 39-17
- “Model Design for AXI4 Slave Interface Generation” on page 40-3

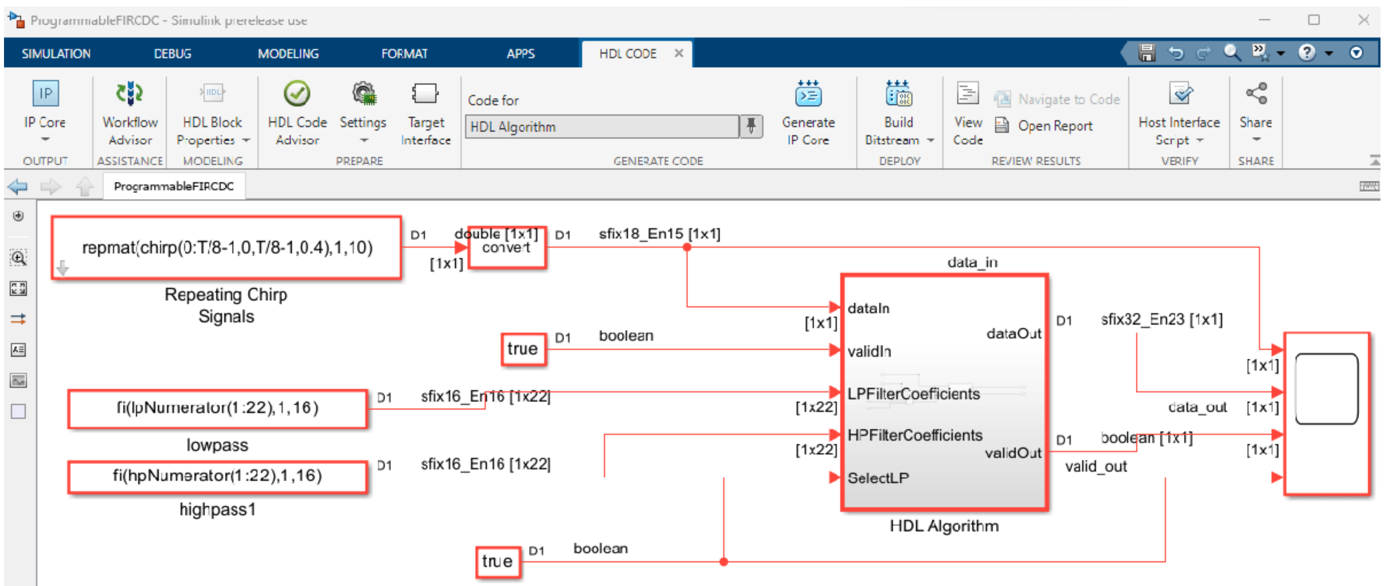
Use Clock Domain Crossing to Run DUT Algorithm and AXI4-Lite Interface at Different Frequencies

In many applications, high-speed data processing is essential. However, configuring these designs through a register interface, such as updating coefficients in an FIR filter, does not require the same high performance. When the device under test (DUT) algorithm and the register interface operate at the same high frequency, you may find it harder to meet timing closure.

This example demonstrates how to use clock domain crossing (CDC) to meet timing requirements when interfacing a high-speed DUT algorithm with a slower AXI4-Lite register interface using HDL Coder™. CDC is a critical aspect of FPGA design, especially when different parts of a system operate at disparate frequencies. By incorporating CDC and back-pressure logic, HDL Coder can insert the necessary logic for communication between clock domains and mitigate the risk of data corruption and timing violations. The example demonstrates how adjusting the register interface to a slower frequency and enabling CDC can allow the design to achieve timing closure.

Generate HDL IP Core with AXI4-Lite Interface

The example model `ProgrammableFIRCDC` has a subsystem named `HDL Algorithm` that has a control signal that switches between two sets of coefficients. The `HDL Algorithm` subsystem includes a Discrete FIR Filter block, and the coefficients used in the filter are defined by the vector inputs.



To enable simulation and testing, this model contains two FIR filters, one with a low-pass response and one with a complementary high-pass response. Both filters are odd-symmetric and have 43 taps. The design specifications for both filters are:

```
Fpass = 0.45; % Passband frequency
Fstop = 0.55; % Stopband frequency
Apass = 1;    % Passband attenuation (dB)
Astop = 60;   % Stopband attenuation (dB)

f = fdesign.lowpass('Fp,Fst,Ap,Ast',Fpass,Fstop,Apass,Astop);
```

```
Hlp = design(f,'equiripple','FilterStructure','dffir'); % Lowpass
Hhp = firlp2hp(Hlp);                                    % Highpass

hpNumerator = Hlp.Numerator;
lpNumerator = Hhp.Numerator;
```

To generate an IP core from the `ProgrammableFIRCDC/HDL Algorithm` subsystem, prepare the model by using the configuration parameters, configure your design using the IP Core editor, and generate the bitstream using the **HDL Code** tab of the Simulink Toolstrip:

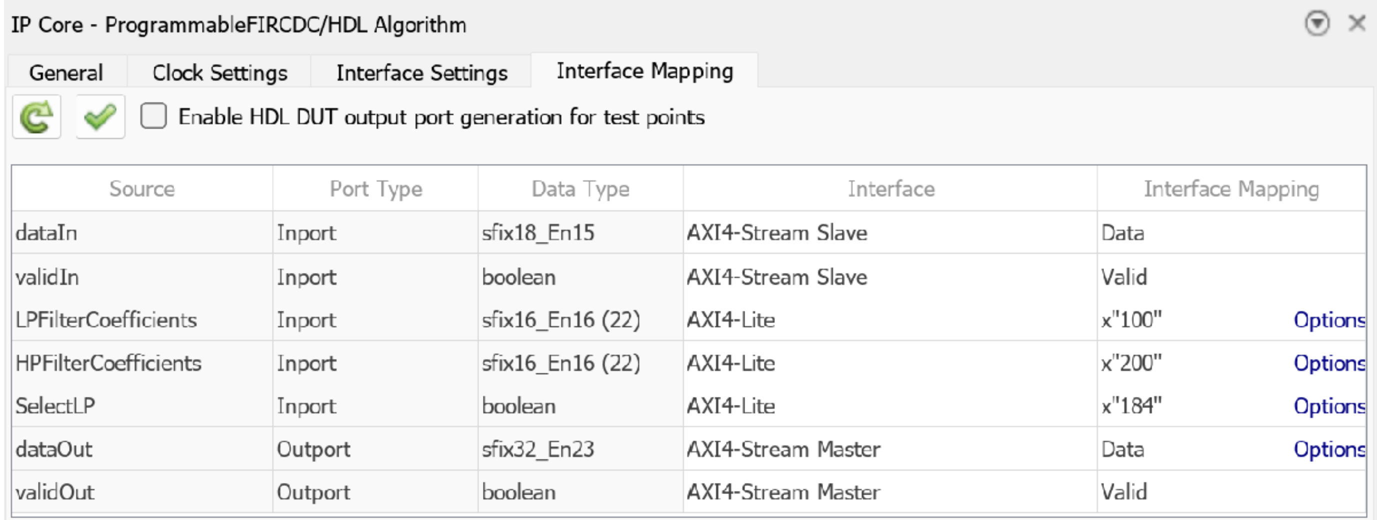
1. In the **Apps** tab, click **HDL Coder**. In the **HDL Code** tab, in the **Output** section, set the drop-down button to **IP Core**.
2. Select the `HDL Algorithm` subsystem which is the DUT for this example. In the **HDL Code** tab, ensure that **Code for** is set to this subsystem. To remember the selection, you can pin this option.
3. Open the **HDL Code Generation > Target** tab of Configuration Parameters dialog box by clicking the **Settings** button.
4. Set the **Target Platform** parameter to `Zedboard`. If this option does not appear, select `Get more` to open the Support Package Installer. In the Support Package Installer, select `HDL Coder Support Package for Xilinx FPGA and SoC Devices` and follow the instructions to complete the installation.
5. Set the **Synthesis Tool** to `Xilinx Vivado`.
6. Set the **Reference Design** parameter to `Default system with AXI4-Stream Interface` and **Target Frequency** to `129.`
7. Click **OK** to save your updated settings.

The `Default system with AXI4-Stream Interface` reference design is configured to use the same clock for the DUT and the AXI4 register interface. The next section demonstrates that using the same high clock frequency for the whole design does not achieve timing closure.

Configure Design and Target Interface

Configure the design to map to the target hardware by mapping the DUT ports to the IP core target hardware and setting the DUT-level IP core options. In this example, map the DUT ports **LPFilterCoefficients**, **HPFilterCoefficients**, and **SelectLP** to `AXI4-Lite` and map the **dataIn**, **ValidIn**, **dataOut**, and **ValidOut** interfaces to `AXI4-Stream`.

1. In Simulink, in the HDL Code tab, click **Target Interface** to open the IP Core editor.
2. In the **IP Core** pane, select **Interface Settings**. Ensure that **Enable readback on the AXI4 registers** is selected and set the **AXI4 Slave port to pipeline register ratio** to `off`.
3. Select the **Interface Mapping** tab to map each DUT port to one of the IP core target interfaces. The generated design can then communicate with the rest of the hardware system when it is deployed. If no mapping table appears, click the Reload IP core settings and interface mapping table from model button to compile the model and repopulate the DUT ports and their data types.
4. For the DUT ports **LPFilterCoefficients**, **HPFilterCoefficients**, and **SelectLP**, set the cells in the **Interface** column to `AXI4-Lite.`
5. For the DUT ports **dataIn**, **ValidIn**, **dataOut**, and **ValidOut,** set the **Interface** column to `AXI4-Stream`.
6. Validate your settings by clicking the Validate IP core settings and interface mapping button .



Source	Port Type	Data Type	Interface	Interface Mapping	
dataIn	Inport	sfix18_En15	AXI4-Stream Slave	Data	
validIn	Inport	boolean	AXI4-Stream Slave	Valid	
LPFilterCoefficients	Inport	sfix16_En16 (22)	AXI4-Lite	x"100"	Options
HPFilterCoefficients	Inport	sfix16_En16 (22)	AXI4-Lite	x"200"	Options
SelectLP	Inport	boolean	AXI4-Lite	x"184"	Options
dataOut	Outport	sfix32_En23	AXI4-Stream Master	Data	Options
validOut	Outport	boolean	AXI4-Stream Master	Valid	

After you configure the IP core settings and mappings, you can generate the bitstream. To generate the bitstream file, in the Simulink Toolstrip, in the **HDL Code** tab, click **Build Bitstream** and wait until the synthesis tool runs in the external window.

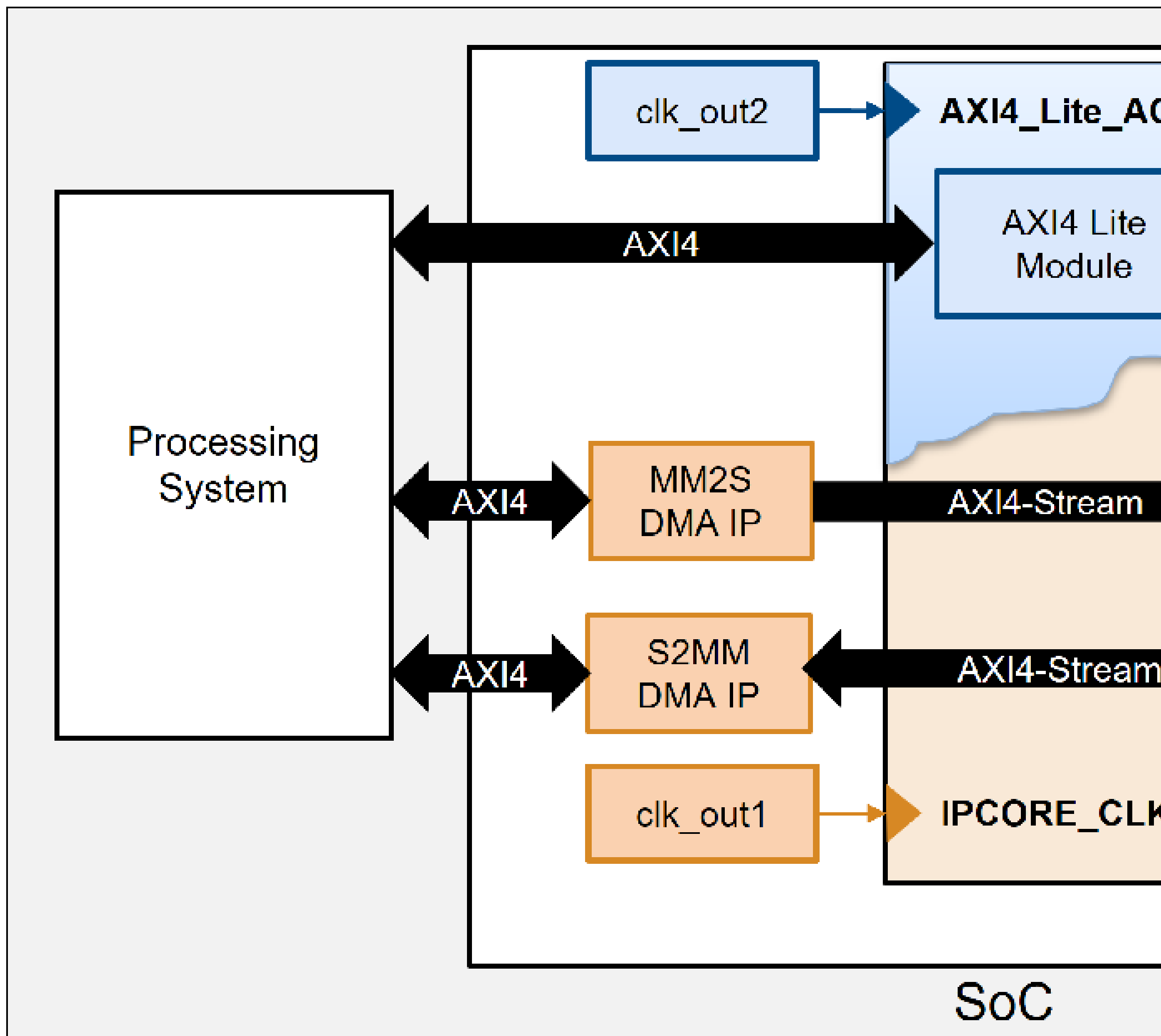
The generated HDL code fails to meet the 129 MHz timing requirement for the clock. The timing requirement is 7.752 ns, or 1/129MHz. In this example, the synthesis tool reported a negative slack, which indicates a timing violation.

```
C:\WINDOWS\SYSTEM32\cmd
INFO: [Netlist 29-17] Analyzing 208 Unisim elements for replacement
INFO: [Netlist 29-28] Unisim Transformation completed in 0 CPU seconds
INFO: [Project 1-479] Netlist was created with Vivado 2023.1.2
INFO: [Project 1-570] Preparing netlist for logic optimization
INFO: [Timing 38-478] Restoring timing data from binary archive.
INFO: [Timing 38-479] Binary timing data restore complete.
INFO: [Project 1-856] Restoring constraints from binary archive.
INFO: [Project 1-853] Binary constraint restore complete.
Reading XDEF placement.
Reading placer database...
Reading XDEF routing.
Read XDEF Files: Time (s): cpu = 00:00:02 ; elapsed = 00:00:03 . Memory (MB): peak = 3344.117 ; gain = 3.328
Restored from archive | CPU: 2.000000 secs | Memory: 0.000000 MB |
Finished XDEF File Restore: Time (s): cpu = 00:00:02 ; elapsed = 00:00:03 . Memory (MB): peak = 3344.117 ; gain = 3.328
Netlist sorting complete. Time (s): cpu = 00:00:01 ; elapsed = 00:00:00.011 . Memory (MB): peak = 3344.117 ; gain = 0.000
INFO: [Project 1-111] Unisim Transformation Summary:
  A total of 41 instances were transformed.
  RAM16X1D => RAM32X1D (RAMD32(x2)): 1 instance
  RAM32M => RAM32M (RAMD32(x6), RAMS32(x2)): 12 instances
  RAM32X1D => RAM32X1D (RAMD32(x2)): 5 instances
  SRLC32E => SRL16E: 23 instances

open_run: Time (s): cpu = 00:00:17 ; elapsed = 00:00:48 . Memory (MB): peak = 3344.117 ; gain = 910.523

 The worst slack is: -0.912 ns
 ERROR- Timing constraints NOT met!

INFO: [Common 17-206] Exiting Vivado at Thu Dec 14 09:12:07 2023...
```

To meet the timing requirements for this design, you can use a slower clock for the AXI4-Lite interface and maintain the fast frequency for the DUT algorithm.

Integrate HDL IP Core into Reference Design with Two Different Frequencies

The FIR filter coefficients do not need to be updated at the same frequency as the data signal being filtered. Consequently, the AXI4-Lite interface can operate at a slower clock speed. In the reference design, connect the AXI4-Lite interface clock and the IP core clock to different clock sources, as shown in this image:



For this example, you use a custom reference design that connects the AXI4-Lite interface clock to a fixed 50 MHz clock, while the IP core clock is set to the value specified for the **Target Frequency** in the Configuration Parameters window. Specify the clock connection for the AXI4-Lite module when you add the register interface in the reference design `plugin_rd.m` file. The custom reference design in this example derives two different clocks from the same clock wizard. Because the first

output, `clk_out1`, connects to the `IPCORE_CLK`, you can specify this connection when you add the clock interface. The second clock output, `clk_out2`, corresponds to a fixed 50 MHz clock which connects to the `AXI4_Lite_ACLK`, you can specify this connection when you add the AXI4 register interface. This code snippet specifies the clock and reset connections for the `Default CDC system with AXI4-Stream Interface` reference design.

```
%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',     'core_clkwiz/clk_out1', ...
    'ResetConnection',     'sys_core_rstgen/peripheral_aresetn',...
    'DefaultFrequencyMHz', 70,...
    'MinFrequencyMHz',     5,...
    'MaxFrequencyMHz',     500,...
    'ClockModuleInstance', 'core_clkwiz',...
    'ClockNumber',         1);

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    ... % Hardware (FPGA) properties
    'InterfaceConnection', 'axi_cpu_interconnect/M00_AXI', ...
    'BaseAddress',         '0x400D0000', ...
    'MasterAddressSpace',  'sys_cpu/Data', ...
     ...% Clock Connection
    'ClockConnection', 'core_clkwiz/clk_out2', ...
    'ResetConnection', 'sys_100m_rstgen1/peripheral_aresetn', ...
    ... % Software (Processor) properties
    'HasProcessorConnection', true, ...
    'DeviceTreeBusNode', '&fpga_axi');
```

To inspect the reference design files, open the HDL coder support package examples root directory. Open this directory by using this command:

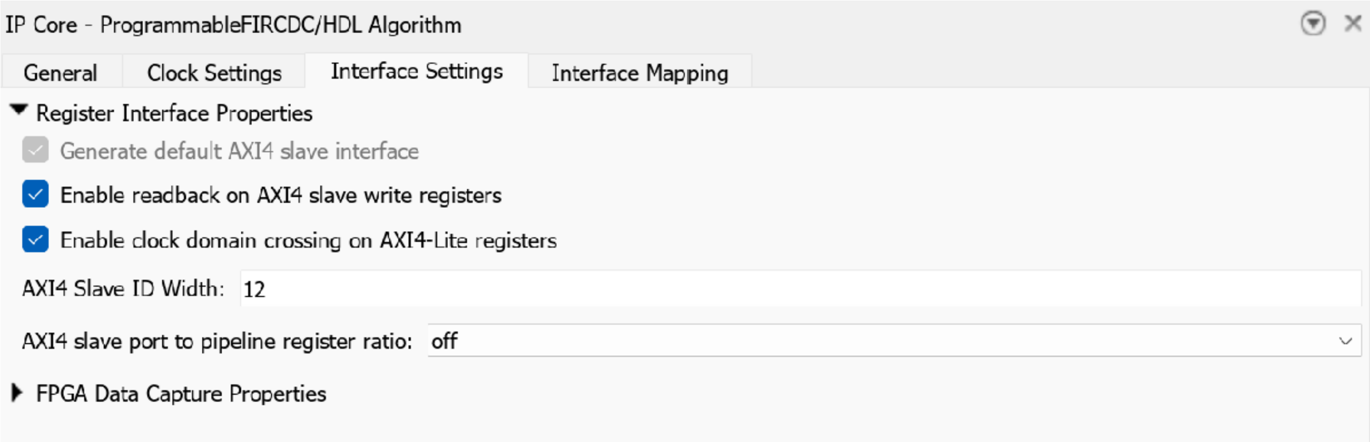
```
cd(fullfile(hdlcoder_amd_examples_root,'ZedBoard'))
```

To configure the model to use CDC, add the reference design to your MATLAB path by using:

```
addpath(genpath(fullfile(hdlcoder_amd_examples_root,'ZedBoard')))
```

Then configure the IP core:

1 Open the **HDL Code Generation > Target** tab of Configuration Parameters dialog box by clicking the **Settings** button.
2 Set the **Reference Design** parameter to `Default CDC system with AXI4-Stream Interface` and click **OK** to save your updated settings.
3 To enable the automatic insertion of the clock domain crossing logic between the AXI4-Stream interface and the DUT, in the **HDL Code** tab, click **Target Interface** to open the IP Core editor. Then in the **Interface Settings** tab, select **Enable the clock domain crossing on AXI4-Lite registers**.



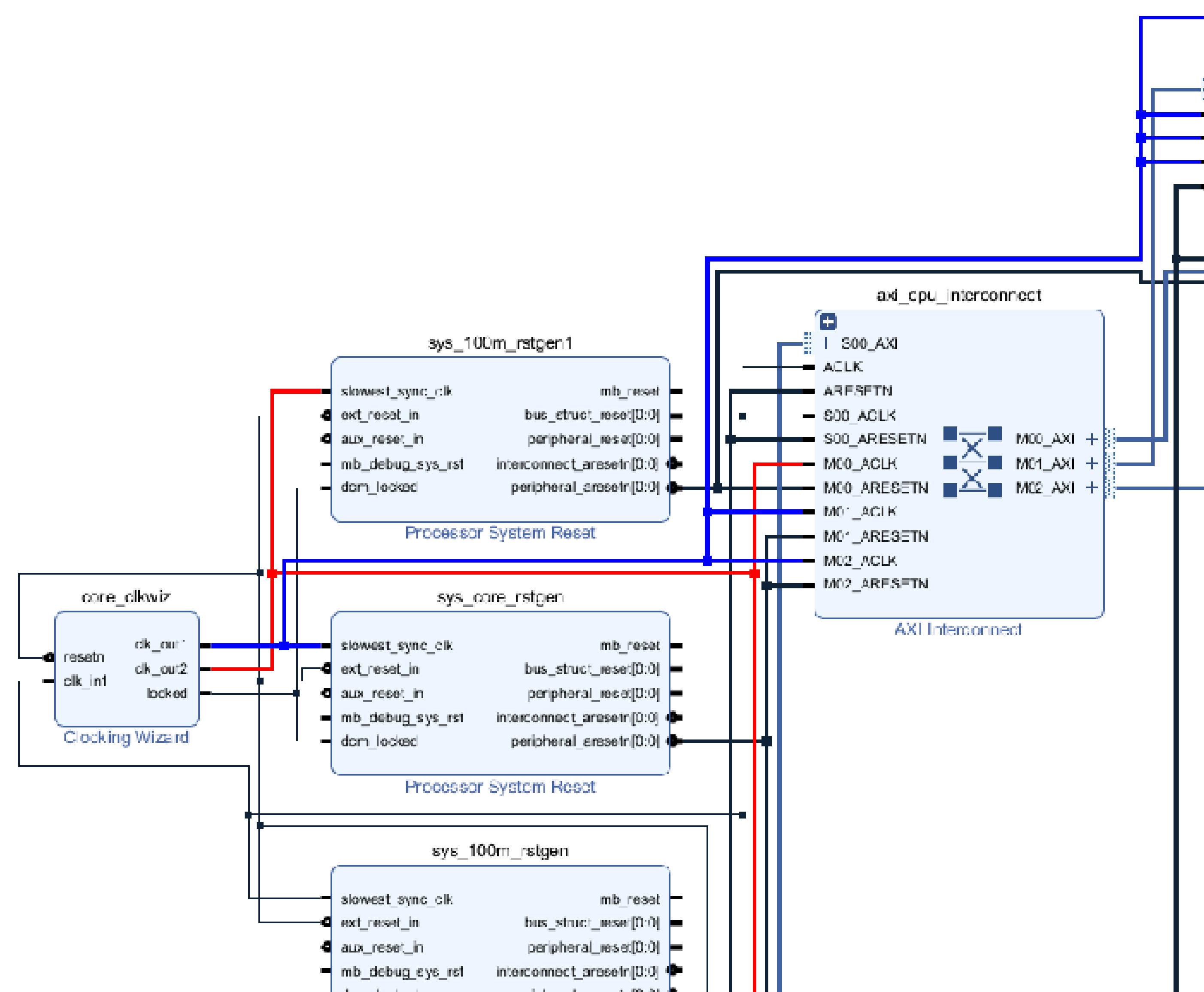
After you configure the IP core settings, you can generate the bitstream. To check the Vivado project:

1. In the **HDL Code** tab, click **Build Bitstream > Create IP Core Project**. This action inserts the generated IP core into the Default CDC system with AXI4-Stream Interface reference design, which includes a clock wizard with two different outputs. The first output is the configurable clock for the IP core, which is highlighted in blue in the Vivado block design, and the second is fixed at 50 MHz for the AXI4-Lite interface, which is highlighted in red in the Vivado block design.
2. Click the link in the **Diagnostic Viewer** to open the generated Vivado project. In Vivado, click **Open Block Design** to view the Zynq design diagram.

Diagnostic Viewer

Create IP Core Project

```
### Starting IP core project creation for 'ProgrammableFIRCDC'.
Generating Xilinx Vivado with IP Integrator project: vivado_ip_prj\vivado_prj.xpr
****** Vivado v2023.1.2 (64-bit)
  **** SW Build 3954437 on Wed Aug  9 23:07:21 MDT 2023
  **** IP Build 3954312 on Thu Aug 10 04:10:56 MDT 2023
  **** SharedData Build 3952698 on Tue Aug 08 21:36:00 MDT 2023
    ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
    ** Copyright 2022-2023 Advanced Micro Devices, Inc. All Rights Reserved.
```



Alternatively, you can directly generate the bitstream. In the Simulink Toolstrip, in the **HDL Code** tab, click **Build Bitstream** and wait until the synthesis tool runs in the external window. The timing requirements are met after the modifications.

```
C:\WINDOWS\SYSTEM32\cmd
INFO: [Project 1-479] Netlist was created with Vivado 2023.1.2
INFO: [Project 1-570] Preparing netlist for logic optimization
INFO: [Timing 38-478] Restoring timing data from binary archive.
INFO: [Timing 38-479] Binary timing data restore complete.
INFO: [Project 1-856] Restoring constraints from binary archive.
INFO: [Project 1-853] Binary constraint restore complete.
Reading XDEF placement.
Reading placer database...
Reading XDEF routing.
Read XDEF Files: Time (s): cpu = 00:00:03 ; elapsed = 00:00:03 . Memory (MB): peak = 3360.066 ; gain = 11.020
Restored from archive | CPU: 3.000000 secs | Memory: 0.000000 MB |
Finished XDEF File Restore: Time (s): cpu = 00:00:03 ; elapsed = 00:00:03 . Memory (MB): peak = 3360.066 ; gain = 11.020
Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00.012 . Memory (MB): peak = 3360.066 ; gain = 0.00
0
INFO: [Project 1-111] Unisim Transformation Summary:
  A total of 41 instances were transformed.
  RAM16X1D => RAM32X1D (RAMD32(x2)): 1 instance
  RAM32M => RAM32M (RAMD32(x6), RAMS32(x2)): 12 instances
  RAM32X1D => RAM32X1D (RAMD32(x2)): 5 instances
  SRLC32E => SRL16E: 23 instances

open_run: Time (s): cpu = 00:00:18 ; elapsed = 00:00:47 . Memory (MB): peak = 3360.066 ; gain = 924.262
get_timing_paths: Time (s): cpu = 00:00:06 ; elapsed = 00:00:05 . Memory (MB): peak = 3514.156 ; gain = 154.090
------------------------------------
Embedded system build completed.
You may close this shell.
------------------------------------
INFO: [Common 17-206] Exiting Vivado at Tue Dec 19 16:53:12 2023...

C:\Users\acuadros\Documents\CDC\CDCExample\vivado_ip_prj>
```

You can then use FPGA I/O API to prototype switching filter coefficients live on FPGA hardware. For more details, see “Prototype FPGA Design on AMD Versal Hardware with Live Data by Using MATLAB Commands” on page 39-241

For more details regarding clock domain crossing in IP core generation, see “Model Design for AXI4 Slave Interface Generation” on page 40-3.

See Also

More About

- “Enable Clock Domain Crossing on AXI4-Lite Interfaces” on page 40-439

Device Tree Generation

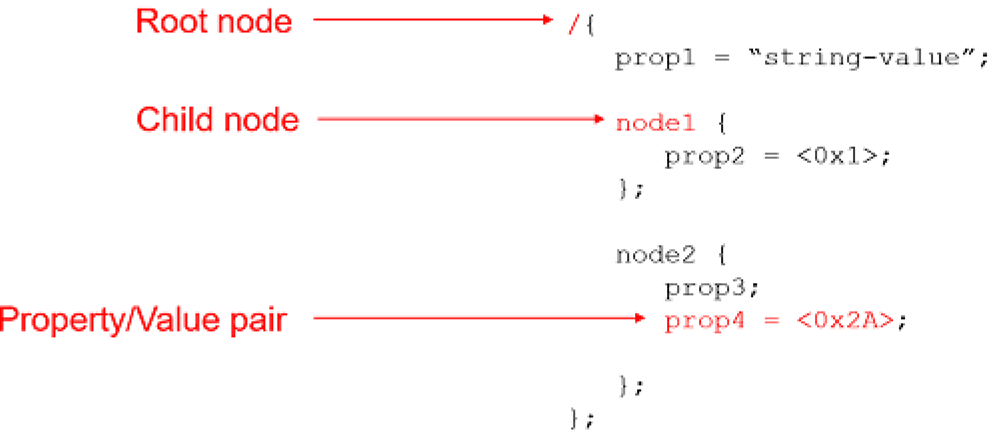
Generate Device Tree for IP Core

Dynamically generate device trees that include nodes for the HDL Coder generated IP core by using the HDL Coder IP Core generation workflow. The generated device tree also includes nodes for your board peripherals, reference design IP cores, and reference design interfaces that access the processing system (PS) of your target field programmable gate array (FPGA) or system on a chip (SoC) board. Program your target board with the generated device tree and bitstream.

Get Started with Device Trees

A device tree is a data structure that describes hardware devices to the operating system running on the target board. The operating system kernel uses the device tree description to manage the available hardware devices. For example, the operating system uses the device tree to load the specific device driver for a given hardware device.

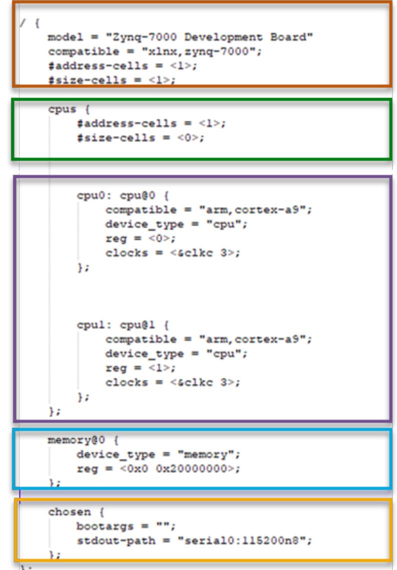
The device tree has a tree data structure format consisting of nodes and properties. The image shows these components of a device tree.

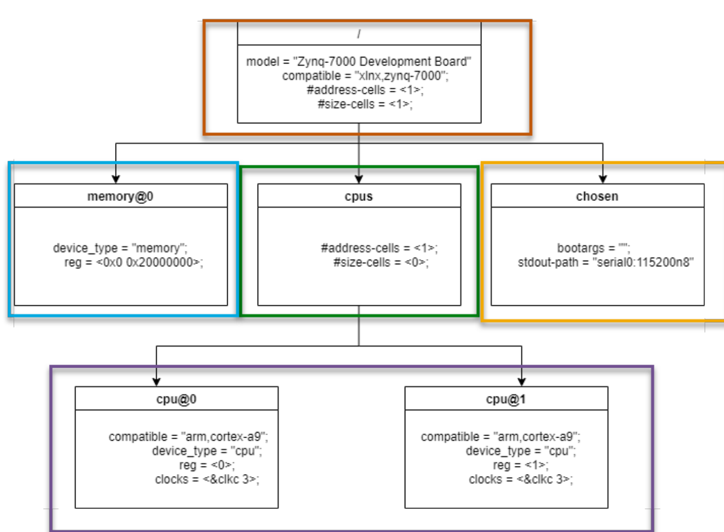


In the device tree:

- Nodes represent devices in the system.
- Each node has properties specified as property-value pairs that describe the characteristics of the device.
- Each node has exactly one parent, except for the root node, which has no parent. The root node is designated by using a forward slash.

This image shows an example device tree. The left side shows the device tree source file. The right side shows a conceptual diagram of the tree structure with nodes and properties. The names of the boxes represent the node names and the text in the box represents the property-value pairs of the nodes.





Use Device Trees with IP Core Generation Workflow

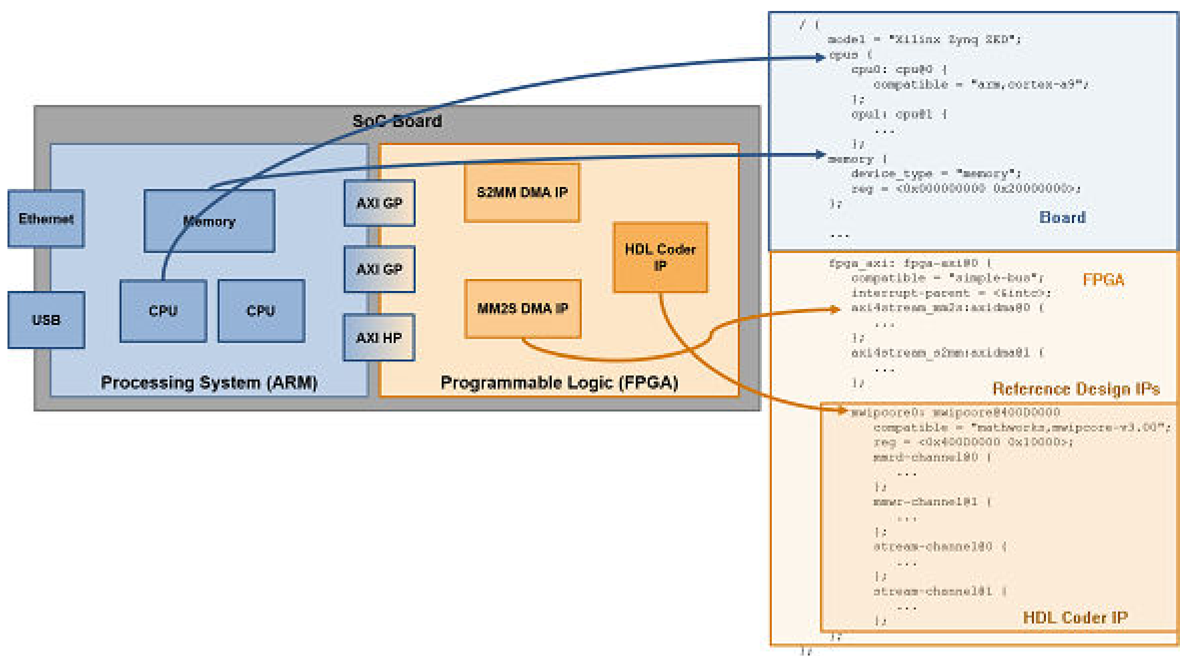
When you deploy a hardware/software design to a SoC device, the device tree informs the PS about IP cores available in the bitstream. The IP cores appear to the PS as devices, which can be accessed from the software application by using device drivers. Therefore, the device tree and bitstream must be synchronized.

You can generate device tree nodes corresponding to the generated IP core by using the IP core generation workflow. The workflow keeps the device tree and bitstream in sync by updating the device tree whenever your IP core changes.

To enable generation of device tree nodes for the HDL Coder generated IP cores, HDL Coder separates the device tree into three distinct segments:

- Board segment— Device tree nodes corresponding to fixed hardware on your board, such as the central processing unit (CPU) and processor peripherals. The board device tree is typically obtained from the board vendor.
- Reference design segment— Device tree nodes corresponding to IP cores in your reference design. This segment of the device tree is typically created when you author a reference design.
- HDL Coder IP Core segment— Device tree nodes corresponding to the IP core generated by HDL Coder. This segment of the device tree does not need to be authored as it can be dynamically generated by HDL Coder.

This image shows an SoC board with its individual device tree segments.



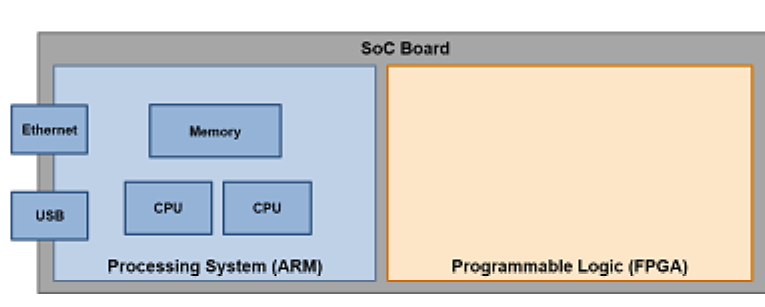
When you deploy your design to the target, HDL Coder automatically combines these segments together into a complete device tree that is used to program the target. These sections walk you through the process of registering and generating the different segments of your device tree.

Register Device Tree for Custom Board

Register the board device tree to your custom board design by using the `addDeviceTree` and `addDeviceTreeIncludeDirectory` methods of the `hdlcoder.Board` object. For more information, see `hdlcoder.Board`, `addDeviceTree`, and `addDeviceTreeIncludeDirectory`.

This image shows the board device tree registered to your custom board.

```
/ {
   model = "Xilinx Zynq ZED";
   cpus {
      cpu0: cpu@0 {
         compatible = "arm,cortex-a9";
      };
      cpu1: cpu@1 {
         ...
      };
   memory {
      device_type = "memory";
      reg = <0x00000000 0x20000000>;
   };

   ...                              Board
```



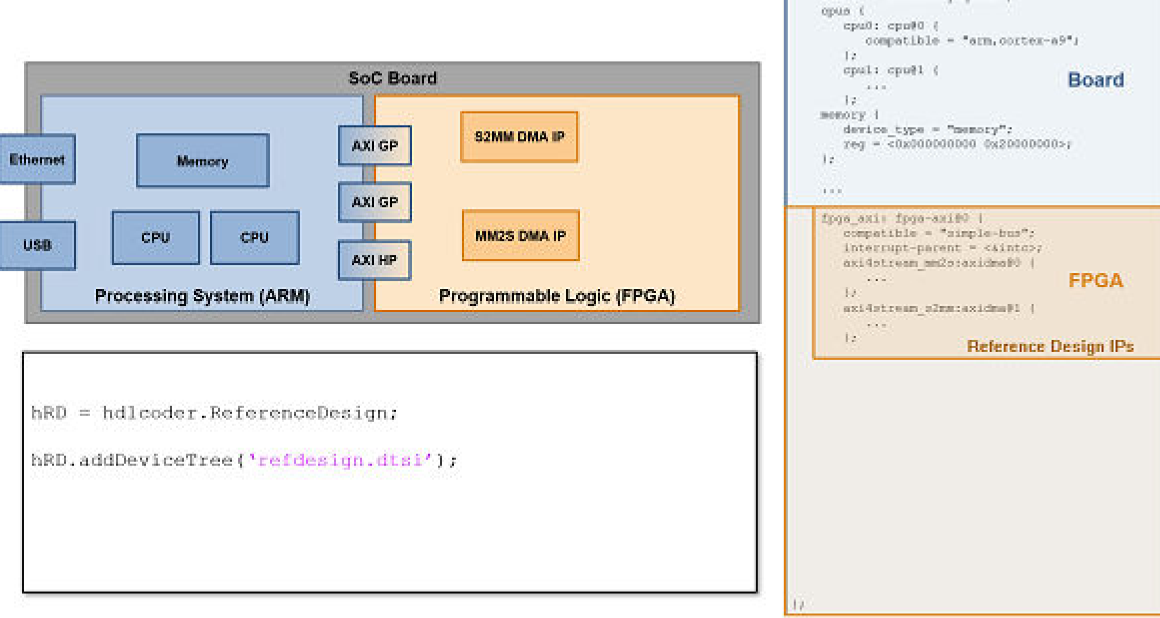

```
hB = hdlcoder.Board;

hB.addDeviceTree('board.dtsi');

hB.addDeviceTreeInclueDirectory('C:\work\devicetrees\zynq');
```

};

Register Device Tree for Custom Reference Design

Register the reference design portion of your device tree when authoring a reference design by using the `addDeviceTree` and `addDeviceTreeIncludeDirectory` methods of the `hdlcoder.ReferenceDesign` object. To register a device tree, your reference design must contain a processing system IP. Enable the `HasProcessingSystem` property of the `hdlcoder.ReferenceDesign` object. See `hdlcoder.ReferenceDesign`, `addDeviceTree`, and `addDeviceTreeIncludeDirectory`.

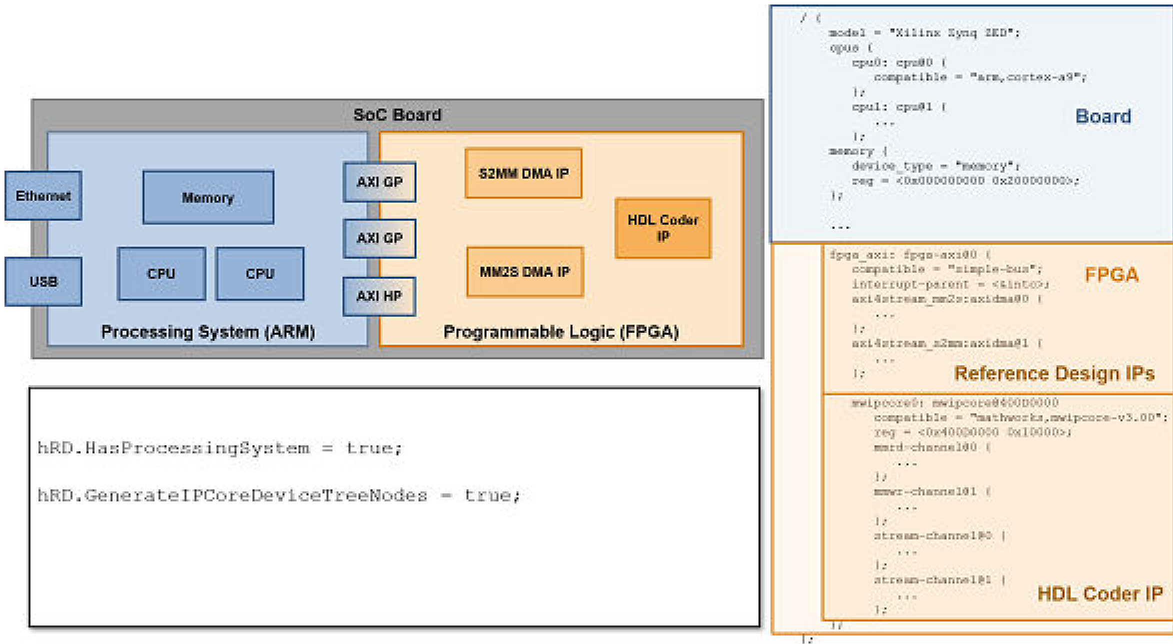
This image shows the reference design for a device tree registered to your custom reference design.



Generate IP Core Device Tree Node

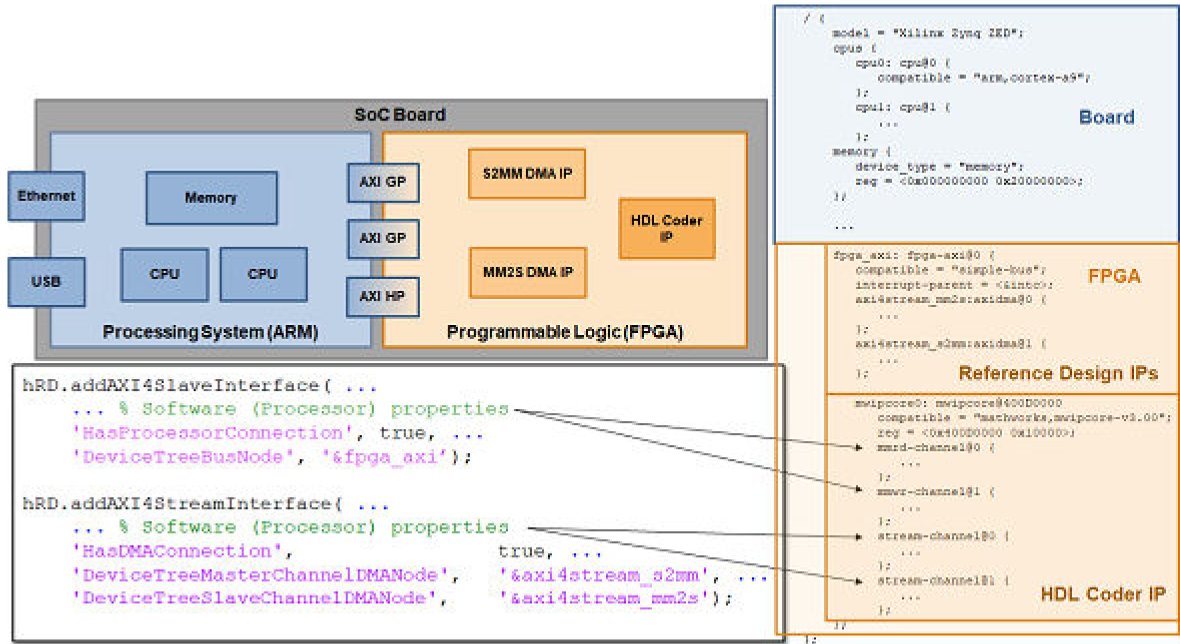
During the IP core generation workflow, HDL Coder generates an IP core and inserts it into your reference design. HDL Coder can also generate corresponding device tree nodes for this IP core and combine them with the board and reference design device trees. Generate the device tree node for your IP core by using the HDL Coder IP core generation workflow. Enable the `GenerateIPCoreDeviceTreeNodes` property of the `hdlcoder.ReferenceDesign` object to generate the device tree nodes for your IP core. See “GenerateIPCoreDeviceTreeNodes”.

This image shows the reference design for a device tree when generating and adding device tree nodes is enabled for your IP core.



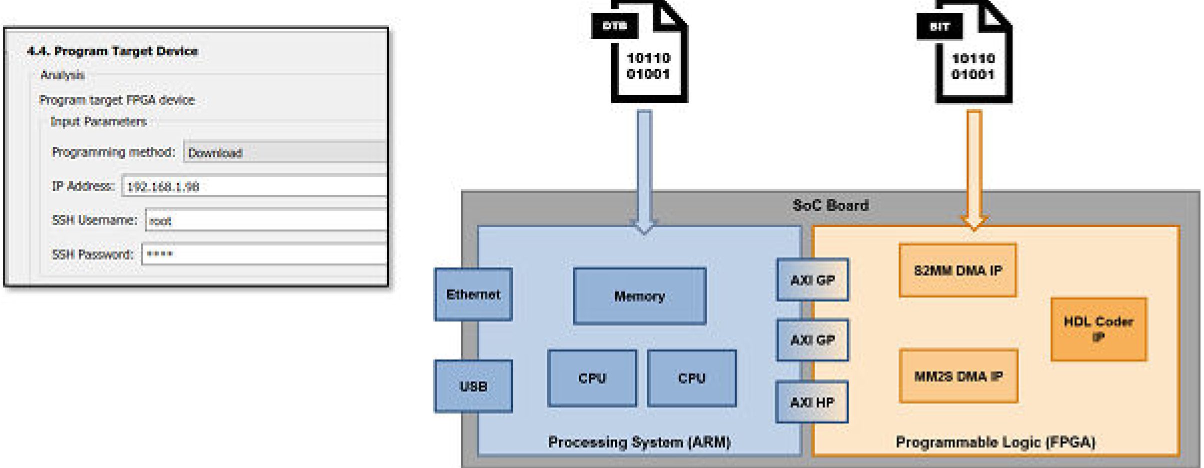
To enable device tree generation for your IP core, you must specify the relevant information for the reference design interfaces that connect to the processing system. Enable the generation of device tree nodes by specifying information for the interfaces that connect to the PS on your target device. Interfaces specify references to other device tree nodes. To enable device tree generation for the AXI4 slave interface, enable the `HasProcessorConnection` property and specify the value for the `DeviceTreeNode` property of the `addAXI4SlaveInterface` method for the `hdlcoder.ReferenceDesign` object. See “HasProcessorConnection”, `addAXI4SlaveInterface`, and “DeviceTreeNodes”. Enable device tree generation for the AXI4 Stream interface by enabling the `HasDMAConnection` and specifying values for the `DeviceTreeMasterChannelDMANode` and `DeviceTreeSlaveChannelDMANode`. See `addAXI4StreamInterface`, “DeviceTreeMasterChannelDMANode”, “DeviceTreeSlaveChannelDMANode”, and “HasDMAConnection”.

This image shows the device tree node generation enabled for the IP Core AXI4 slave and AXI4 stream interfaces.



Deploy Device Tree and Bitstream

Download the generated device tree and bitstream for your custom board, reference design, and IP core to the target device by using the HDL Coder IP core generation workflow. The IP core generation workflow compiles the board, reference design, and IP core device trees into a single device tree and then programs them onto the target device along with the generated bitstream. This image shows the generated device tree and bitstream programmed onto the target device.

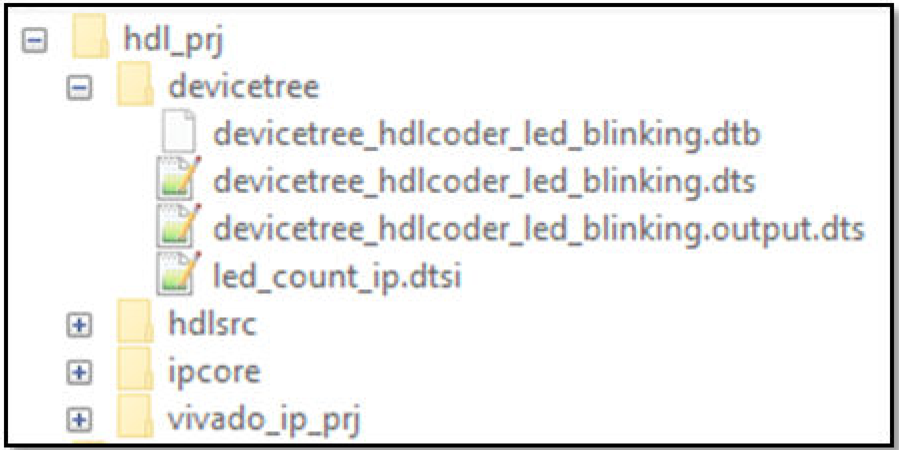


During the download, HDL Coder uses the device tree compiler on the target board to combine the board device tree, reference design device tree, and IP core device tree. HDL Coder then copies the compiled device tree files back to your host computer.

The following device tree files are generated during the IP Core Generation workflow:

- `devicetree_<model-name>.dtb`— The binary device tree file used to program the board. This file is generated by the device tree compiler.
- `devicetree_<model-name>.dts`— The source device tree file provided as an input to the device tree compiler. This file is generated by HDL Coder.
- `devicetree_<model-name>.output.dts`— A combined source file which contains information for the board device tree, reference design device tree, and your IP core device tree. This file can be used for debugging, and is generated by the device tree compiler.
- `<ip-core-name>.dtsi`: The IP core segment of the device tree. This file is generated by HDL Coder.

The image shows the folder location of the generated device tree files for a sample IP core.



Sample Device Tree

The image shows your reference design file with the interface details. On the left is an example of a reference design file with interface details. On the right is the generated device tree file.

```
hRD = hdlcoder.ReferenceDesign;
hRD.addDeviceTree("refdesign.dtsi");
hRD.HasProcessingSystem = true;
hRD.GenerateIPCoreDeviceTreeNodes = true;
hRD.addAXI4StreamInterface(...
    ... % Hardware (FPGA) properties
    'MasterChannelEnable', true,...
    'SlaveChannelEnable', true, ...
    'MasterChannelConnection', 'axi_dma_s2mm/S_AXIS_S2MM', ...
    'SlaveChannelConnection', 'axi_dma_mm2s/M_AXIS_MM2S', ...
    'MasterChannelDataWidth', 32, ...
    'SlaveChannelDataWidth', 32, ...
    ... % Software (Processor) properties
    'HasDMAConnection', true, ...
    'DeviceTreeMasterChannelDMANode', '&axi4stream_s2mm', ...
    'DeviceTreeSlaveChannelDMANode', '&axi4stream_mm2s');
```

```
&mwipcore0 {
    stream-channel@1 {
        reg = <0x1>;
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "mathworks,axi4stream-s2mm-channel-v1.00";
        dma-names = "s2mm";
        dmas = <&axi4stream_s2mm 0>;
        mathworks,dev-name = "s2mm0";
        mathworks,sample-cnt-reg = <0x8>;

        data-channel@0 {
            compatible = "mathworks,iio-data-channel-v1.00";
            mathworks,data-format = u32/32>>0;
            reg = <0x0>;
        };
    };
};
```

See Also

`addDeviceTree` | `addDeviceTreeIncludeDirectory` | `addDeviceTree` | `addDeviceTreeIncludeDirectory` | “HasDMAConnection” | “DeviceTreeMasterChannelDMANode” | “DeviceTreeSlaveChannelDMANode”

External Websites

- https://www.devicetree.org