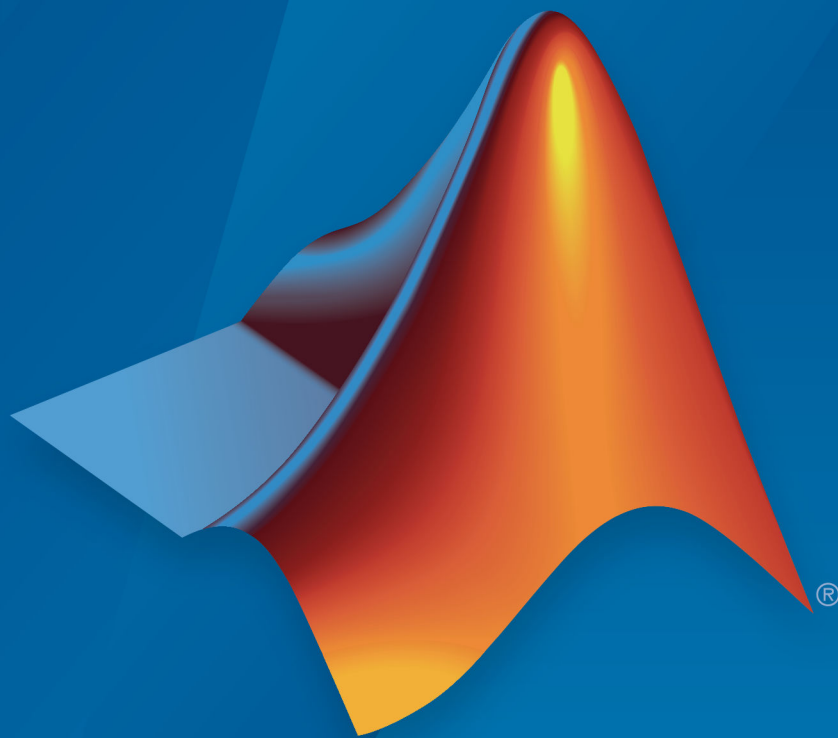


HDL Coder™

User's Guide



MATLAB® & SIMULINK®

R2019b



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 2012-2019 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)

HDL Code Generation from MATLAB

1 Functions Supported for HDL Code Generation

Functions Supported for HDL Code Generation – Alphabetical List	1-2
Functions Supported for HDL Code Generation – Categorical List	1-10
Arithmetic Operations in MATLAB	1-11
Bitwise Operations in MATLAB	1-11
Complex Numbers in MATLAB	1-11
Control Flow in MATLAB	1-12
Logical Operators in MATLAB	1-12
Arrays in MATLAB	1-13
Relational Operators in MATLAB	1-13
Fixed-Point Designer	1-13

2 MATLAB Algorithm Design

Supported MATLAB Data Types, Operators, and Control Flow Statements	2-2
Supported Data Types	2-2
Supported Operators	2-4
Control Flow Statements	2-6

Persistent Variables and Persistent Array Variables	2-9
Persistent Variables	2-9
Persistent Array Variables	2-10
Complex Data Type Support	2-11
Declaring Complex Signals	2-11
Conversion Between Complex and Real Signals	2-12
Support for Vectors of Complex Numbers	2-13
HDL Code Generation for System Objects	2-15
Why Use System Objects?	2-15
Predefined System Objects	2-15
User-Defined System Objects	2-16
Limitations of HDL Code Generation for System Objects	2-16
System object Examples for HDL Code Generation	2-17
Predefined System Objects Supported for HDL Code Generation	2-18
Predefined System Objects in MATLAB Code	2-18
Predefined System Objects in the MATLAB System Block	2-20
Load constants from a MAT-File	2-21
Generate Code for User-Defined System Objects	2-22
How To Create A User-Defined System object	2-22
User-Defined System object Example	2-22
Map Matrices to ROM	2-25
Fixed-Point Bitwise Functions	2-26
Fixed-Point Run-Time Library Functions	2-32
Fixed-Point Function Limitations	2-35
Model State with Persistent Variables and System Objects	2-37
Bitwise Operations in MATLAB for HDL Code Generation	2-41
Bit Shifting and Rotation	2-41
Bit Slicing and Bit Concatenation	2-43

Guidelines for Efficient HDL Code	2-46
MATLAB Design Requirements for HDL Code Generation	2-47
What Is a MATLAB Test Bench?	2-48
MATLAB Test Bench Requirements and Best Practices	2-49
MATLAB Test Bench Requirements	2-49
MATLAB Test Bench Best Practices	2-49

MATLAB Best Practices and Design Patterns for HDL Code Generation

3

Model a Counter for HDL Code Generation	3-2
MATLAB Counter	3-2
MATLAB Code for the Counter	3-3
Best Practices in this Example	3-3
Model a State Machine for HDL Code Generation	3-5
MATLAB State Machines	3-5
MATLAB Code for the Mealy State Machine	3-5
MATLAB Code for the Moore State Machine	3-7
Best Practices	3-9
Generate Hardware Instances For Local Functions	3-10
MATLAB Local Functions	3-10
MATLAB Code for mlhdlc_two_counters.m	3-10
Implement RAM Using MATLAB Code	3-13
Implementation of RAM	3-13
Implement RAM Using a Persistent Array or System object Properties	3-13
Implement RAM Using hdl.RAM	3-14
For-Loop Best Practices for HDL Code Generation	3-16
MATLAB Loops	3-16
Monotonically Increasing Loop Counters	3-16

Persistent Variables in Loops	3-17
Persistent Arrays in Loops	3-18

Fixed-Point Conversion

4

Floating-Point to Fixed-Point Conversion	4-2
Fixed-Point Type Conversion and Refinement	4-15
Working with Generated Fixed-Point Files	4-25
Specify Type Proposal Options	4-32
Log Data for Histogram	4-36
View and Modify Variable Information	4-38
View Variable Information	4-38
Modify Variable Information	4-38
Revert Changes	4-39
Promote Sim Min and Sim Max Values	4-40
Automated Fixed-Point Conversion	4-41
License Requirements	4-41
Automated Fixed-Point Conversion Capabilities	4-41
Code Coverage	4-42
Proposing Data Types	4-45
Locking Proposed Data Types	4-47
Viewing Functions	4-47
Viewing Variables	4-47
Histogram	4-53
Function Replacements	4-55
Validating Types	4-55
Testing Numerics	4-56
Detecting Overflows	4-56
Custom Plot Functions	4-58
Visualize Differences Between Floating-Point and Fixed-Point Results	4-60

Inspecting Data Using the Simulation Data Inspector . .	4-66
What Is the Simulation Data Inspector?	4-66
Import Logged Data	4-66
Export Logged Data	4-66
Group Signals	4-67
Run Options	4-67
Create Report	4-67
Comparison Options	4-67
Enabling Plotting Using the Simulation Data Inspector	4-67
Save and Load Simulation Data Inspector Sessions	4-68
Enable Plotting Using the Simulation Data Inspector . .	4-69
From the UI	4-69
From the Command Line	4-69
Replacing Functions Using Lookup Table Approximations 	4-71
Replace a Custom Function with a Lookup Table	4-72
Using the HDL Coder App	4-72
From the Command Line	4-77
Replace the exp Function with a Lookup Table	4-80
From the UI	4-80
From the Command Line	4-85
Data Type Issues in Generated Code	4-88
Enable the Highlight Option in a Project	4-88
Enable the Highlight Option at the Command Line	4-88
Stowaway Doubles	4-88
Stowaway Singles	4-88
Expensive Fixed-Point Operations	4-88

Code Generation

5

Create and Set Up Your Project	5-2
Create a New Project	5-2
Open an Existing Project	5-4

Add Files to the Project	5-4
Specify Properties of Entry-Point Function Inputs	5-6
When to Specify Input Properties	5-6
Why You Must Specify Input Properties	5-6
Properties to Specify	5-6
Rules for Specifying Properties of Primary Inputs	5-8
Methods for Defining Properties of Primary Inputs	5-8
Basic HDL Code Generation with the Workflow Advisor	5-10
HDL Code Generation from System Objects	5-15
Code Generation Reports	5-20
Report Generation	5-20
Report Location	5-21
Errors and Warnings	5-21
Files and Functions	5-21
MATLAB Source	5-22
MATLAB Variables	5-23
Additional Reports	5-24
Report Limitations	5-24
Generate Instantiable Code for Functions	5-26
How to Generate Instantiable Code for Functions	5-26
Generate Code Inline for Specific Functions	5-26
Limitations for Instantiable Code Generation for Functions	5-26
Integrate Custom HDL Code Into MATLAB Design	5-28
Define the hdl.BlackBox System object	5-28
Use System object In MATLAB Design Function	5-30
Generate HDL Code	5-30
Limitations for hdl.BlackBox	5-33
Enable MATLAB Function Block Generation	5-34
Requirements for MATLAB Function Block Generation	5-34
Enable MATLAB Function Block Generation	5-34
Restrictions for MATLAB Function Block Generation ..	5-34
Results of MATLAB Function Block Generation	5-34

System Design with HDL Code Generation from MATLAB and Simulink	5-36
Generate HDL Code from MATLAB Code Using the Command Line Interface	5-40
Specify the Clock Enable Rate	5-45
Why Specify the Clock Enable Rate?	5-45
How to Specify the Clock Enable Rate	5-45
Specify Test Bench Clock Enable Toggle Rate	5-47
When to Specify Test Bench Clock Enable Toggle Rate	5-47
How to Specify Test Bench Clock Enable Toggle Rate ..	5-47
Generate an HDL Coding Standard Report from MATLAB	5-49
Using the HDL Workflow Advisor	5-49
Using the Command Line	5-51
Generate an HDL Lint Tool Script	5-53
How To Generate an HDL Lint Tool Script	5-53
Generate Board-Independent IP Core from MATLAB Algorithm	5-56
Requirements and Limitations for IP Core Generation ..	5-56
Generate Board-Independent IP Core	5-57
Minimize Clock Enables	5-59
Using the GUI	5-60
Using the Command Line	5-60
Limitations	5-60

Verification

6

Verify Code with HDL Test Bench	6-2
Test Bench Generation	6-6
How Test Bench Generation Works	6-6

Test Bench Data Files	6-6
Test Bench Data Type Limitations	6-6
Use Constants Instead of File I/O	6-7

Deployment

7

Generate Synthesis Scripts	7-2
---	------------

Optimization

8

RAM Mapping for MATLAB Code	8-2
Map Persistent Arrays and dsp.Delay to RAM	8-3
How To Enable RAM Mapping	8-3
RAM Mapping Requirements for Persistent Arrays and System object Properties	8-4
RAM Mapping Requirements for dsp.Delay System Objects	8-6
RAM Mapping Comparison for MATLAB Code	8-8
Pipelining MATLAB Code	8-9
Port Registers	8-9
Input and Output Pipeline Registers	8-9
Operation Pipelining	8-9
Pipeline MATLAB Expressions	8-11
How To Pipeline a MATLAB Expression	8-11
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

Optimize MATLAB Loops	8-15
Loop Streaming	8-15
Loop Unrolling	8-15
How to Optimize MATLAB Loops	8-16
Limitations for MATLAB Loop Optimization	8-16
Constant Multiplier Optimization	8-18
What is Constant Multiplier Optimization?	8-18
Specify Constant Multiplier Optimization	8-19
Distributed Pipelining for Clock Speed Optimization ..	8-20
Map Matrices to Block RAMs to Reduce Area	8-25
Resource Sharing of Multipliers to Reduce Area	8-30
Loop Streaming to Reduce Area	8-39
Constant Multiplier Optimization to Reduce Area	8-45

HDL Workflow Advisor Reference

9

HDL Workflow Advisor	9-2
Overview	9-2
MATLAB to HDL Code and Synthesis	9-7
MATLAB to HDL Code Conversion	9-7
Code Generation: Target Tab	9-7
Code Generation: Coding Style Tab	9-8
Code Generation: Clocks and Ports Tab	9-10
Code Generation: Test Bench Tab	9-12
Code Generation: Optimizations Tab	9-14
Simulation and Verification	9-16
Synthesis and Analysis	9-16

HDL Code Generation from Simulink

10 Model Design for HDL Code Generation

10

Signal and Data Type Support	10-2
Buses	10-2
Enumerations	10-3
Matrices	10-4
Unsupported Signal and Data Types	10-8
Use Simulink Templates for HDL Code Generation	10-9
Create Model Using HDL Coder Model Template	10-9
HDL Coder Model Templates	10-10
Generate DUT Ports for Tunable Parameters	10-23
Prerequisites	10-24
Create and Add Tunable Parameter That Maps to DUT Ports	10-24
Generated Code	10-24
Limitations	10-25
Use Tunable Parameter in Other Blocks	10-25
Generate Parameterized Code for Referenced Models	10-27
Parameterize Referenced Model for HDL Code Generation	10-27
Restrictions	10-27
Generating HDL Code for Subsystems with Array of Buses	10-29
How HDL Coder Generates Code for Array of Buses . .	10-29
Array of Buses Limitations	10-32
Generate HDL Code for Blocks Inside For Each Subsystem	10-33
Model and Debug Test Point Signals with HDL Coder™	10-39

Allocate Sufficient Delays for Floating-Point Operations	
.....	10-49
Problem	10-49
Cause	10-49
Solution	10-50
Optimize Generated HDL Code for Multirate Designs with	
Large Rate Differentials	10-55
Issue	10-55
Description	10-55
Recommendations	10-58
Getting Started with HDL Coder Native Floating-Point	
Support	10-65
Numeric Considerations and IEEE-754 Standard	
Compliance	10-65
Data Type Considerations	10-66
Numeric Considerations with Native Floating-Point ..	10-69
Round to Nearest Rounding Mode	10-69
Denormal Numbers	10-70
Exception Handling	10-70
Relative Accuracy and ULP Considerations	10-71
ULP of Native Floating-Point Operators	10-73
Considerations	10-74
Latency Values of Floating Point Operators	10-77
Math Operations	10-77
Trigonometric and Exponential Operations	10-79
Comparisons and Conversions	10-81
Latency Considerations with Native Floating Point ...	10-83
Generate Target-Independent HDL Code with Native	
Floating-Point	10-92
How HDL Coder Generates Target-Independent HDL Code	
.....	10-92
Enable Native Floating Point and Generate Code	10-93
View Code Generation Report	10-95
Analyze Results	10-97
Limitation	10-99

Verify the Generated Code from Native Floating-Point	10-101
Specify the Tolerance Strategy	10-101
Verify the Generated Code with HDL Test Bench	10-102
Verify the Generated Code with Cosimulation	10-103
Limitation	10-105
Simulink Blocks Supported with Native Floating-Point	10-106
HDL Floating Point Operations Library	10-106
Supported Simulink Blocks in Math Operations Library	10-106
Supported Simulink Blocks in Other Libraries	10-107
Simulink Block Restrictions	10-108
Supported Data Types and Scope	10-110
Supported Data Types	10-110
Unsupported Data Types	10-112
Scope for Variables	10-113
Verilog HDL Import: Import Verilog Code and Generate Simulink Model	10-114
Why Use HDL Import?	10-114
HDL Import Requirements	10-114
How to Import HDL Code	10-115
Model Location	10-115
Errors and Warnings	10-115
Supported Verilog Constructs for HDL Import	10-117
Module Definition and Instantiations	10-117
Data Types and Vectors	10-118
Identifiers and Comments	10-118
Assignments	10-119
Operators	10-119
Conditional and Looping Statements	10-120
Procedural Blocks and Events	10-120
Other Constructs	10-121
Limitations of Verilog HDL Import	10-123
Using the Float Typecast Block	10-138

Code Generation Options in the HDL Coder Dialog Boxes

11

Set HDL Code Generation Options	11-2
HDL Code Generation Options in the Configuration Parameters Dialog Box	11-2
HDL Code Tab in Simulink Toolstrip	11-4
HDL Code Options in the Block Context Menu	11-5
The HDL Block Properties Dialog Box	11-6

HDL Code Generation Pane: General

12

HDL Code Generation Top-Level Pane Overview	12-2
Buttons in the HDL Code Generation Top-Level Pane	12-2
Target	12-3
Generate HDL for	12-3
Language	12-4
Folder	12-5
Restore Model Defaults	12-6
Run Compatibility Checker	12-6
Generate	12-7

HDL Code Generation Pane: Target

13

Target Overview	13-2
Tool and Device	13-3
Synthesis Tool	13-3
Family	13-4
Device	13-5
Package	13-6
Speed	13-7

Target Frequency	13-9
Settings	13-9
Command-Line Information	13-10
See Also	13-10

14

HDL Code Generation Pane: Optimization

Optimization Overview	14-2
Balance delays	14-3
Settings	14-3
Command-Line Information	14-3
See Also	14-4
RAM Mapping	14-5
Map pipeline delays to RAM	14-5
RAM mapping threshold (bits)	14-6
Transform non zero initial value delay	14-8
Settings	14-8
Command-Line Information	14-8
See Also	14-9
Multiplier partitioning threshold	14-10
Settings	14-10
Command-Line Information	14-10
See Also	14-11
Distributed Pipelining	14-12
Hierarchical distributed pipelining	14-12
Distributed pipelining priority	14-13
Clock Rate Pipelining	14-15
Clock-rate pipelining	14-15
Allow clock-rate pipelining of DUT output ports	14-16
Adaptive pipelining	14-18
Settings	14-18
Dependency	14-18

Command-Line Information	14-18
See Also	14-19
Preserve design delays	14-20
Settings	14-20
Command-Line Information	14-20
See Also	14-21
Resource Sharing of Adders and Multipliers	14-22
Share Adders	14-22
Adder sharing minimum bitwidth	14-23
Share Multipliers	14-24
Multiplier sharing minimum bitwidth	14-26
Multiplier promotion threshold	14-27
Resource Sharing of Multiply-Add and Other Blocks ..	14-29
Share Multiply-Add blocks	14-29
Multiply-Add block sharing minimum bitwidth	14-30
Share Atomic subsystems	14-31
Share MATLAB Function blocks	14-32
Share Floating-Point IPs	14-35
Settings	14-35
Dependency	14-35
Command-Line Information	14-35
See Also	14-36
Multicycle Path Constraints	14-37
Enable based constraints	14-37
Register-to-register path info	14-38

HDL Code Generation Pane: Floating Point

15

Floating Point Overview	15-2
Floating Point IP Library	15-3
Settings	15-3
Command-Line Information	15-3
See Also	15-4

Native Floating Point	15-5
Latency Strategy	15-5
Handle Denormals	15-6
Mantissa Multiplier Strategy	15-7
FPGA Floating-Point Libraries	15-10
Initialize IP Pipelines To Zero	15-10
Latency Strategy	15-11
Objective	15-12
IP Settings	15-13

HDL Code Generation Pane: Global Settings

16

Global Settings Overview	16-3
Clock Settings and Timing Controller Postfix	16-4
Clock input port	16-4
Clock inputs	16-5
Clock edge	16-6
Clocked process postfix	16-7
Timing controller postfix	16-8
Reset Settings	16-10
Reset type	16-10
Reset asserted level	16-11
Reset input port	16-13
Clock Enable Settings	16-15
Clock enable input port	16-15
Enable prefix	16-16
Oversampling factor	16-18
Settings	16-18
Dependency	16-18
Command-Line Information	16-18
See Also	16-19
Comment in header	16-20
Settings	16-20

Command-Line Information	16-20
See Also	16-21
Language-Specific Identifiers and File Extensions	16-22
Verilog file extension	16-22
VHDL file extension	16-23
Entity conflict postfix	16-24
Package postfix	16-25
Reserved word postfix	16-26
Module name prefix	16-26
Split entity and architecture	16-28
Split entity file postfix	16-28
Split arch file postfix	16-28
Split entity and architecture	16-29
Complex Signals Postfix	16-31
Complex real part postfix	16-31
Complex imaginary part postfix	16-31
VHDL Architecture and Library Name	16-32
VHDL architecture name	16-32
VHDL library name	16-32
Pipeline postfix	16-33
Settings	16-33
Command-Line Information	16-33
Generate VHDL code for model references into a single library	16-35
Settings	16-35
Dependency	16-35
Command-Line Information	16-35
Generate Statement Labels	16-36
Block generate label	16-36
Output generate label	16-36
Instance generate label	16-37
Vector and Component Instances Labels	16-38
Vector prefix	16-38
Instance postfix	16-38
Instance prefix	16-39

Map file postfix	16-40
Settings	16-40
Command-Line Information	16-40
Input and Output Port Data Types	16-41
Input data type	16-41
Output data type	16-42
Clock Enable output port	16-43
Settings	16-43
Command-Line Information	16-43
See Also	16-43
Minimize Clock Enables and Reset Signals	16-44
Minimize clock enables	16-44
Minimize global resets	16-46
Use trigger signal as clock	16-49
Settings	16-49
Command-Line Information	16-49
Enable HDL DUT port generation for test points	16-51
Settings	16-51
Command-Line Information	16-51
See Also	16-52
RTL Annotations	16-53
Use Verilog `timescale directives	16-53
Verilog timescale specification	16-53
Inline VHDL configuration	16-54
Concatenate type safe zeros	16-55
Emit time/date stamp in header	16-56
Include requirements in block comments	16-57
RTL Customizations for Constants and MATLAB Function	
Blocks	16-58
Inline MATLAB Function block code	16-58
Represent constant values by aggregates	16-59
RTL Customizations for RAMs	16-60
Initialize all RAM blocks	16-60
RAM Architecture	16-61

No-reset registers initialization	16-62
Settings	16-62
Usage Notes	16-62
Command-Line Information	16-63
See Also	16-64
RTL Style	16-65
Use “rising_edge/falling_edge” style for registers	16-65
Minimize intermediate signals	16-66
Scalarize vector ports	16-67
Unroll for Generate Loops in VHDL code	16-69
Generate parameterized HDL code from masked subsystem	16-70
Enumerated Type Encoding Scheme	16-71
Timing Controller Settings	16-73
Optimize timing controller	16-73
Timing controller architecture	16-74
File Comment Customization	16-75
Custom File Header Comment	16-75
Custom File Footer Comment	16-75
Choose Coding Standard and Report Options	16-77
HDL coding standard	16-77
Report options	16-78
Basic Coding Practices	16-80
Check for duplicate names	16-80
Check for HDL keywords in design names	16-81
Check module, instance, entity name length	16-82
Check signal, port, and parameter name length	16-83
RTL Description Rules for clock enables and resets ..	16-86
Check for clock enable signals	16-86
Detect usage of reset signals	16-87
Detect usage of asynchronous reset signals	16-88
RTL Description Rules for Conditionals	16-90
Check for conditional statements in processes	16-90
Check if-else statement chain length	16-91
Check if-else statement nesting depth	16-92

Other RTL Description Rules	16-94
Minimize use of variables	16-94
Check for initial statements that set RAM initial values	16-95
Check multiplier width	16-96
RTL Design Rules	16-98
Check for non-integer constants	16-98
Check line length	16-99
Model Generation for HDL Code	16-101
Generated model	16-101
Validation model	16-102
Naming Options	16-104
Prefix for generated model name	16-104
Suffix for validation model name	16-105
Layout Options	16-107
Auto block placement	16-107
Auto signal routing	16-107
Inter-block horizontal scaling	16-108
Inter-block vertical scaling	16-109
Diagnostics for Optimizations	16-110
Highlight feedback loops inhibiting delay balancing and optimizations	16-110
Highlight blocks inhibiting clock-rate pipelining	16-111
Highlight blocks inhibiting distributed pipelining ...	16-112
Diagnostics for Reals and Black Box Interfaces	16-114
Check for name conflicts in black box interfaces	16-114
Check for presence of reals in generated HDL code .	16-115
Code Generation Output	16-117
Generate HDL code	16-117

Report Overview	17-2
See Also	17-2
Generate traceability report	17-3
Settings	17-3
Dependency	17-3
Command-Line Information	17-3
See Also	17-4
Traceability style	17-5
Settings	17-5
Dependency	17-5
Command-Line Information	17-6
See Also	17-6
Generate model Web view	17-7
Settings	17-7
Dependencies	17-7
Command-Line Information	17-7
See Also	17-8
Generate resource utilization report	17-9
Settings	17-9
Command-Line Information	17-9
See Also	17-10
Generate high-level timing critical path report	17-11
Settings	17-11
Command-Line Information	17-11
See Also	17-12
Generate optimization report	17-13
Settings	17-13
Command-Line Information	17-13
See Also	17-14

Test Bench Overview	18-2
Generate Test Bench Button	18-2
Test Bench Generation Output	18-3
HDL test bench	18-3
Cosimulation model	18-4
SystemVerilog DPI test bench	18-5
Simulation tool	18-6
HDL code coverage	18-7
Test Bench name, data file, and reference Postfix	18-9
Test bench name postfix	18-9
Test bench reference postfix	18-9
Test bench data file name postfix	18-10
Clock Input Signals	18-12
Force clock	18-12
Clock high time (ns)	18-12
Clock low time (ns)	18-13
Setup and Hold Time	18-15
Hold time (ns)	18-15
Setup time (ns)	18-15
Clock Enable and Reset Input Signals	18-17
Force clock enable	18-17
Clock enable delay (in clock cycles)	18-18
Force reset	18-19
Reset length (in clock cycles)	18-20
Test Bench Stimulus and Output	18-21
Hold input data between samples	18-21
Initialize test bench inputs	18-22
Ignore output data checking (number of samples)	18-23
Use file I/O to read/write test bench data	18-24
Multi-file test bench	18-26
Description	18-26
Settings	18-26

Dependency	18-27
Command-Line Information	18-27
See Also	18-27
Floating Point Tolerance	18-28
Floating point tolerance check based on	18-28
Tolerance Value	18-29
Simulation library path	18-30
Settings	18-30
Dependency	18-30
Command-Line Information	18-30
See Also	18-30

HDL Code Generation Pane: EDA Tool Scripts

19

EDA Tool Scripts Overview	19-2
Generate EDA scripts	19-3
Settings	19-3
Command-Line Information	19-3
See Also	19-3
Compilation Script	19-4
Compile file postfix	19-4
Compile initialization	19-4
Compile command for VHDL	19-5
Compile command for Verilog	19-6
Compile termination	19-6
Simulation Script	19-8
Simulation file postfix	19-8
Simulation initialization	19-8
Simulation command	19-9
Simulation waveform viewing command	19-10
Simulation termination	19-10
Simulator flags	19-11

Synthesis Script	19-12
Choose synthesis tool	19-12
Synthesis file postfix	19-14
Synthesis initialization	19-15
Synthesis command	19-16
Synthesis termination	19-17
Additional files to add to synthesis project	19-17
Lint Script	19-19
Choose HDL lint tool	19-19
Lint initialization	19-20
Lint command	19-20
Lint termination	19-21

Modeling Guidelines

20

HDL Modeling Guidelines Severity Levels	20-3
Model Design and Compatibility Guidelines - By	
Numbered List	20-5
Guidelines 1.1: Basic Settings	20-5
Guidelines 1.2: DUT Subsystem and Hierarchical Modeling	
.....	20-6
Guidelines 1.3: Guidelines for Vectors and Buses	20-7
Guidelines 1.4: Guidelines for Clock Bundle Signals ...	20-8
Guidelines 1.5: Modeling Guidelines for Native Floating	
Point	20-8
Guidelines for Supported Blocks and Data Types - By	
Numbered List	20-10
Guidelines 2.1: Blocks in HDL RAMs and HDL Operations	
Library	20-10
Guidelines 2.2: Blocks in Logic and Bit Operations Library	
.....	20-10
Guidelines 2.3: Lookup Table Blocks	20-11
Guidelines 2.4: Ports and Subsystems	20-11
Guideline 2.5: Rate Change and Constant Blocks	20-12
Guidelines 2.6: Data Types	20-12

Guidelines for Speed and Area Optimizations - By	
Numbered List	20-14
Guidelines 3.1: Resource Sharing	20-14
Guidelines 3.2: Clock Rate Pipelining and Distributed Pipelining	20-15
Basic Guidelines for Modeling HDL Algorithm in Simulink	
.....	20-16
Use HDL-Supported Blocks	20-16
Partition Model into DUT and Test Bench	20-18
Avoid Using Double-Byte Characters	20-19
Document Model Features and Attributes	20-20
Guidelines for Model Setup and Checking Model	
Compatibility	20-23
Customize hdlsetup Function Based on Target Application	20-23
Check Subsystem for HDL Compatibility	20-24
Run Model Checks for HDL Coder	20-25
Modeling with Simulink, Stateflow, and MATLAB Function	
Blocks	20-28
Guideline ID	20-28
Severity	20-28
Description	20-28
Terminate Unconnected Block Outputs and Usage of	
Commenting Blocks	20-32
Terminate Unconnected Block Outputs	20-32
Using Comment Out and Comment Through of Blocks	20-34
Identify and Programmatically Change and Display HDL	
Block Parameters	20-38
Adjust Sizes of Constant and Gain Blocks for Identifying Parameters	20-38
Display Parameters that Affect HDL Code Generation	20-39
Change Block Parameters by Using find_system and set_param	20-43
DUT Subsystem Guidelines	20-45
DUT Subsystem Considerations	20-45

Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks	20-46
Insert Handwritten Code into Simulink Modeling Environment	20-48
Hierarchical Modeling Guidelines	20-51
Avoid Constant Block Connections to Subsystem Port Boundaries	20-51
Generate Parameterized HDL Code for Constant and Gain Blocks	20-52
Design Considerations for Matrices and Vectors	20-56
Modeling Requirements for Matrices	20-56
Avoid Generating Ascending Bit Order in HDL Code From Vector Signals	20-58
Use Bus Signals to Improve Readability of Model and Generate HDL Code	20-62
Guidelines for Clock and Reset Signals	20-70
Use Global Oversampling to Create Frequency-Divided Clock	20-70
Create Multirate Model with Integer Clock Multiples by Clock Division	20-70
Use Dual Rate Dual Port RAM for Noninteger Multiple Sample Times	20-74
Asynchronous Clock Modeling in HDL Coder	20-75
Use Global Reset Type Setting Based on Target Hardware	20-77
Modeling with Native Floating Point	20-79
Guideline ID	20-79
Severity	20-79
Description	20-79
Design Considerations for RAM Blocks and Blocks in HDL Operations Library	20-82
RAM Block Access Considerations	20-82
Serial to Parallel Conversion	20-85
Usage of Blocks in Logic and Bit Operations Library ..	20-87
Logical and Arithmetic Bit Shift Operations	20-87

Usage of Logical Operator, Bitwise Operator, and Bit Reduce Blocks	20-90
Use Boolean Data Type for Compare to Constant and Relational Operator Blocks	20-93
Generate FPGA Block RAM from Lookup Tables	20-95
Usage of Different Subsystem Types	20-99
Virtual Subsystem: Use as Top-Level DUT	20-99
Atomic Subsystem: Generate Reusable HDL Files	20-99
Variant Subsystem: Using Variant Subsystems for HDL Code Generation	20-100
Model References: Build Model Design Using Smaller Partitions	20-102
Block Settings of Enabled and Triggered Subsystems	20-104
Usage of Rate Change and Constant Blocks	20-107
Usage of Rate Conversion Blocks	20-107
Use Discrete and Finite Sample Time for Constant Block	20-109
Simulink Data Type Considerations	20-111
Use Boolean for Logical Data and Ufix1 for Numerical Data	20-111
Specify Data Type of Gain Blocks	20-111
Enumerated Data Type Restrictions	20-112
Resource Sharing Settings for Various Blocks	20-114
Resource Sharing of Add Blocks	20-114
Resource Sharing of Gain Blocks	20-115
Resource Sharing of Product Blocks	20-116
Resource Sharing of Multiply-Add Blocks	20-117
Resource Sharing of Subsystems and Floating-Point IPs	20-119
General Considerations for Sharing of Subsystems ..	20-119
Use MATLAB Datapath Architecture for Sharing with MATLAB Function Blocks	20-120
Sharing of Atomic Subsystems	20-121
Resource Sharing of Floating-Point IPs	20-123

Distributed Pipelining and Clock-Rate Pipelining	
Guidelines	20-125
Clock-Rate Pipelining Guidelines	20-125
Recommended Distributed Pipelining Settings	20-126
Insert Distributed Pipeline Registers for Blocks with	
Vector Data Type Inputs	20-129
Guideline ID	20-129
Severity	20-129
Description	20-129

21 Supported Blocks Library and Block Properties

View HDL-Specific Block Documentation	21-2
HDL Block Properties: General	21-3
Overview	21-4
AdaptivePipelining	21-4
BalanceDelays	21-5
ClockRatePipelining	21-6
CodingStyle	21-7
ConstMultiplierOptimization	21-8
ConstrainedOutputPipeline	21-9
DistributedPipelining	21-10
DotProductStrategy	21-11
DSPStyle	21-12
FlattenHierarchy	21-15
InputPipeline	21-16
InstantiateFunctions	21-17
InstantiateStages	21-18
LoopOptimization	21-18
LUTRegisterResetType	21-19
MapPersistentVarsToRAM	21-20
OutputPipeline	21-22
RAMDirective	21-23
ResetType	21-26
SerialPartition	21-28
SharingFactor	21-29
SoftReset	21-29

StreamingFactor	21-31
UseMatrixTypesInHDL	21-31
UsePipelines	21-33
UseRAM	21-34
VariablesToPipeline	21-38
HDL Block Properties: Native Floating Point	21-39
Overview	21-39
CheckResetToZero	21-39
DivisionAlgorithm	21-40
HandleDenormals	21-42
InputRangeReduction	21-43
LatencyStrategy	21-44
NFPCustomLatency	21-46
MantissaMultiplyStrategy	21-47
MaxIterations	21-49
HDL Filter Block Properties	21-51
AdderTreePipeline	21-51
AddPipelineRegisters	21-51
ChannelSharing	21-52
CoeffMultipliers	21-52
DALUTPartition	21-53
DARadix	21-55
FoldingFactor	21-55
MultiplierInputPipeline	21-56
MultiplierOutputPipeline	21-56
NumMultipliers	21-56
ReuseAccum	21-57
SerialPartition	21-57
HDL Filter Architectures	21-60
Fully Parallel Architecture	21-60
Serial Architectures	21-62
Frame-Based Architecture	21-64
Distributed Arithmetic for HDL Filters	21-68
Requirements and Considerations for Generating Distributed Arithmetic Code	21-69
Further References	21-69
Set and View HDL Model and Block Parameters	21-71
Set HDL Block Parameters	21-71

Set HDL Block Parameters for Multiple Blocks	
Programmatically	21-72
View All HDL Block Parameters	21-73
View Non-Default HDL Block Parameters	21-74
View HDL Model Parameters	21-74
Pass through, No HDL, and Cascade Implementations	
.	21-76
Pass-through and No HDL Implementations	21-76
Cascade Architecture Best Practices	21-76
Build a ROM Block with Simulink Blocks	21-77
Wireless Communications Design for FPGAs and ASICs	
.	21-78
From Mathematical Algorithm to Hardware	
Implementation	21-78
HDL-Optimized Blocks	21-81
Reference Applications	21-81
Generate HDL Code and Prototype on FPGA	21-82

22

Generating HDL Code for Multirate Models

Code Generation from Multirate Models	22-2
Clock Enable Generation for a Multirate DUT	22-2
Timing Controller for Multirate Models	22-5
Generate Reset for Timing Controller	22-6
Requirements for Timing Controller Reset Port Generation	
.	22-6
How To Generate Reset for Timing Controller	22-6
Limitations for Timing Controller Reset Port Generation	
.	22-6
Multirate Model Requirements for HDL Code Generation	
.	22-7
Model Configuration Parameters	22-7
Sample Rate	22-7

Blocks To Use For Rate Transitions	22-8
Generate a Global Oversampling Clock	22-9
Why Use a Global Oversampling Clock?	22-9
Requirements for the Oversampling Factor	22-9
Specifying the Oversampling Factor From the GUI	22-10
Specifying the Oversampling Factor From the Command Line	22-11
Resolving Oversampling Rate Conflicts	22-11
Using Triggered Subsystems for HDL Code Generation	22-15
Requirements	22-15
Best Practices	22-15
Using the Signal Builder Block	22-16
Using Trigger As Clock	22-16
Specify Trigger As Clock	22-16
Limitations	22-17
Generate Multicycle Path Information Files	22-19
Overview	22-19
Format and Content of a Multicycle Path Information File	22-20
File Naming and Location Conventions	22-25
Generating Multicycle Path Information Files Using the GUI	22-25
Generating Multicycle Path Information Files Using the Command Line	22-25
Limitations	22-26
Using Multiple Clocks in HDL Coder™	22-28

Generating Bit-True Cycle-Accurate Models

23

Locate Numeric Differences After Speed Optimization	23-2
---	-------------

Automatic Iterative Optimization	24-2
How Automatic Iterative Optimization Works	24-2
Automatic Iterative Optimization Output	24-3
Automatic Iterative Optimization Report	24-3
Requirements for Automatic Iterative Optimization	24-4
Limitations of Automatic Iterative Optimization	24-4
Generated Model and Validation Model	24-5
Generated Model	24-5
Validation Model	24-6
Simplify Constant Operations and Reduce Design Complexity in HDL Coder™	24-9
Optimization with Constrained Overclocking	24-15
Why Constrain Overclocking?	24-15
Optimizations that Overclock Resources	24-15
How to Use Constrained Overclocking	24-16
Constrained Overclocking Limitations	24-16
Resolve Numerical Mismatch with Delay Balancing ..	24-17
Streaming	24-23
What Is Streaming?	24-23
Specify Streaming	24-24
How to Determine Streaming Factor and Sample Time	24-24
Determine Blocks That Support Streaming	24-25
Requirements for Streaming Subsystems	24-25
Streaming Report	24-25
Resource Sharing	24-27
How Resource Sharing Works	24-27
Benefits and Costs of Resource Sharing	24-28
Shareable Resources in Different Blocks	24-28
Specify Resource Sharing	24-29
Limitations for Resource Sharing	24-29
Block Requirements for Resource Sharing	24-29
Resource Sharing Report	24-30

Delay Balancing	24-32
Why Use Delay Balancing?	24-32
Specify Delay Balancing	24-33
Delay Balancing Limitations	24-34
Delay Balancing Report	24-37
Find Feedback Loops	24-38
Specify Highlighting of Feedback Loops	24-38
Remove Highlighting	24-39
Limitations	24-39
Hierarchy Flattening	24-40
What Is Hierarchy Flattening?	24-40
When to Flatten Hierarchy	24-40
Considerations	24-40
How to Flatten Hierarchy	24-41
Limitations for Hierarchy Flattening	24-42
RAM Mapping for Simulink Models	24-43
RAM Mapping With the MATLAB Function Block	24-44
Distributed Pipelining	24-49
What Is Distributed Pipelining?	24-49
Benefits and Costs of Distributed Pipelining	24-51
Requirements for Distributed Pipelining	24-52
Specify Distributed Pipelining	24-52
Limitations of Distributed Pipelining	24-52
Distributed Pipelining Report	24-54
Selected Bibliography	24-54
Hierarchical Distributed Pipelining	24-56
What Is Hierarchical Distributed Pipelining?	24-56
How Hierarchical Distributed Pipelining Works	24-56
Benefits of Hierarchical Distributed Pipelining	24-59
Specify Hierarchical Distributed Pipelining	24-59
Limitations of Hierarchical Distributed Pipelining	24-60
Hierarchical Distributed Pipelining Report	24-60
Selected Bibliography	24-60
Constrained Output Pipelining	24-61
What Is Constrained Output Pipelining?	24-61
When to Use Constrained Output Pipelining	24-61

Requirements for Constrained Output Pipelining	24-61
Specify Constrained Output Pipelining	24-62
Limitations of Constrained Output Pipelining	24-62
Clock-Rate Pipelining	24-63
Rationale for Clock-Rate Pipelining	24-63
How Clock-Rate Pipelining Works	24-64
Clock-Rate Pipelining and Hierarchy Flattening	24-64
Clock-Rate Pipelining for DUT Output Ports	24-65
Best Practices for Clock-Rate Pipelining	24-65
Specify Clock-Rate Pipelining	24-66
Limitations for Clock-Rate Pipelining	24-66
Adaptive Pipelining	24-68
Requirements	24-68
Specify Adaptive Pipelining	24-69
Supported Blocks	24-69
Pipeline Insertion for Lookup Tables	24-70
Pipeline Insertion for Rate Transition and Downsample Blocks	24-71
Pipeline Insertion for Product and Gain Blocks	24-71
Pipeline Insertion for Multiply-Add and Multiply- Accumulate Blocks	24-73
Pipeline Insertion for MATLAB Function Blocks	24-75
Adaptive Pipelining Report	24-76
Critical Path Estimation Without Running Synthesis	24-78
How Critical Path Estimation Works	24-79
How to Use Critical Path Estimation	24-81
Characterized Blocks	24-82
Caveats	24-87
HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture	24-89
Subsystem Optimizations for Filters	24-102
Sharing	24-102
Streaming	24-102
Pipelining	24-103
Area Reduction of Filter Subsystem	24-103
Area Reduction of Multichannel Filter Subsystem	24-107

Remove Redundant Logic and Optimize Unconnected Ports in Design	24-114
--	---------------

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

25

Create and Use Code Generation Reports	25-2
Report Generation	25-2
Code Generation Report	25-2
Summary	25-2
Code Interface Report	25-3
Timing and Area Report	25-3
Optimization Report	25-3
 Navigate Between Simulink Model and HDL Code by Using Traceability	25-5
How Traceability Works	25-5
Generate Traceability Report	25-6
Report Location	25-7
View the Traceability Report	25-7
Code-to-Model Navigation	25-9
Model-to-Code Navigation	25-10
Traceability Report Limitations	25-11
 Web View of Model in Code Generation Report	25-13
About Model Web View	25-13
Generate HTML Code Generation Report with Model Web View	25-13
Model Web View Limitations	25-15
 Generate Code with Annotations or Comments	25-16
Simulink Annotations	25-16
Signal Descriptions	25-16
Text Comments	25-17
Requirements Comments and Hyperlinks	25-17
 Check Your Model for HDL Compatibility	25-21

Show Blocks Supported for HDL Code Generation	25-23
Show Supported Blocks in Library Browser	25-23
Reset Library Browser to Show All Blocks	25-25
Generate a Supported Blocks Report	25-26
Trace Code Using the Mapping File	25-27
Add or Remove the HDL Configuration Component . . .	25-30
What Is the HDL Configuration Component?	25-30
Adding the HDL Coder Configuration Component To a Model	25-30
Removing the HDL Coder Configuration Component From a Model	25-31

HDL Coding Standards

26

HDL Coding Standard Report	26-2
Rule Summary	26-2
Rule Hierarchy	26-3
Rule and Report Customization	26-3
How to Fix Warnings and Errors	26-3
HDL Coding Standards	26-4
Generate an HDL Coding Standard Report from Simulink	26-6
Using the HDL Workflow Advisor	26-6
Using the Command Line	26-8
Basic Coding Practices	26-10
1.A General Naming Conventions	26-11
1.B General Guidelines for Clocks and Resets	26-20
1.C Guidelines for Initial Reset	26-21
1.D Guidelines for Clocks	26-22
1.F Guidelines for Hierarchical Design	26-24
RTL Description Techniques	26-25
2.A Guidelines for Combinational Logic	26-26

2.B Guidelines for “Always” Constructs of Combinational Logic	26-35
2.C Guidelines for Flip-Flop Inference	26-37
2.D Guidelines for Latch Description	26-43
2.E Guidelines for Tristate Buffer	26-44
2.F Guidelines for Always/Process Construct with Circuit Structure into Account	26-46
2.G Guidelines for “IF” Statement Description	26-47
2.H Guidelines for “CASE” Statement Description	26-50
2.I Guidelines for “FOR” Statement Description	26-54
2.J Guidelines for Operator Description	26-56
2.K Guidelines for Finite State Machine Description	26-62
RTL Design Methodology Guidelines	26-64
3.A Guidelines for Creating Function Libraries	26-64
3.B Guidelines for Using Function Libraries	26-66
3.C Guidelines for Test Facilitation Design	26-69
Generate an HDL Lint Tool Script	26-71
How to Generate an HDL Lint Tool Script	26-71

27 Interfacing Subsystems and Models to HDL Code

Model Referencing for HDL Code Generation	27-2
Benefits of Model Referencing for Code Generation	27-2
How To Generate Code for a Referenced Model	27-2
Generate Code for Model Arguments	27-3
Generate Comments	27-3
Limitations	27-3
Generate Black Box Interface for Subsystem	27-5
What Is a Black Box Interface?	27-5
Generate a Black Box Interface for a Subsystem	27-5
Generate Code for a Black Box Subsystem Implementation	27-8
Restriction for Multirate DUTs	27-9
Generate Black Box Interface for Referenced Model	27-10
When to Generate a Black Box Interface	27-10

How to Generate a Black Box Interface	27-10
Caveats and Limitations	27-11
Integrate Custom HDL Code Using DocBlock	27-12
When To Use DocBlock to Integrate Custom Code	27-12
How To Use DocBlock to Integrate Custom Code	27-12
Restrictions	27-13
Example	27-13
Customize Black Box or HDL Cosimulation Interface	
.	27-14
Interface Parameters	27-14
Specify Bidirectional Ports	27-18
Requirements	27-18
How To Specify a Bidirectional Port	27-18
Limitations	27-19
Generate Reusable Code for Atomic Subsystems	27-20
Requirements for Generating Reusable Code for Atomic Subsystems	27-20
Generate Reusable Code for Atomic Subsystems	27-21
Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters	27-23
Create a Xilinx System Generator Subsystem	27-28
Why Use Xilinx System Generator Subsystems?	27-28
Requirements for Xilinx System Generator Subsystems	27-28
How to Create a Xilinx System Generator Subsystem	27-29
Limitations for Code Generation from Xilinx System Generator Subsystems	27-29
Create an Altera DSP Builder Subsystem	27-31
Why Use Altera DSP Builder Subsystems?	27-31
Requirements for Altera DSP Builder Subsystems	27-31
How to Create an Altera DSP Builder Subsystem	27-32
Determine Clocking Requirements for Altera DSP Builder Subsystems	27-32
Limitations for Code Generation from Altera DSP Builder Subsystems	27-33

Using Xilinx System Generator for DSP with HDL Coder	27-34
.....	
Choose a Test Bench for Generated HDL Code	27-38
Generate a Cosimulation Model	27-41
What Is A Cosimulation Model?	27-41
Generating a Cosimulation Model from the GUI	27-42
Structure of the Generated Model	27-48
Launching a Cosimulation	27-54
The Cosimulation Script File	27-57
Complex and Vector Signals in the Generated Cosimulation Model	27-59
Generating a Cosimulation Model from the Command Line	27-60
.....	
Naming Conventions for Generated Cosimulation Models and Scripts	27-60
Limitations for Cosimulation Model Generation	27-61
Pass-Through and No-Op Implementations	27-62
Synchronous Subsystem Behavior with the State Control Block	27-63
What Is a State Control Block?	27-63
State Control Block Modes	27-63
Synchronous Badge for Subsystems by Using Synchronous Mode	27-64
Generate HDL Code with the State Control Block	27-66
Enable and Reset Hardware Simulation Behavior	27-68

Stateflow HDL Code Generation Support

28

Introduction to Stateflow HDL Code Generation	28-2
Overview	28-2
Comments	28-2
Tunable Parameters	28-3
Example	28-3
Restrictions	28-3

Hardware Realization of Stateflow Semantics	28-8
Generate HDL for Mealy and Moore Finite State Machines	
.	28-9
Overview	28-9
Generating HDL Code for a Moore Finite State Machine	
.	28-9
Generating HDL for a Mealy Finite State Machine . . .	28-13
Design Patterns Using Advanced Chart Features	28-17
Temporal Logic	28-17
Graphical Function	28-19
Hierarchy and Parallelism	28-21
Stateless Charts	28-21
Truth Tables	28-23
Initialize Persistent Variables in MATLAB Functions . .	28-28
MATLAB Function Block With No Direct Feedthrough	
.	28-29
State Control Block in Synchronous Mode	28-30
Stateflow Chart Implementing Moore Semantics	28-32

Generating HDL Code with the MATLAB Function Block

29

HDL Applications for the MATLAB Function Block	29-2
Structure of Generated HDL Code	29-2
HDL Applications	29-2
Viterbi Decoder with the MATLAB Function Block	29-4
Code Generation from a MATLAB Function Block	29-5
Counter Model Using the MATLAB Function block	29-5
Setting Up	29-7
Creating the Model and Configuring General Model	
Settings	29-8
Adding a MATLAB Function Block to the Model	29-9
Set Fixed-Point Options for the MATLAB Function Block	
.	29-9

Programming the MATLAB Function Block	29-13
Constructing and Connecting the DUT_eML_Block Subsystem	29-13
Compiling the Model and Displaying Port Data Types	29-16
Simulating the eml_hdl_incrementer_tut Model	29-17
Generating HDL Code	29-18
Generate Instantiable Code for Functions	29-21
How To Generate Instantiable Code for Functions	29-21
Generate Code Inline for Specific Functions	29-22
Limitations for Instantiable Code Generation for Functions	29-22
MATLAB Function Block Design Patterns for HDL	29-23
The eml_hdl_design_patterns Library	29-23
Efficient Fixed-Point Algorithms	29-25
Model State Using Persistent Variables	29-28
Creating Intellectual Property with the MATLAB Function Block	29-29
Nontunable Parameter Arguments	29-30
Modeling Control Logic and Simple Finite State Machines	29-30
Modeling Counters	29-32
Modeling Hardware Elements	29-33
Design Guidelines for the MATLAB Function Block	29-35
Use Compiled External Functions With MATLAB Function Blocks	29-35
Build the MATLAB Function Block Code First	29-35
Use the hdlfmath Utility for Optimized FIMATH Settings	29-35
Use Optimal Fixed-Point Option Settings	29-36
Set the Output Data Type of MATLAB Function Blocks Explicitly	29-36
Using Tunable Parameters	29-36
Run HDL Model Check for MATLAB Function Blocks	29-37
Use MATLAB Datapath Architecture for Enhanced HDL Optimizations	29-37
Distributed Pipeline Insertion for MATLAB Function Blocks	29-39

Generating Scripts for HDL Simulators and Synthesis Tools

30

Generate Scripts for Compilation, Simulation, and Synthesis	30-2
Structure of Generated Script Files	30-3
Properties for Controlling Script Generation	30-4
Enabling and Disabling Script Generation	30-4
Customizing Script Names	30-4
Customizing Script Code	30-5
Examples	30-7
Configure Compilation, Simulation, Synthesis, and Lint Scripts	30-8
Compilation Script Options	30-9
Simulation Script Options	30-11
Synthesis Script Options	30-13
Add Synthesis Attributes	30-18
Configure Synthesis Project Using Tcl Script	30-19

Using the HDL Workflow Advisor

31

Getting Started with the HDL Workflow Advisor	31-2
Open the HDL Workflow Advisor	31-2
Run Tasks in the HDL Workflow Advisor	31-3
Fix HDL Workflow Advisor Warnings or Failures	31-4
Save and Restore the HDL Workflow Advisor State	31-5
View and Save HDL Workflow Advisor Reports	31-7
Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor	31-10

Generate HDL Code for FPGA Floating-Point Target	
Libraries	31-14
Setup for FPGA Floating-Point Library Mapping	31-14
Map to an FPGA Floating-Point Library	31-15
View Code Generation Reports of Floating-Point Library Mapping	31-18
Analyze Results of Floating-Point Library Mapping	31-19
Customize Floating-Point IP Configuration	31-23
Customize the IP Latency with Target Frequency	31-24
Customize the IP Latency with Latency Strategy	31-28
HDL Coder Support for FPGA Floating-Point Library Mapping	31-34
Supported Blocks That Map to FPGA Floating-Point Target IP	31-34
Supported Blocks That Do Not Need to Map to FPGA Floating-Point Target IP	31-37
Limitations for FPGA Floating-Point Library Mapping	31-38
Synthesis Objective to Tcl Command Mapping	31-39
Altera Quartus II	31-39
Xilinx Vivado 2014.4	31-40
Xilinx ISE 14.7 with PlanAhead	31-40
Run HDL Workflow with a Script	31-42
Export an HDL Workflow Script	31-43
Enable or Disable Tasks in HDL Workflow Script	31-43
Run a Single Workflow Task	31-43
Import an HDL Workflow Script	31-44
Generic ASIC/FPGA Workflow Script Example	31-44
FPGA-in-the-Loop Script Example	31-46
FPGA Turnkey Workflow Script Example	31-48
IP Core Generation Workflow Script Example	31-51
Simulink Real-Time FPGA I/O Workflow Example	31-54

32

Generate HDL Code from Simscape Models	32-2
Deploy Simscape™ Plant Models to Speedgoat FPGA I/O Modules	32-14
Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model	32-27
Troubleshoot Conversion of Simscape™ Permanent Magnet Synchronous Motor to HDL-Compatible Simulink Model	32-38
Validate HDL Implementation Model to Simscape Algorithm	32-55
Bridge Rectifier Model	32-55
Increase Validation Logic Tolerance	32-58
Increase Number of Solver Iterations	32-59
Use Larger Floating-Point Precision	32-61
Improve Sampling Rate of HDL Implementation Model Generated from Simscape Algorithm	32-64
Sampling Frequency	32-64
Boost Converter Model	32-64
Reducing Number of Solver Iterations	32-66
Using Oversampling Factor and Latency Strategy	32-67

Simscape HDL Workflow Advisor Tasks

33

Simscape HDL Workflow Advisor Tasks	33-2
Simscape HDL Workflow Advisor folder	33-2
Code generation compatibility folder	33-2
Check solver configuration task	33-3
Check switched linear task	33-3
State-space conversion folder	33-4
Extract Equations	33-4

Discretize Equations	33-5
Implementation model generation folder	33-5
Generate implementation model task	33-5

Simscape HDL Workflow Advisor Tips and Guidelines	33-7
Estimating Resource Consumption Using Algebraic and Differential Variables	33-7
Setting Simulation Stop Time for Extracting Equations	33-8
Changing Sample Time for Discretizing Equations ...	33-10
Using Number of Solver Iterations	33-11
Floating-Point Precision and Numerical Accuracy	33-13

Model Protection in HDL Coder

34

Create Protected Models to Conceal Contents and Generate HDL Code	34-2
How Model Protection Works	34-2
How to Create a Protected Model	34-2
General Protected Model Requirements and Limitations	34-3
Protected Model Restrictions for HDL Code Generation	34-4
Prepare the Parent Model	34-4
Protect the Referenced Model	34-6
Protected Model Report	34-8
Generate HDL Code for Models Referencing Protected Model	34-10
Test Protected Models	34-12
Package and Share Protected Models	34-16
Harness Model	34-16
MAT-File with Base Workspace Definitions	34-16
Simulink Data Dictionary	34-17
Protected Model File Contents	34-17

35

Test Bench Generation	35-2
How Test Bench Generation Works	35-2
Test Bench Data Files	35-2
Test Bench Data Type Limitations	35-3
Use Constants Instead of File I/O	35-3
 Test Bench Block Restrictions	 35-5

FPGA Board Customization

36

FPGA Board Customization	36-2
Feature Description	36-2
Custom Board Management	36-2
FPGA Board Requirements	36-3
 Create Custom FPGA Board Definition	 36-8
 Create Xilinx KC705 Evaluation Board Definition File	
.....	36-9
Overview	36-9
What You Need to Know Before Starting	36-9
Start New FPGA Board Wizard	36-10
Provide Basic Board Information	36-11
Specify FPGA Interface Information	36-13
Enter FPGA Pin Numbers	36-14
Run Optional Validation Tests	36-16
Save Board Definition File	36-18
Use New FPGA Board	36-19
 FPGA Board Manager	 36-22
Introduction	36-22
Filter	36-24
Search	36-24
FIL Enabled/Turnkey Enabled	36-24
Create Custom Board	36-24

Add Board from File	36-24
Get More Boards	36-24
View/Edit	36-25
Remove	36-25
Clone	36-25
Validate	36-25
New FPGA Board Wizard	36-26
Basic Information	36-27
Interfaces	36-28
FIL I/O	36-31
Turnkey I/O	36-34
Validation	36-37
Finish	36-38
FPGA Board Editor	36-39
General Tab	36-39
Interface Tab	36-41

HDL Workflow Advisor Tasks

37

HDL Workflow Advisor Tasks	37-2
HDL Workflow Advisor Tasks Overview	37-4
Set Target Overview	37-5
Set Target Device and Synthesis Tool	37-5
Set Target Reference Design	37-6
Set Target Interface	37-7
Set Target Frequency	37-7
Set Target Interface	37-8
Set Target Interface	37-9
Prepare Model For HDL Code Generation Overview ..	37-10
Check Global Settings	37-11
Check Algebraic Loops	37-11
Check Block Compatibility	37-12
Check Sample Times	37-12
Check FPGA-In-The-Loop Compatibility	37-13
HDL Code Generation Overview	37-13
Set Code Generation Options Overview	37-14
Set Basic Options	37-14

Set Report Options	37-15
Set Advanced Options	37-15
Set Optimization Options	37-15
Set Testbench Options	37-15
Generate RTL Code	37-16
Generate RTL Code and Testbench	37-16
Verify with HDL Cosimulation	37-17
Generate RTL Code and IP Core	37-17
FPGA Synthesis and Analysis Overview	37-18
Create Project	37-19
Perform Synthesis and P/R Overview	37-20
Perform Logic Synthesis	37-21
Perform Mapping	37-21
Perform Place and Route	37-22
Run Synthesis	37-22
Run Implementation	37-22
Annotate Model with Synthesis Result	37-23
Download to Target Overview	37-24
Generate Programming File	37-24
Program Target Device	37-25
Generate Simulink Real-Time Interface	37-25
Save and Restore HDL Workflow Advisor State	37-25
FPGA-In-The-Loop Implementation	37-25
Set FPGA-In-The-Loop Options	37-25
Build FPGA-In-The-Loop	37-26
Check USRP® Compatibility	37-26
Generate FPGA Implementation	37-26
Check SDR Compatibility	37-27
SDR FPGA Implementation	37-27
Set SDR Options	37-27
Build SDR	37-29
Embedded System Integration	37-29
Create Project	37-29
Generate Software Interface Model	37-30
Build FPGA Bitstream	37-30
Program Target Device	37-31

HDL Coder Checks in Model Advisor / HDL Model Checker	
Overview	38-3
Model configuration checks overview	38-5
Check for safe model parameters	38-6
Description	38-6
Results and Recommended Actions	38-7
See Also	38-7
Check for global reset setting for Xilinx and Altera devices	38-8
Description	38-8
Results and Recommended Actions	38-8
See Also	38-8
Check inline configurations setting	38-9
Description	38-9
Results and Recommended Actions	38-9
See Also	38-9
Check algebraic loops	38-10
Description	38-10
Results and Recommended Actions	38-10
See Also	38-10
Check for visualization settings	38-11
Description	38-11
Results and Recommended Actions	38-11
See Also	38-11
Check delay balancing setting	38-12
Description	38-12
Results and Recommended Actions	38-12
See Also	38-12
Check for ports and subsystems overview	38-13

Check for invalid top level subsystem	38-14
Description	38-14
Results and Recommended Actions	38-14
Check initial conditions of enabled and triggered subsystems	38-15
Description	38-15
Results and Recommended Actions	38-15
See Also	38-15
Check for blocks and block settings overview	38-16
Check for infinite and continuous sample time sources	38-17
Description	38-17
Results and Recommended Actions	38-17
See Also	38-17
Check for unsupported blocks	38-18
Description	38-18
Results and Recommended Actions	38-18
Check for large matrix operations	38-19
Description	38-19
Results and Recommended Actions	38-19
See Also	38-19
Check for MATLAB Function block settings	38-20
Description	38-20
Results and Recommended Actions	38-20
See Also	38-20
Check for Stateflow chart settings	38-21
Description	38-21
Results and Recommended Actions	38-21
See Also	38-21
Check for obsolete Unit Delay Enabled/Resettable Blocks	38-22
Description	38-22
Results and Recommended Actions	38-22

Check for unsupported storage class for signal objects	38-23
Description	38-23
Results and Recommended Actions	38-23
Native Floating Point Checks Overview	38-24
Check for single datatypes in the model	38-25
Description	38-25
Results and Recommended Actions	38-25
See Also	38-25
Check for double datatypes in the model with Native Floating Point	38-26
Description	38-26
Results and Recommended Actions	38-26
See Also	38-26
Check for Data Type Conversion blocks with incompatible settings	38-27
Description	38-27
Results and Recommended Actions	38-27
See Also	38-27
Check for HDL Reciprocal block usage	38-28
Description	38-28
Results and Recommended Actions	38-28
See Also	38-28
Check for Relational Operator block usage	38-29
Description	38-29
Results and Recommended Actions	38-29
See Also	38-29
Check for unsupported blocks with Native Floating Point	38-30
Description	38-30
Results and Recommended Actions	38-30
See Also	38-30
Check for blocks with nonzero output latency	38-31
Description	38-31
Results and Recommended Actions	38-31

See Also	38-31
Check blocks with nonzero ulp error	38-32
Description	38-32
Results and Recommended Actions	38-32
See Also	38-32
Industry standard checks overview	38-33
Check VHDL file extension	38-34
Description	38-34
Results and Recommended Actions	38-34
See Also	38-34
Check naming conventions	38-35
Description	38-35
Results and Recommended Actions	38-35
See Also	38-35
Check top-level subsystem/port names	38-36
Description	38-36
Results and Recommended Actions	38-36
See Also	38-36
Check module/entity names	38-37
Description	38-37
Results and Recommended Actions	38-37
See Also	38-37
Check signal and port names	38-38
Description	38-38
Results and Recommended Actions	38-38
See Also	38-38
Check package file names	38-39
Description	38-39
Results and Recommended Actions	38-39
See Also	38-39
Check generics	38-40
Description	38-40
Results and Recommended Actions	38-40
See Also	38-40

Check clock, reset, and enable signals	38-41
Description	38-41
Results and Recommended Actions	38-41
See Also	38-41
Check architecture name	38-42
Description	38-42
Results and Recommended Actions	38-42
See Also	38-42
Check entity and architecture	38-43
Description	38-43
Results and Recommended Actions	38-43
See Also	38-43
Check clock settings	38-44
Description	38-44
Results and Recommended Actions	38-44
See Also	38-44

Using the HDL Model Checker

39

Getting Started with the HDL Model Checker	39-2
Open the HDL Model Checker	39-2
Run Checks In the HDL Model Checker	39-3
Fix HDL Model Checker Warnings or Failures	39-4
View and Save HDL Model Checker Reports	39-5
Run Model Advisor Checks for HDL Coder	39-7
Open the Model Advisor Checks	39-7
Run Checks in the Model Advisor	39-7
Run Checks In Background	39-8
Display Check Results in the Model Advisor Report ...	39-9
Fix Warnings or Failures	39-10
Save and Restore Model Advisor State	39-12
Model Checks in HDL Coder	39-13
Model configuration checks	39-14
Checks for ports and subsystems	39-15

Checks for blocks and block settings	39-15
Native Floating Point checks	39-17
industry standard checks	39-17

Hardware-Software Codesign

40	Hardware-Software Co-Design Basics	
	Custom IP Core Generation	40-2
	Custom IP Core Architectures	40-2
	Target Platform Interfaces	40-3
	Processor/FPGA Synchronization	40-3
	Custom IP Core Generated Files	40-4
	Custom IP Core Report	40-5
	Summary	40-5
	Target Interface Configuration	40-6
	Register Address Mapping	40-6
	IP Core User Guide	40-7
	IP Core File List	40-10
	Generate Board-Independent HDL IP Core from Simulink Model	40-12
	Generate Board-Independent IP Core	40-12
	IP Core without AXI4 Slave Interfaces	40-15
	Requirements and Limitations for IP Core Generation	40-16
	Processor and FPGA Synchronization	40-18
	Free Running Mode	40-18
	Coprocesing - Blocking Mode	40-19
	Coprocesing - Nonblocking With Delay Mode	40-19
	Synchronization of Global Reset Signal to IP Core Clock Domain	40-21

IP Caching for Faster Reference Design Synthesis . . .	40-26
Requirements for Using IP Caching	40-26
What Is an IP Cache?	40-26
How IP Caching Works	40-27
Enable IP Caching	40-28
IP Caching in HDL Coder Reference Designs	40-29
IP Caching in Custom Reference Designs	40-32
Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows	40-34
Step 1: Identify the Timing Failure	40-34
Step 2: Find the Critical Path	40-37
Step 3: Resolve Timing Failures	40-42
Define and Add IP Repository to Custom Reference Design	40-47
Create an IP Repository Folder Structure	40-47
Define IP List Function	40-49
Add IP List Function to Reference Design Project	40-50
Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface	40-52
Vivado-Based Reference Designs	40-53
Qsys-Based Reference Designs	40-55
Meet Timing Requirements Using Enable-Based Multicycle Path Constraints	40-58
How Enable-Based Multicycle Path Constraints Work	40-58
Specify Enable-Based Constraints	40-60
Benefits of Using Enable-Based Constraints	40-61
Modeling Guidelines	40-61
Multicycle Path Constraints for Various Synthesis Tools	40-62
Caveats and Limitations	40-63
Program Target FPGA Boards or SoC Devices	40-65
How to Program Target Device	40-65
Programming Methods	40-67

Hardware-Software Co-Design Workflow for SoC Platforms	41-2
.....	
Model Design for AXI4 Slave Interface Generation . . .	41-11
Considerations	41-11
Map Scalar Ports to AXI4 Slave Interface	41-12
Map Vector Ports to AXI4 Slave Interface	41-13
Read Back Value of AXI4 Slave Interfaces	41-14
Optimize AXI4 Slave Read Back Logic	41-17
Model Design for AXI4-Stream Interface Generation .	41-19
Simplified Streaming Protocol	41-19
Map Scalar Ports To AXI4-Stream Interface	41-20
Map Vector Ports To AXI4-Stream Interface	41-23
Model Designs with Multiple Sample Rates	41-25
Restrictions	41-25
Multirate IP Core Generation	41-27
Board and Reference Design Registration System	41-33
Board, IP Core, and Reference Design Definitions	41-33
Board Registration Files	41-34
Reference Design Registration Files	41-35
Predefined Board and Reference Design Examples . . .	41-36
Register a Custom Board	41-37
Define a Board	41-37
Create a Board Plugin	41-38
Define a Board Registration Function	41-39
Register a Custom Reference Design	41-41
Define a Reference Design	41-41
Create a Reference Design Plugin	41-42
Define a Reference Design Registration Function	41-43
Define Custom Parameters and Callback Functions for	
Custom Reference Design	41-45
Define Custom Parameters and Register Callback Function	
Handle	41-45

Define Custom Callback Functions	41-51
FPGA Programming and Configuration	41-52
Model Design for AXI4-Stream Video Interface Generation	
.....	41-64
Streaming Pixel Protocol	41-64
Protocol Signals and Timing Diagrams	41-65
Model Data and Control Bus Signals	41-67
Model Designs with Multiple Sample Rates	41-71
Video Porch Insertion Logic	41-72
Default Video System Reference Design	41-73
Restrictions	41-74
Model Design for AXI4 Master Interface Generation ..	41-75
Simplified AXI4 Master Protocol - Write Channel	41-75
Simplified AXI4 Master Protocol - Read Channel	41-77
Base Address Register Calculation	41-78
Modeling for AXI4 Master Interfaces	41-80
Model Designs with Multiple Sample Rates	41-83
Reference Designs for IP Core Integration	41-83
Restrictions	41-85
IP Core Generation Workflow for Standalone FPGA	
Devices	41-86
Targeting FPGA Reference Designs with AXI4 Interface	
.....	41-88
Targeting FPGA Reference Designs Without AXI4 Interface	
.....	41-89
Board Support	41-89
IP Core Generation Workflow for Speedgoat I/O Modules	
.....	41-91

HDL Code Generation from MATLAB

Functions Supported for HDL Code Generation

- “Functions Supported for HDL Code Generation — Alphabetical List” on page 1-2
- “Functions Supported for HDL Code Generation — Categorical List” on page 1-10

Functions Supported for HDL Code Generation — Alphabetical List

You can generate efficient HDL code for a subset of MATLAB built-in functions and toolbox functions that you call from MATLAB code. These functions appear in alphabetical order in the following table.

To find supported functions by MATLAB category or toolbox, see “Functions Supported for HDL Code Generation — Categorical List” on page 1-10.

Name	Product	Remarks and Limitations
abs	Fixed-Point Designer™	Double and complex data types not supported.
add	Fixed-Point Designer	—
all	MATLAB	Double data type not supported.
and	MATLAB	—
any	MATLAB	Double data type not supported.
bitand	MATLAB	—
bitand	Fixed-Point Designer	—
bitandreduce	Fixed-Point Designer	—
bitcmp	MATLAB	—
bitcmp	Fixed-Point Designer	—
bitconcat	Fixed-Point Designer	—
bitget	MATLAB	—
bitget	Fixed-Point Designer	—
bitor	MATLAB	—
bitor	Fixed-Point Designer	—
bitorreduce	Fixed-Point Designer	—
bitreplicate	Fixed-Point Designer	—
bitrol	Fixed-Point Designer	—
bitror	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
bitset	MATLAB	—
bitset	Fixed-Point Designer	—
bitshift	MATLAB	For efficient HDL code generation, use the Fixed-Point Designer functions <code>bitsll</code> , <code>bitsrl</code> , or <code>bitsra</code> instead of <code>bitshift</code> .
bitshift	Fixed-Point Designer	—
bitsliceget	Fixed-Point Designer	—
bitsll	Fixed-Point Designer	—
bitsra	Fixed-Point Designer	—
bitsrl	Fixed-Point Designer	—
bitxor	MATLAB	—
bitxor	Fixed-Point Designer	—
bitxorreduce	Fixed-Point Designer	—
ceil	Fixed-Point Designer	—
complex	MATLAB	—
complex	Fixed-Point Designer	—
conj	Fixed-Point Designer	—
convergent	Fixed-Point Designer	—
ctranspose	MATLAB	—
ctranspose	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
divide	Fixed-Point Designer	<ul style="list-style-type: none">• For HDL Code generation, the divisor must be a constant and a power of two.• Non-<code>fi</code> inputs must be constant; that is, their values must be known at compile time so that they can be cast to <code>fi</code> objects.• Complex and imaginary divisors are not supported.• Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.
end	Fixed-Point Designer	—
eps	Fixed-Point Designer	<ul style="list-style-type: none">• Supported for scalar fixed-point signals only.• Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
eq	MATLAB	—
eq	Fixed-Point Designer	—
fi	Fixed-Point Designer	—
fimath	Fixed-Point Designer	—
fix	Fixed-Point Designer	—
floor	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
for	MATLAB	<p>Do not use for loops without static bounds.</p> <p>Do not use the & and operators within conditions of a for statement. Instead, use the && and operators.</p> <p>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.</p>
ge	MATLAB	—
ge	Fixed-Point Designer	—
getlsb	Fixed-Point Designer	—
getmsb	Fixed-Point Designer	—
gt	MATLAB	—
gt	Fixed-Point Designer	—
horzcat	Fixed-Point Designer	—
if	MATLAB	<p>Do not use the & and operators within conditions of an if statement. Instead, use the && and operators.</p> <p>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.</p>
imag	MATLAB	—
int8, int16, int32	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
iscolumn	MATLAB	—
isempty	MATLAB	—
isequal	Fixed-Point Designer	—
isfi	Fixed-Point Designer	—
isfimath	Fixed-Point Designer	—
isfimathlocal	Fixed-Point Designer	—
isfinite	MATLAB	—
isinf	MATLAB	—
isnan	MATLAB	—
isnumeric	MATLAB	—
isnumericitype	Fixed-Point Designer	—
isreal	MATLAB	—
isrow	MATLAB	—
isscalar	MATLAB	—
assigned	Fixed-Point Designer	—
isvector	MATLAB	—
le	MATLAB	—
le	Fixed-Point Designer	—
length	MATLAB	—
logical	MATLAB	—
lowerbound	Fixed-Point Designer	—
lsb	Fixed-Point Designer	—
lt	MATLAB	—
lt	Fixed-Point Designer	—
max	Fixed-Point Designer	—
min	Fixed-Point Designer	—
minus	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
<code>mpower</code>	MATLAB	Both inputs must be scalar, and the exponent input, <code>k</code> , must be an integer.
<code>mpower</code>	Fixed-Point Designer	Both inputs must be scalar, and the exponent input, <code>k</code> , must be a constant integer.
<code>mtimes(A,B)</code>	MATLAB	—
<code>mtimes</code>	Fixed-Point Designer	—
<code>ndims</code>	MATLAB	—
<code>ne</code>	MATLAB	—
<code>ne</code>	Fixed-Point Designer	—
<code>nearest</code>	Fixed-Point Designer	—
<code>not</code>	MATLAB	—
<code>numerictype</code>	Fixed-Point Designer	—
<code>ones</code>	MATLAB	Dimensions must be real, nonnegative integers.
<code>or</code>	MATLAB	—
<code>plus</code>	MATLAB	Inputs cannot be data type <code>logical</code> .
<code>plus</code>	Fixed-Point Designer	Inputs cannot be data type <code>logical</code> .
<code>power</code>	MATLAB	Both inputs must be scalar, and the exponent input, <code>k</code> , must be an integer.
<code>power</code>	Fixed-Point Designer	Both inputs must be scalar, and the exponent input, <code>k</code> , must be a constant integer.
<code>range</code>	Fixed-Point Designer	—
<code>real</code>	MATLAB	—
<code>realmax</code>	Fixed-Point Designer	—
<code>realmin</code>	Fixed-Point Designer	—

1 Functions Supported for HDL Code Generation

Name	Product	Remarks and Limitations
reinterprecast	Fixed-Point Designer	—
repmat	MATLAB	—
rescale	Fixed-Point Designer	—
reshape	MATLAB	—
round	Fixed-Point Designer	—
sfi	Fixed-Point Designer	—
sign	Fixed-Point Designer	—
size	MATLAB	—
sqrt	Fixed-Point Designer	—
sub	Fixed-Point Designer	—
subasgn	Fixed-Point Designer	Supported data types for HDL code generation are listed in “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 2-2.
subref	Fixed-Point Designer	Supported data types for HDL code generation are listed in “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 2-2.
sum	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
switch	MATLAB	The conditional expression in a switch or case statement must use only: <ul style="list-style-type: none"> • 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.
times	MATLAB	Inputs cannot be data type logical.
times	Fixed-Point Designer	Inputs cannot be data type logical.
transpose	MATLAB	—
transpose	MATLAB	—
ufi	Fixed-Point Designer	—
uint8, uint16, uint32	Fixed-Point Designer	—
uminus	Fixed-Point Designer	—
uplus	MATLAB	Inputs cannot be data type logical.
upperbound	Fixed-Point Designer	—
vertcat	Fixed-Point Designer	—
xor	MATLAB	—
zeros	MATLAB	Dimensions must be real, nonnegative integers.

Functions Supported for HDL Code Generation — Categorical List

In this section...
“Arithmetic Operations in MATLAB” on page 1-11
“Bitwise Operations in MATLAB” on page 1-11
“Complex Numbers in MATLAB” on page 1-11
“Control Flow in MATLAB” on page 1-12
“Logical Operators in MATLAB” on page 1-12
“Arrays in MATLAB” on page 1-13
“Relational Operators in MATLAB” on page 1-13
“ Fixed-Point Designer ” on page 1-13

In this section...
“Arithmetic Operations in MATLAB” on page 1-11
“Bitwise Operations in MATLAB” on page 1-11
“Complex Numbers in MATLAB” on page 1-11
“Control Flow in MATLAB” on page 1-12
“Logical Operators in MATLAB” on page 1-12
“Arrays in MATLAB” on page 1-13
“Relational Operators in MATLAB” on page 1-13
“ Fixed-Point Designer ” on page 1-13

You can generate efficient HDL code for a subset of MATLAB built-in functions and toolbox functions that you call from MATLAB code. These functions are listed by MATLAB category or toolbox category in the following tables.

For an alphabetical list of supported functions, see “Functions Supported for HDL Code Generation — Alphabetical List” on page 1-2.

Arithmetic Operations in MATLAB

Name	Remarks and Limitations
<code>ctranspose(A)</code>	—
<code>mpower(A,B)</code>	A and B must be scalar, and B must be an integer.
<code>mtimes(A,B)</code>	—
<code>plus(A,B)</code>	Neither A nor B can be data type logical.
<code>power(A,B)</code>	A and B must be scalar, and B must be an integer.
<code>times(A,B)</code>	Neither A nor B can be data type logical.
<code>transpose(A)</code>	—

Bitwise Operations in MATLAB

Name	Remarks and Limitations
<code>bitand</code>	—
<code>bitcmp</code>	—
<code>bitget</code>	—
<code>bitor</code>	—
<code>bitset</code>	—
<code>bitshift</code>	For efficient HDL code generation, use the Fixed-Point Designer functions <code>bitsll</code> , <code>bitsrl</code> , or <code>bitsra</code> instead of <code>bitshift</code> .
<code>bitxor</code>	—

Complex Numbers in MATLAB

Name	Remarks and Limitations
<code>complex</code>	—
<code>imag</code>	—
<code>real</code>	—

Control Flow in MATLAB

Name	Remarks and Limitations
for	<p>Do not use for loops without static bounds.</p> <p>Do not use the & and operators within conditions of a for statement. Instead, use the && and operators.</p> <p>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.</p>
if	<p>Do not use the & and operators within conditions of an if statement. Instead, use the && and operators.</p> <p>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.</p>
switch	<p>The conditional expression in a switch or case statement must use only:</p> <ul style="list-style-type: none"> • uint8, uint16, uint32, int8, int16, or int32 data types • Scalar data <p>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.</p>

Logical Operators in MATLAB

Name	Remarks and Limitations
and	—
not	—
or	—
xor	—

Arrays in MATLAB

Name	Remarks and Limitations
ones	Dimensions must be real, nonnegative integers.
zeros	Dimensions must be real, nonnegative integers.

Relational Operators in MATLAB

Name	Remarks and Limitations
eq	—
ge	—
gt	—
le	—
lt	—
ne	—

Fixed-Point Designer

HDL code generation support for fixed-point run-time library functions from the Fixed-Point Designer is summarized in the following table. See “Fixed-Point Function Limitations” on page 2-35 for general limitations of fixed-point run-time library functions for code generation.

Function	Remarks and Limitations
abs	Double and complex data types not supported.
add	—
bitand	—
bitandreduce	—
bitcmp	—
bitconcat	—
bitget	—
bitor	—

Function	Remarks and Limitations
bitorreduce	—
bitreplicate	—
bitrol	—
bitror	—
bitset	—
bitshift	—
bitsliceget	—
bitsll	—
bitsra	—
bitsrl	—
bitxor	—
bitxorreduce	—
ceil	—
complex	—
conj	—
convergent	—
ctranspose	—
divide	<ul style="list-style-type: none"> • For HDL Code generation, the divisor must be a constant and a power of two. • Non-<code>fi</code> inputs must be constant; that is, their values must be known at compile time so that they can be cast to <code>fi</code> objects. • Complex and imaginary divisors are not supported. • Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.
eps	<ul style="list-style-type: none"> • Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
eq	—
fi	—

Function	Remarks and Limitations
fimath	—
fix	—
floor	—
ge	—
getlsb	—
getmsb	—
gt	—
horzcat	—
int8, int16, int32	—
isequal	—
isfi	—
isfimath	—
isfimathlocal	—
isnumerictype	—
issigned	—
le	—
lowerbound	—
lsb	—
lt	—
max	—
min	—
minus	—
mpower	Both inputs must be scalar, and the exponent input, k, must be a constant integer.
mtimes	—
nearest	—
numerictype	—
ones	Dimensions must be real, nonnegative integers.

Function	Remarks and Limitations
plus	Inputs cannot be data type logical.
power	Both inputs must be scalar, and the exponent input, k, must be a constant integer.
range	—
realmax	—
realmin	—
reinterpretcast	—
rescale	—
round	—
sfi	—
sign	—
sqrt	—
sub	—
subsasgn	Supported data types for HDL code generation are listed in “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 2-2.
subsref	Supported data types for HDL code generation are listed in “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 2-2.
sum	—
times	Inputs cannot be data type logical.
ufi	—
uint8, uint16, uint32	—
uminus	—
upperbound	—
vertcat	—

MATLAB Algorithm Design

- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 2-2
- “Persistent Variables and Persistent Array Variables” on page 2-9
- “Complex Data Type Support” on page 2-11
- “HDL Code Generation for System Objects” on page 2-15
- “Predefined System Objects Supported for HDL Code Generation” on page 2-18
- “Load constants from a MAT-File” on page 2-21
- “Generate Code for User-Defined System Objects” on page 2-22
- “Map Matrices to ROM” on page 2-25
- “Fixed-Point Bitwise Functions” on page 2-26
- “Fixed-Point Run-Time Library Functions” on page 2-32
- “Model State with Persistent Variables and System Objects” on page 2-37
- “Bitwise Operations in MATLAB for HDL Code Generation” on page 2-41
- “Guidelines for Efficient HDL Code” on page 2-46
- “MATLAB Design Requirements for HDL Code Generation” on page 2-47
- “What Is a MATLAB Test Bench?” on page 2-48
- “MATLAB Test Bench Requirements and Best Practices” on page 2-49

Supported MATLAB Data Types, Operators, and Control Flow Statements

In this section...

“Supported Data Types” on page 2-2

“Supported Operators” on page 2-4

“Control Flow Statements” on page 2-6

When you generate HDL 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	<ul style="list-style-type: none"> <code>uint8</code>, <code>uint16</code>, <code>uint32</code>, <code>uint64</code> <code>int8</code>, <code>int16</code>, <code>int32</code>, <code>int64</code> 	In Simulink®, MATLAB Function block ports must use numeric types <code>sfix64</code> or <code>ufix64</code> for 64-bit data.

Types	Supported Data Types	Restrictions
Real	<ul style="list-style-type: none"> • <code>double</code> • <code>single</code> 	<p>HDL code generated with <code>double</code> or <code>single</code> 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:</p> <ul style="list-style-type: none"> • “Simulink Blocks Supported with Native Floating-Point” on page 10-106 • “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92 • “Signal and Data Type Support” on page 10-2
Character	<code>char</code>	-
Logical	<code>logical</code>	-
Fixed point	<ul style="list-style-type: none"> • Scaled (binary point only) fixed-point numbers • Custom integers (zero binary point) 	<p>Fixed-point numbers with slope (not equal to 1.0) and bias (not equal to 0.0) are not supported.</p> <p>Maximum word size for fixed-point numbers is 128 bits.</p>
Vectors	<ul style="list-style-type: none"> • <code>unordered {N}</code> • <code>row {1, N}</code> • <code>column {N, 1}</code> 	<p>The maximum number of vector elements allowed is 2^{32}.</p> <p>Before a variable is subscripted, it must be fully defined.</p>
Matrices	<code>{N, M}</code>	<p>Matrices are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.</p> <p>Do not use matrices in the testbench.</p>

Types	Supported Data Types	Restrictions
Structures	struct	<p>Arrays of structures are not supported.</p> <p>For the FPGA Turnkey and IP Core Generation workflows, structures are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.</p>
Enumerations	enumeration	<p>Enumeration values must be monotonically increasing.</p> <p>If your target language is Verilog®, all enumeration member names must be unique within the design.</p> <p>Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods:</p> <ul style="list-style-type: none"> • IP Core Generation workflow • FPGA Turnkey workflow • FPGA-in-the-Loop • HDL Cosimulation

Global variables are not supported for HDL code generation.

Supported Operators

Note HDL 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.

Control Flow Statement	Restrictions
for	<p>Do not use for loops without static bounds.</p> <p>Do not use the & and operators within conditions of a for statement. Instead, use the && and operators.</p> <p>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.</p>
if	<p>Do not use the & and operators within conditions of an if statement. Instead, use the && and operators.</p> <p>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.</p>
switch	<p>The conditional expression in a switch or case statement must use only:</p> <ul style="list-style-type: none"> • uint8, uint16, uint32, int8, int16, or int32 data types • Scalar data <p>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.</p>

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

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 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 to RAM” on page 8-3.

Complex Data Type Support

In this section...

“Declaring Complex Signals” on page 2-11

“Conversion Between Complex and Real Signals” on page 2-12

“Support for Vectors of Complex Numbers” on page 2-13

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_SFHDH 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_SFHDH;
```

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 DOWNT0 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 both VHDL and Verilog 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;
```

HDL Code Generation for System Objects

In this section...

“Why Use System Objects?” on page 2-15

“Predefined System Objects” on page 2-15

“User-Defined System Objects” on page 2-16

“Limitations of HDL Code Generation for System Objects” on page 2-16

“System object Examples for HDL Code Generation” on page 2-17

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 2-18.

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 2-22.

Limitations of HDL Code Generation for System Objects

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

- Your design can call the `step` method only once per System object.
- `step` must not be inside a nested conditional statement, such as a nested loop, `if` statement, or `switch` statement.
- `step` must not be 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.
- `UseMatrixTypesInHDL` attribute must be set to ‘off’, if you have a System object in your MATLAB code.
- Enumerations are not supported.
- Global variables are not supported.

Supported Methods

For predefined System Objects, `step` is the only method supported for HDL code generation.

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 `step` method 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 5-15
- “Model State with Persistent Variables and System Objects” on page 2-37
- “Generate Code for User-Defined System Objects” on page 2-22
- “Integrate Custom HDL Code Into MATLAB Design” on page 5-28

Predefined System Objects Supported for HDL Code Generation

In this section...
“Predefined System Objects in MATLAB Code” on page 2-18
“Predefined System Objects in the MATLAB System Block” on page 2-20

Predefined System Objects in MATLAB Code

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

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

HDL Coder supports the following Communications Toolbox System objects for HDL code generation:

- `comm.BPSKModulator`, `comm.BPSKDemodulator`
- `comm.PSKModulator`, `comm.PSKDemodulator`
- `comm.QPSKModulator`, `comm.QPSKDemodulator`
- `comm.ConvolutionalInterleaver`, `comm.ConvolutionalDeinterleaver`
- `comm.ViterbiDecoder`
- `comm.HDLCRCDetector`, `comm.HDLCRCGenerator`
- `comm.HDLRSDecoder`, `comm.HDLRSEncoder`

HDL Coder supports the following DSP System Toolbox System objects for HDL code generation:

- `dsp.Delay`
- `dsp.BiquadFilter`
- `dsp.DCBlocker`
- `dsp.HDLComplexToMagnitudeAngle`
- `dsp.HDLFIRRateConverter`
- `dsp.HDLFFT`, `dsp.HDLIFFT`

- `dsp.HDLChannelizer`
- `dsp.HDLNCO`
- `dsp.FIRFilter`
- `dsp.HDLFIRFilter`

HDL Coder supports the following Vision HDL Toolbox™ System objects for HDL code generation:

- `visionhdl.BilateralFilter`
- `visionhdl.BirdsEyeView`
- `visionhdl.ChromaResampler`
- `visionhdl.ColorSpaceConverter`
- `visionhdl.DemosaicInterpolator`
- `visionhdl.EdgeDetector`
- `visionhdl.GammaCorrector`
- `visionhdl.LookupTable`
- `visionhdl.Histogram`
- `visionhdl.ImageStatistics`
- `visionhdl.ROISelector`
- `visionhdl.LineBuffer`
- `visionhdl.PixelStreamAligner`
- `visionhdl.ImageFilter`
- `visionhdl.MedianFilter`
- `visionhdl.Closing`
- `visionhdl.Dilation`
- `visionhdl.Erosion`
- `visionhdl.Opening`
- `visionhdl.GrayscaleClosing`
- `visionhdl.GrayscaleDilation`
- `visionhdl.GrayscaleErosion`
- `visionhdl.GrayscaleOpening`

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” (Simulink) 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;
```

Generate Code for User-Defined System Objects

In this section...
“How To Create A User-Defined System object” on page 2-22
“User-Defined System object Example” on page 2-22

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`

To use the `outputImpl` and `updateImpl` methods, your System object must also inherit from the `matlab.system.mixin.Nondirect` class.

- 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 `step` method, or 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
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 = step(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 2-15

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 **RAM Mapping Threshold** value.

To learn how to set the RAM mapping threshold in Simulink, see the **RAM mapping threshold (bits)** section in “RAM Mapping” on page 14-5. To learn how to set the RAM mapping threshold in MATLAB, see “How To Enable RAM Mapping” on page 8-3.

- 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.

Fixed-Point Bitwise Functions

The following table summarizes bitwise functions in MATLAB and Fixed-Point Designer that are supported for HDL code generation. The following conventions are used in the table:

- **a, b**: Denote fixed-point integer operands.
- **idx**: Denotes an index to a bit within an operand. Indexes can be scalar or vector, depending on the function.

MATLAB code uses 1-based indexing conventions. In generated HDL code, such indexes are converted to zero-based indexing conventions.

- **lidx, ridx**: denote indexes to the left and right boundaries delimiting bit fields. Indexes can be scalar or vector, depending on the function.
- **val**: Denotes a Boolean value.

Note Indexes, operands, and values passed as arguments bitwise functions can be scalar or vector, depending on the function. For information on the individual functions, see “Bitwise Operations” (Fixed-Point Designer).

MATLAB Syntax	Description	See Also
<code>bitand(a, b)</code>	Bitwise AND	<code>bitand</code>
<code>bitandreduce(a, lidx, ridx)</code>	Bitwise AND of a field of consecutive bits within <code>a</code> . The field is delimited by <code>lidx</code> , <code>ridx</code> . Output data type: <code>ufix1</code> For VHDL, generates the bitwise AND operator operating on a set of individual slices For Verilog, generates the reduce operator: <code>&a[lidx:ridx]</code>	<code>bitandreduce</code>
<code>bitcmp(a)</code>	Bitwise complement	<code>bitcmp</code>

MATLAB Syntax	Description	See Also
<code>bitconcat(a, b)</code> <code>bitconcat([a_vector])</code> <code>bitconcat(a, b,c,d,...)</code>	<p>Concatenate fixed-point operands.</p> <p>Operands can be of different signs.</p> <p>Output data type: <code>ufixN</code>, where <code>N</code> is the sum of the word lengths of <code>a</code> and <code>b</code>.</p> <p>For VHDL, generates the concatenation operator: <code>(a & b)</code></p> <p>For Verilog, generates the concatenation operator: <code>{a , b}</code></p>	<code>bitconcat</code>
<code>bitget(a,idx)</code>	<p>Access a bit at position <code>idx</code>.</p> <p>For VHDL, generates the slice operator: <code>a(idx)</code></p> <p>For Verilog, generates the slice operator: <code>a[idx]</code></p>	<code>bitget</code>
<code>bitor(a, b)</code>	<p>Bitwise OR</p>	<code>bitor</code>
<code>bitorreduce(a, lidx, ridx)</code>	<p>Bitwise OR of a field of consecutive bits within <code>a</code>. The field is delimited by <code>lidx</code> and <code>ridx</code>.</p> <p>Output data type: <code>ufix1</code></p> <p>For VHDL, generates the bitwise OR operator operating on a set of individual slices.</p> <p>For Verilog, generates the reduce operator: <code> a[lidx:ridx]</code></p>	<code>bitorreduce</code>
<code>bitset(a, idx, val)</code>	<p>Set or clear bit(s) at position <code>idx</code>.</p> <p>If <code>val = 0</code>, clears the indicated bit(s). Otherwise, sets the indicated bits.</p>	<code>bitset</code>
<code>bitreplicate(a, n)</code>	<p>Concatenate bits of <code>fi</code> object <code>a</code> <code>n</code> times</p>	<code>bitreplicate</code>

MATLAB Syntax	Description	See Also
<code>bitrol(a, idx)</code>	<p>Rotate left.</p> <p><code>idx</code> must be a positive integer. The value of <code>idx</code> can be greater than the word length of <code>a</code>. <code>idx</code> is normalized to <code>mod(idx, wlen)</code>. <code>wlen</code> is the word length of <code>a</code>.</p> <p>For VHDL, generates the <code>rol</code> operator.</p> <p>For Verilog, generates the following expression (where <code>wl</code> is the word length of <code>a</code>):</p> <pre>a << idx a >> wl - idx</pre>	<code>bitrol</code>
<code>bitror(a, idx)</code>	<p>Rotate right.</p> <p><code>idx</code> must be a positive integer. The value of <code>idx</code> can be greater than the word length of <code>a</code>. <code>idx</code> is normalized to <code>mod(idx, wlen)</code>. <code>wlen</code> is the word length of <code>a</code>.</p> <p>For VHDL, generates the <code>ror</code> operator.</p> <p>For Verilog, generates the following expression (where <code>wl</code> is the word length of <code>a</code>):</p> <pre>a >> idx a << wl - idx</pre>	<code>bitror</code>
<code>bitset(a, idx, val)</code>	<p>Set or clear bit(s) at position <code>idx</code>.</p> <p>If <code>val = 0</code>, clears the indicated bit(s). Otherwise, sets the indicated bits.</p>	<code>bitset</code>

MATLAB Syntax	Description	See Also
bitshift(a, idx)	<p>Note: For efficient HDL code generation, use <code>bitsll</code>, <code>bitsrl</code>, or <code>bitsra</code> instead of <code>bitshift</code>.</p> <p>Shift left or right, based on the positive or negative integer value of <code>idx</code>.</p> <p><code>idx</code> must be an integer.</p> <p>For positive values of <code>idx</code>, shift left <code>idx</code> bits.</p> <p>For negative values of <code>idx</code>, shift right <code>idx</code> bits.</p> <p>If <code>idx</code> is a variable, generated code contains logic for both left shift and right shift.</p> <p>Result values saturate if the <code>overflowMode</code> of <code>a</code> is set to <code>saturate</code>.</p>	bitshift
bitsliceget(a, lidx, ridx)	<p>Access consecutive set of bits from <code>lidx</code> to <code>ridx</code>.</p> <p>Output data type: <code>ufixN</code>, where $N = \text{lidx} - \text{ridx} + 1$.</p>	bitsliceget
bitsll(a, idx)	<p>Shift left logical.</p> <p><code>idx</code> must be a scalar within the range $0 \leq \text{idx} < w_l$</p> <p><code>w_l</code> is the word length of <code>a</code>.</p> <p>Overflow and rounding modes of input operand <code>a</code> are ignored.</p> <p>Generates <code>sll</code> operator in VHDL.</p> <p>Generates <code><<</code> operator in Verilog.</p>	bitsll

MATLAB Syntax	Description	See Also
<code>bitsra(a, idx)</code>	<p>Shift right arithmetic.</p> <p><code>idx</code> must be a scalar within the range</p> $0 \leq \text{idx} < w_l$ <p><code>w_l</code> is the word length of <code>a</code>,</p> <p>Overflow and rounding modes of input operand <code>a</code> are ignored.</p> <p>Generates <code>sra</code> operator in VHDL.</p> <p>Generates <code>>>></code> operator in Verilog.</p>	<code>bitsra</code>
<code>bitsrl(a, idx)</code>	<p>Shift right logical.</p> <p><code>idx</code> must be a scalar within the range</p> $0 \leq \text{idx} < w_l$ <p><code>w_l</code> is the word length of <code>a</code>.</p> <p>Overflow and rounding modes of input operand <code>a</code> are ignored.</p> <p>Generates <code>srl</code> operator in VHDL.</p> <p>Generates <code>>></code> operator in Verilog.</p>	<code>bitsrl</code>
<code>bitxor(a, b)</code>	Bitwise XOR	<code>bitxor</code>

MATLAB Syntax	Description	See Also
<code>bitxorreduce(a, lidx, ridx)</code>	<p>Bitwise XOR reduction.</p> <p>Bitwise XOR of a field of consecutive bits within <code>a</code>. The field is delimited by <code>lidx</code> and <code>ridx</code>.</p> <p>Output data type: <code>ufix1</code></p> <p>For VHDL, generates a set of individual slices.</p> <p>For Verilog, generates the reduce operator: <code>^a[lidx:ridx]</code></p>	<code>bitxorreduce</code>
<code>getlsb(a)</code>	Return value of LSB.	<code>getlsb</code>
<code>getmsb(a)</code>	Return value of MSB.	<code>getmsb</code>

See Also

Related Examples

- “Bitwise Operations in MATLAB for HDL Code Generation” on page 2-41

More About

- “Bitwise Operations” (Fixed-Point Designer)
- “Fixed-Point Run-Time Library Functions” on page 2-32

Fixed-Point Run-Time Library Functions

HDL code generation support for fixed-point run-time library functions from the Fixed-Point Designer is summarized in the following table. See “Fixed-Point Function Limitations” on page 2-35 for general limitations of fixed-point run-time library functions for code generation.

Function	Remarks and Limitations
abs	Double and complex data types not supported.
add	—
bitand	—
bitandreduce	—
bitcmp	—
bitconcat	—
bitget	—
bitor	—
bitorreduce	—
bitreplicate	—
bitrol	—
bitror	—
bitset	—
bitshift	—
bitsliceget	—
bitsll	—
bitsra	—
bitsrl	—
bitxor	—
bitxorreduce	—
ceil	—
complex	—
conj	—

Function	Remarks and Limitations
convergent	—
ctranspose	—
divide	<ul style="list-style-type: none"> • For HDL Code generation, the divisor must be a constant and a power of two. • Non-<code>fi</code> inputs must be constant; that is, their values must be known at compile time so that they can be cast to <code>fi</code> objects. • Complex and imaginary divisors are not supported. • Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.
eps	<ul style="list-style-type: none"> • Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
eq	—
fi	—
fimath	—
fix	—
floor	—
ge	—
getlsb	—
getmsb	—
gt	—
horzcat	—
int8, int16, int32	—
isequal	—
isfi	—
isfimath	—
isfimathlocal	—
isnumerictype	—
issigned	—

Function	Remarks and Limitations
le	—
lowerbound	—
lsb	—
lt	—
max	—
min	—
minus	—
mpower	Both inputs must be scalar, and the exponent input, k, must be a constant integer.
mtimes	—
nearest	—
numerictype	—
ones	Dimensions must be real, nonnegative integers.
plus	Inputs cannot be data type logical.
power	Both inputs must be scalar, and the exponent input, k, must be a constant integer.
range	—
realmax	—
realmin	—
reinterprecast	—
rescale	—
round	—
sfi	—
sign	—
sqrt	—
sub	—
subasgn	Supported data types for HDL code generation are listed in “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 2-2.

Function	Remarks and Limitations
suboref	Supported data types for HDL code generation are listed in “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 2-2.
sum	—
times	Inputs cannot be data type logical.
ufi	—
uint8, uint16, uint32	—
uminus	—
upperbound	—
vertcat	—

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 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” (Fixed-Point Designer) for more information.

See Also

Related Examples

- “Bitwise Operations in MATLAB for HDL Code Generation” on page 2-41

More About

- “Bitwise Operations” (Fixed-Point Designer)
- “Fixed-Point Bitwise Functions” on page 2-26

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

Let us take a quick 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'  
};
```

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_delay_modeling'];
```

```
% create a temporary folder and copy the MATLAB files
```

```
cd(tempdir);
```

```
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
```

```
mkdir(mlhdlc_temp_dir);
```

```
cd(mlhdlc_temp_dir);
```

```
for ii=1:numel(demo_files)
```

```
    copyfile(fullfile(mlhdlc_demo_dir, [demo_files{ii}, '.m*']), mlhdlc_temp_dir);
```

```
end
```

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.

You can refer to the Getting Started with MATLAB to HDL Workflow tutorial 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 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'.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_delay_modeling'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```


Bitwise Operations in MATLAB for HDL 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

Bit Shifting and Rotation Algorithms

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
```

Generated Code

In VHDL code generated for this function, the shift and rotate functions map directly to shift and rotate instructions in VHDL.

```
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;
```

The corresponding Verilog code is similar, except that Verilog does not have native operators for rotate instructions.

```
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

```

Bit Slicing and Bit Concatenation

The `bitsliceget` and `bitconcat` functions map directly to slice and concatenate operators in both VHDL and Verilog.

Bit Slicing and Concatenation Algorithm

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));
```

Generated Code

The following listing shows the corresponding generated VHDL 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;
```

The following listing shows the corresponding generated Verilog code.

```
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
```

See Also

More About

- “Bitwise Operations” (Fixed-Point Designer)
- “Fixed-Point Bitwise Functions” on page 2-26
- “Fixed-Point Run-Time Library Functions” on page 2-32

Guidelines for Efficient HDL Code

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.

For better HDL code and faster code generation, design your MATLAB code according to the following 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 like 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 RAM for storage.
- 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; unroll fewer loops and enable the loop streaming optimization to conserve area at the cost of lower throughput.

MATLAB Design Requirements for HDL Code Generation

Your MATLAB design has the following requirements:

- MATLAB code within the design must be supported for HDL code generation.
- Inputs and outputs must not be matrices or structures.

If you are generating code from the command line, verify your code readiness for code generation with the following 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 the Fixed-Point Designer, see “MATLAB Algorithm Design”.

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 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 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” on page 2-49.

MATLAB Test Bench Requirements and Best Practices

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 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” (MATLAB Coder).
- 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
```

MATLAB Best Practices and Design Patterns for HDL Code Generation

- “Model a Counter for HDL Code Generation” on page 3-2
- “Model a State Machine for HDL Code Generation” on page 3-5
- “Generate Hardware Instances For Local Functions” on page 3-10
- “Implement RAM Using MATLAB Code” on page 3-13
- “For-Loop Best Practices for HDL Code Generation” on page 3-16

Model a Counter for HDL Code Generation

In this section...

- “MATLAB Counter” on page 3-2
- “MATLAB Code for the Counter” on page 3-3
- “Best Practices in this Example” on page 3-3

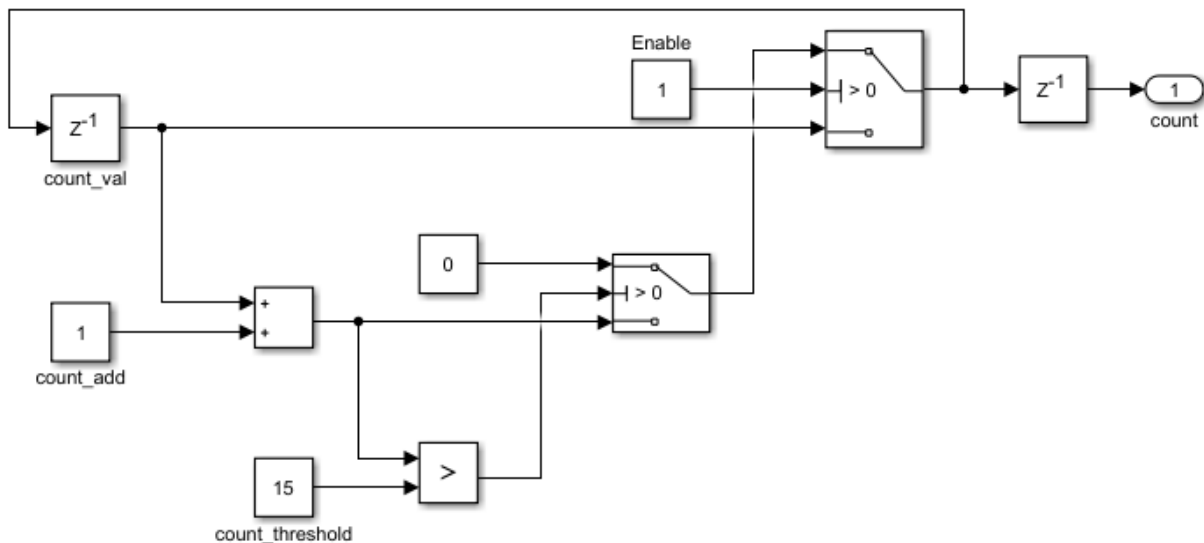
MATLAB Counter

This design pattern shows a MATLAB example of a counter, which is suitable for HDL code generation.

This model demonstrates the following best practices for writing MATLAB code to generate HDL code:

- Initialize persistent variables.
- Read persistent variables before they are modified.

This Simulink model illustrates the counter modeled in this example.



MATLAB Code for the Counter

The function `mlhdlc_counter` is a behavioral model of a four 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.

The following section of code defines the `mlhdlc_counter` function.

```

%#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

```

Best Practices in this Example

This design pattern demonstrates two best practices for writing MATLAB code for HDL code generation:

- Initialize persistent variables to a specific value. In this example, an `if` statement and the `isempty` function initialize the persistent variable. If the persistent variable is not initialized then HDL code cannot be generated.
- Inside a function, read persistent variables before they are modified, in order for the persistent variables to be inferred as registers.

Model a State Machine for HDL Code Generation

In this section...

“MATLAB State Machines” on page 3-5

“MATLAB Code for the Mealy State Machine” on page 3-5

“MATLAB Code for the Moore State Machine” on page 3-7

“Best Practices” on page 3-9

MATLAB State Machines

The following design pattern shows MATLAB examples of Mealy and Moore state machines which are suitable for HDL code generation.

The MATLAB code in these models demonstrates best practices for writing MATLAB models for HDL code generation.

- With a `switch` block, use the `otherwise` statement to account for all conditions.
- Use variables to designate states in a 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.

MATLAB Code for the Mealy State Machine

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.

```

%#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

end
```



```

    case S4,
        if (A)
            Z = true;
            current_state = S1;
        else
            Z = false;
            current_state = S3;
        end
    otherwise,
        Z = false;
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.

```

%#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
end
```

```
        end
    otherwise,
        Z = false;
end
```

Best Practices

This design pattern demonstrates two best practices for writing MATLAB code for HDL 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.

Generate Hardware Instances For Local Functions

In this section...
“MATLAB Local Functions” on page 3-10
“MATLAB Code for mlhdlc_two_counters.m” on page 3-10

MATLAB Local Functions

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. 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;
```

Implement RAM Using MATLAB Code

In this section...

“Implementation of RAM” on page 3-13

“Implement RAM Using a Persistent Array or System object Properties” on page 3-13

“Implement RAM Using hdl.RAM” on page 3-14

Implementation of RAM

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. However, one example uses a persistent array and the other uses an `hdl.RAM` System object to model the RAM behavior.

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.

For a comparison of the ways you can write MATLAB code to map to RAM during HDL code generation, and for an overview of the tradeoffs, see “RAM Mapping Comparison for MATLAB Code” on page 8-8. For more information, see “Map Persistent Arrays and `dsp.Delay` to RAM” on page 8-3.

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;

% execute single step of RAM

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`.

```

%#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;

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

rdAddrCtr = rdAddrCtr + 1;

data_out = ramRdDout;
```

hdl.RAM Restrictions for Code Generation

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 2-16.

For-Loop Best Practices for HDL Code Generation

In this section...
“MATLAB Loops” on page 3-16
“Monotonically Increasing Loop Counters” on page 3-16
“Persistent Variables in Loops” on page 3-17
“Persistent Arrays in Loops” on page 3-18

MATLAB Loops

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.
- 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
```

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
```

Fixed-Point Conversion

- “Floating-Point to Fixed-Point Conversion” on page 4-2
- “Fixed-Point Type Conversion and Refinement” on page 4-15
- “Working with Generated Fixed-Point Files” on page 4-25
- “Specify Type Proposal Options” on page 4-32
- “Log Data for Histogram” on page 4-36
- “View and Modify Variable Information” on page 4-38
- “Automated Fixed-Point Conversion” on page 4-41
- “Custom Plot Functions” on page 4-58
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 4-60
- “Inspecting Data Using the Simulation Data Inspector” on page 4-66
- “Enable Plotting Using the Simulation Data Inspector” on page 4-69
- “Replacing Functions Using Lookup Table Approximations” on page 4-71
- “Replace a Custom Function with a Lookup Table” on page 4-72
- “Replace the exp Function with a Lookup Table” on page 4-80
- “Data Type Issues in Generated Code” on page 4-88

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 that you have been introduced to in the tutorial Getting 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 more details on the requirements for the floating-point design and the testbench, refer to the 'Floating-Point Design Structure' structure section of the Working with Generated Fixed-Point Files tutorial.

```
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)')
```

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fix_prj'];  
  
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);
```

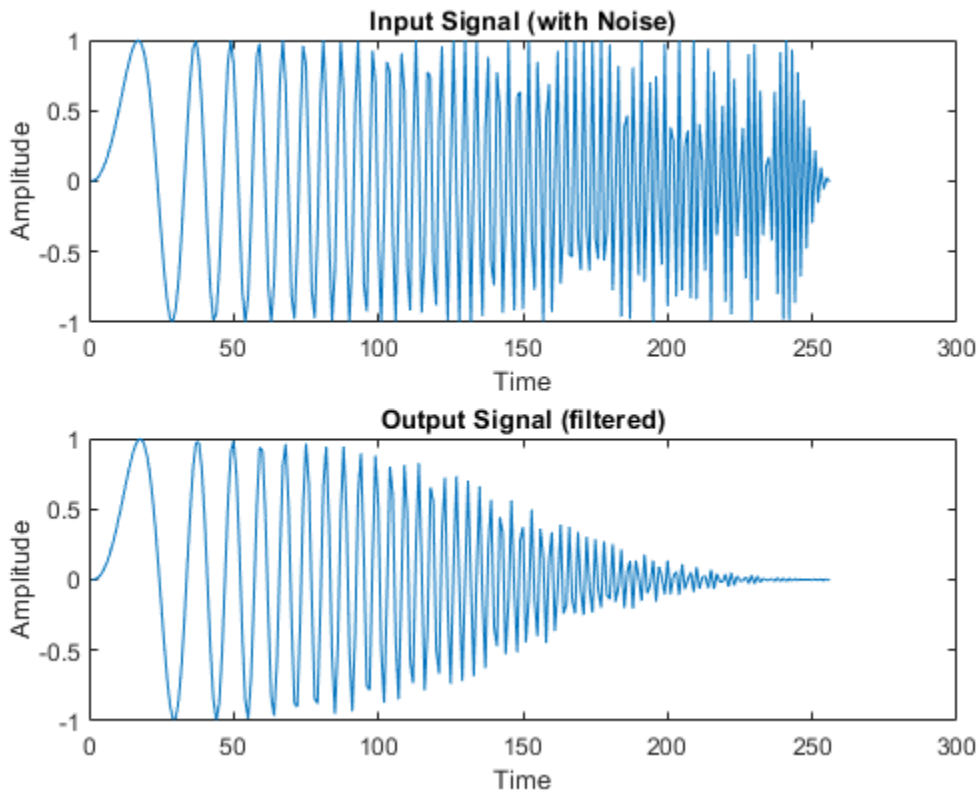


```
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

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.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

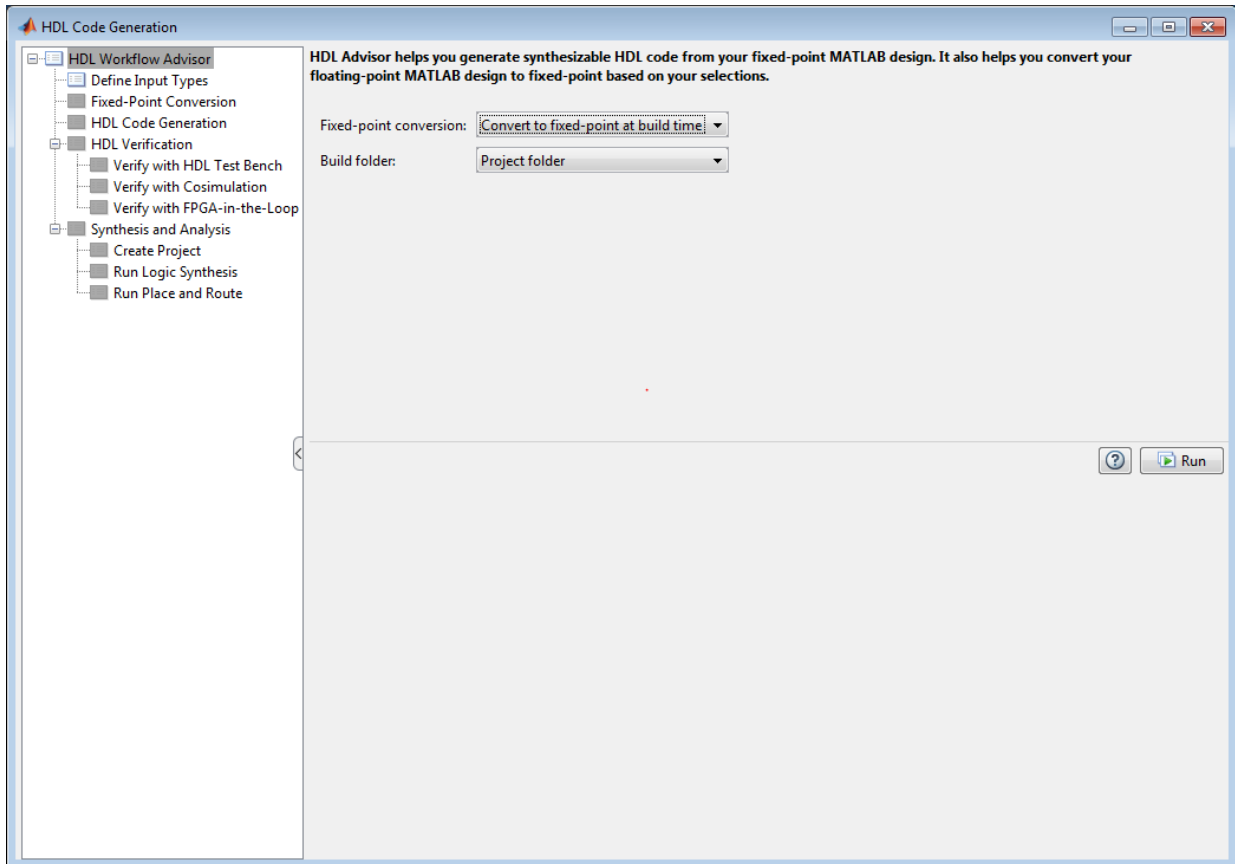
Fixed-Point Code Generation Workflow

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.

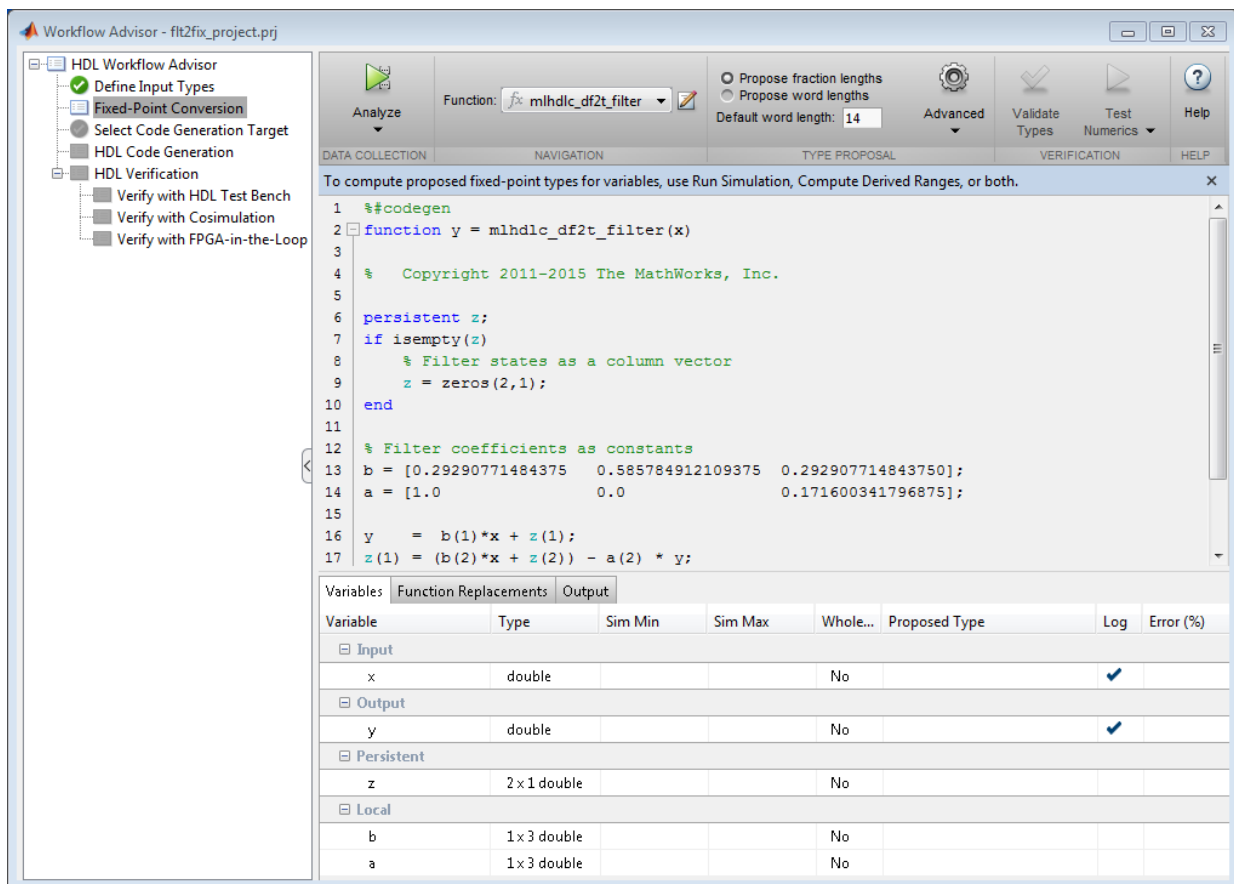
After simulation notice that the input variable 'x' is defined as scalar double 'double(1x1)'

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.



The screenshot shows the HDL Workflow Advisor interface for a project named 'flt2fx_project.prj'. The 'Fixed-Point Conversion' step is selected in the left-hand navigation pane. The main workspace displays the source code for the function 'mldlc_df2t_filter'. The code includes comments and defines variables 'b' and 'a' as constants, and 'z' as a persistent variable. Below the code, a table summarizes the inferred types and simulation ranges for the variables.

Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
Input							
x	double			No		✓	
Output							
y	double			No		✓	
Persistent							
z	2 x 1 double			No			
Local							
b	1 x 3 double			No			
a	1 x 3 double			No			

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.

The screenshot shows the HDL Workflow Advisor interface for a project named 'flt2fx_project.prj'. The 'Fixed-Point Conversion' step is selected in the left-hand navigation pane. The main window displays the MATLAB code for a function named 'mldlc_df2t_filter'. The code includes comments for code generation, copyright information, and filter coefficients. Below the code, a table provides simulation data for various variables.

Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
Input							
x	double	-1	1	No	numerictype(1, 14, 12)	✓	
Output							
y	double	-0.99	1	No	numerictype(1, 14, 13)	✓	
Persistent							
z	2 x 1 double	-0.8	0.8	No	numerictype(1, 14, 13)		
Local							
b	1 x 3 double	0.29	0.59	No	numerictype(0, 14, 14)		
a	1 x 3 double	0	1	No	numerictype(0, 14, 13)		

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's menu to enable logging of histogram data.

The screenshot shows the HDL Workflow Advisor interface for a project named 'flt2fix_project.prj'. The 'Fixed-Point Conversion' step is active. The main window displays the MATLAB code for the `mlhdlc_df2t_filter` function. Below the code is a table of variables with their types and simulation ranges. A histogram window is overlaid on the table, showing the distribution of values for variable `y` and the proposed numeric type.

Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
Input							
x	double	-1	1	No	numerictype(1, 14, 12)	<input checked="" type="checkbox"/>	
Output							
y	double	-0.99				<input checked="" type="checkbox"/>	
Persistent							
z	2 x 1 double	-0.8					
Local							
b	1 x 3 double	0.29					
a	1 x 3 double	0					

The histogram window shows the distribution of values for variable `y`. The x-axis represents bit weights and the y-axis represents the number of occurrences. The proposed numeric type is `numerictype(1, 14, 12)`. The histogram shows that the simulation values covered 97% of the supported range of -2 to 1.9998.

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 screenshot shows the MATLAB Workflow Advisor interface for a project named 'flt2fix_project.prj'. The left-hand pane shows the workflow steps: 'Define Input Types', 'Fixed-Point Conversion' (highlighted), 'Select Code Generation Target', 'HDL Code Generation', and 'HDL Verification'. The central pane displays the MATLAB code for the function 'mlhdlc_df2t_filter'. The code includes comments for code generation, persistent variables, and filter coefficients. The bottom pane shows the output of the 'Validate Types' step, including simulation results and a 'Type Validation Output' section with hyperlinks to generated reports and code.

```

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);

```

Variables | Function Replacements | Output

```

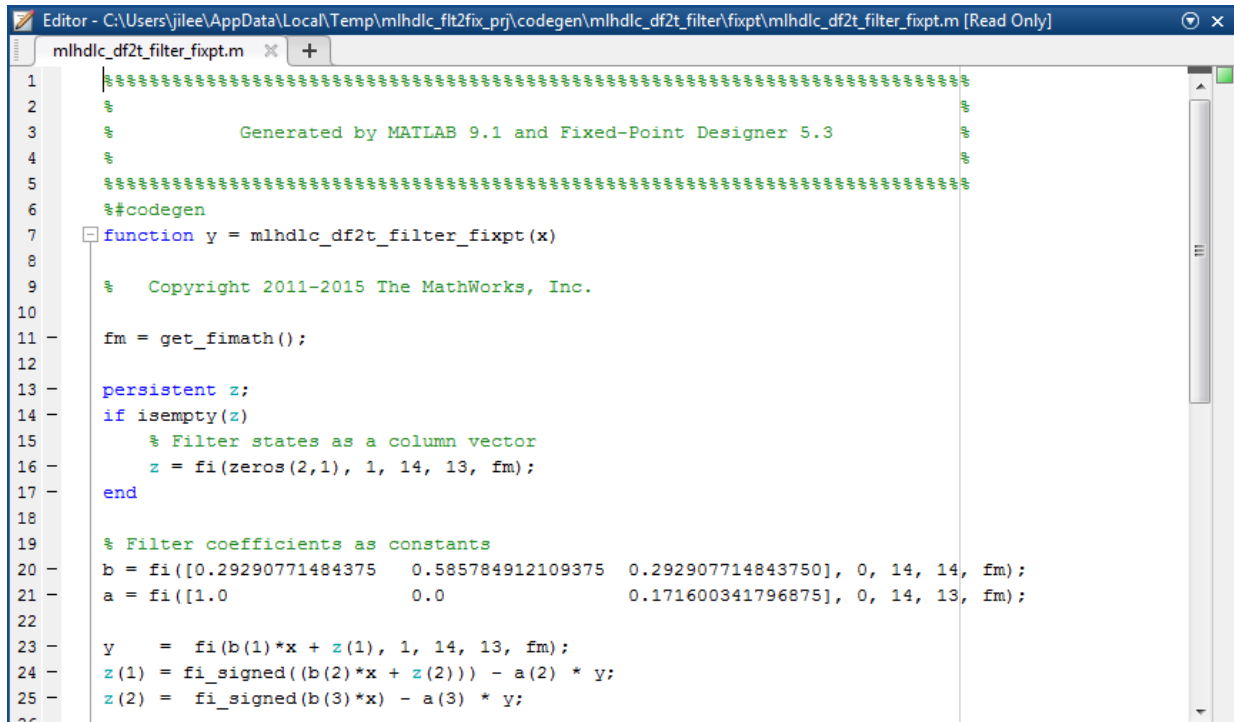
### Analyzing the test bench(es) 'mlhdlc_df2t_filter_tb'
### Begin Floating Point Simulation (Instrumented)
### Floating Point Simulation Completed in 1.0194 sec(s)
### Elapsed Time: 1.5504 sec(s)

Type Validation Output (11/1/16 5:34 PM)

### Generating Type Proposal Report for 'mlhdlc_df2t_filter' mlhdlc\_df2t\_filter\_report.htm
### Generating Fixed Point MATLAB Code mlhdlc\_df2t\_filter\_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper mlhdlc\_df2t\_filter\_wrapper\_fixpt
### Generating Mex file for 'mlhdlc_df2t_filter_wrapper_fixpt'
Code generation successful: View report

```

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.



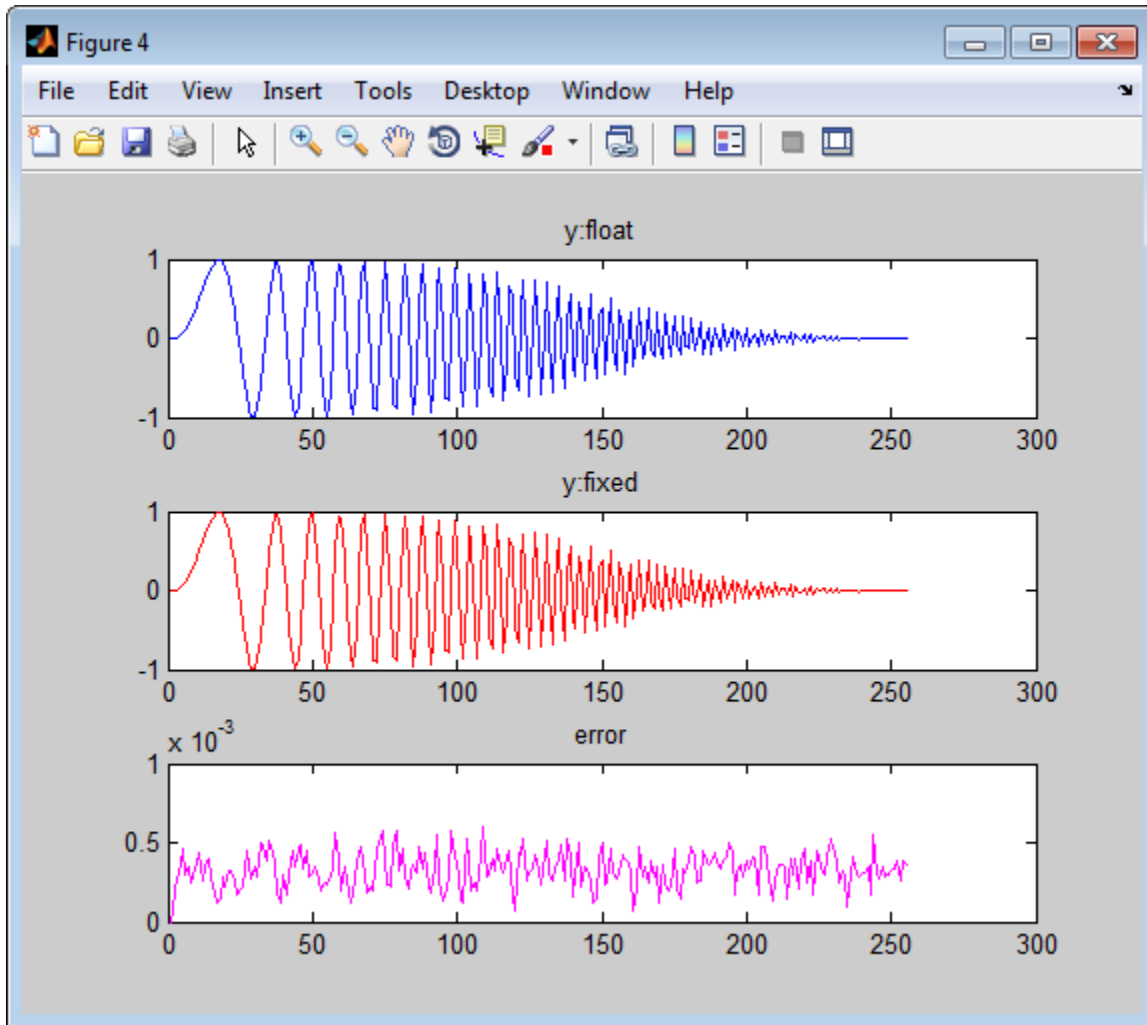
```
Editor - C:\Users\jilee\AppData\Local\Temp\mlhdlc_fit2fix_prj\codegen\mlhdlc_df2t_filter\fixpt\mlhdlc_df2t_filter_fixpt.m [Read Only]
mlhdlc_df2t_filter_fixpt.m x +
1 |%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 |%
3 |%           Generated by MATLAB 9.1 and Fixed-Point Designer 5.3
4 |%
5 |%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6 |%#codegen
7 |function y = mlhdlc_df2t_filter_fixpt(x)
8 |
9 |%   Copyright 2011-2015 The MathWorks, Inc.
10 |
11 |fm = get_fimath();
12 |
13 |persistent z;
14 |if isempty(z)
15 |    % Filter states as a column vector
16 |    z = fi(zeros(2,1), 1, 14, 13, fm);
17 |end
18 |
19 |% Filter coefficients as constants
20 |b = fi([0.29290771484375    0.585784912109375    0.292907714843750], 0, 14, 14, fm);
21 |a = fi([1.0                0.0                0.171600341796875], 0, 14, 13, fm);
22 |
23 |y    = fi(b(1)*x + z(1), 1, 14, 13, fm);
24 |z(1) = fi_signed((b(2)*x + z(2))) - a(2) * y;
25 |z(2) = fi_signed(b(3)*x) - a(3) * y;
26 |
```

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 all 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.

You can refer to the Fixed-Point Type Conversion and Refinement example for more details on how to iterate and refine the numerics of the algorithm in the generated fixed-point code.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab...
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fix_prj'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

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';
%
% MATLAB Design: <matlab:edit('mlhdlc_kalman_hdl') mlhdlc_kalman_hdl>
% MATLAB testbench: <matlab:edit('mlhdlc_kalman_hdl_tb') mlhdlc_kalman_hdl_tb>
%
```

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fix'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
```

```
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);  
  
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

Simulate the Design

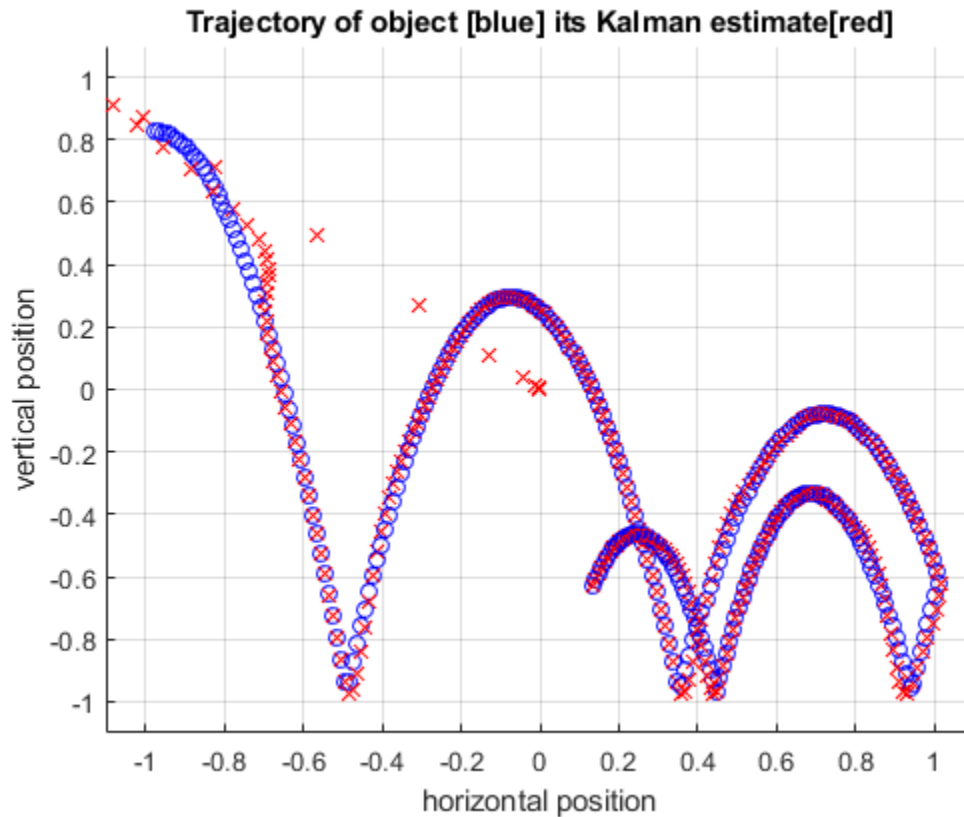
Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
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.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating 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 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 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.

The screenshot shows the HDL Workflow Advisor interface for a project named 'flt2fix_project.prj'. The 'Fixed-Point Conversion' step is selected in the left-hand navigation pane. The central pane displays the MATLAB code for a function named 'mlhdlc_kalman_hdl'. The code includes comments for initialization and measurement matrix setup. The bottom pane shows a table of proposed fixed-point types for variables.

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)		

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 Computing Derived Ranges in fixed-point conversion

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.

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.

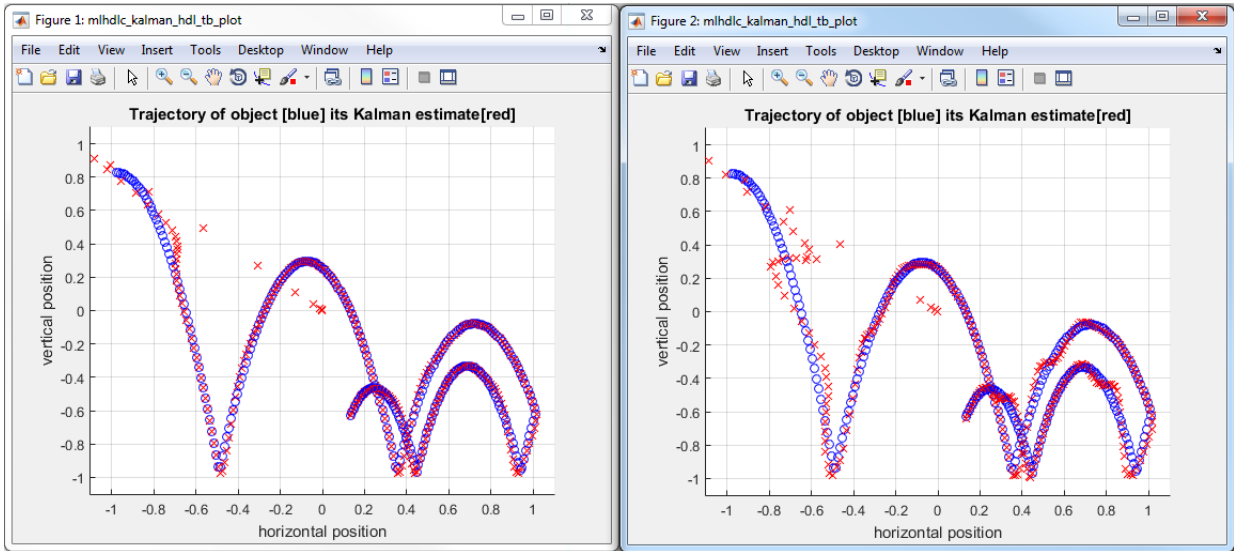
Generate Fixed-Point Code and Verify the Generated Code

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.

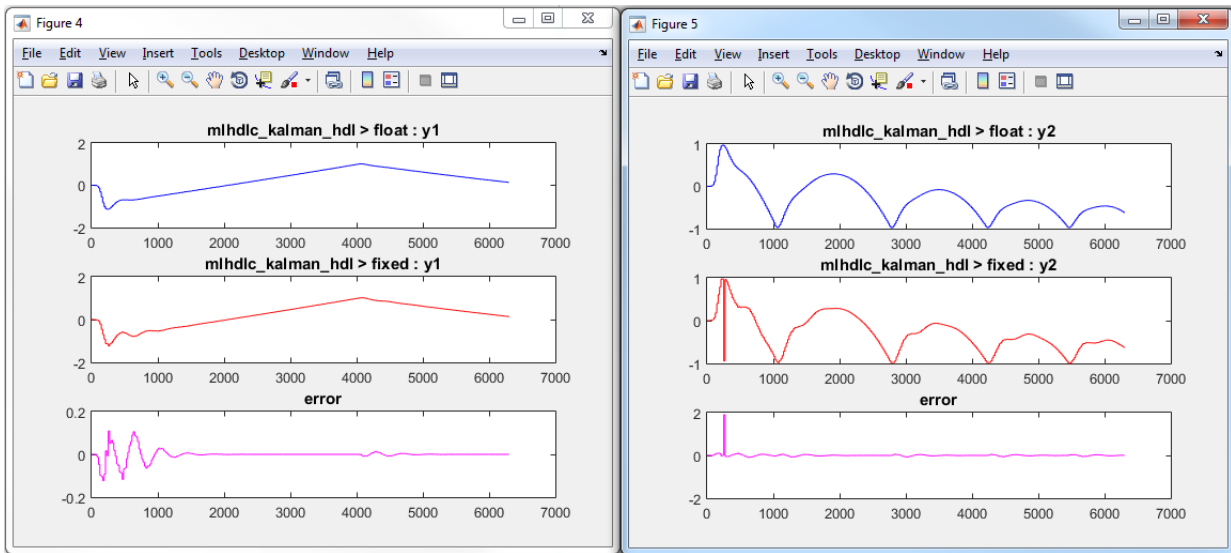
The plot on the left is generated from testbench during the simulation of floating-point code, the one on the right is generated from the simulation of the generated fixed-point code. Notice, the plots do not match.

4 Fixed-Point Conversion



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.

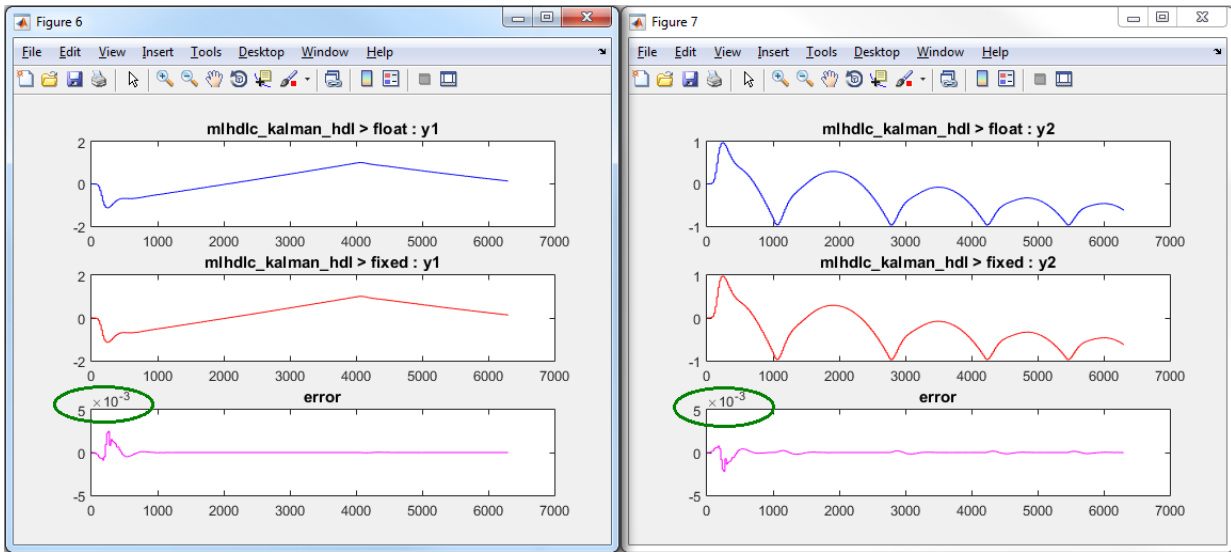
Let us now increase the default word length to 28 bits and repeat the type proposal and validation steps.

- 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.



Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fix'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

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

Create a New Folder and Copy Relevant Files

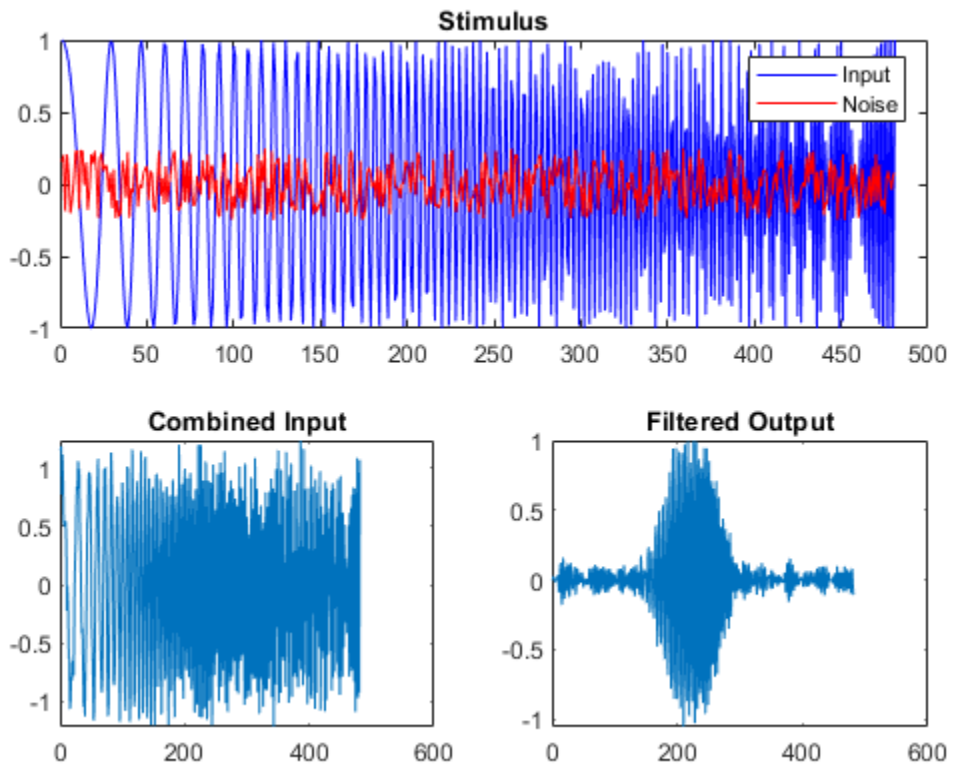
Executing the following lines of code copies the necessary example files into a temporary folder.

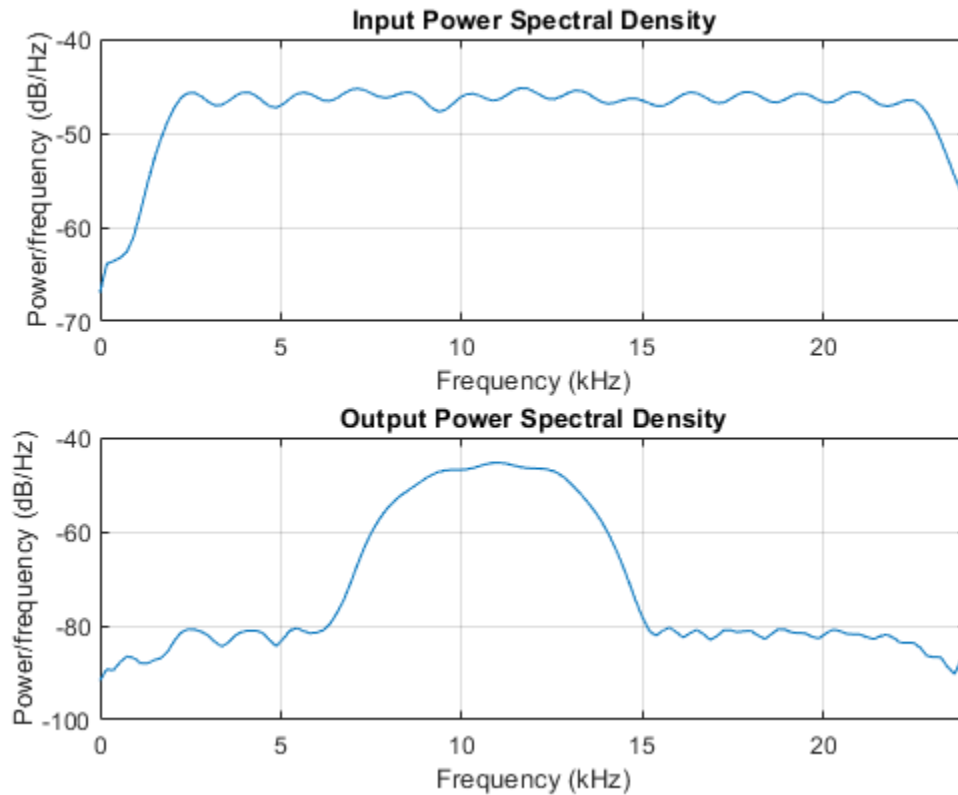
```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fix'];  
  
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);  
  
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

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.

You can refer to the Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Fixed-Point Code Generation Workflow

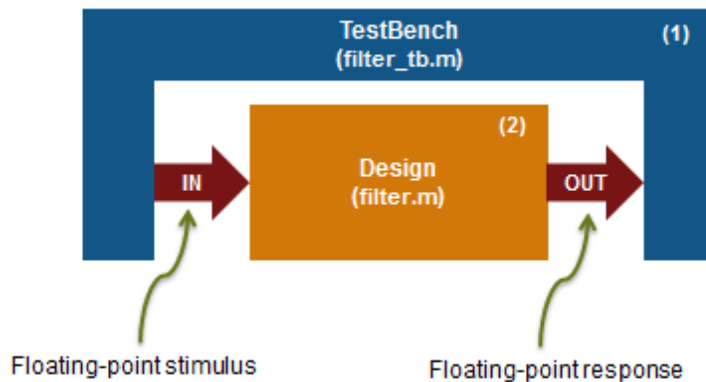
Perform the following tasks in preparation for the fixed-point code generation step:

- 1 Click the Advisor button to launch the Workflow Advisor.
- 2 Choose 'Yes' for the option 'Design needs conversion to fixed-point'.
- 3 Right-click the 'Propose Fixed-Point Types' step.
- 4 Choose 'Run to Selected Task' to execute the instrumented floating-point simulation.

Refer to the Floating-Point to Fixed-Point Conversion tutorial for a more complete description of 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:

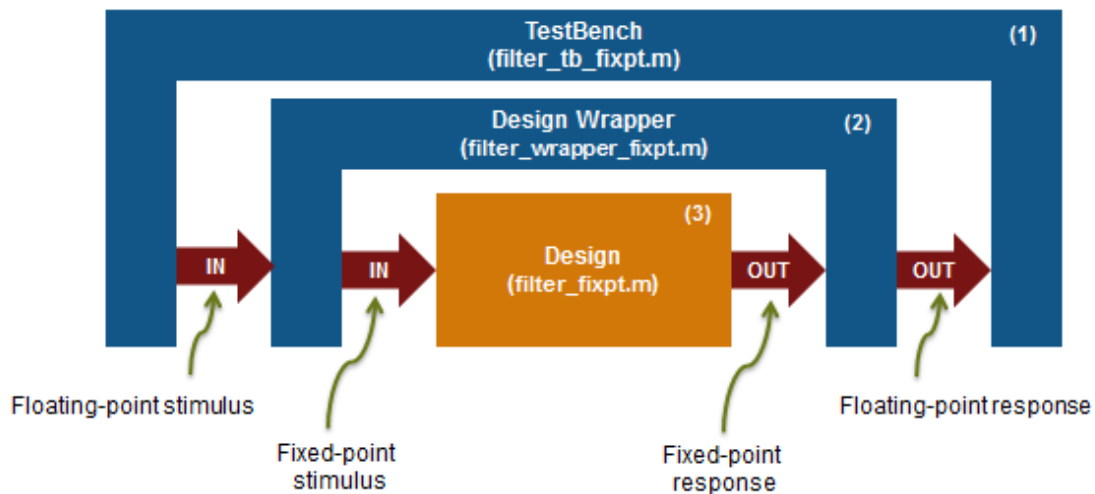
- The testbench 'mlhdlc_filter_tb.m' (1) must be a script or a function with no inputs.
- The design 'mlhdlc_filter.m' (2) must be a function.
- There must be at least one call to the design from the testbench. All call sites contribute when determining the proposed fixed-point types.

- 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

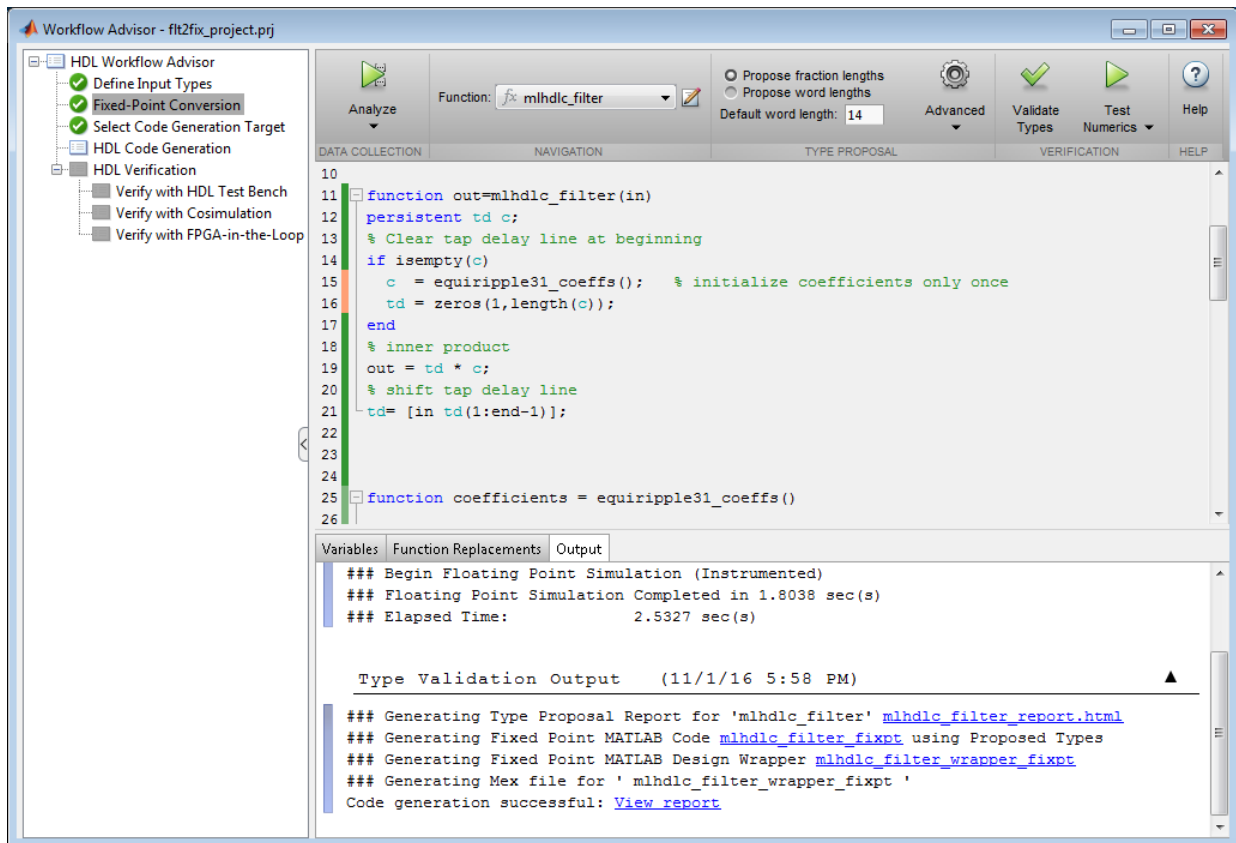
During the type validation step, fixed-point code is generated for this design and compiled to verify that there are no errors when applying the types. The output files will have the following structure.



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/filter/fixpt'.



Click the links to the generated code in the Workflow Advisor log window to examine the generated fixed-point design, wrapper, and test bench.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab...  
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fix'];  
clear mex;  
cd (mlhdlc_demo_dir);  
rmdir(mlhdlc_temp_dir, 's');
```

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	ignore simulation ranges	Propose data types based on derived ranges.
Note Manually-entered static ranges always take precedence over simulation 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.

Advanced Type Proposal Settings	Values	Description
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.
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.

Advanced Type Proposal Settings	Values	Description
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 <code>fimath</code> properties for the generated fixed-point data types.
	Convergent	
	Floor (default)	
	Nearest	The default fixed-point math properties use the <code>Floor</code> rounding and <code>Wrap</code> overflow. These settings generate the most efficient code but might cause problems with overflow.
	Round	
	Zero	
Overflow action	Saturate	
	Wrap (default)	
Product mode	FullPrecision (default)	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.
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	For more information on <code>fimath</code> properties, see “ <code>fimath</code> Object Properties” (Fixed-Point Designer).
	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	

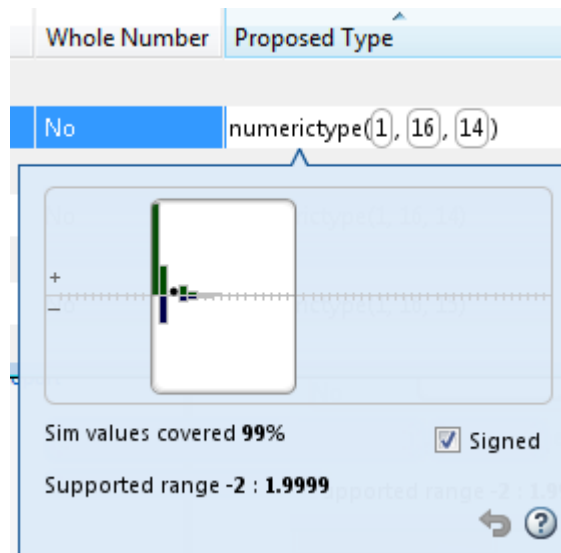
Log Data for Histogram

To log data for histograms:

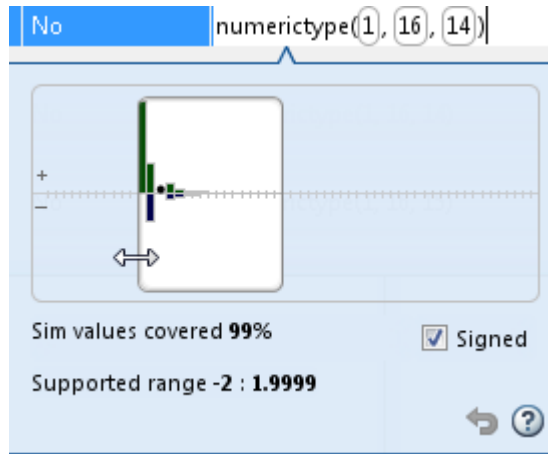
- 1 In the Fixed-Point Conversion window, click **Run Simulation** and select **Log data for histogram**, and then click the Run Simulation button.

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 4-47.

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 4-40.

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 4-40.

Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.

- **Whole Number**

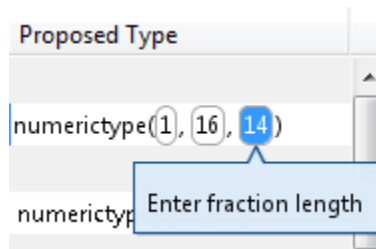
The app uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

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 4-53.

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 4-41
“Automated Fixed-Point Conversion Capabilities” on page 4-41
“Code Coverage” on page 4-42
“Proposing Data Types” on page 4-45
“Locking Proposed Data Types” on page 4-47
“Viewing Functions” on page 4-47
“Viewing Variables” on page 4-47
“Histogram” on page 4-53
“Function Replacements” on page 4-55
“Validating Types” on page 4-55
“Testing Numerics” on page 4-56
“Detecting Overflows” on page 4-56

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 4-47.

For a list of supported MATLAB features and functions, see “MATLAB Language Features Supported for Automated Fixed-Point Conversion” (MATLAB Coder).

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 = [
9          1, -0.942809041582063, 0.333333333333333];
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	<p>One of the following situations:</p> <ul style="list-style-type: none"> 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. <p>Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.</p>
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, 0.333333333333333];
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.
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 4-46.

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

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. 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 4-47.

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 4-55.

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 4-53.

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 4-47.

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:

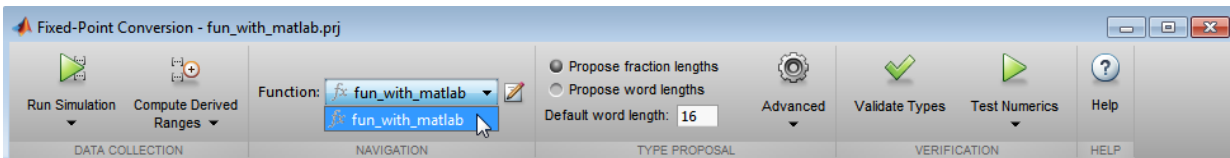
- Manually setting a proposed data type in the Fixed-Point Conversion tool.
- Right-clicking a type proposed by the tool and selecting **Lock computed value**.

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.

The screenshot shows the Fixed-Point Conversion tool interface for a project named 'counter.prj'. The top navigation bar includes buttons for 'Run Simulation', 'Compute Derived Ranges', and a 'Function' dropdown menu. The dropdown menu is open, showing options: 'fx next', 'fx Counter', 'fx next', and 'fx use_counter'. Below the navigation bar, the tool is divided into sections: DATA COLLECTION, NAVIGATION, TYPE PROPOSAL, VERIFICATION, and HELP. The main code window displays the MATLAB class definition for 'Counter'.

```

1 classdef Counter < handle
2     properties
3         Value;
4     end
5
6     methods(Static)
7         function t = MAX_VALUE()
8             t = 128;
9         end
10    end
11
12    methods
13        function this = Counter()
14            this.Value = 0;
15        end
16        function out = next(this)
17            out = this.Value;
18            if this.Value == this.MAX_VALUE
19                this.Value = 0;
20            else
21                this.Value = this.Value + 1;
22            end
23        end
24    end
25 end
  
```

- 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
```

The screenshot shows the 'Fixed-Point Conversion - dut.prj' application window. The main editor displays the following MATLAB code:

```

1 function y = dut(u, v)
2
3     tt1 = foo(u);
4     tt2 = foo([u v]);
5     tt3 = foo(complex(u,v));
6
7     ss1 = bar(u);
8     ss2 = bar([u v]);
9     ss3 = bar(complex(u,v));
10
11     y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);
12
13 end
14
15 function y = foo(u)
16     y = u * 2;
17 end
18
19 function y = bar(u)
20     y = u * 4;
21 end

```

The 'Function:' dropdown menu is open, showing the following options:

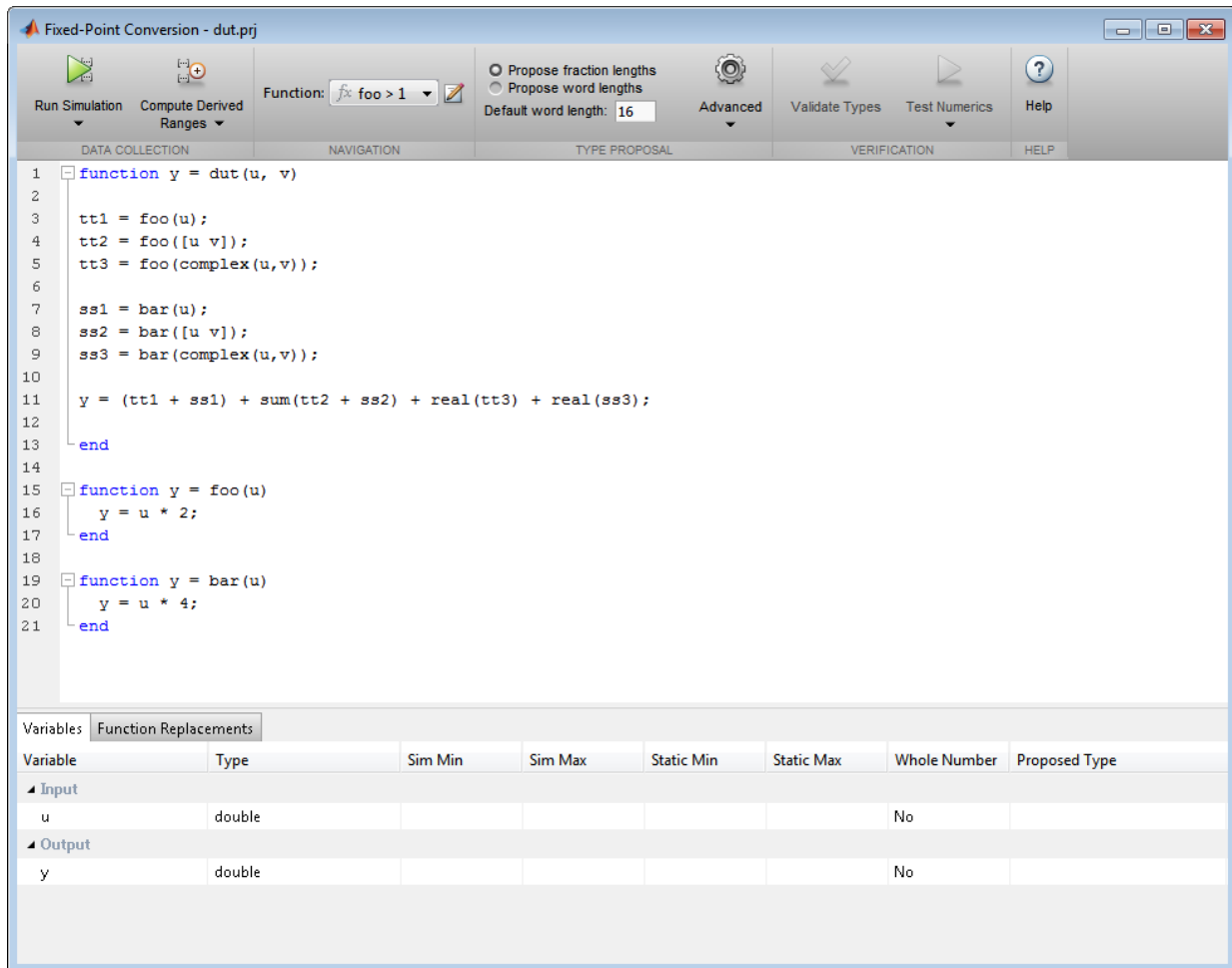
- fix dut
- fix foo > 1
- fix foo > 2
- fix foo > 3
- fix bar > 1
- fix bar > 2
- fix bar > 3

The interface also includes a toolbar with icons for 'Run Simulation', 'Compute Derived Ranges', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. A 'Function Replacements' table is visible at the bottom.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
▲ Input							
u	double					No	
v	double					No	
▲ Output							
y	double					No	
▲ Local							
ss1	double					No	

If you select a specialization, the app displays only the variables used by the specialization.

4 Fixed-Point Conversion



The screenshot shows the 'Fixed-Point Conversion - dut.prj' window. The top toolbar includes 'Run Simulation', 'Compute Derived Ranges', 'Function: foo > 1', 'Propose fraction lengths', 'Propose word lengths', 'Default word length: 16', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. Below the toolbar are tabs for 'DATA COLLECTION', 'NAVIGATION', 'TYPE PROPOSAL', 'VERIFICATION', and 'HELP'. The main area displays source code for three functions: 'dut', 'foo', and 'bar'. The 'Function Replacements' table is visible at the bottom.

```
1 function y = dut(u, v)
2
3     tt1 = foo(u);
4     tt2 = foo([u v]);
5     tt3 = foo(complex(u,v));
6
7     ss1 = bar(u);
8     ss2 = bar([u v]);
9     ss3 = bar(complex(u,v));
10
11     y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);
12
13 end
14
15 function y = foo(u)
16     y = u * 2;
17 end
18
19 function y = bar(u)
20     y = u * 4;
21 end
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
u	double					No	
Output							
y	double					No	

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`.


```

Editor - C:\Work\specializations\codegen\dut_fixpt\dut_fixpt.m [Read Only]
EDITOR PUBLISH VIEW
New Open Save Compare Find Files Go To Comment Insert % Indent Breakpoints Run Run and Advance Run Section Run and Time
FILE NAVIGATE EDIT BREAKPOINTS RUN
dut.m dut_fixpt.m
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 %       Generated by MATLAB 8.4 and Fixed-Point Designer 4.3
4 %
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6 %%codegen
7 function y = dut_fixpt(u, v)
8
9     fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full
10
11     tt1 = fi(foo_s1(u), 0, 5, 0, fm);
12     tt2 = fi(foo_s2([fi(u, 0, 5, 0, fm) v]), 0, 6, 0, fm);
13     tt3 = fi(foo_s3(complex(u,v)), 0, 6, 0, fm);
14
15     ss1 = fi(bar_s1(u), 0, 6, 0, fm);
16     ss2 = fi(bar_s2([fi(u, 0, 5, 0, fm) v]), 0, 7, 0, fm);
17     ss3 = fi(bar_s3(complex(u,v)), 0, 7, 0, fm);
18
19     y = fi((tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3), 0, 9, 0, fm);
20
21 end
22
23 function y = foo_s1(u)
24     fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Fu
25
26     y = fi(u * fi(2, 0, 2, 0, fm), 0, 5, 0, fm);
27 end
28
29 function y = foo_s2(u)
30     fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Fu
31
32     y = fi(u * fi(2, 0, 2, 0, fm), 0, 6, 0, fm);
33 end
34
Ln 1 Col 1

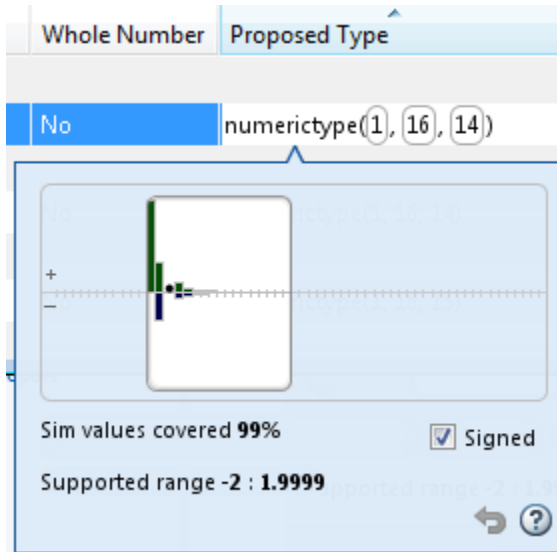
```

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.

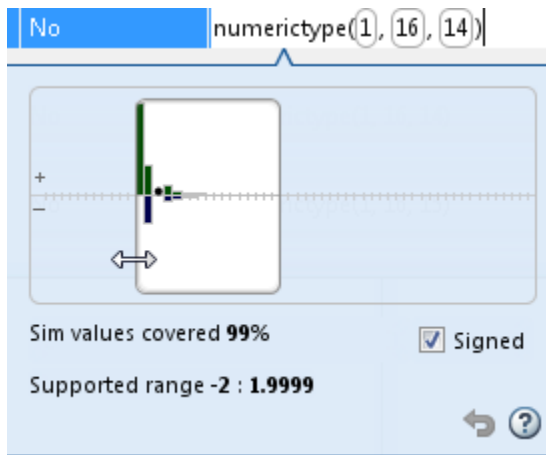
After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

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 `numeric_type(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.

Variables Function Replacements Simulation Output ▾					
Enter a function to replace				Custom Function ▾	+ -
Function or Operator	Replacement				
▾ Custom Function	<i>Function Name</i>				
foo	foo_fixedpoint				
▾ Lookup Table	<i>Interpolation Method</i>	<i>Design Min</i>	<i>Design Max</i>	<i>Number of Points</i>	
exp	None	Auto	Auto	1000	

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” (MATLAB Coder)

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 4-60.

Visualize Differences Between Floating-Point and Fixed-Point Results

This example shows how to configure the `codegen` function to use a custom plot function to compare the behavior of the generated fixed-point code against the behavior of the original floating-point MATLAB code.

By default, when the `LogIOForComparisonPlotting` option is enabled, the conversion process uses a time series based plotting function 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. This example shows how to customize plotting and produce scatter plots at the test numerics step of the fixed-point conversion.

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” (MATLAB).

Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\custom_plot`.
- 2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```
- 3 Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

Type	Name	Description
Function code	<code>myFilter.m</code>	Entry-point MATLAB function

Type	Name	Description
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-
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % escape the '_'s because plot titles treat these as subscripts
    escapedVarName = regexp(varName, '_', '\\_');
    escapedFcnName = regexp(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

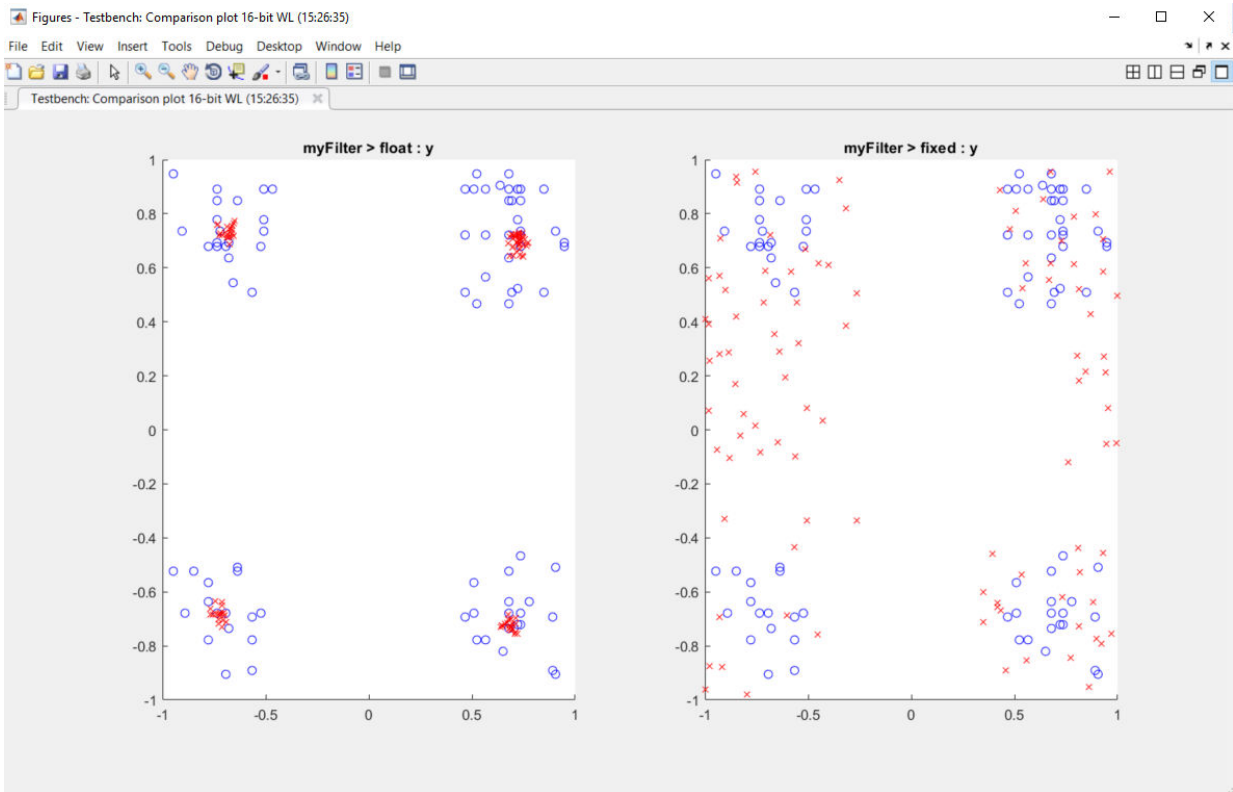
Convert the floating-point MATLAB function, `myFilter`, to fixed-point MATLAB code. You do not need to specify input types for the `codegen` command because it infers the types from the test file.

```
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.

4 Fixed-Point Conversion

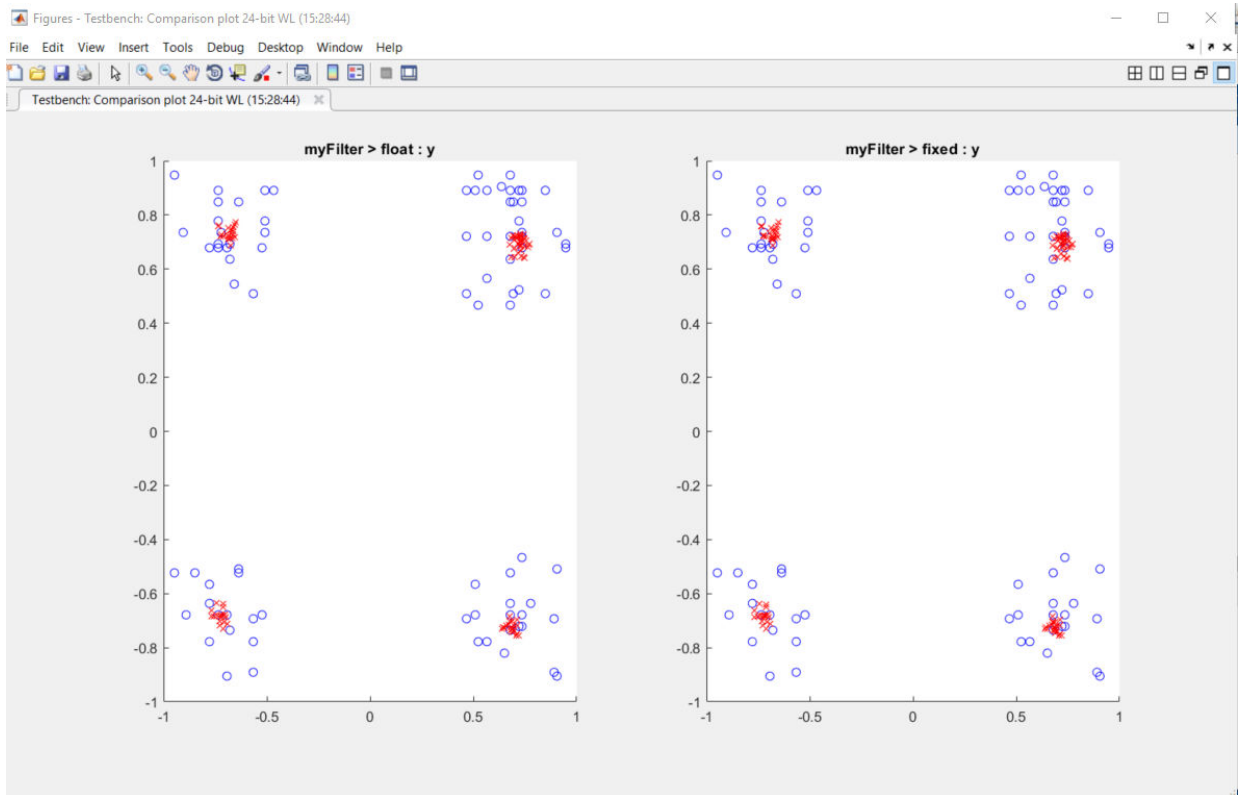


The plot shows that the fixed-point results do not closely match the floating-point results.

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.



Inspecting Data Using the Simulation Data Inspector

In this section...

“What Is the Simulation Data Inspector?” on page 4-66

“Import Logged Data” on page 4-66

“Export Logged Data” on page 4-66

“Group Signals” on page 4-67

“Run Options” on page 4-67

“Create Report” on page 4-67

“Comparison Options” on page 4-67

“Enabling Plotting Using the Simulation Data Inspector” on page 4-67

“Save and Load Simulation Data Inspector Sessions” on page 4-68

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 4-69.

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 4-69

“From the Command Line” on page 4-69

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” (MATLAB Coder).

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” (MATLAB Coder).

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 4-80
- “Replace a Custom Function with a Lookup Table” on page 4-72

Replace a Custom Function with a Lookup Table

In this section...

“Using the HDL Coder App” on page 4-72

“From the Command Line” on page 4-77

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 `fiaccl` 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**.

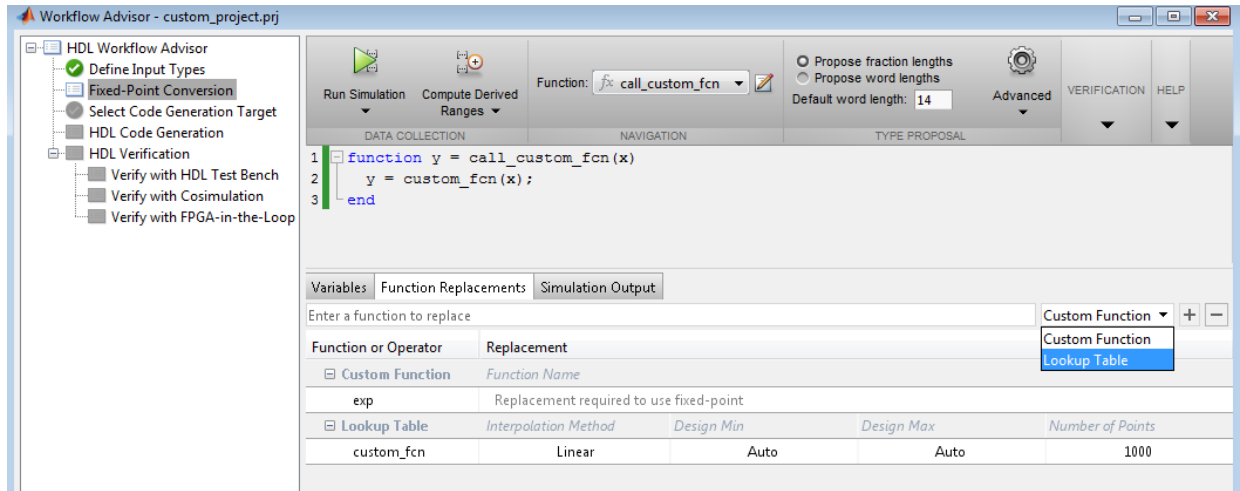
From the test file, HDL Coder determines that `x` is a scalar double.
- 3 Click **Use These Types**.

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 **+**.

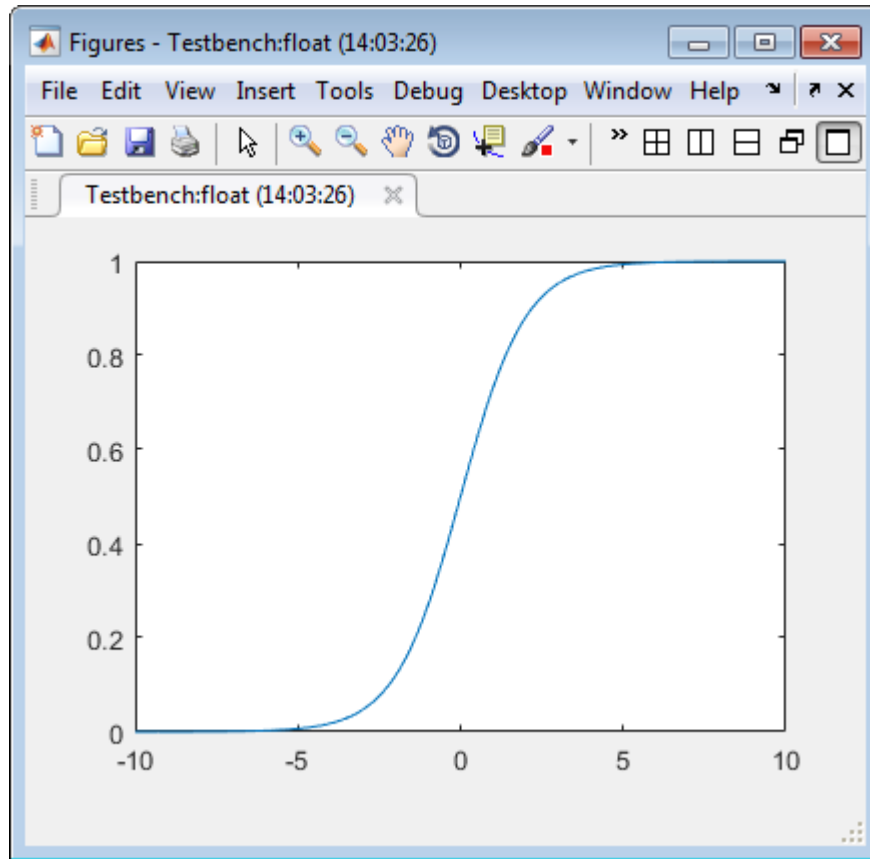
By default, the lookup table uses linear interpolation, 1000 points, and design minimum and maximum values that the app detects by running a simulation or computing derived ranges.

4 Fixed-Point Conversion



- 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.



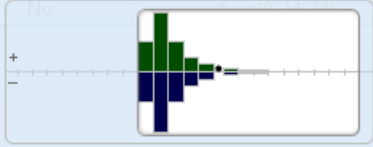
Validate Fixed-Point Types

- 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.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	double	-10	10			No	numerictype(1, 14, 9)
Output							
y	double	0	1				

No



Sim values covered **100%** Signed

Supported range **-16 : 15.998**

- 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`.

- 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” (MATLAB).

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 4-80

More About

- “Replacing Functions Using Lookup Table Approximations” on page 4-71

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.

In this section...
“From the UI” on page 4-80
“From the Command Line” on page 4-85

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.

4 Fixed-Point Conversion

The screenshot shows the HDL Workflow Advisor interface for a project named 'custom_project.prj'. The left-hand pane shows the workflow steps: 'HDL Workflow Advisor' (expanded), 'Define Input Types' (checked), 'Fixed-Point Conversion' (selected), 'Select Code Generation Target', 'HDL Code Generation', and 'HDL Verification' (expanded). Under 'HDL Verification', there are options for 'Verify with HDL Test Bench', 'Verify with Cosimulation', and 'Verify with FPGA-in-the-Loop'. The main workspace shows a code editor with the following code:

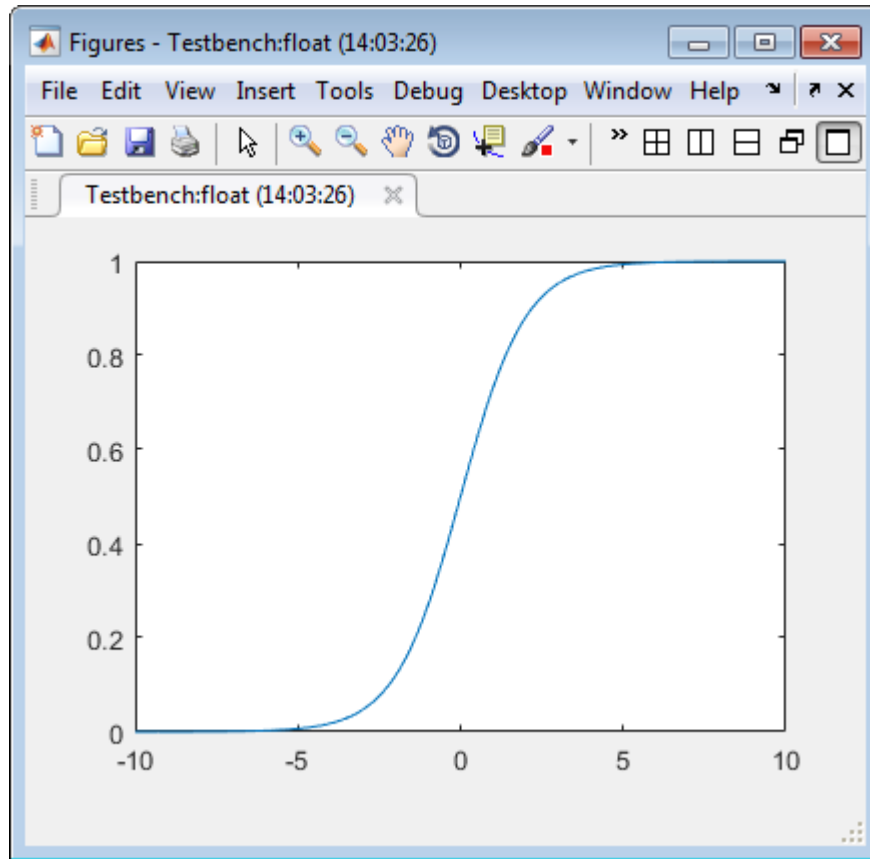
```
1 function y = call_custom_fcn(x)
2   y = custom_fcn(x);
3 end
```

Below the code editor, there are tabs for 'Variables', 'Function Replacements', and 'Simulation Output'. The 'Function Replacements' tab is active, showing a search field 'Enter a function to replace' and a dropdown menu with 'Custom Function' selected. Below this is a table with the following data:

Function or Operator	Replacement	Interpolation Method	Design Min	Design Max	Number of Points
exp	Replacement required to use fixed-point				
custom_fcn	Linear	Auto	Auto	Auto	1000

- 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.



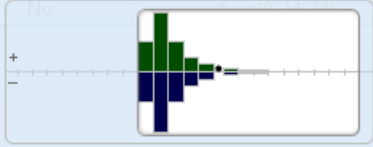
Validate Fixed-Point Types

- 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.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	double	-10	10			No	numerictype(1, 14, 9)
Output							
y	double	0	1				

No



Sim values covered **100%** Signed

Supported range **-16 : 15.998**

- 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`.

- 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” (MATLAB).

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 4-72

More About

- “Replacing Functions Using Lookup Table Approximations” on page 4-71

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” (Fixed-Point Designer).

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.

Code Generation

- “Create and Set Up Your Project” on page 5-2
- “Specify Properties of Entry-Point Function Inputs” on page 5-6
- “Basic HDL Code Generation with the Workflow Advisor” on page 5-10
- “HDL Code Generation from System Objects” on page 5-15
- “Code Generation Reports” on page 5-20
- “Generate Instantiable Code for Functions” on page 5-26
- “Integrate Custom HDL Code Into MATLAB Design” on page 5-28
- “Enable MATLAB Function Block Generation” on page 5-34
- “System Design with HDL Code Generation from MATLAB and Simulink” on page 5-36
- “Generate HDL Code from MATLAB Code Using the Command Line Interface” on page 5-40
- “Specify the Clock Enable Rate” on page 5-45
- “Specify Test Bench Clock Enable Toggle Rate” on page 5-47
- “Generate an HDL Coding Standard Report from MATLAB” on page 5-49
- “Generate an HDL Lint Tool Script” on page 5-53
- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-56
- “Minimize Clock Enables” on page 5-59

Create and Set Up Your Project

In this section...
“Create a New Project” on page 5-2
“Open an Existing Project” on page 5-4
“Add Files to the Project” on page 5-4

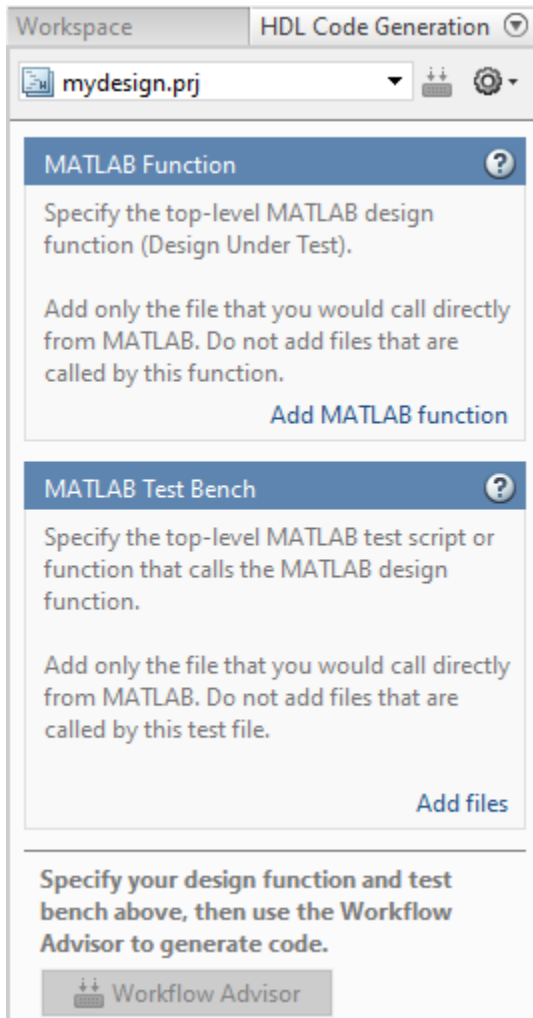
Create a New Project

- 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 5-6.

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 5-6.

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 5-6

“Why You Must Specify Input Properties” on page 5-6

“Properties to Specify” on page 5-6

“Rules for Specifying Properties of Primary Inputs” on page 5-8

“Methods for Defining Properties of Primary Inputs” on page 5-8

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.

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.

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

For	Specify properties				
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
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

Class Name	Description
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 numeric type and fimath properties.

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
<p>“Define Input Properties by Example at the Command Line”</p> <p>Note If you define input properties programmatically in the MATLAB file, you cannot use this method</p>	<ul style="list-style-type: none"> • Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a few primary inputs 	<ul style="list-style-type: none"> • 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

Method	Advantages	Disadvantages
"Define Input Properties Programmatically in the MATLAB File" (MATLAB Coder)	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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.

See Also

Basic HDL Code Generation with the Workflow Advisor

This example shows how to work with MATLAB® HDL Coder™ projects to generate HDL from MATLAB designs.

Introduction

This example helps you familiarize yourself with the following aspects of HDL code generation:

- 1 Generating HDL code from MATLAB design.
- 2 Generating a HDL test bench from a MATLAB test bench.
- 3 Verifying the generated HDL code using a HDL simulator.
- 4 Synthesizing the generated HDL code using a HDL synthesis tool.

MATLAB Design

The MATLAB code used in this example implements a simple symmetric FIR filter. This example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_sfir';  
testbench_name = 'mlhdlc_sfir_tb';
```

- 1 MATLAB Design: mlhdlc_sfir
- 2 MATLAB testbench: mlhdlc_sfir_tb

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir'];
```

```
% Create a temporary folder and copy the MATLAB files.
```

```
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);
```

```
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```


Simulate the Design

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

```
mlhdlc_sfir_tb
```

Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sfir
```

Next, add the file 'mlhdlc_sfir.m' to the project as the MATLAB Function and 'mlhdlc_sfir_tb.m' as the MATLAB Test Bench.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete introduction to creating and populating HDL Coder projects.

Step 1: Generate Fixed-Point MATLAB Code

Right-click the 'Float-to-Fixed Workflow' step and choose the option 'Run this task' to run all the steps to generate fixed-point MATLAB code.

Examine the generated fixed-point MATLAB code by clicking the links in the log window to open the MATLAB code in the editor.

For more details on fixed-point conversion, refer to the Floating-Point to Fixed-Point Conversion tutorial.

The screenshot shows the HDL Workflow Advisor interface. The left pane shows the workflow steps: Define Input Types, Fixed-Point Conversion (selected), Select Code Generation Target, HDL Code Generation, HDL Verification, Verify with HDL Test Bench, Verify with Cosimulation, and Verify with FPGA-in-the-Loop. The main area displays the MATLAB code for the 'mldlc_sfir' function, which is a symmetric FIR filter. The code includes comments and logic for delay registers and multiplier chains.

Below the code, there is a table showing the variables and their properties:

Variable	Type	Sim Min	Sim Max	Who...	Proposed Type	Log	Error (%)
Input							
x_in	double	-1	1	No	numerictype(1, 14, 12)	✓	
h_in1	double	-0.13	-0.13	No	numerictype(1, 14, 15)	✓	
h_in2	double	-0.08	-0.08	No	numerictype(1, 14, 16)	✓	
h_in3	double	0.2	0.2	No	numerictype(0, 14, 16)	✓	
h_in4	double	0.41	0.41	No	numerictype(0, 14, 15)	✓	
Output							
y_out	double	-1.22	1.22	No	numerictype(1, 14, 12)	✓	
delayed_xout	double	-1	1	No	numerictype(1, 14, 12)	✓	

Step 2: Generate HDL Code

This step generates Verilog code from the generated fixed-point MATLAB design, and a Verilog test bench from the MATLAB test bench wrapper.

To set code generation options and generate HDL code:

- 1 Click the 'Code Generation' step to view the HDL code generation options panel.
- 2 In the Target tab, choose 'Verilog' as the 'Language' option.
- 3 Select the 'Generate HDL' and 'Generate HDL test bench' options.

- 4 In the 'Optimizations' tab, choose '1' as the Input and Output pipeline length, and enable the 'Distribute pipeline registers' option.
- 5 In the 'Coding style' tab, choose 'Include MATLAB source code as comments' and 'Generate report' to generate a code generation report with comments and traceability links.
- 6 Click the 'Run' button to generate both the Verilog design and testbench with reports.

Examine the log window and click the links to explore the generated code and the reports.

Step 3: Simulate the Generated Code

In the 'HDL Verification' step, select 'Verify with HDL Test Bench' substep and choose the 'Multi-file test bench' option in 'Test Bench Options' sub-tab. This option helps to generate HDL test bench code and test bench data (stimulus and response) in separate files.

HDL Coder automates the process of generating a HDL test bench and running the generated HDL test bench using the ModelSim® or ISIM™ simulator, and reports if the generated HDL simulation matches the numerics and latency with respect to the fixed-point MATLAB simulation.

Step 4: Synthesize the Generated Code

HDL Coder also creates a Xilinx® ISE™ or Altera® Quartus™ project with the selected options and runs the selected logic synthesis and place-and-route steps for the generated HDL code.

Examine the log window to view the results of synthesis steps.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir'];
clear mex;
```

```
cd (mlhdlc_demo_dir);  
rmdir(mlhdlc_temp_dir, 's');
```

See Also

More About

- “HDL Workflow Advisor” on page 9-2
- “Automatic Verification of Generated HDL Code from MATLAB” (HDL Verifier)
- “FIL Simulation with HDL Workflow Advisor for MATLAB” (HDL Verifier)

HDL Code Generation from System Objects

This example shows how to generate HDL code from MATLAB® code that contains System objects.

MATLAB Design

The MATLAB code used in this example implements a simple symmetric FIR filter and uses the `dsp.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';
```

Let us take a look at the MATLAB design.

```
type(design_name);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
%
% Design pattern covered in this example:
% Filter states modeled using DSP System object (dsp.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 = dsp.Delay;
    h2 = dsp.Delay;
    h3 = dsp.Delay;
    h4 = dsp.Delay;
    h5 = dsp.Delay;
    h6 = dsp.Delay;
    h7 = dsp.Delay;
    h8 = dsp.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');

```

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```

mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sysobj_intro'];

```

```

% Create a temporary folder and copy the MATLAB files.

```

```

cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

```

```

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);

```

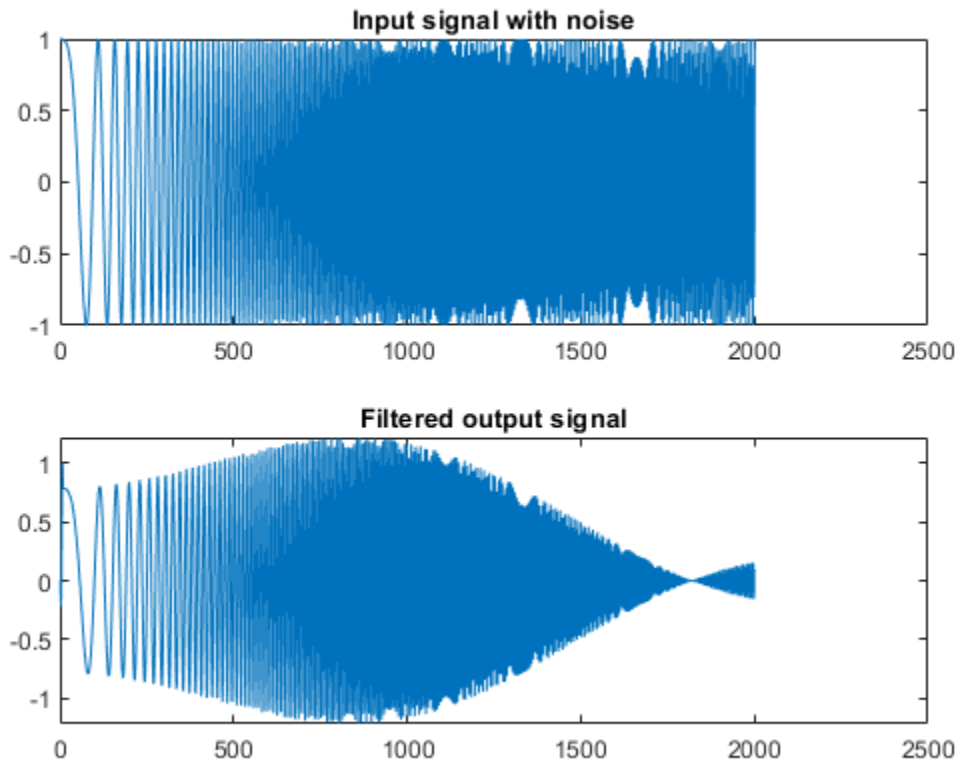
Simulate the Design

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.

You can refer to the Getting Started with MATLAB to HDL Workflow tutorial 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. 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

Refer to the documentation for a list of System objects supported for HDL code generation.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab...  
mlhdlc_temp_dir = [tempdir 'mlhdlc_sysobj_intro'];  
clear mex;  
cd (mlhdlc_demo_dir);  
rmdir(mlhdlc_temp_dir, 's');
```

Code Generation Reports

In this section...
“Report Generation” on page 5-20
“Report Location” on page 5-21
“Errors and Warnings” on page 5-21
“Files and Functions” on page 5-21
“MATLAB Source” on page 5-22
“MATLAB Variables” on page 5-23
“Additional Reports” on page 5-24
“Report Limitations” on page 5-24

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.

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 function in the **MATLAB Source** pane. To see information about the type of a variable or expression, pause over the variable or expression.

In the code pane, syntax highlighting of MATLAB source code helps you to identify MATLAB syntax elements. Syntax highlighting also helps you to identify certain code generation attributes such as whether a function is extrinsic or whether an argument is constant.

Extrinsic Functions

In the MATLAB code, the report identifies an extrinsic function with purple text. The information window indicates that the function is extrinsic.

The screenshot shows a MATLAB code editor window titled "Function: callMyExtrinsic". The code is as follows:

```

1 function z = callMyExtrinsic(a,b)
2 %#codegen
3 coder.extrinsic('myExtrinsic');
4 z = 0;
5 z = myExtrinsic(a,b);
6 disp(z);
7 end
8

```

The call to `myExtrinsic(a,b)` on line 5 is highlighted with a blue dashed border. An information window titled "EXPRESSION INFO" is open over this call, displaying the following details:

- Expression: `myExtrinsic(a,b)`
- Size: `1 x 1`
- Class: `mxArray`
- Message: `myExtrinsic is an extrinsic function.`

Constant Arguments

In the MATLAB code, orange text indicates a compile-time constant argument to an entry-point function or a specialized function. The information window includes the constant value.

The screenshot shows a MATLAB code editor window titled "Function: myadd". The code is as follows:

```


1 function c = myadd(a,b)
2 c = a + b;
3 end

```

The variable `a` in the expression `a + b` on line 2 is highlighted with a blue dashed border. An information window titled "CONSTANT INFO" is open over this variable, displaying the following details:

- Constant: `a`
- Size: `1 x 1`
- Class: `double`
- Complex: `No`
- Value: `2`

Knowing the value of the constant arguments helps you to understand generated function signatures. It also helps you to see when code generation created function specializations for different constant argument values.

To export the value to a variable in the workspace, click .

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 debug errors, such as type mismatch errors, and to understand type propagation.

Visual Indicators on the Variables Tab

This table describes 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		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 26-2

Report Limitations

- The entry-point summary shows individual elements of `varagin` and `vargout`, 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 with the Workflow Advisor” on page 5-10
- “Generate HDL Code from MATLAB Code Using the Command Line Interface” on page 5-40

Generate Instantiable Code for Functions

In this section...

“How to Generate Instantiable Code for Functions” on page 5-26

“Generate Code Inline for Specific Functions” on page 5-26

“Limitations for Instantiable Code Generation for Functions” on page 5-26

You can use the **Generate instantiable code for functions** option to generate a VHDL entity or Verilog 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.

If you enable `InstantiateFunctions` , `UseMatrixTypesInHDL` has no effect.

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 5-28

“Use System object In MATLAB Design Function” on page 5-30

“Generate HDL Code” on page 5-30

“Limitations for `hdl.BlackBox`” on page 5-33

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 both `hdl.BlackBox` and the `matlab.system.mixin.Nondirect` 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 parameter in your custom HDL code.

See Also

`hdl.BlackBox`

Related Examples

- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-56

Enable MATLAB Function Block Generation

In this section...
“Requirements for MATLAB Function Block Generation” on page 5-34
“Enable MATLAB Function Block Generation” on page 5-34
“Restrictions for MATLAB Function Block Generation” on page 5-34
“Results of MATLAB Function Block Generation” on page 5-34

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 Block** 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 5-36.

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

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
```

```
% Create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

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.

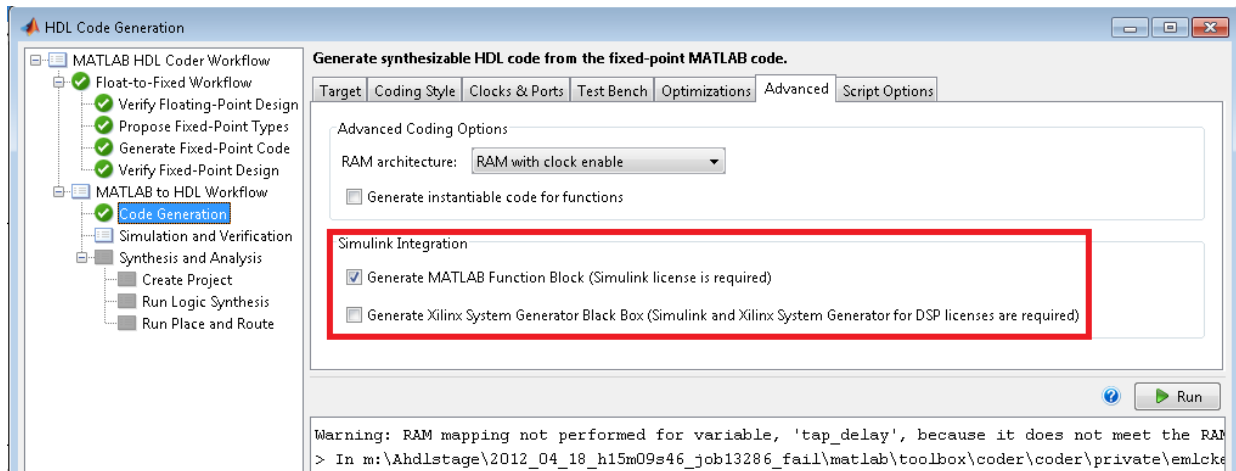
Click the Workflow Advisor button to launch the HDL Workflow Advisor.

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 Advanced tab, enable the Generate MATLAB Function Block option.



Run Floating-Point to Fixed-Point Conversion and Generate Code

To generate a MATLAB Function block, you must also convert your design from floating-point to fixed-point.

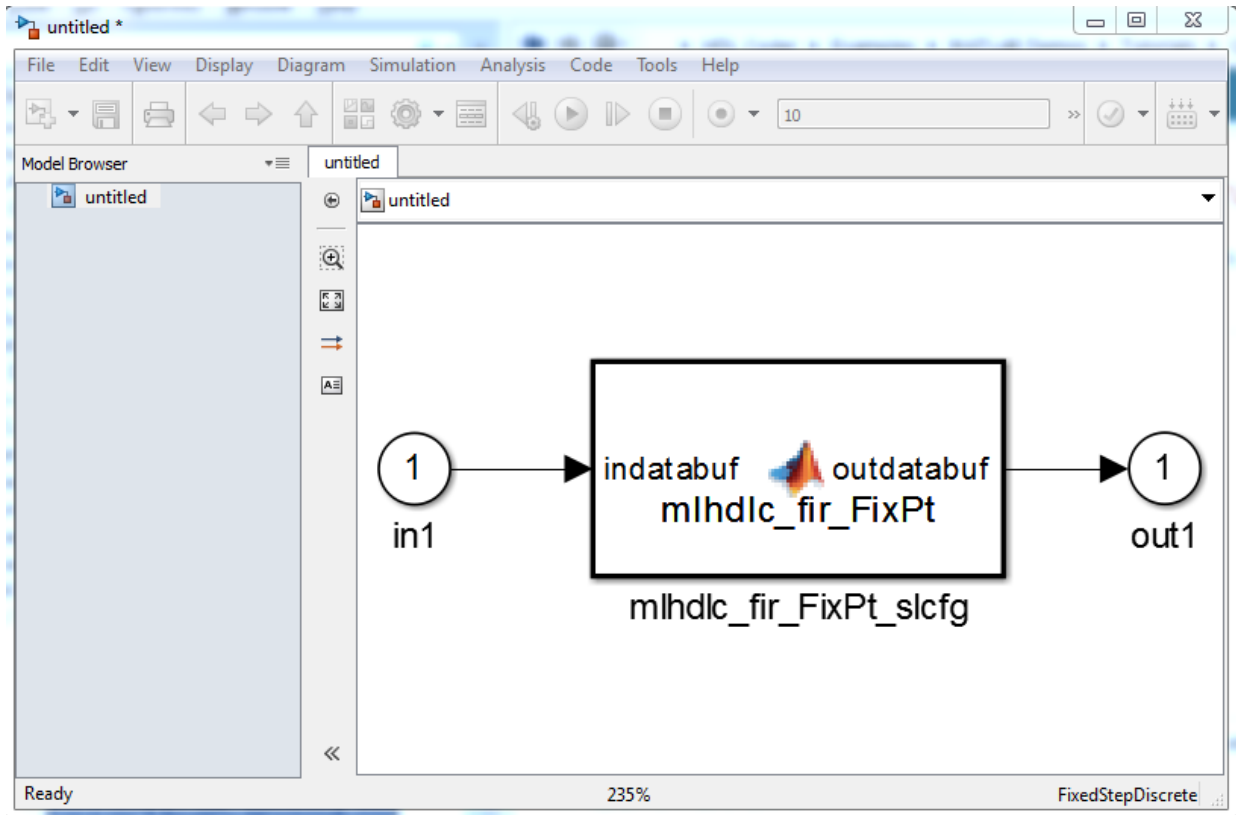
Right-click the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the Generated 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.

Clean Up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Generate HDL Code from MATLAB Code Using the Command Line Interface

This example shows how to use the HDL Coder™ command line interface to generate HDL code from MATLAB® code, including floating-point to fixed-point conversion and FPGA programming file generation.

Overview

HDL code generation with the command-line interface has the following basic steps:

- 1 Create a `fixpt` coder config object. (Optional)
- 2 Create an `hdl` coder config object.
- 3 Set config object parameters. (Optional)
- 4 Run the `codegen` command to generate code.

The HDL Coder command-line interface can use two coder config objects with the `codegen` command. The optional `fixpt` coder config object configures the floating-point to fixed-point conversion of your MATLAB code. The `hdl` coder config object configures HDL code generation and FPGA programming options.

In this example, we explore different ways you can configure your floating-point to fixed-point conversion and code generation.

The example code implements a discrete-time integrator and its test bench.

Copy the Design and Test Bench Files Into a Temporary Folder

Execute the following code to copy the design and test bench files into a temporary folder:

```
close all;
design_name = 'mlhdlc_dti';
testbench_name = 'mlhdlc_dti_tb';

mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_dti'];

cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
```

```
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

Basic Code Generation With Floating-Point to Fixed-Point Conversion

You can generate HDL code and convert the design from floating-point to fixed-point using the default settings.

You need only your design name, `mlhdlc_dti`, and test bench name, `mlhdlc_dti_tb`:

```
close all;

% Create a 'fixpt' config with default settings
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';

% Create an 'hdl' config with default settings
hdlcfg = coder.config('hdl'); %#ok<NASGU>
```

After setting up `fixpt` and `hdl` config objects, run the following `codegen` command to perform floating-point to fixed-point conversion, and generate HDL code.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

If your design already uses fixed-point types and functions, you can skip fixed-point conversion:

```
hdlcfg = coder.config('hdl'); % Create an 'hdl' config with default settings
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
codegen -config hdlcfg mlhdlc_dti
```

The rest of this example describes how to configure code generation using the `hdl` and `fixpt` objects.

Create a Floating-Point to Fixed-Point Conversion Config Object

To perform floating-point to 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_dti_tb';
```

Set 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 length. The following code configures the coder to propose types based on a fraction length of 10:

```
fixptcfg.DefaultFractionLength = 10;  
fixptcfg.ProposeWordLengthsForDefaultFractionLength = true;
```

Set the 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;
```

Enable Data Logging

The code generator runs the test bench with the design before and after 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` config object:

```
fixptcfg.LogIOForComparisonPlotting = true;
```

View the Numeric Type Proposal Report

Configure the code generator to launch the type proposal report once the fixed-point types have been proposed:

```
fixptcfg.LaunchNumericTypesReport = true;
```


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_dti_tb';
```

Set the Target Language

You can generate either VHDL or Verilog code. HDL Coder generates VHDL code by default. To generate Verilog code:

```
hdlcfg.TargetLanguage = 'Verilog';
```

Generate HDL Test Bench Code

Generate an HDL test bench from your MATLAB® test bench:

```
hdlcfg.GenerateHDLTestBench = true;
```

Simulate the Generated HDL Code Using an HDL Simulator

If you want to simulate your generated HDL code using an HDL simulator, you must also generate the HDL test bench.

Enable HDL simulation and use the ModelSim simulator:

```
hdlcfg.SimulateGeneratedCode = true;  
hdlcfg.SimulationTool = 'ModelSim'; % or 'ISIM'
```

Generate an FPGA Programming File

You can generate an FPGA programming file if you have a synthesis tool set up. Enable synthesis, specify a synthesis tool, and specify an FPGA:

```
% Enable Synthesis.  
hdlcfg.SynthesizeGeneratedCode = true;  
  
% Configure Synthesis tool.  
hdlcfg.SynthesisTool = 'Xilinx ISE'; % or 'Altera Quartus II';  
hdlcfg.SynthesisToolChipFamily = 'Virtex7';  
hdlcfg.SynthesisToolDeviceName = 'xc7vh580t';  
hdlcfg.SynthesisToolPackageName = 'hcg1155';  
hdlcfg.SynthesisToolSpeedValue = '-2G';
```

Run Code Generation

Now that you have your `fixpt` and `hdl` config objects set up, run the `codegen` command to perform floating-point to fixed-point conversion, generate HDL code, and generate an FPGA programming file:

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

Specify the Clock Enable Rate

In this section...

“Why Specify the Clock Enable Rate?” on page 5-45

“How to Specify the Clock Enable Rate” on page 5-45

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 **Drive clock enable at** 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 **Drive clock enable at** option, select **Input data rate** or **DUT base rate**.

Drive clock enable at Option	Clock Enable Behavior
Input data rate (default)	<p>Each assertion of the input clock enable produces an output clock enable assertion.</p> <p>You can assert the input clock enable at a maximum rate of once every N clocks. $N = \text{the upsampled clock rate} / \text{original clock rate}$.</p> <p>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.</p>
DUT base rate	<p>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. $N = \text{the upsampled clock rate} / \text{original clock rate}$.</p> <p>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.</p>

Specify Test Bench Clock Enable Toggle Rate

In this section...

“When to Specify Test Bench Clock Enable Toggle Rate” on page 5-47

“How to Specify Test Bench Clock Enable Toggle Rate” on page 5-47

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 5-45.

How to Specify Test Bench Clock Enable Toggle Rate

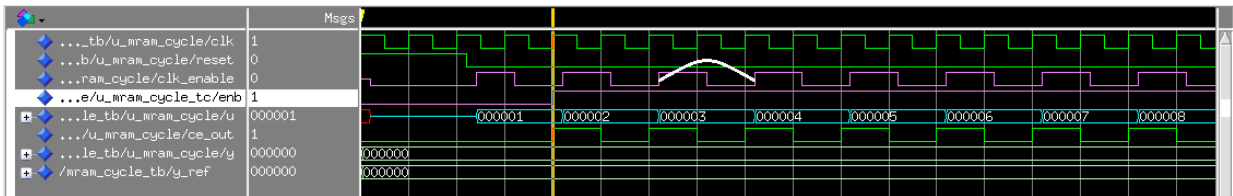
To set your test bench clock enable toggle rate:

- 1 In the HDL Workflow Advisor, select **MATLAB to HDL Workflow > Code Generation**.
- 2 In the **Clocks & Ports** tab, for the **Drive clock enable at** 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.

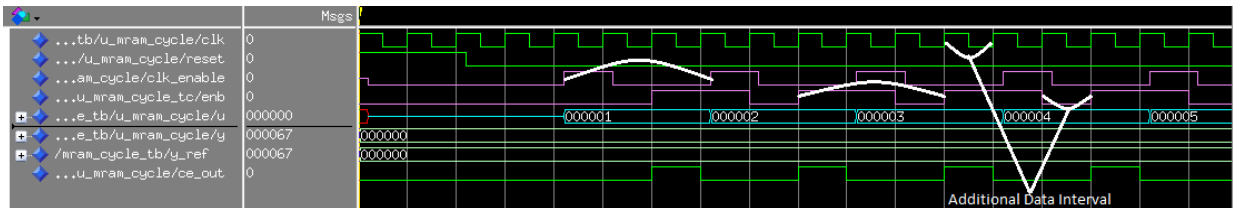
Input data interval, I	Test Bench Clock Enable Behavior
$I = 0$ (default)	Asserts at the maximum input clock enable rate, or once every N cycles. N = the upsampled clock rate / original clock rate.
$I < N$	Not valid; generates an error.

Input data interval, I	Test Bench Clock Enable Behavior
$I = N$	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.



Generate an HDL Coding Standard Report from MATLAB

In this section...
“Using the HDL Workflow Advisor” on page 5-49
“Using the Command Line” on page 5-51

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: <input type="text" value="Industry"/>						
Report options						
<input type="checkbox"/> Do not show passing rules in coding standard report						
Basic coding rules						
<input checked="" type="checkbox"/> Check for duplicate names						
<input checked="" type="checkbox"/> Check for HDL keywords in design names						
<input checked="" type="checkbox"/> Check module, instance, entity name length						
Minimum: <input type="text" value="2"/>						
Maximum: <input type="text" value="32"/>						
<input checked="" type="checkbox"/> Check signal, port, parameter name length						
Minimum: <input type="text" value="2"/>						
Maximum: <input type="text" value="40"/>						
RTL description rules						
<input type="checkbox"/> Check for clock enable signals						
<input type="checkbox"/> Check for reset signals						
<input checked="" type="checkbox"/> Check for asynchronous reset signals						
<input type="checkbox"/> Minimize use of variables						
<input checked="" type="checkbox"/> Check for initial statements that set RAM initial values						
<input checked="" type="checkbox"/> Check number of conditional regions						
Length: <input type="text" value="1"/>						
<input checked="" type="checkbox"/> Check if-else statement chain length						
Length: <input type="text" value="7"/>						
<input type="checkbox"/> Check if-else statement chain 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 26-2
- “Basic Coding Practices” on page 26-10
- “RTL Description Techniques” on page 26-25
- “RTL Design Methodology Guidelines” on page 26-64

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 Board-Independent IP Core from MATLAB Algorithm

In this section...

“Requirements and Limitations for IP Core Generation” on page 5-56

“Generate Board-Independent IP Core” on page 5-57

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 40-12.

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 “HDL Code Generation and FPGA Synthesis from a MATLAB Algorithm”.

- 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 or VHDL code, enter the file names. Enter each file name manually, separated with a semicolon (;), or by using the ... button. The source file language must match your target language.
- 4 In the **Set Target Interface** step, for each port, select an option from the **Target Platform Interfaces** drop-down list.

Ports		Build Log			
Port Name	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin		
[-] Inport					
Blink_frequency_1	numerictype(0, 4, 0)	AXI4	x"100"		
Blink_direction	numerictype(0, 1, 0)	AXI4	x"104"		
[-] Output					
LED	numerictype(0, 8, 0)	External Port			
Read_back	numerictype(0, 8, 0)	AXI4	x"108"		

- 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

- “Using IP Core Generation Workflow from MATLAB: LED Blinking”

More About

- “Custom IP Core Generation” on page 40-2
- “Board and Reference Design Registration System” on page 41-33

Minimize Clock Enables

In this section...

“Using the GUI” on page 5-60

“Using the Command Line” on page 5-60

“Limitations” on page 5-60

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 > Set Code Generation Options > Set Optimization Options > General** 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

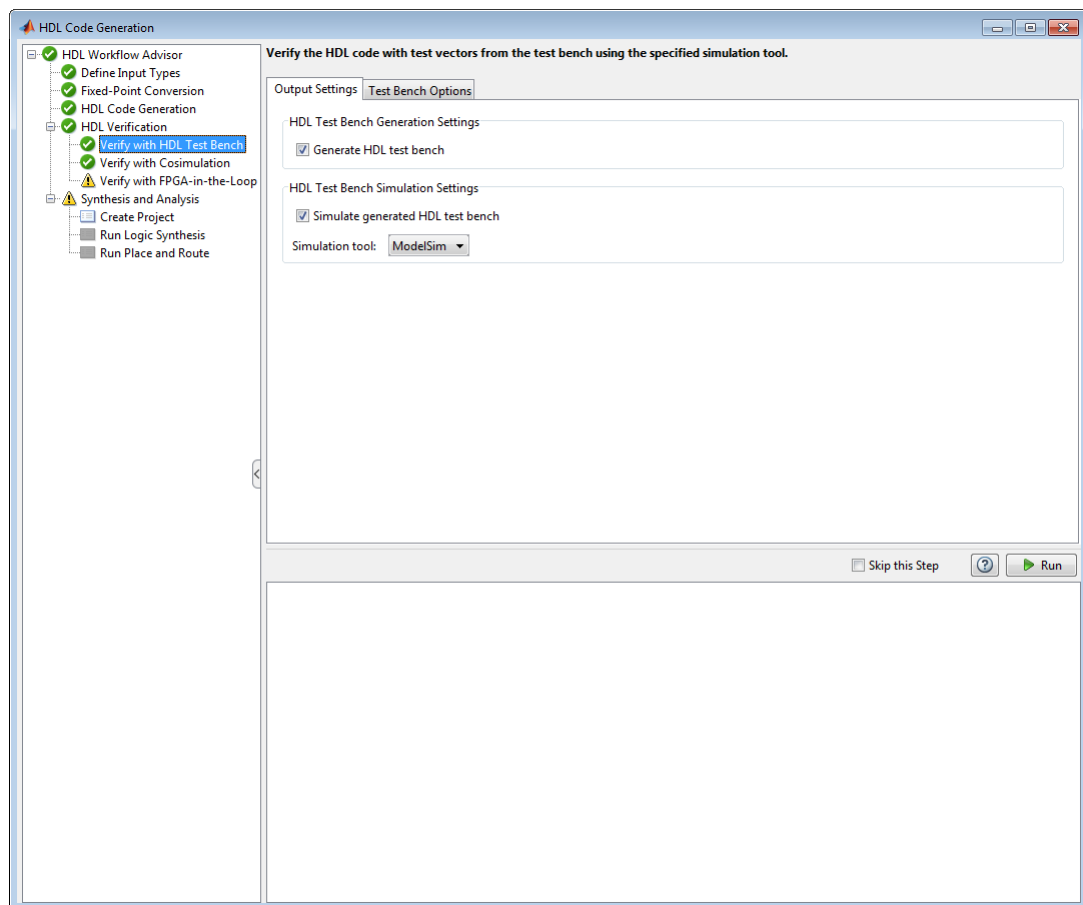
Verification

- “Verify Code with HDL Test Bench” on page 6-2
- “Test Bench Generation” 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.
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 substrate 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 5-47.

HDL Test Bench Parameter	Description
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 6-6
“Test Bench Data Files” on page 6-6
“Test Bench Data Type Limitations” on page 6-6
“Use Constants Instead of File I/O” on page 6-7

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 6-6.

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.

Deployment

Generate Synthesis Scripts

You can generate customized synthesis scripts for the following tools:

- Xilinx Vivado®
- Xilinx ISE
- Microsemi Libero
- Mentor Graphics® Precision
- Altera® Quartus II
- Synopsys® Synplify 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

- “RAM Mapping for MATLAB Code” on page 8-2
- “Map Persistent Arrays and dsp.Delay to RAM” on page 8-3
- “RAM Mapping Comparison for MATLAB Code” on page 8-8
- “Pipelining MATLAB Code” on page 8-9
- “Pipeline MATLAB Expressions” on page 8-11
- “Distributed Pipelining” on page 8-14
- “Optimize MATLAB Loops” on page 8-15
- “Constant Multiplier Optimization” on page 8-18
- “Distributed Pipelining for Clock Speed Optimization” on page 8-20
- “Map Matrices to Block RAMs to Reduce Area” on page 8-25
- “Resource Sharing of Multipliers to Reduce Area” on page 8-30
- “Loop Streaming to Reduce Area” on page 8-39
- “Constant Multiplier Optimization to Reduce Area” on page 8-45

RAM Mapping for MATLAB Code

RAM mapping is an area optimization that maps storage and delay elements in your MATLAB code to RAM. Without this optimization, storage and delay elements are mapped to registers. RAM mapping can therefore reduce the area of your design in the target hardware.

You can map the following MATLAB code elements to RAM:

- persistent array variable
- `dsp.Delay System` object
- `hdl.RAM System` object

Map Persistent Arrays and dsp.Delay to RAM

In this section...

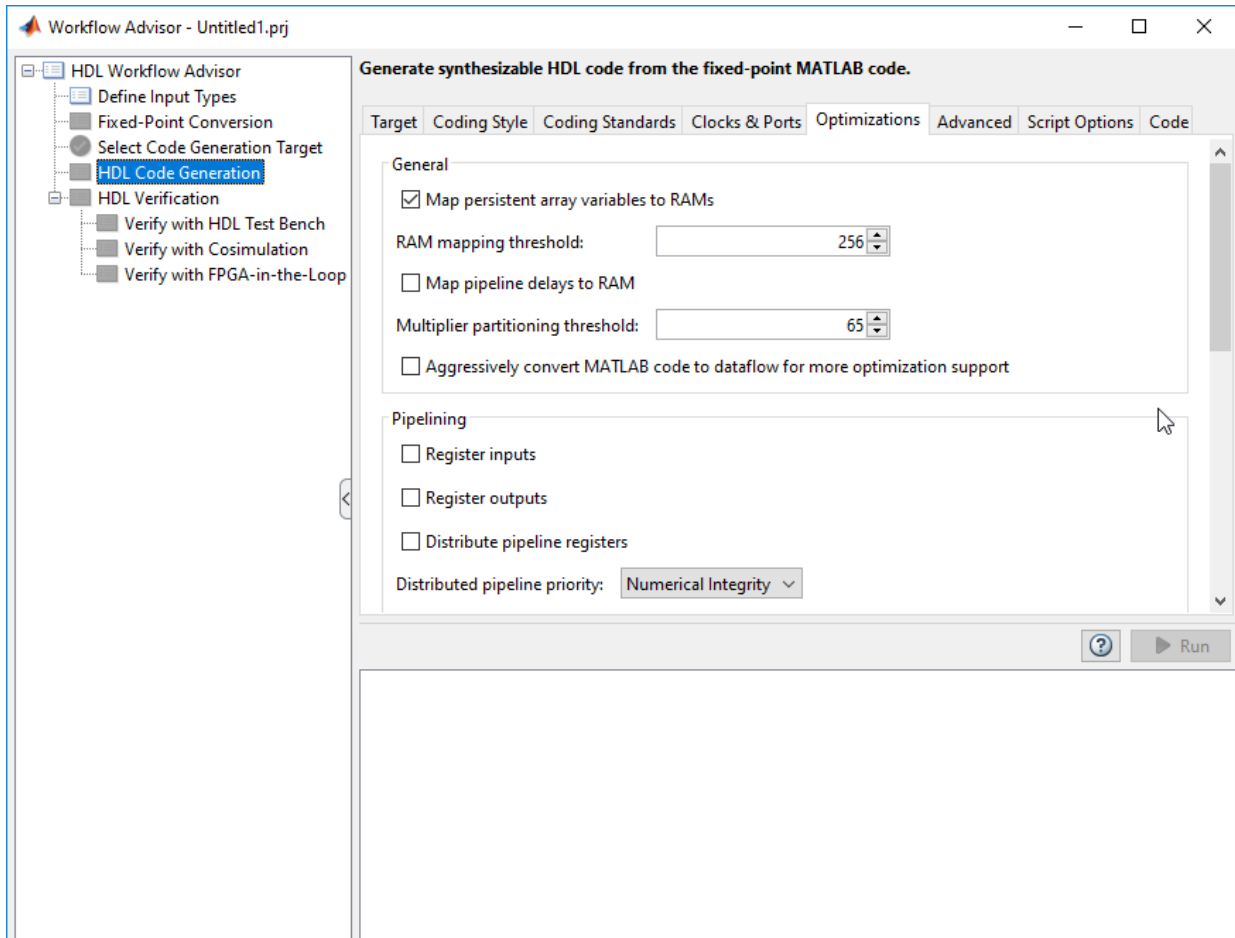
“How To Enable RAM Mapping” on page 8-3

“RAM Mapping Requirements for Persistent Arrays and System object Properties” on page 8-4

“RAM Mapping Requirements for dsp.Delay System Objects” on page 8-6

How To Enable RAM Mapping

- 1 In the HDL Workflow Advisor, select **MATLAB to HDL Workflow > 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, user-defined System object private property, or `dsp.Delay` that you want to map to RAM.



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 8-5.
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 the following conditions are true:

- Each read or write access is for a single element only. For example, submatrix access and array copies are not allowed.
- 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 allowed.
- 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 the following code, r1 does not map to RAM:

```
if (mod(i,2) > 0)
    a = r1(u);
else
    r1(i) = u;
end
```

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 follows:

```
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;
```

- `RAMSize` is greater than or equal to the `RAMMappingThreshold` value. `RAMSize` is the product `NumElements * WordLength * Complexity`.
 - `NumElements` is the number of elements in the array.
 - `WordLength` is the number of bits that represent the data type of the array.
 - `Complexity` is 2 for arrays with a complex base type; 1 otherwise.

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

A summary of the mapping behavior for a `dsp.Delay System` object is in the following table.

Map Persistent Array Variables to RAMs Option	Mapping Behavior
on	<p>A <code>dsp.Delay System</code> object maps to a block RAM when all of the following conditions are true:</p> <ul style="list-style-type: none"> • <code>Length</code> property is greater than 4. • <code>InitialConditions</code> property is 0. • Delay input data type is one of the following: <ul style="list-style-type: none"> • 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. • <code>RAMSize</code> is greater than or equal to the RAM Mapping Threshold value. <ul style="list-style-type: none"> • <code>RAMSize</code> is the product $Length * InputWordLength$. • <code>InputWordLength</code> is the number of bits that represent the input data type. <p>If any of the conditions are false, the <code>dsp.Delay System</code> object maps to registers in the HDL code.</p>
off	<p>A <code>dsp.Delay System</code> object maps to registers in the generated HDL code.</p>

RAM Mapping Comparison for MATLAB Code

hdl.RAM, dsp.Delay, persistent array variables, and user-defined System object private properties can map to RAM, but have different attributes. The following table summarizes the differences.

Attribute	hdl.RAM	dsp.Delay	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 8-6.	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for Persistent Arrays and System object Properties” on page 8-4.
Address generation and port mapping	User specified	Automatic	Automatic
Access scheduling	User specified	Automatically inferred	Automatically inferred
Overclocking	None	None	Local multirate if 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

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 8-11.

Pipeline MATLAB Expressions

In this section...

“How To Pipeline a MATLAB Expression” on page 8-11

“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, 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:

- In a loop.
- A persistent variable that maps to a state element, like a state register or RAM.
- An output of a function. For example, in the following code, you cannot add a pipeline register for an expression containing `y`:

```
function [y] = myfun(x)  
y = x + 5;  
end
```

- 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;
```

You cannot use `coder.hdl.pipeline` to insert a pipeline register for a single variable or other no-op expression. To learn how to insert a pipeline register for a function input variable, see “Port Registers” on page 8-9.

See Also

`coder.hdl.pipeline`

More About

- “Pipelining MATLAB Code” on page 8-9

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.

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.

Optimize MATLAB Loops

In this section...

“Loop Streaming” on page 8-15

“Loop Unrolling” on page 8-15

“How to Optimize MATLAB Loops” on page 8-16

“Limitations for MATLAB Loop Optimization” on page 8-16

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 8-18

“Specify Constant Multiplier Optimization” on page 8-19

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	<p>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.</p> <p>CSD minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.</p>
FCSD	<p>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.</p> <p>This option lets you achieve a greater area reduction than CSD, at the cost of decreasing clock speed.</p>

Constant Multiplier Optimization Value	Description
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**.

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

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];  
  
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);
```

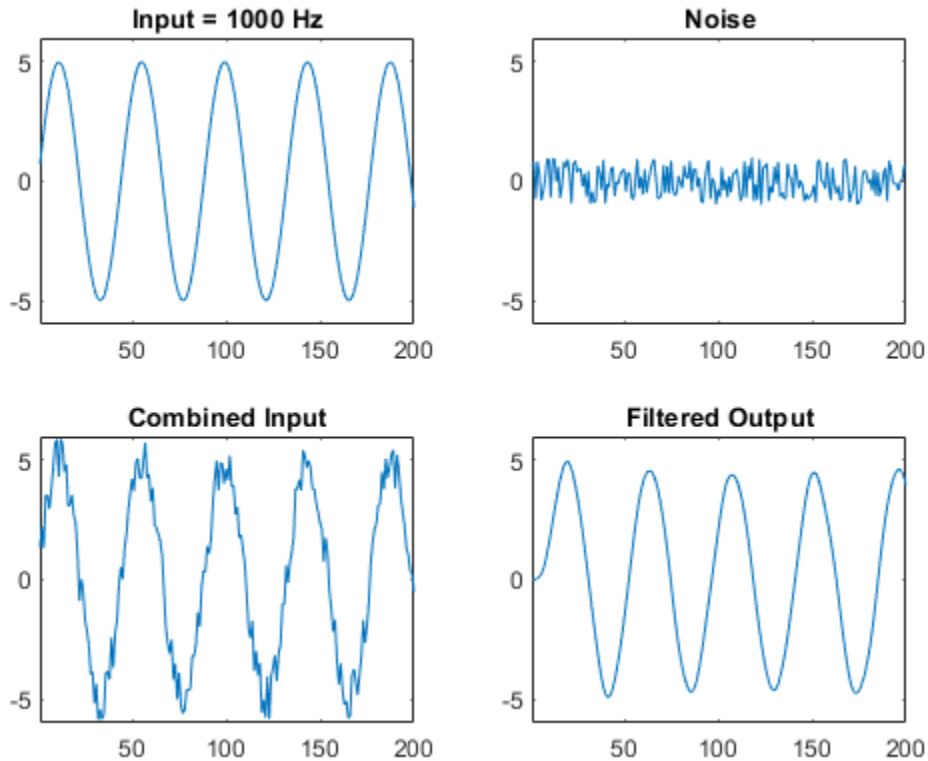


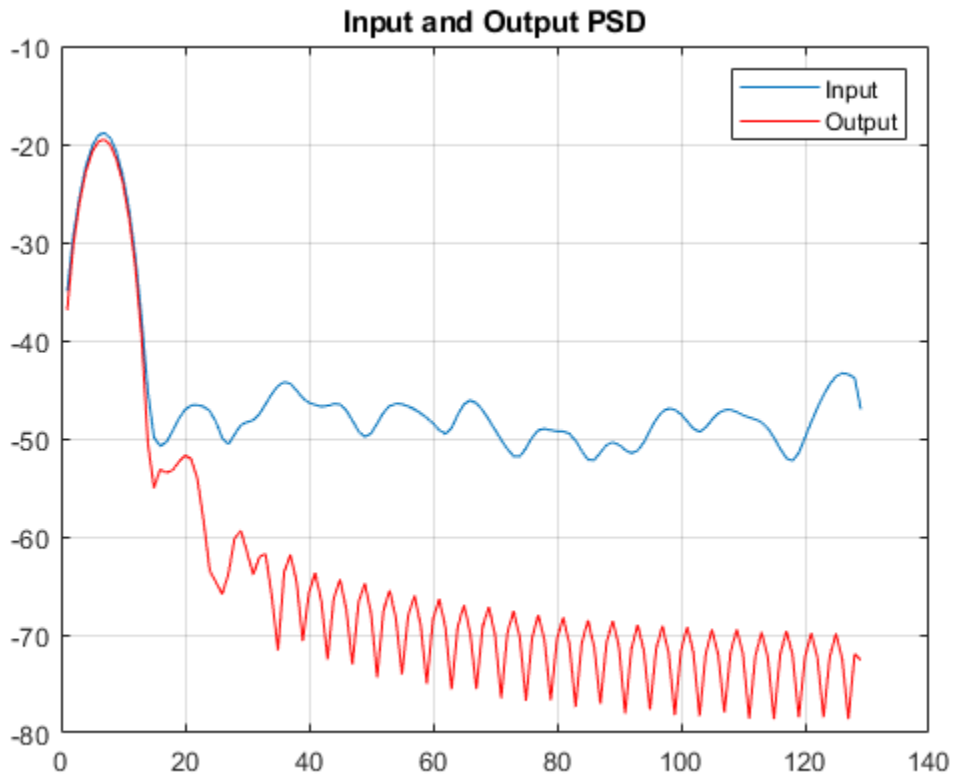
```
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

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
```





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.

Clean Up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

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

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_sobel'];
```

```
% create a temporary folder and copy the MATLAB files
```

```

cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

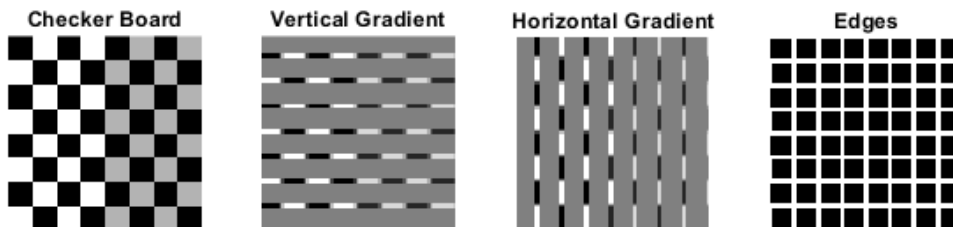
% copy the design files to the temporary directory
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);

```

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
```

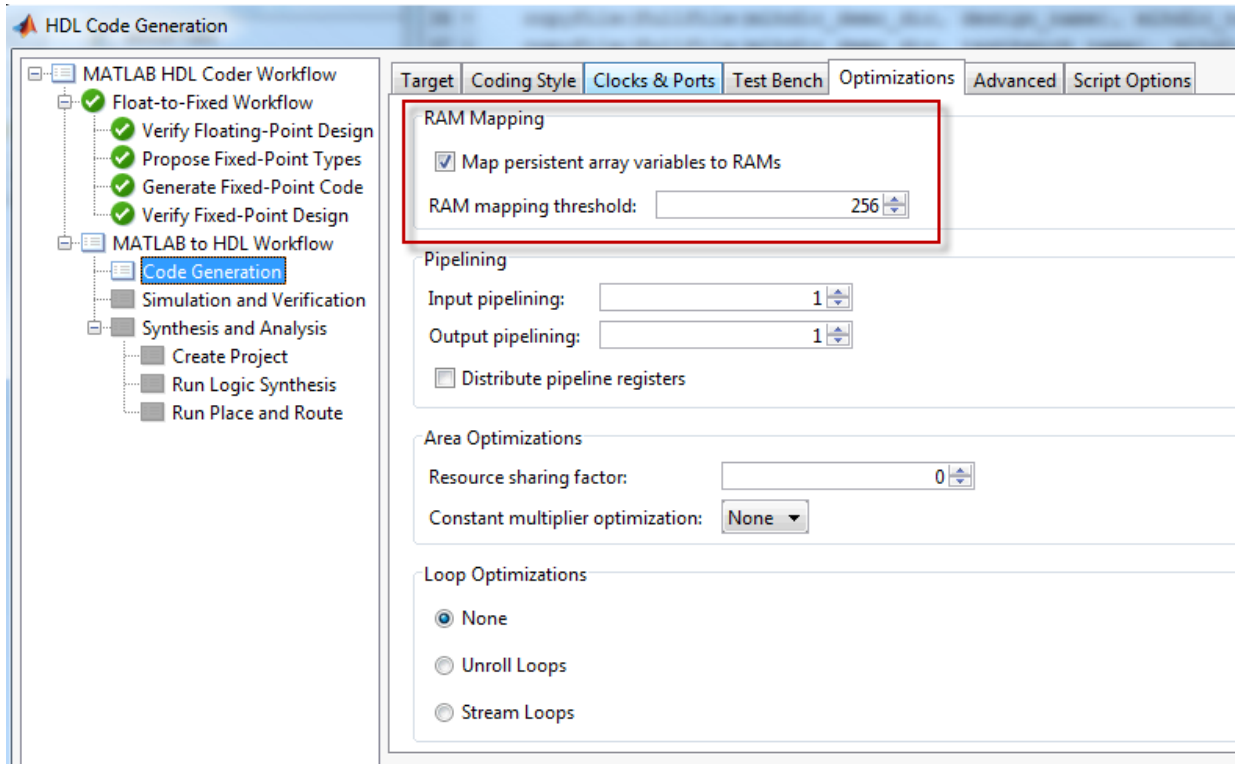
Next, add the file 'mlhdlc_sobel.m' to the project as the MATLAB function, and 'mlhdlc_sobel_tb.m' as the MATLAB test bench.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial 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.

```

### 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)

```

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.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sobel'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

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 will 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 will input a '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';
```

Let us take a 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 Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir_sharing'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

Create a New HDL Coder Project

Run the following command to create a new project:

```
coder -hdlcoder -new mlhdlc_sfir_sharing
```

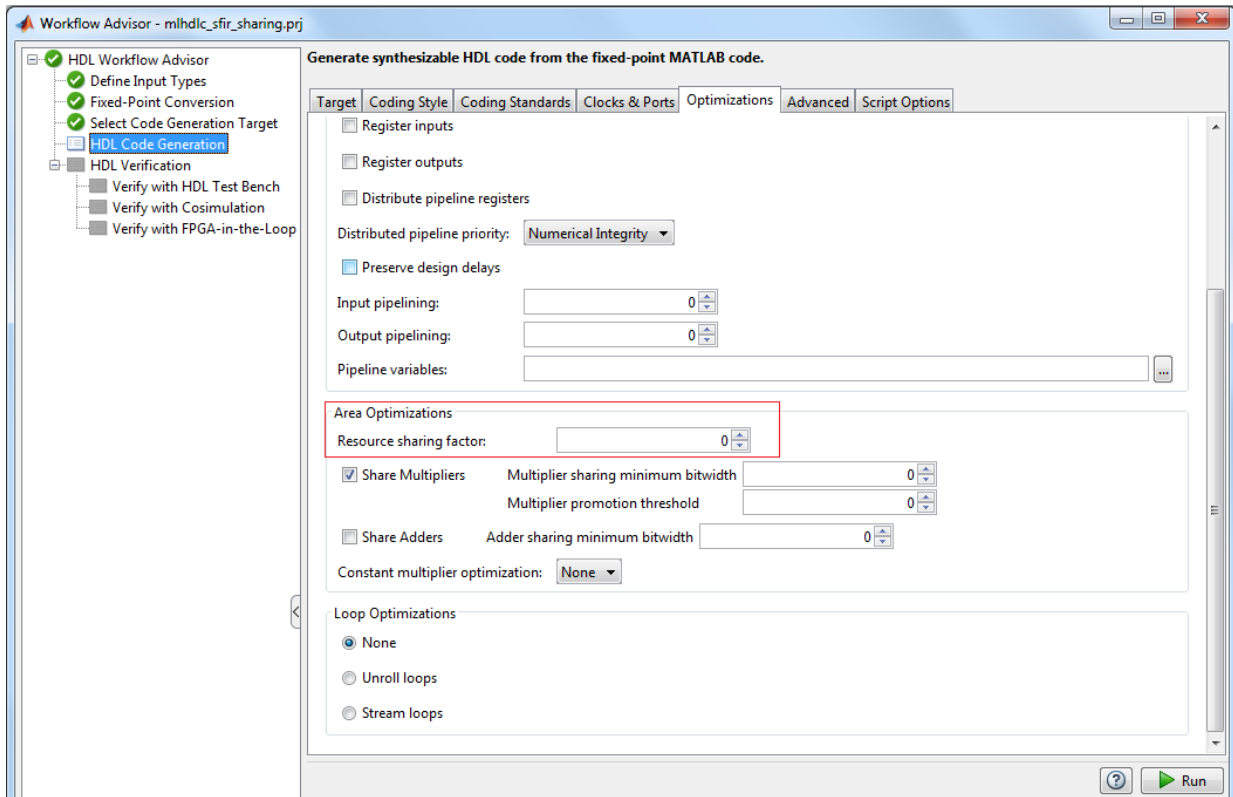
Next, add the file 'mlhdlc_sharing.m' to the project as the MATLAB Function and 'mlhdlc_sharing_tb.m' as the MATLAB Test Bench.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial 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 $N > 1$.



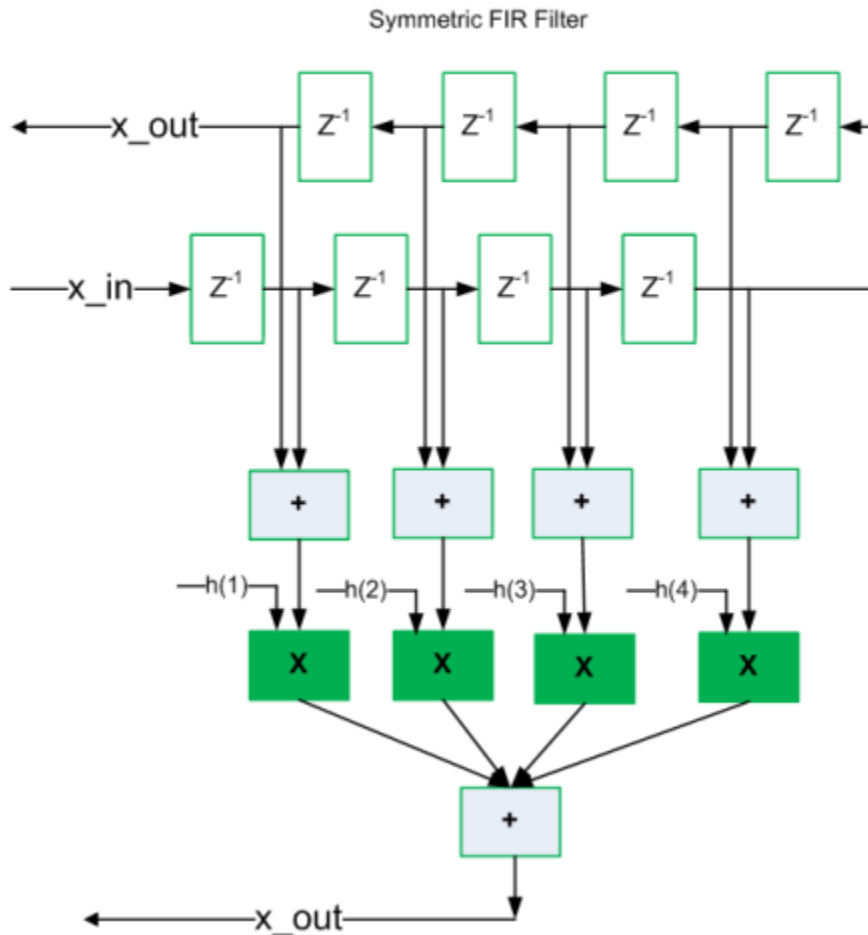
Examine the Resource Report

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

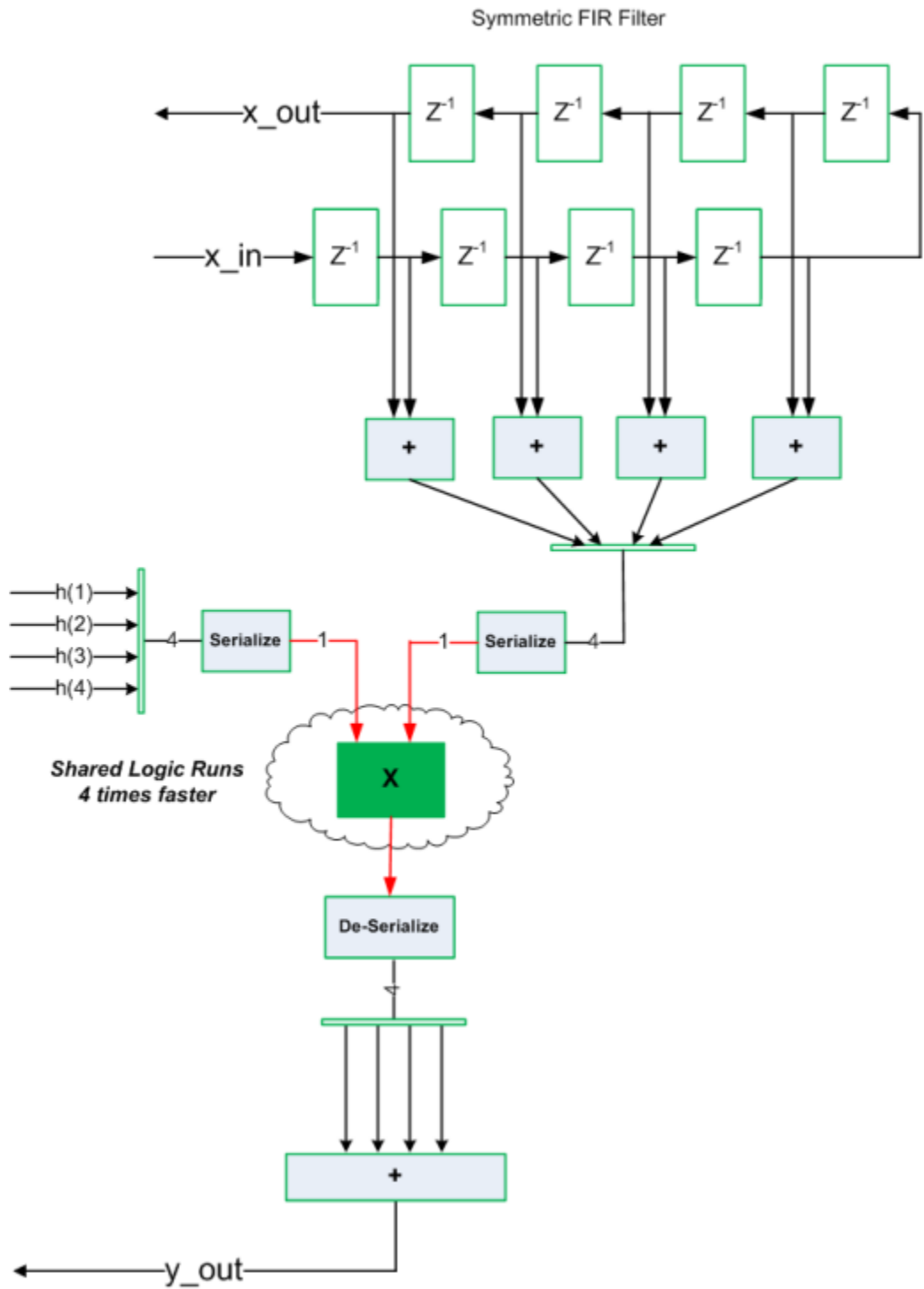
Sharing Architecture

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 conversion derived ranges 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'. For more information, see the Resource Sharing Documentation.

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.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir_sharing'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

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

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
```

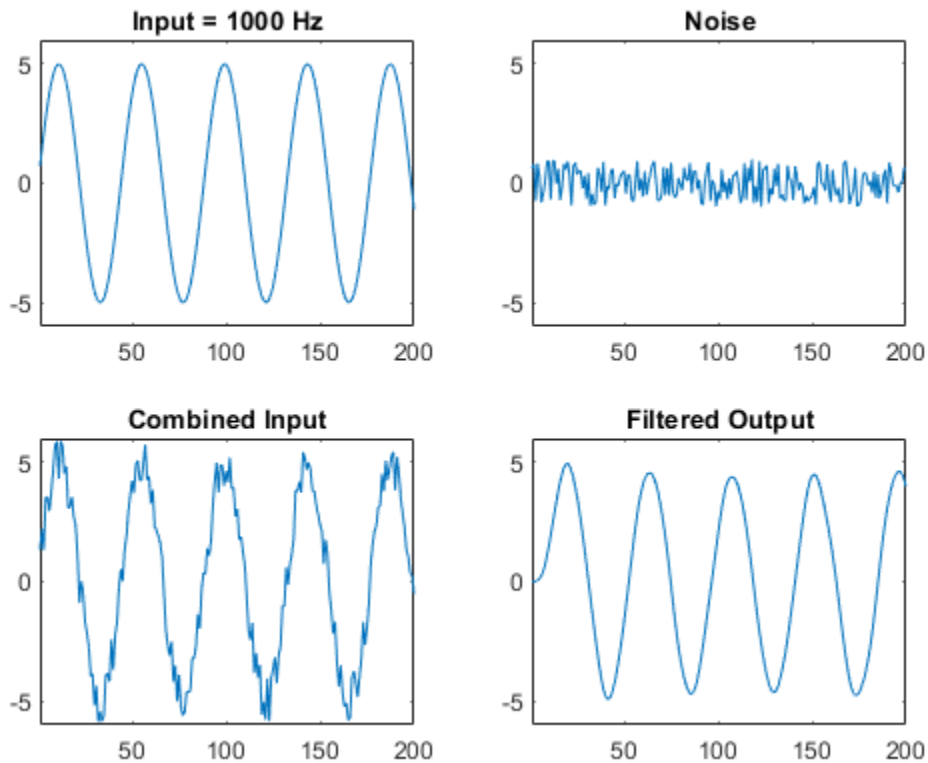
```
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);
```

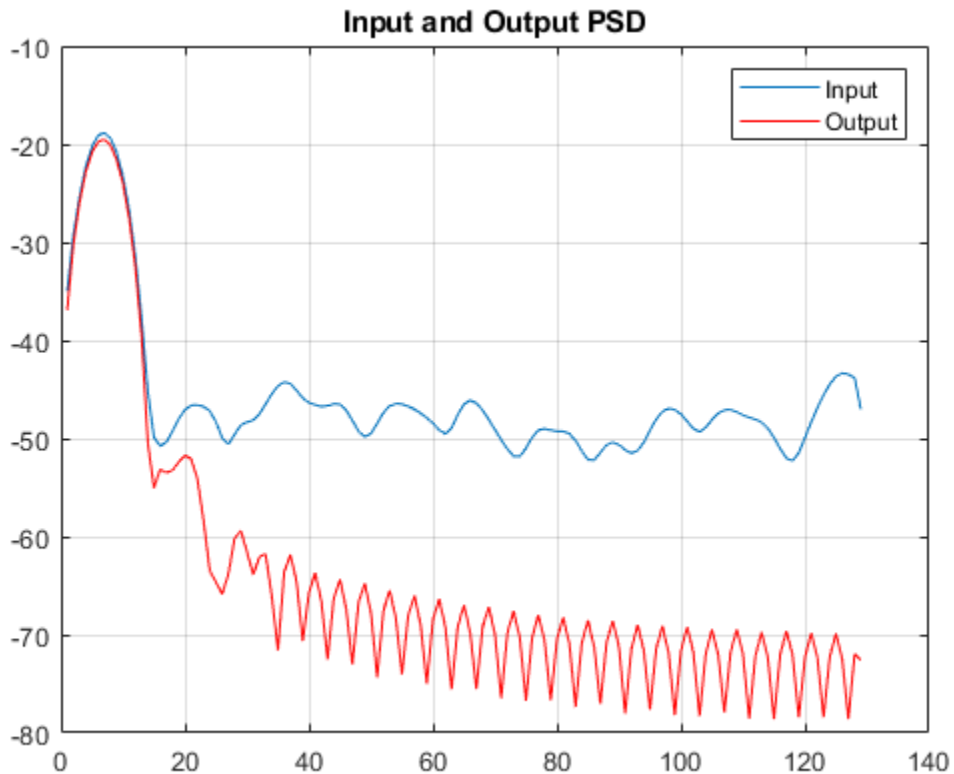
```
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

Simulate the Design

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

```
mlhdlc_fir_tb
```





Creating a New Project From the Command Line

To create a new project, enter the following command:

```
coder -hdlcoder -new fir_project
```

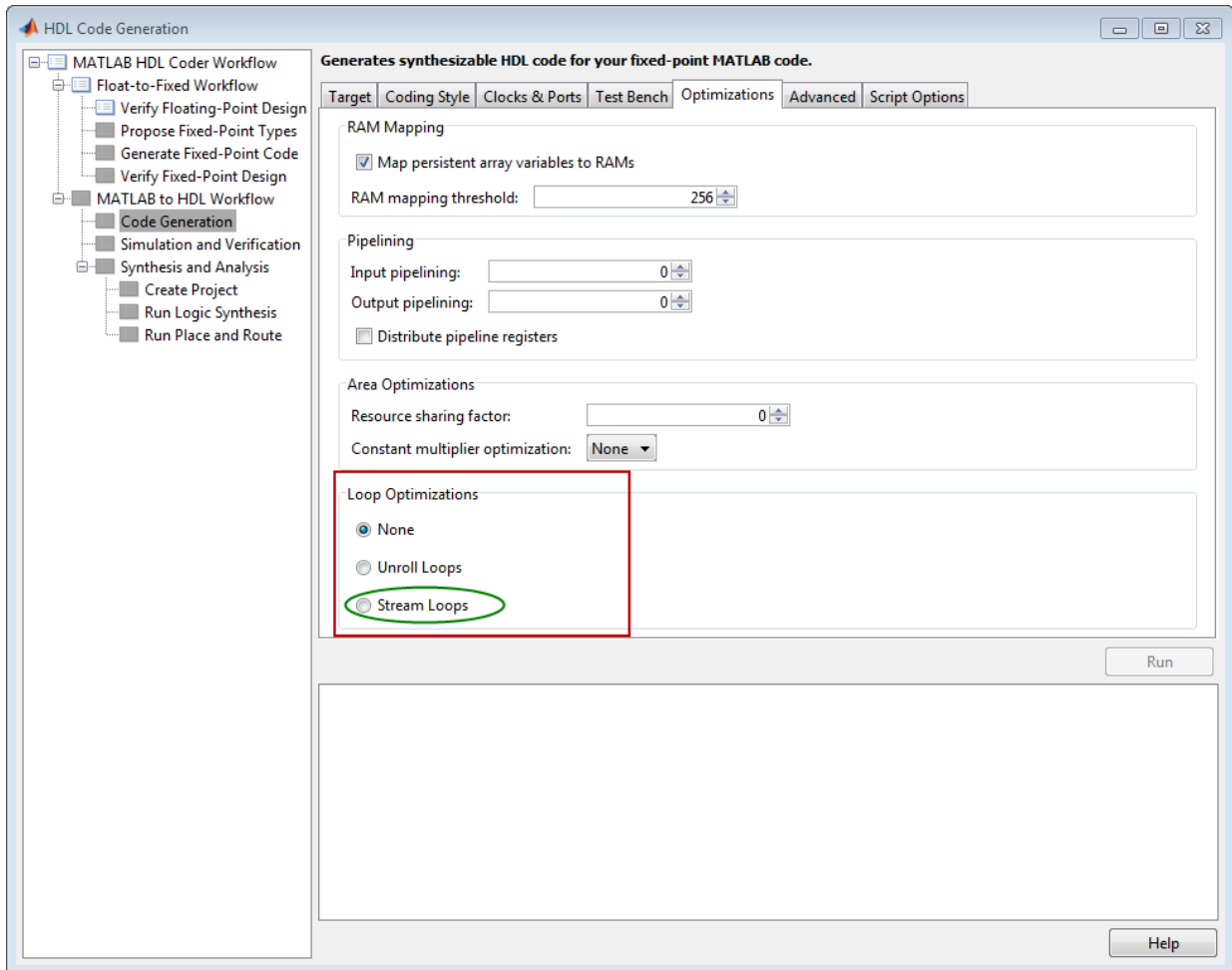
Next, add the file 'mlhdlc_fir.m' to the project as the MATLAB Function and 'mlhdlc_fir_tb.m' as the MATLAB Test Bench.

Launch the Workflow Advisor.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial 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

Known 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.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```


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

$$x_r = 0, 1, -1(\bar{1})$$

For example, here are a couple of signed digit representations for 93:

$$X_{10} = 64 + 16 + 13 = 01011101$$

$$X_{10} = 128 - 32 - 2 - 1 = 10\bar{1}000\bar{1}\bar{1}$$

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.

<i>C</i>	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 × 9, 51 = 3 × 17, 75 = 5 × 15, 85 = 5 × 17, 90 = 2 × 9 × 5, 93 = 3 × 31, 99 = 3 × 33, 102 = 2 × 3 × 17, 105 = 7 × 15, 150 = 2 × 5 × 15, 153 = 9 × 17, 155 = 5 × 31, 165 = 5 × 33, 170 = 2 × 5 × 17, 180 = 4 × 5 × 9, 186 = 2 × 3 × 31, 189 = 7 × 9, 195 = 3 × 65, 198 = 2 × 3 × 33, 204 = 4 × 3 × 17, 210 = 2 × 7 × 15, 217 = 7 × 31, 231 = 7 × 33
3	171 = 3 × 57, 173 = 8 + 165, 179 = 51 + 128, 181 = 1 + 180, 211 = 1 + 210, 213 = 3 × 71, 205 = 5 × 41, 203 = 7 × 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

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_csd'];
```

```
% create a temporary folder and copy the MATLAB files
```

```
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);
```

```
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

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 DOWNT0 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_1 <= - (((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_1, 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' & '0', 28) + resize(a2 & '0' & '0' &
336 p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
337 p22y_out_add_temp_1 <= p22y_out_add_cast + p22y_out_add_cast_1;
338 p22y_out_add_cast_2 <= resize(p22y_out_add_temp_1, 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' & '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', 28) + resize(a4 & '0' & '0' &
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 DOWNT0 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 FCSO Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'FCSO';
```

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 -- FCSO 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' & '0', 28) + resize(p22y_out_factor & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28)
338 p22y_out_add_cst <= 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' & '0' & '0', 28) + resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28)
341 p22y_out_add_cst_1 <= resize(p22y_out_mul_temp_1, 29);
342 p22y_out_add_temp <= p22y_out_add_cst + p22y_out_add_cst_1;
343 p22y_out_add_cst_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' & '0' & '0' & '0' & '0', 28) + resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28)
346 p22y_out_add_cst_3 <= resize(p22y_out_mul_temp_2, 29);
347 -- FCSO 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' & '0' & '0' & '0' & '0', 28) + resize(p22y_out_factor_1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28)
352 p22y_out_add_cst_4 <= resize(p22y_out_mul_temp_3, 29);
353 p22y_out_add_temp_1 <= p22y_out_add_cst_3 + p22y_out_add_cst_4;
354 p22y_out_add_cst_5 <= resize(p22y_out_add_temp_1, 30);
355 p22y_out_add_temp_2 <= p22y_out_add_cst_2 + p22y_out_add_cst_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 FCSO 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.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');
mlhdlc_temp_dir = [tempdir 'mlhdlc_csd'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```


HDL Workflow Advisor Reference

- “HDL Workflow Advisor” on page 9-2
- “MATLAB to HDL Code and Synthesis” on page 9-7

HDL Workflow Advisor

The screenshot displays the HDL Workflow Advisor interface for a project named 'mydesign.pj'. The left sidebar shows a tree view of tasks, with 'Fixed-Point Conversion' selected. The main workspace shows a MATLAB function named 'mldlc_sfir' with the following code:

```

20
21 %#codegen
22 function [y_out, delayed_xout] = mldlc_sfir(x_in, h_in1, h_in2, h_in3, h_in4)
23 % Symmetric FIR Filter
24
25 % declare and initialize the delay registers
26 persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
27 if isempty(ud1)
28     ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
29 end
30
31 % access the previous value of states/registers
32 a1 = ud1 + ud8; a2 = ud2 + ud7;
33 a3 = ud3 + ud6; a4 = ud4 + ud5;
34
35 % multiplier chain
36 m1 = h_in1 * a1; m2 = h_in2 * a2;

```

Below the code editor is a table showing the variable declarations and their properties:

Variable	Type	Sim Min	Sim Max	Whol...	Proposed Type	Lo...	Max Diff
Input							
x_in	double			No			
h_in1	double			No			
h_in2	double			No			
h_in3	double			No			
h_in4	double			No			
Output							
y_out	double			No			
delayed_xout	double			No			
Persistent							
ud1	double			No			
ud2	double			No			

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.

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.

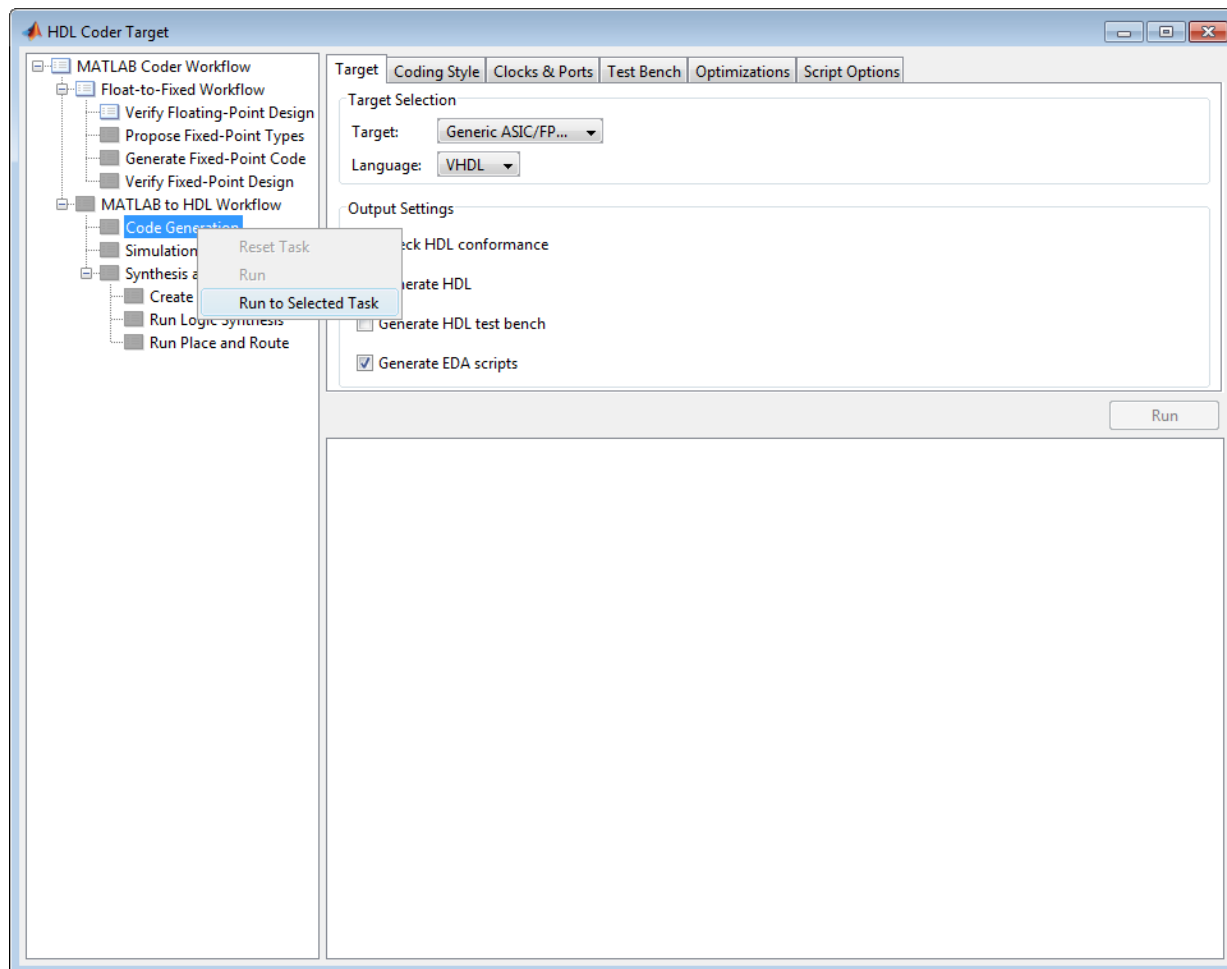
Alternatively, right-click the task to open the context menu. From the context menu, select **Run** to run the selected task.

- 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 9-7

“Code Generation: Target Tab” on page 9-7

“Code Generation: Coding Style Tab” on page 9-8

“Code Generation: Clocks and Ports Tab” on page 9-10

“Code Generation: Test Bench Tab” on page 9-12

“Code Generation: Optimizations Tab” on page 9-14

“Simulation and Verification” on page 9-16

“Synthesis and Analysis” on page 9-16

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 or Verilog) 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 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 or Verilog 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 substrate 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.

Conditions	Recommended Action
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.

HDL Code Generation from Simulink

Model Design for HDL Code Generation

- “Signal and Data Type Support” on page 10-2
- “Use Simulink Templates for HDL Code Generation” on page 10-9
- “Generate DUT Ports for Tunable Parameters” on page 10-23
- “Generate Parameterized Code for Referenced Models” on page 10-27
- “Generating HDL Code for Subsystems with Array of Buses” on page 10-29
- “Generate HDL Code for Blocks Inside For Each Subsystem” on page 10-33
- “Model and Debug Test Point Signals with HDL Coder™” on page 10-39
- “Allocate Sufficient Delays for Floating-Point Operations” on page 10-49
- “Optimize Generated HDL Code for Multirate Designs with Large Rate Differentials” on page 10-55
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- “Numeric Considerations with Native Floating-Point” on page 10-69
- “Latency Values of Floating Point Operators” on page 10-77
- “Latency Considerations with Native Floating Point” on page 10-83
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92
- “Verify the Generated Code from Native Floating-Point” on page 10-101
- “Simulink Blocks Supported with Native Floating-Point” on page 10-106
- “Supported Data Types and Scope” on page 10-110
- “Verilog HDL Import: Import Verilog Code and Generate Simulink Model” on page 10-114
- “Supported Verilog Constructs for HDL Import” on page 10-117
- “Limitations of Verilog HDL Import” on page 10-123
- “Using the Float Typecast Block” on page 10-138

Signal and Data Type Support

In this section...
“Buses” on page 10-2
“Enumerations” on page 10-3
“Matrices” on page 10-4
“Unsupported Signal and Data Types” on page 10-8

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” (Simulink). 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
- Zero-Order Hold

In addition, subsystems, models, and these user-defined functions support buses for simulation and HDL code generation:

- Subsystem
- Model references, see “Model Referencing for HDL Code Generation” on page 27-2.
- 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 values must be monotonically increasing.
- The enumeration strings must have unique names and must not use a reserved keyword in the Verilog or VHDL language.

- If your target language is Verilog, 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
- FPGA Turnkey 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: <ul style="list-style-type: none">• Dead Zone• Relay• Saturation

HDL Coder Block Library	Supported blocks
Discrete	These blocks are supported: <ul style="list-style-type: none"> • 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
HDL Floating Point Operations	The Rounding Function block is supported.
HDL Operations	All blocks in this library are supported.
HDL RAMs	Blocks in this library are not supported.
HDL Subsystems	Blocks in this library are not supported.
Logic and Bit Operations	These blocks are supported: <ul style="list-style-type: none"> • Bit Clear • Bit Concat • Bit Reduce • Bit Rotate • Bit Set • Bit Shift • Bit Slice • Extract Bits • Logical Operator • Relational Operator • Shift Arithmetic
Lookup Tables	Blocks in this library are not supported.

HDL Coder Block Library	Supported blocks
Math Operations	<p>These blocks are supported:</p> <ul style="list-style-type: none"> • 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 block is supported.

HDL Coder Block Library	Supported blocks
Signal Attributes	These blocks are supported: <ul style="list-style-type: none"> • Data Type Conversion • Data Type Duplicate • Probe • Rate Transition • Signal Conversion • Signal Specification
Signal Routing	These blocks are supported: <ul style="list-style-type: none"> • Mux • Multiport Switch • Selector • Switch
Sources	These blocks are supported: <ul style="list-style-type: none"> • Constant • Enumerated Constant • Inport
Sinks	These blocks are supported: <ul style="list-style-type: none"> • Display • Outport • Scope • To Workspace • To File • XY Graph
User-Defined Functions	The MATLAB Function block is supported.

The code generator does not support matrix types at the interfaces of the Subsystem that you generate HDL code for. Use a Reshape block to convert the matrix input to a 1-D array at the interface. Inside the Subsystem, use another Reshape block that converts the 1-D array back to the matrix type with the dimensionality that you specified.

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 10-29

More About

- “Signal Types” (Simulink)
- “About Data Types in Simulink” (Simulink)
- “Composite (Bus) Signals” (Simulink)
- “Use Enumerated Data in Simulink Models” (Simulink)
- “Enumerated Data” (Stateflow)

Use Simulink Templates for HDL Code Generation

In this section...

“Create Model Using HDL Coder Model Template” on page 10-9

“HDL Coder Model Templates” on page 10-10

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

```
sLibraryBrowser
```

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

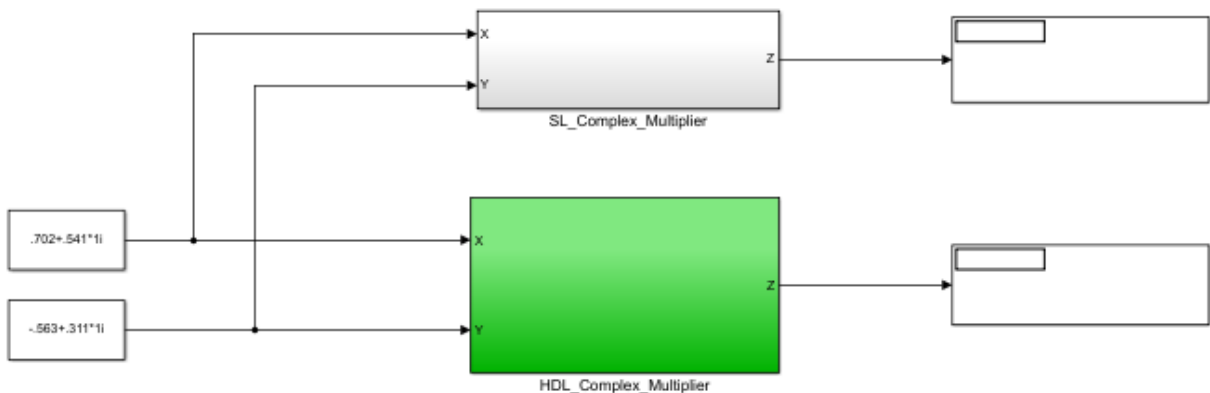
Complex Multiplier

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.

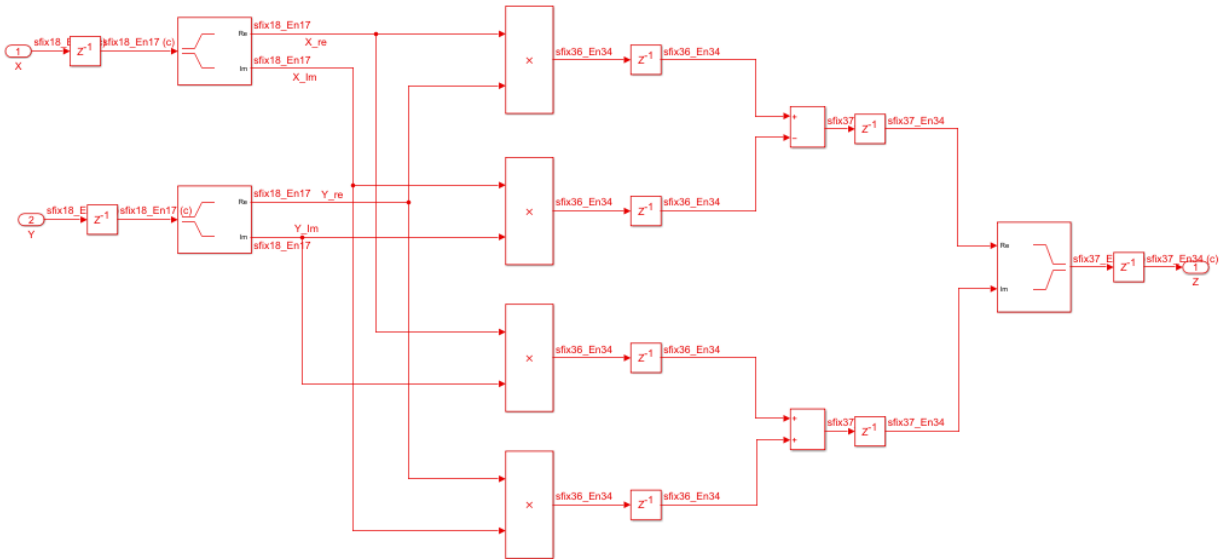
The template applies the following best practices:

- 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.

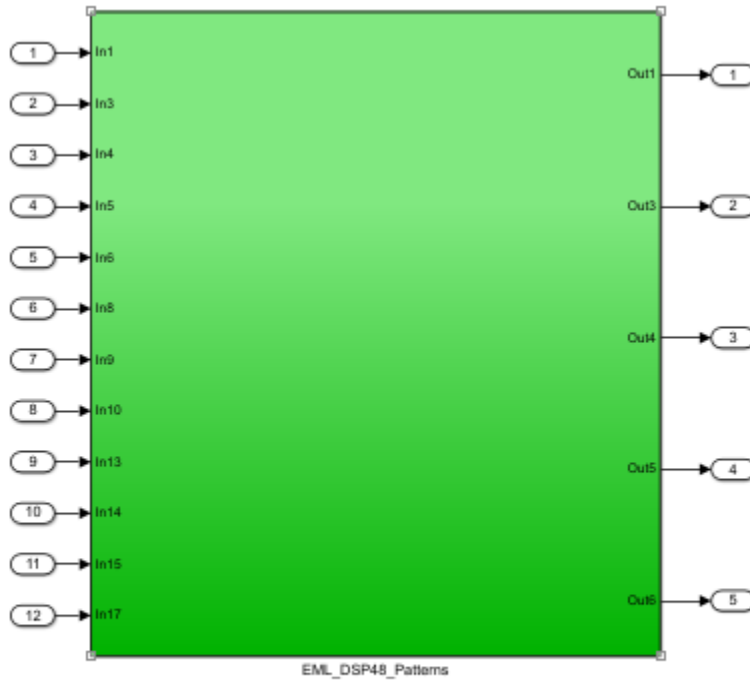


Copyright 2014-2016 The MathWorks, Inc.



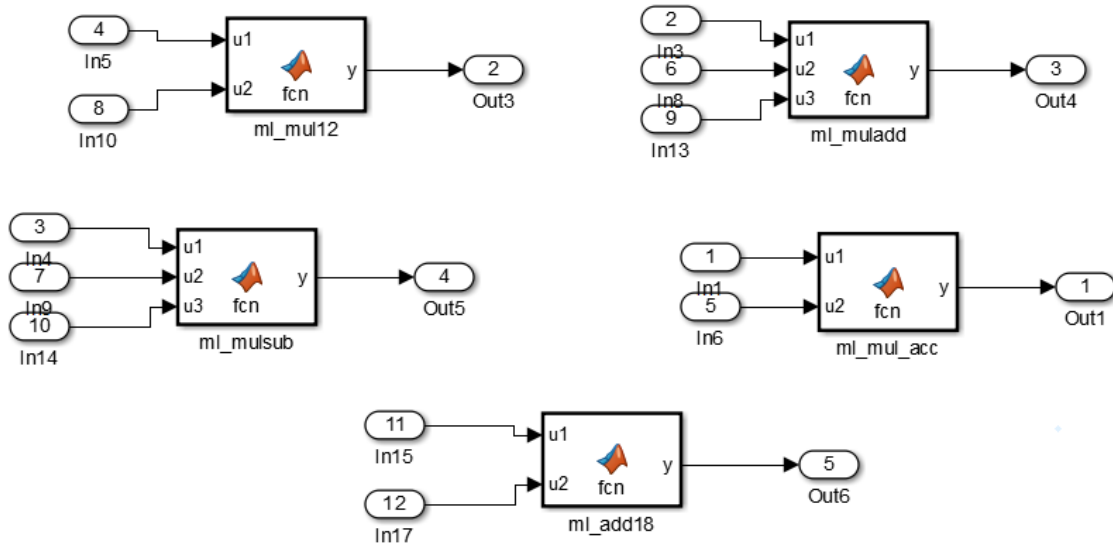
MATLAB Arithmetic

The MATLAB Arithmetic template contains MATLAB arithmetic operations that infer DSP48s in hardware.



Copyright 2014-2016 The MathWorks, Inc.

untitled ▶ EML_DSP_48_Patterns



For example, the `ml_mul_acc` MATLAB Function block shows how to write a multiply-accumulate operation in MATLAB. `hdlfimath` on page 29-35 applies fixed-point math settings for HDL code generation.

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

% design of a 6x6 multiplier
% 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

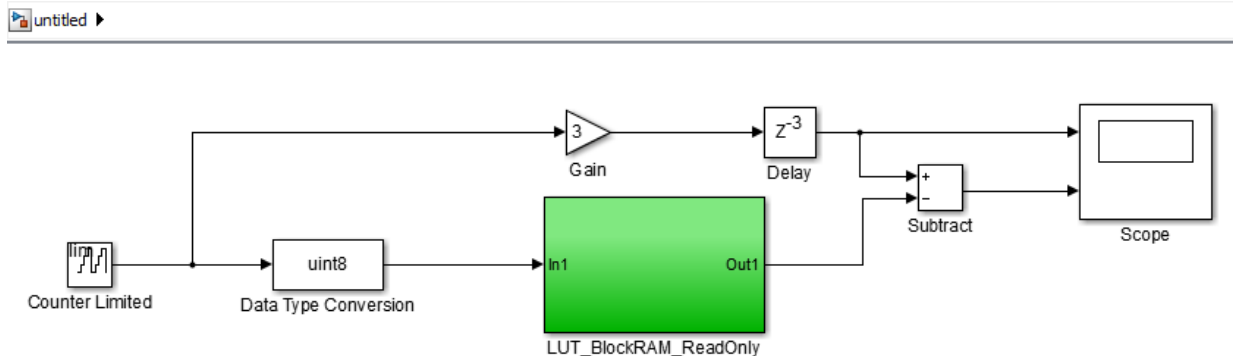
The ROM template is a design pattern that maps to a ROM in hardware.

The template applies the following best practices:

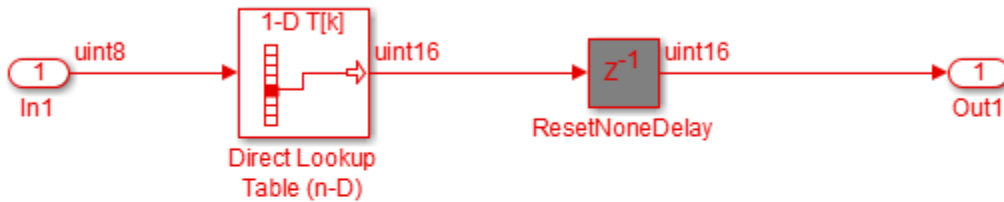
- At the output of the lookup table, there is a Delay block with ResetType = none.
- The lookup table is structured such that the spacing between breakpoints is a power of two.

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.



untitled ▶ LUT_BlockRAM_ReadOnly ▶ HDL ROM



```
x=(0:99);
Scale_by_3_LUT=3*x;
pad=2^nextpow2(length(Scale_by_3_LUT))-length(Scale_by_3_LUT);
Scale_by_3_LUT_pad=[Scale_by_3_LUT;zeros(pad,1)];
```

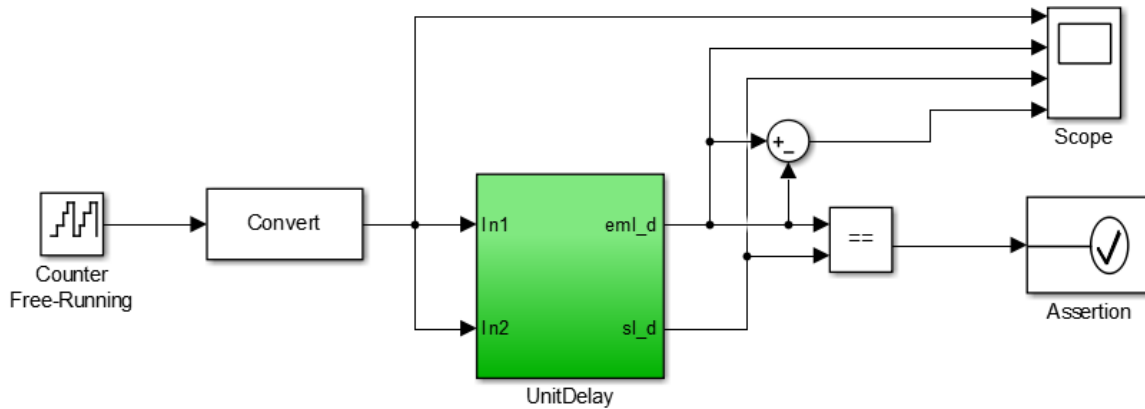
Register

The Register template shows how to model hardware registers:

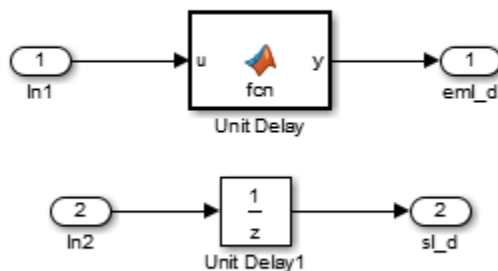
- In Simulink, using the Delay block.
- In MATLAB, using persistent variables.

This design pattern also shows how to use `cast` to propagate data types automatically.

untitled ▶



untitled ▶ UnitDelay



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

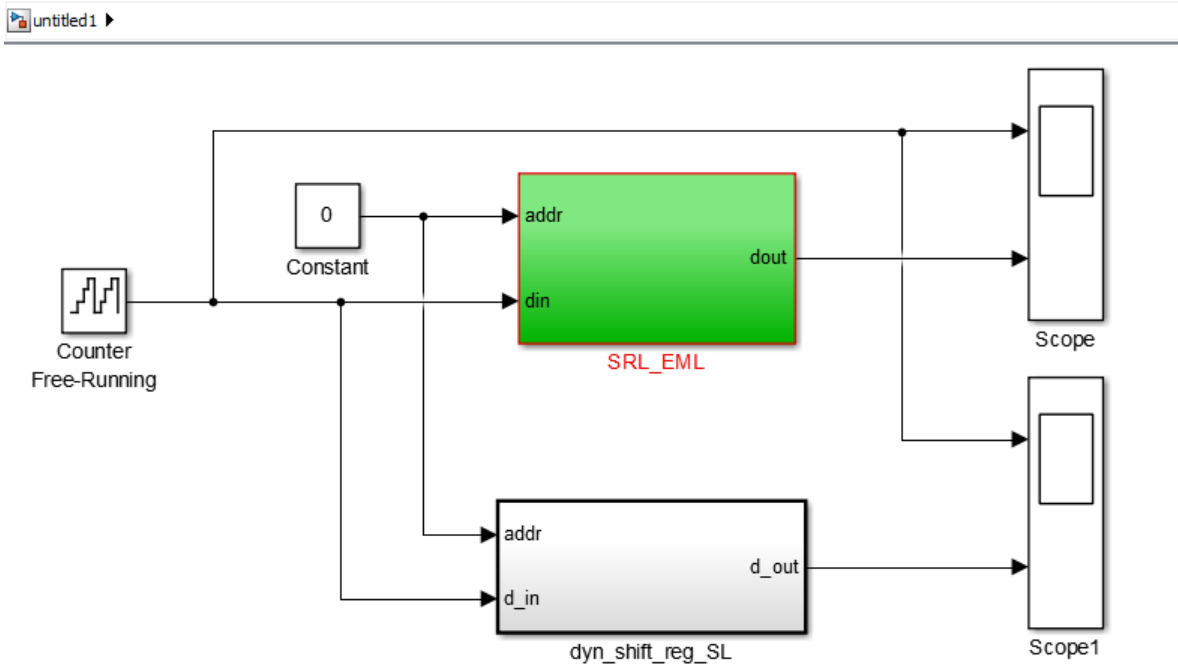
% return delayed input from last sample time hit
y = u_d;

% store the current input
u_d = u;

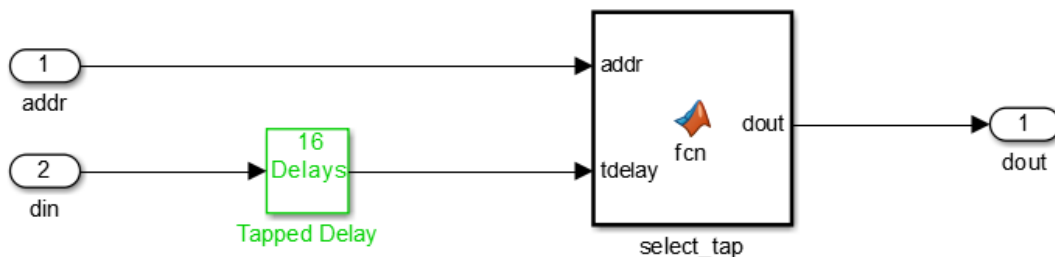
```

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.



untitled1 ▸ SRL_EML



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 and VHDL are zero-based.

The template also applies the following best practices for mapping to an SRL16 in hardware:

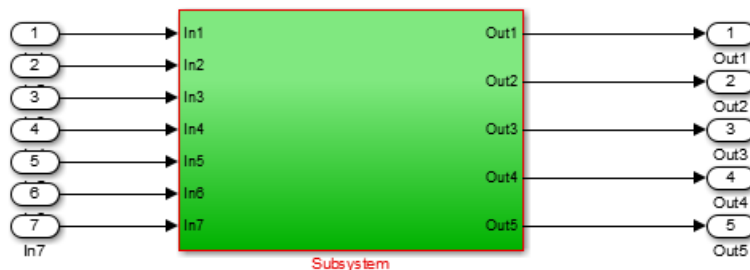
- For the Tapped Delay block:
 - In the Block Parameters dialog box, **Include current input in output vector** is not enabled.
 - In the HDL Block Properties dialog box, **ResetType** is set to none.
- For the Subsystem block, in the HDL Block Properties dialog box, **FlattenHierarchy** is set to on.

Simulink Hardware Patterns

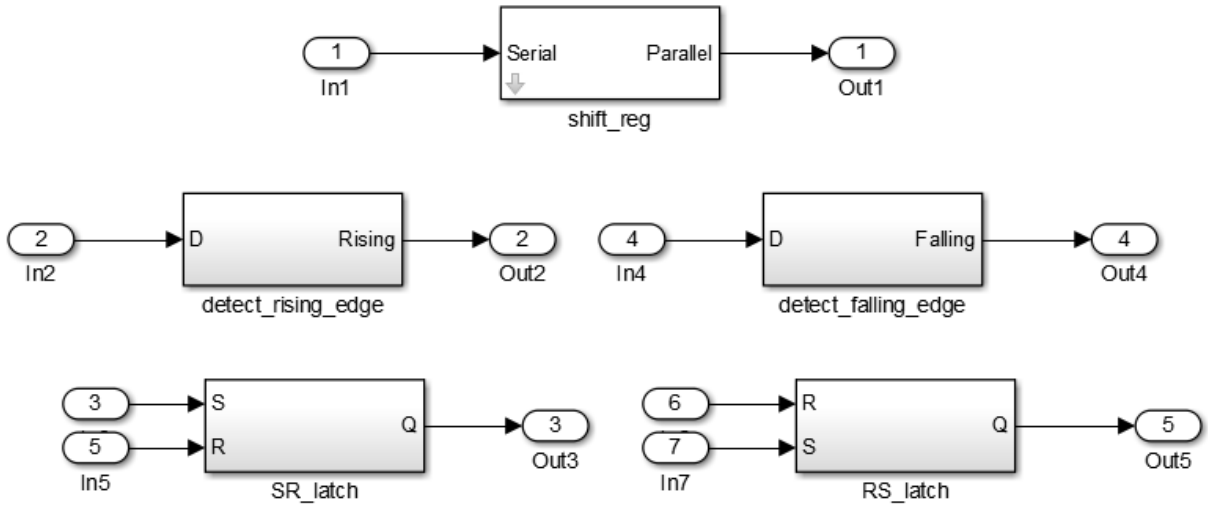
The Simulink Hardware Patterns template contains design patterns for common hardware operations:

- Serial-to-parallel shift register
- Detect rising edge
- Detect falling edge
- SR latch
- RS latch

untitled2 ▶

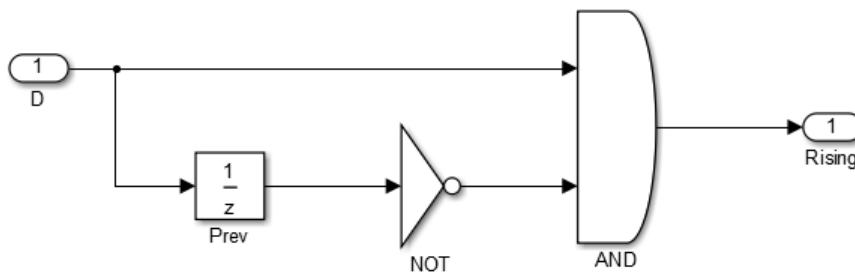


untitled ▶ Subsystem ▶

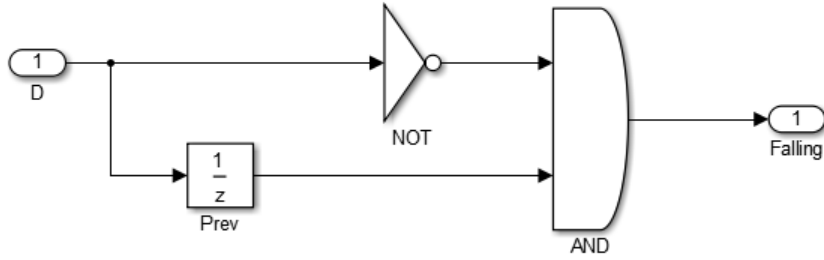


For example, the design patterns for rising edge detection and falling edge detection:

untitled ▶ Subsystem ▶ detect_rising_edge

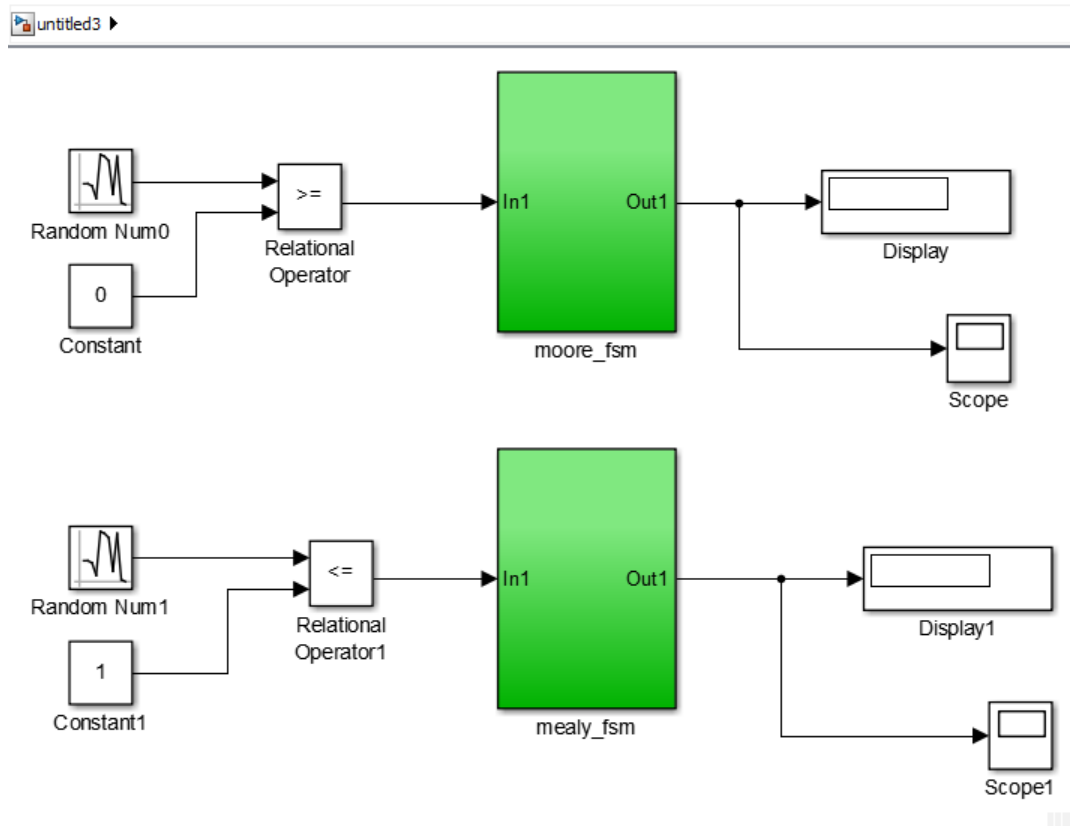


untitled ▸ Subsystem ▸ detect_falling_edge



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 Code Generation” on page 3-5.

See Also

`hdlsetup` | `makehdl`

More About

- “Create Simulink Model for HDL Code Generation”
- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “Hardware Modeling with MATLAB Code”

Generate DUT Ports for Tunable Parameters

In this section...

“Prerequisites” on page 10-24

“Create and Add Tunable Parameter That Maps to DUT Ports” on page 10-24

“Generated Code” on page 10-24

“Limitations” on page 10-25

“Use Tunable Parameter in Other Blocks” on page 10-25

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 10-25.

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?” (Simulink).

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” (Simulink).

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';
```

Alternatively, using the Model Explorer, you can create a tunable parameter and set **Storage Class** to `ExportedGlobal`. See “Create Data Objects from Built-In Data Class Package Simulink” (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 DOWNT0 0); -- sfix16_En5
```

```

        myParam      : IN  std_logic_vector(15 DOWNT0 0); -- sfix16_En5 Tunable port
        Out1         : OUT std_logic_vector(31 DOWNT0 0) -- sfix32_En10
    );
END s;

ARCHITECTURE rtl OF s IS

    -- Signals
    SIGNAL myParam_signed : signed(15 DOWNT0 0); -- sfix16_En5
    SIGNAL In1_signed     : signed(15 DOWNT0 0); -- sfix16_En5
    SIGNAL Gain_out1      : signed(31 DOWNT0 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” on page 16-49 check box is left cleared by default.

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 10-24
- 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 10-27

Generate Parameterized Code for Referenced Models

In this section...

“Parameterize Referenced Model for HDL Code Generation” on page 10-27

“Restrictions” on page 10-27

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 or Verilog 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 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” (Simulink).

- 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 Atomic Subsystems” on page 27-20
- “Generate DUT Ports for Tunable Parameters” on page 10-23

More About

- “Specify a Different Value for Each Instance of a Reusable Model” (Simulink)
- “Model Referencing for HDL Code Generation” on page 27-2

Generating HDL Code for Subsystems with Array of Buses

In this section...

“How HDL Coder Generates Code for Array of Buses” on page 10-29

“Array of Buses Limitations” on page 10-32

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

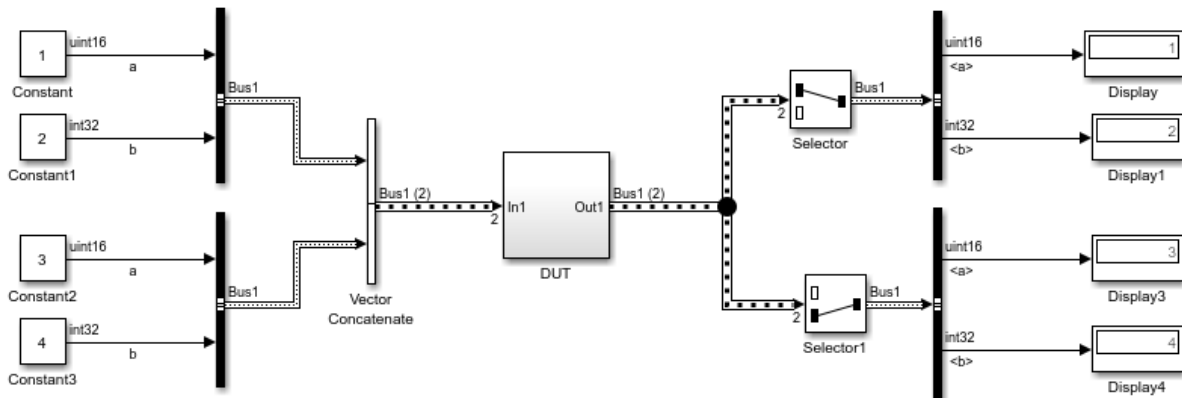
For more information, see “Combine Buses into an Array of Buses” (Simulink).

You can generate HDL code for virtual and nonvirtual blocks that Simulink supports with an array of buses. For more information, see “Use Arrays of Buses in Models” (Simulink).

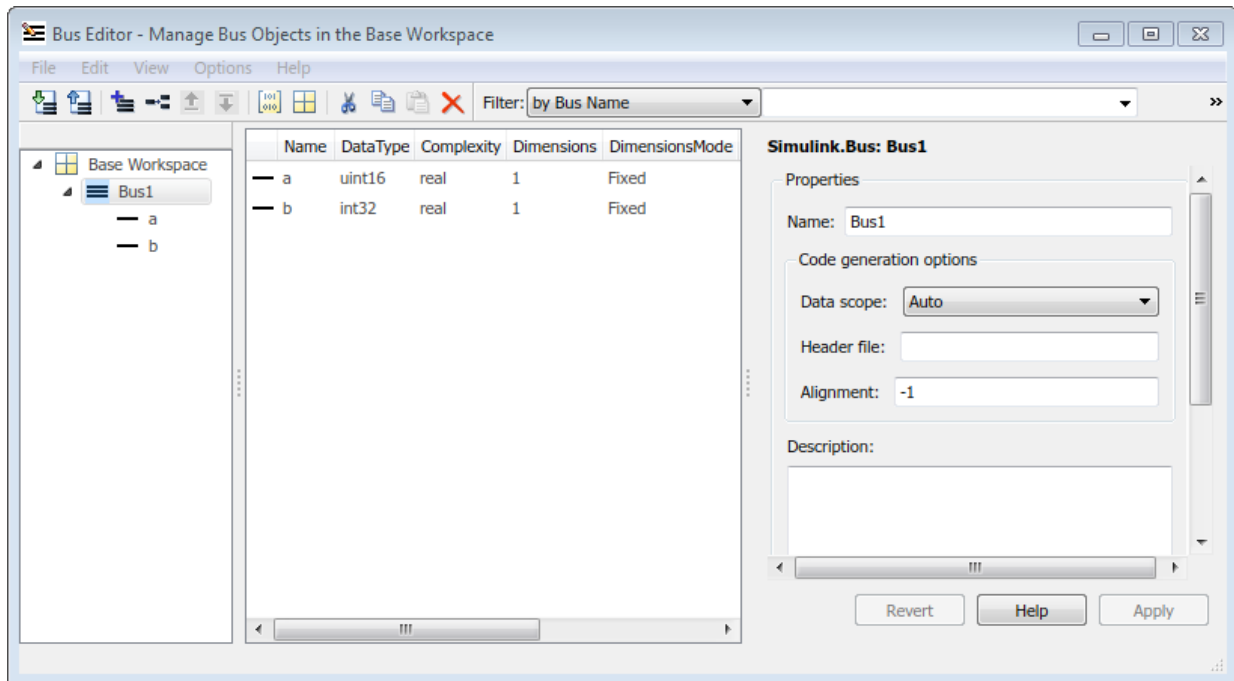
How HDL Coder Generates Code for Array of Buses

HDL Coder expands the array of buses in your Simulink model into the corresponding scalar signals in the generated code.

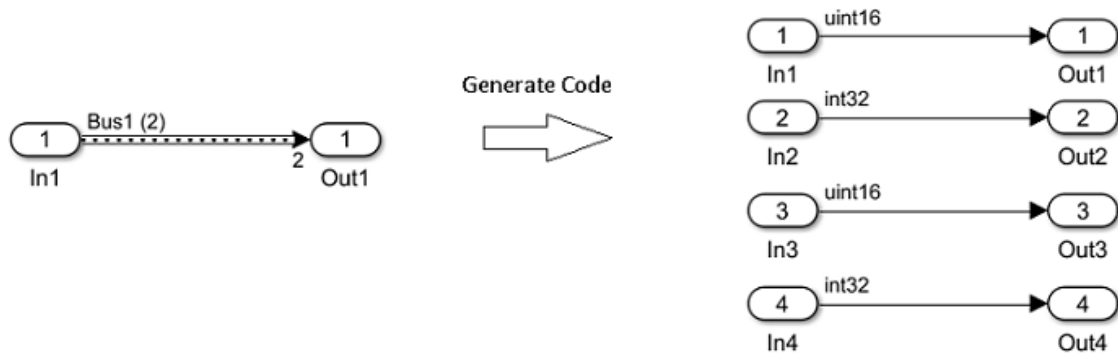
This Simulink model has an array of buses signal at the DUT interface.



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 DOWNT0 0); -- uint16
          In1_1_b : IN    std_logic_vector(31 DOWNT0 0); -- int32
          In1_2_a : IN    std_logic_vector(15 DOWNT0 0); -- uint16
          In1_2_b : IN    std_logic_vector(31 DOWNT0 0); -- int32
          Out1_1_a : OUT   std_logic_vector(15 DOWNT0 0); -- uint16
          Out1_1_b : OUT   std_logic_vector(31 DOWNT0 0); -- int32
          Out1_2_a : OUT   std_logic_vector(15 DOWNT0 0); -- uint16
          Out1_2_b : OUT   std_logic_vector(31 DOWNT0 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 10-2
- “Signal Types” (Simulink)
- “About Data Types in Simulink” (Simulink)
- “Composite (Bus) Signals” (Simulink)
- “Use Enumerated Data in Simulink Models” (Simulink)
- “Enumerated Data” (Stateflow)

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.

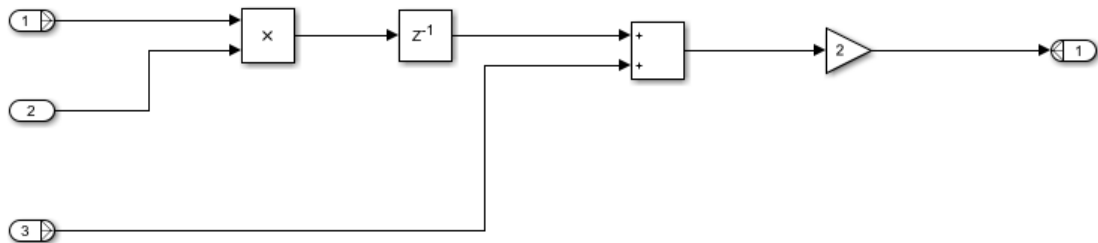
Why Use a For Each Subsystem?

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 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

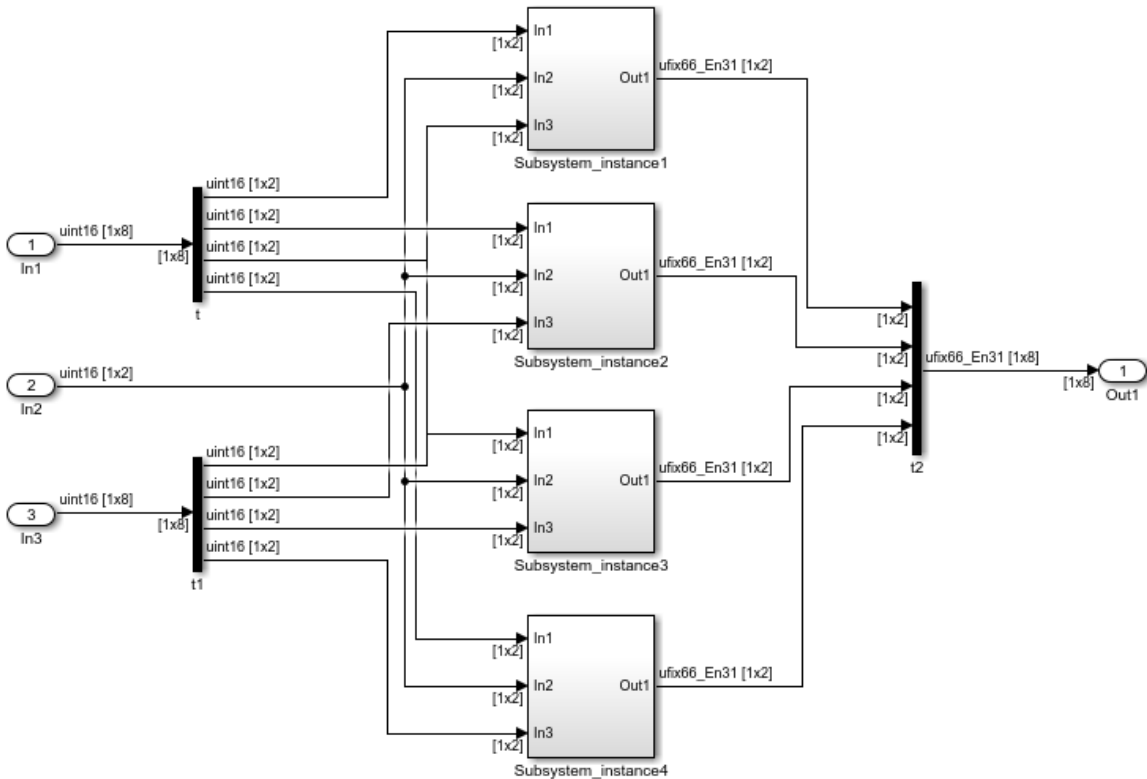
Open the `foreach_subsystem_example1` model. You see this simple algorithm modeled inside a For Each Subsystem block.



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 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;
```

Certain optimizations that you specify can 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.

- Certain optimizations enabled on the subsystem, such as resource sharing and streaming.
- 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.

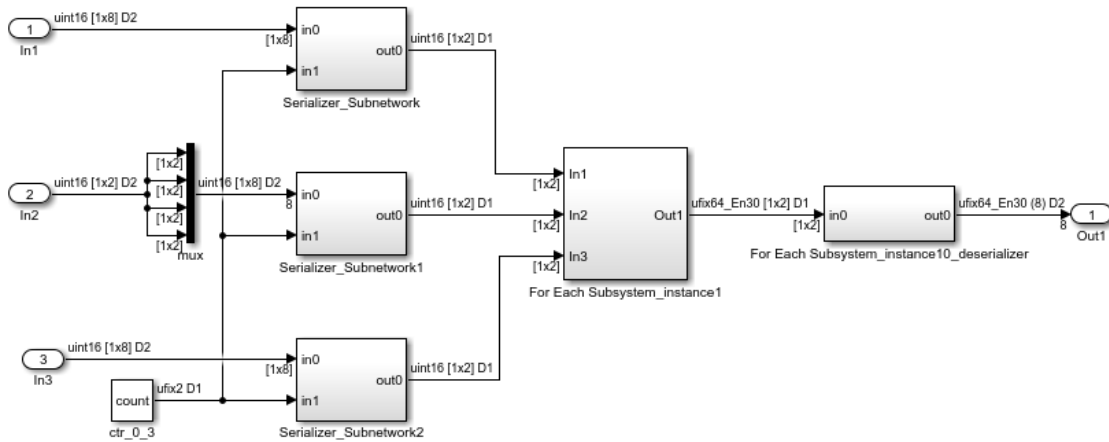
Optimize the For Each Subsystem Algorithm

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.

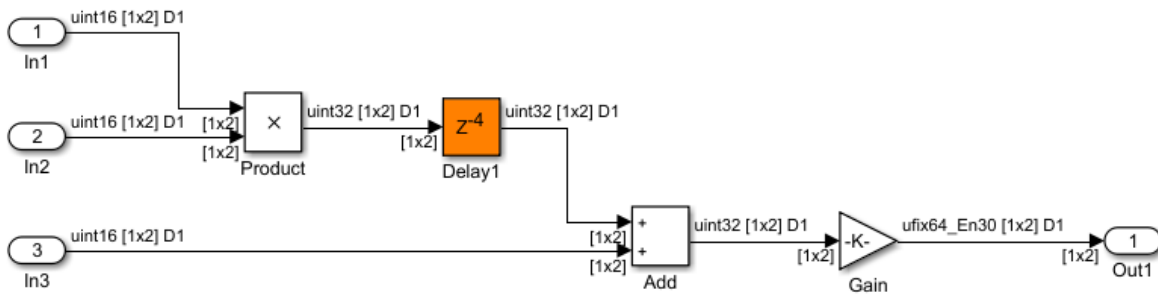
Note: When you enable optimizations on the For Each Subsystem, the generated HDL code does not contain for-generate loops.

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.

See Also

For Each Subsystem

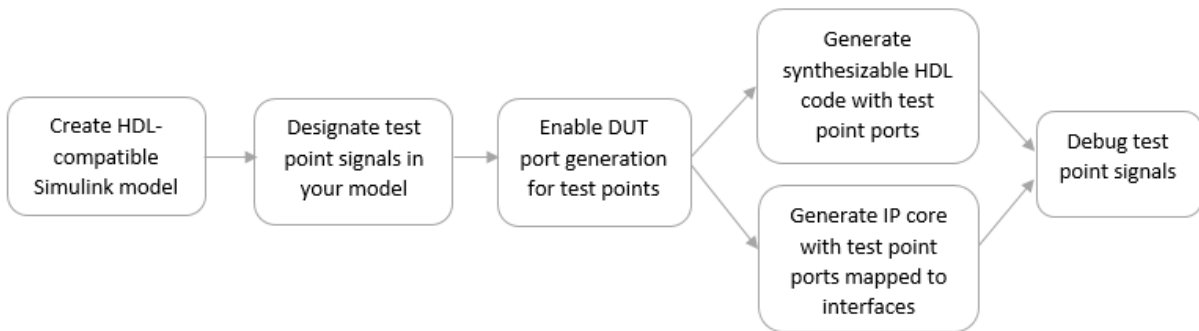
Model and Debug Test Point Signals with HDL Coder™

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.

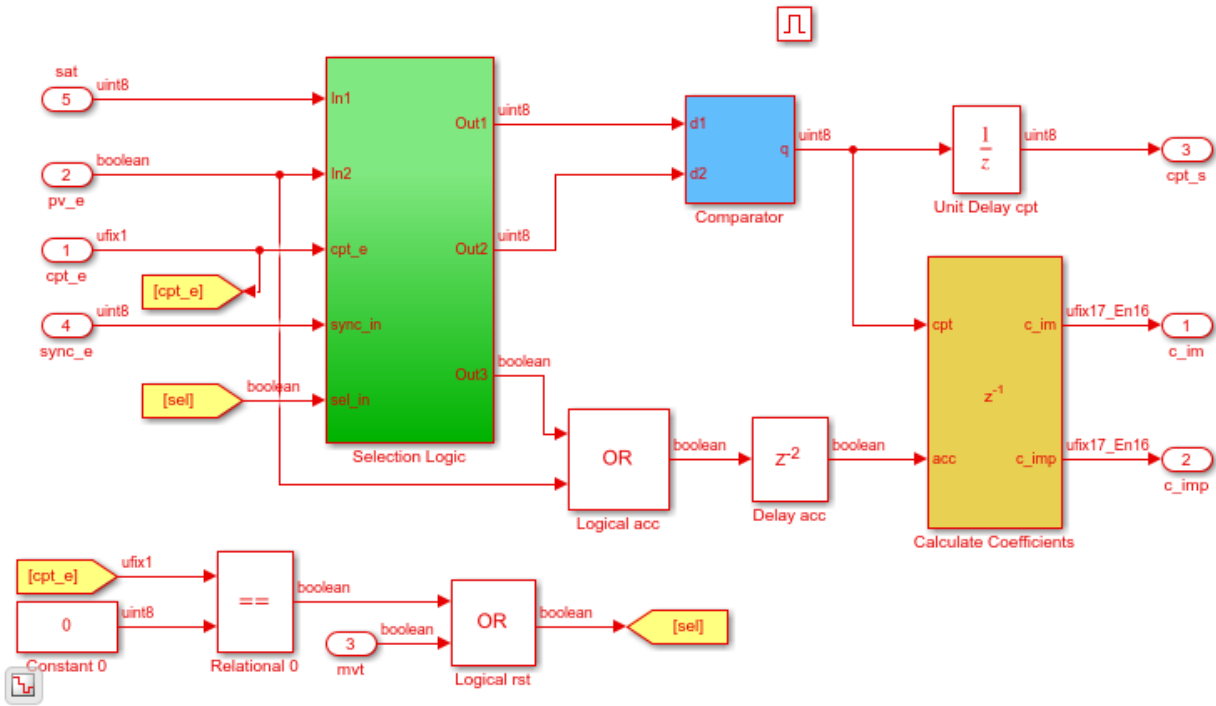


Create HDL-Compatible Model

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_simulink_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 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 Port Generation for Test Points.

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](#).

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_imp	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 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 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
);

```

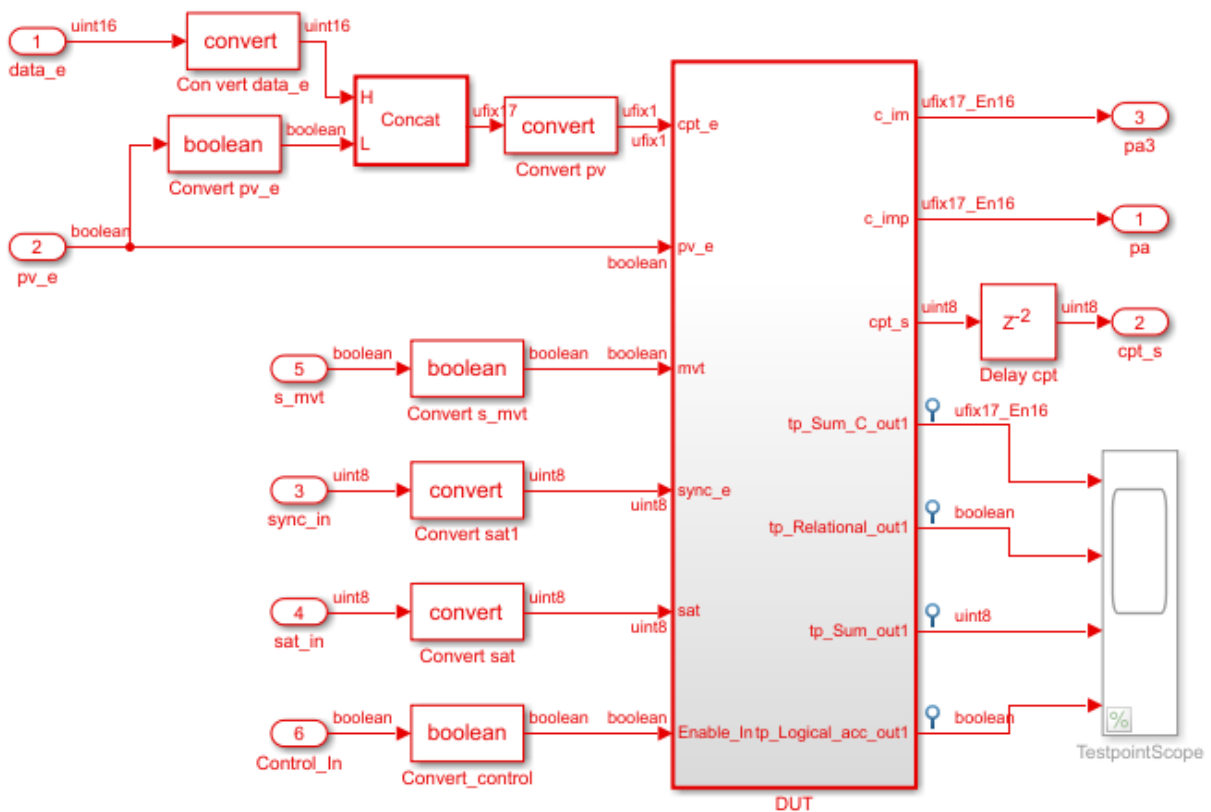
Run the workflow to generate the Software Interface model and integrate the IP core into the target reference design that you specified in the **Set Target Reference Design** task.

To learn more about the IP Core Generation workflow, see Custom IP Core Generation and Custom IP Core Report.

Debug Test Point Signals

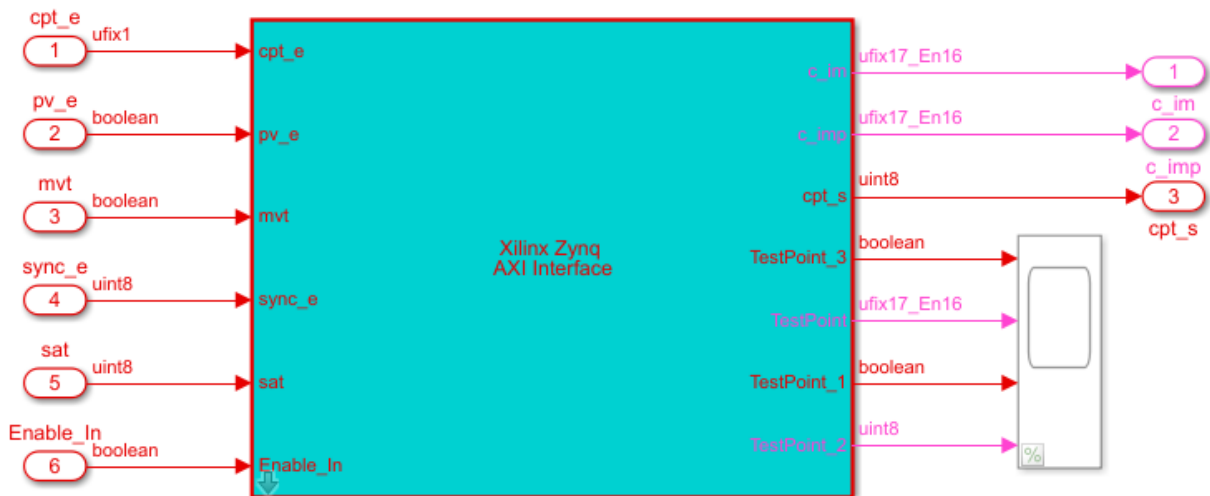
After you generate HDL code or generate an IP core, you can debug the test point signals.

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 Optimizations.
- 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.

- 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.
- 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 can 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.

See Also

More About

- “Test Points” (Simulink)
- “Enable HDL DUT port generation for test points” on page 16-51
- “Generated Model and Validation Model” on page 24-5

Allocate Sufficient Delays for Floating-Point Operations

In this section...

“Problem” on page 10-49

“Cause” on page 10-49

“Solution” on page 10-50

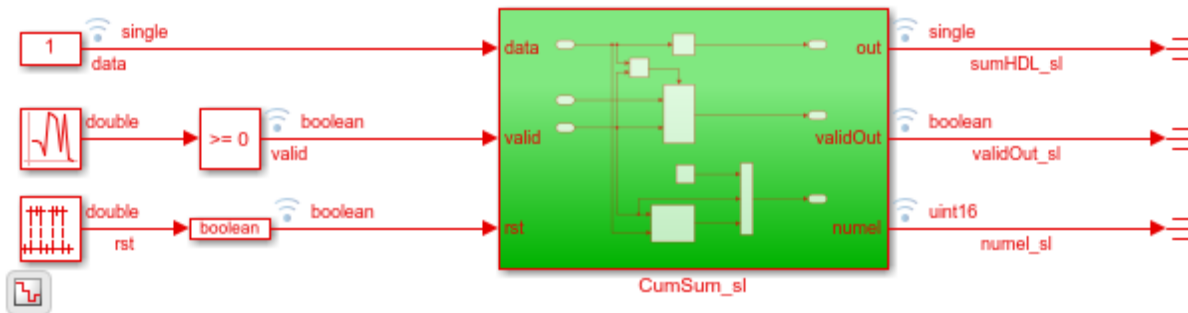
Problem

Sometimes, when generating code from your floating-point algorithm in Simulink, HDL Coder generates an error that it is unable to allocate sufficient number of delays.

Cause

This error message generally occurs when you have Simulink™ blocks performing floating-point operations inside a feedback loop. These blocks have a latency. HDL Coder™ is unable to allocate delays to compensate for the latency, because the code generator needs to add delays and balance them to maintain numerical accuracy.

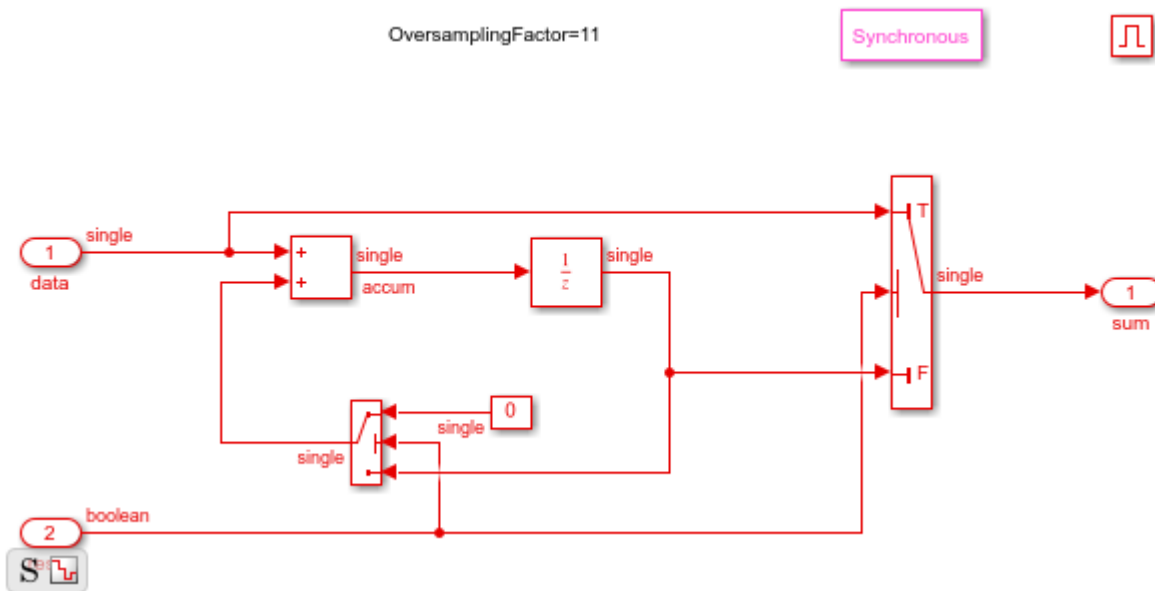
If you open this example model `RunningSum.slx`, you see a Simulink™ model that uses single data types.



To generate HDL code for the `CumSum_sl` Subsystem, right-click the subsystem and select **HDL Code > Generate HDL for Subsystem**. During code generation, HDL Coder™ generates an error:

Unable to allocate delays to compensate for the 11 delays introduced by Add in native floating-point mode. Consider either increasing the oversampling factor, setting the 'Latency Strategy' to 'Zero', or adding the necessary output pipelines via HDL block properties for other blocks in the model to accommodate for the latency introduced by this block.

By using the path to the block mentioned in the error message, navigate to the Add block in the model. This block is inside a feedback loop.



The Add block has a latency of 11. When generating code, HDL Coder™ cannot allocate 11 delays for the block, because it cannot add matching delays to other paths.

This model serves as an example to illustrate the various strategies to solve this problem.

Solution

Strategy 1: Global Oversampling

This modeling paradigm uses the clock-rate pipelining optimization to oversample your design to a clock-rate much faster than the DUT sample rate. To enable this optimization, specify a global oversampling factor for your Simulink model. The floating-point delays

then operate at the faster clock-rate and can be allocated successfully. For more information, see “Clock-Rate Pipelining” on page 24-63.

- 1 Specify an oversampling factor that is equal to or greater than the latency of the floating-point operators that are unable to allocate delays. For the `RunningSum` model, specify an oversampling factor at least equal to 12. To learn about the latency values of the floating-point operators, see “Simulink Blocks Supported with Native Floating-Point” on page 10-106.

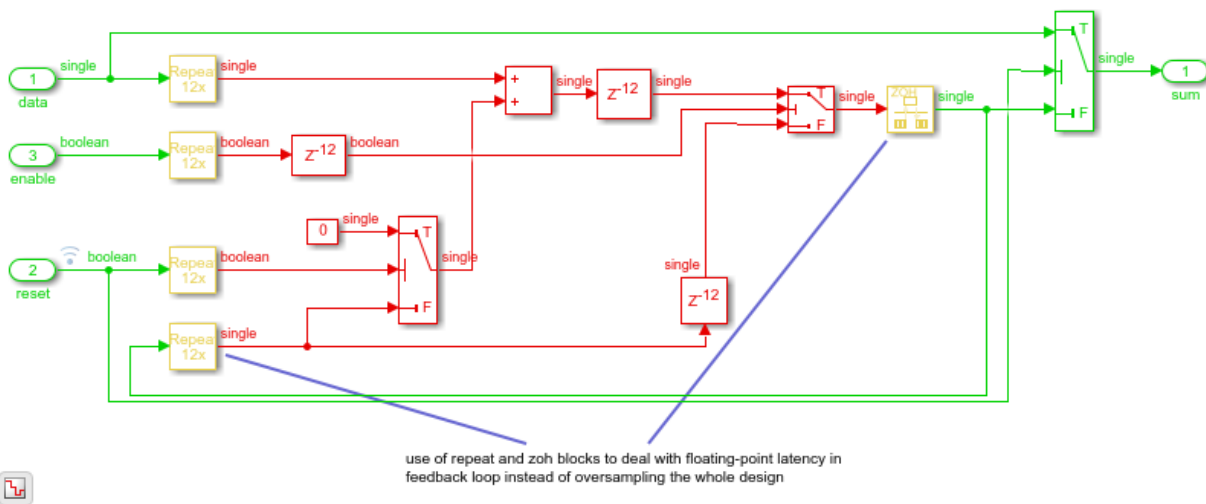
To specify the oversampling factor:

- a In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
 - b Click **Settings**. In the **HDL Code Generation > Global Settings** tab, set **Oversampling factor** to 12.
- 2 Enable hierarchy flattening on the DUT and make sure that subsystems inside the DUT inherit this setting. For the `RunningSum` model, select the `CumSum_sl` subsystem and click **HDL Block Properties** on the **HDL Code** tab, and then set **FlattenHierarchy** to on.

Strategy 2: Local Oversampling

To model your design at the data rate and selectively increase the sample rate of blocks for which HDL Coder™ is unable to allocate delays, use local oversampling. These blocks then operate at the faster clock rate and can accommodate the required number of delays.

If you open the `RunningSum_0Smanual` model and navigate to the Add block, it shows how you can increase the sample rate of the Add block and allocate delays.



- The blocks that are within the boundary of the Repeat and Zero Order Hold blocks operate at the clock rate that is 12 times faster than the sample rate of the model.
- The subsystem has a Delay block of length 12 at the output of the Add block. When generating code, the Add block absorbs this Delay block, which compensates for the latency of the operator. To balance delays, the subsystem contains Delay blocks of length 12 in other paths.

You can now generate HDL code for the CumSum_s1 subsystem. To generate HDL code for the CumSum_s1 Subsystem, right-click the subsystem and select **HDL Code** > **Generate HDL for Subsystem**.

Strategy 3: Delay Blocks

Use this modeling paradigm to model your entire design at the Simulink data rate. For blocks that are unable to accommodate the required number of delays, add a Delay block with a sufficient **Delay length** at the output of the blocks. Specify a **Delay length** that is equal to the latency of the floating-point operator. Make sure that you add matching delays in other paths.

For the RunningSum model, you can add a Delay block of length 12 at the output of the Add block. When generating code, the Add block absorbs this delay, because the block has a latency of 12.

For more information, see “Latency Considerations with Native Floating Point” on page 10-83.

Strategy 4: Use Custom Latency

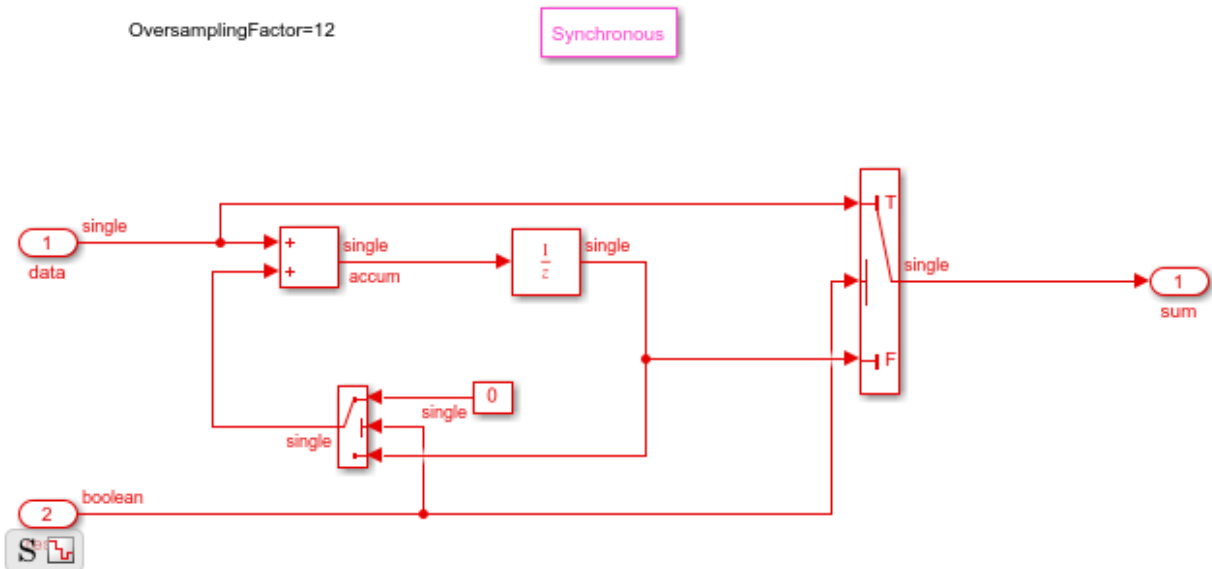
You can use the **Latency Strategy** for various blocks to specify a custom latency value and absorb the additional delays. Using this strategy can optimize your design for trade-offs between:

- Clock frequency and power consumption.
- Oversampling factor and sampling frequency.

To learn more about the trade-offs and blocks for which you can specify a custom latency, see LatencyStrategy.

Note: When you use the custom latency strategy, make sure that the Subsystem that contains the block for which native floating-point is unable to allocate delays is not a conditional subsystem. That is, the Subsystem must not contain trigger, reset, or enable ports.

To see how using a custom latency can resolve the delay allocation issue, open the model RunningSum_Custom.slx.



The model is similar to the original `RunningSum` model but does not have the `Enable` block. Using the `Enable` block can prevent the custom latency strategy from absorbing delays. Specify a custom latency of one for the `Add` block. The `Add` block can then absorb the `Unit Delay` adjacent to the `Add` block.

You can now generate HDL code for the `CumSum_sl` subsystem.

Strategy 5: Zero Latency

You can use the zero latency strategy setting for blocks in your design for which native floating point is unable to allocate delays. By default, blocks in your design inherit the native floating-point settings that you specify in the Configuration Parameters dialog box. To specify a zero latency strategy setting for a block, in the HDL Block Properties dialog box for that block, on the **Native Floating Point** tab, set **LatencyStrategy** to `Zero`.

For the `RunningSum` example, set the **LatencyStrategy** of the `Add` block to `Zero`. To choose the native floating point library and specify zero latency strategy, at the command line, enter:

```
fc = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');  
hdlset_param('RunningSum/CumSum_sl/Subsystem/Add','LatencyStrategy','Zero');
```

Note To obtain good performance on the target FPGA device, it is not recommended to set **Latency Strategy** to `Zero` from the Configuration Parameters dialog box.

See Also

FPToleranceStrategy | FPToleranceValue |
`hdlcoder.createFloatingPointTargetConfig`

Related Examples

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92

Optimize Generated HDL Code for Multirate Designs with Large Rate Differentials

In this section...

“Issue” on page 10-55

“Description” on page 10-55

“Recommendations” on page 10-58

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:

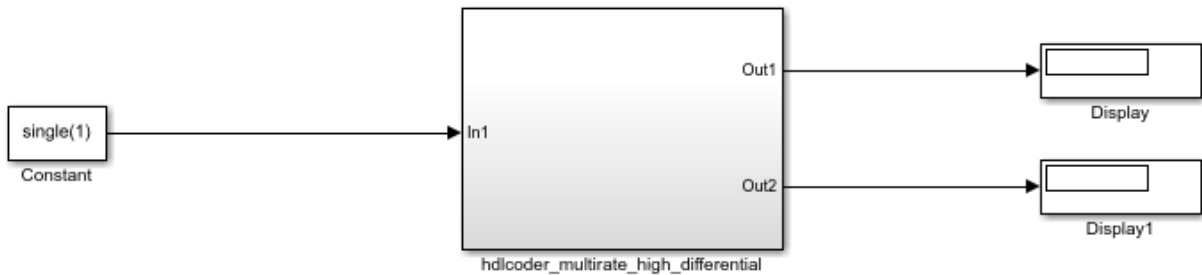
- Multicycle block implementations
- Input and output pipelining
- Distributed pipelining
- Floating-point library mapping
- Native floating-point HDL code generation
- Fixed-point math functions such as reciprocal, sqrt, or divide
- Resource sharing
- Streaming

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

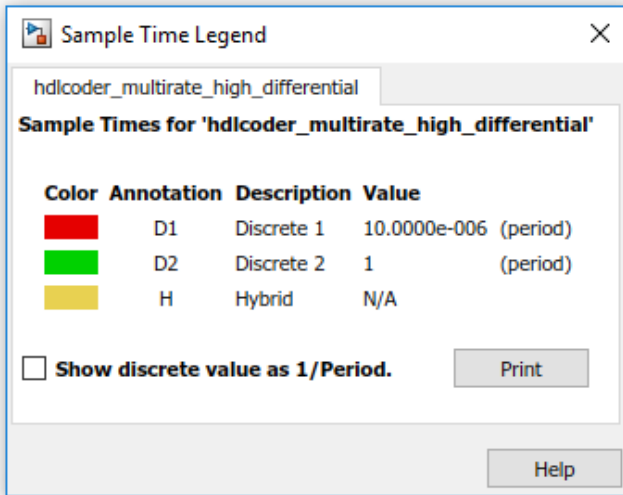
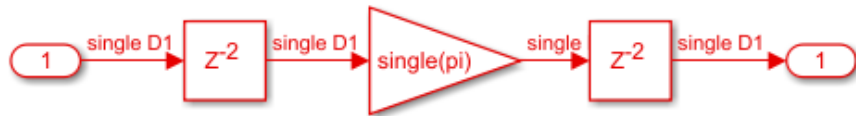
pipeline registers can increase the size of the generated HDL files, and can prevent the design from fitting into an FPGA.

To see an example of how this issue occurs, open this Simulink™ model.

```
open_system('hdlcoder_multirate_high_differential')
```



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.

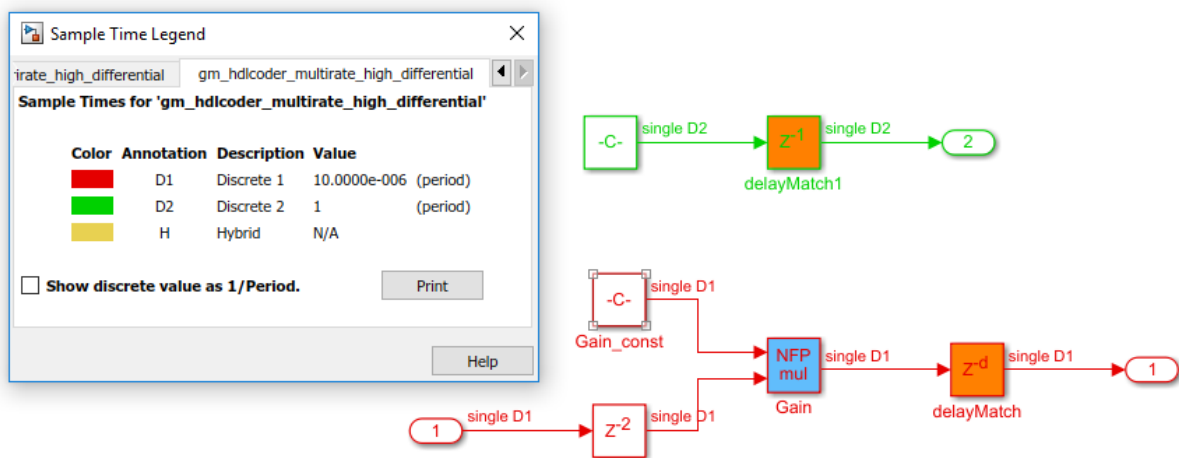


Generate HDL code for the hdlcoder_multirate_high_differential Subsystem and check the output log.

```

### 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 as hdlsrc\hdlcoder_multirate_high_differential\hdlcoder_multirate_high_differential.bc.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_pky.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.

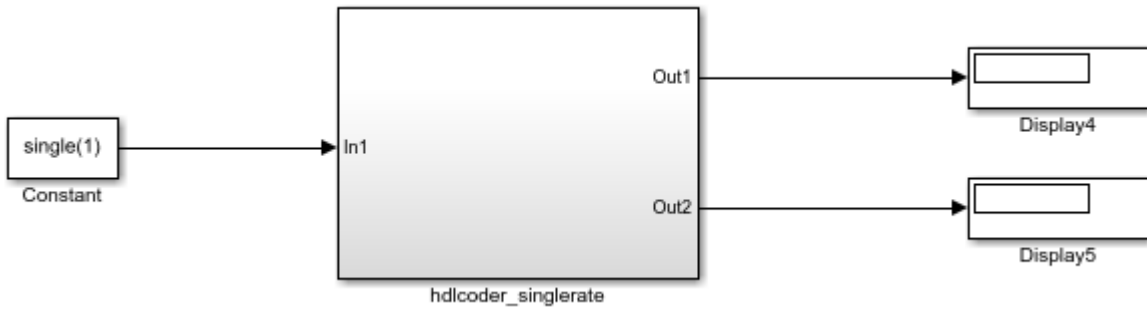
Recommendations

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.



Sample Time Legend

hdlcoder_singlerate

Sample Times for 'hdlcoder_singlerate'

Color	Annotation	Description	Value
	FiM	Fixed in Minor Step	[0,1]
	D1	Discrete 1	10.0000e-006 (period)

Show discrete value as 1/Period. Print

Help



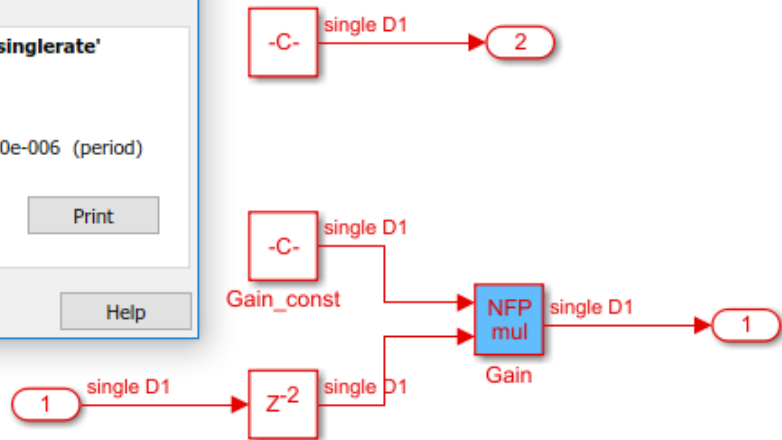
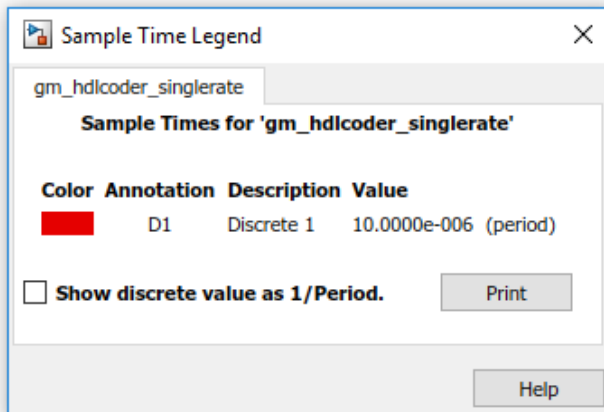
Generate HDL code for the hdlcoder_singlerate Subsystem and check the output log.

```

### 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.

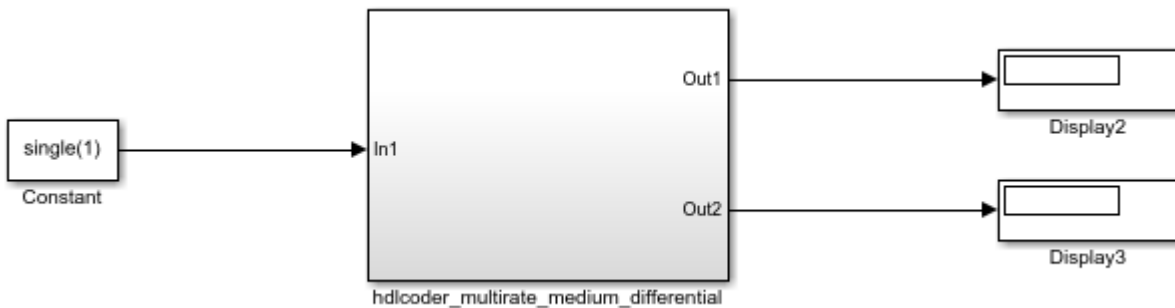
Recommendation 2: Reduce the Rate Differential

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.

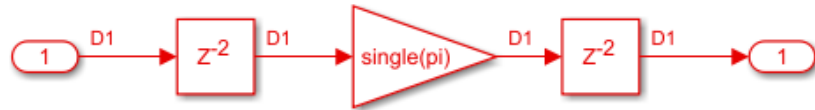
In this example, you can change the sample rate of the Constant block inside the `hdlcoder_multirate_high_differential` Subsystem to reduce the rate differential.

Open this model that has the sample time of the Constant block changed to 0.01.

```
open_system ('hdlcoder_multirate_medium_differential')
```



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.



Sample Time Legend

hdlcoder_multirate_medium_differential

Sample Times for 'hdlcoder_multirate_medium_differential'

Color	Annotation	Description	Value
	FIM	Fixed in Minor Step	[0,1]
	D1	Discrete 1	10.0000e-006 (period)
	D2	Discrete 2	0.01 (period)
	H	Hybrid	N/A

Show discrete value as 1/Period. Print

Help

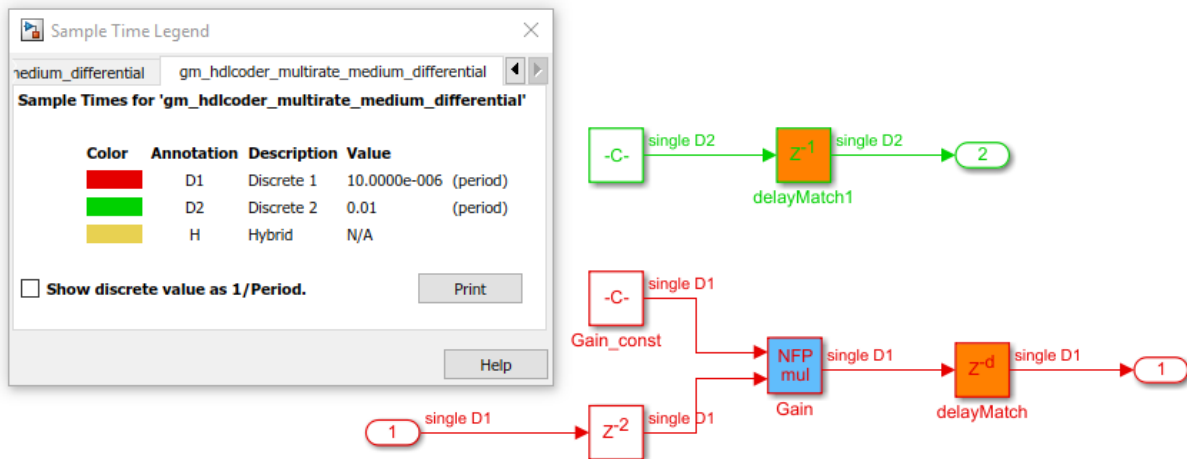


Generate HDL code for the hdlcoder_multirate_medium_differential Subsystem and check the output log.

```

### 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 as hdlsrc\hdlcoder_multirate_medium_differential\hdlcoder_multirate_medium_differential.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

`hdlcoder.createFloatingPointTargetConfig`

Related Examples

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92

Getting Started with HDL Coder Native Floating-Point Support

In this section...

“Numeric Considerations and IEEE-754 Standard Compliance” on page 10-65

“Data Type Considerations” on page 10-66

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.

In your Simulink model:

- You can have 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 single-precision 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.

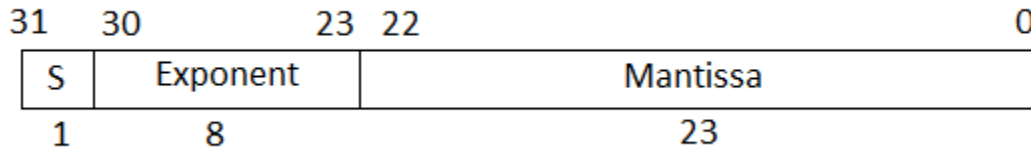
HDL Coder supports:

- 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

HDL Coder generates code in compliance with the IEEE 754-2008 standard of floating-point arithmetic.

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.

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 with Native Floating-Point” on page 10-69.

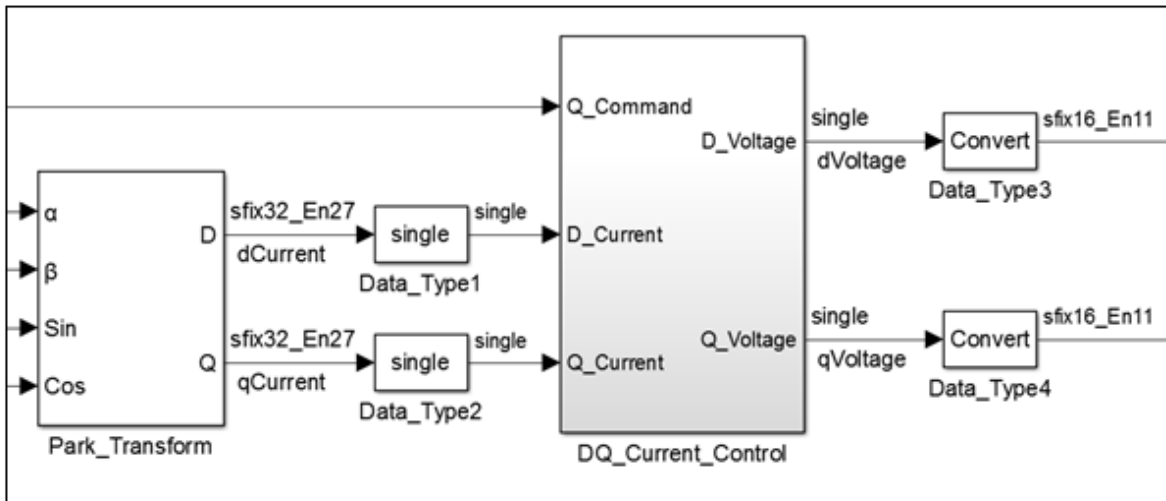
Data Type Considerations

With native floating-point support, HDL Coder supports code generation from Simulink models that contain floating-point signals and fixed-point signals. You might want to 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, 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 20-79

Functions

`hdlcoder.createFloatingPointTargetConfig`

Properties

`FPToleranceStrategy` | `FPToleranceValue`

Related Examples

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92
- “Simulink Blocks Supported with Native Floating-Point” on page 10-106

Numeric Considerations with Native Floating-Point

In this section...
“Round to Nearest Rounding Mode” on page 10-69
“Denormal Numbers” on page 10-70
“Exception Handling” on page 10-70
“Relative Accuracy and ULP Considerations” on page 10-71
“ULP of Native Floating-Point Operators” on page 10-73
“Considerations” on page 10-74

Native floating-point technology can generate HDL code from your floating-point design. 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 generates code that complies with the IEEE-754 standard of floating-point arithmetic. HDL Coder native floating-point 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

Round to Nearest Rounding Mode

HDL Coder native floating-point uses the round to nearest even rounding mode. 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.

Rounding bits	Rounding Action
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 loss of significant digits that can accumulate over subsequent operations and eventually result in unexpected values.

The logic 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 using native floating-point support, you can specify whether you want HDL Coder to handle denormal numbers in your design. By default, the code generator does not check for denormal numbers, which saves area on the target platform.

Exception Handling

If you perform operations such as division by zero or compute the logarithm of a negative number, HDL Coder detects and reports exceptions. The table summarizes the mapping from the encoding of a floating-point number to the value of the number for various kinds of exceptions. x denotes a *don't care* value and can take either a 0 or a 1.

Sign	Exponent	Significand	Value	Description
x	0xFF	0x00000000	$value = (-1)^S \infty$	Infinity

Sign	Exponent	Significand	Value	Description
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 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.

ULP

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 you correctly round the result of an elementary arithmetic operation such as addition, multiplication, and division. A correctly rounded result means that the rounded result is within 0.5 ULP of the exact result.

An ULP of one means adding a 1 to the decimal value of the number. The table shows the approximation of pi to nine decimal digits and how the ULP of one changes the approximate value.

Floating-point number	Value in decimal	IEEE-754 representation for Single Types	ULP
3.141592741	1078530011	0 10000000 10010010000111111011011	0

Floating-point number	Value in decimal	IEEE-754 representation for Single Types	ULP
3.141592979	1078530012	0 10000000 10010010000111111011100	1

The gap between two consecutively representable floating-point numbers varies according to magnitude.

Floating-point number	Value in decimal	IEEE-754 representation for Single Types	ULP
1234567	1234613304	0 10010011 001011101011010000111000	0
1234567.125	1234613305	0 10010011 001011101011010000111001	1

Relative Error

Relative error measures the difference between a floating-point number and the approximation of the real number. Relative error returns the distance from 1.0 to the next larger number. This table shows how the real value of a number changes with the relative accuracy.

Floating-point number	Value in decimal	IEEE-754 representation for Single Types	ULP	Relative error
8388608	1258291200	0 10010110 000000000000000000000000	0	1
8388607	1258291198	0 10010101 111111111111111111111110	1	2.3841858e-07
1	1065353216	0 01111111 000000000000000000000000	0	1.1920929e-07
2	1073741824	0 10000000 000000000000000000000000	1	2.3841858e-07

The magnitude of the relative error depends on the real value of the floating-point number.

In MATLAB, the `eps` function measures the relative accuracy of the floating-point number. For more information, see `eps`.

ULP of Native Floating-Point Operators

Native floating point support in HDL Coder follows IEEE standard of floating-point arithmetic.

All 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.

All 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 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 `FPToleranceStrategy` and `FPToleranceValue`.

Most basic math operators and some advanced math operations have a zero ULP in native floating-point mode. Native floating point support in HDL Coder makes the best attempt to implement these operators to the lowest ULP possible. 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	Units in the Last Place (ULP) error
exp	1
log	1
log10	1
10 ^u	1
pow	1
hypot	1

Trigonometric Functions

Simulink Blocks	Units in the Last Place (ULP) error
sin	2
cos	2
tan	3
asin	2
acos	2
atan	2
atan2	5
sinh	1
cosh	1
tanh	1
asinh	2
acosh	2
atanh	3
sincos	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 `Inf i` whereas the HDL simulation result is `NaN +Inf i`.
- 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 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 20-79

Functions

`hdlcoder.createFloatingPointTargetConfig`

Properties

`FPToleranceStrategy` | `FPToleranceValue`

Related Examples

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92

Latency Values of Floating Point Operators

In this section...

“Math Operations” on page 10-77

“Trigonometric and Exponential Operations” on page 10-79

“Comparisons and Conversions” on page 10-81

HDL Coder native floating-point support can generate HDL code from your floating-point design. HDL Coder supports several Simulink blocks and math and trigonometric functions in native floating-point mode. These tables show the default latency values of these floating-point operations. You can customize these latency values. You can also customize the latency settings for most blocks and design for trade-offs between latency and Fmax by specifying custom latency values. To learn more, see “Latency Considerations with Native Floating Point” on page 10-83.

You can see the latency of these floating point operators in MATLAB by entering these commands.

```
nfpcnfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');  
nfpcnfig.IPCnfig
```

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	Data Type	Minimum Output Latency	Maximum Output Latency
Add	Double	6	11
	Single	6	11
Subtract	Double	6	11
	Single	6	11
Product	Double	6	9
	Single	6	8
Divide	Double	31	61
	Single	17	32
Math Function	Double	30	60
	Single	16	31
Multiply-Add	Single	8	14
Rounding Function	Single	5	5
Unary Minus	Double	-	-
	Single	-	-
Sign	Single	-	-
Abs	Double	-	-
	Single	-	-

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. You can use these blocks with single data types. Double types are unsupported for the blocks.

Math Functions

Simulink Blocks	Minimum Output Latency	Maximum Output Latency
HDL Reciprocal	14	21
Rem	15	24
Mod	16	26
Sqrt	16	28
Reciprocal Sqrt	16	30
Hypot	17	33

Trigonometric and Exponential 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. Double types are unsupported for the blocks.

Trigonometric Functions

Simulink Blocks	Minimum Output Latency	Maximum Output Latency
Sin	27	27
Cos	27	27
Tan	33	33
Sincos	27	27
Asin	17	23
Acos	17	23
Atan	36	36
Atan2	42	42
Sinh	18	30
Cosh	17	27
Tanh	25	43
Asinh	94	94
Acosh	93	93
Atanh	67	67

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.

Exponent/Logarithm/Power

Simulink Blocks	Minimum Output Latency	Maximum Output Latency
Exp	16	26
Pow	33	54
Pow10	16	26
Log	20	27
Log10	17	27

Comparisons and Conversions

This table shows operations related to comparing of numbers 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 types. For the Data Type Conversion block, you can convert between `double` and `single` data types, and between `single` and other fixed-point data types.

Comparisons and Conversions

Simulink Blocks	Data Type	Minimum Output Latency	Maximum Output Latency
Data Type Conversion	Double	3	6
	Single	6	6
Relational Operator	Double	1	3
	Single	1	3
MinMax	Single	3	3

See Also

Modeling Guidelines

“Modeling with Native Floating Point” on page 20-79

Functions

`hdlcoder.createFloatingPointTargetConfig`

Properties

`FPToleranceStrategy` | `FPToleranceValue`

Related Examples

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65

- “Simulink Blocks Supported with Native Floating-Point” on page 10-106

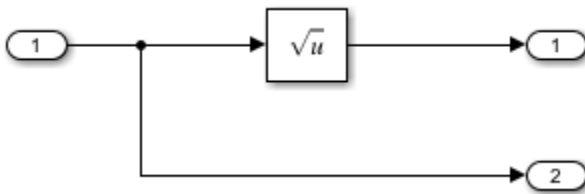
Latency Considerations with Native Floating Point

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

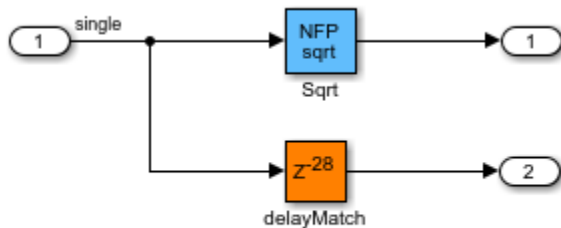
Open the `hdlcoder_nfp_delay_allocation` Simulink™ model. The model uses single data types and computes the square root. The model has a parallel path to illustrate how the code generator balances delays.

```
load_system('hdlcoder_nfp_delay_allocation')  
open_system('hdlcoder_nfp_delay_allocation/DUT')
```

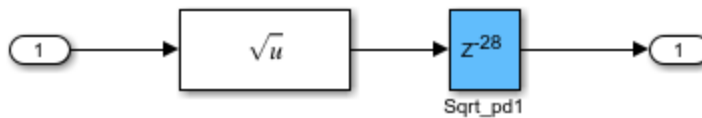


To generate HDL code:

- 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.

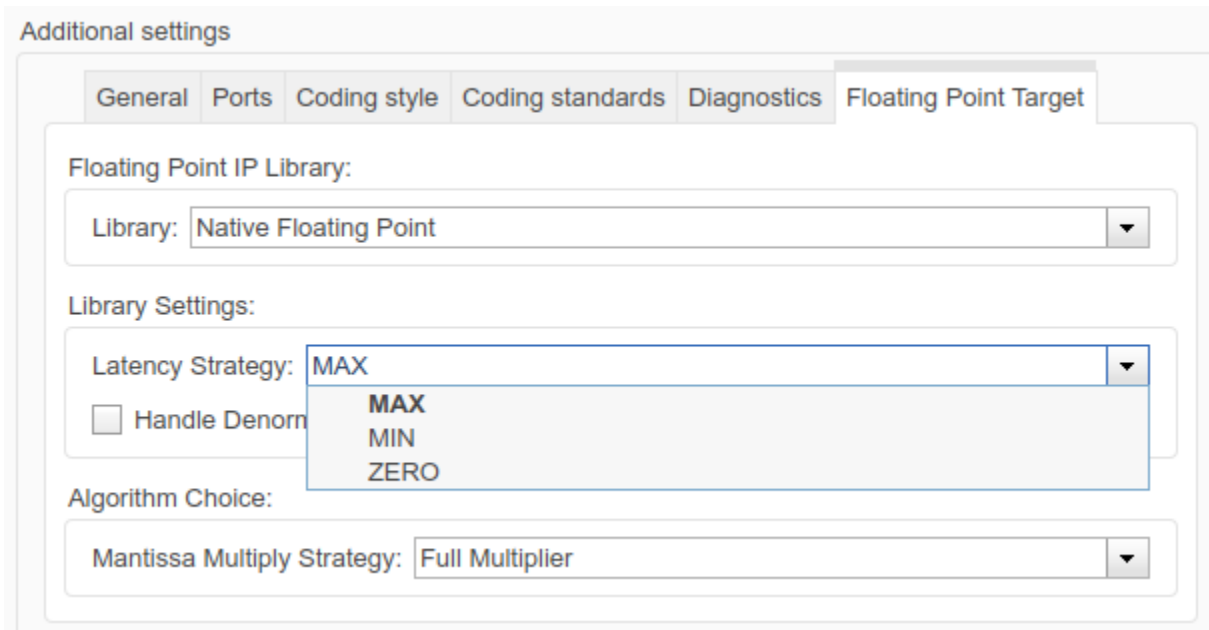
- Delay blocks in the model: If your Simulink model has a latency, HDL Coder™ can absorb some or all of the latency with the native floating-point implementation.

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 > Global Settings > Floating Point Target** tab, for **Library**, select **Native Floating Point**, and then for **Latency Strategy**, select **MAX**, **MIN**, or **ZERO**.



To specify this setting from the command line:

- 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);
```

- 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]
```

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.

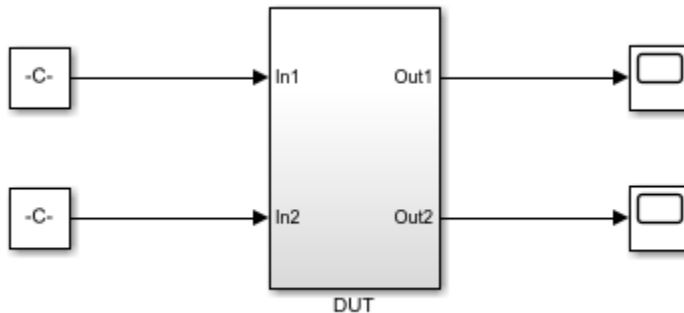
By specifying a custom latency, you can customize your design for trade-offs between:

- 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.

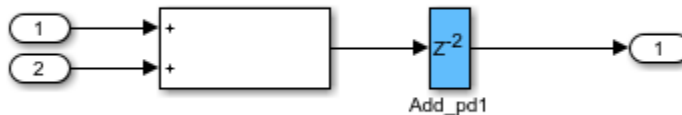
To learn more about this setting and how to specify the latency strategy for a block, see `LatencyStrategy`.

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)
```



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.

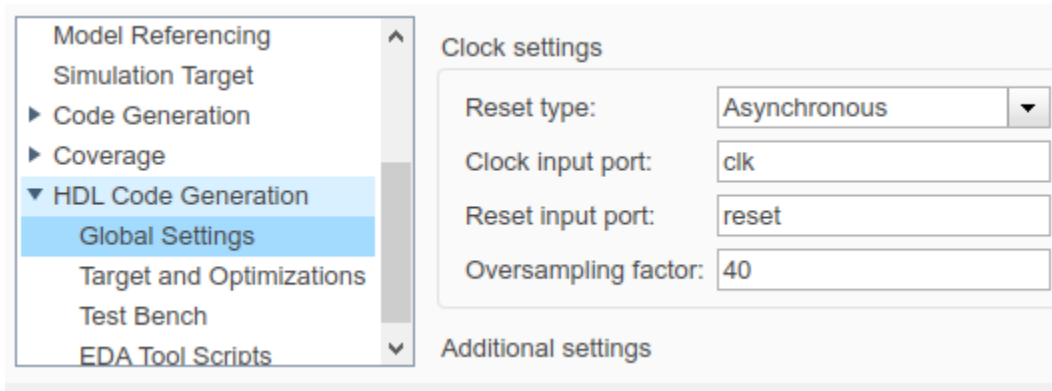


Oversampling Factor

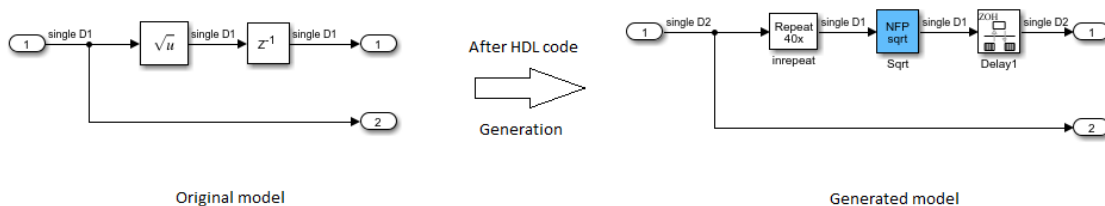
When you design the blocks in your Simulink™ model at the data rate, specify an **Oversampling factor** greater than one. The **Oversampling factor** inserts pipeline registers at a faster clock rate, which improves clock frequency and reduces area usage. To learn more about clock-rate pipelining, see Clock Rate Pipelining.

To see the effect of **Oversampling factor** on the model, in the `hdlcoder_nfp_delay_allocation` model:

- 1 Add a Delay block with **Delay length 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

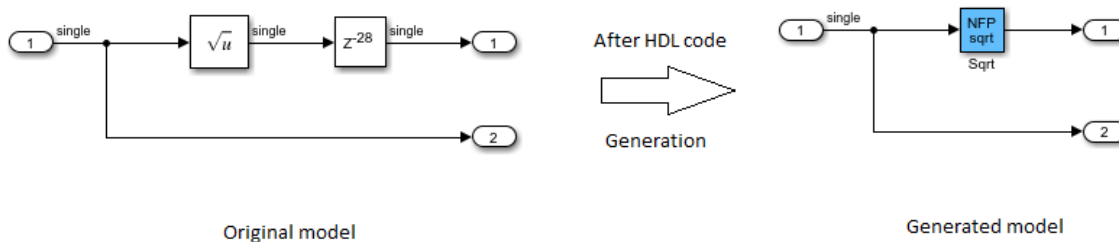
If your Simulink™ model has a Delay block with sufficient **Delay length** adjacent to an operator, HDL Coder™ absorbs the delays as part of the operator latency.

Note: To absorb delays, make sure that you group the delays adjacent to the block.

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.

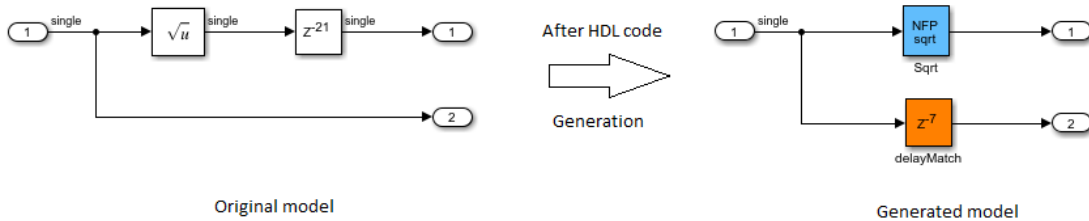


In the generated model, you see that the NFP Sqrt block absorbs the Delay block adjacent to the Sqrt block in your original model. This delay absorption occurs because the operator latency is equal to the **Delay length**. The code generator therefore avoids the additional latency in your model.

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.

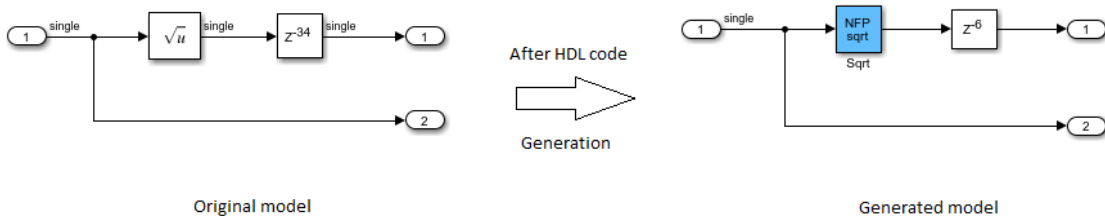


You see that the NFP Sqrt block absorbed a Delay of length 21 and added a matching delay of length 7 in the parallel path because the square root operation requires 28 delays.

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 20-79

Functions

`hdlcoder.createFloatingPointTargetConfig`

Properties

FPToleranceStrategy | FPToleranceValue

Related Examples

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- “Latency Values of Floating Point Operators” on page 10-77
- “Simulink Blocks Supported with Native Floating-Point” on page 10-106

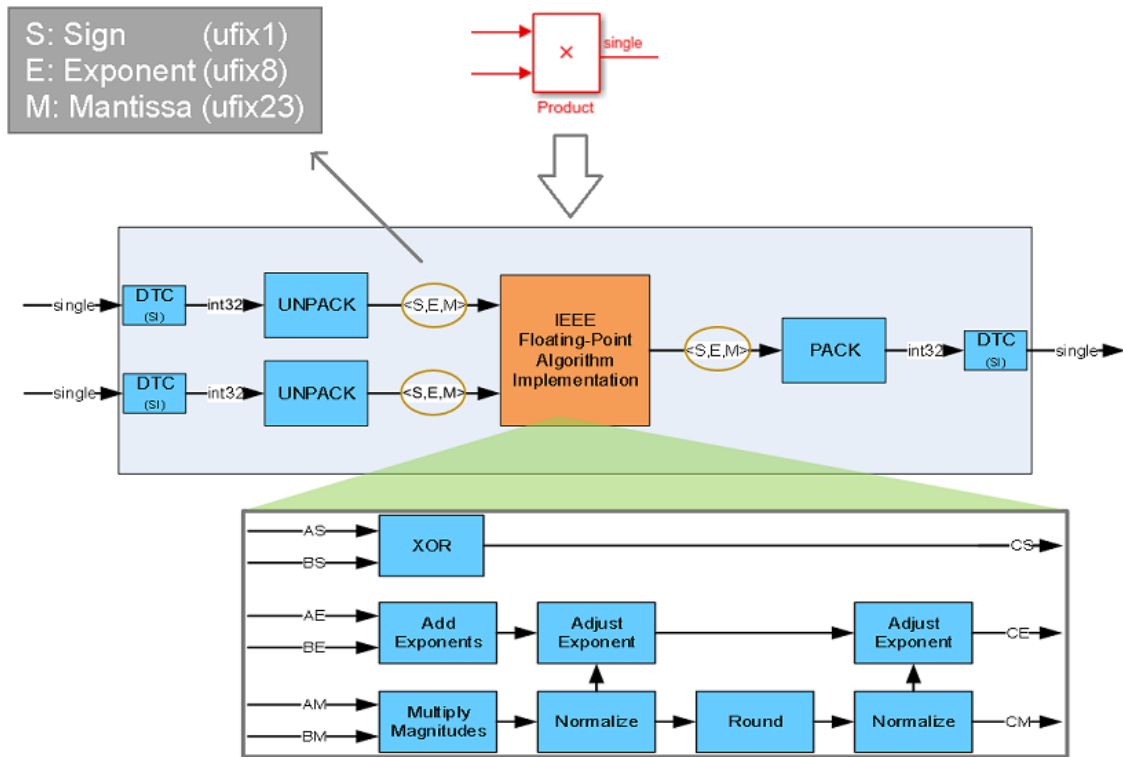
Generate Target-Independent HDL Code with Native Floating-Point

In this section...
“How HDL Coder Generates Target-Independent HDL Code” on page 10-92
“Enable Native Floating Point and Generate Code” on page 10-93
“View Code Generation Report” on page 10-95
“Analyze Results” on page 10-97
“Limitation” on page 10-99

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.

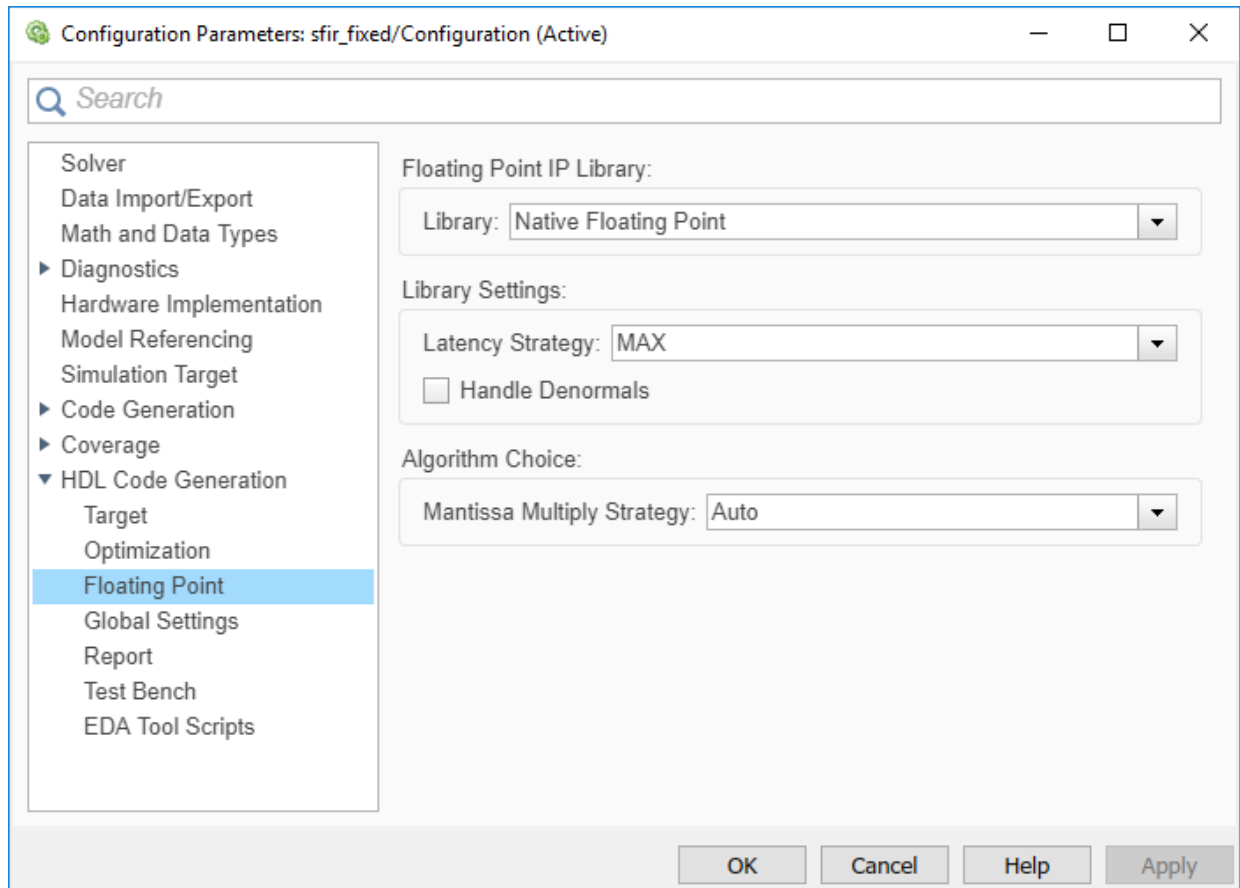
The **Floating-Point Algorithm Implementation** block performs computations on the *S*, *E*, and *M*. With this conversion, the generated HDL code is target-independent. You can deploy the design on any generic FPGA or an ASIC.

Enable Native Floating Point and Generate Code

You can generate code 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, for **Library**, select Native Floating Point.



- 3 Specify the **Latency Strategy** to map your design to maximum or minimum latency or no latency.
- 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” on page 15-6.

- 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” on page 15-7.
- 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 generate HDL code at the command line, 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 10-101.

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**. See also “Create and Use Code Generation Reports” on page 25-2.

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. See also “Create and Use Code Generation Reports” on page 25-2.

Detailed Report

Module nfp_add_comp

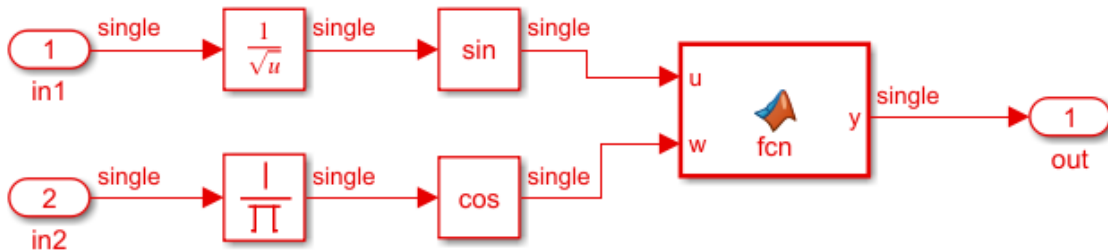
(Latency = "Max")

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.

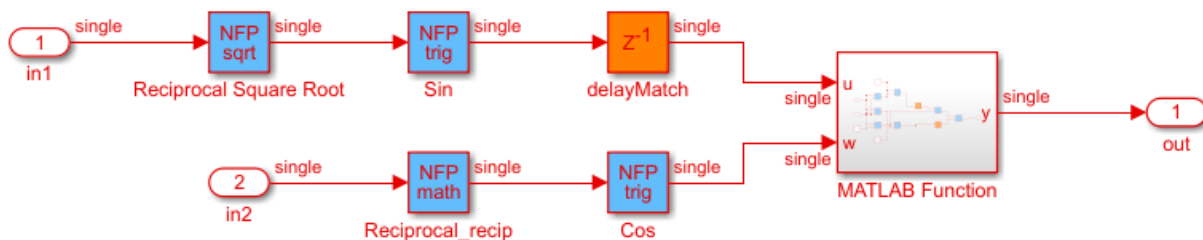


The MATLAB Function block in the Simulink model contains this code.

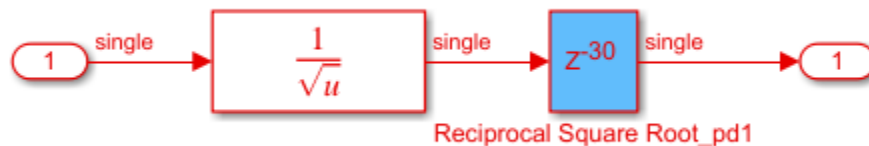
```
function y = fcn(u, w)
%#codegen

y1 = (u+w) * 20;
y2 = w^16;
y3 = (u-w) / 10;
y = y1 + y2 - y3;
```

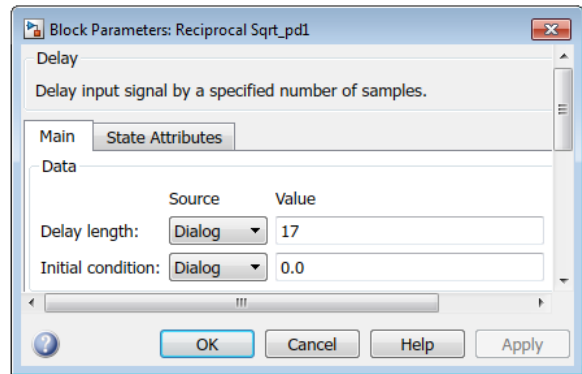
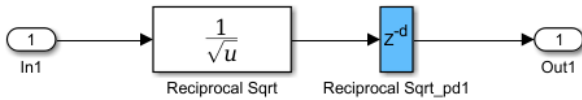
When you generate HDL code, the code generator maps the blocks in your Simulink model to synthesizable native floating-point operators. To see how the code generator implemented the floating-point operations, open the generated model. The blocks **NFP math**, **NFP Sqrt**, and **NFP trig** correspond 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 10-77.



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 24-5.

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 20-79

Functions

`hdlcoder.createFloatingPointTargetConfig`

Properties

`FPToleranceStrategy` | `FPToleranceValue`

Related Examples

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- “Simulink Blocks Supported with Native Floating-Point” on page 10-106

Verify the Generated Code from Native Floating-Point

In this section...

“Specify the Tolerance Strategy” on page 10-101

“Verify the Generated Code with HDL Test Bench” on page 10-102

“Verify the Generated Code with Cosimulation” on page 10-103

“Limitation” on page 10-105

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 10-71.

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`.

Test bench data file name postfix:

Test bench reference postfix:

Use file I/O to read/write test bench data:

Ignore output data checking (number of samples):

Floating point tolerance check based on:

Tolerance Value:

Simulation library path:

- 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 with Native Floating-Point” on page 10-69.

To specify the tolerance strategy at the command-line, use:

- 1 Specify the floating point tolerance check setting by using `FPToleranceStrategy`.

```
hdlset_param('sfir_single', 'FPToleranceStrategy', 'Relative'); % check for floating point error
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP'); % check for floating point error
```

- 2 Based on the `FPToleranceStrategy` setting, enter the tolerance value by using `FPToleranceValue`.

```
hdlset_param('FP_test_16a', 'FPToleranceValue', 1e-06); % if using relative error,
hdlset_param('FP_test_16a', 'FPToleranceValue', 1); % if using ULP error, enter 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**.

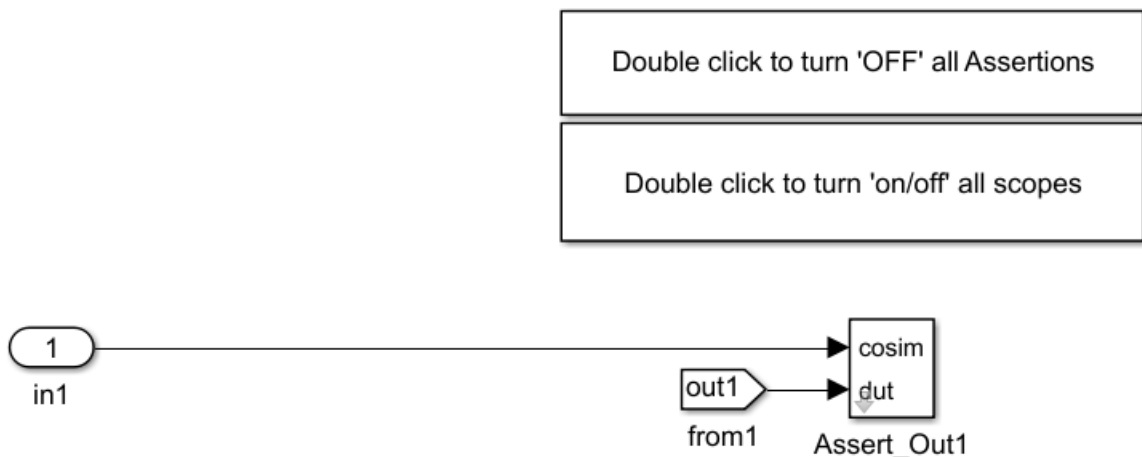
- 2 In the Configuration section, make sure that **Use file I/O to read/write test bench data** is enabled. To generate a test bench that uses constants instead of file I/O, clear **Use file I/O to read/write test bench data**.
- 3 Click **Apply**, and then click **Generate Test Bench**.

To learn more about how HDL test bench generation works, see “Test Bench Generation” on page 6-6.

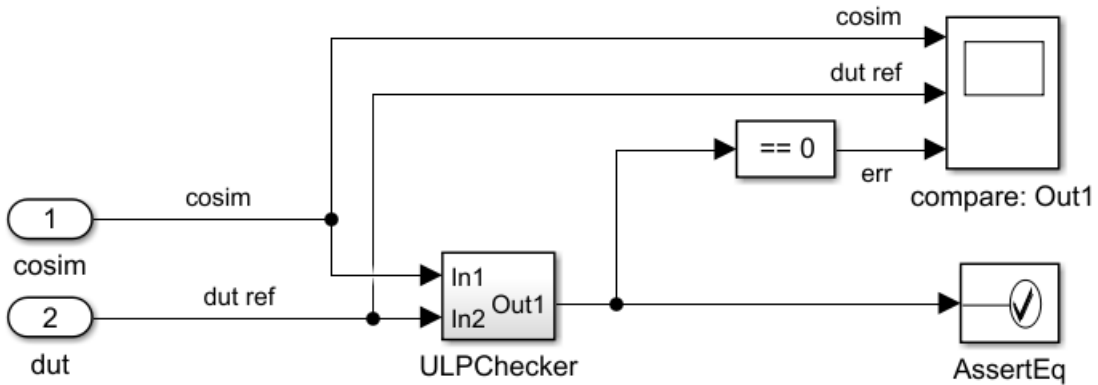
Verify the Generated Code with Cosimulation

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.

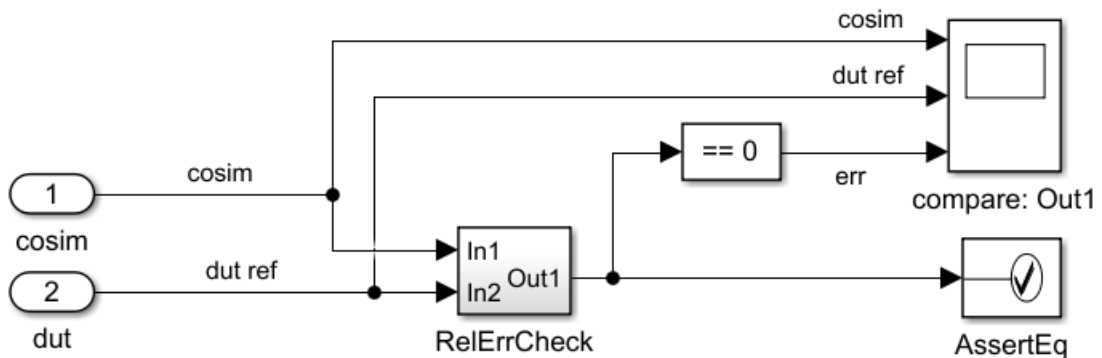


- 4 If you double-click the `Assert_Out1` block, the block parameters show the **ToleranceValue** that you specify.
- 5 To look inside the `Assert_Out1` block, click the mask. If you specify the floating-point tolerance check based on `ulp` error, the model shows a `ULPChecker` block.



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 27-41.

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 20-79

Functions

`hdlcoder.createFloatingPointTargetConfig`

Properties

`FPToleranceStrategy` | `FPToleranceValue`

Related Examples

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- “Numeric Considerations with Native Floating-Point” on page 10-69
- “Simulink Blocks Supported with Native Floating-Point” on page 10-106

Simulink Blocks Supported with Native Floating-Point

In this section...

“HDL Floating Point Operations Library” on page 10-106

“Supported Simulink Blocks in Math Operations Library” on page 10-106

“Supported Simulink Blocks in Other Libraries” on page 10-107

“Simulink Block Restrictions” on page 10-108

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 with native floating-point technology.

HDL Floating Point Operations Library

In the **HDL Floating Point Operations** library, HDL Coder supports all blocks that have single and double data types in the Native Floating Point mode.

Note Certain blocks such as Discrete-Time Integrator and Discrete PID Controller are supported in Native Floating Point mode only when they use zero latency strategy. These blocks contain inherent feedback loops and using a nonvalue for the latency strategy can result in the code generator being unable to allocate delays. For more information, see “Allocate Sufficient Delays for Floating-Point Operations” on page 10-49.

Supported Simulink Blocks in Math Operations Library

In the Math Operations library, these blocks are supported:

Block Name	Supported with single data types	Supported with double data types
Abs	Yes	Yes

Block Name	Supported with single data types	Supported with double data types
Add	Yes	Yes
Assignment	Yes	Yes
Bias	Yes	Yes
Complex to Real-Imag	Yes	Yes
Divide	Yes	Yes
Dot Product	Yes	Yes
Gain	Yes	Yes
Math Function	Yes	No
MinMax	Yes	Yes
Product	Yes	Yes
Product of Elements	Yes	Yes
Real-Imag to Complex	Yes	Yes
Reciprocal Sqrt	Yes	Yes
Reshape	Yes	Yes
Sqrt	Yes	Yes
Subtract	Yes	Yes
Sum	Yes	Yes
Sum of Elements	Yes	Yes
Trigonometric Function	Yes	No
Unary Minus	Yes	Yes
Vector Concatenate	Yes	Yes

Supported Simulink Blocks in Other Libraries

The table shows the list of supported blocks in other **HDL Coder** block libraries.

Block Library	Supported blocks with single and double data types
Discrete	The supported blocks include Zero Order Hold and the set of delay blocks including Integer Delay and Tapped Delay.
HDL Operations	All blocks are supported.
HDL RAMs	All blocks are supported.
HDL Subsystems	All blocks are supported.
Logic and Bit Operations	All blocks are supported.
Lookup Tables	Direct Lookup Table (n-D) and n-D Lookup Table blocks are supported.
Model Verification	All blocks are supported.
Model-Wide Utilities	All blocks are supported.
Ports & Subsystems	Enable, reset, input, and output ports, model references, and subsystem blocks are supported.
Signal Attributes	All blocks are supported.
Signal Routing	All blocks are supported.
Sources	The supported blocks include Inport, Constant, and Ground blocks.
Sinks	All blocks are supported.
User-Defined Functions	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.
- Switch block with input to the control port as a floating-point type.
- 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 20-79

Functions

`hdlcoder.createFloatingPointTargetConfig`

Properties

FPToleranceStrategy | FPToleranceValue

Related Examples

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- “Numeric Considerations with Native Floating-Point” on page 10-69
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92

Supported Data Types and Scope

In this section...
“Supported Data Types” on page 10-110
“Unsupported Data Types” on page 10-112
“Scope for Variables” on page 10-113

Supported Data Types

HDL Coder supports the following subset of MATLAB data types.

Types	Supported Data Types	Restrictions
Integer	<ul style="list-style-type: none"> • uint8, uint16, uint32, uint64 • int8, int16, int32, int64 	In Simulink, MATLAB Function block ports must use numeric types <code>sfixed64</code> or <code>ufixed64</code> for 64-bit data.

Types	Supported Data Types	Restrictions
Real	<ul style="list-style-type: none"> double single 	<p>HDL code generated with <code>double</code> or <code>single</code> data types can be used for simulation, but is not synthesizable.</p> <p>When you have floating-point data types, to generate synthesizable HDL code, use:</p> <ul style="list-style-type: none"> 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 10-65. 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 FPGA Floating-Point Target Libraries” on page 31-14.
Character	<code>char</code>	
Logical	<code>logical</code>	
Fixed point	<ul style="list-style-type: none"> Scaled (binary point only) fixed-point numbers Custom integers (zero binary point) 	<p>Fixed-point numbers with slope (not equal to 1.0) and bias (not equal to 0.0) are not supported.</p> <p>Maximum word size for fixed-point numbers is 128 bits.</p>
Vectors	<ul style="list-style-type: none"> <code>unordered {N}</code> <code>row {1, N}</code> <code>column {N, 1}</code> 	<p>The maximum number of vector elements allowed is 2^{32}.</p> <p>Before a variable is subscripted, it must be fully defined.</p>

Types	Supported Data Types	Restrictions
Matrices	{N, M}	<p>Matrices are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.</p> <p>Do not use matrices in the testbench.</p>
Structures	struct	<p>Arrays of structures are not supported.</p> <p>For the FPGA Turnkey and IP Core Generation workflows, structures are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.</p>
Enumerations	enumeration	<p>Enumeration values must be monotonically increasing.</p> <p>If your target language is Verilog, all enumeration member names must be unique within the design.</p> <p>Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods:</p> <ul style="list-style-type: none"> • IP Core Generation workflow • FPGA Turnkey workflow • FPGA-in-the-Loop • HDL Cosimulation

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.

Verilog HDL Import: Import Verilog Code and Generate Simulink Model

In this section...
“Why Use HDL Import?” on page 10-114
“HDL Import Requirements” on page 10-114
“How to Import HDL Code” on page 10-115
“Model Location” on page 10-115
“Errors and Warnings” on page 10-115

Use HDL import to import synthesizable HDL code into the Simulink modeling environment. 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.

Why Use HDL Import?

By importing the HDL code into Simulink, you can:

- Debug your HDL design in a model-based simulation environment. To debug internal signals, designate the signals as test points and enable HDL DUT port generation for the test point signals. When you generate code, HDL Coder propagates the signals to the top level of your Simulink model.
- Improve the area and timing of your design on the target hardware by using model-level and block-level speed and area optimizations such as resource sharing and distributed pipelining.
- Deploy the design to FPGA and SoC platforms by using the **Generic ASIC/FPGA and IP Core Generation** workflows in the HDL Workflow Advisor.
- Verify the functionality of the HDL design by generating a validation model or an HDL test bench. If you have HDL Verifier™, you can verify the design by using Cosimulation, SystemVerilog DPI Test Bench, or FPGA-in-the-Loop.

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.

How to Import HDL Code

To import the HDL code, at the MATLAB Command Window, run the `importhdl` function. For example, to import a Verilog file `example.v`, at the command line, enter:

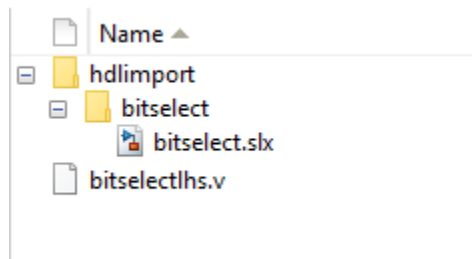
```
importhdl('example.v')
```

The function parses the HDL input file that you specified and generates the corresponding Simulink model, and provides a link to open the model.

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 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, the code generator 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;
```

See Also

Functions

`checkhdl` | `makehdl`

More About

- “Limitations of Verilog HDL Import” on page 10-123

Supported Verilog Constructs for HDL Import

In this section...
“Module Definition and Instantiations” on page 10-117
“Data Types and Vectors” on page 10-118
“Identifiers and Comments” on page 10-118
“Assignments” on page 10-119
“Operators” on page 10-119
“Conditional and Looping Statements” on page 10-120
“Procedural Blocks and Events” on page 10-120
“Other Constructs” on page 10-121

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	-

Verilog Constructs	Supported?	Comments
Local parameter declaration	Yes	-
Parameter declaration	Yes	--
Module instantiation	Yes	<ul style="list-style-type: none"> Recursive module instantiation is not supported. Module instantiation does not support unconnected ports.

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 indexinh	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 (<code>`define</code> , <code>`undef</code> , <code>`ifndef</code> , <code>`else if</code>)	Yes	-

Assignments

Verilog Constructs	Supported?	Comments
Continuous assignment	Yes	-
Blocking assignment	Yes	-
Non-blocking assignment	Yes	-
Procedural assignment (Always block)	Yes	-

Operators

Verilog Constructs	Supported?	Comments
Arithmetic operators (+, -, *, **, /, <<<, >>>)	Yes	-
Logical operators (<<, >>, !, &&, , ==, !=)	Yes	-
Relational operators (>, <, >=, <=, ==, !=)	Yes	-
Bitwise operators (~, &, , ^, ~^, ^~)	Yes	-
Unary operators (+, -)	Yes	Supported for restricted data types

Verilog Constructs	Supported?	Comments
Power operators	Yes	Supported for restricted data types
Conditional operators (?:)	Yes	-
Concatenation	Yes	-
Bit Select	Yes	-
Reduction operators (&, ~&, , ~ , ^, ~^, or ^~)	Yes	-

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	-

Verilog Constructs	Supported?	Comments
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	-
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

More About

- “Verilog HDL Import: Import Verilog Code and Generate Simulink Model” on page 10-114
- “Limitations of Verilog HDL Import” on page 10-123

Limitations of Verilog HDL Import

Use HDL import to import synthesizable HDL code into the Simulink modeling environment. To import the HDL code, use the `importhdl` function.

This table shows limitations of HDL import and Verilog constructs that are not supported when importing your HDL code.

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
Importing of VHDL files.	-	-
Importing of Verilog files from a read-only folder.	-	-
Import of HDL files that use unsupported Verilog constructs.	-	Use the subset of Verilog constructs that is supported by HDL import. To learn more, see "Supported Verilog Constructs for HDL Import" on page 10-117.
Generation of the preprocessing files in a read-only file system that parse the HDL code you input to the <code>importhdl</code> function.	-	-
Attribute instances and comments, which are ignored.	-	-
(#)delay values, such as <code>#25</code> , which are ignored.	-	-
Enumeration data types.	-	-

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
More than one clock signal.	-	-
Modules that are multirate.	-	-
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.	-	-

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>Latch detection.</p>	<p>Presence of latches in the HDL code can result in algebraic loops in the generated Simulink model and result in model compilation failures.</p> <p>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 in an algebraic loop.</p> <pre> module testAlgebraicLoop(input [2:0] in1, output reg [2:0] out1); always@(*) begin if (cond) begin out1 = in1; end end endmodule </pre>	<p>To avoid latch inference, specify all branches in if-else conditions and in case statements.</p> <pre> module testAlgebraicLoop(input cond, input [2:0] in1, output reg [2:0] out1); always@(*) begin if (cond) begin out1 = in1; end else begin out1 = in1 + 1; end end endmodule </pre>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>Partial and complete assignment to the same variable in the Verilog code.</p>	<p>This example shows the Verilog code that performs both partial assignment and complete assignment to the variable <code>out1_reg</code>.</p> <p>The code uses partial assignment to the variable <code>out1_reg</code> in the true branch of the if-condition and complete assignment to <code>out1_reg</code> in the</p> <p>This construct is not supported. Importing the code generates an error.</p> <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>	<p>When you assign a single variable, use either complete assignment or partial assignment.</p> <p>For example, this code shows how you can modify the example to perform a complete assignment to the variable <code>out1_reg</code>. This Verilog construct is supported.</p> <pre> module testCompleteAssign (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 = 3'd0; end else begin out_reg = in1 & in2; end end assign out1 = out1_reg; endmodule </pre> <p>This example shows another Verilog code that performs partial assignment to the</p>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
		<p>variable out1_reg. This construct is also supported.</p> <pre> module testPartialAssign (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[0] = in1[0] & in2[0]; out_reg[1] = in1[1] & in2[1]; out_reg[2] = in1[2] & in2[2]; end end assign out1 = out1_reg; endmodule </pre>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>Performing operations on clock, reset, or clock enable signals in the Verilog code.</p>	<p>When you define signals using names such as <code>clk</code>, <code>rst</code>, and <code>enb</code>, HDL import infers these signals to be the clock, global reset, and clock enable signals.</p> <p>For example, this Verilog code uses an explicit assignment with the clock signal <code>clk</code>. This construct is not supported.</p> <pre> module testOperationOnClkBundl (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 </pre>	<p>Make sure that your Verilog code does not perform operations on any of the signals in the clock bundle.</p> <p>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 <code>clockBundle</code> name-value pair of the <code>importhdl</code> function for possible signal names that are inferred as signals in the clock bundle.</p>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>Sensitivity of clock or reset signal to different clock edges or different reset edges inside the same module.</p>	<p>This means that you cannot have one <code>always</code> block with the clock signal sensitive to the positive edge and the other <code>always</code> block with the clock signal sensitive to the negative edge inside the same module.</p> <p>For example, this Verilog code uses two different edges of the same clock, which is not supported.</p> <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 negeedge */ always@(negeedge clk) begin out2_reg <= in1 in2; end assign out1 = out1_reg; assign out2 = out2_reg; endmodule </pre>	<p>Make sure that your Verilog code does not use both edges of the clock or reset signal in the same module. Use either <code>posedge</code> or <code>negeedge</code> of the clock.</p>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>Realization of synchronous and asynchronous circuits inside the same module.</p>	<p>For example, this Verilog code realizes both circuits in the same module.</p> <pre data-bbox="497 543 905 1145"> 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 posedg reset) begin out2_reg <= in1 in2; end assign out1 = out1_reg; assign out2 = out2_reg; endmodule </pre>	<p>Make sure that your Verilog code realizes either asynchronous circuits or synchronous circuits in the same module.</p>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>Initialization of multiple RAMs that are inferred in the same module.</p>	<p>For example, this Verilog code infers <code>sample_store0</code> and <code>sample_store1</code> as RAMs. This construct is not supported.</p> <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>	<p>Make sure that your Verilog code does not infer more than one RAM in the same module. You can separate each RAM inference into a single module.</p>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>Multiple assignments to the same variable in the true or false path of a Switch block that gets inferred.</p>	<p>For example, this Verilog code uses multiple assignments to the variable out1 in the false branch, which is not supported.</p> <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 end endmodule </pre>	<p>Make sure that your Verilog code does not use more than one assignment to the same variable in the same path of an if-condition. You can assign values to the same variable in different branches.</p>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>Usage of certain constructs when reading initial values from an initial block.</p>	<p>An initial block does not support constructs other than assignment statements or for loops with assignments. For example, this Verilog code uses an if-else condition to assign a value to the variable dout_a, which is not supported.</p> <pre> module testInitial (input cond, output reg [3:0] dout_a, dout_b) parameter AddrWidth = 3; integer i; reg [3:0] ram [7:0]; initial begin for (i=0; i<=2**AddrWidth - 1; i++) ram[i] = 0; end dout_b = 0; if (cond) begin dout_a = 0; end else begin dout_a = 4; end end end endmodule </pre> <p>An initial block supports only constants or constant assignments in the RHS. For</p>	<p>Make sure that initial blocks in your Verilog code:</p> <ul style="list-style-type: none"> • Use only assignment statements and for loops with assignments. • Use only constant values or constant expressions in the RHS. <pre> module testInitial (output reg [3:0] dout_a, dout_b) parameter AddrWidth = 3; integer i; reg [3:0] ram [7:0]; initial begin for (i=0; i<=2**AddrWidth - 1; i++) ram[i] = 0; end dout_b = 0; dout_a = 0; end endmodule </pre>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
	<p>example, this Verilog code assigns a variable <code>write_data</code> to <code>dout_a</code>, which is not supported.</p> <pre> module testInitial (input [3:0] write_data, output reg [3:0] dout_a, dout_b); 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; dout_a = write_data; end end endmodule </pre>	

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>Nonblocking assignments in combinational <code>always</code> blocks and blocking assignments in sequential <code>always</code> blocks.</p>	<p>HDL Import infers a combinational or sequential logic circuit depending on the sensitivity list of the <code>always</code> block.</p> <p>For example, this Verilog code uses a sequential assignment for the variable <code>temp</code> in a combinational <code>always</code> block, which is not supported.</p> <pre> 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 </pre>	<p>Make sure that the type of assignment matches the circuit type inferred by the <code>always</code> block.</p> <pre> 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]; //Proposed change 1: change assignment type always @(*) begin for (i=0;i<=7;i=i+1) begin temp[i] = a[i] + b[i]; end end //Proposed change 2: change circuit type // always @(posedge clk) begin // for (i=0;i<=7;i=i+1) begin // temp[i] <= a[i] + b[i]; // end // end assign c = temp; endmodule </pre>

Unsupported Constructs or Limitations	Description and Example Scenarios	Workaround
<p>All dimensions of signals not referenced at time of usage.</p>	<p>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.</p> <pre> 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]; always @(posedge clk) begin temp_reg [0] <= a[0] + b[0]; temp_reg [1] <= a[1] + b[1]; end assign c = temp; endmodule </pre>	<p>Make sure that all dimensions of a signal are used in the input Verilog code. If you specify an additional dimension, it creates an indexed bit select.</p> <pre> module dataconv(clk,a,b,c); input clk; input wire [2:0] a, b; output wire [2:0] c; reg [7:0] temp_reg [2:0][2:0]; 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]; // This code performs an indexed // temp_reg [1][0][1] = 1'b0; end assign c = temp; endmodule </pre>

See Also

Functions

checkhdl | makehdl

More About

- “Verilog HDL Import: Import Verilog Code and Generate Simulink Model” on page 10-114

Using the Float Typecast Block

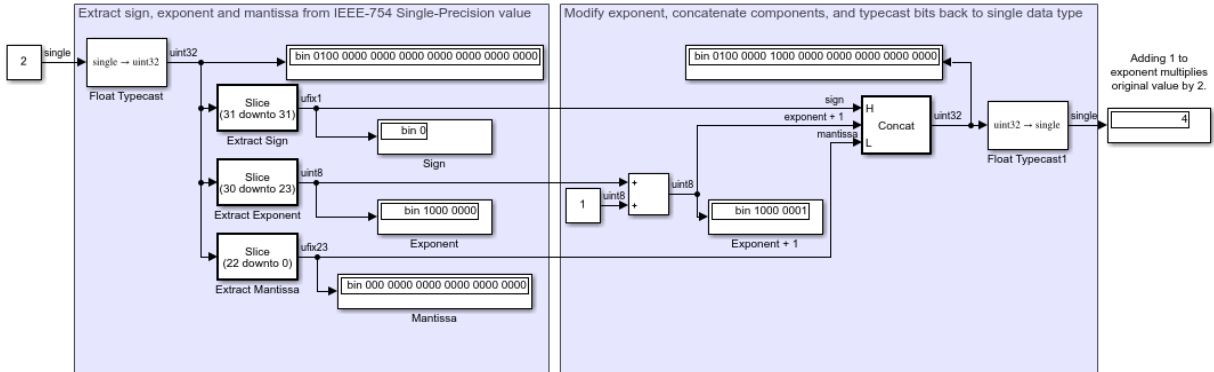
This example shows how you can use the Float Typecast block to extract the sign, exponent, and mantissa bits from a floating-point input, and then convert the bits back to a floating-point output after performing any computations.

Open this model. 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.

ans =

```
Simulink.SimulationOutput:
    tout: [51x1 double]
```

```
SimulationMetadata: [1x1 Simulink.SimulationMetadata]
ErrorMessage: [0x0 char]
```



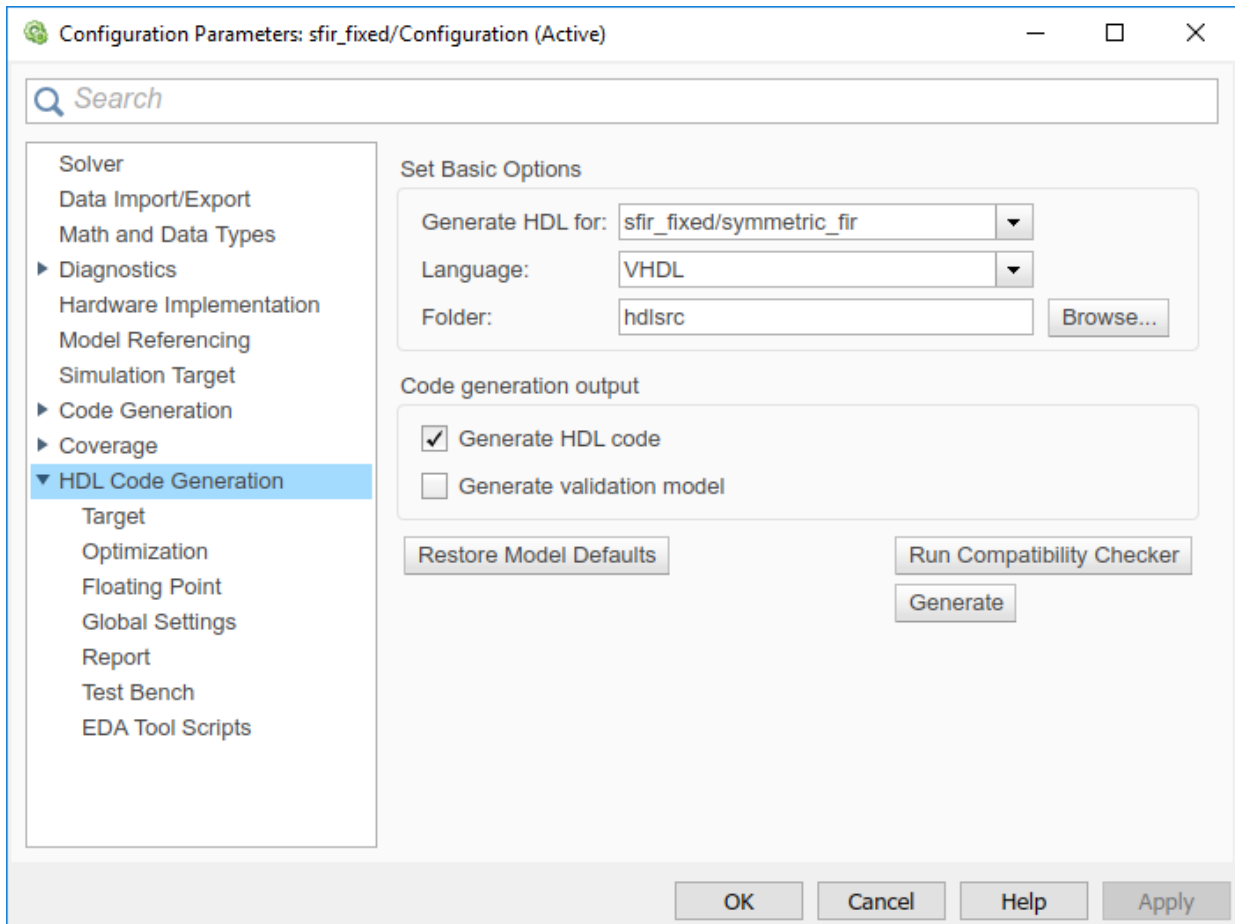
Code Generation Options in the HDL Coder Dialog Boxes

Set HDL Code Generation Options

In this section...
“HDL Code Generation Options in the Configuration Parameters Dialog Box” on page 11-2
“HDL Code Tab in Simulink Toolstrip” on page 11-4
“HDL Code Options in the Block Context Menu” on page 11-5
“The HDL Block Properties Dialog Box” on page 11-6

HDL Code Generation Options in the Configuration Parameters Dialog Box

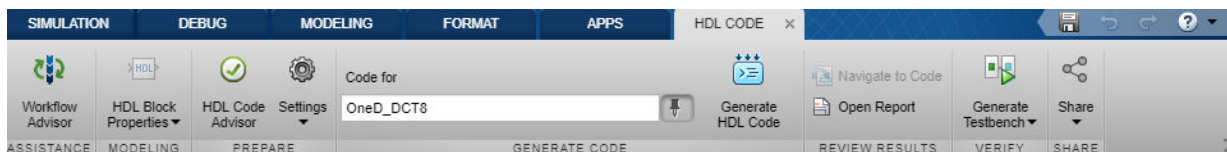
The following figure shows the top-level **HDL Code Generation** pane in the Configuration Parameters dialog box. To open this dialog box, in the Apps gallery, click **HDL Coder**. The **HDL Code** tab appears. In the **Prepare** section, click **Settings**.



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 Coder library in the Simulink Library Browser or open the HDL Block Properties dialog box for a block that you select in your model.
- **HDL Code Advisor:** Open the HDL Model Checker 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 25-30.
- **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 (<code>checkhdl</code>) on the subsystem.	Available only for subsystems.

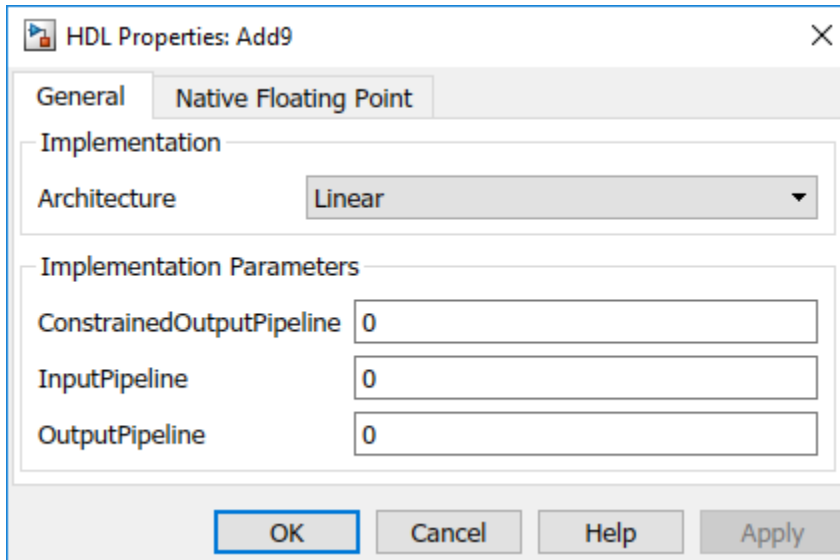
Option	Description	Availability
Generate HDL for Subsystem	Runs the HDL code generator (makehdl) and generates code for the subsystem.	Available only for subsystems.
HDL Coder Properties	Opens the Configuration Parameters dialog box, with the top-level HDL Code Generation pane selected.	Available for 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 21-71 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 25-5.	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 21-71.

HDL Code Generation Pane: General

- “HDL Code Generation Top-Level Pane Overview” on page 12-2
- “Target” on page 12-3

HDL Code Generation Top-Level Pane Overview

The top-level **HDL Code Generation** pane contains buttons that initiate code generation and compatibility checking, and sets code generation parameters.

Buttons in the HDL Code Generation Top-Level Pane

The buttons in the **HDL Code Generation** pane perform functions related to code generation. These buttons are:

Generate: Initiates code generation for the system selected in the **Generate HDL for** menu. See also `makehdl`.

Run Compatibility Checker: Invokes the compatibility checker to examine the system selected in the **Generate HDL for** menu for compatibility problems. See also `checkhdl`.

Browse: Lets you navigate to and select the target folder to which generated code and script files are written. The path to the target folder is entered into the **Folder** field.

Restore Model Defaults: Sets model parameters to their default values.

Target

In this section...

“Generate HDL for” on page 12-3
“Language” on page 12-4
“Folder” on page 12-5
“Restore Model Defaults” on page 12-6
“Run Compatibility Checker” on page 12-6
“Generate” on page 12-7

This section contains parameters in the **HDL Code Generation** pane of the Configuration Parameters dialog box. By using these parameters, you can specify the Subsystem that you want to generate HDL code for, the target HDL language, and the target folder into which code is generated.

Generate HDL for

Select the subsystem or model from which code is generated. The list includes the path to the root model and to subsystems in the model. When you specify this parameter and click the **Generate** button, HDL Coder generates code for the Subsystem that you specify. By default, the HDL code is generated in VHDL language and into the `hdlsrc` folder.

Settings

Default: The top level subsystem in the root model is selected.

Command-Line Information

Property: HDLSubsystem

Type: character vector

Value: A valid path to your subsystem

Default: Path to the top level subsystem in root model

For example, you can generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Specify the subsystem using the property `HDLSubsystem` as an argument to `makehdl`.

```
makehdl('sfir_fixed','HDLSubsystem','sfir_fixed/symmetric_fir')
```

- Pass in the path to the subsystem as a first argument to `makehdl`.

```
makehdl('sfir_fixed/symmetric_fir')
```

See also `makehdl`.

Language

Select the language (VHDL or Verilog) in which code is generated. The selected language is referred to as the target language. When you specify the **Language** and click the **Generate** button, HDL Coder generates code in that language for the Subsystem that is specified by the **Generate HDL for** parameter. By default, the HDL code is generated in VHDL language and into the `hdlsrc` folder.

The generated HDL code complies with these standards:

- VHDL-1993 (IEEE® 1076-1993) or later
- Verilog-2001 (IEEE 1364-2001) or later

Settings

Default: VHDL

VHDL

Generate VHDL code.

Verilog

Generate Verilog code.

Command-Line Information

Property: TargetLanguage

Type: character vector

Value: 'VHDL' | 'Verilog'

Default: 'VHDL'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to generate Verilog code for the `symmetric_fir` subsystem inside the `sfir_fixed` model, use either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','TargetLanguage','Verilog')
```

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TargetLanguage','Verilog')
makehdl('sfir_fixed/symmetric_fir')
```

See also `makehdl`.

Folder

Enter a path to the folder into which code is generated. Alternatively, click **Browse** to navigate to and select a folder. The selected folder is referred to as the target folder. When you specify the **Folder** and click the **Generate** button, HDL Coder generates code into that folder for the Subsystem that is specified by the **Generate HDL for** parameter. By default, the HDL code is generated in VHDL language and into the `hdlsrc` folder.

Settings

Default: The default target folder is a subfolder of your working folder, named `hdlsrc`. HDL Coder writes the generated files into this subfolder. The folder name can be a complete path name, specified as a character vector.

Command-Line Information

Property: `TargetDirectory`

Type: character vector

Value: A valid path to your target folder

Default: `'hdlsrc'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to generate HDL code into a custom target folder for the `symmetric_fir` subsystem inside the `sfir_fixed` model, use either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','TargetDirectory','C:/Temp/hdlsrc')
```

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TargetDirectory','C:/Temp/hdlsrc')
makehdl('sfir_fixed/symmetric_fir')
```

See also `makehdl`.

Restore Model Defaults

This button resets the model-level HDL settings to the default values. The block settings are not changed. To clear the block settings, use `hdlrestoreparams`.

Note If you clear the model-level settings, you cannot restore the previous settings. To restore the settings, close the model without saving and then reopen the model.

Command-Line Information

Function: `hdlrestoreparams`

Type: character vector

Value: model name

Default: ''

Run Compatibility Checker

This setting checks whether the Subsystem that you specify by using **Generate HDL for** is compatible for HDL code generation. The setting generates a HDL Check Report that displays errors, warnings, and messages. See “Check Subsystem for HDL Compatibility” on page 20-24.

Command-Line Information

Function: `checkhdl`

Type: character vector

Value: subsystem or model name

Default: ''

See Also

`checkhdl`

Generate

This setting generates HDL code for the Subsystem that you specify by using **Generate HDL for**. If the Subsystem is not HDL-compatible, the code generator displays errors in the HDL Check Report.

Command-Line Information

Function: makehdl

Type: character vector

Value: subsystem or model name

Default: ''

See Also

makehdl

HDL Code Generation Pane: Target

- “Target Overview” on page 13-2
- “Tool and Device” on page 13-3
- “Target Frequency” on page 13-9

Target Overview

The **Target** pane enables you to specify the target hardware settings.

Tool and Device

In this section...

“Synthesis Tool” on page 13-3

“Family” on page 13-4

“Device” on page 13-5

“Package” on page 13-6

“Speed” on page 13-7

This section contains parameters in the **Tool and Device** section of the **HDL Code Generation > Target** pane of the Configuration Parameters dialog box. By using the parameters in this section, you can specify the synthesis tool, and then select the **Family**, **Device**, **Package**, and **Speed** for your synthesis target.

Synthesis Tool

Specify the synthesis tool for targeting the generated HDL code. To use HDL Coder with one of the supported third-party FPGA synthesis tools, add the tool to the system path using the `hdlsetuptoolpath` function. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and **Speed** with default values for that tool.

Settings

Default: No synthesis tool specified

The options are:

No synthesis tool specified

Select this option if you do not want to perform logic synthesis. You can generate HDL code from your design.

Xilinx Vivado

Specify Xilinx Vivado as the synthesis tool.

Xilinx ISE

Specify Xilinx ISE as the synthesis tool.

Altera Quartus II

Specify Altera Quartus II as the synthesis tool.

Microsemi Libero SoC

Specify Microsemi® Libero® SoC as the synthesis tool.

If your synthesis tool is not one of the **Synthesis tool** options, see “Synthesis Tool Path Setup”.

Command-Line Information

Property: SynthesisTool

Type: character vector

Value: '' | 'Xilinx Vivado' 'Xilinx ISE' 'Altera Quartus II'

Default: ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify `Altera Quartus II` as the `SynthesisTool` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'SynthesisTool', 'Altera Quartus II')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'SynthesisTool', 'Altera Quartus II')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- `hdlsetuptoolpath`
- “Tool Setup”

Family

Specify the target device chip family for your model as a character vector. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and

Speed with default values for that tool. To find the chip family for your target device, at the MATLAB command line, enter `hdlcoder.supportedDevices`. Then, open the linked report and find your target device details.

Settings

Default: ''

Specify the target device chip family for your Simulink model as a character vector.

Command-Line Information

Property: `SynthesisToolChipFamily`

Type: character vector

Value: A valid chip family for the target device

Default: ''

For example, if your `SynthesisTool` is Xilinx Vivado, you can specify `Virtex7` as the `SynthesisToolChipFamily` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'SynthesisToolChipFamily', 'Virtex7')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','SynthesisToolChipFamily', 'Virtex7')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- `hdlsetuptoolpath`
- “Tool Setup”

Device

Specify the target device name for your model as a character vector. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and **Speed** with default values for that tool. To find the device name for your target device, at the MATLAB

command line, enter `hdlcoder.supportedDevices`. Then, open the linked report and find your target device details.

Settings

Default: ''

Specify the target device name for your Simulink model as a character vector.

Command-Line Information

Property: `SynthesisToolDevicename`

Type: character vector

Value: A valid device name for the synthesis tool

Default: ''

You can get the `SynthesisToolDeviceName` when you specify the `SynthesisTool` for your model. Consider that the `SynthesisTool` is set to `Xilinx Vivado` and the `SynthesisToolChipFamily` is set to `Virtex7`.

- To get the default device name, pass the property as an argument to the `hdlget_param` function.

```
hdlget_param('sfir_fixed', ...  
            'SynthesisToolDeviceName')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','SynthesisToolDeviceName', 'xc7v2000t')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- `hdlsetuptoolpath`
- “Tool Setup”

Package

Specify the target device package name for your model as a character vector. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and **Speed** with default values for that tool. To find the device name for your target device, at

the MATLAB command line, enter `hdlcoder.supportedDevices`. Then, open the linked report and find your target device details.

Settings

Default: ''

Specify the target device package name for your Simulink model as a character vector.

Command-Line Information

Property: `SynthesisToolPackageName`

Type: character vector

Value: A valid package name for the synthesis tool

Default: ''

You can get the `SynthesisToolPackageName` when you specify the `SynthesisTool` for your model. Consider that the `SynthesisTool` is set to `Xilinx Vivado` and the `SynthesisToolChipFamily` is set to `Virtex7`.

- To get the default device name, pass the property as an argument to the `hdlget_param` function.

```
hdlget_param('sfir_fixed', ...
            'SynthesisToolPackageName')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','SynthesisToolPackageName', 'fhg1761')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- `hdlsetuptoolpath`
- “Tool Setup”

Speed

Specify the target device speed value for your model as a character vector. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and **Speed** with default values for that tool. To find the chip family for your target device, at

the MATLAB command line, enter `hdlcoder.supportedDevices`. Then, open the linked report and find your target device details.

Settings

Default: ''

Specify the target device speed value for your Simulink model as a character vector.

Command-Line Information

Property: `SynthesisToolSpeedValue`

Type: character vector

Value: A valid speed value for the target device

Default: ''

You can get the `SynthesisToolSpeedValue` when you specify the `SynthesisTool` for your model. Consider that the `SynthesisTool` is set to `Xilinx Vivado` and the `SynthesisToolChipFamily` is set to `Virtex7`.

- To get the default device name, pass the property as an argument to the `hdlget_param` function.

```
hdlget_param('sfir_fixed', ...  
            'SynthesisToolSpeedValue')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','SynthesisToolSpeedValue', '-1')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- `hdlsetuptoolpath`
- “Tool Setup”

Target Frequency

This parameter resides in the **Objectives Settings** section of the **HDL Code Generation** > **Target** pane of the Configuration Parameters dialog box. By using this parameter, you can specify the target frequency in MHz for multiple features and workflows. Before setting the target frequency, make sure that you specify the **Synthesis Tool**.

Settings

Default: 0

This setting is the target frequency in MHz for multiple features and workflows that HDL Coder supports. The supported features are:

- **FPGA floating-point target library mapping:** Specify the target frequency that you want the IP to achieve when you use ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS). If you do not specify the target frequency, HDL Coder sets the target frequency to a default value of 200 MHz. See also “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-14.
- **Adaptive pipelining:** If your design uses multipliers, specify the synthesis tool and the target frequency. Based on these settings, HDL Coder estimates the number of pipelines that can be inserted to improve area and timing on the target platform. If you do not specify the target frequency, HDL Coder uses a target frequency of 0 MHz and does not insert adaptive pipelines. See also “Adaptive Pipelining” on page 24-68.

You can also set the target frequency by using the **Target Frequency (MHz)** setting in the **Set Target Frequency** task in the HDL Workflow Advisor.

Specify the target frequency for these workflows-

- **Generic ASIC/FPGA:** To specify the target frequency that you want your design to achieve. HDL Coder generates a timing constraint file for that clock frequency. It 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. Target frequency is not supported with Microsemi Libero SoC.
- **IP Core Generation:** 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. 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 I/O modules that are supported with Xilinx ISE, specify the target frequency to generate the clock module to produce the clock signal with that frequency.

The Speedgoat I/O modules that are supported with 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.

- **FPGA Turnkey:** To generate the clock module to produce the clock signal with that frequency automatically.

Command-Line Information

Property: TargetFrequency

Type: integer

Value: integer greater than or equal to 0

Default: 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify the `TargetFrequency` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'TargetFrequency', '300')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'TargetFrequency', '300')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`

- “Set Target Frequency” on page 37-7
- “Customize Floating-Point IP Configuration” on page 31-23

HDL Code Generation Pane: Optimization

- “Optimization Overview” on page 14-2
- “Balance delays” on page 14-3
- “RAM Mapping” on page 14-5
- “Transform non zero initial value delay” on page 14-8
- “Multiplier partitioning threshold” on page 14-10
- “Distributed Pipelining” on page 14-12
- “Clock Rate Pipelining” on page 14-15
- “Adaptive pipelining” on page 14-18
- “Preserve design delays” on page 14-20
- “Resource Sharing of Adders and Multipliers” on page 14-22
- “Resource Sharing of Multiply-Add and Other Blocks” on page 14-29
- “Share Floating-Point IPs” on page 14-35
- “Multicycle Path Constraints” on page 14-37

Optimization Overview

The **Optimization** pane enables you to specify various optimizations such as delay balancing, resource sharing and pipelining. To improve the area and timing of your design on the target hardware, specify these settings. This pane also contains a **Multicycle Path Constraints** section. Use the settings in this section to specify the timing requirements that the Synthesis tool must meet for your design.

Balance delays

This parameter is in the **HDL Code Generation > Optimization > General** tab of the Configuration Parameters dialog box.

When you enable certain optimizations such as pipelining or resource sharing, or specify certain block implementations and generate code, HDL Coder introduces pipeline delays along certain signal paths in your model. By default, the **Balance delays** setting is enabled. The code generator detects these pipeline delays introduced along one path and then inserts matching delays on other paths.

To make sure that the generated model after HDL code generation is functionally equivalent to the original Simulink model, leave this setting enabled. If you disable this setting, HDL Coder generates a warning that numerical differences can occur in the validation model. To fix this warning, enable **Balance delays** on the model or run the model check “Check delay balancing setting” on page 38-12.

Settings

Default: On

On

Enables delay balancing on your model. If HDL Coder detects introduction of new delays along one path, matching delays are inserted on the other paths. When delay balancing is enabled, the generated model is functionally equivalent to the original model.

Off

The latency along signal paths might not be balanced, and the generated model might not be functionally equivalent to the original model.

Command-Line Information

Property: BalanceDelays

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `BalanceDelays` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'BalanceDelays', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'BalanceDelays', 'on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Delay Balancing” on page 24-32
- “BalanceDelays” on page 21-5

RAM Mapping

In this section...

“Map pipeline delays to RAM” on page 14-5

“RAM mapping threshold (bits)” on page 14-6

This section contains parameters in the **HDL Code Generation > Target > General** tab of the Configuration Parameters dialog box. Using the parameters in this section, you can reduce the area usage on the target device by trading-off block RAMs for registers. The parameters specify whether you want to map pipeline registers in the generated code to RAM, and the minimum RAM size for mapping to block RAMs on the FPGA.

Map pipeline delays to RAM

Map pipeline registers in the generated HDL code to RAM. Certain speed or area optimizations such as pipelining and resource sharing, or certain block implementations that you specify can insert pipeline registers in the generated HDL code. You can save area on the target device by mapping these pipeline registers to RAM.

Settings

Default: Off

On

Map pipeline registers in the generated HDL code to RAM. To map these registers to block RAMs, the RAM size must be greater than or equal to the RAM mapping threshold in bits. RAM size is the product Delay length * Word length * Vector length * Complex length.

Off

Do not map pipeline registers in the generated HDL code to RAM.

Command-Line Information

Property: MapPipelineDelaysToRAM

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `MapPipelineDelaysToRAM` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'MapPipelineDelaysToRAM','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MapPipelineDelaysToRAM','on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “UseRAM” on page 21-34
- “RAM Mapping for MATLAB Code” on page 8-2

RAM mapping threshold (bits)

Specify the minimum RAM size in bits for mapping to block RAMs. The code generator determines whether to use registers or RAM resources on the FPGA by comparing the RAM size of your design with the RAM mapping threshold that you specify.

Settings

Default: 256

The RAM mapping threshold must be an integer greater than or equal to zero. HDL Coder uses the threshold to determine whether or not to map the following elements to block RAMs instead of to registers:

- Delay blocks
- Persistent arrays in MATLAB Function blocks

Command-Line Information**Property:** RAMMappingThreshold**Type:** integer**Value:** integer greater than or equal to 0**Default:** 256

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can set the `RAMMappingThreshold` to 1024 when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'RAMMappingThreshold','1024')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','RAMMappingThreshold','1024')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “UseRAM” on page 21-34
- “RAM Mapping for MATLAB Code” on page 8-2

Transform non zero initial value delay

This parameter is in the **HDL Code Generation > Optimization > General** tab of the Configuration Parameters dialog box. Enable this option to optimize Delay blocks with non zero initial condition.

Settings

Default: On

On

Transform Delay blocks with nonzero **Initial condition** in your Simulink model to Delay blocks with zero **Initial condition** and some additional logic in the generated HDL code.

By using this transformation, HDL Coder can perform optimizations such as sharing, distributed pipelining, and clock-rate pipelining more effectively, and prevent an assertion from being triggered in the validation model.

Off

Do not transform Delay blocks with nonzero **Initial condition** in your Simulink model.

Command-Line Information

Property: TransformNonZeroInitValDelay

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can set the `TransformNonZeroInitValDelay` property to `on` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'TransformNonZeroInitValDelay','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TransformNonZeroInitValDelay','on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

`makehdl`

Multiplier partitioning threshold

This parameter is in the **HDL Code Generation > Optimization > General** tab of the Configuration Parameters dialog box. Use this parameter to specify the maximum input bit width for multipliers in your design.

Settings

Default: Inf

N, where N is an integer greater than or equal to 2

Partition multipliers so that N is the maximum multiplier input bit width.

This parameter specifies the maximum input bit width for a multiplier. If at least one of the inputs to the multiplier has a bit width greater than the threshold value, the code generator splits the multiplier into smaller multipliers.

To improve hardware mapping results, set the multiplier partitioning threshold to the input bit width of the DSP or multiplier hardware on your target device.

Inf

Do not partition multipliers.

Command-Line Information

Property: MultiplierPartitioningThreshold

Type: integer

Value: integer greater than or equal to 0

Default: Inf

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can set the `MultiplierPartitioningThreshold` to 16 when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
       'MultiplierPartitioningThreshold','16')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'MultiplierPartitioningThreshold', '16')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Multiplier promotion threshold” on page 14-27

Distributed Pipelining

This section contains parameters in the **HDL Code Generation > Optimization > Pipelining** tab of the Configuration Parameters dialog box. Using the parameters in this section, you can improve the timing of your design on the target device. Enable the hierarchical distributed pipelining optimization, and specify whether to prioritize the distributed pipelining algorithm for numerical integrity or performance.

Hierarchical distributed pipelining

Hierarchical distributed pipelining extends the scope of distributed pipelining by distributing delays across subsystem hierarchies. This optimization moves the delays within a Subsystem while preserving the hierarchy.

Settings

Default: Off

On

Enable retiming across a subsystem hierarchy. HDL Coder applies retiming hierarchically from the top-level Subsystem. To move delays inside a Subsystem, in the HDL Block Properties for that Subsystem, set **DistributedPipelining** to on. Hierarchical distributed pipelining stops distributing delays when it reaches a Subsystem that has **DistributedPipelining** set to off.

Off

Distributes pipelines within a Subsystem, if you have **DistributedPipelining** set to on for that Subsystem.

Dependency

If you select the **Preserve design delays** check box, distributed pipelining does not move the design delays.

Command-Line Information

Property: HierarchicalDistPipelining

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `HierarchicalDistPipelining` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'HierarchicalDistPipelining','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','HierarchicalDistPipelining','on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Distributed Pipelining” on page 24-49
- “DistributedPipelining” on page 21-10

Distributed pipelining priority

Specify the priority for your distributed pipelining algorithm.

Settings

Default: Numerical Integrity

Numerical Integrity

Prioritize numerical integrity when distributing pipeline registers.

This option uses a conservative retiming algorithm that does not move registers across a component if the functional equivalence to the original design is unknown.

Performance

Prioritize performance over numerical integrity.

Use this option if your design requires a higher clock frequency and the Simulink behavior does not need to strictly match the generated code behavior. This option

uses a more aggressive retiming algorithm that moves registers across a component even if the modified design's functional equivalence to the original design is unknown.

Command-Line Information

Property: DistributedPipeliningPriority

Type: character vector

Value: 'NumericalIntegrity' | 'Performance'

Default: 'NumericalIntegrity'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `DistributedPipeliningPriority` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'DistributedPipeliningPriority', 'Performance')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'DistributedPipeliningPriority', 'Performance')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Distributed Pipelining” on page 24-49
- “DistributedPipelining” on page 21-10

Clock Rate Pipelining

This section contains parameters in the **HDL Code Generation > Optimization > Pipelining** tab of the Configuration Parameters dialog box. Using the parameters in this section, you can improve the timing of your design on the target device. Enable clock-rate pipelining and allow clock-rate pipelining at the DUT output ports to run the pipeline registers at a faster clock rate on the target FPGA device.

Clock-rate pipelining

If your design contains multicycle paths, use clock-rate pipelining to insert pipeline registers at a clock rate that is faster than the data rate. This optimization improves the clock frequency and reduces the area usage without introducing additional latency. 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.

Settings

Default: On

On

Insert pipeline registers at the clock rate for multi-cycle paths.

Off

Insert pipeline registers at the data rate for multi-cycle paths.

Dependency

If you specify an **Oversampling factor** greater than one, make sure that you select the **Clock-rate pipelining** check box. Clock-rate pipelining identifies regions in your model that run at the same slow data rate and are delimited by Delay blocks or blocks that introduce a rate transition. The code generator 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.

Command-Line Information

Property: ClockRatePipelining

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ClockRatePipelining` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'ClockRatePipelining','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockRatePipelining','on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Oversampling factor” on page 16-18
- “ClockRatePipelining” on page 21-6
- “Clock-Rate Pipelining” on page 24-63

Allow clock-rate pipelining of DUT output ports

For DUT output ports, insert pipeline registers at the clock rate instead of the data rate.

Settings

Default: Off

On

At DUT output ports, insert pipeline registers at clock rate.

Off

At DUT output ports, insert pipeline registers at data rate.

Dependency

When you specify this parameter, make sure that you select the **Clock-rate pipelining** check box.

Command-Line Information

Property: ClockRatePipelineOutputPorts

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ClockRatePipelineOutputPorts` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'ClockRatePipelineOutputPorts','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockRatePipelineOutputPorts','on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “ClockRatePipelining” on page 21-6
- “Clock-Rate Pipelining” on page 24-63
- “Oversampling factor” on page 16-18

Adaptive pipelining

This parameter resides in the **HDL Code Generation > Optimization > Pipelining** tab of the Configuration Parameters dialog box. Use this parameter to insert pipeline registers to the blocks in your design, reduce the area usage, and maximize the achievable clock frequency on the target FPGA device.

Settings

Default: On

On

Insert adaptive pipeline registers in your design. For HDL Coder to insert adaptive pipelines, you must specify the synthesis tool.

Off

Do not insert adaptive pipeline registers.

Dependency

When you specify this parameter, in the **HDL Code Generation > Targetpane**:

- Specify the **Synthesis Tool**. If your design has multipliers, specify the **Synthesis Tool** and the **Target Frequency (MHz)** for adaptive pipeline insertion.
- In the **General** tab, make sure that the **Clock-rate pipelining** check box is selected to insert pipeline registers at the faster clock rate.
- In the **General** tab, make sure that the **Balance delays** check box is selected.
- In the **Resource Sharing** tab, enable **Adders**, and specify the **SharingFactor** on the DUT Subsystem to share resources and insert adaptive pipelines, which saves area and improves timing.

Command-Line Information

Property: AdaptivePipelining

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ClockRatePipelineOutputPorts` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'ClockRatePipelineOutputPorts', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'ClockRatePipelineOutputPorts', 'on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Adaptive pipelining” on page 14-18
- “AdaptivePipelining” on page 21-4

Preserve design delays

This parameter resides in the **HDL Code Generation > Optimization > Pipelining** tab of the Configuration Parameters dialog box. Enable this parameter to prevent distributed pipelining from moving design delays.

Settings

Default: Off

On

Prevent distributed pipelining from moving design delays.

Off

Do not prevent distributed pipelining from moving design delays.

Command-Line Information

Property: PreserveDesignDelays

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `PreserveDesignDelays` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'PreserveDesignDelays','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','PreserveDesignDelays','on')  
makehdl('sfir_fixed/symmetric_fir')
```


See Also

- “Distributed Pipelining” on page 24-49
- “DistributedPipelining” on page 21-10

Resource Sharing of Adders and Multipliers

This section contains parameters in the **HDL Code Generation > Optimization > Resource sharing** tab of the Configuration Parameters dialog box. Enable these parameters to save resources on the target device by specifying whether to share adders and multipliers in your design, and the minimum sharing bitwidth.

Share Adders

Enable this parameter to share adders with the resource sharing optimization. Resource sharing identifies Add or Sum blocks in your design that have two inputs and replaces them with a single Add or Sum block. This optimization saves area on the target FPGA device.

Settings

Default: Off

On

When resource sharing is enabled, this optimization shares adders with a bit width greater than or equal to the **Adder sharing minimum bitwidth**.

Off

Do not share adders.

Dependency

- To share adders in your design, in the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.
- When you specify the **Adder sharing minimum bitwidth**, the code generator shares adders that have a bit width greater than or equal to the minimum bit width. The default minimum bit width for sharing adders is zero.

Command-Line Information

Property: ShareAdders

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareAdders` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
       'ShareAdders','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ShareAdders','on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “SharingFactor” on page 21-29
- “Resource Sharing” on page 24-27
- “Resource Sharing of Multiply-Add and Other Blocks” on page 14-29

Adder sharing minimum bitwidth

Use this parameter to specify the minimum bit width that is required to share adders with the resource sharing optimization.

Settings

Default: 0

01

No minimum bit width for shared adders.

N, where N is an integer greater than 1

When resource sharing and adder sharing are enabled, share adders with a bit width greater than or equal to N.

Dependency

To share adders in your design:

- In the **Resource Sharing** tab, enable the **Adders** setting.
- In the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.

Command-Line Information**Property:** AdderSharingMinimumBitwidth**Type:** integer**Value:** integer greater than or equal to 0**Default:** 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `AdderSharingMinimumBitwidth` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'AdderSharingMinimumBitwidth',16)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','AdderSharingMinimumBitwidth',16)  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “SharingFactor” on page 21-29
- “Resource Sharing” on page 24-27

Share Multipliers

Enable this parameter to share multipliers with the resource sharing optimization. Resource sharing identifies Product or Gain blocks in your design that have two inputs and replaces them with a single Product or Gain block. This optimization saves area on the target FPGA device. Share multipliers with the resource sharing optimization.

Settings**Default:** On

On

When resource sharing is enabled, share multipliers with a bit width greater than or equal to the **Multiplier sharing minimum bitwidth**. For successfully sharing multipliers, the input fixed-point data types must have the same wordlength. The fraction lengths and signs of the fixed-point data types can be different.

Off

Do not share multipliers.

Dependency

- To share multipliers in your design, in the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.
- When you specify the **Multiplier sharing minimum bitwidth**, the code generator shares multipliers that have a bit width greater than or equal to the minimum bit width. The default minimum bit width for sharing multipliers is zero.

Command-Line Information

Property: ShareMultipliers

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareMultipliers` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ShareMultipliers','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ShareMultipliers','on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “SharingFactor” on page 21-29
- “Resource Sharing” on page 24-27
- “Resource Sharing of Multiply-Add and Other Blocks” on page 14-29

Multiplier sharing minimum bitwidth

Use this parameter to specify the minimum bit width that is required to share multipliers with the resource sharing optimization.

Settings

Default: 0

01

No minimum bit width for shared multipliers.

N, where N is an integer greater than 1

When resource sharing and multiplier sharing are enabled, share multipliers with a bit width greater than or equal to N.

Dependency

To share multipliers in your design:

- In the **Resource Sharing** tab, make sure that the **Multipliers** check box is selected.
- In the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.

Command-Line Information

Property: MultiplierSharingMinimumBitwidth

Type: integer

Value: integer greater than or equal to 0

Default: 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `MultiplierSharingMinimumBitwidth` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'MultiplierSharingMinimumBitwidth',16)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MultiplierSharingMinimumBitwidth',16)
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “SharingFactor” on page 21-29
- “Resource Sharing” on page 24-27

Multiplier promotion threshold

To share smaller multipliers with other larger multipliers by using the resource sharing optimization, specify the multiplier promotion threshold. This threshold specifies the maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers.

Settings

Default: 0

0

No difference in word-length between the multipliers. In other words, HDL Coder shares multipliers that have the same word-lengths.

N, where N is an integer greater than 0

Maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers. HDL Coder promotes and shares multipliers with different word-lengths, if the difference in word-lengths is less than or equal to N.

Dependency

To share multipliers in your design:

- In the **Resource Sharing** tab, make sure that the **Multipliers** check box is selected.
- In the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.

Command-Line Information**Property:** MultiplierPromotionThreshold**Type:** integer**Value:** integer greater than or equal to 0**Default:** 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `MultiplierPromotionThreshold` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'MultiplierPromotionThreshold',8)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MultiplierPromotionThreshold',8)  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “SharingFactor” on page 21-29
- “Resource Sharing” on page 24-27

Resource Sharing of Multiply-Add and Other Blocks

This section contains parameters in the **HDL Code Generation > Optimization > Resource sharing** tab of the Configuration Parameters dialog box. Enable these parameters to save resources on the target device by specifying whether to share Multiply-Add blocks, atomic subsystems, and MATLAB Function blocks in your design.

Share Multiply-Add blocks

Share Multiply-Add blocks with the resource sharing optimization.

Settings

Default: On

On

When resource sharing is enabled, share Multiply-Add blocks with a bit width greater than or equal to **Multiply-Add block sharing minimum bitwidth**.

Off

Do not share Multiply-Add blocks.

Dependency

- To share Multiply-Add blocks in your design, in the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.
- When you specify the **Multiply-Add block sharing minimum bitwidth**, the code generator shares Multiply-Add blocks that have a bit width greater than or equal to the minimum bit width. The default minimum bit width for sharing Multiply-Add blocks is zero.

Command-Line Information

Property: ShareMultiplyAdds

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareMultiplyAdds` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'ShareMultiplyAdds', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'ShareMultiplyAdds', 'on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Resource Sharing of Adders and Multipliers” on page 14-22
- “Resource Sharing” on page 24-27
- “Resource Sharing of Adders and Multipliers” on page 14-22

Multiply-Add block sharing minimum bitwidth

Use this parameter to specify the minimum bit width that is required to share Multiply-Add with the resource sharing optimization.

Settings

Default: 0

01

No minimum bit width for shared Multiply-Add blocks.

N, where N is an integer greater than 1

When resource sharing and Multiply-Add block sharing are enabled, share Multiply-Add blocks with a bit width greater than or equal to N.

Dependency

To share Multiply-Add blocks in your design:

- In the **Resource Sharing** tab, make sure that the **Multiply-Add blocks** check box is selected.

- In the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.

Command-Line Information

Property: MultiplierAddSharingMinimumBitwidth

Type: integer

Value: integer greater than or equal to 0

Default: 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `MultiplierAddSharingMinimumBitwidth` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'MultiplierAddSharingMinimumBitwidth',16)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed',MultiplierAddSharingMinimumBitwidth',16)
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Resource Sharing of Adders and Multipliers” on page 14-22
- “Resource Sharing” on page 24-27

Share Atomic subsystems

Share atomic subsystems with the resource sharing optimization.

Settings

Default: On

On

When resource sharing is enabled, share atomic subsystems.

Off

Do not share atomic subsystems.

Dependency

To share Atomic Subsystem blocks in your design, in the HDL Block Properties for the parent DUT Subsystem, specify the **SharingFactor**.

Command-Line Information

Property: ShareAtomicSubsystems

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareMultiplyAdds` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'ShareAtomicSubsystems', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'ShareAtomicSubsystems', 'on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Resource Sharing of Adders and Multipliers” on page 14-22
- “Resource Sharing” on page 24-27
- “Resource Sharing of Adders and Multipliers” on page 14-22

Share MATLAB Function blocks

Share MATLAB Function blocks with the resource sharing optimization.

Settings

Default: On

On

When resource sharing is enabled, share MATLAB Function blocks.

Off

Do not share MATLAB Function blocks.

Dependency

To share MATLAB Function blocks in your design, in the HDL Block Properties for the parent DUT Subsystem, specify the **SharingFactor**.

Command-Line Information

Property: ShareMATLABBlocks

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareMATLABBlocks` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ShareMATLABBlocks', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'ShareMATLABBlocks', 'on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Resource Sharing of Adders and Multipliers” on page 14-22

- “Resource Sharing” on page 24-27
- “Resource Sharing of Adders and Multipliers” on page 14-22

Share Floating-Point IPs

This parameter resides in the **HDL Code Generation > Optimization > Resource Sharing** tab of the Configuration Parameters dialog box. Use this parameter to share floating-point IP blocks in the target hardware with the resource sharing optimization.

Settings

Default: On

On

When you enable resource sharing, HDL Coder shares floating-point IP blocks.

Off

Do not share floating-point IP blocks.

Dependency

To share floating-point IPs:

- In the HDL Block Properties for the parent DUT Subsystem, specify the **SharingFactor**. The number of floating-point IP blocks that get shared depends on the **SharingFactor** that you specify for the subsystem.
- In the **HDL Code Generation > Global Settings > Floating Point Target** tab, set the **Floating Point IP Library** to a value other than None.

Command-Line Information

Property: ShareFloatingPointIP

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareFloatingPointIP` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'ShareFloatingPointIP', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'ShareFloatingPointIP', 'on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Resource Sharing of Adders and Multipliers” on page 14-22
- “Resource Sharing” on page 24-27
- “Resource Sharing of Adders and Multipliers” on page 14-22
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65

Multicycle Path Constraints

In this section...

“Enable based constraints” on page 14-37

“Register-to-register path info” on page 14-38

This section contains parameters in the **Multicycle Path Constraints** section of the **HDL Code Generation > Optimization** pane of the Configuration Parameters dialog box.

Synthesis tools require that data propagates from a source register to a destination register within one clock cycle. However, multicycle paths cannot complete their execution within one clock cycle and therefore cannot meet the timing requirements. To meet the timing requirement of multicycle paths, use the parameters in this section to generate a register-to-register path information file or to generate enable-based constraints that uses the timing controller enable signals.

Enable based constraints

To meet the timing requirement of multicycle paths in your Simulink design in single clock mode, use enable-based constraints. Enable-based constraints relax the timing requirement by allowing multiple clock cycles for data to propagate between the registers. The constraints use the timing controller enable signals to create enable-based register groups, with registers in each group driven by the same clock enable.

Settings

Default: Off

On

When you enable this setting and generate HDL code, HDL Coder generates a constraints file with the naming convention `dutname_constraints`. The format of the file name depends on the synthesis tool that you specify. The constraints file defines the timing requirements of multicycle paths and contains information about the clock multiples for calculating the setup and hold time information.

Off

Do not generate a multicycle path constraints file.

Dependency

If you select the **Enable based constraints** check box, make sure that you clear the **Clock-rate pipelining** check box. Using enable-based multicycle path constraints is an alternative to the clock-rate pipelining optimization. You can clear the **Clock-rate pipelining** check box in the **HDL Code Generation > Target > Pipelining** tab.

Command-Line Information

Parameter: MulticyclePathConstraints

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `MulticyclePathConstraints` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'MulticyclePathConstraints','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MulticyclePathConstraints','on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Meet Timing Requirements Using Enable-Based Multicycle Path Constraints” on page 40-58
- “Use Multicycle Path Constraints to Meet Timing for Slow Paths”

Register-to-register path info

Generate a text file that reports multicycle path constraint information. The text file describes one or more multicycle path constraints that is agnostic to the synthesis tool.

You must convert this information to the format required by the synthesis tool. It is recommended that you use the enable-based constraints setting instead to meet the timing requirements of multicycle paths. When you use that setting, the generated constraints are more robust to name changes in synthesis tools, and are supported with Xilinx Vivado, Xilinx ISE, and Altera Quartus II.

Settings

Default: Off

On

Generate a text file that reports multicycle path constraint information, for use with synthesis tools.

The file name for the multicycle path information file derives from the name of the DUT and the postfix `'_constraints'`, as follows:

DUTname_constraints.txt

For example, if the DUT name is `symmetric_fir`, the name of the multicycle path information file is `symmetric_fir_constraints.txt`.

Off

Do not generate a multicycle path information file.

Command-Line Information

Parameter: `MulticyclePathInfo`

Type: character vector

Value: `'on'` | `'off'`

Default: `'off'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `MulticyclePathInfo` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'MulticyclePathInfo','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MulticyclePathInfo','on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Generate Multicycle Path Information Files” on page 22-19

HDL Code Generation Pane: Floating Point

- “Floating Point Overview” on page 15-2
- “Floating Point IP Library” on page 15-3
- “Native Floating Point” on page 15-5
- “FPGA Floating-Point Libraries” on page 15-10

Floating Point Overview

The **Floating Point** pane enables you to specify the floating point IP library. You can specify whether to generate code the native floating point support in HDL Coder or by instantiating the third-party Intel or Xilinx floating-point libraries.

Floating Point IP Library

This parameter resides in the **HDL Code Generation > Floating Point** pane of the Configuration Parameters dialog box. Use this parameter to specify the floating-point target library.

Settings

Default: NONE

The options are:

None

Select this option if you do not want the design to map to floating-point target libraries.

Native Floating Point

Specify native floating-point as the library. You can specify the latency strategy and whether to handle denormal numbers in your design.

Altera Megafunctions (ALTERA FP FUNCTIONS)

Specify Altera Megafunctions (ALTERA FP FUNCTIONS) as the floating-point target library. You can provide the IP Target frequency.

Altera Megafunctions (ALTFP)

Specify Altera Megafunctions (ALTFP) as the floating-point target library. You can provide the objective and latency strategy for the IP.

Xilinx LogiCORE

Specify Xilinx LogiCORE® as the floating-point target library. You can provide the objective and latency strategy for the IP.

Command-Line Information

To set the floating-point library:

- 1 Create a floating-point target configuration object for the floating-point library. This example shows how to create an `hdlcoder.FloatingPointTargetConfig` object for the Native Floating Point library:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
```

- 2 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

See Also

- `hdlcoder.createFloatingPointTargetConfig`
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-14

Native Floating Point

This section contains parameters in the **HDL Code Generation > Floating Point** pane of the Configuration Parameters dialog box. Use these parameters to specify the latency strategy, whether to handle denormal numbers in your design, and how to perform mantissa multiplication. To specify these settings, **Floating Point IP Library** must be set to Native Floating Point.

Latency Strategy

Specify whether you want the design to map to minimum or maximum latency with native floating-point libraries.

Settings

Default: MAX

The options are:

MIN

Maps to minimum latency for the native floating-point libraries.

MAX

Maps to maximum latency for the native floating-point libraries.

ZERO

Does not use any latency for the native floating-point libraries.

Dependency

To specify this parameter, set the **Floating Point IP Library** to Native Floating Point.

Command-Line Information

To specify the latency strategy:

- 1 Create a floating-point target configuration object for Native Floating Point as the floating-point library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
```

- 2 Specify the `LatencyStrategy` property of the `LibrarySettings` attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.LatencyStrategy = 'MIN';
```

- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)  
makehdl('sfir_single/symmetric_fir')
```

See also

- “`LatencyStrategy`” on page 21-44
- `hdlcoder.createFloatingPointTargetConfig`
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92
- “Latency Considerations with Native Floating Point” on page 10-83

Handle Denormals

Specify whether you want to handle denormal numbers in your design. Denormal numbers are nonzero numbers that are smaller than the smallest normal number.

Settings

Default: Off

On

Inserts additional logic to handle the denormal numbers in your design.

Off

Does not add additional logic to handle the denormal numbers in your design. If the input is a denormal value, HDL Coder treats the value as zero before performing any computations.

Dependency

To specify this parameter, set the **Floating Point IP Library** to `Native Floating Point`.

Command-Line Information

To specify the latency strategy:

- 1 Create a floating-point target configuration object for **Native Floating Point** as the floating-point library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
```

- 2 Specify the **HandleDenormals** property of the **LibrarySettings** attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.HandleDenormals = 'on';
```

- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

See also

- “HandleDenormals” on page 21-42
- `hdlcoder.createFloatingPointTargetConfig`
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92
- “Numeric Considerations with Native Floating-Point” on page 10-69

Mantissa Multiplier Strategy

Specify how you want HDL Coder to implement the mantissa multiplication operation when you have Product blocks in your design.

Settings

Default: Auto

The options are:

Auto

This default option automatically determines how to implement the mantissa multiplication depending on the **Synthesis tool** that you specify.

- If you do not specify a **Synthesis tool**, this setting selects the Full Multiplier implementation by default.
- If you specify Altera Quartus II as the **Synthesis tool**, this setting selects the Full Multiplier implementation.
- If you specify Xilinx Vivado or Xilinx ISE as the **Synthesis tool**, this setting selects the Part Multiplier Part AddShift implementation.

Full Multiplier

Specify this option to use only multipliers for implementing the mantissa multiplication. The multipliers can utilize DSP units on the target device.

Part Multiplier Part AddShift

Specify this option to split 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.

No Multiplier Full AddShift

Select this option to use a combination of adders and multipliers 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.

Dependency

To specify this parameter, set the **Floating Point IP Library** to Native Floating Point.

Command-Line Information

To specify the latency strategy:

- 1 Create a floating-point target configuration object for Native Floating Point as the floating-point library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
```
- 2 Specify the `MantissaMultiplyStrategy` property of the `LibrarySettings` attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.MantissaMultiplyStrategy = 'PartMultiplierPartAddShift';
```
- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

See also

- “MantissaMultiplyStrategy” on page 21-47
- `hdlcoder.createFloatingPointTargetConfig`
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92

FPGA Floating-Point Libraries

This section contains parameters in the **HDL Code Generation > Floating Point** pane of the Configuration Parameters dialog box. Use these parameters to specify the latency strategy, objective, and whether to initialize the pipeline registers in the floating-point target IP to zero.

Initialize IP Pipelines To Zero

Inserts additional logic during HDL code generation to initialize the values of pipeline registers in the Altera floating-point target IP to zero. If you do not select this check box, HDL Coder reports a warning during HDL code generation.

Settings

Default: On

On

Inserts additional logic to initialize pipeline registers in the floating-point target IP to zero.

Off

Does not add additional logic to initialize pipeline registers in the floating-point target IP to zero.

Dependency

To specify this parameter, set the **Floating Point IP Library** to Altera Megafunctions (ALTERA FP FUNCTIONS). Before you set the floating-point library, specify the path to your synthesis tool by using the `hdlsetuptoolpath` function.

Command-Line Information

To specify this setting:

- 1 Create a floating-point target configuration object with Altera Megafunctions (ALTERA FP FUNCTIONS) as the floating-point target library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('AlteraFPFunctions');
```
- 2 Specify the `InitializeIPipelinesToZero` property of the `LibrarySettings` attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.InitializeIPipelinesToZero = 0;
```

- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

See Also

- `hdlcoder.createFloatingPointTargetConfig`
- “Target Frequency” on page 13-9
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-14

Latency Strategy

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or ALTFP Altera megafunction IPs.

Settings

Default: MIN

The options are:

MIN

Maps to minimum latency for the specified floating-point target IP.

MAX

Maps to maximum latency for the specified floating-point target IP.

Dependency

To specify this parameter, set the **Floating Point IP Library** to Altera Megafunctions (ALTFP) or Xilinx LogiCORE. Before you set the floating-point library, specify the path to your synthesis tool by using the `hdlsetuptoolpath` function.

Command-Line Information

To specify this setting:

- 1 Create a floating-point target configuration object with Altera Megafunctions (ALTERA_FP_FUNCTIONS) as the floating-point target library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
```

- 2 Specify the `LatencyStrategy` property of the `LibrarySettings` attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.LatencyStrategy = 'MAX';
```

- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)  
makehdl('sfir_single/symmetric_fir')
```

See also

- `hdlcoder.createFloatingPointTargetConfig`
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-14
- “Customize Floating-Point IP Configuration” on page 31-23

Objective

Specify whether you want to optimize the design for speed or area when mapping to floating-point target libraries.

Settings

Default: SPEED

The options are:

NONE

Select this option if you do not want to optimize the design for speed or area.

SPEED

Select this option to optimize the design for speed.

AREA

Select this option to optimize the design for area.

Dependency

To specify this parameter, set the **Floating Point IP Library** to Altera Megafunctions (ALTFP) or Xilinx LogiCORE. Before you set the floating-point library, specify the path to your synthesis tool by using the `hdlsetuptoolpath` function.

Command-Line Information

To specify this setting:

- 1 Create a floating-point target configuration object with Altera Megafunctions (ALTERA FP FUNCTIONS) as the floating-point target library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
```
- 2 Specify the **Objective** property of the **LibrarySettings** attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.Objective = 'AREA';
```
- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

See also

- `hdlcoder.createFloatingPointTargetConfig`
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-14
- “Customize Floating-Point IP Configuration” on page 31-23

IP Settings

The **IP Settings** section has an IP configuration table with the IP names and data types and additional options to specify a custom latency and any extra arguments.

The options in the IP configuration table depend on the library that you specify.

- If you specify the ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS) library, HDL Coder infers the latency value from the **Target Frequency (MHz)** value.
- If you specify the ALTERA MEGAFUNCTION (ALTFP) or XILINX LOGICORE libraries, HDL Coder infers the IP latency from the **Latency Strategy** setting. The IP

configuration table has two additional columns, **MinLatency** and **MaxLatency**, that contain the minimum and maximum latency values for each IP in the table.

The IP configuration table has these sections:

- **Name:** Contains a list of IP names that HDL Coder map the Simulink blocks to, such as ABS, ADDSUB, and CONVERT.
- **DataType:** Contains a list of IP data types for each IP in the table. These are mostly SINGLE and DOUBLE data types. The CONVERT IP blocks can have DOUBLE_TO_NUMERICTYPE, NUMERICTYPE_TO_DOUBLE data types, and so on.
- **Latency:** The default latency value of -1 means that the IP inherits the latency value from the target frequency or the latency strategy setting depending on the library that you choose. To customize the latency of the IP that your Simulink blocks map to, enter your own custom value for the latency.
- **ExtraArgs:** Specify any additional settings that is specific to the IP.

For example, if you have an Add block with Single data types in your Simulink model, HDL Coder maps the block to the **ADDSUB** IP. If you want to specify a custom latency value, say 8, for the IP, enter the value in the **Latency** column for the IP.

IP Settings

Customize data type conversion IP for:

Name	DataType	MinLatency	MaxLatency	Latency	ExtraArgs
ADDSUB	DOUBLE	12	12	-1	
ADDSUB	SINGLE	12	12	8	CSET c_mult_usage=...
CONVERT	DOUBLE_TO_N...	6	6	-1	
CONVERT	NUMERICTYPE_...	6	6	-1	

cmultusage is a parameter that you can specify with the Xilinx LogiCORE libraries.

Dependency

To specify this parameter, set the **Floating Point IP Library** to Altera Megafunctions (ALTFP) or Xilinx LogiCORE. Before you set the floating-point library, specify the path to your synthesis tool by using the `hdlsetuptoolpath` function.

Command-Line Information

To specify this setting:

- 1 Create a floating-point target configuration object with Altera Megafunctions (ALTERA_FP_FUNCTIONS) as the floating-point target library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
```

- 2 To view the floating-point IP configuration, use the IPConfig object.

```
fpconfig.IPConfig
```

- 3 To customize the latency or specify additional arguments, use the customize method.

```
fpconfig.IPConfig.customize('ADDSUB','Single','Latency',6);
```

- 4 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)  
makehdl('sfir_single/symmetric_fir')
```

See also

- `hdlcoder.createFloatingPointTargetConfig`
- `customize`
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-14
- “Customize Floating-Point IP Configuration” on page 31-23

HDL Code Generation Pane: Global Settings

- “Global Settings Overview” on page 16-3
- “Clock Settings and Timing Controller Postfix” on page 16-4
- “Reset Settings” on page 16-10
- “Clock Enable Settings” on page 16-15
- “Oversampling factor” on page 16-18
- “Comment in header” on page 16-20
- “Language-Specific Identifiers and File Extensions” on page 16-22
- “Split entity and architecture” on page 16-28
- “Complex Signals Postfix” on page 16-31
- “VHDL Architecture and Library Name” on page 16-32
- “Pipeline postfix” on page 16-33
- “Generate VHDL code for model references into a single library” on page 16-35
- “Generate Statement Labels” on page 16-36
- “Vector and Component Instances Labels” on page 16-38
- “Map file postfix” on page 16-40
- “Input and Output Port Data Types” on page 16-41
- “Clock Enable output port” on page 16-43
- “Minimize Clock Enables and Reset Signals” on page 16-44
- “Use trigger signal as clock” on page 16-49
- “Enable HDL DUT port generation for test points” on page 16-51
- “RTL Annotations” on page 16-53
- “RTL Customizations for Constants and MATLAB Function Blocks” on page 16-58
- “RTL Customizations for RAMs” on page 16-60
- “No-reset registers initialization” on page 16-62

- “RTL Style” on page 16-65
- “Timing Controller Settings” on page 16-73
- “File Comment Customization” on page 16-75
- “Choose Coding Standard and Report Options” on page 16-77
- “Basic Coding Practices” on page 16-80
- “RTL Description Rules for clock enables and resets” on page 16-86
- “RTL Description Rules for Conditionals” on page 16-90
- “Other RTL Description Rules” on page 16-94
- “RTL Design Rules” on page 16-98
- “Model Generation for HDL Code” on page 16-101
- “Naming Options” on page 16-104
- “Layout Options” on page 16-107
- “Diagnostics for Optimizations” on page 16-110
- “Diagnostics for Reals and Black Box Interfaces” on page 16-114
- “Code Generation Output” on page 16-117

Global Settings Overview

The **Global Settings** pane enables you to specify detailed characteristics of the generated code, such as HDL element naming, and whether to map to a floating-point IP library.

Clock Settings and Timing Controller Postfix

This section contains parameters in the **Clock settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to specify the clock signal name, the number of clock inputs, the active clock edge, and the postfix for the clock process and the timing controller.

Clock input port

Specify the name for the clock input port in generated HDL code.

Settings

Default: clk

Enter the clock signal name in generated HDL code as a character vector.

For a generated entity `my_filter`, if you specify `'filter_clock'` as the clock signal name, the entity declaration is as shown in this code snippet:

```
ENTITY my_filter IS
  PORT( filter_clock : IN std_logic;
        clk_enable   : IN std_logic;
        reset        : IN std_logic;
        my_filter_in : IN std_logic_vector (15 DOWNT0 0); -- sfix16_En15
        my_filter_out: OUT std_logic_vector (15 DOWNT0 0); -- sfix16_En15
  );
END my_filter;
```

If you specify a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`.

Command-Line Information

Property: ClockInputPort

Type: character vector

Value: A valid identifier in the target language

Default: 'clk'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockInputPort','system_clk')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','ClockInputPort','system_clk')
```

See Also

`makehdl`

Clock inputs

Specify generation of single or multiple clock inputs.

Settings

Default: Single

Single

Generates a single clock input for the DUT. If the DUT is multirate, the input clock is the master clock rate, and a timing controller is synthesized to generate additional clocks as required. It is recommended that you use a single clock signal in your design.

Multiple

Generates a unique clock for each Simulink rate in the DUT. The number of timing controllers generated depends on the contents of the DUT. The oversample factor must be 1 (default) to specify multiple clocks.

Command-Line Information

Property: ClockInputs

Type: character vector

Value: 'Single' | 'Multiple'

Default: 'Single'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockInputs','Multiple')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','ClockInputs','Multiple')
```

See Also

- `makehdl`
- “Check clock settings” on page 38-44

Clock edge

Specify the active clock edge that triggers Verilog always blocks or VHDL process blocks in the generated HDL code.

Settings

Default: Rising.

Rising

The rising edge, or 0-to-1 transition, is the active clock edge.

Falling

The falling edge, or 1-to-0 transition, is the active clock edge.

Command-Line Information

Property: ClockEdge

Type: character vector

Value: 'Rising' | 'Falling'

Default: 'Rising'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockEdge','Falling')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','ClockEdge','Falling')
```

See Also

- `makehdl`
- “Check clock settings” on page 38-44

Clocked process postfix

Specify the postfix as a character vector. The code generator appends this postfix to HDL clock process names.

Settings

Default: `_process`

HDL Coder uses `process` blocks for register operations. The label for each of these blocks is derived from a register name and the postfix `_process`. For example, the code generator derives the label `delay_pipeline_process` in the following block declaration from the register name `delay_pipeline` and the default postfix `_process`.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
    .
    .
    .
```

Command-Line Information

Property: `ClockProcessPostfix`

Type: character vector

Default: `'_process'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockProcessPostfix','delay_postfix')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','ClockProcessPostfix','delay_postfix')
```

See Also

`makehdl`

Timing controller postfix

Specify the postfix as a character vector. The code generator appends this suffix to the DUT name to form the timing controller name.

Settings

Default: `'_tc'`

A timing controller file is generated if the design uses multiple rates, for example:

- When code is generated for a multirate model.
- When an area or speed optimization, or block architecture, introduces local multirate.

The timing controller name is based on the name of the DUT. For example, if the name of your DUT is `my_test`, by default, HDL Coder adds the postfix `_tc` to form the timing controller name, `my_test_tc`.

Command-Line Information

Property: `TimingControllerPostfix`

Type: character vector

Default: `'_tc'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Reset Settings

This section contains parameters in the **Clock Settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Using these parameters, you can specify the reset name, whether to use a synchronous or asynchronous reset, and whether the reset is asserted active-high or active-low.

Reset type

Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers. It is recommended that you specify the **Reset type** as **Synchronous** when you use a Xilinx device and **Asynchronous** when you use an Altera device.

Settings

Default: Asynchronous

Asynchronous

Use asynchronous reset logic. This reset logic samples the reset independent of the clock signal.

The following process block, generated by a Unit Delay block, illustrates the use of asynchronous resets. When the reset signal is asserted, the process block performs a reset, without checking for a clock event.

```
Unit_Delay1_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay1_out1 <= (OTHERS => '0');
  ELSIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      Unit_Delay1_out1 <= signed(x_in);
    END IF;
  END IF;
END PROCESS Unit_Delay1_process;
```

Synchronous

Use synchronous reset logic. This reset logic samples the reset with respect to the clock signal.

The following process block, generated by a Unit Delay block, checks for a clock event, the rising edge, before performing a reset:

```

Unit_Delay1_process : PROCESS (clk)
BEGIN
  IF rising_edge(clk) THEN
    IF reset = '1' THEN
      Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF clk_enable = '1' THEN
      Unit_Delay1_out1 <= signed(x_in);
    END IF;
  END IF;
END PROCESS Unit_Delay1_process;

```

Command-Line Information

Property: ResetType

Type: character vector

Value: 'async' | 'sync'

Default: 'async'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify `sync` as the `ResetType` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```

makehdl('sfir_fixed/symmetric_fir', ...
        'ResetType','async')

```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```

hdlset_param('sfir_fixed','ResetType','async')
makehdl('sfir_fixed/symmetric_fir')

```

See Also

- `makehdl`
- “Check for global reset setting for Xilinx and Altera devices” on page 38-8

Reset asserted level

Specify whether the asserted or active level of the reset input signal is active-high or active-low.

Settings

Default: Active-high

Active-high

Specify that the asserted level of reset input signal is active-high. For example, the following code fragment checks whether `reset` is active high before populating the `delay_pipeline` register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
    .
    .
    .
```

Active-low

Specify that the asserted level of reset input signal is active-low. For example, the following code fragment checks whether `reset` is active low before populating the `delay_pipeline` register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
    IF reset = '0' THEN
        delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
    .
    .
    .
```

Dependency

If you input a logic high value to the **Reset input port**, to reset the registers in your design, set **Reset asserted level** to **Active-high**. If you input a logic low value to the **Reset input port**, to reset the registers in your design, set **Reset asserted level** to **Active-low**.

Command-Line Information

Property: ResetAssertedLevel

Type: character vector

Value: 'active-high' | 'active-low'

Default: 'active-high'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'ResetAssertedLevel', 'active-high')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', 'ResetAssertedLevel', 'active-high')
```

See Also

- `makehdl`
- “Reset input port” on page 16-13

Reset input port

Enter the name for the reset input port in generated HDL code.

Settings

Default: reset

Enter a character vector for the reset input port name in generated HDL code.

For example, if you override the default with `'chip_reset'` for the generating system `myfilter`, the generated entity declaration might look as follows:

```
ENTITY myfilter IS
  PORT( clk           : IN  std_logic;
        clk_enable   : IN  std_logic;
        chip_reset    : IN  std_logic;
        myfilter_in   : IN  std_logic_vector (15 DOWNTO 0);
        myfilter_out  : OUT std_logic_vector (15 DOWNTO 0);
        );
END myfilter;
```

If you specify a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`.

Dependency

If you specify active-high for **Reset asserted level**, the reset input signal is asserted active-high. To reset the registers in the entity, the input value to the **Reset input port** must be high. If you specify active-low for **Reset asserted level**, the reset input signal is asserted active-low. To reset the registers in the entity, the input value to the **Reset input port** must be low.

Command-Line Information

Property: ResetInputPort

Type: character vector

Value: A valid identifier in the target language

Default: 'reset'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify `sync` as the `ResetType` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'ResetInputPort','rstx')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ResetInputPort','rstx')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Reset asserted level” on page 16-11

Clock Enable Settings

This section contains parameters in the **Clock Settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Using these parameters, you can specify the name of the clock enable input port and for internal clock enable signals in the generated code.

Clock enable input port

Specify the name for the clock enable input port in generated HDL code.

Settings

Default: `clk_enable`

Enter the clock enable input port name in generated HDL code as a character vector.

For example, if you specify `'filter_clock_enable'` for the generating subsystem `filter_subsys`, the generated entity declaration might look as follows:

```
ENTITY filter_subsys IS
  PORT( clk           : IN  std_logic;
        filter_clock_enable : IN  std_logic;
        reset         : IN  std_logic;
        filter_subsys_in  : IN  std_logic_vector (15 DOWNTO 0);
        filter_subsys_out : OUT std_logic_vector (15 DOWNTO 0);
  );
END filter_subsys;
```

The clock enable input signal is asserted active-high (1). Thus, the input value must be high for the generated entity's registers to be updated.

If you specify a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`.

Command-Line Information

Property: `ClockEnableInputPort`

Type: character vector

Value: A valid identifier in the target language

Default: `'clk_enable'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'ClockEnableInputPort','clken')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockEnableInputPort','clken')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Clock Enable output port” on page 16-43
- “Clock input port” on page 16-4
- “Reset input port” on page 16-13

Enable prefix

Specify the base name as a character vector for internal clock enables and other flow control signals in generated code.

Settings

Default: 'enb'

Where only a single clock enable is generated, **Enable prefix** specifies the signal name for the internal clock enable signal.

In some cases, the code generator can generate multiple clock enable signals. For example, if you specify a cascade block implementation for certain blocks, multiple clock enable signals are generated. In such cases, **Enable prefix** specifies a base signal name for the first clock enable that is generated. For other clock enable signals, numeric tags are appended to **Enable prefix** to form unique signal names. For example, the following code fragment illustrates two clock enables that were generated when **Enable prefix** was set to 'test_clk_enable':

```
COMPONENT mysys_tc  
    PORT( clk                : IN    std_logic;
```

```

        reset            :   IN    std_logic;
        clk_enable       :   IN    std_logic;
        test_clk_enable  :   OUT   std_logic;
        test_clk_enable_5_1_0 : OUT   std_logic
    );
END COMPONENT;
```

Command-Line Information

Property: EnablePrefix

Type: character vector

Default: 'enb'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'EnablePrefix','int_enable')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','EnablePrefix','int_enable')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Clock Enable output port” on page 16-43
- “Clock enable input port” on page 16-15

Oversampling factor

This parameter resides in the **Clock Settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use this parameter to specify the frequency of the global oversampling clock as a multiple of the model's base rate.

Settings

Default: 1.

Oversampling factor specifies the factor by which the global clock signal is a multiple of the base rate at which the model operates. Use the **Oversampling factor** to integrate the DUT with a larger system that supplies timing signals to other components in the system at the global oversampling clock.

By default, HDL Coder does not generate a global oversampling clock. To generate a global oversampling clock, specify an integer greater than one. If you use a multirate DUT, make sure that other rates in the DUT divide evenly into the global oversampling rate.

Generation of the global oversampling clock affects the generated HDL code and does not affect the simulation behavior of your model.

Dependency

- if you use multiple clocks, the **Oversampling factor** must be set to one. If you want to use an **Oversampling factor** greater than one, set **ClockInputs** to Single.
- If you specify an **Oversampling factor** greater than one, make sure that the clock-rate pipelining optimization is enabled. You can specify this setting in the **HDL Code Generation > Target and Optimizations > Pipelining** tab.

Clock-rate pipelining uses the **Oversampling factor** to convert the slow regions in your model that operate at the base sample rate to the faster clock rate.

Command-Line Information

Property: Oversampling

Type: int

Value: integer greater than or equal to 1

Default: 1

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'Oversampling',5)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','Oversampling',5)  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Generate a Global Oversampling Clock” on page 22-9
- “Clock Rate Pipelining” on page 14-15

Comment in header

This parameter resides in the **General** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use this parameter to specify comment lines in header of generated HDL and test bench files.

Settings

Default: None

Text entered in this field generates a comment line in the header of generated model and test bench files. The code generator adds leading comment characters for the target language. When newlines or linefeeds are included, the code generator emits single-line comments for each newline.

For example, if you specify this comment 'This is a comment line.\nThis is a second line.' for the `symmetric_fir` subsystem inside the `sfir_fixed` model and generate HDL code, the resulting header comment block appears as follows:

```
-----  
--  
-- Module: symmetric_fir  
-- Simulink Path: sfir_fixed/symmetric_fir  
-- Created: 2006-11-20 15:55:25  
-- Hierarchy Level: 0  
--  
-- This is a comment line.  
-- This is a second line.  
--  
-- Simulink model description for sfir_fixed:  
-- This model shows how to use HDL Coder to check, generate,  
-- and verify HDL for a fixed-point symmetric FIR filter.  
--  
-----
```

Command-Line Information

Property: UserComment

Type: character vector

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'UserComment', 'This is a comment line.\nThis is a second line.')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', ...  
            'UserComment', 'This is a comment line.\nThis is a second line.')
```

```
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “File Comment Customization” on page 16-75
- “Generate Code with Annotations or Comments” on page 25-16

Language-Specific Identifiers and File Extensions

This section contains parameters in the **Clock Settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Using these parameters, you can specify the Verilog and VHDL file extensions, entity, module, and package name postfix, and the prefix for module names.

Verilog file extension

Specify the file name extension for generated Verilog files.

Settings

Default: .v

This field specifies the file name extension for generated Verilog files.

Dependency

To enable this option, set the target language to Verilog. You can specify the target language by using the **Language** parameter in the **HDL Code Generation** pane.

Command-Line Information

Property: VerilogFileExtension

Type: character vector

Default: '.v'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'VerilogFileExtension','.v')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','VerilogFileExtension','.v')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Target” on page 12-3

VHDL file extension

Specify the file name extension for generated VHDL files.

Settings

Default: `.vhd`

This field specifies the file name extension for generated VHDL files.

Dependency

To enable this option, set the target language to VHDL. You can specify the target language by using the **Language** parameter in the **HDL Code Generation** pane.

Command-Line Information

Property: `VHDLFileExtension`

Type: character vector

Default: `' .vhd'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'VHDLFileExtension', '.vhd')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'VHDLFileExtension', '.vhd')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- `makehdl`
- “Target” on page 12-3

Entity conflict postfix

Specify the text as a character vector to resolve duplicate VHDL entity or Verilog module names in generated code.

Settings

Default: `_block`

The specified postfix resolves duplicate VHDL entity or Verilog module names.

For example, if HDL Coder detects two entities with the name `MyFilter`, the coder names the first entity `MyFilter` and the second entity `MyFilter_block`.

Command-Line Information

Property: `EntityConflictPostfix`

Type: character vector

Value: A valid character vector in the target language

Default: `'_block'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'EntityConflictPostfix','_entity')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','EntityConflictPostfix','_entity')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

`makehdl`

Package postfix

Specify a text as a character vector to append to the model or subsystem name to form name of a package file.

Settings

Default: `_pkg`

HDL Coder applies this option only if a package file is required for the design.

Dependency

This option is enabled when:

The target language (specified by the **Language** option) is VHDL.

The target language (specified by the **Language** option) is Verilog, and the **Multi-file test bench** option is selected.

Command-Line Information

Property: `PackagePostfix`

Type: character vector

Value: A character vector that is legal in a VHDL package file name

Default: `'_pkg'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'PackagePostfix','_pkg')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','PackagePostfix','_pkg')
makehdl('sfir_fixed/symmetric_fir')
```

Reserved word postfix

Specify a text as a character vector to append to value names, postfix values, or labels that are VHDL or Verilog reserved words.

Settings

Default: `_rsvd`

The reserved word postfix is applied to identifiers (for entities, signals, constants, or other model elements) that conflict with VHDL or Verilog reserved words. For example, if your generating model contains a signal named `mod`, HDL Coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

Command-Line Information

Property: `ReservedWordPostfix`

Type: character vector

Default: `'_rsvd'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'ReservedWordPostfix','_reserved')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ReservedWordPostfix','_reserved')  
makehdl('sfir_fixed/symmetric_fir')
```

Module name prefix

Specify a prefix for every module or entity name in the generated HDL code.

Settings

Default: `''`

Specify a prefix for every module or entity name in the generated HDL code. HDL Coder also applies this prefix to generated script file names.

You can specify the module name prefix to avoid name collisions if you plan to instantiate the generated HDL code multiple times in a larger system.

Command-Line Information

Property: ModulePrefix

Type: character vector

Default: ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Suppose you have a DUT, `myDut`, containing an internal module, `myUnit`. You can prefix the modules within your design with `unit1_` by using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('myDUT', ...  
        'ModulePrefix', 'unit1_')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('myUnit/myDUT', 'ModulePrefix', 'unit1_')  
makehdl('myDUT')
```

In the generated code, your HDL module names are `unit1_myDut` and `unit1_myUnit`, with corresponding HDL file names. Generated script file names also have the `unit1_` prefix.

Split entity and architecture

These settings correspond to the parameters in the **HDL Code Generation > Global Settings > General** tab of the Configuration Parameters dialog box.

Split entity file postfix

Enter a character vector to be appended to the model name to form the name of a generated VHDL entity file.

You can specify an empty character vector for either the **Split entity file postfix** or the **Split arch file postfix**. Both VHDL entity and architecture files cannot have empty postfix values. When you specify both values, make sure that you use different values for the **Split entity file postfix** and the **Split arch file postfix**.

If you input special characters for **Split entity file postfix**, the code generator changes the entity name to a valid HDL name before generating code.

Settings

Default: `_entity`

Dependency

This parameter is enabled by selecting the **Split entity and architecture** check box. When you select this check box, HDL Coder places the VHDL entity and architecture code in separate files.

Command-Line Information

Property: `SplitEntityFilePostfix`

Type: character vector

Default: `'_entity'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Split arch file postfix

Enter a character vector to be appended to the model name to form the name of a generated VHDL architecture file.

You can specify an empty character vector for either the **Split arch file postfix** or the **Split entity file postfix**. Both VHDL entity and architecture files cannot have empty postfix values. When you specify both values, make sure that you use different values for the **Split entity file postfix** and the **Split arch file postfix**.

If you input special characters for **Split arch file postfix**, the code generator changes the architecture name to a valid HDL name before generating code.

Settings

Default: `_arch`

Dependency

This parameter is enabled by selecting the **Split entity and architecture** check box. When you select this check box, HDL Coder places the VHDL entity and architecture code in separate files.

Command-Line Information

Property: `SplitArchFilePostfix`

Type: character vector

Default: `'_arch'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Split entity and architecture

Specify whether generated VHDL entity and architecture code is written to a single VHDL file or to separate files.

Settings

Default: Off

On

VHDL entity and architecture definitions are written to separate files.

Off

VHDL entity and architecture code is written to a single VHDL file.

Tips

The names of the entity and architecture files derive from the base file name (as specified by the generating model or subsystem name). By default, postfix strings identifying the file as an entity (`_entity`) or architecture (`_arch`) are appended to the base file name. You can override the default and specify your own postfix as a character vector.

For example, instead of all generated code residing in `MyFIR.vhd`, you can specify that the code reside in `MyFIR_entity.vhd` and `MyFIR_arch.vhd`.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Selecting this option enables the following parameters:

- **Split entity file postfix**
- **Split architecture file postfix**

You can specify an empty character vector for either the **Split arch file postfix** or the **Split entity file postfix**. Both VHDL entity and architecture files cannot have empty postfix values. When you specify both values, make sure that you use different values for the **Split entity file postfix** and the **Split arch file postfix**.

If you input special characters for the **Split entity file postfix** or the **Split arch file postfix**, the code generator changes the entity name or the architecture name to a valid HDL name before generating code.

Command-Line Information

Property: SplitEntityArch

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Complex Signals Postfix

Complex real part postfix

Specify the character vector to append to real part of complex signal names.

Settings

Default: `'_re'`

Enter a text to be appended to the names generated for the real part of complex signals.

Command-Line Information

Property: `ComplexRealPostfix`

Type: character vector

Default: `'_re'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Complex imaginary part postfix

Specify character vector to append to imaginary part of complex signal names.

Settings

Default: `'_im'`

Enter a character vector to be appended to the names generated for the imaginary part of complex signals.

Command-Line Information

Property: `ComplexImagPostfix`

Type: character vector

Default: `'_im'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

VHDL Architecture and Library Name

VHDL architecture name

Specify the architecture name for your DUT in the generated HDL code.

Settings

Default: 'rtl'

Specify the VHDL architecture name for your DUT in the generated HDL code as a character vector.

Command-Line Information

Property: VHDLArchitectureName

Type: character vector

Default: 'rtl'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

VHDL library name

Specify the target library name for the generated VHDL code.

Settings

Default: 'work'

Target library name for generated VHDL code.

Command-Line Information

Property: VHDLLibraryName

Type: character vector

Default: 'work'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Pipeline postfix

Specify the postfix as a character vector to append to names of input or output pipeline registers generated for pipelined block implementations.

Settings

Default: `'_pipe'`

You can specify a generation of input and/or output pipeline registers for selected blocks. The **Pipeline postfix** option defines a character vector that HDL Coder appends to names of input or output pipeline registers when generating code.

Command-Line Information

Property: PipelinePostfix

Type: character vector

Default: `'_pipe'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Suppose you specify a pipelined output implementation for a Product block in a model, as in the following code:

```
hdlset_param('sfir_fixed/symmetric_fir/Product','OutputPipeline', 2)
```

To append a postfix `'testpipe'` to the generated pipeline register names, use either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl(gcb,'PipelinePostfix','testpipe')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs,'PipelinePostfix','testpipe')
makehdl('myDUT')
```

The following excerpt from generated VHDL code shows process the `PROCESS` code, with postfixed identifiers, that implements two pipeline stages:

```
Product_outtestpipe_process : PROCESS (clk, reset)
BEGIN
```

```
IF reset = '1' THEN
  Product_outtestpipe_reg <= (OTHERS => to_signed(0, 33));
ELSIF clk'EVENT AND clk = '1' THEN
  IF enb = '1' THEN
    Product_outtestpipe_reg(0) <= Product_out1;
    Product_outtestpipe_reg(1) <= Product_outtestpipe_reg(0);
  END IF;
END IF;
END PROCESS Product_outtestpipe_process;
```

Generate VHDL code for model references into a single library

Specify whether VHDL code generated for model references is in a single library, or in separate libraries.

Settings

Default: Off

On

Generate VHDL code for model references into a single library.

Off

For each model reference, generate a separate VHDL library.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: UseSingleLibrary

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Generate Statement Labels

Block generate label

Specify postfix to block labels used for HDL GENERATE statements.

Settings

Default: '_gen'

Specify the postfix as a character vector. HDL Coder appends the postfix to block labels used for HDL GENERATE statements.

Command-Line Information

Property: BlockGenerateLabel

Type: character vector

Default: '_gen'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Output generate label

Specify postfix to output assignment block labels for VHDL GENERATE statements.

Settings

Default: 'outputgen'

Specify the postfix as a character vector. HDL Coder appends this postfix to output assignment block labels in VHDL GENERATE statements.

Command-Line Information

Property: OutputGenerateLabel

Type: character vector

Default: 'outputgen'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Instance generate label

Specify text to append to instance section labels in VHDL GENERATE statements.

Settings

Default: '_gen'

Specify the postfix as a character vector. HDL Coder appends the postfix to instance section labels in VHDL GENERATE statements.

Command-Line Information

Property: InstanceGenerateLabel

Type: character vector

Default: '_gen'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Vector and Component Instances Labels

Vector prefix

Specify prefix to vector names in generated code.

Settings

Default: 'vector_of_'

Specify the prefix as a character vector. HDL Coder appends this prefix to vector names in generated code.

Command-Line Information

Property: VectorPrefix

Type: character vector

Default: 'vector_of_'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Instance postfix

Specify postfix to generated component instance names.

Settings

Default: '' (no postfix appended)

Specify the postfix as a character vector. HDL Coder appends the postfix to component instance names in generated code.

Command-Line Information

Property: InstancePostfix

Type: character vector

Default: ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Instance prefix

Specify prefix to generated component instance names.

Settings

Default: 'u_'

Specify the prefix as a character vector. HDL Coder appends the prefix to component instance names in generated code.

Command-Line Information

Property: InstancePrefix

Type: character vector

Default: 'u_'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Map file postfix

Specify postfix appended to file name for generated mapping file.

Settings

Default: `'_map.txt'`

Specify the postfix as a character vector. HDL Coder appends the postfix to file name for generated mapping file.

For example, if the name of the device under test is `my_design`, HDL Coder adds the postfix `_map.txt` to form the name `my_design_map.txt`.

Command-Line Information

Property: `HDLMapFilePostfix`

Type: character vector

Default: `'_map.txt'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Input and Output Port Data Types

Input data type

Specify the HDL data type for the input ports of the model.

Settings

For VHDL, the options are:

Default: `std_logic_vector`

`std_logic_vector`

Specifies VHDL type `STD_LOGIC_VECTOR`.

`signed/unsigned`

Specifies VHDL type `SIGNED` or `UNSIGNED`.

For Verilog, the options are:

Default: `wire`

In generated Verilog code, the data type for all ports is `'wire'`, and cannot be modified. Therefore, **Input data type** is disabled when the target language is Verilog.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: `InputType`

Type: character vector

Value: (for VHDL) `'std_logic_vector'` | `'signed/unsigned'`

(for Verilog) `'wire'`

Default: (for VHDL) `'std_logic_vector'`

(for Verilog) `'wire'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Output data type

Specify the HDL data type for the output ports of the model.

Settings

For VHDL, the options are:

Default: Same as input data type

Same as input data type

Specifies that output ports of the model have the same type specified by **Input data type**.

`std_logic_vector`

Specifies VHDL type `STD_LOGIC_VECTOR` as the data type of the output port.

`signed/unsigned`

Specifies VHDL type `SIGNED` or `UNSIGNED` as the data type of the output port.

For Verilog, the options are:

Default: `wire`

In generated Verilog code, the data type for all ports is `'wire'`, and cannot be modified. Therefore, **Output data type** is disabled when the target language is Verilog.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: `OutputType`

Type: character vector

Value: (for VHDL) `'std_logic_vector'` | `'signed/unsigned'`
(for Verilog) `'wire'`

Default: If the property is left unspecified, output ports have the same type specified by `InputType`.

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Clock Enable output port

Specify the name for the generated clock enable output port as a character vector.

Settings

Default: `ce_out`

A clock enable output is generated when the design requires one.

Command-Line Information

Property: `ClockEnableOutputPort`

Type: character vector

Default: `'ce_out'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Clock Enable Settings” on page 16-15

Minimize Clock Enables and Reset Signals

Minimize clock enables

Omit generation of clock enable logic for single-rate designs.

Settings

Default: Off

On

For single-rate models, omit generation of clock enable logic wherever possible. The following VHDL code example does not define or examine a clock enable signal. When the clock signal (clk) goes high, the current signal value is output.

```
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;
```

Off

Generate clock enable logic. The following VHDL code extract represents a register with a clock enable (enb)

```
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;
```


Exceptions

In some cases, HDL Coder emits clock enables even when **Minimize clock enables** is selected. These cases are:

- Registers inside Enabled, State-Enabled, and Triggered subsystems.
- Multirate models.
- The coder always emits clock enables for the following blocks:
 - commseqgen2/PN Sequence Generator
 - dspsigops/NCO

Note HDL support for the NCO block will be removed in a future release. Use the NCO HDL Optimized block instead.

- dsprsrcs4/Sine Wave
- hlddemolib/HDL FFT
- built-in/DiscreteFir
- dspmlti4/CIC Decimation
- dspmlti4/CIC Interpolation
- dspmlti4/FIR Decimation
- dspmlti4/FIR Interpolation
- dspadpt3/LMS Filter
- dsparch4/Biquad Filter

Note If your design uses a RAM block such as a Dual Rate Dual Port RAM with the **RAM Architecture** set to Generic RAM without Clock Enable, the code generator ignores the **Minimize clock enables** setting.

Command-Line Information

Property: MinimizeClockEnables

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to minimize Clock Enable signals when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model, use either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'MinimizeClockEnables', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'MinimizeClockEnables', 'on')  
makehdl('sfir_fixed/symmetric_fir')
```

Minimize global resets

Omit generation of reset logic in the HDL code.

Settings

Default: Off

On

When you enable this setting, the code generator tries to minimize or remove the global reset logic from the HDL code. This code snippet corresponds to the Verilog code generated for a Delay block in the Simulink model. The code snippet shows that HDL Coder removed the reset logic.

```
always @(posedge clk)  
begin : Delay_Synchronous_process  
    if (enb) begin  
        Delay_Synchronous_out1 <= DataIn;  
    end  
end
```

Off

When you disable this parameter, HDL Coder generates the global reset logic in the HDL code. This Verilog code snippet shows the reset logic generated for the Delay block.

```

always @(posedge clk or posedge reset)
begin : Delay_Synchronous_process
  if (reset == 1'b1) begin
    Delay_Synchronous_out1 <= 1'b0;
  end
  else begin
    if (enb) begin
      Delay_Synchronous_out1 <= DataIn;
    end
  end
end
end

```

Dependency

If you select **Minimize global resets**, the generated HDL code contains registers that do not have a reset port. If you do not initialize these registers, there can be potential numerical mismatches in the HDL simulation results. To avoid simulation mismatches, you can initialize the registers by using the “No-reset registers initialization” on page 16-62 setting.

By default, the **No-reset registers initialization** setting has the value `Generate initialization inside module`, which means that the code generator initializes the registers as part of the HDL code generated for the DUT. To initialize the registers with the script, set **No-reset registers initialization** to `Generate an external script`. You must use a zero initial value for the blocks in your Simulink model.

Exceptions

Sometimes, when you select **Minimize global resets**, HDL Coder generates the reset logic, if you have:

- Blocks with state that have a nonzero initial value, such as a Delay block with non-zero **Initial Condition**.
- Enumerated data types for blocks with state.
- Subsystem blocks with `BlackBox` HDL architecture where you request a reset signal.
- Multirate models with **Timing controller architecture** set to `default`.

If you set **Timing controller architecture** to `resettable`, HDL Coder generates a reset port for the timing controller. If you set **Minimize global reset signals** to `'on'`, the code generator removes this reset port.

- Truth Table
- Chart
- MATLAB Function block

Command-Line Information**Property:** MinimizeGlobalResets**Type:** character vector**Value:** 'on' | 'off'**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to minimize global reset signals when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model, use either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'MinimizeGlobalResets','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MinimizeGlobalResets','on')  
makehdl('sfir_fixed/symmetric_fir')
```

Use trigger signal as clock

This setting is a parameter in the **HDL Code Generation > Global Settings > Ports** tab of the Configuration Parameters dialog box.

Settings

Default: Off

On

For triggered subsystems, use the trigger input signal as a clock in the generated HDL 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.

Off

For triggered subsystems, do not use the trigger input signal as a clock in the generated HDL code.

Command-Line Information

Property: TriggerAsClock

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to generate HDL code that uses the trigger signal as clock for triggered subsystems within the `sfir_fixed/symmetric_fir` DUT subsystem, use either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl ('sfir_fixed/symmetric_sfir', 'TriggerAsClock', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TriggerAsClock','on')  
makehdl('sfir_fixed/symmetric_fir')
```

Enable HDL DUT port generation for test points

Enable this setting to create DUT output ports for the test point signals in the generated HDL code.

Settings

Default: Off

On

When you enable this setting, the code generator creates DUT output ports for the test point signals in the generated HDL code. You can observe the test point signals and debug your design by connecting a Scope block to the output ports corresponding to these signals.

Off

When you disable this setting, the code generator preserves the test point signals and does not create DUT output ports in the generated HDL code.

Note The code generator ignores this setting when you designate test points for states inside a Stateflow Chart.

Command-Line Information

Property: EnableTestpoints

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, after you designate signals as testpoints for the `sfir_fixed/symmetric_fir` DUT subsystem, to generate DUT output ports in the HDL code, use either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl ('sfir_fixed/symmetric_sfir', 'EnableTestpoints', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','EnableTestpoints','on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

“Model and Debug Test Point Signals with HDL Coder™” on page 10-39

RTL Annotations

Use Verilog ``timescale` directives

Specify use of compiler ``timescale` directives in generated Verilog code.

Settings

Default: On

On

Use compiler ``timescale` directives in generated Verilog code.

Off

Suppress the use of compiler ``timescale` directives in generated Verilog code.

Tip

The ``timescale` directive provides a way of specifying different delay values for multiple modules in a Verilog file. This setting does not affect the generated test bench.

Dependency

This option is enabled when the target language (specified by the **Language** option) is Verilog.

Command-Line Information

Property: UseVerilogTimescale

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Verilog timescale specification

Specify the timescale that you want to use in the generated Verilog code.

Settings

Default: ``timescale 1 ns/1 ns`

HDL Coder applies this option to the timescale directive in the generated Verilog code. You can customize the default timescale and specify a valid, compilable timescale directive. The Verilog language uses this directive to determine the time units and the precision for calculating delay values.

By default, both the time units and precision are 1ns. For example, if you customized the timescale to ``timescale 1 ns/1 ps`, a delay unit becomes 1ns and the value is precise to the nearest 1 ps.

Dependency

This option is enabled when:

- The target language (specified by the **Language** option) is Verilog.
- The **Use Verilog `timescale directives** option is enabled.

Command-Line Information

Property: Timescale

Type: character vector

Value: A character vector that is a valid timescale value

Default: ``timescale 1 ns/1 ns`

Inline VHDL configuration

Specify whether generated VHDL code includes inline configurations.

Settings

Default: On

On

Include VHDL configurations in files that instantiate a component.

Off

Suppress the generation of configurations and require user-supplied external configurations. Use this setting if you are creating your own VHDL configuration files.

Tip

HDL configurations can be either inline with the rest of the VHDL code for an entity or external in separate VHDL source files. By default, HDL Coder includes configurations for a model within the generated VHDL code. If you are creating your own VHDL configuration files, suppress the generation of inline configurations.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: InlineConfigurations

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Concatenate type safe zeros

Specify use of syntax for concatenated zeros in generated VHDL code.

Settings

Default: On

On

Use the type-safe syntax, '0' & '0', for concatenated zeros. Typically, this syntax is preferred.

Off

Use the syntax "000000..." for concatenated zeros. This syntax can be easier to read and more compact, but it can lead to ambiguous types.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information**Property:** SafeZeroConcat**Type:** character vector**Value:** 'on' | 'off'**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Emit time/date stamp in header

Specify whether or not to include time and date information in the generated HDL file header.

Settings**Default:** On On

Include time/date stamp in the generated HDL file header.

```
-- -----  
--  
-- File Name: hdlsrc\symmetric_fir.vhd  
-- Created: 2011-02-14 07:21:36  
--
```

 Off

Omit time/date stamp in the generated HDL file header.

```
-- -----  
--  
-- File Name: hdlsrc\symmetric_fir.vhd  
--
```

By omitting the time/date stamp in the file header, you can more easily determine if two HDL files contain identical code. You can also avoid redundant revisions of the same file when checking in HDL files to a source code management (SCM) system.

Command-Line Information**Property:** DateComment**Type:** character vector**Value:** 'on' | 'off'**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Include requirements in block comments

Enable or disable generation of requirements comments as comments in code or code generation reports.

Settings**Default:** On On

If the model contains requirements comments, include them as comments in code or code generation reports. See “Requirements Comments and Hyperlinks” on page 25-17.

 Off

Do not include requirements as comments in code or code generation reports.

Command-Line Information**Property:** RequirementComments**Type:** character vector**Value:** 'on' | 'off'**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

RTL Customizations for Constants and MATLAB Function Blocks

Inline MATLAB Function block code

Inline HDL code for MATLAB Function blocks.

Settings

Default: Off

On

Inline HDL code for MATLAB Function blocks to avoid instantiation of code for custom blocks.

Off

Instantiate HDL code for MATLAB Function blocks and do not inline.

Command-Line Information

Property: InlineMATLABBlockCode

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to enable inlining of the code:

```
mdl = 'my_custom_block_model';  
hdlset_param(mdl, 'InlineMATLABBlockCode', 'on');
```

For example, to enable instantiation of HDL code:

```
mdl = 'my_custom_block_model';  
hdlset_param(mdl, 'InlineMATLABBlockCode', 'off');
```

Represent constant values by aggregates

Specify whether constants in VHDL code are represented by aggregates, including constants that are less than 32 bits.

Settings

Default: Off

On

HDL Coder represents constants as aggregates. The following VHDL constant declarations show a scalar less than 32 bits represented as an aggregate:

```
GainFactor_gainparam <= (14 => '1', OTHERS => '0');
```

Off

The coder represents constants less than 32 bits as scalars and constants greater than or equal to 32 bits as aggregates. The following VHDL code was generated by default for a value less than 32 bits:

```
GainFactor_gainparam <= to_signed(16384, 16);
```

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: UseAggregatesForConst

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

RTL Customizations for RAMs

Initialize all RAM blocks

Enable or suppress generation of initial signal value for RAM blocks. If you specify a nonzero initial value for the RAM, this setting is ignored.

Settings

Default: On

On

For RAM blocks, generate initial values of '0' for both the RAM signal and the output temporary signal.

Off

For RAM blocks, do not generate initial values for either the RAM signal or the output temporary signal.

Tip

This parameter applies to these RAM blocks in the **HDL Coder > HDL RAMs** Block Library in the Simulink Library Browser:

- Dual Port RAM
- Simple Dual Port RAM
- Single Port RAM
- Dual Rate Dual Port RAM

Command-Line Information

Property: InitializeBlockRAM

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

RAM Architecture

Select RAM architecture with clock enable, or without clock enable, for all RAMs in DUT subsystem.

Settings

Default: RAM with clock enable

Select one of the following options from the menu:

- RAM with clock enable: Generate RAMs with clock enable.
- Generic RAM without clock enable: Generate RAMs without clock enable.

Command-Line Information

Property: RAMArchitecture

Type: character vector

Value: 'WithClockEnable' | 'WithoutClockEnable'

Default: 'WithClockEnable'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

No-reset registers initialization

Specify whether you want to initialize registers without reset and the mode of initialization.

Settings

Default: Generate initialization inside module

The options are:

Do not initialize

HDL Coder does not initialize the registers without a reset port.

Generate an external script

HDL Coder generates a script to initialize registers that do not have a reset port in the generated code.

Generate initialization inside module

HDL Coder initializes the registers that do not have a reset port as part of the HDL code generated for the DUT. In Verilog, an `initial` construct in the corresponding module definition initializes the no-reset registers. In VHDL, the initialization code is part of the signal declaration statements.

Usage Notes

If you have blocks with **ResetType** on page 21-26 set to `none` in your Simulink model or specify the adaptive pipelining optimization, the generated HDL code can contain registers without a reset port. If you do not initialize these registers, there can be potential numerical mismatches in the HDL simulation results, because the registers are insensitive to the global reset logic. To avoid simulation mismatches, use this setting to initialize these registers in the generated code. For better simulation results, if you have registers without a reset port at the boundaries of the DUT, select **Initialize test bench inputs** in the **Test Bench** pane. Setting this property provides an initial value for the data driven to the DUT, and initializes the registers with these values.

Functionality	Script	None value	InsideModule
Generated HDL code for DUT	The script is generated externally and does not affect the HDL code for the DUT.	HDL Coder does not initialize the registers in the generated code.	The code for initializing the registers is part of the HDL code for the DUT.
HDL simulator support	The syntax of the script is compliant with ModelSim® 10.2c or later. Other HDL simulators or older ModelSim versions do not support the syntax of the initialization script. This mode does not support enumeration types, and initializing the registers with non zero values.	There can be numerical mismatches in the HDL simulation results, because this mode does not initialize the registers that do not have a reset port.	All HDL simulators support this initialization mode, and initialize the no-reset registers with appropriate values.
Synthesis tool support	As the script does not affect the HDL code generated for the DUT, all synthesis tools support this initialization mode.	Synthesis tools do not initialize the no-reset registers in this mode.	Later versions of synthesis tools support the initialization constructs in the generated code. However, it is possible that older versions do not synthesize the initialization constructs. To avoid such issues, make sure that synthesis tools can synthesize the generated code.

Command-Line Information

Property: NoResetInitializationMode

Type: character vector

Value: 'InsideModule' | 'None' 'Script'

Default: 'InsideModule'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Minimize global resets” on page 16-46

RTL Style

Use “rising_edge/falling_edge” style for registers

Specify whether generated code uses the VHDL `rising_edge` or `falling_edge` function to detect clock transitions.

Settings

Default: Off

On

Generated code uses the VHDL `rising_edge` or `falling_edge` function.

For example, the following code, generated from a Unit Delay block, uses `rising_edge` to detect positive clock transitions:

```
Unit_Delay1_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay1_out1 <= (OTHERS => '0');
  ELSIF rising_edge(clk) THEN
    IF clk_enable = '1' THEN
      Unit_Delay1_out1 <= signed(x_in);
    END IF;
  END IF;
END PROCESS Unit_Delay1_process;
```

Off

Generated code uses the `'event` syntax.

For example, the following code, generated from a Unit Delay block, uses `clk'event AND clk = '1'` to detect positive clock transitions:

```
Unit_Delay1_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay1_out1 <= (OTHERS => '0');
  ELSIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      Unit_Delay1_out1 <= signed(x_in);
    END IF;
  END IF;
END PROCESS Unit_Delay1_process;
```

Dependency

This option is enabled when the target language is VHDL.

Command-Line Information

Property: UseRisingEdge

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Minimize intermediate signals

Specify whether to optimize HDL code for debuggability or code coverage.

Settings

Default: Off

On

Optimize for code coverage by minimizing intermediate signals. For example, suppose that the generated code with this setting *off* is:

```
const3 <= to_signed(24, 7);
subtractor_sub_cast <= resize(const3, 8);
subtractor_sub_cast_1 <= resize(delayout, 8);
subtractor_sub_temp <= subtractor_sub_cast - subtractor_sub_cast_1;
```

With this setting *on*, HDL Coder optimizes the output to:

```
subtractor_sub_temp <= 24 - (resize(delayout, 8));
```

The code generator removes the intermediate signals `const3`, `subtractor_sub_cast`, and `subtractor_sub_cast_1`.

Off

Optimize for debuggability by preserving intermediate signals.

Command-Line Information

Property: MinimizeIntermediateSignals

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Scalarize vector ports

Flatten vector ports into a structure of scalar ports in VHDL code

Settings

Default: Off

On

When generating code for a vector port, generate a structure of scalar ports.

Off

When generating code for a vector port, generate a type definition and port declaration for the vector port.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: `ScalarizePorts`

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Usage Notes

The `ScalarizePorts` property lets you control how HDL Coder generates VHDL code for vector ports.

For example, consider the subsystem `vsum` in the following figure.



By default, `ScalarizePorts` is 'off'. The coder generates a type definition and port declaration for the vector port `In1` like the following:

```

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;
  
```

Under VHDL typing rules two types declared in this manner are not compatible across design units. This may cause problems if you need to interface two or more generated VHDL code modules.

You can flatten such a vector port into a structure of scalar ports by enabling `ScalarizePorts` in your `makehdl` command, as in the following example.

```
makehdl(gcs, 'ScalarizePorts', 'on')
```

The listing below shows the generated ports.

```

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;
  
```



```
);  
END vsum;
```

Unroll for Generate Loops in VHDL code

Specify whether VHDL FOR and GENERATE loops are unrolled and omitted from generated VHDL code.

Settings

Default: Off

On

Unroll and omit FOR and GENERATE loops from the generated VHDL code. (In Verilog code, loops are always unrolled.)

Off

Include FOR and GENERATE loops in the generated VHDL code.

Tip

- If you are using an electronic design automation (EDA) tool that does not support GENERATE loops, select this option to omit loops from your generated VHDL code.
- Setting this option does not affect results obtained from simulation or synthesis of generated VHDL code.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: LoopUnrolling

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Generate parameterized HDL code from masked subsystem

Generate reusable HDL code for subsystems with the same tunable mask parameters, but with different values.

Settings

Default: Off

On

Generate one reusable HDL file for multiple masked subsystems with different values for the mask parameters. HDL Coder automatically detects subsystems with tunable mask parameters that are sharable.

Inside the subsystem, you can use the mask parameter only in the following blocks and parameters.

Block	Parameter	Limitation
Constant	Constant value on the Main tab of the dialog box	None
Gain	Gain on the Main tab of the dialog box	Parameter data type must be the same for all Gain blocks.

Off

Generate a separate HDL file for each masked subsystem.

Command-Line Information

Property: MaskParameterAsGeneric

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate Reusable Code for Atomic Subsystems” on page 27-20

Enumerated Type Encoding Scheme

Specify the encoding scheme to represent enumeration types in the generated HDL code.

Settings

Default: default

Use `default`, `onehot`, `twohot`, or `binary` encoding scheme to represent enumerated types in the generated HDL code.

default

The code generator uses decimal encoding in Verilog and VHDL-native enumerated types in VHDL. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 2'd0,
is_Chart_IN_s_rx = 2'd1,
is_Chart_IN_s_wait_0 = 2'd2,
is_Chart_IN_s_wait_tb = 2'd3;
```

onehot

The code generator uses a one-hot encoding scheme where a single bit is high to represent each enumeration value. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 4'b0001,
is_Chart_IN_s_rx = 4'b0010,
is_Chart_IN_s_wait_0 = 4'b0100,
is_Chart_IN_s_wait_tb = 4'b1000;
```

This encoding scheme does not support more than 64 enumeration values or number of states.

twohot

The code generator uses a two-hot encoding scheme where two bits are high to represent each enumeration value. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 4'b0011,
```

```
is_Chart_IN_s_rx = 4'b0101,  
is_Chart_IN_s_wait_0 = 4'b0110,  
is_Chart_IN_s_wait_tb = 4'b1001;
```

binary

The code generator uses a binary encoding scheme to represent each enumeration value. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter  
is_Chart_IN_s_idle = 2'b00,  
is_Chart_IN_s_rx = 2'b01,  
is_Chart_IN_s_wait_0 = 2'b10,  
is_Chart_IN_s_wait_tb = 2'b11;
```

In VHDL, the generated code uses `CONSTANT` types to encode nondefault enumeration values in the generated code. For example, this code snippet shows the generated VHDL code when you use the two-hot state encoding for a Stateflow Chart that has four states.

```
PACKAGE s_pkg IS  
  -- Constants  
  -- Two-hot encoded enumeration values for type state_type_is_Chart  
  CONSTANT IN_s_idle      : std_logic_vector(3 DOWNTO 0) :=  
    "0011";  
  CONSTANT IN_s_rx       : std_logic_vector(3 DOWNTO 0) :=  
    "0101";  
  CONSTANT IN_s_wait_0   : std_logic_vector(3 DOWNTO 0) :=  
    "0110";  
  CONSTANT IN_s_wait_tb  : std_logic_vector(3 DOWNTO 0) :=  
    "1001";  
  
END s_pkg;
```

Command-Line Information

Property: EnumEncodingScheme

Type: character vector

Value: 'default' | 'onehot' | 'twohot' | 'binary'

Default: 'default'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Timing Controller Settings

Optimize timing controller

Optimize timing controller entity for speed and code size by implementing separate counters per rate.

Settings

Default: On

On

HDL Coder generates multiple counters (one counter for each rate in the model) in the timing controller code. The benefit of this optimization is that it generates faster logic, and the size of the generated code is usually much smaller.

Off

The coder generates a timing controller that uses one counter to generate all rates in the model.

Tip

A timing controller code file is generated if required by the design, for example:

- When code is generated for a multirate model
- When a cascade block implementation for certain blocks is specified

This file contains a module defining timing signals (clock, reset, external clock enable inputs and clock enable output) in a separate entity or module. In a multirate model, the timing controller entity generates the required rates from a single master clock using one or more counters and multiple clock enables.

The timing controller name derives from the name of the subsystem that is selected for code generation (the DUT), and the current value of the property `TimingControllerPostfix`. For example, if the name of your DUT is `my_test`, in the default case the coder adds the `TimingControllerPostfix_tc` to form the timing controller name `my_test_tc`.

Command-Line Information**Property:** OptimizeTimingController**Type:** character vector**Value:** 'on' | 'off'**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Timing controller architecture

Specify whether to generate a reset for the timing controller.

Settings**Default:** default

resettable

Generate a reset for the timing controller. If you select this option, the **Clock inputs** value must be Single.

default

Do not generate a reset for the timing controller.

Command-Line Information**Property:** TimingControllerArch**Type:** character vector**Value:** 'resettable' | 'default'**Default:** 'default'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

File Comment Customization

Custom File Header Comment

Specify a custom file header comment in the generated HDL code.

Default: ''

With **Custom File Header Comment**, you can enter custom comments to appear as header in the generated HDL file for your design.

For example, you can specify arguments such as title, author, modified date, and so on.

```
// =====
// Title           : <%Title%>
// Project         : <%Project%>
// Author          : <%Author%>
//
// Revision        : $Revision$
// Date Modified   : $Date$
// =====
```

Command-Line Information

Property: CustomFileHeaderComment

Type: character vector

Default: ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Custom File Footer Comment

Specify a custom file header comment in the generated HDL code.

Default: ''

With **Custom File Footer Comment**, you can enter custom comments to appear as footer in the generated HDL file for your design.

For example, you can specify arguments such as revision, generated log file, revision number, and so on.

```
//=====
//  xxxxxx
//=====
//  $Log$
//  Revision 1.2  2009/12/14 04:38:51  sxxxxxx
//  Initial revision
//
//=====
```

Command-Line Information**Property:** CustomFileFooterComment**Type:** character vector**Default:** ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Choose Coding Standard and Report Options

This section contains parameters in the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to generate HDL code that adheres to the guidelines recommended by Industry coding standards.

HDL coding standard

Specify whether to enable the Industry coding standard guidelines that the generated HDL code must conform to.

Settings

Default: None

None

Generate generic synthesizable HDL code. The generated code need not conform with the Industry standard guidelines.

Industry

Generate synthesizable HDL code that follows the industry standard rules supported by HDL Coder. When you specify the **Industry** setting, the code generator enables the **Report options** check box and rules that you can customize in the **Coding Standards** tab.

When you specify the **Industry** setting and generate code, HDL Coder generates a standards compliance report. The report displays errors, warnings, messages, and lists the corresponding rules. To filter the report such that the passing rules do not appear, clear the **Report options** check box.

Command-Line Information

Property: HDLCodingStandard

Type: character vector

Value: 'None' | 'Industry'

Default: 'None'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the Industry standard guidelines compliance for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','HDLCodingStandard','Industry')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','HDLCodingStandard','Industry')
```

See Also

- `makehdl`
- “HDL Coding Standards” on page 26-4
- HDL Coding Standard Customization Properties

Report options

Specify whether to filter the coding standard report such that the passing rules do not appear. By default, the report displays pass, errors, warnings, messages, and lists the corresponding rules.

Settings

Default: Off

On

Show only rules with errors or warnings. The code generator filters out messages and passing rules from the report.

Off

Show all rules in the report including the messages and passing rules.

Dependency

To clear the **Report options** check box, set the **HDL coding standard** parameter to `Industry`.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```
- 2 Set the `ShowPassingRules` property of the HDL coding standard customization object.

For example, to omit passing rules from the report, enter:

```
cso.ShowPassingRules.enable = false;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...  
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “HDL Coding Standards” on page 26-4
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Basic Coding Practices

These parameters belong to the **Basic coding rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to customize basic coding rules that are specified by the Industry standard guidelines. These rules correspond to naming conventions that your design uses.

Check for duplicate names

Specify whether to check for duplicate names in the design. This check corresponds to CGSL-1.A.A.5 of the Industry standard guidelines.

Settings

Default: On

On

Check for duplicate names.

Off

Do not check for duplicate names.

Dependency

To clear the **Check for duplicate names** check box, set the **HDL coding standard** parameter to `Industry`.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```
- 2 Set the `DetectDuplicateNamesCheck` property of the HDL coding standard customization object.

For example, to disable the check for duplicate names, enter:

```
cso.DetectDuplicateNamesCheck.enable = false;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...  
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “Basic Coding Practices” on page 26-10
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check for HDL keywords in design names

Specify whether to check for HDL keywords in design names. This check corresponds to CGSL-1.A.A.3 of the Industry standard guidelines.

Settings

Default: On

On

Check for HDL keywords in design names.

Off

Do not check for HDL keywords in design names.

Dependency

To clear the **Check for HDL keywords in design names** check box, set the **HDL coding standard** parameter to `Industry`.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the HDLKeywords property of the HDL coding standard customization object.

For example, to disable the check for HDL keywords in design names, enter:

```
cso.HDLKeywords.enable = false;
```

- 3 Set the HDLCodingStandardCustomizations property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...  
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “Basic Coding Practices” on page 26-10
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check module, instance, entity name length

Specify whether to check module, instance, and entity name length. This check corresponds to CGSL-1.A.C.3 of the Industry standard guidelines.

Settings

Default: On

On

Check module, instance, and entity name length.

Minimum

Minimum name length, specified as a positive integer. The default is 2.

Maximum

Maximum name length, specified as a positive integer. The default is 32.

Off

Do not check module, instance, and entity name length.

Dependency

To clear the **Check module, instance, entity name length** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the `ModuleInstanceEntityNameLength` property of the HDL coding standard customization object.

For example, to enable the check for module, instance, and entity name length, with 5 as the minimum length and 30 as the maximum length, enter:

```
cso.ModuleInstanceEntityNameLength.enable = true;
cso.ModuleInstanceEntityNameLength.length = [5 30];
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “Basic Coding Practices” on page 26-10
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check signal, port, and parameter name length

Specify whether to check signal, port, and parameter name length. This check corresponds to CGSL-1.A.B.1 of the Industry standard guidelines.

Settings

Default: On

On

Check signal, port, and parameter name length.

Minimum

Minimum name length, specified as a positive integer. The default is 2.

Maximum

Maximum name length, specified as a positive integer. The default is 40.

Off

Do not check signal, port, and parameter name length.

Dependency

To clear the **Check signal, port, and parameter name length** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the `SignalPortParamNameLength` property of the HDL coding standard customization object.

For example, to enable the check for signal, port, and parameter name length, with 5 as the minimum length and 30 as the maximum length, enter:

```
cso.SignalPortParamNameLength.enable = true;  
cso.SignalPortParamNameLength.length = [5 30];
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...  
       'HDLCodingStandardCustomizations', cso);
```


See Also

- `makehdl`
- “Basic Coding Practices” on page 26-10
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

RTL Description Rules for clock enables and resets

These parameters belong to the **RTL description rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to customize RTL description rules for clock enable and reset signals that are specified by the Industry standard guidelines.

Check for clock enable signals

Specify whether to check for clock enable signals in the generated code. This check corresponds to CGSL-2.C.C.4 of the Industry standard guidelines.

Settings

Default: Off

On

Minimize clock enables during code generation, then check for clock enable signals in the generated code.

Off

Do not check for clock enable signals in the generated code.

Dependency

To select the **Check for clock enable signals** check box, set the **HDL coding standard** parameter to Industry.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```
- 2 Set the `MinimizeClockEnableCheck` property of the HDL coding standard customization object.

For example, to minimize clock enables and check for clock enable signals in the generated code, enter:

```
cso.MinimizeClockEnableCheck.enable = true;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...
        'HDLCodingStandardCustomizations', cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-25
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Detect usage of reset signals

Specify whether to check for reset signals in the generated code. This check corresponds to CGSL-2.C.C.5 of the Industry standard guidelines.

Settings

Default: Off

On

Minimize reset signals in the generated code, then check for reset signals after code generation.

Off

Do not check for reset signals in the generated code.

Dependency

To select the **Detect usage of reset signals** check box, set the **HDL coding standard** parameter to `Industry`.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the RemoveResetCheck property of the HDL coding standard customization object.

For example, to check for reset signals, enter:

```
cso.RemoveResetCheck.enable = true;
```

- 3 Set the HDLCodingStandardCustomizations property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...  
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-25
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Detect usage of asynchronous reset signals

Specify whether to check for asynchronous reset signals in the generated code. This check corresponds to CGSL-2.C.C.6 of the Industry standard guidelines.

Settings

Default: Off

On

Check for asynchronous reset signals in the generated code.

Off

Do not check for asynchronous reset signals in the generated code.

Dependency

To clear the **Detect usage of asynchronous reset signals** check box, set the **HDL coding standard** parameter to Industry.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the `AsynchronousResetCheck` property of the HDL coding standard customization object.

For example, to minimize use of variables, enter:

```
cso.AsynchronousResetCheck.enable = true;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...  
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-25
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

RTL Description Rules for Conditionals

These parameters belong to the **RTL description rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to customize RTL description rules for conditional and if-else statements that are specified by the Industry standard guidelines.

Check for conditional statements in processes

Specify whether to check for length of conditional statements that are described separately within a process. This check corresponds to CGSL-2.F.B.1 of the Industry standard guidelines.

Settings

Default: On

On

Check for length of conditional statements in a process. The default length is 1.

Off

Do not check for length of conditional statements in a process.

Dependency

To clear the **Check for conditional statements in processes** check box, set the **HDL coding standard** parameter to `Industry`.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```
- 2 Set the `ConditionalRegionCheck` property of the HDL coding standard customization object.

For example, to check for four conditional statements in a process, enter:

```
cso.ConditionalRegionCheck.enable = true;
cso.ConditionalRegionCheck.length = 4;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...
        'HDLCodingStandardCustomizations', cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-25
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check if-else statement chain length

Specify whether to check if-else statement chain length. This check corresponds to CGSL-2.G.C.1c of the Industry standard guidelines.

Settings

Default: On

On

Check if-else statement chain length.

Length

Maximum if-else statement chain length, specified as a positive integer. The default is 7.

Off

Do not check if-else statement chain length.

Dependency

To clear the **Check if-else statement chain length** check box, set the **HDL coding standard** parameter to `Industry`.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the `IfElseChain` property of the HDL coding standard customization object.

For example, to check for if-else statement chains with length greater than 5, enter:

```
cso.IfElseChain.enable = true;  
cso.IfElseChain.length = 5;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...  
        'HDLCodingStandardCustomizations', cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-25
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check if-else statement nesting depth

Specify whether to check if-else statement nesting depth. This check corresponds to CGSL-2.G.C.1a of the Industry standard guidelines.

Settings

Default: On

On

Check if-else statement nesting depth.

Depth

Maximum if-else statement nesting depth, specified as a positive integer. The default is 3.

Off

Do not check if-else statement nesting depth.

Dependency

To clear the **Check if-else statement nesting depth** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the `IfElseNesting` property of the HDL coding standard customization object.

For example, to enable the check for if-else statement nesting depth with a maximum depth of 5, enter:

```
cso.IfElseNesting.enable = true;
cso.IfElseNesting.depth = 5;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...
        'HDLCodingStandardCustomizations', cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-25
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Other RTL Description Rules

These parameters belong to the **RTL description rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to customize RTL description rules of the Industry standard guidelines. These rules pertain to checking the multiplier width, whether to minimize use of variables, and initial statements to provide initial value for RAMs.

Minimize use of variables

Specify whether to minimize use of variables. This check corresponds to CGSL-2.G of the Industry standard guidelines.

Settings

Default: Off

On

Minimize use of variables.

Off

Do not minimize use of variables.

Dependency

To select the **Minimize use of variables** check box, set the **HDL coding standard** parameter to Industry.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the `MinimizeVariableUsage` property of the HDL coding standard customization object.

For example, to minimize use of variables, enter:

```
cso.MinimizeVariableUsage.enable = true;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-25
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check for initial statements that set RAM initial values

Specify whether to check for initial statements that set RAM initial values. This check corresponds to CGSL-2.C.D.1 of the Industry standard guidelines.

Settings

Default: On

On

Check for initial statements that set RAM initial values

Off

Do not check for initial statements that set RAM initial values.

Dependency

To clear the **Check for initial statements that set RAM initial values** check box, set the **HDL coding standard** parameter to `Industry`.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the `InitialStatements` property of the HDL coding standard customization object.

For example, to disable the check for initial statements that set RAM initial values, enter:

```
cso.InitialStatements.enable = false;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...  
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-25
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check multiplier width

Specify whether to check multiplier bit width. This check corresponds to CGSL-2.J.F.5 of the Industry standard guidelines.

Settings

Default: On

On

Check multiplier width.

Maximum

Maximum multiplier bit width, specified as a positive integer. The default is 16.

Off

Do not check multiplier width.

Dependency

To clear the **Check multiplier width** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **MultiplierBitWidth** property of the HDL coding standard customization object.

For example, to enable the check for multiplier width with a maximum bit width of 32, enter:

```
cso.MultiplierBitWidth.enable = true;  
cso.MultiplierBitWidth.width = 32;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...  
        'HDLCodingStandardCustomizations', cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-25
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

RTL Design Rules

This section contains parameters in the **RTL design rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to check for presence of non-integer constants and the line wrap length in the generated HDL code.

Check for non-integer constants

Specify whether to check for non-integer constants. This check corresponds to CGSL-3.B.D.1 of the Industry standard guidelines.

Settings

Default: On

On

Check for non-integer constants.

Off

Do not check for non-integer constants.

Dependency

To clear the **Check for non-integer constants** check box, set the **HDL coding standard** parameter to Industry.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```
- 2 Set the NonIntegerTypes property of the HDL coding standard customization object.

For example, to disable the check for non-integer constants, enter:

```
cso.NonIntegerTypes.enable = false;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...  
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Design Rules” on page 16-98
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check line length

Specify whether to check line lengths in the generated HDL code. This check corresponds to CGSL-3.A.D.5 of the Industry standard guidelines.

Settings

Default: On

On

Check line length.

Maximum

Maximum number of characters in a line, specified as a positive integer. The default is 110.

Off

Do not check line length.

Dependency

To clear the **Check line length** check box, set the **HDL coding standard** parameter to `Industry`.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the `LineLength` property of the HDL coding standard customization object.

For example, to enable the check line length with a maximum character length of 80, enter:

```
cso.HDLKeywordsLineLength.enable = true;  
cso.HDLKeywordsLineLength.length = 80;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...  
        'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Design Rules” on page 16-98
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Model Generation for HDL Code

In the Configuration Parameter dialog box, you can select the types of the model that you want to generate. Select **HDL Code Generation > Global Settings > Model Generation**.

You can customize the name of the generated model and the validation model by using “Naming Options” on page 16-104. To control the layout of the generated models, use the “Layout Options” on page 16-107.

Generated model

Enable or disable generation of the generated model that shows latency and numeric differences between your Simulink DUT and the generated HDL code. Delays that the coder inserts are highlighted in the generated model.

Note When you select **Generated model**, the **Naming options** and **Layout options** become available.

Settings

Default: On

On

Select this setting to generate the generated model. By default, HDL Coder generates code and the generated model. To generate only the generated model, clear the **Generate HDL code** check box.

Off

Clear this setting when you do not want to generate the generated model. When you click the **Generate** button, HDL Coder generates code for the model.

Command-Line Information

Property: GenerateModel

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

By default, the `GenerateHDLCode` property is enabled. You can use this property in conjunction with the `GenerateModel` property to specify whether to generate the generated model and the HDL code. To generate the code and the generated model, run `makehdl`.

```
makehdl('sfir_fixed/symmetric_fir')
```

If you want to generate only the generated model, disable the `GenerateHDLCode` property and run `makehdl`.

```
hdlset_param('sfir_fixed', 'GenerateModel','on');  
hdlset_param('sfir_fixed', 'GenerateHDLCode','off');  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Balance delays” on page 14-3
- “Generated Model and Validation Model” on page 24-5
- “Prefix for generated model name” on page 16-104

Validation model

Enable or disable generation of a validation model that verifies the functional equivalence of the original model with the generated model. The validation model contains the original and the generated DUT models. You can use the generated DUT model to observe the effect of block settings and optimizations such as resource sharing, streaming, and delay balancing.

If you enable generation of a validation model, make sure that delay balancing is enabled on the model. In the **HDL Code Generation > Optimization > General** tab, select the **Balance delays** check box. Delay balancing keeps the generated DUT model synchronized with the original DUT model. Validation fails when there is a mismatch between delays in the original DUT model and delays in the generated DUT model.

Settings

Default: Off

On

Select this setting to generate the validation model. By default, HDL Coder generates code and the validation model. To generate only the validation model, clear the **Generate HDL code** check box.

Off

Clear this setting when you do not want to generate the validation model. When you click the **Generate** button, HDL Coder generates code for the model.

Command-Line Information

Property: GenerateValidationModel

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

By default, the `GenerateHDLCode` property is enabled. You can use this property in conjunction with the `GenerateValidationModel` property to specify whether to generate the validation model and the HDL code. To generate the code and the validation model, enable the `GenerateValidationModel` property with `makehdl`.

```
hdlset_param('sfir_fixed', 'GenerateValidationModel','on');  
makehdl('sfir_fixed/symmetric_fir')
```

If you want to generate only the validation model, disable the `GenerateHDLCode` property and enable the `GenerateValidationModel` property with `makehdl`.

```
hdlset_param('sfir_fixed', 'GenerateValidationModel','on');  
hdlset_param('sfir_fixed', 'GenerateHDLCode','off');  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Balance delays” on page 14-3
- “Generated Model and Validation Model” on page 24-5
- “Suffix for validation model name” on page 16-105

Naming Options

This section contains different naming options available in **HDL Code Generation > Global Settings** pane under **Model Generation** tab. You can control the prefix for the generated model name and the suffix for the validation model name.

Prefix for generated model name

Specify the prefix to the generated model name.

Settings

Default: 'gm_'

Specify the prefix as a character vector. HDL Coder appends the prefix to name of generated model.

Command-Line Information

Property: GeneratedModelNamePrefix

Type: character vector

Default: 'gm_'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to indicate that you are using the generated model as a software interface model, you can use the prefix `sm_`. Specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model by using either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'GeneratedModelNamePrefix','sm_')
```

- When you use `hdlset_param`, set the parameter on the model, and then generate HDL code by using `makehdl`.

```
hdlset_param('sfir_fixed','GeneratedModelNamePrefix','sm_')  
makehdl('sfir_fixed/symmetric_fir')
```

Dependency

To specify **Prefix for generated model**, select **Generated model**.

See Also

- “Generated Model” on page 24-5
- “Suffix for validation model name” on page 16-105
- “Generated Model and Validation Model” on page 24-5

Suffix for validation model name

Specify the suffix for the validation model name.

Settings

Default: `'_vnl'`

Specify the suffix as a character vector. HDL Coder appends the suffix to name of validation model.

Command-Line Information

Property: `ValidationModelNameSuffix`

Type: character vector

Default: `'_vnl'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to indicate that you are using the generated model as a software interface model, you can use the suffix `_sm` for the validation model name. Specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model by using either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ValidationModelNameSuffix','_sm')
```

- When you use `hdlset_param`, set the parameter on the model, and then generate HDL code by using `makehdl`.

```
hdlset_param('sfir_fixed','ValidationModelNameSuffix','_sm')
makehdl('sfir_fixed/symmetric_fir')
```

Dependency

To specify **Suffix for validation model**, select **Generated model** and **Validation model**.

See Also

- “Validation model” on page 16-102
- “Prefix for generated model name” on page 16-104
- “Generated Model and Validation Model” on page 24-5

Layout Options

Different layout options are available in **HDL Code Generation > Global Settings** pane under the **Model Generation** tab. You can control placement of blocks, routing of signals within the models, and scaling of blocks.

Auto block placement

Specify automatic placement of blocks in the HDL model.

Settings

Default: On

Command-Line Information

Property: autoplacement

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param`:

```
hdlset_param(gcs, 'autoplacement', 'on')
```

Dependency

To select **Auto block placement**, first select **Generated model**.

See Also

“Model Generation for HDL Code” on page 16-101

Auto signal routing

Specify automatic routing of signals in the generated HDL model.

Settings

Default: On

Command-Line Information**Property:** autoroute**Type:** character vector**Value:** 'on' | 'off'**Default:** 'on'

To set this property, use `hdlset_param`:

```
hdlset_param(gcs, 'autoroute', 'on')
```

Dependency

To select **Auto signal routing**, first select **Auto block placement**.

See Also

“Model Generation for HDL Code” on page 16-101

Inter-block horizontal scaling

Scale the generated model horizontally. You can use this setting with **Inter-block vertical scaling** depending on how tightly packed or loosely packed you want the model to appear.

Settings**Default:** 1.7**Command-Line Information****Property:** InterBlkHorzScale**Type:** positive integer | positive double**Default:** 1.7

To set this property, use `hdlset_param`:

```
hdlset_param(gcs, 'InterBlkHorzScale', 1.7)
```

Dependency

To select **Inter-block horizontal scaling**, first select **Auto block placement**.

See Also

“Model Generation for HDL Code” on page 16-101

Inter-block vertical scaling

Scale the generated model vertically. You can use this setting with **Inter-block horizontal scaling** depending on how tightly packed or loosely packed you want the model to appear.

Settings

Default: 1.2

Command-Line Information

Property: InterBlkVertScale

Type: positive integer|positive double

Default: 1.2

To set this property, use `hdlset_param`:

```
hdlset_param(gcs, 'InterBlkVertScale', 1.2)
```

Dependency

To select **Inter-block vertical scaling**, first select **Auto block placement**.

See Also

“Model Generation for HDL Code” on page 16-101

Diagnostics for Optimizations

This section contains parameters in the **Diagnostics** option under **Advanced** tab in the Configuration Parameters dialog box. Select **HDL Code Generation > Global Settings**. To highlight blocks and feedback loops that inhibit delay balancing, distributed pipelining, clock-rate pipelining, and other optimizations, use these parameters.

Highlight feedback loops inhibiting delay balancing and optimizations

Feedback loops in your Simulink model can inhibit delay balancing and optimizations such as resource sharing and streaming. Use this setting to generate a script that highlights feedback loops.

When you generate the feedback loop highlighting script, HDL Coder generates another script that clears the highlighting of feedback loops in your model. To turn off highlighting, click the link to the `clearhighlighting` script.

Settings

Default: On

On

Generate a MATLAB script that highlights feedback loops in the original model and the generated model. When you run the script, the code generator highlights the feedback loops using different colors. The highlighting script is saved in the same target folder as the generated HDL code.

It is recommended that you leave this setting enabled so that you can identify the feedback loops and further optimize your design.

Off

Do not generate a script to highlight feedback loops.

Command-Line Information

Property: HighlightFeedbackLoops

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','HighlightFeedbackLoops','off')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','HighlightFeedbackLoops','off')
```

See Also

- “Find Feedback Loops” on page 24-38
- “Delay Balancing” on page 24-32
- “Generated Model and Validation Model” on page 24-5

Highlight blocks inhibiting clock-rate pipelining

Certain blocks in your Simulink model can inhibit clock-rate pipelining and therefore delimit clock-rate pipelining regions. Use this setting to generate a script to highlight the blocks.

When you generate the clock-rate pipelining highlighting script, HDL Coder generates another script that clears the highlighting. To turn off highlighting, click the link to the `clearhighlighting` script.

Settings

Default: On

On

Generate a MATLAB script that highlights blocks in the original model and the generated model that are inhibiting clock-rate pipelining.

It is recommended that you leave this setting enabled so that you can identify the blocks that delimit the clock-rate pipelining regions and further optimize your design.

Off

Do not generate a script to highlight blocks that are inhibiting clock-rate pipelining.

Command-Line Information

Property: HighlightClockRatePipeliningDiagnostic

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','HighlightClockRatePipeliningDiagnostic','off')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','HighlightClockRatePipeliningDiagnostic','off')
```

See Also

- “Clock Rate Pipelining” on page 14-15
- “Diagnostics for Reals and Black Box Interfaces” on page 16-114
- “Generated Model and Validation Model” on page 24-5

Highlight blocks inhibiting distributed pipelining

Certain blocks in your Simulink model can act as barriers for the distributed pipelining optimization. Use this setting to generate a script to highlight the blocks that are inhibiting distributed pipelining.

When you generate the highlighting script that displays distributed pipelining barriers, HDL Coder generates another script that clears the highlighting. To turn off highlighting, click the link to the `clearhighlighting` script.

Settings

Default: On

On

Generate a MATLAB script that highlights blocks that are inhibiting distributed pipelining in the original model and the generated model.

It is recommended that you leave this setting enabled so that you can identify the blocks that are barriers for distributed pipelining and further optimize your design.

Off

Do not generate a script to highlight blocks that are inhibiting distributed pipelining.

Command-Line Information

Property: DistributedPipeliningBarriers

Type: character vector

Value: 'on' | 'off'

Default: 'on'

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','DistributedPipeliningBarriers','off')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','DistributedPipeliningBarriers','off')
```

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Distributed Pipelining” on page 14-12
- “Diagnostics for Reals and Black Box Interfaces” on page 16-114
- “Generated Model and Validation Model” on page 24-5

Diagnostics for Reals and Black Box Interfaces

This section contains parameters in the **Diagnostics** option under **Advanced** tab in the Configuration Parameters dialog box. Select **HDL Code Generation > Global Settings**. To check for name conflicts in black box interfaces and for the presence of reals in the generated HDL code, use these parameters.

Check for name conflicts in black box interfaces

Specify whether to check for duplicate module or entity names in generated HDL code and black box interface HDL code.

Settings

Default: Warning

None

Do not check for black box subsystems that have the same HDL module name as a generated HDL module name.

Warning

Check for black box subsystems that have the same HDL module name as a generated HDL module name. Display a warning if matching names are found.

Error

Check for black box subsystems that have the same HDL module name as a generated HDL module name. Display an error if matching names are found.

Command-Line Information

Property: DetectBlackBoxNameCollision

Type: character vector

Value: 'None' | 'Warning' | 'Error'

Default: 'Warning'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','DetectBlackBoxNameCollision','None')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','DetectBlackBoxNameCollision','None')
```

See Also

- `makehdl`
- “Generate Black Box Interface for Subsystem” on page 27-5
- “Diagnostics for Optimizations” on page 16-110

Check for presence of reals in generated HDL code

Specify whether to check for reals in the generated HDL code.

Settings

Default: Error

None

Do not check for reals in the generated HDL code.

Warning

Checks and warns of any presence of real data types in the generated HDL code. Real data types in the generated HDL code are not synthesizable on target FPGA devices.

Error

Checks and generates an error if there are any real data types in the generated HDL code. If you are generating code for simulation purposes and not for synthesizing your design, you can change this setting to Warning or None. To generate synthesizable HDL code, set the **Floating Point IP Library** to Native Floating Point.

Command-Line Information

Property: `TreatRealsInGeneratedCodeAs`

Type: character vector

Value: 'None' | 'Warning' | 'Error'

Default: 'Error'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TreatRealsInGeneratedCodeAs','Warning')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','TreatRealsInGeneratedCodeAs','Warning')
```

See Also

- `makehdl`
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- “Diagnostics for Optimizations” on page 16-110

Code Generation Output

You can specify whether or not to generate HDL code by using the **Generate HDL code** parameter. In the Configuration Parameters dialog box, select **HDL Code Generation > Global Settings > Advanced > Code generation output**.

Generate HDL code

Enable or disable HDL code generation for the model or Subsystem. To specify the Subsystem that you want to generate HDL code for, use the **Generate HDL for** parameter. Then, click the **Generate** button in the **HDL Code Generation** pane. By default, the HDL code is generated in VHDL language and put into the `hdlsrc` folder.

Settings

Default: On

On

Select this setting to generate HDL code.

Off

When you clear this setting, you cannot generate HDL code for the model.

Command-Line Information

Property: GenerateHDLCode

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

By default, the `GenerateHDLCode` property is selected. To generate code, use the `makehdl` function. For example, this command generates HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model.

```
makehdl('sfir_fixed/symmetric_fir')
```

Control Code Generation Output

Property: CodeGenerationOutput

Type: character vector

Value: 'GenerateHDLCode' |

'GenerateHDLCodeAndDisplayGeneratedModel' 'DisplayGeneratedModelOnly'

Default: 'GenerateHDLCode'

By default, HDL Coder creates a model called the generated model when you generate HDL code. The generated model uses HDL-specific block implementations, and it implements the area and speed optimizations that you specify in your Simulink model. The code generator creates the generated model but does not display the model by default. To control display of the generated model, use the `CodeGenerationOutput` property.

This example shows how to generate HDL code, and then display the generated model by using `makehdl`.

```
makehdl('sfir_fixed/symmetric_fir', ...  
        'CodeGenerationOutput', 'GenerateHDLCodeAndDisplayGeneratedModel')
```

If you specify `DisplayGeneratedModelOnly`, the code generator displays the generated model but does not proceed to code generation.

See Also

- `makehdl`
- “Generated Model and Validation Model” on page 24-5
- “Model Generation for HDL Code” on page 16-101

HDL Code Generation Pane: Report

- “Report Overview” on page 17-2
- “Generate traceability report” on page 17-3
- “Traceability style” on page 17-5
- “Generate model Web view” on page 17-7
- “Generate resource utilization report” on page 17-9
- “Generate high-level timing critical path report” on page 17-11
- “Generate optimization report” on page 17-13

Report Overview

When you use the parameters in the **Report** pane, HDL Coder creates a Code Generation Report when generating HDL code for your model or Subsystem. The Code Generation Report contains a **Summary**, a **Code Interface Report**, and one or more of these reports.

- A traceability report that you can use to trace from the generated HDL code to the model and from the model to HDL code.
- A resource utilization report that contains the number of hardware resources used in the HDL code.
- An optimization report that displays the result of optimizations such as streaming, sharing, distributed pipelining, and floating-point target-specific information that was implemented in the generated code.
- A web view of the model that you can use to navigate between the generate code and your Simulink model.

See Also

- “Create and Use Code Generation Reports” on page 25-2
- `makehdl`

Generate traceability report

Enable or disable generation of an HTML code generation report with hyperlinks from code to model and model to code. The report provides line-level traceability for each block in your Simulink model. When you click the hyperlink beside a certain line of code in the report, HDL Coder highlights the corresponding block in your Simulink model. When you select a certain block in your model, the report highlights all lines of code corresponding to that block.

Settings

Default: Off

On

Create and display a traceability report section in the HTML code generation report. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the traceability report.

Off

Do not create an HTML code generation report.

Dependency

When you select this check box, you can select the **Traceability style**. By default, the **Traceability style** is Line Level.

Command-Line Information

Property: Traceability

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can generate a traceability report when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the Traceability property as an argument to `makehdl`.
`makehdl('sfir_fixed/symmetric_fir','Traceability','on')`
- Enable the Traceability property using `hdlset_param` and then use `makehdl`.
`hdlset_param('sfir_fixed','Traceability','on')`
`makehdl('sfir_fixed/symmetric_fir')`

You can use the `RequirementComments` property to generate hyperlinked requirements comments within the HTML code generation report. The requirements comments link to the corresponding requirements documents for your model.

See Also

- “Create and Use Code Generation Reports” on page 25-2
- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 25-5
- `makehdl`

Traceability style

You can use **Traceability style** to specify whether you want to generate line-level or comment-based hyperlinks in the traceability report.

Settings

Default: Line Level

The options are:

Line Level

By default, HDL Coder generates a line-level traceability report that contains hyperlinks from each line of HDL code to the corresponding block in your Simulink model. The traceability report that is generated by using this style does not contain hyperlinked comments above the HDL code corresponding to a certain block. When you select a certain block and navigate to the HDL code, the code generator highlights all lines of code corresponding to that block.

Comment Based

If you specify generation of a comment-based traceability report, the report contains hyperlinked comments above a block of HDL code. The comments contain a traceability tag that contains a searchable pattern of the format `<system>/blockname`. `<system>` is the root model or a Subsystem inside the model, and `blockname` is the name of the block inside that model or Subsystem.

For example, if you have a model, `foo`, that has a Subsystem, `outer`, and a nested Subsystem, `Inner`, then the `<System>` tag is:

- `<Root>`: `foo`
- `<S1>`: `foo/outer`
- `<S2>`: `foo/outer/inner`

Dependency

To specify this setting, select the **Generate traceability report** check box.

Command-Line Information

Property: TraceabilityStyle

Type: character vector

Value: 'LineLevel' | 'CommentBased'

Default: 'LineLevel'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, when you generate a traceability report for the `symmetric_fir` subsystem inside the `sfir_fixed` model, specify the `TraceabilityStyle` by using either of these methods:

- Pass in the `TraceabilityStyle` property as an argument to `makehdl`.

```
makehdl('sfir_fixed/symmetric_fir','Traceability','on',...  
        'TraceabilityStyle','CommentBased')
```

- Enable the `TraceabilityStyle` property using `hdlset_param`, and then use `makehdl`.

```
hdlset_param('sfir_fixed','Traceability','on')  
hdlset_param(gcs,'TraceabilityStyle','CommentBased')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Create and Use Code Generation Reports” on page 25-2
- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 25-5
- `makehdl`

Generate model Web view

Include the model Web view in the HDL Code Generation report to navigate between the code and model within the same window. With a model Web view, you can click a link in the generated code to highlight the corresponding block in the model. Using this capability, you can review, analyze, and debug the generated HDL code. You can share your model and generated code outside of the MATLAB environment.

Settings

Default: Off

On

Include model Web view in the Code Generation report. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the model web view.

Off

Do not include model Web view in the Code Generation report.

Dependencies

To include a Web view (Simulink Report Generator) of the model in the Code Generation report, you must have Simulink Report Generator™ installed.

Command-Line Information

Parameter: HDLGenerateWebview

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can generate a model web view when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the HDLGenerateWebview property as an argument to makehdl.
`makehdl('sfir_fixed/symmetric_fir','HDLGenerateWebview','on')`
- Enable the HDLGenerateWebview property using hdlset_param and then use makehdl.

```
hdlset_param('sfir_fixed','HDLGenerateWebview','on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Create and Use Code Generation Reports” on page 25-2
- “Web View of Model in Code Generation Report” on page 25-13
- makehdl

Generate resource utilization report

Enable or disable generation of an HTML resource utilization report. The report contains a summary and detailed information about the number of hardware resources, such as multipliers, adders, and registers that are used in the generated HDL code. If you have floating-point data types in your model, you can generate HDL code with native floating point support or map your design to Intel or Xilinx FPGA floating-point libraries. The resource utilization report displays a target-specific report corresponding to FPGA floating-point library mapping and a resource report corresponding to HDL code in native floating-point mode.

Settings

Default: Off

On

Create and display an HTML resource utilization report. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the resource utilization report.

Off

Do not create an HTML resource utilization report.

Command-Line Information

Property: ResourceReport

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can generate a resource utilization report when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the `Traceability` property as an argument to `makehdl`.

```
makehdl('sfir_fixed/symmetric_fir','ResourceReport','on')
```

- Enable the Traceability property using `hdlset_param` and then use `makehdl`.

```
hdlset_param('sfir_fixed','ResourceReport','on')  
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Create and Use Code Generation Reports” on page 25-2
- `makehdl`

Generate high-level timing critical path report

Specify whether to generate a highlighting script that shows the estimated critical path. The report displays the critical path delay and generates a highlighting script as a link that you can click to highlight the estimated critical path in the generated model. If your design contains blocks without timing information, the report displays the link to another highlighting script that is generated to highlight those blocks.

Settings

Default: Off

On

Generate a highlighting script that shows the estimated critical path. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the critical path estimation report.

To estimate the critical path for single-precision floating-point models, use the Native Floating Point mode. In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Floating Point Target** tab, set **Library** to Native Floating Point

Off

Do not calculate the estimated critical path.

Command-Line Information

Property: CriticalPathEstimation

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can generate a critical path estimation report when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the `CriticalPathEstimation` property as an argument to `makehdl`.
`makehdl('sfir_fixed/symmetric_fir','CriticalPathEstimation','on')`
- Enable the `CriticalPathEstimation` property using `hdlset_param` and then use `makehdl`.
`hdlset_param('sfir_fixed','CriticalPathEstimation','on')`
`makehdl('sfir_fixed/symmetric_fir')`

See Also

- “Create and Use Code Generation Reports” on page 25-2
- “Critical Path Estimation Without Running Synthesis” on page 24-78
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65
- `makehdl`

Generate optimization report

Enable or disable generation of an HTML optimization report. The report contains information about the results of distributed pipelining, streaming, sharing, delay balancing, and adaptive pipelining optimizations that are implemented in the generated code. The report includes hyperlinks back to referenced blocks, subsystems, or validation models. If you have floating-point data types in your model, you can generate HDL code with native floating point support or map your design to Intel or Xilinx FPGA floating-point libraries. When you map to FPGA floating-point libraries, the optimization report displays a target code generation section that displays the target device summary and a link to the generated model.

Settings

Default: Off

On

Create and display an HTML optimization report. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the optimization report.

Off

Do not create an HTML optimization report.

Command-Line Information

Property: OptimizationReport

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can generate an optimization report when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the `OptimizationReport` property as an argument to `makehdl`.

```
makehdl('sfir_fixed/symmetric_fir','OptimizationReport','on')
```

- Enable the `OptimizationReport` property using `hdlset_param` and then use `makehdl`.

```
hdlset_param('sfir_fixed','OptimizationReport','on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Create and Use Code Generation Reports” on page 25-2
- `makehdl`

HDL Code Generation Pane: Test Bench

- “Test Bench Overview” on page 18-2
- “Test Bench Generation Output” on page 18-3
- “Test Bench name, data file, and reference Postfix” on page 18-9
- “Clock Input Signals” on page 18-12
- “Setup and Hold Time” on page 18-15
- “Clock Enable and Reset Input Signals” on page 18-17
- “Test Bench Stimulus and Output” on page 18-21
- “Multi-file test bench” on page 18-26
- “Floating Point Tolerance” on page 18-28
- “Simulation library path” on page 18-30

Test Bench Overview

The **Test Bench** pane lets you set options that determine characteristics of generated test bench code.

Generate Test Bench Button

The **Generate Test Bench** button initiates test bench generation for the system selected in the **Generate HDL for** menu on the parent HDL Code Generation pane. Make sure that the system selected is the DUT. Testbench generation is disabled if you select the entire model. See also `makehdltb`.

Test Bench Generation Output

HDL test bench

Enable or disable HDL test bench generation.

Settings

Default: selected

On

Enable generation of HDL test bench code. The code generator creates a HDL test bench by running a Simulink simulation to capture input vectors and expected output data for your DUT.

This test bench is the default test bench that HDL Coder generates for your model. If you have not already generated code for your model, running HDL test bench generation also generates code for your DUT.

Specify your HDL simulator in the **Simulation tool** menu. HDL Coder generates build-and-run scripts for the simulator that you specify.

Off

Suppress generation of HDL test bench code. You can use this option when you use an alternate test bench.

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

This check box enables the options in the **Configuration** section of the **Test Bench** pane. Select a **Simulation tool** to generate scripts to build and run the test bench.

Command-Line Information

Property: GenerateHDLTestBench

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdltb`. To view the property value, use `hdlget_param`.

For example, to generate a HDL test bench for the `sfir_fixed/symmetric_fir` Subsystem, pass the DUT as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir')
```

Cosimulation model

Enable or disable generation of a model including a HDL Cosimulation block. This option requires an HDL Verifier license. After you select this check box, specify your **Simulation tool**. You can select Mentor Graphics ModelSim or Cadence Incisive® for cosimulation. Custom script settings are not supported with this test bench.

Settings

Default: not selected

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

Property: GenerateCoSimBlock

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Property: GenerateCoSimModel

Type: character vector

Value: 'ModelSim' | 'Incisive' | 'None'

Default: 'ModelSim'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate a Cosimulation Model” on page 27-41

SystemVerilog DPI test bench

Enable or disable generation of the SystemVerilog DPI test bench. Select your HDL simulator at **Simulation tool**. For SystemVerilog DPI test bench you can select Mentor Graphics ModelSim, Cadence Incisive, SynopsysVCS®, or Xilinx Vivado. Custom script settings are not supported with this test bench.

When you set this property, the coder generates a direct programming interface (DPI) component for your entire Simulink model, including your DUT and data sources. Your entire model must support C code generation with Simulink Coder. The coder generates a SystemVerilog test bench that compares the output of the DPI component with the output of the HDL implementation of your DUT. The coder also builds shared libraries and generates a simulation script for the simulator you select.

Consider using this option if the default HDL test bench takes a long time to generate or simulate. Generation of a DPI test bench is sometimes faster than the default version because it does not run a full Simulink simulation to create the test bench data. Simulation of a DPI test bench with a large data set is faster than the default version because it does not store the input or expected data in a separate file.

To use this feature, you must have HDL Verifier and Simulink Coder licenses. To run the SystemVerilog testbench with generated VHDL code, you must have a mixed-language simulation license for your HDL simulator.

Settings

Default: not selected

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

Property: GenerateSVPITestBench

Type: character vector

Value: 'ModelSim' | 'Incisive' | 'Custom' | 'VCS' | 'Vivado'

Default: 'ModelSim'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

GenerateSVDPItestbenchSimulationTool

“Verify HDL Design With Large Data Set Using SystemVerilog DPI Test Bench”

Simulation tool

Simulator where you will run the generated test benches. The tool generates a script to build and run your HDL code and test bench.

Settings

- **Mentor Graphics ModelSim:** This option is the default. HDL Coder generates the selected types of test benches for use with Mentor Graphics ModelSim.
- **Cadence Incisive:** The coder generates the selected types of test benches for use with Cadence Incisive.
- **Custom:** Selecting this option enables the custom script options on the **EDA Tool Scripts** pane.
- **VCS:** This simulator is supported only for **SystemVerilog DPI test bench**.
- **Vivado:** This simulator is supported only for **SystemVerilog DPI test bench**.

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

For HDL test bench, use the `SimulationTool` property. For cosimulation, use the `GenerateCosimModel` property. For SystemVerilog DPI test bench, use the `GenerateSVDPItestbench` property.

Property: `SimulationTool`

Type: character vector

Value: 'Mentor Graphics ModelSim' | 'Cadence Incisive' | 'Custom'

Default: 'Mentor Graphics ModelSim'

Property: `GenerateCosimModel`

Type: character vector

Value: 'ModelSim' | 'Incisive' | None

Default: 'ModelSim'
Property: GenerateSVPITestbench
Type: character vector
Value: 'ModelSim' | 'Incisive' | 'Custom' | 'VCS' | 'Vivado'
Default: 'ModelSim'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

SimulationTool

HDL code coverage

Enable or disable HDL code coverage flags in the generated simulator scripts

With this option enabled, when you run the HDL simulation, code coverage is collected for your generated test bench. Specify your HDL simulator in the `SimulationTool` property. The coder generates build-and-run scripts for the simulator you specify.

Settings

Default: not selected

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

Property: HDLCodeCoverage
Type: character vector
Value: 'on' | 'off'
Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

SimulationTool

Test Bench name, data file, and reference Postfix

Test bench name postfix

Specify a suffix appended to the test bench name.

Settings

Default: `_tb`

For example, if the name of your DUT is `my_test`, HDL Coder adds the default postfix `_tb` to form the name `my_test_tb`.

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

Property: `TestBenchPostFix`

Type: character vector

Default: `'_tb'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`TestBenchPostFix`

Test bench reference postfix

Specify a character vector to be appended to names of reference signals generated in test bench code.

Settings

Default: `'_ref'`

Reference signal data is represented as arrays in the generated test bench code. The character vector specified by **Test bench reference postfix** is appended to the generated signal names.

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

Parameter: TestBenchReferencePostFix

Type: character vector

Default: '_ref'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

TestBenchReferencePostFix

Test bench data file name postfix

Specify suffix added to test bench data file name when generating multi-file test bench.

Settings

Default: '_data'

HDL Coder applies the **Test bench data file name postfix** character vector only when generating a multi-file test bench (i.e., when **Multi-file test bench** is selected).

For example, if the name of your DUT is `my_test`, and **Test bench name postfix** has the default value `_tb`, the coder adds the postfix `_data` to form the test bench data file name `my_test_tb_data`.

Dependency

This parameter is enabled by **Multi-file test bench**.

Command-Line Information

Property: TestBenchDataPostFix

Type: character vector

Default: '_data'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

TestBenchDataPostFix

Clock Input Signals

Force clock

Specify whether the test bench forces clock input signals.

Settings

Default: On

On

The test bench forces the clock input signals. When this option is selected, the clock high and low time settings control the clock waveform.

Off

A user-defined external source forces the clock input signals.

Dependencies

This property enables the **Clock high time** and **Clock low time** options. This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: ForceClock

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

ForceClock

Clock high time (ns)

Specify the period, in nanoseconds, during which the test bench drives clock input signals high (1).

Settings**Default:** 5

The **Clock high time** and **Clock low time** properties define the period and duty cycle for the clock signal. Using the defaults, the clock signal is a square wave (50% duty cycle) with a period of 10 ns.

Dependency

This parameter is enabled when **Force clock** is selected.

Command-Line Information**Property:** ClockHighTime**Type:** integer**Value:** positive integer**Default:** 5

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

ClockHighTime

Clock low time (ns)

Specify the period, in nanoseconds, during which the test bench drives clock input signals low (0).

Settings**Default:** 5

The **Clock high time** and **Clock low time** properties define the period and duty cycle for the clock signal. Using the defaults, the clock signal is a square wave (50% duty cycle) with a period of 10 ns.

Dependency

This parameter is enabled when **Force clock** is selected.

Command-Line Information

Property: ClockLowTime

Type: integer

Value: positive integer

Default: 5

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

ClockLowTime

Setup and Hold Time

Hold time (ns)

Specify a hold time, in nanoseconds, for input signals and forced reset input signals.

Settings

Default: 2 (given the default clock period of 10 ns)

The hold time defines the number of nanoseconds that reset input signals and input data are held past the clock rising edge. The hold time is expressed as a positive integer or double (with a maximum of 6 significant digits after the decimal point).

Tips

- The specified hold time must be less than the clock period (specified by the **Clock high time** and **Clock low time** properties).
- This option applies to reset input signals only if **Force reset** is selected.

Dependencies

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: HoldTime

Type: integer

Value: positive integer

Default: 2

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

HoldTime

Setup time (ns)

Display setup time for data input signals.

Settings

Default: None

This is a display-only field, showing a value computed as (clock period - HoldTime) in nanoseconds.

Dependency

The value displayed in this field depends on the clock rate and the values of the **Hold time** property.

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Because this is a display-only field, a corresponding command-line property does not exist.

See Also

HoldTime

Clock Enable and Reset Input Signals

Force clock enable

Specify whether the test bench forces clock enable input signals.

Settings

Default: On

On

The test bench forces the clock enable input signals to active-high (1) or active-low (0), depending on the setting of the clock enable input value.

Off

A user-defined external source forces the clock enable input signals.

Dependencies

This property enables the **Clock enable delay (in clock cycles)** option.

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: ForceClockEnable

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

ForceClockEnable

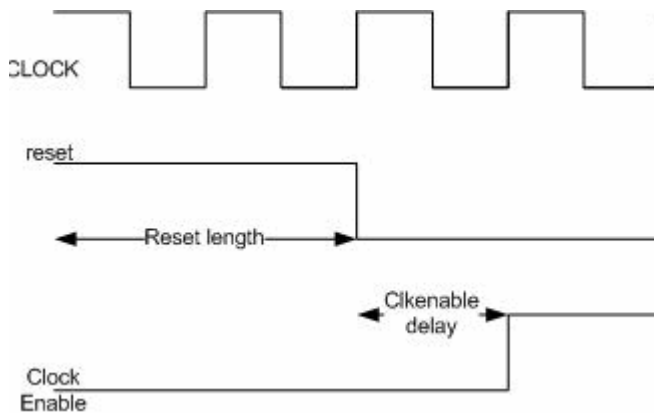
Clock enable delay (in clock cycles)

Define elapsed time (in clock cycles) between deassertion of reset and assertion of clock enable.

Settings

Default: 1

The **Clock enable delay (in clock cycles)** property defines the number of clock cycles elapsed between the time the reset signal is deasserted and the time the clock enable signal is first asserted. In the figure below, the reset signal (active-high) deasserts after 2 clock cycles and the clock enable asserts after a clock enable delay of 1 cycle (the default).



Dependency

This parameter is enabled when **Force clock enable** is selected.

Command-Line Information

Property: TestBenchClockEnableDelay

Type: integer

Default: 1

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

TestBenchClockEnableDelay

Force reset

Specify whether the test bench forces reset input signals.

Settings

Default: On

On

The test bench forces the reset input signals.

Off

A user-defined external source forces the reset input signals.

Tips

If you select this option, you can use the **Hold time** option to control the timing of a reset.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: ForceReset

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

ForceReset

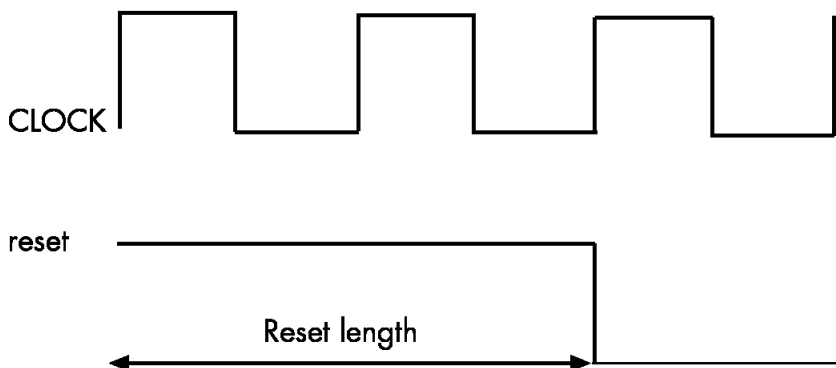
Reset length (in clock cycles)

Define length of time (in clock cycles) during which reset is asserted.

Settings

Default: 2

The **Reset length (in clock cycles)** property defines the number of clock cycles during which reset is asserted. **Reset length (in clock cycles)** must be an integer greater than or equal to 0. The following figure illustrates the default case, in which the reset signal (active-high) is asserted for 2 clock cycles.



Dependency

This parameter is enabled when **Force reset** is selected.

Command-Line Information

Property: Resetlength

Type: integer

Default: 2

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Test Bench Stimulus and Output

Hold input data between samples

Specify how long subrate signal values are held in valid state.

Settings

Default: On

On

Data values for subrate signals are held in a valid state across N base-rate clock cycles, where N is the number of base-rate clock cycles that elapse per subrate sample period. (N >= 2.)

Off

Data values for subrate signals are held in a valid state for only one base-rate clock cycle. For the subsequent base-rate cycles, data is in an unknown state (expressed as 'X') until leading edge of the next subrate sample period.

Tip

In most cases, the default (On) is the best setting for **Hold input data between samples**. This setting matches the behavior of a Simulink simulation, in which subrate signals are held valid through each base-rate clock period.

In some cases (for example modeling memory or memory interfaces), it is desirable to clear **Hold input data between samples**. In this way you can obtain diagnostic information about when data is in an invalid ('X') state.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: HoldInputDataBetweenSamples

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HoldInputDataBetweenSamples`

Initialize test bench inputs

Specify initial value driven on test bench inputs before data is asserted to DUT.

Settings

Default: Off

On

Initial value driven on test bench inputs is '0'.

Off

Initial value driven on test bench inputs is 'X' (unknown).

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: `InitializeTestBenchInputs`

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`InitializeTestBenchInputs`

Ignore output data checking (number of samples)

Specify number of samples during which output data checking is suppressed.

Settings

Default: 0

The value must be a positive integer.

When the value of **Ignore output data checking (number of samples)**, N, is greater than zero, the test bench suppresses output data checking for the first N output samples after the clock enable output (`ce_out`) is asserted.

When using pipelined block implementations, output data may be in an invalid state for some number of samples. To avoid spurious test bench errors, determine this number and set **Ignore output data checking (number of samples)** accordingly.

Be careful to specify N as a number of samples, not as a number of clock cycles. For a single-rate model, these are equivalent, but they are not equivalent for a multirate model.

You should use **Ignore output data checking (number of samples)** in cases where there is a state (register) initial condition in the HDL code that does not match the Simulink state, including the following specific cases:

- When you set the `DistributedPipelining` property to 'on' for the MATLAB Function block (see “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-39)
- When you set the `ResetType` property to 'None' for the following blocks:
 - `commcnvintrlv2/Convolutional Deinterleaver`
 - `commcnvintrlv2/Convolutional Interleaver`
 - `commcnvintrlv2/General Multiplexed Deinterleaver`
 - `commcnvintrlv2/General Multiplexed Interleaver`
 - `dpsigops/Delay`
 - `simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Enabled`
 - `simulink/Commonly Used Blocks/Unit Delay`
 - `simulink/Discrete/Delay`

- simulink/Discrete/Memory
- simulink/Discrete/Tapped Delay
- simulink/User-Defined Functions/MATLAB Function
- sflib/Chart
- sflib/Truth Table
- When generating a black box interface to existing manually written HDL code

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: IgnoreDataChecking

Type: integer

Default: 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

IgnoreDataChecking

Use file I/O to read/write test bench data

Create and use data files for reading and writing test bench input and output data.

Settings

Default: On

On

Create and use data files for reading and writing test bench input and output data.

Off

Use constants in the test bench for DUT stimulus and reference data.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: UseFileIOInTestBench

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

UseFileIOInTestBench

Multi-file test bench

Divide generated test bench into helper functions, data, and HDL test bench code files.

Description

You can use this setting to specify how you want to divide files that contain the test bench code, data, and helper functions.

The file names are derived from the name of the DUT, the **Test bench name postfix** property, and the **Test bench data file name postfix** property as:

DUTname_TestBenchPostfix_TestBenchDataPostfix

For example, if the DUT name is `symmetric_fir`, and the target language is VHDL, the default test bench file names are:

- `symmetric_fir_tb.vhd`: test bench code
- `symmetric_fir_tb_pkg.vhd`: helper functions package
- `symmetric_fir_tb_data.vhd`: data package

If the DUT name is `symmetric_fir` and the target language is Verilog, the default test bench file names are:

- `symmetric_fir_tb.v`: test bench code
- `symmetric_fir_tb_pkg.v`: helper functions package
- `symmetric_fir_tb_data.v`: test bench data

Settings

Default: Off

On

Write three separate HDL files. There is a separate file for test bench code, helper functions, and test bench data.

Off

Write two separate HDL files. One file contains the HDL test bench code. The other file contains the helper functions package and test bench data.

Dependency

When this property is selected, **Test bench data file name postfix** is enabled.

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: MultifileTestBench

Type: character vector

Value: 'on' | 'off'

Default: 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

MultifileTestBench

Floating Point Tolerance

Floating point tolerance check based on

When you map your design to the native floating-point libraries or the floating-point target libraries, specify the floating-point tolerance check option.

Settings

Default: relative error

Select one of these options from the dropdown menu:

- `relative error`: This is the default option. When you verify the generated code by using HDL Testbench, HDL Coder checks for the floating-point tolerance of the native floating-point library or the floating-point target library that your design mapped to based on the relative error.
- `ulp error`: When you verify the generated code by using HDL Testbench, HDL Coder checks for the floating-point tolerance of the native floating-point library or the floating-point target library that your design mapped to based on the ULP error.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: FPToleranceStrategy

Type: character vector

Value: 'relative' | 'ULP'

Default: 'relative'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

FPToleranceStrategy

Tolerance Value

Enter the tolerance value based on the floating-point tolerance check setting that you specify.

Settings

Default: 1e-07

The value must be a positive integer or a double data type.

The default tolerance value depends on the floating-point tolerance check setting that you specify. When you set the **Floating point tolerance check based on** to:

- `relative error`, the default is a **Tolerance Value** of 1e-07. When you use this floating-point tolerance check setting, specify the tolerance value as a double data type. You can specify a **Tolerance Value**, N, that is less than or equal to 1e-07.
- `ulp error`, the default is a **Tolerance Value** of 0. When you use this floating-point tolerance check setting, specify the tolerance value as an integer. You can specify a **Tolerance Value**, N, that is greater than or equal to 0.

Command-Line Information

Property: FPToleranceValue

Type: double | integer

Default: 1e-07

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

FPToleranceValue

Simulation library path

Specify the path to your compiled Altera or Xilinx simulation libraries.

Settings

Default: ' '

Specify the path to the compiled Altera or Xilinx simulation libraries. Altera provides the simulation model files in `\quartus\eda\sim_lib` folder.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: SimulationLibPath

Type: character vector

Default: ' '

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

SimulationLibPath

HDL Code Generation Pane: EDA Tool Scripts

- “EDA Tool Scripts Overview” on page 19-2
- “Generate EDA scripts” on page 19-3
- “Compilation Script” on page 19-4
- “Simulation Script” on page 19-8
- “Synthesis Script” on page 19-12
- “Lint Script” on page 19-19

EDA Tool Scripts Overview

The **EDA Tool Scripts** pane lets you set the options that control generation of script files for third-party HDL simulation and synthesis tools.

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/or synthesize generated HDL code.

Settings

Default: On

On

Generation of script files is enabled.

Off

Generation of script files is disabled.

Command-Line Information

Parameter: EDAScriptGeneration

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- EDAScriptGeneration

Compilation Script

Compile file postfix

Specify a postfix to append to the DUT or test bench name to form the compilation script file name.

Settings

Default: `_compile.do`

For example, if the name of the device under test or test bench is `my_design`, HDL Coder adds the postfix `_compile.do` to form the name `my_design_compile.do`.

Command-Line Information

Property: `HDLCompileFilePostfix`

Type: character vector

Default: `'_compile.do'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- `HDLCompileFilePostfix`

Compile initialization

Format name passed to `fprintf` to write the `Init` section of the compilation script.

Settings

Default: `vlib %s\n`

The `Init` phase of the script performs required setup actions, such as creating a design library or a project file.

The implicit argument, `%s`, 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'`.

Command-Line Information**Property:** HDLCompileInit**Type:** character vector**Default:** 'vlib %s\n'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- HDLCompileInit

Compile command for VHDL

Format name passed to `fprintf` to write the `Cmd` section of the compilation script for VHDL files.

Settings**Default:** `vcom %s %s\n`

The command-per-file phase (`Cmd`) of the script is called iteratively, once per generated HDL file. On each call, a different file name is passed in.

The two implicit arguments in the `compile` command 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).

Command-Line Information**Property:** HDLCompileVHDLCmd**Type:** character vector**Default:** 'vcom %s %s\n'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- HDLCompileVHDLCmd

Compile command for Verilog

Format name passed to `fprintf` to write the `Cmd` section of the compilation script for Verilog files.

Settings

Default: `vlog %s %s\n`

The command-per-file phase (`Cmd`) of the script is called iteratively, once per generated HDL file. On each call, a different file name is passed in.

The two implicit arguments in the compile command are the contents of the `SimulatorFlags` property and the file name of the current entity or module. To omit the flags, set `SimulatorFlags` property to `' '` (the default).

Command-Line Information

Property: `HDLCompileVerilogCmd`

Type: character vector

Default: `'vlog %s %s\n'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- `HDLCompileVerilogCmd`

Compile termination

Format name passed to `fprintf` to write the termination portion of the compilation script.

Settings

Default: empty character vector

The termination phase (`Term`) 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.

Command-Line Information**Property:** HDLCompileTerm**Type:** character vector**Default:** ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- HDLCompileTerm

Simulation Script

Simulation file postfix

Specify a postfix to append to the DUT or test bench name to form the simulation script file name.

Settings

Default: `_sim.do`

For example, if the name of the device under test or test bench is `my_design`, HDL Coder adds the postfix `_sim.do` to form the name `my_design_sim.do`.

Command-Line Information

Property: `HDLsimFilePostfix`

Type: character vector

Default: `'_sim.do'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- `HDLsimFilePostfix`

Simulation initialization

Format name passed to `fprintf` to write the initialization section of the simulation script.

Settings

Default: The default is

```
['onbreak resume\nonerror resume\n']
```

The `Init` phase of the script performs required setup actions, such as creating a design library or a project file.

Command-Line Information**Property:** HDLSimInit**Type:** character vector**Default:** ['onbreak resume\nonerror resume\n']

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- HDLSimInit

Simulation command

Format name passed to `fprintf` to write the simulation command.

Settings**Default:** `vsim -novopt %s.%s\n`

If your target language is VHDL, the first implicit argument is the value of **VHDL library name**. If your target language is Verilog, the first implicit argument is 'work'.

The second implicit argument is the top-level module or entity name.

Command-Line Information**Property:** HDLSimCmd**Type:** character vector**Default:** 'vsim -novopt %s.%s\n'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- HDLSimCmd

Simulation waveform viewing command

Specify the waveform viewing command written to simulation script.

Settings

Default: `add wave sim:%s\n`

The implicit argument, %s, adds the signal paths for the DUT top-level input, output, and output reference signals.

Command-Line Information

Property: HDLSimViewWaveCmd

Type: character vector

Default: 'add wave sim:%s\n'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- HDLSimViewWaveCmd

Simulation termination

Format name passed to `fprintf` to write the termination portion of the simulation script.

Settings

Default: `run -all\n`

The termination phase (Term) 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.

Command-Line Information

Property: HDLSimTerm

Type: character vector

Default: 'run -all\n'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- `HDLsimTerm`

Simulator flags

Specify simulator flags to apply to generated compilation scripts.

Settings

Default: `' '` (no simulator flags)

Specify simulator flags to apply to generated compilation scripts as a character vector. The simulator flags are specific to your application and the simulator you are using. For example, if you must use the 1076-1993 VHDL compiler, specify the flag `-93`.

The flags you specify with this option are added to the compilation command in generated compilation scripts. The simulation command is specified by the `HDLCompileVHDLcmd` or `HDLCompileVerilogCmd` properties.

Command-Line Information

Property: `SimulatorFlags`

Type: character vector

Default: `' '`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- `SimulatorFlags`
- `HDLCompileVerilogCmd`
- `HDLCompileVHDLcmd`

Synthesis Script

Choose synthesis tool

Enable or disable generation of synthesis scripts, and select the synthesis tool for which HDL Coder generates scripts.

Settings

Default: None

None

When you select **None**, HDL Coder does not generate a synthesis script. The coder clears and disables the fields in the **Synthesis script** pane.

Xilinx ISE

Generate a synthesis script for Xilinx ISE. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_ise.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Microsemi Libero

Generate a synthesis script for Microsemi Libero. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_libero.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Mentor Graphics Precision

Generate a synthesis script for Mentor Graphics Precision. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_precision.tcl`

- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Altera Quartus II

Generate a synthesis script for Altera Quartus II. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_quartus.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Synopsys Synplify Pro

Generate a synthesis script for Synopsys Synplify Pro. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_synplify.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Xilinx Vivado

Generate a synthesis script for Xilinx Vivado. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_vivado.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Custom

Generate a custom synthesis script. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_custom.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with example TCL script code.

Command-Line Information**Property:** HDLSynthTool**Type:** character vector**Value:** 'None' | 'ISE' | 'Liberio' | 'Precision' | 'Quartus' | 'Synplify' | 'Vivado' | 'Custom'**Default:** 'None'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

HDLSynthTool

Synthesis file postfix

Specify a postfix to append to file name for generated synthesis scripts.

Settings**Default:** None.

Your choice of synthesis tool (from the **Choose synthesis tool** pulldown menu) 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
```

For example, if the DUT name is `my_design` and the choice of synthesis tool is Synopsys Synplify Pro, HDL Coder adds the postfix `_synplify.tcl` to form the name `my_design_synplify.tcl`.

Dependency

To use this setting, the **Choose synthesis tool** or `HDLSynthTool` property must be set to a value other than `None`.

Command-Line Information**Property:** HDLSynthFilePostfix**Type:** character vector**Default:** none

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- HDLSynthFilePostfix

Synthesis initialization

Format name passed to `fprintf` to write the initialization section of the synthesis script.

Settings**Default:** none.

Your choice of synthesis tool (from the **Choose synthesis tool** pulldown menu) sets the **Synthesis initialization** string. The content of the string is specific to the selected synthesis tool.

The default is a synthesis project creation command passed as a format string to `fprintf` to write the `Init` section of the synthesis script. The implicit argument, `%s`, is the top-level module or entity name.

Dependency

To use this setting, the **Choose synthesis tool** or `HDLSynthTool` property must be set to a value other than `None`.

Command-Line Information**Property:** HDLSynthInit**Type:** character vector**Default:** none

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- HDLSynthInit

Synthesis command

Format name passed to `fprintf` to write the synthesis command.

Settings

Default: none.

Your choice of synthesis tool (from the **Choose synthesis tool** pulldown menu) sets the **Synthesis command** string. The content of the string is specific to the selected synthesis tool.

The default is a format string passed to `fprintf` to write the `Cmd` section of the synthesis script. The implicit argument, `%s`, is the filename of the entity or module.

To avoid issues when generating synthesis scripts for various tools, retain both format specifiers (`%s`).

Dependency

To use this setting, the **Choose synthesis tool** or `HDLSynthTool` property must be set to a value other than `None`.

Command-Line Information

Property: `HDLSynthCmd`

Type: character vector

Default: none

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- `HDLSynthCmd`

Synthesis termination

Specify a format name that is passed to `fprintf` to write the termination portion of the synthesis script.

Settings

Default: none

Your choice of synthesis tool (from the **Choose synthesis tool** pulldown menu) sets the **Synthesis termination** string. The content of the string is specific to the selected synthesis tool.

The default is a format name passed to `fprintf` to write the Term section of the synthesis script. The termination string does not take arguments.

Dependency

To use this setting, the **Choose synthesis tool** or `HDLSynthTool` property must be set to a value other than `None`.

Command-Line Information

Property: `HDLSynthTerm`

Type: character vector

Default: none

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- `HDLSynthTerm`

Additional files to add to synthesis project

Include additional HDL or constraint files in synthesis project.

Settings

Default: `''` (no files added)

Additional project files, such as HDL source files (.v, .vhd) or constraint files (.ucf), that you want to include in your synthesis project, specified as a character vector. Separate file names with a semicolon (;).

You cannot use this setting to include Tcl files. To specify synthesis project Tcl files, use the `AdditionalProjectCreationTclFiles` property of the `hdlcoder.WorkflowConfig` object.

Command-Line Information

Property: `SynthesisProjectAdditionalFiles`

Type: character vector

Default: ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- `SynthesisProjectAdditionalFiles`
- `hdlcoder.WorkflowConfig`

Lint Script

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.

After you select an HDL lint tool, the **Lint initialization**, **Lint command** and **Lint termination** fields are enabled.

Settings

Default: None

None

When you select **None**, the coder does not generate a lint script. The coder clears and disables the fields in the **Lint script** pane.

Ascent Lint

Generate a lint script for Real Intent Ascent Lint.

HDL Designer

Generate a lint script for Mentor Graphics HDL Designer.

Leda

Generate a lint script for Synopsys Leda.

SpyGlass

Generate a lint script for Atrenta SpyGlass.

Custom

Generate a custom synthesis script.

Command-Line Information

Property: HDLLintTool

Type: character vector

Value: 'None' | 'AscentLint' | 'Leda' | 'SpyGlass' | 'Custom'

Default: 'None'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Generate an HDL Lint Tool Script” on page 26-71
- HDLLintTool

Lint initialization

Enter an initialization text for your HDL lint script.

Command-Line Information

Property: HDLLintInit

Type: character vector

Default: none

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Generate an HDL Lint Tool Script” on page 26-71
- HDLLintInit

Lint command

Enter the command for your HDL lint script.

Command-Line Information

Property: HDLLintCmd

Type: character vector

Default: none

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Generate an HDL Lint Tool Script” on page 26-71
- HDLLintCmd

Lint termination

Enter a termination character vector for your HDL lint script.

Command-Line Information

Property: HDLLintTerm

Type: character vector

Default: none

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- “Generate an HDL Lint Tool Script” on page 26-71
- HDLLintTerm

Modeling Guidelines

- “HDL Modeling Guidelines Severity Levels” on page 20-3
- “Model Design and Compatibility Guidelines - By Numbered List” on page 20-5
- “Guidelines for Supported Blocks and Data Types - By Numbered List” on page 20-10
- “Guidelines for Speed and Area Optimizations - By Numbered List” on page 20-14
- “Basic Guidelines for Modeling HDL Algorithm in Simulink” on page 20-16
- “Guidelines for Model Setup and Checking Model Compatibility” on page 20-23
- “Modeling with Simulink, Stateflow, and MATLAB Function Blocks” on page 20-28
- “Terminate Unconnected Block Outputs and Usage of Commenting Blocks” on page 20-32
- “Identify and Programmatically Change and Display HDL Block Parameters” on page 20-38
- “DUT Subsystem Guidelines” on page 20-45
- “Hierarchical Modeling Guidelines” on page 20-51
- “Design Considerations for Matrices and Vectors” on page 20-56
- “Use Bus Signals to Improve Readability of Model and Generate HDL Code” on page 20-62
- “Guidelines for Clock and Reset Signals” on page 20-70
- “Modeling with Native Floating Point” on page 20-79
- “Design Considerations for RAM Blocks and Blocks in HDL Operations Library” on page 20-82
- “Usage of Blocks in Logic and Bit Operations Library” on page 20-87
- “Generate FPGA Block RAM from Lookup Tables” on page 20-95
- “Usage of Different Subsystem Types” on page 20-99
- “Usage of Rate Change and Constant Blocks” on page 20-107
- “Simulink Data Type Considerations” on page 20-111
- “Resource Sharing Settings for Various Blocks” on page 20-114
- “Resource Sharing of Subsystems and Floating-Point IPs” on page 20-119

- “Distributed Pipelining and Clock-Rate Pipelining Guidelines” on page 20-125
- “Insert Distributed Pipeline Registers for Blocks with Vector Data Type Inputs” on page 20-129

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 20-5
- “Guidelines for Supported Blocks and Data Types - By Numbered List” on page 20-10

- “Guidelines for Speed and Area Optimizations - By Numbered List” on page 20-14

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 20-3.

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 Model Checker. To learn more about the HDL Model Checker, see “Getting Started with the HDL Model Checker” on page 39-2.

Guidelines 1.1: Basic Settings

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.1.1	“Use HDL-Supported Blocks” on page 20-16	Recommended	None
1.1.2	“Partition Model into DUT and Test Bench” on page 20-18	Recommended	None
1.1.3	“Avoid Using Double-Byte Characters” on page 20-19	Mandatory	None
1.1.4	“Document Model Features and Attributes” on page 20-20	Recommended	None
1.1.5	“Customize hdlsetup Function Based on Target Application” on page 20-23	Recommended	Model Check: “Check for safe model parameters” on page 38-6

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.1.6	“Check Subsystem for HDL Compatibility” on page 20-24	Recommended	None
1.1.7	“Run Model Checks for HDL Coder” on page 20-25	Recommended	None
1.1.8	“Modeling with Simulink, Stateflow, and MATLAB Function Blocks” on page 20-28	Informative	None
1.1.9	“Terminate Unconnected Block Outputs” on page 20-32	Mandatory	None
1.1.10	“Using Comment Out and Comment Through of Blocks” on page 20-34	Mandatory	None
1.1.11	“Adjust Sizes of Constant and Gain Blocks for Identifying Parameters” on page 20-38	Recommended	None
1.1.12	“Display Parameters that Affect HDL Code Generation” on page 20-39	Recommended	None
1.1.13	“Change Block Parameters by Using find_system and set_param” on page 20-43	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 20-45	Strongly Recommended	Model Check: “Check for invalid top level subsystem” on page 38-14

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.2.2	“Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks” on page 20-46	Strongly Recommended	None
1.2.3	“Insert Handwritten Code into Simulink Modeling Environment” on page 20-48	Informative	None
1.2.4	“Avoid Constant Block Connections to Subsystem Port Boundaries” on page 20-51	Mandatory	None
1.2.5	“Generate Parameterized HDL Code for Constant and Gain Blocks” on page 20-52	Recommended	None

Guidelines 1.3: Guidelines for Vectors and Buses

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.3.1	“Modeling Requirements for Matrices” on page 20-56	Mandatory	Model Check: “Check for large matrix operations” on page 38-19
1.3.2	“Avoid Generating Ascending Bit Order in HDL Code From Vector Signals” on page 20-58	Strongly Recommended	None
1.3.3	“Use Bus Signals to Improve Readability of Model and Generate HDL Code” on page 20-62	Informative	None

Guidelines 1.4: Guidelines for 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 20-70	Informative	Model Check: “Check for invalid top level subsystem” on page 38-14
1.4.2	“Create Multirate Model with Integer Clock Multiples by Clock Division” on page 20-70	Mandatory	None
1.4.3	“Use Dual Rate Dual Port RAM for Noninteger Multiple Sample Times” on page 20-74	Mandatory	None
1.4.4	“Asynchronous Clock Modeling in HDL Coder” on page 20-75	Recommended	None
1.4.5	“Use Global Reset Type Setting Based on Target Hardware” on page 20-77	Strongly Recommended	Model check: “Use Global Reset Type Setting Based on Target Hardware” on page 20-77

Guidelines 1.5: Modeling Guidelines for Native Floating Point

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
1.5.1	“Modeling with Native Floating Point” on page 20-79	Recommended	None

See Also

More About

- “Guidelines for Supported Blocks and Data Types - By Numbered List” on page 20-10
- “Guidelines for Speed and Area Optimizations - By Numbered List” on page 20-14

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 20-3.

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 Model Checker. To learn more about the HDL Model Checker, see “Getting Started with the HDL Model Checker” on page 39-2.

Guidelines 2.1: Blocks in 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 20-82	Recommended	None
2.1.2	“Serial to Parallel Conversion” on page 20-85	Recommended	None

Guidelines 2.2: Blocks in 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 20-87	Informative	None

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.2.2	“Usage of Logical Operator, Bitwise Operator, and Bit Reduce Blocks” on page 20-90	Informative	None
2.2.3	“Use Boolean Data Type for Compare to Constant and Relational Operator Blocks” on page 20-93	Strongly Recommended	Model Check: “Check for Relational Operator block usage” on page 38-29

Guidelines 2.3: Lookup Table Blocks

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.3.1	“Generate FPGA Block RAM from Lookup Tables” on page 20-95	Strongly 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 Top-Level DUT” on page 20-99	Mandatory	Model Check: “Check for invalid top level subsystem” on page 38-14
2.4.2	“Atomic Subsystem: Generate Reusable HDL Files” on page 20-99	Recommended	None
2.4.3	“Variant Subsystem: Using Variant Subsystems for HDL Code Generation” on page 20-100	Mandatory	None

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.4.4	“Model References: Build Model Design Using Smaller Partitions” on page 20-102	Recommended	None
2.4.5	“Block Settings of Enabled and Triggered Subsystems” on page 20-104	Mandatory	Model check: “Check initial conditions of enabled and triggered subsystems” on page 38-15

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 20-107	Recommended	None
2.5.2	“Use Discrete and Finite Sample Time for Constant Block” on page 20-109	Mandatory	Model Check: “Check for infinite and continuous sample time sources” on page 38-17

Guidelines 2.6: Data Types

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
2.6.1	“Use Boolean for Logical Data and Ufix1 for Numerical Data” on page 20-111	Mandatory	None
2.6.2	“Specify Data Type of Gain Blocks” on page 20-111	Recommended	None
2.6.3	“Enumerated Data Type Restrictions” on page 20-112	Mandatory	None

See Also

More About

- “Model Design and Compatibility Guidelines - By Numbered List” on page 20-5
- “Guidelines for Speed and Area Optimizations - By Numbered List” on page 20-14

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 20-3.

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 Model Checker. To learn more about the HDL Model Checker, see “Getting Started with the HDL Model Checker” on page 39-2.

Guidelines 3.1: Resource Sharing

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
3.1.1	“Resource Sharing of Add Blocks” on page 20-114	Recommended	None
3.1.2	“Resource Sharing of Gain Blocks” on page 20-115	Recommended	None
3.1.3	“Resource Sharing of Product Blocks” on page 20-116	Recommended	None
3.1.4	“Resource Sharing of Multiply-Add Blocks” on page 20-117	Recommended	None
3.1.5	“General Considerations for Sharing of Subsystems” on page 20-119	Recommended	None

Guideline ID	Title	Severity	Associated Model Check/ Coding Standard Rule
3.1.6	“Use MATLAB Datapath Architecture for Sharing with MATLAB Function Blocks” on page 20-120	Recommended	None
3.1.7	“Sharing of Atomic Subsystems” on page 20-121	Recommended	None
3.1.8	“Resource Sharing of Floating-Point IPs” on page 20-123	Recommended	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 20-125	Informative	None
3.2.2	“Recommended Distributed Pipelining Settings” on page 20-126	Recommended	None
3.2.3	“Insert Distributed Pipeline Registers for Blocks with Vector Data Type Inputs” on page 20-129	Informative	None

See Also

More About

- “Model Design and Compatibility Guidelines - By Numbered List” on page 20-5
- “Guidelines for Supported Blocks and Data Types - By Numbered List” on page 20-10

Basic Guidelines for Modeling HDL Algorithm in Simulink

In this section...
“Use HDL-Supported Blocks” on page 20-16
“Partition Model into DUT and Test Bench” on page 20-18
“Avoid Using Double-Byte Characters” on page 20-19
“Document Model Features and Attributes” on page 20-20

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 20-3.

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.

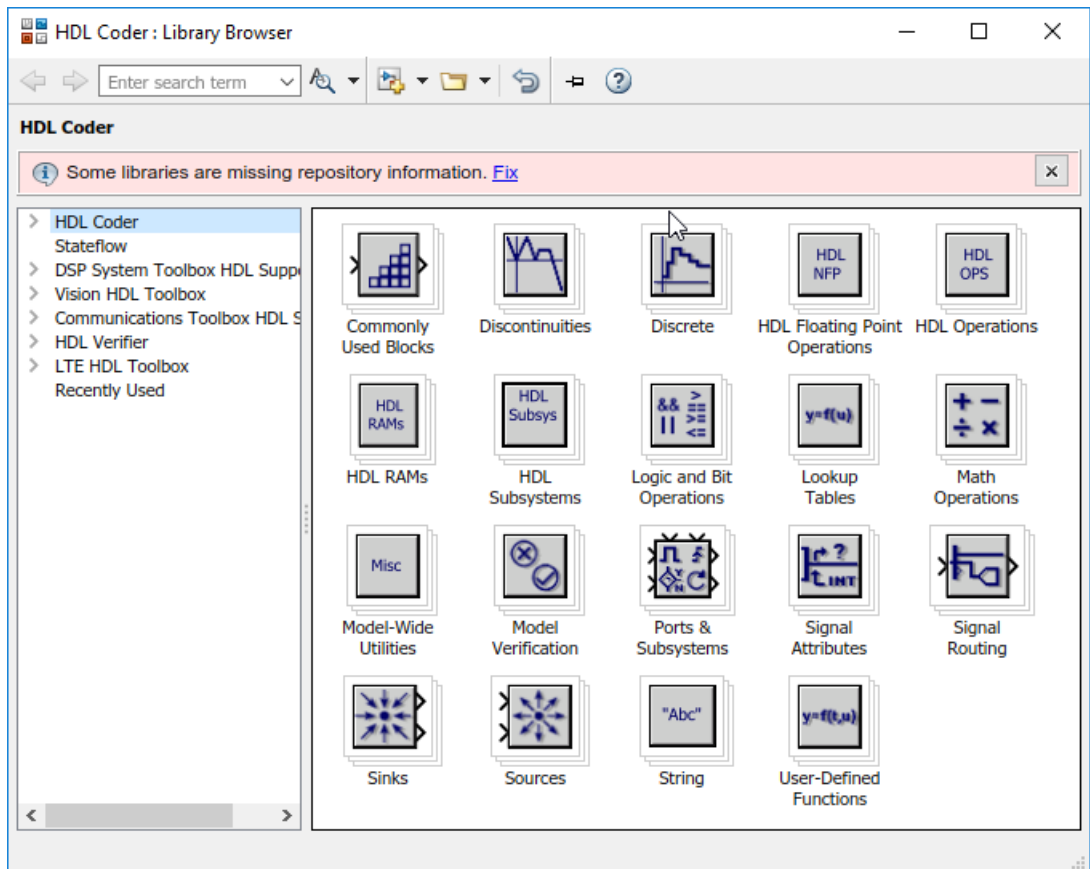
You can find additional HDL-supported blocks in these Simulink block libraries:

- **DSP System Toolbox HDL Support**
- **Communications Toolbox HDL Support**
- **Vision HDL Toolbox**
- **LTE 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

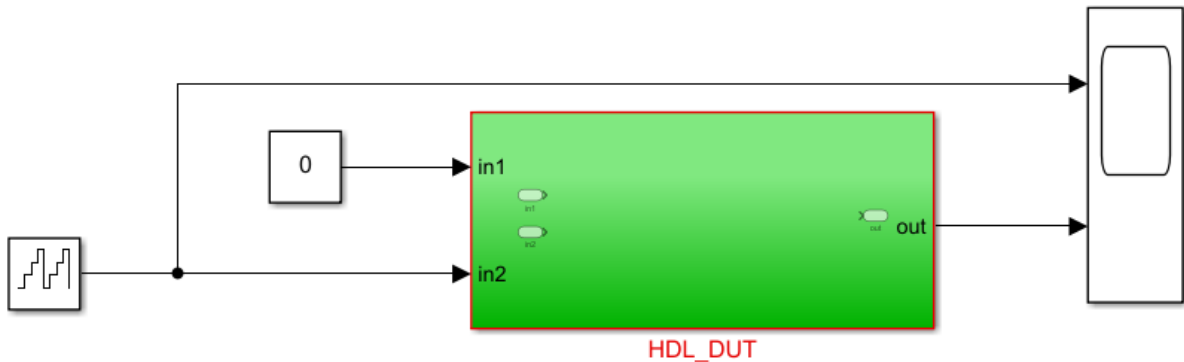
When you create your Simulink model for HDL code generation, the Subsystem that you want to generate HDL code for is the Design-Under-Test (DUT). This Subsystem contains Simulink blocks that can be implemented on your target FPGA or ASIC device. You can further partition the logic inside the DUT into 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 Simulinkmodel.
- 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.

Note: This model is configured with 'hdlsetup'



Add your design targeted for ASIC/FPGA inside HDL_DUT and then run the following command:
makehdl('HDL_DUT')

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 “File Comment Customization” on page 16-75.

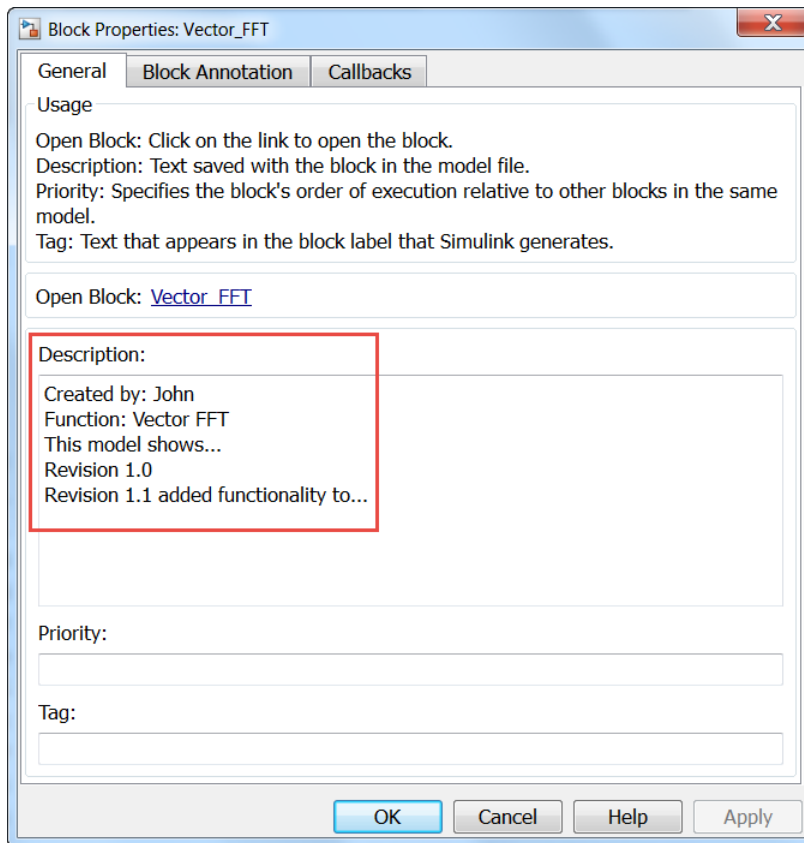
- **Model and Block Annotations, Text Comments, and Requirement Comments**

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 Code with Annotations or Comments” on page 25-16.

- **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

`checkhdl` | `hdl`lib | `hdlmodelchecker`

Modeling Guidelines

“Guidelines for Model Setup and Checking Model Compatibility” on page 20-23 |
“Modeling with Simulink, Stateflow, and MATLAB Function Blocks” on page 20-28

More About

- “Use Simulink Templates for HDL Code Generation” on page 10-9
- “Create Simulink Model for HDL Code Generation”
- “Show Supported Blocks in Library Browser” on page 25-23

Guidelines for Model Setup and Checking Model Compatibility

In this section...

“Customize hdlsetup Function Based on Target Application” on page 20-23

“Check Subsystem for HDL Compatibility” on page 20-24

“Run Model Checks for HDL Coder” on page 20-25

Use these guidelines to setup 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 20-3.

Customize hdlsetup Function Based on Target Application

Guideline ID

1.1.5

Severity

Strongly Recommended

Description

Before generating code, you must configure the model. To configure the model, you can use the `hdlsetup` function. The `hdlsetup` function uses the `set_param` function to set up models for HDL code generation. The settings include using a fixed-step discrete solver, specifying ASIC/FPGA as the hardware type, and so on. To see the settings that `hdlsetup` function saves on the model, run this command:

```
edit hdlsetup.m
```

Some of the settings that the `hdlsetup` function saves on the model may not be suitable for your target application. In such cases, you can customize the `hdlsetup.m` file such that it runs only those commands required for your target application. For example, you can disable some of the solver settings in the Configuration Parameters and instead enable certain model parameters such as displaying port data types, and so on.

```
% 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 see a custom `hdlsetup` function that consists of these commands and specifies some of the HDL-specific settings required for HDL code generation, open the file `myhdlsetup.m`.

```
edit myhdlsetup.m
```

You see that this custom `myhdlsetup` file also 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 25-21.

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 Model Checker or the Simulink Model Advisor checks for **HDL Coder**.

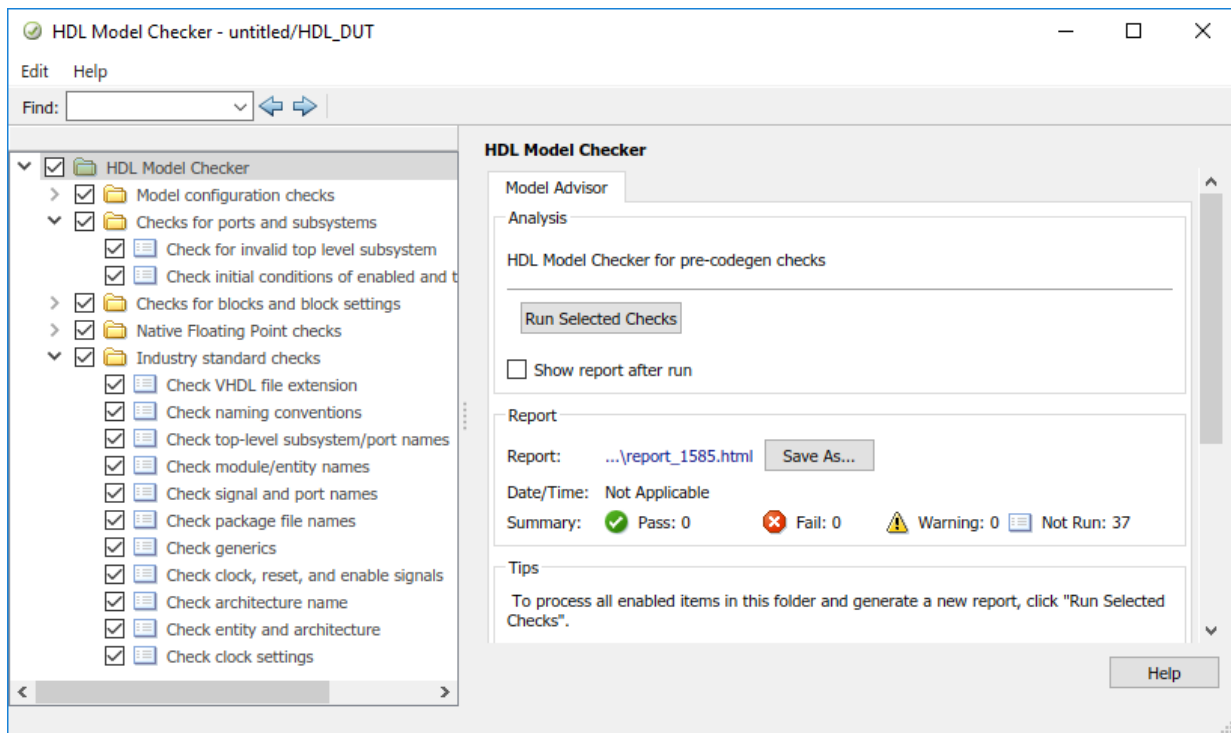
To open the HDL Model Checker:

- 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 > Check Model Compatibility**.

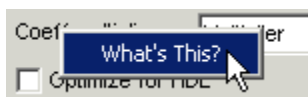
- At the command line, use the `hdlmodelchecker` function:

```
hdlmodelchecker(gcb)
```

When you run this command, the HDL Model Checker appears.



You may not have to run all checks in the HDL Model Checker. 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

`checkhdl` | `hdllib` | `hdlmodelchecker`

Modeling Guidelines

“Basic Guidelines for Modeling HDL Algorithm in Simulink” on page 20-16

More About

- “Use Simulink Templates for HDL Code Generation” on page 10-9
- “Create Simulink Model for HDL Code Generation”
- “Show Supported Blocks in Library Browser” on page 25-23

Modeling with Simulink, Stateflow, and MATLAB Function Blocks

In this section...
“Guideline ID” on page 20-28
“Severity” on page 20-28
“Description” on page 20-28

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 20-3.

Guideline ID

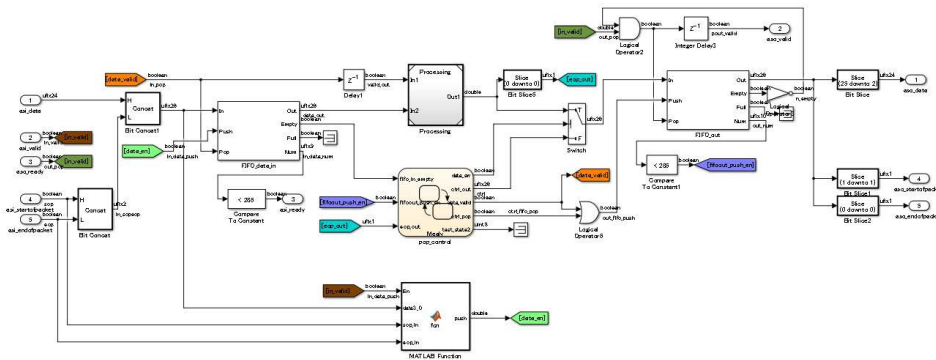
1.1.8

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 “Verilog HDL Import: Import Verilog Code and Generate Simulink Model” on page 10-114.

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 20-16

More About

- “Verify with HDL Cosimulation” on page 37-17
- “Show Supported Blocks in Library Browser” on page 25-23
- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “Introduction to Stateflow HDL Code Generation” on page 28-2
- “Generate Black Box Interface for Subsystem” on page 27-5
- “Generate Black Box Interface for Referenced Model” on page 27-10

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 20-3.

Terminate Unconnected Block Outputs

Guideline ID

1.1.9

Severity

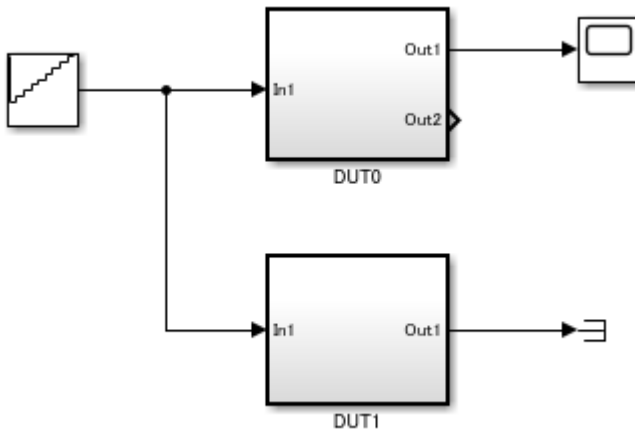
Mandatory

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')
```



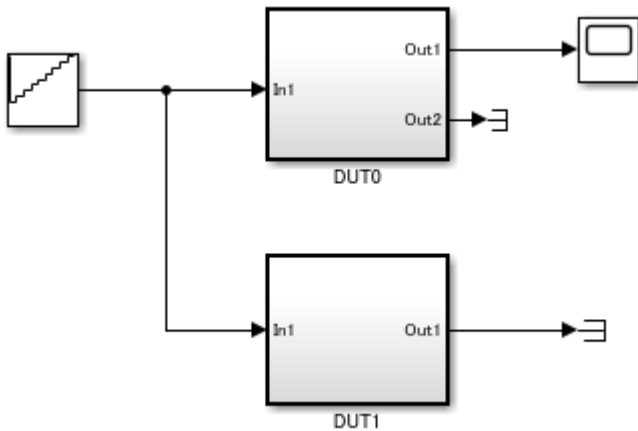
If you generate HDL code for this Subsystem, HDL Coder™ generates this error:

error in validation model generation: Failed to find source for output 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

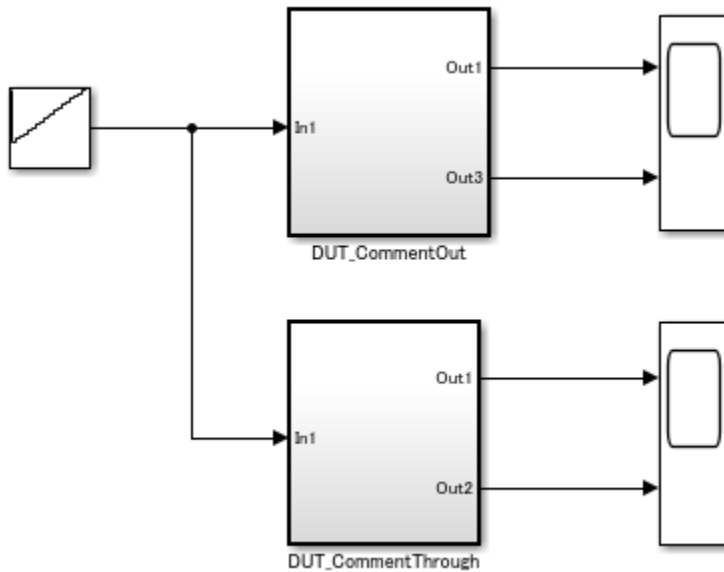
Mandatory

Description

HDL Coder™ does not support code generation for a block or blocks that you comment through.

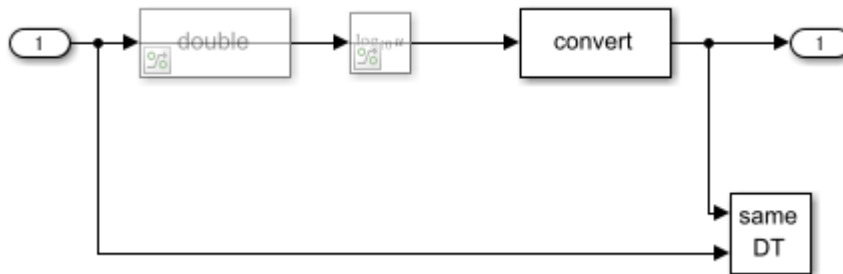
This model illustrates two Subsystems DUT_CommentOut and DUT_CommentThrough.

```
open_system('hdlcoder_comment_through_out')
```



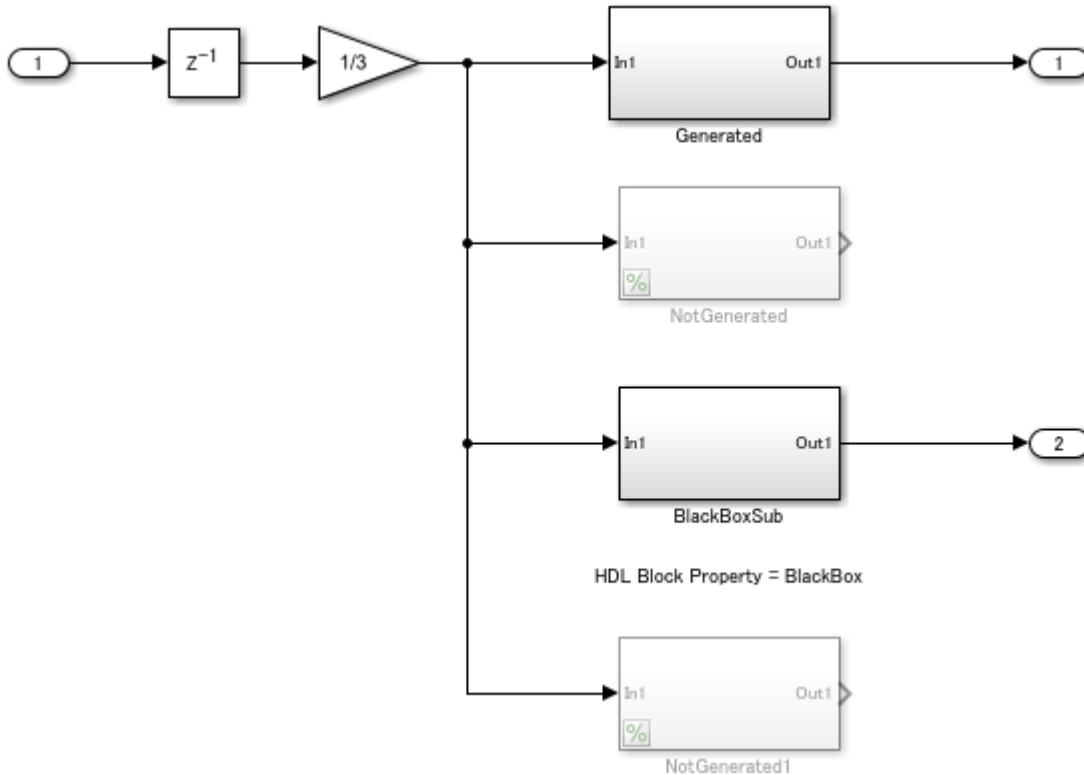
The `Dut_CommentThrough` Subsystem contains a Subsystem that is commented through. HDL Coder™ generates an error for this Subsystem when generating code. In addition, the code generator does not support comment through for DUT input in testbench.

```
open_system('hdlcoder_comment_through_out/DUT_CommentThrough/Generated')
```



The code generator supports blocks that are commented 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 two subsystems that are commented out.

```
open_system('hdlcoder_comment_through_out/DUT_CommentOut')
```



When you generate code for `Dut_CommentOut`, HDL Coder ignores these subsystems. The `Generated` Subsystem has blocks that are commented out and outputs a zero value.

You see this VHDL code generated for this Subsystem which indicates a constant zero value assigned to `Out1`.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
```

```
ENTITY Generated IS
  PORT( Out1
```

```
: OUT std_logic_vector(15 DOWNT0 0) -- s
```



```
    );  
END Generated;  
  
ARCHITECTURE rtl OF Generated IS  
    -- Signals  
    SIGNAL TmpGroundAtData_Type_DuplicateInport1_out1 : signed(15 DOWNT0 0); -- sfix16_  
  
BEGIN  
    -- Unsupported Block  
  
    TmpGroundAtData_Type_DuplicateInport1_out1 <= to_signed(16#0000#, 16);  
  
    Out1 <= std_logic_vector(TmpGroundAtData_Type_DuplicateInport1_out1);  
  
END rtl;
```

See Also

Modeling Guidelines

“Basic Guidelines for Modeling HDL Algorithm in Simulink” on page 20-16

More About

- “Comment Out and Comment Through” (Simulink)

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 20-3.

Adjust Sizes of Constant and Gain Blocks for Identifying Parameters

Guideline ID

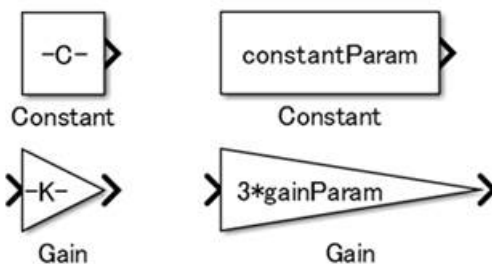
1.1.11

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

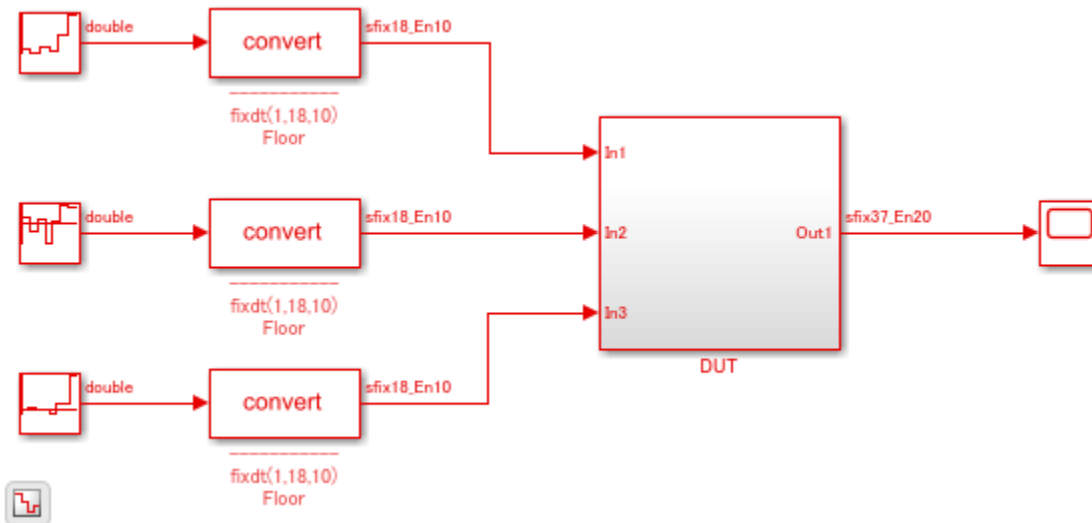
Recommended

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.

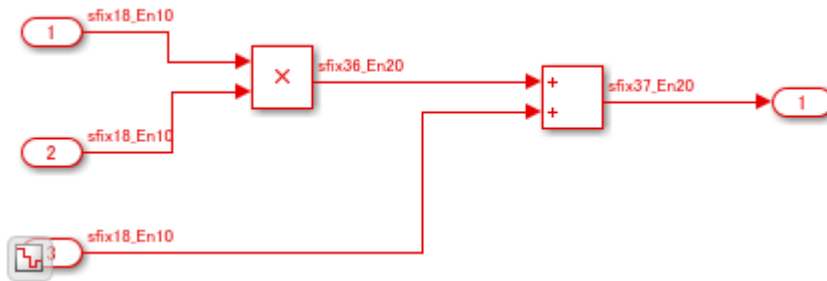
For example, open the model `hdlcoder_block_annotation_HDL_params.slx`.

```
open_system('hdlcoder_block_annotation_HDL_params')
set_param('hdlcoder_block_annotation_HDL_params','SimulationCommand','Update')
```



The DUT Subsystem performs a simple multiply-add operation.

```
open_system('hdlcoder_block_annotation_HDL_params/DUT')
```



There are HDL block parameters saved on the model. To see the 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', 'HierarchicalDistPipelining', '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', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'ResourceReport', 'on');
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 model with the HDL block parameters saved on the model, use the showHdlBlockParams script attached with the example.

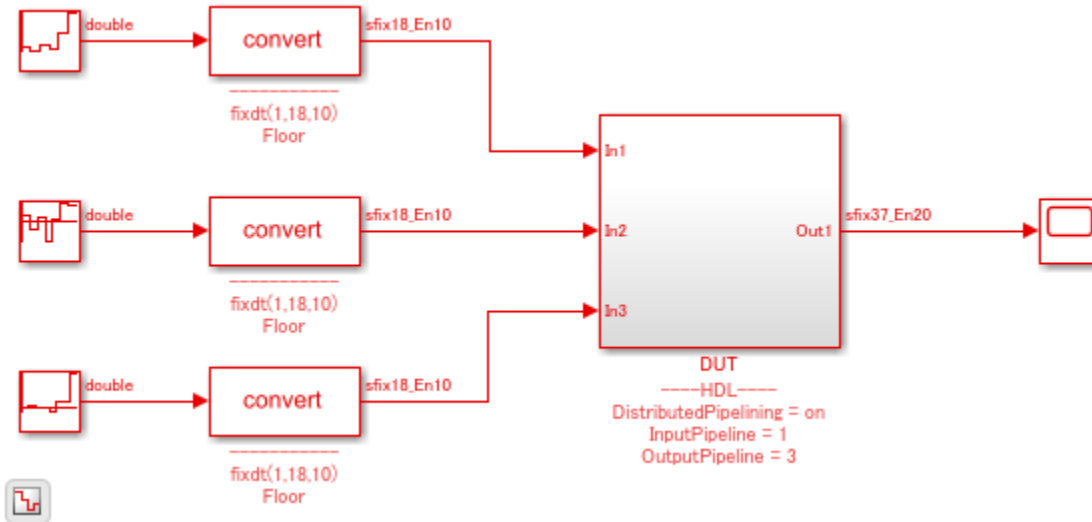
```
showHdlBlockParams('hdlcoder_block_annotation_HDL_params/DUT','on')
```

```
Add block annotation for hdlcoder_block_annotation_HDL_params/DUT.
----HDL----\nDistributedPipelining = on\nInputPipeline = 1\nOutputPipeline = 3
```

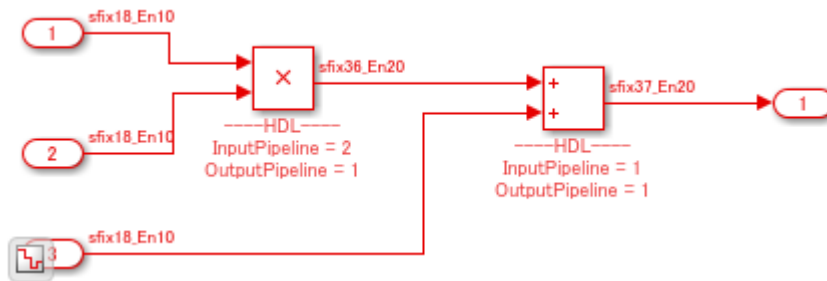
```
Add block annotation for hdlcoder_block_annotation_HDL_params/DUT/Add.
----HDL----\nInputPipeline = 1\nOutputPipeline = 1
```

```
Add block annotation for hdlcoder_block_annotation_HDL_params/DUT/Product.
----HDL----\nInputPipeline = 2\nOutputPipeline = 1
```

```
open_system('hdlcoder_block_annotation_HDL_params')
```



```
open_system('hdlcoder_block_annotation_HDL_params/DUT')
```

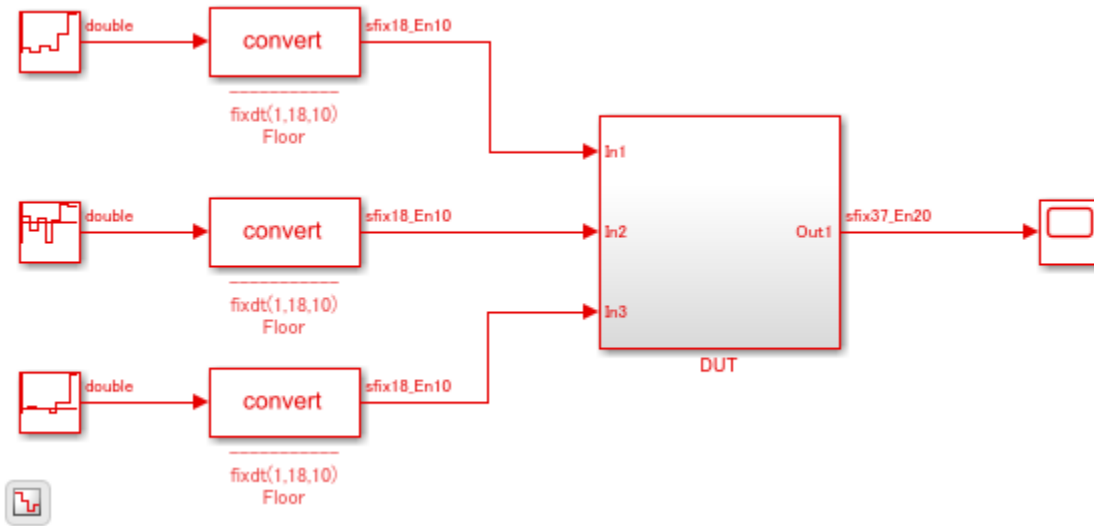


To remove the HDL block parameters annotation from the model, run the `showHdlBlockParams` set to off.

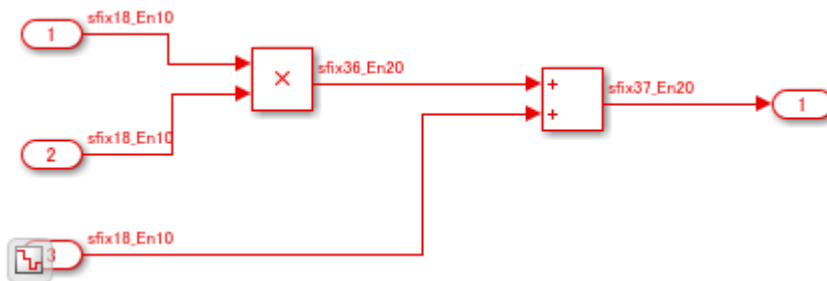
```
showHdlBlockParams('hdlcoder_block_annotation_HDL_params/DUT', 'off')
```

```
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/In1 are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/In2 are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/In3 are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/Add are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/Product are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/Out1 are removed
```

```
open_system('hdlcoder_block_annotation_HDL_params')
```



```
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` | `hdlsaveparams` | `set_param`

More About

- “Specify Block Properties” (Simulink)

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 20-3.

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 20-18.

Consider using these recommended settings when you design the DUT Subsystem for HDL code generation.

For a top-level DUT:

- 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 38-14.
- Make sure that the **HDL Architecture** of the DUT is not specified as a **BlackBox**. See “BlackBox Subsystems” on page 20-30.
- Connect output signals that are unconnected to a Terminator block. To learn more, see “Terminate Unconnected Block Outputs” on page 20-32.
- Make sure that the DUT does not contain blocks that you comment through. HDL Coder does not support code generation for these blocks. You can comment out blocks for code generation when the output signals are unused. See also “Using Comment Out and Comment Through of Blocks” on page 20-34.

If you use a Subsystem that is hierarchically below the top-level Subsystem as the DUT and generate HDL code, HDL Coder converts it to a model reference. After you generate HDL code, in the `hdlsrc` directory, you see a Simulink model with references to the validation and cosimulation model. In some cases, it can result in conflicts in reference file names and longer code generation time.

For a non-top-level 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 20-99.
- Leave the **HDL Architecture** of the DUT to default.

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.

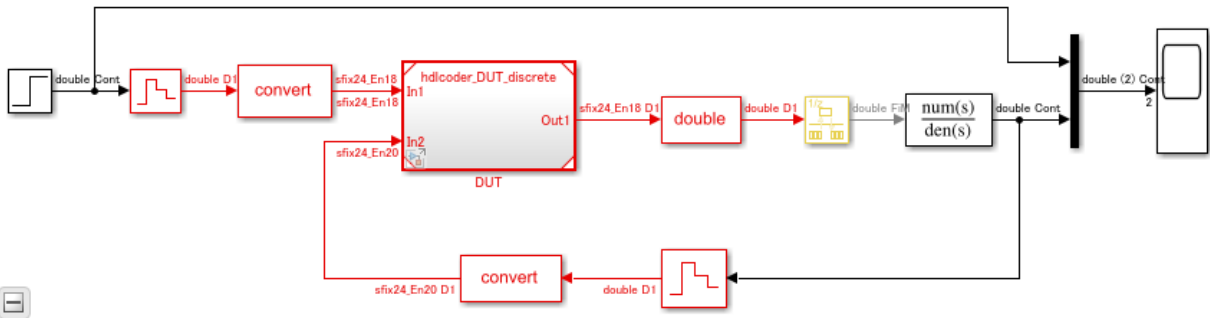
Note: To avoid issues such as reference model file confliction and longer code generation time, it is recommended that you avoid using a Subsystem that is hierarchically below the top-level Subsystem as the DUT.

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 reference

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'
```



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'
```



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 10-117
- “Limitations of Verilog HDL Import” on page 10-123

- **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 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 27-5 .

- **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 or Verilog 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 Using DocBlock” on page 27-12.

- **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 27-2

- “Customize Black Box or HDL Cosimulation Interface” on page 27-14
- “Generate a Cosimulation Model” on page 27-41

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 20-3.

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.

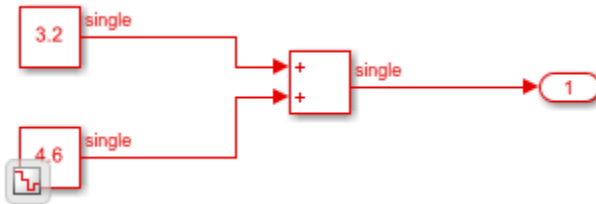
If you use floating-point data types with the `Native Floating Point` mode enabled, and input constant values to an arithmetic operator such as an Add block, HDL Coder™ replaces the Add block with a Constant block when generating code. This optimization can result in a Constant block directly connected 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.

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')
```

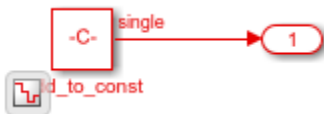


```
open_system('hdlcoder_constant_subsystem_boundary/DUT/Constant_subsys2')
```



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

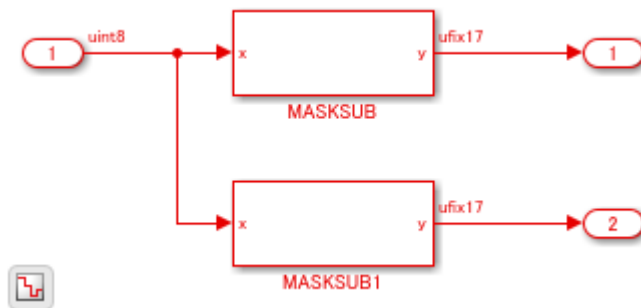
Description

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 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');
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 DOWNT0 0); -- ufix17
         y            : OUT std_logic_vector(16 DOWNT0 0) -- ufix17
         );
  END COMPONENT;

  -- Component Configuration Statements
  FOR ALL : MASKSUB
    USE ENTITY work.MASKSUB(rtl);

  -- Signals
  SIGNAL MASKSUB_out1 : std_logic_vector(16 DOWNT0 0); -- ufix17
  SIGNAL MASKSUBI_out1 : std_logic_vector(16 DOWNT0 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;
```

See Also

Functions

makehdl | makehdltb

More About

- “Generate Parameterized Code for Referenced Models” on page 10-27
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder™” on page 24-9
- “Remove Redundant Logic and Optimize Unconnected Ports in Design” on page 24-114

Design Considerations for Matrices and Vectors

These guidelines recommended 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 20-3.

Modeling Requirements for Matrices

Guideline ID

1.3.1

Severity

Mandatory

Description

HDL Coder™ does not support matrix data types at the DUT interfaces. Before the 2-D matrix signals enter the DUT Subsystem, convert the signals to 1-D vectors by using a Reshape block. Inside the DUT Subsystem, you can convert the vectors back to matrices by using Reshape blocks, and then perform matrix computations. After performing computations, you must convert the matrices back to vector signals at the DUT output interface. Outside the DUT interface, you can convert the vector signals back to matrices.

Modeling Considerations

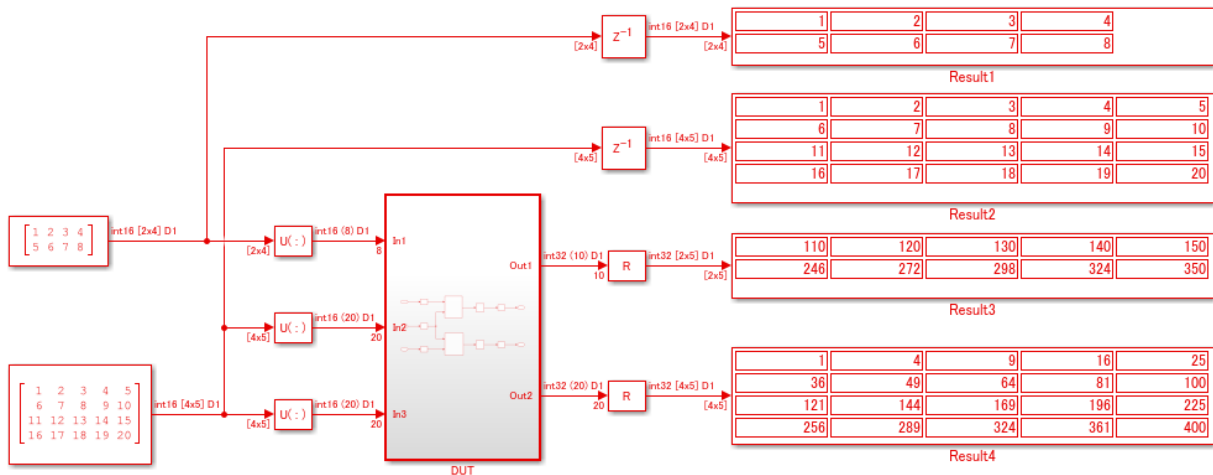
- When you use the Reshape block to convert vectors to 2D matrices, make sure that you specify the right **Output dimensions**.
- When you use the Product block, use the right **Multiplication** mode. By using this mode, you can perform either matrix multiplication or element-wise multiplication. The 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.

- When you extract matrix data, use Selector and Assignment blocks. Make sure that you do not use fixed-point data types for the index inpt ports for the blocks.

Example

This example shows how to use matrix types in HDL Coder™. Open this model `hdlcoder_matrix_multiply`. The model contains a Reshape block that converts the matrix input to a 1-D vector at the DUT Subsystem interface.

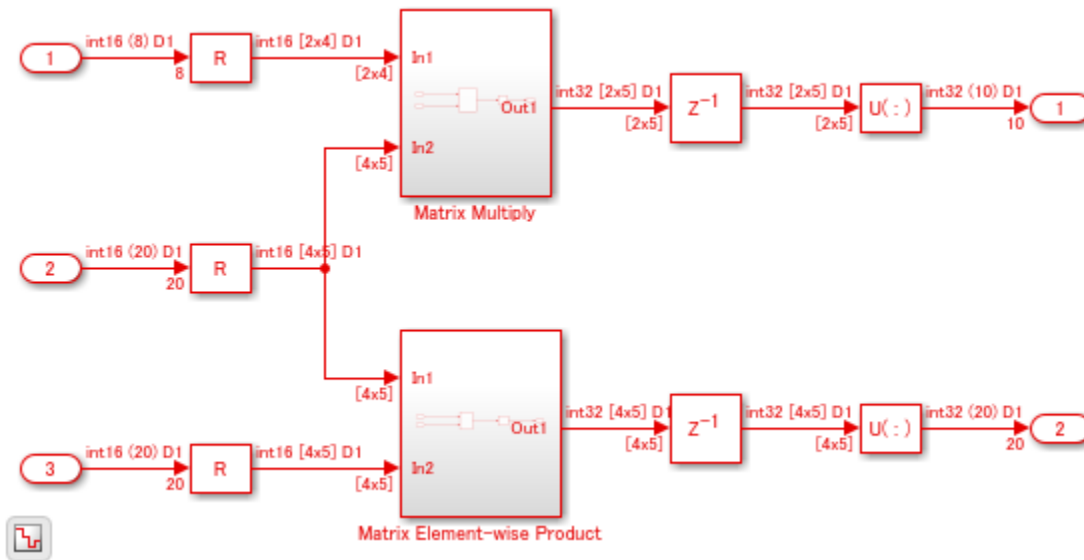
```
open_system('hdlcoder_matrix_multiply')
set_param('hdlcoder_matrix_multiply', 'SimulationCommand', 'update')
sim('hdlcoder_matrix_multiply')
```



Copyright 2018 The MathWorks, Inc.

If you open the DUT Subsystem, you see two subsystems. The Reshape blocks convert the 1-D array back to the 2-by-2 matrices for input to the subsystems. One subsystem uses a Matrix Multiply block and the other subsystem performs element-wise multiplication. The output result is converted back to vectors.

```
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.

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
<p>Delay block with a Delay length greater than 1.</p>	<p>This example illustrates the generated code for a Delay block with a Delay length of 5.</p> <pre> ENTITY Subsystem1 IS PORT(clk : IN std_logic; reset : IN std_logic; enb : IN std_logic; In1 : IN std_logic; -- ufix1 [5] Out1 : OUT std_logic); END Subsystem1; ARCHITECTURE rtl OF Subsystem1 -- Signals SIGNAL Delay_reg : std_logic_vector(0 TO 4); -- ufix1 [5] SIGNAL Delay_out1 : std_logic; </pre>	<p>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.</p> <pre> ENTITY Subsystem1 IS PORT(clk : IN std_logic; reset : IN std_logic; enb : IN std_logic; In1 : IN std_logic; -- ufix1 [5] Out1 : OUT std_logic); END Subsystem1; ARCHITECTURE rtl OF Subsystem1 IS -- Signals SIGNAL Delay_out1 : std_logic; -- SIGNAL Delay1_out1 : std_logic; -- SIGNAL Delay2_out1 : std_logic; -- SIGNAL Delay3_out1 : std_logic; -- SIGNAL Delay4_out1 : std_logic; -- </pre>

Scenario	Problem Example	Workaround
<p>Combining multiple input signals to a vector signal using the Mux block.</p>	<p>This example illustrates the generated code when you use a Mux block to combine 4 input signals.</p> <pre> ENTITY Subsystem IS PORT(In1 : IN std_logic_vector(0 TO 3) Out1 : OUT std_logic_vector(0 TO 3)); END Subsystem; ARCHITECTURE rtl OF Subsystem IS -- Signals SIGNAL Mux_out1 : std_logic_vector(0 TO 3); </pre>	<p>Use a Bit Concat block to combine the input signals. This example illustrates the generated code for this block by concatenating 4 input signals.</p> <pre> ENTITY Subsystem IS PORT(In1 : IN std_logic_vector(0 TO 3) Out1 : OUT std_logic_vector(0 TO 3)); END Subsystem; ARCHITECTURE rtl OF Subsystem IS -- Signals SIGNAL Bit_Concat_out1 : unsigned(3 DOWNTO 0); </pre>
<p>Using a Constant block to generate vector signals.</p>	<p>This example illustrates the generated code when you use a Constant block to generate a vector of 4 scalar boolean signals.</p> <pre> ENTITY Subsystem2 IS PORT(Out1 : OUT std_logic_vector(0 TO 3)); END Subsystem2; ARCHITECTURE rtl OF Subsystem2 IS -- Signals SIGNAL Constant_out1 : std_logic_vector(0 TO 3); </pre>	<p>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.</p> <pre> ENTITY Subsystem2 IS PORT(Out1 : OUT std_logic_vector(0 TO 3)); 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); </pre>

See Also

Functions

makehdl | makehdltb

More About

- “Signal and Data Type Support” on page 10-2
- “RTL Description Techniques” on page 26-25

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.

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. 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.

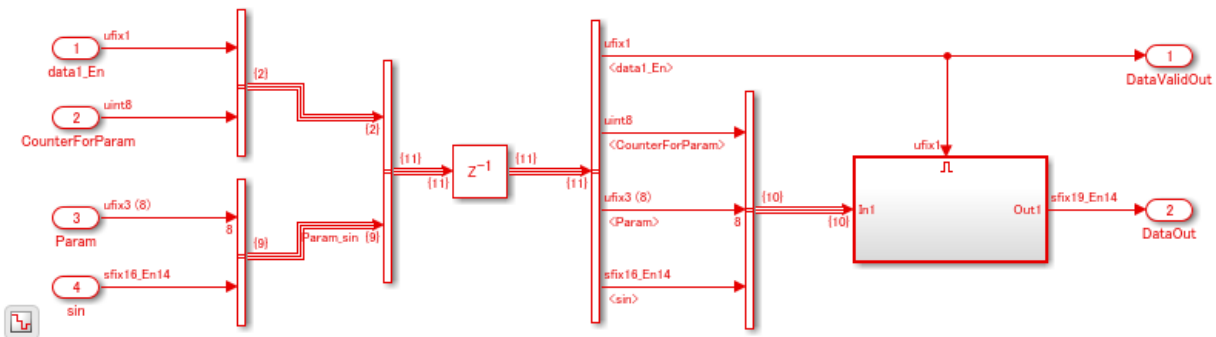
See Signal and Data Type Support for blocks that support HDL code generation with buses.

Create Bus Signals

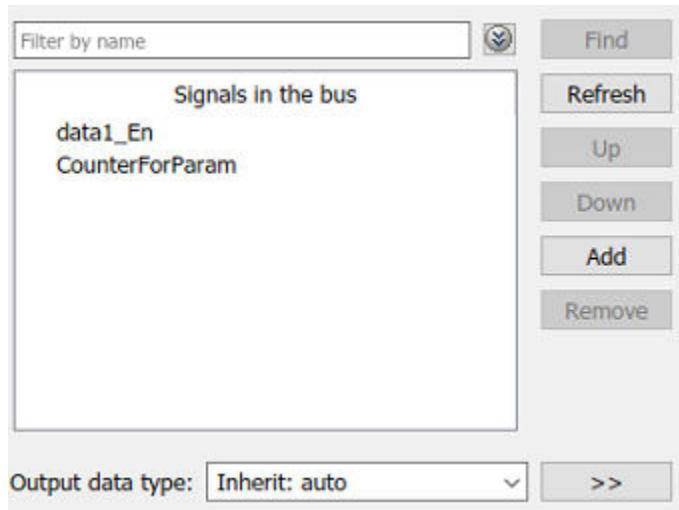
You can create bus signals by using Bus Creator blocks. A Bus Creator block assigns a name to each signal that it creates. You can then refer to signals by name when you search for their sources.

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 **Signals 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 **Signals in the bus** as `signal1` and `Param_sin`.

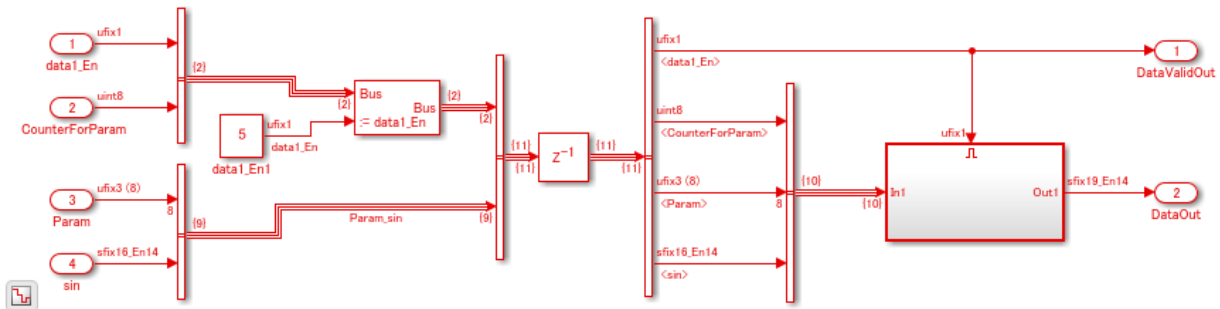
See also Nest Buses.

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 **Signals 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;

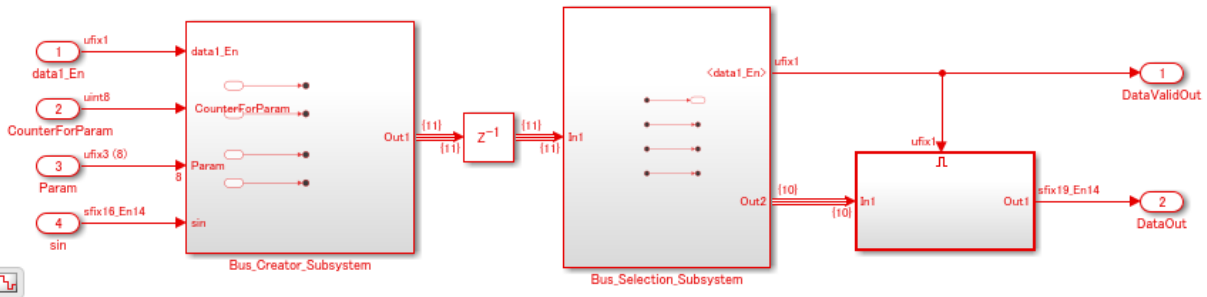
endmodule // HDL_DUT
```

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.

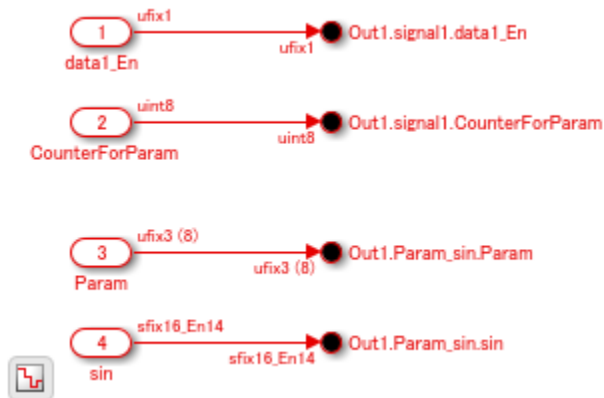
For example, open the model `hdlcoder_bus_nested_simplified`. This model is functionally equivalent to the `hdlcoder_bus_nested` model but is a more simplified version.

```
open_system('hdlcoder_bus_nested_simplified')
set_param('hdlcoder_bus_nested_simplified','SimulationCommand','Update')
open_system('hdlcoder_bus_nested_simplified/HDL_DUT')
```



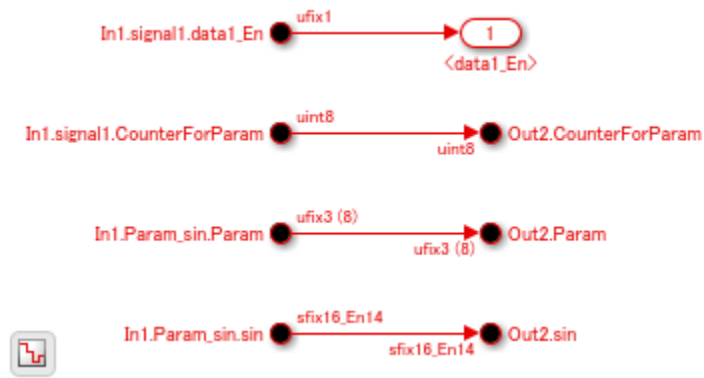
The model has two Subsystems that perform bus creation and bus selection by using Bus Element blocks. The Bus_Creator_Subsystem combines the Outport blocks with the Bus Creator blocks to create Out Bus Element blocks.

```
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 Bus Interfaces](#).

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**.

- Getting Started with Buses
- Convert Virtual Bus to a 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](#).

See Also

Functions

`makehdl` | `makehdltb`

More About

- [“Signal and Data Type Support”](#) on page 10-2
- [“Use Arrays of Buses in Models”](#) (Simulink)
- [“Convert Models to Use Arrays of Buses”](#) (Simulink)

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 20-3.

Use Global Oversampling to Create Frequency-Divided Clock

Guideline ID

1.4.1

Severity

Informative

Description

You can assign a frequency-divided clock rate for HDL code generation to be 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 assign an **Oversampling factor** of 50. You specify the “Oversampling factor” on page 16-18 in the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. To learn more, see “Generate a Global Oversampling Clock” on page 22-9.

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 master 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:

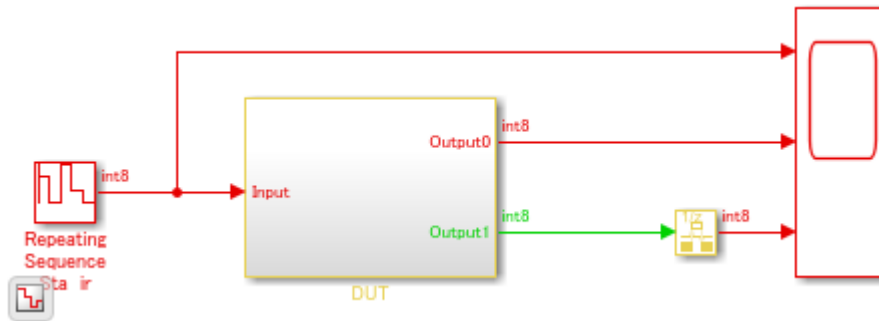
- Divide block with Newton-Raphson implementation.
- Reciprocal block with ReciprocalSqrtBasedNewton implementation.
- Sum of Elements and Product of Elements blocks with Cascade architecture.
- Sqrt with SqrtBasedNewton and Reciprocal Sqrt with ReciprocalRsqrBasedNewton implementation.

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.

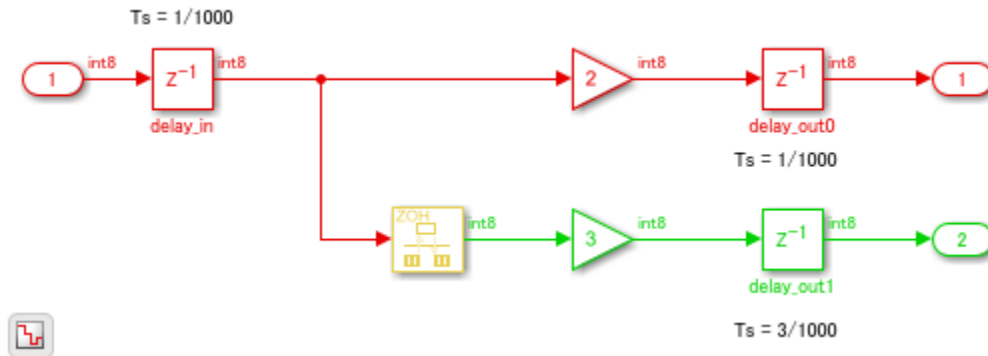
This model illustrates how to create a multirate design by using a Rate Transition block.

```
load_system('hdlcoder_multiclock')
set_param('hdlcoder_multiclock','SimulationCommand','Update')
open_system('hdlcoder_multiclock')
```



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: [0.9000 0 0]
        Annotation: 'D1'
        OwnerBlock: []
        ComponentSampleTimes: [0x0 Simulink.SampleTime]

```

```
sampletime_D2 =
```

```
    SampleTime with properties:
```

```

        Value: [0.0030 0]
        Description: 'Discrete 2'
        ColorRGBValue: [0 0.8200 0]
        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.

Use Dual Rate Dual Port RAM for Noninteger Multiple Sample Times

Guideline ID

1.4.3

Severity

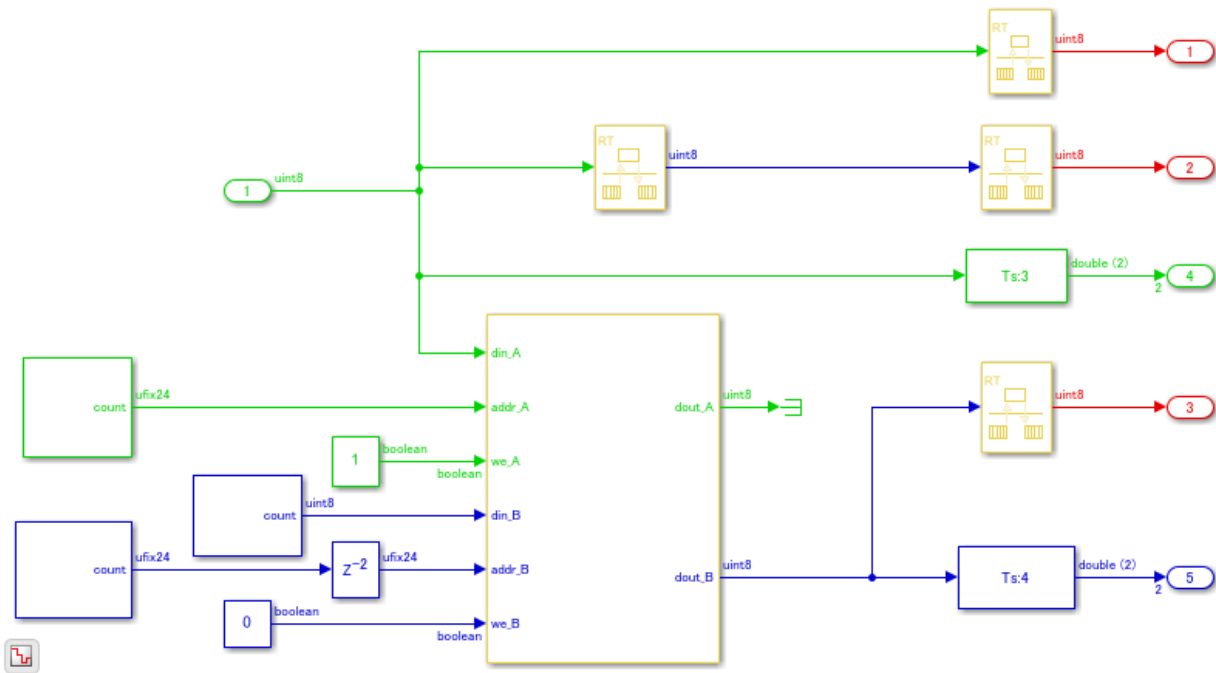
Mandatory

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 block access.

Asynchronous Clock Modeling in HDL Coder

Guideline ID

1.4.4

Severity

Recommended

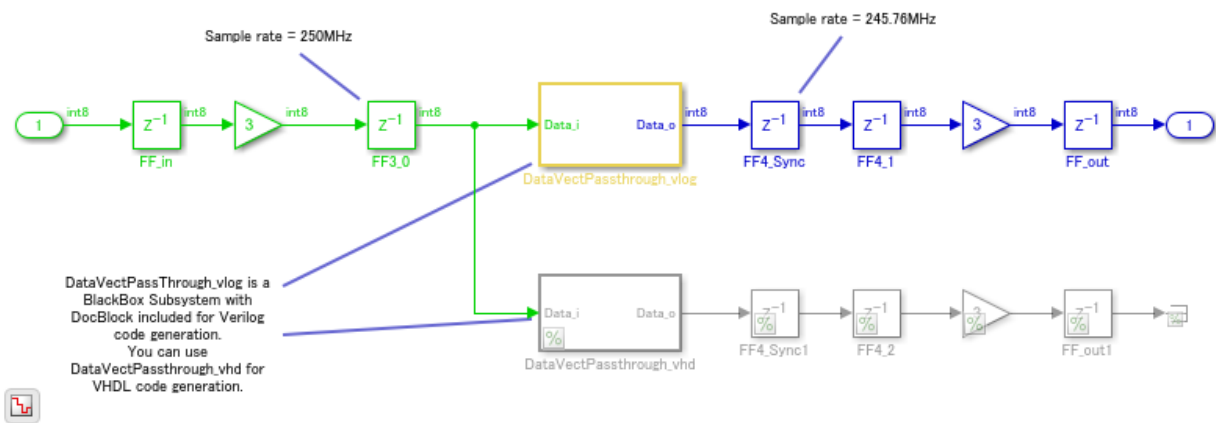
Description

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 Model Checker, run the model check “Check for global reset setting for Xilinx and Altera devices” on page 38-8.

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

Simulink Configuration Parameters

“Timing Controller Settings” on page 16-73 | “Clock Settings and Timing Controller Postfix” on page 16-4

More About

- “Multirate Model Requirements for HDL Code Generation” on page 22-7
- “Code Generation from Multirate Models” on page 22-2
- “Getting Started with the HDL Model Checker” on page 39-2

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 20-3.

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 10-65.

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 10-66.

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, set “Share Adders” on page 14-22 to on.
- Floating-point multipliers, make sure “Share Multipliers” on page 14-24 is set to on.
- Other floating-point resources, set “Share Floating-Point IPs” on page 14-35 to on.

See also “Resource Sharing” on page 24-27.

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 10-77.

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 10-83.

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 | makehdl**l**tb

Simulink Configuration Parameters

“Floating Point IP Library” on page 15-3 | “Native Floating Point” on page 15-5

More About

- “Numeric Considerations with Native Floating-Point” on page 10-69
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-92
- “Verify the Generated Code from Native Floating-Point” on page 10-101

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 20-3.

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. Each row in the table describes a RAM block that is larger in circuit size than the RAM block in the previous row. The generated HDL code for these blocks maps to RAM in most FPGAs.

Block Name	Recommended Usage
Single Port RAM System	<p>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.</p> <p>The block implementation uses a MATLAB System block that uses the <code>hdl.RAM System</code> 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.</p> <p>The block does not support boolean inputs. Cast <code>boolean</code> types to <code>ufix1</code> for input to the block.</p>
Simple Dual Port RAM System	<p>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.</p> <p>The block implementation uses a MATLAB System block that uses the <code>hdl.RAM System</code> 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.</p> <p>The block does not support boolean inputs. Cast <code>boolean</code> types to <code>ufix1</code> for input to the block.</p>

Block Name	Recommended Usage
Dual Port RAM System	<p>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.</p> <p>The block implementation uses a MATLAB System block that uses the <code>hdl.RAM System</code> 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.</p> <p>The block does not support boolean inputs. Cast <code>boolean</code> types to <code>ufix1</code> for input to the block.</p>
Dual Rate Dual Port RAM	<p>This block does not have an equivalent System object-based implementation.</p> <p>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.</p> <p>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.</p>

Block Name	Recommended Usage
HDL FIFO	<p>The HDL FIFO block stores a sequence of samples in a first in, first out (FIFO) register.</p> <p>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 $1/N$ such that N is a positive integer. For example:</p> <ul style="list-style-type: none"> • If you specify the ratio as 2, the output sample time is twice the input sample time. The outputs run slower than the input. • If you specify the ratio as $1/2$, the output sample time is half the input sample time. The outputs run faster than the input. <p>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.</p> <p>The input and output rates of the FIFO block are synchronous to each other. For an example of asynchronous FIFO modeling by using the HDL FIFO block, open the model <code>hdlcoder_asynchronous_fifo</code>.</p> <pre>open_system('hdlcoder_asynchronous_fifo')</pre>

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®”

More About

- “HDL Code Generation from hdl.RAM System Object”
- “Map Matrices to Block RAMs to Reduce Area”

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 20-3.

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 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.

Block/Function Name	Parameter/ Operation	Verilog Code Equivalent	VHDL code equivalent	Comments
	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.
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.

Block/Function Name	Parameter/Operation	Verilog Code Equivalent	VHDL code equivalent	Comments
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 shift	>>	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

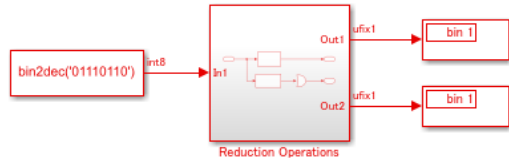
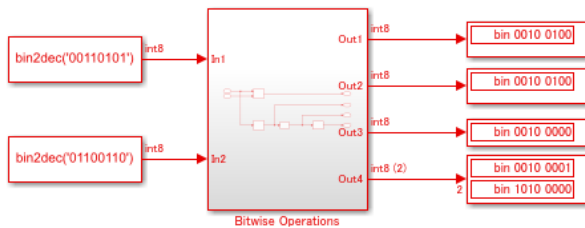
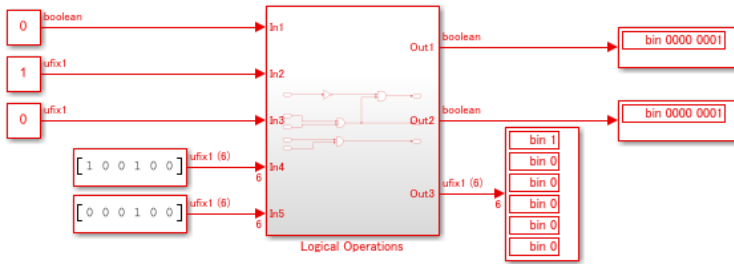
Severity

Informative

Description

To learn how you can use 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')
```

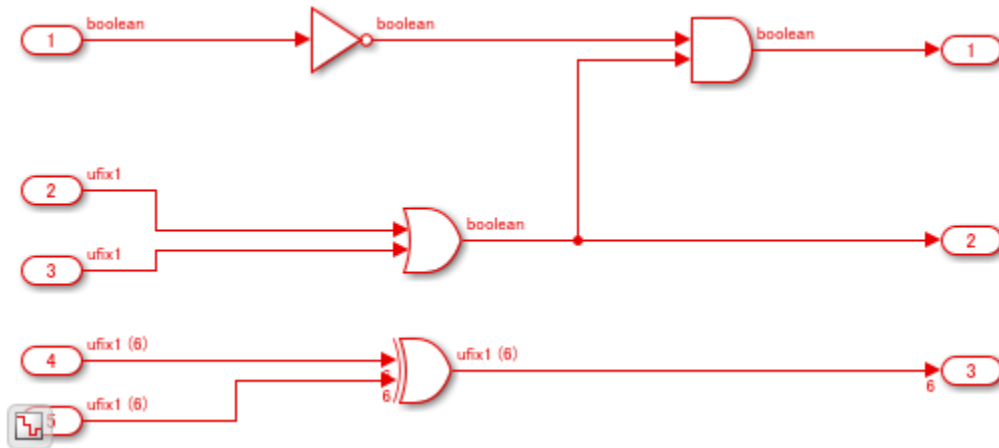


For single-bit operations that use Boolean or ufix1 data types, use a Logical Operator block. To view the operation as a logical circuit symbol, in the Block Parameters dialog box of the block, specify the **Icon shape** as Distinctive. You can also input vectors that have Boolean or ufix1 data types to the block.

Note: Boolean and ufix1 are different data types. Mixing these data types within the same model or using them interchangeably is not recommended. For more information, see Simulink Data Type Considerations.

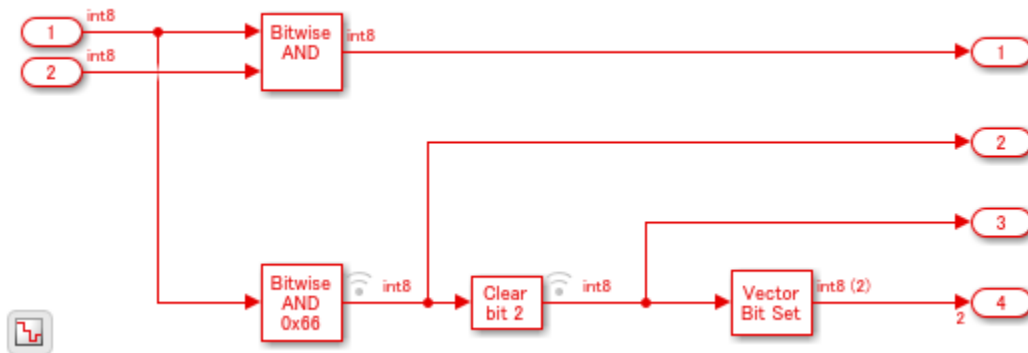
For an example of using the block, open the Logical Operations Subsystem.

```
open_system('hdlcoder_logical_bitwise_operations/Logical Operations')
```



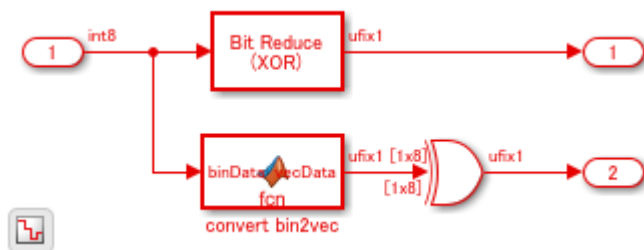
For bitwise operations on two or more bits that use integer or fixed-point data types, use the Bitwise Operator block. For an example, double-click the Bitwise Operations Subsystem.

```
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 Data Type 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 38-29.

See Also

Functions

makehdl

Blocks

Bitwise Operator | Extract Bits

More About

- “Bitwise Operations in MATLAB for HDL Code Generation” on page 2-41

Generate FPGA Block RAM from Lookup Tables

You can follow these guidelines to learn about how you can map the lookup table blocks to RAM to save area on the target FPGA device.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see HDL Modeling Guidelines Severity Levels.

Guideline ID

2.3.1

Severity

Strongly Recommended

Description

Follow this guideline to learn how to map a Lookup Table block in your design to a Block RAM on the FPGA. The severity level of the guideline indicates the level of compliance requirements. To learn more, see HDL Modeling Guidelines Severity Levels.

To map lookup tables to a block RAM, you can use the adaptive pipelining optimization. This optimization is enabled 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 adaptive pipelining optimization, you must:

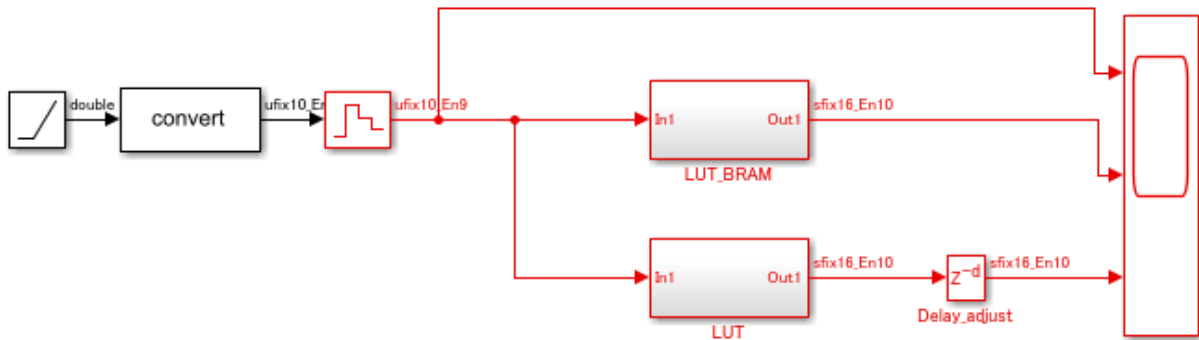
- Make sure that **AdaptivePipelining** is enabled on the model.
- Specify the synthesis tool.

To learn more about adaptive pipelining, see Adaptive Pipelining.

If you do not want to use the adaptive pipelining optimization for the entire design, you can selectively enable this optimization for certain Subsystems in your design, or create the same modeling pattern in your design that is 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.

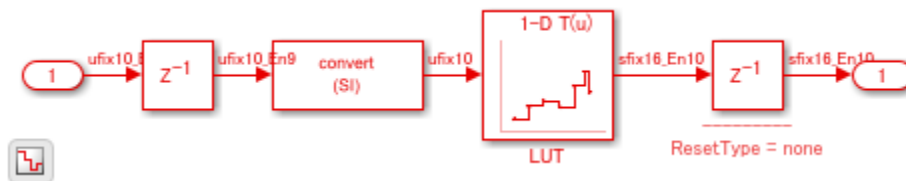
The adaptive pipelining optimization is disabled on this model.

```
hdlget_param('hdlcoder_LUT_BRAM_mapping','AdaptivePipelining')
```

```
ans =  
      'off'
```

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, it efficiently maps to Block RAM. This figure displays the synthesis results for the LUT_BRAM Subsystem.

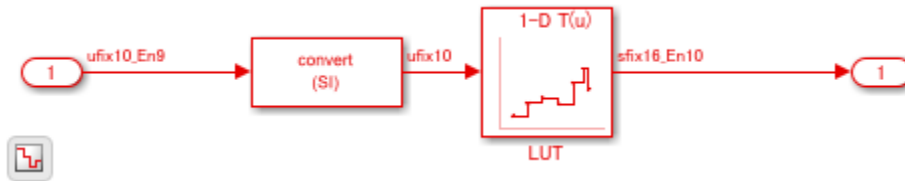
Parsed resource report file: [LUT_OK_utilization_synth.rpt.](#)

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')
```



As adaptive pipelining is disabled on the model, synthesizing this Subsystem maps the logic to LUTs instead of utilizing the Block RAMs. This figure displays the synthesis results for the LUT Subsystem.

Parsed resource report file: [LUT_utilization_synth.rpt.](#)

Resource summary	
Resource	Usage
Slice LUTs	138
Slice Registers	0
DSPs	0
Block RAM Tile	0

Parsed timing report file: [timing_post_map.rpt.](#)

See Also

Functions

makehdl

Simulink Configuration Parameters

“RAM Mapping” on page 14-5 | “MapPersistentVarsToRAM” on page 21-20 | “Adaptive pipelining” on page 14-18

More About

- “Adaptive Pipelining” on page 24-68
- “RAM Mapping for Simulink Models” on page 24-43
- “RAM Mapping With the MATLAB Function Block” on page 24-44

Usage of Different Subsystem Types

You can follow these guidelines to learn how to use different types of subsystems in your design and model your algorithm hierarchically. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 20-3.

Virtual Subsystem: Use as Top-Level 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.

If you use a non-top-level DUT as a virtual subsystem, the conversion to model reference may change the subsystem semantics because the execution sequence of a referenced model is equivalent to an Atomic Subsystem. To preserve subsystem semantics, specify the DUT as a nonvirtual subsystem before HDL code generation and verification. Nonvirtual subsystem types include Atomic Subsystem, model reference, Variant Subsystem, and a variant model. You can learn about these subsystem types in the preceding sections.

To determine whether a subsystem is virtual, you can use the `get_param` function with the parameter `IsSubsystemVirtual`. For example:

```
get_param('sfir_fixed/symmetric_fir', 'IsSubsystemVirtual')
```

Atomic Subsystem: Generate Reusable HDL Files

Guideline ID

2.4.2

Severity

Recommended

Description

You can specify the DUT as an Atomic Subsystem. To specify a Subsystem as an Atomic Subsystem, in the Block Parameters dialog box of that Subsystem, select **Treat as atomic unit**. Use atomic subsystems to generate a single HDL file for identical instances of the subsystems that you use at lower levels of a hierarchy. To learn more, see “Generate Reusable Code for Atomic Subsystems” on page 27-20.

To enable resource sharing on a subsystem unit, specify all subsystems that you want to share as atomic subsystems. To learn more, see “General Considerations for Sharing of Subsystems” on page 20-119.

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. At simulation time, a variant control decides which among 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 simulation time.

- 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 unique to the active configuration that is chosen at code generation time.
- 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. In this case, `Recip` is 1, and the `Recip` Subsystem becomes active during simulation.

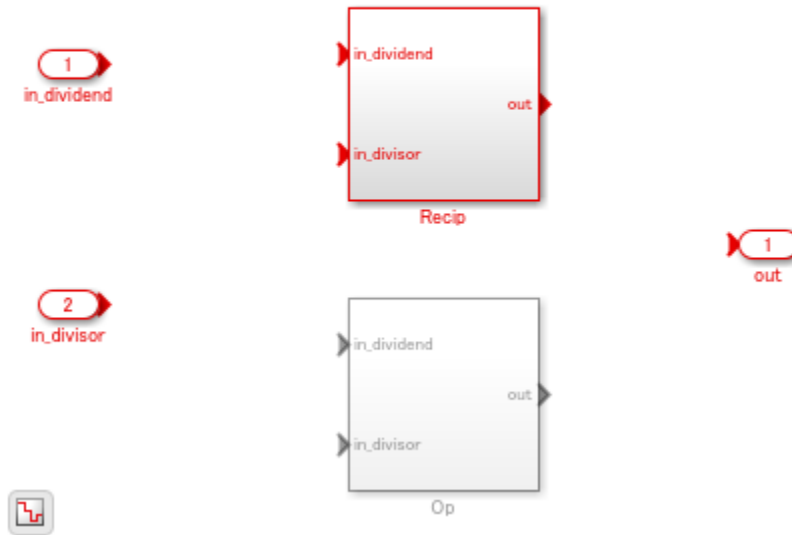
```
load_system('hdlcoder_variant_subsystem_design')
set_param('hdlcoder_variant_subsystem_design','SimulationCommand','Update')
open_system('hdlcoder_variant_subsystem_design/DUT/Divide')
```

```
variantRecip =
Simulink.Variant
    Condition: 'Recip == 1'
```

```
variantOp =
Simulink.Variant
    Condition: 'Recip == 0'
```

```
Recip =
    1
```

- 1) Only subsystems can be added as variant choices at this level
- 2) Blocks cannot be connected at this level as connectivity is automatically determined at simulation, based on the active variant



To generate HDL code, run this command:

```
makehdl('hdlcoder_variant_subsystem_design/DUT')
```

You see that a HDL file with the name `Recip.vhd` is generated because the code is generated for the `Recip` Subsystem which is active at code generation time.

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](#).

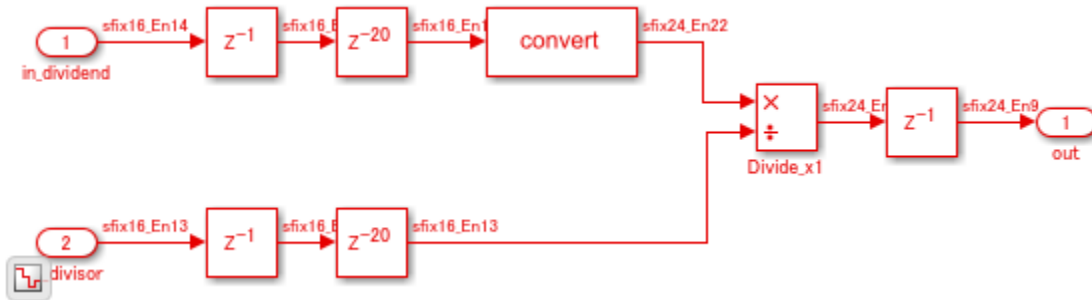
For an example, open the model `hdlcoder_divide_parentmodel.slx`. When you double-click the DUT Subsystem, you see a Model block that references the model `hdlcoder_divide_referencedmodel`.

```
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')
```



To generate HDL code, enter this command:

```
makehdl('hdlcoder_divide_parentmodel/DUT')
```

For more information, see Model Referencing for HDL Code Generation.

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.
- Make sure that the initial condition of the Triggered Subsystem must be zero.
- You can add unit delays to the output signals of the Triggered Subsystem. The unit delays prevent the code generator 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 22-15.

For other preferences when configuring the Triggered Subsystem block for HDL code generation, see “HDL Code Generation” (Simulink) 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.
- Make sure that the initial condition of the Enabled Subsystem is zero.
- You can add unit delays to the output signals of the Enabled Subsystem. The unit delays prevent the code generator from inserting additional bypass registers in the HDL code.
- You can add a State Control block in **Synchronous** mode inside the Enabled Subsystem. The State Control block converts the Enabled Subsystem block to an Enabled Synchronous Subsystem block. This block generates more efficient and hardware-friendly HDL code. To learn more, see “Synchronous Subsystem Behavior with the State Control Block” on page 27-63.

For other preferences when configuring the Enabled Subsystem block for HDL code generation, see “HDL Code Generation” (Simulink) on the Enabled Subsystem page.

See Also

Functions

makehdl

Simulink Configuration Parameters

“Use trigger signal as clock” on page 16-49

Related Examples

- “Resettable Subsystem Support in HDL Coder™”

More About

- “Using Enabled and Triggered Subsystems” (Simulink)

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 20-3.

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	Yes	No	When you use this block, consider the bypass register in the HDL code and the impact on hardware resource usage.
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 Repeat and Rate Transition blocks which do not insert a bypass register, if the input and output clocks are not synchronous to each other, you can insert one sample delay in your design by adding a Delay block to the model.

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.

It is recommended that you use the Rate Transition block because you can leave the block parameters to the default settings for HDL code generation. You use the **Initial Condition** parameter of the block when reducing the sample rate. This parameter setting is not used when the code generator increases the sample rate.

Use Discrete and Finite Sample Time for Constant Block

Guideline ID

2.5.2

Severity

Mandatory

Description

By default, the sample time of a Constant block is `inf`. When you connect a Constant block with sample time of `inf` to other blocks in your design, it hinders speed and area optimizations. Optimizations such as retiming, sharing, and streaming use the clock rate information to improve the speed and area of your design.

When you use the Constant block, set the sample time of the blocks to `-1`. To identify Constant blocks that have infinite sample time in your design, in the Simulink model window, select **Display > Sample Time > Colors**. The Sample Time Legend then displays the Constant blocks that have `Inf` sample time.

You can identify and change the sample time of all Constant 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 20-38.
- Run the check “Check for infinite and continuous sample time sources” on page 38-17 in the HDL Model Checker. 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.

See Also

Functions

`makehdl`

More About

- “Multirate Model Requirements for HDL Code Generation” on page 22-7
- “Code Generation from Multirate Models” on page 22-2

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 20-3.

Use Boolean for Logical Data and Ufix1 for Numerical Data

Guideline ID

2.6.1

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

2.6.2

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

2.6.3

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 must use monotonically increasing enumeration values. For example, see this code:

```
classdef BasicColors < Simulink.IntEnumType
enumeration
    Red(0)
    Yellow(1)
    Blue(2)
end
```

```
methods (Static)
    function retVal = getDefaultVal()
        retVal = BasicColors.Blue;
    end
end
end
```

- You cannot perform arithmetic operations such as `*`, `/`, `-`, and `+` with enumeration values.
- You cannot perform comparison operations such as `>`, `<`, `>=`, `<=`, `==`, and `~=` with enumeration values. You can perform a `<>` operation or a conditional branch such as `if` or `switch`.

See Also

Functions

`makehdl`

More About

- “Signal and Data Type Support” on page 10-2

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 24-27.

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 20-3.

Resource Sharing of Add Blocks

Guideline ID

3.1.1

Severity

Recommended

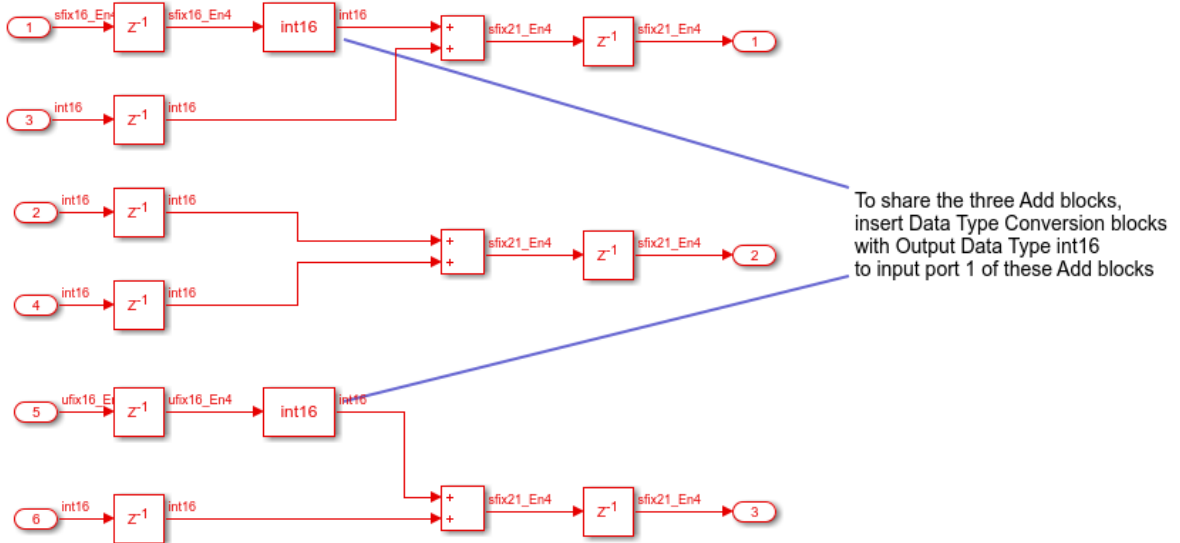
Description

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 21-31 for Add blocks with vector inputs or outputs.
- Specify the “SharingFactor” on page 21-29 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 `fcscd`.
 - **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 21-31 for Gain blocks with vector inputs or outputs.
- Specify the “SharingFactor” on page 21-29 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 21-31 for the subsystems that contain Product blocks with vector inputs or outputs.
- Specify the “SharingFactor” on page 21-29 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 24-9.
- 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 21-29.

See Also

Simulink Configuration Parameters

Resource Sharing of Adders and Multipliers | Resource Sharing of Multiply-Add and Other Blocks

Related Examples

- “Resource Sharing of Multipliers to Reduce Area”
- “Resource Sharing For Area Optimization”
- “Single-rate Resource Sharing Architecture”

More About

- “Streaming” on page 24-23
- “Resource Sharing of Subsystems and Floating-Point IPs” on page 20-119

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 24-27.

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 20-3.

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 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 subsystems without enable semantics.
- Specify a “SharingFactor” on page 21-29 that is greater than or equal to the number of subsystems that you want to share.

For example, if you have 10 atomic 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 24-27.

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 24-89.

Sharing of Atomic Subsystems

Guideline ID

3.1.7

Severity

Recommended

Description

HDL Coder can share Atomic Subsystem blocks that have the same Simulink checksum and the same HDL block properties.

To share Atomic 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 atomic subsystems inside enabled subsystems with synchronous semantics is not supported. To share resources, use enabled subsystems with classic semantics.

You cannot share atomic 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)
- CORDIC architecture
- 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 24-34.

HDL Coder can share Atomic Subsystem blocks that have the same Simulink checksum and the same HDL block properties.

If you want to share Atomic 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.

You cannot share atomic 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)
- CORDIC architecture
- 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 24-34.

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**.

See Also

Simulink Configuration Parameters

Resource Sharing of Adders and Multipliers | Resource Sharing of Multiply-Add and Other Blocks

Related Examples

- “Resource Sharing of Multipliers to Reduce Area”
- “Resource Sharing For Area Optimization”
- “Single-rate Resource Sharing Architecture”

More About

- “Resource Sharing” on page 24-27
- “Streaming” on page 24-23

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 20-3.

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

In addition, for designs with multiple hierarchies, to improve opportunities for clock-rate 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 24-63.

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.

To use this optimization for a Subsystem, set the **DistributedPipelining** HDL block property set to on.

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” on page 21-9: Make sure that the total number of delays that are inserted including anyinput and output pipelining that you specify is greater than or equal to the value that you specify for **ConstrainedOutputPipeline** on the Subsystem.
- “Hierarchical distributed pipelining” on page 14-12: Select this option if you want to apply the distributed pipelining optimization across multiple subsystem hierarchy. Make sure that the top-level Subsystem and each subsystem in the hierarchy has the **DistributedPipelining** HDL block property set to on.

Note If you cannot enable **DistributedPipelining** on the top-level Subsystem, you can enable **FlattenHierarchy**, which enables pipelining with other blocks at a lower model hierarchy.

- “Clock Rate Pipelining” on page 14-15: 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” on page 14-16: 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.
- “Preserve design delays” on page 14-20: Select 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.
- “Distributed pipelining priority” on page 14-13: 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 for which you want to apply the optimization must meet these requirements:

- Make sure that the Subsystem that you apply this optimization on does not contain any feedback loops.
- Use blocks that are supported for distributed pipelining. For a list of unsupported blocks, see “Limitations of Distributed Pipelining” on page 24-52. As a workaround:
 - Place some of the unsupported blocks such as Dot Product inside another Subsystem that does not have distributed pipelining enabled.
 - Change the **Distributed pipelining priority** to **Performance** for certain blocks such as Enabled Subsystem.
- The **Sample Time** of the blocks must be discrete. If you have blocks with **Sample Time** set to **Inf**, change them to **-1**. To identify and change the sample time programmatically, see “Change Block Parameters by Using `find_system` and `set_param`” on page 20-43.
- Remove any input ports on Scope blocks to avoid generation of infinite sample time.

See Also

Simulink Configuration Parameters Distributed Pipelining

Related Examples

- “Distributed Pipelining: Speed Optimization”

- “Distributed Pipelining for Clock Speed Optimization”

More About

- “Distributed Pipelining” on page 24-49

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 effectiely 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 20-3.

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

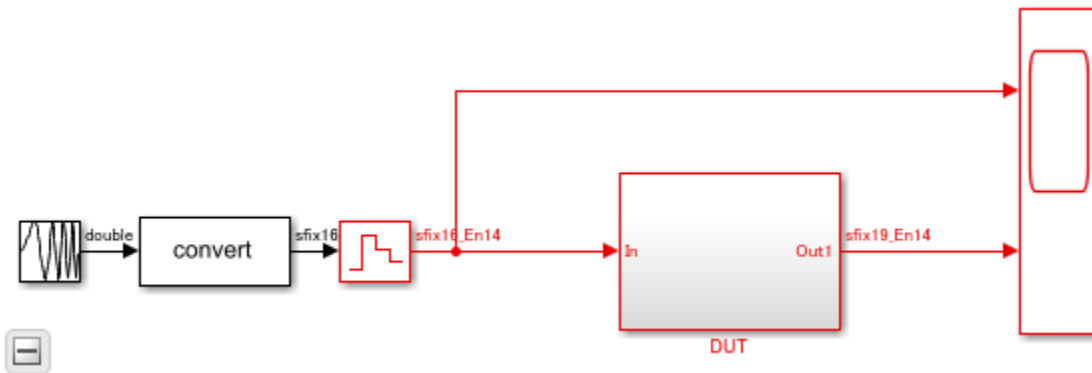
- 1 In the HDL Block Properties for the blocks, set **Architecture** to:
 - Tree for adders, multipliers, and MinMax blocks. Distributed pipeline register insertion does not support Linear and Cascade architectures.

- 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')
```



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)

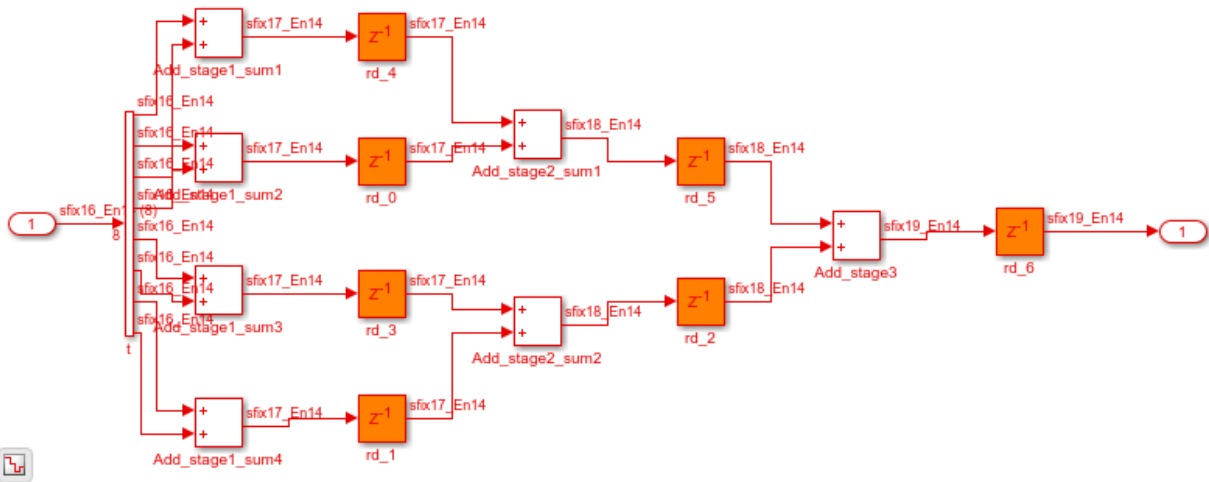
Registers	Count
17-bit	4
18-bit	2
19-bit	1

Generated Model

Generated model after the transformation: [gm_hdlcoder_distributed_pipelining_so_e](#)

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_so_e')
set_param('gm_hdlcoder_distributed_pipelining_so_e','SimulationCommand','Update')
open_system('gm_hdlcoder_distributed_pipelining_so_e/DUT/Add')
```

See Also

Simulink Configuration Parameters Distributed Pipelining

Related Examples

- “Distributed Pipelining: Speed Optimization”
- “Distributed Pipelining for Clock Speed Optimization”

More About

- “Distributed Pipelining” on page 24-49

Supported Blocks Library and Block Properties

- “View HDL-Specific Block Documentation” on page 21-2
- “HDL Block Properties: General” on page 21-3
- “HDL Block Properties: Native Floating Point” on page 21-39
- “HDL Filter Block Properties” on page 21-51
- “HDL Filter Architectures” on page 21-60
- “Distributed Arithmetic for HDL Filters” on page 21-68
- “Set and View HDL Model and Block Parameters” on page 21-71
- “Pass through, No HDL, and Cascade Implementations” on page 21-76
- “Build a ROM Block with Simulink Blocks” on page 21-77
- “Wireless Communications Design for FPGAs and ASICs” on page 21-78

View HDL-Specific Block Documentation

To view HDL-specific documentation for a block in your model, 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. You can also find HDL-specific block documentation in “Model Design”.

See Also

hdllib

Related Examples

- “Set and View HDL Model and Block Parameters” on page 21-71
- “Show Blocks Supported for HDL Code Generation” on page 25-23

More About

- “HDL Block Properties: General” on page 21-3
- “HDL Filter Block Properties” on page 21-51

HDL Block Properties: General

In this section...

“Overview” on page 21-4
“AdaptivePipelining” on page 21-4
“BalanceDelays” on page 21-5
“ClockRatePipelining” on page 21-6
“CodingStyle” on page 21-7
“ConstMultiplierOptimization” on page 21-8
“ConstrainedOutputPipeline” on page 21-9
“DistributedPipelining” on page 21-10
“DotProductStrategy” on page 21-11
“DSPStyle” on page 21-12
“FlattenHierarchy” on page 21-15
“InputPipeline” on page 21-16
“InstantiateFunctions” on page 21-17
“InstantiateStages” on page 21-18
“LoopOptimization” on page 21-18
“LUTRegisterResetType” on page 21-19
“MapPersistentVarsToRAM” on page 21-20
“OutputPipeline” on page 21-22
“RAMDirective” on page 21-23
“ResetType” on page 21-26
“SerialPartition” on page 21-28
“SharingFactor” on page 21-29
“SoftReset” on page 21-29
“StreamingFactor” on page 21-31
“UseMatrixTypesInHDL” on page 21-31
“UsePipelines” on page 21-33
“UseRAM” on page 21-34

In this section...

“VariablesToPipeline” on page 21-38

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 21-71 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.

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” on page 14-18.

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.
- 2 Select **HDL Code > HDL Block Properties** .
- 3 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`.

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 disable delay balancing for any subsystem within a model, you must set the model-level delay balancing parameter, `BalanceDelays`, to 'off'.

To learn how to set model-level delay balancing, see “Balance delays” on page 14-3.

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 the clock-rate pipelining parameter, `ClockRatePipelining`, to 'off' for that subsystem.

To learn how to set model-level clock-rate pipelining, see “Clock Rate Pipelining” on page 14-15.

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** For **ClockRatePipelining**, select **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 `hdlset_param`.

CodingStyle

When you use Multiport Switch blocks, use the `CodingStyle` parameter to specify whether you want to generate HDL code with if-else or 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
'if-else' (Default)	Generate if-else statements in the Verilog code or when-else statements in the VHDL code for a Multiport Switch block.
'case'	Generate case statements in the Verilog 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 **if-else** or **case**.

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'
```

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 24-61

DistributedPipelining

The `DistributedPipelining` parameter enables pipeline register distribution, a speed optimization that enables you to increase your clock speed by reducing your critical path.

The following table shows the effect of the `DistributedPipelining` and `OutputPipeline` parameters.

DistributedPipelining	OutputPipeline, nStages	Result
'off' (default)	Unspecified (<i>nStages</i> defaults to 0)	HDL Coder does not insert pipeline registers.
	<i>nStages</i> > 0	The coder inserts <i>nStages</i> output registers at the output of the subsystem, MATLAB Function block, or Stateflow chart.
'on'	Unspecified (<i>nStages</i> defaults to 0)	The coder does not insert pipeline registers. <code>DistributedPipelining</code> has no effect.
	<i>nStages</i> > 0	The coder distributes <i>nStages</i> registers inside the subsystem, MATLAB Function block, or Stateflow chart, based on critical path analysis.

To achieve further optimization of code generated with distributed pipelining, perform retiming during RTL synthesis, if possible.

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)” on page 18-23
- IgnoreDataChecking

See Also

- “Distributed Pipelining” on page 24-49
- “Specify Distributed Pipelining” on page 24-52
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-39

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.
'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_dsp48` 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_dsp48 = "yes") wire signed [32:0] m4_out1;
	'VHDL'	attribute multstyle : string ; attribute multstyle of m4_out1 : signal is "dsp" ;	attribute use_dsp48 : string ; attribute use_dsp48 of m4_out1 : signal is "yes" ;
'off'	'Verilog'	(* multstyle = "logic") wire signed [32:0] m4_out1;	(* use_dsp48 = "no" *) wire signed [32:0] m4_out1;

DSPStyle Value	TargetLanguage Value	SynthesisTool Value	
		'Altera Quartus II'	'Xilinx ISE' 'Xilinx Vivado'
	'VHDL'	attribute multstyle : string ; attribute multstyle of m4_out1 : signal is "logic" ;	attribute use_dsp48 : string ; attribute use_dsp48 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” on page 16-70.

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 black box implementation or model reference.

- A Triggered Subsystem when “Use trigger signal as clock” on page 16-49 is enabled.
- A masked subsystem that contains any of the following:
 - Bus.
 - Enumerated data type.
 - Lookup table blocks: 1-D Lookup Table, 2-D Lookup Table, Cosine HDL Optimized, Direct LookupTable (n-D), Prelookup, Sine HDL Optimized, n-D Lookup Table.
 - MATLAB System block.
 - Stateflow blocks: Chart, State Transition Table, Sequence Viewer.
 - Blocks with a pass-through or no-op implementation. See “Pass through, No HDL, and Cascade Implementations” on page 21-76.

Note This option removes subsystem boundaries before code generation. It does not necessarily generate HDL code with a completely flat hierarchy.

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.

The following code specifies an input pipeline depth of two stages for each Sum block in the model:

```
sblocks = find_system(gcf, '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” on page 16-33.

InstantiateFunctions

For the MATLAB Function block, you can use the **InstantiateFunctions** parameter to generate a VHDL entity or Verilog 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 or Verilog 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.

If you enable `InstantiateFunctions`, `UseMatrixTypesInHDL` has no effect.

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 24-89.

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_mlfm`:

```
hdlset_param('my_mlfm', '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.

LUTRegisterResetType Value	Description
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:

- NCO HDL Optimized
- 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 21-20.

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 the following conditions are true:

- Each read or write access is for a single element only. For example, submatrix access and array copies are not allowed.
- 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 allowed.
- 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 the following code, r1 does not map to RAM:

```

if (mod(i,2) > 0)
    a = r1(u);
else
    r1(i) = u;
end

```

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 follows:

```

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;

```

- `RAMSize` is greater than or equal to the `RAMMappingThreshold` value. `RAMSize` is the product `NumElements * WordLength * Complexity`.
 - `NumElements` is the number of elements in the array.
 - `WordLength` is the number of bits that represent the data type of the array.
 - `Complexity` is 2 for arrays with a complex base type; 1 otherwise.

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 (bits)** section in “RAM Mapping” on page 14-5.

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 24-44.

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.

The following code specifies an output pipeline depth of two stages for each Sum block in the model:

```
sblocks = find_system(gcf, '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” on page 16-33.

See also “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-39.

RAMDirective

RAMDirective lets you specify whether you want to map the RAM blocks in your Simulink model to distributed RAMs, block RAMs, or UltraRAM memory. When you select 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.

RAMDirective Value	Description
none (default)	Do not generate the <code>ramstyle</code> attribute in the HDL code. The synthesis tool determines the type of inferred RAM for mapping the RAM blocks in your model.
distributed	<p>Generate HDL attribute for mapping the RAM blocks in your model to distributed RAMs. Distributed RAMs are constructed with LUT. These RAMs are faster but occupy a larger number of LUT slices on the FPGA.</p> <p>This VHDL code shows the <code>ramstyle</code> attribute set to <code>distributed</code>:</p> <pre data-bbox="555 904 1317 961">attribute ram_style: string; attribute ram_style of ram : signal is "distributed";</pre> <p>This Verilog code shows the <code>ramstyle</code> attribute set to <code>distributed</code>:</p> <pre data-bbox="555 1078 985 1104">(* ram_style = "distributed" *)</pre>

RAMDirective Value	Description
block	<p>Generate HDL attribute for mapping the RAM blocks in your model to block RAMs. A block RAM is a dedicated memory unit on the FPGA device. The number of block RAMs available depends on the FPGA device that you are deploying the HDL code to. Sizes of block RAMs can be 4kb, 8kb, 16kb, and 32kb.</p> <p>To map your RAM blocks to block RAM:</p> <ul style="list-style-type: none"> Specify the synthesis tool. You must target a Xilinx device that contains block RAM resources. <hr/> <p>Note If the target device does not contain block RAMs, the synthesis tool ignores this attribute and might infer the RAM as distributed RAMs or LUT slices.</p> <hr/> <ul style="list-style-type: none"> Enter a target frequency greater than zero. <p>This VHDL code shows the <code>ramstyle</code> attribute set to <code>block</code>:</p> <pre>attribute ram_style: string; attribute ram_style of ram : signal is "block";</pre> <p>This Verilog code shows the <code>ramstyle</code> attribute set to <code>block</code>:</p> <pre>(* ram_style = "block" *)</pre>

RAMDirective Value	Description
ultra	<p>Generate HDL attribute for mapping the RAM blocks in your model to UltraRAM memory. An UltraRAM is a dedicated memory block on the target FPGA. The number of UltraRAM memory units available depends on the FPGA device that you are deploying the HDL code to. UltraRAM units are larger than block RAMs and can be as large as 500Mb in size.</p> <p>To map your RAM blocks to UltraRAM:</p> <ul style="list-style-type: none"> Specify Xilinx Vivado as the synthesis tool. You must target a Xilinx device that contains UltraRAM resources, such as Virtex® UltraScale+™. <hr/> <p>Note If the target device does not contain UltraRAM memory, the synthesis tool ignores this attribute and might infer the RAM as distributed RAMs or LUT slices. To map the RAM blocks to block RAMs instead, set RAMDirective to block.</p> <hr/> <ul style="list-style-type: none"> Enter a target frequency greater than zero. The RAM blocks in your design must follow a fixed-read behavior and have a single clock interface. In the HDL RAMs library, except for Dual Port RAM and Dual Rate Dual Port RAM blocks, you can map all other RAM blocks to UltraRAM. The RAM blocks must not have an initial value specified. When you use the RAM System blocks such as Single Port RAM System, Specify the RAM initial value must be set to 0. On device reset, all memory locations in the UltraRAM are initialized to zero. <p>This VHDL code shows the ramstyle attribute set to ultra:</p> <pre>attribute ram_style: string; attribute ram_style of ram : signal is "ultra";</pre> <p>This Verilog code shows the ramstyle attribute set to ultra:</p> <pre>(* ram_style = "ultra" *)</pre>

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 from the HDL Block Properties dialog box:

- 1 Right-click the RAM block.
- 2 Select **HDL Code > HDL Block Properties** .
- 3 For **RAMDirective**, select **none**, **distributed**, **block**, or **ultra**.

Note For the Dual Port RAM block, you cannot specify **ultra** as the **RAMDirective** because the block does not have a fixed read behavior.

To set **RAMDirective** for a block from the command line, use `hdlset_param`. For example, to set **RAMDirective** to **ultra** for a Single Port RAM block inside a subsystem, `my_dut`:

```
hdlset_param('my_dut/Single Port RAM', 'RAMDirective', 'ultra');
```

See also `hdlset_param`.

ResetType

Use the **ResetType** block parameter to suppress reset logic generation.

ResetType Value	Description
default	Generate reset logic.

ResetType Value	Description
none	<p>Do not generate reset logic.</p> <p>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.</p> <p>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) in “Test Bench Stimulus and Output” on page 18-21.</p>

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** in “Reset Settings” on page 16-10.

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 21-57.

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 24-27.

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 Resetable block :

```

always @(posedge clk or posedge reset)
begin : Unit_Delay_Resetable_process
  if (reset == 1'b1) begin
    Unit_Delay_Resetable_zero_delay <= 1'b1;
    Unit_Delay_Resetable_switch_delay <= 2'b00;
  end
  else begin
    if (enb) begin
      Unit_Delay_Resetable_zero_delay <= 1'b0;
      if (UDR_reset == 1'b1) begin
        Unit_Delay_Resetable_switch_delay <= 2'b00;
      end
      else begin
        Unit_Delay_Resetable_switch_delay <= In1;
      end
    end
  end
end
end

assign Unit_Delay_Resetable_1 =
  (UDR_reset ||
   Unit_Delay_Resetable_zero_delay ? 1'b1 : 1'b0);
assign out0 = (Unit_Delay_Resetable_1 == 1'b1 ? 2'b00 :
  Unit_Delay_Resetable_switch_delay);

```

When `SoftReset` set to 'on', the following code is generated for a Unit Delay Resetable block :

```

always @(posedge clk or posedge reset)
begin : Unit_Delay_Resetable_process
  if (reset == 1'b1) begin
    Unit_Delay_Resetable_reg <= 2'b00;
  end
  else begin
    if (enb) begin
      if (UDR_reset != 1'b0) begin
        Unit_Delay_Resetable_reg <= 2'b00;
      end
      else begin
        Unit_Delay_Resetable_reg <= In1;
      end
    end
  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 24-23.

UseMatrixTypesInHDL

The `UseMatrixTypesInHDL` block property specifies whether to generate 2-D matrices in HDL code when you have MATLAB matrices in your MATLAB Function block.

UseMatrixTypesInHDL Setting	Description
off (default)	Generate HDL vectors with index computation logic for MATLAB matrices. This option can use more area in the synthesized hardware.

UseMatrixTypesInHDL Setting	Description
on	<p>Generate HDL matrices for MATLAB matrices. This option can save area in the synthesized hardware.</p> <p>The following requirements apply:</p> <ul style="list-style-type: none"> You cannot use matrices at the block input or output ports. Matrix elements cannot be complex or struct data types. You cannot use linear indexing to specify matrix elements. For example, if you have a 3x3 matrix, A, you cannot use A(4). Instead, use A(2,1). <p>You can also use a colon operator in either the row or column subscript, but not both. For example, you can use A(3,1:3) and A(2:3,1), but not A(2:3,1:3).</p> <ul style="list-style-type: none"> InstantiateFunctions must be set to 'off'. If you enable InstantiateFunctions, UseMatrixTypesInHDL has no effect. UseMatrixTypesInHDL has no effect if you have System objects in your MATLAB code.

To generate 2-D matrices in HDL code:

- 1 Right-click the MATLAB Function block and select **HDL Code > HDL Block Properties**.
- 2 For **UseMatrixTypesInHDL**, select **on**.

Alternatively, at the command line, use `makehdl` or `hdlset_param` to set the `UseMatrixTypesInHDL` block property to 'on'.

For example, suppose you have a model, `myModel`, with a subsystem, `dutSubsys`, that contains a MATLAB Function block, `myMLFcn`. To generate 2-D matrices in HDL code for `myMLFcn`, enter:

```
hdlset_param('myModel/dutSubsys/myMLFcn', 'UseMatrixTypesInHDL', 'on')
```

UsePipelines

You can use this mode with Product blocks in Divide and Reciprocal modes. This property becomes available when you set the HDL architecture for the blocks to `ShiftAdd`. This architecture uses a non-restoring division algorithm that performs multiple shift and add operations to compute the quotient. The `ShiftAdd` architecture provides improved accuracy compared to the Newton-Raphson approximation method.

When you use the `ShiftAdd` architecture, you can use the `UsePipelines` implementation parameter to specify whether to use a pipelined or non-pipelined implementation of the non-restoring division.

UsePipelines Setting	Mapping Behavior
on (default)	Use a pipelined implementation of the non-restoring shift and add operation for Divide and Reciprocal blocks. This setting adds more delays to your design but achieves a higher maximum clock frequency on the target FPGA device. The number of pipelines inserted matches the number of iterations that the algorithm requires to compute the quotient or reciprocal.
off	Use a non-pipelined implementation of the non-restoring shift and add operation for Divide and Reciprocal blocks. This setting does not add delays to your design. As division and reciprocal are resource-intensive operations, to achieve higher clock frequencies on the target FPGA, set UsePipelines to on.

Set UsePipelines for Divide and Reciprocal Blocks

To set `UsePipelines` for a subsystem from the HDL Block Properties dialog box:

- 1 Right-click the subsystem.
- 2 Select **HDL Code > HDL Block Properties** .
- 3 For **UsePipelines**, select **on** or **off**.

To set `UsePipelines` for a block from the command line, use `hdlset_param`. For example, to turn off `UsePipelines` for a Divide block inside a subsystem, `my_dut`:

```
hdlset_param('my_dut/divide', 'UsePipelines', 'off');
```

See also `hdlset_param`.

UseRAM

The UseRAM implementation parameter enables using RAM-based mapping for a block instead of mapping to a shift register.

UseRAM Setting	Mapping Behavior
off	<p>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 21-37.</p>
on	<p>The delay maps to a dual-port RAM block when the following conditions are true:</p> <ul style="list-style-type: none"> • Initial value of the delay is zero. • The Delay block does not have an external reset or enable port. • Delay length > 4. • Delay has one of the following set of numeric and data type attributes: <ul style="list-style-type: none"> • (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) • RAMSize is greater than or equal to the RAMMappingThreshold value. RAMSize is the product DelayLength * WordLength * ComplexLength. <ul style="list-style-type: none"> • DelayLength is the number of delays that the Delay block specifies. • WordLength is the number of bits that represent the data type of the delay. • ComplexLength is 2 for complex signals; 1 otherwise. <p>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 21-35.</p>

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:

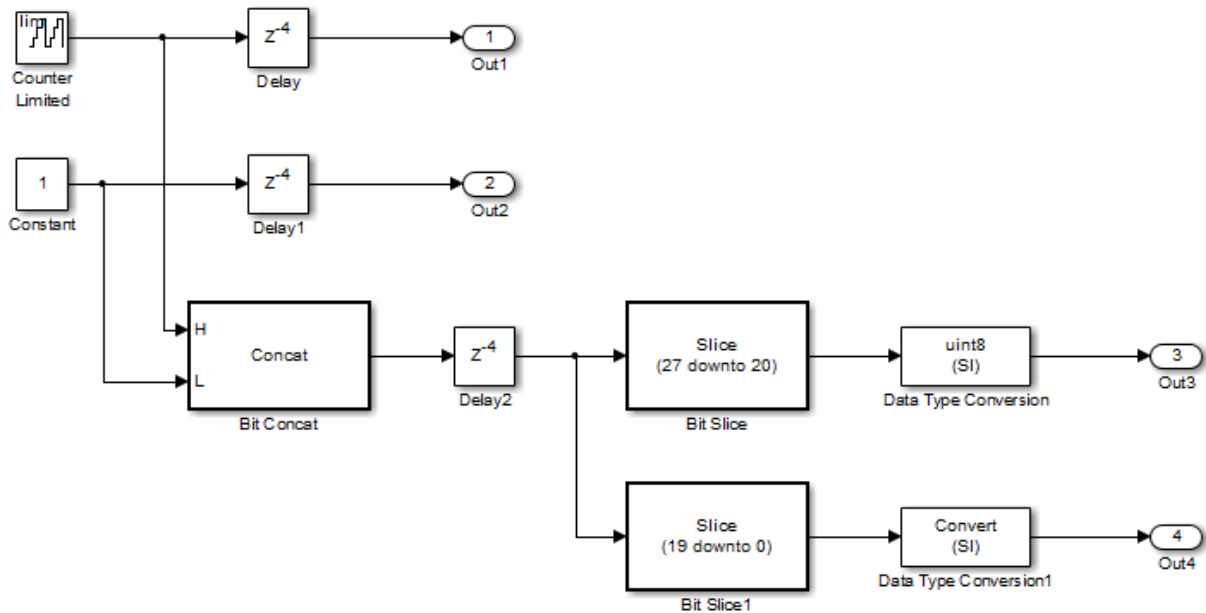
- Increased occupancy on a single RAM
- Sharing of address generation logic, which minimizes duplication of identical HDL code
- Mapping of delays to a RAM when the *individual* delays do not satisfy the threshold

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 $\text{DelayLength} * \text{WordLength} * \text{VectorLength} * \text{ComplexLength}$.
 - DelayLength is the total number of delays.
 - WordLength is the number of bits that represent the data type of the merged delay.
 - VectorLength is the number of elements in a vector delay. VectorLength is 1 for a scalar delay.
 - ComplexLength is 2 for complex delays; 1 otherwise.

Example of Multiple Delays Mapping to a Block RAM

RAMMappingThreshold for the following 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. The 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

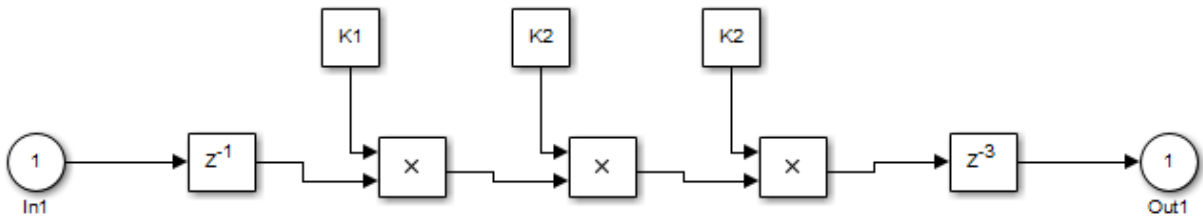
When UseRAM is off for a Delay block, HDL Coder maps the delay to a shift register by default. However, the coder changes the UseRAM setting to on and tries to map the delay to a RAM under the following conditions:

- 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 8-11.

HDL Block Properties: Native Floating Point

In this section...

“Overview” on page 21-39

“CheckResetToZero” on page 21-39

“DivisionAlgorithm” on page 21-40

“HandleDenormals” on page 21-42

“InputRangeReduction” on page 21-43

“LatencyStrategy” on page 21-44

“NFPCustomLatency” on page 21-46

“MantissaMultiplyStrategy” on page 21-47

“MaxIterations” on page 21-49

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 21-71 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.

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 $\text{rem}(a,b)=0$. 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)	<p>The default Radix-2 mode performs repeated subtractions by computing one bit of the quotient in each iteration.</p> <p>To design for lower area usage while trading off for latency, use the Radix-2 mode in combination with the LatencyStrategy set to MAX.</p>
Radix-4	<p>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.</p> <p>To design for lower latency while trading off for area, use the Radix-4 mode in combination with the LatencyStrategy set to MAX.</p>

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 10-70.

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.
'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 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`.

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 in native floating-point mode. The property specifies whether you want the blocks in your design to map to minimum, maximum, or a custom latency for the native floating-point 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.

LatencyStrategy Setting	Description
'Custom'	During code generation, HDL Coder adds latency equal to the value that you specify for the NFPCustomLatency setting 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 21-46.

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 10-83.

Set Latency Strategy For a Block

To set latency strategy 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 **inherit**, **Max**, **Min**, **Zero**, or **Custom**.
- 4 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.

You can set **LatencyStrategy** to **Custom** for these blocks with single and double data types.

- Add
- Subtract
- Product
- Math Function in Reciprocal mode
- Gain
- Divide

- Relational Operator
- Data Type Conversion

You can also set **LatencyStrategy** to **Custom** for these blocks with single data types.

- Sqrt
- Reciprocal Sqrt
- Sum of Elements
- Product of Elements

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', 'MIN')
```

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**. For details, see the "HDL Code Generation" section of each block page.

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', 'LatencyStrategy', '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 <code>mod</code> and <code>rem</code> functions in <code>Native Floating Point</code> 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 <code>mod</code> and <code>rem</code> functions in <code>Native Floating Point</code> 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 <code>mod</code> and <code>rem</code> functions in <code>Native Floating Point</code> 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

- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65

HDL Filter Block Properties

In this section...

“AdderTreePipeline” on page 21-51
“AddPipelineRegisters” on page 21-51
“ChannelSharing” on page 21-52
“CoeffMultipliers” on page 21-52
“DALUTPartition” on page 21-53
“DARadix” on page 21-55
“FoldingFactor” on page 21-55
“MultiplierInputPipeline” on page 21-56
“MultiplierOutputPipeline” on page 21-56
“NumMultipliers” on page 21-56
“ReuseAccum” on page 21-57
“SerialPartition” on page 21-57

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 21-64.

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 needed by your design.

For filter architecture diagrams that indicate where the pipeline stages are added, see “HDL Filter Architectures” on page 21-60.

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 “Generate HDL Code for Multichannel FIR Filter” (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 enable 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 21-68.

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	$FL = \text{length}(\text{find}(\text{Hd.numerator} \sim= 0))$
Direct-form asymmetrical FIR, direct-form symmetrical FIR	$FL = \text{ceil}(\text{length}(\text{find}(\text{Hd.numerator} \sim= 0))/2)$

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
<i>Vector</i> : 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.	$FL = \text{size}(\text{polyphase}(\text{Hm}), 2)$
<i>Matrix</i> : 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 .	$p = \text{polyphase}(\text{Hm});$ $FL_i = \text{length}(\text{find}(p(i, :)));$ <p>where i is the index to the ith row of the polyphase matrix of the multirate filter. The ith row of the matrix p represents the ith subfilter.</p>

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 $\text{mod}(W, \log_2(N)) = 0$, 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 21-69.

See “Distributed Arithmetic for HDL Filters” on page 21-68.

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 21-56. 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 21-60.

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 21-60.

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 21-55 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 21-60

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	<p>[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:</p> <ul style="list-style-type: none"> • 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	<p>[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.</p>
Cascade-serial with automatically optimized partitioning	Omit this property.

For more information on parallel and serial filter architectures, see “HDL Filter Architectures” on page 21-60.

This property is also used for Min/Max blocks with cascade-serial architectures. For how to configure Min/Max cascades, see “SerialPartition” on page 21-28.

See Also

More About

- “Set and View HDL Model and Block Parameters” on page 21-71
- “HDL Block Properties: General” on page 21-3

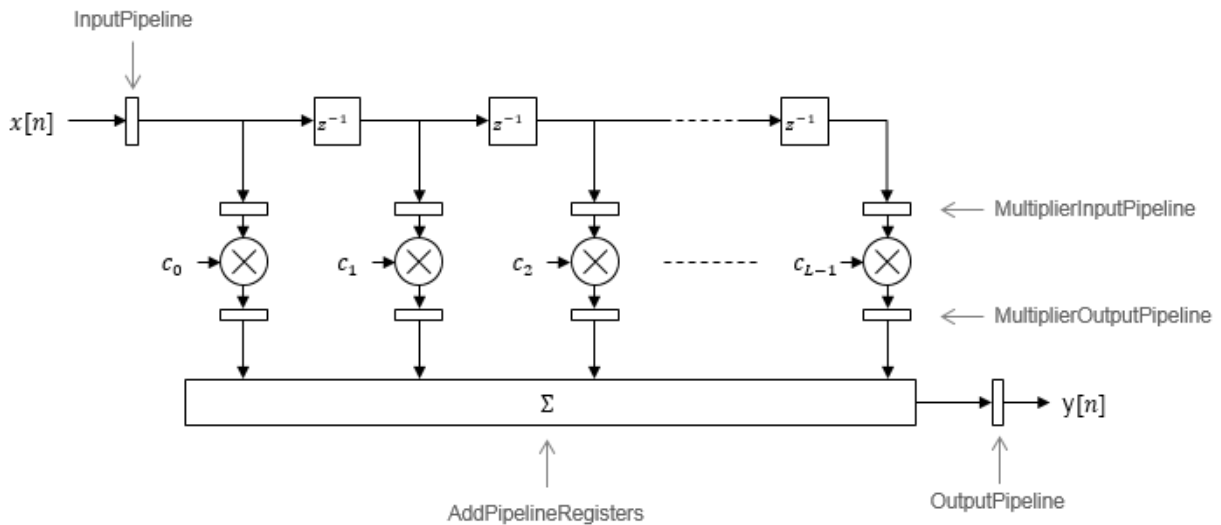
HDL Filter Architectures

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 21-57 and “ReuseAccum” on page 21-57 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 21-51 for scalar input filters, and “AdderTreePipeline” on page 21-51 for frame-based filters. Specify pipeline stages before and after each multiplier with MultiplierInputPipeline on page 21-56 and MultiplierOutputPipeline on page 21-56. Set the number of pipeline stages before and after the filter using “InputPipeline” on page 21-16 and “OutputPipeline” on page 21-22. The architecture diagrams show the locations of the various configurable pipeline stages.

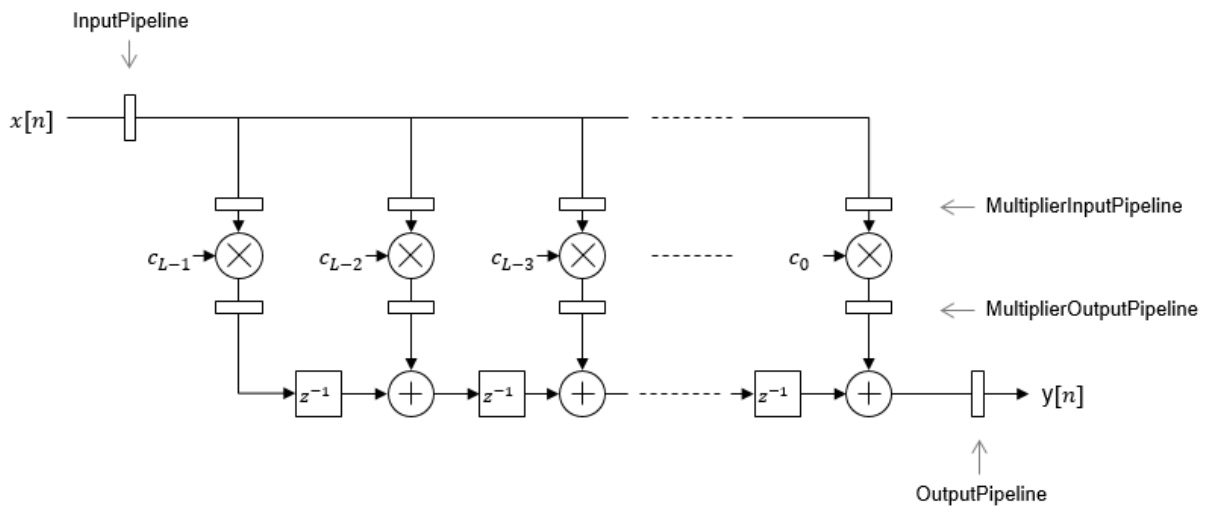
Fully Parallel Architecture

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 21-57 and “`ReuseAccum`” on page 21-57 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

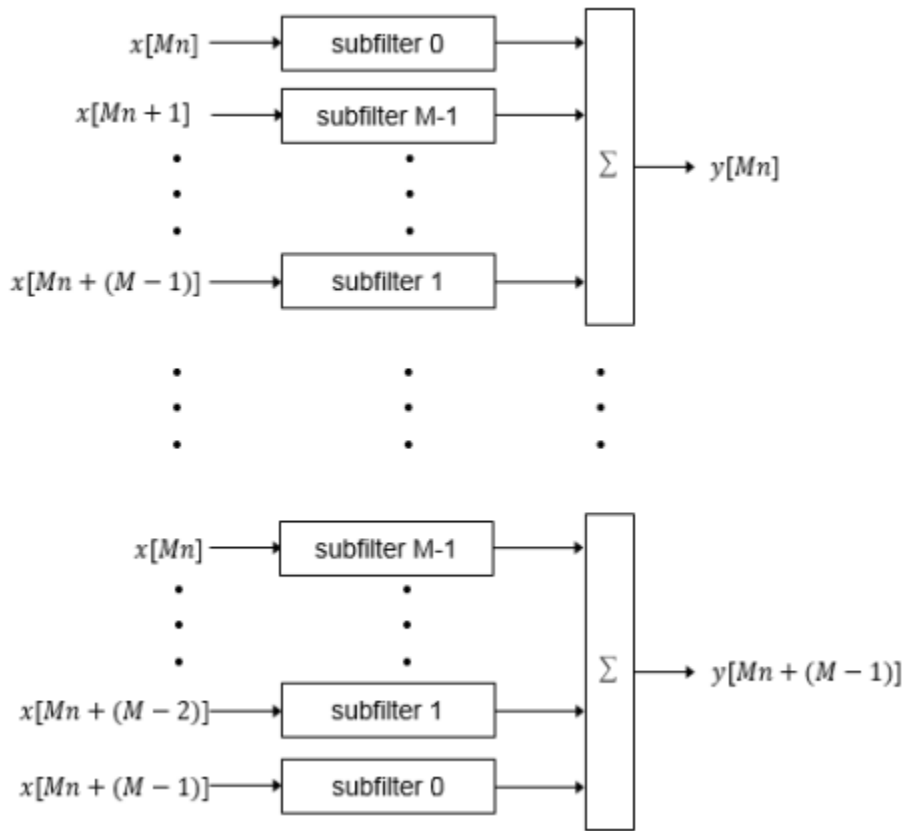
Serialization of a filter increases the total latency of the design by one clock cycle. The serial architectures use an accumulator (an adder with a register) to add the products sequentially. An additional final register is used to store the summed result of all the serial partitions, requiring an extra 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.

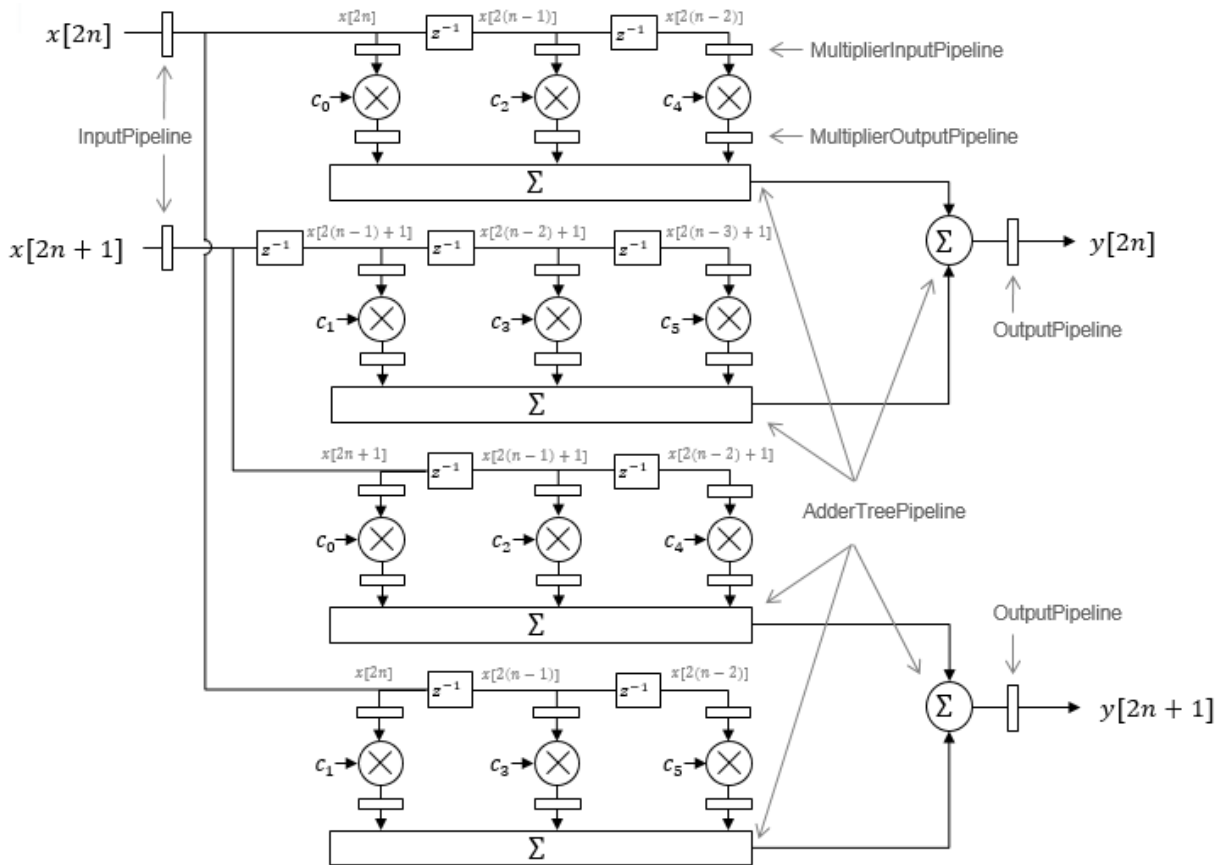


Each of the subfilters includes every M th coefficient. The subfilter results are added so that each output sample is the sum of each of the coefficients multiplied with one input sample.

$$\begin{aligned}\text{subfilter } 0 &= c_0, c_M, \dots \\ \text{subfilter } 1 &= c_1, c_{M+1}, \dots \\ &\dots \\ \text{subfilter } M-1 &= c_{M-1}, c_{2M-1}, \dots\end{aligned}$$

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 21-51 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 21-56 and MultiplierOutputPipeline on page 21-56.



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 21-51

- “Distributed Arithmetic for HDL Filters” on page 21-68

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 21-53
- Improve Performance with Parallelism: “DARadix” on page 21-55

For information on the theoretical foundations of DA, see “Further References” on page 21-69.

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 21-71

“Set HDL Block Parameters for Multiple Blocks Programmatically” on page 21-72

“View All HDL Block Parameters” on page 21-73

“View Non-Default HDL Block Parameters” on page 21-74

“View HDL Model Parameters” on page 21-74

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 as needed. 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 =

    4x1 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(gcf, '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

`hdldispblkparams` | `hdldispmdlparams` | `hdlget_param` | `hdlset_param`

More About

- “HDL Block Properties: General” on page 21-3
- “HDL Block Properties: Native Floating Point” on page 21-39

Pass through, No HDL, and Cascade Implementations

Pass-through and No HDL Implementations

Implementation	Description
Pass-through implementations	<p>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:</p> <ul style="list-style-type: none"> • Convert 1-D to 2-D • Reshape • Signal Conversion • Signal Specification
No HDL	<p>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.</p> <p>You can also use this implementation as an alternative implementation for subsystems.</p>

For more information related to special-purpose implementations, see “External Component Interfaces”.

Cascade Architecture Best Practices

HDL Coder supports cascade implementations for the Sum of Elements, Product of Elements, and MinMax blocks. These implementations require multiple clock cycles to process their inputs; therefore, their inputs must be kept unchanged for their entire sample-time period. Generated test benches accomplish this by using a register to drive the inputs.

A recommended design practice, when integrating generated HDL code with other HDL code, is to provide registers at the inputs. While not strictly required, adding registers to the inputs improves timing and avoids problems with data stability for blocks that require multiple clock cycles to process their inputs.

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:

```
hdlcoderramrom
```

Wireless Communications Design for FPGAs and ASICs

In this section...

“From Mathematical Algorithm to Hardware Implementation” on page 21-78

“HDL-Optimized Blocks” on page 21-81

“Reference Applications” on page 21-81

“Generate HDL Code and Prototype on FPGA” on page 21-82

Deploying algorithmic models to FPGA 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 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 LTE HDL Toolbox, allow you to start with a mathematical model, such as MATLAB code from LTE Toolbox™ or 5G Toolbox™, and design a hardware implementation of that algorithm that is suitable for FPGAs and ASICs.

From Mathematical Algorithm to Hardware Implementation

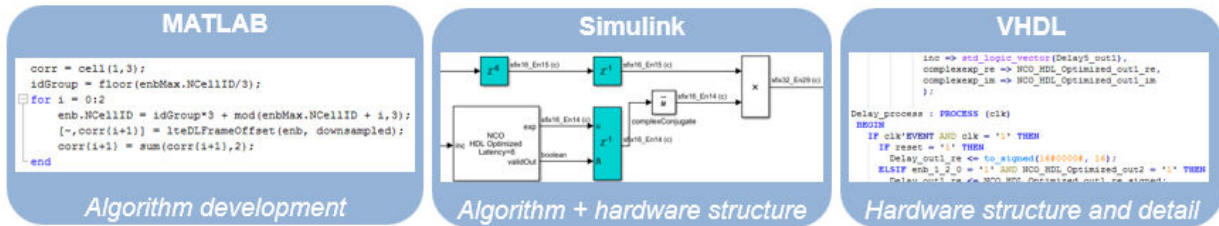
Wireless communications design often starts with algorithm development and testing using MATLAB functions. MATLAB code, which usually operates on matrices of floating-point data, is good for developing mathematical algorithms, manipulating large data sets, and visualizing data.

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. Usually this tradeoff means operating on streaming data, and using some logic to control the storage and flow of data. Hardware engineers usually 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

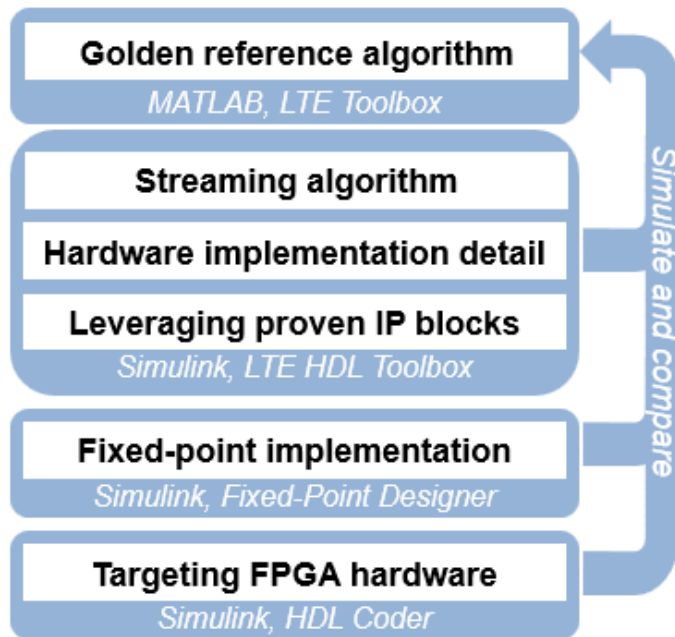
a Simulink model that you can use to automatically generate HDL code by using HDL Coder.

This diagram shows the design progression from mathematical algorithm in MATLAB, to hardware-compatible implementation in Simulink, and then the generated VHDL code.

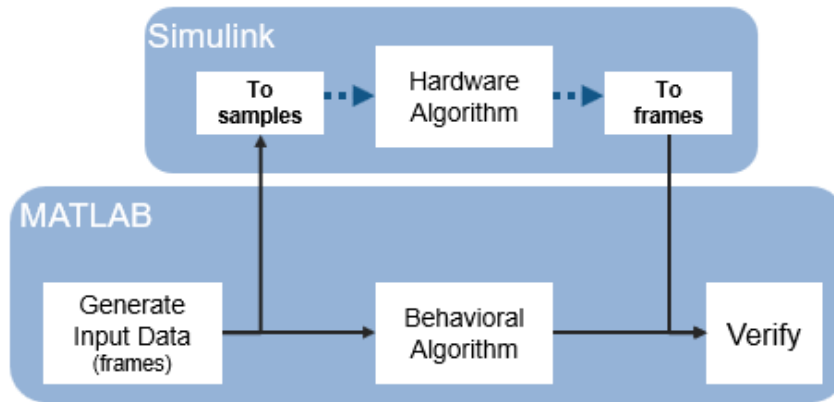


While both MATLAB and Simulink support automatic generation of HDL code, you must construct your design with hardware requirements in mind, and Simulink is better-suited for cycle-based modeling for hardware. It can represent parallel data paths and streaming data with control signals to manage the timing 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 or 5G wireless standards, you can use LTE Toolbox and 5G Toolbox functions to create a golden reference in MATLAB. Then transition to Simulink and create a hardware-compatible implementation by using library blocks from LTE HDL Toolbox and blocks from Communications Toolbox and DSP System 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 LTE HDL Toolbox implement encoders, decoders, modulators, demodulators, and sequence generators for use in an LTE, 5G, or general 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 Communications Toolbox and DSP System 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 [LTE HDL Toolbox Block List \(HDL Code Generation\)](#), [Communications Toolbox Block List \(HDL Code Generation\)](#), and [DSP System Toolbox Block List \(HDL Code Generation\)](#).

Reference Applications

LTE HDL Toolbox provides reference applications that contain hardware-ready implementations of large parts of the LTE physical layer. These designs are verified

against the "golden reference" functions provided by LTE Toolbox. They have also been tested on FPGA boards to confirm that they encode and decode over-the-air waveforms and use a reasonable amount of hardware resources. They are designed to be modular, scalable, and extensible so you can insert additional physical channels. The receiver design was tested using LTE waveforms captured off-the-air on three different continents.

The suite of LTE reference applications includes:

- Primary and secondary synchronization signal (PSS/SSS) generation and detection
- Downlink shared control channel detector and master information block (MIB) generation and recovery
- First system information block (SIB1) decoder
- Hardware-software interface models for MIB and SIB1 bit parsing and channel estimation data indexing
- Waveform generation for multiple-antenna transmission
- Support for FDD and TDD for both transmitter and receiver applications

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

LTE 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 a hardware support package such as Communications Toolbox Support Package for Xilinx Zynq®-Based Radio.

See Also

External Websites

- [LTE HDL Toolbox](#)
- [HDL Coder](#)

Generating HDL Code for Multirate Models

- “Code Generation from Multirate Models” on page 22-2
- “Timing Controller for Multirate Models” on page 22-5
- “Generate Reset for Timing Controller” on page 22-6
- “Multirate Model Requirements for HDL Code Generation” on page 22-7
- “Generate a Global Oversampling Clock” on page 22-9
- “Using Triggered Subsystems for HDL Code Generation” on page 22-15
- “Generate Multicycle Path Information Files” on page 22-19
- “Using Multiple Clocks in HDL Coder™” on page 22-28

Code Generation from Multirate Models

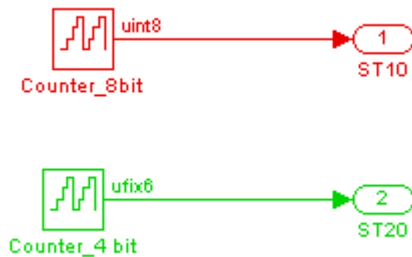
HDL Coder supports HDL code generation for single-clock and multiple clock multirate models. Your model can include blocks running at multiple sample rates:

- 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.

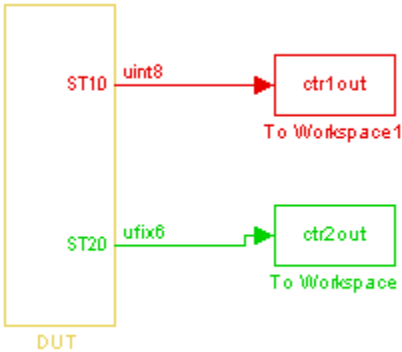
In general, generating HDL code for a multirate model does not differ greatly from generating HDL code for a single-rate model. However, there are a few requirements and restrictions on the configuration of the model and the use of specialized blocks (such as Rate Transitions) that apply to multirate models. For details, see “Multirate Model Requirements for HDL Code Generation” on page 22-7.

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.

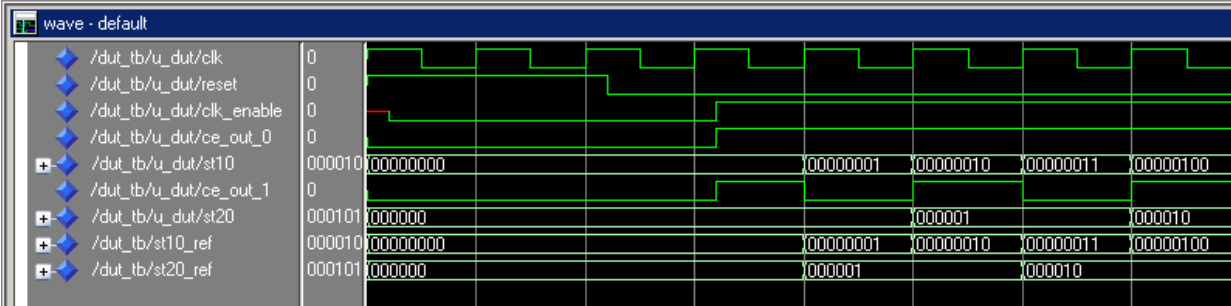


The following listing shows the VHDL entity declaration generated for the DUT.

```
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 timing controller entity generates the required rates from a single master clock, using one or more counters to create multiple clock enables. The master clock rate is the fastest rate in the model in single clock mode. In multiple clock mode, it can be any clock in the DUT. The outputs of the timing controller are clock enable signals running at rates an integer multiple slower than the timing controller's master clock

When using single clock mode, HDL code generated from multirate models employs a single master clock that corresponds to the base rate of the DUT. When using multiple clock mode, HDL code generated from multirate models employs one clock input for each rate in the DUT. The number of timing controllers generated in multiple clock mode depends on the design in the DUT.

Each timing controller entity definition is written to a separate code file. The timing controller file and entity names derive from the name of the subsystem that is selected for code generation (the DUT). To form the timing controller name, HDL Coder appends the value of the `TimingControllerPostfix` property to the DUT name.

To learn more, see “Using Multiple Clocks in HDL Coder™” on page 22-28.

Generate Reset for Timing Controller

In this section...

“Requirements for Timing Controller Reset Port Generation” on page 22-6

“How To Generate Reset for Timing Controller” on page 22-6

“Limitations for Timing Controller Reset Port Generation” on page 22-6

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 Turnkey
- FPGA-in-the-Loop
- Custom IP core generation

Multirate Model Requirements for HDL Code Generation

In this section...

“Model Configuration Parameters” on page 22-7

“Sample Rate” on page 22-7

“Blocks To Use For Rate Transitions” on page 22-8

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.
 - **Tasking mode:** Must be explicitly set to `SingleTasking`. Do not set **Tasking mode** to `Auto`.
- `hdlsetup` configures the following **Diagnostics / Sample time** options for all models:
 - **Multitask rate transition:** error
 - **Single task rate transition:** 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 rate transition** and **Single task rate transition** to error to detect illegal rate transitions before code is generated.

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...

“Why Use a Global Oversampling Clock?” on page 22-9

“Requirements for the Oversampling Factor” on page 22-9

“Specifying the Oversampling Factor From the GUI” on page 22-10

“Specifying the Oversampling Factor From the Command Line” on page 22-11

“Resolving Oversampling Rate Conflicts” on page 22-11

Why Use a Global Oversampling Clock?

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 such 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 factor. The oversampling factor expresses the desired rate of the global oversampling clock as a multiple of the base rate of your model.

When you specify an oversampling factor, HDL Coder generates the global oversampling clock and derives the required timing signals from clock signal. Generation of the global oversampling clock affects only generated HDL code. The clock does not affect the simulation behavior of your model.

Requirements for the Oversampling Factor

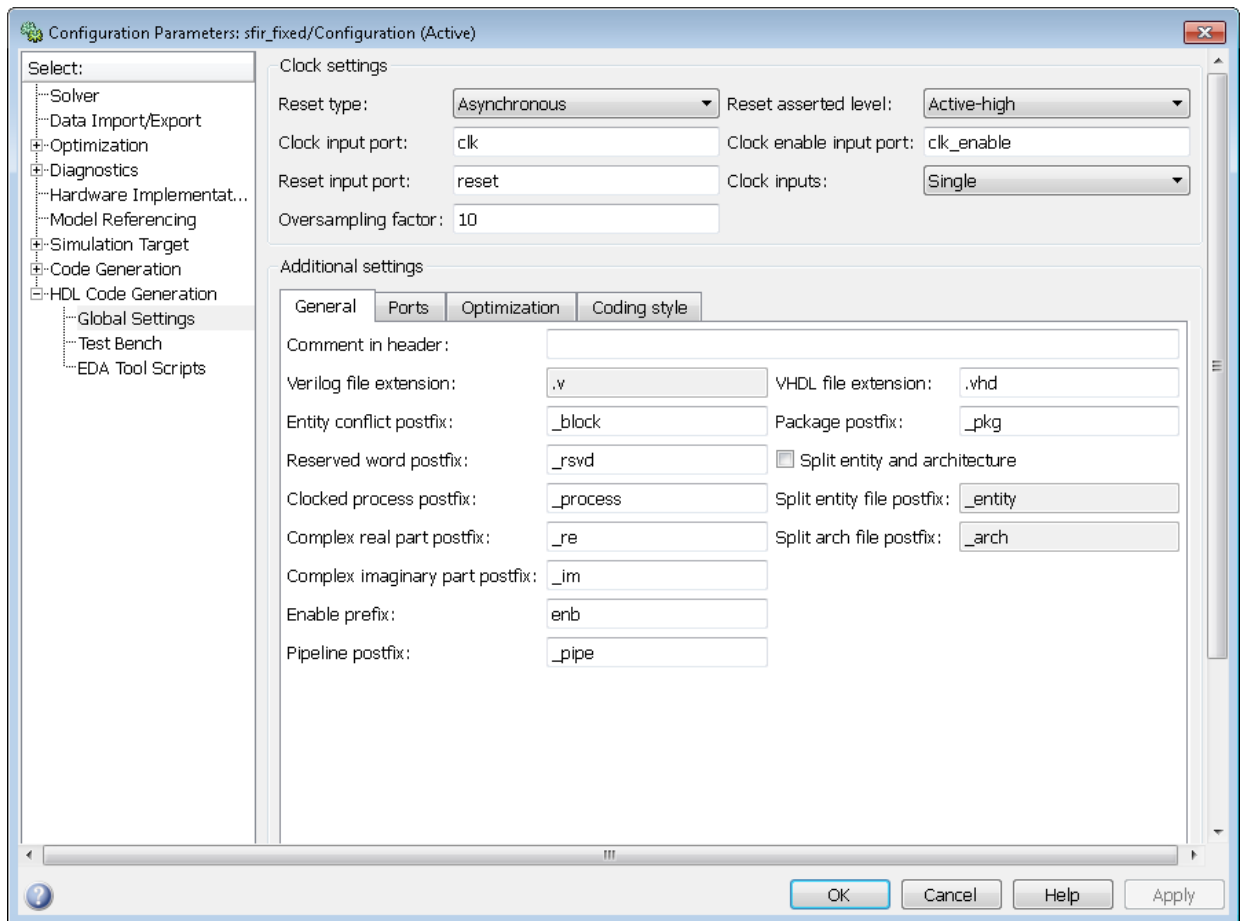
When you specify the oversampling factor for a global oversampling clock, note these requirements:

- The oversampling factor must be an integer greater than or equal to 1.
- The default value 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 22-11 .

Specifying the Oversampling Factor From the GUI

You can specify the oversampling factor for a global clock from the GUI as follows:

- 1 Select the **HDL Code Generation > Global Settings** pane in the Configuration Parameters dialog box.
- 2 For **Oversampling factor** in the **Clock settings** section, enter the desired oversampling factor. In the following figure, **Oversampling factor** specifies a global oversampling clock that runs at ten times the base rate of the model.



- 3 Click **Generate** on the **HDL Code Generation** pane to initiate code generation.

HDL Coder reports the oversampling clock rate:

```
### Begin VHDL Code Generation
### MESSAGE: The design requires 10 times faster clock with respect to the base rate = 1.
### Working on symmetric_fir_tc ashdlsrc\symmetric_fir_tc.vhd
### Working on sfir_fixed/symmetric_fir ashdlsrc\symmetric_fir.vhd
### HDL Code Generation Complete.
```

Specifying the Oversampling Factor From the Command Line

You can specify the oversampling factor for a global clock from the command line by setting the `Oversampling` property with `hdlset_param` or `makehdl`. The following example specifies an oversampling factor of 7:

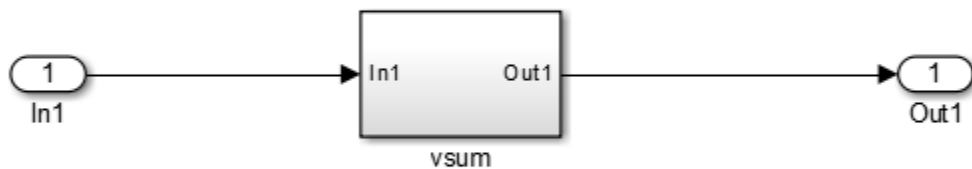
```
>> makehdl(gcb,'Oversampling', 7)
### Generating HDL for 'sfir_fixed/symmetric_fir'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### Begin VHDL Code Generation
### MESSAGE: The design requires 7 times faster clock with respect to the base rate = 1.
### Working on symmetric_fir_tc ashdlsrc\symmetric_fir_tc.vhd
### Working on sfir_fixed/symmetric_fir ashdlsrc\symmetric_fir.vhd
### HDL Code Generation Complete.
```

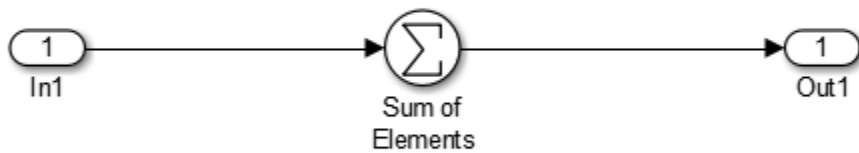
Resolving Oversampling Rate Conflicts

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 (SumCascadeHDL Emission) 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/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.
...
```

Using Triggered Subsystems for HDL Code Generation

In this section...

“Requirements” on page 22-15
“Best Practices” on page 22-15
“Using the Signal Builder Block” on page 22-16
“Using Trigger As Clock” on page 22-16
“Specify Trigger As Clock” on page 22-16
“Limitations” on page 22-17

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.

Requirements

Each triggered subsystem input or output data signal must have delays immediately outside and immediately inside the subsystem. These delays act as a synchronization interface between the regions running at different rates.

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 Builder Block

When you connect outputs from a Signal Builder 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 Builder 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 Builder 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 > Optimization** 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')
```

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.
- The data type of the trigger signal is either `boolean` or `ufix1`.
- Outputs of the triggered subsystem have an initial value of 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`.
- 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.

Generate Multicycle Path Information Files

In this section...

“Overview” on page 22-19

“Format and Content of a Multicycle Path Information File” on page 22-20

“File Naming and Location Conventions” on page 22-25

“Generating Multicycle Path Information Files Using the GUI” on page 22-25

“Generating Multicycle Path Information Files Using the Command Line” on page 22-25

“Limitations” on page 22-26

Overview

HDL Coder implements multirate systems in HDL by generating a master clock running at the model's base rate, and generating subrate timing signals from the master clock (see also “Code Generation from Multirate Models” on page 22-2). The propagation time between two subrate registers can be more than one cycle of the master clock. A multicycle path is a path between two such registers.

When synthesizing HDL code, it is often useful to provide an analysis of multicycle register-to-register paths to the synthesis tool. If the synthesis tool can identify multicycle paths, you may be able to:

- Realize higher clock rates from your multirate design.
- Reduce the area of your design.
- Reduce the execution time of the synthesis tool.

Using the **Generate multicycle path information** option (or the equivalent `MulticyclePathInfo` property for `makehdl`) you can instruct the coder to analyze multicycle paths in the generated code, and generate a multicycle path information file.

A multicycle path information file is a text file that describes one or more multicycle path constraints. A multicycle path constraint is a timing exception - it relaxes the default constraints on the system timing by allowing signals on a given path to have a longer propagation time. When using multiple clock mode, the file also contains clock definitions.

Typically a synthesis tool gives every signal a time budget of exactly 1 clock cycle to propagate from a source register to a destination register. A timing exception defines a

path multiplier , N, that informs the synthesis tool that a signal has N clock cycles ($N > 1$) to propagate from the source to destination register. The path multiplier expresses some number of cycles of a *relative clock* at either the source or destination register. Where a timing exception is defined for a path, the synthesis tool has more flexibility in meeting the timing requirements for that path and for the system as a whole.

The generated multicycle path information file does not follow the native constraint file format of a particular synthesis tool. The file contains the multicycle path information required by popular synthesis tools. You can manually convert this information to multicycle path constraints in the format required by your synthesis tool, or write a script or tool to perform the conversion. The next section describes the format of a multicycle path constraint file in detail.

Format and Content of a Multicycle Path Information File

The following listing shows a simple multicycle path information file.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Constraints Report
%   Module: Sbs
%   Model: mSbs.mdl
%
%   File Name:hdlsrc/Sbs_constraints.txt
%   Created: 2009-04-10 09:50:10
%   Generated by MATLAB 7.9 and HDL Coder 1.6
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Multicycle Paths
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FROM : Sbs.boolireg; TO : Sbs.booloreg; PATH_MULT : 2; RELATIVE_CLK : source,
      Sbs.clk;
FROM : Sbs.boolireg_v<0>; TO : Sbs.booloreg_v<0>; PATH_MULT : 2;
      RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.doubireg; TO : Sbs.douboreg; PATH_MULT : 2; RELATIVE_CLK : source,
      Sbs.clk;
FROM : Sbs.doubireg_v<0>; TO : Sbs.douboreg_v<0>; PATH_MULT : 2;
      RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.intireg(7:0); TO : Sbs.intoreg(7:0); PATH_MULT : 2;
      RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.intireg_v<0>(7:0);TO : Sbs.intoreg_v<0>(7:0);PATH_MULT : 2
      RELATIVE_CLK : source,Sbs.clk;

```

The first section of the file is a header that identifies the source model and gives other information about how HDL Coder generated the file. This section terminates with the following comment lines:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Multicycle Paths
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Note For a single-rate model or a model without multicycle paths, the coder generates only the header section of the file.

The main body of the file follows. This section contains a flat table, each row of which defines a multicycle path constraint.

Each constraint consists of four fields. The format of each field is one of the following:

- KEYWORD : field;
- KEYWORD : subfield1,... subfield_N;

The keyword identifies the type of information contained in the field. The keyword string in each field terminates with a space followed by a colon.

The delimiter between fields is the semicolon. Within a field, the delimiter between subfields is the comma.

The following table defines the fields of a multicycle path constraint, in left-to-right order.

Keyword : field (or subfields)	Field Description
FROM : <i>src_reg_path</i> ;	The source (or FROM) register of a multicycle path in the system. The value of <i>src_reg_path</i> is the HDL path of the source register's output signal. See also "Register Path Syntax for FROM : and TO : Fields" on page 22-22 .
TO : <i>dst_reg_path</i> ;	The destination (or TO) register of a multicycle path in the system. The FROM register drives the TO register in the HDL code. The value of <i>dst_reg_path</i> is the HDL path of the destination register's output signal. See also "Register Path Syntax for FROM : and TO : Fields" on page 22-22.

Keyword : field (or subfields)	Field Description
PATH_MULT : <i>N</i> ;	<p>The path multiplier defines the number of clock cycles that a signal has to propagate from the source to destination register. The RELATIVE_CLK field describes the clock associated with the path multiplier (the relative clock for the path).</p> <p>The path multiplier value <i>N</i> indicates that the signal has <i>N</i> clock cycles of its relative clock to propagate from source to destination register.</p> <p>The coder does not report register-to-register paths where <i>N</i> = 1, because this is the default path multiplier.</p>
RELATIVE_CLK : <i>relclock</i> , <i>sysclock</i> ;	<p>The RELATIVE_CLK field contains two comma-delimited subfields. Each subfield expresses the location of the relative clock in a different form, for the use of different synthesis tools. The subfields are:</p> <ul style="list-style-type: none"> • <i>relclock</i>: Since HDL Coder currently generates only single-clock systems, this subfield takes the value <i>source</i>. In a multi-clock system, the relative clock associated with a multicycle path could be either the source or destination register of the path, and this subfield could take on either of the values <i>source</i> or <i>destination</i>. This usage is reserved for future release of the coder. • <i>sysclock</i>: This subfield is intended for use with synthesis tools that require the actual propagation time for a multicycle path. <i>sysclock</i> provides the path to the system's top-level clock (e.g., <i>Sbs.clk</i>) You can use the period of this clock and the path multiplier to calculate the propagation time for a given path.

Register Path Syntax for FROM : and TO : Fields

The FROM : and TO: fields of a multipath constraint provide the path to a source or destination register and information about the signal data type, size, and other characteristics.

Fixed Point Signals

For fixed point signals, the register path has the form

`reg_path<ps> (hb:lb)`

where:

- `reg_path` is the HDL hierarchical path of the signal. The delimiter between hierarchical levels is the period, for example: `Sbs.u_H1.initreg`.
- `<ps>`: Part select (zero-origin integer index) for vector signals. Angle brackets `<>` delimit the part select field
- `(hb:lb)`: Bit select field, indicated from high-order bit to low-order bit. The signal width `(hb:lb)` is the same as the defined width of the signal in the HDL code. This representation does not necessarily imply that the bits of the FROM : register are connected to the corresponding bits of the TO : register. The actual bit-to-bit connections are determined during synthesis.

Boolean and Double Signals

For boolean and double signals, the register path has the form

`reg_path<ps>`

where:

- `reg_path` is the HDL hierarchical path of the signal. The delimiter between hierarchical levels is the period (`.`), for example: `Sbs.u_H1.initreg`.
- `<ps>`: Part select (zero-origin integer index) for vector signals. Angle brackets `<>` delimit the part select field

For boolean and double signals, no bit select field is present.

Note The format does not distinguish between boolean and double signals.

Examples

The following table gives several examples of register-to-register paths as represented in a multicycle path information file.

Path	Description
FROM : <code>Sbs.intireg(7:0)</code> ; TO : <code>Sbs.intoreg(7:0)</code> ;	Both signals are fixed point and eight bits wide.

Path	Description
FROM : Sbs.intireg; TO : Sbs.intoreg;	Both signals are either boolean or double.
FROM : Sbs.intireg<0>(7:0); TO : Sbs.intoreg<1>(7:0);	The FROM signal is the first element of a vector. The TO signal is the second element of a vector. Both signals are fixed point and eight bits wide.
FROM : Sbs.u_H1.intireg(7:0); TO : Sbs.intoreg(7:0);	The signal <code>intireg</code> is defined in the module <code>H1</code> , and <code>H1</code> is inside the module <code>Sbs</code> . <code>u_H1</code> is the instance name of <code>H1</code> in <code>Sbs</code> . Both signals are fixed point and eight bits wide.

Ordering of Multicycle Path Constraints

For a given model or subsystem, the ordering of multicycle path constraints within a multicycle path information file may vary depending on whether the target language is VHDL or Verilog, and on other factors. The ordering of constraints may also change in future versions of the coder. When you design scripts or other tools that process multicycle path information file, do not build in any assumptions about the ordering of multicycle path constraints within a file.

Clock Definitions

When you use multiple clock mode, the multicycle path information file also contains a "Clock Definitions" section, as shown in the following listing. This section is located after the header and before the "Multicycle Paths" section.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clock Definitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
CLOCK: Sbs.clk PERIOD: 0.05
CLOCK: Sbs.clk_1_2 BASE_CLOCK: Sbs.clk MULTIPLIER: 2 PERIOD: 0.1

```

The following table defines the fields for the clock definitions.

Keyword : field (or subfields)	Field Description
CLOCK: clock_name	Each clock in the design has a CLOCK definition line.
PERIOD: float_value	The Simulink rate (floating point value) associated with this CLOCK.

Keyword : field (or subfields)	Field Description
BASE_CLOCK: base_clock_name	Names the master clock. This field does not appear on the master clock.
MULTIPLIER: int_value	Gives the ratio of the period of this clock to the master clock. This field does not appear on the master clock.

File Naming and Location Conventions

The file name for the multicycle path information file derives from the name of the DUT and the postfix string '_constraints', as follows:

DUTname_constraints.txt

For example, if the DUT name is `symmetric_fir`, the name of the multicycle path information file is `symmetric_fir_constraints.txt`.

HDL Coder writes the multicycle path information file to the target .

Generating Multicycle Path Information Files Using the GUI

To enable generation of multicycle path information files, select **Register-to-register path info** in the **Multicycle Path Constraints** section of the **HDL Code Generation > Target and Optimizations** pane of the Configuration Parameters dialog box.

When you select this check box and generate code for your model, the code generator creates a multicycle path information file.

Generating Multicycle Path Information Files Using the Command Line

To generate a multicycle path information file from the command line, pass in the property/value pair 'MulticyclePathInfo', 'on' to `makehdl`, as in the following example.

```
>> dut = 'hdlfirtdecim_multicycle/Subsystem';
>> makehdl(dut, 'MulticyclePathInfo','on');
### Generating HDL for 'hdlfirtdecim_multicycle/Subsystem'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 1 message.
```

```

### MESSAGE: For the block 'hdlfirtdecim_multicycle/Subsystem/downsamp0'
The initial condition may not be used when the sample offset is 0.

### Begin VHDL Code Generation
### Working on Subsystem_tc as hdlsrc/Subsystem_tc.vhd
### Working on hdlfirtdecim_multicycle/Subsystem as hdlsrc/Subsystem.vhd
### Generating package file hdlsrc/Subsystem_pkg.vhd
### Finishing multicycle path connectivity analysis.
### Writing multicycle path information in hdlsrc/Subsystem_constraints.txt
### HDL Code Generation Complete.

```

Limitations

Unsupported Blocks and Implementations

The following table lists block implementations (and associated Simulink blocks) that will not contribute to multicycle path constraints information.

Implementation	Block(s)
SumCascadeHDL Emission	Add, Subtract, Sum, Sum of Elements
ProductCascadeHDL Emission	Product, Product of Elements
MinMaxCascadeHDL Emission	MinMax, Maximum, Minimum
ModelReferenceHDL Instantiation	Model
SubsystemBlackBoxHDL Instantiation	Subsystem
RamBlockDualHDL Instantiation	Dual Port RAM
RamBlockSimpDualHDL Instantiation	Simple Dual Port RAM
RamBlockSingleHDL Instantiation	Single Port RAM

Limitations on MATLAB Function Blocks and Stateflow Charts

Loop-Carried Dependencies

HDL Coder does not generate constraints for MATLAB Function blocks or Stateflow charts that contain a `for` loop with a loop-carried dependency.

Indexing Vector or Matrix Variables

In order to generate constraints for a vector or matrix index expression, the index expression must be one of the following:

- A constant

- A for loop induction variable

For example, in the following example of code for a MATLAB Function block, the index expression `reg(i)` does not generate constraints.

```
function y = fcn(u)
%#codegen

N=length(u);
persistent reg;
if isempty(reg)
    reg = zeros(1,N);
end

y = reg;

for i = 1:N-1
    reg(i) = u(i) + reg(i+1);
end
reg(N) = u(N);
```

File Generation Time

Tip Generation of constraint files for large models can be slow.

Using Multiple Clocks in HDL Coder™

This example shows how to instantiate multiple top-level synchronous clock input ports in HDL Coder.

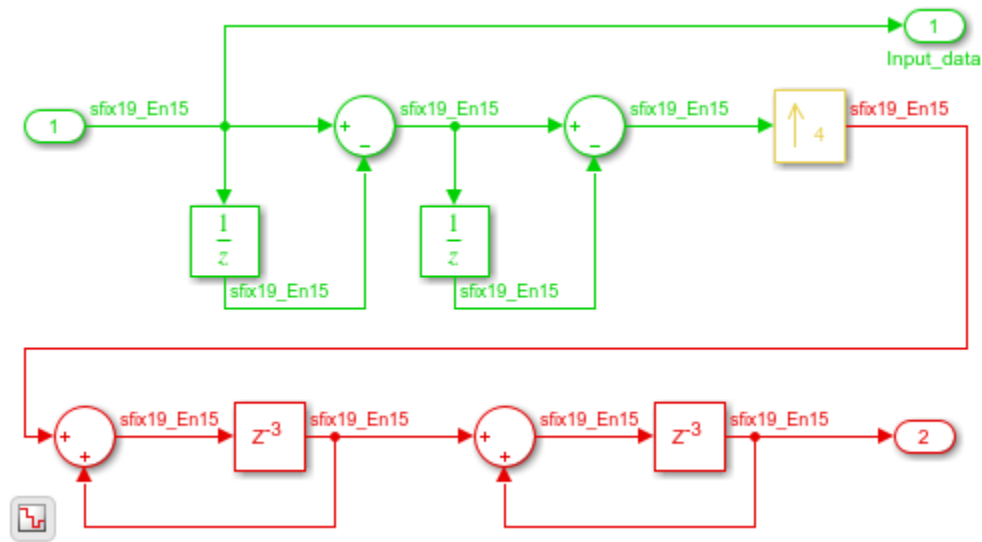
Overview of Clocking Modes

HDL Coder has two clocking modes: one that generates a single clock input to the Device Under Test (DUT), and one that will generate 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 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 corresponding to a separate rate in the model. Transitions between rates often 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.

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 will have a set of three signals that do not appear in the Simulink diagram added to it. Collectively these are a clock bundle, containing signals for clock, master 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');
```

```
### Generating HDL for 'hdlcoder_clockdemo/DUT'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_clockdemo')">hdlcoder_clockdemo</a>.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_clockdemo'.
### Working on DUT_tc as hdlsrc\hdlcoder_clockdemo\DUT_tc.vhd.
### Working on hdlcoder_clockdemo/DUT as hdlsrc\hdlcoder_clockdemo\DUT.vhd.
### Generating package file hdlsrc\hdlcoder_clockdemo\DUT_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc19b_1192687_11188\ib6011188\hdlcoder_clockdemo\report.html')">C:\TEMP\Bdoc19b_1192687_11188\ib6011188\hdlcoder_clockdemo\report.html</a>.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc19b_1192687_11188\ib6011188\hdlcoder_clockdemo\report.html.
### HDL check for 'hdlcoder_clockdemo' complete with 0 errors, 0 warnings, and 1 message.
### 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 as shown here, 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' parameter must be set to 'multiple'. This may be done on the makehdl command line or by changing the "Clock inputs" setting to "Multiple" on the HDL Configuration Parameters Global Settings tab.

```
makehdl('hdlcoder_clockdemo/DUT', 'ClockInputs', 'multiple');

### Generating HDL for 'hdlcoder_clockdemo/DUT'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_clockdemo/DUT')">hdlcoder_clockdemo/DUT</a>.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_clockdemo'.
### Working on DUT_tc_d1 as hdlsrc\hdlcoder_clockdemo\DUT_tc_d1.vhd.
### Working on hdlcoder_clockdemo/DUT as hdlsrc\hdlcoder_clockdemo\DUT.vhd.
### Generating package file hdlsrc\hdlcoder_clockdemo\DUT_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc19b_1192687_11188\ib6011188\hdlcoder_clockdemo\hdlcoder_clockdemo_report.html')">C:\TEMP\Bdoc19b_1192687_11188\ib6011188\hdlcoder_clockdemo\hdlcoder_clockdemo_report.html</a>.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc19b_1192687_11188\ib6011188\hdlcoder_clockdemo\hdlcoder_clockdemo_report.html.
### HDL check for 'hdlcoder_clockdemo' complete with 0 errors, 0 warnings, and 1 message.
### 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

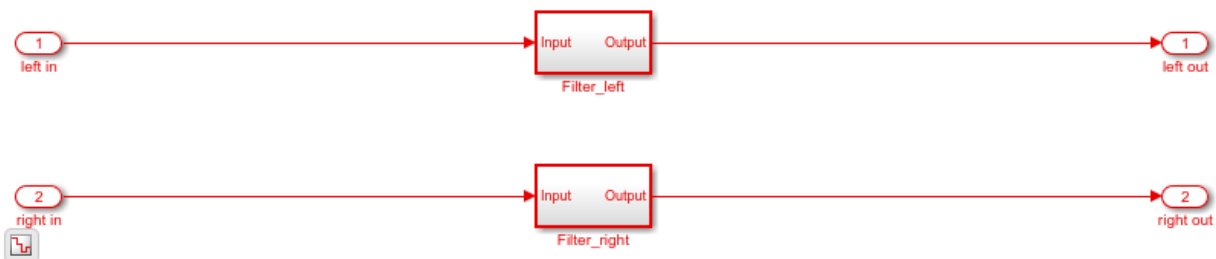
filter modules in hardware. With this configuration, multiple clock mode still only generates one clock, just as single clock mode does.

```

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');

### Generating HDL for 'hdlcoder_audiofiltering/Audio filter'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcod
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### Working on hdlcoder_audiofiltering/Audio filter/Filter_left as hdlsrc\hdlcoder_aud
### Working on hdlcoder_audiofiltering/Audio filter as hdlsrc\hdlcoder_audiofiltering\
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\B
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc19b_1192687_11188\ib6
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 0 r
### 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. The Clock Summary Report shows the timing information for the two clocks.

```

bdclose gm_hdlcoder_audiofiltering;
hdlset_param('hdlcoder_audiofiltering/Audio filter', 'SharingFactor', 2);
makehdl('hdlcoder_audiofiltering/Audio filter', 'Traceability', 'on');

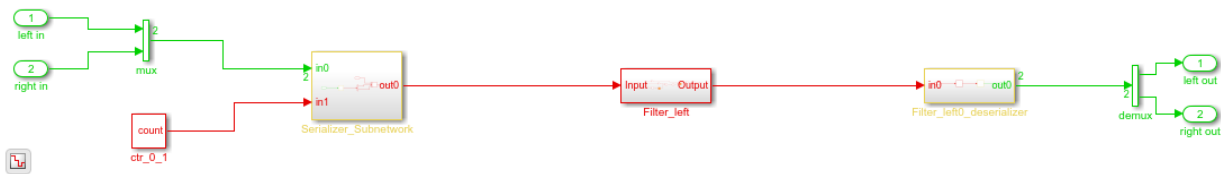
```

```

open_system('gm_hdlcoder_audiofiltering/Audio filter');
set_param('gm_hdlcoder_audiofiltering', 'SimulationCommand', 'update');

### Generating HDL for 'hdlcoder_audiofiltering/Audio filter'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcod
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences th
### Output port 0: 1 cycles.
### Output port 1: 1 cycles.
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### MESSAGE: The design requires 2 times faster clock with respect to the base rate = 0
### Working on hdlcoder_audiofiltering/Audio filter/Filter_left as hdlsrc\hdlcoder_aud
### Working on hdlcoder_audiofiltering/Audio filter as hdlsrc\hdlcoder_audiofiltering\
### Generating package file hdlsrc\hdlcoder_audiofiltering\Audio_filter_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\B
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc19b_1192687_11188\ib6
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 1 r
### HDL code generation complete.

```

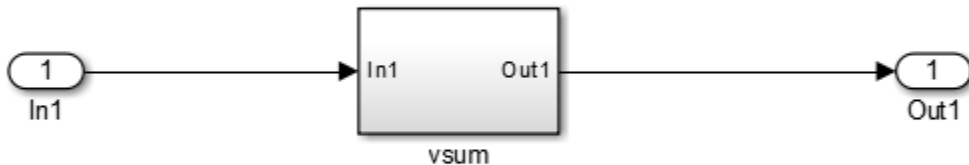


Generating Bit-True Cycle-Accurate Models

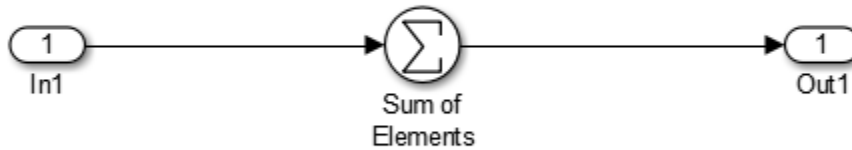
Locate Numeric Differences After Speed Optimization

This example first selects a speed-optimized Sum block implementation for simple model that computes a vector sum. It then examines a generated model and locates the numeric changes introduced by the optimization.

The model, `simplevectorsum`, 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.

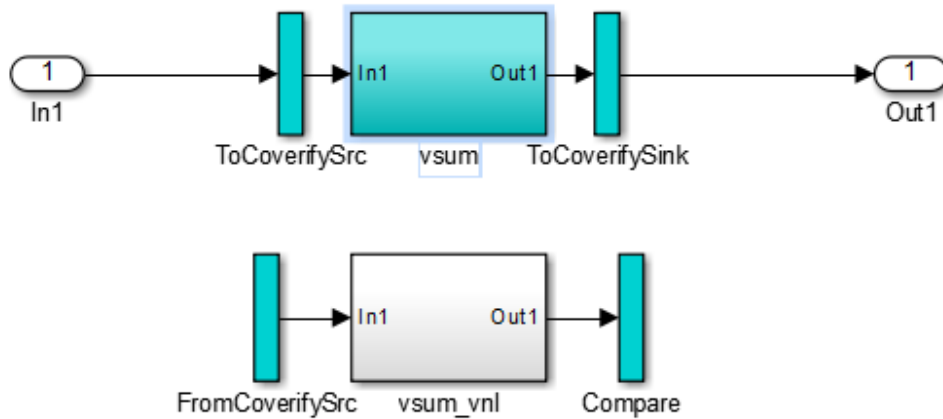


The model is configured to use the Tree implementation when generating HDL code for the Sum block within the `vsum` subsystem. This implementation, optimized for minimal latency, generates a tree-shaped structure of adders for the Sum block.

To select a nondefault implementation for an individual block:

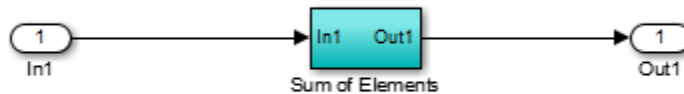
- 1 Right-click the block and select **HDL Code > HDL Block Properties** .
- 2 In the HDL Properties dialog box, select the desired implementation from the **Architecture** menu.
- 3 Click **Apply** and close the dialog box.

After code generation, you can view the validation model, `gm_simplevectorsum_tree_vnl`.

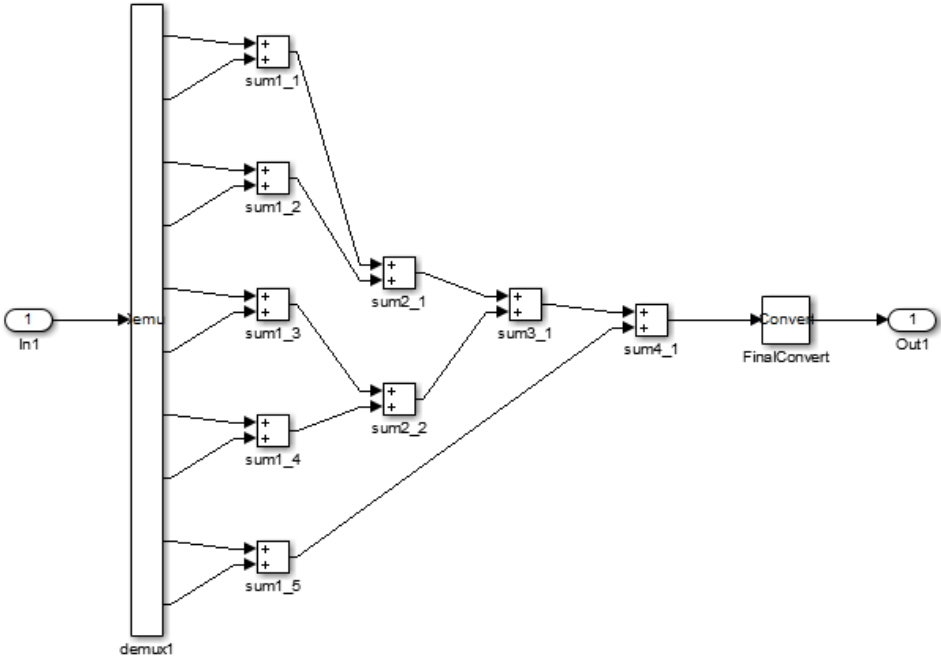


The vsum subsystem has been highlighted in cyan. This highlighting indicates that the subsystem differs in some respect from the vsum subsystem of the original model.

The following figure shows the vsum subsystem in the generated model. Observe that the Sum block is now implemented as a subsystem, which also appears highlighted.



The following figure shows the internal structure of the Sum subsystem.



The generated model implements the vector sum as a tree of adders (Sum blocks). The vector input signal is demultiplexed and connected, as five pairs of operands, to the five leftmost adders. The widths of the adder outputs increase from left to right, as required to avoid overflow in computing intermediate results.

Optimization

- “Automatic Iterative Optimization” on page 24-2
- “Generated Model and Validation Model” on page 24-5
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder™” on page 24-9
- “Optimization with Constrained Overclocking” on page 24-15
- “Resolve Numerical Mismatch with Delay Balancing” on page 24-17
- “Streaming” on page 24-23
- “Resource Sharing” on page 24-27
- “Delay Balancing” on page 24-32
- “Find Feedback Loops” on page 24-38
- “Hierarchy Flattening” on page 24-40
- “RAM Mapping for Simulink Models” on page 24-43
- “RAM Mapping With the MATLAB Function Block” on page 24-44
- “Distributed Pipelining” on page 24-49
- “Hierarchical Distributed Pipelining” on page 24-56
- “Constrained Output Pipelining” on page 24-61
- “Clock-Rate Pipelining” on page 24-63
- “Adaptive Pipelining” on page 24-68
- “Critical Path Estimation Without Running Synthesis” on page 24-78
- “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 24-89
- “Subsystem Optimizations for Filters” on page 24-102
- “Remove Redundant Logic and Optimize Unconnected Ports in Design” on page 24-114

Automatic Iterative Optimization

In this section...
“How Automatic Iterative Optimization Works” on page 24-2
“Automatic Iterative Optimization Output” on page 24-3
“Automatic Iterative Optimization Report” on page 24-3
“Requirements for Automatic Iterative Optimization” on page 24-4
“Limitations of Automatic Iterative Optimization” on page 24-4

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.

There are two ways to use `hdlcoder.optimizeDesign` to optimize your clock frequency:

- **Best clock frequency:** You specify the maximum number of iterations you want HDL Coder to perform, and the coder iterates to minimize the critical path in your design.
- **Target clock frequency:** You specify a clock frequency target for your design and the maximum number of iterations you want HDL Coder to perform. The coder iterates until it meets your target clock frequency or reaches the maximum number of iterations.

HDL Coder can also 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 specify your clock frequency goal and the maximum number of iterations. HDL Coder performs the following steps for each iteration:

- 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 with 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 `hdlsrc/your_model_name`
- 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`.

HDL Coder also saves

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 the following:

- Summary Section, with:
 - Final critical path latency.
 - Critical path latency and elapsed time for each iteration.
- Diagnostic Section, with:
 - 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 `hdlcoder.optimizeDesign`. 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 both the original model and generated model that are part of the critical path.

You may see a message that says a signal or component on the critical path cannot be traced back to the original model. HDL Coder may not be able to map its internal representation of your design back to the original design. Each optimization iteration changes the internal representation, so the final representation can have a structure that is different from your original design.

Requirements for Automatic Iterative Optimization

Your synthesis tool must be Xilinx ISE or Xilinx Vivado, and 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.
- Running automatic iterative optimization can take a long time, depending on the complexity of your design. To help mitigate the time cost, `hdlcoder.optimizeDesign` can regenerate code from a previous run, or resume from an interrupted run.
- Automatic iterative optimization is available from the command line only.
- 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 the synthesis tool generates after place and route, but is faster to obtain.

See Also

`hdlcoder.optimizeDesign`

Generated Model and Validation Model

In this section...

“Generated Model” on page 24-5

“Validation Model” on page 24-6

HDL Coder enables you to view the effect of HDL optimizations and block settings in the 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 an intermediate model 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 DUT and the generated HDL code. Delays that the code generator inserts are highlighted in the generated model in various colors. See the table below for different delays in code generation and the corresponding highlight color.

Delays	Highlight Color
Block implementation	Cyan
RAM mapping	
Constrained output pipelining	Green
Distributed pipelining	Orange
Input and output pipelining	
Delay balancing	
Clock-rate pipelining	

With timing and area optimizations, the generated model is substantially different from the original model. For example, you can see additional integer delays next to blocks if you request optimizations and additional balanced delays wherever necessary to maintain the accuracy of the algorithm. You see additional rates in the model if you request resource sharing or streaming optimization where the same operator is time multiplexed across multiple operations.

The generated model is used in RTL testbench generation. The input stimulus and output response are captured from the generated model instead of the original model because the generated model reflects the algorithms 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 `makehdl` or `hdlset_param`. See “Prefix for generated model name” on page 16-104.

You can also specify various options for the layout of the generated model. See “Layout Options” on page 16-107.

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. For example:

- 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 23-2.

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 `makehdl` or `hdlset_param`.

Customize the Validation Model

To customize the suffix of the generated validation name, use the `ValidationModelNameSuffix` property with `makehdl` or `hdlset_param`. See “Suffix for validation model name” on page 16-105.

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.

See Also

More About

- “Delay Balancing” on page 24-32

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 the 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.

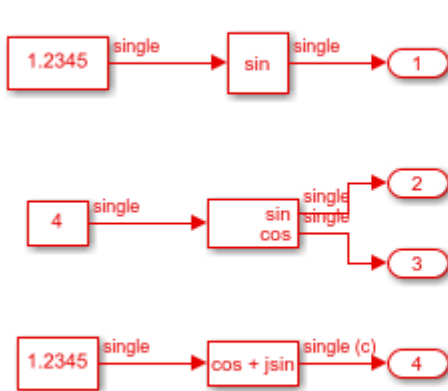
For example, $c = 3 * 5$ becomes $c = 15$.

This optimization works with any Simulink block that supports HDL code generation. For example:

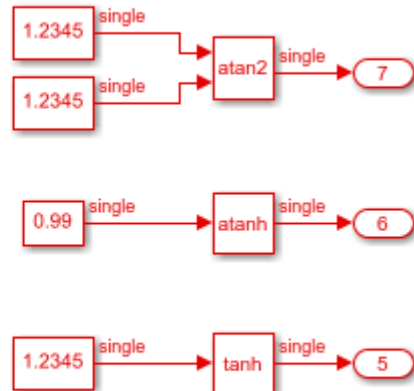
- Open the model `hdlcoder_constant_simplification`. Double-click the **Trigonometric Functions Subsystem**.

```
open_system('hdlcoder_constant_simplification')
open_system('hdlcoder_constant_simplification/Trigonometric Functions')
set_param('hdlcoder_constant_simplification', 'SimulationCommand', 'update');
```

Basic sin and cos functions



Arctangent and Hyperbolic functions

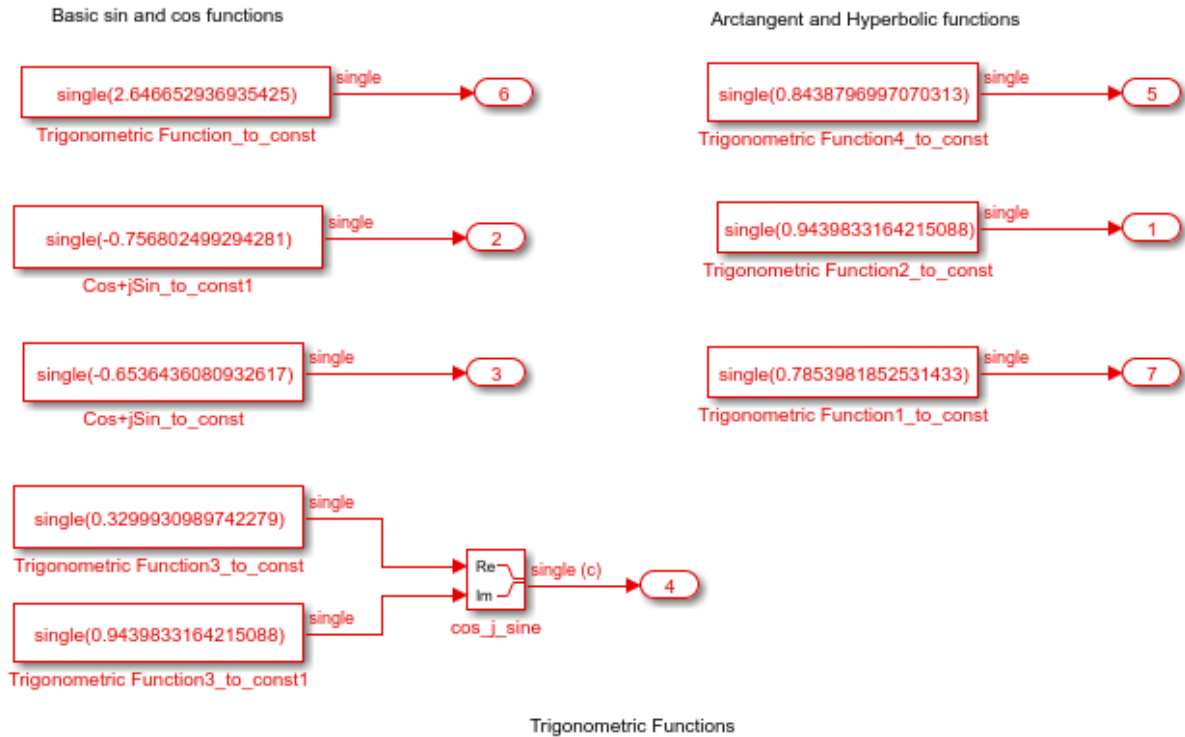


Trigonometric Functions

- 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.

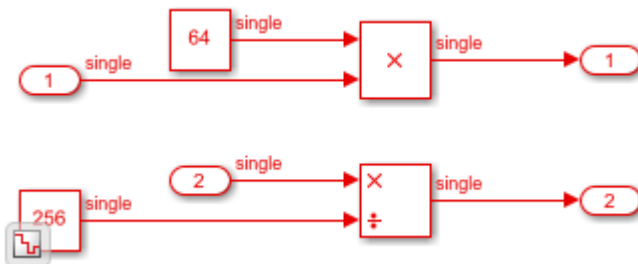
For example, $r1 = r2 / 2$ becomes $r1 = r2 \gg 1$.

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.

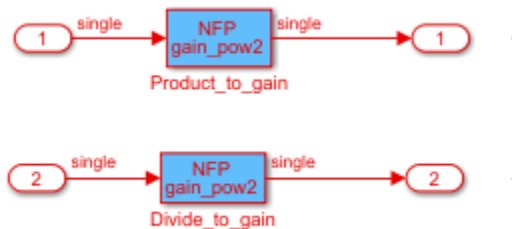
```
open_system('hdlcoder_slow_operation_replacement')
open_system('hdlcoder_slow_operation_replacement/DUT')
set_param('hdlcoder_slow_operation_replacement', 'SimulationCommand', 'update');
```



- 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.

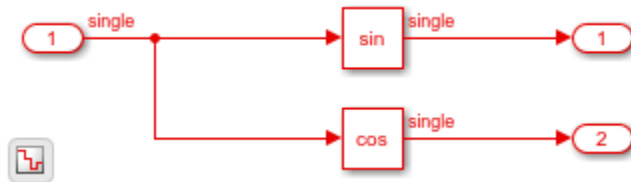
Combination of Multiple Operations

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, 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, on the **HDL Code Generation > Floating Point** pane, set **Floating-Point IP Library** to **Native Floating Point**. To learn about native floating-point support in HDL Coder, see [Generate Target-Independent HDL Code with Native Floating-Point](#).
- The optimizations preserve all comments from the blocks in the generated code. To learn about specifying comments to blocks, see [Generate Code with Annotations or Comments](#).
- 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](#).
- 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 24-5
- [“Remove Redundant Logic and Optimize Unconnected Ports in Design”](#) on page 24-114

Optimization with Constrained Overclocking

In this section...

“Why Constrain Overclocking?” on page 24-15

“Optimizations that Overclock Resources” on page 24-15

“How to Use Constrained Overclocking” on page 24-16

“Constrained Overclocking Limitations” on page 24-16

Why Constrain Overclocking?

Area and timing optimizations that you specify can result in upsampled rates in your design. For example, when you use the resource sharing optimization, the code generator overclocks the shared resources by an overclocking factor (OCF). The OCF depends on the number of shareable resources, N , and the **SharingFactor**, SF , 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 overclocking, use the **Oversampling factor** in conjunction with clock-rate pipelining to constrain the overclocking of your design.

Optimizations that Overclock Resources

Area and speed optimizations, and certain block implementations that you specify result in overclocking the resources in your design. For example, the following 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 using the **oversampling** parameter. 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 factor**. You can specify the **Oversampling factor** in the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box.

By default, the clock-rate pipelining optimization is enabled, and it works in conjunction with the **Oversampling factor** to make the DUT sample time slower than the actual clock rate. You can design your model at the base sample time and then set the **Oversampling factor** to N. This setting gives HDL Coder 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 specifying an **Oversampling factor** greater than 1, **ClockInputs** must be set to **Single**.

See Also

More About

- “Oversampling factor” on page 16-18
- “Clock-Rate Pipelining” on page 24-63

Resolve Numerical Mismatch with Delay Balancing

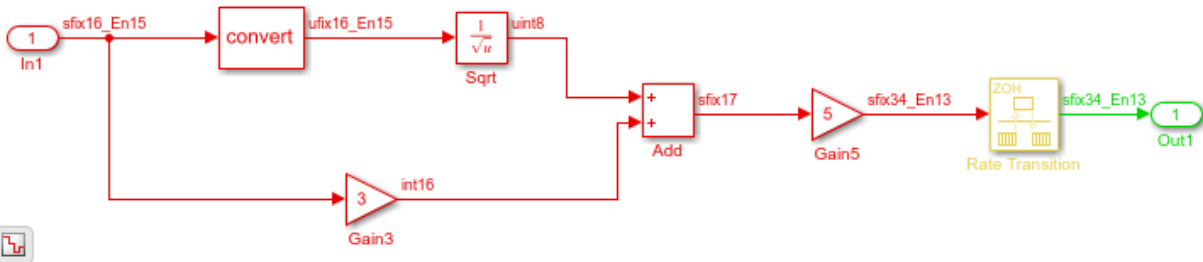
This example shows how to use delay balancing to resolve a numerical mismatch between the generated model and original model after HDL code generation.

Problem

The issue is that simulating the validation model results in a numerical 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, 'GenerateValidationModel', 'on', ...
        'TargetDirectory', 'C:/Temp/hdlsrc')
```

```
### Generating HDL for 'hdlcoder_resolve_delaybalancing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_resolve_delaybalancing')">hdlcoder_resolve_delaybalancing</a>.
### Starting HDL check.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_resolve_delaybalancing')">gm_hdlcoder_resolve_delaybalancing</a>.
### 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.
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_iv as C:\Temp\hdlsrc
```

```

### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_core as C:\Temp\hdl
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt as C:\Temp\hdlsrc\hdlcod
### Working on Subsystem_tc as C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem
### Working on hdlcoder_resolve_delaybalancing/Subsystem as C:\Temp\hdlsrc\hdlcoder_res
### Generating package file C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_pl
### Creating HDL Code Generation Check Report file://C:\Temp\hdlsrc\hdlcoder_resolve_d
### HDL check for 'hdlcoder_resolve_delaybalancing' complete with 0 errors, 2 warnings
### HDL code generation complete.

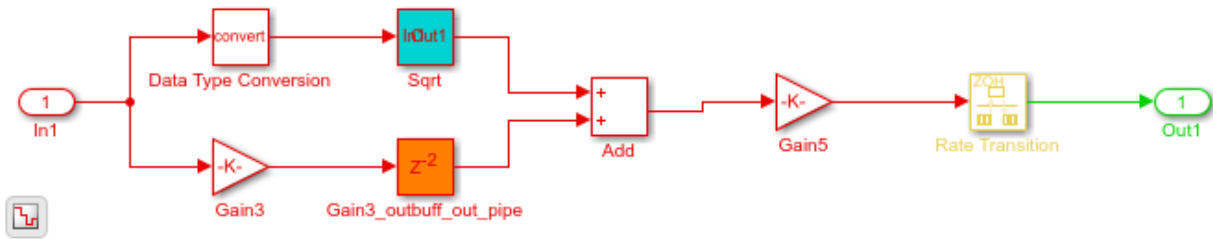
```

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 numerical results produced by the original DUT.

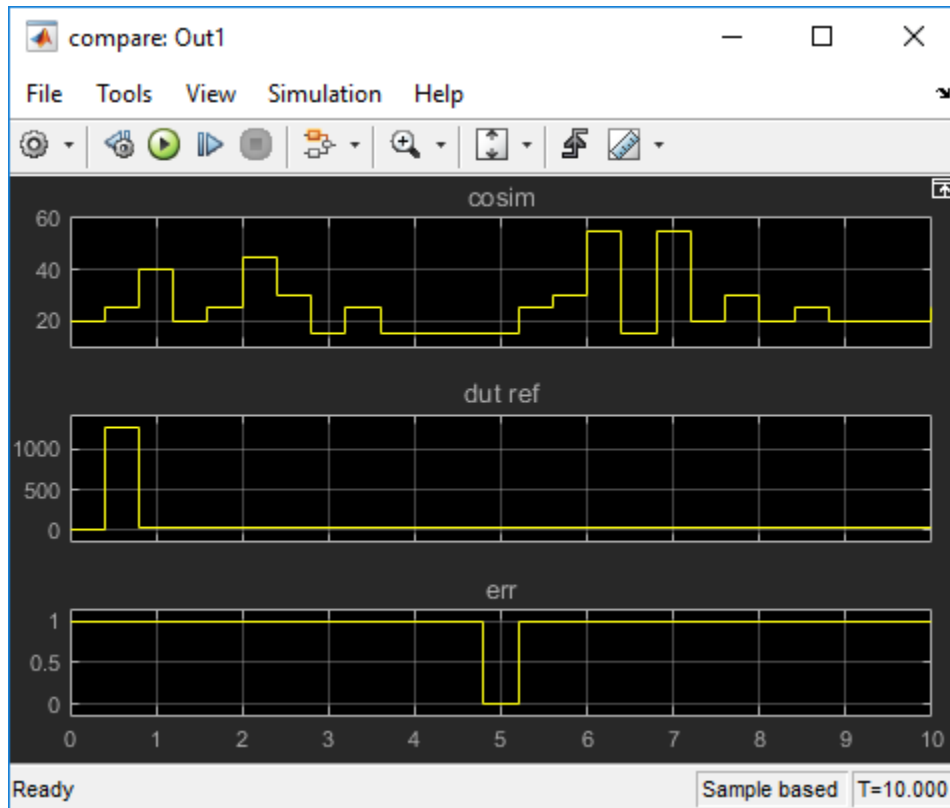
```

valmodelname = 'gm_hdlcoder_resolve_delaybalancing_vnl';
valmodelsys = 'gm_hdlcoder_resolve_delaybalancing_vnl/Subsystem';
load_system(valmodelname)
open_system(valmodelsys)
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 numerical 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)
```

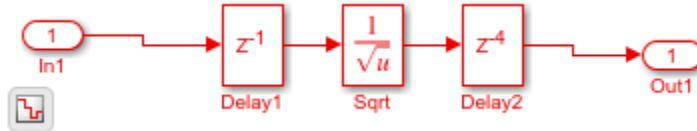
```
% Set Model 'hdlcoder_resolve_delaybalancing' HDL parameters
hdlset_param('hdlcoder_resolve_delaybalancing', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_resolve_delaybalancing', 'GenerateHDLTestBench', 'off');
hdlset_param('hdlcoder_resolve_delaybalancing', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_resolve_delaybalancing', 'HDLSubsystem', 'hdlcoder_resolve_delay
```

```
% Set Gain HDL parameters
hdlset_param('hdlcoder_resolve_delaybalancing/Subsystem/Gain3', 'OutputPipeline', 2);

hdlset_param('hdlcoder_resolve_delaybalancing/Subsystem/Sqrt', 'Architecture', 'RecipS
```

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, 'BalanceDelays', 'on', ...
        'GenerateValidationModel', 'on', ...
        'TargetDirectory', 'C:/Temp/hdlsrc')

### Generating HDL for 'hdlcoder_resolve_delaybalancing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlco
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced addit
```



```

### The delay balancing feature has automatically inserted matching delays for compensa
### The DUT requires an initial pipeline setup latency. Each output port experiences th
### Output port 0: 2 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_resolve_
### 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
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_iv as C:\Temp\hdlsrc
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_core as C:\Temp\hdl
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt as C:\Temp\hdlsrc\hdlcode
### Working on Subsystem_tc as C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem
### Working on hdlcoder_resolve_delaybalancing/Subsystem as C:\Temp\hdlsrc\hdlcoder_res
### Generating package file C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_pl
### Creating HDL Code Generation Check Report file://C:\Temp\hdlsrc\hdlcoder_resolve_de
### HDL check for 'hdlcoder_resolve_delaybalancing' complete with 0 errors, 0 warnings.
### HDL code generation complete.

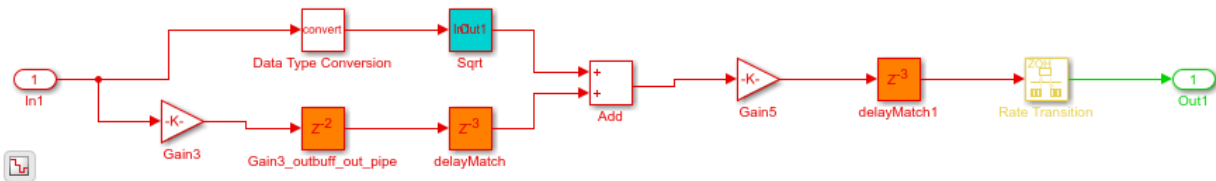
```

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.

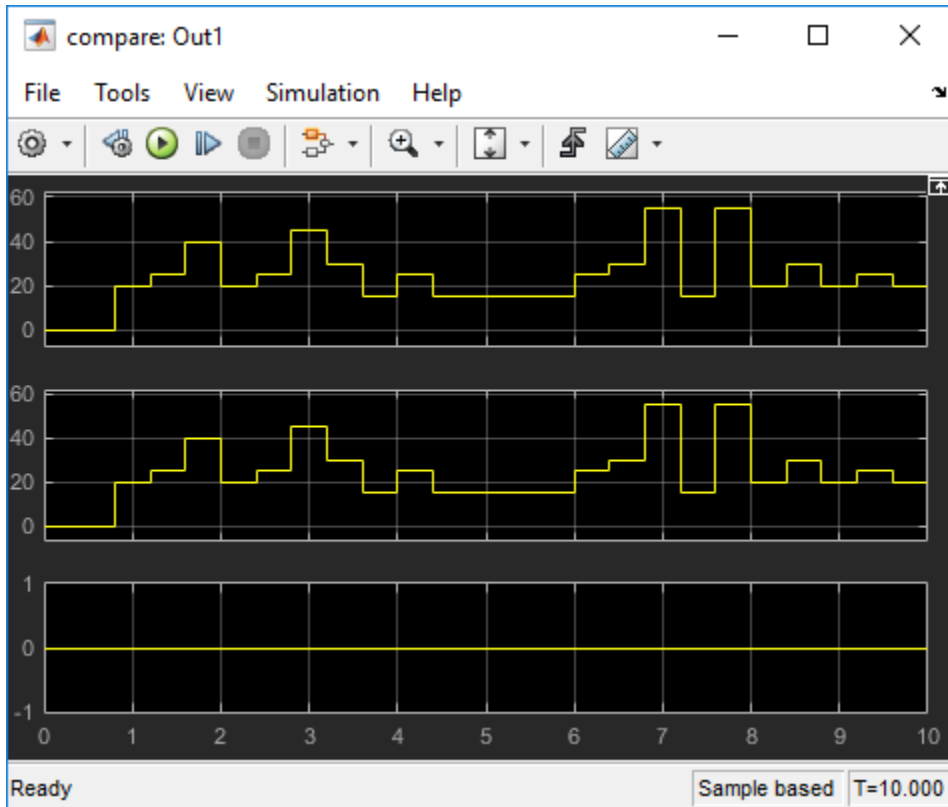
```

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™”

More About

- “Delay Balancing” on page 24-32
- “Generated Model and Validation Model” on page 24-5
- “Check delay balancing setting” on page 38-12

Streaming

In this section...

“What Is Streaming?” on page 24-23

“Specify Streaming” on page 24-24

“How to Determine Streaming Factor and Sample Time” on page 24-24

“Determine Blocks That Support Streaming” on page 24-25

“Requirements for Streaming Subsystems” on page 24-25

“Streaming Report” on page 24-25

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 24-63.

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 24-5.

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 $S_{base} / \text{Oversampling}$. 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. As a best practice, run the `checkhdl` function before generating streaming code for a subsystem. `checkhdl` reports blocks in your subsystem that are incompatible with 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 24-27
- “Clock-Rate Pipelining” on page 24-63

Resource Sharing

In this section...

“How Resource Sharing Works” on page 24-27
 “Benefits and Costs of Resource Sharing” on page 24-28
 “Shareable Resources in Different Blocks” on page 24-28
 “Specify Resource Sharing” on page 24-29
 “Limitations for Resource Sharing” on page 24-29
 “Block Requirements for Resource Sharing” on page 24-29
 “Resource Sharing Report” on page 24-30

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 *SF* for a subsystem or a MATLAB Function block. HDL Coder tries to identify a certain number of identical, shareable resources *N* up to *SF*. How the code generator shares these resources depends on *N*, *SF*, and the `Oversampling` factor.

By default, the `Oversampling` factor is 1, and resource sharing overclocks the shared resources by an overclocking factor (OCF) that depends on the remainder of *SF* and *N*.

```

if rem(SF,N) == 0
    OCF = N;
else
    OCF = SF;
end
  
```

If you specify an `Oversampling` factor greater than 1, your design operates a faster clock rate on the target hardware because clock-rate pipelining is enabled by default. 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:

$$\text{Overclocking factor} = (\text{block_rate} \div \text{DUT_base_rate}) \times \text{Oversampling}$$

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 24-5.

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 20-120.

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);
```

Limitations for Resource Sharing

HDL Coder does not support model references for resource sharing.

Block Requirements for Resource Sharing

Blocks to be shared must have the following requirements:

- Single-rate.
- No infinite sample rate. The DUT must not contain blocks with **Sample time** set to **Inf**. For example, Constant blocks must have **Sample time** set to **-1**. To set the sample time to **-1** for all Constant blocks in your DUT, use the following MATLAB code:

```
blks = find_system(dut, 'BlockType', 'Constant');
for i = 1:length(blks)
    set_param(blks{i}, 'SampleTime', '-1');
end
```

- 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 20-114
- “Resource Sharing of Subsystems and Floating-Point IPs” on page 20-119

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.

See Also

Related Examples

- Resource Sharing For Area Optimization
- “Single-rate Resource Sharing Architecture”

More About

- “Clock-Rate Pipelining” on page 24-63
- “Streaming” on page 24-23

Delay Balancing

In this section...

“Why Use Delay Balancing?” on page 24-32

“Specify Delay Balancing” on page 24-33

“Delay Balancing Limitations” on page 24-34

“Delay Balancing Report” on page 24-37

Why Use Delay Balancing?

HDL Coder supports several optimizations, block implementations, and options that introduce discrete delays into the model, with the goal of more efficient hardware usage or achieving higher clock rates. Examples include:

- *Optimizations*: Optimizations such as output pipelining, streaming, or resource sharing can introduce delays.
- *Cascading*: Some blocks support cascade implementations, which introduce a cycle of delay in the generated code.
- *Block implementations*: Some block implementations such as the Newton-Raphson architecture inherently introduce delays in the generated code.

When optimizations or block implementation options introduce delays along the critical path in a model, the numerics of the original model and generated model or HDL code can differ because equivalent delays are not introduced on other, parallel signal paths. Manual insertion of compensating delays along the other paths is possible, but is error prone and does not scale well to large models with many signal paths or multiple sample rates.

To help you solve this problem, HDL Coder supports delay balancing. By default, delay balancing is enabled on the model. The code generator detects introduction of new delays along one path, and then inserts matching delays on the other paths. When delay balancing is enabled, the generated model is functionally equivalent to the original model. It is not recommended that you disable delay balancing on the model. If you disable this setting, HDL Coder generates a warning that numerical differences can occur in the validation model. To fix this warning, enable **Balance delays** on the model or run the model check “Check delay balancing setting” on page 38-12.

Specify Delay Balancing

You can set delay balancing for an entire model. For finer control, you can also set delay balancing for subsystems within the top-level DUT subsystem.

Set Delay Balancing for a Model

Use the following `makehdl` properties to set delay balancing for a model:

- **BalanceDelays**: By default, model-level delay balancing is enabled, and subsystems within the model inherit the model-level setting. To learn how to set delay balancing for a model, see “Balance delays” on page 14-3.
- **GenerateValidationModel**: By default, validation model generation is disabled. When you enable delay balancing, generate a validation model to view delays and other differences between your original model and the generated model. To learn how to enable validation model generation, see the **Generate validation model** section in “Model Generation for HDL Code” on page 16-101.

For example, the following commands generate HDL code with delay balancing and generate a validation model.

```
dut = 'ex_rsqrt_delaybalancing/Subsystem';  
makehdl(dut, 'BalanceDelays', 'on', 'GenerateValidationModel', 'on');
```

Disable Delay Balancing for a Subsystem

You can disable delay balancing for an entire model or disable a subsystem within the top-level DUT subsystem. For example, if you do not want to balance delays for a control path, you can put the control path in a subsystem, and disable delay balancing for that subsystem.

To disable delay balancing for a subsystem within the top-level DUT subsystem, disable delay balancing at the model level. When you disable delay balancing for the model, the validation model does not compensate for latency inserted in the generated model due to optimizations or block implementations. The validation model can therefore show mismatches between the original model and generated model.

To disable delay balancing for a subsystem within the top-level DUT subsystem:

- 1 Disable delay balancing for the model.
- 2 Enable delay balancing for the top-level DUT subsystem.
- 3 Disable delay balancing for a subsystem within the DUT subsystem.

To learn how to set delay balancing for a subsystem, see “Set Delay Balancing For a Subsystem” on page 21-5.

Delay Balancing Limitations

If delay balancing is unsuccessful, `hdlcoder.optimizeDesign` cannot optimize the generated HDL code.

Unsupported Blocks

The following blocks and subsystems do not support delay balancing:

- Triggered Subsystem
- Atomic Subsystem
- HDLCosimulation
- Data Type Duplicate
- Decrement To Zero
- Frame Conversion
- Ground
- FFT HDL Optimized
- LMS Filter
- Model Reference
- To VCD File
- Magnitude-Angle to Complex

Block Mode Restrictions

The following block implementations do not support delay balancing:

- `hdldefaults.ConstantSpecialHDL Emission`
- `hdldefaults.NoHDL`

Subsystem-Level Restrictions

HDL Coder does not support delay balancing, if:

- There are multiple instances of an Atomic Subsystem in different conditional subsystems.

In the Block Parameters dialog box of the Atomic Subsystem, you can set **Function packaging** to **Nonreusable function**.

- The “BalanceDelays” on page 21-5 block property for all instances of an Atomic Subsystem or Model Reference resolves to a different value.

To fix this error, disable `BalanceDelays` for all instances of the Atomic Subsystem or Model Reference.

- The block is inside a conditional subsystem and has pipeline delays.
- 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.

Other Restrictions

HDL Coder does not support delay balancing, if:

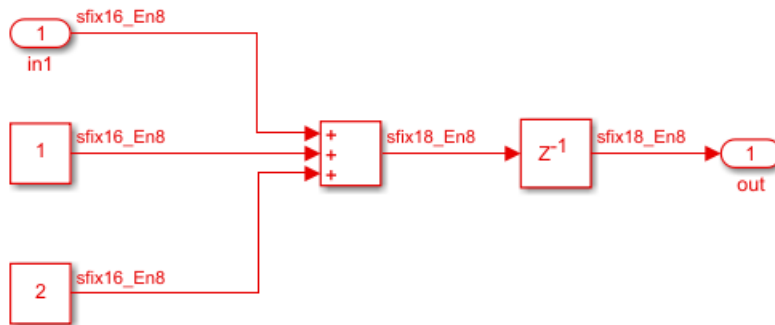
- Delays are introduced in a feedback loop and HDL Coder cannot balance the path delays. For example, if you apply clock-rate pipelining inside a feedback loop, HDL Coder introduces a delay at the clock-rate, and can cause delay balancing to fail.

To reduce the number of clock-rate delays, increase the “Oversampling factor” on page 16-18.

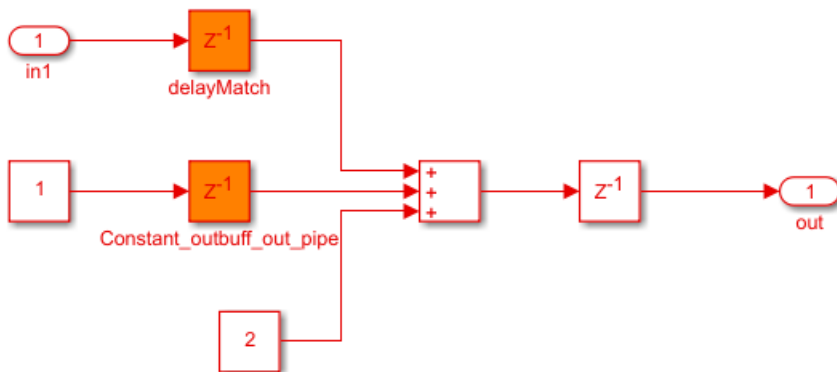
- The sample time is not discrete or the ratio of sample times of the fastest to the slowest rate is too large.

Delay Balancing with Constants

When you have Constant blocks as inputs inside the DUT Subsystem for which delay balancing is enabled, you see an initial simulation mismatch in the validation model. Consider this model inside a DUT Subsystem. The Constant block that outputs a value of 1 has the HDL block property **OutputPipeline** set to 1 .



This figure displays the generated validation model. You see that delay balancing added a matching delay to the input port to balance the pipeline register inserted for the Delay block. The code generator does not insert a matching delay on the parallel path containing the Constant block with the value 2 because the output value of the block is a constant. This delay not inserted results in an initial simulation mismatch



To resolve the simulation mismatch, in the validation model, manually add a matching delay at the output of the Constant block with the value 2.

Delay Balancing Report

To see the delay balancing 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 code for each subsystem, model reference, or MATLAB Function block, HDL Coder produces the optimization report. In the report, select the **Delay Balancing** section of the report.

The Delay Balancing Report shows latency changes, pipeline delay and phase delay at the output ports, and the number of pipelines added at the output ports to match the delays. If delay balancing fails, the report mentions the criteria that was violated and displays the link to any block or subsystem that caused delay balancing to fail.

See Also

Related Examples

- “Delay Balancing and Validation Model Workflow In HDL Coder™”

More About

- “Resolve Numerical Mismatch with Delay Balancing” on page 24-17
- “Create and Use Code Generation Reports” on page 25-2
- “Validation Model” on page 24-6

Find Feedback Loops

In this section...

“Specify Highlighting of Feedback Loops” on page 24-38

“Remove Highlighting” on page 24-39

“Limitations” on page 24-39

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. This setting is available:

- In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Advanced** tab, select **Highlight feedback loops inhibiting delay balancing and optimizations**.
- 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 (').

See Also

More About

- “Diagnostics for Optimizations” on page 16-110
- “Create and Use Code Generation Reports” on page 25-2
- “Generated Model and Validation Model” on page 24-5

Hierarchy Flattening

In this section...

“What Is Hierarchy Flattening?” on page 24-40

“When to Flatten Hierarchy” on page 24-40

“Considerations” on page 24-40

“How to Flatten Hierarchy” on page 24-41

“Limitations for Hierarchy Flattening” on page 24-42

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” on page 16-70.

- 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`.

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 black box implementation or model reference.
- A Triggered Subsystem when “Use trigger signal as clock” on page 16-49 is enabled.
- A masked subsystem that contains any of the following:
 - Bus.
 - Enumerated data type.
 - Lookup table blocks: 1-D Lookup Table, 2-D Lookup Table, Cosine HDL Optimized, Direct LookupTable (n-D), Prelookup, Sine HDL Optimized, n-D Lookup Table.
 - MATLAB System block.
 - Stateflow blocks: Chart, State Transition Table, Sequence Viewer.
 - Blocks with a pass-through or no-op implementation. See “Pass through, No HDL, and Cascade Implementations” on page 21-76.

Note This option removes subsystem boundaries before code generation. It does not necessarily generate HDL code with a completely flat hierarchy.

RAM Mapping for Simulink Models

RAM mapping is an area optimization. You can map to RAMs in HDL code by using:

- UseRAM to map delays to RAM. For details, see “UseRAM” on page 21-34.
- MapPersistentVarsToRAM to map persistent arrays in a MATLAB Function block to RAM. For details, see “MapPersistentVarsToRAM” on page 21-20.
- 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
- Blocks with a RAM implementation.

See Also

Simulink Configuration Parameters

“RAM Mapping” on page 14-5

More About

- “UseRAM” on page 21-34
- “MATLAB Function Block Design Patterns for HDL” on page 29-23
- “MapPersistentVarsToRAM” on page 21-20
- “RAM Mapping With the MATLAB Function Block” on page 24-44

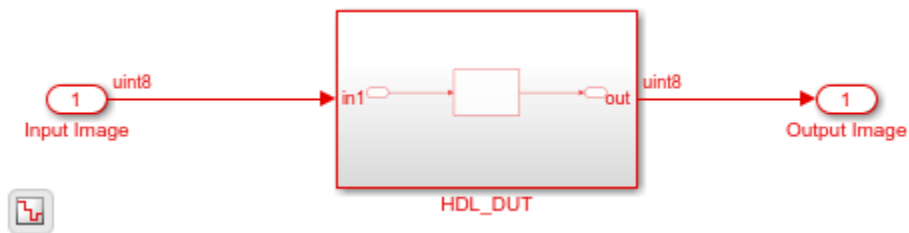
RAM Mapping With the MATLAB Function Block

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

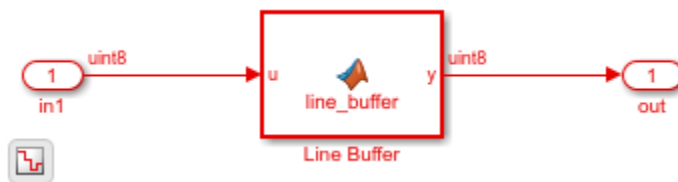
Open the model `hdlcoder_ram_mapping_matlab_function`.

```
open_system('hdlcoder_ram_mapping_matlab_function')
set_param('hdlcoder_ram_mapping_matlab_function', 'SimulationCommand', 'Update')
```



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.

```
m1_subsys = 'hdlcoder_ram_mapping_matlab_function/HDL_DUT/Line Buffer';
hdlset_param(m1_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(bits)`.

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.

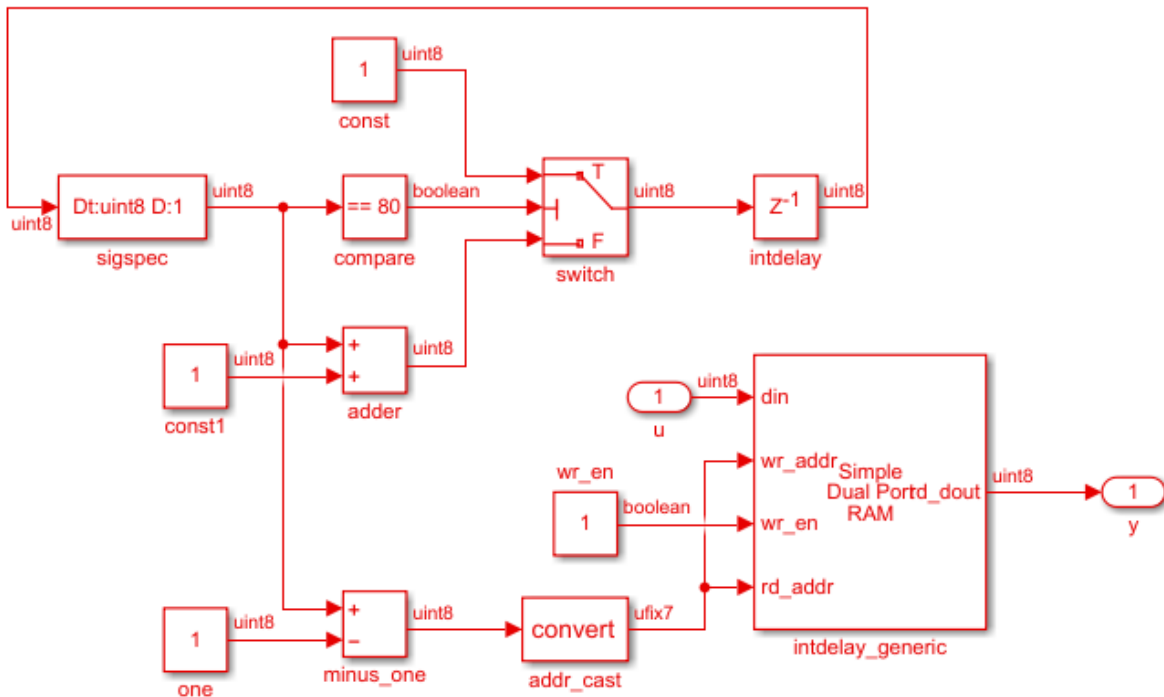
1. Enable the MATLAB Datapath HDL architecture and then set the MapPersistentVarsToRAM parameter on the MATLAB Function block.

```
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

More About

- “RAM Mapping” on page 14-5
- “MATLAB Function Block Design Patterns for HDL” on page 29-23
- “MapPersistentVarsToRAM” on page 21-20

Distributed Pipelining

In this section...

“What Is Distributed Pipelining?” on page 24-49

“Benefits and Costs of Distributed Pipelining” on page 24-51

“Requirements for Distributed Pipelining” on page 24-52

“Specify Distributed Pipelining” on page 24-52

“Limitations of Distributed Pipelining” on page 24-52

“Distributed Pipelining Report” on page 24-54

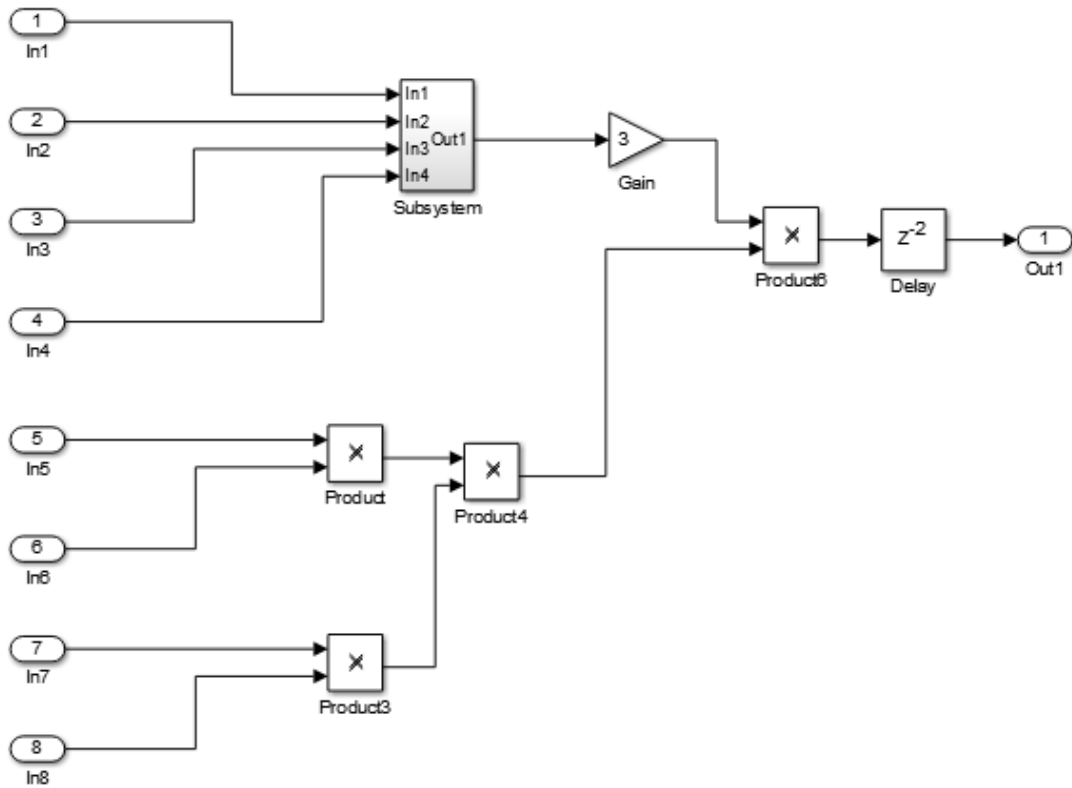
“Selected Bibliography” on page 24-54

What Is Distributed Pipelining?

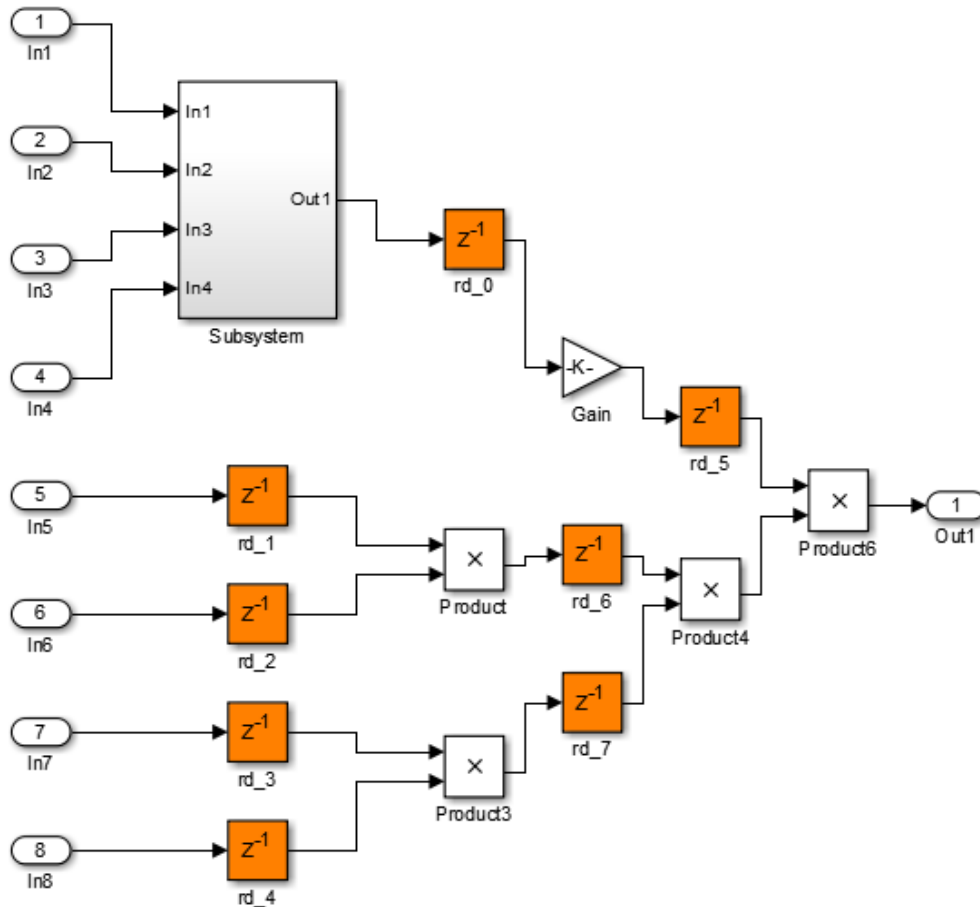
Distributed pipelining, or register retiming, is a speed optimization that moves existing delays in a design to reduce the critical path while preserving functional behavior.

The HDL Coder software uses an adaptation of the Leiserson-Saxe retiming algorithm.

For example, in the following model, there is a delay of 2 at the output.



The following 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.

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.

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 specify distributed pipelining for a subsystem, and Stateflow charts and MATLAB Function blocks within a subsystem. See “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-39.

To specify distributed pipelining using the UI:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the Subsystem and then click **HDL Block Properties**. Set **DistributedPipelining** to **on** and click **OK**.
- Right-click the Subsystem and select **HDL Code > HDL Block Properties**. Set **DistributedPipelining** to **on** and click **OK**.

To enable distributed pipelining, on the command line, enter:

```
hdlset_param('path/to/block', 'DistributedPipelining', 'on')
```

Tip Output data could 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)” on page 18-23
 - IgnoreDataChecking
-

Limitations of Distributed Pipelining

The distributed pipelining optimization has the following limitations:

- Your pipelining results might not be optimal in hardware because the operator latencies in your target hardware may differ from the estimated operator latencies used by the distributed pipelining algorithm.
- The HDL Coder software generates pipeline registers at the outputs in the following situations instead of distributing the registers to reduce critical path:
 - Stateflow chart containing a state, local variable, or a matrix with statically unresolvable index.
- HDL Coder distributes pipeline registers around the following blocks instead of within them:
 - Model
 - Sum (Cascade implementation)
 - 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
- If you enable distributed pipelining for a subsystem that contains these blocks, HDL Coder generates a message during code generation. To fix this message, place these blocks inside one or more subsystems within the original subsystem, and disable hierarchical distributed pipelining. 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
- Frame Conversion

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. 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 **Distributed Pipelining** section, you see the effect of the distributed pipelining optimization. If distributed pipelining is unsuccessful, the report shows diagnostic messages and offending blocks that caused distributed pipelining to fail.

If distributed pipelining is successful, the report displays comparative listings of registers before and after you apply the distributed pipelining transform.

Selected Bibliography

Leiserson, C.E, and James B. Saxe. "Retiming Synchronous Circuitry." *Algorithmica*. Vol. 6, Number 1, 1991, pp. 5-35.

See Also

More About

- “Distributed Pipelining” on page 14-12
- “Diagnostics for Optimizations” on page 16-110

Hierarchical Distributed Pipelining

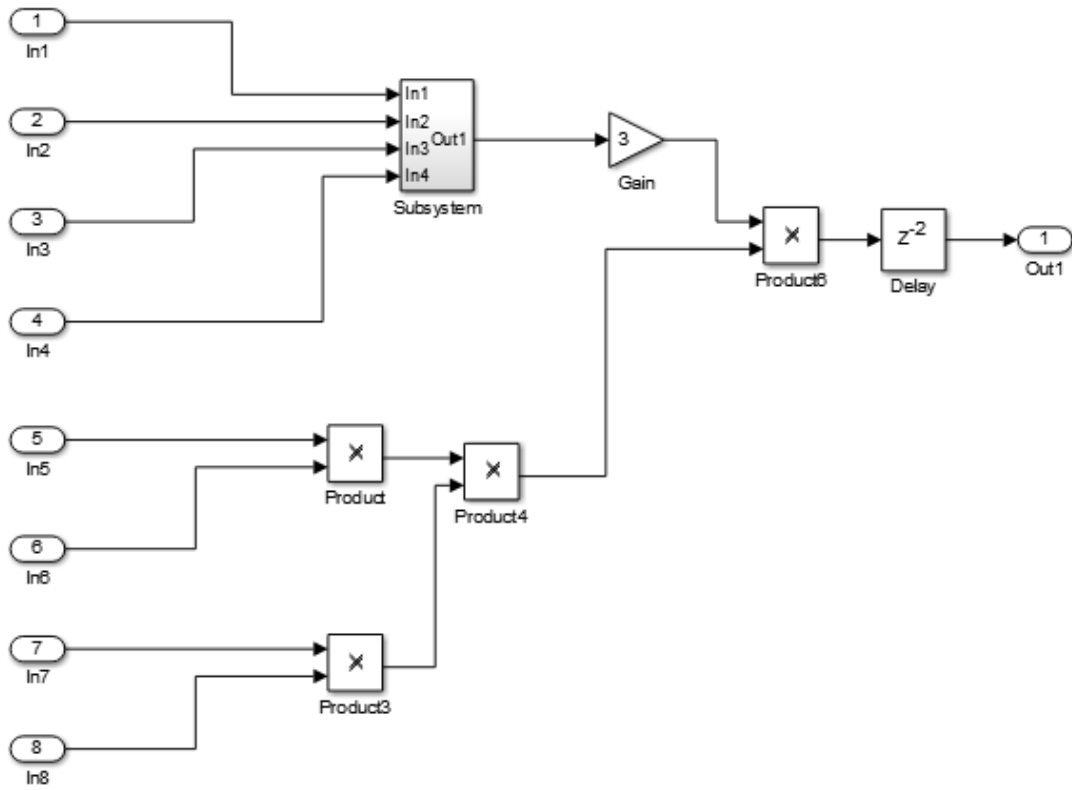
What Is Hierarchical Distributed Pipelining?

Hierarchical distributed pipelining extends the scope of distributed pipelining by moving delays across hierarchical boundaries within a subsystem while preserving subsystem hierarchy.

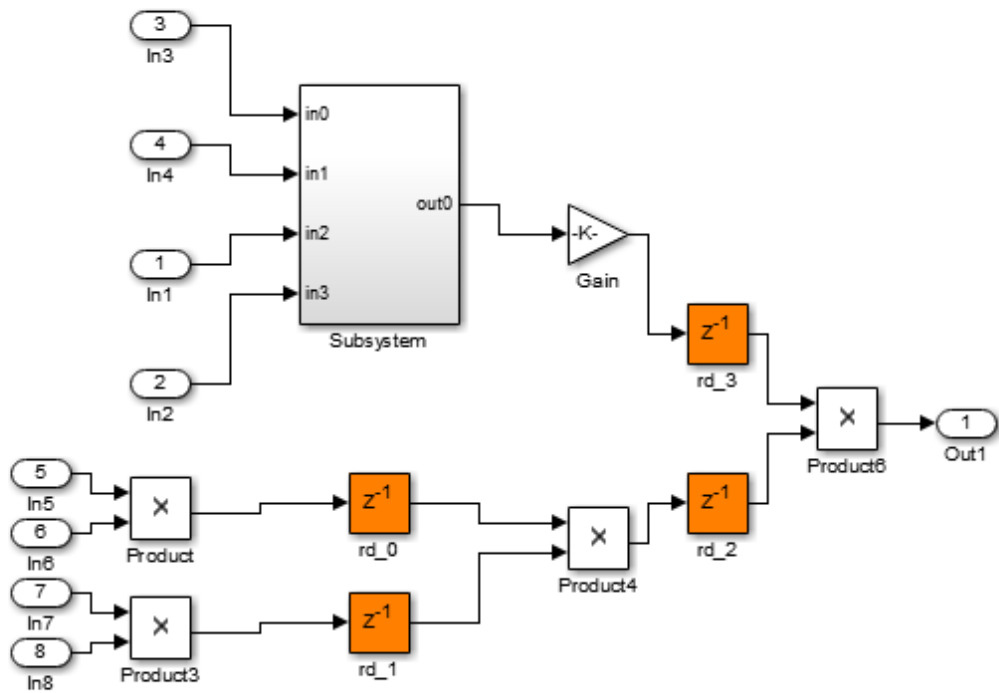
If a subsystem in the hierarchy does not have distributed pipelining enabled, HDL Coder does not move delays across that subsystem.

How Hierarchical Distributed Pipelining Works

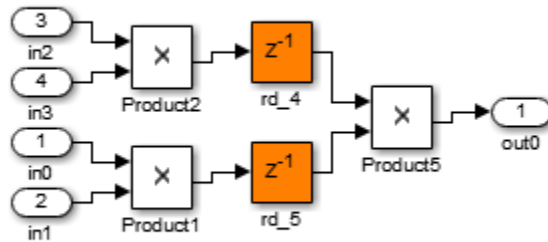
For example, the following model has one level of subsystem hierarchy:



The following diagram shows the model after applying hierarchical distributed pipelining:



The subsystem now contains pipeline registers:



Benefits of Hierarchical Distributed Pipelining

Hierarchical distributed pipelining enables distributed pipelining to operate on a larger part of your design, which increases the chance that distributed pipelining can further reduce your critical path.

Hierarchical distributed pipelining preserves the original subsystem hierarchy, which enables you to trace the changes that occur during pipelining for nested Subsystem blocks.

Specify Hierarchical Distributed Pipelining

You can specify hierarchical distributed pipelining for your model. To specify hierarchical distributed pipelining using the UI, in the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Click the Subsystem and then click **HDL Block Properties**. Set **DistributedPipelining** to **on**

- 1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
- 2 Click **Settings**. In the **HDL Code Generation > Optimization > Pipelining** tab, select **Hierarchical distributed pipelining** and click **OK**.

To enable hierarchical distributed pipelining, on the command line, enter:

```
hdlset_param('modelname', 'HierarchicalDistPipelining', 'on')
```

Limitations of Hierarchical Distributed Pipelining

Hierarchical distributed pipelining must be disabled if your DUT subsystem contains a model reference.

Hierarchical Distributed Pipelining Report

To see the hierarchical distributed pipelining 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 **Distributed Pipelining** section, you see the effect of the hierarchical distributed pipelining optimization. If hierarchical distributed pipelining is unsuccessful, the report shows diagnostic messages and offending blocks that caused hierarchical distributed pipelining to fail.

If hierarchical distributed pipelining is successful, the report displays colored sections to distinguish between different regions where HDL Coder applied hierarchical distributed pipelining.

Selected Bibliography

Leiserson, C.E, and James B. Saxe. "Retiming Synchronous Circuitry." *Algorithmica*. Vol. 6, Number 1, 1991, pp. 5-35.

Constrained Output Pipelining

In this section...

“What Is Constrained Output Pipelining?” on page 24-61

“When to Use Constrained Output Pipelining” on page 24-61

“Requirements for Constrained Output Pipelining” on page 24-61

“Specify Constrained Output Pipelining” on page 24-62

“Limitations of Constrained Output Pipelining” on page 24-62

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 24-63
- “How Clock-Rate Pipelining Works” on page 24-64
- “Clock-Rate Pipelining and Hierarchy Flattening” on page 24-64
- “Clock-Rate Pipelining for DUT Output Ports” on page 24-65
- “Best Practices for Clock-Rate Pipelining” on page 24-65
- “Specify Clock-Rate Pipelining” on page 24-66
- “Limitations for Clock-Rate Pipelining” on page 24-66

When you enable speed and area optimizations that insert pipeline registers, use the clock-rate pipelining optimization to identify multicycle paths in your design. Clock-rate pipelining inserts pipeline registers at the faster clock rate, which improves clock frequency without introducing additional latency or by adding minimal latency.

Rationale for Clock-Rate Pipelining

The code generator introduces pipelines when you specify certain block implementations or enable some optimizations on the model, such as:

- Multi-cycle block implementations
- Input and output pipelining
- Distributed pipelining
- Floating-point library mapping
- Native floating-point HDL code generation
- Resource sharing
- Streaming

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

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 DUT base sample time are part of the slow path, and are potential candidates for clock-rate pipelining. In these slow paths, the code generator enables optimizations to introduce pipeline delays at the clock rate.

If you specify an “Oversampling factor” on page 16-18 greater than one, the DUT sample time becomes slower than the actual clock rate. The code generator determines the maximum number of clock-rate pipelines that it can insert based on the DUT-to-block sample time ratio and the oversampling factor.

Maximum number of clock-rate delays = $(block_rate \div DUT_base_rate) \times Oversampling$

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. The code generator 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, the code generator introduces several clock-rate pipelines corresponding to the ratio of the data rate to the clock rate.

HDL Coder generates a script that highlights blocks in your model that are obstacles to clock-rate pipelining and a script to clear the highlighting. Sometimes, if the code generator is unable to implement resource sharing or streaming at the clock rate, it displays a code generation error with a recommendation for changing the `Oversampling` value. To turn off highlighting, either click the `clearhighlighting` script in the MATLAB Command Window, or, in Simulink, select **Display > Remove Highlighting**.

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 24-40.

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, the code generator 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 or VHDL files for the various Subsystem blocks in your design.

Clock-Rate Pipelining for DUT Output Ports

To insert DUT output port pipeline registers at the clock rate instead of the data rate, select the **Allow clock-rate pipelining of DUT output ports** option or use the `ClockRatePipelineOutputPorts` property. This option changes the timing of your DUT interface because it changes the sample time of your DUT output ports from a slow rate to the clock rate. To adjust for the difference in timing, HDL Coder generates messages that provides 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.`

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 with the original output. The RTL test bench logs the output data at the input of the Rate Transition and compares it with the DUT output in the RTL simulation.

Best Practices for Clock-Rate Pipelining

- If your design uses a Rate Transition block, switch the Rate Transition block with a Downsample block that has nonzero **Sample offset**. Clock-rate pipelining optimizes the Downsample block by avoiding the additional latency that the Rate Transition block can introduce, which saves area and timing.
- Design your DUT at one rate and specify the **Oversampling factor**. Avoid using Rate Transition, Upsample, Downsample, or other rate-changing blocks.

Specify Clock-Rate Pipelining

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 21-6.

Limitations for Clock-Rate Pipelining

These blocks inhibit clock-rate pipelining, and therefore delimit clock-rate pipelining regions:

- Counter Free-Running
- Counter Limited
- Deserializer1D
- Discrete PID Controller
- Dual Port RAM
- Dual Rate Dual Port RAM
- FFT HDL Optimized
- HDL Cosimulation
- HDL FIFO
- HDL Counter
- Hit Crossing

- HDL Minimum Resource FFT
- HDL Streaming FFT
- MATLAB System, if it uses persistent variables
- Rate Transition
- Serializer1D
- Simple Dual Port RAM
- Single Port RAM
- Subsystem, if FlattenHierarchy is not enabled

The code generator does not support clock-rate pipelining for:

- Black box subsystem or black box model reference blocks.
- Subsystems that contain blocks not supported for clock-rate pipelining.
- Altera DSP Builder subsystems.
- Xilinx System Generator subsystems
- Communications Toolbox blocks.
- DSP System Toolbox blocks, except for Delay and Discrete FIR Filter.
- Stateflow blocks.

See Also

Related Examples

- “Clock Rate Pipelining”

More About

- “Clock Rate Pipelining” on page 14-15
- “Delay Balancing” on page 24-32
- “Hierarchy Flattening” on page 24-40

Adaptive Pipelining

In this section...

“Requirements” on page 24-68

“Specify Adaptive Pipelining” on page 24-69

“Supported Blocks” on page 24-69

“Pipeline Insertion for Lookup Tables” on page 24-70

“Pipeline Insertion for Rate Transition and Downsample Blocks” on page 24-71

“Pipeline Insertion for Product and Gain Blocks” on page 24-71

“Pipeline Insertion for Multiply-Add and Multiply-Accumulate Blocks” on page 24-73

“Pipeline Insertion for MATLAB Function Blocks” on page 24-75

“Adaptive Pipelining Report” on page 24-76

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 the **Oversampling factor**.
- Resource sharing which saves area and timing because the code generator shares resources and inserts adaptive pipeline registers.

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.
- 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.

Disable Adaptive Pipelining for a Model

By default, adaptive pipelining is enabled at the model level. You can disable 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:
 - 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 **Adaptive pipelining** and click **OK**.
- At the command line, use the `makehdl` or `hdlset_param` function to set “Adaptive pipelining” on page 14-18 to `off`.

Disable Adaptive Pipelining for a Subsystem

If you want HDL Coder to selectively disable adaptive pipelines for a subsystem in your model, set **AdaptivePipelining** to `off` for that subsystem.

To learn how to set adaptive pipelining for a subsystem, see “Set Adaptive Pipelining For a Subsystem” on page 21-4.

Supported Blocks

Adaptive pipelining supports these lookup tables, multipliers, multiply accumulate, and rate transition blocks for automatic pipeline insertion.

- n-D Lookup Table

- Direct Lookup Table (n-D)
- Sine HDL Optimized
- Cosine HDL Optimized
- Downsample
- Rate Transition
- Product
- Gain
- Multiply-Add
- Multiply-Accumulate
- MATLAB Function

Pipeline Insertion for Lookup Tables

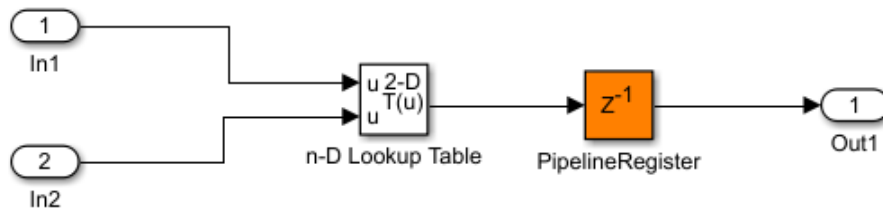
Lookup tables are blocks in the **HDL Coder > Lookup Tables** library including the Sine HDL Optimized and Cosine HDL Optimized blocks.

To insert adaptive pipelines for lookup table blocks:

- 1 Specify the target device.
- 2 Make sure that **Interpolation method** is set to Flat.

When generating code, HDL Coder inserts a register without reset at the output of the lookup table. The combination of lookup table block and register without reset can potentially map to RAM blocks on the FPGA.

This figure is the generated model for an n-D Lookup Table block with Xilinx Virtex7 as the target FPGA device.



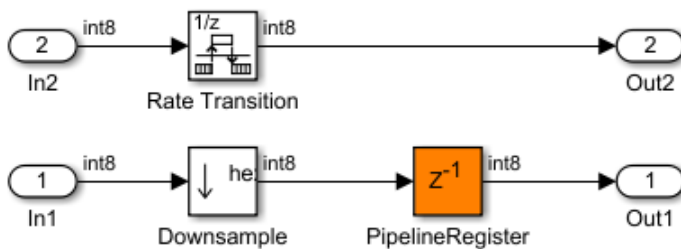
Pipeline Insertion for Rate Transition and Downsample Blocks

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.



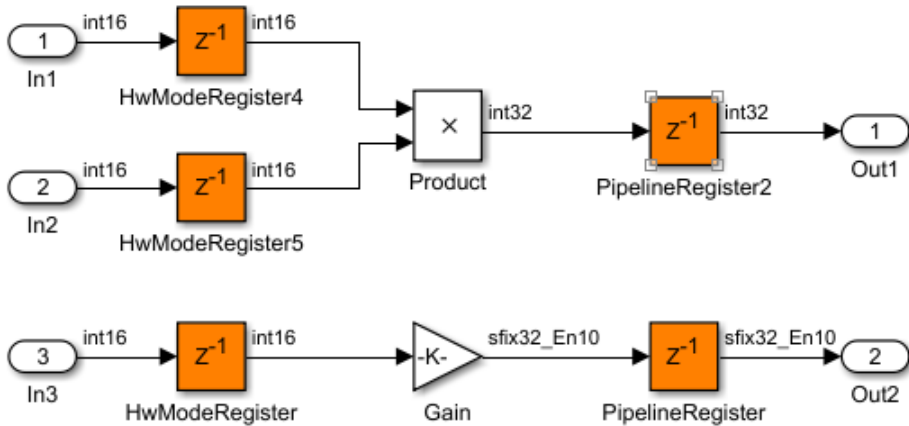
Pipeline Insertion for Product and Gain Blocks

To insert adaptive pipelines for these blocks:

- 1 Specify the target device.
- 2 Specify a target frequency greater than zero.

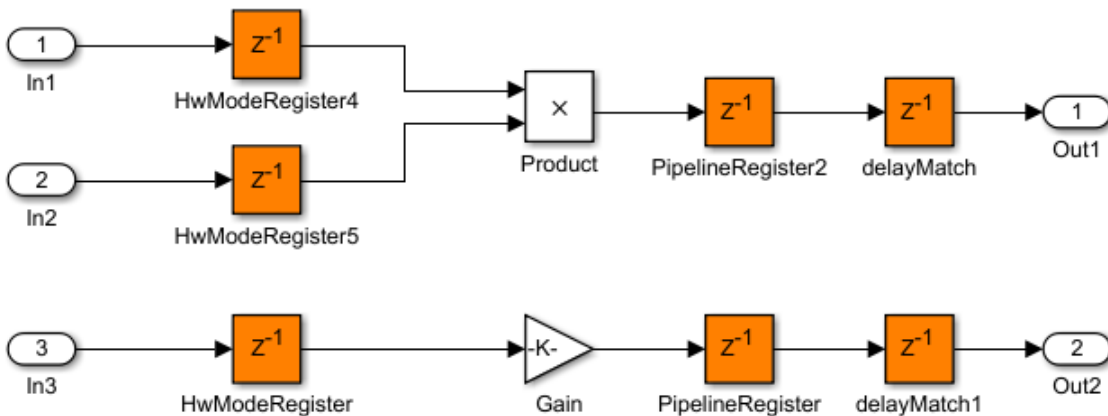
When generating code, HDL Coder inserts registers at the input and output ports of the blocks. The combination of multipliers with the registers can potentially map to DSP units on the target device.

This figure is the generated model for the Product, Gain, and Multiply-Add 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:

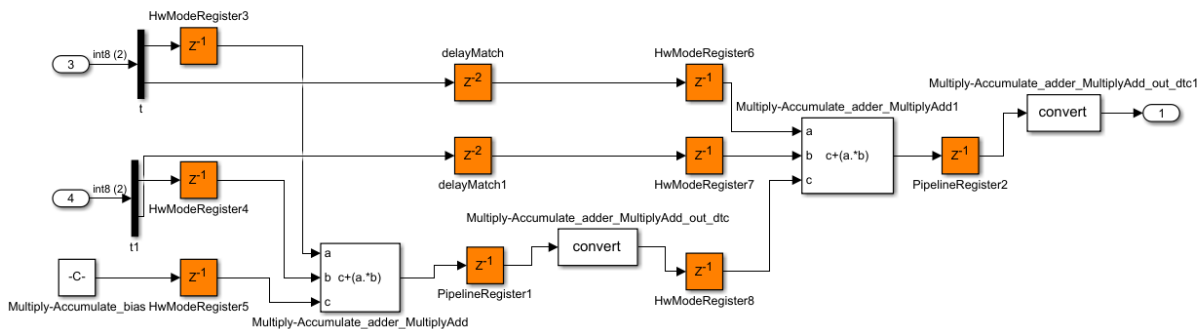
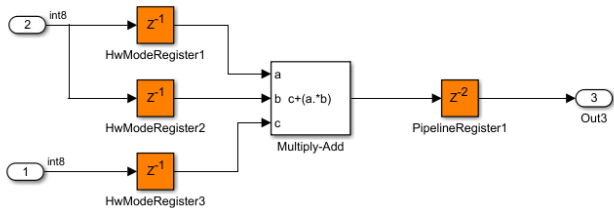
- 1 Specify the target device.
- 2 Specify a target frequency greater than zero.
- 3 Use the `Parallel` HDL architecture for the Multiply-Accumulate block. For an input vector of size N , this architecture uses N Multiply-Add blocks in series to compute the result.

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.

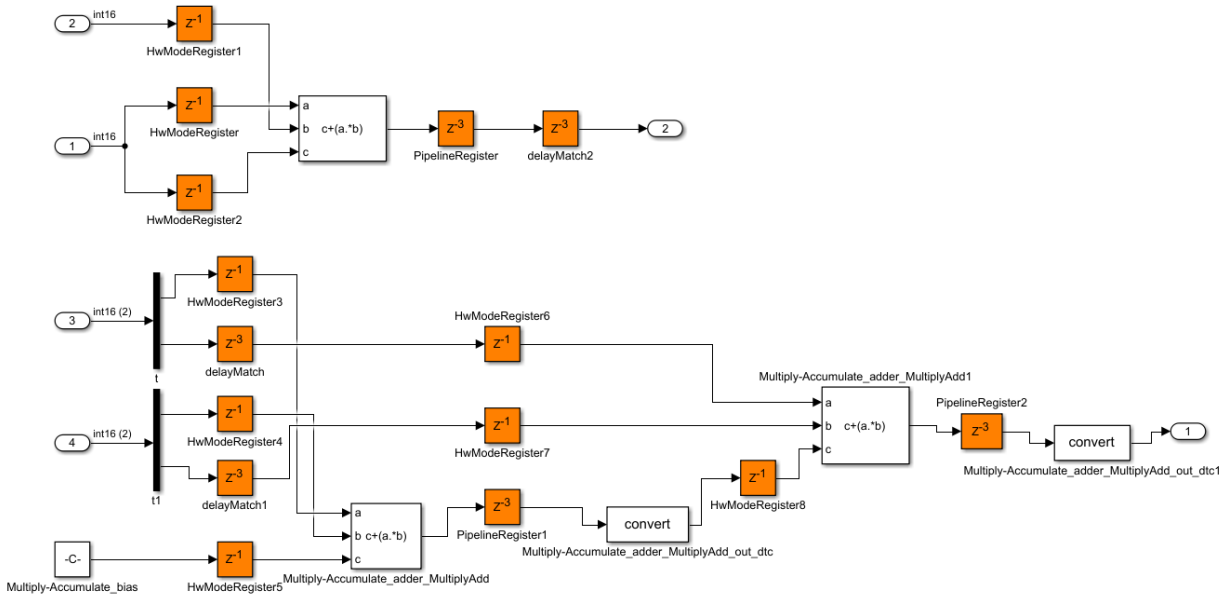
This figure is the generated model for the Multiply-Add and Multiply-Accumulate with Intel Arria10 as the target FPGA device and a target frequency of 500 MHz. The inputs to

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.

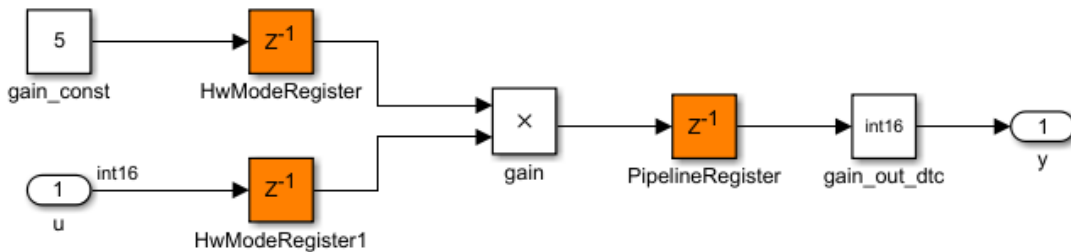
HDL Coder treats MATLAB Function blocks with architecture set to MATLAB Datapath like regular Subsystems. The code generator converts the MATLAB algorithm to a Simulink block diagram. If the Simulink diagram uses blocks that are supported by adaptive pipelining such as Product or Add, the code generator 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 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 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 29-2.

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. To enable the Code Generation report, in the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select **Generate optimization report**.

When you generate code, HDL Coder produces the Code Generation report. Select the **Adaptive Pipelining** section of the Optimization report.

The Adaptive Pipelining report displays the status of the adaptive pipelining optimization and whether HDL Coder inserted adaptive pipelines in your design.

If adaptive pipelining is successful, the report displays the blocks for which HDL Coder inserted pipeline registers, the number of pipeline registers inserted, and any additional notes. Click the link to the block to see the pipeline registers inserted to the blocks in your design.

If adaptive pipelining fails, the report displays the criteria that caused adaptive pipelining to fail.

See Also

More About

- “Balance delays” on page 14-3
- “AdaptivePipelining” on page 21-4
- “Clock-Rate Pipelining” on page 24-63

Critical Path Estimation Without Running Synthesis

In this section...

“How Critical Path Estimation Works” on page 24-79

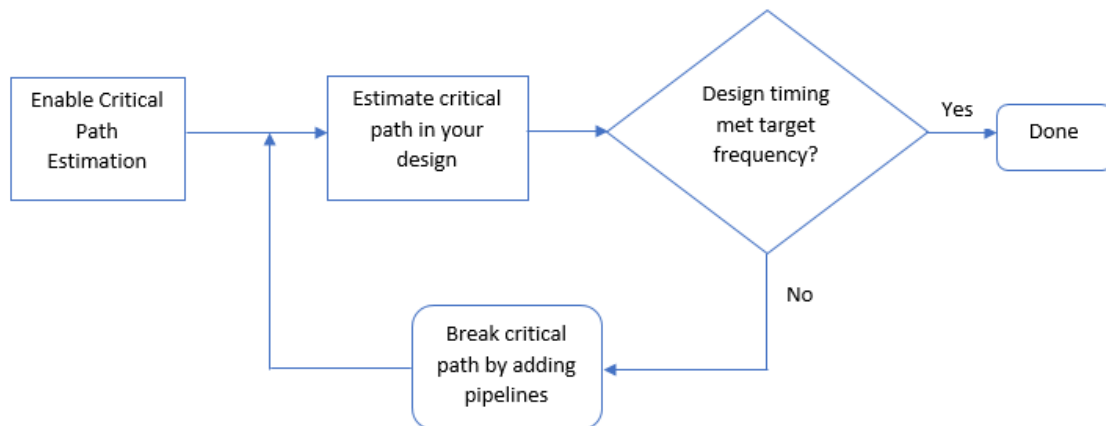
“How to Use Critical Path Estimation” on page 24-81

“Characterized Blocks” on page 24-82

“Caveats” on page 24-87

Critical path is a combinational path between an input and output that has the maximum delay. Use the HDL Coder software to find the critical path in your design. 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 do 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. With critical path estimation, you do not have to run synthesis or generate HDL code. Critical path estimation speeds up this iterative process of finding the critical path, and then optimizing the critical path until your design timing meets the target frequency that you want.



Estimating the critical path without using synthesis tools can lead to inaccurate timing results. Critical path estimation is intended to speed up the design iteration process.

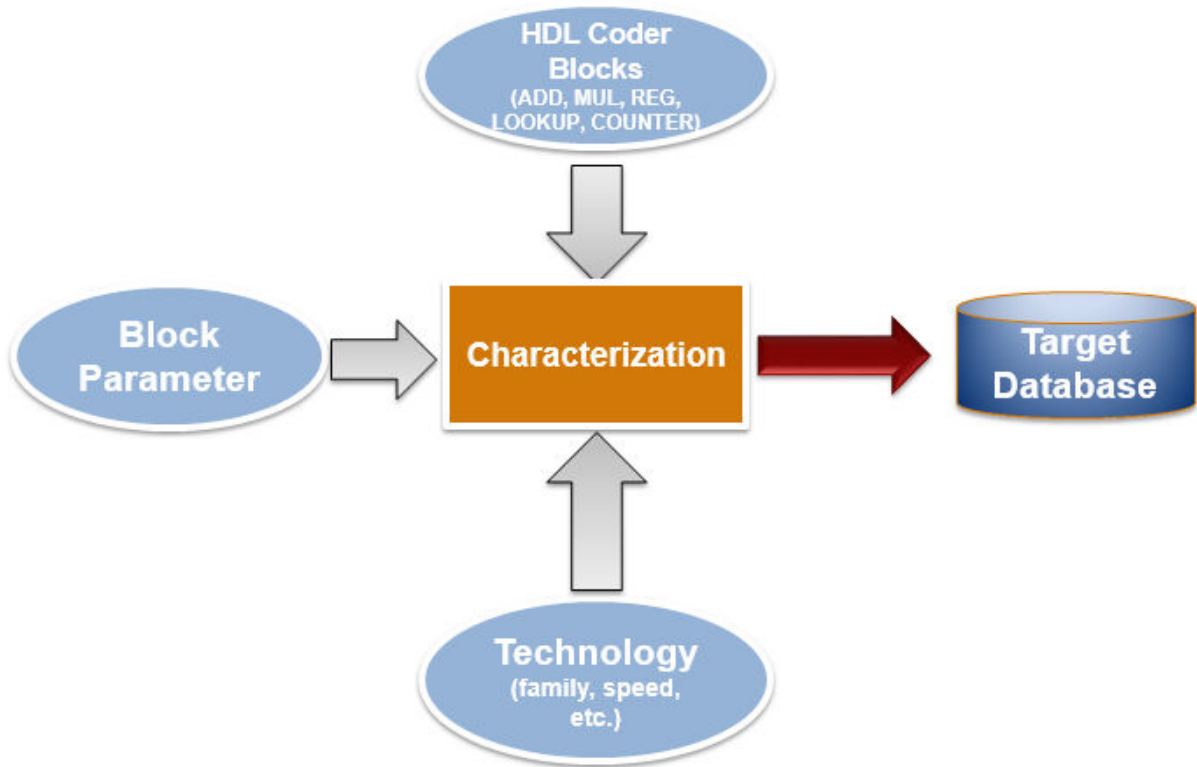
Critical path estimation is an alternative to annotating the critical path by performing **FPGA Synthesis and Analysis** with the HDL Workflow Advisor.

How Critical Path Estimation Works

HDL Coder finds the estimated critical path by performing static timing analysis with timing data from target-specific timing databases. HDL Coder has timing databases for these target devices:

- Altera Cyclone V
- Altera Stratix V
- Xilinx Virtex-7, speed grade -1
- Xilinx Zynq, 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.

How to Use Critical Path Estimation

You can estimate the critical path for your design either 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**. The **HDL Code** tab appears.
 - b 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, set **Floating Point IP Library** to Native Floating Point.
- 4 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 script this workflow or generate the report at the command line, enter these commands. Specify the `modelName` and `dutname` based on the design that you want to estimate the critical path for. This example uses the `sfir_single` model.

```
% Specify the 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');
```

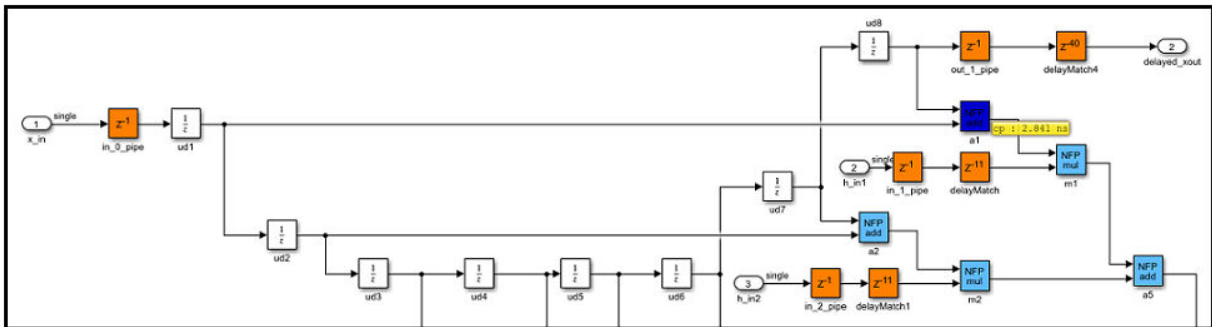
```

% If your design contains single data types,
% enable native floating-point support
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
hdlset_param(modelname, 'FloatingPointTargetConfig', fpconfig);

% Generate the 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.



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 are using 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 shows 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	Single (Native Floating Point)
Abs	✓	✓
Add	✓ (The block cannot have more than two inputs)	✓
Subtract	✓	✓
Product	✓	✓
Gain	✓	✓
Divide	✓	✓
HDL Reciprocal	✓	✓
Rounding Function	✓	✓
Unary Minus	✓	✓
Sign	✓	✓
Reshape	✓	✓
Complex to Real-Imag	✓	✓

Math Functions

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Reciprocal	✓	✓
Hypot	-	✓
Rem	-	✓
Mod	-	✓
Sqrt	✓	✓
Reciprocal Sqrt	✓	✓

Trigonometric Functions

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Sin	-	✓
Cos	-	✓
Tan	-	✓
Sincos	-	✓
Asin	-	✓
Acos	-	✓
Atan	-	✓
Atan2	-	✓
Sinh	-	✓
Cosh	-	✓
Tanh	-	✓
Atanh	-	✓

Exponent/Logarithm/Power

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Exp	-	✓
Gain to power of two	-	✓
Pow10	-	✓
Log	-	✓
Log10	-	✓

Conversions and Comparisons

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Data Type Conversion	✓	✓
Float Typecast	-	✓
Relational Operator	✓	✓
Compare To Constant	✓	✓
MinMax	✓	✓

Logic and Bit Operations

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Bit Concat	✓	-
Extract Bits	✓	-
Bit Shift	✓	-
Bit Slice	✓	-
Bitwise Operator	✓	-
Logical Operator	✓	✓

Delays and Signal Routing

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Unit Delay	✓	✓
Delay	✓	✓
Bus Creator	✓	✓
Bus Selector	✓	✓
Demux	✓	✓
Multiport Switch	✓	✓
Selector	✓	✓
Switch	✓	✓

HDL Operations and HDL RAMs

Simulink Blocks	Fixed Point	Single (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	Single (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	Single (Native Floating Point)
MATLAB Function	-	✓

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, 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 24-89.

Caveats

Critical Path Estimation with Native Floating Point

If you have single data types in your design and you use the Native Floating Point mode, 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, and 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.

In addition, critical path estimation with native floating-point does is not supported for blocks with **LatencyStrategy** set to Custom.

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. In addition, 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 only changes 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 delay by using an estimation factor. Without running place and route, it is difficult to accurately account for routing delay.
- 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.

See Also

`hdlcoder.FloatingPointTargetConfig` | `makehdl`

More About

- “Create and Use Code Generation Reports” on page 25-2
- “Generated Model and Validation Model” on page 24-5
- “Distributed Pipelining” on page 8-14
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-65

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.

This architecture is the default setting for MATLAB Function blocks with floating-point types. By enabling the MATLAB Datapath 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 and hierarchical 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

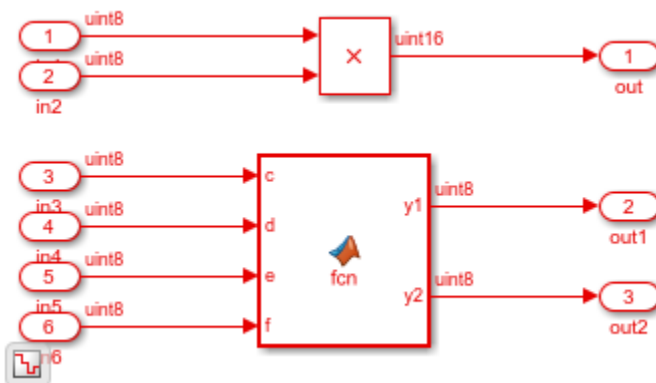
optimizations inside the MATLAB Function block, examine the generated HDL code for the block. This architecture does not apply the optimizations across the MATLAB Function block boundary with other Simulink blocks.

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')
```

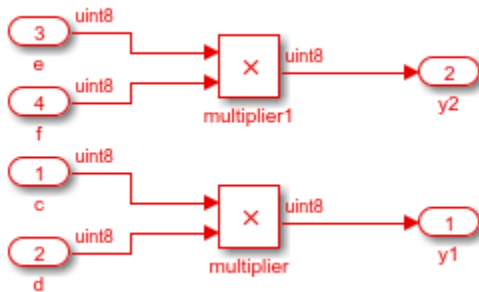
```
function [y1, y2] = fcn(c, d, e, f)
%#codegen

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.

```
m_lsubsys = 'hdlcoder_MLFB_simple_datapath/HDL_DUT/MATLAB Function';
hdlset_param(m_lsubsys, '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

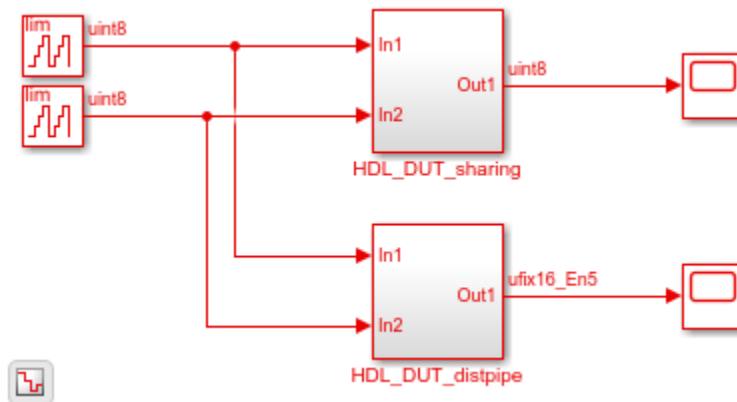
[Highlight shared resources and diagnostics](#)

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](#).

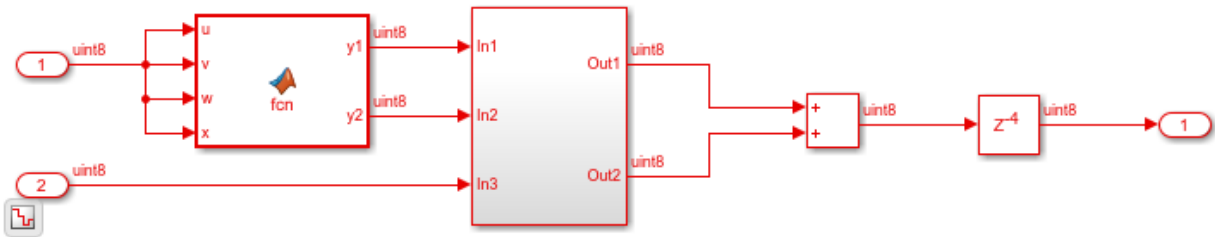
```
open_system('hdlcoder_MLFB_share_pipeline')
set_param('hdlcoder_MLFB_share_pipeline','SimulationCommand','Update')
```



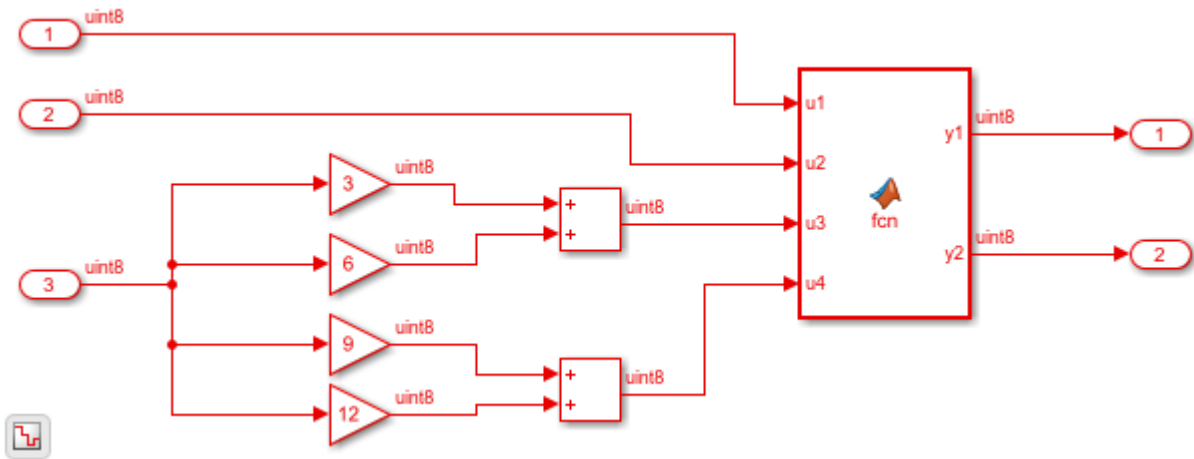
The model contains two DUT subsystems at the top level `HDL_DUT_sharing` and `HDL_DUT_distpipe`. The subsystems illustrate how you can use resource sharing and

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.

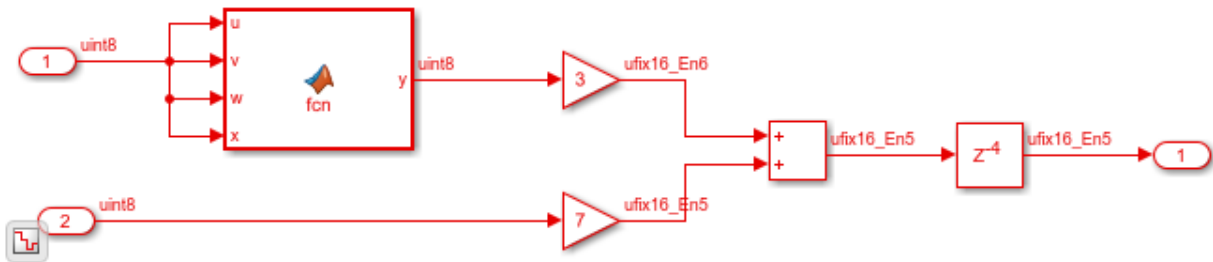
```
open_system('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing')
```



```
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_pipe');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'Oversampling', 40);
hdlset_param('hdlcoder_MLFB_share_pipeline', 'ResourceReport', 'on');
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 on the 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](#)

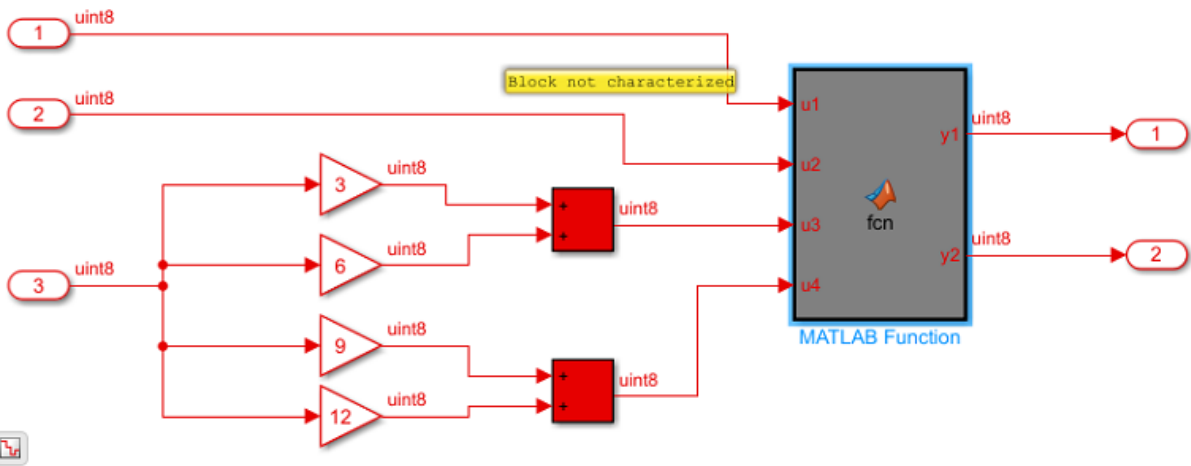
SharingFactor: 8

[Highlight shared resources and diagnostics](#)

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	9x9 -> 17	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, you see that the code generator moved pipelines inside the HDL_DUT_distpipe subsystem but did not distribute pipelines inside the MATLAB Function block.

Detailed Report

Subsystem: MATLAB Function

Implementation Parameters: DistributedPipelining: 'on'; InputPipeline: 0; OutputPipeline: 0

Status: Distributed Pipelining unsuccessful.

Subsystem: HDL_DUT_distpipe

Implementation Parameters: DistributedPipelining: 'on'; InputPipeline: 0; OutputPipeline: 0

Status: Distributed Pipelining successful.

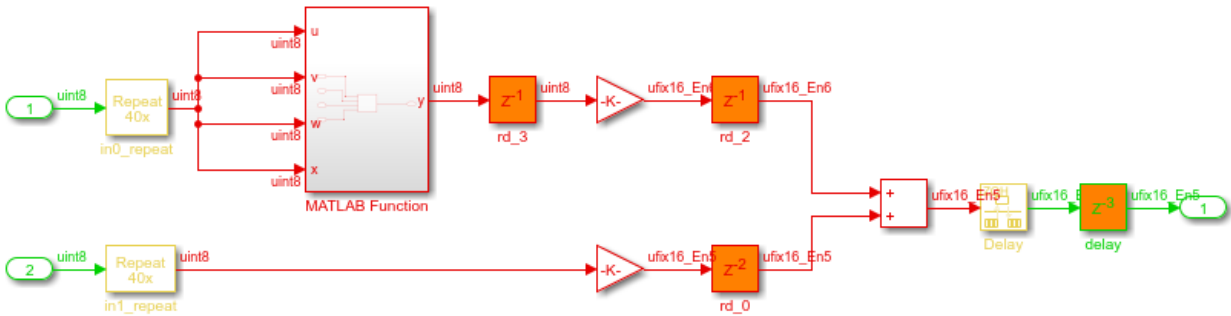
Before Distributed Pipelining : 7 registers (104 flip-flops)

Registers	Count
16-bit	6
8-bit	1

After Distributed Pipelining : 7 registers (104 flip-flops)

Registers	Count
8-bit	1
16-bit	6

This figure displays how distributed pipelining moved pipeline registers inside the subsystem.



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');
```




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](#)

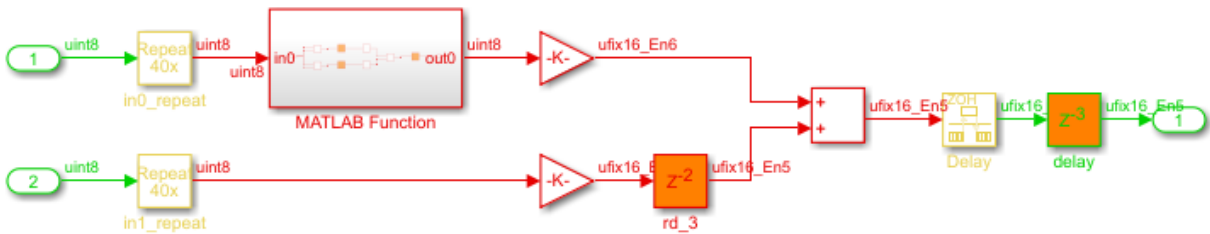
Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	9x9 -> 17	8	multiplier	
2	Sum	8+8 -> 8	4	adder	
3	Sum	8+8 -> 8	2	Add1	

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.

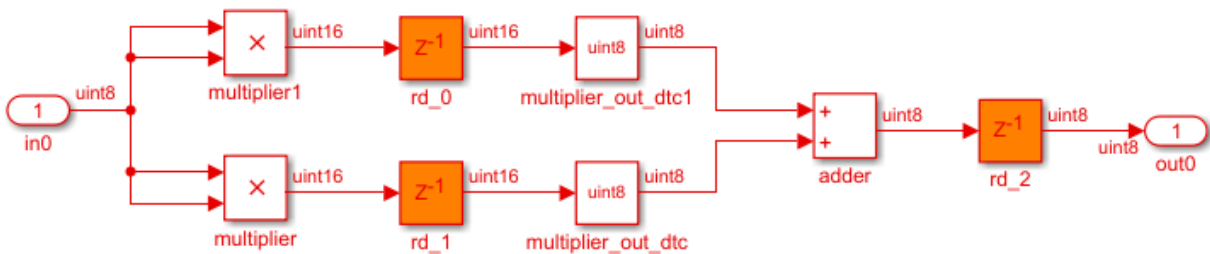
You can use the distributed pipelining optimization with the `Distributedpipe_MLFB` subsystem. Enable hierarchical distributed pipelining at the top level and set the HDL architecture of the MATLAB Function to MATLAB Datapath with **DistributedPipelining** set to on.

```
hdlset_param('hdlcoder_MLFB_share_pipeline', 'HierarchicalDistPipelining', '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');
```

After you generate HDL code, open the generated model. The code generator uses hierarchical distributed pipelining and distributed pipelining to move the pipeline registers across the blocks and inside the MATLAB Function Subsystem.



This figure displays how distributed pipelining moved pipeline registers inside the MATLAB Function Subsystem.



See Also

Blocks

MATLAB Function

Functions

hdlsaveparams | makehdl

More About

- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-39
- “RAM Mapping With the MATLAB Function Block” on page 24-44

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 24-27 and the “Area Reduction of Filter Subsystem” on page 24-103 example.

You can also use a **SharingFactor** with multichannel filters. See “Area Reduction of Multichannel Filter Subsystem” on page 24-107.

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 24-23 and the “Area Reduction of Multichannel Filter Subsystem” on page 24-107 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:

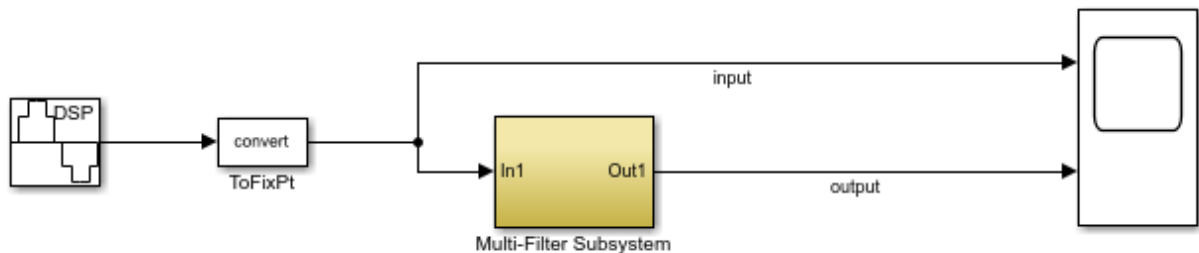
- **InputPipeline** and **OutputPipeline**
- **MultiplierInputPipeline** and **MultiplierOutputPipeline**
- **AddPipelineRegisters**

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 24-63.

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.



The model includes a sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

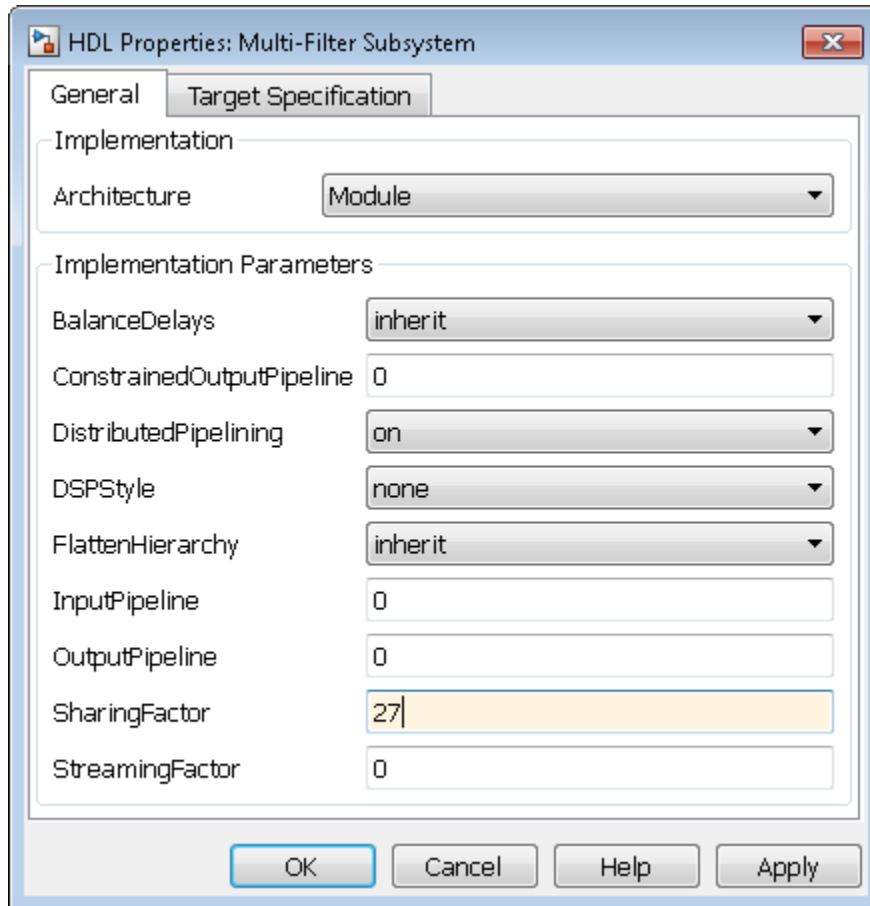


The subsystem contains a Discrete FIR Filter block and a Biquad Filter block. This design demonstrates how the optimization tools share resources between multiple filter blocks.

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

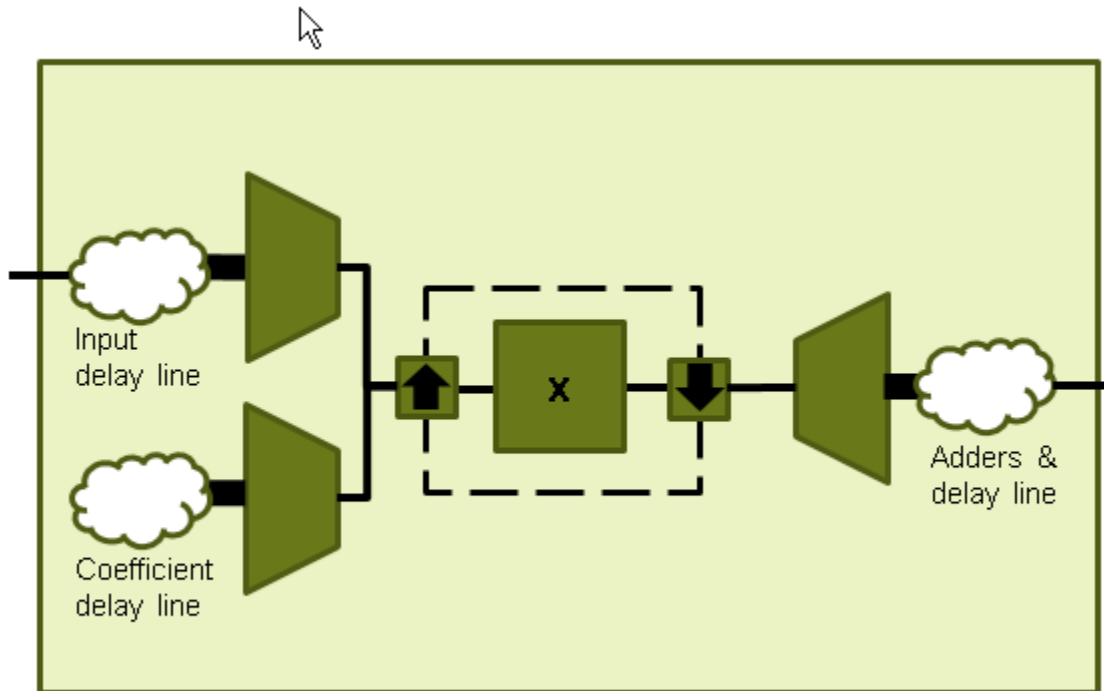
To enable streaming optimization for the **Multi-Filter Subsystem**, right-click the subsystem and select **HDL Code > HDL Block Properties**.



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.

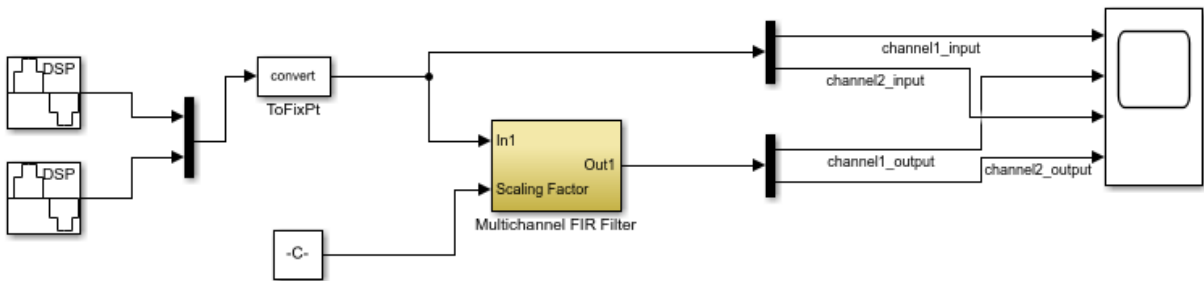


In the **Code Generation Report** window, click **High-level Resource Report**. The generated HDL code now uses one multiplier.

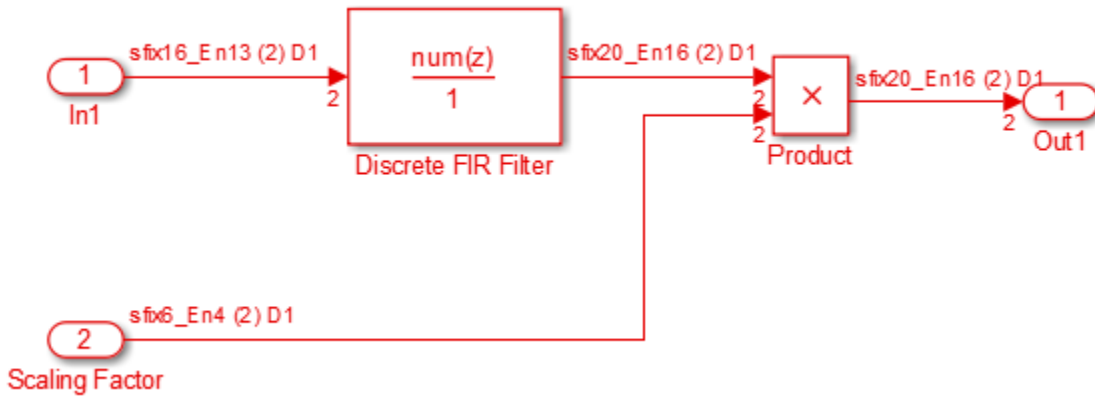
Multipliers	1
Adders/Subtractors	48
Registers	102
Total 1-Bit Registers	2457
RAMs	0
Multiplexers	7
I/O Bits	52

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.

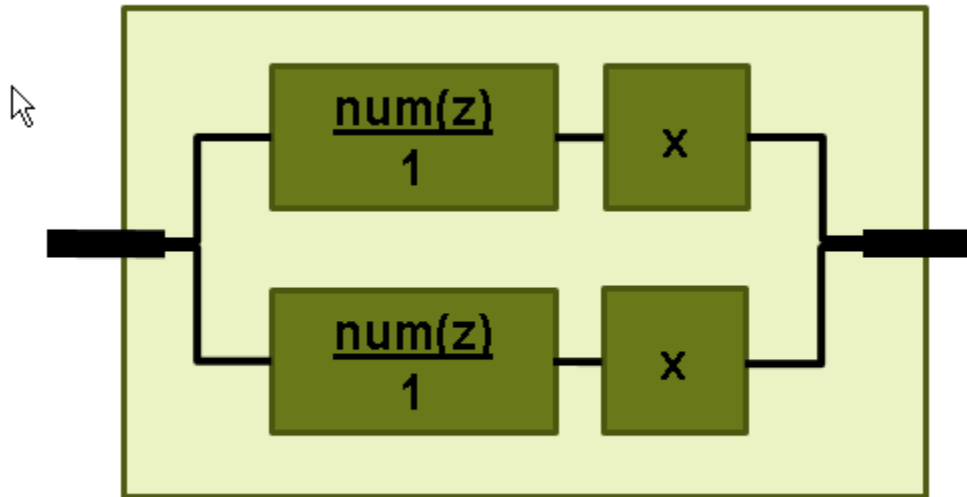


The model includes a two-channel sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

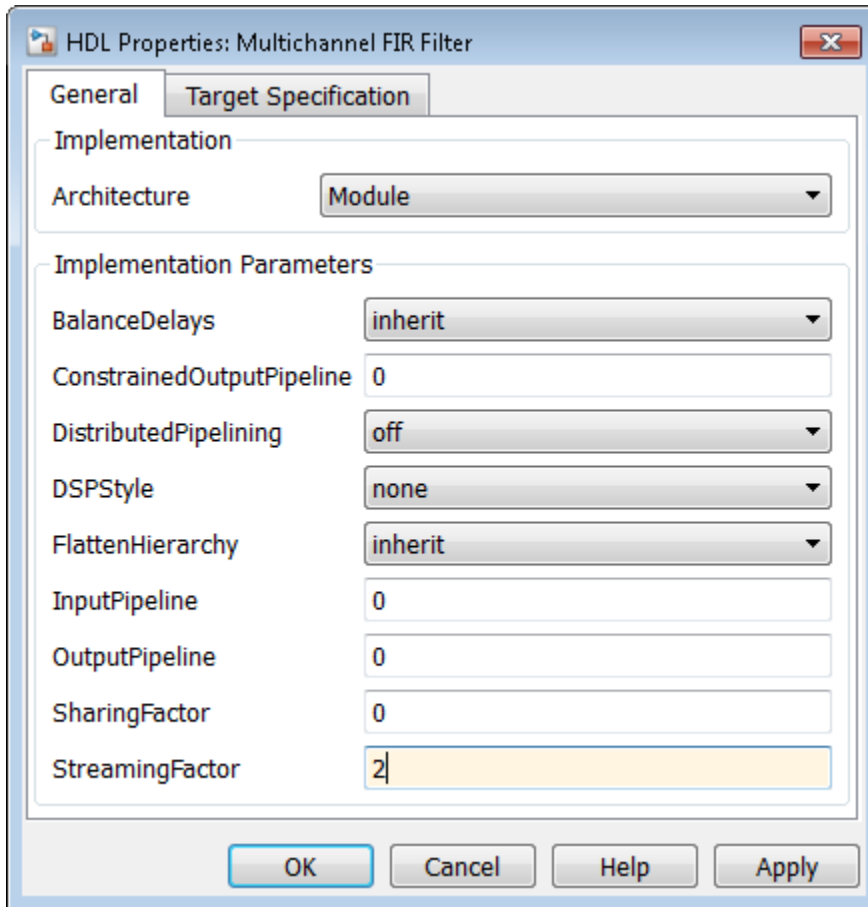


The subsystem contains a Discrete FIR Filter block and a constant multiplier. The multiplier is included to show the optimizations operating over all eligible logic in a subsystem.

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.



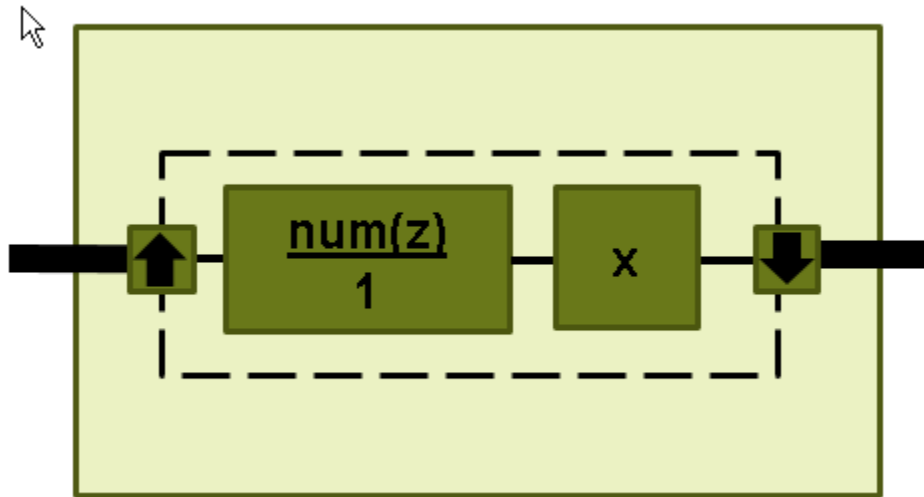
To enable streaming optimization for the Multichannel FIR Filter Subsystem, right-click the subsystem and select **HDL Code > HDL Block Properties**.



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**.

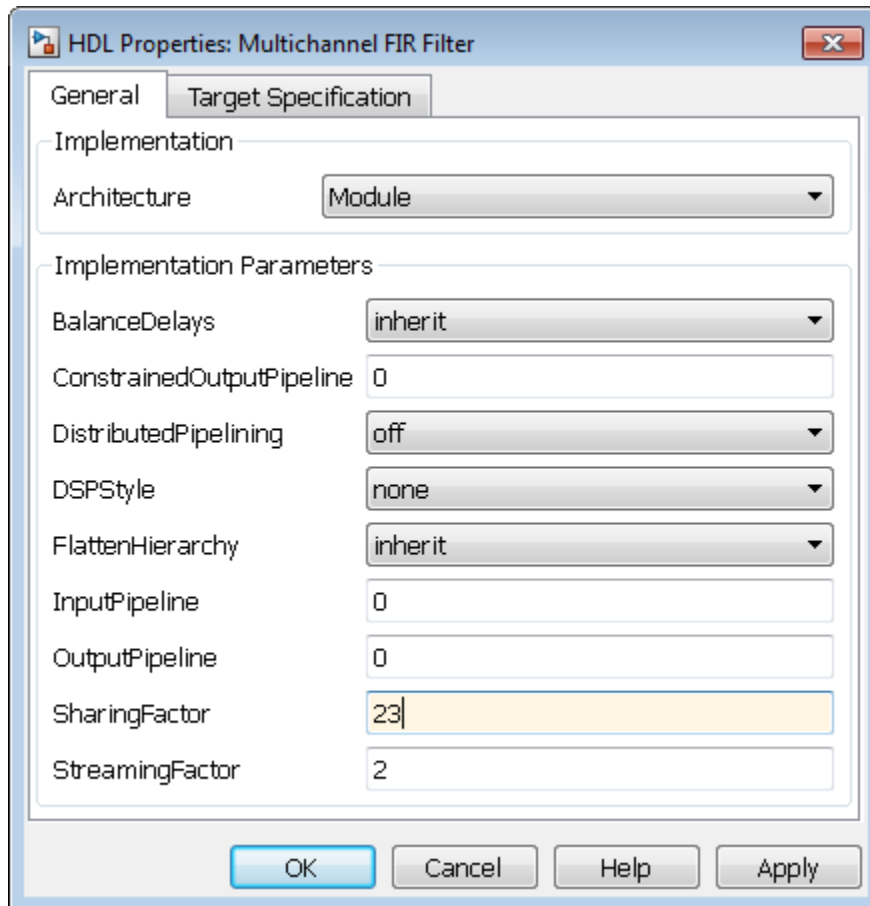
With the streaming factor applied, the logic for one channel is instantiated once and run at twice the rate of the original model.



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
```

See Also

More About

- “Resource Sharing” on page 24-27
- “Streaming” on page 24-23
- “Clock-Rate Pipelining” on page 24-63

Remove Redundant Logic and Optimize Unconnected Ports in Design

When you use floating-point data types in Native Floating Point mode, if your design contains redundant logic or unconnected ports, HDL Coder™ removes the unconnected port, or any component, or part of the HDL code that does not contribute to the output.

Remove Redundant Logic

Components that do not contribute to the output in the design are removed during HDL code generation. Examples include:

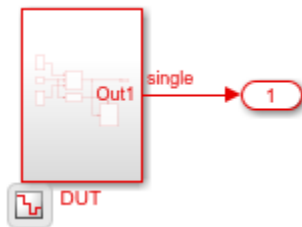
- A Switch block that receives a constant input at the control port
- A Subsystem block with no active output

HDL Coder evaluates the constant conditional values available at compile time. This simplifies the design.

To illustrate how this optimization simplifies your design:

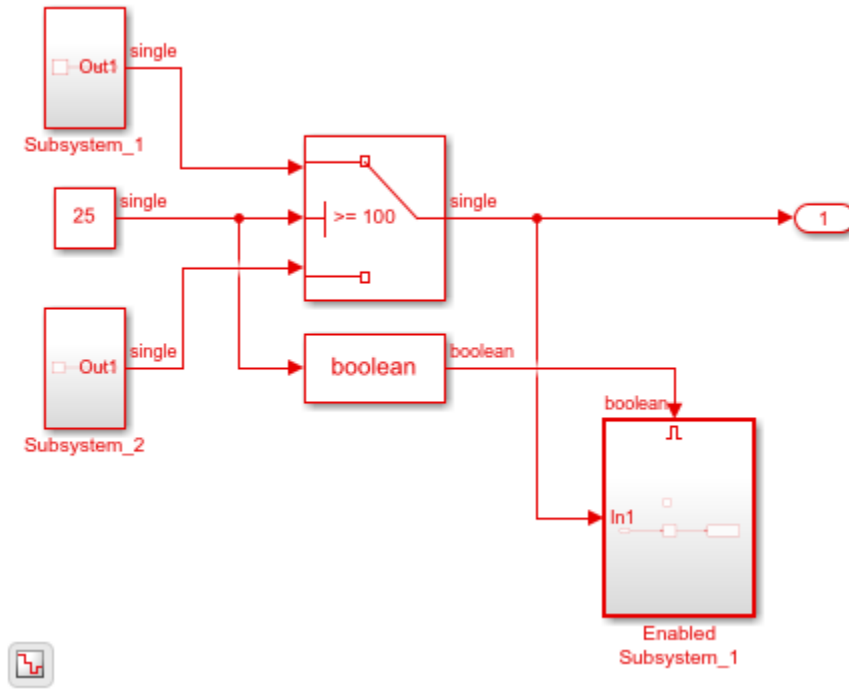
1. Open the model `hdlcoder_remove_redundant_logic` and then open the DUT block.

```
open_system('hdlcoder_remove_redundant_logic.slx')
set_param('hdlcoder_remove_redundant_logic', 'SimulationCommand', 'update');
```



2. Open the DUT Subsystem block.

```
open_system('hdlcoder_remove_redundant_logic/DUT')
```



3. To generate HDL code for the design, at the MATLAB® command prompt, enter:
`makehdl('hdlcoder_remove_redundant_logic/DUT')`
4. Open the generated model. Double-click the DUT Subsystem.



HDL Coder™ evaluated the switch condition at compile time to pass the input from Subsystem_2 to the output, and eliminated Subsystem_1 input branch. In this example, there is no active output generated from the EnabledSubsystem_1 block. The EnabledSubsystem_1 block is removed during HDL code generation.

Removing redundant logic reduces the code size and avoids potential synthesis failures with downstream tools when you deploy the generated code onto a target platform. This optimization improves the performance of your design on the target hardware.

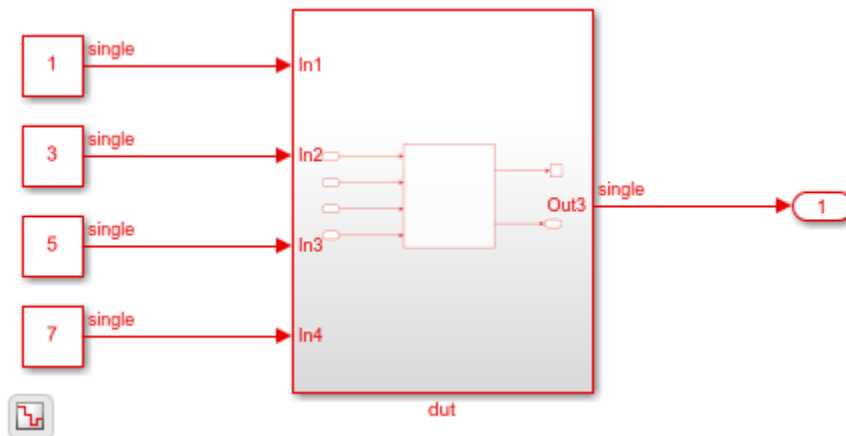
Optimize Unconnected Ports

During HDL code generation, the unconnected ports from the generated code are removed without removing ports from top-level DUT models or subsystems. This optimization includes removing unconnected vector and scalar ports, bus element ports, and bus ports. Removing unconnected ports improves the readability of the generated VHDL/Verilog code and reduces code size and area usage. The reduction avoids synthesis failure caused by unused ports in the HDL Coder™ generated VHDL/Verilog code.

To illustrate how unconnected ports are removed from a subsystem during HDL code generation:

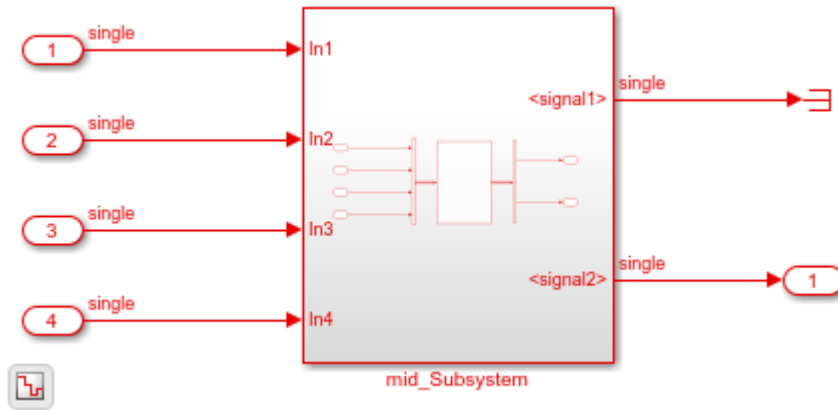
1. Open the model `hdlcoder_RemoveUnconnectedPorts` containing bus element ports and a port connected to an inactive output.

```
open_system('hdlcoder_RemoveUnconnectedPorts.slx')
set_param('hdlcoder_RemoveUnconnectedPorts', 'SimulationCommand', 'update');
```



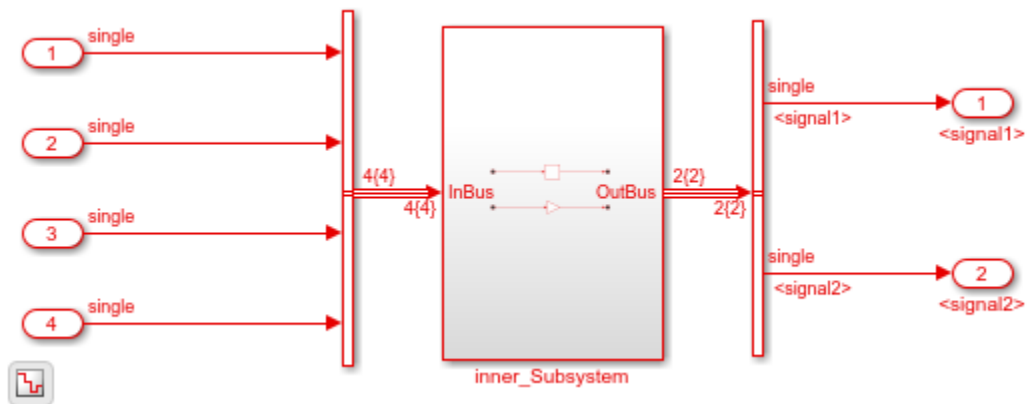
2. Open the dut Subsystem block.

```
open_system('hdlcoder_RemoveUnconnectedPorts/dut')
```

3. 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')
```



4. To generate HDL code for the design, at the MATLAB® command prompt, enter:

```
makehdl('hdlcoder_RemoveUnconnectedPorts/dut')
```

The generated code `mid_Subsystem.v` shows the code with the unconnected port optimization. Here, the unconnected ports are removed during HDL code generation.

Following is an example of code with and without the unconnected port optimization.

Code Before Optimization	Code After Optimization
<pre>21 module mid_Subsystem 22 (clk, 23 reset, 24 enb, 25 In1, 26 In2, 27 In3, 28 In4, 29 alphasignal1, 30 alphasignal2); ..</pre>	<pre>21 module mid_Subsystem 22 (clk, 23 reset, 24 enb, 25 In3, 26 alphasignal2); 27</pre>

Note: Removing redundant logic and the unconnected port optimizations do not affect traceability support.

Limitations

- Only floating-point designs are optimized. Redundant logic or unconnected ports in Fixed-point designs are not affected.
- Only unconnected data ports are removed. Control ports are not removed.
- Ports of a referenced model are not deleted.

Related Topics

- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder™” on page 24-9
- “Diagnostics for Optimizations” on page 16-110

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

- “Create and Use Code Generation Reports” on page 25-2
- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 25-5
- “Web View of Model in Code Generation Report” on page 25-13
- “Generate Code with Annotations or Comments” on page 25-16
- “Check Your Model for HDL Compatibility” on page 25-21
- “Show Blocks Supported for HDL Code Generation” on page 25-23
- “Trace Code Using the Mapping File” on page 25-27
- “Add or Remove the HDL Configuration Component” on page 25-30

Create and Use Code Generation Reports

Report Generation

The HDL Coder software creates and displays an HTML code generation report when you select one or more of the following options. You can specify the UI options in the **HDL Code Generation > Report** pane of the Configuration Parameters dialog box.

GUI option	makehdl Property
Generate traceability report	Traceability
Generate resource utilization report	ResourceReport
Generate high-level timing critical path report	CriticalPathEstimation
Generate optimization report	OptimizationReport
Generate model Web view	HDLGenerateWebview

When you generate code, the Code Generation Report appears in a separate window.

Code Generation Report

The Code Generation Report is an HTML file that includes a **Summary**, a **Clock Summary**, a **Code Interface Report**, and one or more of the following optional sections:

- Traceability report
- Resource utilization report
- High-level timing critical path report
- Optimization report
- Model web view

Summary

The **Summary** lists information about the model, the DUT, the date of code generation, and top-level coder settings. The **Summary** also lists model properties that have nondefault values.

Code Interface Report

The **Code Interface Report** shows the DUT input and output port names, data types, and bit widths. The report displays links corresponding to each input port and output port in your Simulink model.

Timing and Area Report

When you select **Generate resource utilization report**, HDL Coder adds a **Timing and Area Report** section to the Code Generation Report. This section of the report contains the following subsections:

- **High-level Resource Report:** This section The **Summary** section summarizes multipliers, adders/subtractors, and registers consumed by the device under test (DUT).

The **Detailed Report** section contains more information on the resources that each subsystem uses. Wherever possible, the detailed report links back to corresponding blocks in your model. The **Detailed Report** section also contains a **Registers** section. This section displays the total 1-bit registers that is calculated as the sum of products over the bit widths of the registers and their frequency of occurrence.

- **Target-Specific Report:** When you request target-specific code generation on the model, this subsection shows the resource utilization report.

Optimization Report

When you select **Generate optimization report**, HDL Coder adds an Optimization Report section, with three subsections:

- **Distributed Pipelining:** If a subsystem has the `DistributedPipelining` option enabled, this subsection displays comparative listings of registers before and after you apply the distributed pipelining transform.
- **Streaming and Sharing:** Summary and detailed information about the subsystems for which you specify sharing or streaming optimizations and the delay balancing summary.
- **Target Code Generation:** Summary, status, and path delay information about the subsystems after target code generation.
- **Delay Balancing:** Lists the number of pipeline delays and phase delays added at the output ports to match the delays.

See Also

More About

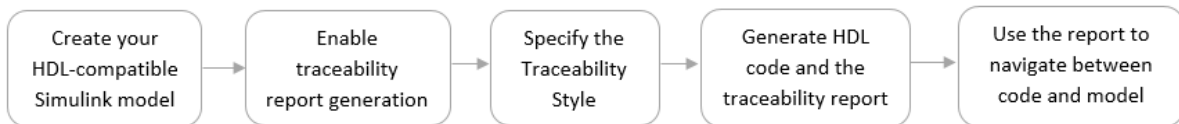
- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 25-5
- “Critical Path Estimation Without Running Synthesis” on page 24-78
- “Web View of Model in Code Generation Report” on page 25-13

Navigate Between Simulink Model and HDL Code by Using Traceability

In this section...

“How Traceability Works” on page 25-5
“Generate Traceability Report” on page 25-6
“Report Location” on page 25-7
“View the Traceability Report” on page 25-7
“Code-to-Model Navigation” on page 25-9
“Model-to-Code Navigation” on page 25-10
“Traceability Report Limitations” on page 25-11

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” on page 17-5.

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” on page 17-3.

- 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 **TraceabilityStyle**.
 - At the command line, use `hdlset_param` to specify the `TraceabilityStyle` property on the model.

To learn more about this parameter, see “Traceability style” on page 17-5.

- 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 “Folder” on page 12-5.

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	Output
<S2>/delayed_xout	Output

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

Subsystem: [sfil_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” on page 16-57.

Code-to-Model Navigation

To navigate from the HDL code to the model:

- 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**.

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

Subsystem: [sfr_fixed/symmetric_fir](#)

Object Name	Code Location
<S2>/a1	symmetric_fir.vhd:208, 209, 210
<S2>/a2	symmetric_fir.vhd:212, 213, 214
<S2>/a3	symmetric_fir.vhd:216, 217, 218



```

206
207
208 a1_add_cast <= resize(ud8_out1, 17);
209 a1_add_cast_1 <= resize(ud1_out1, 17);
210 a1_out1 <= a1_add_cast + a1_add_cast_1;
211
212 a2_add_cast <= resize(ud7_out1, 17);
213 a2_add_cast_1 <= resize(ud2_out1, 17);
214 a2_out1 <= a2_add_cast + a2_add_cast_1;
215
    
```



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**.

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

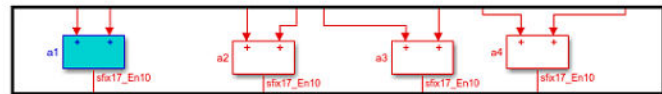
Subsystem: [sfir_fixed/symmetric_fir](#)

Object Name	Code Location
<S2>/a1	symmetric_fir.vhd:216
<S2>/a2	symmetric_fir.vhd:221
<S2>/a3	symmetric_fir.vhd:226



```

214
215
216 -- <S2>/a1
217 a1_add_cast <= resize(ud8_out1, 17);
218 a1_add_cast_1 <= resize(ud1_out1, 17);
219 a1_out1 <= a1_add_cast + a1_add_cast_1;
220
221 -- <S2>/a2
222 a2_add_cast <= resize(ud7_out1, 17);
223 a2_add_cast_1 <= resize(ud2_out1, 17);
224 a2_out1 <= a2_add_cast + a2_add_cast_1;
225
    
```



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

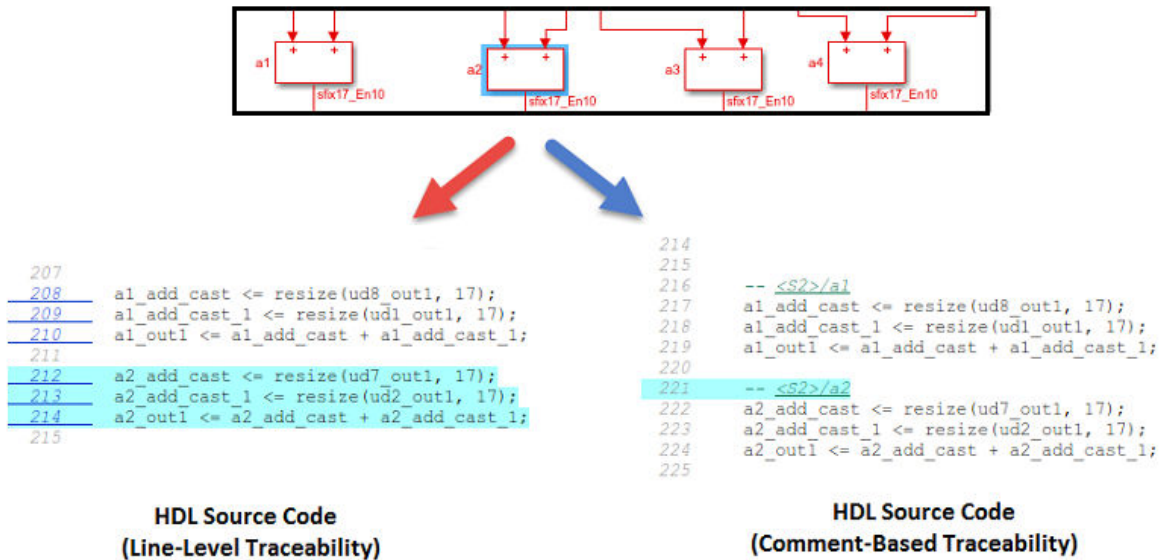
Use model-to-code traceability to select a component at any level of the model and view the code references to that component in the traceability report. For tracing, you can select these objects:

- 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 either of these approaches.

- 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 25-2

Web View of Model in Code Generation Report

In this section...

“About Model Web View” on page 25-13

“Generate HTML Code Generation Report with Model Web View” on page 25-13

“Model Web View Limitations” on page 25-15

About Model Web View

To review and analyze the generated code, it is helpful to navigate between the code and model. 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.

A Simulink Report Generator license is required to include a Web view (Simulink Report Generator) of the model in the code generation report.

Browser Requirements for Web View

Web view requires a Web browser that supports Scalable Vector Graphics (SVG). Web view uses SVG to render and navigate models.

You can use the following Web browsers:

- Mozilla Firefox Version 1.5 or later, which has native support for SVG. To download the Firefox browser, go to www.mozilla.com/.
- The Microsoft® Internet Explorer® Web browser with the Adobe® SVG Viewer plug-in. To download the Adobe SVG Viewer plug-in, go to www.adobe.com/svg/.
- Apple Safari Web browser

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 Open the mcombo model.
- 2 Open the **Configuration Parameters** dialog box or **Model Explorer** and navigate to the **HDL Code Generation** pane.
- 3 Under **Code generation report**, select **Generate model Web view**.
- 4 Click the **Generate** button.

After building the model and generating code, the code generation report opens in a MATLAB Web browser.

- 5 In the left navigation pane, select a source code file. The corresponding source code is displayed in the right pane and includes hyperlinks.

combo	
Parameter Attributes	
ShowPort...	FromPortIcon
Permissions	ReadWrite
TreatAsAto...	off
MinAlgLoop...	off
SystemSa...	-1
RTWSystem...	Auto
TreatAsGro...	on
Other	

- 6 Click a link in the code. The model Web view displays and highlights the corresponding block in the model.
- 7 To highlight the generated code for a block in your model, click the block. The corresponding code is highlighted in the source code pane.
- 8 To go back to the code generation report for the top model, at the top of the left navigation pane, click the **Back** button until the top model's report is displayed.

For more information about exploring a model in a Web view, see “Navigate the Web View” (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 referenced model block in the top model. Instead, open the code generation report for the referenced model by clicking a link under **Referenced Models** in the left navigation pane.
- 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, see “Open Code Generation Report” (Simulink Coder).
- 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.

Generate Code with Annotations or Comments

In this section...
“Simulink Annotations” on page 25-16
“Signal Descriptions” on page 25-16
“Text Comments” on page 25-17
“Requirements Comments and Hyperlinks” on page 25-17

The following sections describe how to use the HDL Coder software to add text annotations to generated code, in the form of model annotations, text comments or requirements comments.

Simulink Annotations

You can enter text directly on the block diagram as Simulink annotations. HDL Coder renders text from Simulink annotations as plain text comments in generated code. The comments are generated at the same level in the model hierarchy as the subsystem(s) that contain the annotations, as if they were Simulink blocks.

See “Describe Models Using Annotations” (Simulink) for general information on annotations.

Signal Descriptions

You can 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. Then, select the **Documentation** tab, and in the **Description** section, enter a description for the signal. For the signal description, use ASCII characters because non-ASCII characters in the generated code can potentially interfere with downstream synthesis and lint tools. In some cases, due to certain 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

You can enter text comments at any level of the model by placing a DocBlock at the desired level and entering text comments. HDL Coder renders text from the DocBlock in 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.

Set the **Document type** parameter of the DocBlock to `Text`. HDL Coder does not support the HTML or RTF options.

See DocBlock for general information on the DocBlock.

Requirements Comments and Hyperlinks

You can assign requirement comments to blocks.

If your model includes requirements comments, you can choose to render the comments in one of the following formats:

- *Text comments in generated code:* To include requirements as text comments in code, use the defaults for **Include requirements in block comments** (on) and **Generate traceability report** (off) in the Configuration Parameters dialog box.

If you generate code from the command line, set the `Traceability` and `RequirementComments` properties:

```
makehdl(gcb, 'Traceability', 'off', 'RequirementComments', 'on');
```

The following figure highlights text requirements comments generated for a Gain block from the mcombo model.

```

36 BEGIN
37     In1_signed <= signed(In1);
38
39     --
40     -- Block requirements for <S10>/Gain
41     -- 1. Gain Requirements Sect 1
42     -- 2. Gain Requirements Sect 2
43     Gain_gainparam <= to_signed(16384, 16);
44
45     Gain_out1 <= resize(In1_signed(15 DOWNT0 0) & '0'
46
47
48     Out1 <= std_logic_vector(Gain_out1);
49
50 END rtl;

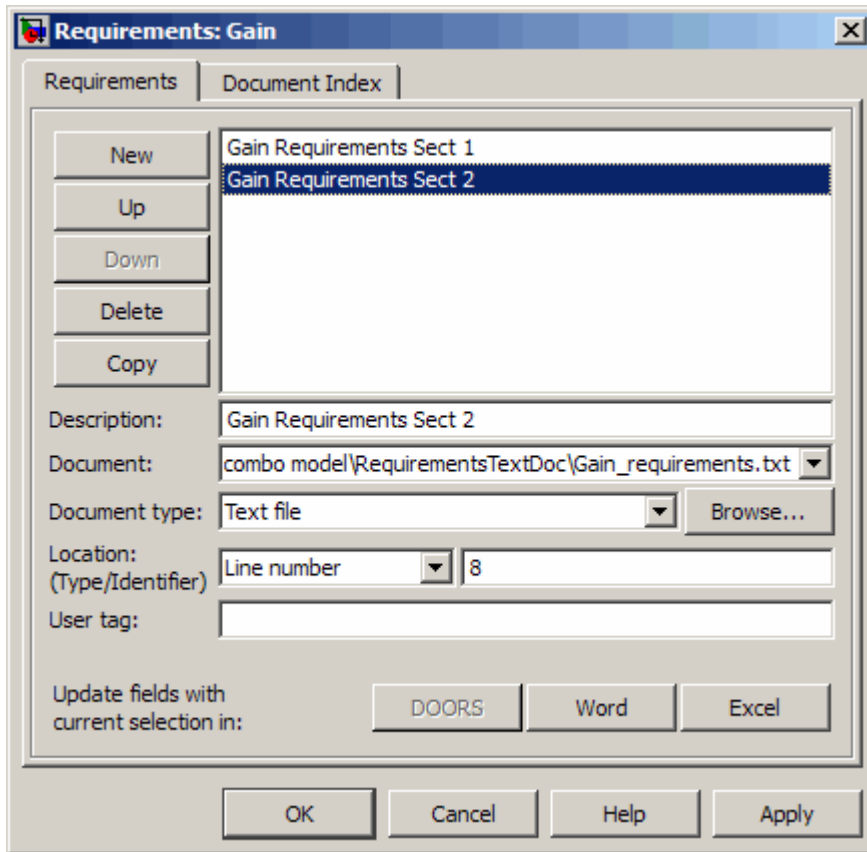
```

- *Hyperlinked comments:* To include requirements comments as hyperlinked comments in an HTML code generation report, select both **Generate traceability report** and **Include requirements in block comments** in the Configuration Parameters dialog box.

If you generate code from the command line, set the `Traceability` and `RequirementComments` properties:

```
makehdl(gcb, 'Traceability', 'on', 'RequirementComments', 'on');
```

The comments include links back to a requirements document associated with the block and to the block within the original model. For example, the following figure shows two requirements links assigned to a Gain block. The links point to sections of a text requirements file.



The following figure shows hyperlinked requirements comments generated for the Gain block.

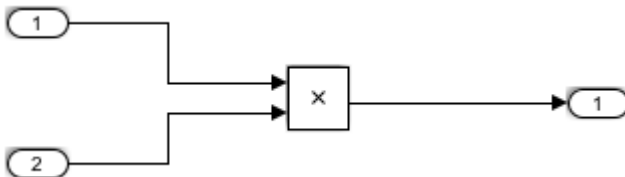
```
36 BEGIN
37     In1_signed <= signed(In1);
38
39     -- <S10>/Gain
40     --
41     --
42     -- Block requirements for <S10>/Gain
43     -- 1. Gain Requirements Sect 1
44     -- 2. Gain Requirements Sect 2
45     Gain_gainparam <= to_signed(16384, 16);
46
47     Gain_out1 <= resize(In1_signed(15 DOWNTO 0) &
48
49
50     Out1 <= std_logic_vector(Gain_out1);
51
52 END rtl;
```

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')

### Starting HDL check.
### Creating HDL Code Generation Check Report file://C:\HDL_Checks\hdlsrc\hdlcoder_pro
### HDL check for 'hdlcoder_product_mixed_types' complete with 1 errors, 1 warnings, an
```

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.

Show Blocks Supported for HDL Code Generation

The `hdlLib` function displays the blocks that are compatible with for HDL code generation in the Library Browser. It only displays those blocks for which you have a license. If you construct models using blocks from this Library Browser view, your models are compatible with HDL code generation.

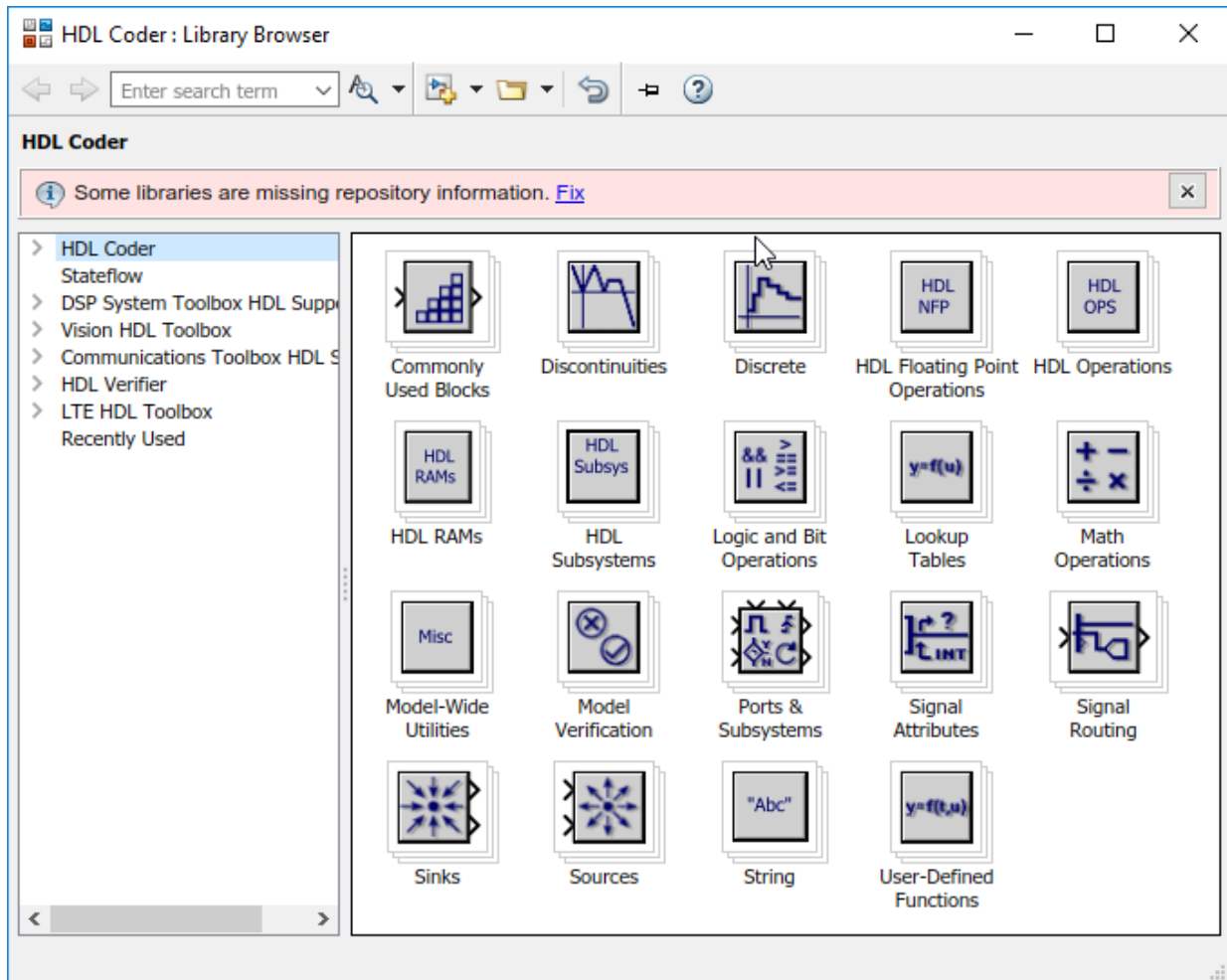
Parameter settings for blocks in this Library Browser view are compatible with HDL code generation, and therefore can differ from the default settings.

Show Supported Blocks in Library Browser

To display the blocks that are compatible with HDL code generation:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select **HDL Block Properties > Open HDL Block Library**.
- Alternatively, at the command prompt, enter:

```
hdlLib
```



The Library Browser opens. You can drag and drop the blocks from the Library Browser into your model.

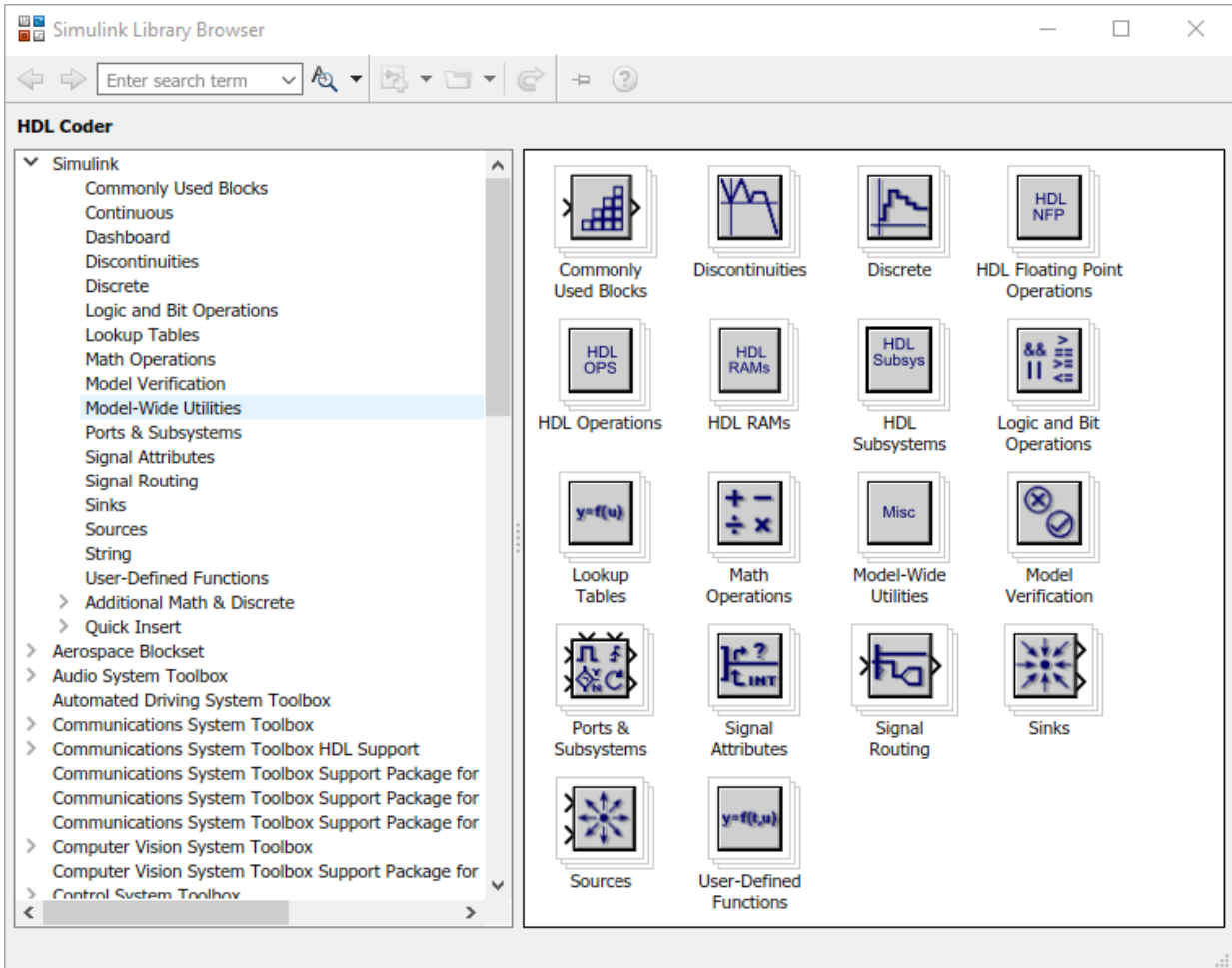
If you close and reopen the Library Browser in the same MATLAB session, you continue to see only the blocks that are compatible with HDL code generation. To reset the Library


Browser view to show all blocks, click the  button.

Reset Library Browser to Show All Blocks

To reset the Library Browser view so that it shows all blocks, regardless of HDL code generation compatibility, at the command prompt, enter:

```
hdlLib('off')
```



To change the Library Browser view to show only those blocks that are compatible with HDL code generation, click the  button.

Generate a Supported Blocks Report

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 the following 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

`hdllib`

More About

- “Create Simulink Model for HDL Code Generation”
- “View HDL-Specific Block Documentation” on page 21-2

Trace Code Using the Mapping File

Note This section refers to generated VHDL entities or Verilog 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 or Verilog 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 or Verilog module that was generated from that system. The mapping file contains one entry per line.

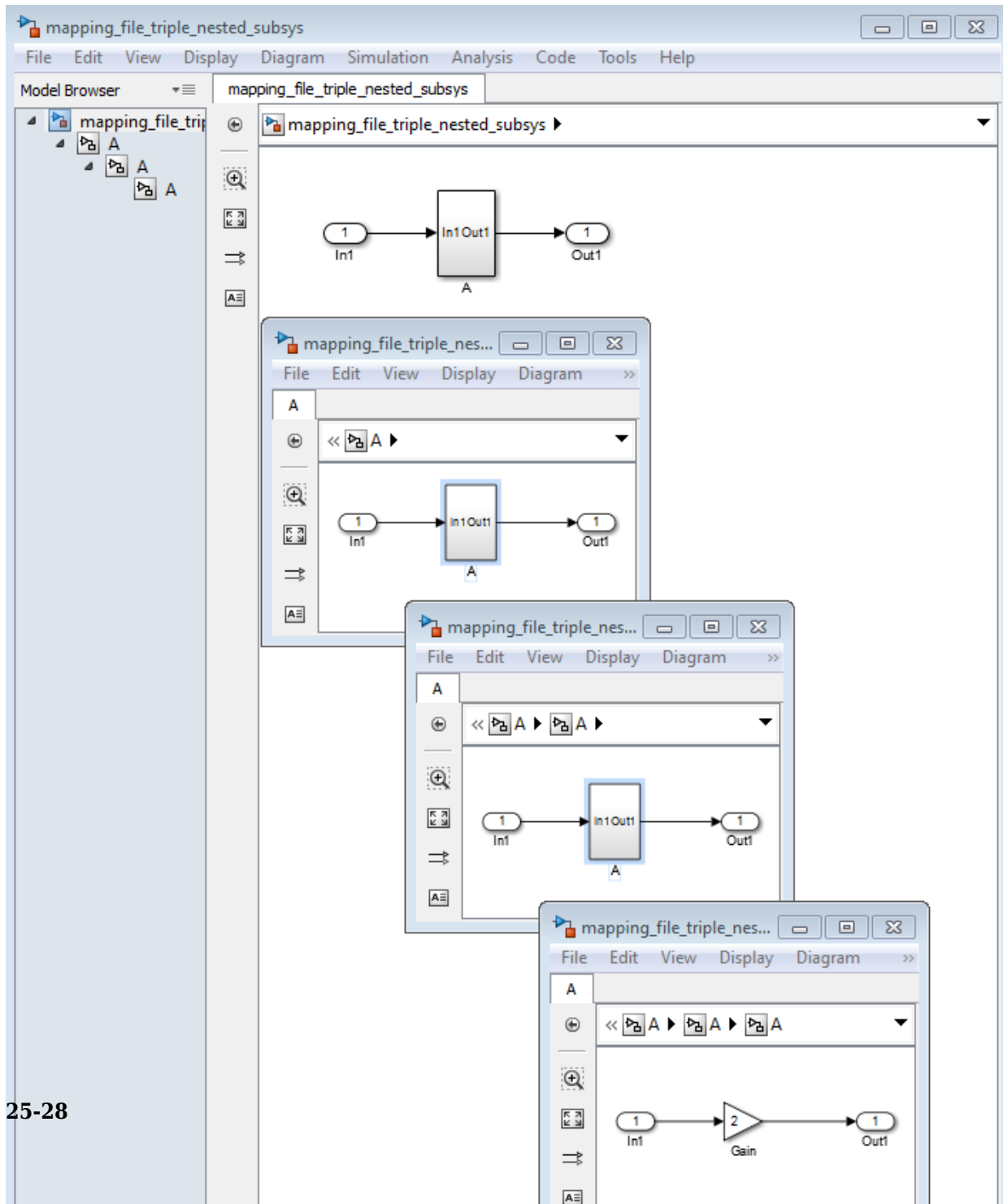
In simple cases, the mapping file may contain only one entry. For example, the `symmetric_fir` subsystem of the `sfir_fixed` model generates the following mapping file:

```
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.



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...

“What Is the HDL Configuration Component?” on page 25-30

“Adding the HDL Coder Configuration Component To a Model” on page 25-30

“Removing the HDL Coder Configuration Component From a Model” on page 25-31

What Is the 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 the **HDL Code Generation** pane in the Configurations Parameters dialog box and set HDL code generation options. Normally, you do not need to interact with the HDL configuration component. However, there are situations where you might want to add or remove the HDL configuration component:

- A model that was created on a system that did not have HDL Coder installed does not have the HDL configuration component attached. In this case, you might want to add the HDL configuration component to the model.
- If a previous user removed the HDL configuration component, you might want to add the component back to the model.
- If a model will be running on some systems that have HDL Coder installed, and on other systems that do not, you might want to keep the model consistent between both environments. If so, you might want to remove the HDL configuration component from the model.

Adding the HDL Coder Configuration Component To a Model

To add the HDL Coder configuration component to a model, In the Simulink Toolstrip, on the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. On the **HDL Code** tab, select **Settings > Add HDL Coder Configuration to Model**.

Removing the HDL Coder Configuration Component From a Model

To remove the HDL Coder configuration component from a model, on the **HDL Code** tab, select **Settings > Remove HDL Coder Configuration from Model**.

HDL Coding Standards

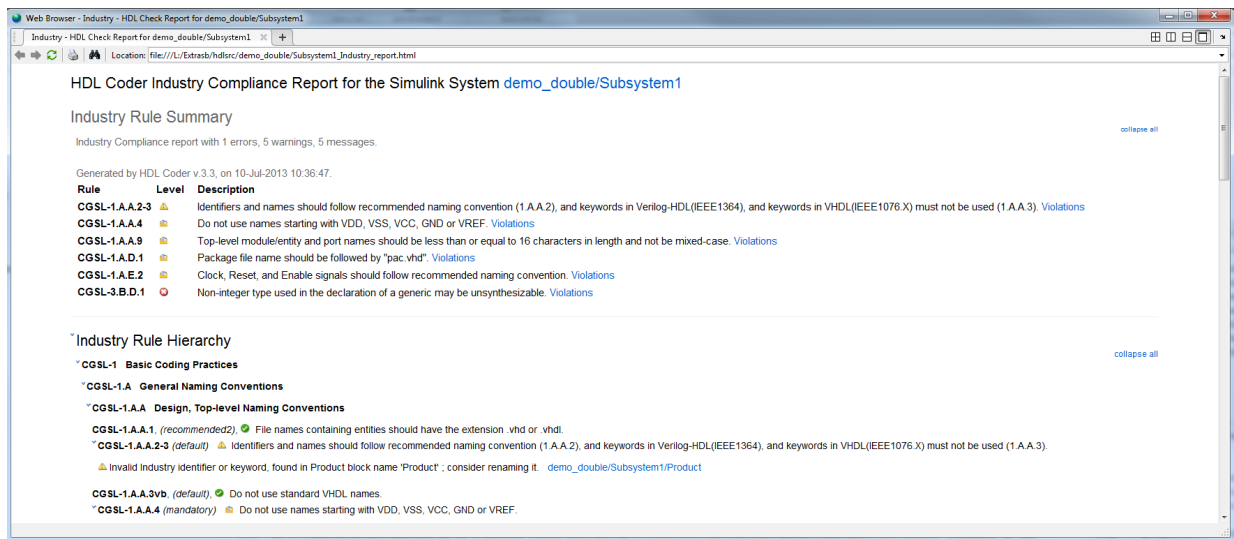
- “HDL Coding Standard Report” on page 26-2
- “HDL Coding Standards” on page 26-4
- “Generate an HDL Coding Standard Report from Simulink” on page 26-6
- “Basic Coding Practices” on page 26-10
- “RTL Description Techniques” on page 26-25
- “RTL Design Methodology Guidelines” on page 26-64
- “Generate an HDL Lint Tool Script” on page 26-71

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 26-4.

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 26-10
- “RTL Description Techniques” on page 26-25
- “RTL Design Methodology Guidelines” on page 26-64

See Also

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-49
- “Generate an HDL Coding Standard Report from Simulink” on page 26-6

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 26-2.

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 26-10.
- Section 2: “RTL Description Techniques” on page 26-25.
- Section 3: “RTL Design Methodology Guidelines” on page 26-64.

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 5-49
- “Generate an HDL Coding Standard Report from Simulink” on page 26-6

Generate an HDL Coding Standard Report from Simulink

In this section...
“Using the HDL Workflow Advisor” on page 26-6
“Using the Command Line” on page 26-8

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 with the HDL Workflow Advisor:

- 1 In the **HDL Code Generation** task, in **Set Code Generation Options > Set Advanced Options**, select the **Coding standards** tab.
- 2 For **HDL coding standard**, select **Industry** and click **Apply**.

Additional settings

General	Ports	Optimization	Coding style	Coding standards	Diagnostics	Floating Point Target
Choose coding standard						
HDL coding standard: <input type="text" value="Industry"/>						
Report options						
<input type="checkbox"/> Do not show passing rules in coding standard report						
Basic coding rules						
<input checked="" type="checkbox"/> Check for duplicate names						
<input checked="" type="checkbox"/> Check for HDL keywords in design names						
<input checked="" type="checkbox"/> Check module, instance, entity name length						
Minimum <input type="text" value="2"/>						
Maximum <input type="text" value="32"/>						
<input checked="" type="checkbox"/> Check signal, port, parameter name length						
Minimum <input type="text" value="2"/>						
Maximum <input type="text" value="40"/>						
RTL description rules						
<input type="checkbox"/> Check for clock enable signals						
<input type="checkbox"/> Detect usage of reset signals						
<input type="checkbox"/> Detect usage of asynchronous reset signals						
<input type="checkbox"/> Minimize use of variables						
<input checked="" type="checkbox"/> Check for initial statements that set RAM initial values						
<input checked="" type="checkbox"/> Check for conditional statements in processes						
Length <input type="text" value="1"/>						
<input checked="" type="checkbox"/> Check if-else statement chain length						
Length <input type="text" value="7"/>						
<input checked="" type="checkbox"/> Check if-else statement nesting depth						
Depth <input type="text" value="3"/>						
<input checked="" type="checkbox"/> Check multiplier width						
Maximum <input type="text" value="16"/>						

- 3 Optionally, using the other options in the **Coding standards** tab, customize the coding standard rules and click **Apply**.

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

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-49
- “Generate an HDL Coding Standard Report from Simulink” on page 26-6

More About

- “HDL Coding Standard Report” on page 26-2
- “Basic Coding Practices” on page 26-10
- “RTL Description Techniques” on page 26-25
- “RTL Design Methodology Guidelines” on page 26-64

Basic Coding Practices

In this section...
"1.A General Naming Conventions" on page 26-11
"1.B General Guidelines for Clocks and Resets" on page 26-20
"1.C Guidelines for Initial Reset" on page 26-21
"1.D Guidelines for Clocks" on page 26-22
"1.F Guidelines for Hierarchical Design" on page 26-24

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: 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 <code>VHDLFileExtension</code> property from the command line.
1.A.A.2 Message	Verilog/VHDL: 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 (_).

Rule / Severity	Message	Problem	Recommendations
1.A.A.3 Message	Verilog/VHDL: 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: 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: Do not use case variants of name in the same scope.	<p>Two or more names in your design, within the same scope, are identical except for case.</p> <p>For example, the names <code>foo</code> and <code>Foo</code> cannot be in the same scope.</p>	<p>Update the names in your design so that no two names within the same scope differ only in case.</p> <p>You can disable this rule checking by using the <code>DetectDuplicateNamesCheck</code> property of the HDL coding standard customization object.</p>
1.A.A.6 Warning	<p>Verilog: Primary port names or module names must follow recommended naming convention.</p> <p>VHDL: Component name should be same as its corresponding entity name.</p>	HDL Coder generates code that complies with this rule for Verilog and VHDL.	No action required.

Rule / Severity	Message	Problem	Recommendations
1.A.A.9 Warning	Verilog/VHDL: 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 ModuleInstanceEnt ityNameLength 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: 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.	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 <code>ModuleInstanceEntityNameLength</code> property of the HDL coding standard customization object.
	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.		

1.A.C Signal Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.C.3 Error	Verilog: Signal names, port names, parameter names, define 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 <code>SignalPortParamNameLength</code> property of the HDL coding standard customization object.
	VHDL: Signal names, variable names, type names, label names, and function names should be between 2 and 40 characters in length.		

1.A.D File, Package, and Parameter Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.D.1 Warning	Verilog: 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: Macros defined outside a module must not be used in the module.	HDL Coder does not generate macros in the Verilog code, or redefine constants in the VHDL code.	No action required.
	VHDL: Constants should not be redefined.		
1.A.D.9 Warning	Verilog: 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.

Rule / Severity	Message	Problem	Recommendations
	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: 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.

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: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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 5-49
- “Generate an HDL Coding Standard Report from Simulink” on page 26-6

More About

- “HDL Coding Standard Report” on page 26-2
- “RTL Description Techniques” on page 26-25
- “RTL Design Methodology Guidelines” on page 26-64

RTL Description Techniques

In this section...

"2.A Guidelines for Combinational Logic" on page 26-26

"2.B Guidelines for "Always" Constructs of Combinational Logic" on page 26-35

"2.C Guidelines for Flip-Flop Inference" on page 26-37

"2.D Guidelines for Latch Description" on page 26-43

"2.E Guidelines for Tristate Buffer" on page 26-44

"2.F Guidelines for Always/Process Construct with Circuit Structure into Account" on page 26-46

"2.G Guidelines for "IF" Statement Description" on page 26-47

"2.H Guidelines for "CASE" Statement Description" on page 26-50

"2.I Guidelines for "FOR" Statement Description" on page 26-54

"2.J Guidelines for Operator Description" on page 26-56

"2.K Guidelines for Finite State Machine Description" on page 26-62

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: 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: Check using RTL parsing tool for error prevention.	HDL Coder generates VHDL and Verilog 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: 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 <code>std_logic</code> or <code>std_logic_vector</code> data types to describe ports of an entity.	At the inputs and outputs, HDL Coder uses <code>std_logic</code> or <code>std_logic_vector</code> to describe the ports.	No action required.
2.A.B.2-3 Error	Verilog: Do not use nonblocking assignment, or input argument as input in function description.	The generated HDL code complies with this rule for Verilog.	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: Task constructs should not be used in the design.	HDL Coder does not use tasks or fork-join constructs in the Verilog 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.

Rule / Severity	Message	Problem	Recommendations
2.A.B.5 Error	Verilog: Clock edges should not be used in a task description.	When generating Verilog code, HDL Coder does not use clock edges in a task description.	No action required.
2.A.B.6 Error	VHDL: Specify range for <code>std_logic_vector</code> .	HDL Coder complies with this rule, because the generated VHDL code specifies the range that <code>std_logic_vector</code> uses.	No action required.

2.A.C Bit Width Matching Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.C.1-2 Error	Verilog: 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: 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.

Rule / Severity	Message	Problem	Recommendations
2.A.C.4-5 Error	Verilog: In function description, do not assign global signals, and return value assignment must be the last statement.	HDL Coder generates Verilog 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: Bit-wise operators must be used instead of logical operators in multi-bit operations.	In the generated Verilog code, HDL Coder complies with this rule for multibit operators.	No action required.
2.A.D.6 Message	Verilog: 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: 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: 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: 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: 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: 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: 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).	No action required.
	Verilog: 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: 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: 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: 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)	No action required.
2.C.A.4-5b Warning	Verilog/VHDL: 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: 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.

Rule / Severity	Message	Problem	Recommendations
2.C.A.7 Message	Verilog/VHDL: 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.
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: 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: 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: 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.

Rule / Severity	Message	Problem	Recommendations
2.C.C.6 Warning	Verilog/VHDL: 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: 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: 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: 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: 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: 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: 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.

Rule / Severity	Message	Problem	Recommendations
2.E.A.6-9 Error	Verilog/VHDL: 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: 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: 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 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 <code>ConditionalRegionCheck</code> property of the HDL coding standard customization object.
2.F.B.2 Error	Verilog/VHDL: 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: 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

Rule / Severity	Message	Problem	Recommendations
2.G.C.1a-b Message	Verilog/VHDL: 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.	The MATLAB code contains an if-elseif statement with more than N levels of nesting. By default, N is 3.	<p>Modify if-elseif statements in your MATLAB code so there are N or fewer levels of nesting.</p> <p>For example, the following if-elseif pseudocode contains three levels of nesting:</p> <pre data-bbox="1029 722 1326 895">if ... if ... if ... else else else else</pre> <p>You can customize this rule by using the <code>IfElseNesting</code> property of the HDL coding standard customization object.</p>

Rule / Severity	Message	Problem	Recommendations
2.G.C.1c Message	Verilog/VHDL: Chain of if...else if constructs must not be exceed default number of levels.	The generated HDL code contains an if-elseif statement with more than seven branches.	<p>Modify if-elseif statements in your MATLAB code so that the number of branches is seven or fewer.</p> <p>For example, the following if-elseif pseudocode contains three branches:</p> <pre>if ... elseif ... elseif ... else</pre> <p>You can customize this rule by using the IfElseChain property of the HDL coding standard customization object.</p>

2.G.D Begin-End Decorator Constraints

Rule / Severity	Message	Problem	Recommendations
2.G.D.2-3 Message	Verilog/VHDL: Attach begin-end to "if" statements.	The generated HDL code complies with these code constructs.	No action required.
	Verilog: 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: 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: 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: 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 <i>don't care</i> value in the default clause.	No action required.
2.H.C.5 Warning	Verilog/VHDL: 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.

Rule / Severity	Message	Problem	Recommendations
2.H.C.6-7 Message	Verilog/VHDL: 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 <i>don't care</i> 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: 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: Do not use <code>parallel_case</code> 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 <code>parallel_case</code> directive and generates code that complies with these constructs.	No action required.

2.1 Guidelines for “FOR” Statement Description

2.1.A Loop Body Constraints

Rule / Severity	Message	Problem	Recommendations
2.I.A.2a-b Message	Verilog: 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: Loop variable of "for" construct must have a constant value inside the construct and must not be used outside the construct.	HDL Coder generates the right loop constructs and complies with this rule.	No action required.
	Verilog: The loop termination condition must not be a constant.		

2.1.B Non-Constant Operation Constraints

Rule / Severity	Message	Problem	Recommendations
2.I.B.4 Error	Verilog/VHDL: 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.

2.1.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: 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: 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: 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.

Rule / Severity	Message	Problem	Recommendations
	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: 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: 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: 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: 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: 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: 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: 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.	<p>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.</p> <p>You can customize this rule by using the <code>MultiplierBitWidth</code> property of the HDL coding standard customization object.</p>

2.J.G Common Sub-Expression Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.G.2 Warning	Verilog/VHDL: 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: 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: 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: 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: 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 5-49
- “Generate an HDL Coding Standard Report from Simulink” on page 26-6

More About

- “HDL Coding Standard Report” on page 26-2
- “Basic Coding Practices” on page 26-10
- “RTL Design Methodology Guidelines” on page 26-64

RTL Design Methodology Guidelines

In this section...
"3.A Guidelines for Creating Function Libraries" on page 26-64
"3.B Guidelines for Using Function Libraries" on page 26-66
"3.C Guidelines for Test Facilitation Design" on page 26-69

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: The order of module port declarations 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: 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: Multiple assignments should not be made in one line.	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.
	Verilog/VHDL: The maximum number of characters in one line should not be more than N.		

3.A.F Generic Usage Constraints

Rule / Severity	Message	Problem	Recommendations
3.A.F.1 Reference	Verilog: 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: 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: 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: 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: 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: Non-integer type used in the declaration of a generic may be unsynthesizable.	The generated HDL code contains a noninteger data type.	<p>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.</p> <p>Alternatively, modify your design so that it does not use floating-point data types.</p> <p>You can disable this rule checking by using the <code>NonIntegerTypes</code> property of the HDL coding standard customization object.</p>
3.B.D.3 Error	Verilog: 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: Internal clocks and asynchronous sets/resets must be controllable from external pins.	<p>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.</p> <p>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.</p>	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: 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: 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: 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 5-49
- “Generate an HDL Coding Standard Report from Simulink” on page 26-6

More About

- “HDL Coding Standard Report” on page 26-2
- “Basic Coding Practices” on page 26-10
- “RTL Description Techniques” on page 26-25

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 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')
```

Interfacing Subsystems and Models to HDL Code

- “Model Referencing for HDL Code Generation” on page 27-2
- “Generate Black Box Interface for Subsystem” on page 27-5
- “Generate Black Box Interface for Referenced Model” on page 27-10
- “Integrate Custom HDL Code Using DocBlock” on page 27-12
- “Customize Black Box or HDL Cosimulation Interface” on page 27-14
- “Specify Bidirectional Ports” on page 27-18
- “Generate Reusable Code for Atomic Subsystems” on page 27-20
- “Create a Xilinx System Generator Subsystem” on page 27-28
- “Create an Altera DSP Builder Subsystem” on page 27-31
- “Using Xilinx System Generator for DSP with HDL Coder” on page 27-34
- “Choose a Test Bench for Generated HDL Code” on page 27-38
- “Generate a Cosimulation Model” on page 27-41
- “Pass-Through and No-Op Implementations” on page 27-62
- “Synchronous Subsystem Behavior with the State Control Block” on page 27-63

Model Referencing for HDL Code Generation

In this section...
“Benefits of Model Referencing for Code Generation” on page 27-2
“How To Generate Code for a Referenced Model” on page 27-2
“Generate Code for Model Arguments” on page 27-3
“Generate Comments” on page 27-3
“Limitations” on page 27-3

Benefits of Model Referencing for 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.

However, HDL Coder treats `If any changes detected` and `If any changes in known dependencies detected` as the same. For example, if you set **Rebuild** to either `If any changes detected` or `If any changes in known dependencies detected`, HDL Coder regenerates code for referenced models only when the referenced models have changed.

How To Generate Code for a Referenced Model

When generating code, if you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, consider using the `ScalarizePorts` property to generate non-conflicting port definitions. For more information, see “Scalarize vector ports” on page 16-67.

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 module or VHDL entity for instances of a referenced model with different model argument values, see “Generate Parameterized Code for Referenced Models” on page 10-27.

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.
- Model block cannot be a masked subsystem.
- Multiple model references that refer to the same model must have the same HDL block properties.
- Referenced models cannot be protected models.
- Hierarchical distributed pipelining must be disabled.

HDL Coder cannot move registers across a model reference. Therefore, referenced models can inhibit these optimizations:

- Distributed pipelining
- Constrained output pipelining
- Streaming

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 27-5

“Generate a Black Box Interface for a Subsystem” on page 27-5

“Generate Code for a Black Box Subsystem Implementation” on page 27-8

“Restriction for Multirate DUTs” on page 27-9

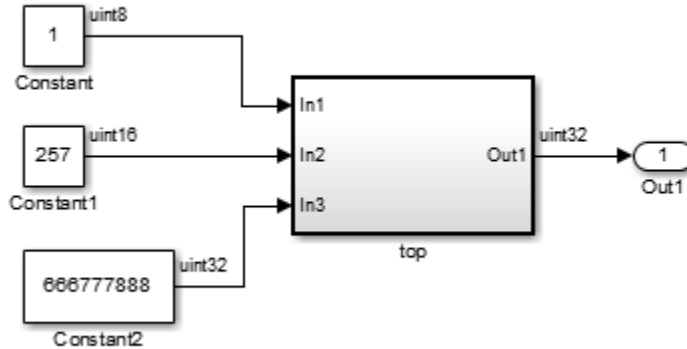
What Is a Black Box Interface?

A *black box* interface for a subsystem is a generated VHDL component or Verilog 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.

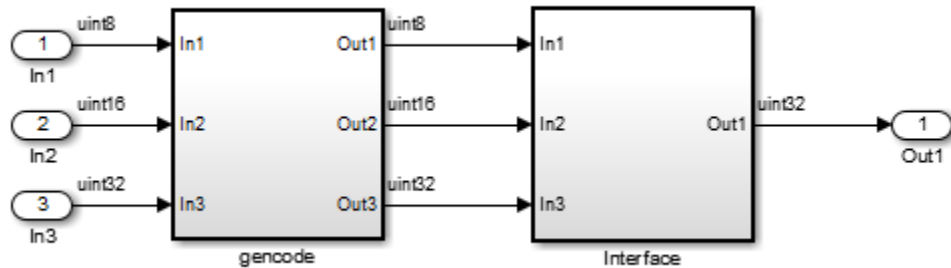
The black box implementation is available only for subsystem blocks below the level of the DUT. Virtual and atomic subsystem blocks of custom libraries that are below the level of the DUT also work with black box implementations.

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:



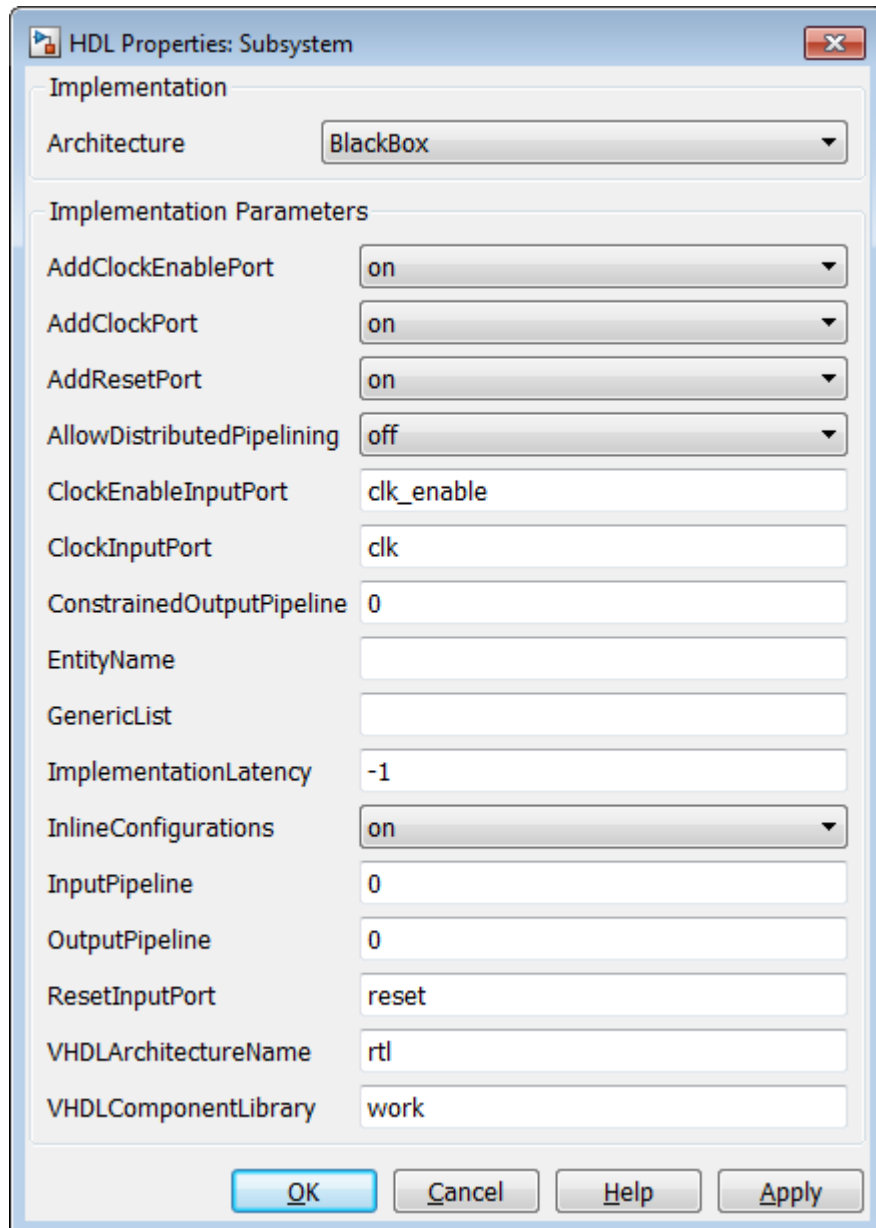
Suppose that you want to generate HDL code from top, with a black box interface from the Interface subsystem. To specify a black box interface:

- 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 image shows a dialog box titled "HDL Properties: Subsystem". It has a standard Windows-style title bar with a close button. The dialog is divided into two main sections: "Implementation" and "Implementation Parameters".

Implementation Section:

- Architecture: A dropdown menu set to "BlackBox".

Implementation Parameters Section:

- AddClockEnablePort: A dropdown menu set to "on".
- AddClockPort: A dropdown menu set to "on".
- AddResetPort: A dropdown menu set to "on".
- AllowDistributedPipelining: A dropdown menu set to "off".
- ClockEnableInputPort: A text field containing "clk_enable".
- ClockInputPort: A text field containing "clk".
- ConstrainedOutputPipeline: A text field containing "0".
- EntityName: An empty text field.
- GenericList: An empty text field.
- ImplementationLatency: A text field containing "-1".
- InlineConfigurations: A dropdown menu set to "on".
- InputPipeline: A text field containing "0".
- OutputPipeline: A text field containing "0".
- ResetInputPort: A text field containing "reset".
- VHDLArchitectureName: A text field containing "rtl".
- VHDLComponentLibrary: A text field containing "work".

At the bottom of the dialog, there are four buttons: "OK", "Cancel", "Help", and "Apply".

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 27-14 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 27-14 describes how you can rename or suppress generation of these signals, and customize other aspects of the generated interface.

Restriction for Multirate DUTs

You can generate at most one clock port and one clock enable port for a black box subsystem. Therefore, the black box subsystem must be single-rate even if it is in a multirate DUT.

See Also

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 27-14
- “Generate Black Box Interface for Referenced Model” on page 27-10
- “Integrate Custom HDL Code Using DocBlock” on page 27-12

Generate Black Box Interface for Referenced Model

In this section...
“When to Generate a Black Box Interface” on page 27-10
“How to Generate a Black Box Interface” on page 27-10
“Caveats and Limitations” on page 27-11

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 27-2.

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 27-14.
- 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 vector ports” on page 16-67.

See Also

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 27-14
- “Generate Black Box Interface for Subsystem” on page 27-5
- “Integrate Custom HDL Code Using DocBlock” on page 27-12

Integrate Custom HDL Code Using DocBlock

In this section...
“When To Use DocBlock to Integrate Custom Code” on page 27-12
“How To Use DocBlock to Integrate Custom Code” on page 27-12
“Restrictions” on page 27-13
“Example” on page 27-13

You can use one or more DocBlock blocks to integrate custom HDL code into your design.

When To Use DocBlock to Integrate Custom Code

If you want to keep the HDL code with 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 or Verilog code.

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.

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 IP or a library from a third party, use a black box subsystem on page 27-5 or black box model reference on page 27-10.

How To 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. To learn more about customizing the black box interface, see “Customize Black Box or HDL Cosimulation Interface” on page 27-14.
- 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, either Verilog or VHDL.
- 5 In the DocBlock, enter the HDL code for your custom Verilog 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 must have **TargetLanguage** set to VHDL, and the other must have **TargetLanguage** set to Verilog.

When generating code, HDL Coder only integrates the code from the DocBlock that matches the target language for code generation.

Example

The `hdlcoderIncludeCustomHdlUsingDocBlockExample` model shows how to integrate custom VHDL and Verilog code into your design with the DocBlock block.

See Also

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 27-14
- “Generate Black Box Interface for Subsystem” on page 27-5
- “Generate Black Box Interface for Referenced Model” on page 27-10

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.

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 .
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 .
AllowDistributedPipelining	on off Default: off	If on, allow HDL Coder to move registers across the block, from input to output or output to input.
ClockEnableInputPort	Default: clk_enable	Specifies HDL name for block's clock enable input port.

Parameter Name	Values	Description
ClockInputPort	Default: <code>clk</code>	Specifies HDL name for block's clock input signal.
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 <code>entity</code> or Verilog <code>module</code> name generated for the block.

Parameter Name	Values	Description
GenericList	<p>Pass a cell array variable that contains cell arrays each with two or three strings, or enter a cell array of cell arrays that each contain two or three strings. The strings represent the name, value, and optional data type of a VHDL generic or Verilog parameter. The default data type is integer.</p> <p>Default: none</p>	<p>Specifies a list of VHDL generic or Verilog parameter name-value pairs, each with an optional data type specification, to pass to a subsystem with a BlackBox implementation.</p> <p>For example, in the HDL Block Properties dialog box, enter <code>{'name', 'value', 'type'}</code>, or, if the data type is integer, enter <code>{'name', 'value'}</code>.</p> <p>To set <code>GenericList</code> using <code>hdlset_param</code>, at the command line, enter:</p> <pre>hdlset_param (blockname, 'GenericList', {' 'name', 'value', 'type'}));</pre> <p>If the data type is integer, at the command line, enter:</p> <pre>hdlset_param (blockname, 'GenericList', {' 'name ', 'value'}));</pre>
ImplementationLatency	<p>-1 0 positive integer</p> <p>Default: -1</p>	<p>Specifies the additional latency of the external component in time steps, relative to the Simulink block.</p> <p>If 0 or greater, this value is used for delay balancing. Your inputs and outputs must operate at the same rate.</p> <p>If -1, latency is unknown. This disables delay balancing.</p>

Parameter Name	Values	Description
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.
ResetInputPort	Default: reset	Specifies HDL name for block's reset input.
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 27-5
- “Generate Black Box Interface for Referenced Model” on page 27-10
- “Integrate Custom HDL Code Using DocBlock” on page 27-12
- “Specify Bidirectional Ports” on page 27-18

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 or VHDL `inout` keyword.

In the FPGA Turnkey workflow, you can use the bidirectional ports to connect to external RAM.

In this section...

“Requirements” on page 27-18

“How To Specify a Bidirectional Port” on page 27-18

“Limitations” on page 27-19

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

- In the FPGA Turnkey workflow, in the **Target platform interfaces table**, you must map a bidirectional port to either Specify FPGA Pin {'LSB', ..., 'MSB'} or one of the other interfaces where the interface bitwidth exactly matches your bidirectional port bitwidth.

For example, you can map a 32-bit bidirectional port to the Expansion Headers J6 Pin 2-64[0:31] interface.

- You cannot generate a Verilog 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 27-5
- “Generate Black Box Interface for Referenced Model” on page 27-10
- “Integrate Custom HDL Code Using DocBlock” on page 27-12
- “Customize Black Box or HDL Cosimulation Interface” on page 27-14

Generate Reusable Code for Atomic Subsystems

In this section...

“Requirements for Generating Reusable Code for Atomic Subsystems” on page 27-20

“Generate Reusable Code for Atomic Subsystems” on page 27-21

“Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters” on page 27-23

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

To generate reusable HDL 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 the `InlineParams` property to `on`. Enabling this parameter is the same as setting `DefaultParameterBehavior` to `Inlined`. Setting `InlineParams` to `off` changes `DefaultParameterBehavior` value to `Tunable`.

- The atomic subsystems must be identical, or identical except for their mask parameter values.
 - `MaskParameterAsGeneric` must be `on`. For more information, see “Generate parameterized HDL code from masked subsystem” on page 16-70.
 - Mask parameters must be nontunable. The code generator does not share atomic subsystems with mask parameters that are tunable.
 - Mask parameter data types cannot be `double` or `single`.
 - The tunable parameter must be used in only `Constant` or `Gain` 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.

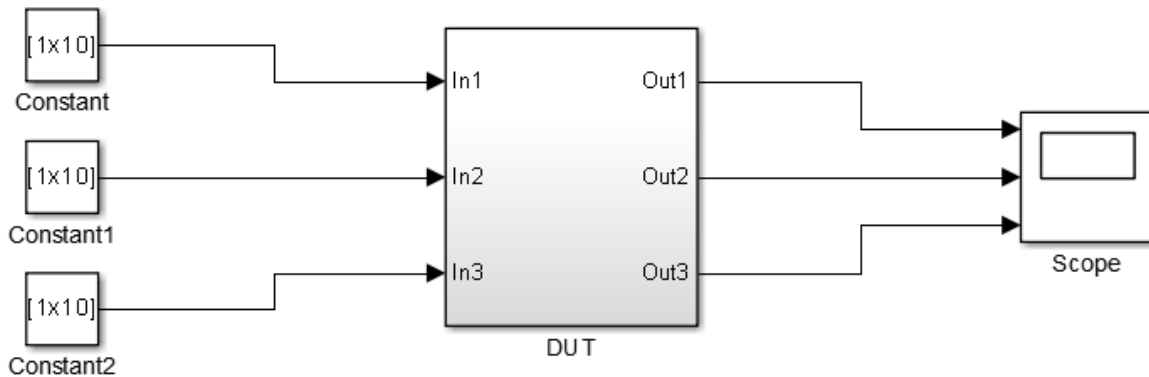
Generate Reusable Code for Atomic Subsystems

If your design contains identical atomic subsystems, the coder generates one HDL module or entity for the subsystem and instantiates it multiple times.

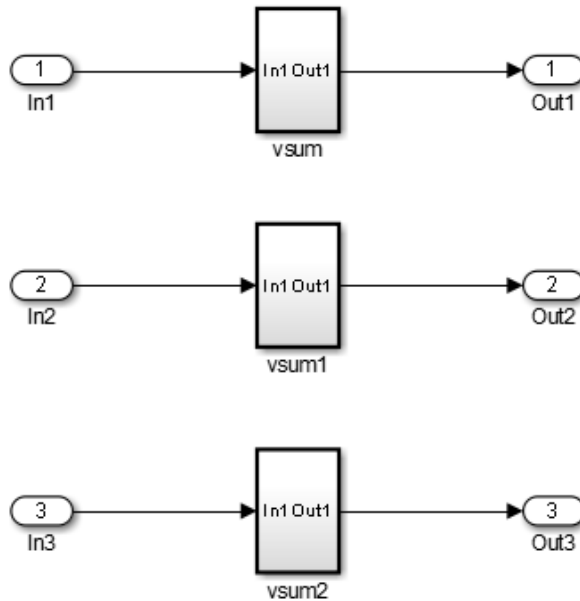
Example

The `hdlcoder_reusable_code_identical_subsystem` model shows an example of a DUT subsystem containing three identical atomic subsystems.

 `hdlcoder_reusable_code_identical_subsystem` ▶



hdlcoder_reusable_code_identical_subsystem ▶ DUT ▶



HDL Coder generates a single VHDL file, `vsum.vhd`, for the three subsystems.

```

makehdl('hdlcoder_reusable_code_identical_subsystem/DUT')

### Generating HDL for 'hdlcoder_reusable_code_identical_subsystem/DUT'.
### Starting HDL check.
### Generating new validation model: gm_hdlcoder_reusable_code_identical_subsystem_vnl.
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_reusable_code_identical_subsystem'.
### Working on hdlcoder_reusable_code_identical_subsystem/DUT/vsum/Sum of Elements as
hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\Sum_of_Elements.vhd.
### Working on hdlcoder_reusable_code_identical_subsystem/DUT/vsum as
hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\vsum.vhd.
### Working on hdlcoder_reusable_code_identical_subsystem/DUT as
hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\DUT.vhd.
### Generating package file hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\DUT_pkg.vhd.
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'hdlcoder_reusable_code_identical_subsystem' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
  
```

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 [10]
        Out1        : OUT  std_logic_vector(19 DOWNT0 0) -- sfix20
        );
END COMPONENT;

-- Component Configuration Statements
FOR ALL : vsum
  USE ENTITY work.vsum(rtl);

-- Signals
SIGNAL vsum_out1      : std_logic_vector(19 DOWNT0 0); -- ufix20
SIGNAL vsum1_out1    : std_logic_vector(19 DOWNT0 0); -- ufix20
SIGNAL vsum2_out1    : std_logic_vector(19 DOWNT0 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

If your design contains 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.

To generate reusable code for identical atomic subsystems, enable `MaskParameterAsGeneric` for the model. By default, `MaskParameterAsGeneric` is disabled.

For example, to enable the generation of reusable code for the atomic subsystems with tunable parameters in the `hdlcoder_reusable_code_parameterized_subsystem` model, enter:

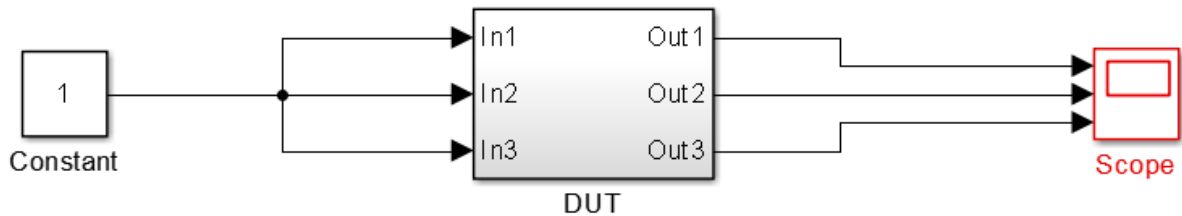
```
hdlset_param('hdlcoder_reusable_code_parameterized_subsystem', 'MaskParameterAsGeneric', 'on')
```

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.

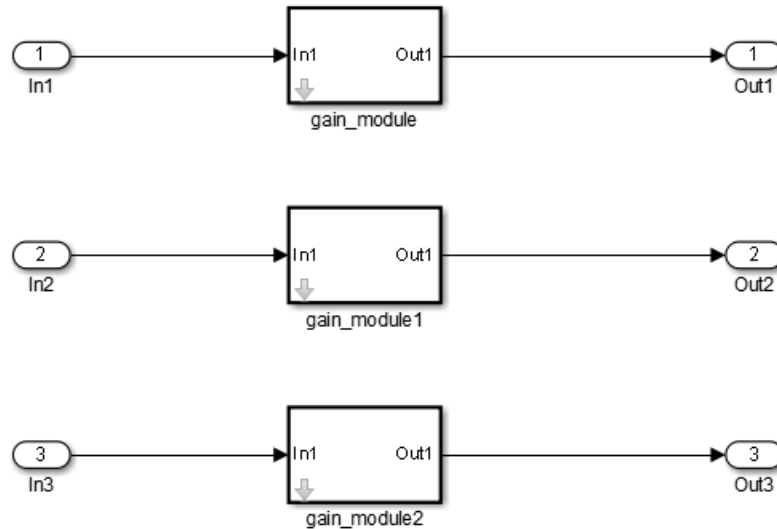
Example

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.

 `hdlcoder_reusable_code_parameterized_subsystem` ▶



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`.

With `MaskParameterAsGeneric` enabled, HDL Coder generates a single source file, `gain_module.v`, for the three gain module subsystems.

```

makehdl('hdlcoder_reusable_code_parameterized_subsystem/DUT', 'MaskParameterAsGeneric', 'on', ...
        'TargetLanguage', 'Verilog')

### Generating HDL for 'hdlcoder_reusable_code_parameterized_subsystem/DUT'.
### Starting HDL check.
### Begin Verilog Code Generation for 'hdlcoder_reusable_code_parameterized_subsystem'.
### Working on hdlcoder_reusable_code_parameterized_subsystem/DUT/gain_module as
    hdlsrc\hdlcoder_reusable_code_parameterized_subsystem\gain_module.v.
### Working on hdlcoder_reusable_code_parameterized_subsystem/DUT as
    hdlsrc\hdlcoder_reusable_code_parameterized_subsystem\DUT.v.
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'hdlcoder_reusable_code_parameterized_subsystem' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
  
```

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
```

See Also

More About

- “Generate parameterized HDL code from masked subsystem” on page 16-70
- “Generate parameterized HDL code from masked subsystem” on page 16-70
- “Generate Parameterized Code for Referenced Models” on page 10-27
- “Create and Add Tunable Parameter That Maps to DUT Ports” on page 10-24

Create a Xilinx System Generator Subsystem

In this section...

“Why Use Xilinx System Generator Subsystems?” on page 27-28

“Requirements for Xilinx System Generator Subsystems” on page 27-28

“How to Create a Xilinx System Generator Subsystem” on page 27-29

“Limitations for Code Generation from Xilinx System Generator Subsystems” on page 27-29

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 both Simulink and Xilinx blocks in your model provides the following benefits:

- 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; HDL Coder generates code from Simulink blocks.
- HDL Coder area and speed optimizations for Simulink components.

Requirements for Xilinx System Generator Subsystems

You must 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 on 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 with:

- 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 27-29.

How to 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.

For an example of HDL code generation from a Xilinx System Generator subsystem, see “Using Xilinx System Generator for DSP with HDL Coder” on page 27-34.

Limitations for Code Generation from Xilinx System Generator Subsystems

Code generation from Xilinx System Generator (XSG) subsystems has the following 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 `sfixed_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.

Create an Altera DSP Builder Subsystem

In this section...

“Why Use Altera DSP Builder Subsystems?” on page 27-31

“Requirements for Altera DSP Builder Subsystems” on page 27-31

“How to Create an Altera DSP Builder Subsystem” on page 27-32

“Determine Clocking Requirements for Altera DSP Builder Subsystems” on page 27-32

“Limitations for Code Generation from Altera DSP Builder Subsystems” on page 27-33

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.

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 27-32 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.

Using Xilinx System Generator for DSP with HDL Coder

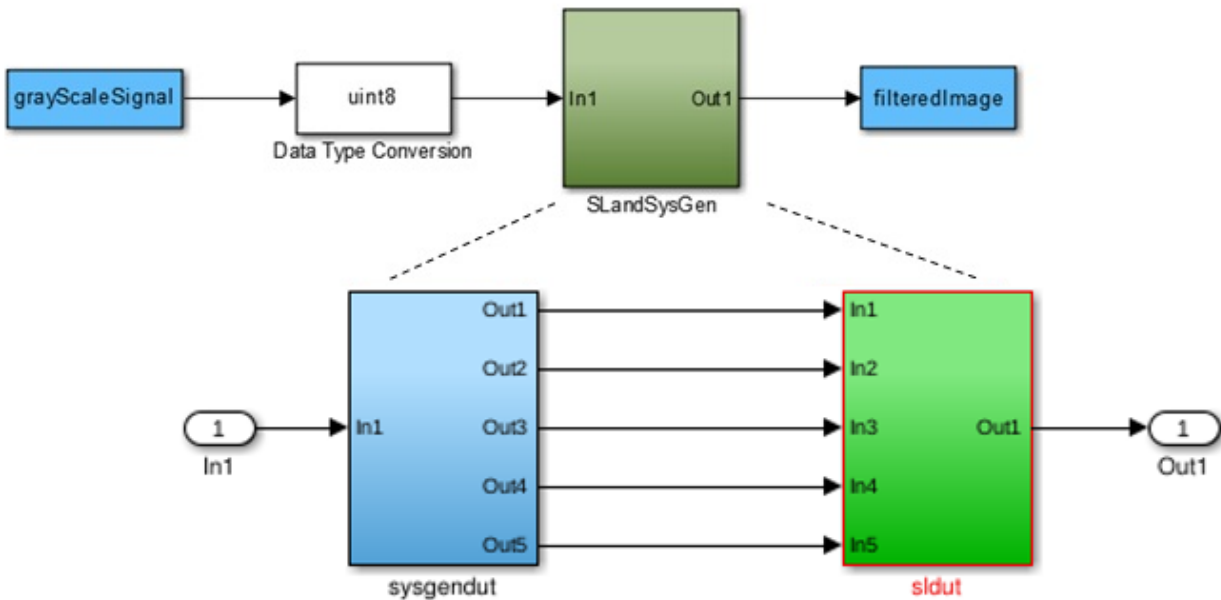
This example shows how to use Xilinx® System Generator for DSP with HDL Coder™.

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 in a Xilinx® System Generator Subsystem (hdlcoder_slsysgen/SLandSysGen/sysgendut). System Generator optimizes these blocks for Xilinx® 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_slsysgen');  
open_system('hdlcoder_slsysgen/SLandSysGen');
```

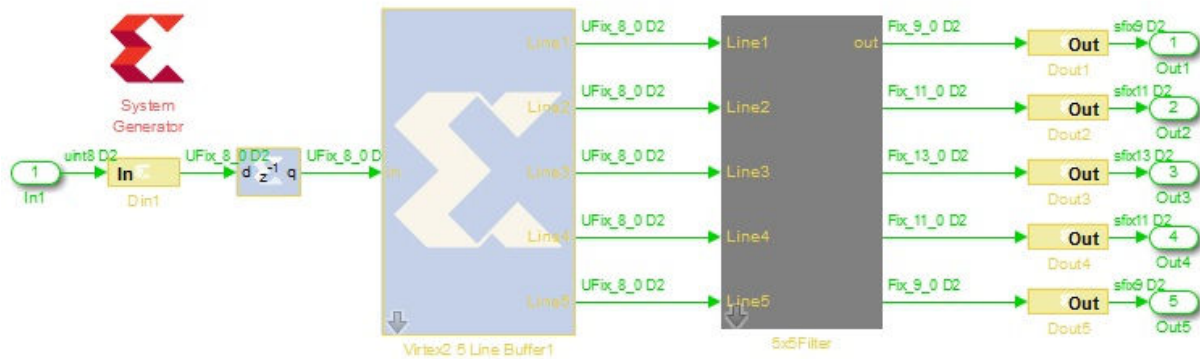


Create Xilinx® System Generator Subsystem

To create a Xilinx® System Generator subsystem:

- 1 Put the Xilinx® blocks in one subsystem and set its architecture to "Module" (the default value).
- 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.

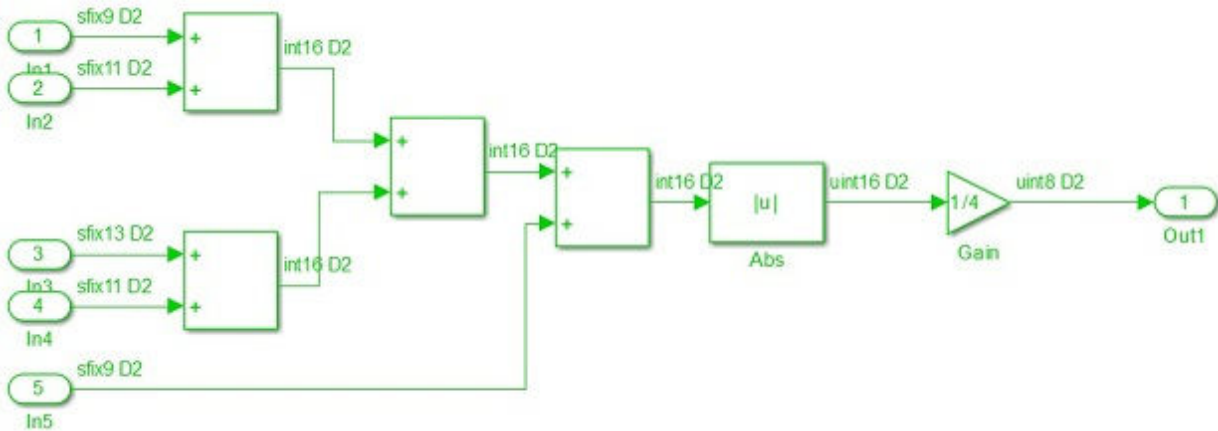
Gateway In blocks must not do non-trivial data type conversion. For example, a Gateway In block can convert between uint8 and UFix_8_0, but changing data sign, word length, or fraction length is not allowed.

Perform Model-Level Optimizations for Simulink® Components

In this example, a sum tree is modeled with Simulink® blocks. The distributed pipelining feature can take care of the speed optimization.

Here the Output Pipeline property of hdlcoder_slsgen/SLandSysGen/Simulink Subsystem is set to "4" and the Distributed Pipelining property is set to "on". Pipeline registers inserted by the distributed pipelining feature will be pushed 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_slsgen/SLandSysGen/sldut');
```



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_slsgen/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 35-2.
- 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 27-41.
- 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 With Large Data Set Using SystemVerilog DPI Test Bench”.
- 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 `makehdl tb`.

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		<ul style="list-style-type: none"> Fast compile time in HDL simulator 	<ul style="list-style-type: none"> 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
Cosimulation model	<ul style="list-style-type: none"> HDL Verifier 	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> HDL Verifier Simulink Coder 	<ul style="list-style-type: none"> 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 	<ul style="list-style-type: none"> Run test in HDL simulator No tunable parameters in stimulus generation

Test Bench	License Requirements	Pros	Cons
FPGA-in-the-loop	<ul style="list-style-type: none"> • HDL Verifier • HDL Verifier Support Package for Xilinx FPGA Boards or HDL Verifier Support Package for Intel FPGA Boards 	<ul style="list-style-type: none"> • Run tests from Simulink, including changing parameters to affect input stimulus • Prototype hardware implementation of your DUT 	<ul style="list-style-type: none"> • Long generation time due to synthesis into FPGA • Hardware setup

See Also

More About

- “Set HDL Code Generation Options” on page 11-2

Generate a Cosimulation Model

In this section...

“What Is A Cosimulation Model?” on page 27-41

“Generating a Cosimulation Model from the GUI” on page 27-42

“Structure of the Generated Model” on page 27-48

“Launching a Cosimulation” on page 27-54

“The Cosimulation Script File” on page 27-57

“Complex and Vector Signals in the Generated Cosimulation Model” on page 27-59

“Generating a Cosimulation Model from the Command Line” on page 27-60

“Naming Conventions for Generated Cosimulation Models and Scripts” on page 27-60

“Limitations for Cosimulation Model Generation” on page 27-61

Note To use this feature, your installation must include an HDL Verifier license.

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 or Cadence Incisive.
- 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 .

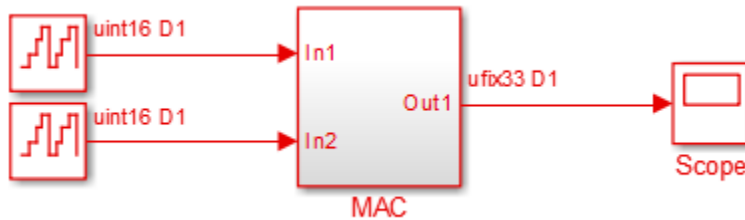
In addition to the generated model, HDL Coder generates a TCL script that launches and configures your cosimulation tool. Comments in the script file document clock, reset, and other timing signal information defined by the coder for the cosimulation tool.

Generating a Cosimulation Model from the GUI

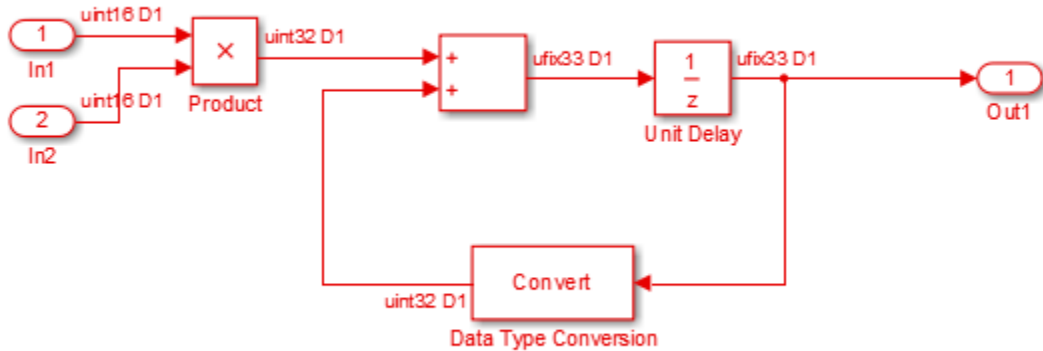
This example demonstrates the process for generating a cosimulation model. The example model, `hdl_cosim_demo1`, implements a simple multiply and accumulate (MAC) algorithm. Open the model by entering the name at the MATLAB command line:

```
hdl_cosim_demo1
```

The following figure shows the top-level model.

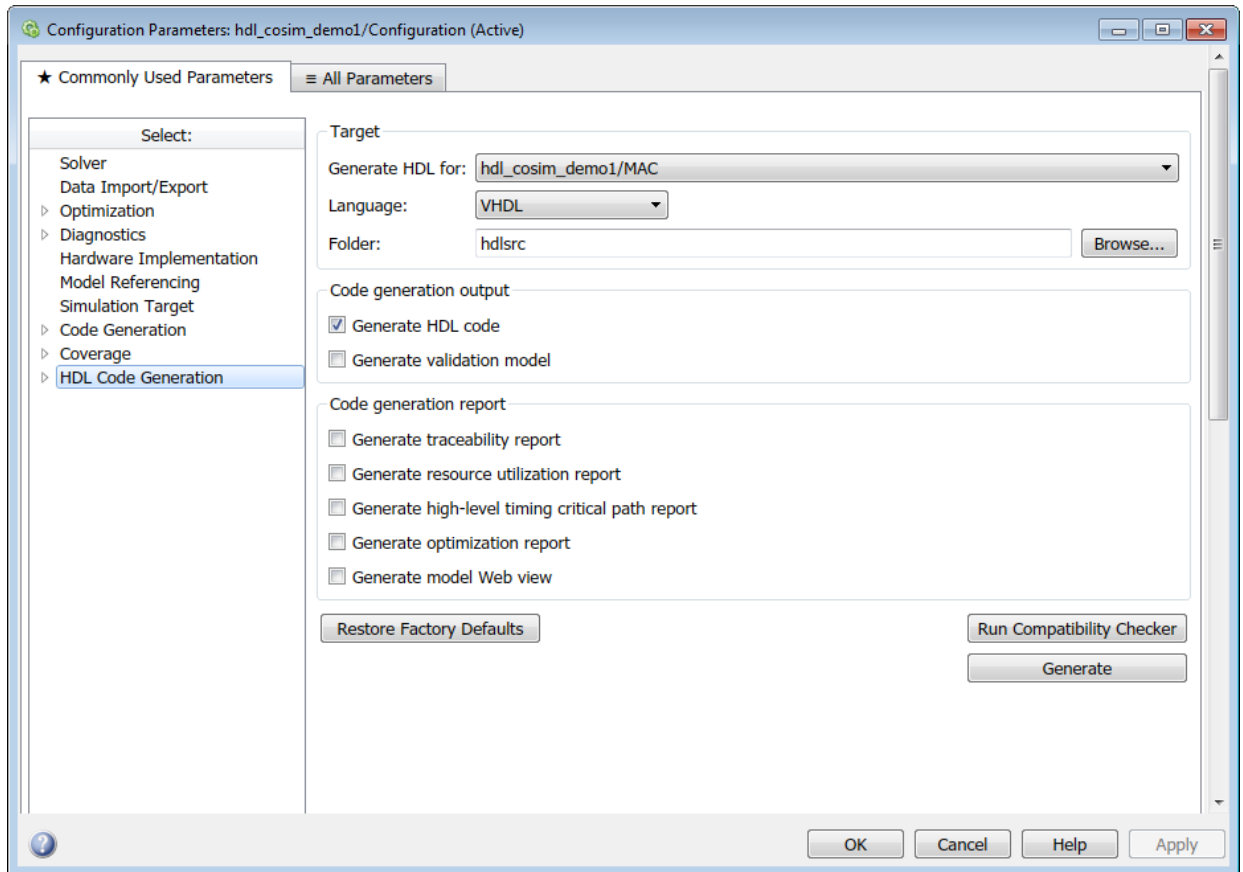


The DUT is the MAC subsystem.



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 In the **HDL Code Generation** pane of the Configuration Parameters dialog box, select the DUT for code generation. In this case, it is `hdl_cosim_demo1/MAC`.



- 2 Click **Apply**.
- 3 Click **Generate**. HDL Coder displays progress messages, as shown in the following listing:

```

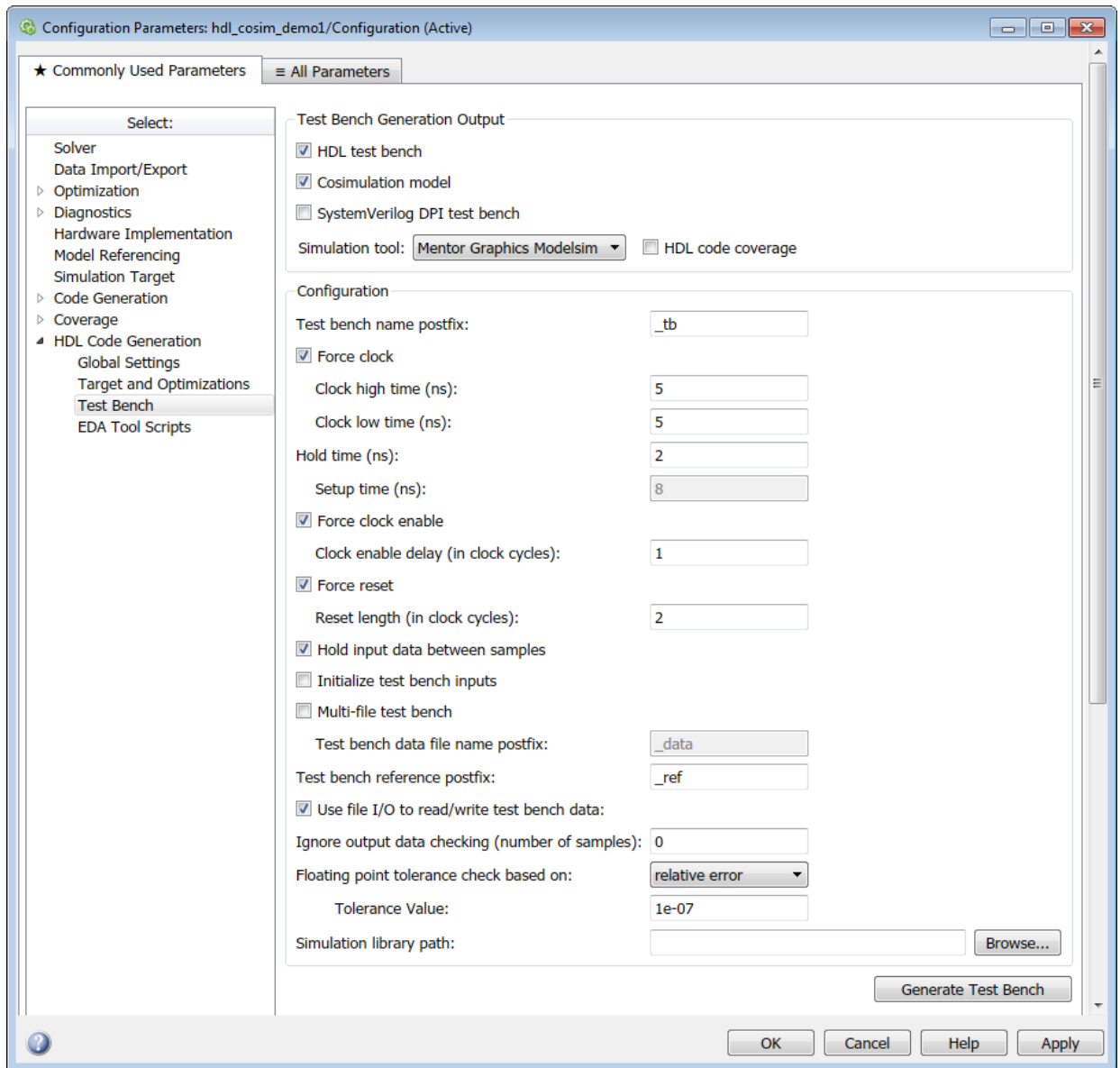
### Applying HDL Code Generation Control Statements
### Starting HDL Check.
### HDL Check Complete with 0 error, 0 warning and 0 message.

### 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 pull-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 27-57).

4 Click **Apply**.

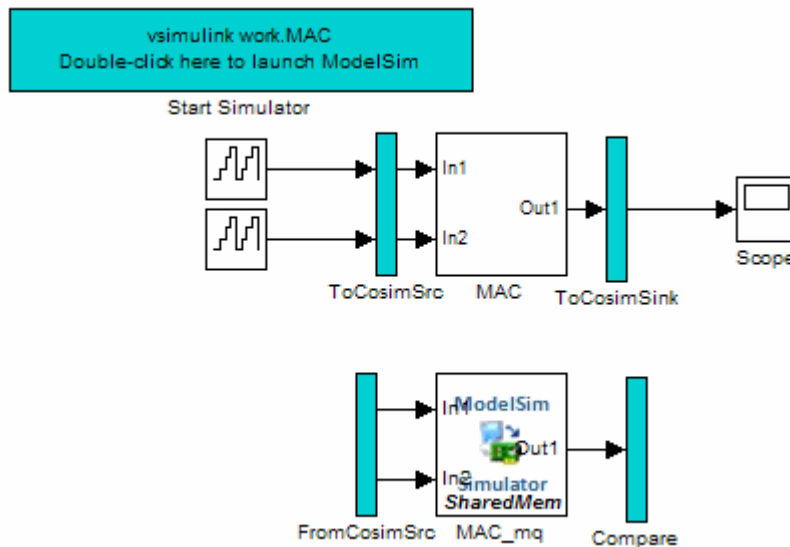
Next, generate test bench code and the cosimulation model:

- 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 27-60.

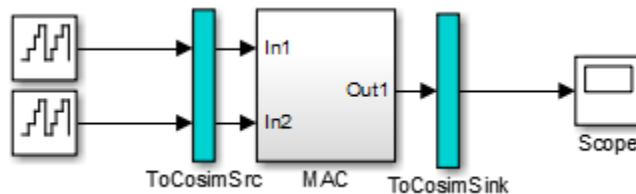
The next section, “Structure of the Generated Model” on page 27-48, describes the features of the model. Before running a cosimulation, become familiar with these features.

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

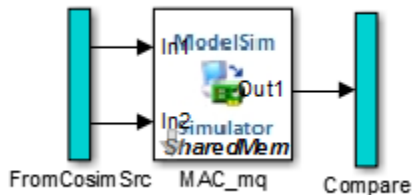
The model comprises two parallel signal paths. The simulation path, located in the upper half of the model window, is nearly identical to the original DUT. The purpose of the simulation path is to execute a normal Simulink simulation and provide a reference signal for comparison to the cosimulation results. 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 27-51).

Cosimulation Path

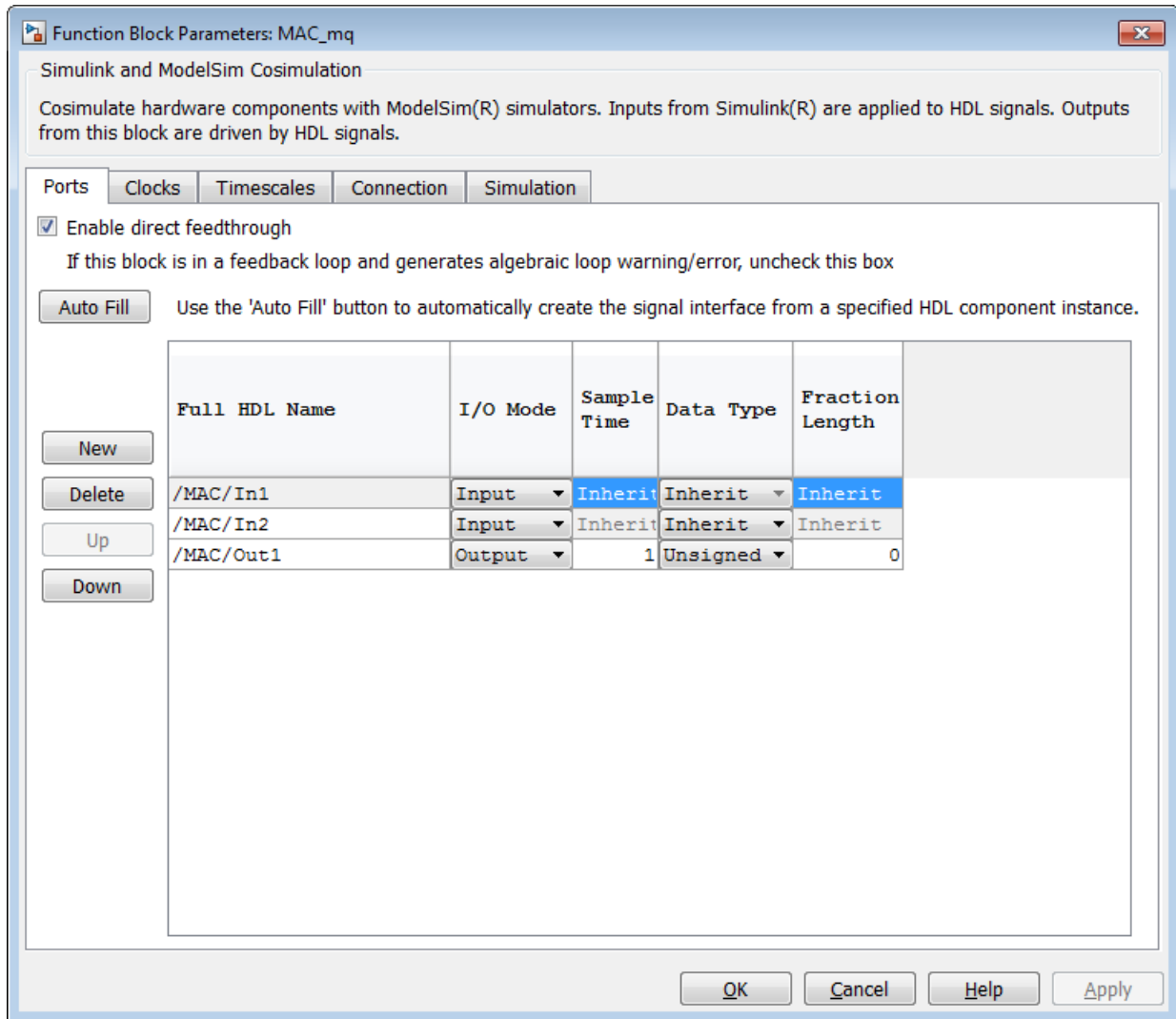
The cosimulation path, located in the lower half of the model window, contains the generated HDL Cosimulation block. The following figure shows the cosimulation path.



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 27-51).

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 27-53 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.

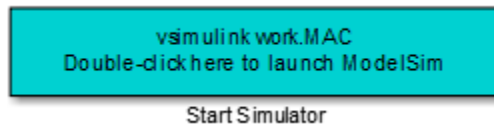


HDL Coder sets the **Full HDL Name**, **Sample Time**, **Data Type**, and other fields as required by the model. HDL Coder also configures other HDL Cosimulation block parameters under the **Timescales** and **Tcl** panes.

Tip HDL Coder configures the generated HDL Cosimulation block for the Shared Memory connection method.

Start Simulator Control

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.

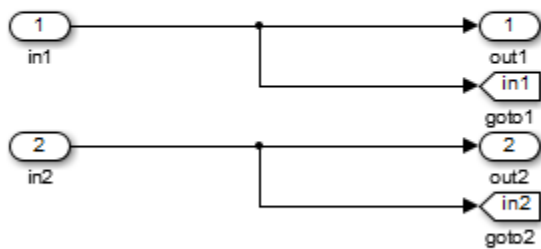


The commands executed when you double-click the Start Simulator icon launch and set up the cosimulation tool, but they do not start the actual cosimulation. “Launching a Cosimulation” on page 27-54 describes how to run a cosimulation with the generated model.

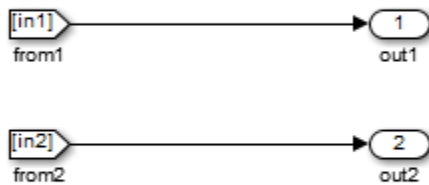
Signal Routing Between Simulation and Cosimulation Paths

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.

gm_hdl_cosim_demo1_mq ▶ ToCosimSrc



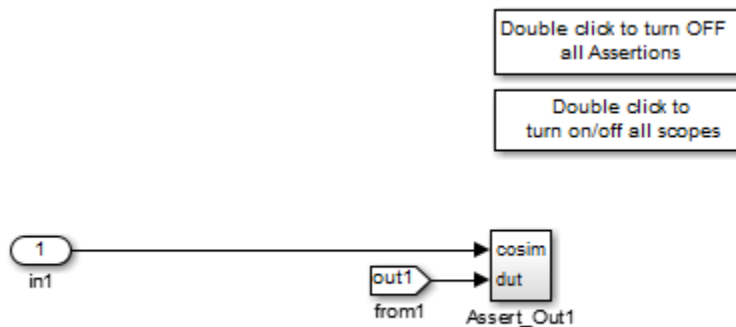
gm_hdl_cosim_demo1_mq ▶ FromCosimSrc



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 27-59 for further information.

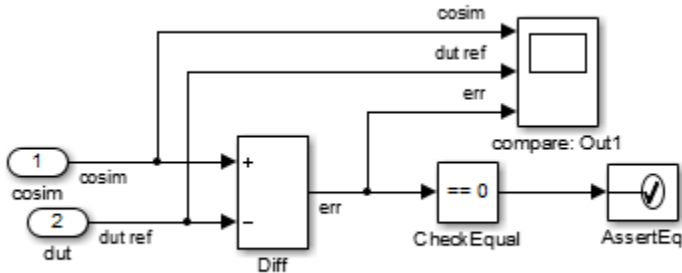
Controlling Assertions and Scope Displays

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.

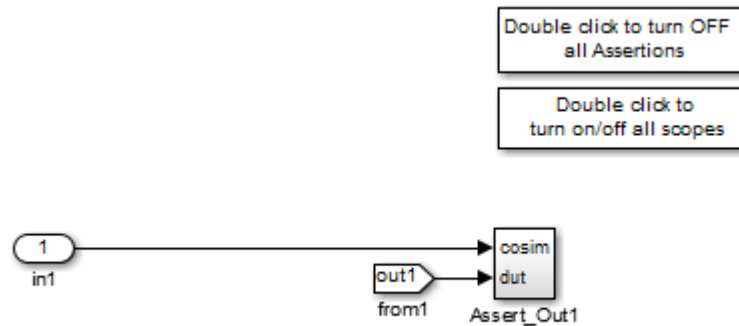
Tip Assertion messages are warnings and do not stop simulation.

Launching a Cosimulation

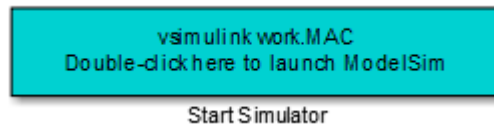
To run a cosimulation with the generated model:

- 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.

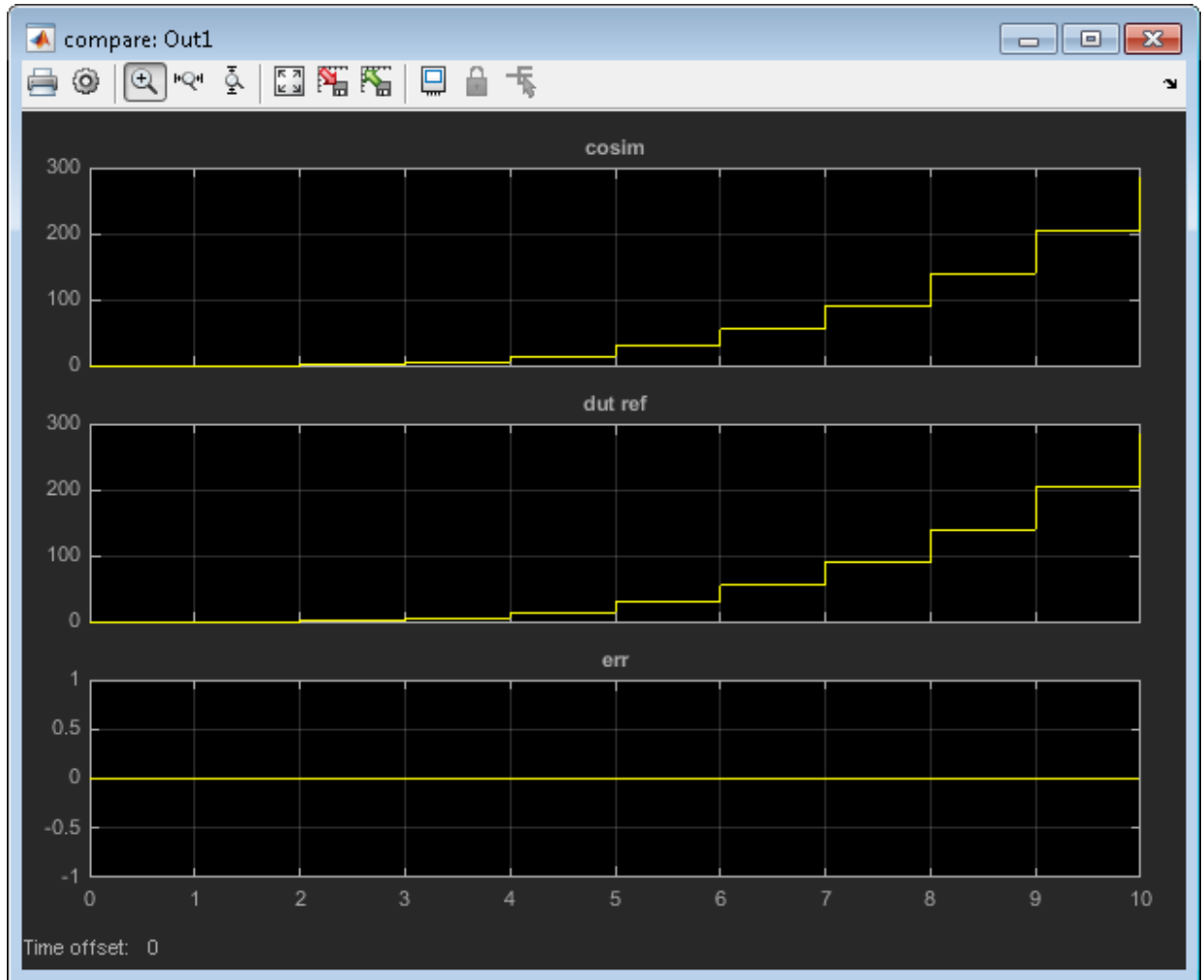
As the cosimulation runs, the HDL simulator displays messages like the following.

```
# 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
%
%  $N = (\text{ClockPeriod} / \text{DutBaseSampleTime}) * \text{OverClockFactor}$ 
% 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

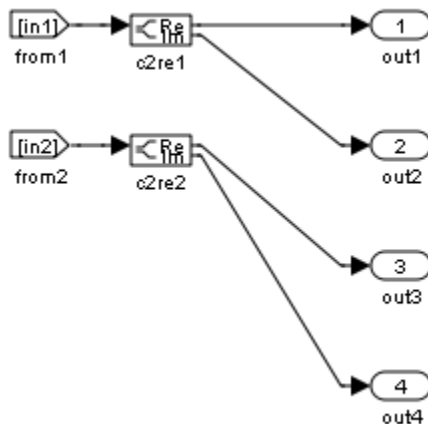
```

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

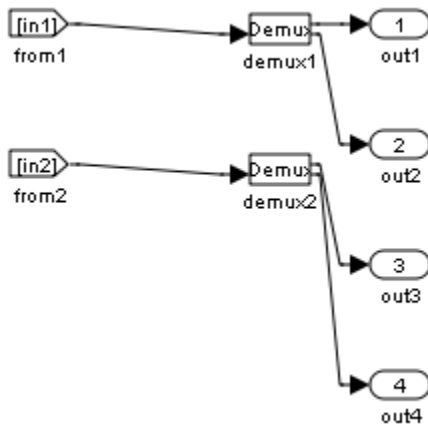
The generated cosimulation model automatically breaks complex inputs into real and imaginary parts. The following figure shows a FromCosimSrc subsystem that receives two complex input signals. The subsystem breaks the inputs into real and imaginary parts before passing them to the subsystem outputs.



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.

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'` and `'in'`.
- *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 **Enable 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 21-22). 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.

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: <ul style="list-style-type: none"> • 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 27-63

“State Control Block Modes” on page 27-63

“Synchronous Badge for Subsystems by Using Synchronous Mode” on page 27-64

“Generate HDL Code with the State Control Block” on page 27-66

“Enable and Reset Hardware Simulation Behavior” on page 27-68

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.

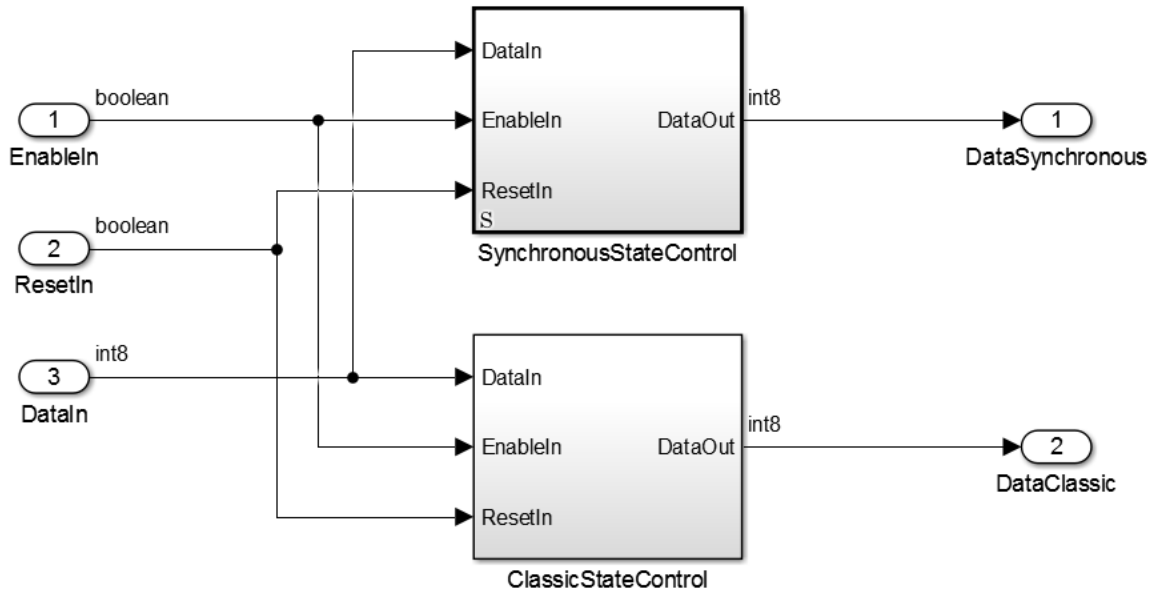
Functionality	Synchronous mode	Classic mode
<p>Simulink simulation behavior</p> <ul style="list-style-type: none"> • Initialize method: Initializes states. • Update method: Updates states. • Output method: Computes output values. 	<p>The update method only updates states. The output method computes the output values at each time step.</p> <p>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.</p>	<p>The update method updates states and computes the output values.</p> <p>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.</p>
HDL simulation behavior	More efficient on hardware.	Less efficient on hardware.
HDL code generation behavior	<p>Generated HDL code is cleaner and uses fewer resources on hardware.</p> <p>For example, when you have enabled subsystems, HDL Coder does not generate bypass registers for each state update and uses fewer hardware resources.</p>	<p>Generated HDL code is not as clean and uses more hardware resources.</p> <p>For example, when you have enabled subsystems, HDL Coder generates bypass registers for each state update and uses more resources.</p>

To learn more about when you can use the State Control block, see State Control.

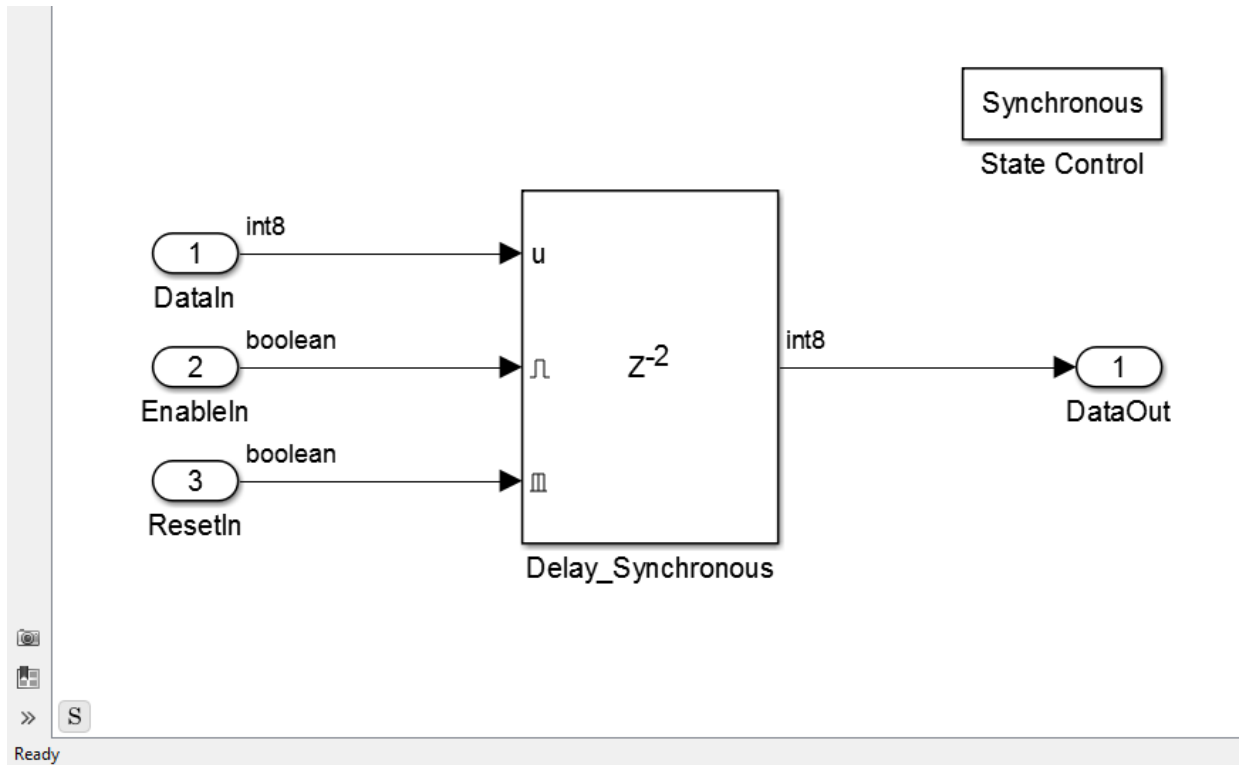
Synchronous Badge for Subsystems by Using Synchronous Mode

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

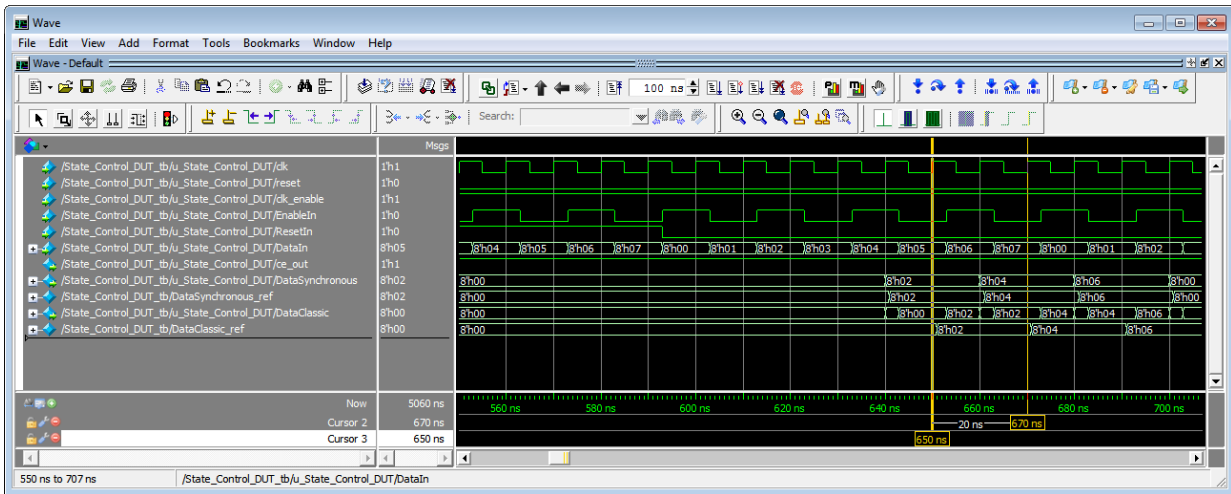
The following table shows a comparison of the HDL code generated from the Delay block for Classic and Synchronous modes of the State Control block.

Functionality	Synchronous mode	Classic mode
HDL code generation. Settings applied: <ul style="list-style-type: none"> • Language: Verilog • Reset type: Synchronous 	<pre> `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; input EnableIn; input ResetIn; output signed [7:0] DataOut; reg signed [7:0] Delay_Synchronous_reg; wire signed [7:0] Delay_Synchronous_reg_next[0]; wire signed [7:0] Delay_Synchronous_reg_next[1]; always @(posedge clk) begin : Delay_Synchronous_process if (reset == 1'b1 ResetIn == 1'b1) begin Delay_Synchronous_reg[0] always 8'b00000000; Delay_Synchronous_reg[1] <= 8'b00000000; 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 </pre>	<pre> `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 [0:1] Delay_Synchronous_bypass; reg signed [0:1] Delay_Synchronous_reg; reg signed [0:1] Delay_Synchronous_reg_next[0]; reg signed [0:1] Delay_Synchronous_reg_next[1]; wire signed [7:0] Delay_Synchronous_reg; wire signed [7:0] Delay_Synchronous_reg_next[0]; wire signed [7:0] Delay_Synchronous_reg_next[1]; always @(posedge clk) begin : Delay_Synchronous_process if (reset == 1'b1 ResetIn == 1'b1) begin Delay_Synchronous_bypass <= 8'sb0; Delay_Synchronous_reg[0] <= 8'sb0; Delay_Synchronous_reg[1] <= 8'sb0; end else begin if (enb && EnableIn) begin Delay_Synchronous_bypass <= 8'sb1; Delay_Synchronous_reg[0] <= Delay_Synchronous_reg_next[0]; Delay_Synchronous_reg[1] <= Delay_Synchronous_reg_next[1]; end end end </pre>

Functionality	Synchronous mode	Classic mode
	<pre> 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]; end assign DataOut = Delay_Synchronous_out1; Delay_Synchronous_reg[1]); assign Delay_Synchronous_out1 = (Enable endmodule // SynchronousStateControl </pre> <ul style="list-style-type: none"> • 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. 	<pre> Delay_Synchronous_bypass <= De Delay_Synchronous_reg[0] <= De Delay_Synchronous_reg[1] <= De endDataIn; Delay_Synchronous_reg[0]; end assign Delay_Synchronous_delay_out = (Synchronous_reg[1]); assign Delay_Synchronous_out1 = (Enabl Delay_Synchronous_bypass); assign Delay_Synchronous_bypass_next = assign Delay_Synchronous_reg_next[0] = assign Delay_Synchronous_reg_next[1] = assign DataOut = Delay_Synchronous_out endmodule // ClassicStateControl </pre> <ul style="list-style-type: none"> • Generated HDL code is less cleaner and requires more hardware resources as HDL Coder generates bypass registers. • The update method updates states and computes the output values.

Enable and Reset Hardware Simulation Behavior

Refer to the above Simulink model that shows a Delay block that uses Classic and Synchronous modes of the State Control block. The following diagram shows the ModelSim simulation behavior for the Delay block.



- When **ResetIn** signal is high, the **DataClassic** and **DataSynchronous** signals produce the same output.
- When both **ResetIn** and **EnableIn** signals are low, the **DataSynchronous** signal holds its value and changes only when the **EnableIn** signal becomes high at the next active clock edge. The **DataClassic** signal values change when the **EnableIn** signal is low as it processes new input values. The **DataClassic** signal values match the **DataSynchronous** signal values when the **EnableIn** becomes high.

For information about how to generate HDL code and simulate your design in ModelSim, see “Generate HDL Code from Simulink Model”.

See Also

State Control

Stateflow HDL Code Generation Support

- “Introduction to Stateflow HDL Code Generation” on page 28-2
- “Hardware Realization of Stateflow Semantics” on page 28-8
- “Generate HDL for Mealy and Moore Finite State Machines” on page 28-9
- “Design Patterns Using Advanced Chart Features” on page 28-17
- “Initialize Persistent Variables in MATLAB Functions” on page 28-28

Introduction to Stateflow HDL Code Generation

In this section...
“Overview” on page 28-2
“Comments” on page 28-2
“Tunable Parameters” on page 28-3
“Example” on page 28-3
“Restrictions” on page 28-3

Overview

Stateflow charts provide concise descriptions of complex system behavior 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 or Verilog) that implements the design embodied in the model. You can simulate and synthesize generated HDL code using industry standard tools, and then map your system designs into FPGAs and ASICs.

In general, generation of VHDL or Verilog 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 that is consistent with HDL. This broad subset lets you generate HDL code from existing models without significant remodeling effort.
- Generate bit-true, cycle-accurate HDL code that is fully compatible with Stateflow simulation semantics.

Comments

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.

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 10-23.

Example

The `hdlcodercfir` model shows how to generate HDL code for a subsystem that includes Stateflow charts.

To open the model, at the command line, enter:

```
hdlcodercfir
```

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.

However, 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 the algorithm by using MATLAB Function on page 10-97 blocks or other “Simulink Blocks Supported with Native Floating-Point” on page 10-106.

Chart Initialization

You must enable the chart property **Execute (enter) Chart at Initialization**. This option executes the update chart function immediately following chart initialization. The option is required for HDL because outputs must be available at time 0 (hardware reset). “Execution of a Chart at Initialization” (Stateflow) describes existing restrictions under this property.

The reset action must not entail the delay of combinatorial logic. Therefore, do not perform arithmetic in initialization actions.

To generate HDL code that is more readable and has better synthesis results, enable the **Initialize Outputs Every Time Chart Wakes Up** chart property. If you use a Moore state machine, HDL Coder generates an error if you disable the chart property.

If you disable **Initialize Outputs Every Time Chart Wakes Up**, the chart output is persistent, so the generated HDL code must internally register the output values.

Imported Code

A chart intended for HDL code generation must be entirely self-contained. The following 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 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 (via 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 are not supported.

Input and Output Events

HDL Coder supports the use of input and output events with Stateflow charts, subject to the following 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 using a graphical function.

Other 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.

- Do not define local events in a chart from which HDL code is generated.

Do not use the following implicit events:

- `enter`
- `exit`
- `change`

You can use the following implicit events:

- `wakeup`
- `tick`

You can use temporal logic if the base events are limited to these types of implicit events.

Note Absolute-time temporal logic is not supported for HDL code generation.

- Do not use recursion through graphical functions. HDL Coder does not currently 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. However, 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

[Sequence Viewer](#) | [State Transition Table](#) | [Truth Table](#)

Related Examples

- “Generate HDL for Mealy and Moore Finite State Machines” on page 28-9
- “Design Patterns Using Advanced Chart Features” on page 28-17

More About

- “Hardware Realization of Stateflow Semantics” on page 28-8

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

[Sequence Viewer](#) | [State Transition Table](#) | [Truth Table](#)

Related Examples

- “Generate HDL for Mealy and Moore Finite State Machines” on page 28-9
- “Design Patterns Using Advanced Chart Features” on page 28-17

More About

- “Introduction to Stateflow HDL Code Generation” on page 28-2

Generate HDL for Mealy and Moore Finite State Machines

In this section...

“Overview” on page 28-9

“Generating HDL Code for a Moore Finite State Machine” on page 28-9

“Generating HDL for a Mealy Finite State Machine” on page 28-13

Overview

Stateflow charts support modeling of three types of state machines:

- Classic (default)
- Mealy
- Moore

Mealy and Moore state machines differ in the following ways:

- The outputs of a Mealy state machine are a function of the current state and inputs.
- The outputs of a Moore state machine are 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 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 Chart.

Open the `hdlcoder_fsm_mealy_moore` model for an example that shows how to model Mealy and Moore charts.

Generating HDL Code for a Moore Finite State Machine

When generating HDL code for a chart that models a Moore state machine:

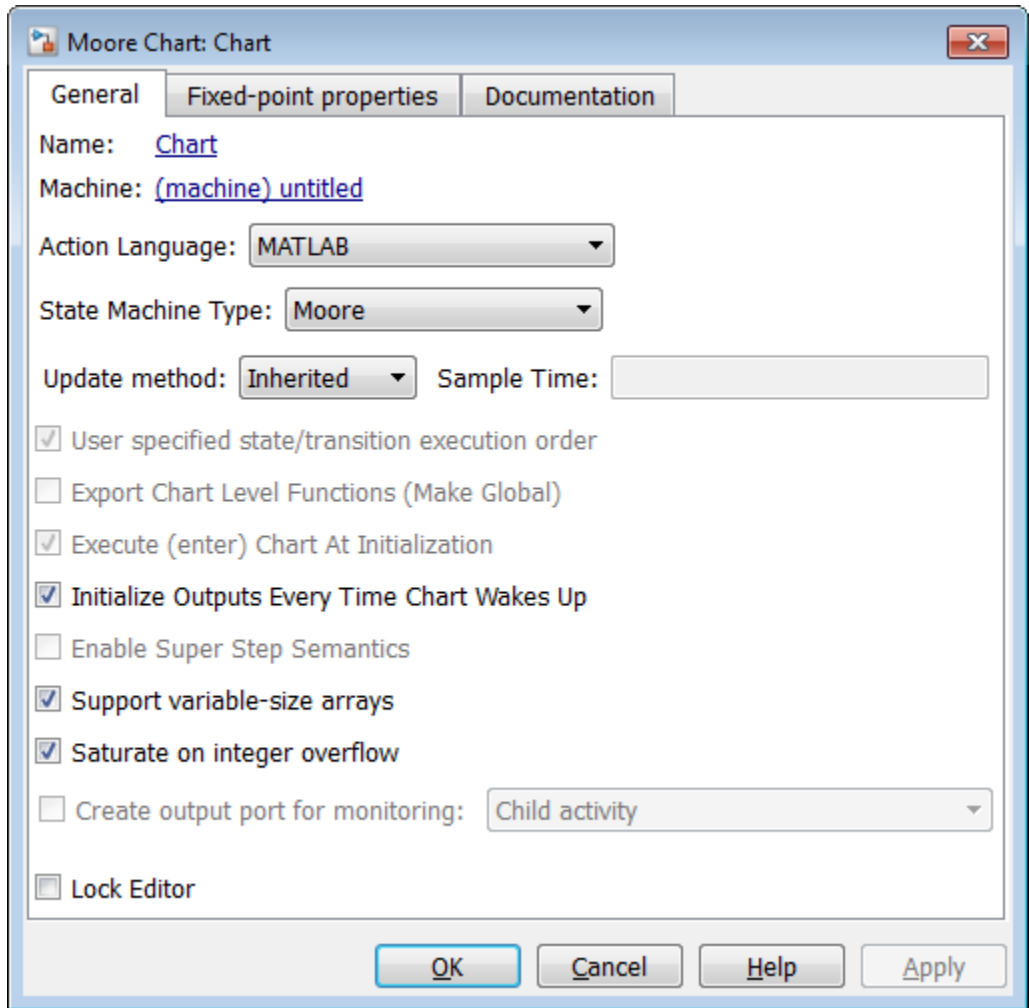
- The chart must meet the general code generation requirements as described in Chart.
- Actions must occur in states only. These actions must be unlabeled.

Moore actions must be associated with states, because output computation must be dependent only on states, not input. The configuration of active states at time step t determines output. If state S is active when a chart wakes up at time t , it contributes to the output whether or not it remains active into time $t+1$.

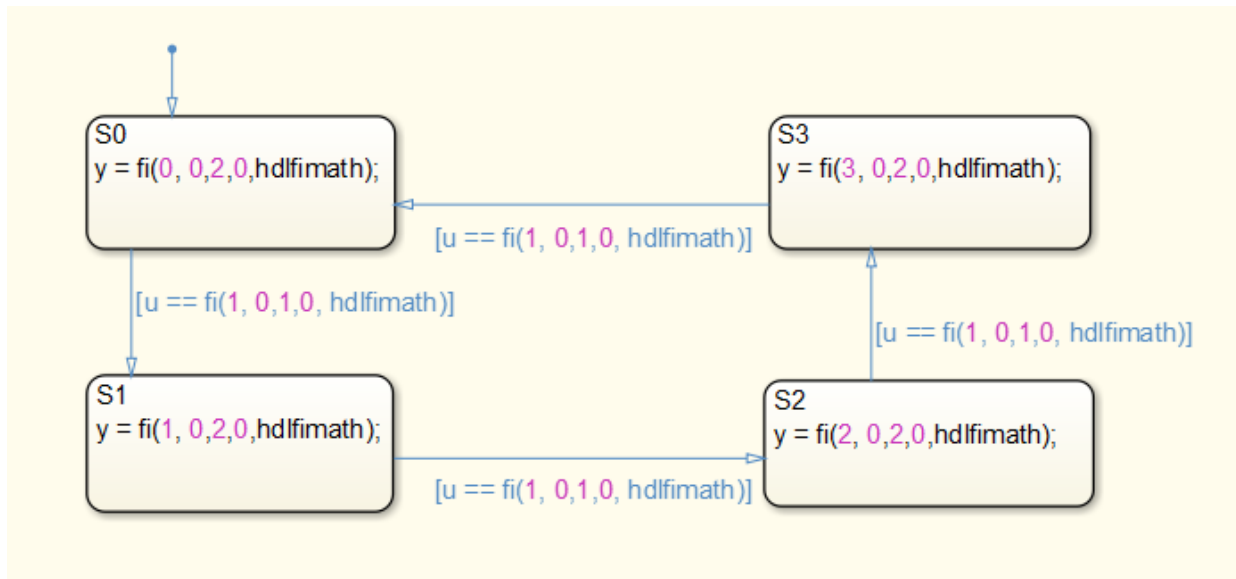
- Do not call Simulink functions.

This prevents output from depending on input in ways that would be difficult for the HDL code generator to verify.

- Make sure that you enable the chart property **Initialize Outputs Every Time Chart Wakes Up** as shown in the figure.



The following figure shows a Stateflow chart of a Moore state machine that uses MATLAB as the action language.



The Verilog code generated for the Moore chart:

```

always @(posedge clk or posedge reset)
begin : Moore_Chart_1_process
  if (reset == 1'b1) begin
    is_Moore_Chart <= is_Moore_Chart_IN_S0;
  end
  else begin
    if (enb) begin
      case ( is_Moore_Chart )
        is_Moore_Chart_IN_S0 :
          begin
            if (u == 8'sb00000001) begin
              is_Moore_Chart <= is_Moore_Chart_IN_S1;
            end
          end
        is_Moore_Chart_IN_S1 :
          begin
            if (u == 8'sb00000001) begin
              is_Moore_Chart <= is_Moore_Chart_IN_S2;
            end
          end
        is_Moore_Chart_IN_S2 :
          begin
            if (u == 8'sb00000001) begin
              is_Moore_Chart <= is_Moore_Chart_IN_S3;
            end
          end
        default :
          begin
            if (u == 8'sb00000001) begin
              is_Moore_Chart <= is_Moore_Chart_IN_S0;
            end
          end
      endcase
    end
  end
end
  
```

```

        end
      end
    endcase
  end
end
end

always @(is_Moore_Chart) begin
  y_1 = 2'b00;
  case ( is_Moore_Chart)
    is_Moore_Chart_IN_S0 :
      begin
        y_1 = 2'b00;
      end
    is_Moore_Chart_IN_S1 :
      begin
        y_1 = 2'b01;
      end
    is_Moore_Chart_IN_S2 :
      begin
        y_1 = 2'b10;
      end
    default :
      begin
        y_1 = 2'b11;
      end
  endcase
end

assign y = y_1;

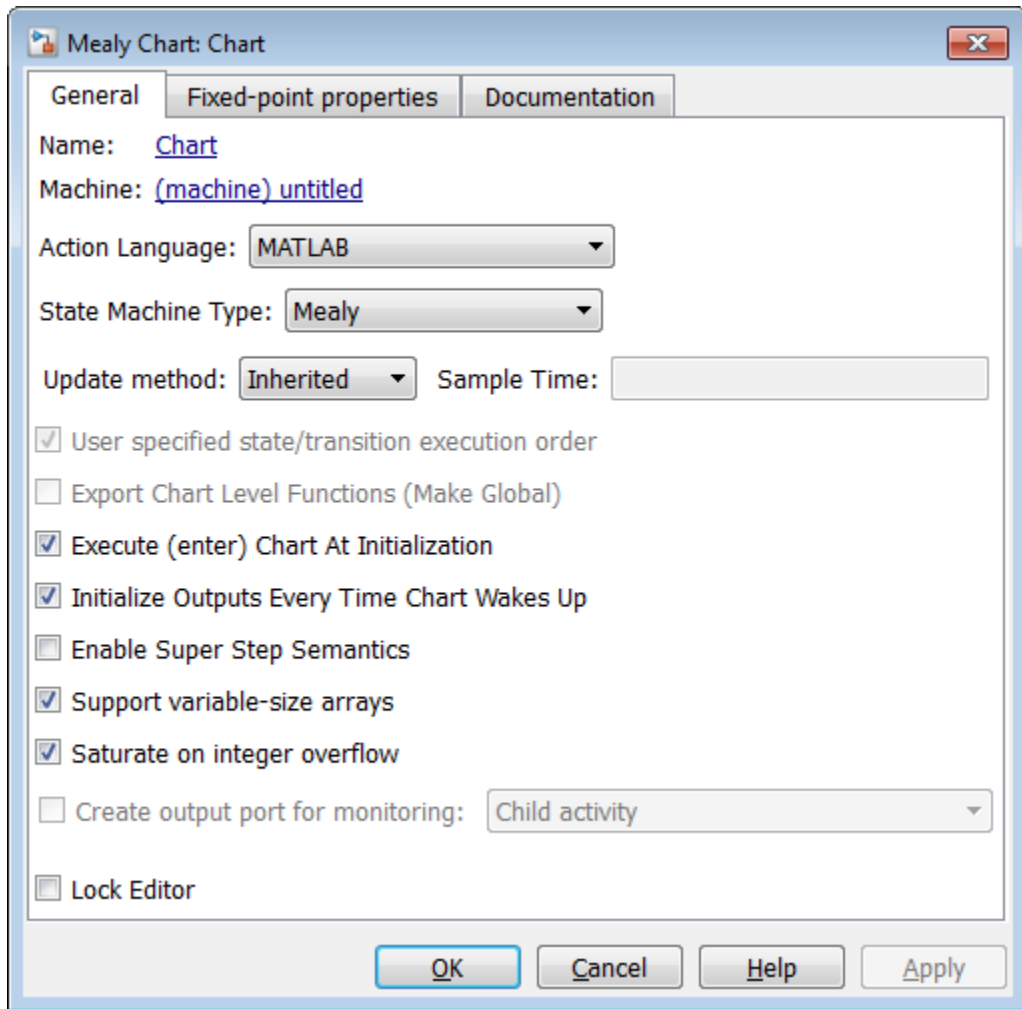
```

For an example that shows Mealy and Moore state machines that are appropriate for HDL code generation, open the `hdlcoder_fsm_mealy_moore` model.

Generating HDL for a Mealy Finite State Machine

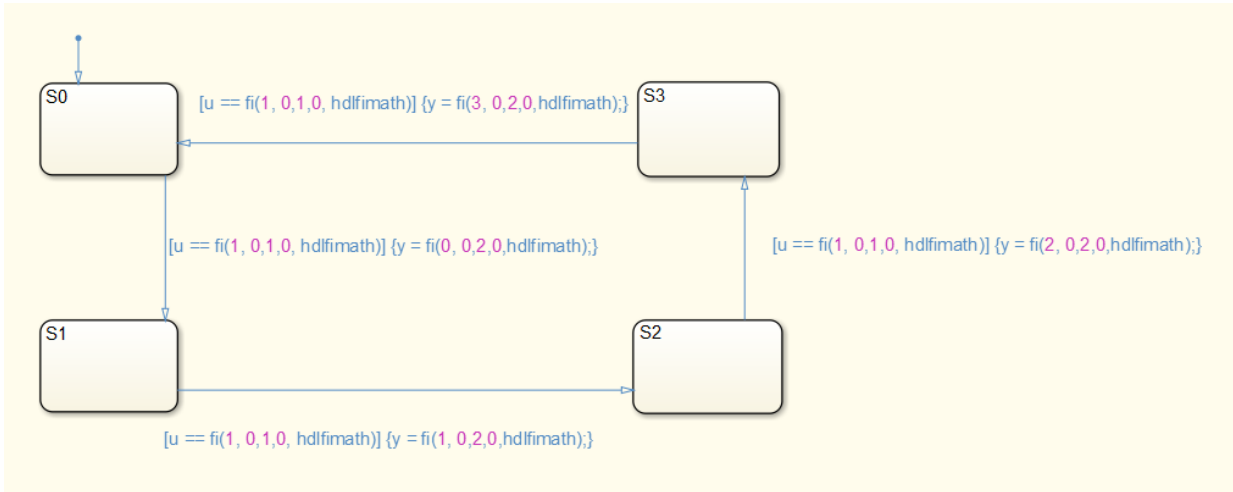
When generating HDL code for a chart that models a Mealy state machine:

- The chart must meet the general code generation requirements as described in Chart.
- Actions must be associated with inner and outer transitions only.
- For better synthesis results and more readable HDL code, we recommend enabling the chart property **Initialize Outputs Every Time Chart Wakes Up**, as shown in the following figure. If you disable **Initialize Outputs Every Time Chart Wakes Up**, the chart output is persistent, so the generated HDL code must internally register the output values.



Mealy actions are associated with transitions. In Mealy machines, output computation is expected to be driven by the change on inputs. In fact, the dependence of output on input is the fundamental distinguishing factor between the formal definitions of Mealy and Moore machines. The requirement that actions be given on transitions is to some degree stylistic, rather than required, to enforce Mealy semantics. However, it is natural that output computation follows input conditions on input, because transition conditions are primarily input conditions in any machine type.

The following figure shows an example of a chart that models a Mealy state machine using MATLAB as the action language.



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
  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;
```

For an example that shows Mealy and Moore state machines that are appropriate for HDL code generation, open the `hdlcoder_fsm_mealy_moore` model.

See Also

Chart

More About

- “Design Patterns Using Advanced Chart Features” on page 28-17
- “Introduction to Stateflow HDL Code Generation” on page 28-2
- “Hardware Realization of Stateflow Semantics” on page 28-8

Design Patterns Using Advanced Chart Features

In this section...

“Temporal Logic” on page 28-17

“Graphical Function” on page 28-19

“Hierarchy and Parallelism” on page 28-21

“Stateless Charts” on page 28-21

“Truth Tables” on page 28-23

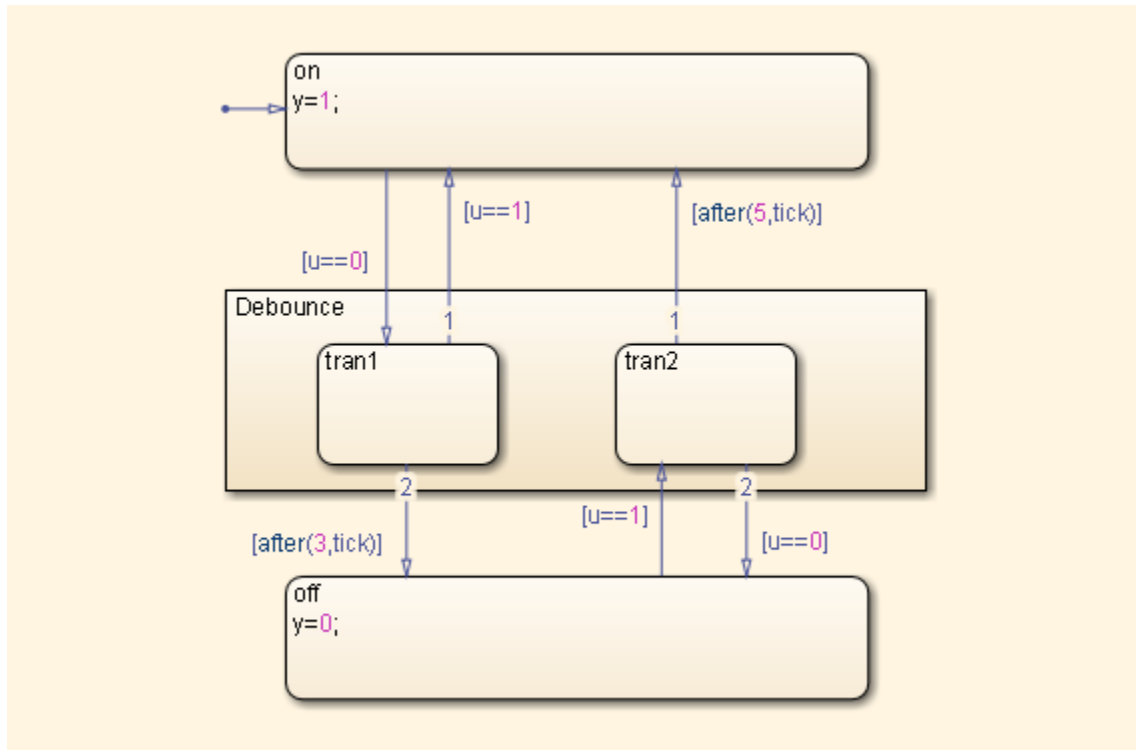
Temporal Logic

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.

Note Absolute-time temporal logic is not supported for HDL code generation.

For detailed information about temporal logic, 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_il_temp >= 3 THEN
      is_Chart_next <= IN_off;
      y_reg_next <= 0.0;
    END IF;
  WHEN IN_tran2 =>
    IF temporalCounter_il_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_il_temp := to_unsigned(0, 8);
    END IF;
  WHEN OTHERS =>
    IF u = 0.0 THEN
      is_Chart_next <= IN_tran1;
      temporalCounter_il_temp := to_unsigned(0, 8);
    END IF;
END CASE;

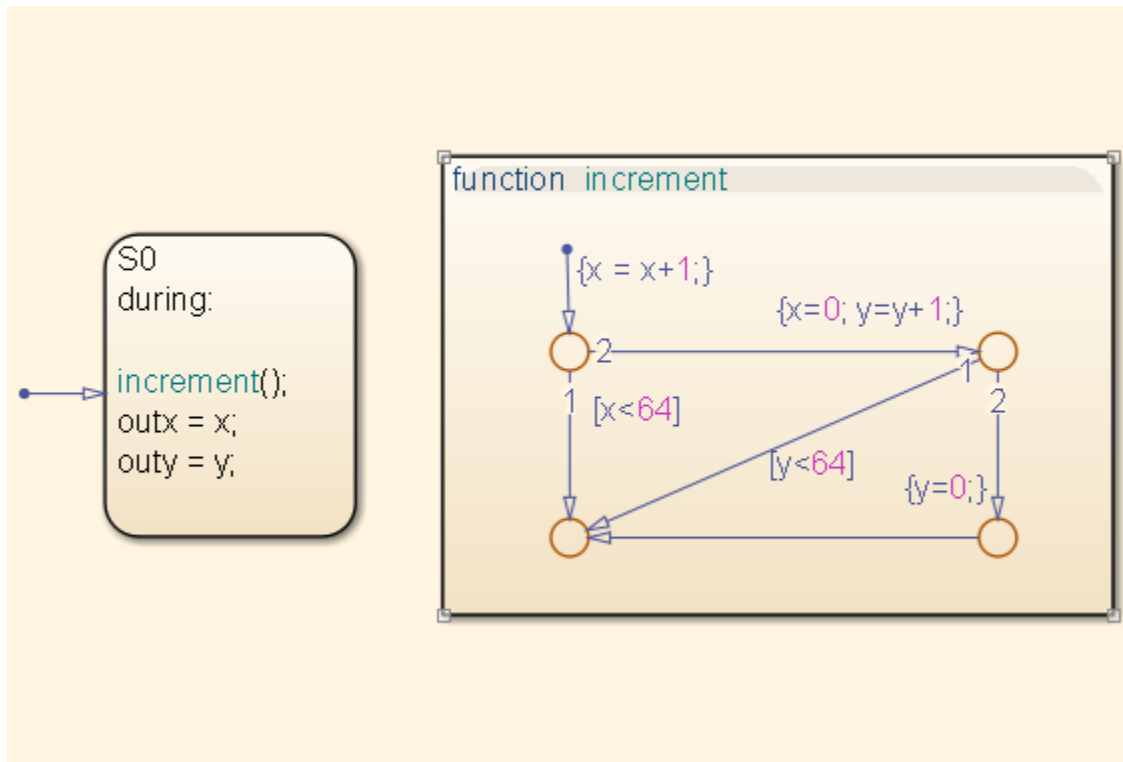
temporalCounter_il_next <= temporalCounter_il_temp;
END PROCESS Chart_1_output;

```

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 DOWNT0 0);
  VARIABLE y_temp : unsigned(7 DOWNT0 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 “Hierarchy 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{

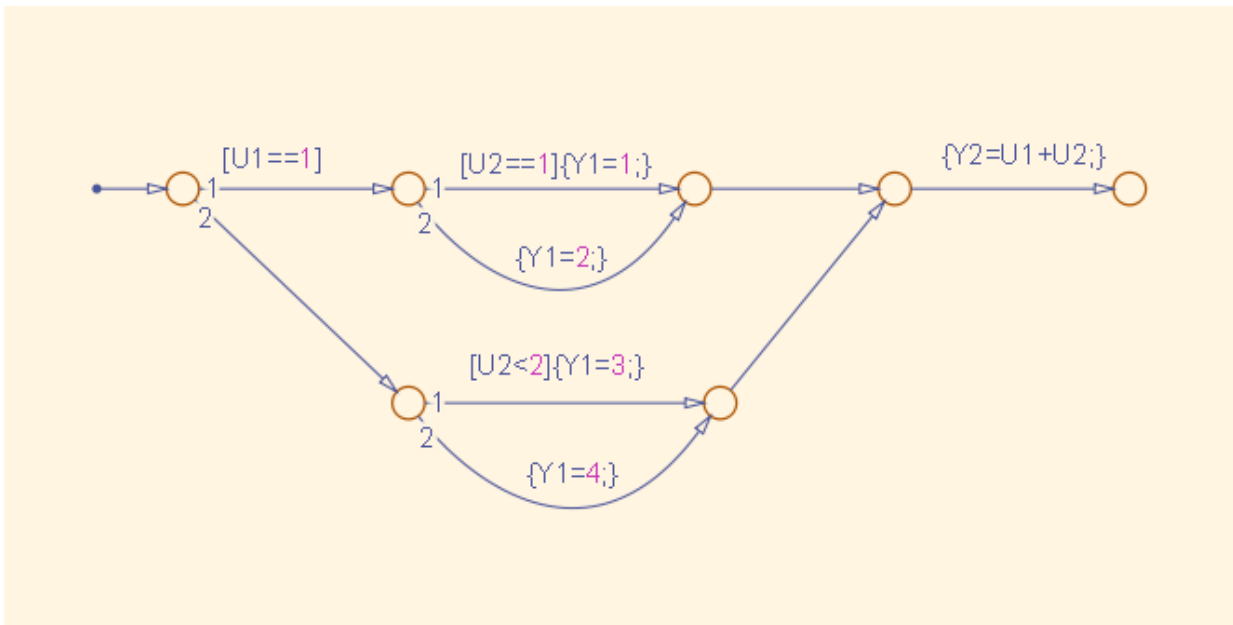
```

```

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;
    END IF;
  END PROCESS;

```

```

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:

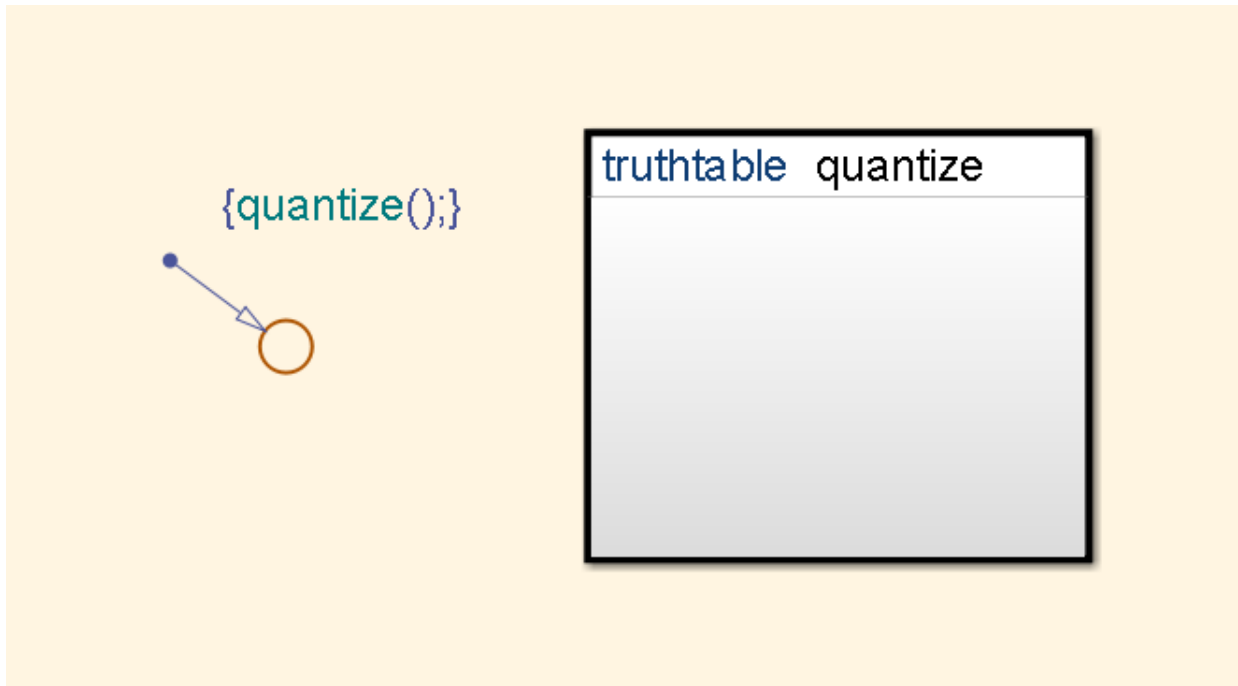
```

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		$U \leq 10$	T	-	-	-	-
2		$U \leq 17$	-	T	-	-	-
3		$U \leq 45$	-	-	T	-	-
4		$U \leq 52$	-	-	-	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		$Y = 5$

28-25

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) or `Truth_Table.v` (for Verilog).

See Also

[Sequence Viewer](#) | [State Transition Table](#) | [Truth Table](#)

Related Examples

- “Generate HDL for Mealy and Moore Finite State Machines” on page 28-9

More About

- “Hardware Realization of Stateflow Semantics” on page 28-8
- “Introduction to Stateflow HDL Code Generation” on page 28-2

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. For code generation, functions must initialize a persistent variable if it is empty. For more information, see `persistent`.

When programming MATLAB functions in these situations:

- MATLAB Function blocks with no direct feedthrough
- MATLAB Function blocks in models that contain State Control blocks in Synchronous mode
- MATLAB functions in Stateflow charts that implement Moore machine semantics

The specialized semantics impact how a function initializes its persistent data. Because the initialization must be independent of the input to the function, follow these guidelines:

- The function initializes its persistent variables only by accessing constants.
- The control flow of the function does not depend on whether the initialization occurs.

For example, this function has a persistent variable `n`.

```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = u;
        y = 1;
        return
    end

    y = n;
    n = n + u;
end
```

This type of initialization results in an error because the initial value of `n` depends on the input `u` and the `return` statement interrupts the normal control flow of the function.

To correct the error, initialize the persistent variable by setting it to a constant value and remove the `return` statement. For example, this function initializes the persistent variable without producing an error.

```
function y = fcn(u)
    persistent n
```

```

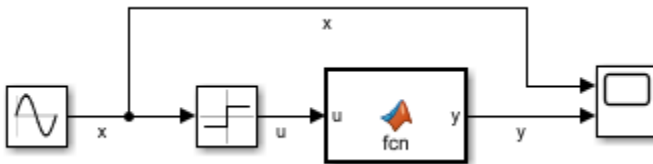
if isempty(n)
    n = 1;
end

y = n;
n = n + u;
end

```

MATLAB Function Block With No Direct Feedthrough

This model contains a MATLAB function block that defines the function `fcn`, described previously. The input `u` is a square wave with values of 1 and -1.

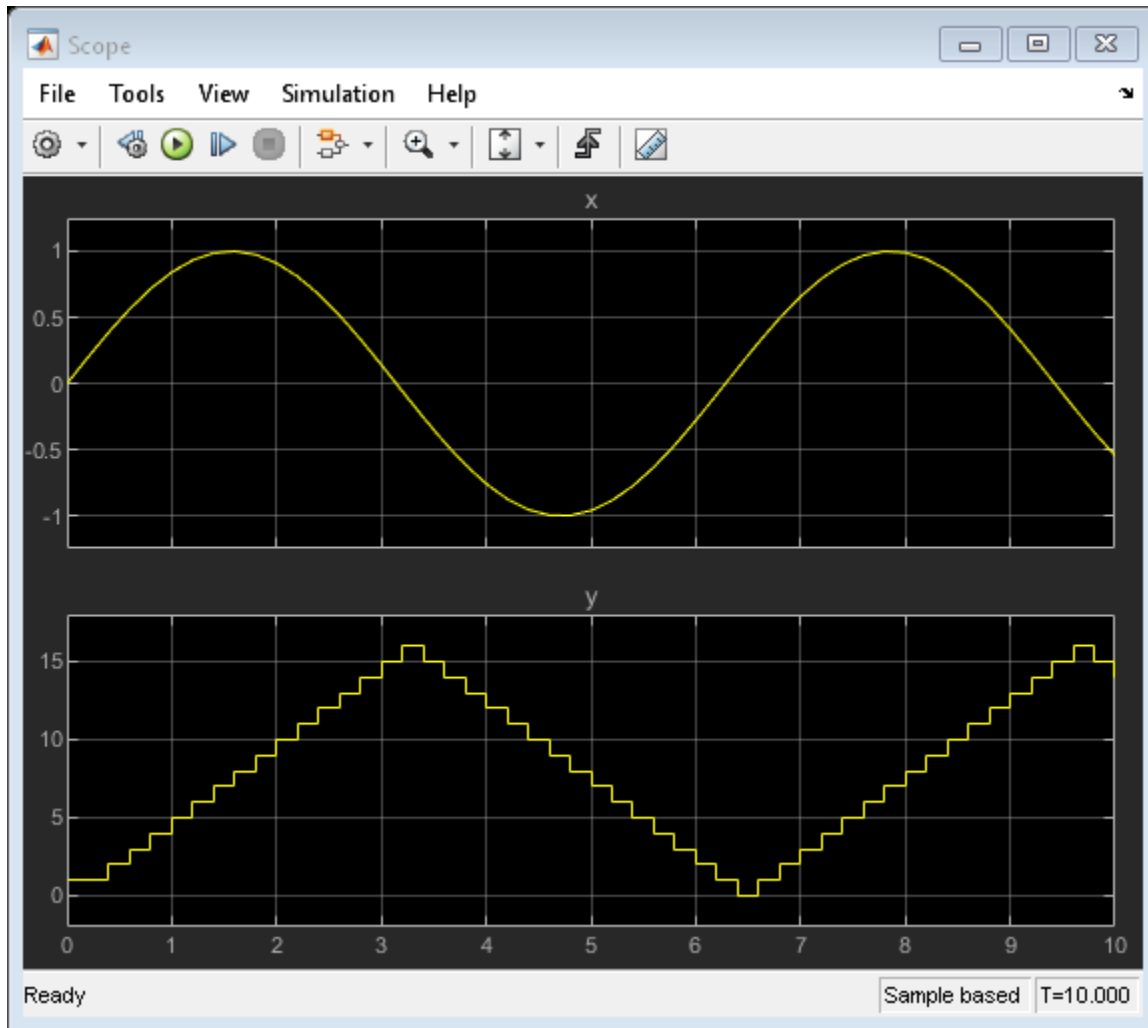


In the MATLAB function block:

- The initial value of the persistent variable `n` depends on the input `u`.
- The return statement interrupts the normal control flow of the function.

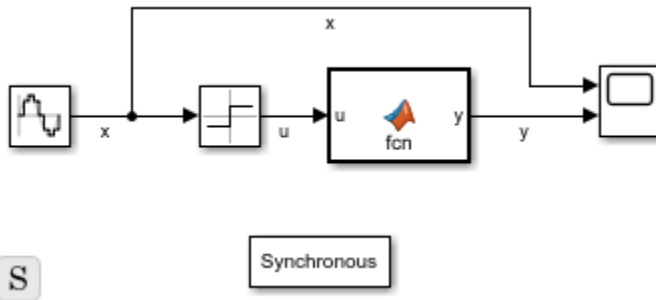
Because the `Allow direct feedthrough` check box is cleared, the initialization results in an error.

If you modify the function so it initializes `n` independently of the input, then you can simulate an error-free model.



State Control Block in Synchronous Mode

This model contains a MATLAB function block that defines the function `fcn`, described previously. The input `u` is a square wave with values of 1 and -1.

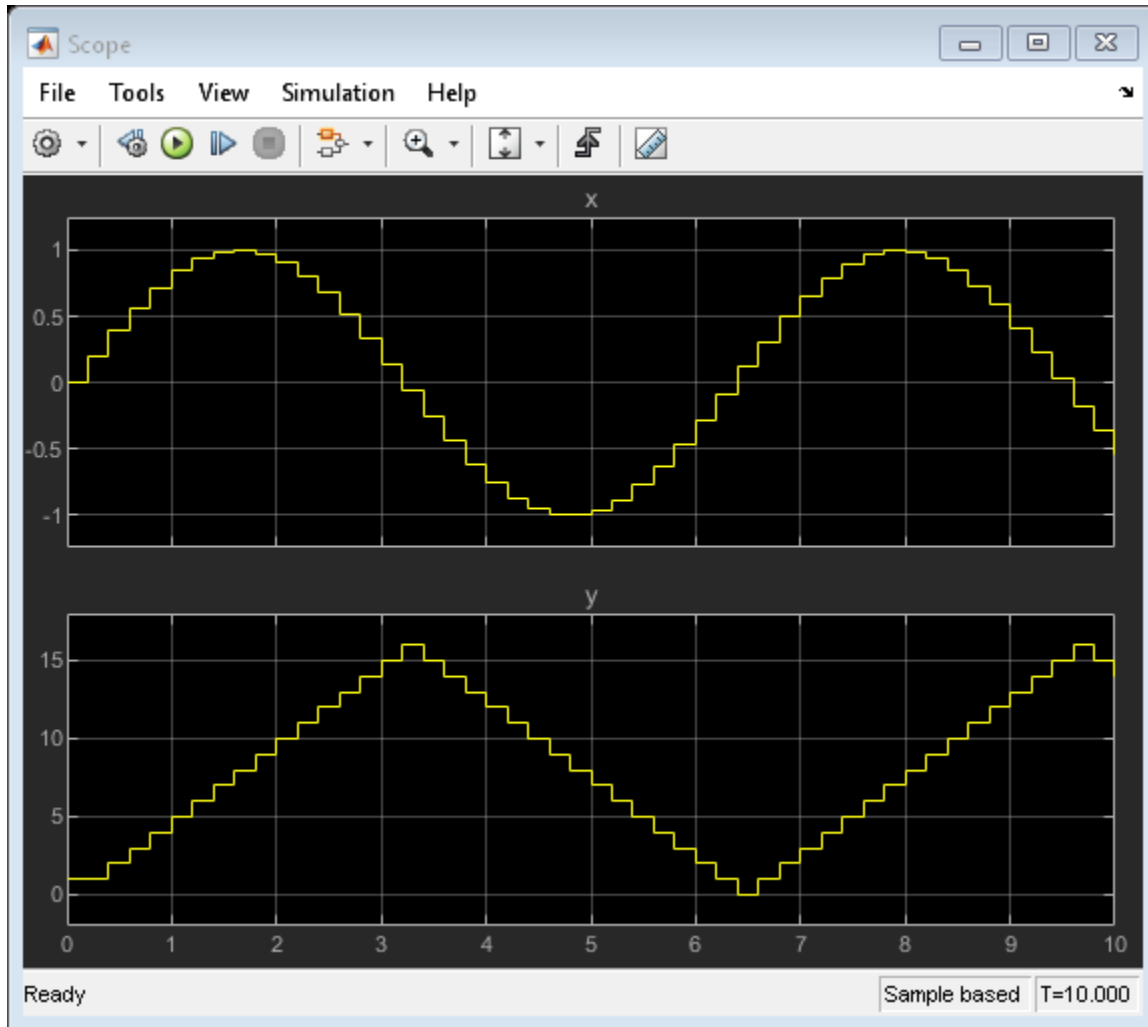


In the MATLAB function block:

- The initial value of the persistent variable n depends on the input u .
- The `return` statement interrupts the normal control flow of the function.

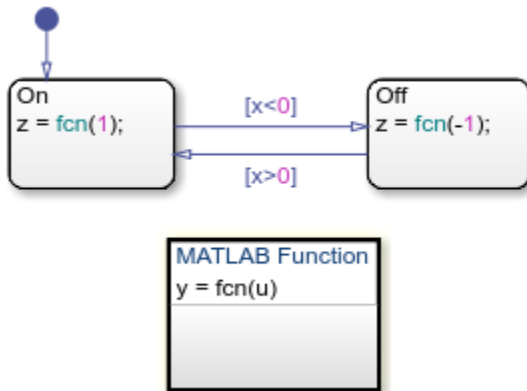
Because the model contains a State Control block in Synchronous mode, the initialization results in an error.

If you modify the function so it initializes n independently of the input, then you can simulate an error-free model.



Stateflow Chart Implementing Moore Semantics

This model contains a Stateflow chart that implements Moore machine semantics. The chart contains a MATLAB function that defines the function `fcn`, described previously. The input `u` has values of 1 and -1 that depend on the state of the chart.

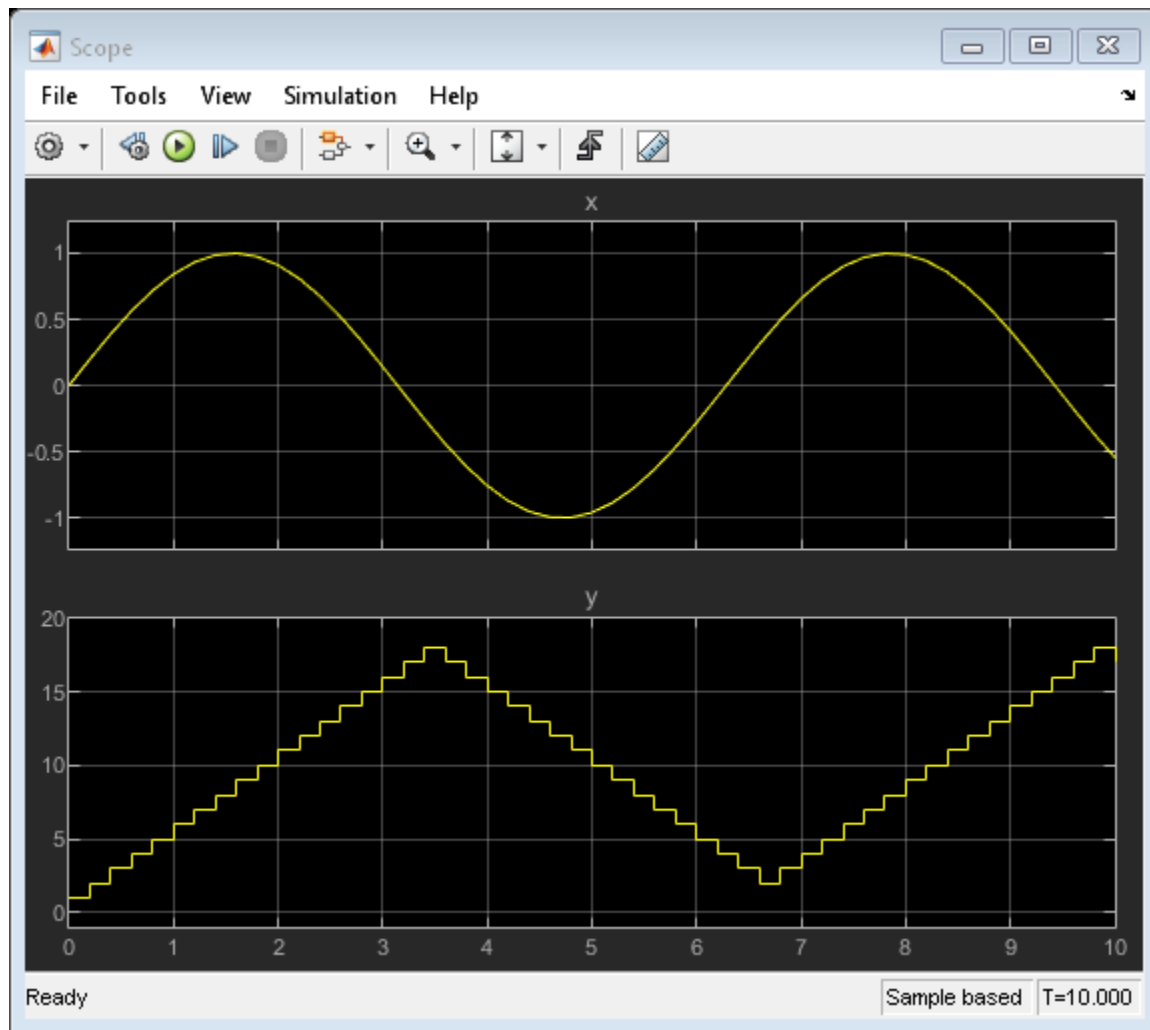


In the MATLAB function:

- The initial value of the persistent variable n depends on the input u .
- The `return` statement interrupts the normal control flow of the function.

Because the chart implements Moore semantics, the initialization results in an error.

If you modify the function so it initializes n independently of the input, then you can simulate an error-free model.



See Also

Chart | MATLAB Function | State Control | persistent

More About

- “Use Nondirect Feedthrough in a MATLAB Function Block” (Simulink)
- “Synchronous Subsystem Behavior with the State Control Block” on page 27-63
- “Design Considerations for Moore Charts” (Stateflow)

Generating HDL Code with the MATLAB Function Block

- “HDL Applications for the MATLAB Function Block” on page 29-2
- “Viterbi Decoder with the MATLAB Function Block” on page 29-4
- “Code Generation from a MATLAB Function Block” on page 29-5
- “Generate Instantiable Code for Functions” on page 29-21
- “MATLAB Function Block Design Patterns for HDL” on page 29-23
- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-39

HDL Applications for the MATLAB Function Block

In this section...
“Structure of Generated HDL Code” on page 29-2
“HDL Applications” on page 29-2

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 24-89.

See Also

“Check for MATLAB Function block settings” on page 38-20

More About

- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “Code Generation from a MATLAB Function Block” on page 29-5
- “MATLAB Function Block Design Patterns for HDL” on page 29-23
- “Generate DUT Ports for Tunable Parameters” on page 10-23

Viterbi Decoder with the MATLAB Function Block

hdlcoderviterbi2 models a Viterbi decoder, incorporating a MATLAB Function block for use in simulation and HDL code generation. To open the model, type the following at the MATLAB command prompt:

```
hdlcoderviterbi2
```

See Also

“Check for MATLAB Function block settings” on page 38-20

More About

- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “Code Generation from a MATLAB Function Block” on page 29-5
- “MATLAB Function Block Design Patterns for HDL” on page 29-23
- “Generate DUT Ports for Tunable Parameters” on page 10-23

Code Generation from a MATLAB Function Block

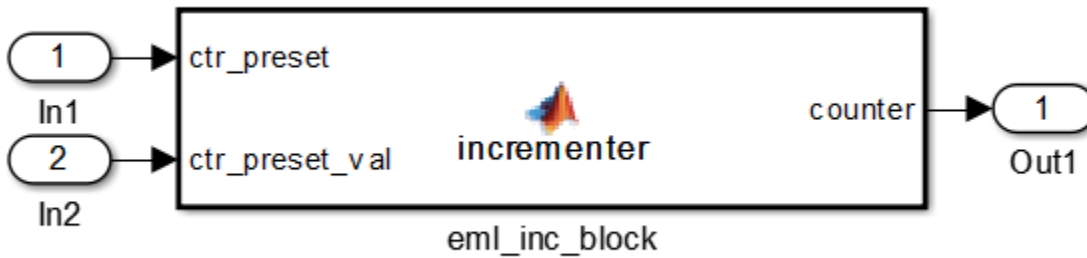
In this section...

“Counter Model Using the MATLAB Function block” on page 29-5
“Setting Up” on page 29-7
“Creating the Model and Configuring General Model Settings” on page 29-8
“Adding a MATLAB Function Block to the Model” on page 29-9
“Set Fixed-Point Options for the MATLAB Function Block” on page 29-9
“Programming the MATLAB Function Block” on page 29-13
“Constructing and Connecting the DUT_eML_Block Subsystem” on page 29-13
“Compiling the Model and Displaying Port Data Types” on page 29-16
“Simulating the eml_hdl_incrementer_tut Model” on page 29-17
“Generating HDL Code” on page 29-18

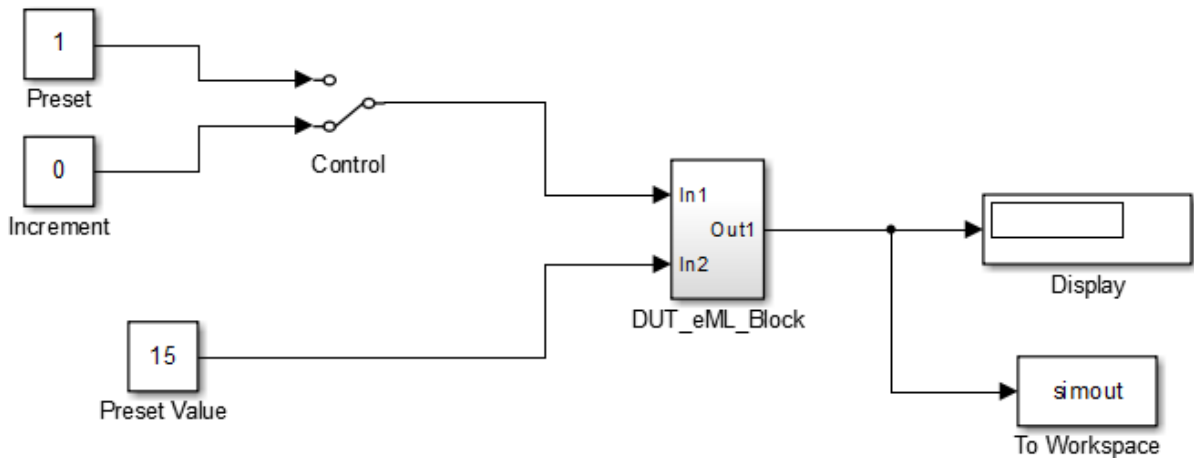
Counter Model Using the MATLAB Function block

In this tutorial, you construct and configure a simple model, `eml_hdl_incrementer_tut`, and then generate VHDL code from the model. `eml_hdl_incrementer_tut` includes a MATLAB Function block that implements a simple fixed-point counter function, `incrementer`. The `incrementer` function is invoked once during each sample period of the model. The function maintains a persistent variable `count`, which is either incremented or reinitialized to a preset value (`ctr_preset_val`), depending on the value passed in 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, `DUT_eML_Block`. The subsystem functions as the device under test (DUT) from which you generate HDL code.



The root-level model drives the subsystem and includes Display and To Workspace blocks for use in simulation. (The Display and To Workspace blocks do not generate HDL code.)



Tip If you do not want to construct the model step by step, or do not have time, you can open the completed model by entering the name at the command prompt:

eml_hdl_incrementer

After you open the model, save a copy of it to your local folder as
eml_hdl_incrementer_tut.

The Incrementer Function Code

The following code listing gives the complete incrementer function definition:

```
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);

function hdl_fi = getfi(val)

nt = numerictype(0,14,0);
fm = hdlfimath;
hdl_fi = fi(val, nt, fm);
```

Setting Up

Before you begin building the example model, set up a working folder for your model and generated code.

Setting Up a folder

- 1 Start MATLAB.
- 2 Create a folder named `eml_tut`, for example:

```
mkdir D:\work\eml_tut
```

The `eml_tut` folder stores the model you create, and also contains sub-folders and generated code. The location of the folder does not matter, except that it should not be within the MATLAB tree.

- 3 Make the `eml_tut` folder your working folder, for example:

```
cd D:\work\eml_tut
```

Creating the Model and Configuring General Model Settings

In this section, you create a model and set some parameters to values recommended for HDL code generation `hdlsetup` command. The `hdlsetup` command uses the `set_param` function to set up models for HDL code generation quickly and consistently. See “Customize `hdlsetup` Function Based on Target Application” on page 20-23 for further information about `hdlsetup`.

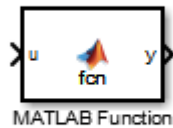
To set the model parameters:

- 1 Create a new model.
- 2 Save the model as `eml_hdl_incrementer_tut`.
- 3 At the MATLAB command prompt, type:

```
hdlsetup('eml_hdl_incrementer_tut');
```
- 4 Open the Configuration Parameters dialog box.
- 5 Set the following **Solver** options, which are useful in simulating this model:
 - **Fixed step size:** 1
 - **Stop time:** 5
- 6 Click **OK** to save your changes and close the Configuration Parameters dialog box.
- 7 Save your model.

Adding a MATLAB Function Block to the Model

- 1 Open the Simulink Library Browser. Then, select the Simulink/User-Defined Functions library.
- 2 Select the MATLAB Function block from the library window and add it to the model.



- 3 Change the block label from MATLAB Function to eml_inc_block.



- 4 Save the model.
- 5 Close the Simulink Library Browser.

Set Fixed-Point Options for the MATLAB Function Block

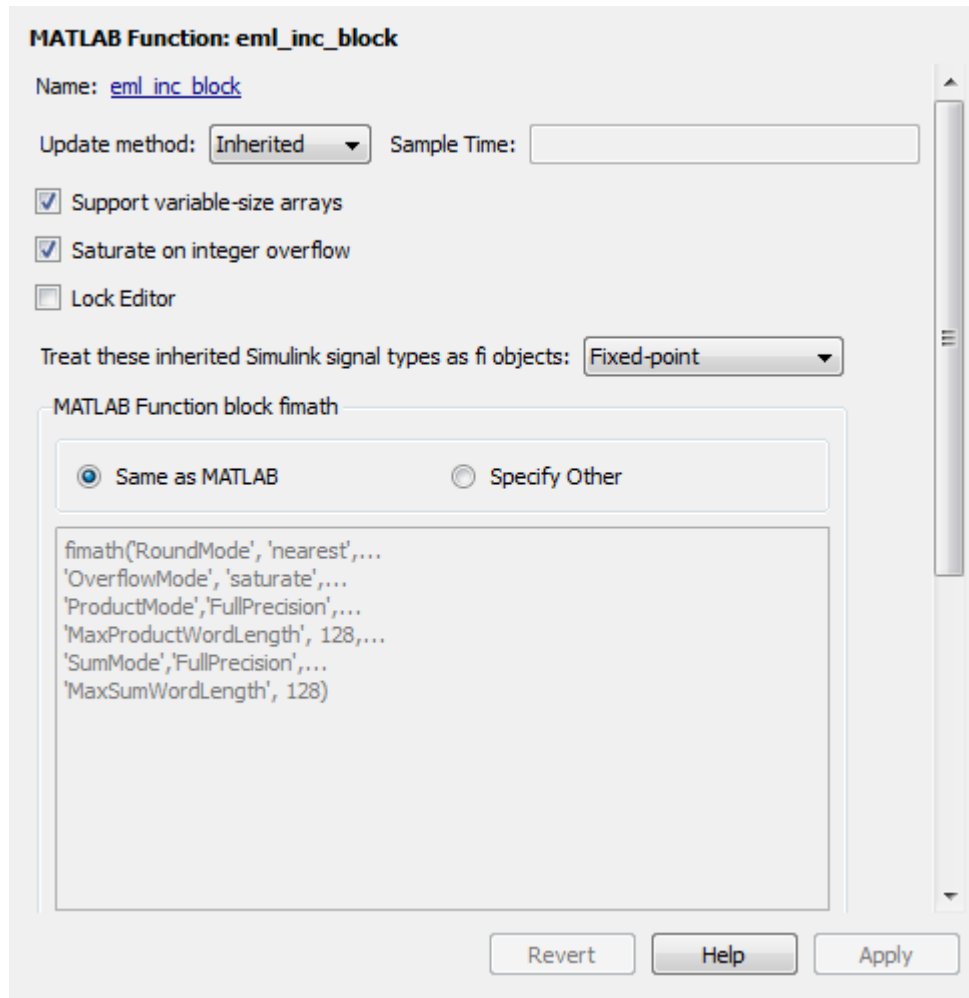
This section describes how to set up the `fimath` specification and other fixed-point options that are recommended for efficient HDL code generation from the MATLAB Function block. The recommended settings are:

- ProductMode property of the `fimath` specification: 'FullPrecision'

- SumMode property of the fimath specification: 'FullPrecision'
- **Treat these inherited signal types as fi objects** option: Fixed-point (This is the default setting.)

Configure the options as follows:

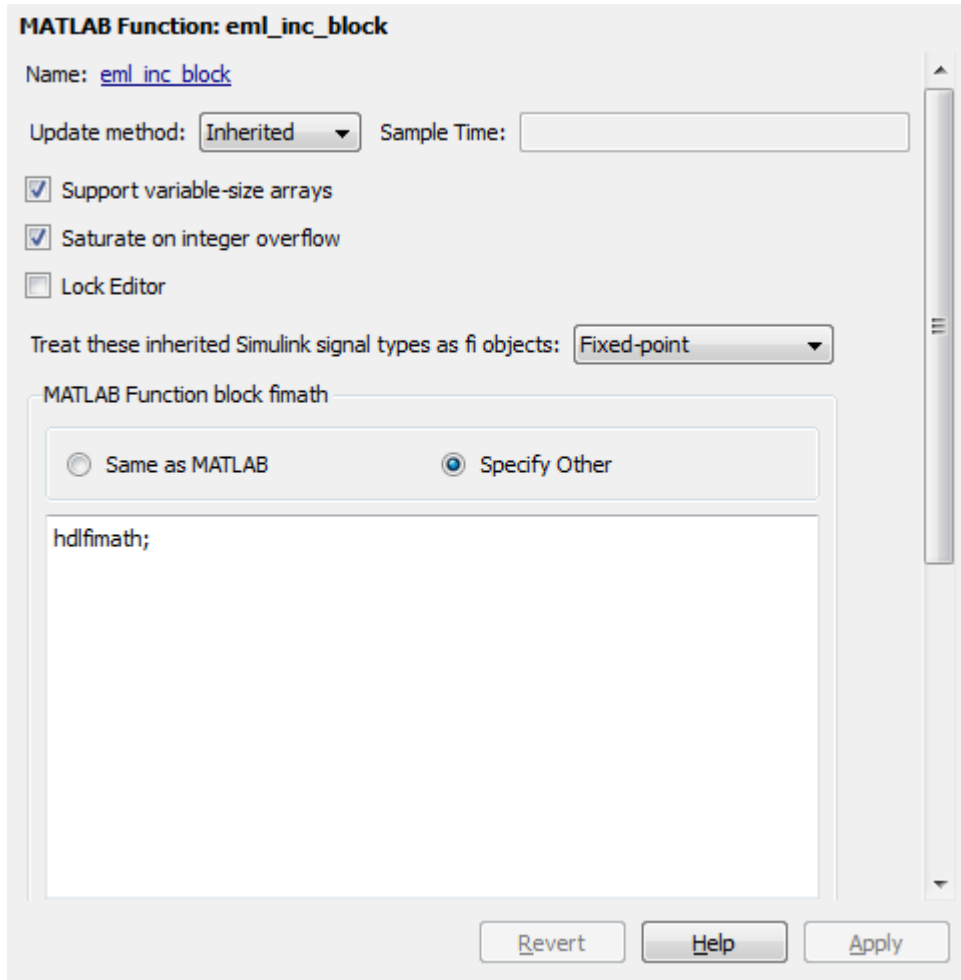
- 1** Open the `eml_hdl_incrementer_tut` model that you created in “Adding a MATLAB Function Block to the Model” on page 29-9.
- 2** Double-click the MATLAB Function block to open it for editing. The MATLAB Function Block Editor appears.
- 3** Click **Edit Data**. The Ports and Data Manager dialog box opens, displaying the default `fimath` specification and other properties for the MATLAB Function block.



- 4 Select **Specify Other**. Selecting this option enables the **MATLAB Function block `fimath`** text entry field.
- 5 The `hdlfimath` function is a utility that defines a FIMATH specification that is optimized for HDL code generation. Replace the default **MATLAB Function block `fimath`** specification with a call to `hdlfimath` as follows:

```
hdlfimath;
```

- 6 Click **Apply**. The MATLAB Function block properties should now appear as shown in the following figure.



- 7 Close the Ports and Data Manager.
- 8 Save the model.

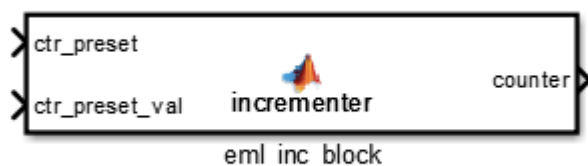
Programming the MATLAB Function Block

The next step is add code to the MATLAB Function block to define the `incrementer` function, and then use diagnostics to check for errors.

- 1 Open the `eml_hdl_incrementer_tut` model that you created in “Adding a MATLAB Function Block to the Model” on page 29-9.
- 2 Double-click the MATLAB Function block to open it for editing.
- 3 In the MATLAB Function Block Editor, delete the default code.
- 4 Copy the complete `incrementer` function definition from the listing given in “The Incrementer Function Code” on page 29-7, and paste it into the editor.
- 5 Save the model. Doing so updates the model window, redrawing the MATLAB Function block.

Changing the function header of the MATLAB Function block makes the following changes to the block icon:

- The function name in the middle 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.
- 6 Resize the block to make the port labels more legible.



- 7 Save the model again.

Constructing and Connecting the DUT_eML_Block Subsystem

This section assumes that you have completed “Programming the MATLAB Function Block” on page 29-13 without encountering an error. In this section, you construct a

subsystem containing the `incrementer` function block, to be used as the device under test (DUT) from which to generate HDL code. You then set the port data types and connect the subsystem ports to the model.

Constructing the DUT_eML_Block Subsystem

Construct a subsystem containing the `incrementer` function block as follows:

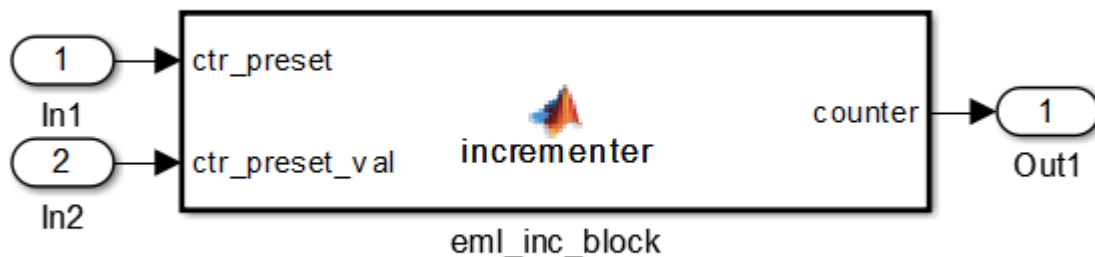
- 1 Click the `incrementer` function block.
- 2 Select **Diagram > Subsystem & Model Reference > Create Subsystem from Selection**.

A subsystem, labeled `Subsystem`, is created in the model window.

- 3 Change the `Subsystem` label to `DUT_eML_Block`.

Setting Port Data Types for the MATLAB Function Block

- 1 Double-click the subsystem to view its interior. As shown in the following figure, the subsystem contains the `incrementer` function block, with input and output ports connected.



- 2 Double-click the `incrementer` function block to open the MATLAB Function Block Editor.
- 3 In the editor, click **Edit Data** to open the Ports and Data Manager.
- 4 Select the `ctr_preset` entry in the port list on the left. Click the button labeled `>>` to display the Data Type Assistant. Set **Mode** for this port to `Built in`. Set **Data**

- type** to `boolean`. Click the button labeled `<<` to close the Data Type Assistant. Click **Apply**.
- 5 Select the `ctr_preset_val` entry in the port list on the left. Click the button labeled `>>` to display the Data Type Assistant. Set **Mode** for this port to `Fixed point`. Set **Signedness** to `Unsigned`. Set **Word length** to 14. Click the button labeled `<<` to close the Data Type Assistant. Click **Apply**.
 - 6 Select the `counter` entry in the port list on the left. Click the button labeled `>>` to display the Data Type Assistant. Verify that **Mode** for this port is set to `Inherit: Same as Simulink`. Click the button labeled `<<` to close the Data Type Assistant. Click **Apply**.
 - 7 Close the Ports and Data Manager dialog box and the MATLAB Function Block Editor.
 - 8 Save the model and close the `DUT_eML_Block` subsystem.

Connecting Subsystem Ports to the Model

Next, connect the ports of the `DUT_eML_Block` subsystem to the model as follows:

- 1 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`.
- 2 Make a copy of the `Preset Constant` block. Set its value to 0, and change its block label to `Increment`.
- 3 From the Signal Routing library, add a Manual Switch block to the model. Change its label to `Control`. Connect its output to the `In1` port of the `DUT_eML_Block` subsystem.
- 4 Connect the `Preset Constant` block to the upper input of the `Control` switch block. Connect the `Increment Constant` block to the lower input of the `Control` switch block.
- 5 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`.
- 6 Connect the `Preset Value Constant` block to the `In2` port of the `DUT_eML_Block` subsystem.
- 7 From the Sinks library, add a Display block to the model. Connect it to the `Out1` port of the `DUT_eML_Block` subsystem.
- 8 From the Sinks library, add a To Workspace block to the model. Route the output signal from the `DUT_eML_Block` subsystem to the To Workspace block.

- 9 Save the model.

Checking the Function for Errors

Use the built-in diagnostics of MATLAB Function blocks to test for syntax errors:

- 1 Open the `eml_hdl_incrementer_tut` model.
- 2 Double-click the MATLAB Function block `incrementer` to open it for editing.
- 3 In the MATLAB Function Block Editor, select **Build Model > Build** to compile and build the MATLAB Function block code.

The build process displays some progress messages. These messages include some warnings, because the ports of the MATLAB Function block are not yet connected to signals. You can ignore these warnings.

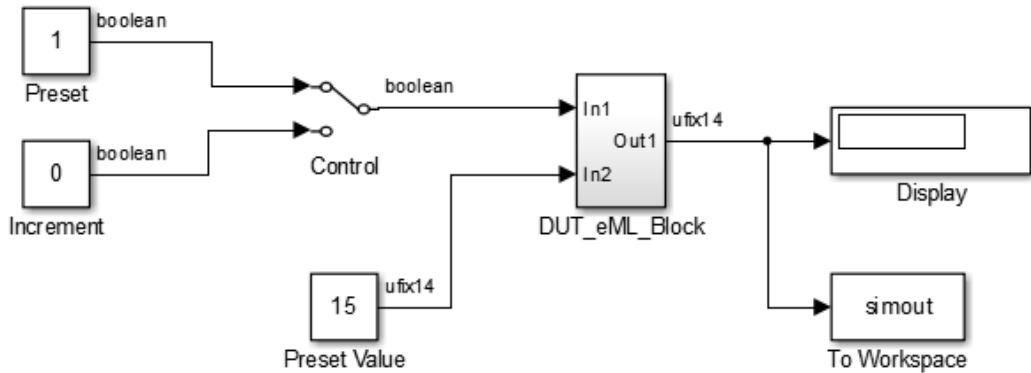
The build process builds an S-function for use in simulation. The build process includes generation of C code for the S-function. The code generation messages you see during the build process refer to generation of C code, not HDL code generation.

When the build concludes without encountering an error, a message window appears indicating that parsing was successful. If errors are found, the Diagnostics Manager lists them. See the MATLAB Function block documentation for information on debugging MATLAB Function block build errors.

Compiling the Model and Displaying Port Data Types

In this section you enable the display of port data types and then compile the model. Model compilation verifies the model structure and settings, and updates the model display.

- 1 Select **Display > Signals & Ports > Port Data Types**.
- 2 Select **Simulation > Update Diagram** (or press **Ctrl+D**) to compile the model. This triggers a rebuild of the code. After the model compiles, the block diagram updates to show the port data types.

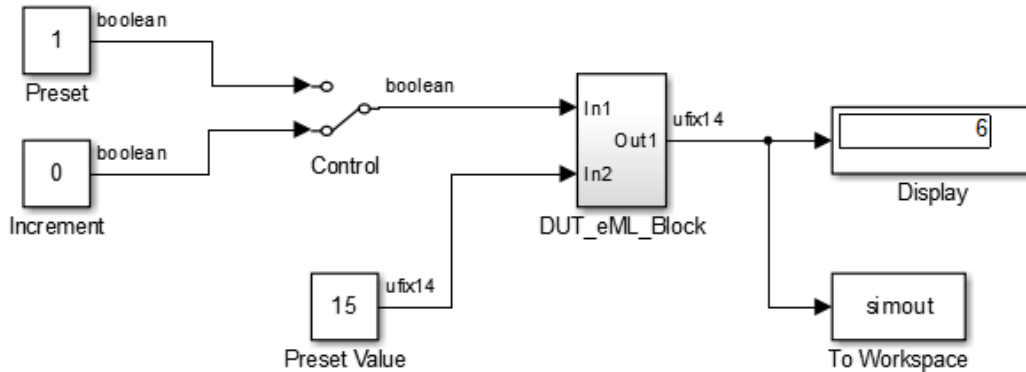


3 Save the model.

Simulating the `eml_hdl_incremter_tut` Model

Start simulation. 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 `ctr_preset` port, the simulation returns a value of 6:



You might want to experiment with the results of toggling the Control switch, changing the Preset Value constant, and changing the total simulation time. You might also want to examine the workspace variable `simout`, which is bound to the To Workspace block.

Generating HDL Code

In this section, you select the `DUT_eML_Block` subsystem for HDL code generation, set basic code generation options, and then generate VHDL code for the subsystem.

Selecting the Subsystem for Code Generation

Select the `DUT_eML_Block` subsystem for code generation:

- 1 Open the Configuration Parameters dialog box and click the **HDL Code Generation** pane.
- 2 Select `eml_hdl_incrementer_tut/DUT_eML_Block` from the **Generate HDL for** list.
- 3 Click **Apply**.

Generating VHDL Code

In the Configuration Parameters dialog box, the top-level **HDL Code Generation** options should now be set as follows:

- The **Generate HDL for** field specifies the `eml_hdl_incrementer_tut/DUT_eML_Block` subsystem for code generation.
- The **Language** field specifies (by default) generation of VHDL code.
- The **Folder** field specifies (by default) that the code generation target folder is a subfolder of your working folder, named `hdlsrc`.

Before generating code, select **Current Folder** from the **Layout** menu in the MATLAB Command Window. This displays the Current Folder browser, which lets you easily access your working folder and the files that are generated within it.

To generate code:

- 1 Click the **Generate** button.

HDL Coder compiles the model before generating code. Depending on model display options (such as port data types), the appearance of the model might change after code generation.

- 2 As code generation proceeds, the coder displays progress messages. The process should complete with a message like the following:

```
### HDL Code Generation Complete.
```

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 A folder icon for the `hdlsrc` folder is now visible in the Current Folder browser. To view generated code and script files, double-click the `hdlsrc` folder icon.
- 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:
 - `eml_inc_blk.vhd`: VHDL code. This file contains entity and architecture code implementing the actual computations generated for the MATLAB Function block.

- `DUT_eML_Block.vhd`: VHDL code. This file contains an entity definition and RTL architecture that provide a black box interface to the code generated in `eML_inc_blk.vhd`.

The structure of these code files is analogous to the structure of the model, in which the `DUT_eML_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_eML_Block_compile.do`: Mentor Graphics ModelSim compilation script (`vcom` command) to compile the VHDL code in the two `.vhd` files.
 - `DUT_eML_Block_synplify.tcl`: Synplify® synthesis script.
 - `DUT_eML_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 25-27).
- 5 To view the generated VHDL code in the MATLAB Editor, double-click the `DUT_eML_Block.vhd` or `eML_inc_blk.vhd` file icons in the Current Folder browser.

See Also

“Check for MATLAB Function block settings” on page 38-20

More About

- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “HDL Applications for the MATLAB Function Block” on page 29-2
- “MATLAB Function Block Design Patterns for HDL” on page 29-23
- “Generate DUT Ports for Tunable Parameters” on page 10-23

Generate Instantiable Code for Functions

In this section...

“How To Generate Instantiable Code for Functions” on page 29-21

“Generate Code Inline for Specific Functions” on page 29-22

“Limitations for Instantiable Code Generation for Functions” on page 29-22

For the MATLAB Function block, you can use the **InstantiateFunctions** parameter to generate a VHDL entity or Verilog 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 or Verilog 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.

If you enable `InstantiateFunctions`, `UseMatrixTypesInHDL` has no effect.

See Also

“Check for MATLAB Function block settings” on page 38-20

More About

- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “Code Generation from a MATLAB Function Block” on page 29-5
- “MATLAB Function Block Design Patterns for HDL” on page 29-23
- “Generate DUT Ports for Tunable Parameters” on page 10-23

MATLAB Function Block Design Patterns for HDL

In this section...

“The eml_hdl_design_patterns Library” on page 29-23

“Efficient Fixed-Point Algorithms” on page 29-25

“Model State Using Persistent Variables” on page 29-28

“Creating Intellectual Property with the MATLAB Function Block” on page 29-29

“Nontunable Parameter Arguments” on page 29-30

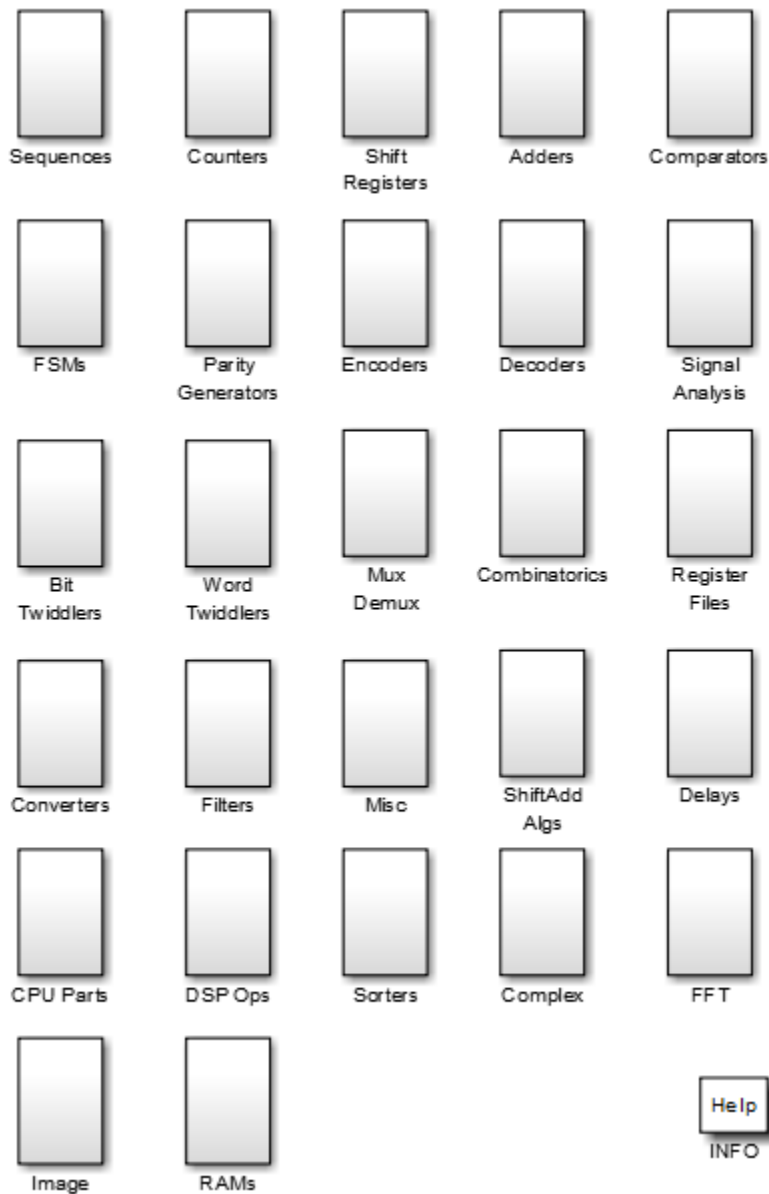
“Modeling Control Logic and Simple Finite State Machines” on page 29-30

“Modeling Counters” on page 29-32

“Modeling Hardware Elements” on page 29-33

The eml_hdl_design_patterns Library

The eml_hdl_design_patterns library is an extensive collection of examples demonstrating useful applications of the MATLAB Function block in HDL code generation.



To open the library, type the following command at the MATLAB prompt:

```
eml_hdl_design_patterns
```

You can use many blocks in the library as cookbook examples of various hardware elements, as follows:

- Copy a block from the library to your model and use it as a computational unit.
- Copy the code from the block and use 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. Use the **Inline MATLAB Function block code** check box in the **HDL Code Generation > Global Settings > Coding style** section of the Configuration Parameters dialog box. For more information, see “Inline MATLAB Function block code” on page 16-58.

Note Do not use the **Inline MATLAB Function block code** setting with the MATLAB Datapath architecture of the MATLAB Function block. Use **FlattenHierarchy** instead. For more information, see “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 24-89.

Efficient Fixed-Point Algorithms

The MATLAB Function block supports floating-point arithmetic and also fixed point arithmetic by using the Fixed-Point Designer `fi` function. This function supports rounding and saturation modes that are useful for coding 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 (for example, `Slice`, `Extend`, `Reduce`, `Concat`, etc.) that are native to VHDL and Verilog.

The following discussion shows how HDL code generated from the MATLAB Function block follows cast-before-sum semantics, in which addition and subtraction operands are cast to the result type before the addition or subtraction is performed.

Open the `eml_hdl_design_patterns` library and select the `Combinatorics/eml_expr` block. `eml_expr` implements a simple expression containing addition, subtraction, and multiplication operators with differing fixed point data types. The

generated HDL code shows the conversion of this expression with fixed point operands. The MATLAB Function block uses the following code:

```
% fixpt arithmetic expression
expr = (a*b) - (a+b);

% cast the result to (sfix7_En4) output type
y = fi(expr, 1, 7, 4);
```

The default `fimath` specification for the block determines the behavior of arithmetic expressions using fixed point operands inside the MATLAB Function block:

```
fimath(...
    'RoundMode', 'ceil',...
    'OverflowMode', 'saturate',...
    'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
    'SumMode', 'FullPrecision', 'SumWordLength', 32,...
    'CastBeforeSum', true)
```

The data types of operands and output are as follows:

- a: (sfix5_En2)
- b: (sfix5_En3)
- y: (sfix7_En4)

Before HDL code generation, this expression is broken down internally into many steps.

```
expr = (a*b) - (a+b);
```

The steps include:

```
1  tmul = a * b;
2  tadd = a + b;
3  tsub = tmul - tadd;
4  y = tsub;
```

Based on the `fimath` settings as described in “Design Guidelines for the MATLAB Function Block” on page 29-35, this expression is further broken down internally as follows:

- Based on the specified `ProductMode`, 'FullPrecision', the output type of `tmul` is computed as (sfix10_En5).

- Since the `CastBeforeSum` property is set to `'true'`, step 2 is broken down as follows:

```
t1 = (sfix7_En3) a;
t2 = (sfix7_En3) b;
tadd = t1 + t2;
```

`sfix7_En3` is the result sum type after aligning binary points and accounting for an extra bit to account for possible overflow.

- Based on intermediate types of `tmul` (`sfix10_En5`) and `tadd` (`sfix7_En3`) the result type of the subtraction in step 3 is computed as `sfix11_En5`. Accordingly, step 3 is broken down as follows:

```
t3 = (sfix11_En5) tmul;
t4 = (sfix11_En5) tadd;
tsub = t3 - t4;
```

- Finally, the result is cast to a smaller type (`sfix7_En4`) leading to the following final expression statements:

```
tmul = a * b;
t1 = (sfix7_En3) a;
t2 = (sfix7_En3) b;
tadd = t1 + t2;
t3 = (sfix11_En5) tmul;
t4 = (sfix11_En5) tadd;
tsub = t3 - t4;
y = (sfix7_En4) tsub;
```

The following listings show the generated VHDL and Verilog code from the `eml_expr` block.

This is the 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 DOWNT0 7) /= "000"))
```

```

        OR ((expr(10) = '0') AND (expr(7 DOWNT0 1) = "01111111"))
    ELSE
"1000000" WHEN (expr(10) = '1') AND (expr(9 DOWNT0 7) /= "111")
    ELSE
        std_logic_vector(expr(7 DOWNT0 1) + ("0" & expr(0)));
END fsm_SFHDL;

```

This is the Verilog code:

```

//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]})));

```

These code excerpts show that the generated HDL code from the MATLAB Function block represents the bit-true behavior of fixed point arithmetic expressions using high-level HDL operators. The HDL code is generated using HDL coding rules like high level `bitselect` and `partselect` replication operators and explicit sign extension and resize operators.

Model State Using Persistent Variables

In the MATLAB Function block programming model, state-holding elements are represented as persistent variables. A variable that is declared `persistent` retains its value across function calls in software, and across sample time steps during simulation.

Please note that your MATLAB code *must* read the persistent variable before it is written if you want HDL Coder to infer a register in the HDL code. The code generator displays a warning message if your code does not follow this rule.

The following example shows the `unit_delay` block, which delays the input sample, `u`, by one simulation time step. `u` is a fixed-point operand of type `sfixed`. `u_d` is a persistent variable that holds the input sample.

```

function y = fcn(u)
persistent u_d;
if isempty(u_d)

```



```

    u_d = fi(-1, numerictype(u), fimath(u));
end

% return delayed input from last sample time hit
y = u_d;

% store the current input to be used later
u_d = u;

```

Because this code intends for `u_d` to infer a register during HDL code generation, `u_d` is read in the assignment statement, `y = u_d`, before it is written in `u_d = u`.

HDL Coder generates the following HDL code for the `unit_delay` block.

```

ENTITY Unit_Delay IS
  PORT (
    clk : IN std_logic;
    clk_enable : IN std_logic;
    reset : IN std_logic;
    u : IN std_logic_vector(15 DOWNT0 0);
    y : OUT std_logic_vector(15 DOWNT0 0));
END Unit_Delay;

ARCHITECTURE fsm_SFHD OF Unit_Delay IS

BEGIN
  initialize_Unit_Delay : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      y <= std_logic_vector(to_signed(0, 16));
    ELSIF clk'EVENT AND clk = '1' THEN
      IF clk_enable = '1' THEN
        y <= u;
      END IF;
    END IF;
  END PROCESS initialize_Unit_Delay;

```

Initialization of persistent variables is moved into the master reset region in the initialization process.

Refer to the `Delays` subsystem in the `eml_hdl_design_patterns` library to see how vectors of persistent variables can be used to model integer delay, tap delay, and tap delay vector blocks. These design patterns are useful in implementing sequential algorithms that carry state between executions of the MATLAB Function block in a model.

Creating Intellectual Property with the MATLAB Function Block

The MATLAB Function block helps you author intellectual property and create alternate implementations of part of an algorithm. By using MATLAB Function blocks in this way,

you can guide the detailed operation of the HDL code generator even while writing high-level algorithms.

For example, the subsystem `Comparators` in the `eml_hdl_design_patterns` library includes several alternate algorithms for finding the minimum value of a vector. The `Comparators/eml_linear_min` block finds the minimum of the vector in a linear mode serially. The `Comparators/eml_tree_min` block compares the elements in a tree structure. The tree implementation can achieve a higher clock frequency by adding pipeline registers between the $\log_2(N)$ stages. (See `eml_hdl_design_patterns/Filters` for an example.)

Now consider replacing the simple comparison operation in the `Comparators` blocks with an arithmetic operation (for example, addition, subtraction, or multiplication) where intermediate results must be quantized. Using `fimath` rounding settings, you can fine tune intermediate value computations before intermediate values feed into the next stage. You can use this technique for tuning the generated hardware or customizing your algorithm.

Nontunable Parameter Arguments

You can declare a nontunable parameter for a MATLAB Function block by setting its **Scope** to **Parameter** in the Ports and Data Manager GUI, and clearing the **Tunable** option.

A nontunable parameter does not appear as a signal port on the block. Parameter arguments for MATLAB Function blocks take their values from parameters defined in a parent Simulink masked subsystem or from variables defined in the MATLAB base workspace, not from signals in the Simulink model.

Modeling Control Logic and Simple Finite State Machines

MATLAB Function block control constructs such as `switch/case` and `if-elseif-else`, coupled with fixed point arithmetic operations let you model control logic quickly.

The `FSMs/mealy_fsm_blk` and `FSMs/moore_fsm_blk` blocks in the `eml_hdl_design_patterns` library provide example implementations of Mealy and Moore finite state machines in the MATLAB Function block.

The following listing implements a Moore state machine.

```
function Z = moore_fsm(A)
```

```

persistent moore_state_reg;
if isempty(moore_state_reg)
    moore_state_reg = fi(0, 0, 2, 0);
end

S1 = 0;
S2 = 1;
S3 = 2;
S4 = 3;

switch uint8(moore_state_reg)
    case S1,
        Z = true;
        if (~A)
            moore_state_reg(1) = S1;
        else
            moore_state_reg(1) = S2;
        end
    case S2,
        Z = false;
        if (~A)
            moore_state_reg(1) = S1;
        else
            moore_state_reg(1) = S2;
        end
    case S3,
        Z = false;
        if (~A)
            moore_state_reg(1) = S2;
        else
            moore_state_reg(1) = S3;
        end
    case S4,
        Z = true;
        if (~A)
            moore_state_reg(1) = S1;
        else
            moore_state_reg(1) = S3;
        end
    otherwise,
        Z = false;
end

```

In this example, a persistent variable (`moore_state_reg`) models state variables. The output depends only on the state variables, thus modeling a Moore machine.

The FSMs/`mealy_fsm_blk` block in the `eml_hdl_design_patterns` library implements a Mealy state machine. A Mealy state machine differs from a Moore state machine in that the outputs depend on inputs as well as state variables.

The MATLAB Function block can quickly model simple state machines and other control-based hardware algorithms (such as pattern matchers or synchronization-related controllers) 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.

Modeling Counters

To implement arithmetic and control logic algorithms in MATLAB Function blocks intended for HDL code generation, there are some simple HDL related coding requirements:

- The top level MATLAB Function block must be called once per time step.
- It must be possible to fully unroll program loops.
- Persistent variables with reset values and update logic must be used to hold values across simulation time steps.
- Quantized data variables must be used inside loops.

The following script shows how to model a synchronous up/down counter with preset values and control inputs. The example provides both master reset control of persistent state variables and local reset control using block inputs (e.g. `presetClear`). The `isempty` condition enters the initialization process under the control of a synchronous reset. The `presetClear` section is implemented in the output section in the generated HDL code.

Both the up and down case statements implementing the count loop require that the values of the counter are quantized after addition or subtraction. By default, the MATLAB Function block automatically propagates fixed-point settings specified for the block. In this script, however, fixed-point settings for intermediate quantities and constants are explicitly specified.

```
function [Q, QN] = up_down_ctr(upDown, presetClear, loadData, presetData)

% up down result
% 'result' synthesizes into sequential element

result_nt = numerictype(0,4,0);
result_fm = fimath('OverflowMode', 'saturate', 'RoundMode', 'floor');

initVal = fi(0, result_nt, result_fm);

persistent count;
if isempty(count)
    count = initVal;
end

if presetClear
    count = initVal;
elseif loadData
    count = presetData;
elseif upDown
    inc = count + fi(1, result_nt, result_fm);
    -- quantization of output
    count = fi(inc, result_nt, result_fm);
```

```

else
    dec = count - fi(1, result_nt, result_fm);
    -- quantization of output
    count = fi(dec, result_nt, result_fm);
end

Q = count;
QN = bitcmp(count);

```

Modeling Hardware Elements

The following code example shows how to model shift registers in MATLAB Function block code by using the `bitsliceget` and `bitconcat` functions. This function implements a serial input and output shifters with a 32-bit fixed-point operand input. See the Shift Registers/`shift_reg_lby32` block in the `eml_hdl_design_patterns` library for more details.

```

function sr_out = fcn(shift, sr_in)
%shift register 1 by 32

persistent sr;
if isempty(sr)
    sr = fi(0, 0, 32, 0, 'fimath', fimath(sr_in));
end

% return sr[31]
sr_out = getmsb(sr);

if (shift)
    % sr_new[32:1] = sr[31:1] & sr_in
    sr = bitconcat(bitsliceget(sr, 31, 1), sr_in);
end

```

The following code example shows VHDL process code generated for the `shift_reg_lby32` block.

```

shift_reg_lby32 : PROCESS (shift, sr_in, sr)
BEGIN
    sr_next <= sr;
    -- MATLAB Function Function 'Subsystem/shift_reg_lby32': '<S2>:1'
    -- shift register 1 by 32
    -- '<S2>:1:1'
    -- return sr[31]
    -- '<S2>:1:10'
    sr_out <= sr(31);

    IF shift /= '0' THEN
        -- '<S2>:1:12'
        -- sr_new[32:1] = sr[31:1] & sr_in
        -- '<S2>:1:14'
        sr_next <= sr(30 DOWNT0 0) & sr_in;
    END IF;

END PROCESS shift_reg_lby32;

```

The `Shift Registers/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.

Browse the `eml_hdl_design_patterns` model for other useful hardware elements that can be easily implemented using the MATLAB Function block.

See Also

“Check for MATLAB Function block settings” on page 38-20

More About

- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “Code Generation from a MATLAB Function Block” on page 29-5
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-39
- “Generate DUT Ports for Tunable Parameters” on page 10-23

Design Guidelines for the MATLAB Function Block

In this section...

“Use Compiled External Functions With MATLAB Function Blocks” on page 29-35

“Build the MATLAB Function Block Code First” on page 29-35

“Use the hdlfimath Utility for Optimized FIMATH Settings” on page 29-35

“Use Optimal Fixed-Point Option Settings” on page 29-36

“Set the Output Data Type of MATLAB Function Blocks Explicitly” on page 29-36

“Using Tunable Parameters” on page 29-36

“Run HDL Model Check for MATLAB Function Blocks” on page 29-37

“Use MATLAB Datapath Architecture for Enhanced HDL Optimizations” on page 29-37

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” (MATLAB Coder) 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, select **Build** from the **Tools** menu in the MATLAB Function Block Editor (or press **CTRL+B**).

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 does not support 'floor' rounding mode. Use 'round' mode or change the signed integer division operations to unsigned integer division.

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'
```

- In the Ports and Data Manager dialog box, select the **tunable** check box.

For details, see “Generate DUT Ports for Tunable Parameters” on page 10-23.

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 38-20 in the HDL Model Checker. 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 “Getting Started with the HDL Model Checker” on page 39-2.

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 24-89.

See Also

“Check for MATLAB Function block settings” on page 38-20

More About

- “Code Generation from a MATLAB Function Block” on page 29-5
- “MATLAB Function Block Design Patterns for HDL” on page 29-23
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-39
- “Generate DUT Ports for Tunable Parameters” on page 10-23

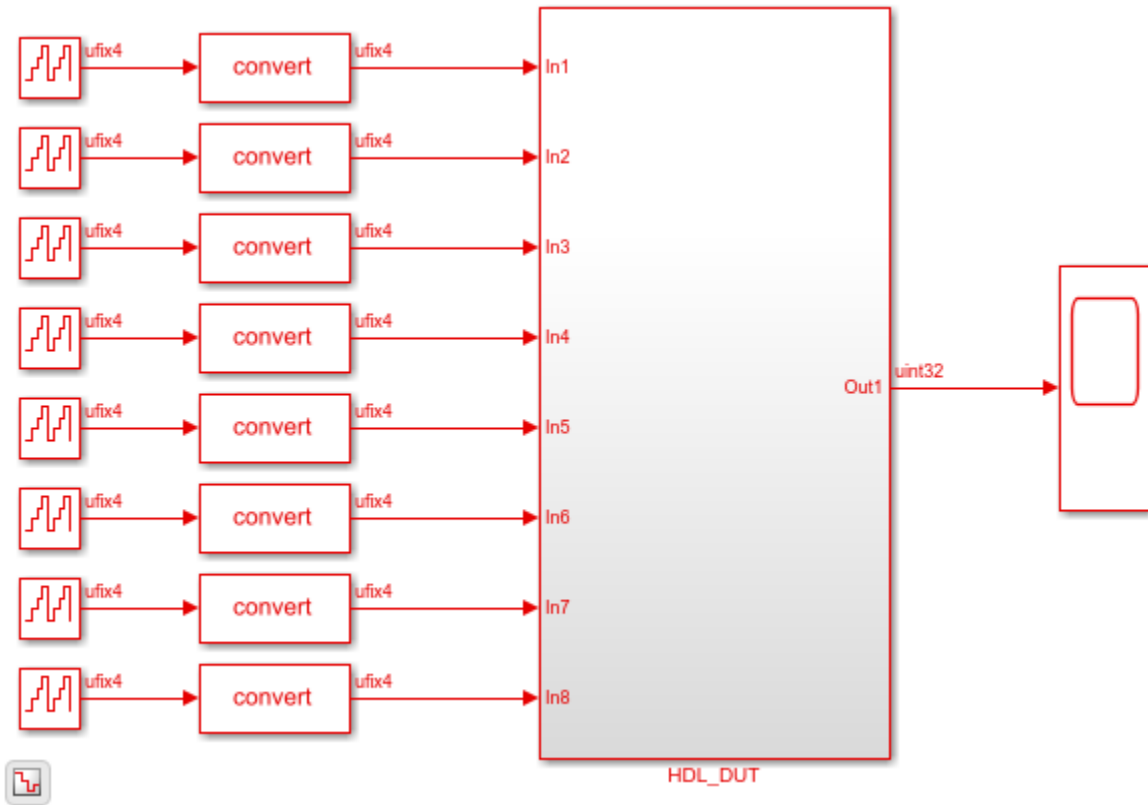
Distributed Pipeline Insertion for MATLAB Function Blocks

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.

Multiplier Chain Model

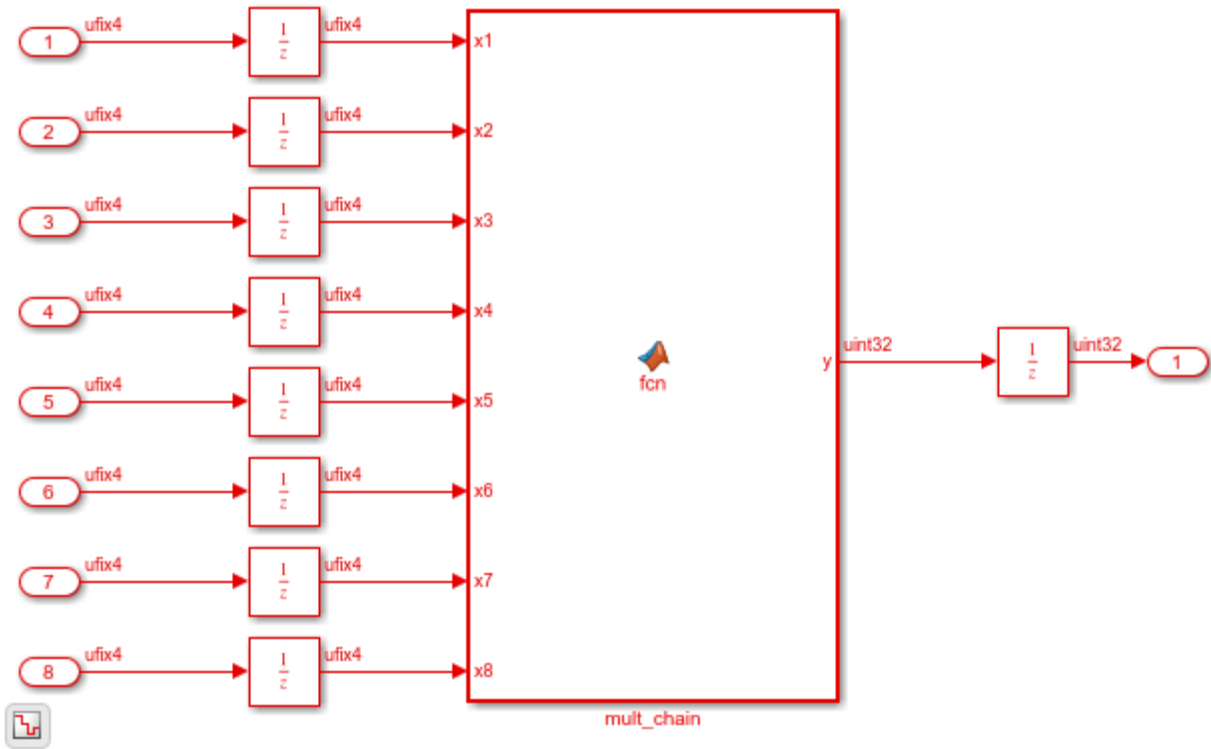
This example shows how to distribute pipeline registers in a simple model that chains five multiplications.

```
open_system('hdlcoder_distpipe_multiplier_chain')  
set_param('hdlcoder_distpipe_multiplier_chain','SimulationCommand','Update')
```



The HDL_DUT Subsystem is the DUT for which you want to generate HDL code. The subsystem drives a MATLAB Function block `mult_chain`.

```
open_system('hdlcoder_distpipe_multiplier_chain/HDL_DUT')
```



To see the chain of multiplications, open the MATLAB Function block.

```
open_system('hdlcoder_distpipe_multiplier_chain/HDL_DUT/mult_chain')
```

```

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.

```

ml_subsys = 'hdlcoder_distpipe_multiplier_chain/HDL_DUT/mult_chain';
hdlset_param(ml_subsys, 'OutputPipeline', 2)

```

2. Specify the MATLAB Datapath architecture. This architecture treats the MATLAB Function block like a regular Subsystem. You can then apply various optimizations across the MATLAB Function blocks with other blocks in your Simulink® model.

```

hdlset_param(ml_subsys, 'architecture', 'MATLAB Datapath');

```

3. Enable the distributed pipelining optimization on the block. To see the results of the optimization, enable generation of the Optimization Report. To apply the optimization across hierarchies in your model, enable hierarchical distributed pipelining on the model and distributed pipelining on all subsystems.

```

hdlset_param('hdlcoder_distpipe_multiplier_chain', ...
             'HierarchicalDistPipe', 'on', 'OptimizationReport', 'on')
hdlset_param('hdlcoder_distpipe_multiplier_chain/HDL_DUT', 'DistributedPipelining', 'on')
hdlset_param(ml_subsys, 'DistributedPipelining', 'on');

```

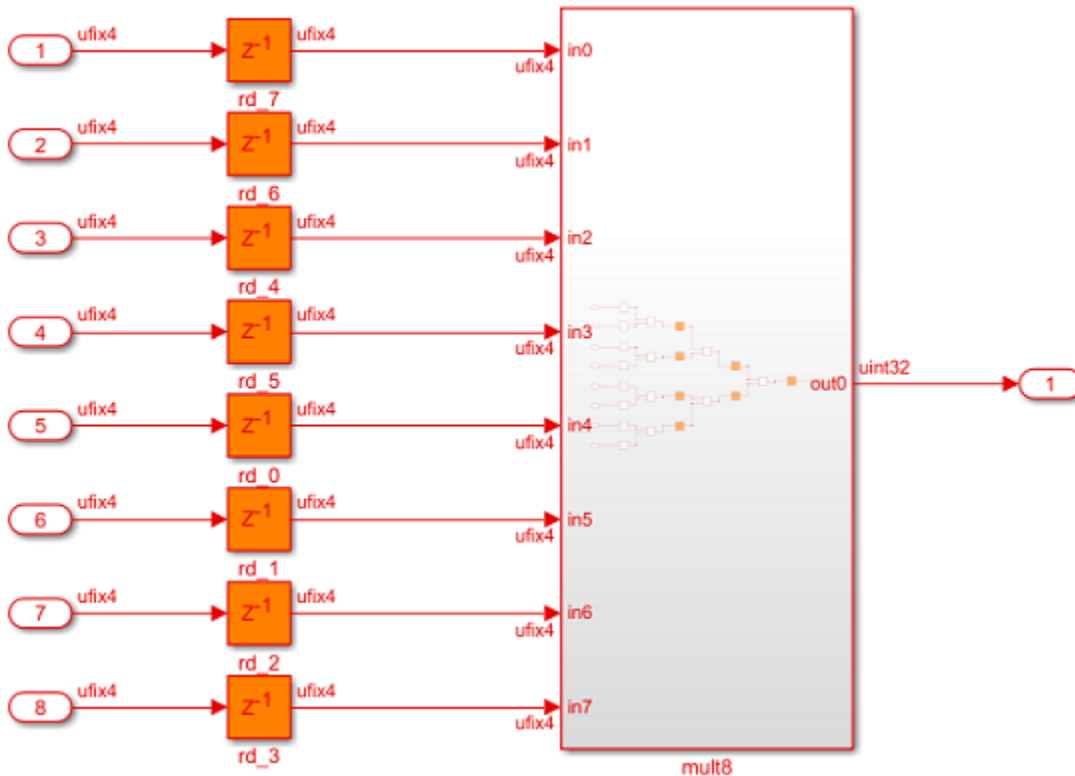
4. Generate HDL Code for the HDL_DUT subsystem.

```
makehdl('hdlcoder_distpipe_multiplier_chain/HDL_DUT/mult_chain')
```

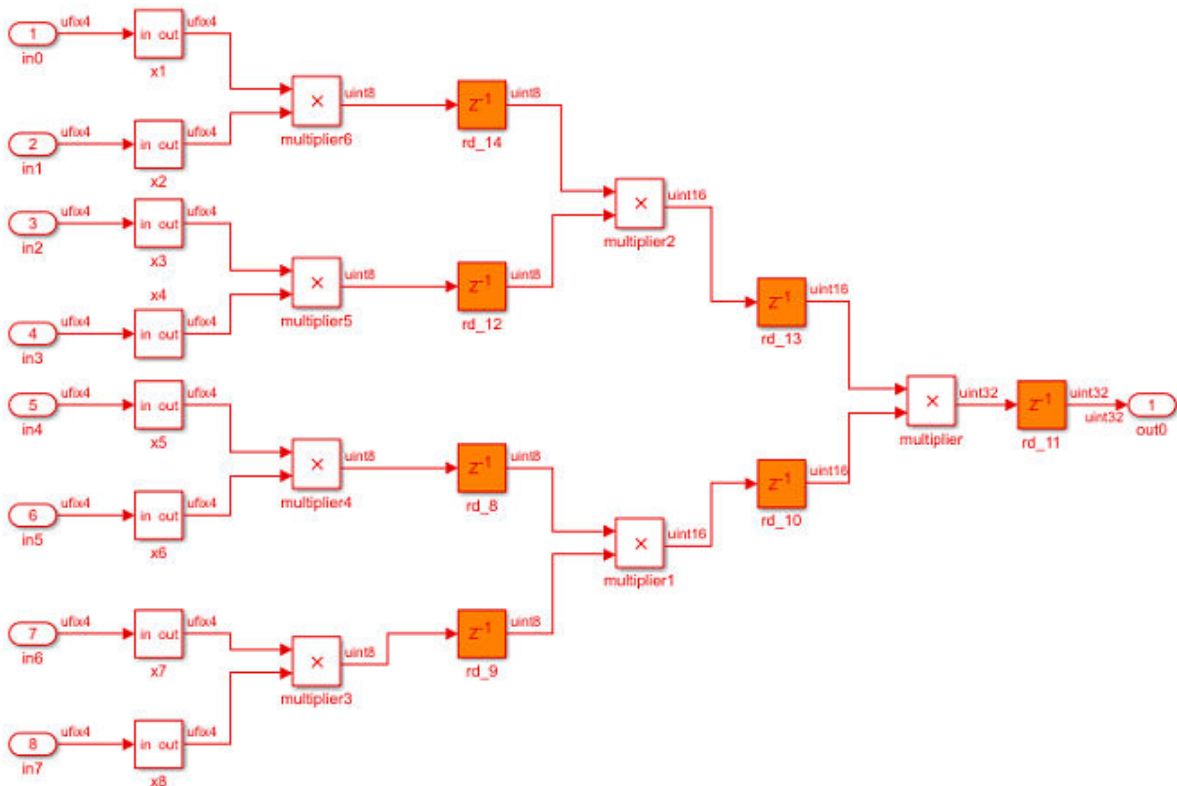
By default, HDL Coder generates VHDL code in the `hdlsrc` folder.

Analyze Results of Optimization

In the Distributed Pipelining report, 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 creates a Subsystem in place of the MATLAB Function block. The optimization can then distribute the pipeline registers and the unit delay that you added inside the Subsystem to optimize the multiplier chain and improve timing. Open the `mult_chain` Subsystem.



See Also

“Check for MATLAB Function block settings” on page 38-20

More About

- “Design Guidelines for the MATLAB Function Block” on page 29-35
- “Code Generation from a MATLAB Function Block” on page 29-5
- “MATLAB Function Block Design Patterns for HDL” on page 29-23
- “Generate DUT Ports for Tunable Parameters” on page 10-23

Generating Scripts for HDL Simulators and Synthesis Tools

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Structure of Generated Script Files” on page 30-3
- “Properties for Controlling Script Generation” on page 30-4
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-8
- “Add Synthesis Attributes” on page 30-18
- “Configure Synthesis Project Using Tcl Script” on page 30-19

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 30-4.
- 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 30-8.

Structure of Generated Script Files

A generated EDA script consists of three sections, 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 or Verilog 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

This section describes how to set properties in the `makehdl` or `makehdltb` functions to enable or disable script generation and customize the names and content of generated script files.

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	<code>HDLCompileFilePostfix</code>	<code>_compile.do</code>
Simulation	<code>HDLSimFilePostfix</code>	<code>_sim.do</code>
Synthesis	<code>HDLSynthFilePostfix</code>	Depends on the selected synthesis tool. See <code>HDLSynthTool</code> .

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: <code>HDLCompileInit</code> Default: <code>'vlib %s\n'</code>	Format name passed to <code>fprintf</code> to write the <code>Init</code> section of the compilation script. The implicit argument is the contents of the <code>VHDLLibraryName</code> property, which defaults to <code>'work'</code> . You can override the default <code>Init</code> string (<code>'vlib work\n'</code>) by changing the value of <code>VHDLLibraryName</code> .
Name: <code>HDLCompileVHDLCmd</code> Default: <code>'vcom %s %s\n'</code>	Format name passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for VHDL files. The two implicit arguments are the contents of the <code>SimulatorFlags</code> property and the file name of the current entity or module. To omit the flags, set <code>SimulatorFlags</code> to <code>' '</code> (the default).
Name: <code>HDLCompileVerilogCmd</code> Default: <code>'vlog %s %s\n'</code>	Format name passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for Verilog files. The two implicit arguments are the contents of the <code>SimulatorFlags</code> property and the file name of the current entity or module. To omit the flags, set <code>SimulatorFlags</code> to <code>' '</code> (the default).
Name: <code>HDLCompileTerm</code> Default: <code>' '</code>	Format name passed to <code>fprintf</code> to write the termination portion of the compilation script.

Property Name and Default	Description
Name: HDLSimInit Default: ['onbreak resume\n',... 'onerror resume\n']	Format name passed to <code>fprintf</code> to write the initialization section of the simulation script.
Name: HDLSimCmd Default: 'vsim -novopt %s.%s\n'	Format name passed to <code>fprintf</code> to write the simulation command. If your target language is VHDL, the first implicit argument is the value of the <code>VHDLLibraryName</code> property. If your target language is Verilog, 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 <code>fprintf</code> 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 <code>fprintf</code> 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 <code>HDLSynthTool</code> .
Name: HDLSynthInit	Format name passed to <code>fprintf</code> to write the Init section of the synthesis script. The content of the format name is specific to the selected synthesis tool. See <code>HDLSynthTool</code> .
Name: HDLSynthCmd	Format name passed to <code>fprintf</code> to write the Cmd section of the synthesis script. The content of the string is specific to the selected synthesis tool. See <code>HDLSynthTool</code> .

Property Name and Default	Description
Name: HDLSynthTerm	<p>Format name passed to <code>fprintf</code> to write the Term section of the synthesis script.</p> <p>The content of the string is specific to the selected synthesis tool. See <code>HDLSynthTool</code>.</p>

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`.

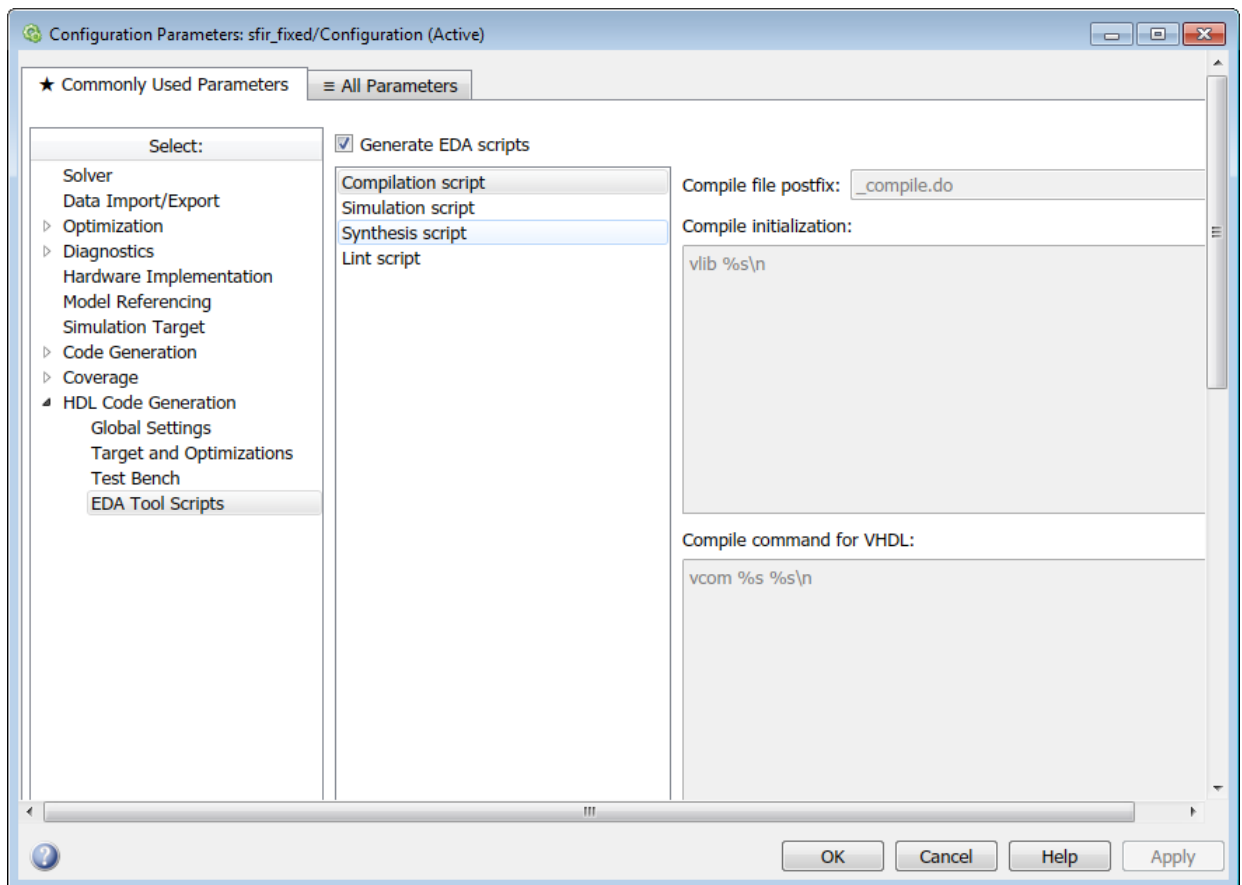
```
set src_dir "./hdlsrc"
set prj_dir "synprj"
file mkdir ../$prj_dir
cd ../$prj_dir
project new symmetric_fir.ise
xfile add ../$src_dir/symmetric_fir.vhd
project set family Virtex4
project set device xc4vsx35
project set package ff668
project set speed -10
process run "Synthesize - XST"
```

Configure Compilation, Simulation, Synthesis, and Lint Scripts

You set options that configure script file generation on the **EDA Tool Scripts** pane. These options correspond to the properties described in “Properties for Controlling Script Generation” on page 30-4.

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.



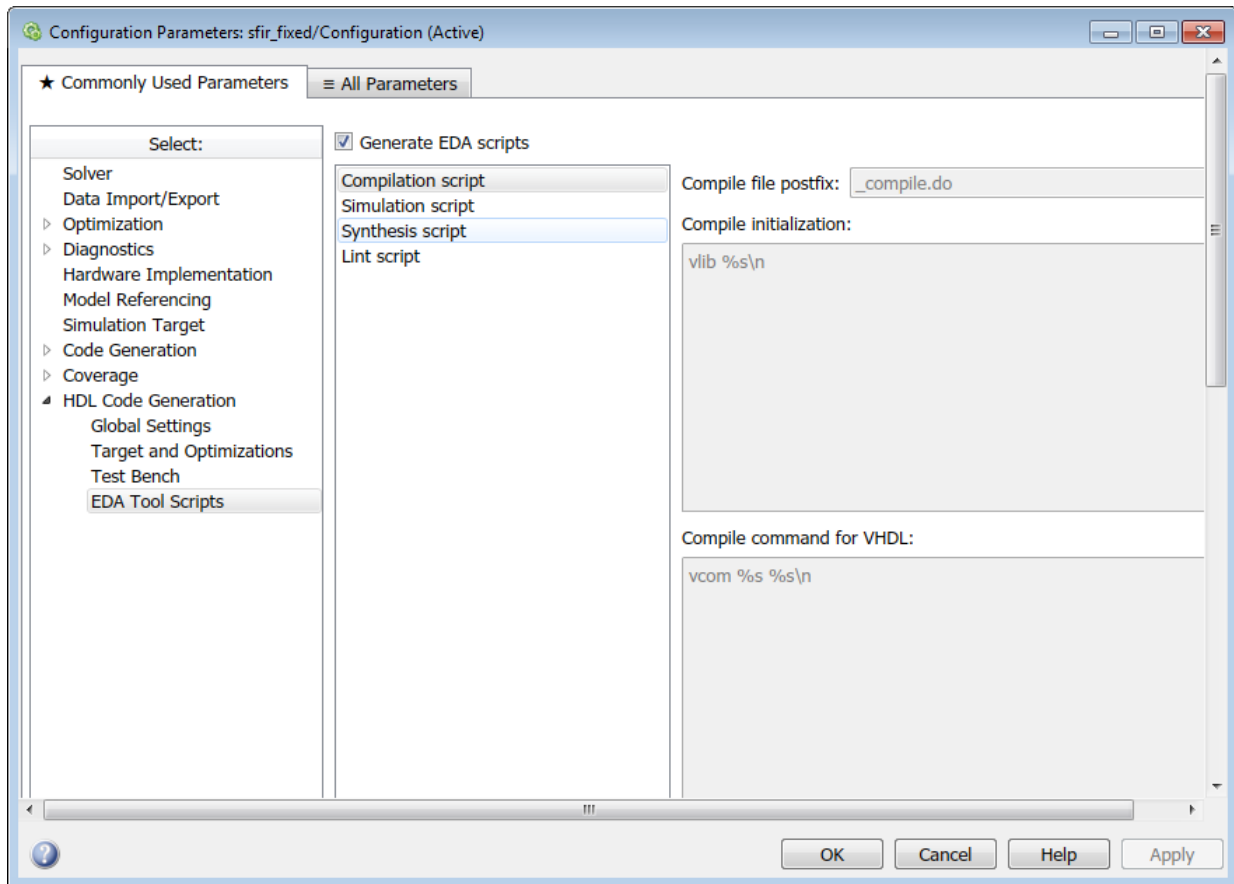
- 3** The **Generate EDA scripts** option controls the generation of script files. By default, this option is selected.

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 or Verilog code. See “Compilation Script Options” on page 30-9 for further information.
 - **Simulation script:** Options related to customizing scripts for HDL simulators. See “Simulation Script Options” on page 30-11 for further information.
 - **Synthesis script:** Options related to customizing scripts for synthesis tools. See “Synthesis Script Options” on page 30-13 for further information.

Compilation Script Options

The following figure shows the **Compilation script** pane, with options set to their default values.



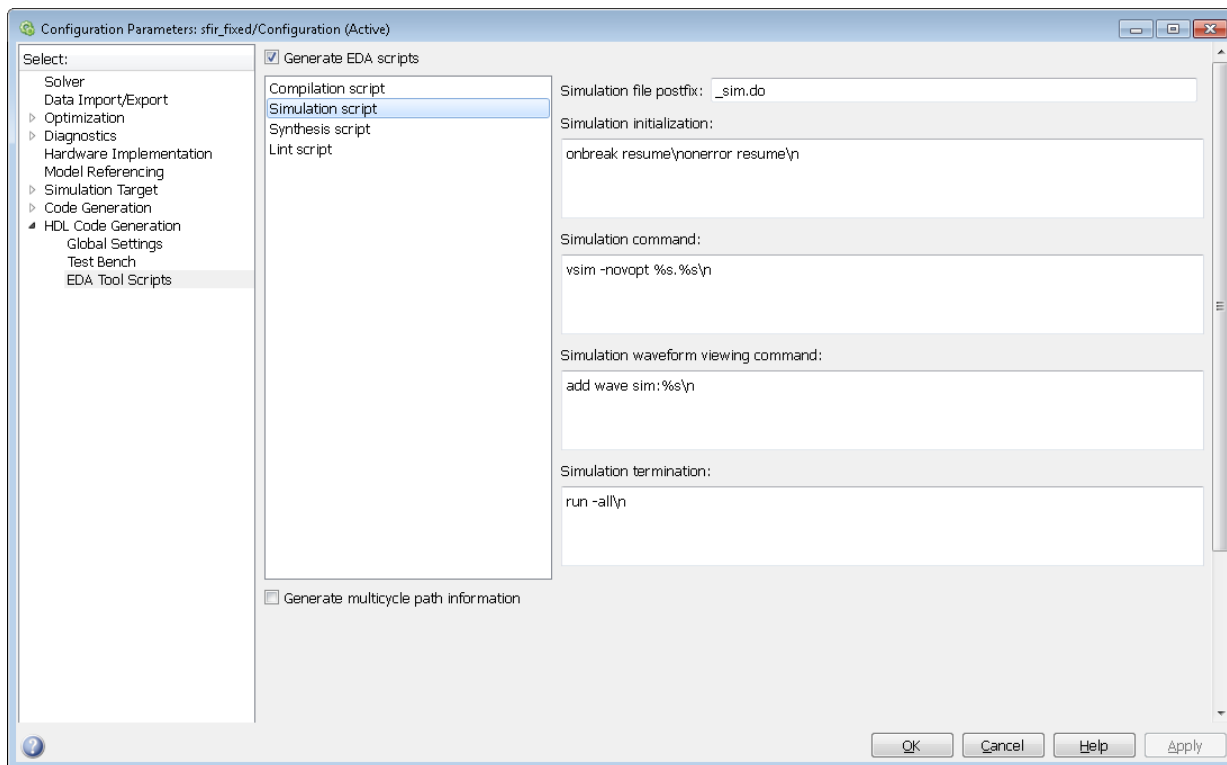
The following table summarizes the **Compilation script** options.

Option and Default	Description
Compile file postfix' '_compile.do'	Postfix appended to the DUT name or test bench name to form the script file name.

Option and Default	Description
Name: Compile initialization Default: 'vlib %s\n'	Format name passed to <code>fprintf</code> to write the <code>Init</code> section of the compilation script. The argument is the contents of the <code>VHDLLibraryName</code> property, which defaults to 'work'. You can override the default <code>Init</code> 'vlib work\n' by changing the value of <code>VHDLLibraryName</code> .
Name: Compile command for VHDL Default: 'vcom %s %s\n'	Format name passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for VHDL files. The two arguments are the contents of the <code>SimulatorFlags</code> property option and the filename of the current entity or module. To omit the flags, set <code>SimulatorFlags</code> to '' (the default).
Name: Compile command for Verilog Default: 'vlog %s %s\n'	Format name passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for Verilog files. The two arguments are the contents of the <code>SimulatorFlags</code> property and the filename of the current entity or module. To omit the flags, set <code>SimulatorFlags</code> to '' (the default).
Name: Compile termination Default: ''	Format name passed to <code>fprintf</code> 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.



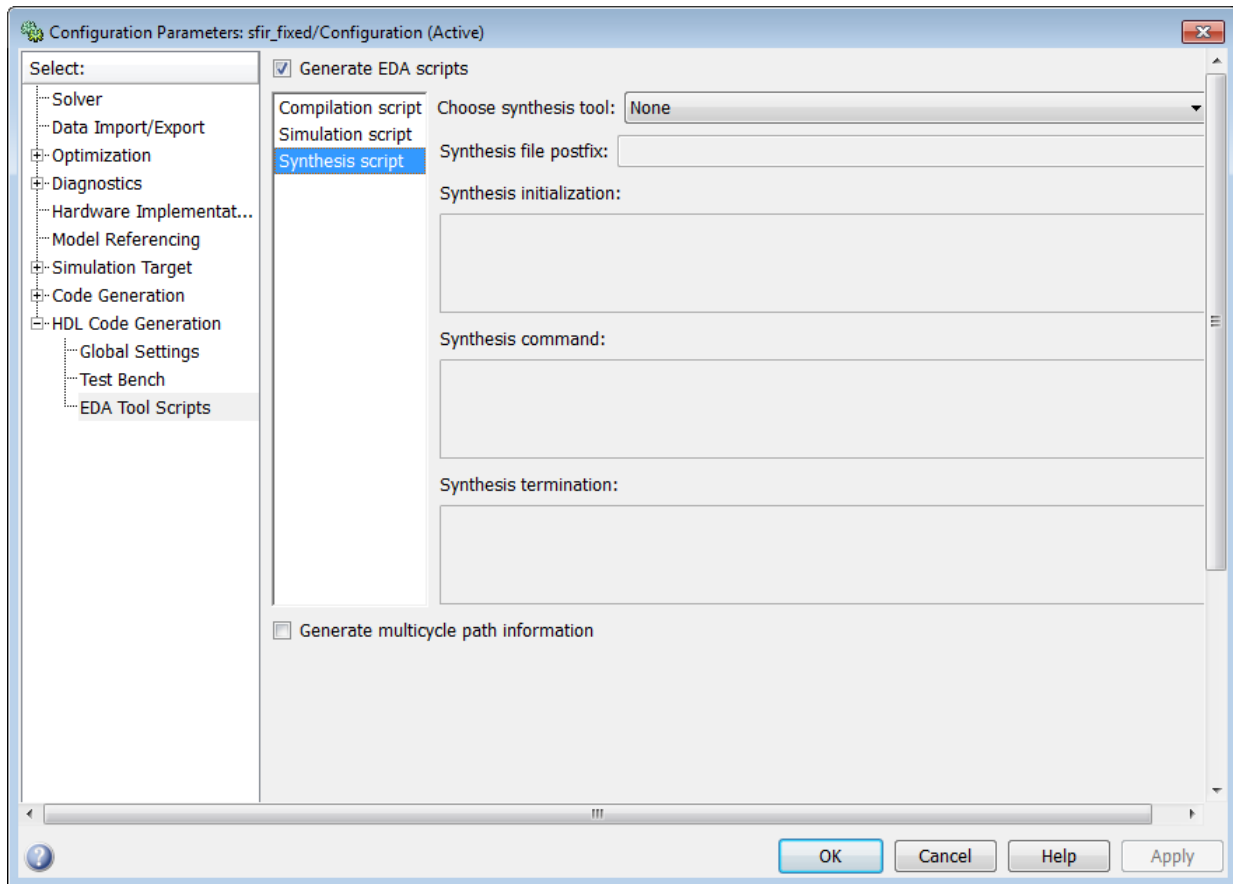
The following table summarizes the **Simulation script** options.

Option and Default	Description
Simulation file postfix '_sim.do'	Postfix appended to the model name or test bench name to form the simulation script file name.
Simulation initialization Default: ['onbreak resume\nnonerror resume\n']	Format name passed to <code>fprintf</code> to write the initialization section of the simulation script.

Option and Default	Description
Simulation command Default: 'vsim -novopt %s.%s\n'	Format name passed to <code>fprintf</code> to write the simulation command. If your <code>TargetLanguage</code> is 'VHDL', the first implicit argument is the value of <code>VHDLLibraryName</code> . If your <code>TargetLanguage</code> is 'Verilog', 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 <code>fprintf</code> 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 <code>fprintf</code> to write the Term portion of the simulation script.

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.

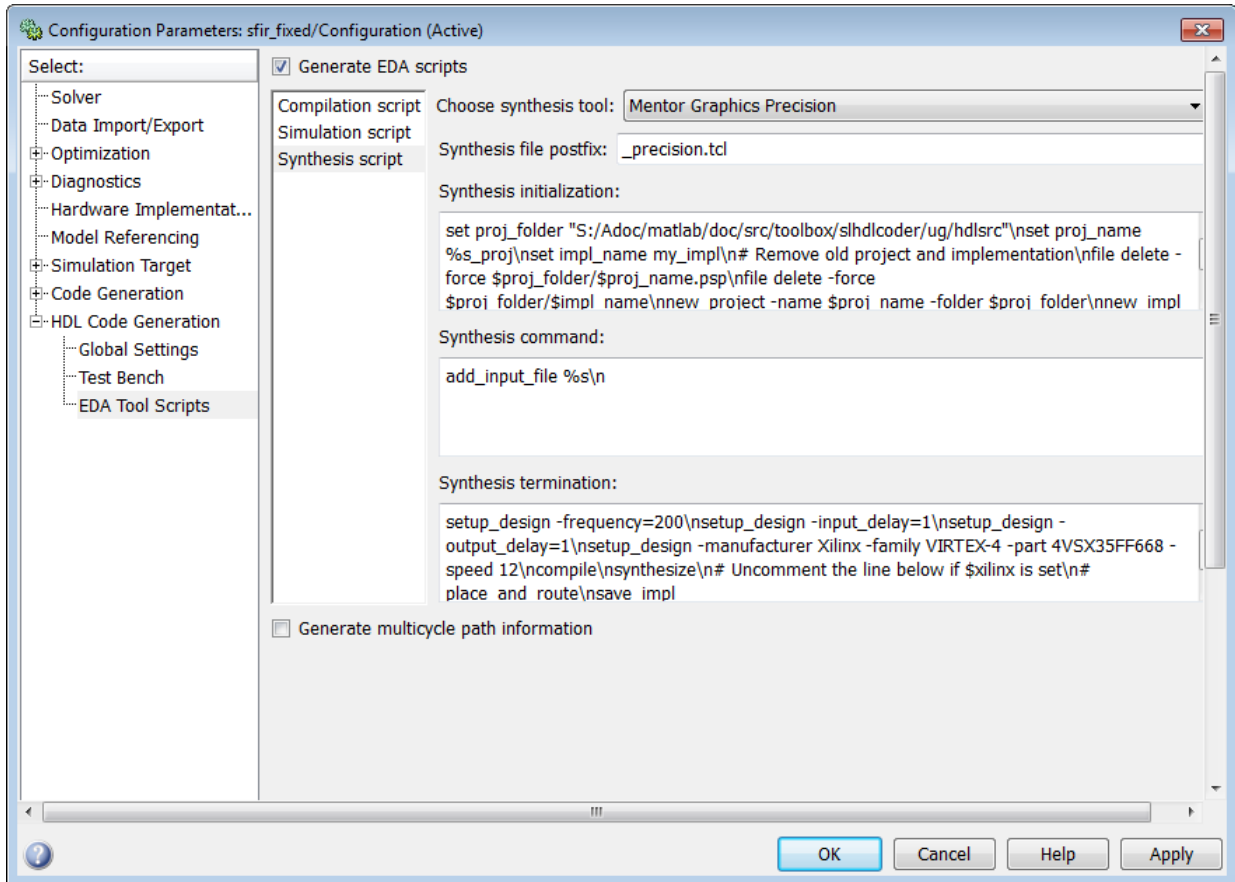


To enable synthesis script generation, select a synthesis tool from the **Choose synthesis tool** menu.

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	<p>None (default): do not generate a synthesis script</p> <p>Xilinx ISE: generate a synthesis script for Xilinx ISE</p> <p>Microsemi Libero: generate a synthesis script for Microsemi Libero</p> <p>Mentor Graphics Precision: generate a synthesis script for Mentor Graphics Precision</p> <p>Altera Quartus II: generate a synthesis script for Altera Quartus II</p> <p>Synopsys Synplify Pro: generate a synthesis script for Synopsys Synplify Pro</p> <p>Xilinx Vivado: generate a synthesis script for Xilinx Vivado</p> <p>Custom: generate a custom synthesis script</p>
Synthesis file postfix	<p>Your choice of synthesis tool sets the postfix for generated synthesis file names to one of the following:</p> <pre>_ise.tcl _libero.tcl _precision.tcl _quartus.tcl _synplify.tcl _vivado.tcl _custom.tcl</pre>
Synthesis initialization	<p>Format name passed to <code>fprintf</code> to write the <code>Init</code> 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.</p> <p>The content of the string is specific to the selected synthesis tool.</p>
Synthesis command	<p>Format name passed to <code>fprintf</code> to write the <code>Cmd</code> section of the synthesis script. The implicit argument is the file name of the entity or module.</p> <p>The content of the string is specific to the selected synthesis tool.</p>

Option Name	Description
Synthesis termination	Format name passed to <code>fprintf</code> to write the Term section of the synthesis script. The content of the string is specific to the selected synthesis tool.

Add Synthesis Attributes

To learn how to add synthesis attributes in the generated HDL code for multiplier mapping, see “DSPStyle” on page 21-12.

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

- “Getting Started with the HDL Workflow Advisor” on page 31-2
- “Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor” on page 31-10
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-14
- “Customize Floating-Point IP Configuration” on page 31-23
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 31-34
- “Synthesis Objective to Tcl Command Mapping” on page 31-39
- “Run HDL Workflow with a Script” on page 31-42

Getting Started with the HDL Workflow Advisor

In this section...

“Open the HDL Workflow Advisor” on page 31-2

“Run Tasks in the HDL Workflow Advisor” on page 31-3

“Fix HDL Workflow Advisor Warnings or Failures” on page 31-4

“Save and Restore the HDL Workflow Advisor State” on page 31-5

“View and Save HDL Workflow Advisor Reports” on page 31-7

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.

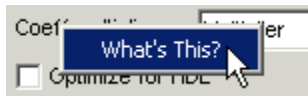
For how to use the HDL Workflow Advisor to generate HDL code from a MATLAB script, see “Basic HDL Code Generation with the Workflow Advisor” on page 5-10.

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 depends 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.

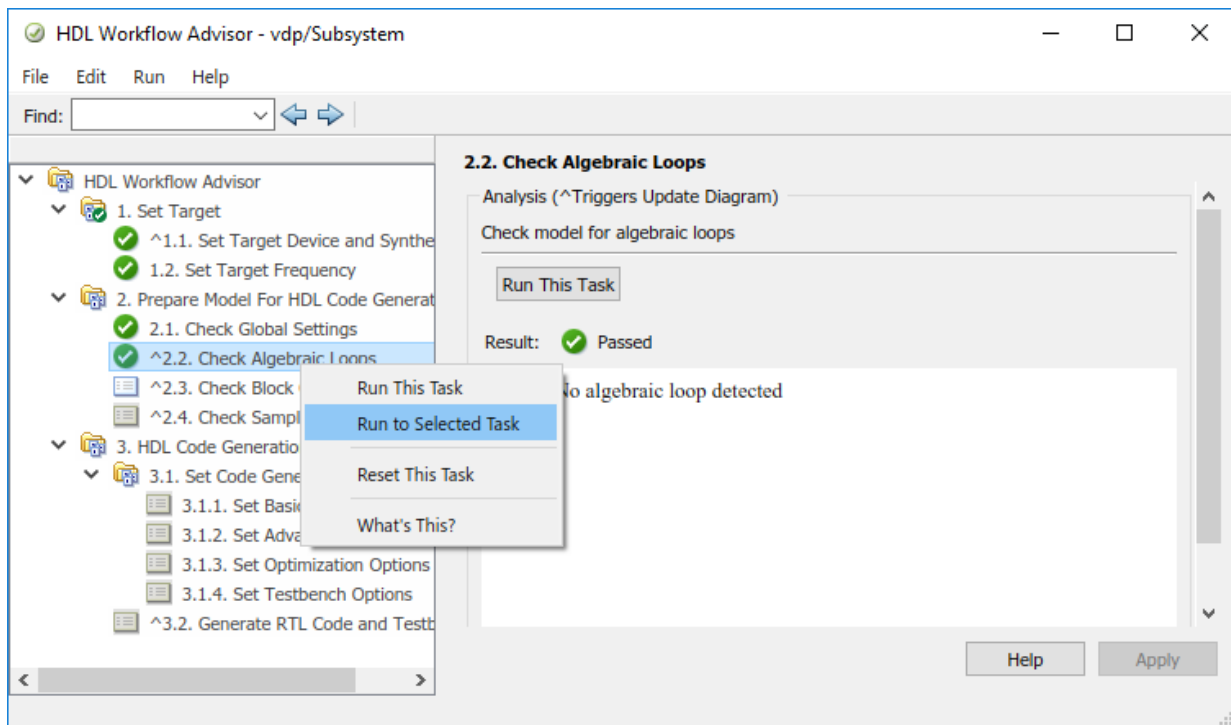
Run Tasks in the HDL Workflow Advisor

In the HDL Workflow Advisor window, you can run individual tasks, a group of tasks, or all the tasks in the workflow. For example, before you generate HDL code, prepare the Simulink model for HDL code generation. In the **Set Target** folder, for each individual task, you can specify the target device settings and the target frequency. Then, 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 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**.

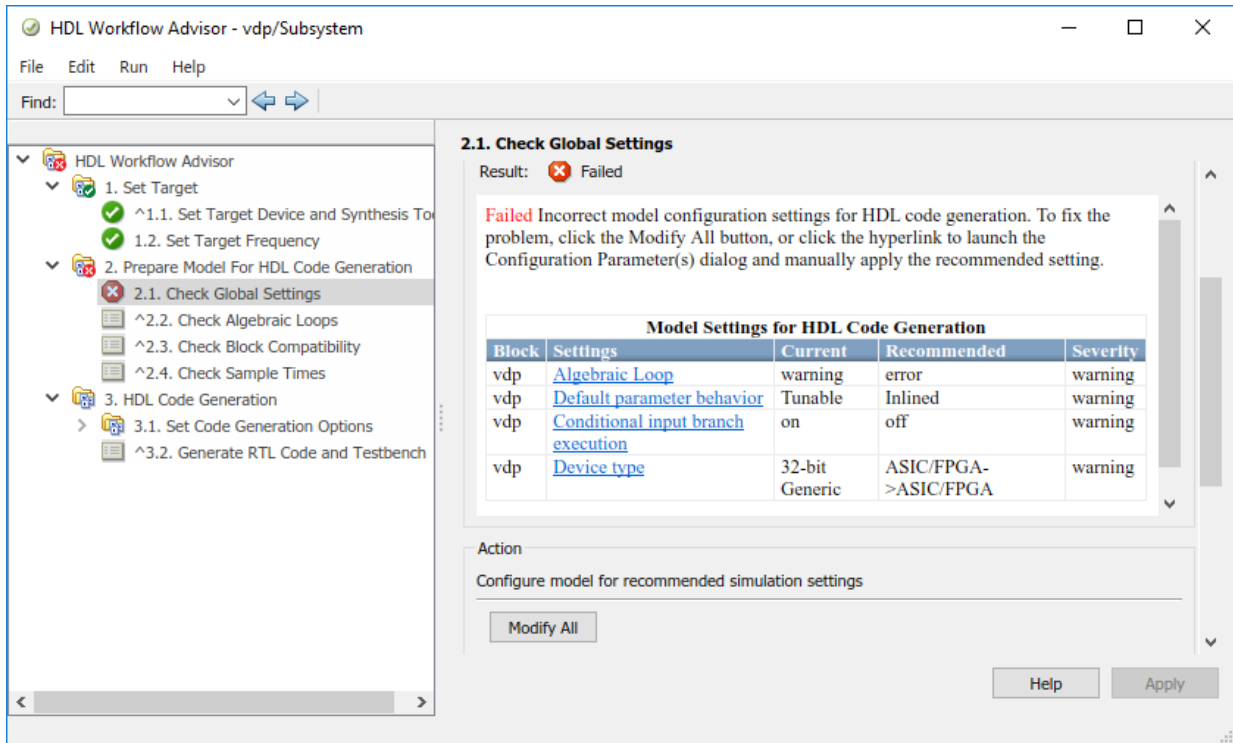


To rerun a task that you have already run, click **Reset This Task**. HDL Coder then resets the task and all tasks that follow it. For example, to customize the basic options for generating HDL code after running the **Generate RTL Code and Testbench** task, right-click the **Set Basic Options** task 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.



For example, to apply the correct model configuration settings that the code generator reported in the **Result** subpane, click the **Modify All** button. After you click **Modify All**, the **Result** subpane reports the changes that were applied, and resets the task. You can now run this task.

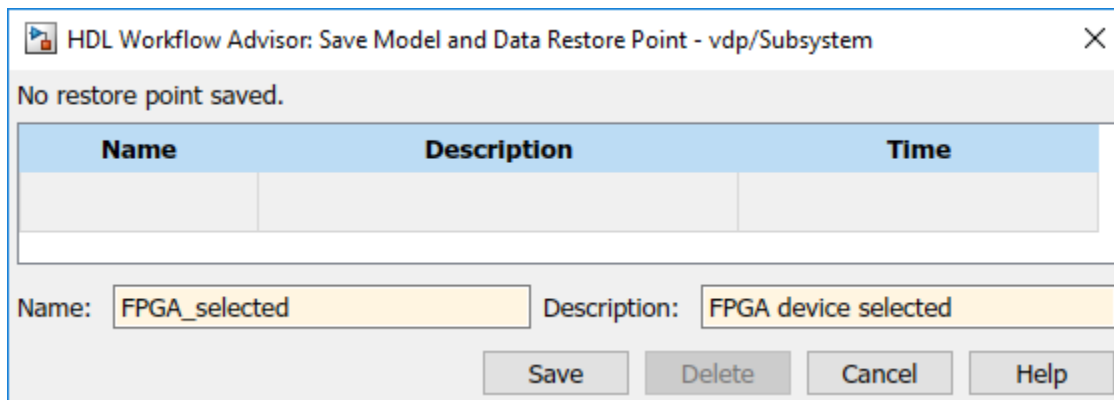
Save and Restore the HDL Workflow Advisor State

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.

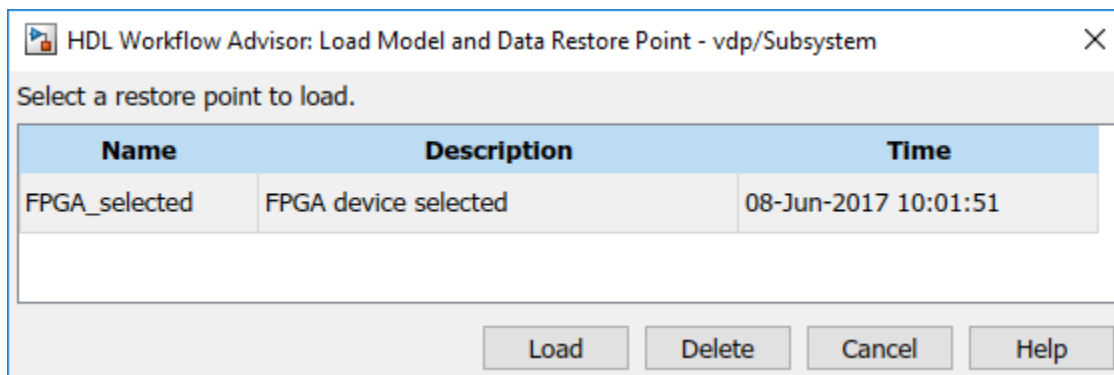
The save and restore process does not:

- 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.

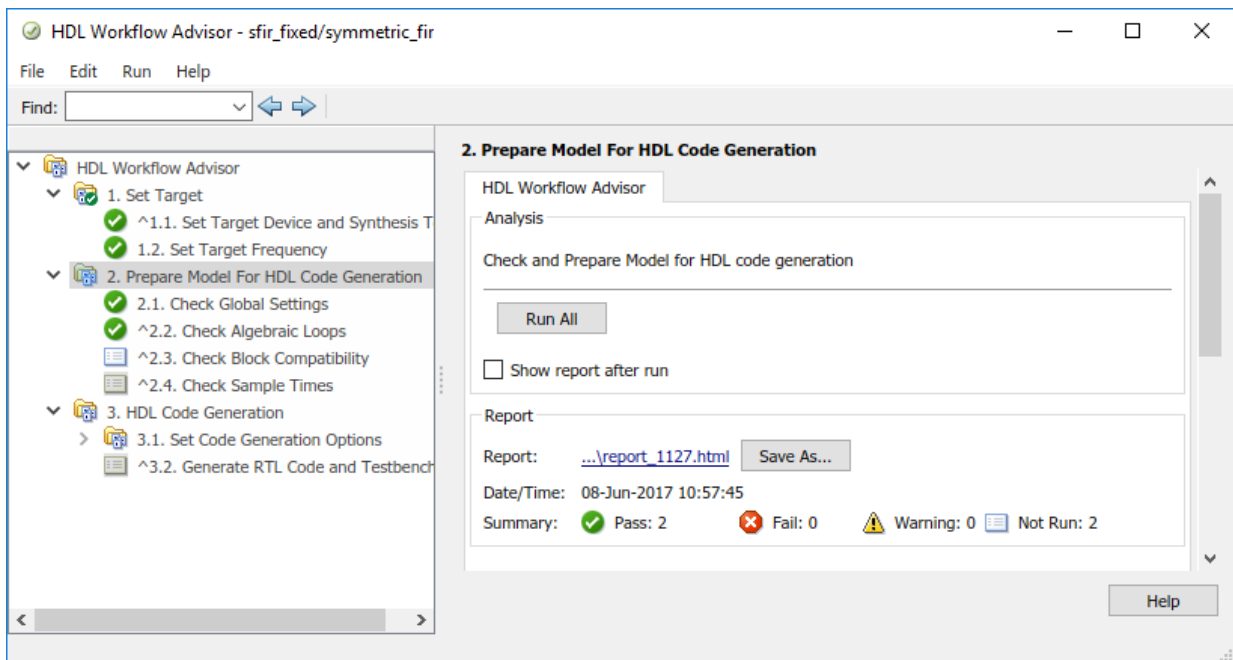


To restore a Workflow Advisor state, in the HDL Workflow Advisor, select **File > Load Restore Point**. Select the restore point that you want to load and click **Load**. When you load a restore point, the HDL Workflow Advisor warns that the restoration overwrites the current settings.



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 the 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.

Model Advisor Report - sfir_fixed.mdl

Simulink version: 9.0 Model version: 1.70
System: sfir_fixed/symmetric_fir Current run: 08-Jun-2017 10:57:45

1 item with a timestamp different than 08-Jun-2017 10:57:45
Treat as Referenced Model: off

Run Summary

Pass	Fail	Warning	Not Run	Total
2	0	0	2	4

2. Prepare Model For HDL Code Generation

2.1. Check Global Settings (08-Jun-2017 10:54:13)
Passed Correct Simulation settings for HDL code generation

Input Parameters Selection

Name	Value
Ignore warnings	false

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.WorkflowConfig.clearAllTasks |
hdlcoder.WorkflowConfig.setAllTasks | hdlcoder.runWorkflow

Classes

hdlcoder.WorkflowConfig

Related Examples

- “HDL Workflow Advisor Tasks” on page 37-2
- “HDL Code Generation and FPGA Synthesis Using Simulink HDL Workflow Advisor”

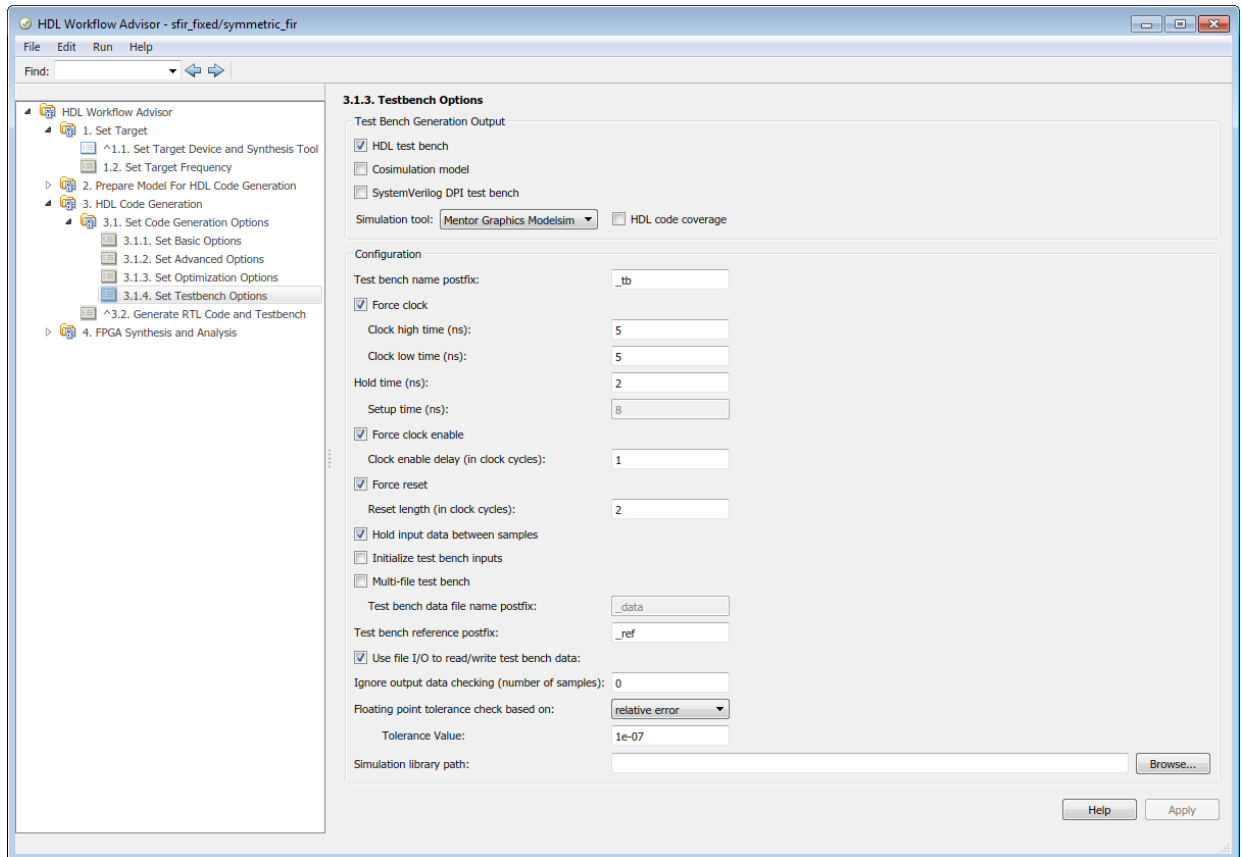
Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor

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.

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 Using Simulink HDL Workflow Advisor”.
- 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 27-38. 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`.

Questa Coverage Report

Number of tests run: 1

Passed: 0
Warning: 1
Error: 0
Fatal: 0

[List of tests included in report...](#)

[List of global attributes included in report...](#)

Coverage Summary by Structure:

Design Scope	Coverage
symmetric_fir_tb	59.91%
u_symmetric_fir	97.03%
symmetric_fir_tb_pkg	0.00%
to_hex	0.00%
to_hex_1	0.00%
to_hex_2	0.00%
to_hex_3	0.00%
to_hex_4	0.00%

Coverage Summary by Type:

Total Coverage:						73.69%	53.71%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage	
Statements	200	154	46	1	77.00%	77.00%	
Branches	131	103	28	1	78.62%	78.62%	
FEC Expressions	19	12	7	1	63.15%	63.15%	
FEC Conditions	10	3	7	1	30.00%	30.00%	
Toggles	3076	2261	815	1	73.50%	73.50%	
Assertions	1	0	1	1	0.00%	0.00%	

See Also

More About

- “Choose a Test Bench for Generated HDL Code” on page 27-38

Generate HDL Code for FPGA Floating-Point Target Libraries

In this section...

“Setup for FPGA Floating-Point Library Mapping” on page 31-14

“Map to an FPGA Floating-Point Library” on page 31-15

“View Code Generation Reports of Floating-Point Library Mapping” on page 31-18

“Analyze Results of Floating-Point Library Mapping” on page 31-19

Mapping to a floating-point library enables you to synthesize your floating-point design without having to do floating-point to fixed-point conversion. Eliminating the floating-point to fixed-point 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. Altera Megafunctions and Xilinx LogiCORE IP are examples of such libraries.

In the HDL Coder block library, a subset of Simulink blocks support floating-point library mapping. See “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 31-34.

Setup for FPGA Floating-Point Library Mapping

To map your floating-point design to an Altera or Xilinx FPGA floating-point library:

- 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:

```
hdlset_param(model, 'SynthesisToolChipFamily', 'Arria10', ...
                  'SynthesisToolDeviceName', '10AS066H2F34E1SG', ...
                  'SynthesisToolPackageName', '', ...
                  'SynthesisToolSpeedValue', '')
```

- To set up the path to your synthesis tool executable file, use `hdlsetuptoolpath`. For example, to set the path to the Altera Quartus II synthesis tool:

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', 'ToolPath', ...
                 'C:\altera\14.0\quartus\bin\quartus.exe');
```

See “Synthesis Tool Path Setup”.

- Set up your Altera or Xilinx FPGA floating-point simulation libraries. See “FPGA Simulation Library Setup”.

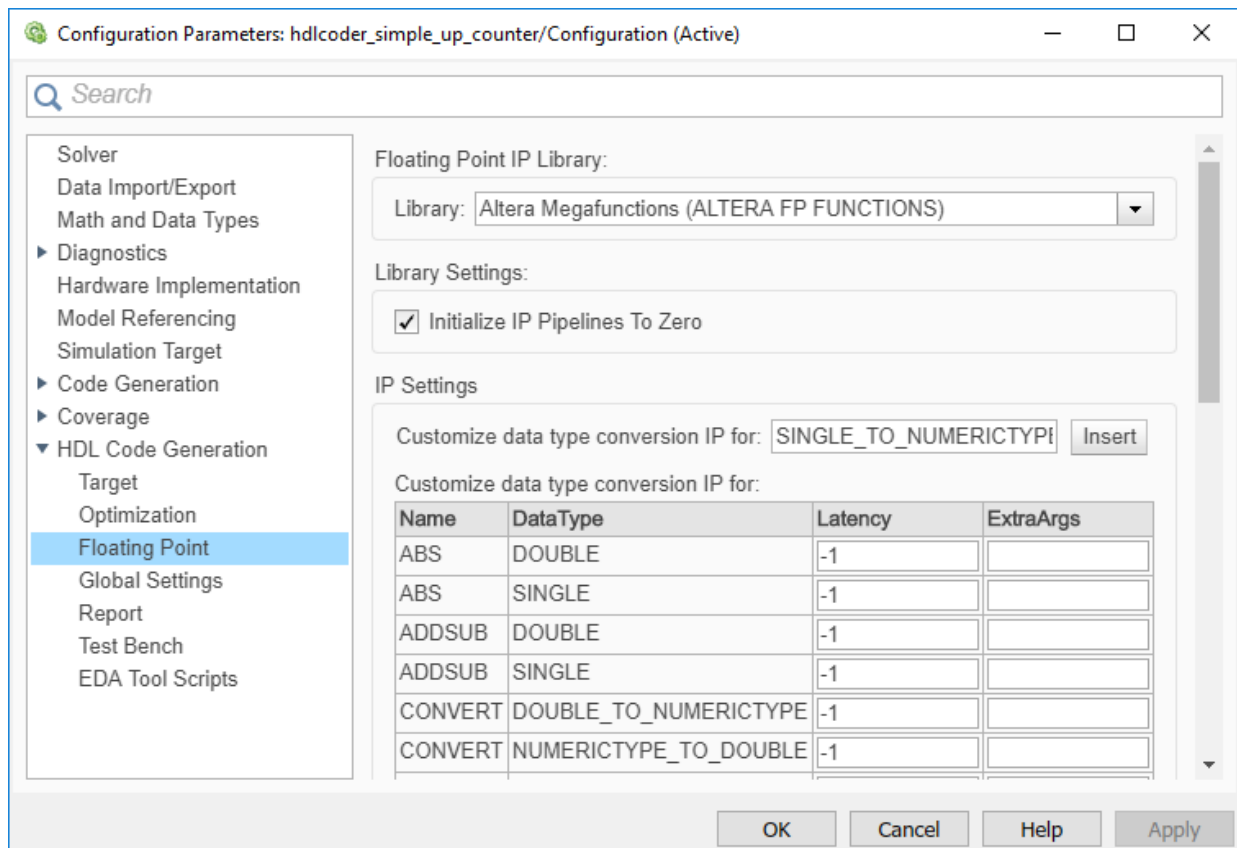
Map to an FPGA Floating-Point Library

You can map your Simulink model to floating-point target libraries from the Configuration Parameters dialog box or from the command line.

From the Configuration Parameters Dialog Box

To map to an FPGA floating-point library:

- 1** In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Click **Settings**.
- 2** In the **HDL Code Generation > Floating Point Target** pane, select the floating-point IP library.



- For Xilinx LogiCORE IP, select **XILINX LOGICORE** as the library. For Altera megafunction IP, you can select **ALTERA MEGAFUNCTION (ALTFP)** or **ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)** as the library.
- If you choose **ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)** as the library, the **Initialize IP Pipelines to Zero** option becomes available. Select the **Initialize IP Pipelines to Zero** option to initialize pipeline registers in the IP to zero. In the **Target and Optimizations** pane, enter the target frequency that you want the floating-point IP to map to.

Note When mapping to ALTERA FP FUNCTIONS, the target language must be set to VHDL.

When you choose the ALTERA FP FUNCTIONS library, an IP Configuration table appears. By using the data type table, you can customize the IP settings of the floating-point target library. For more information, see “Customize the IP Latency with Target Frequency” on page 31-24.

- 5 If you choose **XILINX LOGICORE** or **ALTERA MEGAFUNCTION (ALTFP)** as the library, select the **Latency Strategy** and **Objective** for the IP.

When you choose these libraries, an IP Configuration table appears. By using the data type table, you can customize the latency of the floating-point target IP. For more information, see “Customize the IP Latency with Latency Strategy” on page 31-28.

- 6 To share floating-point IP resources, on the **HDL Code Generation > Target and Optimizations > Resource Sharing** tab, make sure that **Floating-point IPs** is enabled. The number of floating-point IP blocks that get shared depends on the **SharingFactor** that you specify on the subsystem.
- 7 Click **Apply**. On the Simulink Toolstrip, click **Generate HDL Code**.

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 By using the `hdlcoder.createFloatingPointTargetConfig` function, create a `hdlcoder.FloatingPointTargetConfig` object for the 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 ALTERA FP FUNCTIONS library with the default settings:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTERAFPFUNCTIONS');
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
```

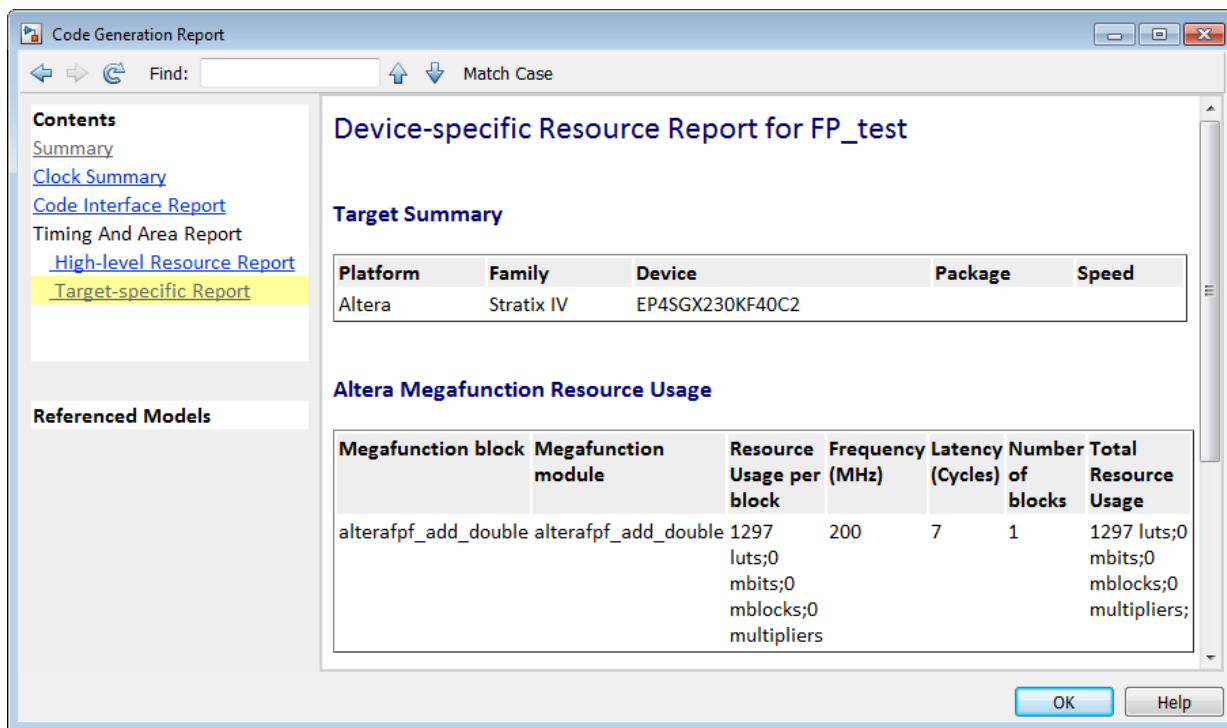
- 2 You can customize the IP settings based on the floating-point library that you specify. For more information, see “Customize Floating-Point IP Configuration” on page 31-23.
- 3 Use `makehdl` to generate HDL code from the subsystem.

View Code Generation Reports of Floating-Point Library Mapping

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 learn how to generate these reports, see “Create and Use Code Generation Reports” on page 25-2.

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**.



The screenshot shows the 'Code Generation Report' window. The 'Contents' pane on the left has the 'Target-specific Report' item highlighted. The main area displays the 'Device-specific Resource Report for FP_test'.

Target Summary

Platform	Family	Device	Package	Speed
Altera	Stratix IV	EP4SGX230KF40C2		

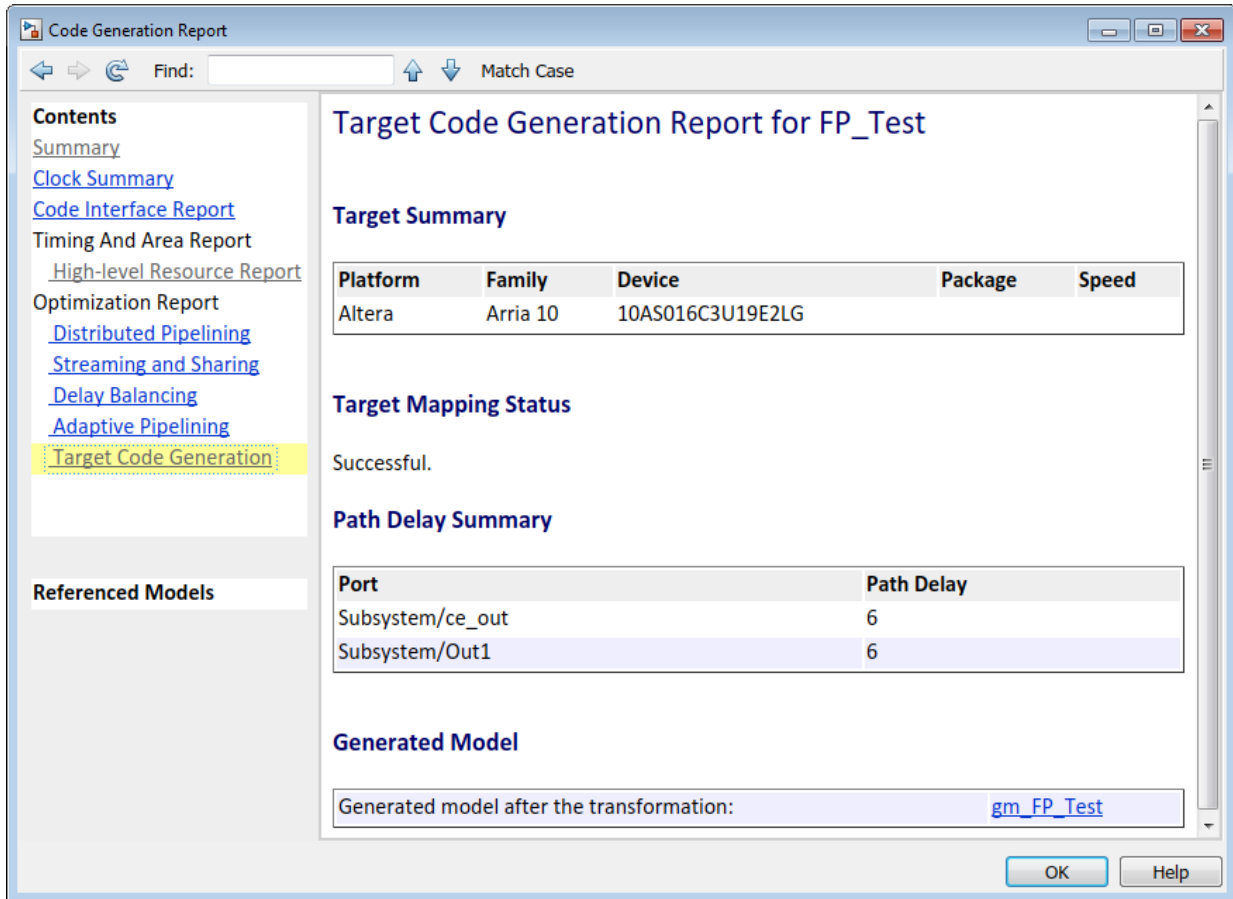
Altera Megafunction Resource Usage

Megafunction block	Megafunction module	Resource Usage per block	Frequency (MHz)	Latency (Cycles)	Number of blocks	Total Resource Usage
alterafpf_add_double	alterafpf_add_double	1297 luts;0 mbits;0 mblocks;0 multipliers	200	7	1	1297 luts;0 mbits;0 mblocks;0 multipliers

Buttons for 'OK' and 'Help' are visible at the bottom right of the window.

Target Code Generation Report

In the Code Generation Report, the **Target Code Generation** section in the Optimization Report shows the status of optimization settings applied to the model. The report shows whether HDL Coder successfully generated floating-point target code.



The screenshot shows the 'Code Generation Report' window. The left sidebar contains a 'Contents' list with 'Target Code Generation' highlighted. The main area displays the 'Target Code Generation Report for FP_Test' with the following sections:

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

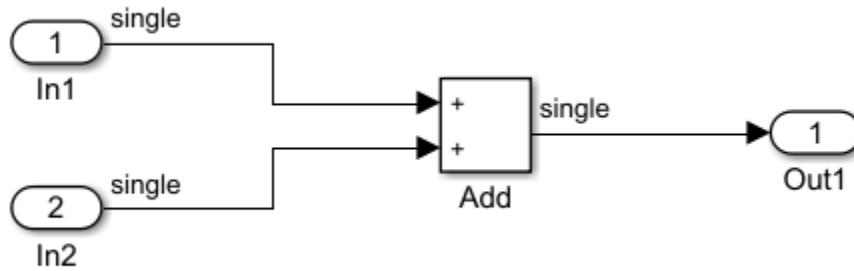
Generated Model

Generated model after the transformation: [gm_FP_Test](#)

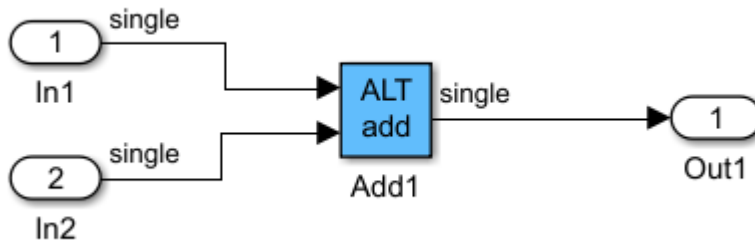
Buttons for 'OK' and 'Help' are visible at the bottom right.

Analyze Results of Floating-Point Library Mapping

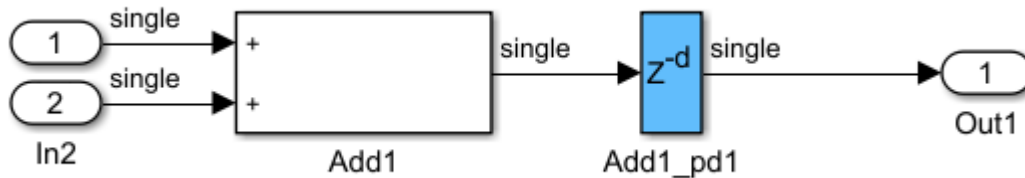
You can get the latency information of the floating-point target IP from the generated model after HDL code generation. For example, consider this add block in Simulink with inputs of double data type.



- 1 After HDL code generation, the optimization report for target code generation displays a link to a generated model. To see the floating-point target library that your Simulink block mapped to, double-click the subsystem in the generated model.

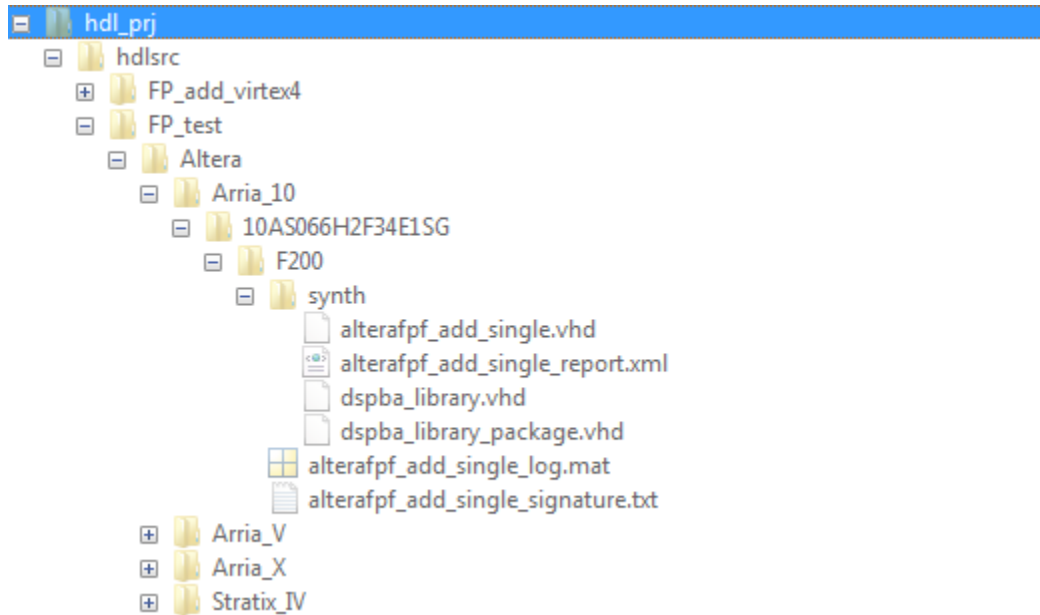


- 2 Double-click the **ALT add** block. The length of the delay block is the latency of the floating-point target IP.



To learn more about the generated model, see “Generated Model and Validation Model” on page 24-5.

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, taking less time when successively generating code for the same floating-point target IP.

See Also

Related Examples

- “FPGA Floating-Point Library IP Mapping”

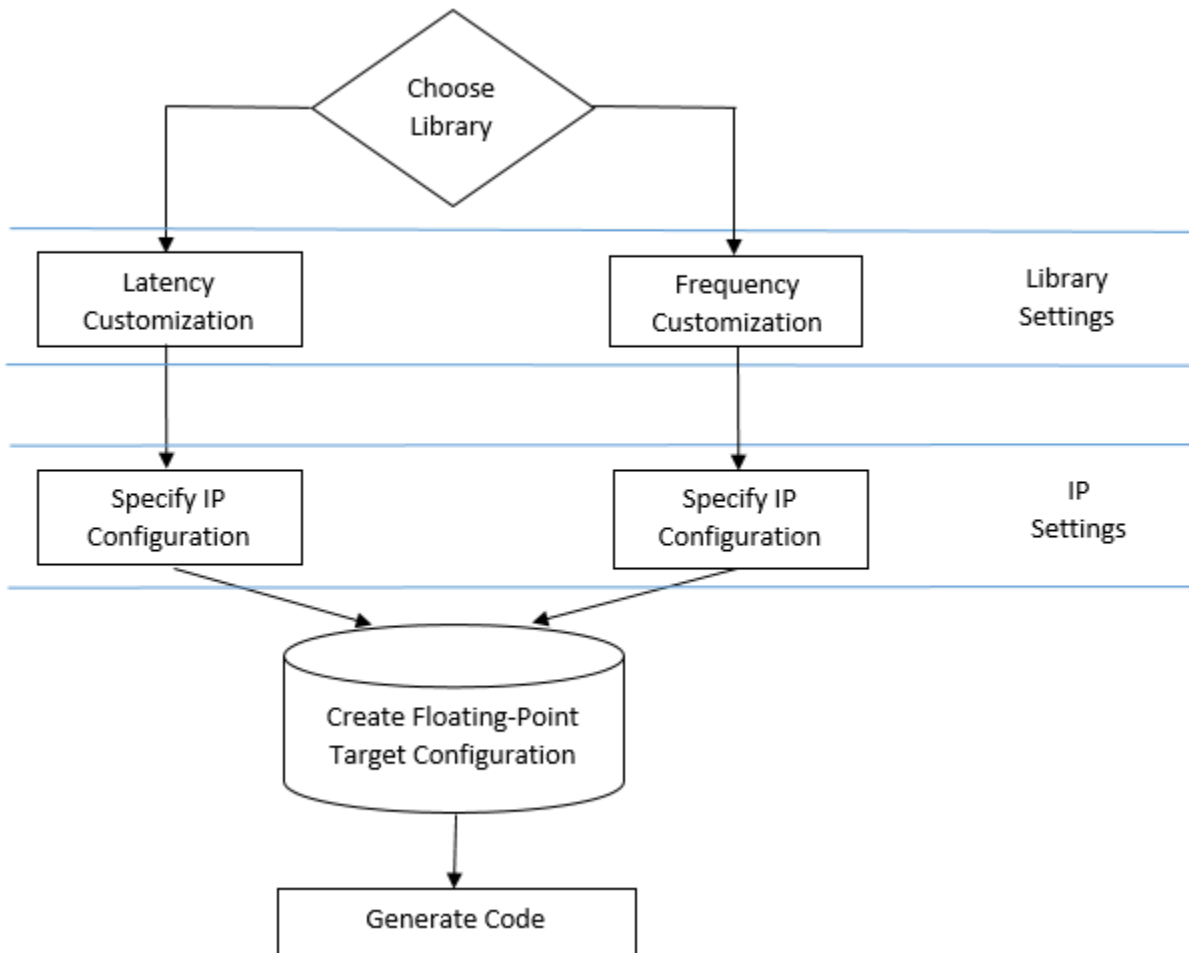
More About

- “Customize Floating-Point IP Configuration” on page 31-23

- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 31-34

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.

In this section...

“Customize the IP Latency with Target Frequency” on page 31-24

“Customize the IP Latency with Latency Strategy” on page 31-28

Customize the IP Latency with Target Frequency

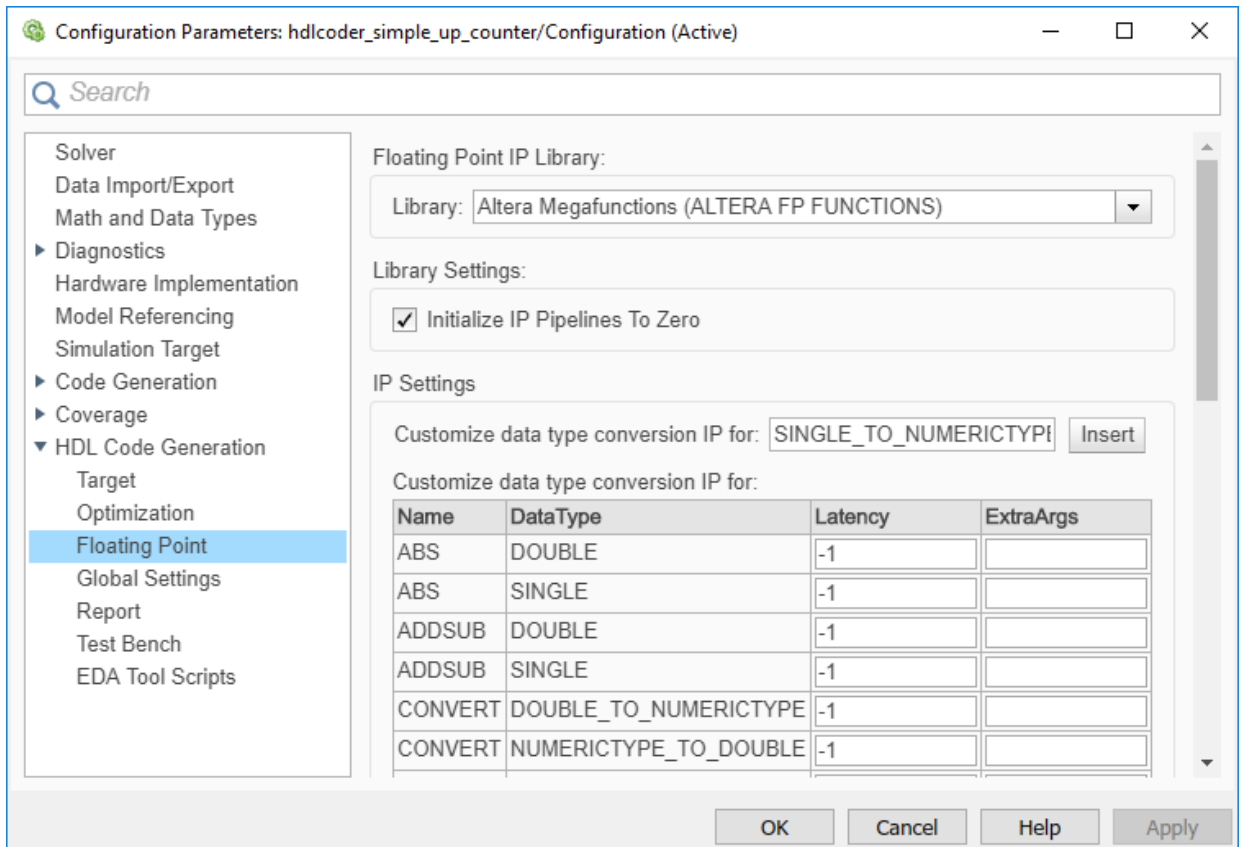
To specify the target frequency that you want the IP to achieve, use the **Altera Megafunctions (ALTERA FP Functions)** library. HDL Coder infers the latency of the IP based on the target frequency value. If you do not specify the target frequency, HDL Coder sets the target frequency to a default value of 200 MHz.

You can customize the IP latency by using the **Target Frequency** setting in the Configuration Parameters dialog box or the `TargetFrequency` property from the command line.

From the UI

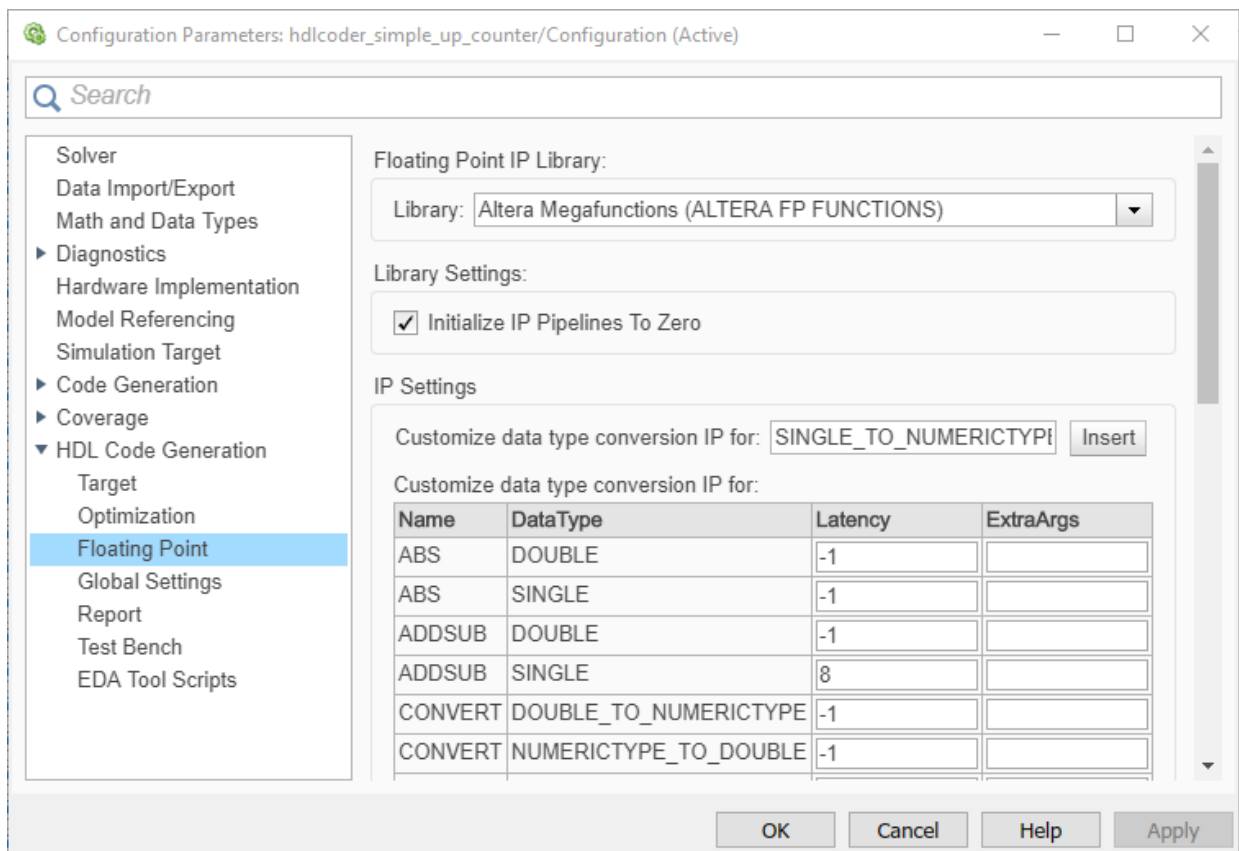
To customize the IP latency by using the target frequency setting:

- 1** Specify the library:
 - a** In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Click **Settings**.
 - b** On the **HDL Code Generation > Floating Point Target** pane, for **Library**, select **Altera Megafunctions (ALTERA FP Functions)**.



- Specify the target frequency: In the **Target** pane, for **Target Frequency (MHz)**, enter 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 a default value of 200 MHz.
- Specify the library settings: By using the **Initialize IP Pipelines to Zero** option, you can specify 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 **Initialize IP Pipelines to Zero** option set to true.
- Specify the IP settings: In the IP configuration table, you can optionally specify a custom latency and any additional settings specific to the IP.

- In the **Latency** column of the table, 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 of the table, you can specify additional settings specific to the IP.



5 Generate code: Click **Apply**. On the Simulink Toolstrip, click **Generate HDL Code**.

At the Command Line

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, for an `sfir_single` model, to create a floating-point target configuration for the Altera Megafunctions (ALTERA FP FUNCTIONS) library with the default settings, enter:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTERAFPFUNCTIONS');  
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
```

To see the default settings for the floating-point IP, enter `fpconfig`.

```
fpconfig =
```

```
FloatingPointTargetConfig with properties:
```

```
Library: 'ALTERAFPFUNCTIONS'  
LibrarySettings: [1x1 fpconfig.FrequencyDrivenMode]  
IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

- 2 Specify the target frequency: If you choose ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS) as the 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 library settings: Specify whether you want to initialize the pipeline registers in the IP to zero. Use the `InitializeIPipelinesToZero` property of the `fpconfig.LibrarySettings` function.

For example, to set the `InitializeIPipelinesToZero` property to false, enter:

```
fpconfig.LibrarySettings.InitializeIPipelinesToZero = false;
```

To see the library settings that you have applied, enter `fpconfig.LibrarySettings`.

```
ans =

    FrequencyDrivenMode with properties:

        InitializeIPipelinesToZero: 0
```

To avoid potential numerical mismatches in the HDL simulation, it is recommended to leave `InitializeIPipelinesToZero` set to `true`.

- 4 Specify the IP settings: With the `IPConfig` method, use the `Latency` and `ExtraArgs` to customize the latency of the IP and specify any 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.IPConfig.customize('ADDSUB', 'SINGLE', 'Latency', 8);
```

To see the IP settings that you have applied, enter `fpconfig.IPConfig`.

```
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

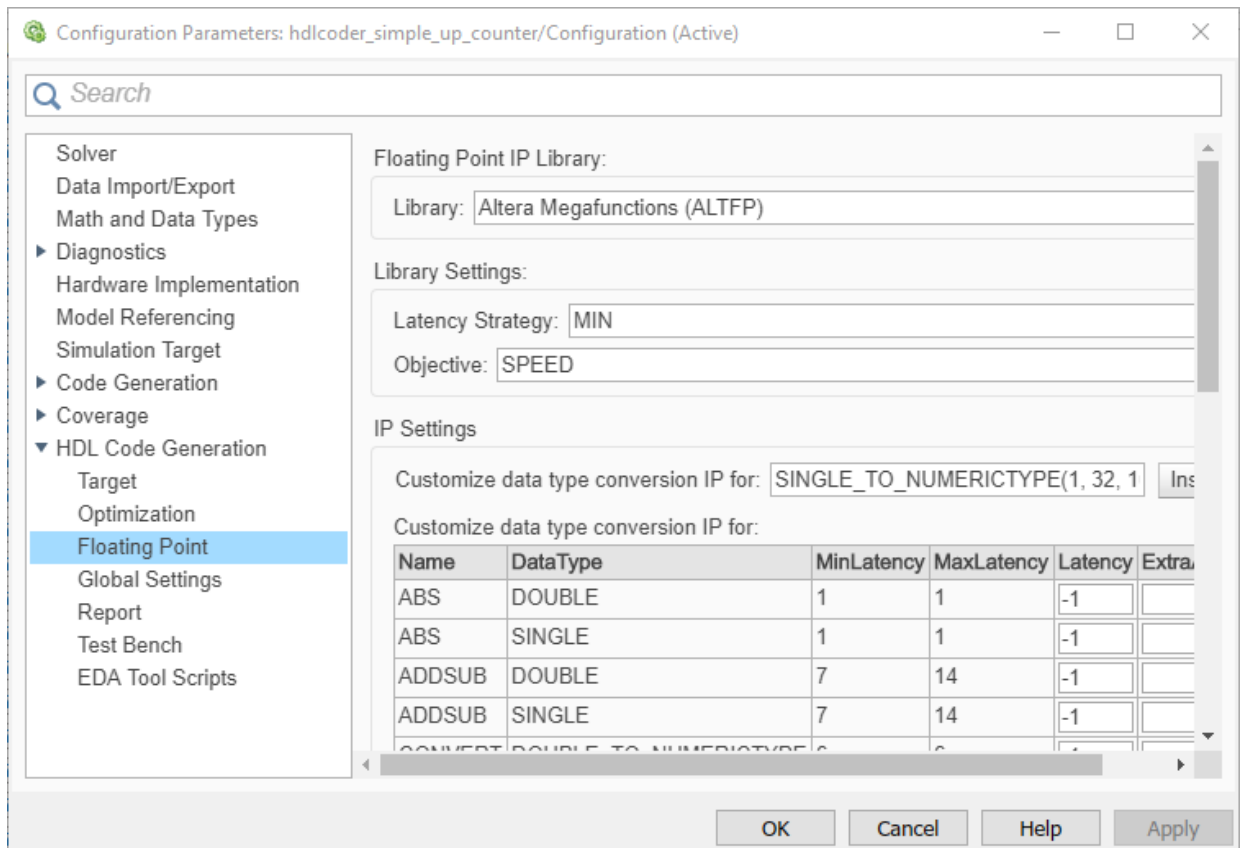
Simulink model to maximum or minimum latency. HDL Coder infers the latency of the IP from the latency strategy setting.

You can customize the IP latency in the Configuration Parameters dialog box or from the command line.

From the UI

To customize the IP latency with the latency strategy setting:

- 1 Specify the library: In the Configuration Parameters dialog box, on the **HDL Code Generation > Floating Point Target** pane, for **Library**, select ALTERA MEGAFUNCTION (ALTFP) or XILINX LOGICORE.

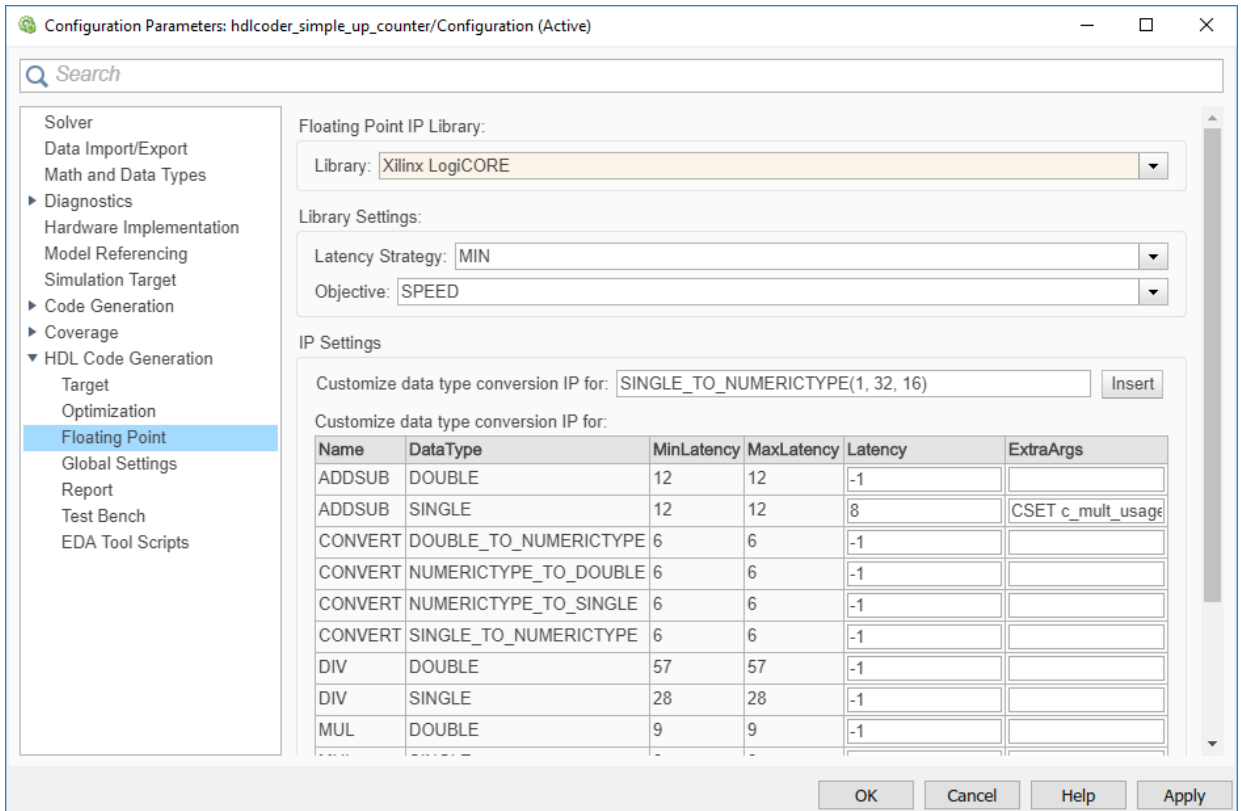


- 2 Specify the library settings: For **Latency Strategy**, specify whether to map your Simulink model to the minimum or maximum latency for the IP. For **Objective**, specify whether to optimize for speed or area.
- 3 Specify the IP settings: An IP configuration table appears that contains the IP types, and their maximum and minimum latencies. In the table, you can optionally specify a custom latency and any additional settings specific to the IP.
 - In the **Latency** column of the table, the default latency value of -1 means that the IP inherits the latency value from the library settings. To customize the latency of the IP that your Simulink blocks map to, enter a value for the latency.

For example, when mapping to the **ADDSUB** IP with Xilinx LogiCORE, if you specify a latency of 8, the latency of the IP changes to 8 instead of the default value of 12.

- In the **ExtraArgs** column of the table, specify any additional settings specific to the IP.

For example, when mapping to Xilinx LogiCORE IP, for **ExtraArgs**, you can specify the parameter **c_mult_usage** to control the DSP resources that you want to use. To learn more about the parameter usage and syntax, see the IP library documentation.



- 4 Generate code: Click **Apply**. On the Simulink Toolstrip, click **Generate HDL Code**.

From the Command Line

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, for an `sfir_single` model, to create a floating-point target configuration for the ALTERA MEGAFUNCTION (ALTFP) library with the default settings, enter:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('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 with the `Objective` and `LatencyStrategy` of the `fpconfig.LibrarySettings` function.

For example, to customize the ALTERA MEGAFUNCTION (ALTFP) library to use the maximum latency and objective as area, enter:

```
fpconfig.LibrarySettings.Objective = 'AREA';  
fpconfig.LibrarySettings.LatencyStrategy = 'MAX';
```

To see the library settings that you have applied, enter `fpconfig.LibrarySettings`.

```
ans =
```

```
LatencyDrivenMode with properties:
```

```
LatencyStrategy: 'MAX'  
Objective: 'AREA'
```

```
fpconfig is a variable of type hdlcoder.FloatingPointTargetConfig.
```

- 3 Specify the IP settings: With the `IPConfig` method, use the `Latency` and `ExtraArgs` to customize the latency of the IP and specify any 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 to specify the DSP resource usage with the `multusage` parameter:

```
fpconfig.IPConfig.customize('ADDSUB', 'SINGLE', 'Latency', 8, 'ExtraArgs', 'CSET c_
```

To see the IP settings that you have applied, enter `fpconfig.IPConfig`.

```
ans =
```

Name	DataType	MinLatency	MaxLatency	Latency
'ADDSUB'	'DOUBLE'	7	14	-1
'ADDSUB'	'SINGLE'	7	14	8
'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

Related Examples

- “FPGA Floating-Point Library IP Mapping”

More About

- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-14
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 31-34

HDL Coder Support for FPGA Floating-Point Library Mapping

In this section...

“Supported Blocks That Map to FPGA Floating-Point Target IP” on page 31-34

“Supported Blocks That Do Not Need to Map to FPGA Floating-Point Target IP” on page 31-37

“Limitations for FPGA Floating-Point Library Mapping” on page 31-38

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	✓		—

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Add	✓	✓	—
Bias	✓	✓	—
Compare To Constant	✓	✓	—
Compare To Zero	✓	✓	—
Data Type Conversion	✓	✓	<ul style="list-style-type: none"> 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, input bitwidth must be between 1 and 128 bits.
Decrement Real World	✓	✓	—
Discrete FIR Filter	✓	✓	—
Discrete Transfer Fcn	✓	✓	—
Discrete-Time Integrator	✓	✓	—
Divide	✓	✓	—
Dot Product	✓	✓	
Gain	✓	✓	—
Math Function	✓		<ul style="list-style-type: none"> Set the Function attribute in Block Parameters dialog box to either reciprocal, log or exp.
MinMax	✓	✓	—
Multiply-Add	✓	✓	—

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Product	✓	✓	<ul style="list-style-type: none"> Product block with more than two inputs is not supported.
Product of Elements	✓	✓	<ul style="list-style-type: none"> The Architecture in HDL Block Properties must be set to Tree.
Reciprocal Sqrt	✓		—
Relational Operator	✓	✓	—
Sqrt	✓	✓	—
Subtract	✓	✓	—
Sum	✓	✓	<ul style="list-style-type: none"> Sum block with 3 ports is not supported. The block cannot have more than two inputs.
Sum of Elements	✓	✓	<ul style="list-style-type: none"> The Architecture in HDL Block Properties must be set to Tree.
Trigonometric Function	✓		<ul style="list-style-type: none"> 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 Incompatibility SinCosPipeline global property using <code>hdlset_param</code>.
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
- Switch block with control input other than $u2 \sim \theta$.
- 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, FPGA Turnkey, 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”

More About

- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-14
- “Customize Floating-Point IP Configuration” on page 31-23

Synthesis Objective to Tcl Command Mapping

In this section...

"Altera Quartus II" on page 31-39

"Xilinx Vivado 2014.4" on page 31-40

"Xilinx ISE 14.7 with PlanAhead" on page 31-40

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	<pre>set_global_assignment -name OPTIMIZATION_TECHNIQUE "Area" set_global_assignment -name FITTER_EFFORT "Standard Fit"</pre>

Synthesis objective	Tcl Commands
Compile Optimized	<pre>set_global_assignment -name OPTIMIZATION_TECHNIQUE "Balanced" set_global_assignment -name FITTER_EFFORT "Fast Fit"</pre>
Speed Optimized	<pre>set_global_assignment -name OPTIMIZATION_TECHNIQUE "Speed" set_global_assignment -name FITTER_EFFORT "Standard Fit"</pre>

Xilinx Vivado 2014.4

If your tool version is different, the Tcl commands are slightly different.

Synthesis objective	Tcl Commands
Area Optimized	<pre>set_property strategy {Vivado Synthesis Defaults} [get_runs synth_1] set_property strategy "Area_Explore" [get_runs impl_1]</pre>
Compile Optimized	<pre>set_property strategy "Flow_RuntimeOptimized" [get_runs synth1] set_property strategy "Flow_Quick" [get_runs impl_1]</pre>
Speed Optimized	<pre>set_property strategy {Vivado Synthesis Defaults} [get_runs synth_1] set_property strategy "Performance_Explore" [get_runs impl_1]</pre>

Xilinx ISE 14.7 with PlanAhead

If your tool version is different, the Tcl commands are slightly different.

Synthesis objective	Tcl Commands
Area Optimized	<pre>set_property strategy "AreaReduction" [get_runs synth_1] set_property strategy "MapCoverArea" [get_runs impl_1]</pre>
Compile Optimized	<pre>set_property strategy "{XST Defaults}" [get_runs synth_1] set_property strategy "{ISE Defaults}" [get_runs impl_1]</pre>
Speed Optimized	<pre>set_property strategy "TimingWithIOBPacking" [get_runs synth_1] set_property strategy "MapTiming" [get_runs impl_1]</pre>

See Also

Related Examples

- “Getting Started with the HDL Workflow Advisor” on page 31-2
- “HDL Workflow Advisor Tasks” on page 37-2

Run HDL Workflow with a Script

In this section...

- | |
|--|
| <ul style="list-style-type: none">“Export an HDL Workflow Script” on page 31-43“Enable or Disable Tasks in HDL Workflow Script” on page 31-43“Run a Single Workflow Task” on page 31-43“Import an HDL Workflow Script” on page 31-44“Generic ASIC/FPGA Workflow Script Example” on page 31-44“FPGA-in-the-Loop Script Example” on page 31-46“FPGA Turnkey Workflow Script Example” on page 31-48“IP Core Generation Workflow Script Example” on page 31-51“Simulink Real-Time FPGA I/O Workflow Example” on page 31-54 |
|--|

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.

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)
- FPGA Turnkey
- 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.

Enable or Disable Tasks in HDL Workflow Script

To disable all workflow tasks, update the workflow configuration object with the `hdlcoder.WorkflowConfig.clearAllTasks` method.

To reenable all workflow tasks, update the workflow configuration object with the `hdlcoder.WorkflowConfig.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 `hdlcoder.WorkflowConfig.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', 'ffgl761');
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','Generi

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% 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.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.m
```

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\h
%   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-

```

```
% 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
```

FPGA Turnkey Workflow Script Example

This example shows how to configure and run an exported HDL workflow script.

This script is an FPGA Turnkey workflow script that targets a Xilinx Virtex 5 development board. It uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:24:32 on 08/07/2015
% Parameter Values:
%   Filename   : 'S:\turnkey_workflow_example.m'
%   Overwrite  : true
%   Comments   : true
%   Headers    : true
%   DUT        : 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA'

%% Load the Model
load_system('hdlcoderUARTServoControllerExample');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'HDLSubsystem', 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisTool', 'Xilinx ISE');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolChipFamily', 'Virtex5');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolDeviceName', 'xc5vsx50t');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolPackageName', 'ff1136');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'TargetPlatform', 'Xilinx Virtex-5 ML506 development board');
hdlset_param('hdlcoderUARTServoControllerExample', 'Workflow', 'FPGA Turnkey');

% Set Inport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterface', 'RS-232 Serial Port Rx');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterfaceMapping', '[0]');

% Set Output HDL parameters
```

```
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_txd', ...
    'IOInterface', 'RS-232 Serial Port Tx');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_txd', ...
    'IOInterfaceMapping', '[0]');

% Set Output HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/version', ...
    'IOInterface', 'LEDs General Purpose [0:7]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/version', ...
    'IOInterfaceMapping', '[0:3]');

% Set Output HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/pwm_output', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/pwm_output', ...
    'IOInterfaceMapping', '[0]');

% Set Output HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug1', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug1', ...
    'IOInterfaceMapping', '[1]');

% Set Output HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug2', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug2', ...
    'IOInterfaceMapping', '[2]');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx ISE', ...
    'TargetWorkflow','FPGA Turnkey');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskPerformLogicSynthesis = true;
hWC.RunTaskPerformMapping = true;
```

```

hWC.RunTaskPerformPlaceAndRoute = true;
hWC.RunTaskGenerateProgrammingFile = true;
hWC.RunTaskProgramTargetDevice = false;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set Properties related to Perform Mapping Task
hWC.SkipPreRouteTimingAnalysis = true;

% Set Properties related to Perform Place and Route Task
hWC.IgnorePlaceAndRouteErrors = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
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 `turnkey_workflow_example.m`, at the command line, enter:

```
turnkey_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"');

% 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"');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface', 'External Port');

```



```
% Set Output 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"');

%% 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 Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskGenerateSoftwareInterfaceModel = 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.m
```

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 I0333-325K 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\h
%   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 I0333-325K');
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
```

```
% 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.WorkflowConfig.clearAllTasks` |
`hdlcoder.WorkflowConfig.setAllTasks` | `hdlcoder.runWorkflow`

Classes

`hdlcoder.WorkflowConfig`

Related Examples

- “Getting Started with the HDL Workflow Advisor” on page 31-2
- “HDL Workflow Advisor Tasks” on page 37-2

Simscape to HDL Workflow

- “Generate HDL Code from Simscape Models” on page 32-2
- “Deploy Simscape™ Plant Models to Speedgoat FPGA I/O Modules” on page 32-14
- “Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model” on page 32-27
- “Troubleshoot Conversion of Simscape™ Permanent Magnet Synchronous Motor to HDL-Compatible Simulink Model” on page 32-38
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-55
- “Improve Sampling Rate of HDL Implementation Model Generated from Simscape Algorithm” on page 32-64

Generate HDL Code from Simscape Models

This example uses a halfwave rectifier model to illustrate how you can develop your plant model in Simscape™ and use Simscape HDL Workflow Advisor to generate HDL code for your model.

Why Generate HDL Code?

To perform hardware-in-the-loop (HIL) simulation with smaller timesteps and increased accuracy, you can deploy the plant models to the FPGAs on board the Speedgoat I/O modules. 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 generated code onto the FPGA platforms. Using this capability, you can model and deploy complex physical systems in Simscape that previously took long time to model by using Simulink™ blocks.

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. The example models include:

- Boost Converter
- Bridge Rectifier
- Buck Converter
- Halfwave Rectifier
- Three Phase Rectifier
- Two Level Converter Ideal
- Two Level Converter Igbt

All examples are prefixed with `sschdlex` and postfixed with `Example`. For example, to open the Boost Converter model, in the MATLAB™ command window, enter:

```
load_system('sschdlexBoostConverterExample')  
open_system('sschdlexBoostConverterExample/Simscape_system')
```

Restrictions for HDL Code Generation from Simscape Models

HDL Coder™ does not support code generation from Simscape networks that contain:

- Events

- Mode charts
- Delays
- Runtime parameters
- Periodic sources
- Simscape™ Multibody™ blocks
- Simscape Electrical Specialized Power Systems blocks

Guidelines for Modeling Simscape for HDL Compatibility

1. Create a Simscape model by using switched linear blocks. Switched linear blocks are blocks such as diodes and switches. These blocks are 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. Add Simulink-PS Converter blocks at the input ports and PS-Simulink Converter blocks at the output ports.

2. Configure the solver options for HDL code generation by using a Solver Configuration block. In the block parameters of this block:

- Select **Use local solver**.
- Use Backward Euler as the **Solver type**.
- Specify a discrete sample time, T_s .

3. Enclose the blocks inside a Subsystem and provide the test inputs.

4. Configure the model for HDL code generation by running the `hdlsetup` function. `hdlsetup` configures the solver settings such as using a fixed-step solver, specifies the simulation start and stop times, and so on. To run the command for your `current_model`:

```
hdlsetup('current_model')
```

5. Verify Simscape model compatibility by using the `simscape.findNonLinearBlocks` function. This function detects nonlinear blocks in your Simscape model. Provide the path to your Simscape model as an argument to this function. It returns the names of nonlinear blocks.

For example: To verify presence of nonlinear blocks in Half Wave Rectifier Model, in the MATLAB command window, enter:

```
simscape.findNonLinearBlocks('sschdlexHalfWaveRectifierExample')
```

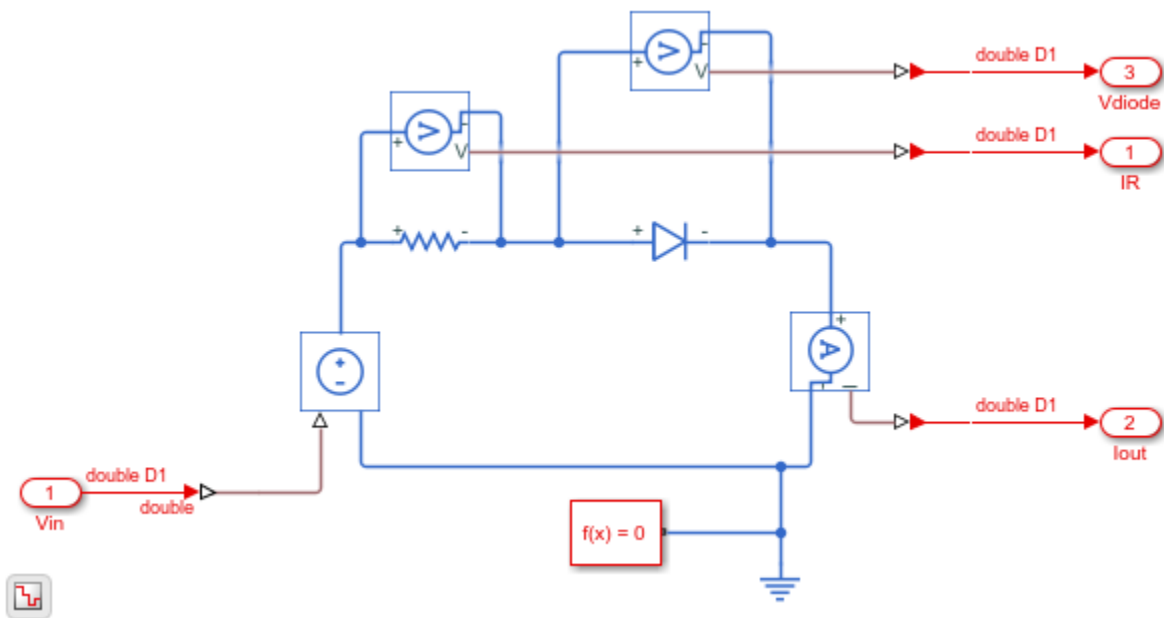
The Halfwave Rectifier Model

To open the half-wave rectifier model, in the MATLAB Command Window, enter:

```
open_system('sschdlexHalfWaveRectifierExample')
```

Save this model locally as HalfWaveRectifier_HDL to run the workflow.

```
load_system('HalfWaveRectifier_HDL')
open_system('HalfWaveRectifier_HDL/Simscape_system')
set_param('HalfWaveRectifier_HDL', 'SimulationCommand', 'update');
```

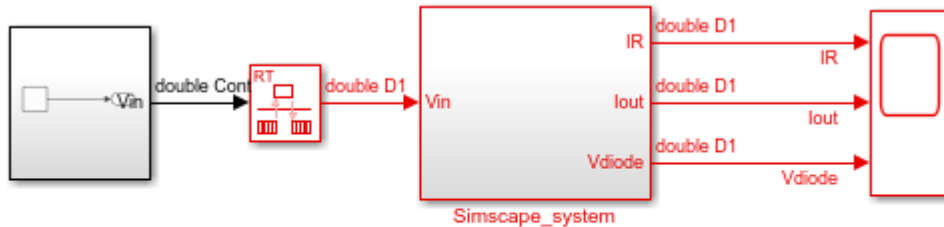


The Simscape model uses switched linear blocks such as Diodes and Resistors to model the design. The model has Simulink-PS Converter blocks at the input port and PS-Simulink converter blocks at the output ports. To verify that you configured the solver settings correctly, open the Solver Configuration block.

At the top level of the model, you see a Simscape_system block that 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 input stimulus and the output from the model, connect the Sine Wave input to the Scope block.

```
open_system('HalfWaveRectifier_HDL')
```



Copyright 2018 The MathWorks, Inc.

To configure the half-wave rectifier model for HDL compatibility, in the MATLAB command window, enter:

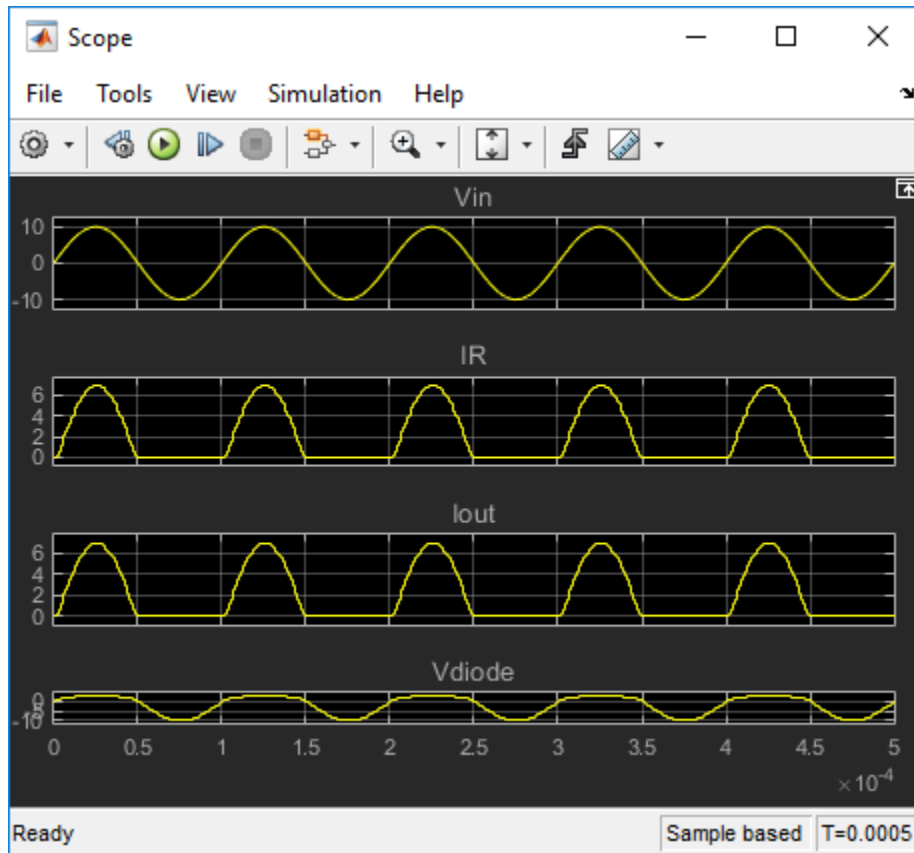
```
hdlsetup('HalfWaveRectifier_HDL')
```

Simulate and Verify Functionality of Simscape Algorithm

To see the simulation results, simulate the model and then open the Scope block.

```
sim('HalfWaveRectifier_HDL')
```

This figure shows simulation results with the sine wave input and the outputs from Simscape_system.



Open Simscape HDL Workflow Advisor

To generate an HDL implementation model from which you can generate code, use the Simscape HDL Workflow Advisor. To open the Advisor, run this command:

```
sschdladvisor('HalfWaveRectifier_HDL')
```

Updating Model Advisor cache...

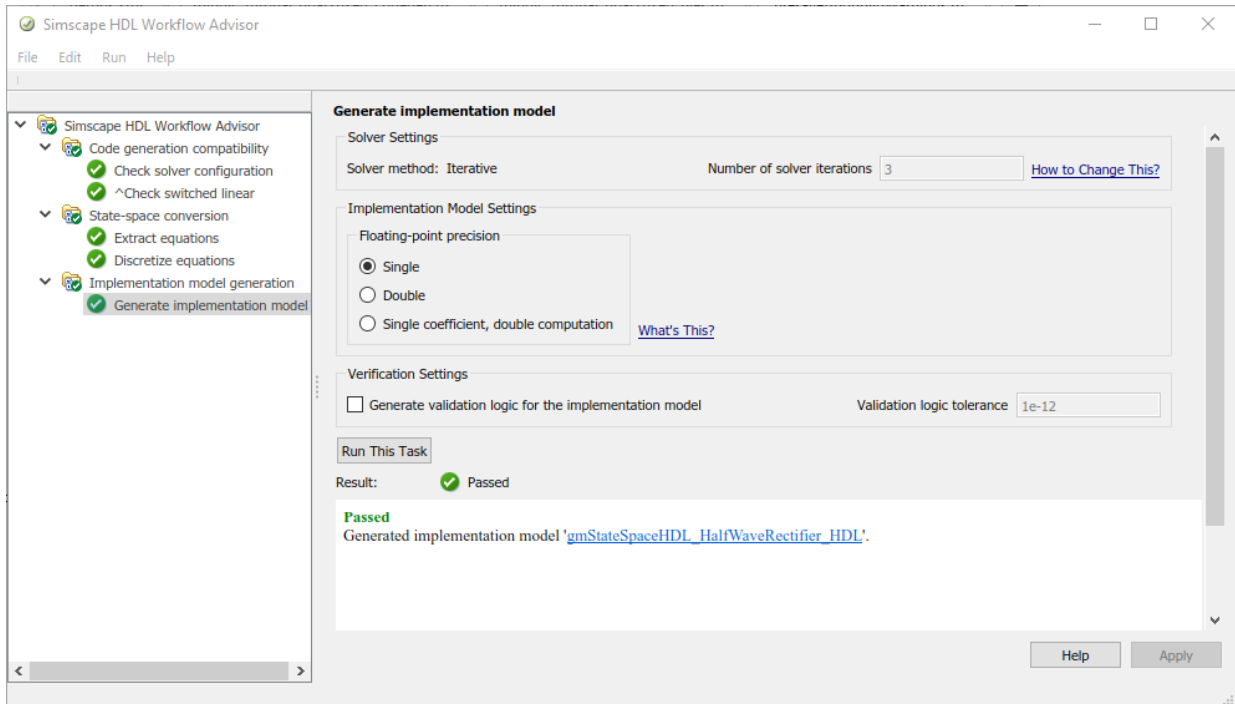
Model Advisor cache updated. For new customizations, to update the cache, use the Advisor.

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.

Run Simscape HDL Workflow Advisor

To run the workflow, in the Simscape HDL Workflow Advisor, right-click the **Generate implementation model** task and select **Run to Selected Task**.



If the task passes, you see a link to the implementation model.

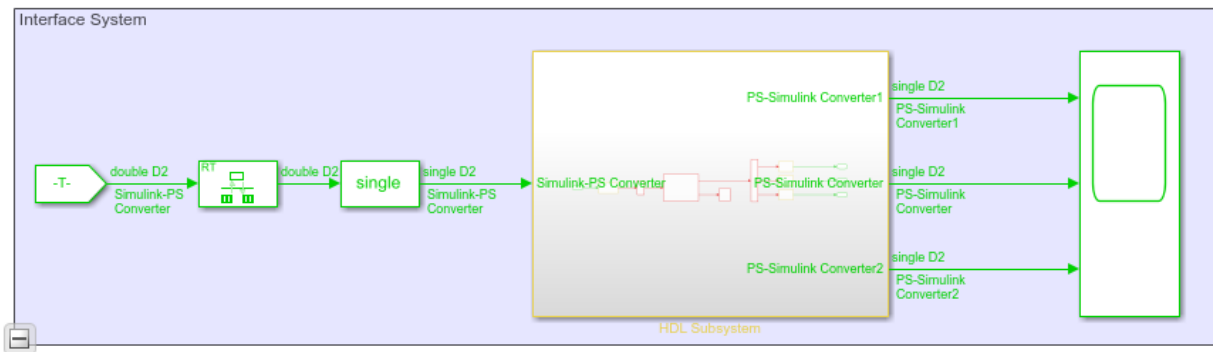
In some cases, your Simscape algorithm may not be compatible for generating an implementation model using the Simscape HDL Workflow Advisor. In such cases, running certain tasks in the Advisor can result in the task to fail. To learn how you can make it HDL compatible, see

- Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model.
- Troubleshoot Conversion of Simscape Permanent Magnet Synchronous Motor to HDL-Compatible Simulink Model.

Open HDL Implementation Model

To see the implementation model, in the **Generate implementation model task**, click the link.

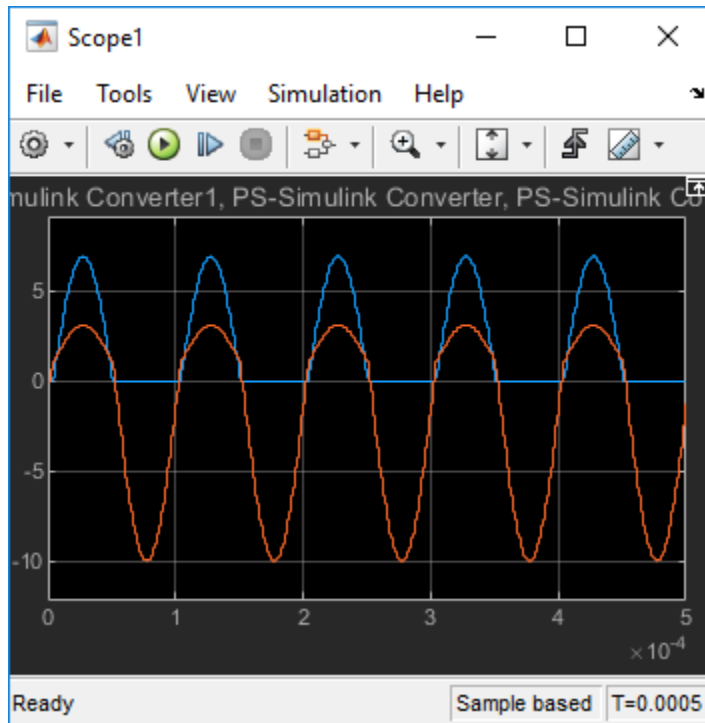
```
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL')
set_param('gmStateSpaceHDL_HalfWaveRectifier_HDL','SimulationCommand','Update')
```



The model contains two subsystems. The Subsystem block contains the Simscape algorithm that you modeled. From and Goto blocks inside this Subsystem provide the same Sine Wave input to the HDL Subsystem.

The HDL Subsystem models the state-space representation that you generated from the Simscape model. 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/HDL Subsystem/HDL Algorithm')
```

The simulation results from the HDL implementation model matches that of the original plant model. Therefore, we can verify that the plant simulation model is correctly transformed into an HDL implementation model.

HDL code is generated for the HDL Subsystem block inside this model.

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', 'FloatingPointTargetConfiguration', 'LatencyStrategy', 'MIN') ...
);
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'MaskParameterAsGeneric', 'on');
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'Oversampling', 60);
```

```
% Set SubSystem HDL parameters
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL/HDL Subsystem', 'FlattenHierarchy')

hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL/HDL Subsystem/HDL Algorithm/Mode S
```

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, for the model transformation advisor to allocate enough delays for the operators inside the feedback loops, the model uses clock-rate pipelining in conjunction with a large value for the **Oversampling factor**. An **Oversampling factor** of 100 and the clock-rate pipelining optimization is saved on this model.

For more information, see:

- Clock-Rate Pipelining
- Oversampling Factor
- Allocate Sufficient Delays for Floating-Point Operations

Before you generate HDL code, it is recommended to 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](#).

Run these commands to save validation model generation settings on your Simulink model:

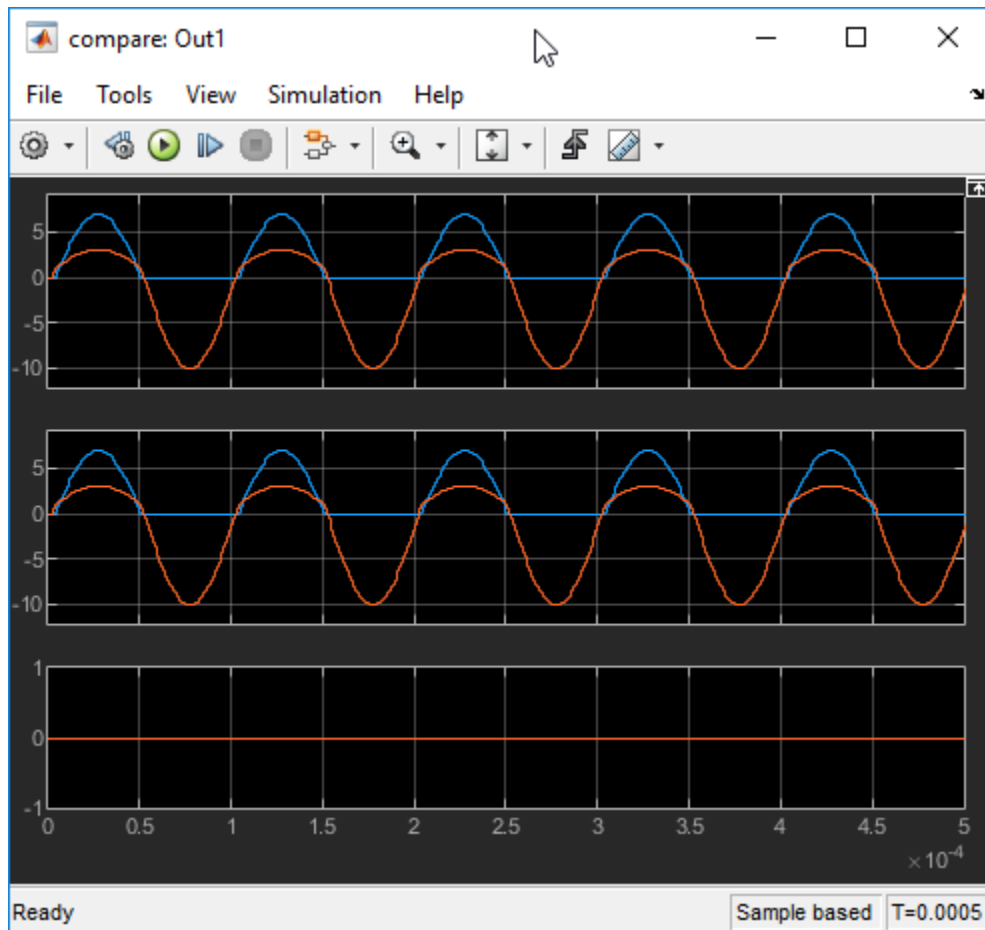
```
HDLmodelName = 'gmStateSpaceHDL_HalfWaveRectifier_HDL';
hdlset_param(HDLmodelName, 'TargetDirectory', 'C:/Temp/hdlsrc');
hdlset_param(HDLmodelName, 'GenerateValidationModel', 'on');
```

To generate HDL code, run this command:

```
makehdl('gmStateSpaceHDL_HalfWaveRectifier_HDL/HDL Subsystem');
```

The generated HDL code and validation model are saved in C:/Temp/hdlsrc directory. The generated code is saved as HDL_Subsystem_tc.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_vn1 Subsystem of the validation model. Then, open the Assert_Out1 block. To see the simulation results after HDL code generation, open the Compare: Out1 Scope block:



The top graph represents the output of the generated model, and the middle graph represents the output of the implementation model. Since the output generated by both the models are exactly matching, the error between them is zero, which is represented in the last graph.

Optionally, you can deploy the HDL code on a Hardware platform. For more information, see [Deploy Simscape plant models to Speedgoat FPGA IO modules](#).

See Also

Functions

`checkhdl` | `makehdl`

More About

- [“Getting Started with Simscape Electrical” \(Simscape Electrical\)](#)
- [“Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model”](#) on page 32-27
- [“Validate HDL Implementation Model to Simscape Algorithm”](#) on page 32-55

Deploy Simscape™ Plant Models to Speedgoat FPGA I/O Modules

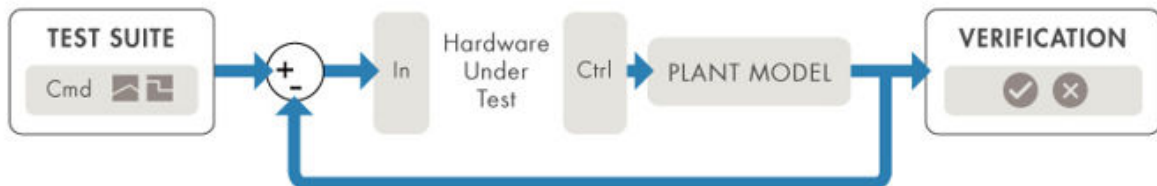
This example shows how you can deploy the Simscape plant models on Speedgoat FPGA I/O modules by using the HDL Workflow Advisor. This workflow is a two-step process.

- 1 Develop the Simscape model and convert it into an implementation model by using the Simscape HDL Workflow Advisor. HDL code is generated from this implementation model. For more information, see [Generate HDL Code from Simscape Models](#).
- 2 Deploy HDL code to a Speedgoat I/O module by using the HDL Workflow Advisor.

Why Deploy a Simulink Model to Speedgoat FPGA Modules

You can use the HDL Workflow Advisor to deploy the Simulink™ model to Speedgoat FPGA I/O modules. Simulating the plant model on the FPGA provides:

- **Real-time Simulation:** Hardware-in-the-loop provides real-time simulation of your Simscape plant model.



- **Hardware Acceleration:** The speed of simulating physical systems increases by implementing it on hardware as reconfigurable FPGAs provide rapid hardware prototyping. You can use this capability to model complex physical systems.

Set Up and Configuration

To deploy the Simscape plant models onto Speedgoat FPGA modules:

1. **Install Xilinx Vivado®**

Speedgoat FPGA IO333-325K uses Xilinx Vivado. If it is not already present, install Xilinx Vivado v2018.2. Then, set the tool path to the installed Xilinx Vivado 2018.2 executable. To set the tool path, use the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2018.2\bin\vi
```

2. Set Up I/O Module

To run the simulation of the Simscape plant model in real time on hardware, you must set up the I/O module. For information on setting up the I/O module, see Xilinx HDL Software for Speedgoat I/O Hardware.

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.
- Generating HDL code, a test bench, and scripts to build and run the code and test bench.
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Completing the automated workflows for deployment on hardware platforms such as System-on-Chip(SoC), FPGAs, and Speedgoat I/O modules.

This example shows how to use the HDL Workflow Advisor to deploy HDL code on Speedgoat IO333-325K module that uses Xilinx Vivado. For example, to open the HDL Workflow Advisor for a Subsystem inside the model, enter:

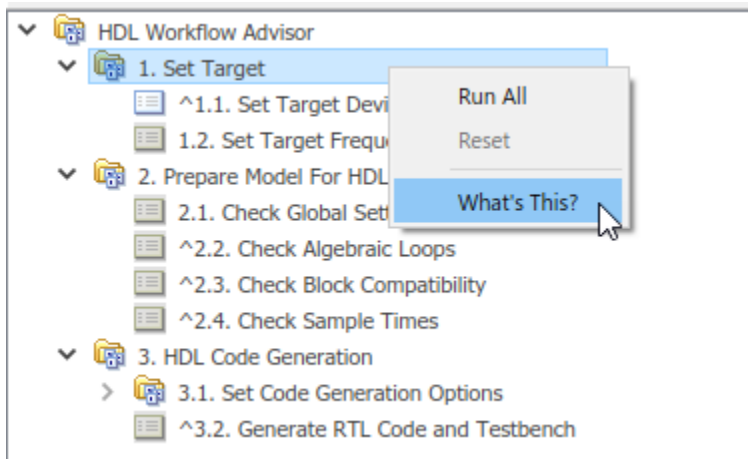
```
load_system('sschdlexTwoLevelConverterIgbtExample')  
hdladvisor('sschdlexTwoLevelConverterIgbtExample/Simscape_system')
```

For more information, see `hdladvisor`.

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 the 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 depends 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.

To learn more about each individual task, right-click that task, and select **What's This?**.



For more information, see Getting Started with the HDL Workflow Advisor.

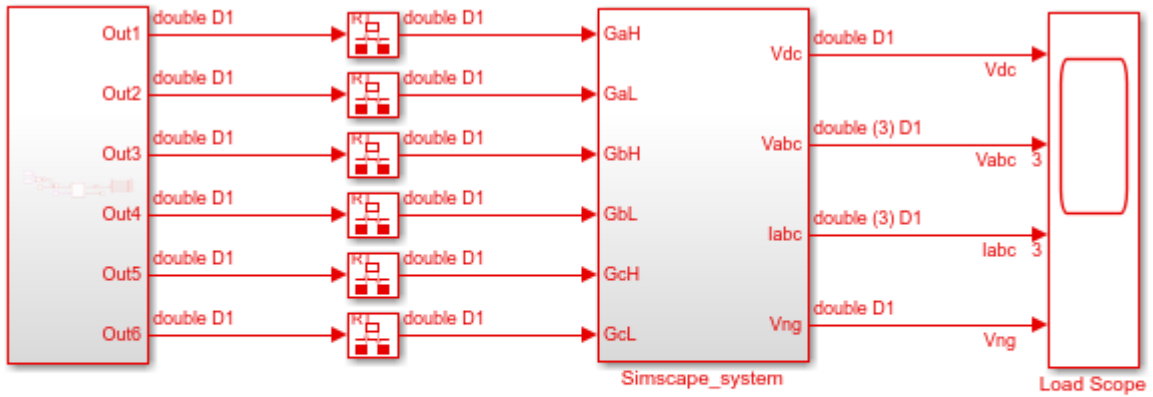
Two Level Ideal Converter Model

This example uses a Two-Level Ideal converter Simscape plant model. To open this model, enter:

```
open_system('sschdlexTwoLevelConverterIdealExample')
```

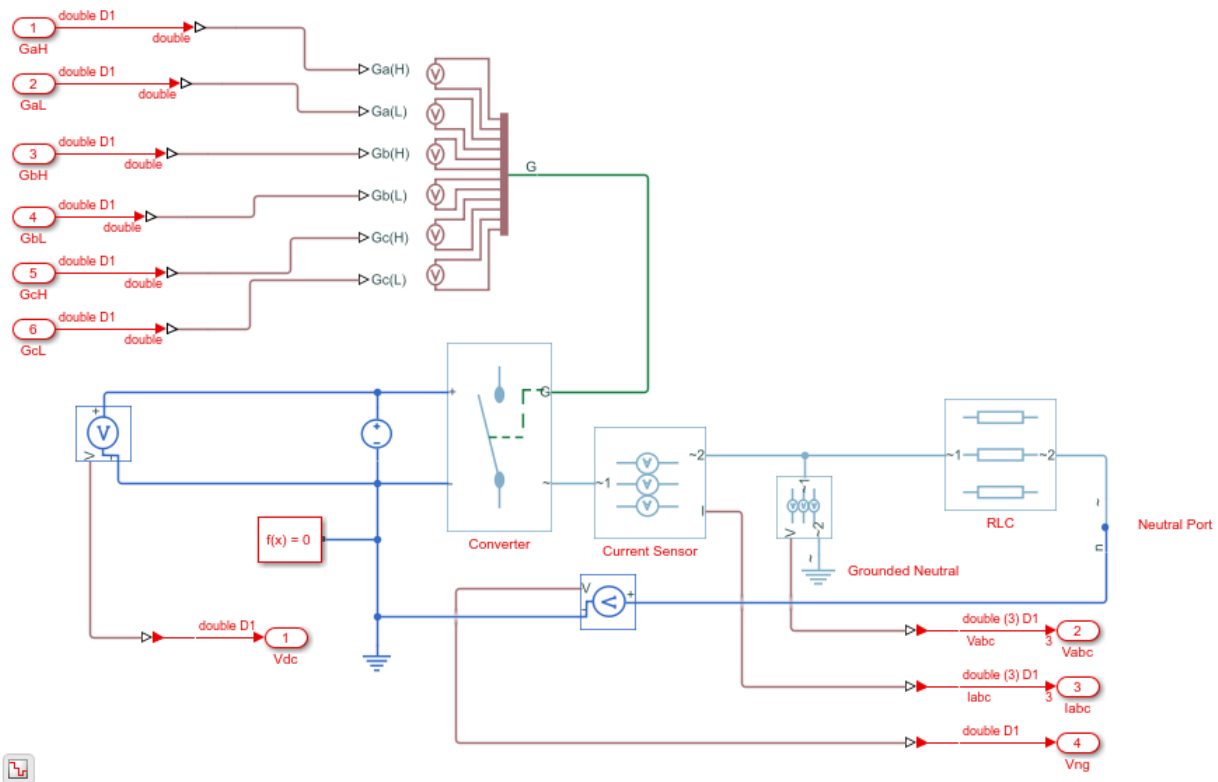
Save this model locally as TwoLevelConverter_HDL.slx to run this workflow.

```
open_system('TwoLevelConverter_HDL')
set_param('TwoLevelConverter_HDL','SimulationCommand','update')
```



Copyright 2018 The MathWorks, Inc.

```
open_system('TwoLevelConverter_HDL/Simscape_system')
```



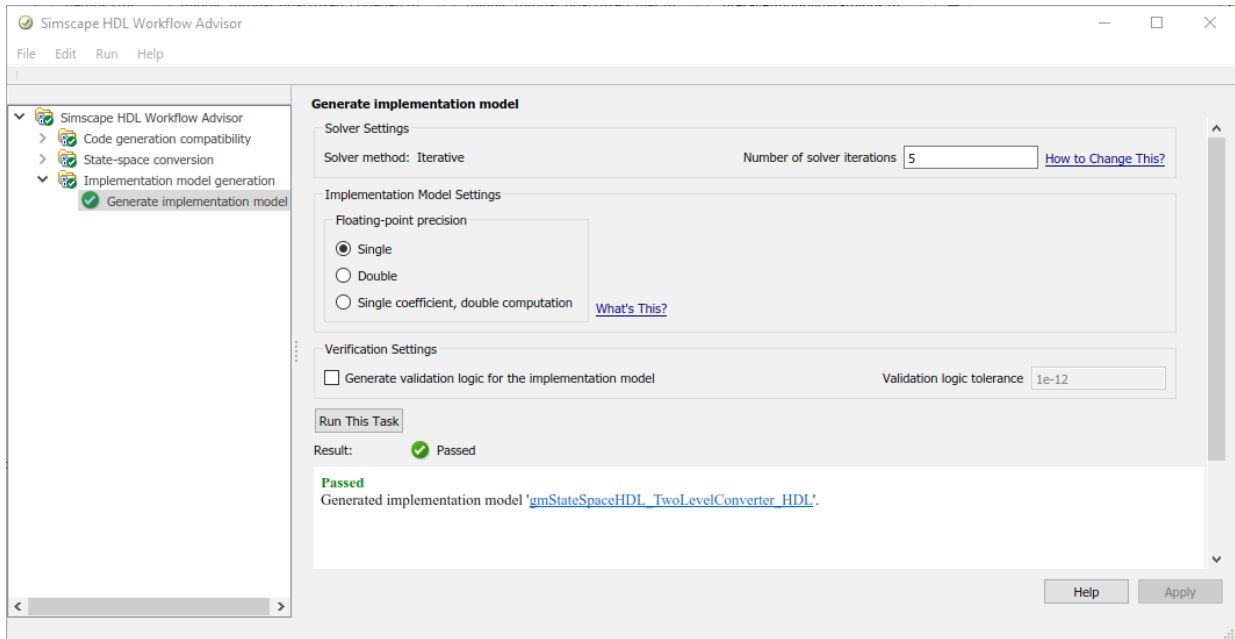
The Simscape subsystem receives six-switch controlling pulses as input. The Simscape subsystem acts as a generator that uses a two-level, carrier-based PWM method to:

- 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.

Generate HDL Implementation Model

To generate an implementation model, use the Simscape HDL Workflow Advisor. Enter:

```
sschdladvisor('TwoLevelConverter_HDL')
```



To generate the implementation model, in the Simscape HDL Workflow Advisor, keep the default settings for all the tasks, and then run the tasks. You see a link to the model in the **Generate implementation model** task.

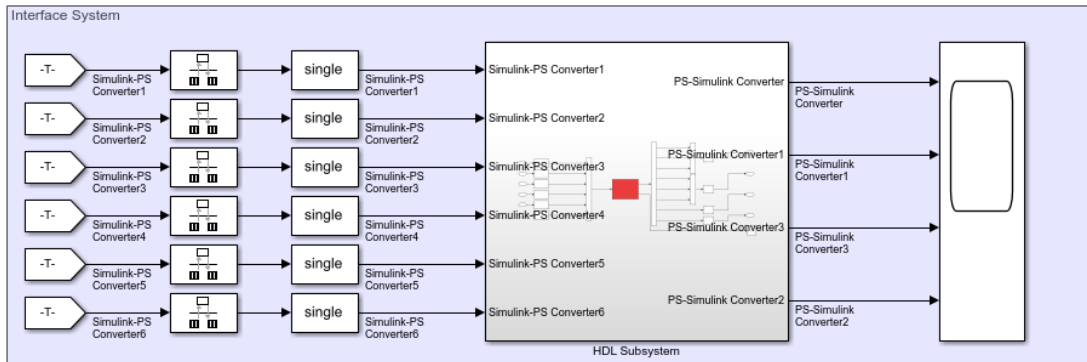
To learn more about the Simscape HDL Workflow Advisor, see:

- `sschdladvisor`
- Generate HDL code from Simscape Models

The Implementation Model

To open the implementation model, enter:

```
open_system('gmStateSpaceHDL_TwoLevelConverter_HDL')
```



The model contains two subsystems. The HDL Subsystem models the state-space representation that you generated from the Simscape model. The ports of this subsystem use the same name as the Simulink-PS Converter and PS-Simulink Converter blocks that you use 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. From and Goto blocks inside this subsystem provide the same input as that of the original model to the HDL Subsystem.

Deploy Two Level IGBT Converter Model to Speedgoat IO333-325K Module

This example shows how to deploy the implementation model of Two Level IGBT Converter to Speedgoat IO333-325K FPGA module by using the HDL Workflow Advisor. The Speedgoat IO333 FPGA module uses Xilinx Vivado and IP Core Generation Infrastructure. Before you run the Workflow Advisor, make sure that you have specified the path to the installed Xilinx Vivado executable.

1. Open HDL Workflow Advisor

To open the HDL Workflow Advisor for the Implementation model, enter:

```
hdladvisor('gmStateSpaceHDL_TwoLevelConverter_HDL/HDL Subsystem')
```

2. In **Set Target Device and Synthesis Tool** task, set these parameters and select **Run This Task**:

- **Target workflow** as Simulink Real-Time FPGA I/O
- **Target platform** as Speedgoat IO333-325K

- **Synthesis tool** as Xilinx Vivado

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: Simulink Real-Time FPGA I/O

Target platform: Speedgoat IO333-325K Launch Board Manager

Synthesis tool: Xilinx Vivado Tool version: 2017.4 Refresh

Family: Kintex7 Device: xc7k325t

Package: ffg900 Speed: -2

Project folder: hdl_prj Browse...

3. In **Set Target Reference Design** task, select a value of x4 for the parameter PCIe lanes, and select **Run This Task**.

1.2. Set Target Reference Design

Analysis (^Triggers Update Diagram)

Set target reference design options

Input Parameters

Reference design: Speedgoat IO333-325K-06-V001

Reference design tool version: 2017.4 Ignore tool version mismatch

Reference design parameters

Parameter	Value
PCIe lanes	X4

4. In **Set Target Interface** task, map the Input and Output single data type ports to PCIe Interface and select **Run This Task**.

1.3. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Processor/FPGA synchronization:

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
Simulink-PS Converte...	Inport	single	PCIe Interface	x"100"
Simulink-PS Converte...	Inport	single	PCIe Interface	x"108"
Simulink-PS Converte...	Inport	single	PCIe Interface	x"104"
Simulink-PS Converte...	Inport	single	PCIe Interface	x"10C"
Simulink-PS Converte...	Inport	single	PCIe Interface	x"110"
Simulink-PS Converte...	Inport	single	PCIe Interface	x"114"
PS-Simulink Converter	Output	single (3)	PCIe Interface	x"120"
PS-Simulink Converte...	Output	single (3)	PCIe Interface	x"140"
PS-Simulink Converte...	Output	single	PCIe Interface	x"118"

5. In **Set Target Frequency** task, select a target frequency that is within the range. If the target frequency is set to higher values, it results in a failure to generate the bitstream when you run task **Build FPGA Bitstream**. This example has **Target Frequency** set to 50 MHz.

1.4. Set Target Frequency

Analysis

Set Target Frequency

Input Parameters

Target Frequency (MHz):

Default (MHz):

Frequency Range (MHz):

6. Right-click **Generate RTL Code and IP Core** task and select **Run to Selected Task**. This step generates a warning if the model uses vector data types. Click the link in the warning, select **Scalarize vector ports**, and rerun the task.
7. 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.

4.1. Create Project

Analysis

Create project for embedded system tool

Input Parameters

Embedded system tool:

Project folder:

Synthesis objective:

Enable IP caching

Result: ✔ Passed

Passed Create Project.

```

Task "Create Project" successful.
Generated logfile: hdl\_prj\hdlsrc\qmStateSpaceHDLTwoLevelConverter\HDL\workflow\_task\_CreateProject.log
Generating Xilinx Vivado with IP Integrator project: hdl\_prj\vivado\_ip\_prj\vivado\_prj.xpr
***** Vivado v2017.4 (64-bit)
**** SW Build 2086221 on Fri Dec 15 20:55:39 MST 2017
**** IP Build 2085800 on Fri Dec 15 22:25:07 MST 2017
** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.

source vivado_create_prj.tcl
# create_project vivado_prj {} -part xc7k325tffg900-2 -force
# set_property target_language VHDL [current_project]
# set defaultRepoPath ./ipcore\

```

8. When the **Generate Simulink Real-Time interface** task passes, you see a link to open the Simulink Real-Time Interface Model. Select this link.

5.1. Generate Simulink Real-Time interface

Analysis

Generate Simulink Real-Time interface

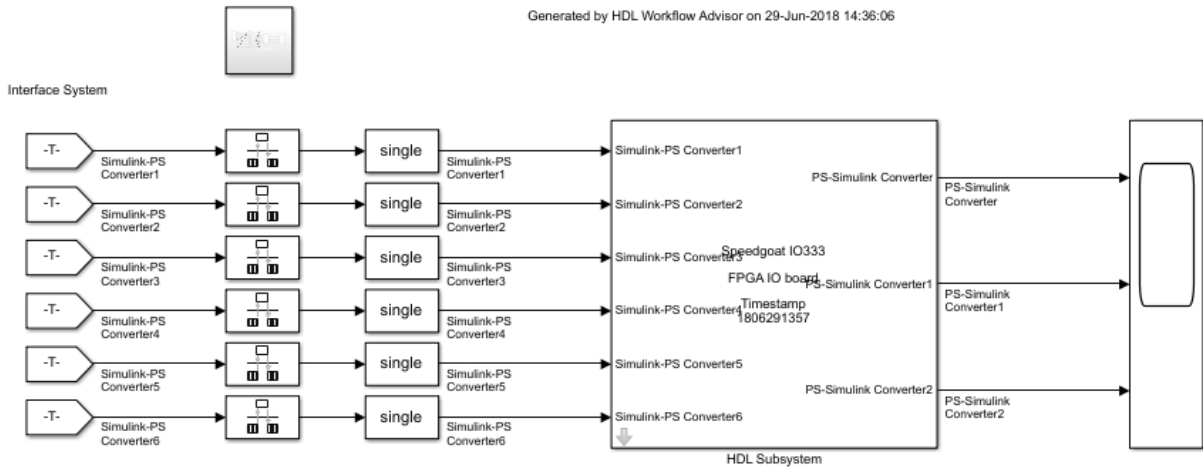
Run This Task

Result: ✔ Passed

Passed Generate Simulink Real-Time Interface.

Generating new Simulink Real-Time Interface model: [gm_gmStateSpaceHDL.TwoLevelConverter HDL. slrt](#)

Simulink Real-Time Interface model generation complete.



Export HDL Workflow to Script

Optionally, you can:

- Save the HDL Workflow Advisor settings to script and run the script using command line.
- Import the settings to modify it and rerun it using the HDL Workflow Advisor User Interface.

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.

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.

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 I0333-325K module, which uses the Xilinx Vivado synthesis tool.

To edit the exported script in MATLAB command window, enter:

```
edit('hdlworkflow_slrt.m')
```

For more information, see Run HDL Workflow with a Script

See Also

Functions

checkhdl | makehdl

More About

- “IP Core Generation Workflow for Speedgoat I/O Modules” on page 41-91
- “FPGA Programming and Configuration” on page 41-52
- “Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model” on page 32-27
- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-55

Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model

This example shows how to modify a Simscape™ plant model to generate an HDL-compatible Simulink® model with HDL Coder™. HDL code is then generated from this Simulink model.

Introduction

The Simscape plant model is converted to an HDL-compatible Simulink model by using the Simscape HDL Workflow Advisor. To run the Advisor, you run the `sshdladvisor` function for the model.

The Simscape HDL Workflow Advisor generates an HDL implementation model from which you generate HDL code. Before you generate the implementation model, make sure that the Simscape plant model is compatible for generation of the implementation model using the Simscape HDL Workflow Advisor. For more information, see [Generate HDL Code from Simscape Models](#).

In some cases, the Simscape plant model may not be compatible for generation of the implementation model using the Simscape HDL Workflow Advisor. For HDL compatibility, you modify the Simscape plant model and then run the Simscape HDL Workflow Advisor. This example illustrates the DC Motor Control plant model. The model contains a nonlinear Friction block. You can use the approach in this example to convert Simscape models with few nonlinear blocks to a HDL-compatible Simulink model.

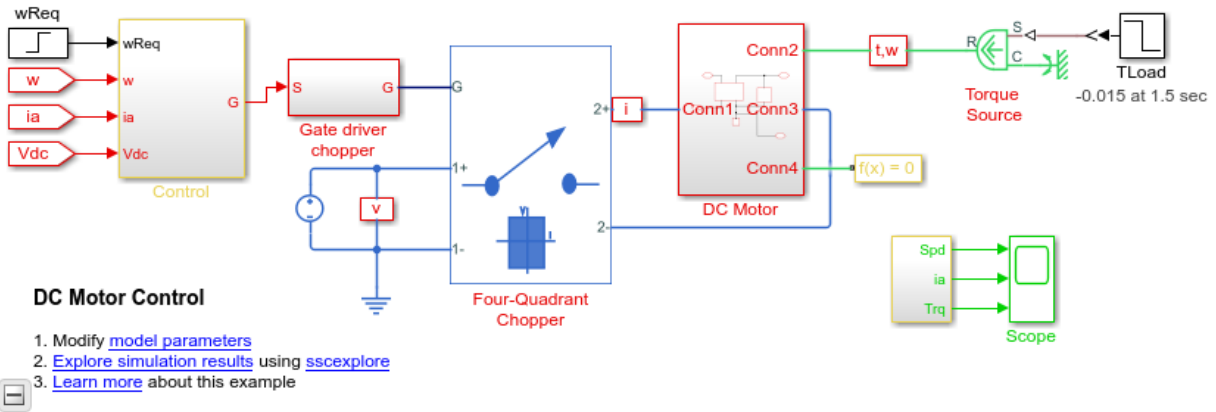
DC Motor Control Model

The DC Motor Control model is a physical model developed in Simscape. The model contains nonlinear elements and requires certain modifications for implementation model generation.

```
open_system('ee_dc_motor_control')
```

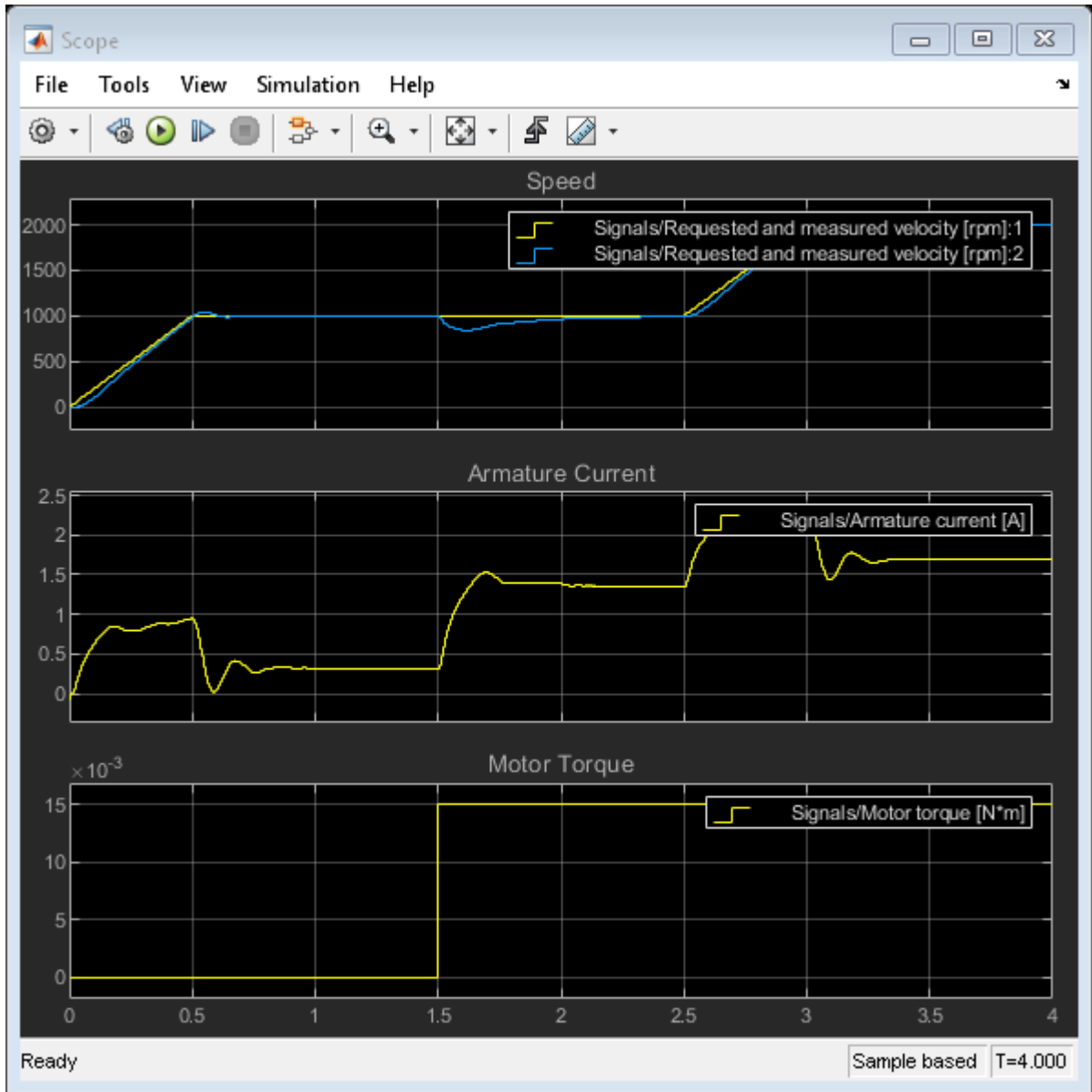
Enclose the DC Motor and Friction block inside a Subsystem and save the model as `ee_dc_motor_control_original`.

```
open_system('ee_dc_motor_control_original')  
set_param('ee_dc_motor_control_original', 'SimulationCommand', 'Update')
```



DC motor control is used as a speed control structure. A PWM controlled four-quadrant Chopper is used to feed the DC motor. The DC motor consists of Rotational Electromechanical Converter, Resistor, Inductance, Friction block and an Inertia block. The control subsystem includes the outer speed-control loop, the inner current-control loop and the PWM generation.

```
sim('ee_dc_motor_control_original')
open_system('ee_dc_motor_control_original/Scope')
```

Make DC Motor Model HDL-Compatible

To convert the model to a model that is compatible for conversion with Simscape HDL Workflow Advisor:

1. Detect presence of nonlinear components or blocks in the model. To verify the presence of nonlinear blocks in Simscape plant model, enter:

```
simscape.findNonlinearBlocks('ee_dc_motor_control_original')
```

```
Found network that contains nonlinear equations in the following blocks:  
'ee_dc_motor_control_original/DC Motor/Friction'
```

```
The number of linear or switched linear networks in the model is 0.  
The number of nonlinear networks in the model is 1.
```

```
ans =
```

```
1x1 cell array
```

```
{'ee_dc_motor_control_original/DC Motor/Friction'}
```

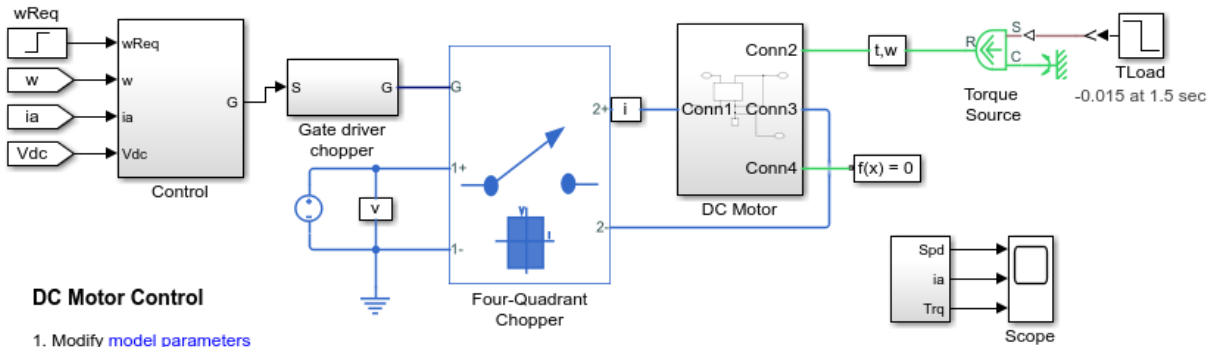
The Simscape plant model has a nonlinear block, which is the Friction block.

2. For HDL compatibility, the model must not contain nonlinear elements. Remove the Friction block from the model.

3. To simulate the model faster and to reduce the time that the Simscape HDL Workflow Advisor takes to extract the state-space equations, reduce the stop time of this model. In the Simulink Toolstrip, on the Simulation tab, change **Stop Time** to 1.

Save the changes into a new model as ee_dc_motor_control_modified.

```
open_system('ee_dc_motor_control_modified')  
set_param('ee_dc_motor_control_original', 'SimulationCommand', 'Update')
```

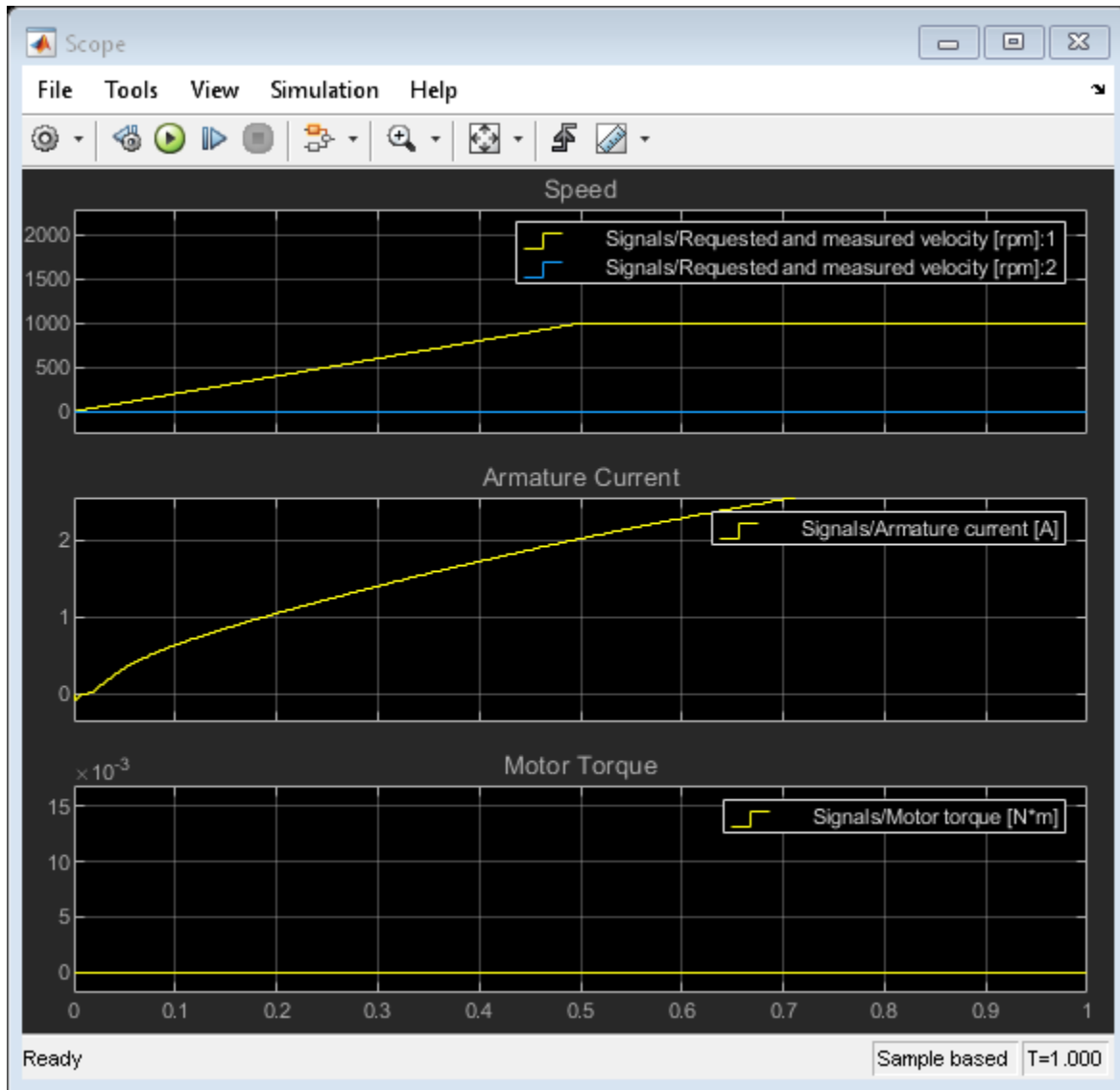


DC Motor Control

1. Modify [model parameters](#)
2. [Explore simulation results](#) using [sscexplore](#)
3. [Learn more](#) about this example

To see the simulation results of the modified model, run these commands:

```
sim('ee_dc_motor_control_modified')
open_system('ee_dc_motor_control_modified/Scope')
```

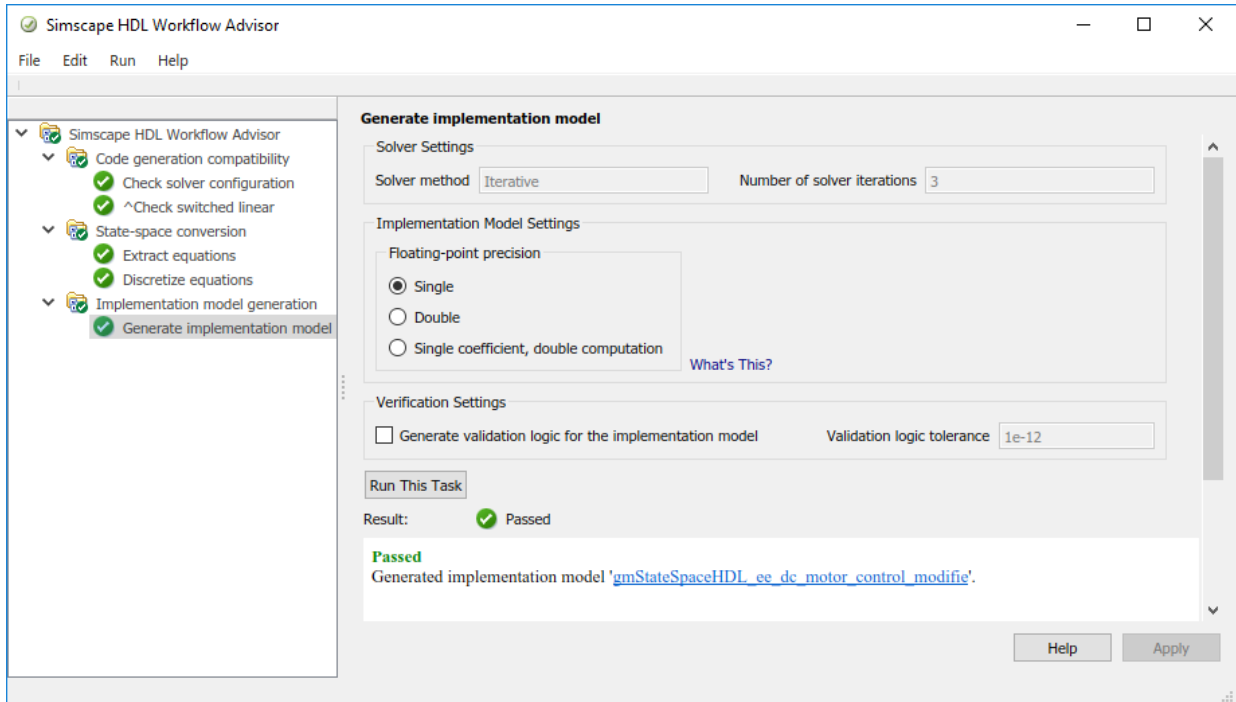


Run Simscape HDL Workflow Advisor and Verify Simulation Results

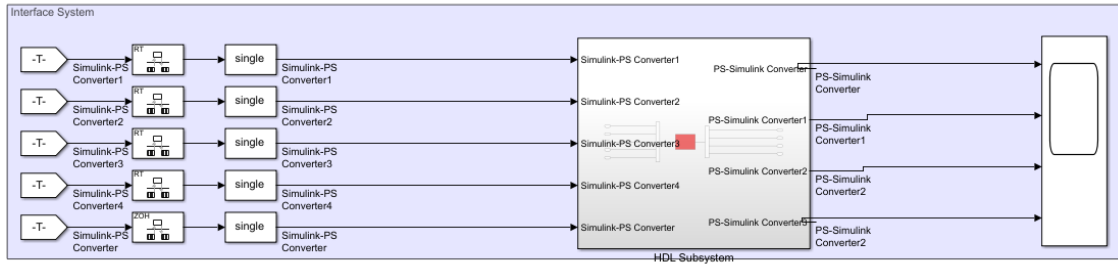
To open the Simscape HDL Workflow Advisor, run the `sschdladvisor` for your model.

```
sschdladvisor('ee_dc_motor_control_modified')
```

To generate the implementation model, in the Simscape HDL Workflow Advisor, leave all tasks to the default settings and then run the tasks. You see a link to the model in the **Generate implementation model** task.



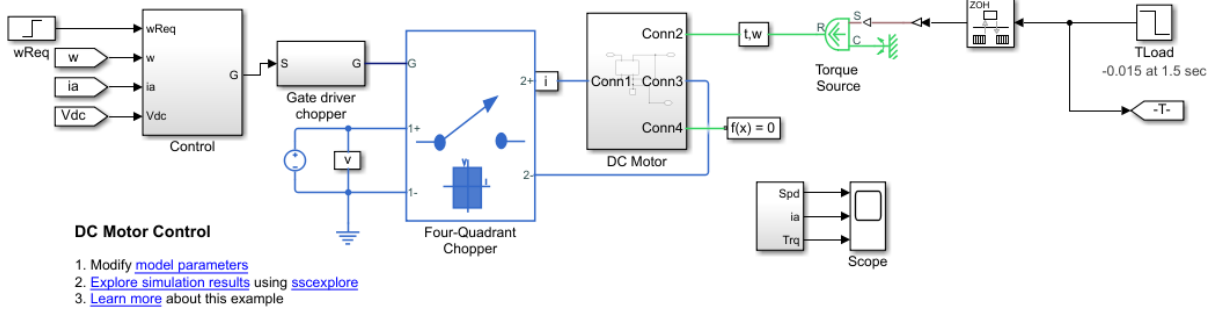
Click the link to open the implementation model.



Simulate Implementation model and Generate HDL code

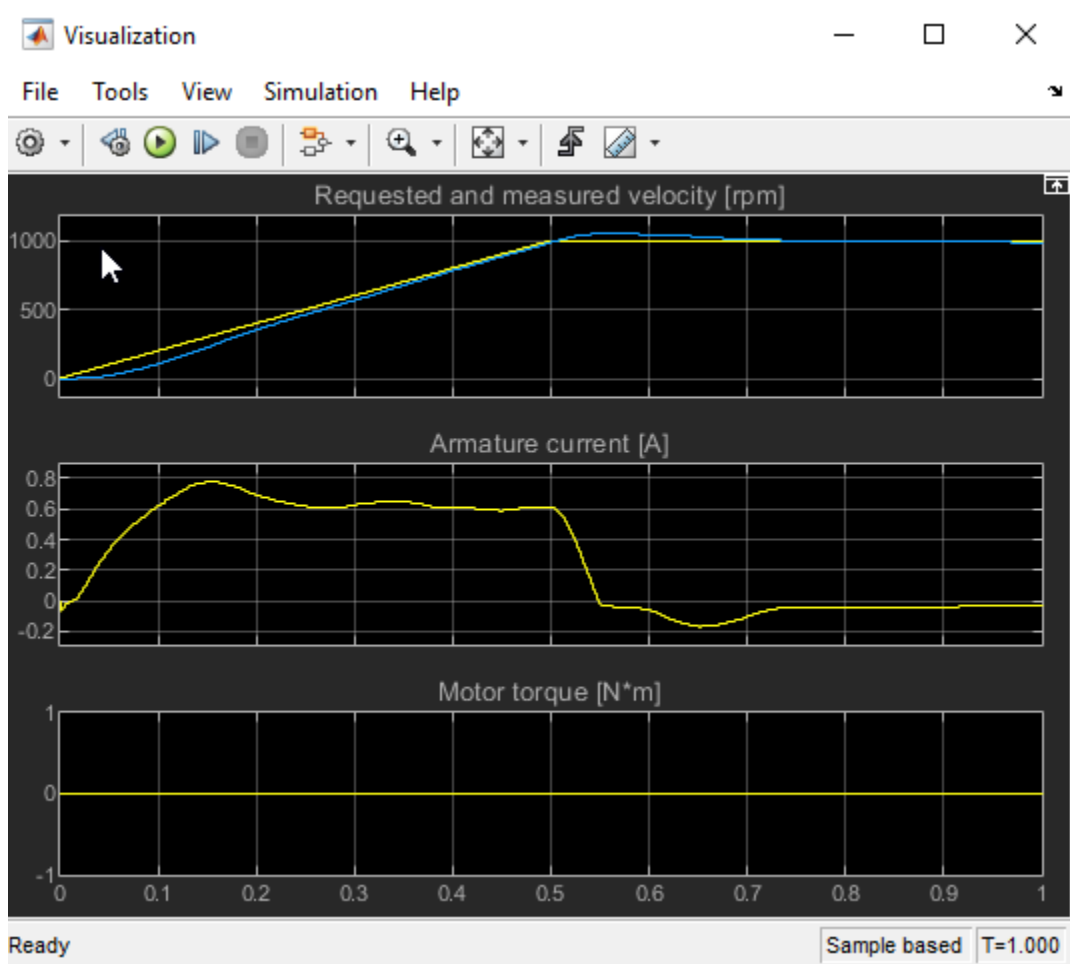
Before you can generate HDL code from the model, you must change the sample time and specify certain settings that make the model compatible for HDL code generation. The sample time of modified plant model is T_s and the number of solver iterations to compute the modes is 3. Therefore, you must change the sample time of the model. To specify the HDL-compatible settings:

- 1 In the Configuration Parameters dialog box:
 - On the **Solver** pane, set **Fixed-step size (fundamental sample time)** to $T_s/3$ and select **Treat each discrete rate as a separate task**.
 - On the **Diagnostics > Sample Time** pane, set **Multitask rate transition** and **Single task rate transition** to error.
- 1 Add a Rate Transition block in your Simscape model that is placed inside the Subsystem block in your implementation model as illustrated in figure below.



To simulate the model, run this command and then open the Scope block to see the results:

```
sim('gmStateSpaceHDL_ee_dc_motor_control_modifie')
```



You see that the output generated by the modified Simscape plant model matches the output generated by the implementation model.

Generate HDL Code and Validation Model

Before you generate HDL code, it is recommended that you enable generation of the validation model. The validation model compares the output of the generated model after code generation and the modified Simscape plant model. To learn more, see [Generated Model and Validation Model](#).

To save validation model generation settings on your Simulink model, run this command:

```
hdlset_param('gmStateSpaceHDL_ee_dc_motor_control_modifie', 'GenerateValidationModel',
```

To generate HDL code, run this command:

```
makehdl('gmStateSpaceHDL_ee_dc_motor_control_modified/HDL Subsystem')
```

By default, HDL Coder generates VHDL code. To generate Verilog code, run this command:

```
makehdl('gmStateSpaceHDL_ee_dc_motor_control_modifie/HDL Subsystem', 'TargetLanguage',
```

The generated HDL code and the validation model is saved in the `hdlsrc` directory. The generated code is saved as `HDL_Subsystem_tc.vhd`. You can also verify the simulation results by running the validation model

```
gm_gmStateSpaceHDL_ee_dc_motor_control_modifie_vnl.slx.
```

See Also

Functions

`checkhdl` | `makehdl`

More About

- “Generate HDL Code from Simscape Models” on page 32-2
- “Getting Started with Simscape Electrical” (Simscape Electrical)
- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-55

Troubleshoot Conversion of Simscape™ Permanent Magnet Synchronous Motor to HDL-Compatible Simulink Model

This example shows how to modify a Simscape™ plant model that is continuous-time and contains nonlinear elements to generate an HDL-compatible Simulink™ model. You can then generate HDL code for this Simulink model.

Introduction

The Simscape HDL Workflow Advisor converts the Simscape plant model to an HDL-compatible implementation model from which you generate HDL code. In some cases, the Simscape plant model might not be compatible for implementation model generation. In such cases, you first modify the Simscape plant model and then run the Advisor.

This example illustrates how to modify a permanent magnet synchronous motor model for compatibility with Simscape HDL Workflow Advisor. The model is continuous time and contains many nonlinear components. You modify this model to a discrete-time switched linear model and then run the Simscape HDL Workflow Advisor.

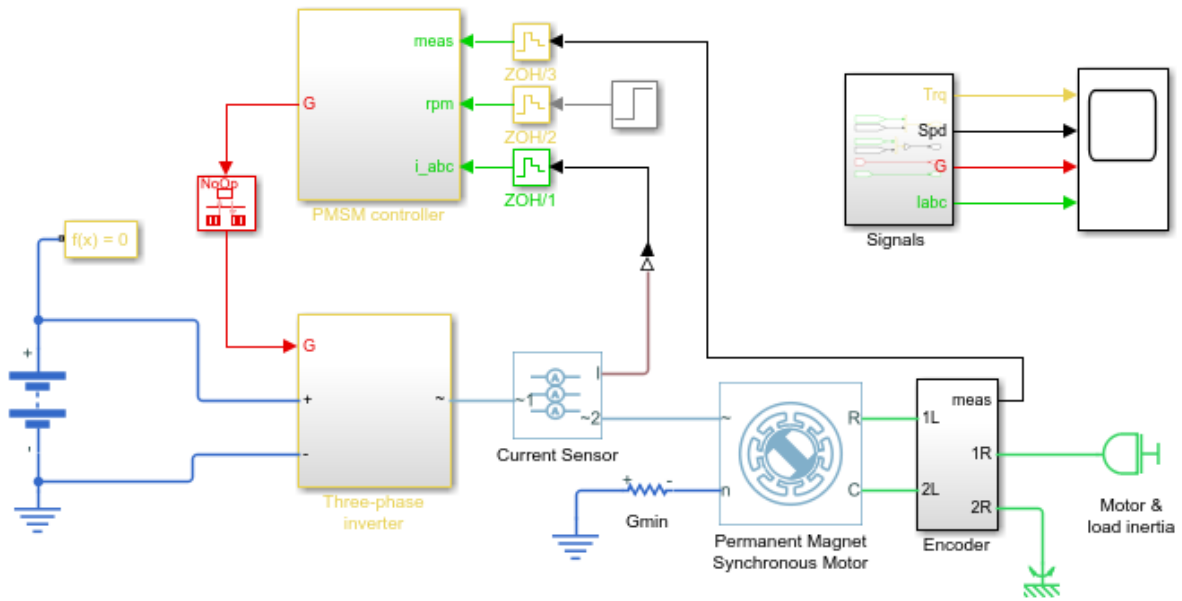
Permanent Magnet Synchronous Motor Model

The permanent magnet synchronous motor model is a physical system in Simscape. To open the model, run this command:

```
open_system('ee_pmsm_drive')
```

Save this model as `ee_pmsm_drive_original.slx`

```
open_system('ee_pmsm_drive_original')  
set_param('ee_pmsm_drive_original','SimulationCommand','Update')
```



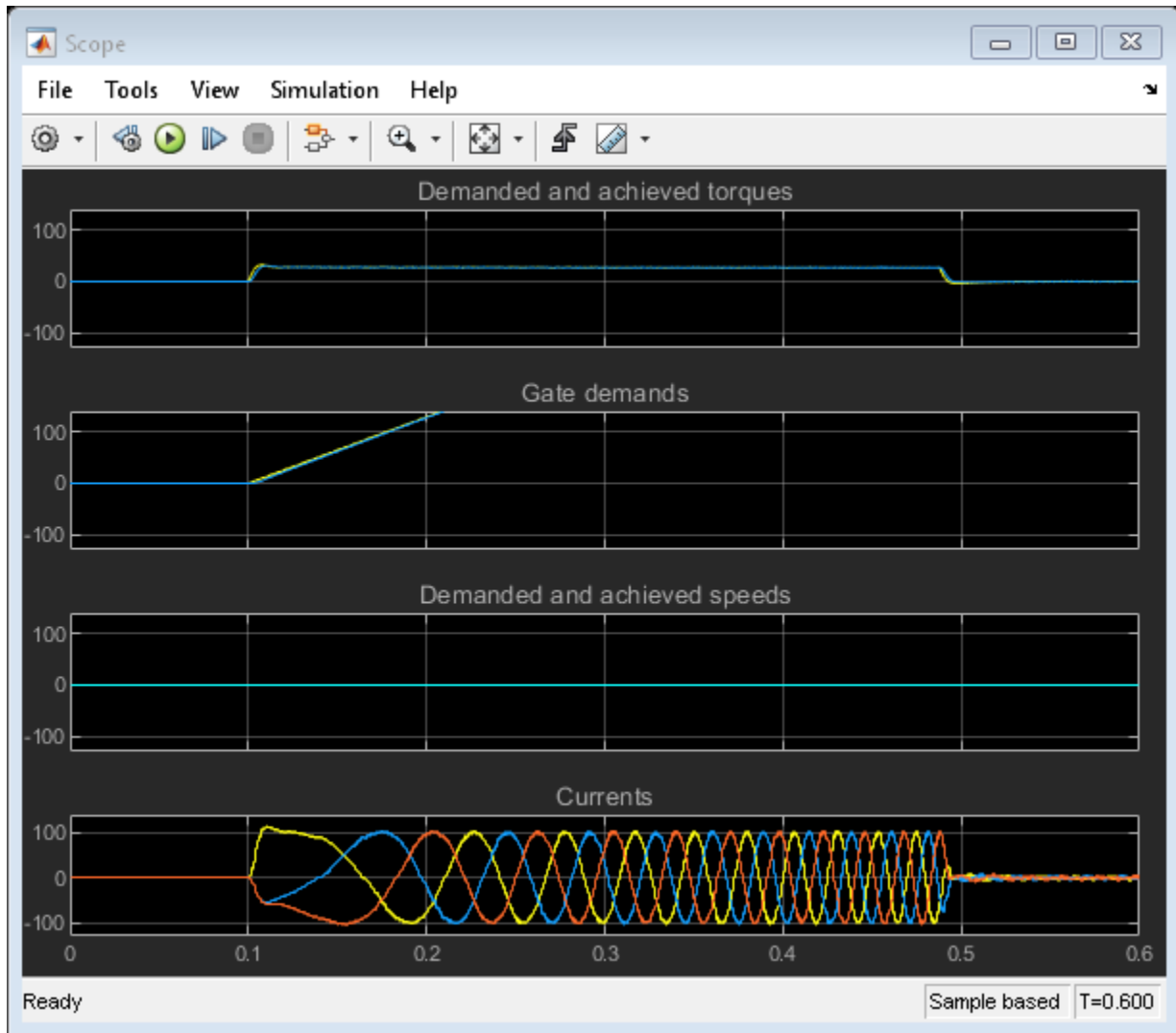
Three-Phase PMSM Drive

1. [Explore simulation results](#) using [sscexplore](#)
2. [Learn more](#) about this example



The model contains a Permanent Magnet Synchronous Machine (PMSM) and inverter-sized that you can use in a typical hybrid vehicle. The inverter is connected to the vehicle battery. To see how the model works, simulate the model.

```
sim('ee_pmsm_drive_original')
open_system('ee_pmsm_drive_original/Scope')
```



This model is a continuous time system. To use this model with Simscape HDL Workflow Advisor, convert the model into a discrete system. Then you modify the model to use blocks that are compatible for the Simscape to HDL workflow.

Convert Continuous-Time Model to Fixed-Step Discrete Model

1. Configure the solver options for HDL code generation by using a Solver Configuration block. In the block parameters:

- Select **Use local solver**.
- Use **Backward Euler** as the **Solver type**.
- Specify a discrete **Sample time**, T_s .

2. Modify the Solver settings of the model. On the **Modeling** tab, click **Model Settings**. On the **Solver** pane:

- Set **Solver selection type** to **Fixed-Step**.
- Set **Solver** to **discrete (no continuous states)**.
- Set **Fixed-step size (fundamental sample time)** to T_s .
- In the section **Tasking and sample time options**, clear **Treat each discrete rate as a separate task**.

3. Modify the display settings of your model. On the **Debug** tab, select **Information Overlays > Sample Time > Colors**. Review the Sample Time Legend for blocks that have a sample time other than T_s , or run at a continuous time scale. Double-click the Step block and set the **Sample time** to T_s .

4. For faster simulation, ignore the zero-sequence parameters of the PMSM. Double-click the Permanent Magnet Synchronous Motor block and set **Zero Sequence** to **Exclude**.

The model is now a fixed-step discrete system. Simulate the model and compare the **Torque Demand** and **Motor Torque** signals in the Simulation Data Inspector. The signals differ by more than the tolerance levels toward the end of simulation but are within acceptable limits.



You use a two-step process to convert the Simscape plant model to a HDL-compatible implementation model:

1. Implement a Simulink model that replaces the nonlinear part of the Simscape algorithm by using equivalent Simulink blocks.
2. Modify this model to use blocks that are compatible for Simscape to HDL workflow.

Replace Nonlinear Simscape Blocks with Equivalent Simulink Implementation

1. To make the Simscape plant model HDL-Compatible, identify the presence of any nonlinear components or blocks in the model:

```
simscape.findNonlinearBlocks('ee_pmsm_drive_original')
```

```
Found network that contains nonlinear equations in the following blocks:  
'ee_pmsm_drive_original/Permanent Magnet Synchronous Motor'
```

```
The number of linear or switched linear networks in the model is 0.  
The number of nonlinear networks in the model is 1.
```

```
ans =
```

```
1x1 cell array
```

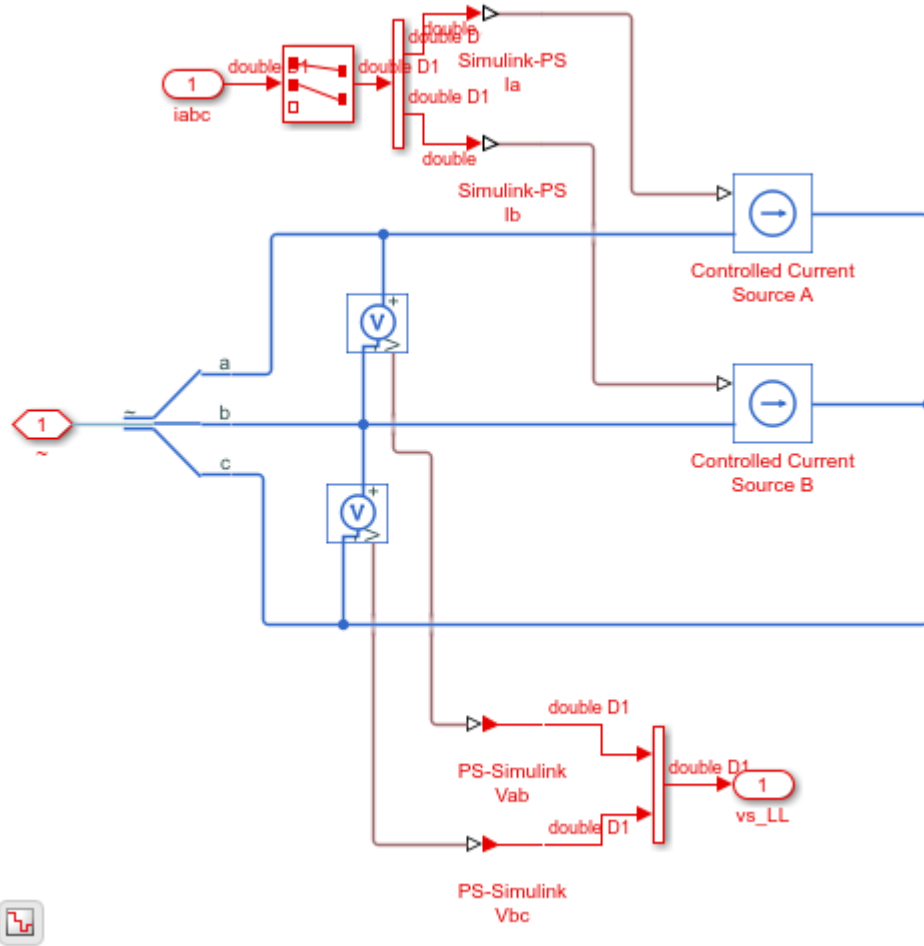
```
{'ee_pmsm_drive_original/Permanent Magnet Synchronous Motor'}
```

The Simscape plant model has a nonlinear block, which is the PMSM block.

2. Replace the PMSM block, Encoder block, Gmin resistor, and Motor & Load Inertia block with Simulink blocks that perform the equivalent algorithm.

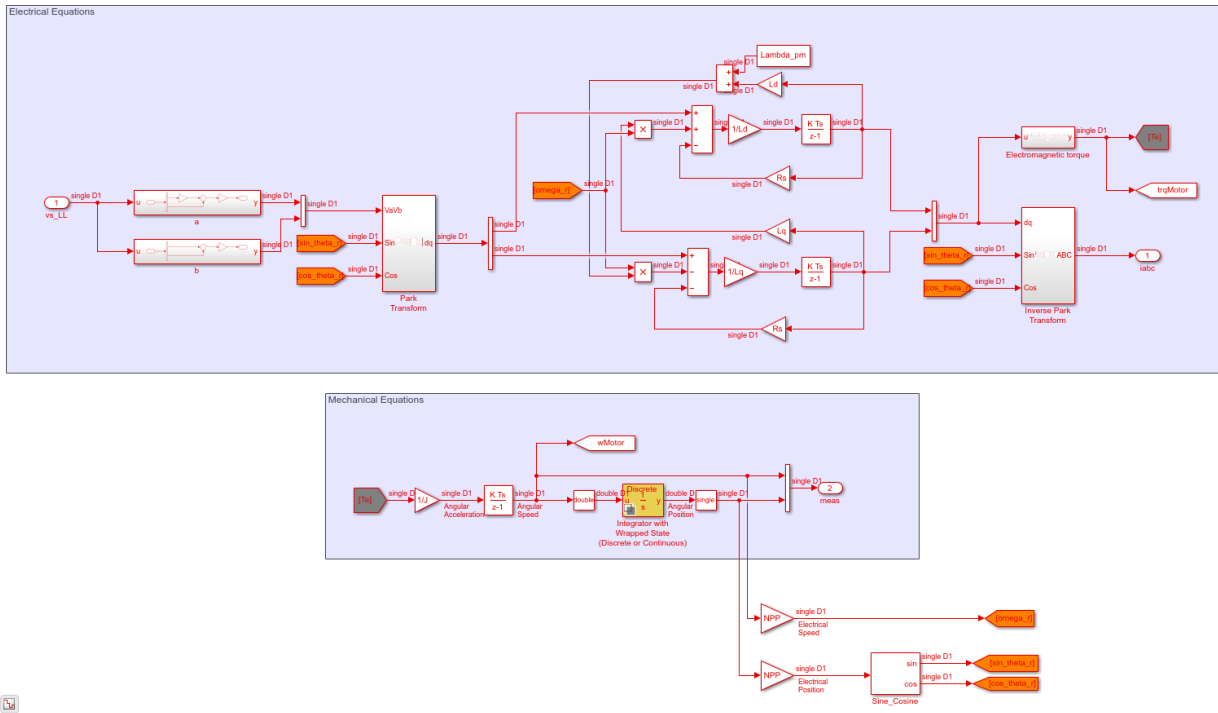
The `ee_pmsm_drive_singleSL` model illustrates how you modify the Electrical Interface and Permanent Magnet Synchronous Motor (Simulink) subsystems. To implement the Electrical Interface block, you use Controlled Current Sources.

```
load_system('ee_pmsm_drive_singleSL')  
set_param('ee_pmsm_drive_singleSL', 'SimulationCommand', 'update')  
open_system('ee_pmsm_drive_singleSL/Electrical Interface')
```



The interface to the PMSM is isolated from the implementation. To implement the PMSM by using Simulink blocks, you use Electrical Equations and Mechanical Equations. Inside the Park Transform and Inverse Park Transform blocks, eliminate the Sine and Cosine blocks.

```
open_system('ee_pmsm_drive_singleSL/Permanent Magnet Synchronous Motor (Simulink)', 'f
```

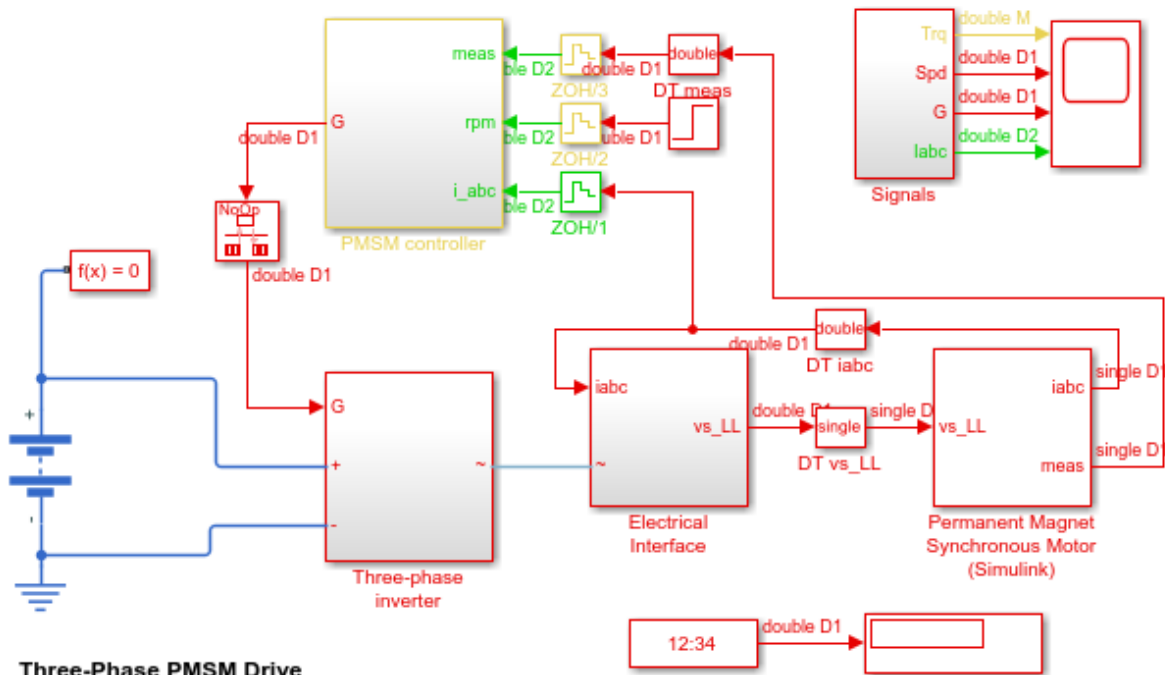



Modify Simulink Model for Compatibility with HDL Implementation Model Generation

1. To save resource usage on the target hardware, add Data Type Conversion blocks to convert the model to use `single` data types.
2. Add a Digital Clock with **Sample time** `Ts` at the top level of the model. Connect the clock to a Display block.
3. Replace the Three-Phase Current Sensor Simscape block by feeding the controller with three-phase currents coming from the PMSM model.

This figure illustrates the top level of the model with the above changes.

```
open_system('ee_pmsm_drive_singleSL')
```



Three-Phase PMSM Drive

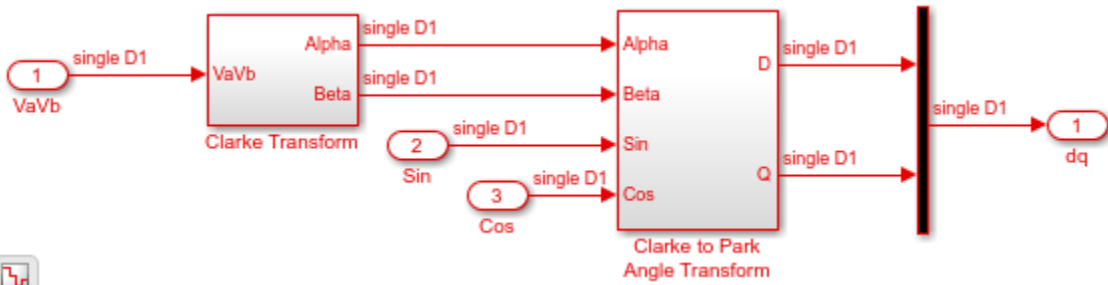
1. [Explore simulation results](#) using [sscexplore](#)
2. [Learn more](#) about this example



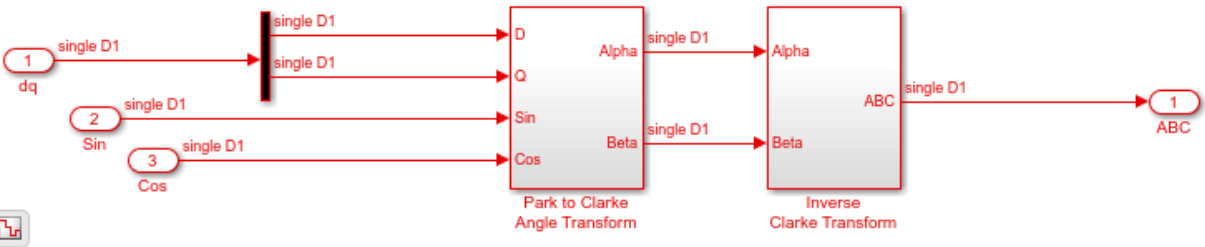
4. To reduce the FPGA area footprint for the:

- Park Transform block, you add Clarke Transform and Clarke to Park Angle Transform blocks.
- Inverse Park Transform block, you add Inverse Park to Clarke Angle Transform and Inverse Clarke Transform blocks.

```
open_system('ee_pmsm_drive_singleSL/Permanent Magnet Synchronous Motor (Simulink)/Park
```



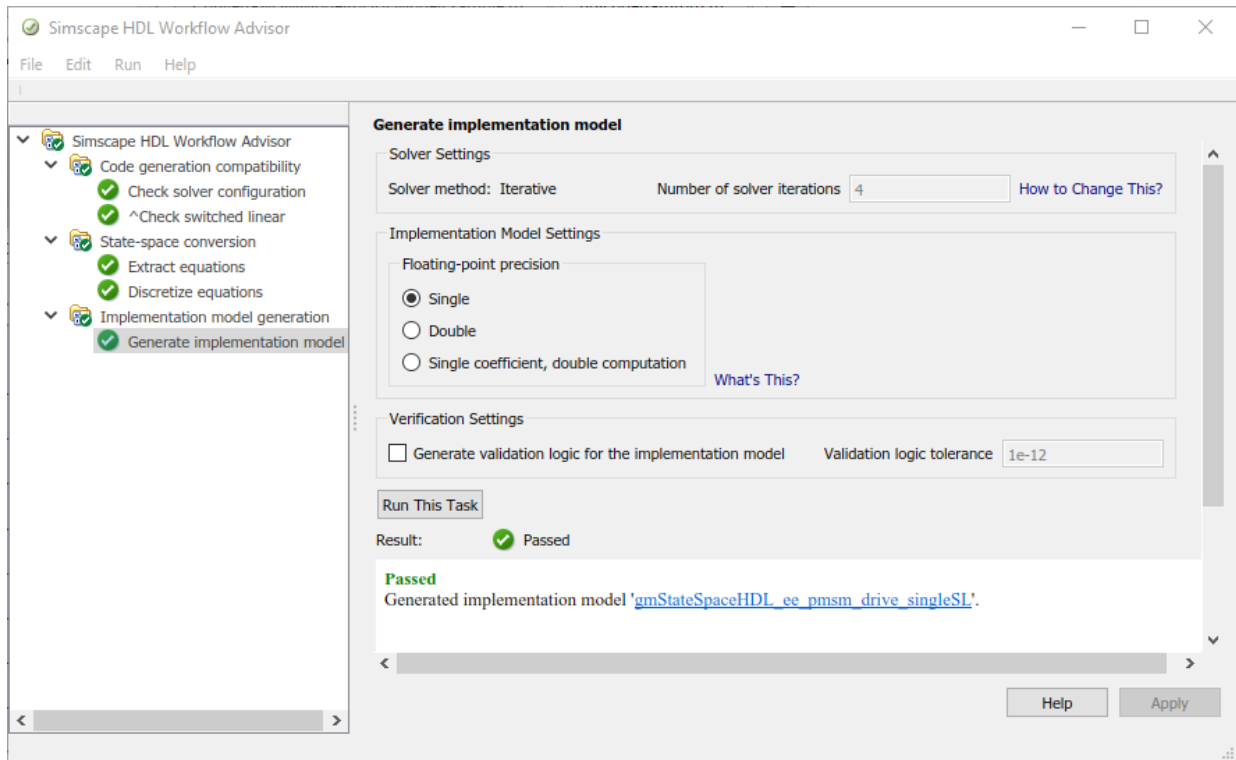
```
open_system('ee_pmsm_drive_singleSL/Permanent Magnet Synchronous Motor (Simulink)/Inverse')
```



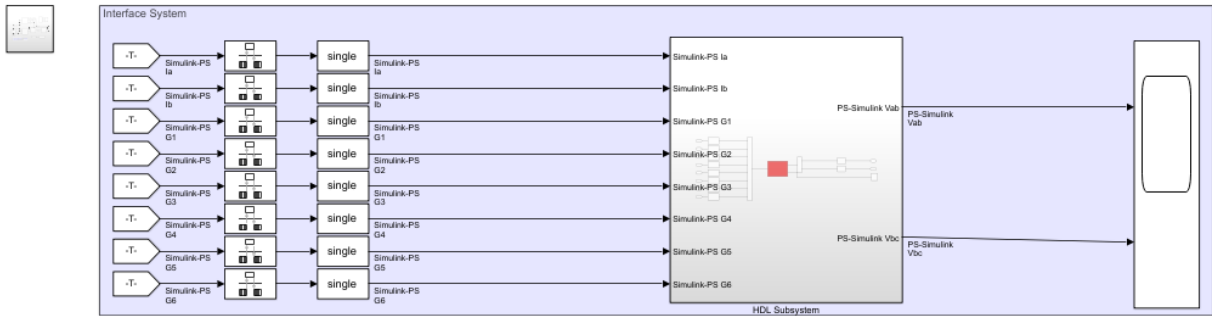
Run Simscape HDL Workflow Advisor

To open the Simscape HDL Workflow Advisor, run the `sschdladvisor` function for your model:

```
sschdladvisor('ee_pmsm_drive_singleSL')
```



To generate the implementation model, in the Simscape HDL Workflow Advisor, leave the default settings and then run the tasks. To open the implementation model, in the **Generate implementation model** task, click the link.



Reconfigure Implementation Model for HDL Code Generation

Save this model as gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL. To reconfigure the single-precision implementation model for HDL code generation:

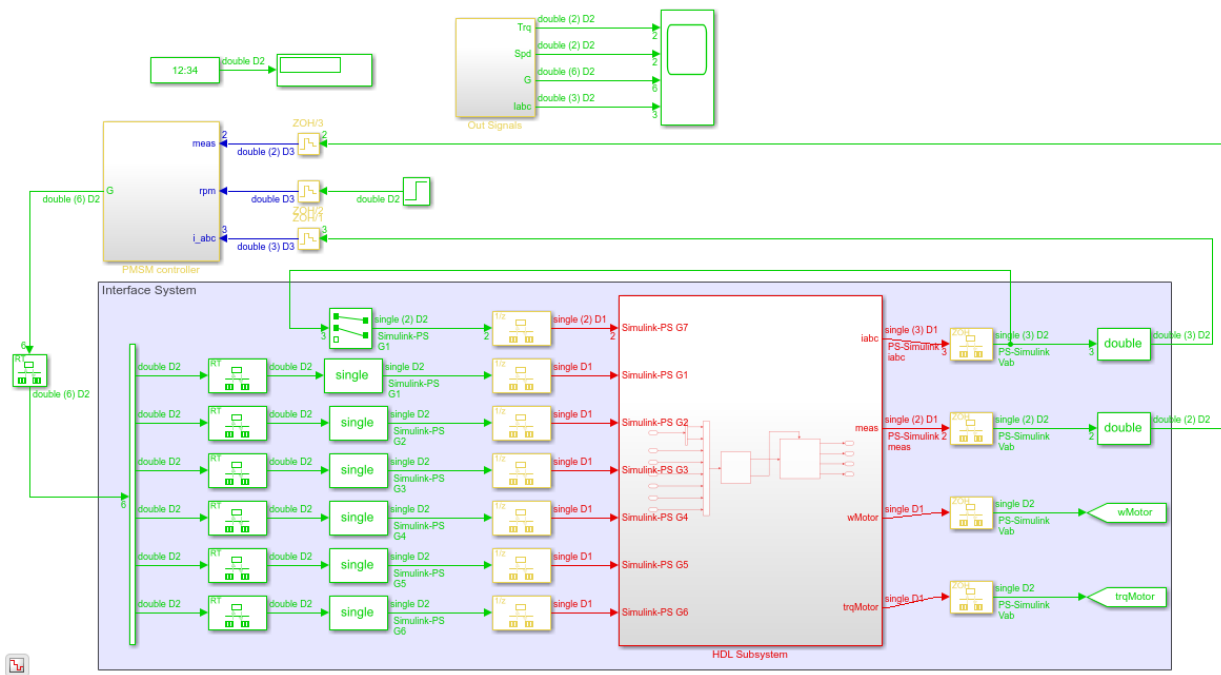
1. Run the hdlsetup function on the model.

```
hdlsetup('gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL')
```

2. Modify the **Fixed-step size** to $T_s/4$ because the default **Number of Iterations** is 4.

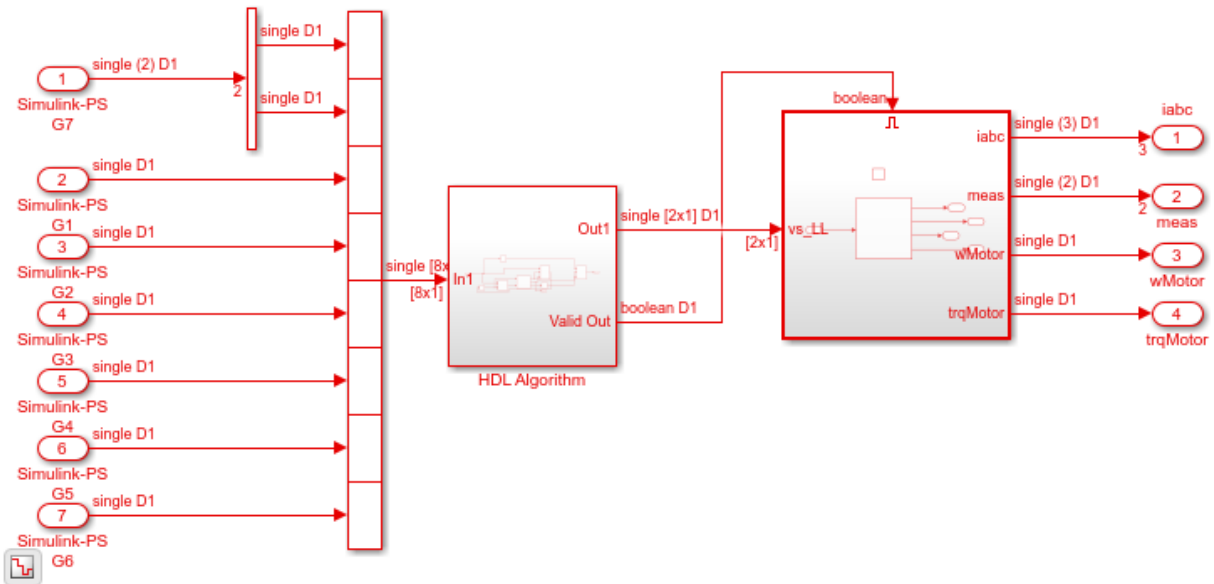
3. Remove the PMSM Controller and Permanent Magnet Synchronous Motor (Simulink) subsystems from the Subsystem that contains the original Simscape model. Remove this Subsystem and place the PMSM Controller subsystem with the HDL Subsystem at the top level of the model.

```
model_name = 'gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL';
dut_name = 'gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL/HDL Subsystem';
open_system(model_name)
set_param(model_name, 'SimulationCommand', 'Update')
```



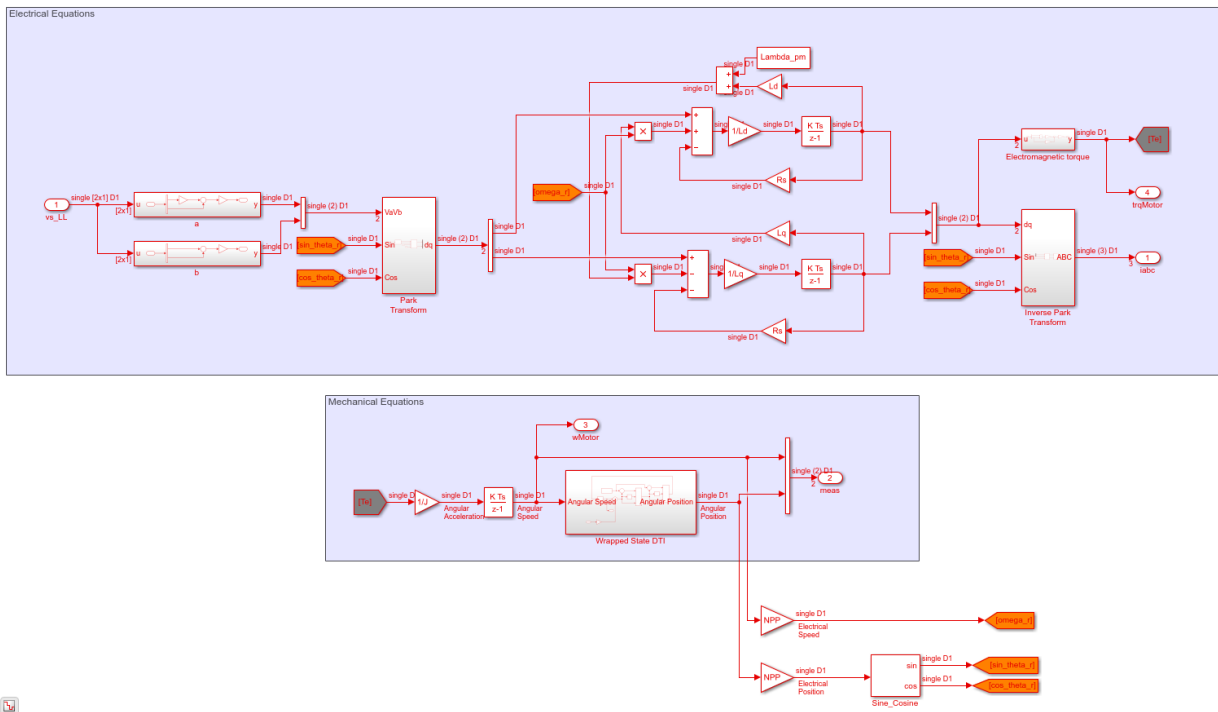
4. Inside the HDL Subsystem, the HDL Algorithm Subsystem has a Valid Out signal. Place the Permanent Magnet Synchronous Motor (Simulink) subsystem inside an Enabled Subsystem and connect the vs_LL input port to the Valid Out signal.

```
open_system(dut_name)
```



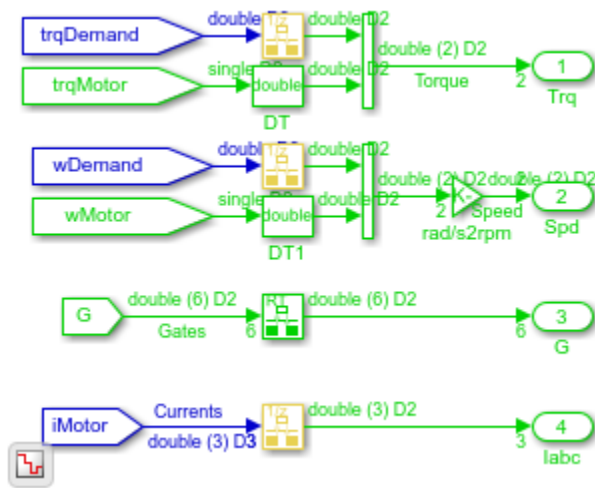
5. Inside the Permanent Magnet Synchronous Motor (Simulink) subsystem, several Discrete-Time Integrator blocks have the sample time as T_s . Change the sample time of these blocks to -1. The Integrator with Wrapped State (Discrete or Continuous) block is not compatible for HDL code generation. Replace the Integrator with Wrapped State (Discrete or Continuous) with this Wrapped State DTI Subsystem.

`open_system([dut_name, '/Enabled Subsystem/Permanent Magnet Synchronous Motor (Simulink)`



6. Simulate the model. If the simulation displays errors that indicate a rate mismatch, add Rate Transition blocks between the ports that resolve this mismatch. Inside the Out Signals Subsystem, to provide the output signals to the Scope block at the same rate, place Rate Transition blocks.

```
open_system([model_name, '/Out Signals'])
```

Generate HDL Code

Before you generate HDL code, to compare the output of the generated model after code generation with the modified Simscape plant model, specify validation model generation.

```
hdlset_param(model_name, 'GenerateValidationModel', 'on');
```

To learn more, see [Generated Model and Validation Model](#).

To generate HDL code, run this command:

```
makehdl('gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL/HDL Subsystem')
```

By default, HDL Coder generates VHDL code. To generate Verilog code, run this command:

```
makehdl('gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL/HDL Subsystem', 'TargetLanguage', 'V')
```

The code generator saves the generated HDL code and the validation model in the `hdlsrc` folder. The generated code is saved as `HDL_Subsystem_tc.vhd`. To see the resource usage information of your design, view the [Code Generation Report](#).

To open the validation model, after you generate HDL code, open the `gm_gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL_vnl.slx` model.

See Also

Functions

`checkhdl` | `makehdl`

More About

- “Generate HDL Code from Simscape Models” on page 32-2
- “Deploy Simscape™ Plant Models to Speedgoat FPGA I/O Modules” on page 32-14
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-55
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-7
- “Getting Started with Simscape Electrical” (Simscape Electrical)

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.

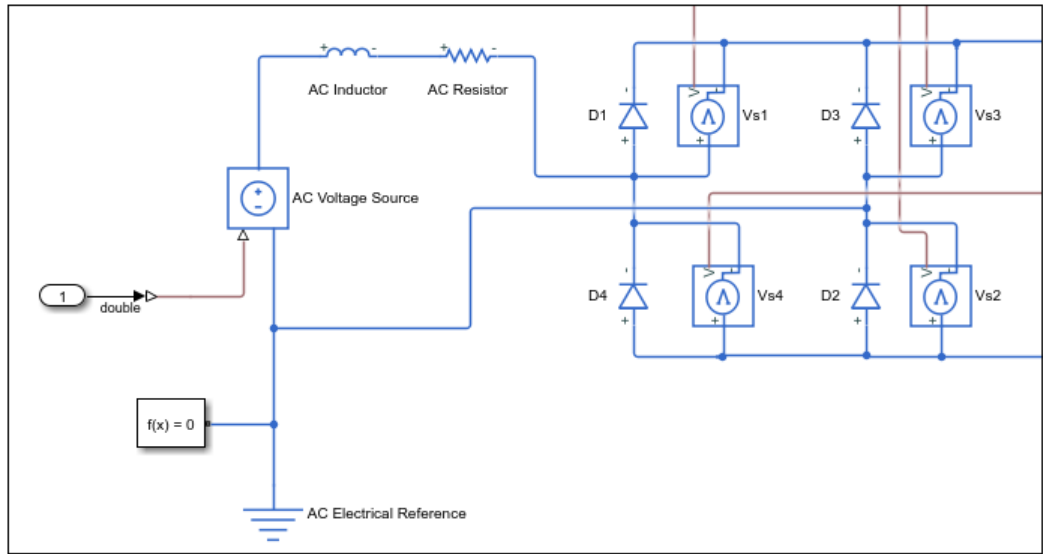
Bridge Rectifier Model

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:

```
open_system('sschdlexBridgeRectifierExample')  
open_system('sschdlexBridgeRectifierExample/Simscape_system')
```

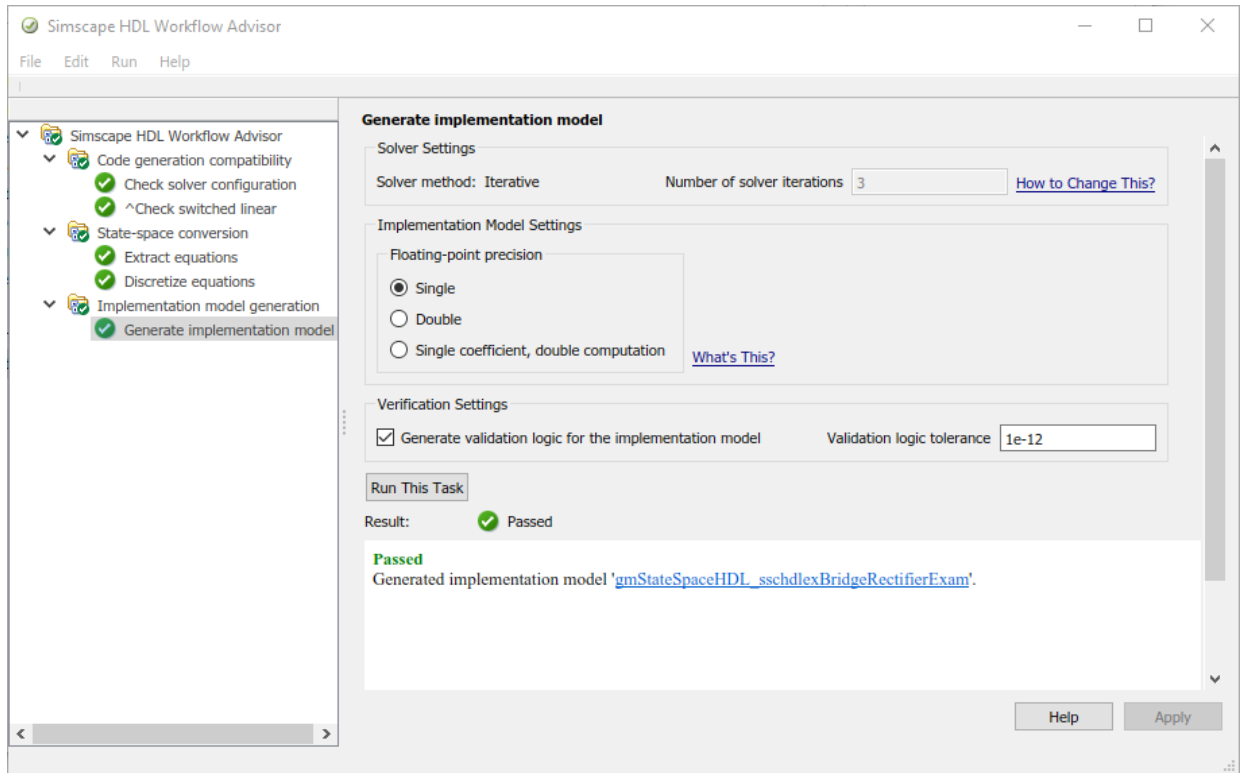
Inside the `Simscape_system`, you see four diodes arranged in a bridge configuration. For both positive and negative input values, this configuration provides a positive, rectified output.



- 2 Open the Simscape HDL Workflow Advisor for your model:

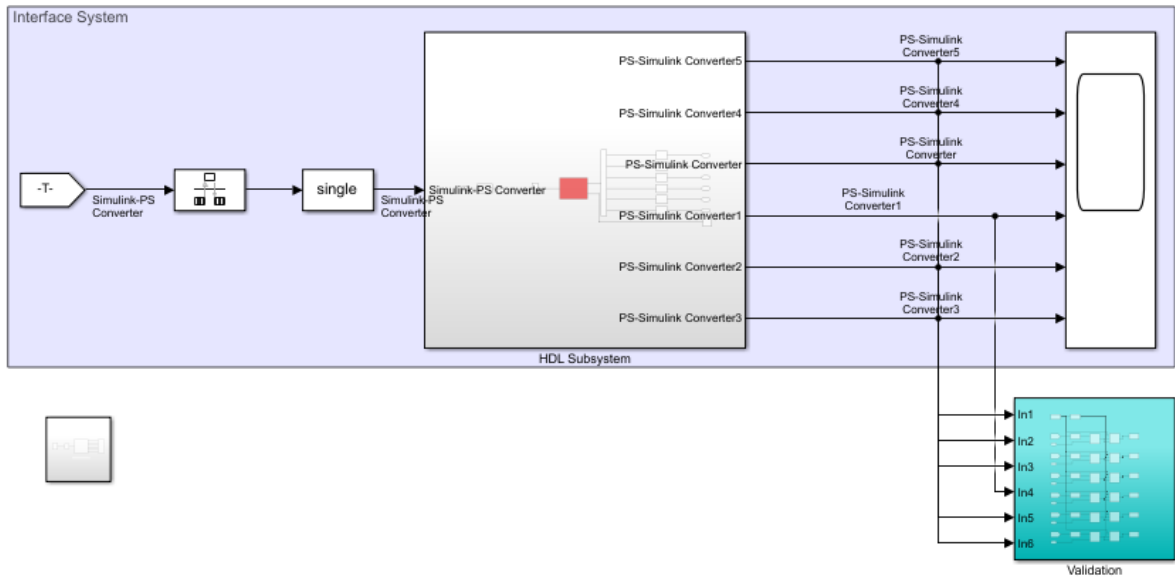
```
sschdladvisor('sschdlexBridgeRectifierExample')
```

- 3 Right-click the **Get state-space parameters** task and select **Run to Selected Task** to run all tasks in the Advisor except for the **Generate implementation model** 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

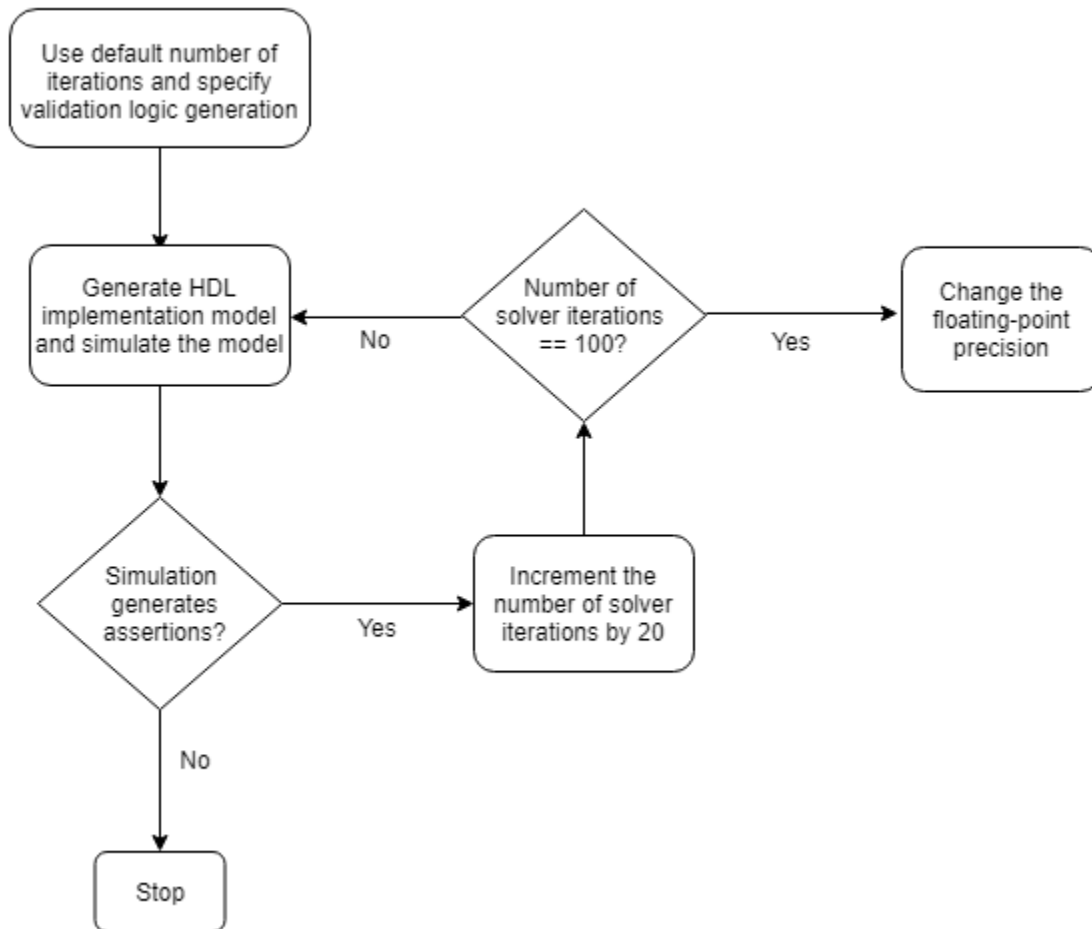
Conversion of a Simscape algorithm to an equivalent HDL implementation leads to rounding errors. By default, the **Validation logic tolerance** is set to $1e-12$. This tolerance value is relatively small and can be difficult to achieve especially with single-precision data types in the HDL implementation 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.

To improve the numeric accuracy of the generated HDL implementation model and resolve the mismatch, increase the number of solver iterations. This flowchart illustrates how to change the **Number of solver iterations**.



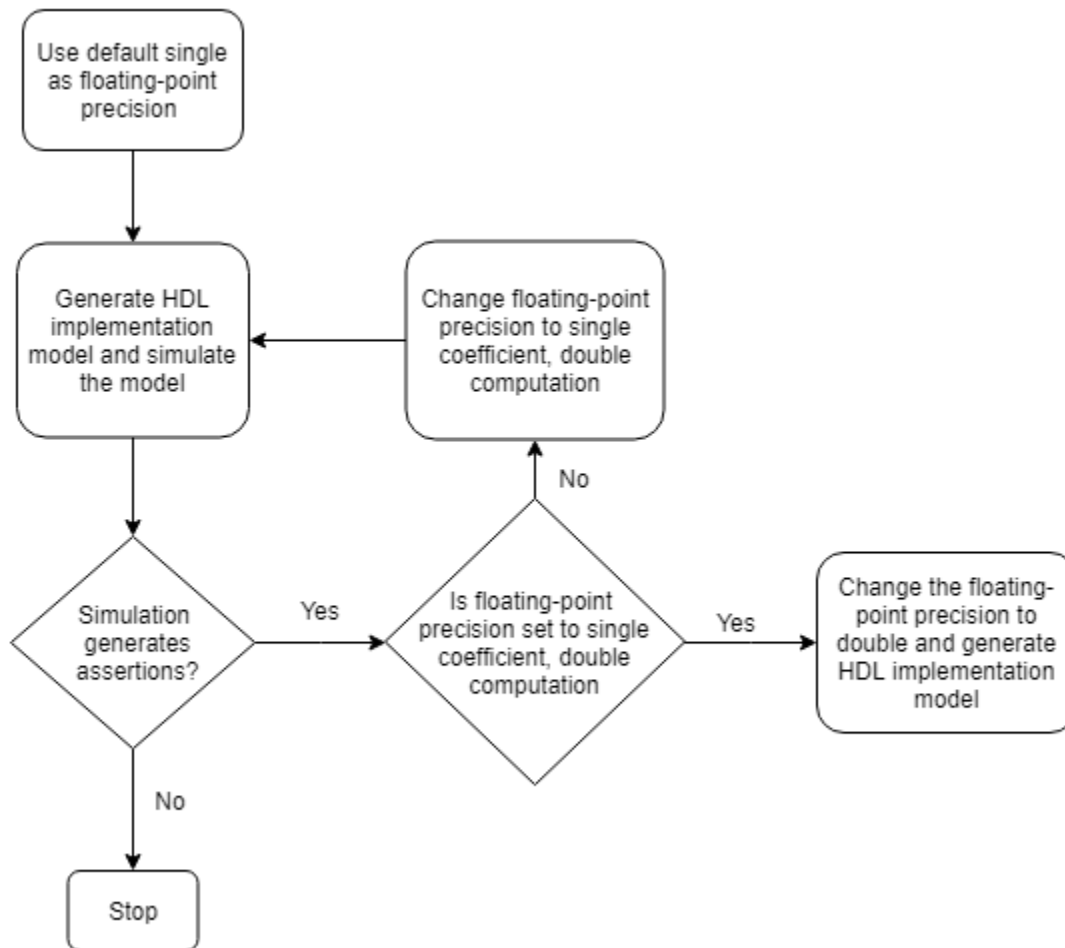
Note When you increase the number of solver iterations, the code generator changes the sample time of the generated HDL implementation model. A large number of iterations can increase the simulation time significantly.

Use Larger Floating-Point Precision

You can use the **Floating-point 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.

Floating-Point Precision	Description
Double	Using double floating-point precision increases the numerical accuracy of the generated model. Using double data types in your model can increase the area consumption and reduce the target frequency.
Single	This is the default setting for floating-point 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 **Floating Point 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 **Floating Point 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

`simscape.findNonlinearBlocks` | `sschdladvisor`

More About

- “Generate HDL Code from Simscape Models” on page 32-2
- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-7
- “Deploy Simscape™ Plant Models to Speedgoat FPGA I/O Modules” on page 32-14

Improve Sampling Rate of HDL Implementation Model Generated from 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. 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 33-2.

Sampling Frequency

When you generate HDL code and deploy the plant model onto an FPGA, you may want to improve the sampling frequency. The sampling frequency depends on these parameters:

- FPGA clock frequency
- Oversampling factor
- Number of solver iterations

$$\text{Sampling Frequency} = \text{FPGA clock frequency} / (\text{Oversampling factor} * \text{Number of solver iterations})$$

To improve the sampling rate, you want to maximize the FPGA clock frequency, and minimize the oversampling factor and number of solver iterations. As you improve the sampling rate, 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. To learn more about how this block is used in your model before running the Simscape HDL Workflow Advisor, see “Generate HDL Code from Simscape Models” on page 32-2.

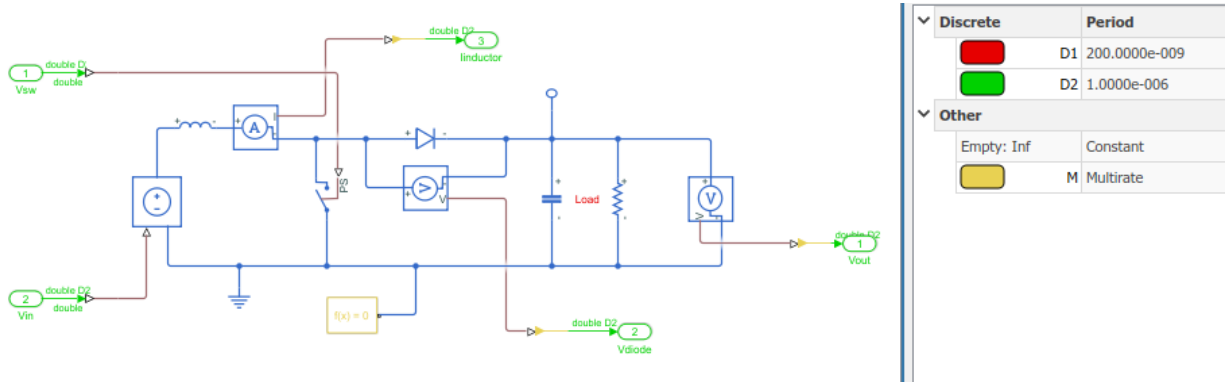
The preceding section uses the boost converter model as an example to illustrate how you can modify the oversampling factor and the number of solver iterations to improve the sampling rate.

Boost Converter Model

This example uses the boost converter model to illustrate the change in 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:

```
open_system('sschdlexBoostConverterExample')
open_system('sschdlexBoostConverterExample/Simscape_system')
```



You see that the model runs at a sample time $1e-6$. The sample time of $200e-9$ corresponds to the sample time of the sources that drive the Simscape algorithm.

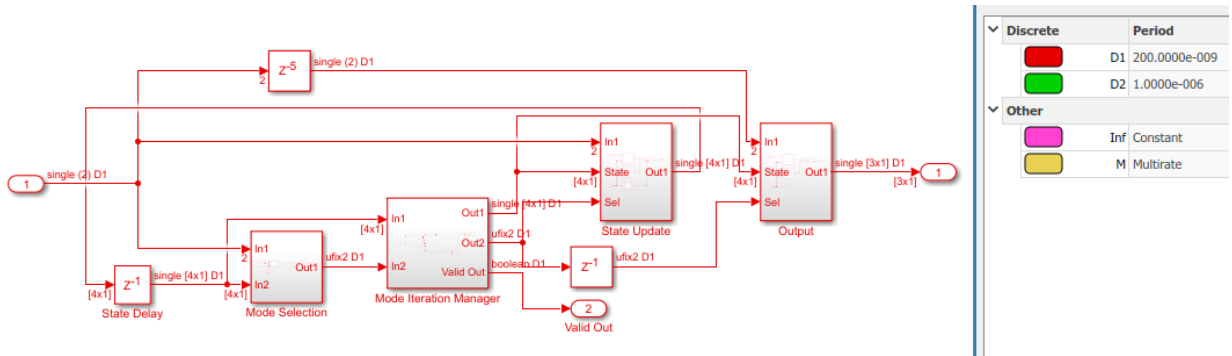
- 2 Open the Simscape HDL Workflow Advisor for your model:

```
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 $200e-9$, 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_BoostConverter_HDL' HDL parameters
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL', ...
'FloatingPointTargetConfiguration', hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint' ...
, 'LatencyStrategy', 'MIN') ...
);
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL', 'HDLSubsystem', 'gmStateSpaceHDL_BoostConverter_HDL');
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL', 'MaskParameterAsGeneric', 'on');
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL', 'Oversampling', 60);

% Set SubSystem HDL parameters
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL/HDL Subsystem', 'FlattenHierarchy', 'on');

% Set SubSystem HDL parameters
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL/HDL Subsystem/HDL Algorithm/State Update/Multiply State', ...
'SharingFactor', 1);
```

The HDL parameters that are saved indicate that the model has the native floating-point mode enabled and uses an **Oversampling factor** of 60 and has **Latency Strategy** set to MIN. This default combination of HDL parameters offers an optimal trade-off between oversampling factor and the target FPGA clock frequency and improves the sampling frequency. If you want to further improve the sampling frequency, you can reduce the number of iterations and the oversampling factor as illustrated below.

Reducing 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.

To improve the sampling rate, you want to 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 must calculate, and so on.

In the **Generate implementation model** task of the Simscape HDL Workflow Advisor:

- 1 Start by reducing the **Number of solver iterations** to a value such as 3
- 2 Select **Generate validation logic for the implementation model**, and then generate the HDL implementation model.
- 3 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. To learn how to resolve the mismatch, see “Validate HDL Implementation Model to Simscape Algorithm” on page 32-55.

Note To resolve the mismatch, it is recommended that you do not change the **Floating-point precision** to double. 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.

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 “Strategy 1: Global Oversampling” on page 10-50.

You vary the oversampling factor and latency strategy of the floating-point operator in conjunction. The default oversampling factor of 60 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. 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 60, and **Latency Strategy** set to Min provides the optimal sampling frequency.

See Also

Functions

`simscape.findNonlinearBlocks` | `sschdladvisor`

More About

- “Solvers for Real-Time Simulation” (Simscape)
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-7
- “Latency Considerations with Native Floating Point” on page 10-83
- “Deploy Simscape™ Plant Models to Speedgoat FPGA I/O Modules” on page 32-14

Simscape HDL Workflow Advisor Tasks

- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-7

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:

```
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

In the Simscape HDL Workflow Advisor, for summary information on each Simscape HDL Workflow Advisor folder or task, right-click that folder or task and select **What's This?**.

Simscape HDL Workflow Advisor folder

The Simscape HDL Workflow Advisor consists of various tasks that you can use to convert your Simscape model to an HDL implementation model. You can generate code for the HDL Subsystem in this model. The Simscape HDL Workflow Advisor consists of folders that perform these tasks:

- The **Code generation compatibility** folder consists of tasks that check whether the model is a switched linear system and uses the correct solver configuration settings.
- The **State-space conversion** folder consists of tasks that derive the state-space parameters from your model for generating the implementation model.
- The **Implementation model generation** folder consists of a task that generates the HDL implementation model from the state-space parameters.

To learn more about each folder or task, right-click that folder or task, and select **What's This?**.

Code generation compatibility folder

The tasks in the **Code generation compatibility** folder check whether:

- You have correctly specified the Solver Configuration settings.
- Your model uses switched linear blocks.

Check solver configuration task

The **Check solver configuration** task checks whether you have specified the correct settings for the Solver Configuration block in your Simscape model.

The Advisor checks whether you specified:

- **Use local solver**
- Backward Euler as **Solver type**.
- A discrete sample time, T_s .

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 switched linear task

The **Check switched linear** task checks whether you use switched linear blocks in your Simscape model.

For this task to pass, the model that you use must contain linear or switched linear blocks. Nonlinear blocks and time-varying blocks (such as Variable Inductor and Variable Capacitor blocks) are not supported.

Linear blocks are blocks that are defined by a linear relationship. For example, a resistor is a linear block since it is defined by the equation $V = IR$. Similarly, an inductor is linear because it is defined by $V = d/dt I L$. Switched linear blocks are blocks such as diodes or switches. These blocks are 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.

This task runs `simscape.findNonlinearBlocks` on your model to check for the presence of nonlinear blocks. For example:

```
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

If your model contains nonlinear blocks, running this task fails. In the **Result** log, you see links provided to the nonlinear blocks in your model. To continue the workflow, replace the nonlinear blocks with switched linear blocks, and rerun the task.

When this task passes, it displays:

- A message indicating that the model is switched linear.

- The number of algebraic and differential variables with links to the blocks in your Simscape model that are related to these variables.

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 folder

Before you can generate the HDL implementation model, run the task in this folder to derive the state-space parameters from your model. The tasks in this model:

- Simulate the Simscape model to extract the differential algebraic equations.
- Discretize the differential algebraic equations to generate an abstract state space representation that represents the model in the form of linear modes.

Extract Equations

The **Extract Equations** task simulates your Simscape model to extract the differential algebraic equations. This task derives the **Simulation stop time** value from the original Simscape model.

If this task passes, it displays the number of states, inputs, outputs, modes, and differential variables. The task also displays a message if the model is purely linear and does not contain any nonlinear elements.

The number of modes is limited by the number of switches present in your Simscape model. The maximum number of modes possible are $2^{(\text{number of switches})}$. All the modes that the Advisor generates are executed as per the input parameters by using a switching logic. A valid number of modes are selected depending on the design of your Simscape model.

Discretize Equations

This task discretizes the differential algebraic equations and generates an abstract discrete state-space representation. This task represents the model in the form of linear modes. Each mode is represented by a set of state-space matrices. This task derives the **Discrete sample time** value from the original Simscape model.

If this task passes, it displays the **Discrete sample time** and number of parameters, modes, and so on.

Passed

Summary of the state-space representation:

- Discrete sample time: 1e-05

Parameter	Parameter size
A	7 x 7 x 3
B	7 x 1 x 3
F0	7 x 1 x 3
C	6 x 7 x 3
D	6 x 1 x 3
Y0	6 x 1 x 3

Implementation model generation folder

The task in the **Implementation model generation** folder generates 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. If the task **Generate implementation model** in this folder passes, it provides a link to the implementation model.

Generate implementation model task

To generate an HDL implementation model from the discrete state-space representation, run this task. The HDL implementation model contains a HDL `Subsystem` that models

the state-space equations by using the state-space parameters derived by running the **Get state-space parameters** task. The HDL Subsystem block represents the DUT for which you can generate HDL code.

Before you run this task, you can:

- Specify a custom value for the **Number of Solver iterations** setting. To learn more, see “Using Number of Solver Iterations” on page 33-11.
- Use the **Floating-point precision** setting to specify whether the HDL Subsystem in the generated implementation model stores matrix types in `single` or `double` and computes the results in `single` or `double` data types. To learn more, see “Floating-Point Precision and Numerical Accuracy” on page 33-13.
- Use 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. You can specify a tolerance for the numerical correctness by using the **Validation logic tolerance**. The **Validation logic tolerance** is an absolute value. For example, you can specify a tolerance value of `1e-12`.

If the task passes, you see a link to the implementation model.

See Also

More About

- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-7
- “Generate HDL Code from Simscape Models” on page 32-2
- “Deploy Simscape™ Plant Models to Speedgoat FPGA I/O Modules” on page 32-14

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:

```
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

The Simscape HDL Workflow Advisor consists of 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 Switched Linear** task, the task reports the number of differential and algebraic variables. For example, this figure illustrates that there are two differential and two algebraic variables in the boost converter example model.

Details

Number of Discrete Variables: 7

Number of Differential Variables: 2

Source	Value
Simscape_system.AC_Inductor.i.L	Inductor current
Simscape_system.Capacitor.vc	Capacitor voltage

Number of Algebraic Variables: 5

Source	Value
Simscape_system.AC_Inductor.v	Voltage
Simscape_system.AC_Resistor.n.v	Voltage
Simscape_system.Capacitor.i	Current
Simscape_system.D1.n.v	Voltage
Simscape_system.D4.i	Current

By viewing the number of algebraic and differential variables, you can determine how the design consumes resources on the FPGA device. If N_d is the number of differential variables and N_a is the number of algebraic variables, the resource usage on the target hardware varies according to the relation $N_d * (N_d + N_a)$. 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 Extracting Equations

Change Simulation Stop Time

When you run the **Extract 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

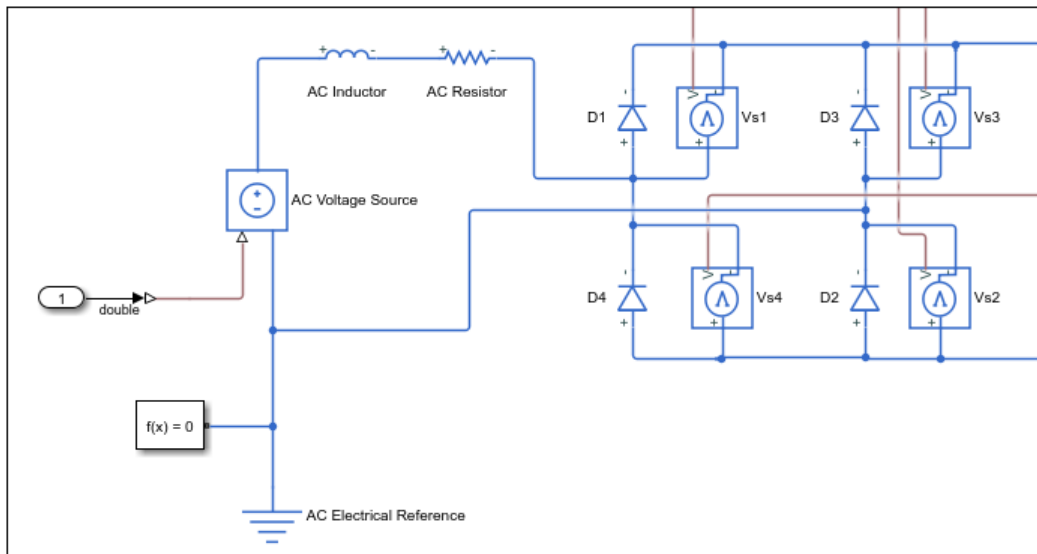
In the **Extract Equations** task, when you extract the differential algebraic equations, the Simscape HDL Workflow Advisor simulates the model to cover the nonlinear range of Simscape blocks. This task can take a long time depending on the number of switching elements in the Simscape model.

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, you do not have to reach the entire 2^n modes. For example, consider this bridge rectifier model. To open this model, enter:

```
open_system('sschdlexBridgeRectifierExample')
```

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 $2^4 = 16$ 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 Discretizing Equations

Change Sample Time

When you run the **Discretize 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 **Discretize 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 in the previous task, **Extract Equations**. The **Discretize Equations** task runs much faster than the **Extract Equations** task because the Advisor only has to discretize the differential algebraic equations to state-space parameters.

The **Discretize Equations** 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?

In the **Generate implementation model** task, you can specify the **Number of solver iterations**. 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. By default, the **Number of Solver iterations** is 4 for switched linear models and 1 for linear models.

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.

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 specify the **Use fixed-cost runtime consistency iterations** setting, you cannot change the **Number of solver iterations** setting on this task. To change the number of solver iterations, on the Solver Configuration block, change the **Nonlinear iterations** parameter and 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 32-59.

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 32-66.

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 becomes the same as the **Number of solver iterations** setting in the **Generate implementation model** task.

By default, **Nonlinear iterations** is set to 2. When you run the **Generate implementation model** task, the Advisor sets the **Number of solver iterations** to 2 and this setting cannot be modified. To modify the **Number of solver iterations**, either:

- Change **Nonlinear iterations**.
- Clear **Use fixed-cost runtime consistency iterations** and then change 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).

Floating-Point Precision and Numerical Accuracy

Use the **Floating-point precision** setting to specify whether you want the algorithm inside the HDL Subsystem in the generated implementation model to use `single` or `double` data types when performing the matrix computations.

Floating-Point Precision	Description
Double	Using double floating-point precision increases the numerical accuracy of the generated model. Using double data types in your model can increase the area consumption and reduce the target frequency.
Single	This is the default setting for floating-point precision.
Single coefficient, double computation	This mode offers a tradeoff between <code>Single</code> and <code>Double</code> modes of floating-point precision. To save memory usage, the coefficients that are stored in <code>single</code> . The matrix computations are then performed in <code>double</code> for improved accuracy.

To learn more about the floating-point precision settings and tradeoffs, see “Use Larger Floating-Point Precision” on page 32-61.

See Also

More About

- “Generate HDL Code from Simscape Models” on page 32-2
- “Deploy Simscape™ Plant Models to Speedgoat FPGA I/O Modules” on page 32-14

Model Protection in HDL Coder

- “Create Protected Models to Conceal Contents and Generate HDL Code” on page 34-2
- “Test Protected Models” on page 34-12
- “Package and Share Protected Models” on page 34-16

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 model reference. The protected model includes derived files to support the optional functionalities that you specify.

How Model Protection Works

This workflow illustrates how you can create a protected model with simulation and HDL code generation support, if you have a HDL Coder license. 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” (Simulink Coder).

When you protect a model, you can allow the user of the protected model to:

- Open a read-only web view of the model, including model contents and block parameters.
- Simulate the model in accelerator (default), rapid accelerator, and normal modes.
- Generate HDL code for a model that includes the protected model.
- Generate C code for a model that includes the protected model, if you have Simulink Coder.
- Generate code for the protected model through the standalone interface, if you have Embedded Coder 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:

- The Model block context menu.
- The `Simulink.ModelReference.protect` function.
- The Simulink Editor menu bar. To create a protected model from the current model, select **File > Export Model To > Protected Model**.

When you create a protected model:

- By default, Simulink creates and stores a protected version of the model in the current working folder. The protected model 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.
- The protected model file consists of the model itself and supporting files, depending on the options that you select when you create the protected model.

If your protected model requires supporting files, such as base workspace definitions or a data dictionary, include these files with the model when you share the protected model.

General Protected Model Requirements and Limitations

When you create a protected model, consider these requirements:

- You must have a HDL 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” (Simulink).

Protected Model Restrictions for HDL Code Generation

These configurations are not supported when you create a protected model that has HDL code generation support.

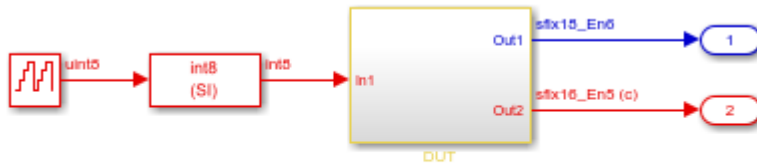
- The protected model must use the same configuration parameters as the top level that it is referenced from.
- The solver settings that you specify in the **Solver** pane of the Configuration Parameters dialog box must be *Fixed-step* and *auto*.
- You must not enable these settings in the Configuration 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” (Simulink Coder).

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 `hdlcoder_protected_model_parent_harness`.

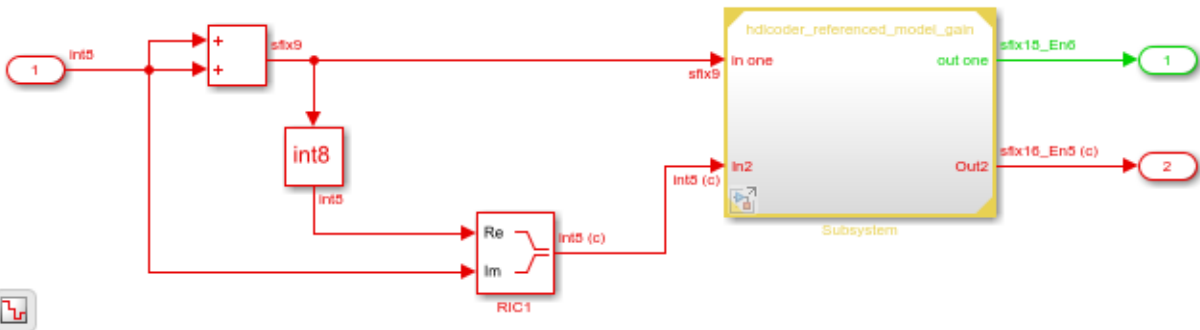
```
open_system('hdlcoder_protected_model_parent_harness')  
set_param('hdlcoder_protected_model_parent_harness','SimulationCommand','Update')
```



Copyright 2015 The MathWorks, Inc.

Navigate to the Model block in your parent model. If you double-click the DUT Subsystem, and then open the mynested Subsystem, you see a Model block that references the model `hdlcoder_referenced_model_gain`.

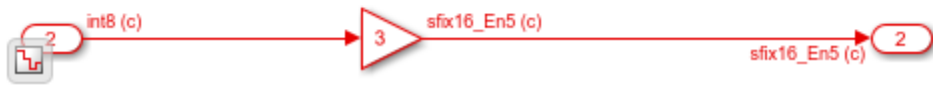
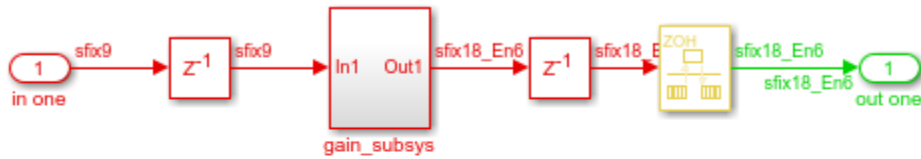
```
open_system('hdlcoder_protected_model_parent_harness/DUT/mynested')
```



Open the Model block and make sure that a *modelname* with the extension `.slx` is specified in the **Model name** field. 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.

In this case, the Model block is referencing the model `hdlcoder_referenced_model_gain.slx`, the model that you want to protect. Double-click the Model block or open the model `hdlcoder_referenced_model_gain` in a separate window.

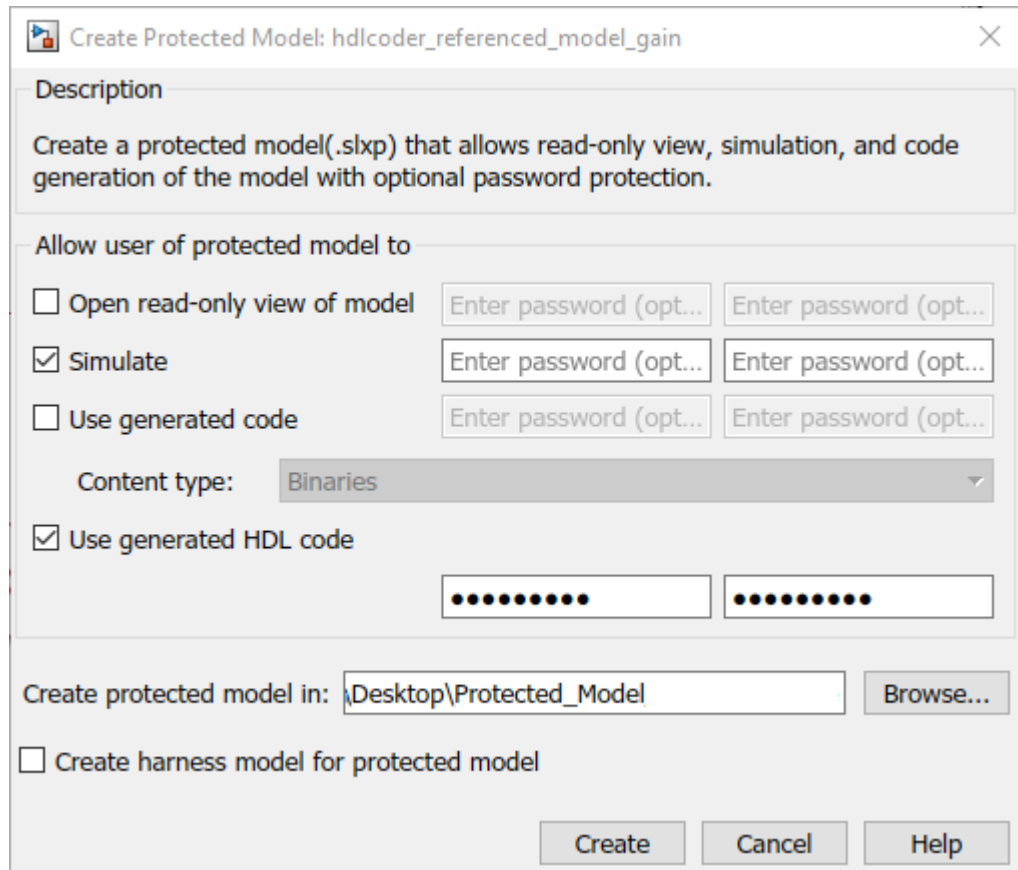
```
open_system('hdlcoder_referenced_model_gain')
set_param('hdlcoder_referenced_model_gain', 'SimulationCommand', 'Update')
```



Protect the Referenced Model

In this example, you use the context menu to convert a Model block to a protected model.

- 1 Right-click the Model block and select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**.



- 2 In the Create Protected Model dialog box, select the **Simulate** dialog box. This option allows the protected model user to simulate the model that references the protected model.
- 3 If you have Simulink Coder or Embedded Coder, you can specify additional settings such as enable code generation support with password protection by using the **Use generated code** check box or specify a **Code interface**. To learn more about these options, see “Protect Models to Conceal Contents” (Embedded Coder).
- 4 Select the **Use generated HDL code** check box 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.

- 5 In the **Create protected model in** field, specify the folder path for the protected model. The default value is the current working folder.
- 6 To create a harness model for the protected model, select the **Create harness model for protected model** check box. In this example, leave the check box cleared. Click **Create**.

HDL Coder then checks compatibility of the model for HDL code generation and 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 34-16.

If you selected the check box when you created the protected model, the harness model opens as a new, untitled model that contains only a Model block that references the protected model. To use the protected model, reference it through a Model block such as the one included in the harness model. The **Simulation mode** for the Model block is set to **Accelerator**. You cannot change the mode. For more information, see “Reference Protected Models from Third Parties” (Simulink).

To learn more about these UI options, see “Create Protected Model” (Simulink Coder).

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 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 screenshot shows a window titled "Protected Model report" with a search bar and "Match Case" option. The left sidebar contains a "Contents" panel with "Summary" selected. The main content area displays the following sections:

Summary for hdlcoder_referenced_model_gain

Environment

Environment information for protected model "hdlcoder_referenced_model_gain"

Model Version	1.47
Simulink version	9.3
Simulink Coder Version	9.1 (R2019a) 20-Nov-2018
HDL Coder version	3.14
Protected model generated on	Fri Dec 7 11:42:56 2018
Platform	win64

Configuration settings at the time of protected model creation: [click to open](#)

Supported functionality

Supported functionality for protected model "hdlcoder_referenced_model_gain"

Read-only view support	Off
Simulation support	On
Code generation support	Off
HDL Code generation support	On with password protection
Concurrent tasking support	Off

Licenses

Licenses required to use protected model "hdlcoder_referenced_model_gain"

- Simulink
- Fixed-Point Designer
- HDL Coder

Buttons for "OK" and "Help" are located at the bottom right of the window.

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.
 - **Licenses**, listing licenses required to run the protected model.
- An **Interface Report**, including model interface information such as input and output specifications, interface parameters, and data stores.

To generate a report when using the `Simulink.ModelReference.protect` function, set the 'Report' option to True.

Generate HDL Code for Models Referencing Protected Model

If the protected model created 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

Functions

`Simulink.ModelReference.modifyProtectedModel` |
`Simulink.ModelReference.protect`

More About

- “Test Protected Models” on page 34-12
- “Package and Share Protected Models” on page 34-16

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 34-2.

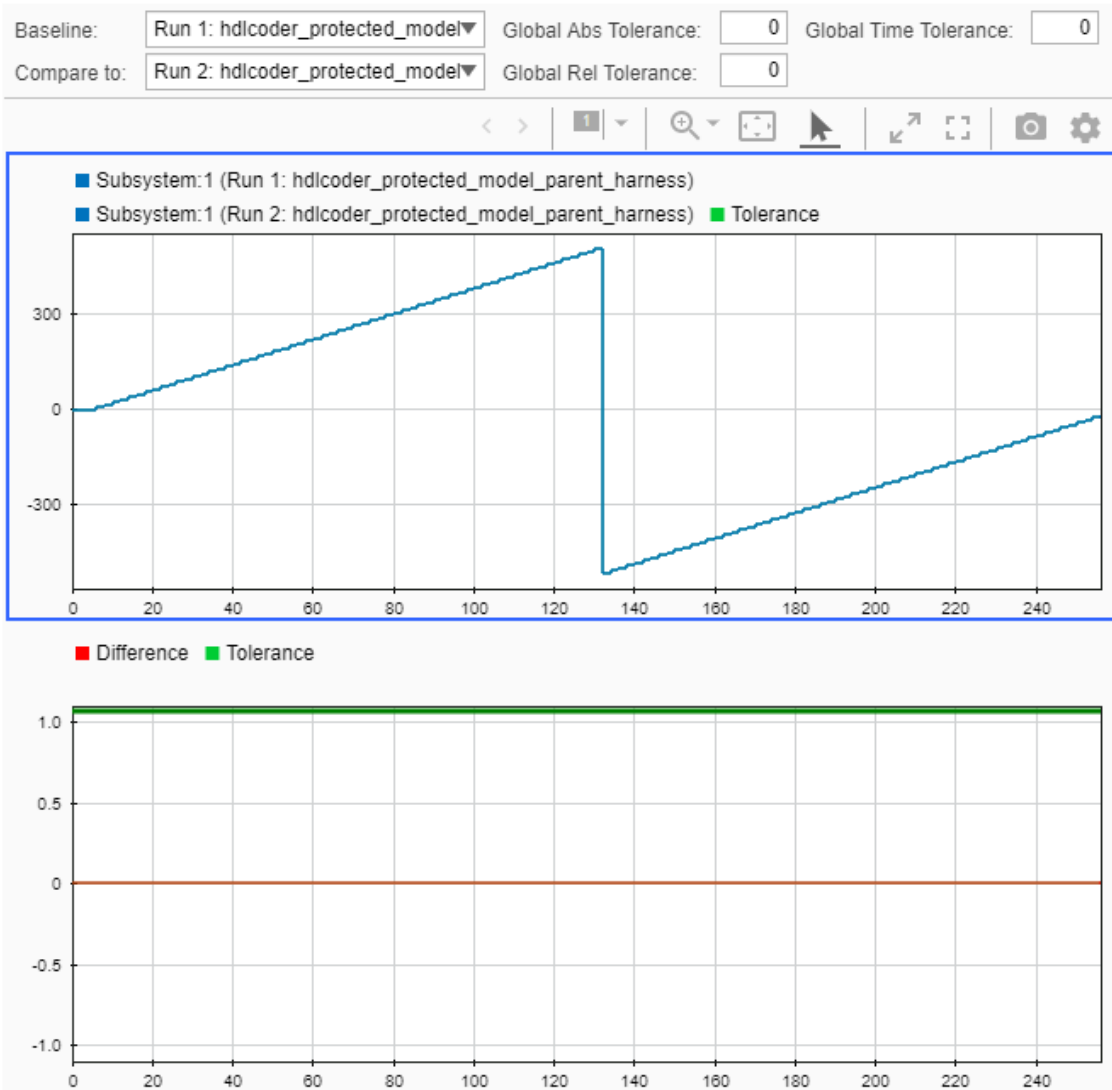
- 1 If it is not already open, open the model `hdlcoder_protected_model_parent_harness`.

```
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 34-2.

- 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.modifyProtectedModel` |
`Simulink.ModelReference.protect`

More About

- “Export Signal Data Using Signal Logging” (Simulink)
- “Compare Simulation Data” (Simulink)
- “Package and Share Protected Models” on page 34-16

Package and Share Protected Models

In addition to the protected model file (.slxp), you can include additional files in the protected model package. Some ways to deliver the protected model package are:

- Provide the .slxp file and other supporting files as separate files.
- Combine the files into a ZIP or other container file.
- Combine the files by using a manifest. For more information, see “Export Files in a Manifest” (Simulink).
- 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.

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.

The following base workspace variables 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 “Protected Models for Model Reference” (Simulink). 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 selected when you created the protected model. The derived files are unpacked when you or a third-party use the protected model in simulation. You do not need to package these derived files with the protected model.

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 illustrates the files that are unpacked depending on the options that you specified. If you specified the **Use generated code** or **Code Interface** options when creating the protected model, additional files are unpacked in the derived folder. To learn about these files, see “Protected Model File Contents” (Simulink Coder).

Protected Model Derived Files

Supported Functionality	Derived Files
Created a protected model for simulation only and the referencing model is in Normal mode	The <i>model.mexext</i> file is placed in the build folder.
Created protected model for simulation only and referencing model is in Accelerator or Rapid Accelerator mode.	<p>These files are unpacked in the <i>slprj/sim/</i> folder:</p> <ul style="list-style-type: none"> • <i>slprj/sim/model/*.h</i> • <i>slprj/sim/model/modellib.a</i> (or <i>modellib.lib</i>) • <i>slprj/sim/model/tmwinternal/*</i> • <i>slprj/sim/_sharedutils/*</i> <p>For the protected model report, these additional files are unpacked (but not in the build folder):</p> <ul style="list-style-type: none"> • <i>slprj/sim/model/html/*</i> • <i>slprj/sim/model/buildinfo.mat</i>
Created protected model with HDL code generation support.	<p>The files are unpacked in the <i>hdlsrc</i> folder: (Additional files depend on whether you enabled support for other options such as code generation).</p> <ul style="list-style-type: none"> • <i>hdlsrc/model/model.vhd</i> (or <i>model.v</i> if you specified Verilog as the Target language). • <i>hdlsrc/model/Subsystem.vhd</i> (or <i>Subsystem.v</i> 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). • <i>hdlsrc/model/model_pkg.vhd</i> (This file is not generated if you specified Verilog as the Target language of the model that you protected). • <i>hdlsrc/model/model_report.html</i> • <i>hdlsrc/model/gm_model.slxp</i> (This is a generated protected model. If you use cosimulation, HDL Coder instantiates this generated protected model).

See Also

Functions

`Simulink.ModelReference.modifyProtectedModel` |
`Simulink.ModelReference.protect`

More About

- “Create Protected Models to Conceal Contents and Generate HDL Code” on page 34-2
- “Test Protected Models” on page 34-12

HDL Test Bench

- “Test Bench Generation” on page 35-2
- “Test Bench Block Restrictions” on page 35-5

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.

In this section...
“How Test Bench Generation Works” on page 35-2
“Test Bench Data Files” on page 35-2
“Test Bench Data Type Limitations” on page 35-3
“Use Constants Instead of File I/O” on page 35-3

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 35-3.

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 35-5
- “Choose a Test Bench for Generated HDL Code” on page 27-38

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 36-2
- “Create Custom FPGA Board Definition” on page 36-8
- “Create Xilinx KC705 Evaluation Board Definition File” on page 36-9
- “FPGA Board Manager” on page 36-22
- “New FPGA Board Wizard” on page 36-26
- “FPGA Board Editor” on page 36-39

FPGA Board Customization

In this section...
“Feature Description” on page 36-2
“Custom Board Management” on page 36-2
“FPGA Board Requirements” on page 36-3

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 36-22: portal to adding, importing, deleting, and otherwise managing board definition files.
- “New FPGA Board Wizard” on page 36-26: 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 36-39: user interface for viewing or editing board information.

To begin, review the “FPGA Board Requirements” on page 36-3 and then follow the steps described in “Create Custom FPGA Board Definition” on page 36-8.

FPGA Board Requirements

- “FPGA Device” on page 36-3
- “FPGA Design Software” on page 36-3
- “General Hardware Requirements” on page 36-3
- “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 36-4
- “JTAG Connection Requirements for FPGA-in-the-Loop” on page 36-6

FPGA Device

Select one of the following links to view a current list of supported FPGA device families:

- For use with FPGA-in-the-loop (FIL), see “Supported FPGA Device Families for Board Customization” (HDL Verifier).
- For use with FPGA Turnkey, see “Supported FPGA Device Families for Board Customization”.

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 or FPGA Turnkey.

Workflow	Required Tools
FPGA-in-the-loop	<ul style="list-style-type: none"> • HDL Verifier • Fixed-Point Designer
FPGA Turnkey	<ul style="list-style-type: none"> • HDL Coder • Simulink • 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 36-4).

- **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 36-4
- “Ethernet PHY Interface” on page 36-5
- “Special Timing Considerations for RGMII” on page 36-5
- “Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface” on page 36-5

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 36-31.

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	<ul style="list-style-type: none"> • 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 <i>before</i> 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. <ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • For Windows operating systems: Xilinx Vivado executable directory must be on system path. • For Linux operating systems: Digilent Adept2

Vendor	Required Hardware	Required Software
	FTDI USB-JTAG cable <ul style="list-style-type: none">• Supported for boards with onboard FT4232H, FT232H, or FT2232H devices implementing USB-to JTAG	Supported for Windows operating systems. <hr/> Note FTDI USB JTAG support is only available for MATLAB as AXI Master and for FPGA Data Capture. <hr/>

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 New Board**. For a description of all the tasks you can perform with the FPGA Board Manager, see “FPGA Board Manager” on page 36-22.
- 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 36-26.
- 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 36-18.

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 36-9 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 36-9
“What You Need to Know Before Starting” on page 36-9
“Start New FPGA Board Wizard” on page 36-10
“Provide Basic Board Information” on page 36-11
“Specify FPGA Interface Information” on page 36-13
“Enter FPGA Pin Numbers” on page 36-14
“Run Optional Validation Tests” on page 36-16
“Save Board Definition File” on page 36-18
“Use New FPGA Board” on page 36-19

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.

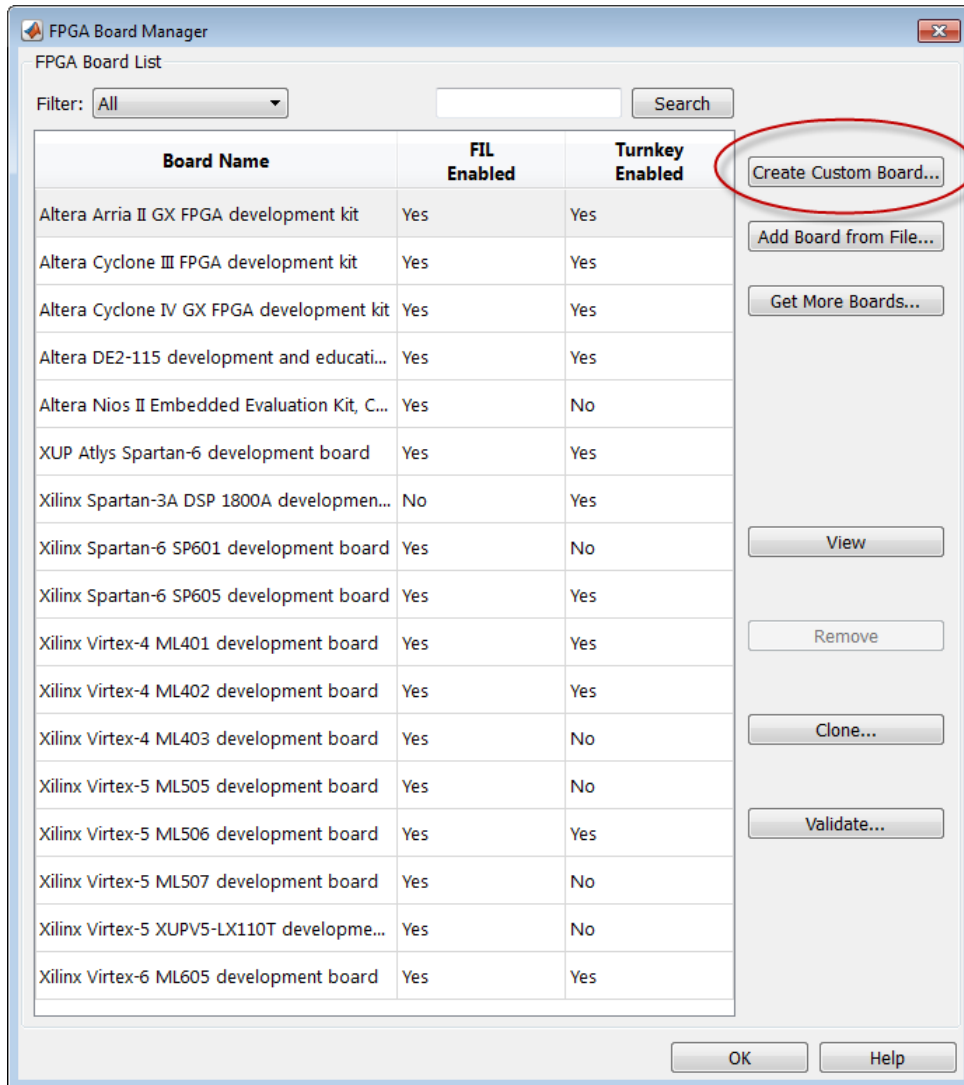
- For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

- 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:

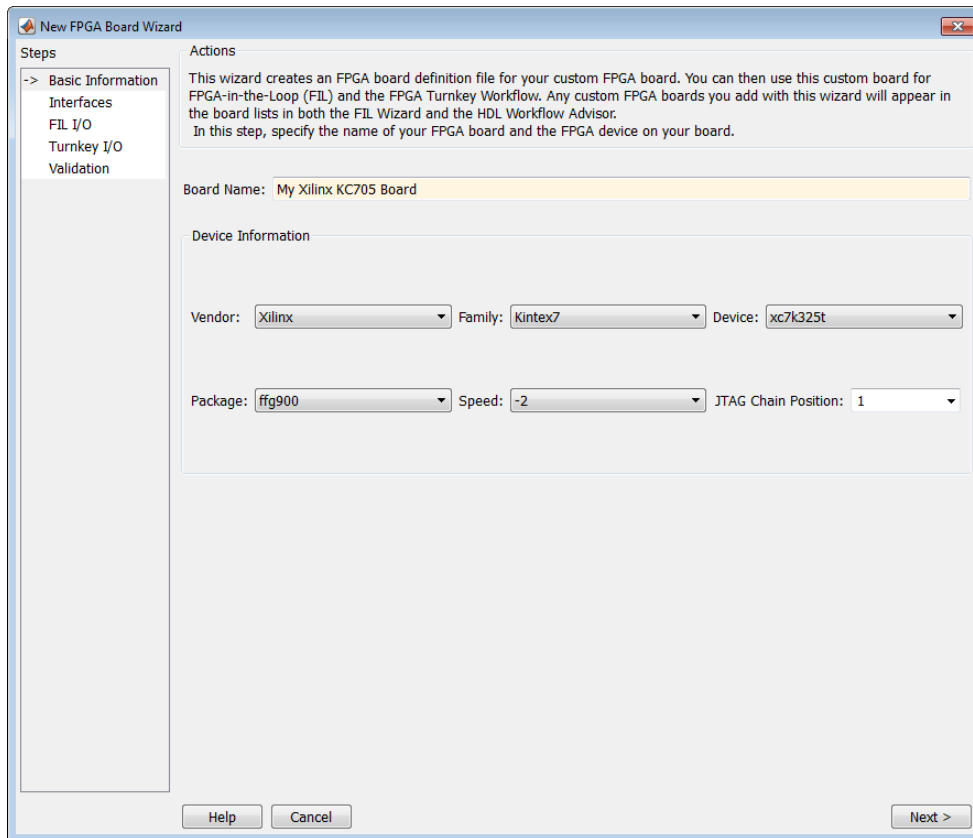
```
>>fpgaBoardManager
```
- 2** Click **Create Custom Board** to open the New FPGA Board wizard.



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



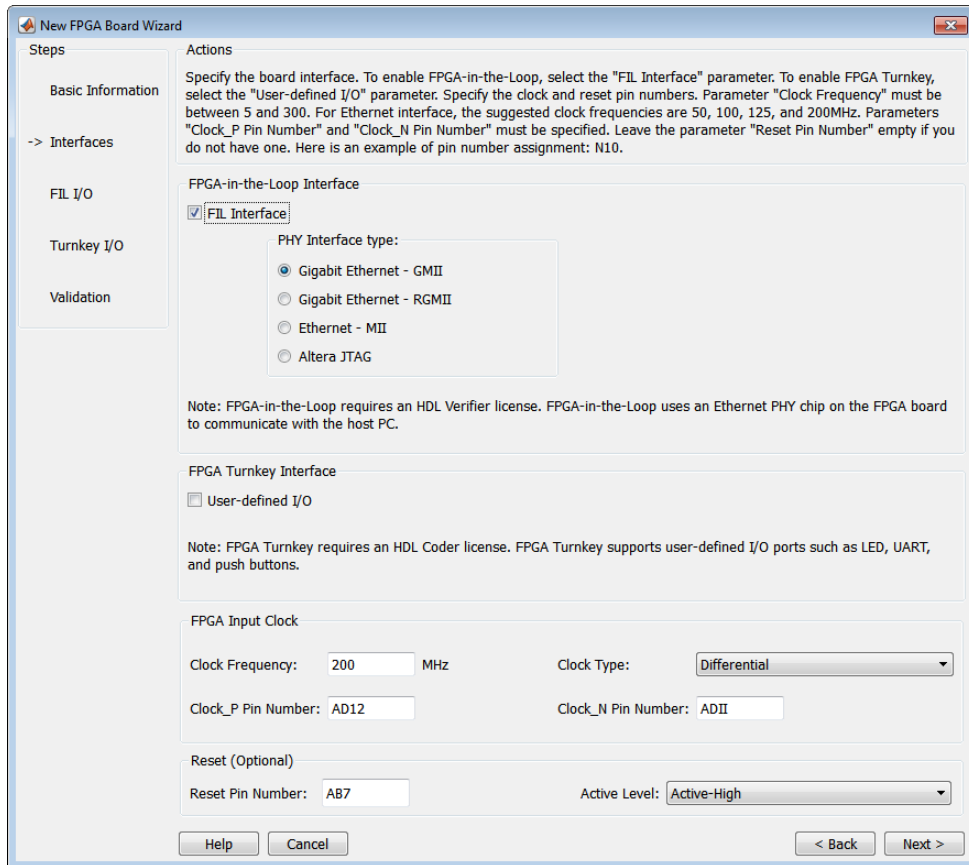
The information you just entered can be found in the KC705 Evaluation Board for the Kintex-7 FPGA User Guide.

- 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 FIL/I/O pane, enter the numbers for each FPGA pin. This information is required.

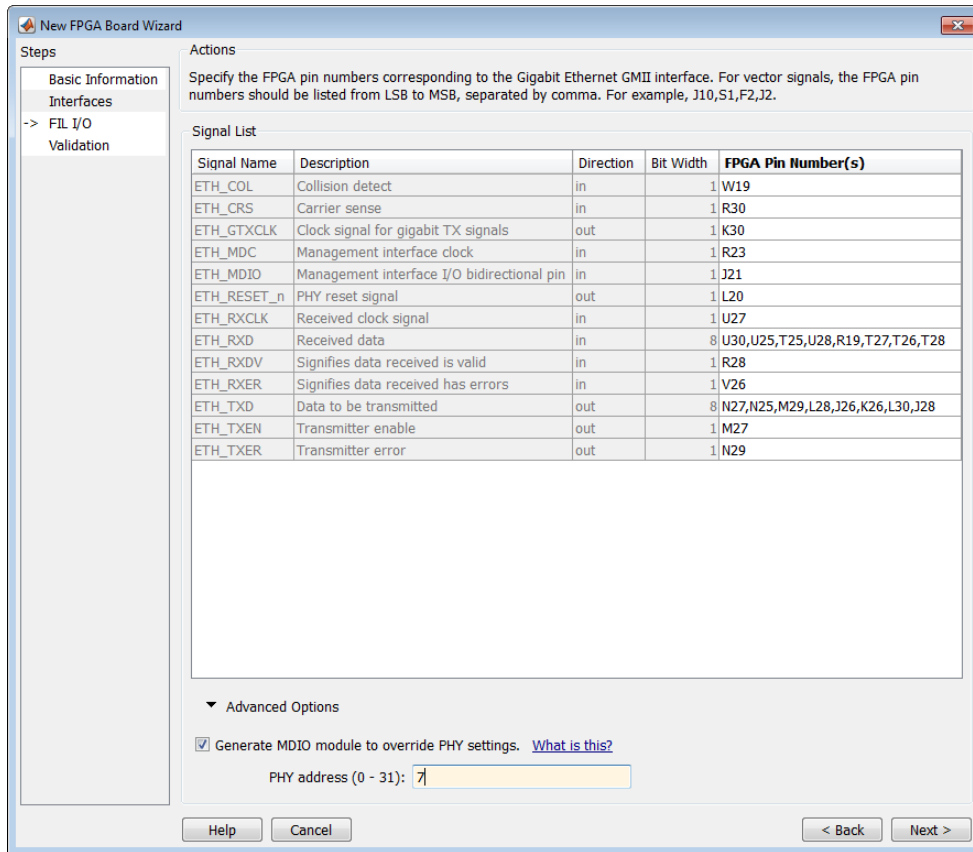
Pin numbers for RXD and TXD signals are entered from the least significant digit (LSD) to the most significant digit (MSB), separated by a comma.

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
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**.

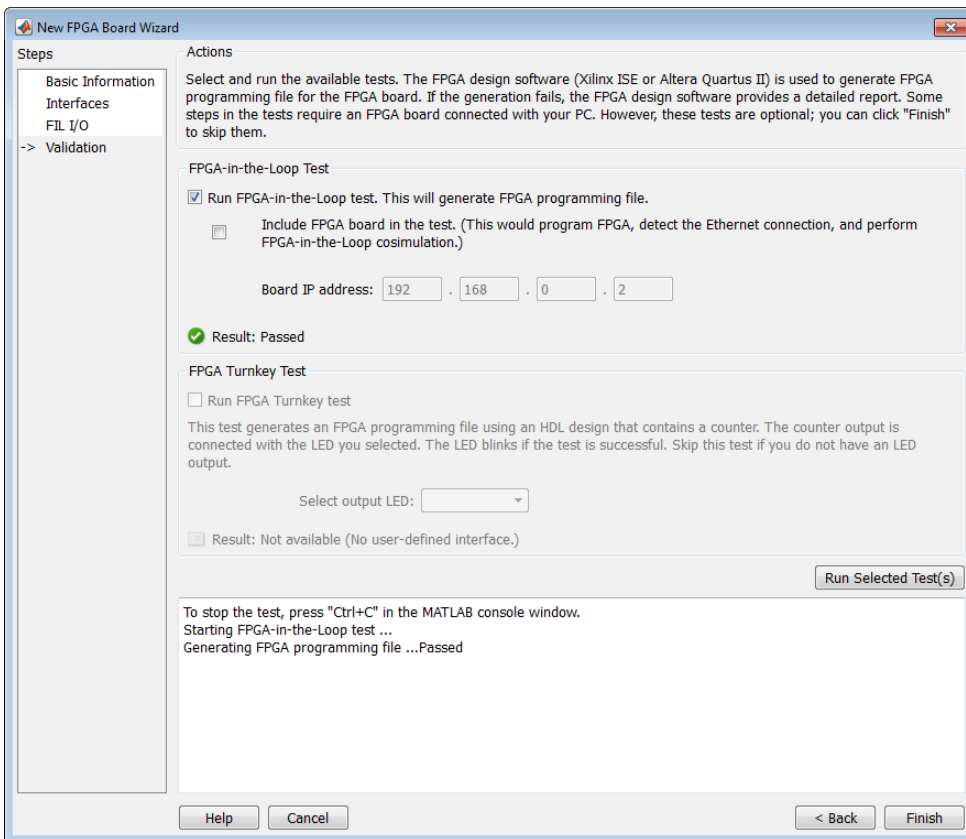
Run Optional Validation Tests

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.

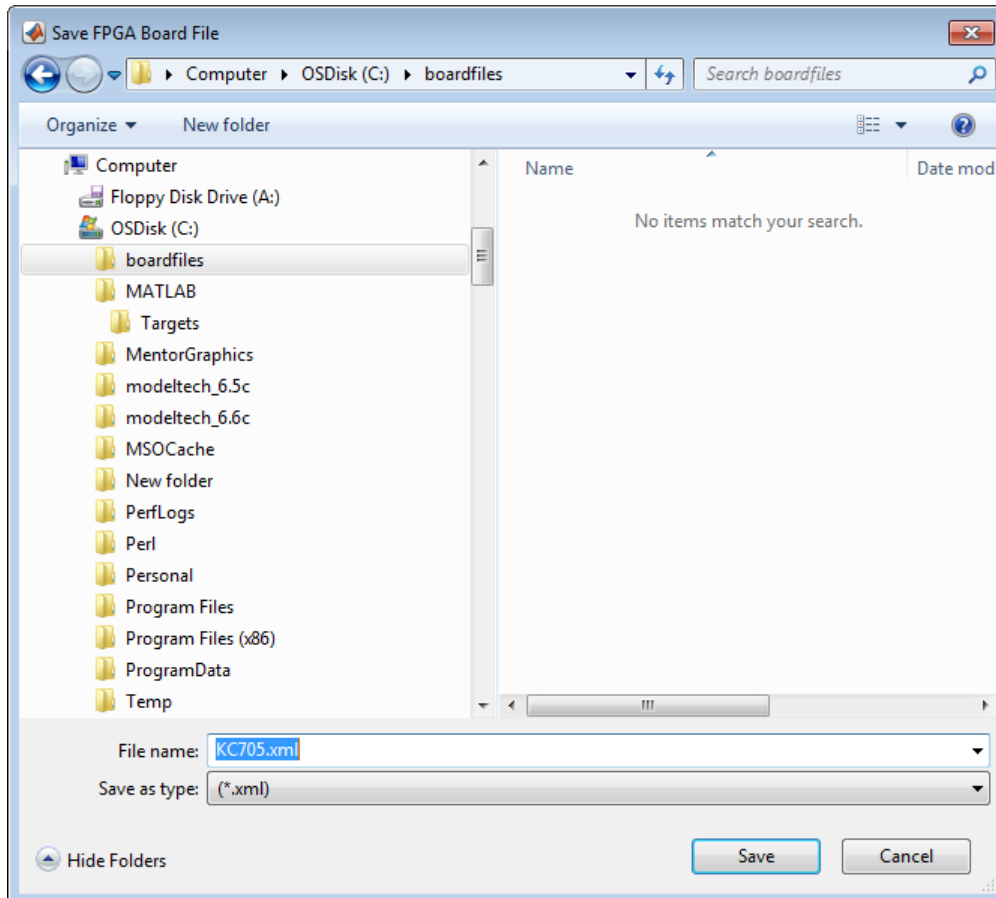
To run this test, perform the following actions.

- 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

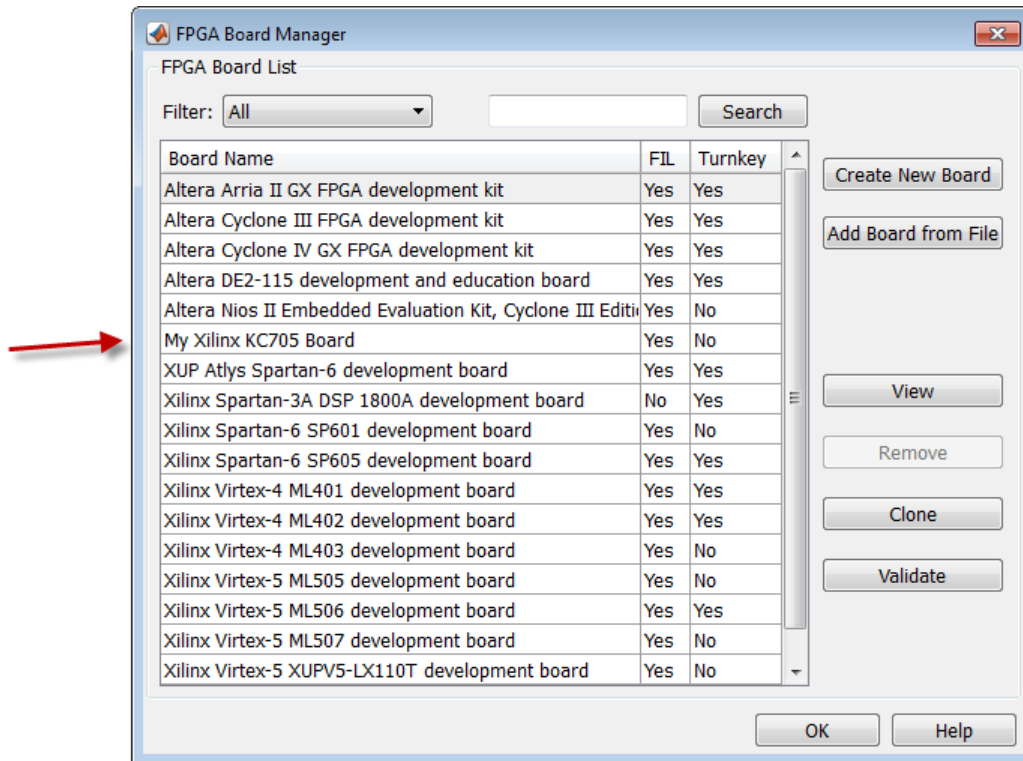
- 1 Click **Finish** to exit the New FPGA Board wizard. A **Save As** dialog box pops up and asks for the location of the FPGA board definition file. For this example, save as `C:\boardfiles\KC705.xml`.



- 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.

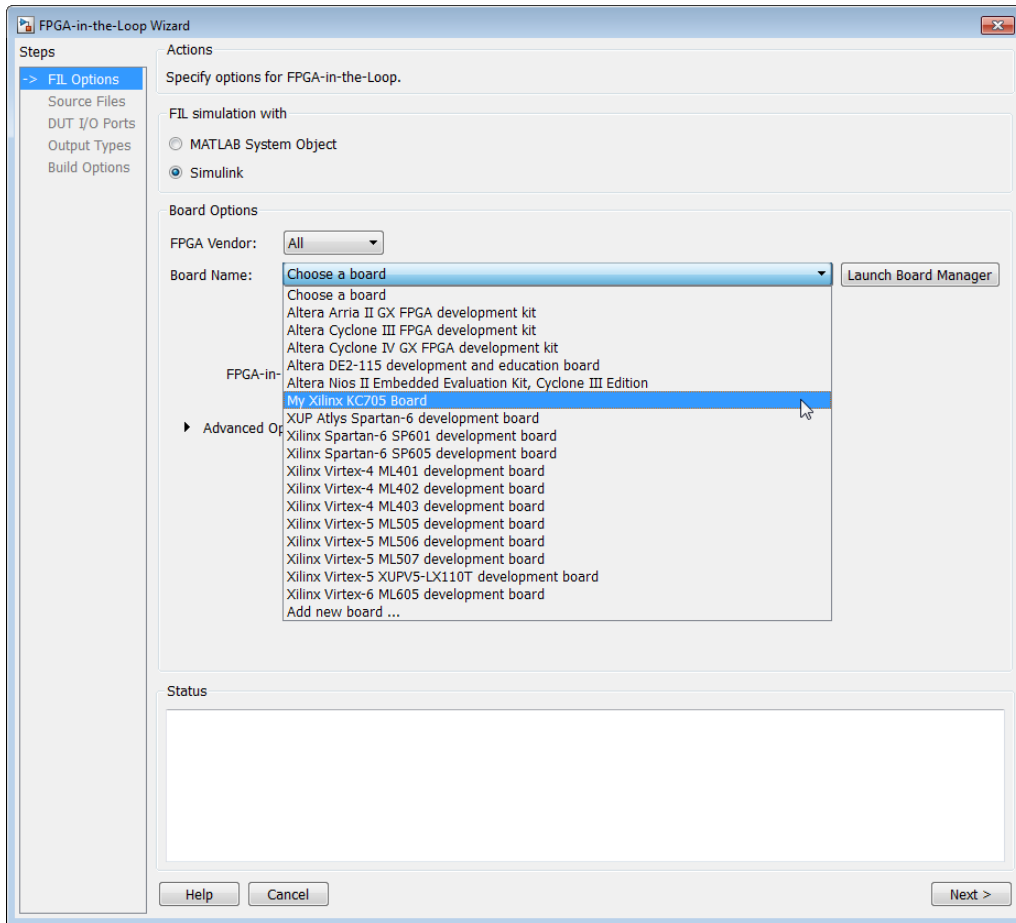


Click **OK** to close the FPGA Board Manager.

- 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
```

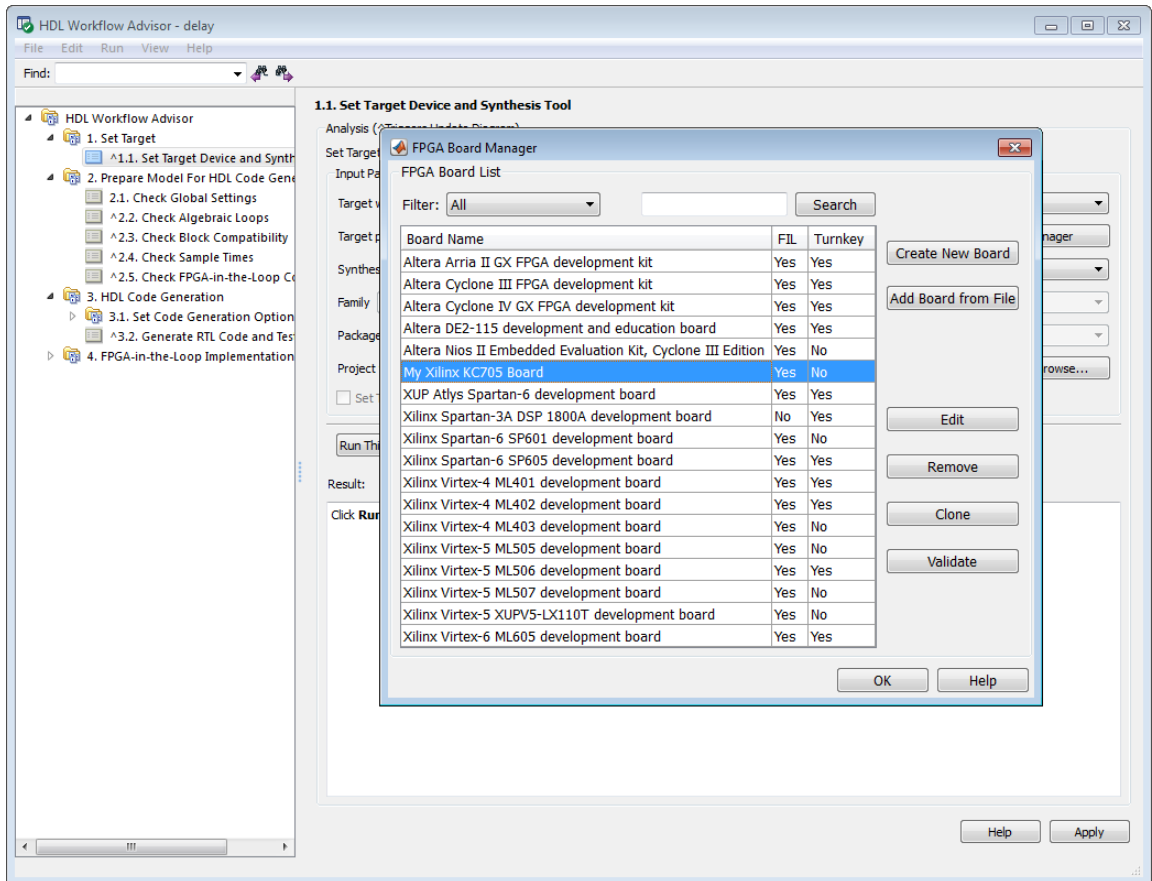
The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.



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.



FPGA Board Manager

In this section...

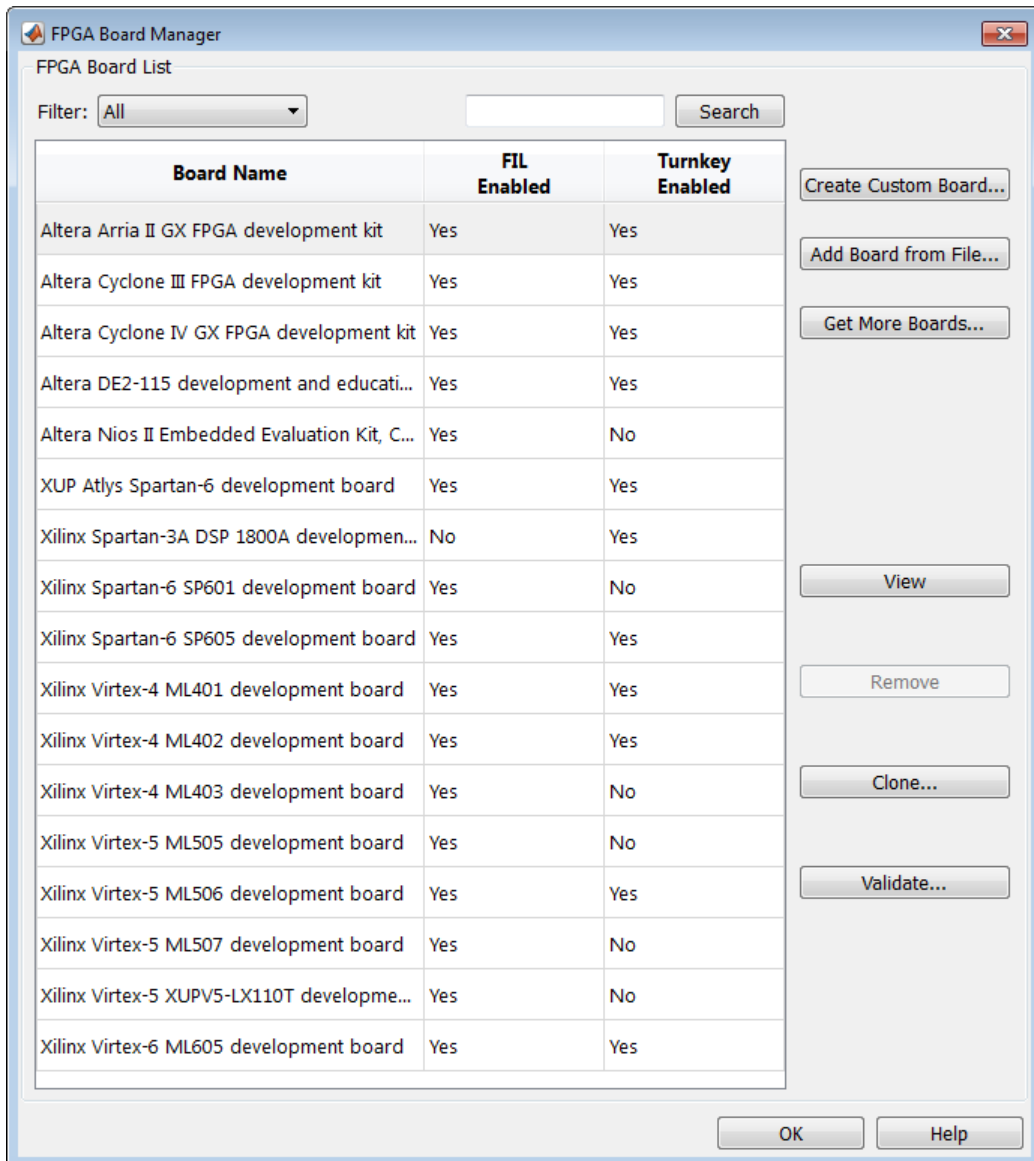
“Introduction” on page 36-22
“Filter” on page 36-24
“Search” on page 36-24
“FIL Enabled/Turnkey Enabled” on page 36-24
“Create Custom Board” on page 36-24
“Add Board from File” on page 36-24
“Get More Boards” on page 36-24
“View/Edit” on page 36-25
“Remove” on page 36-25
“Clone” on page 36-25
“Validate” on page 36-25

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 36-26. You can find the process for creating a board definition file in “Create Custom FPGA Board Definition” on page 36-8.

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 36-39.

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 36-16.

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 36-3 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 36-27
“Interfaces” on page 36-28
“FIL I/O” on page 36-31

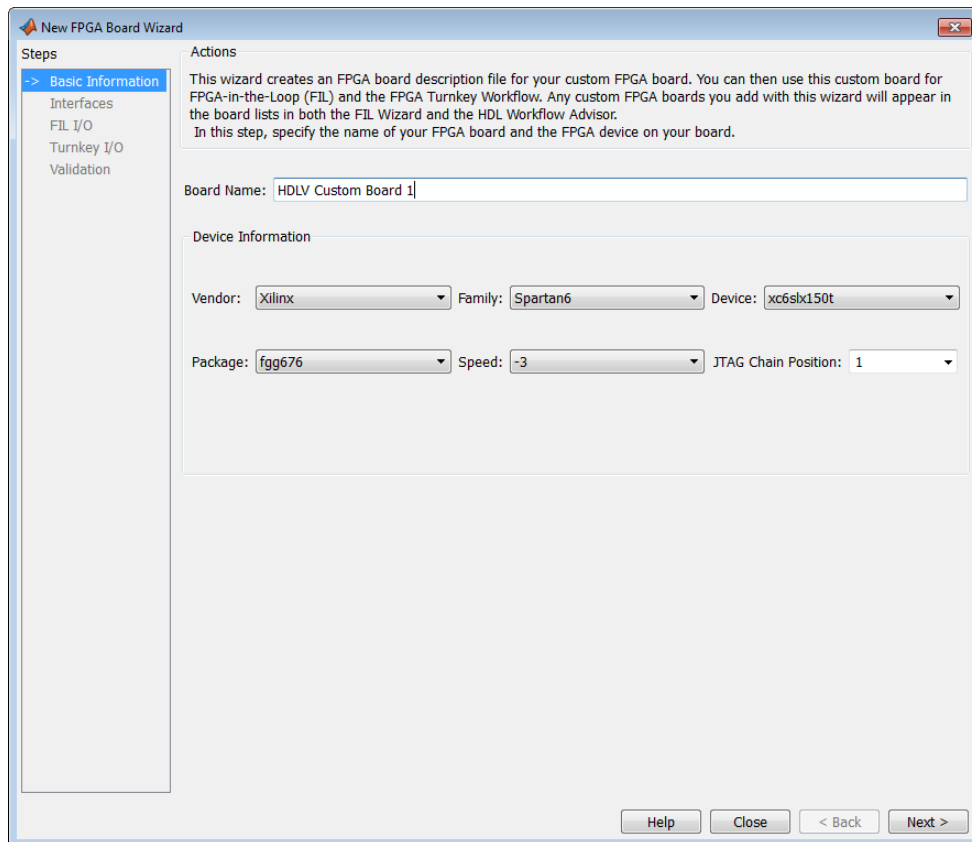
In this section...

“Turnkey I/O” on page 36-34

“Validation” on page 36-37

“Finish” on page 36-38

Basic Information



The screenshot shows the "New FPGA Board Wizard" dialog box. The "Steps" pane on the left lists "Basic Information", "Interfaces", "FIL I/O", "Turnkey I/O", and "Validation". The "Basic Information" step is selected and highlighted. The "Actions" pane contains the following text: "This wizard creates an FPGA board description file for your custom FPGA board. You can then use this custom board for FPGA-in-the-Loop (FIL) and the FPGA Turnkey Workflow. Any custom FPGA boards you add with this wizard will appear in the board lists in both the FIL Wizard and the HDL Workflow Advisor. In this step, specify the name of your FPGA board and the FPGA device on your board." Below this text is a text input field for "Board Name" containing "HDLV Custom Board 1". Under the "Device Information" section, there are several dropdown menus: "Vendor" is set to "xilinx", "Family" is set to "Spartan6", "Device" is set to "xc6sxc150t", "Package" is set to "fgg676", "Speed" is set to "-3", and "JTAG Chain Position" is set to "1". At the bottom right, there are buttons for "Help", "Close", "< Back", and "Next >".

Board Name: Enter a unique board name.

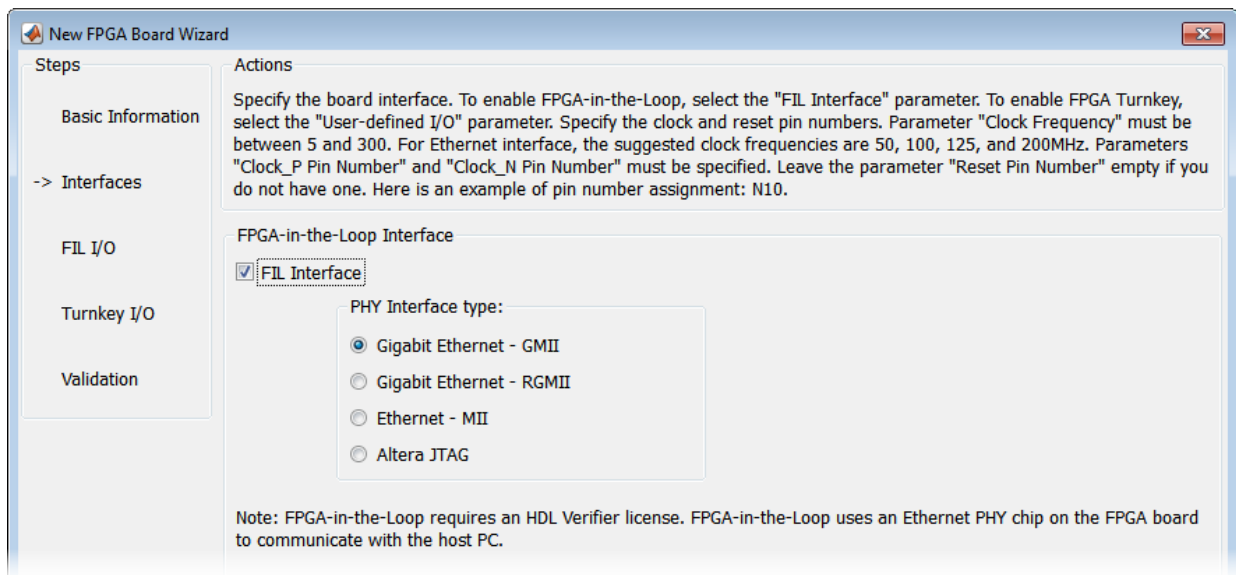
Device Information:

- **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 36-28
- “FIL Interface for Xilinx Boards” on page 36-29
- “FPGA Turnkey Interface” on page 36-30
- “FPGA Input Clock and Reset” on page 36-30

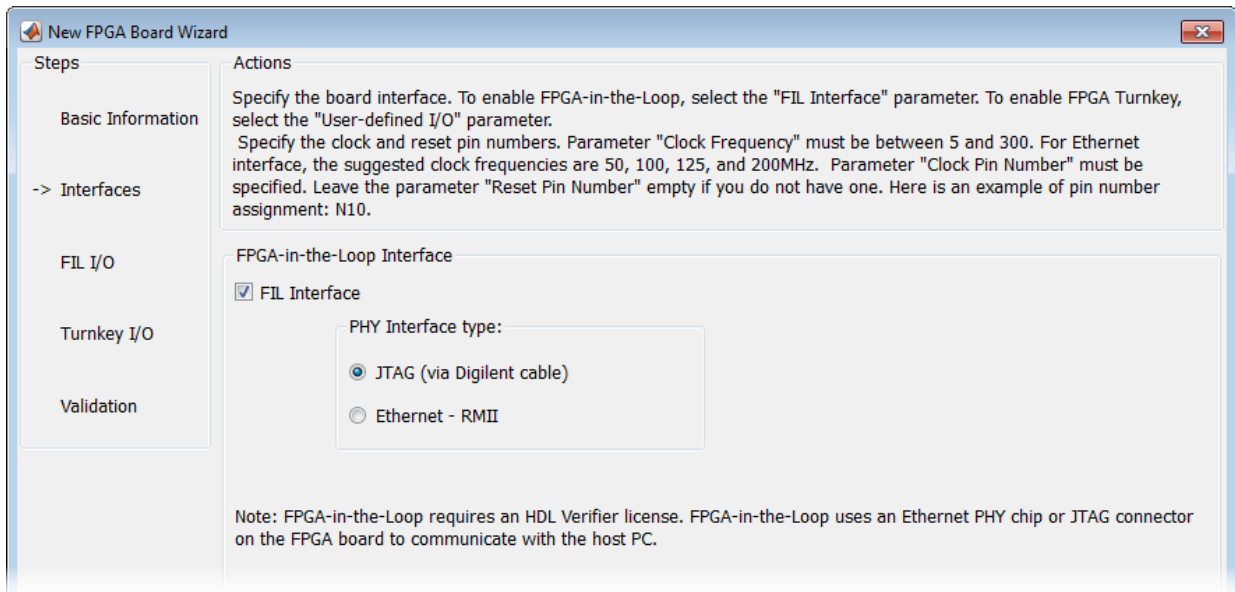
FIL Interface for Altera Boards



- 1 **FPGA-in-the-Loop:** To use this board with FIL, select **FIL Interface**.
- 2 Select one of the following **PHY Interface types**:
 - **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 types**:

- **JTAG (via Digilent cable)** (Xilinx boards only)
- **Ethernet – RMI**

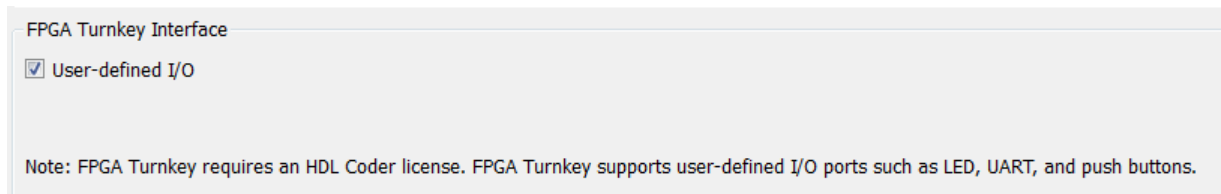
Note Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

For more information on how to set up the JTAG connection for Xilinx boards, see “JTAG with Digilent Cable Setup” on page 36-41.

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.

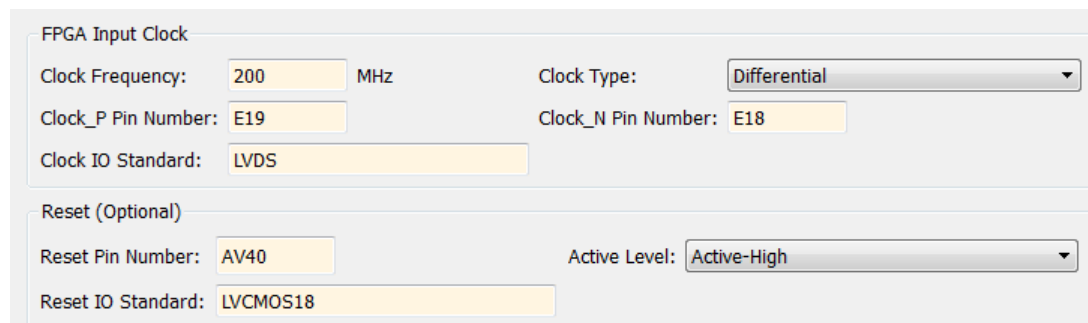
FPGA Turnkey Interface



The screenshot shows a configuration window titled "FPGA Turnkey Interface". It contains a checked checkbox labeled "User-defined I/O". Below the checkbox is a note: "Note: FPGA Turnkey requires an HDL Coder license. FPGA Turnkey supports user-defined I/O ports such as LED, UART, and push buttons."

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



The screenshot shows a configuration window titled "FPGA Input Clock" and "Reset (Optional)".

FPGA Input Clock:

- Clock Frequency: 200 MHz
- Clock Type: Differential
- Clock_P Pin Number: E19
- Clock_N Pin Number: E18
- Clock IO Standard: LVDS

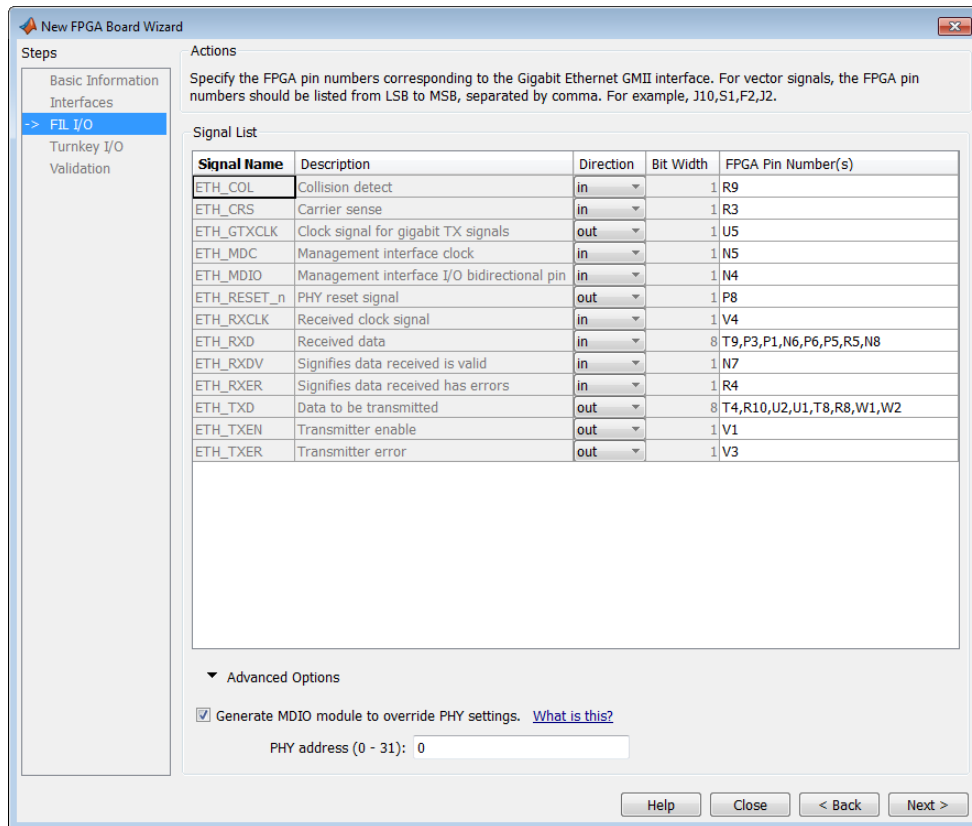
Reset (Optional):

- Reset Pin Number: AV40
- Active Level: Active-High
- Reset IO Standard: LVCMOS18

- 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:

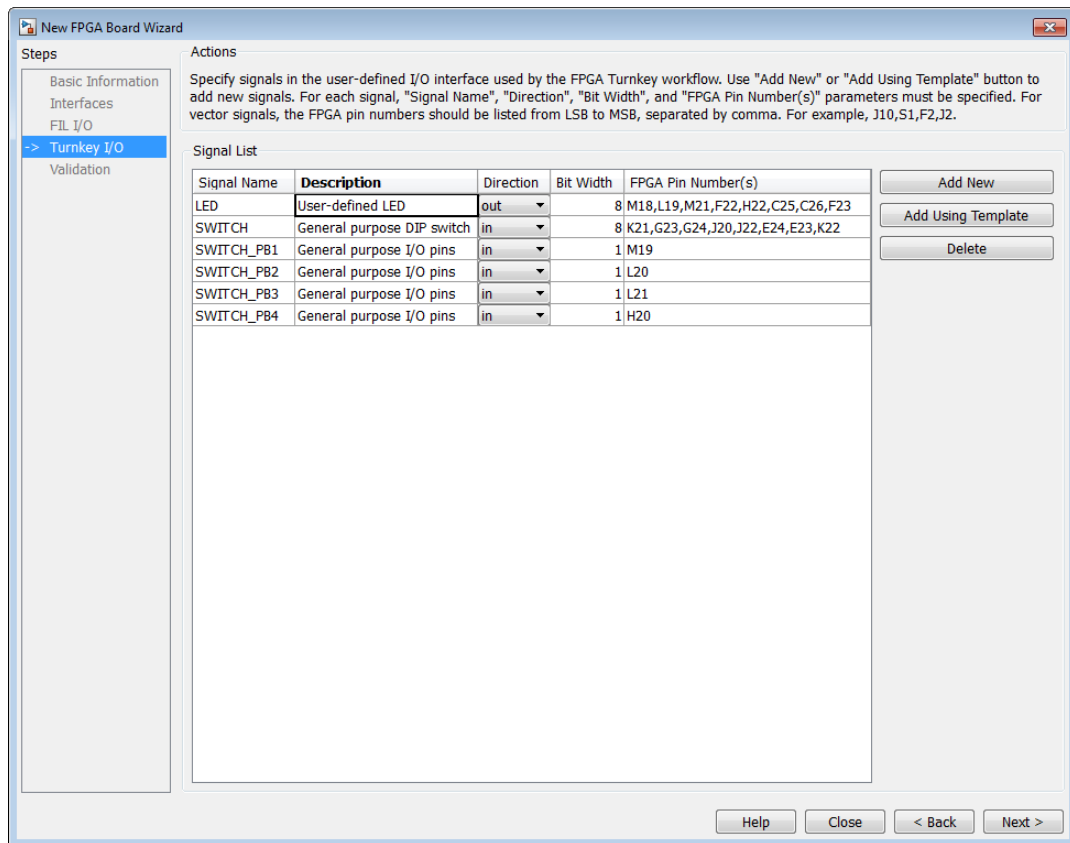
- **GMII mode:** The PHY device can start up using other modes, such as RGMII/SGMII. The generated MDIO module sets the PHY chip in GMII mode.
- **RGMII mode:** The PHY device can start up using other modes, such as GMII/SGMII. The generated MDIO module sets the PHY device in RGMII mode. In addition, 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 Mb/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:

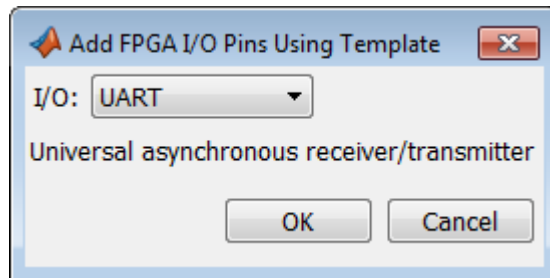
- A generic signal name
- Description
- Direction
- Bit width

You can change the values in any of these prepopulated fields.

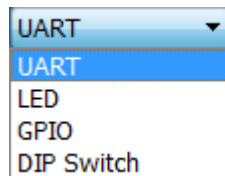
Delete: Delete the selected signal from list.

The following example demonstrates using the **Add Using Template** feature.

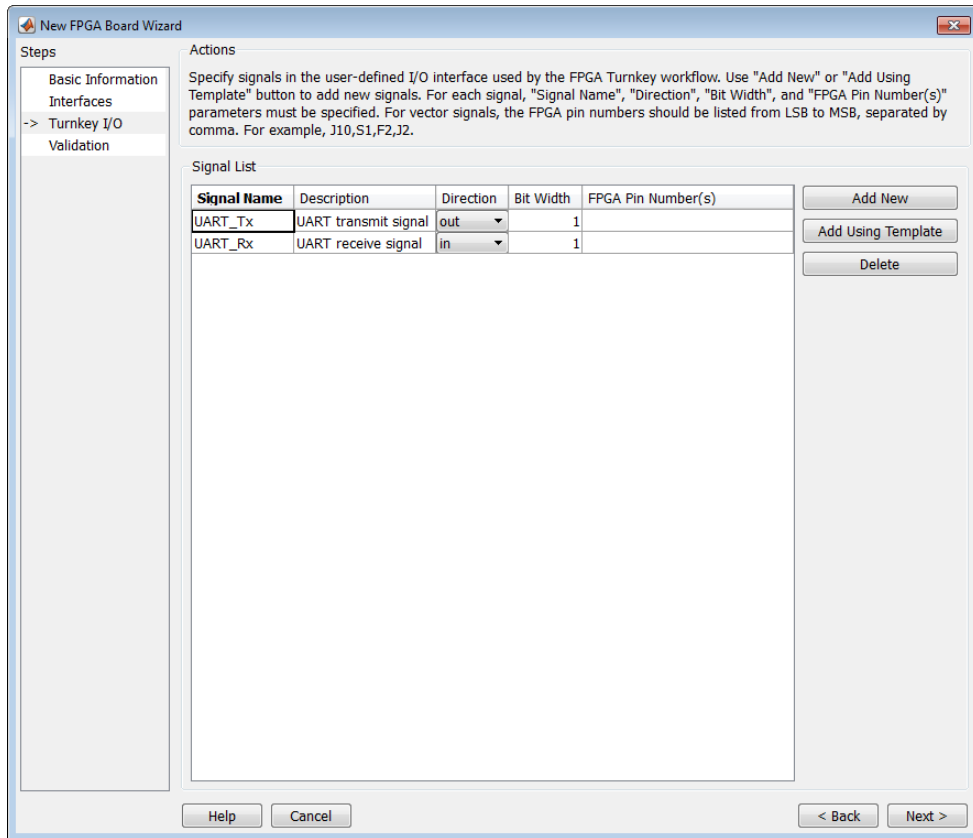
- 1 In the Turnkey I/O dialog box, click **Add Using Template**.
- 2 You can now view the template dialog box.



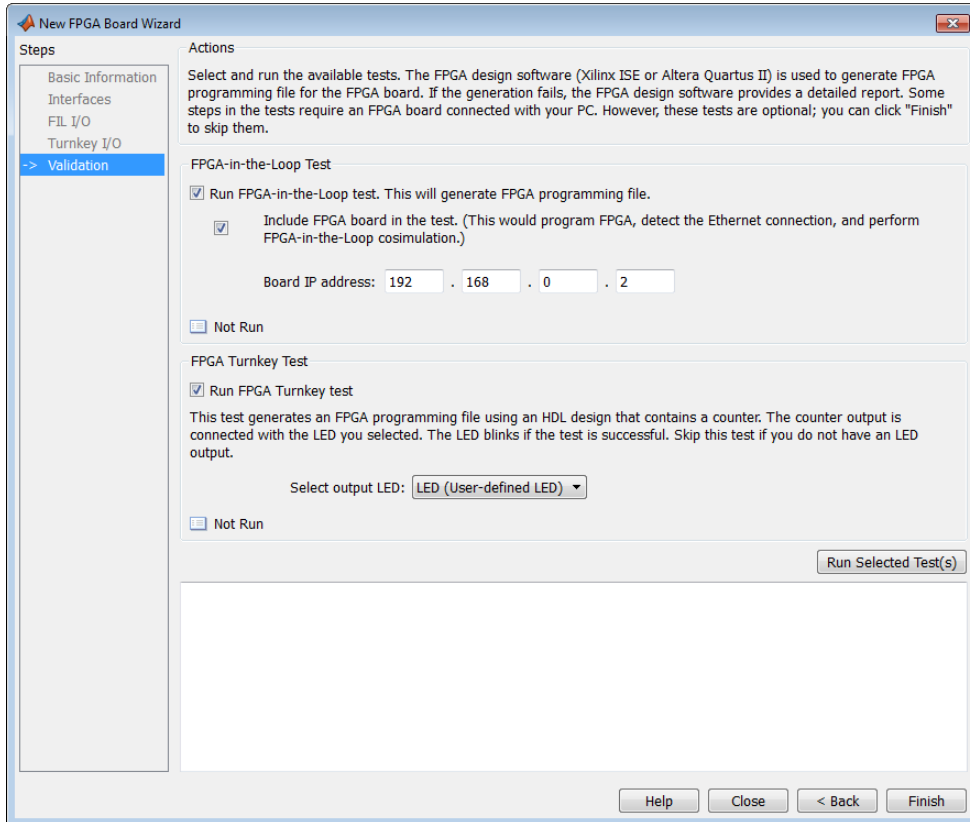
- 3 Pull down the I/O list and select from the following options:



- 4 Click **OK**.
- 5 The wizard adds the specified signal (or signals) to the I/O list.



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 36-18.

FPGA Board Editor

In this section...

“General Tab” on page 36-39

“Interface Tab” on page 36-41

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

The screenshot shows the 'General Tab' of the FPGA Board Editor. The dialog box is titled 'Xilinx Virtex-7 VC707 development board - Copy (S:\Xilinx_pcie\zedboard\camera\matlab\new...)' and contains the following fields and options:

- Action:** Specify your FPGA board information.
- General Tab:** Selected.
- Board Name:** My Xilinx Virtex-7 VC707 development board
- File Location:** S:\Xilinx_pcie\zedboard\camera\matlab\newboard - Copy.xml
- Device Information:**
 - Vendor: Xilinx
 - Family: Virtex7
 - Device: xc7vx485t
 - Package: ffg1761
 - Speed: -2
 - JTAG Chain Position: 1
- FPGA Input Clock:**
 - Clock Frequency: 200 MHz
 - Clock Type: Differential
 - Clock_P Pin Number: E19
 - Clock_N Pin Number: E18
 - Clock IO Standard: LVDS
- Reset (Optional):**
 - Reset Pin Number: AV40
 - Active Level: Active-High
 - Reset IO Standard: LVCMOS18

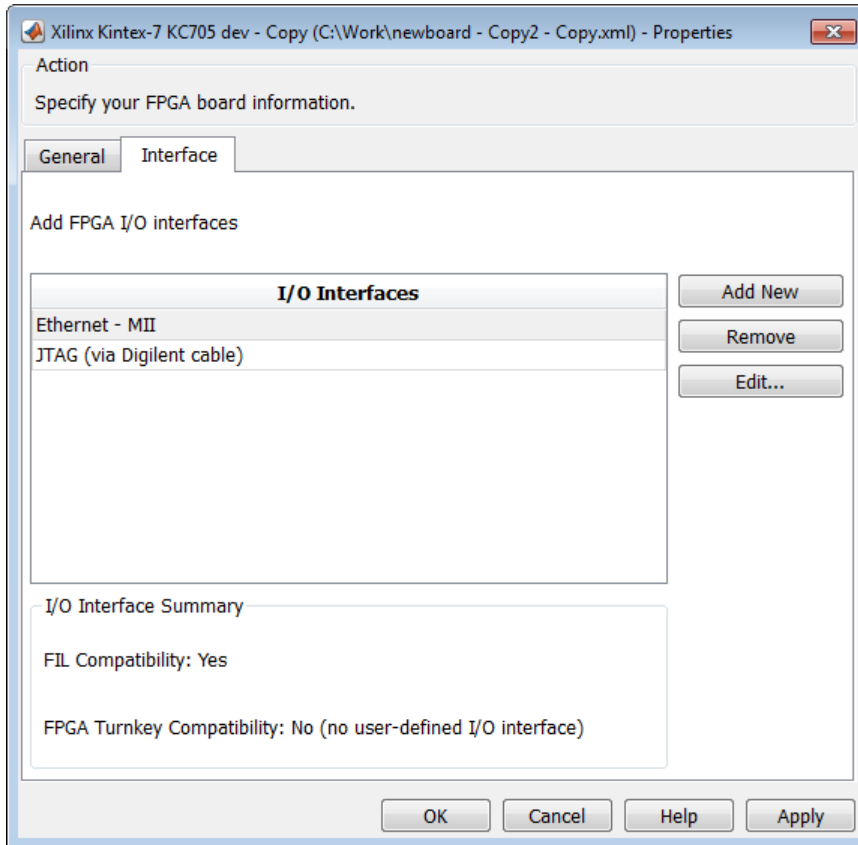
Buttons at the bottom: OK, Cancel, Help, Apply.

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.

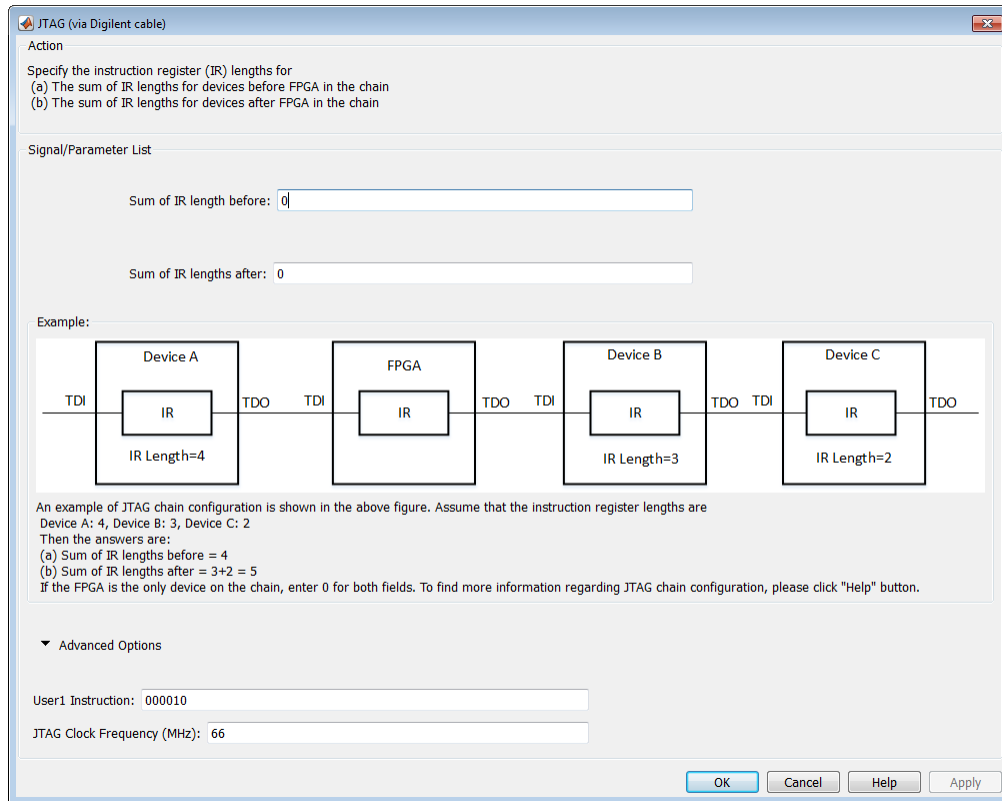
Interface Tab



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:

- Connect the FPGA board to your computer using the JTAG cable. Turn on the board.
- 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\2014.2\data\parts\xilinx\artix7\artix7\xa7a35t\cpg236`.

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 37-4
- “Set Target Overview” on page 37-5
- “Set Target Device and Synthesis Tool” on page 37-5
- “Set Target Reference Design” on page 37-6
- “Set Target Interface” on page 37-7
- “Set Target Frequency” on page 37-7
- “Set Target Interface” on page 37-8
- “Set Target Interface” on page 37-9
- “Prepare Model For HDL Code Generation Overview” on page 37-10
- “Check Global Settings” on page 37-11
- “Check Algebraic Loops” on page 37-11
- “Check Block Compatibility” on page 37-12
- “Check Sample Times” on page 37-12
- “Check FPGA-In-The-Loop Compatibility” on page 37-13
- “HDL Code Generation Overview” on page 37-13
- “Set Code Generation Options Overview” on page 37-14
- “Set Basic Options” on page 37-14
- “Set Report Options” on page 37-15
- “Set Advanced Options” on page 37-15
- “Set Optimization Options” on page 37-15
- “Set Testbench Options” on page 37-15
- “Generate RTL Code” on page 37-16
- “Generate RTL Code and Testbench” on page 37-16
- “Verify with HDL Cosimulation” on page 37-17
- “Generate RTL Code and IP Core” on page 37-17
- “FPGA Synthesis and Analysis Overview” on page 37-18
- “Create Project” on page 37-19

In this section...

“Perform Synthesis and P/R Overview” on page 37-20

“Perform Logic Synthesis” on page 37-21

“Perform Mapping” on page 37-21

“Perform Place and Route” on page 37-22

“Run Synthesis” on page 37-22

“Run Implementation” on page 37-22

“Annotate Model with Synthesis Result” on page 37-23

“Download to Target Overview” on page 37-24

“Generate Programming File” on page 37-24

“Program Target Device” on page 37-25

“Generate Simulink Real-Time Interface” on page 37-25

“Save and Restore HDL Workflow Advisor State” on page 37-25

“FPGA-In-The-Loop Implementation” on page 37-25

“Set FPGA-In-The-Loop Options” on page 37-25

“Build FPGA-In-The-Loop” on page 37-26

“Check USRP® Compatibility” on page 37-26

“Generate FPGA Implementation” on page 37-26

“Check SDR Compatibility” on page 37-27

“SDR FPGA Implementation” on page 37-27

“Set SDR Options” on page 37-27

“Build SDR” on page 37-29

“Embedded System Integration” on page 37-29

“Create Project” on page 37-29

“Generate Software Interface Model” on page 37-30

“Build FPGA Bitstream” on page 37-30

“Program Target Device” on page 37-31

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; 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.

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 desired 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 would impede code generation, and provide advice on how to fix such problems.

- **HDL Code Generation:** This category supports all HDL-related options of the Configuration Parameters dialog, 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 RTL code. An HDL Verifier license 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 31-2

Set Target Overview

The tasks in the **Set Target** folder enable you to 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, FPGA Turnkey, 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.
- **Set Target Frequency:** Select the target clock rate for the FPGA implementation of your design.

For summary 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 31-2

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 pulldown menu that lists the devices that HDL Workflow Advisor currently supports.

Description

This task displays the following options:

- **Target Workflow:** A pulldown menu that lists the possible workflows that HDL Workflow Advisor supports. Choose from:
 - Generic ASIC/FPGA
 - FPGA-in-the-loop
 - FPGA Turnkey
 - Simulink Real-Time FPGA I/O

- IP Core Generation
- Customization for the USRP(R) device
- Software Defined Radio
- **Target platform:** A pulldown menu that lists the devices the HDL Workflow Advisor currently 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 check box displays the current synthesis tool version.

Set Target Reference Design

The **Set Target Reference Design** 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 pulldown 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 41-33.
- **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:** Lists the parameters of the reference design. These 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 41-45.

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 (input or output), and data type for each port on your DUT.
- A pulldown 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 Frequency

Specify the target frequency for these workflows:

- **Generic ASIC/FPGA:** To 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. Target frequency is not supported with Microsemi Libero SoC.
- **IP Core Generation:** 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. 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 with 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 with 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.

- **FPGA Turnkey:** To generate the clock module to produce the clock signal with that frequency automatically.

See Also

“Target Frequency” on page 13-9

Set Target Interface

Select a processor-FPGA synchronization mode, and map your DUT input and output ports to I/O resources on the target device.

Description

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 before the processor continues.
- **Coprocessing - nonblocking with delay** (not supported for IP Core Generation workflow) 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 the FPGA processing time is long relative to the processor sample time, or you do not want the processor to wait for the FPGA to finish before the processor continues.

This setting is saved with the model as the `ProcessorFPGASynchronization` HDL block property for the DUT block.

The Target Platform Interface Table shows:

- The name, port type (input or output), and data type for each port on your DUT.
- A pulldown 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 40-18
- “Custom IP Core Generation” on page 40-2
- “FPGA Programming and Configuration” on page 41-52

Set Target Interface

Select a processor-FPGA synchronization mode, and map your DUT input and output ports to I/O resources on the target device. Optionally, specify a reference design.

Description

Reference design: Select the predefined embedded system integration project into which HDL Coder inserts your generated IP core.

Reference design path: Enter the path to your downloaded reference design components. This field is available only if the specified **Reference design** requires downloadable components.

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 before the processor continues.
- **Coprocessing - nonblocking with delay** (not supported for IP Core Generation workflow) 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 the FPGA processing time is long relative to the processor sample time, or you do not want the processor to wait for the FPGA to finish before the processor continues.

This setting is saved with the model as the `ProcessorFPGASynchronization` HDL block property for the DUT block.

The Target Platform Interface Table shows:

- The name, port type (input or output), and data type for each port on your DUT.

- A dropdown 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 40-18
- “Custom IP Core Generation” on page 40-2
- “FPGA Programming and Configuration” on page 41-52

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 would raise 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 the following checks:

- **Check Global Settings:** Check model parameters for compatibility with HDL code generation.
- **Check Algebraic Loops:** Check the model for algebraic loops.
- **Check Block Compatibility:** Check that blocks in the model support HDL code generation.
- **Check Sample Times:** Check the solver options, tasking mode, and rate transition diagnostic settings, given the model's sample times.
- **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 license

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(TM) 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 31-2

Check Global Settings

Check Global Settings checks model-wide parameter settings for HDL code generation compatibility.

Description

This check examines the model parameters for compatibility with HDL code generation and flags conditions that would 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, you should fix settings that are tagged as **error**.

Tip

To set reported settings to their recommended values, click the **Modify All** button. You can then run the check again and proceed to the next check.

See Also

“Model configuration checks” on page 39-14

Check Algebraic Loops

Detect algebraic loops in the model.

Description

The HDL Coder software does not support HDL code generation for models in which algebraic loop conditions exist. **Check Algebraic Loops** examines the model and fails the check if it detects an algebraic loop. Eliminate algebraic loops from your model before proceeding with further HDL Workflow Advisor checks or code generation.

See Also

- “Algebraic Loop Concepts” (Simulink)
- “Check algebraic loops” on page 38-10

Check Block Compatibility

Check the DUT for unsupported blocks.

Description

Check Block Compatibility checks blocks within the DUT for compatibility with HDL code generation. The check fails if it encounters blocks that HDL Coder does not support. The HDL Workflow Advisor reports incompatible blocks, including the full path to each block.

See Also

- “View HDL-Specific Block Documentation” on page 21-2
- “Check for unsupported blocks” on page 38-18
- “Check for unsupported blocks with Native Floating Point” on page 38-30

Check Sample Times

Check the solver, sample times, and tasking mode settings for the model.

Description

Check Sample Times checks the solver options, sample times, tasking mode, and rate transition diagnostics for HDL code generation compatibility. Solver options that the HDL Coder software requires or recommends are:

- **Type:** Fixed-step. (The coder currently supports variable-step solvers under limited conditions. See `hdlsetup` for details.)

- **Solver:** Discrete (no continuous states). Other fixed-step solvers could be selected, but this option is usually the best one for simulating discrete systems.
- **Tasking mode:** SingleTasking. The coder does not currently support models that execute in multitasking mode. Do not set **Tasking mode** to Auto.
- **Multitask rate transition** and **Single task rate transition** diagnostic options: set to Error.

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 of the **HDL Code Generation** pane of 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 Code Generation Options Overview

The tasks in the **Set Code Generation Options** folder enable you to set and validate HDL code and test bench generation parameters. Each task of the **Set Code Generation Options** folder supports options of the **HDL Code Generation** pane of the Configuration Parameters dialog box and the Model Explorer. The tasks are:

- **Set Basic Options:** Set parameters that affect overall code generation.
- **Set Report Options:** Set parameters that affect the code generation report.
- **Set Advanced Options:** Set parameters that specify detailed characteristics of the generated code, such as HDL element naming and whether certain optimizations apply.
- **Set Optimization Options:** Set parameters that specify optimizations such as resource sharing and pipelining to improve area and timing.
- **Set Testbench Options:** Set options that determine characteristics of generated test bench code.

To run the tasks in the **Set Code Generation Options** folder automatically, select the folder and click **Run All**.

Set Basic Options

Set parameters that affect overall code generation.

Description

The **Set Basic Options** task sets options that are fundamental to HDL code generation. These options include selecting the DUT and selecting the target language. The basic options are the same as those found in the top-level **HDL Code Generation** pane of the Configuration Parameters dialog box, except that the **Code generation output** group is omitted.

See Also

- “Target” on page 12-3
- “Code Generation Output” on page 16-117

Set Report Options

Set parameters that specify the sections that you want to see in the Code Generation Report.

The options are same as those found in the **HDL Code Generation > Report** pane of the Configuration Parameters dialog box and the Model Explorer.

Set Advanced Options

Set parameters that specify detailed characteristics of the generated code.

Description

The advanced options are the same as those found in the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box and the Model Explorer.

Set Optimization Options

Set parameters that specify optimizations such as resource sharing and pipelining to improve area and timing.

Description

The optimization options are the same as those found in the **HDL Code Generation > Target and Optimizations** pane of the Configuration Parameters dialog box and the Model Explorer.

Set Testbench Options

Set options that determine characteristics of generated test bench code.

Description

The test bench options are the same as those found in the **HDL Code Generation > Test Bench** pane of the Configuration Parameters dialog box and the Model Explorer.

Generate RTL Code

Generate RTL code and HDL top-level wrapper.

Description

The **Generate RTL Code** task generates RTL code and an HDL top-level wrapper for the DUT subsystem. It also generates a constraint file that contains pin mapping information and clock constraints.

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(es) selected in **Set Testbench Options**.
- **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 using cosimulation between the HDL Simulator and the Simulink test bench. This step shows only if you selected **Cosimulation model**, and specified an HDL simulator, in **Set Testbench Options**.

Generate RTL Code and IP Core

Select and initiate generation of RTL code and 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 or VHDL code, enter the file names. Enter each file name manually, separated with a semicolon (;), or by using the **Add** button. The source file language must match your target language.

This setting is saved with the model as the `IPCoreAdditionalFiles` HDL block property for the DUT block.

- **FPGA Data Capture buffer size:** The buffer size uses values that are $128 \cdot 2^n$, 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 1048576 ($=128 \cdot 2^{13}$).

This setting is saved with the model as the `IPDataCaptureBufferSize` HDL block property for the DUT block.

- **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.

- **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. In addition, make sure that you do not map any of the DUT ports to AXI4 or AXI4-Lite interfaces. You can only map the ports to External or Internal IO interfaces, or AXI4-Stream interface with TLAST mapping.

This setting is saved with the model as the `GenerateDefaultAXI4Slave` HDL block property for the DUT block.

See Also

- “Custom IP Core Generation” on page 40-2
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 40-12
- “Custom IP Core Report” on page 40-5

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.
- Launch 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 Using Simulink HDL Workflow Advisor”

Create Project

Create FPGA synthesis project for 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 completes, 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.

To learn how the synthesis objectives map to Tcl commands, see “Synthesis Objective to Tcl Command Mapping” on page 31-39.

Additional source files

Enter additional HDL source files 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 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 Using Simulink HDL Workflow Advisor”
- “Synthesis Objective to Tcl Command Mapping” on page 31-39

Perform Synthesis and P/R Overview

Launch 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 launch 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 Using Simulink HDL Workflow Advisor”

Perform Logic Synthesis

Launch supported FPGA synthesis tool and synthesize the generated HDL code.

Description

The **Perform Logic Synthesis** 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.

See Also

“HDL Code Generation and FPGA Synthesis Using Simulink HDL Workflow Advisor”

Perform Mapping

Launches supported FPGA synthesis tool and maps the synthesized logic design to the target FPGA.

Description

The **Perform Mapping** task:

- Launches 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.
- Also 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 Using Simulink HDL Workflow Advisor”

Perform Place and Route

Launches the synthesis tool in the background and runs a Place and Route process.

Description

The **Perform Place and Route** 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.
- Also emits post-routing timing information for use in critical path analysis and back annotation of your source model.
- Displays a log in the **Result** subpane.

Tips

If you select **Skip this task**, the HDL Workflow Advisor executes the workflow, but omits the **Perform Place and Route** task, marking it **Passed**. You might want to select **Skip this task** if you prefer to do place and route work manually.

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 Using Simulink HDL Workflow Advisor”

Run Synthesis

Launches Xilinx Vivado and executes the Vivado **Synthesis** step.

Enable **Skip pre-route timing analysis** if you do not want to do early timing estimation.

Run Implementation

Launches 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. Select **Skip this task** if you prefer to do place and route work manually.

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. At your option, 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.

If **Generate FPGA top level wrapper** is selected 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 3 critical paths. Select the number of paths you want to annotate.

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 Using Simulink HDL Workflow Advisor”

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 31-2

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 via 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 in to 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 31-2.

FPGA-In-The-Loop 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

Use this option for setting 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. You will need to do so 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 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, the following actions occur:

- 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 window. In this window, the FPGA design software performs synthesis, fit, place-and-route, timing analysis, and FPGA programming file generation. When the process completes, a message in the command prompts you to close the window.
- FPGA-in-the-loop builds a testbench model around the generated FIL block.

Check USRP® Compatibility

The model must have two input ports and two output ports of signed 16-bit signals.

Generate FPGA Implementation

This step initiates FPGA programming file creation. For Input Parameters, enter the path to the Ettus Research USRP® FPGA files you previously downloaded. If you have not yet downloaded these files, see the Support Package for USRP® Radio documentation.

When this step completes, see the instructions for downloading the programming file to the FPGA and running the simulation in the Support Package for USRP® Radio documentation for FPGA Targeting.

Check SDR Compatibility

The DUT must adhere to certain signal interface requirements. During Check SDR Compatibility, the following interface checks are performed (Inputs and Outputs go through the same checks).

- Must include single complex signal, two scalar signals, or single vectored signal of size 2
- Must have a bitwidth of 16
- Must be signed
- Must be single rate
- If have vectored ports must use Scalarize Vectors option
- If have multiple rates, must use Single clock
- Must use synchronous reset
- Must use active-high reset
- Must use a user overclocking factor of 1

All error checks are done for a given task run and reported in a table. This allows a single iteration to fix all errors.

SDR FPGA Implementation

The SDR FPGA integrates customer logic as generated in previous steps as well as SDR-specific code to provide data and control paths between an RF board and the host.

This step consists of the following tasks:

- Set SDR Options: Choose customization options
- Build SDR: Generate FPGA programming file for an SDR target.

Set SDR Options

Choose customization options for the completion of the SDR FPGA implementation.

SDR FPGA Component Options

- **RF board for target**

Choose one of the following:

- Epic Bitshark FMC-1Rx RevB
- Epic Bitshark FMC-1Rx RevC

- **Folder with vendor HDL source code**

Specify the folder that contains the RF interface HDL downloaded from the vendor support site. Use **Browse** to navigate to the correct folder.

- **User logic synthesis frequency**

Specify the maximum frequency at which you want to run your design. This value must be greater than the sampling frequencies for ADC and DAC as specified in the ADI FCOMMS or Epiq Bitshark™ block.

- **User logic data path**

Select either the Receiver data path or the Transmitter data path.

Radio IP Addresses

- **Board IP address**

Set the board's IP address in this field if it is not the default IP address (192.168.10.1).

- **Board MAC address**

Under most circumstances, you do not need to change the Board MAC address. However, you need to do so 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 for additional boards so that each address is unique.

Additional Source and Project Files for the HDL Design

Specify files you want included in the ISE or Vivado project. You should include only file types supported by ISE or Vivado. If an included file does not exist, the HDL Workflow Advisor cannot create the project.

- **File:** Name of file added to design (with **Add**).

- **File Type:** File type. The software will attempt to determine the file type automatically, but you may override the selection. Options are VHDL, Verilog, EDIF netlist, VQM netlist, QSF file, Constraints, and Others.
- **Add:** Add a new file to the list.
- **Remove:** Removes the currently selected file from the list.
- **Up:** Moves the currently selected file up the list.
- **Down:** Moves the currently selected file down the list.

Show full paths to source files (checkbox) triggers a full path display. Leaving this box unchecked displays only the file name.

Build SDR

The HDL Workflow Advisor creates a new Xilinx ISE or Vivado project and adds the following:

- All the necessary files from the FPGA repository
- The generated HDL files for the selected subsystem and algorithm

If no errors are found during FPGA project generation and syntax checking, the FPGA programming file generation process starts. You can view this process in an external command shell and monitor its progress. When the process is finished, a message in the command window prompts you to close the window.

Embedded System Integration

Tasks in this folder integrate your generated HDL IP core with 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 31-39.

Generate Software Interface Model

Generate a software interface model with IP core driver blocks for embedded C code generation.

After you generate the software interface model, you can generate C code from it using Embedded Coder.

Skip this task: Select this option if you want to provide your own embedded C code, or do not have an Embedded Coder license.

Operating system: Select your target operating system.

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.

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 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
```

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 bistream, device tree, and system initialization scripts to the SD card on the Zynq board, and keeps the bistream on the SD card persistently.

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 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 40-65.

HDL Model Checker

- “HDL Coder Checks in Model Advisor / HDL Model Checker Overview” on page 38-3
- “Model configuration checks overview” on page 38-5
- “Check for safe model parameters” on page 38-6
- “Check for global reset setting for Xilinx and Altera devices” on page 38-8
- “Check inline configurations setting” on page 38-9
- “Check algebraic loops” on page 38-10
- “Check for visualization settings” on page 38-11
- “Check delay balancing setting” on page 38-12
- “Check for ports and subsystems overview” on page 38-13
- “Check for invalid top level subsystem” on page 38-14
- “Check initial conditions of enabled and triggered subsystems” on page 38-15
- “Check for blocks and block settings overview” on page 38-16
- “Check for infinite and continuous sample time sources” on page 38-17
- “Check for unsupported blocks” on page 38-18
- “Check for large matrix operations” on page 38-19
- “Check for MATLAB Function block settings” on page 38-20
- “Check for Stateflow chart settings” on page 38-21
- “Check for obsolete Unit Delay Enabled/Resettable Blocks” on page 38-22
- “Check for unsupported storage class for signal objects” on page 38-23
- “Native Floating Point Checks Overview” on page 38-24
- “Check for single datatypes in the model” on page 38-25
- “Check for double datatypes in the model with Native Floating Point” on page 38-26
- “Check for Data Type Conversion blocks with incompatible settings” on page 38-27
- “Check for HDL Reciprocal block usage” on page 38-28
- “Check for Relational Operator block usage” on page 38-29
- “Check for unsupported blocks with Native Floating Point” on page 38-30

- “Check for blocks with nonzero output latency” on page 38-31
- “Check blocks with nonzero ulp error” on page 38-32
- “Industry standard checks overview” on page 38-33
- “Check VHDL file extension” on page 38-34
- “Check naming conventions” on page 38-35
- “Check top-level subsystem/port names” on page 38-36
- “Check module/entity names” on page 38-37
- “Check signal and port names” on page 38-38
- “Check package file names” on page 38-39
- “Check generics” on page 38-40
- “Check clock, reset, and enable signals” on page 38-41
- “Check architecture name” on page 38-42
- “Check entity and architecture” on page 38-43
- “Check clock settings” on page 38-44

HDL Coder Checks in Model Advisor / HDL Model Checker Overview

The **HDL Coder** checks in the Model Advisor or the HDL Model Checker 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:

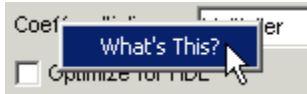
- HDL Model Checker UI, see “Getting Started with the HDL Model Checker” on page 39-2.
- Model Advisor, see “Run Model Advisor Checks for HDL Coder” on page 39-7.

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 10-92.
- **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 26-4.

To learn more about each individual check, right-click that check, and select **What's This?**.



See also “Model Checks in HDL Coder” on page 39-13.

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 safe model parameters

Check ID: `com.mathworks.HDL.ModelChecker.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 rate transition to error.
Set MultiTaskRateTransMsg to error.	Set Multitask rate transition to error.
Set SaveOutput to off.	Disable the Save output check box.
Set SaveTime to off.	Disable the Save time check box.
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 <code>set_param</code> or <code>hdlsetup</code> .	Set Default parameter behavior to Inlined. If you want to set this parameter in the Configuration Parameters dialog box, you must have Simulink Coder.
Set DataTypeOverride to off.	No dialog box prompt.

Command-Line Parameter Setting	Configuration Parameter Setting
Set ProdHWDeviceType to ASIC/FPGA->ASIC/FPGA.	Set Device vendor to ASIC/FPGA.

The check ensures that you use these settings in the Simulink Editor.

Command-Line Parameter Setting	Simulink Editor Setting
Set ShowLineDimensions to on.	On the Display > Signals & Ports menu, set Signal Dimensions .
Set SignalLoggingSaveFormat to Dataset.	No dialog box prompt.

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 Simulink Model for HDL Code Generation”

Check for global reset setting for Xilinx and Altera devices

Check ID: `com.mathworks.HDL.ModelChecker.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” on page 16-10

Check inline configurations setting

Check ID: `com.mathworks.HDL.ModelChecker.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.

See Also

`InlineConfigurations`

Check algebraic loops

Check ID: `com.mathworks.HDL.ModelChecker.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” (Simulink)

Check for visualization settings

Check ID: `com.mathworks.HDL.ModelChecker.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 warning if your Simulink model violates either or both of these settings.

Results and Recommended Actions

To fix this warning, 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” (Fixed-Point Designer)
- “View Sample Time Information” (Simulink)

Check delay balancing setting

Check ID: `com.mathworks.HDL.ModelChecker.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 24-32
- “Balance delays” on page 14-3

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 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 or a For Each Subsystem block, this check displays a warning and provides a link to the Subsystem.

Results and Recommended Actions

To fix this warning, place this Subsystem inside another Subsystem, and then use that Subsystem as the DUT. You can then rerun the check.

Check initial conditions of enabled and triggered subsystems

Check ID: com.mathworks.HDL.ModelChecker.runEnTrigInitConChecks

Check for initial condition of enabled and triggered subsystems.

Description

This check verifies that any Enabled Subsystem blocks or Triggered Subsystem blocks in your Simulink model have a zero initial condition. If you have output ports with a nonzero initial condition, running this check displays a warning and provides links to the output ports.

Results and Recommended Actions

To fix this warning, click **Modify Settings** to set the initial condition of the output ports of the Enabled and Triggered Subsystem blocks to zero. You can then rerun the check.

See Also

- Triggered Subsystem
- Enabled Subsystem

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 source blocks with infinite or continuous sample time.

Description

By default, the sample time of a Constant block is `inf`. When you connect a Constant block with sample time of `inf` to other blocks in your design, it hinders speed and area optimizations. Optimizations such as retiming, sharing, and streaming use the clock rate information to improve the speed and area of your design. For more details, see the “HDL Code Generation” (Simulink) section of the Constant 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?” (Simulink)
- “Usage of Rate Change and Constant Blocks” on page 20-107

Check for unsupported blocks

Check ID: `com.mathworks.HDL.ModelChecker.runBlockSupportChecks`

Check for unsupported blocks for HDL code generation.

Description

This check displays a warning if your model uses blocks that are not supported for HDL code generation.

Results and Recommended Actions

To fix this warning, 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 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 10-2

Check for MATLAB Function block settings

Check ID: `com.mathworks.HDL.ModelChecker.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 29-35

Check for Stateflow chart settings

Check ID:

`com.mathworks.HDL.ModelChecker.runStateflowChartSettingsChecks`

Check HDL compatible settings for Stateflow Chart blocks.

Description

This check displays a warning if your model uses Stateflow Chart blocks that have settings incompatible for HDL code generation. The settings include checking whether the Stateflow Chart has **Execute (enter) Chart At Initialization** set, whether the **State Machine Type** is Moore, and so on.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and 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 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 27-63.

Check for unsupported storage class for signal objects

Check ID:`com.mathworks.HDL.ModelChecker.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)

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 10-92.

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 for single data types in the model.

Description

This check detects whether your Simulinkmodel uses single data types and displays a warning if the **Floating Point IP Library** is not set to **Native Floating Point**.

Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator sets **Native Floating Point** as the **Floating Point IP Library**. You can then rerun the check.

See Also

“Getting Started with HDL Coder Native Floating-Point Support” on page 10-65

Check for double datatypes in the model with Native Floating Point

Check ID: com.mathworks.HDL.ModelChecker.runDoubleDatatypeChecks

Check for double data types in the model.

Description

This check displays a warning when your Simulinkmodel uses double data types with **Floating Point IP Library** set to Native Floating Point. The check passes when your model uses double data types but does not have the **Floating Point IP Library** set to Native Floating Point.

Results and Recommended Actions

To fix this warning, update your design by converting double data types to single and then set the **Floating Point IP Library** to Native Floating Point. You can then rerun the check.

Note The check passes when your model uses double data types but does not have the **Floating Point IP Library** set to Native Floating Point. However, double data types in your model can generate reals in the HDL code which is not synthesizable. To generate synthesizable HDL code, convert double data types to single and set the **Floating Point IP Library** to Native Floating Point.

See Also

“Getting Started with HDL Coder Native Floating-Point Support” on page 10-65

Check for Data Type Conversion blocks with incompatible settings

Check ID: `com.mathworks.HDL.ModelChecker.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 10-65

Check for HDL Reciprocal block usage

Check ID: `com.mathworks.HDL.ModelChecker.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 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 for unsupported blocks with Native Floating Point.

Description

This check displays a warning 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 warning, make sure that your design uses blocks that are supported for HDL code generation with `Native Floating Point`.

See Also

“Simulink Blocks Supported with Native Floating-Point” on page 10-106

Check for blocks with nonzero output latency

Check ID: `com.mathworks.HDL.ModelChecker.runNFPLatencyChecks`

Check for blocks that have nonzero output latency with Native Floating Point.

Description

This check detects blocks in your Simulink model that have a nonzero output latency with Native Floating Point. When you run the check, the **Result** subpane displays hyperlinks to the blocks that have nonzero output latency, and the minimum and maximum latency values of the blocks.

Results and Recommended Actions

Native floating point operators are computation-intensive and can have nonzero output latency. When you generate code, HDL Coder figures out this latency.

For blocks with nonzero latency that are reported by this check, consider the effect of this latency in the validation model. In the generated model and validation model, you see the additional delays that the code generator added to account for the latency. If your design contains parallel paths, delay balancing adds matching delays to balance those paths. You can use this information to customize your algorithm for latency considerations.

See Also

“Latency Considerations with Native Floating Point” on page 10-83

Check blocks with nonzero ulp error

Check ID: `com.mathworks.HDL.ModelChecker.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 with Native Floating-Point” on page 10-69

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 26-4.

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 VHDL file extension

Check ID: `com.mathworks.HDL.ModelChecker.runFileExtensionChecks`

Check file extensions of VHDL files containing entities.

Description

This check detects whether the file extension is `.vhd` when you generate code with VHDL 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 to `.vhd`.

See Also

Rule 1.A.A.1 of “Basic Coding Practices” on page 26-10.

Check naming conventions

Check ID: `com.mathworks.HDL.ModelChecker.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 26-10.

Check top-level subsystem/port names

Check ID: `com.mathworks.HDL.ModelChecker.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 26-10.

Check module/entity names

Check ID: `com.mathworks.HDL.ModelChecker.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 26-10.

Check signal and port names

Check ID: `com.mathworks.HDL.ModelChecker.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 26-10.

Check package file names

Check ID: `com.mathworks.HDL.ModelChecker.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 26-10.

Check generics

Check ID: `com.mathworks.HDL.ModelChecker.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 26-10.

Check clock, reset, and enable signals

Check ID: `com.mathworks.HDL.ModelChecker.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_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 26-10.

Check architecture name

Check ID: `com.mathworks.HDL.ModelChecker.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 26-10.

Check entity and architecture

Check ID:

`com.mathworks.HDL.ModelChecker.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 26-10.

Check clock settings

Check ID: `com.mathworks.HDL.ModelChecker.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 26-10.

Using the HDL Model Checker

- “Getting Started with the HDL Model Checker” on page 39-2
- “Run Model Advisor Checks for HDL Coder” on page 39-7
- “Model Checks in HDL Coder” on page 39-13

Getting Started with the HDL Model Checker

In this section...

“Open the HDL Model Checker” on page 39-2

“Run Checks In the HDL Model Checker” on page 39-3

“Fix HDL Model Checker Warnings or Failures” on page 39-4

“View and Save HDL Model Checker Reports” on page 39-5

The HDL Model Checker 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 Model Checker produces a report that lists suboptimal conditions or settings, and then proposes better model configuration settings.

The HDL Model Checker has these caveats:

- If you reference one model in another by using a Model block, the HDL Model Checker 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 Model Checker for the referenced model, and then run the checks.
- If you run the checks on masked library blocks in your Simulink model, the Model Checker 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 Model Checker

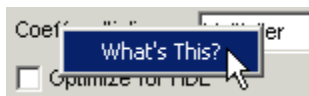
To open the HDL Model Checker:

- 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 > Check Model Compatibility**.

- At the command line, enter `hdlmodelchecker('system')`. *system* is a handle or name of the model or subsystem that you want to check. For more information, see `hdlmodelchecker`.

In the HDL Model Checker, 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 Model Checker 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?**.



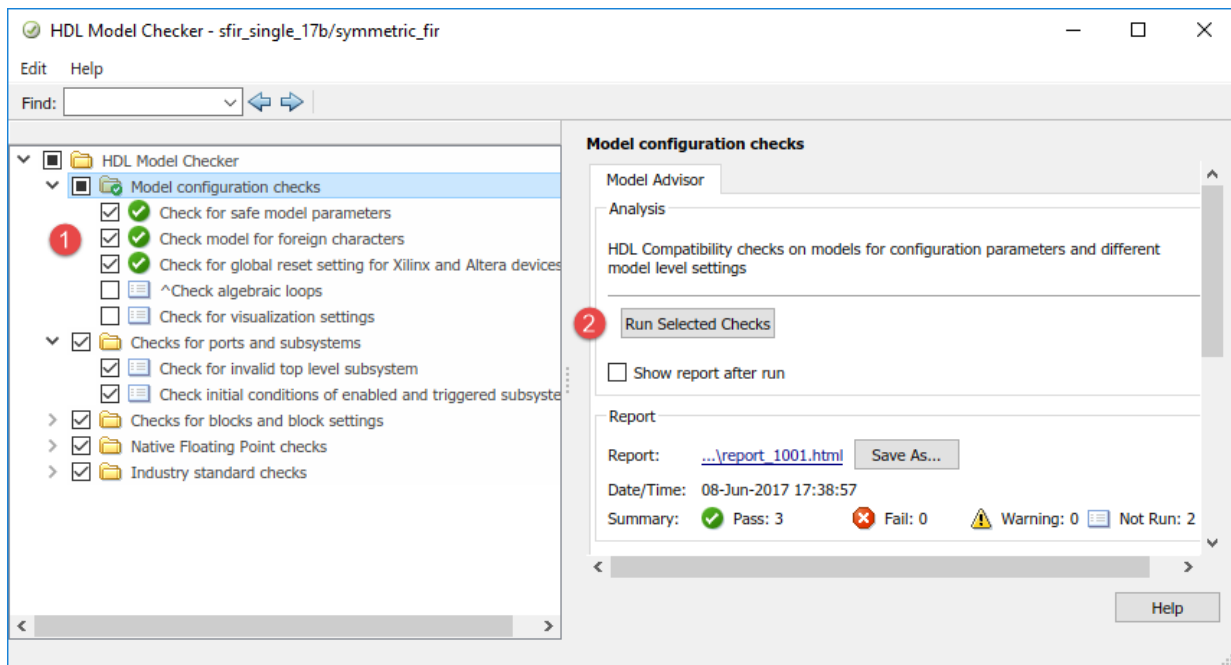
Run Checks In the HDL Model Checker

In the HDL Model Checker 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 Model Checker window, you can run a group of checks within a folder.

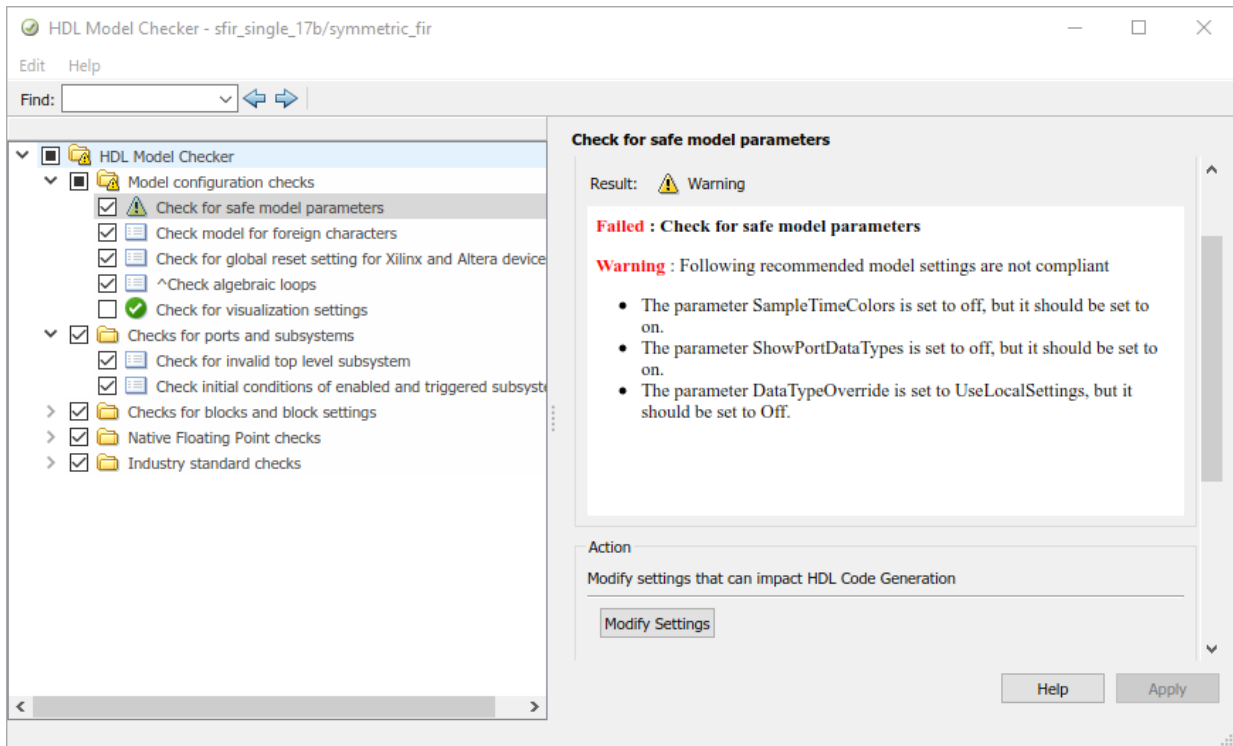
- 1 Select the checks that you want to run.
- 2 Select the folder that contains these checks and then click **Run Selected Checks**.

This example shows how to run selected checks in the **Model configuration checks** folder.



Fix HDL Model Checker Warnings or Failures

In the HDL Model Checker, 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 Model Checker recommended settings. This example displays the incorrect model settings that caused the **Check for safe model parameters** to fail.



To apply the correct 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 Model Checker Reports

When you run checks in the HDL Model Checker, HDL Coder generates an HTML report of the check results. Each folder in the HDL Model Checker 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 Model Checker, 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.

Filter checks

Passed

Failed

Warning

Not Run

Keywords

Navigation

[Model configuration checks](#)

View

[Scroll to top](#)

[Hide check details](#)

Model Advisor Report - sfir_single_17b.slx

Simulink version: 9.0 Model version: 1.82

System: sfir_single_17b/symmetric_fir Current run: 08-Jun-2017 17:38:57

Treat as Referenced Model: off

Run Summary

Pass	Fail	Warning	Not Run	Total
3	0	0	2	5

Model configuration checks

Check for safe model parameters

Passed : Check for safe model parameters

Check model for foreign characters

Check that the characters in the model can be represented in the current encoding.

Passed

All the characters in the model can be represented in the current encoding.

As you run the checks, the HDL Model Checker 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

- “Model Checks in HDL Coder” on page 39-13

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 appropriate. HDL Coder integrates the checks in the HDL Model Checker with the Model Advisor.

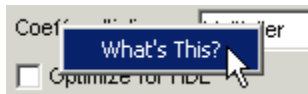
Open the Model Advisor Checks

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?**.



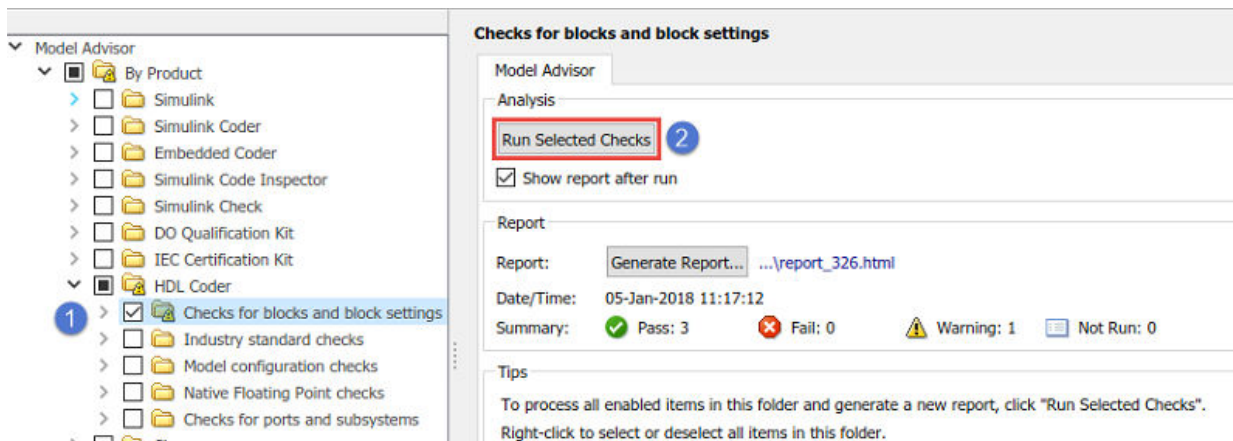
Run Checks in the Model Advisor


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 Selected Checks**


For example, to run all the checks in the **Checks for blocks and block settings** folder, select the folder, and then click **Run Selected Checks**.




You can also click the  button to run the selected checks in the Model Advisor.

Run Checks In Background

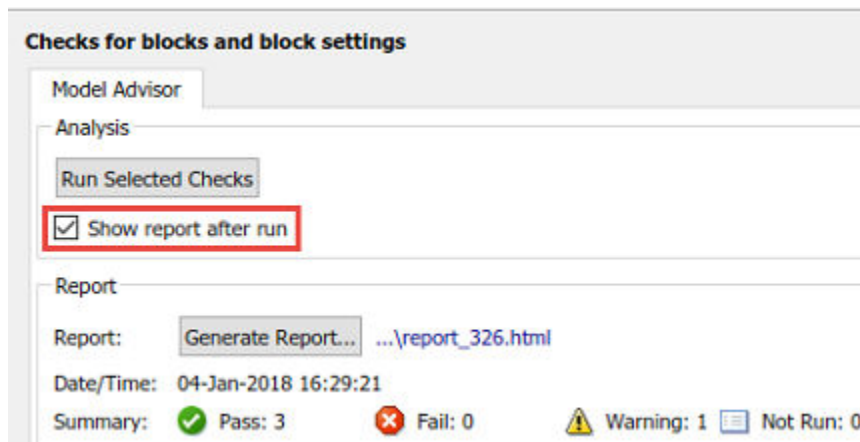
If you have Parallel Computing Toolbox™, you can run the model checks in the background. You can continue working on the model during analysis. The analysis does not reflect changes that you make to your model while Model Advisor is running in the background.

To run the Model Advisor checks in background, before you select and run the checks, click the **Run checks in background** toggle, .

When you run the checks, the Model Advisor starts an analysis on a parallel processor. To stop running checks in the background, in the Model Advisor Window, click **Stop background run**, .

Display Check Results in the Model Advisor Report

To display an HTML report of the check results, before you run the checks, select **Show report after run**. You can specify this setting to generate a report for all the checks in the **HDL Coder** folder, or for all checks within a subfolder, such as the **Checks for blocks and block settings**.



If you did not select **Show report after run**, you can generate a report after you run the checks by selecting **Generate Report**. Specify the **Directory**, **Filename**, and **Format** of the HTML report that want to generate.

This report shows typical results for a run of the **Checks for blocks and block settings** folder.

The screenshot shows the Model Advisor Report interface. On the left, there is a 'Filter checks' panel with four options: 'Passed' (checkbox), 'Failed' (checkbox), 'Warning' (checkbox, highlighted with a red box), and 'Not Run' (checkbox). Below this is a 'Keywords' search box. Further down are 'Navigation' and 'View' sections with links like 'Scroll to top' and 'Hide check details'. The main report area is titled 'Model Advisor Report - sfir_single.slx' and includes the following information:

- Simulink version: 9.1
- Model version: 1.94
- System: sfir_single/symmetric_fir
- Current run: 05-Jan-2018 11:17:12
- Treat as Referenced
- Model: off


The 'Run Summary' section displays a table of results:

Pass	Fail	Warning	Not Run	Total
3	0	1	0	4

Below the summary, there is a section titled 'Checks for blocks and block settings' with a sub-section 'Check for MATLAB Function block settings'.

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.

You can also use the Model Advisor highlighting capability to color-highlight Simulink blocks and Stateflow charts in your model, which indicates the analysis results. To

highlight blocks, in the Model Advisor window, select **Highlighting > Enable Highlighting**.

To fix warnings or failures, in the **Result** subpane, review the recommended actions to make changes to your model. When you fix a warning or failure, to verify that the check passes, rerun the check.

Some checks have an **Action** subpane. This example displays the incorrect MATLAB Function block settings that caused the check to display a warning.

The screenshot shows the Model Advisor interface for a check titled "Check for MATLAB Function block settings". The interface is divided into three main sections: Analysis, Result, and Action.

- Analysis:** Contains the check name "Check for MATLAB Function block settings" and a "Run This Check" button.
- Result:** Shows a "Warning" icon and text. It includes two warning messages:
 - Warn :** Check for MATLAB Function block settings
 - Warning :** Following blocks have 'Saturate on integer overflow' set to 'on'
 - [sfir_single/symmetric_fir/MATLAB Function](#)
 - Warning :** The following blocks should have parameter 'MATLAB Function fimath' set to 'Other:User Defined'. The fimath string should have 'RoundMode' set to 'Floor', 'OverflowMode' set to wrap, and 'ProductMode' and 'SumMode' set to 'FullPrecision'.
 - [sfir_single/symmetric_fir/MATLAB Function](#)
- Action:** Contains the instruction "Modify MATLAB Function block settings to be compatible for HDL code generation" and a "Modify Settings" button.

When you select **Modify Settings**, the **Result** subpane 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.

Save and Restore Model Advisor State

By default, the Simulink software saves the state of the most recent Model Advisor session. The next time that you activate the Model Advisor, it returns to that state. You can also save the current settings of the Model Advisor to a named restore point. A *restore point* is a snapshot in time of the model, base workspace, and Model Advisor. Later, you can restore the same settings by loading the restore point data into the Model Advisor.

You can use this data restore point to revert changes to your model in response to recommendations from the Model Advisor. For example, you can save a model and restore point to undo your changes if the Model Advisor reports a warning after running a certain check. You can also restore the default configuration of the Model Advisor. In the Model Advisor window, select **Settings > Restore Default Configuration**.

To save the Model Advisor state, in the Model 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 Model Advisor state, in the Model Advisor Window, select **File > Load Restore Point**. Select the restore point and click **Load**. When you load a restore point, the Model Advisor warns that the restoration overwrites the current settings.

See Also

More About

- “Model Checks in HDL Coder” on page 39-13
- “Run Model Advisor Checks” (Simulink)

Model Checks in HDL Coder

In this section...

“Model configuration checks” on page 39-14

“Checks for ports and subsystems” on page 39-15

“Checks for blocks and block settings” on page 39-15

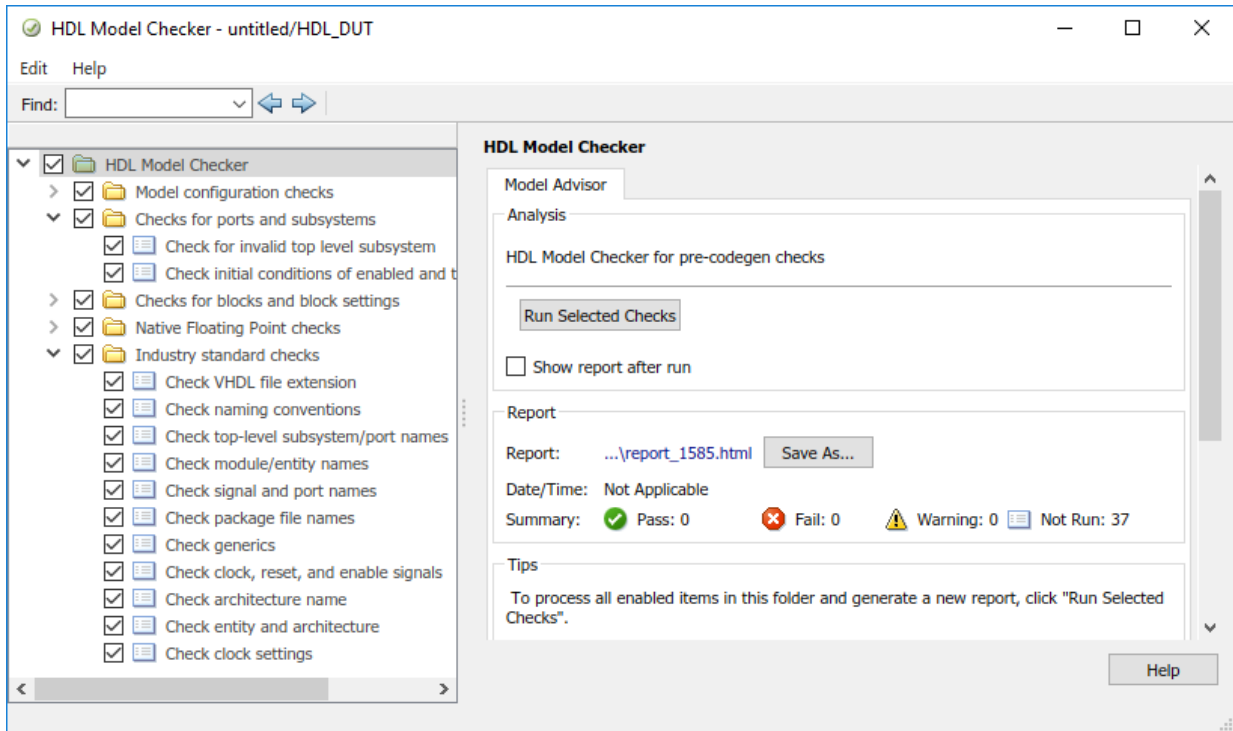
“Native Floating Point checks” on page 39-17

“industry standard checks” on page 39-17

The HDL Model Checker and the Model Advisor checks in HDL Coder verify and update your Simulink model or subsystem for compatibility with HDL code generation. The Model Checker 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 generator displays the result as a pass 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 safe model parameters” on page 38-6	Check for model parameters set up for HDL code generation.
“Check model for foreign characters” (Simulink)	Search the model for unresolved library links, where the specified library block cannot be found.

Check Name	Description
“Check for global reset setting for Xilinx and Altera devices” on page 38-8	Check asynchronous reset setting for Altera devices and synchronous reset setting for Xilinx devices.
“Check inline configurations setting” on page 38-9	Check whether you have <code>InlineConfigurations</code> enabled.
“Check algebraic loops” on page 38-10	Check model for algebraic loops.
“Check for visualization settings” on page 38-11	Check model for display settings: port data types and sample time color coding.
“Check delay balancing setting” on page 38-12	Check <code>Balance Delays</code> is enabled.

Note If you use the Model Advisor, you see the “Check model for foreign characters” (Simulink) in the **Simulink** folder.

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 38-14	Check for subsystems that cannot be at the top level for HDL code generation.
“Check initial conditions of enabled and triggered subsystems” on page 38-15	Check for initial condition of enabled and triggered subsystems.

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 38-17	Check source blocks with continuous sample time.
“Check for unsupported blocks” on page 38-18	Check for unsupported blocks for HDL code generation.
“Check for large matrix operations” on page 38-19	Check for large matrix operations.
“Identify unconnected lines, input ports, and output ports” (Simulink)	Check for unconnected lines or ports.
“Identify disabled library links” (Simulink)	Search model for disabled library links.
“Identify unresolved library links” (Simulink)	Search the model for unresolved library links, where the specified library block cannot be found.
“Check for MATLAB Function block settings” on page 38-20	Check HDL compatible settings for MATLAB Function blocks.
“Check for Stateflow chart settings” on page 38-21	Check HDL compatible settings for Stateflow Chart blocks.
“Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition” (Simulink)	Identify Delay, Unit Delay, or Zero-Order Hold blocks that are used for rate transition. Replace these blocks with actual Rate Transition blocks.
“Check for unsupported storage class for signal objects” on page 38-23	Check whether signal object storage class is 'ExportedGlobal' or 'ImportedExtern' or 'ImportedExternPointer'

Note If you use the Model Advisor, you see the “Identify unconnected lines, input ports, and output ports” (Simulink), “Identify disabled library links” (Simulink), “Identify unresolved library links” (Simulink), and “Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition” (Simulink) 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 10-92.

Check Name	Description
“Check for single datatypes in the model” on page 38-25	Check for single data types in the model.
“Check for double datatypes in the model with Native Floating Point” on page 38-26	Check for double data types in the model.
“Check for Data Type Conversion blocks with incompatible settings” on page 38-27	Check conversion mode of Data Type Conversion blocks.
“Check for HDL Reciprocal block usage” on page 38-28	Check HDL Reciprocal blocks are not using floating point types.
“Check for Relational Operator block usage” on page 38-29	Check Relational Operator blocks which use floating point types have boolean outputs.
“Check for unsupported blocks with Native Floating Point” on page 38-30	Check for unsupported blocks with native floating-point.
“Check for blocks with nonzero output latency” on page 38-31	Check for blocks that have nonzero output latency with native floating-point.
“Check blocks with nonzero ulp error” on page 38-32	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 VHDL file extension” on page 38-34	Check file extensions of VHDL files containing entities.
“Check naming conventions” on page 38-35	Check standard keywords used by EDA tools.
“Check top-level subsystem/port names” on page 38-36	Check top-level module/entity and port names.
“Check module/entity names” on page 38-37	Check module/entity names.
“Check signal and port names” on page 38-38	Check signal and port name lengths.
“Check package file names” on page 38-39	Check file name containing packages.
“Check generics” on page 38-40	Check generics at top-level subsystem.
“Check clock, reset, and enable signals” on page 38-41	Check naming convention for clock, reset, and enable signals.
“Check architecture name” on page 38-42	Check VHDL architecture name in the generated HDL code.
“Check entity and architecture” on page 38-43	Check whether the VHDL entity and architecture are described in the same file.
“Check clock settings” on page 38-44	Check constraints on clock signals.

For more information, see:

- “HDL Coding Standards” on page 26-4
- “Basic Coding Practices” on page 26-10
- “RTL Description Techniques” on page 26-25
- “RTL Design Methodology Guidelines” on page 26-64

See Also

hdlmodelchecker

More About

- “Getting Started with the HDL Model Checker” on page 39-2
- “Run Model Advisor Checks for HDL Coder” on page 39-7

Hardware-Software Codesign

Hardware-Software Co-Design Basics

- “Custom IP Core Generation” on page 40-2
- “Custom IP Core Report” on page 40-5
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 40-12
- “Processor and FPGA Synchronization” on page 40-18
- “Synchronization of Global Reset Signal to IP Core Clock Domain” on page 40-21
- “IP Caching for Faster Reference Design Synthesis” on page 40-26
- “Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows” on page 40-34
- “Define and Add IP Repository to Custom Reference Design” on page 40-47
- “Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface” on page 40-52
- “Meet Timing Requirements Using Enable-Based Multicycle Path Constraints” on page 40-58
- “Program Target FPGA Boards or SoC Devices” on page 40-65

Custom IP Core Generation

In this section...

- “Custom IP Core Architectures” on page 40-2
- “Target Platform Interfaces” on page 40-3
- “Processor/FPGA Synchronization” on page 40-3
- “Custom IP Core Generated Files” on page 40-4

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 40-12
- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-56

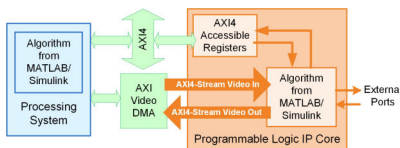
Custom IP Core Architectures

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 40-12.

The *Algorithm from MATLAB/Simulink* block represents your DUT. HDL Coder generates the rest of the IP core based on your target platform interface settings and processor/FPGA synchronization mode.

Target Platform Interfaces

You can map each port in your DUT to one of the following target platform interfaces in the IP core:

- **AXI4-Lite:** Use this slave interface to access control registers or for lightweight data transfer. HDL Coder generates memory-mapped registers and allocates address offsets for the ports you map to this interface.
- **AXI4:** Use this slave interface to connect to components that support burst data transmission. HDL Coder generates memory-mapped registers and allocates address offsets for the ports you map to this interface.
- **AXI4-Stream Video:** Use this interface to send or receive a 32-bit scalar video data stream.
- **External ports:** Use external ports to connect to FPGA external IO pins, or to other IP cores with external ports.

To learn more about the AXI4, AXI4-Lite and AXI4-Stream Video protocols, refer to your target hardware documentation.

Processor/FPGA Synchronization

HDL Coder generates synchronization logic in the IP core based on the processor/FPGA synchronization mode you choose.

When generating a custom IP core, the following processor/FPGA synchronization options are available:

- **Free running** (default)
- **Coprocessing – blocking**

To learn more about the processor/FPGA synchronization modes, see “Processor and FPGA Synchronization” on page 40-18.

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 the following generated files:

- 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.

See Also

Related Examples

- “IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705”
- “IP Core Generation Workflow Without an Embedded ARM Processor: Arrow DECA MAX 10 FPGA Evaluation Kit”

More About

- “Multirate IP Core Generation” on page 41-27

Custom IP Core Report

In this section...

“Summary” on page 40-5
 “Target Interface Configuration” on page 40-6
 “Register Address Mapping” on page 40-6
 “IP Core User Guide” on page 40-7
 “IP Core File List” on page 40-10

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	Output	uint8 (3)	AXI4	x"160"
Out2	Output	uint16 (100)	AXI4	x"A00"
Out3	Output	single	AXI4	x"11C"
Out4	Output	uint32	AXI4	x"120"

To learn more about processor/FPGA synchronization modes, see “Processor and FPGA Synchronization” on page 40-18.

To learn more about target platform interfaces, see “Custom IP Core Generation” on page 40-2.

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 Outputport Out3
Out4_Data	0x120	data register for Outputport Out4
Out1_Data	0x160	data register for Outputport 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 Outputport 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.

IP Core User Guide

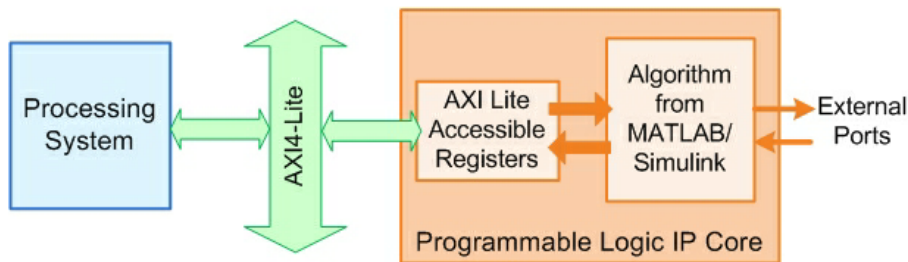
The IP Core User Guide section gives a high-level overview of the system architecture, describes the processor and FPGA synchronization mode, and gives instructions for integrating the IP core in your embedded system integration environment.

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.

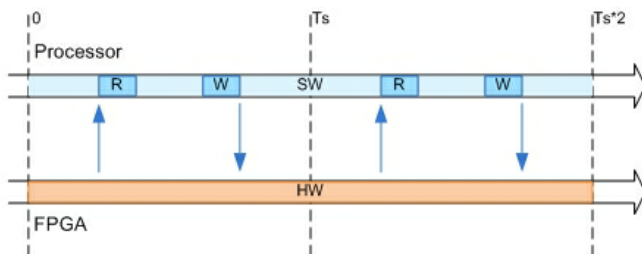


This IP core also supports the **External Port** interface. To connect the external ports to the FPGA external IO pins, add FPGA pin assignment constraints in the Xilinx EDK environment.

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.

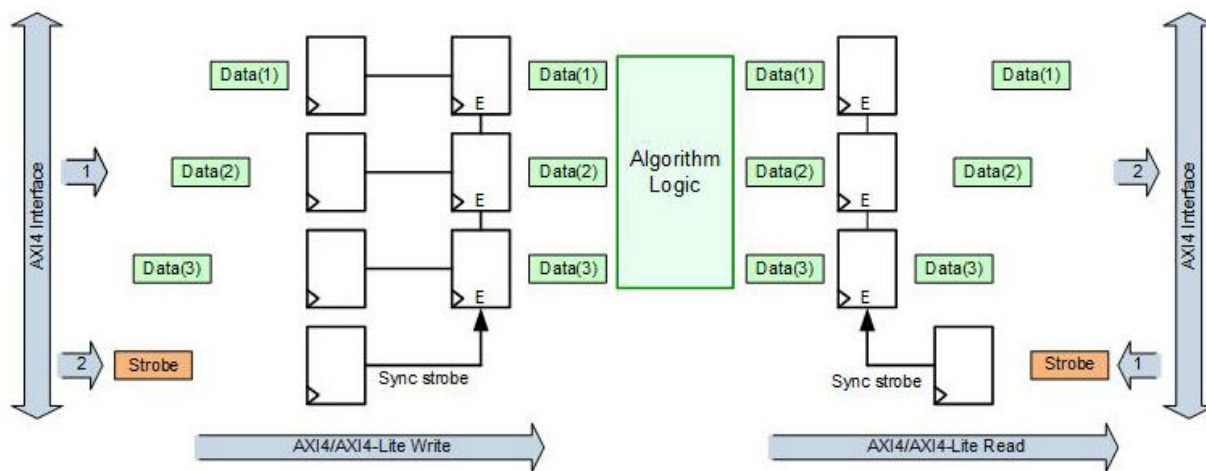


If you use vector data signals at the DUT interface, the IP core report displays this section that shows how the code generator synchronizes vector data across the AXI4 interface.

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\vhd\led_counter_pkg.vhd](#)

[hdl\vhd\led_counter.vhd](#)

[hdl\vhd\hdlcoder_led_blinking_led_counter_pcore_dut.vhd](#)

[hdl\vhd\hdlcoder_led_blinking_led_counter_pcore_axi_lite_module.vhd](#)

[hdl\vhd\hdlcoder_led_blinking_led_counter_pcore_addr_decoder.vhd](#)

[hdl\vhd\hdlcoder_led_blinking_led_counter_pcore_axi_lite.vhd](#)

[hdl\vhd\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 40-2
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 41-2
- “Multirate IP Core Generation” on page 41-27

Generate Board-Independent HDL IP Core from Simulink Model

In this section...

“Generate Board-Independent IP Core” on page 40-12

“IP Core without AXI4 Slave Interfaces” on page 40-15

“Requirements and Limitations for IP Core Generation” on page 40-16

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 by using the `hdlcoder.ReferenceDesign` class. To learn more, see:

- “Define Custom Board and Reference Design for Zynq Workflow”
- “Define Custom Board and Reference Design for Intel SoC Workflow”

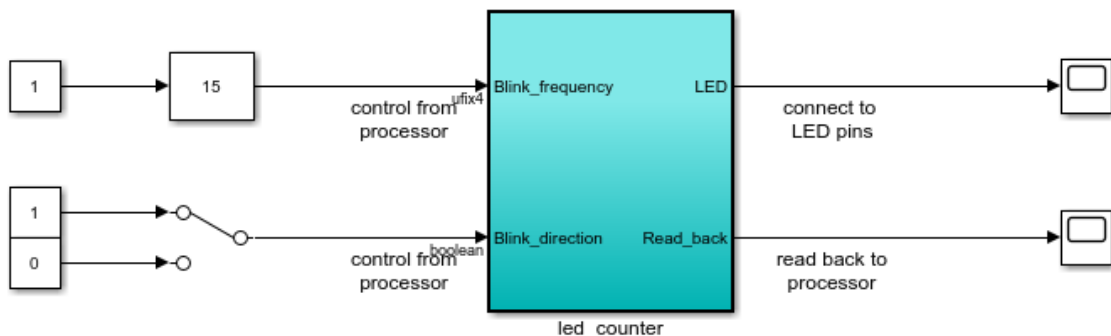
Generate Board-Independent IP Core

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 your DUT in your 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.

- Set the path to the installed synthesis tool for the target device by using the `hdlsetuptoolpath` function. For example, if your synthesis tool is Xilinx Vivado, enter this command:

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
    'C:\Xilinx\Vivado\2018.2\bin\vivado.bat');
```

See “Supported Third-Party Tools and Hardware” for latest supported version of the synthesis tool.

- 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, then click **Apply**. You can map each DUT port to one of AXI4-Lite, AXI4, AXI4-Stream, AXI4-Stream Video, or External Port interfaces. To learn more about these interfaces, see “Target Platform Interfaces” on page 40-3.

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	Output	uint8	External Port ▼	
Read_back	Output	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 40-52.

- 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. Click **Run This Task**.

Note If you mapped any of the DUT ports to AXI4 slave interfaces in the **Set Target Interface** task, even if you clear this check box, the code generator ignores this setting and maps the ports to AXI4 slave interfaces.

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**.

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.

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 make sure that you 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 of such reference designs, see:

- “IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705”
- “IP Core Generation Workflow Without an Embedded ARM Processor: Arrow DECA MAX 10 FPGA Evaluation Kit”

When you generate an HDL IP core without AXI4 slave interfaces, certain restrictions apply. See “IP Core without AXI4 Slave Interface Restrictions” on page 40-17.

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. Coprocessing – blocking mode is not supported.

IP Core without AXI4 Slave Interface Restrictions

- You can only map the ports to External or 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 40-2
- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-56
- “Board and Reference Design Registration System” on page 41-33

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.
- Use the **Simulink Real-Time FPGA I/O** workflow.

The following synchronization modes are available:

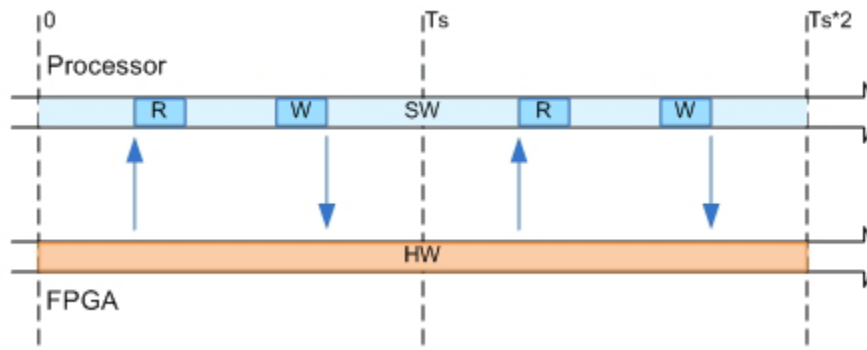
- Free running (default)
- Coprocessing – blocking
- Coprocessing – nonblocking with delay (available only for the **Simulink Real-Time FPGA I/O** workflow)

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.

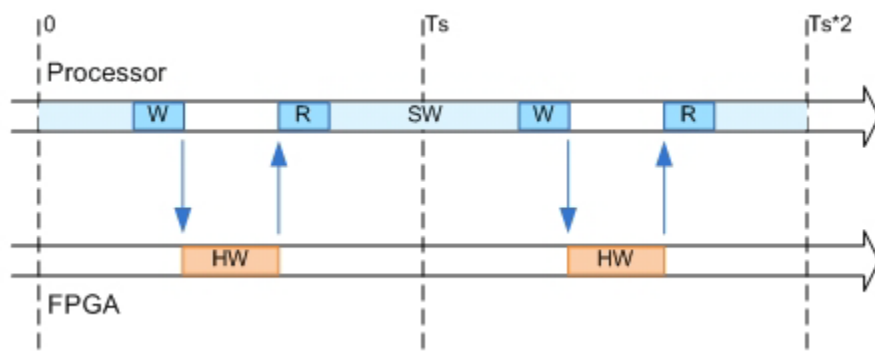


Coprocessing - Blocking Mode

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.



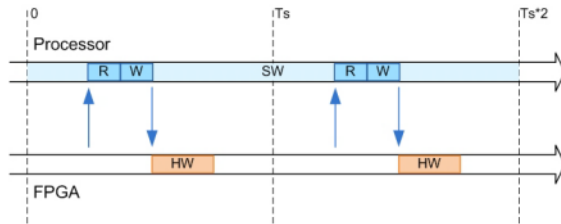
The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor writes to the FPGA, then stops and waits for an indication that the FPGA has finished processing before continuing to run. Each time the FPGA runs, it executes the logic generated for one DUT subsystem sample time.

Coprocessing - Nonblocking With Delay Mode

In delayed nonblocking coprocessor mode, HDL Coder automatically generates synchronization logic for the FPGA so that the processor and FPGA run in tandem. This mode is only available to Speedgoat IO modules that use Xilinx ISE with the **Simulink Real-Time FPGA I/O** workflow.

Select **Coprocessing - nonblocking with delay** as the **Processor/FPGA synchronization mode** when the FPGA processing time is long relative to the processor sample time, or you do not want the processor to wait for the FPGA to finish before the processor continues to run.

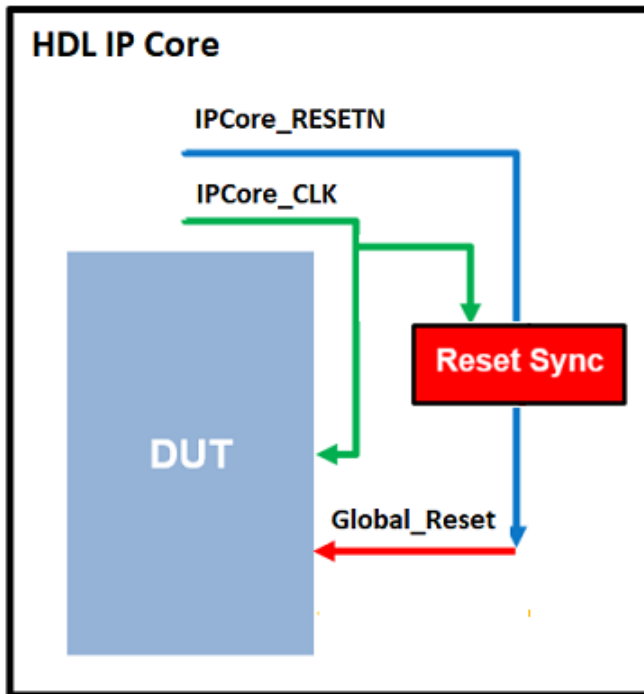
The following diagram shows how the processor and FPGA run in delayed nonblocking coprocessor mode.



The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor reads FPGA data from the previous sample time, then writes to the FPGA and continues to run without waiting for the FPGA to finish. Each time the FPGA runs, it executes the logic generated for one DUT subsystem sample time.

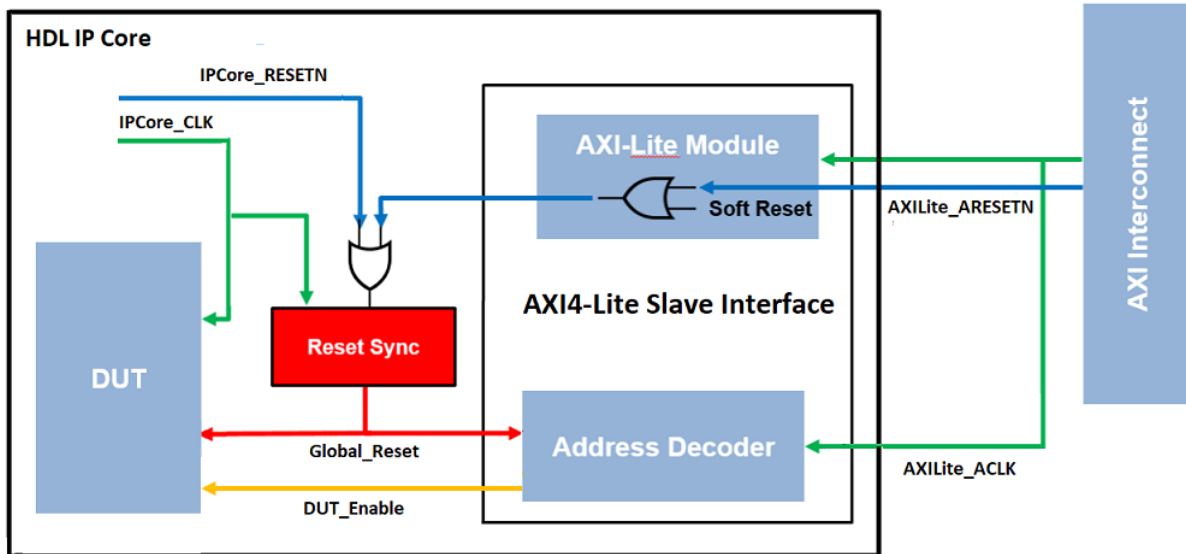
Synchronization of Global Reset Signal to IP Core Clock Domain

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 40-12.



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 for the ARM processor core. The global reset signal in

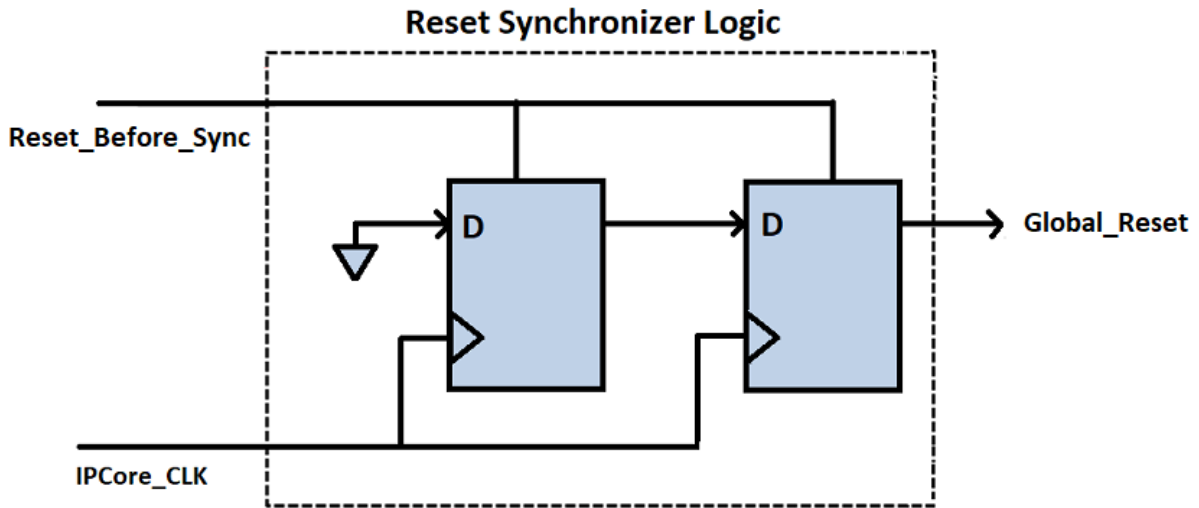
this case drives the HDL IP core for the DUT and the Address Decoder logic in the AXI4 slave wrapper.



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 40-2
- “Custom IP Core Report” on page 40-5
- “Processor and FPGA Synchronization” on page 40-18

IP Caching for Faster Reference Design Synthesis

In this section...

- “Requirements for Using IP Caching” on page 40-26
- “What Is an IP Cache?” on page 40-26
- “How IP Caching Works” on page 40-27
- “Enable IP Caching” on page 40-28
- “IP Caching in HDL Coder Reference Designs” on page 40-29
- “IP Caching in Custom Reference Designs” on page 40-32

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.

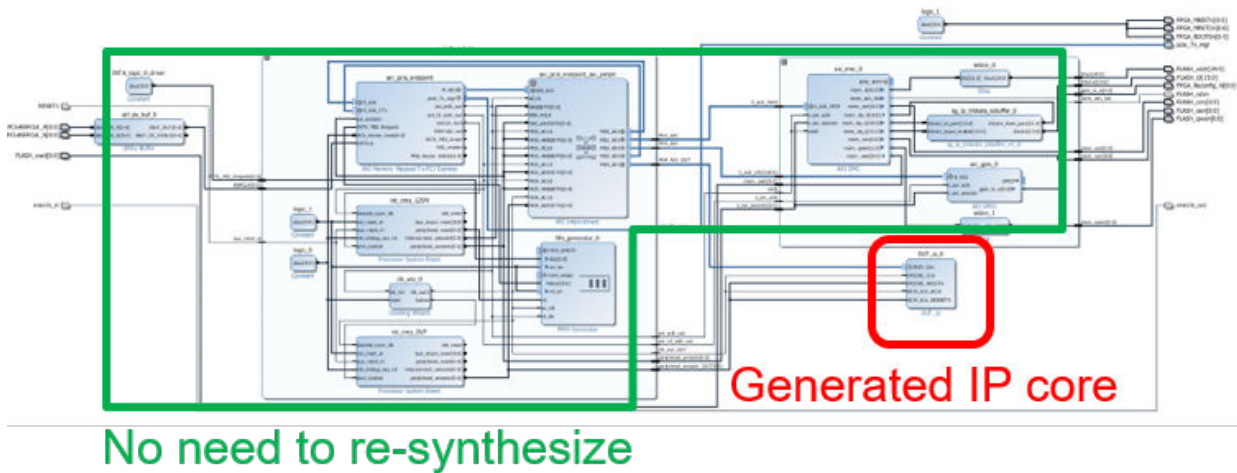
To reuse the IP cache when you run the workflow, the IP synthesis has to match the hash index in the IP cache. The hash index match corresponds to a hit in the IP cache. To hit the IP cache in subsequent runs, use the same:

- 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.

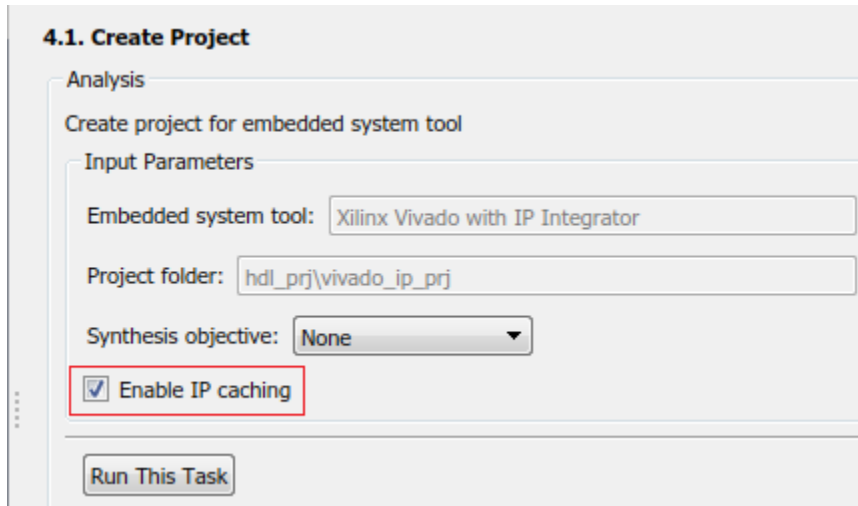


For large reference designs, the OOC flow improves synthesis run time, because you do not have to resynthesize the IP when you modify your design and run the workflow. To learn more about the OOC workflow and IP synthesis options, refer to the Xilinx documentation.

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 I0333-325K 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.

4.3. Build FPGA Bitstream

Analysis

Synthesis and generate bitstream for embedded system on FPGA

Input Parameters

Run build process externally

Tcl file for synthesis build: Default ▾ Browse...

Run This Task

Result: ✓ Passed

Passed Build Embedded System.

Synthesis Tool Log:

Task "Build FPGA Bitstream" successful.
 Generated logfile: [hdl_pri\hdlsrc\axi_video_basic\workflow_task_BuildFPGABitstream.log](#)
 WARNING: Default location for XILINX_VIVADO_HLS not found:

```
***** Vivado v2016.2 (64-bit)
**** SW Build 1577090 on Thu Jun  2 16:32:40 MDT 2016
**** IP Build 1577682 on Fri Jun  3 12:00:54 MDT 2016
** Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.
```

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 = 3575924
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_YCbCrtoRGB_0_0, cacheID = e71459f
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_xbar_0, cacheID = d0f0971cb77bcae
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_axis2hdmi_0_0, cacheID = 7601a322
...
```

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 40-2
- “Custom IP Core Report” on page 40-5
- “Board and Reference Design Registration System” on page 41-33
- “Run HDL Workflow with a Script” on page 31-42

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 40-34
“Step 2: Find the Critical Path” on page 40-37
“Step 3: Resolve Timing Failures” on page 40-42

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” for information about how to run the IP Core Generation workflow.

- “FPGA Programming and Configuration” on page 41-52 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.

4.3. Build FPGA Bitstream

Analysis

Synthesis and generate bitstream for embedded system on FPGA

Input Parameters

Run build process externally

Tcl file for synthesis build: Default ▾ Browse...

Run This Task

Result: ✔ Passed

Warning Run build process externally: The system build has been launched in an external shell, please check the external console to make sure the bitstream generation is completed. The system build may fail if the timing constraints are not met by the synthesis tool. If a timing failure occurs, the bitstream will be renamed to "system_timingfailure.sof". For more details on how to resolve the timing failure, please follow the "[Article on timing failure](#)" and also read the "[timing report](#)" for details. The system build may also fail because of other reasons, please read the log in the external console to identify reason.

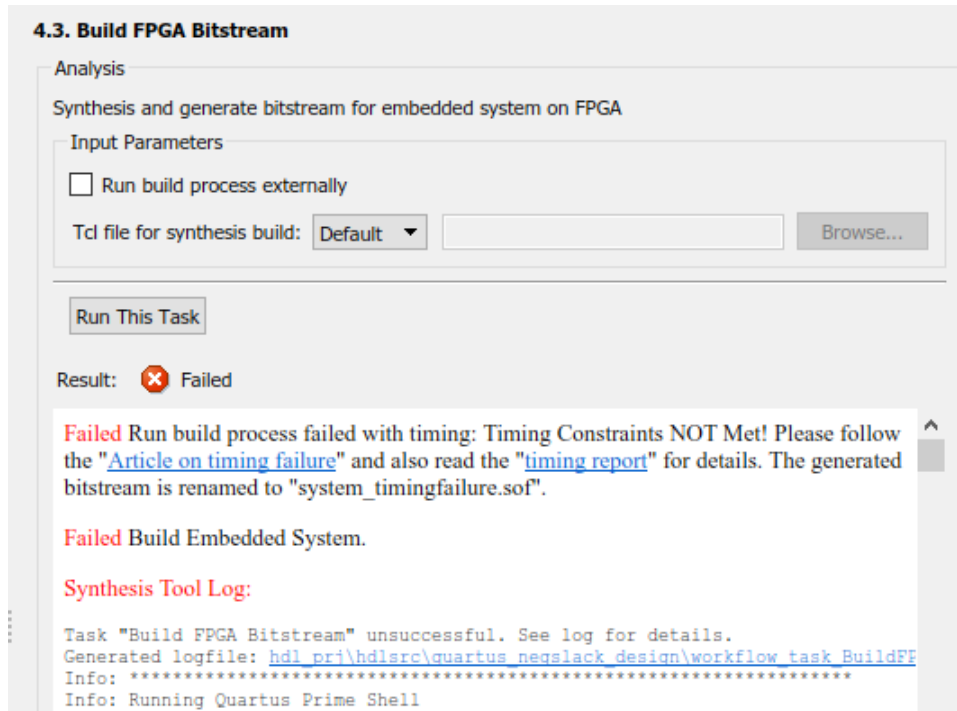
Passed Build Embedded System.

Synthesis Tool Log:

```
Task "Build FPGA Bitstream" successful.
Generated logfile: hdl_prj\hdlsrc\quartus_negslack_design\workflow_task_BuildFPGA
Running embedded system build outside MATLAB.
Please check external shell for system build progress.
```

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

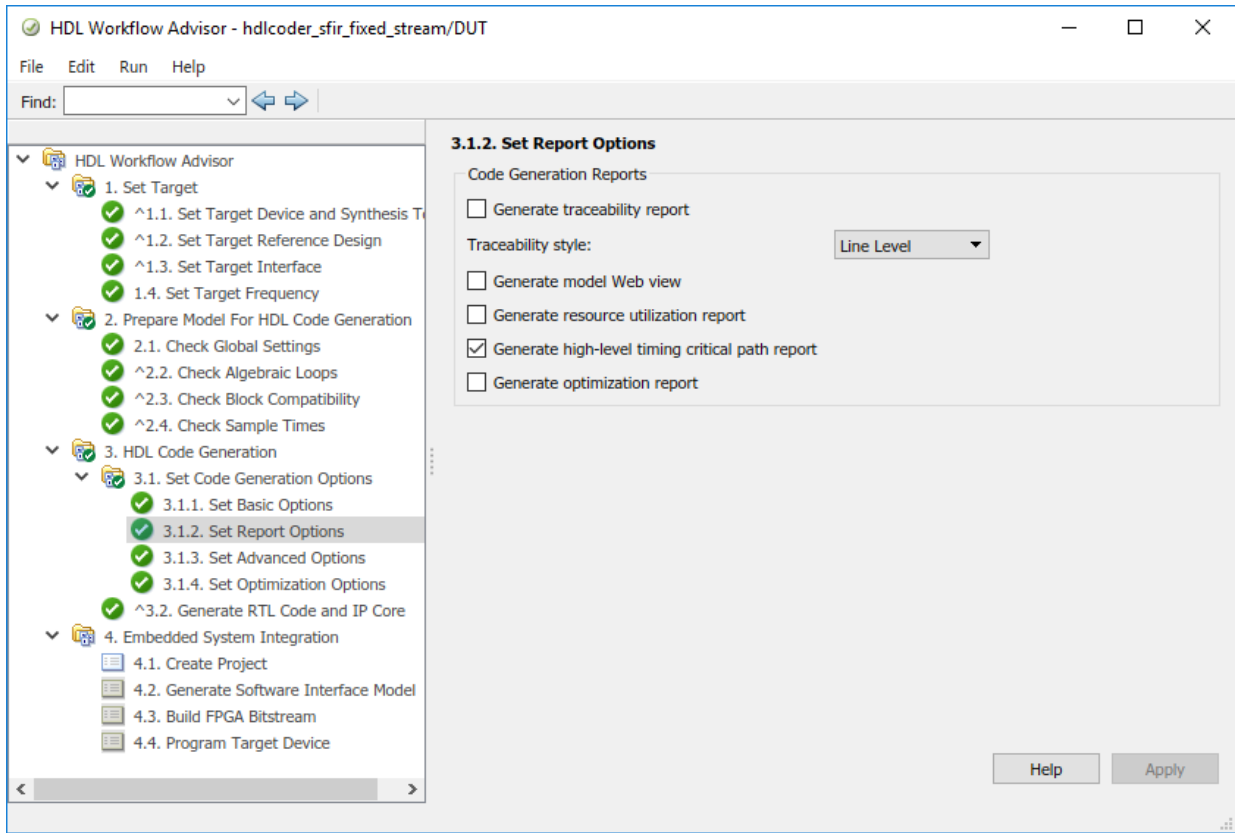
```
-----
Slack (VIOLATED) : -2.478ns (required time - arrival time)
Source:           system_top_i/led_count_ip_0/U0/u_led_count_ip_dut_inst/
                  u_led_count_ip_src_led_counter/HDL_Counter1_out1_reg[0]/C
                  (rising edge-triggered cell FDRE clocked by clk_out1_system_top_clk_wiz_0_0
                   {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%))
-----
```

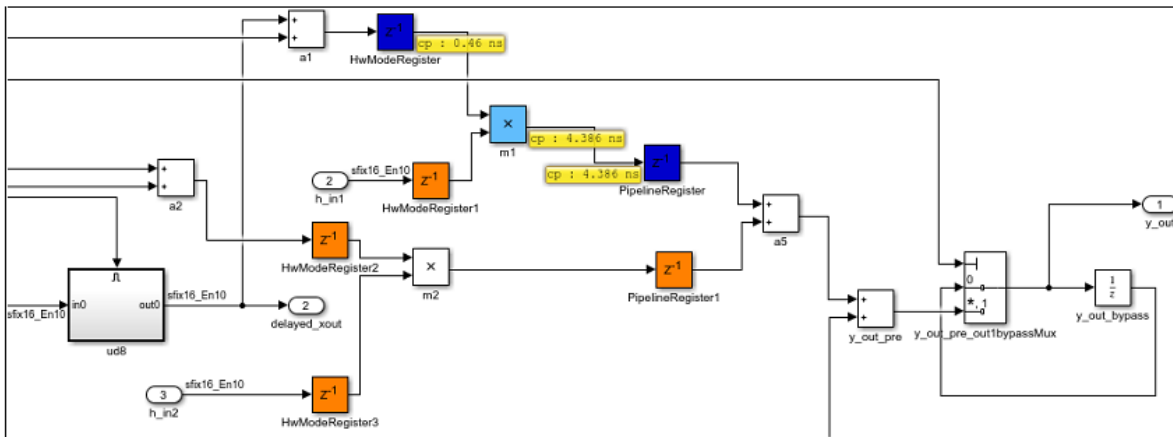
Strategy 2: Estimate Critical Path Without Running Synthesis

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 24-78.

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 31-42.

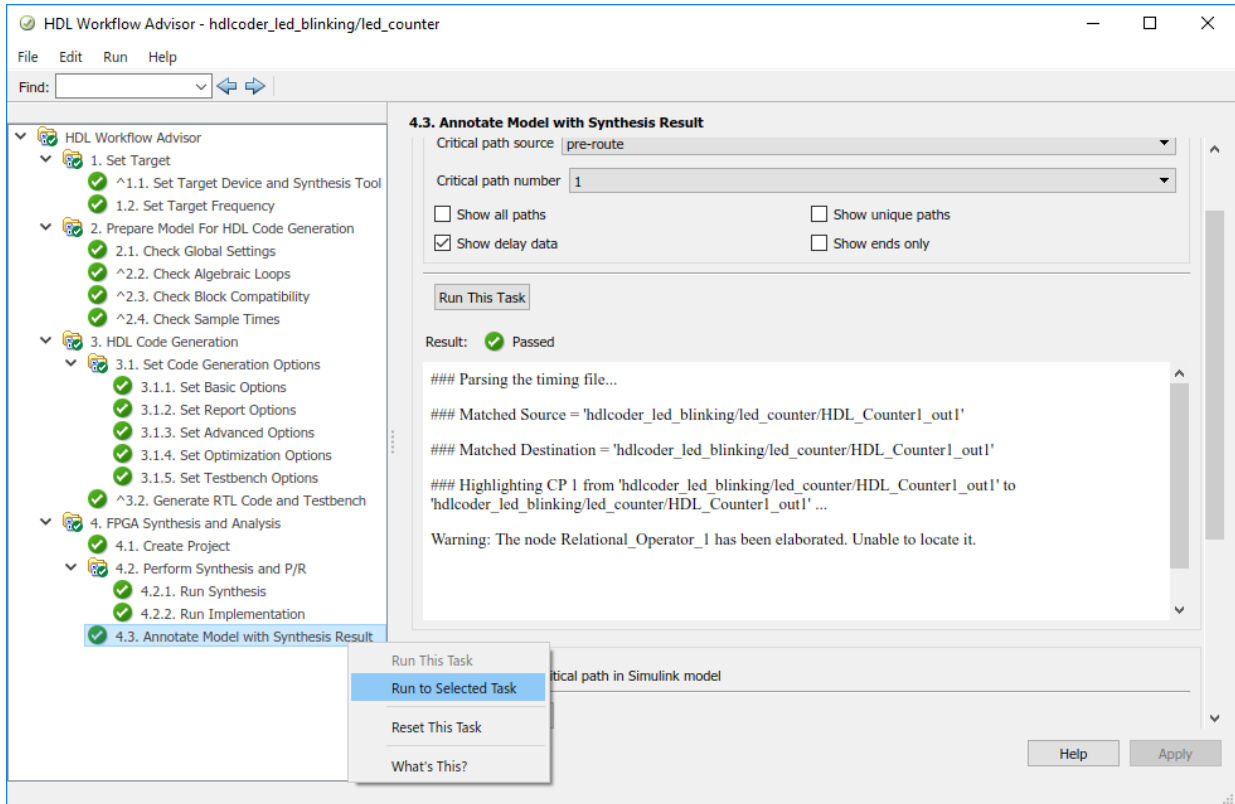
To use backannotation:

- 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.

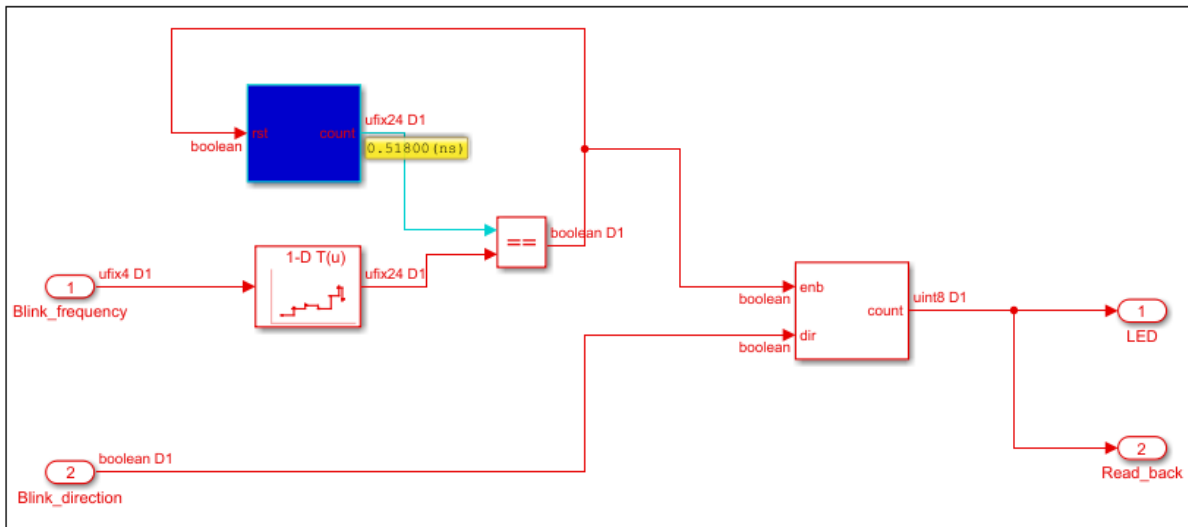
When you specify these settings, HDL Coder resets this task and all tasks that follow it.

- 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.



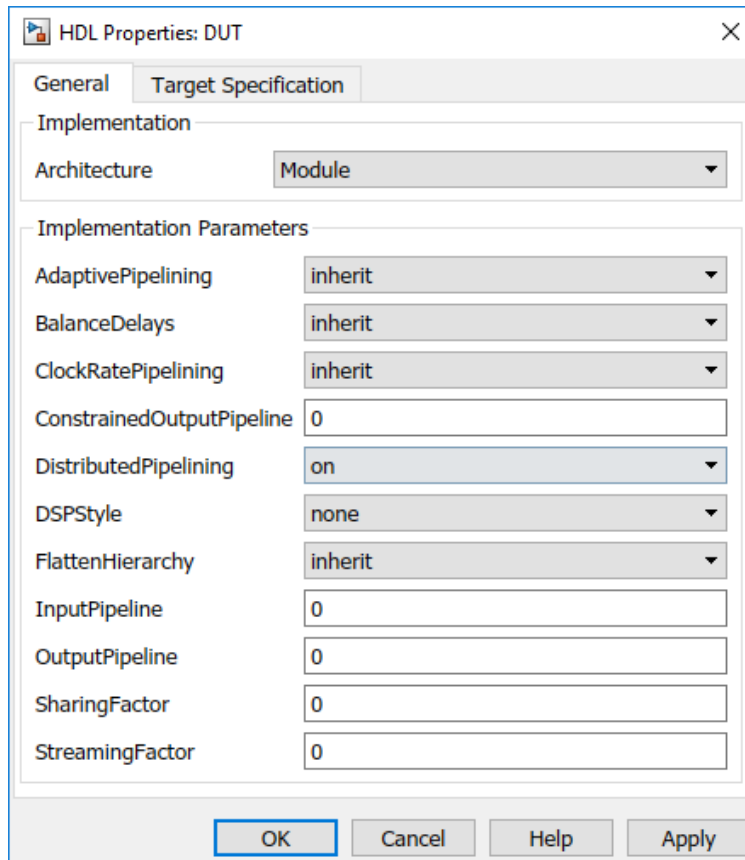
Step 3: Resolve Timing Failures

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 and hierarchical distributed pipelining optimizations move the existing delays you have in your design across the subsystem hierarchy. When you use hierarchical distributed pipelining enabled, make sure that all Subsystem blocks have DistributedPipelining enabled. 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.



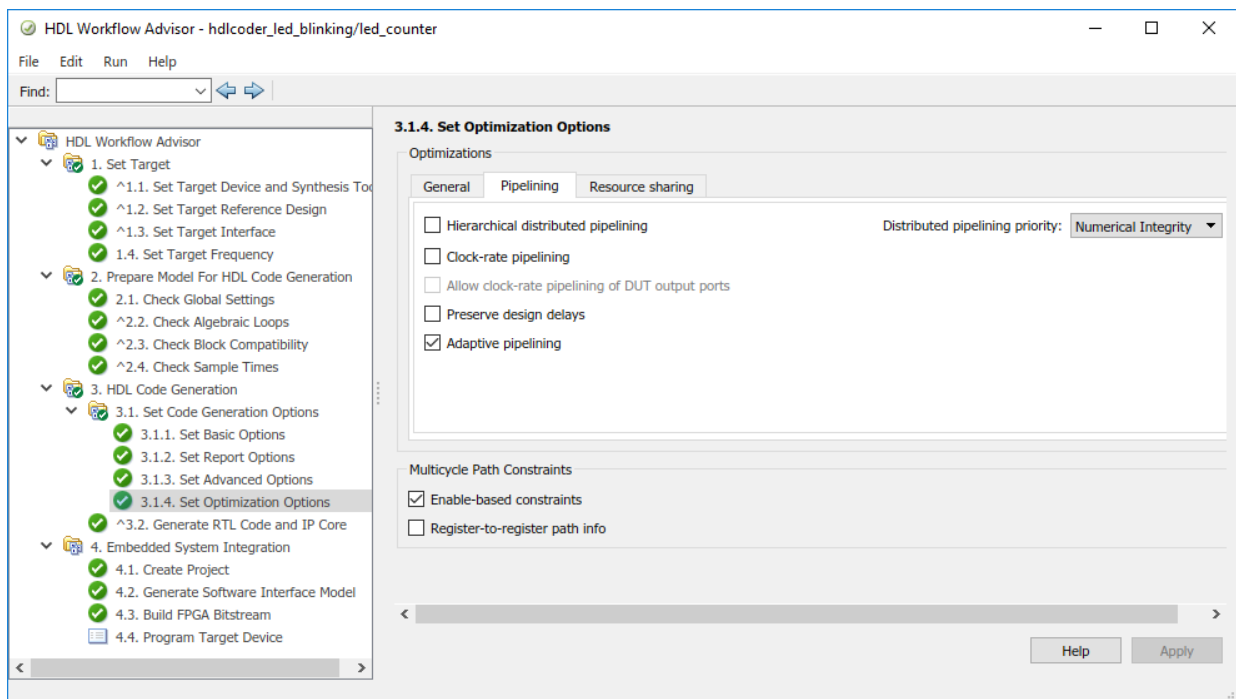
If distributed pipelining is unable to move the registers, you can add Delay blocks to your model, and then enable the **Preserve design delays** setting. Reset the **Check Global Settings** task and run the workflow to the **Build FPGA Bitstream** task. You can iterate on this approach and use other 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 master 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”.



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” on page 13-9.

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 Workflow 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 41-2
- “Custom IP Core Generation” on page 40-2
- “IP Core Generation Workflow for Speedgoat I/O Modules” on page 41-91
- “Program Target FPGA Boards or SoC Devices” on page 40-65

Define and Add IP Repository to Custom Reference Design

In this section...

“Create an IP Repository Folder Structure” on page 40-47

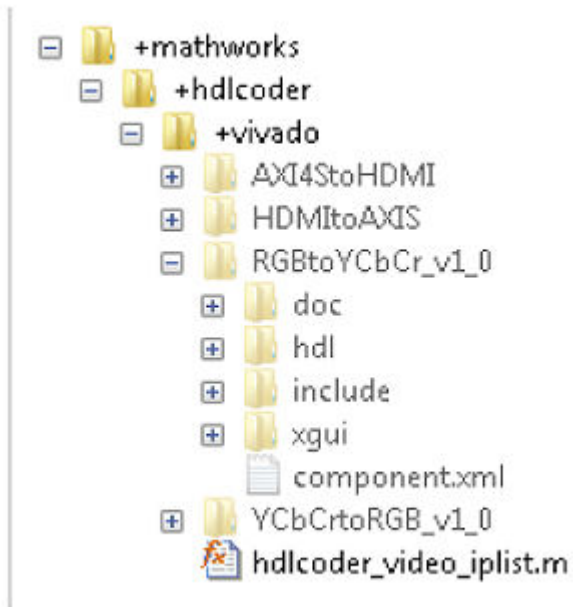
“Define IP List Function” on page 40-49

“Add IP List Function to Reference Design Project” on page 40-50

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.

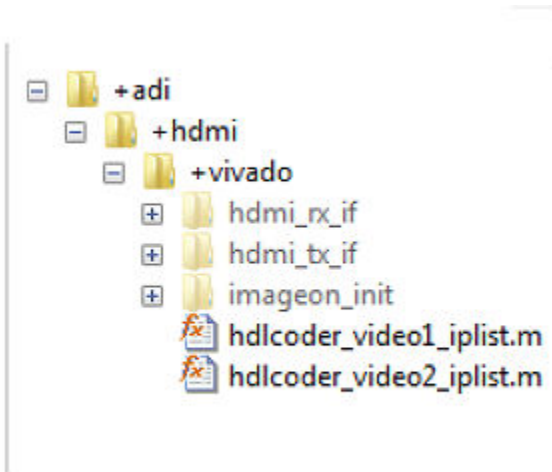
Create an IP Repository Folder Structure

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_ip_list 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_ip_list 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 = {};
```

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 = {'2016.2'};

%% 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 and Register Custom Board and Reference Design for SoC Workflow

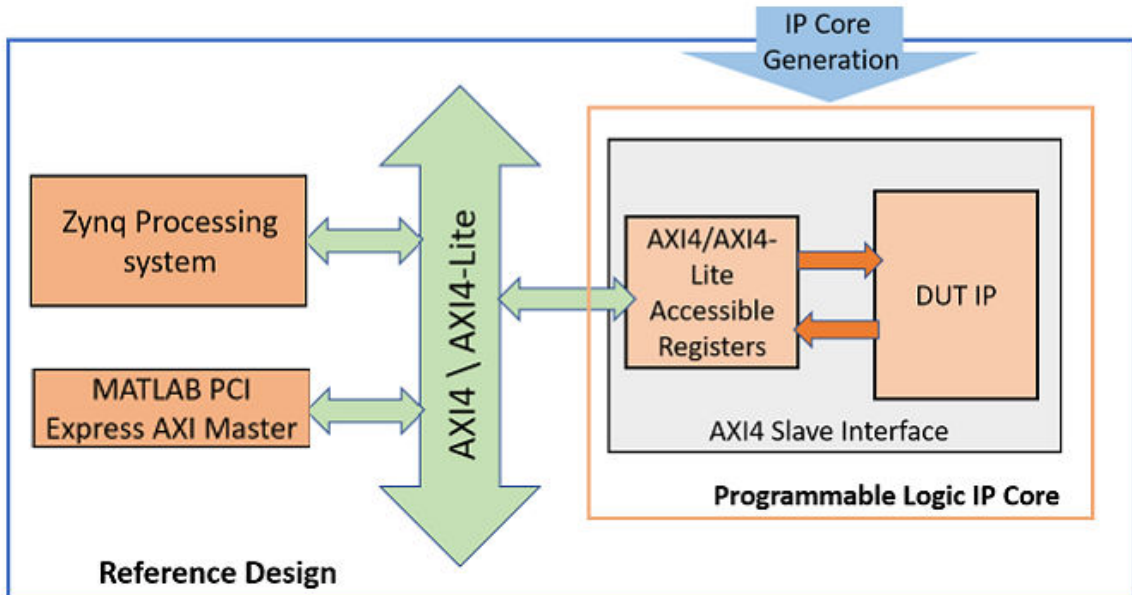
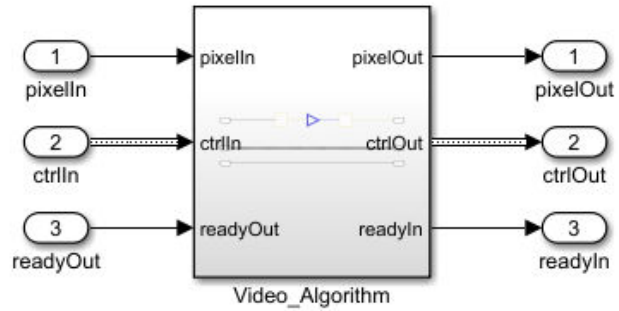
More About

- “Board and Reference Design Registration System” on page 41-33
- “Register a Custom Board” on page 41-37
- “Register a Custom Reference Design” on page 41-41

Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface

In this section...
“Vivado-Based Reference Designs” on page 40-53
“Qsys-Based Reference Designs” on page 40-55

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 Master 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 MATLAB as AXI
% Master 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_master_0/axi4m'}, ...
  'IDWidth',             13);

% ...
```

In this example, the two AXI Master IP are the HDL Verifier MATLAB as AXI Master IP and the ARM processor. Based on the syntax of the `MasterAddressSpace`, for the HDL Verifier MATLAB as AXI Master IP, the AXI Master Instance Name is `hdlverifier_axi_master_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', 'MATLAB_as_AXI_Master_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 MATLAB as AXI Master IP, the AXI Master Instance Name is `MATLAB_as_AXI_Master_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_jw_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”
- “Define Custom Board and Reference Design for Intel SoC Workflow”

More About

- “Board and Reference Design Registration System” on page 41-33
- “Register a Custom Board” on page 41-37
- “Register a Custom Reference Design” on page 41-41

Meet Timing Requirements Using Enable-Based Multicycle Path Constraints

In this section...

“How Enable-Based Multicycle Path Constraints Work” on page 40-58

“Specify Enable-Based Constraints” on page 40-60

“Benefits of Using Enable-Based Constraints” on page 40-61

“Modeling Guidelines” on page 40-61

“Multicycle Path Constraints for Various Synthesis Tools” on page 40-62

“Caveats and Limitations” on page 40-63

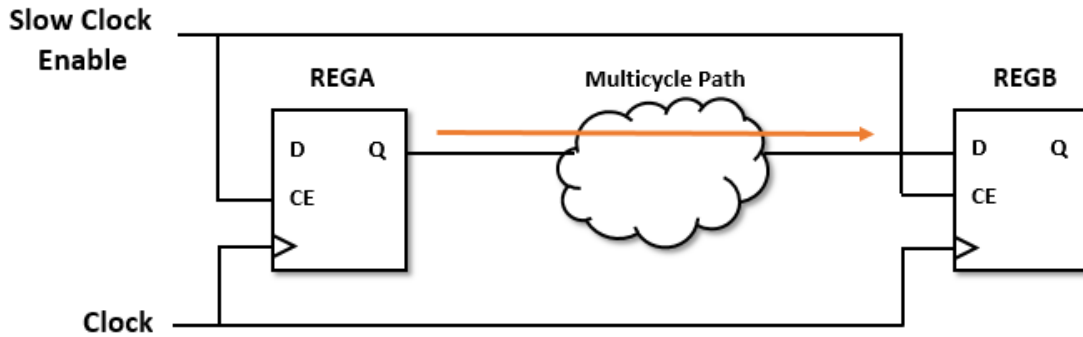
If your Simulink model contains multiple sample rates or uses speed and area optimizations that insert pipeline registers, your design can have multicycle paths. Multicycle paths are data paths between two registers that operate at a sample rate slower than the FPGA clock rate and therefore take multiple 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 master 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 master 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.

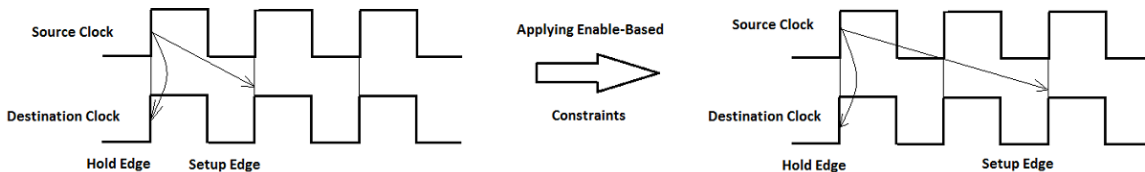
How Enable-Based Multicycle Path Constraints Work

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 to 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 specify the multicycle path information to the synthesis tool, it is not recommended to use the **Register-to-register path info** setting in the **Target and Optimizations** pane. If you use this setting, the code generator outputs a text file that describes the multicycle path information in a format that is not native to a particular synthesis tool. You must convert this information to the multicycle path constraints format required by your synthesis tool.

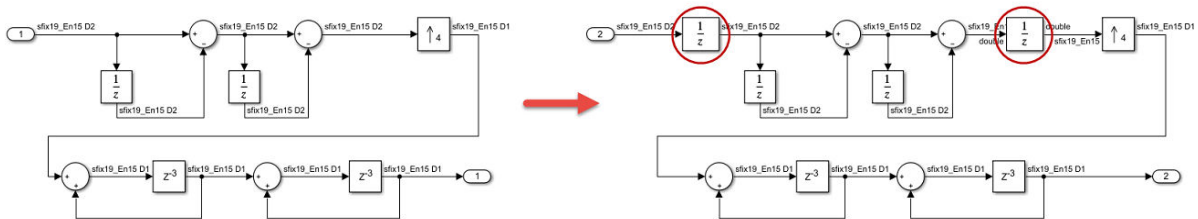
When you use the enable-based constraints setting:

- The generated constraints are more robust to name changes in synthesis tools.
- HDL code generation is faster than when you use the **Register-to-register path info** setting.
- The **Target workflow** can be Generic ASIC/FPGA, FPGA Turnkey, 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 `hdlcoder_clockdemo` at the command line in MATLAB, you see a multirate CIC Interpolation filter implemented in single clock mode. 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 -s
set_multicycle_path 3 -to [get_fanouts $enbreg -through [get_pins -hier *|ena]] -end -s
```

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}}]
set reglist [get_cells -of [filter [all_fanout -flat -endpoints_only $enbregnet] IS_ENA]]
```

```
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. This code snippet shows this keyword in the synthesis netlist when you run any of the supported target workflows.

```
...
ATTRIBUTE mcp_info: string

ATTRIBUTE mcp_info 0F 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 snippet shows that 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
```

This code snippet shows the clock constraints that get generated when you run the Generic ASIC/FPGA, FPGA Turnkey, or the Simulink Real-Time FPGA I/O workflow with Xilinx ISE.

```
# 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.

Caveats and Limitations

- The multicycle path constraints file is not supported with the FPGA-in-the-Loop workflow.

- The **IP Core Generation** workflow does not generate a clock constraint and therefore does not support multicycle path constraints generation with Xilinx ISE.
- If the slow-rate region is not bounded by registers, multicycle path constraints requires 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 in 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 therefore may not retain the slow-rate registers in your design.
- HDL Coder does not generate multicycle path constraints for single-rate models.
- The code generator does not output the multicycle path constraints file if you use the multiple clock mode.

See Also

Related Examples

- “Use Multicycle Path Constraints to Meet Timing for Slow Paths”

More About

- “Multicycle Path Constraints” on page 14-37
- “Clock-Rate Pipelining” on page 24-63
- “Adaptive Pipelining” on page 24-68
- “Generate Multicycle Path Information Files” on page 22-19

Program Target FPGA Boards or SoC Devices

In this section...

“How to Program Target Device” on page 40-65

“Programming Methods” on page 40-67

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 31-2.

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.

Result:  Passed

Passed Build Embedded System.

Synthesis Tool Log:

```
Task "Build FPGA Bitstream" successful.
Generated logfile: hdl\_prj\hdlsrc\hdlcoder\_led\_blinking\workflow\_task\_BuildFPGABitstream.log

Running embedded system build outside MATLAB.
Please check external shell for system build progress.

The generated bitstream file is located at: hdl_prj\vivado_ip_prj\vivado_prj.runs\impl_1\system_top_wrapper.bit

Generate an HDL Workflow Command-Line Interface script to program the target device: hdlworkflow\_ProgramTargetDevice.m.
```

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.RunTaskGenerateSoftwareInterfaceModel = 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 31-42.

Programming Methods

Download

If you have Embedded Coder, and use the reference designs for Xilinx 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 Zynqboard, and then keeps the bitstream on the SD card persistently.

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 = {'2017.2'};

% ...

hRD.CallbackCustomProgrammingMethod =
    @my_reference_design.callback_CustomProgrammingMethod;
```

See Also

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow

More About

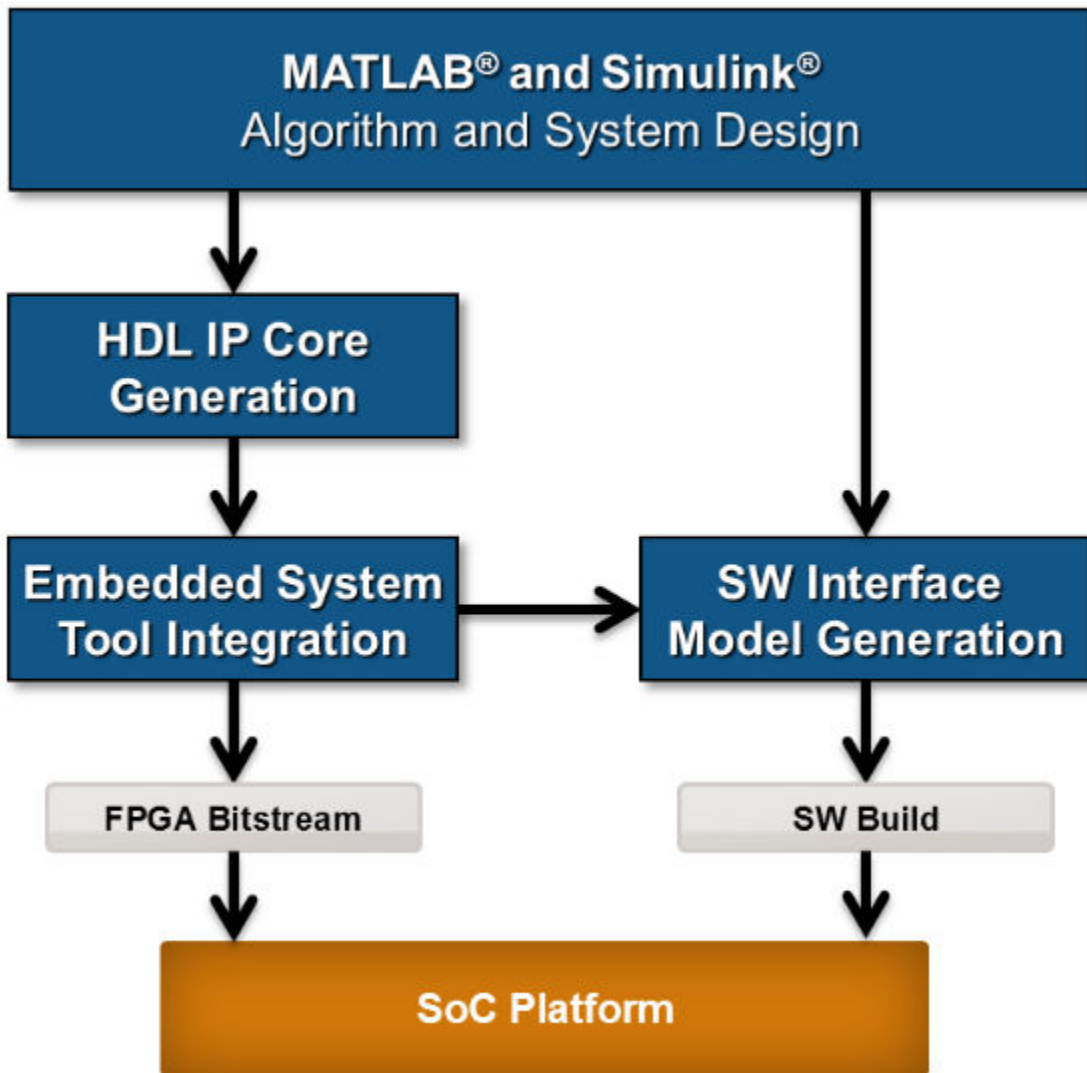
- “Program Target Device” on page 37-31
- “IP Core Generation Workflow for Standalone FPGA Devices” on page 41-86
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 41-2

Target SoC Platforms and Speedgoat Boards

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 41-2
- “Model Design for AXI4 Slave Interface Generation” on page 41-11
- “Model Design for AXI4-Stream Interface Generation” on page 41-19
- “Multirate IP Core Generation” on page 41-27
- “Board and Reference Design Registration System” on page 41-33
- “Register a Custom Board” on page 41-37
- “Register a Custom Reference Design” on page 41-41
- “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 41-45
- “FPGA Programming and Configuration” on page 41-52
- “Model Design for AXI4-Stream Video Interface Generation” on page 41-64
- “Model Design for AXI4 Master Interface Generation” on page 41-75
- “IP Core Generation Workflow for Standalone FPGA Devices” on page 41-86
- “IP Core Generation Workflow for Speedgoat I/O Modules” on page 41-91

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.

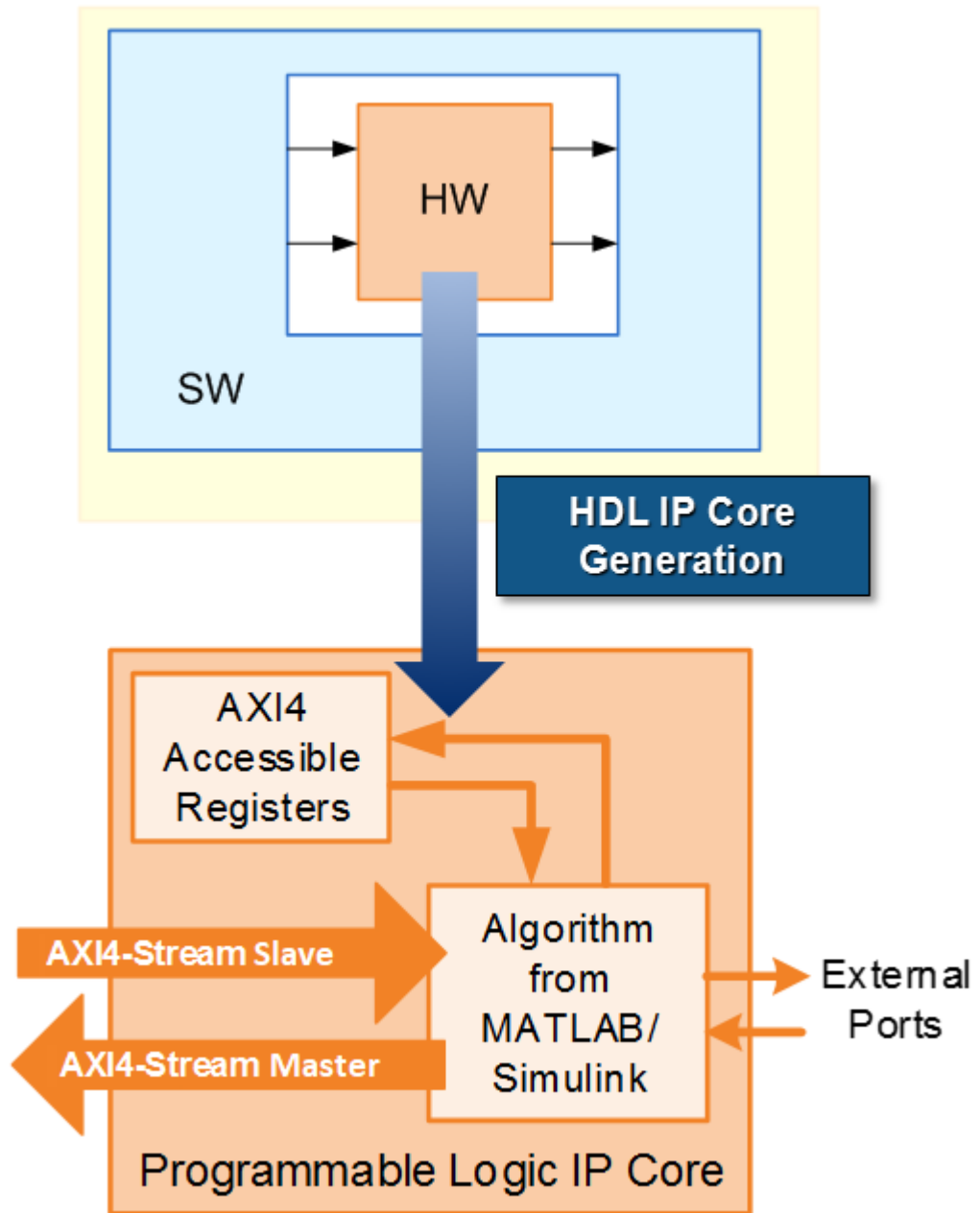


- 1 *MATLAB and Simulink Algorithm and System Design*: You begin by implementing your design in MATLAB or Simulink. When the design behavior meets your requirements, decide how to partition your design: which parts you want to run in hardware, and which parts you want to run in embedded software.

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.

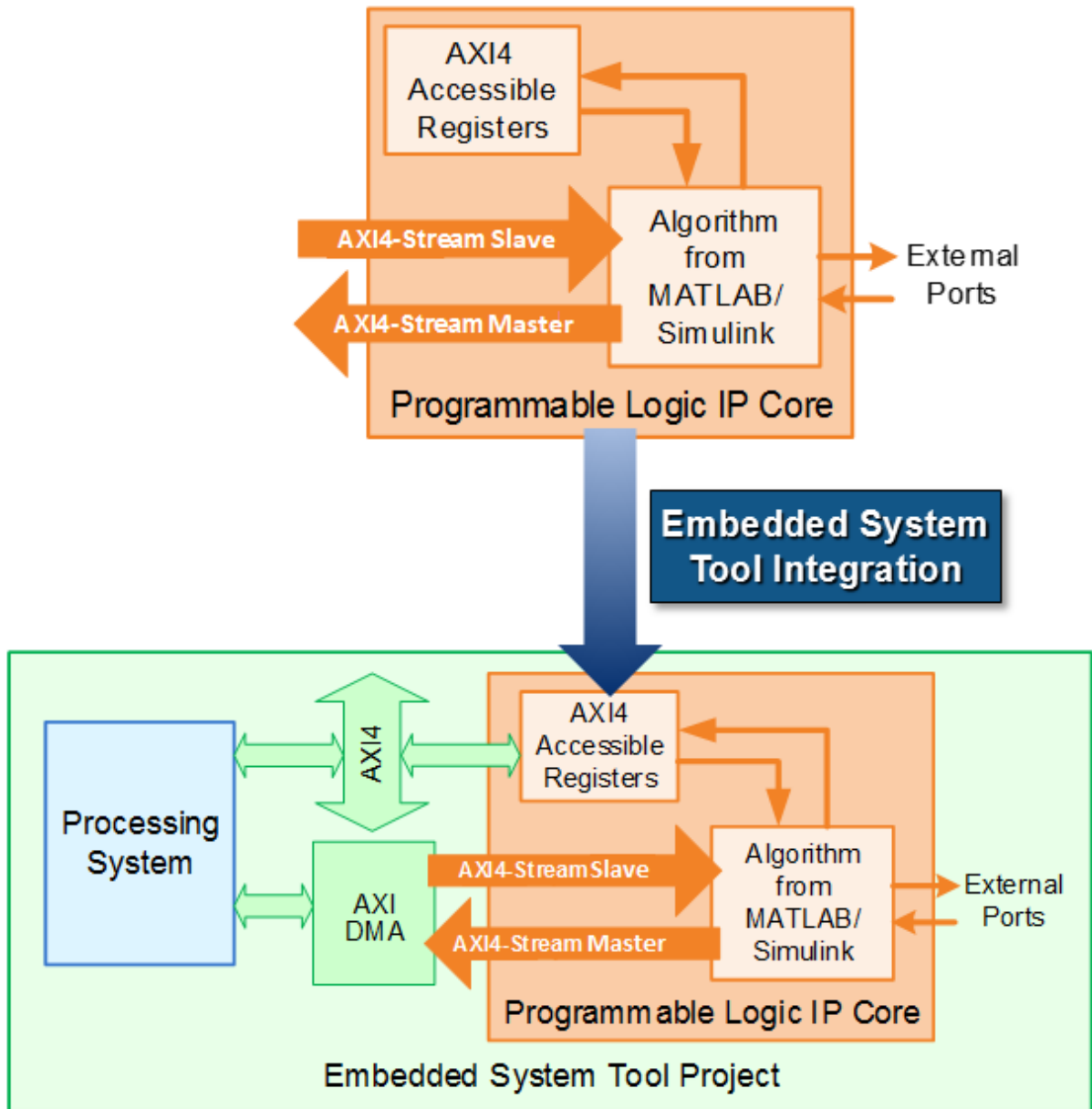
The following diagram shows a design that has been partitioned into a hardware part, in orange, and software part, in blue. HDL IP core generation creates an IP core from the hardware part of the model. The IP core includes hardware interface components such as AXI4 accessible registers, AXI4 or AXI4-Lite 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.

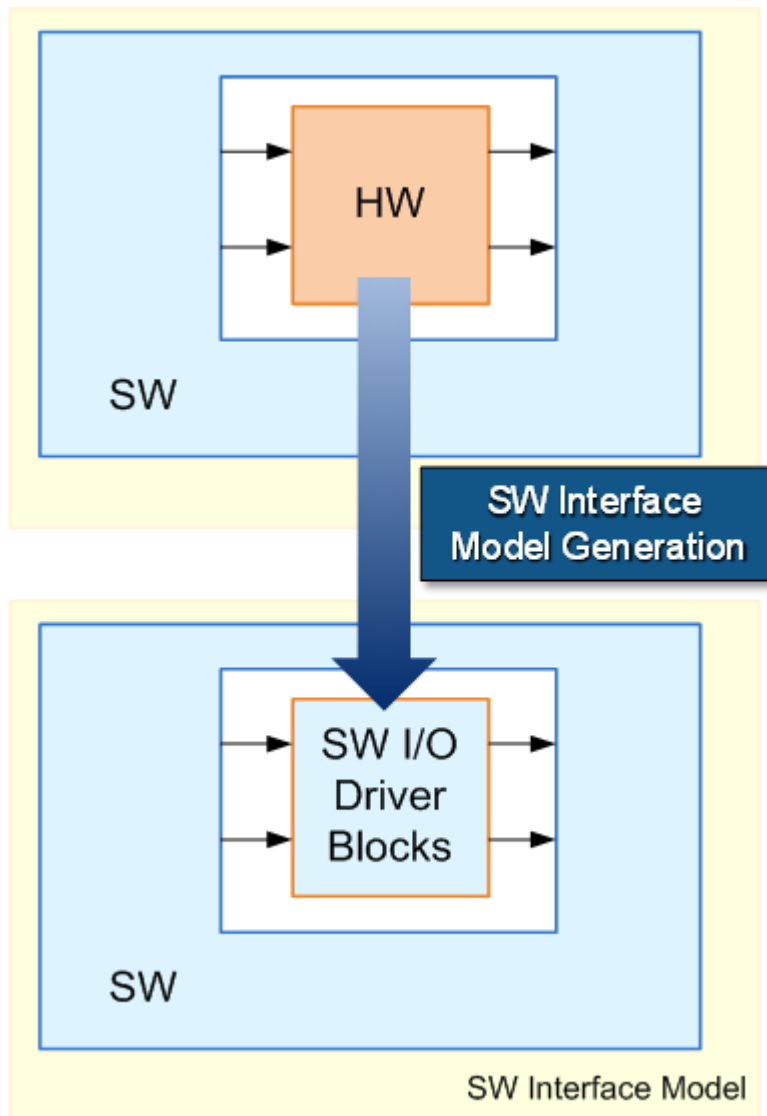


- 4 *SW Interface Model Generation* (requires a Simulink license and Embedded Coder license): In the HDL Workflow Advisor, after you generate the IP core and insert it into the reference design, you can optionally generate a software interface model. The software interface model is your original model with AXI driver blocks replacing the hardware part.

If you have an Embedded Coder license, you can automatically generate the software interface model, 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.

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

Related Examples

- “Xilinx Zynq Platform”
- “Intel SoC Devices”

Model Design for AXI4 Slave Interface Generation

In this section...

“Considerations” on page 41-11

“Map Scalar Ports to AXI4 Slave Interface” on page 41-12

“Map Vector Ports to AXI4 Slave Interface” on page 41-13

“Read Back Value of AXI4 Slave Interfaces” on page 41-14

“Optimize AXI4 Slave Read Back Logic” on page 41-17

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 that you want to map to the AXI4 slave interfaces. HDL Coder then maps the data ports to memory-mapped AXI4 slave interfaces 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 or vector 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.
- You can use a single-rate design or a design with multiple sample rates without any restrictions when mapping the DUT ports to AXI4 slave interfaces by using the Free Running synchronization mode for **Processor/FPGA Synchronization**.
- You can use the Coprocessing-Blocking mode for **Processor/FPGA Synchronization** when mapping to AXI4 or AXI-Lite interfaces. Other interface types such as AXI4-Stream and AXI4 Master do not support this mode. See also “Processor and FPGA Synchronization” on page 40-18.

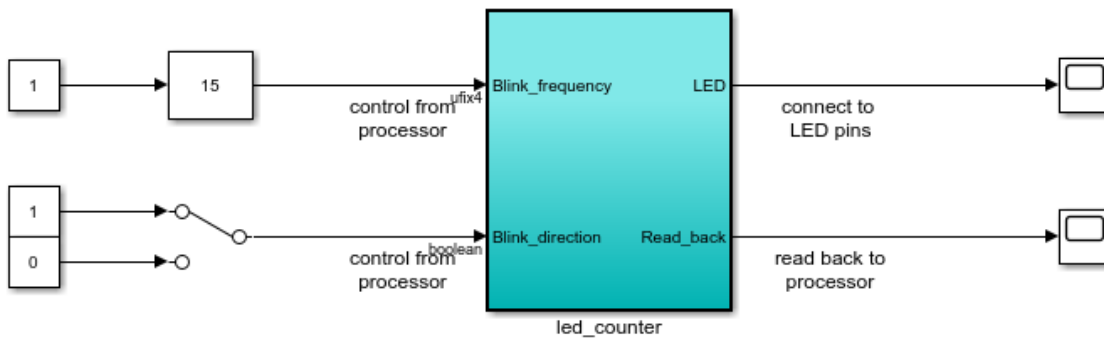
Map Scalar Ports to AXI4 Slave Interface

When you use scalar data types at the DUT interface ports, you can map the interface ports directly 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`.

```
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.

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 input values of the hardware subsystem. In the embedded software, this means that the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. 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”
- “Getting Started with Targeting Intel SoC Devices”

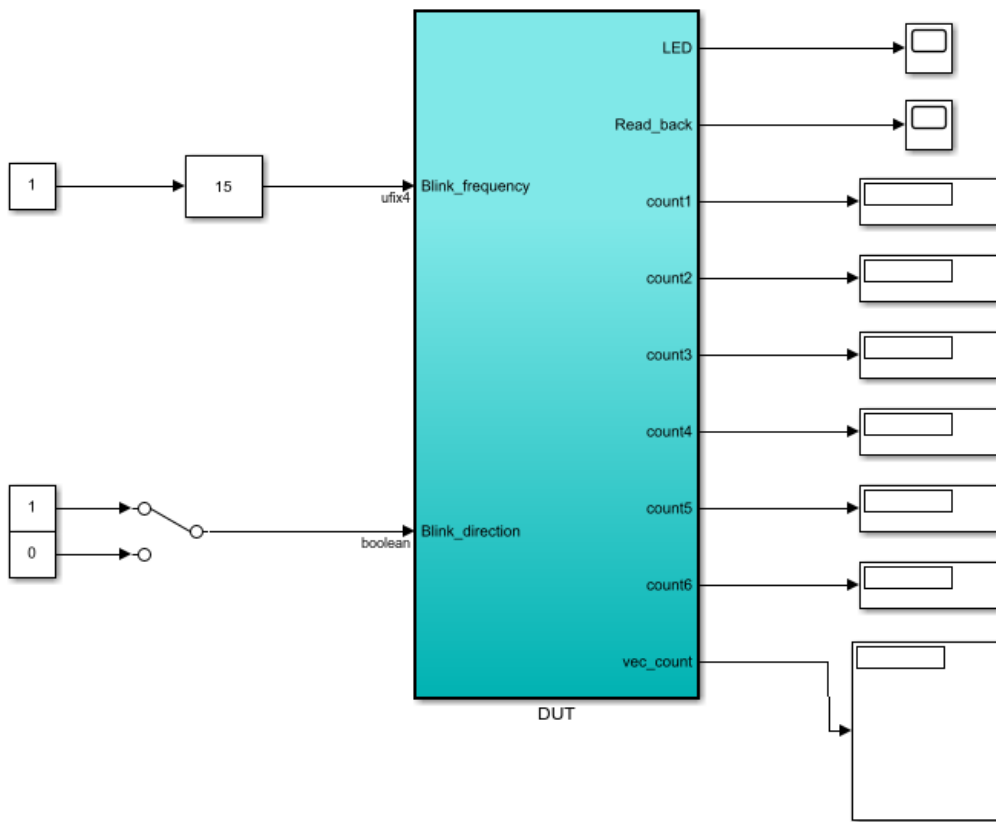
Map Vector Ports to AXI4 Slave Interface

When you use vector data types at the DUT interface ports, you can map the interface ports directly 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, which makes 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, in the **Set Target Interface** task, you see that the input ports and output ports map to AXI4-Lite interfaces.

To learn more, see “IP Core Generation Workflow with a MicroBlaze processor: Xilinx Kintex-7 KC705”.

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 MATLAB as AXI Master 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 this task.

3.2. Generate RTL Code and IP Core

Analysis (^Triggers Update Diagram)

Generate RTL code and IP core for embedded system

Input Parameters

IP core name:

IP core version:

IP core folder:

IP repository:

Additional source files:

FPGA Data Capture buffer size:

Generate IP core report

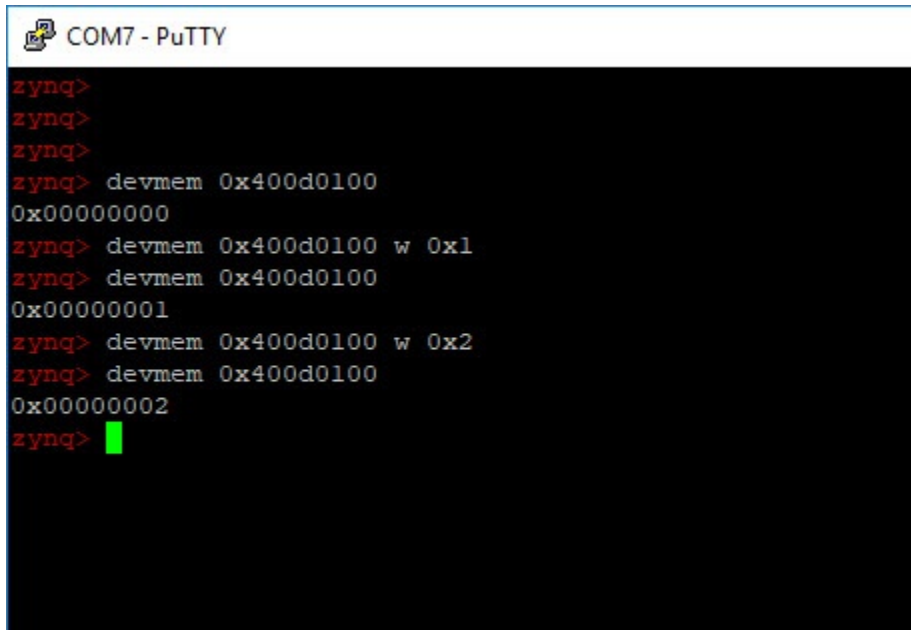
Enable readback on AXI4 slave write registers

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');
```

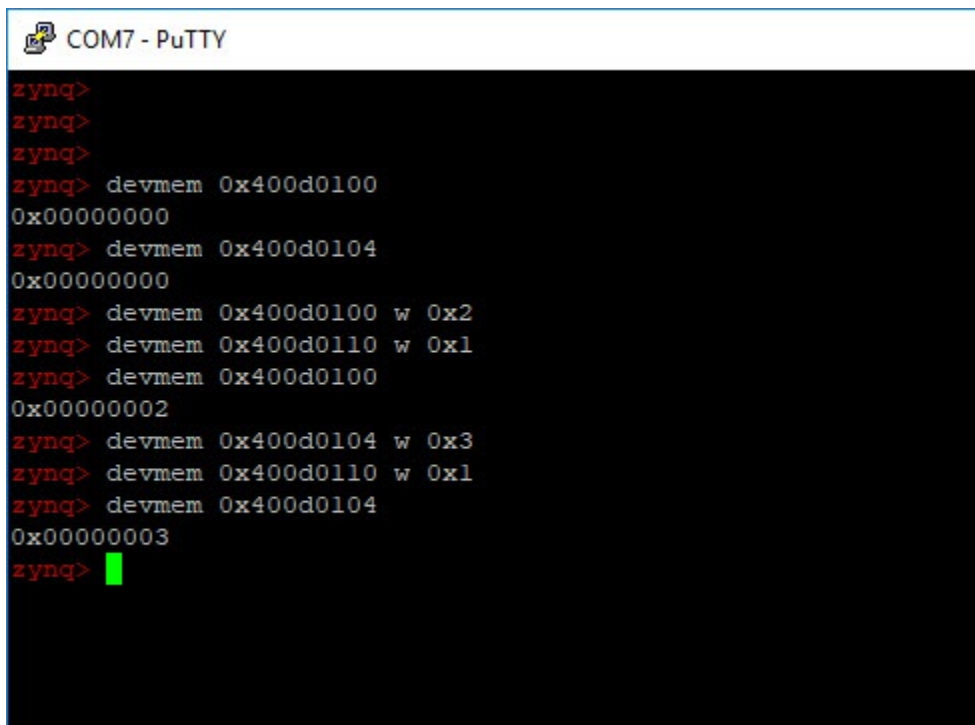
These examples show how you can read back values by using the `devmem` command in the Linux console with a program such as PuTTY.

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 Back 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 select a number between 5 and 50 for this ratio.

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 **AXI4SlavePortToPipelineRegisterRatio** 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

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 41-2
- “Custom IP Core Generation” on page 40-2

Model Design for AXI4-Stream Interface Generation

In this section...

- “Simplified Streaming Protocol” on page 41-19
- “Map Scalar Ports To AXI4-Stream Interface” on page 41-20
- “Map Vector Ports To AXI4-Stream Interface” on page 41-23
- “Model Designs with Multiple Sample Rates” on page 41-25
- “Restrictions” on page 41-25

With the HDL Coder software, you can implement a simplified, streaming protocol in your model. The software generates AXI4-Stream interfaces in the IP core.

Simplified Streaming Protocol

When you want to generate an AXI4-Stream interface in your IP core, in your DUT interface, implement the following signals:

- Data
- Valid

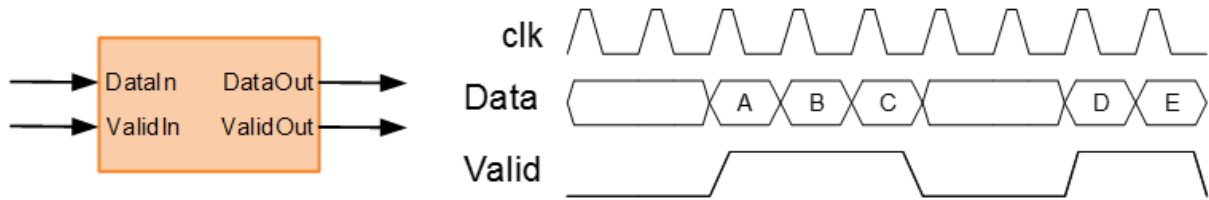
When you map scalar DUT ports to an AXI4-Stream interface, you can optionally model the following signals and map them to the AXI4-Stream interface:

- Ready
- Other protocol signals, such as:
 - TSTRB
 - TKEEP
 - TLAST
 - TID
 - TDEST
 - TUSER

Data and Valid Signals

When the Data signal is valid, the Valid signal is asserted.

Note 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.



Map Scalar Ports To AXI4-Stream Interface

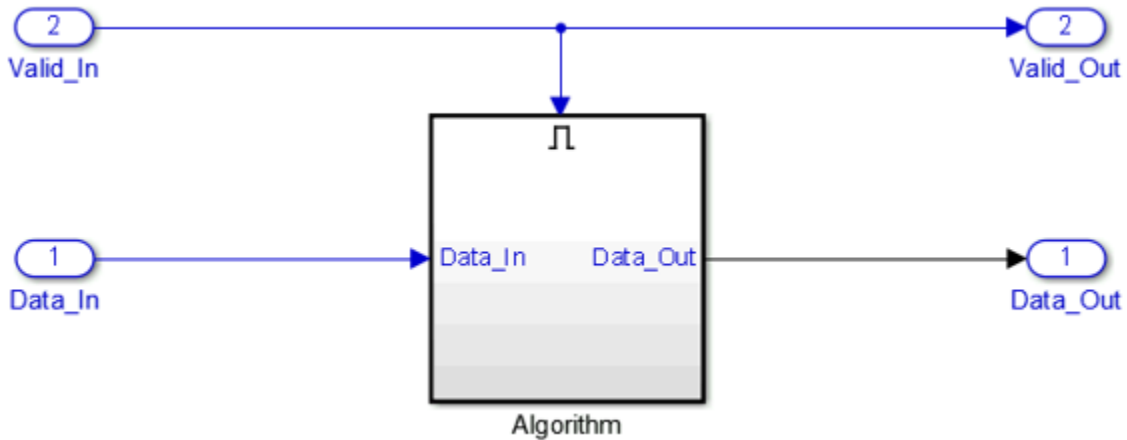
If you want to generate a hardware IP core, but do not need to model and simulate the interaction between the software and hardware, use scalar data ports at your DUT interface. Map the data ports to AXI4-Stream interfaces.

Data and Valid Signal Modeling Pattern

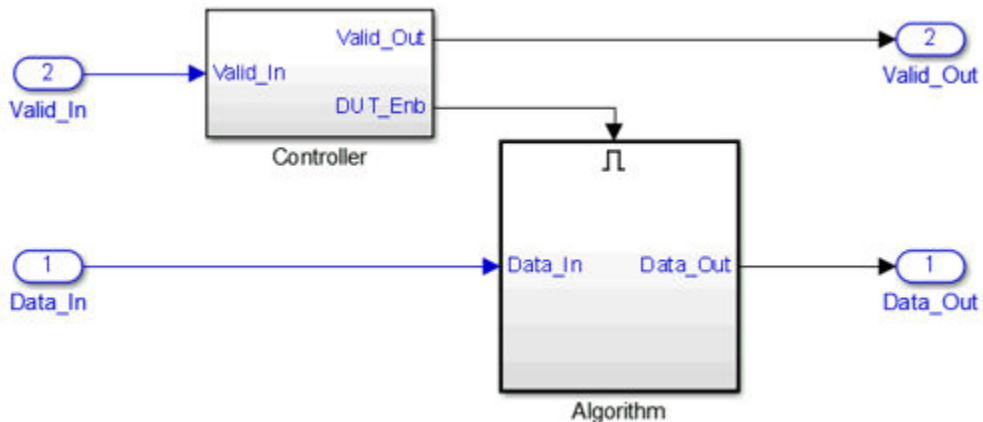
To model the Data and Valid signals in Simulink:

- 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.



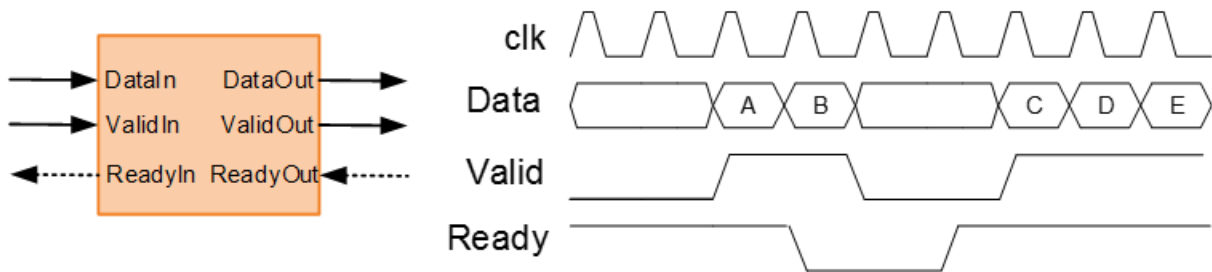
Ready Signal (Optional)

The AXI4-Stream interfaces in your DUT can optionally include a Ready signal. In a Slave interface, the Ready signal enables you to apply back pressure. In a Master interface, the Ready signal enables you to respond to back pressure.

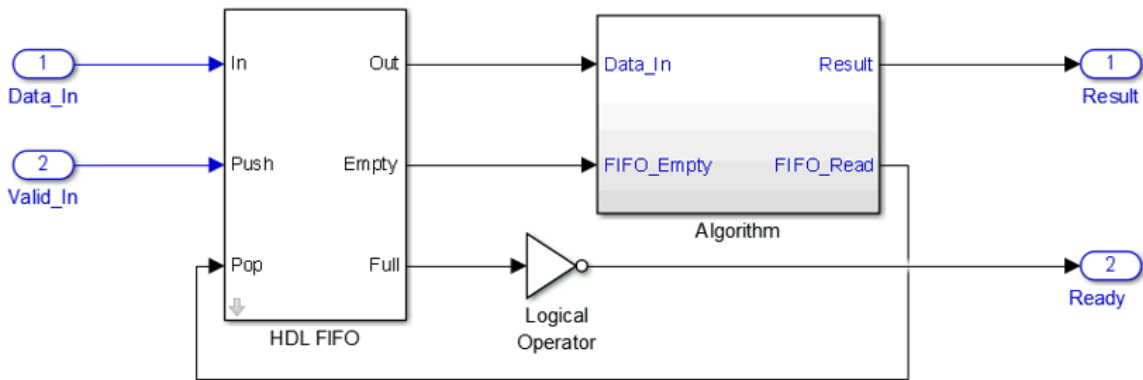
If you model the Ready signal in your AXI4-Stream interfaces, your Master interface ignores the Data and Valid signals one clock cycle after the Ready signal is deasserted. You can start sending Data and Valid signals once the Ready signal is asserted. You can send one more Data and Valid signal after the Ready signal is deasserted.

If you do not model the Ready signal, HDL Coder generates the signal and the associated back pressure logic.

Note This diagram illustrates the relationship between the Data, Valid, and Ready signals 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.



For example, if you have a FIFO in your DUT to store a frame of data, to apply back pressure to the upstream component, you can model the Ready signal based on the FIFO Full signal.



Note If you enable delay balancing, the coder can insert one or more delays on the Ready signal. Disable delay balancing for the Ready signal path.

Other Protocol Signals (optional)

You can optionally model other AXI4-Stream protocol signals. If you model only the required Data and Valid signals, the coder generates the TREADY and TLAST AXI4-Stream protocol signals.

If you do not model the TLAST signal, the coder generates a programmable register in the IP core so that you can specify your packet size. The details of the programmable packet size register are in your IP core generation report.

Map Vector Ports To AXI4-Stream Interface

If you want to model and simulate the system interaction between the software and hardware, and generate code for the software driver, use vector data ports at your DUT interface. Map the data ports to AXI4-Stream interfaces.

Data and Valid Signal Modeling Requirements

When you map vector ports to AXI4-Stream interfaces, your model has these requirements:

- Connect each DUT input vector data port to a Serializer1D block.

The Serializer1D block must have a ValidOut port and the Ratio set to the vector bit width.

- Connect each DUT output vector data port to a Deserializer1D block.

The Deserializer1D block must have a ValidIn port and the Ratio set to the vector bit width.

- Connect each scalar port that maps to an AXI4-Lite interface to a Rate Transition block.

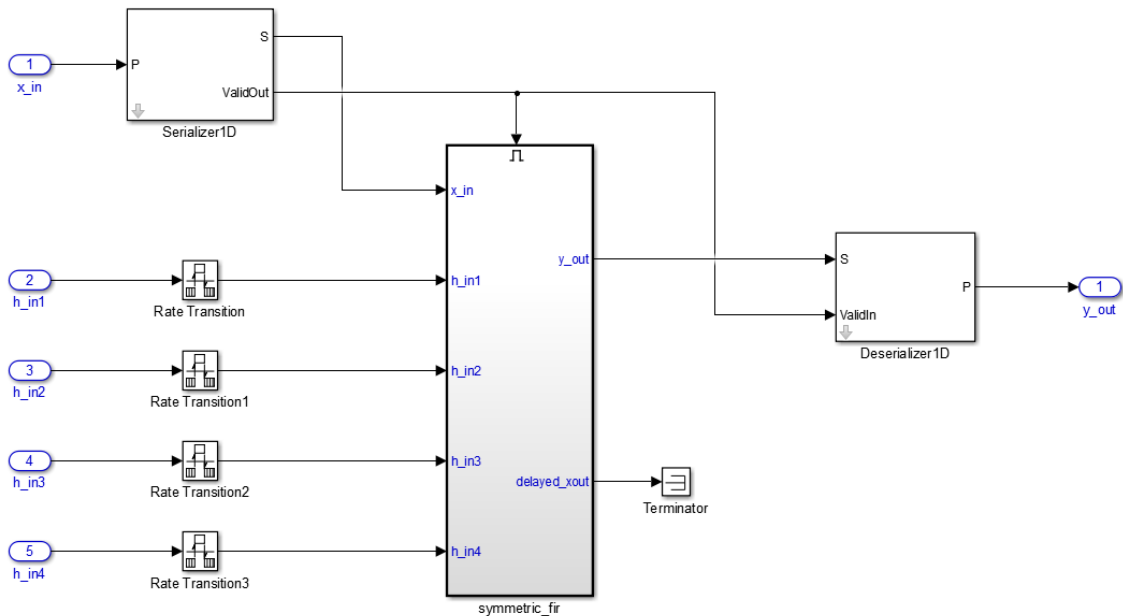
The Ratio in the Rate Transition block must match the Ratio in the Serializer1D and Deserializer1D blocks.

- Each scalar port that maps to an external port must have the same sample time as the streaming algorithm subsystem.

The streaming algorithm subsystem follows the same Data and Valid signal modeling pattern as for mapping scalar ports to an AXI4-Stream interfaces. See “Data and Valid Signal Modeling Pattern” on page 41-20.

Example

For an example that shows how to map vector ports to AXI4-Stream interfaces, open the `hdlcoder_sfir_fixed_vector` model. In the `hdlcoder_sfir_fixed_vector` model, `symmetric_fir` is the streaming algorithm 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-Stream Master or AXI4-Stream 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 41-27.

Restrictions

When you map scalar or vector DUT ports to AXI4-Stream interfaces:

- The DUT can have at most one AXI4-Stream Master channel and one AXI4-Stream Slave channel.
- 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.

When you map vector DUT ports to AXI4-Stream interfaces, you cannot use protocol signals other than Data and Valid. For example, Ready and TLAST are not supported.

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.

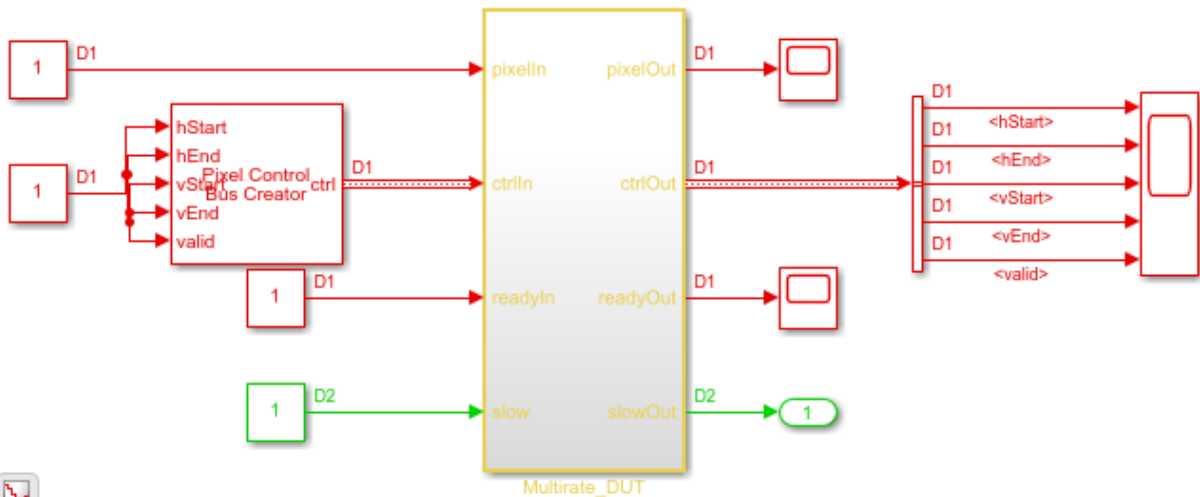
These examples illustrate how you can model your design with multiple sample rates when using AXI4-Stream, AXI4-Stream Video, or AXI4-Master Master interfaces.

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.

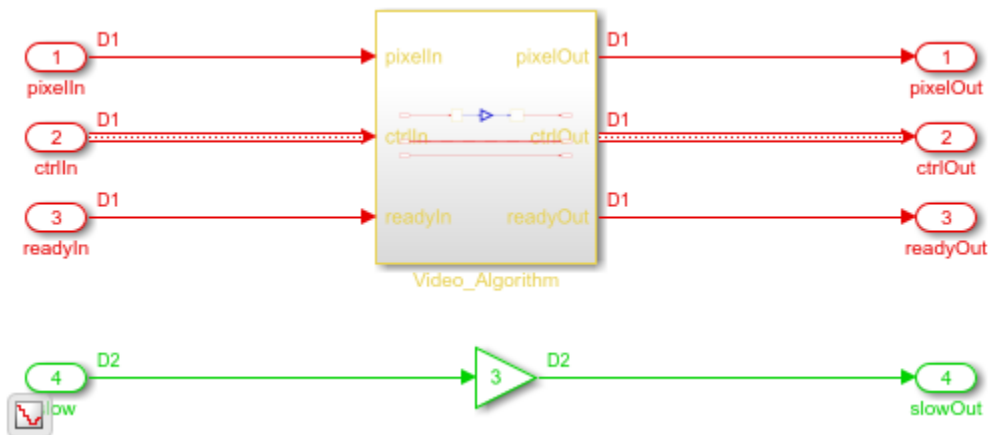
For an example, open the model `hdlcoder_axi_video_multirate`.

```
load_system('hdlcoder_axi_video_multirate')
set_param('hdlcoder_axi_video_multirate', 'SimulationCommand', 'update')
open_system('hdlcoder_axi_video_multirate')
```



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.

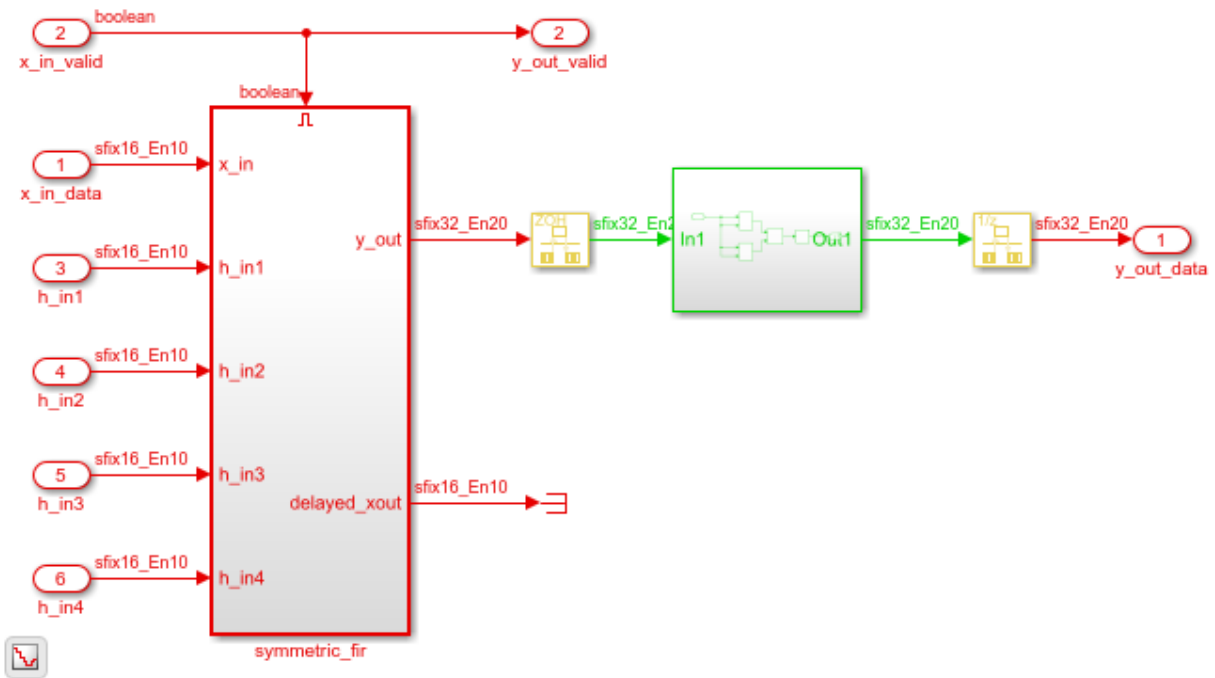
See also Model Design for AXI4-Stream Video Interface Generation.

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.

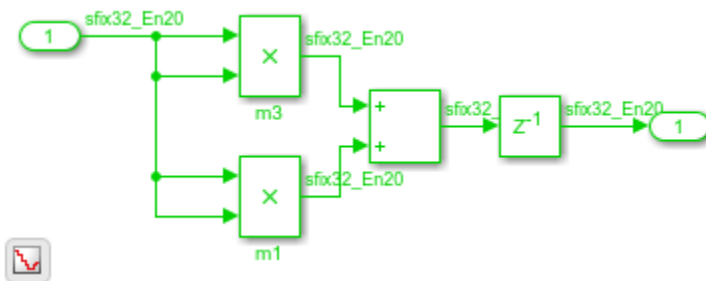
For an example, open the model `hdlcoder_axi_multirate_sharing`

```
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');
hdlset_param('hdlcoder_axi_multirate_sharing', 'ReferenceDesign', 'Default system with AXI4-Stream');
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.

See Also

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 41-2
- “Custom IP Core Generation” on page 40-2

Board and Reference Design Registration System

In this section...

“Board, IP Core, and Reference Design Definitions” on page 41-33

“Board Registration Files” on page 41-34

“Reference Design Registration Files” on page 41-35

“Predefined Board and Reference Design Examples” on page 41-36

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.

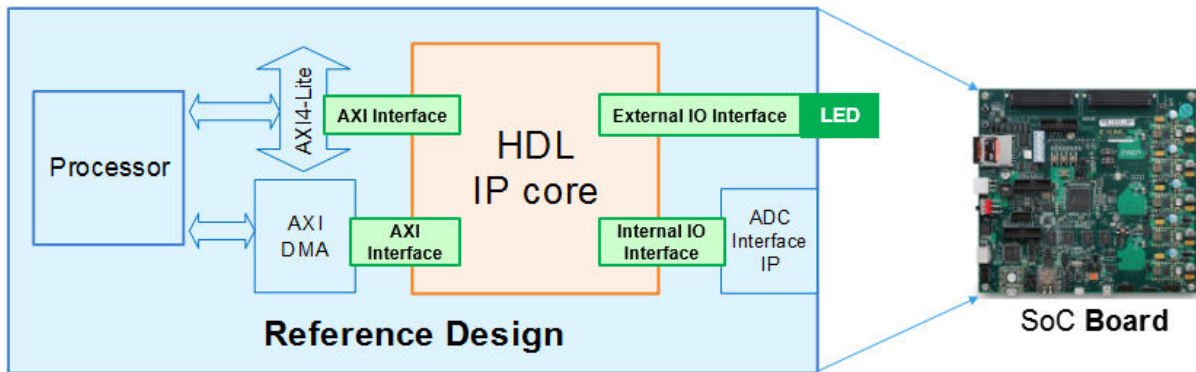
Board, IP Core, and Reference Design Definitions

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** dropdown 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** dropdown list.

Predefined Board and Reference Design Examples

For examples of working board and reference design definitions, refer to the predefined Altera SoC and Xilinx Zynq board plugins that include predefined reference design plugins:

- *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/alterasoc/+AlteraCycloneV/*
- *support_package_installation_folder/toolbox/hdlcoder/supportpackages/alterasoc/+ArrowSoCKit/*

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Register a Custom Board” on page 41-37
- “Register a Custom Reference Design” on page 41-41
- Define and Register Custom Board and Reference Design for SoC Workflow

Register a Custom Board

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.

In this section...

“Define a Board” on page 41-37

“Create a Board Plugin” on page 41-38

“Define a Board Registration Function” on page 41-39

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.FGPAPackage  = '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 41-41
- Define and Register Custom Board and Reference Design for SoC Workflow

More About

- “Board and Reference Design Registration System” on page 41-33

Register a Custom Reference Design

In this section...

“Define a Reference Design” on page 41-41

“Create a Reference Design Plugin” on page 41-42

“Define a Reference Design Registration Function” on page 41-43

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 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 (Vivado 2014.2)';
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 = {'2015.4'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'design_led.tcl');

hRD.CustomFiles = {'ZYB0_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 41-45.

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.Vivado2015_4.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 41-37
- “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 41-45
- Define and Register Custom Board and Reference Design for SoC Workflow

More About

- “Board and Reference Design Registration System” on page 41-33

Define Custom Parameters and Callback Functions for Custom Reference Design

In this section...

“Define Custom Parameters and Register Callback Function Handle” on page 41-45

“Define Custom Callback Functions” on page 41-51

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 = {'2018.3'};

%% 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 JTAG MATLAB as AXI Master IP Insertion. The IP
% insertion setting is visible in the 'Set Target Reference Design'
% task of the HDL Workflow Advisor. By default, the
% AddJTAGMATLABasAXIMasterParameter property is set to 'true'.
hRD.AddJTAGMATLABasAXIMasterParameter = 'true';
hRD.JTAGMATLABasAXIMasterDefaultValue = 'on';

%% 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 '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 JTAG MATLAB as AXI Master IP

By default, HDL Coder adds a parameter **Insert JTAG MATLAB as AXI Master (HDL Verifier Required)** to all reference designs. When you set this parameter to `on`, the code generator automatically inserts the JTAG MATLAB AXI Master IP into your reference design. By using the JTAG MATLAB AXI Master IP, you can easily access the AXI registers in the generated DUT IP core on an FPGA board from MATLAB through the JTAG connection. See also “Set Up for MATLAB AXI Master” (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).
- Do not target standalone boards that do not have the `hrd.addAXI4SlaveInterface` or boards that are based on Xilinx ISE.

The code generator adjusts the **AXI4 Slave ID Width** to accommodate the MATLAB as AXI Master 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 JTAG MATLAB AXI Master IP inserted in the reference design.

In the previous example code, the reference design defines the `AddJTAGMATLABasAXIMasterParameter` and `JTAGMATLABasAXIMasterDefaultValue` properties of the `hdlcoder.ReferenceDesign` class. These properties control the default behavior of the **Insert JTAG MATLAB as AXI Master (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 JTAG MATLAB as AXI Master (HDL Verifier Required)** parameter is displayed in the **Set Target Reference Design** task and the value is set to `off`. This example code illustrates the default behavior.

```
%% Default behavior of Insert JTAG as AXI Master option
```

```
% This parameter controls visibility of the option in
```

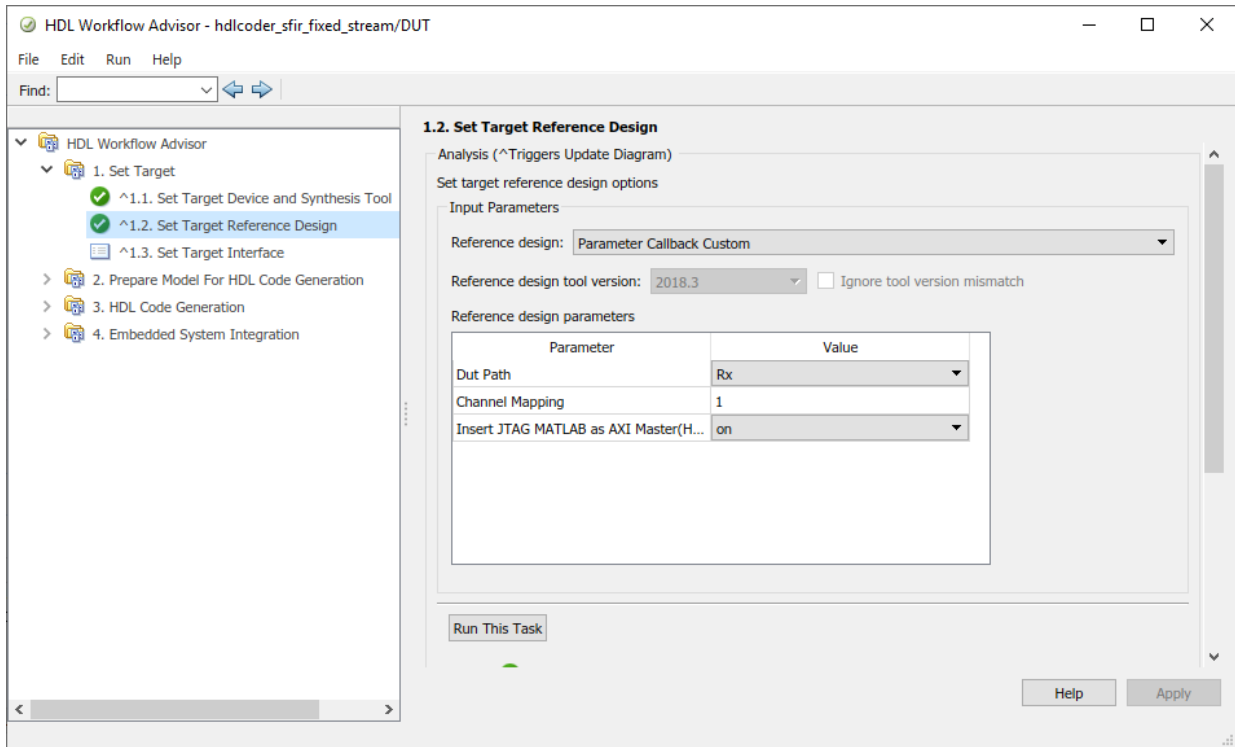
```
% 'Set Target Reference Design Task' of HDL Workflow Advisor
% By default, the parameter value is set to 'true',
% which means that the option is displayed in the UI. If
% you do not want the parameter to be displayed, set this
% value to 'false'.
hRD.AddJTAGMATLABasAXIMasterParameter = 'true';

% This parameter controls the value of the option in the
% the 'Set Target Reference Design Task' task. By default,
% the value is 'off', which means that the parameter is
% displayed in the task and the value is off. To enable
% automatic insertion of JTAG AXI Master IP in the reference
% design, set this value to 'on'. In that case, the
% AddJTAGMATLABasAXIMasterParameter must be set to 'true'.
hRD.JTAGMATLABasAXIMasterDefaultValue = 'off';
```

For an example, see Using JTAG MATLAB as AXI Master to control the HDL Coder IP Core.

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 JTAG MATLAB as AXI Master (HDL Verifier Required)** option set to on.



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 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	PostTargetReferenceDesignFcn	Enable custom validations. For an example that shows how you can validate that the Reset type is Synchronous, see <code>PostTargetReferenceDesignFcn</code> .
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 <code>PostTargetInterfaceFcn</code> .
Create Project	PostCreateProjectFcn	Specify custom settings when HDL Coder creates the project. For an example, see <code>PostCreateProjectFcn</code> .
Generate Software Interface Model	PostSWInterfaceFcn	Change the generated software interface model. For an example, see <code>PostSWInterfaceFcn</code> .
Build FPGA Bitstream	PostBuildBitstreamFcn	Specify custom settings when you build the FPGA bitstream. For an example, see <code>PostBuildBitstreamFcn</code> .

Workflow Advisor Task	Callback Property Name	Functionality
Program Target Device	CallbackCustomProgrammingMethod	Specify a custom FPGA programming method. For an example, see <code>CallbackCustomProgrammingMethod</code> .

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 41-37
- “Register a Custom Reference Design” on page 41-41
- Define and Register Custom Board and Reference Design for SoC Workflow

More About

- “Board and Reference Design Registration System” on page 41-33

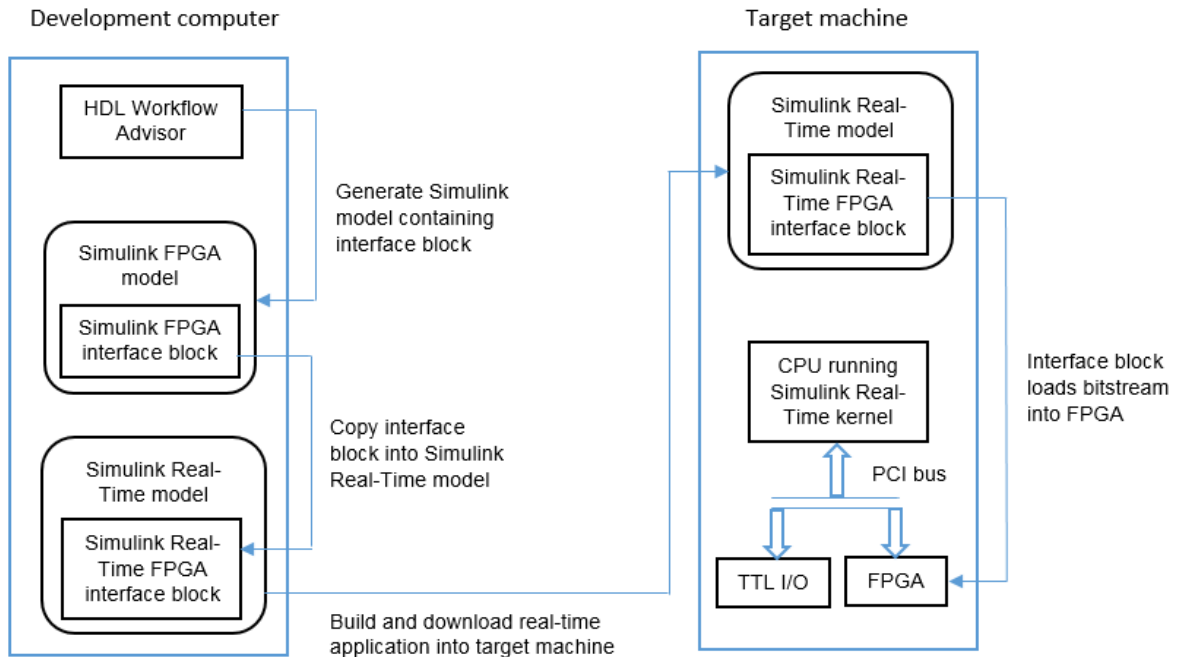
FPGA Programming and Configuration

This example shows how to implement a Simulink® algorithm on a Speedgoat FPGA I/O board by using HDL Workflow Advisor to:

- 1 Specify an FPGA board and its I/O interface.
- 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 I/O board during real-time application execution. You add the generated subsystem to your Simulink Real-Time domain model.

The entire workflow looks like this figure.



This example uses the Speedgoat IO331. You can use any FPGA I/O module supported by Simulink Real-Time and HDL Coder that meets the speed, size, and pinout requirements of the model.

Requirements and Preconditions

HDL Coder™

Before you start, complete an FPGA subsystem plan.

For the IO331 board, HDL Workflow Advisor requires the Xilinx® ISE toolset. To install this toolset, in the Command Window, type:

```
hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', toolpath)
```

where *toolpath* is the full path to the synthesis tool executable.

For the toolset requirements of other boards, see Supported Third-Party Tools and Hardware (HDL Coder).

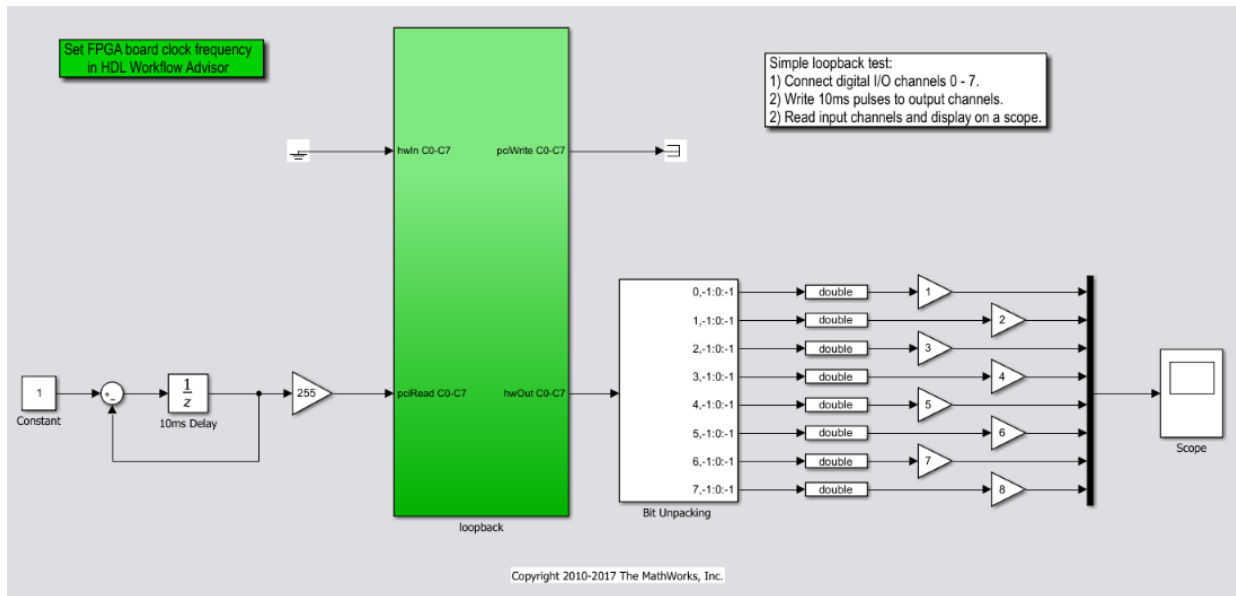
Step 1. Simulink Domain Model

The Simulink FPGA domain model contains a subsystem (algorithm) to be programmed onto the FPGA chip. Using this model, you can test your FPGA algorithm in a simulation environment before you download the algorithm to an FPGA board.

- 1 Create a Simulink model that contains the algorithm that you want to load onto the FPGA, in this case a loopback test.
- 2 Place the algorithm to be programmed on the FPGA inside a Subsystem block. The model can include other blocks and subsystems for testing. However, one subsystem must contain the FPGA algorithm.
- 3 Set or confirm the subsystem inport and outport names and data types. The HDL Coder HDL Workflow Advisor uses these settings for routing and mapping algorithm signals to I/O connector channels.
- 4 Save the model.

This model is your FPGA domain model. It represents the simulation sample rate of the clock on your FPGA board. For example, the Speedgoat IO331 has an onboard 125 MHz clock. One second of simulation equals 125e6 iterations of the model.

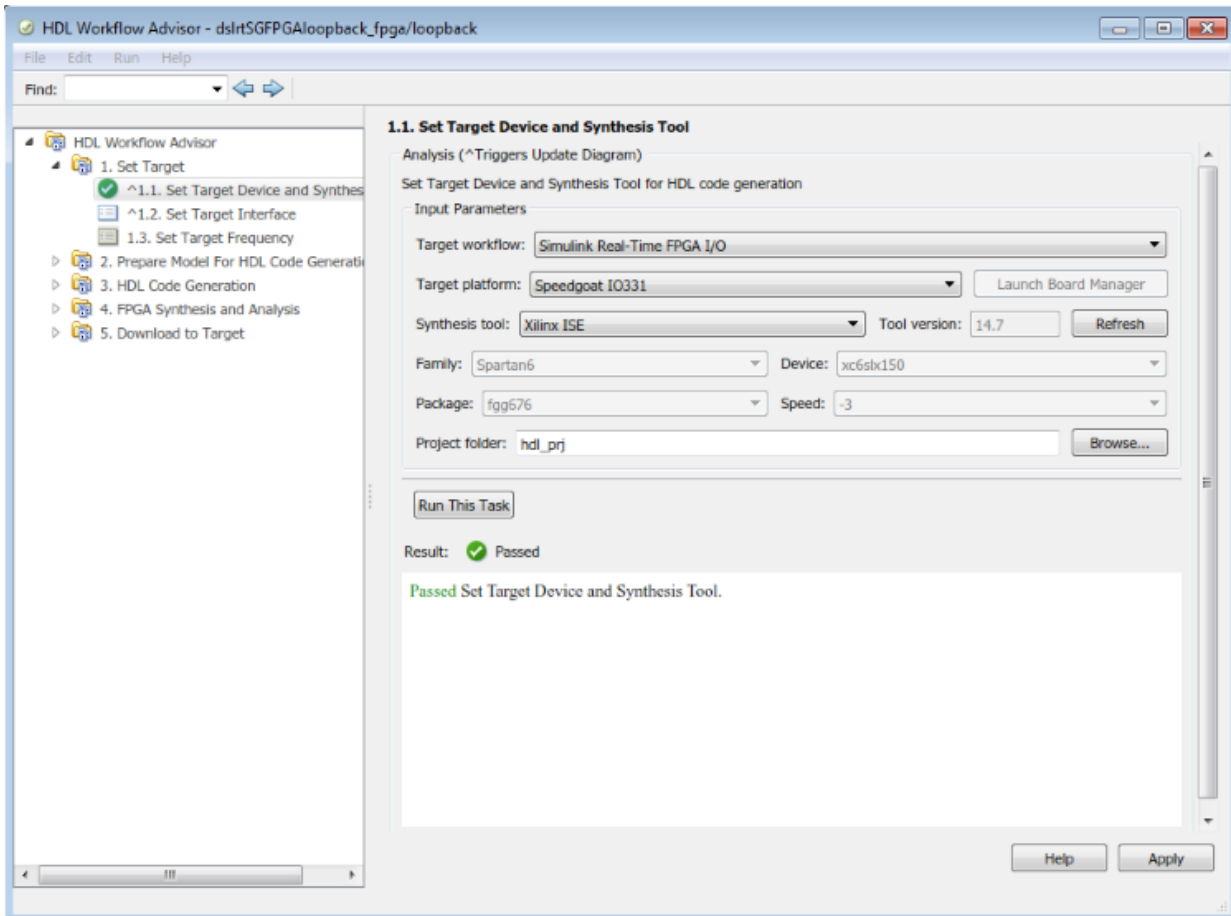
For an example of an FPGA domain model, see `ds1rtSGFPGAloopback_fpga`. The `ServoSystem` subsystem contains the FPGA algorithm.



Step 2. FPGA Target Configuration

This procedure uses the `dsLrtSGFPGAloopback_fpga` example. You must have already created an FPGA subsystem (algorithm) in an FPGA domain model and developed an FPGA subsystem plan.

- 1 Open the FPGA domain model `dsLrtSGFPGAloopback_fpga`.
- 2 In the FPGA model, right-click the FPGA subsystem (ServoSystem). From the context menu, select **HDL Code > HDL Workflow Advisor**. The HDL Workflow Advisor dialog box displays several tasks for the subsystem. Address only your required subset of the tasks.
- 3 Expand the **Set Target** folder and select task **1.1 Set Target Device and Synthesis Tool**.
- 4 Set **Target Workflow** to Simulink Real-Time FPGA I/O.
- 5 From the **Target platform** list, select the Speedgoat FPGA I/O board installed in your Speedgoat target machine, in this case the Speedgoat I0331. Check that HDL Workflow advisor sets the synthesis tool to the Xilinx® ISE Design Suite.
- 6 Click **Run This Task**.

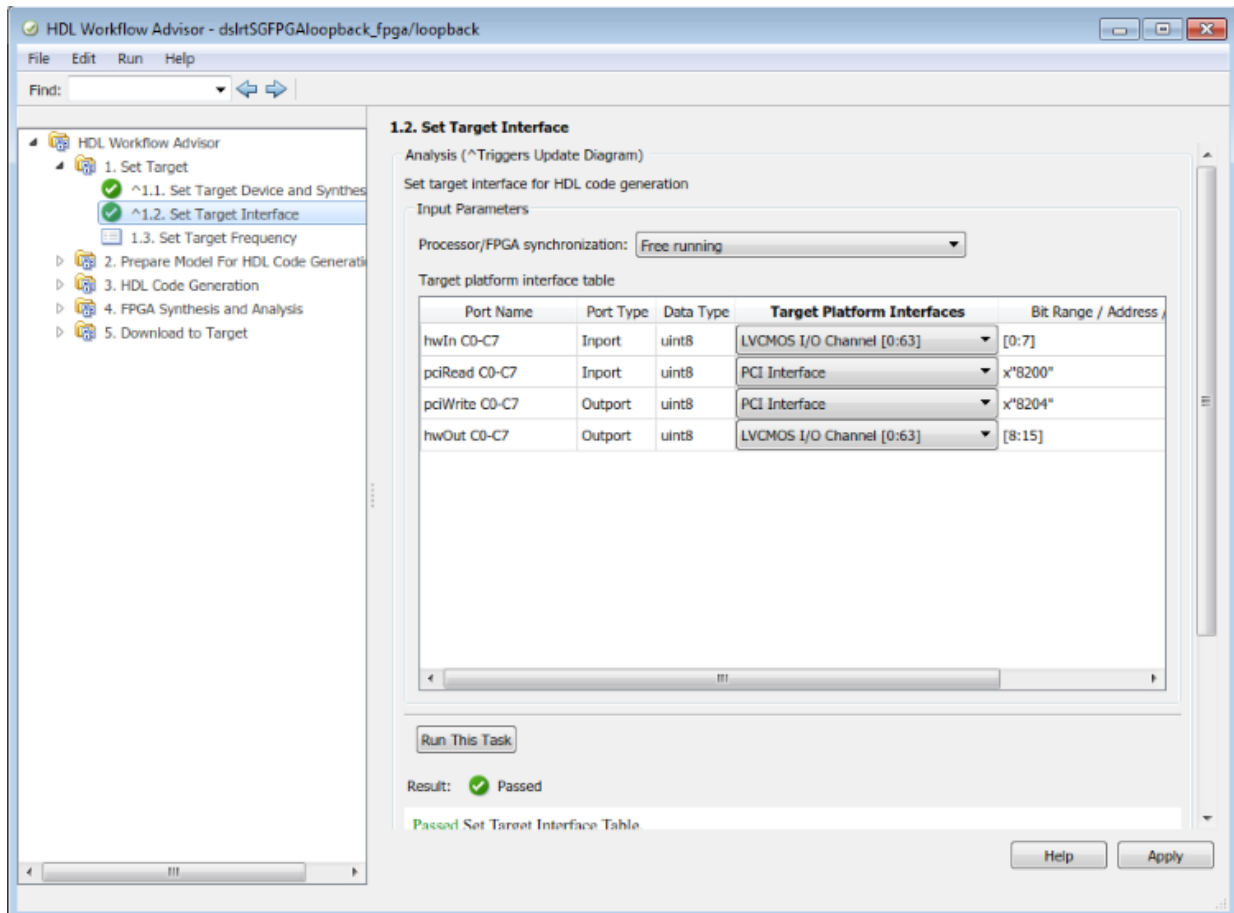


Step 3. FPGA Target Interface Configuration

You must have already configured the FPGA target.

- 1 In the **Set Target** folder, select task **1.2 Set Target Interface**.
- 2 In the **Processor/FPGA synchronization** box, select Free running.
- 3 For signals hwIn and hwOut, in the **Target Platform Interfaces** column, select LVCMOS I/O Channel [0:63]. In the **Bit Range/Address/FPGA Pin** column, enter the channel value for each signal, or take the defaults.

- 4 For signals `pciRead` and `pciWrite`, in the **Target Platform Interfaces** column, select **PCI Interface**. In the **Bit Range/Address/FPGA Pin** column, use the automatically generated values. Do not enter PCI address values.
- 5 Click **Run This Task**.



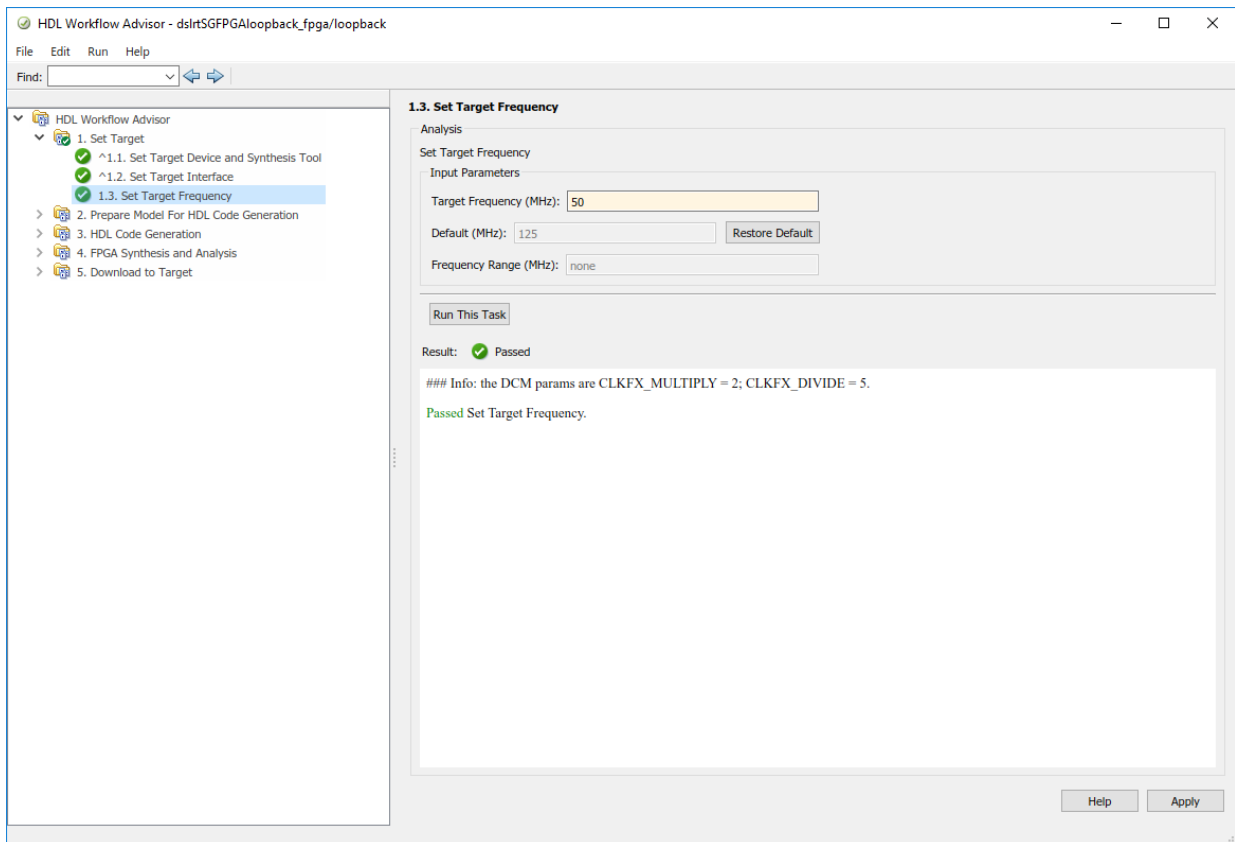
Step 4. FPGA Target Frequency Configuration

You must have already configured the FPGA target interface.

- 1 In the **Set Target** folder, select task **1.3 Set Target Frequency** (optional). The **Set Target Frequency** pane contains fields showing the FPGA input clock frequency

(fixed) and the FPGA system clock frequency. The FPGA system clock frequency defaults to the FPGA input clock frequency.

- 2 To specify a different system clock frequency (for example, 50 MHz), type the new value in the field **FPGA system clock frequency (MHz)**. For the permitted range for the system clock rate, see the Speedgoat board characteristics table. The system sometimes sets a value different from the one you specified.
- 3 Click **Run This Task**.



Step 5. Simulink Real-Time Interface Subsystem Generation

This procedure generates an interface subsystem file for the `dxpcSGFPGAloopback_fpga` example.

Assign distinct names to blocks that contain different HDL code. The name of the interface subsystem file is derived directly from the block name. If two blocks containing different HDL code have the same name, the names collide and one of the blocks gets the wrong code.

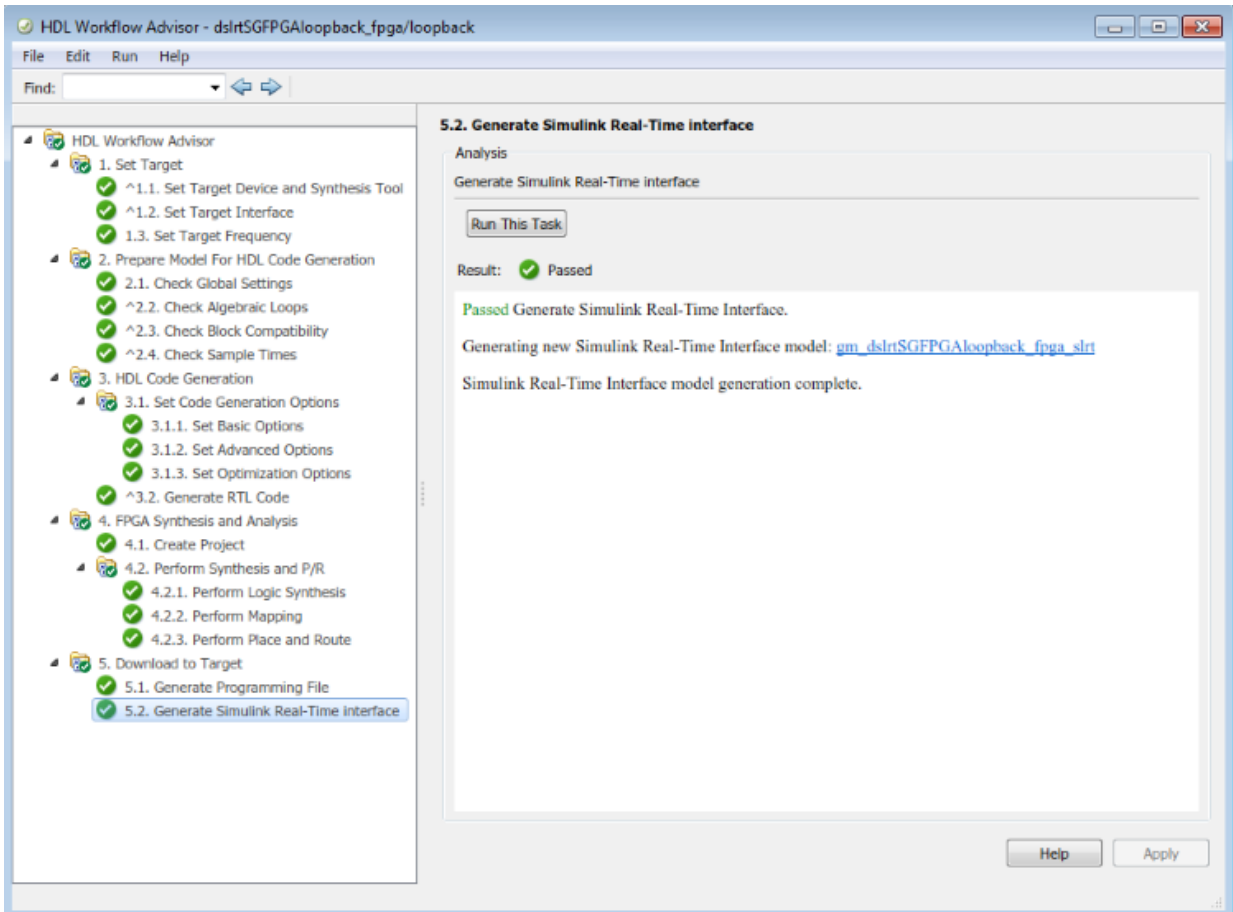
You must have already configured the FPGA target interface and the required target frequency. If you have specified vector inports or outports, you must have already selected the **Scalarize vector ports** check box. This check box is on the **Coding style** tab of node **Global Settings**, under node **HDL Code Generation** in the Configuration Parameters dialog box.

- 1 Expand the **Download to Target** folder, and right-click task **5.2 Generate Simulink Real-Time Interface**.
- 2 In this pane, click **Run To Selected Task**.

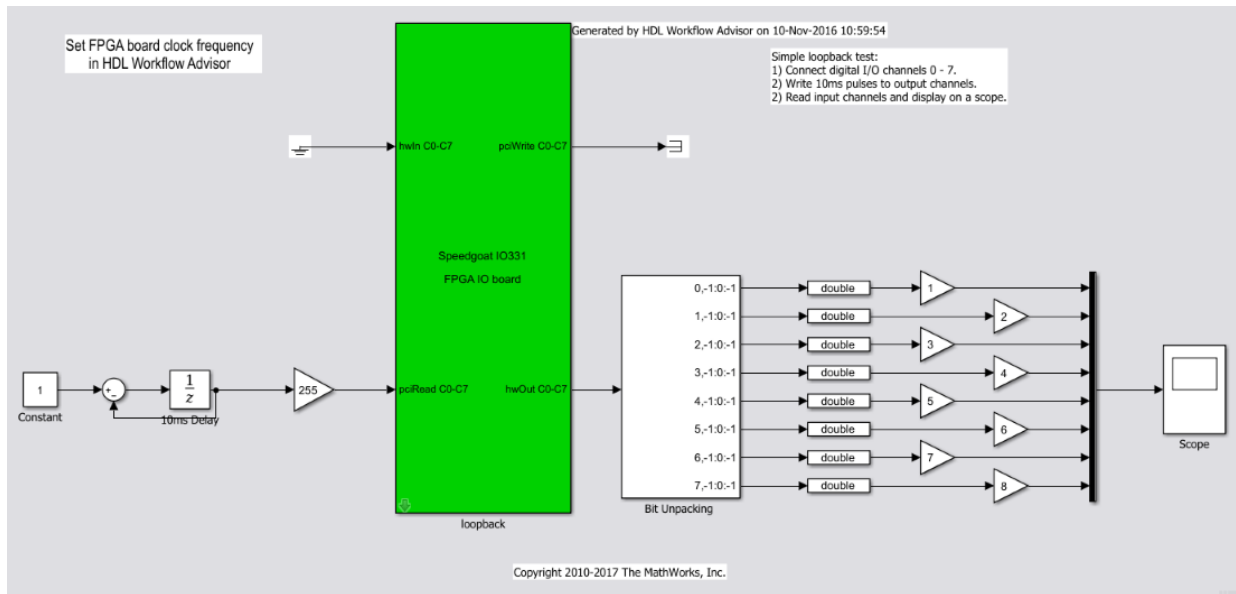
This action:

- Runs the remaining tasks.
- Creates the FPGA bitstream file in the `hdlsrc` folder. The Simulink Real-Time interface subsystem references this bitstream file during the build and download process.
- Generates a model named `gm_dslrtSGFPGAloopback_fpga_slrt`, which contains the Simulink Real-Time interface subsystem.

Here is an example of the HDL Coder HDL Workflow Advisor after this action.



The generated interface subsystem looks like this figure.



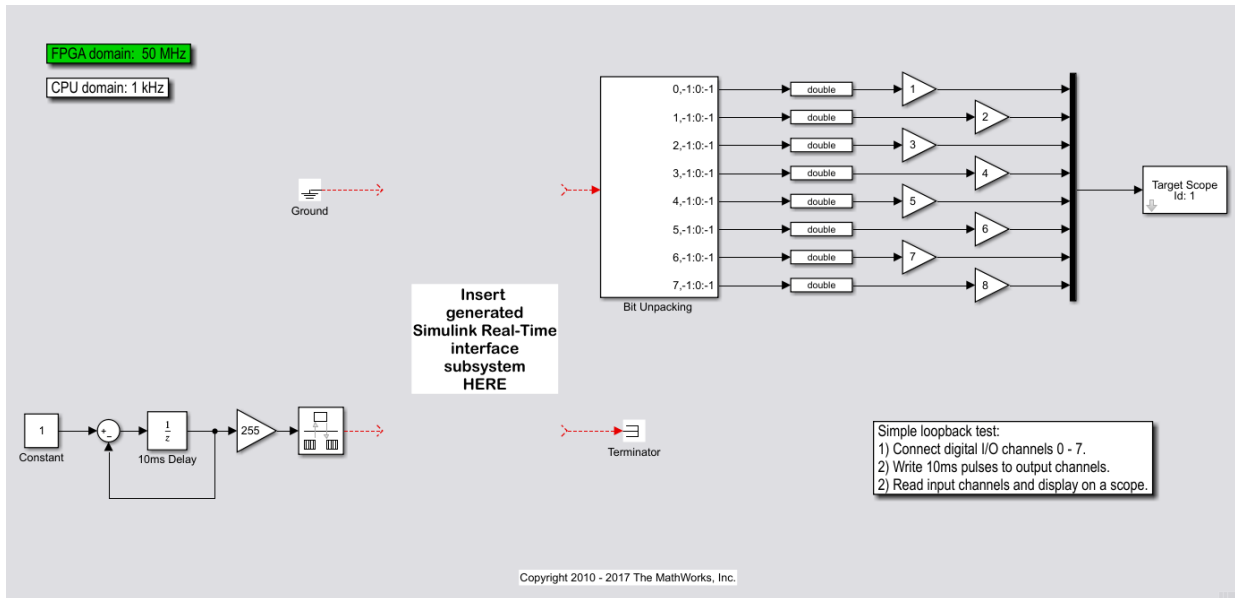
This generated model contains a masked subsystem with the same name as the subsystem in the Simulink FPGA domain model. Although the appearance is similar, this subsystem does not contain the Simulink algorithm. Instead, the algorithm is implemented in an FPGA bitstream. You reference and load this algorithm into the FPGA from this subsystem.

Step 6. Simulink Real-Time Domain Model

Using the Simulink Real-Time software, transform a Simulink or Stateflow® domain model into a Simulink Real-Time domain model and execute it on a Speedgoat target machine for real-time testing applications. After creating a Speedgoat FPGA interface subsystem. You can then include the FPGA board in your Simulink Real-Time domain model by inserting the interface subsystem.

- 1 Create a Simulink Real-Time domain model with the functionality that you want to simulate with the FPGA algorithm. Leave the inports and outports of the FPGA subsystem disconnected.
- 2 Save the model.

The Simulink Real-Time domain model looks like this figure. See example model `dslrtSGFPGAloopback_slrt`.



Step 7. Simulink Real-Time Interface Subsystem Integration

In the Simulink Real-Time interface subsystem mask, set three parameters:

- Device index
- PCI slot
- Sample time

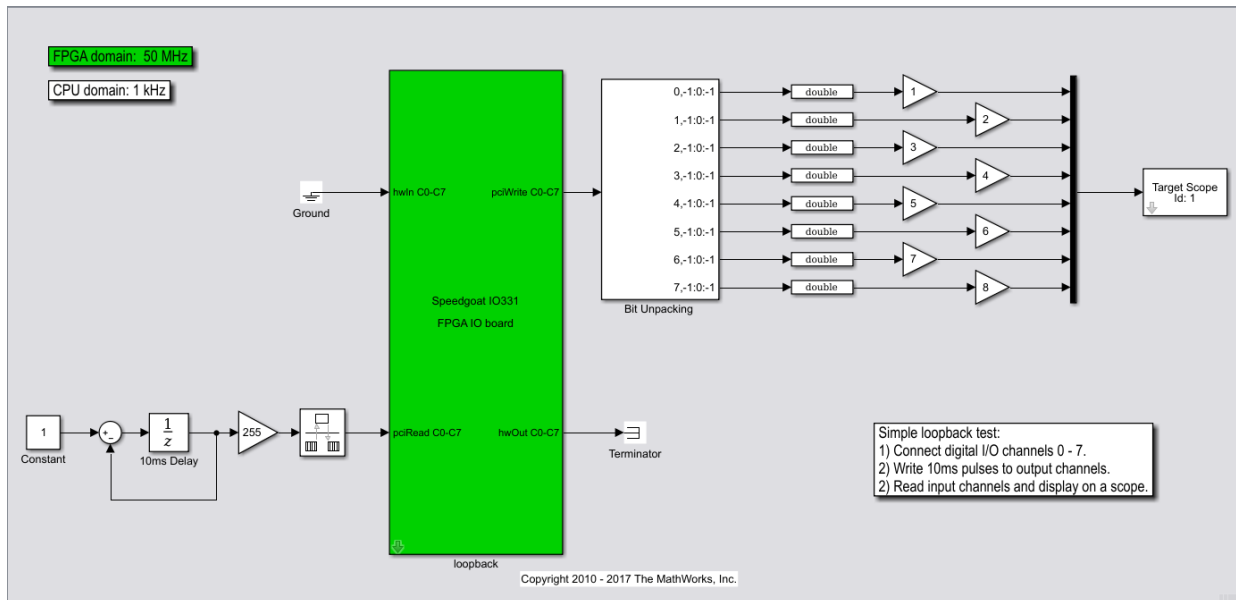
To integrate the interface subsystem:

- 1 In the Simulink editor, open `gm_dslrtSGFPGAloopback_fpga_slrt`.
- 2 Copy the Simulink Real-Time interface subsystem and paste it into the Simulink Real-Time domain model.
- 3 Save or discard `gm_dslrtSGFPGAloopback_fpga_slrt`. You can recreate it as required using the HDL Coder HDL Workflow Advisor.
- 4 In the domain model, connect signals to the inports and outports of the interface subsystem.
- 5 Set the block parameters according to the FPGA I/O boards in your Speedgoat target machine.

- If you have a single FPGA I/O board, leave the device index and PCI slot at the default values. You can set the sample time or leave it at -1 for inheritance.
- If you have multiple FPGA I/O boards, give each board a unique device index.
- If you have two or more boards of the same type (for example, two Speedgoat IO331 boards), specify the PCI slot ([bus, slot]) for each board. Get this information with the `SimulinkRealTime.target.getPCIInfo` function.

6 Save the model.

The updated Simulink Real-Time domain model looks like this figure. See example model `dslrtSGFPGAloopback_slrt_wiss`.



Step 8. Real-Time Application Execution

To do this procedure, you must have already created a Simulink Real-Time domain model that includes a Simulink Real-Time interface subsystem generated from the HDL Coder HDL Workflow Advisor.

- 1 Configure the Speedgoat target machine and connect it to the development computer.

- 2 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.
- 3 If you are using I/O lines (channels), confirm that you have connected the lines to the external hardware under test.

The start and stop of the Simulink Real-Time model controls the start and stop of the FPGA algorithm. The FPGA algorithm executes at the clock frequency of the FPGA I/ O board, while the real-time application executes in accordance with the model sample time.

See Also

Related Examples

- “Processor and FPGA Synchronization” on page 40-18
- “IP Core Generation Workflow for Speedgoat I/O Modules” on page 41-91
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 41-2

Model Design for AXI4-Stream Video Interface Generation

In this section...
“Streaming Pixel Protocol” on page 41-64
“Protocol Signals and Timing Diagrams” on page 41-65
“Model Data and Control Bus Signals” on page 41-67
“Model Designs with Multiple Sample Rates” on page 41-71
“Video Porch Insertion Logic” on page 41-72
“Default Video System Reference Design” on page 41-73
“Restrictions” on page 41-74

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.

Streaming Pixel Protocol

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:

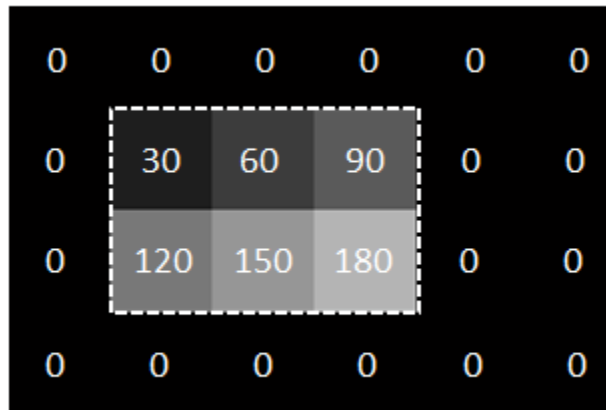
- hStart
- hEnd
- vStart
- vEnd
- valid

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.

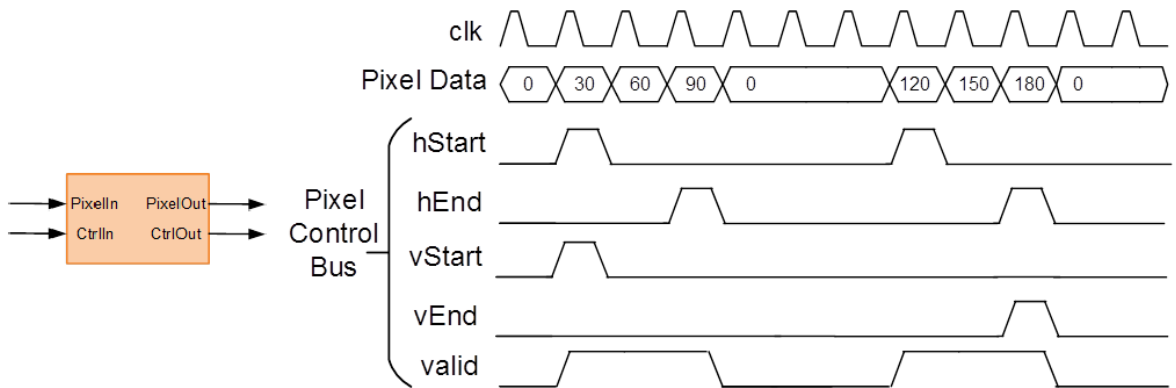
Protocol Signals and Timing Diagrams

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.



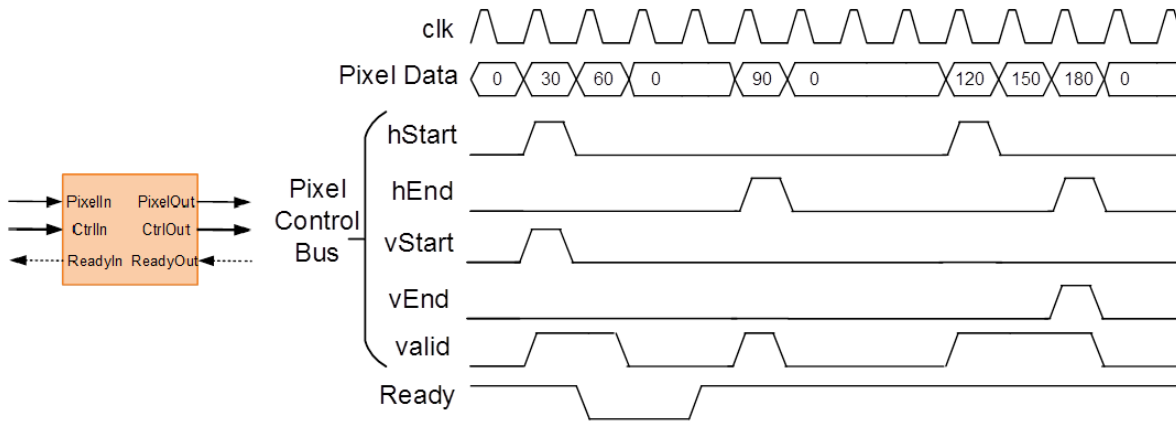
The **Pixel Data** signal is the primary video signal that is transferred across the AXI4-Stream Video interface. When the **Pixel Data** signal is valid, the **valid** signal is asserted.

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.

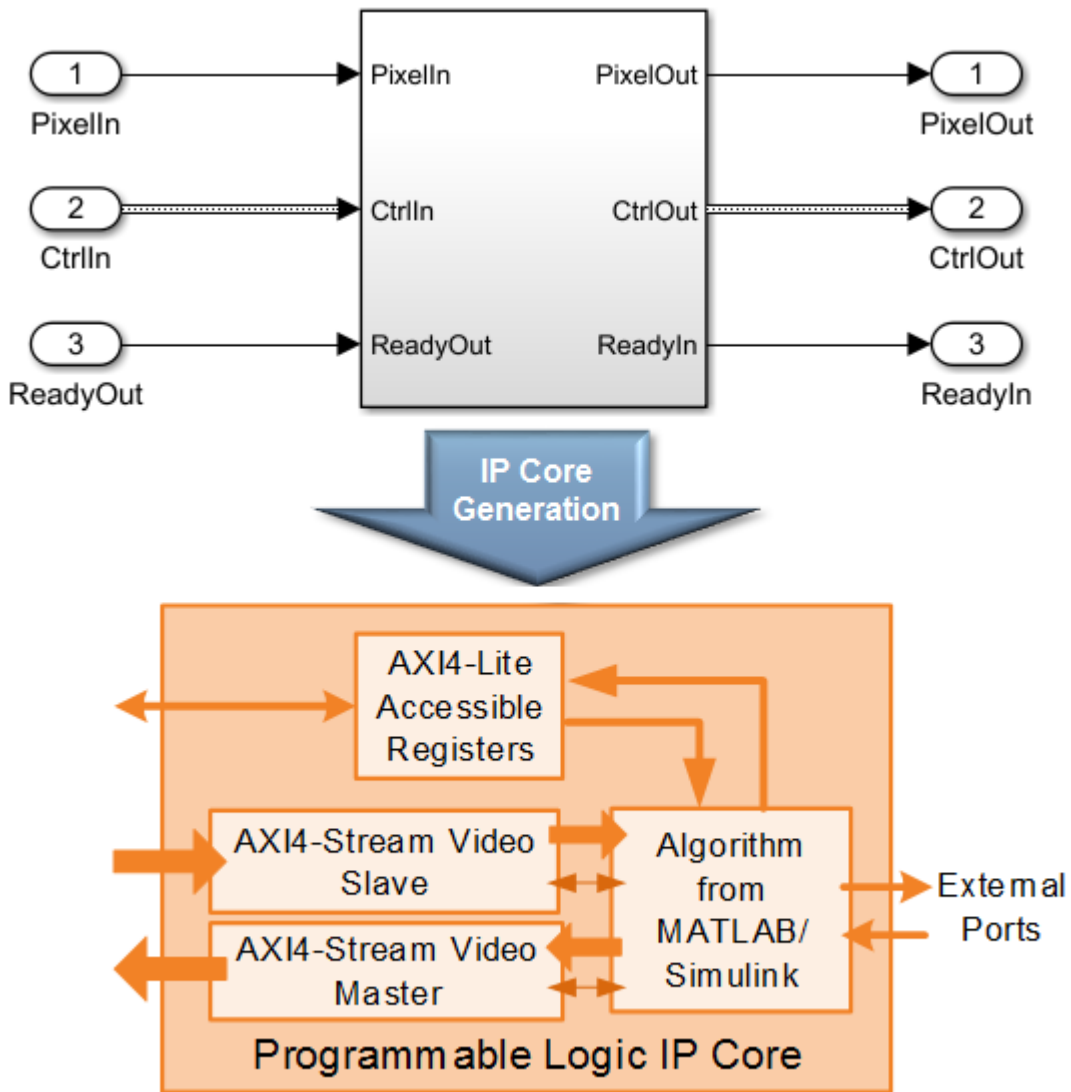
In a Slave interface, with the **Ready** signal, you can apply back pressure. In a Master interface, with the **Ready** signal, you can respond to back pressure.

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.

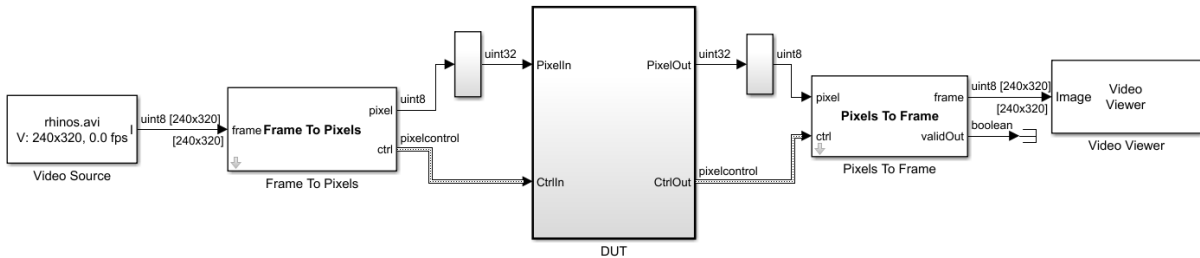
If you do not model the **Ready** signal, HDL Coder generates the associated backpressure logic.

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.

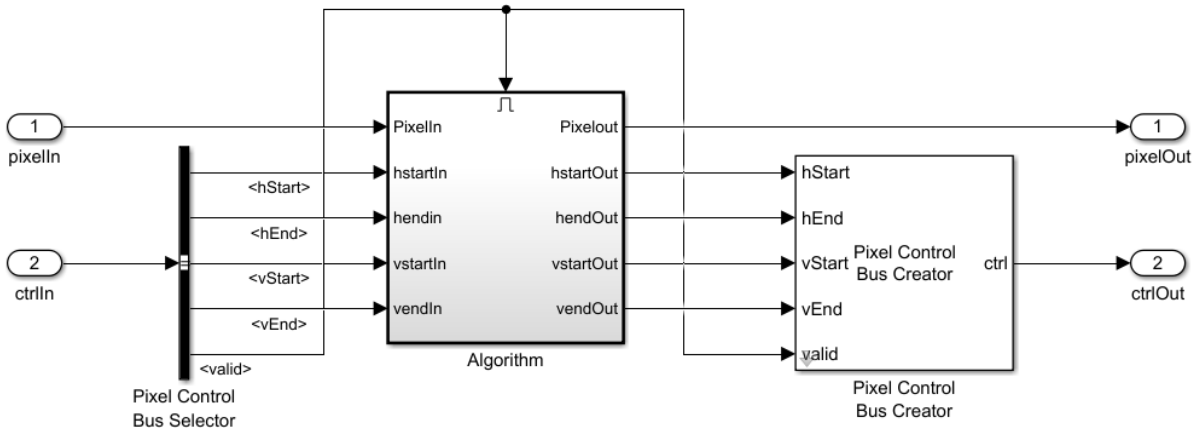


The Frame To Pixels and Pixels To Frame blocks perform the conversion between the video frames and the **Pixel Data** and **Pixel Control Bus** at the DUT interface. To use these blocks, you must have the Vision HDL Toolbox installed.

See also Frame To Pixels and Pixels To Frame in the Vision HDL Toolbox documentation.

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.



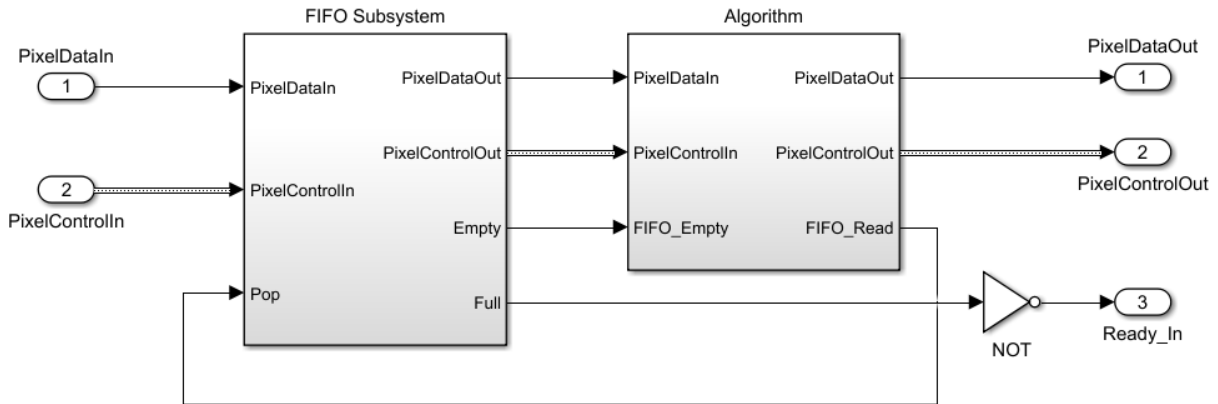
You can directly connect the **valid** signal from the **Pixel Control Bus** to the Enable port. If you do not have the Vision HDL Toolbox software, replace the Pixel Control Bus Selector and Pixel Control Bus Creator blocks with the Bus Selector and Bus Creator blocks respectively.

Ready Signal Modeling

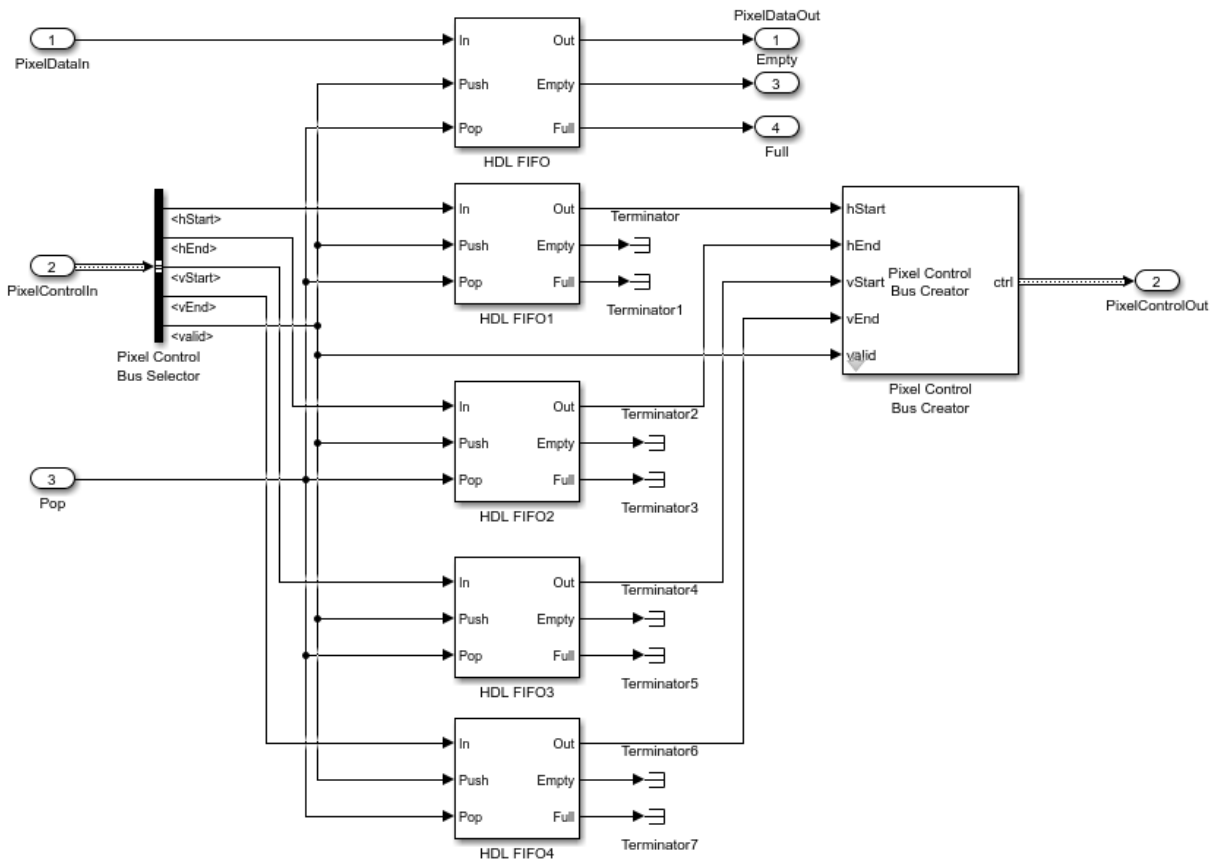
The AXI4-Stream Video interfaces in your DUT can optionally include a **Ready** signal.

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.

Model Designs with Multiple Sample Rates

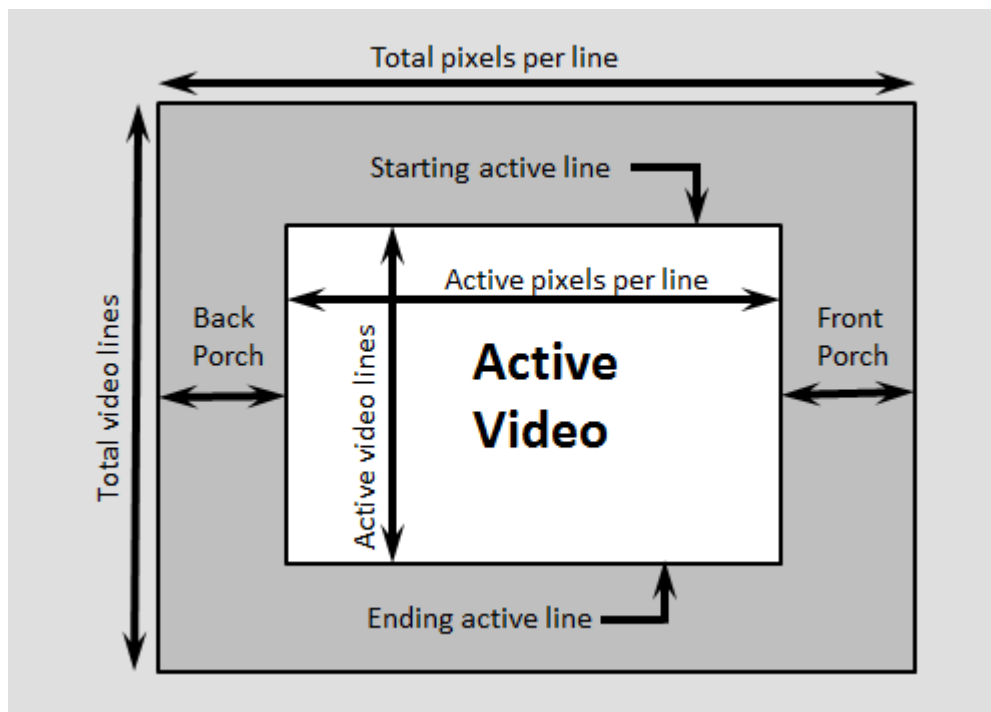
The HDL Coder software supports designs with multiple sample rates when you run the IP Core Generation workflow. When you map the interface ports to AXI4-Stream Video Master or AXI4-Stream Video Slave interfaces, to use multiple sample rates, ensure that the DUT ports that map to these AXI4 interfaces run at the fastest rate of the design after HDL code generation.

To learn more, see “Multirate IP Core Generation” on page 41-27.

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

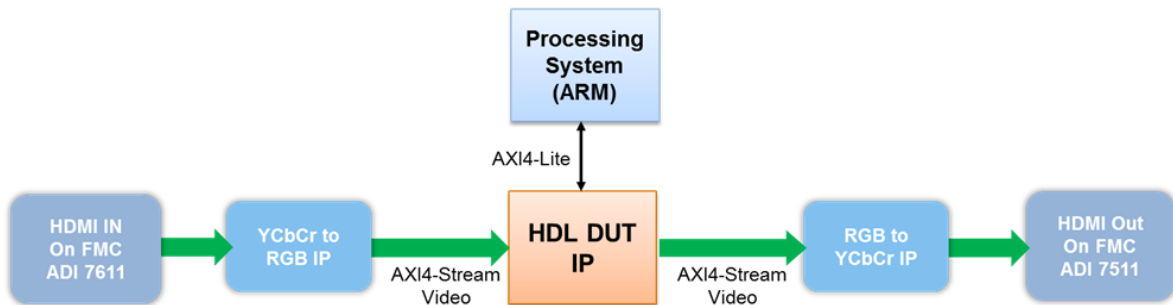
pixel stream. By using the AXI4-Lite registers in the generated IP core, you can customize these porch parameters for each video frame:

- 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 Computer Vision Toolbox™ Support Package for Xilinx Zynq-Based Hardware.

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.
- The DUT can have at most one AXI4-Stream Video Master channel and one AXI4-Stream Video Slave channel.
- Xilinx Zynq-7000 must be your target platform.
- You must use Xilinx Vivado as your synthesis tool.
- **Processor/FPGA synchronization** must be Free running.

See Also

More About

- “Model Design for AXI4-Stream Interface Generation” on page 41-19
- “Streaming Pixel Interface” (Vision HDL Toolbox)

Model Design for AXI4 Master Interface Generation

In this section...

“Simplified AXI4 Master Protocol - Write Channel” on page 41-75

“Simplified AXI4 Master Protocol - Read Channel” on page 41-77

“Base Address Register Calculation” on page 41-78

“Modeling for AXI4 Master Interfaces” on page 41-80

“Model Designs with Multiple Sample Rates” on page 41-83

“Reference Designs for IP Core Integration” on page 41-83

“Restrictions” on page 41-85

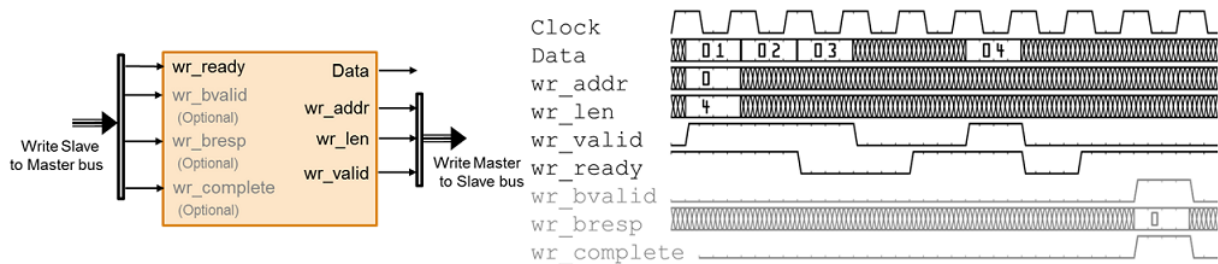
For designs that require accessing large data sets from an external memory, model your algorithm with a simplified AXI4 Master protocol. When you run the IP Core Generation workflow, HDL Coder generates an IP core with AXI4 Master interfaces. The AXI4 Master interface can communicate between your design and the external memory controller IP by using the AXI4 Master protocol. Use the AXI4 Master interface when your:

- Design targets multiframe 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 nonstreaming 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

You can use the simplified AXI4 Master protocol to map to AXI4 Master interfaces. Use the simplified AXI4 Master write protocol for a write transaction and the simplified AXI4 Master read protocol for a read transaction.

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 out 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`. `wr_addr` specifies the starting address that DUT wants to write to. The `wr_len` signal corresponds to the number of data elements in this write transaction. Data can be sent as long as `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.

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.
 - `wr_len`: The number of data values that you want to transfer, sampled at the first cycle of the transaction.
 - `wr_valid`: When this control signal becomes high, it indicates that the Data signal sampled at the output is valid.

Input Signals

Model the Write Slave to Master bus that consists of:

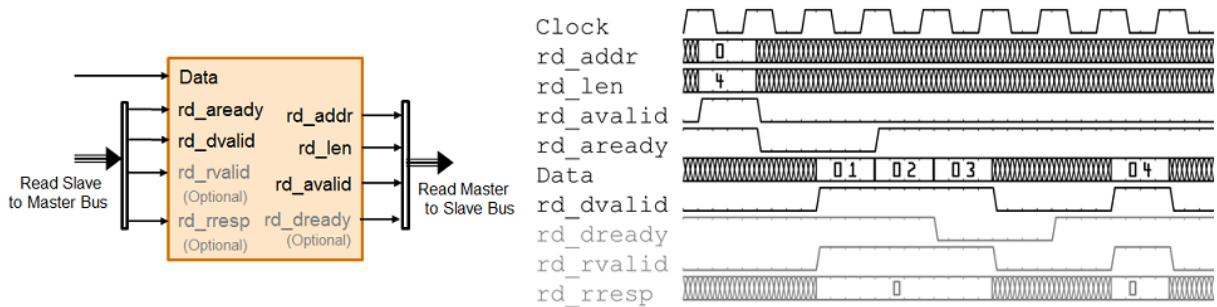
- `wr_ready`: This signal corresponds to the backpressure from the slave IP core or external memory. When this control signal goes high, it indicates that data can be sent. When `wr_ready` is low, the DUT must stop sending data within one clock cycle. You can also use the `wr_ready` signal to determine 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.

- `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. `wr_bvalid` 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_complete` (optional signal): Control signal that when remains high for one clock cycle indicates that the write transaction has completed. This signal asserts at the last `wr_bvalid` of the burst.

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.
- `rd_len`: The number of data values that you want to read, sampled at the first cycle of the transaction.
- `rd_avalid`: Control signal that specifies whether the read request is valid.

At the DUT input interface, implement the `rd_aready` signal. This signal is part of the `Read Slave to Master bus` and indicates when to accept read requests. You can monitor the `rd_aready` signal to determine whether the DUT can send consecutive burst requests. When `rd_aready` becomes high, it indicates that the DUT can send a read request in the next clock cycle.

Read Response

At the DUT input interface, model the `Data` and `Read Slave to Master bus` signals.

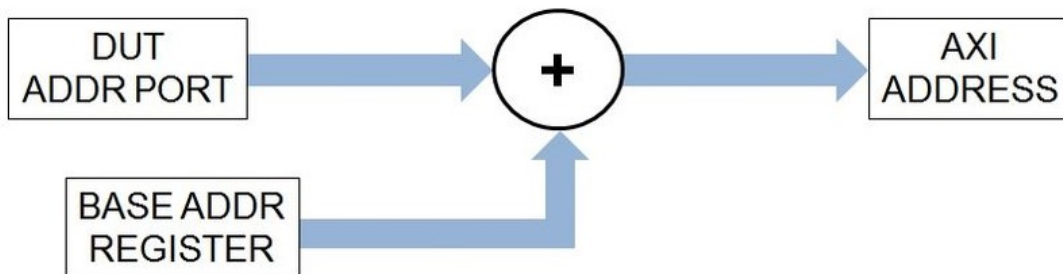
- `Data`: The data that is returned from the read request.
- `Read Slave to Master bus` that consists of:
 - `rd_dvalid`: Control signal which indicates that the `Data` returned from the read request is valid.
 - `rd_rvalid` (optional signal): response signal from the slave IP core that you can use for diagnosis purposes.
 - `rd_rresp` (optional signal): Response signal from the slave IP core that indicates the status of the read transaction.

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 `rd_dready` to logic high.

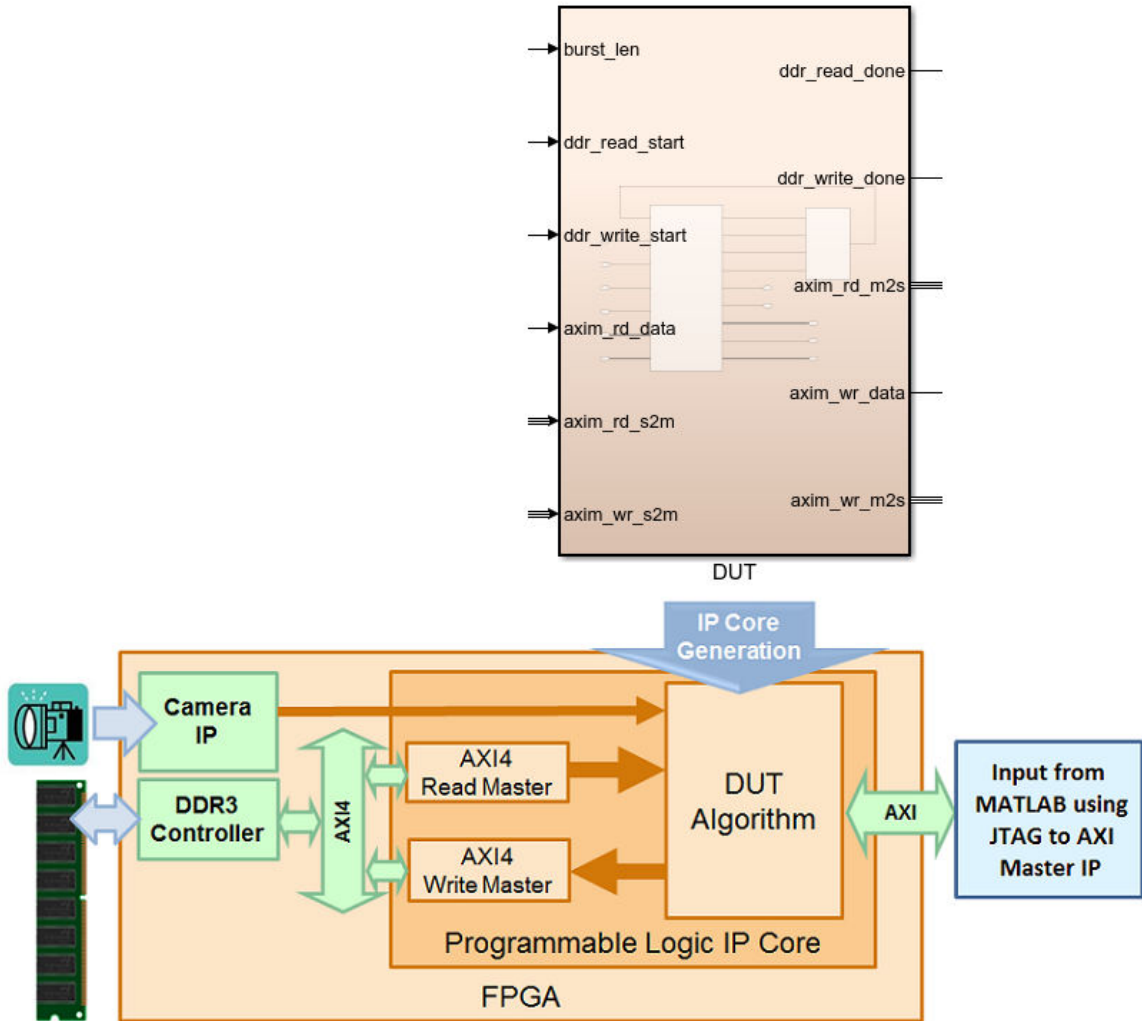
Base Address Register Calculation

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

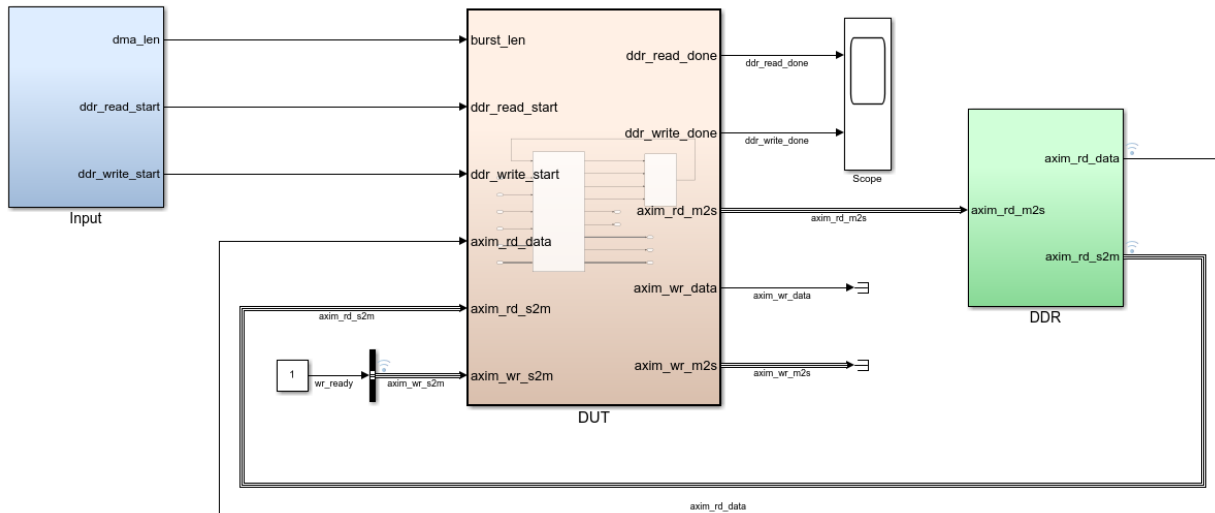
register is added to the address that is specified by the DUT ADDR port to form the AXI4 Master address. This capability enables the driver to use an addressing mode that programs a fixed register address with the base address of a buffer. The programmed address together with the DUT ADDR port is used to index the buffer. By default, the registers take a value of zero, if you do not use them.



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 back to a different address in the DDR memory.



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

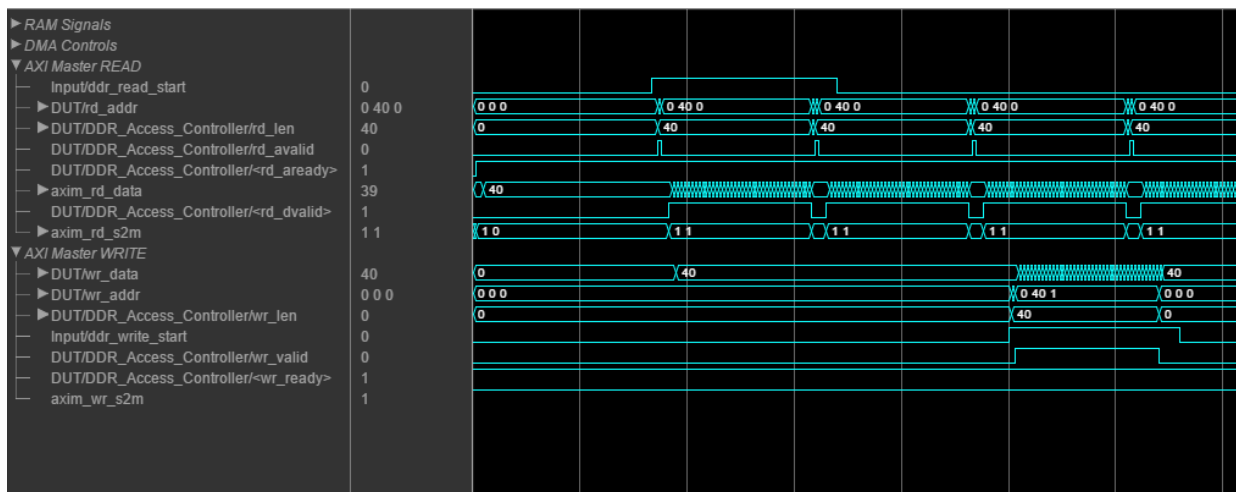
The `DDR Read Controller` is modeled as a state machine with four states: `INIT`, `IDLE`, `READ_BURST_START`, and `DATA_COUNT`. The `INIT` state initializes the read signals and the RAM input signals. When the start signal goes high, the state machine switches to the `IDLE` state, and then waits for the `rd_ready` signal to become high. When `rd_ready` 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.



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.

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.

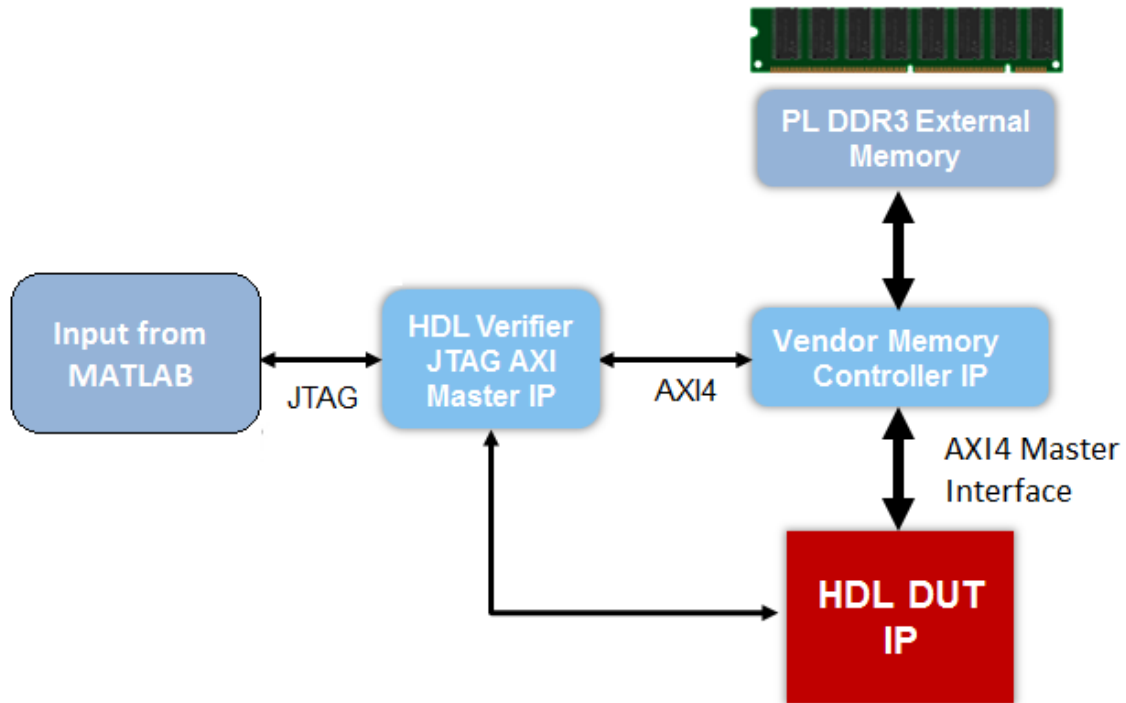
To learn more, see “Multirate IP Core Generation” on page 41-27.

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 Altera 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 Master IP that is provided by HDL Verifier. After you run the FPGA design on the board, using the JTAG AXI Master 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 Master 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.
- **Data type support:** For Data port, use scalar data types. Vector data types are not supported.

See Also

Related Examples

- “Performing Large Matrix Operation on FPGA using External Memory”

More About

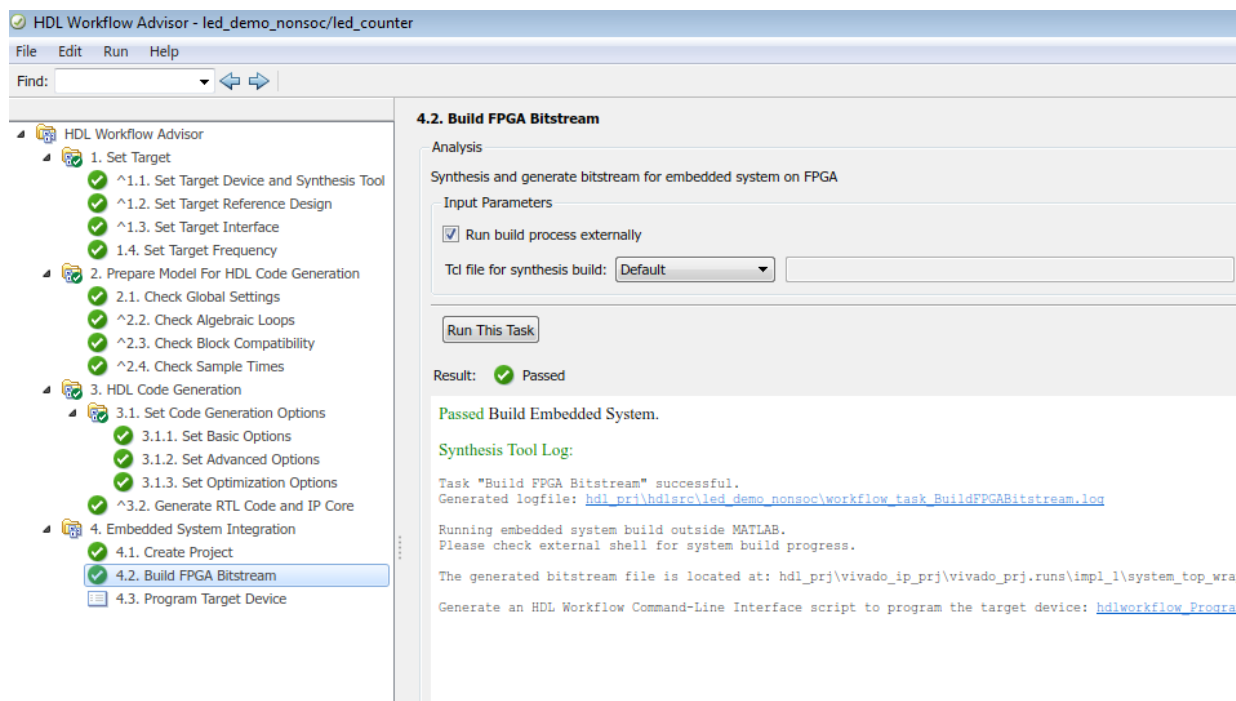
- “Model Design for AXI4-Stream Interface Generation” on page 41-19
- “Streaming Pixel Interface” (Vision HDL Toolbox)

IP Core Generation Workflow for Standalone FPGA Devices

In this section...
“Targeting FPGA Reference Designs with AXI4 Interface” on page 41-88
“Targeting FPGA Reference Designs Without AXI4 Interface” on page 41-89
“Board Support” on page 41-89

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 40-2.

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 does not have a **Generate Software Interface Model** task.



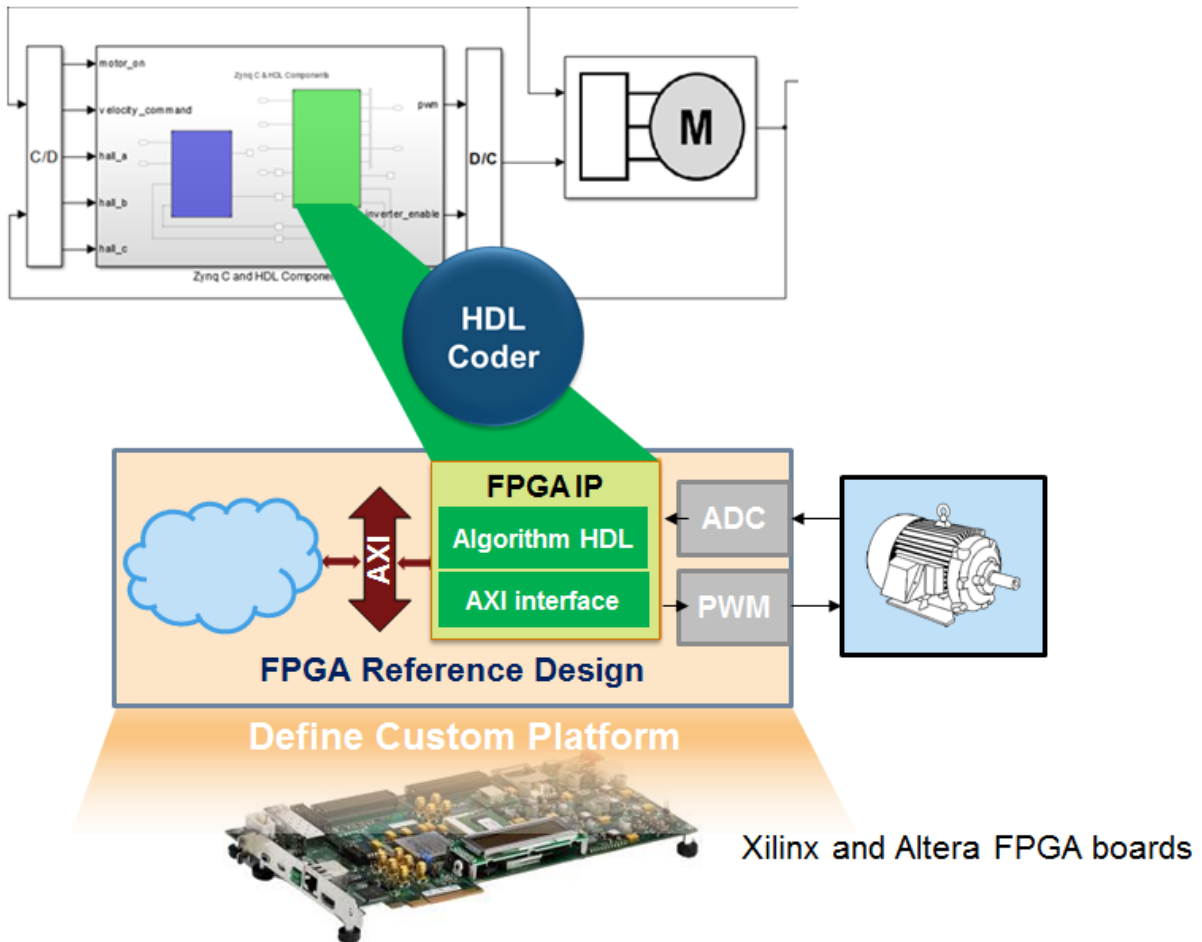
The workflow for the FPGA boards has these features:

- **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 41-33.



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.

See Also

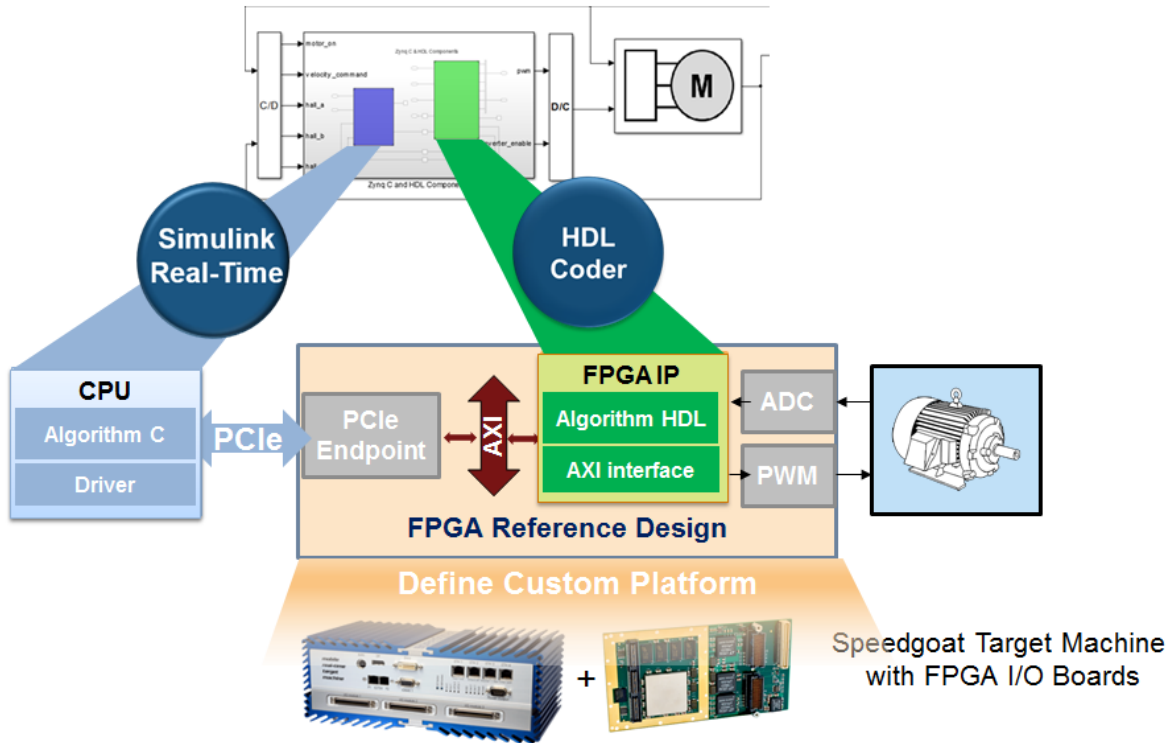
Related Examples

- “IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705”

IP Core Generation Workflow for Speedgoat I/O Modules

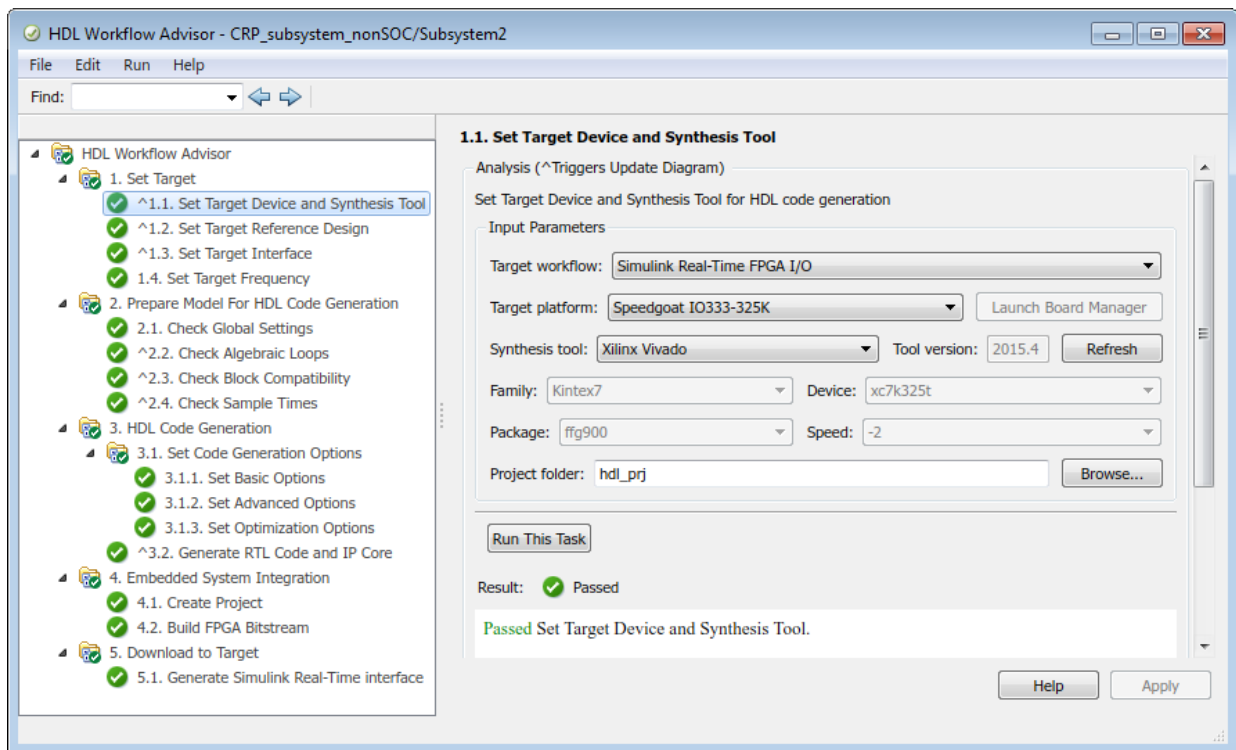
For the Speedgoat I/O modules that support Xilinx Vivado, HDL Coder uses the IP Core Generation workflow infrastructure to generate a reusable HDL IP core. The workflow produces an IP core report that displays the target interface configuration and the code generator settings that you specify. You can integrate the IP core into a larger design by adding it in an embedded system integration environment. See “Custom IP Core Generation” on page 40-2.

This figure shows how the software generates an IP core with an AXI interface and integrates the IP core into the FPGA reference design.

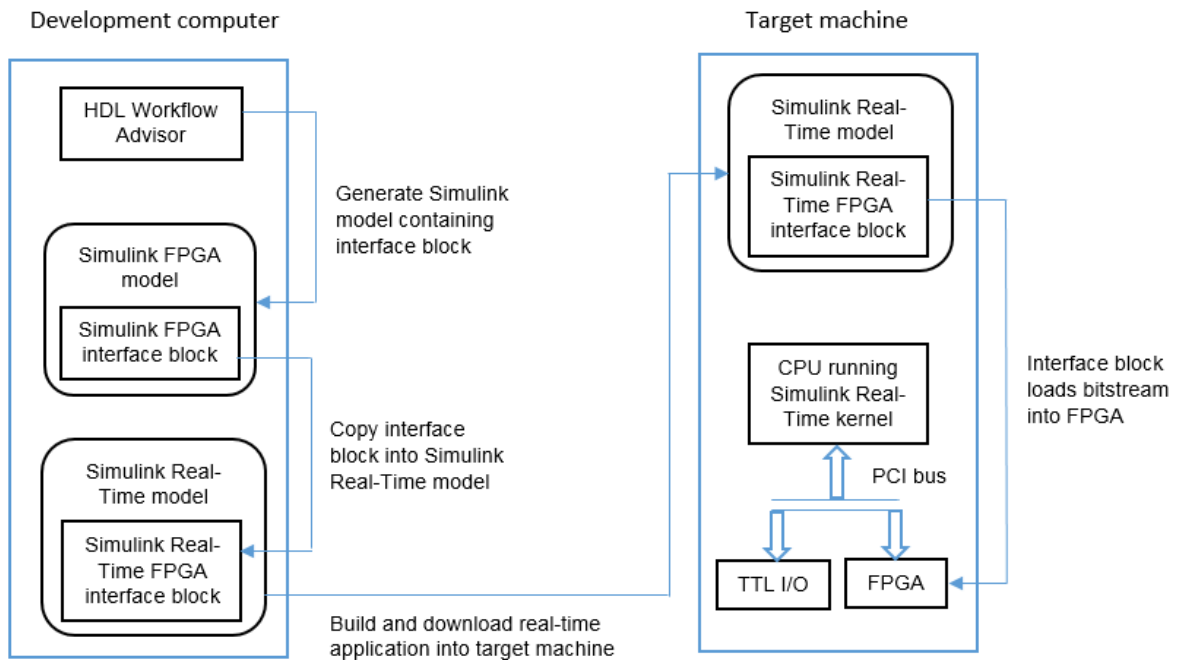


HDL Coder supports the Speedgoat I0333–325K with the Simulink Real-Time FPGA I/O workflow. This workflow uses the IP Core Generation workflow infrastructure and has these key features:

- Support for 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 board during real-time execution.



To learn more about the workflow, see “FPGA Programming and Configuration” (Simulink Real-Time).

See Also

More About

- “FPGA Programming and Configuration” (Simulink Real-Time)
- “Custom IP Core Generation” on page 40-2

External Websites

- www.speedgoat.com/support

