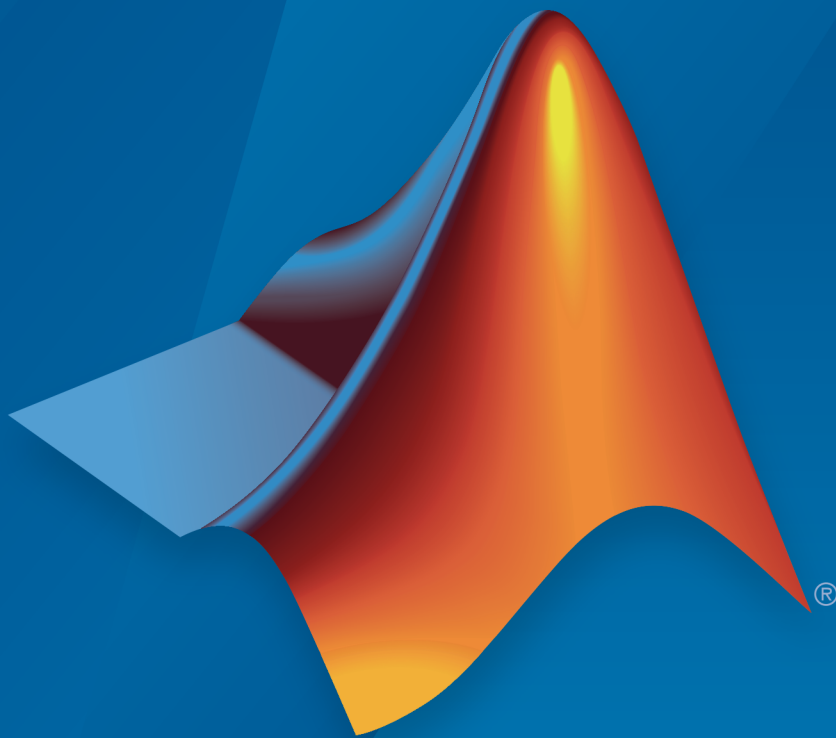


HDL Coder™

User's Guide



MATLAB®

R2015a



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.
3 Apple Hill Drive
Natick, MA 01760-2098

HDL Coder™ User's Guide

© COPYRIGHT 2012-2015 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)

HDL Code Generation from MATLAB

MATLAB Algorithm Design

1

Data Types and Scope	1-2
Supported Data Types	1-2
Unsupported Data Types	1-3
Scope for Variables	1-3
Operators	1-4
Arithmetic Operators	1-4
Relational Operators	1-4
Logical Operators	1-5
Control Flow Statements	1-6
Vector Function Limitations Related to Control Statements	1-7
Persistent Variables	1-8
Persistent Array Variables	1-10
Complex Data Type Support	1-11
Declaring Complex Signals	1-11
Conversion Between Complex and Real Signals	1-12
Support for Vectors of Complex Numbers	1-12
HDL Code Generation for System Objects	1-14
Why Use System Objects?	1-14
Predefined System Objects	1-14
User-Defined System Objects	1-14

Limitations of HDL Code Generation for System Objects	1-15
System object Examples for HDL Code Generation ...	1-16
Predefined System Objects Supported for HDL Code Generation	1-17
Predefined System Objects in MATLAB Code	1-17
Predefined System Objects in the MATLAB System Block	1-18
Load constants from a MAT-File	1-19
Generate Code for User-Defined System Objects	1-20
How To Create A User-Defined System object	1-20
User-Defined System object Example	1-20
Map Matrices to ROM	1-23
Fixed-Point Bitwise Functions	1-24
Overview	1-24
Bitwise Functions Supported for HDL Code Generation	1-24
Fixed-Point Run-Time Library Functions	1-30
Fixed-Point Function Limitations	1-34
Model State with Persistent Variables and System Objects	1-35
Bit Shifting and Bit Rotation	1-39
Bit Slicing and Bit Concatenation	1-42
Guidelines for Efficient HDL Code	1-44
MATLAB Design Requirements for HDL Code Generation	1-45
What Is a MATLAB Test Bench?	1-46
MATLAB Test Bench Requirements and Best Practices	1-47
MATLAB Test Bench Requirements	1-47
MATLAB Test Bench Best Practices	1-47

MATLAB Best Practices and Design Patterns for HDL Code Generation

2

Model a Counter for HDL Code Generation	2-2
MATLAB Counter	2-2
MATLAB Code for the Counter	2-3
Best Practices in this Example	2-4
Model a State Machine for HDL Code Generation	2-5
MATLAB State Machines	2-5
MATLAB Code for the Mealy State Machine	2-5
MATLAB Code for the Moore State Machine	2-7
Best Practices	2-9
Generate Hardware Instances For Local Functions	2-10
MATLAB Local Functions	2-10
MATLAB Code for mlhdlc_two_counters.m	2-10
Implement RAM Using MATLAB Code	2-13
Implementation of RAM	2-13
Implement RAM Using a Persistent Array or System object Properties	2-13
Implement RAM Using hdl.RAM	2-14
For-Loop Best Practices for HDL Code Generation	2-16
MATLAB Loops	2-16
Monotonically Increasing Loop Counters	2-16
Persistent Variables in Loops	2-17
Persistent Arrays in Loops	2-17

Fixed-Point Conversion

3

Floating-Point to Fixed-Point Conversion	3-2
Fixed-Point Type Conversion and Refinement	3-16
Working with Generated Fixed-Point Files	3-25

Specify Type Proposal Options	3-32
Log Data for Histogram	3-36
View and Modify Variable Information	3-39
View Variable Information	3-39
Modify Variable Information	3-39
Revert Changes	3-41
Promote Sim Min and Sim Max Values	3-42
Automated Fixed-Point Conversion	3-43
License Requirements	3-43
Automated Fixed-Point Conversion Capabilities	3-43
Code Coverage	3-45
Proposing Data Types	3-48
Locking Proposed Data Types	3-50
Viewing Functions	3-50
Viewing Variables	3-51
Histogram	3-57
Function Replacements	3-59
Validating Types	3-60
Testing Numerics	3-60
Detecting Overflows	3-60
Custom Plot Functions	3-62
Visualize Differences Between Floating-Point and Fixed-Point Results	3-64
Inspecting Data Using the Simulation Data Inspector	3-70
What Is the Simulation Data Inspector?	3-70
Import Logged Data	3-70
Export Logged Data	3-70
Group Signals	3-71
Run Options	3-71
Create Report	3-71
Comparison Options	3-71
Enabling Plotting Using the Simulation Data Inspector	3-71
Save and Load Simulation Data Inspector Sessions	3-72
Enable Plotting Using the Simulation Data Inspector	3-73
From the UI	3-73
From the Command Line	3-73

Replacing Functions Using Lookup Table	
Approximations	3-75
Replace a Custom Function with a Lookup Table	3-76
From the UI	3-76
From the Command Line	3-84
Replace the exp Function with a Lookup Table	3-87
From the UI	3-87
From the Command Line	3-95
Data Type Issues in Generated Code	3-98
Enable the Highlight Option in a Project	3-98
Enable the Highlight Option at the Command Line ...	3-98
Stowaway Doubles	3-98
Stowaway Singles	3-98
Expensive Fixed-Point Operations	3-98

Code Generation

4

Create and Set Up Your Project	4-2
Create a New Project	4-2
Open an Existing Project	4-4
Add Files to the Project	4-4
Primary Function Input Specification	4-6
When to Specify Input Properties	4-6
Why You Must Specify Input Properties	4-6
Properties to Specify	4-6
Rules for Specifying Properties of Primary Inputs	4-8
Methods for Defining Properties of Primary Inputs	4-8
Basic HDL Code Generation with the Workflow	
Advisor	4-10
HDL Code Generation from System Objects	4-14
Generate Instantiable Code for Functions	4-19
How to Generate Instantiable Code for Functions	4-19

Generate Code Inline for Specific Functions	4-19
Limitations for Instantiable Code Generation for Functions	4-19
Integrate Custom HDL Code Into MATLAB Design . . .	4-21
Define the hdl.BlackBox System object	4-21
Use System object In MATLAB Design Function	4-23
Generate HDL Code	4-23
Limitations for hdl.BlackBox	4-26
Enable MATLAB Function Block Generation	4-27
Requirements for MATLAB Function Block Generation	4-27
Enable MATLAB Function Block Generation	4-27
Results of MATLAB Function Block Generation	4-27
System Design with HDL Code Generation from MATLAB and Simulink	4-28
Generate Xilinx System Generator Black Box Block . .	4-32
Requirements for System Generator Black Box Block Generation	4-32
Enable System Generator Black Box Block Generation	4-32
Results of System Generator Black Box Block Generation	4-33
Generate Xilinx System Generator for DSP Black Box from MATLAB HDL Design	4-34
Generate HDL Code from MATLAB Code Using the Command Line Interface	4-40
Specify the Clock Enable Rate	4-45
Why Specify the Clock Enable Rate?	4-45
How to Specify the Clock Enable Rate	4-45
Specify Test Bench Clock Enable Toggle Rate	4-47
When to Specify Test Bench Clock Enable Toggle Rate	4-47
How to Specify Test Bench Clock Enable Toggle Rate .	4-47
Generate an HDL Coding Standard Report from MATLAB	4-49
Using the HDL Workflow Advisor	4-49
Using the Command Line	4-51

Generate an HDL Lint Tool Script	4-53
How To Generate an HDL Lint Tool Script	4-53
Generate a Board-Independent IP Core from MATLAB	4-55
Generate a Board-Independent IP Core	4-55
Requirements and Limitations for IP Core Generation	4-57
Minimize Clock Enables	4-58
Using the GUI	4-59
Using the Command Line	4-59
Limitations	4-59

Verification

5

Verify Code with HDL Test Bench	5-2
Generate Test Bench With File I/O	5-5
When to Use File I/O In Test Bench	5-5
How Test Bench Generation with File I/O Works	5-5
Test Bench Data Files	5-5
How to Generate Test Bench with File I/O	5-6
Limitations When Using File I/O In Test Bench	5-6

Deployment

6

Generate Synthesis Scripts	6-2
---	------------

Optimization

7

RAM Mapping	7-2
--------------------------	------------

Map Persistent Arrays and dsp.Delay to RAM	7-3
How To Enable RAM Mapping	7-3
RAM Mapping Requirements for Persistent Arrays and System object Properties	7-4
RAM Mapping Requirements for dsp.Delay System Objects	7-6
RAM Mapping Comparison for MATLAB Code	7-8
Pipelining MATLAB Code	7-9
Port Registers	7-9
Input and Output Pipeline Registers	7-9
Operation Pipelining	7-9
Register Inputs and Outputs	7-10
Insert Input and Output Pipeline Registers	7-11
Pipeline MATLAB Expressions	7-12
How To Pipeline a MATLAB Expression	7-12
Limitations of Pipelining for MATLAB Expressions ...	7-13
Distributed Pipelining	7-14
What is Distributed Pipelining?	7-14
Benefits and Costs of Distributed Pipelining	7-14
Selected Bibliography	7-14
Optimize MATLAB Loops	7-15
Loop Streaming	7-15
Loop Unrolling	7-15
How to Optimize MATLAB Loops	7-15
Limitations for MATLAB Loop Optimization	7-16
Constant Multiplier Optimization	7-18
Specify Constant Multiplier Optimization	7-20
Distributed Pipelining for Clock Speed Optimization .	7-21
Map Matrices to Block RAMs to Reduce Area	7-28
Resource Sharing of Multipliers to Reduce Area	7-33

Loop Streaming to Reduce Area	7-42
Constant Multiplier Optimization to Reduce Area ...	7-48

HDL Workflow Advisor Reference

8

HDL Workflow Advisor	8-2
Overview	8-2
 MATLAB to HDL Code and Synthesis	8-6
MATLAB to HDL Code Conversion	8-6
Code Generation: Target Tab	8-6
Code Generation: Coding Style Tab	8-7
Code Generation: Clocks and Ports Tab	8-9
Code Generation: Test Bench Tab	8-11
Code Generation: Optimizations Tab	8-13
Simulation and Verification	8-15
Synthesis and Analysis	8-15

HDL Code Generation from Simulink

Model Design for HDL Code Generation

9

Signal and Data Type Support	9-2
Overview	9-2
Buses	9-2
Enumerations	9-2
Unsupported Signal and Data Types	9-3
 Simulink Templates For HDL Code Generation	9-4
Create Model Using HDL Coder Model Template	9-4
HDL Coder Model Templates	9-4

Generate DUT Ports For Tunable Parameters	9-19
Create and Add Tunable Parameter that Maps to DUT Ports	9-19
Generated Code	9-20
Limitations	9-21
Use Tunable Parameter in Other Blocks	9-21

Code Generation Options in the HDL Coder Dialog Boxes

10

Set HDL Code Generation Options	10-2
HDL Code Generation Options in the Configuration Parameters Dialog Box	10-2
HDL Code Generation Options in the Model Explorer .	10-3
Code Menu	10-4
HDL Code Options in the Block Context Menu	10-5
The HDL Block Properties Dialog Box	10-6
HDL Code Generation Pane: General	10-8
HDL Code Generation Top-Level Pane Overview	10-10
Generate HDL for	10-12
Language	10-13
Folder	10-14
Generate HDL code	10-15
Generate validation model	10-16
Generate traceability report	10-17
Generate resource utilization report	10-18
Generate high-level timing critical path report	10-19
Generate optimization report	10-19
Generate model Web view	10-21
HDL Code Generation Pane: Global Settings	10-22
Global Settings Overview	10-26
Reset type	10-27
Reset asserted level	10-28
Clock input port	10-29
Clock enable input port	10-30
Reset input port	10-31
Clock inputs	10-32

Oversampling factor	10-33
Clock edge	10-33
Comment in header	10-35
Verilog file extension	10-36
VHDL file extension	10-37
Entity conflict postfix	10-38
Package postfix	10-39
Reserved word postfix	10-40
Split entity and architecture	10-42
Clocked process postfix	10-43
Complex real part postfix	10-43
Complex imaginary part postfix	10-44
Split entity file postfix	10-44
Split arch file postfix	10-46
Module name prefix	10-47
Enable prefix	10-48
Pipeline postfix	10-49
Timing controller postfix	10-49
Generate VHDL code for model references into a single library	10-50
Input data type	10-51
Output data type	10-53
Clock enable output port	10-55
Balance delays	10-56
Use trigger signal as clock	10-56
Optimize timing controller	10-57
Minimize clock enables	10-58
Multiplier partitioning threshold	10-60
Max oversampling	10-61
RAM mapping threshold (bits)	10-62
Timing controller architecture	10-62
Max computation latency	10-63
Hierarchical distributed pipelining	10-64
Clock-rate pipelining	10-65
Allow clock-rate pipelining of DUT output ports	10-65
Preserve design delays	10-66
Distributed pipelining priority	10-67
Adders	10-68
Multipliers	10-68
Atomic subsystems	10-69
MATLAB Function blocks	10-70
Adder sharing minimum bitwidth	10-70
Multiplier sharing minimum bitwidth	10-71
Represent constant values by aggregates	10-73

Use “rising_edge/falling_edge” style for registers	10-74
Loop unrolling	10-75
Use Verilog `timescale directives	10-76
Inline VHDL configuration	10-77
Concatenate type safe zeros	10-78
Emit time/date stamp in header	10-79
Scalarize vector ports	10-80
Minimize intermediate signals	10-81
Include requirements in block comments	10-82
Inline MATLAB Function block code	10-83
Generate parameterized HDL code from masked subsystem	10-84
Initialize all RAM blocks	10-85
RAM Architecture	10-85
HDL coding standard	10-86
Do not show passing rules in coding standard report .	10-87
Check for duplicate names	10-87
Check for HDL keywords in design names	10-88
Check for initial statements that set RAM initial values	10-89
Check module, instance, and entity name length	10-90
Check signal, port, and parameter name length	10-91
Minimize use of clock enable signals	10-92
Detect usage of reset signals	10-93
Minimize use of variables	10-94
Check if-else statement chain length	10-95
Check if-else statement nesting depth	10-96
Check multiplier width	10-97
Check for non-integer constants	10-98
Check line length	10-99
Highlight feedback loops inhibiting delay balancing and optimizations	10-100
Highlight blocks inhibiting clock-rate pipelining . . .	10-101
Highlight blocks inhibiting distributed pipelining . .	10-102
HDL Code Generation Pane: Test Bench	10-104
Test Bench Overview	10-106
HDL test bench	10-107
Cosimulation blocks	10-108
Cosimulation model for use with:	10-109
Test bench name postfix	10-110
Force clock	10-111
Clock high time (ns)	10-112
Clock low time (ns)	10-113

Hold time (ns)	10-114
Setup time (ns)	10-115
Force clock enable	10-116
Clock enable delay (in clock cycles)	10-117
Force reset	10-118
Reset length (in clock cycles)	10-119
Hold input data between samples	10-121
Initialize test bench inputs	10-122
Multi-file test bench	10-123
Test bench reference postfix	10-125
Test bench data file name postfix	10-126
Use file I/O to read/write test bench data	10-127
Ignore output data checking (number of samples)	10-127
HDL Code Generation Pane: EDA Tool Scripts	10-130
EDA Tool Scripts Overview	10-132
Generate EDA scripts	10-133
Generate multicycle path information	10-134
Compile file postfix	10-135
Compile initialization	10-136
Compile command for VHDL	10-137
Compile command for Verilog	10-138
Compile termination	10-139
Simulation file postfix	10-140
Simulation initialization	10-141
Simulation command	10-142
Simulation waveform viewing command	10-143
Simulation termination	10-144
Choose synthesis tool	10-145
Synthesis file postfix	10-148
Synthesis initialization	10-149
Synthesis command	10-150
Synthesis termination	10-151
Choose HDL lint tool	10-151
Lint initialization	10-152
Lint command	10-153
Lint termination	10-153

11 Supported Blocks Library and Block Properties

Generate a Supported Blocks Report	11-2
Generate a Library of Supported Blocks	11-3
View HDL-Specific Block Documentation	11-3
HDL Block Properties	11-4
Overview	11-4
BalanceDelays	11-5
ConstMultiplierOptimization	11-6
ConstrainedOutputPipeline	11-7
DistributedPipelining	11-8
DSPStyle	11-9
FlattenHierarchy	11-12
InputPipeline	11-13
InstantiateFunctions	11-13
InstantiateStages	11-15
LoopOptimization	11-15
LUTRegisterResetType	11-16
MapPersistentVarsToRAM	11-16
OutputPipeline	11-18
ResetType	11-19
SerialPartition	11-21
SharingFactor	11-21
SoftReset	11-21
StreamingFactor	11-23
UseMatrixTypesInHDL	11-23
UseRAM	11-24
VariablesToPipeline	11-28
HDL Filter Block Properties	11-29
AddPipelineRegisters	11-29
ChannelSharing	11-29
CoeffMultipliers	11-30
DALUTPartition	11-30
DARadix	11-32
FoldingFactor	11-33
MultiplierInputPipeline	11-33
MultiplierOutputPipeline	11-33
NumMultipliers	11-34
ReuseAccum	11-34

SerialPartition	11-34
Configuring HDL Filter Architectures	11-36
Fully Parallel Architecture	11-36
Serial Architectures	11-36
Distributed Arithmetic for HDL Filters	11-38
Requirements and Considerations for Generating	
Distributed Arithmetic Code	11-39
Further References	11-39
Set and View HDL Block Parameters	11-41
Set HDL Block Parameters from the GUI	11-41
Set HDL Block Parameters from the Command Line ..	11-41
View All HDL Block Parameters	11-42
View Non-Default HDL Block Parameters	11-42
Set HDL Block Parameters for Multiple Blocks	11-44
View HDL Model Parameters	11-46
Pass through, No HDL, and Cascade	
Implementations	11-47
Pass-through and No HDL Implementations	11-47
Cascade Architecture Best Practices	11-47
Test Bench Block Restrictions	11-48
Build a ROM Block with Simulink Blocks	11-49

Generating HDL Code for Multirate Models

12

Code Generation from Multirate Models	12-2
Clock Enable Generation for a Multirate DUT	12-2
Timing Controller for Multirate Models	12-5

Generate Reset for Timing Controller	12-6
Requirements for Timing Controller Reset Port	
Generation	12-6
How To Generate Reset for Timing Controller	12-6
Limitations for Timing Controller Reset Port	
Generation	12-6
Multirate Model Requirements for HDL Code	
Generation	12-7
Configuring Model Parameters	12-7
Sample Rate Requirements	12-7
Block Configuration and Restrictions For Multirate	
DUTs	12-8
Generate a Global Oversampling Clock	12-10
Why Use a Global Oversampling Clock?	12-10
Requirements for the Oversampling Factor	12-10
Specifying the Oversampling Factor From the GUI . .	12-11
Specifying the Oversampling Factor From the Command	
Line	12-12
Resolving Oversampling Rate Conflicts	12-12
Use Trigger As Clock in Triggered Subsystems	12-16
When To Use Trigger As Clock	12-16
Requirements For Using Trigger As Clock	12-16
How To Specify Trigger As Clock	12-16
Limitations When Using Trigger As Clock	12-17
Generate Multicycle Path Information Files	12-18
Overview	12-18
Format and Content of a Multicycle Path Information	
File	12-19
File Naming and Location Conventions	12-24
Generating Multicycle Path Information Files Using the	
GUI	12-24
Generating Multicycle Path Information Files Using the	
Command Line	12-24
Limitations	12-25
Using Multiple Clocks in HDL Coder™	12-27

Generating Bit-True Cycle-Accurate Models

13

Generated Model and Validation Model	13-2
Generated Model	13-2
Validation Model	13-3
Locate Numeric Differences After Speed Optimization	13-5

Optimization

14

Automatic Iterative Optimization	14-2
How Automatic Iterative Optimization Works	14-2
Automatic Iterative Optimization Output	14-3
Automatic Iterative Optimization Report	14-3
Requirements for Automatic Iterative Optimization ..	14-4
Limitations of Automatic Iterative Optimization	14-4
Optimization with Constrained Overclocking	14-5
Why Constrain Overclocking?	14-5
When to Use Constrained Overclocking	14-5
Set Overclocking Constraints	14-6
Constrained Overclocking Limitations	14-6
Maximum Oversampling Ratio	14-8
What Is the Maximum Oversampling Ratio?	14-8
Specify Maximum Oversampling Ratio	14-8
Maximum Oversampling Ratio Limitations	14-9
Maximum Computation Latency	14-10
What Is Maximum Computation Latency?	14-10
Specify Maximum Computation Latency	14-11
Maximum Computation Latency Restrictions	14-11
Streaming	14-12
What Is Streaming?	14-12
Specify Streaming	14-13

Requirements and Limitations for Streaming	14-13
Area Reduction with Streaming	14-15
The Validation Model	14-19
Resource Sharing	14-23
What Is Resource Sharing?	14-23
Specify Resource Sharing	14-24
Requirements and Limitations for Resource Sharing .	14-24
Resource Sharing Information in Reports	14-27
Check Compatibility for Resource Sharing	14-29
Delay Balancing	14-30
Why Use Delay Balancing	14-30
Specify Delay Balancing	14-30
Delay Balancing Limitations	14-32
Resolve Numerical Mismatch with Delay Balancing .	14-33
Find Feedback Loops	14-37
Using the HDL Workflow Advisor	14-37
Using the Configuration Parameters Dialog Box	14-38
Using the Command Line	14-38
Remove Highlighting	14-38
Limitations	14-38
Hierarchy Flattening	14-39
What Is Hierarchy Flattening?	14-39
When to Flatten Hierarchy	14-39
Prerequisites for Hierarchy Flattening	14-39
Options for Hierarchy Flattening	14-40
How to Flatten Hierarchy	14-40
Limitations for Hierarchy Flattening	14-41
RAM Mapping	14-42
RAM Mapping with the MATLAB Function Block . . .	14-43
Insert Distributed Pipeline Registers in a Subsystem	14-46

Distributed Pipelining and Hierarchical Distributed Pipelining	14-51
What Is Distributed Pipelining?	14-51
Benefits and Costs of Distributed Pipelining	14-53
Requirements for Distributed Pipelining	14-54
Specify Distributed Pipelining	14-54
Limitations of Distributed Pipelining	14-54
What Is Hierarchical Distributed Pipelining?	14-56
Benefits of Hierarchical Distributed Pipelining	14-59
Specify Hierarchical Distributed Pipelining	14-59
Limitations of Hierarchical Distributed Pipelining ..	14-60
Distributed Pipelining Workflow	14-60
Selected Bibliography	14-60
Constrained Output Pipelining	14-61
What Is Constrained Output Pipelining?	14-61
When to Use Constrained Output Pipelining	14-61
Requirements for Constrained Output Pipelining ...	14-61
Specify Constrained Output Pipelining	14-62
Limitations of Constrained Output Pipelining	14-62
Reduce Critical Path with Distributed Pipelining ..	14-63
Clock-Rate Pipelining	14-70
Reasons for Clock-Rate Pipelining	14-70
How Clock-Rate Pipelining Works	14-70
Best Practices for Clock-Rate Pipelining	14-71
When to Disable Clock-Rate Pipelining	14-71
How to Specify Clock-Rate Pipelining	14-71
Limitations for Clock-Rate Pipelining	14-71
Find Estimated Critical Paths Without Synthesis	
Tools	14-74
When To Use Critical Path Estimation	14-74
How Critical Path Estimation Works	14-74
How to Use Critical Path Estimation	14-76
Characterized Blocks	14-76
Inaccuracy in Critical Path Estimation	14-78

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

15

Create and Use Code Generation Reports	15-2
Information Included in Code Generation Reports	15-2
HDL Code Generation Report Summary	15-3
Resource Utilization Report	15-4
Optimization Report	15-6
Hierarchical Distributed Pipelining in the Optimization Report	15-8
Traceability Report	15-9
Traceability Report Overview	15-9
Generating a Traceability Report from Configuration Parameters	15-13
Generating a Traceability Report from the Command Line	15-16
Keeping the Report Current	15-18
Tracing from Code to Model	15-18
Tracing from Model to Code	15-20
Mapping Model Elements to Code Using the Traceability Report	15-23
Traceability Report Limitations	15-24
Web View of Model in Code Generation Report	15-26
About Model Web View	15-26
Generate HTML Code Generation Report with Model Web View	15-26
Model Web View Limitations	15-28
Generate Code with Annotations or Comments	15-29
Simulink Annotations	15-29
Text Comments	15-29
Requirements Comments and Hyperlinks	15-29
Check Your Model for HDL Compatibility	15-33
Create a Supported Blocks Library	15-36

Trace Code Using the Mapping File	15-38
Add or Remove the HDL Configuration Component .	15-41
What Is the HDL Configuration Component?	15-41
Adding the HDL Coder Configuration Component To a Model	15-41
Removing the HDL Coder Configuration Component From a Model	15-41

HDL Coding Standards

16

HDL Coding Standard Report	16-2
Rule Summary	16-2
Rule Hierarchy	16-2
Rule and Report Customization	16-3
How To Fix Warnings and Errors	16-3
HDL Coding Standards	16-4
Generate an HDL Coding Standard Report from Simulink	16-5
Using the HDL Workflow Advisor	16-5
Using the Command Line	16-7
HDL Coding Standard Rules	16-9
Generate an HDL Lint Tool Script	16-15
How To Generate an HDL Lint Tool Script	16-15

Interfacing Subsystems and Models to HDL Code

17

Model Referencing for HDL Code Generation	17-2
Benefits of Model Referencing for Code Generation ...	17-2
How To Generate Code for a Referenced Model	17-2
Limitations for Model Reference Code Generation	17-3

Generate Black Box Interface for Subsystem	17-4
What Is a Black Box Interface?	17-4
Generate a Black Box Interface for a Subsystem	17-4
Generate Code for a Black Box Subsystem	
Implementation	17-8
Restriction for Multirate DUTs	17-9
Generate Black Box Interface for Referenced Model	17-10
When to Generate a Black Box Interface	17-10
How to Generate a Black Box Interface	17-10
Integrate Custom HDL Code Using DocBlock	17-12
When To Use DocBlock to Integrate Custom Code	17-12
How To Use DocBlock to Integrate Custom Code	17-12
Restrictions	17-13
Example	17-13
Customize Black Box or HDL Cosimulation Interface	17-14
Interface Parameters	17-14
Specify Bidirectional Ports	17-17
Requirements	17-17
How To Specify a Bidirectional Port	17-17
Limitations	17-18
Generate Reusable Code for Atomic Subsystems	17-19
Requirements for Generating Reusable Code for Atomic	
Subsystems	17-19
Generate Reusable Code for Atomic Subsystems	17-19
Generate Reusable Code for Atomic Subsystems with	
Tunable Mask Parameters	17-22
Create a Xilinx System Generator Subsystem	17-27
Why Use Xilinx System Generator Subsystems?	17-27
Requirements for Xilinx System Generator	
Subsystems	17-27
How to Create a Xilinx System Generator Subsystem	17-28
Limitations for Code Generation from Xilinx System	
Generator Subsystems	17-28
Create an Altera DSP Builder Subsystem	17-29
Why Use Altera DSP Builder Subsystems?	17-29
Requirements for Altera DSP Builder Subsystems	17-29

How to Create an Altera DSP Builder Subsystem . . .	17-30
Determine Clocking Requirements for Altera DSP Builder Subsystems	17-30
Limitations for Code Generation from Altera DSP Builder Subsystems	17-31
Using Xilinx System Generator for DSP with HDL	
Coder	17-32
Generate a Cosimulation Model	17-36
What Is A Cosimulation Model?	17-36
Generating a Cosimulation Model from the GUI	17-37
Structure of the Generated Model	17-42
Launching a Cosimulation	17-49
The Cosimulation Script File	17-52
Complex and Vector Signals in the Generated Cosimulation Model	17-54
Generating a Cosimulation Model from the Command Line	17-56
Naming Conventions for Generated Cosimulation Models and Scripts	17-57
Limitations for Cosimulation Model Generation	17-57
Pass-Through and No-Op Implementations	17-59

Stateflow HDL Code Generation Support

18

Introduction to Stateflow HDL Code Generation	18-2
Overview	18-2
Example	18-2
Hardware Realization of Stateflow Semantics	18-3
Generate HDL for Mealy and Moore Finite State Machines	18-4
Overview	18-4
Generating HDL Code for a Moore Finite State Machine	18-4
Generating HDL for a Mealy Finite State Machine	18-8

Design Patterns Using Advanced Chart Features . . .	18-12
Temporal Logic	18-12
Graphical Function	18-14
Hierarchy and Parallelism	18-16
Stateless Charts	18-16
Truth Tables	18-18

Generating HDL Code with the MATLAB Function Block

19

HDL Applications for the MATLAB Function Block . .	19-2
Viterbi Decoder with the MATLAB Function Block . .	19-3
Code Generation from a MATLAB Function Block . . .	19-4
Counter Model Using the MATLAB Function block . . .	19-4
Setting Up	19-6
Creating the Model and Configuring General Model Settings	19-7
Adding a MATLAB Function Block to the Model	19-8
Set Fixed-Point Options for the MATLAB Function Block	19-10
Programming the MATLAB Function Block	19-14
Constructing and Connecting the DUT_eML_Block Subsystem	19-15
Compiling the Model and Displaying Port Data Types	19-17
Simulating the eml_hdl_incrementer_tut Model	19-18
Generating HDL Code	19-19
Generate Instantiable Code for Functions	19-22
How To Generate Instantiable Code for Functions . . .	19-22
Generate Code Inline for Specific Functions	19-23
Limitations for Instantiable Code Generation for Functions	19-23
MATLAB Function Block Design Patterns for HDL . .	19-24
The eml_hdl_design_patterns Library	19-24
Efficient Fixed-Point Algorithms	19-26
Model State Using Persistent Variables	19-29

Creating Intellectual Property with the MATLAB Function Block	19-30
Nontunable Parameter Arguments	19-31
Modeling Control Logic and Simple Finite State Machines	19-31
Modeling Counters	19-32
Modeling Hardware Elements	19-33
Design Guidelines for the MATLAB Function Block .	19-35
Introduction	19-35
Use Compiled External Functions With MATLAB Function Blocks	19-35
Build the MATLAB Function Block Code First	19-35
Use the hdlfimath Utility for Optimized FIMATH Settings	19-36
Use Optimal Fixed-Point Option Settings	19-38
Set the Output Data Type of MATLAB Function Blocks Explicitly	19-40
Distributed Pipeline Insertion for MATLAB Function Blocks	19-41
Overview	19-41
Distributed Pipelining in a Multiplier Chain	19-41

Generating Scripts for HDL Simulators and Synthesis Tools

20

Generate Scripts for Compilation, Simulation, and Synthesis	20-2
Structure of Generated Script Files	20-3
Properties for Controlling Script Generation	20-4
Enabling and Disabling Script Generation	20-4
Customizing Script Names	20-4
Customizing Script Code	20-5
Examples	20-7

Configure Compilation, Simulation, Synthesis, and Lint Scripts	20-8
Compilation Script Options	20-9
Simulation Script Options	20-10
Synthesis Script Options	20-12
Add Synthesis Attributes	20-16
Configure Synthesis Project Using Tcl Script	20-17

Using the HDL Workflow Advisor

21

What Is the HDL Workflow Advisor?	21-2
Open the HDL Workflow Advisor	21-3
Using the HDL Workflow Advisor Window	21-6
Save and Restore HDL Workflow Advisor State	21-9
How the Save and Restore Process Works	21-9
Limitations of the Save and Restore Process	21-9
Save the HDL Workflow Advisor State	21-9
Restore the HDL Workflow Advisor State	21-11
Fix a Workflow Advisor Warning or Failure	21-13
View and Save HDL Workflow Advisor Reports	21-15
Viewing HDL Workflow Advisor Reports	21-15
Saving HDL Workflow Advisor Reports	21-18
FPGA Floating-Point Library Mapping	21-20
What is an FPGA Floating-Point Library?	21-20
Why Map to an FPGA Floating Point Library?	21-20
Supported Floating-Point Operations	21-20
Limitations for FPGA Floating-Point Library Mapping	21-21
Map to an FPGA Floating-Point Library	21-23
Setup for FPGA Floating-Point Library Mapping	21-23
How to Map to an FPGA Floating-Point Library	21-23

Analyze Results of Floating-Point Library Mapping . . .	21-25
FPGA Synthesis and Analysis	21-26
FPGA Synthesis and Analysis Tasks Overview	21-26
Creating a Synthesis Project	21-26
Performing Synthesis, Mapping, and Place and Route	21-28
Annotating Your Model with Critical Path	
Information	21-31
Automated Workflows for Specific Targets and Tools	21-36

HDL Test Bench

22

Generate Test Bench With File I/O	22-2
When to Use File I/O In Test Bench	22-2
How Test Bench Generation with File I/O Works	22-2
Test Bench Data Files	22-2
How to Generate Test Bench with File I/O	22-3
Limitations When Using File I/O In Test Bench	22-4

FPGA Board Customization

23

FPGA Board Customization	23-2
Feature Description	23-2
Custom Board Management	23-2
FPGA Board Requirements	23-3
Create Custom FPGA Board Definition	23-7
Create Xilinx KC705 Evaluation Board Definition	
File	23-8
Overview	23-8
What You Need to Know Before Starting	23-8
Start New FPGA Board Wizard	23-9
Provide Basic Board Information	23-10

Specify FPGA Interface Information	23-12
Enter FPGA Pin Numbers	23-13
Run Optional Validation Tests	23-15
Save Board Definition File	23-17
Use New FPGA Board	23-18
FPGA Board Manager	23-21
Introduction	23-21
Filter	23-23
Search	23-23
FIL Enabled/Turnkey Enabled	23-23
Create Custom Board	23-23
Add Board from File	23-23
Get More Boards	23-23
View/Edit	23-24
Remove	23-24
Clone	23-24
Validate	23-24
New FPGA Board Wizard	23-25
Basic Information	23-26
Interfaces	23-27
FIL I/O	23-31
Turnkey I/O	23-33
Validation	23-36
Finish	23-37
FPGA Board Editor	23-38
General Tab	23-38
Interface Tab	23-40

HDL Workflow Advisor Tasks

24

HDL Workflow Advisor Tasks	24-2
HDL Workflow Advisor Tasks Overview	24-3
Set Target Overview	24-5
Set Target Device and Synthesis Tool	24-6
Set Target Library	24-7
Set Target Interface	24-8

Set Target Frequency	24-9
Set Target Interface	24-10
Set Target Interface	24-11
Prepare Model For HDL Code Generation Overview	24-12
Check Global Settings	24-13
Check Algebraic Loops	24-14
Check Block Compatibility	24-15
Check Sample Times	24-16
Check FPGA-in-the-Loop Compatibility	24-17
HDL Code Generation Overview	24-18
Set Code Generation Options Overview	24-19
Set Basic Options	24-20
Set Advanced Options	24-21
Set Testbench Options	24-22
Generate RTL Code and Testbench	24-23
Generate RTL Code and IP Core	24-25
FPGA Synthesis and Analysis Overview	24-26
Create Project	24-27
Perform Synthesis and P/R Overview	24-28
Perform Logic Synthesis	24-29
Perform Mapping	24-30
Perform Place and Route	24-31
Run Synthesis	24-31
Run Implementation	24-31
Annotate Model with Synthesis Result	24-33
Download to Target Overview	24-35
Generate Programming File	24-36
Program Target Device	24-37
Generate Simulink Real-Time Interface	24-38
Save and Restore HDL Workflow Advisor State	24-39
FPGA-in-the-Loop Implementation	24-39
Set FIL Options	24-39
Build FPGA-in-the-Loop	24-40
Check USRP [®] Compatibility	24-40
Verify with HDL Cosimulation	24-40
Generate FPGA Implementation	24-40
Check SDR Compatibility	24-40
SDR FPGA Implementation	24-41
Set SDR Options	24-41
Build SDR	24-43
Embedded System Integration	24-43
Create Project	24-43
Generate Software Interface Model	24-43
Build FPGA Bitstream	24-44

Hardware-Software Codesign

25	Hardware-Software Codesign Basics	
	Hardware-Software Codesign Workflow	25-2
	Custom IP Core Generation	25-5
	Custom IP Core Architectures	25-5
	Target Platform Interfaces	25-6
	Processor/FPGA Synchronization	25-6
	Custom IP Core Generated Files	25-7
	Custom IP Core Report	25-8
	Summary	25-8
	Target Interface Configuration	25-8
	Register Address Mapping	25-9
	IP Core User Guide	25-10
	IP Core File List	25-11
	Generate a Board-Independent IP Core from	
	Simulink	25-13
	Generate a Board-Independent IP Core	25-13
	Requirements and Limitations for IP Core Generation	25-16
	Processor and FPGA Synchronization	25-17
	Free Running Mode	25-17
	Coprocessing – Blocking Mode	25-18
	Coprocessing – Nonblocking With Delay Mode	25-18

Hardware-Software Codesign Workflow for SoC Platforms	26-2
Board and Reference Design Registration System ..	26-11
Board, IP Core, and Reference Design Definitions ..	26-11
Board Registration Files	26-11
Reference Design Registration Files	26-12
Predefined Board and Reference Design Examples ..	26-13
Register a Custom Board	26-15
Define a Board	26-15
Create a Board Plugin	26-16
Define a Board Registration Function	26-17
Register a Custom Reference Design	26-18
Define a Reference Design	26-18
Create a Reference Design Plugin	26-19
Define a Reference Design Registration Function ..	26-20
Generate Simulink Real-Time Interface for Speedgoat Boards	26-22
Select a Speedgoat Target Device	26-22
Set the Target Interface for Speedgoat Boards	26-25
Code Generation, Synthesis, and Generation of Simulink Real-Time Interface Subsystem	26-28

System Objects

What Are System Objects?	27-2
---------------------------------------	-------------

System Objects vs. MATLAB Functions	27-3
System Objects vs. MATLAB Functions	27-3
Process Audio Data Using Only MATLAB Functions Code	27-3
Process Audio Data Using System Objects	27-4
System Design and Simulation in MATLAB	27-6
System Objects in Simulink	27-7
System Objects in the MATLAB Function Block	27-7
System Objects in the MATLAB System Block	27-7
System Object Methods	27-8
What Are System Object Methods?	27-8
The Step Method	27-8
Common Methods	27-9
System Design in MATLAB Using System Objects ...	27-11
Create Components for Your System	27-11
Configure Components for Your System	27-11
Assemble Components to Create Your System	27-13
Run Your System	27-14
Reconfigure Your System During Runtime	27-15

Define New System Objects

28

Define Basic System Objects	28-2
Change Number of Step Inputs or Outputs	28-4
Validate Property and Input Values	28-8
Initialize Properties and Setup One-Time Calculations	28-11
Set Property Values at Construction Time	28-13
Reset Algorithm State	28-15

Define Property Attributes	28-17
Hide Inactive Properties	28-21
Limit Property Values to Finite String Set	28-23
Process Tuned Properties	28-26
Release System Object Resources	28-28
Define Composite System Objects	28-30
Define Finite Source Objects	28-33
Save System Object	28-35
Load System Object	28-38
Clone System Object	28-42
Define System Object Information	28-43
Specify Locked Input Size	28-45
Use Update and Output for Nondirect Feedthrough .	28-47
Methods Timing	28-50
Setup Method Call Sequence	28-50
Step Method Call Sequence	28-50
System Object Input Arguments and ~ in Code	
Examples	28-52
What Are Mixin Classes?	28-53
Best Practices for Defining System Objects	28-54

HDL Code Generation from MATLAB

MATLAB Algorithm Design

- “Data Types and Scope” on page 1-2
- “Operators” on page 1-4
- “Control Flow Statements” on page 1-6
- “Persistent Variables” on page 1-8
- “Persistent Array Variables” on page 1-10
- “Complex Data Type Support” on page 1-11
- “HDL Code Generation for System Objects” on page 1-14
- “Predefined System Objects Supported for HDL Code Generation” on page 1-17
- “Load constants from a MAT-File” on page 1-19
- “Generate Code for User-Defined System Objects” on page 1-20
- “Map Matrices to ROM” on page 1-23
- “Fixed-Point Bitwise Functions” on page 1-24
- “Fixed-Point Run-Time Library Functions” on page 1-30
- “Model State with Persistent Variables and System Objects” on page 1-35
- “Bit Shifting and Bit Rotation” on page 1-39
- “Bit Slicing and Bit Concatenation” on page 1-42
- “Guidelines for Efficient HDL Code” on page 1-44
- “MATLAB Design Requirements for HDL Code Generation” on page 1-45
- “What Is a MATLAB Test Bench?” on page 1-46
- “MATLAB Test Bench Requirements and Best Practices” on page 1-47

Data Types and Scope

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"> • <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.
Real	<ul style="list-style-type: none"> • <code>double</code> • <code>single</code> 	HDL code generated with <code>double</code> or <code>single</code> data types can be used for simulation, but is not synthesizable.
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>
Structures	<code>struct</code>	Structures are supported in the body of the design algorithm, but

Types	Supported Data Types	Restrictions
		<p>are not supported as inputs to the top-level design function.</p> <p>Do not use structures in the testbench.</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

In the current release, the following data types are not supported:

- Cell array
- Inf

Scope for Variables

Global variables are not supported for HDL code generation.

Operators

Arithmetic Operators

HDL Coder supports the arithmetic operators (and equivalent MATLAB functions) listed in the following table.

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)	
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.
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.

Relational Operators

HDL Coder supports the relational operators (and equivalent MATLAB functions) listed in the following table.

Relation	Operator Syntax	Equivalent Function
Less than	A<B	lt(A,B)

Relation	Operator Syntax	Equivalent Function
Less than or equal to	$A \leq B$	<code>le(A,B)</code>
Greater than or equal to	$A \geq B$	<code>ge(A,B)</code>
Greater than	$A > B$	<code>gt(A,B)</code>
Equal	$A == B$	<code>eq(A,B)</code>
Not equal	$A \neq B$	<code>ne(A,B)</code>

Logical Operators

HDL Coder supports the logical operators (and equivalent MATLAB functions) listed in the following table.

Relation	Operator Syntax	M Function Equivalent	Notes
Logical And	$A \& B$	<code>and(A,B)</code>	
Logical Or	$A B$	<code>or(A,B)</code>	
Logical Xor	$A \text{ xor } B$	<code>xor(A,B)</code>	
Logical And (short circuiting)	$A \&\& B$	N/A	Use short circuiting logical operators within conditionals. See also “Control Flow Statements” on page 1-6.
Logical Or (short circuiting)	$A B$	N/A	Use short circuiting logical operators within conditionals. See also “Control Flow Statements” on page 1-6.
Element complement	$\sim A$	<code>not(A)</code>	

Control Flow Statements

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

Control Flow Statement	Restrictions
for	<p>Do not use <code>for</code> loops without static bounds.</p> <p>Do not use the <code>&</code> and <code> </code> operators within conditions of a <code>for</code> statement. Instead, use the <code>&&</code> and <code> </code> operators.</p> <p>HDL Coder does not support nonscalar expressions in the conditions of <code>for</code> statements. Instead, use the <code>all</code> or <code>any</code> functions to collapse logical vectors into scalars.</p>
if	<p>Do not use the <code>&</code> and <code> </code> operators within conditions of an <code>if</code> statement. Instead, use the <code>&&</code> and <code> </code> operators.</p> <p>HDL Coder does not support nonscalar expressions in the conditions of <code>if</code> statements. Instead, use the <code>all</code> or <code>any</code> functions to collapse logical vectors into scalars.</p>
switch	<p>The conditional expression in a <code>switch</code> or <code>case</code> statement must use only:</p> <ul style="list-style-type: none"> • <code>uint8</code>, <code>uint16</code>, <code>uint32</code>, <code>int8</code>, <code>int16</code>, or <code>int32</code> data types • Scalar data <p>If multiple <code>case</code> statements make assignments to the same variable, the numeric type and <code>fimath</code> specification for that variable must be the same in every <code>case</code> statement.</p>

The following control flow statements are not supported:

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

Vector Function Limitations Related to Control Statements

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

- `isequal`
- `bitrevorder`

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”.

Complex Data Type Support

In this section...

“Declaring Complex Signals” on page 1-11

“Conversion Between Complex and Real Signals” on page 1-12

“Support for Vectors of Complex Numbers” on page 1-12

Declaring Complex Signals

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

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

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

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

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

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

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

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

Conversion Between Complex and Real Signals

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

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

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

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

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

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

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

Support for Vectors of Complex Numbers

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

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

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

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

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

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

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

```
TYPE vector_of_unsigned4 IS ARRAY (NATURAL RANGE <>) OF unsigned(3 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 1-14
“Predefined System Objects” on page 1-14
“User-Defined System Objects” on page 1-14
“Limitations of HDL Code Generation for System Objects” on page 1-15
“System object Examples for HDL Code Generation” on page 1-16

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 System Toolbox™ are supported for HDL code generation. For a list, see “Predefined System Objects Supported for HDL Code Generation”.

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”.

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.
- Enumerations 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:

- If your design uses the `output` and `update` methods, it can call each method only once per System object.
- Public properties must be nontunable.
- 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”
- “Model State with Persistent Variables and System Objects”
- “Generate Code for User-Defined System Objects”
- “Integrate Custom HDL Code Into MATLAB Design”

Predefined System Objects Supported for HDL Code Generation

In this section...

“Predefined System Objects in MATLAB Code” on page 1-17

“Predefined System Objects in the MATLAB System Block” on page 1-18

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 System Toolbox System objects for HDL code generation:

- `comm.BPSKModulator`, `comm.BPSKDemodulator`
- `comm.PSKModulator`, `comm.PSKDemodulator`
- `comm.QPSKModulator`, `comm.QPSKDemodulator`
- `comm.RectangularQAMModulator`, `comm.RectangularQAMDemodulator`
- `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.Maximum`
- `dsp.Minimum`
- `dsp.BiquadFilter`
- `dsp.DCBlocker`
- `dsp.HDLComplexToMagnitudeAngle`
- `dsp.HDLFFT`, `dsp.HDLIFFT`

- `dsp.HDLNCO`
- `dsp.FIRFilter`

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

- `visionhdl.ChromaResampler`
- `visionhdl.ColorSpaceConverter`
- `visionhdl.DemosaicInterpolator`
- `visionhdl.EdgeDetector`
- `visionhdl.GammaCorrector`
- `visionhdl.LookupTable`
- `visionhdl.Histogram`
- `visionhdl.ImageStatistics`
- `visionhdl.ImageFilter`
- `visionhdl.MedianFilter`
- `visionhdl.Closing`
- `visionhdl.Dilation`
- `visionhdl.Erosion`
- `visionhdl.Opening`

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 [MATLAB System](#).

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 1-20

“User-Defined System object Example” on page 1-20

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:
 - `setup` and `step`
 - `setup`, `output`, and `update`

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

- 3 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.
- 4 Write a test bench function that exercises the top-level design function.
- 5 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 `setup` and `step` 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
    end
end
```

```

        Count
    end
    methods
        function obj = CounterSysObj(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end

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

```

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

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

```

function y = myDesign(u)

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

y = 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

```

end

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

More About

- “HDL Code Generation for System Objects”

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 “RAMMappingThreshold”. To learn how to set the RAM mapping threshold in MATLAB, see “How To Enable RAM Mapping”.

- 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

In this section...
“Overview” on page 1-24
“Bitwise Functions Supported for HDL Code Generation” on page 1-24

Overview

HDL Coder supports many bitwise functions that operate on fixed-point integers of arbitrary length. For more information about these bitwise functions, see “Bitwise Operations” in the Fixed-Point Designer™ documentation.

This section describes HDL code generation support for these functions. “Bitwise Functions Supported for HDL Code Generation” on page 1-24 summarizes the supported functions, with notes that describe considerations specific to HDL code generation. “Bit Slicing and Bit Concatenation” on page 1-42 and “Bit Shifting and Bit Rotation” on page 1-39 provide usage examples, with corresponding MATLAB and generated HDL code.

Bitwise Functions Supported for HDL Code Generation

The following table summarizes MATLAB bitwise functions that are supported for HDL code generation. The Description column notes considerations that are specific to HDL. 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” in the Fixed-Point Designer documentation.

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

MATLAB Syntax	Description	See Also
<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:</p> <pre> a[lidx:ridx]</pre>	<p><code>bitorreduce</code></p>
<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>	<p><code>bitset</code></p>
<code>bitreplicate(a, n)</code>	<p>Concatenate bits of <code>fi</code> object <code>a</code> <code>n</code> times</p>	<p><code>bitreplicate</code></p>
<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>	<p><code>bitrol</code></p>

MATLAB Syntax	Description	See Also
bitror(a, idx)	<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>	bitror
bitset(a, idx, val)	<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>	bitset
bitshift(a, idx)	<p>Note: For efficient HDL code generation, use <code>bitsll</code>, <code>bitsrl</code>, or <code>bitsra</code> <i>instead of</i> <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

MATLAB Syntax	Description	See Also
<code>bitsliceget(a, lidx, ridx)</code>	<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>	<code>bitsliceget</code>
<code>bitsll(a, idx)</code>	<p>Shift left logical.</p> <p><code>idx</code> must be a scalar within the range</p> $0 \leq \text{idx} < \text{wl}$ <p><code>wl</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>	<code>bitsll</code>
<code>bitsra(a, idx)</code>	<p>Shift right arithmetic.</p> <p><code>idx</code> must be a scalar within the range</p> $0 \leq \text{idx} < \text{wl}$ <p><code>wl</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>

MATLAB Syntax	Description	See Also
<code>bitsrl(a, idx)</code>	<p>Shift right logical.</p> <p><code>idx</code> must be a scalar within the range</p> $0 \leq \text{idx} < w1$ <p><code>w1</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>
<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:</p> $\text{\^}a[lidx:ridx]$	<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>

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 1-34 for general limitations of fixed-point run-time library functions for code generation.

Function	Restrictions
abs	Double data type not supported.
add	None
all	Double data type not supported.
any	Double data type not supported.
bitand	None
bitandreduce	None
bitcmp	None
bitconcat	None
bitget	None
bitor	None
bitorreduce	None
bitreplicate	None
bitrol	None
bitror	None
bitset	None
bitshift	None
bitsliceget	None
bitsll	None
bitsra	None
bitsrl	None
bitxor	None
bitxorreduce	None
ceil	None

Function	Restrictions
<code>complex</code>	None
<code>conj</code>	None
<code>convergent</code>	None
<code>ctranspose</code>	None
<code>divide</code>	<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>.
<code>end</code>	None
<code>eps</code>	<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.
<code>eq</code>	None
<code>fi</code>	None
<code>fimath</code>	None
<code>fix</code>	None
<code>floor</code>	None
<code>ge</code>	None
<code>getlsb</code>	None
<code>getmsb</code>	None
<code>gt</code>	None
<code>horzcat</code>	None
<code>imag</code>	None
<code>int8, int16, int32</code>	None
<code>iscolumn</code>	None

Function	Restrictions
isempty	None
isequal	None
isfi	None
isfimath	None
isfimathlocal	None
isfinite	None
isinf	None
isnan	None
isnumeric	None
isnumericitype	None
isreal	None
isrow	None
isscalar	None
issigned	None
isvector	None
le	None
length	None
logical	None
lowerbound	None
lsb	None
lt	None
max	None
min	None
minus	None
mpower	Both inputs must be scalar, and the exponent input, k, must be a constant integer.
mtimes	None
ndims	None

Function	Restrictions
ne	None
nearest	None
numberofelements	None
numerictype	None
plus	Inputs cannot be data type logical .
power	Both inputs must be scalar, and the exponent input, k, must be a constant integer.
range	None
real	None
realmax	None
realmin	None
reinterpretcast	None
repmat	None
rescale	None
reshape	None
round	None
sfi	None
sign	None
size	None
sqrt	None
sub	None
subsasgn	Supported data types for HDL code generation are listed in “Supported Data Types” on page 1-2
subsref	Supported data types for HDL code generation are listed in “Supported Data Types” on page 1-2
sum	None
times	Inputs cannot be data type logical .
transpose	None

Function	Restrictions
<code>ufi</code>	None
<code>uint8</code> , <code>uint16</code> , <code>uint32</code>	None
<code>uminus</code>	None
<code>uplus</code>	Inputs cannot be data type <code>logical</code> .
<code>upperbound</code>	None
<code>vertcat</code>	None

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 Not Supported for C/C++ Code Generation” for more information.

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

HDL Coder™ only supports the 'step' method of the System object 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');
```

Bit Shifting and Bit Rotation

HDL Coder supports shift and rotate functions that mimic HDL-specific operators without saturation and rounding logic.

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
```

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
```

For more fixed-point bitwise functions you can use in MATLAB code intended for HDL code generation, see “Fixed-Point Bitwise Functions”.

Bit Slicing and Bit Concatenation

This section describes how to 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));
```

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

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
```

For more fixed-point bitwise functions you can use in MATLAB code intended for HDL code generation, see “Fixed-Point Bitwise Functions”.

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 *sin*, *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.* 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 1-47.

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

Model a Counter for HDL Code Generation

In this section...
“MATLAB Counter” on page 2-2
“MATLAB Code for the Counter ” on page 2-3
“Best Practices in this Example” on page 2-4

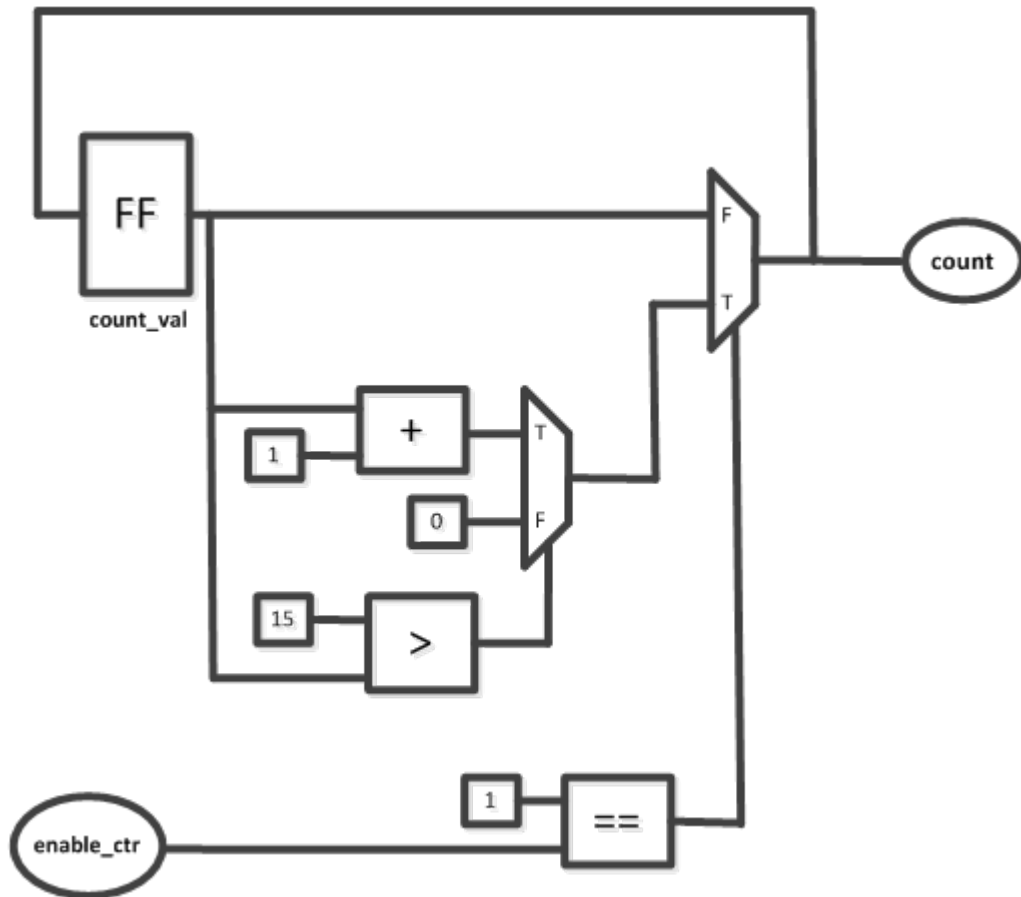
MATLAB Counter

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.

The schematic below shows the counter modeled in this example.



MATLAB Code for the Counter

The function `m1hdlc_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 2-5

“MATLAB Code for the Mealy State Machine” on page 2-5

“MATLAB Code for the Moore State Machine” on page 2-7

“Best Practices” on page 2-9

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 = S1;
        else
            Z = true;
            current_state = S2;
        end

    case S3,

        if (A)
            Z = false;
            current_state = S2;
        else
            Z = true;
            current_state = S3;
        end

    case S4,
```

```

        if (A)
            Z = true;
            current_state = S1;
        else
            Z = false;
            current_state = S3;
        end

        otherwise,

            Z = false;
    end
end

```

MATLAB Code for the Moore State Machine

The following MATLAB code defines the `mlhdlc_fsm_moore` function. A persistent variable represents the current state, and a `switch` block uses the current state to determine the output and new state. In each `case` in the `switch` block, an `if-else` statement calculates the new state and output. The value of the state is represented by numerical variables.

```

%#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 = S2;
        end

    case S3,
        Z = false;

        if (~A)
            curr_state = S2;
        else
            curr_state = S3;
        end

    case S4,
        Z = true;
        if (~A)
            curr_state = S1;
        else
            curr_state = S3;
        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 2-10
“MATLAB Code for mlhdlc_two_counters.m” on page 2-10

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 a 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 2-13

“Implement RAM Using a Persistent Array or System object Properties” on page 2-13

“Implement RAM Using hdl.RAM ” on page 2-14

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”. For more information, see “Map Persistent Arrays and `dsp.Delay to RAM`”.

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”.

For-Loop Best Practices for HDL Code Generation

In this section...
“MATLAB Loops” on page 2-16
“Monotonically Increasing Loop Counters” on page 2-16
“Persistent Variables in Loops” on page 2-17
“Persistent Arrays in Loops” on page 2-17

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 3-2
- “Fixed-Point Type Conversion and Refinement” on page 3-16
- “Working with Generated Fixed-Point Files” on page 3-25
- “Specify Type Proposal Options” on page 3-32
- “Log Data for Histogram” on page 3-36
- “View and Modify Variable Information” on page 3-39
- “Automated Fixed-Point Conversion” on page 3-43
- “Custom Plot Functions” on page 3-62
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 3-64
- “Inspecting Data Using the Simulation Data Inspector” on page 3-70
- “Enable Plotting Using the Simulation Data Inspector” on page 3-73
- “Replacing Functions Using Lookup Table Approximations” on page 3-75
- “Replace a Custom Function with a Lookup Table” on page 3-76
- “Replace the exp Function with a Lookup Table” on page 3-87
- “Data Type Issues in Generated Code” on page 3-98

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)

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);
```

```
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'];

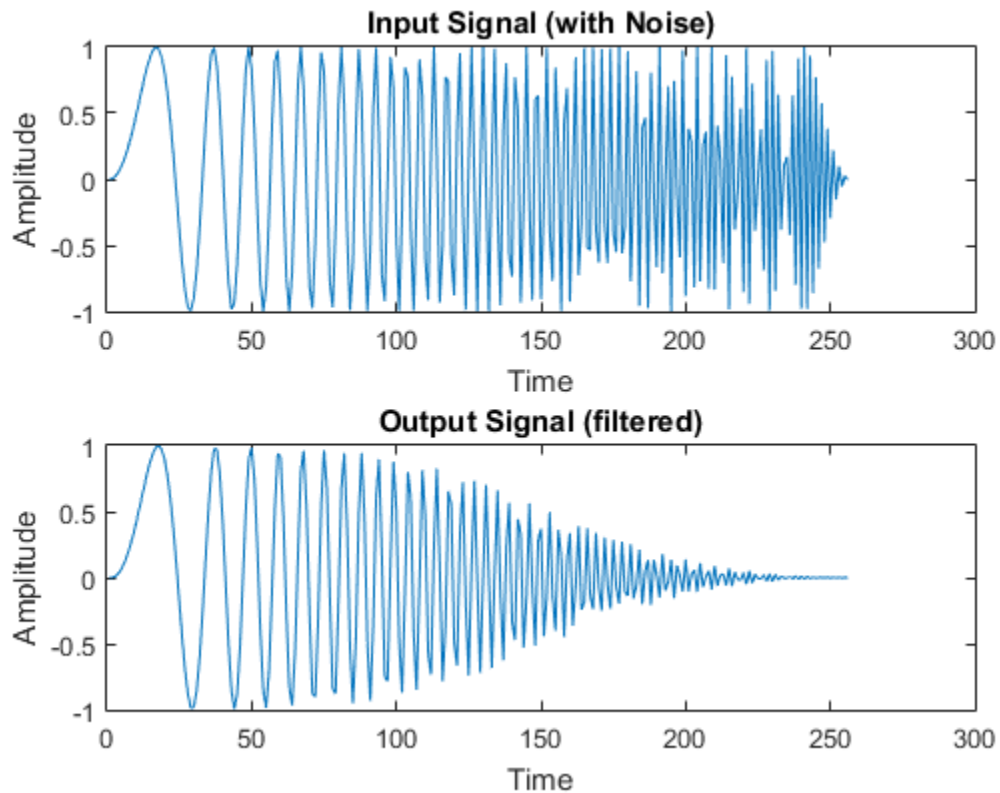
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

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.

```
m1hdlc_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.

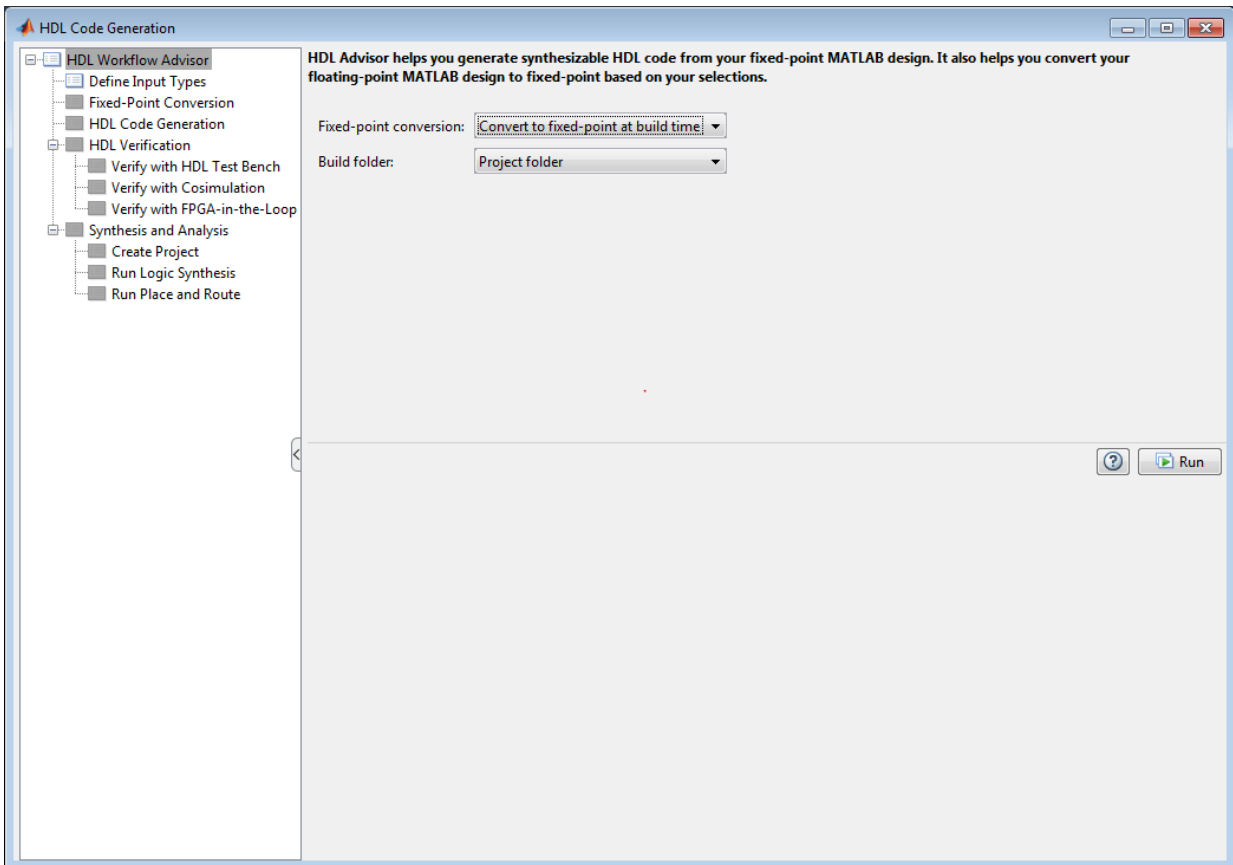
Fixed-Point Code Generation Workflow

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)'

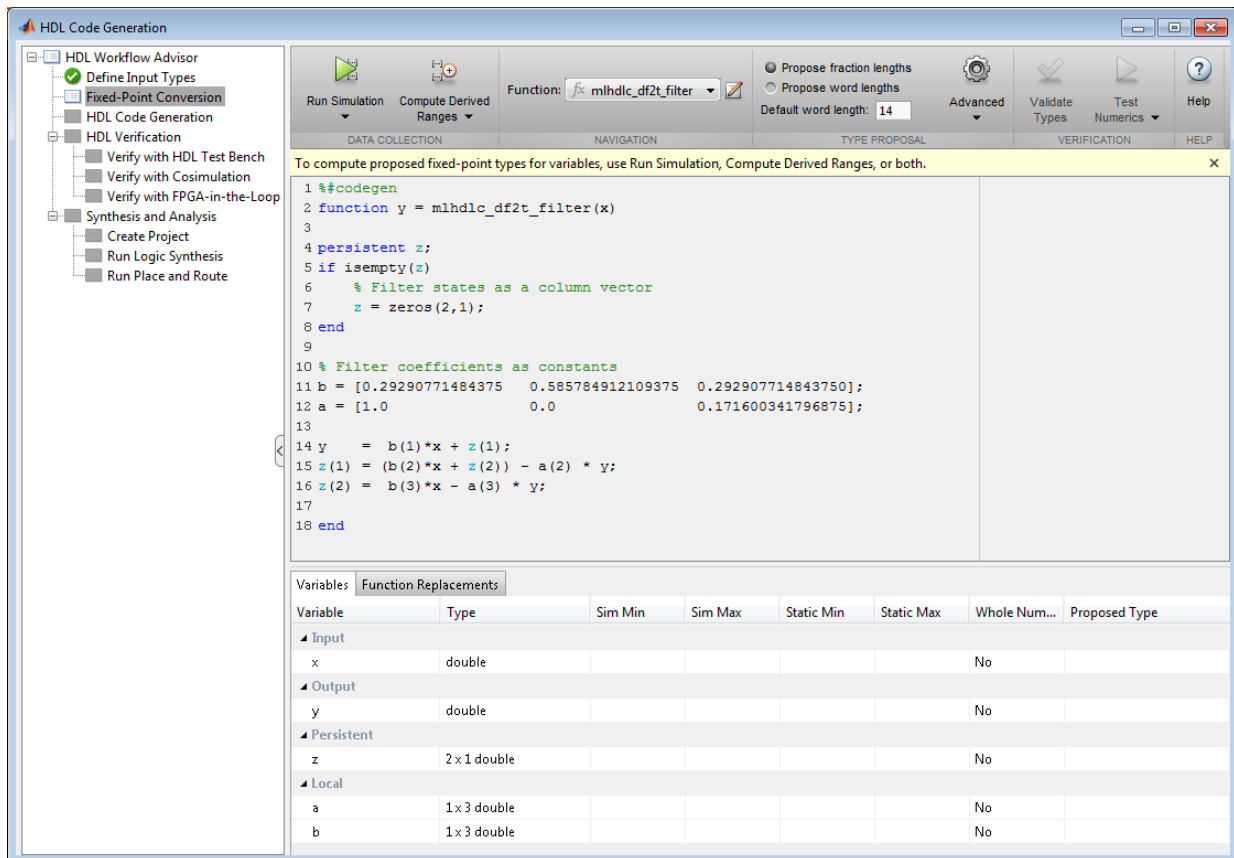
Step 3: Run Simulation

- 1 Click on the 'Fixed-Point Conversion' step.

3 Fixed-Point Conversion

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 Code Generation tool interface. The 'Fixed-Point Conversion' step is selected in the project tree. The main area displays the MATLAB code for the `mlhdlc_df2t_filter` function. Below the code, a table shows the inferred types and simulation ranges for variables `x`, `y`, `z`, `a`, and `b`.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
Input							
x	double					No	
Output							
y	double					No	
Persistent							
z	2 x 1 double					No	
Local							
a	1 x 3 double					No	
b	1 x 3 double					No	

1 Click the 'Run Simulation' step.

Notice that the 'Sim Min' and 'Sim Max' table is now populated with simulation ranges. Based on default wordlength settings fixed-point types are proposed.

The screenshot shows the HDL Code Generation tool interface. On the left is the HDL Workflow Advisor tree with 'Fixed-Point Conversion' selected. The main area displays MATLAB code for a filter function. Below the code is a table with tabs for 'Variables', 'Function Replacements', and 'Simulation Output'. The 'Simulation Output' tab is active, showing a table of variable simulation ranges.

```

1 %%codegen
2 function y = mlhdlc_df2t_filter(x)
3
4 persistent z;
5 if isempty(z)
6     % Filter states as a column vector
7     z = zeros(2,1);
8 end
9
10 % Filter coefficients as constants
11 b = [0.29290771484375    0.585784912109375    0.292907714843750];
12 a = [1.0                0.0                0.171600341796875];
13
14 y = b(1)*x + z(1);
15 z(1) = (b(2)*x + z(2)) - a(2) * y;
16 z(2) = b(3)*x - a(3) * y;
17
18 end
    
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
▲ Input							
x	double	-1	1			No	numerictype(1, 14, 12)
▲ Output							
y	double	-0.994520660237777	1			No	numerictype(1, 14, 12)
▲ Persistent							
z	2 x 1 double	-0.8	0.8			No	numerictype(1, 14, 13)
▲ Local							
a	1 x 3 double	0	1			No	numerictype(0, 14, 13)
b	1 x 3 double	0.29	0.59			No	numerictype(0, 14, 14)

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.

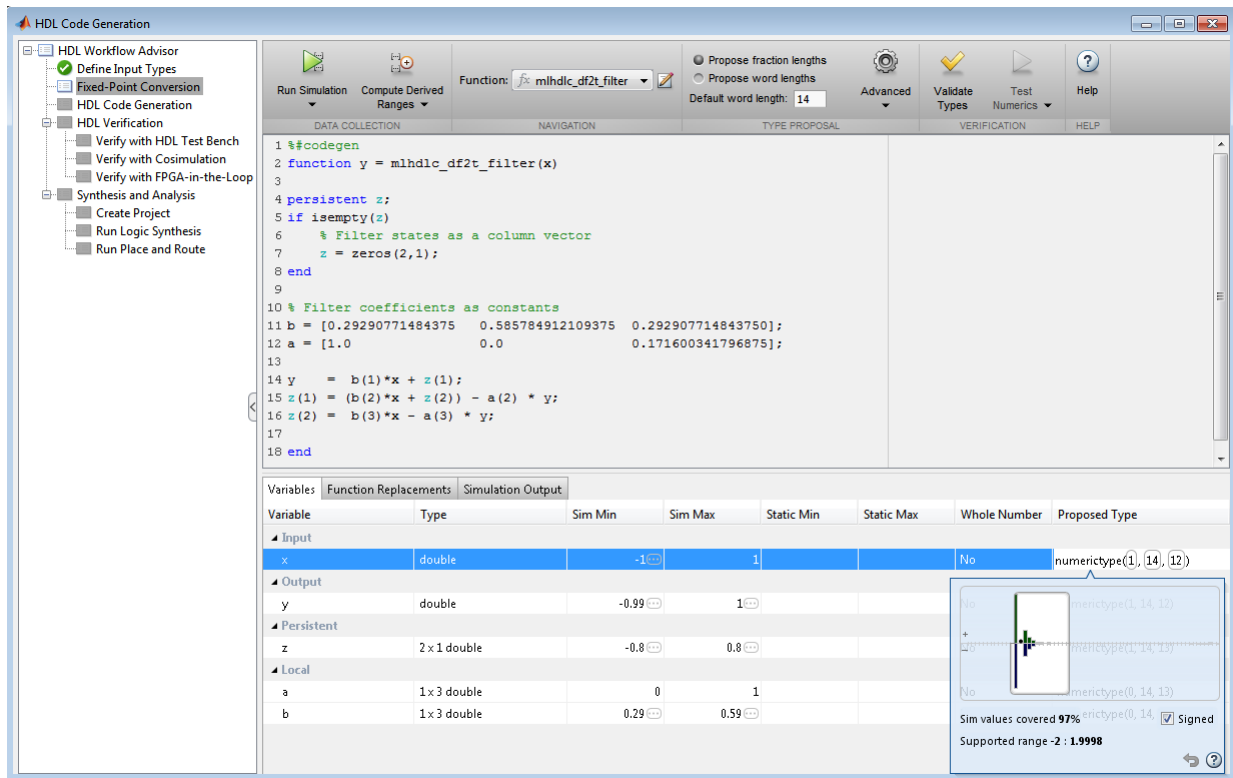
The type table contains the following information for each variable existing in the floating-point MATLAB design, organized by function:

- 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.

3 Fixed-Point Conversion

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 under 'Run simulation' button to enable logging of histogram data.



The screenshot displays the HDL Code Generation interface. On the left, the HDL Workflow Advisor shows the 'Fixed-Point Conversion' step selected. The main workspace shows the MATLAB code for the `mlhdlc_df2t_filter` function. Below the code, a table lists the proposed fixed-point types for variables `x`, `y`, `z`, `a`, and `b`.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
<code>x</code>	double	-1	1			No	<code>numeric(1, 14, 12)</code>
Output							
<code>y</code>	double	-0.99	1				
Persistent							
<code>z</code>	2 x 1 double	-0.8	0.8				
Local							
<code>a</code>	1 x 3 double	0	1				
<code>b</code>	1 x 3 double	0.29	0.59				

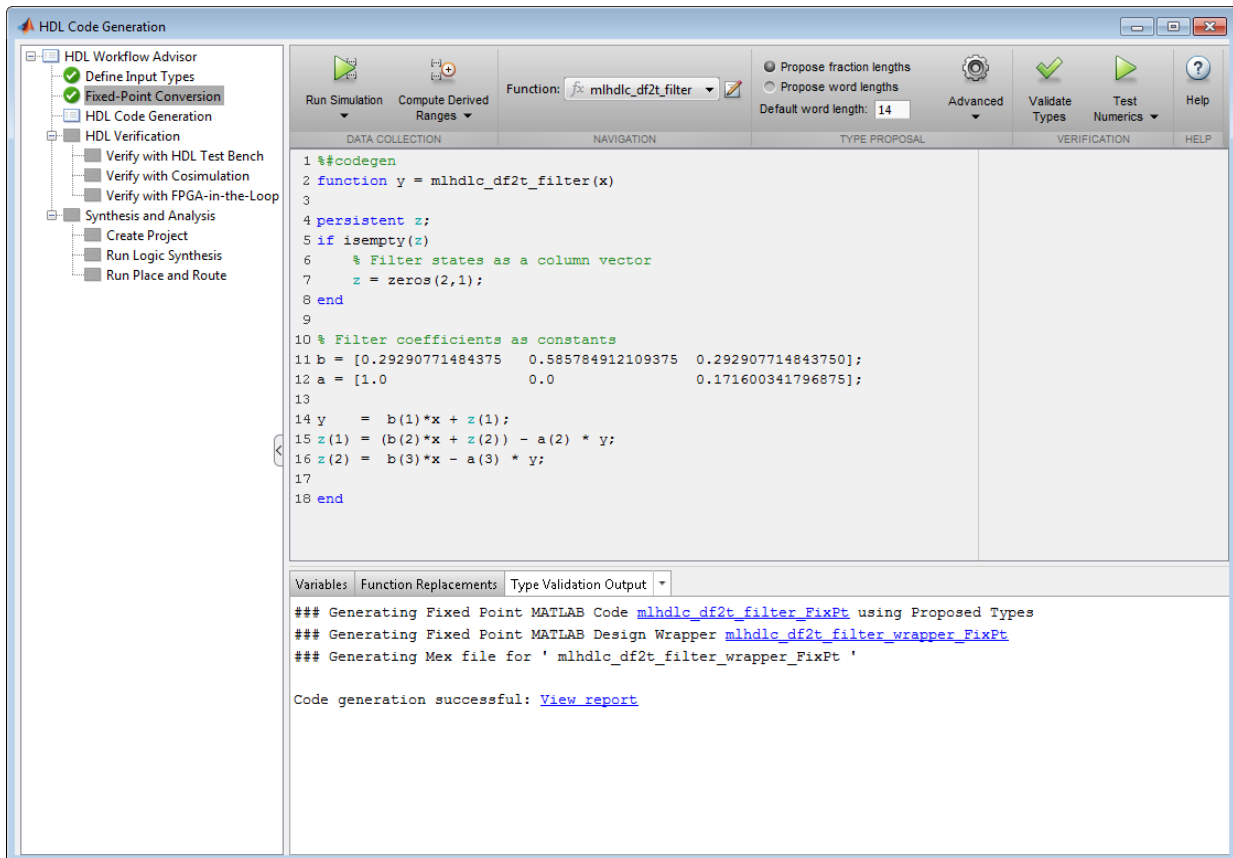
A histogram view is overlaid on the table, showing the distribution of simulation values for variable `x`. The histogram is centered around 0, with a bounding box indicating the proposed fixed-point type `numeric(1, 14, 12)`. The histogram shows that 97% of simulation values are covered by the proposed type, with a 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 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.

The screenshot shows the MATLAB Editor interface with two windows open. The top window displays the original floating-point function `mlhdlc_df2t_filter.m`, and the bottom window displays the generated fixed-point function `mlhdlc_df2t_filter_FixPt.m`.

Original Function (`mlhdlc_df2t_filter.m`):

```

1  %#codegen
2  function y = mlhdlc_df2t_filter(x)
3
4  persistent z a b;
5  if isempty(z)
6      % Filter states as a column vector
7      z = zeros(2,1);
8
9      % Filter coefficients as constants
10     b = [0.29290771484375    0.585784912109375    0.292907714843750];
11     a = [1.0                0.0                0.171600341796875];
12 end
13
14 y = b(1)*x + z(1);
15 z(1) = (b(2)*x + z(2)) - a(2) * y;
16 z(2) = b(3)*x - a(3) * y;
17
18 end
19

```

Generated Fixed-Point Function (`mlhdlc_df2t_filter_FixPt.m`):

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  %       Generated by MATLAB 7.14, MATLAB Coder 2.2 and HDL Coder 3.0
4  %
5  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6  %#codegen
7  function y = mlhdlc_df2t_filter_FixPt(x)
8
9  fm = hdlfimath;
10 persistent z a b
11 if isempty( z )
12     % Filter states as a column vector
13     z = fi(zeros( 2, 1 ), 1, 14, 13, fm);
14     % Filter coefficients as constants
15     b = fi([ 0.29290771484375, 0.585784912109375, 0.292907714843750 ], 0, 14, 14, fm);
16     a = fi([ 1.0, 0.0, 0.171600341796875 ], 0, 14, 13, fm);
17 end
18 y = fi(b( 1 )*x + z( 1 ), 1, 14, 12, fm);
19 z( 1 ) = fi((b( 2 )*x + z( 2 )) - a( 2 )*y, 1, 14, 13, fm);
20 z( 2 ) = fi(b( 3 )*x - a( 3 )*y, 1, 14, 13, fm);
21 end
22

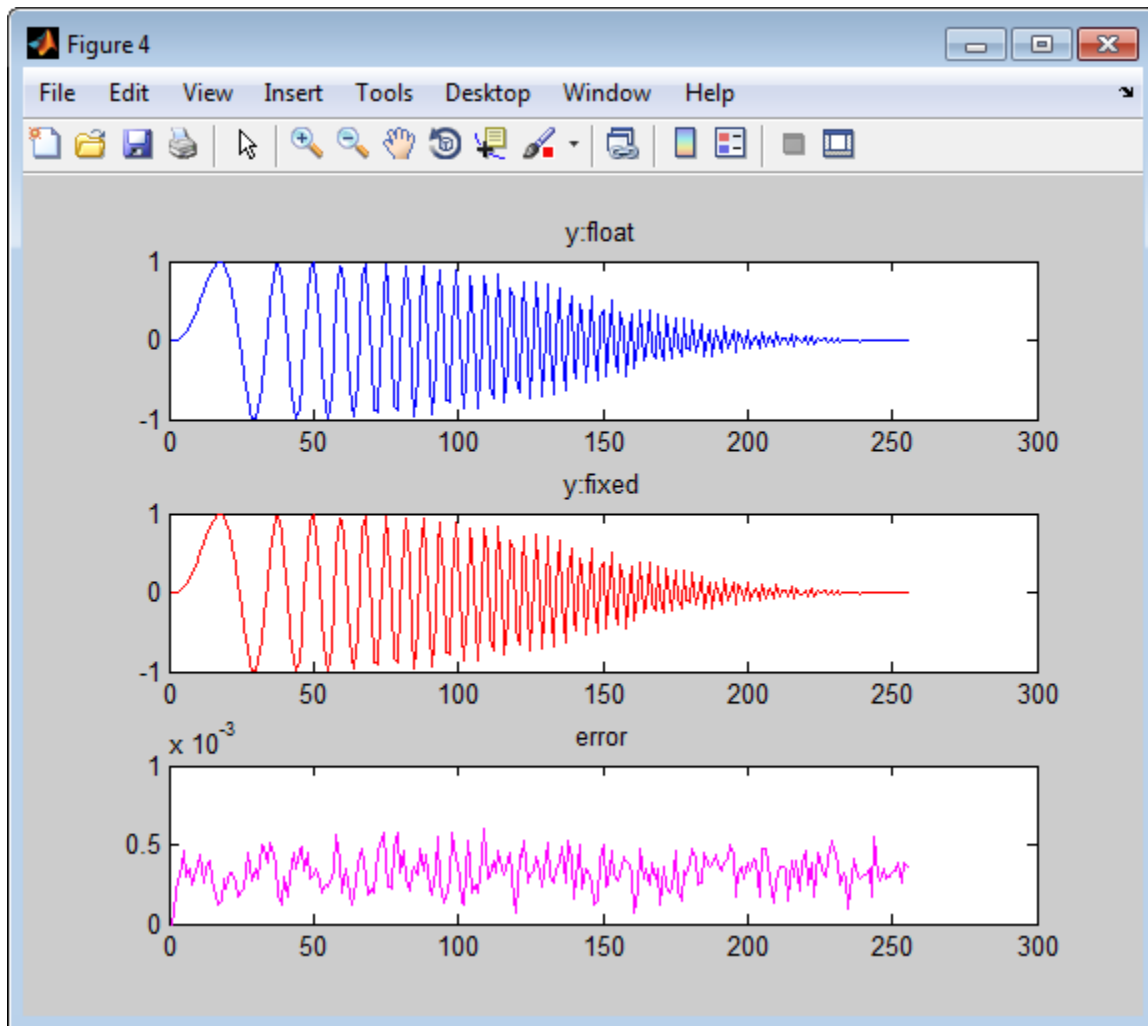
```

Step 5: Test Numerics

- 1 Click on the 'Test Numerics' step.

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

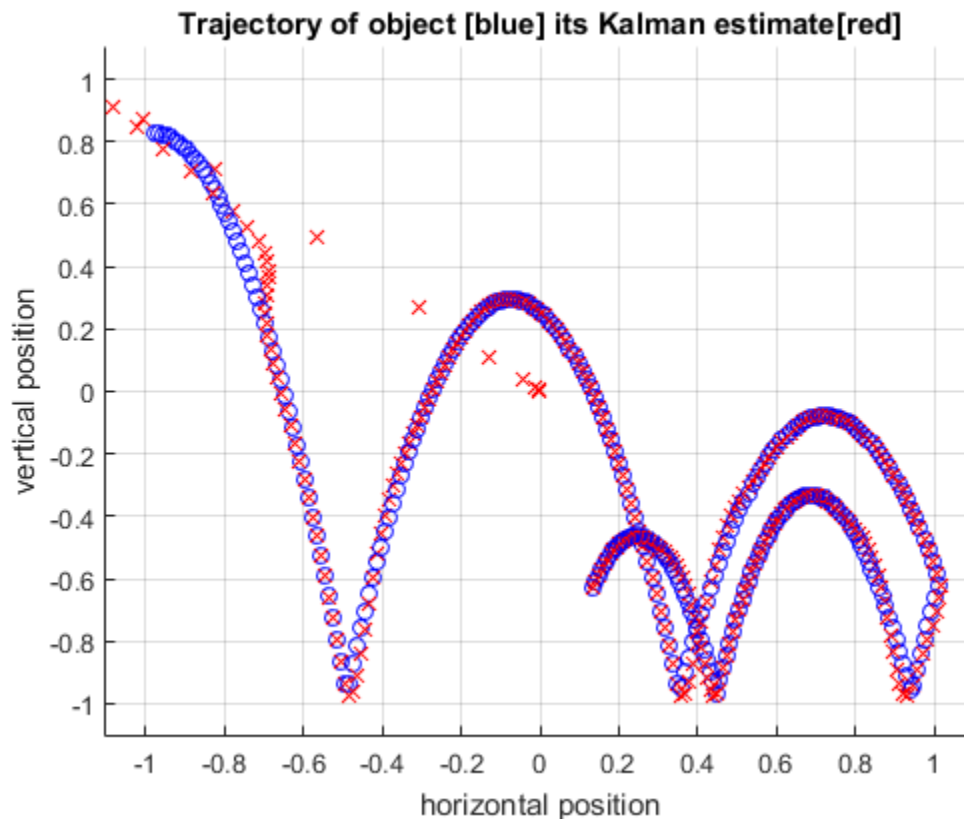
Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
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 'Run Simulation' 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. The top toolbar has a red box around the 'Propose fraction lengths' and 'Propose word lengths' buttons. The 'Default word length' is set to 22. Below the toolbar is a MATLAB code editor showing a function named 'mhdic_kalman_hdl'. The code includes comments for state transition matrix, measurement matrix, and initial conditions. At the bottom, a 'Variables' table displays simulation results for various variables, including their types, simulation ranges, and proposed fixed-point types.

Variable	Function	Replacements	Verification Output	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input										
x				2x1 double		-0.98	1.03		No	numersType(1, 22, 20)
Output										
y1				double	-1.14		1.03		No	numersType(1, 22, 20)
y2				double	-0.98		0.98		No	numersType(1, 22, 21)
dx_out_q				double	0		1		Yes	numersType(0, 1, 0)
Persistent										
state				double		1	5		Yes	numersType(0, 1, 0)
x_est				6x1 double	-1.14		1.03		No	numersType(1, 22, 20)
p_est				6x5 double	0		472.78		No	numersType(0, 22, 13)
y				2x1 double	-1.14		1.03		No	numersType(1, 22, 20)
x_prd				6x1 double	-1.35		1.37		No	numersType(1, 22, 20)
p_prd				6x5 double	0		896.74		No	numersType(0, 22, 12)
z_prd				2x1 double	-1.35		1.37		No	numersType(1, 22, 20)

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	Static Min	Static Max	Whole Number	Proposed Type
Input							
z	2 x 1 double		-0.98	1.01		No	numerictype(1, 22, 20)
Output							
y1	double		-1.14	1.01		No	numerictype(1, 22, 20)
y2	double		-0.98	0.98		No	numerictype(1, 22, 21)
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, 22, 20)
p_est	6 x 6 double		0	472.78		No	numerictype(0, 22, 15)
y	2 x 1 double		-1.14	1.01		No	numerictype(1, 22, 20)
x_prd	6 x 1 double		-1.35	1.17		No	numerictype(1, 22, 20)
p_prd	6 x 6 double		0	896.74		No	numerictype(0, 22, 12)
z_prd	2 x 1 double		-1.35	1.17		No	numerictype(1, 22, 20)
S	2 x 2 double		0	1896.74		No	numerictype(0, 22, 11)
B	2 x 6 double		0	896.74		No	numerictype(0, 22, 12)
klm_gain	6 x 2 double		0	0.47		No	numerictype(0, 22, 23)
dv_out	double		0	1		Yes	numerictype(0, 1, 0)
backslash_dv_out	double		0	1		Yes	numerictype(0, 1, 0)
Local							
dt	double		1	1		Yes	numerictype(0, 1, 0)
A	6 x 6 double		0	1		Yes	numerictype(0, 1, 0)
H	2 x 6 double		0	1		Yes	numerictype(0, 1, 0)
Q	6 x 6 double		0	1		Yes	numerictype(0, 1, 0)
R	2 x 2 double		0	1000		Yes	numerictype(0, 10, 0)

Examine the types proposed in the above table for variables instrumented in the top-level design.

Floating-Point Range for variable 'B':

- Simulation Info: SimMin: 0, SimMax: 896.74.., Whole Number: No
- Type Proposed: numerictype(0,22,12) (Signedness: Unsigned, WordLength: 22, FractionLength: 12)

The floating-point range:

- Has the same number of bits as the 'Default word length'.
- Uses the minimum number of bits to completely represent the range.
- Uses the rest of the bits to represent the precision.

Integer Range for variable 'A':

- Simulation Info: SimMin: 0, SimMax: 1, Whole Number: Yes

- Type Proposed: numerictype(0,1,0) (Signedness: Unsigned, WordLength: 1, FractionLength: 0)

The integer range:

- Has the minimum number of bits to represent the whole integer range.
- Has no fractional bits.

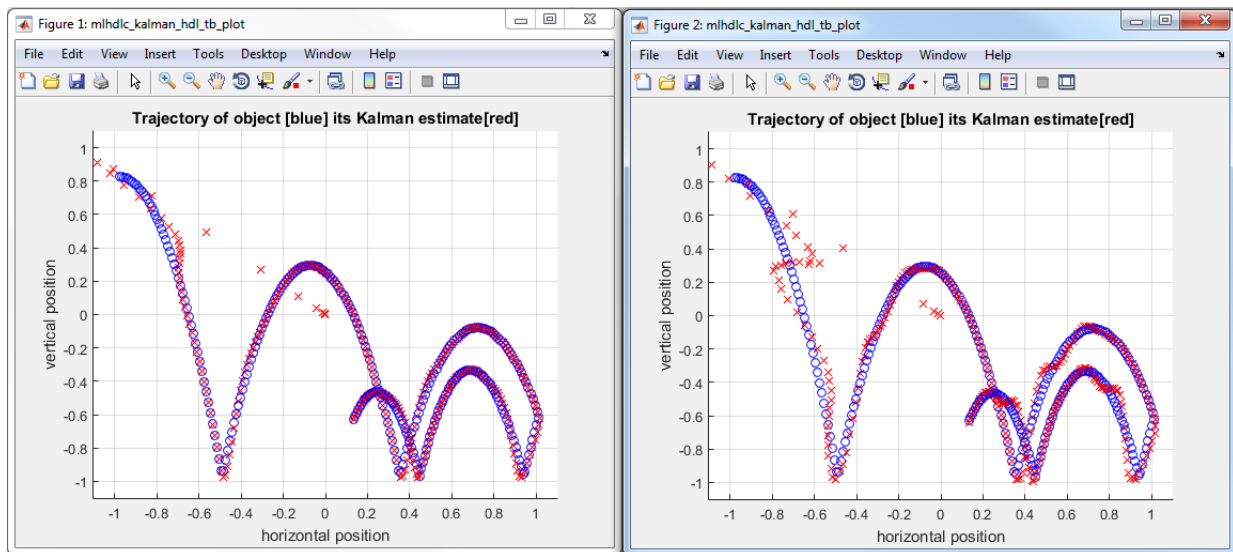
All the information in the table is editable, persists across iterations, and is saved with your code generation project.

Generate Fixed-Point Code and Verify the Generated Code

Based on the numeric types proposed for a default word length of 22, continue with fixed-point code generation and verification steps and observe the plots.

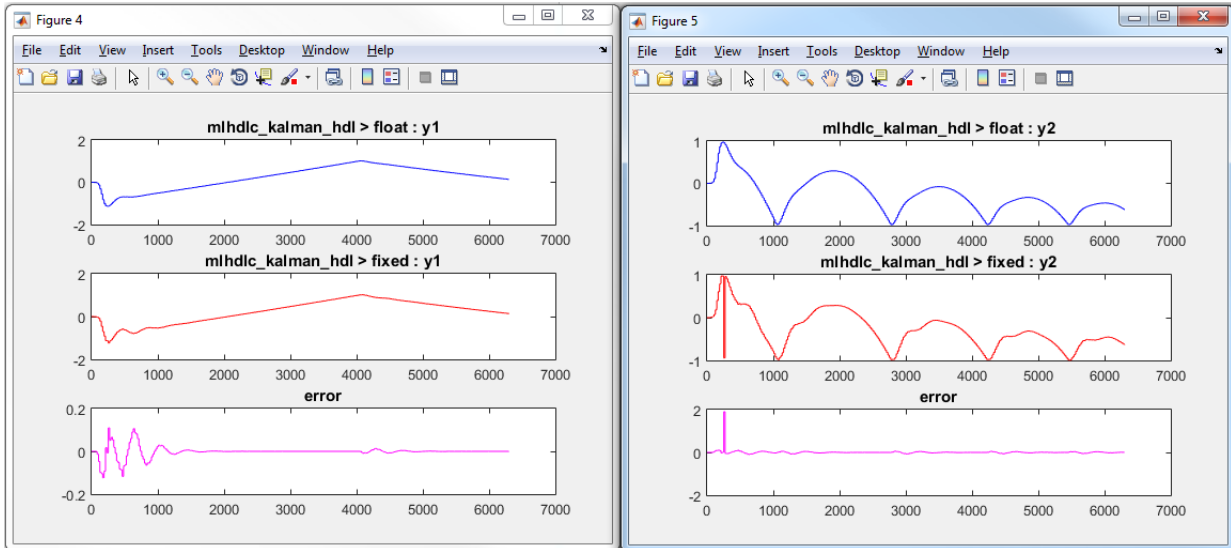
- 1 Click on 'Validate Types' to apply computed fixed-point types.
- 2 Next choose the option 'Log inputs and outputs for comparison plots' and then click on the 'Test Numerics' to rerun the testbench on the fixed-point code.

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



Having chosen comparison plots option you will see additional plots that compare the floating and fixed point simulation results for each output variable.

Examine the error graph for each output variable. It is very high for this particular design.



Iterate on the Results

One way to reduce the error is to increase 'Default word length' and repeat the fixed-point conversion.

In this example design, when a word length of 22 bits is chosen there is a lot of truncation error when representing the precision. More bits are required to the right of the binary point to reduce the truncation errors.

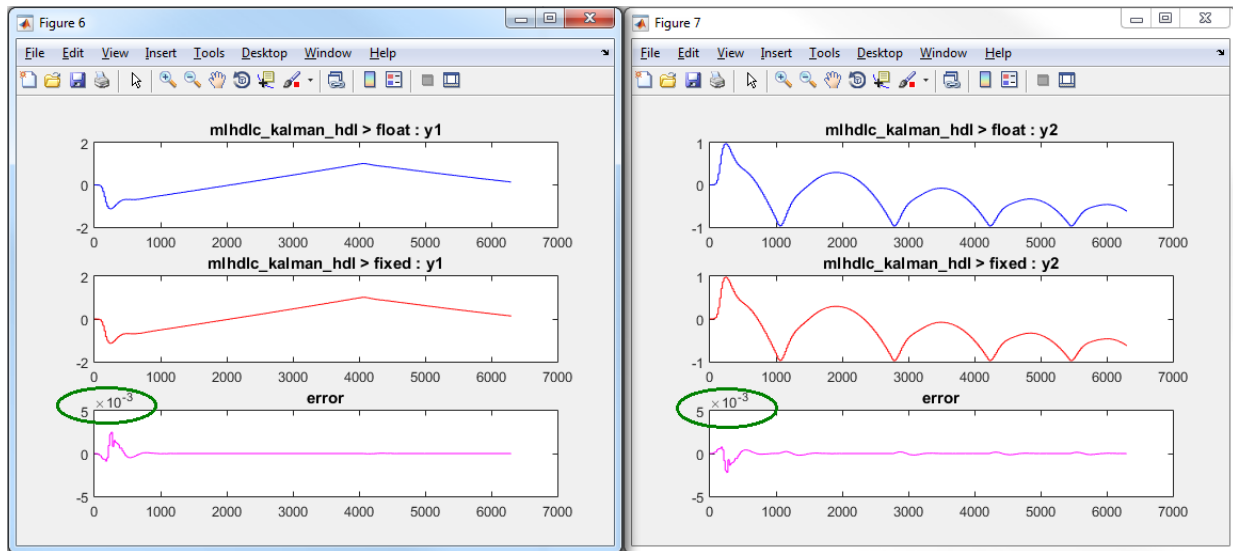
Let us now increase the default word length to 28 bits and repeat the type proposal and validation steps.

- 1 Select a 'Default word length' of 28.

Changing default word length automatically triggers the type proposal step and new fixed-point types are proposed based on the new word length setting. Also notice that type validation needs to be rerun and numerics need to be verified again.

- 1 Click on 'Validate Types'.
- 2 Click on 'Test Numerics' to rerun the testbench on the fixed-point code.

Once these steps are complete, re-examine the comparison plots and notice that the error is now roughly three orders of magnitude smaller.



Clean up the Generated Files

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

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matl...
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

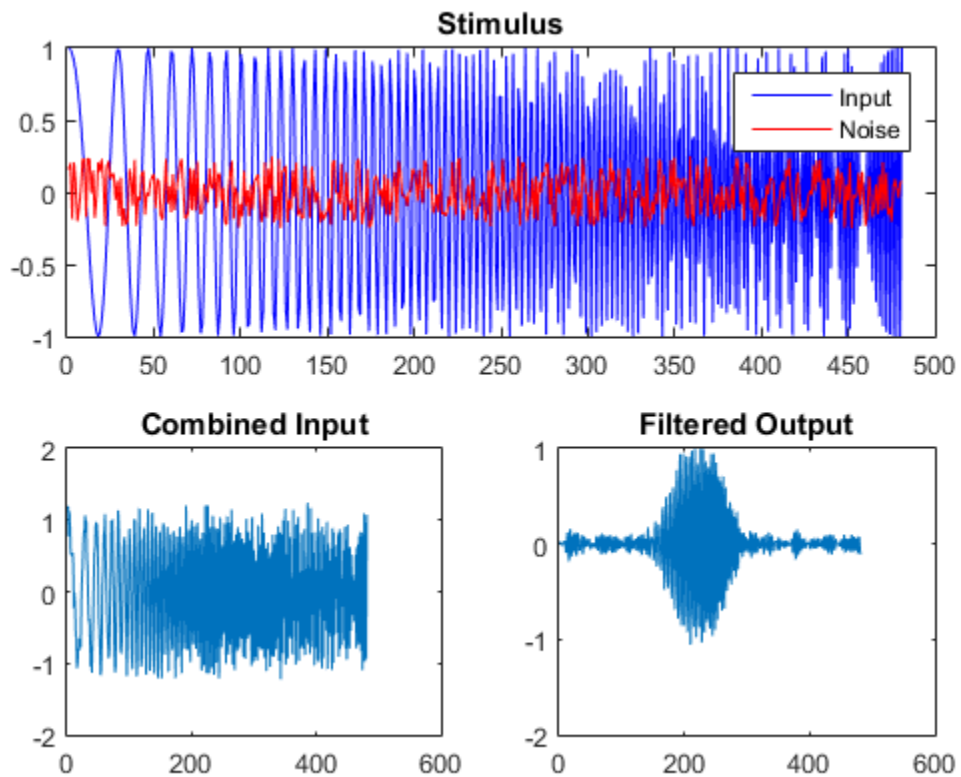
Executing the following lines of code copies the necessary example files into a temporary folder.

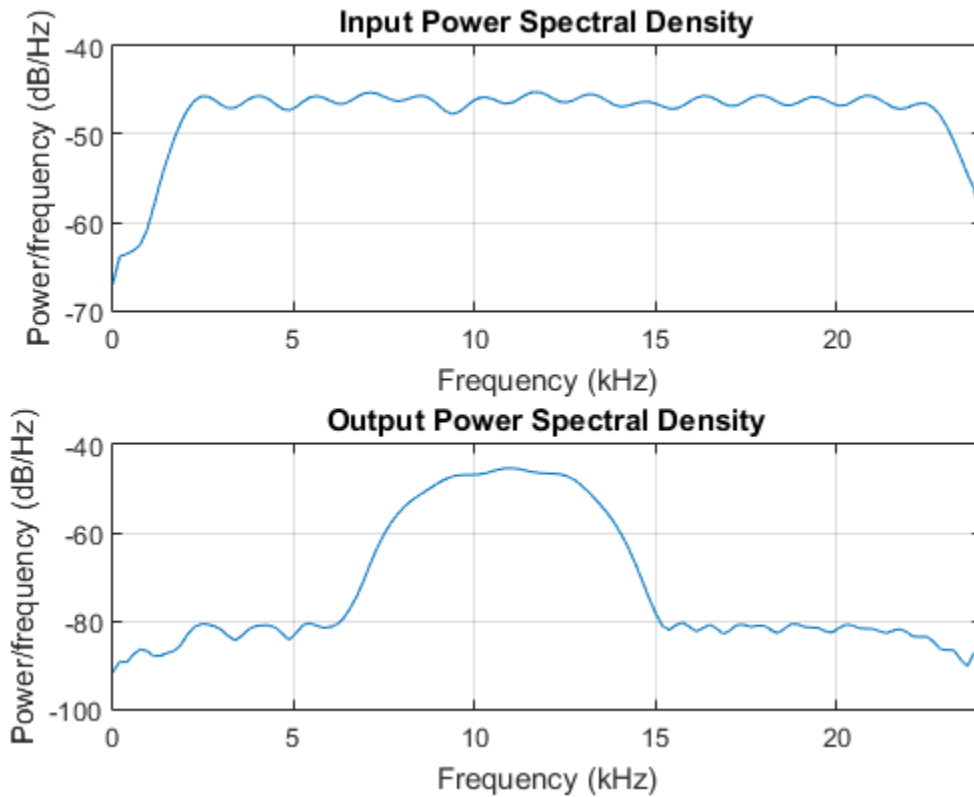
```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fix'];  
  
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);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

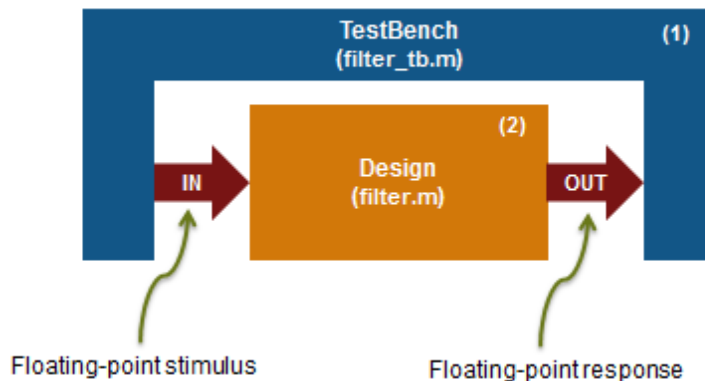
Perform the following tasks in preparation for the fixed-point code generation step:

- 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:

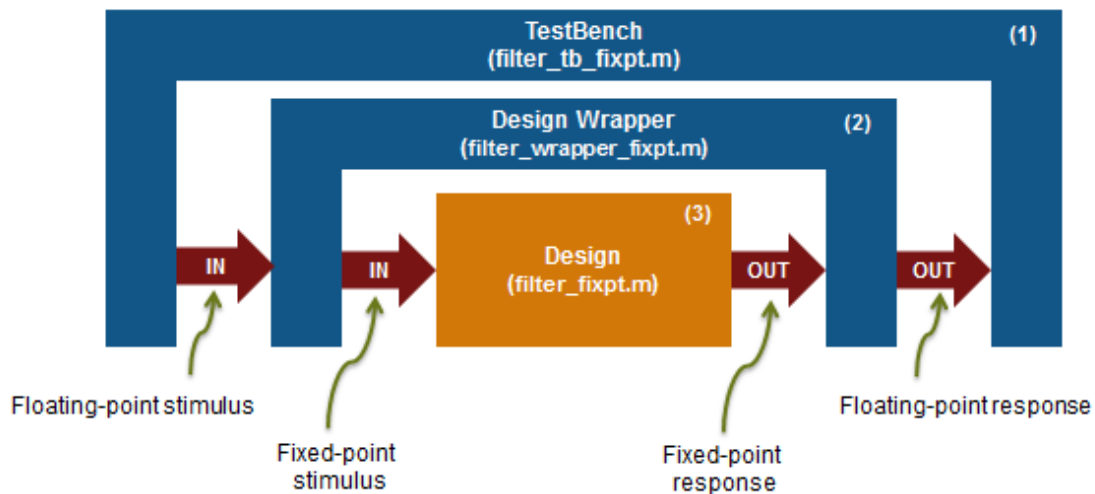
- The testbench 'mlhdlc_filter_tb.m' (1) must be a script or a function with no inputs.
- The design 'mlhdlc_filter.m' (2) must be a function.
- There must be at least one call to the design from the testbench. All call sites contribute when determining the proposed fixed-point types.

- 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

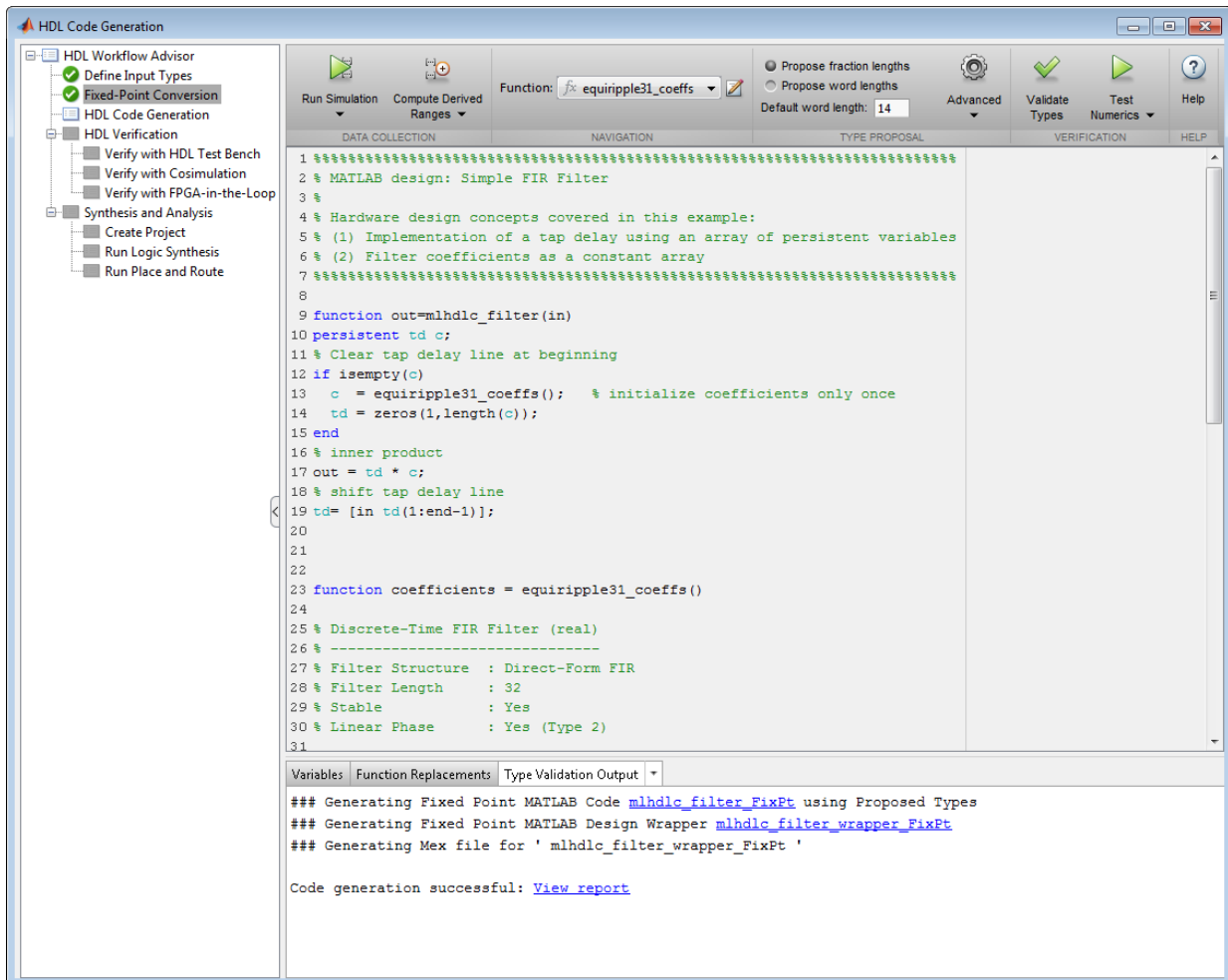
During the type validation step, fixed-point code is generated for this design and compiled to verify that there are no errors when applying the types. The output files will have the following structure.



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.
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

Advanced Type Proposal Settings	Values	Description
		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.

fimath Settings	Values	Description
Rounding method	Ceiling	Specify the fimath properties for the generated fixed-point data types.
	Convergent	

fimath Settings	Values	Description
	Floor (default)	The default fixed-point math properties use the Floor rounding and Wrap overflow. These settings generate the most efficient code but might cause problems with overflow.
	Nearest	
	Round	
	Zero	
Overflow action	Saturate	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.
	Wrap (default)	
Product mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Product word length	32 (default) any positive integer	Word length, in bits, of the product data type
Sum word length	32 (default) any positive integer	Word length, in bits, of the sum data type

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.

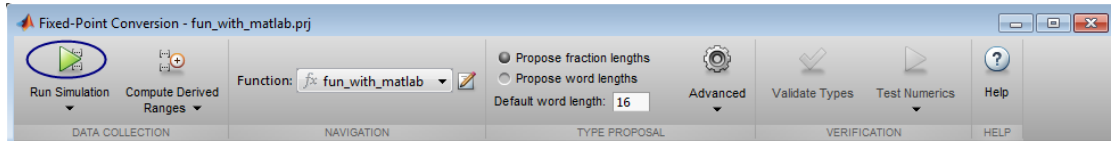
Plotting and Reporting Settings	Values	Description
Custom plot function	Empty string	Specify the name of a custom plot function to use for comparison plots.
Plot with Simulation Data Inspector	No (default)	

Plotting and Reporting Settings	Values	Description
	Yes	Specify whether to use the Simulation Data Inspector for comparison plots.
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:

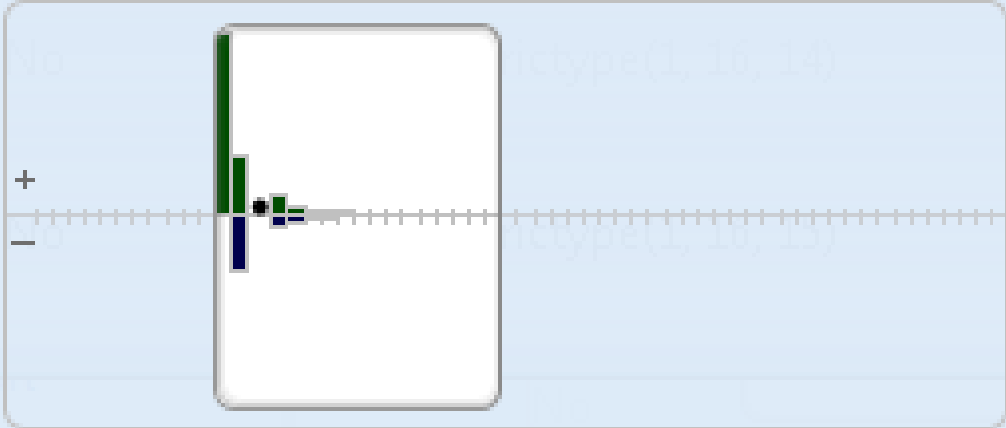
- 1 In the Fixed-Point Conversion window, click **Run Simulation** and select **Log data for histogram**, and then click the Run Simulation button.



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.

Whole Number	Proposed Type
No	numerictype(1, 16, 14)



Sim values covered **99%**

Supported range **-2 : 1.9999**

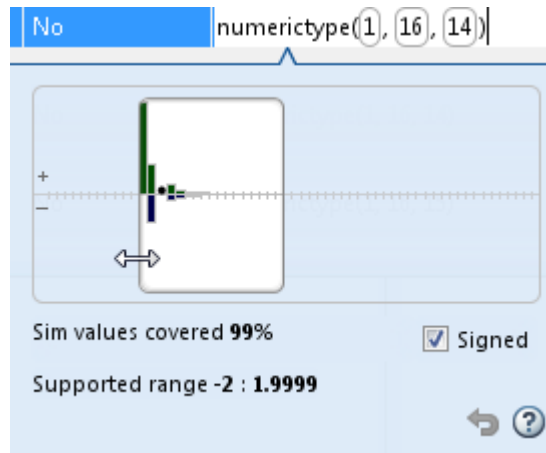
Signed

↶ ?


- 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

On the **Convert to Fixed Point** page of the app, you can view information about the variables in the MATLAB functions. To view information about the variables that you select in the **Source Code** pane, use the **Variables** tab or place your cursor over a variable in the code window. For more information, see “Viewing Variables” on page 3-51.

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**. The app highlights occurrences of the variable in the code.

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 3-42.

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 3-42.

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

- **Whole Number**

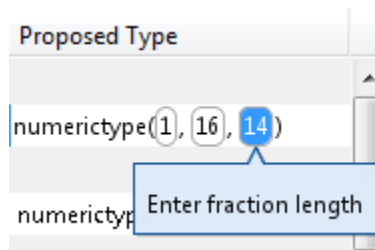
The app uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

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

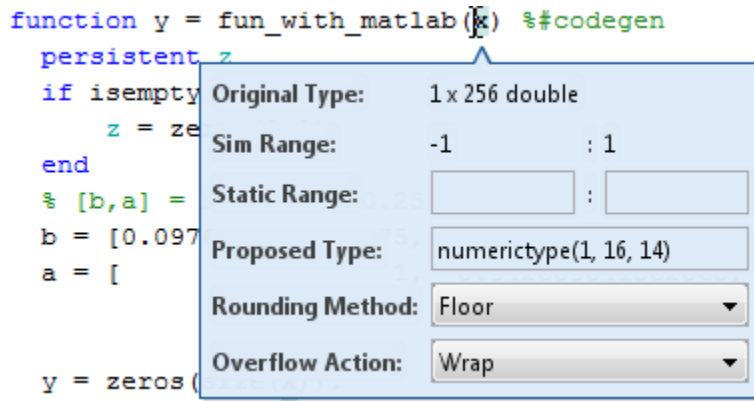
- **Proposed Type**

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 **ProposedType** 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 3-57.

Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select **Reset entire table**.

Simulation Output	Variables	Function Replacements						
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type	
<ul style="list-style-type: none"> Input <ul style="list-style-type: none"> x: 1 x 256 double, Sim Min: -1, Sim Max: 1, Proposed Type: numerictype(1, 16, 14) Output <ul style="list-style-type: none"> y: 1 x 256 double, Sim Min: -0.97, Sim Max: 1.06, Proposed Type: numerictype(1, 16, 14) Persistent <ul style="list-style-type: none"> z: 2 x 1 double, Sim Min: -0.89, Sim Max: 0.96, Proposed Type: numerictype(1, 16, 15) Local <ul style="list-style-type: none"> b: 1 x 3 double, Sim Min: 0.1, Sim Max: 0.2, Proposed Type: numerictype(0, 16, 18) a: 1 x 3 double, Sim Min: -0.94, Sim Max: 1, Proposed Type: numerictype(1, 16, 14) i: double, Sim Min: 1, Sim Max: 256, Proposed Type: numerictype(0, 9, 0) 								

A context menu is shown over the 'z' row with the following options:

- Copy sim range
- Copy sim ranges for all top-level inputs
- Copy sim ranges for all persistent variables
- Copy sim ranges for all global variables
- Clear this static range
- Clear all manually entered static ranges
- 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.

Simulation Output	Variables	Function Replacements	Static Min	Static Max	Whole Number	Decimal	Type
<div style="display: flex; justify-content: space-between;"> ▲ Input Static Min </div>							
x	1×256 double		-1	1			type(1, 16, 14)
<div style="display: flex; justify-content: space-between;"> ▲ Output Static Max </div>							
y	1×256 double		-0.97	1.06			type(1, 16, 14)
<div style="display: flex; justify-content: space-between;"> ▲ Persistent Reset entire table </div>							

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 3-43
“Automated Fixed-Point Conversion Capabilities” on page 3-43
“Code Coverage” on page 3-45
“Proposing Data Types” on page 3-48
“Locking Proposed Data Types” on page 3-50
“Viewing Functions” on page 3-50
“Viewing Variables” on page 3-51
“Histogram” on page 3-57
“Function Replacements” on page 3-59
“Validating Types” on page 3-60
“Testing Numerics” on page 3-60
“Detecting Overflows” on page 3-60

License Requirements

Fixed-point conversion requires the following licenses:

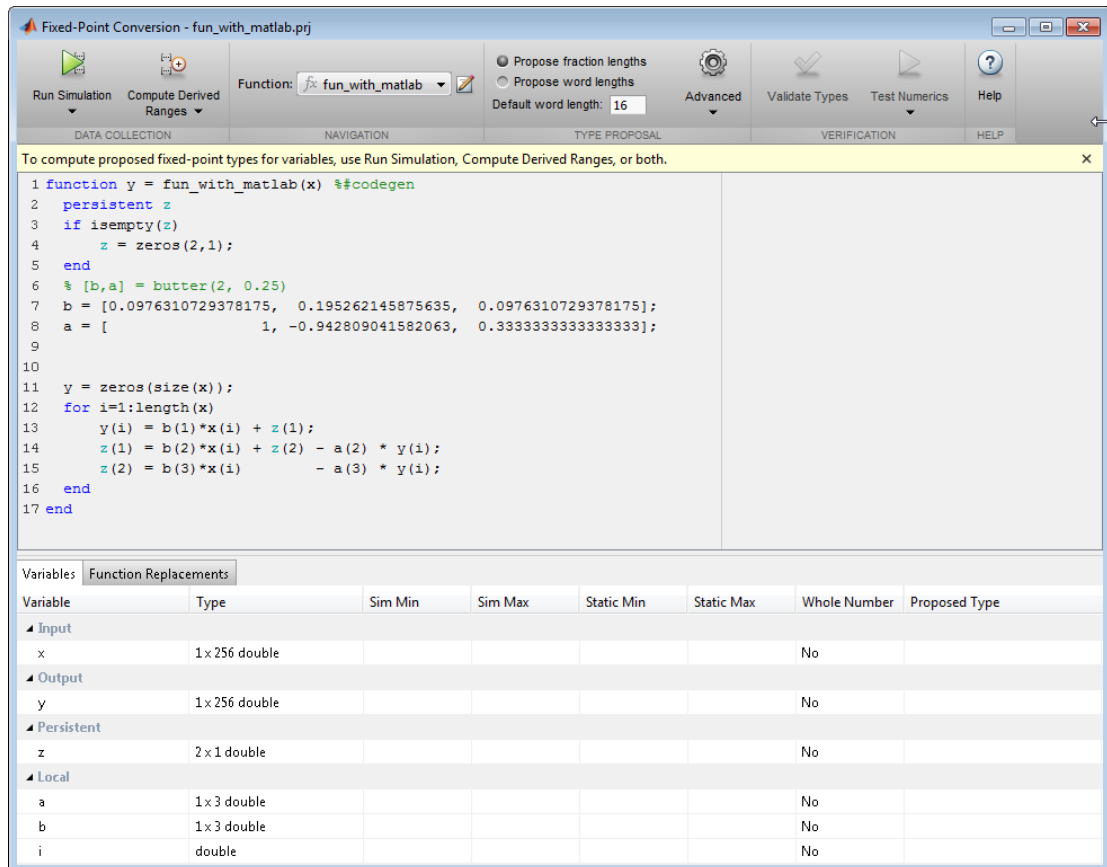
- Fixed-Point Designer
- MATLAB Coder™

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 3-50.

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



During fixed-point conversion, you can:

- Verify that your test files cover the full intended operating range of your algorithm using code coverage results.
- Propose fraction lengths based on default word lengths.
- Propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.
- Validate that you can build your project with the proposed data types.
- Test numerics by running the test bench with the fixed-point types applied.

- View a histogram of bits used by each variable.
- Detect overflows.

Code Coverage

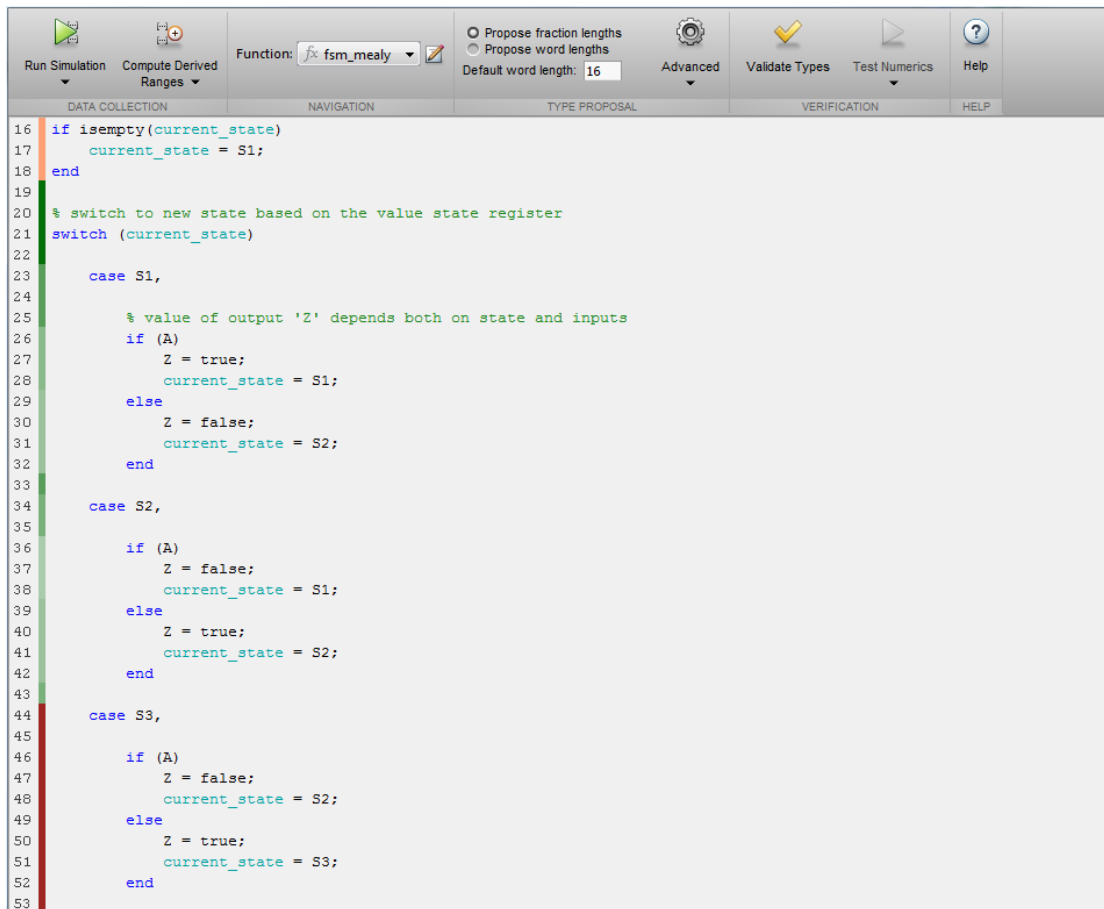
By default, the Fixed-Point Conversion tool shows code coverage results. Your test files should exercise the algorithm over its full operating range so that the simulation ranges are accurate. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want. Reviewing code coverage results helps you verify that your test file is exercising the algorithm adequately. If the code coverage is inadequate, modify the test file 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.

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 might speed up simulation. To turn off code coverage, in the Fixed-Point Conversion tool:

- 1 Click **Run Simulation**.
- 2 Clear **Show code coverage**.

The tool covers basic MATLAB control constructs and shows statement coverage for basic blocks of code. The tool displays a color-coded coverage bar to the left of the code.

Coverage Bar Color	How Often Code is Executed During Test File Simulation
Dark green	Always
Light green	Sometimes
Orange	Once
Red	Never



The screenshot displays the MATLAB Simulink interface with a code editor showing the implementation of an FSM mealy function. The code is as follows:

```
16 if isempty(current_state)
17     current_state = S1;
18 end
19
20 % switch to new state based on the value state register
21 switch (current_state)
22
23     case S1,
24
25         % value of output 'Z' depends both on state and inputs
26         if (A)
27             Z = true;
28             current_state = S1;
29         else
30             Z = false;
31             current_state = S2;
32         end
33
34     case S2,
35
36         if (A)
37             Z = false;
38             current_state = S1;
39         else
40             Z = true;
41             current_state = S2;
42         end
43
44     case S3,
45
46         if (A)
47             Z = false;
48             current_state = S2;
49         else
50             Z = true;
51             current_state = S3;
52         end
53
```

The interface includes a toolbar with icons for 'Run Simulation', 'Compute Derived Ranges', 'Function: fsm_mealy', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. Below the toolbar are tabs for 'DATA COLLECTION', 'NAVIGATION', 'TYPE PROPOSAL', 'VERIFICATION', and 'HELP'. The code editor shows a vertical coverage bar on the left side of the code, with a red bar at the top and a green bar at the bottom, indicating coverage levels for different parts of the code.

When you position your cursor over the coverage bar, the color highlighting extends over the code and the tool displays more information about how often the code is executed. For MATLAB constructs that affect control flow (if-elseif-else, switch-case, for-continue-break, return), it displays statement coverage as a percentage coverage for basic blocks inside these constructs.

The screenshot shows the MATLAB Simulink interface with the code coverage results for the `fsm_mealy` function. The code is color-coded as follows:

- Line 16: `if isempty(current_state)` - Executed once (Orange)
- Line 17: `current_state = S1;` - Executed once (Orange)
- Line 18: `end` - Executed once (Orange)
- Line 20: `% switch to new state based on the value state register` - Reached 58% of the time (Dark Green)
- Line 21: `switch (current_state)` - Reached 58% of the time (Dark Green)
- Line 22: (Blank) - Reached 58% of the time (Dark Green)
- Line 23: `case S1,` - Reached 58% of the time (Dark Green)
- Line 24: (Blank) - Reached 58% of the time (Dark Green)
- Line 25: `% value of output 'Z' depends both on state and inputs` - Reached 58% of the time (Dark Green)
- Line 26: `if (A)` - Reached 33% of the time (Light Green)
- Line 27: `Z = true;` - Reached 33% of the time (Light Green)
- Line 28: `current_state = S1;` - Reached 33% of the time (Light Green)
- Line 29: `else` - Reached 25% of the time (Light Green)
- Line 30: `Z = false;` - Reached 25% of the time (Light Green)
- Line 31: `current_state = S2;` - Reached 25% of the time (Light Green)
- Line 32: `end` - Reached 25% of the time (Light Green)
- Line 33: (Blank) - Reached 58% of the time (Dark Green)
- Line 34: `case S2,` - Reached 42% of the time (Light Green)
- Line 35: (Blank) - Reached 42% of the time (Light Green)
- Line 36: `if (A)` - Reached 17% of the time (Light Green)
- Line 37: `Z = false;` - Reached 17% of the time (Light Green)
- Line 38: `current_state = S1;` - Reached 17% of the time (Light Green)
- Line 39: `else` - Reached 25% of the time (Light Green)
- Line 40: `Z = true;` - Reached 25% of the time (Light Green)
- Line 41: `current_state = S2;` - Reached 25% of the time (Light Green)
- Line 42: `end` - Reached 25% of the time (Light Green)
- Line 43: (Blank) - Reached 42% of the time (Light Green)
- Line 44: `case S3,` - Not reached (Dark Red)
- Line 45: (Blank) - Not reached (Dark Red)
- Line 46: `if (A)` - Not reached (Dark Red)
- Line 47: `Z = false;` - Not reached (Dark Red)
- Line 48: `current_state = S2;` - Not reached (Dark Red)
- Line 49: `else` - Not reached (Dark Red)
- Line 50: `Z = true;` - Not reached (Dark Red)
- Line 51: `current_state = S3;` - Not reached (Dark Red)
- Line 52: `end` - Not reached (Dark Red)
- Line 53: (Blank) - Not reached (Dark Red)

To verify that your test file is testing your algorithm over the intended operating range, review the code coverage results and take action as described in the following table.

Coverage Bar Color	Action Required
Dark green	None
Light green	Review percentage coverage and verify that it is reasonable based on your algorithm. If there are areas of code that you expect to be executed more frequently, modify your test file or add more test files to increase coverage.

Coverage Bar Color	Action Required
Orange	This is expected behavior for initialization code, for example, the initialization of persistent variables. For other cases, verify that this behavior is reasonable for your algorithm. If there are areas of code that you expect to be executed more frequently, modify your test file or add more test files to increase coverage.
Red	If the code that is not executed is an error condition, this is acceptable behavior. If the code should be executed, modify the test file or add another test file to extend coverage. If the code is written conservatively and has upper and lower boundary limits and you cannot modify the test file to reach this code, add static minimum and maximum values (see “Computing Derived Ranges”).

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 3-50.

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 3-59.

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 3-57.

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 3-50.

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:

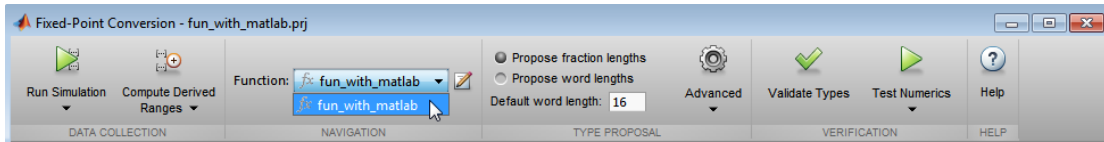
- Manually setting a proposed data type in the Fixed-Point Conversion tool.
- Right-clicking a type proposed by the tool and selecting **Lock computed value**.

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.

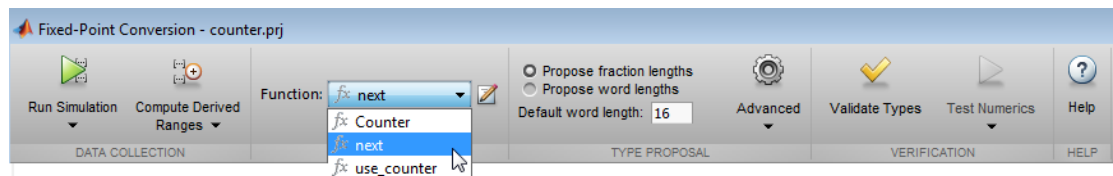
You can also view and edit variable information in the code pane by placing your cursor over a variable name.

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 MATLAB Fixed-Point Designer interface. The top toolbar includes buttons for 'Run Simulation', 'Compute Derived Ranges', 'Function: f: next', 'Propose fraction lengths', 'Propose word lengths', 'Default word length: 16', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. Below the toolbar, the code for the 'next' function is displayed in a text editor. The code is as follows:

```

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

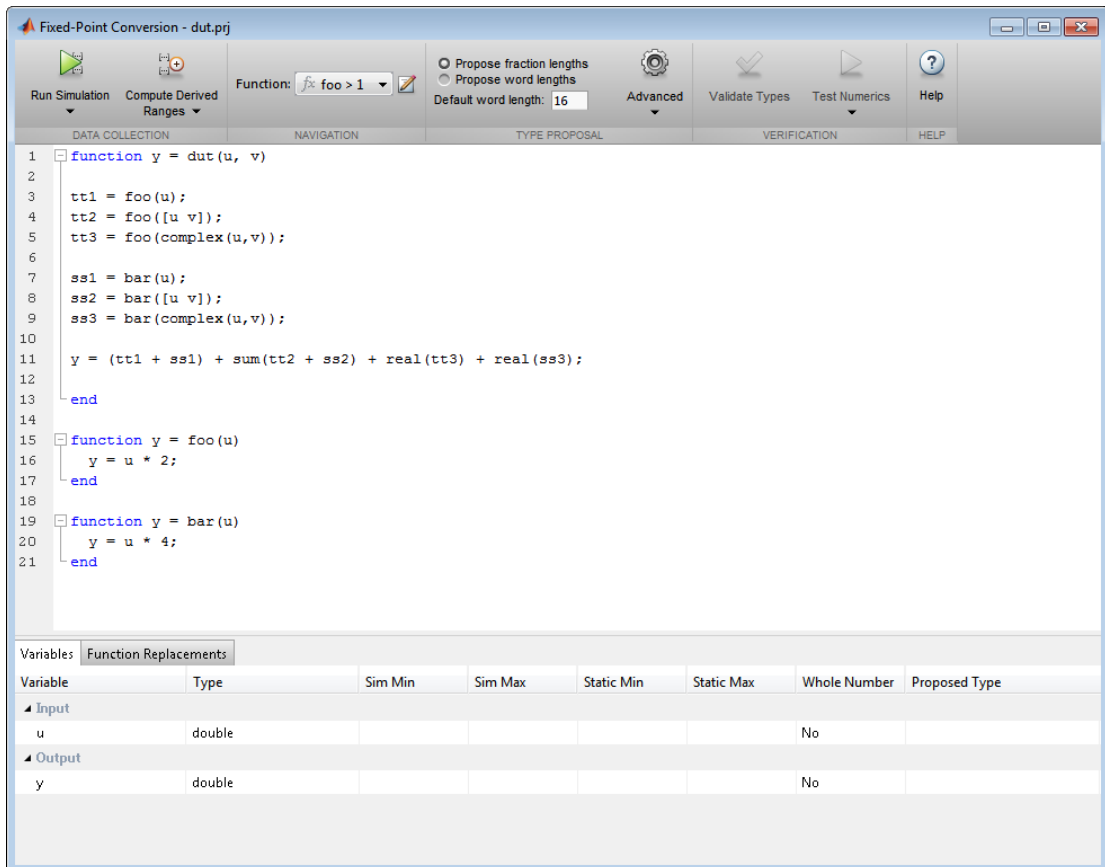
function y = bar(u)
    y = u * 4;
end
```

The screenshot shows the 'Fixed-Point Conversion - dut.prj' application window. The main area is a code editor displaying MATLAB code for functions 'dut', 'foo', and 'bar'. A dropdown menu is open over the 'Function:' field, showing specializations: 'dut', 'foo > 1', 'foo > 2', 'foo > 3', 'bar > 1', 'bar > 2', and 'bar > 3'. The 'Function:' field currently shows 'dut'. Below the code editor is a 'Variables' table with a 'Function Replacements' tab selected.

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.

3 Fixed-Point Conversion



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 `f00 > 1` is named `f00_s1`.

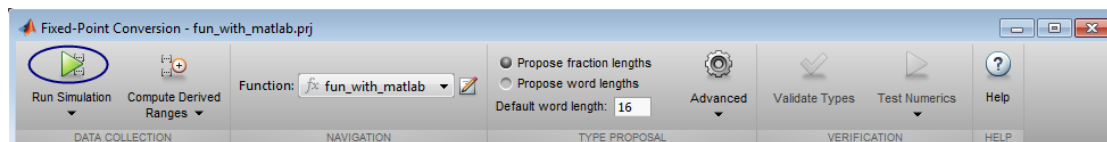

```

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
10
11  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full
12
13  tt1 = fi(foo_s1(u), 0, 5, 0, fm);
14  tt2 = fi(foo_s2([fi(u, 0, 5, 0, fm) v]), 0, 6, 0, fm);
15  tt3 = fi(foo_s3(complex(u,v)), 0, 6, 0, fm);
16
17  ss1 = fi(bar_s1(u), 0, 6, 0, fm);
18  ss2 = fi(bar_s2([fi(u, 0, 5, 0, fm) v]), 0, 7, 0, fm);
19  ss3 = fi(bar_s3(complex(u,v)), 0, 7, 0, fm);
20
21  y = fi((tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3), 0, 9, 0, fm);
22
23  end
24
25  function y = foo_s1(u)
26  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full
27  y = fi(u * fi(2, 0, 2, 0, fm), 0, 5, 0, fm);
28  end
29
30  function y = foo_s2(u)
31  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full
32  y = fi(u * fi(2, 0, 2, 0, fm), 0, 6, 0, fm);
33  end
34

```

Histogram

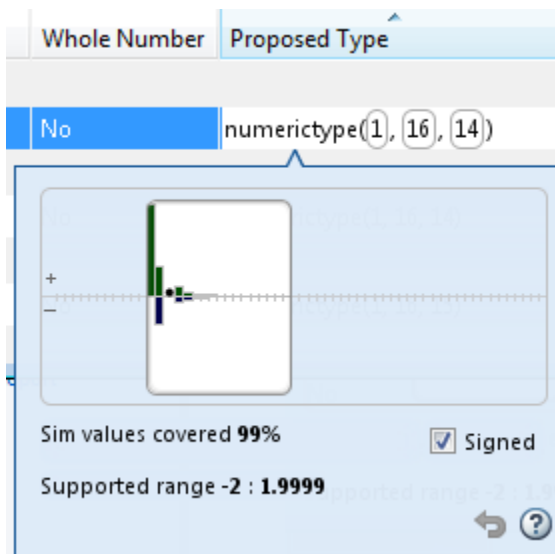
To log data for histograms, in the Fixed-Point Conversion window, click **Run Simulation** and select **Log data for histogram**, and then click the **Run Simulation** button.



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

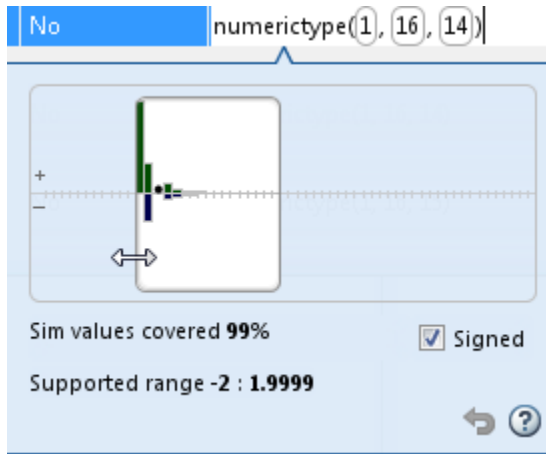
The histogram provides the range of the proposed data type and the percentage of simulation values that the proposed data type covers. The bit weights are displayed along

the X-axis, and the percentage of occurrences along the Y-axis. Each bin in the histogram corresponds to a bit in the binary word. For example, this histogram displays the range for a variable of type `numerictype(1, 16, 14)`.



You can view the effect of changing the proposed data types by:

- Dragging the edges of the bounding box in the histogram window to change the proposed data type.



- Selecting or clearing **Signed**.

To revert to the types proposed by the automatic conversion, in the histogram window,

click .

Function Replacements

If your MATLAB code uses functions that do not have fixed-point support, the tool lists these functions on the **Function Replacements** tab. You can choose to replace unsupported functions with a custom function replacement or with a lookup table.

Function or Operator	Replacement								
Variables Function Replacements Simulation Output									
Enter a function to replace Custom Function ▾ + -									
▲ Custom Function foo	Function Name foo_fixedpoint								
▲ Lookup Table exp	<table border="1"> <thead> <tr> <th>Interpolation Method</th> <th>Design Min</th> <th>Design Max</th> <th>Number of Points</th> </tr> </thead> <tbody> <tr> <td>None</td> <td>Auto</td> <td>Auto</td> <td>1000</td> </tr> </tbody> </table>	Interpolation Method	Design Min	Design Max	Number of Points	None	Auto	Auto	1000
Interpolation Method	Design Min	Design Max	Number of Points						
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”

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”.

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 (for most platforms, a default C compiler is supplied with MATLAB) For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

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-Point conversion.
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % convert from cell to matrix
    floatVals = cell2mat(floatVals);
    fixedVals = cell2mat(fixedVals);

    % 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;

```

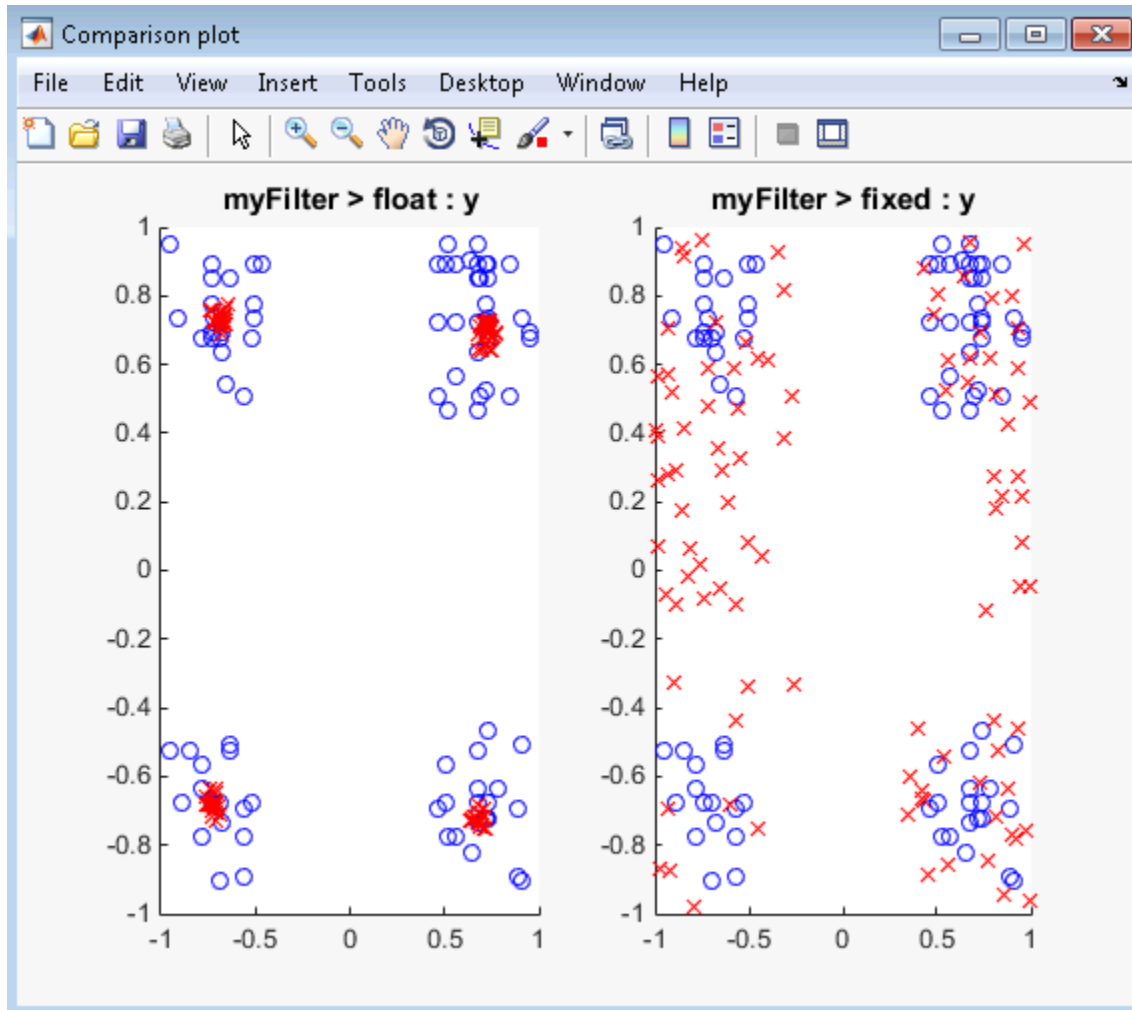
Convert to Fixed Point

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

```
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The conversion process generates fixed-point code using a default word length of 16 and then runs a fixed-point simulation by running the `myFilterTest.m` function and calling the fixed-point version of `myFilter.m`.

Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the conversion process uses this function to generate the comparison plot.

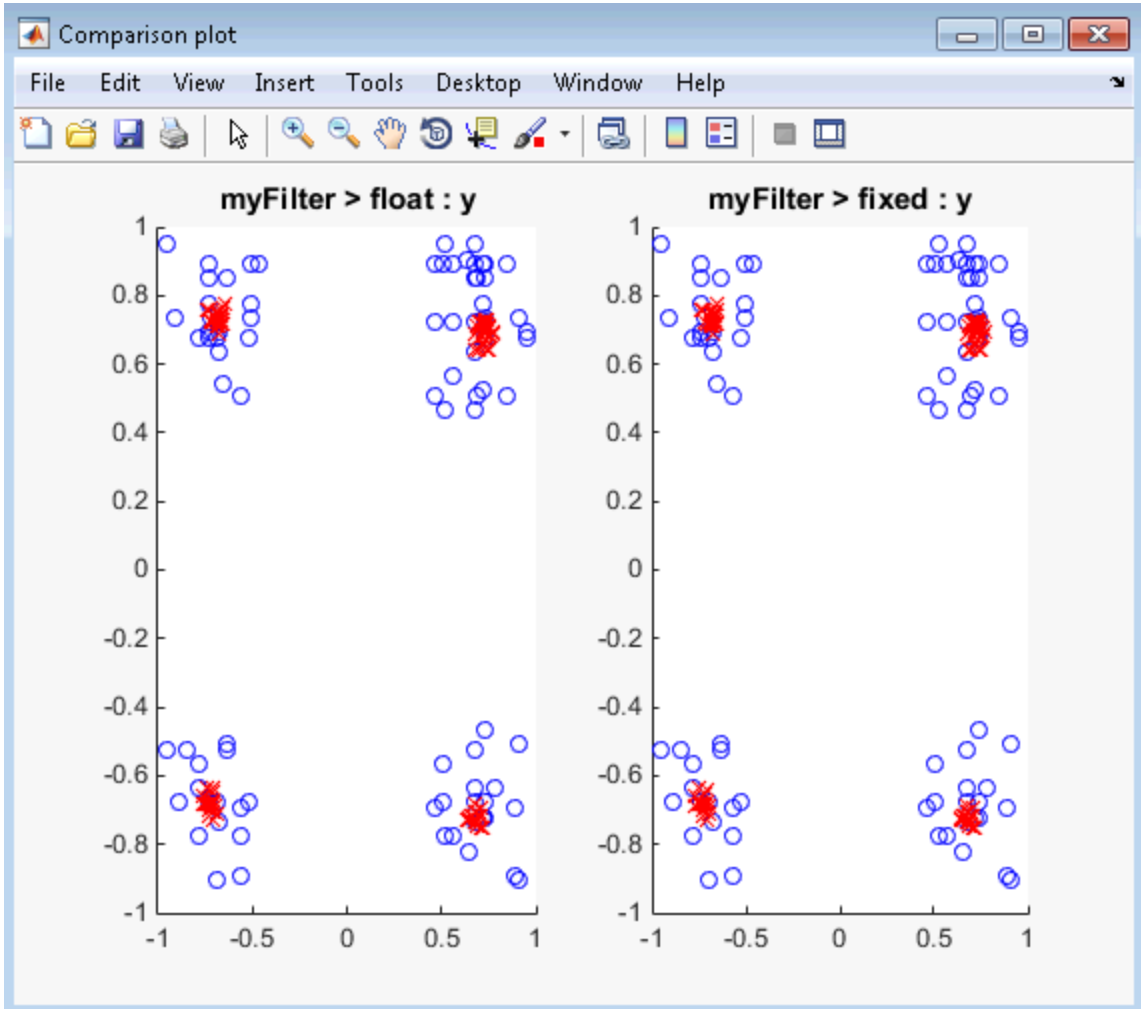


The plot shows that the fixed-point results do not closely match the floating-point results.

Increase the word length to 24 and then convert to fixed point again.

```
fxptcfg.DefaultWordLength = 24;  
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The increased word length improved the results. This time, the plot shows that the fixed-point results match the floating-point results.



Inspecting Data Using the Simulation Data Inspector

In this section...

“What Is the Simulation Data Inspector?” on page 3-70

“Import Logged Data” on page 3-70

“Export Logged Data” on page 3-70

“Group Signals” on page 3-71

“Run Options” on page 3-71

“Create Report” on page 3-71

“Comparison Options” on page 3-71

“Enabling Plotting Using the Simulation Data Inspector” on page 3-71

“Save and Load Simulation Data Inspector Sessions” on page 3-72

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 to top**.

- Specify a Run Naming Rule

To specify run naming rules, in the Simulation Data Inspector toolbar, click **Run Configuration**.

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”.

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 3-73

“From the Command Line” on page 3-73

From the UI

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point logged input and output data. In the Fixed-Point Conversion tool:

- 1 Click **Advanced**.
- 2 In the Advanced Settings dialog box, set **Plot with Simulation Data Inspector** to **Yes**.
- 3 At the Test Numerics stage in the conversion process, click **Test Numerics**, select **Log inputs and outputs for comparison plots**, and then click



For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges”.

From the Command Line

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point input and output data logged using the function. At the MATLAB command line:

- 1 Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'dti_test';
```

- 2 Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

```
fixptcfg.TestNumerics = true;
```

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

- 3 Generate fixed-point MATLAB code using `codegen`.

```
codegen -float2fixed fixptcfg -config cfg dti
```

For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges”.

Replacing Functions Using Lookup Table Approximations

The software provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. These functions must be on the MATLAB path.

You can use this capability to handle functions that are not supported for fixed point and to replace your own custom functions. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. You can control the interpolation method and number of points in the lookup table. By adjusting these settings, you can tune the behavior of replacement function to match the behavior of the original function as closely as possible.

The fixed-point conversion process generates one lookup table approximation per call site of the function that needs replacement.

To use lookup table approximations, see:

- `coder.approximation`
- “Replace the exp Function with a Lookup Table”
- “Replace a Custom Function with a Lookup Table”

Replace a Custom Function with a Lookup Table

In this section...
“From the UI” on page 3-76
“From the Command Line” on page 3-84

From the UI

This example shows how to replace a custom function with a lookup table approximation function using the Fixed-Point Conversion tool.

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`, that 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 MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this example.

- 2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `custom_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new custom_project.prj
```

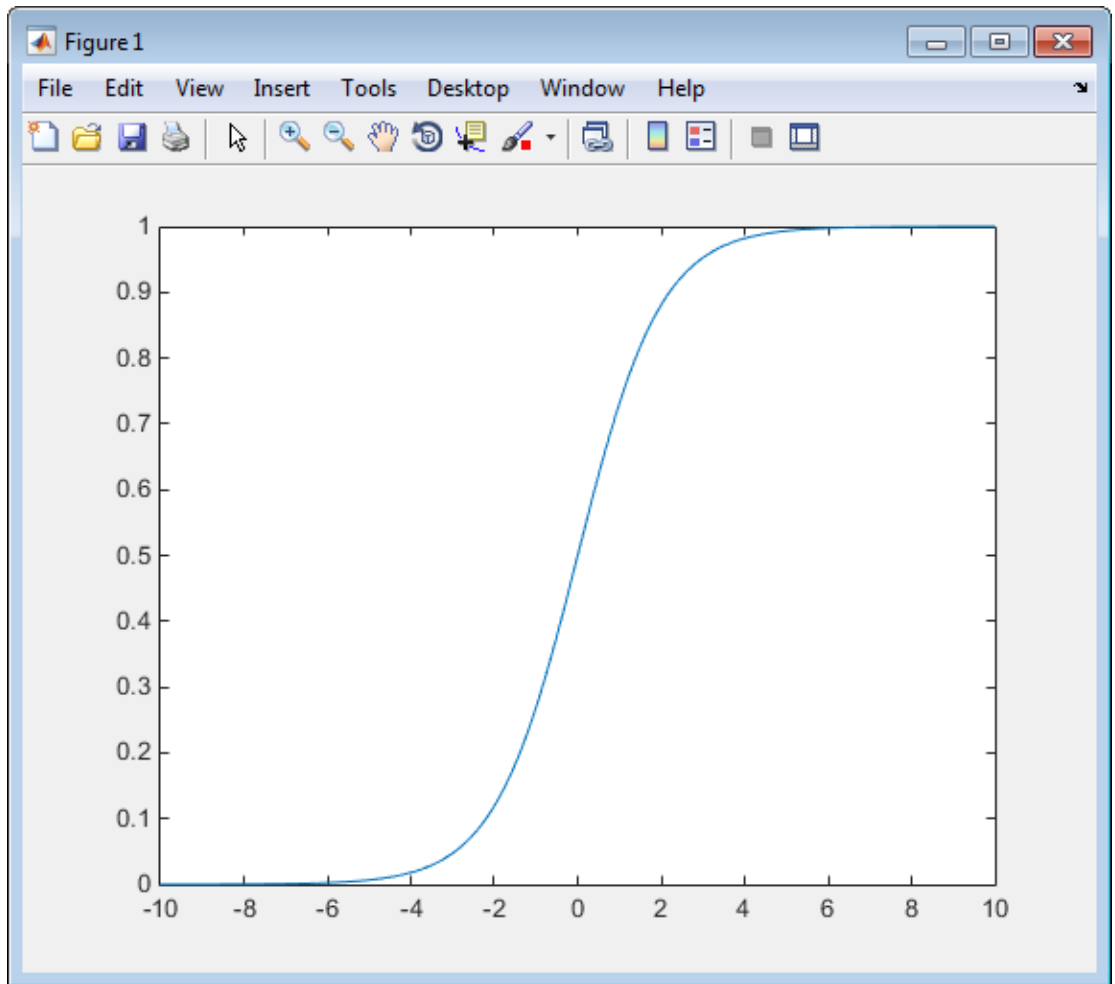
By default, the project opens in the MATLAB workspace.

- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `call_custom_fcn.m` and then click **OK** to add the file to the project.

Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `custom_test` as a test file and then click **Run**.

The test file runs and plots the output.



- 3 MATLAB Coder determines from the test file that `x` is a scalar double.
- 4 In the Autodefine Input Types dialog box, click **Use These Types**.

MATLAB Coder sets the type of `x` to `double(1x1)`.

Fixed-Point Conversion

- 1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.

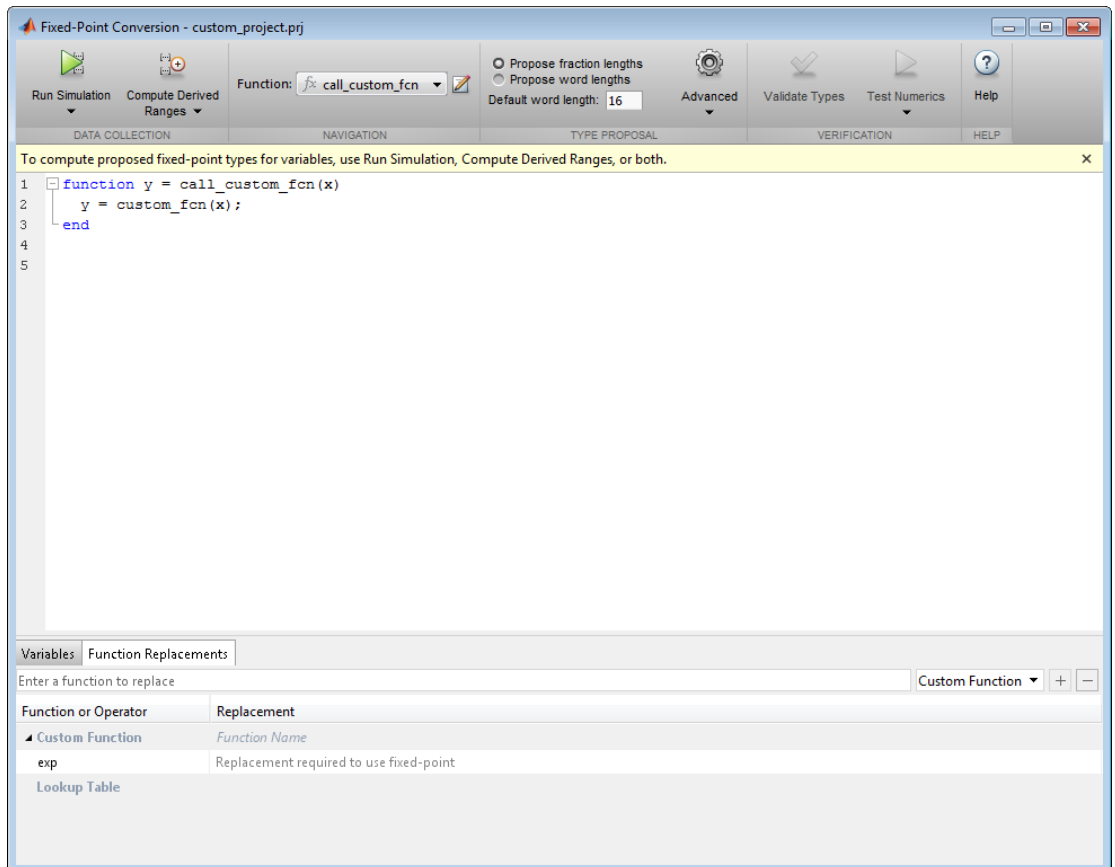
The project indicates that you must first define the fixed-point data types.

- 2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

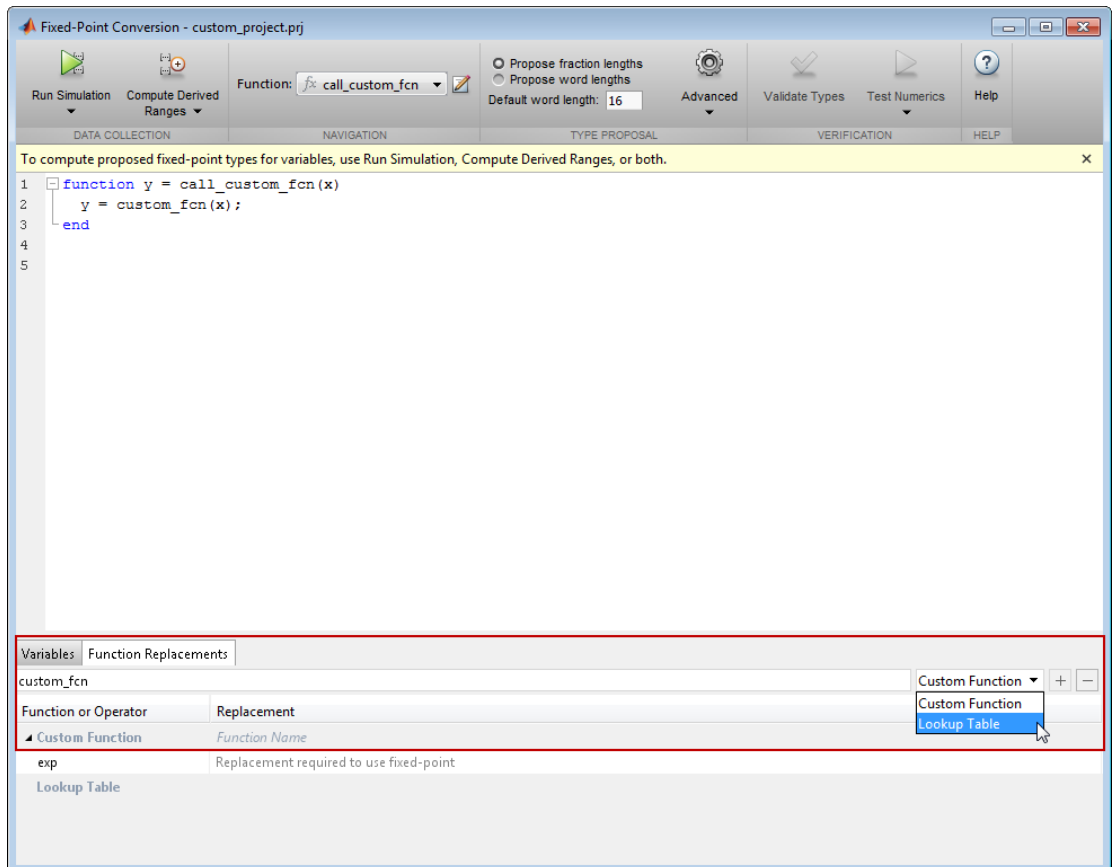
The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code.

- 3 Select the **Function Replacements** tab.

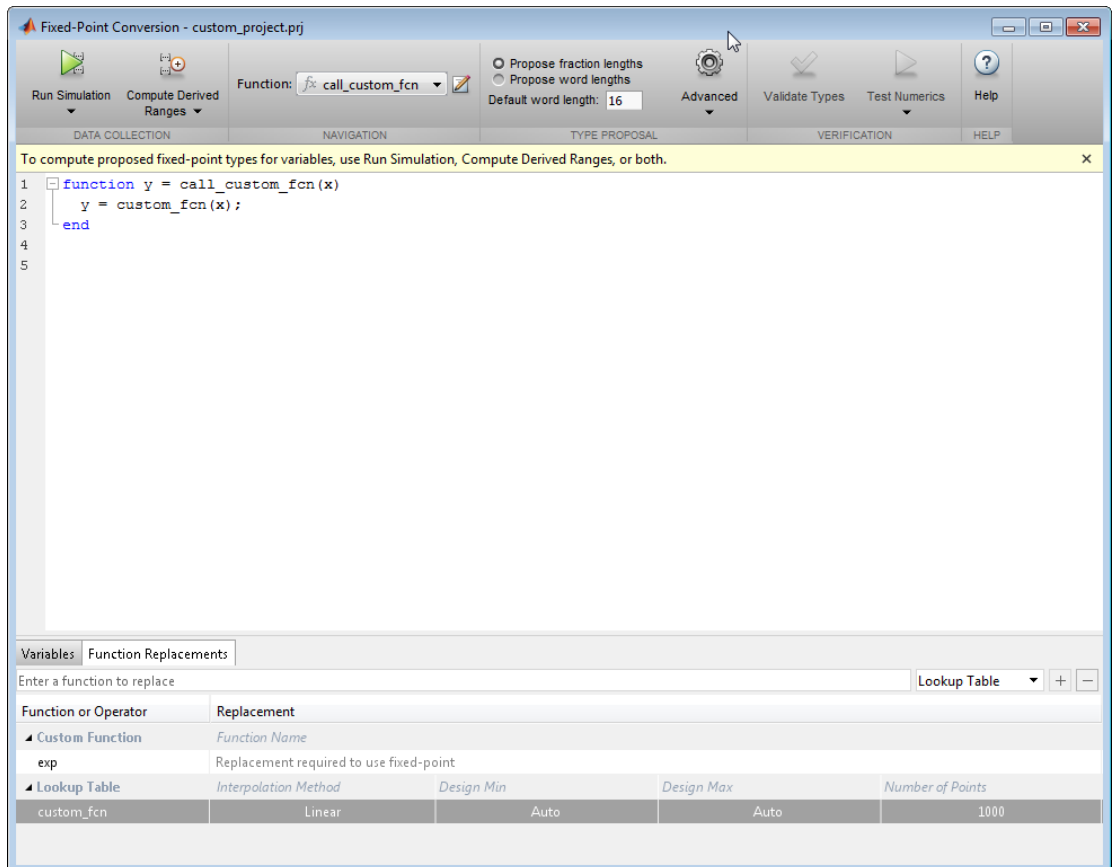
3 Fixed-Point Conversion



- 4 Enter the name of the function to replace, `custom_fcn`, select **Lookup Table**, and then click **+**.



The tool adds `custom_fcn` to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation, 1000 points, and the design minimum and maximum values that the app detects by either running a simulation or computing derived ranges.

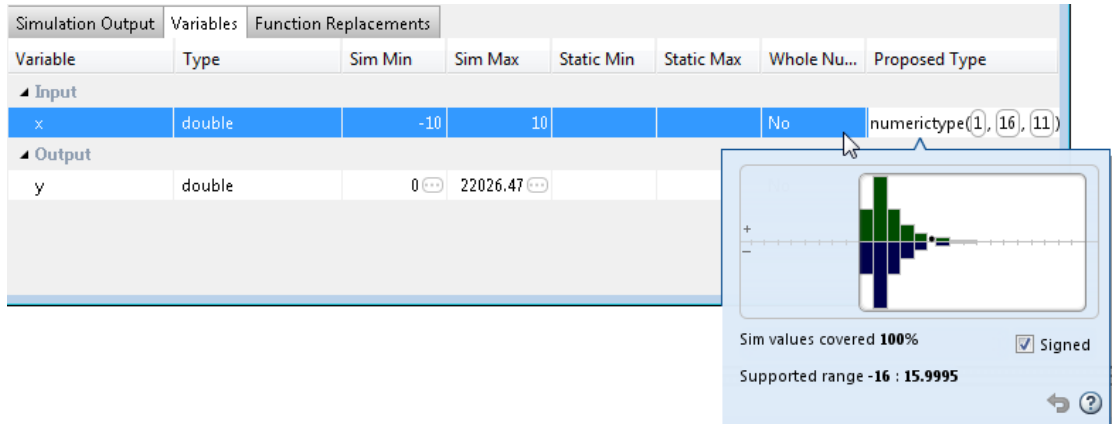


- 5 Click **Run Simulation**, select **Log data for histogram** and verify that the `custom_test` file is selected as a test file to run.
- 6 Click the **Run Simulation** button.

The simulation runs and the tool displays simulation minimum and maximum ranges 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. The **Validate Types** option is now enabled.

- 7 Examine the proposed types and verify that they 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 covered by the proposed data type.



- 8 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types and generates a fixed-point code, `call_custom_fcn_fixpt`.

- 9 On the Type Validation Output tab, click the `call_custom_fcn_fixpt` link to view the generated fixed-point code.

The conversion process generates a lookup table approximation, `custom_fcn1`, for the `custom_fcn` function.

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 = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', ...
               'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, ...
               'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    y = fi(custom_fcn1(x), 0, 16, 16, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function.

From the Command Line

This example shows how to replace a custom function with a lookup table approximation function using the function.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB) For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

Create a MATLAB function, `custom_fcn.m`. This is the function that you want to replace.

```
function y = custom_fcn(x)
    y = 1./(1+exp(-x));
end
```

Create a wrapper function that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

Create a test file, `custom_test.m`, that uses `call_custom_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = call_custom_fcn( x(itr) );
end
```

```
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, `custom_fcn1`, 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 = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
        'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, ...
        'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    y = fi(custom_fcn1(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”

More About

- “Replacing Functions Using Lookup Table Approximations”

Replace the exp Function with a Lookup Table

In this section...

“From the UI” on page 3-87

“From the Command Line” on page 3-95

From the UI

This example shows how to replace a custom function with a lookup table approximation function using the Fixed-Point Conversion tool.

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`, that 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 MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this example.

- 2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `custom_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new custom_project.prj
```

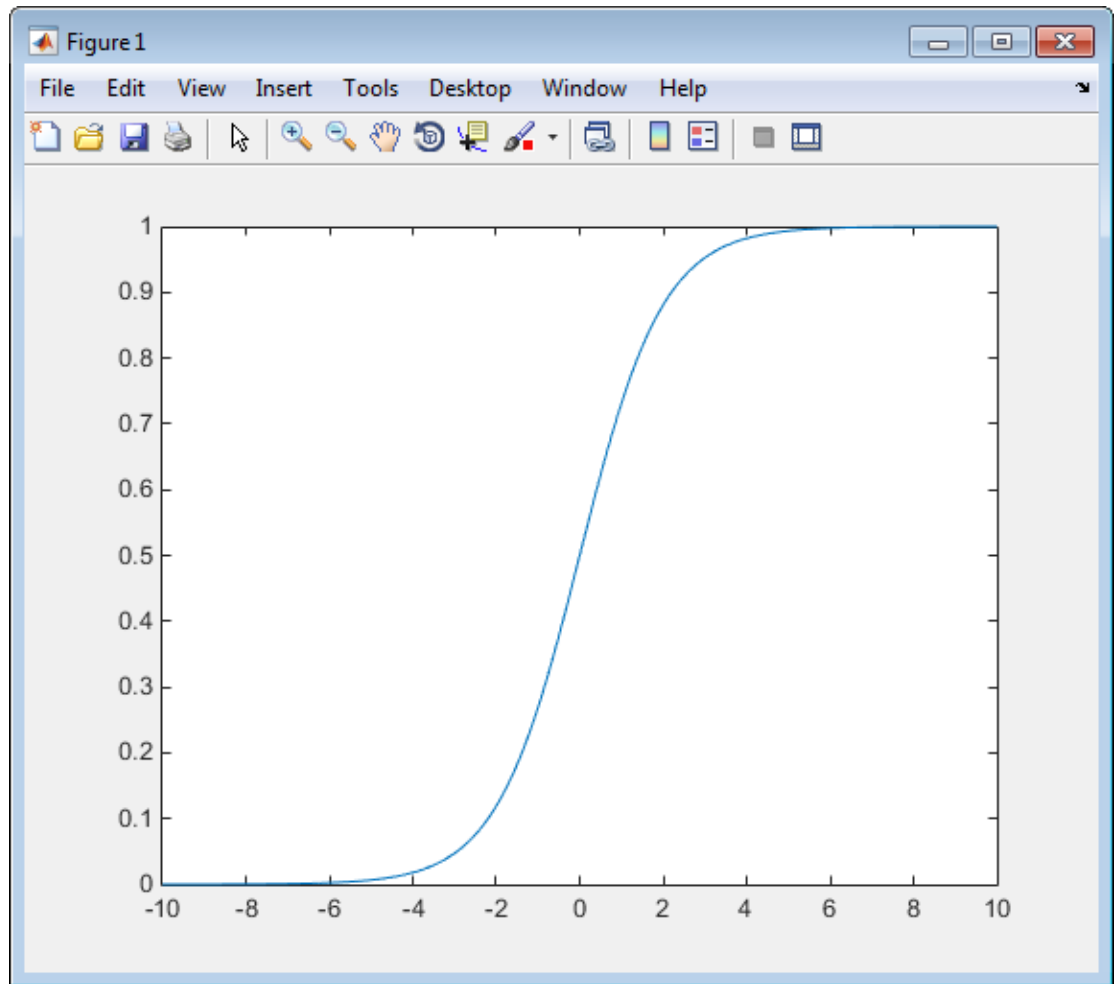
By default, the project opens in the MATLAB workspace.

- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `call_custom_fcn.m` and then click **OK** to add the file to the project.

Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `custom_test` as a test file and then click **Run**.

The test file runs and plots the output.



- 3 MATLAB Coder determines from the test file that x is a scalar double.
- 4 In the Autodefine Input Types dialog box, click **Use These Types**.

MATLAB Coder sets the type of x to `double(1x1)`.

Fixed-Point Conversion

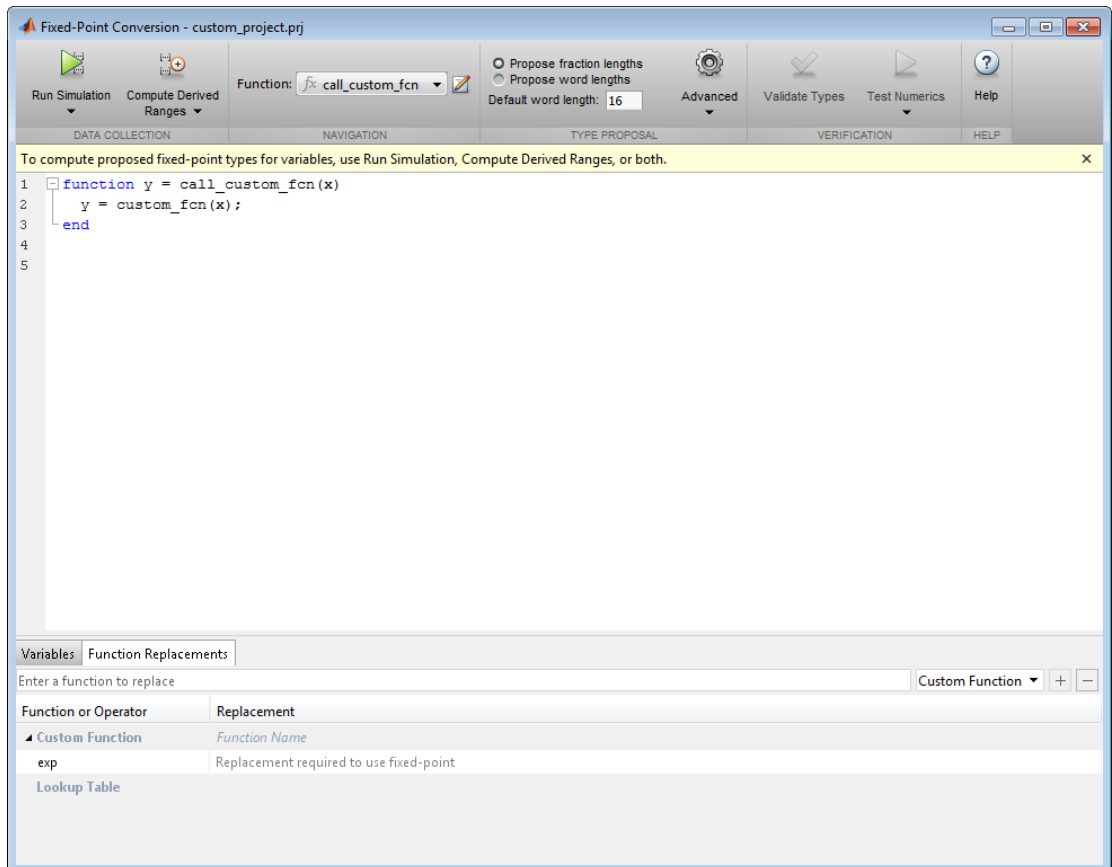
- 1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.

The project indicates that you must first define the fixed-point data types.

- 2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

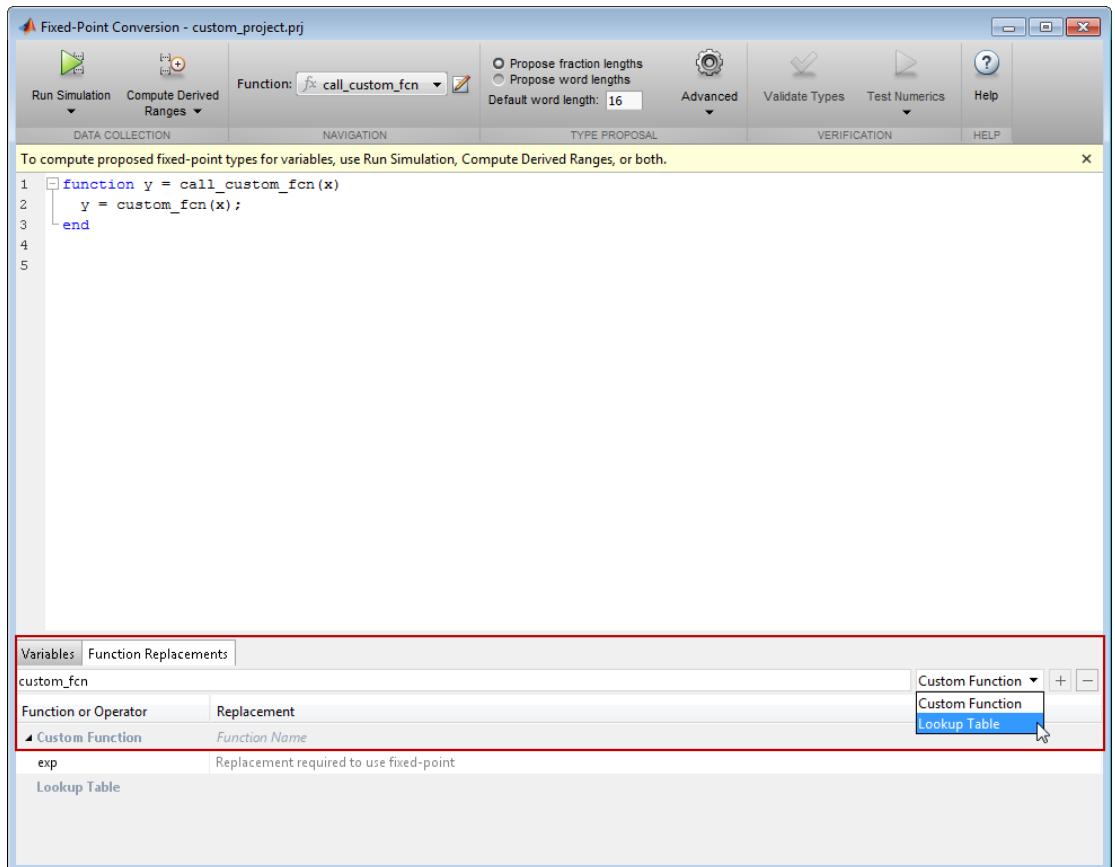
The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code.

- 3 Select the **Function Replacements** tab.

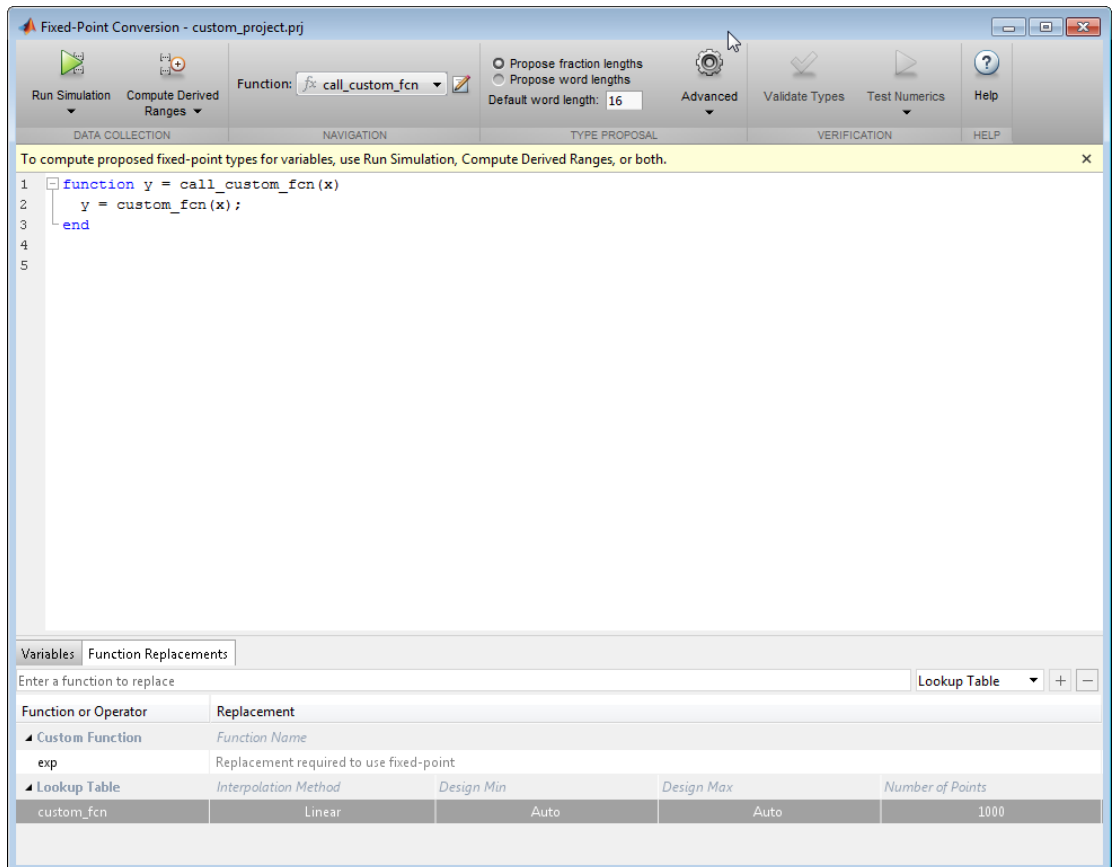


- 4 Enter the name of the function to replace, `custom_fcn`, select `Lookup Table`, and then click `+`.

3 Fixed-Point Conversion



The tool adds `custom_fcn` to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation, 1000 points, and the design minimum and maximum values that the app detects by either running a simulation or computing derived ranges.

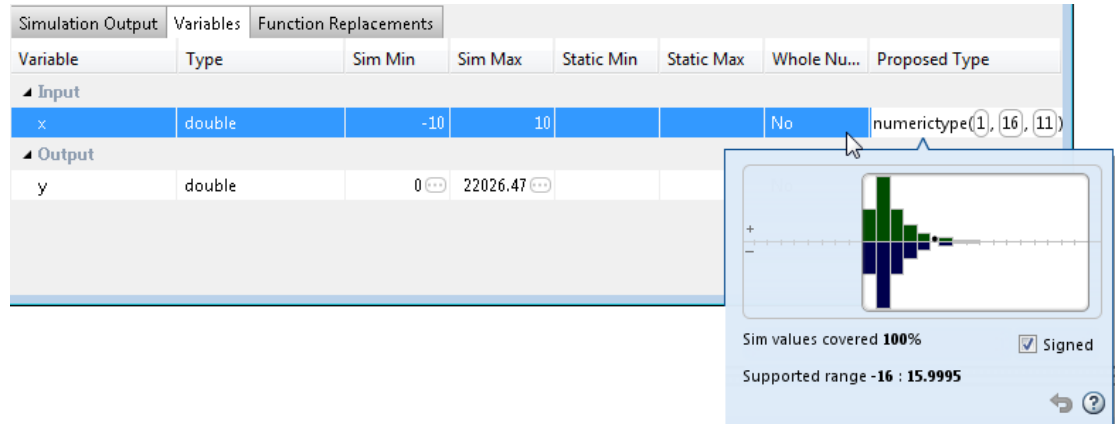


- 5 Click **Run Simulation**, select **Log data for histogram** and verify that the `custom_test` file is selected as a test file to run.
- 6 Click the **Run Simulation** button.

The simulation runs and the tool displays simulation minimum and maximum ranges 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. The **Validate Types** option is now enabled.

- 7 Examine the proposed types and verify that they 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 covered by the proposed data type.



- 8 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types and generates a fixed-point code, `call_custom_fcn_fixpt`.

- 9 On the Type Validation Output tab, click the `call_custom_fcn_fixpt` link to view the generated fixed-point code.

The conversion process generates a lookup table approximation, `custom_fcn1`, for the `custom_fcn` function.

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 = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
               'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...
               'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    y = fi(custom_fcn1(x), 0, 16, 16, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function.

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 (for most platforms, a default C compiler is supplied with MATLAB). For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

Create Algorithm and Test Files

- 1 Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

```
function y = my_fcn(x)
    y = exp(x);
end
```

- 2 Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = my_fcn( x(itr) );
end
plot( x, y );
```

Configure Approximation

Create a function replacement configuration object to approximate the `exp` function, using the default settings of linear interpolation and 1000 points in the lookup table.

```
q = coder.approximation('exp');
```

Set Up Configuration Object

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'my_fcn_test';  
fixptcfg.TestNumerics = true;  
fixptcfg.DefaultWordLength = 16;  
fixptcfg.addApproximation(q);
```

Convert to Fixed Point

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg my_fcn
```

View Generated Fixed-Point Code

To view the generated fixed-point code, click the link to `my_fcn_fixpt`.

The generated code contains a lookup table approximation, `exp1`, 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 = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...  
        'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, ...  
        'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);  
  
    y = fi(exp1(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”

More About

- “Replacing Functions Using Lookup Table Approximations”

Data Type Issues in Generated Code

Within the fixed-point conversion HTML 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 by highlighting expressions in the MATLAB code which result in cumbersome

multiplication or division, or expensive rounding in generated code. For more information on optimizing generated fixed-point code, see “Tips for Making Generated Code More Efficient”.

Cumbersome Operations

Cumbersome operations most often occur due to insufficient range of output. Avoid inputs to a multiply or divide operation that has word lengths larger than the base integer type of your processor. Operations with larger word lengths can be handled in software, but this approach requires much more code and is much slower.

Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method. This check identifies expensive rounding operations in multiplication and division.

Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, when comparing an unsigned integer to a signed integer, one of the inputs must first be cast to the signedness of the other before the comparison operation can be performed. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

Code Generation

- “Create and Set Up Your Project” on page 4-2
- “Primary Function Input Specification” on page 4-6
- “Basic HDL Code Generation with the Workflow Advisor” on page 4-10
- “HDL Code Generation from System Objects” on page 4-14
- “Generate Instantiable Code for Functions” on page 4-19
- “Integrate Custom HDL Code Into MATLAB Design” on page 4-21
- “Enable MATLAB Function Block Generation” on page 4-27
- “System Design with HDL Code Generation from MATLAB and Simulink” on page 4-28
- “Generate Xilinx System Generator Black Box Block” on page 4-32
- “Generate Xilinx System Generator for DSP Black Box from MATLAB HDL Design” on page 4-34
- “Generate HDL Code from MATLAB Code Using the Command Line Interface” on page 4-40
- “Specify the Clock Enable Rate” on page 4-45
- “Specify Test Bench Clock Enable Toggle Rate” on page 4-47
- “Generate an HDL Coding Standard Report from MATLAB” on page 4-49
- “Generate an HDL Lint Tool Script” on page 4-53
- “Generate a Board-Independent IP Core from MATLAB” on page 4-55
- “Minimize Clock Enables” on page 4-58

Create and Set Up Your Project

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

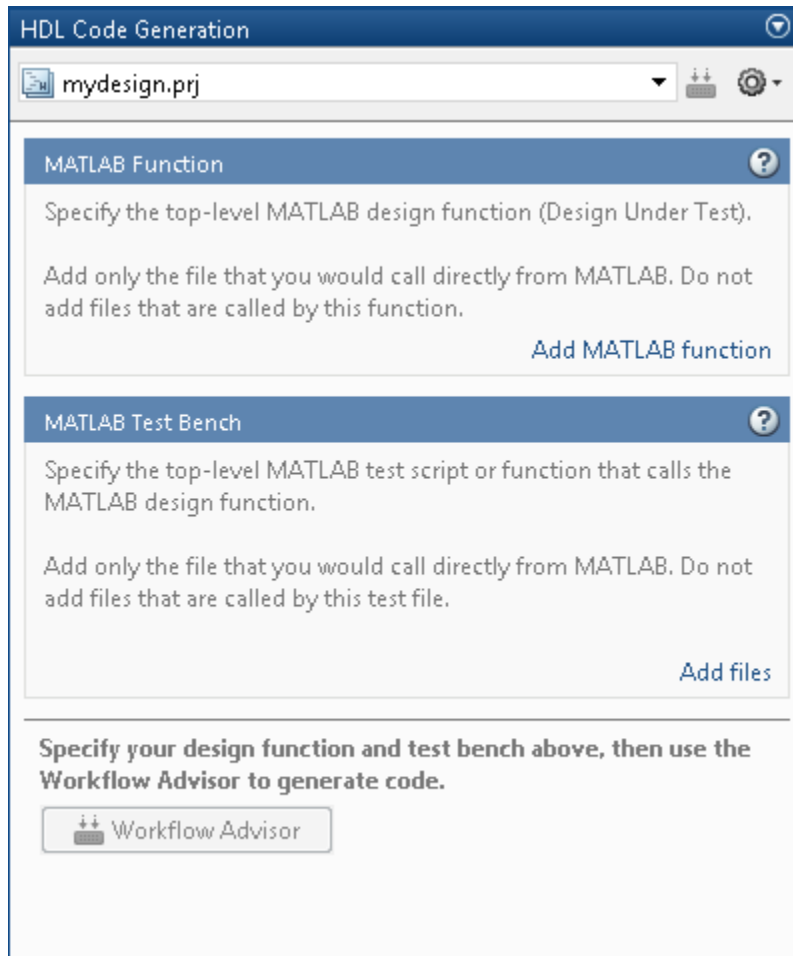
Create a New Project

- 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 “Primary Function Input Specification”.

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 “Primary Function Input Specification”.

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**.

Primary Function Input Specification

In this section...

“When to Specify Input Properties” on page 4-6

“Why You Must Specify Input Properties” on page 4-6

“Properties to Specify” on page 4-6

“Rules for Specifying Properties of Primary Inputs” on page 4-8

“Methods for Defining Properties of Primary Inputs” on page 4-8

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
numericity	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

Class Name	Description
<code>uint16</code>	16-bit unsigned integer array
<code>int32</code>	32-bit signed integer array
<code>uint32</code>	32-bit unsigned integer array
<code>single</code>	Single-precision floating-point or fixed-point number array
<code>double</code>	Double-precision floating-point or fixed-point number array
<code>embedded.fi</code>	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules.

- You must specify the class of all primary inputs. If you do not specify the size or complexity of primary inputs, they default to real scalars.
- For each primary function input whose class is fixed point (`fi`), you must specify the input `numericType` and `fiMath` properties.

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
<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
<p>“Define Input Properties Programmatically in the MATLAB File”</p>	<ul style="list-style-type: none"> • Integrated with MATLAB code; no need to redefine properties each time you invoke HDL Coder 	<ul style="list-style-type: none"> • Uses complex syntax • HDL Coder project files do not currently recognize properties defined programmatically. If you

Method	Advantages	Disadvantages
	<ul style="list-style-type: none">• Provides documentation of property specifications in the MATLAB code• Efficient for specifying memory-intensive inputs such as large structures	are using a project, you must reenter the input types in the project.

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 Code Generation tool interface. The left sidebar contains the HDL Workflow Advisor with the following steps:

- Define Input Types
- Fixed-Point Conversion
- HDL Code Generation
- HDL Verification
 - Verify with HDL Test Bench
 - Verify with Cosimulation
 - Verify with FPGA-in-the-Loop
- Synthesis and Analysis
 - Create Project
 - Run Logic Synthesis
 - Run Place and Route

The main area displays the following Verilog code:

```

21 %#codegen
22 function [y_out, delayed_xout] = mlhdlc_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;
37 m3 = h_in3 * a3; m4 = h_in4 * a4;
38
39 % adder chain
40 a5 = m1 + m2; a6 = m3 + m4;
41

```

The bottom panel shows a table of variables and their types:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
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)
x_in	double	-1	1			No	numerictype(1, 14, 12)
Output							

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');
```

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
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%#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
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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.
```

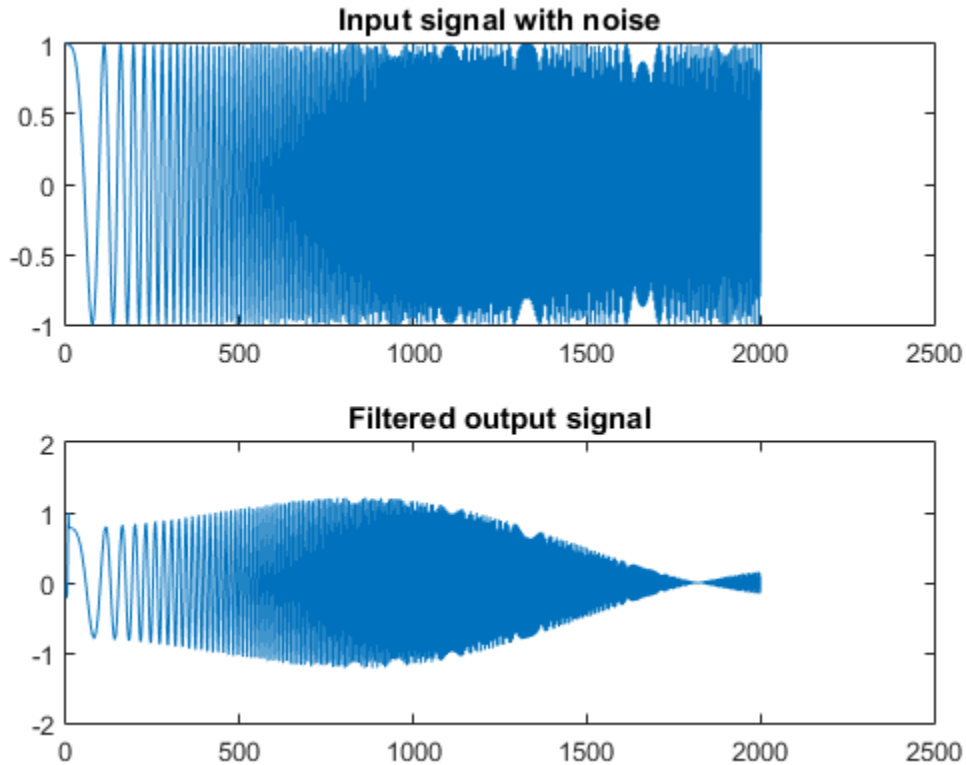
```
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);
```

```
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');
```

Generate Instantiable Code for Functions

In this section...

“How to Generate Instantiable Code for Functions” on page 4-19

“Generate Code Inline for Specific Functions” on page 4-19

“Limitations for Instantiable Code Generation for Functions” on page 4-19

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 `PartitionFunctions` property to `true`. For example, to create a `coder.HdlConfig` object and enable instantiable code generation for functions:

```
hdlcfg = coder.config('hdl');  
hdlcfg.PartitionFunctions = 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 `PartitionFunctions` , `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 4-21

“Use System object In MATLAB Design Function” on page 4-23

“Generate HDL Code” on page 4-23

“Limitations for `hdl.BlackBox`” on page 4-26

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
```

```
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 = hd1.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 a Board-Independent IP Core from MATLAB”

Enable MATLAB Function Block Generation

In this section...

“Requirements for MATLAB Function Block Generation” on page 4-27

“Enable MATLAB Function Block Generation” on page 4-27

“Results of MATLAB Function Block Generation” on page 4-27

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;
```

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”.

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.

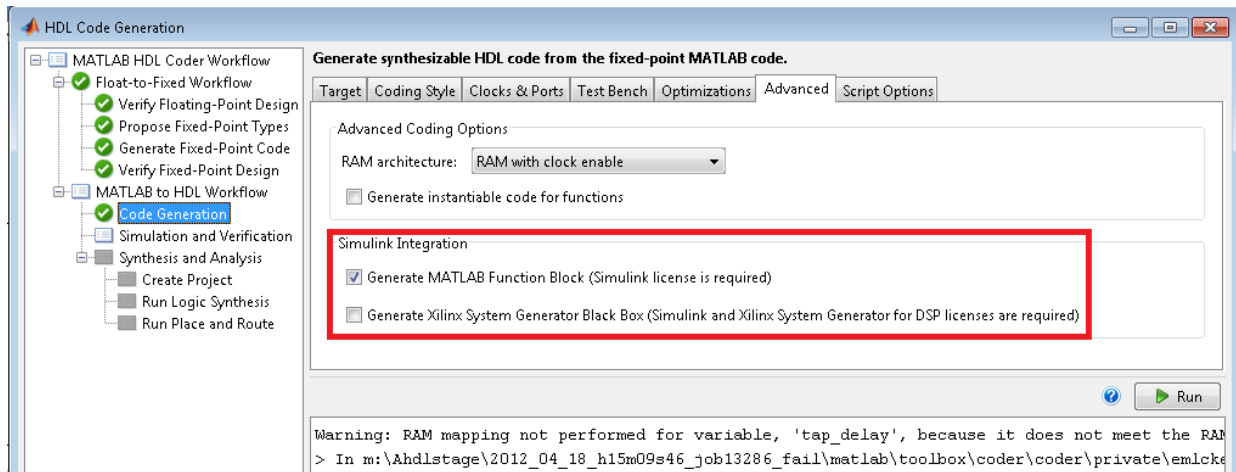
Click the Workflow Advisor button to launch the HDL Workflow Advisor.

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.

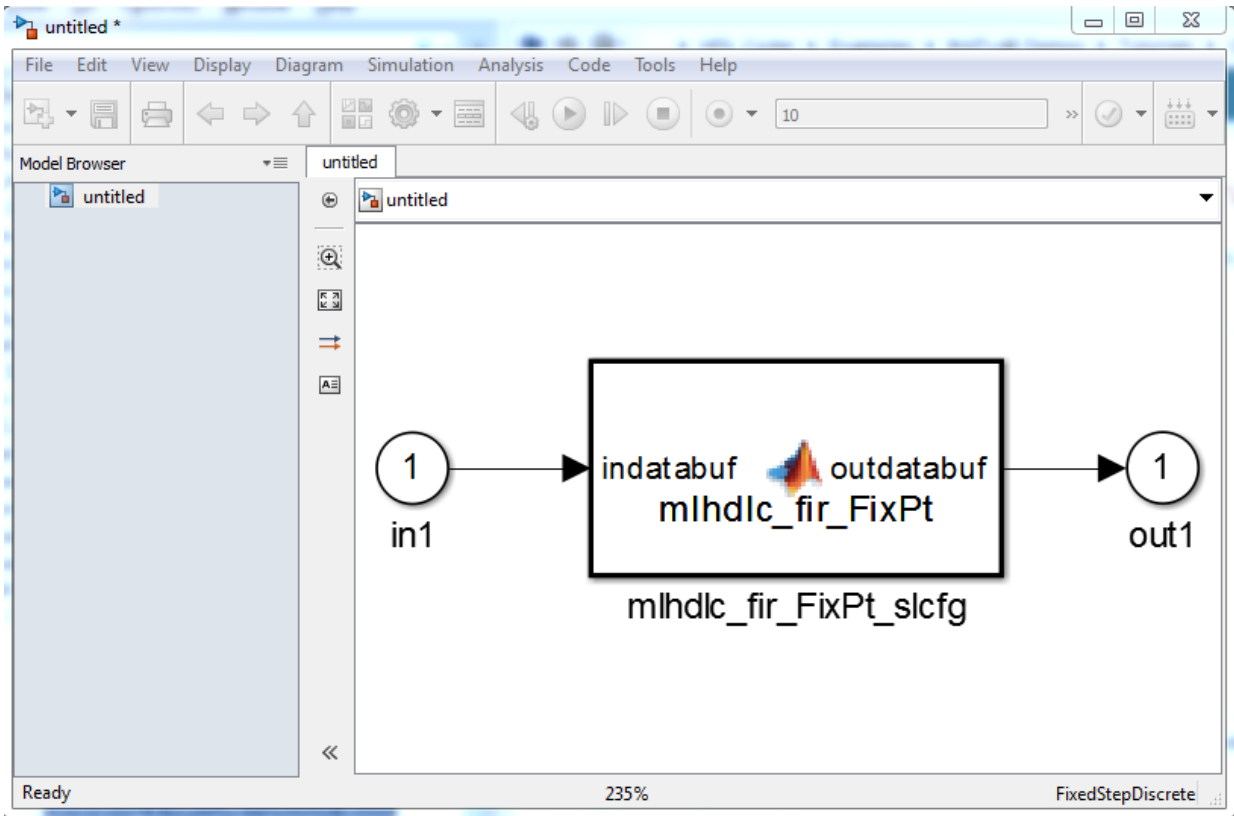
Right-click the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

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 Xilinx System Generator Black Box Block

In this section...

“Requirements for System Generator Black Box Block Generation” on page 4-32

“Enable System Generator Black Box Block Generation” on page 4-32

“Results of System Generator Black Box Block Generation” on page 4-33

Requirements for System Generator Black Box Block Generation

You must have Xilinx® ISE Design Suite 13.4 or later to generate a System Generator Black Box block.

To verify your System Generator setup, at the command line, enter:

```
xlVersion
```

Enable System Generator Black Box Block Generation

Using the GUI

To enable System Generator Black Box 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 Xilinx System Generator Black Box** option.
- 3 In the **Clocks & Ports** tab, set the following fields:
 - For **Clock input port**, enter **clk**.
 - For **Clock enable input port**, enter **ce**.
 - For **Drive clock enable at**, select **DUT base rate**.

Using the Command Line

To enable System Generator Black Box block generation, at the command line, enter:

```
hdlcfg = coder.config('hdl');  
hdlcfg.GenerateXSGBlock = true;
```

```
hdlcfg.ClockInputPort = 'clk';  
hdlcfg.ClockEnableInputPort = 'ce';  
hdlcfg.EnableRate = 'DutBaseRate';
```

Results of System Generator Black Box Block Generation

After you generate HDL code, you have:

- An XSG subsystem.
- A System Generator Black Box block within the XSG subsystem.
- A System Generator Black Box configuration M-function.

You can use the XSG subsystem in a Simulink model, or use the Black Box block and Black Box configuration M-function in a Xilinx System Generator design.

To learn more about generating a System Generator Black Box block, see “Using Xilinx System Generator for DSP with HDL Coder”.

Generate Xilinx System Generator for DSP Black Box from MATLAB HDL Design

This example shows how to generate a Xilinx® System Generator for DSP Black Box block from a MATLAB® HDL design.

Introduction

HDL Coder can generate a System Generator Black Box block and configuration file from your MATLAB HDL design. After designing an algorithm in MATLAB for HDL code generation, you can then integrate it into a larger system as a Xilinx System Generator Black Box block.

HDL Coder places the generated Black Box block in a Xilinx System Generator (XSG) subsystem. XSG subsystems work with blocks from both Simulink® and Xilinx System Generator, so you can use the generated black box block to build a larger system for simulation and code generation.

MATLAB Design

The MATLAB code in the example implements 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 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

To simulate the design with the test bench to make sure there are no runtime errors before code generation, enter the following command:

```
mlhdlc_fir_tb
```

Create 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.

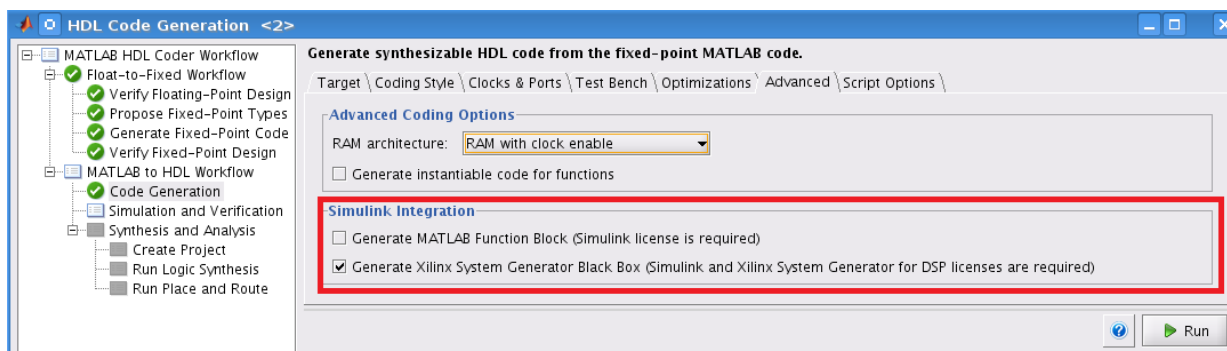
Click the Workflow Advisor button to launch the HDL Workflow Advisor.

Generate a Xilinx System Generator for DSP Black Box

To generate a Xilinx System Generator Black Box from a MATLAB HDL design, you must have Xilinx System Generator configured. Enter the following command to check System Generator availability:

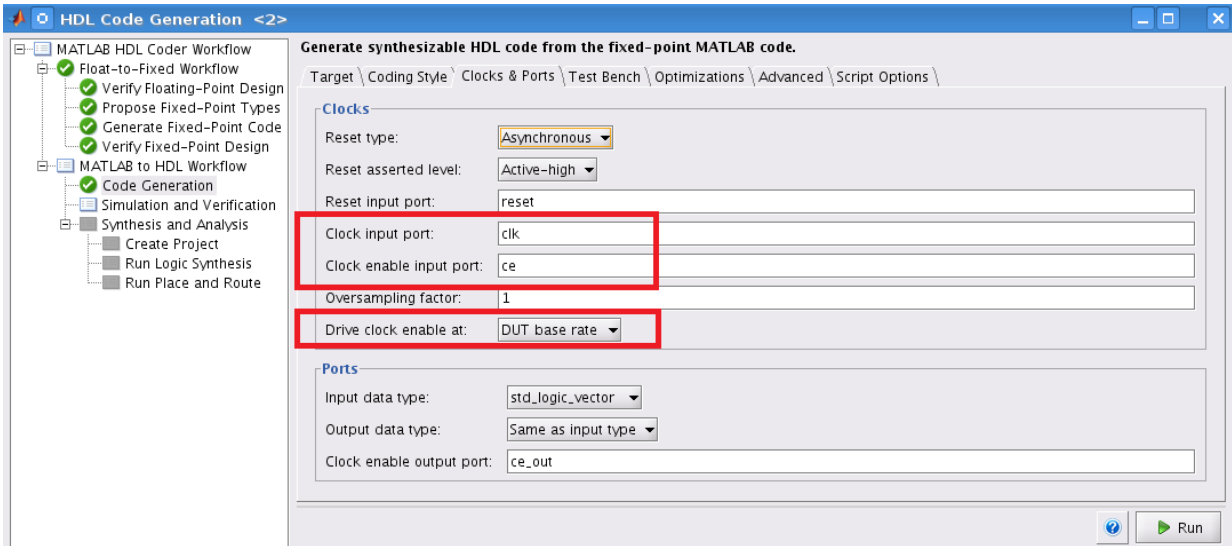
```
xlVersion
```

In the Advanced tab of the Workflow Advisor, enable the Generate Xilinx System Generator Black Box option:



To generate code compatible with a Xilinx System Generator Black Box, set:

- 'Clock input port' to 'clk'
- 'Clock enable input port' to 'ce'
- 'Drive clock enable at' to 'DUT base rate'



Run Fixed-Point Conversion and Generate Code

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

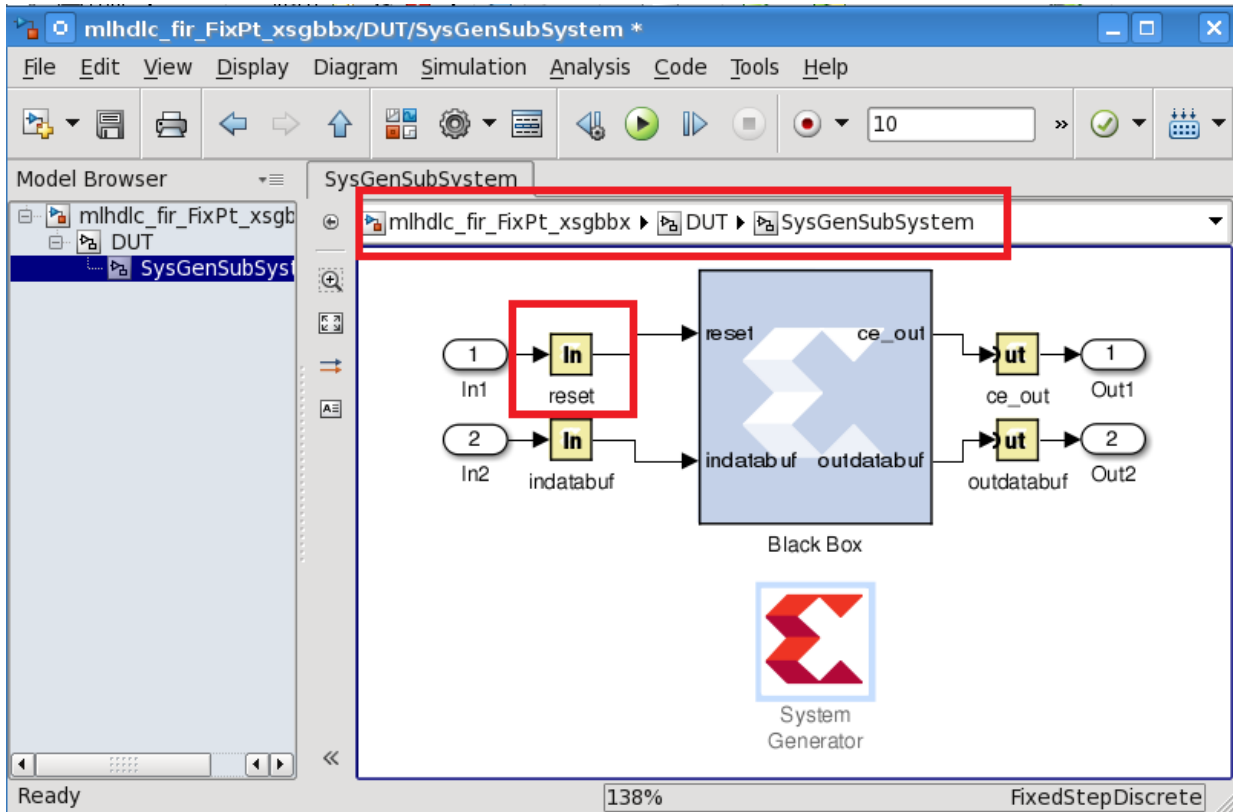
Examine the Generated Model and Config File

A new model opens after HDL code generation. It contains a subsystem called DUT at the top level.

The DUT subsystem has an XSG subsystem called SysGenSubSystem, which contains:

- A Xilinx System Generator Black Box block
- A System Generator block
- Gateway-in blocks

- Gateway-out blocks



Notice that in addition to the data ports, there is a reset port on the black box interface, while 'clk' and 'ce' are registered to System Generator by the Black Box configuration file.

The configuration file and your new model are saved in the same directory with generated HDL code. You can open the configuration file by entering the following command:

```
edit('codegen/mlhdlc_fir/hdlsrc/mlhdlc_fir_FixPt_xsgbbcfg.m');
```

The screenshot shows a MATLAB Editor window titled "Editor - /tmp/mlhdlc_fir/codegen/mlhdlc_fir/hdlsrc/mlhdlc_fir_FixPt_xsgbbxcfg.m". The window has three tabs: EDITOR, PUBLISH, and VIEW. The toolbar includes icons for New, Open, Save, Print, Compare, Source Control, Insert, Comment, Indent, Go To, Find, Breakpoints, Run, and Run and Advance. The code in the editor is as follows:

```

1
2  function mlhdlc_fir_FixPt_xsgbbxcfg(this_block)
3      % Set target language
4      this_block.setTopLevelLanguage('VHDL');
5      % Set top entity name
6      this_block.setEntityName('mlhdlc_fir_FixPt');
7      % Set the combinational flag
8      this_block.tagAsCombinational;
9      % Set inport names
10     this_block.addSimulinkInport('reset');
11     this_block.addSimulinkInport('indatabuf');
12     % Set outport names and types
13     this_block.addSimulinkOutport('ce_out');
14     ce_out_obj = this_block.port('ce_out');
15     ce_out_obj.setType('UFix_1_0');
16     this_block.port('ce_out').useHDLVector(false);
17     this_block.addSimulinkOutport('outdatabuf');
18     outdatabuf_obj = this_block.port('outdatabuf');
19     outdatabuf_obj.setType('Fix_14_10');
20     % Set inport types and types are known
21     if (this_block.inputTypesKnown)
22         if(this_block.port('reset').width ~= 1)
23             this_block.setError('Input data type for port "reset" must have wid
24         end
25         this_block.port('reset').useHDLVector(false);
26         if(this_block.port('indatabuf').width ~= 14)
27             this_block.setError('Input data type for port "indatabuf" must have
28         end
29     end
30

```

You can now use the generated Xilinx System Generator Black Box block and configuration file in a larger system design.

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');
```

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 creating 'fixpt' and 'hdl' config objects set up, run the following codegen command to perform floating-point to fixed-point conversion, generate HDL code.

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

Alternatively, 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 coder 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 coder 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 coder 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 data differences introduced by fixed-point conversion.

Enable data logging in the 'fixpt' config object:

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

View the Numeric Type Proposal Report

Configure the coder to launch the type proposal report after the coder has proposed fixed-point types:

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

Specify a Type For a Design Variable

If you want to specify the fixed-point data type for a variable in your design, you can create a type specification, set its fields, and associate it with the variable.

The type specification has the following fields:

- `IsInteger`: Can be true or false

- `ProposedType`: A type string, like 'ufix15' or 'int32'.
- `RoundingMethod`: Can be 'ceil', 'convergent', 'fix', 'floor', 'nearest', or 'round'.
- `OverflowAction`: Can be 'saturate' or 'wrap'.

Create a type specification and associate it with the 'delayed_xout' variable:

Create a type specification object.

```
typeSpec = coder.FixPtTypeSpec;
```

Set fields in the typeSpec object.

```
typeSpec.ProposedType = 'ufix15';  
typeSpec.RoundingMethod = 'nearest';  
typeSpec.OverflowAction = 'saturate';
```

Associate the type specification with the variable, 'yt'.

```
fixptcfg.addTypeSpecification('mlhdlc_dti', 'yt', typeSpec)
```

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. The 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 4-45

“How to Specify the Clock Enable Rate” on page 4-45

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)	Each assertion of the input clock enable produces an output clock enable assertion. You can assert the input clock enable at a maximum rate of once every N clocks.

Drive clock enable at Option	Clock Enable Behavior
	<p>N = the upsampled clock rate / 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>
<p>DUT base rate</p>	<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 = the upsampled clock rate / 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 4-47

“How to Specify Test Bench Clock Enable Toggle Rate” on page 4-47

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 4-45.

How to Specify Test Bench Clock Enable Toggle Rate

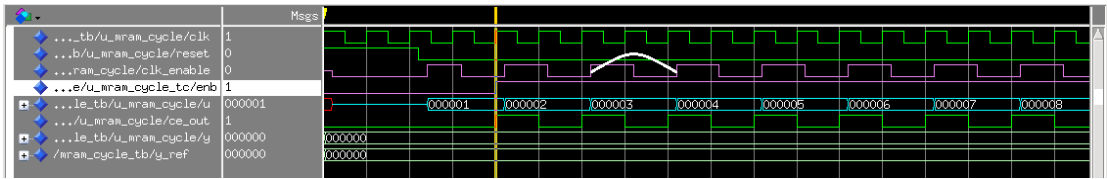
To set your test bench clock enable toggle rate:

- 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.

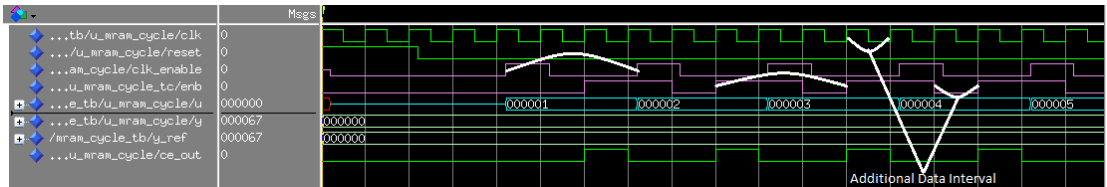
Input data interval, I	Test Bench Clock Enable Behavior
$I = 0$ (default)	Asserts at the maximum input clock enable rate, or once every N cycles. N = the upsampled clock rate / original clock rate.
$I < N$	Not valid; generates an error.

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 4-49
“Using the Command Line” on page 4-51

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 reduced clock enable signals						
<input type="checkbox"/> Check for 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 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.
- 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 Properties

More About

- “HDL Coding Standard Report”
- “HDL Coding Standard Rules”

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 strings. If you want to generate a script for a custom lint tool, you must specify the initialization, command, and termination strings.

HDL Coder writes the initialization, command, and termination strings 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 a Board-Independent IP Core from MATLAB

In this section...

“Generate a Board-Independent IP Core” on page 4-55

“Requirements and Limitations for IP Core Generation” on page 4-57

Generate a Board-Independent IP Core

To generate a board-independent IP core to use in an embedded system integration environment, such as Altera® Qsys, Xilinx EDK, or Xilinx IP Integrator:

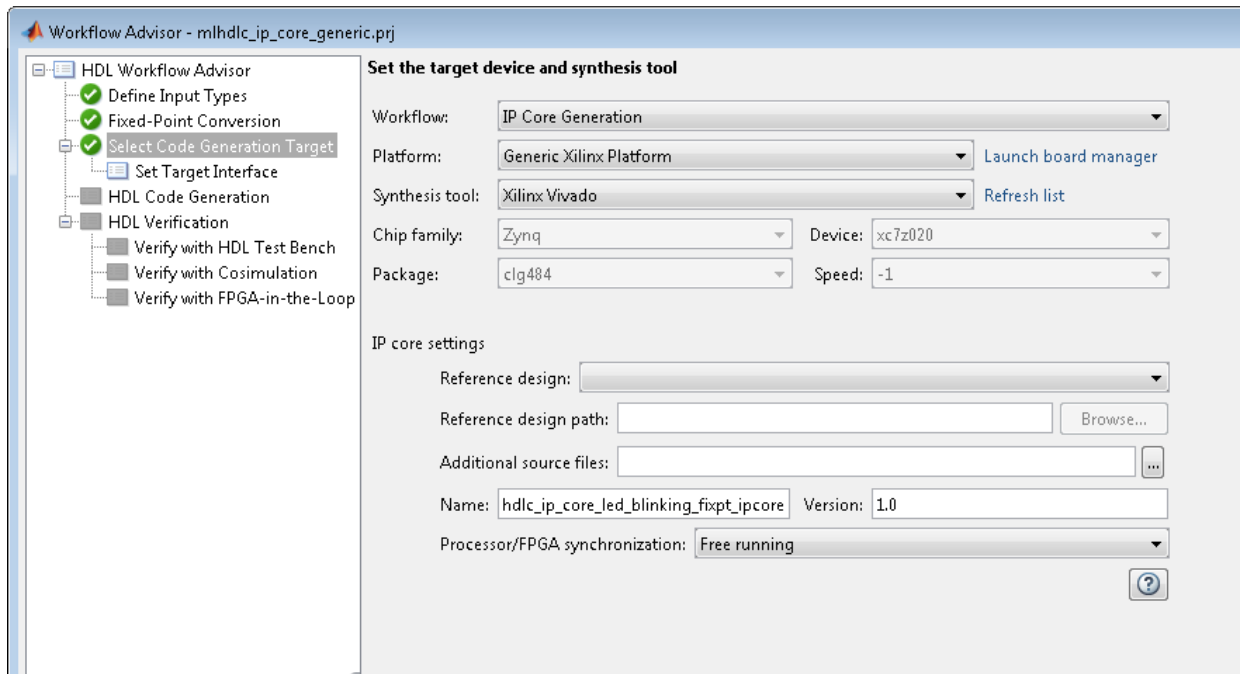
- 1 Create an HDL Coder project containing your MATLAB design and test bench, or open an existing project.
- 2 In the HDL Workflow Advisor, define input types and perform fixed-point conversion.

To learn how to convert your design to fixed-point, see “HDL Code Generation 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 coder automatically sets **Synthesis tool**. For example, if you select **Generic Xilinx Platform**, **Synthesis tool** automatically changes to **Xilinx Vivado**. You can change the **Synthesis tool** to **Xilinx ISE**.

- **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.

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, 4, 0)	External Port	
Read_back	numerictype(0, 4, 0)	AxI4	x"108"

- 5 In the **HDL Code Generation** step, optionally specify code generation options, then click **Run**.
- 6 In the HDL Workflow Advisor message pane, click the IP core report link to view detailed documentation for your generated IP core.

Requirements and Limitations for IP Core Generation

You cannot map to both an AXI4 interface and AXI4-Lite interface in the same IP core.

To map your design function inputs or outputs to an AXI4-Lite interface, the input and outputs must:

- Have a bit width less than or equal to 32 bits.
- Be scalar.

When mapping design function inputs or outputs to an AXI4-Stream Video interface, the following requirements apply:

- 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 mode.

Minimize Clock Enables

In this section...

“Using the GUI” on page 4-59

“Using the Command Line” on page 4-59

“Limitations” on page 4-59

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, in the **HDL Code Generation > Clocks & Ports** tab, select **Minimize clock enables**.

Using the Command Line

To minimize clock enables, in the `coder.HdlConfig` configuration object, set the `MinimizeClockEnables` property to `true`. For example:

```
hdlCfg = coder.config('hdl')
hdlCfg.MinimizeClockEnables = true;
```

Limitations

If you specify area optimizations that the coder implements by increasing the clock rate in certain regions of the design, you cannot minimize clock enables. The following optimizations prevent clock enable minimization:

- Resource sharing
- RAM mapping
- Loop streaming

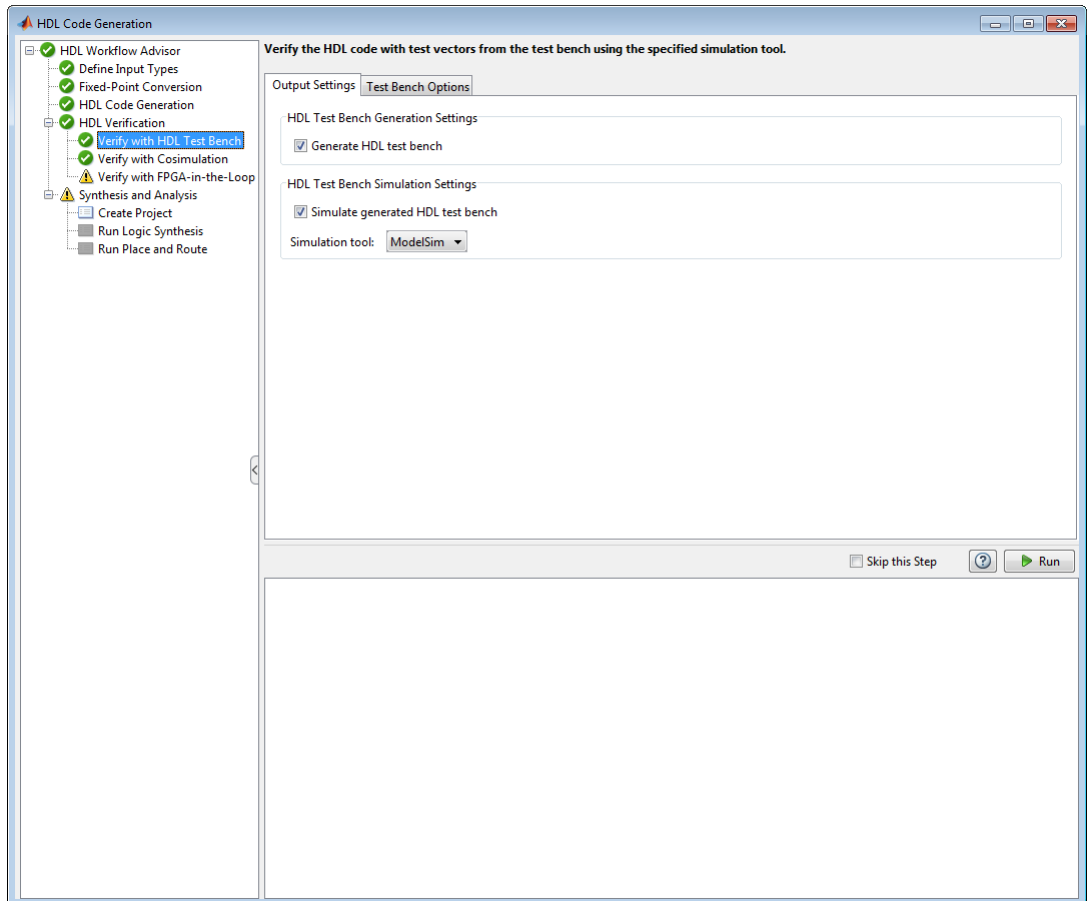
Verification

- “Verify Code with HDL Test Bench” on page 5-2
- “Generate Test Bench With File I/O” on page 5-5

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”.

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 string to append to name of test bench data file when generating multi-file test bench.
Test bench reference postfix	Specify the string 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**.

Generate Test Bench With File I/O

In this section...

“When to Use File I/O In Test Bench” on page 5-5

“How Test Bench Generation with File I/O Works” on page 5-5

“Test Bench Data Files” on page 5-5

“How to Generate Test Bench with File I/O” on page 5-6

“Limitations When Using File I/O In Test Bench” on page 5-6

When to Use File I/O In Test Bench

By default, HDL Coder generates an HDL testbench that contains the simulation data as constants. If you have a long running simulation, the generated HDL test bench contains a large amount of data, and therefore requires more memory to run in an HDL simulator.

Generate your test bench with file I/O when your MATLAB or Simulink simulation is long, or you experience memory constraints while running your HDL simulation.

How Test Bench Generation with File I/O Works

By default, when you generate an HDL test bench, HDL Coder writes the stimulus and reference data from your simulation as constants in the test bench code.

When you enable the **Use file I/O to read/write test bench data** option in the HDL Workflow Advisor and generate a test bench, HDL Coder saves the DUT input and output 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 and compares the actual DUT output with the expected output, which is also saved in .dat files. This saves memory compared to the default option.

Note that reference data is delayed by 1 clock cycle in the waveform viewer compared to default test bench generation. This is due to the delay in reading data from files.

Test Bench Data Files

Stimulus and reference data for each DUT input and output is saved in a separate test bench data file (.dat), with the following exceptions:

- 2 files are generated for the real and imaginary parts of complex data.
- Constant DUT input data is written to the test bench as constants, the same as for the default option.

Vector input or output data is saved as a single file.

How to Generate Test Bench with File I/O

To create and use data files for reading and writing test bench input and output data:

- 1 In the HDL Workflow Advisor, select the **HDL Verification > Verify with HDL Test Bench** task.
- 2 In the **Test bench Options** tab, enable the **Use file I/O for test bench** option.

Limitations When Using File I/O In Test Bench

To use file I/O in your test bench, the following limitations apply:

- Double and single data types at DUT inputs and outputs are not supported.
- If your target language is VHDL, the **Scalarize vector ports** option must be off.

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” on page 7-2
- “Map Persistent Arrays and dsp.Delay to RAM” on page 7-3
- “RAM Mapping Comparison for MATLAB Code” on page 7-8
- “Pipelining MATLAB Code” on page 7-9
- “Register Inputs and Outputs” on page 7-10
- “Insert Input and Output Pipeline Registers” on page 7-11
- “Pipeline MATLAB Expressions” on page 7-12
- “Distributed Pipelining” on page 7-14
- “Optimize MATLAB Loops” on page 7-15
- “Constant Multiplier Optimization” on page 7-18
- “Specify Constant Multiplier Optimization” on page 7-20
- “Distributed Pipelining for Clock Speed Optimization” on page 7-21
- “Map Matrices to Block RAMs to Reduce Area” on page 7-28
- “Resource Sharing of Multipliers to Reduce Area” on page 7-33
- “Loop Streaming to Reduce Area” on page 7-42
- “Constant Multiplier Optimization to Reduce Area” on page 7-48

RAM Mapping

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...

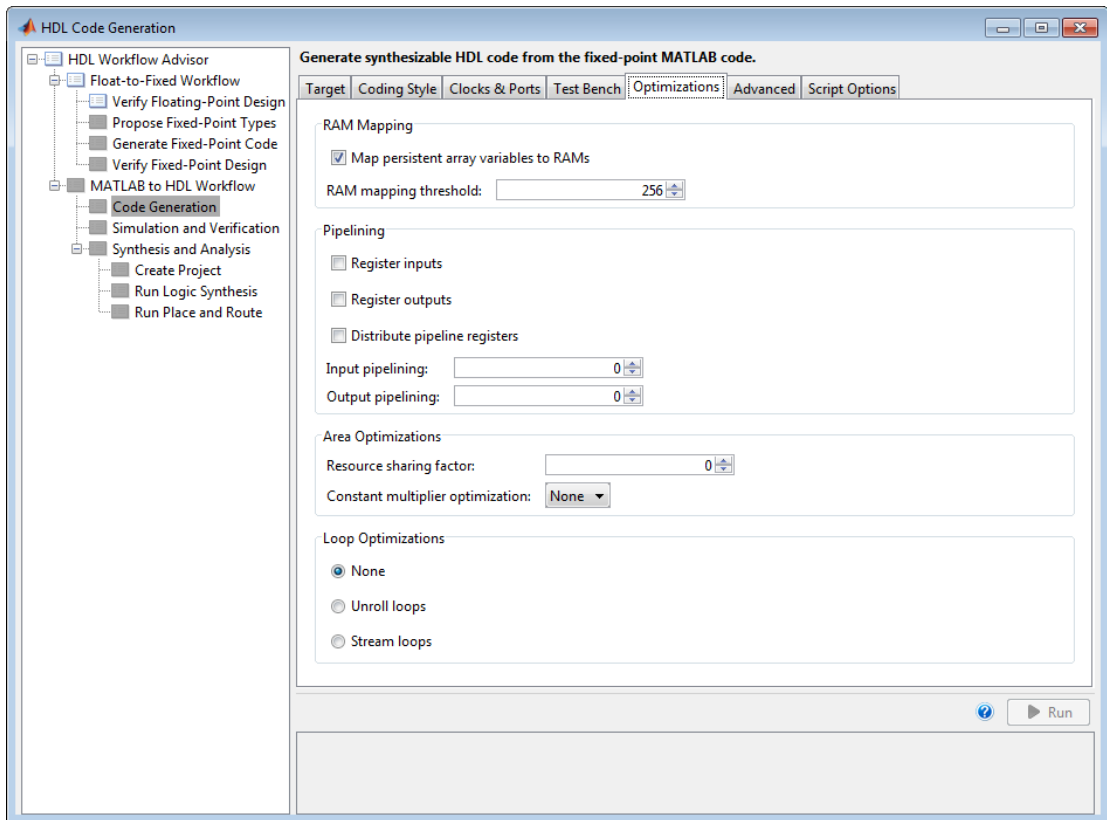
“How To Enable RAM Mapping” on page 7-3

“RAM Mapping Requirements for Persistent Arrays and System object Properties” on page 7-4

“RAM Mapping Requirements for dsp.Delay System Objects” on page 7-6

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 7-5.
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</code> System 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.

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

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”.	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for Persistent Arrays and System object Properties”.
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 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 considered good design practice for synchronous interfaces.

Port registers are not affected by distributed pipelining.

To learn how to insert port registers, see “Register Inputs and Outputs”.

Input and Output Pipeline Registers

You can insert multiple input and output pipeline stages. These input and output pipeline registers can move during distributed pipelining 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 learn how to insert input and output pipeline registers, see “Insert Input and Output Pipeline Registers”.

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”.

Register Inputs and Outputs

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.

To learn more about input and output port registers, see “Port Registers”.

Insert Input and Output Pipeline Registers

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.

To learn more about input and output pipeline registers, see “Input and Output Pipeline Registers”.

Pipeline MATLAB Expressions

In this section...

“How To Pipeline a MATLAB Expression” on page 7-12

“Limitations of Pipelining for MATLAB Expressions” on page 7-13

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, and specify the number of registers.

You can insert pipeline registers in the generated HDL code:

- At the output of the entire right-hand 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

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, an expression containing the `t` or `pvar` variables cannot be pipelined:

```
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 “Register Inputs and Outputs”.

See Also

`coder.hdl.pipeline`

More About

- “Pipelining MATLAB Code”

Distributed Pipelining

In this section...
“What is Distributed Pipelining?” on page 7-14
“Benefits and Costs of Distributed Pipelining” on page 7-14
“Selected Bibliography” on page 7-14

What is Distributed Pipelining?

Distributed pipelining, or register retiming, is a speed optimization that moves existing delays within 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 7-15

“Loop Unrolling” on page 7-15

“How to Optimize MATLAB Loops” on page 7-15

“Limitations for MATLAB Loop Optimization” on page 7-16

With loop optimization you can stream or unroll loops in generated code. Loop streaming optimizes for area, and loop unrolling optimizes for speed.

Loop Streaming

HDL Coder streams a loop by instantiating the loop body once and using that instance for each loop iteration. The coder oversamples the loop body instance to keep the generated loop functionally equivalent to the original loop. You can partially stream a loop.

The advantage of loop streaming is decreased area because the loop body is instantiated fewer times. The disadvantage of loop streaming is lower speed.

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, so that the generated code uses a loop statement that contains multiple instances of the original loop body and fewer iterations than the original loop.

The unrolled code can participate in distributed pipelining and resource sharing optimizations. Distributed pipelining can increase speed; resource sharing can decrease area.

Overall, however, the multiple instances created by loop unrolling are likely to increase area. Loop unrolling also makes the code less readable.

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 Open the HDL Workflow Advisor.
- 2 In the left pane, select **HDL Workflow Advisor > HDL Code Generation**.
- 3 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, it takes precedence over `coder.hdl.loopspec`, and `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 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.
- 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. It must be specified 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

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>
Auto	<p>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.</p> <p>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).</p>

To learn how to specify constant multiplier optimization, see “Specify Constant Multiplier Optimization”.

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**.

To learn more about the constant multiplier optimization options, see “Constant Multiplier Optimization”.

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.

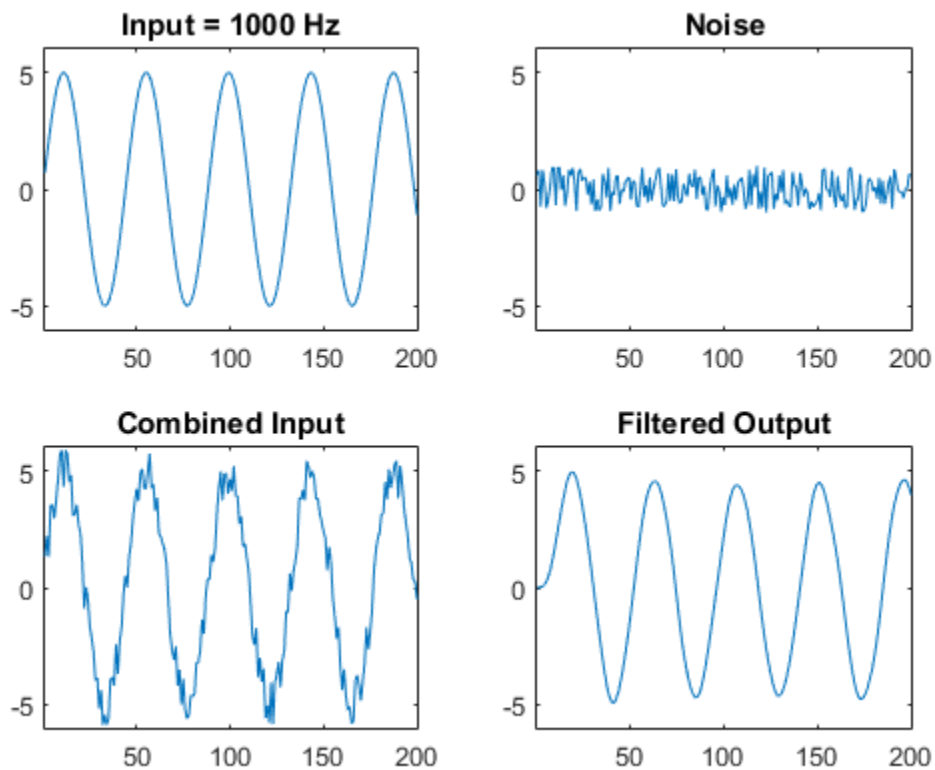
```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlab');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];  
  
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
```

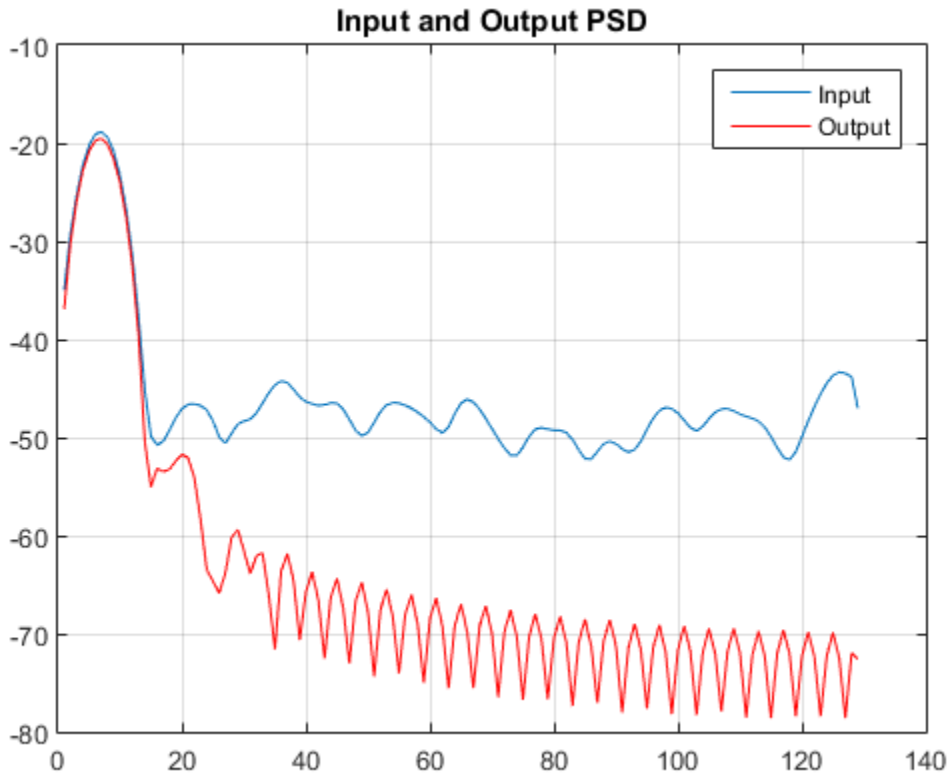
```
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);  
  
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

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 New Project From the Command Line

```
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.

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.

Distributed Pipelining

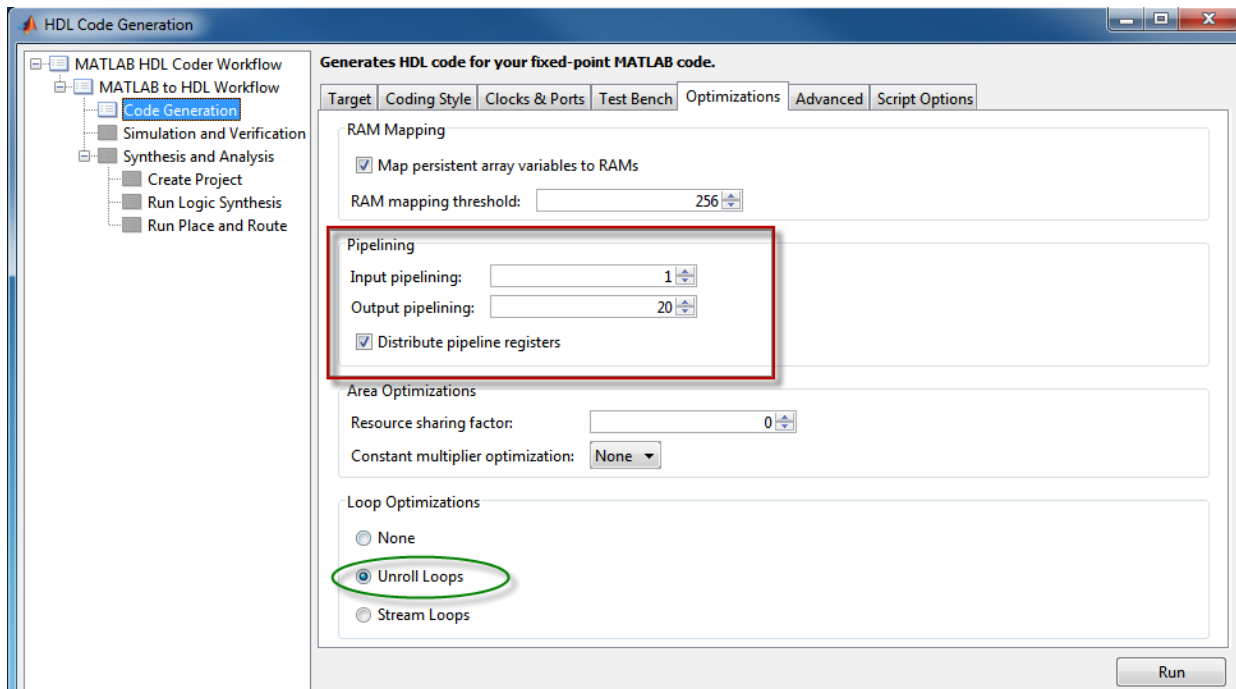
To increase the clock speed, the user can set a number of input and output pipeline stages for any design. In this particular example Input pipelining option is set to '1' and

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.



Run Fixed-Point Conversion and HDL Code Generation

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

Examine the Synthesis Results

Run the logic synthesis step with the following default options if you have ISE installed on your machine.

Create synthesis project for supported synthesis tool.

Synthesis Tool Selection

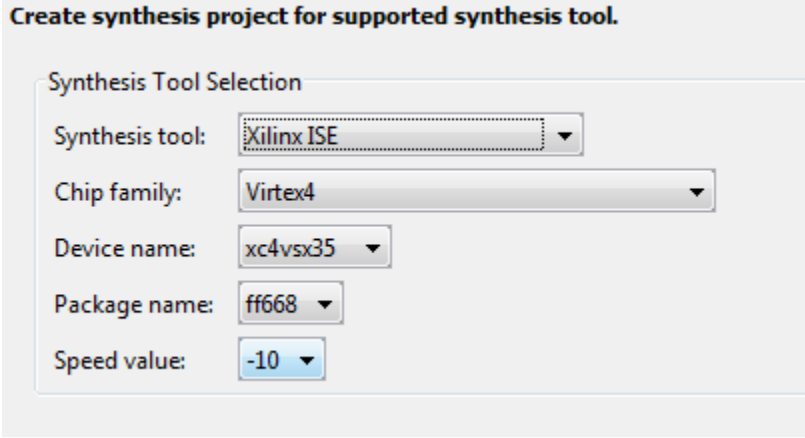
Synthesis tool: Xilinx ISE

Chip family: Virtex4

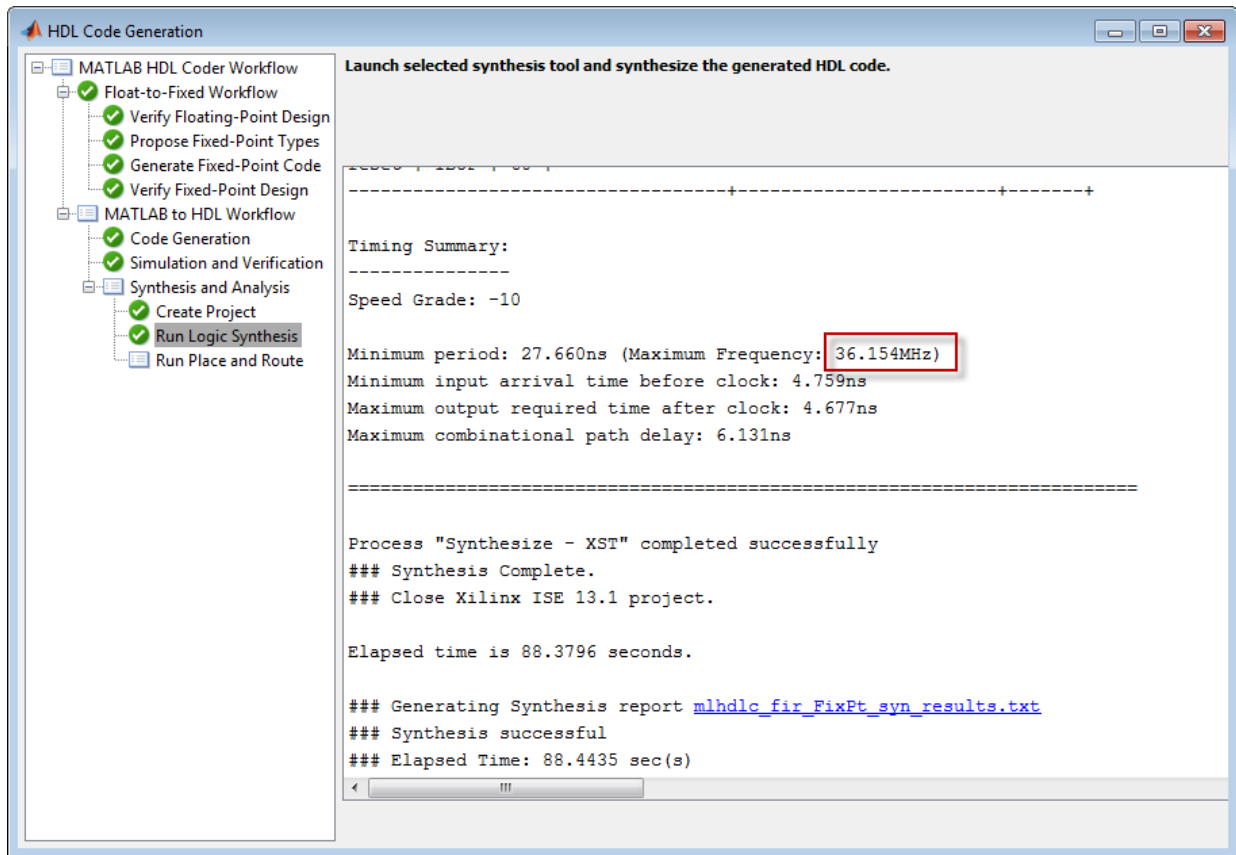
Device name: xc4vsx35

Package name: ff668

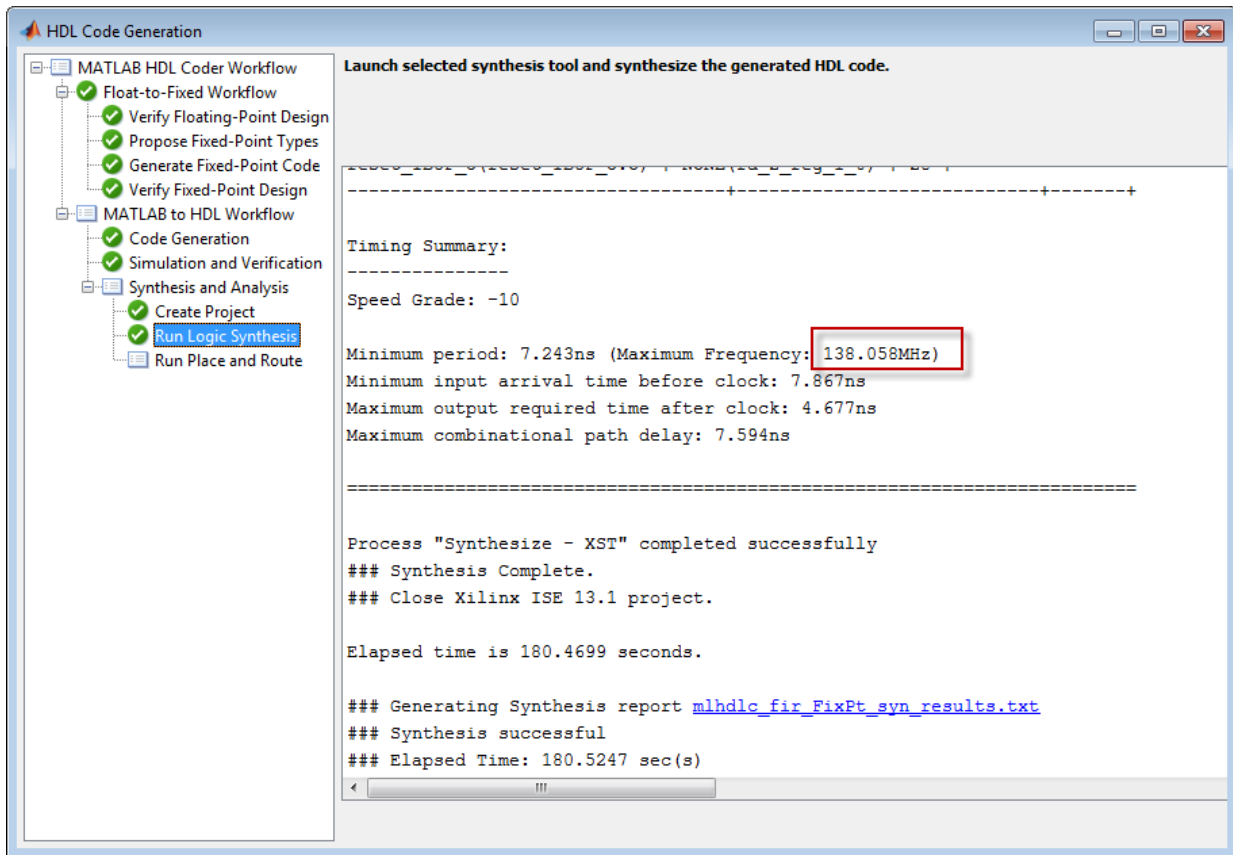
Speed value: -10



In the synthesis report, note the clock frequency reported by the synthesis tool without any optimization options enabled.



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);

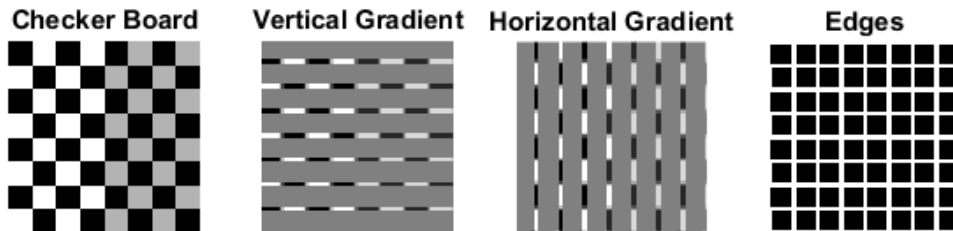
% copy the design files to the temporary directory
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);

```

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
```

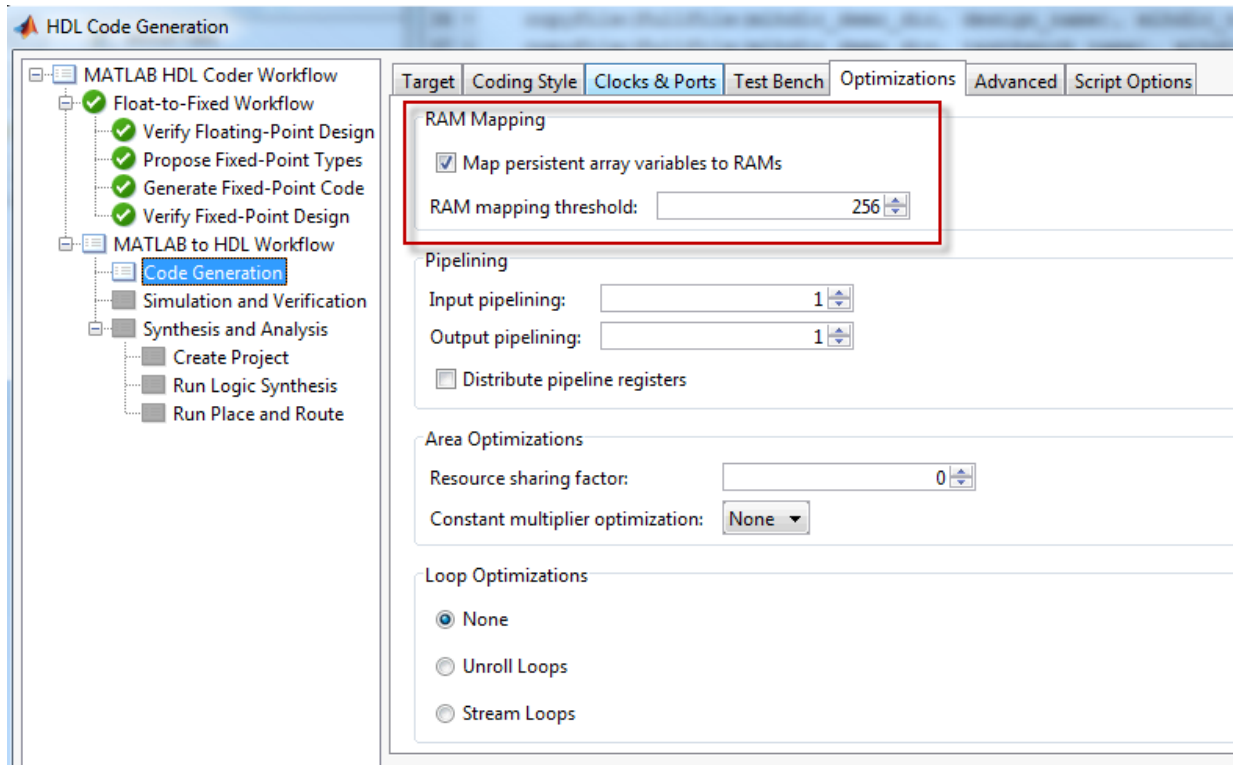
Next, add the file 'mlhdlc_sobel.m' to the project as the MATLAB function, and 'mlhdlc_sobel_tb.m' as the MATLAB test bench.

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.

```

Run

### Begin VHDL Code Generation
### Working on mlhdlc_sobel_FixPt/u_d_ram/DualPortRAM_128x9b as DualPortRAM\_128x9b.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram as u\_d\_ram.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram/DualPortRAM_128x9b as DualPortRAM\_128x9b\_block.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram as u\_d\_ram\_block.vhd
### Working on mlhdlc_sobel_FixPt as mlhdlc\_sobel\_FixPt.vhd
### Generating package file mlhdlc\_sobel\_FixPt\_pkg.vhd
### The DUT requires an initial pipeline setup latency. Each output port experiences these
additional delays
### Output port 0: 4 cycles
### Output port 1: 4 cycles
### Generating Resource Utilization Report resource\_report.html
### Elapsed Time: 33.1382 sec(s)

Help

```

A warning message appears for each persistent matrix variable not mapped to RAM.

Examine the Resource Report

Take a look at the generated resource report, which shows the number of RAMs inferred, by following the 'Resource Utilization report...' link in the generated code window.

Multipliers	0
Adders/Subtractors	19
Registers	29
RAMs	2
Multiplexers	5

Additional Notes on RAM Mapping

- Persistent matrix variable accesses must be in unconditional regions, i.e., outside any if-else, switch case, or for-loop code.
- MATLAB functions can have any number of RAM matrices.
- All matrix variables in MATLAB that are declared persistent and meet the threshold criteria get mapped to RAMs.
- A warning is shown when a persistent matrix does not get mapped to RAM.
- Read-dependent write data cycles are not allowed: you cannot compute the write data as a function of the data read from the matrix.
- Persistent matrices cannot be copied as a whole or accessed as a sub matrix: matrix access (read/write) is allowed only on single elements of the matrix.
- Mapping persistent matrices with non-zero initial values to RAMs is not supported.

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 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 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.

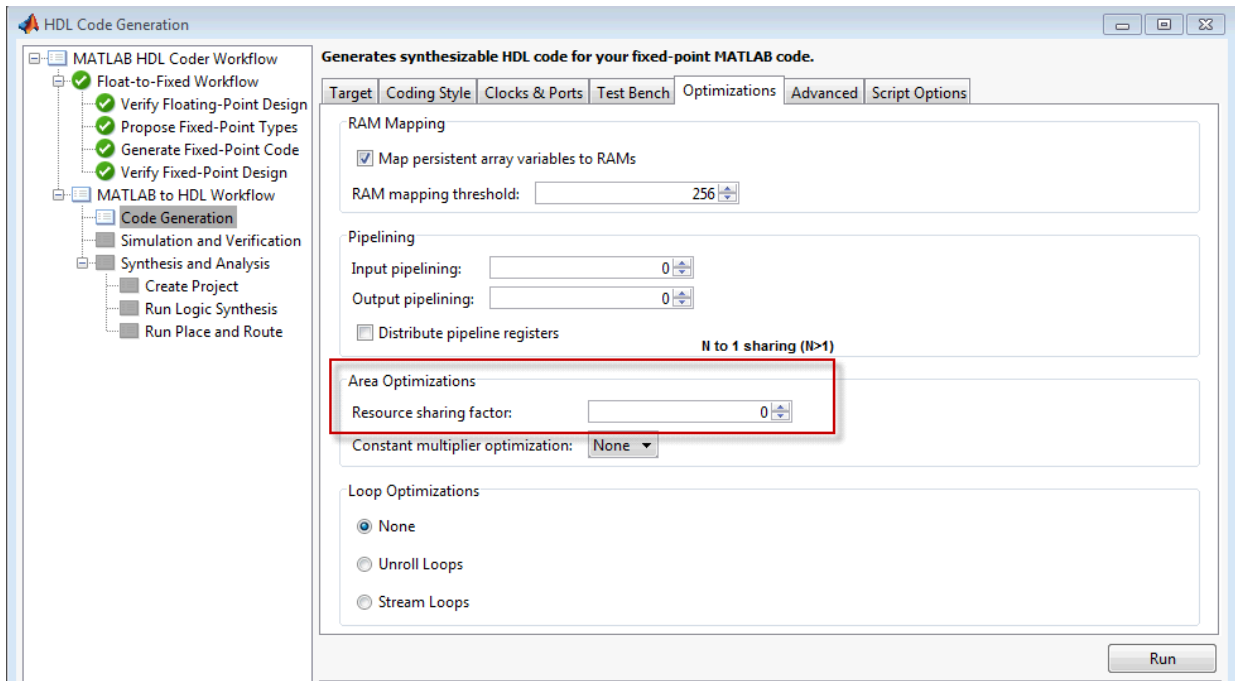
You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

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 $N > 1$.

Note that you can only share those multipliers with same type of outputs and operands. See the following section on Fixed-point conversion below for details.



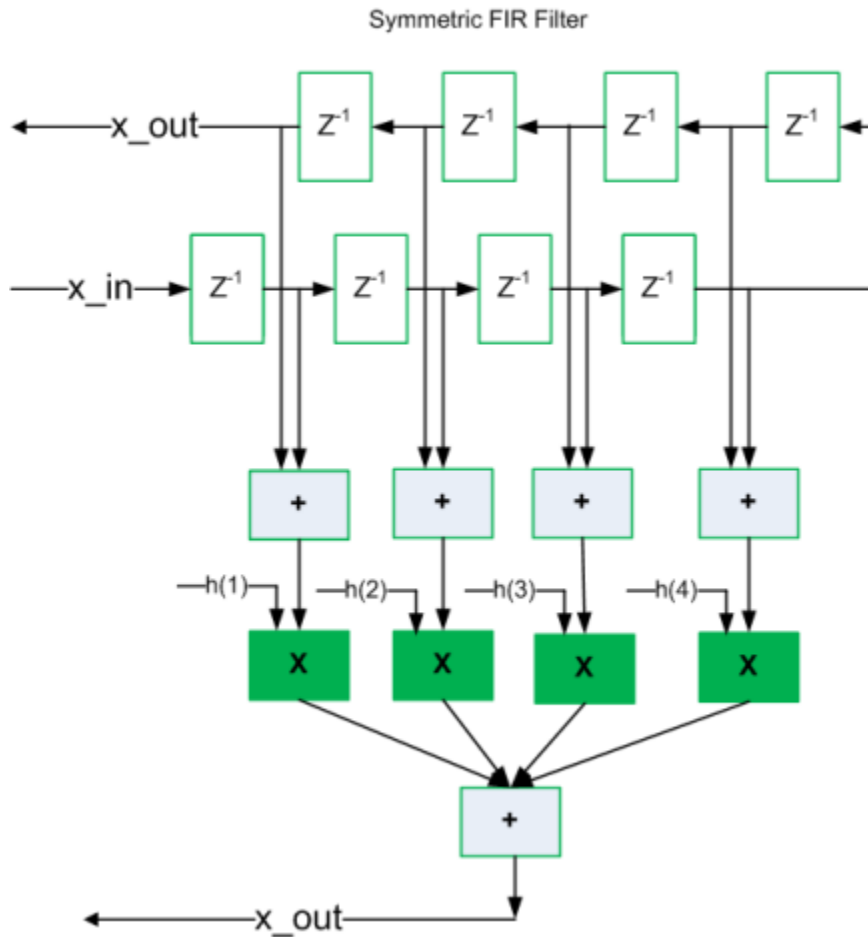
Examine the Resource Report

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

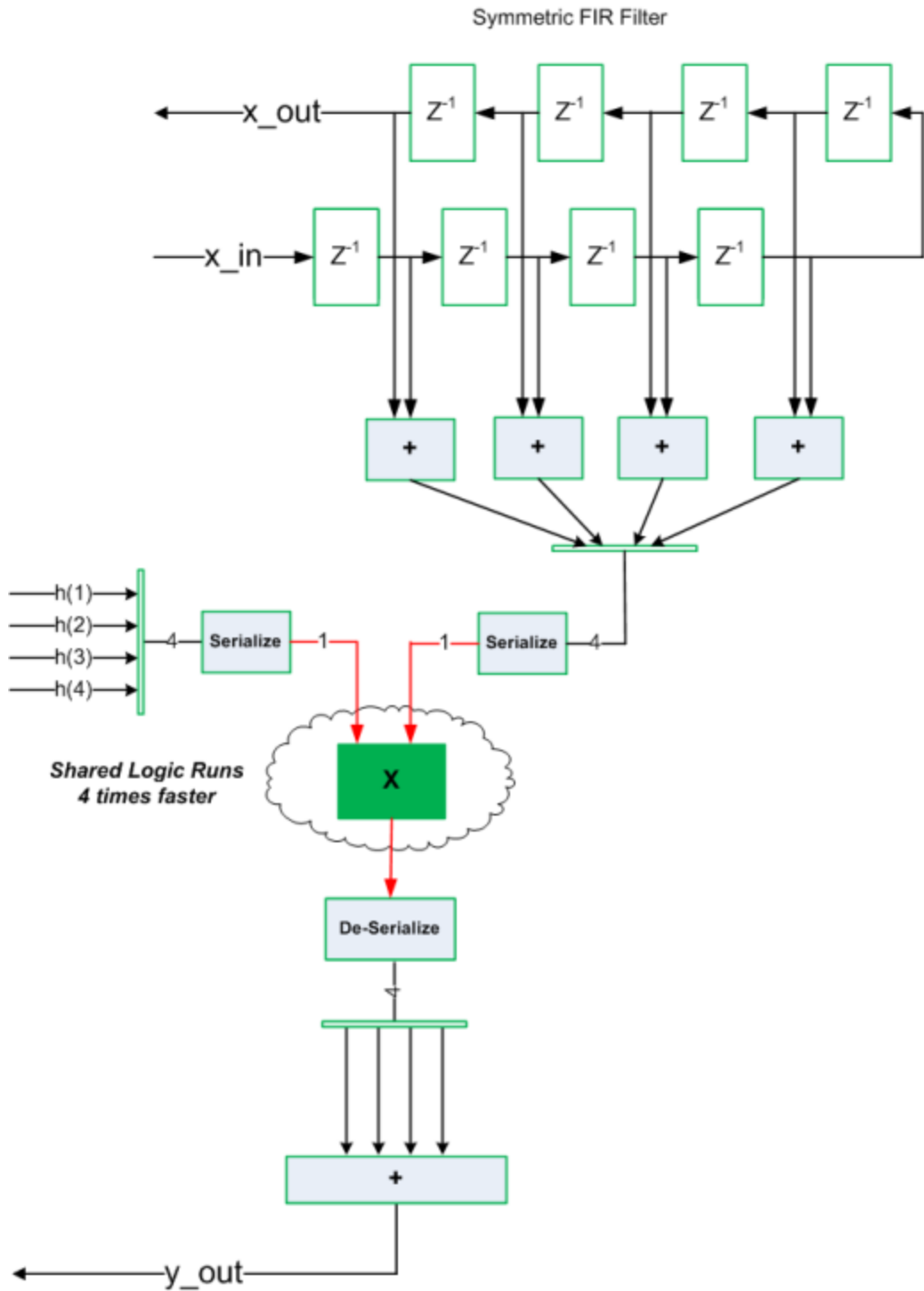
Sharing Architecture

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.

Note that you can only share those multipliers with same type of outputs and operands. You can ensure this happens in the fixed-point conversion step, by using the same numeric type for variables 'a1', 'a2', 'a3' and 'a4'; e.g. `numerictype(1,14,12)`.

The detailed example Fixed-point conversion derived ranges provides a tutorial for updating the type proposal settings during fixed-point conversion.

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.

Known Limitations

Sharing two or more multipliers requires that operands of all the multipliers match exactly in terms of numeric type, size, and complexity.

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'];
```

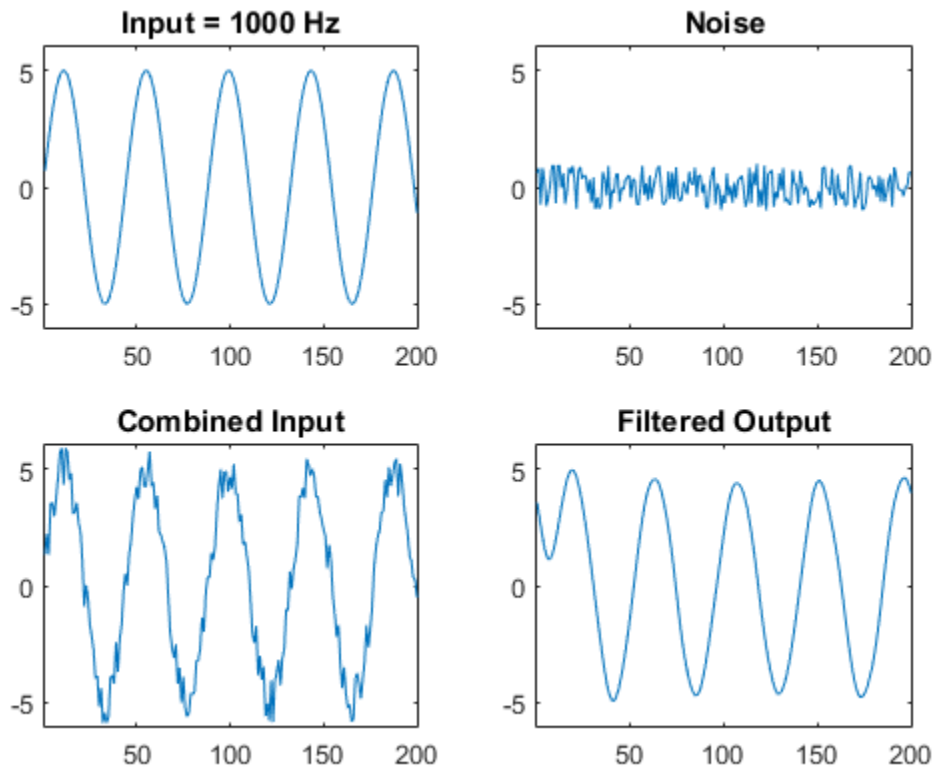
```
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);
```

```
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
```

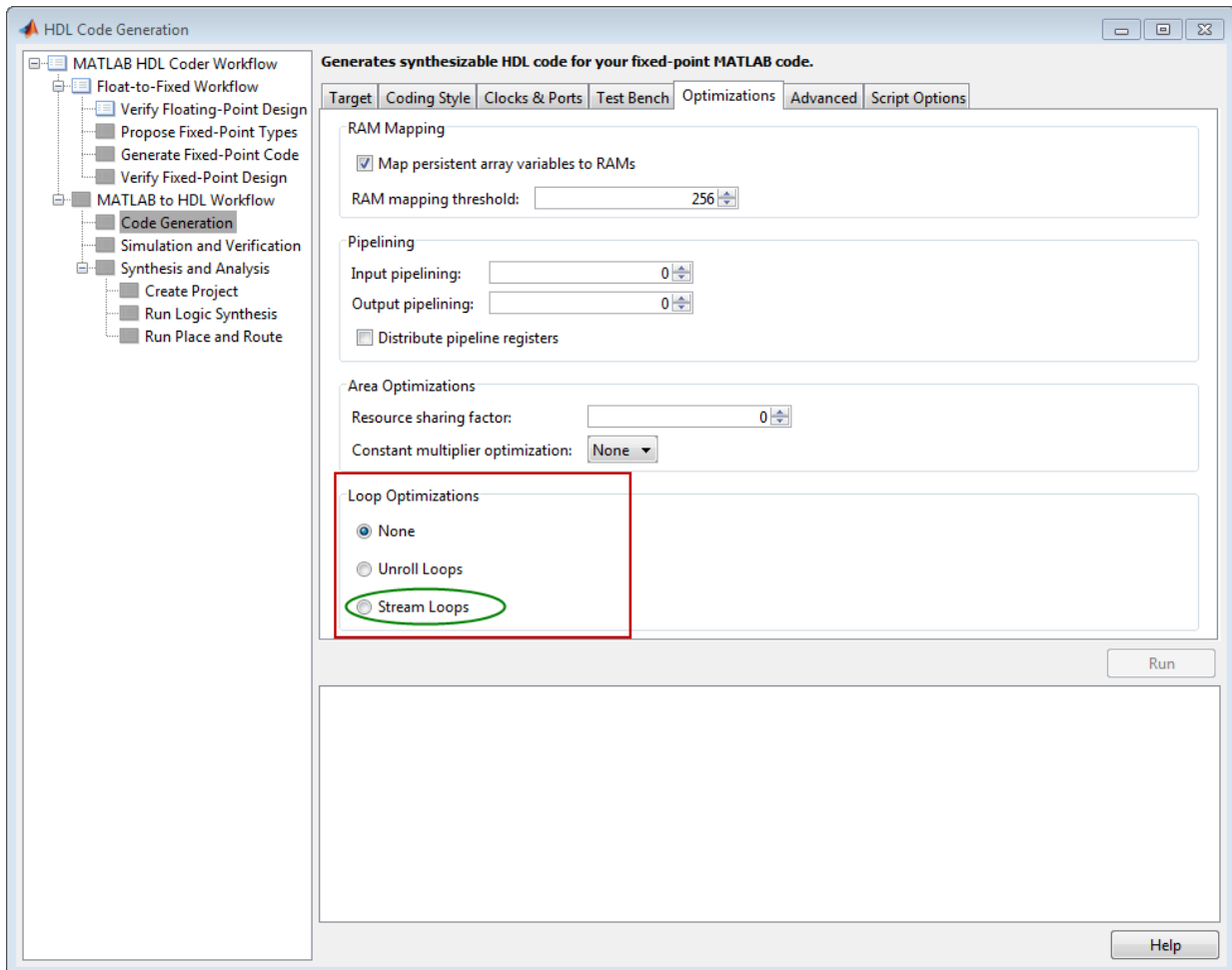
Next, add the file 'mlhdlc_fir.m' to the project as the MATLAB Function and 'mlhdlc_fir_tb.m' as the MATLAB Test Bench.

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.

Introduction

This tutorial shows how the use of canonical signed digit (CSD) representation of multiplier constants (for example, in gain coefficients or filter coefficients) can significantly reduce 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.

$$X_{10} = \sum_{r=0}^{B-1} x_r \cdot 2^r$$

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 non-zero elements.

Here are some properties of CSD numbers:

- 1 No two consecutive bits in a CSD number are non-zero
- 2 CSD representation is guaranteed to have minimum number of non-zero bits

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

```

FCSD Multiplier

A combination of factorization and CSD representation of a constant multiplier can lead to further reduction in hardware cost (number of adders).

FCSD can further reduce the number of adders in the above constant multiplier:

```

y = 231 * x
y = (7 * 33) * x
y_tmp = (x << 5) + x
y = (y_tmp << 3) - y_tmp           % cost of FCSD: 2 Adders

```

CSD/FCSD Costs

This table shows the costs (C) of all 8-bit multipliers.

C	Coefficient
0	1, 2, 4, 8, 16, 32, 64, 128, 256
1	3, 5, 6, 7, 9, 10, 12, 14, 15, 17, 18, 20, 24, 28, 30, 31, 33, 34, 36, 40, 48, 56, 60, 62, 63, 65, 66, 68, 72, 80, 96, 112, 120, 124, 126, 127, 129, 130, 132, 136, 144, 160, 192, 224, 240, 248, 252, 254, 255
2	11, 13, 19, 21, 22, 23, 25, 26, 27, 29, 35, 37, 38, 39, 41, 42, 44, 46, 47, 49, 50, 52, 54, 55, 57, 58, 59, 61, 67, 69, 70, 71, 73, 74, 76, 78, 79, 81, 82, 84, 88, 92, 94, 95, 97, 98, 100, 104, 108, 110, 111, 113, 114, 116, 118, 119, 121, 122, 123, 125, 131, 133, 134, 135, 137, 138, 140, 142, 143, 145, 146, 148, 152, 156, 158, 159, 161, 162, 164, 168, 176, 184, 188, 190, 191, 193, 194, 196, 200, 208, 216, 220, 222, 223, 225, 226, 228, 232, 236, 238, 239, 241, 242, 244, 246, 247, 249, 250, 251, 253
3	43, 45, 51, 53, 75, 77, 83, 85, 86, 87, 89, 90, 91, 93, 99, 101, 102, 103, 105, 106, 107, 109, 115, 117, 139, 141, 147, 149, 150, 151, 153, 154, 155, 157, 163, 165, 166, 167, 169, 170, 172, 174, 175, 177, 178, 180, 182, 183, 185, 186, 187, 189, 195, 197, 198, 199, 201, 202, 204, 206, 207, 209, 210, 212, 214, 215, 217, 218, 219, 221, 227, 229, 230, 231, 233, 234, 235, 237, 243, 245
4	171, 173, 179, 181, 203, 205, 211, 213
Minimum costs through factorization	
2	45 = 5 × 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 New Project From the Command Line

Create a new project by entering the following command:

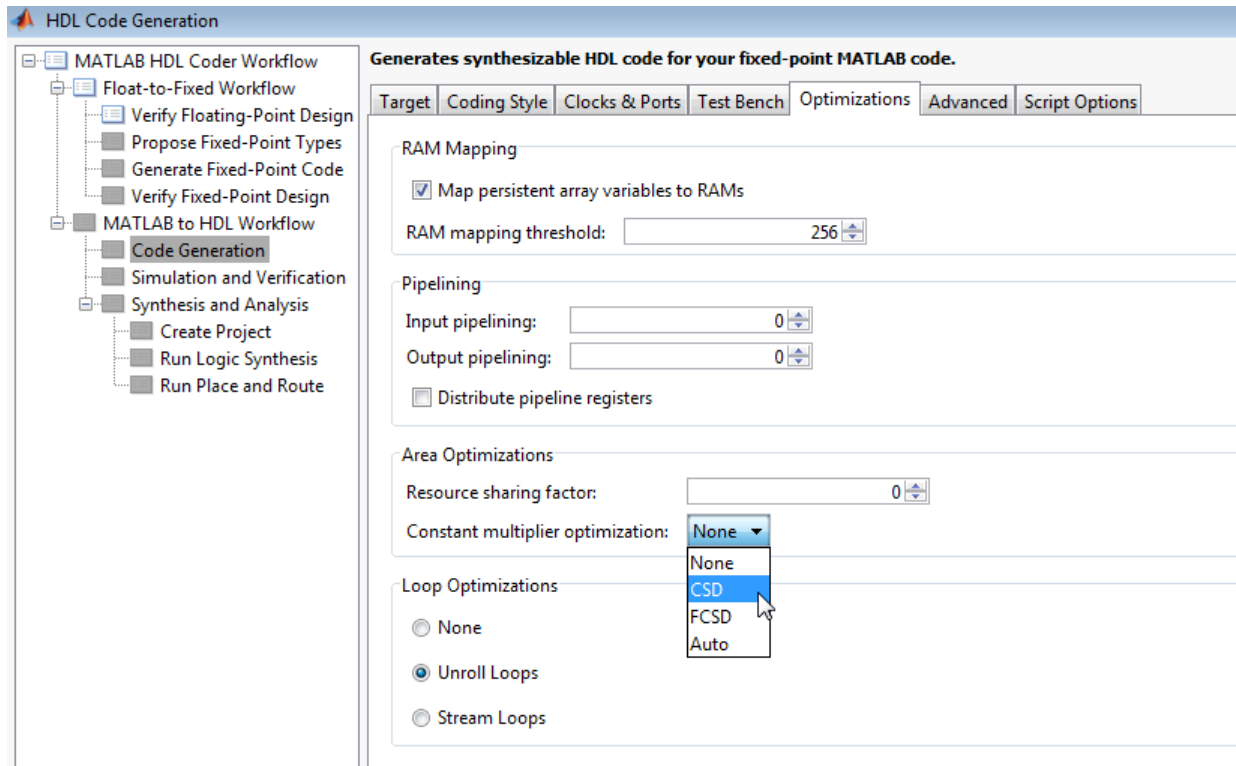
```
coder -hdlcoder -new csd_prj
```

Next, add the file 'mlhdlc_csd.m' to the project as the MATLAB Function and 'mlhdlc_csd_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.

Explore CSD Optimization

Look in the Optimizations tab to explore the constant multiplier optimization options.



Generate Code without Constant Multiplier Optimization

- 1 Launch the Workflow Advisor.
- 2 Click the 'Code Generation' step.
- 3 In the Optimizations tab, leave the 'Constant multiplier optimization' option as 'None'.
- 4 Enable the 'Unroll Loops' option to inline multiplier constants.
- 5 Right-click 'Code Generation' and choose 'Run the task' to run all the steps from the beginning through HDL code generation.
- 6 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

- 1 Launch the Workflow Advisor.
- 2 Click the 'Code Generation' step.
- 3 In the Optimizations tab, choose 'CSD as the 'Constant multiplier optimization' option.
- 4 Enable the 'Unroll Loops' option to inline multiplier constants.
- 5 Right-click 'Code Generation and select 'Run the task' to run all the steps from the beginning through HDL code generation.
- 6 Examine the generated code.

```

329 -- filtered output
330 --'mldlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
331 -- CSD Encoding (2194) : 0100010010010; Cost (Adders) = 3
332 p22y_out_mul_temp <= - (((resize(a1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a1 & '0' &
333 p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
334 -- CSD Encoding (1373) : 0101011001'01; Cost (Adders) = 5
335 p22y_out_mul_temp_1 <= - (((((resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '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_cast <= p22y_out_add_cast + p22y_out_add_cast_1;
338 p22y_out_add_cast_2 <= resize(p22y_out_add_cast, 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' & '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' & '0', 28) + resize(a4 & '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

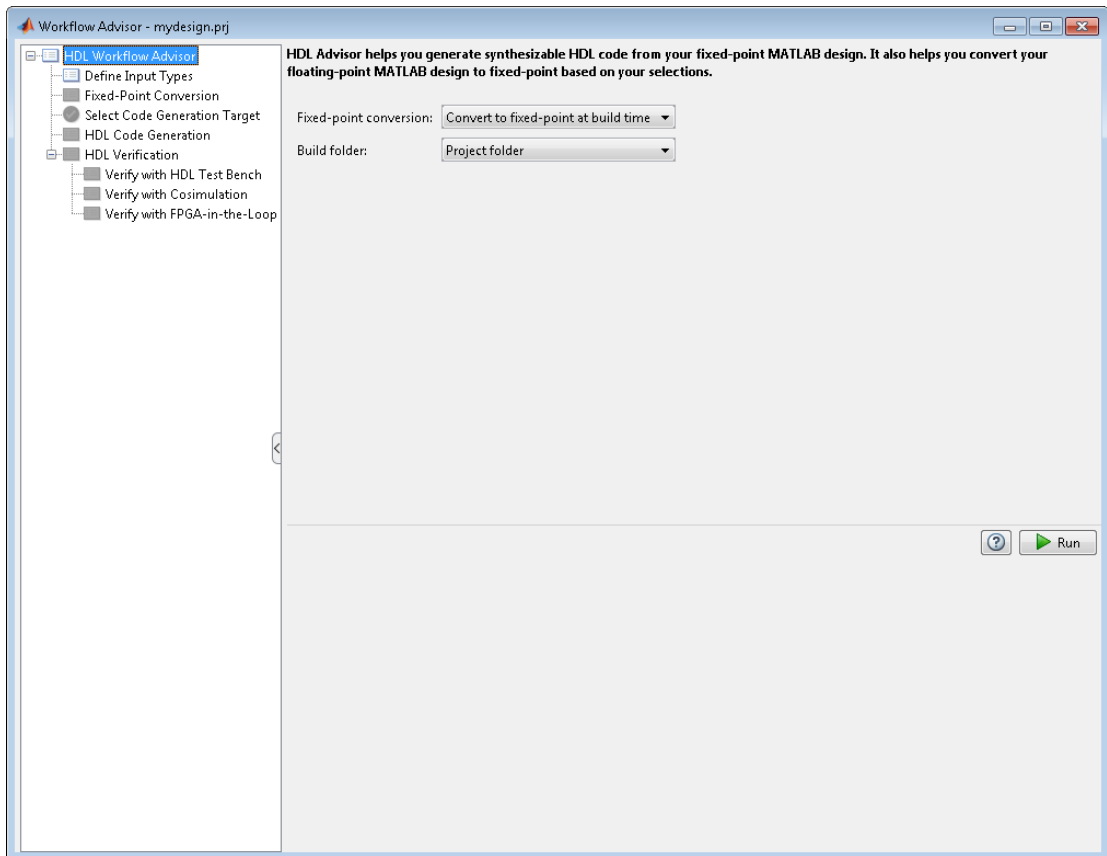
Generate Code with FCSD Optimization

- 1 Launch the Workflow Advisor.
- 2 Click the 'Code Generation' step.
- 3 In the Optimizations tab, choose 'FCSD' as the 'Constant multiplier optimization' option.
- 4 Enable the 'Unroll Loops' option to inline multiplier constants.

HDL Workflow Advisor Reference

- “HDL Workflow Advisor” on page 8-2
- “MATLAB to HDL Code and Synthesis” on page 8-6

HDL Workflow Advisor



Overview

The HDL Workflow Advisor is a tool that supports a suite of tasks covering the stages of the ASIC and FPGA design process, including converting floating-point MATLAB algorithms to fixed-point algorithms. Some tasks perform code validation or checking; others run the HDL code generator or third-party tools. Each folder at the top level of the HDL Workflow Advisor contains a group of related tasks that you can select and run.

Use the HDL Workflow Advisor to:

- Convert floating-point MATLAB algorithms to fixed-point algorithms.

If you already have a fixed-point MATLAB algorithm, set **Design needs conversion to Fixed Point?** to **NO** to skip this step.

- Generate HDL code from fixed-point MATLAB algorithms.
- Simulate the HDL code using a third-party simulation tool.
- Synthesize the HDL code and run a mapping process that maps the synthesized logic design to the target FPGA.
- Run a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.

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.

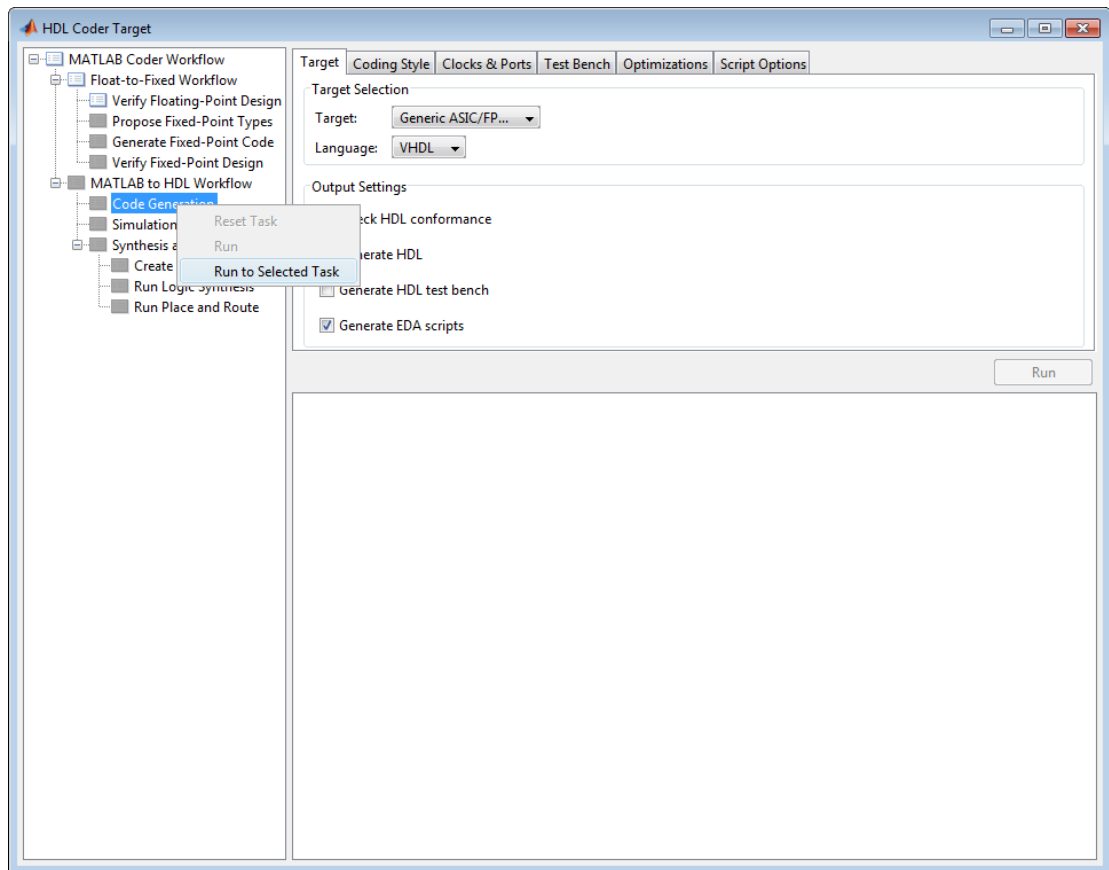
Alternatively, right-click the task to open the context menu. From the context menu, select **Run** to run the selected task.

- 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 8-6

“Code Generation: Target Tab” on page 8-6

“Code Generation: Coding Style Tab” on page 8-7

“Code Generation: Clocks and Ports Tab” on page 8-9

“Code Generation: Test Bench Tab” on page 8-11

“Code Generation: Optimizations Tab” on page 8-13

“Simulation and Verification” on page 8-15

“Synthesis and Analysis” on page 8-15

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 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 string, 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 string to resolve duplicate VHDL entity or Verilog module names in generated code.

Default: _block

Reserved word postfix

Specify a string to append to value names, postfix values, or labels that are VHDL or Verilog reserved words.

Default: _rsvd

Clocked process postfix

Specify a string to append to HDL clock process names.

Default: _process

Complex real part postfix

Specify a string to append to real part of complex signal names.

Default: '_re'

Complex imaginary part postfix

Specify a string to append to imaginary part of complex signal names.

Default: '_im'

Pipeline postfix

Specify a string to append to names of input or output pipeline registers.

Default: '_pipe'

Enable prefix

Specify the base name string 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 string 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 string appended to names of reference signals generated in test bench code.

Default: '_ref'

Ignore data checking (number of samples)

Specify number of samples during which output data checking is suppressed.

Default: 0

Use fiaccel to accelerate test bench logging

To generate a test bench, HDL Coder simulates the original MATLAB code. Use the Fixed-Point Designer `fiaccel` function to accelerate this simulation and accelerate test bench logging.

Default: On

Code Generation: Optimizations Tab

Optimization settings

Input Parameters

Map persistent array variables to RAMs

Select to map persistent array variables to RAMs instead of mapping to shift registers.

Default: Off

Dependencies:

- **RAM Mapping Threshold**
- **Persistent variable names for RAM Mapping**

RAM Mapping Threshold

Specify the minimum RAM size required for mapping persistent array variables to RAMs.

Default: 256

Persistent variable names for RAM Mapping

Provide the names of the persistent variables to map to RAMs.

Default: None

Input Pipelining

Specify number of pipeline registers to insert at top level input ports. Can improve performance and help to meet timing constraints.

Default: 0

Output Pipelining

Specify number of pipeline registers to insert at top level output ports. Can improve performance and help to meet timing constraints.

Default: 0

Distribute Pipeline Registers

Reduces critical path by changing placement of registers in design. Operates on all registers, including those inserted using the **Input Pipelining** and **Output Pipelining** parameters, and internal design registers.

Default: Off

Sharing Factor

Number of additional sources that can share a single resource, such as a multiplier. To share resources, set **Sharing Factor** to 2 or higher; a value of 0 or 1 turns off sharing.

In a design that performs identical multiplication operations, HDL Coder can reduce the number of multipliers by the sharing factor. This can significantly reduce area.

Default: 0

Simulation and Verification

Simulates the generated HDL code using the selected simulation tool.

Input Parameters

Simulation tool

Lists the available simulation tools.

Default: None

Skip this step

Default: Off

Results and Recommended Actions

Conditions	Recommended Action
No simulation tool available on system path.	Add your simulation tool path to the MATLAB system path, then restart MATLAB. For more information, see “Synthesis Tool Path Setup”.

Synthesis and Analysis

This folder contains tasks to create a synthesis project for the HDL code. The task then runs the synthesis and, optionally, runs place and route to generate a circuit description suitable for programming an ASIC or FPGA.

Input Parameters

Skip this step

Default: Off

Skip this step if you are interested only in simulation or you do not have a synthesis tool.

Create Project

Create synthesis project for supported synthesis tool.

Description

This task creates a synthesis project for the selected synthesis tool and loads the project with the HDL code generated for your MATLAB algorithm.

You can select the family, device, package, and speed that you want.

When the project creation is complete, the HDL Workflow Advisor displays a link to the project in the right pane. Click this link to view the project in the synthesis tool's project window.

Input Parameters**Synthesis Tool**

Select from the list:

- **Altera Quartus II**

Generate a synthesis project for Altera Quartus II. When you select this option, HDL Coder sets:

- **Chip Family** to Stratix II
- **Device Name** to EP2S60F1020C4

You can manually change these settings.

- **Xilinx ISE**

Generate a synthesis project for Xilinx ISE. When you select this option, HDL Coder:

- Sets **Chip Family** to Virtex4
- Sets **Device Name** to xc4vsx35
- Sets **Package Name** to ff6...
- Sets **Speed Value** to -...

You can manually change these settings.

Default: No Synthesis Tool Specified

When you select **No Synthesis Tool Specified**, HDL Coder does not generate a synthesis project. It clears and disables the fields in the **Synthesis Tool Selection** pane.

Chip Family

Target device family.

Default: None

Device Name

Specific target device, within selected family.

Default: None

Package Name

Available package choices. The family and device determine these choices.

Default: None

Speed Value

Available speed choices. The family, device, and package determine these choices.

Default: None

Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails to create project.	Read the error message returned by synthesis tool, then check the synthesis tool version, and check that you have write permission for the project folder.
Synthesis tool does not appear in dropdown list.	Add your synthesis tool path to the MATLAB system path, then restart MATLAB. For more information, see “Synthesis Tool Path Setup”.

Run Logic Synthesis

Launch selected synthesis tool and synthesize the generated HDL code.

Description

This task:

- Launches the synthesis tool in the background.
- Opens the previously generated synthesis project, compiles HDL code, synthesizes the design, and emits netlists and related files.

- Displays a synthesis log in the **Result** subpane.

Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails when running place and route.	Read the error message returned by the synthesis tool, modify the MATLAB code, then rerun from the beginning of the HDL Coder workflow.

Run Place and Route

Launches the synthesis tool in the background and runs a Place and Route process.

Description

This task:

- Launches the synthesis tool in the background.
- Runs a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.
- Displays a log in the Result subpane.

Input Parameters

Skip this step

If you select **Skip this step**, the HDL Workflow Advisor executes the workflow, but omits the Perform Place and Route, marking it Passed. You might want to select **Skip this step** if you prefer to do place and route work manually.

Default: Off

Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails when running place and route.	Read the error message returned by the synthesis tool, modify the MATLAB code, then rerun from the beginning of the HDL Coder workflow.

HDL Code Generation from Simulink

Model Design for HDL Code Generation

- “Signal and Data Type Support” on page 9-2
- “Simulink Templates For HDL Code Generation” on page 9-4
- “Generate DUT Ports For Tunable Parameters” on page 9-19

Signal and Data Type Support

In this section...
“Overview” on page 9-2
“Buses” on page 9-2
“Enumerations” on page 9-2
“Unsupported Signal and Data Types” on page 9-3

Overview

HDL Coder supports code generation for Simulink signal types and data types with a few special cases.

Buses

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.

Bus Support Limitations

You cannot generate code for designs that use the following:

- Array of buses
- Black box subsystem connected to a bus
- Black box model reference connected to a bus

Enumerations

You can generate code for Simulink, MATLAB, or Stateflow enumerations within your design.

Requirements

- The enumeration values must be monotonically increasing.

- 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

Unsupported Signal and Data Types

Variable-size signals are not supported for code generation.

More About

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

Simulink Templates For HDL Code Generation

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. You can use `hdlsetup` to configure an existing model for HDL code generation.


To learn how to model hardware for efficient HDL code generation, create a model using an HDL Coder model template.

In this section...

“Create Model Using HDL Coder Model Template” on page 9-4

“HDL Coder Model Templates” on page 9-4

Create Model Using HDL Coder Model Template

- 1 Open the Simulink Library Browser. Click the **Simulink Library** button or, at the command prompt, enter `simulink`.
- 2 Click the New Model arrow  and select **From Template**.
- 3 In the Simulink Template Gallery, under HDL Coder, double-click a model template.

A new model, with the template contents and settings, opens in the Simulink Editor. Click **File > Save as** to save the model.

HDL Coder Model Templates

- “Complex Multiplier” on page 9-5
- “MATLAB Arithmetic” on page 9-6
- “ROM” on page 9-9
- “Register” on page 9-10
- “SRL” on page 9-12
- “Simulink Hardware Patterns” on page 9-14
- “State Machine in MATLAB” on page 9-17

Complex Multiplier

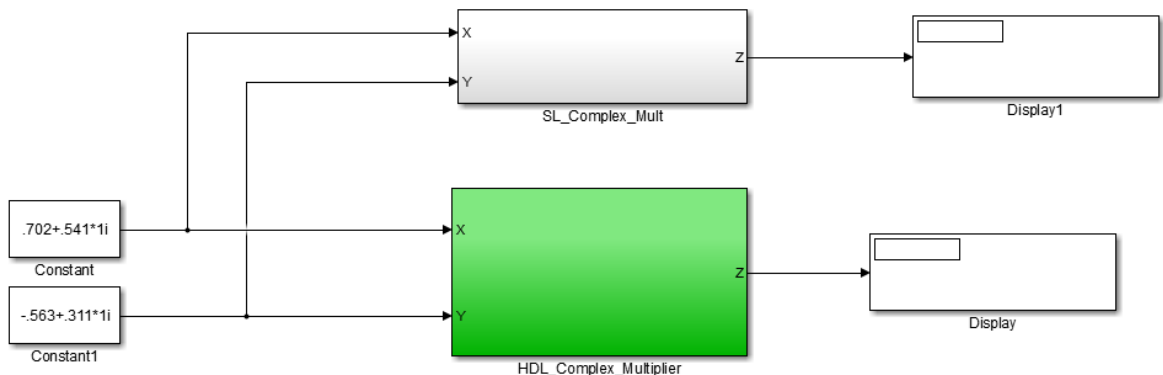
The Complex Multiplier template shows how to model a complex multiplier-accumulator and manually pipeline the intermediate stages. The hardware implementation of complex multiplication uses four multipliers and two adders.

The template applies the following best practices:

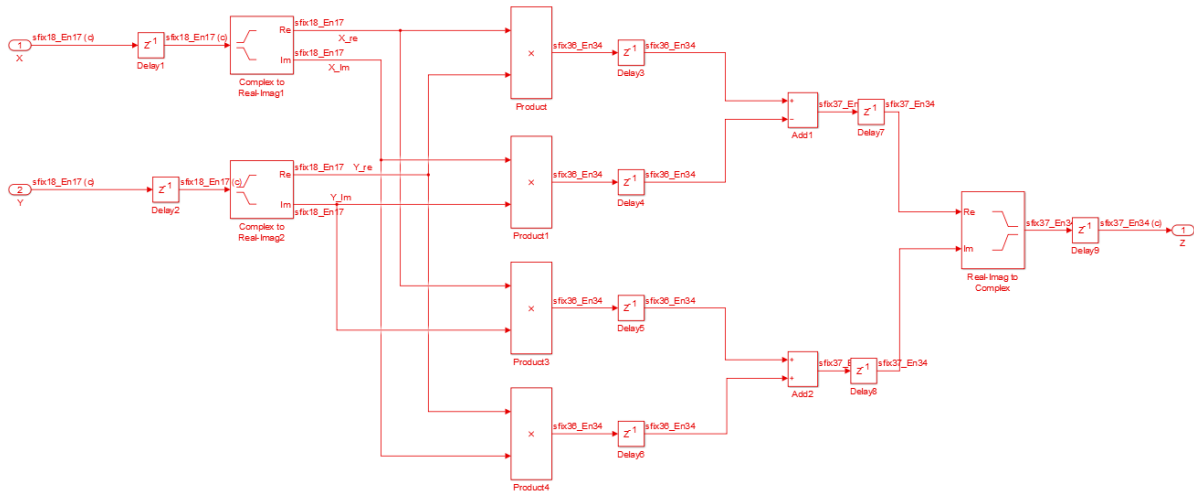
- In the Configuration Parameters dialog box, in **HDL Code Generation > Global Settings**, **Reset type** is set to **Synchronous**.
- To improve speed, Delay blocks, which map to registers in hardware, are at the inputs and outputs of the multipliers and adders.
- To support the output data of a full-precision 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.

untitled ▶



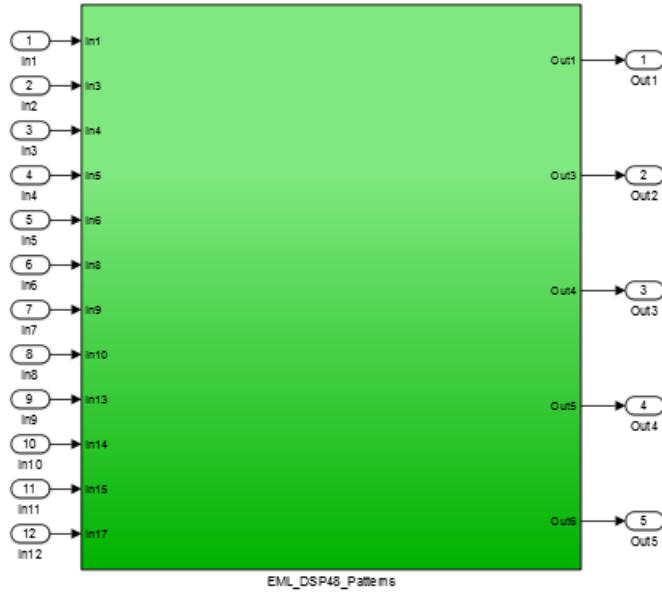
untitled HDL_Complex_Multiplier



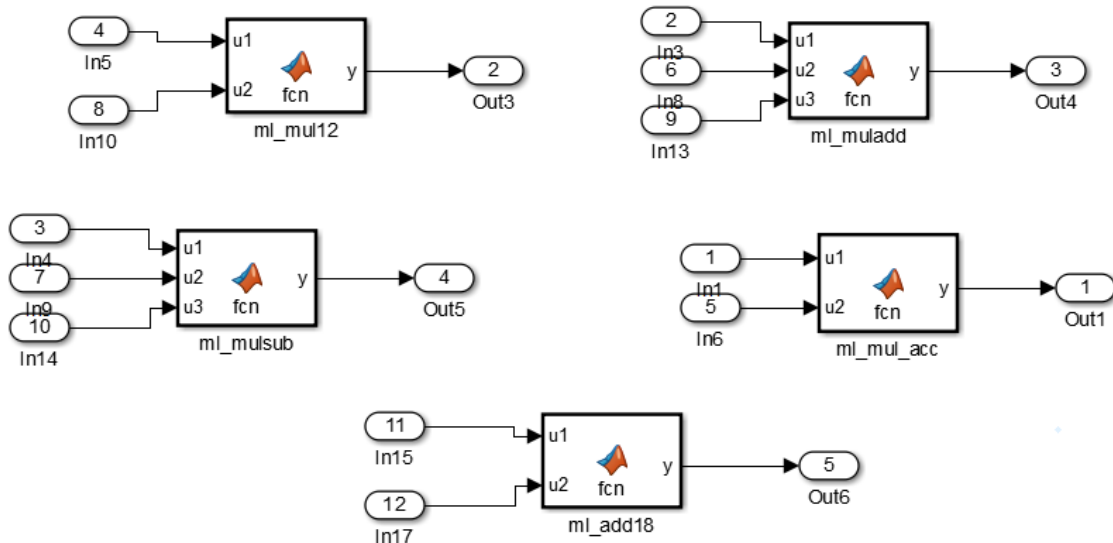
MATLAB Arithmetic

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

untitled ▶



untitled ▶ EML_DSP48_Patterns



For example, the `m1_mul_acc` MATLAB Function block shows how to write a multiply-accumulate operation in MATLAB. `hdlfimath` applies fixed-point math settings that are appropriate 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.

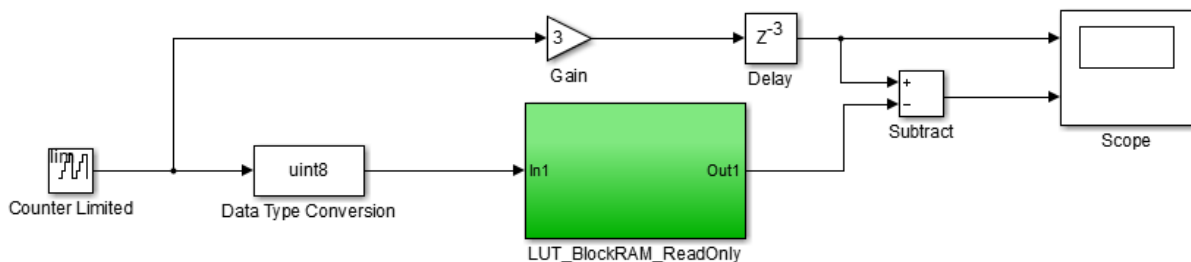
The template applies the following best practices:

- 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.

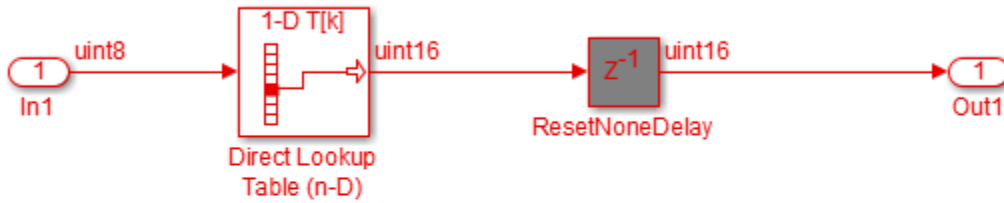
Using table dimensions that are a power of two enables HDL Coder to generate shift operations instead of division operations. If necessary, pad the table with zeros.

- 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 ▶



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)];
```

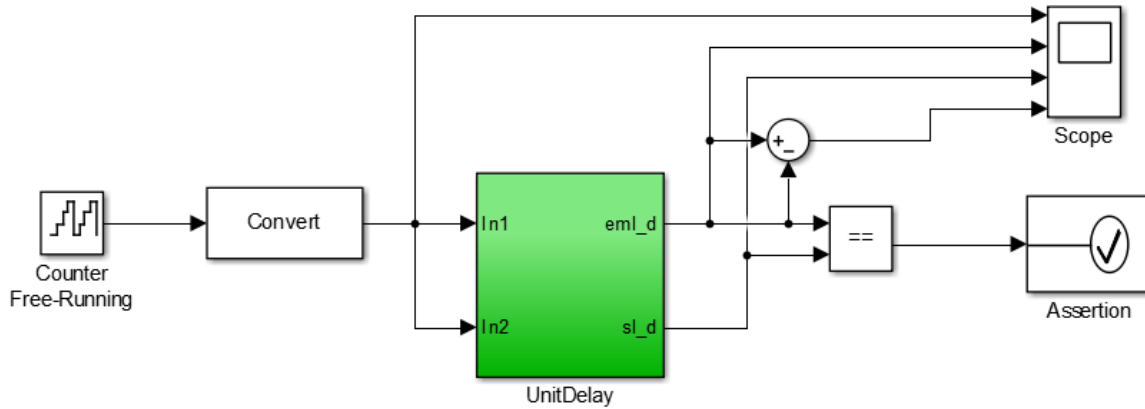
Register

The Register template shows how to model hardware registers:

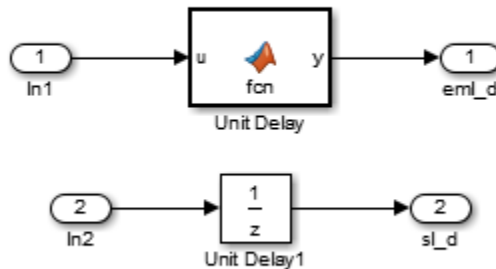
- In Simulink, using the Delay block.
- In MATLAB, using persistent variables.

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

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

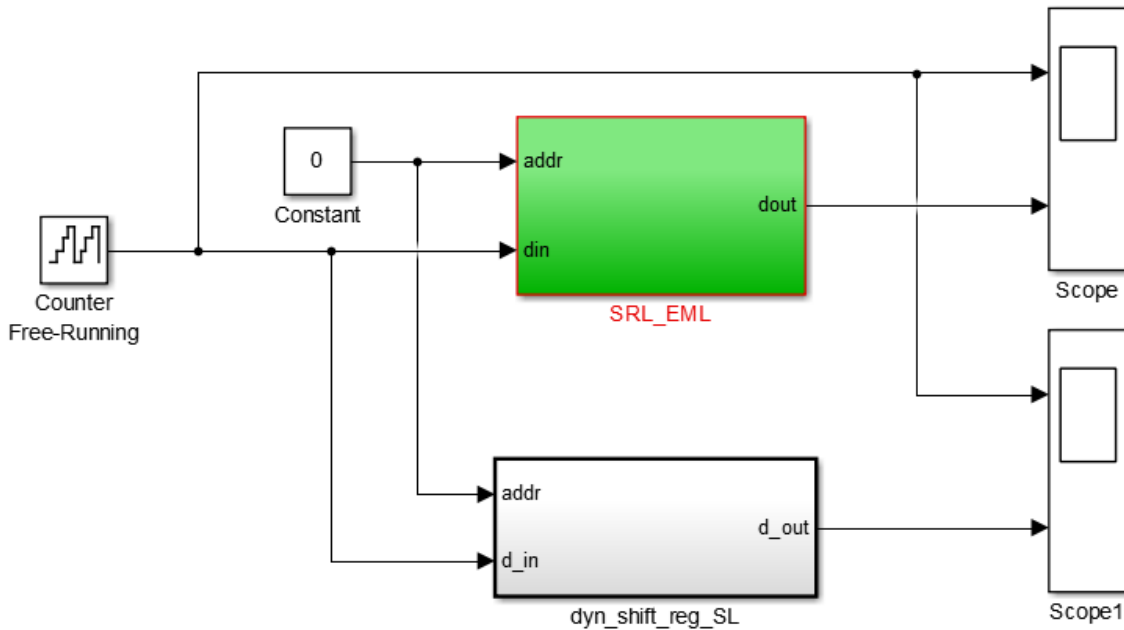
% return delayed input from last sample time hit
y = u_d;

% store the current input
u_d = u;
```

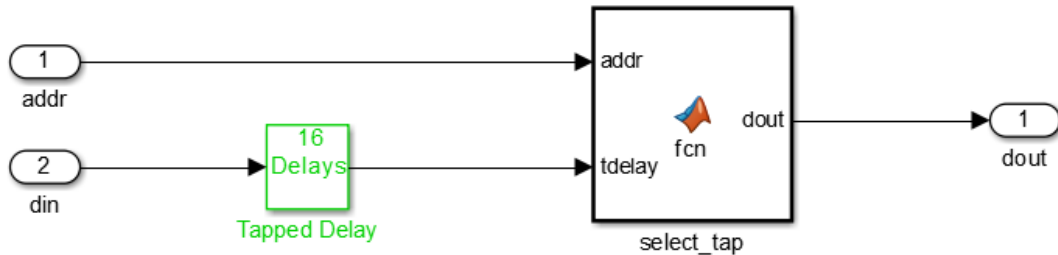
SRL

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 ▶



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` is incremented 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:

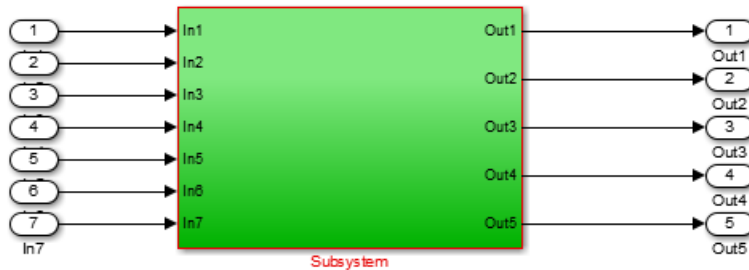
- In the Block Parameters dialog box, **Include current input in output vector** is not enabled.
- In the HDL Block Properties dialog box, **ResetType** is set to none.
- For the Subsystem block, in the HDL Block Properties dialog box, **FlattenHierarchy** is set to on.

Simulink Hardware Patterns

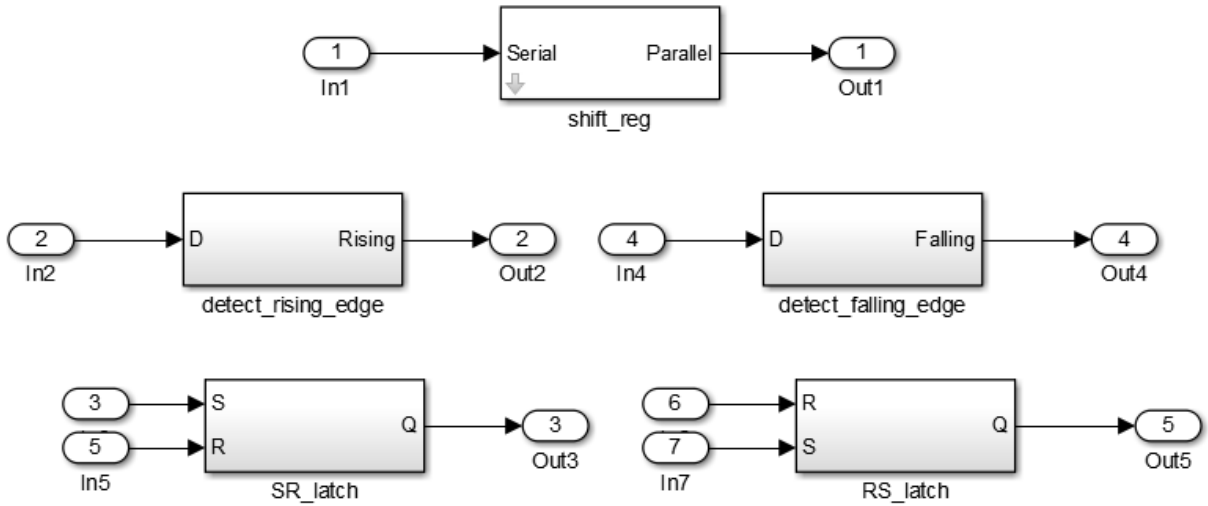
The Simulink Hardware Patterns template contains design patterns for common hardware operations:

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

untitled2 ▶

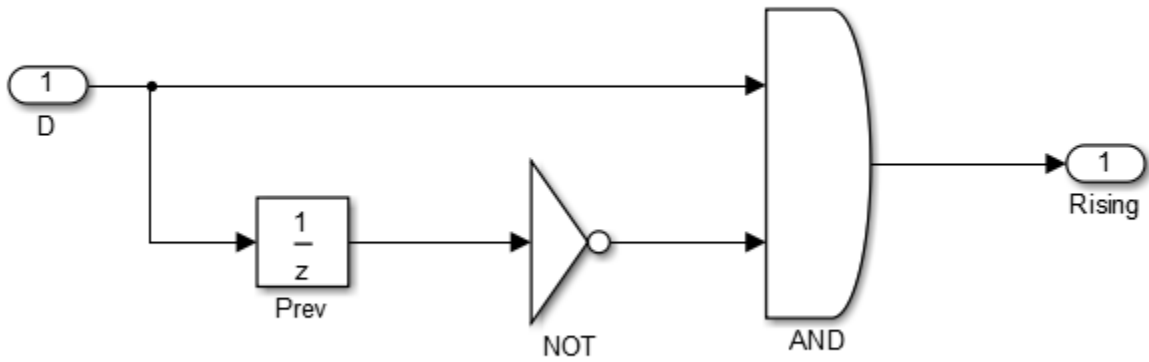


untitled ▶ ▶ Subsystem ▶

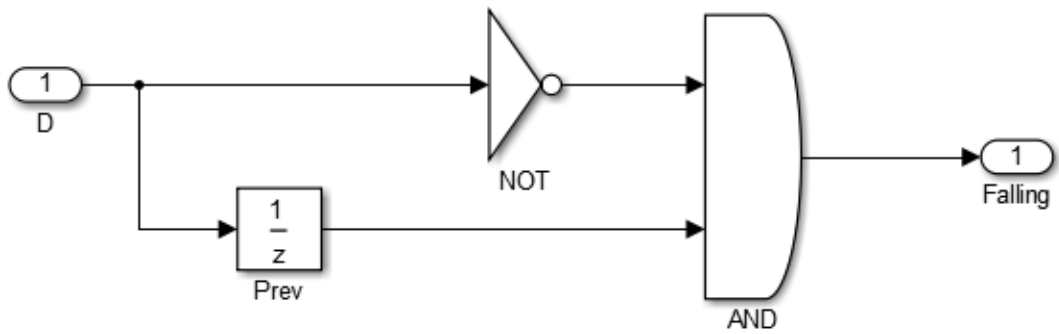


For example, the design patterns for rising edge detection and falling edge detection:

untitled ▸ Subsystem ▸ detect_rising_edge

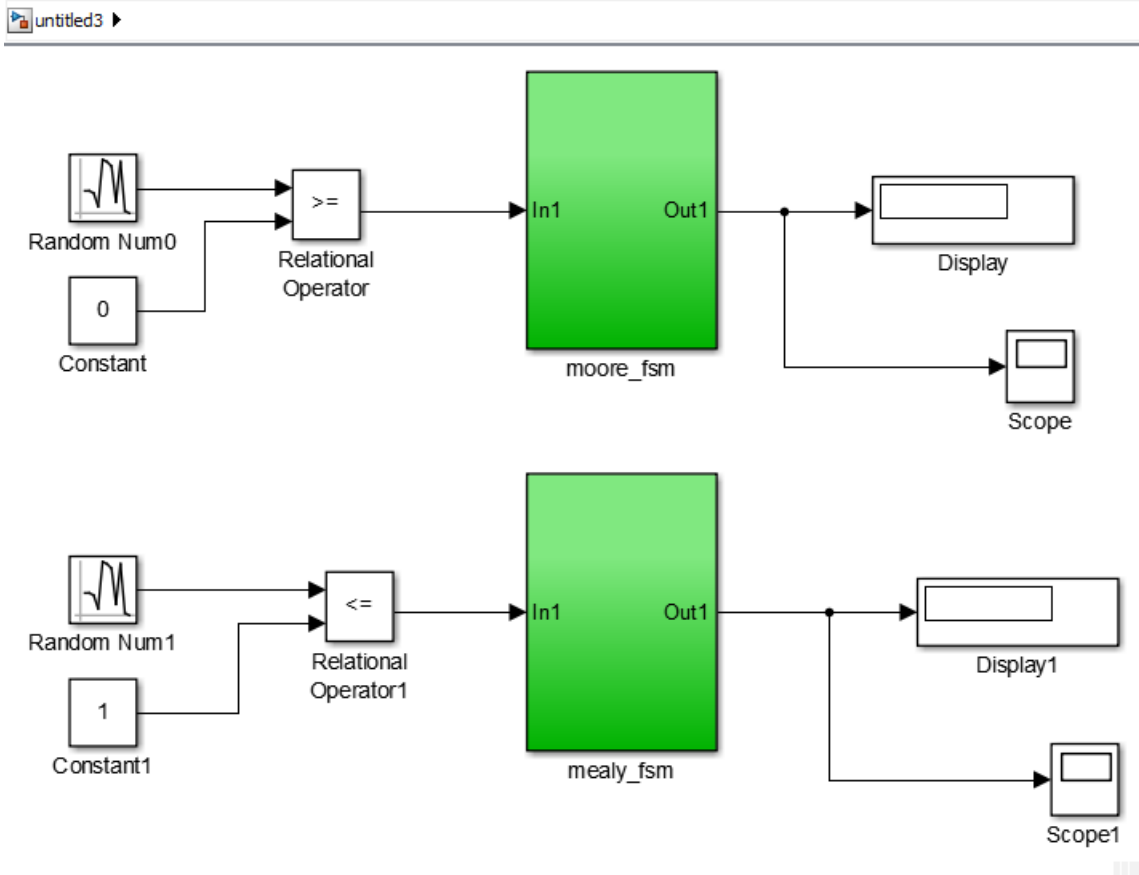


untitled ▸ Subsystem ▸ detect_falling_edge



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”.

See Also

`hdlsetup` | `makehdl`

More About

- “Prepare Simulink Model For HDL Code Generation”
- “Design Guidelines for the MATLAB Function Block”
- “Hardware Modeling with MATLAB Code”

Generate DUT Ports For Tunable Parameters

In this section...

“Create and Add Tunable Parameter that Maps to DUT Ports” on page 9-19

“Generated Code” on page 9-20

“Limitations” on page 9-21

“Use Tunable Parameter in Other Blocks” on page 9-21

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 as the value in one of the following blocks:

- Gain
- Constant

The Gain or Constant with the tunable parameter can be at any level of the DUT hierarchy, including within a model reference.

If you use a tunable parameter in a block other than Gain or Constant, code is generated inline and does not map to DUT ports. Therefore, to use the value of a tunable parameter in a MATLAB Function, Chart, or Truth Table block, add an input port, and connect the new input port to the output of a Constant block.

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 “Using the Model Explorer to Create Data Objects”.

- 2 In your Simulink design, in a Gain or Constant block, enter the tunable parameter:
 - As the **Constant value** in a Constant block.
 - As the **Gain** parameter in a Gain 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*:

```
ENTITY s IS
  PORT( In1          : IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En5
        myParam      : IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En5
        Out1         : OUT   std_logic_vector(31 DOWNTO 0) -- sfix32_En10
        );
END s;

ARCHITECTURE rtl OF s IS

  -- Signals
  SIGNAL myParam_signed  : signed(15 DOWNTO 0); -- sfix16_En5
  SIGNAL In1_signed      : signed(15 DOWNTO 0); -- sfix16_En5
  SIGNAL Gain_out1       : signed(31 DOWNTO 0); -- sfix32_En10

BEGIN
  myParam_signed <= signed(myParam);

  In1_signed <= signed(In1);

  Gain_out1 <= myParam_signed * In1_signed;

  Out1 <= std_logic_vector(Gain_out1);

END rtl;
```


Limitations

You cannot use HDL cosimulation with a DUT that uses tunable parameters in Gain or Constant blocks.

Use Tunable Parameter in Other Blocks

To use the value of a tunable parameter in a MATLAB Function, Chart, or Truth Table block:

- 1** Create the tunable parameter and use it in a Constant block.
- 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.

Code Generation Options in the HDL Coder Dialog Boxes

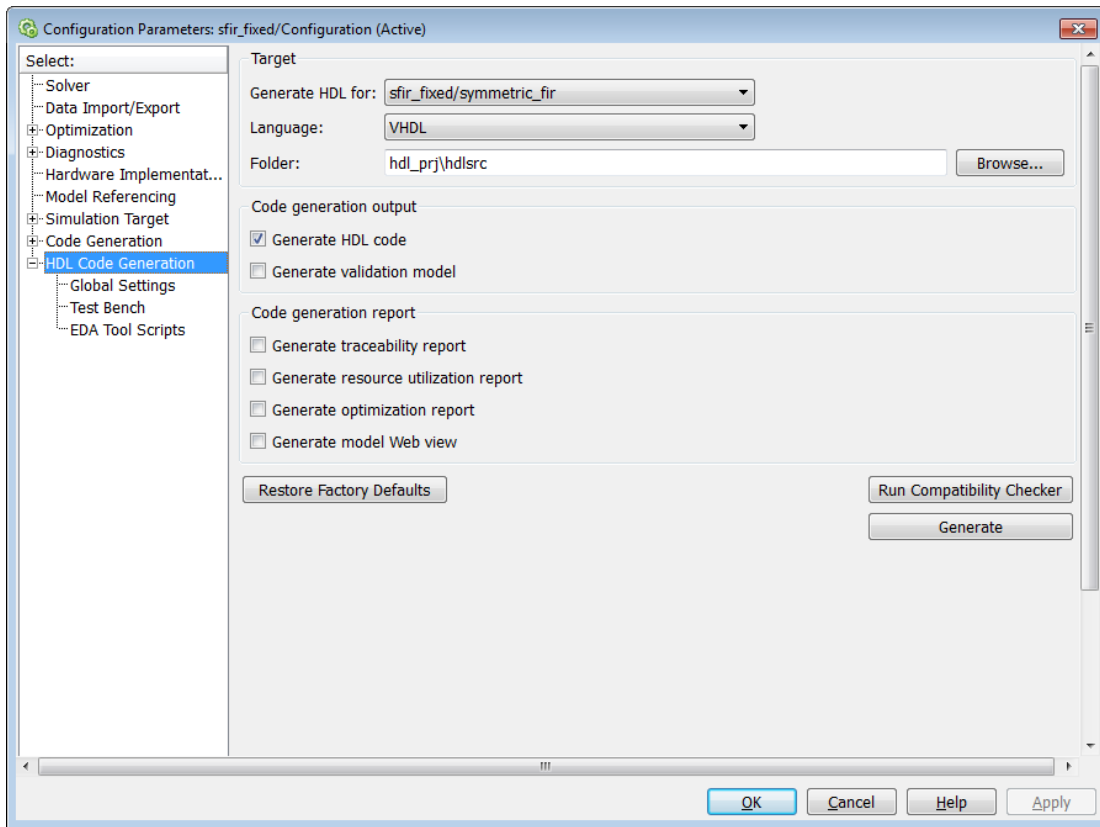
- “Set HDL Code Generation Options” on page 10-2
- “HDL Code Generation Pane: General” on page 10-8
- “HDL Code Generation Pane: Global Settings” on page 10-22
- “HDL Code Generation Pane: Test Bench” on page 10-104
- “HDL Code Generation Pane: EDA Tool Scripts” on page 10-130

Set HDL Code Generation Options

In this section...
“HDL Code Generation Options in the Configuration Parameters Dialog Box” on page 10-2
“HDL Code Generation Options in the Model Explorer” on page 10-3
“Code Menu” on page 10-4
“HDL Code Options in the Block Context Menu” on page 10-5
“The HDL Block Properties Dialog Box” on page 10-6

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, select **Simulation > Model Configuration Parameters** in the Simulink window. Then select **HDL Code Generation** from the list on the left.



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.

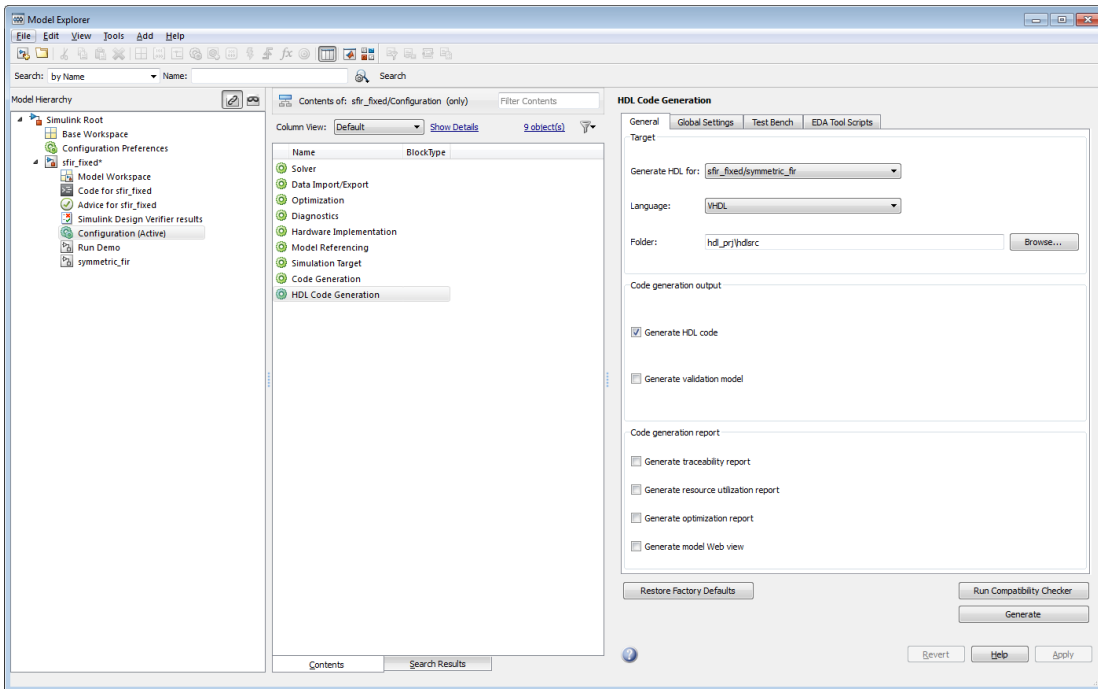
HDL Code Generation Options in the Model Explorer

The following figure shows the top-level **HDL Code Generation** pane as displayed in the Contents pane of the Model Explorer.

To view this dialog box:

- 1 Open the Model Explorer.

- 2 Select your model's active configuration set in the **Model Hierarchy** tree on the left.
- 3 Select **HDL Code Generation** from the list in the Contents pane.



Code Menu

The **Code > HDL Code** submenu provides shortcuts to the HDL code generation options. You can also use this submenu to initiate code generation.

Options include:

- **HDL Workflow Advisor:** Open the HDL Workflow Advisor.
- **Options:** Open the **HDL Code Generation** pane in the Configuration Parameters dialog box.
- **Generate HDL:** Initiate HDL code generation; equivalent to the **Generate** button in the Configuration Parameters dialog box or Model Explorer.
- **Generate Test Bench:** Initiate test bench code generation; equivalent to the **Generate Test Bench** button in the Configuration Parameters dialog box or Model

Explorer. If you do not select a subsystem in the **Generate HDL for** menu, the **Generate Test Bench** menu option is not available.

- **Add HDL Coder Configuration to Model or Remove HDL Coder 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. If you need to add or remove the HDL Code Generation configuration component to or from a model, use this option to do so. For more information, see “Add or Remove the HDL Configuration Component” on page 15-41.

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 coder 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.

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.
Generate HDL for Subsystem	Runs the HDL code generator (<code>makehdl</code>) 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	Available for blocks or subsystems.

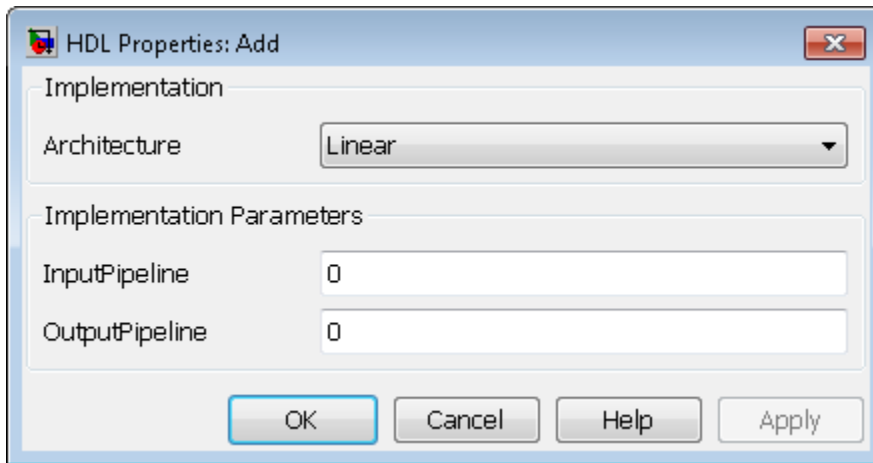
Option	Description	Availability
	HDL Block Parameters” for more information.	
HDL Workflow Advisor	Opens the HDL Workflow Advisor for the subsystem.	Available only for subsystems.
Navigate to Code	Activates the HTML code generation report window, displaying the beginning of the code generated for the selected block or subsystem. See “Tracing from Model to Code” on page 15-20 for more information.	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.

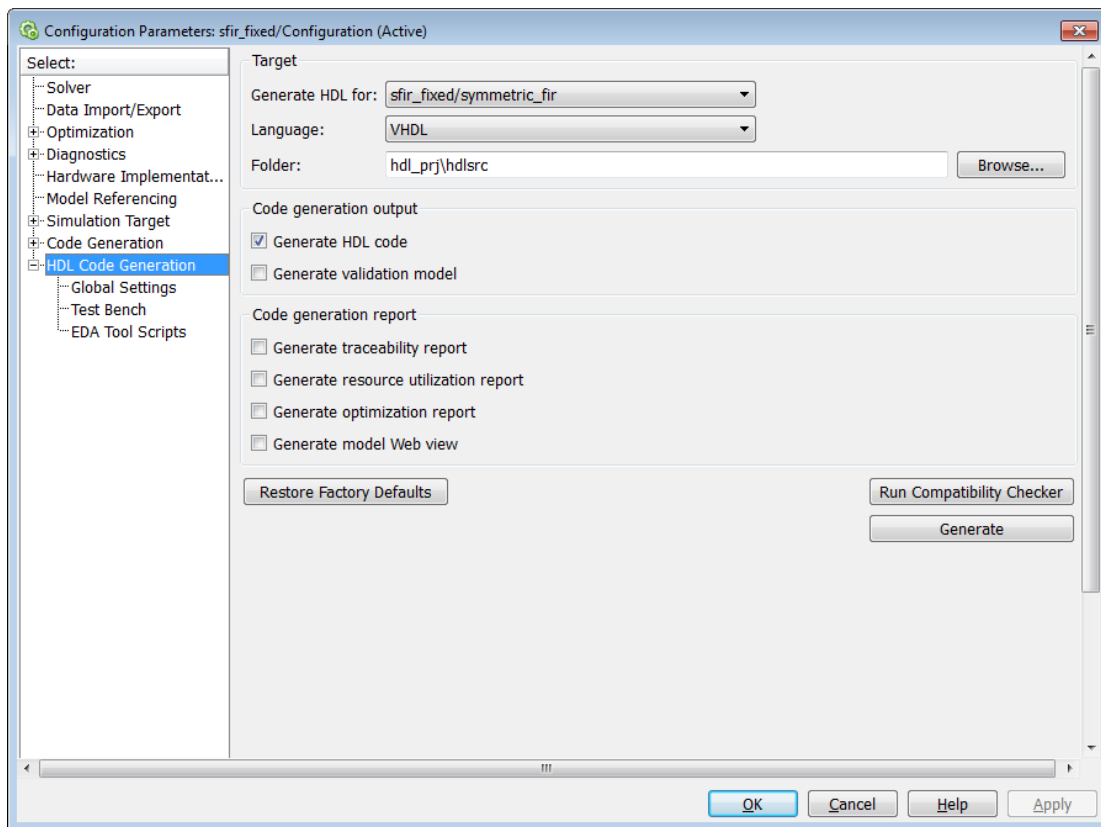
Most block implementations support a number of *implementation parameters* that let you control further details of code generation for the block. The HDL Properties dialog box lets you set implementation parameters for a block.

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 Block Parameters” for detailed information.

HDL Code Generation Pane: General



In this section...

“HDL Code Generation Top-Level Pane Overview” on page 10-10

“Generate HDL for” on page 10-12

“Language” on page 10-13

“Folder” on page 10-14

“Generate HDL code” on page 10-15

“Generate validation model” on page 10-16

“Generate traceability report” on page 10-17

In this section...

“Generate resource utilization report” on page 10-18

“Generate high-level timing critical path report” on page 10-19

“Generate optimization report” on page 10-19

“Generate model Web view” on page 10-21

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 Factory Defaults: Sets model parameters to their default values.

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.

Settings

Default: The root model is selected.

Command-Line Information

Pass in the path to the model or subsystem for which code is to be generated as the first argument to `makehdl`.

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.

Settings

Default: VHDL

VHDL

Generate VHDL code.

Verilog

Generate Verilog code.

Command-Line Information

Property: TargetLanguage

Type: string

Value: 'VHDL' | 'Verilog'

Default: 'VHDL'

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

See Also

- TargetLanguage
- 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.

Settings

Default: The default target folder is a subfolder of your working folder, named `hdlsrc`.

Command-Line Information

Property: `TargetDirectory`

Type: `string`

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`.

See Also

- `TargetDirectory`
- `makehdl`

Generate HDL code

Enable or disable HDL code generation for the model.

Settings

Default: On



On

Generate HDL code.



Off

Do not generate HDL code.

Command-Line Information

Property: GenerateHDLCode

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

GenerateHDLCode

Generate 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 both the original and the generated DUT models.

If you enable generation of a validation model, also enable delay balancing to keep 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

Generate the validation model.

Off

Do not generate the validation model.

Command-Line Information

Property: GenerateValidationModel

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

GenerateValidationModel, BalanceDelays

Generate traceability report

Enable or disable generation of an HTML code generation report with hyperlinks from code to model and model to code.

Settings

Default: Off

On

Create and display an HTML code generation report. See “Create and Use Code Generation Reports”.

Off

Do not create an HTML code generation report.

Command-Line Information

Property: Traceability

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

Traceability

Generate resource utilization report

Enable or disable generation of an HTML resource utilization report

Settings

Default: Off

On

Create and display an HTML resource utilization report. The report contains information about the number of hardware resources (multipliers, adders, registers) used in the generated HDL code. The report includes hyperlinks to the referenced blocks in the model. See “Create and Use Code Generation Reports”.

Off

Do not create an HTML resource utilization report.

Command-Line Information

Property: ResourceReport

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

ResourceReport

Generate high-level timing critical path report

Specify whether to generate a highlighting script that shows the estimated critical path.

Settings

Default: Off

On

Generate a highlighting script that shows the estimated critical path.

The highlighting script is displayed as a link you can click to highlight the estimated critical path in the generated model. If your design contains blocks without timing information, another highlighting script is generated to highlight those blocks.

Off

Do not calculate the estimated critical path.

Command-Line Information

Property: CriticalPathEstimation

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

- `CriticalPathEstimation`
- `CriticalPathEstimationFile`
- `BlocksWithNoCharacterizationFile`
- “Find Estimated Critical Paths Without Synthesis Tools”

Generate optimization report

Enable or disable generation of an HTML optimization report

Settings

Default: Off

On

Create and display an HTML optimization report. The report contains information about the results of streaming, sharing, and distributed pipelining optimizations that were implemented in the generated code. The report includes hyperlinks back to referenced blocks, subsystems, or validation models. See “Create and Use Code Generation Reports”.

Off

Do not create an HTML optimization report.

Command-Line Information

Property: OptimizationReport

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

OptimizationReport

Generate model Web view

Include the model Web view in the code generation report to navigate between the code and model within the same window. You can share your model and generated code outside of the MATLAB environment. You must have a Simulink Report Generator™ license to include a Web view of the model in the code generation report.

Settings

Default: Off

On

Include model Web view in the code generation report.

Off

Omit model Web view in the code generation report.

Command-Line Information

Parameter: GenerateWebview

Type: string

Value: 'on' | 'off'

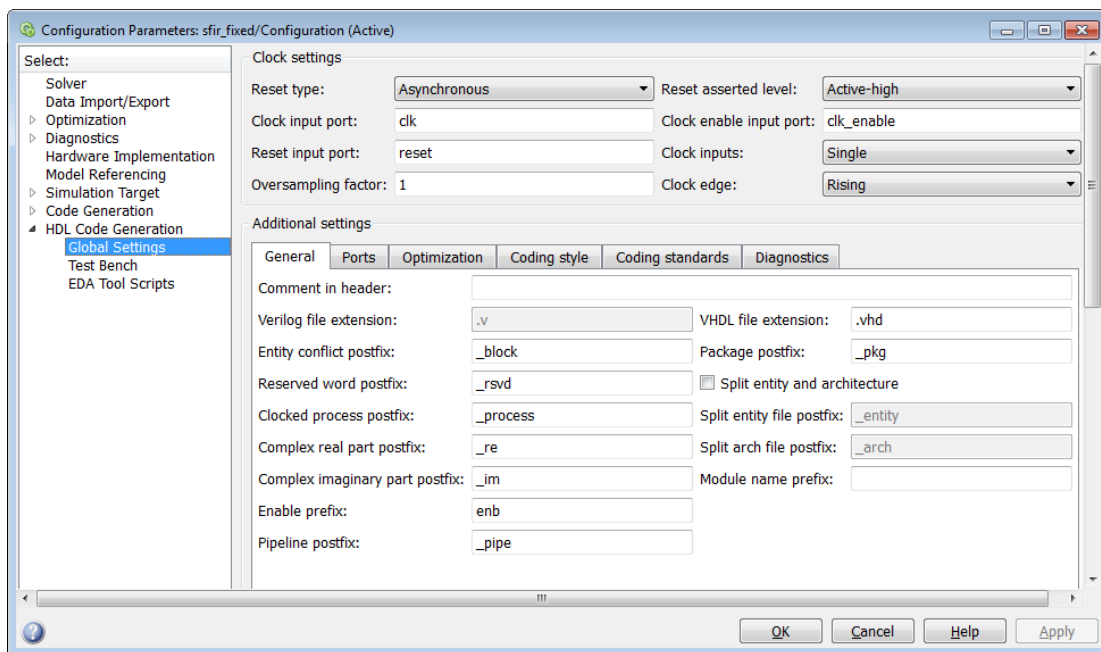
Default: 'off'

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

See Also

“Web View of Model in Code Generation Report”

HDL Code Generation Pane: Global Settings



In this section...

“Global Settings Overview” on page 10-26

“Reset type” on page 10-27

“Reset asserted level” on page 10-28

“Clock input port” on page 10-29

“Clock enable input port” on page 10-30

“Reset input port” on page 10-31

“Clock inputs” on page 10-32

“Oversampling factor” on page 10-33

“Clock edge” on page 10-33

“Comment in header” on page 10-35

“Verilog file extension” on page 10-36

In this section...

- “VHDL file extension” on page 10-37
- “Entity conflict postfix” on page 10-38
- “Package postfix” on page 10-39
- “Reserved word postfix” on page 10-40
- “Split entity and architecture” on page 10-42
- “Clocked process postfix” on page 10-43
- “Complex real part postfix” on page 10-43
- “Complex imaginary part postfix” on page 10-44
- “Split entity file postfix” on page 10-44
- “Split arch file postfix” on page 10-46
- “Module name prefix” on page 10-47
- “Enable prefix” on page 10-48
- “Pipeline postfix” on page 10-49
- “Timing controller postfix” on page 10-49
- “Generate VHDL code for model references into a single library” on page 10-50
- “Input data type” on page 10-51
- “Output data type” on page 10-53
- “Clock enable output port” on page 10-55
- “Balance delays” on page 10-56
- “Use trigger signal as clock” on page 10-56
- “Optimize timing controller” on page 10-57
- “Minimize clock enables” on page 10-58
- “Multiplier partitioning threshold” on page 10-60
- “Max oversampling” on page 10-61
- “RAM mapping threshold (bits)” on page 10-62
- “Timing controller architecture” on page 10-62
- “Max computation latency” on page 10-63
- “Hierarchical distributed pipelining” on page 10-64
- “Clock-rate pipelining” on page 10-65

In this section...

- “Allow clock-rate pipelining of DUT output ports” on page 10-65
- “Preserve design delays” on page 10-66
- “Distributed pipelining priority” on page 10-67
- “Adders” on page 10-68
- “Multipliers” on page 10-68
- “Atomic subsystems” on page 10-69
- “MATLAB Function blocks” on page 10-70
- “Adder sharing minimum bitwidth” on page 10-70
- “Multiplier sharing minimum bitwidth” on page 10-71
- “Represent constant values by aggregates” on page 10-73
- “Use “rising_edge/falling_edge” style for registers” on page 10-74
- “Loop unrolling” on page 10-75
- “Use Verilog ``timescale` directives” on page 10-76
- “Inline VHDL configuration” on page 10-77
- “Concatenate type safe zeros” on page 10-78
- “Emit time/date stamp in header” on page 10-79
- “Scalarize vector ports” on page 10-80
- “Minimize intermediate signals” on page 10-81
- “Include requirements in block comments” on page 10-82
- “Inline MATLAB Function block code” on page 10-83
- “Generate parameterized HDL code from masked subsystem” on page 10-84
- “Initialize all RAM blocks” on page 10-85
- “RAM Architecture” on page 10-85
- “HDL coding standard” on page 10-86
- “Do not show passing rules in coding standard report” on page 10-87
- “Check for duplicate names” on page 10-87
- “Check for HDL keywords in design names” on page 10-88
- “Check for initial statements that set RAM initial values” on page 10-89
- “Check module, instance, and entity name length” on page 10-90

In this section...

“Check signal, port, and parameter name length” on page 10-91

“Minimize use of clock enable signals” on page 10-92

“Detect usage of reset signals” on page 10-93

“Minimize use of variables” on page 10-94

“Check if-else statement chain length” on page 10-95

“Check if-else statement nesting depth” on page 10-96

“Check multiplier width” on page 10-97

“Check for non-integer constants” on page 10-98

“Check line length” on page 10-99

“Highlight feedback loops inhibiting delay balancing and optimizations” on page 10-100

“Highlight blocks inhibiting clock-rate pipelining” on page 10-101

“Highlight blocks inhibiting distributed pipelining” on page 10-102

Global Settings Overview

The **Global Settings** pane enables you to specify detailed characteristics of the generated code, such as HDL element naming and whether certain optimizations are applied.

Reset type

Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers.

Settings

Default: Asynchronous

Asynchronous

Use asynchronous reset logic.

Synchronous

Use synchronous reset logic.

Command-Line Information

Property: ResetType

Type: string

Value: 'async' | 'sync'

Default: 'async'

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

See Also

ResetType

Reset asserted level

Specify whether the asserted (active) level of reset input signal is active-high or active-low.

Settings

Default: Active-high

Active-high

Asserted (active) level of reset input signal is active-high (1).

Active-low

Asserted (active) level of reset input signal is active-low (0).

Command-Line Information

Property: ResetAssertedLevel

Type: string

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`.

See Also

ResetAssertedLevel

Clock input port

Specify the name for the clock input port in generated HDL code.

Settings

Default: clk

Enter a string value to be used as the clock signal name in generated HDL code. If you specify a string that is 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: string

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`.

See Also

ClockInputPort

Clock enable input port

Specify the name for the clock enable input port in generated HDL code.

Settings

Default: `clk_enable`

Enter a string value to be used as the clock enable input port name in generated HDL code. If you specify a string that is 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`.

Tip

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.

Command-Line Information

Property: `ClockEnableInputPort`

Type: `string`

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`.

See Also

`ClockEnableInputPort`

Reset input port

Enter the name for the reset input port in generated HDL code.

Settings

Default: reset

Enter a string value to be used as the reset input port name in generated HDL code. If you specify a string that is 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`.

Tip

If the reset asserted level is set to active-high, the reset input signal is asserted active-high (1) and the input value must be high (1) for the entity's registers to be reset. If the reset asserted level is set to active-low, the reset input signal is asserted active-low (0) and the input value must be low (0) for the entity's registers to be reset.

Command-Line Information

Property: ResetInputPort

Type: string

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`.

See Also

ResetInputPort

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.

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.

Command-Line Information

Property: ClockInputs

Type: string

Value: 'Single' | 'Multiple'

Default: 'Single'

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

See Also

ClockInputs

Oversampling factor

Specify frequency of global oversampling clock as a multiple of the model's base rate.

Settings

Default: 1.

Oversampling factor specifies the *oversampling factor* of a global oversampling clock. The oversampling factor expresses the desired rate of the global oversampling clock as a multiple of your model's base rate. By default, HDL Coder does not generate a global oversampling clock.

If you want to generate a global oversampling clock:

- The **Oversampling factor** must be an integer greater than or equal to 1.
- In a multirate DUT, other rates in the DUT must divide evenly into the global oversampling 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`.

See Also

Generating a Global Oversampling Clock
Oversampling

Clock edge

Specify active clock edge.

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: string

Value: 'Rising' | 'Falling'

Default: 'Rising'

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

See Also

ClockEdge

Comment in header

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 in the string, the code generator emits single-line comments for each newline.

Command-Line Information

Property: UserComment

Type: string

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

See Also

UserComment

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

This option is enabled when the target language (specified by the **Language** option) is Verilog.

Command-Line Information

Property: VerilogFileExtension

Type: string

Default: '.v'

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

See Also

VerilogFileExtension

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.

Dependencies

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: VHDLFileExtension

Type: string

Default: '.vhd'

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

See Also

VHDLFileExtension

Entity conflict postfix

Specify the string used 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, in the default case, if HDL Coder detects two entities with the name `MyFilt`, the coder names the first entity `MyFilt` and the second instance `MyFilt_entity`.

Command-Line Information

Property: `EntityConflictPostfix`

Type: `string`

Value: A valid string in the target language

Default: `'_block'`

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

See Also

`EntityConflictPostfix`

Package postfix

Specify a string 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: `string`

Value: A string 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`.

See Also

`PackagePostfix`

Reserved word postfix

Specify a string 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: `string`

Default: `'_rsvd'`

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

See Also

`ReservedWordPostfix`

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 string.

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`.

Dependencies

This option is enabled when the target language (specified by the **Language** option) is Verilog.

Selecting this option enables the following parameters:

- **Split entity file postfix**
- **Split architecture file postfix**

Command-Line Information

Property: `SplitEntityArch`

Type: `string`

Value: `'on' | 'off'`

Default: `'off'`

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

See Also

`SplitEntityArch`

Clocked process postfix

Specify a string to append 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 coder derives the label `delay_pipeline_process` from the register name `delay_pipeline` and the default postfix string `_process`.

Command-Line Information

Property: `ClockProcessPostfix`

Type: `string`

Default: `'_process'`

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

See Also

`ClockProcessPostfix`

Complex real part postfix

Specify string to append to real part of complex signal names.

Settings

Default: `'_re'`

Enter a string to be appended to the names generated for the real part of complex signals.

Command-Line Information**Property:** ComplexRealPostfix**Type:** string**Default:** '_re'

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

See Also

ComplexRealPostfix

Complex imaginary part postfix

Specify string to append to imaginary part of complex signal names.

Settings**Default:** '_im'

Enter a string to be appended to the names generated for the imaginary part of complex signals.

Command-Line Information**Property:** ComplexImagPostfix**Type:** string**Default:** '_im'

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

See Also

ComplexImagPostfix

Split entity file postfix

Enter a string to be appended to the model name to form the name of a generated VHDL entity file.

Settings**Default:** `_entity`

Dependencies

This parameter is enabled by **Split entity and architecture**.

Command-Line Information

Property: SplitEntityFilePostfix

Type: string

Default: '_entity'

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

See Also

SplitEntityFilePostfix

Split arch file postfix

Enter a string to be appended to the model name to form the name of a generated VHDL architecture file.

Settings

Default: `_arch`

Dependency

This parameter is enabled by **Split entity and architecture**.

Command-Line Information

Property: `SplitArchFilePostfix`

Type: `string`

Default: `'_arch'`

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

See Also

`SplitArchFilePostfix`

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: string

Default: ''

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

See Also

ModulePrefix

Enable prefix

Specify the base name string 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, multiple clock enables are generated (for example, when a cascade block implementation for certain blocks is specified). 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: string

Default: 'enb'

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

See Also

EnablePrefix

Pipeline postfix

Specify string 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 string that HDL Coder appends to names of input or output pipeline registers.

Command-Line Information

Property: `PipelinePostfix`

Type: `string`

Default: `'_pipe'`

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

See Also

`PipelinePostfix`

Timing controller postfix

Specify the suffix appended 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:** string**Default:** '_tc'

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

See Also

TimingControllerPostfix

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.

Dependencies

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information**Property:** UseSingleLibrary**Type:** string**Value:** 'on' | 'off'**Default:** 'off'

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

See Also

UseSingleLibrary

Input data type

Specify the HDL data type for the model's input ports.

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'`. 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: `string`

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`.

See Also

InputType

Output data type

Specify the HDL data type for the model's output ports.

Settings

For VHDL, the options are:

Default: Same as input data type

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`.

For Verilog, the options are:

Default: `wire`

In generated Verilog code, the data type for all ports is `'wire'`. 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: `string`

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`.

See Also

OutputType

Clock enable output port

Specify the name for the generated clock enable output.

Settings

Default: `ce_out`

A clock enable output is generated when the design requires one.

Command-Line Information

Property: `ClockEnableOutputPort`

Type: `string`

Default: `'ce_out'`

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

See Also

`ClockEnableOutputPort`

Balance delays

Enable delay balancing.

Settings

Default: On

On

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: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

“Delay Balancing”

Use trigger signal as clock

Enable to use trigger input signal as clock in generated HDL code.

Settings

Default: Off

On

For triggered subsystems, use the trigger input signal as a clock in the generated HDL code.

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: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

TriggerAsClock

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 string 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: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

`OptimizeTimingController`

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.

- dspsrcs4/Sine Wave
- hlldemolib/HDL FFT
- built-in/DiscreteFir

- dspmlti4/CIC Decimation
- dspmlti4/CIC Interpolation
- dspmlti4/FIR Decimation
- dspmlti4/FIR Interpolation
- dspadpt3/LMS Filter
- dsparch4/Biquad Filter

Command-Line Information**Property:** MinimizeClockEnables**Type:** string**Value:** 'on' | 'off'**Default:** 'off'

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

See Also`MinimizeClockEnables`**Multiplier partitioning threshold**

Specify the maximum 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 bit width.

To improve hardware mapping results, set the multiplier partitioning threshold to the 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`.

See Also

MultiplierPartitioningThreshold

Max oversampling

Specify the maximum oversampling ratio. The oversampling ratio is the sample rate after optimizations divided by the original model sample rate.

Use **Max oversampling** with **Max computation latency** to prevent or reduce overclocking by constraining resource sharing and streaming optimizations.

Settings

Default: 0

0

Do not set a limit on the maximum sample rate.

1

Do not allow oversampling.

N, where N is an integer greater than 1

Allow oversampling up to N times the original model sample rate.

Command-Line Information

Property: MaxOversampling

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`.

See Also

- MaxOversampling

- MaxComputationLatency

RAM mapping threshold (bits)

Specify the minimum RAM size for mapping to block RAMs instead of to registers.

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`.

See Also

- RAMMappingThreshold
- “UseRAM”
- “MapPersistentVarsToRAM”

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: string

Value: 'resettable' | 'default'

Default: 'default'

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

See Also

TimingControllerArch

Max computation latency

Specify the maximum number of time steps for which your inputs are guaranteed to be stable.

Use **Max computation latency** with **Max oversampling** to prevent or reduce overclocking by constraining resource sharing and streaming optimizations.

Settings

Default: 1

1

DUT input data can change every cycle.

N, where N is an integer greater than 1

DUT input data can change every N cycles.

Command-Line Information

Property: MaxComputationLatency

Type: integer

Value: positive integer

Default: 1

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

See Also

- `MaxComputationLatency`
- `MaxOversampling`

Hierarchical distributed pipelining

Specify that retiming be applied across a subsystem hierarchy.

Settings

Default: Off

On

Enable retiming across a subsystem hierarchy. HDL Coder applies retiming hierarchically down, until it reaches a subsystem where **DistributedPipelining** is off.

Off

Distribute pipelining only within a subsystem.

Command-Line Information

Property: `HierarchicalDistPipelining`

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

- `HierarchicalDistPipelining`
- “DistributedPipelining”

Clock-rate pipelining

Insert pipeline registers at the clock rate instead of the data rate for multi-cycle paths in your design.

Settings

Default: On

On

Insert pipeline registers at clock rate for multi-cycle paths.

Off

Insert pipeline registers at data rate for multi-cycle paths.

Command-Line Information

Property: ClockRatePipelining

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

- `ClockRatePipelining`
- “Clock-Rate Pipelining”

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.

Command-Line Information

Property: ClockRatePipelineOutputPorts

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

- `ClockRatePipelineOutputPorts`
- `ClockRatePipelining`
- “Clock-Rate Pipelining”

Preserve design delays

Enable 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: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

- PreserveDesignDelays
- “DistributedPipelining”

Distributed pipelining priority

Specify priority for 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: string

Value: 'NumericalIntegrity' | 'Performance'

Default: 'NumericalIntegrity'

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

See Also

- DistributedPipeliningPriority
- “DistributedPipelining”

Adders

Enable to share adders with the resource sharing optimization.

Settings

Default: Off

On

When resource sharing is enabled, share adders with a bit width greater than or equal to **Adder sharing minimum bitwidth**.

Off

Do not share adders.

Command-Line Information

Property: ShareAdders

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

- ShareAdders
- AdderSharingMinimumBitwidth
- “Resource Sharing”

Multipliers

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 **Multiplier sharing minimum bitwidth**.

Off

Do not share multipliers.

Command-Line Information

Property: ShareMultipliers

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

- ShareMultipliers
- MultiplierSharingMinimumBitwidth
- “Resource Sharing”

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.

Command-Line Information

Property: ShareAtomicSubsystems

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

- `ShareAtomicSubsystems`
- “Resource Sharing”

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.

Command-Line Information

Property: `ShareMATLABBlocks`

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

- `ShareMATLABBlocks`
- “Resource Sharing”

Adder sharing minimum bitwidth

Minimum bit width for shared adders.

To share only larger adders with the resource sharing optimization, specify the minimum adder bit width.

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.

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`.

See Also

- AdderSharingMinimumBitwidth
- ShareAdders
- “Resource Sharing”

Multiplier sharing minimum bitwidth

Minimum bit width for shared multipliers.

To share only larger multipliers with the resource sharing optimization, specify the minimum multiplier bit width.

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.

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`.

See Also

- MultiplierSharingMinimumBitwidth
- ShareMultipliers
- “Resource Sharing”

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);
```

Dependencies

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: UseAggregatesForConst

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

UseAggregatesForConst

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.

Off

Generated code uses the `'event` syntax.

Dependencies

This option is enabled when the target language is VHDL.

Command-Line Information

Property: UseRisingEdge

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

UseRisingEdge

Loop unrolling

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.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: LoopUnrolling

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

LoopUnrolling

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: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

UseVerilogTimescale

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.

Dependencies

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: InlineConfigurations

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

InlineConfigurations

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.

Dependencies

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: SafeZeroConcat

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

SafeZeroConcat

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: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

DateComment

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.

Dependencies

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: ScalarizePorts

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

ScalarizePorts

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 coder 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: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

MinimizeIntermediateSignals

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”.

Off

Do not include requirements as comments in code or code generation reports.

Command-Line Information

Property: RequirementComments

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

RequirementComments

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: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

InlineMATLABBlockCode

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 HDL file for multiple masked subsystems with different values for tunable mask parameters. HDL Coder automatically detects atomic subsystems with tunable mask parameters that are shareable.

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: string

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”
- `MaskParameterAsGeneric`

Initialize all RAM blocks

Enable or suppress generation of initial signal value for RAM blocks.

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.

Command-Line Information

Property: InitializeBlockRAM

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

`InitializeBlockRAM`

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:** string**Value:** 'WithClockEnable' | 'WithoutClockEnable'**Default:** 'WithClockEnable'

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

See Also

RAMArchitecture

HDL coding standard

Specify an HDL coding standard.

Settings**Default:** None

None

Generate generic synthesizable HDL code.

Industry

Generate HDL code that follows the industry standard rules supported by the HDL Coder software. When this option is enabled, the coder generates a standard compliance report.

Command-Line Information**Property:** HDLCodingStandard**Type:** string**Value:** 'None' | 'Industry'**Default:** 'None'

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

See Also

HDLCodingStandard

Do not show passing rules in coding standard report

Specify whether to show rules without errors or warnings in the coding standard report.

Settings

Default: Off

- On
Show only rules with errors or warnings.
- Off
Show all rules.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso =hdl.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

HDL Coding Standard Customization

Check for duplicate names

Specify whether to check for duplicate names in the design (CGSL-1.A.A.5).

Settings

Default: On

On

Check for duplicate names.

Off

Do not check for duplicate names.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso =hdl.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

HDL Coding Standard Customization

Check for HDL keywords in design names

Specify whether to check for HDL keywords in design names (CGSL-1.A.A.3).

Settings

Default: On

On

Check for HDL keywords in design names.

Off

Do not check for HDL keywords in design names.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdl.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

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 (CGSL-2.C.D.1).

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.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdl.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

HDL Coding Standard Customization

Check module, instance, and entity name length

Specify whether to check module, instance, and entity name length (CGSL-1.A.C.3).

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.



Do not check module, instance, and entity name length.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdl.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

HDL Coding Standard Customization

Check signal, port, and parameter name length

Specify whether to check signal, port, and parameter name length (CGSL-1.A.B.1).

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.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso =hdl.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

HDL Coding Standard Customization

Minimize use of clock enable signals

Specify whether to check for clock enable signals in the generated code (CGSL-2.C.C.4).

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.

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

- HDL Coding Standard Customization
- “Minimize clock enables” on page 10-58

Detect usage of reset signals

Specify whether to check for reset signals in the generated code (CGSL-2.C.C.5).

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.

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

HDL Coding Standard Customization

Minimize use of variables

Specify whether to minimize use of variables (CGSL-2.G).

Settings**Default:** Off

On
Minimize use of variables.

Off
Do not minimize use of variables.

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

HDL Coding Standard Customization

Check if-else statement chain length

Specify whether to check if-else statement chain length (CGSL-2.G.C.1c).

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.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso =hdl.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

HDL Coding Standard Customization

Check if-else statement nesting depth

Specify whether to check if-else statement nesting depth (CGSL-2.G.C.1a).

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.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso =hdl.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

HDL Coding Standard Customization

Check multiplier width

Specify whether to check multiplier bit width (CGSL-2.J.F.5).

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.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdl.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

HDL Coding Standard Customization

Check for non-integer constants

Specify whether to check for non-integer constants (CGSL-3.B.D.1).

Settings

Default: On

On

Check for non-integer constants.

Off

Do not check for non-integer constants.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdl.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

HDL Coding Standard Customization

Check line length

Specify whether to check line lengths in the generated HDL code (CGSL-3.A.D.5).

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.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdl.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

HDL Coding Standard Customization

Highlight feedback loops inhibiting delay balancing and optimizations

Specify whether to generate a script to highlight feedback loops that are inhibiting delay balancing and optimizations.

Settings

Default: Off

On

Generate a MATLAB script that highlights feedback loops in the original model and generated model.

Off

Do not generate a script to highlight feedback loops.

Command-Line Information

Property: HighlightFeedbackLoops

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

- HighlightFeedbackLoops
- HighlightFeedbackLoopsFile
- “Find Feedback Loops”

Highlight blocks inhibiting clock-rate pipelining

Specify whether to generate a script to highlight blocks that are inhibiting clock-rate pipelining.

Settings

Default: Off

On

Generate a MATLAB script that highlights blocks in the original model and generated model that are inhibiting clock-rate pipelining.

Off

Do not generate a script to highlight blocks that are inhibiting clock-rate pipelining.

Command-Line Information

Property: HighlightClockRatePipeliningDiagnostic

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

- HighlightClockRatePipeliningDiagnostic
- HighlightClockRatePipeliningFile

Highlight blocks inhibiting distributed pipelining

Specify whether to generate a script to highlight blocks that are inhibiting distributed pipelining.

Settings

Default: Off

On

Generate a MATLAB script that highlights blocks that are inhibiting distributed pipelining in the original model and generated model.

Off

Do not generate a script to highlight blocks that are inhibiting distributed pipelining.

Command-Line Information

Property: DistributedPipeliningBarriers

Type: string

Value: 'on' | 'off'

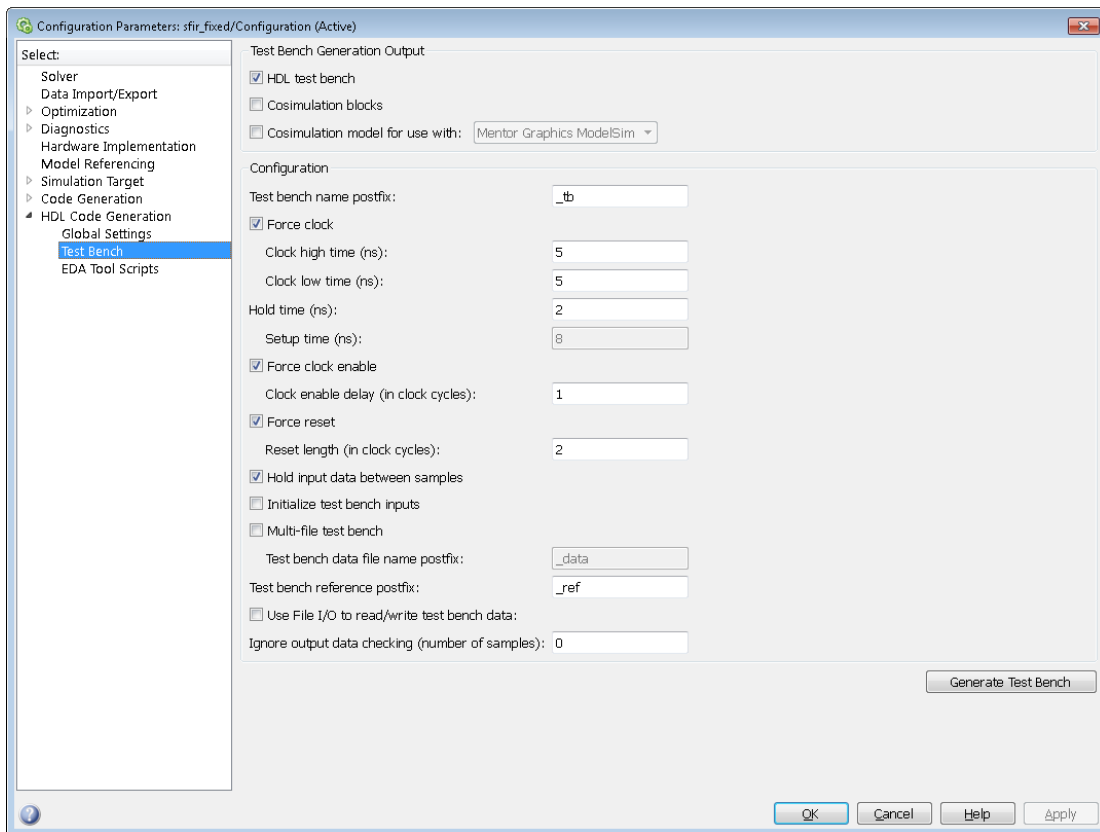
Default: 'off'

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

See Also

- `DistributedPipeliningBarriers`
- `DistributedPipeliningBarriersFile`

HDL Code Generation Pane: Test Bench



In this section...

- “Test Bench Overview” on page 10-106
- “HDL test bench” on page 10-107
- “Cosimulation blocks” on page 10-108
- “Cosimulation model for use with:” on page 10-109
- “Test bench name postfix” on page 10-110
- “Force clock” on page 10-111
- “Clock high time (ns)” on page 10-112

In this section...

“Clock low time (ns)” on page 10-113

“Hold time (ns)” on page 10-114

“Setup time (ns)” on page 10-115

“Force clock enable” on page 10-116

“Clock enable delay (in clock cycles)” on page 10-117

“Force reset” on page 10-118

“Reset length (in clock cycles)” on page 10-119

“Hold input data between samples” on page 10-121

“Initialize test bench inputs” on page 10-122

“Multi-file test bench” on page 10-123

“Test bench reference postfix” on page 10-125

“Test bench data file name postfix” on page 10-126

“Use file I/O to read/write test bench data” on page 10-127

“Ignore output data checking (number of samples)” on page 10-127

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. See also `makehdltb`.

HDL test bench

Enable or disable HDL test bench generation.

Settings

Default: On



On

Enable generation of HDL test bench code that can interface to the DUT.



Off

Suppress generation of HDL test bench code.

Dependencies

This check box enables the options in the **Configuration** section of the **Test Bench** pane.

Command-Line Information

Property: GenerateHDLTestBench

Type: string

Value: 'on' | 'off'

Default: 'on'

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

See Also

“Generating VHDL Test Bench Code”

Cosimulation blocks

Generate a model containing HDL Cosimulation block(s) for use in testing the DUT.

Settings

Default: Off

On

When you select this option, HDL Coder generates and opens a model that contains one or more HDL Cosimulation blocks. The coder generates cosimulation blocks if your installation includes one or more of the following:

- HDL Verifier™ for use with Mentor Graphics ModelSim®
- HDL Verifier for use with Cadence Incisive®

The coder configures the generated HDL Cosimulation blocks to conform to the port and data type interface of the DUT selected for code generation. By connecting an HDL Cosimulation block to your model in place of the DUT, you can cosimulate your design with the desired simulator.

Off

Do not generate HDL Cosimulation blocks.

Dependencies

This check box enables the other options in the **Configuration** section of the **Test Bench** pane.

Command-Line Information

Property: GenerateCoSimBlock

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

GenerateCoSimBlock

Cosimulation model for use with:

Specify the HDL cosimulator for use with the generated HDL Cosimulation block and model

Settings

Default: Mentor Graphics ModelSim

Select one of the following options from the dropdown menu:

- **Mentor Graphics ModelSim:** This option is the default. HDL Coder generates and opens a Simulink model that contains an HDL Cosimulation block specifically for use with Mentor Graphics ModelSim.
- **Cadence Incisive:** The coder generates and opens a Simulink model that contains an HDL Cosimulation block specifically for use with Cadence Incisive.

Command-Line Information

Property: GenerateCosimModel

Type: string

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

`GenerateCoSimModel`

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`.

Command-Line Information

Property: `TestBenchPostFix`

Type: `string`

Default: `'_tb'`

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

See Also

`TestBenchPostFix`

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.

Command-Line Information

Property: ForceClock

Type: string

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

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.

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.

Command-Line Information

Because this is a display-only field, a corresponding command-line property does not exist.

See Also

HoldTime

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.

Command-Line Information

Property: ForceClockEnable

Type: string

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

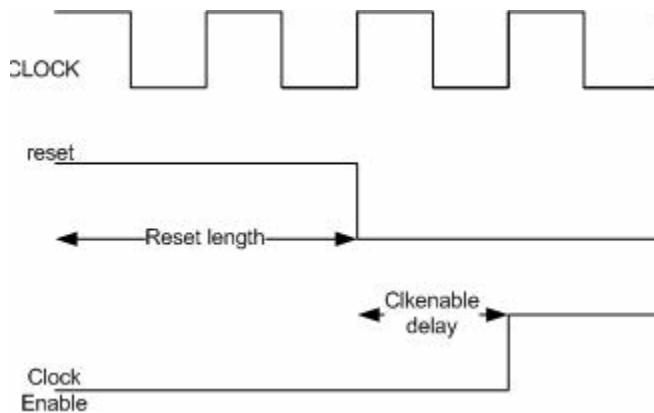
Clock enable delay (in clock cycles)

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.

Command-Line Information

Property: ForceReset

Type: string

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

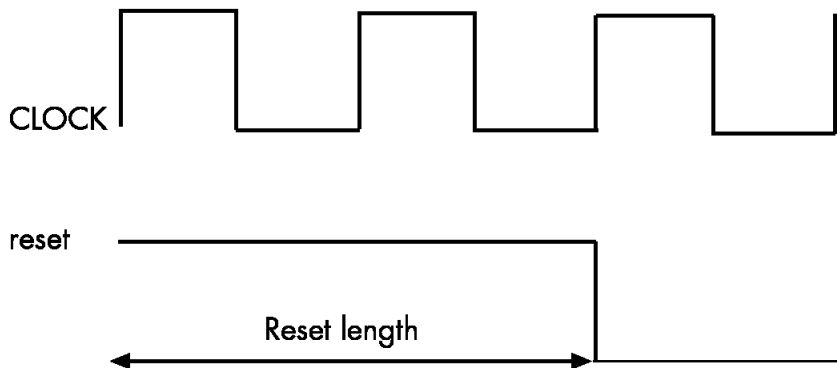
Reset length (in clock cycles)

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`.

See Also

ResetLength

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.

Command-Line Information

Property: HoldInputDataBetweenSamples

Type: string

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).

Command-Line Information

Property: InitializeTestBenchInputs

Type: string

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

Multi-file test bench

Divide generated test bench into helper functions, data, and HDL test bench code files.

Settings

Default: Off

On

Write separate files for test bench code, helper functions, and test bench data. 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 follows:

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

Off

Write a single test bench file containing the HDL test bench code, helper functions, and test bench data.

Dependency

When this property is selected, **Test bench data file name postfix** is enabled.

Command-Line Information

Property: MultifileTestBench

Type: string

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`

Test bench reference postfix

Specify a string 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 string specified by **Test bench reference postfix** is appended to the generated signal names.

Command-Line Information

Parameter: TestBenchReferencePostFix

Type: string

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** string 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: string

Default: '_data'

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

See Also

TestBenchDataPostFix

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: Off



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.

Command-Line Information

Property: UseFileIOInTestBench

Type: string

Value: 'on' | 'off'

Default: 'off'

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

See Also

UseFileIOInTestBench

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”)
- 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
 - dspsigops/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

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`

HDL Code Generation Pane: EDA Tool Scripts

The screenshot shows the 'HDL Code Generation Pane: EDA Tool Scripts' dialog box. On the left is a tree view under 'Select:' with 'EDA Tool Scripts' selected. The main area has a checked checkbox 'Generate EDA scripts' and a list of script types: 'Compilation script', 'Simulation script', 'Synthesis script', and 'Lint script'. Below this is a checkbox for 'Generate multicyle path information'. On the right, there are four text input fields: 'Compile file postfix:' (with '_compile.do' entered), 'Compile initialization:' (with 'vlib %s\n' entered), 'Compile command for VHDL:' (with 'vcom %s %s\n' entered), and 'Compile command for Verilog:' (with 'vlog %s %s\n' entered). A 'Compile termination:' field is empty.

In this section...

- “EDA Tool Scripts Overview” on page 10-132
- “Generate EDA scripts” on page 10-133
- “Generate multicyle path information” on page 10-134
- “Compile file postfix” on page 10-135
- “Compile initialization” on page 10-136
- “Compile command for VHDL” on page 10-137
- “Compile command for Verilog” on page 10-138
- “Compile termination” on page 10-139
- “Simulation file postfix” on page 10-140
- “Simulation initialization” on page 10-141
- “Simulation command” on page 10-142
- “Simulation waveform viewing command” on page 10-143

In this section...

“Simulation termination” on page 10-144

“Choose synthesis tool” on page 10-145

“Synthesis file postfix” on page 10-148

“Synthesis initialization” on page 10-149

“Synthesis command” on page 10-150

“Synthesis termination” on page 10-151

“Choose HDL lint tool” on page 10-151

“Lint initialization” on page 10-152

“Lint command” on page 10-153

“Lint termination” on page 10-153

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: string

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”
- EDAScriptGeneration

Generate multicycle path information

Generate a file that reports multicycle path constraint information.

Settings

Default: Off

On

Generate a text file that reports multicycle path constraint information, for use with synthesis tools.

Off

Do not generate a multicycle path information file.

Command-Line Information

Parameter: MulticyclePathInfo

Type: string

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 Multicycle Path Information Files”
- `MulticyclePathInfo`

Compile file postfix

Specify a postfix string appended 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: `string`

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”
- `HDLCompileFilePostfix`

Compile initialization

Specify a format string 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: `string`

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”
- `HDLCompileInit`

Compile command for VHDL

Specify a format string 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: `string`

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”
- `HDLCompileVHDLCmd`

Compile command for Verilog

Specify a format string 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: `string`

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”
- `HDLCompileVerilogCmd`

Compile termination

Specify a format string passed to `fprintf` to write the termination portion of the compilation script.

Settings

Default: empty string

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: string

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”
- `HDLCompileTerm`

Simulation file postfix

Specify a postfix string appended 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: `string`

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”
- `HDLSimFilePostfix`

Simulation initialization

Specify a format string passed to `fprintf` to write the initialization section of the simulation script.

Settings

Default: The default string 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: `string`

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”
- `HDLsimInit`

Simulation command

Specify a format string 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 the `VHDLLibraryName` 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.

Command-Line Information

Property: `HDLSimCmd`

Type: `string`

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”
- `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: string

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”
- HDLSimViewWaveCmd

Simulation termination

Specify a format string 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: `string`

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”
- `HDLSimTerm`

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: string

Value: 'None' | 'ISE' | 'Libero' | '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 string appended 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`.

Command-Line Information

Property: HDLSynthFilePostfix

Type: string

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”
- HDLSynthFilePostfix

Synthesis initialization

Specify a format string 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 string is a format string passed to `fprintf` to write the `Init` section of the synthesis script. The default string is a synthesis project creation command. The implicit argument, `%s`, is the top-level module or entity name.

Command-Line Information

Property: HDLSynthInit

Type: string

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”
- HDLSynthInit

Synthesis command

Specify a format string 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 string 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.

Command-Line Information

Property: HDLSynthCmd

Type: string

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”
- HDLSynthCmd

Synthesis termination

Specify a format string 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 string is a format string passed to `fprintf` to write the **Term** section of the synthesis script. The termination string does not take arguments.

Command-Line Information

Property: HDLSynthTerm

Type: string

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”
- HDLSynthTerm

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: string

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”
- HDLLintTool

Lint initialization

Enter an initialization string for your HDL lint script.

Command-Line Information

Property: HDLLintInit

Type: string

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”
- HDLLintInit

Lint command

Enter the command for your HDL lint script.

Command-Line Information

Property: HDLLintCmd

Type: string

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”
- HDLLintCmd

Lint termination

Enter a termination string for your HDL lint script.

Command-Line Information

Property: HDLLintTerm

Type: string

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”
- HDLLintTerm

Supported Blocks Library and Block Properties

- “Generate a Supported Blocks Report” on page 11-2
- “Generate a Library of Supported Blocks” on page 11-3
- “HDL Block Properties” on page 11-4
- “HDL Filter Block Properties” on page 11-29
- “Configuring HDL Filter Architectures” on page 11-36
- “Distributed Arithmetic for HDL Filters” on page 11-38
- “Set and View HDL Block Parameters” on page 11-41
- “Set HDL Block Parameters for Multiple Blocks” on page 11-44
- “View HDL Model Parameters” on page 11-46
- “Pass through, No HDL, and Cascade Implementations” on page 11-47
- “Test Bench Block Restrictions” on page 11-48
- “Build a ROM Block with Simulink Blocks” on page 11-49

Generate a Supported Blocks Report

To generate an HTML table that summarizes blocks supported for HDL Code generation:

- 1 Enter the following at the MATLAB command line:

```
hdl1lib('html');
```

After `hdl1lib` creates the `hdl1supported` library, you see the following:

```
### HDL Supported Block List hdlblklist.html
```

- 2 Click the `hdlblklist.html` link to see the generated block list.

See also “Create a Supported Blocks Library”.

Generate a Library of Supported Blocks

You can automatically generate a library of blocks supported for HDL code generation, `hdlsupported`. You can then create your code generation subsystem using blocks from the library.

To learn how to generate the library, see `hdl1lib`.

To learn how to set HDL block implementations and parameters, see “Set and View HDL Block Parameters”.

View HDL-Specific Block Documentation

To view HDL-specific documentation for each block in `hdlsupported`:

- 1 Right-click the block and select **HDL Code > HDL Block Properties**.
- 2 To view the block documentation, click **Help**.

You can also view HDL-specific block documentation in “Supported Blocks”.

HDL Block Properties

In this section...

“Overview” on page 11-4
“BalanceDelays” on page 11-5
“ConstMultiplierOptimization” on page 11-6
“ConstrainedOutputPipeline” on page 11-7
“DistributedPipelining” on page 11-8
“DSPStyle” on page 11-9
“FlattenHierarchy” on page 11-12
“InputPipeline” on page 11-13
“InstantiateFunctions” on page 11-13
“InstantiateStages” on page 11-15
“LoopOptimization” on page 11-15
“LUTRegisterResetType” on page 11-16
“MapPersistentVarsToRAM” on page 11-16
“OutputPipeline” on page 11-18
“ResetType” on page 11-19
“SerialPartition” on page 11-21
“SharingFactor” on page 11-21
“SoftReset” on page 11-21
“StreamingFactor” on page 11-23
“UseMatrixTypesInHDL” on page 11-23
“UseRAM” on page 11-24
“VariablesToPipeline” on page 11-28

Overview

Block implementation parameters enable you to control details of the code generated for specific block implementations. See “Set and View HDL Block Parameters” to learn how to select block implementations and parameters in the GUI or the command line.

Property names are strings. 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.

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 `BalanceDelays`.

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`.

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”

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, <i>nStages</i>	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)”
 - `IgnoreDataChecking`
-

See Also

- “Distributed Pipelining and Hierarchical Distributed Pipelining”
- “Specify Distributed Pipelining”
- “Distributed Pipeline Insertion for MATLAB Function Blocks”

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, Model Variants 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 use_dsp48 : string ; attribute multstyle of m4_out1 : signal is "dsp" ;	attribute use_dsp48 : string ; attribute use_dsp48 of m4_out1 : signal is "yes" ;

DSPStyle Value	TargetLanguage Value	SynthesisTool Value	
		'Altera Quartus II'	'Xilinx ISE' 'Xilinx Vivado'
'off'	'Verilog'	(* multstyle = "logic" *) wire signed [32:0] m4_out1;	(* use_dsp48 = "no" *) wire signed [32:0] m4_out1;
	'VHDL'	attribute use_dsp48 : 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.

Prerequisites For Hierarchy Flattening

To flatten hierarchy, a subsystem must have the following block properties.

Property	Required value
DistributedPipelining	'off'
StreamingFactor	0
SharingFactor	0

To flatten hierarchy, you must also have the `MaskParameterAsGeneric` global property set to 'off'. For more information, see `MaskParameterAsGeneric`.

How To Flatten Hierarchy

To set hierarchy flattening using the HDL Block Properties dialog box:

- 1 Right-click the subsystem.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 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:

- Atomic and instantiated in the design more than once.
- A black box implementation or model reference.
- A triggered subsystem when `TriggerAsClock` is enabled
- A masked subsystem.

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(gcb, 'BlockType', 'Sum');
for ii=1:length(sblocks),hdlset_param(sblocks{ii},'InputPipeline', 2), end;
```

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 string in the `makehdl` property `PipelinePostfix`. See `PipelinePostfix` for an example.

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.

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_m1fn`:

```
hdlset_param('my_m1fn', 'LoopOptimization', 'Streaming')
```

See also `hdlset_param`.

Limitations for MATLAB Function Block Loop Optimization

HDL Coder cannot stream a loop if:

- The loop index counts down. The loop index must increase by 1 on each iteration.
- There are 2 or more nested loops at the same level of hierarchy within another loop.
- Any particular persistent variable is updated both inside and outside a loop.

HDL Coder can stream a loop when the persistent variable is:

- Updated inside the loop and read outside the loop.
- Read within the loop and updated outside the loop.

LUTRegisterResetType

Use the `LUTRegisterResetType` block parameter to control synthesis of a LUT into a ROM structure on an FPGA.

LUTRegisterResetType Value	Description
default	LUT output register has default reset logic. When you generate HDL, the LUT will be synthesized as registers.
none	LUT output register has no reset logic. When you generate HDL, the LUT will be synthesized as a ROM.

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 11-17.

RAM Mapping Restrictions

When you enable RAM mapping, a persistent array or user-defined System object private property maps to a block RAM when all of 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 “HDL Code Generation Pane: Global Settings”.

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”.

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(gcb, 'BlockType', 'Sum');
for ii=1:length(sblocks),hdlset_param(sblocks{ii},'OutputPipeline', 2), end;
```

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`. See `PipelinePostfix` for an example.

See also “Distributed Pipeline Insertion for MATLAB Function Blocks”.

ResetType

Use the `ResetType` block parameter to suppress reset logic generation.

ResetType Value	Description
default	Generate reset logic.
none	Do not generate reset logic. Reset is not applied to generated registers. Therefore, mismatches between Simulink and the generated code occur for some number of samples during the initial phase, when registers are not fully loaded. To avoid test bench errors during the initial phase, determine the number of samples required to fully load the registers. Then, set the Ignore output data checking (number of samples) option accordingly. See also <code>IgnoreDataChecking</code> .

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 **ResetType**.

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”.

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”.

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

SoftReset Value	Description
	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
    end
  end
end

```

```

        end
        else begin
            Unit_Delay_Resettable_reg <= In1;
        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”.

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.
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 <code>struct</code> data types. You cannot use linear indexing to specify matrix elements. For example, if you have a 3x3 matrix, <code>A</code>, you cannot use <code>A(4)</code>. Instead, use <code>A(2,1)</code>. <p>You can also use a colon operator in either the row or column subscript, but not both. For example, you</p>

UseMatrixTypesInHDL Setting	Description
	<p>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"> • <code>InstantiateFunctions</code> must be set to 'off'. If you enable <code>InstantiateFunctions</code>, <code>UseMatrixTypesInHDL</code> has no effect.

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')
```

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	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 11-27.
on	<p>The delay maps to a dual-port RAM block when all of the following conditions are true:</p> <ul style="list-style-type: none"> • Initial value of the delay is zero. • Delay length > 4. • Delay has one of the following set of numeric and data type attributes:

UseRAM Setting	Mapping Behavior
	<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. • 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 11-25.</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

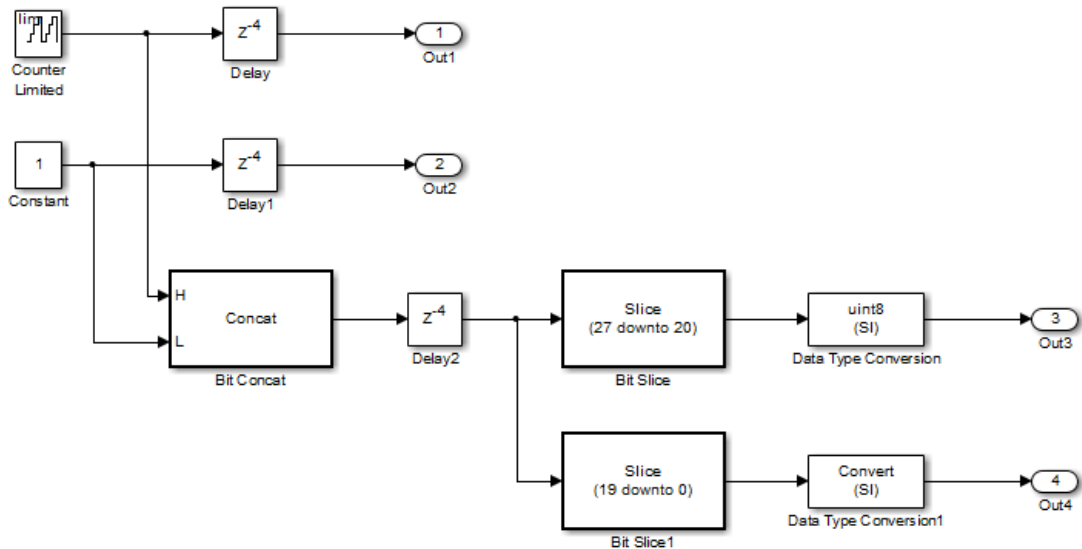
The following rules control whether or not multiple delays can merge into one delay:

- 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 DelayLength * WordLength * VectorLength * 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

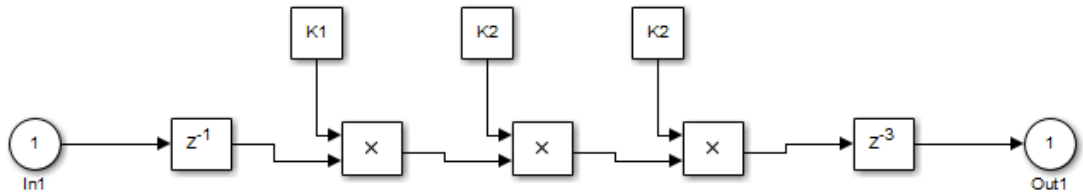
When **UseRAM** is **off** for a Delay block, HDL Coder maps the delay to a shift register by default. However, the coder changes the **UseRAM** setting to **on** and tries to map the delay to a RAM under the following conditions:

- 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
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 string, with spaces separating the variables.

See also “Pipeline MATLAB Expressions”.

HDL Filter Block Properties

In this section...

“AddPipelineRegisters” on page 11-29
“ChannelSharing” on page 11-29
“CoeffMultipliers” on page 11-30
“DALUTPartition” on page 11-30
“DARadix ” on page 11-32
“FoldingFactor” on page 11-33
“MultiplierInputPipeline” on page 11-33
“MultiplierOutputPipeline” on page 11-33
“NumMultipliers” on page 11-34
“ReuseAccum” on page 11-34
“SerialPartition” on page 11-34

AddPipelineRegisters

You can use this parameter to insert a pipeline register between stages of computation in a filter. The default value is `off`.

Take note of the following limitations when applying `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. However, equivalent delays are not introduced on other, parallel signal paths. To balance delays, use `OutputPipeline` on parallel data paths.

ChannelSharing

You can use the `ChannelSharing` implementation parameter with a multi-channel 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.

An example of how to use `ChannelSharing` is in DSP System Toolbox Examples under "Generate HDL Code for Multichannel FIR Filter".

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.

For more information on Distributed Arithmetic architectures, see “Distributed Arithmetic for HDL Filters”

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
<code>dfilt.dffir</code>	<code>FL = length(find(Hd.numerator~= 0))</code>
<code>dfilt.dfsymfir</code> <code>dfilt.dfasymfir</code>	<code>FL = ceil(length(find(Hd.numerator~= 0))/2)</code>

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 (`mfilt.firdecim` and `mfilt.firinterp`), 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 a fine control for partitioning each subfilter.

The following table shows the FL calculations for each type of LUT partition.

LUT Partition Specified As...	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 = size(polyphase(Hm), 2)
<i>Matrix</i> : determine the subfilter length FL _{<i>i</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>i</i>} .	<pre>p = polyphase(Hm); FLi = length(find(p(i,:)));</pre> <p>where <i>i</i> is the index to the <i>i</i>th row of the polyphase matrix of the multirate filter. The <i>i</i>th row of the matrix <i>p</i> represents the <i>i</i>th 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 (`dfilt.dfsymfir`) and asymmetrical (`dfilt.dfasymfir`) filters, see “Considerations for Symmetrical and Asymmetrical Filters” on page 11-39.

For more information on Distributed Arithmetic architectures, see “Distributed Arithmetic for HDL Filters”

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 11-34 property. You must select one property or the other; you may not use both. If neither `FoldingFactor` or `NumMultipliers` is specified, 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.

Take note of the following limitation when applying `MultiplierInputPipeline`:

- Pipeline stages introduce delays along the path in the model that contains the affected filter. However, equivalent delays are not introduced on other, parallel signal paths. To balance delays, use `OutputPipeline` on parallel data paths.

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.

Take note of the following limitation when applying `MultiplierOutputPipeline`:

- Pipeline stages introduce delays along the path in the model that contains the affected filter. However, equivalent delays are not introduced on other, parallel signal paths. To balance delays, use `OutputPipeline` on parallel data paths.

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 11-33 property. You must select one property or the other; you may not use both. If neither FoldingFactor or NumMultipliers is specified, 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 “Configuring HDL Filter Architectures”

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 of length equal 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 can be equal. For example, for a filter length of 8, 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.</p>
Cascade-serial with automatically optimized partitioning	Omit this property.

For more information on parallel and serial filter architectures, see “Configuring HDL Filter Architectures”.

This property is also used for Min/Max blocks with cascade-serial architectures. For how to configure Min/Max cascades, see “SerialPartition”.

Configuring 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.

You configure a serial architecture using the “SerialPartition” and “ReuseAccum” parameters.

Fully Parallel Architecture

This is the default option. 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.

Serial Architectures

Serial architectures reuse hardware resources in time, saving chip area. The available serial architecture options are:

- *Fully serial*: A fully serial architecture conserves area by reusing multiplier and adder resources sequentially. For example, a four-tap filter design would use 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 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 will be 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. For example, you could specify a four-tap filter with two partitions, each having two taps. The system clock would run 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 non-cascade partly serial architecture.

To generate a cascade-serial architecture, you 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 handle latency, HDL Coder inserts a Delay block into the generated model after the filter block.

Use Full Precision Filter Settings

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.

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 11-30
- Improve Performance with Parallelism: “DARadix ” on page 11-32

For information on the theoretical foundations of DA, see “Further References” on page 11-39.

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 (`dfilt.dfsymfir`) and asymmetrical (`dfilt.dfasymfir`) 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 Block Parameters

In this section...

“Set HDL Block Parameters from the GUI” on page 11-41

“Set HDL Block Parameters from the Command Line” on page 11-41

“View All HDL Block Parameters” on page 11-42

“View Non-Default HDL Block Parameters” on page 11-42

For a list of HDL block properties, see “HDL Block Properties”.

Set HDL Block Parameters from the GUI

You can view and set HDL-related block properties, such as implementation and implementation parameters, at the individual block level. To open the HDL Properties dialog box:

- 1 Right-click the block and select **HDL Code > HDL Block Properties**.
The HDL Properties dialog box opens.
- 2 Modify the block properties as desired.
- 3 Click OK.

Set HDL Block Parameters from the Command Line

`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 architecture for the same block, enter the following at the command line:

```
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]
```

See also `hdlset_param` and `hdlget_param`.

View All HDL Block Parameters

`hdldispblkparams` displays the HDL block parameters available for a specified block.

The following example displays HDL block parameters and values for the currently selected block.

```
hdldispblkparams(gcb, 'all')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL Block Parameters ('simplevectorsum/vsum/Sum of
Elements')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Implementation
Architecture : Linear
Implementation Parameters
InputPipeline : 0
OutputPipeline : 0
```

See also `hdldispblkparams`.

View Non-Default HDL Block Parameters

The following example displays only HDL block parameters that have non-default values for the currently selected block.

```
hdldispblkparams(gcb)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL Block Parameters ('simplevectorsum/vsum/Sum of
Elements')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Implementation

Architecture : Linear

Implementation Parameters

OutputPipeline : 3

See also `hdlispblkparams`.

Set HDL Block Parameters for Multiple Blocks

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.

See the Simulink documentation for detailed information about `find_system`.

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.

- 1 Open the `sfir_fixed` model.
- 2 Click on the `sfir_fixed/symmetric_fir` subsystem to select it.
- 3 Locate all Product blocks within the subsystem as follows:

```
prodblocks = find_system(gcf, 'BlockType', 'Product')  
  
prodblocks =  
  
    'sfir_fixed/symmetric_fir/Product'  
    'sfir_fixed/symmetric_fir/Product1'  
    'sfir_fixed/symmetric_fir/Product2'  
    'sfir_fixed/symmetric_fir/Product3'
```

- 4 Set the output pipeline depth to 2 for all selected blocks.

```
for ii=1:length(prodblocks), hdlset_param(prodblocks{ii}, 'OutputPipeline', 2), end;
```

- 5 To verify the settings, display the value of the `OutputPipeline` parameter for the blocks .

```
for ii=1:length(prodblocks), hdlget_param(prodblocks{ii}, 'OutputPipeline'), end;  
  
ans =  
  
    2  
  
ans =  
  
    2  
  
ans =  
  
    2  
  
ans =
```


2

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'
```

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.

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:

- SimRF™
- SimDriveline™
- SimEvents®
- SimMechanics™
- SimPowerSystems™
- Simscape™

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
```


Generating HDL Code for Multirate Models

- “Code Generation from Multirate Models” on page 12-2
- “Timing Controller for Multirate Models” on page 12-5
- “Generate Reset for Timing Controller” on page 12-6
- “Multirate Model Requirements for HDL Code Generation” on page 12-7
- “Generate a Global Oversampling Clock” on page 12-10
- “Use Trigger As Clock in Triggered Subsystems” on page 12-16
- “Generate Multicycle Path Information Files” on page 12-18
- “Using Multiple Clocks in HDL Coder™” on page 12-27

Code Generation from Multirate Models

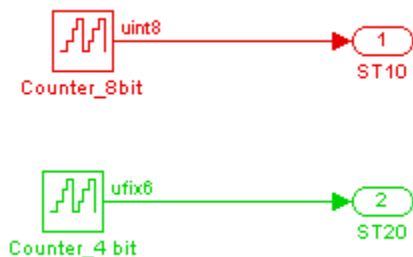
HDL Coder supports HDL code generation for single-clock and multiple clock multirate models. Your model can include blocks running at multiple sample rates:

- 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.

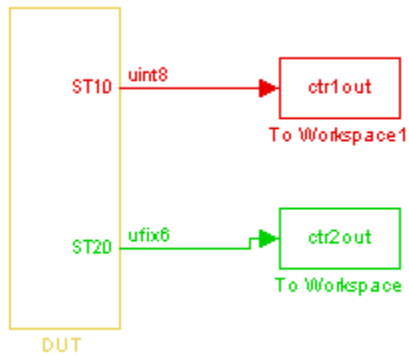
In general, generating HDL code for a multirate model does not differ greatly from generating HDL code for a single-rate model. However, there are a few requirements and restrictions on the configuration of the model and the use of specialized blocks (such as Rate Transitions) that apply to multirate models. For details, see “Multirate Model Requirements for HDL Code Generation” on page 12-7.

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.

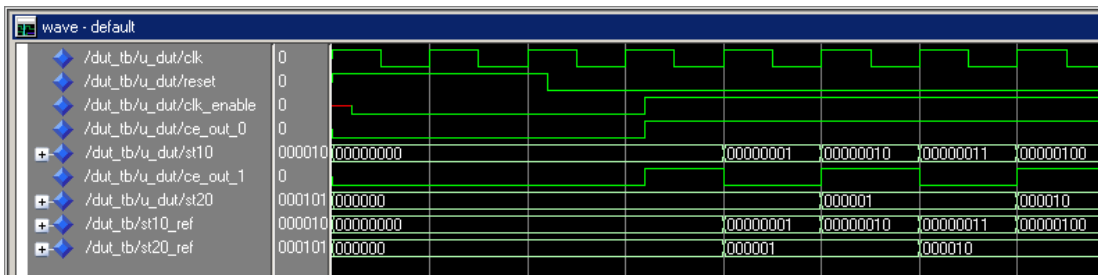
```

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™”.

Generate Reset for Timing Controller

In this section...

“Requirements for Timing Controller Reset Port Generation” on page 12-6

“How To Generate Reset for Timing Controller” on page 12-6

“Limitations for Timing Controller Reset Port Generation” on page 12-6

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...

“Configuring Model Parameters” on page 12-7

“Sample Rate Requirements” on page 12-7

“Block Configuration and Restrictions For Multirate DUTs” on page 12-8

Configuring Model 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 Requirements

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.

Block Configuration and Restrictions For Multirate DUTs

- “Subsystem with Black Box Interface” on page 12-8
- “Rate Transition” on page 12-8
- “Upsample” on page 12-8
- “Downsample” on page 12-8
- “Delay and Zero-Order Hold” on page 12-9

Subsystem with Black Box Interface

HDL code generation is not supported for multirate DUTs that contain a subsystem with a black box interface.

Rate Transition

Rate Transition blocks must be explicitly inserted into the signal path when blocks running at different rates are connected. For general information about the Rate Transition block, see the Rate Transition block documentation.

Make sure the data transfer properties for Rate Transition blocks are set as follows:

- **Ensure deterministic data transfer:** Selected.
- **Ensure data integrity during data transfer:** Selected.

Upsample

When configuring Upsample blocks, set **Frame based mode** to **Maintain input frame size**.

When the Upsample block is in this mode, **Initial conditions** has no effect on generated code.

Downsample

Configure Downsample blocks as follows:

- Set **Frame based mode** to **Maintain input frame size**.
- Set **Sample based mode** to **Allow multirate**.

Given these Downsample block settings, **Initial conditions** has no effect on generated code if **Sample offset** is set to 0.

Delay and Zero-Order Hold

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 12-10

“Requirements for the Oversampling Factor” on page 12-10

“Specifying the Oversampling Factor From the GUI” on page 12-11

“Specifying the Oversampling Factor From the Command Line” on page 12-12

“Resolving Oversampling Rate Conflicts” on page 12-12

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:

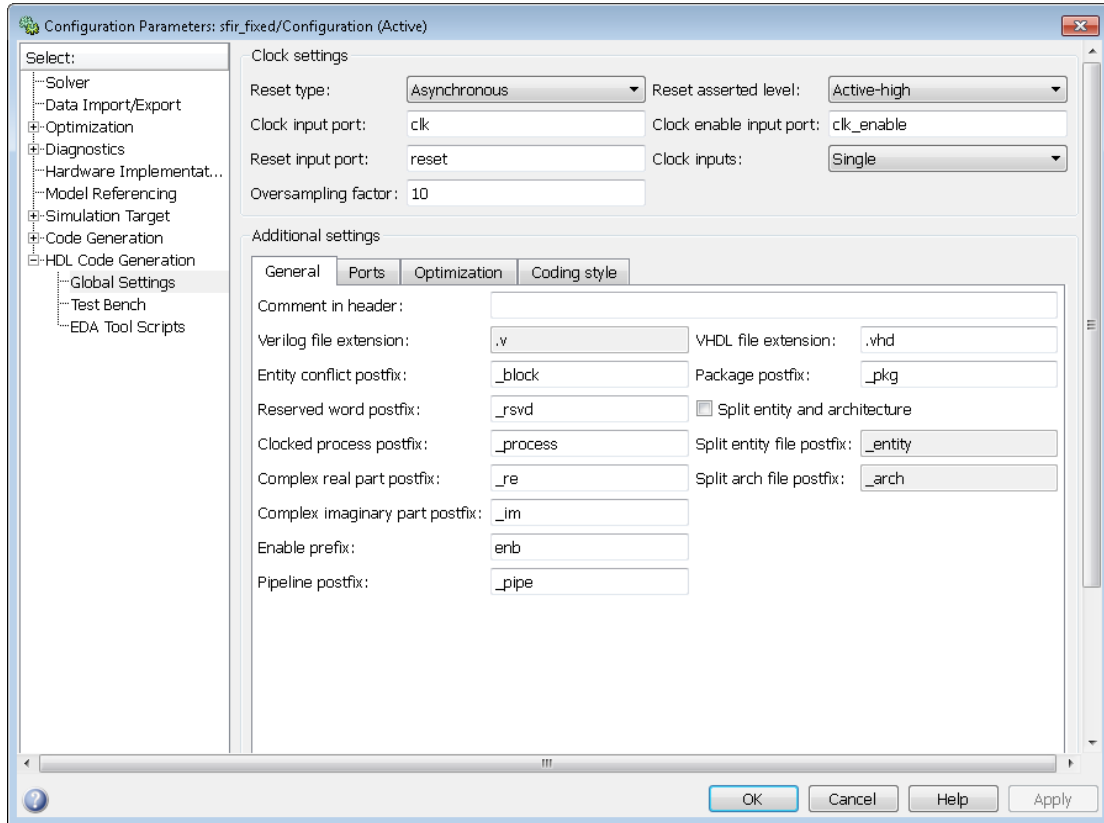
- The oversampling factor must be an integer greater than or equal to 1.
- The default value is 1. In the default case, HDL Coder does not generate a global oversampling clock.
- Some DUTs require multiple sampling rates for their internal operations. In such cases, the other rates in the DUT must divide evenly into the global oversampling

rate. For more information, see “Resolving Oversampling Rate Conflicts” on page 12-12 .

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:

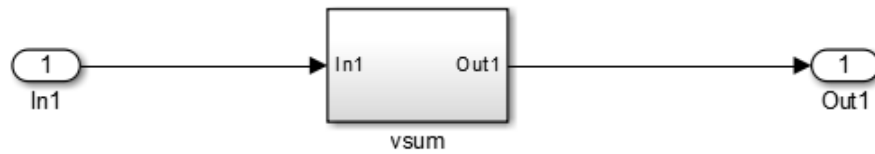
```
>> makehdl(gcb,'Oversampling', 7)
### Generating HDL for 'sfir_fixed/symmetric_fir'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### 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.
```

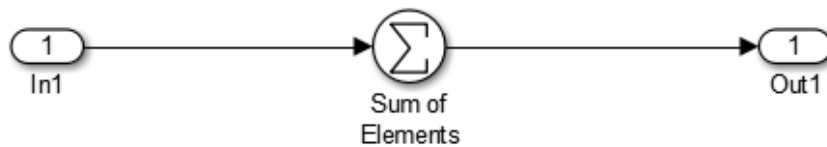
Resolving Oversampling Rate Conflicts

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.
...

```

Use Trigger As Clock in Triggered Subsystems

In this section...

“When To Use Trigger As Clock” on page 12-16

“Requirements For Using Trigger As Clock” on page 12-16

“How To Specify Trigger As Clock” on page 12-16

“Limitations When Using Trigger As Clock” on page 12-17

When To Use 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.

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.

Requirements For Using Trigger As Clock

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.

How To Specify Trigger As Clock

Using the Configuration Parameters Dialog Box

In **HDL Code Generation > Global Settings > Optimization** tab, select **Use trigger signal as clock**.

Using the HDL Workflow Advisor

In the **HDL Code Generation > Set Code Generation Options > Set Advanced Options > Optimization** tab, select **Use trigger signal as clock**.

At the Command Line

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 When Using Trigger As Clock

Using the trigger as clock for triggered subsystems can result in timing mismatches of one cycle during testbench simulation.

Generate Multicycle Path Information Files

In this section...

“Overview” on page 12-18

“Format and Content of a Multicycle Path Information File” on page 12-19

“File Naming and Location Conventions” on page 12-24

“Generating Multicycle Path Information Files Using the GUI” on page 12-24

“Generating Multicycle Path Information Files Using the Command Line” on page 12-24

“Limitations” on page 12-25

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 12-2). 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.booliereg; TO : Sbs.booloreg; PATH_MULT : 2; RELATIVE_CLK : source,
Sbs.clk;
FROM : Sbs.booliereg_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 12-21 .
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 12-21.

Keyword : field (or subfields)	Field Description
PATH_MULT : <i>N</i> ;	<p>The <i>path multiplier</i> 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 <i>relative clock</i> 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 source. 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 source or destination. 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. sysclock provides the path to the system's top-level clock (e.g., Sbs.clk) 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 : Sbs.intireg(7:0); TO : Sbs.intoreg(7:0);	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 H1, and H1 is inside the module Sbs. <code>u_H1</code> is the instance name of H1 in Sbs. 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

By default, HDL Coder does not generate multicycle path information files. To enable generation of multicycle path information files, select **Generate multicycle path information** in the **HDL Code Generation > EDA Tool Scripts** pane of the Configuration Parameters dialog box.

When you select **Generate multicycle path information**, the coder generates a multicycle path information file each time you initiate code generation.

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 ashdlsrc\Subsystem_tc.vhd
### Working on hdlfirtdecim_multicycle/Subsystem ashdlsrc\Subsystem.vhd
### Generating package filehdlsrc\Subsystem_pkg.vhd
### Finishing multicycle path connectivity analysis.
### Writing multicycle path information inhdlsrc\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.

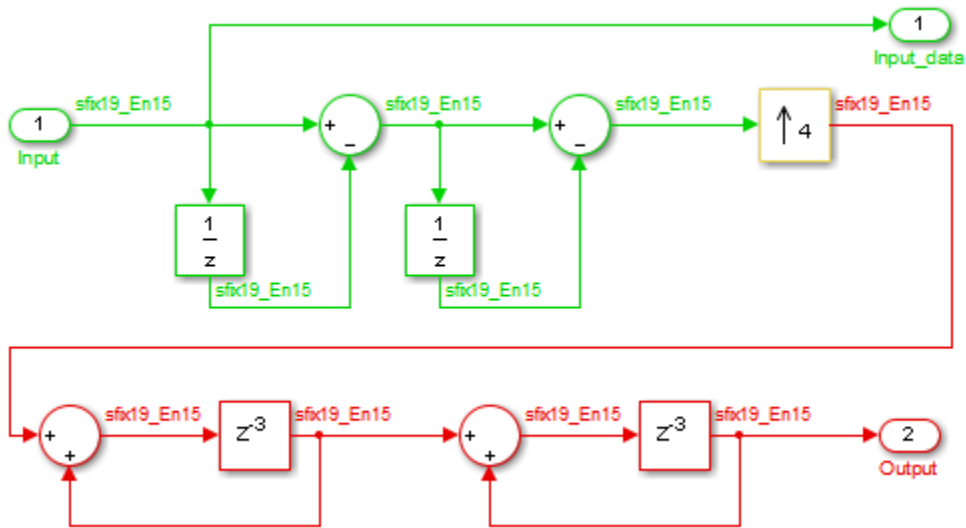
Overview of Clocking Modes

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'.
### 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 in C:\TEMP\BR2015ad_189898_5888\tpb
### Creating HDL Code Generation Check Report file:///C:\TEMP\BR2015ad_189898_5888\tpb
### 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'.
### 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 in C:\TEMP\BR2015ad_189898_5888\tpb
### Creating HDL Code Generation Check Report file:///C:\TEMP\BR2015ad_189898_5888\tpb
### 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

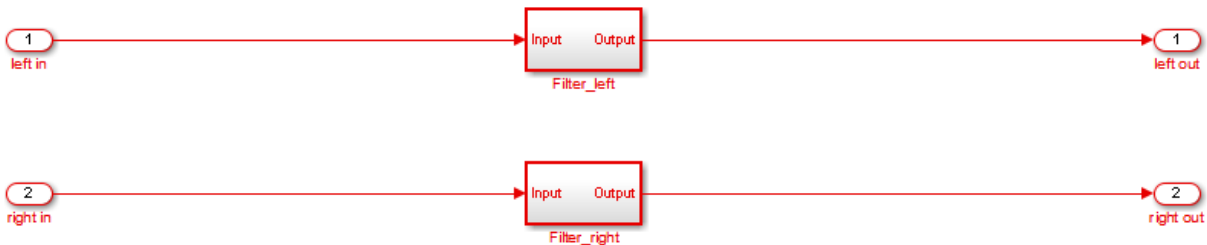
Multiple synchronous clocks can be useful even for a design with only a single Simulink rate. Various optimizations can require clock rates faster than indicated in the original model. The following example demonstrates an audio filtering model that applies the same filter on the left and right channels. 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'.
### 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 in C:\TEMP\BR2015ad_189898_5888\tp
### Creating HDL Code Generation Check Report file:///C:\TEMP\BR2015ad_189898_5888\tpb
### 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'.
### Starting HDL check.

```

```
### The DUT requires an initial pipeline setup latency. Each output port experiences t
### 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 in C:\TEMP\BR2015ad_189898_5888\tp
### Creating HDL Code Generation Check Report file:///C:\TEMP\BR2015ad_189898_5888\tpb
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 1 r
### HDL code generation complete.
```



Generating Bit-True Cycle-Accurate Models

- “Generated Model and Validation Model” on page 13-2
- “Locate Numeric Differences After Speed Optimization” on page 13-5

Generated Model and Validation Model

In this section...

“Generated Model” on page 13-2

“Validation Model” on page 13-3

Generated Model

Before generating code, HDL Coder creates a behavioral model of the HDL code, called the *generated model*. The generated model uses HDL-specific block implementations, and it implements the area and speed optimizations that you specify in your Simulink model.

The generated model is an intermediate 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.

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`.

Highlight Color	Delay Type
Cyan	Block implementation
	RAM mapping
Green	Constrained output pipelining
Orange	Distributed pipelining
	Input and output pipelining
	Delay balancing
	Clock-rate pipelining

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, so 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”.

Customize the Generated Model

To customize the generated model, use the following properties with `makehdl` or `hdlset_param`:

- `GeneratedModelName`
- `GeneratedModelNamePrefix`
- `CodeGenerationOutput`

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 with 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, with compensating delays inserted.
- Original inputs, routed to both the original model and generated model.

- Scopes for comparing and viewing the outputs of the original model and generated model.

Generate A Validation Model

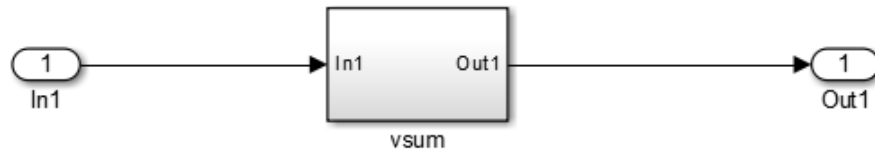
To generate a validation model:

- In the Configuration Parameters dialog box, in the **HDL Code Generation** pane, enable **Generate 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`.

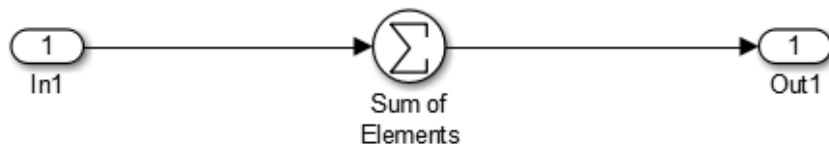
Locate Numeric Differences After Speed Optimization

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_tree`, 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.

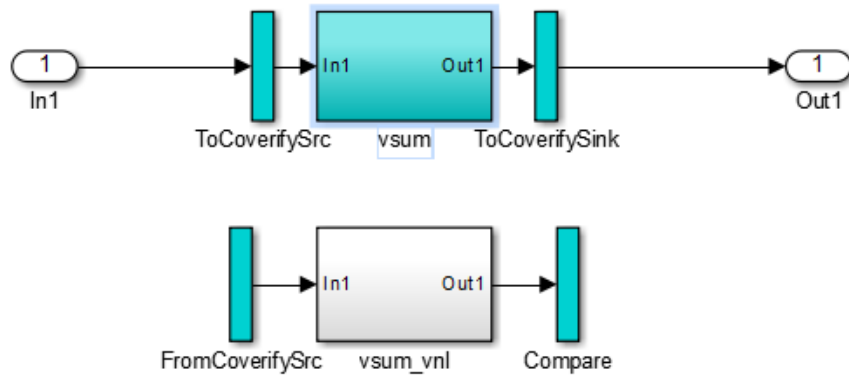


The model is configured to use the **Tree** implementation when generating HDL code for the Sum block within the `vsum` subsystem. This implementation, optimized for minimal latency, generates a tree-shaped structure of adders for the Sum block.

To select a nondefault implementation for an individual block:

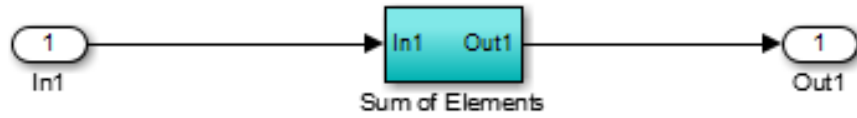
- 1 Right-click the block and select **HDL Code > HDL Block Properties** .
- 2 In the HDL Properties dialog box, select the desired implementation from the **Architecture** menu.
- 3 Click **Apply** and close the dialog box.

After code generation, you can view the validation model, `gm_simplevectorsum_tree_vnl`.

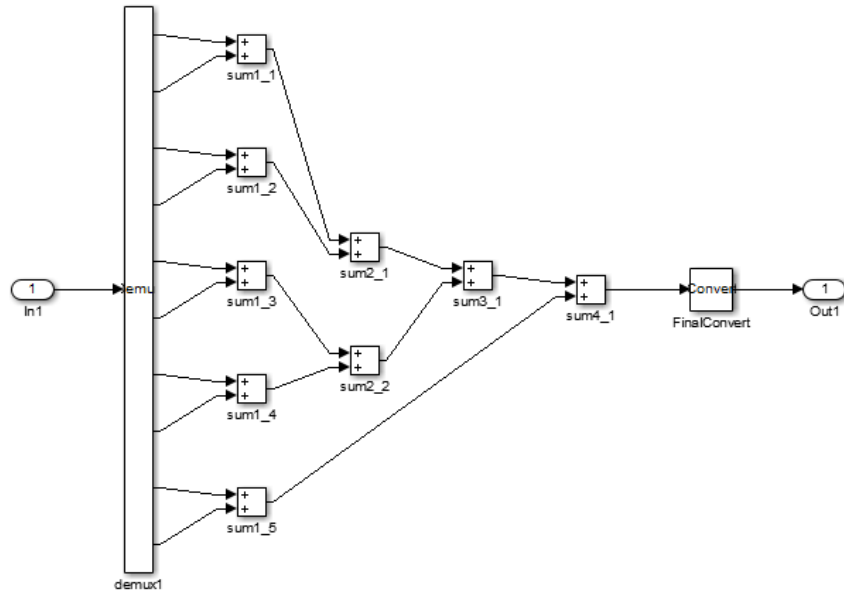


The `vsum` subsystem has been highlighted in cyan. This highlighting indicates that the subsystem differs in some respect from the `vsum` subsystem of the original model.

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 14-2
- “Optimization with Constrained Overclocking” on page 14-5
- “Maximum Oversampling Ratio” on page 14-8
- “Maximum Computation Latency” on page 14-10
- “Streaming” on page 14-12
- “Area Reduction with Streaming” on page 14-15
- “Resource Sharing” on page 14-23
- “Check Compatibility for Resource Sharing” on page 14-29
- “Delay Balancing” on page 14-30
- “Resolve Numerical Mismatch with Delay Balancing” on page 14-33
- “Find Feedback Loops” on page 14-37
- “Hierarchy Flattening” on page 14-39
- “RAM Mapping” on page 14-42
- “RAM Mapping with the MATLAB Function Block” on page 14-43
- “Insert Distributed Pipeline Registers in a Subsystem” on page 14-46
- “Distributed Pipelining and Hierarchical Distributed Pipelining” on page 14-51
- “Constrained Output Pipelining” on page 14-61
- “Reduce Critical Path with Distributed Pipelining” on page 14-63
- “Clock-Rate Pipelining” on page 14-70
- “Find Estimated Critical Paths Without Synthesis Tools” on page 14-74

Automatic Iterative Optimization

In this section...
“How Automatic Iterative Optimization Works” on page 14-2
“Automatic Iterative Optimization Output” on page 14-3
“Automatic Iterative Optimization Report” on page 14-3
“Requirements for Automatic Iterative Optimization” on page 14-4
“Limitations of Automatic Iterative Optimization” on page 14-4

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`

Optimization with Constrained Overclocking

In this section...

“Why Constrain Overclocking?” on page 14-5

“When to Use Constrained Overclocking” on page 14-5

“Set Overclocking Constraints” on page 14-6

“Constrained Overclocking Limitations” on page 14-6

Why Constrain Overclocking?

Overclocking can cause your design clock rate to exceed the maximum clock rate of your target hardware when your original clock rate is high. Without constrained overclocking, automated speed and area optimizations can modify the design implementation architecture and often result in local upsampling.

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

When to Use Constrained Overclocking

When using area and speed optimizations, you can specify constraints on overclocking using the **Max oversampling** and **Max computation latency** parameters. 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 **Max oversampling**. You can use **Max computation latency** to give 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.

To learn more about the **Max oversampling** parameter, see “Maximum Oversampling Ratio”.

To learn more about the **Max computation latency** parameter, see “Maximum Computation Latency”.

Set Overclocking Constraints

You can use the `MaxOversampling` and `MaxComputationLatency` parameters to constrain overclocking when optimizing area and speed.

The following table shows how to set `MaxOversampling` and `MaxComputationLatency` for different design implementation results:

Desired implementation result	Without Optimizations	With Optimizations
Unlimited overclocking	<code>MaxOversampling = 0</code>	<code>MaxOversampling = 0</code> Max computation latency <code>> 1</code>
Overclocking with constraints	<code>MaxOversampling > 1</code>	<code>MaxOversampling > 1</code> <code>MaxComputationLatency > 1</code>
No overclocking (single rate)	<code>MaxOversampling = 1</code>	<code>MaxOversampling = 1</code> <code>MaxComputationLatency > 1</code>

To learn how to specify `MaxOversampling` and `MaxComputationLatency`, see:

- “Specify Maximum Oversampling Ratio”
- “Specify Maximum Computation Latency”

Constrained Overclocking Limitations

When you constrain overclocking, the following limitations apply:

- If you set **Max oversampling** = 1, your DUT must be single-rate .
- Loop streaming and RAM mapping are disabled when you set **Max oversampling** = 1, even if **Max computation latency** > 1.

Maximum Oversampling Ratio

In this section...

“What Is the Maximum Oversampling Ratio?” on page 14-8

“Specify Maximum Oversampling Ratio” on page 14-8

“Maximum Oversampling Ratio Limitations” on page 14-9

What Is the Maximum Oversampling Ratio?

The **Max oversampling** ratio is the maximum ratio of the final design implementation sample rate to the original sample rate. This parameter enables you to limit clock frequency.

The following table shows the possible values for the maximum oversampling ratio and how they affect the design implementation.

Max Oversampling value	Effect on design implementation
Inf (default)	Sample rate is unconstrained.
1	Single-rate implementation; no overclocking.
> 1	Oversampling is allowed, but limited to the specified maximum.

Specify Maximum Oversampling Ratio

Using Configuration Parameters Dialog Box

In the Configuration Parameters dialog box, you can specify the maximum oversampling ratio:

- 1 In **HDL Code Generation > Global Settings**, click the **Optimization** tab.
- 2 For **Max oversampling**, enter your maximum oversampling ratio.

Using HDL Workflow Advisor

In the HDL Workflow Advisor, you can specify the maximum oversampling ratio:

- 1 In the **HDL Code Generation > Set Code Generation Options > Set Advanced Options** task, click the **Optimization** tab.

2 For **Max oversampling**, enter your maximum oversampling ratio.

On the Command Line

On the command line, set the `MaxOversampling` property using `makehdl` or `hdlset_param`.

For example, to set the maximum oversampling ratio to 4 for a subsystem, *dut*, in your model, *myModel*, enter:

```
hdlset_param ('myModel/dut', 'MaxOversampling', 4)
```

Maximum Oversampling Ratio Limitations

When the maximum oversampling ratio is 1, the following limitations apply:

- DUT subsystem must be single-rate.
- Delay balancing for the model must be enabled.
- There can be at most 1 subsystem within a subsystem hierarchy that has a nondefault `SharingFactor` or `StreamingFactor` setting.
- You cannot instantiate multiple times a subsystem with a nondefault `SharingFactor` or `StreamingFactor` setting in its subsystem hierarchy.

Maximum Computation Latency

In this section...

“What Is Maximum Computation Latency?” on page 14-10

“Specify Maximum Computation Latency” on page 14-11

“Maximum Computation Latency Restrictions” on page 14-11

What Is Maximum Computation Latency?

The **Max computation latency** parameter enables you to specify a time budget for HDL Coder when performing a single computation. Within this time budget, HDL Coder does its best to optimize your design without exceeding the **Max oversampling** ratio.

When you set a **Max computation latency**, N , each Simulink time step takes N time steps in the implemented design.

The following table shows the possible values for the maximum computation latency and their effect on the design implementation.

Max computation latency value, N	Description
$N = 1$ (default) or $N < 1$	<ul style="list-style-type: none"> Design implementation captures the DUT inputs every clock cycle. If maximum oversampling ratio is set to 1, most area optimizations are not possible. If maximum oversampling ratio is greater than 1, coder implements optimizations with local overclocking.
$N > 1$	<ul style="list-style-type: none"> Design implementation captures the DUT inputs once every N clock cycles, starting with first cycle after reset. DUT outputs are held stable for N cycles. Coder can perform optimizations without oversampling. You cannot set the maximum computation latency to Inf.

Specify Maximum Computation Latency

Using Configuration Parameters Dialog Box

In the Configuration Parameters dialog box, you can specify the maximum computation latency:

- 1 In **HDL Code Generation > Global Settings**, click the **Optimization** tab.
- 2 For **Max computation latency**, enter the number of cycles HDL Coder can use to implement a computation.

Using HDL Workflow Advisor

In the HDL Workflow Advisor, you can specify the maximum oversampling ratio:

- 1 In the **HDL Code Generation > Set Code Generation Options > Set Advanced Options** task, click the **Optimization** tab.
- 2 For **Max computation latency**, enter the number of cycles HDL Coder can use to implement a computation.

On the Command Line

On the command line, set the `MaxComputationLatency` property using `makehdl` or `hdlset_param`.

For example, if you know the inputs change at most every 1000 cycles for your DUT subsystem, `dut`, in your model, `myModel`, enter:

```
hdlset_param ('myModel/dut', 'MaxComputationLatency', 1000)
```

Maximum Computation Latency Restrictions

The maximum computation latency feature has the following restrictions:

- You cannot set the maximum computation latency to `Inf`.

Streaming

In this section...

“What Is Streaming?” on page 14-12

“Specify Streaming” on page 14-13

“Requirements and Limitations for Streaming” on page 14-13

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, the coder generates *fully parallel* implementations for vector computations. For example, the coder realizes a vector sum as a number of 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. Where 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. Depending on the width of the data path, you can also specify streaming factors between these extrema.

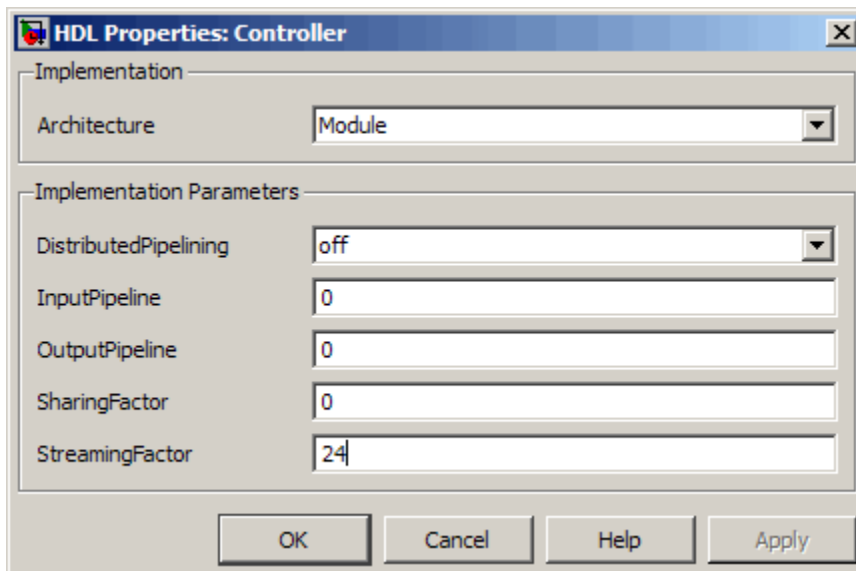
If you know the maximal vector dimensions and the sample rate for a subsystem, you can compute the possible streaming factors and resulting sample rates for the subsystem. However, even if the requested streaming factor is mathematically possible, the subsystem must meet other criteria for streaming. See “Requirements and Limitations for Streaming” on page 14-13 for details.

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. You can limit the oversampling ratio to meet target hardware clock constraints. For details, see “Optimization with Constrained Overclocking”.

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”.

Specify Streaming

You apply streaming at the subsystem level. Specify the streaming factor by setting the subsystem HDL parameter `StreamingFactor`. You can set `StreamingFactor` in the HDL Properties dialog for a subsystem, as shown in the following figure.



Alternatively, you can set `StreamingFactor` using the `hdlset_param` function, as in the following example.

```
dut = 'ex_pdcontroller_multi_instance/Controller';
hdlset_param(dut, 'StreamingFactor', 24);
```

Requirements and Limitations for Streaming

This section describes the criteria for streaming that subsystems must meet.

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 streaming request fails.

HDL Coder cannot apply the streaming optimization to a model reference.

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 can be either of the following:

- Of width $V_{stream} = (V_{dim} / N_{st})$
- Scalar

If the original subsystem operated with a sample time S , then the streamed subsystem operates with a sample time of S / N_{st} .

Checks and 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 it meets all the following criteria:

- The streaming factor N_{st} must be a perfect divisor of the vector width V_{dim} .
- The subsystem must be a single-rate subsystem that does not contain rate changes or rate transitions.

Because of this requirement, do not specify HDL implementations that are inherently multirate for blocks within the subsystem. For example, using the Cascade implementation (for the Sum, Product, MinMax, and other blocks) is not allowed within a streamed subsystem.

- All vector data paths in the subsystem must have the same widths.
- The subsystem must not contain nested subsystems.
- All blocks within the subsystem must support streaming. HDL Coder supports many blocks for streaming. As a best practice, run `checkhdl` 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 streaming request fails.

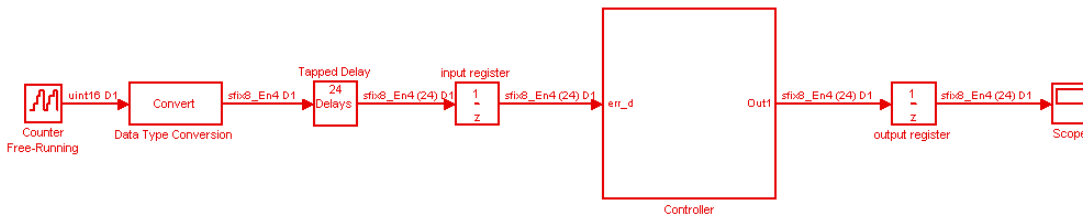
If the requested streaming factor cannot be implemented, HDL Coder generates non-streaming code. It is good practice to generate an Optimization Report. The Streaming and Sharing page of the report provides information about conditions that prevent streaming.

Area Reduction with Streaming

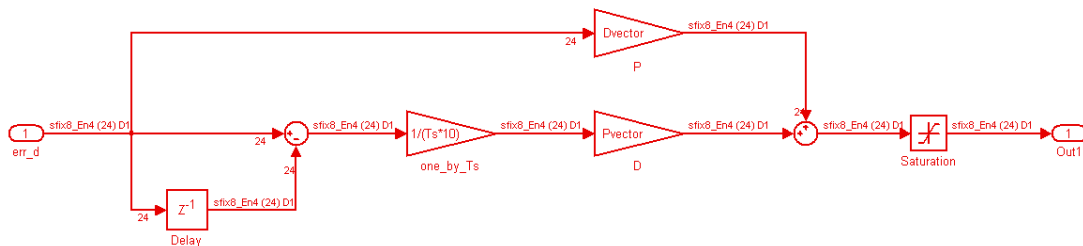
This example illustrates:

- Specification of a streaming factor for a subsystem
- Generation of HDL code and a validation model for the subsystem.

The following example is a single-rate model that drives the **Controller** subsystem with a vector signal of width 24.



The following figure shows the **Controller** subsystem, which is the DUT in this example.



By generating HDL code and a report on resource utilization, you can determine how many multipliers, adders/subtractors, registers, RAMs, and multiplexers are generated from this DUT in the default case. To do so, type the following commands:

```

dut = 'ex_pdcontroller_multi_instance/Controller';
hdlset_param(dut,'StreamingFactor', 0);
makehdl(dut,'ResourceReport','on');

```

The following figure shows the Resource Utilization Report for the Controller subsystem. The report shows the number of multipliers, adders/subtractors, registers, RAMs, and multiplexers that the HDL Coder software generates.

Resource Utilization Report for ex_pdcontroller_multi_instance

Summary

Multipliers	46
Adders/Subtractors	48
Registers	24
RAMs	0
Multiplexers	24

Detailed Report

[Expand all] [Collapse all]

Report for Subsystem: [Controller](#)

Multipliers (46)

[+] 8x8-bit Multiply : 46

Adders/Subtractors (48)

[+] 32x32-bit Adder : 24

[+] 32x32-bit Subtractor : 24

Registers (24)

[+] 8-bit Register : 24

Multiplexers (24)

[+] 8-bit 3-to-1 Multiplexer : 24

If you choose an optimal StreamingFactor for the DUT, you can achieve a drastic reduction in the number of multipliers and adders/subtractors generated. The following

commands set **StreamingFactor** to the largest possible value for this subsystem and then generate VHDL code and a Resource Utilization Report.

```
dut = 'ex_pdcontroller_multi_instance/Controller';
hdlset_param(dut,'StreamingFactor', 24);
makehdl(dut,'ResourceReport','on', 'GenerateValidationModel','on');
```

During code generation, HDL Coder reports latency in the generated model. It also reports generation of the validation model.

```
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### The DUT requires an initial pipeline setup latency. Each output port experiences
these additional delays
### Output port 0: 1 cycles

### Generating new validation model: gm_ex_pdcontroller_multi_instance4_vnl.mdl
### Validation Model Generation Complete.

### Begin VHDL Code Generation
### MESSAGE: The design requires 24 times faster clock with respect to the base rate = 2.
### Working on ex_pdcontroller_multi_instance/Controller/err_d_serializercomp
as hdlsrc\err_d_serializercomp.vhd
### Working on ex_pdcontroller_multi_instance/Controller/Saturation_out1_serializercomp
as hdlsrc\Saturation_out1_serializercomp.vhd
### Working on ex_pdcontroller_multi_instance/Controller/kconst_serializercomp
as hdlsrc\kconst_serializercomp.vhd
### Working on ex_pdcontroller_multi_instance/Controller/kconst_serializercomp1
as hdlsrc\kconst_serializercomp1.vhd
### Working on Controller_tc as hdlsrc\Controller_tc.vhd
### Working on ex_pdcontroller_multi_instance/Controller as hdlsrc\Controller.vhd
### Generating package file hdlsrc\Controller_pkg.vhd
### Generating HTML files for code generation report in
C:\hdlsrc\html\ex_pdcontroller_multi_instance directory ...

### HDL Code Generation Complete.
```

After code generation completes, you can view the results of the **StreamingFactor** optimization. In the Resource Utilization Report, you can see that HDL Coder generates only 2 multipliers and 2 adders for the Controller subsystem.

Resource Utilization Report for ex_pdcontroller_multi_instance

Summary

Multipliers	2
Adders/Subtractors	2
Registers	210
RAMs	0
Multiplexers	97

Detailed Report

[Expand all] [Collapse all]

Report for Subsystem: [Controller](#)

Multipliers (2)

[+] 8x8-bit Multiply : 2

Adders/Subtractors (2)

[+] 32x32-bit Adder : 1

[+] 32x32-bit Subtractor : 1

Registers (210)

1-bit Register : 3

[+] 8-bit Register : 207

Multiplexers (97)

1-bit 2-to-1 Multiplexer : 3

8-bit 2-to-1 Multiplexer : 27

8-bit 45-to-1 Multiplexer : 66

[+] 8-bit 3-to-1 Multiplexer : 1

HDL Coder also produces a Streaming and Sharing report that shows:

- The StreamingFactor value that you specified

- The other usable `StreamingFactor` values for this subsystem
- Latency (delays) introduced in the generated model
- A hyperlink to the validation model, if generated

Streaming and Sharing Report for `ex_pdcontroller_multi_instance`

Subsystem	StreamingFactor	SharingFactor
Controller	24	0

Streaming Report

Subsystem: [Controller](#)

StreamingFactor: 24

Streaming successful with factor 24. Other possible factors: [2 3 4 6 8 12 24]

Sharing Report

No subsystem(s) found with SharingFactor > 0

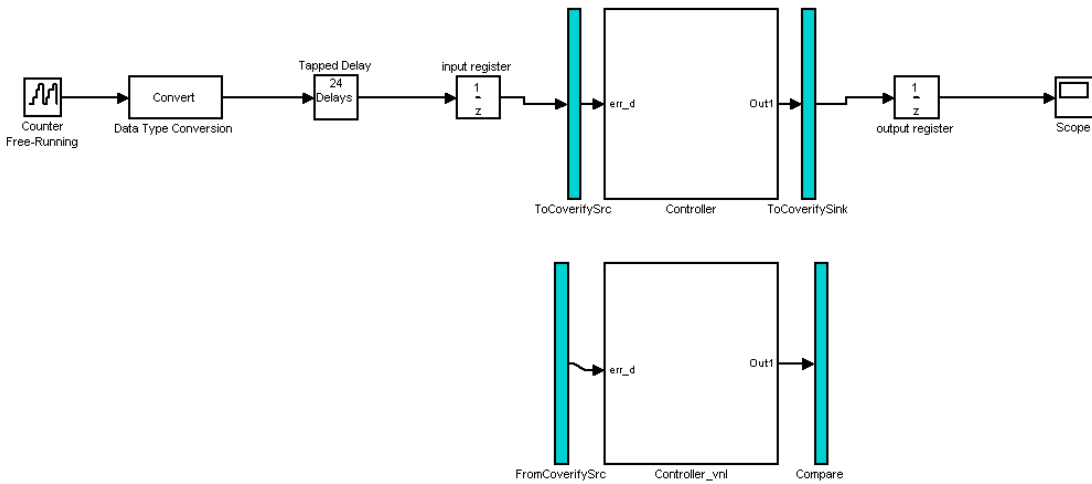
Path Delay Summary

Port	Path Delay
Controller/ce_out	1
Controller/Out1	1

Validation model: [gm_ex_pdcontroller_multi_instance4_vnl](#)

The Validation Model

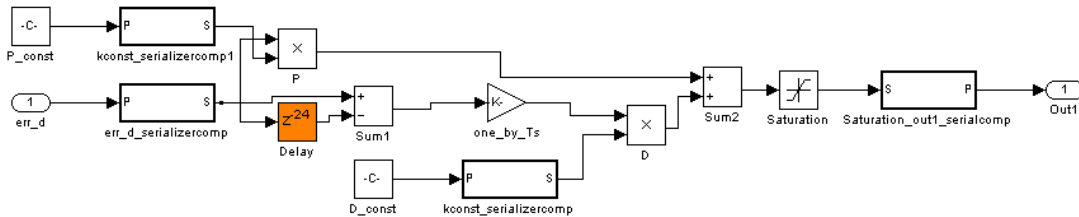
The following figure shows the validation model generated for the Controller subsystem.



The lower section of the validation model contains a copy of the original DUT (Controller_vnl). This single-rate subsystem runs at its original rate.

The upper section of the validation model contains the streaming version of the DUT (Controller). Internally, this subsystem runs at a different rate than the original DUT.

The following figure shows the interior of the Controller subsystem.

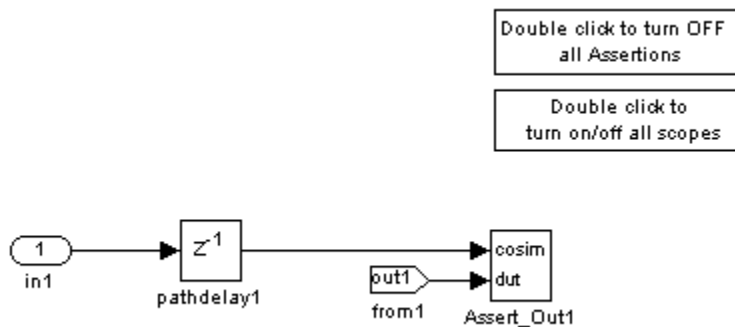


Inspection of the Controller subsystem shows that it is a multirate subsystem, having two rates that operate as follows:

- Inputs and outputs run at the same rate as the exterior model.
- Dual-rate Serializer blocks receive vector data at the original rate and output a stream of scalar values at the higher (24x) rate.

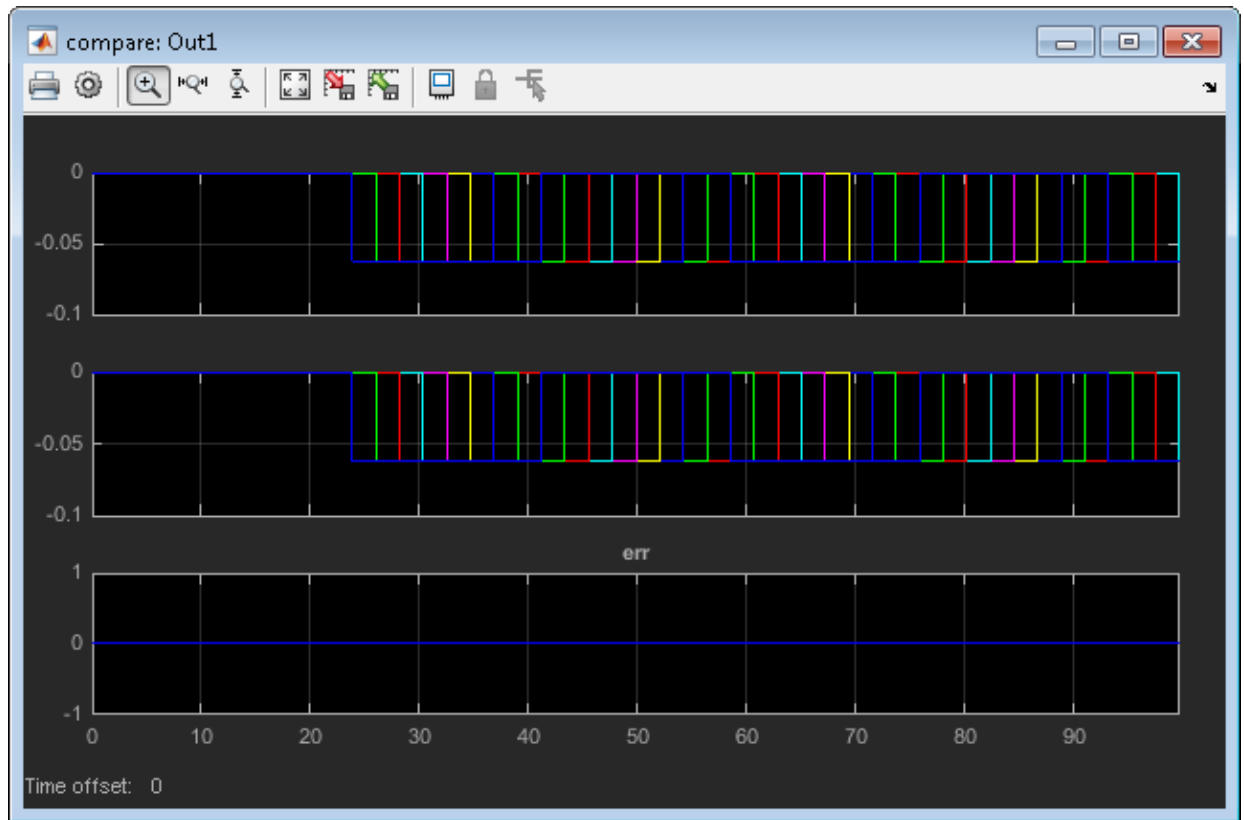
- Interior blocks between Serializers and Deserializer run at the higher rate.
- The Deserializer block receives scalar values at the higher rate and buffers values into a 24-element output vector running at the original rate.

The Compare subsystem (see following figure) receives and compares outputs from the Controller and Controller_vnl subsystems. To compensate for the latency of the Controller subsystem (reported during code generation), input from the Controller_vnl subsystem is delayed by one clock cycle. A discrepancy between the outputs of the two subsystems triggers an assertion.



To verify that a generated model with streaming is bit-true to its original counterpart in a validation model:

- 1 Open the Compare subsystem.
- 2 Double click the **Double click to turn on/off all scopes** button.
- 3 Run the validation model.
- 4 Observe the **compare:Out1** scope. The error signal display shows a line through zero, indicating that the data comparisons were equal.



Resource Sharing

In this section...

“What Is Resource Sharing?” on page 14-23

“Specify Resource Sharing” on page 14-24

“Requirements and Limitations for Resource Sharing” on page 14-24

“Resource Sharing Information in Reports” on page 14-27

What Is Resource Sharing?

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.

You can specify a *sharing factor*, N , for a subsystem, model reference, or MATLAB Function block. HDL Coder tries to identify up to N shareable resources, and, by default, oversamples by a factor of N to generate an area-optimized implementation with the original latency. If the coder cannot identify N shareable resources, it shares as many as it can, but still oversamples by a factor of N .

You can limit the oversampling ratio to meet target hardware clock constraints. For details, see “Optimization with Constrained Overclocking”.

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”.

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 or model reference, HDL Coder identifies and shares functionally equivalent instances of the following types of blocks:

- Gain
- Product

- Atomic Subsystem
- MATLAB Function
- Model reference

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.

Specify Resource Sharing

Specify Resource Sharing from the UI

To specify resource sharing from the UI:

- 1 Right-click the subsystem, model reference, or MATLAB Function block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 In the **SharingFactor** field, enter the number of shareable resources.

Specify Resource Sharing from the Command Line

Set the `SharingFactor` using `hdlset_param`, as in the following example.

```
dut = 'ex_dimcheck/Channel';  
hdlset_param(dut, 'SharingFactor', 3);
```

Requirements and Limitations for Resource Sharing

On a Subsystem block, model reference, or MATLAB Function block, you can specify the resource sharing optimization.

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
```

Blocks to be shared have the following requirements:

- Single-rate.
- If the block is within a feedback loop, at least one Unit Delay or Delay block connected to each output port.
- If you set the maximum oversampling ratio to 1, shared resources cannot be inside feedback loops.

If you want to share Atomic Subsystem blocks:

- The only state elements that these blocks can contain are:
 - Delay
 - Unit Delay
 - Unit Delay Enabled
 - Unit Delay Resettable
 - Unit Delay Enabled Resettable

These blocks must have the **Initial condition** parameter set to 0.

- These blocks must not contain a subsystem that does not meet the requirements for resource sharing.

If you want to share MATLAB Function blocks, they must not use:

- Persistent variables
- Loop streaming
- Output pipelining

If you want to share model reference instances, all model references that point to the same referenced model must have the same rate after optimizations and rate

propagation. The model reference final rate can differ from the original rate, but all model references that point to the same referenced model must have the same final rate.

Functionally Equivalent Blocks for Resource Sharing

Block Type	Functionally equivalent if they have...
Product	<ul style="list-style-type: none"> • Equivalent input and output data types • Equivalent rounding and saturation modes
Gain	<ul style="list-style-type: none"> • Equivalent input and output data types • Equivalent rounding and saturation modes <p>Gain constants can have different values, but they must have the same data type.</p>
Atomic Subsystem MATLAB Function Model reference	<ul style="list-style-type: none"> • The same Simulink checksum. For atomic subsystems and MATLAB Function blocks, use <code>Simulink.Subsystem.getChecksum</code> to determine the checksum. For Model references, use <code>Simulink.BlockDiagram.getChecksum</code>. • The same HDL block properties.

In addition, if you use the `DSPStyle` block property, HDL Coder does not share multipliers that have different synthesis attribute settings.

Data Dependency Limitation

Specify the sharing factor as the exact number of shareable resources when both of the following are true:

- A shareable resource depends on output data from another shareable resource.
- One or more outputs of a shareable resource is a vector.

In this situation, if your sharing factor does not match the actual number of shareable resources, HDL Coder does not perform resource sharing.

Limitations for Atomic Subsystem Sharing

You cannot apply resource sharing to a subset of instances for a particular atomic subsystem; you must share all instances. If you want to share a subset of atomic subsystem instances, change the remaining instances to virtual subsystems.

The only blocks with state elements that can be part of a shared atomic subsystem are:

- Delay
- Unit Delay
- Unit Delay Enabled
- Unit Delay Resettable
- Unit Delay Enabled Resettable

Therefore, 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
- Communications System Toolbox blocks
- DSP System Toolbox blocks
- Stateflow blocks
- Blocks that are not supported for delay balancing. For details, see “Delay Balancing Limitations”.

Resource Sharing Information in Reports

If you generate a code generation report, for each subsystem, model reference, or MATLAB Function block that implements sharing, the report includes the following information:

- Success: Provides a list of resource usage changes caused by sharing.

- Failure: Identifies which criterion was violated.
- Latency changes.

Related Examples

- Resource Sharing For Area Optimization
- Resource Sharing and Streaming with Oversampling Constraints

Check Compatibility for Resource Sharing

To determine whether your model is compatible for resource sharing:

- 1 Before generating code, run `checkhdl` and eliminate general compatibility issues.
- 2 In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select **Generate optimization report**.
- 3 Set the sharing factor for the DUT and generate code.
- 4 After code generation completes, inspect the Optimization Report. The report shows incompatible blocks or other conditions that can cause a resource sharing request to fail.
- 5 If the Optimization Report shows problems, fix them and repeat these steps.

See also “Requirements and Limitations for Resource Sharing”.

Delay Balancing

In this section...

“Why Use Delay Balancing” on page 14-30

“Specify Delay Balancing” on page 14-30

“Delay Balancing Limitations” on page 14-32

Why Use Delay Balancing

The HDL Coder software 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 inherently introduce delays in the generated code. “Resolve Numerical Mismatch with Delay Balancing” on page 14-33 discusses one such implementation.

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*. When you enable delay balancing and the coder detects introduction of new delays along one path, the coder inserts matching delays on the other paths. When delay balancing is enabled, the generated model is functionally equivalent to the original model.

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 **BalanceDelays**.
- **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 **GenerateValidationModel**.

For example, the following commands generate HDL code with delay balancing and generate a validation model.

```
dut = 'ex_rsqrtdelaybalancing/Subsystem';  
makehdl(dut, 'BalanceDelays', 'on', 'GenerateValidationModel', 'on');
```

For more information about the validation model, see “Validation Model”.

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”.

Delay Balancing Limitations

The following blocks do not support delay balancing:

- Cosimulation
- Data Type Duplicate
- Decrement To Zero
- Frame Conversion
- Ground
- HDL FFT
- LMS Filter
- Model Reference
- To VCD File
- Magnitude-Angle to Complex

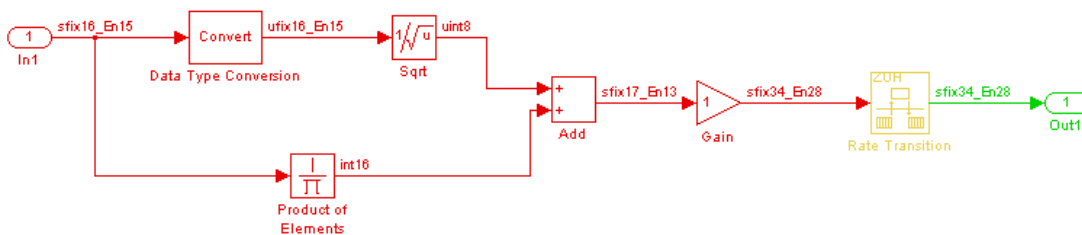
The following block implementations do not currently support delay balancing:

- `hdldefaults.ConstantSpecialHDLEmission`
- `hdldefaults.NoHDL`

Resolve Numerical Mismatch with Delay Balancing

This example shows a simple case where the VHDL implementation of a block introduces delays that cause a numerical mismatch between the original DUT and the generated model and HDL code. The example then demonstrates how to use delay balancing to fix the mismatch.

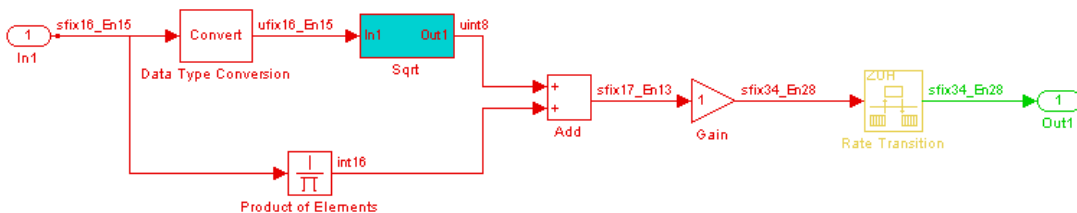
The following figure shows the DUT for the `ex_rsqrt_delaybalancing` model. The DUT is a simple multirate subsystem that includes a Reciprocal Square Root block, `Sqrt`. A Rate Transition block downsamples the output signal to a lower sample rate.



Generate HDL code without delay balancing and generate a validation model:

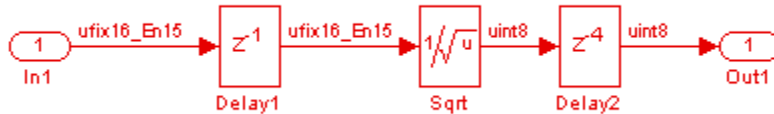
```
dut = 'ex_rsqrt_delaybalancing/Subsystem';
makehdl(dut, 'BalanceDelays', 'off', 'GenerateValidationModel', 'on');
```

Examination of the generated model shows that HDL Coder has implemented the `Sqrt` block as a subsystem:



The following figure shows that the generated `Sqrt` subsystem introduces a total of 5 cycles of delay. (This behavior is inherent to the Reciprocal Square Root block

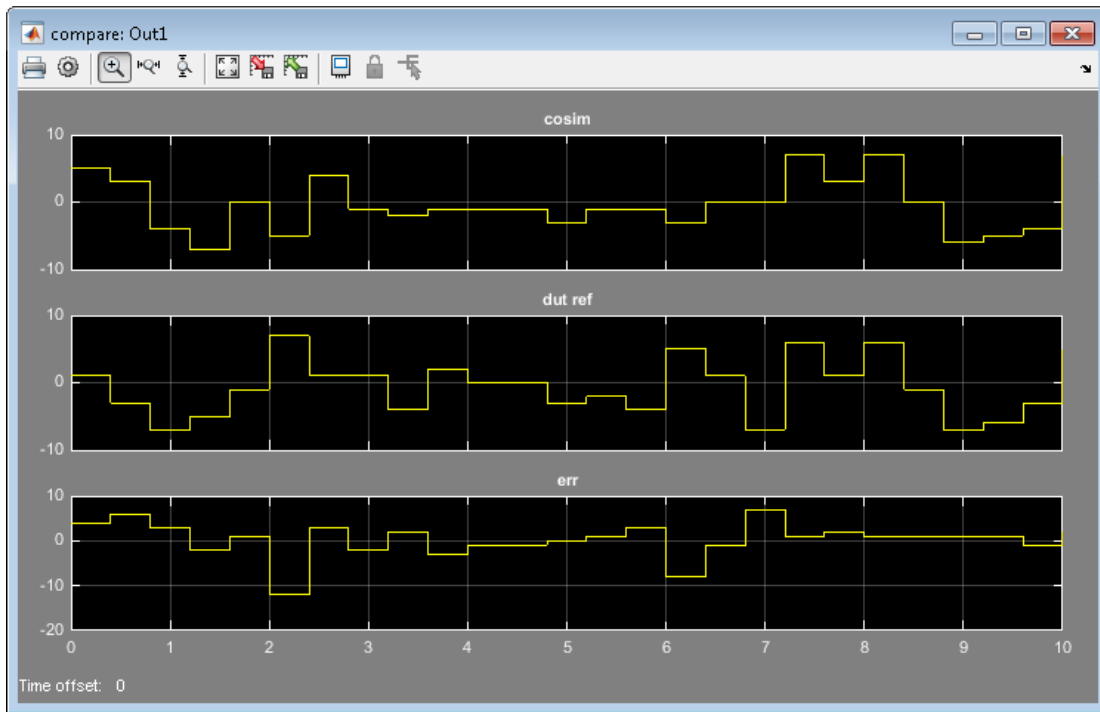
implementation.) These delays map to registers in the generated HDL code when **UseRAM** is off.



The scope in the following figure shows the results of a comparison run between the original and generated models. The scope displays the following signals, in descending order:

- The outputs from the original model
- The outputs from the generated model
- The difference between the two

The difference is nonzero, indicating a numerical mismatch between the original and generated models.



Two factors cause this discrepancy:

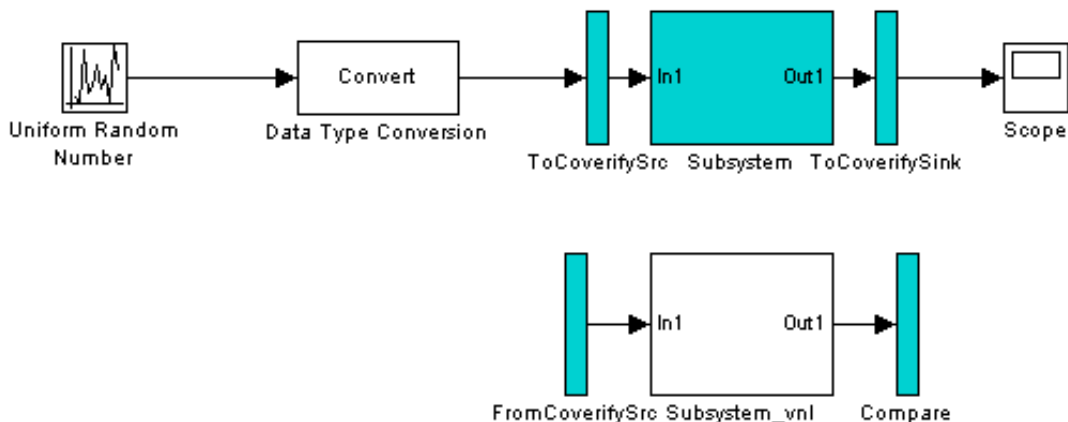
- The input signal branches into two parallel paths (to the `Sqrt` and product blocks) but only the branch to the `Sqrt` block introduces delays.
- The downsampling caused by the rate transition drops samples.

You can solve these problems by manually inserting delays in the generated model. However, using delay balancing produces more consistent results.

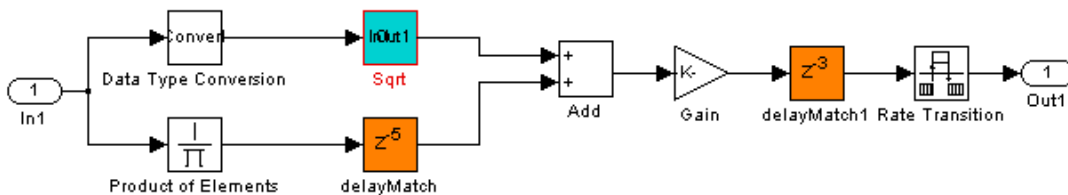
Generate HDL code with delay balancing and generate a validation model:

```
dut = 'ex_rsqrt_delaybalancing/Subsystem';
makehdl(dut, 'BalanceDelays', 'on', 'GenerateValidationModel', 'on');
```

The following figure shows the validation model. The lower subsystem is identical to the original DUT. The upper subsystem represents the HDL implementation of the DUT.



The upper subsystem (shown in the following figure) represents the HDL implementation of the DUT. To balance the 5-cycle delay from the `Sqrt` subsystem, HDL Coder has inserted a 5-cycle delay on the parallel data path. The coder has also inserted a 3-cycle delay before the Rate Transition to offset the effect of downsampling.



Find Feedback Loops

In this section...

“Using the HDL Workflow Advisor” on page 14-37

“Using the Configuration Parameters Dialog Box” on page 14-38

“Using the Command Line” on page 14-38

“Remove Highlighting” on page 14-38

“Limitations” on page 14-38

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

Using the HDL Workflow Advisor

In the **HDL Code Generation > Set Code Generation Options > Set Advanced Options > Diagnostics** tab, select **Highlight feedback loops inhibiting delay balancing and optimizations**.

To customize the script name, enter a file name in the **Feedback loop highlighting script file name** field.

Using the Configuration Parameters Dialog Box

In the **HDL Code Generation > Global Settings > Diagnostics** tab, select **Highlight feedback loops inhibiting delay balancing and optimizations**.

To customize the script name, enter a file name in the **Feedback loop highlighting script file name** field.

Using the Command Line

To generate a feedback loop highlighting script programmatically, use the `HighlightFeedbackLoops` and `HighlightFeedbackLoopsFile` properties with `makehdl` or `hdlset_param`.

For example:

- To generate a feedback loop highlight script for a model, *myModel*, enter:

```
hdlset_param ('myModel', 'HighlightFeedbackLoops', 'on');
```
- To generate a feedback loop highlighting script with the file name, *myHighlightScript*, for a model, *myModel*, enter:

```
hdlset_param ('myModel', 'HighlightFeedbackLoops', 'on');  
hdlset_param ('myModel', 'HighlightFeedbackLoopsFile', 'myHighlightScript');
```

Remove Highlighting

To turn off highlighting, in Simulink, select **Display > Remove Highlighting**.

Limitations

Feedback loop highlighting cannot highlight blocks that have names that contain a single quote (').

See Also

`HighlightFeedbackLoops` | `HighlightFeedbackLoopsFile`

More About

- “Optimization Report”

Hierarchy Flattening

In this section...

“What Is Hierarchy Flattening?” on page 14-39

“When to Flatten Hierarchy” on page 14-39

“Prerequisites for Hierarchy Flattening” on page 14-39

“Options for Hierarchy Flattening” on page 14-40

“How to Flatten Hierarchy” on page 14-40

“Limitations for Hierarchy Flattening” on page 14-41

What Is Hierarchy Flattening?

Hierarchy flattening enables you to remove subsystem hierarchy from the HDL code generated from your design.

The HDL Coder software 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

Flatten hierarchy to:

- Enable more extensive area and speed optimization.
- Reduce the number of HDL output files. For every subsystem flattened, HDL Coder generates one less HDL output file.

Avoid flattening hierarchy if you want to preserve one-to-one mapping from subsystem name to HDL `module` or `entity` name. Not flattening hierarchy makes the HDL code more readable.

Prerequisites for Hierarchy Flattening

To flatten hierarchy, a subsystem must have the following block properties.

Property	Required value
DistributedPipelining	'off'
StreamingFactor	0
SharingFactor	0

To flatten hierarchy, you must also have the `MaskParameterAsGeneric` global property set to 'off'. For more information, see `MaskParameterAsGeneric`.

Options for Hierarchy Flattening

By default, a subsystem inherits its hierarchy flattening setting from the parent subsystem. However, you can enable or disable flattening for individual subsystems.

The 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.

How to Flatten Hierarchy

To set hierarchy flattening using the HDL Block Properties dialog box:

- 1 Right-click the subsystem.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 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:

- Atomic and instantiated in the design more than once.
- A black box implementation or model reference.
- A triggered subsystem when `TriggerAsClock` is enabled
- A masked subsystem.

Note: This option removes subsystem boundaries before code generation. It does not necessarily generate HDL code with a completely flat hierarchy.

RAM Mapping

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”.
- `MapPersistentVarsToRAM` to map persistent arrays in a MATLAB Function block to RAM. For details, see “MapPersistentVarsToRAM”.
- RAM blocks from the HDL Operations library:
 - Dual Port RAM
 - Dual Rate Dual Port RAM
 - Simple Dual Port RAM
 - Single Port RAM
- Blocks with a RAM implementation.

RAM Mapping with the MATLAB Function Block

This example shows how to map persistent arrays to RAM using the `MapPersistentVarsToRAM` block-level parameter. The resource report shows the area improvement with RAM mapping.

- 1 Open the `hdlcoder_sobel_serial_em1` model.

`hdlcoder_sobel_serial_em1`

The `sobel_edge_hardware` subsystem contains `sobel_edge_em1`, a MATLAB Function block that uses persistent arrays. To view the MATLAB code, double-click the `sobel_edge_em1` block.

`hdlcoder_sobel_serial_em1` ▶ `sobel_edge_hardware`



- 2 In the `sobel_edge_hardware` subsystem, right-click the `sobel_edge_em1` block and select **HDL Code > HDL Block Properties**.
- 3 Set **MapPersistentVarsToRAM** to **off** and click **OK** to disable RAM mapping.
- 4 In the **Simulation > Model Configuration Parameters > HDL Code Generation** pane, enable **Generate resource utilization report** and click **Apply**.
- 5 Click **Generate** to generate HDL code. The Code Generation Report appears.
- 6 Select **High-level Resource Report**.

Generic Resource Report for hdlcoder_sobel_serial_eml

Summary

Multipliers	0
Adders/Subtractors	18
Registers	218
RAMs	0
Multiplexers	5

The design uses 218 registers and no RAM.

- 7 Now, enable RAM mapping: right-click the `sobel_edge_eml` block, select **HDL Code > HDL Block Properties**, and set **MapPersistentVarsToRAM** to **on**. Click **OK**.
- 8 In the **Simulation > Model Configuration Parameters > HDL Code Generation** pane, click **Generate** to generate HDL code. The Code Generation Report appears.
- 9 Select **High-level Resource Report**.

Generic Resource Report for hdlcoder_sobel_serial_eml

Summary

Multipliers	1
Adders/Subtractors	19
Registers	25
RAMs	2
Multiplexers	7

The design now uses 25 registers and 2 RAMs.

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.

For more information, see:

- “MATLAB Function Block Design Patterns for HDL”
- “MapPersistentVarsToRAM”

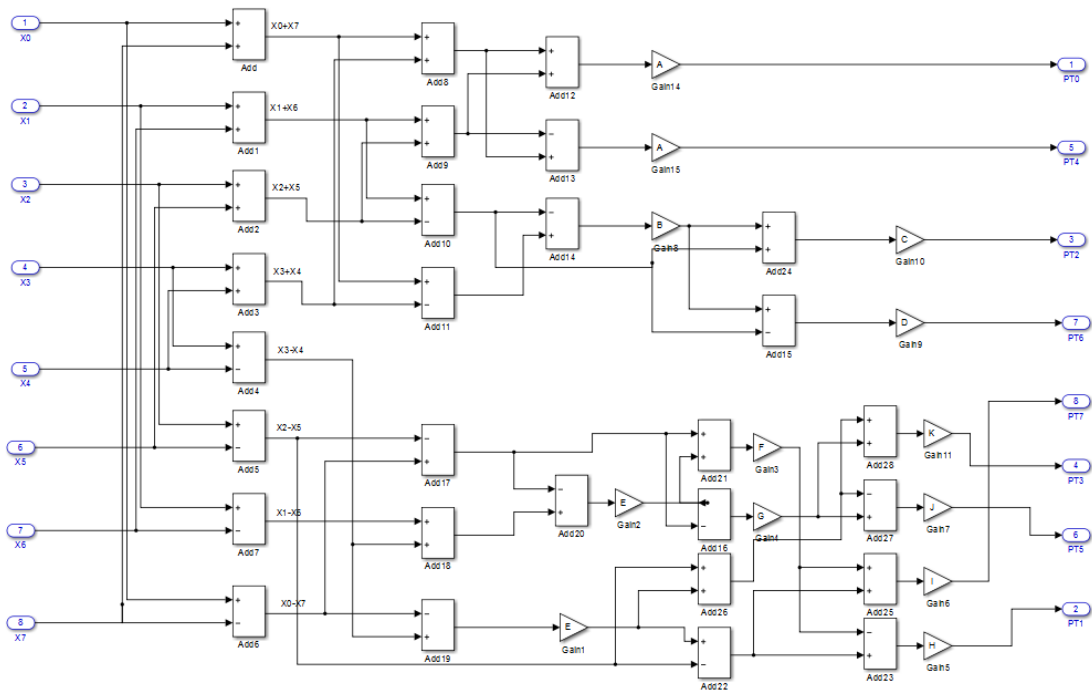
Insert Distributed Pipeline Registers in a Subsystem

This example shows how to use distributed pipelining with the `dct8_fixed` model.

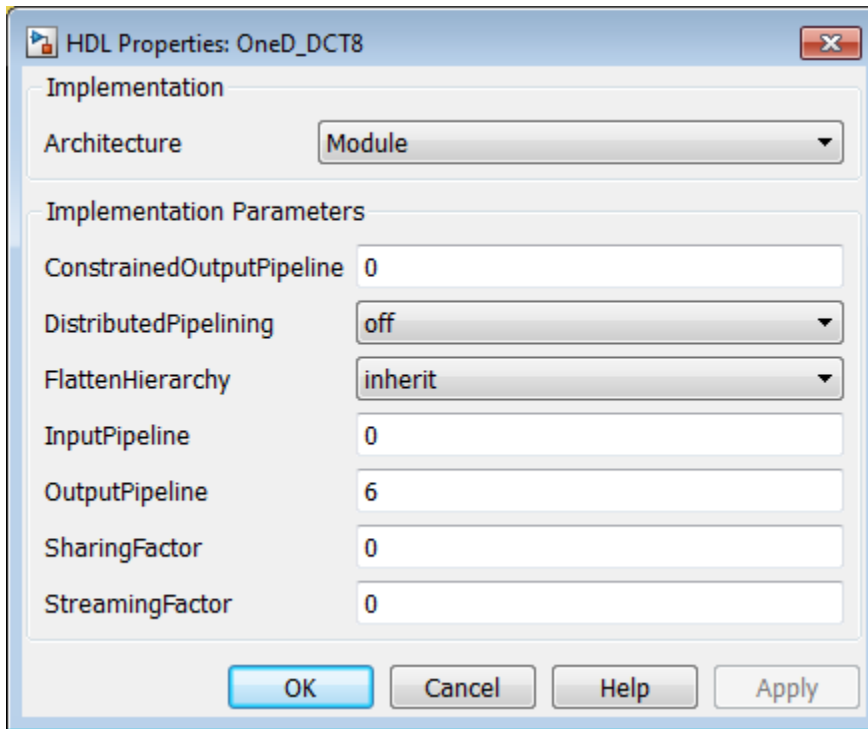
This example uses the following optimizations:

- Output pipelining
- Distributed pipelining

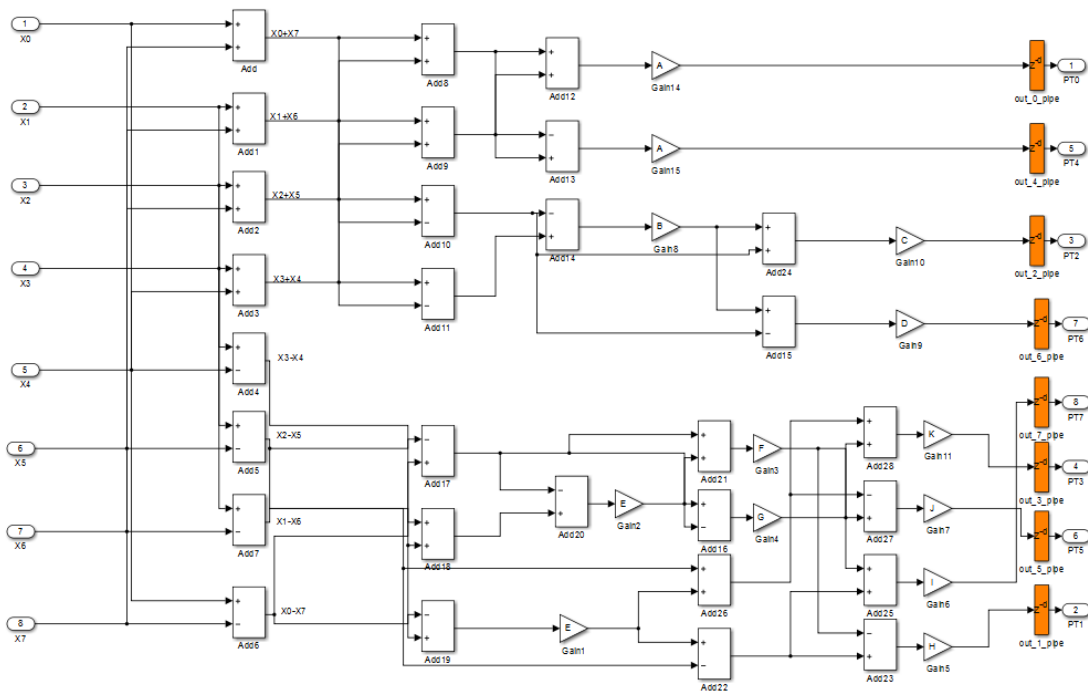
Open the model by typing `dct8_fixed` at the MATLAB prompt. The DUT is the `dct8_fixed/OneD_DCT8` subsystem.



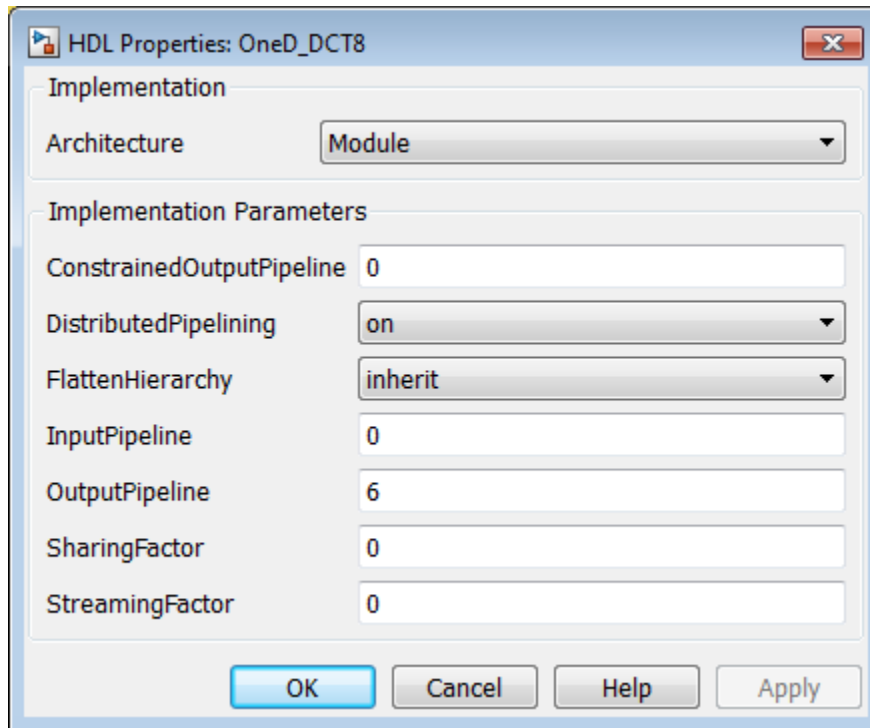
Set **DistributedPipelining** to off and **OutputPipeline** to 6 to insert 6 pipeline stages at the outputs of the DUT.



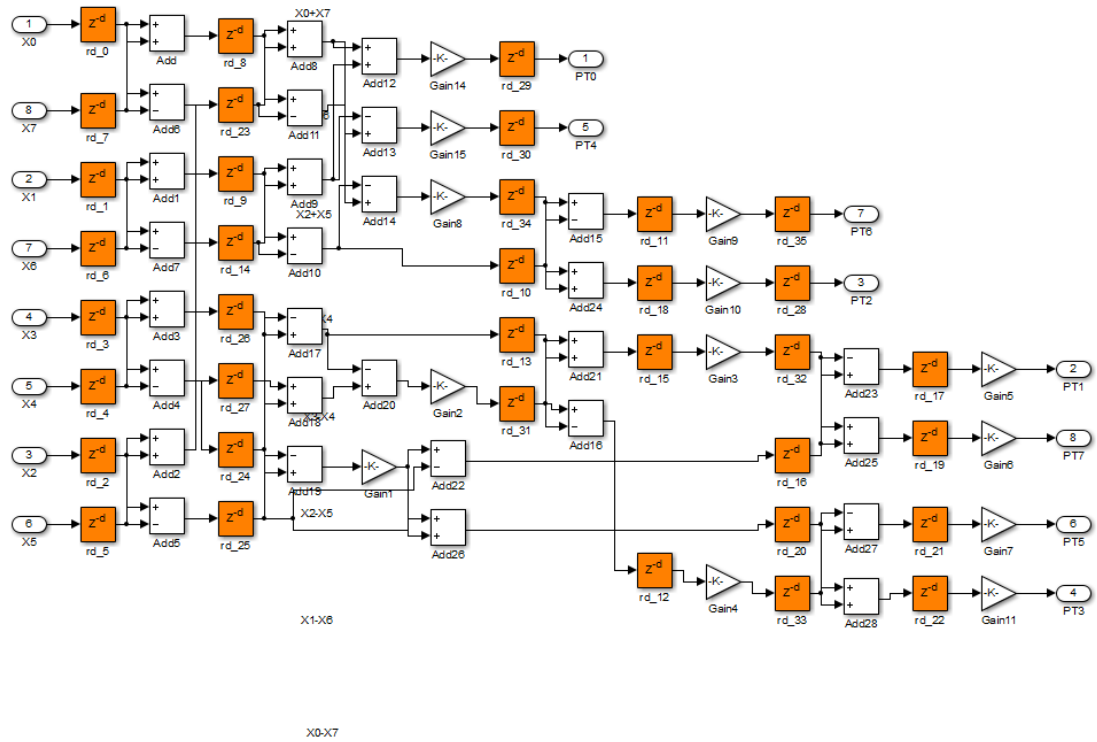
The generated model shows the placement of pipeline registers as highlighted delays at the outputs of the DUT. For more information about generated models, see “Generated Model and Validation Model”.



Set **DistributedPipelining** to on and **OutputPipeline** to 6 to distribute 6 pipeline stages for each signal path in the DUT.



The generated model shows the distribution of pipeline registers as highlighted delays within each signal path. There are 6 pipeline registers for each path.



Distributed Pipelining and Hierarchical Distributed Pipelining

In this section...

“What Is Distributed Pipelining?” on page 14-51

“Benefits and Costs of Distributed Pipelining” on page 14-53

“Requirements for Distributed Pipelining” on page 14-54

“Specify Distributed Pipelining” on page 14-54

“Limitations of Distributed Pipelining” on page 14-54

“What Is Hierarchical Distributed Pipelining?” on page 14-56

“Benefits of Hierarchical Distributed Pipelining” on page 14-59

“Specify Hierarchical Distributed Pipelining” on page 14-59

“Limitations of Hierarchical Distributed Pipelining” on page 14-60

“Distributed Pipelining Workflow” on page 14-60

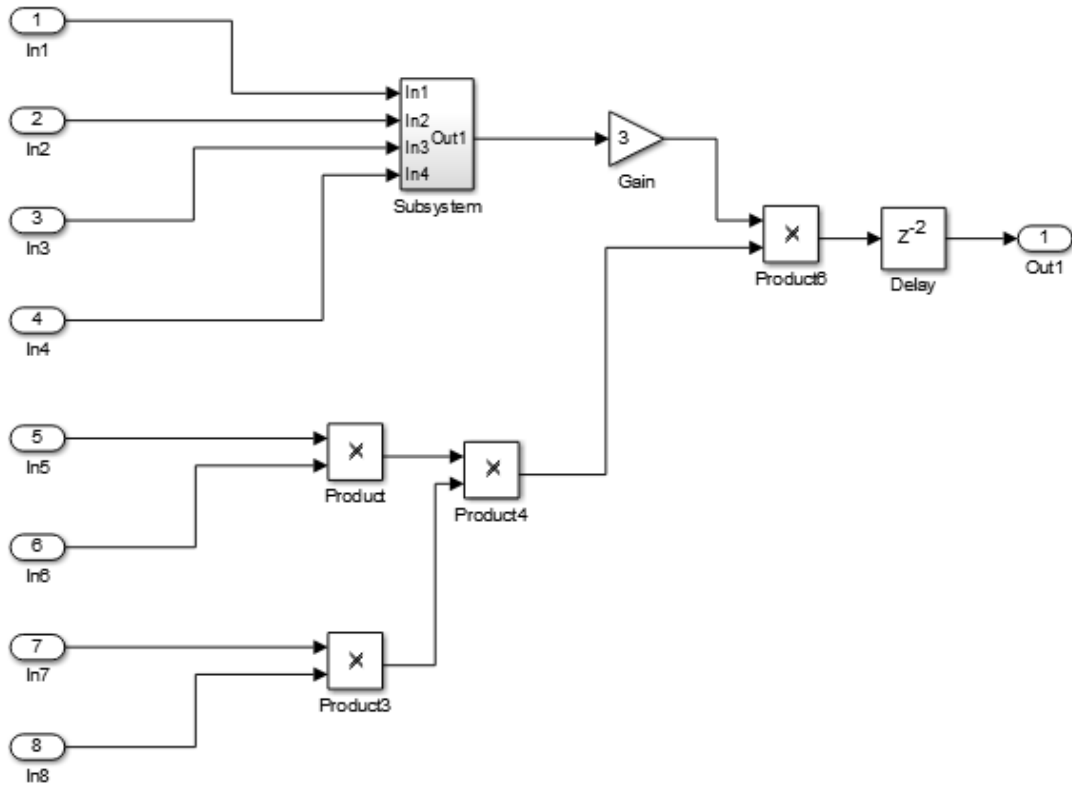
“Selected Bibliography” on page 14-60

What Is Distributed Pipelining?

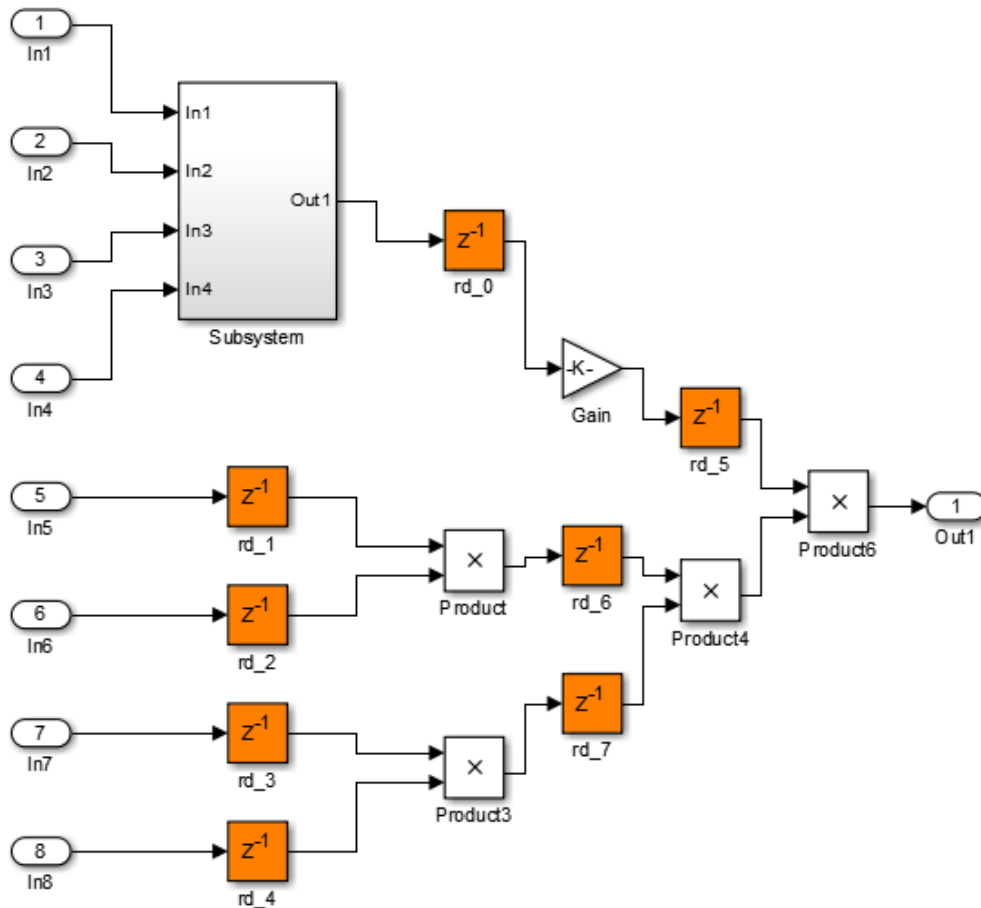
Distributed pipelining, or register retiming, is a speed optimization that moves existing delays within a design to reduce the critical path while preserving functional behavior.

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 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.
- MATLAB Function block within a subsystem. For details, see “Distributed Pipeline Insertion for MATLAB Function Blocks”.
- Stateflow chart within a subsystem.

To specify distributed pipelining using the UI:

- 1 Right-click the block and select **HDL Code > HDL Block Properties**.
- 2 Set **DistributedPipelining** to **on** and click **OK**.

To enable distributed pipelining, on the command line, enter:

```
hdlset_param('path/to/block', 'DistributedPipelining', 'on')
```

To disable distributed pipelining, on the command line, enter:

```
hdlset_param('path/to/block', 'DistributedPipelining', 'off')
```

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)”
 - `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:
 - A MATLAB Function block or Stateflow chart contains a matrix with a statically unresolvable index.
 - A Stateflow chart contains a state or local variable.
- HDL Coder distributes pipeline registers around the following blocks instead of within them:
 - Model
 - Sum (Cascade implementation)
 - Product (Cascade implementation)
 - MinMax (Cascade implementation)
 - 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 blocks in the following list, HDL Coder issues an error message and terminates code generation. To enable code generation to proceed, place these blocks inside one or more subsystems within the original subsystem and disable hierarchical distributed pipelining. HDL Coder distributes pipeline registers around nested Subsystem blocks.
 - Tapped Delay
 - M-PSK Demodulator Baseband
 - M-PSK Modulator Baseband

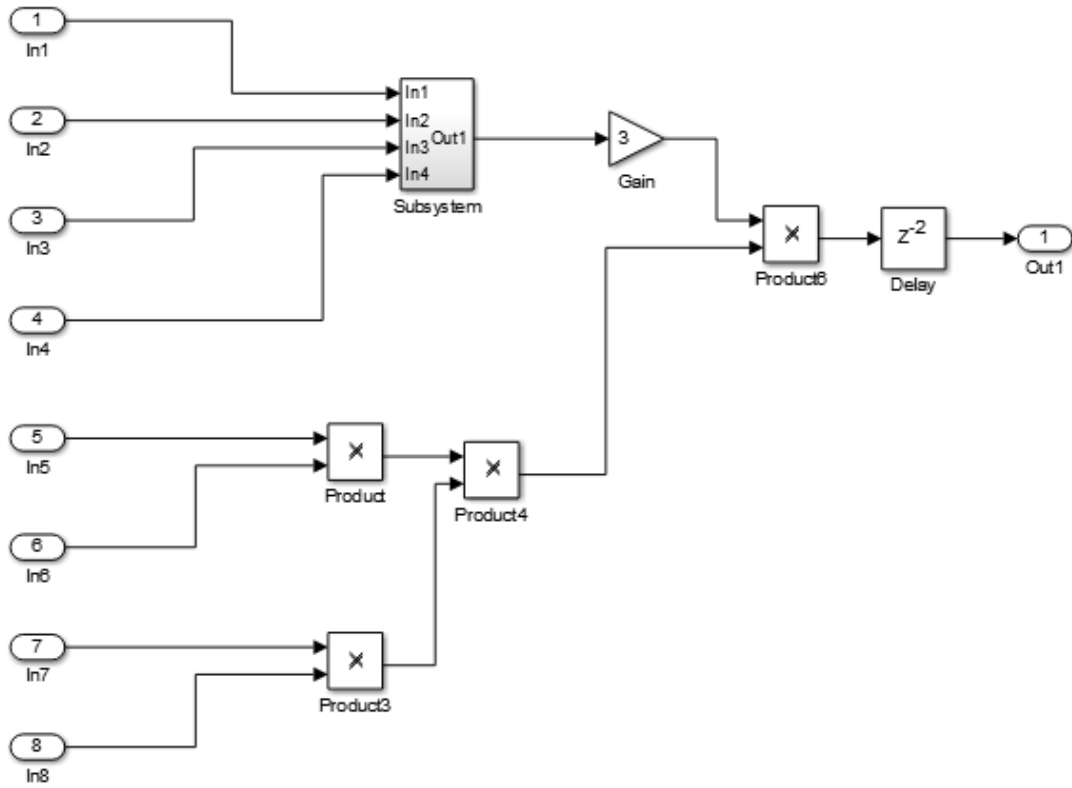
- QPSK Demodulator Baseband
- QPSK Modulator Baseband
- BPSK Demodulator Baseband
- BPSK Modulator Baseband
- PN Sequence Generator
- dsp_sigops/Repeat
- HDL Counter
- dsp_adpt3/LMS Filter
- dsp_srcs4/Sine Wave
- comm_nvcod2/Viterbi Decoder
- Triggered Subsystem
- Counter Limited
- Counter Free-Running
- Frame Conversion

What Is Hierarchical Distributed Pipelining?

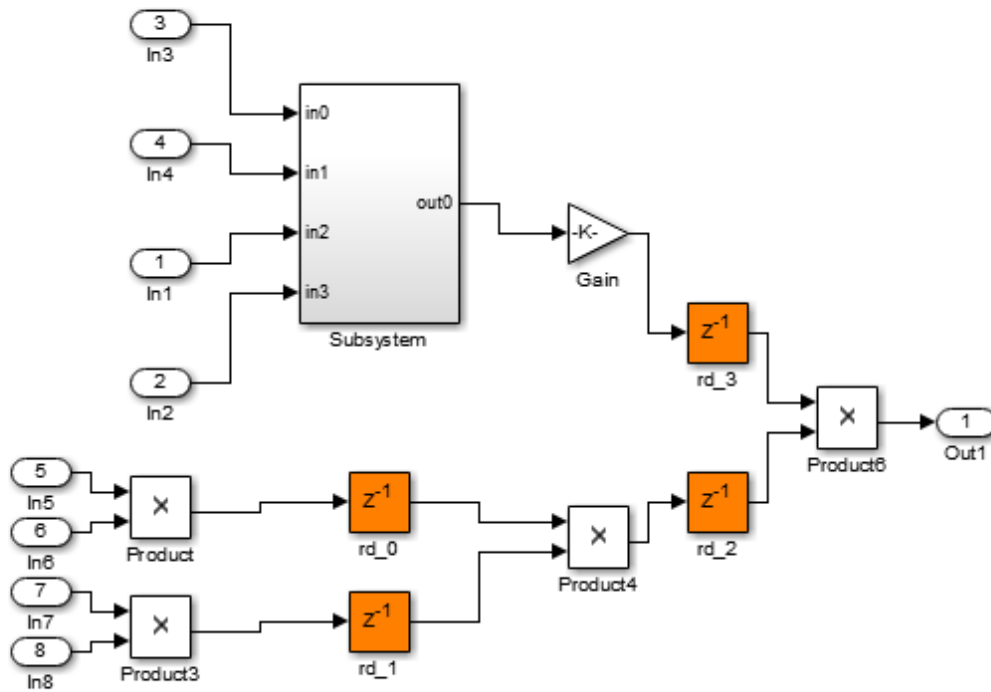
Hierarchical distributed pipelining extends the scope of distributed pipelining by moving delays across hierarchical boundaries within a subsystem while preserving subsystem hierarchy.

If a subsystem in the hierarchy does not have distributed pipelining enabled, HDL Coder does not move delays across that subsystem.

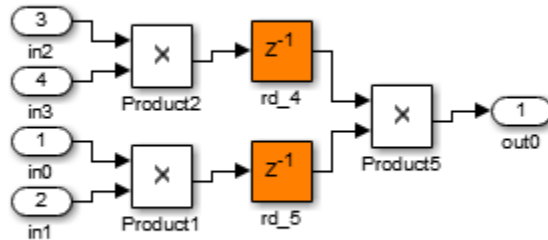
For example, the following model has one level of subsystem hierarchy:



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 distributed pipelining using the UI:

- 1 Right-click the DUT subsystem and select **HDL Code > HDL Coder Properties**.
- 2 In the **HDL Code Generation > Global Settings** pane, select the **Optimization** tab.
- 3 Select **Hierarchical distributed pipelining** and click **OK**.

To enable hierarchical distributed pipelining, on the command line, enter:

```
hdlset_param('modelname', 'HierarchicalDistPipelining', 'on')
```

To disable hierarchical distributed pipelining, on the command line, enter:

```
hdlset_param('modelName', 'HierarchicalDistPipelining', 'off')
```

Limitations of Hierarchical Distributed Pipelining

Hierarchical distributed pipelining must be disabled if your DUT subsystem contains a model reference.

Distributed Pipelining Workflow

For an example that shows how to use distributed pipelining to reduce your critical path, including delay insertion, see “Reduce Critical Path with 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 14-61

“When to Use Constrained Output Pipelining” on page 14-61

“Requirements for Constrained Output Pipelining” on page 14-61

“Specify Constrained Output Pipelining” on page 14-62

“Limitations of Constrained Output Pipelining” on page 14-62

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:

- 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, on 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)
```

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.


Reduce Critical Path with Distributed Pipelining

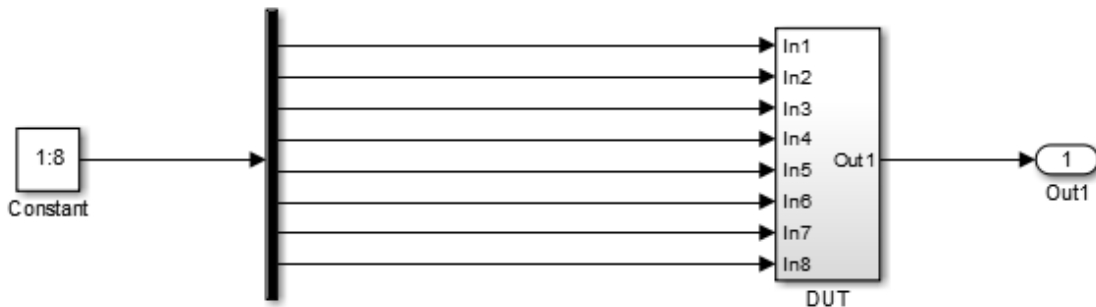
This example shows how to reduce your critical path using distributed pipelining, hierarchical distributed pipelining, output pipelining, and constrained output pipelining.

Before you begin, make sure that you have a synthesis tool set up. If you do not have a synthesis tool set up, you can follow this example, but you will not see maximum clock period results in the Result subpane. This example uses Xilinx ISE, but you can use another synthesis tool.

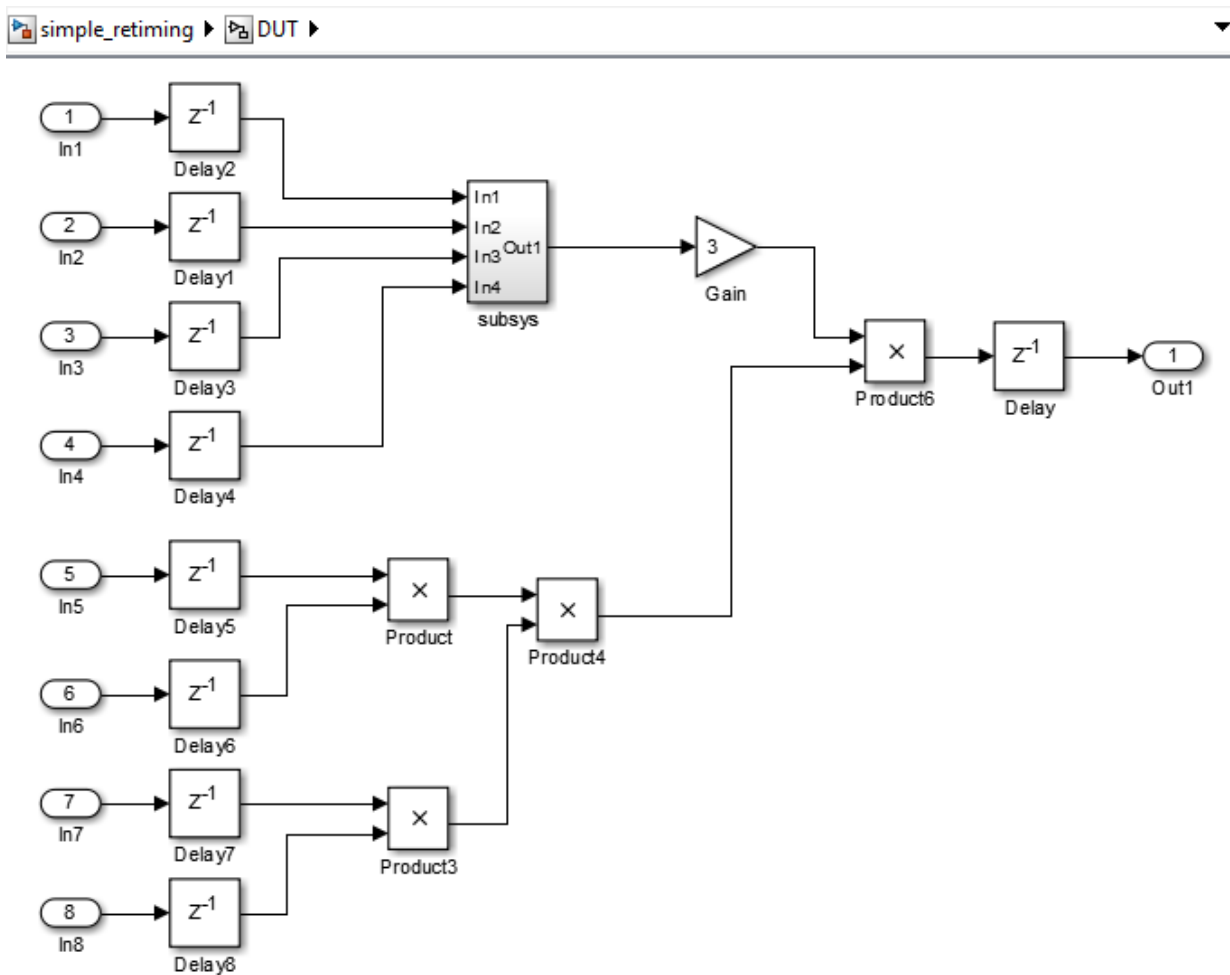
Open the `simple_retiming` model.

```
addpath(fullfile(docroot, 'toolbox', 'hdlcoder', 'examples'));
simple_retiming
```

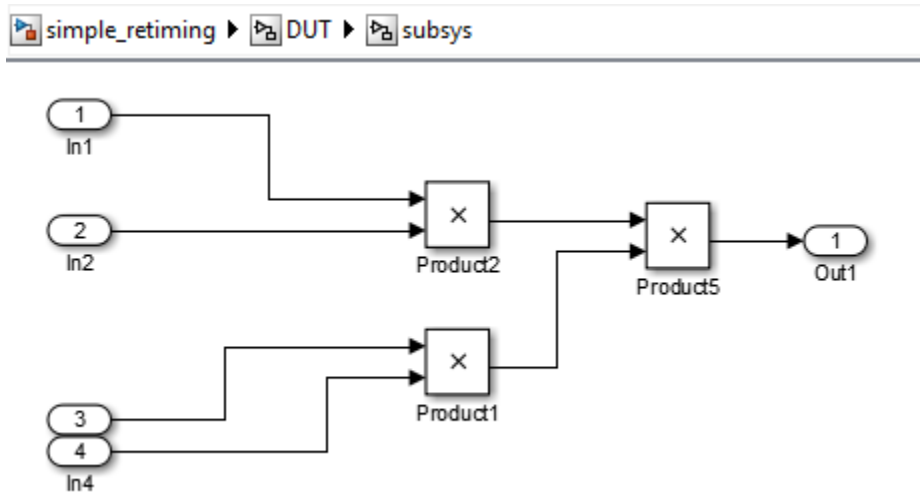
 simple_retiming ▶



The top-level subsystem is the design under test (DUT). The DUT subsystem contains one subsystem, `subsys`, and other blocks.



The **subsys** block in DUT contains a copy of the three **Product** blocks in the lower half of the diagram.



Right-click the DUT subsystem and select **HDL Code > HDL Workflow Advisor** to open the HDL Workflow Advisor.

In the **Set Target > Set Target Device and Synthesis Tool** pane, for **Synthesis tool**, select **Xilinx ISE**.

The **FPGA Synthesis and Analysis** task appears.

On the left, expand the **FPGA Synthesis and Analysis > Perform Synthesis and P/R** item.

Right-click **Perform Logic Synthesis** and select **Run to Selected Task**.

Minimum period: 36.042ns (Maximum Frequency: 27.745MHz)

When HDL Coder finishes, you see the minimum clock period near the bottom of the **Results** subpane.

Next, use distributed pipelining to improve your timing results.

In the model, right-click the DUT subsystem and select **HDL Code > HDL Block Properties**.

In the HDL Block Properties dialog box, for **DistributedPipelining**, select **on** to enable distributed pipelining, and click **OK**.

In the DUT subsystem, right-click the `subsys` block, select **HDL Code > HDL Block Properties**, and for **Distributed Pipelining**, select **on**. Click **OK**.

In the HDL Workflow Advisor, in the **HDL Code Generation > Set Code Generation Options > Set Basic Options** pane, enable **Generate optimization report**. Click **Apply**.

In the HDL Workflow Advisor, right-click **FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Logic Synthesis** and select **Run to Selected Task**.

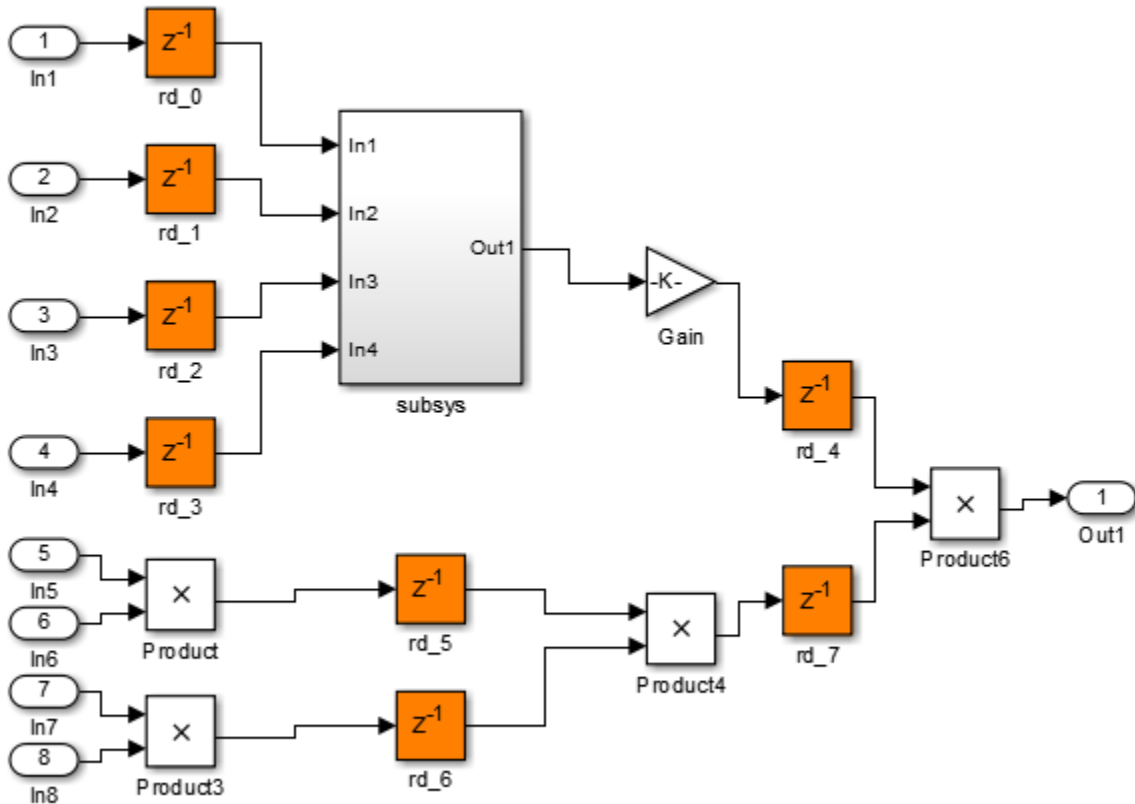
Minimum period: 22.025ns (Maximum Frequency: 45.403MHz)

The maximum clock frequency has increased.

In the Code Generation Report, click **Distributed Pipelining** to open the distributed pipelining report.

Under Generated Model, click the `gm_simple_retiming` link to open the generated model. You can see that HDL Coder redistributed the delay blocks.

gm_simple_retiming ▸ DUT ▸



In the **subsys** block, there are no delay blocks because hierarchical distributed pipelining is not enabled.

Next, use hierarchical distributed pipelining to further decrease the critical path.

In the HDL Workflow Advisor, in the **HDL Code Generation > Set Code Generation Options > Set Advanced Options > Optimization** tab, enable **Hierarchical distributed pipelining** and click **Apply**.

Right-click **FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Logic Synthesis** and select **Run to Selected Task** to rerun synthesis.

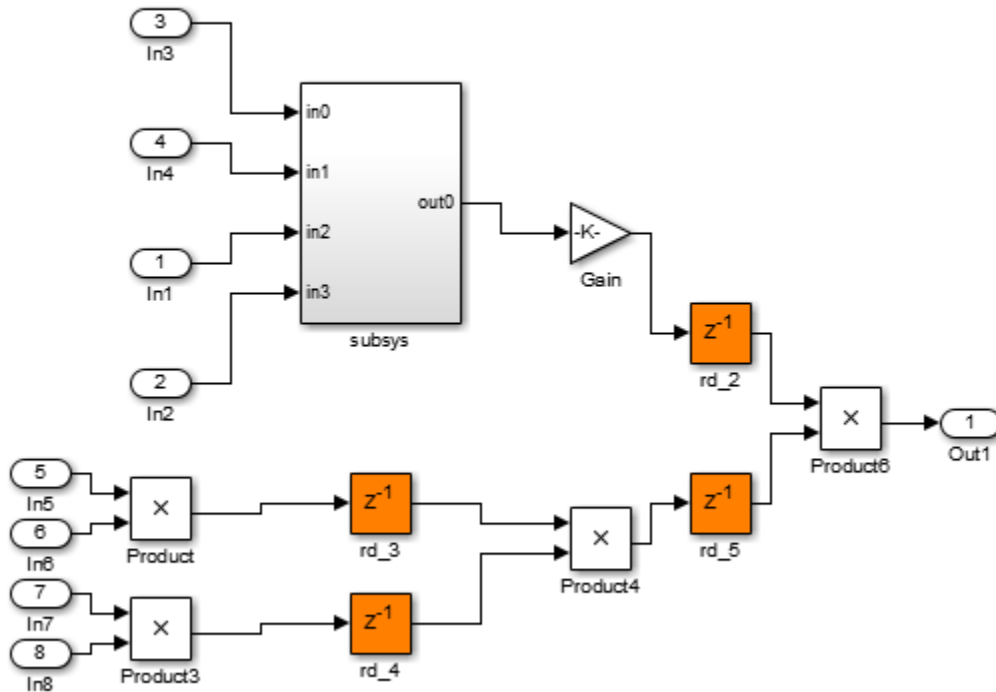
Minimum period: 17.263ns (Maximum Frequency: 57.928MHz)

The maximum clock frequency has increased.

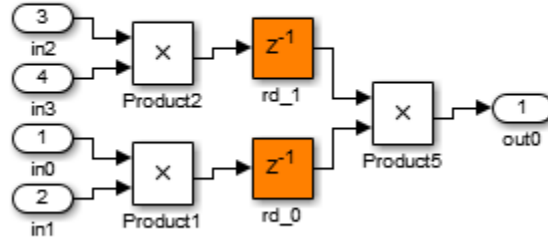
In the Code Generation Report, click **Distributed Pipelining** to open the distributed pipelining report. Click the `gm_simple_retiming` link to open the generated model.

HDL Coder has distributed delays in the DUT and within the subsystem block.

`gm_simple_retiming` ▶ `DUT` ▶



gm_simple_retiming ▸ DUT ▸ subsystem



Now, use constrained output pipelining to further reduce the critical path.

In the `simple_retiming` model, open the `subsys` block within the DUT subsystem.

Right-click the `Product5` block, and select **HDL Code > HDL Block Properties**.

For **OutputPipeline**, enter 1, and for **ConstrainedOutputPipeline**, enter 1. Click **OK**.

This adds a pipeline register and constrains it at the output of `Product5`.

In the HDL Workflow Advisor, right-click **Set Target > Set Target Device and Synthesis Tool** and select **Reset This Task**.

Right-click **FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Logic Synthesis** and select **Run to Selected Task** to rerun synthesis.

Minimum period: 10.266ns (Maximum Frequency: 97.410MHz)

The maximum clock frequency is now higher.

Clock-Rate Pipelining

In this section...

“Reasons for Clock-Rate Pipelining” on page 14-70

“How Clock-Rate Pipelining Works” on page 14-70

“Best Practices for Clock-Rate Pipelining” on page 14-71

“When to Disable Clock-Rate Pipelining” on page 14-71

“How to Specify Clock-Rate Pipelining” on page 14-71

“Limitations for Clock-Rate Pipelining” on page 14-71

For speed optimizations that insert pipeline registers, clock-rate pipelining identifies multi-cycle paths in your design and inserts pipeline registers at the clock rate instead of the data rate. Clock-rate pipelining can improve clock frequency at the cost of additional pipeline registers.

When the optimization is in a slow-rate region or multi-cycle path of the design, clock rate pipelining enables the software to perform optimizations without adding extra latency, or by adding minimal latency. It also enables optimizations such as pipelining and floating-point library mapping inside feedback loops.

Reasons for Clock-Rate Pipelining

HDL Coder restructures your design architecture to implement your Simulink design in hardware. For some block implementations and optimizations, the coder must introduce delay. For example, the following options introduce delay:

- Multi-cycle block implementations
- Input and output pipelining
- Floating-point library mapping
- Delay balancing

How Clock-Rate Pipelining Works

In blocks where there is a difference between the DUT sample time and the block sample time, or `Oversampling` is greater than 1, the coder can insert delays at the faster rate without adding latency, or by adding minimal latency, to the design.

The coder determines the maximum number of fast-rate delays it can insert based on the DUT-to-block sample time ratio and the oversampling factor:

Maximum number of clock-rate delays = $(block_rate \div DUT_base_rate) \times \text{Oversampling}$

For the following options, HDL Coder can introduce delays at the faster clock rate:

- Input and output pipelining
- Distributed pipelining
- Floating-point library mapping
- Multi-cycle HDL block implementations

Clock-rate pipelining does not affect the clock rate for delay blocks that are part of your original Simulink model.

Best Practices for Clock-Rate Pipelining

To increase the opportunities for clock-rate pipelining, enable hierarchy flattening.

When to Disable Clock-Rate Pipelining

If you want to reduce area at the cost of speed, disable clock-rate pipelining.

How to Specify Clock-Rate Pipelining

Clock-rate pipelining is enabled by default. You can disable clock-rate pipelining in one of the following ways:

- In the HDL Workflow Advisor, in the **HDL Code Generation > Set Code Generation Options > Set Advanced Options > Optimization** tab, select **Clock-rate pipelining**.
- In the Configuration Parameters dialog box, in **HDL Code Generation > Global Settings > Optimization** tab, select **Clock-rate pipelining**
- At the command line, use `makehdl` or `hdlset_param` to set the `ClockRatePipelining` property to `off`.

Limitations for Clock-Rate Pipelining

Clock-rate pipelining takes priority over resource sharing. For parts of the design where you specify resource sharing, HDL Coder inserts delays at the slower rate.

The following 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 Function
- MATLAB System
- Rate Transition
- Serializer1D
- Simple Dual Port RAM
- Single Port RAM
- Subsystem, if `FlattenHierarchy` is not enabled.
- Tapped Delay

Similarly, the coder 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 System Toolbox blocks
- DSP System Toolbox blocks, except for Delay

- Stateflow blocks

See Also

ClockRatePipelining | Oversampling

Find Estimated Critical Paths Without Synthesis Tools

In this section...
“When To Use Critical Path Estimation” on page 14-74
“How Critical Path Estimation Works” on page 14-74
“How to Use Critical Path Estimation” on page 14-76
“Characterized Blocks” on page 14-76
“Inaccuracy in Critical Path Estimation” on page 14-78

When To Use Critical Path Estimation

Use critical path estimation to quickly find the most likely timing critical paths in your design. With critical path estimation, you can quickly iterate through the process of finding and pipelining the critical path until your design is likely to meet timing.

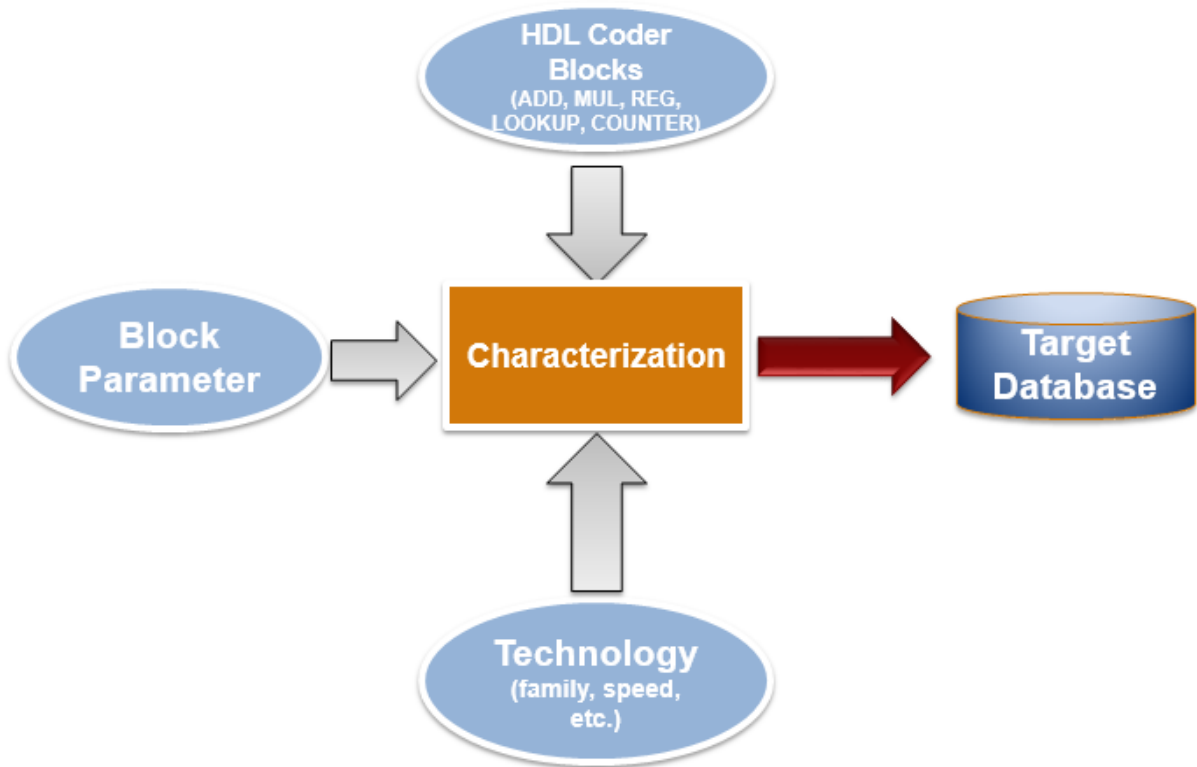
The advantage of estimating the critical path without running synthesis tools is faster iteration, but the disadvantage is inaccuracy. Critical path estimation is intended to speed up the design iteration process, and is not intended to provide you with accurate timing results.

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.

Target-Specific Timing Databases

HDL Coder has created timing databases by characterizing basic design components, such as Simulink blocks, block architectures, and subcomponents of those blocks, for specific target devices.



Target Devices Supported For Critical Path Estimation

HDL Coder has timing databases for the following target devices:

- Altera Cyclone V
- Xilinx Virtex[®]-7, speed grade -1
- Xilinx Zynq[®], speed grade -1

Analyzing the Design

The coder analyzes your design to decompose it into the blocks and subcomponents in the timing databases. If your design is composed of blocks or subcomponents in the timing databases, the coder 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, the estimated critical path may not match your actual critical path.

The timing numbers and estimated critical path are more accurate if your target hardware is one of the target devices supported for critical path estimation. If your target hardware is not a supported device, or is not in the same device family, it is likely you will still find the critical path, but the timing numbers may not be accurate.

How to Use Critical Path Estimation

- 1 Enable critical path estimation.
 - In the HDL Workflow Advisor, in the **HDL Code Generation > Set Code Generation Options > Set Basic Options**, select **Generate high-level timing critical path report**.
 - At the command line, set `CriticalPathEstimation` to on:

```
hdlset_param('modelName','CriticalPathEstimation','on');
```

- 2 Generate HDL code.

The critical path estimation message in the MATLAB command window or the HDL Workflow Advisor **Result** pane includes a link you can click to run the highlighting script.

- 3 Run the critical path highlighting script to see which blocks are on the critical path.
- 4 Break the critical path by adding pipeline registers.
- 5 Regenerate code.

Characterized Blocks

The following blocks have been characterized and are part of the timing database for each supported target device:

- Abs
- Add
- Bit Concat
- Bit Extract

- Bit Shift
- Bit Slice
- Bitwise Operator
- Bus Creator
- Bus Selector
- Buffer
- Complex to Real-Imag
- Compare To Constant
- Constant
- Convert 1-D to 2-D
- Counter Free-Running
- Counter Limited
- Data Type Conversion
- Delay
- Demux
- Deserializer1D
- Dual Port RAM
- Dual Rate Dual Port RAM
- Gain
- HDL Counter
- HDL Reciprocal
- Logical Operator
- Product
- Multiport Switch
- n-D Lookup Table
- Rate Transition
- Reciprocal Sqrt
- Relational Operator
- Reshape
- Selector

- Serializer1D
- Signal Conversion
- Signal Specification
- Simple Dual Port RAM
- Single Port RAM
- Switch
- Sqrt
- Unary Minus
- Unit Delay

Inaccuracy in Critical Path Estimation

Routing Delay

Critical path estimation tries to account for routing delay using an estimation factor. Without running place and route, however, it is difficult to accurately account for routing delay.

Uncharacterized Blocks

Blocks that are not in the list of characterized blocks, but are inferred to be combinational, are treated as zero-delay combinational blocks.

All other blocks are treated as registers.

Unsupported Target Devices

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.

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

- “Create and Use Code Generation Reports” on page 15-2
- “Resource Utilization Report” on page 15-4
- “Optimization Report” on page 15-6
- “Traceability Report” on page 15-9
- “Web View of Model in Code Generation Report” on page 15-26
- “Generate Code with Annotations or Comments” on page 15-29
- “Check Your Model for HDL Compatibility” on page 15-33
- “Create a Supported Blocks Library” on page 15-36
- “Trace Code Using the Mapping File” on page 15-38
- “Add or Remove the HDL Configuration Component” on page 15-41

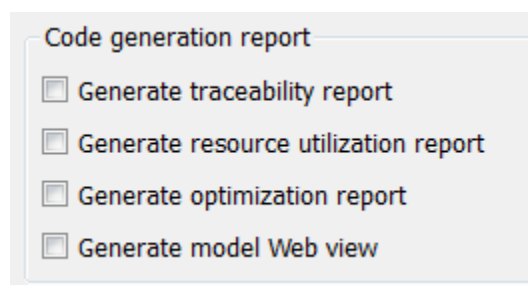
Create and Use Code Generation Reports

Information Included in Code Generation Reports

The HDL Coder software creates and displays an HDL Code Generation Report when you select one or more of the following options:

GUI option	makehdl Property
Generate traceability report	Traceability
Generate resource utilization report	ResourceReport
Generate optimization report	OptimizationReport
Generate model Web view	GenerateWebview

These options appear in the **Code generation report** panel of the **HDL Code Generation** pane of the Configuration Parameters dialog box:



The HDL Code Generation Report is an HTML report that includes a Summary and one or more of the following optional sections:

- Traceability Report
- Resource Utilization Report
- “Optimization Report” on page 15-6
- “Web View of Model in Code Generation Report” on page 15-26

HDL Code Generation Report Summary

All reports include a Summary section. 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 Generation Report

Back Forward Search...

Contents

- Summary**
- [Clock Summary](#)
- [Resource Utilization Report](#)
- [High-level Resource Report](#)
- [Target-specific Report](#)
- [Optimization Report](#)
- [Distributed Pipelining](#)
- [Streaming and Sharing](#)
- [Target Code Generation](#)
- [Traceability Report](#)

Generated Source Files

- [combo_pkg.vhd](#)
- [Gain_Subsystem.vhd](#)
- [Chart.vhd](#)
- [MATLAB_Function.vhd](#)
- [combo.vhd](#)

HDL Code Generation Report Summary for mcombo

Summary

Model	mcombo
Model version	1.146
HDL Coder version	3.1
HDL code generated on	2012-08-14 17:20:02
HDL code generated for	combo
Target Language	VHDL
Target Directory	hdlsrc

Non-default model properties

GenerateWebview	on
HDLSubsystem	mcombo/combo
OptimizationReport	on
ResourceReport	on
Traceability	on

Non-default block properties

No blocks found with non-default properties.

Simulink Root

- mcombo
 - combo
 - Chart
 - Gain_Subsystem
 - MATLAB Function

combo

OK Help

Resource Utilization Report

When you select **Generate resource utilization report**, HDL Coder adds a Resource Utilization Report section. The Resource Utilization Report summarizes multipliers, adders/subtractors, and registers consumed by the device under test (DUT). It also includes a detailed report on resources used by each subsystem. The detailed report includes (wherever possible) links back to corresponding blocks in your model.

The Resource Utilization Report is useful for analysis of the effects of optimizations, such as resource sharing and streaming. A typical Resource Utilization Report looks like this:

Resource Utilization Report for dct8_fixed

Summary

Multipliers	13
Adders/Subtractors	29
Registers	0
RAMs	0
Multiplexers	0

Detailed Report

[Expand all] [Collapse all]

Report for Subsystem: [OneD_DCT8](#)

Multipliers (13)

[+] 8x8-bit Multiply : 5
[+] 16x16-bit Multiply : 4
[+] 32x32-bit Multiply : 4

Adders/Subtractors (29)

[+] 8x8-bit Adder : 8
[+] 17x17-bit Adder : 3
[+] 33x33-bit Adder : 2
[+] 8x8-bit Subtractor : 11
[+] 17x17-bit Subtractor : 3
[+] 33x33-bit Subtractor : 2

Optimization Report

When you select **Generate optimization report**, HDL Coder adds an Optimization Report section, with two subsections:

- **Distributed Pipelining:** this subsection shows details of subsystem-level distributed pipelining if any subsystems have the `DistributedPipelining` option enabled. Details include comparative listings of registers and flip-flops before and after applying the distributed pipelining transform.
- **Streaming and Sharing:** this subsection shows both summary and detailed information about the subsystems for which sharing or streaming is requested.

A typical Distributed Pipelining Report looks something like this:

Distributed Pipelining Report for dct8_fixed

Summary

HDL Code Generation Parameter: HierarchicalDistPipelining: 'off'

Subsystems with 'DistributedPipelining' set 'on':

Subsystem	InputPipeline	OutputPipeline
OneD_DCT8	2	2

Detailed Report

Subsystem: [OneD_DCT8](#)

Implementation Parameters: DistributedPipelining: 'on'; InputPipeline: 2; OutputPipeline: 2

Status: Distributed pipelining successful.

Before Distributed Pipelining : 32 registers (640 flip-flops)

Registers	Count
32-bit	16
8-bit	16

After Distributed Pipelining : 32 registers (552 flip-flops)

Registers	Count
32-bit	10
8-bit	15
16-bit	7

Generated Model

Generated model after Distributed Pipelining: [gm_dct8_fixed0](#)

Hierarchical Distributed Pipelining in the Optimization Report

If `HierarchicalDistPipelining` is on, the Optimization Report uses colored sections to distinguish between different regions where HDL Coder applies hierarchical distributed pipelining:

Detailed Report

Hierarchical distributed pipelining region 1:

Status for hierarchical distributed pipelining starting at [Hlp](#) : Distributed pipelining successful.

Details within this hierarchical distributed pipelining region:

Subsystem: [Hlp](#)

Implementation Parameters: DistributedPipelining: 'on'; InputPipeline: 0; OutputPipeline: 0

Before Distributed Pipelining : 2 registers (32 flip-flops)

Registers	Count
16-bit	2

After Distributed Pipelining : 1 registers (16 flip-flops)

Registers	Count
16-bit	1

Subsystem: [Section1](#)

Implementation Parameters: DistributedPipelining: 'on'; InputPipeline: 0; OutputPipeline: 0

Before Distributed Pipelining : 0 registers (0 flip-flops)

After Distributed Pipelining : 9 registers (288 flip-flops)

Registers	Count
32-bit	9

Related Examples

- “Find Feedback Loops”

Traceability Report

In this section...

“Traceability Report Overview” on page 15-9

“Generating a Traceability Report from Configuration Parameters” on page 15-13

“Generating a Traceability Report from the Command Line” on page 15-16

“Keeping the Report Current” on page 15-18

“Tracing from Code to Model” on page 15-18

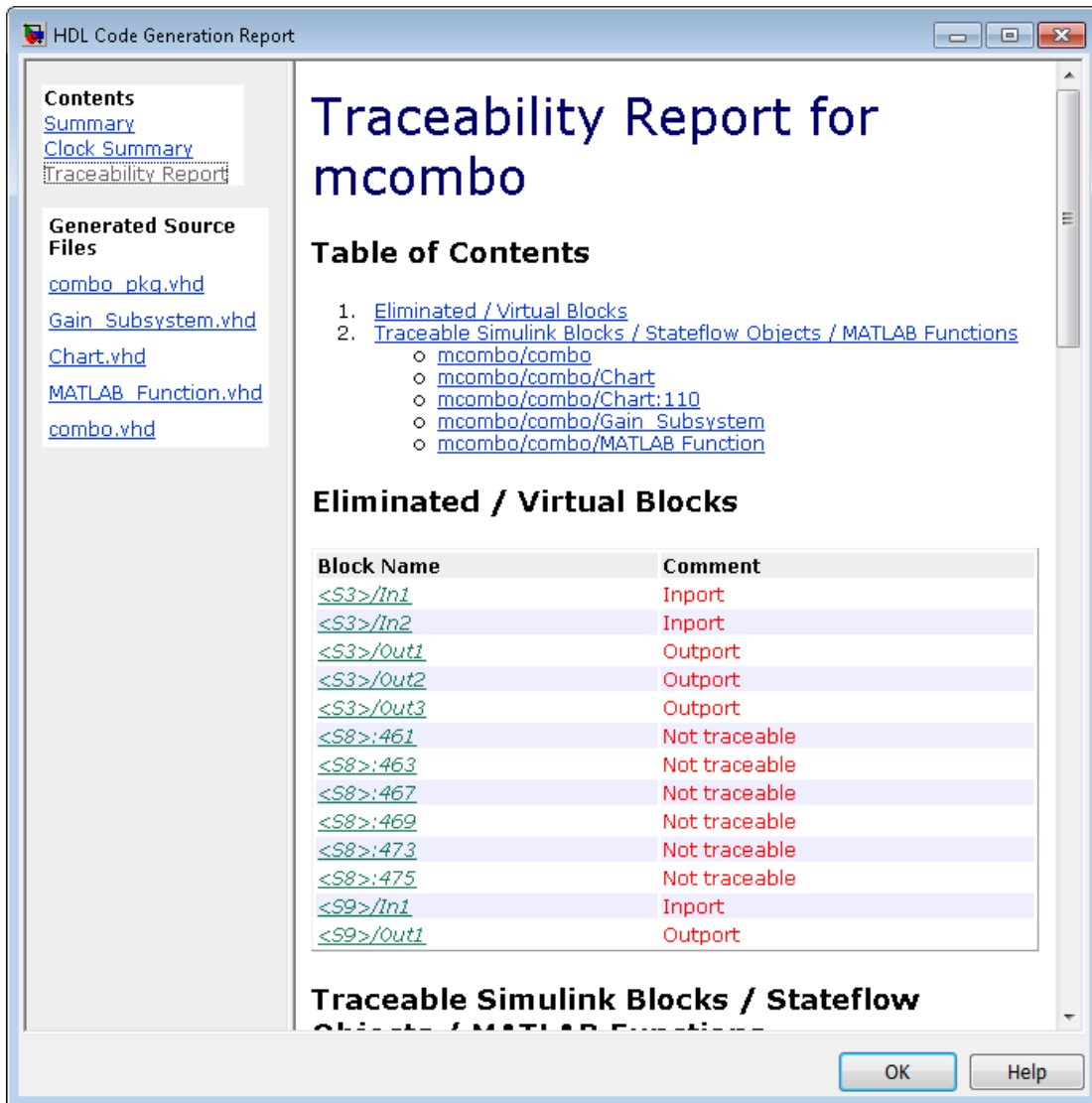
“Tracing from Model to Code” on page 15-20

“Mapping Model Elements to Code Using the Traceability Report” on page 15-23

“Traceability Report Limitations” on page 15-24

Traceability Report Overview

Even a relatively small model can generate hundreds of lines of HDL code. The HDL Coder software provides the traceability report section to help you navigate more easily between the generated code and your source model. When you enable traceability, HDL Coder creates and displays an HTML code generation report. You can generate reports from the Configuration Parameters dialog box or the command line. A typical traceability report looks something like this:



The traceability report has several subsections:

- The **Traceability Report** lists **Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions**, providing a complete mapping between model elements and

code. The **Eliminated / Virtual Blocks** section of the report accounts for blocks that are untraceable.

- The **Generated Source Files** table contains hyperlinks that let you view generated HDL code in a MATLAB web browser window. This view of the code includes hyperlinks that let you view the blocks or subsystems from which the code was generated. You can click the names of source code files generated from your model to view their contents in a MATLAB web browser window. The report supports two types of linkage between the model and generated code:
 - *Code-to-model* hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click the hyperlinks to view the relevant blocks or subsystems in a Simulink model window.
 - *Model-to-code* linkage lets you view the generated code for any block in the model. To highlight a block's generated code in the HTML report, right-click the block and select **HDL Code > Navigate to Code**.

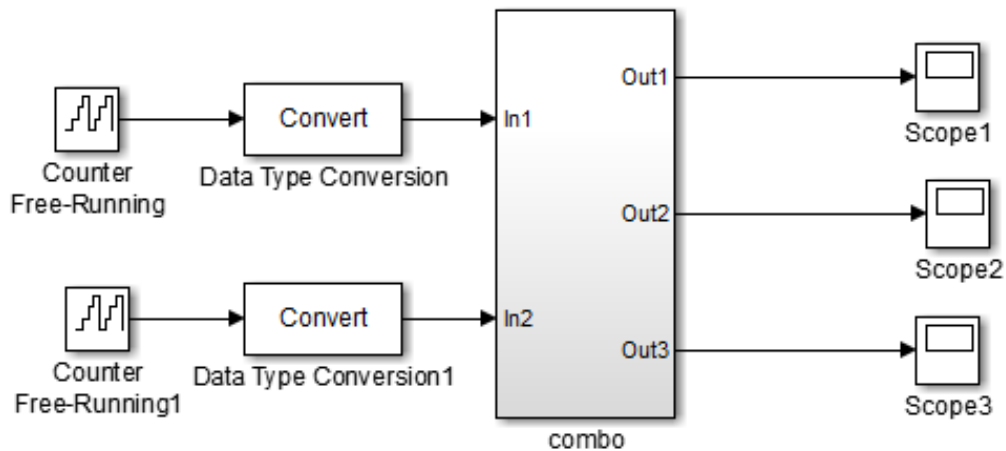
Note: If your model includes blocks that have requirements comments, you can also render the comments as hyperlinked comments within the HTML code generation report. See “Requirements Comments and Hyperlinks” on page 15-29 for more information.

In the following sections, the `mcombo` example model illustrates stages in the workflow for generating code generation reports from the Configuration Parameters dialog box and from the command line.

To open the model, enter:

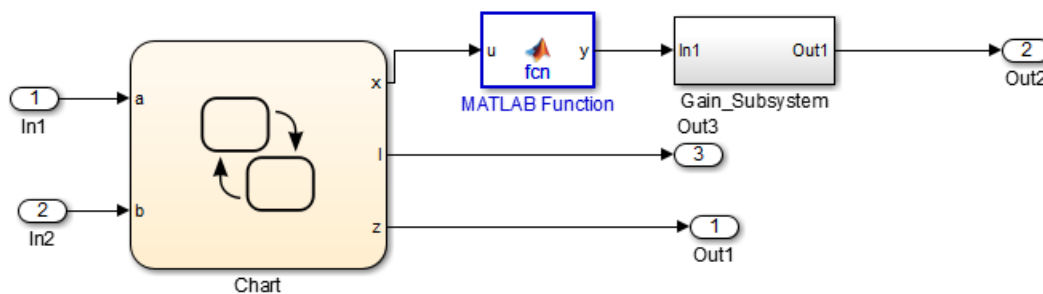
```
mcombo
```

The root-level `mcombo` model appears as follows:



Copyright 2008-2010 The MathWorks, Inc.

HDL Coder supports report generation for models, subsystems, blocks, Stateflow charts, and MATLAB Function blocks. This example uses the `combo` subsystem, shown in the following figure. The `combo` subsystem includes a Stateflow chart, a MATLAB Function block, and a subsystem.



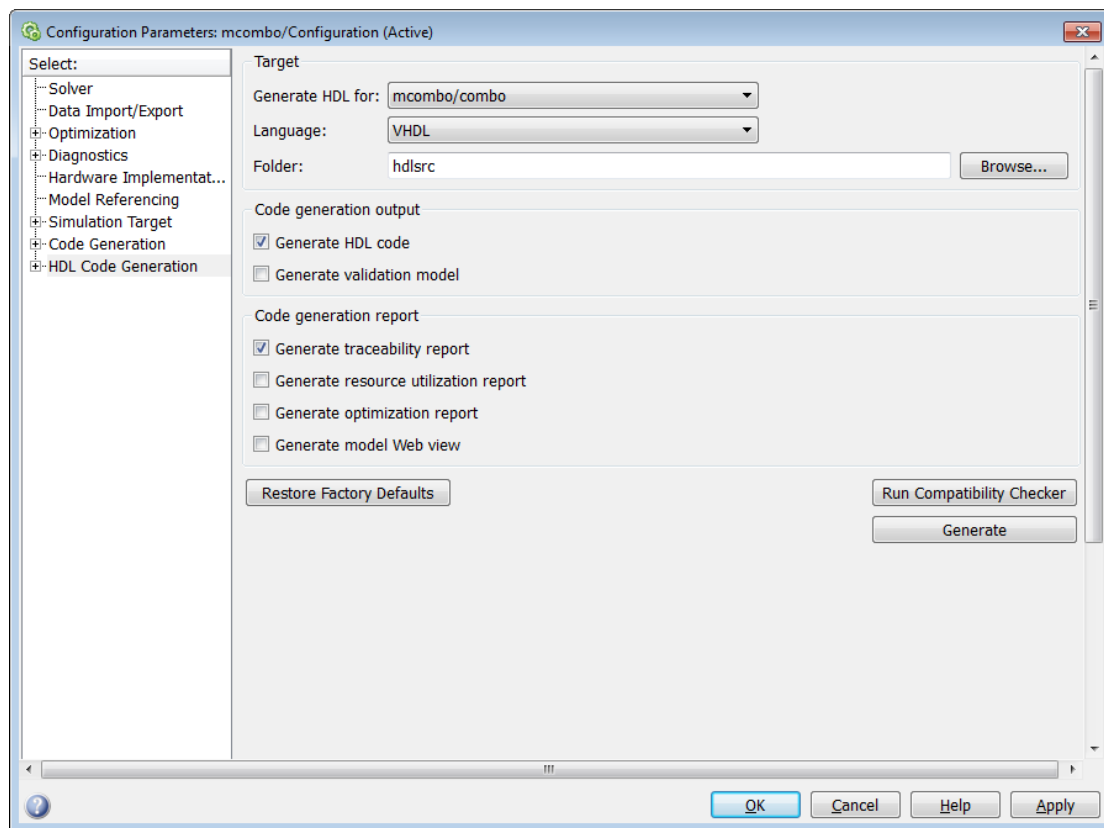
Generating a Traceability Report from Configuration Parameters

To generate a HDL Coder code generation report from the Configuration Parameters dialog box:

- 1 Verify that the model is open.
- 2 Open the Configuration Parameters dialog box and navigate to the **HDL Code Generation** pane.
- 3 To enable report generation, select **Generate traceability report**.

If your model includes blocks that have requirements comments, you can also select **Include requirements in block comments** in the **HDL Code Generation > Global Settings > Coding style** pane to render the comments as hyperlinked comments in the HTML code generation report. See “Requirements Comments and Hyperlinks” on page 15-29 for more information.

- 4 Verify that **Generate HDL for** specifies the correct DUT for HDL code generation. You can generate reports for the root-level model or for subsystems, blocks, Stateflow charts, or MATLAB Function blocks.
- 5 Click **Apply**.



- 6 Click **Generate** to initiate code and report generation.

When you select **Generate traceability report**, HDL Coder generates HTML report files as part of the code generation process. Report file generation is the final phase of that process. As code generation proceeds, the coder displays progress messages. The process completes with messages similar to the following:

```
### Generating HTML files for traceability in slprj\hdl\mcombo\html directory ...  
### HDL Code Generation Complete.
```

When code generation is complete, the HTML report appears in a new window:

Contents
[Summary](#)
[Clock Summary](#)
[Traceability Report](#)

Generated Source Files
[combo_pkg.vhd](#)
[Gain_Subsystem.vhd](#)
[Chart.vhd](#)
[MATLAB_Function.vhd](#)
[combo.vhd](#)

Traceability Report for mcombo

Table of Contents

- [Eliminated / Virtual Blocks](#)
- [Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions](#)
 - [mcombo/combo](#)
 - [mcombo/combo/Chart](#)
 - [mcombo/combo/Chart:110](#)
 - [mcombo/combo/Gain_Subsystem](#)
 - [mcombo/combo/MATLAB_Function](#)

Eliminated / Virtual Blocks

Block Name	Comment
<S3>/In1	Inport
<S3>/In2	Inport
<S3>/Out1	Output
<S3>/Out2	Output
<S3>/Out3	Output
<S8>:461	Not traceable
<S8>:463	Not traceable
<S8>:467	Not traceable
<S8>:469	Not traceable
<S8>:473	Not traceable
<S8>:475	Not traceable
<S9>/In1	Inport
<S9>/Out1	Output

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

OK Help

- 7 To view the different report sections or view the generated code files, click the hyperlinks in the **Contents** pane of the report window.

Tip HDL Coder writes the code generation report files to a folder in the `hdlsrc\html\` folder of the build folder. The top-level HTML report file is named `system_codegen_rpt.html`, where `system` is the name of the model, subsystem, or other component selected for code generation. However, because the coder automatically opens this file after report generation, you do not need to access the HTML files directly. Instead, navigate the report using the links in the top-level window.

For more information on using the report you generate for tracing, see:

- “Tracing from Code to Model” on page 15-18
- “Tracing from Model to Code” on page 15-20
- “Mapping Model Elements to Code Using the Traceability Report” on page 15-23

Generating a Traceability Report from the Command Line

To generate a HDL Coder code generation report from the command line:

- 1 Open your model by entering:

```
open_system('model_name');
```

- 2 Specify the desired device under test (DUT) for code generation by entering:

```
gcb = 'path_to_DUT';
```

You can generate reports for the root-level model or for subsystems, blocks, Stateflow charts, or MATLAB Function blocks. If you do not specify a subsystem, block, Stateflow chart, or MATLAB Function block, HDL Coder generates a report for the top-level model.

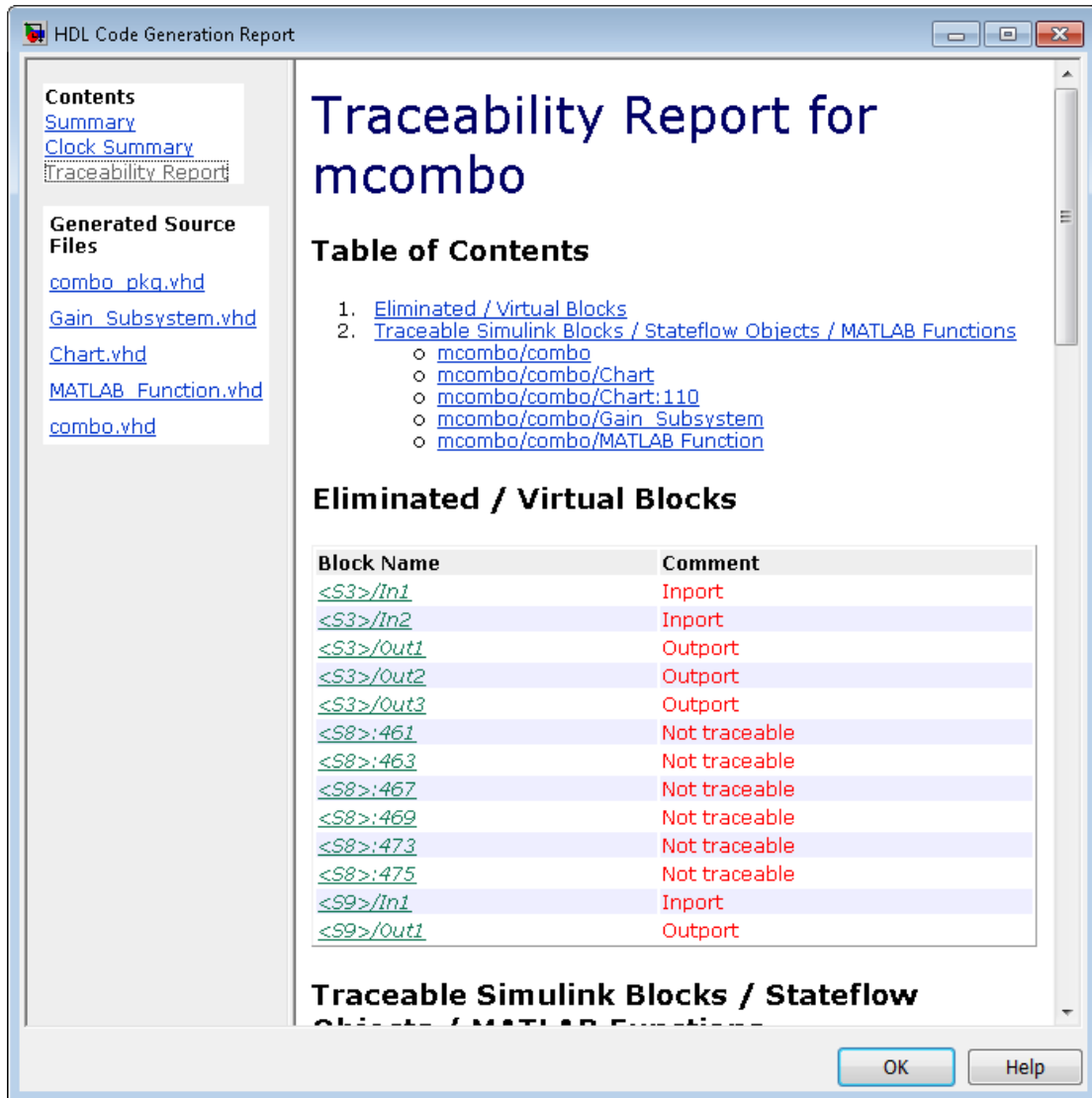
- 3 Enable the `makehdl` property `Traceability` by entering:

```
makehdl(gcb, 'Traceability', 'on');
```

When you enable traceability, HDL Coder generates HTML report files as part of the code generation process. Report file generation is the final phase of that process. As code generation proceeds, the coder displays progress messages. The process completes with messages similar to the following:

```
### Generating HTML files for traceability in slprj\hdl\mcombo\html directory ...  
### HDL Code Generation Complete.
```

When code generation is complete, the HTML report appears in a new window:



- To view the different report sections or view the generated code files, click the hyperlinks in the **Contents** pane of the report window.

Tip HDL Coder writes the code generation report files to a folder in the `hdlsrc\html\` folder of the build folder. The top-level HTML report file is named `system_codegen_rpt.html`, where *system* is the name of the model, subsystem, or other component selected for code generation. However, because the coder automatically opens this file after report generation, you do not need to access the HTML files directly. Instead, navigate the report using the links in the top-level window.

For more information on using the report you generate for tracing, see:

- “Tracing from Code to Model” on page 15-18
- “Tracing from Model to Code” on page 15-20
- “Mapping Model Elements to Code Using the Traceability Report” on page 15-23

Keeping the Report Current

If you generate a code generation report for a model, and subsequently make changes to the model, the report might become invalid.

To keep your code generation report current, you should regenerate HDL code and the report after modifying the source model.

If you close and then reopen a model without making changes, the report remains valid.

Tracing from Code to Model

To trace from generated code to your model:

- 1 Generate code and open an HTML report for the desired DUT (see “Generating a Traceability Report from Configuration Parameters” on page 15-13 or “Generating a Traceability Report from the Command Line” on page 15-16).
- 2 In the left pane of the HTML report window, click the desired file name in the **Generated Source Files** table to view a source code file.

The following figure shows a view of the source file `Gain_Subsystem.vhd`.

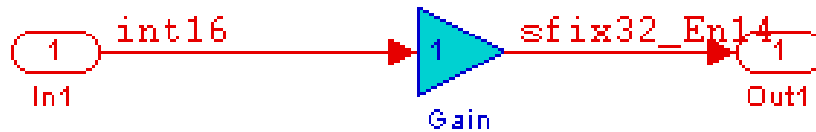

```

13  -- Module: Gain_Subsystem
14  -- Source Path: mcombo/combo/Gain_Subsystem
15  -- Hierarchy Level: 1
16  --
17  -----
18  LIBRARY IEEE;
19  USE IEEE.std_logic_1164.ALL;
20  USE IEEE.numeric_std.ALL;
21
22  ENTITY Gain_Subsystem IS
23      PORT( In1                : IN    std_
24            Out1              : OUT   std_
25            );
26  END Gain_Subsystem;
27
28
29  ARCHITECTURE rtl OF Gain_Subsystem IS
30
31      -- Signals
32      SIGNAL In1_signed        : signed(15 DO
33      SIGNAL Gain_out1        : signed(31 DO
34
35  BEGIN
36      In1_signed <= signed(In1);
37
38      -- <S9>/Gain
39      Gain_out1 <= resize(In1_signed & '0' & '0' & '0' & '0'
40
41      Out1 <= std_logic_vector(Gain_out1);
42
43  END rtl;
44
45

```

- 3 In the HTML report window, click a link to highlight the corresponding source block.

For example, in the HTML report shown in the previous figure, you could click the hyperlink for the Gain block (highlighted) to view that block in the model. Clicking the hyperlink locates and displays the corresponding block in the Simulink model window.



Tracing from Model to Code

Model-to-code traceability lets you select a component at any level of the model, and view the code references to that component in the HTML code generation report. You can select the following objects for tracing:

- Subsystem
- Simulink block
- MATLAB Function block
- Stateflow chart, or the following elements of a Stateflow chart:
 - State
 - Transition
 - Truth table
 - MATLAB function inside a chart

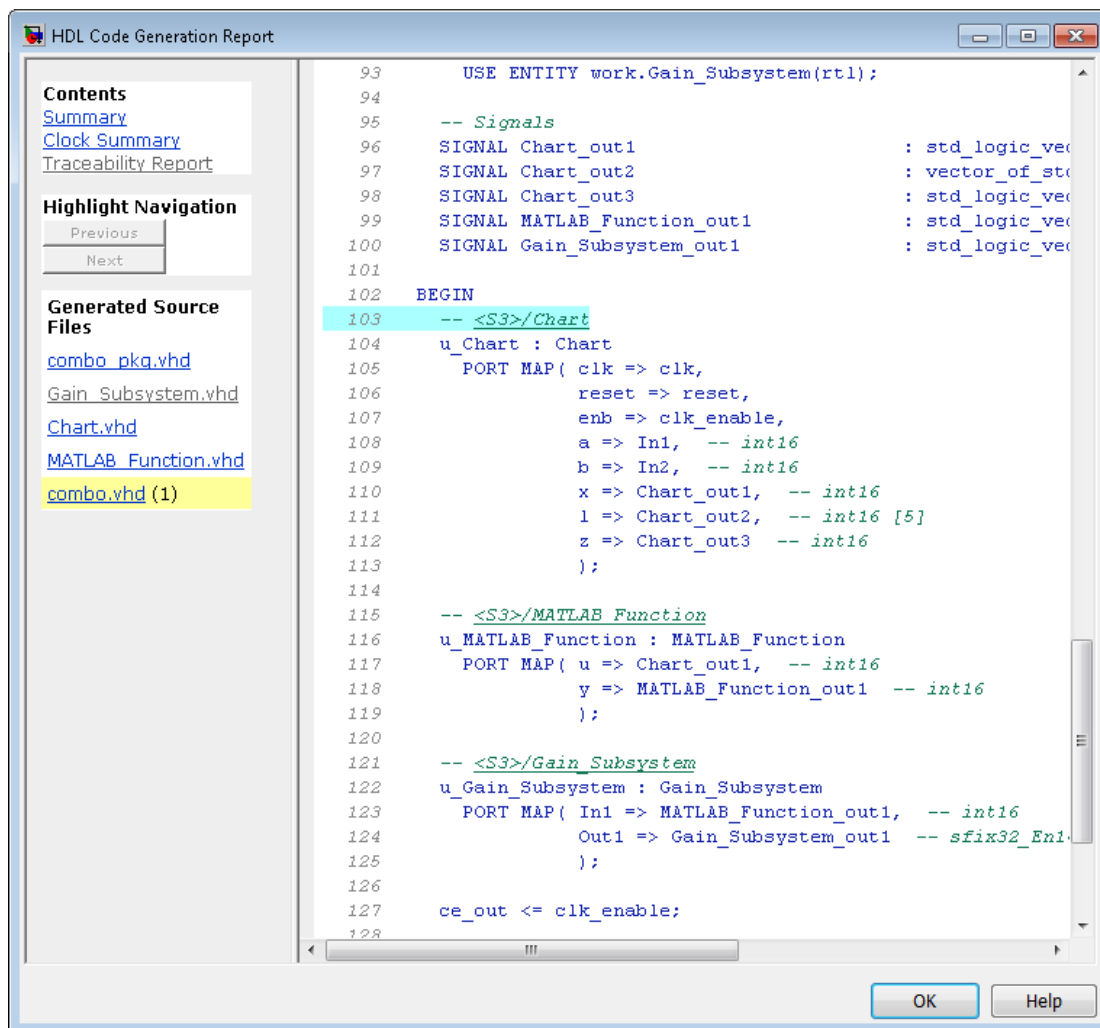
To trace a model component:

- 1 Generate code and open an HTML report for the desired DUT (see “Generating a Traceability Report from Configuration Parameters” on page 15-13 or “Generating a Traceability Report from the Command Line” on page 15-16).

Tip If you have not generated code for the model, HDL Coder disables the **HDL Code > Navigate to Code** menu item.

- 2** In the model window, right-click the component and select **HDL Code > Navigate to Code**.
- 3** Selecting **Navigate to Code** activates the HTML code generation report.

The following figure shows the result of tracing the Stateflow chart within the combo subsystem.



In the right pane of the report, the highlighted tag `<S3>/Chart` indicates the beginning of the code generated code for the chart.

In the left pane of the report, the total number of highlighted lines of code (in this case, 1) appears next to the source file name `combo.vhd`.

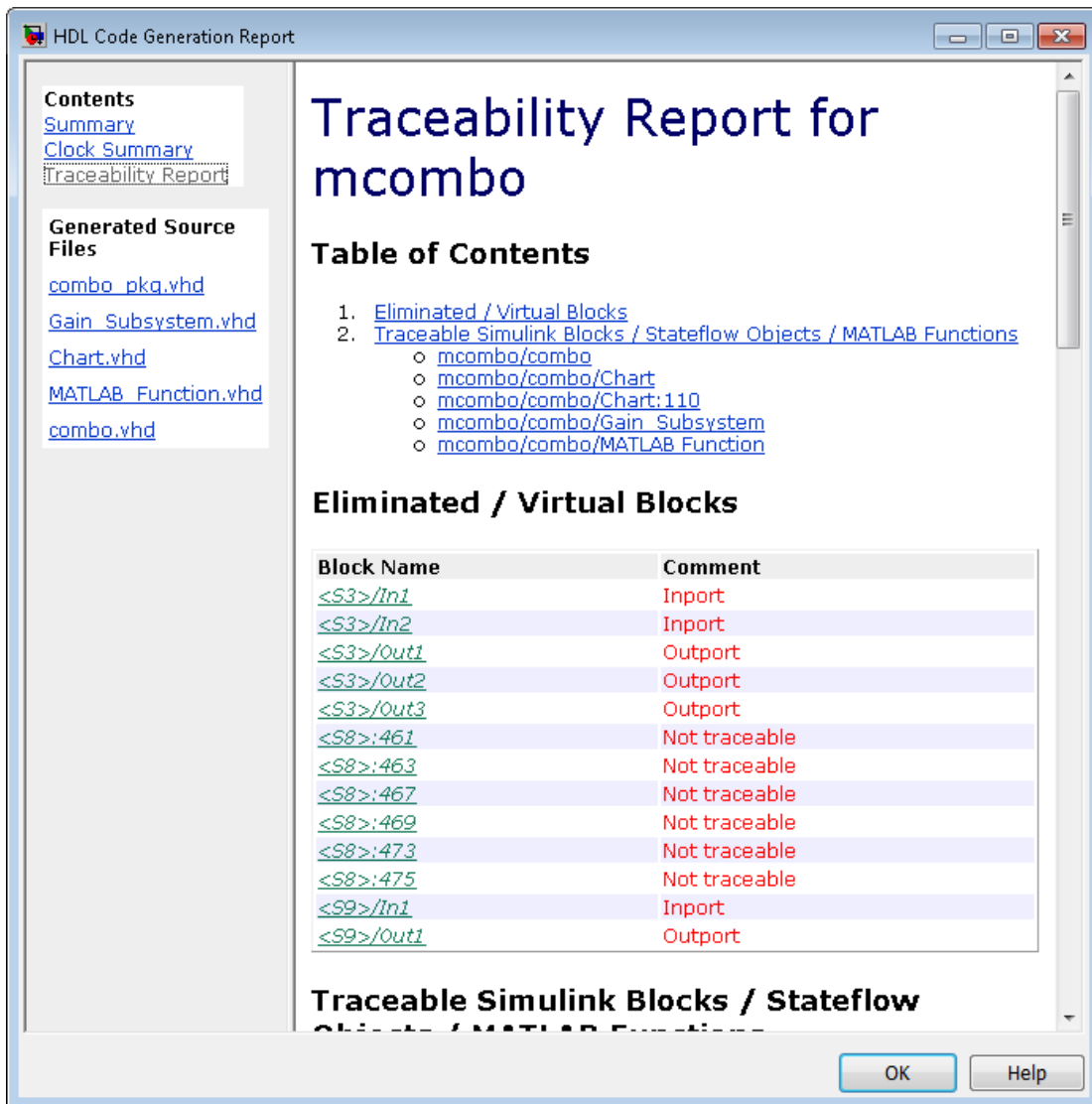
The left pane of the report also contains **Previous** and **Next** buttons. These buttons help you navigate through multiple instances of code generated for a selected component. In this example, there is only one instance, so the buttons are disabled.

Mapping Model Elements to Code Using the Traceability Report

The **Traceability Report** section of the report provides a complete mapping between model elements and code. The **Traceability Report** summarizes:

- **Eliminated / virtual blocks:** accounts for blocks that are untraceable because they are not included in generated code
- Traceable model elements, including:
 - **Traceable Simulink blocks**
 - **Traceable Stateflow objects**
 - **Traceable MATLAB functions**

The following figure shows the beginning of a traceability report generated for the `combo` subsystem of the `mcombo` model.



Traceability Report Limitations

The following limitations apply to HDL Coder HTML code generation reports:

- 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.
- Some types of subsystems are not traceable from model to code at the subsystem block level:
 - Virtual subsystems
 - Masked subsystems
 - Nonvirtual subsystems for which code has been optimized away

If you cannot trace a subsystem at the subsystem level, you might be able to trace individual blocks within the subsystem.

Web View of Model in Code Generation Report

In this section...

“About Model Web View” on page 15-26

“Generate HTML Code Generation Report with Model Web View” on page 15-26

“Model Web View Limitations” on page 15-28

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 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.

The screenshot shows the 'Code Generation Report' web browser interface. The window title is 'Code Generation Report'. On the left is a navigation pane with 'Contents' (Summary, Clock Summary) and 'Generated Source Files' (combo_pkg.vhd, Gain_Subsystem.vhd, Chart.vhd, MATLAB_Function.vhd, combo.vhd). The main area shows Verilog code for 'Chart' and 'MATLAB_Function' blocks. Below the code is a block diagram of the 'combo' model, where the 'Chart' block is highlighted in red, corresponding to the selected source file.

- 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” in the Simulink Report Generator documentation.

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”.
- For a subsystem build, the traceability hyperlinks of the root level inport and output blocks are disabled.
- “Traceability Limitations” 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 15-29

“Text Comments” on page 15-29

“Requirements Comments and Hyperlinks” on page 15-29

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 “Annotations” in the Simulink documentation for general information on annotations.

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 in the Simulink documentation 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.

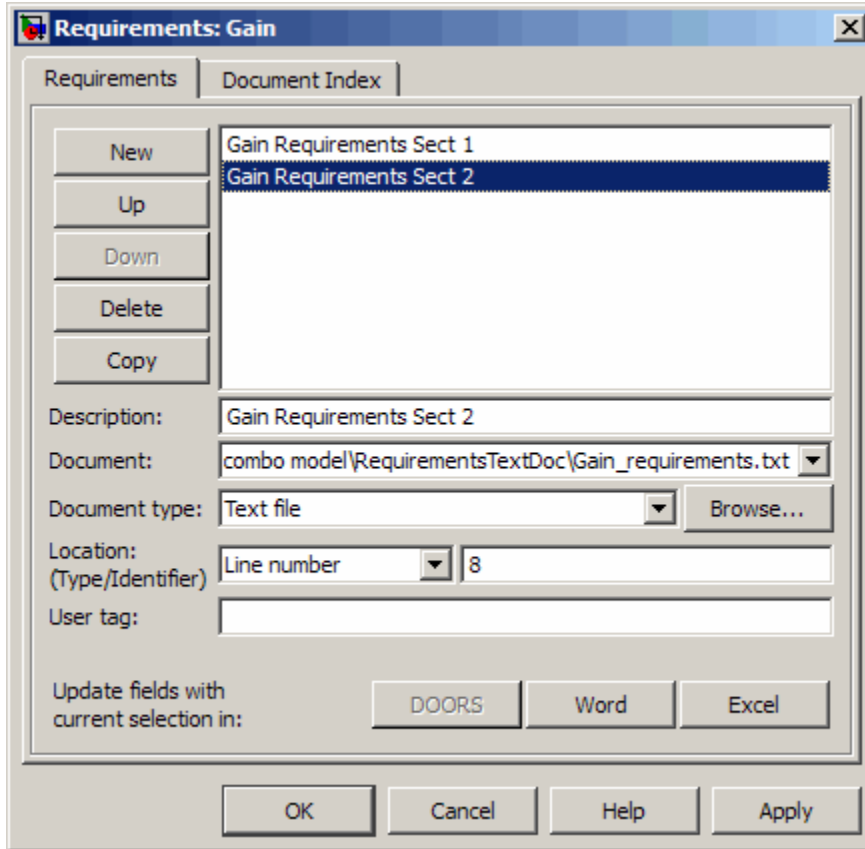
```
36 BEGIN
37     In1_signed <= signed(In1);
38
39     --
40     -- Block requirements for <S10>/Gain
41     -- 1. Gain Requirements Sect 1
42     -- 2. Gain Requirements Sect 2
43     Gain_gainparam <= to_signed(16384, 16);
44
45     Gain_out1 <= resize(In1_signed(15 DOWNTO 0) & '0'
46
47
48     Out1 <= 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

The HDL compatibility checker lets you check whether a subsystem or model is compatible with HDL code generation. You can run the compatibility checker from the command line or from the GUI.

To run the compatibility checker from the command line, use the `checkhdl` function. The syntax of the function is

```
checkhdl('system')
```

where *system* is the device under test (DUT), typically a subsystem within the current model.

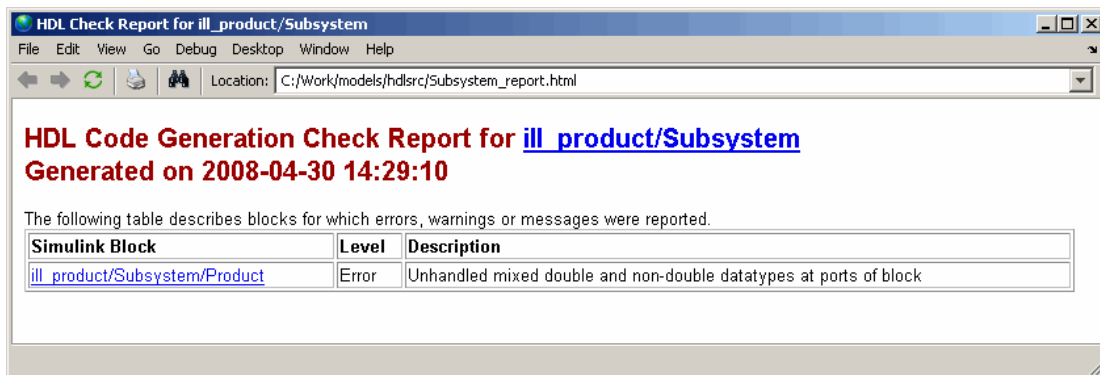
To run the compatibility checker from the GUI:

- 1 Open the Configuration Parameters dialog box or the Model Explorer. Select the **HDL Code Generation** pane.
- 2 Select the subsystem you want to check from the **Generate HDL for** list.
- 3 Click the **Run Compatibility Checker** button.

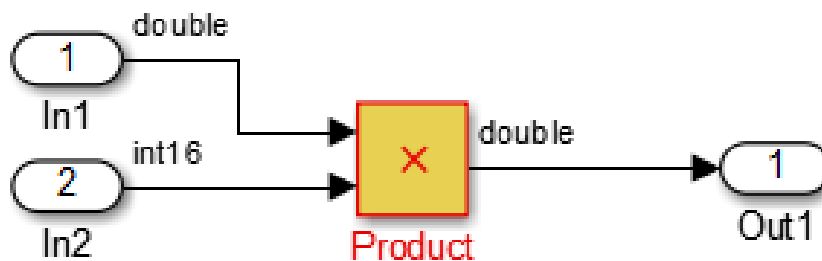
The HDL compatibility checker examines the specified system for compatibility problems, such as use of unsupported blocks, illegal data type usage, etc. The HDL compatibility checker generates an HDL Code Generation Check Report, which is stored in the target folder. The report file naming convention is `system_report.html`, where *system* is the name of the subsystem or model 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 is hyperlinked to the block or subsystem that caused the problem. When you click the hyperlink, the block of interest highlights and displays (provided that the model referenced by the report is open).

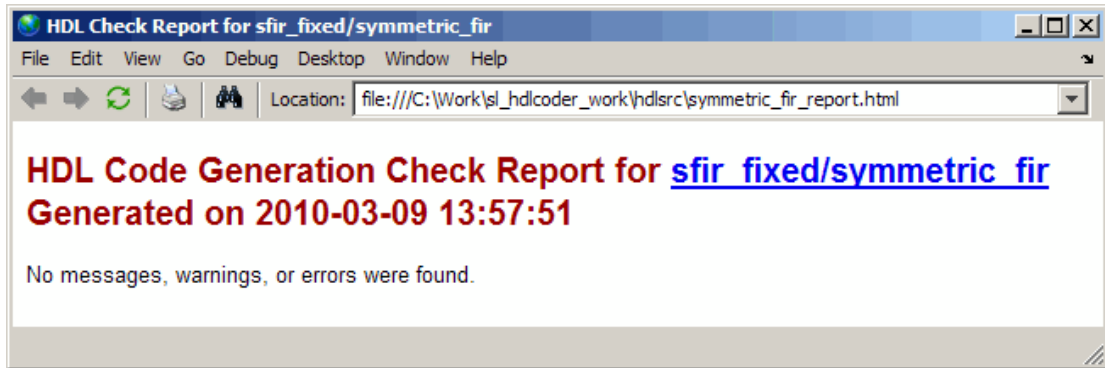
The following figure shows an HDL Code Generation Check Report that was generated for a subsystem with a Product block that was configured with a mixture of double and integer port data types. This configuration is legal in a model, but incompatible with HDL code generation.



When you click the hyperlink in the left column, the subsystem containing the offending block opens. The block of interest is highlighted, as shown in the following figure.



The following figure shows an HDL Code Generation Check Report that was generated for a subsystem that passed its compatibility checks. In this case, the report contains only a hyperlink to the subsystem that was checked.



Create a Supported Blocks Library

The `hdl1lib` function creates a library of blocks that are currently supported for HDL code generation. The block library, `hdlsupported`, affords quick access to supported blocks. By constructing models using blocks from this library, your models will be compatible with HDL code generation.

The set of supported blocks will change in future releases of HDL Coder. To keep the `hdlsupported` library current, you should rebuild the library each time you install a new release. To create the library:

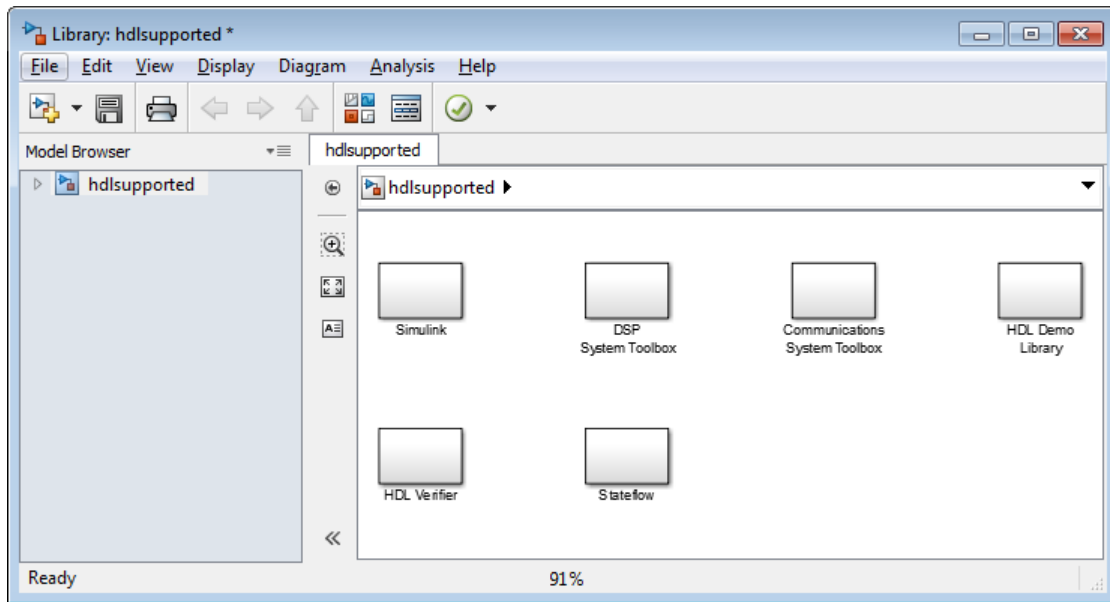
- 1 Type the following at the MATLAB prompt:

```
hdl1lib
```

`hdl1lib` starts generation of the `hdlsupported` library. Many libraries load during the creation of the `hdlsupported` library. When `hdl1lib` completes generation of the library, it does not unload these libraries.

- 2 After the library is generated, you must save it to a folder of your choice. You should retain the file name `hdlsupported`, because this document refers to the supported blocks library by that name.

The following figure shows the top-level view of the `hdlsupported` library.



Parameter settings for blocks in the `hdlsupported` library might differ from corresponding blocks in other libraries.

For detailed information about supported blocks and HDL block implementations, see “Set and View HDL Block Parameters”.

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.

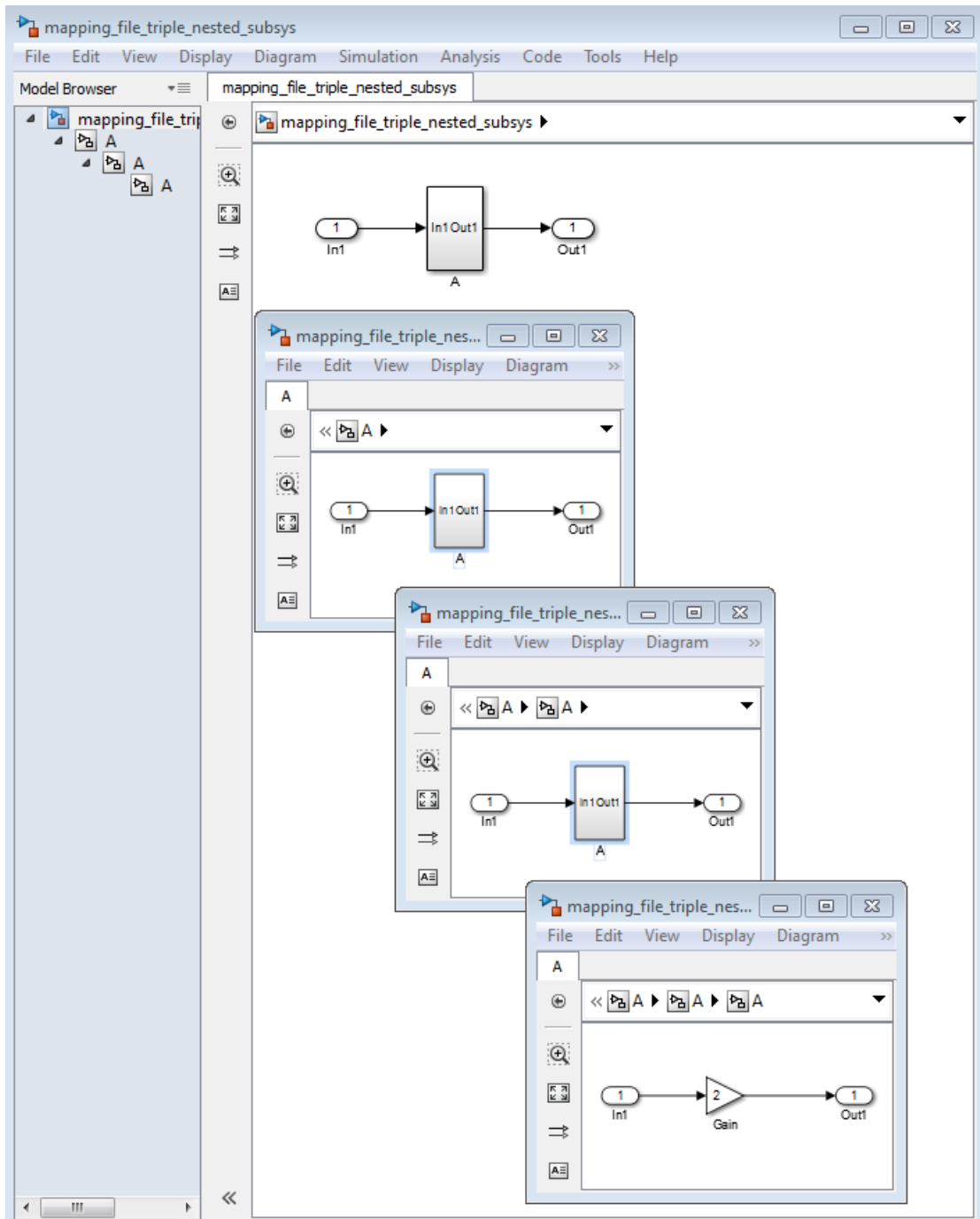
In simple cases, the mapping file may contain only one entry. For example, the `symmetric_fir` subsystem of the `sfir_fixed` model generates the following mapping file:

```
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 15-41

“Adding the HDL Coder Configuration Component To a Model” on page 15-41

“Removing the HDL Coder Configuration Component From a Model” on page 15-41

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:

- 1 In the Simulink Editor, select **Code > HDL Code**.
- 2 Select **Add HDL Coder Configuration to Model**.
- 3 Save the model.

Removing the HDL Coder Configuration Component From a Model

To remove the HDL Coder configuration component from a model:

- 1 In the Simulink Editor, select **Code > HDL Code**, and select **Remove HDL Coder Configuration from Model**.

HDL Coder displays a confirmation message.

- 2 Click **Yes** to confirm that you want to remove the HDL Coder configuration component.
- 3 Save the model.

HDL Coding Standards

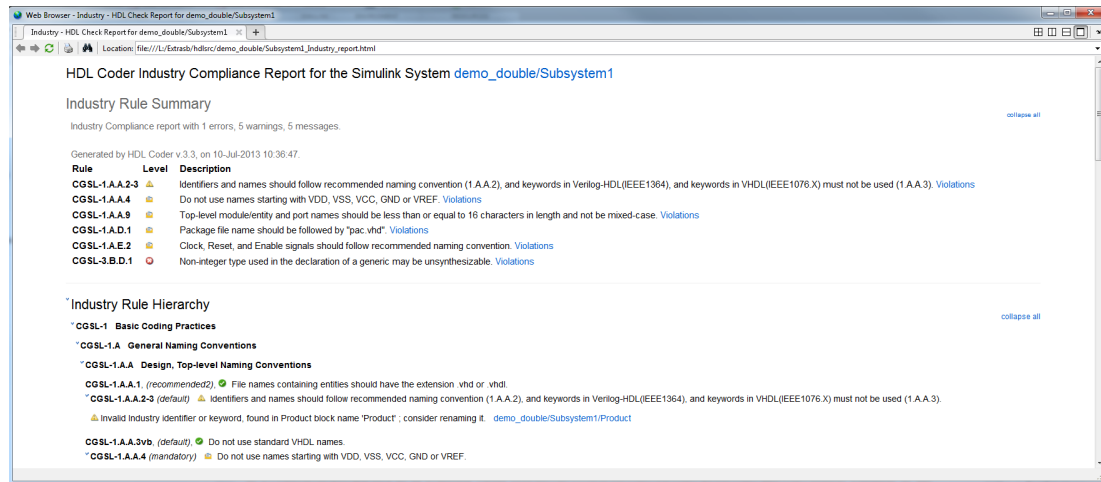
- “HDL Coding Standard Report” on page 16-2
- “HDL Coding Standards” on page 16-4
- “Generate an HDL Coding Standard Report from Simulink” on page 16-5
- “HDL Coding Standard Rules” on page 16-9
- “Generate an HDL Lint Tool Script” on page 16-15

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”.

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 Properties.

How To Fix Warnings and Errors

To learn more about warnings and errors you can fix by modifying your design, see “HDL Coding Standard Rules”.

HDL Coding Standards

HDL coding standards give 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

HDL Coder can generate HDL code that follows industry standard rules, and can generate a report that shows how well your generated HDL code conforms to industry coding standards. You can customize some of the coding standard rules. The coder can also generate third-party lint tool scripts to use to check your generated HDL code.

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB”
- “Generate an HDL Coding Standard Report from Simulink”

More About

- “HDL Coding Standard Report”

Generate an HDL Coding Standard Report from Simulink

In this section...
“Using the HDL Workflow Advisor” on page 16-5
“Using the Command Line” on page 16-7

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, 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

Choose coding standard

HDL coding standard: Industry ▾

Report options

Do not show passing rules in coding standard report

Basic coding rules

Check for duplicate names

Check for HDL keywords in design names

Check module, instance, entity name length

Minimum 2

Maximum 32

Check signal, port, parameter name length

Minimum 2

Maximum 40

RTL description rules

Minimize use of clock enable signals

Detect usage of reset signals

Minimize use of variables

Check for initial statements that set RAM initial values

Check if-else statement chain length

Length 7

Check if-else statement nesting depth

Depth 3

Check multiplier width

Maximum 16

- 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` 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 Properties

More About

- “HDL Coding Standard Report”
- “HDL Coding Standard Rules”

HDL Coding Standard Rules

When you generate an HDL coding standard report, the following industry standard rules may appear. You can fix errors or warnings related to these rules by updating your design, or by customizing the associated HDL coding standard rule.

Rule / Severity	Message	Problem	How to fix
1.A.A.2 <i>Message</i>	Identifiers and names should follow recommended naming convention.	A name in the design does not start with a letter, or contains a character other than a number, letter, or underscore.	Update the names in your design so that they start with a letter of the alphabet (a - z, A - Z), and contain only alphanumeric characters (a - z, A - Z, 0 - 9) and underscores (_).
1.A.A.3 <i>Message</i>	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 by using the <code>HDLKeywords</code> property of the HDL coding standard customization object.
1.A.A.5 <i>Error</i>	Do not use case variants of names in the same scope. (Verilog) Do not use names that differ in case only, within the same scope. (VHDL)	Two or more names in your design, within the same scope, are identical except for case. For example, the names <code>foo</code> and <code>Foo</code> cannot be in the same scope.	Update the names in your design so that no two names within the same scope differ only in case. You can disable this rule by using the <code>DetectDuplicateNamesCheck</code> property of the HDL coding standard customization object.
1.A.A.9 <i>Warning</i>	Top-level module/entity and port	A top-level module, entity, or port name	Update the indicated name in your design so

Rule / Severity	Message	Problem	How to fix
	names should be less than or equal to 16 characters in length and not be mixed-case.	in the generated code is longer than 16 characters, or uses letters with mixed case.	that it is less than or equal to 16 characters long, and all letters are lowercase, or all letters are uppercase.
1.A.B.1 <i>Error</i>	Module and Instance names should be between 2 and 32 characters in length. (Verilog) Entity names and instance names should be between 2 and 32 characters in length. (VHDL)	A module, instance, or entity name in the generated code is less than 2 characters long, or more than 32 characters long.	Update function names or subsystem names in your design to be between 2 and 32 characters long. You can customize this rule by using the SignalPortParamNameLength property of the HDL coding standard customization object.
1.A.C.3 <i>Error</i>	Signal names, port names, parameter names, define names and function names should be between 2 and 40 characters in length. (Verilog) Signal names, variable names, type names, label names and function names should be between 2 and 40 characters in length. (VHDL)	A signal, port, parameter, define, or function name in the generated code is less than 2 characters long, or more than 40 characters long.	Update the indicated name in your design so that it is between 2 and 40 characters long. You can customize this rule by using the ModuleInstanceEntityNameLength property of the HDL coding standard customization object.

Rule / Severity	Message	Problem	How to fix
1.A.D.1 <i>Warning</i>	<p>Include files must have extensions that match ".h", ".vh", ".inc", and ".h", ".inc", ".ht", ".tsk" for testbench. (Verilog)</p> <p>Package file name should be followed by "pac.vhd". (VHDL)</p>	<p>The filename extension of an include file is not one of the standard extensions.</p>	<p>Set the Verilog file extension or VHDL file extension to one of the standard extensions.</p> <p>Use the Verilog file extension and VHDL file extension option in the HDL Workflow Advisor, or the <code>VerilogFileExtension</code> and <code>VHDLFileExtension</code> properties from the command line.</p>
2.C.D.1 <i>Error</i>	<p>Do not specify flip-flop (or RAM) initial value using initial construct.</p>	<p>The generated HDL code for your design contains an unsynthesizable initial statement.</p>	<p>Disable the Initialize block RAM or Initialize all RAM blocks option in the HDL Workflow Advisor.</p> <p>You can disable this rule by using the <code>InitialStatements</code> property of the HDL coding standard customization object.</p>

Rule / Severity	Message	Problem	How to fix
2.G.C.1a <i>Message</i>	Nesting in if-else constructs should not be deeper than N levels.	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 3 levels of nesting:</p> <pre>if ... if ... if ... else else else</pre> <p>You can customize this rule by using the IfElseNesting property of the HDL coding standard customization object.</p>

Rule / Severity	Message	Problem	How to fix
2.G.C.1c <i>Message</i>	<p>Chain of if...else if constructs must not be exceed default number of levels. (Verilog)</p> <p>The chain of "if-elsif" construct must not be longer than default number of levels. (VHDL)</p>	<p>The generated HDL code contains an if-elseif statement with more than 7 branches.</p>	<p>Modify if-elseif statements in your MATLAB code so that the number of branches is 7 or fewer.</p> <p>For example, the following if-elseif pseudocode contains 3 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.J.F.5 <i>Warning</i>	<p>Large multipliers must not be described using the multiplication operator with RTL.</p>	<p>The generated HDL code contains a multiplication operator (*) where the output of the multiplication has a bitwidth of 16 or greater.</p>	<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 MultiplierBitWidth property of the HDL coding standard customization object.</p>

Rule / Severity	Message	Problem	How to fix
3.A.D.5 <i>Message</i>	The maximum number of characters in one line should not be more than N.	The generated HDL code contains lines greater than N characters. You may 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 <code>LineLength</code> property of the HDL coding standard customization object.
3.B.D.1 <i>Error</i>	Non-integer type used in the declaration of a generic may be unsynthesizable.	The generated HDL code contains a non-integer data type.	Modify your design so that it does not use fixed-point data types. You can disable this rule by using the <code>NonIntegerTypes</code> property of the HDL coding standard customization object.

See Also

Properties

HDL Coding Standard Customization

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB”
- “Generate an HDL Coding Standard Report from Simulink”

More About

- “HDL Coding Standard Report”

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 strings. If you want to generate a script for a custom lint tool, you must specify the initialization, command, and termination strings.

HDL Coder writes the initialization, command, and termination strings 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, you must 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 strings, 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 strings:

```
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 17-2
- “Generate Black Box Interface for Subsystem” on page 17-4
- “Generate Black Box Interface for Referenced Model” on page 17-10
- “Integrate Custom HDL Code Using DocBlock” on page 17-12
- “Customize Black Box or HDL Cosimulation Interface” on page 17-14
- “Specify Bidirectional Ports” on page 17-17
- “Generate Reusable Code for Atomic Subsystems” on page 17-19
- “Create a Xilinx System Generator Subsystem” on page 17-27
- “Create an Altera DSP Builder Subsystem” on page 17-29
- “Using Xilinx System Generator for DSP with HDL Coder” on page 17-32
- “Generate a Cosimulation Model” on page 17-36
- “Pass-Through and No-Op Implementations” on page 17-59

Model Referencing for HDL Code Generation

In this section...

“Benefits of Model Referencing for Code Generation” on page 17-2

“How To Generate Code for a Referenced Model” on page 17-2

“Limitations for Model Reference Code Generation” on page 17-3

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

Using the UI

To generate HDL code for referenced model 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.

Tip 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 nonconflicting port definitions.

Using the Command Line

To generate HDL code for a referenced model using the command line:

- 1 Set the **Architecture** property of the Model block to **ModelReference**.
- 2 Generate HDL code for your DUT subsystem.

For example, to generate HDL code for a DUT subsystem, `mydut`, that includes a model reference, `referenced_model`, at the command line, enter:

```
hdlset_param ('mydut/referenced_model', 'Architecture', 'ModelReference');  
makehdl ('mydut');
```

Tip 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 nonconflicting port definitions.

Limitations for Model Reference Code Generation

For model reference code generation restrictions, see `Model`.

Generate Black Box Interface for Subsystem

In this section...
“What Is a Black Box Interface?” on page 17-4
“Generate a Black Box Interface for a Subsystem” on page 17-4
“Generate Code for a Black Box Subsystem Implementation” on page 17-8
“Restriction for Multirate DUTs” on page 17-9

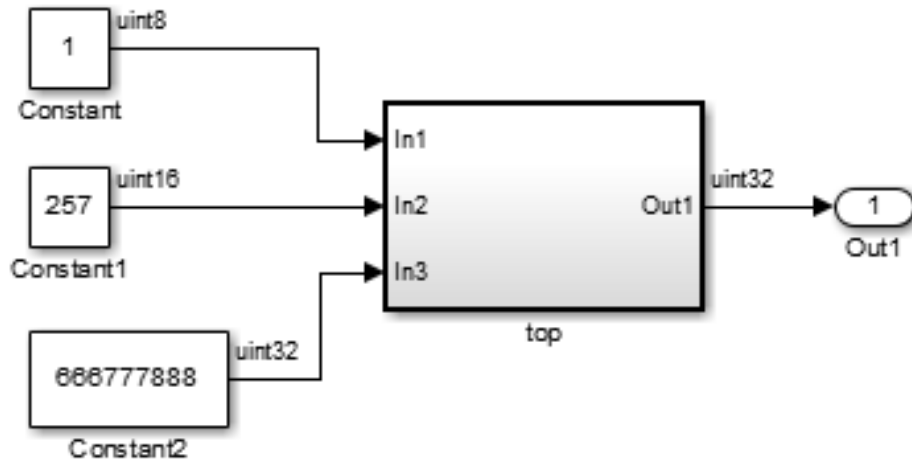
What Is a Black Box Interface?

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.

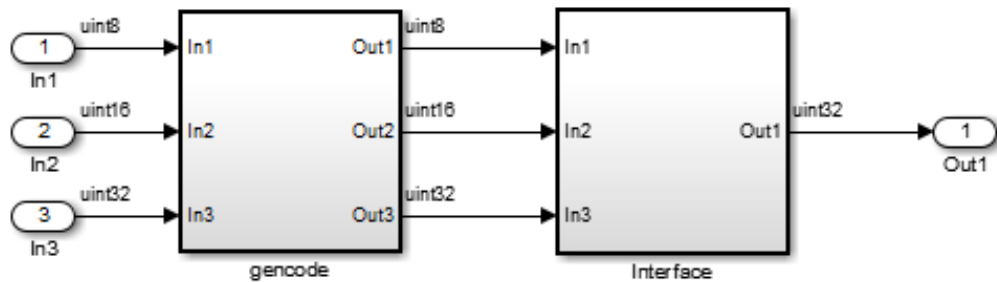
The black box implementation is available only for subsystem blocks below the level of the DUT. Virtual and atomic subsystem blocks of custom libraries that are below the level of the DUT also work with black box implementations.

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:

HDL Properties: Subsystem

Implementation

Architecture

Implementation Parameters

AddClockEnablePort

AddClockPort

AddResetPort

AllowDistributedPipelining

ClockEnableInputPort

ClockInputPort

ConstrainedOutputPipeline

EntityName

GenericList

ImplementationLatency

InlineConfigurations

InputPipeline

OutputPipeline

ResetInputPort

VHDLArchitectureName

VHDLComponentLibrary

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 17-14 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 17-14 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.

More About

- “Customize Black Box or HDL Cosimulation Interface”
- “Generate Black Box Interface for Referenced Model”
- “Integrate Custom HDL Code Using DocBlock”

Generate Black Box Interface for Referenced Model

In this section...
“When to Generate a Black Box Interface” on page 17-10
“How to Generate a Black Box Interface” on page 17-10

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”.

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”.
- 2 Generate HDL code for your DUT subsystem.

Note: The `checkhdl` function does not check port data types within the referenced model.

Tip 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 nonconflicting port definitions.

More About

- “Customize Black Box or HDL Cosimulation Interface”
- “Generate Black Box Interface for Subsystem”
- “Integrate Custom HDL Code Using DocBlock”

Integrate Custom HDL Code Using DocBlock

In this section...

“When To Use DocBlock to Integrate Custom Code” on page 17-12

“How To Use DocBlock to Integrate Custom Code” on page 17-12

“Restrictions” on page 17-13

“Example” on page 17-13

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 or black box model reference.

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 to match your custom HDL code interface. To learn more about customizing the black box interface, see “Customize Black Box or HDL Cosimulation Interface”.
- 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.

More About

- “Customize Black Box or HDL Cosimulation Interface”
- “Generate Black Box Interface for Subsystem”
- “Generate Black Box Interface for Referenced Model”

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.
ClockInputPort	Default: clk	Specifies HDL name for block's clock input signal.

Parameter Name	Values	Description
EntityName	Default: Entity name string is derived from the block name, and modified when necessary to generate a legal VHDL entity name.	Specifies VHDL entity or Verilog module name generated for the block.
GenericList	Default: An empty cell array of string data. Each element of the cell array is another cell array of the form { 'Name', 'Value', 'Type' }, where 'Type' is optional. If you omit 'Type', 'integer' is passed as the data type.	Specifies a list of parameter/value pairs (with optional data type specification) in string format to pass to a subsystem with a BlackBox implementation.
ImplementationLatency	-1 0 positive integer Default: -1	Specifies the additional latency of the external component in time steps, relative to the Simulink block. If 0 or greater, this value is used for delay balancing. Your inputs and outputs must operate at the same rate. If -1, latency is unknown. This disables delay balancing.
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.

Parameter Name	Values	Description
OutputPipeline	Default: 0	Specifies the number of output pipeline stages (pipeline depth) in the generated code.
ResetInputPort	Default: <code>reset</code>	Specifies HDL name for block's reset input.
VHDLArchitectureName (VHDL only)	Default: <code>rtl</code>	Specifies RTL architecture name generated for the block. The architecture name is generated only if InlineConfigurations is on.
VHDLComponentLibrary (VHDL only)	Default: <code>work</code>	Specifies the library from which to load the VHDL component.

More About

- “Generate Black Box Interface for Subsystem”
- “Generate Black Box Interface for Referenced Model”
- “Integrate Custom HDL Code Using DocBlock”
- “Specify Bidirectional Ports”

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 17-17

“How To Specify a Bidirectional Port” on page 17-17

“Limitations” on page 17-18

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.
- Simulink does not support bidirectional ports, so you cannot simulate the bidirectional behavior in Simulink.

More About

- “Generate Black Box Interface for Subsystem”
- “Generate Black Box Interface for Referenced Model”
- “Integrate Custom HDL Code Using DocBlock”
- “Customize Black Box or HDL Cosimulation Interface”

Generate Reusable Code for Atomic Subsystems

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.

In this section...

“Requirements for Generating Reusable Code for Atomic Subsystems” on page 17-19

“Generate Reusable Code for Atomic Subsystems” on page 17-19

“Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters” on page 17-22

Requirements for Generating Reusable Code for Atomic Subsystems

To generate reusable HDL code for atomic subsystems:

- `HandleAtomicSubsystem` must be on.
- The `InlineParams Simulink` configuration parameter must be on.
- The atomic subsystems must be identical, or identical except for their mask parameter values.

If the atomic subsystems are identical except for their mask parameter values:

- `MaskParameterAsGeneric` must be on.
- Mask parameters must be 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.

If you do not want to generate reusable code for identical atomic subsystems, or for atomic subsystems that are identical except for their tunable mask parameter values, disable the `HandleAtomicSubsystem` property. By default, `HandleAtomicSubsystem` is enabled.

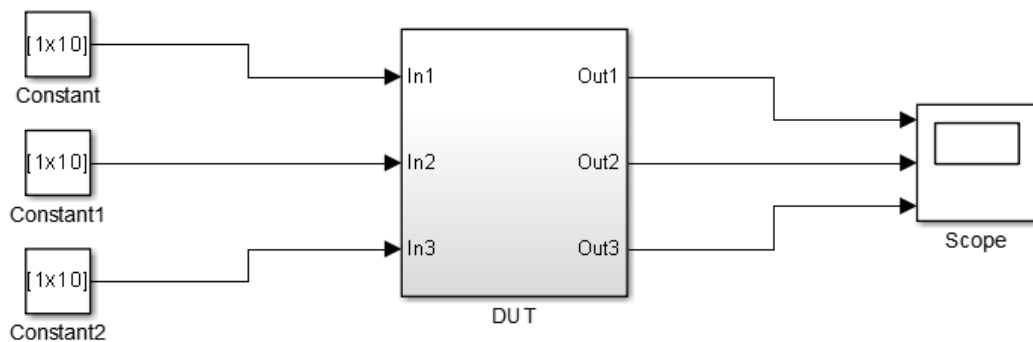
For example, to disable `HandleAtomicSubsystem` for the `hdlcoder_reusable_code_identical_subsystem` model, enter:

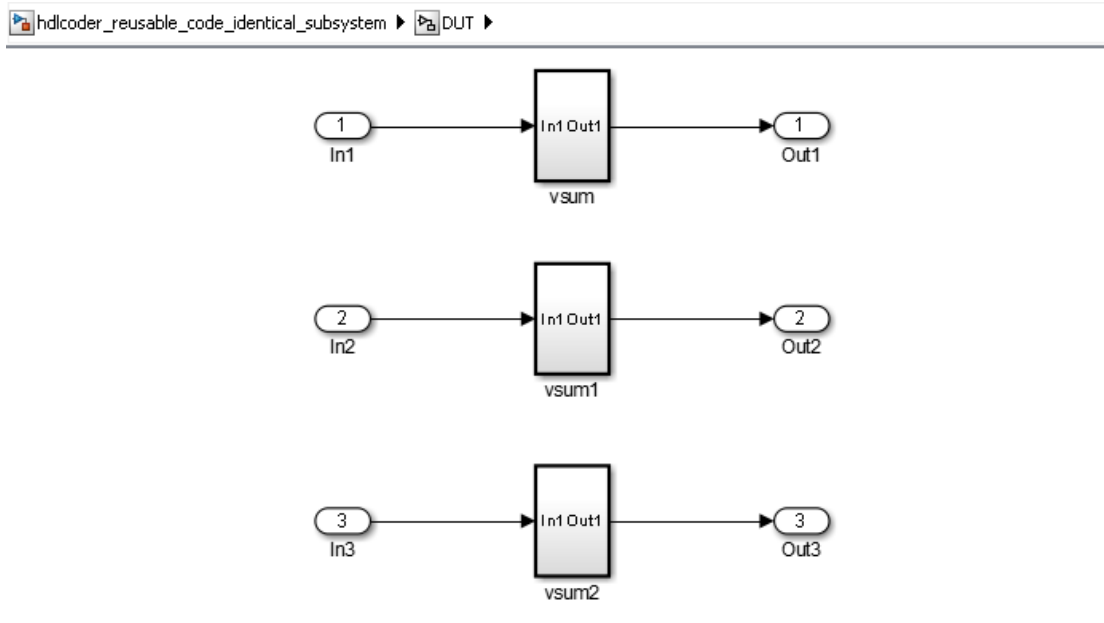
```
hdlset_param('hdlcoder_reusable_code_identical_subsystem','HandleAtomicSubsystem','off')
```

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 ▶





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 DOWNTO 0)  -- sfix20
    );
END COMPONENT;

-- Component Configuration Statements
FOR ALL : vsum
    USE ENTITY work.vsum(rtl);

-- Signals
SIGNAL vsum_out1                          : std_logic_vector(19 DOWNTO 0); -- ufix20
SIGNAL vsum1_out1                          : std_logic_vector(19 DOWNTO 0); -- ufix20
SIGNAL vsum2_out1                          : std_logic_vector(19 DOWNTO 0); -- ufix20

BEGIN
    u_vsum : vsum
        PORT MAP( In1 => In1, -- int16 [10]
                  Out1 => vsum_out1 -- sfix20
                );

    u_vsum1 : vsum
        PORT MAP( In1 => In2, -- int16 [10]
                  Out1 => vsum1_out1 -- sfix20
                );

    u_vsum2 : vsum
        PORT MAP( In1 => In3, -- int16 [10]
                  Out1 => vsum2_out1 -- sfix20
                );

    Out1 <= vsum_out1;

    Out2 <= vsum1_out1;

    Out3 <= vsum2_out1;

END rtl;

```

Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters

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:

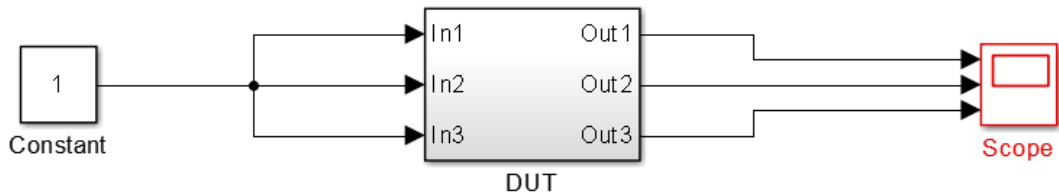
```
hdlset_param('hdlcoder_reusable_code_parameterized_subsystem', 'MaskParameterAsGeneric', 'on')
```

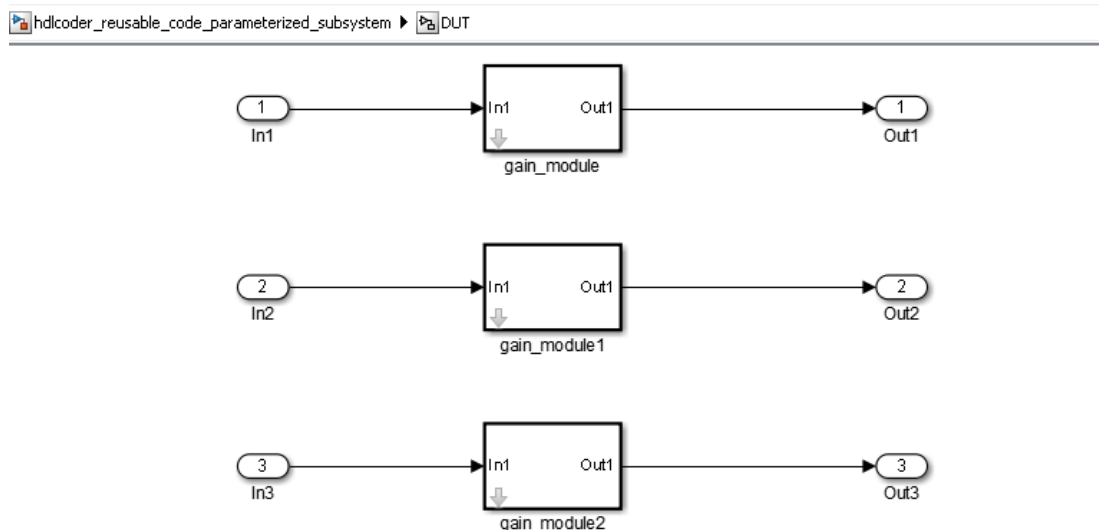
Alternatively, in the Configuration Parameters dialog box, in the **HDL Code Generation > Global Settings > Coding Style** tab, enable the **Generate parameterized HDL code from masked subsystem** option.

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` ▶





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

[HandleAtomicSubsystem](#) | [MaskParameterAsGeneric](#)

More About

- “Generate parameterized HDL code from masked subsystem”

Create a Xilinx System Generator Subsystem

In this section...

“Why Use Xilinx System Generator Subsystems?” on page 17-27

“Requirements for Xilinx System Generator Subsystems” on page 17-27

“How to Create a Xilinx System Generator Subsystem” on page 17-28

“Limitations for Code Generation from Xilinx System Generator Subsystems” on page 17-28

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, you must use ISE Design Suite 13.4 or later.

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 17-28.

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 17-32.

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 17-29

“Requirements for Altera DSP Builder Subsystems” on page 17-29

“How to Create an Altera DSP Builder Subsystem” on page 17-30

“Determine Clocking Requirements for Altera DSP Builder Subsystems” on page 17-30

“Limitations for Code Generation from Altera DSP Builder Subsystems” on page 17-31

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” 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™.

Setup for Xilinx® System Generator

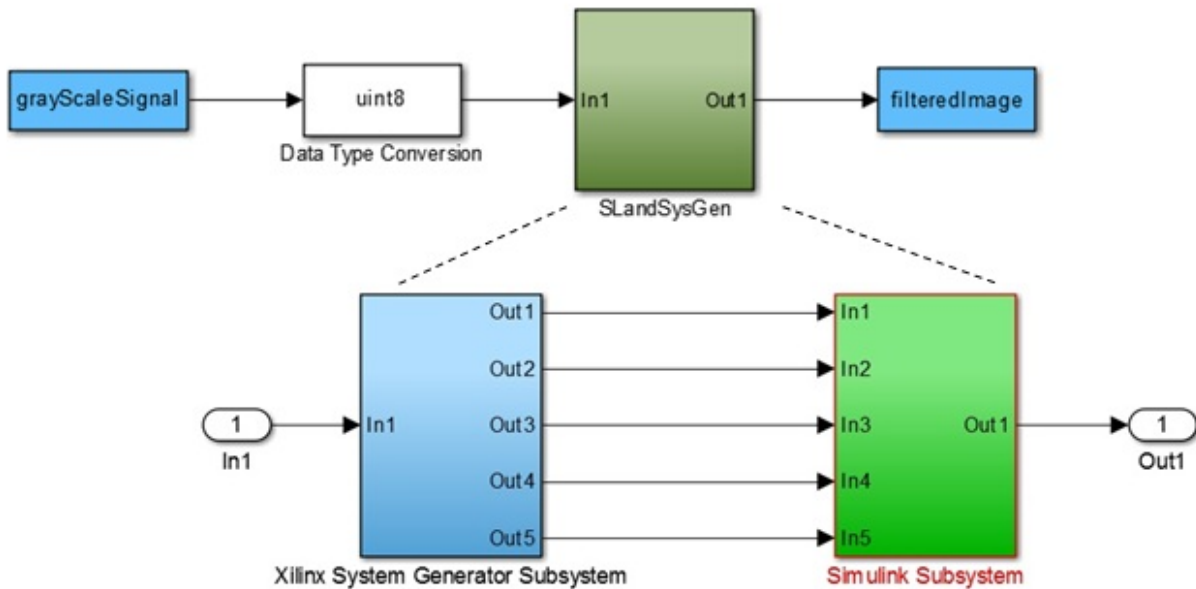
In order to use the Xilinx® System Generator Subsystem block, you must have Xilinx® ISE 13.4 set up with Simulink®.

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_slssystemgen/SLandSysGen/Xilinx System Generator Subsystem). 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_slssystemgen');  
open_system('hdlcoder_slssystemgen/SLandSysGen');
```

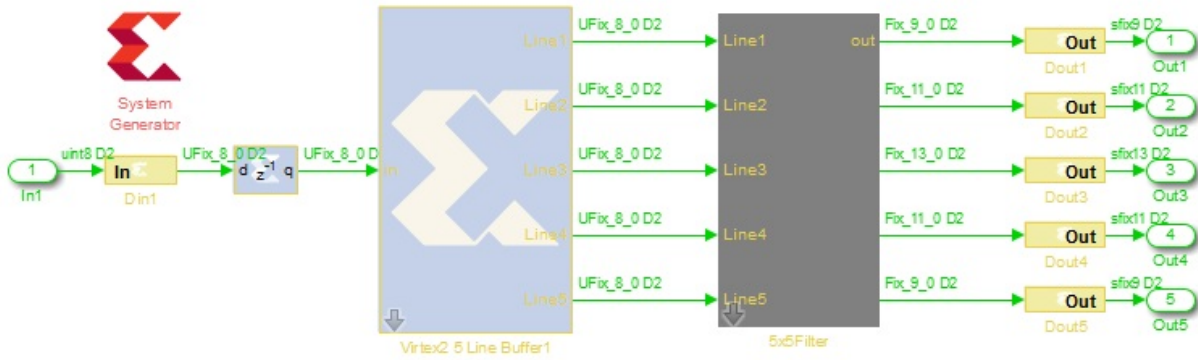



Create Xilinx® System Generator Subsystem

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_slsgen/SLandSysGen/Xilinx System Generator Subsystem');
```



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.

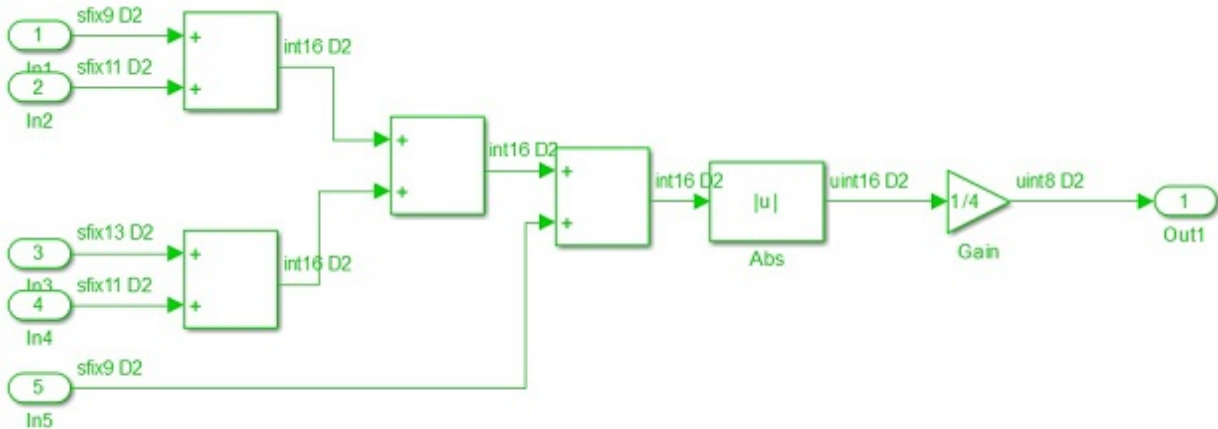
Gateway In blocks must not do non-trivial data type conversion. For example, a Gateway In block can convert between uint8 and UFix_8_0, but changing data sign, word length, or fraction length is not allowed.

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_slsysgen/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_slsysgen/SLandSysGen/Simulink Subsystem');
```



Generate HDL Code

You can use either `makehdl` at the command line or HDL Workflow Advisor to generate HDL code. To use `makehdl`:

```
makehdl('hdlcoder_slsysgen/SLandSysGen');
```

You can also generate a testbench, simulate, and synthesize the design as you would for any other model.

Generate a Cosimulation Model

In this section...

- “What Is A Cosimulation Model?” on page 17-36
- “Generating a Cosimulation Model from the GUI” on page 17-37
- “Structure of the Generated Model” on page 17-42
- “Launching a Cosimulation” on page 17-49
- “The Cosimulation Script File” on page 17-52
- “Complex and Vector Signals in the Generated Cosimulation Model” on page 17-54
- “Generating a Cosimulation Model from the Command Line” on page 17-56
- “Naming Conventions for Generated Cosimulation Models and Scripts” on page 17-57
- “Limitations for Cosimulation Model Generation” on page 17-57

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 .

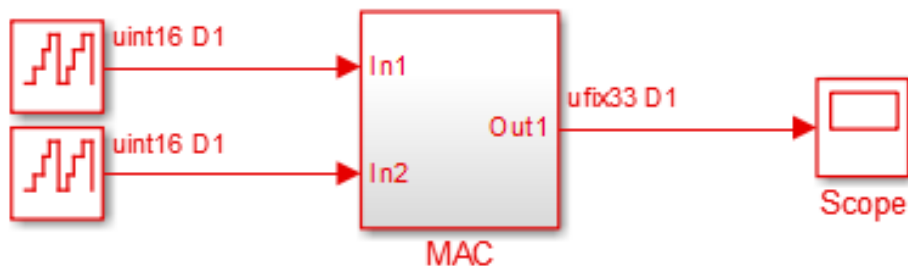
In addition to the generated model, HDL Coder generates a TCL script that launches and configures your cosimulation tool. Comments in the script file document clock, reset, and other timing signal information defined by the coder for the cosimulation tool.

Generating a Cosimulation Model from the GUI

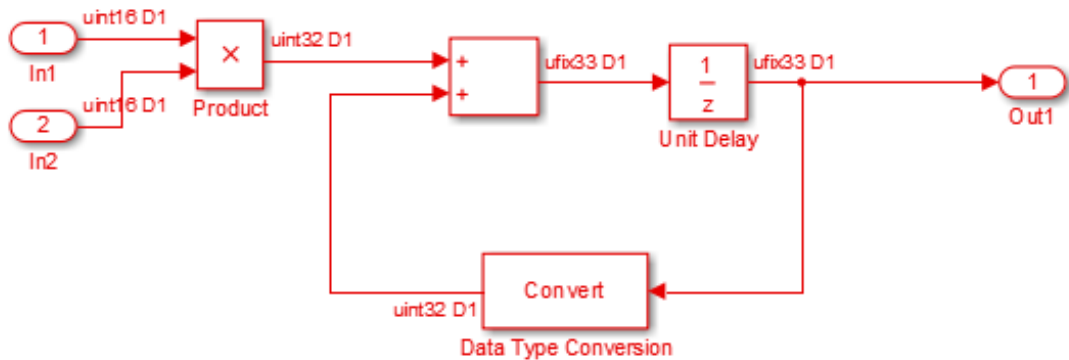
This example demonstrates the process for generating a cosimulation model. The example model, `hdl_cosim_demo1`, implements a simple multiply and accumulate (MAC) algorithm. Open the model by entering the name at the MATLAB command line:

```
hdl_cosim_demo1
```

The following figure shows the top-level model.

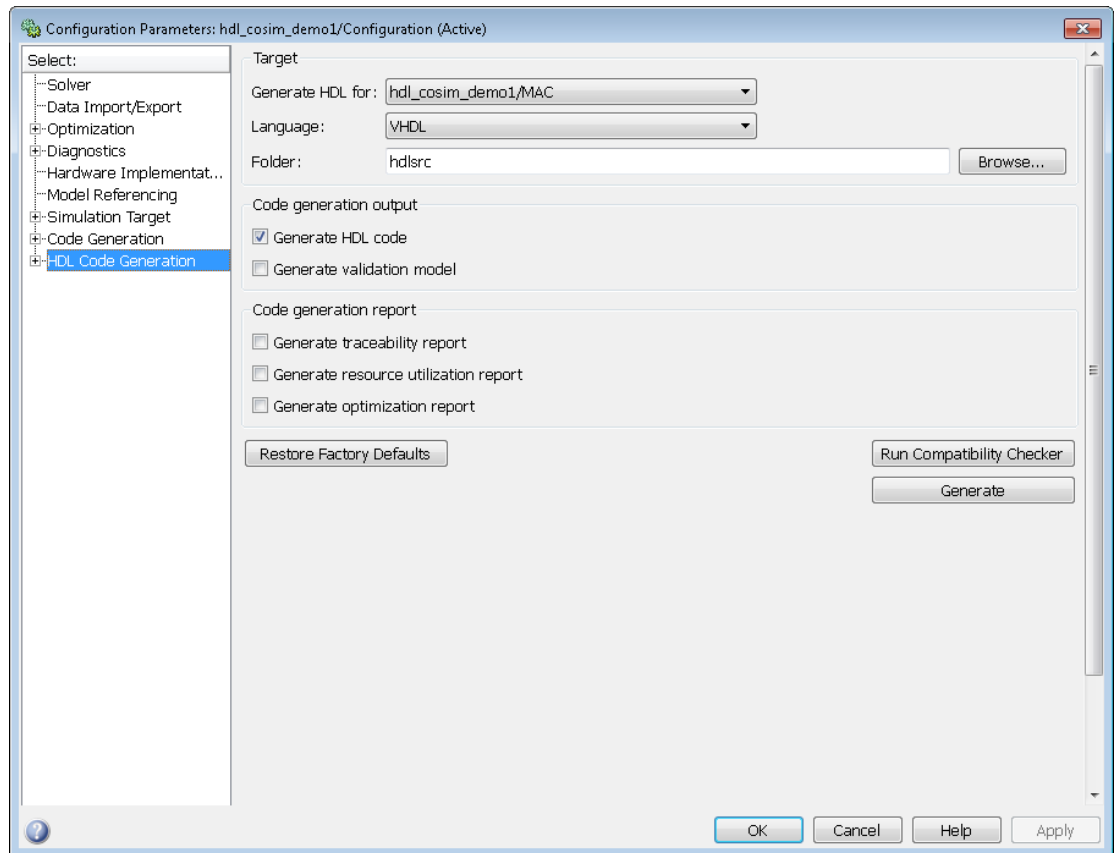


The DUT is the MAC subsystem.



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:

```

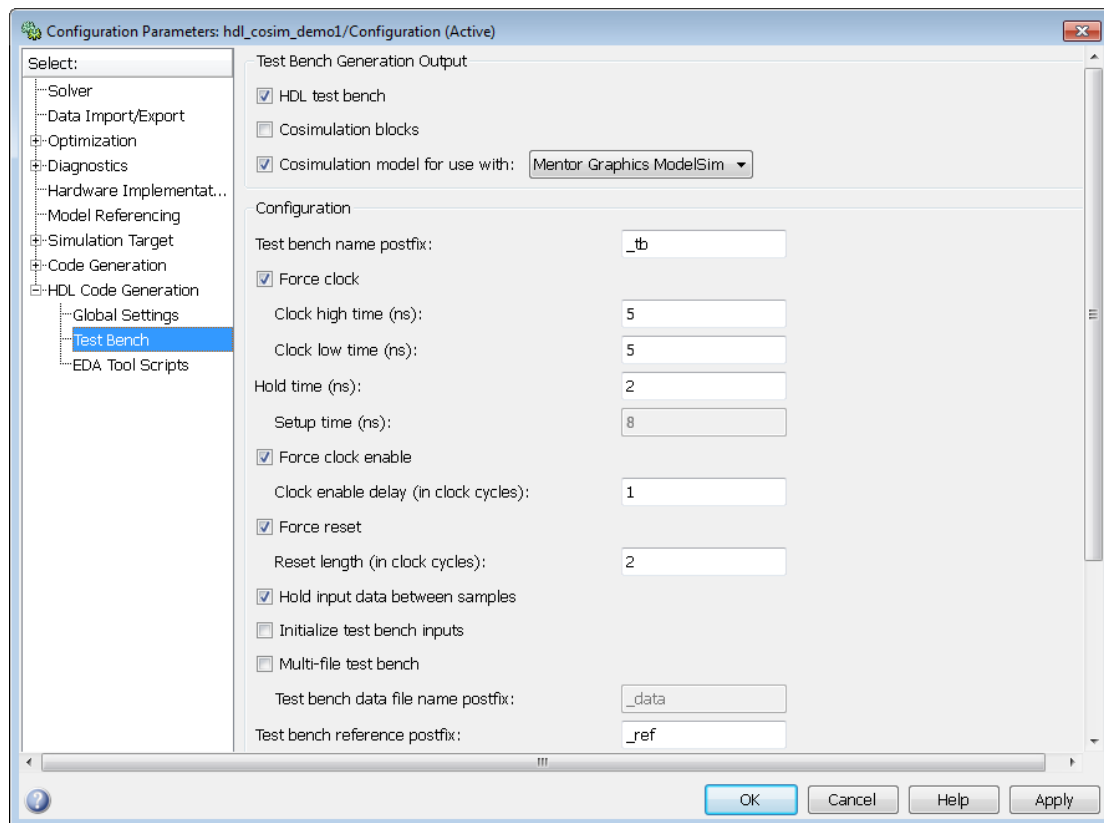
### Applying HDL Code Generation Control Statements
### Starting HDL Check.
### HDL Check Complete with 0 error, 0 warning and 0 message.

### 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 for use with:** option. Selecting this check box enables the pulldown menu to the right.



- 3 Select the desired cosimulation tool from the dropdown menu.
- 4 Configure required test bench options. HDL Coder records option settings in a generated script file (see “The Cosimulation Script File” on page 17-52).
- 5 Click **Apply**.

Next, generate test bench code and the cosimulation model:

- 1 Click **Generate Test Bench**. HDL Coder displays progress messages as shown in the following listing:

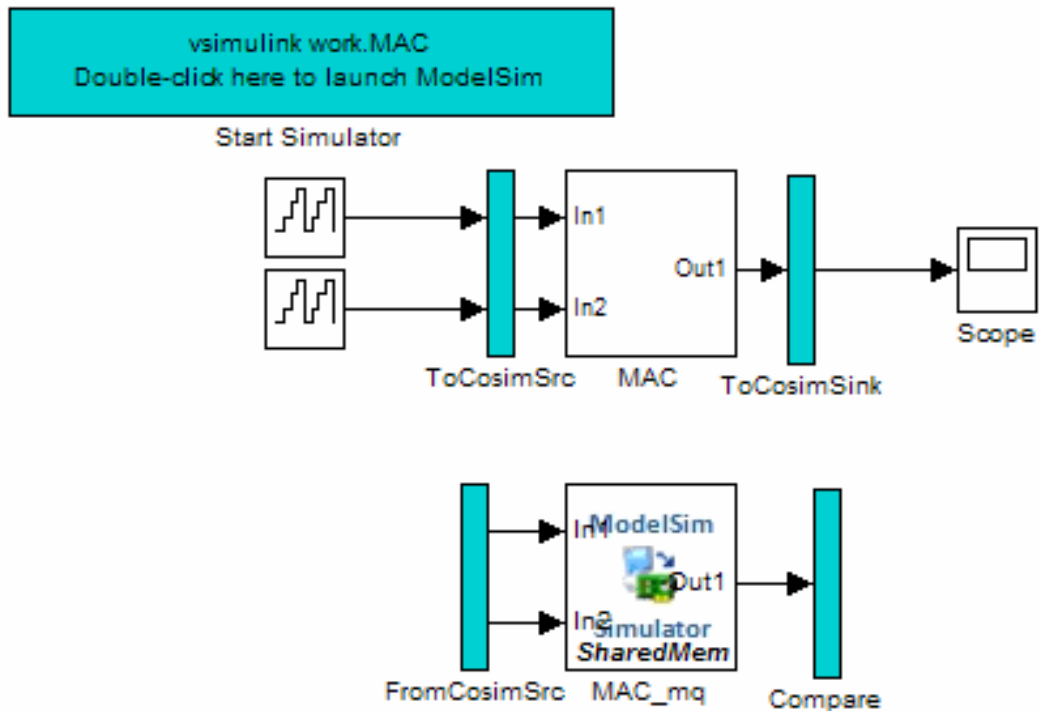

```

### 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 17-57.

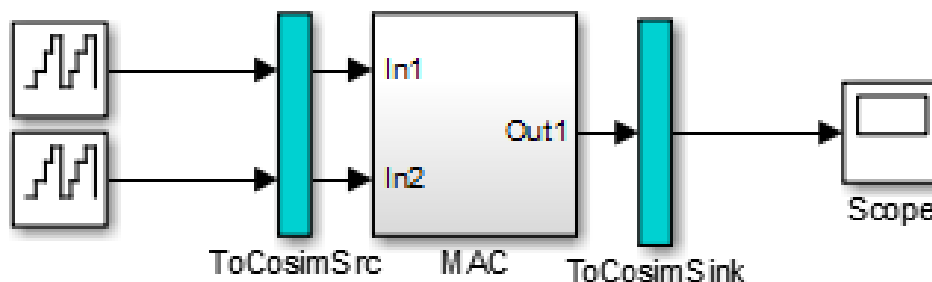
The next section, “Structure of the Generated Model” on page 17-42, describes the features of the model. Before running a cosimulation, become familiar with these features.

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

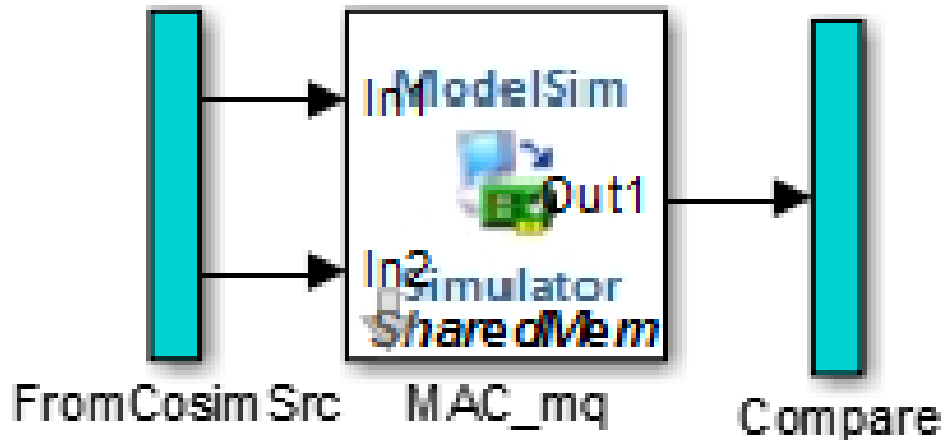
The model comprises two parallel signal paths. The *simulation path*, located in the upper half of the model window, is nearly identical to the original DUT. The purpose of the simulation path is to execute a normal Simulink simulation and provide a reference signal for comparison to the cosimulation 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 17-45).

Cosimulation Path

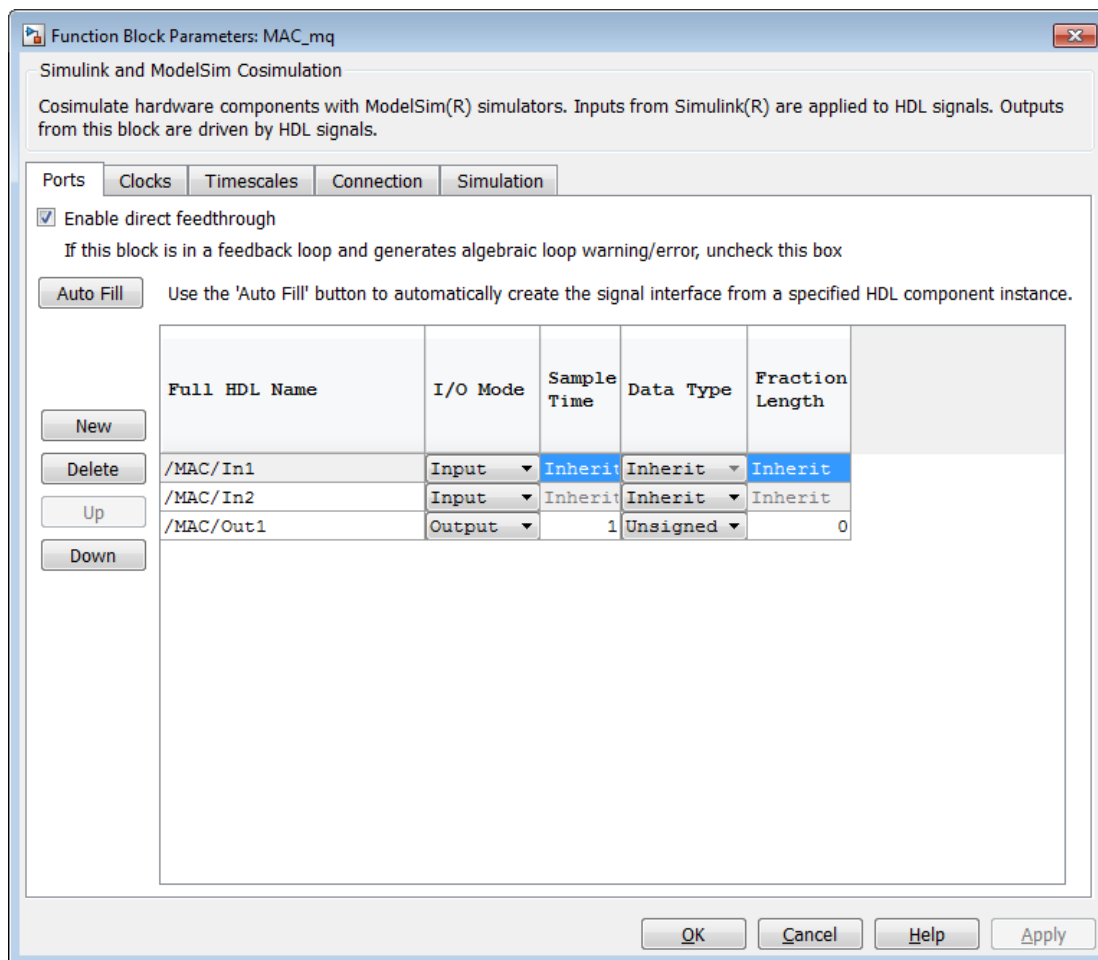
The *cosimulation path*, located in the lower half of the model window, contains the generated HDL Cosimulation block. The following figure shows the cosimulation path.



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 17-45).

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 17-47 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.

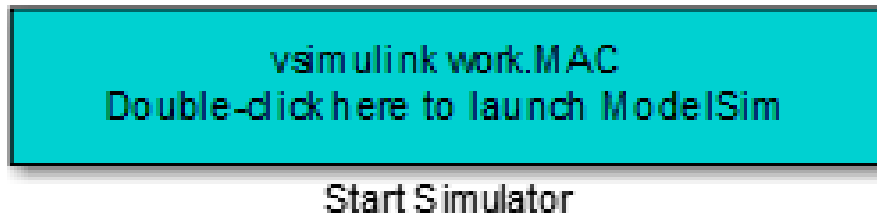


HDL Coder sets the **Full HDL Name**, **Sample Time**, **Data Type**, and other fields as required by the model. HDL Coder also configures other HDL Cosimulation block parameters under the **Timescales** and **Tcl** panes.

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.

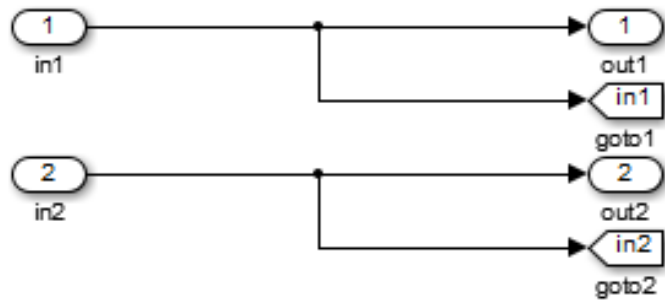


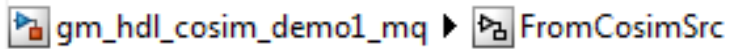
The commands executed when you double-click the Start Simulator icon launch and set up the cosimulation tool, but they do not start the actual cosimulation. “Launching a Cosimulation” on page 17-49 describes how to run a cosimulation with the generated model.

Signal Routing Between Simulation and Cosimulation Paths

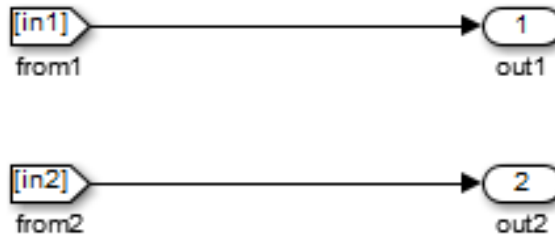
The generated model routes signals between the simulation and cosimulation paths using Goto and From blocks. For example, the Goto blocks in the ToCosimSrc subsystem route each DUT input signal to a corresponding From block in the FromCosimSrc subsystem. The following figures show the Goto and From blocks in each subsystem.

gm_hdl_cosim_demo1_mq ▶ ToCosimSrc





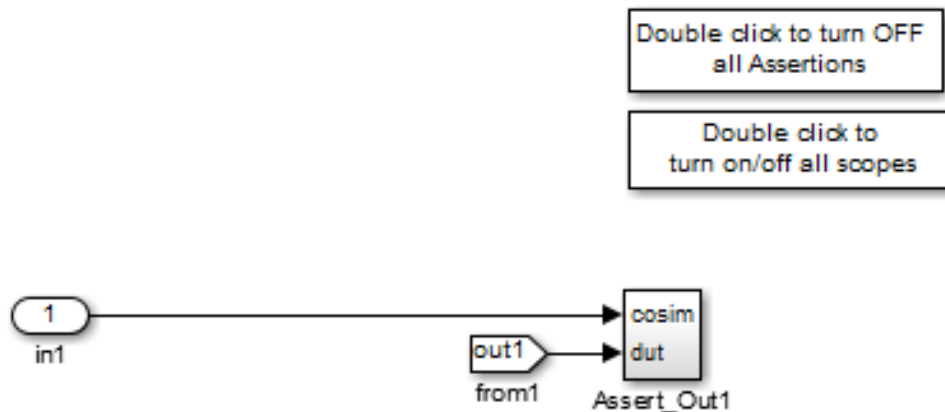
gm_hdl_cosim_demo1_mq ▶ FromCosimSrc



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 17-54 for further information.

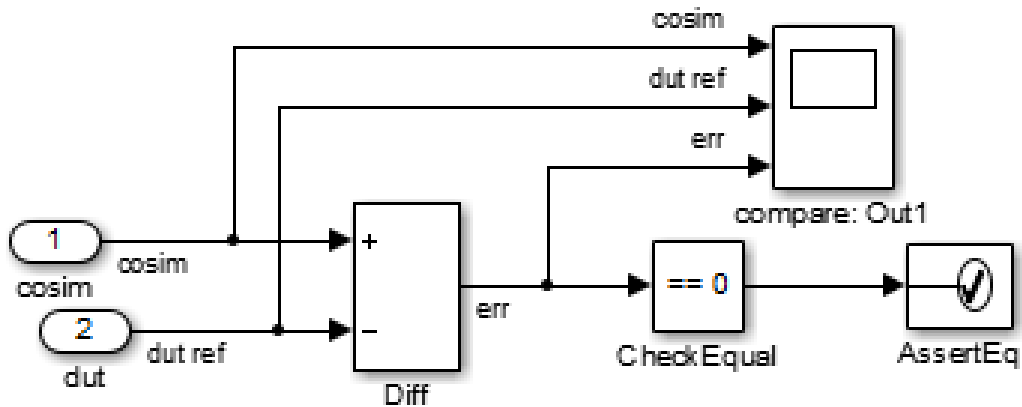
Controlling Assertions and Scope Displays

The **Compare** subsystem lets you control the display of signals on scopes, and warning messages from assertions. The following figure shows the **Compare** subsystem for the gm_hdl_cosim_demo1_mq0 model.



For each output of the DUT, HDL Coder generates an assertion checking subsystem (`Assert_OutN`). The subsystem computes the difference (`err`) between the original DUT output (`dut_ref`) and the corresponding cosimulation output (`cosim`). The subsystem routes the comparison result to an Assertion block. If the comparison result is not zero, the Assertion block reports the discrepancy.

The following figure shows the `Assert_Out1` subsystem for the `gm_hdl_cosim_demo1_mq0` model.



This subsystem also routes the `dut_ref`, `cosim`, and `err` signals to a Scope for display at the top level of the model.

By default, the generated cosimulation model enables all assertions and displays all Scopes. Use the buttons on the **Compare** subsystem to disable assertions or hide Scopes.

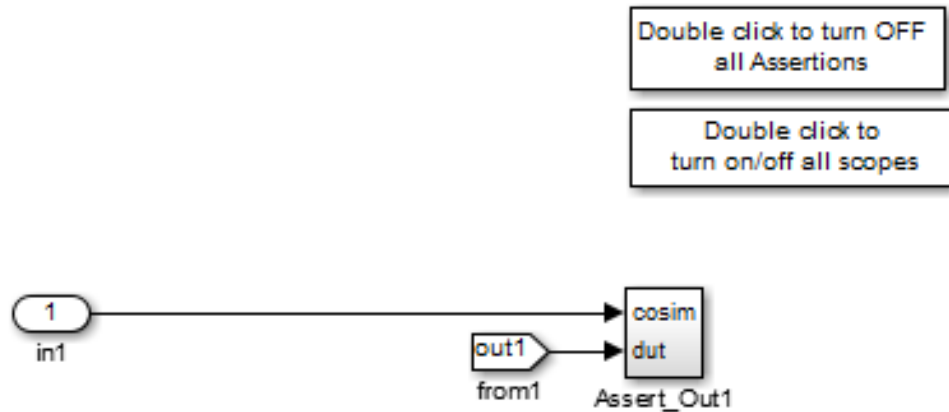
Tip Assertion messages are warnings and do not stop simulation.

Launching a Cosimulation

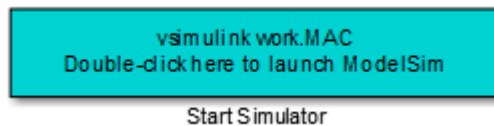
To run a cosimulation with the generated model:

- 1 Double-click the **Compare** subsystem to configure Scopes and assertion settings.

If you want to disable Scope displays or assertion warnings before starting your cosimulation, use the buttons on the **Compare** subsystem (shown in the following figure).



- 2 Double-click the Start Simulator control.



The Start Simulator control launches your HDL simulator (in this case, HDL Verifier for use with Mentor Graphics ModelSim).

The HDL simulator in turn executes a startup script. In this case the startup script consists of the TCL commands located in `gm_hdl_cosim_demo1_mq0_tcl.m`. When the HDL simulator finishes executing the startup script, it displays a message like the following.

```
# Ready for cosimulation...
```

- 3 In the Simulink Editor for the generated model, start simulation.

As the cosimulation runs, the HDL simulator displays messages like the following.

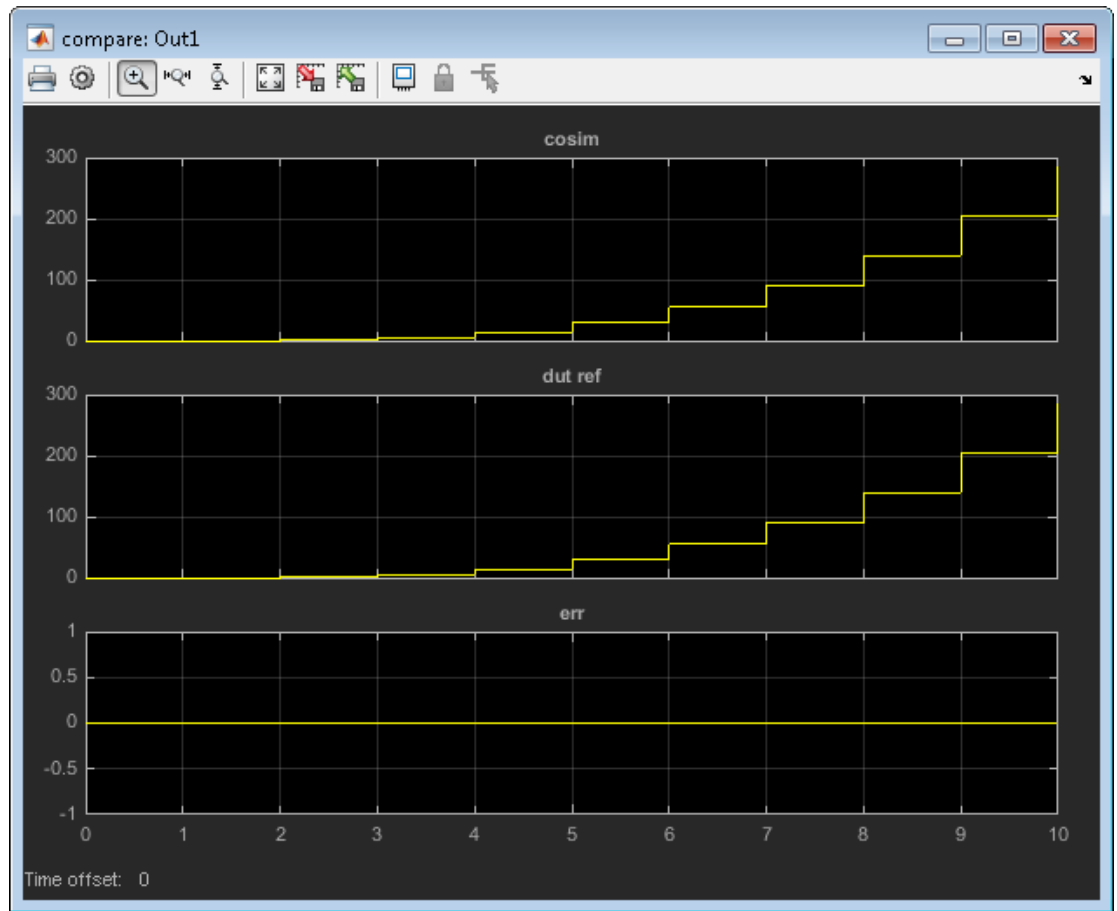
```
# Running Simulink Cosimulation block.
# Chip Name: --> hdl_cosim_demo1/MAC
# Target language: --> vhdl
# Target directory: --> hdlsrc
```

```
# Fri Jun 05 4:26:34 PM Eastern Daylight Time 2009  
# Simulation halt requested by foreign interface.  
# done
```

At the end of the cosimulation, if you have enabled Scope displays, the compare scope displays the following signals:

- `cosim`: The result signal output by the HDL Cosimulation block.
- `dut_ref`: The reference output signal from the DUT.
- `err`: The difference (error) between these two outputs.

The following figure shows these signals.



The Cosimulation Script File

The generated script file has two sections:

- A comment section that documents model settings that are relevant to cosimulation.
- A function that stores several lines of TCL code into a variable, `tclCmds`. The cosimulation tools execute these commands when you launch a cosimulation.

Header Comments Section

The following listing shows the comment section of a script file generated for the hdl_cosim_demo1 model:

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auto generated cosimulation 'tclstart' script
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Source Model      : hdl_cosim_demo1.mdl
% Generated Model   : gm_hdl_cosim_demo1.mdl
% Cosimulation Model : gm_hdl_cosim_demo1_mq.mdl
%
% Source DUT        : gm_hdl_cosim_demo1_mq/MAC
% Cosimulation DUT  : gm_hdl_cosim_demo1_mq/MAC_mq
%
% File Location     : hdlsrc/gm_hdl_cosim_demo1_mq_tcl.m
% Created           : 2009-06-16 10:51:01
%
% Generated by MATLAB 7.9 and HDL Coder 1.6
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ClockName         : clk
% ResetName         : reset
% ClockEnableName   : clk_enable
%
% ClockLowTime      : 5ns
% ClockHighTime     : 5ns
% ClockPeriod       : 10ns
%
% ResetLength       : 20ns
% ClockEnableDelay  : 10ns
% HoldTime          : 2ns
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ModelBaseSampleTime : 1
% OverClockFactor      : 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mapping of DutBaseSampleTime to ClockPeriod
%
% N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
% 1 sec in Simulink corresponds to 10ns in the HDL
% Simulator(N = 10)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ResetHighAt       : (ClockLowTime + ResetLength + HoldTime)
% ResetRiseEdge     : 27ns
% ResetType          : async
% ResetAssertedLevel : 1
%
% ClockEnableHighAt : (ClockLowTime + ResetLength + ClockEnableDelay + HoldTime)
% ClockEnableRiseEdge : 37ns
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

The comments section comprises the following subsections:

- *Header comments:* This section documents the files names for the source and generated models and the source and generated DUT.
- *Test bench settings:* This section documents the `makehdltb` property values that affect cosimulation model generation. The generated TCL script uses these values to initialize the cosimulation tool.
- *Sample time information:* The next two sections document the base sample time and oversampling factor of the model. HDL Coder uses `ModelBaseSampleTime` and `OverClockFactor` to map the clock period of the model to the HDL cosimulation clock period.
- *Clock, clock enable, and reset waveforms:* This section documents the computations of the duty cycle of the `clk`, `clk_enable`, and `reset` signals.

TCL Commands Section

The following listing shows the TCL commands section of a script file generated for the `hdl_cosim_demo1` model:

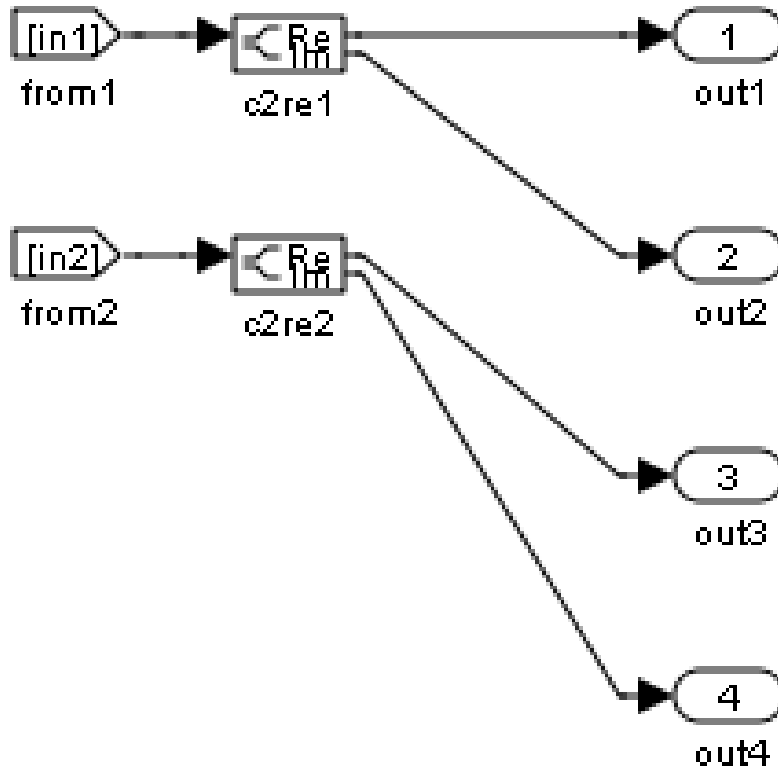
```
function tclCmds = gm_hdl_cosim_demo1_mq_tcl
tclCmds = {
    'do MAC_compile.do',...% Compile the generated code
    'vsimulink work.MAC',...% Initiate cosimulation
    'add wave /MAC/clk',...% Add wave commands for chip input signals
    'add wave /MAC/reset',...
    'add wave /MAC/clk_enable',...
    'add wave /MAC/In1',...
    'add wave /MAC/In2',...
    'add wave /MAC/ce_out',...% Add wave commands for chip output signals
    'add wave /MAC/Out1',...
    'set UserTimeUnit ns',...% Set simulation time unit
    'puts "...',...
    'puts "Ready for cosimulation..."',...
};
end
```

Complex and Vector Signals in the Generated Cosimulation Model

Input signals of complex or vector data types require insertion of additional elements into the cosimulation path. this section describes these elements.

Complex Signals

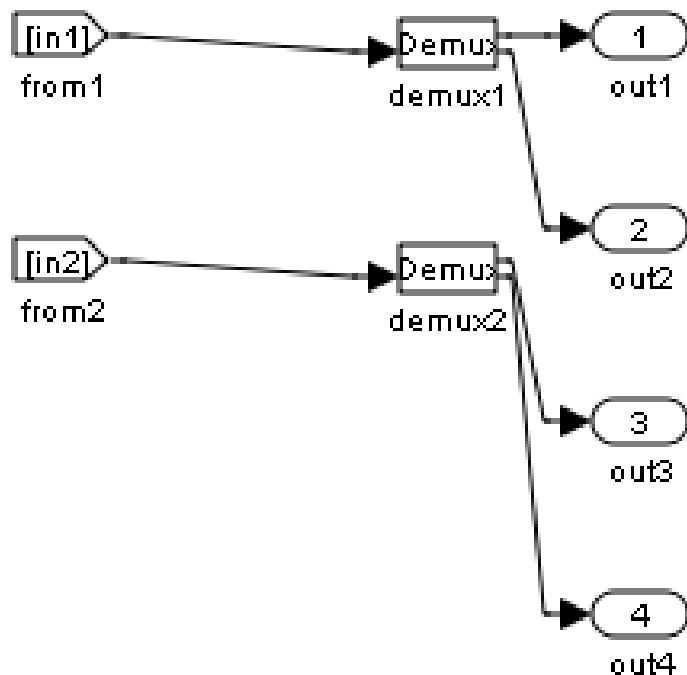
The generated cosimulation model automatically breaks complex inputs into real and imaginary parts. The following figure shows a `FromCosimSrc` subsystem that receives two complex input signals. The subsystem breaks the inputs into real and imaginary parts before passing them to the subsystem outputs.



The model maintains the separation of real and imaginary components throughout the cosimulation path. The Compare subsystem performs separate comparisons and separate scope displays for the real and imaginary signal components.

Vector Signals

The generated cosimulation model flattens vector inputs. The following figure shows a FromCosimSrc subsystem that receives two vector input signals of dimension 2. The subsystem flattens the inputs into scalars before passing them to the subsystem outputs.



Generating a Cosimulation Model from the Command Line

To generate a cosimulation model from the command line, pass the `GenerateCosimModel` property to the `makehdltb` function. `GenerateCosimModel` takes one of the following property values:

- `'ModelSim'`: generate a cosimulation model configured for HDL Verifier for use with Mentor Graphics ModelSim.
- `'Incisive'`: generate a cosimulation model configured for HDL Verifier for use with Cadence Incisive.

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.
- Combinatorial output paths (caused by absence of registers in the generated code) have a latency of one extra cycle in cosimulation. This causes a one cycle discrepancy in the comparison between the simulation and cosimulation outputs. To avoid this discrepancy, the **Enable direct feedthrough for HDL design with pure combinational datapath** 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”). 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	<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>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.</p>

Stateflow HDL Code Generation Support

- “Introduction to Stateflow HDL Code Generation” on page 18-2
- “Hardware Realization of Stateflow Semantics” on page 18-3
- “Generate HDL for Mealy and Moore Finite State Machines” on page 18-4
- “Design Patterns Using Advanced Chart Features” on page 18-12

Introduction to Stateflow HDL Code Generation

In this section...
“Overview” on page 18-2
“Example” on page 18-2

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.

Example

The Pipelined Configurable FIR example illustrates HDL code generation from a subsystem that includes Stateflow charts.

To view the example, open the HDL Coder documentation, click **Examples**, and open the Pipelined Configurable FIR example.

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.

Generate HDL for Mealy and Moore Finite State Machines

In this section...
“Overview” on page 18-4
“Generating HDL Code for a Moore Finite State Machine” on page 18-4
“Generating HDL for a Mealy Finite State Machine” on page 18-8

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:

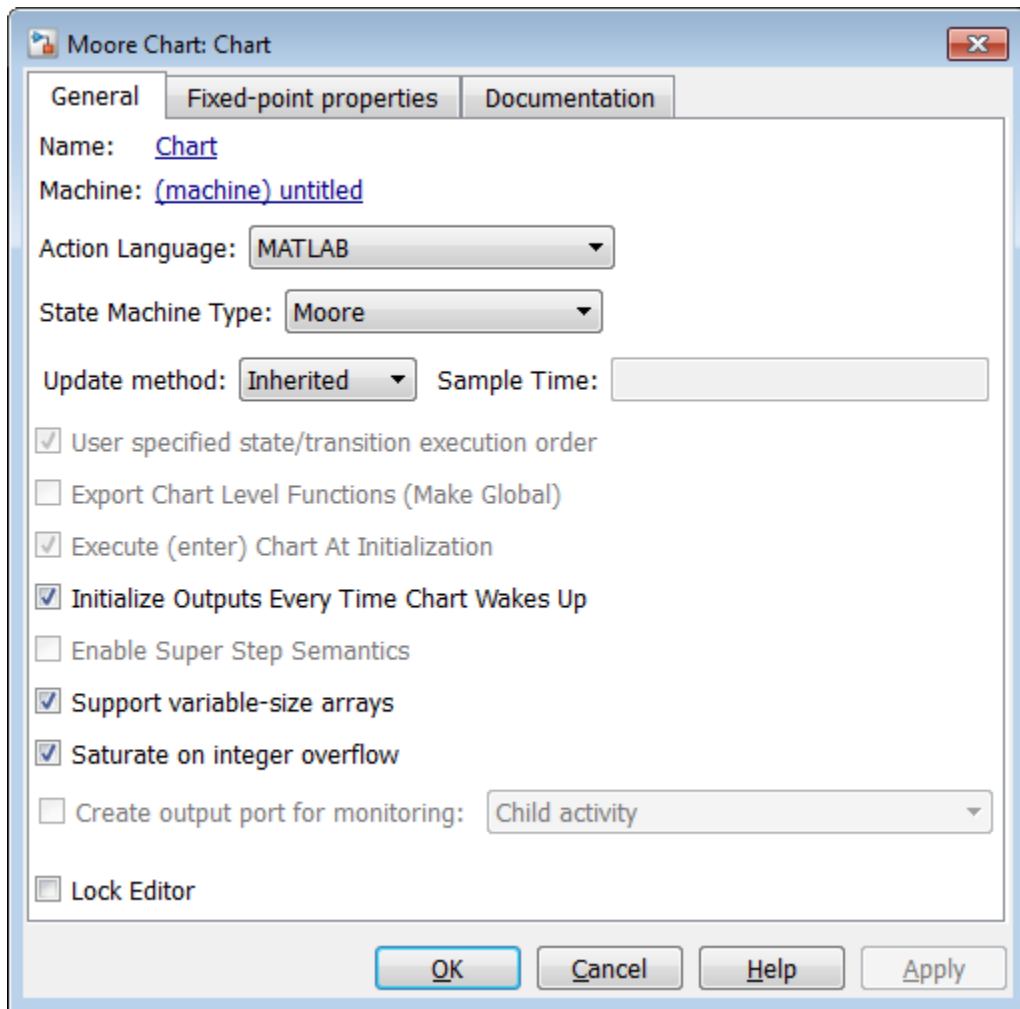
- The chart must meet the general code generation requirements as described in Chart.
- Actions must occur in states only. These actions must be unlabeled.

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 $t+1$.

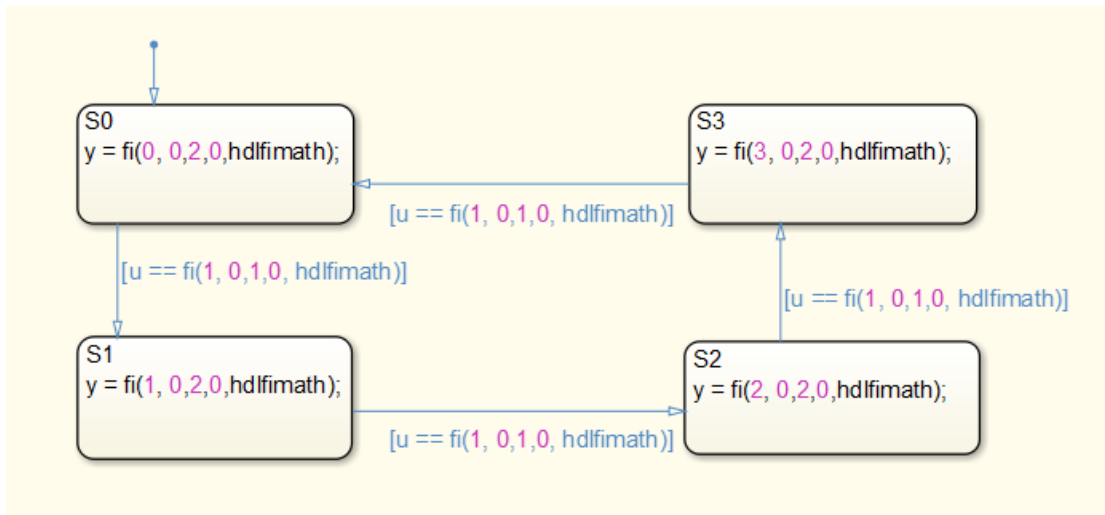
- Do not call Simulink functions.

This prevents output from depending on input in ways that would be difficult for the HDL code generator to verify.

- 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.



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
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

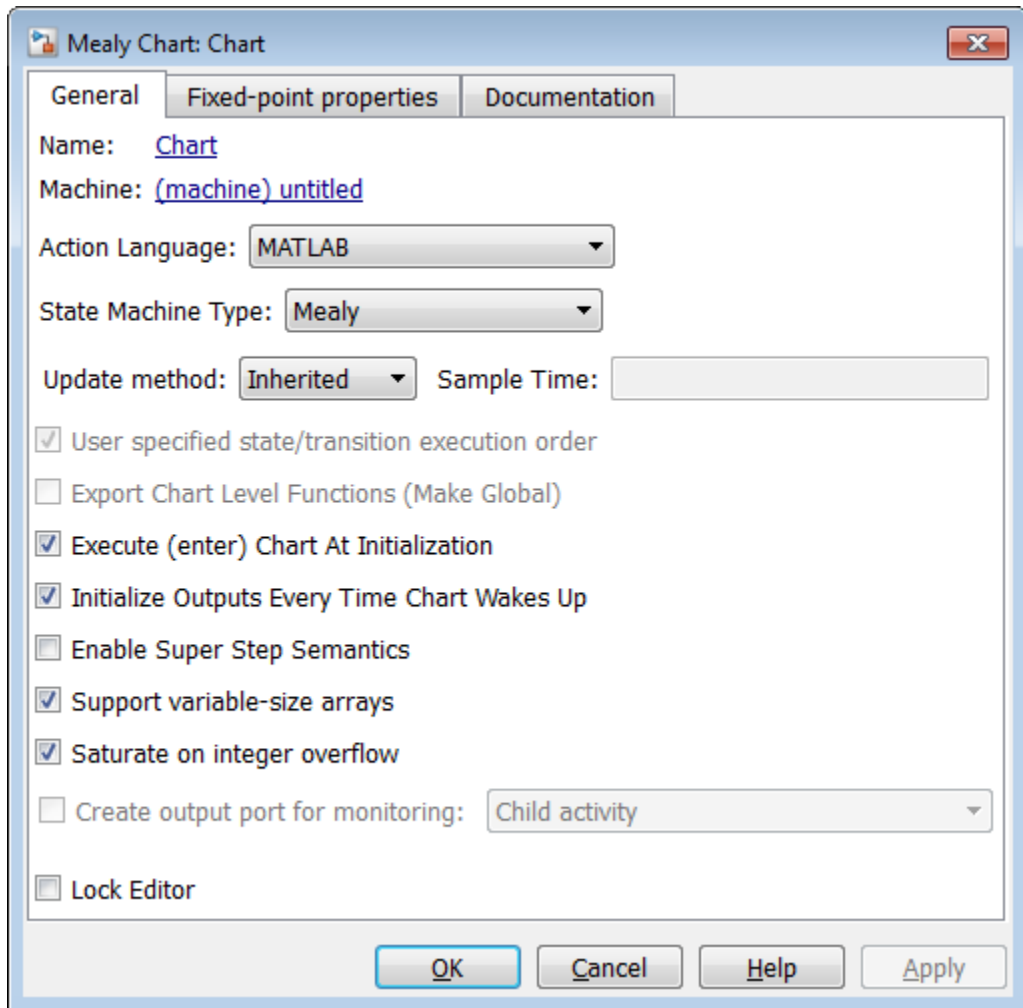
assign y = y_1;
```

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:

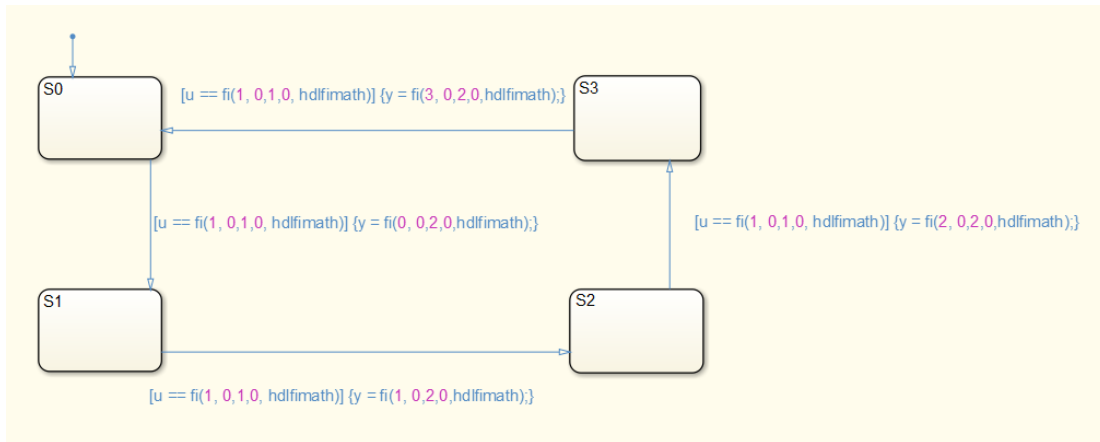
- The chart must meet the general code generation requirements as described in [Chart](#).
- Actions must be associated with inner and outer transitions only.
- For better synthesis results and more readable HDL code, we recommend enabling the chart property **Initialize Outputs Every Time Chart Wakes 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
  
```

```
        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”

Design Patterns Using Advanced Chart Features

In this section...
“Temporal Logic” on page 18-12
“Graphical Function” on page 18-14
“Hierarchy and Parallelism” on page 18-16
“Stateless Charts” on page 18-16
“Truth Tables” on page 18-18

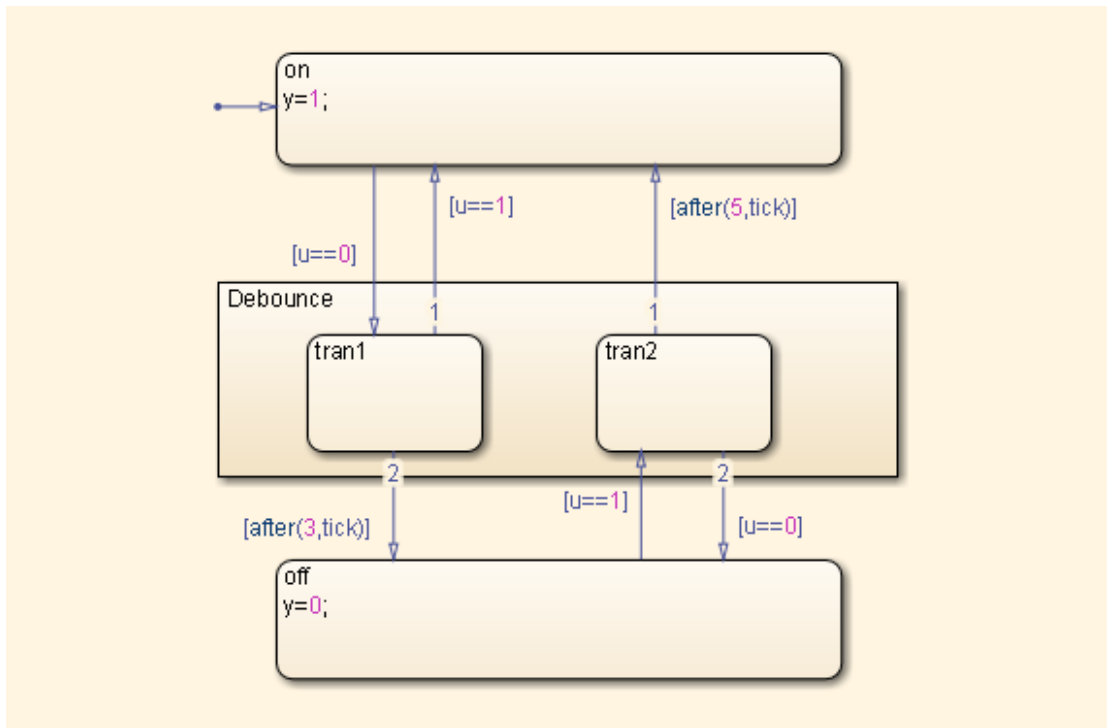
Temporal Logic

Stateflow temporal logic operators (such as `after`, `before`, or `every`) are Boolean operators that operate on recurrence counts of Stateflow events. Temporal logic operators can appear only in conditions on transitions that originate from states, and in state actions. Although temporal logic does not introduce new events into a Stateflow model, it is useful to think of the change of value of a temporal logic condition as an event. You can use temporal logic operators in many cases where a counter is required. A common use case would be to use temporal logic to implement a time-out counter.

Note: Absolute-time temporal logic is not supported for HDL code generation.

For detailed information about temporal logic, see “Control Chart Execution Using Temporal Logic”.

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.



The following code excerpt shows VHDL code generated from this chart.

```

Chart_1_output : PROCESS (is_Chart, u, temporalCounter_i1, y_reg)
  VARIABLE temporalCounter_i1_temp : unsigned(7 DOWNTO 0);
  BEGIN
    temporalCounter_i1_temp := temporalCounter_i1;
    is_Chart_next <= is_Chart;
    y_reg_next <= y_reg;
    IF temporalCounter_i1 < 7 THEN
      temporalCounter_i1_temp := temporalCounter_i1 + 1;
    END IF;

    CASE is_Chart IS
      WHEN IN_tran1 =>
        IF u = 1.0 THEN
          is_Chart_next <= IN_on;
          y_reg_next <= 1.0;
        ELSIF temporalCounter_i1_temp >= 3 THEN
          is_Chart_next <= IN_off;
          y_reg_next <= 0.0;
        END IF;
      WHEN IN_tran2 =>
        IF temporalCounter_i1_temp >= 5 THEN
          is_Chart_next <= IN_on;
          y_reg_next <= 1.0;
        END IF;
    END CASE;
  END PROCESS;

```

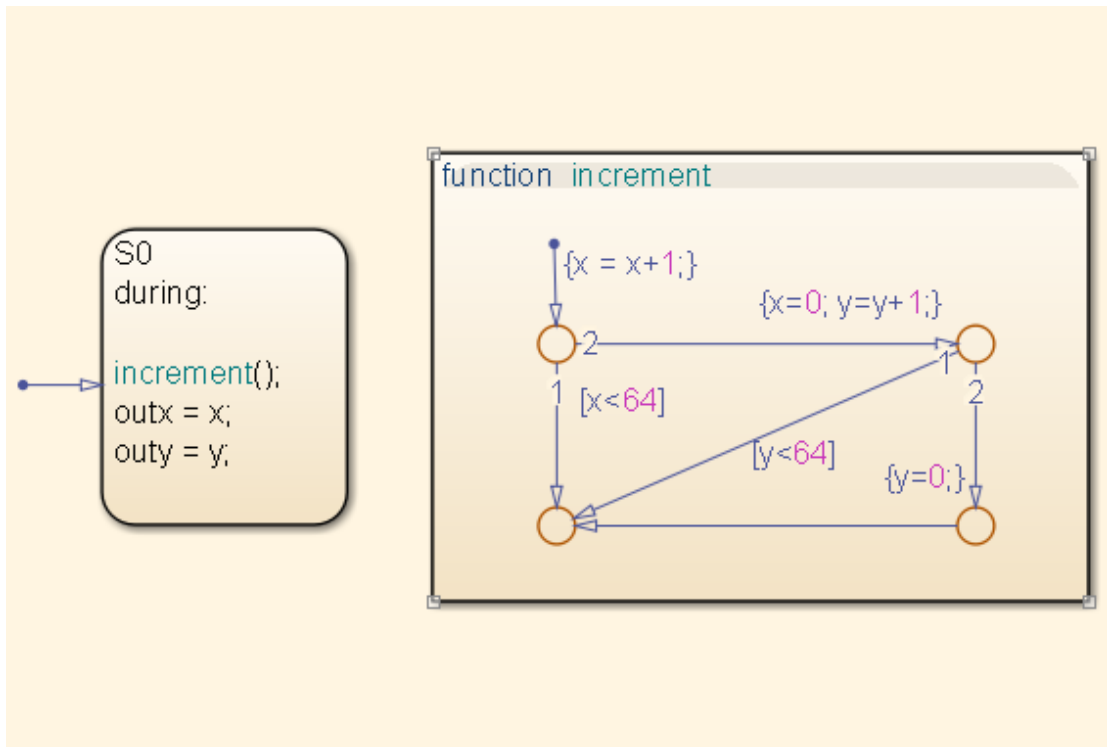
```
    ELSIF u = 0.0 THEN
        is_Chart_next <= IN_off;
        y_reg_next <= 0.0;
    END IF;
    WHEN IN_off =>
        IF u = 1.0 THEN
            is_Chart_next <= IN_tran2;
            temporalCounter_i1_temp := to_unsigned(0, 8);
        END IF;
        WHEN OTHERS =>
            IF u = 0.0 THEN
                is_Chart_next <= IN_tran1;
                temporalCounter_i1_temp := to_unsigned(0, 8);
            END IF;
        END CASE;

    temporalCounter_i1_next <= temporalCounter_i1_temp;
END PROCESS Chart_1_output;
```

Graphical Function

A graphical function is a function defined graphically by a flow diagram. Graphical functions reside in a chart along with the diagrams that invoke them. Like MATLAB functions and C functions, graphical functions can accept arguments and return results. Graphical functions can be invoked in transition and state actions.

The following figure shows a graphical function that implements a 64-by-64 counter.



The following code excerpt shows VHDL code generated for this graphical function.

```
x64_counter_sf : PROCESS (x, y, outx_reg, outy_reg)
-- local variables
  VARIABLE x_temp : unsigned(7 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 “Stateflow Hierarchy of Objects” and “Execution Order for Parallel States”.

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.

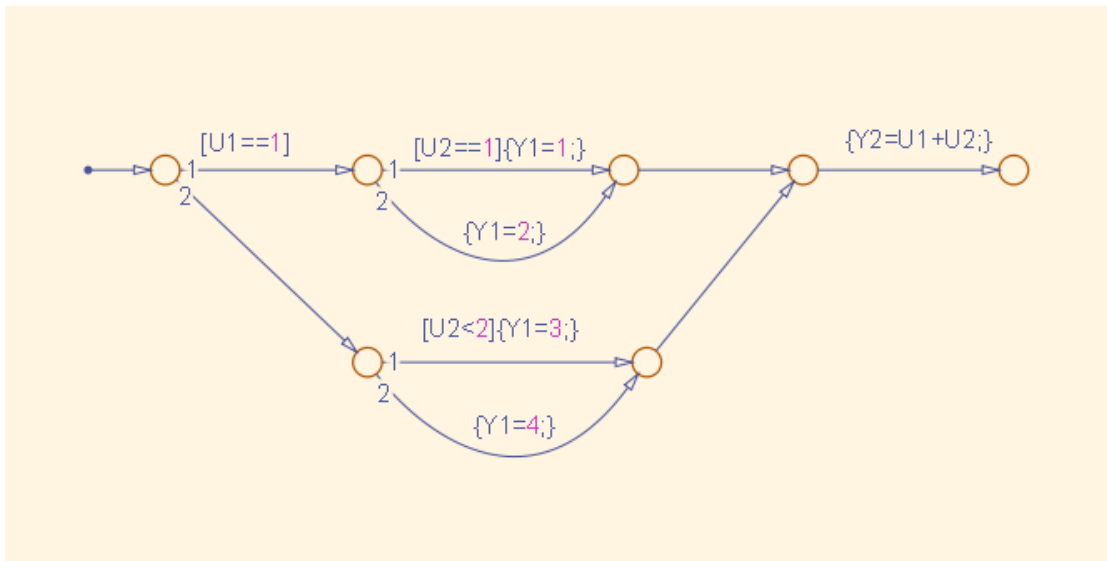
```
if(U1==1) {
    if(U2==1) {
        Y = 1;
    }else{
        Y = 2;
    }
}
```

```

    }
  }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;
    ELSE
      Y1_reg_next <= to_unsigned(3, 8);
    END IF;
  END PROCESS;

```

```
        END IF;
    ELSIF unsigned(U2) < to_unsigned(2, 8) THEN
        Y1_reg_next <= to_unsigned(3, 8);
    ELSE
        Y1_reg_next <= to_unsigned(4, 8);
    END IF;

    Y2_reg_next <= tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(unsigned(U1), 9),10)
    + tmw_to_unsigned(tmw_to_unsigned(unsigned(U2), 9), 10), 8);
END PROCESS Chart;
```

Truth Tables

HDL Coder supports HDL code generation for:

- Truth Table functions within a Stateflow chart
- Truth Table blocks in Simulink models

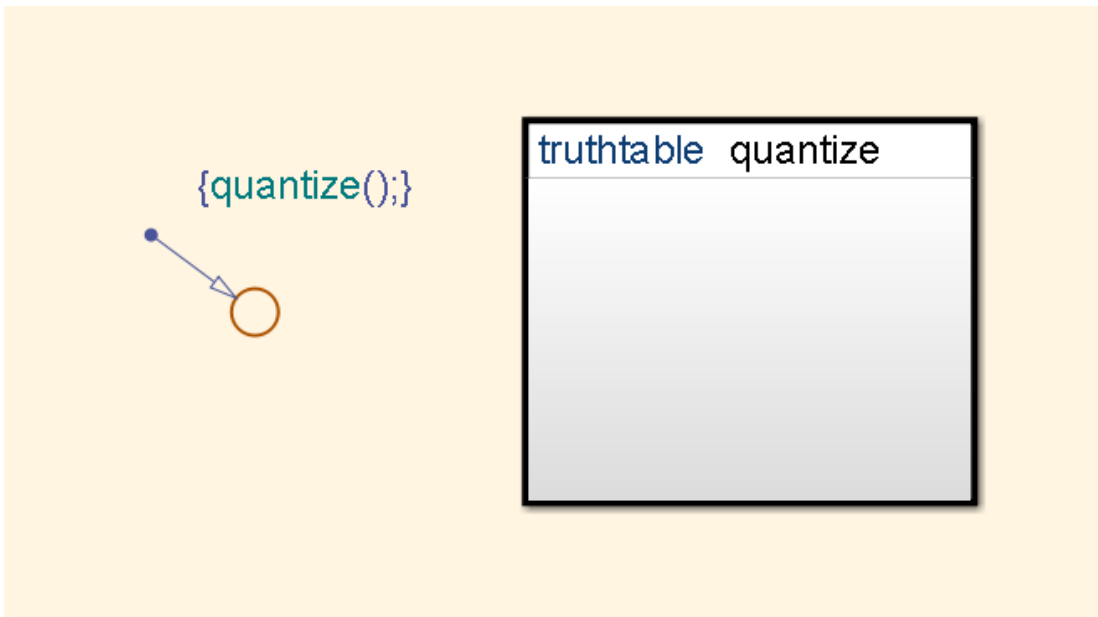
This section examines a Truth Table function in a chart, and the VHDL code generated for the chart.

Truth Tables are well-suited for implementing compact combinatorial logic. A typical application for Truth Tables is to implement nonlinear quantization or threshold logic. Consider the following logic:

```
Y = 1 when 0  <= U <= 10
Y = 2 when 10 < U <= 17
Y = 3 when 17 < U <= 45
Y = 4 when 45 < U <= 52
Y = 5 when 52 < U
```

A stateless chart with a single call to a Truth Table function can represent this logic succinctly.

The following figure shows the `quantizer` chart, containing the Truth Table.



The following figure shows the threshold logic, as displayed in the Truth Table Editor.

Stateflow (truth table) sf_truth_table/quantizer/quantizer.quantize

File Edit Settings Add Help

Condition Table

	Description	Condition	D1	D2	D3	D4	D5
1		$U \leq 10$	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		$Y = 5$

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).

Generating HDL Code with the MATLAB Function Block

- “HDL Applications for the MATLAB Function Block” on page 19-2
- “Viterbi Decoder with the MATLAB Function Block” on page 19-3
- “Code Generation from a MATLAB Function Block” on page 19-4
- “Generate Instantiable Code for Functions” on page 19-22
- “MATLAB Function Block Design Patterns for HDL” on page 19-24
- “Design Guidelines for the MATLAB Function Block” on page 19-35
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 19-41

HDL Applications for the MATLAB Function Block

The MATLAB Function block contains a MATLAB function in a model. The function's inputs and outputs are represented by 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, the HDL Coder software generates two main HDL code 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.

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 deinterleavers
- Modulators and demodulators
- Multipath channel models; impairment models
- Timing recovery algorithms
- Viterbi algorithm; Maximum Likelihood Sequence Estimation (MLSE)
- Adaptive equalizer algorithms

Viterbi Decoder with the MATLAB Function Block

`hdlcoderviterbi2` models a Viterbi decoder, incorporating an MATLAB Function block for use in simulation and HDL code generation. To open the model, type the following at the MATLAB command prompt:

```
hdlcoderviterbi2
```

Code Generation from a MATLAB Function Block

In this section...

“Counter Model Using the MATLAB Function block” on page 19-4

“Setting Up” on page 19-6

“Creating the Model and Configuring General Model Settings” on page 19-7

“Adding a MATLAB Function Block to the Model” on page 19-8

“Set Fixed-Point Options for the MATLAB Function Block” on page 19-10

“Programming the MATLAB Function Block” on page 19-14

“Constructing and Connecting the DUT_eML_Block Subsystem” on page 19-15

“Compiling the Model and Displaying Port Data Types” on page 19-17

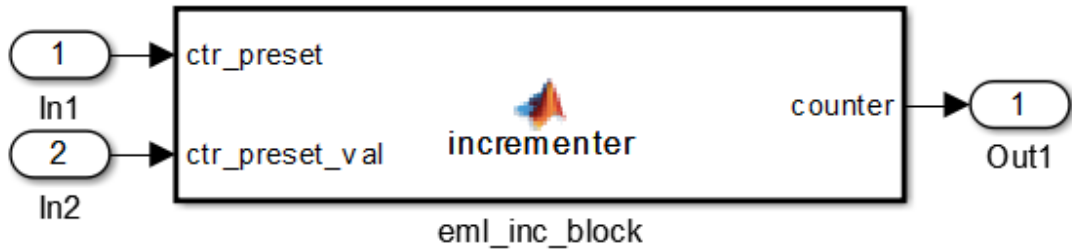
“Simulating the eml_hdl_incrementer_tut Model” on page 19-18

“Generating HDL Code” on page 19-19

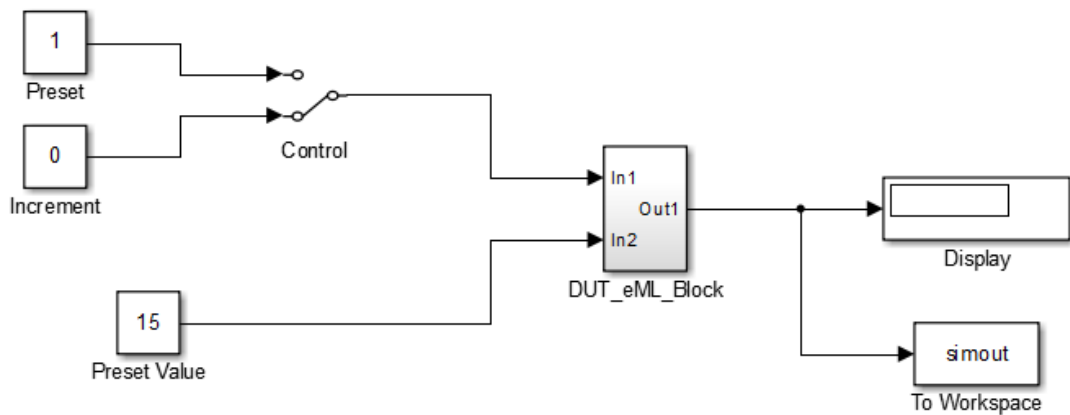
Counter Model Using the MATLAB Function block

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 subfolders 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.m` command. The `hdlsetup` command uses the `set_param` function to set up models for HDL code generation quickly and consistently. See “Initializing Model Parameters with `hdlsetup`” 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 em1_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 19-8.
- 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.

MATLAB Function: eml_inc_block

Name: [eml_inc_block](#)

Update method: Sample Time:

Support variable-size arrays

Saturate on integer overflow

Lock Editor

Treat these inherited Simulink signal types as fi objects:

MATLAB Function block **fimath**

Same as MATLAB Specify Other

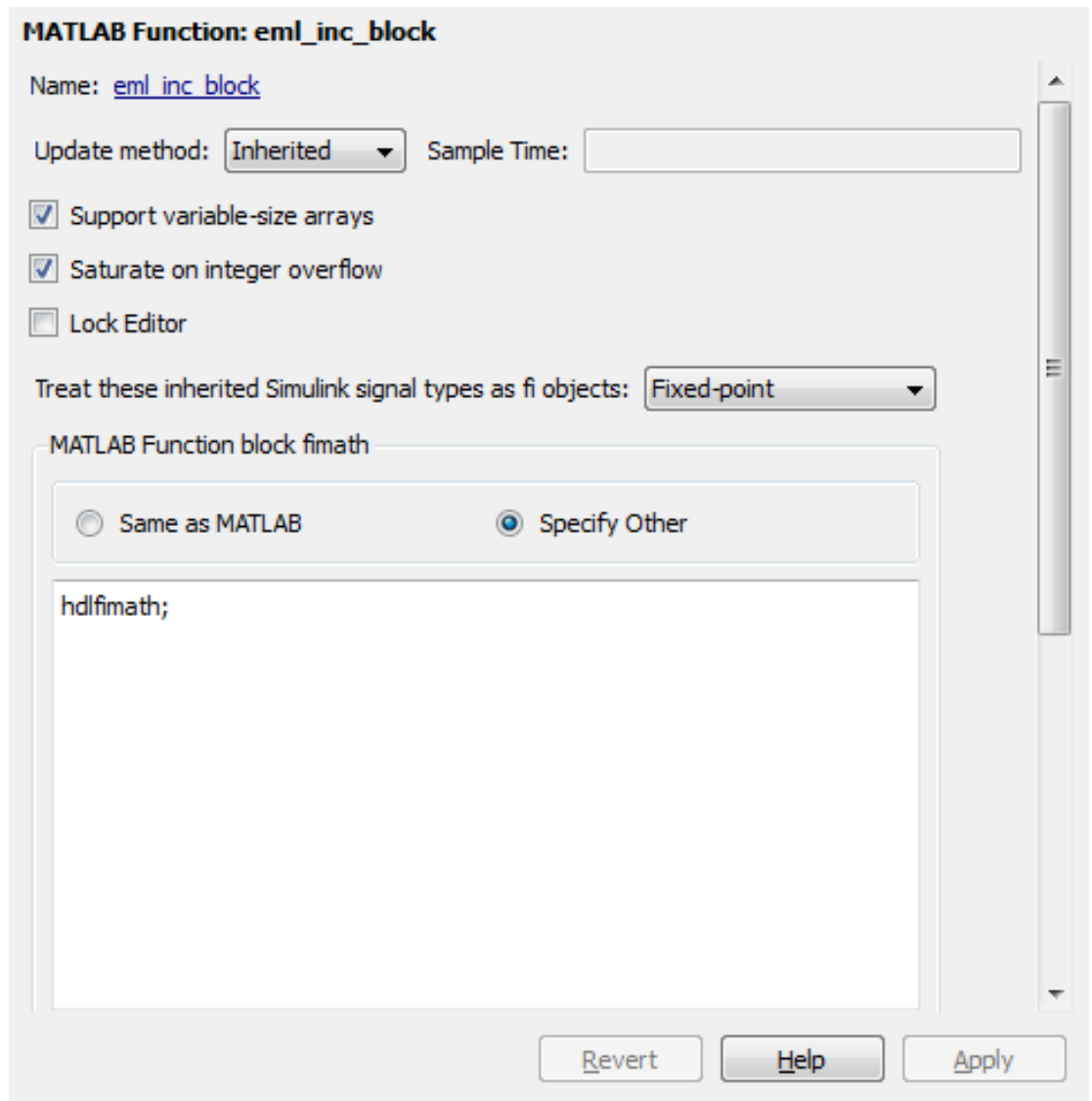
```
fimath('RoundMode', 'nearest',...  
'OverflowMode', 'saturate',...  
'ProductMode', 'FullPrecision',...  
'MaxProductWordLength', 128,...  
'SumMode', 'FullPrecision',...  
'MaxSumWordLength', 128)
```

- 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.

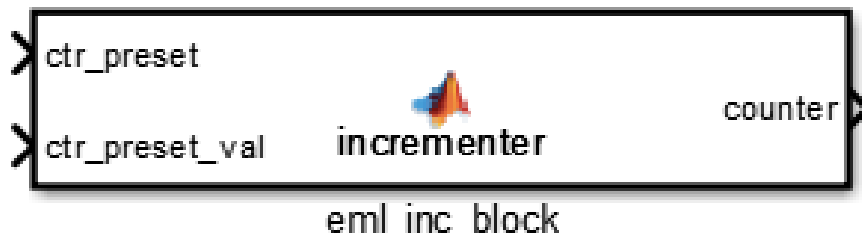
Programming the MATLAB Function Block

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 19-8.
- 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 19-6, 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 19-14 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:

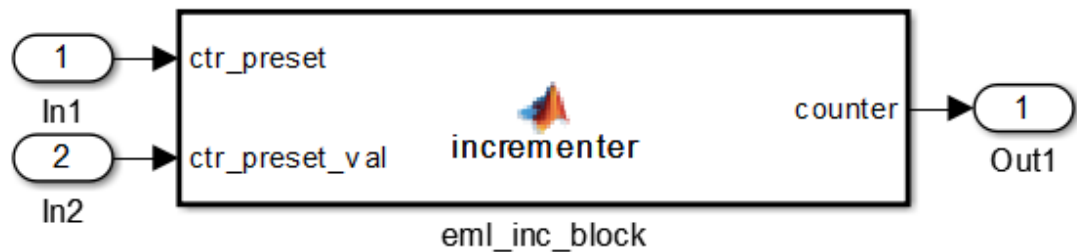
- 1 Click the `incrementer` function block.
- 2 Select **Diagram > Subsystem & Model Reference > Create Subsystem from Selection**.

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.

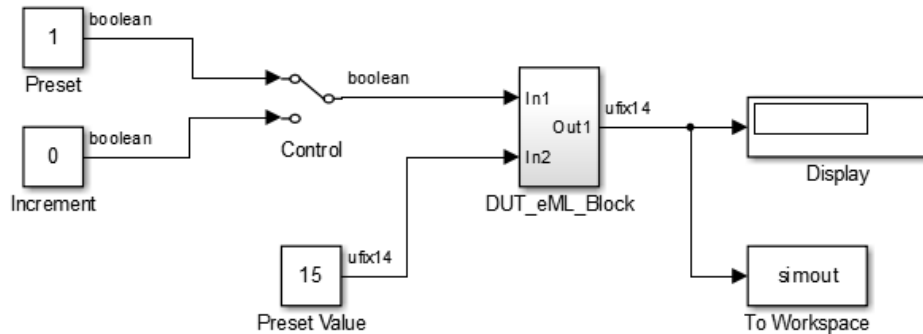
The build process builds an S-function for use in simulation. The build process includes generation of C code for the S-function. The code generation messages you see during the build process refer to generation of C code, not HDL code generation.

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.

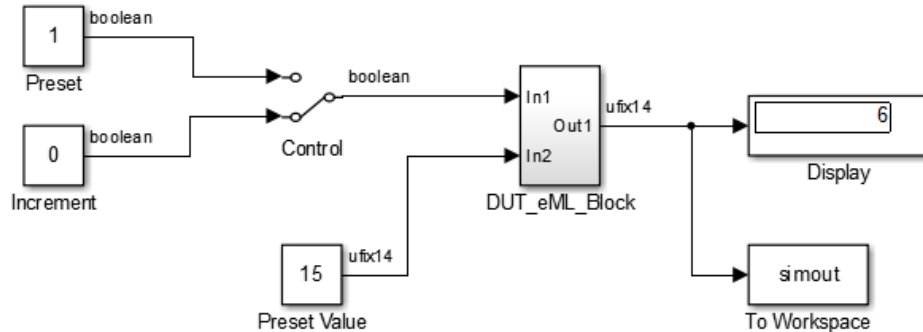


3 Save the model.

Simulating the eml_hdl_incrementer_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 **OK**.

Generating VHDL Code

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 15-38).
- 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.

Generate Instantiable Code for Functions

In this section...

“How To Generate Instantiable Code for Functions” on page 19-22

“Generate Code Inline for Specific Functions” on page 19-23

“Limitations for Instantiable Code Generation for Functions” on page 19-23

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.

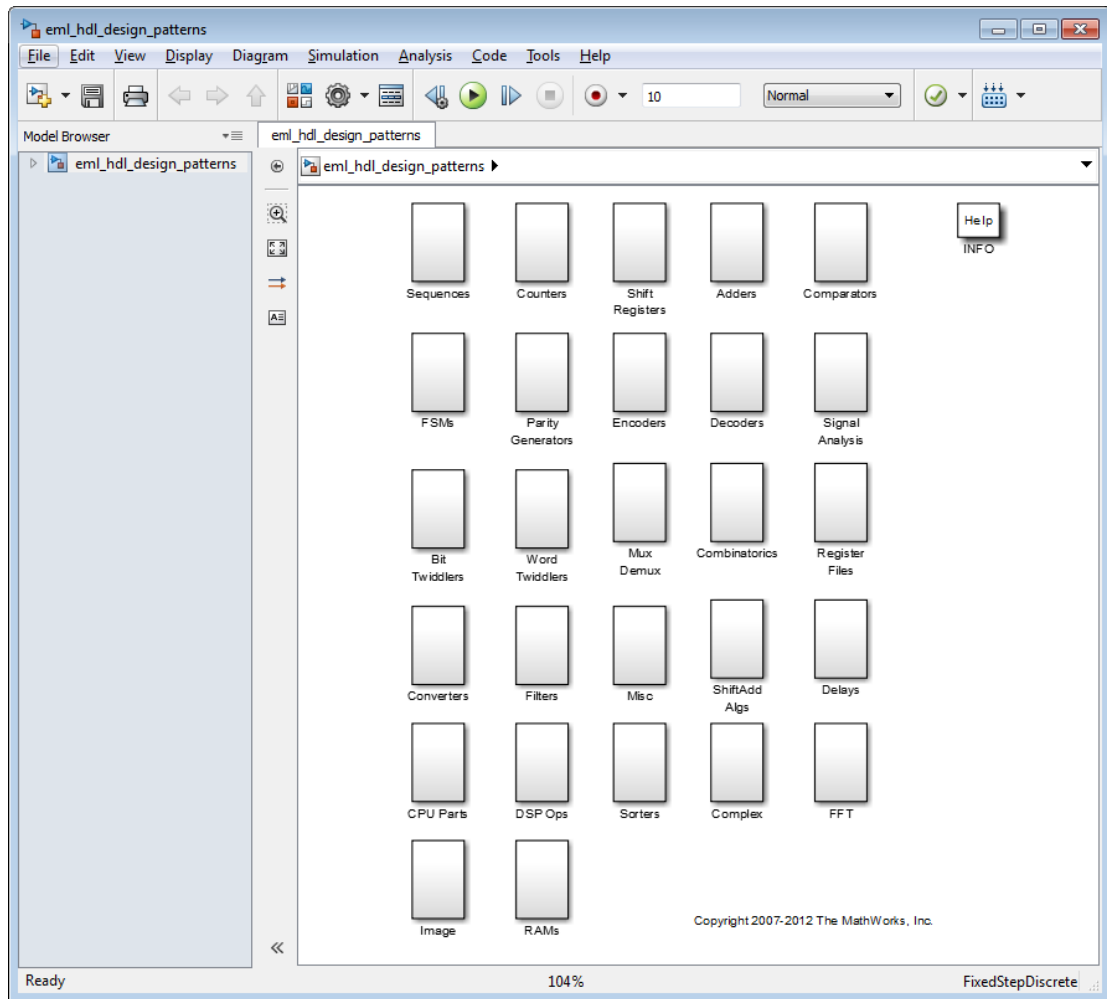
If you enable `InstantiateFunctions`, `UseMatrixTypesInHDL` has no effect.

MATLAB Function Block Design Patterns for HDL

In this section...
“The eml_hdl_design_patterns Library” on page 19-24
“Efficient Fixed-Point Algorithms” on page 19-26
“Model State Using Persistent Variables” on page 19-29
“Creating Intellectual Property with the MATLAB Function Block” on page 19-30
“Nontunable Parameter Arguments” on page 19-31
“Modeling Control Logic and Simple Finite State Machines” on page 19-31
“Modeling Counters” on page 19-32
“Modeling Hardware Elements” on page 19-33

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”.

Efficient Fixed-Point Algorithms

The MATLAB Function block supports fixed point arithmetic 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 (sfixed7_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, the operation

```
expr = (a*b) - (a+b);
```

is broken down internally into the following substeps:

```
1  tmul = a * b;
2  tadd = a + b;
3  tsub = tmul - tadd;
4  y = tsub;
```

Based on the `fimath` settings (see “Design Guidelines for the MATLAB Function Block” on page 19-35) 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', substep 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 substep 3 is computed as `sfix11_En5`. Accordingly, substep 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.

VHDL code excerpt:

```
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) = "0111111"))
    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;
```

Verilog code excerpt:

```
//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 coder 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 `sfix6`. `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 DOWNTO 0);
    y : OUT std_logic_vector(15 DOWNTO 0));
END Unit_Delay;
```

```
ARCHITECTURE fsm_SFHDL 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.

Only *nontunable* parameters are supported for HDL code generation. If you declare parameter arguments in MATLAB Function block code that is intended for HDL code generation, be sure to clear the **Tunable** option for each such parameter argument.

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_1by32** 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_1by32` block.

```
shift_reg_1by32 : PROCESS (shift, sr_in, sr)
    BEGIN
        sr_next <= sr;
        -- MATLAB Function Function 'Subsystem/shift_reg_1by32': '<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 DOWNTO 0) & sr_in;
        END IF;

    END PROCESS shift_reg_1by32;
```

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.

Design Guidelines for the MATLAB Function Block

In this section...

“Introduction” on page 19-35

“Use Compiled External Functions With MATLAB Function Blocks” on page 19-35

“Build the MATLAB Function Block Code First” on page 19-35

“Use the hdlfmath Utility for Optimized FIMATH Settings” on page 19-36

“Use Optimal Fixed-Point Option Settings” on page 19-38

“Set the Output Data Type of MATLAB Function Blocks Explicitly” on page 19-40

Introduction

This section describes recommended practices when using the MATLAB Function block for HDL code generation.

By setting MATLAB Function block options as described in this section, you can significantly increase the efficiency of generated HDL code. See “Set Fixed-Point Options for the MATLAB Function Block” on page 19-10 for an example.

Use Compiled External Functions With MATLAB Function Blocks

The HDL Coder software supports HDL code generation from MATLAB Function blocks that include compiled external functions. This feature enables you to write reusable MATLAB code and call it from multiple MATLAB Function blocks.

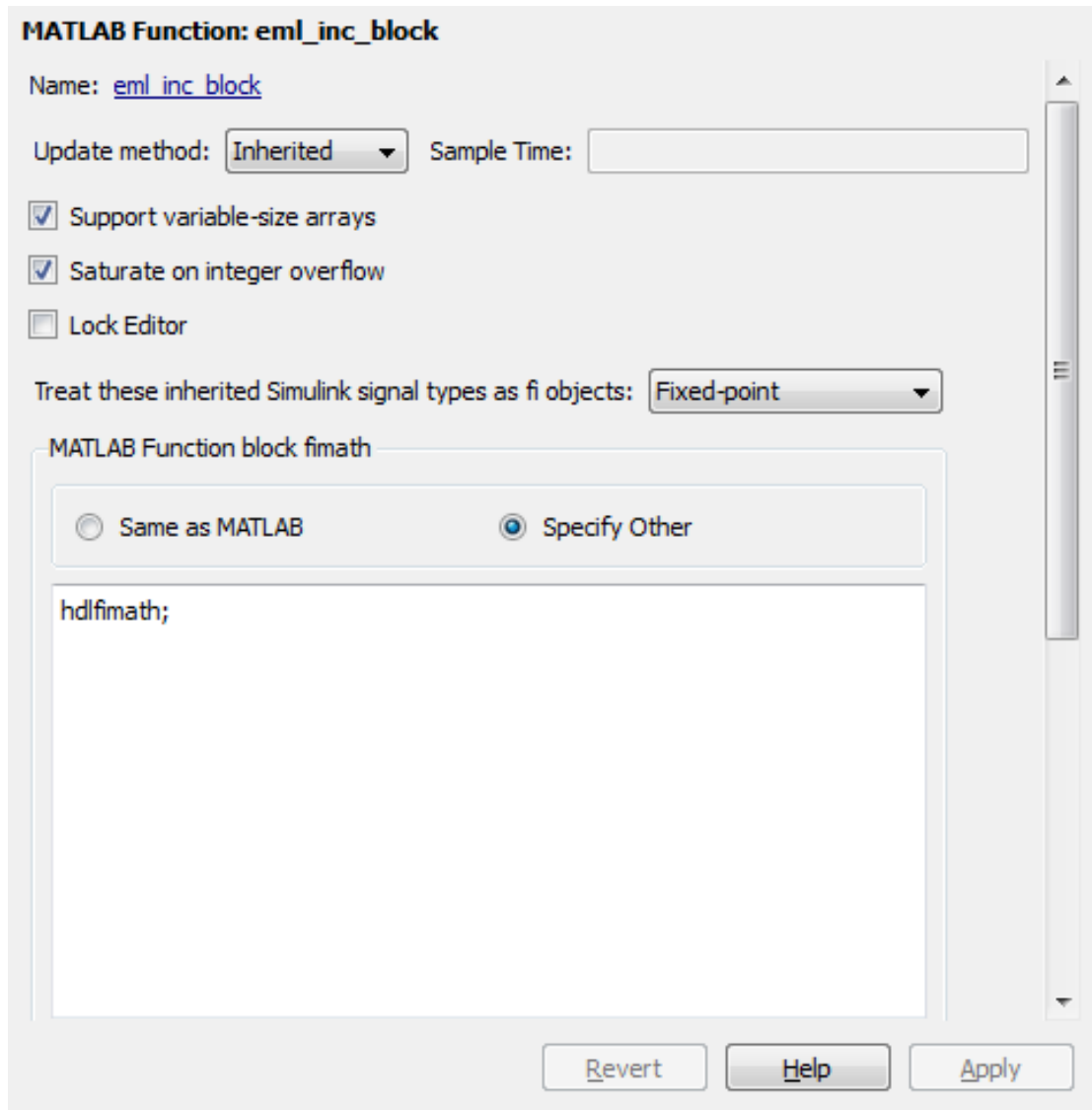
Such functions must be defined in files that are on the MATLAB Function block path. Use the `##codegen` compilation directive to indicate that the MATLAB code is suitable for code generation. See “Function Definition” for information on how to create, compile, and invoke external functions.

Build the MATLAB Function Block Code First

Before generating HDL code for a subsystem containing a MATLAB Function block, build the MATLAB Function block code to check for errors. To build the code, 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. Replace the default **MATLAB Function block `fimath`** specification with a call to the `hdlfimath` function, as shown in the following figure.



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);
```

Note: Use of 'floor' rounding mode for signed integer division will cause an error at code generation time. The HDL division operator does not support 'floor' rounding mode. Use 'round' mode, or else change the signed integer division operations to unsigned integer division.

Note: 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, as shown in the following figure.

MATLAB Function: eml_inc_block

Name: [eml_inc_block](#)

Update method: Inherited Sample Time:

Support variable-size arrays

Saturate on integer overflow

Lock Editor

Treat these inherited Simulink signal types as fi objects: Fixed-point

MATLAB Function block fimath

Same as MATLAB Specify Other

```
fimath('RoundMode', 'nearest', ...  
'OverflowMode', 'saturate', ...  
'ProductMode', 'FullPrecision', ...  
'MaxProductWordLength', 128, ...  
'SumMode', 'FullPrecision', ...  
'MaxSumWordLength', 128)
```

Revert Help Apply

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.

Distributed Pipeline Insertion for MATLAB Function Blocks

In this section...

“Overview” on page 19-41

“Distributed Pipelining in a Multiplier Chain” on page 19-41

Overview

Distributed pipeline insertion is a special optimization for HDL code generated from MATLAB Function blocks or Stateflow charts. Distributed pipeline insertion lets you achieve higher clock rates in your HDL applications, at the cost of some amount of latency caused by the introduction of pipeline registers.

For general information on distributed pipeline insertion, including limitations, see “DistributedPipelining”.

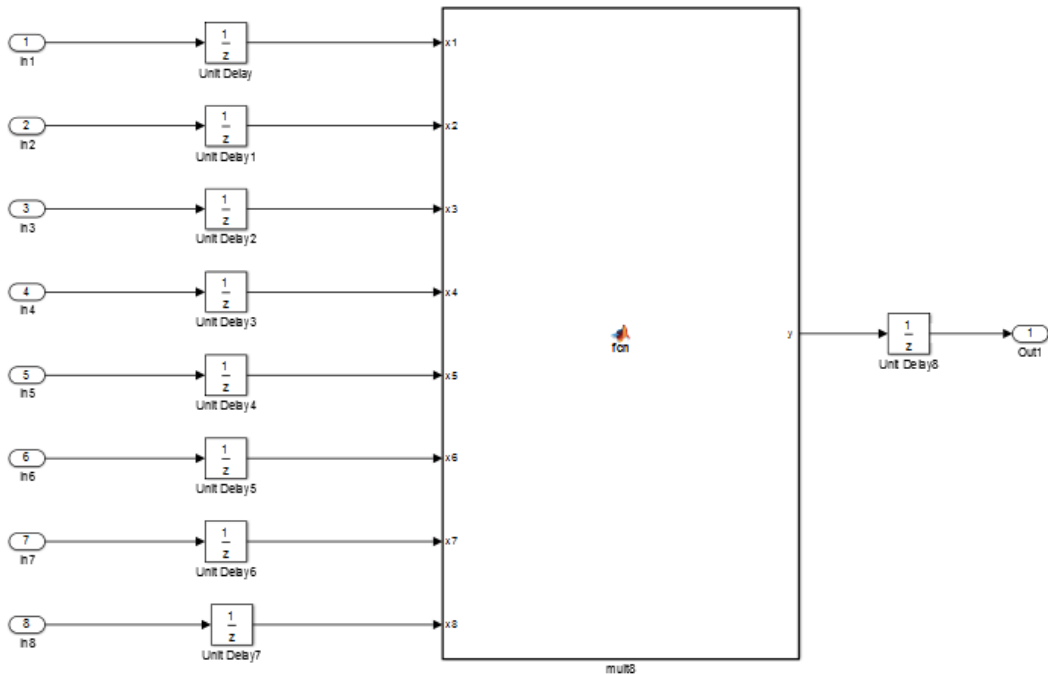
Distributed Pipelining in a Multiplier Chain

This example shows distributed pipeline insertion in a simple model that implements a chain of 5 multiplications.

To open the model, enter the following:

```
mpipe_multichain
```

The root level model contains a subsystem `multi_chain`. The `multi_chain` subsystem functions as the device under test (DUT) from which to generate HDL code. The subsystem drives a MATLAB Function block, `mult8`. The following figure shows the subsystem.



The following shows a chain of multiplications as coded in the `mul8` MATLAB Function block:

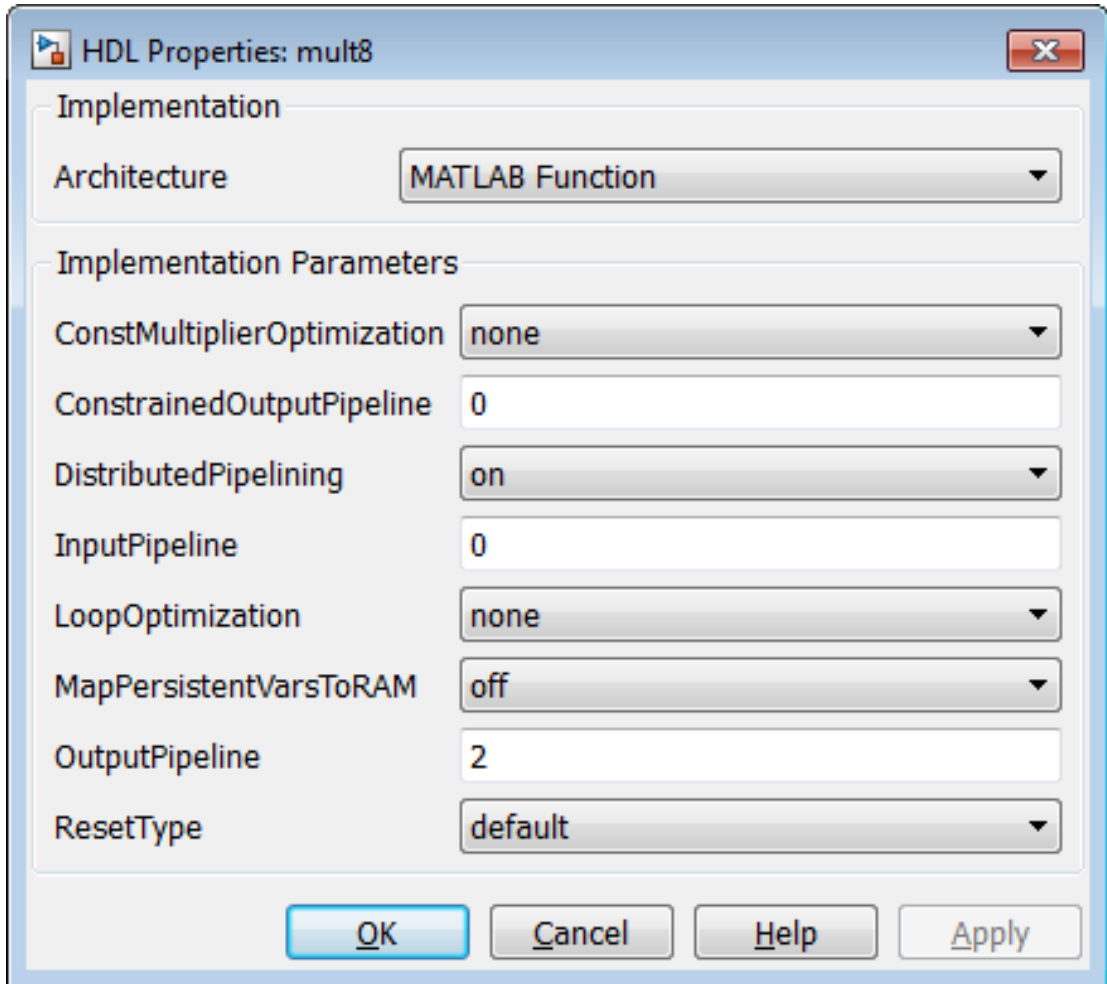
```
function y = fcn(x1,x2,x3,x4,x5,x6,x7,x8)
% A chained multiplication:
% y = (x1*x2)*(x3*x4)*(x5*x6)*(x7*x8)

y1 = x1 * x2;
y2 = x3 * x4;
y3 = x5 * x6;
y4 = x7 * x8;

y5 = y1 * y2;
y6 = y3 * y4;

y = y5 * y6;
```

To apply distributed pipeline insertion to this block, use the HDL Properties dialog box for the `mult8` block. Specify generation of two pipeline stages for the MATLAB Function block, and enable the distributed pipeline optimization:



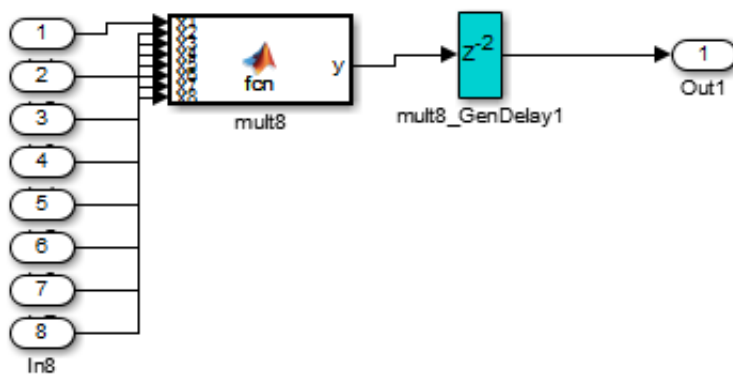
In the Configuration Parameters dialog box, the top-level **HDL Code Generation** options specify that:

- VHDL code is generated from the subsystem `mpipe_multichain/mult_chain`.

- HDL Coder will generate code and display the generated model.

The insertion of two pipeline stages into the generated HDL code results in a latency of two clock cycles. In the generated model, a delay of two clock cycles is inserted before the output of the `mpipe_multchain/mult_chain/mult8` subsystem so that simulations of the model reflect the behavior of the generated HDL code. The following figure shows the inserted Delay block.

gm_mpipe_multchain_vnl ▶ mult_chain ▶ mult8 ▶ mult8



The following listing shows the complete architecture section of the generated code. Comments generated by HDL Coder indicate the pipeline register definitions.

```
ARCHITECTURE fsm_SFHDL OF mult8 IS
```

```

SIGNAL pipe_var_0_1 : signed(7 DOWNTO 0); -- Pipeline reg from stage 0 to stage 1
SIGNAL b_pipe_var_0_1 : signed(7 DOWNTO 0); -- Pipeline reg from stage 0 to stage 1
SIGNAL c_pipe_var_0_1 : signed(7 DOWNTO 0); -- Pipeline reg from stage 0 to stage 1
SIGNAL d_pipe_var_0_1 : signed(7 DOWNTO 0); -- Pipeline reg from stage 0 to stage 1
SIGNAL pipe_var_1_2 : signed(7 DOWNTO 0); -- Pipeline reg from stage 1 to stage 2
SIGNAL b_pipe_var_1_2 : signed(7 DOWNTO 0); -- Pipeline reg from stage 1 to stage 2
SIGNAL pipe_var_0_1_next : signed(7 DOWNTO 0);
SIGNAL b_pipe_var_0_1_next : signed(7 DOWNTO 0);
SIGNAL c_pipe_var_0_1_next : signed(7 DOWNTO 0);
SIGNAL d_pipe_var_0_1_next : signed(7 DOWNTO 0);
SIGNAL pipe_var_1_2_next : signed(7 DOWNTO 0);

```

```

SIGNAL b_pipe_var_1_2_next : signed(7 DOWNTO 0);
SIGNAL y1 : signed(7 DOWNTO 0);
SIGNAL y2 : signed(7 DOWNTO 0);
SIGNAL y3 : signed(7 DOWNTO 0);
SIGNAL y4 : signed(7 DOWNTO 0);
SIGNAL y5 : signed(7 DOWNTO 0);
SIGNAL y6 : signed(7 DOWNTO 0);
SIGNAL mul_temp : signed(15 DOWNTO 0);
SIGNAL mul_temp_0 : signed(15 DOWNTO 0);
SIGNAL mul_temp_1 : signed(15 DOWNTO 0);
SIGNAL mul_temp_2 : signed(15 DOWNTO 0);
SIGNAL mul_temp_3 : signed(15 DOWNTO 0);
SIGNAL mul_temp_4 : signed(15 DOWNTO 0);
SIGNAL mul_temp_5 : signed(15 DOWNTO 0);

BEGIN
  initialize_mult8 : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      pipe_var_0_1 <= to_signed(0, 8);
      b_pipe_var_0_1 <= to_signed(0, 8);
      c_pipe_var_0_1 <= to_signed(0, 8);
      d_pipe_var_0_1 <= to_signed(0, 8);
      pipe_var_1_2 <= to_signed(0, 8);
      b_pipe_var_1_2 <= to_signed(0, 8);
    ELSIF clk'EVENT AND clk= '1' THEN
      IF clk_enable= '1' THEN
        pipe_var_0_1 <= pipe_var_0_1_next;
        b_pipe_var_0_1 <= b_pipe_var_0_1_next;
        c_pipe_var_0_1 <= c_pipe_var_0_1_next;
        d_pipe_var_0_1 <= d_pipe_var_0_1_next;
        pipe_var_1_2 <= pipe_var_1_2_next;
        b_pipe_var_1_2 <= b_pipe_var_1_2_next;
      END IF;
    END IF;
  END PROCESS initialize_mult8;

  -- This block supports an embeddable subset of the MATLAB language.
  -- See the help menu for details.
  --y = (x1+x2)+(x3+x4)+(x5+x6)+(x7+x8);
  mul_temp <= signed(x1) * signed(x2);

  y1 <= "01111111" WHEN (mul_temp(15) = '0') AND (mul_temp(14 DOWNTO 7) /= "00000000")
    ELSE "10000000" WHEN (mul_temp(15) = '1') AND (mul_temp(14 DOWNTO 7) /= "11111111")
    ELSE mul_temp(7 DOWNTO 0);

  mul_temp_0 <= signed(x3) * signed(x4);

  y2 <= "01111111" WHEN (mul_temp_0(15) = '0') AND (mul_temp_0(14 DOWNTO 7) /= "00000000")
    ELSE "10000000" WHEN (mul_temp_0(15) = '1') AND (mul_temp_0(14 DOWNTO 7) /= "11111111")
    ELSE mul_temp_0(7 DOWNTO 0);

  mul_temp_1 <= signed(x5) * signed(x6);

  y3 <= "01111111" WHEN (mul_temp_1(15) = '0') AND (mul_temp_1(14 DOWNTO 7) /= "00000000")
    ELSE "10000000" WHEN (mul_temp_1(15) = '1') AND (mul_temp_1(14 DOWNTO 7) /= "11111111")
    ELSE mul_temp_1(7 DOWNTO 0);

  mul_temp_2 <= signed(x7) * signed(x8);

  y4 <= "01111111" WHEN (mul_temp_2(15) = '0') AND (mul_temp_2(14 DOWNTO 7) /= "00000000")
    ELSE "10000000" WHEN (mul_temp_2(15) = '1') AND (mul_temp_2(14 DOWNTO 7) /= "11111111")

```

```
ELSE mul_temp_2(7 DOWNT0 0);

mul_temp_3 <= pipe_var_0_1 * b_pipe_var_0_1;

y5 <= "01111111" WHEN (mul_temp_3(15) = '0') AND (mul_temp_3(14 DOWNT0 7) /= "00000000")
ELSE "10000000" WHEN (mul_temp_3(15) = '1') AND (mul_temp_3(14 DOWNT0 7) /= "11111111")
ELSE mul_temp_3(7 DOWNT0 0);

mul_temp_4 <= c_pipe_var_0_1 * d_pipe_var_0_1;

y6 <= "01111111" WHEN (mul_temp_4(15)='0') AND (mul_temp_4(14 DOWNT0 7) /= "00000000")
ELSE "10000000" WHEN (mul_temp_4(15) = '1') AND (mul_temp_4(14 DOWNT0 7) /= "11111111")
ELSE mul_temp_4(7 DOWNT0 0);

mul_temp_5 <= pipe_var_1_2 * b_pipe_var_1_2;

y <= "01111111" WHEN (mul_temp_5(15) = '0') AND (mul_temp_5(14 DOWNT0 7) /= "00000000")
ELSE "10000000" WHEN (mul_temp_5(15) = '1') AND (mul_temp_5(14 DOWNT0 7) /= "11111111")
ELSE std_logic_vector(mul_temp_5(7 DOWNT0 0));

b_pipe_var_1_2_next <= y6;
pipe_var_1_2_next <= y5;
d_pipe_var_0_1_next <= y4;
c_pipe_var_0_1_next <= y3;
b_pipe_var_0_1_next <= y2;
pipe_var_0_1_next <= y1;
END fsm_SFHDL;
```


Generating Scripts for HDL Simulators and Synthesis Tools

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 20-2
- “Structure of Generated Script Files” on page 20-3
- “Properties for Controlling Script Generation” on page 20-4
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 20-8
- “Add Synthesis Attributes” on page 20-16
- “Configure Synthesis Project Using Tcl Script” on page 20-17

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 20-4.
- 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 20-8.

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 strings to the script generator. Some of these format strings 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, script names are generated by appending a postfix string to the model or subsystem name *system*.

When you generate test bench code, script names are generated by appending 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 string 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 strings 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 substring **Init** in the property name.
- Properties that apply to the command-per-file phase (**Cmd**) are identified by the substring **Cmd** in the property name.
- Properties that apply to the termination (**Term**) phase are identified by the substring **Term** in the property name.

Property Name and Default	Description
Name: HDLCompileInit Default: 'vlib %s\n'	Format string passed to <code>fprintf</code> to write the Init section of the compilation script. The implicit argument is the contents of the <code>VHDLLibraryName</code> property, which defaults to 'work'. You can override the default Init string ('vlib work\n') by changing the value of <code>VHDLLibraryName</code> .
Name: HDLCompileVHDLCmd Default: 'vcom %s %s\n'	Format string passed to <code>fprintf</code> to write the Cmd 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 '' (the default).
Name: HDLCompileVerilogCmd Default: 'vlog %s %s\n'	Format string passed to <code>fprintf</code> to write the Cmd 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 '' (the default).
Name: HDLCompileTerm Default: ''	Format string passed to <code>fprintf</code> to write the termination portion of the compilation script.
Name: HDLSimInit Default:	Format string passed to <code>fprintf</code> to write the initialization section of the simulation script.

Property Name and Default	Description
<pre>['onbreak resume\n', ... 'onerror resume\n']</pre>	
<p>Name: HDLSimCmd</p> <p>Default: 'vsim -novopt %s.%s\n'</p>	<p>Format string passed to <code>fprintf</code> to write the simulation command.</p> <p>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'.</p> <p>The second implicit argument is the top-level module or entity name.</p>
<p>Name: HDLSimViewWaveCmd</p> <p>Default: 'add wave sim:%s\n'</p>	<p>Format string 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.</p>
<p>Name: HDLSimTerm</p> <p>Default: 'run -all\n'</p>	<p>Format string passed to <code>fprintf</code> to write the <code>Term</code> portion of the simulation script. The string is a synthesis project creation command.</p> <p>The content of the string is specific to the selected synthesis tool. See <code>HDLSynthTool</code>.</p>
<p>Name: HDLSynthInit</p>	<p>Format string passed to <code>fprintf</code> to write the <code>Init</code> section of the synthesis script.</p> <p>The content of the string is specific to the selected synthesis tool. See <code>HDLSynthTool</code>.</p>
<p>Name: HDLSynthCmd</p>	<p>Format string passed to <code>fprintf</code> to write the <code>Cmd</code> section of the synthesis script.</p> <p>The content of the string is specific to the selected synthesis tool. See <code>HDLSynthTool</code>.</p>
<p>Name: HDLSynthTerm</p>	<p>Format string passed to <code>fprintf</code> to write the <code>Term</code> 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`.

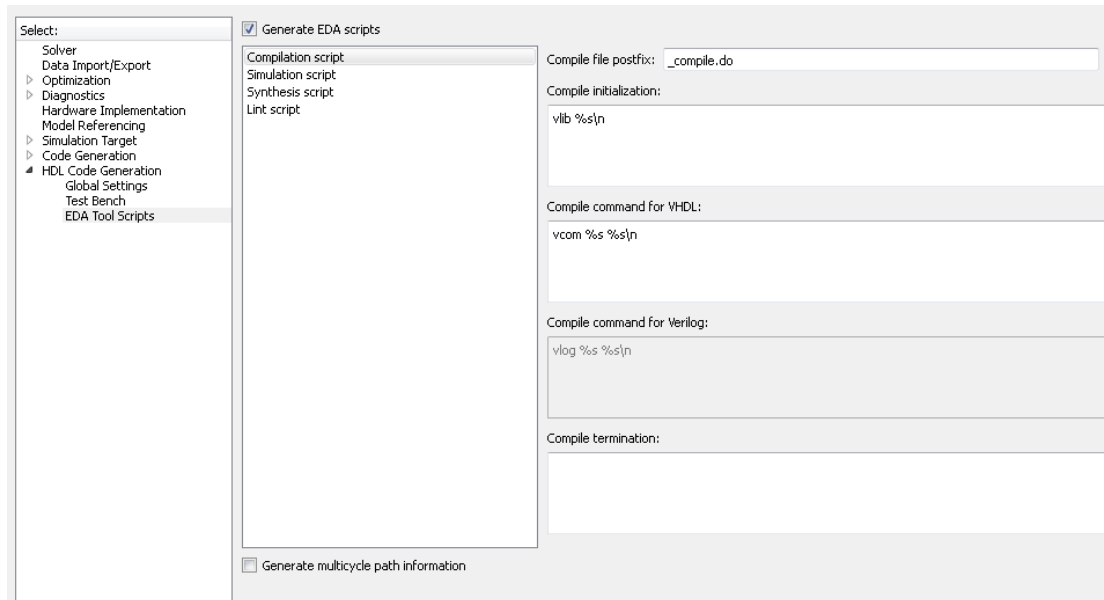
```
set src_dir "./hdlsrc"
set prj_dir "synprj"
file mkdir ../$prj_dir
cd ../$prj_dir
project new symmetric_fir.ise
xfile add ../$src_dir/symmetric_fir.vhd
project set family Virtex4
project set device xc4vsx35
project set package ff668
project set speed -10
process run "Synthesize - XST"
```

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 20-4.

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.

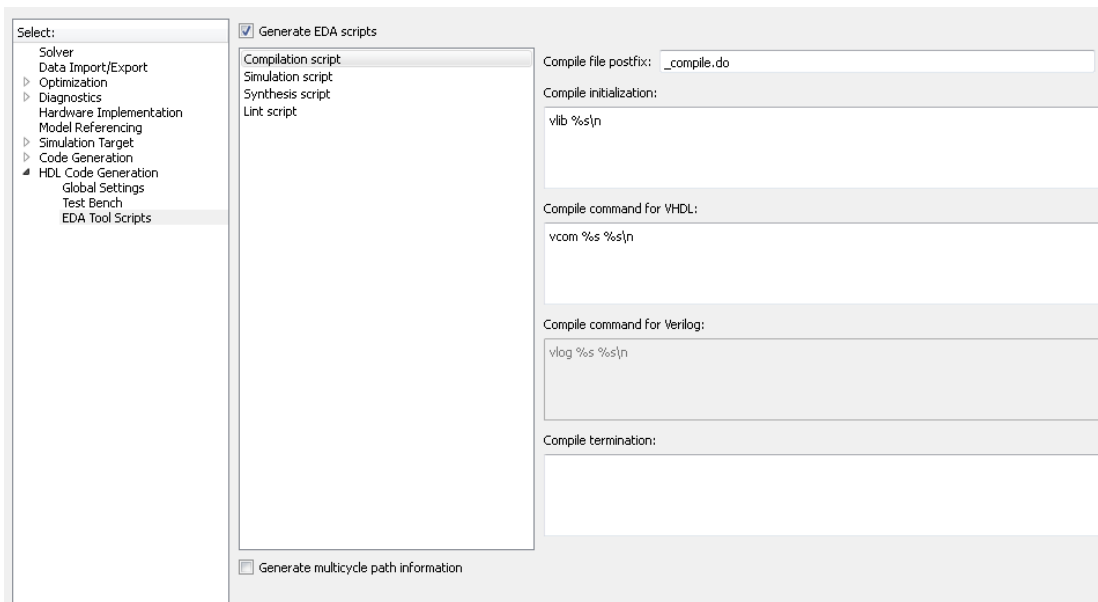
If you want to disable script generation, clear this check box and click **Apply**.

- 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 20-9 for further information.
- **Simulation script:** Options related to customizing scripts for HDL simulators. See “Simulation Script Options” on page 20-10 for further information.
- **Synthesis script:** Options related to customizing scripts for synthesis tools. See “Synthesis Script Options” on page 20-12 for further information.

Compilation Script Options

The following figure shows the **Compilation script** pane, with options set to their default values.



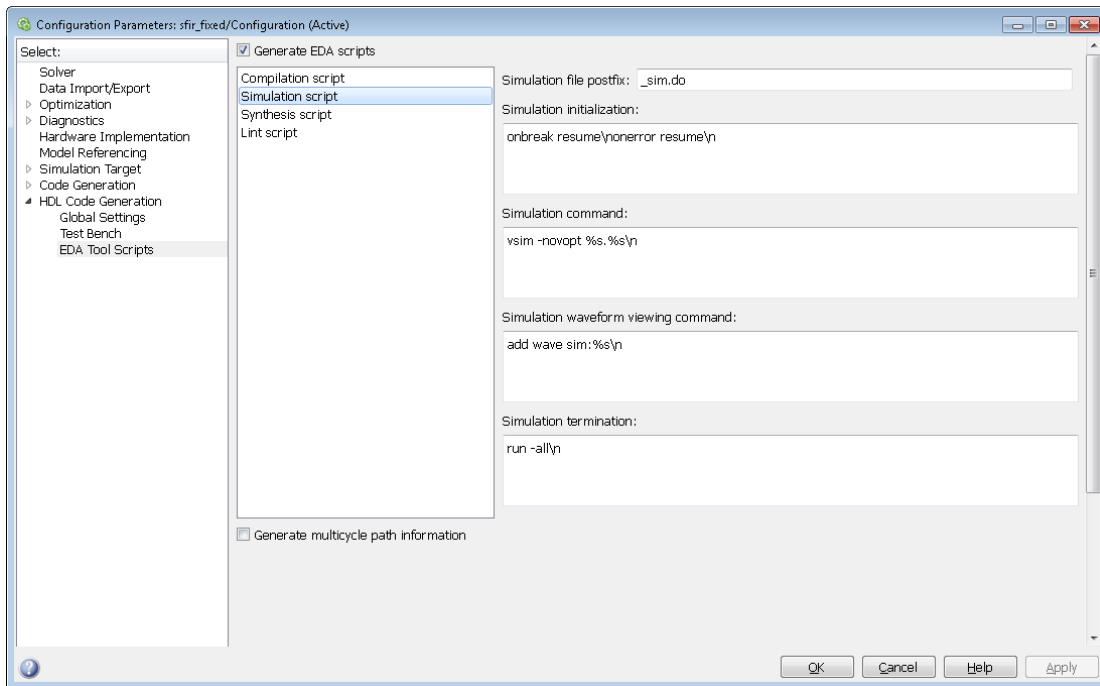
The following table summarizes the **Compilation script** options.

Option and Default	Description
Compile file postfix'	Postfix string appended to the DUT name or test bench name to form the script file name.

Option and Default	Description
'_compile.do'	
Name: Compile initialization Default: 'vlib %s\n'	Format string 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> string ('vlib work\n' by changing the value of <code>VHDLlibraryName</code> .
Name: Compile command for VHDL Default: 'vcom %s %s\n'	Format string 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 string 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 string 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.



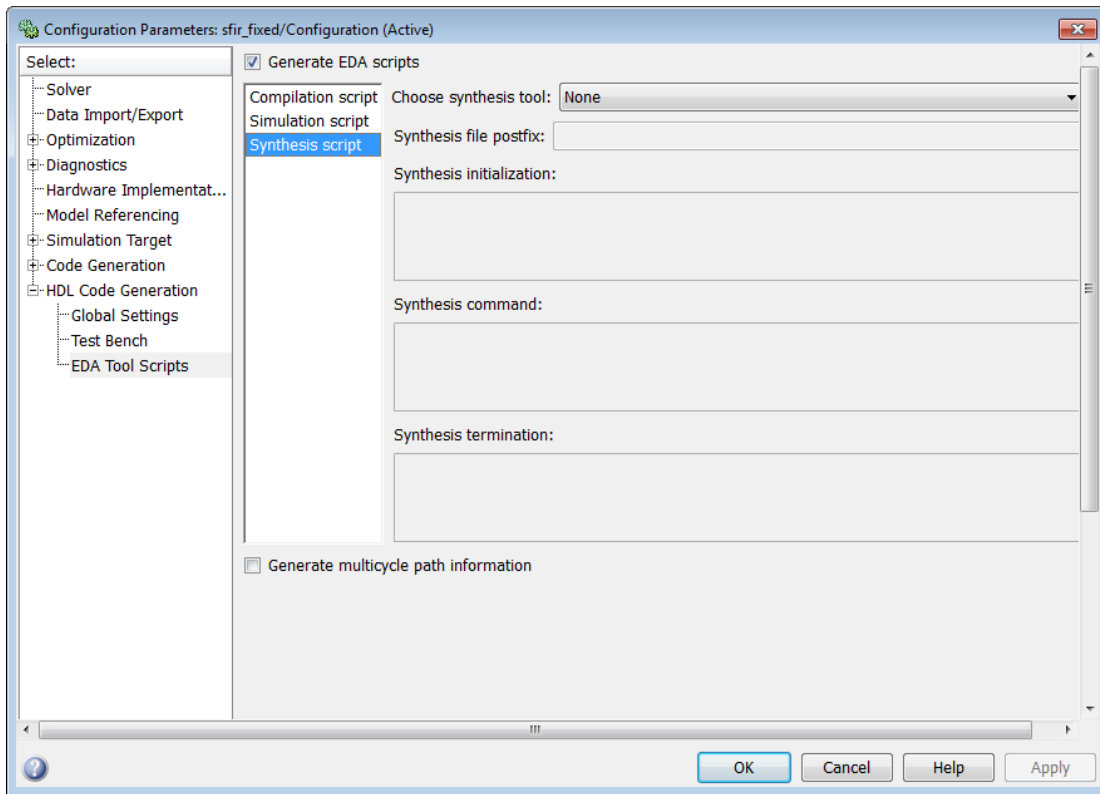
The following table summarizes the **Simulation script** options.

Option and Default	Description
Simulation file postfix Default: '_sim.do'	Postfix string appended to the model name or test bench name to form the simulation script file name.
Simulation initialization Default: ['onbreak resume\nonerror resume\n']	Format string passed to <code>fprintf</code> to write the initialization section of the simulation script.
Simulation command Default: 'vsim -novopt %s.%s\n'	Format string 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'.

Option and Default	Description
	The second implicit argument is the top-level module or entity name.
Simulation waveform viewing command Default: 'add wave sim:%s\n'	Format string 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 string 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.

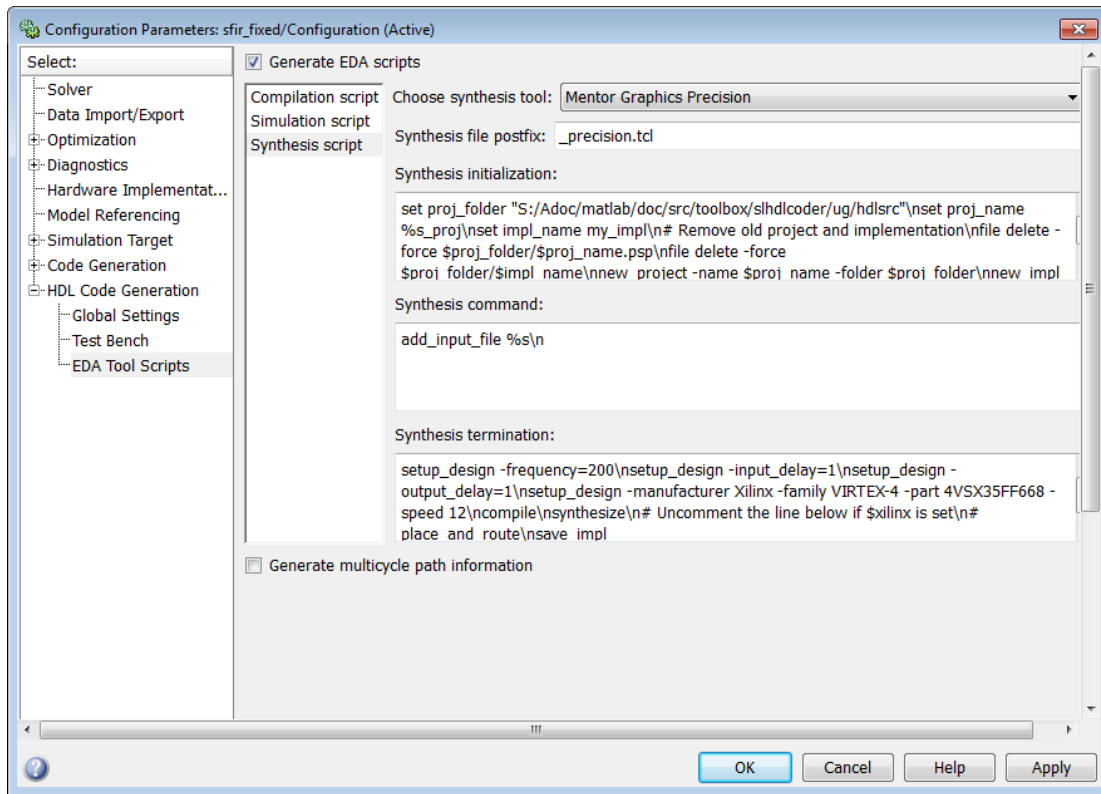


To enable synthesis script generation, select a synthesis tool from the **Choose synthesis tool** menu.

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	None (default): do not generate a synthesis script Xilinx ISE: generate a synthesis script for Xilinx ISE Microsemi Libero: generate a synthesis script for Microsemi Libero Mentor Graphics Precision: generate a synthesis script for Mentor Graphics Precision Altera Quartus II: generate a synthesis script for Altera Quartus II Synopsys Synplify Pro: generate a synthesis script for Synopsys Synplify Pro Xilinx Vivado: generate a synthesis script for Xilinx Vivado

Option Name	Description
	Custom: generate a custom synthesis script
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 string 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 string 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>
Synthesis termination	<p>Format string passed to <code>fprintf</code> to write the <code>Term</code> section of the synthesis script.</p> <p>The content of the string is specific to the selected synthesis tool.</p>

Add Synthesis Attributes

To learn how to add synthesis attributes in the generated HDL code for multiplier mapping, see “DSPStyle”.

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

- “What Is the HDL Workflow Advisor?” on page 21-2
- “Open the HDL Workflow Advisor” on page 21-3
- “Using the HDL Workflow Advisor Window” on page 21-6
- “Save and Restore HDL Workflow Advisor State” on page 21-9
- “Fix a Workflow Advisor Warning or Failure” on page 21-13
- “View and Save HDL Workflow Advisor Reports” on page 21-15
- “FPGA Floating-Point Library Mapping” on page 21-20
- “Map to an FPGA Floating-Point Library” on page 21-23
- “FPGA Synthesis and Analysis” on page 21-26
- “Automated Workflows for Specific Targets and Tools” on page 21-36

What Is the HDL Workflow Advisor?

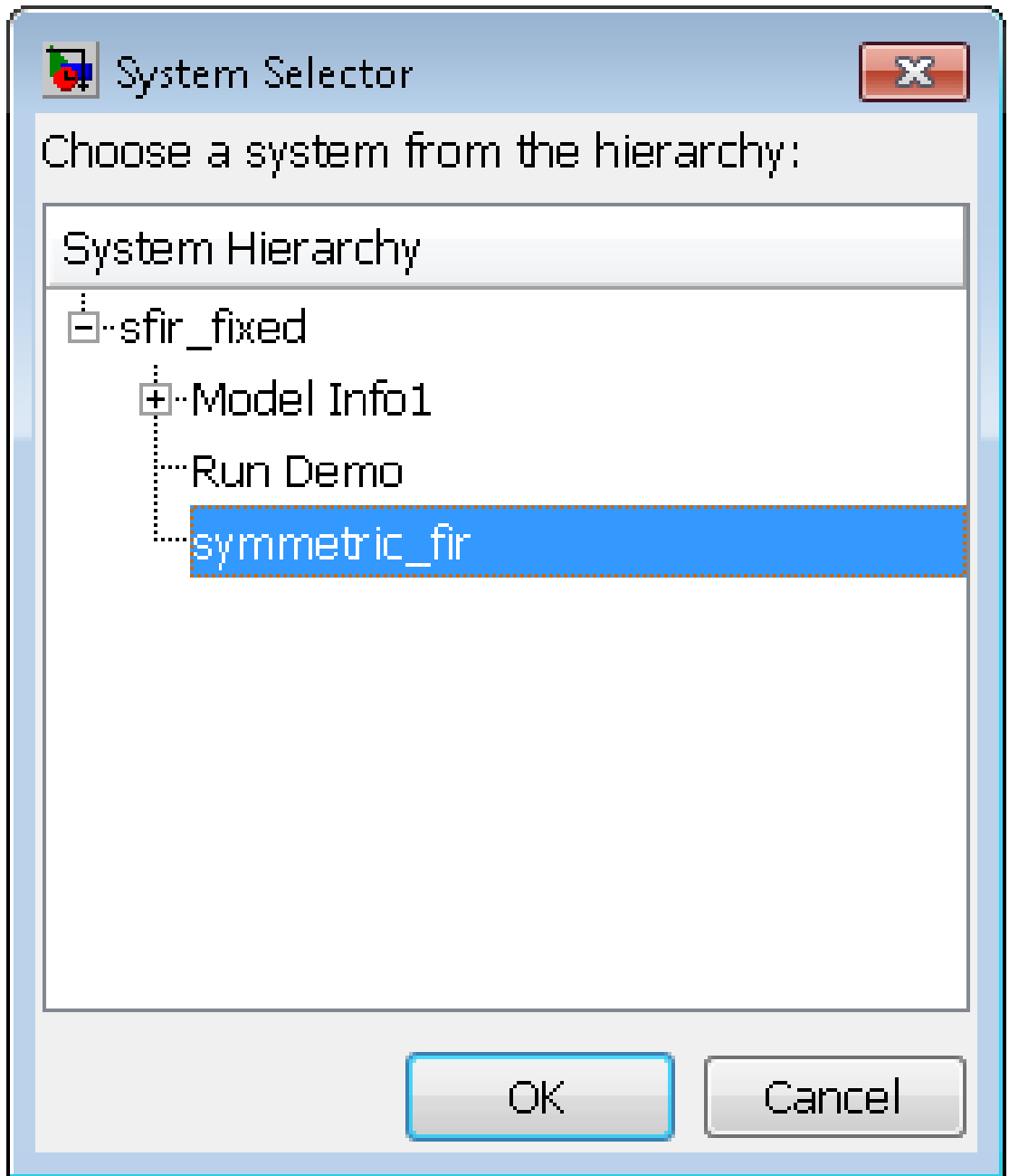
The HDL Workflow Advisor is a tool that supports and integrates the stages of the FPGA design process, such as:

- Checking the Simulink model for HDL code generation compatibility
- Automatically fixing model settings that are incompatible with HDL code generation
- Generation of RTL code, RTL test bench, a cosimulation model, or a combination of these
- Synthesis and timing analysis through integration with third-party synthesis tools
- Back annotation of the Simulink model with critical path and other information obtained during synthesis
- Complete automated workflows for selected FPGA development target devices and Simulink Real-Time™, including FPGA-in-the-Loop simulation

Open the HDL Workflow Advisor

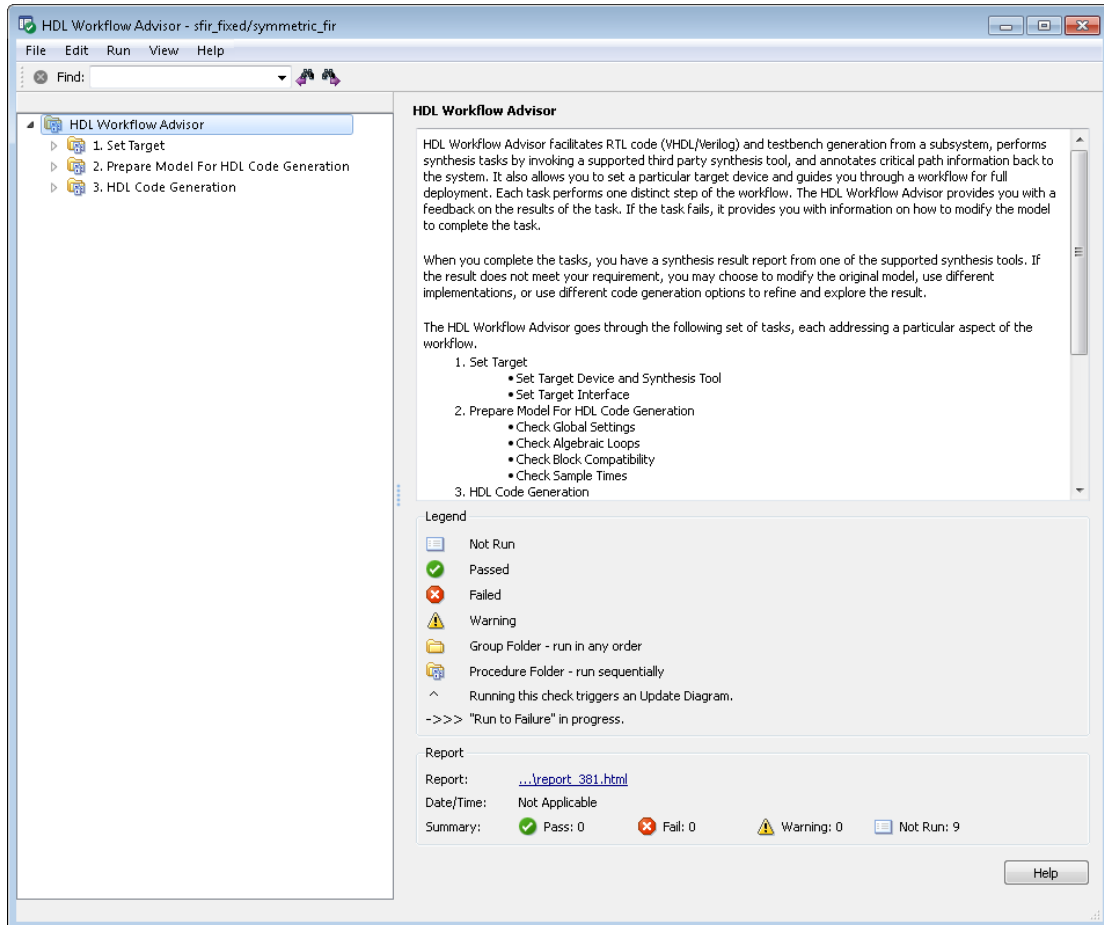
To start the HDL Workflow Advisor from a model:

- 1 Open your model.
- 2 Select **Code > HDL Code > HDL Workflow Advisor**.
- 3 In the System Selector window, select the DUT that you want to review. In the following figure, the `symmetric_fir` subsystem is the selected DUT.



4 Click OK.

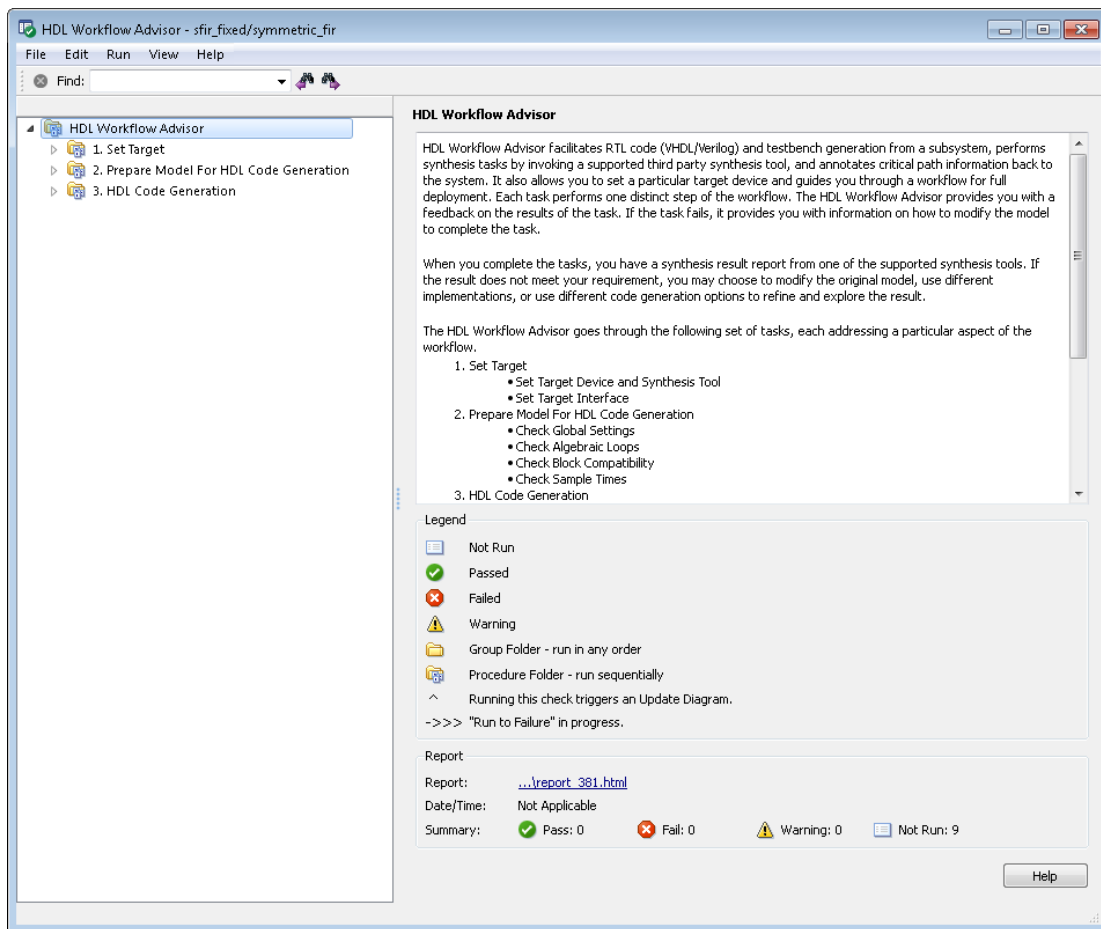
The HDL Workflow Advisor initializes and appears.



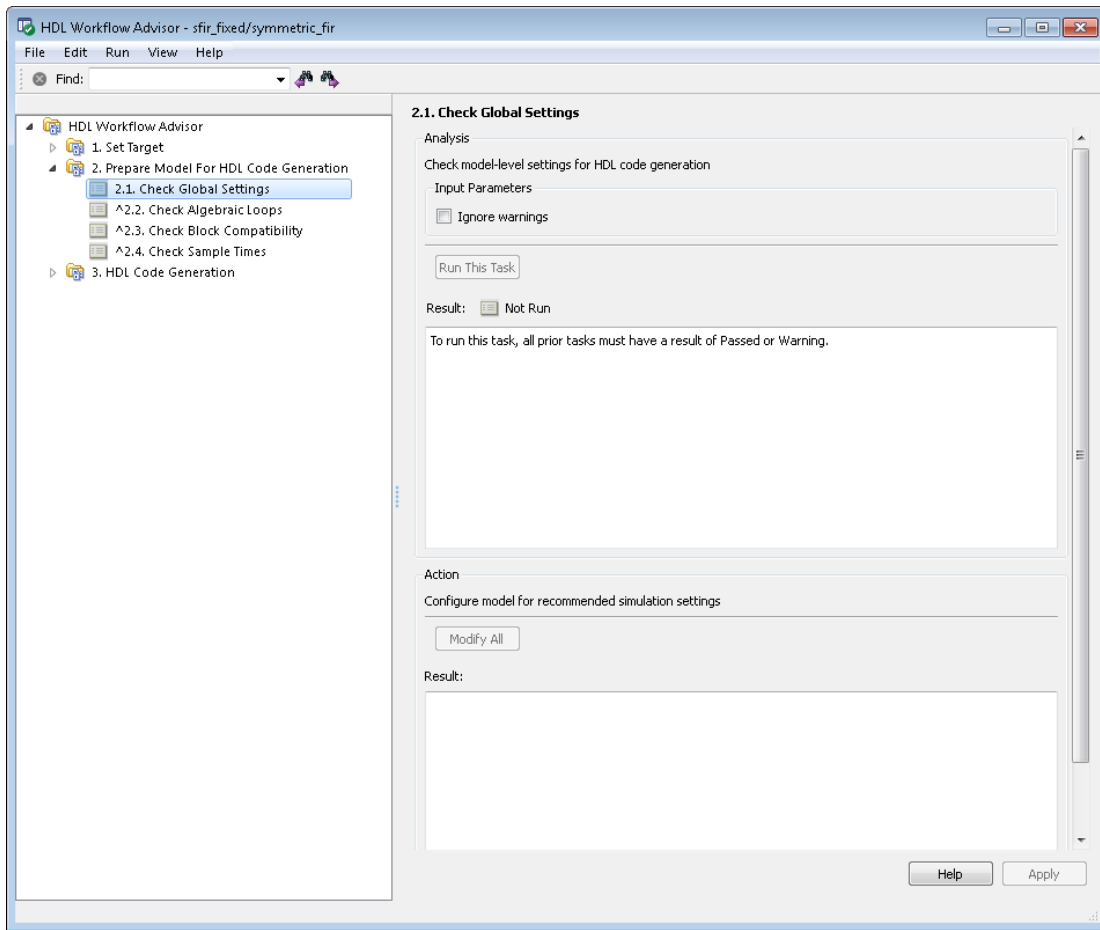
To start the HDL Workflow Advisor from the command line, enter `hdladvisor(system)`, where *system* is a handle or name of the model or subsystem that you want to check. For more information, see the `hdladvisor` function reference page.

Using the HDL Workflow Advisor Window

The following figure shows the top-level view of the HDL Workflow Advisor. The left pane lists the folders in the HDL Workflow Advisor hierarchy. Each folder represents a group or category of related tasks.



Expanding the folders shows available tasks in each folder. The following figure shows the expanded **Prepare Model For HDL Code Generation** folder, with the **Check Global Settings** task selected.



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 content 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 (for example, setting code or test bench generation parameters), the right pane displays many parameter and option settings.

When you right-click a folder or an individual task in the left pane, a context menu appears. The context menu lets you:

- Select a task or a group of tasks to run sequentially.
- Reset the status of one or more tasks to **Not Run**. Resetting status enables you to rerun tasks.
- View context-sensitive help (CSH) for an individual task.

Save and Restore HDL Workflow Advisor State

In this section...

“How the Save and Restore Process Works” on page 21-9

“Limitations of the Save and Restore Process” on page 21-9

“Save the HDL Workflow Advisor State” on page 21-9

“Restore the HDL Workflow Advisor State” on page 21-11

How the Save and Restore Process Works

By default, the HDL Coder software saves the state of the most recent HDL Workflow Advisor session. The next time 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*. At a later time, you can restore the same settings by loading the restore point data into the HDL Workflow Advisor.

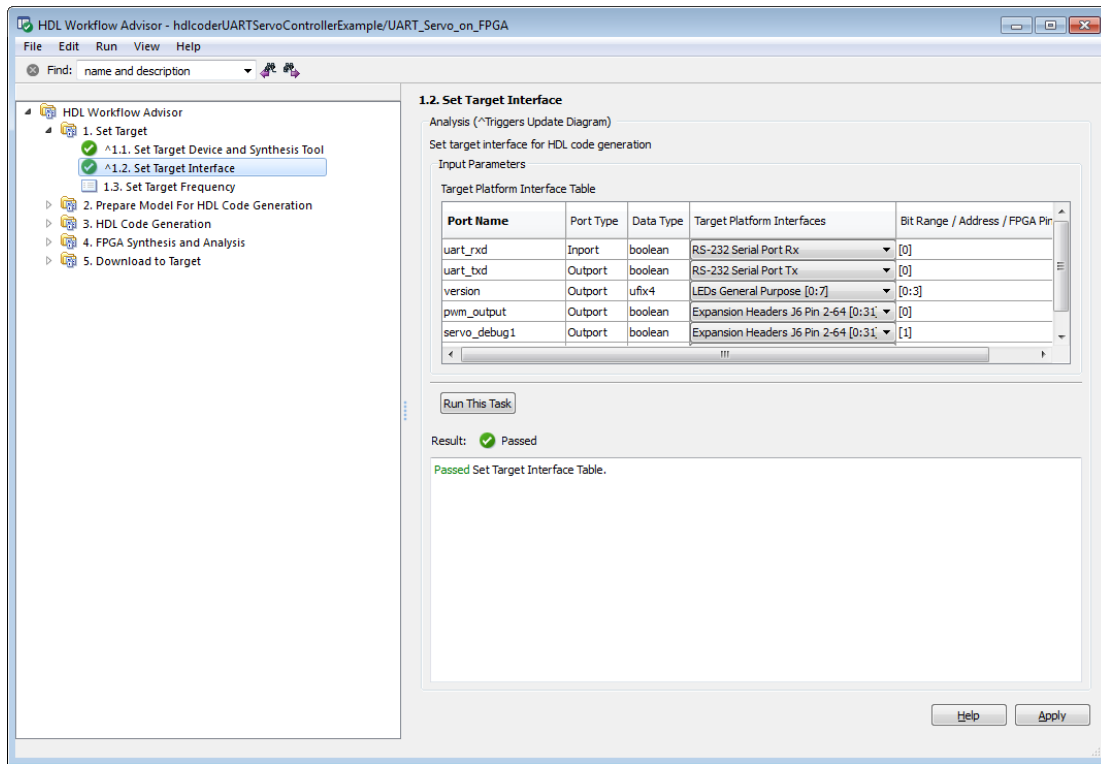
Limitations of the Save and Restore Process

The save and restore process has the following limitations:

- Operations that you perform outside the HDL Workflow Advisor is not included in the save/restore process.
- The state of HDL Workflow Advisor tasks involving third-party tools are not saved or restored.

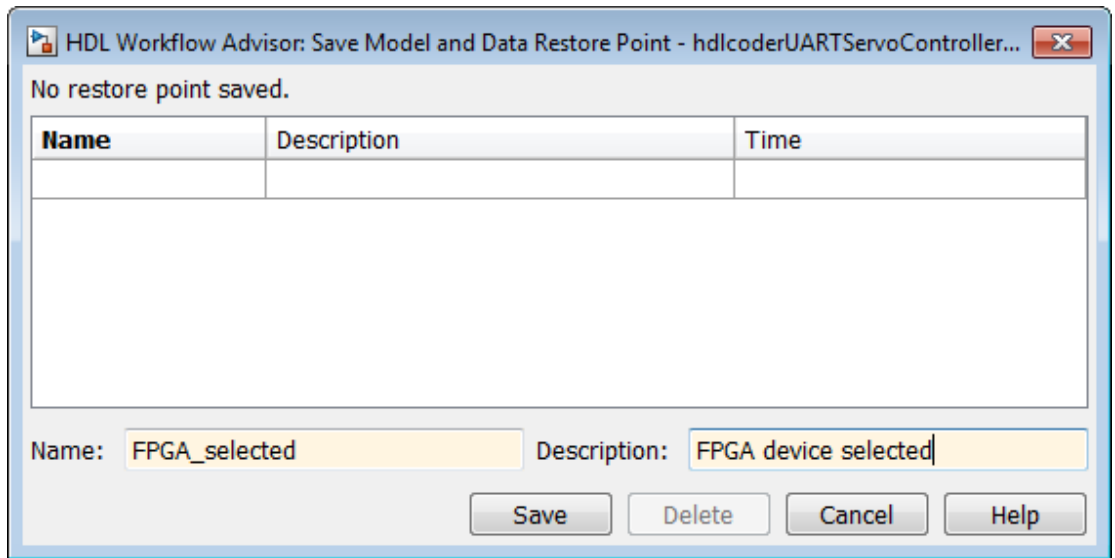
Save the HDL Workflow Advisor State

You can create and save a restore point after completion of a task sequence. For example, the following figure shows the HDL Workflow Advisor after completion of the **Set Target Interface** task.



To save the HDL Workflow Advisor settings:

- 1 In the HDL Workflow Advisor, select **File > Save Restore Point As**.
- 2 In the **Name** field, enter a name for the restore point.
- 3 In the **Description** field, you can add an optional description of the restore point.

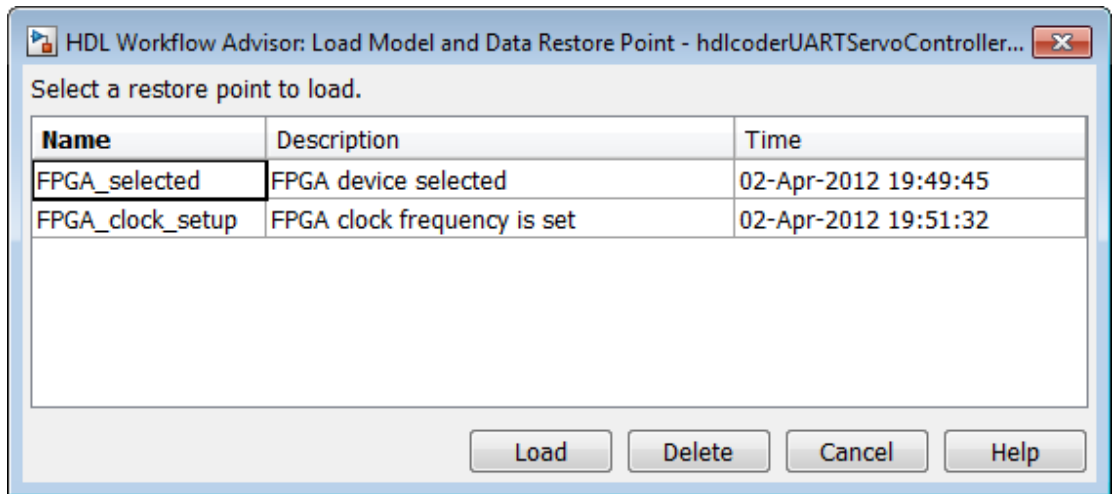


- 4 Click **Save**. The HDL Workflow Advisor saves a restore point of the current settings.

Restore the HDL Workflow Advisor State

To load a restore point:

- 1 In the HDL Workflow Advisor, select **File > Load Restore Point**.



- 2 Select the restore point that you want.
- 3 Click **Load**.

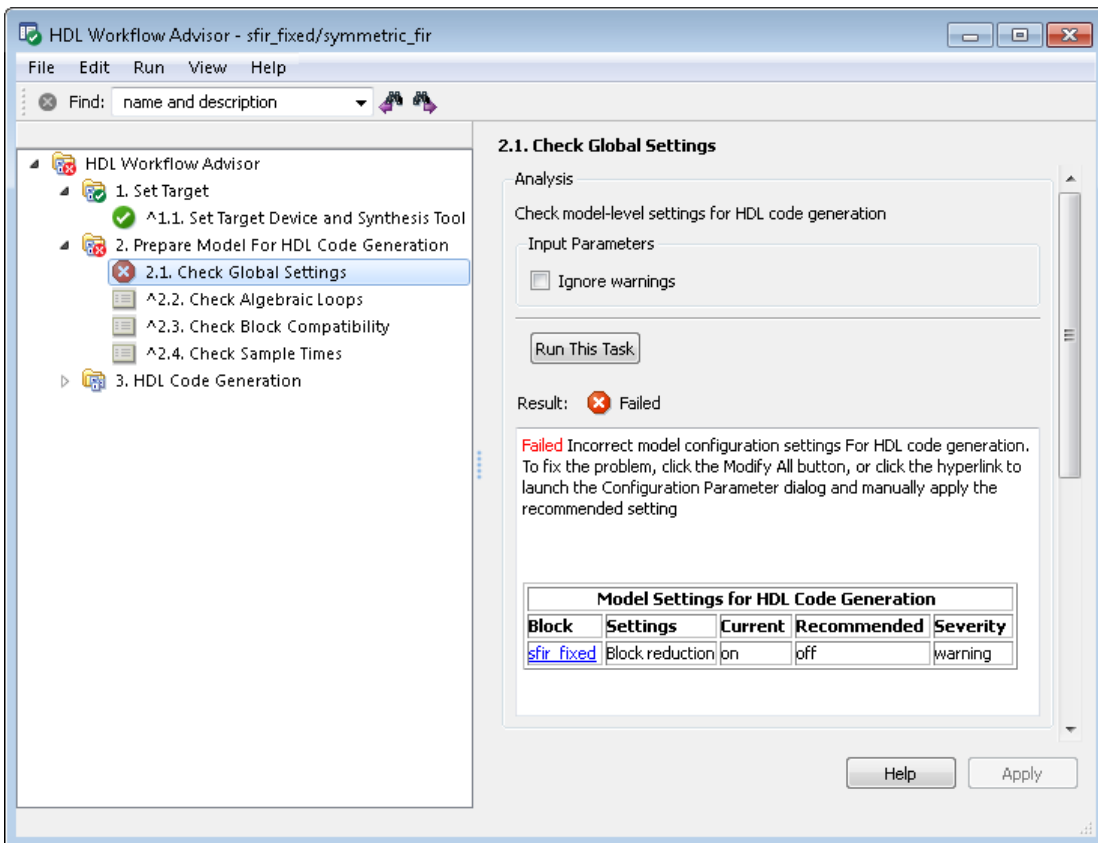
The HDL Workflow Advisor issues a warning that the restoration will overwrite current settings.

- 4 Click **Load** to load the restore point you selected. The HDL Workflow Advisor restores the previously saved state.

Fix a Workflow Advisor Warning or Failure

If a task terminates due to a warning or failure condition, the right pane of the HDL Workflow Advisor shows information about the problems. This information appears in a **Result** subpane. The **Result** subpane also suggests model settings you can use to fix the problems.

Some tasks have an **Action** subpane that lets you apply the recommended actions listed in the **Result** subpane automatically. In the following example, the **Check Global Settings** task has failed, displaying an incorrect model setting in the **Result** pane.



The **Action** subpane, below the **Result** subpane, contains a **Modify All** button. To fix the problems that appear in the **Result** subpane, click the **Modify All** button.

After you click **Modify All**, the **Result** subpane reports the changes that were applied. The task status is reset, enabling you to rerun the task and proceed to the following tasks.

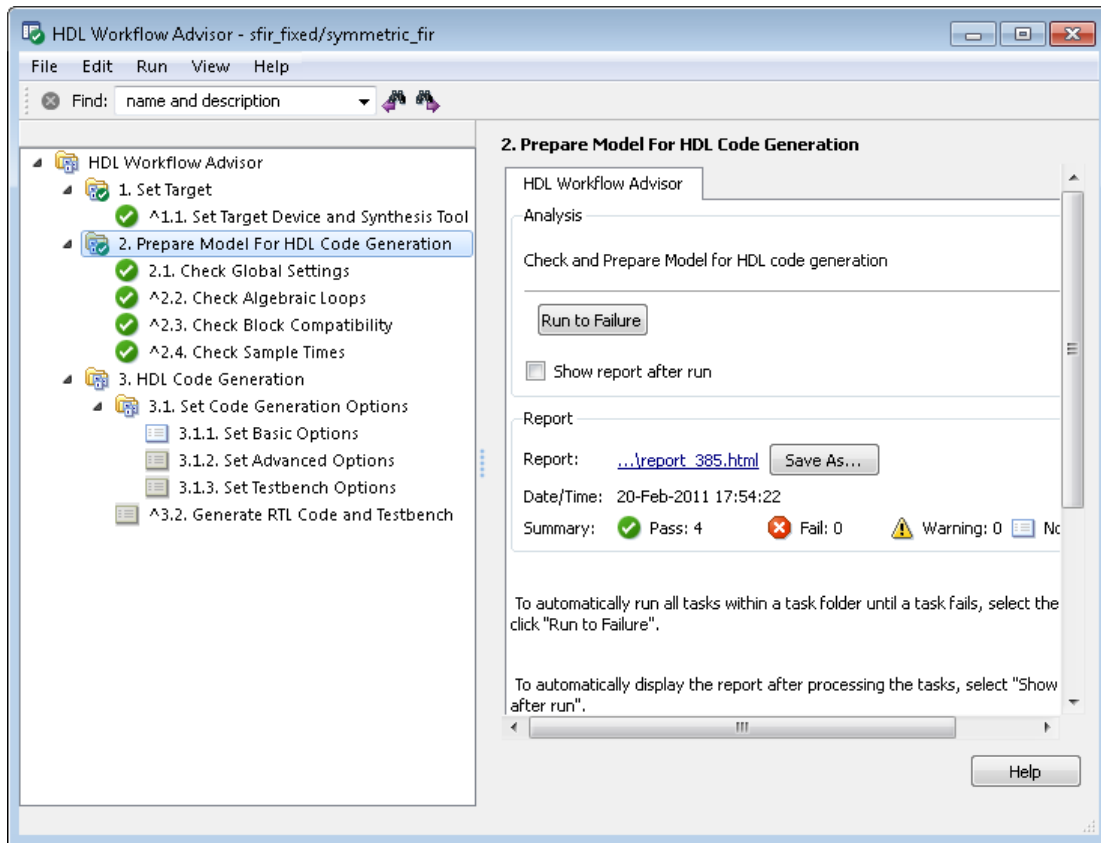
View and Save HDL Workflow Advisor Reports

In this section...
“Viewing HDL Workflow Advisor Reports” on page 21-15
“Saving HDL Workflow Advisor Reports” on page 21-18

Viewing HDL Workflow Advisor Reports

When the HDL Workflow Advisor runs tasks, it automatically generates an HTML report of task results. Each folder in the HDL Workflow Advisor contains a report for the checks within that folder and its subfolders.

You can access reports by selecting a folder and clicking the link in the **Report** subpane. In the following example, the **Prepare Model For HDL Code Generation** folder is selected.



The following report shows typical results for a run of the **Prepare Model For HDL Code Generation** tasks.

Report name: Model Advisor - 2. Prepare Model For HDL Code Generation





Simulink version: 7.7

Model version: 1.67

System: sfir_fixed/symmetric_fir

Current run: 20-Feb-2011 20:51:24

Run Summary

<input checked="" type="checkbox"/> Pass	<input checked="" type="checkbox"/> Fail	<input checked="" type="checkbox"/> Warning	<input checked="" type="checkbox"/> Not Run	Total
 4	 0	 0	 0	4

2. Prepare Model For HDL Code Generation

 2.1. Check Global Settings**Passed** Correct Simulation settings for HDL code generation

 2.2. Check Algebraic Loops**Passed** No algebraic loop detected

 2.3. Check Block Compatibility**Passed** Running Check Block Compatibility passed.

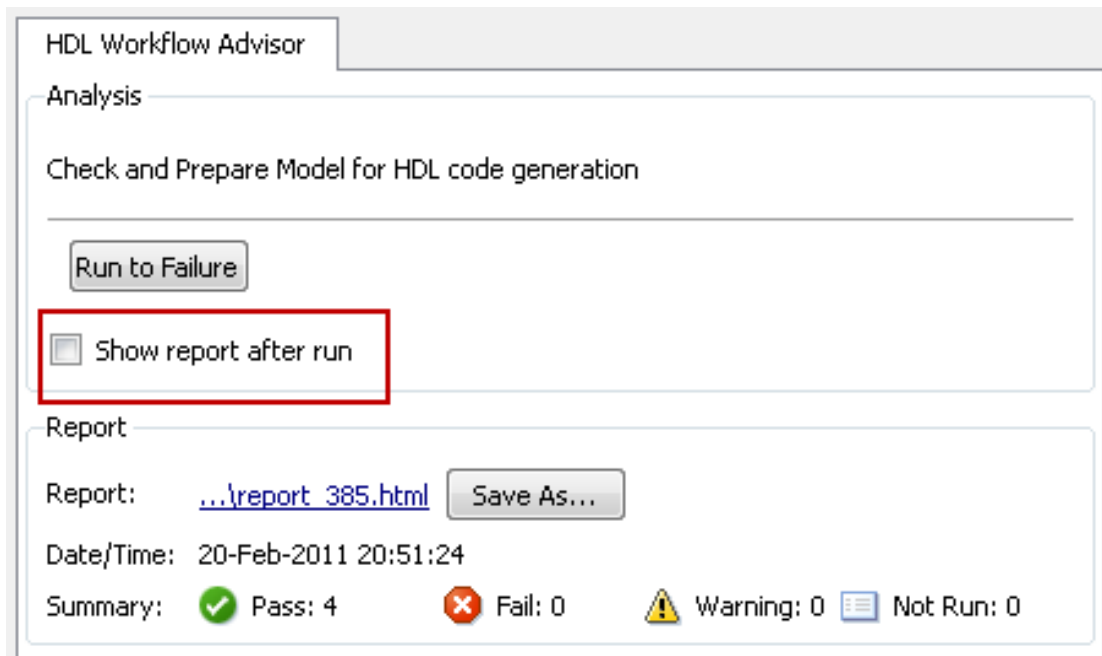
 2.4. Check Sample Times**Passed** Running Check Sample Times passed.

As you run checks, the HDL Workflow Advisor updates the reports with the latest information for each check in the folder. A message appears in the report when you run the checks at different times. Time stamps indicate when checks have been run. The time of the current run appears at the top right of the report. Checks that occurred during previous runs have a time stamp following the check name.

You can manipulate the report to show only what you are interested in viewing as follows:

- The check boxes under **Run Summary** allow you to view only the checks with the status that you are interested in viewing. For example, you can remove the checks that have not run by clearing the check box next to the **Not Run** status.
- Minimize folder results in the report by clicking the minus sign next to the folder name. When you minimize a folder, the report updates to display a run summary for that folder.

You can view the report for a folder automatically each time the folder's tasks run. To do this, select **Show report after run**:



Saving HDL Workflow Advisor Reports

You can archive an HDL Workflow Advisor report by saving it to a new location. To save a report:

- 1 In the HDL Workflow Advisor, navigate to the folder that contains the report you want to save.

- 2 Select the folder that you want. The right pane of the HDL Workflow Advisor shows information about that folder, including a **Report** subpane.
- 3 In the **Report** subpane, click **Save As**.
- 4 In the Save As dialog box, navigate to the location where you want to save the report, and click **Save**. The HDL Workflow Advisor saves the report to the new location.

Note: If you rerun the HDL Workflow Advisor, the report is updated in the working folder, not in the save location. You can find the full path to the report in the title bar of the report window. Typically, the report is within the working folder: `s1prj\modeladvisor\HDLAdv_model_name\DUT_name\`.

FPGA Floating-Point Library Mapping

In this section...

“What is an FPGA Floating-Point Library?” on page 21-20

“Why Map to an FPGA Floating Point Library?” on page 21-20

“Supported Floating-Point Operations” on page 21-20

“Limitations for FPGA Floating-Point Library Mapping” on page 21-21

What is an FPGA Floating-Point Library?

An FPGA floating-point library is a set of floating-point IP blocks that is optimized for synthesis on specific FPGA hardware.

Altera Megafunctions and Xilinx LogiCORE IP are examples of such libraries.

Why Map to an FPGA Floating Point Library?

Mapping to an FPGA 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 has the following advantages:

- Reduces the loss of data precision.
- Enables you to model a wider dynamic range.
- Saves time by skipping a step in the code generation process.

Supported Floating-Point Operations

Xilinx LogiCORE IP Floating-Point Operation Support

HDL Coder can map to the following Xilinx LogiCORE IP floating-point operations:

- add
- subtract
- multiply
- divide
- comparison

- conversion
- square root

Altera Megafunction Floating-Point Operation Support

HDL Coder can map to the following Altera Megafunction floating-point operations:

- absolute value
- adder
- comparator
- converter
- divider
- exponential
- inverse
- inverse square root
- multiplier
- natural logarithm
- square root
- subtractor
- trigonometric cosine
- trigonometric sine

Limitations for FPGA Floating-Point Library Mapping

If your synthesis tool is Xilinx Vivado, you cannot use FPGA floating-point library mapping.

Data type limitations:

- Complex data type is not supported.
- Conversion between double and single precision data types is not supported.

Unsupported Simulink blocks:

- MATLAB Function
- Chart

- Truth Table
- FFT
- Lookup Tables
- RAMs
- MinMax
- DTI
- Counters

Unsupported Simulink block modes:

- Sum with - ports.
- Sum with more than 2 inputs.
- Product with more than 2 inputs.
- Switch with a control input other than `u2 ~= 0`.
- Sum of Elements with an architecture other than `Tree`.
- Product of Elements with an architecture other than `Tree`.

HDL Coder workflow restrictions:

- IP Core Generation workflow is not supported.
- For FPGA Turnkey and Simulink Real-Time FPGA I/O workflows, your DUT ports cannot use floating-point data types.
-

Map to an FPGA Floating-Point Library

In this section...

“Setup for FPGA Floating-Point Library Mapping” on page 21-23

“How to Map to an FPGA Floating-Point Library” on page 21-23

“Analyze Results of Floating-Point Library Mapping” on page 21-25

Setup for FPGA Floating-Point Library Mapping

To map your floating-point design to an Altera or Xilinx FPGA floating-point library, you must:

- Know which Altera or Xilinx FPGA you are using.
 - If you are using a Xilinx FPGA, set up your Xilinx FPGA floating-point library tool. See “Xilinx FPGA Floating-Point Library Setup”.
- Set up your FPGA synthesis tool. See “Synthesis Tool Path Setup”.

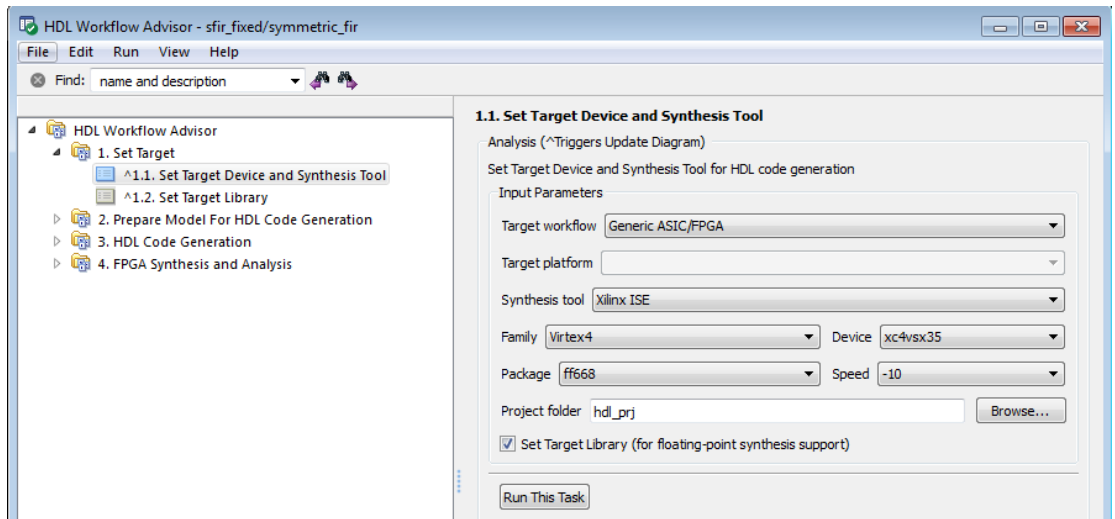
Note: If you are using Altera Quartus 10.1 or 11.0, you must turn on the `AlteraBackwardIncompatibleSinCosPipeline` global property using `hdlset_param`. For example, to turn on `AlteraBackwardIncompatibleSinCosPipeline` for a model, `my_dut`, enter the following at the command line:

```
hdlset_param('my_dut','AlteraBackwardIncompatibleSinCosPipeline','on')
```

How to Map to an FPGA Floating-Point Library

To map to an FPGA floating-point library:

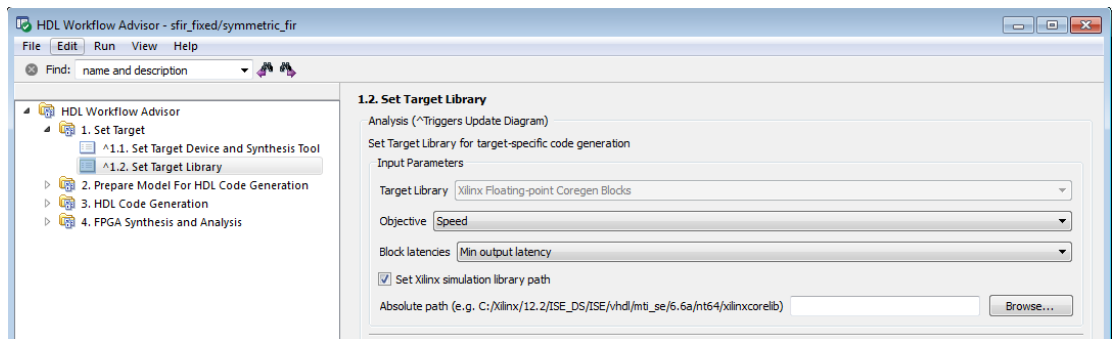
- 1 Open the HDL Workflow Advisor.
- 2 In the left pane, click **HDL Workflow Advisor > Set Target > Set Target Device and Synthesis Tool**. The following Set Target Device and Synthesis Tool pane appears.



- 3 For **Target workflow**, select **Generic ASIC/FPGA**.
- 4 Select your **Synthesis tool** from the dropdown menu. The **Set Target Library (for floating-point synthesis support)** checkbox becomes available.

The **Set Target Library (for floating-point synthesis support)** option is not available if your synthesis tool is Xilinx Vivado.

- 5 Select the **Family**, **Device**, **Package**, and **Speed** of your synthesis target.
- 6 Select **Set Target Library (for floating-point synthesis support)**. A new task, **Set Target Library**, appears in the left pane.
- 7 In the left pane, click **Set Target Library** to see the following pane.



- 8 Select **Objective** and **Block latencies**.
- 9 (For **Xilinx** devices only) If you wish to enter the path to the pre-compiled simulation library, select **Set Xilinx simulation library path** and enter the **Absolute path**. Otherwise, HDL Coder automatically detects the simulation library path.

Analyze Results of Floating-Point Library Mapping

To see your FPGA floating-point library mapping results, enable generation of the Resource Utilization Report and Optimization Report before you begin code generation. To learn how to generate these reports, see “Create and Use Code Generation Reports”.

The Resource Utilization Report shows the number of target-specific hardware resources used by your design. To learn more about the Resource Utilization Report, see “Resource Utilization Report”.

The Optimization Report shows whether HDL Coder was able to meet the minimum or maximum block latencies you chose from the Set Target Library pane. To learn more about the Optimization Report, see “Optimization Report”.

FPGA Synthesis and Analysis

In this section...

“FPGA Synthesis and Analysis Tasks Overview” on page 21-26

“Creating a Synthesis Project” on page 21-26

“Performing Synthesis, Mapping, and Place and Route” on page 21-28

“Annotating Your Model with Critical Path Information” on page 21-31

FPGA Synthesis and Analysis Tasks Overview

The tasks in the **FPGA Synthesis and Analysis** folder let you run third-party FPGA synthesis and analysis tools without leaving the HDL Workflow Advisor environment. Tasks in this category include:

- Creation of FPGA synthesis projects for supported FPGA synthesis tools
- Launching supported FPGA synthesis tools to perform synthesis, mapping, and place/route tasks
- Annotation of your original model with critical path information obtained from the synthesis tools

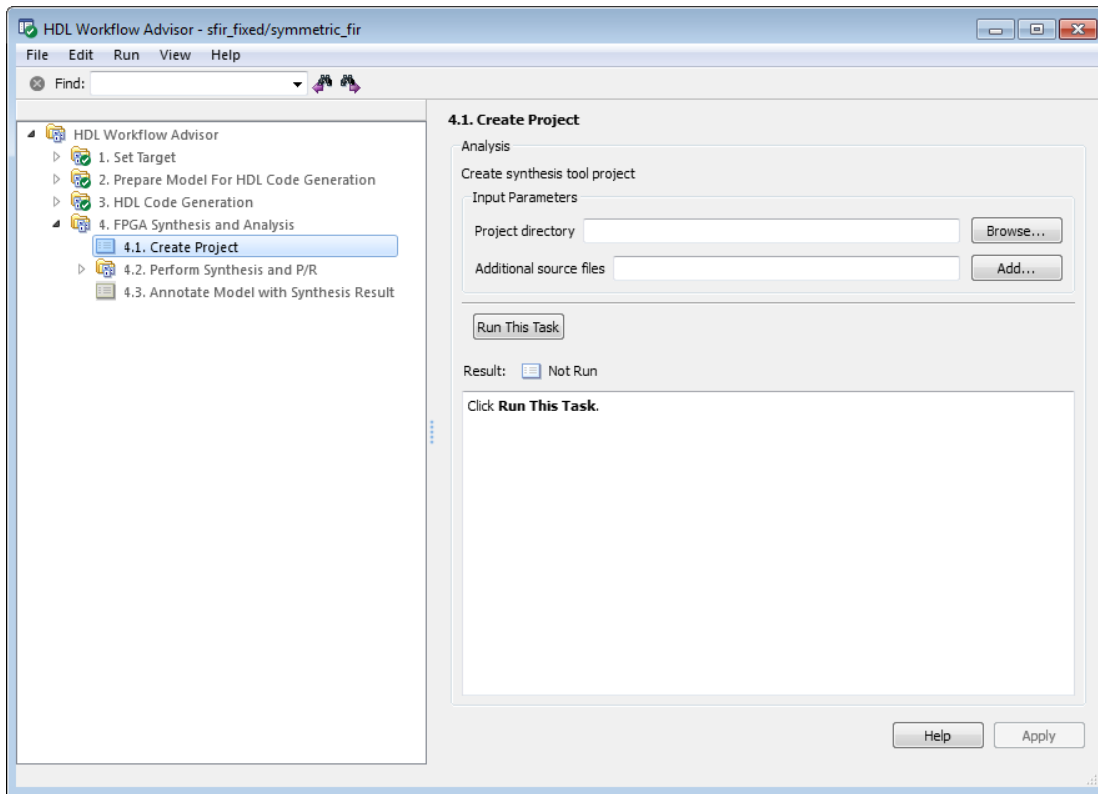
Note: A supported synthesis tool must be installed, and the synthesis tool executable must be on the system path to perform the tasks in the **FPGA Synthesis and Analysis** folder. See “Third-Party Synthesis Tools” for more information.

Creating a Synthesis Project

The **Create Project** task does the following:

- Realizes a synthesis project for the tool from the previously generated HDL code
- Creates a link to the project files in the **Result** subpane
- (Optional) Launches the synthesis tool and opens the synthesis project

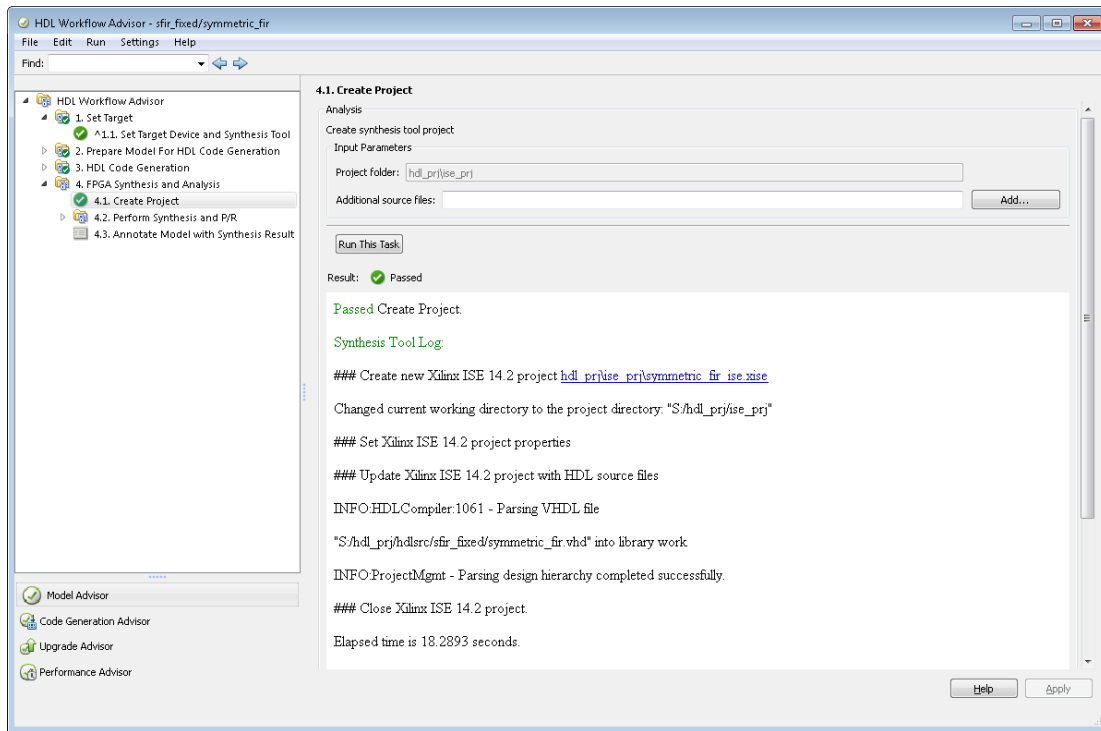
The following figure shows the **Create Project** task in an enabled state, after HDL code generation.



The **Create Project** task parameters are:

- **Project directory:** The HDL Workflow Advisor writes the project files to a subfolder of the `hdlsrc` folder. You can enter the path to an alternative folder, or click the **Browse** button to navigate to the desired folder.
- **Additional source files:** To include HDL files (or other synthesis files, such as UCF or SDC files) that the code does not generate in your synthesis project, enter the full path to the desired files. Click the **Add** button to locate each file.

The following figure shows the HDL Workflow Advisor after passing the **Create Project** task. If you want to view the synthesis project, click the hyperlink in the **Result** subpane. This link launches the synthesis tool and opens the synthesis project.



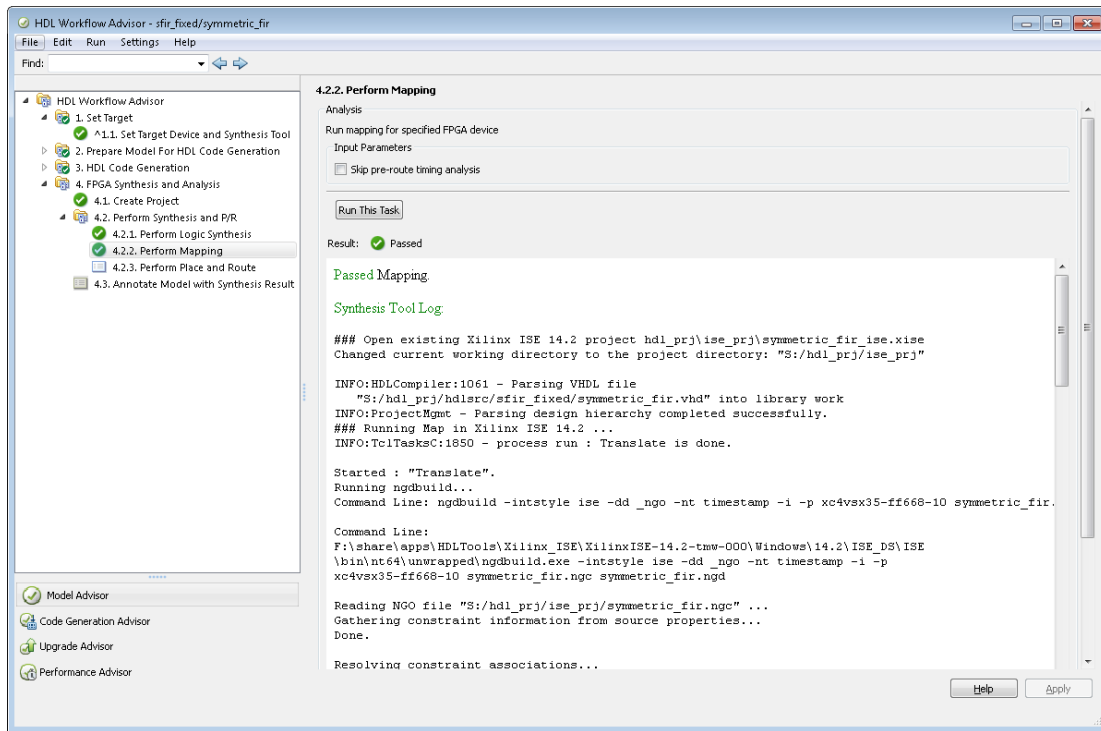
Performing Synthesis, Mapping, and Place and Route

Performing Logic Synthesis

The **Perform Logic Synthesis** task does the following:

- 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.

The **Perform Logic Synthesis** task does not have input parameters. The following figure shows the HDL Workflow Advisor after passing the **Perform Logic Synthesis** task.



Performing Place and Route

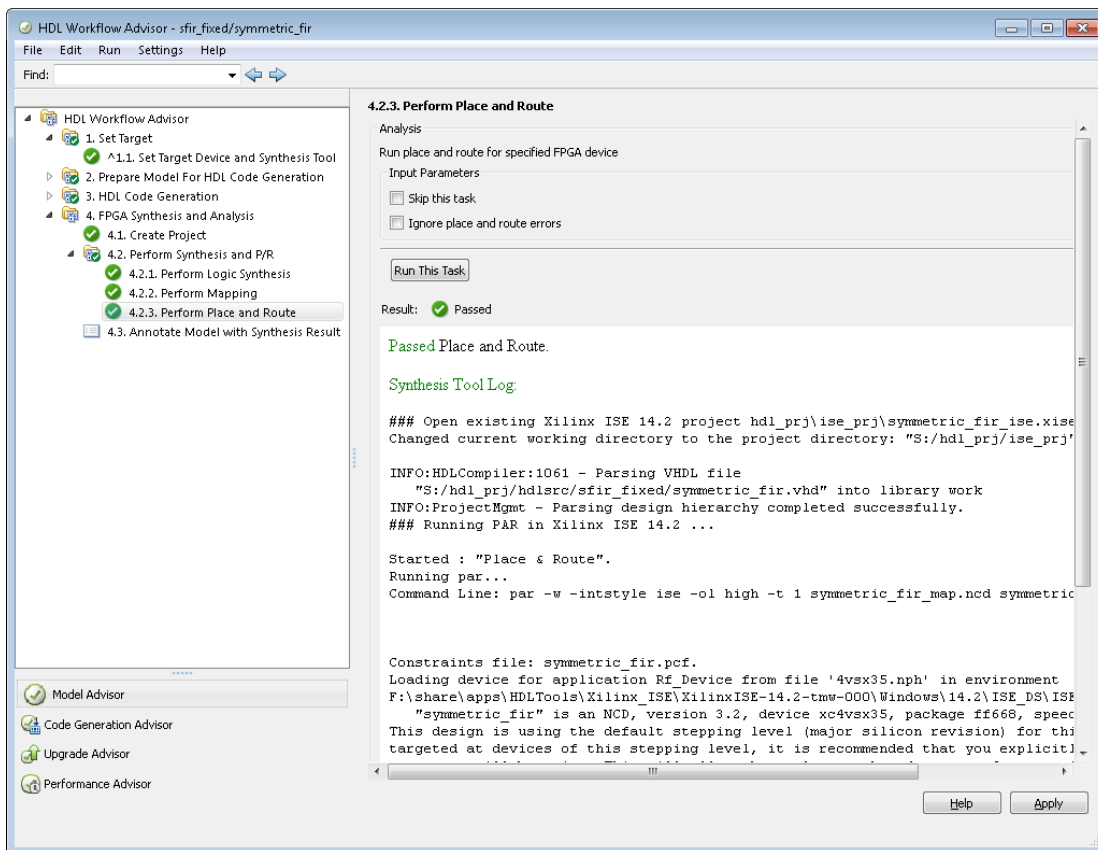
The **Perform Place and Route** task does the following:

- Launches the synthesis tool in the background.
- Runs a place and route process using the circuit description produced by the mapping process, and emits a circuit description suitable for programming an FPGA.
- Emits pre- and post-routing timing information for use in critical path analysis and back annotation of your source model.
- Displays a log in the **Result** subpane.

Unlike other tasks in the HDL Workflow Advisor hierarchy, **Perform Place and Route** is optional. If you select **Skip this task** in the right-hand pane, the HDL Workflow Advisor executes the workflow, but omits the **Perform Place and Route** task, marking it **Passed**. Select **Skip this task** if you prefer to do place and route work manually.

If the **Perform Place and Route** task fails, you can select **Ignore place and route errors** to continue to the **Annotate Model with Synthesis Result** task. This allows you to use post-mapping timing results to find critical paths in your model even if place and route fails.

The following figure shows the HDL Workflow Advisor after passing the **Perform Place and Route** task.



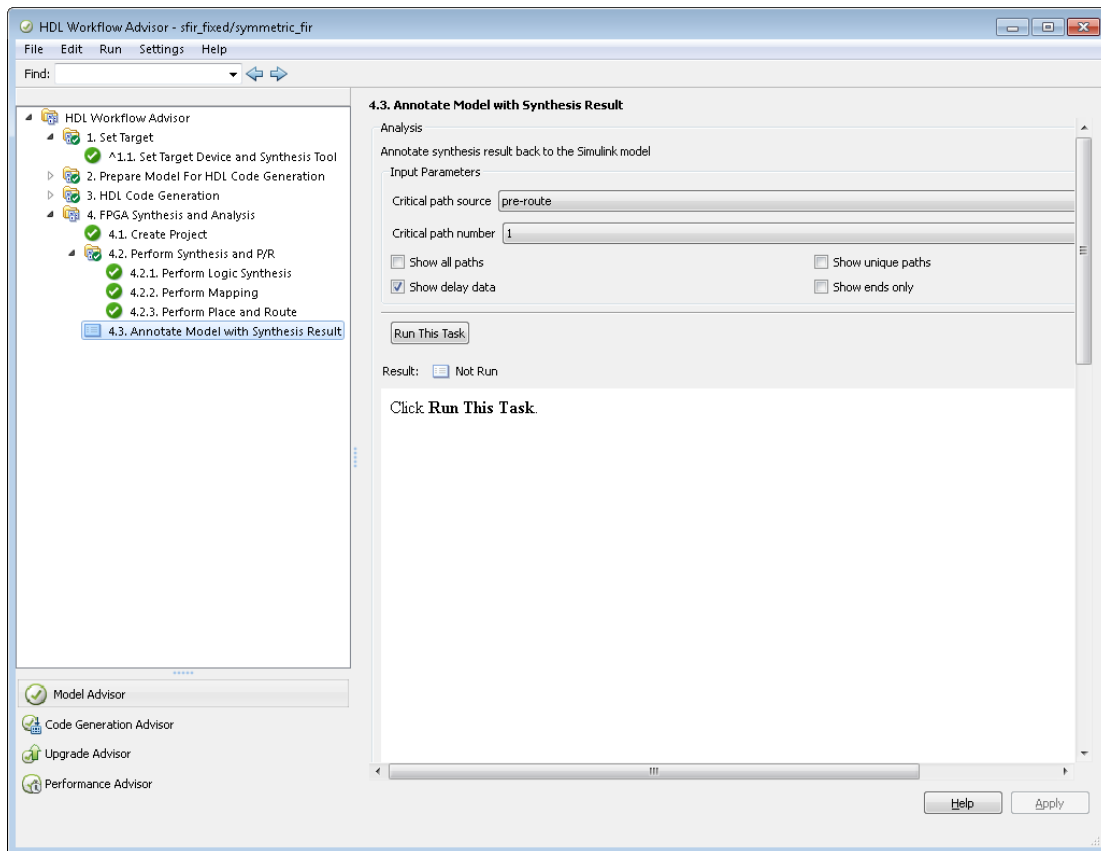
Annotating Your Model with Critical Path Information

The **Annotate Model with Synthesis Result** task helps you identify critical paths in your model. In this task, you can analyze pre- or post-routing timing information from

the **Perform Place and Route** task and visually highlight one or more critical paths in your model.

Note: If the **Annotate Model with Synthesis Result** task is not available, clear the check box for **Generate FPGA top level wrapper** in the **Generate RTL Code and Testbench** task.

The following figure shows the **Annotate Model with Synthesis Result** task in an enabled state.



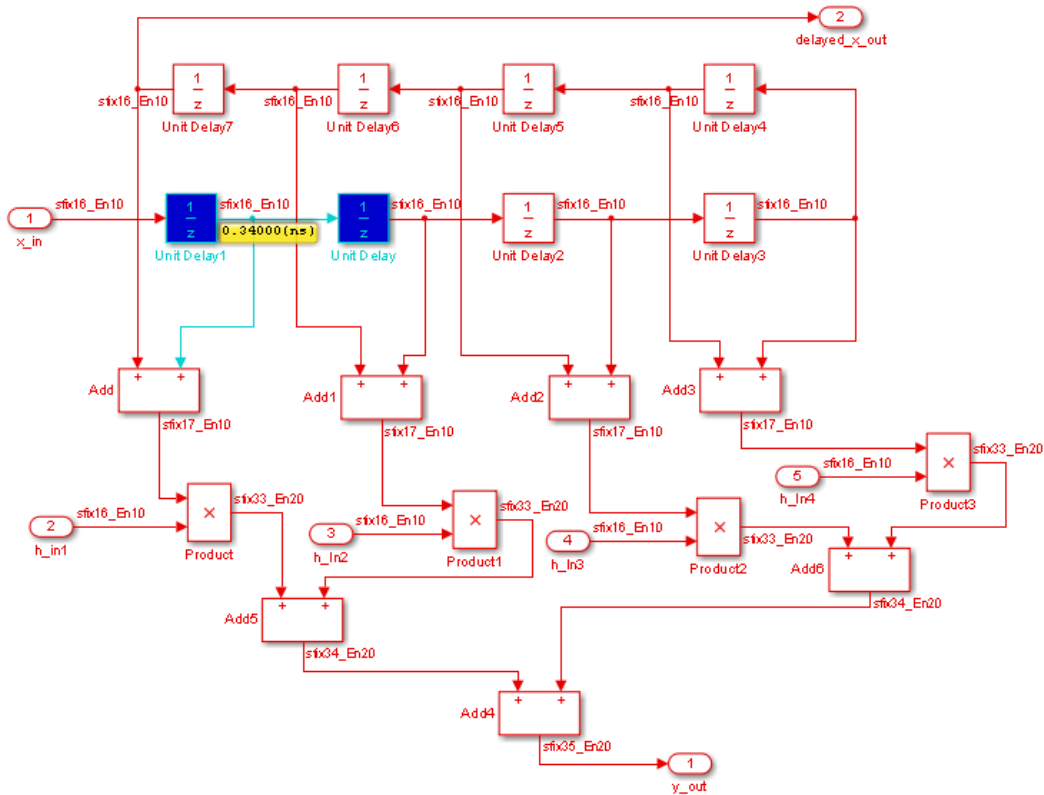
The task parameters are:

- **Critical path source:** Select **pre-route** or **post-route**. The default is **pre-route**.

Note that the **pre-route** option is unavailable when **Skip pre-route timing analysis** is enabled in the **Perform Mapping** task.

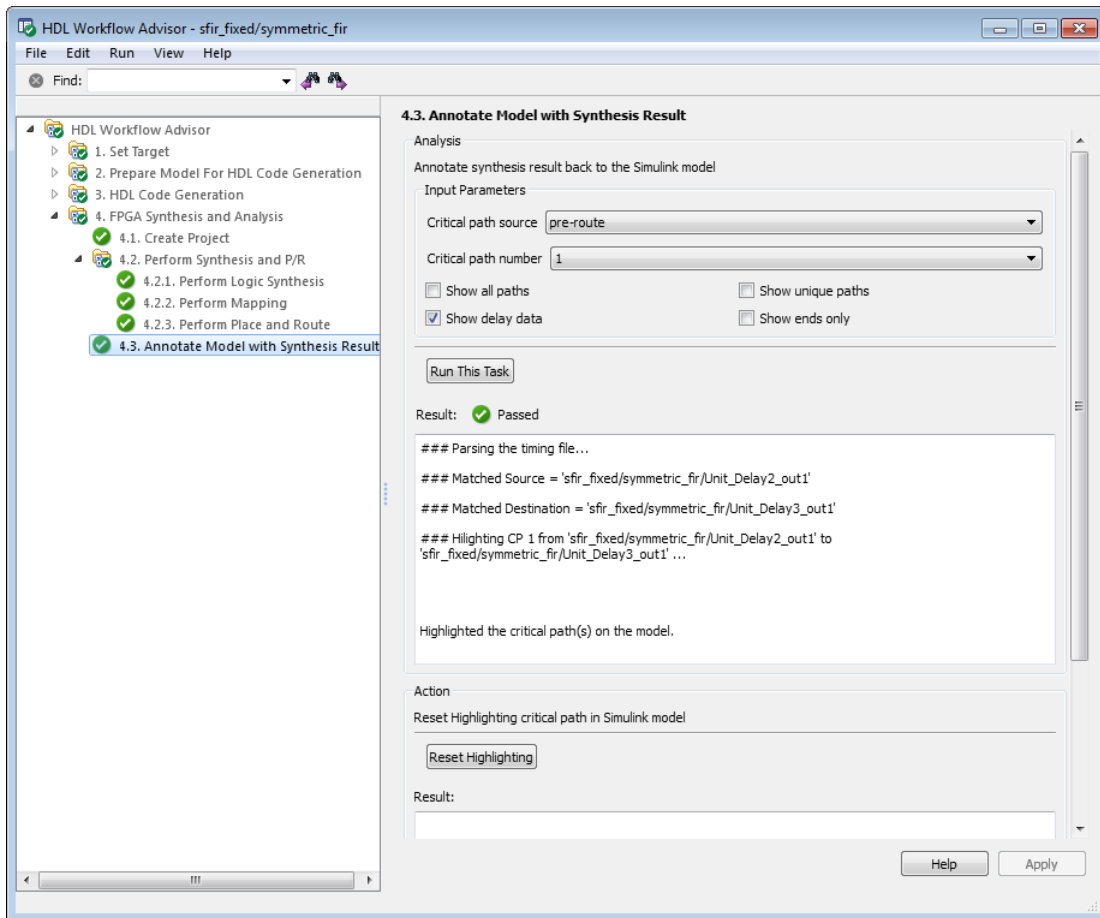
- **Critical path number:** You can annotate up to 3 critical paths. Select the number of paths you want to annotate. The default is 1.
- **Show all paths:** Show critical paths, including duplicate paths. The default is **off**.
- **Show unique paths:** Show only the first instance of a path that is duplicated. The default is **off**.
- **Show delay data:** Annotate the cumulative timing delay on each path. The default is **on**.
- **Show ends only:** Show the endpoints of each path, but omit the connecting signal lines. The default is **off**.

When the **Annotate Model with Synthesis Result** task runs to completion, HDL Coder displays the DUT with critical path information highlighted. The following figure shows a subsystem after critical path annotation. Using default options, the annotation includes the endpoints, signal lines, and delay data.



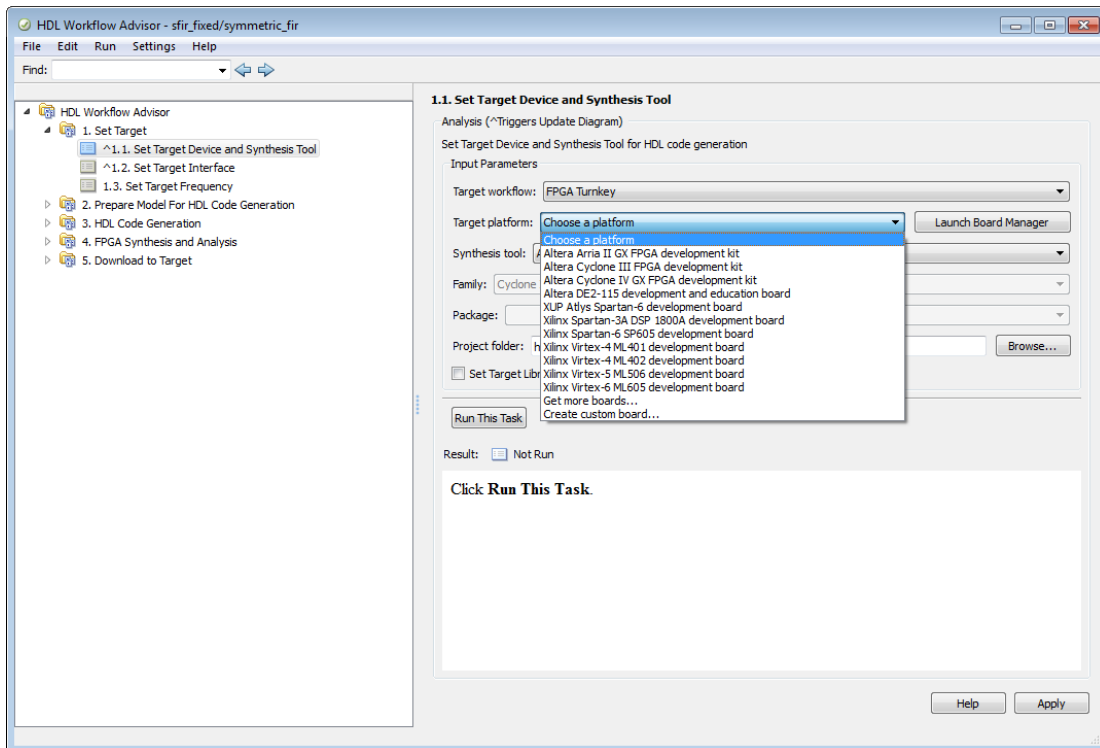
After the **Annotate Model with Synthesis Result** task runs to completion, the HDL Workflow Advisor enables the **Reset Highlighting** button in the **Action** subpane. When you click this button, the HDL Workflow Advisor:

- Clears critical path annotations from the model.
- Resets the **Annotate Model with Synthesis Result** task.



Automated Workflows for Specific Targets and Tools

The HDL Workflow Advisor helps you perform complete automated workflows for a number of target devices. For the **Target workflow** you select, the **Target platform** menu lists the supported target devices.



After you select the desired target device and configure its I/O interface, you can let the HDL Workflow Advisor perform the subsequent model checking, HDL code generation, and FPGA synthesis and analysis tasks, without your intervention. For information on automated workflows for specific types of targets, see:

- “Generate Simulink Real-Time Interface for Speedgoat Boards”
- “Program Standalone Xilinx FPGA Development Board from Simulink”
- “Program Standalone Altera FPGA Development Board from Simulink”
- “Custom IP Core Generation”

- “Hardware-Software Codesign Workflow for SoC Platforms”

HDL Test Bench

Generate Test Bench With File I/O

In this section...

“When to Use File I/O In Test Bench” on page 22-2

“How Test Bench Generation with File I/O Works” on page 22-2

“Test Bench Data Files” on page 22-2

“How to Generate Test Bench with File I/O” on page 22-3

“Limitations When Using File I/O In Test Bench” on page 22-4

When to Use File I/O In Test Bench

By default, HDL Coder generates an HDL testbench that contains the simulation data as constants. If you have a long running simulation, the generated HDL test bench contains a large amount of data, and therefore requires more memory to run in an HDL simulator.

Generate your test bench with file I/O when your MATLAB or Simulink simulation is long, or you experience memory constraints while running your HDL simulation.

How Test Bench Generation with File I/O Works

By default, when you generate an HDL test bench, HDL Coder writes the stimulus and reference data from your simulation as constants in the test bench code.

When you enable the **Use file I/O to read/write test bench data** option in the HDL Workflow Advisor and generate a test bench, HDL Coder saves the DUT input and output 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 and compares the actual DUT output with the expected output, which is also saved in .dat files. This saves memory compared to the default option.

Note that reference data is delayed by 1 clock cycle in the waveform viewer compared to default test bench generation. This is due to the delay in reading data from files.

Test Bench Data Files

Stimulus and reference data for each DUT input and output is saved in a separate test bench data file (.dat), with the following exceptions:

- 2 files are generated for the real and imaginary parts of complex data.
- Constant DUT input data is written to the test bench as constants, the same as for the default option.

Vector input or output data is saved as a single file.

How to Generate Test Bench with File I/O

Using the HDL Workflow Advisor

To generate a test bench that uses file I/O from the HDL Workflow Advisor:

- 1 In the **HDL Code Generation > Set Code Generation Options > Set Testbench Options** task, enable **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, enable **Generate RTL testbench** and click **Apply**.

After you generate code, the message window shows links to the test bench data files (.dat).

Using the Command Line

To generate a test bench that uses file I/O, use the `UseFileIOInTestBench` parameter with `makehdltb`.

For example, to generate a Verilog test bench using file I/O for a DUT subsystem, `sfir_fixed/symmetric_fir`, enter:

```
makehdltb('sfir_fixed/symmetric_fir','TargetLanguage','Verilog',...
          'UseFileIOInTestBench','on');

### Begin TestBench generation.
### Generating HDL TestBench for 'sfir_fixed/symmetric_fir'.
### Begin simulation of the model 'gm_sfir_fixed'...
### Collecting data...
### Generating test bench: hdlsrc\sfir_fixed\symmetric_fir_tb.v
### Creating stimulus vectors...
### Generating test bench data file: hdlsrc\sfir_fixed\x_in.dat
### Generating test bench data file: hdlsrc\sfir_fixed\y_out.dat
### Generating test bench data file: hdlsrc\sfir_fixed\delayed_x_out.dat
### HDL TestBench generation complete.
```

Limitations When Using File I/O In Test Bench

To use file I/O in your test bench, the following limitations apply:

- Double and single data types at DUT inputs and outputs are not supported.
- If your target language is VHDL, the **Scalarize vector ports** option must be off.

FPGA Board Customization

- “FPGA Board Customization” on page 23-2
- “Create Custom FPGA Board Definition” on page 23-7
- “Create Xilinx KC705 Evaluation Board Definition File” on page 23-8
- “FPGA Board Manager” on page 23-21
- “New FPGA Board Wizard” on page 23-25
- “FPGA Board Editor” on page 23-38

FPGA Board Customization

In this section...

“Feature Description” on page 23-2

“Custom Board Management” on page 23-2

“FPGA Board Requirements” on page 23-3

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 23-21: portal to adding, importing, deleting, and otherwise managing board definition files.
- “New FPGA Board Wizard” on page 23-25: 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 23-38: user interface for viewing or editing board information.

To begin, review the “FPGA Board Requirements” on page 23-3 and then follow the steps described in “Create Custom FPGA Board Definition” on page 23-7.

FPGA Board Requirements

- “FPGA Device” on page 23-3
- “FPGA Design Software” on page 23-3
- “General Hardware Requirements” on page 23-3
- “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 23-4
- “JTAG Connection Requirements for FPGA-in-the-Loop” on page 23-6

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” in the HDL Verifier documentation.
- For use with FPGA Turnkey, see “Supported FPGA Device Families for Board Customization” in the HDL Coder documentation.

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 23-4).

- **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 PC 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 23-4
- “Ethernet PHY Interface” on page 23-4
- “Special Timing Considerations for RGMII” on page 23-5
- “Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface” on page 23-5

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 PC. 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 (RGMII v1.3).

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. (See “FIL I/O” on page 23-31 for the usage of the MDIO module.)

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

- “Altera JTAG Connection Requirements” on page 23-6
- “Xilinx JTAG Connection Requirements” on page 23-6

Altera JTAG Connection Requirements

- Supported Altera FPGA board
- USB Blaster I or USB Blaster II download cable
- Altera Quartus II

- Windows[®]

Requires Quartus II version 13.0 or higher, Quartus II executable folder must be on system path

- Linux[®]

Requires Quartus II version 13.1 or higher, Quartus II library folder must be on `LD_LIBRARY_PATH` *before* starting MATLAB, only 64-bit Quartus are supported

- Installation of USB Blaster I or II cable driver

Xilinx JTAG Connection Requirements

- Supported Xilinx FPGA board
- Digilent[®] download cable. If your board has a standard Xilinx 14-pin JTAG connector, you can obtain the HS2 cable from Digilent.
- Xilinx Vivado 2014.2
 - For Windows operating systems: Xilinx Vivado 2014.2. Vivado executable folder must be on system path.
 - For Linux operating systems: Xilinx Vivado 2014.2 and Digilent Adept2.

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 23-21.
- 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 23-25.
- 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 23-17.

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 23-8 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 23-8

“What You Need to Know Before Starting” on page 23-8

“Start New FPGA Board Wizard” on page 23-9

“Provide Basic Board Information” on page 23-10

“Specify FPGA Interface Information” on page 23-12

“Enter FPGA Pin Numbers” on page 23-13

“Run Optional Validation Tests” on page 23-15

“Save Board Definition File” on page 23-17

“Use New FPGA Board” on page 23-18

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.

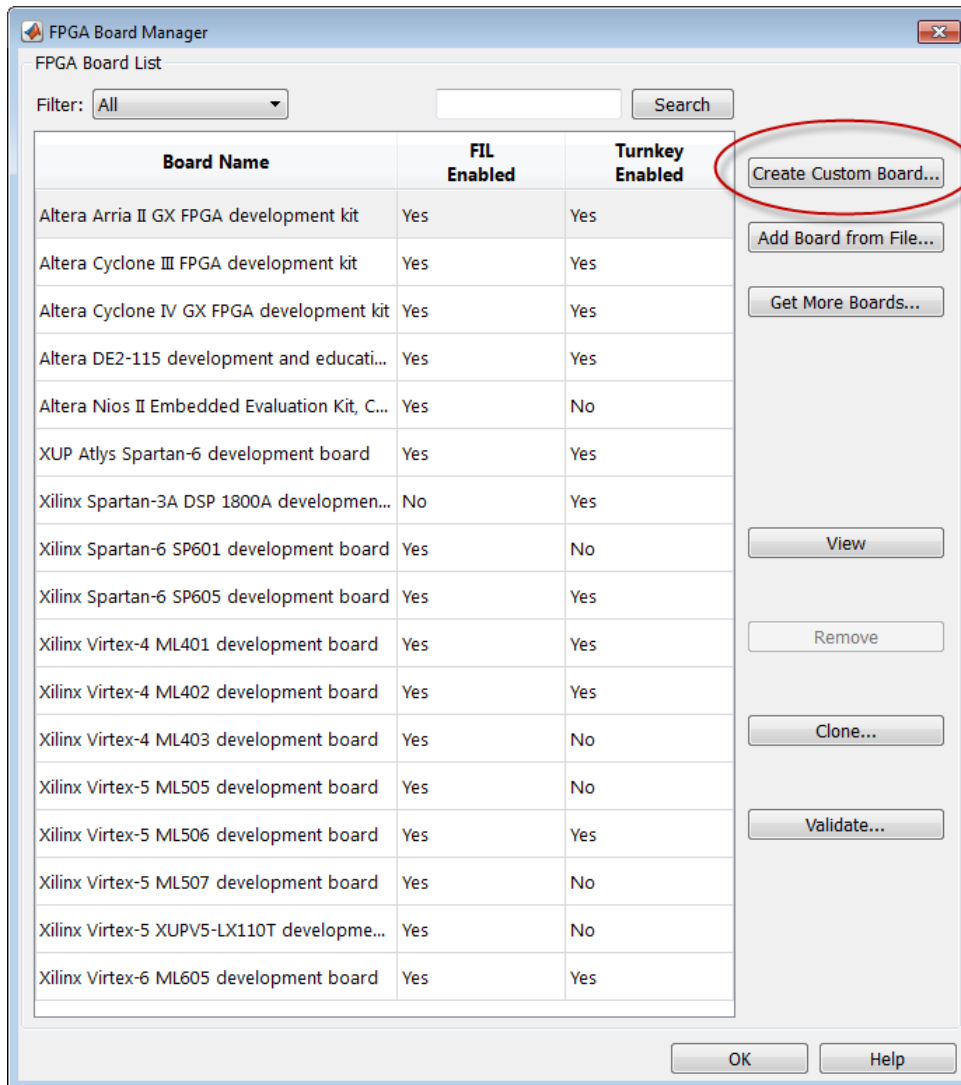
- For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

- 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:

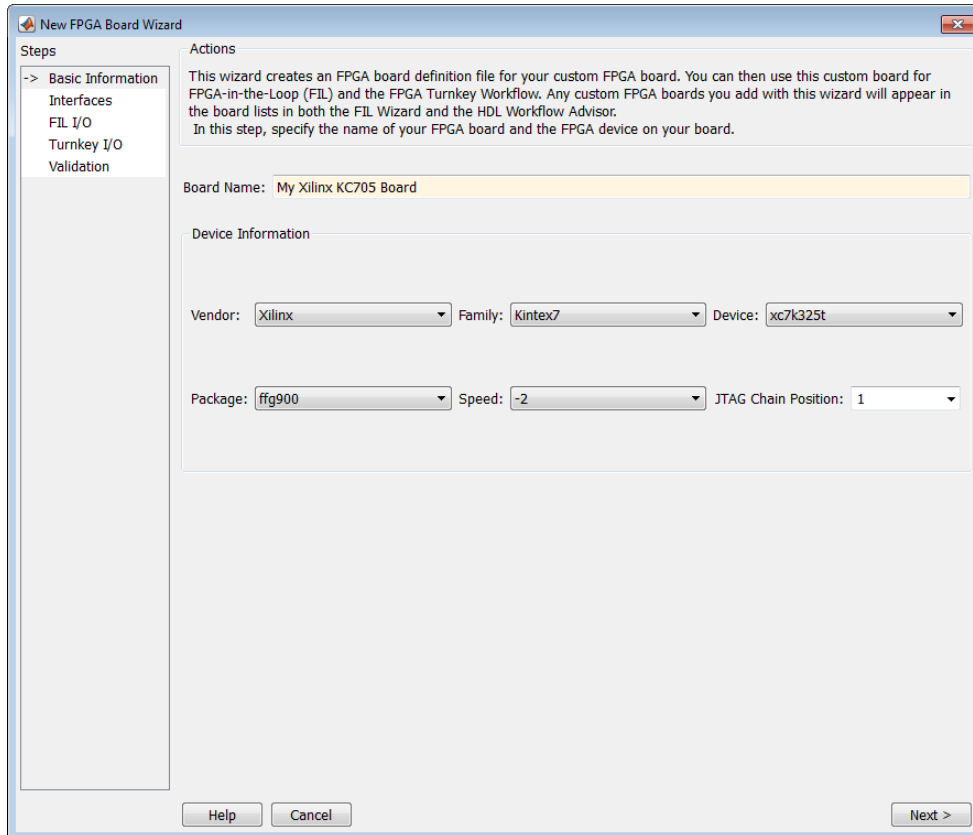
```
>>fpgaBoardManager
```
- 2 Click **Create Custom Board** to open the New FPGA Board wizard.



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.

New FPGA Board Wizard

Steps

- Basic Information
- > Interfaces
- FIL I/O
- Turnkey I/O
- Validation

Actions

Specify the board interface. To enable FPGA-in-the-Loop, select the "FIL Interface" parameter. To enable FPGA Turnkey, select the "User-defined I/O" parameter. Specify the clock and reset pin numbers. Parameter "Clock Frequency" must be between 5 and 300. For Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200MHz. Parameters "Clock_P Pin Number" and "Clock_N Pin Number" must be specified. Leave the parameter "Reset Pin Number" empty if you do not have one. Here is an example of pin number assignment: N10.

FPGA-in-the-Loop Interface

FIL Interface

PHY Interface type:

- Gigabit Ethernet - GMII
- Gigabit Ethernet - RGMII
- Ethernet - MII
- Altera JTAG

Note: FPGA-in-the-Loop requires an HDL Verifier license. FPGA-in-the-Loop uses an Ethernet PHY chip on the FPGA board to communicate with the host PC.

FPGA Turnkey Interface

User-defined I/O

Note: FPGA Turnkey requires an HDL Coder license. FPGA Turnkey supports user-defined I/O ports such as LED, UART, and push buttons.

FPGA Input Clock

Clock Frequency: 200 MHz Clock Type: Differential

Clock_P Pin Number: AD12 Clock_N Pin Number: ADII

Reset (Optional)

Reset Pin Number: AB7 Active Level: Active-High

Help Cancel < Back Next >

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.

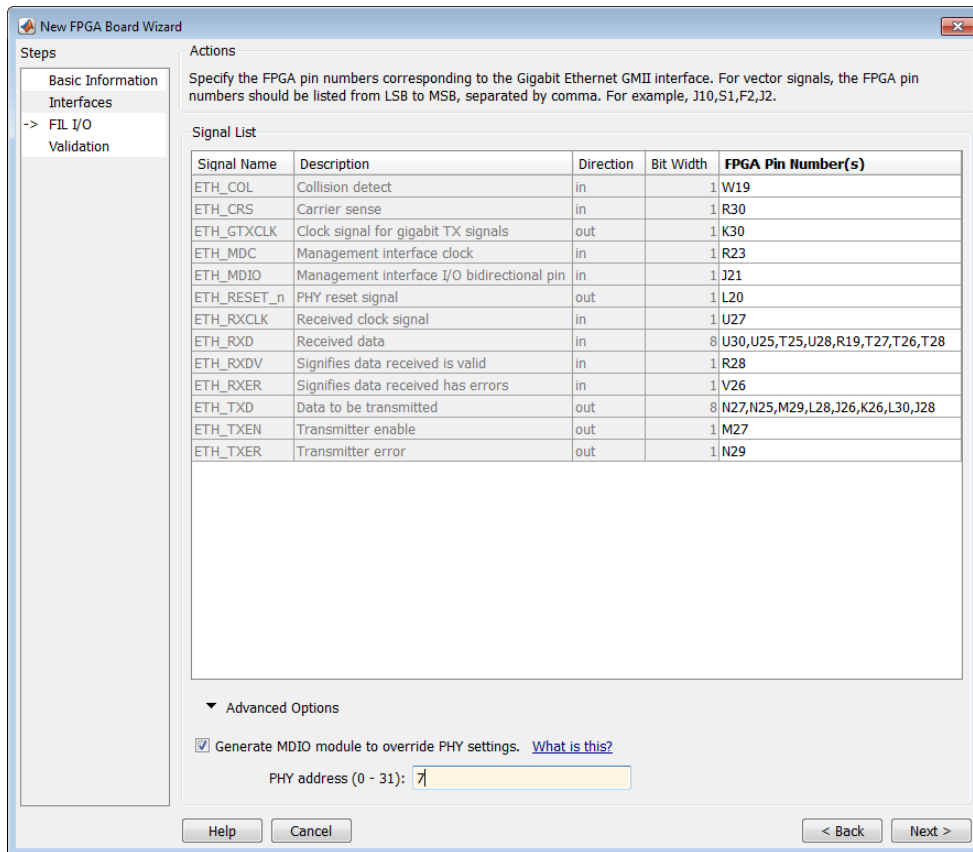
Pin numbers for RXD and TXD signals are entered from the least significant digit (LSD) to the most significant digit (MSB), separated by a comma.

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**.

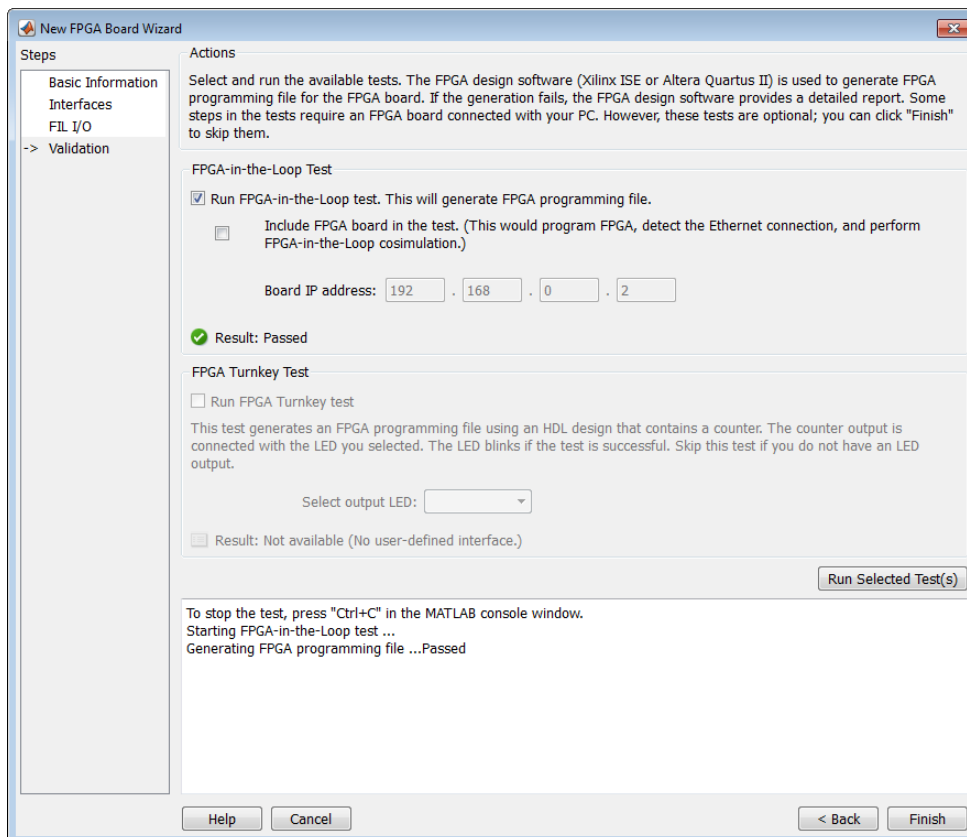
Run Optional Validation Tests

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.

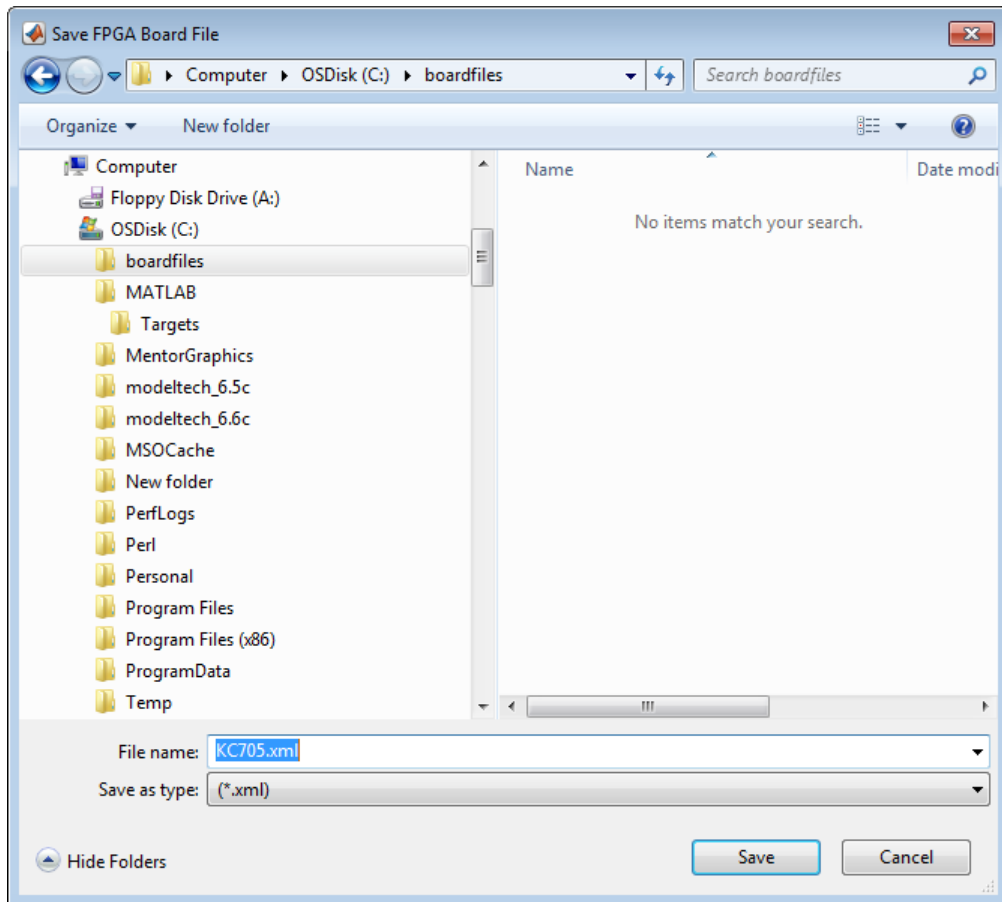
To run this test, perform the following actions.

- 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

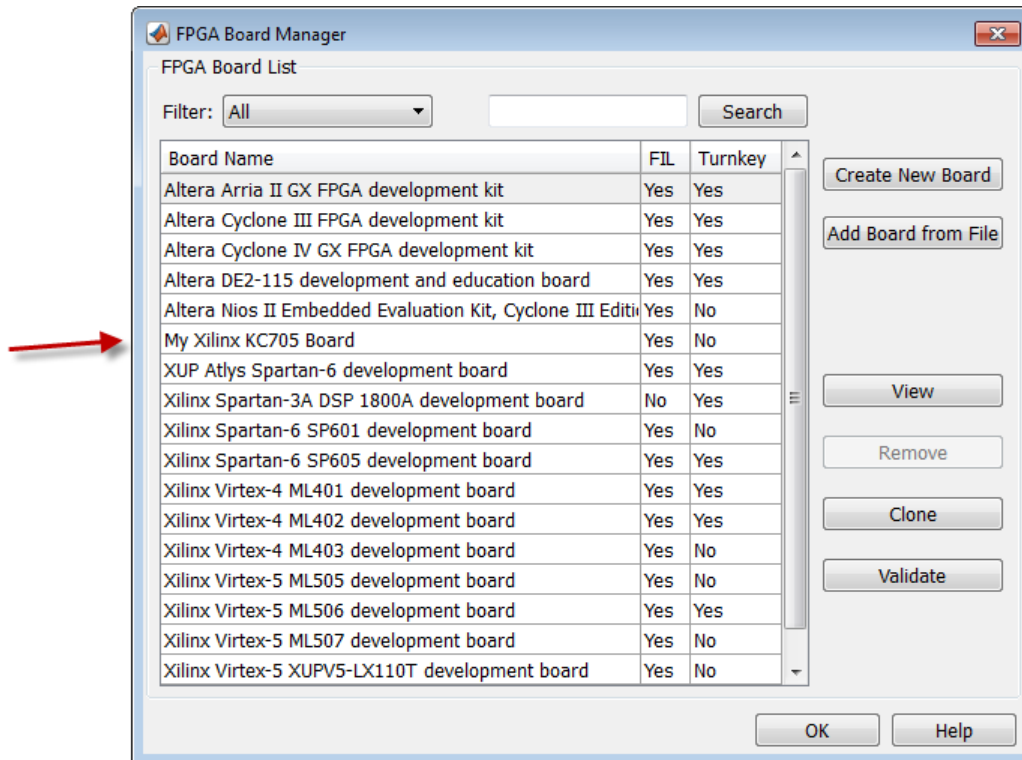
- 1 Click **Finish** to exit the New FPGA Board wizard. A **Save As** dialog box pops up and asks for the location of the FPGA board definition file. For this example, save as `C:\boardfiles\KC705.xml`.



- 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.

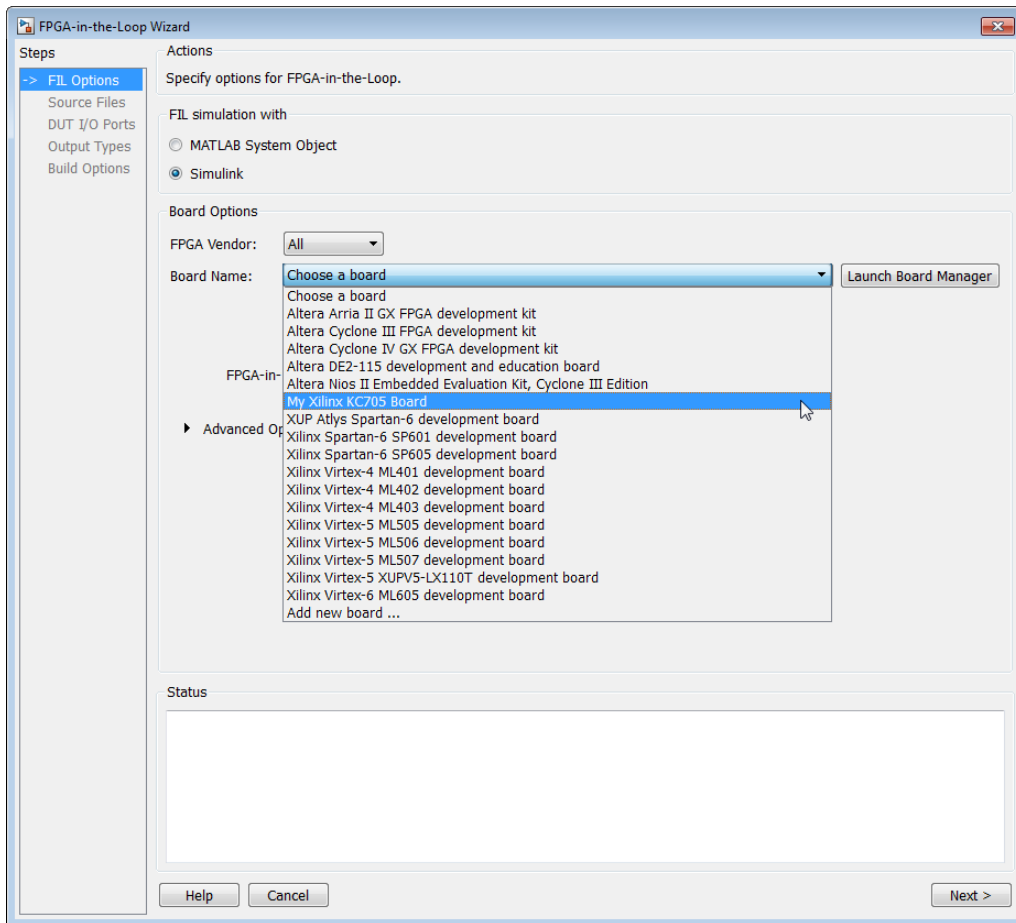


Click **OK** to close the FPGA Board Manager.

- 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
```

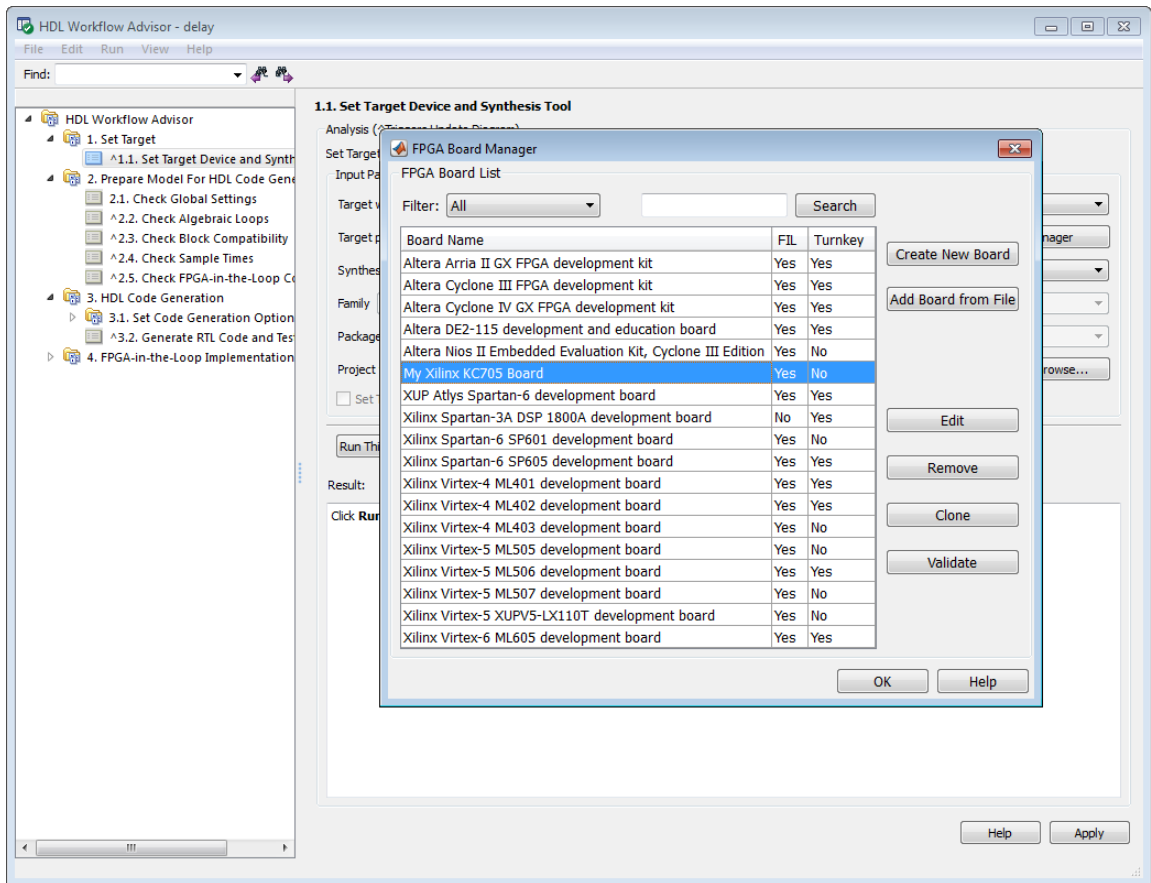
The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.



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...

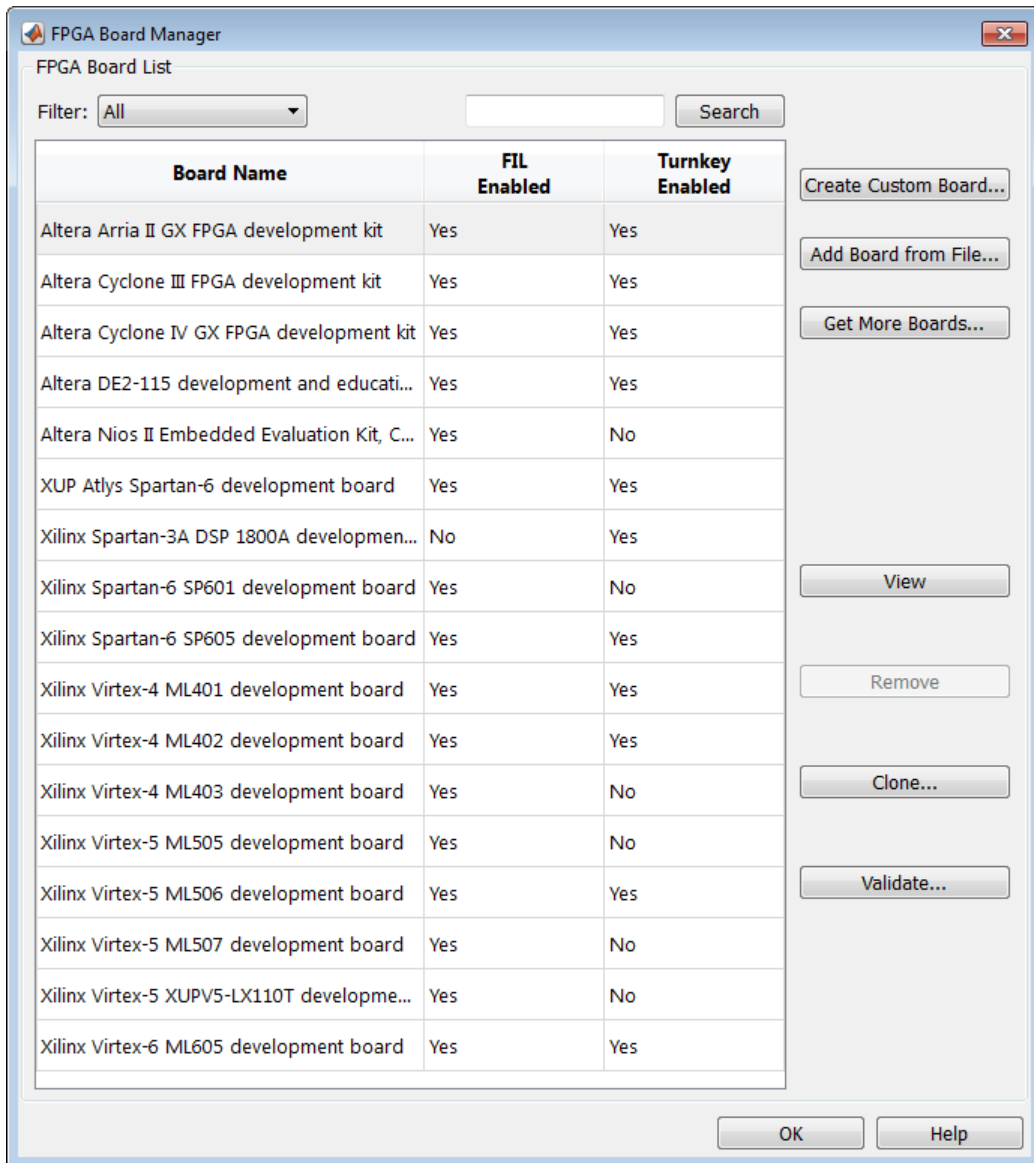
“Introduction” on page 23-21
“Filter” on page 23-23
“Search” on page 23-23
“FIL Enabled/Turnkey Enabled” on page 23-23
“Create Custom Board” on page 23-23
“Add Board from File” on page 23-23
“Get More Boards” on page 23-23
“View/Edit” on page 23-24
“Remove” on page 23-24
“Clone” on page 23-24
“Validate” on page 23-24

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 23-25. You can find the process for creating a board definition file in “Create Custom FPGA Board Definition” on page 23-7.

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.

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 23-38.

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 23-15.

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 23-3 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 23-26
“Interfaces” on page 23-27
“FIL I/O” on page 23-31

In this section...

“Turnkey I/O” on page 23-33

“Validation” on page 23-36

“Finish” on page 23-37

Basic Information

The screenshot shows the 'New FPGA Board Wizard' dialog box, currently on the 'Basic Information' step. The 'Steps' pane on the left lists 'Basic Information', 'Interfaces', 'FIL I/O', 'Turnkey I/O', and 'Validation'. The 'Actions' pane on the right 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.'

The 'Board Name' field contains the text 'HDLV Custom Board 1'. Below this is the 'Device Information' section, which includes 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 of the dialog are buttons for 'Help', 'Close', '< Back', and 'Next >'.

Board Name: Enter a unique board name.

Device Information:

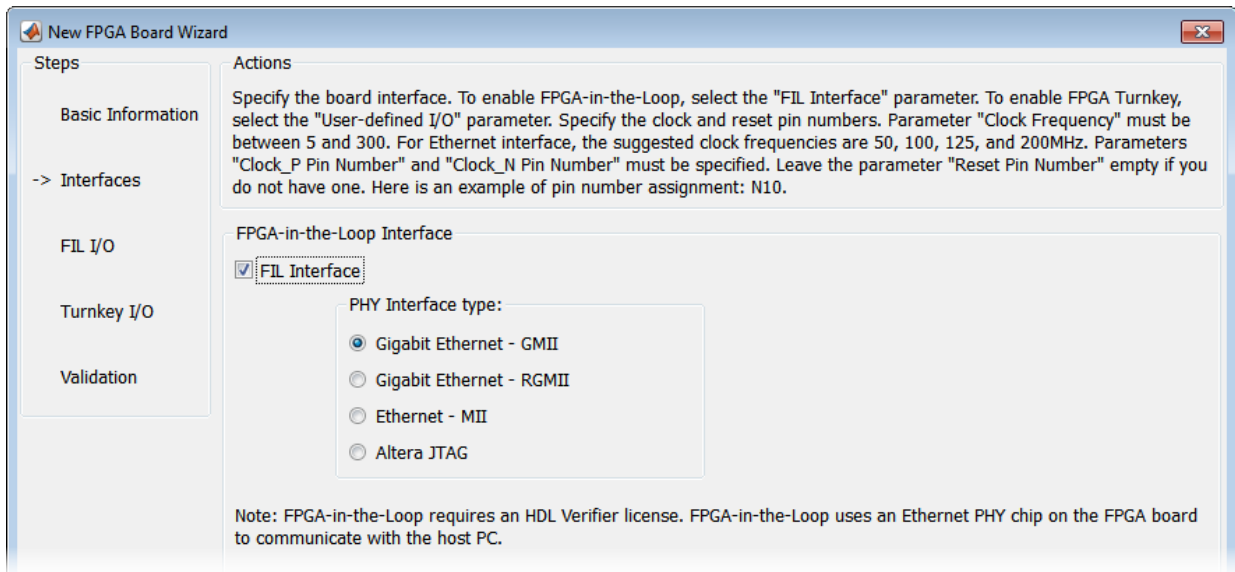
- **Vendor:** Xilinx or Altera

- **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 23-27
- “FIL Interface for Xilinx Boards” on page 23-28
- “FPGA Turnkey Interface” on page 23-29
- “FPGA Input Clock and Reset” on page 23-29

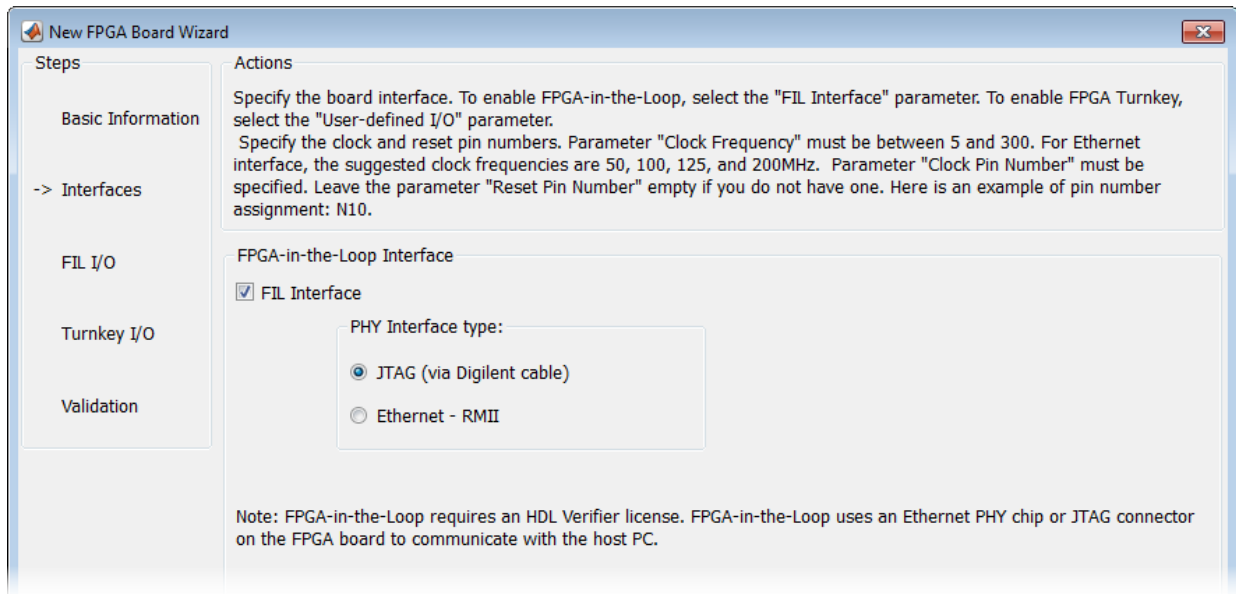
FIL Interface for Altera Boards



- 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 — RMII**

Note: Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

For more information on how to set up the JTAG connection for Xilinx boards, see “JTAG with Digilent Cable Setup” on page 23-40.

Limitations

When you simulate your FPGA design through a Digilent JTAG cable, you cannot use any other debugging feature that requires access to the JTAG; for example, the Vivado Logic Analyzer.

FPGA Turnkey Interface

FPGA Turnkey Interface

User-defined I/O

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

FPGA Input Clock

Clock Frequency: MHz Clock Type:

Clock_P Pin Number: Clock_N Pin Number:

Clock IO Standard:

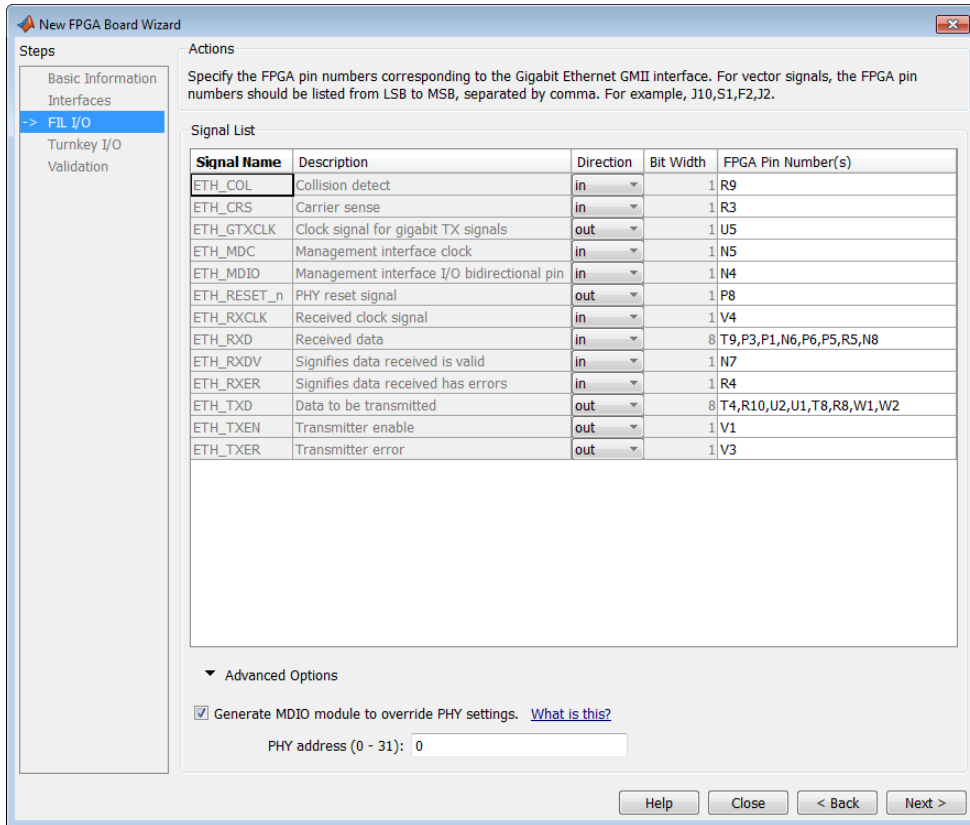
Reset (Optional)

Reset Pin Number: Active Level:

Reset IO Standard:

- 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



Note: You provide FIL I/O for an Ethernet connection only.

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.

Generate MDIO module to override PHY settings: See the next section on FPGA Board Management Data Input/Output Bus (MDIO) to determine when to use this feature. If you do select this option, enter the PHY address.

What Is the Management Data Input/Output Bus?

Management Data Input/Output (MDIO) is a serial bus, defined in the IEEE 802.3 standard, that connects MAC devices and Ethernet PHY devices. The FPGA MAC uses the MDIO bus to set control registers in the Ethernet PHY device on the board.

Currently only the Marvell 88E1111 PHY chip is supported by this MDIO module implementation. Do not select this check box if you are not using Marvell 88E1111.

The generated MDIO module is used to perform the following operations:

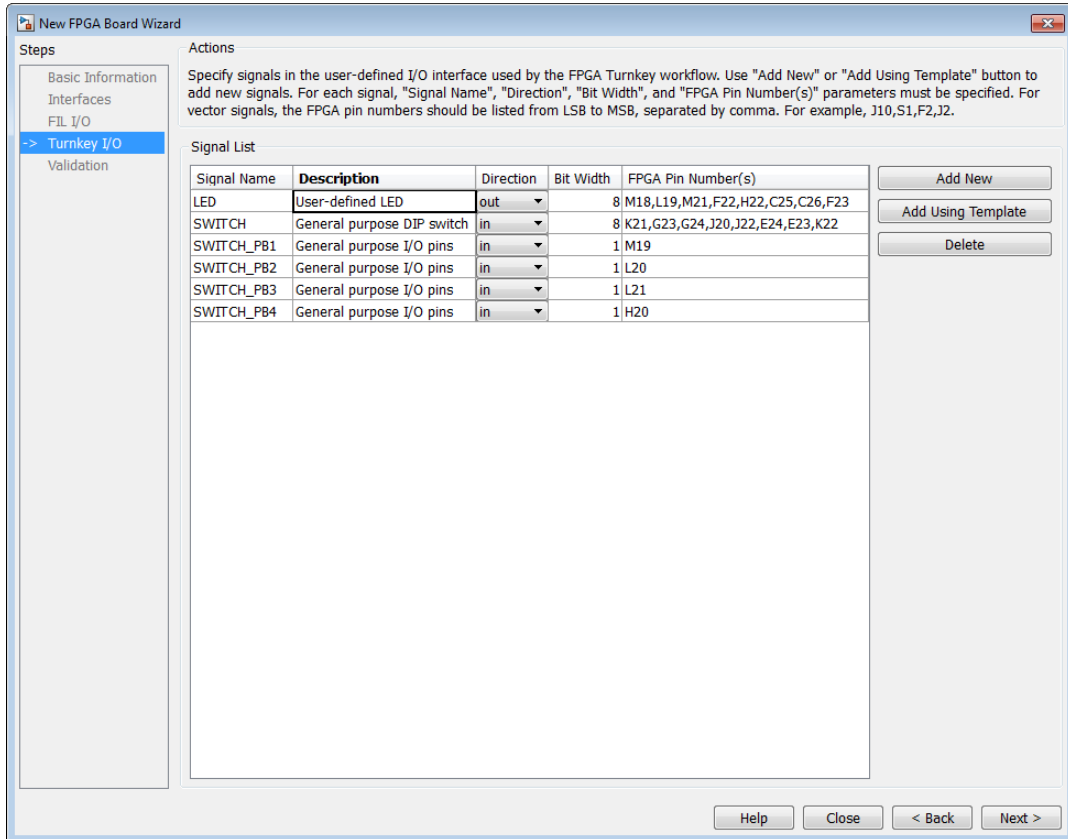
- **GMII mode:** The PHY device can start up using other modes, such as RGMII/SGMII. The generated MDIO module sets the PHY chip in GMII mode.
- **RGMII mode:** The PHY device can start up using other modes, such as GMII/SGMII. The generated MDIO module sets the PHY device in RGMII mode. In addition, the module sets the PHY chip to add internal delay for RX and TX clocks.
- **SGMII mode:** The PHY device can start up using other modes, such as RGMII/GMII. The generated MDIO module sets the PHY chip in SGMII mode.
- **MII mode:** The generated MDIO module sets the PHY device in GMII compatible mode. The module also sets the autonegotiation register to remove the 1000 Base-T capability advertisement. This reset ensures that the autonegotiation process does not select 1000 Mbits/s speed, which is not supported in MII mode.

When To Select MDIO: Select the **Generate MDIO module to override PHY settings** option when both the following conditions are met:

- The onboard Ethernet PHY device is Marvell 88E1111.
- The PHY device startup settings are not compatible with the FPGA MAC. The MDIO modules for different PHY modes must override these settings, as previously described.

Specifying the PHY Address: The PHY address is a 5-bit integer. The value is determined by the CONFIG[0] and CONFIG[1] pin on Marvell 88E1111 PHY device. See the board manual for this value.

Turnkey I/O



Note: Provide FIL I/O for an Ethernet connection only. Define at least one output port for the Turnkey I/O interface.

Signal List: Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas. The number of pin numbers must match the bit width of the corresponding signal.

Add New: You are prompted to enter all entries in the signal list manually.

Add Using Template: The wizard prepopulates a new signal entry for UART, LED, GPIO, or DIP Switch signals with the following:

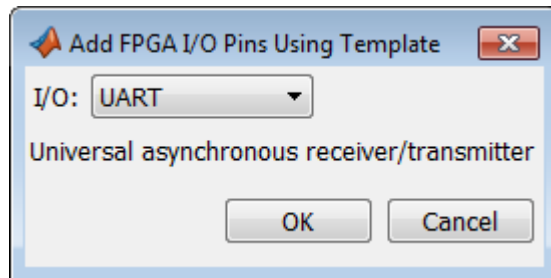
- A generic signal name
- Description
- Direction
- Bit width

You can change the values in any of these prepopulated fields.

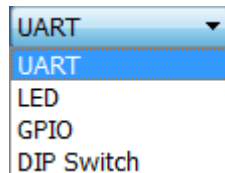
Delete: Delete the selected signal from list.

The following example demonstrates using the **Add Using Template** feature.

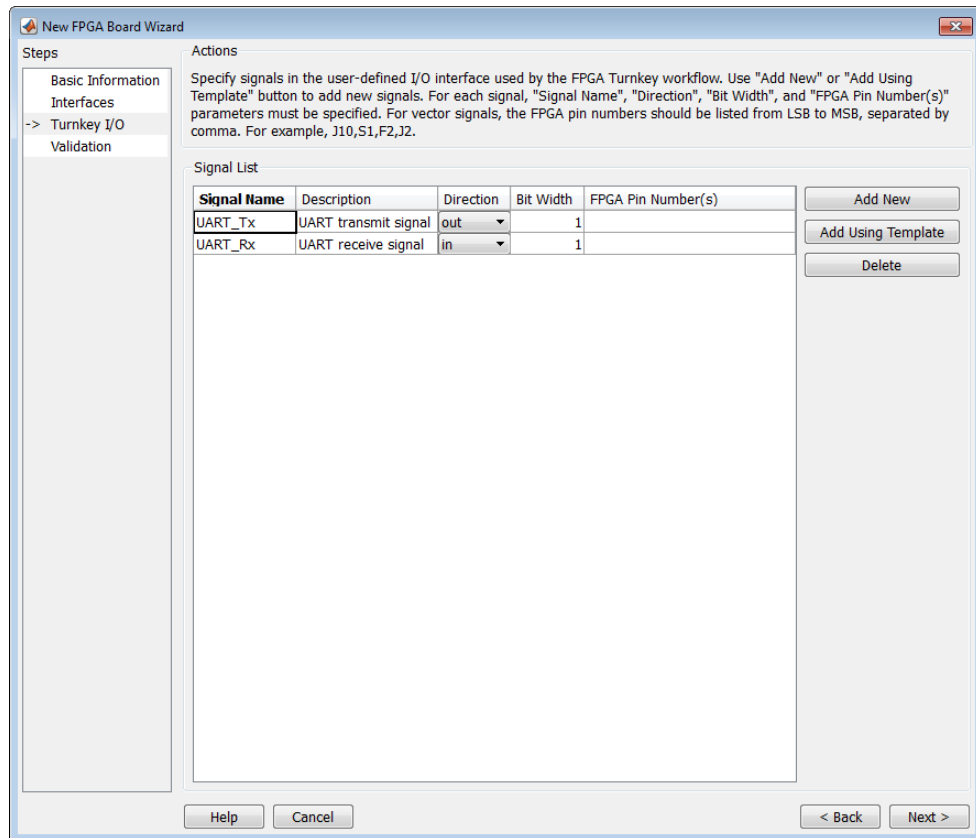
- 1 In the Turnkey I/O dialog box, click **Add Using Template**.
- 2 You can now view the template dialog box.



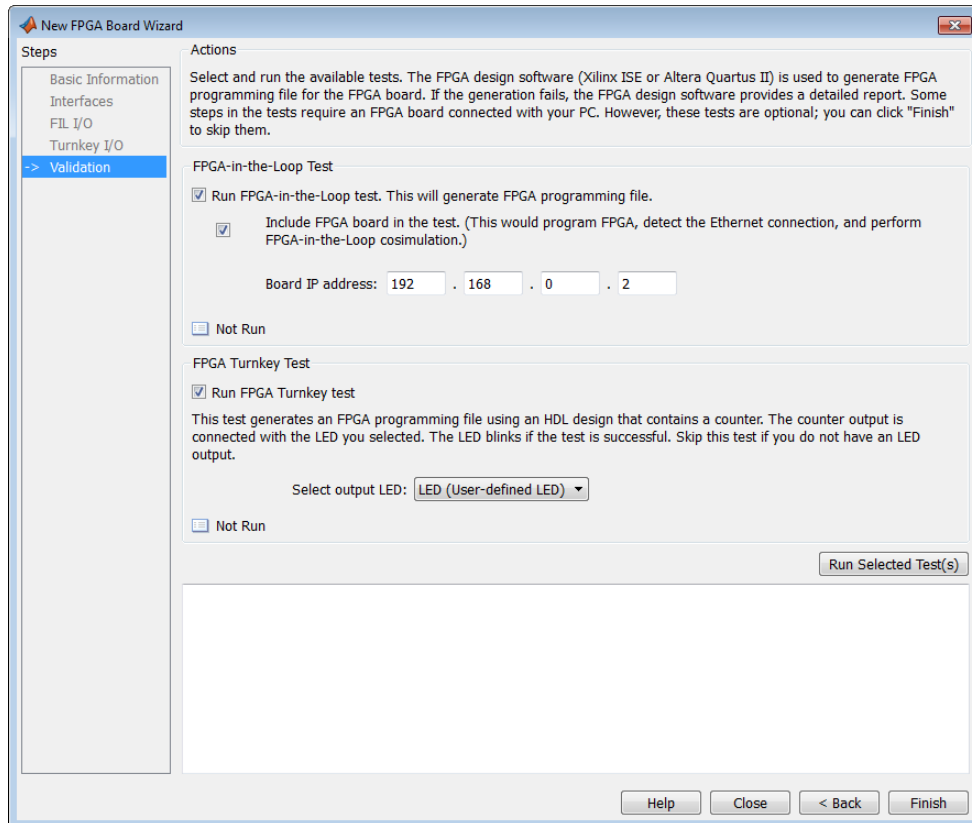
- 3 Pull down the I/O list and select from the following options:



- 4 Click **OK**.
- 5 The wizard adds the specified signal (or signals) to the I/O list.



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 23-17.

FPGA Board Editor

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.

In this section...

“General Tab” on page 23-38

“Interface Tab” on page 23-40

General Tab

Xilinx Virtex-7 VC707 development board - Copy (S:\Xilinx_pcie\zedboard\camera\matlab\new...)

Action
Specify your FPGA board information.

General **Interface**

Enter the basic information about your FPGA board such as board name, FPGA specification, and clock and reset pin numbers.

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

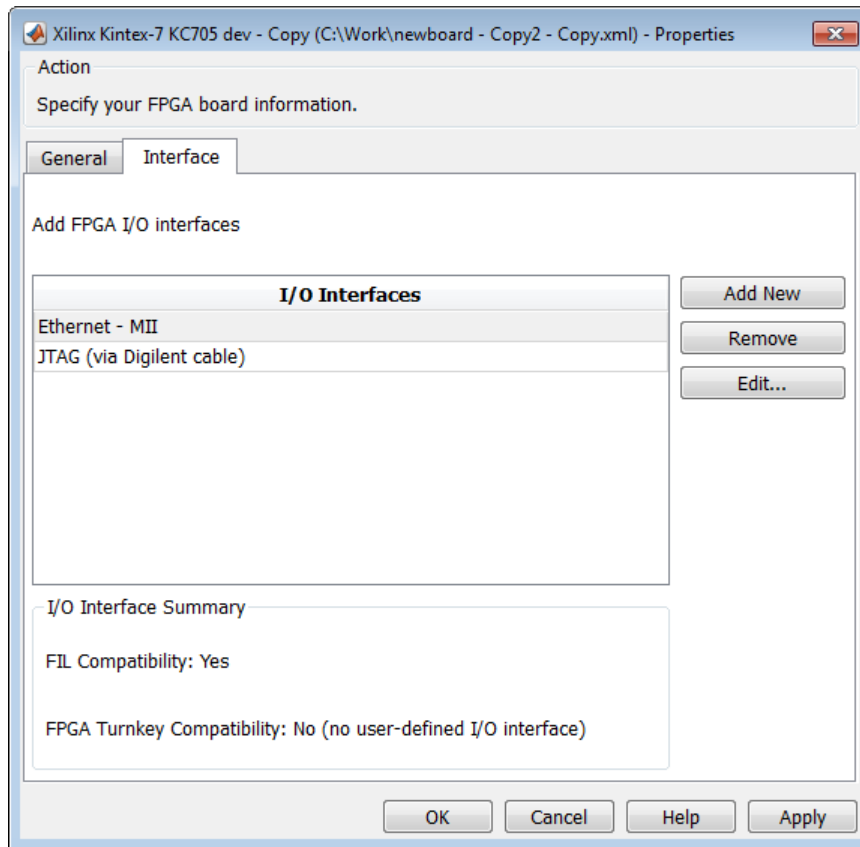
OK Cancel Help Apply

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.

JTAG (via Digilent cable)

Action

Specify the instruction register (IR) lengths for

(a) The sum of IR lengths for devices before FPGA in the chain

(b) The sum of IR lengths for devices after FPGA in the chain

Signal/Parameter List

Sum of IR length before:

Sum of IR lengths after:

Example:

Diagram illustrating a JTAG chain configuration:

```

graph LR
    TDI --> DeviceA[Device A  
IR Length=4]
    DeviceA -- TDO --> TDI2[TDI]
    TDI2 --> FPGA[FPGA]
    FPGA -- TDO --> TDI3[TDI]
    TDI3 --> DeviceB[Device B  
IR Length=3]
    DeviceB -- TDO --> TDI4[TDI]
    TDI4 --> DeviceC[Device C  
IR Length=2]
    DeviceC -- TDO --> TDO_out[TDO]
  
```

An example of JTAG chain configuration is shown in the above figure. Assume that the instruction register lengths are Device A: 4, Device B: 3, Device C: 2. Then the answers are:

(a) Sum of IR lengths before = 4

(b) Sum of IR lengths after = 3+2 = 5

If the FPGA is the only device on the chain, enter 0 for both fields. To find more information regarding JTAG chain configuration, please click "Help" button.

Advanced Options

User1 Instruction:

JTAG Clock Frequency (MHz):

OK Cancel Help Apply

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:

HDL Workflow Advisor Tasks

HDL Workflow Advisor Tasks

In this section...

- “HDL Workflow Advisor Tasks Overview” on page 24-3
- “Set Target Overview” on page 24-5
- “Set Target Device and Synthesis Tool” on page 24-6
- “Set Target Library” on page 24-7
- “Set Target Interface” on page 24-8
- “Set Target Frequency” on page 24-9
- “Set Target Interface” on page 24-10
- “Set Target Interface” on page 24-11
- “Prepare Model For HDL Code Generation Overview” on page 24-12
- “Check Global Settings” on page 24-13
- “Check Algebraic Loops” on page 24-14
- “Check Block Compatibility” on page 24-15
- “Check Sample Times” on page 24-16
- “Check FPGA-in-the-Loop Compatibility” on page 24-17
- “HDL Code Generation Overview” on page 24-18
- “Set Code Generation Options Overview” on page 24-19
- “Set Basic Options” on page 24-20
- “Set Advanced Options” on page 24-21
- “Set Testbench Options” on page 24-22
- “Generate RTL Code and Testbench” on page 24-23
- “Generate RTL Code and IP Core” on page 24-25
- “FPGA Synthesis and Analysis Overview” on page 24-26
- “Create Project” on page 24-27
- “Perform Synthesis and P/R Overview” on page 24-28
- “Perform Logic Synthesis” on page 24-29
- “Perform Mapping” on page 24-30
- “Perform Place and Route” on page 24-31

In this section...

“Run Synthesis” on page 24-31

“Run Implementation” on page 24-31

“Annotate Model with Synthesis Result” on page 24-33

“Download to Target Overview” on page 24-35

“Generate Programming File” on page 24-36

“Program Target Device” on page 24-37

“Generate Simulink Real-Time Interface” on page 24-38

“Save and Restore HDL Workflow Advisor State” on page 24-39

“FPGA-in-the-Loop Implementation” on page 24-39

“Set FIL Options” on page 24-39

“Build FPGA-in-the-Loop” on page 24-40

“Check USRP® Compatibility” on page 24-40

“Verify with HDL Cosimulation” on page 24-40

“Generate FPGA Implementation” on page 24-40

“Check SDR Compatibility” on page 24-40

“SDR FPGA Implementation” on page 24-41

“Set SDR Options” on page 24-41

“Build SDR” on page 24-43

“Embedded System Integration” on page 24-43

“Create Project” on page 24-43

“Generate Software Interface Model” on page 24-43

“Build FPGA Bitstream” on page 24-44

“Program Target Device” on page 24-44

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:

- **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 specifically 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 might 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

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 Overview

The tasks in the **Set Target** folder enable you to select a target FPGA device and define the I/O interface to be generated for the device. The **Set Target** folder contains the following tasks:

- **Set Target Device and Synthesis Tool:** Select a target FPGA device and synthesis tools.
- **Set Target Interface:** Use the Target Platform Interface Table to assign each port on your DUT to an I/O resource on the target device.

See Also

For summary information on each **Set Target** task, select the task icon and then click the HDL Workflow Advisor **Help** button.

Set Target Device and Synthesis Tool

Set Target Device and Synthesis Tool 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. Select a Xilinx or Altera tool to make the **Set Target Library (for floating-point synthesis support)** option available.

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.
- **Set Target Library (for floating-point synthesis support):** Select to map to an FPGA target-specific floating-point library. Enabling this option causes the **Set Target Library** task to appear on the left. This option is not available if your synthesis tool is Xilinx Vivado.

See Also

For information on the Set Target Library task, see “Set Target Library” on page 24-7.

Set Target Library

Target library: The selected FPGA floating-point target library.

Objective: Choose to optimize your generated HDL code for **Speed** or **Area**.

Block latencies: Select the block latencies to use.

Set Xilinx simulation path: Select to enter the location of your pre-compiled Xilinx simulation library (`xilinxcorelib`). Do not select this option if you wish HDL Coder to automatically detect the location of the simulation library. This option is available only if you selected a Xilinx synthesis tool in the **Set Target Device and Synthesis Tool** task. If the pre-compiled Xilinx simulation library is unavailable, HDL Coder issues a warning.

Absolute path: Enter the location of the simulation library. This option is available if **Set Xilinx simulation path** is selected.

See Also

For more information on targeting FPGA floating-point library blocks, see “FPGA Floating-Point Library Mapping”.

Set Target Interface

Set Target Interface displays properties of input and output ports on your model, 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 model
- 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

Automatically generate clock module for FPGA Turnkey or Simulink Real-Time FPGA I/O targets.

Leave entry unchanged if you wish to use the default value (same as input).

Set Target Interface

Select a processor-FPGA synchronization mode, and map your model's 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 automatically generate synchronization logic for the FPGA 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 automatically generate synchronization logic for the FPGA 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 model.
- 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”
- “Custom IP Core Generation”
- “Generate Simulink Real-Time Interface for Speedgoat Boards”

Set Target Interface

Select a processor-FPGA synchronization mode, and map your model's 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 automatically generate synchronization logic for the FPGA 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 automatically generate synchronization logic for the FPGA 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 model.
- 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”
- “Custom IP Core Generation”
- “Generate Simulink Real-Time Interface for Speedgoat Boards”

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 2 input ports and 2 output ports of signed 16-bit signals.

This option is available only if you select `Customization for the USRP(TM) Device` for Target workflow.

See Also

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.

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 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.

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. You should eliminate algebraic loops from your model before proceeding with further HDL Workflow Advisor checks or code generation.

See Also

For information about algebraic loops, see “Algebraic Loops” in the Simulink documentation.

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

See “Generate a Library of Supported Blocks” for a complete list of supported blocks and their implementations.

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

For HDL code and model compatibilities with FPGA-in-the-Loop processing, see “Prepare DUT For FIL Interface Generation”.

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

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. See “HDL Code Generation Pane: General ” for information on each parameter.
- **Set Advanced Options:** Set parameters that specify detailed characteristics of the generated code, such as HDL element naming and whether certain optimizations apply. See “HDL Code Generation Pane: Global Settings ” for information on each parameter.
- **Set Testbench Options:** Set options that determine characteristics of generated test bench code. See “HDL Code Generation Pane: Test Bench ” for information on each parameter.

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

See also “HDL Code Generation Pane: General”.

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.

See Also

See also “HDL Code Generation Pane: Global Settings” on page 10-22.

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.

See Also

See also “HDL Code Generation Pane: Test Bench ”.

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 RTL testbench:** Generate an RTL test bench in the target language.
- **Generate cosimulation model** (requires HDL Verifier): Generate a cosimulation model. Selecting this check box enables the next option.
- **Cosimulation model for use with:** Select one of the following options from the menu:
 - **Mentor Graphics ModelSim:** This option is the default. If your installation includes HDL Verifier for use with Mentor Graphics ModelSim, HDL Coder generates and opens a Simulink model that contains an HDL Cosimulation block for Mentor Graphics ModelSim.
 - **Cadence Incisive:** If your installation includes HDL Verifier for use with Cadence Incisive, HDL Coder generates and opens a Simulink model that contains an HDL Cosimulation block for Cadence Incisive.
- **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.

- **Generate FPGA top level wrapper:** Generate an HDL code wrapper and a constraint file that contains pin map information and clock constraints. When you select a specific target device in the **Set Target Device and Synthesis Tool** task, **Generate FPGA top level wrapper** is automatically selected. Generating this wrapper enables generation of the corresponding programming file for the **Generate Programming File** task in the **Download to Target** folder.

When you select **Generate FPGA top level wrapper**, the task **Annotate Model with Synthesis Result** is not available in the **FPGA Synthesis and Analysis**

folder. To perform back-annotation analysis, clear the check box for **Generate FPGA top level wrapper**.

See Also

See also “Generating a Simulink Model for Cosimulation with an HDL Simulator”.

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.

- **Generate IP core report:** Select this option to generate HTML documentation for the IP core.

See Also

- “Custom IP Core Generation”
- “Generate a Board-Independent IP Core from Simulink”
- “Custom IP Core Report”

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”.

The tasks in the folder are:

- **Create Project**
- **Perform Synthesis and P/R**
- **Annotate Model with Synthesis Result**

See Also

See also “FPGA Synthesis and Analysis”.

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.

Enter additional files you want included in your synthesis project. Enter each file name manually, separated with a semicolon (;), or by using the **Add** button.

For example, you can include HDL source files (.vhd or .v), a constraint file (.ucf or .sdc), or a Tcl script (.tcl) to execute after creating the project.

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.

For a list of supported third-party synthesis tools, see “Third-Party Synthesis Tools”.

See Also

See also “Creating a Synthesis Project”.

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”.

See Also

See also “FPGA Synthesis and Analysis”

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

See also “Performing Synthesis, Mapping, and Place and Route”.

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

See also “Performing Synthesis, Mapping, and Place and Route”.

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

See also “Performing Synthesis, Mapping, and Place and Route”.

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

“Annotating Your Model with Critical Path Information”

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.

See Also

For summary information on each **Download to Target** task, select the task icon and then click the HDL Workflow Advisor **Help** button.

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_xpc.mdl
```

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*. At a later time, you can restore the same settings by loading the restore point data into the HDL Workflow Advisor.

See Also

For detailed information on how to create, save, and load a restore point, see “Save and Restore HDL Workflow Advisor State”.

FPGA-in-the-Loop Implementation

Set FIL options and run FIL processing.

Set FIL 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 board's IP address 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 will attempt 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 window 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 2 input ports and 2 output ports of signed 16-bit signals.

Verify with HDL Cosimulation

Run this step to verify the generated HDL using cosimulation between the HDL Simulator and the Simulink test bench.

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
- Analog Devices AD FMCOMMS1-ABZ RevB

- **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 FMCOMMS 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 Input Parameter	Description
Embedded system tool	Embedded design tool.
Project folder	Folder where your generated project files are saved.

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.

Add IP core device driver: Select to insert the IP core node into the operating system device tree on the SD card on your board, reboot the operating system, and add the IP core driver as a loadable kernel module. To use this option, your board must be connected. Back up your SD card before you run this task.

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.

Program Target Device

Program the connected target device.

Click **Run** to program your connected target device.

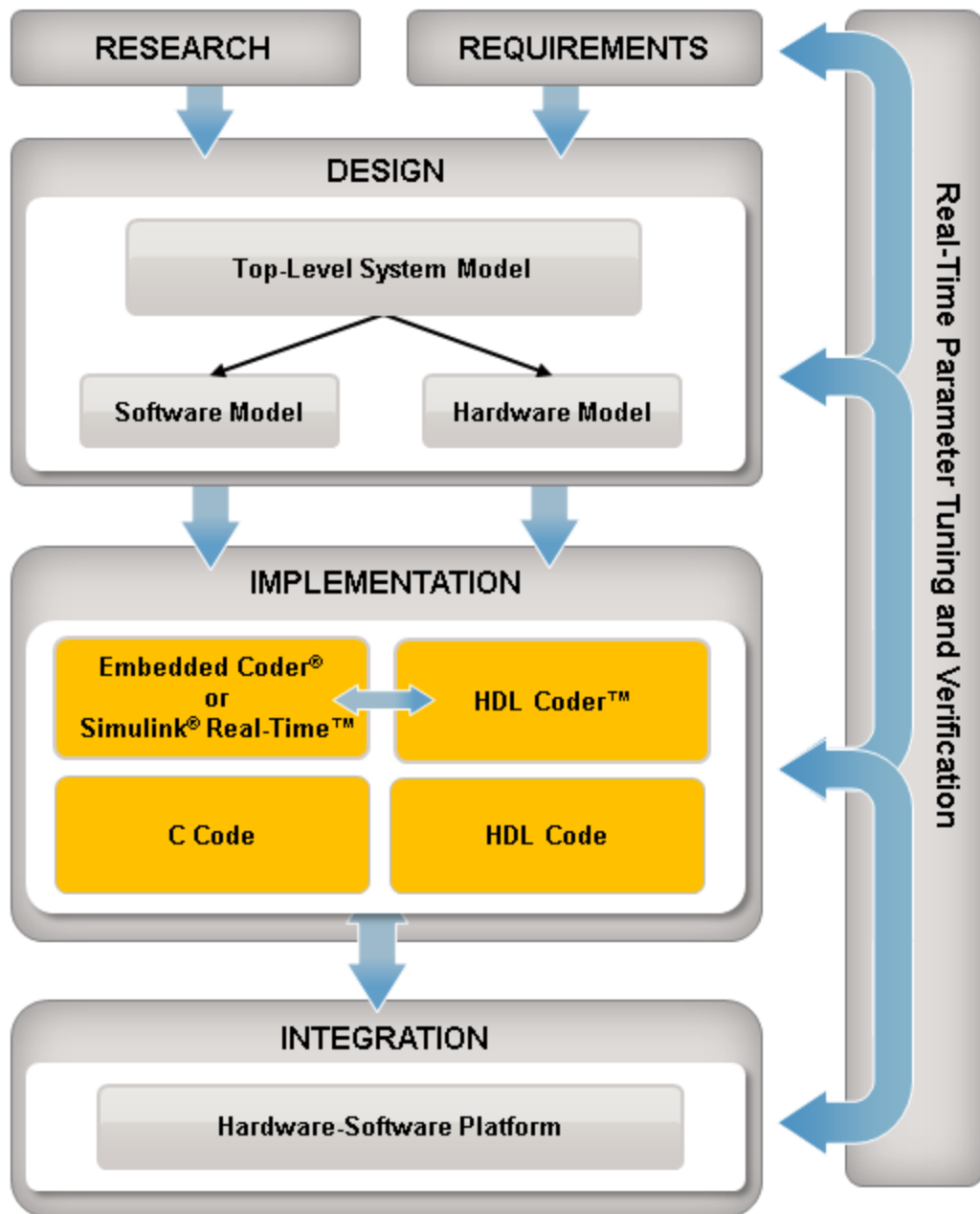
Hardware-Software Codesign

Hardware-Software Codesign Basics

- “Hardware-Software Codesign Workflow” on page 25-2
- “Custom IP Core Generation” on page 25-5
- “Custom IP Core Report” on page 25-8
- “Generate a Board-Independent IP Core from Simulink” on page 25-13
- “Processor and FPGA Synchronization” on page 25-17

Hardware-Software Codesign Workflow

The hardware-software codesign workflow is similar in concept for all target platforms.



- 1 Create a design based on your research and requirements.
- 2 Within your design, create software and hardware partitions to deploy as hardware and software on your target platform.
- 3 For the hardware partition, generate HDL code; for the software partition, write or generate C code.
- 4 Build, download, and integrate the HDL and C code on your target platform.

Custom IP Core Generation

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 Altera Qsys, Xilinx EDK, or Xilinx IP Integrator.

To learn how to generate a custom IP core from Simulink, see “Generate a Board-Independent IP Core from Simulink”.

To learn how to generate a custom IP core from a MATLAB design, see “Generate a Board-Independent IP Core from MATLAB”.

In this section...

“Custom IP Core Architectures” on page 25-5

“Target Platform Interfaces” on page 25-6

“Processor/FPGA Synchronization” on page 25-6

“Custom IP Core Generated Files” on page 25-7

Custom IP Core Architectures

You can generate an IP core with an AXI4 or AXI4-Lite interface. You can also generate an IP core with an AXI4 or AXI4-Lite interface and AXI4-Stream Video interfaces.

An IP core with an AXI4 or AXI4-Lite interface:



An IP core with an AXI4 or AXI4-Lite interface and AXI4-Stream Video interfaces:



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”.

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.

Custom IP Core Report

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.

In this section...
“Summary” on page 25-8
“Target Interface Configuration” on page 25-8
“Register Address Mapping” on page 25-9
“IP Core User Guide” on page 25-10
“IP Core File List” on page 25-11

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	hdlcoder_led_blinking_led_counter_pcore
IP core version	v1.00.a
IP core folder	hdl_prj\ipcore\hdlcoder_led_blinking_led_counter_pcore_v1_00_a
Target platform	Generic Xilinx Platform
Target language	VHDL
Model	hdlcoder_led_blinking
Model version	1.65
HDL Coder version	3.3
IP core generated on	26-Mar-2013 12:05:14
IP core generated for	led_counter

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 [hdlcoder_led_blinking](#) :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
Blink_frequency	Inport	ufix4	AXI4-Lite	x"100"
Blink_direction	Inport	boolean	AXI4-Lite	x"104"
LED	Outport	uint8	External Port	
Read_back	Outport	uint8	External Port	

To learn more about processor/FPGA synchronization modes, see “Processor and FPGA Synchronization”.

To learn more about target platform interfaces, see “Custom IP Core Generation”.

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-Lite bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
Blink_frequency_Data	0x100	data register for port Blink_frequency
Blink_direction_Data	0x104	data register for port Blink_direction

The register address mapping is also in the following C header file for you to use when programming the processor: [include\hdlcoder_led_blinking_led_counter_pcore_addr.h](#)

The IP core name is appended to the register names to avoid name conflicts.

IP Core User Guide

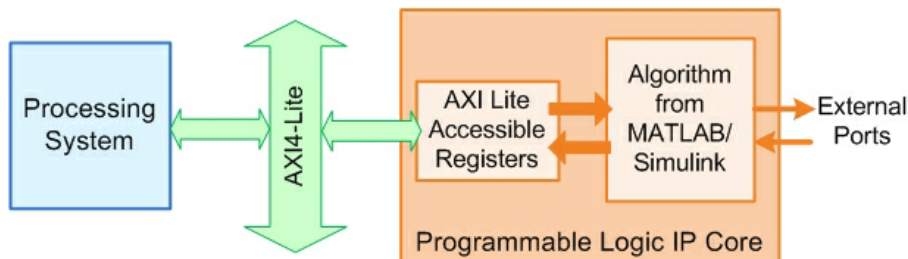
The IP Core User Guide section gives a high-level overview of the system architecture, describes the processor and FPGA synchronization mode, and gives instructions for integrating the IP core in your embedded system integration environment.

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.

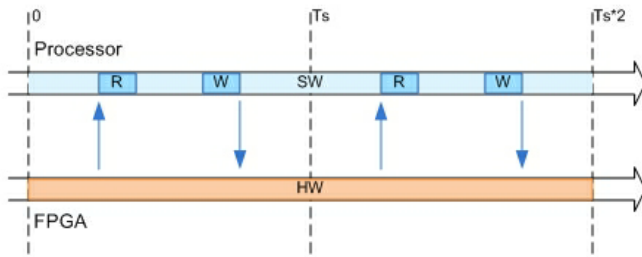


This IP core also support the **External Port** interface. To connect the external ports to the FPGA external IO pins, add FPGA pin assignment constraints in the Xilinx EDK environment.

The following figure is an example of a processor/FPGA synchronization description.

Processor/FPGA Synchronization

The **Free running** mode means there is no explicit synchronization between embedded processor software execution (SW) and the IP core (HW). SW and HW runs independently. The data written from the processor to IP core takes effect immediately, and the data read from the IP core is the latest data available on the IP core output ports.



The following figure is an example of instructions for integrating the IP core into your embedded system integration 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](#)

Generate a Board-Independent IP Core from Simulink

In this section...

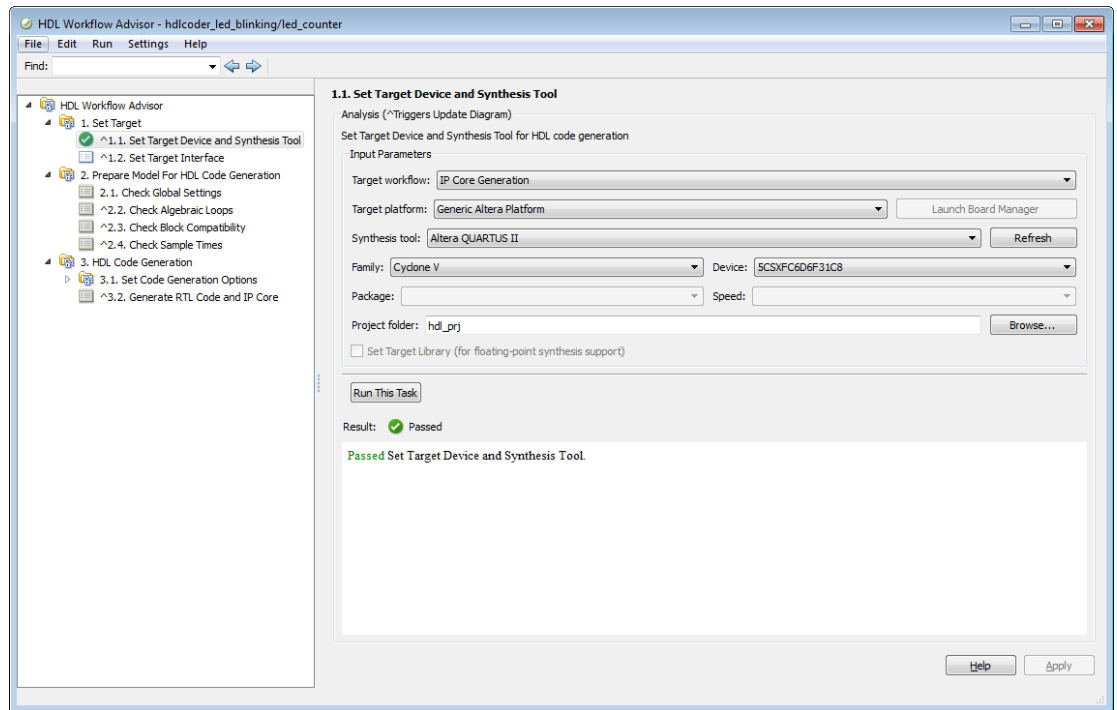
“Generate a Board-Independent IP Core” on page 25-13

“Requirements and Limitations for IP Core Generation” on page 25-16

Generate a Board-Independent IP Core

To generate a board-independent custom IP core to use in an embedded system integration environment, such as Altera Qsys, Xilinx EDK, or Xilinx IP Integrator:

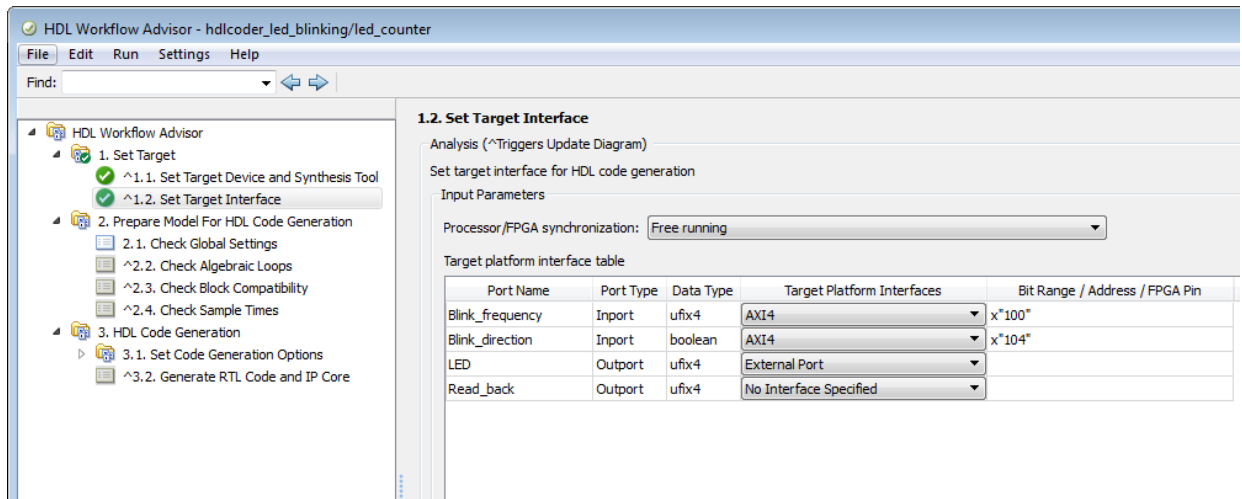
- 1 Select your DUT in your Simulink model and open the HDL Workflow Advisor.
- 2 In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select IP Core Generation.
- 3 For **Target platform**, select **Generic Altera Platform** or **Generic Xilinx Platform** and click **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 the following interfaces:

- **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 Port:** Use the external ports to connect to FPGA external IO pins, or to other IP cores with external ports.



- 5 If you want to set options in the other HDL Workflow Advisor tasks, set them.
- 6 In the **HDL Code Generation > Generate RTL Code and IP Core** task, set the following fields:
 - **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.
 - **Generate IP core report:** Enable this option to generate HTML documentation for the IP core.
- 7 Right-click the **HDL Code Generation > Generate RTL Code and IP Core** task and select **Run to Selected 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.

To learn more about custom IP core generation, see “Custom IP Core Generation”.

Requirements and Limitations for IP Core Generation

To generate a custom IP core:

- The DUT must be an atomic system.
- There cannot be both an AXI4 interface and AXI4-Lite interface in the same IP core.
- The DUT cannot contain Xilinx System Generator blocks or Altera DSP Builder Advanced blocks.
- If your target language is VHDL, and your synthesis tool is Xilinx ISE or Altera Quartus II, the DUT cannot contain a model reference.

To map your DUT ports to an AXI4-Lite interface, the input and output ports must:

- Have a bit width less than or equal to 32 bits.
- Be scalar.

When mapping your DUT ports to an AXI4-Stream Video interface, the following requirements and limitations apply:

- Ports must have a 32-bit width.
- Ports must be scalar.
- The model must be single rate.
- 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 mode.

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:

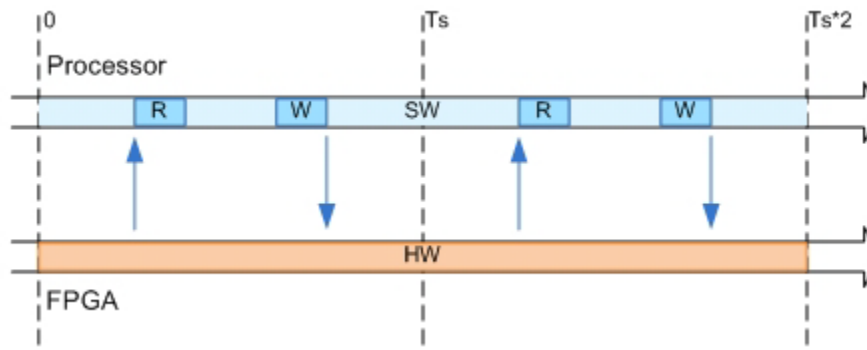
- Free running (default)
- Coprocessing – blocking
- Coprocessing – nonblocking with delay (available only for the **Simulink Real-Time FPGA I/O** workflow)

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.

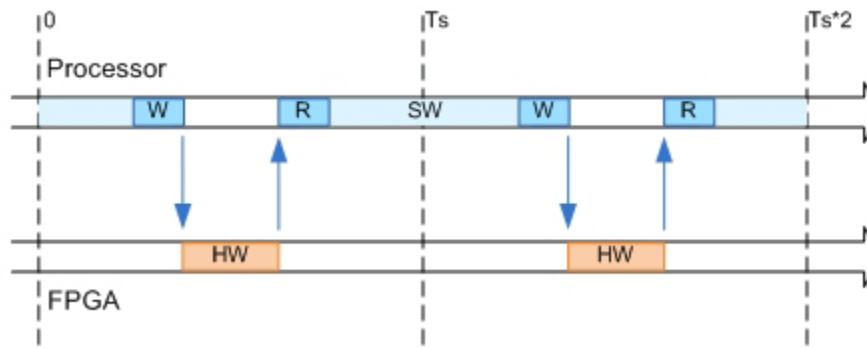


Coprocessing – Blocking Mode

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.



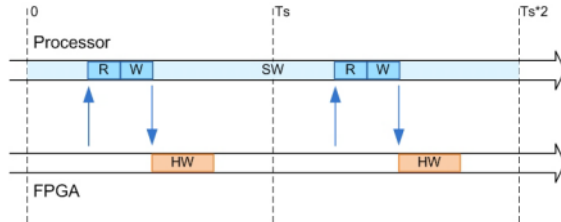
The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor writes to the FPGA, then stops and waits for an indication that the FPGA has finished processing before continuing to run. Each time the FPGA runs, it executes the logic generated for one DUT subsystem 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 available only for 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.



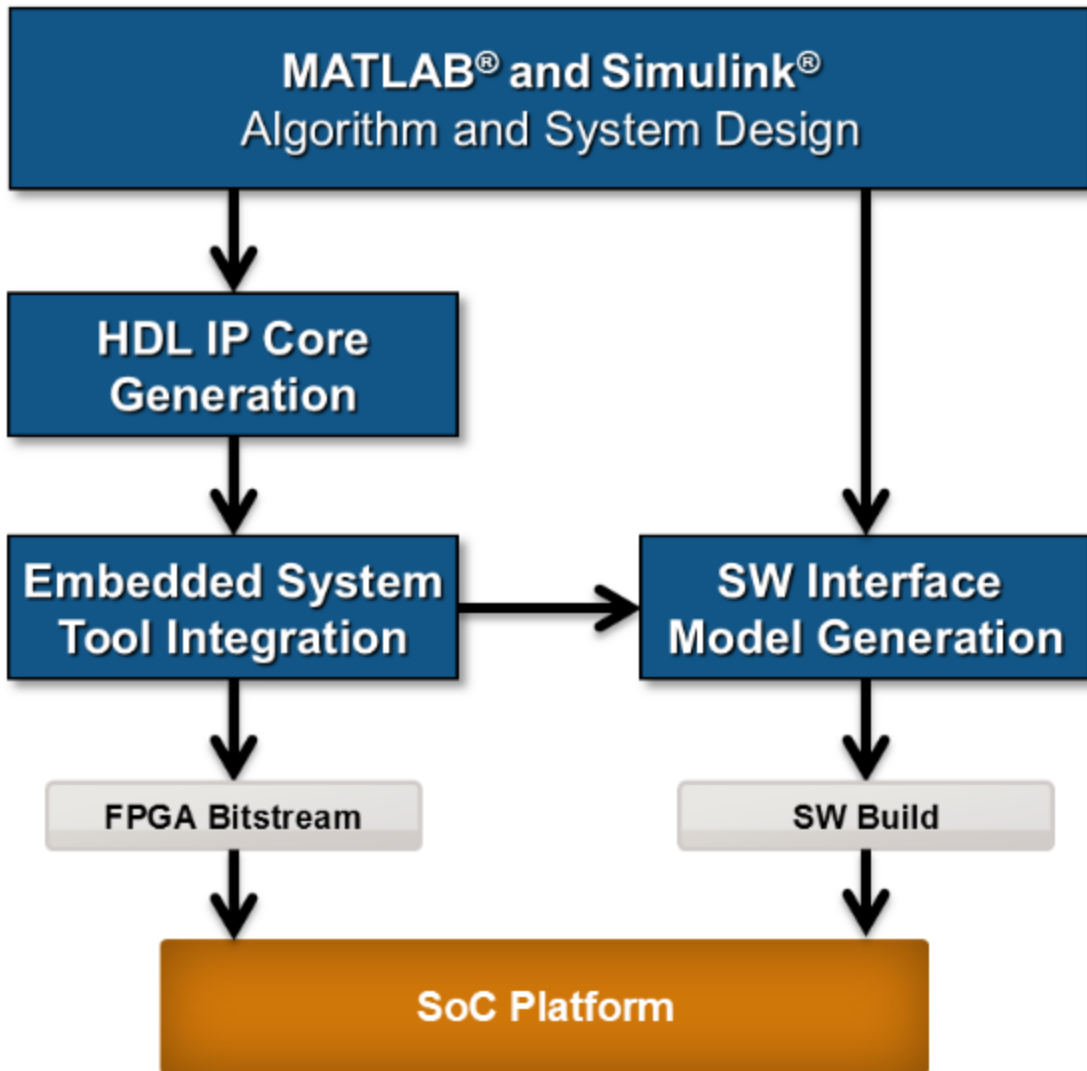
The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor reads FPGA data from the previous sample time, then writes to the FPGA and continues to run without waiting for the FPGA to finish. Each time the FPGA runs, it executes the logic generated for one DUT subsystem sample time.

Target SoC Platforms and Speedgoat Boards

- “Hardware-Software Codesign Workflow for SoC Platforms” on page 26-2
- “Board and Reference Design Registration System” on page 26-11
- “Register a Custom Board” on page 26-15
- “Register a Custom Reference Design” on page 26-18
- “Generate Simulink Real-Time Interface for Speedgoat Boards” on page 26-22

Hardware-Software Codesign Workflow for SoC Platforms

The HDL Coder hardware-software codesign workflow helps automate the deployment of your MATLAB and Simulink design to a Zynq-7000 platform or Altera SoC platform. You can explore the best ways to partition and deploy your design by iterating through the following workflow.

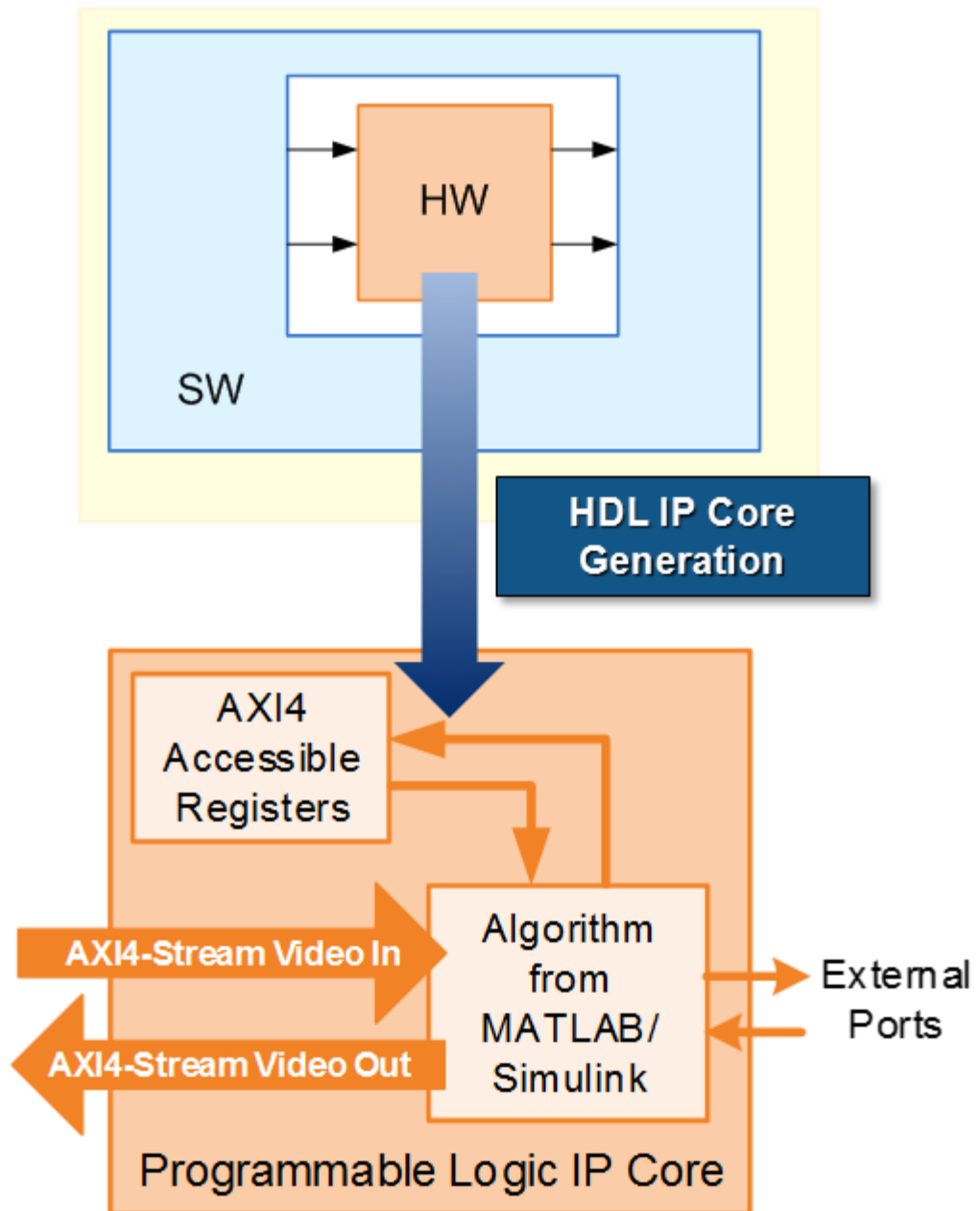


- 1 *MATLAB and Simulink Algorithm and System Design*: You begin by implementing your design in MATLAB or Simulink. When the design behavior meets your requirements, decide how to partition your design: which parts you want to run in hardware, and which parts you want to run in embedded software.

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.

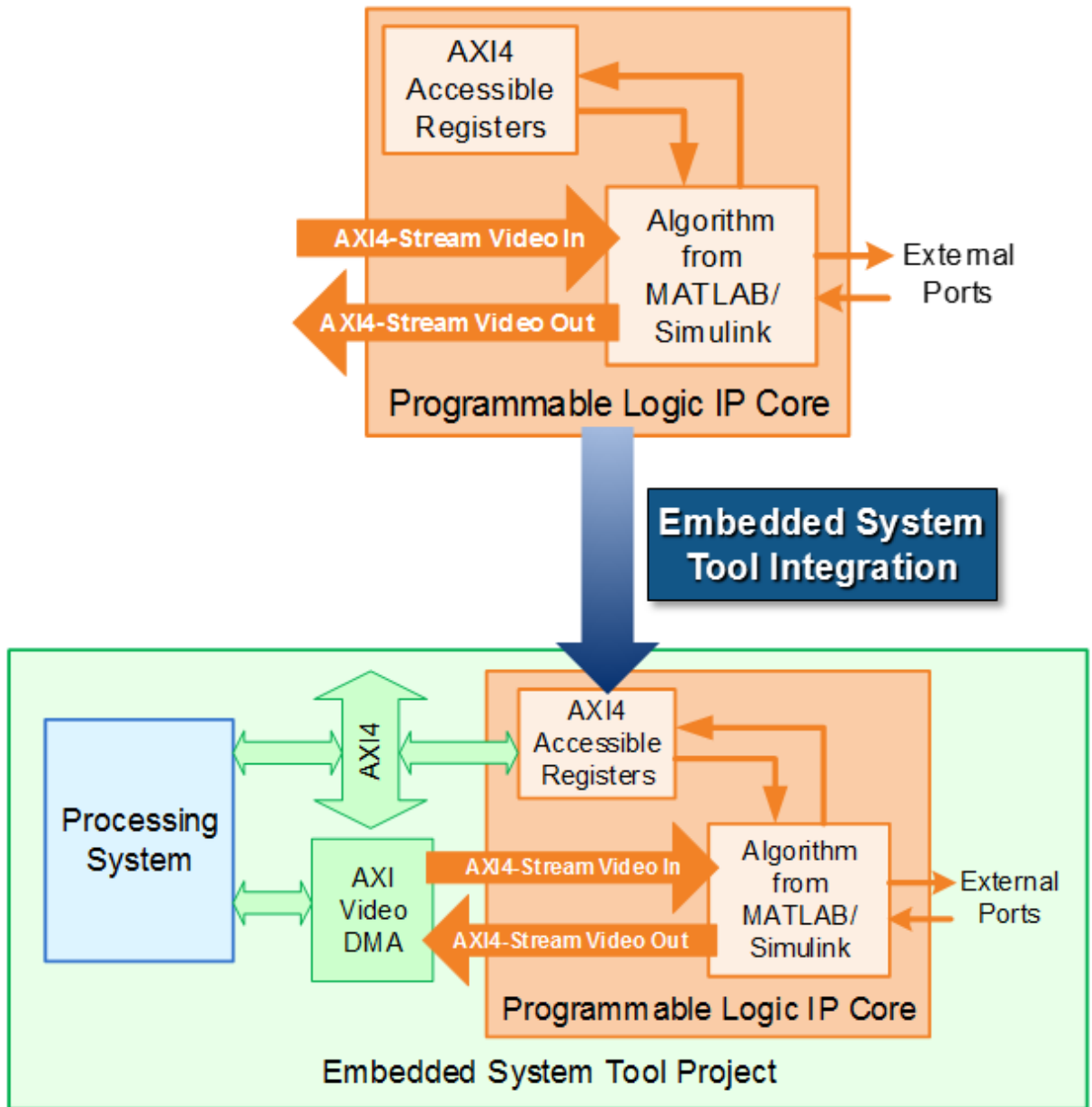
The following diagram shows a design that has been partitioned into a hardware part, in orange, and software part, in blue. HDL IP core generation creates an IP core from the hardware part of the model. The IP core includes hardware interface components such as AXI4 or AXI4-Lite interface-accessible registers, AXI4 or AXI4-Lite interfaces, AXI4-Stream Video 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 the elements the Altera 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 shows the relationship between the reference design, in green, and the generated IP core, in orange.

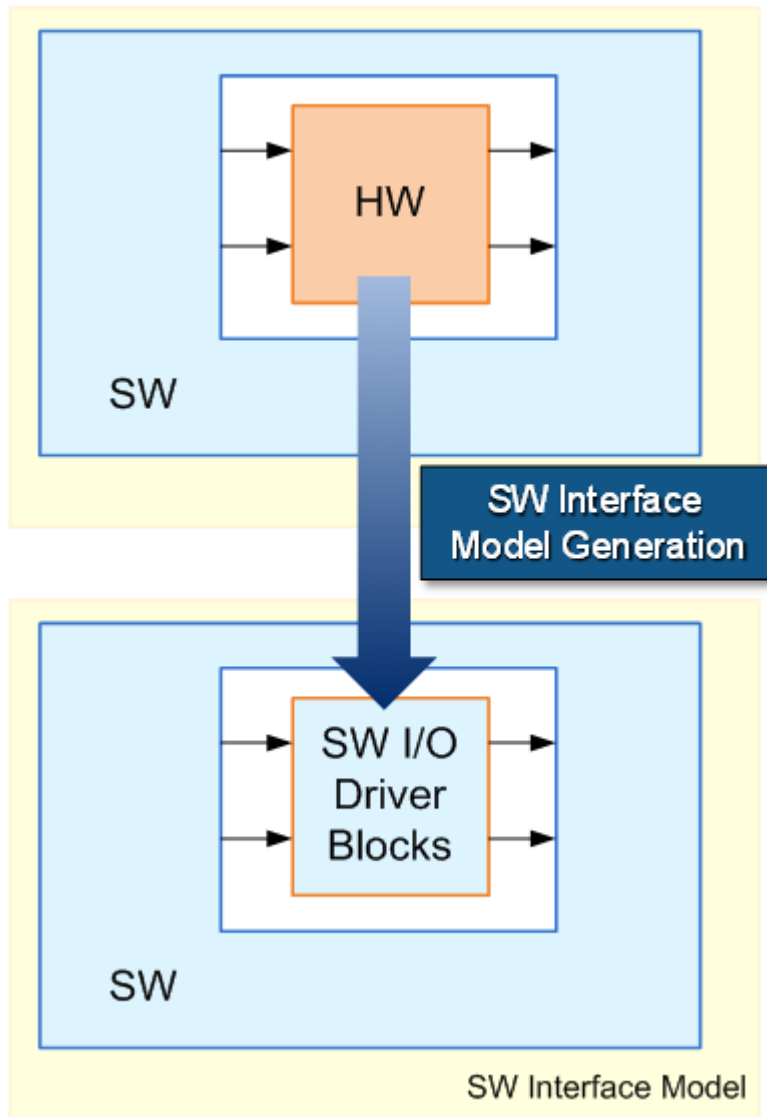


- 4** *SW Interface Model Generation* (requires a Simulink license and Embedded Coder license): In the HDL Workflow Advisor, after you generate the IP core and insert it into the reference design, you can optionally generate a software interface model. The software interface model is your original model 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.

Related Examples

- “Xilinx Zynq Platform”
- “Altera SoC Platform”

Board and Reference Design Registration System

In this section...

“Board, IP Core, and Reference Design Definitions” on page 26-11

“Board Registration Files” on page 26-11

“Reference Design Registration Files” on page 26-12

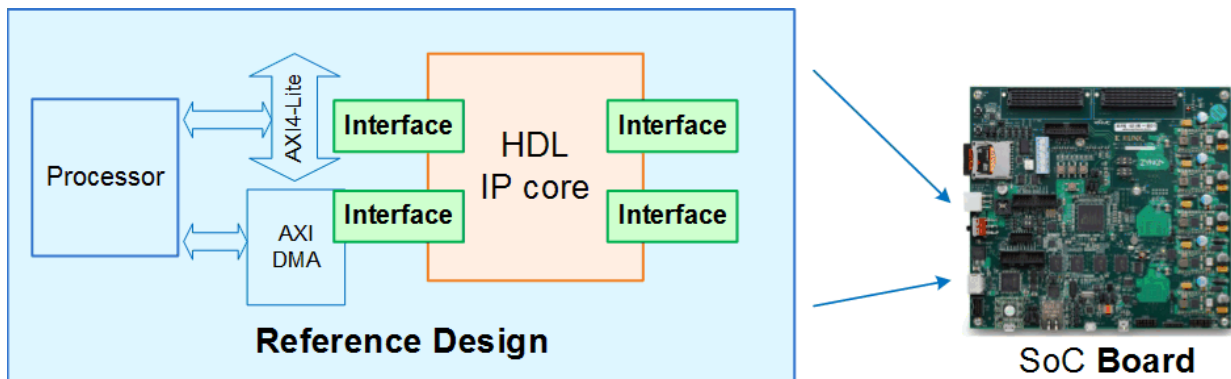
“Predefined Board and Reference Design Examples” on page 26-13

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.

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.

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.

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”

- “Register a Custom Reference Design”
- 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 26-15

“Create a Board Plugin” on page 26-16

“Define a Board Registration Function” on page 26-17

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.createBoard;

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 strings.
- 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 strings.

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”
- Define and Register Custom Board and Reference Design for SoC Workflow

More About

- “Board and Reference Design Registration System”

Register a Custom Reference Design

In this section...

“Define a Reference Design” on page 26-18

“Create a Reference Design Plugin” on page 26-19

“Define a Reference Design Registration Function” on page 26-20

To register a custom reference design, you must:

- 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.

In this section...

“Define a Reference Design” on page 26-18

“Create a Reference Design Plugin” on page 26-19

“Define a Reference Design Registration Function” on page 26-20

Define a Reference Design

Requirements

A reference design definition must be:

- A MATLAB function that returns an `hdlcoder.ReferenceDesign` object.

The reference design definition function can have any name.

- In the reference design plugin folder.

How To Define A Reference Design

- 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 the characteristics of your embedded system design.
- 3 Optionally, to check that the definition is complete, run the `validateReferenceDesign` method.

For example, the following code defines a reference design:

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.createReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Demo system (Vivado 2014.2)';
hRD.BoardName = 'Digilent Zynq ZyBo';

% Tool information
hRD.SupportedToolVersion = {'2014.2'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'design_led.tcl');

hRD.CustomFiles = {'ZYBO_zynq_def.xml'};
%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection', 'clk_wiz_0/clk_out1', ...
    'ResetConnection', 'proc_sys_reset_0/peripheral_aresetn');

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_interconnect_0/M00_AXI', ...
    'BaseAddress', '0x40010000', ...
    'MasterAddressSpace', 'processing_system7_0/Data');
```

Create a Reference Design Plugin

Requirements

A reference design 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 reference design plugin can be a folder named +vivado_base_ref_design.

- Must be on the MATLAB path.

- Must contain the 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.

How 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.
- 2 Save your reference design definition file in the new folder.
- 3 Save any files required by the embedded system design project in the new folder.
- 4 Add the folder to your MATLAB path.

Define a Reference Design Registration Function

Requirements

A reference design registration function:

- Must be named `hdlcoder_ref_design_customization.m`.
- Returns the associated board name, specified as a string, and a list of reference design plugins, specified as a cell array of strings.
- Must be on the MATLAB path.

How 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 string, and a list of reference design plugins, specified as a cell array of strings.

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.Vivado2014_2.plugin_rd', ...
     };

boardName = 'Digilent Zynq ZyBo';
```

end

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Register a Custom Board”
- Define and Register Custom Board and Reference Design for SoC Workflow

More About

- “Board and Reference Design Registration System”

Generate Simulink Real-Time Interface for Speedgoat Boards

In this section...

“Select a Speedgoat Target Device” on page 26-22

“Set the Target Interface for Speedgoat Boards” on page 26-25

“Code Generation, Synthesis, and Generation of Simulink Real-Time Interface Subsystem” on page 26-28

This example shows how to generate a hardware-in-the-loop interface for Speedgoat board programming with Simulink Real-Time using the **Simulink Real-Time FPGA I/O** workflow.

To run this example, you must:

- Have a license for Simulink Real-Time software.
- Use Xilinx ISE 10.1.

Select a Speedgoat Target Device

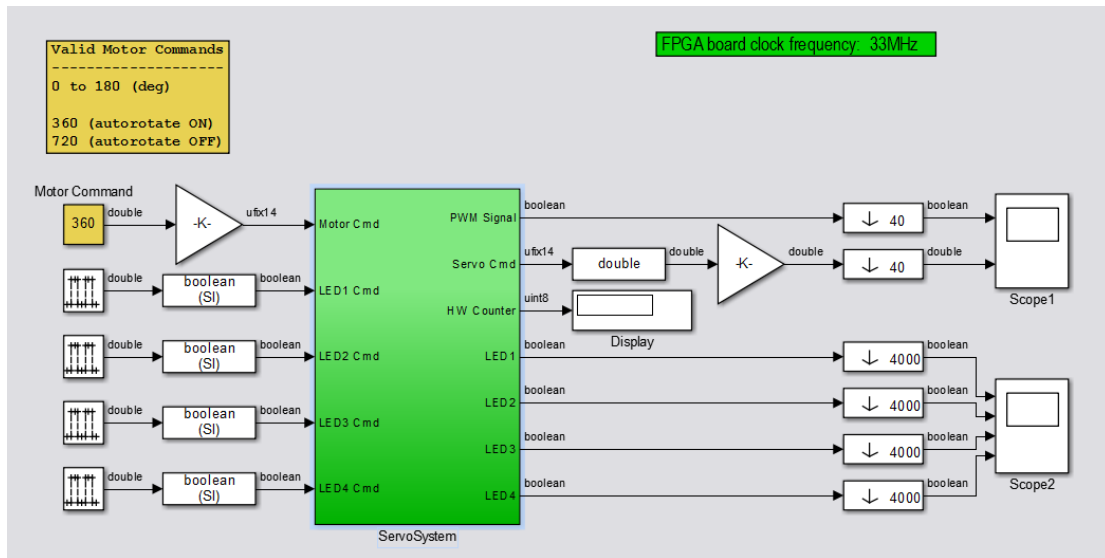
Note: Before selecting a Speedgoat target device, see “Third-Party Synthesis Tools”.

To select a target Speedgoat board:

- 1 Open the model.

`dxpcSGI0301servo_fpga`

The `ServoSystem` subsystem is the device under test (DUT) for HDL code generation.

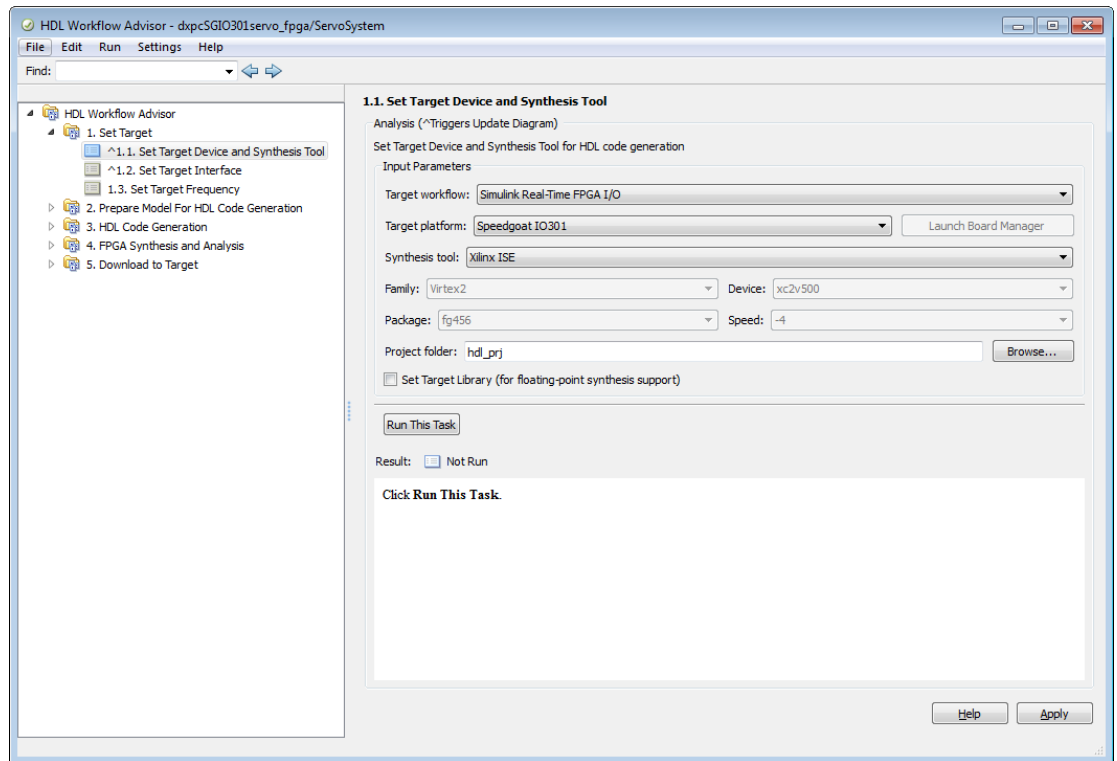


- 2 Right-click the ServoSystem block, and select **HDL Code > HDL Workflow Advisor**.
- 3 In the HDL Workflow Advisor, select **Set Target > Set Target Device and Synthesis Tool**.
- 4 For **Target workflow**, select **Simulink Real-Time FPGA I/O**.

On the left, the **Set Target Interface** and **Set Target Frequency** steps appear under **Set Target** along with the **FPGA Synthesis and Analysis** and **Download to Target** tasks.

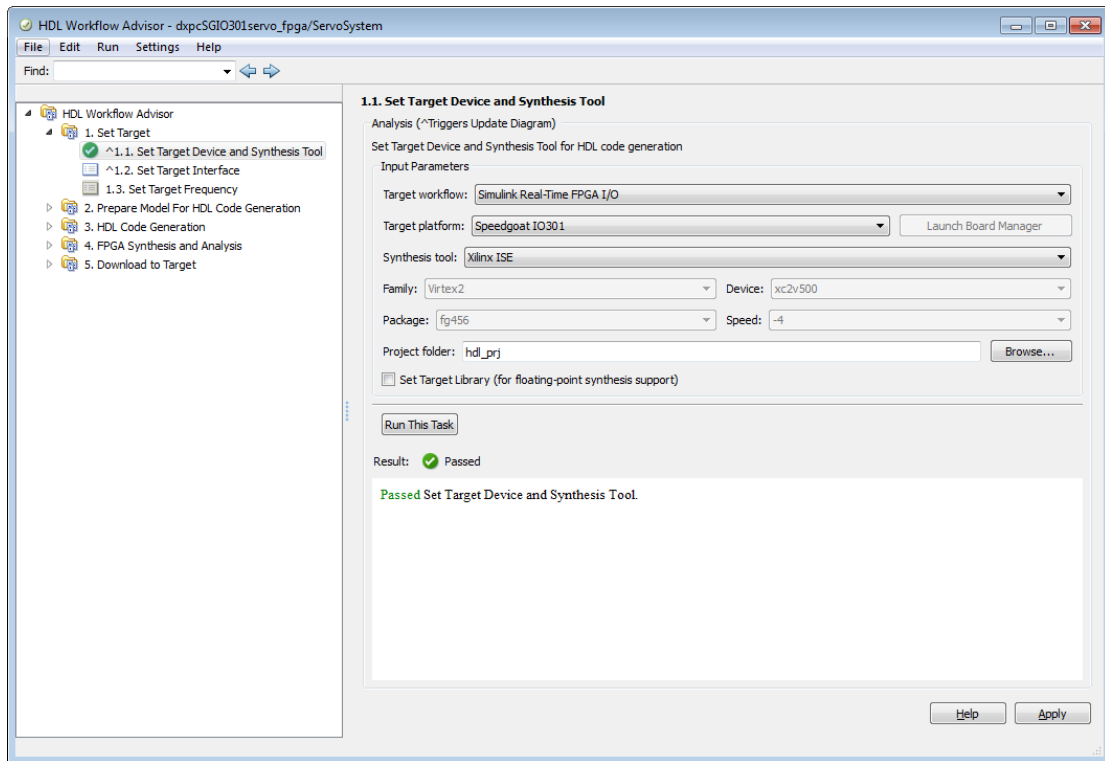
- 5 From the **Target platform** menu, select the Speedgoat IO301 board.

Simulink Real-Time and HDL Workflow Advisor support the same set of Speedgoat devices. For a list of supported boards, see “FPGA Support”.



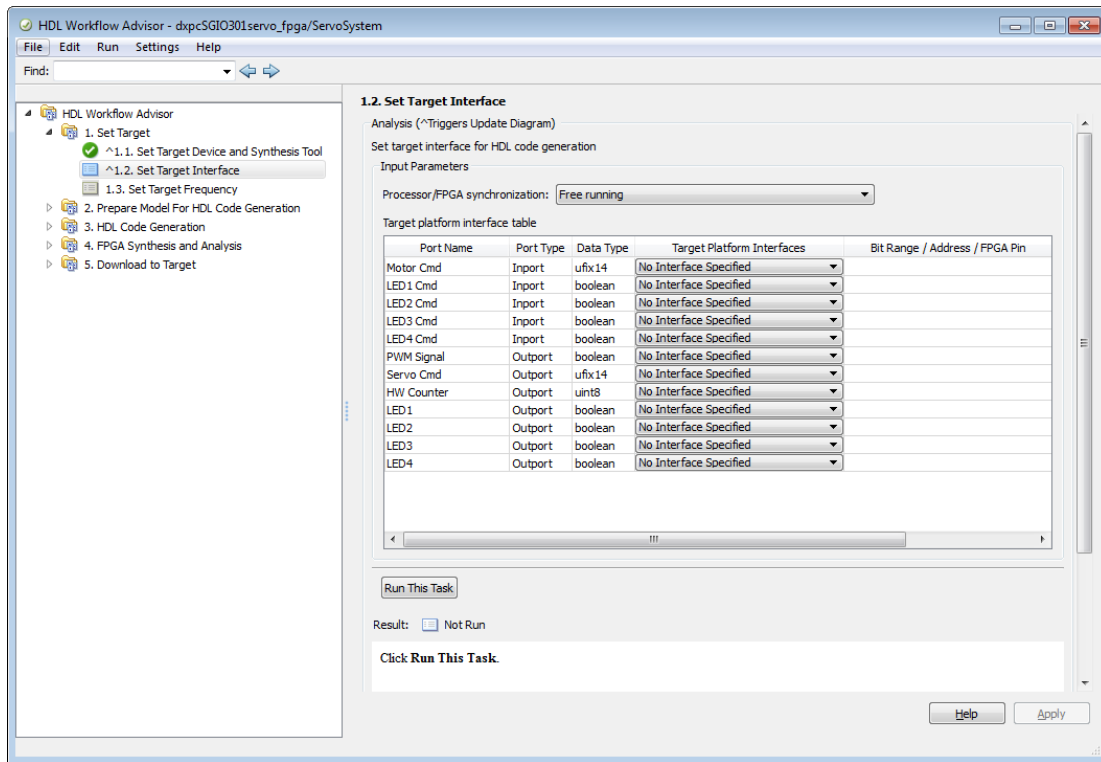
6 Click **Run This Task**.

After the **Set Target Device and Synthesis Tool** task is complete, the HDL Workflow Advisor enables the next task in the hierarchy, **Set Target Interface**. After the **Set Target Device and Synthesis Tool** task runs, the HDL Workflow Advisor looks like this figure.



Set the Target Interface for Speedgoat Boards

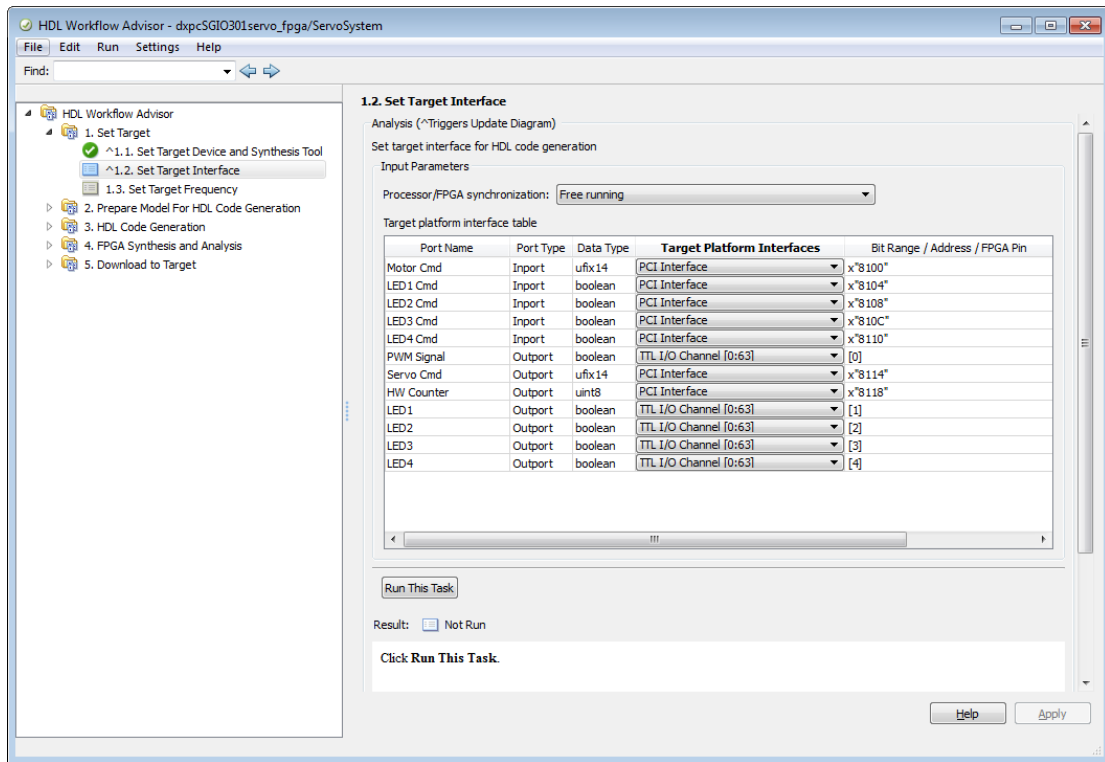
The **Set Target Interface** task in the HDL Workflow Advisor enables you to define how the inputs and outputs of the DUT map to the inputs and outputs of your Speedgoat target device.



Using the **Target Platform Interface** and **Bit Range / Address / FPGA Pin** columns, you can allocate each port on the DUT to an I/O resource on the target device. To allocate ports:

- 1 In the left pane of the HDL Workflow Advisor, select the **Set Target Interface** task.
- 2 In the Target Platform Interface Table, for each port you want to allocate, click the **Target Platform Interfaces** column and select an I/O resource from the dropdown list. Click **Apply**.

This figure shows the Target Platform Interface Table for an example configuration. All ports have been allocated to a PCI Interface address or a single bit on the TTL I/O Connector.



Note: At least one output port must be allocated to the target device. If all ports are left unallocated, the **Set Target Interface** task shows an error and terminates. For information about the I/O resource options, refer to the documentation for your target board.

- 3 Click **Run This Task**.
- 4 In the **Set Target Frequency** task, set your FPGA clock frequency and click **Run This Task**.

Code Generation, Synthesis, and Generation of Simulink Real-Time Interface Subsystem

After selecting the target device and configuring its port interface, you can enable the HDL Workflow Advisor to perform the next sequence of tasks automatically. These tasks include:

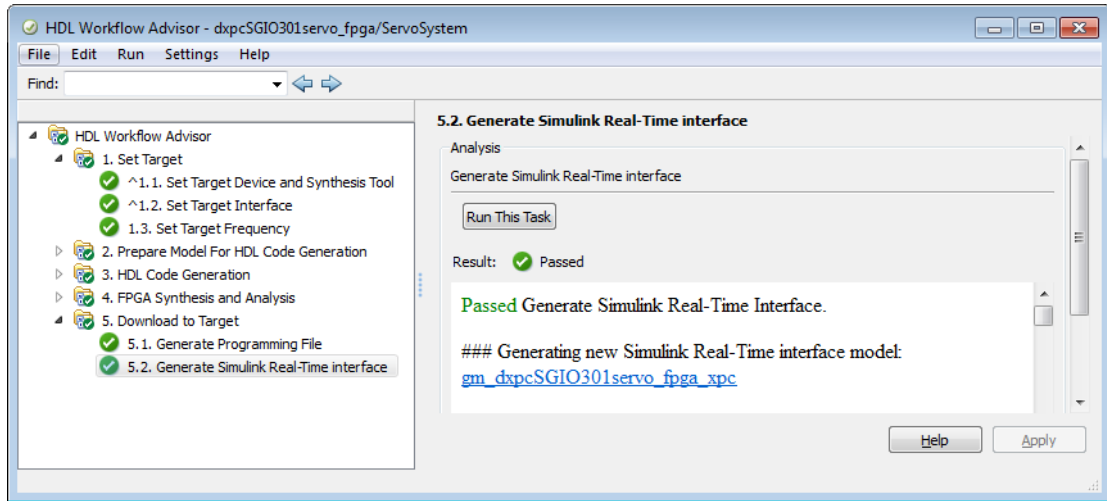
- **Prepare Model For HDL Code Generation:** Checking the model for HDL code generation compatibility.
- **HDL Code Generation:** Setting HDL-related options of the Model Configuration Parameters dialog box and generating HDL code.
- **FPGA Synthesis and Analysis:** Executing synthesis and timing analysis in Xilinx ISE; back annotating the model with critical path information obtained during synthesis.
- **Download to Target :** Generating an FPGA programming file and a model that contains a Simulink Real-Time interface subsystem.

Note: The **Download to Target** tasks do not actually download anything to a target device. They create an interface subsystem that you can plug into a Simulink Real-Time model.

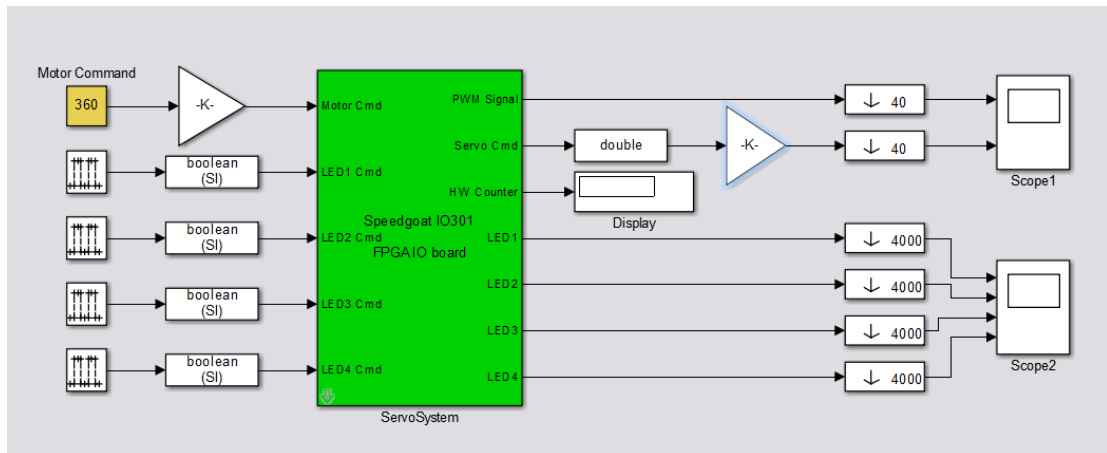
To run this sequence of tasks automatically:

- 1 Open the **Download to Target** task group.
- 2 Right-click **Generate Simulink Real-Time interface** and select **Run to Selected Task**.
- 3 As the **Run to Selected Task** sequence executes, the HDL Workflow Advisor displays a progress indicator for each task.

After the task sequence is complete, you see the **Result** subpane.



- 4 The **Result** pane displays a link to a generated model — `gm_dxpcSGIO301servo_fpga_xpc`. Click the link to open the model.



The model contains the Simulink Real-Time interface subsystem. This new subsystem replaces the DUT (`ServoSystem`) in the original model. It replaces the internals of the original DUT with a Simulink Real-Time FPGA block and other blocks to work with the algorithm on the FPGA.

- 5 Save the `gm_dxpcSGI0301servo_fpga_xpc` model.
- 6 To learn how to use the generated model with Simulink Real-Time, see “FPGA Modeling” in the Simulink Real-Time documentation.

System Objects

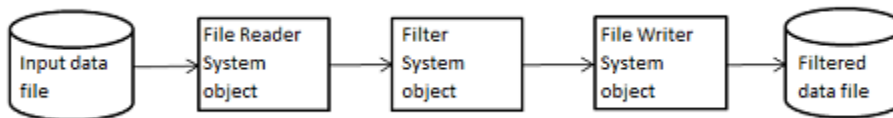
System Objects

- “What Are System Objects?” on page 27-2
- “System Objects vs. MATLAB Functions” on page 27-3
- “System Design and Simulation in MATLAB” on page 27-6
- “System Objects in Simulink” on page 27-7
- “System Object Methods” on page 27-8
- “System Design in MATLAB Using System Objects” on page 27-11

What Are System Objects?

A System object is a specialized kind of MATLAB object. System Toolboxes include System objects and most System Toolboxes also have MATLAB functions and Simulink blocks. System objects are designed specifically for implementing and simulating dynamic systems with inputs that change over time. Many signal processing, communications, and controls systems are dynamic. In a dynamic system, the values of the output signals depend on both the instantaneous values of the input signals and on the past behavior of the system. System objects use internal states to store that past behavior, which is used in the next computational step. As a result, System objects are optimized for iterative computations that process large streams of data, such as video and audio processing systems.

For example, you could use System objects in a system that reads data from a file, filters that data and then writes the filtered output to another file. Typically, a specified amount of data is passed to the filter in each loop iteration. The file reader object uses a state to keep track of where in the file to begin the next data read. Likewise, the file writer object keeps tracks of where it last wrote data to the output file so that data is not overwritten. The filter object maintains its own internal states to assure that the filtering is performed correctly. This diagram represents a single loop of the system.



Many System objects support:

- Fixed-point arithmetic (requires a Fixed-Point Designer license)
- C code generation (requires a MATLAB Coder or Simulink Coder license)
- HDL code generation (requires an HDL Coder license)
- Executable files or shared libraries generation (requires a MATLAB Compiler™ license)

Note: Check your product documentation to confirm fixed-point, code generation, and MATLAB Compiler support for the specific System objects you want to use.

System Objects vs. MATLAB Functions

In this section...

“System Objects vs. MATLAB Functions” on page 27-3

“Process Audio Data Using Only MATLAB Functions Code” on page 27-3

“Process Audio Data Using System Objects” on page 27-4

System Objects vs. MATLAB Functions

Many System objects have MATLAB function counterparts. For simple, one-time computations use MATLAB functions. However, if you need to design and simulate a system with many components, use System objects. Using System objects is also appropriate if your computations require managing internal states, have inputs that change over time or process large streams of data.

Building a dynamic system with different execution phases and internal states using only MATLAB functions would require complex programming. You would need code to initialize the system, validate data, manage internal states, and reset and terminate the system. System objects perform many of these managerial operations automatically during execution. By combining System objects in a program with other MATLAB functions, you can streamline your code and improve efficiency.

Process Audio Data Using Only MATLAB Functions Code

This example shows how to write MATLAB function-only code for reading audio data.

The code reads audio data from a file, filters it, and then plays the filtered audio data. The audio data is read in frames. This code produces the same result as the System objects code in the next example, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Obtain the total number of samples and the sampling rate from the source file.

```
audioInfo = audioinfo(fname);  
maxSamples = audioInfo.TotalSamples;  
fs = audioInfo.SampleRate;
```

Define the filter to use.

```
b = fir1(160, .15);
```

Initialize the filter states.

```
z = zeros(1, numel(b)-1);
```

Define the amount of audio data to process at one time, and initialize the while loop index.

```
frameSize = 1024;  
nIdx = 1;
```

Define the while loop to process the audio data.

```
while nIdx <= maxSamples(1)-frameSize+1  
    audio = audioread(fname, [nIdx nIdx+frameSize-1]);  
    [y,z] = filter(b,1, audio,z);  
    sound(y, fs);  
    nIdx = nIdx+frameSize;  
end
```

The loop uses explicit indexing and state management, which can be a tedious and error-prone approach. You must have detailed knowledge of the states, such as, sizes and data types. Another issue with this MATLAB-only code is that the `sound` function is not designed to run in real time. The resulting audio is very choppy and barely audible.

Process Audio Data Using System Objects

This example shows how to write System objects code for reading audio data.

The code uses System objects from the DSP System Toolbox software to read audio data from a file, filter it, and then play the filtered audio data. This code produces the same result as the MATLAB code shown previously, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Define the System object to read the file.

```
audioIn = dsp.AudioFileReader(fname, 'OutputDataType', 'single');
```

Define the System object to filter the data.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Define the System object to play the filtered audio data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

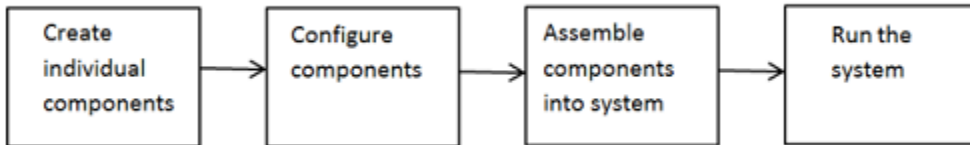
Define the while loop to process the audio data.

```
while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);  % Filter the data
    step(audioOut,y);        % Play the filtered data
end
```

This System objects code avoids the issues present in the MATLAB-only code. Without requiring explicit indexing, the file reader object manages the data frame sizes while the filter manages the states. The audio player object plays each audio frame as it is processed.

System Design and Simulation in MATLAB

System objects allow you to design and simulate your system in MATLAB. You use System objects in MATLAB as shown in this diagram.



- 1** Create individual components — Create the System objects to use in your system. See “Create Components for Your System” on page 27-11 for information. In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See “Define Basic System Objects”.
- 2** Configure components — If necessary, change the objects’ property values to model your particular system. All System object properties have default values that you may be able to use without changing them. See “Configure Components for Your System” on page 27-11 for information.
- 3** Assemble components into system — Write a MATLAB program that includes those System objects, connecting them using MATLAB variables as inputs and outputs to simulate your system. See “Assemble Components to Create Your System” on page 27-13 for information.
- 4** Run the system — Run your program, which uses the `step` method to run your system’s System objects. You can change tunable properties while your system is running. See “Run Your System” on page 27-14 and “Reconfigure Your System During Runtime” on page 27-15 for information.

System Objects in Simulink

In this section...
“System Objects in the MATLAB Function Block” on page 27-7
“System Objects in the MATLAB System Block” on page 27-7

System Objects in the MATLAB Function Block

You can include System object code in Simulink models using the MATLAB Function block. Your function can include one or more System objects. Portions of your system may be easier to implement in the MATLAB environment than directly in Simulink. Many System objects have Simulink block counterparts with equivalent functionality. Before writing MATLAB code to include in a Simulink model, check for existing blocks that perform the desired operation.

System Objects in the MATLAB System Block

You can include individual System objects that you create with a class definition file into Simulink using the MATLAB System block. This provides one way to add your own algorithm blocks into your Simulink models. For information, see “System Object Integration” in the Simulink documentation.

System Object Methods

In this section...
“What Are System Object Methods?” on page 27-8
“The Step Method” on page 27-8
“Common Methods” on page 27-9

What Are System Object Methods?

After you create a System object, you use various object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<method>(<handle>)`, such as `step(H)`, plus possible extra input arguments.

System objects use a minimum of two commands to process data—a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

The Step Method

The `step` method is the key System object method. You use `step` to process data using the algorithm defined by that object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object. For more information about the step method and other available methods, see the descriptions in “Common Methods” on page 27-9.

Common Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
<code>step</code>	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the <code>step</code> method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The <code>step</code> method returns regular MATLAB variables. Example: <code>Y = step(H,X)</code>
<code>release</code>	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. For System objects, use the <code>release</code> method instead of a destructor.
<code>reset</code>	Resets the internal states of a locked object to the initial values for that object and leaves the object locked
<code>getNumInputs</code>	Returns the number of inputs (excluding the object itself) expected by the <code>step</code> method. This number varies for an object depending on whether any properties enable additional inputs.
<code>getNumOutputs</code>	Returns the number of outputs expected from the <code>step</code> method. This number varies for an object depending on whether any properties enable additional outputs.
<code>getDiscreteState</code>	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the <code>step</code> method on it or after you have released the object), the states are empty. If the object has no discrete states, <code>getDiscreteState</code> returns an empty structure.
<code>clone</code>	Creates another object of the same type with the same property values
<code>isLocked</code>	Returns a logical value indicating whether the object is locked.

Method	Description
isDone	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns false .
info	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.

System Design in MATLAB Using System Objects

In this section...

“Create Components for Your System” on page 27-11

“Configure Components for Your System” on page 27-11

“Assemble Components to Create Your System” on page 27-13

“Run Your System” on page 27-14

“Reconfigure Your System During Runtime” on page 27-15

Create Components for Your System

A System object is a component you can use to create your system in MATLAB. System objects support fixed- or variable-size data. *Variable-size data* is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at initialization time, and therefore, cannot change at run time.

Note: If you use a function to create and use a System object, specify the object creation using conditional code. This will prevent errors if that function is called within a loop.

These examples use System objects that are predefined in the software. You can also create your own System objects (see).

For a list of predefined System objects that are supported for HDL code generation, see “Predefined System Objects Supported for HDL Code Generation”. You can also create your own System objects (see “Define Basic System Objects”)

Configure Components for Your System

When to Configure Components

If you did not set an object's properties when you created it and do not want to use default values, you must explicitly set those properties. Some properties allow you to change their values while your system is running. See “Reconfigure Your System During Runtime” on page 27-15 for information.

Most properties are independent of each other. However, some System object properties enable or disable another property or limit the values of another property. To avoid

errors or warnings, you should set the controlling property before setting the dependent property.

Display Component Property Values

To display the current property values for an object, type that object's handle name at the command line (such as `audioIn`). To display the value of a specific property, type `objecthandle.propertyname` (such as `audioIn.FileName`).

Configure Component Property Values

This example shows how to configure the components for your system by setting the component objects' properties.

Use this procedure if you have created your components separately from configuring them. If you have not yet created your components, use the procedure below that describes how to create and configure your components at the same time.

For the file reader object, specify the file to read and set the output data type.

```
audioIn.FileName = 'speech_dft_8kHz.wav';  
audioIn.OutputDataType = 'single';
```

For the filter object, specify the filter numerator coefficients using the `fir1` function, which specifies the lowpass filter order and the cutoff frequency.

```
filtLP.Numerator = fir1(160,.15);
```

For the audio player object, specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut.SampleRate = audioIn.SampleRate;
```

Create and Configure Components at the Same Time

This example shows how to create your System object components and configure the desired properties at the same time. To avoid errors or warnings for dependent properties, you should set the controlling property before setting the dependent property. Use this procedure if you have not already created your components.

Create the file reader object, specify the file to read, and set the output data type.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...  
    'OutputDataType','single')
```

Create the filter object and specify the filter numerator using the `fir1` function. Specify the lowpass filter order and the cutoff frequency of the `fir1` function.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Create the audio player object and specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

After you create the components, you can assemble them in your system.

Assemble Components to Create Your System

- “Connect Inputs and Outputs” on page 27-13
- “Code for the Whole System” on page 27-13

Connect Inputs and Outputs

After you have determined the components you need and have created and configured your System objects, assemble your system. You use the System objects like other MATLAB variables and include them in MATLAB code. You can pass MATLAB variables into and out of System objects.

The main difference between using System objects and using functions is the `step` method. The `step` method is the processing command for each System object and is customized for that specific System object. This method initializes your objects and controls data flow and state management of your system. You typically use `step` within a loop.

You use the output from an object’s `step` method as the input to another object’s `step` method. For some System objects, you can use properties of those objects to change the number of inputs or outputs. To verify that the appropriate number of input and outputs are being used, you can use `getNumInputs` and `getNumOutputs` on any System object. For information on all available System object methods, see “System Object Methods” on page 27-8.

Code for the Whole System

This example shows how to write the full code for reading, filtering, and playing a file of audio data.

You can type this code on the command line or put it into a program file.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...
    'OutputDataType','single');
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);

while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);  % Filter the data
    step(audioOut,y);        % Play the filtered data
end
```

The while loop uses the `isDone` method to read through the entire file. The `step` method is used on each object inside the loop.

Now, you are ready to run your system.

Run Your System

- “How to Run Your System” on page 27-14
- “What You Cannot Change While Your System Is Running” on page 27-14

How to Run Your System

Run your code either by typing directly at the command line or running a file containing your program. When you run the code for your system, the `step` method instructs each object to process data through that object.

What You Cannot Change While Your System Is Running

The first call to the `step` method initializes and then locks your object. When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. Use the `isLocked` method to verify whether an object is locked. When the object is locked, you cannot change:

- Number of inputs or outputs
- Data type of inputs or outputs
- Data type of any tunable property
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data
- Value of any nontunable property

To make changes to your system while it is running, see “Reconfigure Your System During Runtime” on page 27-15.

Reconfigure Your System During Runtime

- “When Can You Change Component Properties?” on page 27-15
- “Change Input Complexity or Dimensions” on page 27-15

When Can You Change Component Properties?

When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. You can use `isLocked` on any System object to verify whether it is locked or not. When processing is complete, you can use the `release` method to unlock a System object.

Some object properties are *tunable*, which enables you to change them even if the object is locked. Unless otherwise specified, System objects properties are nontunable. Refer to the object’s reference page to determine whether an individual property is tunable. Typically, tunable properties are not critical to how the System object processes data.

Change Input Complexity or Dimensions

During simulation, some System objects do not allow complex data if the object was initialized with real data. You cannot change any input complexity during code generation.

You can change the value of a tunable property without a warning or error being produced. For all other changes at run time, an error occurs.

Define New System Objects

- “Define Basic System Objects” on page 28-2
- “Change Number of Step Inputs or Outputs” on page 28-4
- “Validate Property and Input Values” on page 28-8
- “Initialize Properties and Setup One-Time Calculations” on page 28-11
- “Set Property Values at Construction Time” on page 28-13
- “Reset Algorithm State” on page 28-15
- “Define Property Attributes” on page 28-17
- “Hide Inactive Properties” on page 28-21
- “Limit Property Values to Finite String Set” on page 28-23
- “Process Tuned Properties” on page 28-26
- “Release System Object Resources” on page 28-28
- “Define Composite System Objects” on page 28-30
- “Define Finite Source Objects” on page 28-33
- “Save System Object” on page 28-35
- “Load System Object” on page 28-38
- “Clone System Object” on page 28-42
- “Define System Object Information” on page 28-43
- “Specify Locked Input Size” on page 28-45
- “Use Update and Output for Nondirect Feedthrough” on page 28-47
- “Methods Timing” on page 28-50
- “System Object Input Arguments and ~ in Code Examples” on page 28-52
- “What Are Mixin Classes?” on page 28-53
- “Best Practices for Defining System Objects” on page 28-54

Define Basic System Objects

This example shows how to create a basic System object that increments a number by one.

The class definition file contains the minimum elements required to define a System object.

Create the Class Definition File

- 1 Create a MATLAB file named `AddOne.m` to contain the definition of your System object.

```
edit AddOne.m
```

- 2 Subclass your object from `matlab.System`. Insert this line as the first line of your file.

```
classdef AddOne < matlab.System
```

- 3 Add the `stepImpl` method, which contains the algorithm that runs when users call the `step` method on your object. You always set the `stepImpl` method access to `protected` because it is an internal method that users do not directly call or run.

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle.

In this example, instead of passing in the object handle, `~` is used to indicate that the object handle is not used in the function. Using `~` instead of an object handle prevents warnings about unused variables from occurring.

By default, the number of inputs and outputs are both one. To change the number of inputs or outputs, use the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively.

```
methods (Access = protected)
    function y = stepImpl(~,x)
        y = x + 1;
    end
end
```

Note: Instead of manually creating your class definition file, you can use an option on the **New > System Object** menu to open a template. The **Basic** template opens a simple

System object template. The **Advanced** template includes more advanced features of System objects, such as backup and restore. The **Simulink Extension** template includes additional customizations of the System object for use in the Simulink MATLAB System block. You then can edit the template file, using it as guideline, to create your own System object.

Complete Class Definition File for Basic System Object

```
classdef AddOne < matlab.System
% ADDONE Compute an output value one greater than the input value

% All methods occur inside a methods declaration.
% The stepImpl method has protected access
methods (Access = protected)

    function y = stepImpl(~,x)
        y = x + 1;
    end
end
end
```

See Also

matlab.System | stepImpl

Related Examples

- “Change Number of Step Inputs or Outputs” on page 28-4

More About

- “System Design and Simulation in MATLAB”

Change Number of Step Inputs or Outputs

This example shows how to specify two inputs and two outputs for the `step` method.

If you specify the inputs and outputs to the `stepImpl` method, you do not need to specify the `getNumInputsImpl` and `getNumOutputsImpl` methods. If you have a variable number of inputs or outputs (using `varargin` or `varargout`), include the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively, in your class definition file.

Note: You should only use `getNumInputsImpl` or `getNumOutputsImpl` methods to change the number of System object inputs or outputs. Do not use any other handle objects within a System object to change the number of inputs or outputs.

You always set the `getNumInputsImpl` and `getNumOutputsImpl` methods access to `protected` because they are internal methods that users do not directly call or run.

Update the Algorithm for Multiple Inputs and Outputs

Update the `stepImpl` method to specify two inputs and two outputs. You do not need to implement associated `getNumInputsImpl` or `getNumOutputsImpl` methods.

```
methods (Access = protected)
    function [y1,y2] = stepImpl(~,x1,x2)
        y1 = x1 + 1;
        y2 = x2 + 1;
    end
end
```

Update the Algorithm and Associated Methods

Update the `stepImpl` method to use `varargin` and `varargout`. In this case, you must implement the associated `getNumInputsImpl` and `getNumOutputsImpl` methods to specify two or three inputs and outputs.

```
methods (Access = protected)
    function varargout = stepImpl(obj,varargin)
        varargout{1} = varargin{1}+1;
        varargout{2} = varargin{2}+1;
        if (obj.numInputsOutputs == 3)
            varargout{3} = varargin{3}+1;
        end
end
```

```

end

function validatePropertiesImpl(obj)
    if ~((obj.numInputsOutputs == 2) || ...
        (obj.numInputsOutputs == 3))
        error('Only 2 or 3 input and outputs allowed.');
```

```

    end
end

function numIn = getNumInputsImpl(obj)
    numIn = 3;
    if (obj.numInputsOutputs == 2)
        numIn = 2;
    end
end

function numOut = getNumOutputsImpl(obj)
    numOut = 3;
    if (obj.numInputsOutputs == 2)
        numOut = 2;
    end
end
end
end

```

Use this syntax to run the algorithm with two inputs and two outputs.

```

x1 = 3;
x2 = 7;
[y1,y2] = step(AddOne,x1,x2);

```

To change the number of inputs or outputs, you must release the object before rerunning it.

```

release(AddOne)
x1 = 3;
x2 = 7;
x3 = 10
[y1,y2,y3] = step(AddOne,x1,x2,x3);

```

Complete Class Definition File with Multiple Inputs and Outputs

```

classdef AddOne < matlab.System
% ADDONE Compute output values one greater than the input values

    % This property is nontunable and cannot be changed

```

```
% after the setup or step method has been called.
properties (Nontunable)
    numInputsOutputs = 3;    % Default value
end

% All methods occur inside a methods declaration.
% The stepImpl method has protected access
methods (Access = protected)
    function varargout = stepImpl(obj,varargin)
        if (obj.numInputsOutputs == 2)
            varargout{1} = varargin{1}+1;
            varargout{2} = varargin{2}+1;
        else
            varargout{1} = varargin{1}+1;
            varargout{2} = varargin{2}+1;
            varargout{3} = varargin{3}+1;
        end
    end
end

function validatePropertiesImpl(obj)
    if ~(obj.numInputsOutputs == 2) ||...
        (obj.numInputsOutputs == 3)
        error('Only 2 or 3 input and outputs allowed.');
```

```
    end
```

```
end
```

```
function numIn = getNumInputsImpl(obj)
```

```
    numIn = 3;
```

```
    if (obj.numInputsOutputs == 2)
```

```
        numIn = 2;
```

```
    end
```

```
end
```

```
function numOut = getNumOutputsImpl(obj)
```

```
    numOut = 3;
```

```
    if (obj.numInputsOutputs == 2)
```

```
        numOut = 2;
```

```
    end
```

```
end
```

```
end
```

end

Related Examples

- “Validate Property and Input Values” on page 28-8
- “Define Basic System Objects” on page 28-2

More About

- “System Object Input Arguments and ~ in Code Examples” on page 28-52

Validate Property and Input Values

This example shows how to verify that the user's inputs and property values are valid.

Validate Properties

This example shows how to validate the value of a single property using `set.PropertyName` syntax. In this case, the *PropertyName* is `Increment`.

```
methods
    % Validate the properties of the object
    function set.Increment(obj, val)
        if val >= 10
            error('The increment value must be less than 10');
        end
        obj.Increment = val;
    end
end
```

This example shows how to validate the value of two interdependent properties using the `validatePropertiesImpl` method. In this case, the `UseIncrement` property value must be `true` and the `WrapValue` property value must be less than the `Increment` property value.

```
methods (Access = protected)
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue > obj.Increment
            error('Wrap value must be less than increment value');
        end
    end
end
```

Validate Inputs

This example shows how to validate that the first input is a numeric value.

```
methods (Access = protected)
    function validateInputsImpl(~, x)
        if ~isnumeric(x)
            error('Input must be numeric');
        end
    end
end
```



```
end
```

Complete Class Definition File with Property and Input Validation

```
classdef AddOne < matlab.System
% ADDONE Compute an output value by incrementing the input value

% All properties occur inside a properties declaration.
% These properties have public access (the default)
properties (Logical)
    UseIncrement = true
end

properties (PositiveInteger)
    Increment = 1
    WrapValue = 10
end

methods
% Validate the properties of the object
function set.Increment(obj,val)
    if val >= 10
        error('The increment value must be less than 10');
    end
    obj.Increment = val;
end
end

methods (Access = protected)
function validatePropertiesImpl(obj)
    if obj.UseIncrement && obj.WrapValue > obj.Increment
        error('Wrap value must be less than increment value');
    end
end

% Validate the inputs to the object
function validateInputsImpl(~,x)
    if ~isnumeric(x)
        error('Input must be numeric');
    end
end

function out = stepImpl(obj,in)
    if obj.UseIncrement
        out = in + obj.Increment;
    end
end
end
```

```
    else
      out = in + 1;
    end
  end
end
end
end
```

Note: All inputs default to variable-size inputs. See “Change Input Complexity or Dimensions” for more information.

Related Examples

- “Define Basic System Objects” on page 28-2

More About

- “Methods Timing” on page 28-50
- “Property Set Methods”
- “System Object Input Arguments and ~ in Code Examples” on page 28-52

Initialize Properties and Setup One-Time Calculations

This example shows how to write code to initialize and set up a System object.

In this example, you initialize the internal state and count for a counter System object. You do these initialization tasks one time during setup, rather than every time you call the step method.

Define Public Properties to Initialize

In this example, you define the public `Threshold` property and specify the value of that property. Users cannot change *nontunable* properties after the `setup` method has been called.

```
properties (Nontunable)
    Threshold = int32(1)
end
```

Define Private Properties to Initialize

Users cannot access *private* properties directly, but only through methods of the System object. In this example, you define `State` and `Count` as private properties.

```
properties (Access=private)
    State
    Count
end
```

Define Setup

You use the `setupImpl` method to perform setup and initialization tasks. You should include code in the `setupImpl` method that you want to execute one time only. The `setupImpl` method is called once during the first call to the `step` method. In this example, you initialize the counter state and count value.

```
methods (Access=protected)
    function setupImpl(obj, ~)
        % Initialize states
        obj.Count = int32(0);
        obj.State = int32(0);
    end
end
```

Complete Class Definition File with Initialization and Setup

```
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 + u(1); % Increment count
            end
            y = obj.State;          % Delay output
            obj.State = obj.Count; % Put new value in state
        end
    end
end
```

See Also

setupImpl | stepImpl

Related Examples

- “Define Property Attributes” on page 28-17

More About

- “Methods Timing” on page 28-50

Set Property Values at Construction Time

This example shows how to define a System object constructor and allow it to accept name-value property pairs as input.

Set Properties to Use Name-Value Pair Input

Define the System object constructor, which is a method that has the same name as the class (MyFile in this example). Within that method, you use the `setProperties` method to make all public properties available for input when the user constructs the object. `nargin` is a MATLAB function that determines the number of input arguments. `varargin` indicates all of the object's public properties.

```
methods
    function obj = MyFile(varargin)
        setProperties(obj,nargin,varargin{:});
    end
end
```

Complete Class Definition File with Constructor Setup

```
classdef MyFile < matlab.System
% MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        Filename = 'default.bin' % the name of the file to create
        Access = 'wb' % The file access string (write, binary)
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access = private)
        pFileID; % The identifier of the file to open
    end

    methods
        % You call setProperties in the constructor to let
        % a user specify public properties of object as
        % name-value pairs.
        function obj = MyFile(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end
end
```

```
end

methods (Access = protected)
    % In setup allocate any resources, which in this case is
    % opening the file.
    function setupImpl(obj)
        obj.pFileID = fopen(obj.Filename,obj.Access);
        if obj.pFileID < 0
            error('Opening the file failed');
        end
    end
end

% This System object™ writes the input to the file.
function stepImpl(obj,data)
    fwrite(obj.pFileID,data);
end

% Use release to close the file to prevent the
% file handle from being left open.
function releaseImpl(obj)
    fclose(obj.pFileID);
end
end
end
```

See Also

nargin

Related Examples

- “Define Property Attributes” on page 28-17
- “Release System Object Resources” on page 28-28

Reset Algorithm State

This example shows how to reset an object state.

Reset Counter to Zero

pCount is an internal counter property of the System object obj. The user calls the reset method on the locked object, which calls the resetImpl method. In this example, pCount resets to 0.

Note: When resetting an object's state, make sure you reset the size, complexity, and data type correctly.

```
methods (Access = protected)
    function resetImpl(obj)
        obj.pCount = 0;
    end
end
```

Complete Class Definition File with State Reset

```
classdef Counter < matlab.System
% Counter System object™ that increments a counter

    properties (Access = private)
        pCount
    end

    methods (Access = protected)
        % In step, increment the counter and return
        % its value as an output
        function c = stepImpl(obj)
            obj.pCount = obj.pCount + 1;
            c = obj.pCount;
        end

        % Reset the counter to zero.
        function resetImpl(obj)
            obj.pCount = 0;
        end
    end
end
```

end

See “Methods Timing” on page 28-50 for more information.

More About

- “Methods Timing” on page 28-50

Define Property Attributes

This example shows how to specify property attributes.

Property attributes, which add details to a property, provide a layer of control to your properties. In addition to the MATLAB property attributes, System objects can use these three additional attributes—`nontunable`, `logical`, and `positiveInteger`. To specify multiple attributes, separate them with commas.

Specify Property as Nontunable

Use the *nontunable* attribute for a property when the algorithm depends on the value being constant once data processing starts. Defining a property as nontunable may improve the efficiency of your algorithm by removing the need to check for or react to values that change. For code generation, defining a property as nontunable allows the memory associated with that property to be optimized. You should define all properties that affect the number of input or output ports as nontunable.

System object users cannot change nontunable properties after the `setup` or `step` method has been called. In this example, you define the `InitialValue` property, and set its value to 0.

```
properties (Nontunable)
    InitialValue = 0;
end
```

For HDL code generation, only nontunable public properties are supported.

Specify Property as Logical

Logical properties have the value, `true` or `false`. System object users can enter 1 or 0 or any value that can be converted to a logical. The value, however, displays as `true` or `false`. You can use sparse logical values, but they must be scalar values. In this example, the `Increment` property indicates whether to increase the counter. By default, `Increment` is tunable property. The following restrictions apply to a property with the `Logical` attribute,

- Cannot also be `Dependent` or `PositiveInteger`
- Default value must be `true` or `false`. You cannot use 1 or 0 as a default value.

```
properties (Logical)
    Increment = true
end
```

Specify Property as Positive Integer

In this example, the private property `MaxValue` is constrained to accept only real, positive integers. You cannot use sparse values. The following restriction applies to a property with the `PositiveInteger` attribute,

- Cannot also be `Dependent` or `Logical`

```
properties (PositiveInteger)
    MaxValue
end
```

Specify Property as DiscreteState

If your algorithm uses properties that hold state, you can assign those properties the `DiscreteState` attribute. Properties with this attribute display their state values when users call `getDiscreteStateImpl` via the `getDiscreteState` method. The following restrictions apply to a property with the `DiscreteState` attribute,

- Numeric, logical, or `fi` value, but not a scaled double `fi` value
- Does not have any of these attributes: `Nontunable`, `Dependent`, `Abstract`, `Constant`, or `Transient`.
- No default value
- Not publicly settable
- `GetAccess` = `Public` by default
- Value set only using the `setupImpl` method or when the `System` object is locked during `resetImpl` or `stepImpl`

In this example, you define the `Count` property.

```
properties (DiscreteState)
    Count;
end
```

Complete Class Definition File with Property Attributes

```
classdef Counter < matlab.System
% Counter Increment a counter to a maximum value

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
```

```
% The initial value of the counter
InitialValue = 0
end
properties (Nontunable, PositiveInteger)
% The maximum value of the counter
MaxValue = 3
end

properties (Logical)
% Whether to increment the counter
Increment = true
end

properties (DiscreteState)
% Count state variable
Count
end

methods (Access = protected)
% In step, increment the counter and return its value
% as an output

function c = stepImpl(obj)
    if obj.Increment && (obj.Count < obj.MaxValue)
        obj.Count = obj.Count + 1;
    else
        disp(['Max count, ' num2str(obj.MaxValue) ',reached'])
    end
    c = obj.Count;
end

% Setup the Count state variable
function setupImpl(obj)
    obj.Count = 0;
end

% Reset the counter to one.
function resetImpl(obj)
    obj.Count = obj.InitialValue;
end
end
```

end

More About

- “Class Attributes”
- “Property Attributes”
- “Methods Timing” on page 28-50

Hide Inactive Properties

This example shows how to hide the display of a property that is not active for a particular object configuration.

Hide an inactive property

You use the `isInactivePropertyImpl` method to hide a property from displaying. If the `isInactiveProperty` method returns `true` to the property you pass in, then that property does not display.

```
methods (Access = protected)
    function flag = isInactivePropertyImpl(obj,propertyName)
        if strcmp(propertyName,'InitialValue')
            flag = obj.UseRandomInitialValue;
        else
            flag = false;
        end
    end
end
```

Complete Class Definition File with Hidden Inactive Property

```
classdef Counter < matlab.System
    % Counter Increment a counter

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        % Allow the user to set the initial value
        UseRandomInitialValue = true
        InitialValue = 0
    end

    % The private count variable, which is tunable by default
    properties (Access = private)
        pCount
    end

    methods (Access = protected)
        % In step, increment the counter and return its value
        % as an output
        function c = stepImpl(obj)
            obj.pCount = obj.pCount + 1;
        end
    end
end
```

```
    c = obj.pCount;
end

% Reset the counter to either a random value or the initial
% value.
function resetImpl(obj)
    if obj.UseRandomInitialValue
        obj.pCount = rand();
    else
        obj.pCount = obj.InitialValue;
    end
end

% This method controls visibility of the object's properties
function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
        flag = obj.UseRandomInitialValue;
    else
        flag = false;
    end
end
end
end
end
```

Limit Property Values to Finite String Set

This example shows how to limit a property to accept only a finite set of string values.

Specify a Set of Valid String Values

String sets use two related properties. You first specify the user-visible property name and default string value. Then, you specify the associated hidden property by appending “Set” to the property name. You must use a capital “S” in “Set.”

In the “Set” property, you specify the valid string values as a cell array of the `matlab.system.Stringset` class. This example uses `Color` and `ColorSet` as the associated properties.

```
properties
    Color = 'blue'
end

properties (Hidden,Transient)
    ColorSet = matlab.system.StringSet({'red','blue','green'});
end
```

Complete Class Definition File with String Set

```
classdef Whiteboard < matlab.System
% Whiteboard Draw lines on a figure window
%
% This System object™ illustrates the use of StringSets

    properties
        Color = 'blue'
    end

    properties (Hidden,Transient)
        % Let them choose a color
        ColorSet = matlab.system.StringSet({'red','blue','green'});
    end

    methods (Access = protected)
        function stepImpl(obj)
            h = Whiteboard.getWhiteboard();
            plot(h, ...
                randn([2,1]),randn([2,1]), ...
                'Color',obj.Color(1));
        end
    end
end
```

```
end
function releaseImpl(obj)
    cla(Whiteboard.getWhiteboard());
    hold on
end
end

methods (Static)
function a = getWhiteboard()
    h = findobj('tag','whiteboard');
    if isempty(h)
        h = figure('tag','whiteboard');
        hold on
    end
    a = gca;
end
end
end
```

String Set System Object Example

```
%%
% Each call to step draws lines on a whiteboard

%% Construct the System object
hGreenInk = Whiteboard;
hBlueInk = Whiteboard;

% Change the color
% Note: Press tab after typing the first single quote to
% display all enumerated values.
hGreenInk.Color = 'green';
hBlueInk.Color = 'blue';

% Take a few steps
for i=1:3
    hGreenInk.step();
    hBlueInk.step();
end

%% Clear the whiteboard
hBlueInk.release();

%% Display System object used in this example
```



```
type('Whiteboard.m');
```

See Also

`matlab.system.StringSet`

Process Tuned Properties

This example shows how to specify the action to take when a tunable property value changes during simulation.

The `processTunedPropertiesImpl` method is useful for managing actions to prevent duplication. In many cases, changing one of multiple interdependent properties causes an action. With the `processTunedPropertiesImpl` method, you can control when that action is taken so it is not repeated unnecessarily.

Control When a Lookup Table Is Generated

This example of `processTunedPropertiesImpl` causes the `pLookupTable` to be regenerated when either the `NumNotes` or `MiddleC` property changes.

```
methods (Access = protected)
    function processTunedPropertiesImpl(obj)
        propChange = isChangedProperty(obj,obj.NumNotes)||...
                    isChangedProperty(obj,obj.MiddleC)
        if propChange
            obj.pLookupTable = obj.MiddleC *...
                (1+log(1:obj.NumNotes)/log(12));
        end
    endend
```

Complete Class Definition File with Tuned Property Processing

```
classdef TuningFork < matlab.System
    % TuningFork Illustrate the processing of tuned parameters
    %

    properties
        MiddleC = 440
        NumNotes = 12
    end

    properties (Access = private)
        pLookupTable
    end

    methods (Access = protected)
        function resetImpl(obj)
            obj.MiddleC = 440;
            obj.pLookupTable = obj.MiddleC * ...

```

```
        (1+log(1:obj.NumNotes)/log(12));
end

function hz = stepImpl(obj,noteShift)
    % A noteShift value of 1 corresponds to obj.MiddleC
    hz = obj.pLookupTable(noteShift);
end

function processTunedPropertiesImpl(obj)
    propChange = isChangedProperty(obj,obj.NumNotes)||...
                isChangedProperty(obj,obj.MiddleC)
    if propChange
        obj.pLookupTable = obj.MiddleC *...
            (1+log(1:obj.NumNotes)/log(12));
    end
end
end
```

Release System Object Resources

This example shows how to release resources allocated and used by the System object. These resources include allocated memory, files used for reading or writing, etc.

Release Memory by Clearing the Object

This method allows you to clear the axes on the Whiteboard figure window while keeping the figure open.

```
methods
    function releaseImpl(obj)
        cla(Whiteboard.getWhiteboard());
        hold on
    end
end
```

Complete Class Definition File with Released Resources

```
classdef Whiteboard < matlab.System
% Whiteboard Draw lines on a figure window
%
% This System object™ shows the use of StringSets
%
    properties
        Color = 'blue'
    end

    properties (Hidden)
        % Let user choose a color
        ColorSet = matlab.system.StringSet({'red','blue','green'});
    end

    methods (Access = protected)
        function stepImpl(obj)
            h = Whiteboard.getWhiteboard();
            plot(h, ...
                randn([2,1]), randn([2,1]), ...
                'Color',obj.Color(1));
        end

        function releaseImpl(obj)
            cla(Whiteboard.getWhiteboard());
            hold on
        end
    end
end
```

```
        end
    end

    methods (Static)
        function a = getWhiteboard()
            h = findobj('tag','whiteboard');
            if isempty(h)
                h = figure('tag','whiteboard');
                hold on
            end
            a = gca;
        end
    end
end
```

Related Examples

- “Initialize Properties and Setup One-Time Calculations” on page 28-11

Define Composite System Objects

This example shows how to define System objects that include other System objects.

This example defines a filter System object from an FIR System object and an IIR System object.

Store System Objects in Properties

To define a System object from other System objects, store those objects in your class definition file as properties. In this example, FIR and IIR are separate System objects defined in their own class-definition files. You use those two objects to calculate the `pFir` and `pIir` property values.

```
properties (Nontunable, Access = private)
    pFir % Store the FIR filter
    pIir % Store the IIR filter
end

methods
    function obj = Filter(varargin)
        setProperties(obj,nargin,varargin{:});
        obj.pFir = FIR(obj.zero);
        obj.pIir = IIR(obj.pole);
    end
end
```

Complete Class Definition File of Composite System Object

```
classdef Filter < matlab.System
% Filter System object with a single pole and a single zero
%
% This System object illustrates composition by
% composing an instance of itself.
%

    properties (Nontunable)
        zero = 0.01
        pole = 0.5
    end

    properties (Nontunable,Access = private)
        pZero % Store the FIR filter
        pPole % Store the IIR filter
    end
end
```

```

end

methods
    function obj = Filter(varargin)
        setProperties(obj,nargin,varargin{:});
        % Create instances of FIR and IIR as
        % private properties
        obj.pZero = Zero(obj.zero);
        obj.pPole = Pole(obj.pole);
    end
end

methods (Access = protected)
    function setupImpl(obj,x)
        setup(obj.pZero,x);
        setup(obj.pPole,x);
    end

    function resetImpl(obj)
        reset(obj.pZero);
        reset(obj.pPole);
    end

    function y = stepImpl(obj,x)
        y = step(obj.pZero,x) + step(obj.pPole,x);
    end
end
end

```

Class Definition File for IIR Component of Filter

```

classdef Pole < matlab.System

    properties (Nontunable)
        Den = 1
    end

    properties (Access = private)
        tap = 0
    end

    methods
        function obj = Pole(varargin)
            setProperties(obj,nargin,varargin{:}, 'Den');
        end
    end
end

```

```
end

methods (Access = protected)
function y = stepImpl(obj,x)
    y = x + obj.tap * obj.Den;
    obj.tap = y;
end
end

end
```

Class Definition File for FIR Component of Filter

```
classdef Zero < matlab.System

    properties (Nontunable)
        Num = 1
    end

    properties (Access = private)
        tap = 0
    end

    methods
        function obj = Zero(varargin)
            setProperties(obj,nargin,varargin{:},'Num');
        end
    end

    methods (Access = protected)
        function y = stepImpl(obj,x)
            y = x + obj.tap * obj.Num;
            obj.tap = x;
        end
    end

end
```

See Also

nargin

Define Finite Source Objects

This example shows how to define a System object that performs a specific number of steps or specific number of reads from a file.

Use the FiniteSource Class and Specify End of the Source

- 1 Subclass from finite source class.

```
classdef RunTwice < matlab.System & ...
    matlab.system.mixin.FiniteSource
```

- 2 Specify the end of the source with the `isDoneImpl` method. In this example, the source has two iterations.

```
methods (Access = protected)
    function bDone = isDoneImpl(obj)
        bDone = obj.NumSteps==2
    end
```

Complete Class Definition File with Finite Source

```
classdef RunTwice < matlab.System & ...
    matlab.system.mixin.FiniteSource
    % RunTwice System object that runs exactly two times
    %
    properties (Access = private)
        NumSteps
    end

    methods (Access = protected)
        function resetImpl(obj)
            obj.NumSteps = 0;
        end

        function y = stepImpl(obj)
            if ~obj.isDone()
                obj.NumSteps = obj.NumSteps + 1;
                y = obj.NumSteps;
            else
                y = 0;
            end
        end

        function bDone = isDoneImpl(obj)
```

```
        bDone = obj.NumSteps==2;
    end
end
end
```

More About

- “What Are Mixin Classes?” on page 28-53
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 28-52

Save System Object

This example shows how to save a System object.

Save System Object and Child Object

Define a `saveObjectImpl` method to specify that more than just public properties should be saved when the user saves a System object. Within this method, use the default `saveObjectImpl@matlab.System` to save public properties to the struct, `s`. Use the `saveObject` method to save child objects. Save protected and dependent properties, and finally, if the object is locked, save the object's state.

```
methods (Access = protected)
    function s = saveObjectImpl(obj)
        s = saveObjectImpl@matlab.System(obj);
        s.child = matlab.System.saveObject(obj.child);
        s.protectedprop = obj.protectedprop;
        s.pdependentprop = obj.pdependentprop;
        if isLocked(obj)
            s.state = obj.state;
        end
    end
end
```

Complete Class Definition Files with Save and Load

The `Counter` class definition file sets up an object with a count property. This counter is used in the `MySaveLoader` class definition file to count the number of child objects.

```
classdef Counter < matlab.System
    properties(DiscreteState)
        Count
    end
    methods (Access=protected)
        function setupImpl(obj, ~)
            obj.Count = 0;
        end
        function y = stepImpl(obj, u)
            if u > 0
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
    end
end
```

```
end

classdef MySaveLoader < matlab.System

    properties (Access = private)
        child
        pdependentprop = 1
    end

    properties (Access = protected)
        protectedprop = rand;
    end

    properties (DiscreteState = true)
        state
    end

    properties (Dependent)
        dependentprop
    end

    methods
        function obj = MySaveLoader(varargin)
            obj@matlab.System();
            setProperties(obj,nargin,varargin{:});
        end

        function set.dependentprop(obj, value)
            obj.pdependentprop = min(value, 5);
        end

        function value = get.dependentprop(obj)
            value = obj.pdependentprop;
        end
    end

    methods (Access = protected)
        function setupImpl(obj)
            obj.state = 42;
            obj.child = Counter;
        end
        function out = stepImpl(obj,in)
            obj.state = in + obj.state + obj.protectedprop + obj.pdependentprop;
            out = step(obj.child, obj.state);
        end
    end
end
```

```
end

% Serialization
methods (Access = protected)
function s = saveObjectImpl(obj)
    % Call the base class method
    s = saveObjectImpl@matlab.System(obj);

    % Save the child System objects
    s.child = matlab.System.saveObject(obj.child);

    % Save the protected & private properties
    s.protectedprop = obj.protectedprop;
    s.pdependentprop = obj.pdependentprop;

    % Save the state only if object locked
    if isLocked(obj)
        s.state = obj.state;
    end
end

function loadObjectImpl(obj,s,wasLocked)
    % Load child System objects
    obj.child = matlab.System.loadObject(s.child);

    % Load protected and private properties
    obj.protectedprop = s.protectedprop;
    obj.pdependentprop = s.pdependentprop;

    % Load the state only if object locked
    if wasLocked
        obj.state = s.state;
    end

    % Call base class method to load public properties
    loadObjectImpl@matlab.System(obj,s,wasLocked);
end
end
end
```

Related Examples

- “Load System Object” on page 28-38

Load System Object

This example shows how to load and save a System object.

Load System Object and Child Object

Define a `loadObjectImpl` method to load a previously saved System object. Within this method, use the `matlab.System.loadObject` to load the child System object, load protected and private properties, load the state if the object is locked, and use `loadObjectImpl` from the base class to load public properties.

```
methods (Access = protected)
    function loadObjectImpl(obj,s,wasLocked)
        obj.child = matlab.System.loadObject(s.child);

        obj.protectedprop = s.protectedprop;
        obj.pdependentprop = s.pdependentprop;

        if wasLocked
            obj.state = s.state;
        end

        loadObjectImpl@matlab.System(obj,s,wasLocked);
    end
end
```

Complete Class Definition Files with Save and Load

The Counter class definition file sets up an object with a count property. This counter is used in the MySaveLoader class definition file to count the number of child objects.

```
classdef Counter < matlab.System
    properties (DiscreteState)
        Count
    end
    methods (Access=protected)
        function setupImpl(obj, ~)
            obj.Count = 0;
        end
        function y = stepImpl(obj, u)
            if u > 0
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
    end
end
```

```
    end
  end
end

classdef MySaveLoader < matlab.System

  properties (Access = private)
    child
    pdependentprop = 1
  end

  properties (Access = protected)
    protectedprop = rand;
  end

  properties (DiscreteState = true)
    state
  end

  properties (Dependent)
    dependentprop
  end

  methods
    function obj = MySaveLoader(varargin)
      obj@matlab.System();
      setProperties(obj,nargin,varargin{:});
    end

    function set.dependentprop(obj, value)
      obj.pdependentprop = min(value, 5);
    end

    function value = get.dependentprop(obj)
      value = obj.pdependentprop;
    end
  end

  methods (Access = protected)
    function setupImpl(obj)
      obj.state = 42;
      obj.child = Counter;
    end
    function out = stepImpl(obj,in)
      obj.state = in + obj.state + obj.protectedprop + obj.pdependentprop;
    end
  end
end
```

```
        out = step(obj.child, obj.state);
    end
end

% Serialization
methods (Access = protected)
function s = saveObjectImpl(obj)
    % Call the base class method
    s = saveObjectImpl@matlab.System(obj);

    % Save the child System objects
    s.child = matlab.System.saveObject(obj.child);

    % Save the protected & private properties
    s.protectedprop = obj.protectedprop;
    s.pdependentprop = obj.pdependentprop;

    % Save the state only if object locked
    if isLocked(obj)
        s.state = obj.state;
    end
end

function loadObjectImpl(obj,s,wasLocked)
    % Load child System objects
    obj.child = matlab.System.loadObject(s.child);

    % Load protected and private properties
    obj.protectedprop = s.protectedprop;
    obj.pdependentprop = s.pdependentprop;

    % Load the state only if object locked
    if wasLocked
        obj.state = s.state;
    end

    % Call base class method to load public properties
    loadObjectImpl@matlab.System(obj,s,wasLocked);
end
```


end

Related Examples

- “Save System Object” on page 28-35

Clone System Object

This example shows how to clone a System object.

Clone System Object

You can define your own clone method, which is useful for copying objects without saving their state. The default `cloneImpl` method copies both a System object™ and its current state. If an object is locked, the default `cloneImpl` creates a cloned object that is also locked. An example of when you may want to write your own clone method is for cloning objects that handle resources. These objects cannot allocate resources twice and you would not want to save their states. To write your clone method, use the `saveObject` and `loadObject` methods to perform the clone within the `cloneImpl` method.

```
methods (Access = protected)
    function obj2 = cloneImpl(obj1)
        s = saveObject (obj1);
        obj2 = loadObject(s);
    end
end
```

Complete Class Definition File with Clone

```
classdef PassThrough < matlab.System
    methods (Access = protected)
        function y = stepImpl(~,u)
            y = u;
        end
        function obj2 = cloneImpl(obj1)
            s = matlab.System.saveObject(obj1);
            obj2 = matlab.System.loadObject(s);
        end
    end
end
```

Define System Object Information

This example shows how to define information to display for a System object.

Define System Object Info

You can define your own `info` method to display specific information for your System object. The default `infoImpl` method returns an empty struct. This `infoImpl` method returns detailed information when the `info` method is called using `info(x, 'details')` or only count information if it is called using `info(x)`.

```
methods (Access = protected)
    function s = infoImpl(obj,varargin)
        if nargin>1 && strcmp('details',varargin(1))
            s = struct('Name','Counter',...
                'Properties', struct('CurrentCount', ...
                    obj.pCount,'Threshold',obj.Threshold));
        else
            s = struct('Count',obj.pCount);
        end
    end
end
```

Complete Class Definition File with InfoImpl

```
classdef Counter < matlab.System
    % Counter Count values above a threshold

    properties
        Threshold = 1
    end

    properties (DiscreteState)
        Count
    end

    methods (Access = protected)
        function setupImpl(obj)
            obj.Count = 0;
        end

        function resetImpl(obj)
            obj.Count = 0;
        end
    end
end
```

```
function y = stepImpl(obj,u)
    if (u > obj.Threshold)
        obj.Count = obj.Count + 1;
    end
    y = obj.Count;
end

function s = infoImpl(obj,varargin)
    if nargin>1 && strcmp('details',varargin(1))
        s = struct('Name','Counter',...
            'Properties', struct('CurrentCount', ...
                obj.pCount, 'Threshold',obj.Threshold));
    else
        s = struct('Count',obj.pCount);
    end
end
end
end
```

Specify Locked Input Size

This example shows how to specify whether the size of a System object input is locked. The size of a locked input cannot change until the System object is unlocked. Use the `step` method and run the object to lock it. Use `release` to unlock the object.

Use the `isInputSizeLockedImpl` method to specify that the input size is locked.

```
methods (Access = protected)
    function flag = isInputSizeLockedImpl(~,~)
        flag = true;
    end
end
```

View the method in the complete class definition file.

```
classdef Counter < matlab.System
    %Counter Count values above a threshold

    properties
        Threshold = 1
    end

    properties (DiscreteState)
        Count
    end

    methods
        function obj = Counter(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end

    methods (Access=protected)
        function resetImpl(obj)
            obj.Count = 0;
        end

        function y = stepImpl(obj, u1)
            if (any(u1 >= obj.Threshold))
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
    end
end
```

```
function flag = isInputSizeLockedImpl(~,~)
    flag = true;
end
end
end
```

Use Update and Output for Nondirect Feedthrough

This example shows how to implement nondirect feedthrough for a System object using the `updateImpl`, `outputImpl` and `isInputDirectFeedthroughImpl` methods. In nondirect feedthrough, the object's outputs depend only on the internal states and properties of the object, rather than the input at that instant in time. You use these methods to separate the output calculation from the state updates of a System object. This enables you to use that object in a feedback loop and prevent algebraic loops.

Subclass from the Nondirect Mixin Class

To use the `updateImpl`, `outputImpl`, and `isInputDirectFeedthroughImpl` methods, you must subclass from both the `matlab.System` base class and the `Nondirect` mixin class.

```
classdef IntegerDelaySysObj < matlab.System & ...
    matlab.system.mixin.Nondirect
```

Implement Updates to the Object

Implement an `updateImpl` method to update the object with previous inputs.

```
methods (Access = protected)
    function updateImpl(obj,u)
        obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
    end
end
```

Implement Outputs from Object

Implement an `outputImpl` method to output the previous, not the current input.

```
methods (Access = protected)
    function [y] = outputImpl(obj,~)
        y = obj.PreviousInput(end);
    end
end
```

Implement Whether Input Is Direct Feedthrough

Implement an `isInputDirectFeedthroughImpl` method to indicate that the input is nondirect feedthrough.

```
methods (Access = protected)
```

```
function flag = isInputDirectFeedthroughImpl(~,~)
    flag = false;
end
end
```

Complete Class Definition File with Update and Output

```
classdef intDelaySysObj < matlab.System &...
    matlab.system.mixin.Nondirect &...
    matlab.system.mixin.CustomIcon
    % intDelaySysObj Delay input by specified number of samples.

    properties
        InitialOutput = 0;
    end
    properties (Nontunable)
        NumDelays = 1;
    end
    properties (DiscreteState)
        PreviousInput;
    end

    methods (Access = protected)
        function validatePropertiesImpl(obj)
            if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
                error('Number of delays must be positive non-zero scalar value.');
            end
            if (numel(obj.InitialOutput)>1)
                error('Initial Output must be scalar value.');
            end
        end

        function setupImpl(obj)
            obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
        end

        function resetImpl(obj)
            obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
        end

        function [y] = outputImpl(obj,~)
            y = obj.PreviousInput(end);
        end
        function updateImpl(obj, u)
            obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
        end
    end
end
```



```
    end
    function flag = isInputDirectFeedthroughImpl(~,~)
        flag = false;
    end
end
end
```

See Also

`matlab.system.mixin.Nondirect` | `outputImpl` | `updateImpl`

More About

- “What Are Mixin Classes?” on page 28-53
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 28-52

Methods Timing

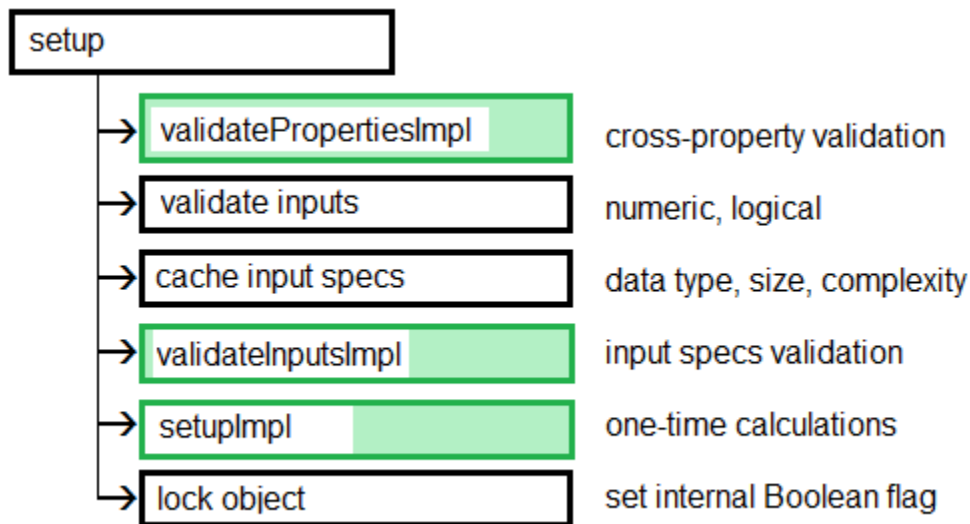
In this section...

“Setup Method Call Sequence” on page 28-50

“Step Method Call Sequence” on page 28-50

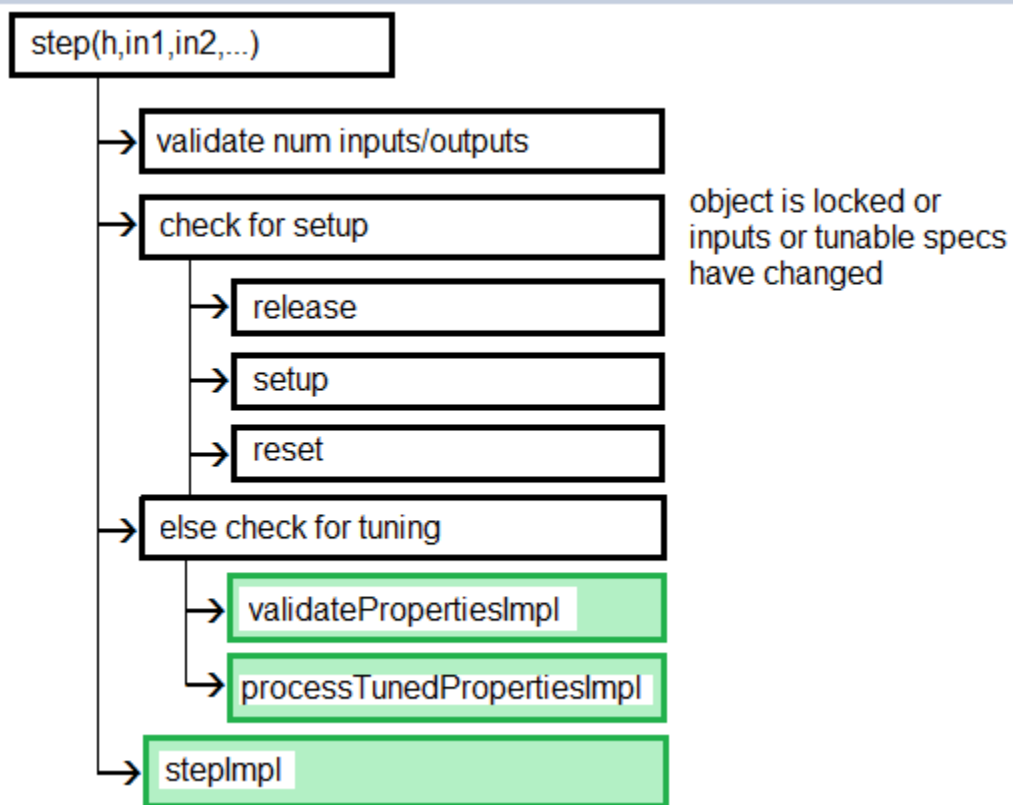
Setup Method Call Sequence

This hierarchy shows the actions performed when you call the `setup` method.



Step Method Call Sequence

This hierarchy shows the actions performed when you call the `step` method.



See Also

`setupImpl` | `stepImpl`

Related Examples

- “Define Basic System Objects” on page 28-2

More About

- “What Are System Object Methods?”
- “The Step Method”

System Object Input Arguments and ~ in Code Examples

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle. In many examples, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables.

What Are Mixin Classes?

Mixin classes are partial classes that you can combine in various combinations to form desired behaviors using multiple inheritance. System objects are composed of a base class, `matlab.System` and may include one or more mixin classes. You specify the base class and mixin classes on the first line of your class definition file.

The following mixin classes are available for use with System objects.

- `matlab.system.mixin.CustomIcon` — Defines a block icon for System objects in the MATLAB System block
- `matlab.system.mixin.FiniteSource` — Adds the `isDone` method to System objects that are sources
- `matlab.system.mixin.Nondirect` — Allows the System object, when used in the MATLAB System block, to support nondirect feedthrough by making the runtime callback functions, `output` and `update` available
- `matlab.system.mixin.Propagates` — Enables System objects to operate in the MATLAB System block using the interpreted execution

Best Practices for Defining System Objects

A System object is a specialized kind of MATLAB object that is optimized for iterative processing. Use System objects when you need to call the `step` method multiple times or process data in a loop. When defining your own System object, use the following suggestions to help your code run efficiently.

- Define all one-time calculations in the `setupImpl` method and cache the results in a private property. Use the `stepImpl` method for repeated calculations.
- If properties are accessed more than once in the `stepImpl` method, cache those properties as local variables inside the method. A typical example of multiple property access is a loop. Iterative calculations using cached local variables run faster than calculations that must access the properties of an object. When the calculations for the method complete, you can save the local cached results back to the properties of that System object. Copy frequently used tunable properties into private properties. This best practice also applies to the `updateImpl` and `outputImpl` methods.

In this example, `k` is accessed multiple times in each loop iteration, but is saved to the object property only once.

```
function y = stepImpl(obj,x)
    k = obj.MyProp;
    for p=1:100
        y = k * x;
        k = k + 0.1;
    end
    obj.MyProp = k;
end
```

- Do not use string comparisons or string-based switch statements in the `stepImpl` method. Instead, create a method handle in `setupImpl`. This handle points to a method in the same class definition file. Use that handle in a loop in `stepImpl`.

This example shows how to use method handles and cached local variables in a loop to implement an efficient object. In `setupImpl`, choose `myMethod1` or `myMethod2` based on a string comparison and assign the method handle to the `pMethodHandle` property. Because there is a loop in `stepImpl`, assign the `pMethodHandle` property to a local method handle, `myFun`, and then use `myFun` inside the loop.

```
classdef MyClass < matlab.System
    function setupImpl(obj)
        if strcmp(obj.Method, 'Method1')
```

```

        obj.pMethodHandle = @myMethod1;
    else
        obj.pMethodHandle = @myMethod2;
    end
end
function y = stepImpl(obj,x)
    myFun = obj.pMethodHandle;
    for p=1:1000
        y = myFun(obj,x)
    end
end
end
function y = myMethod1(x)
    y = x+1;
end
function y = myMethod2(x)
    y = x-1;
end
end
end

```

- If the number of System object inputs does not change, do not implement the `getNumInputsImpl` method. Also do not implement the `getNumInputsImpl` method when you explicitly list the inputs in the `stepImpl` method instead of using `varargin`. The same caveats apply to the `getNumOutputsImpl` and `varargout` outputs.
- For the `getNumInputsImpl` and `getNumOutputsImpl` methods, if you set the return argument from an object property, that object property must have the `Nontunable` attribute.
- If the variables in a method do not need to retain their values between calls use local scope for those variables in that method.
- For properties that do not change, define them in as `Nontunable` properties. `Tunable` properties have slower access times than `Nontunable` properties
- Use the `protected` or `private` attribute instead of the `public` attribute for a property, whenever possible. Some `public` properties have slower access times than `protected` and `private` properties.
- Avoid using customized `step`, `get`, or `set` methods, whenever possible.
- Avoid using string comparisons within customized `step`, `get`, or `set` methods, whenever possible. Use `setupImpl` for string comparisons instead.

