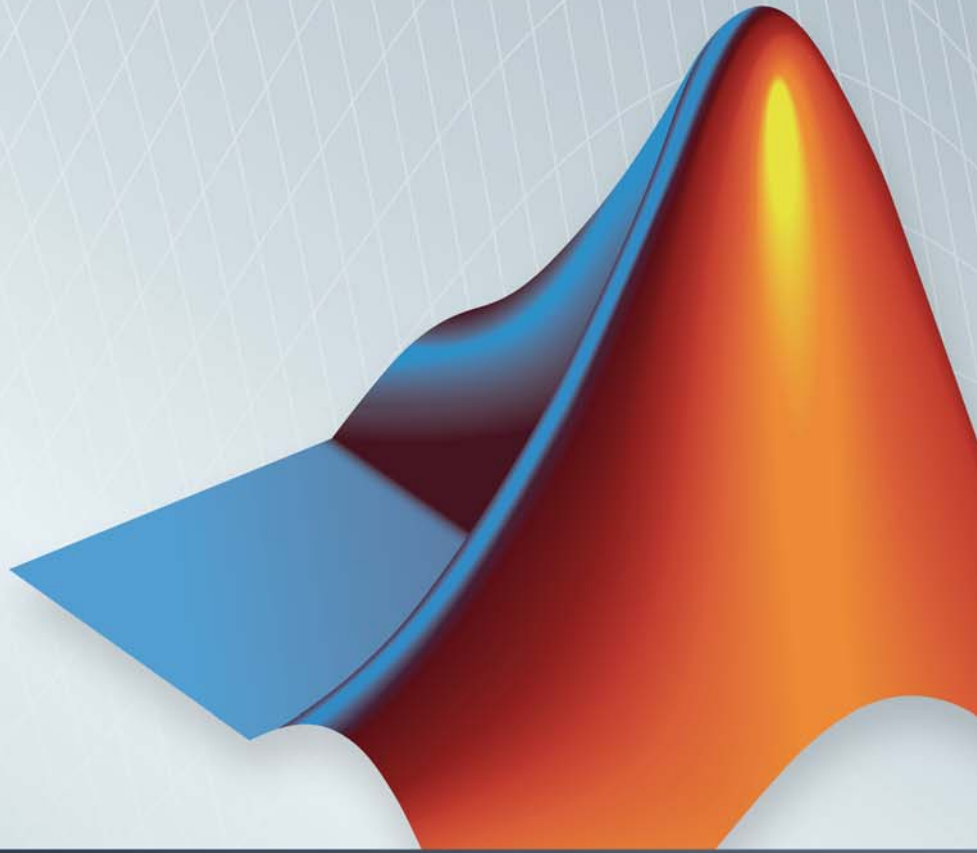


HDL Coder™

User's Guide

R2013b



MATLAB®



How to Contact MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

For contact information about worldwide offices, see the MathWorks Web site.

HDL Coder™ User's Guide

© COPYRIGHT 2012-2013 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)

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	1-5
Logical Operators	1-5
Control Flow Statements	1-7
Vector Function Limitations Related to Control Statements	1-8
Persistent Variables	1-9
Persistent Array Variables	1-11
Complex Data Type Support	1-12
Declaring Complex Signals	1-12
Conversion Between Complex and Real Signals	1-13
Arithmetic Operations on Complex Numbers	1-14
Support for Vectors of Complex Numbers	1-18
Other Operations on Complex Numbers	1-19
System Objects	1-21
Why Use System Objects?	1-21
Predefined System Objects Supported for HDL Code Generation	1-21
User-Defined System Objects	1-22

Limitations of HDL Code Generation for System Objects ..	1-22
System object Examples for HDL Code Generation	1-23
Load constants from a MAT-File	1-24
Generate Code for User-Defined System Objects	1-25
Map Matrices to ROM	1-27
Fixed-Point Bitwise Functions	1-28
Overview	1-28
Bitwise Functions Supported for HDL Code Generation ..	1-28
Fixed-Point Run-Time Library Functions	1-35
Fixed-Point Function Limitations	1-39
Model State with Persistent Variables and System Objects	1-41
Bit Shifting and Bit Rotation	1-45
Bit Slicing and Bit Concatenation	1-48
Guidelines for Efficient HDL Code	1-50
MATLAB Design Requirements for HDL Code Generation	1-51
What Is a MATLAB Test Bench?	1-52
MATLAB Test Bench Requirements and Best Practices	1-53
MATLAB Test Bench Requirements	1-53
MATLAB Test Bench Best Practices	1-53

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	2-13
Implement RAM Using hdlram	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	2-18

Fixed-Point Conversion

3

Floating-Point to Fixed-Point Conversion	3-2
Fixed-Point Type Conversion and Refinement	3-18

Working with Generated Fixed-Point Files	3-29
Specify Type Proposal Options	3-37
Log Data for Histogram	3-40
View and Modify Variable Information	3-43
View Variable Information	3-43
Modify Variable Information	3-43
Revert Changes	3-45
Promote Sim Min and Sim Max Values	3-46
Automated Fixed-Point Conversion	3-47
License Requirements	3-47
Fixed-Point Conversion Capabilities	3-47
Code Coverage	3-49
Proposing Data Types	3-53
Viewing Functions	3-55
Viewing Variables	3-55
Histogram	3-57
Function Replacements	3-58
Validating Types	3-58
Testing Numerics	3-59

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	4-7
Rules for Specifying Properties of Primary Inputs	4-12
Methods for Defining Properties of Primary Inputs	4-12

Define Input Properties by Example at the Command Line	4-13
Specify Constant Inputs at the Command Line	4-16
Specify Variable-Size Inputs at the Command Line	4-18
Basic HDL Code Generation with the Workflow	
Advisor	4-20
HDL Code Generation from System Objects	4-29
Generate Instantiable Code for Functions	4-34
How to Generate Instantiable Code for Functions	4-34
Generate Code Inline for Specific Functions	4-34
Limitations for Instantiable Code Generation for Functions	4-34
Enable MATLAB Function Block Generation	4-35
Requirements for MATLAB Function Block Generation ..	4-35
Enable MATLAB Function Block Generation	4-35
Results of MATLAB Function Block Generation	4-35
System Design with HDL Code Generation from MATLAB and Simulink	4-37
Generate Xilinx System Generator Black Box Block ..	4-42
Requirements for System Generator Black Box Block Generation	4-42
Enable System Generator Black Box Block Generation ...	4-42
Results of System Generator Black Box Block Generation	4-43
Generate Xilinx System Generator for DSP Black Box from MATLAB HDL Design	4-44
Generate HDL Code from MATLAB Code Using the Command Line Interface	4-51
Specify the Clock Enable Rate	4-57
Why Specify the Clock Enable Rate?	4-57
How to Specify the Clock Enable Rate	4-57

Specify Test Bench Clock Enable Toggle Rate	4-59
When to Specify Test Bench Clock Enable Toggle Rate ...	4-59
How to Specify Test Bench Clock Enable Toggle Rate	4-59
Generate an HDL Coding Standard Report	4-61
Using the HDL Workflow Advisor	4-61
Using the Command Line	4-61
Generate an HDL Lint Tool Script	4-63
How To Generate an HDL Lint Tool Script	4-63

Verification

5

Verify Code with HDL Test Bench	5-2
Generate Test Bench With File I/O	5-6
When to Use File I/O In Test Bench	5-6
How Test Bench Generation with File I/O Works	5-6
Test Bench Data Files	5-7
How to Generate Test Bench with File I/O	5-7
Limitations When Using File I/O In Test Bench	5-7

Deployment

6

Program Standalone FPGA with FPGA Turnkey	
Workflow	6-2
Before You Begin	6-2
Create a Project	6-2
Convert Design To Fixed-Point	6-3
Map Design Ports to Target Interface	6-3
Generate Programming File and Download To	
Hardware	6-5

Hardware and Software Codesign for Xilinx Zynq-7000 Platform	6-7
Hardware and Software Codesign Workflow	6-8
Generate Synthesis Scripts	6-17
Install Support for Altera FPGA Boards	6-18
Install Support for Xilinx FPGA Boards	6-19
Install Support for Xilinx Zynq-7000 Platform	6-20

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	7-4
RAM Mapping Requirements for dsp.Delay System Objects	7-7
RAM Mapping Comparison for MATLAB Code	7-9
Pipelining	7-10
Port Registers	7-10
Input and Output Pipeline Registers	7-10
Variable Pipelining	7-10
Register Inputs and Outputs	7-11
Insert Input and Output Pipeline Registers	7-12

Distributed Pipelining	7-13
What is Distributed Pipelining?	7-13
Benefits and Costs of Distributed Pipelining	7-13
Selected Bibliography	7-13
Pipeline MATLAB Variables	7-14
Using the HDL Workflow Advisor	7-14
Using the Command Line Interface	7-14
Limitations of MATLAB Variable Pipelining	7-14
Optimize MATLAB Loops	7-16
How to Optimize MATLAB Loops	7-16
Limitations for MATLAB Loop Optimization	7-16
Constant Multiplier Optimization	7-17
Specify Constant Multiplier Optimization	7-19
Distributed Pipelining for Clock Speed Optimization	7-20
Map Matrices to Block RAMs to Reduce Area	7-29
Resource Sharing of Multipliers to Reduce Area	7-34
Loop Streaming to Reduce Area	7-43
Constant Multiplier Optimization to Reduce Area	7-50

HDL Workflow Advisor Reference

8

HDL Workflow Advisor	8-2
Overview	8-2
MATLAB to HDL Code and Synthesis	8-7

MATLAB to HDL Code Conversion	8-7
Code Generation: Target Tab	8-7
Code Generation: Coding Style Tab	8-8
Code Generation: Clocks and Ports Tab	8-10
Code Generation: Test Bench Tab	8-12
Code Generation: Optimizations Tab	8-15
Simulation and Verification	8-16
Synthesis and Analysis	8-17

HDL Code Generation from Simulink®

Code Generation Options in the HDL Coder Dialog Boxes

9

Set HDL Code Generation Options	9-2
HDL Code Generation Options in the Configuration	
Parameters Dialog Box	9-2
HDL Code Generation Options in the Model Explorer	9-3
Code Menu	9-4
HDL Code Options in the Block Context Menu	9-5
The HDL Block Properties Dialog Box	9-6
HDL Code Generation Pane: General	9-8
HDL Code Generation Top-Level Pane Overview	9-10
Generate HDL for	9-12
Language	9-13
Folder	9-14
Generate HDL code	9-15
Generate validation model	9-16
Generate traceability report	9-17
Generate resource utilization report	9-18
Generate optimization report	9-19
Generate model Web view	9-20
HDL Code Generation Pane: Global Settings	9-22
Global Settings Overview	9-25
Reset type	9-26
Reset asserted level	9-27

Clock input port	9-28
Clock enable input port	9-29
Reset input port	9-30
Clock inputs	9-31
Oversampling factor	9-32
Comment in header	9-33
Verilog file extension	9-34
VHDL file extension	9-35
Entity conflict postfix	9-36
Package postfix	9-37
Reserved word postfix	9-38
Module name prefix	9-38
Split entity and architecture	9-40
Split entity file postfix	9-42
Split arch file postfix	9-43
Clocked process postfix	9-44
Enable prefix	9-45
Pipeline postfix	9-46
Complex real part postfix	9-47
Complex imaginary part postfix	9-48
Input data type	9-49
Output data type	9-50
Clock enable output port	9-52
Balance delays	9-53
Hierarchical distributed pipelining	9-54
Optimize timing controller	9-55
Minimize clock enables	9-57
RAM mapping threshold (bits)	9-60
Max oversampling	9-61
Max computation latency	9-62
Represent constant values by aggregates	9-63
Use rising_edge for registers	9-64
Loop unrolling	9-65
Use Verilog `timescale directives	9-66
Inline VHDL configuration	9-67
Concatenate type safe zeros	9-68
Emit time/date stamp in header	9-69
HDL coding standard	9-70
Scalarize vector ports	9-71
Minimize intermediate signals	9-72
Include requirements in block comments	9-73
Inline MATLAB Function block code	9-74
Generate parameterized HDL code from masked subsystem	9-75

Initialize all RAM blocks	9-76
RAM Architecture	9-76
HDL Code Generation Pane: Test Bench	9-78
Test Bench Overview	9-80
HDL test bench	9-81
Cosimulation blocks	9-82
Cosimulation model for use with:	9-84
Test bench name postfix	9-85
Force clock	9-86
Clock high time (ns)	9-87
Clock low time (ns)	9-88
Hold time (ns)	9-89
Setup time (ns)	9-90
Force clock enable	9-91
Clock enable delay (in clock cycles)	9-92
Force reset	9-94
Reset length (in clock cycles)	9-95
Hold input data between samples	9-97
Initialize test bench inputs	9-98
Multi-file test bench	9-99
Test bench reference postfix	9-101
Test bench data file name postfix	9-102
Use file I/O to read/write test bench data	9-103
Ignore output data checking (number of samples)	9-103
HDL Code Generation Pane: EDA Tool Scripts	9-106
EDA Tool Scripts Overview	9-108
Generate EDA scripts	9-109
Generate multicycle path information	9-110
Compile file postfix	9-111
Compile initialization	9-112
Compile command for VHDL	9-113
Compile command for Verilog	9-114
Compile termination	9-115
Simulation file postfix	9-116
Simulation initialization	9-117
Simulation command	9-118
Simulation waveform viewing command	9-119
Simulation termination	9-120
Choose synthesis tool	9-121
Synthesis file postfix	9-123
Synthesis initialization	9-124

Synthesis command	9-125
Synthesis termination	9-126
Choose HDL lint tool	9-126
Lint initialization	9-127
Lint command	9-128
Lint termination	9-128

Specifying Block Implementations and Parameters for HDL Code Generation

10

Set and View HDL Block Parameters	10-2
Set HDL Block Parameters from the GUI	10-2
Set HDL Block Parameters from the Command Line	10-2
View All HDL Block Parameters	10-3
View Non-Default HDL Block Parameters	10-4
Set HDL Block Parameters for Multiple Blocks	10-5
View HDL Model Parameters	10-7

Guide to Supported Blocks and Block Implementations

11

Generate a Supported Blocks Report	11-2
Blocks Supported for HDL Code Generation	11-3
Blocks with Multiple Implementations	11-16
Block Implementations	11-16
Pass-through and No HDL Implementations	11-27
Cascade Implementation Best Practices	11-27

Block-Specific Usage, Requirements, and Restrictions	11-29
Block Usage, Requirements, and Restrictions	11-29
Restrictions on Use of Blocks in the Test Bench	11-49
Block Implementation Parameters	11-50
Overview	11-50
BalanceDelays	11-51
ConstMultiplierOptimization	11-52
CoeffMultipliers	11-54
ConstrainedOutputPipeline	11-56
Distributed Arithmetic Implementation Parameters for	
Digital Filter Blocks	11-57
DistributedPipelining	11-70
FlattenHierarchy	11-71
InputPipeline	11-73
InstantiateFunctions	11-74
LoopOptimization	11-75
MapPersistentVarsToRAM	11-77
OutputPipeline	11-80
Pipelining Implementation Parameters for Filter Blocks ..	11-81
RAM	11-85
ResetType	11-86
ShiftRegister	11-88
UseRAM	11-90
Speed vs. Area Optimizations for FIR Filter	
Implementations	11-95
Interface Generation Parameters	11-102
Blocks That Support Complex Data	11-103
Complex Coefficients and Data Support for the Digital	
Filter and Biquad Filter Blocks	11-107
Blocks That Support Buses	11-109
Supported Bus Blocks	11-109
Settings and Requirements	11-109
Limitations	11-112
See Also	11-113
Lookup Table Block Support	11-114
n-D Lookup Table	11-114
Prelookup	11-115

Direct Lookup Table (n-D)	11-116
1-D Lookup Table	11-117
2-D Lookup Table	11-117

Generating HDL Code for Multirate Models

12

Code Generation from Multirate Models	12-2
Configure Multirate Models for HDL Code	
Generation	12-3
Overview	12-3
Configuring Model Parameters	12-3
Configuring Sample Rates in the Model	12-4
Block Configuration and Restrictions For Multirate DUTs	12-4
Code Generation from a Multirate DUT	12-6
Generate a Global Oversampling Clock	12-9
Why Use a Global Oversampling Clock?	12-9
Requirements for the Oversampling Factor	12-9
Specifying the Oversampling Factor From the GUI	12-10
Specifying the Oversampling Factor From the Command Line	12-12
Resolving Oversampling Rate Conflicts	12-12
Generate Multicycle Path Information Files	12-16
Overview	12-16
Format and Content of a Multicycle Path Information File	12-17
File Naming and Location Conventions	12-23
Generating Multicycle Path Information Files Using the GUI	12-23
Generating Multicycle Path Information Files Using the Command Line	12-24
Limitations	12-24
Example of Generating a Multicycle Path Information File	12-26

HDL Properties for Controlling Multirate Code	
Generation	12-27
Overview	12-27
HoldInputDataBetweenSamples	12-27
OptimizeTimingController	12-27

The hdl demolib Block Library

13

Open the hdl demolib Library	13-2
RAM Blocks	13-3
Overview of RAM Blocks	13-3
Dual Port RAM Block	13-5
Simple Dual Port RAM Block	13-7
Single Port RAM Block	13-8
Code Generation for RAM Blocks	13-11
Limitations for RAM Blocks	13-12
Generate RAM Without Clock Enable Ports	13-13
Build a ROM Block with Simulink Blocks	13-14
HDL Counter	13-15
Overview	13-15
Counter Modes	13-15
Control Ports	13-17
Defining the Counter Data Type and Size	13-20
HDL Implementation and Implementation Parameters ..	13-21
Parameters and Dialog Box	13-22
HDL FFT	13-27
Overview	13-27
Block Inputs and Outputs	13-27
HDL Implementation and Implementation Parameters ..	13-30
Parameters and Dialog Box	13-30
Signal Processing with the HDL FFT Block	13-35

HDL FIFO	13-36
Overview	13-36
Block Inputs and Outputs	13-36
HDL Implementation and Implementation Parameters ..	13-37
Parameters and Dialog Box	13-37
HDL Streaming FFT	13-40
Overview	13-40
HDL Streaming FFT Block Example	13-40
Block Inputs and Outputs	13-40
Timing Description	13-41
HDL Implementation and Implementation Parameters ..	13-45
Parameters and Dialog Box	13-45
Bitwise Operators	13-50
Overview of Bitwise Operator Blocks	13-50
Bit Concat	13-52
Bit Reduce	13-55
Bit Rotate	13-57
Bit Shift	13-59
Bit Slice	13-61

Generating Bit-True Cycle-Accurate Models

14

What Is the Generated Model?	14-2
Locate Numeric Differences After Speed Optimization	14-4
View Latency Differences After Area Optimization ...	14-9
Defaults and Options for Generated Models	14-12
Defaults for Model Generation	14-12
GUI Options	14-13
Generated Model Properties for makehdl	14-15
Limitations for Generated Models	14-17

Fixed-Point Limitation	14-17
Double-Precision Limitation	14-17
Model Properties Not Supported for Generated Models ...	14-18

Optimization

15

Optimization With Constrained Overclocking	15-2
Why Constrain Overclocking?	15-2
When to Use Constrained Overclocking	15-2
Set Overclocking Constraints	15-3
Constrained Overclocking Limitations	15-4
Maximum Oversampling Ratio	15-5
What Is the Maximum Oversampling Ratio?	15-5
Specify Maximum Oversampling Ratio	15-5
Maximum Oversampling Ratio Limitations	15-6
Maximum Computation Latency	15-7
What Is Maximum Computation Latency?	15-7
Specify Maximum Computation Latency	15-8
Maximum Computation Latency Restrictions	15-8
Streaming	15-9
What is Streaming?	15-9
Specify Streaming	15-10
Requirements and Limitations for Streaming	15-10
Area Reduction with Streaming	15-13
The Validation Model	15-20
What Is the Validation Model?	15-24
Resource Sharing	15-25
What Is Resource Sharing?	15-25
Benefits and Costs of Resource Sharing	15-26
Specify Resource Sharing	15-26
Requirements for Resource Sharing	15-27

Resource Sharing Information in Reports	15-29
Check Compatibility for Resource Sharing	15-30
Delay Balancing	15-31
Why Use Delay Balancing	15-31
Specify Delay Balancing	15-32
Delay Balancing Limitations	15-33
Resolve Numerical Mismatch with Delay Balancing ..	15-34
Hierarchy Flattening	15-38
What Is Hierarchy Flattening?	15-38
When To Flatten Hierarchy	15-38
Prerequisites For Hierarchy Flattening	15-38
Options For Hierarchy Flattening	15-39
How To Flatten Hierarchy	15-39
Limitations For Hierarchy Flattening	15-40
Loop Optimization	15-41
Loop Streaming	15-41
Loop Unrolling	15-41
Optimize Loops in the MATLAB Function Block	15-42
MATLAB Function Block Loop Optimization Options	15-42
How to Optimize MATLAB Function Block Loops	15-42
Limitations for MATLAB Function Block Loop Optimization	15-43
RAM Mapping	15-44
RAM Mapping with the MATLAB Function Block	15-45
Insert Distributed Pipeline Registers in a Subsystem ..	15-48
Distributed Pipelining and Hierarchical Distributed Pipelining	15-53
What is Distributed Pipelining?	15-53
Benefits and Costs of Distributed Pipelining	15-55

Requirements for Distributed Pipelining	15-56
Specify Distributed Pipelining	15-56
Limitations of Distributed Pipelining	15-57
What is Hierarchical Distributed Pipelining?	15-59
Benefits of Hierarchical Distributed Pipelining	15-61
Specify Hierarchical Distributed Pipelining	15-61
Limitations of Hierarchical Distributed Pipelining	15-62
Distributed Pipelining Workflow	15-62
Selected Bibliography	15-62
Constrained Output Pipelining	15-63
What is Constrained Output Pipelining?	15-63
When To Use Constrained Output Pipelining	15-63
Requirements for Constrained Output Pipelining	15-63
Specify Constrained Output Pipelining	15-64
Limitations of Constrained Output Pipelining	15-64
Pipeline Variables in the MATLAB Function Block ...	15-65
Using the HDL Block Properties Dialog Box	15-65
Using the Command Line	15-65
Limitations of Variable Pipelining	15-65
Reduce Critical Path With Distributed Pipelining	15-67

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

16

Create and Use Code Generation Reports	16-2
Information Included in Code Generation Reports	16-2
HDL Code Generation Report Summary	16-3
Resource Utilization Report	16-5
Optimization Report	16-7
Hierarchical Distributed Pipelining in the Optimization Report	16-9

Traceability Report	16-10
Traceability Report Overview	16-10
Generating a Traceability Report from Configuration Parameters	16-14
Generating a Traceability Report from the Command Line	16-17
Keeping the Report Current	16-20
Tracing from Code to Model	16-20
Tracing from Model to Code	16-22
Mapping Model Elements to Code Using the Traceability Report	16-25
Traceability Report Limitations	16-27
Web View of Model in Code Generation Report	16-28
About Model Web View	16-28
Generate HTML Code Generation Report with Model Web View	16-29
Model Web View Limitations	16-33
Generate Code with Annotations or Comments	16-34
Simulink Annotations	16-34
Text Comments	16-34
Requirements Comments and Hyperlinks	16-35
Check Your Model for HDL Compatibility	16-38
Create a Supported Blocks Library	16-41
Trace Code Using the Mapping File	16-43
Add or Remove the HDL Configuration Component ...	16-46
What Is the HDL Configuration Component?	16-46
Adding the HDL Coder Configuration Component To a Model	16-46
Removing the HDL Coder Configuration Component From a Model	16-47

17

HDL Coding Standard Report	17-2
Rule Summary	17-2
Rule Hierarchy	17-3
How To Fix Warnings and Errors	17-3
HDL Coding Standards	17-4
Generate an HDL Coding Standard Report	17-5
Using the HDL Workflow Advisor	17-5
Using the Command Line	17-5
HDL Coding Standard Rules	17-7
Generate an HDL Lint Tool Script	17-11
How To Generate an HDL Lint Tool Script	17-11

Interfacing Subsystems and Models to HDL Code

18

Generate Black Box Interface for Subsystem	18-2
What Is a Black Box Interface?	18-2
Generate a Black Box Interface for a Subsystem	18-2
Generate Code for a Black Box Subsystem	
Implementation	18-6
Restriction for Multirate DUTs	18-7
Generate Reusable Code for Atomic Subsystems	18-8
Generate Reusable Code for Identical Atomic	
Subsystems	18-8
Generate Reusable Code for Atomic Subsystems with	
Tunable Mask Parameters	18-12
Model Reference Code Generation Options	18-17

Model Referencing for HDL Code Generation	18-18
Benefits of Model Referencing for Code Generation	18-18
How To Generate Code for a Referenced Model	18-18
Limitations for Model Reference Code Generation	18-19
Generate Black Box Interface for Referenced Model ..	18-21
When to Generate a Black Box Interface	18-21
How to Generate a Black Box Interface	18-21
Generate Code for Enabled and Triggered	
Subsystems	18-23
Code Generation for Enabled Subsystems	18-23
Code Generation for Triggered Subsystems	18-24
Best Practices for Using Enabled and Triggered	
Subsystems	18-26
Create a Xilinx System Generator Subsystem	18-27
Why Use Xilinx System Generator Subsystems?	18-27
Requirements for Xilinx System Generator Subsystems ..	18-27
How to Create a Xilinx System Generator Subsystem	18-28
Limitations for Code Generation from Xilinx System	
Generator Subsystems	18-28
Create an Altera DSP Builder Subsystem	18-30
Why Use Altera DSP Builder Subsystems?	18-30
Requirements for Altera DSP Builder Subsystems	18-30
How to Create an Altera DSP Builder Subsystem	18-31
Determine Clocking Requirements for Altera DSP Builder	
Subsystems	18-31
Limitations for Code Generation from Altera DSP Builder	
Subsystems	18-32
Using Xilinx System Generator for DSP with HDL	
Coder	18-33
Code Generation for HDL Cosimulation Blocks	18-37
Generate a Cosimulation Model	18-39
What Is A Cosimulation Model?	18-39
Generating a Cosimulation Model from the GUI	18-40
Structure of the Generated Model	18-46

Launching a Cosimulation	18-53
The Cosimulation Script File	18-55
Complex and Vector Signals in the Generated Cosimulation Model	18-58
Generating a Cosimulation Model from the Command Line	18-60
Naming Conventions for Generated Cosimulation Models and Scripts	18-61
Limitations for Cosimulation Model Generation	18-61
Customize the Generated Interface	18-63
Pass-Through and No-Op Implementations	18-66

Stateflow HDL Code Generation Support

19

Introduction to Stateflow HDL Code Generation	19-2
Overview	19-2
Examples	19-2
Requirements for Stateflow HDL Code Generation ...	19-4
Overview	19-4
Location of Charts in the Model	19-4
Data Type Usage	19-4
Chart Initialization	19-5
Registered Output	19-5
Restrictions on Imported Code	19-6
Using Input and Output Events	19-6
Using For Loops	19-6
Other Restrictions	19-7
Map Chart Semantics to HDL	19-9
Hardware Realization of Stateflow Semantics	19-9
Restrictions for HDL Realization	19-11
Generate HDL for Mealy and Moore Finite State Machines	19-13

Overview	19-13
Generating HDL for a Mealy Finite State Machine	19-14
Generating HDL Code for a Moore Finite State Machine ..	19-18
Structure a Model for HDL Code Generation	19-24
Design Patterns Using Advanced Chart Features	19-25
Temporal Logic	19-25
Graphical Function	19-27
Hierarchy and Parallelism	19-29
Stateless Charts	19-30
Truth Tables	19-32

Generating HDL Code with the MATLAB Function Block

20

HDL Applications for the MATLAB Function Block ...	20-2
Viterbi Decoder with the MATLAB Function Block ...	20-3
Code Generation from a MATLAB Function Block	20-4
Counter Model Using the MATLAB Function block	20-4
Setting Up	20-7
Creating the Model and Configuring General Model Settings	20-8
Adding a MATLAB Function Block to the Model	20-8
Set Fixed-Point Options for the MATLAB Function Block	20-10
Programming the MATLAB Function Block	20-14
Constructing and Connecting the DUT_eML_Block Subsystem	20-15
Compiling the Model and Displaying Port Data Types ...	20-18
Simulating the eml_hdl_incrementer_tut Model	20-19
Generating HDL Code	20-20
MATLAB Function Block Design Patterns for HDL ...	20-23
The eml_hdl_design_patterns Library	20-23

Efficient Fixed-Point Algorithms	20-25
Model State Using Persistent Variables	20-29
Creating Intellectual Property with the MATLAB Function Block	20-30
Nontunable Parameter Arguments	20-31
Modeling Control Logic and Simple Finite State Machines	20-31
Modeling Counters	20-33
Modeling Hardware Elements	20-35
Design Guidelines for the MATLAB Function Block ...	20-37
Introduction	20-37
Use Compiled External Functions With MATLAB Function Blocks	20-37
Build the MATLAB Function Block Code First	20-38
Use the hdlfimath Utility for Optimized FIMATH Settings	20-38
Use Optimal Fixed-Point Option Settings	20-40
Set the Output Data Type of MATLAB Function Blocks Explicitly	20-41
Distributed Pipeline Insertion for MATLAB Function Blocks	20-42
Overview	20-42
Distributed Pipelining in a Multiplier Chain	20-42
Limitations for MATLAB Function Block Code Generation	20-49
MATLAB Language Support	20-50

Generating Scripts for HDL Simulators and Synthesis Tools

21

Generate Scripts for Compilation, Simulation, and Synthesis	21-2
--	-------------

Structure of Generated Script Files	21-3
Properties for Controlling Script Generation	21-4
Enabling and Disabling Script Generation	21-4
Customizing Script Names	21-4
Customizing Script Code	21-5
Examples	21-7
Control Script Generation with the EDA Tool Scripts	
Pane	21-9
Compilation Script Options	21-10
Simulation Script Options	21-11
Synthesis Script Options	21-13

Using the HDL Workflow Advisor

22

What Is the HDL Workflow Advisor?	22-3
Open the HDL Workflow Advisor	22-4
Using the HDL Workflow Advisor Window	22-7
Save and Restore HDL Workflow Advisor State	22-10
How the Save and Restore Process Works	22-10
Limitations of the Save and Restore Process	22-10
Save the HDL Workflow Advisor State	22-10
Restore the HDL Workflow Advisor State	22-12
Fix a Workflow Advisor Warning or Failure	22-14
View and Save HDL Workflow Advisor Reports	22-17
Viewing HDL Workflow Advisor Reports	22-17
Saving HDL Workflow Advisor Reports	22-21
Map to an FPGA Floating-Point Library	22-22
What is an FPGA Floating-Point Library?	22-22

Why Map to an FPGA Floating Point Library?	22-22
Supported Floating-Point Operations	22-22
Setup for FPGA Floating-Point Library Mapping	22-24
How to Map to an FPGA Floating-Point Library	22-24
FPGA Floating-Point Library Mapping Results Analysis ..	22-26
Limitations for FPGA Floating-Point Library Mapping ...	22-26
FPGA Synthesis and Analysis	22-28
FPGA Synthesis and Analysis Tasks Overview	22-28
Creating a Synthesis Project	22-28
Performing Synthesis, Mapping, and Place and Route	22-30
Annotating Your Model with Critical Path Information ..	22-35
Automated Workflows for Specific Targets and Tools	22-40
Generate xPC Target Interface for Speedgoat Boards	22-42
Select a Speedgoat Target Device	22-42
Set the Target Interface for Speedgoat Boards	22-45
Code Generation, Synthesis, and Generation of xPC Target Interface Subsystem	22-48
Target Altera FPGA Development Boards	22-51
Before You Begin	22-51
Open the Model	22-51
Select the Target Device	22-52
Set Target Interface and Target Frequency	22-53
Generate Code, Synthesize, and Program Target Device ..	22-56
Target Xilinx FPGA Development Boards	22-58
Before You Begin	22-58
Open the Model	22-58
Select the Target Device	22-59
Set Target Interface and Target Frequency	22-60
Generate Code, Synthesize, and Program Target Device ..	22-63
Custom IP Core Generation	22-65
Custom IP Core Architectures	22-65
Target Platform Interfaces	22-66
Processor/FPGA Synchronization	22-67

Custom IP Core Generated Files	22-67
Custom IP Core Report	22-68
Summary	22-68
Target Interface Configuration	22-69
Register Address Mapping	22-70
IP Core User Guide	22-71
IP Core File List	22-74
Hardware and Software Codesign for Xilinx Zynq-7000 Platform	22-75
Generate a Custom IP Core	22-76
Generate a Generic Custom IP Core	22-76
Generate a Custom IP Core for the Zynq-7000 Platform ..	22-77
Requirements and Limitations for Custom IP Core Generation	22-78
Processor and FPGA Synchronization	22-80
Free Running Mode	22-80
Coprocesing – Blocking Mode	22-81
Coprocesing – Nonblocking With Delay Mode	22-81
Hardware and Software Codesign Workflow	22-83
Install Support for Altera FPGA Boards	22-92
Install Support for Xilinx FPGA Boards	22-93
Install Support for Xilinx Zynq-7000 Platform	22-94

HDL Test Bench

23

Generate Test Bench With File I/O	23-2
When to Use File I/O In Test Bench	23-2
How Test Bench Generation with File I/O Works	23-2

Test Bench Data Files	23-3
How to Generate Test Bench with File I/O	23-3
Limitations When Using File I/O In Test Bench	23-4

FPGA Board Customization

24

FPGA Board Customization	24-2
Feature Description	24-2
Custom Board Management	24-2
FPGA Board Requirements	24-3
Create Custom FPGA Board Definition	24-7
Create Xilinx KC705 Evaluation Board Definition	
File	24-8
Overview	24-8
What You Need to Know Before Starting	24-8
Start New FPGA Board Wizard	24-9
Provide Basic Board Information	24-10
Specify FPGA Interface Information	24-12
Enter FPGA Pin Numbers	24-13
Run Optional Validation Tests	24-15
Save Board Definition File	24-17
Use New FPGA Board	24-18
FPGA Board Manager	24-22
Introduction	24-22
Filter	24-24
Search	24-24
FIL Enabled/Turnkey Enabled	24-24
Create Custom Board	24-24
Add Board From File	24-24
Get More Boards	24-24
View/Edit	24-25
Remove	24-25
Clone	24-25
Validate	24-25

New FPGA Board Wizard	24-26
Basic Information	24-28
Interfaces	24-29
FIL I/O	24-31
Turnkey I/O	24-33
Validation	24-36
Finish	24-36
FPGA Board Editor	24-37
General	24-37
Interface	24-39

HDL Workflow Advisor Tasks

25

HDL Workflow Advisor Tasks	25-2
HDL Workflow Advisor Tasks Overview	25-3
Set Target Overview	25-6
Set Target Device and Synthesis Tool	25-7
Set Target Library	25-9
Set Target Interface	25-10
Set Target Frequency	25-11
Set Target Interface	25-12
Prepare Model For HDL Code Generation Overview	25-13
Check Global Settings	25-14
Check Algebraic Loops	25-15
Check Block Compatibility	25-16
Check Sample Times	25-17
Check FPGA-in-the-Loop Compatibility	25-18
HDL Code Generation Overview	25-19
Set Code Generation Options Overview	25-20
Set Basic Options	25-21
Set Advanced Options	25-22
Set Testbench Options	25-23
Generate RTL Code and Testbench	25-24
Generate RTL Code and IP Core	25-26
FPGA Synthesis and Analysis Overview	25-27
Create Project	25-28
Perform Synthesis and P/R Overview	25-29
Perform Logic Synthesis	25-30

Perform Mapping	25-31
Perform Place and Route	25-32
Annotate Model with Synthesis Result	25-33
Download to Target Overview	25-35
Generate Programming File	25-36
Program Target Device	25-37
Generate xPC Target Interface	25-38
Save and Restore HDL Workflow Advisor State	25-39
FPGA-in-the-Loop Implementation	25-39
Set FIL Options	25-39
Build FPGA-in-the-Loop	25-39
Check USRP® Compatibility	25-40
Verify with HDL Cosimulation	25-40
Generate FPGA Implementation	25-40
Check SDR Compatibility	25-40
SDR FPGA Implementation	25-41
Set SDR Options	25-41
Build SDR	25-43
Embedded System Integration	25-44
Create Project	25-44
Generate Software Interface Model	25-44
Build FPGA Bitstream	25-45
Program Target Device	25-45

Code Generation Control Files

26

READ THIS FIRST: Control File Compatibility and	
Conversion Issues	26-2
Conversion From Use of Control Files Recommended	26-2
Detaching Existing Models From Control Files	26-2
Applying Control File Settings	26-3
Backwards Compatibility	26-3
Overview of Control Files	26-4
What Is a Control File?	26-4
Selectable Block Implementations and Implementation	
Parameters	26-5
Implementation Mappings	26-5

Structure of a Control File	26-7
Code Generation Control Objects and Methods	26-8
Overview	26-8
hdlnewcontrol	26-8
forEach	26-8
forAll	26-13
set	26-13
generateHDLFor	26-14
hdlnewcontrolfile	26-15
Using Control Files in the Code Generation Process ..	26-16
Where to Locate Your Control Files	26-16
Making Your Control Files More Portable	26-16
Specifying Block Implementations and Parameters in the Control File	26-17
Overview	26-17
Generating Selection/Action Statements with the hdlnewforeach Function	26-18
Generating Black Box Control Statements Using hdlnewblackbox	26-23

Support Packages

Support Packages

27

Support Packages and Support Package Installer	27-2
What Is a Support Package?	27-2
What Is Support Package Installer?	27-2
Install This Support Package on Other Computers ...	27-4
Open Examples for This Support Package	27-6
Using the Help Browser	27-6

Using the Block Library	27-8
Using Support Package Installer	27-9

Index

HDL Code Generation from MATLAB

- Chapter 1, “MATLAB Algorithm Design”
- Chapter 2, “MATLAB Best Practices and Design Patterns for HDL Code Generation”
- Chapter 3, “Fixed-Point Conversion”
- Chapter 4, “Code Generation”
- Chapter 5, “Verification”
- Chapter 6, “Deployment”
- Chapter 7, “Optimization”
- Chapter 8, “HDL Workflow Advisor Reference”

MATLAB Algorithm Design

- “Data Types and Scope” on page 1-2
- “Operators” on page 1-4
- “Control Flow Statements” on page 1-7
- “Persistent Variables” on page 1-9
- “Persistent Array Variables” on page 1-11
- “Complex Data Type Support” on page 1-12
- “System Objects” on page 1-21
- “Load constants from a MAT-File” on page 1-24
- “Generate Code for User-Defined System Objects” on page 1-25
- “Map Matrices to ROM” on page 1-27
- “Fixed-Point Bitwise Functions” on page 1-28
- “Fixed-Point Run-Time Library Functions” on page 1-35
- “Model State with Persistent Variables and System Objects” on page 1-41
- “Bit Shifting and Bit Rotation” on page 1-45
- “Bit Slicing and Bit Concatenation” on page 1-48
- “Guidelines for Efficient HDL Code” on page 1-50
- “MATLAB Design Requirements for HDL Code Generation” on page 1-51
- “What Is a MATLAB Test Bench?” on page 1-52
- “MATLAB Test Bench Requirements and Best Practices” on page 1-53

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"> • uint8, uint16, uint32, • int8, int16, int32 	
Real	<ul style="list-style-type: none"> • double • single 	HDL code generated with double or single data types can be used for simulation, but is not synthesizable.
Character	char	
Logical	logical	
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"> • unordered {N} • row {1, N} • column {N, 1} 	<p>The maximum number of vector elements allowed is 2^{32}.</p> <p>Before a variable is subscripted, it must be fully defined.</p>

Types	Supported Data Types	Restrictions
Matrices	{N, M}	Matrices are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function. Do not use matrices in the testbench.
Structures	struct	Structures are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function. Do not use structures in the testbench.

Unsupported Data Types

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

- Cell array
- Enumeration
- 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	$A < B$	<code>lt(A,B)</code>
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 1-7.

Relation	Operator Syntax	M Function Equivalent	Notes
Logical Or (short circuiting)	$A \mid B$	N/A	Use short circuiting logical operators within conditionals. See also “Control Flow Statements” on page 1-7.
Element complement	$\sim A$	not(A)	

Control Flow Statements

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

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

The following control flow statements are not supported:

- while
- break
- continue

- return
- parfor

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

To learn how persistent array variables map to RAM, see “Map Persistent Arrays and dsp.Delay to RAM” on page 7-3.

Complex Data Type Support

In this section...

“Declaring Complex Signals” on page 1-12

“Conversion Between Complex and Real Signals” on page 1-13

“Arithmetic Operations on Complex Numbers” on page 1-14

“Support for Vectors of Complex Numbers” on page 1-18

“Other Operations on Complex Numbers” on page 1-19

Declaring Complex Signals

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

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

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

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

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

```

ARCHITECTURE fsm_SFHDH OF complex_decl IS

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

```

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

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

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

Conversion Between Complex and Real Signals

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

```

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

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

```

The 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;
```

Arithmetic Operations on Complex Numbers

When generating HDL code, the coder supports the following arithmetic operators for complex numbers composed of base types (integer, fixed-point, double):

- Addition (+)
- Subtraction (-)
- Multiplication (*)

The coder supports division only for the Fixed-Point Designer™ `divide` function (see `divide`). The `divide` function is supported only if the base type of both complex operands is fixed-point.

As shown in the following example, the default sum and product mode for fixed-point objects is `FullPrecision`, and the `CastBeforeSum` property defaults to `true`.

```
fm = hdlfmath
```

```

fm =

        RoundMode: floor
      OverflowMode: wrap
        ProductMode: FullPrecision
MaxProductWordLength: 128
        SumMode: FullPrecision
MaxSumWordLength: 128
      CastBeforeSum: true

```

Given fixed-point operands, the coder follows full-precision cast before sum semantics. Each addition or subtraction increases the result width by one bit. Further casting is required to bring the results back to a smaller bit width.

In the following example function, two complex operands (with real and imaginary `ufix4` components) are summed, with a complex result having real and imaginary `ufix5` components. The result is then cast back to the original bit width.

```

function z = fcn(x, y)
% addition of two complex numbers x,y of type 'ufix4'

% x+y will have 'ufix5' type
z = x+y;

% to cast the result back to 'ufix4'
% z = fi(x + y, numerictype(x), fimath(x));

```

The following example shows VHDL code generated from this function.

```

ENTITY complex_add_entity IS
  PORT (
    clk : IN std_logic;
    clk_enable : IN std_logic;
    reset : IN std_logic;
    x_re : IN std_logic_vector(3 DOWNTO 0);
    x_im : IN std_logic_vector(3 DOWNTO 0);
    y_re : IN std_logic_vector(3 DOWNTO 0);
    y_im : IN std_logic_vector(3 DOWNTO 0);

```

```

        z_re : OUT std_logic_vector(4 DOWNTO 0);
        z_im : OUT std_logic_vector(4 DOWNTO 0));
END complex_add_entity;

ARCHITECTURE fsm_SFHDH OF complex_add_entity IS

BEGIN
    -- addition of two complex numbers x,y of type 'ufix4'
    -- x+y will have 'ufix5' type
    z_re <= std_logic_vector(resize(unsigned(x_re), 5) +
                            resize(unsigned(y_re), 5));

    z_im <= std_logic_vector(resize(unsigned(x_im), 5) +
                            resize(unsigned(y_im), 5));

    -- to cast the result back to 'ufix4' use
    -- z = fi(x + y, numericitytype(x), fimath(x));

END fsm_SFHDH;

```

Similarly, for the product operation in FullPrecision mode, the result bit width increases to the sum of the lengths of the individual operands. Further casting is required to bring the results back to a smaller bit width.

The following example function shows how the product of two complex operands (with real and imaginary ufix4 components) can be cast back to the original bit width.

```

function z = fcn(x, y)
% Multiplication of two complex numbers x,y of type 'ufix4'

% x*y will have 'ufix8' type
z = x * y;

% to cast the result back to 'ufix4'
% z = fi(x * y, numericitytype(x), fimath(x));

```

The following example shows VHDL code generated from this function.

```

ENTITY complex_mul IS

```

```

PORT (
    clk : IN std_logic;
    clk_enable : IN std_logic;
    reset : IN std_logic;
    x_re : IN std_logic_vector(3 DOWNTO 0);
    x_im : IN std_logic_vector(3 DOWNTO 0);
    y_re : IN std_logic_vector(3 DOWNTO 0);
    y_im : IN std_logic_vector(3 DOWNTO 0);
    z_re : OUT std_logic_vector(8 DOWNTO 0);
    z_im : OUT std_logic_vector(8 DOWNTO 0));
END complex_mul;

ARCHITECTURE fsm_SFHDL OF complex_mul IS

    SIGNAL pr1 : unsigned(7 DOWNTO 0);
    SIGNAL pr2 : unsigned(7 DOWNTO 0);
    SIGNAL pr1in : unsigned(8 DOWNTO 0);
    SIGNAL pr2in : unsigned(8 DOWNTO 0);
    SIGNAL pre : unsigned(8 DOWNTO 0);
    SIGNAL pi1 : unsigned(7 DOWNTO 0);
    SIGNAL pi2 : unsigned(7 DOWNTO 0);
    SIGNAL pi1in : unsigned(8 DOWNTO 0);
    SIGNAL pi2in : unsigned(8 DOWNTO 0);
    SIGNAL pim : unsigned(8 DOWNTO 0);

BEGIN
    -- addition of two complex numbers x,y of type 'ufix4'
    -- x*y will have 'ufix8' type
    pr1 <= unsigned(x_re) * unsigned(y_re);
    pr2 <= unsigned(x_im) * unsigned(y_im);
    pr1in <= resize(pr1, 9);
    pr2in <= resize(pr2, 9);
    pre <= pr1in - pr2in;
    pi1 <= unsigned(x_re) * unsigned(y_im);
    pi2 <= unsigned(x_im) * unsigned(y_re);
    pi1in <= resize(pi1, 9);
    pi2in <= resize(pi2, 9);
    pim <= pi1in + pi2in;
    z_re <= std_logic_vector(pre);

```

```
z_im <= std_logic_vector(pim);  
-- to cast the result back to 'ufix4'  
-- z = fi(x * y, numericity(x), fimath(x));  
END fsm_SFHDL;
```

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;
```

Other Operations on Complex Numbers

The coder supports the following functions with complex operands:

- `complex`
- `real`
- `imag`
- `conj`
- `transpose`
- `ctranspose`
- `isnumeric`
- `isreal`
- `isscalar`

The `isreal` function, which returns 0 for complex numbers, is particularly useful for writing functions that behave differently based on whether the input is a complex or real signal.

```
function y = fcn(u)

% output is same as input if 'u' is real
% output is conjugate of input if 'u' is complex

if isreal(u)
    y = u;
else
    y = conj(u);
end
```

For detailed information on these functions, see “Functions Supported for Code Acceleration or C Code Generation”.

System Objects

In this section...

“Why Use System Objects?” on page 1-21

“Predefined System Objects Supported for HDL Code Generation” on page 1-21

“User-Defined System Objects” on page 1-22

“Limitations of HDL Code Generation for System Objects” on page 1-22

“System object Examples for HDL Code Generation” on page 1-23

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 Supported for HDL Code Generation

The coder supports the following Fixed-Point Designer System object for HDL code generation:

- `hdlram`

The 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`

The coder supports the following DSP System Toolbox™ System objects for HDL code generation:

- `dsp.Delay`
- `dsp.Maximum`
- `dsp.Minimum`
- `dsp.BiquadFilter`
- `dsp.HDLNCO`

User-Defined System Objects

You can create user-defined System objects for HDL code generation. For an example, see “Generate Code for User-Defined System Objects” on page 1-25.

Limitations of HDL Code Generation for System Objects

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

- `step` is the only method supported for HDL code generation.
- Your design can call the `step` method only once per System object.
- `step` must not be inside a loop or a conditional statement.

- System objects must be declared persistent.
- You can use the `dsp.Delay` System object only in feed-forward delay modeling.

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

- Public properties must be nontunable.
- Initial and reset values for properties must be compile-time constant.
- Child objects cannot be user-defined System objects.
- You must use fixed-point data types. Automatic fixed-point conversion is not supported.

System object Examples for HDL Code Generation

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

- “HDL Code Generation from System Objects” on page 4-29
- “Model State with Persistent Variables and System Objects” on page 1-41

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, `sivals.mat`, that contains fixed-point values of `sin` by entering the following commands in MATLAB:

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

You can then generate HDL code from the following MATLAB code, which loads the constants from `sivals.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('sivals.mat');  
end  
sv = pConstStruct.sivals;
```

Generate Code for User-Defined System Objects

This example shows how to generate HDL code for a user-defined System object.

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

```
classdef CounterSysObj < matlab.System

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

    methods (Access=protected)
        function setupImpl(obj, ~)
            % Initialize states
            obj.Count = int32(0);
            obj.State = int32(0);
        end
        function resetImpl(obj)
            % Reset 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
```

```
        end
    end
end
```

The `stepImpl` method implements the System object functionality. The `setupImpl` and `resetImpl` methods define the initial and reset 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;
end

y = step(obj, u);

end
```

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

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

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

To learn more about HDL code generation for user-defined System objects, including design advantages and restrictions, see “System Objects” on page 1-21.

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” on page 11-79. To learn how to set the RAM mapping threshold in MATLAB, see “How To Enable RAM Mapping” on page 7-3.

- Read accesses to the matrix must not be within a feedback loop.

If your MATLAB code meets these requirements, the 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 1-28

“Bitwise Functions Supported for HDL Code Generation” on page 1-28

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 1-28 summarizes the supported functions, with notes that describe considerations specific to HDL code generation. “Bit Slicing and Bit Concatenation” on page 1-48 and “Bit Shifting and Bit Rotation” on page 1-45 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:</p> <p><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, b, c, d, ...)</code>	<p>Concatenate fixed-point operands. 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>

MATLAB Syntax	Description	See Also
<code>bitget(a,idx)</code>	<p>Access a bit at position <code>idx</code>.</p> <p>For VHDL, generates the slice operator: <code>a(idx)</code></p> <p>For Verilog, generates the slice operator: <code>a[idx]</code></p>	<code>bitget</code>
<code>bitor(a, b)</code>	<p>Bitwise OR</p>	<code>bitor</code>
<code>bitorreduce(a, lidx, ridx)</code>	<p>Bitwise OR of a field of consecutive bits within <code>a</code>. The field is delimited by <code>lidx</code> and <code>ridx</code>.</p> <p>Output data type: <code>ufix1</code></p> <p>For VHDL, generates the bitwise OR operator operating on a set of individual slices.</p> <p>For Verilog, generates the reduce operator:</p> <p><code> a[lidx:ridx]</code></p>	<code>bitorreduce</code>
<code>bitset(a, idx, val)</code>	<p>Set or clear bit(s) at position <code>idx</code>.</p> <p>If <code>val = 0</code>, clears the indicated bit(s). Otherwise, sets the indicated bits.</p>	<code>bitset</code>
<code>bitreplicate(a, n)</code>	<p>Concatenate bits of <code>fi</code> object <code>a</code> <code>n</code> times</p>	<code>bitreplicate</code>

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

MATLAB Syntax	Description	See Also
<code>bitshift(a, idx)</code>	<p>Note: For efficient HDL code generation, use <code>bitsll</code>, <code>bitsrl</code>, or <code>bitsra</code> <i>instead</i> of <code>bitshift</code>.</p> <p>Shift left or right, based on the positive or negative integer value of <code>idx</code>.</p> <p><code>idx</code> must be an integer.</p> <p>For positive values of <code>idx</code>, shift left <code>idx</code> bits.</p> <p>For negative values of <code>idx</code>, shift right <code>idx</code> bits.</p> <p>If <code>idx</code> is a variable, generated code contains logic for both left shift and right shift.</p> <p>Result values saturate if the <code>overflowMode</code> of <code>a</code> is set to <code>saturate</code>.</p>	<code>bitshift</code>
<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 $N = \text{ridx} - \text{lidx} + 1$.</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>

MATLAB Syntax	Description	See Also
<code>bitsra(a, idx)</code>	<p>Shift right arithmetic.</p> <p><code>idx</code> must be a scalar within the range</p> $0 \leq \text{idx} < w1$ <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>sra</code> operator in VHDL.</p> <p>Generates <code>>>></code> operator in Verilog.</p>	<code>bitsra</code>
<code>bitsrl(a, idx)</code>	<p>Shift right logical.</p> <p><code>idx</code> must be a scalar within the range</p> $0 \leq \text{idx} < w1$ <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>

MATLAB Syntax	Description	See Also
<code>bitxorreduce(a, lidx, ridx)</code>	<p>Bitwise XOR reduction.</p> <p>Bitwise XOR of a field of consecutive bits within a. The field is delimited by lidx and ridx.</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> <p><code>^a[lidx:ridx]</code></p>	<code>bitxorreduce</code>
<code>getlsb(a)</code>	Return value of LSB.	<code>getlsb</code>
<code>getmsb(a)</code>	Return value of MSB.	<code>getmsb</code>

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 1-39 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

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

Function	Restrictions
imag	None
int8, int16, int32	None
iscolumn	None
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

Function	Restrictions
minus	None
mpower	Both inputs must be scalar, and the exponent input, k, must be a constant integer.
mtimes	None
ndims	None
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
reinterprecast	None
repmat	None
rescale	None
reshape	None
round	None
sfi	None
sign	None
size	None
sqrt	None
sub	None

Function	Restrictions
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
ufi	None
uint8, uint16, uint32	None
uminus	None
uplus	Inputs cannot be data type logical.
upperbound	None
vertcat	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.m', ...  
    'mlhdlc_sysobj_sobel_tb.m', ...  
    'mlhdlc_sobel.m', ...  
    'mlhdlc_sobel_tb.m'  
};
```

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', 'hdlcoderdemo');  
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}), 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', 'hdlcoderdemo');
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
```

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
```

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. The 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.

The 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 1-53.

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.* The 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

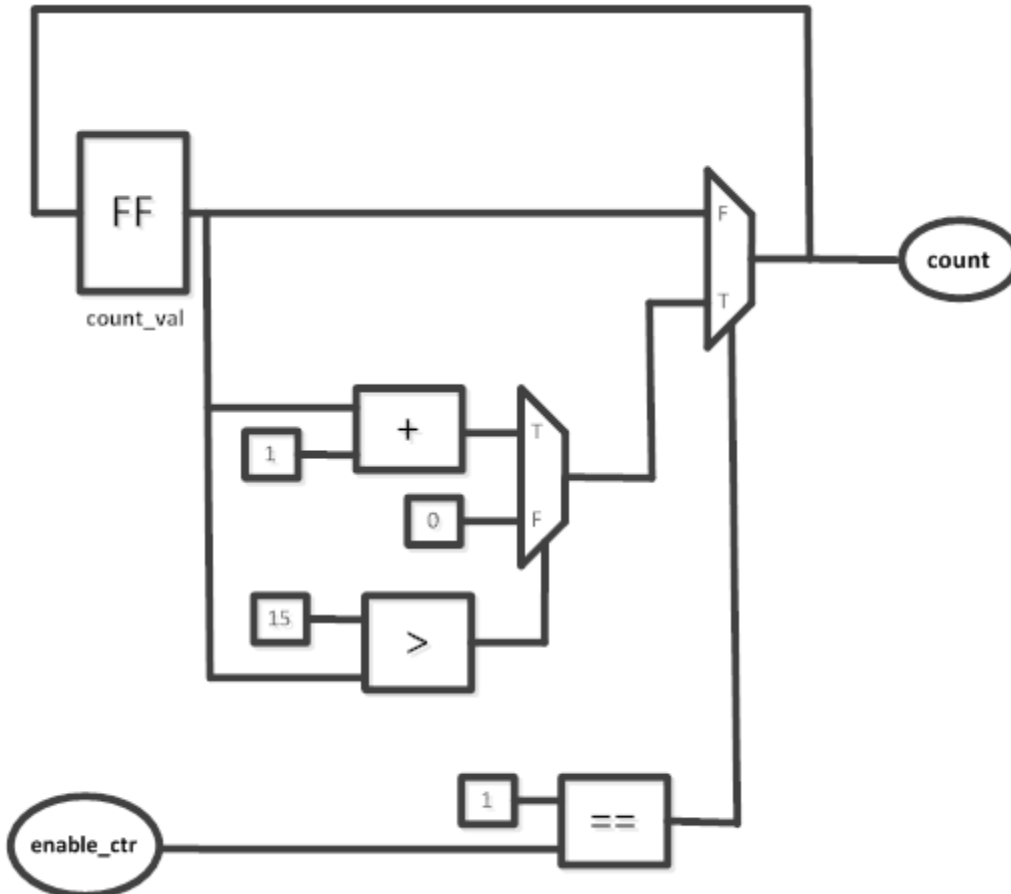
MATLAB Counter

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

```

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 `m1hdlc_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 `m1hdlc_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” on page 2-13

“Implement RAM Using `hdlram`” on page 2-14

Implementation of RAM

These examples demonstrate two methods of writing MATLAB code mapped to RAM during HDL code generation: persistent arrays and `hdlram` System objects. The examples model the same line delay in MATLAB. However, one example uses the a persistent array and the other uses an `hdlram` System object to model the RAM behavior. This line delay uses memory in a ring structure. Data is written in one position and read from another position in such a way that the data written is read after a specific number of cycles. To parameterize the delay length, the RAM write address is generated by a counter. The read address is generated by adding a constant value to the write 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 trade-offs, see: “RAM Mapping Comparison for MATLAB Code” on page 7-9. For information on mapping persistent arrays and `dsp.Delay` System objects to RAM, see: “Map Persistent Arrays and `dsp.Delay` to RAM” on page 7-3.

Implement RAM Using a Persistent Array

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.

```
##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 hd1ram

This example shows a line delay that implements the RAM behavior using `hd1ram` with the function, `mlhdlc_hd1ram_sysobj`. In this function, the `step` method of the `hd1ram` System object reads and writes to specific locations in `hRam`.

```
    %#codegen
    function data_out = mlhdlc_hd1ram_sysobj(data_in)
    persistent hRam;
    if isempty(hRam)
        hRam = hd1ram('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;
```

hdlram Restrictions for Code Generation

Code generation from hdlram has the same restrictions as code generation from other System objects. For details, see “Limitations of HDL Code Generation for System Objects” on page 1-22.

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 2-18

MATLAB Loops

Some best practices for using loops in MATLAB code for HDL code generation are:

- Use monotonically increasing loop counters, with increments of 1, to minimize the amount of hardware generated in the HDL code.
- If you want to use the loop streaming optimization:
 - When assigning new values to persistent variables inside a loop, do not use other persistent variables on the right side of the assignment. Instead, use an intermediate variable.
 - If a loop modifies any elements in a persistent array, the loop should modify all of the elements in the persistent array.

Monotonically Increasing Loop Counters

By using monotonically increasing loop counters with increments of 1, you can reduce the amount of hardware in the generated HDL code. The following loop is an example of a monotonically increasing loop counter with increments of 1.

```
a=1;  
for i=1:10  
    a=a+1;  
end
```

If a loop counter increases by an increment other than 1, the generated HDL code can require additional adders. Due to this additional hardware, do not use the following type of loop.

```

a=1;
for i=1:2:10
    a=a+1;
end

```

If a loop counter decreases, the generated HDL code can require additional adders. Due to this additional hardware, do not use the following type of loop.

```

a=1;
for i=10:-1:1
    a=a+1;
end

```

Persistent Variables in Loops

If a loop contains multiple persistent variables, when you assign values to persistent variables, use intermediate variables that are not persistent on the right side of the assignment. This practice makes dependencies clear to the compiler and assists internal optimizations during the HDL code generation process. If you want to use the loop streaming optimization to reduce the amount of generated hardware, this practice is recommended.

In the following example, `var1` and `var2` are persistent variables. `var1` is used on the right side of the assignment. Because a persistent variable is on the right side of an assignment, do not use this type of loop:

```

for i=1:10
    var1 = 1 + i;
    var2 = var1 * 2;
end

```

Instead of using `var1` on the right side of the assignment, use an intermediate variable that is not persistent. This example demonstrates this with the intermediate variable `var_intermediate`.

```

for i=1:10
    var_intermediate = 1 + i;
    var1 = var_intermediate;
    var2 = var_intermediate * 2;
end

```

Persistent Arrays in Loops

If a loop modifies elements in a persistent array, make sure that the loop modifies all of the elements in the persistent array. If all elements of the persistent array are not modified within the loop, the 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 3-18
- “Working with Generated Fixed-Point Files” on page 3-29
- “Specify Type Proposal Options” on page 3-37
- “Log Data for Histogram” on page 3-40
- “View and Modify Variable Information” on page 3-43
- “Automated Fixed-Point Conversion” on page 3-47

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.m';  
testbench_name = 'mlhdlc_df2t_filter_tb.m';
```

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', 'hdlcoderdemo');  
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fix_prj'];
```

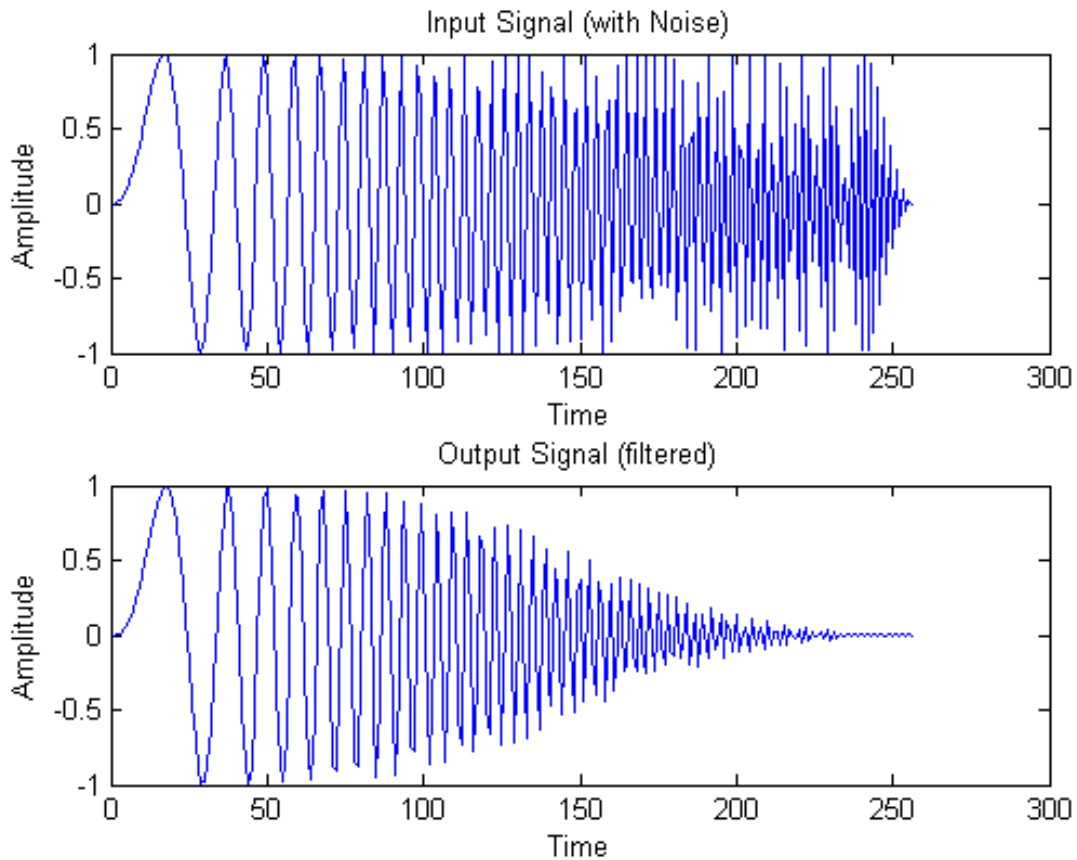
```
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);
```

```
copyfile(fullfile(mlhdlc_demo_dir, design_name), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), mlhdlc_temp_dir);
```

Simulate the Design

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

```
mlhdlc_df2t_filter_tb
```



Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project
```

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

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

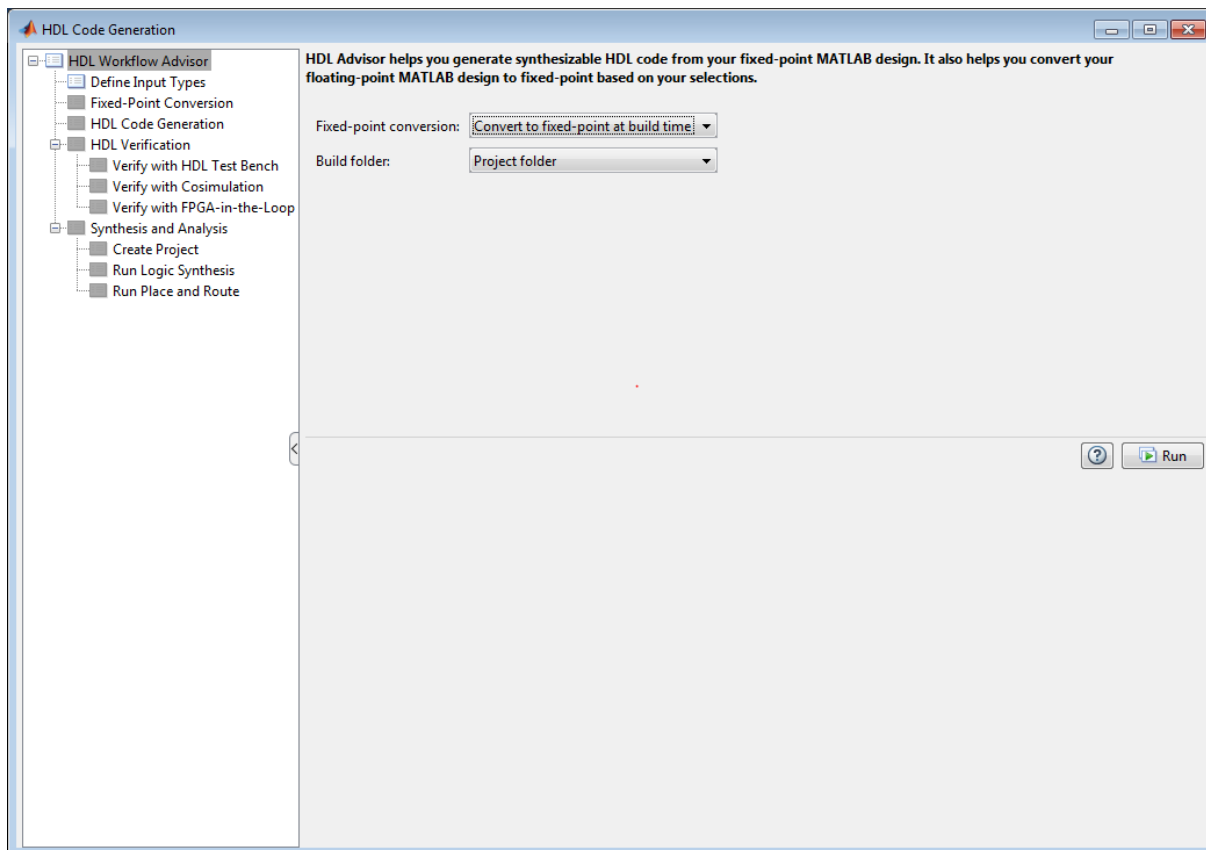
Fixed-Point Code Generation Workflow

The floating-point to fixed-point conversion workflow allows you to:

- Verify that the floating-point design is code generation compliant
- Propose fixed-point types based on simulation data and word length settings
- Allow the user to manually adjust the proposed fixed-point types
- Validate the proposed fixed-point types
- Verify that the generated fixed-point MATLAB code has the desired numeric accuracy

Step 1: Launch Workflow Advisor

- 1** Click on the Workflow Advisor button to launch the HDL Workflow Advisor.
- 2** Choose 'Convert to fixed-point at build time' for the option 'Fixed-point conversion'.



Step 2: Define Input Types

In this step you can define input types manually or by specifying and running the testbench.

- 1 Click 'Run' to execute this step.

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

Step 3: Run Simulation

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

The design is compiled with the input types defined in the previous step and after the compilation is successful the variable table shows inferred types for all the functions in the design.

In this step, the original design is instrumented so that the minimum and maximum values for all variables in the design are collected during simulation.

The screenshot shows the HDL Code Generation window. On the left is the HDL Workflow Advisor tree with 'Fixed-Point Conversion' selected. The main area shows a code editor with the following code:

```

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
    
```

Below the code editor is a table with two tabs: 'Variables' and 'Function Replacements'. The 'Variables' tab is active, showing the following data:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
Input							
x	double					No	
Output							
y	double					No	
Persistent							
z	2x1 double					No	
Local							
a	1x3 double					No	
b	1x3 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 a tree view of the HDL Workflow Advisor. The main area displays MATLAB code for a filter function. Below the code is a table showing simulation output for various variables.

```

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.

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 shows the HDL Code Generation tool interface. The main window displays MATLAB code for a function named `mlhdlc_df2t_filter`. Below the code, there is a table showing the proposed fixed-point types for variables. A simulation output window is also visible, showing a histogram of values for a variable.

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

Simulation Output Window:

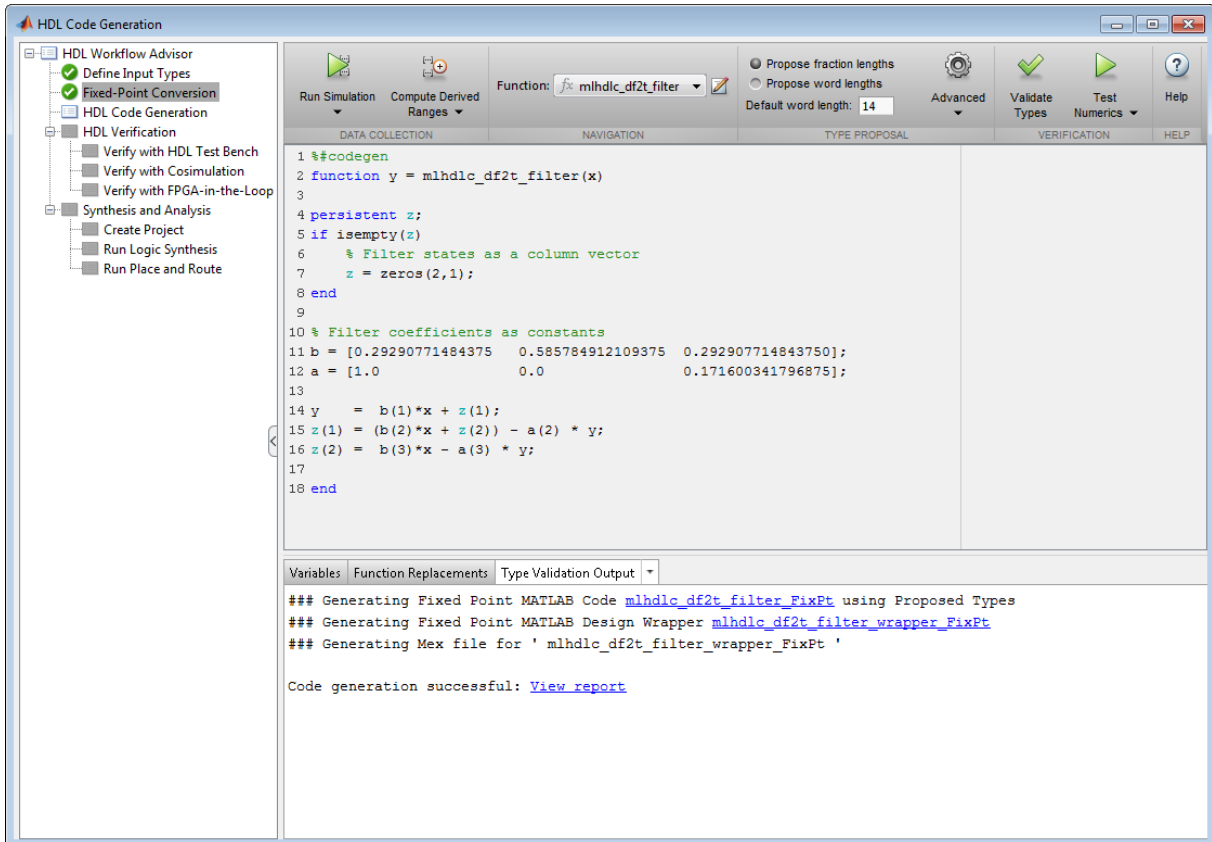
Sim values covered 97%
Supported range -2 : 1.9998

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 3: 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 image shows a MATLAB Editor window with two files open. The top file is `mlhdlc_df2t_filter.m`, which contains the original floating-point implementation of a discrete-time filter. The bottom file is `mlhdlc_df2t_filter_FixPt.m`, which is the fixed-point version generated by MATLAB HDL Coder. The fixed-point version uses `hdlfimath` for arithmetic and `fi` for fixed-point data types.

```

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

```

```

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

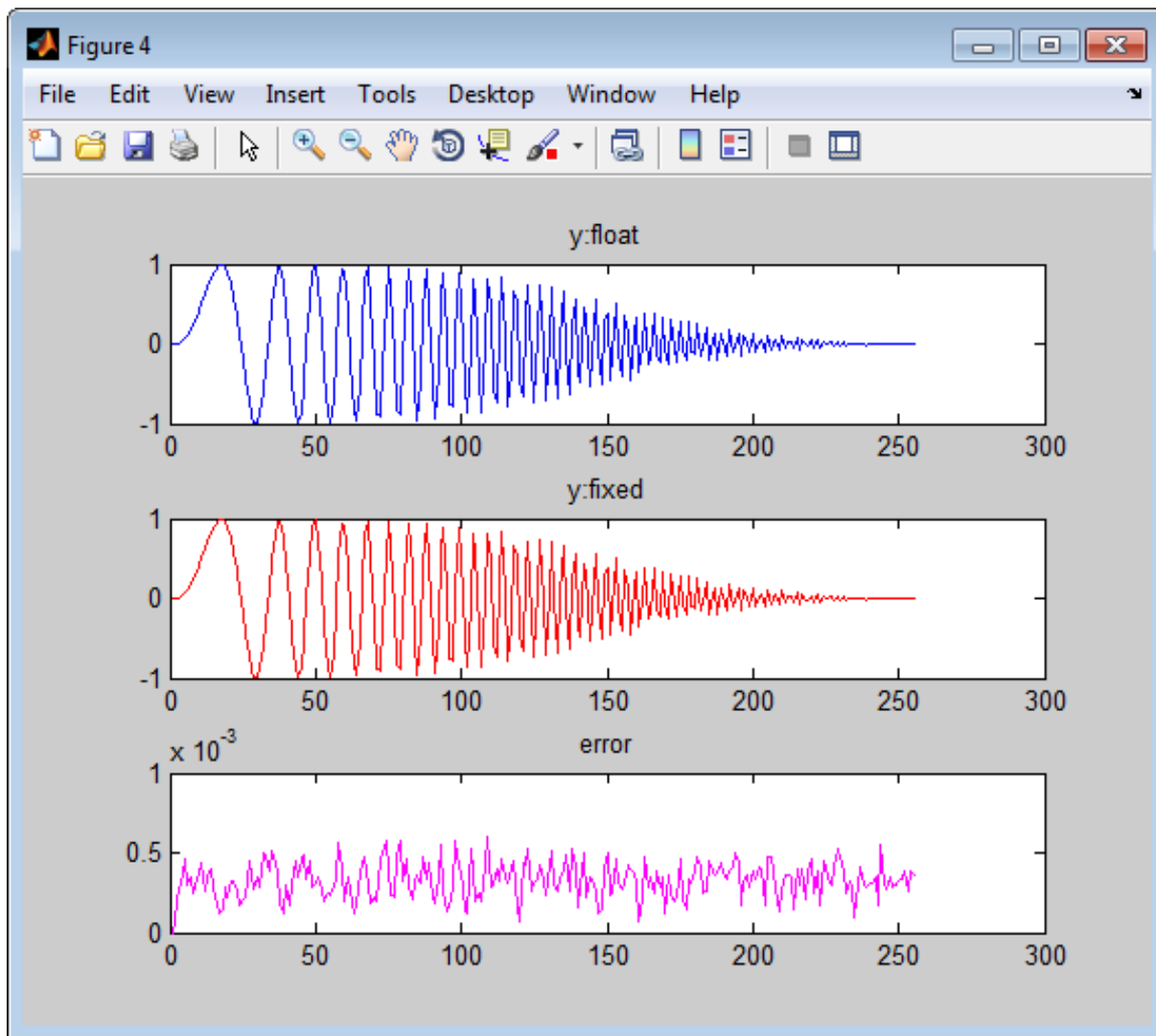
```

Step 4: 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 5: 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', 'hdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fixprj'];
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 a Kalman filter suitable for C 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.

```
design_name = 'mlhdlc_kalman_c.m';  
testbench_name = 'mlhdlc_kalman_c_tb.m';
```

- 1 MATLAB Design: mlhdlc_kalman_c
- 2 MATLAB testbench: mlhdlc_kalman_c_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', 'hdlcoderdemo
```



```
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), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), mlhdlc_temp_dir);
```

Simulate the Design

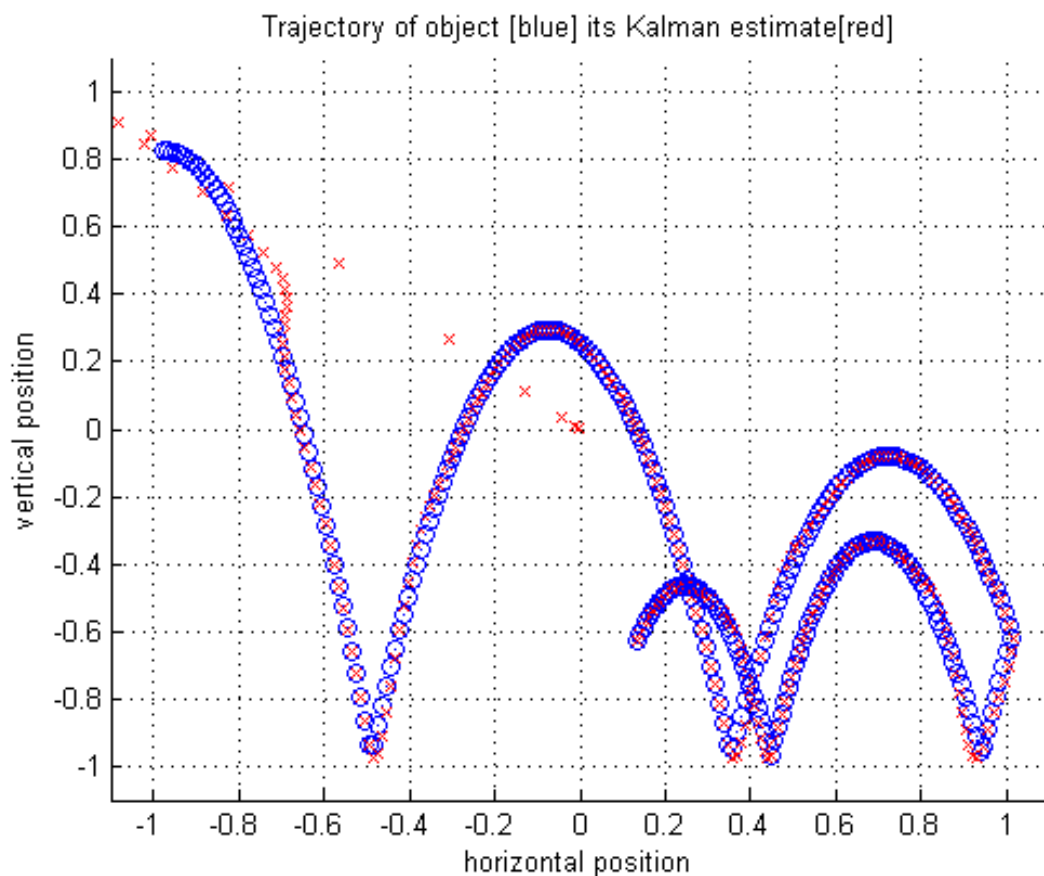
Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_kalman_c_tb
```

```
Running -----> mlhdlc_kalman_c_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_c.m' to the project as the MATLAB Function and 'mlhdlc_kalman_c_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 14 and run the 'Propose Fixed-Point Types' step.

The screenshot shows the HDL Code Generation tool interface. The 'Fixed-Point Conversion' step is selected in the workflow advisor. The 'Propose fraction lengths' and 'Propose word lengths' options are visible, with the default word length set to 14. A table at the bottom shows the proposed fixed-point types for variables z, y1, y2, and p_est.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
Input							
z	2 x 1 double	-0.98	1.01			No	numerictype(1, 14, 12)
Output							
y1	double	-1.14	1.01			No	numerictype(1, 14, 12)
y2	double	-0.98	0.98			No	numerictype(1, 14, 12)
Persistent							
p_est	6 x 6 double		472.78			No	numerictype(0, 14, 5)

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 14 bits.

Variables		Function Replacements		Simulation Output			
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
Input							
z	2 x 1 double	-0.98	1.01			No	numerictype(1, 14, 12)
Output							
y1	double	-1.14	1.01			No	numerictype(1, 14, 12)
y2	double	-0.98	0.98			No	numerictype(1, 14, 12)
Persistent							
p_est	6 x 6 double	0	472.78			No	numerictype(0, 14, 5)
x_est	6 x 1 double	-1.14	1.01			No	numerictype(1, 14, 12)
Local							
A	6 x 6 double	0	1			Yes	numerictype(0, 1, 0)
B	2 x 6 double	0	896.74			No	numerictype(0, 14, 4)
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, 11, 0)
S	2 x 2 double	0	1896.74			No	numerictype(0, 14, 3)
dt	double	1	1			Yes	numerictype(0, 1, 0)
klm_gain	6 x 2 double	0	0.47			No	numerictype(0, 14, 15)
p_prd	6 x 6 double	0	896.74			No	numerictype(0, 14, 4)
x_prd	6 x 1 double	-1.35	1.17			No	numerictype(1, 14, 12)
y	2 x 1 double	-1.14	1.01			No	numerictype(1, 14, 12)
z_prd	2 x 1 double	-1.35	1.17			No	numerictype(1, 14, 12)

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(0,14,4) (Signedness: Unsigned, WordLength: 14, FractionLength: 4)

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:

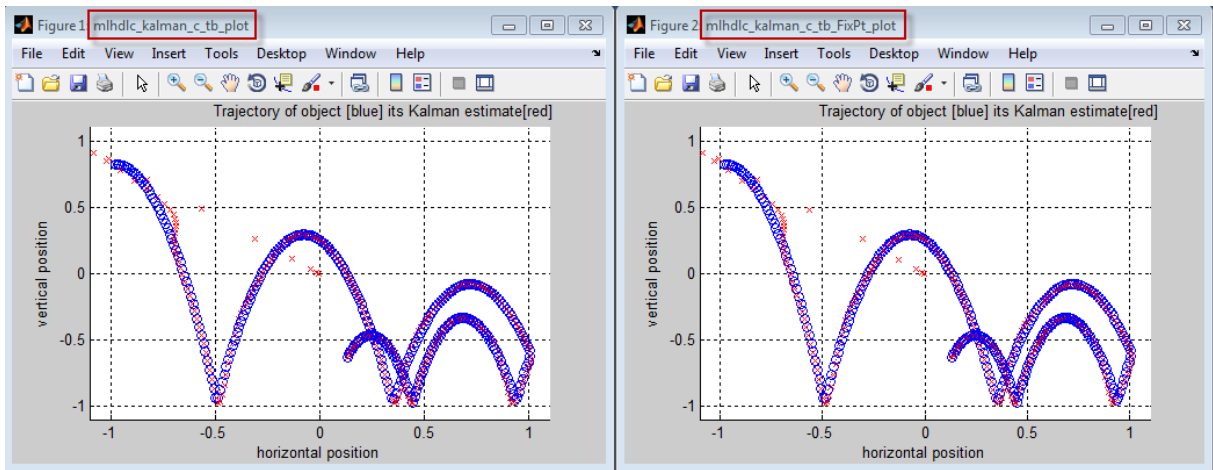
- Has the minimum number of bits to represent the whole integer range.
- Has no fractional bits.

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

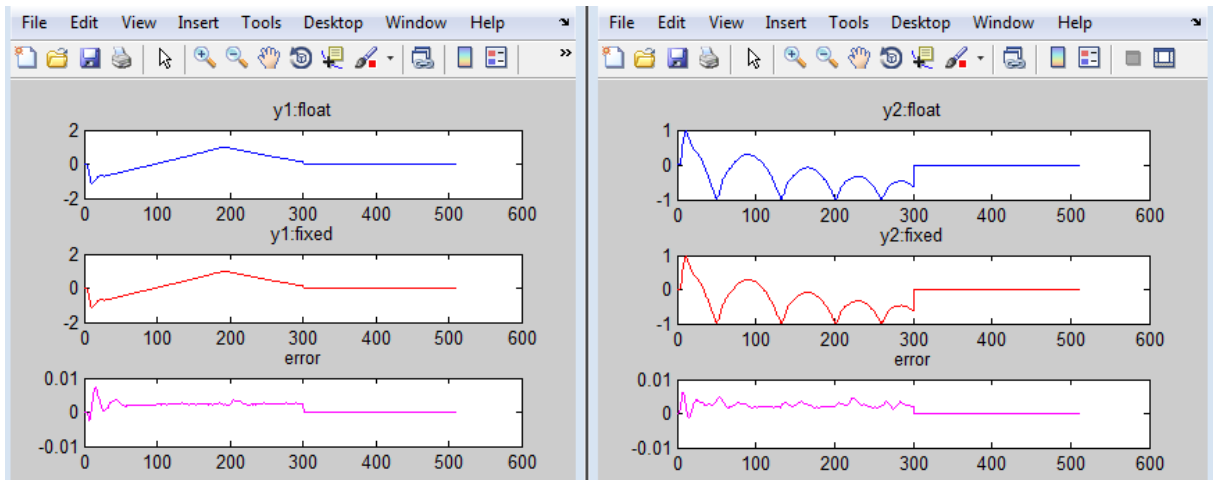
Based on the numeric types proposed for a default word length of 14, continue with fixed-point code generation and verification steps and observe the plots.

- 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.



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 14 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.

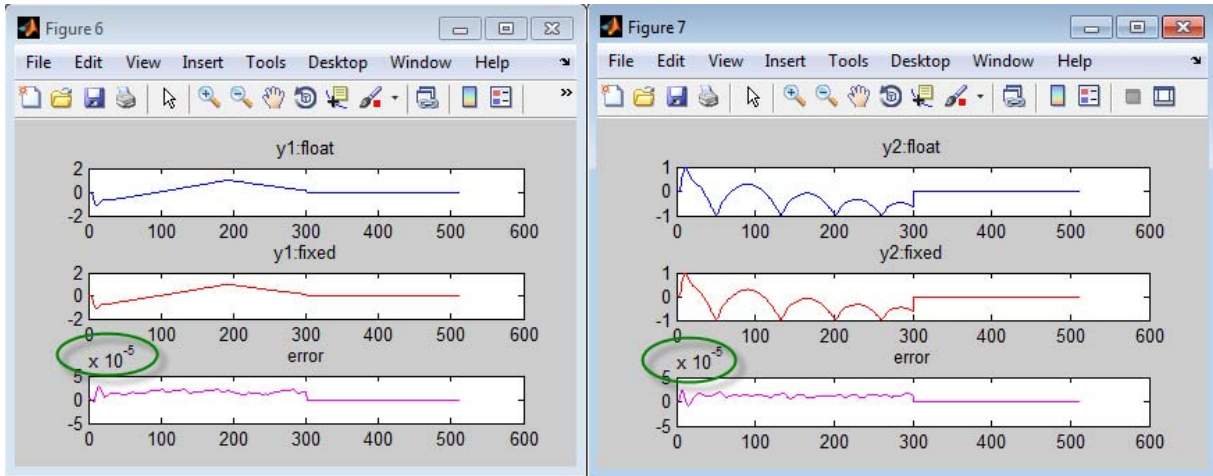
Let us now increase the default word length to 22 bits and repeat the type proposal and validation steps.

- 1** Select a 'Default word length' of 22.

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', 'hdlcoderdemo');
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.m';  
testbench_name = 'mlhdlc_filter_tb.m';
```

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', 'hdlcoderdemo');  
mlhdlc_temp_dir = [tempdir 'mlhdlcflt2fix'];
```

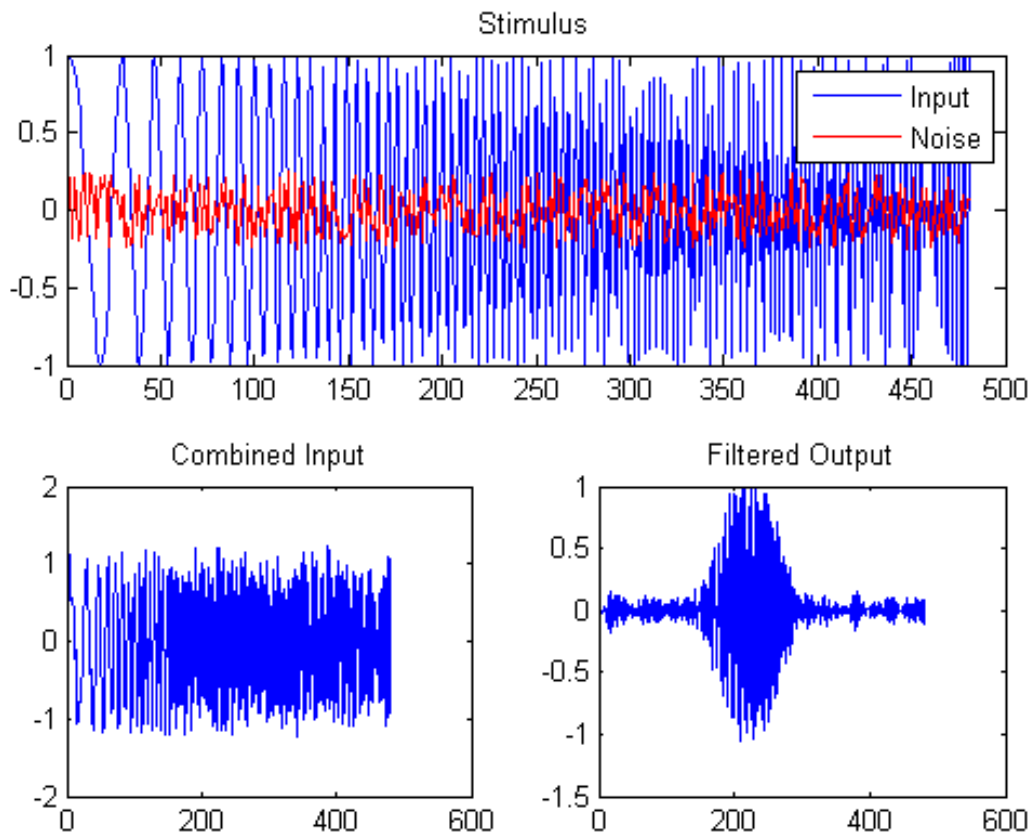
```
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);
```

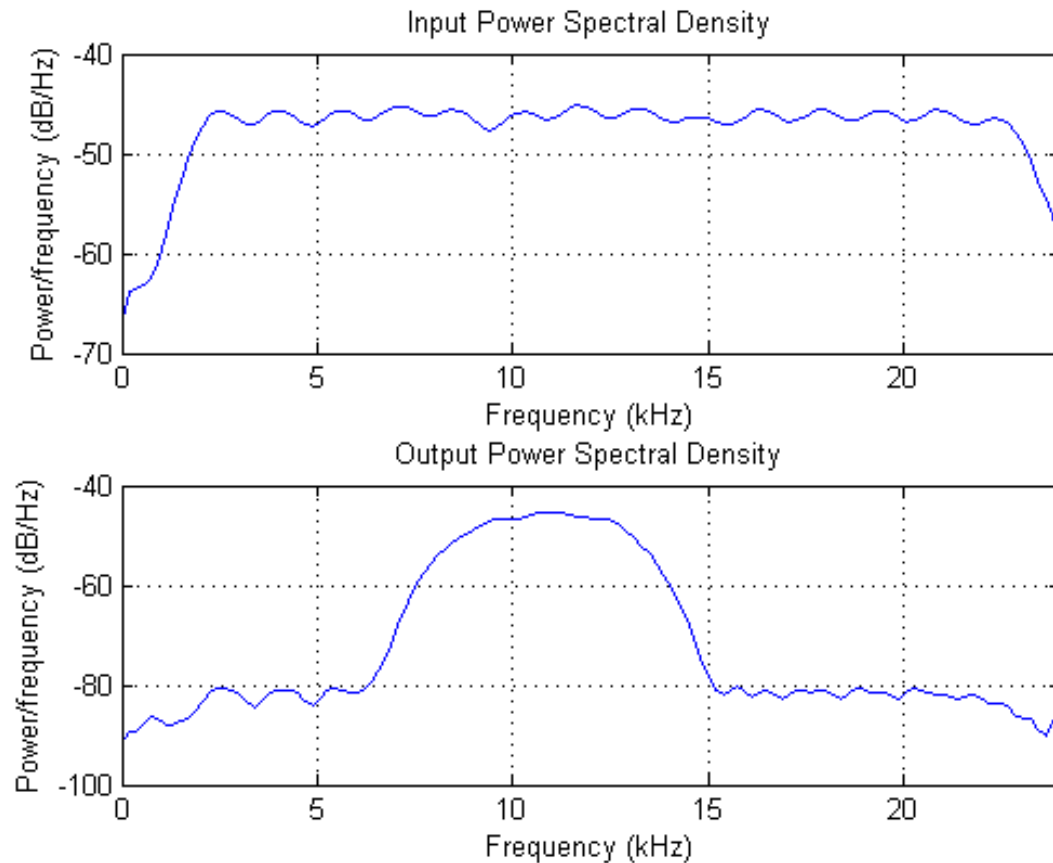
```
copyfile(fullfile(mlhdlc_demo_dir, design_name), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), mlhdlc_temp_dir);
```

Simulate the Design

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

```
m1hdlc_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

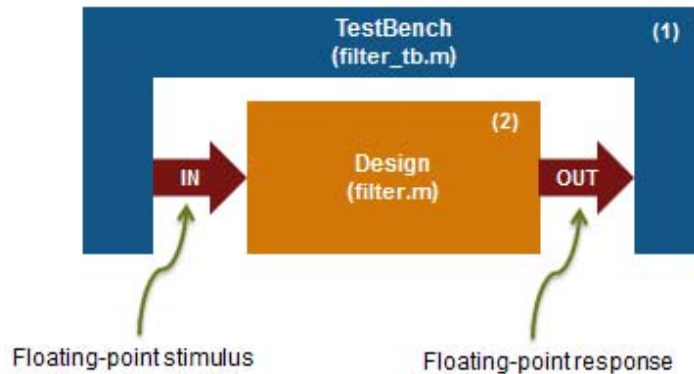
Perform the following tasks in preparation for the fixed-point code generation step:

- 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.



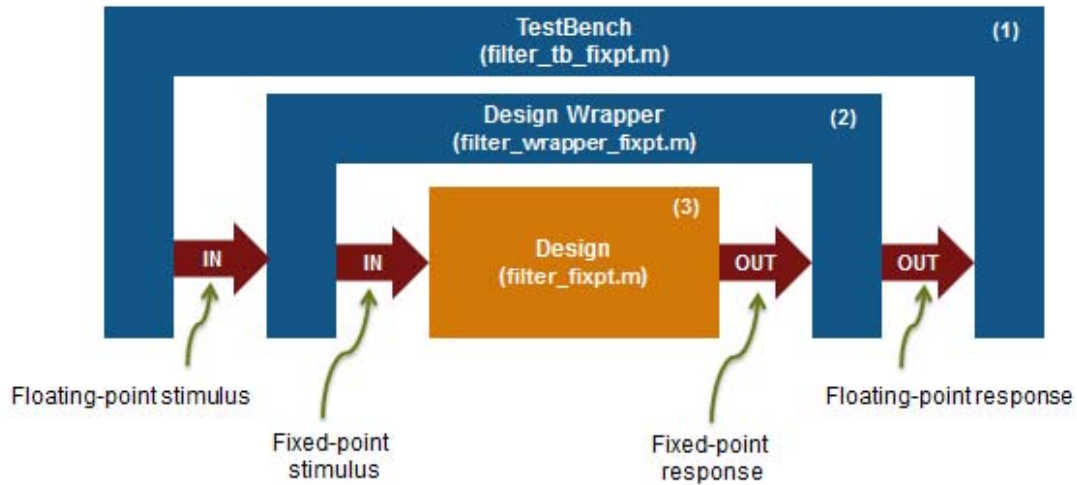
For floating-point to fixed-point conversion, the following requirements apply to the original design and the testbench:

- 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

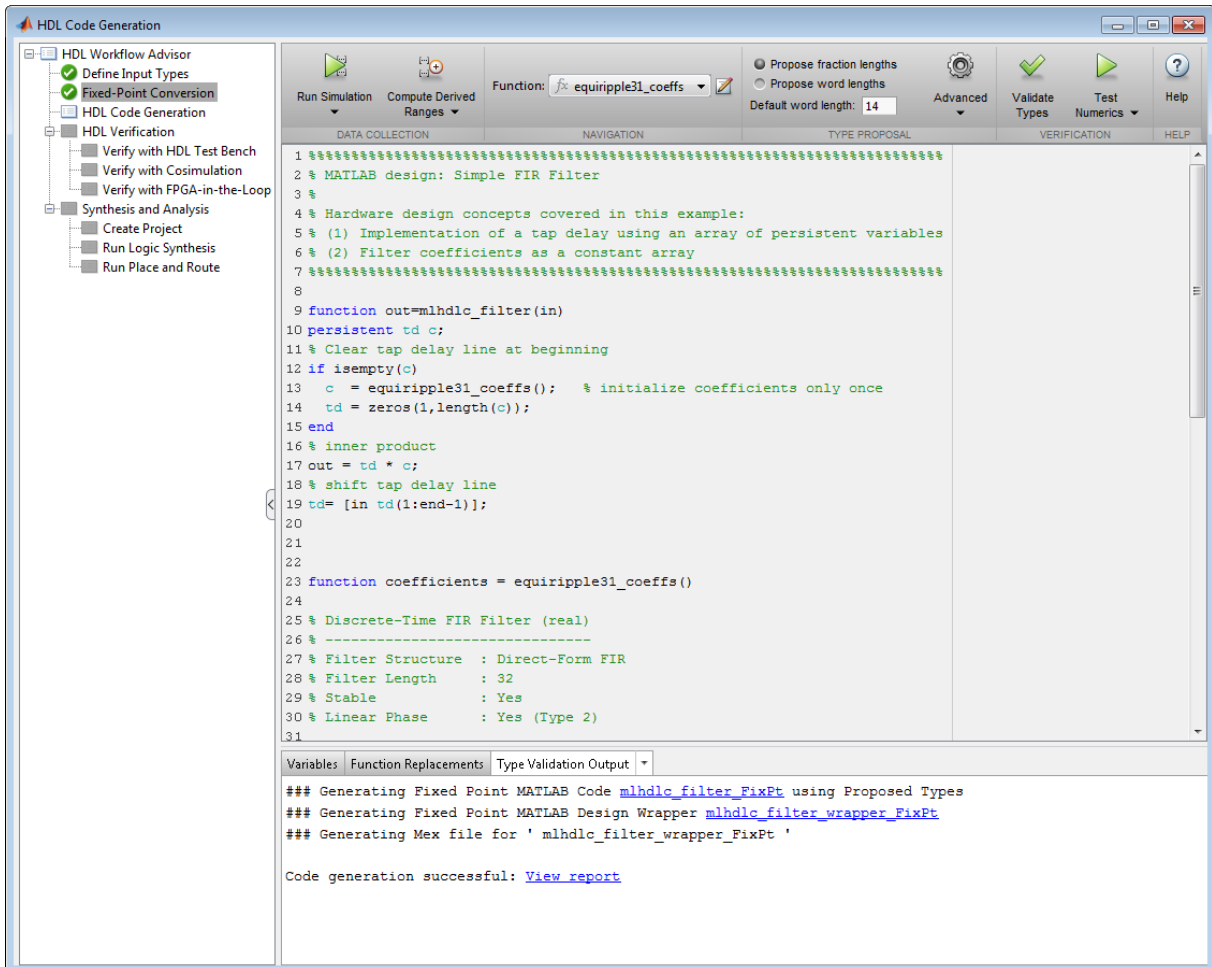
During the type validation step, fixed-point code is generated for this design and compiled to verify that there are no errors when applying the types. The output files will have the following structure.



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'.



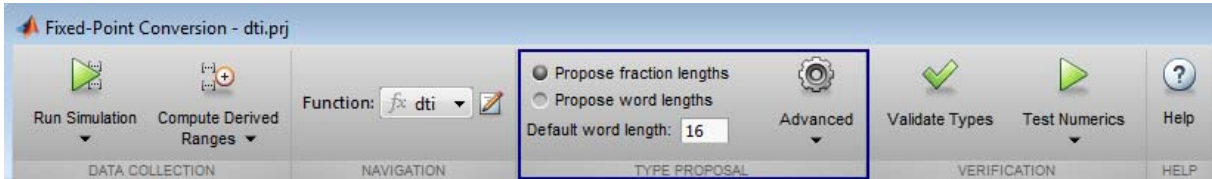
Click the links to the generated code in the Workflow Advisor log window to examine the generated fixed-point design, wrapper, and test bench.

Clean up the Generated Files

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

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

Specify Type Proposal Options



You specify whether to propose fraction lengths or word lengths in the Fixed-Point Conversion window **Type Proposal** options. By default, the software proposes fraction lengths for a default word length of 16.

To customize fixed-point type proposals, use the **Advanced** settings.

Advanced Setting	Values	Description
When proposing types	ignore simulation ranges	Propose data types based only on derived ranges
	ignore derived ranges	Propose data types based only on simulation ranges
	use all collected data (default)	Proposed data types based on both simulation and derived ranges
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.

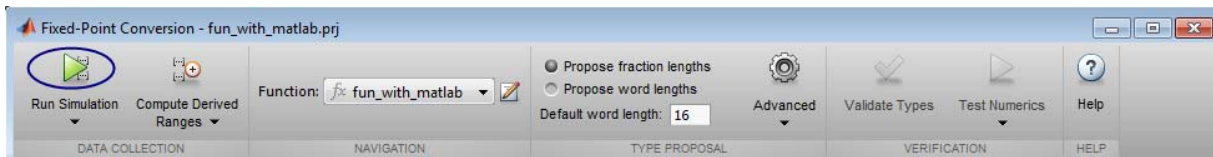
Advanced Setting	Values	Description
Safety margin for sim min/max (%)	0 (default)	<p>Specify safety factor for simulation minimum and maximum values.</p> <p>The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different 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.</p>
Generated fixed-point file name suffix	_fixpt (default)	<p>Specify the suffix to add to the generated fixed-point file names. For example, by default, if you generate a static library for a project named test, the generated files are in the subfolder codegen\lib\test_fixpt. The generated static library is named test.lib, but the generated C code files use the suffix, for example, test_fixpt.c.</p>
Transform for-loop index variables	No (default)	
	Yes	

Advanced Setting		Values	Description
fimath	Rounding method	Ceiling	Specify the <code>fimath</code> properties for the generated fixed-point data types. The default fixed-point math properties use the <code>Floor</code> rounding and <code>Wrap</code> overflow because they are the default actions in C. These settings generate the most efficient code but might cause problems with overflow.
		Convergent	
		Floor (default)	
		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			

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.

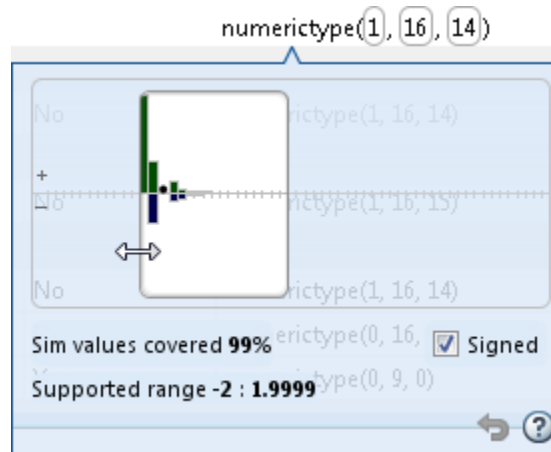


The simulation runs and the simulation minimum and maximum ranges are displayed on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.

- 2 To view a histogram for a variable, click the variable's **Proposed Type** field.

The screenshot shows a software interface with two tabs: "Whole Number" and "Proposed Type". The "Proposed Type" tab is selected, displaying "No" and the configuration `numerictype(1, 16, 14)`. A mouse cursor is hovering over the `14` parameter. A callout window is open, showing a histogram with a white bounding box around the data. Below the histogram, it displays "Sim values covered 99%" and "Supported range -2 : 1.9999". There is a "Signed" checkbox which is checked. At the bottom right of the callout, there are icons for "enable feature" and a help icon.

- 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

To view information about the variables in the MATLAB function selected in the **Navigation** pane, use the **Variables** tab or place your cursor over a variable in the code window. For more information, see “Viewing Variables” on page 3-55.

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 variables in the MATLAB code pane and on the **Variables** tab, use **Ctrl+F**. The tool highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

Modify Variable Information

If you modify variable information, the tool highlights the values in bold. 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 3-46.
Editing this field does not trigger static range analysis, but the tool 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 3-46.

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

- **Whole Number**

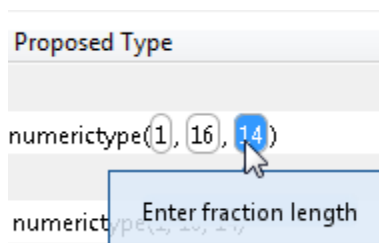
The Fixed-Point Conversion tool uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

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

- **Proposed Type**

You can modify the signedness, word length, and fraction length settings individually by:

- On the **Variables** tab, by modifying the value in the **ProposedType** field.



- In the code window, by selecting a variable and then modifying the **ProposedType** field.

(x)

Original Type: 1 x 256 double

Sim Range: -1 : 1

Static Range: :

Proposed Type: numerictype(1, 16, 14)

Rounding Method: Floor

Overflow Action: Wrap

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**.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1 x 256 double	-1	1			No	numerictype(1, 16, 12)
Output							
y	1 x 256 double	-0.97	1.06				numerictype(1, 16, 14)
Persistent							
z	2 x 1 double	-0.89	0.96				numerictype(1, 16, 15)
Local							
a	1 x 3 double	-0.94	1				numerictype(1, 16, 14)
b	1 x 3 double	0.1	0.2			No	numerictype(0, 16, 18)
i	double	1	256			Yes	numerictype(0, 9, 0)

- Copy sim ranges for all top-level inputs
- Copy sim ranges for all persistent variables
- Clear all manually entered static ranges
- Reset entire table**

- To revert the type of a selected variable to the type computed by the tool, 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 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

The Fixed-Point Conversion tool allows you to 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.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max
Input					
x	1 x 256 double	-1	1		
Output					
y	1 x 256 double	-0.97	1.06		

Copy sim ranges for all top-level inputs
 Copy sim ranges for all persistent variables
 Clear all manually entered static ranges
 Reset entire table

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 inputs**.

Automated Fixed-Point Conversion

In this section...

“License Requirements” on page 3-47

“Fixed-Point Conversion Capabilities” on page 3-47

“Code Coverage” on page 3-49

“Proposing Data Types” on page 3-53

“Viewing Functions” on page 3-55

“Viewing Variables” on page 3-55

“Histogram” on page 3-57

“Function Replacements” on page 3-58

“Validating Types” on page 3-58

“Testing Numerics” on page 3-59

License Requirements

Fixed-point conversion requires the following licenses:

- Fixed-Point Designer
- MATLAB Coder™

Fixed-Point Conversion Capabilities

You can convert floating-point MATLAB code to fixed-point code using the Fixed-Point Conversion tool in MATLAB CoderHDL Coder projects. You can choose to propose data types based on simulation range data, derived (also known as static) range data, or both.

The screenshot shows the 'Fixed-Point Conversion' tool window for a project named 'fun_with_matlab.prj'. The interface includes a toolbar with options like 'Run Simulation', 'Compute Derived Ranges', 'Propose fraction lengths', 'Propose word lengths', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. The 'Function' dropdown is set to 'fun_with_matlab' and the 'Default word length' is 16. A yellow banner at the top of the code editor reads: 'To compute proposed fixed-point types for variables, use Run Simulation, Compute Derived Ranges, or both.'

```

1 function y = fun_with_matlab(x) %#codegen
2 persistent z
3 if isempty(z)
4     z = zeros(2,1);
5 end
6 % [b,a] = butter(2, 0.25)
7 b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
8 a = [1, -0.942809041582063, 0.333333333333333];
9
10
11 y = zeros(size(x));
12 for i=1:length(x)
13     y(i) = b(1)*x(i) + z(1);
14     z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
15     z(2) = b(3)*x(i) - a(3) * y(i);
16 end
17 end
    
```

Below the code editor is a table with two tabs: 'Variables' and 'Function Replacements'. The 'Variables' tab is active, showing a table of variable types and proposed fixed-point types.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
▲ Input							
x	1x256 double					No	
▲ Output							
y	1x256 double					No	
▲ Persistent							
z	2x1 double					No	
▲ Local							
a	1x3 double					No	
b	1x3 double					No	
i	double					No	

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 filetest bench with the fixed-point types applied.
- View a histogram of bits used by each variable.

Fixed-Point Conversion Limitations

Fixed-point conversion does not support MATLAB classes.

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 Fixed-Point Designer interface. At the top, there is a toolbar with icons for 'Run Simulation', 'Compute Derived Ranges', 'Function: fsm_mealy', 'Propose fraction lengths', 'Propose word lengths', 'Default word length: 16', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. Below the toolbar, the code editor shows the following MATLAB code:

```

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 code is annotated with a vertical coverage bar on the left side. The bar is colored red for lines 16-18, green for lines 20-32, and blue for lines 34-52. The red bar indicates that the code in these lines is never executed. The green bar indicates that the code in these lines is executed, but the percentage of execution is not shown. The blue bar indicates that the code in these lines is executed, and the percentage of execution is shown as a percentage of the total code execution.

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 code coverage tool interface. The top toolbar includes buttons for 'Run Simulation', 'Compute Derived Ranges', 'Function: fsm_mealy', 'Propose fraction lengths', 'Propose word lengths', 'Default word length: 16', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. Below the toolbar, the code is displayed with coverage bars on the right side of each line. 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     case S2,
34
35         if (A)
36             Z = false;
37             current_state = S1;
38         else
39             Z = true;
40             current_state = S2;
41         end
42     case S3,
43
44         if (A)
45             Z = false;
46             current_state = S2;
47         else
48             Z = true;
49             current_state = S3;
50         end
51     end
52 end
53

```

The coverage results are as follows:

- Line 16: Executed once (Orange)
- Line 17: Reached 58% of the time (Green)
- Line 18: Reached 58% of the time (Green)
- Line 19: Reached 58% of the time (Green)
- Line 20: Reached 58% of the time (Green)
- Line 21: Reached 58% of the time (Green)
- Line 22: Reached 58% of the time (Green)
- Line 23: Reached 58% of the time (Green)
- Line 24: Reached 58% of the time (Green)
- Line 25: Reached 58% of the time (Green)
- Line 26: Reached 33% of the time (Green)
- Line 27: Reached 25% of the time (Green)
- Line 28: Reached 25% of the time (Green)
- Line 29: Reached 25% of the time (Green)
- Line 30: Reached 25% of the time (Green)
- Line 31: Reached 25% of the time (Green)
- Line 32: Reached 25% of the time (Green)
- Line 33: Reached 58% of the time (Green)
- Line 34: Reached 42% of the time (Green)
- Line 35: Reached 42% of the time (Green)
- Line 36: Reached 17% of the time (Green)
- Line 37: Reached 17% of the time (Green)
- Line 38: Reached 17% of the time (Green)
- Line 39: Reached 25% of the time (Green)
- Line 40: Reached 25% of the time (Green)
- Line 41: Reached 25% of the time (Green)
- Line 42: Reached 25% of the time (Green)
- Line 43: Reached 42% of the time (Green)
- Line 44: Not reached (Red)
- Line 45: Not reached (Red)
- Line 46: Not reached (Red)
- Line 47: Not reached (Red)
- Line 48: Not reached (Red)
- Line 49: Not reached (Red)
- Line 50: Not reached (Red)
- Line 51: Not reached (Red)
- Line 52: Not reached (Red)
- Line 53: Not reached (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.
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.

Running a Simulation

When you open the Fixed-Point Conversion tool, it generates an instrumented MEX function for your entry-point MATLAB file. 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 3-58.

Before running a simulation, specify the test file or filestest bench that you want to run. When you run a simulation, the tool runs the test filetest bench, calling the instrumented MEX function. If you modify the MATLAB design code, the tool automatically generates an updated MEX function before running a test filethe test bench.

If the test filetest bench runs successfully, the simulation minimum and maximum values and the proposed types are displayed on the **Variables** tab. If the test filetest bench fails, the errors are displayed on the **Simulation Output** tab.

Test filesThe 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 filetest bench covers the operating range of the algorithm with the desired accuracy. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the conversion tool merges the simulation results.

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 “Log Data for Histogram”“Log Data for Histogram” on page 3-40.

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 for all input variables. To improve the analysis, enter as much static range information as possible for other variables. You can promote simulation ranges to use as static ranges. Alternatively, if you know what data type your hardware target uses, set the proposed data type to match this type.

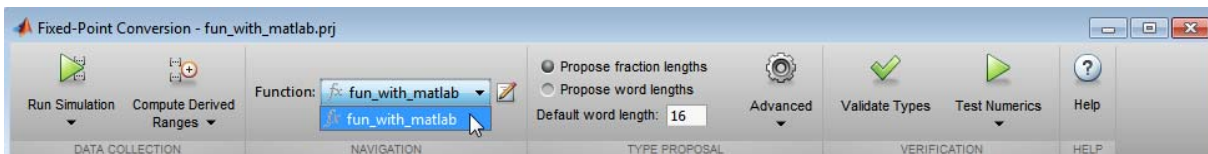
When you select **Compute Derived Ranges**, the tool runs a derived range analysis to compute static ranges for variables in your MATLAB algorithm. When the analysis is complete, the static ranges are displayed on the **Variables** tab. If the run produces $+/-\text{Inf}$ derived ranges, consider defining 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.

Viewing Functions

You can view a list of functions in your project on the **Navigation** pane. This list also includes function specializations. When you select a function from the list, the MATLAB code for that function is displayed in the Fixed-Point Conversion tool.



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.

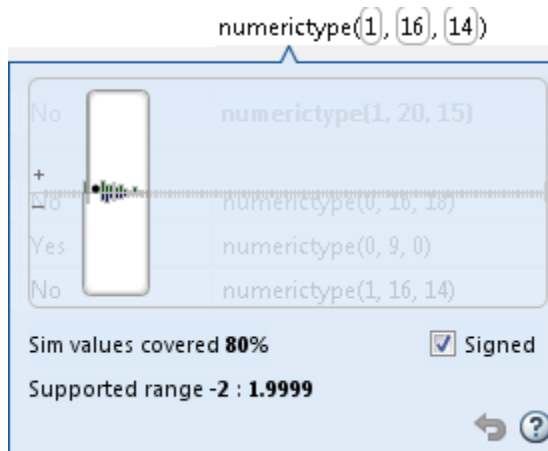
- The proposed fixed-point data type for the specified word (or fraction) length. Proposed data types use the `numerictype` notation. For example, `numerictype(1,16,12)` denotes a signed fixed-point type with a word length of 16 and a fraction length of 12. `numerictype(0,16,12)` denotes an unsigned fixed-point type with a word length of 16 and a fraction length of 12.

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.

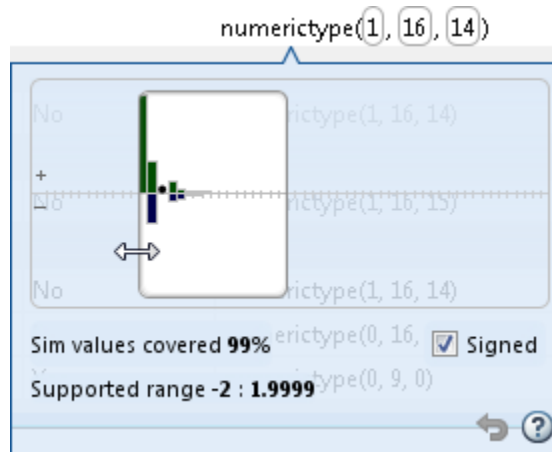
Histogram

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 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 filetest bench to define inputs or run a simulation, the tool uses this test filetest bench to test numerics. Optionally, you can add test files and select to run more than one test file. 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.

By default, the Fixed-Point Conversion tool runs the test files that you added and selected for running the simulation. You can add test files and select to run more than one test file to test numerics.

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.

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 4-20
- “HDL Code Generation from System Objects” on page 4-29
- “Generate Instantiable Code for Functions” on page 4-34
- “Enable MATLAB Function Block Generation” on page 4-35
- “System Design with HDL Code Generation from MATLAB and Simulink” on page 4-37
- “Generate Xilinx System Generator Black Box Block” on page 4-42
- “Generate Xilinx System Generator for DSP Black Box from MATLAB HDL Design” on page 4-44
- “Generate HDL Code from MATLAB Code Using the Command Line Interface” on page 4-51
- “Specify the Clock Enable Rate” on page 4-57
- “Specify Test Bench Clock Enable Toggle Rate” on page 4-59
- “Generate an HDL Coding Standard Report” on page 4-61
- “Generate an HDL Lint Tool Script” on page 4-63

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

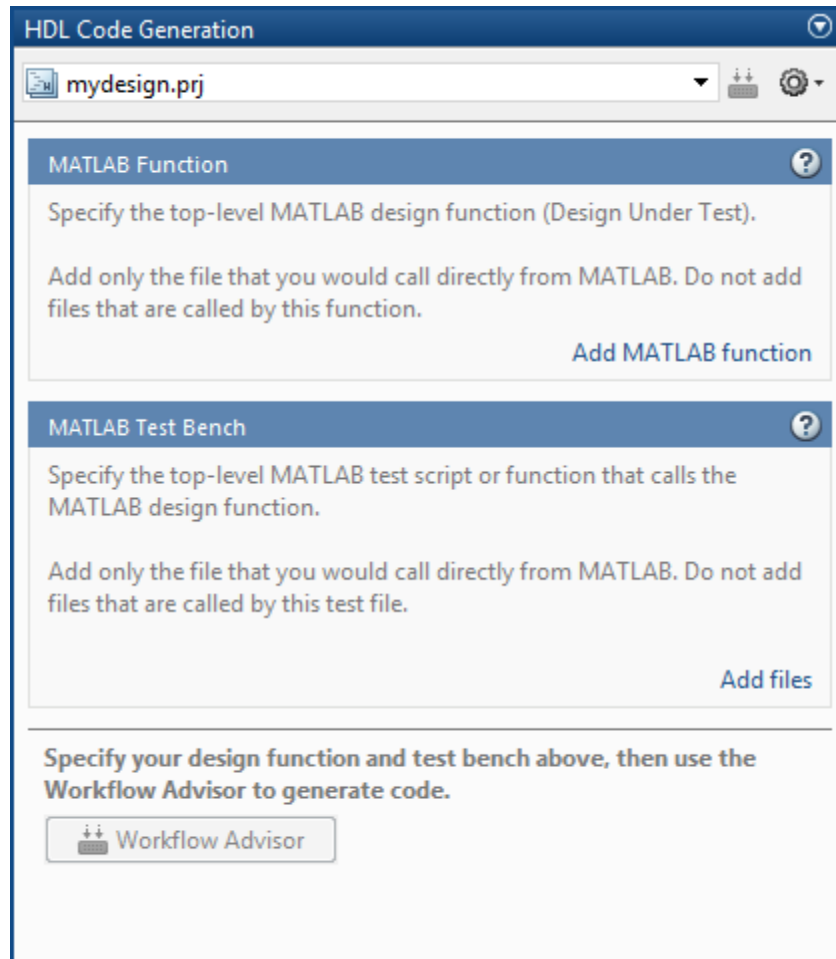
Create a New Project

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” on page 4-6.

Add a MATLAB Test Bench

You must add a MATLAB test bench unless your design does not need fixed-point conversion and you do not want to generate an RTL test bench. If you do not add a test bench, you must define the inputs to your top-level MATLAB function. For more information, see “Primary Function Input Specification” on page 4-6.

To add a test bench, do one of the following:

- In the project panel, under **MATLAB Test Bench**, click the **Add MATLAB test bench** link and browse to the file.
- Drag a file from the current folder and drop it in the project pane under **MATLAB Test Bench**.

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 4-7

“Rules for Specifying Properties of Primary Inputs” on page 4-12

“Methods for Defining Properties of Primary Inputs” on page 4-12

“Define Input Properties by Example at the Command Line” on page 4-13

“Specify Constant Inputs at the Command Line” on page 4-16

“Specify Variable-Size Inputs at the Command Line” on page 4-18

When to Specify Input Properties

If you supply a test bench for your MATLAB algorithm, you do not need to manually specify the primary function inputs. The HDL Coder software uses the test bench to infer the data types.

Why You Must Specify Input Properties

Because C and C++ are statically typed languages, MATLAB CoderHDL Coder Fixed-Point Designer must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, MATLAB CoderHDL CoderFixed-Point Designer 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 MATLAB CoderHDL CoderFixed-Point Designer. If your primary function has no input parameters, MATLAB CoderHDL CoderFixed-Point Designer 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 MATLAB Coder projects, if you want a different type to appear in the generated code, specify the type. Otherwise, the inputs default to real, scalar doubles.
- When generating code with `codegen`, you must specify the type of these inputs using the `-args` option.

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
Fixed-point inputs	✓	✓	✓	✓	✓
Each field in a structure input	<p>Specify properties for each field according to its class</p> <p>When a primary input is a structure, the code generation software treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition:</p> <ul style="list-style-type: none"> • For each field of input structures, specify class, size, and complexity. • For each field that is fixed-point class, also specify <code>numerictype</code>, and <code>fimath</code>. 				
Other inputs	✓	✓	✓		

For...	Specify properties...				
	Class	Size	Complexity	numericity	fimath
Fixed-point inputs	✓	✓	✓	✓	✓
Other inputs	✓	✓	✓		

The following data types are not supported for primary function inputs, although you can use them within the primary function:

- structure
- matrix

Variable-size data is not supported in the test bench or the primary function.

Default Property Values

MATLAB CoderHDL CoderFixed-Point Designer assigns the following default values for properties of primary function inputs.

Property	Default
class	double
size	scalar
complexity	real
numericity	No default
fimath	MATLAB default fimath object

Property	Default
class	double
size	scalar
complexity	real

Property	Default
numerictype	No default
fimath	hdlfimath

Specifying Default Values for Structure Fields. In most cases, when you don't explicitly specify values for properties, MATLAB CoderHDL CoderFixed-Point Designer uses defaults except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore, you might need to specify default values for properties of structure fields. For examples, see “Specifying Class and Size of Scalar Structure” and “Specifying Class and Size of Structure Array”.

Specifying Default fimath Values for MEX Functions. MEX functions generated with MATLAB CoderFixed-Point Designer use the default `fimath` value in effect at compile time. If you do not specify a default `fimath` value, MATLAB CoderFixed-Point Designer uses the MATLAB default `fimath`. The MATLAB factory default has the following properties:

```
RoundingMethod: Nearest
OverflowAction: Saturate
ProductMode: FullPrecision
SumMode: FullPrecision
CastBeforeSum: true
```

For more information, see “`fimath` for Sharing Arithmetic Rules”.

When running MEX functions that depend on the default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a run-time warning, alerting you to a mismatch between the compile-time and run-time `fimath` values.

For example, suppose you define the following MATLAB function `test`:

```
function y = test %#codegen
y = fi(0);
```

The function `test` constructs a `fi` object without explicitly specifying a `fimath` object. Therefore, `test` relies on the default `fimath` object in effect at compile

time. At the MATLAB prompt, generate the MEX function `test_mex` to use the factory setting of the MATLAB default `fimath`:

```
codegen test
% codegen generates a MEX function, test_mex,
% in the current folder
```

Next, run `test_mex` to display the MATLAB default `fimath` value:

```
test_mex

ans =

      0

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 15
```

Now create a local MATLAB `fimath` value. so you no longer use the default setting:

```
F = fimath('RoundingMethod','Floor');
```

Finally, clear the MEX function from memory and rerun it:

```
clear test_mex
test_mex
```

The mismatch is detected and causes an error:

```
??? This function was generated with a different default
fimath than the current default.
```

```
Error in ==> test_mex
```

Supported Classes

The following table presents the class names supported by MATLAB CoderHDL CoderFixed-Point Designer.

Class Name	Description
logical	Logical array of true and false values
char	Character array
int8	8-bit signed integer array
uint8	8-bit unsigned integer array
int16	16-bit signed integer array
uint16	16-bit unsigned integer array
int32	32-bit signed integer array
uint32	32-bit unsigned integer array
int64	64-bit signed integer array
uint64	64-bit unsigned integer array
single	Single-precision floating-point or fixed-point number array
double	Double-precision floating-point or fixed-point number array
struct	Structure array
embedded.fi	Fixed-point number array

Class Name	Description
logical	Logical array of true and false values
char	Character array
int8	8-bit signed integer array
uint8	8-bit unsigned integer array
int16	16-bit signed integer array
uint16	16-bit unsigned integer array
int32	32-bit signed integer array
uint32	32-bit unsigned integer array

Class Name	Description
single	Single-precision floating-point or fixed-point number array
double	Double-precision floating-point or fixed-point number array
embedded.fi	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

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

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

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
“Specifying Properties of Primary Function Inputs in a Project”	<ul style="list-style-type: none"> • If you are working in a MATLAB CoderHDL Coder project, easy to use • Does not alter original MATLAB code • MATLAB CoderHDL Coder saves the definitions in the project file 	<ul style="list-style-type: none"> • Not efficient for specifying memory-intensive inputs such as large structures and arrays
“Define Input Properties by Example at the Command Line” on page 4-13	<ul style="list-style-type: none"> • Easy to use • Does not alter original MATLAB code 	<ul style="list-style-type: none"> • Must be specified at the command line every time you invoke <code>codegen</code> (unless you use a script)

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"> • Designed for prototyping a function that has a small number of primary inputs 	<ul style="list-style-type: none"> • 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 MATLAB CoderHDL Coder • Provides documentation of property specifications in the MATLAB code • Efficient for specifying memory-intensive inputs such as large structures 	<ul style="list-style-type: none"> • Uses complex syntax • MATLAB CoderHDL Coder project files do not currently recognize properties defined programmatically. If you are using a project, you must reenter the input types in the project.

Define Input Properties by Example at the Command Line

- “Command Line Option -args” on page 4-14
- “Rules for Using the -args Option” on page 4-14
- “Specifying Properties of Primary Inputs by Example at the Command Line” on page 4-14
- “Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line” on page 4-15

Command Line Option `-args`

The `codegen` function provides a command-line option `-args` for specifying the properties of primary (entry-point) function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values. Using this option, you specify the properties of inputs at the same time as you generate code for the MATLAB function with `codegen`. If you have a test function or script that calls the entry-point MATLAB function with the required types, you can use `coder.getArgTypes` to determine the types of the function inputs. `coder.getArgTypes` returns a cell array of `coder.Type` objects that you can pass to `codegen` using the `-args` option. For more information, see the `coder.getArgTypes` function reference information.

See “Specifying General Properties of Primary Inputs” for `codegen`.

Rules for Using the `-args` Option

When using the `-args` command-line option to define properties by example, follow these rules:

- The cell array of sample values must contain the same number of elements as primary function inputs.
- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature — for example, the first element in the cell array defines the properties of the first primary function input.

Note If you specify an empty cell array with the `-args` option, `codegen` interprets this to mean that the function takes no inputs; a compile-time error occurs if the function does have inputs.

Specifying Properties of Primary Inputs by Example at the Command Line

Consider a MATLAB function that adds its two inputs:

```
function y = mcf(u,v)
%#codegen
y = u + v;
```


The following examples show how to specify different properties of the primary inputs *u* and *v* by example at the command line:

- Use a literal cell array of constants to specify that both inputs are real scalar doubles:

```
codegen mcf -args {0,0}
```

- Use a literal cell array of constants to specify that input *u* is an unsigned 16-bit, 1-by-4 vector and input *v* is a scalar double:

```
codegen mcf -args {zeros(1,4,'uint16'),0}
```

- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

```
a = uint8([1;2;3;4])
b = uint8([5;6;7;8])
ex = {a,b}
codegen mcf -args ex
```

Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line

To generate a MEX function or C/C++ code for fixed-point MATLAB code, you must install Fixed-Point Designer software.

Consider a MATLAB function that calculates the square root of a fixed-point number:

```
##codegen
function y = sqrtfi(x)
y = sqrt(x);
```

To specify the properties of the primary fixed-point input *x* by example on the MATLAB command line, follow these steps:

- 1 Define the `numericType` properties for *x*, as in this example:

```
T = numericType('WordLength',32,...
               'FractionLength',23,...
               'Signed',true);
```

- 2** Define the `fimath` properties for `x`, as in this example:

```
F = fimath('SumMode','SpecifyPrecision',...
          'SumWordLength',32,...
          'SumFractionLength',23,...
          'ProductMode','SpecifyPrecision',...
          'ProductWordLength',32,...
          'ProductFractionLength',23);
```

- 3** Create a fixed-point variable with the `numericType` and `fimath` properties you just defined, as in this example:

```
myeg = { fi(4.0,T,F) };
```

- 4** Compile the function `sqrtfi` using the `codegen` command, passing the variable `myeg` as the argument to the `-args` option, as in this example:

```
codegen sqrtfi -args myeg;
```

Specify Constant Inputs at the Command Line

In cases where you know your primary inputs will not change at run time, it is more efficient to specify them as constant values than as variables to eliminate unnecessary overhead in generated code. Common uses of constant inputs are for flags that control how an algorithm executes and values that specify the sizes or types of data.

You can define inputs to be constants using the `-args` command-line option with a `coder.Constant` object, as in this example:

```
-args {coder.Constant(constant_input)}
```

This expression specifies that an input will be a constant with the size, class, complexity, and value of *constant_input*.

Calling Functions with Constant Inputs

`codegen` compiles constant function inputs into the generated code. As a result, the MEX function signature differs from the MATLAB function signature. At run time you supply the constant argument to the MATLAB function, but not to the MEX function.

For example, consider the following function `identity` which copies its input to its output:

```
function y = identity(u) %#codegen
y = u;
```

To generate a MEX function `identity_mex` with a constant input, at the MATLAB prompt, type the following command:

```
codegen identity -args {coder.Constant(42)}
```

To run the MATLAB function, supply the constant argument:

```
identity(42)
```

You get the following result:

```
ans =
    42
```

Now, try running the MEX function with this command:

```
identity_mex
```

You should get the same answer.

Specifying a Structure as a Constant Input

Suppose you define a structure `tmp` in the MATLAB workspace to specify the dimensions of a matrix:

```
tmp = struct('rows', 2, 'cols', 3);
```

The following MATLAB function `rowcol` accepts a structure input `p` to define matrix `y`:

```
function y = rowcol(u,p) %#codegen
y = zeros(p.rows,p.cols) + u;
```

The following example shows how to specify that primary input `u` is a double scalar variable and primary input `p` is a constant structure:

```
codegen rowcol -args {0,coder.Constant(tmp)}
```

Specify Variable-Size Inputs at the Command Line

Variable-size data is data whose size might change at run time. MATLAB supports bounded and unbounded variable-size data for code generation. *Bounded variable-size data* has fixed upper bounds. This data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds. This data must be allocated on the heap. You can define inputs to have one or more variable-size dimensions — and specify their upper bounds — using the `-args` option and `coder.typeof` function:

```
-args {coder.typeof(example_value, size_vector, variable_dims)}
```

Specifies a variable-size input with:

- Same class and complexity as *example_value*
- Same size and upper bounds as *size_vector*
- Variable dimensions specified by *variable_dims*

When you enable dynamic memory allocation, you can specify `Inf` in the size vector for dimensions with unknown upper bounds at compile time.

When *variable_dims* is a scalar, it is applied to all the dimensions, with the following exceptions:

- If the dimension is 1 or 0, which are fixed.
- If the dimension is unbounded, which is always variable size.

For more information, see `coder.typeof` and “Generate Code for Variable-Size Data”.

Specifying a Variable-Size Vector Input

- 1 Write a function that computes the average of every `n` elements of a vector `A` and stores them in a vector `B`:

```
function B = nway(A,n) %#codegen
```

```

% Compute average of every N elements of A and put them in B.

coder.extrinsic('error');
if ((mod(numel(A),n) == 0) && (n>=1 && n<=numel(A)))
    B = ones(1,numel(A)/n);
    k = 1;
    for i = 1 : numel(A)/n
        B(i) = mean(A(k + (0:n-1)));
        k = k + n;
    end
else
    B = zeros(1,0);
    error('n <= 0 or does not divide number of elements evenly');
end

```

- 2** Specify the first input A as a vector of double values. Its first dimension stays fixed in size and its second dimension can grow to an upper bound of 100. Specify the second input n as a double scalar.

```
codegen -report nway -args {coder.typeof(0,[1 100],1),1}
```

- 3** As an alternative, assign the `coder.typeof` expression to a MATLAB variable, then pass the variable as an argument to `-args`:

```
vareg = coder.typeof(0,[1 100],1)
codegen -report nway -args {vareg, 0}
```

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.m';  
testbench_name = 'mlhdlc_sfir_tb.m';
```

- 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', 'hdlcoderdemo');  
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), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), 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.

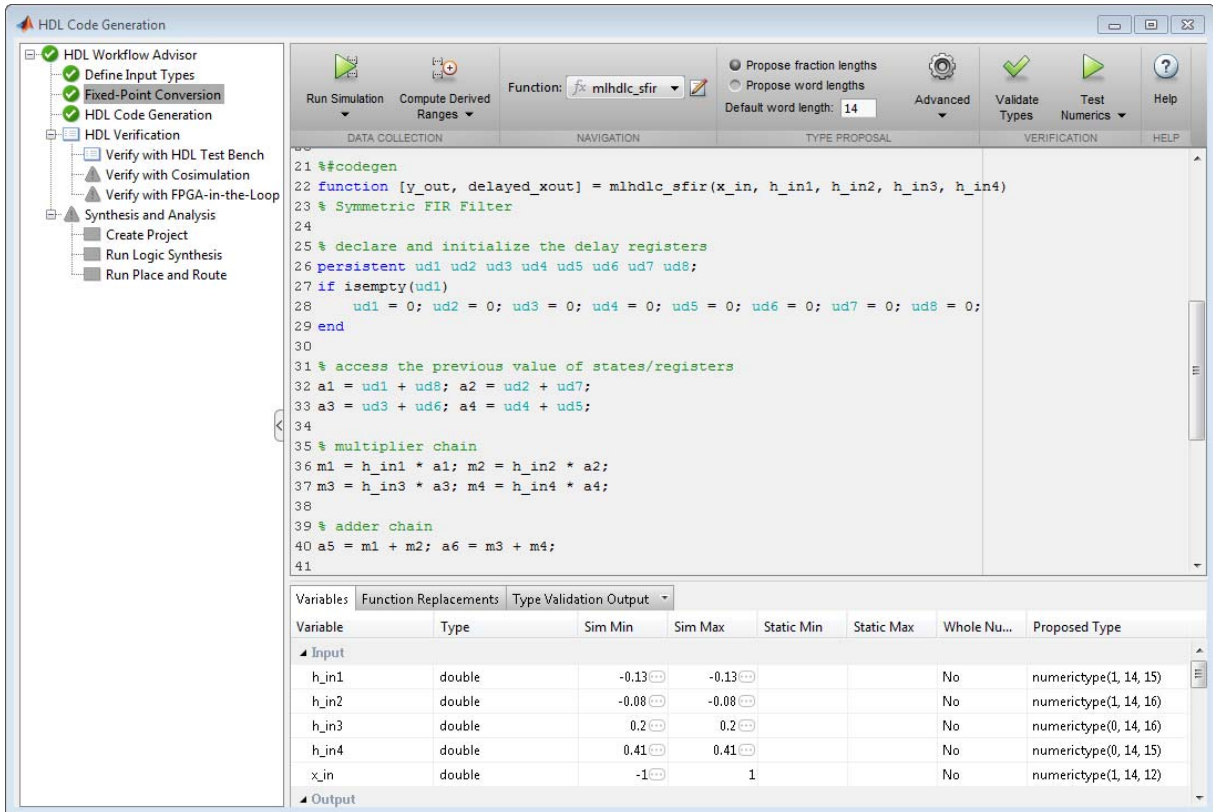
You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete introduction to creating and populating HDL Coder projects.

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.



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 Test bench tab, choose the 'Multi-file test bench' option to generate test bench code and test bench data (stimulus and response) in separate files.
- 5** In the 'Optimizations' tab, choose '1' as the Input and Output pipeline length, and enable the 'Distribute pipeline registers' option.
- 6** 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.
- 7** Click the 'Run' button to generate both the Verilog design and testbench with reports.

Code Generation Options

Target Selection

Language:

Output Settings

- Check HDL conformance
- Generate HDL
- Generate HDL test bench
- Generate EDA scripts

Test Bench Options

- Multi file test bench

Optimization Options

- Distribute pipeline registers

Input pipelining:

Output pipelining:

Coding Style Options

Generated Code Comments

- Preserve MATLAB code comments
- Include MATLAB source code as comments

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

```


### Begin Verilog Code Generation
### Working on mlhdlc_sfir_FixPt as mlhdlc\_sfir\_FixPt.v
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays
### Output port 0: 2 cycles
### Output port 1: 2 cycles
### Output port 2: 2 cycles
### Generating Resource Utilization Report resource\_report.html Resource Report


### Begin Test Bench Generation
### Collect Test Bench Stimulus and Response
### Begin Simulation to log data

### Simulation to log data completed in 2.4955 sec(s)
### ### Accounting for Output port Latency: 2 cycles
### Collecting data...
### Begin HDL test bench file generation with logged samples
### Generating test bench package: mlhdlc\_sfir\_FixPt\_tb\_pkg.v
### Generating test bench data file: mlhdlc\_sfir\_FixPt\_tb\_data.v
### Generating test bench: mlhdlc\_sfir\_FixPt\_tb.v

Code generation successful: View report MATLAB Report with
### Elapsed Time: 55.3389 sec(s) Traceability Links

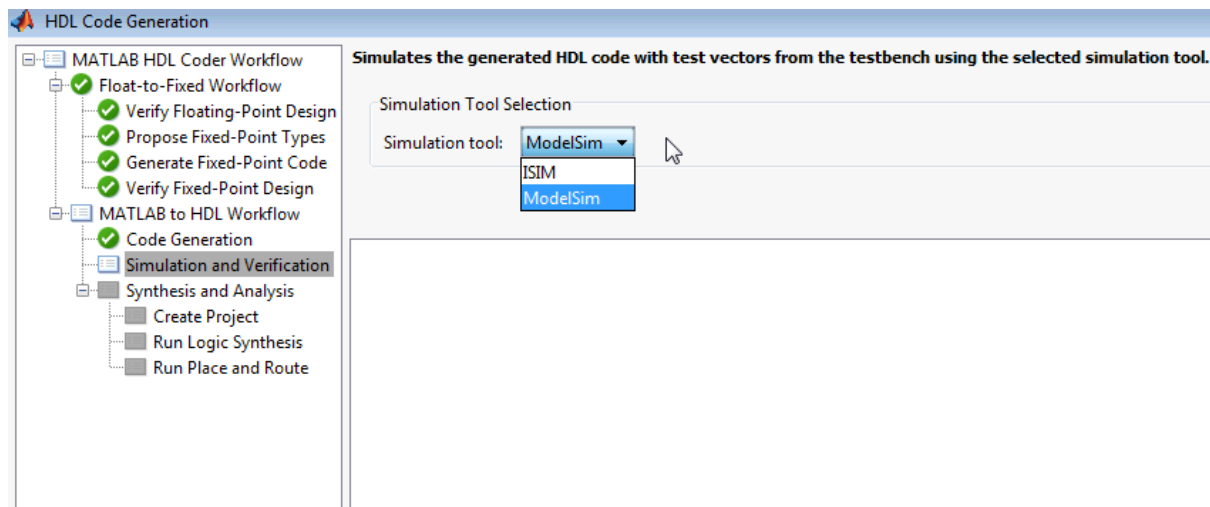
```

 **Verilog Code**

 **Verilog Testbench**

Step 3: Simulate the Generated Code

HDL Coder automates the process of 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.

HDL Code Generation

- ✓ MATLAB HDL Coder Workflow
 - ✓ Float-to-Fixed Workflow
 - ✓ Verify Floating-Point Design
 - ✓ Propose Fixed-Point Types
 - ✓ Generate Fixed-Point Code
 - ✓ Verify Fixed-Point Design
 - ✓ MATLAB to HDL Workflow
 - ✓ Code Generation
 - ✓ Simulation and Verification
 - ⚠ Synthesis and Analysis
 - ✓ Create Project
 - ✓ Run Logic Synthesis
 - Run Place and Route

Create synthesis project for supported synthesis tool.

Synthesis Tool Selection

Synthesis tool:

Chip family:

Device name:

Package name:

Speed value:

```
### Creating Synthesis Project for 'mlhdlc_sfir_FixPt'
```

```
### Create new Xilinx ISE 13.1 project
```

```
C:\Users\kkintali\AppData\Local\Temp\mlhdlc_sfir\codegen\mlhdlc_sfir\hdlsrc\ise_prj\mlhdlc_sfir_FixPt_ise.xise
```

```
### Set Xilinx ISE 13.1 project properties
```

```
### Update Xilinx ISE 13.1 project with HDL source files
```

```
INFO:HDLCompiler:1693 - Analyzing Verilog file
```

```
"C:/Users/kkintali/AppData/Local/Temp/mlhdlc_sfir/codegen/mlhdlc_sfir/hdlsrc
```

```
/mlhdlc_sfir_FixPt.v" into library work
```

```
INFO:ProjectMgmt:659 - Parsing design hierarchy completed successfully.
```

```
### Close Xilinx ISE 13.1 project.
```

```
Elapsed time is 93.1982 seconds.
```

```
### Synthesis Project creation successful
```

```
### Elapsed Time: 106.8257 sec(s)
```

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

HDL Code Generation

Launch selected synthesis tool and synthesize the generated HDL code.

Asynchronous Control Signals Information:

Control Signal	Buffer (FF name)	Load
reset	IBUF	321

Timing Summary:

Speed Grade: -10

Minimum period: 5.527ns (Maximum Frequency: 180.933MHz)
 Minimum input arrival time before clock: 6.132ns
 Maximum output required time after clock: 6.717ns
 Maximum combinational path delay: 7.093ns

=====
 Process "Synthesize - XST" completed successfully
 ### Synthesis Complete.
 ### Close Xilinx ISE 13.1 project.
 object.
 Elapsed time is 101.9205 seconds.
 ### Generating Synthesis report [mlhdlc_sfir_FixPt_syn_results.txt](#)
 ### Synthesis successful
 ### Elapsed Time: 101.9728 sec(s)

Clean up the Generated Files

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

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemo');
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.m';
testbench_name = 'mlhdlc_sysobj_ex_tb.m';
```

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)
% Symmetric FIR Filter

persistent h1 h2 h3 h4 h5 h6 h7 h8;
if isempty(h1)
    h1 = dsp.Delay('FrameBasedProcessing', false);
    h2 = dsp.Delay('FrameBasedProcessing', false);
    h3 = dsp.Delay('FrameBasedProcessing', false);
    h4 = dsp.Delay('FrameBasedProcessing', false);
    h5 = dsp.Delay('FrameBasedProcessing', false);
    h6 = dsp.Delay('FrameBasedProcessing', false);
    h7 = dsp.Delay('FrameBasedProcessing', false);
```

```
        h8 = dsp.Delay('FrameBasedProcessing', false);
    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, a);
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', 'hdlcoderdemo');
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);

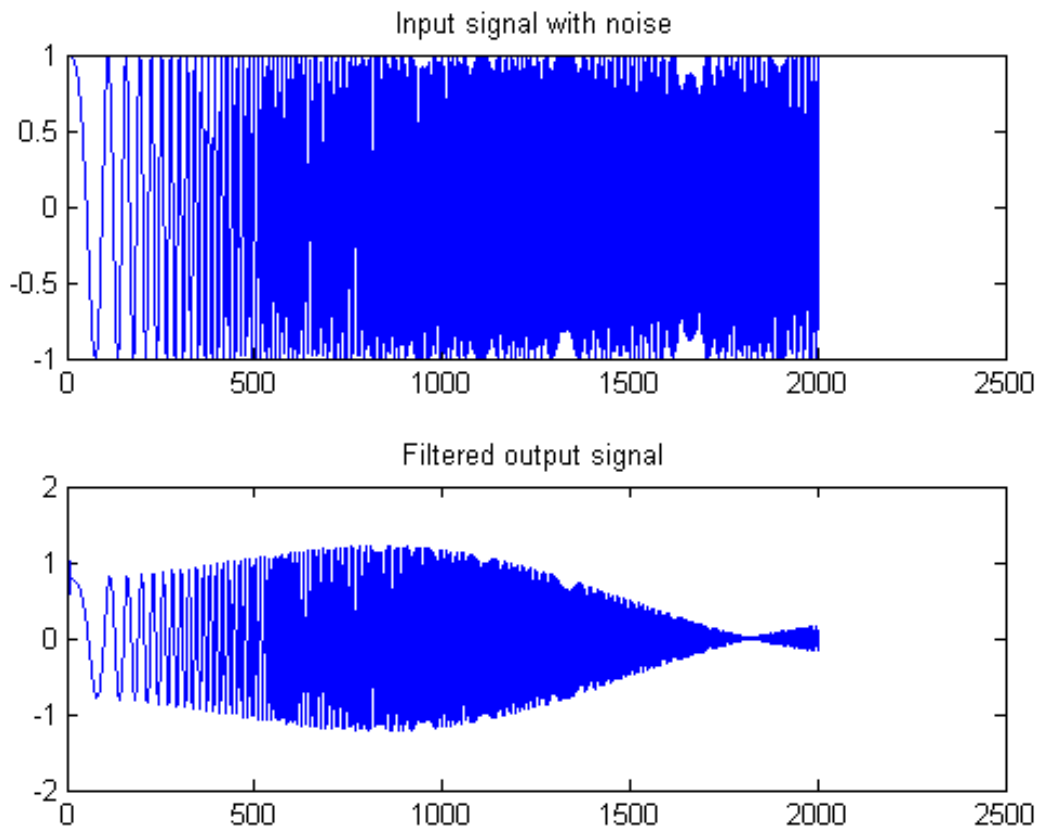
```

```
copyfile(fullfile(mlhdlc_demo_dir, design_name), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), 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', 'hdlcoderdem  
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 4-34

“Generate Code Inline for Specific Functions” on page 4-34

“Limitations for Instantiable Code Generation for Functions” on page 4-34

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:

- 1 In the HDL Workflow Advisor, select the **HDL Code Generation** task.
- 2 In the **Advanced** tab, select **Generate instantiable code for functions**.

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.

Enable MATLAB Function Block Generation

In this section...

“Requirements for MATLAB Function Block Generation” on page 4-35

“Enable MATLAB Function Block Generation” on page 4-35

“Results of MATLAB Function Block Generation” on page 4-35

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 Box** 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” on page 4-37.

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.m';  
testbench_name = 'mlhdlc_fir_tb.m';
```

- 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', 'hdlcoderdemo');
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), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), 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.

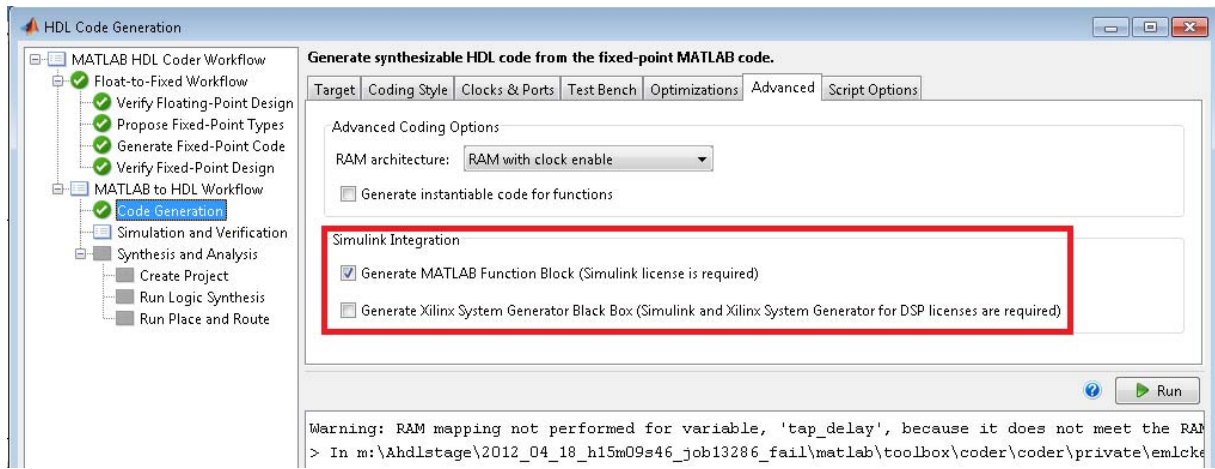
Click the Workflow Advisor button to launch the HDL Workflow Advisor.

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.

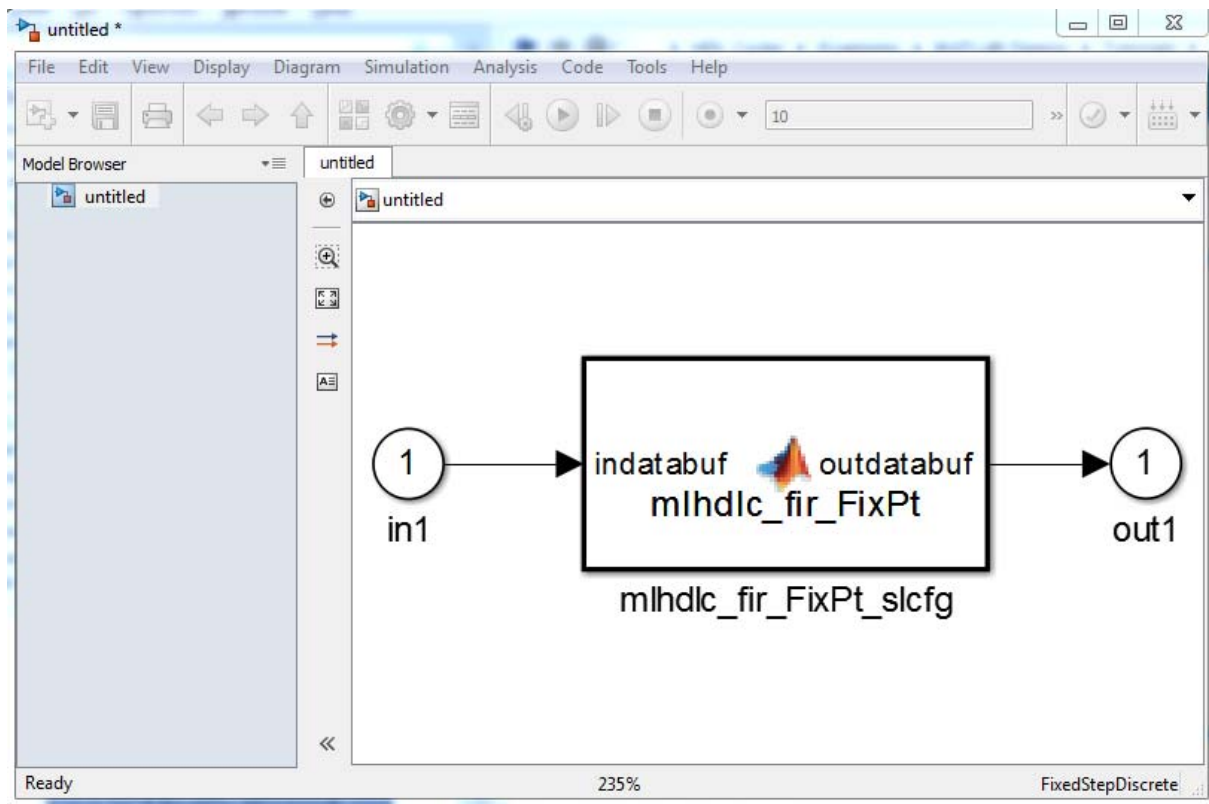
Right-click the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

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', 'hdlcoderdemo');  
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 4-42
“Enable System Generator Black Box Block Generation” on page 4-42
“Results of System Generator Black Box Block Generation” on page 4-43

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” on page 18-33.

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.m';  
testbench_name = 'mlhdlc_fir_tb.m';
```

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', 'hdlcoderdemo');  
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), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), 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.

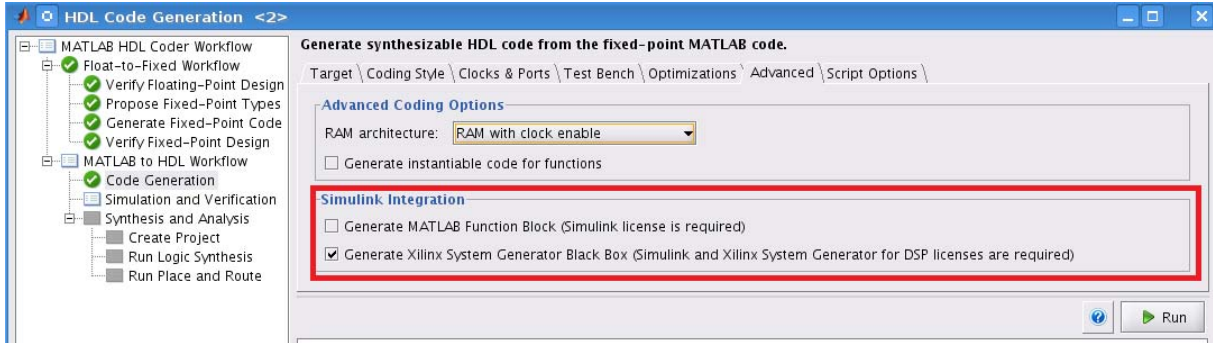
Click the Workflow Advisor button to launch the HDL Workflow Advisor.

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:

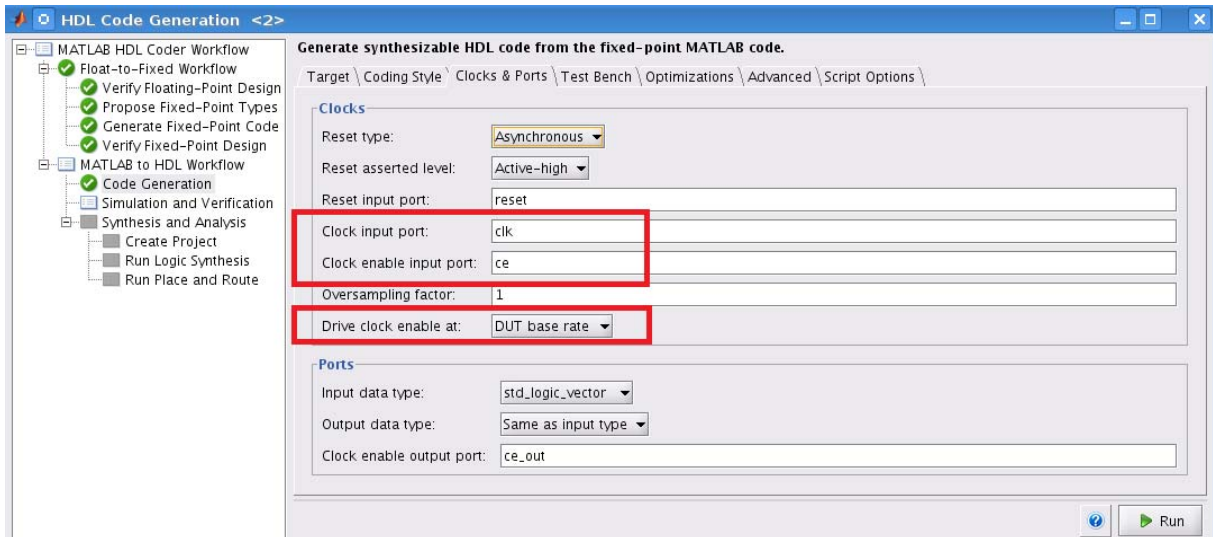
```
xlVersion
```

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

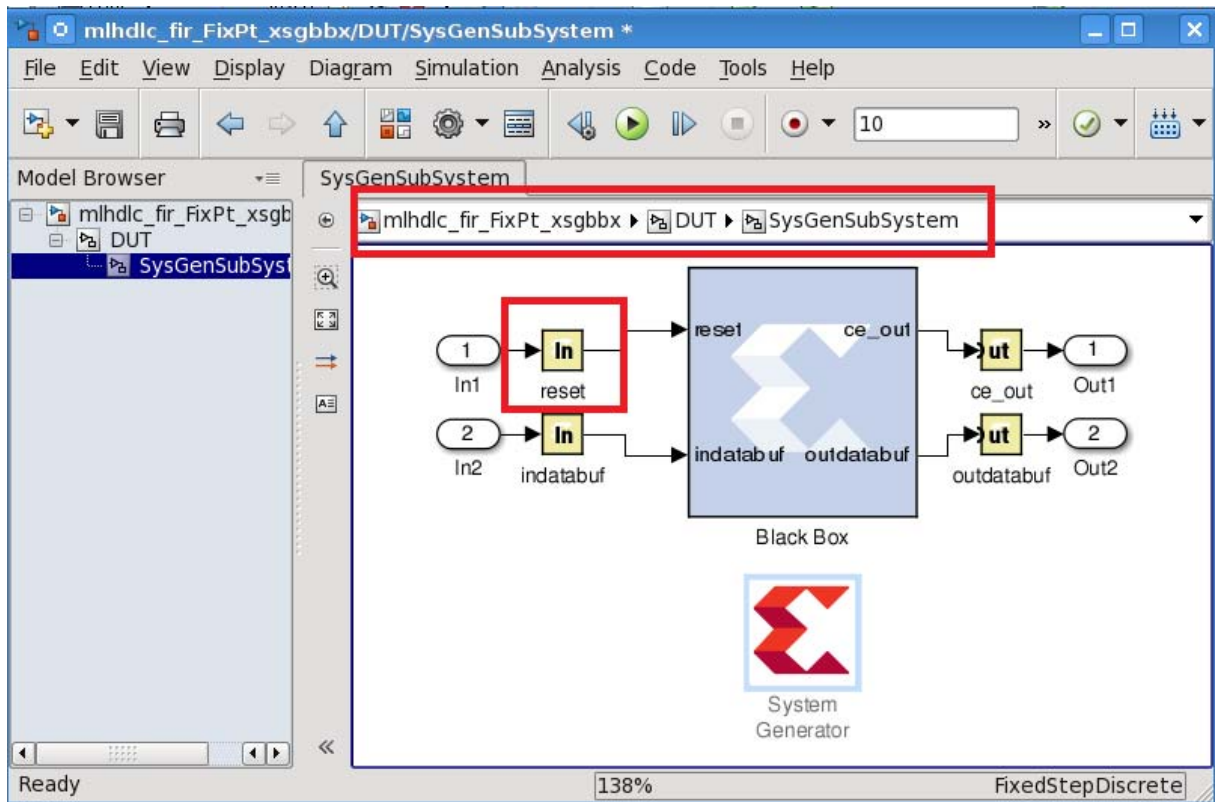
Right-click the 'Code Generation' step and choose the 'Run to selected task' option to run all the steps from the beginning through HDL code generation.

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_xsgbbxcfg.m');
```

```

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', 'hdlcoderdemo');
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.m';
testbench_name = 'mlhdlc_dti_tb.m';
```

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemo');
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), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), 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;
fixptcfg = coder.config('fixpt'); % Create a 'fixpt' config with default settings
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';
hdlcfg = coder.config('hdl'); % Create an 'hdl' config with default settings
```

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 4-57

“How to Specify the Clock Enable Rate” on page 4-57

Why Specify the Clock Enable Rate?

When the coder performs area optimizations, it might upsample parts of your design (DUT), and thereby introduce an increase in your required DUT clock frequency.

If the coder upsamples your design, it generates a message indicating the ratio between the new clock frequency and your original clock frequency. For example, the following message indicates that your design’s new required clock frequency is 4 times higher than the original frequency:

The design requires 4 times faster clock with respect to the base rate = 1

This frequency increase introduces a rate mismatch between your input clock enable and output clock enable, because the output clock enable runs at the slower original clock frequency.

With the **Drive clock enable at** option, you can choose whether to drive the input clock enable at the faster rate (**DUT base rate**) or at a rate that is less than or equal to the original clock enable rate (**Input data rate**).

How to Specify the Clock Enable Rate

- 1 In the HDL Workflow Advisor, select **MATLAB to HDL Workflow > Code Generation**. Click the **Clocks & Ports** tab.
- 2 For the **Drive clock enable at** option, select **Input data rate** or **DUT base rate**.

Drive clock enable at Option	Clock Enable Behavior
Input data rate (default)	<p>Each assertion of the input clock enable produces an output clock enable assertion.</p> <p>You can assert the input clock enable at a maximum rate of once every N clocks. N = 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>
DUT base rate	<p>Input clock enable rate does not match the output clock enable rate. You must assert the input clock enable with your input data N times to get 1 output clock enable assertion. N = 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 4-59

“How to Specify Test Bench Clock Enable Toggle Rate” on page 4-59

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. $N = \frac{\text{upsampled clock rate}}{\text{original clock rate}}$. Refer to the clock enable behavior for **Input data rate**, in “Specify the Clock Enable Rate” on page 4-57.

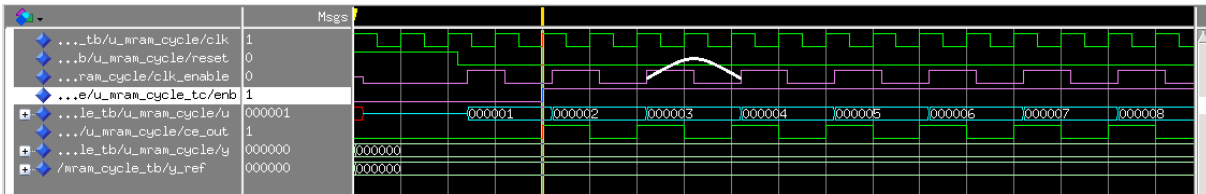
How to Specify Test Bench Clock Enable Toggle Rate

To set your test bench clock enable toggle rate:

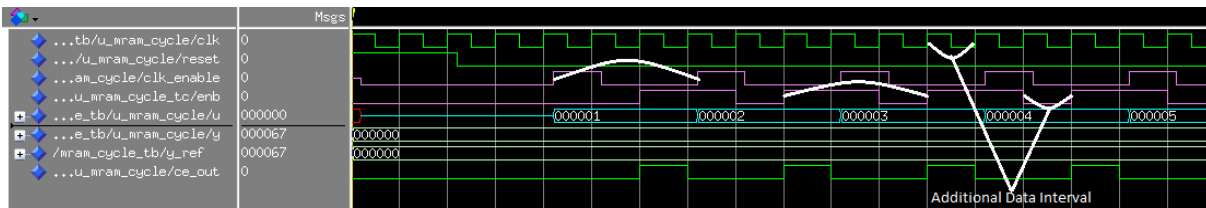
- 1** In the HDL Workflow Advisor, select **MATLAB to HDL Workflow > Code Generation**.
- 2** In the **Clocks & Ports** tab, for the **Drive clock enable at** option, select **Input data rate**.
- 3** In the **Test Bench** tab, for **Input data interval**, enter 0 or an integer greater than the maximum 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.
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

In this section...

“Using the HDL Workflow Advisor” on page 4-61

“Using the Command Line” on page 4-61

To learn more about the HDL coding standard report, see “HDL Coding Standard Report” on page 17-2.

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 Style** tab.
- 2** For **HDL coding standard**, select **Industry**.
- 3** 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
### 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.

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. The coder can generate the following lint tool script file formats:

- Leda
- SpyGlass
- Custom

You can further customize the script with initialization, termination, and command strings.

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 **SpyGlass**, **Leda**, or **Custom**.
- 4** Enter text to customize the **Lint script initialization**, **Lint script command**, and **Lint script termination** 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 `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 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, termination, and command strings, use the `HDLLintTool`, `HDLLintInit`, `HDLLintTerm`, and `HDLLintCmd` 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.HDLLintTerm = 'myTermination';  
hdlcfg.HDLLintCmd = 'myCommand';
```

After you generate code, the command window shows a link to the lint tool script.

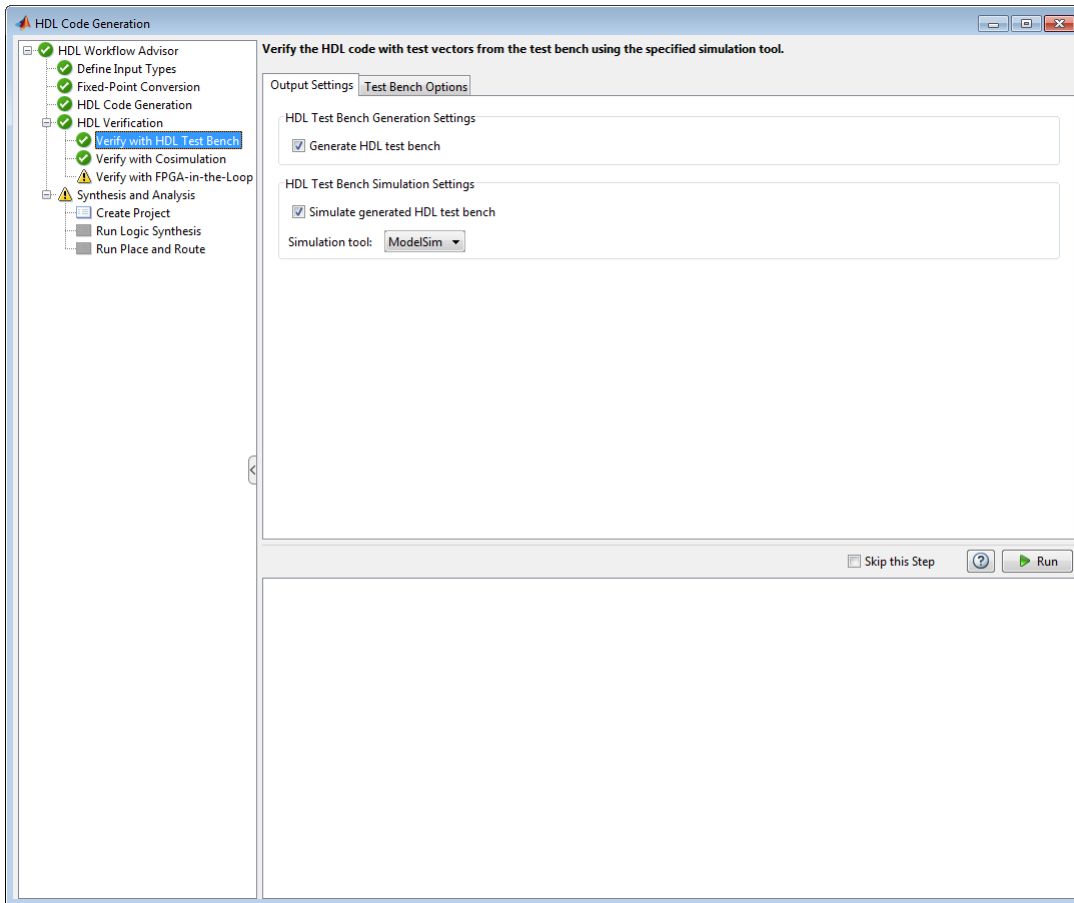
Verification

- “Verify Code with HDL Test Bench” on page 5-2
- “Generate Test Bench With File I/O” on page 5-6

Verify Code with HDL Test Bench

Simulate the generated HDL design under test (DUT) with test vectors from the test bench using the specified simulation tool.

- 1 Start the MATLAB to HDL Workflow Advisor.



- 2 At step **HDL Verification**, click **Verify with HDL Test Bench**.
- 3 Select **Generate HDL test bench**.

This option enables HDL Coder to generate HDL test bench code from your MATLAB test script.

- 4** Optionally, select **Simulate generated HDL test bench**. This option enables MATLAB to simulate the HDL test bench with the HDL DUT.

If you select this option, you must also select the **Simulation tool**.

- 5** For **Test Bench Options**, select and set the optional parameters according to the descriptions in the following table.

HDL Test Bench Parameter	Description
Test bench name postfix	Specify the postfix for the test bench name.
Force clock	Enable for test bench to force clock input signals.
Clock high time (ns)	Specify the number of nanoseconds the clock is high.
Clock low time (ns)	Specify the number of nanoseconds the clock is low.
Hold time (ns)	Specify the hold time for input signals and forced reset signals.
Force clock enable	Enable to force clock enable.
Clock enable delay (in clock cycles)	Specify time (in clock cycles) between deassertion of reset and assertion of clock enable.
Force reset	Enable for test bench to force reset input signals.
Reset length (in clock cycles)	Specify time (in clock cycles) between assertion and deassertion of reset.
Hold input data between samples	Enable to hold substrate signals between clock samples.

HDL Test Bench Parameter	Description
Input data interval	Specifies the number of clock cycles between assertions of clock enable. For more information, see “Specify Test Bench Clock Enable Toggle Rate” on page 4-59.
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 5-6

“How Test Bench Generation with File I/O Works” on page 5-6

“Test Bench Data Files” on page 5-7

“How to Generate Test Bench with File I/O” on page 5-7

“Limitations When Using File I/O In Test Bench” on page 5-7

When to Use File I/O In Test Bench

By default, the 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, the 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, the 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

- “Program Standalone FPGA with FPGA Turnkey Workflow” on page 6-2
- “Hardware and Software Codesign for Xilinx Zynq-7000 Platform” on page 6-7
- “Hardware and Software Codesign Workflow” on page 6-8
- “Generate Synthesis Scripts” on page 6-17
- “Install Support for Altera FPGA Boards” on page 6-18
- “Install Support for Xilinx FPGA Boards” on page 6-19
- “Install Support for Xilinx Zynq-7000 Platform” on page 6-20

Program Standalone FPGA with FPGA Turnkey Workflow

In this section...

“Before You Begin” on page 6-2

“Create a Project” on page 6-2

“Convert Design To Fixed-Point” on page 6-3

“Map Design Ports to Target Interface” on page 6-3

“Generate Programming File and Download To Hardware” on page 6-5

This example shows how to program a standalone FPGA with your MATLAB design, using the FPGA Turnkey workflow.

The target device in this example is a Xilinx Virtex-5 ML506 development board, but you can use this workflow with other Altera® or Xilinx FPGAs. For a list of supported devices, see “FPGA Turnkey Hardware”.

Before You Begin

To use the FPGA Turnkey workflow, you must:

- Have your synthesis tool path set up. To learn how to setup your synthesis tool path, see “Synthesis Tool Path Setup”.
- Connect the target device if you want to program it immediately. If the target device is not connected, you can still generate the programming file.

Create a Project

- 1 Create a new folder to hold your project files.
- 2 At the MATLAB command line, open the design and test bench files:

```
edit mlhdlc_turnkey_led_blinking.m
edit mlhdlc_turnkey_led_blinking_tb.m
```
- 3 Save copies of the design and test bench files to your new project folder.
- 4 Change directory to your new project folder.

- 5 At the MATLAB command line, enter:

```
hdlcoder
```

The project creation pane opens.

- 6 For **Name**, enter `myproject.prj` and click **OK**.
- 7 Under **MATLAB Function**, click **Add MATLAB Function** and select `mlhdlc_turnkey_led_blinking.m`.
- 8 Under **MATLAB Test Bench**, click **Add files** and select `mlhdlc_turnkey_led_blinking_tb.m`.
- 9 Click the **Workflow Advisor** button to open the HDL Workflow Advisor.

Convert Design To Fixed-Point

- 1 Right-click the **Define Input Types** task and select **Run This Task**.
- 2 In the **Fixed-Point Conversion** task, click **Advanced** and set the **Safety margin for sim min/max (%)** to 0.
- 3 On the left, right-click the **Fixed-Point Conversion** task and select **Run This Task**.

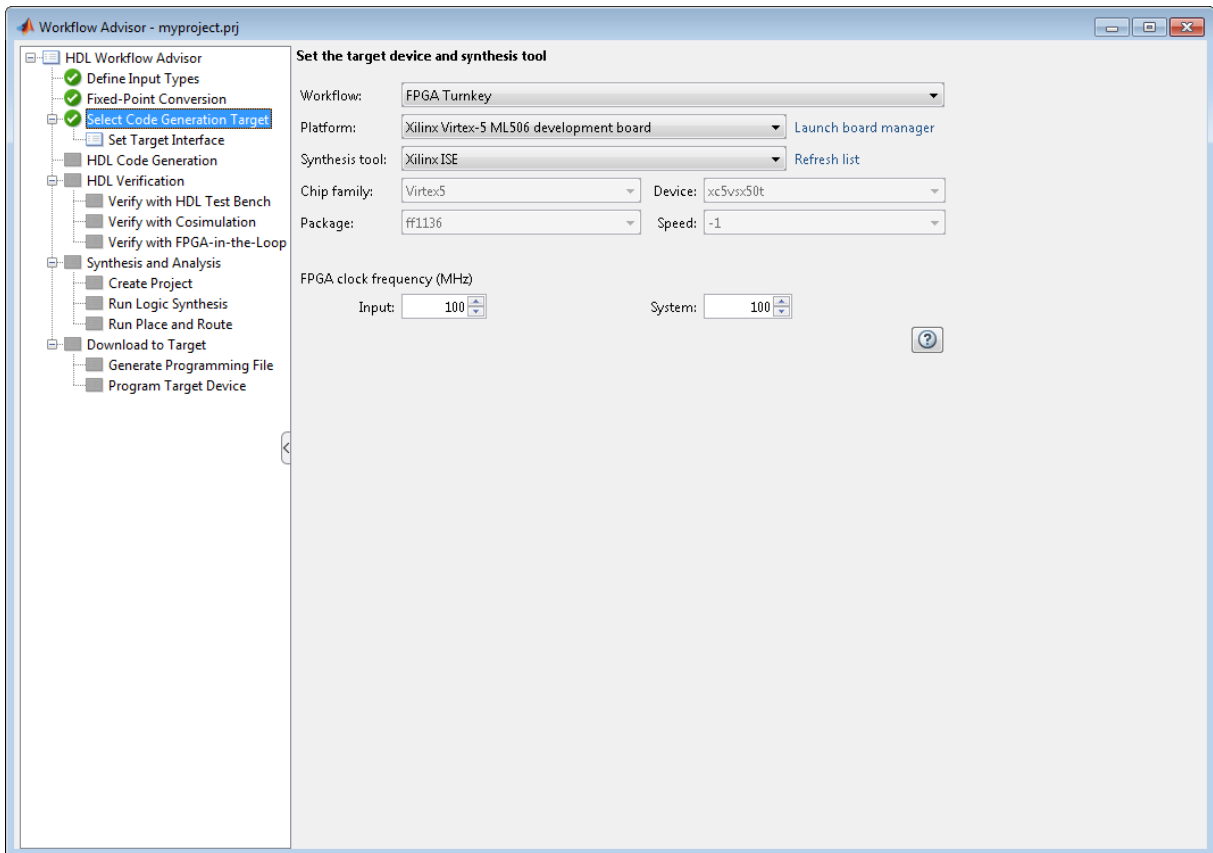
Map Design Ports to Target Interface

In the **Select Code Generation Target** task, select the FPGA Turnkey workflow and Xilinx Virtex-5 ML506 development board as follows:

- 1 For **Workflow**, select **FPGA Turnkey**.
- 2 For **Platform**, select **Xilinx Virtex-5 ML506 development board**. If your target device is not in the list, select **Get more** to download the support package.

The coder automatically sets **Chip family**, **Device**, **Package**, and **Speed** according to your platform selection.

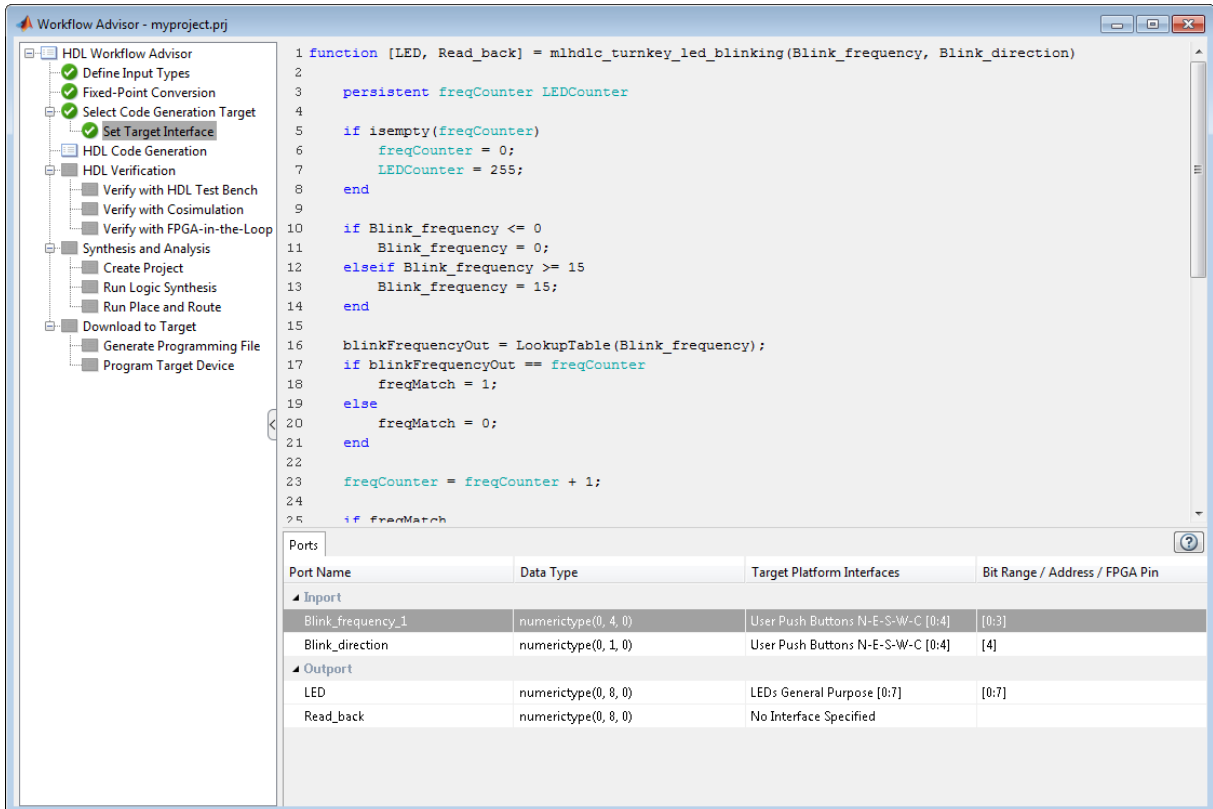
- 3 For FPGA clock frequency, for both **Input** and **System**, enter 100.



4 In the **Set Target Interface** task, map the design input and output ports to interfaces on the target device by setting the fields in the **Target Platform Interfaces** column as follows:

- Blink_frequency_1 to **User Push Buttons N-E-S-W-C [0:4]**
- Blink_direction to **User Push Buttons N-E-S-W-C [0:4]**
- LED to **LEDs General Purpose [0:7]**

You can leave the Read_back port unmapped.



The HDL Workflow Advisor applies your settings immediately.

Generate Programming File and Download To Hardware

You can generate code, perform synthesis and analysis, and download the design to the target hardware using the default settings:

- 1 For the **Synthesis and Analysis** task group, uncheck the **Skip this Step** option.
- 2 For the **Download to Target** task group, uncheck the **Skip this Step** option.

- 3** Right-click **Download to Target > Generate Programming File** and select **Run to Selected Task**.
- 4** If your target hardware is connected and ready to program, select the **Program Target Device** subtask and click **Run**.

Hardware and Software Codesign for Xilinx Zynq-7000 Platform

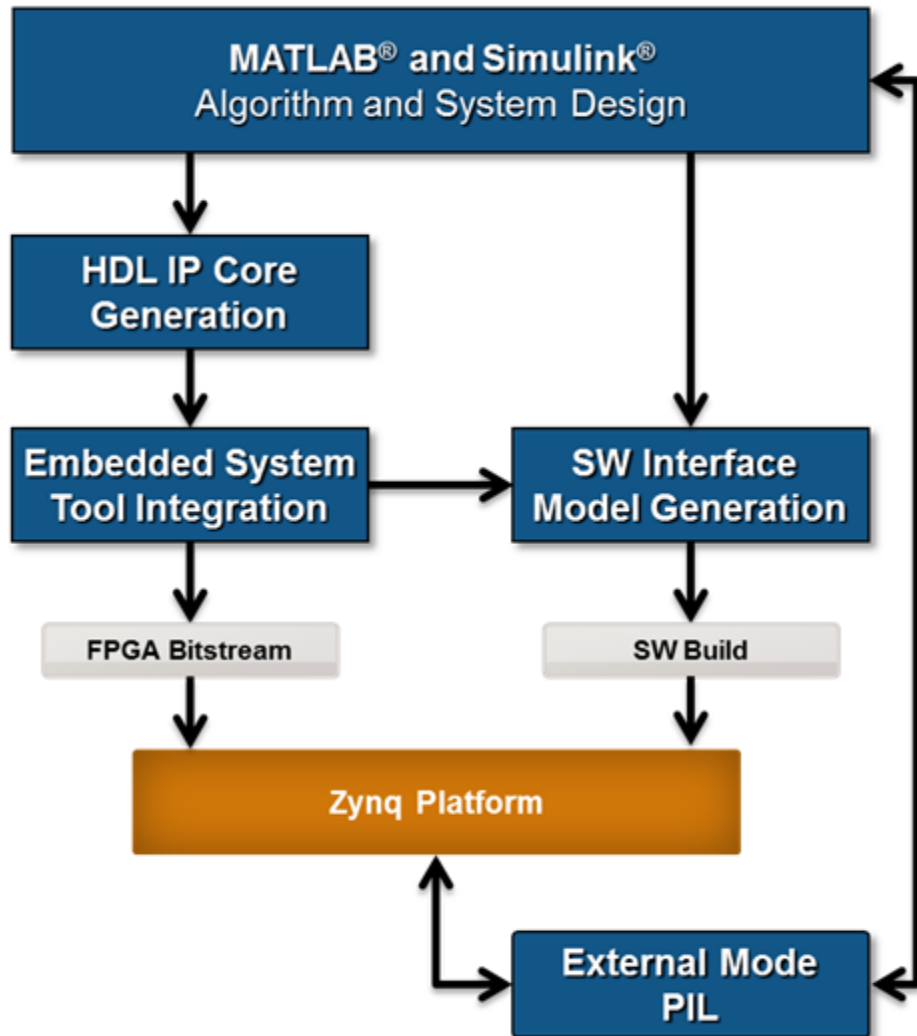
For an example that shows the hardware and software codesign workflow for the Xilinx Zynq®-7000 Platform, see `mlhdlc_tutorial_ip_core_led_blinking`.

The example shows how to:

- Set up your Zynq hardware.
- Generate an IP core for your MATLAB design.
- Include the embedded software.
- Integrate hardware and software in an EDK project and program the Xilinx Zynq All Programmable SoC.

Hardware and Software Codesign Workflow

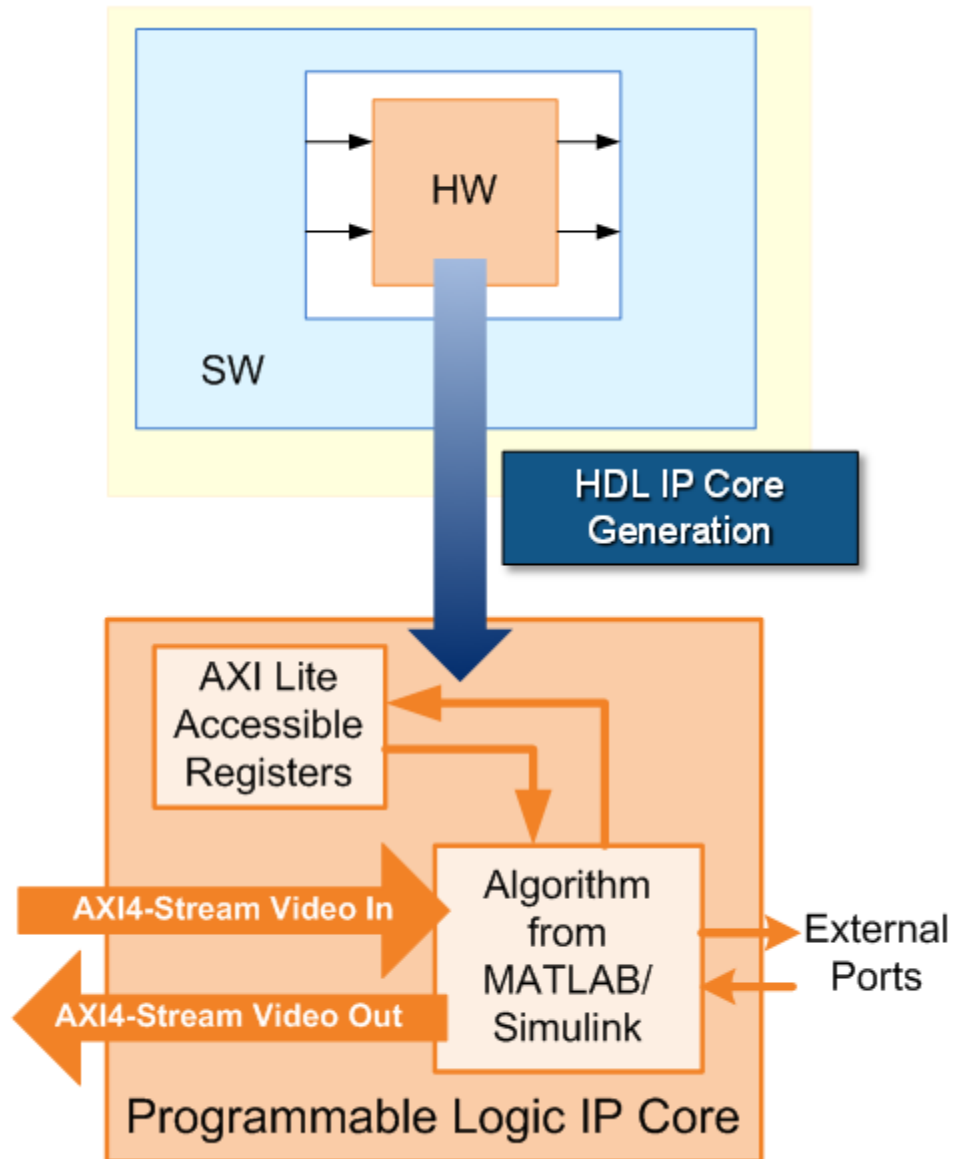
The hardware and software codesign workflow helps automate the deployment of your MATLAB and Simulink design to a Zynq-7000 All Programmable SoC. You can explore the best ways to partition and deploy your design by iterating through the following workflow.



- 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.

- 2** *HDL IP Core Generation*: Enclose the hardware part of your design in an atomic Subsystem block, and use the HDL Workflow Advisor to define and generate an HDL IP core. For more information, see “Custom IP Core Generation” on page 22-65.

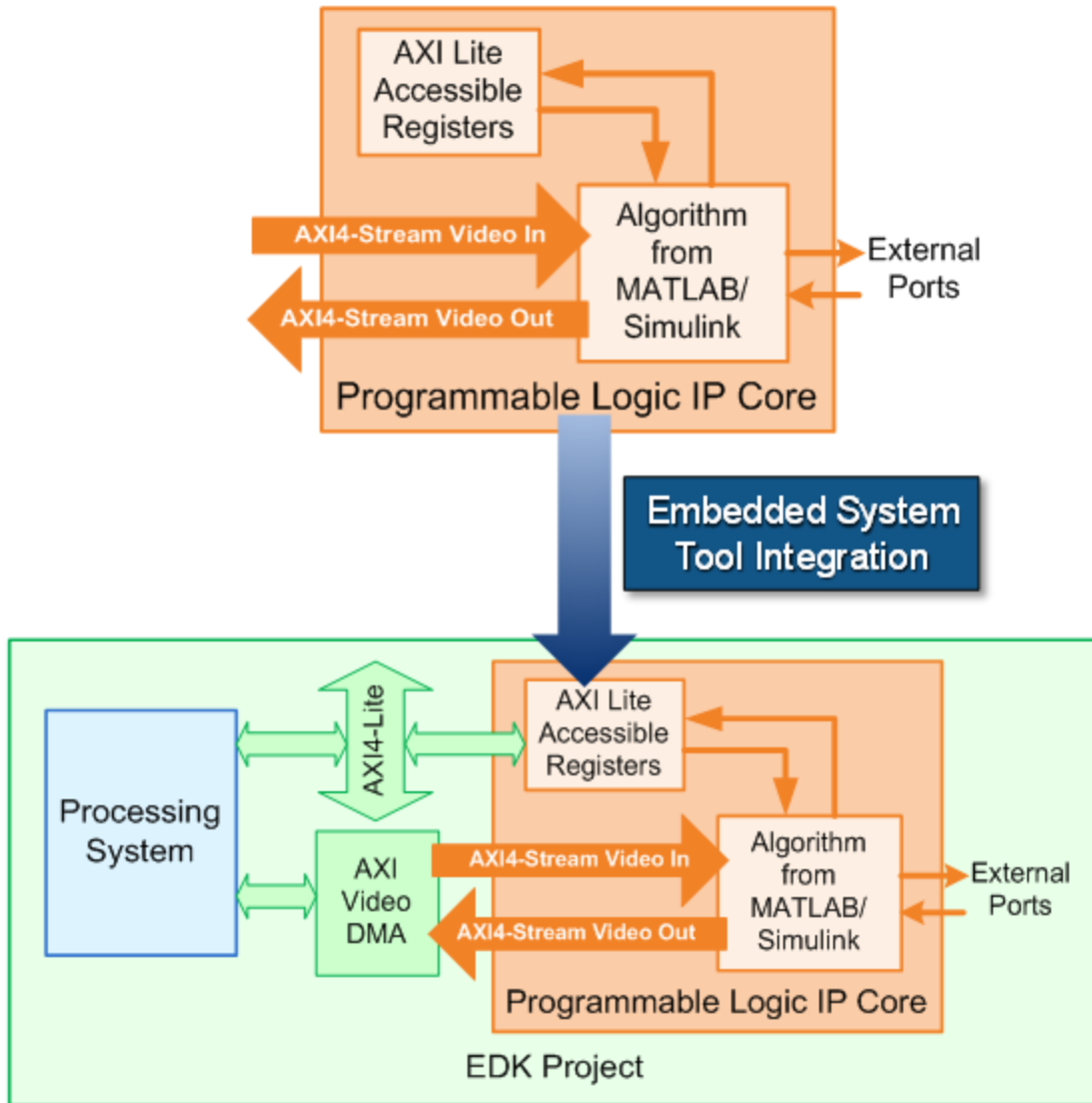
The following diagram shows a model that has been partitioned into a hardware part, in orange, and software part, in blue. HDL IP core generation creates an IP core from the hardware part of the model. The IP core hardware interface includes components such as AXI4-Lite interface-accessible registers, 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 Zynq hardware.

The *reference design* is a predefined EDK project. It contains all the elements the Xilinx software needs to deploy your design to the Zynq platform, except for the custom IP core and embedded software that you generate.

The following diagram shows the relationship between the reference design, in green, and the generated IP core, in orange.

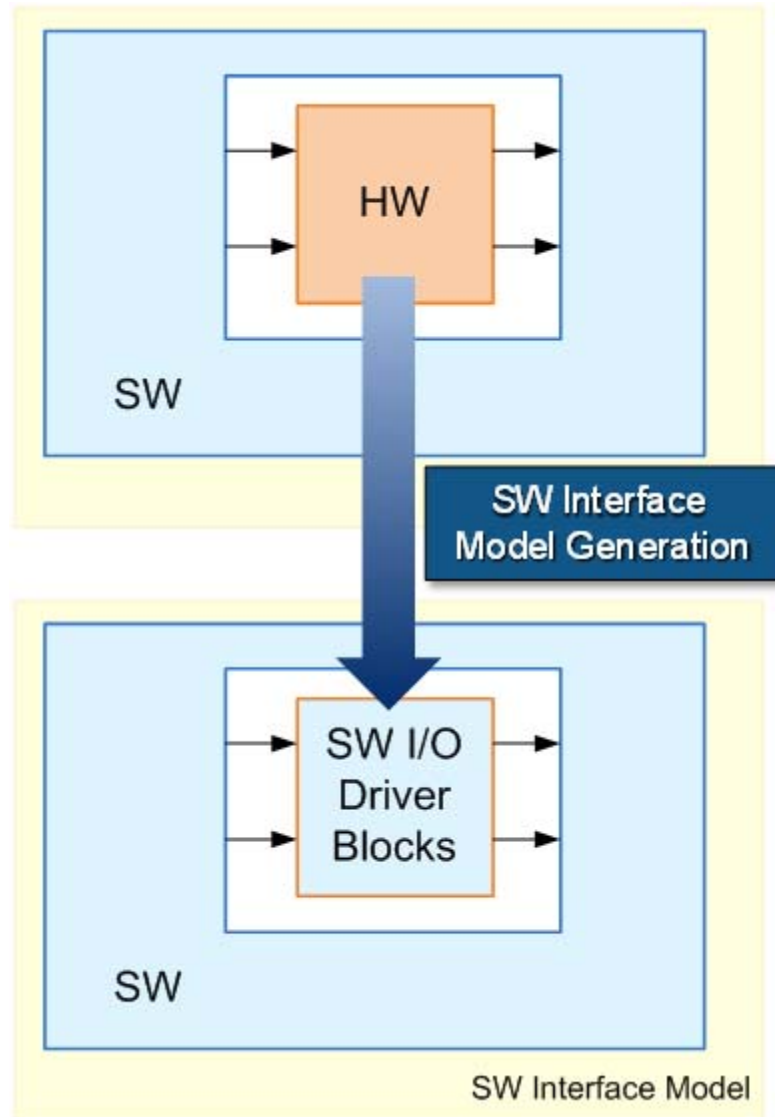


- 4** *SW Interface Model Generation* (requires a Simulink license): In the HDL Workflow Advisor, after you generate the IP core and insert it into the reference design, you can optionally generate a software interface model.

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 embedded code from the software interface model, build it, 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** *Zynq Platform and External Mode PIL*: Using the HDL Workflow Advisor, you program your FPGA bitstream to the Zynq 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.

Generate Synthesis Scripts

You can generate customized synthesis scripts for the following tools:

- 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 script (.tcl) is in the same folder as your generated HDL code.

Install Support for Altera FPGA Boards

You can use the HDL Coder FPGA Turnkey workflow with Altera FPGA boards by installing the related support package.

To install the **HDL Coder Support Package for Altera FPGA Boards** from the HDL Workflow Advisor:

- 1** In the **Select Code Generation Target** task, for **Workflow**, select **FPGA Turnkey**.
- 2** For **Platform**, select **Get more boards** to open the Support Package Installer.
- 3** In the Support Package Installer, select **Altera FPGA Boards** and follow the instructions provided by Support Package Installer to complete the installation.

Install Support for Xilinx FPGA Boards

You can use the HDL Coder FPGA Turnkey workflow with Xilinx FPGA boards by installing the related support package.

To install the **HDL Coder Support Package for Xilinx FPGA Boards** from the HDL Workflow Advisor:

- 1** In the **Select Code Generation Target** task, for **Workflow**, select **FPGA Turnkey**.
- 2** For **Platform**, select **Get more boards** to open the Support Package Installer.
- 3** In the Support Package Installer, select **Xilinx FPGA Boards** and follow the instructions provided by Support Package Installer to complete the installation.

Install Support for Xilinx Zynq-7000 Platform

You can use the HDL Coder IP core generation workflow with the Xilinx Zynq-7000 Platform by installing the related support package.

To install the **HDL Coder Support Package for Xilinx Zynq-7000 Platform** from the HDL Workflow Advisor:

- 1** In the **Select Code Generation Target** task, for **Workflow**, select **IP Core Generation**.
- 2** For **Platform**, select **Get more** to open the Support Package Installer.
- 3** In the Support Package Installer, select **Xilinx Zynq-7000** and follow the instructions provided by Support Package Installer to complete the installation.

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 7-9
- “Pipelining” on page 7-10
- “Register Inputs and Outputs” on page 7-11
- “Insert Input and Output Pipeline Registers” on page 7-12
- “Distributed Pipelining” on page 7-13
- “Pipeline MATLAB Variables” on page 7-14
- “Optimize MATLAB Loops” on page 7-16
- “Constant Multiplier Optimization” on page 7-17
- “Specify Constant Multiplier Optimization” on page 7-19
- “Distributed Pipelining for Clock Speed Optimization” on page 7-20
- “Map Matrices to Block RAMs to Reduce Area” on page 7-29
- “Resource Sharing of Multipliers to Reduce Area” on page 7-34
- “Loop Streaming to Reduce Area” on page 7-43
- “Constant Multiplier Optimization to Reduce Area” on page 7-50

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
- `hdlram System` object

Map Persistent Arrays and dsp.Delay to RAM

In this section...

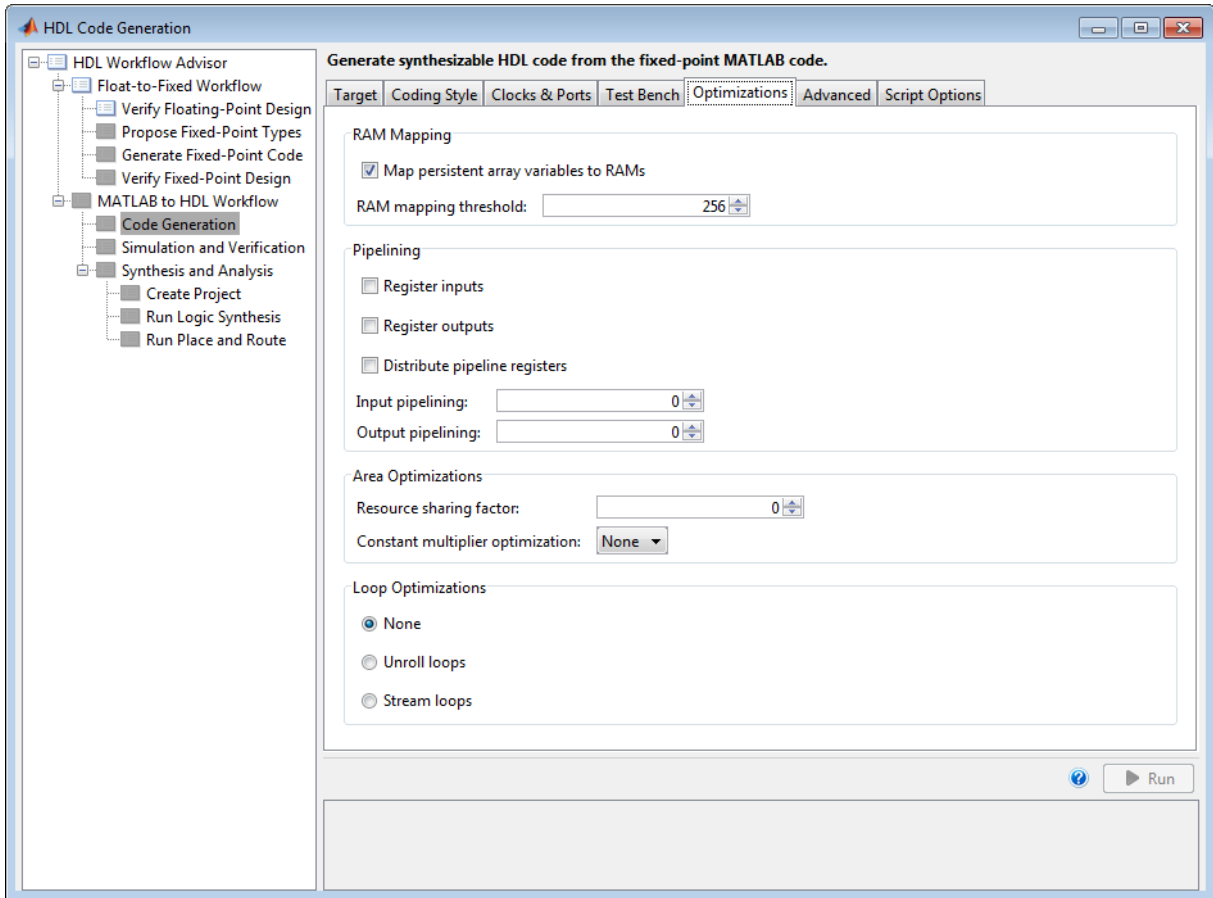
“How To Enable RAM Mapping” on page 7-3

“RAM Mapping Requirements for Persistent Arrays” on page 7-4

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

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 or `dsp.Delay` that you want to map to RAM.



RAM Mapping Requirements for Persistent Arrays

A summary of the mapping behavior for persistent arrays is in the following table.

Map Persistent Array Variables to RAMs Setting	Mapping Behavior
on	<p>A persistent array maps to a block RAM when all of the following conditions are true:</p> <ul style="list-style-type: none"> • Each read or write access is for a single element only. For example, submatrix access is 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. • 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: <pre> if (mod(i,2) > 0) a = r1(u); else r1(i) = u; end </pre> <p>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:</p> <pre> temp = mod(i,2); if (temp > 0) a = r1(u); else r1(i) = u; end </pre>

Map Persistent Array Variables to RAMs Setting	Mapping Behavior
	<ul style="list-style-type: none">• The persistent array value depends on external inputs. For example, in the following code, <code>bigarray</code> does not map to RAM because it does not depend on <code>u</code>: <pre>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;</pre>• RAMSize is greater than or equal to the RAM Mapping Threshold value. RAMSize is the product <code>NumElements * WordLength * Complexity</code>.<ul style="list-style-type: none">▪ <code>NumElements</code> is the number of elements in the array.▪ <code>WordLength</code> is the number of bits that represent the data type of the array.▪ <code>Complexity</code> is 2 for complex signals; 1 otherwise.

Map Persistent Array Variables to RAMs Setting	Mapping Behavior
	If any of the conditions are false, the persistent array maps to registers in the HDL code.
off	Persistent arrays map to registers in the generated 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 dsp.Delay System object maps to a block RAM when all of the following conditions are true:</p> <ul style="list-style-type: none"> • Length property is greater than 4. • InitialConditions 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. • RAMSize is greater than or equal to the RAM Mapping Threshold value.

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

RAM Mapping Comparison for MATLAB Code

`hdlram`, `dsp.Delay`, and persistent array variables can map to RAM, but have different attributes. The following table summarizes the differences.

Attribute	<code>hdlram</code>	<code>dsp.Delay</code>	Persistent Arrays
RAM mapping criteria	Unconditionally maps to RAM	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for <code>dsp.Delay</code> System Objects” on page 7-7.	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for Persistent Arrays” on page 7-4.
Address generation and port mapping	User specified	Automatic	Automatic
Access scheduling	User specified	Automatically inferred	Automatically inferred
Overclocking	None	None	Local multirate if access schedule requires it.
Latency with respect to simulation in MATLAB.	0	0	2 cycles if local multirate; 1 cycle otherwise.
RAM type	User specified	Dual port	Dual port

Pipelining

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” on page 7-11.

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” on page 7-12.

Variable Pipelining

Variable pipelining inserts a register at the output of a specific variable in your MATLAB code. If you know a specific variable is part of the critical path, you can add a pipeline register at the output of that variable to reduce your critical path.

To learn how to insert a pipeline register at the output of a variable, see “Pipeline MATLAB Variables” on page 7-14.

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” on page 7-10.

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” on page 7-10.

Distributed Pipelining

In this section...
“What is Distributed Pipelining?” on page 7-13
“Benefits and Costs of Distributed Pipelining” on page 7-13
“Selected Bibliography” on page 7-13

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 coder 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.

Pipeline MATLAB Variables

In this section...

“Using the HDL Workflow Advisor” on page 7-14

“Using the Command Line Interface” on page 7-14

“Limitations of MATLAB Variable Pipelining” on page 7-14

You can insert a pipeline register at the output of a specific MATLAB variable.

To learn more about pipelining, see “Pipelining” on page 7-10.

Using the HDL Workflow Advisor

To pipeline MATLAB variables from the HDL Workflow Advisor:

- 1 In the **HDL Code Generation** task, open the **Optimizations** tab.
- 2 In the **Pipeline variables** field, enter MATLAB variable names for which you want the coder to insert an output register. Separate variable names with a space.

Using the Command Line Interface

To pipeline MATLAB variables, set the `PipelineVariables` property of your 'hdl' coder config object.

For example, if you have an 'hdl' coder config object, `cfg`, pipeline the variables `v1`, `v2`, and `v3` by entering the following at the command line:

```
cfg.PipelineVariables = 'v1 v2 v3'
```

Limitations of MATLAB Variable Pipelining

The coder cannot insert a pipeline register for a MATLAB variable if it is:

- In a conditional statement or 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 use variable pipelining to add a pipeline register for `y`:

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

- In a data feedback loop. For example, in the following code, the `t` and `pvar` variables cannot be pipelined:

```
persistent pvar;
t = u + pvar;
pvar = t + v;
```

Optimize MATLAB Loops

In this section...
“How to Optimize MATLAB Loops” on page 7-16
“Limitations for MATLAB Loop Optimization” on page 7-16

How to Optimize MATLAB Loops

To select a loop optimization in the Workflow Advisor:

- 1 Open the Workflow Advisor.
- 2 In the left pane, select **MATLAB HDL Coder Workflow > MATLAB to HDL Workflow > Code Generation**.
- 3 Select the **Optimizations** tab.
- 4 For **Loop Optimizations**, select **None**, **Unroll Loops**, or **Stream Loops**.

To learn more about loop streaming and loop unrolling, see “Loop Optimization” on page 15-41.

Limitations for MATLAB Loop Optimization

The 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.

The 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.

Constant Multiplier Optimization

The **Constant multiplier optimization** option enables you to specify use of canonic 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, the 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 canonic 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	When you specify this option, the 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.

Constant Multiplier Optimization Value	Description
	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).

To learn how to specify constant multiplier optimization, see “Specify Constant Multiplier Optimization” on page 7-19.

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” on page 7-17.

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.m';  
testbench_name = 'mlhdlc_fir_tb.m';
```

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', 'hdlcoderdemo');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
```

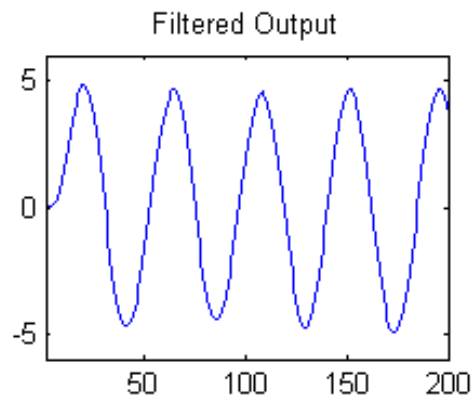
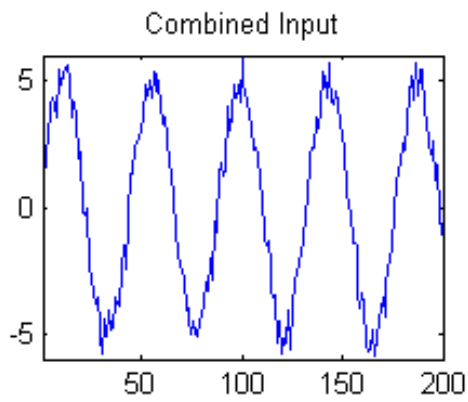
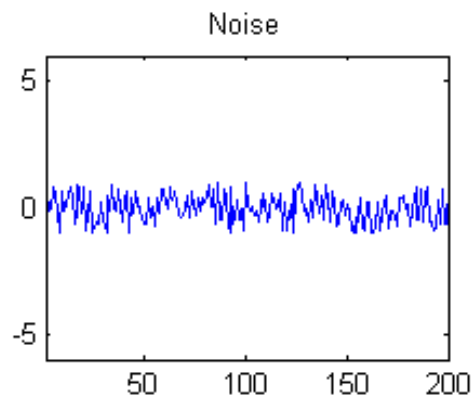
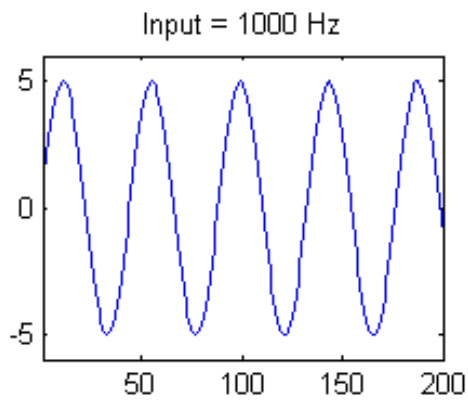
```
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

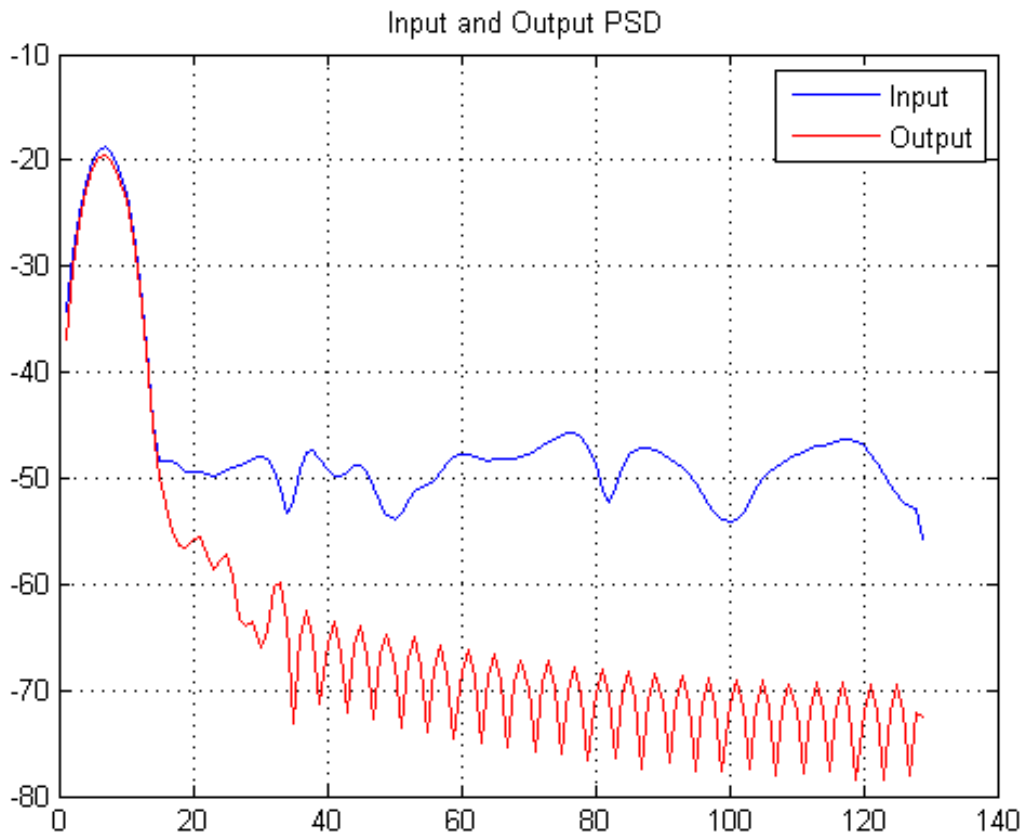
copyfile(fullfile(mlhdlc_demo_dir, design_name), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), mlhdlc_temp_dir);
```

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.

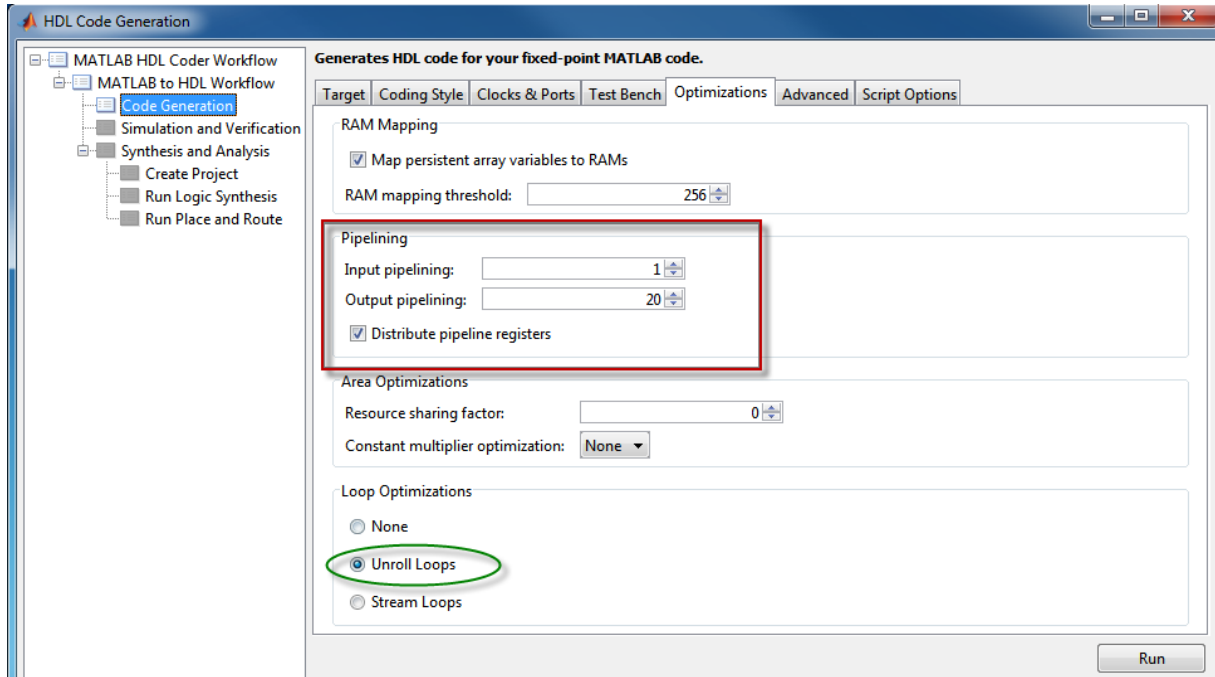
Distributed Pipelining

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.



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

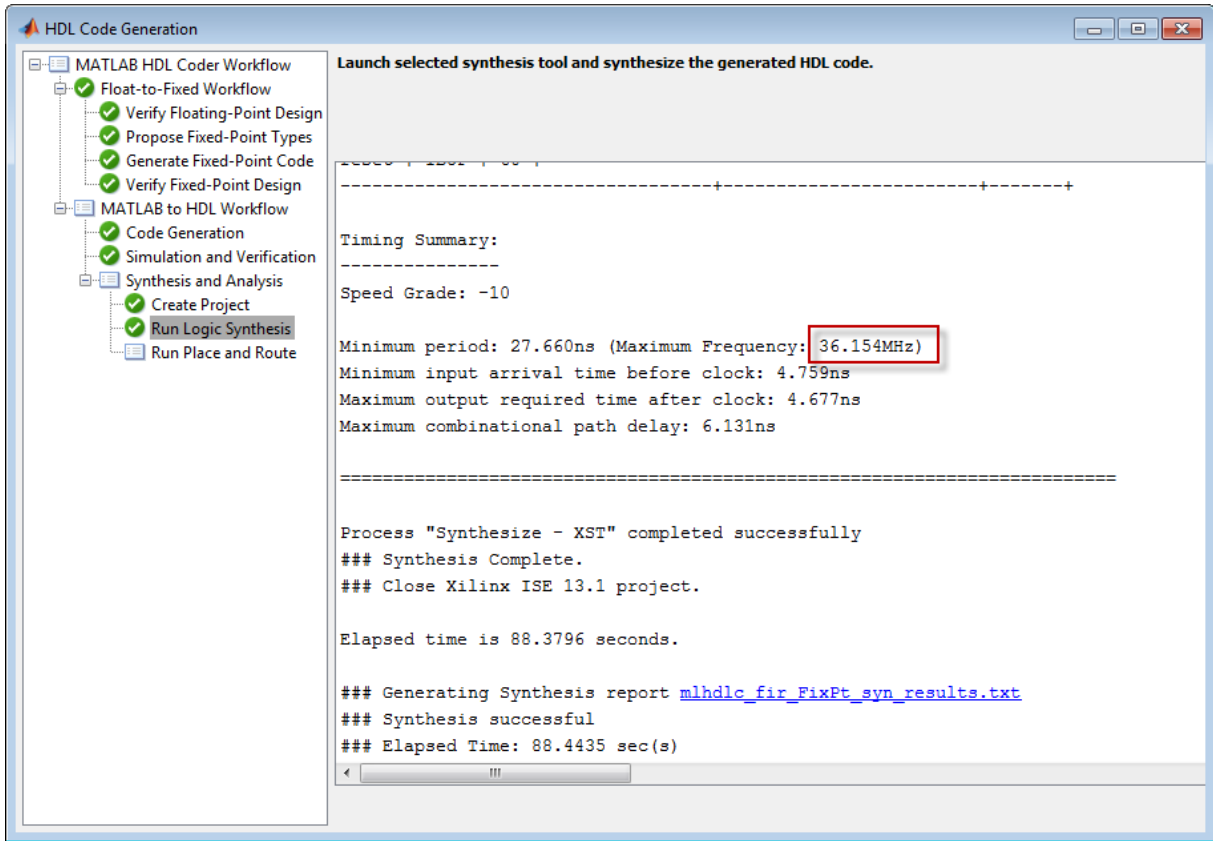
Chip family: Virtex4

Device name: xc4vsx35

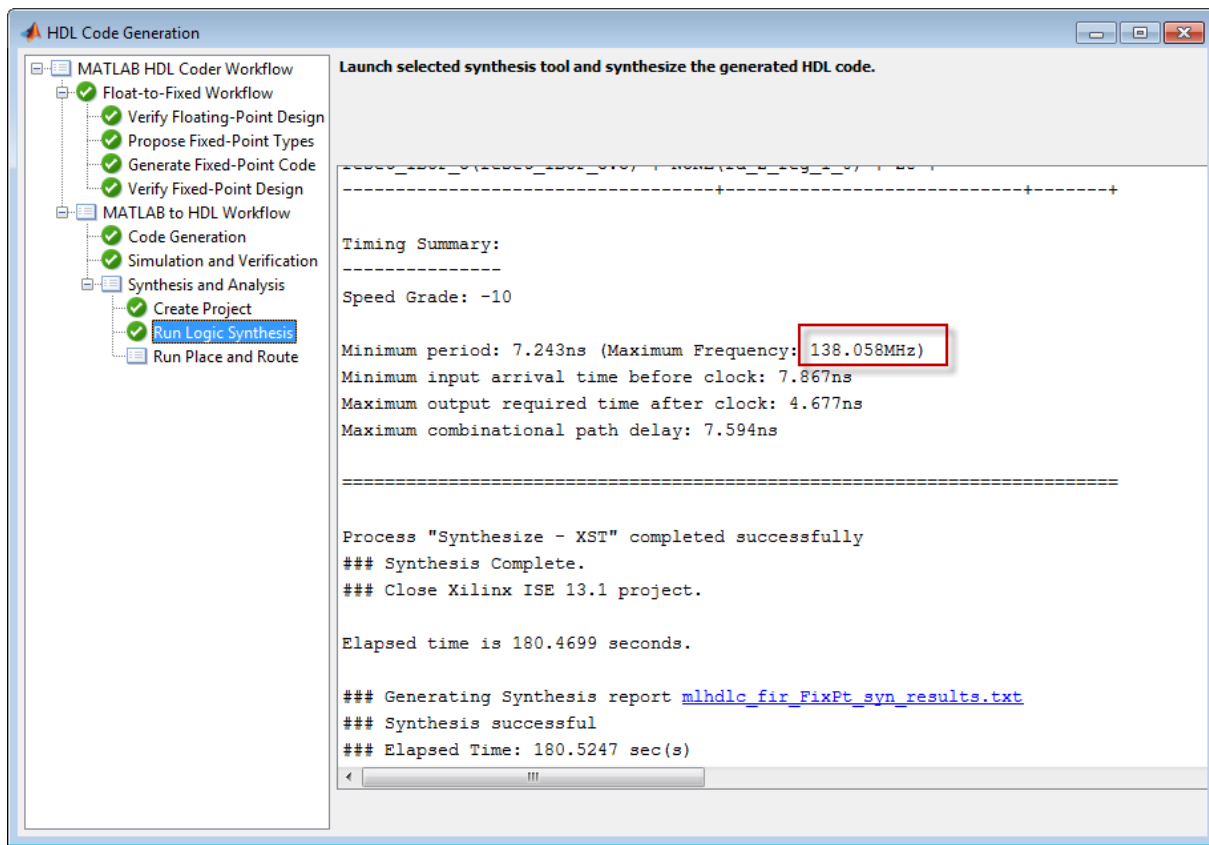
Package name: ff668

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', 'hdlcoderdemo');
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.m';  
testbench_name = 'mlhdlc_sobel_tb.m';
```

- 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', 'hdlcoderdemo');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_sobel'];
```

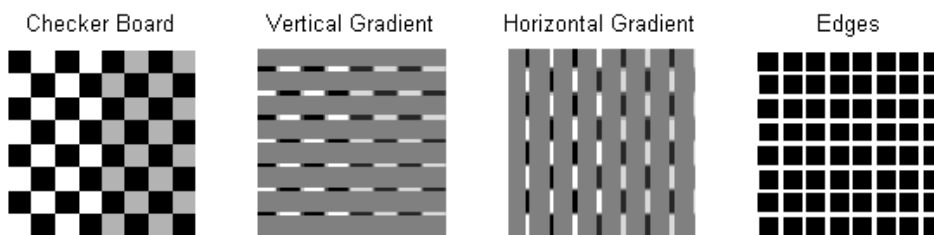
```
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);
```

```
% copy the design files to the temporary directory  
copyfile(fullfile(mlhdlc_demo_dir, design_name), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), 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
```

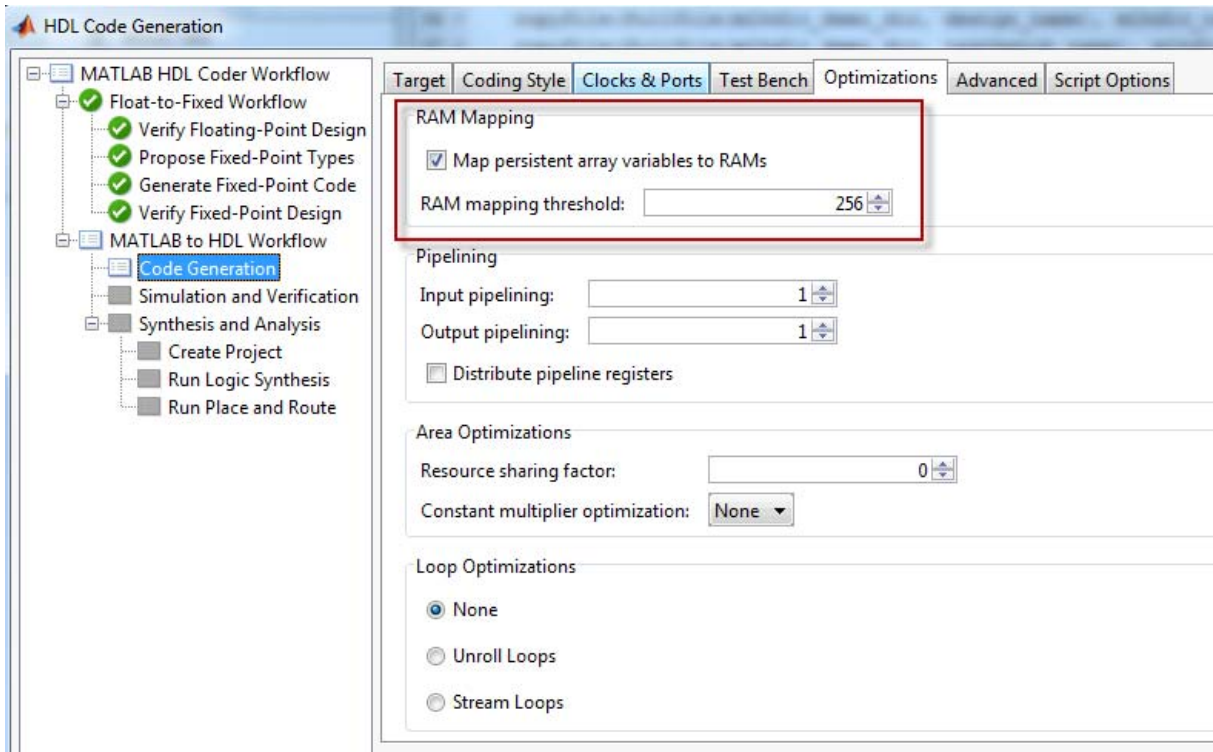
Next, add the file 'mlhdlc_sobel.m' to the project as the MATLAB function, and 'mlhdlc_sobel_tb.m' as the MATLAB test bench.

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.

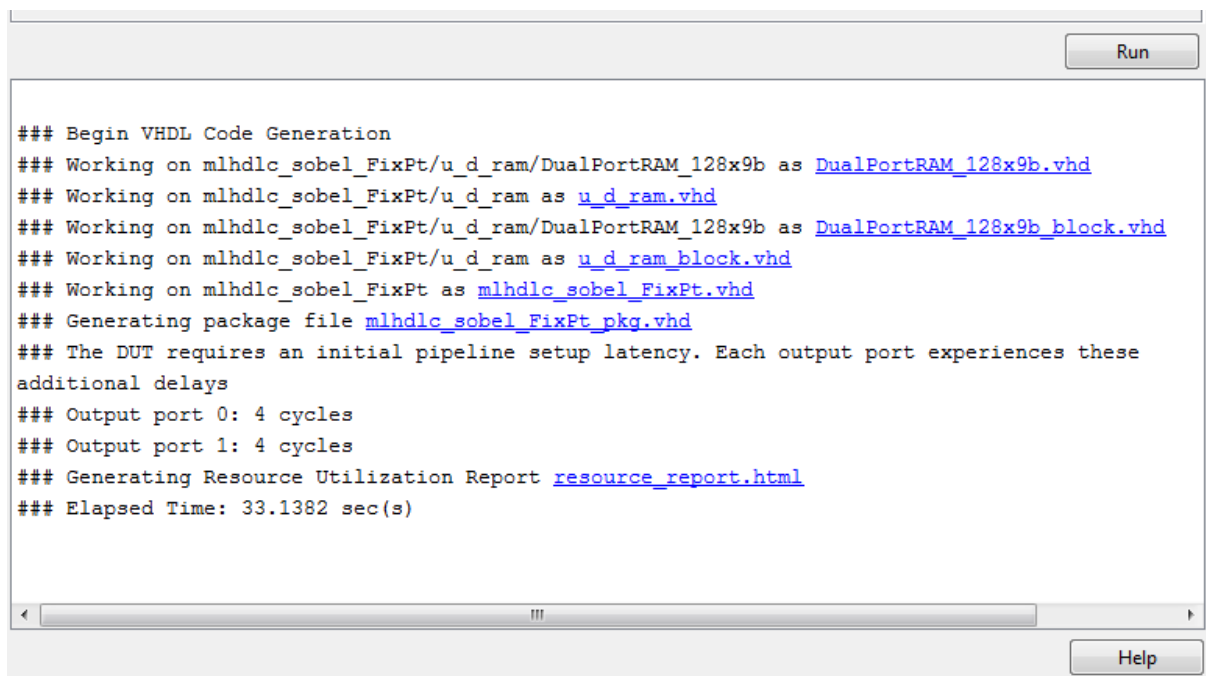


Run Fixed-Point Conversion and HDL Code Generation

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.



The screenshot shows a log window with a 'Run' button at the top right and a 'Help' button at the bottom right. The log text is as follows:

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

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

Examine the Resource Report

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

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

Additional Notes on RAM Mapping

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

Clean up the Generated Files

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

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemo');
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.

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.m';
testbench_name = 'mlhdlc_sharing_tb.m';
```

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', 'hdlcoderdemo');
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), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), mlhdlc_temp_dir);
```

Create a New HDL Coder Project

Run the following command to create a new project:

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

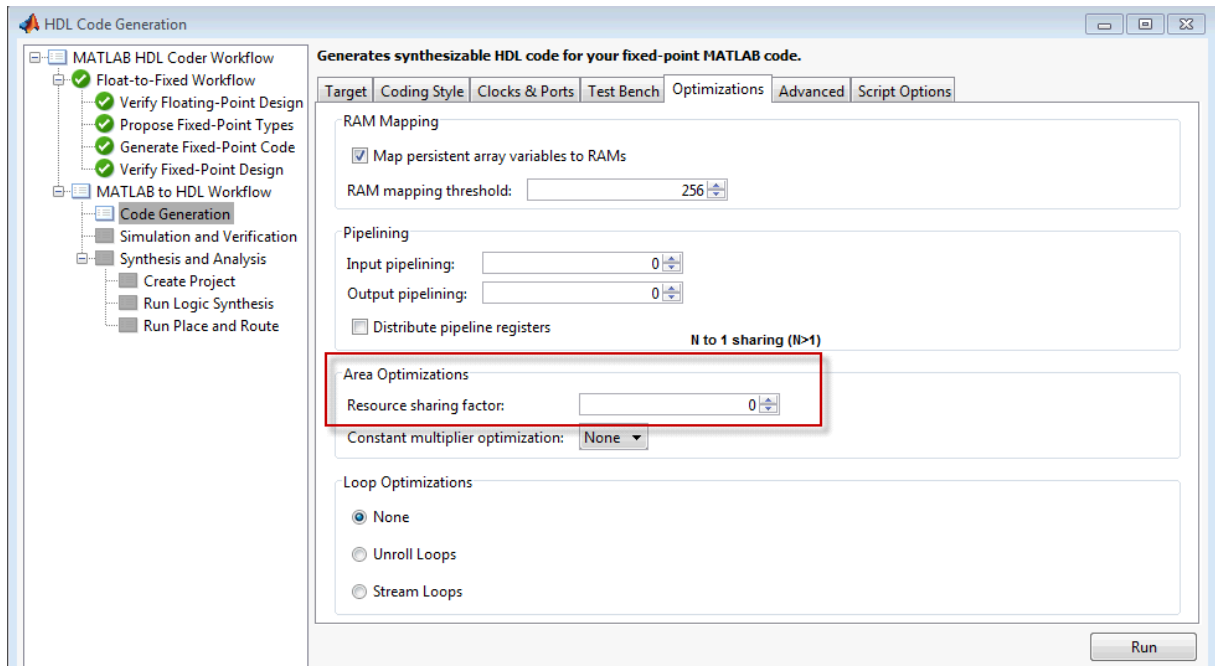
Next, add the file 'mlhdlc_sharing.m' to the project as the MATLAB Function and 'mlhdlc_sharing_tb.m' as the MATLAB Test Bench.

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

Realize an N-to-1 Mapping of Multipliers

Turn on the resource sharing optimization by setting the 'Resource Sharing Factor' to a positive integer value.

This parameter specifies 'N' in the N-to-1 hardware mapping. Choose a value of $N > 1$.



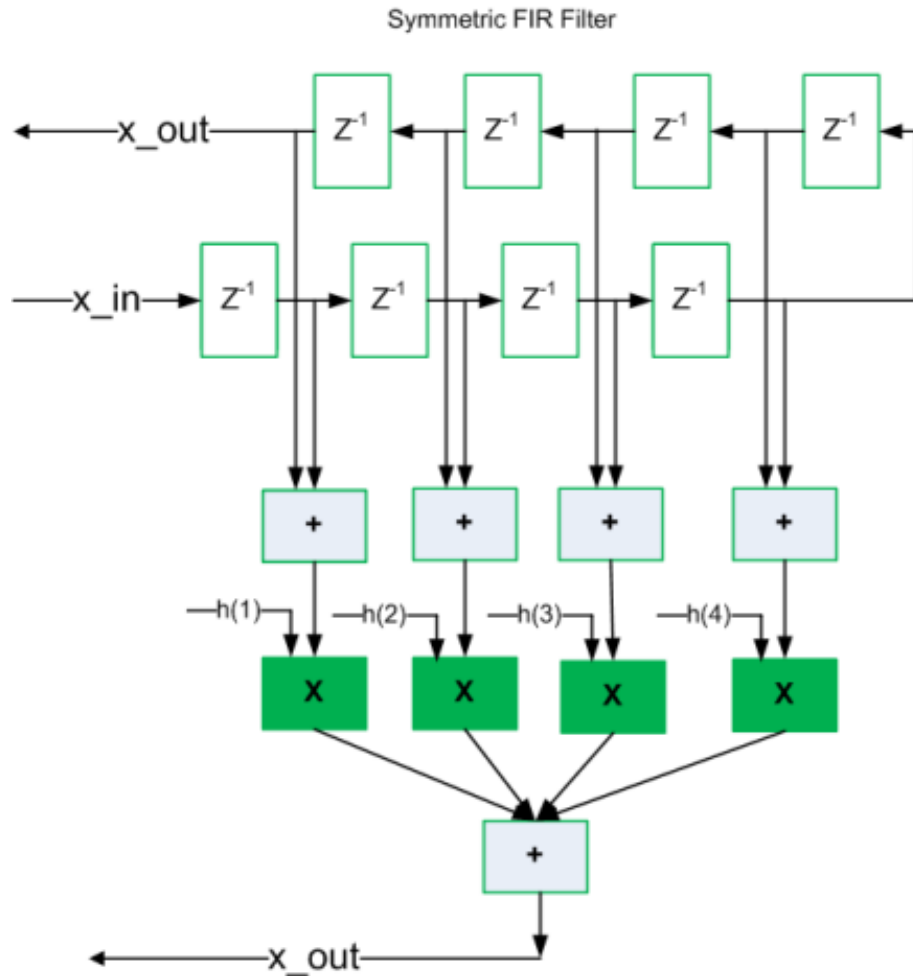
Examine the Resource Report

There are 4 multiplication operators in this example design. Generating HDL with a 'SharingFactor' of 4 will result in only one multiplier in the generated code.

Multipliers	1
Adders/Subtractors	7
Registers	29
RAMs	0
Multiplexers	12

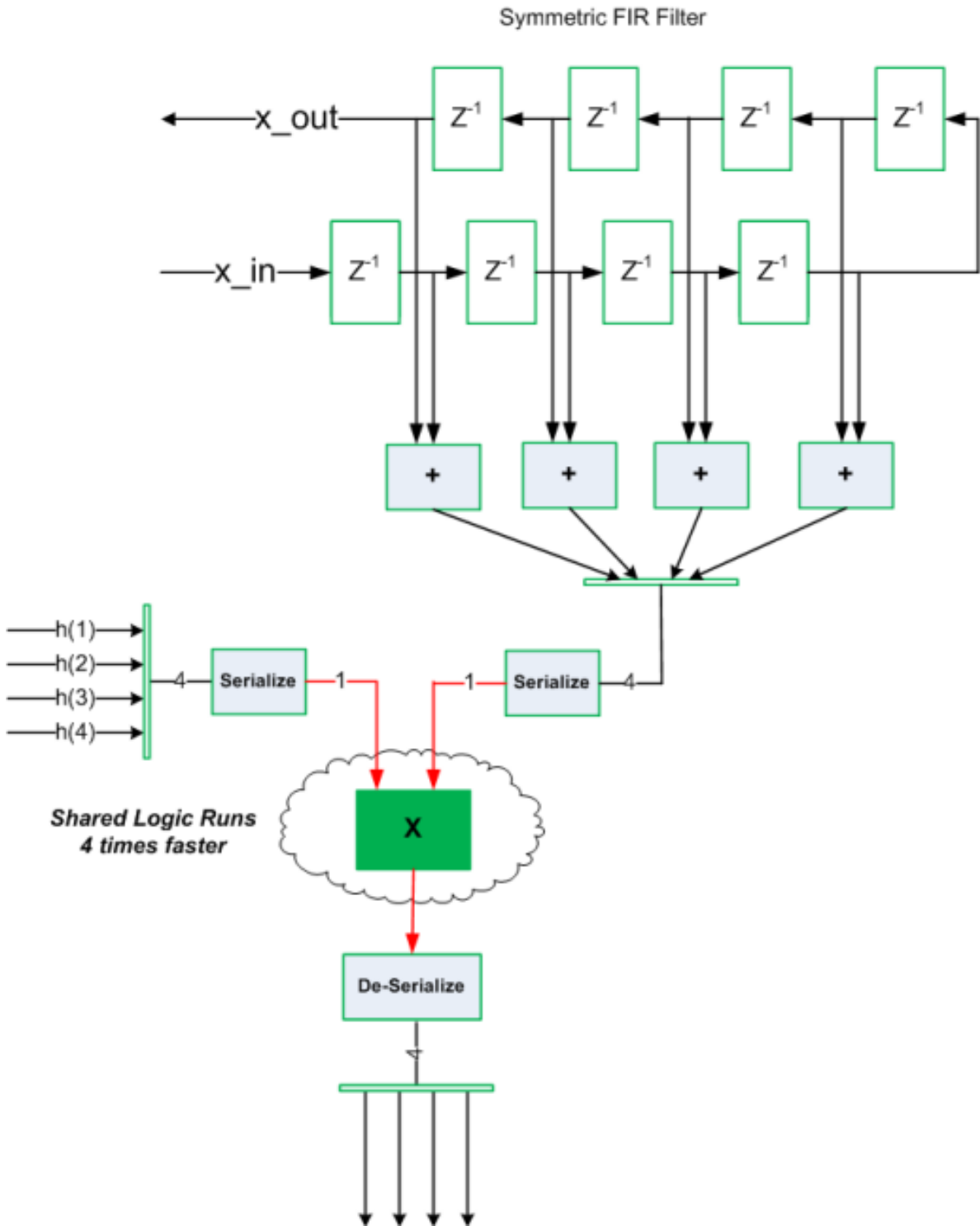
Sharing Architecture

The following figure shows how the algorithm is implemented in hardware when we synthesize the generated code without turning on the sharing optimization.



The following figure shows the sharing architecture automatically implemented by HDL Coder when the sharing optimization option is turned on.

The inputs to the shared multiplier are time-multiplexed at a faster rate (in this case 4x faster and denoted in red). The outputs are then routed to the respective consumers at a slower rate (in green).



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.

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', 'hdlcoderdemo');
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.m';  
testbench_name = 'mlhdlc_fir_tb.m';
```

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.

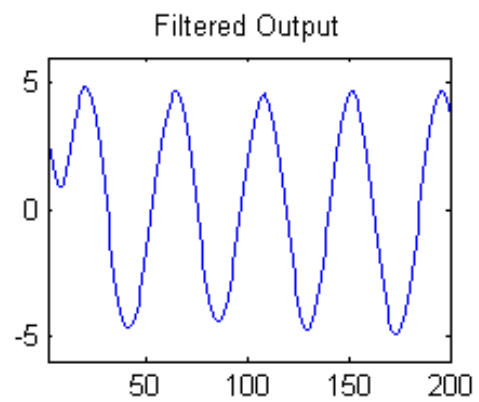
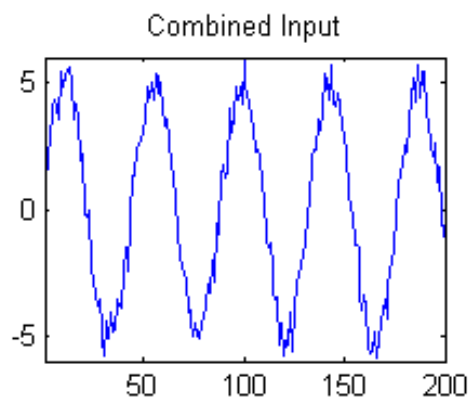
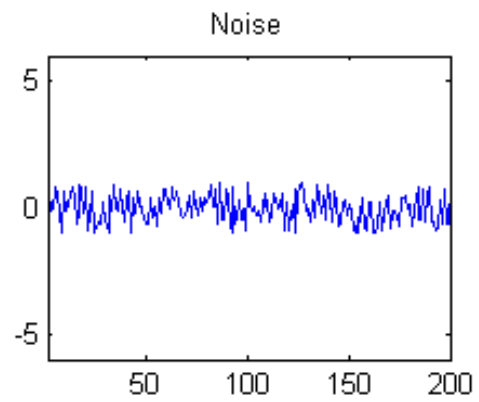
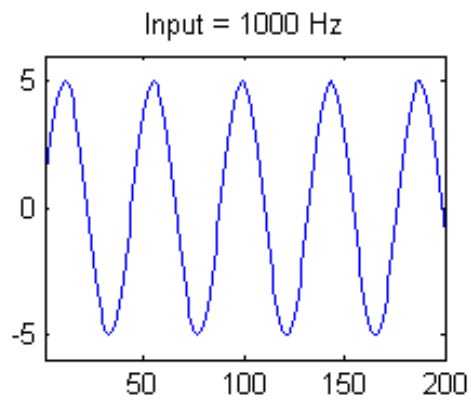
```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemo');  
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];  
  
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
```

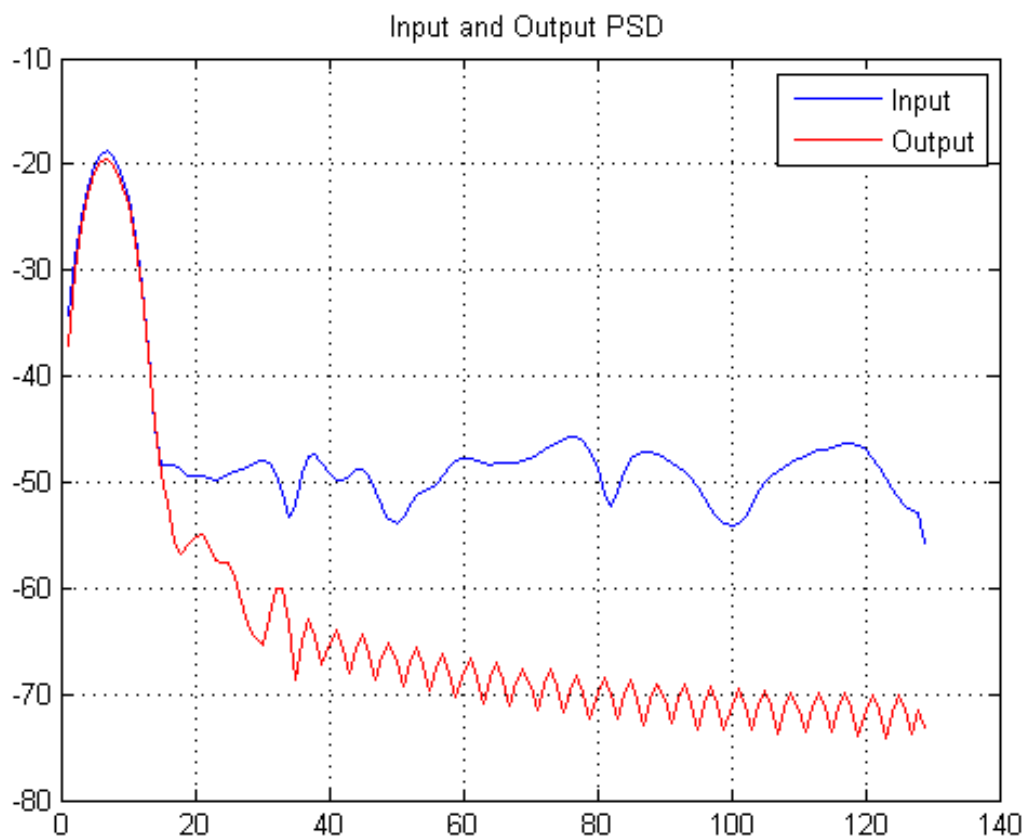
```
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);  
  
copyfile(fullfile(mlhdlc_demo_dir, design_name), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), mlhdlc_temp_dir);
```

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
```

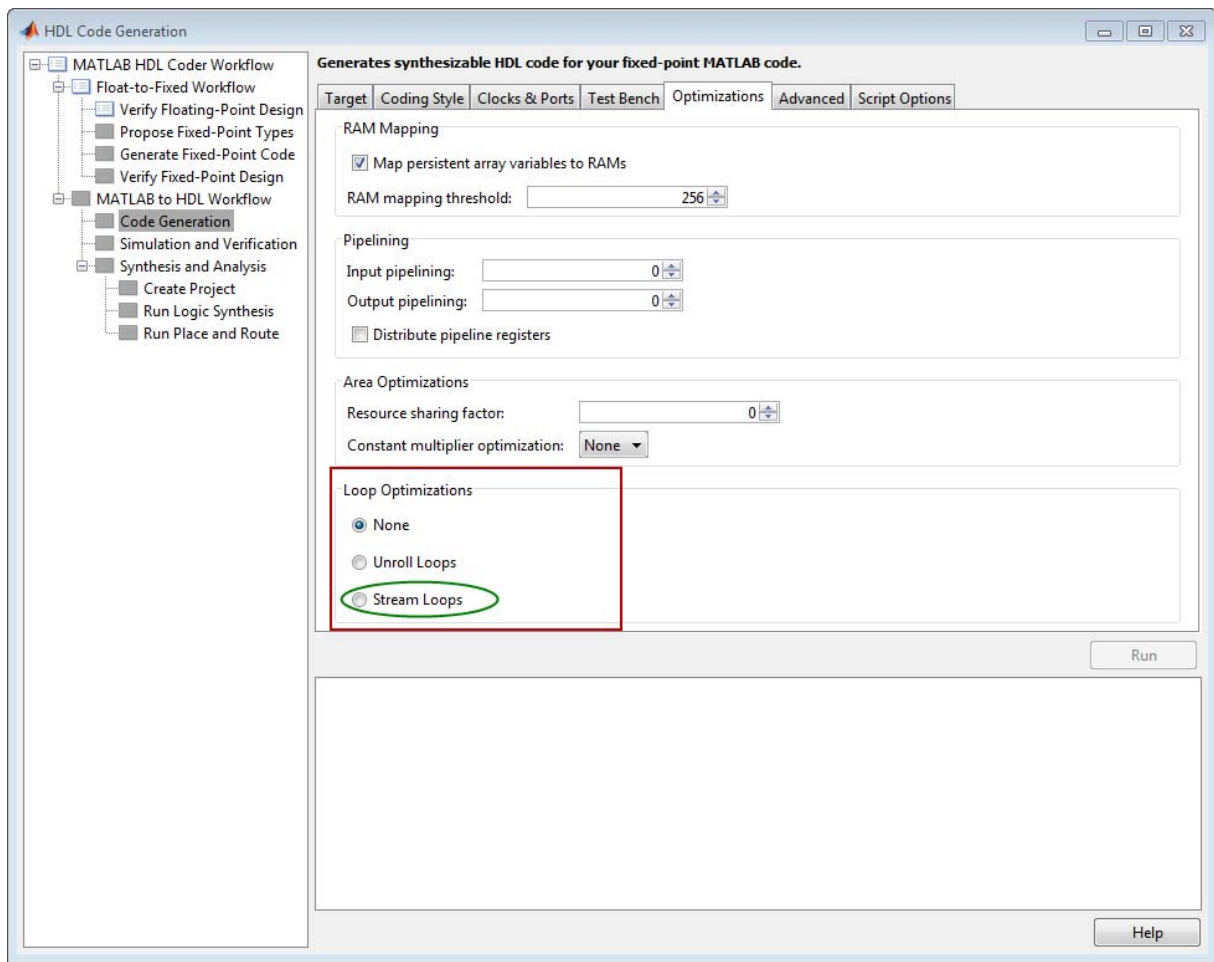
Next, add the file 'mlhdlc_fir.m' to the project as the MATLAB Function and 'mlhdlc_fir_tb.m' as the MATLAB Test Bench.

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', 'hdlcoderdemo');
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:

$$\begin{aligned}
 y &= 231 * x \\
 &= (11100111) * x && \% 231 \text{ in binary form} \\
 &= (1001'01001') * x && \% 231 \text{ in signed digit form} \\
 &= (256 - 32 + 8 - 1) * x && \% \\
 &= (x \ll 8) - (x \ll 5) + (x \ll 3) - x && \% \text{ cost of CSD: 3 Adders}
 \end{aligned}$$

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:

$$\begin{aligned}
 y &= 231 * x \\
 &= (7 * 33) * x \\
 &= (x \ll 8 - x) * (x \ll 5 + x) && \% \text{ cost of FCSD: 2 Adders}
 \end{aligned}$$

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 = $2 \times 9 \times 5$, 93 = 3×31 , 99 = 3×33 , 102 = $2 \times 3 \times 17$, 105 = 7×15 , 150 = $2 \times 5 \times 15$, 153 = 9×17 , 155 = 5×31 , 165 = 5×33 , 170 = $2 \times 5 \times 17$, 180 = $4 \times 5 \times 9$, 186 = $2 \times 3 \times 31$, 189 = 7×9 , 195 = 3×65 , 198 = $2 \times 3 \times 33$, 204 = $4 \times 3 \times 17$, 210 = $2 \times 7 \times 15$, 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.m';  
testbench_name = 'mlhdlc_csd_tb.m';
```

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', 'hdlcoderdemo');  
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), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), 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:

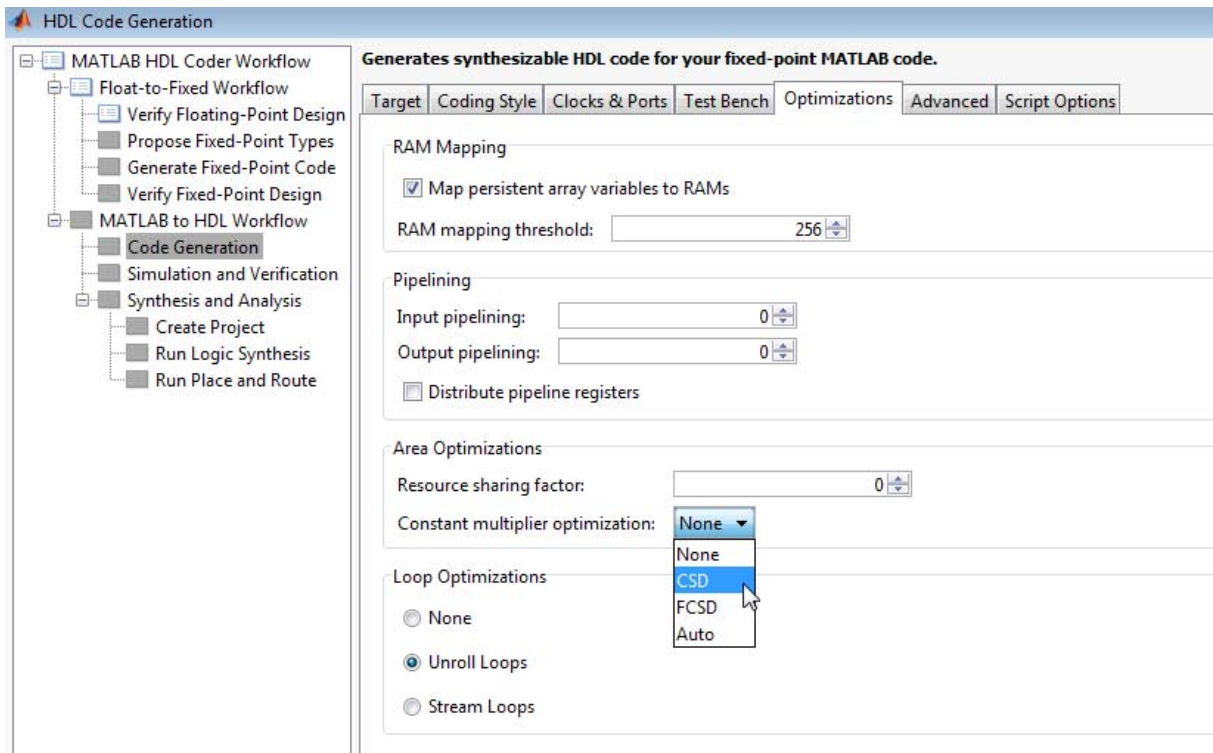
```
coder -hdlcoder -new csd_prj
```

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 DOWNTO 13);
345

```

Take a look at the resource report for adder and multiplier usage without the CSD optimization.

Multipliers	4
Adders/Subtractors	7
Registers	23
RAMs	0
Multiplexers	0

Generate Code with CSD Optimization

- 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 --'mlhdlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
331 -- CSD Encoding (2194) : 0100010010010; Cost (Adders) = 3
332 p22y_out_mul_temp_1 <= - (((resize(a1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a1 & '0' &
333 p22y_out_add_cast <= resize(p22y_out_mul_temp_1, 29);
334 -- CSD Encoding (1373) : 0101011001'01; Cost (Adders) = 5
335 p22y_out_mul_temp_1 <= - (((((resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a2 & '0' & '0' &
336 p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
337 p22y_out_add_temp_1 <= p22y_out_add_cast + p22y_out_add_cast_1;
338 p22y_out_add_cast_2 <= resize(p22y_out_add_temp_1, 30);
339 -- CSD Encoding (3319) : 0110100001'001'; Cost (Adders) = 4
340 p22y_out_mul_temp_2 <= (((resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a3 & '0' & '0' &
341 p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
342 -- CSD Encoding (6658) : 01101000000010; Cost (Adders) = 3
343 p22y_out_mul_temp_3 <= ((resize(a4 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a4 & '0' & '0' &
344 p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
345 p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
346 p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
347 p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
348 y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);

```

Examine the code with comments that outline the CSD encoding for all the constant multipliers.

Look at the resource report and notice that with the CSD optimization, the number of multipliers is reduced to zero and multipliers are replaced by shifts and adders.

Multipliers	0
Adders/Subtractors	24
Registers	23
RAMs	0
Multiplexers	0

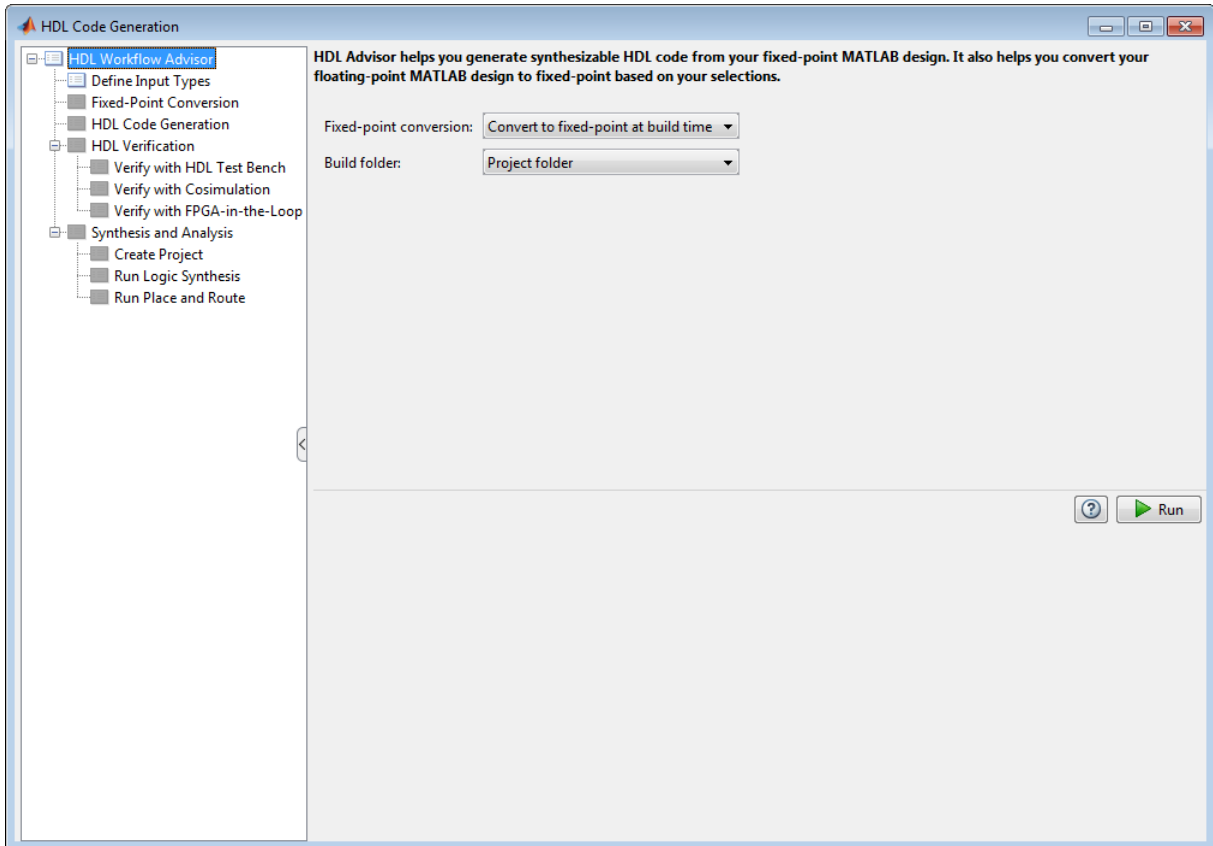
Generate Code with FCSD Optimization

- 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.
- 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.

HDL Workflow Advisor Reference

- “HDL Workflow Advisor” on page 8-2
- “MATLAB to HDL Code and Synthesis” on page 8-7

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.

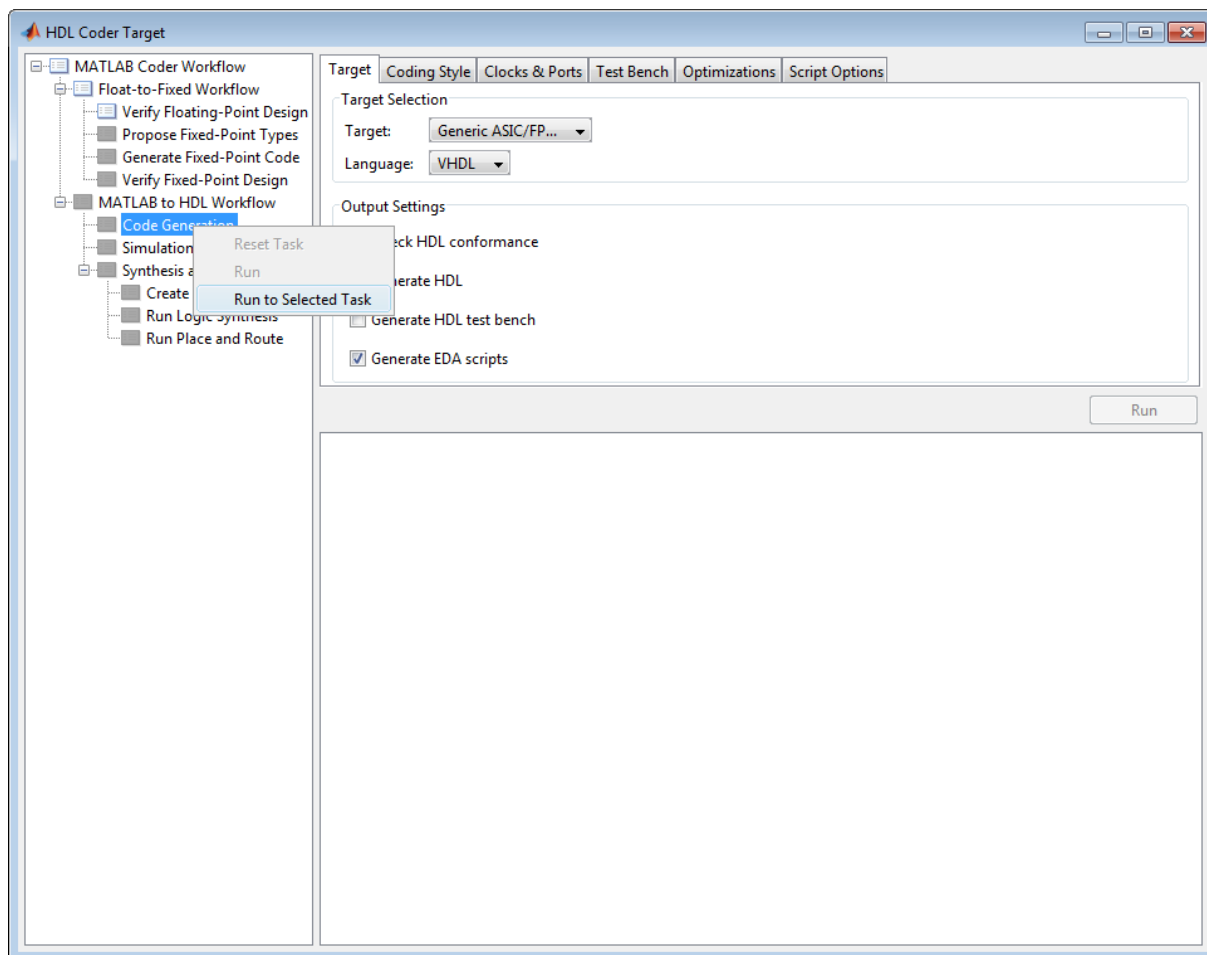
4 Once you have fixed a **Warning** or **Failed** task, rerun the task by clicking **Run**.

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 8-7

“Code Generation: Target Tab” on page 8-7

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

“Code Generation: Clocks and Ports Tab” on page 8-10

“Code Generation: Test Bench Tab” on page 8-12

“Code Generation: Optimizations Tab” on page 8-15

“Simulation and Verification” on page 8-16

“Synthesis and Analysis” on page 8-17

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. The 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

The 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, the 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, the 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, the 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, the 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, the 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

- Chapter 9, “Code Generation Options in the HDL Coder Dialog Boxes”
- Chapter 10, “Specifying Block Implementations and Parameters for HDL Code Generation”
- Chapter 11, “Guide to Supported Blocks and Block Implementations”
- Chapter 12, “Generating HDL Code for Multirate Models”
- Chapter 13, “The `hdldemo1ib` Block Library”
- Chapter 14, “Generating Bit-True Cycle-Accurate Models”
- Chapter 15, “Optimization”
- Chapter 16, “Code Generation Reports, HDL Compatibility Checker, Block Support Library, and Code Annotation”
- Chapter 17, “HDL Coding Standards”
- Chapter 18, “Interfacing Subsystems and Models to HDL Code”
- Chapter 19, “Stateflow HDL Code Generation Support”
- Chapter 20, “Generating HDL Code with the MATLAB Function Block”
- Chapter 21, “Generating Scripts for HDL Simulators and Synthesis Tools”
- Chapter 22, “Using the HDL Workflow Advisor”
- Chapter 23, “HDL Test Bench”
- Chapter 24, “FPGA Board Customization”
- Chapter 25, “HDL Workflow Advisor Tasks”

- Chapter 26, “Code Generation Control Files”

Code Generation Options in the HDL Coder Dialog Boxes

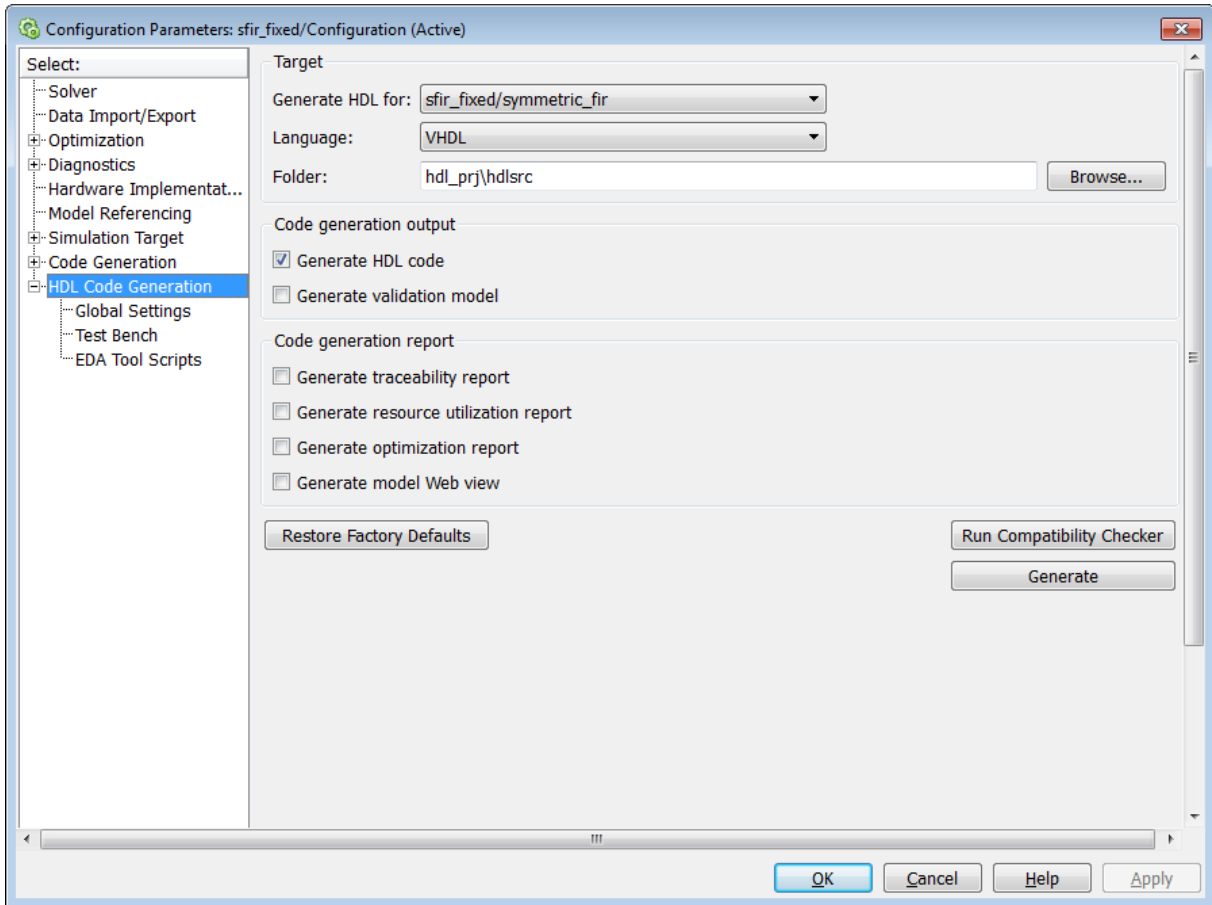
- “Set HDL Code Generation Options” on page 9-2
- “HDL Code Generation Pane: General” on page 9-8
- “HDL Code Generation Pane: Global Settings” on page 9-22
- “HDL Code Generation Pane: Test Bench” on page 9-78
- “HDL Code Generation Pane: EDA Tool Scripts” on page 9-106

Set HDL Code Generation Options

In this section...
“HDL Code Generation Options in the Configuration Parameters Dialog Box” on page 9-2
“HDL Code Generation Options in the Model Explorer” on page 9-3
“Code Menu” on page 9-4
“HDL Code Options in the Block Context Menu” on page 9-5
“The HDL Block Properties Dialog Box” on page 9-6

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.



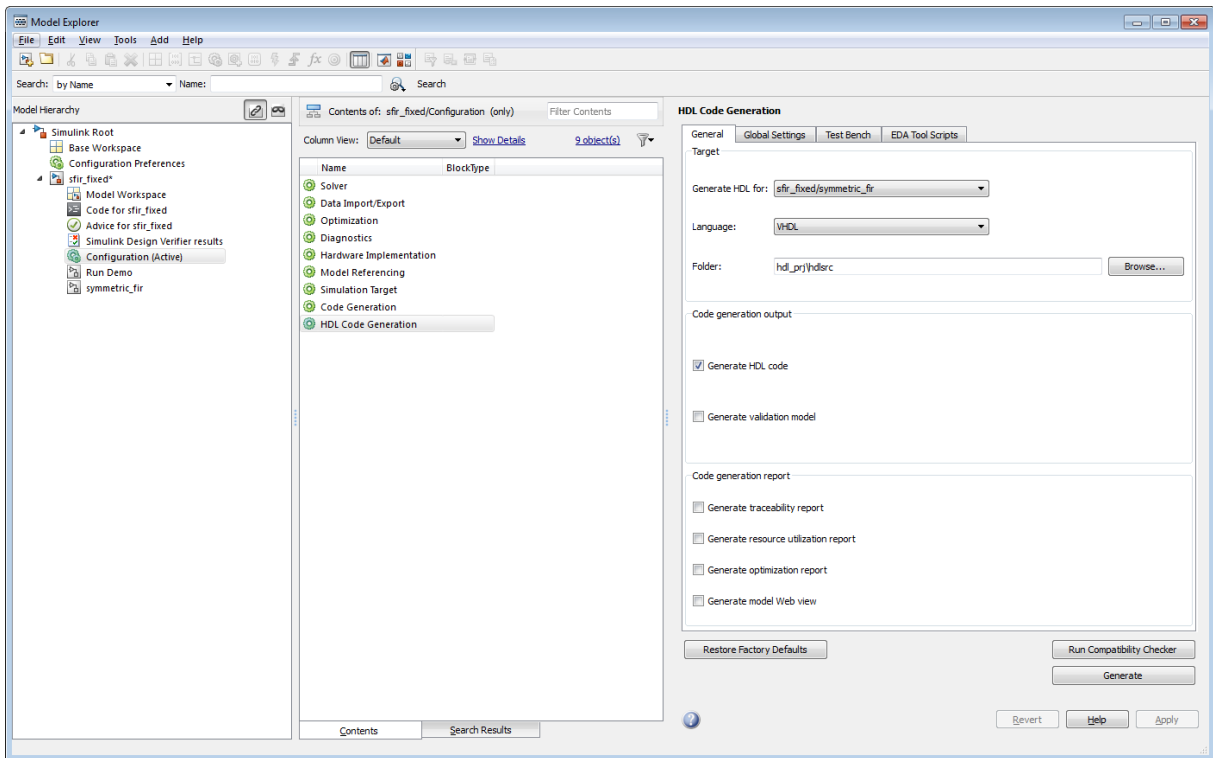
Note When the **HDL Code Generation** pane of the Configuration Parameters dialog box appears, clicking the **Help** button displays general help for the Configuration Parameters dialog box.

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 the 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 16-46.

HDL Code Options in the Block Context Menu

When you right-click a block that the 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 (checkhdl) on the subsystem.	Available only for subsystems.
Generate HDL for Subsystem	Runs the HDL code generator (makehdl) and generates code for the subsystem.	Available only for subsystems.
HDL Coder Properties	Opens the Configuration Parameters dialog box, with the top-level HDL Code Generation pane selected.	Available for blocks or subsystems.
HDL Block Properties	Opens a block properties dialog box for the block or subsystem. See “The HDL Block Properties Dialog Box” on page 9-6 for more information.	Available for blocks or subsystems.
HDL Workflow Advisor	Opens the HDL Workflow Advisor for the subsystem.	Available only for subsystems.
Navigate to Code	Activates the HTML code generation report window, displaying the beginning of the code generated for the selected block or subsystem. See “Tracing from Model to Code” on page 16-22 for more information.	Enabled when both code and a traceability report have been generated for the block or subsystem.

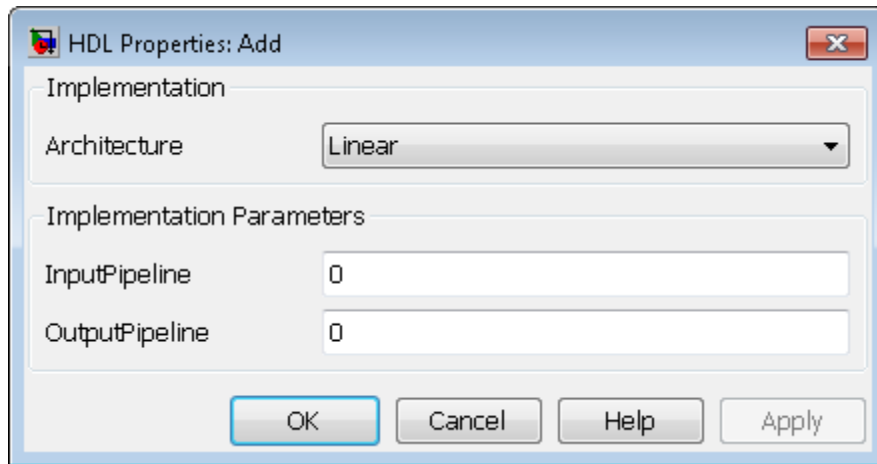
The HDL Block Properties Dialog Box

The 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.

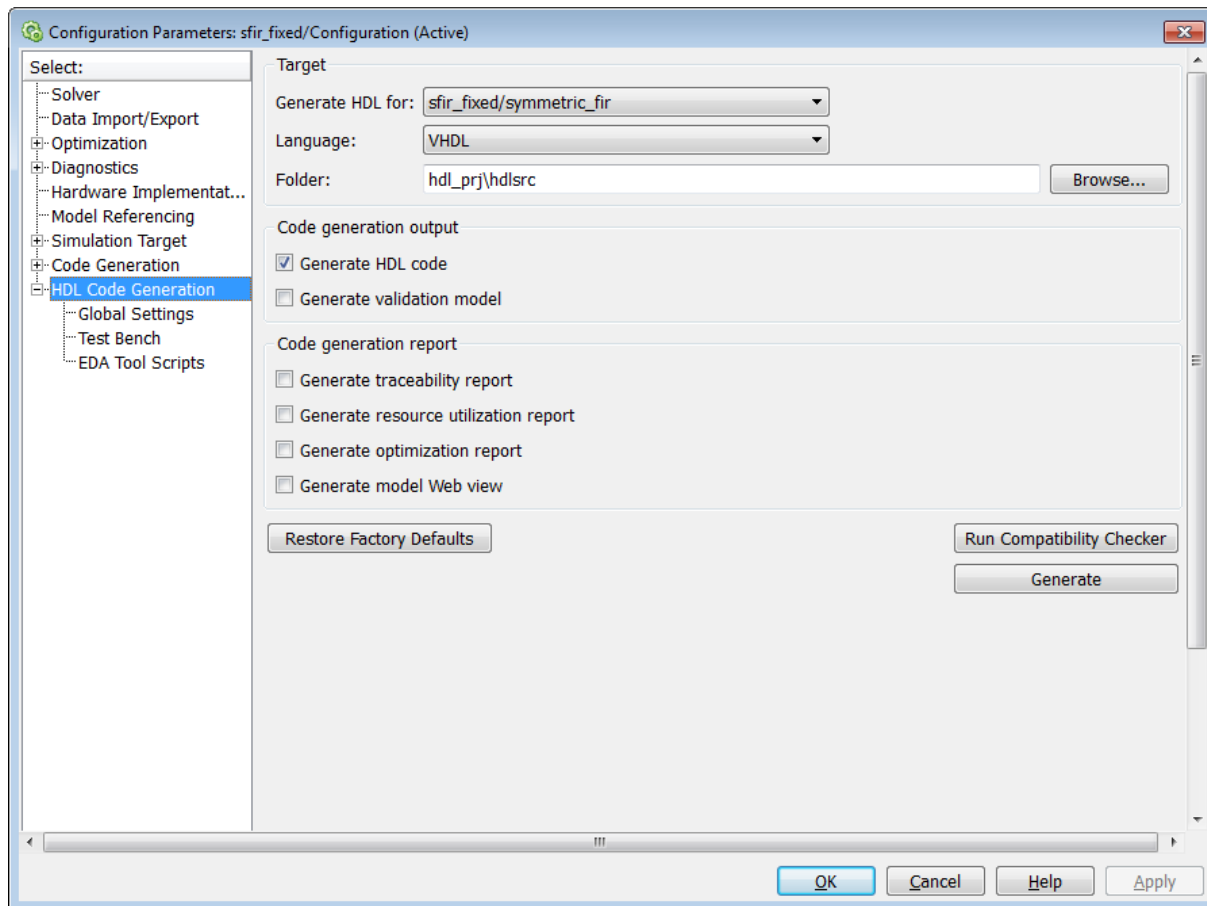
Most block implementations support a number of *implementation parameters* that let you control further details of code generation for the block. The HDL Properties dialog box lets you set implementation parameters for a block.

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” on page 10-2 for detailed information.

HDL Code Generation Pane: General



In this section...

“HDL Code Generation Top-Level Pane Overview” on page 9-10

“Generate HDL for” on page 9-12

“Language” on page 9-13

“Folder” on page 9-14

“Generate HDL code” on page 9-15

In this section...

“Generate validation model” on page 9-16

“Generate traceability report” on page 9-17

“Generate resource utilization report” on page 9-18

“Generate optimization report” on page 9-19

“Generate model Web view” on page 9-20

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; also (if the model has a control file linked to it) unlinks the control file from the model.

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'

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'`

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'

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'

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 [Creating and Using a Code Generation Report](#).



Off

Do not create an HTML code generation report.

Command-Line Information

Property: Traceability

Type: string

Value: 'on' | 'off'

Default: 'off'

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 [Creating and Using a Code Generation Report](#).

Off

Do not create an HTML resource utilization report.

Command-Line Information

Property: ResourceReport

Type: string

Value: 'on' | 'off'

Default: 'off'

See Also

ResourceReport

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 [Creating and Using a Code Generation Report](#).



Off

Do not create an HTML optimization report.

Command-Line Information

Property: OptimizationReport

Type: string

Value: 'on' | 'off'

Default: 'off'

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.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled and selected by **Create code generation report**.
- To enable traceability between the code and model, select **Code-to-model** and **Model-to-code**.

Command-Line Information

Parameter: GenerateWebview

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

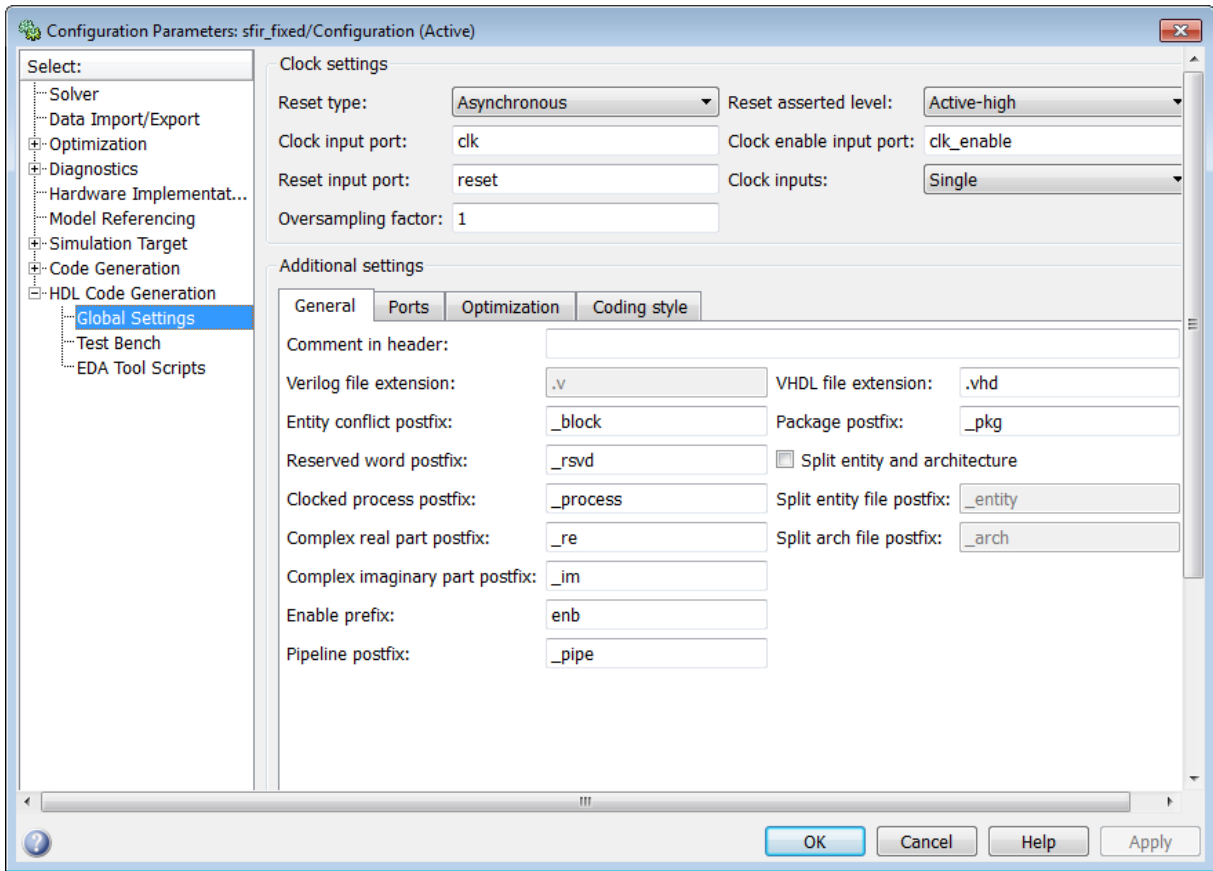
See Also

“Web View of Model in Code Generation Report”

See Also

“Web View of Model in Code Generation Report” on page 16-28

HDL Code Generation Pane: Global Settings



In this section...

“Global Settings Overview” on page 9-25

“Reset type” on page 9-26

“Reset asserted level” on page 9-27

“Clock input port” on page 9-28

“Clock enable input port” on page 9-29

In this section...

“Reset input port” on page 9-30

“Clock inputs” on page 9-31

“Oversampling factor” on page 9-32

“Comment in header” on page 9-33

“Verilog file extension” on page 9-34

“VHDL file extension” on page 9-35

“Entity conflict postfix” on page 9-36

“Package postfix” on page 9-37

“Reserved word postfix” on page 9-38

“Module name prefix” on page 9-38

“Split entity and architecture” on page 9-40

“Split entity file postfix” on page 9-42

“Split arch file postfix” on page 9-43

“Clocked process postfix” on page 9-44

“Enable prefix” on page 9-45

“Pipeline postfix” on page 9-46

“Complex real part postfix” on page 9-47

“Complex imaginary part postfix” on page 9-48

“Input data type” on page 9-49

“Output data type” on page 9-50

“Clock enable output port” on page 9-52

“Balance delays” on page 9-53

“Hierarchical distributed pipelining” on page 9-54

“Optimize timing controller” on page 9-55

“Minimize clock enables” on page 9-57

“RAM mapping threshold (bits)” on page 9-60

In this section...

“Max oversampling” on page 9-61

“Max computation latency” on page 9-62

“Represent constant values by aggregates” on page 9-63

“Use rising_edge for registers” on page 9-64

“Loop unrolling” on page 9-65

“Use Verilog `timescale directives” on page 9-66

“Inline VHDL configuration” on page 9-67

“Concatenate type safe zeros” on page 9-68

“Emit time/date stamp in header” on page 9-69

“HDL coding standard” on page 9-70

“Scalarize vector ports” on page 9-71

“Minimize intermediate signals” on page 9-72

“Include requirements in block comments” on page 9-73

“Inline MATLAB Function block code” on page 9-74

“Generate parameterized HDL code from masked subsystem” on page 9-75

“Initialize all RAM blocks” on page 9-76

“RAM Architecture” on page 9-76

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'

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'

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'`

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'`

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'

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'

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, the 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

See Also

Generating a Global Oversampling Clock
Oversampling

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

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 '`

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'

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 the 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'`

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`

The 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'`

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`, the coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

Command-Line Information

Property: `ReservedWordPostfix`

Type: `string`

Default: `'_rsvd'`

See Also

`ReservedWordPostfix`

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. The 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: ''

See Also

ModulePrefix

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'

See Also

SplitEntityArch

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'`

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'`

See Also

`SplitArchFilePostfix`

Clocked process postfix

Specify a string to append to HDL clock process names.

Settings

Default: `_process`

The 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'`

See Also

`ClockProcessPostfix`

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'

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 the coder appends to names of input or output pipeline registers.

Command-Line Information

Property: PipelinePostfix

Type: string

Default: `'_pipe'`

See Also

PipelinePostfix

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'`

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'`

See Also

`ComplexImagPostfix`

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'`

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`.

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'`

See Also

`ClockEnableOutputPort`

Balance delays

Enable delay balancing.

Settings

Default: On



On

If the 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'

See Also

Delay Balancing

Hierarchical distributed pipelining

Specify that retiming be applied across a subsystem hierarchy.

Settings

Default: Off



On

Enable retiming across a subsystem hierarchy. The 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'

See Also

- HierarchicalDistPipelining
- DistributedPipelining

Optimize timing controller

Optimize timing controller entity for speed and code size by implementing separate counters per rate.

Settings

Default: On



On

The 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'

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, the 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
 - dspigops/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
- hdlldemolib/HDL FFT
- built-in/DiscreteFir
- dspmlti4/CIC Decimation
- dspmlti4/CIC Interpolation
- dspmlti4/FIR Decimation
- dspmlti4/FIR Interpolation
- dspadpt3/LMS Filter
- dsparch4/Biquad Filter
- dsparch4/Digital Filter

Command-Line Information

Property: MinimizeClockEnables

Type: string

Value: 'on' | 'off'

Default: 'off'

See Also

`MinimizeClockEnables`

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. The 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

See Also

- RAMMappingThreshold
- UseRAM
- MapPersistentVarsToRAM

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

See Also

- MaxOversampling
- MaxComputationLatency

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

See Also

- MaxComputationLatency
- MaxOversampling

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

The 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'

See Also

UseAggregatesForConst

Use rising_edge for registers

Specify whether or not generated code uses the VHDL `rising_edge` function to check for rising edges when operating on registers.

Settings

Default: Off



On

Generated code uses the VHDL `rising_edge` function to check for rising edges when operating on registers.



Off

Generated code checks for clock events when operating on registers.

Dependencies

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

Command-Line Information

Property: `UseRisingEdge`

Type: `string`

Value: `'on' | 'off'`

Default: `'off'`

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'

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'

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, the 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'

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'

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'

See Also

DateComment

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 coder. When this option is enabled, the coder generates a standard compliance report.

Command-Line Information

Property: HDLCodingStandard

Type: string

Value: 'None' | 'Industry'

Default: 'None'

See Also

HDLCodingStandard

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'

See Also

ScalarizePorts

Minimize intermediate signals

Specify whether to optimize HDL code for debuggability or code coverage.

Settings

Default: Off



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*, the 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`.



Optimize for debuggability by preserving intermediate signals.

Command-Line Information

Property: MinimizeIntermediateSignals

Type: string

Value: 'on' | 'off'

Default: 'off'

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” on page 16-35.



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'

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'

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. The 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'

See Also

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'

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

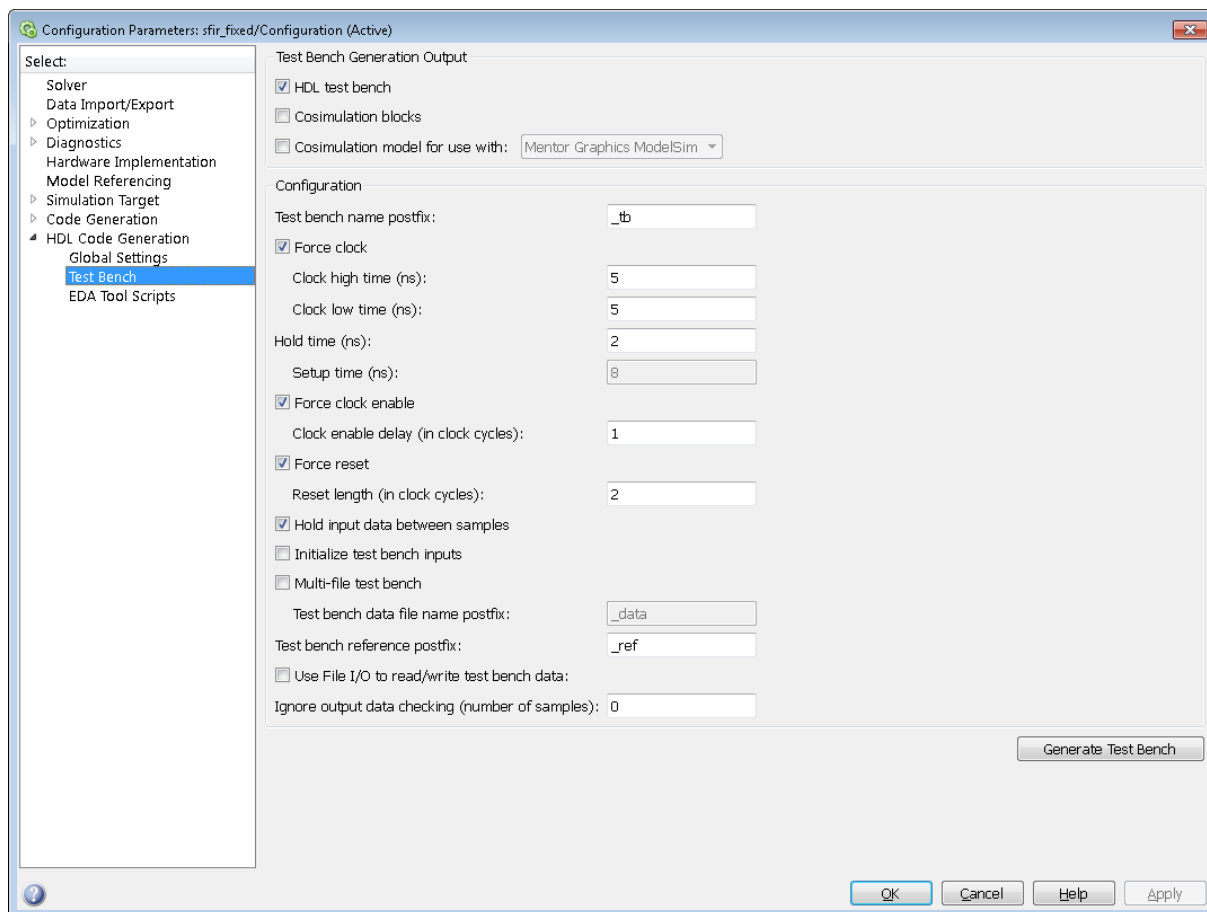
Value: 'WithClockEnable' | 'WithoutClockEnable'

Default: 'WithClockEnable'

See Also

RAMArchitecture

HDL Code Generation Pane: Test Bench



In this section...

“Test Bench Overview” on page 9-80

“HDL test bench” on page 9-81

“Cosimulation blocks” on page 9-82

“Cosimulation model for use with:” on page 9-84

In this section...

- “Test bench name postfix” on page 9-85
- “Force clock” on page 9-86
- “Clock high time (ns)” on page 9-87
- “Clock low time (ns)” on page 9-88
- “Hold time (ns)” on page 9-89
- “Setup time (ns)” on page 9-90
- “Force clock enable” on page 9-91
- “Clock enable delay (in clock cycles)” on page 9-92
- “Force reset” on page 9-94
- “Reset length (in clock cycles)” on page 9-95
- “Hold input data between samples” on page 9-97
- “Initialize test bench inputs” on page 9-98
- “Multi-file test bench” on page 9-99
- “Test bench reference postfix” on page 9-101
- “Test bench data file name postfix” on page 9-102
- “Use file I/O to read/write test bench data” on page 9-103
- “Ignore output data checking (number of samples)” on page 9-103

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.

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, the 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'

See Also

GenerateCoSimBlock

Cosimulation model for use with:

Generate model containing HDL Cosimulation block for cosimulation

Settings

Default: Off

On

Selecting this option enables the dropdown menu to the right of the check box. 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, the 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, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for Cadence Incisive.

Off

Do not generate HDL Cosimulation model.

Dependencies

This check box enables the other options in the **Configuration** section of the **Test Bench** pane.

Command-Line Information

Property: GenerateCosimModel

Type: string

Value: 'on' | 'off'

Default: 'off'

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`, the coder adds the default postfix `_tb` to form the name `my_test_tb`.

Command-Line Information

Property: `TestBenchPostFix`

Type: `string`

Default: `'_tb'`

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 high time** options.

Command-Line Information

Property: ForceClock
Type: string
Value: 'on' | 'off'
Default: 'on'

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: A positive integer

Default: 5

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: A positive integer

Default: 5

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: A positive integer

Default: 2

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'

See Also

ForceClockEnable

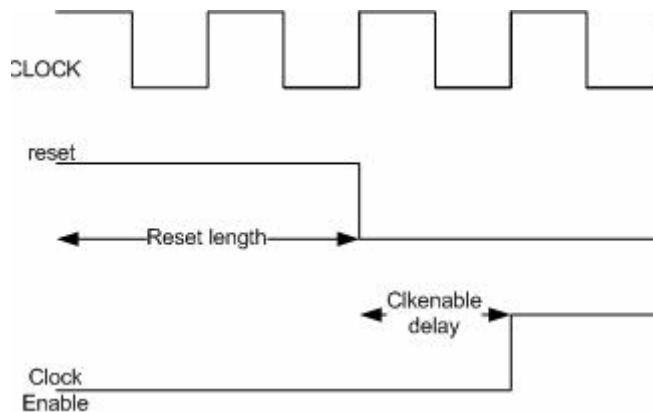
Clock enable delay (in clock cycles)

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

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'

See Also

ForceReset

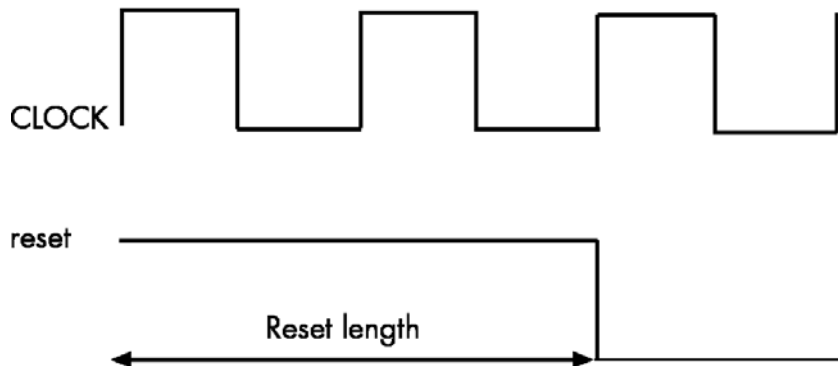
Reset length (in clock cycles)

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

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 is ≥ 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'

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'

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'

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'`

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'

The 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'

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.

See Also

UseFileIOInTestBench

Command-Line Information

Property: UseFileIOInTestBench

Type: string

Value: 'on' | 'off'

Default: 'off'

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 N of **Ignore output data checking (number of samples)** 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 specify the 'DistributedPipelining', 'on' parameter for the MATLAB Function block (see “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 20-42)
- When you specify the { 'ResetType', 'None' } parameter for the following block types:
 - 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

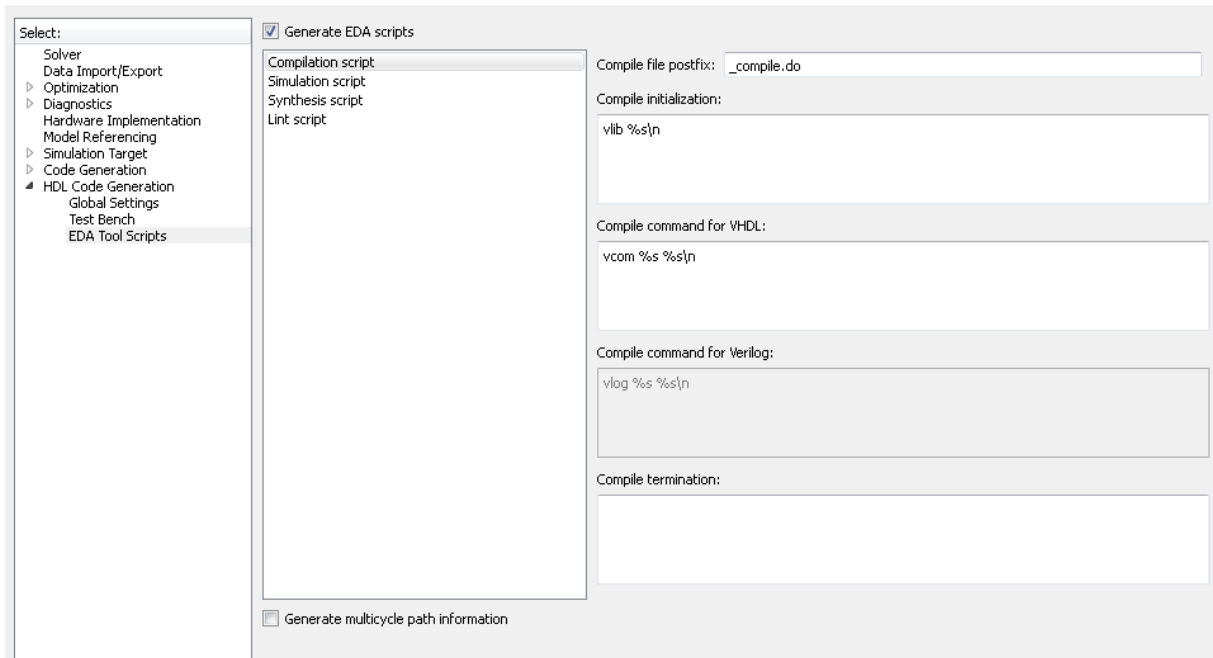
Type: integer

Default: 0

See Also

IgnoreDataChecking

HDL Code Generation Pane: EDA Tool Scripts



In this section...

- “EDA Tool Scripts Overview” on page 9-108
- “Generate EDA scripts” on page 9-109
- “Generate multicycle path information” on page 9-110
- “Compile file postfix” on page 9-111
- “Compile initialization” on page 9-112
- “Compile command for VHDL” on page 9-113
- “Compile command for Verilog” on page 9-114
- “Compile termination” on page 9-115
- “Simulation file postfix” on page 9-116
- “Simulation initialization” on page 9-117

In this section...

“Simulation command” on page 9-118

“Simulation waveform viewing command” on page 9-119

“Simulation termination” on page 9-120

“Choose synthesis tool” on page 9-121

“Synthesis file postfix” on page 9-123

“Synthesis initialization” on page 9-124

“Synthesis command” on page 9-125

“Synthesis termination” on page 9-126

“Choose HDL lint tool” on page 9-126

“Lint initialization” on page 9-127

“Lint command” on page 9-128

“Lint termination” on page 9-128

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'

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- 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'

See Also

- Generating a Multicycle Path Information File
- 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`, the coder adds the postfix `_compile.do` to form the name `my_design_compile.do`.

Command-Line Information

Property: `HDLCompileFilePostfix`

Type: `string`

Default: `'_compile.do'`

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- `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 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'`

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- `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 or once per signal. On each call, a different file or signal name is passed in.

The two 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'`

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- `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 or once per signal. On each call, a different file or signal name is passed in.

The two 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'`

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- `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: `''`

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- `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`, the coder adds the postfix `_sim.do` to form the name `my_design_sim.do`.

Command-Line Information

Property: `HDLSimFilePostfix`

Type: `string`

Default: `'_sim.do'`

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- `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']

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- HDLSimInit

Simulation command

Specify a format string passed to `fprintf` to write the simulation command.

Settings

Default: `vsim -novopt work.%s\n`

The implicit argument is the top-level module or entity name.

Command-Line Information

Property: HDLSimCmd

Type: string

Default: `'vsim -novopt work.%s\n'`

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- HDLSimCmd

Simulation waveform viewing command

Specify the waveform viewing command written to simulation script.

Settings

Default: `add wave sim:%s\n`

The implicit argument 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'`

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- 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'`

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- `HDLSimTerm`

Choose synthesis tool

Enable or disable generation of synthesis scripts, and select the synthesis tool for which the coder generates scripts.

Settings

Default: None

None

When you select None, the 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.

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' | 'Custom'

Default: 'None'

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  
_custom.tcl
```

For example, if the DUT name is `my_design` and the choice of synthesis tool is Synopsys Synplify Pro, the coder adds the postfix `_synplify.tcl` to form the name `my_design_synplify.tcl`.

Command-Line Information

Property: HDLSynthFilePostfix

Type: string

Default: none

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- 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 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 is the top-level module or entity name. The content of the string is specific to the selected synthesis tool.

Command-Line Information

Property: HDLSynthInit

Type: string

Default: none

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- 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 default string is a format string passed to `fprintf` to write the `Cmd` section of the synthesis script. The argument is the filename of the entity or module. The content of the string is specific to the selected synthesis tool.

Command-Line Information

Property: HDLSynthCmd

Type: string

Default: none

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- 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 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. The content of the string is specific to the selected synthesis tool.

Command-Line Information

Property: HDLSynthTerm

Type: string

Default: none

See Also

- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9
- HDLSynthTerm

Choose HDL lint tool

Enable or disable generation of an HDL lint script, and select the HDL lint tool for which the 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.

SpyGlass

Generate a lint script for Atrenta SpyGlass.

Leda

Generate a lint script for Synopsys Leda.

Custom

Generate a custom synthesis script.

Command-Line Information

Property: HDLLintTool

Type: string

Value: 'None' | 'SpyGlass' | 'Leda' | 'Custom'

Default: 'None'

See Also

- “Generate an HDL Lint Tool Script” on page 17-11
- HDLLintTool

Lint initialization

Enter an initialization string for your HDL lint script.

Command-Line Information

Property: HDLLintInit

Type: string

Default: none

See Also

- “Generate an HDL Lint Tool Script” on page 17-11
- HDLLintInit

Lint command

Enter the command for your HDL lint script.

Command-Line Information

Property: HDLLintCmd

Type: string

Default: none

See Also

- “Generate an HDL Lint Tool Script” on page 17-11
- HDLLintCmd

Lint termination

Enter a termination string for your HDL lint script.

Command-Line Information

Property: HDLLintTerm

Type: string

Default: none

See Also

- “Generate an HDL Lint Tool Script” on page 17-11
- HDLLintTerm

Specifying Block Implementations and Parameters for HDL Code Generation

- “Set and View HDL Block Parameters” on page 10-2
- “Set HDL Block Parameters for Multiple Blocks” on page 10-5
- “View HDL Model Parameters” on page 10-7

Set and View HDL Block Parameters

In this section...
“Set HDL Block Parameters from the GUI” on page 10-2
“Set HDL Block Parameters from the Command Line” on page 10-2
“View All HDL Block Parameters” on page 10-3
“View Non-Default HDL Block Parameters” on page 10-4

For a list of HDL block properties, see “Block Implementation Parameters” on page 11-50.

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 `hdldispblkparams`.

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'
```


Guide to Supported Blocks and Block Implementations

- “Generate a Supported Blocks Report” on page 11-2
- “Blocks Supported for HDL Code Generation” on page 11-3
- “Blocks with Multiple Implementations” on page 11-16
- “Block-Specific Usage, Requirements, and Restrictions” on page 11-29
- “Block Implementation Parameters” on page 11-50
- “Blocks That Support Complex Data” on page 11-103
- “Blocks That Support Buses” on page 11-109
- “Lookup Table Block Support” on page 11-114

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:

```
hdllib('html');
```

After `hdllib` creates the `hdlsupported` 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” on page 16-41.

Blocks Supported for HDL Code Generation

You can automatically generate a library or report of supported blocks. To learn more, see `hdl1lib`.

See “Set and View HDL Block Parameters” on page 10-2 to learn how to set block implementations and parameters in the GUI or the command line.

The following tables summarize blocks that the coder supports for HDL code generation.

Simulink Blocks

Block	Additional Guidelines and Restrictions
1-D Lookup Table	See “1-D Lookup Table” on page 11-117.
Abs	
Add	
Assertion	
Assignment	
Bias	
Bit Clear	
Bit Set	
Bitwise Operator	
Bus Creator	
Bus Selector	
Check Discrete Gradient	
Check Dynamic Gap	
Check Dynamic Lower Bound	
Check Dynamic Range	
Check Dynamic Upper Bound	

Block	Additional Guidelines and Restrictions
Check Input Resolution	
Check Static Gap	
Check Static Lower Bound	
Check Static Range	
Check Static Upper Bound	
Compare To Constant	
Compare To Zero	
Complex to Real-Imag	
Constant	See “Constant” on page 11-17.
Counter Free-Running	
Counter Limited	
Data Type Conversion	See “Data Type Conversion Block Requirements and Restrictions” on page 11-35.
Data Type Duplicate	
Data Type Propagation	
Decrement Real World	
Decrement Stored Integer	
Delay	
Demux	
Direct Lookup Table (n-D)	See “Lookup Table Block Support” on page 11-114.

Block	Additional Guidelines and Restrictions
Discrete FIR Filter	See “CoeffMultipliers” on page 11-54, “Distributed Arithmetic Implementation Parameters for Digital Filter Blocks” on page 11-57 , “Pipelining Implementation Parameters for Filter Blocks” on page 11-81 , and “Speed vs. Area Optimizations for FIR Filter Implementations” on page 11-95.
Discrete Transfer Fcn	
Discrete-Time Integrator	See “Discrete-Time Integrator Requirements and Restrictions” on page 11-37.
Display	
Divide	<p>To perform a divide operation, connect a Product block to a Divide block in reciprocal mode.</p> <p>To select reciprocal mode, in the Divide block dialog box, set the Number of inputs to /.</p> <p>See “Divide (reciprocal)” on page 11-18.</p>
DocBlock	
Dot Product	
Enable	See “Generate Code for Enabled and Triggered Subsystems” on page 18-23.
Extract Bits	
Floating Scope	

Block	Additional Guidelines and Restrictions
From	Buses are not supported for HDL code generation. See “Blocks That Support Buses” on page 11-109.
Gain	See “Gain” on page 11-19.
Goto	Buses are not supported for HDL code generation. See “Blocks That Support Buses” on page 11-109.
Ground	
Increment Real World	
Increment Stored Integer	
Index Vector	
Inport	
Logical Operator	
MATLAB Function	
Magnitude-Angle to Complex	
Math Function	Supported functions: <ul style="list-style-type: none"> • reciprocal: see “Math Function (reciprocal)” on page 11-21. • conj: see “Math Function (conj)” on page 11-20. • hermitian: see “Math Function (hermitian)” on page 11-20. • transpose: see “Math Function (transpose)” on page 11-22.
Matrix Concatenate	
Memory	
MinMax	See “MinMax” on page 11-22.
Model	

Block	Additional Guidelines and Restrictions
Model Info	
Multiport Switch	
Mux	Buses are not supported for HDL code generation. See “Blocks That Support Buses” on page 11-109.
Outport	
Prelookup	See “Lookup Table Block Support” on page 11-114.
Product	See “Product” on page 11-23.
Product of Elements	
Rate Transition	
Real-Imag to Complex	
Reciprocal Sqrt	See “Reciprocal Sqrt” on page 11-24.
Relational Operator	
Relay	
Reshape	
Saturation	
Saturation Dynamic	
Scope	
Selector	
Shift Arithmetic	
Sign	
Signal Conversion	
Signal Specification	
Sqrt	See “Sqrt” on page 11-24.

Block	Additional Guidelines and Restrictions
Stop Simulation	
Subsystem	See “Subsystem” on page 11-26.
Subtract	
Sum	See “Sum” on page 11-26.
Sum of Elements	
Switch	
Tapped Delay	
Terminator	
To File	
To Workspace	
Trigger	See “Generate Code for Enabled and Triggered Subsystems” on page 18-23.
Trigonometric Function	<p>Supported functions (CORDIC approximation method only):</p> <ul style="list-style-type: none"> • sin • cos • cos + jsin • sincos <p>See “Trigonometric Function Block Requirements and Restrictions” on page 11-43.</p>
Unary Minus	
Unit Delay	
Unit Delay Enabled	
Unit Delay Enabled Resettable	

Block	Additional Guidelines and Restrictions
Unit Delay Resettable	
Vector Concatenate	
XY Graph	
Zero-Order Hold	
n-D Lookup Table	See “Lookup Table Block Support” on page 11-114.

Communications System Toolbox Blocks

Block	Additional Restrictions and Guidelines
BPSK Demodulator Baseband	
BPSK Modulator Baseband	
Convolutional Deinterleaver	See “Convolutional Interleaver and Deinterleaver Block Requirements and Restrictions” on page 11-34.
Convolutional Encoder	See “Convolutional Encoder Block Requirements and Restrictions” on page 11-33.
Convolutional Interleaver	See “Convolutional Interleaver and Deinterleaver Block Requirements and Restrictions” on page 11-34.
General CRC Generator HDL Optimized	
General CRC Syndrome Detector HDL Optimized	
General Multiplexed Interleaver	See “General Multiplexed Interleaver and Deinterleaver Block Requirements and Restrictions” on page 11-38.

Block	Additional Restrictions and Guidelines
General Multiplexed Deinterleaver	See “General Multiplexed Interleaver and Deinterleaver Block Requirements and Restrictions” on page 11-38.
Integer-Input RS Encoder HDL Optimized	
Integer-Output RS Decoder HDL Optimized	
M-PSK Demodulator Baseband	
M-PSK Modulator Baseband	
PN Sequence Generator	See “PN Sequence Generator Block Requirements and Restrictions” on page 11-41.
QPSK Demodulator Baseband	
QPSK Modulator Baseband	
Rectangular QAM Demodulator Baseband	See “Rectangular QAM Demodulator Baseband Block Requirements and Restrictions” on page 11-42.
Rectangular QAM Modulator Baseband	See “Rectangular QAM Modulator Baseband Block Requirements and Restrictions” on page 11-42.
Viterbi Decoder	See “Viterbi Decoder Block Requirements and Restrictions” on page 11-44 and “Pipelining the Traceback Unit” on page 11-46.

DSP System Toolbox Blocks

Block	Additional Restrictions and Guidelines
Biquad Filter	See “Biquad Filter Block Requirements and Restrictions” on page 11-30, “Pipelining Implementation Parameters for Filter Blocks” on page 11-81, and “CoeffMultipliers” on page 11-54.
CIC Decimation	See “Multirate CIC Decimation and Multirate FIR Decimation Blocks Requirements and Restrictions” on page 11-39, and “Pipelining Implementation Parameters for Filter Blocks” on page 11-81.
CIC Interpolation	See “Multirate CIC Decimation and Multirate FIR Decimation Blocks Requirements and Restrictions” on page 11-39, “Multirate CIC Interpolation and Multirate FIR Interpolation Blocks Requirements and Restrictions” on page 11-40, and “Pipelining Implementation Parameters for Filter Blocks” on page 11-81
Convert 1-D to 2-D	
DSP Constant (Obsolete)	
Data Type Conversion	See “Data Type Conversion Block Requirements and Restrictions” on page 11-35.
Delay	

Block	Additional Restrictions and Guidelines
Digital Filter	See “Digital Filter Block Requirements and Restrictions” on page 11-35 and “CoeffMultipliers” on page 11-54, “Distributed Arithmetic Implementation Parameters for Digital Filter Blocks” on page 11-57, “Pipelining Implementation Parameters for Filter Blocks” on page 11-81, and “Speed vs. Area Optimizations for FIR Filter Implementations” on page 11-95.
Downsample	
FIR Decimation	See “Multirate CIC Decimation and Multirate FIR Decimation Blocks Requirements and Restrictions” on page 11-39, “CoeffMultipliers” on page 11-54, “Distributed Arithmetic Implementation Parameters for Digital Filter Blocks” on page 11-57, “Speed vs. Area Optimizations for FIR Filter Implementations” on page 11-95, and “Pipelining Implementation Parameters for Filter Blocks” on page 11-81.
FIR Interpolation	See “Multirate CIC Interpolation and Multirate FIR Interpolation Blocks Requirements and Restrictions” on page 11-40, “Pipelining Implementation Parameters for Filter Blocks” on page 11-81, “CoeffMultipliers” on page 11-54, “Distributed Arithmetic Implementation Parameters for Digital Filter Blocks” on page 11-57, and “Speed vs. Area Optimizations

Block	Additional Restrictions and Guidelines
	for FIR Filter Implementations” on page 11-95.
Frame Conversion	
LMS Filter	See “LMS Filter Usage and Restrictions” on page 11-38.
Matrix Viewer	
Maximum	See “Maximum” on page 11-22.
Minimum	See “Minimum” on page 11-22.
Multiport Selector	
NCO	<p>See “NCO Block Requirements and Restrictions” on page 11-41.</p> <hr/> <p>Note HDL support for the NCO block will be removed in a future release. Use the NCO HDL Optimized block instead.</p> <hr/>
NCO HDL Optimized	
Repeat	
Signal To Workspace	
Sine Wave	See “Sine Wave Block Requirements and Restrictions” on page 11-43.
Spectrum Scope	
Time Scope	
Upsample	
Variable Selector	
Vector Scope	
Waterfall	

HDL Verifier Blocks

Block	Additional Restrictions and Guidelines
HDL Cosimulation	See “Code Generation for HDL Cosimulation Blocks” on page 18-37.
To VCD File	

Stateflow Blocks

Block	Additional Restrictions and Guidelines
Chart	See “Stateflow® Chart Usage”.
Chart(MATLAB)	See “Stateflow Chart Usage”.
State Transition Table	See “Stateflow Chart Usage”.
Truth Table	See “Stateflow Chart Usage”.

HDL Demo Library Blocks

Block	Description
Bit Concat	See “Bit Concat” on page 13-52.
Bit Reduce	See “Bit Reduce” on page 13-55.
Bit Rotate	See “Bit Rotate” on page 13-57.
Bit Shift	See “Bit Shift” on page 13-59.
Bit Slice	See “Bit Slice” on page 13-61.
Dual Port RAM	See “Dual Port RAM Block” on page 13-5.
HDL Counter	See “HDL Counter” on page 13-15.
HDL FFT	See “HDL FFT” on page 13-27.
HDL FIFO	See “HDL FIFO” on page 13-36.

Block	Description
HDL Streaming FFT	See “HDL Streaming FFT” on page 13-40.
Simple Dual Port RAM	See “Simple Dual Port RAM Block” on page 13-7.
Single Port RAM	See “Single Port RAM Block” on page 13-8.

Blocks with Multiple Implementations

In this section...
“Block Implementations” on page 11-16
“Pass-through and No HDL Implementations” on page 11-27
“Cascade Implementation Best Practices” on page 11-27

The following tables describe implementation details and usage restrictions for supported blocks with more than one HDL implementation.

See “Set and View HDL Block Parameters” on page 10-2 to learn how to set block implementations and parameters in the GUI or using the command line.

Block Implementations

- “Constant” on page 11-17
- “Divide” on page 11-17
- “Divide (reciprocal)” on page 11-18
- “Gain” on page 11-19
- “Math Function (conj)” on page 11-20
- “Math Function (hermitian)” on page 11-20
- “Math Function (reciprocal)” on page 11-21
- “Math Function (transpose)” on page 11-22
- “Maximum” on page 11-22
- “Minimum” on page 11-22
- “MinMax” on page 11-22
- “Model” on page 11-23
- “Product” on page 11-23
- “Product (divide)” on page 11-24
- “Reciprocal Sqrt” on page 11-24

- “Sqrt” on page 11-24
- “Subsystem” on page 11-26
- “Sum” on page 11-26

Constant

Implementations	Parameters	Description
default Constant	None	This implementation emits the value of the Constant block.
Logic Value	None	By default, this implementation emits the character 'Z' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'ZZZZ'.
	{'Value', 'Z'}	Use this parameter value if the signal is in a high-impedance state. This implementation emits the character 'Z' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'ZZZZ'.
	{'Value', 'X'}	Use this parameter value if the signal is in an unknown state. This implementation emits the character 'X' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'XXXX'.

Note

The `Logic Value` implementation does not support the `double` data type. If you specify this implementation for a Constant of type `double`, a code-generation error occurs.

Divide

To perform an HDL-optimized divide operation, connect a Product block to a Divide block in reciprocal mode. For information about the Divide block in reciprocal mode, see “Divide (reciprocal)” on page 11-18.

In default mode, the Divide block supports only integer data types for HDL code generation.

Implementations	Parameters	Description
default Linear	None	Generate a divide (/) operator in the HDL code.

Divide (reciprocal)

The Divide block is in reciprocal mode when **Number of Inputs** is set to /. In reciprocal mode, the Divide block has the HDL block implementations described in the following table.

Implementations	Parameters	Description
default Linear	None	When you compute a reciprocal, compute $1/N$ using the HDL divide (/) operator to implement the division.
RecipNewton	{'Iterations', N}	Use the iterative Newton method. Select this option to optimize area. The argument N specifies the number of iterations. The default value for N is 3. The recommended value for N is between 2 and 10. The coder generates a message if N is outside the recommended range.
RecipNewtonSingleRate	{'Iterations', N}	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation. The argument N specifies the number of iterations. The default value for N is 3. The recommended value for N is between 2 and 10. The coder generates a message if N is outside the recommended range.

When you use the Divide block in reciprocal mode, the following restrictions apply:

- The input must be scalar and must have integer or fixed-point (signed or unsigned) data type.
- The output must be scalar and have integer or fixed-point (signed or unsigned) data type.
- Only the Zero rounding mode is supported.
- The **Saturate on integer overflow** option on the block must be selected.

Gain

Implementations	Parameters	Description
default	'ConstMultiplierOptimization', 'none' (Default)	By default, the coder does not perform CSD or FCSD optimizations. Code generated for the Gain block retains multiplier operations.
	'ConstMultiplierOptimization', 'CSD'	When you specify this option, the generated code decreases the area used by the model while maintaining or increasing clock speed, using canonic 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.
	'ConstMultiplierOptimization', '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

Implementations	Parameters	Description
	<p data-bbox="387 430 828 487">'ConstMultiplierOptimization', 'auto'</p>	<p data-bbox="850 322 1307 413">reduction. FCSD lets you achieve a greater area reduction than CSD, at the cost of decreasing clock speed.</p> <p data-bbox="850 430 1329 812">When you specify this option, the 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).</p>

Math Function (conj)

Implementations	Description
ComplexConjugate	Compute complex conjugate. See Math Function in the Simulink documentation.

Math Function (hermitian)

Implementations	Description
Hermitian	Compute hermitian. See Math Function in the Simulink documentation.

Math Function (reciprocal)

Implementations	Parameters	Description
Math (default) Reciprocal	None	Compute reciprocal as $1/N$, using the HDL divide (/) operator to implement the division.
RecipNewton	{ 'Iterations', N }	Use the iterative Newton method. Select this option to optimize area. The argument N specifies the number of iterations. The default value for N is 3. The recommended value for N is between 2 and 10. The coder generates a message if N is outside the recommended range.
RecipNewtonSingleRate	{ 'Iterations', N }	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation. The argument N specifies the number of iterations. The default value for N is 3. The recommended value for N is between 2 and 10. The coder generates a message if N is outside the recommended range.

When you use a reciprocal implementation, consider the following:

- Input must be scalar and must have integer or fixed-point (signed or unsigned) data type.
- The output must be scalar and have integer or fixed-point (signed or unsigned) data type.
- Only the Zero rounding mode is supported.
- The **Saturate on integer overflow** option on the block must be selected.

Math Function (transpose)

Implementations	Description
Transpose	Compute array transpose. See Math Function in the Simulink documentation.

Maximum

Implementations	Description
default Tree	The Tree implementation is large and slow but has minimal latency.
Cascade	This implementation is optimized for latency * area, with medium speed. See “Cascade Implementation Best Practices” on page 11-27.

Minimum

Implementations	Description
default Tree	The Tree implementation is large and slow but has minimal latency.
Cascade	This implementation is optimized for latency * area, with medium speed. See “Cascade Implementation Best Practices” on page 11-27.

MinMax

Implementations	Description
default Tree	The Tree implementation is large and slow but has minimal latency.
Cascade	This implementation is optimized for latency * area, with medium speed. See “Cascade Implementation Best Practices” on page 11-27.

Model

Implementations	Description
BlackBox (default)	Use the BlackBox implementation to generate an HDL interface to an external component, such as legacy HDL code. For more information, see “Generate Black Box Interface for Referenced Model” on page 18-21.
ModelReference	Use the ModelReference implementation when you want to generate code from a referenced model. For more information, see “How To Generate Code for a Referenced Model” on page 18-18.

Product

Implementations	Description
Linear (default)	Generates a chain of N operations (multipliers) for N inputs.
Tree	<p>This implementation has minimal latency but is large and slow. It generates a tree-shaped structure of multipliers.</p> <p>Note: Product blocks that have a vector input with two or more elements support Tree and Cascade.</p>
Cascade	<p>This implementation optimizes latency * area and is faster than the tree implementation. It computes partial products and cascades multipliers.</p> <p>Note: Product blocks that have a vector input with two or more elements support Tree and Cascade.</p> <p>See “Cascade Implementation Best Practices” on page 11-27.</p>

Product (divide)

For block implementations of the Product block in divide mode, see “Divide” on page 11-17.

Note The Product block is in divide mode when the **Number of Inputs** is set to */.

Reciprocal Sqrt

Implementations	Description
SqrtFunction (default) RecipSqrtNewton	Use the iterative Newton method. Select this option to optimize area.
RecipSqrtNewtonSingleRate	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.

When you generate HDL code from the Reciprocal Sqrt block, the following restrictions apply:

- In the Block Parameters dialog box, in the **Algorithm** tab, for **Method**, select **Newton-Raphson**.
- Input must be an unsigned scalar value.
- Output is a fixed-point scalar value.

Sqrt

Implementation	Parameter	Description
SqrtFunction (default) SqrtBitset	{'UseMultiplier', 'on'}	Use a multiply/add algorithm (Simulink default algorithm).
	{'UseMultiplier', 'off'}	Use a bitset shift/addition algorithm.

Implementation	Parameter	Description
SqrtNewton	{ 'Iterations', N }	<p>Use the iterative Newton method. Select this option to optimize area.</p> <p>The argument N specifies the number of iterations.</p> <p>The default value for N is 3.</p> <p>The recommended value for N is between 2 and 10. The coder generates a message if N is outside the recommended range.</p>
SqrtNewtonSingleRate	{ 'Iterations', N }	<p>Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.</p> <p>The argument N specifies the number of iterations.</p> <p>The default value for N is 3.</p> <p>The recommended value for N is between 2 and 10. The coder generates a message if N is outside the recommended range.</p>
SqrtTargetLibrary	None	Use the Altera or Xilinx target library.

When you generate HDL code from the Sqrt block, the following restrictions apply:

- Input must be an unsigned scalar value.
- Output is a fixed-point scalar value.

Subsystem

Implementation	Description
BlackBox	<p>This implementation generates a black-box interface for subsystems. That is, the generated HDL code includes only the input/output port definitions for the subsystem. In this way, you can use a subsystem in your model to generate an interface to existing manually written HDL code.</p> <p>The black-box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of clock signals.</p>
default No HDL	<p>This implementation completely removes the subsystem from the generated code. Thus, you can use a subsystem in simulation but treat it as a “no-op” in the HDL code.</p>

For more information on code generation for subsystems, see “External Component Interfaces”.

Sum

Implementations	Description
default Linear	<p>Generates a chain of N operations (adders) for N inputs.</p> <p>Note: The coder supports Tree and Cascade for Sum blocks that have a single vector input with multiple elements.</p>
Tree	<p>This implementation has minimal latency but is large and slow. Generates a tree-shaped structure of adders.</p>
Cascade	<p>This implementation optimizes latency * area and is faster than the tree implementation. It computes partial sums and cascades adders.</p>

Implementations	Description
	See “Cascade Implementation Best Practices” on page 11-27.

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. The 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. 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 Implementation Best Practices

The 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.

Block-Specific Usage, Requirements, and Restrictions

In this section...
“Block Usage, Requirements, and Restrictions” on page 11-29
“Restrictions on Use of Blocks in the Test Bench” on page 11-49

Block Usage, Requirements, and Restrictions

The following blocks have specific requirements and restrictions for HDL code generation.

- “Biquad Filter Block Requirements and Restrictions” on page 11-30
- “Convolutional Encoder Block Requirements and Restrictions” on page 11-33
- “Convolutional Interleaver and Deinterleaver Block Requirements and Restrictions” on page 11-34
- “Data Type Conversion Block Requirements and Restrictions” on page 11-35
- “Digital Filter Block Requirements and Restrictions” on page 11-35
- “Discrete FIR Filter Requirements and Restrictions” on page 11-35
- “Discrete Transfer Fcn Requirements and Restrictions” on page 11-36
- “Discrete-Time Integrator Requirements and Restrictions” on page 11-37
- “FIR Decimation Requirements and Restrictions” on page 11-37
- “FIR Interpolation Requirements and Restrictions” on page 11-37
- “General Multiplexed Interleaver and Deinterleaver Block Requirements and Restrictions” on page 11-38
- “LMS Filter Usage and Restrictions” on page 11-38
- “Magnitude-Angle to Complex Block Requirements and Restrictions” on page 11-39
- “Multirate CIC Decimation and Multirate FIR Decimation Blocks Requirements and Restrictions” on page 11-39

- “Multirate CIC Interpolation and Multirate FIR Interpolation Blocks Requirements and Restrictions” on page 11-40
- “NCO Block Requirements and Restrictions” on page 11-41
- “PN Sequence Generator Block Requirements and Restrictions” on page 11-41
- “Reciprocal Sqrt Block Requirements and Restrictions” on page 11-42
- “Rectangular QAM Demodulator Baseband Block Requirements and Restrictions” on page 11-42
- “Rectangular QAM Modulator Baseband Block Requirements and Restrictions” on page 11-42
- “Sine Wave Block Requirements and Restrictions” on page 11-43
- “Trigonometric Function Block Requirements and Restrictions” on page 11-43
- “Viterbi Decoder Block Requirements and Restrictions” on page 11-44

Biquad Filter Block Requirements and Restrictions

- “General Guidelines” on page 11-30
- “Programmable Filter Support” on page 11-30
- “Serial Architecture Support” on page 11-31

General Guidelines.

- Data vector and frame inputs are not supported for HDL code generation.
- **Initial conditions** must be set to zero. HDL code generation is not supported for nonzero initial states.
- **Optimize unity scale values** must be selected.

Programmable Filter Support. The coder supports programmable filters for Biquad Filters. Fully parallel and applicable serial architectures are supported.

- 1 Select **Input port(s)** as coefficient source on the filter block mask.

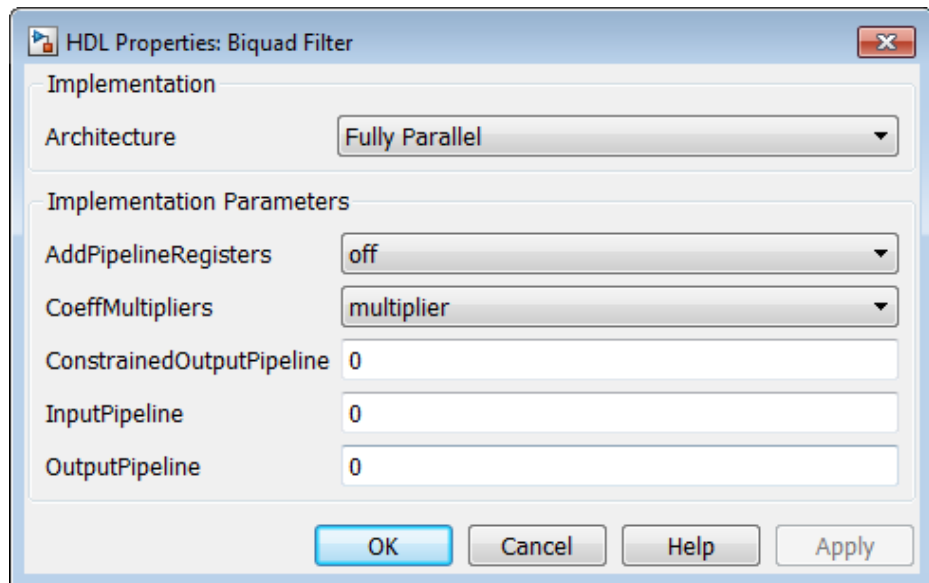
- 2 Connect the coefficient port with a vector signal.
- 3 Specify the implementation architecture and parameters from the HDL Coder property interface.
- 4 Generate HDL code.

Programmable filters are not supported for:

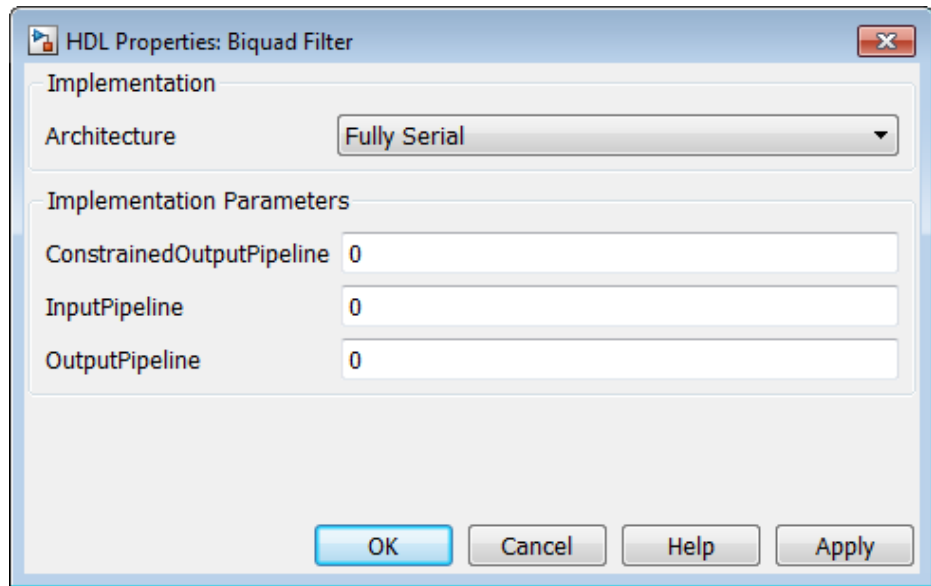
- CoeffMultipliers as `csd` or `factored-csd`

Serial Architecture Support. Biquad Filter block supports fully parallel, fully serial, and partly serial architectures.

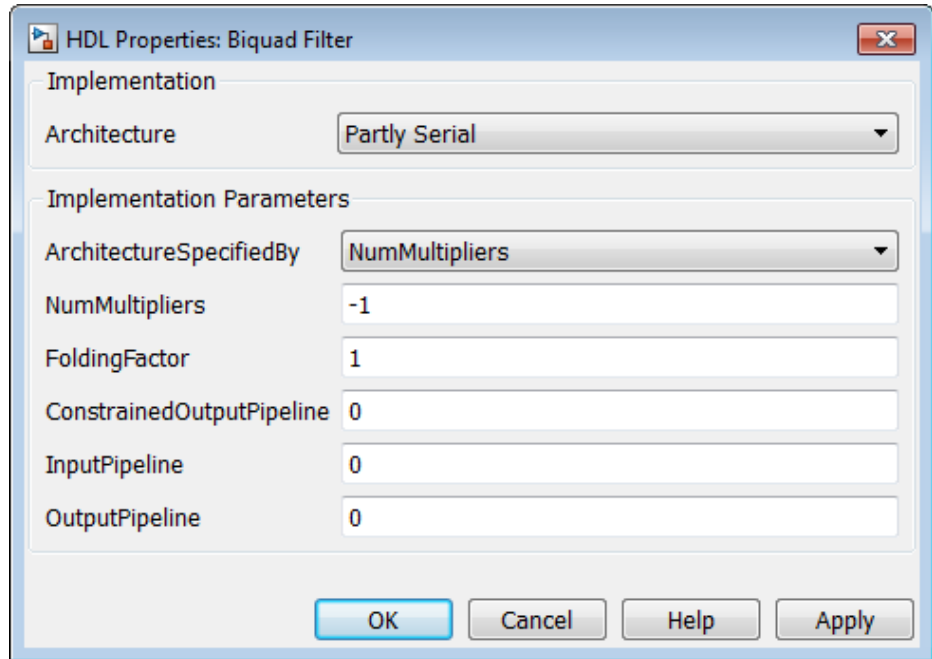
- Fully Parallel (default): `AddPipelineRegisters`, `CoeffMultipliers`, `ConstrainedOutputPipeline`, `InputPipeline`, `OutputPipeline`



- Fully Serial: `ConstrainedOutputPipeline`, `InputPipeline`, `OutputPipeline`



- Partly Serial: ArchitectureSpecifiedBy, NumMultipliers, FoldingFactor, ConstrainedOutputPipeline, InputPipeline, OutputPipeline



Convolutional Encoder Block Requirements and Restrictions

Input data requirements:

- Must be sample-based,
- Must have `boolean` or `ufix1` data type.

The coder supports only the following coding rates:

- $\frac{1}{2}$ to $\frac{1}{7}$
- $\frac{2}{3}$

The coder supports only constraint lengths for 3 to 9.

Trellis structure must be specified by the `poly2trellis` function.

The coder supports the following **Operation mode** settings:

- Continuous
- Reset on nonzero input via port

If you select this mode, you must select the **Delay reset action to next time step** option. When you select this option, the Convolutional Encoder block finishes its current computation before executing a reset.

Convolutional Interleaver and Deinterleaver Block Requirements and Restrictions

Shift Register Based Implementations. The default implementations for the Convolutional Interleaver and Deinterleaver blocks are shift register based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` to 'none'.

Note that when you set `ResetType` to 'none', reset is not applied to the shift registers. 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 spurious test bench errors, determine the number of samples required to fully load the shift registers. Then, set the **Ignore output data checking (number of samples)** option accordingly. (You can use the `IgnoreDataChecking` property for this purpose, if you are using the command-line interface.)

RAM Based Implementations. When you select the RAM implementation for a Convolutional Interleaver or Deinterleaver block, the coder uses RAM resources instead of shift registers. The implementation has the following limitations:

When you select the RAM implementation for a Convolutional Interleaver or Deinterleaver block, the coder uses RAM resources instead of shift registers.

- Double or single data types are not supported for either input or output signals.
- **Initial conditions** for the block must be set to zero.
- At least two rows of interleaving are required .

Data Type Conversion Block Requirements and Restrictions

If you configure a Data Type Conversion block for double to fixed-point or fixed-point to double conversion, a warning displays during code generation.

Digital Filter Block Requirements and Restrictions

- If you select the Digital Filter block **Discrete-time filter object** option, you must have the DSP System Toolbox software to generate code for the block.
- **Initial conditions** must be set to zero. HDL code generation is not supported for nonzero initial states.
- The coder does not support the Digital Filter block **Input port(s)** option for HDL code generation.
- The Digital Filter block supports complex data for all filter structures except decimators and interpolators. See “Complex Coefficients and Data Support for the Digital Filter and Biquad Filter Blocks” on page 11-107.

Discrete FIR Filter Requirements and Restrictions

- “General Guidelines” on page 11-35
- “Multichannel Filter Support” on page 11-36
- “Programmable Filter Support” on page 11-36

General Guidelines.

- The coder does not support unsigned inputs for the Discrete FIR Filter block.
- **Initial conditions** must be set to zero. HDL code generation is not supported for nonzero initial states.
- The coder does not support the following options of the Discrete FIR Filter block:
 - **Filter Structure** : Lattice MA

Multichannel Filter Support. The coder supports the use of vector inputs for Discrete FIR Filters.

- 1 Connect vector signals to Discrete FIR block input port.
- 2 Specify **Input processing** as `Elements as channels (sample based)`.
- 3 Specify architecture and implementation parameters.
- 4 Specify channel sharing option as on for fully parallel support.
- 5 Generate HDL code.

Programmable Filter Support. The coder supports programmable filters for Discrete FIR Filters. Fully parallel and applicable serial architectures are supported.

- 1 Select **Input port(s)** as coefficient source on the filter block mask.
- 2 Connect the coefficient port with a vector signal.
- 3 Specify the implementation architecture and parameters from the HDL Coder property interface.
- 4 Generate HDL code.

Programmable filters are not supported for:

- Implementations for which you specify the coefficients by dialog parameters (for example, complex input and coefficients with serial architecture)
- Distributed Arithmetic (DA)
- `CoeffMultipliers` as `csd` or `factored-csd`

For an example, see `Generate HDL Code for FIR Programmable Filter`.

Discrete Transfer Fcn Requirements and Restrictions

The following limitations apply to HDL code generation from the Discrete Transfer Fcn block:

- You must use the **Inherit: Inherit via internal rule** option for data type propagation if, and only if, the input data type is double.
- Frame, matrix, and vector input data types are not supported.
- The leading denominator coefficient (a0) must be 1 or -1.

The Discrete Transfer Fcn block cannot participate in the following optimizations:

- Resource sharing
- Distributed pipelining

Discrete-Time Integrator Requirements and Restrictions

- Use of state ports is not supported for HDL code generation. Clear the **Show state port** option.
- Use of external initial conditions is not supported for HDL code generation. Set **Initial condition source** to Internal.
- Width of input and output signals must not exceed 32 bits.

FIR Decimation Requirements and Restrictions

Initial conditions must be set to zero. HDL code generation is not supported for nonzero initial states.

See also “Multirate CIC Decimation and Multirate FIR Decimation Blocks Requirements and Restrictions” on page 11-39.

FIR Interpolation Requirements and Restrictions

Initial conditions must be set to zero. HDL code generation is not supported for nonzero initial states.

See also “Multirate CIC Interpolation and Multirate FIR Interpolation Blocks Requirements and Restrictions” on page 11-40.

General Multiplexed Interleaver and Deinterleaver Block Requirements and Restrictions

Shift Register Based Implementations. The default implementations for the General Multiplexed Interleaver and Deinterleaver blocks are shift register based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` to 'none'.

Note that when you set `ResetType` to 'none', reset is not applied to the shift registers. 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 spurious test bench errors, determine the number of samples required to fully load the shift registers. Then, set the **Ignore output data checking (number of samples)** option accordingly. (You can use the `IgnoreDataChecking` property for this purpose, if you are using the command-line interface)

LMS Filter Usage and Restrictions

By default, the LMS Filter implementation uses a linear sum for the FIR section of the filter.

The LMS Filter implements a tree summation (which has a shorter critical path) under the following conditions:

- The LMS Filter is used with real data
- The word length of the Accumulator W^u data type is at least $\text{ceil}(\log_2(\text{filter length}))$ bits wider than the word length of the Product W^u data type
- The Accumulator W^u data type has the same fraction length as the Product W^u data type

The LMS Filter block has the following restrictions for HDL code generation:

- The coder does not support the Normalized LMS algorithm of the LMS Filter.
- The Reset port supports only Boolean and unsigned inputs.
- The Adapt port supports only Boolean inputs.

- **Filter length** must be greater than or equal to 2.

Magnitude-Angle to Complex Block Requirements and Restrictions

The Magnitude-Angle to Complex block supports HDL code generation when you set **Approximation method** to CORDIC.

Multirate CIC Decimation and Multirate FIR Decimation Blocks Requirements and Restrictions

The following requirements apply to both the Multirate CIC Decimation and Multirate FIR Decimation blocks:

- The coder supports both **Coefficient source** options (**Dialog parameters** or **Multirate filter object (MFILT)**).
- When you select **Multirate filter object (MFILT)**:
 - You can enter either a filter object name or a direct filter specification in the **Multirate filter variable** field.
- Vector and frame inputs are not supported for HDL code generation.

For the Multirate FIR Decimation block:

- When you select **Multirate filter object (MFILT)**, the filter object specified in the **Multirate filter variable** field must be either a `mfilt.firdecim` object or a `mfilt.firtdecim` object. If you specify some other type of filter object, an error will occur.
- When you select **Dialog parameters**, the following fixed-point options are not supported for HDL code generation:
 - Slope and Bias scaling
 - Inherit via internal rule

For the Multirate CIC Decimation block:

- When you select **Multirate filter object (MFILT)**, the filter object specified in the **Multirate filter variable** field must be a `mfilt.cicdecim` object. If you specify some other type of filter object, an error will occur.

- When you select **Dialog parameters**, the **Filter Structure** option Zero-latency decimator is not supported for HDL code generation. Select Decimator in the **Filter Structure** pulldown menu.

Multirate CIC Interpolation and Multirate FIR Interpolation Blocks Requirements and Restrictions

The following requirements apply to both the Multirate CIC Interpolation and Multirate FIR Interpolation blocks:

- The coder supports both **Coefficient source** options (**Dialog parameters** or **Multirate filter object (MFILT)**).
- When you select **Multirate filter object (MFILT)**:
 - You can enter either a filter object name or a direct filter specification in the **Multirate filter variable** field.
- Vector and frame inputs are not supported for HDL code generation.

For the Multirate FIR Interpolation block:

- When you select **Multirate filter object (MFILT)**, the filter object specified in the **Multirate filter variable** field must be a `mfilt.firinterp` object. If you specify some other type of filter object, an error will occur.
- When you select **Dialog parameters**, the following fixed-point options are not supported for HDL code generation:
 - **Coefficients**: Slope and Bias scaling
 - **Product Output**: Inherit via internal rule

For the Multirate CIC Interpolation block:

- When you select **Multirate filter object (MFILT)**, the filter object specified in the **Multirate filter variable** field must be a `mfilt.cicinterp` object. If you specify some other type of filter object, an error will occur.
- When you select **Dialog parameters**, the **Filter Structure** option Zero-latency interpolator is not supported for HDL code generation. Select Interpolator in the **Filter Structure** dropdown menu.

NCO Block Requirements and Restrictions

Note HDL support for the NCO block will be removed in a future release. Use the NCO HDL Optimized block instead.

Inputs:

- The phase increment and phase offset support only integer or fixed-point data types.
- The phase increment and phase offset can be either scalar or vector values.

Outputs:

- The coder supports only fixed point data types for the quantization error (Qerr) port and output signals.

Parameters:

- The coder does not support **Add internal dither** for vector inputs
- If **Quantize phase** is selected, **Number of quantized accumulator bits** should be greater than or equal to 4. A checkhdl error occurs when there are fewer than 4 quantized accumulator bits.
- If **Quantize phase** is deselected, the accumulator **Word length** should be greater than or equal to 4. A checkhdl error occurs when there are fewer than 4 accumulator bits.

PN Sequence Generator Block Requirements and Restrictions

Inputs:

- You can select Input port as the **Output mask source** on the block. However, in this case the Mask input signal must be a vector of data type `ufix1`.
- If **Reset on nonzero input** is selected, the input to the Rst port must have data type Boolean.

Outputs:

- Outputs of type `double` are not supported for HDL code generation. All other output types (including bit packed outputs) are supported.

Reciprocal Sqrt Block Requirements and Restrictions

When using this block for HDL code generation, set the **Method** parameter to `Newton-Raphson`.

Rectangular QAM Demodulator Baseband Block Requirements and Restrictions

The coder has the following requirements and restrictions for the Rectangular QAM Demodulator Baseband block:

- The block does not support single or double data types for HDL code generation.
- The coder supports the following **Output type** options:
 - Integer
 - Bit is supported only if the **Decision Type** selected is `Hard decision`.
- The coder requires that **Normalization Method** be set to `Minimum Distance Between Symbols`, with a **Minimum distance** of 2.
- The coder requires that **Phase offset (rad)** be set to a value that is multiple a of $\pi/4$.

Rectangular QAM Modulator Baseband Block Requirements and Restrictions

The coder has the following requirements and restrictions for the Rectangular QAM Modulator Baseband block:

- The block does not support single or double data types for HDL code generation.
- When **Input Type** is set to `Bit`, the block does not support HDL code generation for input types other than `boolean` or `ufix1`.

The Rectangular QAM Modulator Baseband block does not support HDL code generation when the input type is set to `Bit` but the block input is actually multibit (`uint16`, for example).

Sine Wave Block Requirements and Restrictions

For HDL code generation, you must select the following Sine Wave block settings:

- **Computation method:** Table lookup
- **Sample mode:** Discrete

Output:

- The output port cannot have data types `single` or `double`.

Trigonometric Function Block Requirements and Restrictions

The Trigonometric Function block supports HDL code generation for the following functions:

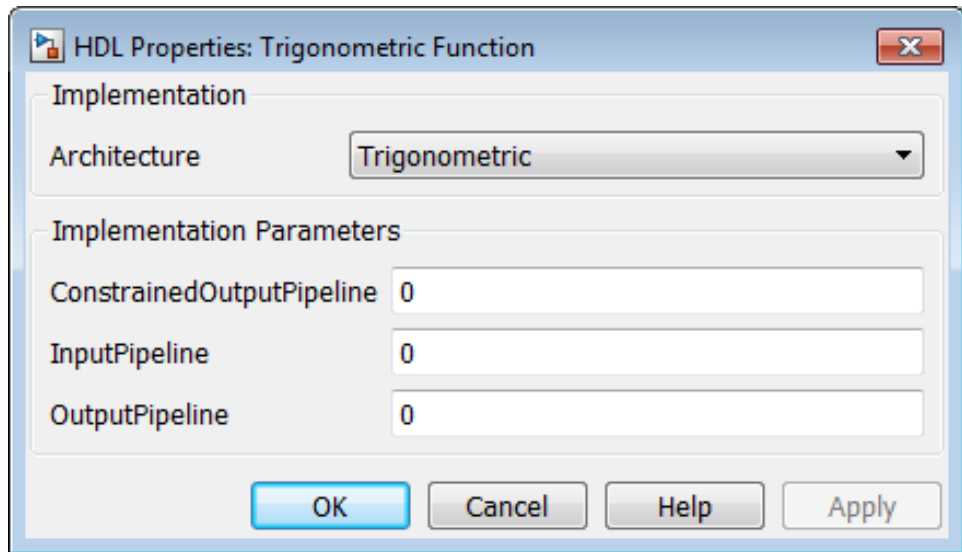
Trigonometric Function Block Implementation	Supported Functions	Supported Approximation Methods
default Trigonometric	<code>sin</code>	CORDIC
	<code>cos</code>	CORDIC
	<code>cos + jsin</code>	CORDIC
	<code>sincos</code>	CORDIC

For the `sin` and `cos` functions, unsigned data types are supported for CORDIC approximations.

The coder gives an error when:

- You select an unsupported function on the Trigonometric Function block.
- You select an **Approximation method** other than CORDIC.

Use the default implementation for the Trigonometric Function block, as shown in the following figure.



See also Trigonometric Function, cordicsin, cordiccos, and cordicsincos.

Viterbi Decoder Block Requirements and Restrictions

- “General Guidelines” on page 11-44
- “Limitations” on page 11-45
- “Input and Output Data Types” on page 11-45
- “Pipelining the Traceback Unit” on page 11-46
- “RAM-Based Traceback” on page 11-46
- “Viterbi Decoder Example” on page 11-49

General Guidelines. The coder currently supports the following features of the Viterbi Decoder block:

- Non-recursive encoder/decoder with feed-forward trellis and simple shift register generation configuration
- Sample based input
- Decoder rates from 1/2 to 1/7
- Constraint length from 3 to 9

Limitations. When you generate code for the Viterbi Decoder block, observe the following limitations:

- **Punctured code:** Do not select this option. Punctured code requires frame-based input, which the coder does not support.
- **Decision type:** the coder does not support the Unquantized decision type.
- **Error if quantized input values are out of range:** The coder does not support this option.
- **Operation mode:** The coder supports only the Continuous mode.
- **Enable reset input port:** HDL support is provided when you enable both **Enable reset input port** and **Delay reset action to next time step**. You must select Continuous operation mode.

Input and Output Data Types.

- When **Decision type** is set to `Soft decision`, the HDL implementation of the Viterbi Decoder block supports fixed-point inputs and output. For input, the fixed-point data type must be `ufixN`, where N is the number of soft decision bits. Signed built-in data types (`int8`, `int16`, `int32`) are not supported. For output, the HDL implementation of the Viterbi Decoder block supports block-supported output data types.
- When **Decision type** is set to `Hard decision`, the block supports input with data types `ufix1` and `Boolean`. For output, the HDL implementation of the Viterbi Decoder block supports block-supported output data types.
- The HDL implementation of the Viterbi Decoder block does not support double and single input data types are not supported. The block does not support floating point output for fixed-point inputs.

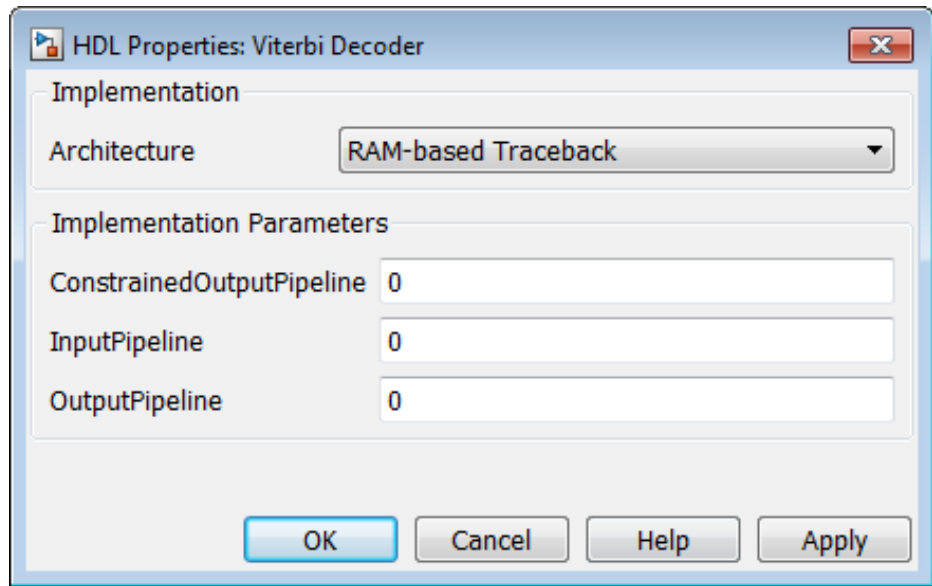
Pipelining the Traceback Unit. The Viterbi Decoder block decodes every bit by tracing back through a traceback depth that you define for the block. The block implements a complete traceback for each decision bit, using registers to store the minimum state index and branch decision in the traceback decoding unit. You can specify that the traceback decoding unit be pipelined in order to improve the speed of the generated circuit. You can add pipeline registers to the traceback unit by specifying the number of traceback stages per pipeline register. To do this, use the `TracebackStagesPerPipeline` implementation parameter.

The `TracebackStagesPerPipeline` implementation parameter lets you balance the circuit performance based on system requirements. A smaller parameter value indicates the requirement to add more registers to increase the speed of the traceback circuit. Increasing the number results in a lower number of registers along with a decrease in the circuit speed.

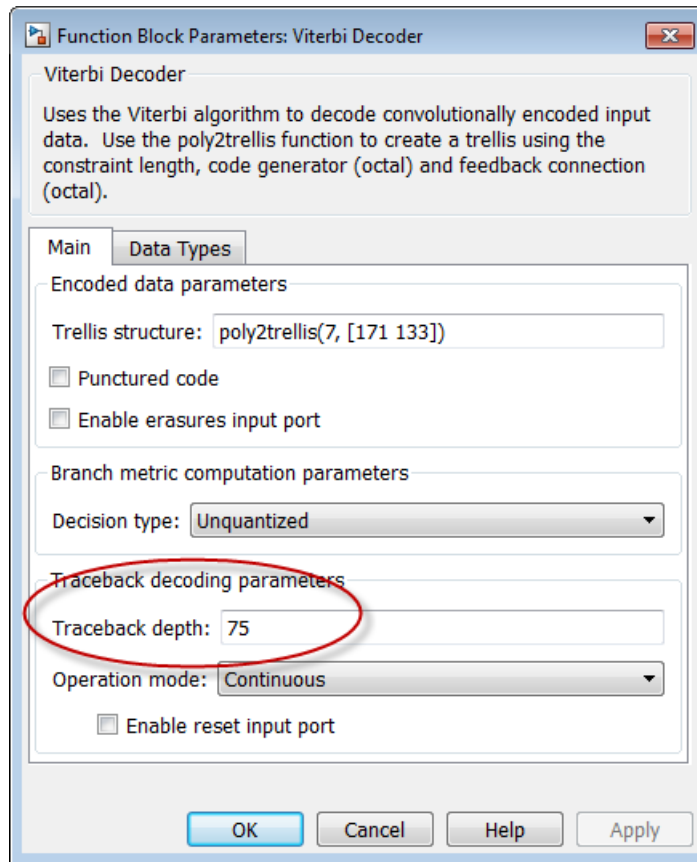
See the “HDL Code Generation for Viterbi Decoder” example model for an example using `TracebackStagesPerPipeline`.

RAM-Based Traceback. Instead of using registers, you can choose to use RAMs to save the survivor branch information.

- 1 Set the HDL Architecture property of the Viterbi Decoder block to RAM-based Traceback.



- 2 Set the traceback depth on the Viterbi Decoder block mask.



RAM-based traceback and register-based traceback differ in the following ways:

- The RAM-based implementation traces back through one set of data to find the initial state to decode the previous set of data. The register-based implementation combines the traceback and decode operations into one step and uses the best state found from the minimum operation as the decoding initial state.
- RAM-based implementation traces back through M samples, decodes the previous M bits in reverse order, and releases one bit in order at each clock

cycle, whereas the register-based implementation decodes one bit after a complete traceback.

Because of the differences in the two traceback algorithms, the RAM-based implementation produces different numerical results than the register-based traceback. A longer traceback depth, for example, 10 times the constraint length, is recommended in the RAM-based traceback to achieve a similar bit error rate (BER) as the register-based implementation. The size of RAM required for the implementation depends on the trellis and the traceback depth.

See HDL Code Generation for Viterbi Decoder.

Viterbi Decoder Example. The “HDL Code Generation for Viterbi Decoder” example demonstrates HDL code generation for a fixed-point Viterbi Decoder block, with pipelined traceback decoding. To open the example, type the following command:

```
showdemo commviterbihdl_m
```

Restrictions on Use of Blocks in the Test Bench

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™

Block Implementation Parameters

In this section...
“Overview” on page 11-50
“BalanceDelays” on page 11-51
“ConstMultiplierOptimization” on page 11-52
“CoeffMultipliers” on page 11-54
“ConstrainedOutputPipeline” on page 11-56
“Distributed Arithmetic Implementation Parameters for Digital Filter Blocks” on page 11-57
“DistributedPipelining” on page 11-70
“FlattenHierarchy” on page 11-71
“InputPipeline” on page 11-73
“InstantiateFunctions” on page 11-74
“LoopOptimization” on page 11-75
“MapPersistentVarsToRAM” on page 11-77
“OutputPipeline” on page 11-80
“Pipelining Implementation Parameters for Filter Blocks” on page 11-81
“RAM” on page 11-85
“ResetType” on page 11-86
“ShiftRegister” on page 11-88
“UseRAM” on page 11-90
“Speed vs. Area Optimizations for FIR Filter Implementations” on page 11-95
“Interface Generation Parameters” on page 11-102

Overview

Block implementation parameters enable you to control details of the code generated for specific block implementations. See “Set and View HDL Block

Parameters” on page 10-2 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.

The `BalanceDelays` options for a subsystem are listed in the following table.

BalanceDelays Setting	Description
'inherit' (default)	Use the delay balancing setting of the parent subsystem. If this
'on'	Balance delays for this subsystem.
'off'	Do not balance delays for this subsystem, even if the parent subsystem has delay balancing enabled.

Prerequisites for Subsystem Delay Balancing

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 canonic signed digit (CSD) or factored CSD optimizations for processing coefficient multiplier operations in code generated for the following blocks:

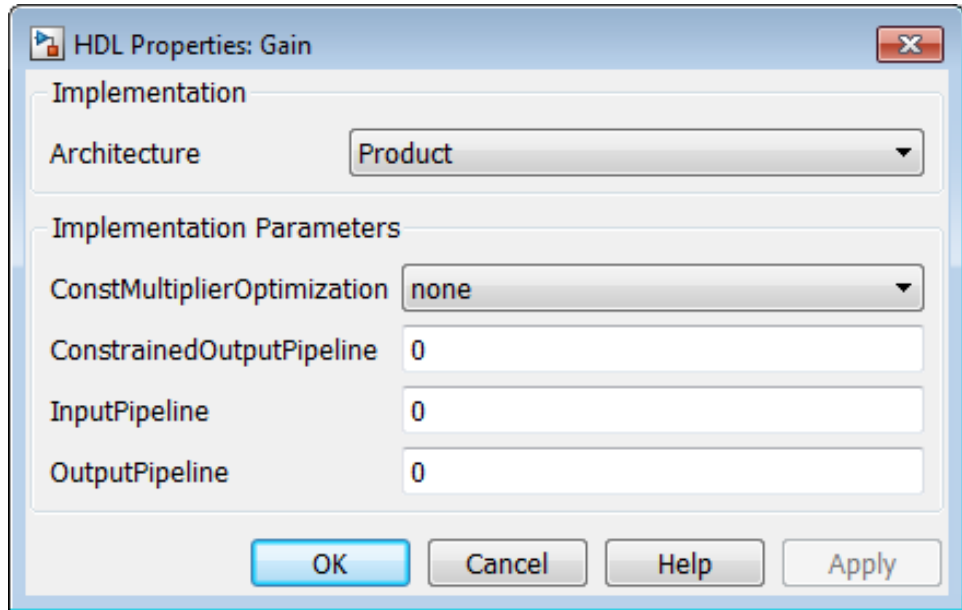
- Gain
- Stateflow chart
- Truth Table
- MATLAB Function

The following table shows the `ConstMultiplierOptimization` parameter values.

Implementations	Parameters	Description
default	'ConstMultiplierOptimization', 'none' <i>(Default)</i> 'ConstMultiplierOptimization', 'CSD'	By default, the coder does not perform CSD or FCSD optimizations. Code generated for the Gain block retains multiplier operations. When you specify this option, the generated code decreases the area used by the model while maintaining or increasing clock
		speed, using canonic signed digit (CSD) techniques. CSD replaces multiplier operations with add and subtract operations. CSD minimizes

Implementations	Parameters	Description
	'ConstMultiplierOptimization', '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.
	'ConstMultiplierOptimization', 'auto'	When you specify this option, the 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 following figure shows the ConstMultiplierOptimization option in the HDL Block properties dialog box.

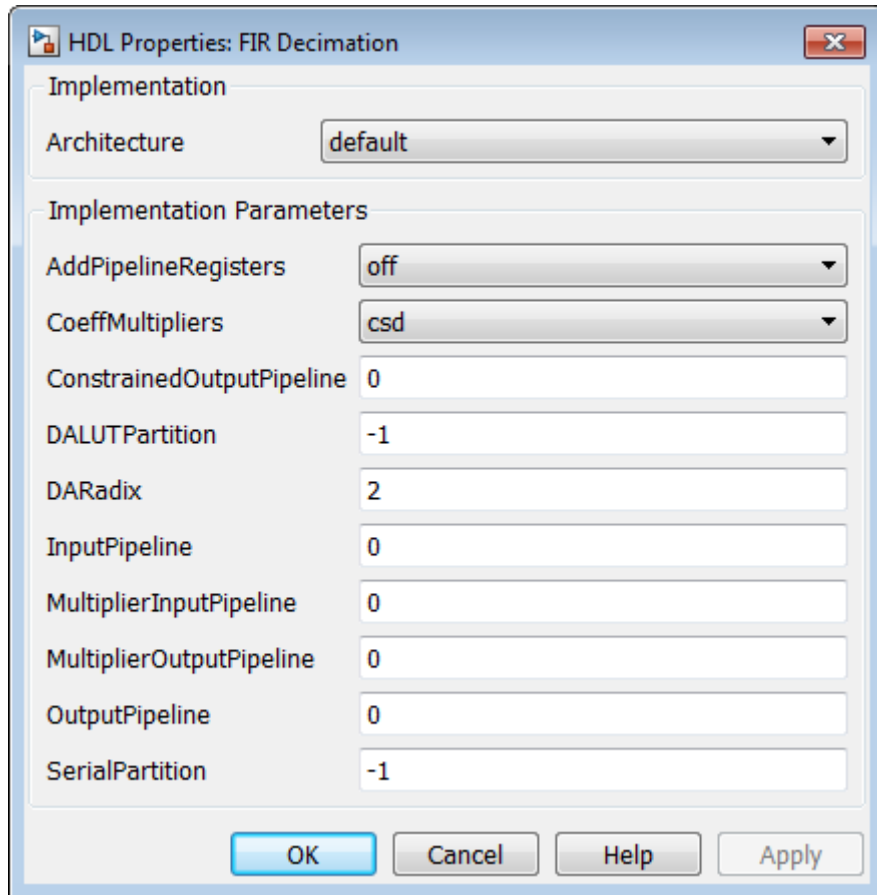


CoeffMultipliers

The `CoeffMultipliers` implementation parameter lets you specify use of canonic 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.

In the following figure, the HDL Block Properties dialog box specifies that code generated for a FIR Decimation block in the model uses the CSD optimization.



The coder supports `CoeffMultipliers` for the filter block implementations shown in the following table.

Block	Implementation
dsparch4/Biquad Filter	default
dsparch4/Digital Filter	default

Block	Implementation
dspmlti4/FIR Decimation	default
dspmlti4/FIR Interpolation	default
simulink/Discrete/Discrete FIR Filter	default

ConstrainedOutputPipeline

The `ConstrainedOutputPipeline` parameter enables you to specify a nonnegative number of registers at the outputs of a block.

Use constrained output pipelining when you want to place registers at specific locations in your design. Distributed pipelining does not redistribute registers you specify with constrained output pipelining.

The coder redistributes existing delays within your design to try to meet your constraint. When 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 registers to your design using input or 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, on the command line, enter:

```
hdlset_param(path_to_block, 'ConstrainedOutputPipeline', number_of_output_re
```

For example, to constrain 6 registers at the output ports of a subsystem, `subsys`, in your model, `mymodel`, enter:

```
hdlset_param('mymodel/subsys', 'ConstrainedOutputPipeline', 6)
```


See Also

- “Constrained Output Pipelining” on page 15-63

Distributed Arithmetic Implementation Parameters for Digital Filter Blocks

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.

The coder supports distributed arithmetic (DA) implementations for the blocks described in the following table.

Block	Implementation	FIR Structures That Support DA
dsparch4/Digital Filter	default	<ul style="list-style-type: none"> • dfilt.dffir • dfilt.dfsymfir • dfilt.dfasymdir
simulink/Discrete/Discrete FIR Filter	default	<ul style="list-style-type: none"> • dfilt.dffir • dfilt.dfsymfir • dfilt.dfasymdir

Block	Implementation	FIR Structures That Support DA
dspmlti4/FIR Decimation	default	mfilt.firdecim
dspmlti4/FIR Interpolation	Fully Parallel (default) Partly Serial Fully Serial Distributed Arithmetic (DA)	mfilt.firinterp

This section briefly summarizes the operation of DA. For references on the theoretical foundations of DA, see “Further References” on page 11-69.

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-wide 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.

Improving Performance with Parallelism

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 control the degree of parallelism by specifying the *DARadix* implementation parameter. *DARadix* lets you specify the number of bits processed simultaneously in DA (see “*DARadix* Implementation Parameter” on page 11-67).

Reducing LUT Size

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 control how the LUT is partitioned in DA by specifying the `DALUTPartition` implementation parameter (see “`DALUTPartition Implementation Parameter`” on page 11-60).

Requirements and Considerations for Generating Distributed Arithmetic Code

You can control how DA code is generated by using the `DALUTPartition` and `DARadix` implementation parameters. Before using these parameters, review the following general requirements, restrictions, and other considerations for generation of DA code.

Requirements Specific to Filter Type. 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:

- “`DALUTPartition Implementation Parameter`” on page 11-60
- “`DARadix Implementation Parameter`” on page 11-67

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.

DALUTPartition Implementation Parameter

`DALUTPartition` enables DA code generation and specifies the number and size of LUT partitions used for DA.

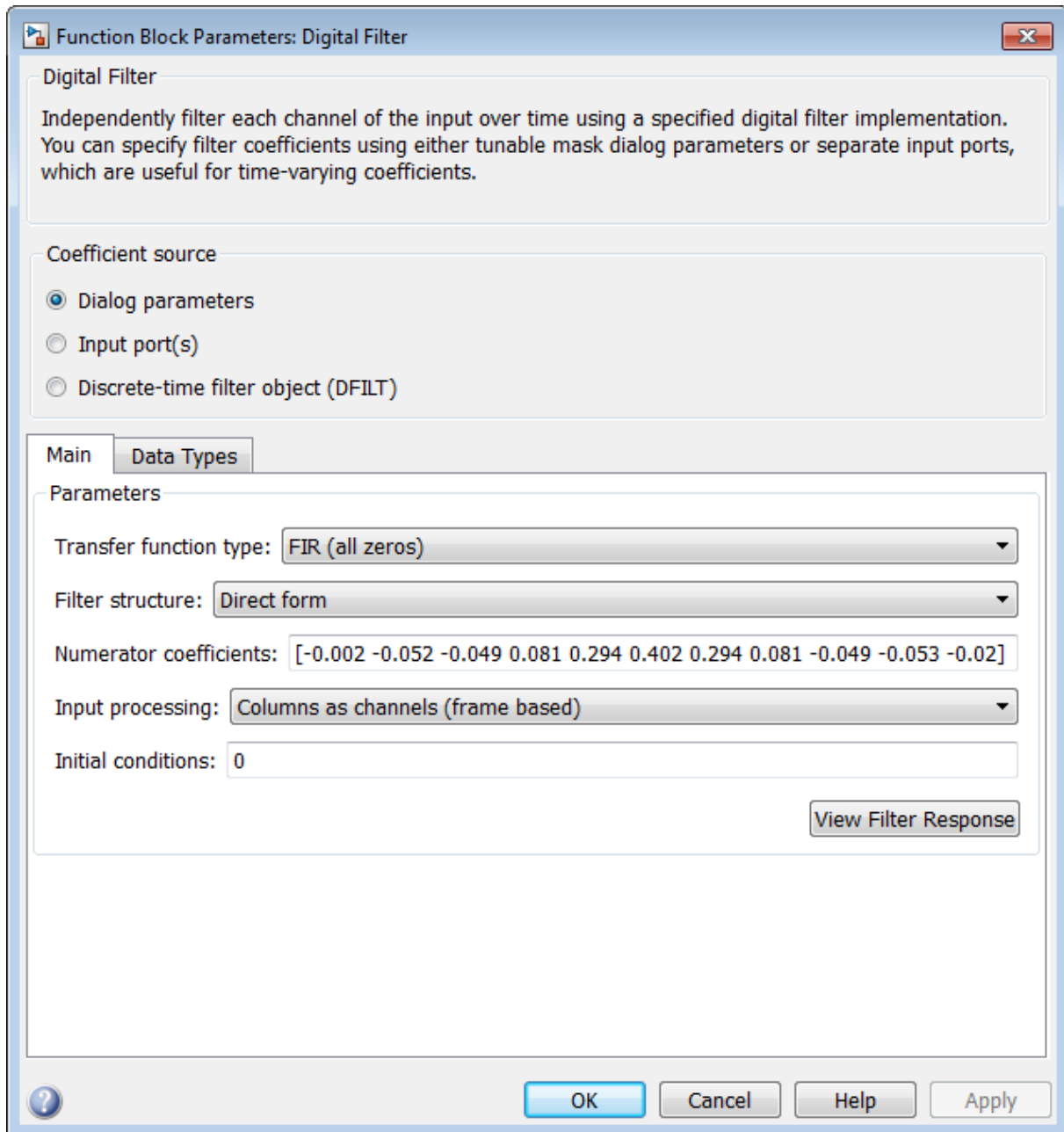
Specify LUT partitions as a vector of integers [p1 p2 . . . pN] where:

- 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 (see “Specifying DALUTPartition for Single-Rate Filters” on page 11-61).

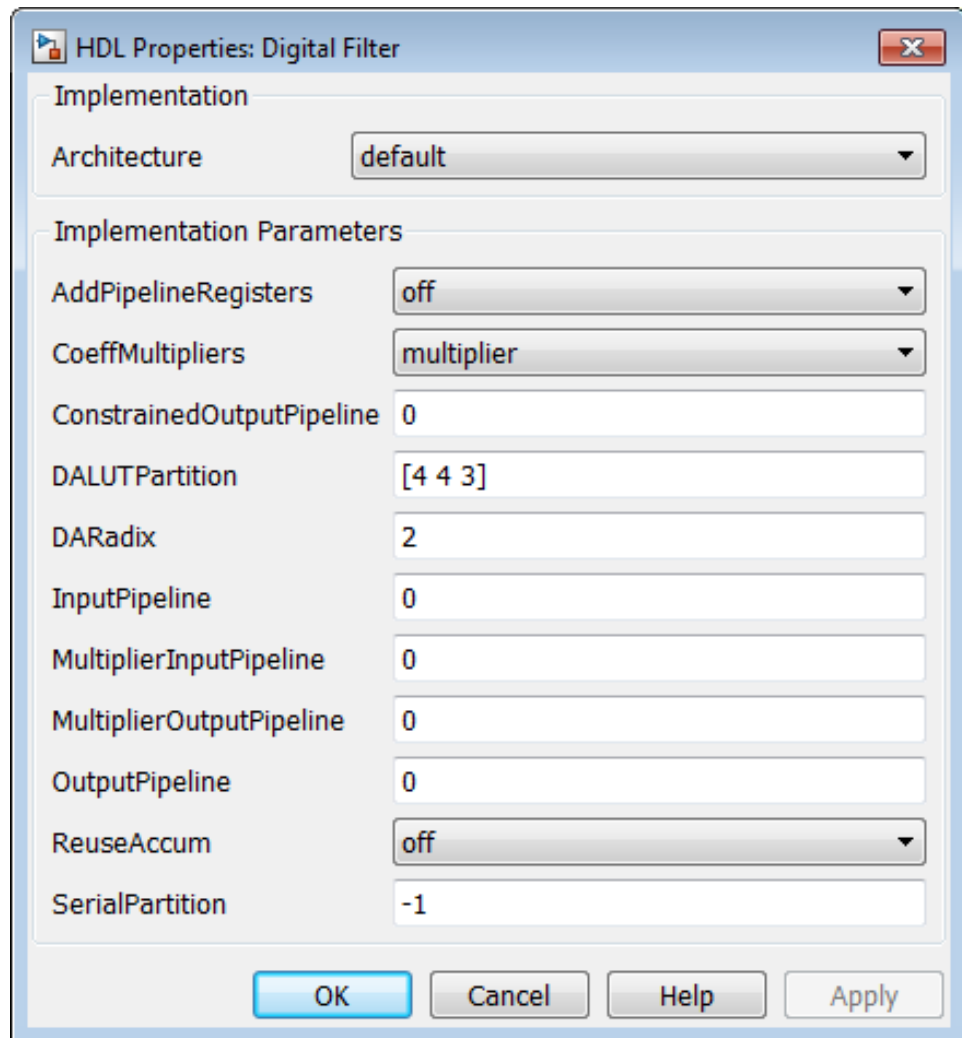
Specifying DALUTPartition for Single-Rate Filters. To determine the LUT partition for one of the supported single-rate filter types, calculate FL as shown in the following table. Then, specify the partition as a vector whose elements sum to FL.

Filter Type	Filter Length (FL) Calculation
dfilt.dffir	FL = length(find(Hd.numerator~= 0))
dfilt.dfsymfir dfilt.dfasymfir	FL = ceil(length(find(Hd.numerator~= 0))/2)

The following figure shows a Digital Filter configured for a direct form FIR filter of length 11.



The following figure shows how to set one possible LUT partitioning for this filter:



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. For example, the following figure shows a Digital Filter configuration for a direct form FIR filter of length 5.

Function Block Parameters: Digital Filter

Digital Filter

Independently filter each channel of the input over time using a specified digital filter implementation. You can specify filter coefficients using either tunable mask dialog parameters or separate input ports, which are useful for time-varying coefficients.

Coefficient source

Dialog parameters

Input port(s)

Discrete-time filter object (DFILT)

Main | **Data Types**

Parameters

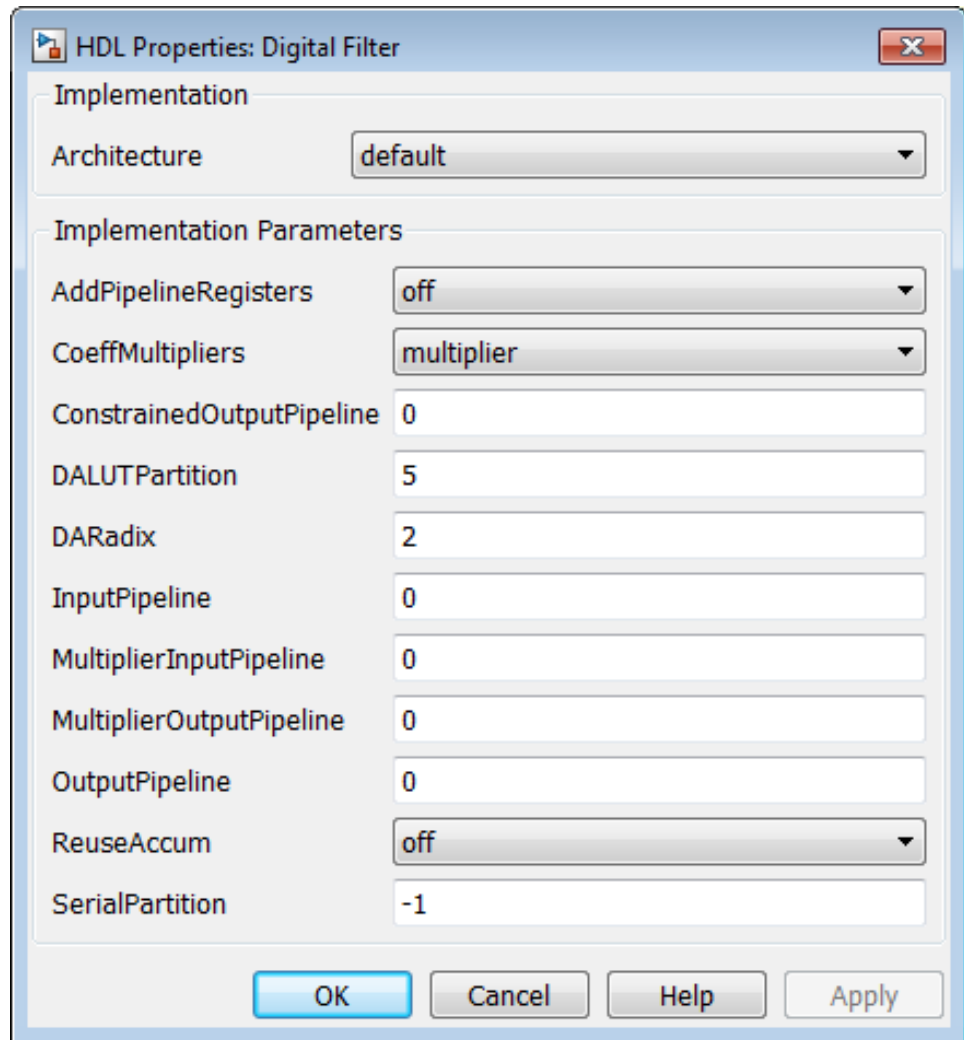
Transfer function type:

Filter structure:

Numerator coefficients:

Input processing:

Initial conditions:



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
<p><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.</p>	$FL = \text{size}(\text{polyphase}(H_m), 2)$
<p><i>Matrix</i>: determine the subfilter length FL_i based on the polyphase decomposition of the filter, as shown in the Filter Length (FL) Calculation column to the right. Specify the LUT partition for each subfilter as a row vector whose elements sum to FL_i.</p>	<pre>p = polyphase(Hm); FL_i = length(find(p(i,:)));</pre> <p>where i is the index to the ith row of the polyphase matrix of the multirate filter. The ith row of the matrix p represents the ith subfilter.</p>

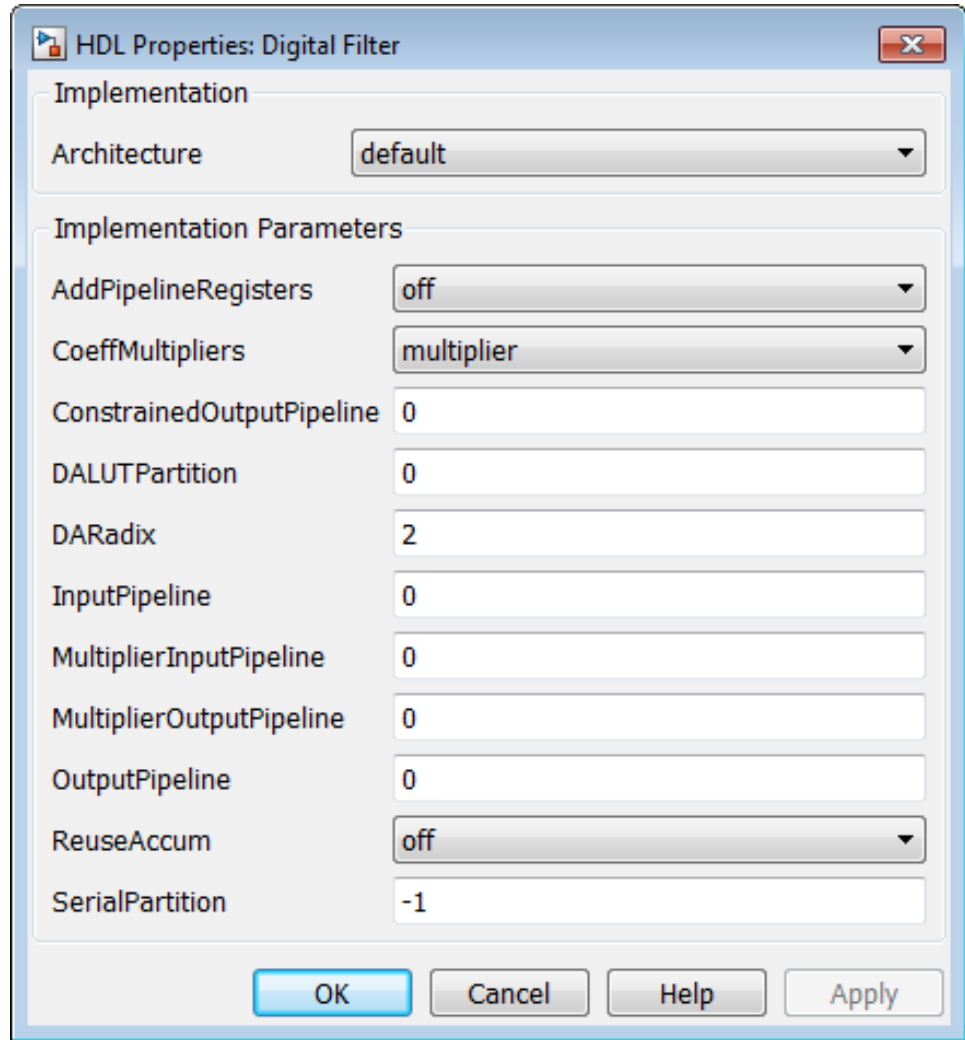
DARadix Implementation Parameter

DARadix specifies 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.

You can set the DARadix implementation parameter in the HDL Properties dialog for a filter block as shown in the following figure.



Note When setting a `DARadix` value for symmetrical (`dfilt.dfsymfir`) and asymmetrical (`dfilt.dfasymfir`) filters, see “Considerations for Symmetrical and Asymmetrical Filters” on page 11-69.

Special Cases

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.
- The 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.

Holding Input Data in a Valid State. In filters with a DA architecture, data can be delivered to the outputs N cycles ($N \geq 2$) later than the inputs. You can use the `HoldInputDataBetweenSamples` model property to determine how long (in terms of clock cycles) input data values are held in a valid state, as follows:

- When `HoldInputDataBetweenSamples` is set 'on' (the default), input data values are held in a valid state across N clock cycles.
- When `HoldInputDataBetweenSamples` is set 'off', data values are held in a valid state for only one clock cycle. For the next $N-1$ cycles, data is in an unknown state (expressed as 'X') until the next input sample is clocked in.

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

DistributedPipelining

The `DistributedPipelining` parameter enables pipeline register distribution, a speed optimization that enables you to increase your clock speed by reducing your critical path.

The following table shows the effect of the `DistributedPipelining` and `OutputPipeline` parameters.

DistributedPipelining	OutputPipeline, nStages	Result
'off' (default)	Unspecified (nStages defaults to 0)	The coder does not insert pipeline registers.
	nStages > 0	The coder inserts nStages output registers at the output of the subsystem, MATLAB Function block, or Stateflow chart.
'on'	Unspecified (nStages defaults to 0)	The coder does not insert pipeline registers. <code>DistributedPipelining</code> has no effect.
	nStages > 0	The coder distributes nStages 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:

- “Use file I/O to read/write test bench data” on page 9-103
 - IgnoreDataChecking
-

See Also

- “Distributed Pipelining and Hierarchical Distributed Pipelining” on page 15-53
- “Specify Distributed Pipelining” on page 15-56
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 20-42

FlattenHierarchy

FlattenHierarchy enables you to remove subsystem hierarchy from the HDL code generated from your design.

The FlattenHierarchy options for a subsystem are listed in the following table.

FlattenHierarchy Setting	Description
'inherit' (default)	Use the hierarchy flattening setting of the parent subsystem. If this
'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:

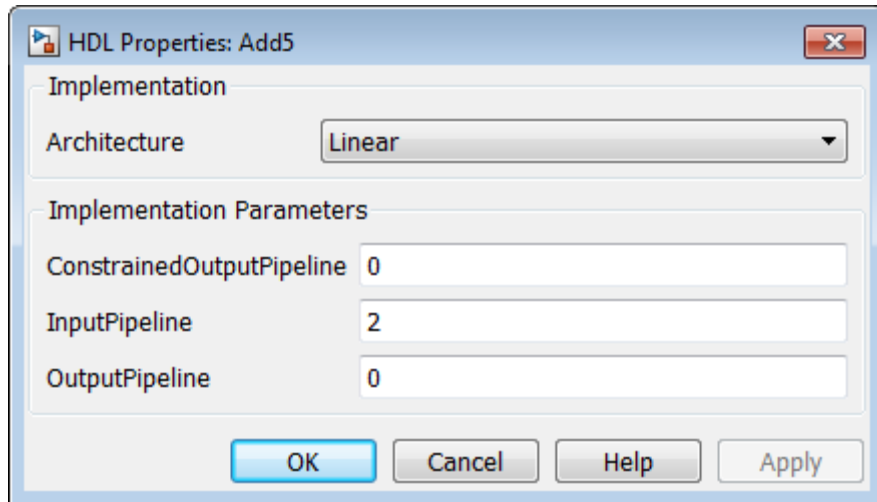
- Atomic and instantiated in the design more than once.
- A black box implementation or model reference.
- An enabled or triggered subsystem.
- A masked subsystem.

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 figure shows the `InputPipeline` parameter set to 2 in the HDL Properties dialog box for an Add block .



The following code specifies an input pipeline depth of two stages for each Sum block in the model:

```
sblocks = find_system(gcf, 'BlockType', 'Sum');
for ii=1:length(sblocks),hdlset_param(sblocks{ii},'InputPipeline', 2), end;
```

When generating code for pipeline registers, the 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. The 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.

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.

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.

The LoopOptimization options for the MATLAB Function block are listed in the following table.

LoopOptimization Setting	Description
'none' (default)	Do not optimize loops.
'Unrolling'	Unroll loops.
'Streaming'	Stream loops.

How to Optimize MATLAB Function Block For Loops

To select a loop optimization using the HDL Block Properties dialog box:

- 1 Right-click the MATLAB Function block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **LoopOptimization**, select **none**, **Unrolling**, or **Streaming**.

To select a loop optimization from the command line, use `hdlset_param`. For example, to turn on loop streaming for a MATLAB Function block, `my_mlfm`:

```
hdlset_param('my_mlfm', 'LoopOptimization', 'Streaming')
```

See also `hdlset_param`.

Limitations for MATLAB Function Block Loop Optimization

The coder cannot stream a loop if:

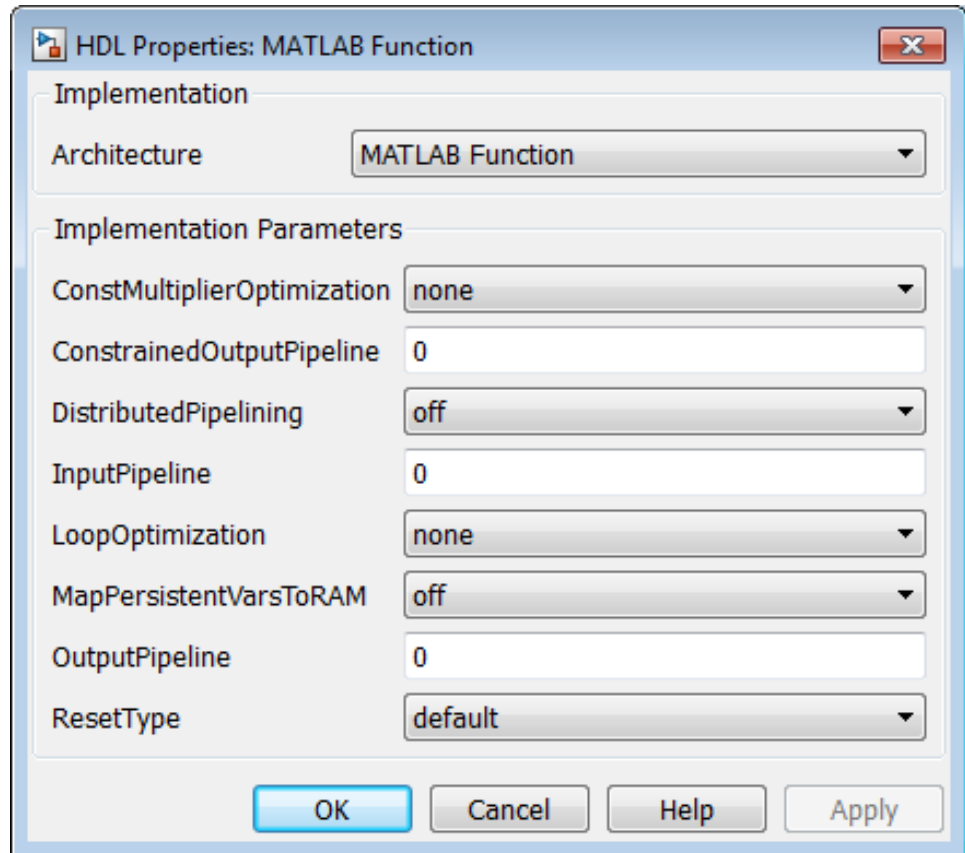
- The loop index counts down. The loop index must increase by 1 on each iteration.
- There are 2 or more nested loops at the same level of hierarchy within another loop.
- Any particular persistent variable is updated both inside and outside a loop.

The 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.

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.



Mapping Behavior for Persistent Arrays

MapPersistentVarsToRAM Setting	Mapping Behavior
off	Persistent arrays map to registers in the generated HDL code.
on	<p>A persistent array maps to a block RAM when all of the following conditions are true:</p> <ul style="list-style-type: none"> • 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. • 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: <pre data-bbox="556 885 808 1041"> if (mod(i,2) > 0) a = r1(u); else r1(i) = u; end </pre> <p>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:</p> <pre data-bbox="556 1267 808 1458"> temp = mod(i,2); if (temp > 0) a = r1(u); else r1(i) = u; end </pre> <ul style="list-style-type: none"> • The persistent array value depends on external inputs.

MapPersistentVarsToRAM Setting	Mapping Behavior
	<p>For example, in the following code, <code>bigarray</code> does not map to RAM because it does not depend on <code>u</code>:</p> <pre>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;</pre> <ul style="list-style-type: none"> • <code>RAMSize</code> is greater than or equal to the <code>RAMMappingThreshold</code> value. <code>RAMSize</code> is the product <code>NumElements * WordLength * Complexity</code>. <ul style="list-style-type: none"> ▪ <code>NumElements</code> is the number of elements in the array. ▪ <code>WordLength</code> is the number of bits that represent the data type of the array. ▪ <code>Complexity</code> is 2 for arrays with a complex base type; 1 otherwise. <p>If any of the above conditions is false, the persistent array maps to a register in the HDL code.</p>

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” on page 9-22.

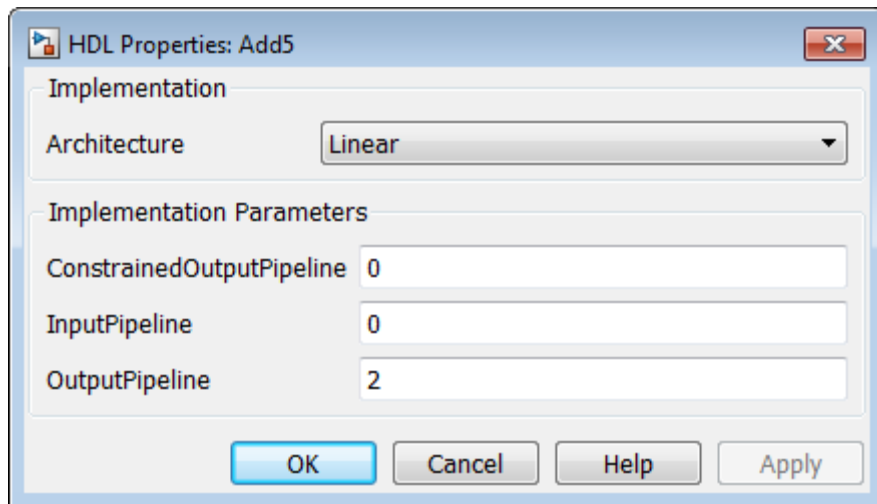
Example

For an example that shows how to map persistent array variables to RAM in a MATLAB Function block, see “RAM Mapping with the MATLAB Function Block” on page 15-45.

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 figure shows the OutputPipeline parameter set to 2 in the HDL Properties dialog box for an Add block .



The following code specifies an output pipeline depth of two stages for each Sum block in the model:

```
sblocks = find_system(gcf, 'BlockType', 'Sum');
```



```
for ii=1:length(sblocks),hdlset_param(sblocks{ii},'OutputPipeline', 2), end;
```

When generating code for pipeline registers, the 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.

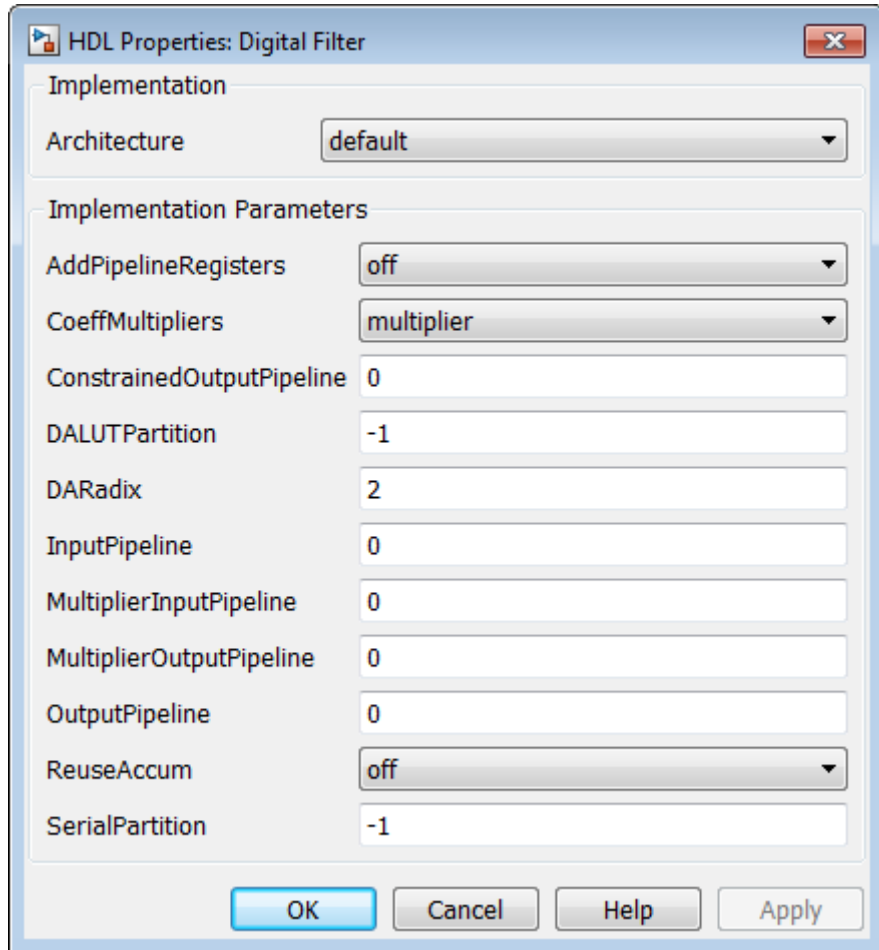
See also “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 20-42.

Pipelining Implementation Parameters for Filter Blocks

The following implementation parameters for filter blocks provide block-specific pipelining support.

- `AddPipelineRegisters` (Default: `off`): Inserts a pipeline register between stages of computation in a filter.
- `MultiplierInputPipeline` (Default: `0`): Generates a specified number of pipeline stages at multiplier inputs for FIR filter structures.
- `MultiplierOutputPipeline` (Default: `0`): Generates a specified number of pipeline stages at multiplier outputs for FIR filter structures.

The following figure shows these parameters, set to their default values, in the HDL Block Properties dialog box for a Digital Filter block.



The following table summarizes the filter blocks that support one or more of these parameters:

Filter Block	Supports AddPipelineRegisters	Supports MultiplierInputPipeline	Supports MultiplierOutputPipeline
Digital Filter	Yes	Yes	Yes
Discrete FIR Filter	Yes	Yes	Yes

Filter Block	Supports AddPipelineRegisters	Supports MultiplierInputPipeline	Supports MultiplierOutputPipeline
FIR Decimation	Yes	Yes	Yes
FIR Interpolation	Yes	Yes	Yes
CIC Decimation	Yes	N/A	N/A
CIC Interpolation	Yes	N/A	N/A
Biquad Filter	Yes	N/A	N/A

AddPipelineRegisters Details

The following table summarizes how enabling AddPipelineRegisters causes the different filter implementations to place pipeline registers, and the resultant latency.

Filter Block	Pipeline Register Placement	Latency (clock cycles)
Digital Filter (FIR, Asymmetric FIR, and Symmetric FIR filters)	A pipeline register is added between levels of a tree-based adder.	Where FL is the filter length: $\text{ceil}(\log_2(\text{FL}))$
Digital Filter (FIR Transposed)	A pipeline register is added after the products.	1
Digital Filter (IIR SOS)	Pipeline registers are added between the filter sections.	Where NS is number of sections: NS - 1

Filter Block	Pipeline Register Placement	Latency (clock cycles)
FIR Decimation	One pipeline register is added between levels of a tree-based adder, and one is added after the products.	Where NZ is the number of non-zero coefficients: $\text{ceil}(\log_2(NZ))$
FIR Interpolation	A pipeline register is added between levels of a tree-based adder.	Where PL is polyphase filter length: $\text{ceil}(\log_2(PL)) - 1$
CIC Decimation	A pipeline register is added between the comb stages of the differentiators .	Where NS is number of sections (at the input side): NS - 1
CIC Interpolation	A pipeline register is added between the comb stages of the differentiators.	Where NS is number of sections NS
Biquad Filter	Pipeline registers are added between the filter sections.	Where NS is number of sections: NS - 1

Limitations

Take note of the following limitations when applying `AddPipelineRegisters`, `MultiplierInputPipeline`, and `MultiplierOutputPipeline`:

- For FIR Filters, the coder places pipeline stages in the adder tree structure. In cases where the filter datapath is not full precision, this causes numeric differences between the original model and the generated model. To avoid such discrepancies, the coder modifies the filter block parameters in the generated model to full precision.
- Pipeline stages inserted by `AddPipelineRegisters`, `MultiplierInputPipeline`, and `MultiplierOutputPipeline` introduce delays along the path in the model that contains the affected filter. However, equivalent delays are not introduced on other, parallel

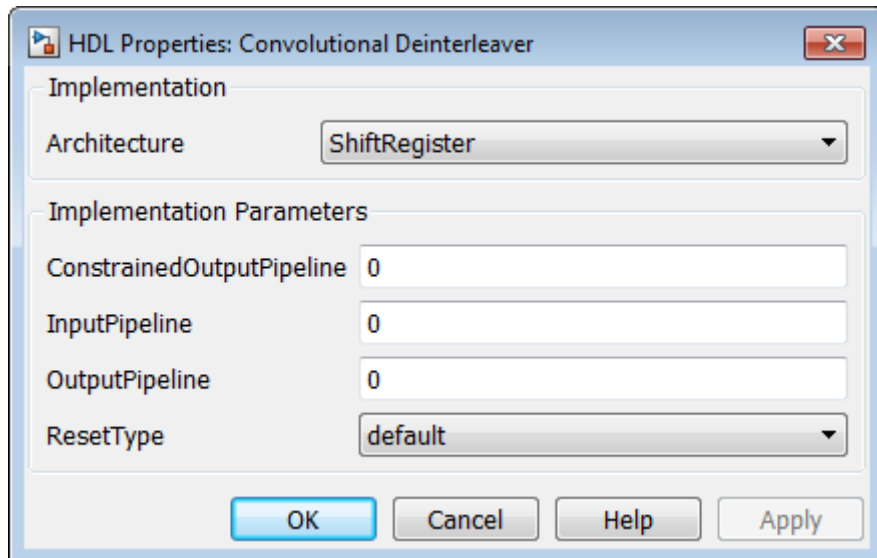
signal paths. To balance delays, use OutputPipeline on parallel data paths.

RAM

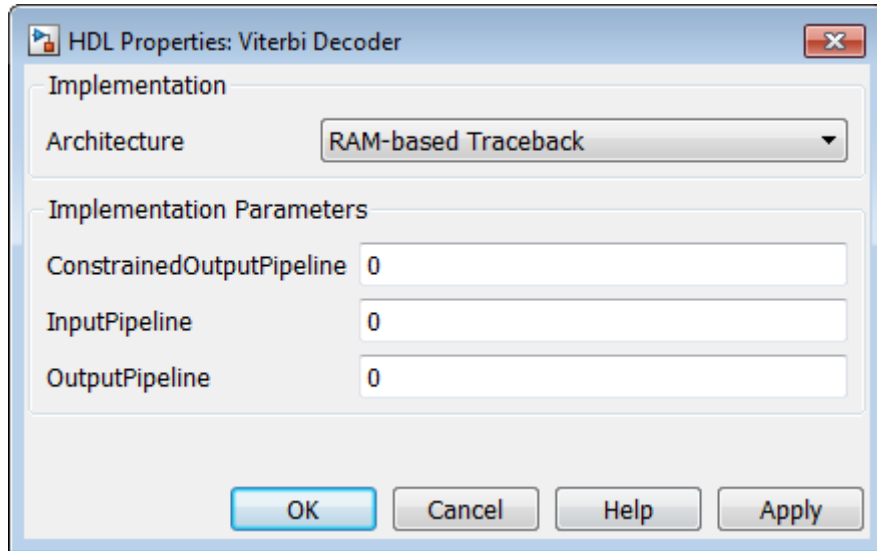
The following blocks support RAM based implementations as an alternative to shift register based implementations.

- commcnvintrlv2/Convolutional Deinterleaver
- commcnvintrlv2/Convolutional Interleaver
- commcnvcod2/Viterbi Decoder

The following figure shows the RAM and shift register options in the HDL Properties dialog box for a Convolutional Deinterleaver.



The following figure shows the RAM-based traceback option in the HDL Properties dialog box for a Viterbi Decoder.



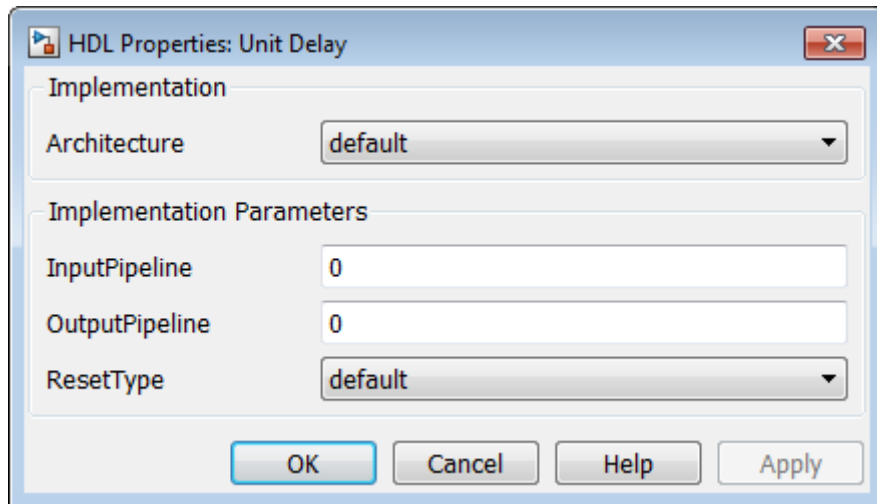
ResetType

The ResetType implementation parameter lets you suppress generation of reset logic for the following block types:

- commcnvtrlv2/Convolutional Deinterleaver
- commcnvtrlv2/Convolutional Interleaver
- commcnvtrlv2/General Multiplexed Deinterleaver
- commcnvtrlv2/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

The following figure shows the reset type option in the HDL Properties dialog box for a Unit Delay block.



When you specify `ResetType` as 'default', the coder follows the Global Settings/Advanced **Reset type** option for the specified blocks.

When you specify `ResetType` as 'none' for a selection of one or more blocks, the coder overrides the Global Settings/Advanced **Reset type** option for the specified blocks only. Reset signals and synchronous or asynchronous reset logic (as specified by **Reset type**) is still generated as required for other blocks.

Note that when you set `ResetType` to 'none', reset is not applied to generated registers. 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 spurious test bench errors, determine the number of samples required to fully load the registers. Then, set the **Ignore output data checking (number of samples)** option accordingly. (You can use

the `IgnoreDataChecking` property for this purpose, if you are using the command-line interface.) See also `IgnoreDataChecking`.

The following code specifies suppression of reset logic for a specific unit delay block within a subsystem.

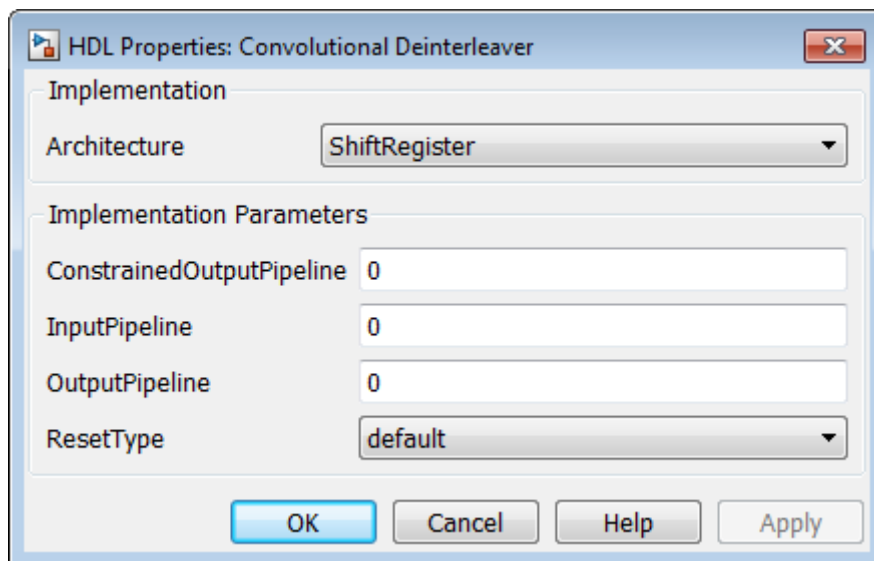
```
hdlset_param('rst_exam/ADut/UnitDelay1','ResetType','none');
```

ShiftRegister

The following blocks support shift register based implementations. (See “Convolutional Interleaver and Deinterleaver Block Requirements and Restrictions” on page 11-34.)

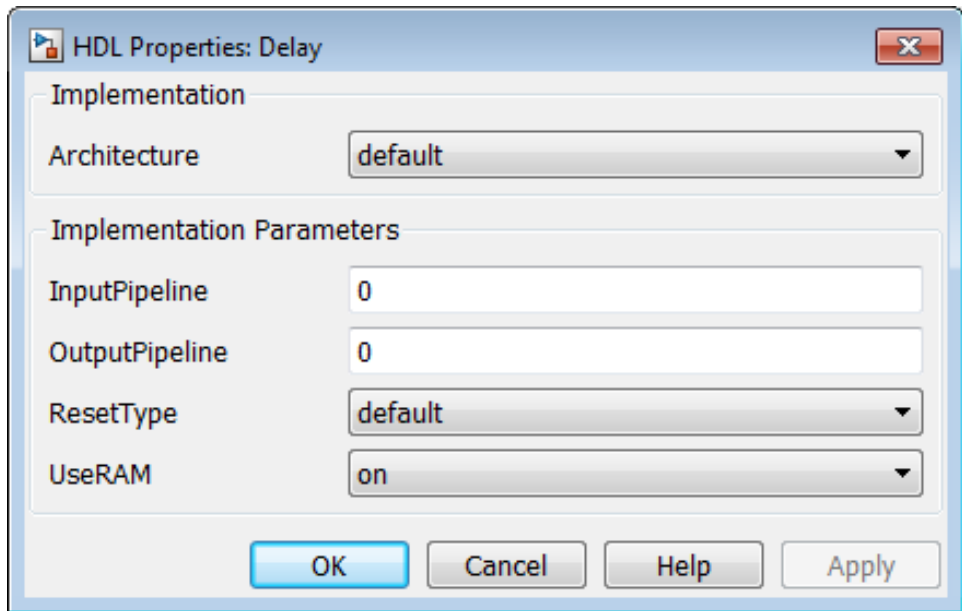
- `commcnvintrlv2/Convolutional Deinterleaver`
- `commcnvintrlv2/Convolutional Interleaver`
- `commcnvintrlv2/General Multiplexed Deinterleaver`
- `commcnvintrlv2/General Multiplexed Interleaver`

The following figure shows the shift register option in the HDL Properties dialog box for a Convolutional Deinterleaver .



UseRAM

The UseRAM implementation parameter enables using RAM-based mapping for a block instead of mapping to a shift register. 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 of a Single Delay to a RAM

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 11-94.
on	The delay maps to a dual-port RAM block when all of the following conditions are true: <ul style="list-style-type: none"> • Initial value of the delay is zero. • Delay length > 4.

UseRAM Setting	Mapping Behavior
	<ul style="list-style-type: none"> • Delay has one of the following set of numeric and data type attributes: <ul style="list-style-type: none"> ▪ (a) Real scalar with a non-floating-point data type (such as signed integer, unsigned integer, fixed point, or Boolean) ▪ (b) Complex scalar with real and imaginary parts that use non-floating-point data type ▪ (c) Vector where each element is either (a) or (b) • RAMSize is greater than or equal to the RAMMappingThreshold value. RAMSize is the product DelayLength * WordLength * ComplexLength. <ul style="list-style-type: none"> ▪ DelayLength is the number of delays that the Delay block specifies. ▪ WordLength is the number of bits that represent the data type of the delay. ▪ ComplexLength is 2 for complex signals; 1 otherwise. <p>If any condition is false, the delay maps to a shift register in the HDL code unless it merges with other delays to map to a single RAM. For more information, see “Mapping of Multiple Delays to a RAM” on page 11-92.</p>

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” on page 9-22.

Mapping of Multiple Delays to a RAM

The coder can also merge several delays of equal length into one delay and then map the merged delay to a single RAM. This optimization provides the following benefits:

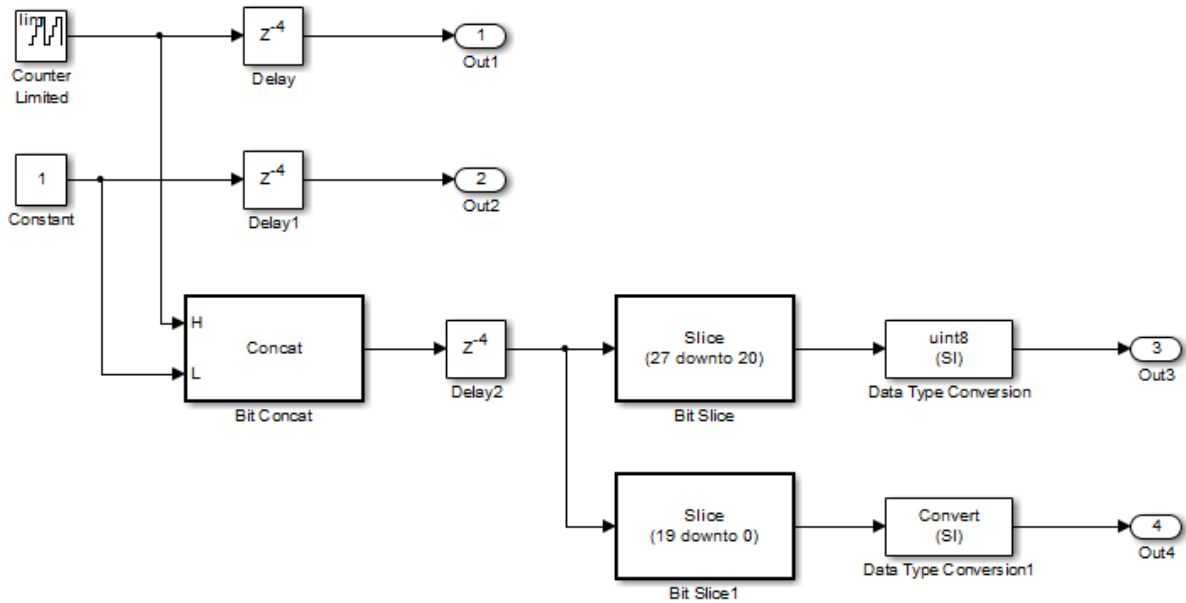
- Increased occupancy on a single RAM
- Sharing of address generation logic, which minimizes duplication of identical HDL code
- Mapping of delays to a RAM when the *individual* delays do not satisfy the threshold

The following rules control whether or not multiple delays can merge into one delay:

- The delays must:
 - Be at the same level of the subsystem hierarchy.
 - Use the same compiled sample time.
 - Have UseRAM set to on, or be generated by streaming or resource sharing.
 - Have the same ResetType setting, which cannot be none.
- The total word length of the merged delay cannot exceed 128 bits.
- The RAMSize of the merged delay is greater than or equal to the RAMMappingThreshold value. RAMSize is the product $\text{DelayLength} * \text{WordLength} * \text{VectorLength} * \text{ComplexLength}$.
 - DelayLength is the total number of delays.
 - WordLength is the number of bits that represent the data type of the merged delay.
 - VectorLength is the number of elements in a vector delay. VectorLength is 1 for a scalar delay.
 - ComplexLength is 2 for complex delays; 1 otherwise.

Example of Multiple Delays Mapping to a 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 UserRAM 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, the 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

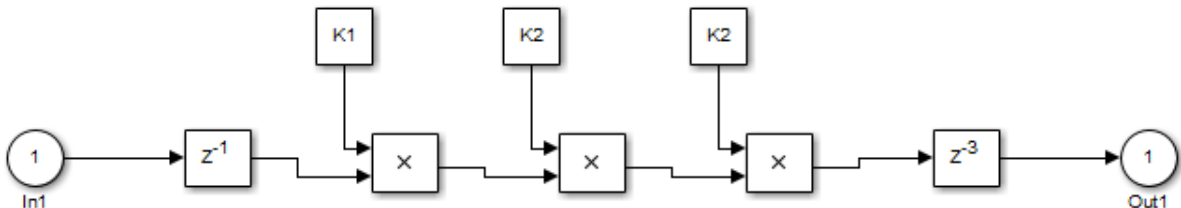
When UseRAM is off for a Delay block, the coder maps the delay to a shift register by default. However, the coder changes the UseRAM setting to on and tries to map the delay to a RAM under the following conditions:

- 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. The 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 the 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

Speed vs. Area Optimizations for FIR Filter Implementations

Overview of Speed vs. Area Optimizations

The coder provides options that extend your control over speed vs. area tradeoffs in the realization of FIR filter designs. To achieve the desired tradeoff, you can either specify a *fully parallel* architecture for generated HDL filter code, or choose one of several *serial* architectures. “Parallel and Serial Architectures” on page 11-96 describes the supported architectures.

The following blocks support these architecture options:

- dsparch4/Digital Filter
- dspmlti4/FIR Decimation
- dspmlti4/FIR.Interpolation
- simulink/Discrete/Discrete FIR Filter

You can specify the full range of parallel and serial architecture options using implementation parameters, as described in “Implementation Parameters for Specifying Speed vs. Area Tradeoffs” on page 11-97

Parallel and Serial Architectures

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 coder provides a range of serial architecture options, summarized below. These architectures have a latency of one clock period (see “Latency in Serial Architectures” on page 11-97).

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 (see “Implementation Parameters for Specifying Speed vs. Area Tradeoffs” on page 11-97. If you do not specify the serial partitions, the 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, the coder inserts a Delay block into the generated model after the filter block.

Implementation Parameters for Specifying Speed vs. Area Tradeoffs

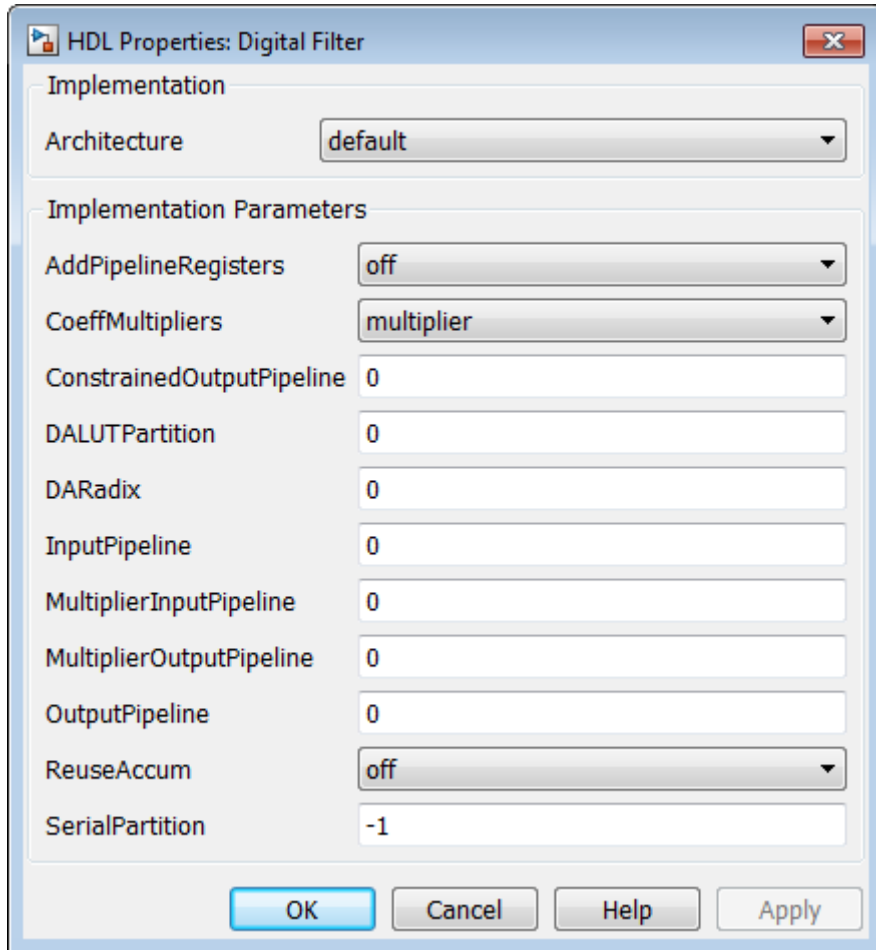
By default, `makehdl` generates filter code using a fully parallel architecture. If you want to generate FIR filter code with a fully parallel architecture, you do not need to specify this explicitly.

Two implementation parameters specify serial architecture options when generating code via `makehdl`:

- `'SerialPartition'`: This parameter specifies the serial partitioning of the filter.

- 'ReuseAccum': This parameter enables or disables accumulator reuse.

The following figure shows these parameters (at default values) on the HDL Properties dialog box for a Digital Filter block.

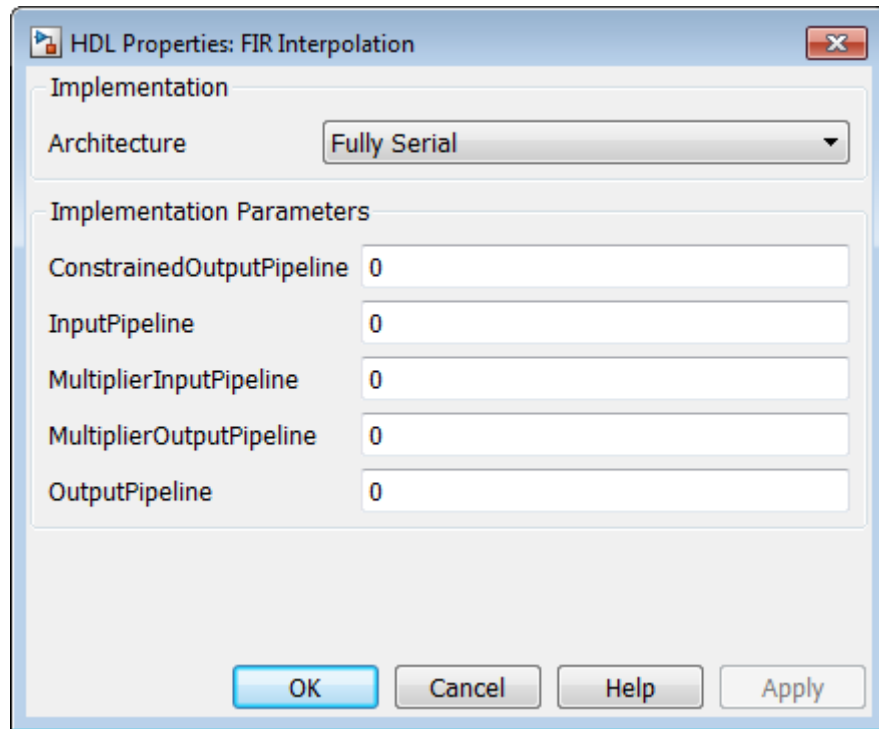


The table below summarizes how to set these parameters to generate the desired architecture.

To Generate This Architecture...	Set SerialPartition to...	Set ReuseAccum to...
Fully parallel	Omit this property	Omit this property
Fully serial	N, where N is the length of the filter	Not specified, or 'off'
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. 	'off'

To Generate This Architecture...	Set SerialPartition to...	Set ReuseAccum to...
Cascade-serial with explicitly specified partitioning	[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. The values of the vector elements must be in descending order, except that the last two element must be equal. For example, for a filter of length 9, partitions such as [5 4] or [4 3 2] would be legal, but the partitions [3 3 3] or [3 2 4] would raise an error at code generation time.	'on'
Cascade-serial with automatically optimized partitioning	Omit this property	'on'

FIR Interpolation Block Exception. The SerialPartition property is set automatically for you on the FIR Interpolation Block when you select Fully Serial architecture.



Filter Block Settings and Limitations. When you specify `SerialPartition` and `ReuseAccum` for a Digital Filter block, observe the following constraints.

- If you specify **Dialog parameters** as the `Coefficient` source:
 - Set **Transfer function type** to FIR (all zeros).
 - Select **Filter structure** as one of: Direct form,, Direct form symmetric, or Direct form asymmetric.
- If you specify **Discrete-time filter object** as the `Coefficient` source, the filter object must be one of the following:
 - `dfilt.dffir`
 - `dfilt.dfsymfir`
 - `dfilt.dfasymfir`

When you specify `SerialPartition` and `ReuseAccum` for a Discrete FIR Filter block, select **Filter structure** as one of the following:

- Direct form
- Direct form symmetric
- Direct form asymmetric

Observe the following limitations for FIR Decimation filters:

- The coder supports `SerialPartition` only for the FIR Direct Form structure.
- Accumulator reuse is not supported.

The coder supports serial partitioning for filter blocks only if all settings of the filter block are in full precision.

Use Full Precision Filter Settings. The coder supports serial partitioning for filter blocks only if all settings of the filter block are in full precision.

Interface Generation Parameters

Some block implementation parameters let you customize features of an interface generated for the following block types:

- `simulink/Ports & Subsystems/Model`
- `built-in/Subsystem`
- `lfilelib/HDL Cosimulation`
- `modelsimlib/HDL Cosimulation`

For example, you can specify generation of a black box interface for a subsystem, and pass parameters that specify the generation and naming of clock, reset, and other ports in HDL code. For more information about interface generation parameters, see “Customize the Generated Interface” on page 18-63.

Blocks That Support Complex Data

You can use complex signals in the test bench without restriction.

In the device under test (DUT) selected for HDL code generation, support for complex signals is limited to a subset of the blocks that the coder supports. These blocks appear in the following table. Some restrictions apply for some of these blocks.

Note All blocks listed support the `InputPipeline` and `OutputPipeline` implementation parameters.

Complex data expands 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).

Simulink Block	Restrictions
dspadpt3/LMS Filter	
dspindex/Variable Selector	
dsparch4/Biquad Filter	See “Complex Coefficients and Data Support for the Digital Filter and Biquad Filter Blocks” on page 11-107
dsparch4/Digital Filter	See “Complex Coefficients and Data Support for the Digital Filter and Biquad Filter Blocks” on page 11-107
dspindex/Multiport Selector	

Simulink Block	Restrictions
dspsigattribs/Convert 1-D to 2-D	
dspsigattribs/Frame Conversion	
dspsigops/Delay	
dspsigops/Downsample	
dspsigops/NCO	<hr/> <p>Note HDL support for the NCO block will be removed in a future release. Use the NCO HDL Optimized block instead.</p> <hr/>
dspsigops/Upsample	
dspsrcs4/DSP Constant	
dspsrcs4/Sine Wave	
hlddemolib/Dual Port RAM	
hlddemolib/Simple Dual Port RAM	
hlddemolib/Single Port RAM	
hlddemolib/HDL FFT	
hlddemolib/HDL Streaming FFT	
sflib/Chart	
simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Enabled	
simulink/Commonly Used Blocks/Constant	
simulink/Commonly Used Blocks/Data Type Conversion	
simulink/Commonly Used Blocks/Demux	

Simulink Block	Restrictions
simulink/Commonly Used Blocks/Gain	
simulink/Commonly Used Blocks/Ground	
simulink/Commonly Used Blocks/Product	
simulink/Commonly Used Blocks/Sum	
simulink/Commonly Used Blocks/Mux	
simulink/Commonly Used Blocks/Relational Operator	~= and == operators only
simulink/Commonly Used Blocks/Switch	
simulink/Commonly Used Blocks/Unit Delay	
simulink/Discrete/Delay	
simulink/Discrete/Memory	
simulink/Discrete/Zero-Order Hold	
simulink/Discrete/Tapped Delay	
simulink/Logic and Bit Operations/Compare To Constant	
simulink/Logic and Bit Operations/Compare To Zero	
simulink/Logic and Bit Operations/Shift Arithmetic	
simulink/Lookup Tables/1-D Lookup Table	
simulink/Math Operations/Add	

Simulink Block	Restrictions
simulink/Math Operations/Assignment	
simulink/Math Operations/Complex to Real-Imag	
simulink/Math Operations/Unary Minus	
simulink/Math Operations/Math Function	The conj, hermitian, and transpose functions support complex data.
simulink/Math Operations/Matrix Concatenate	
simulink/Math Operations/Product of Elements	Only the default (linear) implementation supports complex data. Complex division is not supported.
simulink/Math Operations/Real-Imag to Complex	
simulink/Math Operations/Reshape	
simulink/Math Operations/Subtract	Only the default (linear) implementation supports complex data.
simulink/Math Operations/Sum of Elements	Only the default (linear) implementation supports complex data.
simulink/Math Operations/Vector Concatenate	
simulink/Signal Attributes/Rate Transition	
simulink/Signal Attributes/Signal Conversion	

Simulink Block	Restrictions
simulink/Signal Attributes/Signal Specification	
simulink/Signal Routing/Index Vector	
simulink/Signal Routing/Multiport Switch	
simulink/Signal Routing/Selector	
simulink/User-Defined Functions/MATLAB Function	See also “Complex Data Type Support” on page 1-12.

Complex Coefficients and Data Support for the Digital Filter and Biquad Filter Blocks

The coder supports use of complex coefficients and complex input signals for all filter structures of the Digital Filter and Biquad Filter blocks, except decimators and interpolators. In many cases, you can use complex data and complex coefficients in combination. The following table shows the filter structures that support complex data and/or coefficients, and the permitted combinations.

Filter Structure	Complex Data	Complex Coefficients	Complex Data and Coefficients
dfilt.dffir	Y	Y	Y
dfilt.dfsymfir	Y	Y	Y
dfilt.dfasymfir	Y	Y	Y
dfilt.dffirt	Y	Y	Y
dfilt.scalar	Y	Y	Y
dfilt.delay	Y	N/A	N/A
mfilt.cicdecim	Y	N/A	N/A
mfilt.cicinterp	Y	N/A	N/A
mfilt.firdecim	Y	Y	N

Filter Structure	Complex Data	Complex Coefficients	Complex Data and Coefficients
<code>mfilt.firinterp</code>	Y	Y	N
<code>dfilt.df1sos</code>	Y	Y	Y
<code>dfilt.df1tsos</code>	Y	Y	Y
<code>dfilt.df2sos</code>	Y	Y	Y
<code>dfilt.df2tsos</code>	Y	Y	Y

Blocks That Support Buses

In this section...

“Supported Bus Blocks” on page 11-109
“Settings and Requirements” on page 11-109
“Limitations” on page 11-112
“See Also” on page 11-113

Supported Bus Blocks

The following bus-capable blocks support HDL code generation:

- Bus Creator
- Bus Selector

If you want to use buses in your design *and* you want to generate HDL code, then you must use these blocks for the buses in your design. No other bus-capable blocks support HDL code generation.

Settings and Requirements

Although the Bus Creator and Bus Selector blocks do not have new GUI elements for code generation, there are some conditions you must meet for HDL code generation.

Requirements

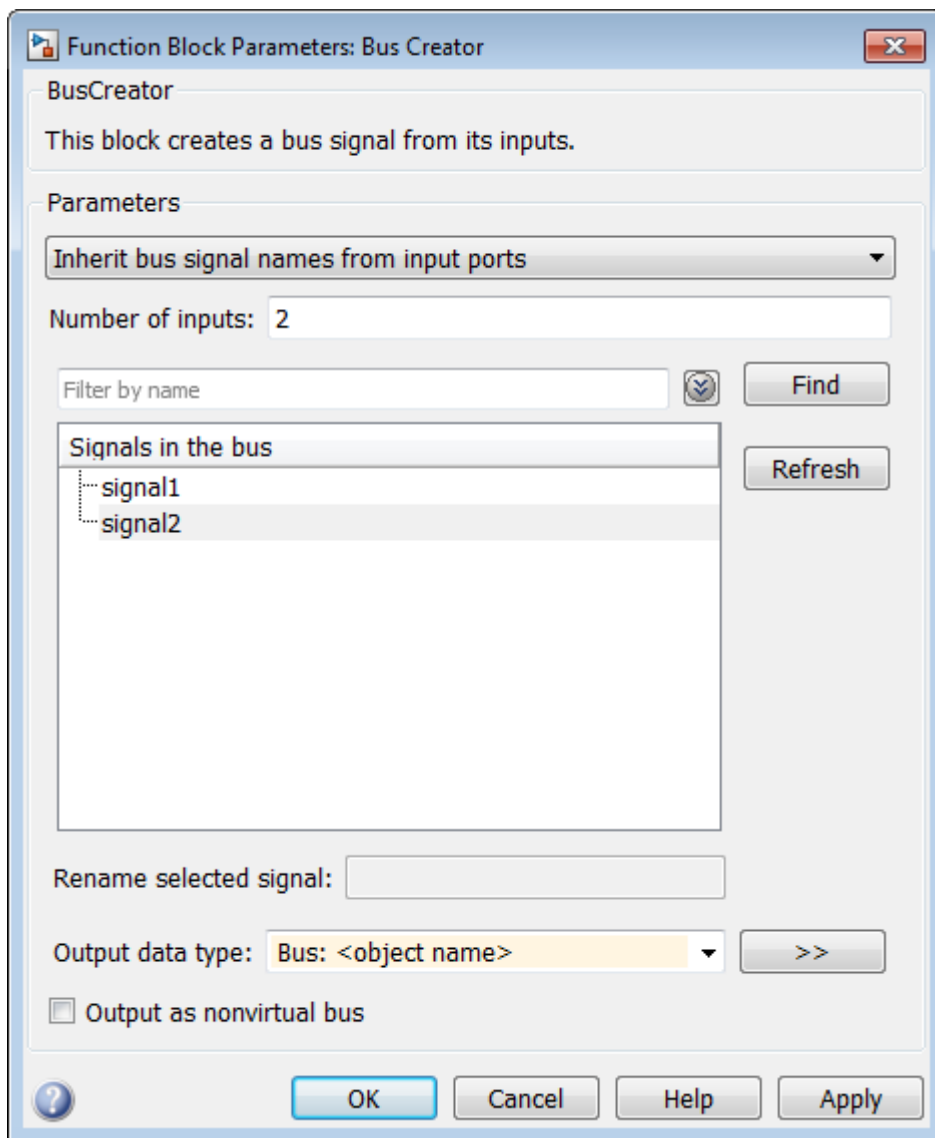
Make sure the buses at the level you want to generate code from are connected to either a Bus Creator or Bus Selector block.

Settings

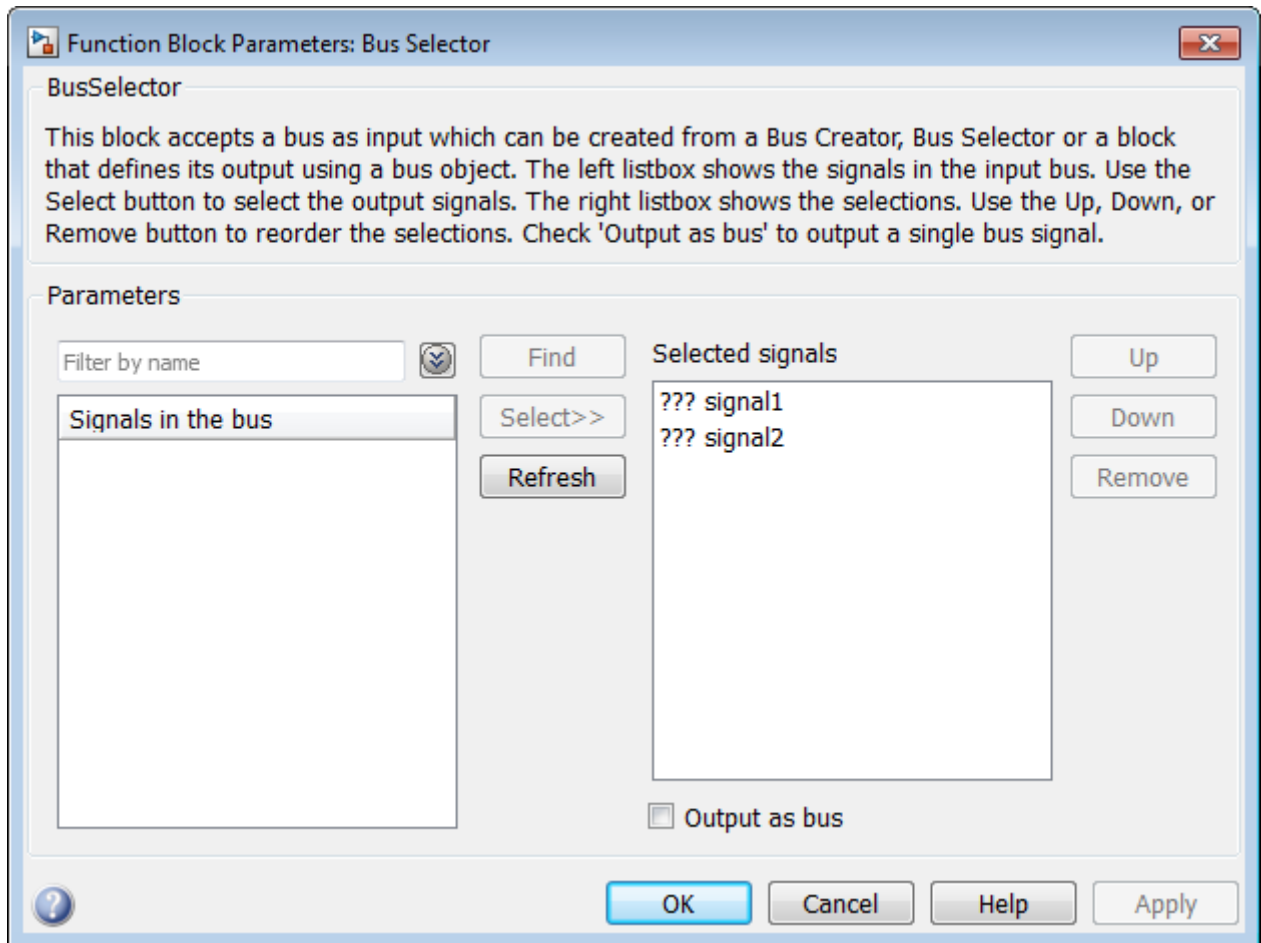
- Perform *one* of the following tasks to enable HDL code generation:
 - Run function `hdlsetup` at the MATLAB command prompt.
 - Set the **Simulation > Configuration Parameters > Diagnostics > Connectivity**→**Mux blocks used to**

create bus signals parameter to error. For details, see “Prevent Bus and Mux Mixtures”.

- In the Bus Creator dialog:
 - Make sure **Output as nonvirtual bus** is *not* checked.
 - Make sure Bus Creator output is a BusObject.



- In the Bus Selector dialog, make sure **Output as a bus** is *not* checked.



Limitations

HDL code generation for buses does not support:

- Buses that do not have BusObject data type
- Non-virtual bus
- Output as a bus (for Bus Selector)

- Other bus-capable blocks
- Bus input/output at top-level subsystem port

See Also

- Bus Creator
- Bus Selector

Lookup Table Block Support

The coder supports the following lookup table (LUT) blocks:

- simulink/Lookup Tables/n-D Lookup Table
- simulink/Lookup Tables/Prelookup
- simulink/Lookup Tables/Direct Lookup Table (n-D)
- simulink/Lookup Tables/1-D Lookup Table
- simulink/Lookup Tables/2-D Lookup Table

When you configure a lookup table block for HDL code generation, observe the requirements and limitations described in the following sections.

n-D Lookup Table

Required Block Settings

- **Number of table dimensions:** The coder supports a maximum dimension of 2.
- **Index search method:** Select Evenly spaced points.
- **Extrapolation method:** The coder supports only Clip. The coder does not support extrapolation beyond the table bounds.
- **Interpolation method:** The coder supports only Flat or Linear.
- **Diagnostic for out-of-range input:** Select Error. If you select other options, the coder displays a warning.
- **Use last table value for inputs at or above last breakpoint:** Select this check box.
- **Require all inputs to have the same data type:** Select this check box.
- **Fraction:** Select Inherit: Inherit via internal rule.
- **Integer rounding mode:** Select Zero, Floor, or Simplest.

Avoid Generation of Divide Operator

The coder gives a warning if it encounters conditions under which a division operation is required to match the model's simulation behavior. The conditions described in this section will cause the n-D Lookup Table block to emit a divide operator. When you use the n-D Lookup Table block for HDL code generation, you should avoid the following conditions:

- If the block is configured to use interpolation, a division operator will be required. To avoid this, set **Interpolation method** : to Flat.
- The second way depends on the table spacing. HDL code generation requires the block to use the "Evenly Spaced Points" algorithm. The block mapping from the input data type to the zero-based table index in general requires a division. When the breakpoint spacing is an exact power of 2, this divide is implemented as a shift instead of as a divide. To adjust the breakpoint spacing, you can adjust the number of breakpoints in the table and/or the difference between the left and right bounds of the breakpoint range.

Table Data Typing and Sizing

- It is good practice to structure your table such that the spacing between breakpoints is a power of two. The coder issues a warning if the breakpoint spacing does not meet this condition. When the breakpoint spacing is a power of two, you can replace division operations in the prelookup step with right-shift operations.
- Table data must resolve to a nonfloating-point data type.
- All ports on the block require scalar values.

Prelookup

Required Block Settings

- **Index search method**: Select Evenly spaced points.
- **Extrapolation method**: Select Clip.
- **Diagnostic for out-of-range input**: Select Error. If you select other options, the coder displays a warning.

- **Use last breakpoint for input at or above upper limit:** Select this check box.
- **Breakpoint data type:** Select Inherit: Same as input.
- **Integer rounding mode:** Select Zero, Floor, or Simplest.

Table Data Typing and Sizing

- It is good practice to structure your table such that the spacing between breakpoints is a power of two. The coder issues a warning if the breakpoint spacing does not meet this condition. When the breakpoint spacing is a power of two, you can replace division operations in the prelookup step with right-shift operations.
- All ports on the block require scalar values.
- The coder permits floating-point data for breakpoints.

Direct Lookup Table (n-D)

Required Block Settings

- **Number of table dimensions:** The coder supports a maximum dimension of 2.
- **Inputs select this object from table:** Select Element.
- **Make table an input:** Clear this check box.
- **Diagnostic for out-of-range input:** Select Error. If you select other options, the coder displays a warning.

Table Data Typing and Sizing

- It is good practice to size each dimension in the table to be a power of two. The coder issues a warning if the length of a dimension (*except* the innermost dimension) is not a power of two. By following this practice, you can avoid multiplications during table indexing operations and realize a more efficient table in hardware.

- Table data must resolve to a nonfloating-point data type. The coder examines the output port to verify that its data type meets this requirement.
- All ports on the block require scalar values.

1-D Lookup Table

The 1-D Lookup Table block is subject to the same limitations as the n-D Lookup Table block. See “n-D Lookup Table” on page 11-114 for detailed information.

2-D Lookup Table

The 2-D Lookup Table block is subject to the same limitations as the n-D Lookup Table block. See “n-D Lookup Table” on page 11-114 for detailed information.

Generating HDL Code for Multirate Models

- “Code Generation from Multirate Models” on page 12-2
- “Configure Multirate Models for HDL Code Generation” on page 12-3
- “Code Generation from a Multirate DUT” on page 12-6
- “Generate a Global Oversampling Clock” on page 12-9
- “Generate Multicycle Path Information Files” on page 12-16
- “HDL Properties for Controlling Multirate Code Generation” on page 12-27

Code Generation from Multirate Models

The 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.

A *timing controller* entity generates the required rates from a single master clock using one or more counters, creating 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

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, the coder appends the value of the `TimingControllerPostfix` property to the DUT name.

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.

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 “Configure Multirate Models for HDL Code Generation” on page 12-3.

Configure Multirate Models for HDL Code Generation

In this section...

“Overview” on page 12-3

“Configuring Model Parameters” on page 12-3

“Configuring Sample Rates in the Model” on page 12-4

“Block Configuration and Restrictions For Multirate DUTs” on page 12-4

Overview

Certain requirements and restrictions apply to multirate models that are intended for HDL code generation. This section provides guidelines on how to configure model and block parameters to meet these requirements.

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. Setting **Multitask rate transition** and **Single task rate transition** to error to detect illegal rate transitions before code is generated.

Configuring Sample Rates in the Model

The 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 12-4
- “Rate Transition” on page 12-4
- “Upsample” on page 12-5
- “Downsample” on page 12-5
- “Delay and Zero-Order Hold” on page 12-5

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:

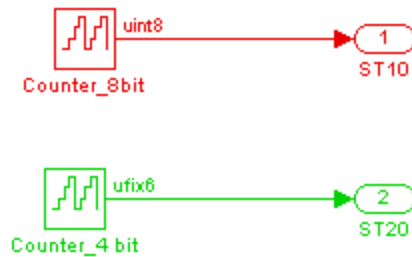
- Delay
- Tapped Delay
- Unit Delay
- Unit Delay Enabled
- Zero-Order Hold

The Delay blocks listed should be configured to have the same input and output sample rates.

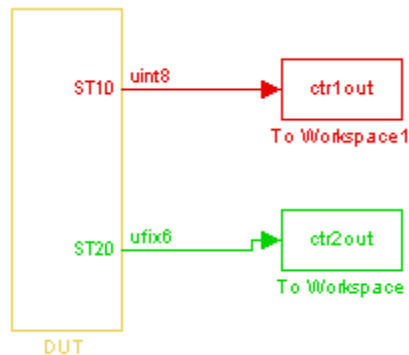
Zero-Order Hold blocks must be configured with inherited (–1) sample times.

Code Generation from 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.

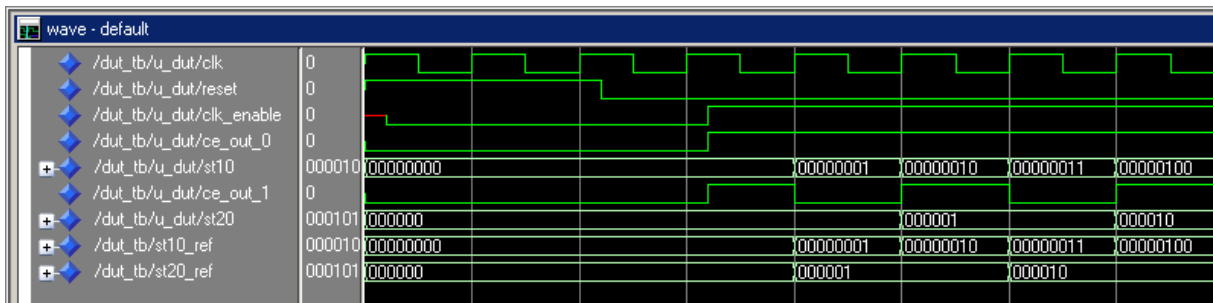
```

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.

Generate a Global Oversampling Clock

In this section...

“Why Use a Global Oversampling Clock?” on page 12-9

“Requirements for the Oversampling Factor” on page 12-9

“Specifying the Oversampling Factor From the GUI” on page 12-10

“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, the 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:

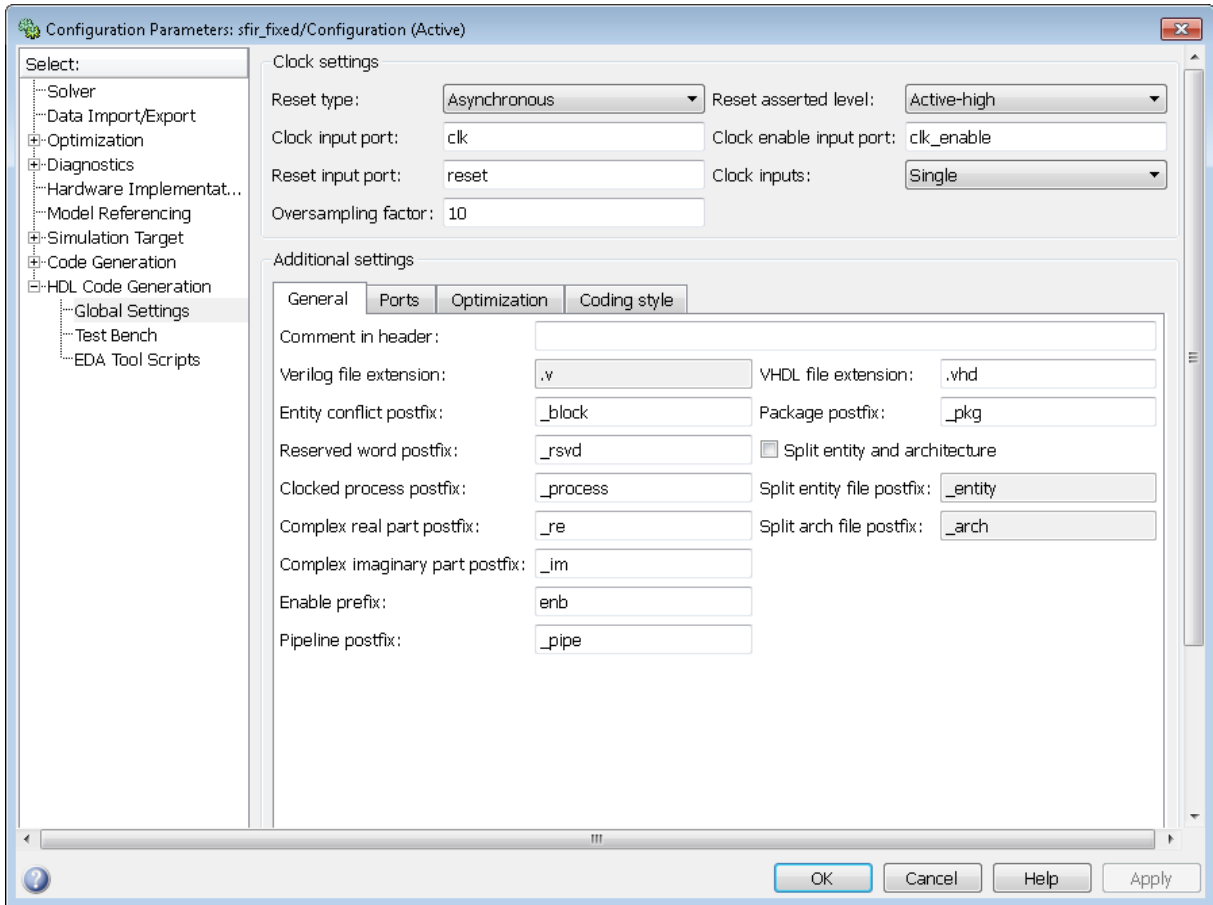
- The oversampling factor must be an integer greater than or equal to 1.
- The default value is 1. In the default case, the coder does not generate a global oversampling clock.

- 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.

The 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 as hdlsrc\symmetric_fir_tc.vhd
### Working on sfir_fixed/symmetric_fir as hdlsrc\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', N property in the makehdl command. The following example specifies an oversampling factor of 7:

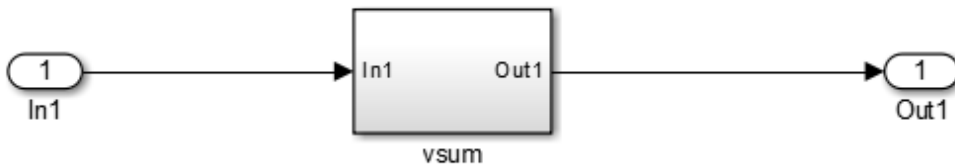
```
>> makehdl(gcb,'Oversampling', 7)
### Generating HDL for 'sfir_fixed/symmetric_fir'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### Begin VHDL Code Generation
### MESSAGE: The design requires 7 times faster clock with respect to the base rate = 1.
### Working on symmetric_fir_tc as hdlsrc\symmetric_fir_tc.vhd
### Working on sfir_fixed/symmetric_fir as hdlsrc\symmetric_fir.vhd
### HDL Code Generation Complete.
```

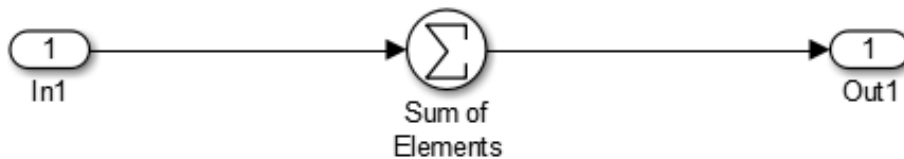
Resolving Oversampling Rate Conflicts

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 (also discussed in “View Latency Differences After Area Optimization” on page 14-9).

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. The 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.
...
```

Generate Multicycle Path Information Files

In this section...

“Overview” on page 12-16

“Format and Content of a Multicycle Path Information File” on page 12-17

“File Naming and Location Conventions” on page 12-23

“Generating Multicycle Path Information Files Using the GUI” on page 12-23

“Generating Multicycle Path Information Files Using the Command Line” on page 12-24

“Limitations” on page 12-24

“Example of Generating a Multicycle Path Information File” on page 12-26

Overview

The 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.boolireg; TO : Sbs.booloreg; PATH_MULT : 2; RELATIVE_CLK : source,
      Sbs.clk;
FROM : Sbs.boolireg_v<0>; TO : Sbs.booloreg_v<0>; PATH_MULT : 2;
      RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.doubireg; TO : Sbs.douboreg; PATH_MULT : 2; RELATIVE_CLK : source,
      Sbs.clk;
FROM : Sbs.doubireg_v<0>; TO : Sbs.douboreg_v<0>; PATH_MULT : 2;
      RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.intireg(7:0); TO : Sbs.intoreg(7:0); PATH_MULT : 2;
      RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.intireg_v<0>(7:0); TO : Sbs.intoreg_v<0>(7:0); PATH_MULT : 2
      RELATIVE_CLK : source, Sbs.clk;
```

The first section of the file is a header that identifies the source model and gives other information about how the 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 : src_reg_path;	The source (or FROM) register of a multicycle path in the system. The value of src_reg_path 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 : dst_reg_path;	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 dst_reg_path 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.
PATH_MULT : N;	<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 N indicates that the signal has N 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 N = 1, because this is the default path multiplier.</p>
RELATIVE_CLK : relclock, sysclock;	<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"> • relclock: Since the 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. • sysclock: 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.
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 intireg is defined in the module H1, and H1 is inside the module Sbs. u_H1 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.
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`.

The coder writes the multicycle path information file to the target .

Generating Multicycle Path Information Files Using the GUI

By default, the 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 as hdlsrc\Subsystem_tc.vhd
### Working on hdlfirtdecim_multicycle/Subsystem as hdlsrc\Subsystem.vhd
### Generating package file hdlsrc\Subsystem_pkg.vhd
### Finishing multicycle path connectivity analysis.
### Writing multicycle path information in hdlsrc\Subsystem_constraints.txt
### HDL Code Generation Complete.
```

Limitations

Unsupported Blocks and Implementations

The following table lists block implementations (and associated Simulink blocks) that will not contribute to multicycle path constraints information.

Implementation	Block(s)
SumCascadeHDLEmission	Add, Subtract, Sum, Sum of Elements
ProductCascadeHDLEmission	Product, Product of Elements
MinMaxCascadeHDLEmission	MinMax, Maximum, Minimum
ModelReferenceHDLInstantiation	Model
SubsystemBlackBoxHDLInstantiation	Subsystem

Implementation	Block(s)
RamBlockDualHDLInstantiation	Dual Port RAM
RamBlockSimpDualHDLInstantiation	Simple Dual Port RAM
RamBlockSingleHDLInstantiation	Single Port RAM

Limitations on MATLAB Function Blocks and Stateflow Charts

Loop-Carried Dependencies. The 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.

Example of Generating a Multicycle Path Information File

The “Getting Started with Multicycle Path Constraint Generation” example illustrates generation of a multicycle path information file using a model of a decimating filter. To open the example, enter the following at the command line:

```
showdemo hdlmulticyclepath
```


HDL Properties for Controlling Multirate Code Generation

In this section...

“Overview” on page 12-27

“HoldInputDataBetweenSamples” on page 12-27

“OptimizeTimingController” on page 12-27

Overview

This section summarizes coder properties that provide additional control over multirate code generation.

HoldInputDataBetweenSamples

This property determines how long (in terms of base rate clock cycles) data values for subrate signals are held in a valid state.

When 'on' (the default), data values for subrate signals are held in a valid state across each subrate sample period.

When 'off', data values for subrate signals are held in a valid state for only one base-rate clock cycle. See HoldInputDataBetweenSamples for details.

OptimizeTimingController

This property specifies whether the timing controller generates the required rates using multiple counters per rate (the default) or a single counter. The use of multiple counters optimizes generated code for speed and area. See OptimizeTimingController for details.

The hdl demolib Block Library

- “Open the hdl demolib Library” on page 13-2
- “RAM Blocks” on page 13-3
- “Generate RAM Without Clock Enable Ports” on page 13-13
- “Build a ROM Block with Simulink Blocks” on page 13-14
- “HDL Counter” on page 13-15
- “HDL FFT” on page 13-27
- “Signal Processing with the HDL FFT Block” on page 13-35
- “HDL FIFO” on page 13-36
- “HDL Streaming FFT” on page 13-40
- “Bitwise Operators” on page 13-50

Open the hdl demolib Library

The hdl demolib library provides HDL-specific implementations supporting simulation and code generation for:

- Single and dual-port RAMs
- Counter with single-shot and free-running modes
- Minimum resource FFT
- Operations on bits and bit fields
- FIFO (Queue)

These blocks are implemented as subsystems. The blocks provide HDL-specific functionality that is not currently supported by other Simulink blocks.

To open the hdl demolib library, type the following command at the MATLAB prompt:

```
hdl demolib
```

RAM Blocks

In this section...

- “Overview of RAM Blocks” on page 13-3
- “Dual Port RAM Block” on page 13-5
- “Simple Dual Port RAM Block” on page 13-7
- “Single Port RAM Block” on page 13-8
- “Code Generation for RAM Blocks” on page 13-11
- “Limitations for RAM Blocks” on page 13-12

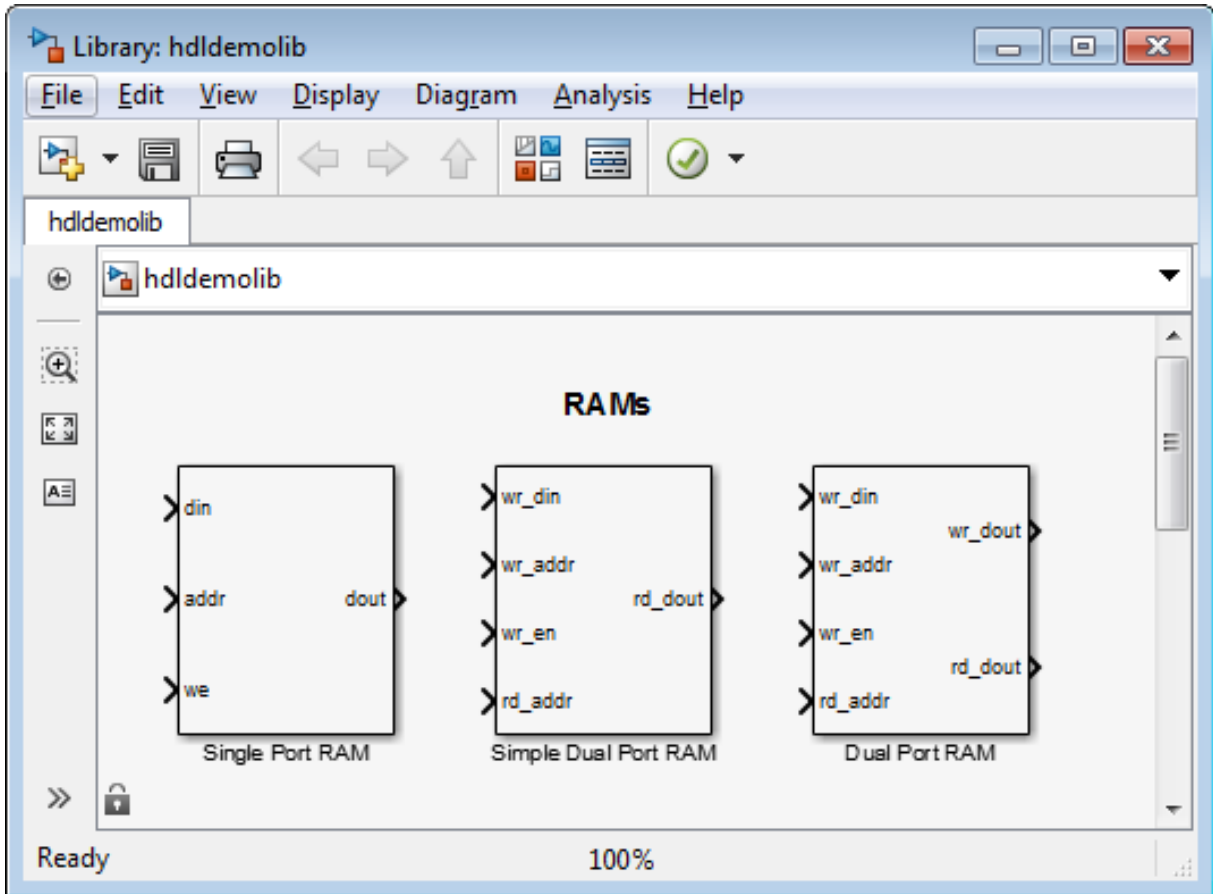
Overview of RAM Blocks

The RAM blocks let you:

- Simulate the behavior of a single-port or dual-port RAM in your model.
- Generate an interface to the inputs and outputs of the RAM in HDL code.
- Generate RTL code that can be inferred as a RAM by most synthesis tools, for most FPGAs.

The RAM blocks are grouped together in the `hdl1demolib` library, as shown in the following figure. The library provides three type of RAM blocks:

- Dual Port RAM
- Simple Dual Port RAM
- Single Port RAM



To open the library, type the following command at the MATLAB prompt:

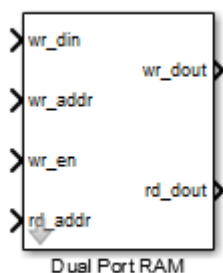
```
hd1demo1ib
```

Then, drag the desired RAM block from the `hd1demo1ib` library to your model, and set the block parameters and connect signals following the guidelines in the following sections.

Dual Port RAM Block

Dual Port RAM Block Ports and Parameters

The following figure shows the Dual Port RAM block.



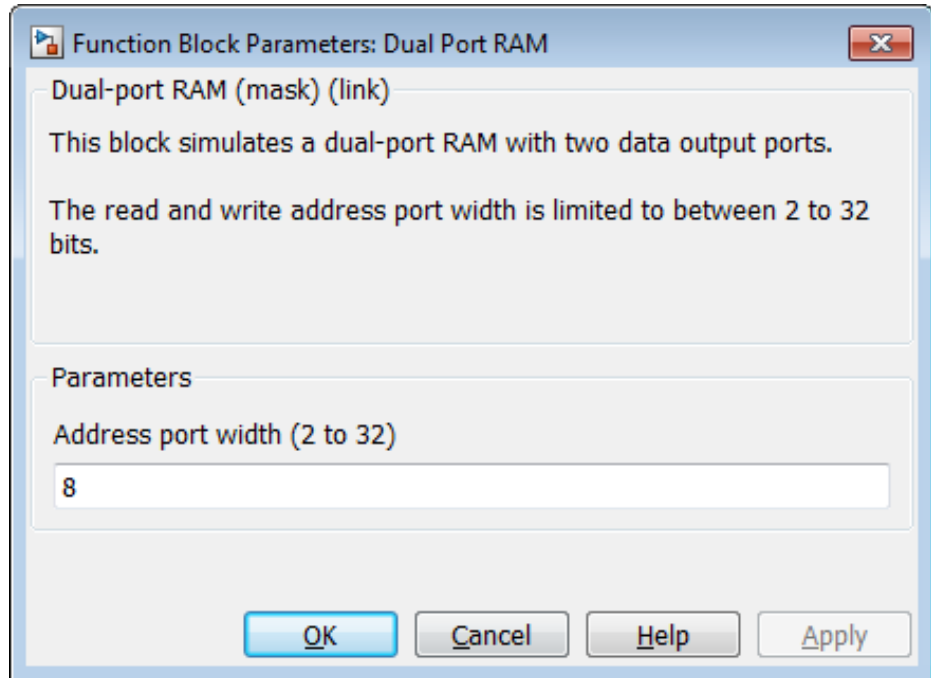
The block has the following input and output ports:

- `wr_din`: Data input. Only scalar signals can be connected to this port. The data type of the input signal can be fixed point, integer, or complex, and can be of any width. The port inherits the width and data type of its input signal.
- `wr_addr`, `rd_addr`: Write and read address ports, respectively.

To set the width of the address ports, enter the desired width value (minimum width 2 bits, maximum width 32 bits) into the **Address port width** field of the block GUI, as shown in the following figure. The default width is 8 bits.

The data type of signals connected to these ports must be unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0.

Vector signals are not accepted at the address ports.



- `wr_en`: Write enable. This port must be connected to a Boolean signal.
- `wr_dout`, `rd_dout`: Output ports with read data for addresses `wr_addr` and `rd_addr`, respectively.

Tip If data output at the write port is not required, you can achieve better RAM inference with synthesis tools by using the Simple Dual Port RAM block rather than the Dual Port RAM block.

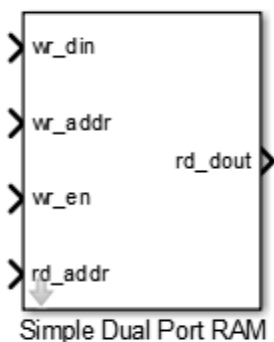
Read-During-Write Behavior

During a write, new data appears at the output of the write port (`wr_dout`) of the Dual Port RAM block. If a read operation is performed at the same address at the read port, old data is read at the output (`rd_dout`).

Simple Dual Port RAM Block

Simple Dual Port RAM Block Ports and Parameters

The following figure shows the Simple Dual Port RAM block.



This block is similar to the Dual Port RAM. It differs from Dual Port RAM in its read-during-write behavior, and it does not have the data output at the write port (`wr_dout`).

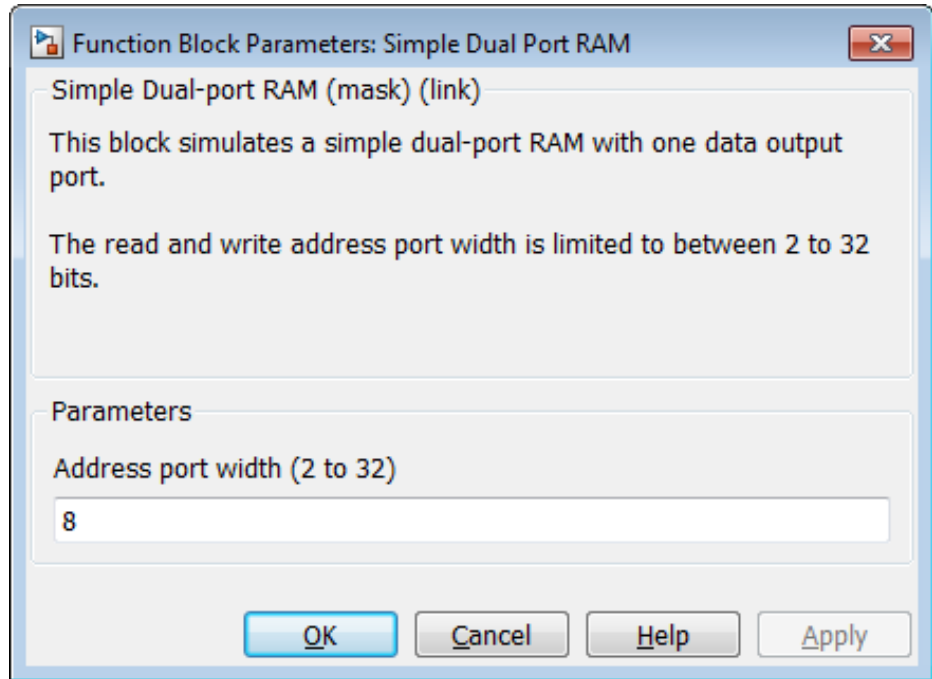
The block has the following input and output ports:

- `wr_din`: Data input. Only scalar signals can be connected to this port. The data type of the input signal can be fixed point, integer, or complex, and can be of any width. The port inherits the width and data type of its input signal.
- `wr_addr`, `rd_addr`: Write and read address ports, respectively.

To set the width of the address ports, enter the desired width value (minimum width 2 bits, maximum width 32 bits) into the **Address port width** field of the block GUI, as shown in the following figure. The default width is 8 bits.

The data type of signals connected to these ports must be unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0.

Vector signals are not accepted at the address ports.



- `wr_en`: Write enable. This port must be connected to a Boolean signal.
- `rd_dout`: Output port with read data for addresses `wr_addr` and `rd_addr`, respectively.

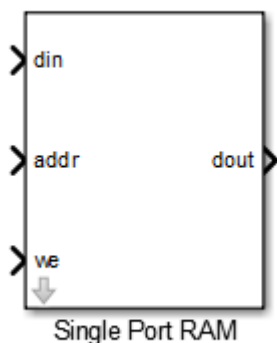
Read-During-Write Behavior

During a write operation, if a read operation is performed at the same address at the read port, old data is read at the output.

Single Port RAM Block

Single Port RAM Block Ports and Parameters

The following figure shows the Single Port RAM block.



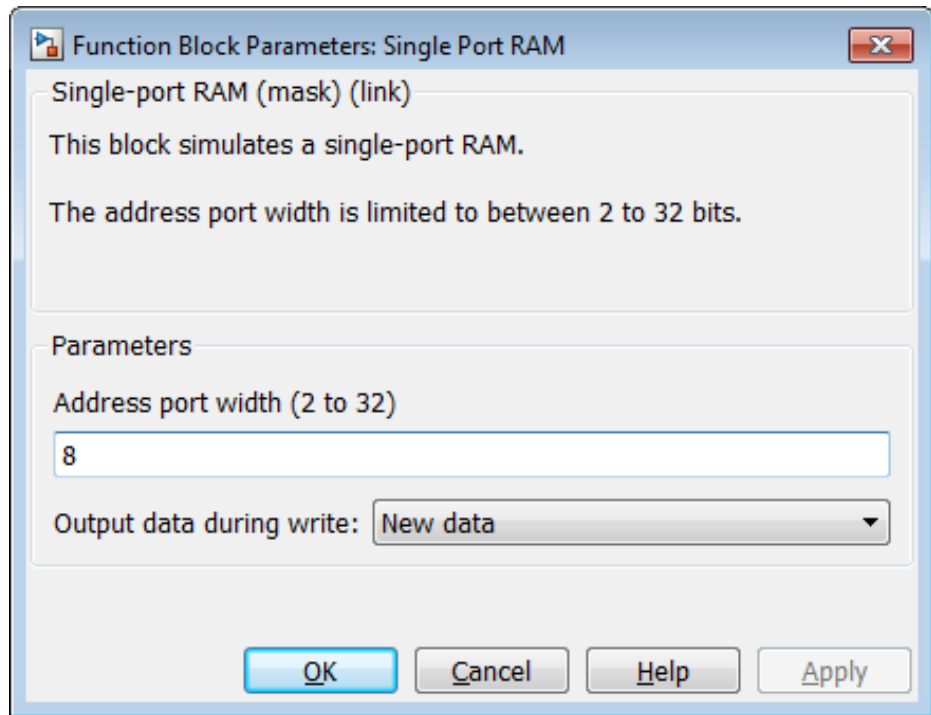
The block has the following input and output ports:

- **din** : Data input. Only scalar signals can be connected to this port. The data type of the input signal can be fixed point, integer, or complex, and can be of any width. The port inherits the width and data type of its input signal.
- **addr**: Write address port.

To set the width of the address ports, enter the desired width value (minimum width 2 bits, maximum width 32 bits) into the **Address port width** field of the block GUI, as shown in the following figure. The default width is 8 bits.

The data type of signals connected to these ports must be unsigned integer (**uintN**) or unsigned fixed point (**ufixN**) with a fraction length of 0.

Vector signals are not accepted at the address ports.



- we: Write enable. This port must be connected to a Boolean signal.
- dout: Output port with data for address addr.

Read-During-Write Behavior

The **Output data during write** dropdown menu provides options that control how the RAM handles output/read data. These options are:

- New data (default): During a write, new data appears at the output port (dout).
- Old data: During a write, old data appears at the output port (dout).

Note Depending on your synthesis tool and target device, the setting of **Output data during write** may affect the result of RAM inference. See “Limitations for RAM Blocks” on page 13-12 for further information on read-during-write behavior in hardware.

Code Generation for RAM Blocks

Code generation for a RAM block creates a separate file, *blockname.ext*, where *blockname* is derived from the name of the RAM block, and *ext* is the target language filename extension.

Code generated for RAM blocks has:

- A latency of 1 clock cycle for read data output.
- No reset signal, since some synthesis tools do not infer a RAM from HDL code if it includes a reset.

Most synthesis tools infer a RAM from the generated HDL code with default settings. However, your synthesis tool may not map the generated code to RAM for the following reasons:

- Small RAM size: your synthesis tool may use registers to implement a small RAM for better performance.
- A clock enable signal is present. You can suppress generation of a clock enable signal in RAM blocks, as described in “Implement RAMs With or Without Clock Enable” on page 13-11.

Please note that code generated to initialize a RAM is intended for simulation only, and may be ignored by synthesis tools.

Implement RAMs With or Without Clock Enable

The `RAMArchitecture` property enables or suppresses generation of clock enable logic for all RAM blocks in a subsystem. You can set `RAMArchitecture` to the following values:

- `'WithClockEnable'` (default): Generates RAMs using HDL templates that include a clock enable signal, and an empty RAM wrapper.

- 'WithoutClockEnable': Generates RAMs without clock enables, and a RAM wrapper that implements the clock enable logic.

Some synthesis tools may not infer RAMs with a clock enable. Set `GlobalRAMArchitecture` to 'WithoutClockEnable' if your synthesis tool does not support RAM structures with a clock enable, and cannot map your generated HDL code to FPGA RAM resources. To learn how to generate RAMs without clock enables for your design, see the Getting Started with RAM and ROM example. To open the example, type the following command at the MATLAB prompt:

```
hdlcoderramrom
```

Limitations for RAM Blocks

The following limitations apply to the use of RAM blocks in HDL code generation:

- If you use RAM blocks to perform concurrent read and write operations, you should manually verify the read-during-write behavior in hardware. The read-during-write behavior of the RAM blocks in Simulink matches that of the generated behavioral HDL code. However, a synthesis tool may not follow the same behavior during RAM inferring, causing the read-during-write behavior in hardware to differ from the behavior of the Simulink model or generated HDL code. Actual read-during-write behavior in hardware depends on how synthesis tools infer RAM from generated HDL code, and on the hardware architecture of the target device.

Generate RAM Without Clock Enable Ports

The RAM blocks in the `hdldemo1lib` library implement RAM structures using HDL templates that include a clock enable signal.

However, some synthesis tools do not support RAM inference with a clock enable. As an alternative, the coder provides a generic style of HDL templates that do not use a clock enable signal for the RAM structures. The generic RAM template implements clock enable with logic in a wrapper around the RAM.

You may want to use the generic RAM style if your synthesis tool does not support RAM structures with a clock enable, and cannot map generated HDL code to FPGA RAM resources. To learn how to use generic style RAM for your design, see the Getting Started with RAM and ROM example. To open the example, type the following command at the MATLAB prompt:

```
hdlcoderramrom
```

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
```


HDL Counter

In this section...

“Overview” on page 13-15

“Counter Modes” on page 13-15

“Control Ports” on page 13-17

“Defining the Counter Data Type and Size” on page 13-20

“HDL Implementation and Implementation Parameters” on page 13-21

“Parameters and Dialog Box” on page 13-22

Overview



The HDL Counter block implements a free-running or count-limited hardware counter that supports signed and unsigned integer and fixed-point data types.

The counter emits its value for the current sample time from the **count** output. By default, the counter does not have input ports. Optionally, you can add control ports that let you enable, disable, load, or reset the counter, or set the direction (positive or negative) of the counter.

Counter Modes

The HDL Counter supports two operation modes, selected from the **Counter type** dropdown menu.

Free Running Mode (default)

The counter is initialized to the value defined by the **Initial value** parameter upon assertion of a reset signal. The reset signal can be either the model's global reset, or a reset received through an optional **Local reset port** that you can define on the HDL Counter block.

On each sample time, the value defined by the **Step value** parameter is added to the counter, and the counter emits its current value at the count output. When the counter value overflows or underflows the counter's word size, the counter wraps around and continues the counting sequence until reset is asserted or the model stops running.

By default, the positive or negative direction of the count is determined by the sign of the **Step value**. Optionally, you can define a **Count direction** control port on the HDL Counter block.

Free Running Mode Examples. For a 4-bit unsigned integer counter with an **Initial value** of 0 and a **Step value** of 5, the counter output sequence is

0, 5, 10, 15, 4, 9, 14, 3, . . .

For a 4-bit signed integer counter with an **Initial value** of 0 and a **Step value** of -2, the counter output sequence is

0, -2, -4, -6, -8, 6, 4, 2, 0, -2, -4, . . .

Count Limited Mode

The counter is initialized to the value defined by the **Initial value** parameter upon assertion of a reset signal. The reset signal can be either the model's global reset, or a reset received through an optional **Local reset port** that you can define on the HDL Counter block.

On each sample time, the value defined by the **Step value** parameter is added to the counter, and the current value is tested for equality with the value defined by the **Count to value** parameter. If the current value equals the **Count to value**, the counter is reloaded with the initial value. The counter then emits its current value at the count output.

If the counter value overflows or underflows the counter's word size, the counter wraps around and continues the counting sequence. The sequence continues until reset is asserted or the model stops running.

The condition for resetting the counter is exact equality. For some combinations of **Initial value**, **Step value**, and **Count to value**, the counter value may not reach the **Count to value**, or the counter may overflow and iterate through the counter range some number of times before reaching the **Count to value**.

By default, the positive or negative direction of the count is determined by the sign of the **Step value**. Optionally, you can define a **Count direction** control port on the HDL Counter block.

Count Limited Mode Examples. For an 8-bit signed integer counter with an **Initial value** of 0, a **Step value** of 2, and a **Count to value** of 8, the counter output sequence is

0 2 4 6 8 0 ...

For a 3-bit unsigned integer counter with an **Initial value** of 0, a **Step value** of 3, and a **Count to value** of 7, the counter output sequence is

0 3 6 1 4 7 0 3 6 1 4 7 ...

For a 3-bit unsigned integer counter with an **Initial value** of 0, a **Step value** of 2, and a **Count to value** of 7, the counter output sequence does not reach the **Count to value**:

0 2 4 6 0 2 4 6 ...

Control Ports

By default, the HDL Counter does not have inputs. Control ports are optional inputs that you can add to the block to:

- Reset the counter independently from the global reset logic.
- Load the counter with a value.
- Enable or disable the counter.
- Set the positive or negative direction of the counter.

The following figure shows the HDL Counter block configured with all available control ports.



The following characteristics apply to control ports:

- Control ports are synchronous.
- All control ports except the load value input have Boolean data type.
- Control ports must have the same sample time.
- If control ports exist on the block, the HDL Counter block inherits its sample time from the ports, and the **Sample time** parameter on the block dialog box is disabled.
- Signals at control ports are active-high.

Creating Control Ports for Loading and Resetting the Counter

By default, the counter is loaded (or reloaded) with the defined **Initial value** at the following times:

- When the model's global reset is asserted
- (In **Count limited** mode only) When the counter value equals the **Count to** value

You can further control reset and load behavior with signals connected to control ports. You can add these control ports to the block via the following options:

Local reset port: Select this option to create a reset input port on the block. The local reset port is labeled `rst`. The `rst` port should be connected to a Boolean signal. When this signal is set to 1, the counter resets to its initial value.

Load ports: When you select this option, two input ports, labeled `load` and `load_val`, are created on the block. The `load` port should be connected to a Boolean signal. When this signal is set to 1, the counter is loaded with the value at the `load_val` input. The load value must have the same data type as the counter.

Enabling or Disabling the Counter

When you select the **Count enable** port option, a control port labeled `enb` is created on the block. The `enb` port should be connected to a Boolean signal. When this signal is set to 0, the counter is disabled and the current counter value is held at the output. When the `enb` signal is set to 1, the counter resumes operation.

Controlling the Counter Direction

By default, the negative or positive direction of the counter is determined by the sign of the **Step value**. When you select the **Count direction** port option, a control port labeled `dir` is created on the block. The `dir` port should be connected to a Boolean signal. The `dir` signal determines the direction of the counter as follows:

- When the `dir` signal is set to 1, the step value is added to the current counter value to compute the next value.
- When the `dir` signal is set to 0, the step value is subtracted from the current counter value to compute the next value.

The following table summarizes the effect of the **Count direction** port.

Count Direction Signal Value	Step Value Sign	Actual Count Direction
1	+ (Positive)	Up
1	- (Negative)	Down
0	+ (Positive)	Down
0	- (Negative)	Up

Priority of Control Signals

The following table defines the priority of control signals, and shows how the counter value is set in relation to the control signals.

rst	load	enb	dir	Next Counter Value
1	–	–	–	initial value
0	1	–	–	load_val value
0	0	0	–	current value
0	0	1	1	current value + step value
0	0	1	0	current value - step value

Defining the Counter Data Type and Size

The HDL Counter block supports signed and unsigned integer and fixed-point data types. Use the following parameters to set the data type:

Output data type: Select Signed or Unsigned. The default is Unsigned.

Word length: Enter the desired number of bits (including the sign bit) for the counter.

Default: 8

Minimum: 1 if **Output data type** is Unsigned, 2 if **Output data type** is Signed

Maximum: 125

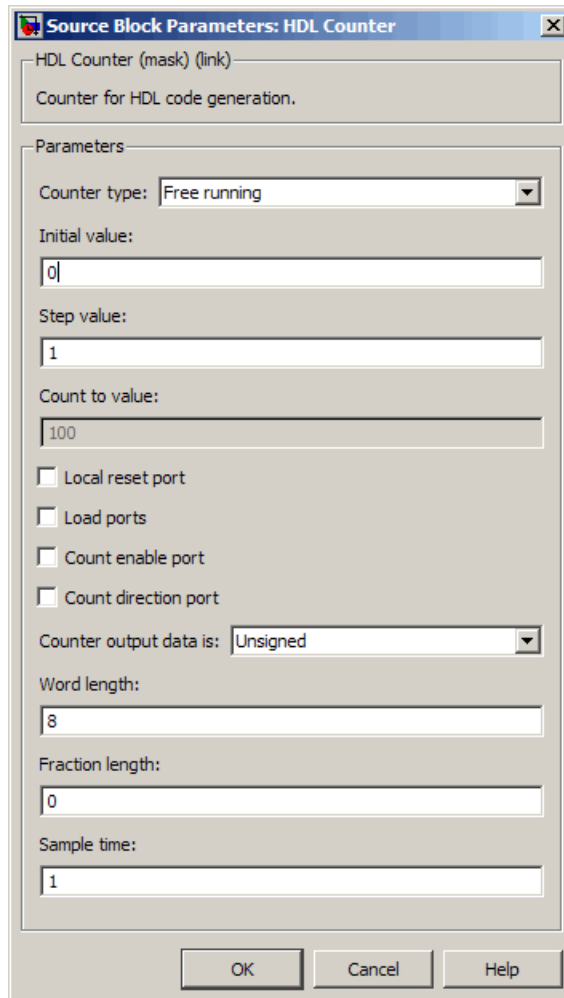
Fraction length: To define an integer counter, accept the default **Fraction length** of 0. To define a fixed-point counter, enter the number of bits to the right of the binary point.

HDL Implementation and Implementation Parameters

Implementation: default

Implementation Parameters: InputPipeline, OutputPipeline

Parameters and Dialog Box



Counter type

Default: Free running

This dropdown menu selects the operation mode of the counter (see “Counter Modes” on page 13-15). The operation modes are:

- Free running
- Count limited

When `Count limited` is selected, the **Count to value** field is enabled.

Initial value

Default: 0

By default, the counter is loaded (or reloaded) with the defined **Initial value** at the following times:

- When the model's global reset is asserted.
- (In **Count limited** mode only) When the counter value equals the **Count to value**. See also "Count Limited Mode" on page 13-16.

Step value

Default: 1

The **Step value** is an increment that is added to the counter on each sample time. By default (i.e., in the absence of a count direction control signal) the sign of the step value determines the count direction (see also "Controlling the Counter Direction" on page 13-19).

Set **Step value** to a nonzero value that can be represented in the counter's data type precision without rounding. The magnitude (absolute value) of the step value must be a number that can be represented with the counter's data type.

For a signed N-bit integer counter:

- The range of counter values is $-(2^{N-1}) \dots (2^{N-1} - 1)$.
- The range of legal step values is $-(2^{N-1} - 1) \dots (2^{N-1} - 1)$ (zero is excluded).

For example, for a 4-bit signed integer counter, the counter range is $[-8 \dots 7]$, but the ranges of legal step values are $[-7 \dots -1]$ and $[1 \dots 7]$.

Count to value

Default: 100

The **Count to value** field is enabled when the **Count limited counter mode** is selected. When the counter value is equal to the **Count to value**, the counter resets to the **Initial value** and continues counting. The condition for resetting the counter is exact equality. For some combinations of **Initial value**, **Step value**, and **Count to value**, the counter value may not reach the **Count to value**, or the counter may overflow and iterate through the counter range some number of times before reaching the **Count to value** (see “Count Limited Mode” on page 13-16).

Set **Count to value** to a value that is not equal to the **Initial value**.

Local reset port

Default: cleared

Select this option to create a reset input port on the block. Only Boolean signals should be connected to this port. The port is labeled **rst**. See “Creating Control Ports for Loading and Resetting the Counter” on page 13-18.

Load ports

Default: cleared

Select this option to create load and load value input ports on the block. The ports are labeled **load** and **load_val**, respectively. The signal applied to the **load** port must be Boolean. The signal applied to the **load_val** port must have the same data type as the counter. See also “Creating Control Ports for Loading and Resetting the Counter” on page 13-18.

Count enable port

Default: cleared

Select this option to create a count enable input port on the block. Only Boolean signals should be connected to this port. The port is labeled **enb**. See also “Enabling or Disabling the Counter” on page 13-19.

Count direction port

Default: cleared

Select this option to create a count direction input port on the block. Only Boolean signals should be connected to this port. The port is labeled dir. See also “Controlling the Counter Direction” on page 13-19.

Counter output data is:

Default: Unsigned

This dropdown menu selects whether the counter output is signed or unsigned.

Word length

Default: 8

Word length is a positive integer that defines the size, in bits, of the counter.

Minimum: 1 if **Output data type** is Unsigned, 2 if **Output data type** is Signed

Maximum: 125

Fraction length

Default: 0

To define an integer counter, accept the default **Fraction length** of 0. To define a fixed-point counter, enter the number of bits to the right of the binary point.

Default: 0

Sample time

Default: 1

If the HDL Counter block does not have input ports, the **Sample time** field is enabled, and an explicit sample time must be defined. Enter the desired sample time, or accept the default.

If the HDL Counter block has input ports, this field is disabled, and the block sample time is inherited from the input signals. All input signals must have the same sample time setting. (See also “Control Ports” on page 13-17.)

HDL FFT

In this section...

“Overview” on page 13-27

“Block Inputs and Outputs” on page 13-27

“HDL Implementation and Implementation Parameters” on page 13-30

“Parameters and Dialog Box” on page 13-30

Overview

The HDL FFT block implements a minimum resource FFT architecture.

In the current release, the HDL FFT block supports the Radix-2 with decimation-in-time (DIT) algorithm for FFT computation. See the FFT block reference section in the DSP System Toolbox documentation for more information about this algorithm.

The results returned by the HDL FFT block are bit-for-bit compatible with results returned by the DSP System Toolbox FFT block.

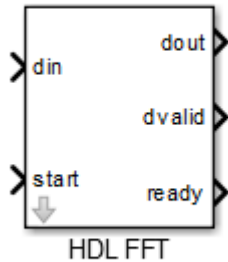
The operation of the HDL FFT block differs from the DSP System Toolbox FFT block, due to the requirements of hardware realization. The HDL FFT block:

- Requires serial input
- Generates serial output
- Operates in burst I/O mode

The HDL FFT block provides handshaking signals to support these features (see “Block Inputs and Outputs” on page 13-27).

Block Inputs and Outputs

As shown in the following figure, the HDL FFT block has two input ports and three output ports. Two of these ports are for data input and output signals. The other ports are for control signals.



The input ports are:

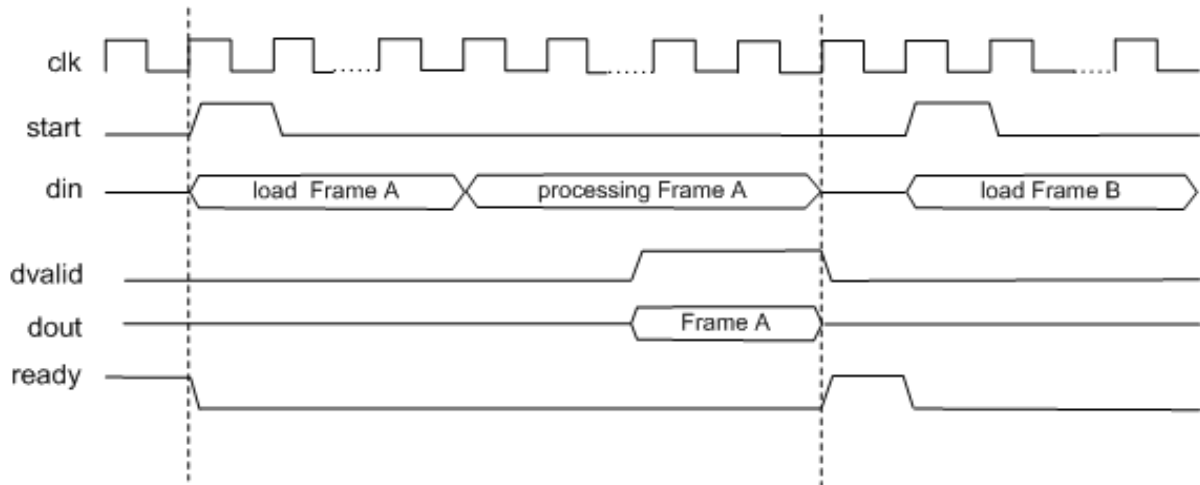
- **din**: The input data signal. A complex signal is required.
- **start**: Boolean control signal. When this signal is asserted true (1), the HDL FFT block initiates processing of a data frame.

The output ports are:

- **dout**: Data output signal. The Radix-2 with DIT algorithm produces output with linear ordering.
- **dvalid**: Boolean control signal. The HDL FFT block asserts this signal true (1) when a burst of valid output data is available at the **dout** port.
- **ready**: Boolean control signal. The HDL FFT block asserts this signal true (1) to indicate that it is ready to process a new frame.

Configuring Control Signals

For efficient hardware deployment of the HDL FFT block, the timing of the block's input and output data streams must be considered carefully. The following figure shows the timing relationships between the system clock and the **start**, **ready**, and **dvalid** signals.



When `ready` is asserted, the `start` signal (active high) triggers the FFT block. The high cycle period of the `start` signal does not affect the behavior of the block.

One clock cycle after the `start` trigger, the block begins to load data and the `ready` signal is deasserted. During the interval when the block is loading, processing, and outputting data, `ready` is low and the `start` signal is ignored.

The `dvalid` signal is asserted high for N clock cycles (where N is the FFT length) after processing is complete. `ready` is asserted again after the N -point FFT outputs are sent out.

The expression `Tcycle` denotes the total number of clock cycles required by the HDL FFT block to complete an FFT of length N . `Tcycle` is defined as follows:

- Where $N > 8$

$$\text{Tcycle} = 3N/2 - 2 + \log_2(N) * (N/2 + 3);$$

- Where $N = 8$

$$\text{Tcycle} = 3N/2 - 1 + \log_2(N) * (N/2 + 3);$$

Given `Tcycle`, you can then define the period between assertions of the HDL FFT start signal in a way that is suitable to your application. In the “Using the Minimum Resource HDL FFT” example, this period is computed and assigned to the variable `startLen`, as follows:

```
if (N<=8)
startLen = (ceil(Tcycle/N)+1)*N;
else
startLen = ceil(Tcycle/N)*N;
end
```

In the example model, `startLen` determines the period of a Pulse Generator that drives the HDL FFT block’s `start` input. These values are computed in the model’s initialization function (`InitFcn`), which is defined in the **Callbacks** pane of the Simulink Model Explorer.

The HDL FFT block asserts and deasserts the `ready` and `dvalid` signals automatically. These signals are routed to the model components that write to and read from the HDL FFT block.

HDL Implementation and Implementation Parameters

Implementation: default

Implementation Parameters: InputPipeline, OutputPipeline

Parameters and Dialog Box

The following figure shows the HDL FFT block dialog box, with parameters at their default settings.

Function Block Parameters: HDL FFT

HDL FFT (mask) (link)

HDL FFT block. This block reads serial input and generates serial output.

Parameters

FFT Length
8

Rounding mode Floor

Overflow mode Saturate

Sine table Same word length as input

Sine table word length
10

Product output Same as input

Product word length
16

Product fraction length
13

Accumulator Same as input

Accumulator word length
18

Accumulator fraction length
10

Output Same as input

Output word length
16

Output fraction length
8

OK Cancel Help Apply

FFT Length

Default: 8

The FFT length must be a power of 2, in the range $2^3 .. 2^{16}$.

Rounding mode

Default: Floor

The HDL FFT block supports all rounding modes of the DSP System Toolbox FFT block. See also the FFT block reference section in the DSP System Toolbox documentation.

Overflow mode

Default: Saturate

The HDL FFT block supports all overflow modes of the DSP System Toolbox FFT block. See also the FFT block reference section in the DSP System Toolbox documentation.

Sine table

Default: Same word length as input

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values is equal to the word length minus one.

- When you select **Same word length as input**, the word length of the sine table values match that of the input to the block.
- When you select **Specify word length**, you can enter the word length of the sine table values, in bits, in the **Sine table word length** field. The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; they always saturate and round to Nearest.

Product output

Default: Same as input

Use this parameter to specify how you want to designate the product output word and fraction lengths:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits, in the **Product word length** and **Product fraction length** fields.

Accumulator

Default: Same as input

Use this parameter to specify how you want to designate the accumulator word and fraction lengths:

When you select **Same as product output**, these characteristics match those of the product output.

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits, in the **Accumulator word length** and **Accumulator fraction length** fields.

Output

Default: Same as input

Choose how you specify the output word length and fraction length:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits, in the **Output word length** and **Output fraction length** fields.

Note The HDL FFT block skips the divide-by-two operation on butterfly outputs for fixed-point signals.

Signal Processing with the HDL FFT Block

To get started with the HDL FFT block, run the “Using the Minimum Resource HDL FFT” example, which is located in the HDL Coder/Signal Processing example library.

The example illustrates the use of the HDL FFT block in simulation. The model includes buffering and control logic that handles serial input and output. In the example, a complex source signal is stored as a series of samples in a FIFO. Samples from the FIFO are processed serially by the HDL FFT block, which emits a stream of scalar FFT data.

For comparison, the same source signal is also processed by the frame-based DSP System Toolbox FFT block. The output frames from the DSP System Toolbox FFT block are buffered into a FIFO and compared to the output of the HDL FFT block. Examination of the results shows the outputs to be identical.

HDL FIFO

In this section...

“Overview” on page 13-36

“Block Inputs and Outputs” on page 13-36

“HDL Implementation and Implementation Parameters” on page 13-37

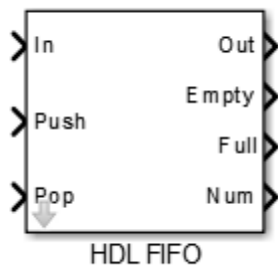
“Parameters and Dialog Box” on page 13-37

Overview

The HDL FIFO block stores a sequence of input samples in a first in, first out (FIFO) register. The HDL FIFO block closely resembles the Queue block of the DSP System Toolbox, but with HDL-related enhancements such as multirate support.

Block Inputs and Outputs

The following figure shows the HDL FIFO block with input and output ports enabled.



The input ports are:

- In: The data input signal.
- Push: Control signal. When this port receives a value of 1, the block pushes the input at the In port onto the end of the FIFO register.

- **Pop:** Control signal. When this port receives a value of 1, the block pops the first element off the FIFO register and holds the Out port at that value

The output ports are:

- **Out:** The data output signal.
- **Empty:** The block asserts this signal true (1) when the FIFO register is empty. Display of this port is optional.
- **Full:** The block asserts this signal true (1) to indicate that the FIFO register is full. Display of this port is optional.
- **Num:** The current number of data values in the FIFO register. Display of this port is optional.

In the event that two or more of the control input ports are triggered at the same time step, the operations execute in the following order:

1 Pop

2 Push

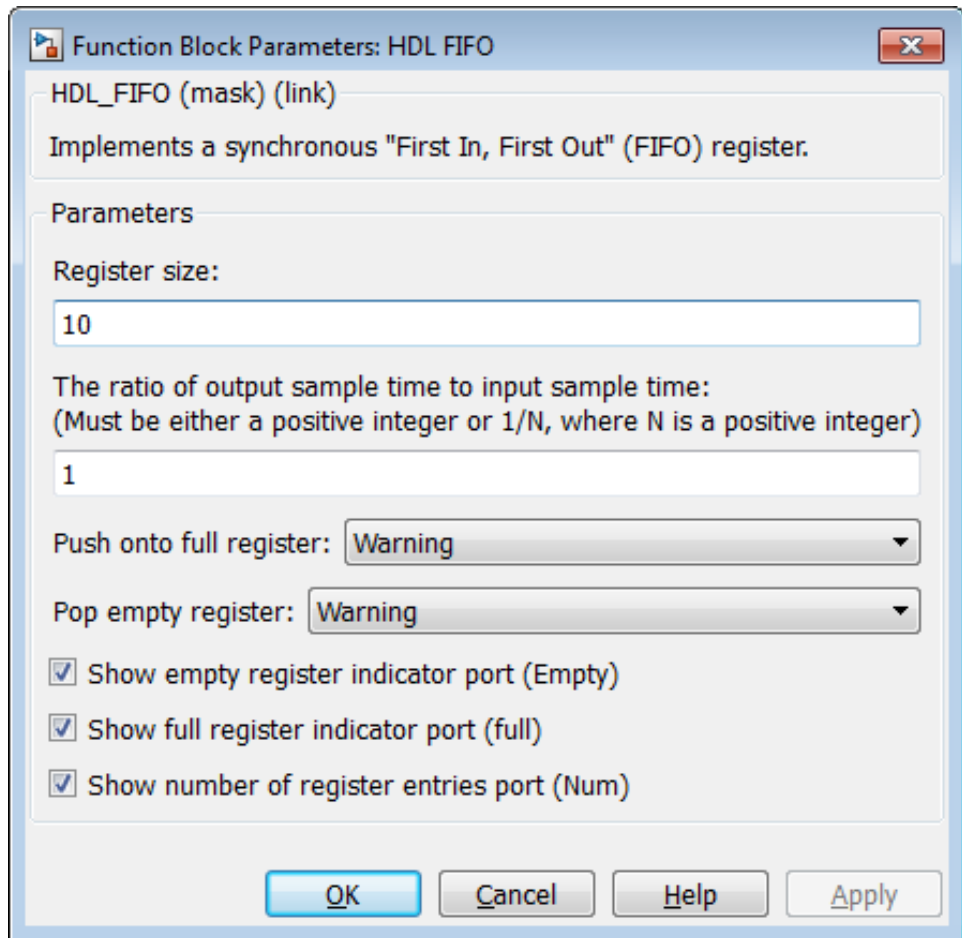
HDL Implementation and Implementation Parameters

Implementation: default

Implementation Parameters: InputPipeline, OutputPipeline

Parameters and Dialog Box

The following figure shows the HDL FIFO block dialog box, with parameters at their default settings.



- **Register size:** Specify the number of entries that the FIFO register can hold.
Default: 10
- **The ratio of output sample time to input sample time:** Inputs (In, Push) and outputs (Out, Pop) can run at different sample times. Enter the ratio of output sample time to input sample time. The value must be a positive integer or 1/N, where N is a positive integer.

For example:

- If you enter 2, the output sample time is twice the input sample time, meaning the outputs run slower.
- If you enter 1/2, the output sample time is half the input sample time, meaning the outputs run faster.

The Full, Empty, and Num signals run at the faster rate.

Default: 1

- **Push onto full register:** Response (Ignore, Error, or Warning) to a trigger received at the Push port when the register is full.

Default: Warning

- **Pop empty register:** Response (Ignore, Error, or Warning) to a trigger received at the Pop port when the register is empty.

Default: Warning

- **Show empty register indicator port (Empty):** Enable the Empty output port, which is high (1) when the FIFO register is empty, and low (0) otherwise.
- **Show full register indicator port (Full):** Enable the Full output port, which is high (1) when the FIFO register is full, and low (0) otherwise.
- **Show number of register entries port (Num):** Enable the Num output port, which tracks the number of entries currently on the queue.

HDL Streaming FFT

In this section...

“Overview” on page 13-40

“HDL Streaming FFT Block Example” on page 13-40

“Block Inputs and Outputs” on page 13-40

“Timing Description” on page 13-41

“HDL Implementation and Implementation Parameters” on page 13-45

“Parameters and Dialog Box” on page 13-45

Overview

The HDL Streaming FFT block supports the Radix-2 with decimation-in-frequency (DIF) algorithm for FFT computation. See the FFT block reference section in the DSP System Toolbox documentation for more information about this algorithm.

The HDL Streaming FFT block returns results identical to results returned by the Radix-2 DIF algorithm of the DSP System Toolbox FFT block.

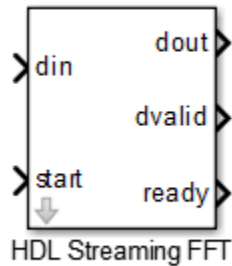
HDL Streaming FFT Block Example

To get started with the HDL Streaming FFT block, run the “OFDM Receiver with 512-Point Streaming I/O FFT” example, which is in the HDL Coder/Signal Processing example library.

The example implements a simple OFDM transmitter and receiver. The model compares the results obtained from the DSP System Toolbox FFT block to results obtained from the HDL Streaming FFT block.

Block Inputs and Outputs

As shown in the following figure, the HDL Streaming FFT block has two input ports and three output ports. Two of these ports are for data input and output signals. The other ports are for control signals.



The block has the following input ports:

- **din**: The input data signal. The coder requires a complex fixed-point signal.
- **start**: Boolean control signal. When **start** asserts true (1), the HDL Streaming FFT block initiates processing of a data frame.

The block has the following output ports:

- **dout**: Data output signal.
- **dvalid**: Boolean control signal. The HDL Streaming FFT block asserts this signal true (1) when a stream of valid output data is available at the **dout** port.
- **ready**: Boolean control signal. The HDL Streaming FFT block asserts this signal true (1) to indicate that it is ready to process a new frame.

Timing Description

The HDL Streaming FFT block operates in one of two modes:

- *Continuous data streaming* mode: In this mode, the HDL Streaming FFT block expects to receive a continuous stream of data at **din**. After an initial delay, the block produces a continuous stream of data at **dout**.
- *Non-continuous data streaming* mode: In this mode, the HDL Streaming FFT block receives non-continuous bursts of streaming data at **din**. After

an initial delay, the block produces non-continuous bursts of streaming data at `dout`.

The behavior of the control signals determines the timing mode of the block.

Continuous Data Streaming Timing

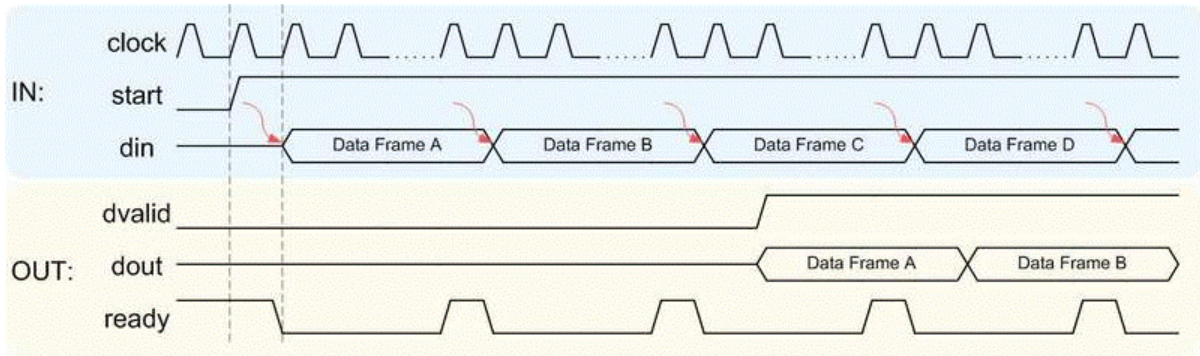
Assertion of the `start` signal (active high) triggers processing by the HDL Streaming FFT block. To initiate continuous data stream processing, assert the `start` signal in one of the following ways:

- Hold the `start` signal high (as shown in Continuous Data Streaming with Start Signal Held High on page 13-43).
- Pulse the `start` signal every `N` clock cycles, where `N` is the FFT length (as shown in Continuous Data Streaming With Pulsed Start Signal on page 13-43).

One clock cycle after the `start` trigger, the block begins to load data at `din`. After the first frame of streaming data, the block starts to receive the next frame of streaming data.

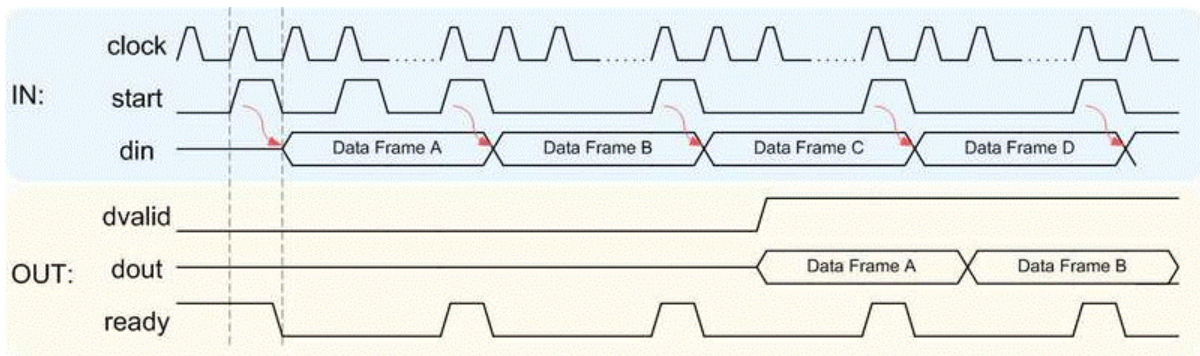
Meanwhile, the block performs the FFT calculation on the incoming data frames and outputs the results continuously at `dout`. The HDL Streaming FFT block asserts and deasserts the `ready` and `dvalid` signals automatically. The block asserts `dvalid` high whenever the output data stream is valid. The block asserts `ready` high to indicate that the block is ready to load a new data frame. When `ready` is low, the block ignores the `start` signal.

The following figures illustrate continuous data streaming. Each data frame corresponds to a stream of `N` input data values, where `N` is the FFT length.



Continuous Data Streaming with Start Signal Held High

Note The start signal can be a single cycle pulse; it need not be held high for the entire data frame. When processing for a frame begins, further pulses on start do not affect processing of that frame. However, a start pulse must occur at the beginning of each data frame.



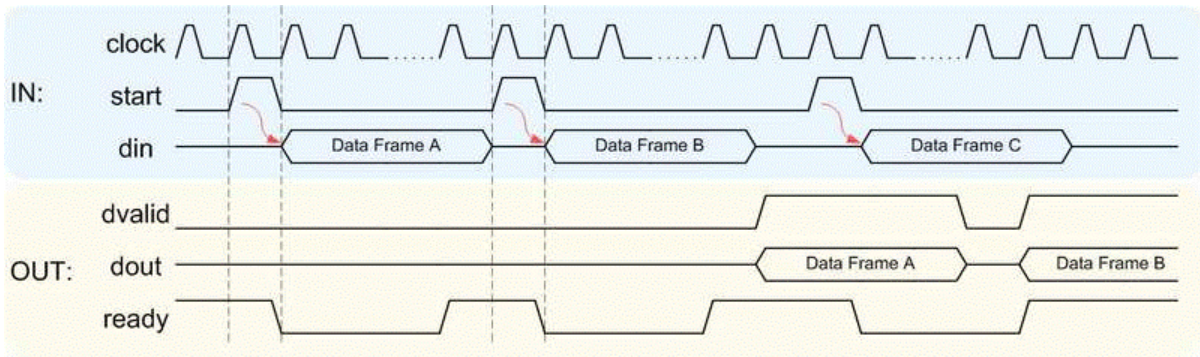
Continuous Data Streaming With Pulsed Start Signal

Non-Continuous Data Streaming Timing

In this mode, the HDL Streaming FFT block receives continuous bursts of streaming data at din. After an initial delay, the block produces non-continuous bursts of streaming data at dout. Breaks occur between data frames when the following condition exist:

- The start signal does not assert every N clock cycles (where N is the FFT length)
- The start signal is not continuously held high.

Non-continuous data streaming mode allows you more flexibility in determining the intervals between input data streams.

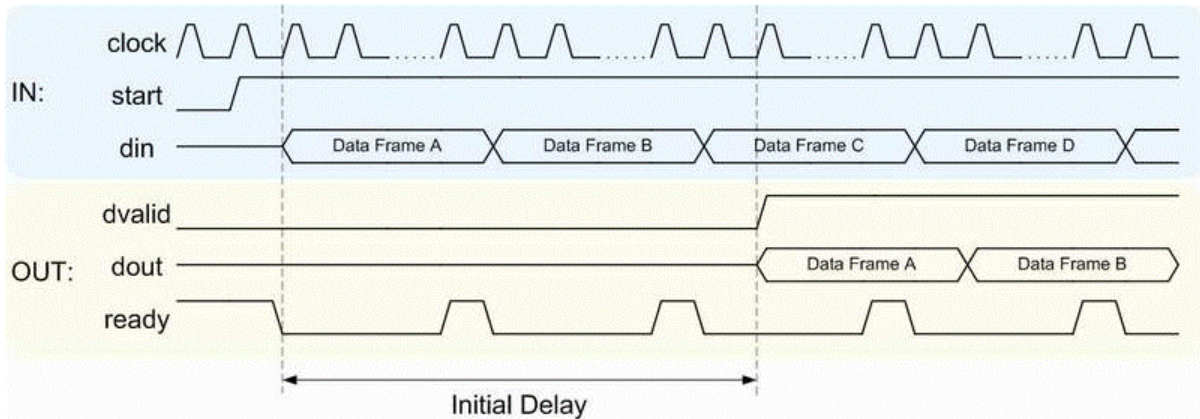


Initial Delay

The initial delay of the HDL Streaming FFT block is the interval between the following times:

- The time the block begins to receive the first frame of input data
- The time the block asserts dvalid and produces the first valid output data.

The initial delay represents the time the block uses to load a data frame, calculate the FFT, and output the beginning of the first output frame. The following figure illustrates the initial delay.



If you select the block option **Display computed initial delay on mask**, the block icon displays the initial delay. The display represents the delay time as Z^{-n} , where n is the delay time in samples.

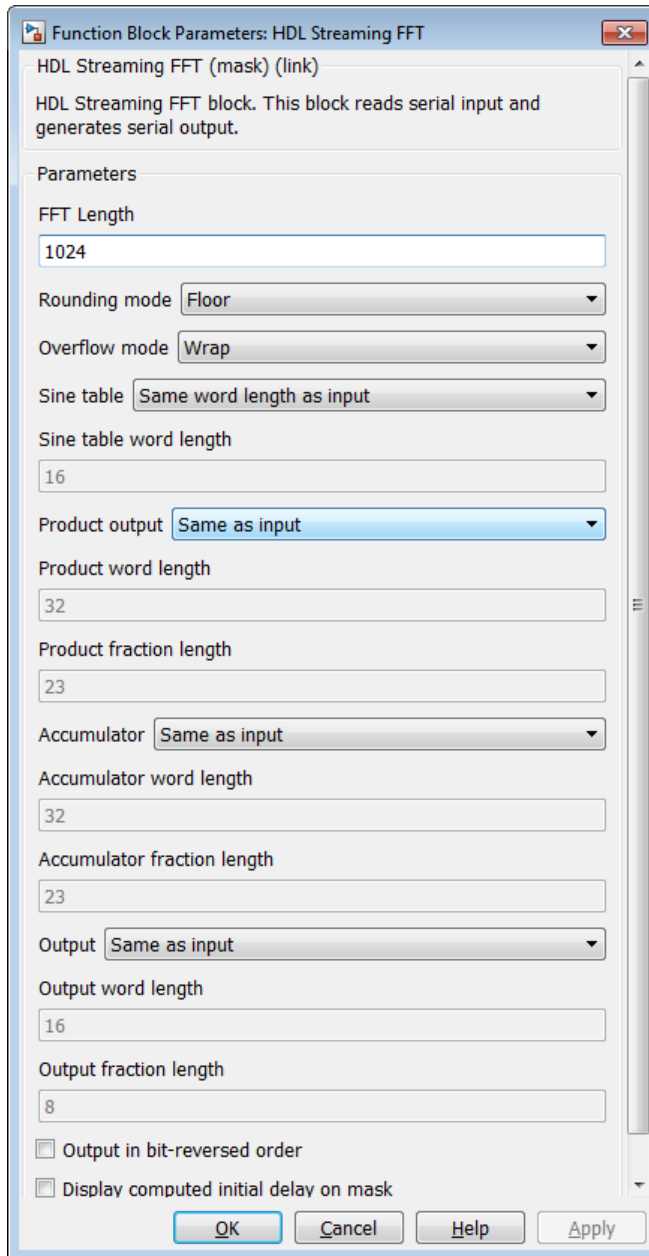
HDL Implementation and Implementation Parameters

Implementation: default

Implementation Parameters: InputPipeline, OutputPipeline

Parameters and Dialog Box

The following figure shows the HDL Streaming FFT block dialog box, with parameters at their default settings.



FFT Length

Default: 1024

The FFT length must be a power of 2, in the range 2^3 to 2^{16} .

Rounding mode

Default: Floor

The HDL Streaming FFT block supports all rounding modes of the DSP System Toolbox FFT block. See also the FFT block reference section in the DSP System Toolbox documentation.

Overflow mode

Default: Wrap

The HDL Streaming FFT block supports all overflow modes of the DSP System Toolbox FFT block. See also the FFT block reference section in the DSP System Toolbox documentation.

Sine table

Default: Same word length as input

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values is equal to the word length minus one.

- When you select **Same word length as input**, the word lengths of the sine table values match the word lengths of the block inputs.
- When you select **Specify word length**, you can enter the word length of the sine table values, in bits, in the **Sine table word length** field. The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters. They always saturate and round to Nearest.

Product output

Default: Same as input

Use this parameter to specify how you want to designate the product output word and fraction lengths:

- When you select **Same as input**, these characteristics match the characteristics of the input to the block.
- **Binary point scaling**: Enter the word length and the fraction length of the product output, in bits, in the **Product word length** and **Product fraction length** fields.

Accumulator

Default: Same as input

Use this parameter to specify how you want to designate the accumulator word and fraction lengths:

When you select **Same as product output**, these characteristics match the characteristics of the product output.

- When you select **Same as input**, these characteristics match the characteristics of the input to the block.
- **Binary point scaling**: Enter the word length and the fraction length of the accumulator, in bits, in the **Accumulator word length** and **Accumulator fraction length** fields.

Output

Default: Same as input

Choose how you specify the output word length and fraction length:

- **Same as input**: these characteristics match the characteristics of the input to the block.
- **Binary point scaling**: lets you enter the word length and fraction length of the output, in bits, in the **Output word length** and **Output fraction length** fields.

Output in bit-reversed order

Default: Off

- On: The output data stream is in bit-reversed order.
- Off: The output data stream is in natural order.

For more information about the effects of bit reversal, see “Linear and Bit-Reversed Output Order” in the DSP System Toolbox documentation.

Display computed initial delay on mask

Default: Off

- On: The block icon displays the initial delay as Z^{-n} , where n is the delay time in samples.
- Off: The block icon does not display the initial delay.

Note **Sine table**, **Product output**, **Accumulator**, and **Output** do not support:

- Inherit via internal rule
 - Slope and bias scaling
-

Bitwise Operators

In this section...

“Overview of Bitwise Operator Blocks” on page 13-50

“Bit Concat” on page 13-52

“Bit Reduce” on page 13-55

“Bit Rotate” on page 13-57

“Bit Shift” on page 13-59

“Bit Slice” on page 13-61

Overview of Bitwise Operator Blocks

The Bitwise Operator sublibrary provides commonly used operations on bits and bit fields.

Bitwise Operator blocks support:

- Scalar and vector inputs
- Fixed-point, integer (signed or unsigned), and Boolean data types
- A maximum word size of 128 bits

Bitwise Operator blocks do not currently support:

- Double, single, or complex data types
- Matrix inputs

To open the Bitwise Operators sublibrary, double-click its icon

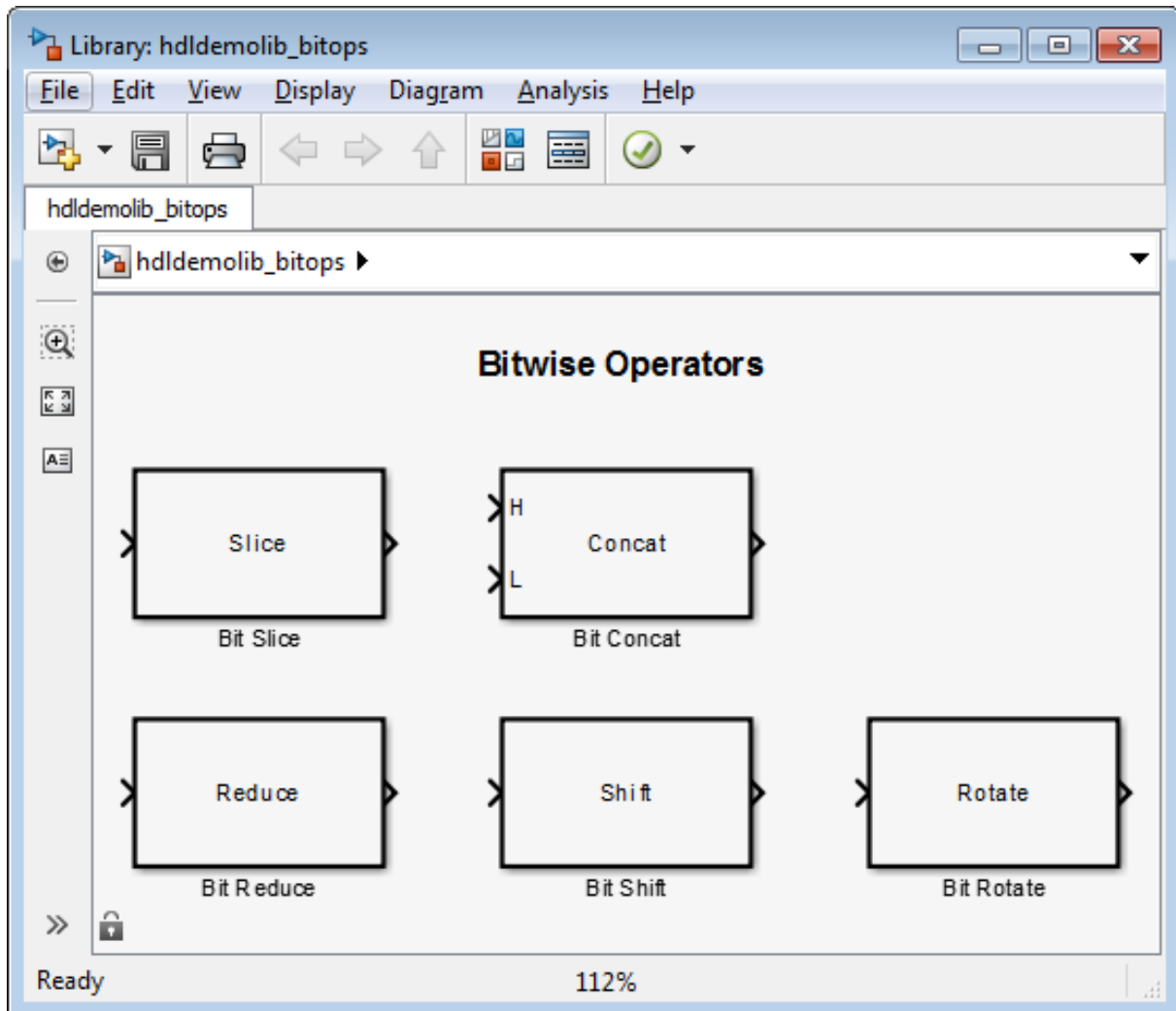


Bitwise Operators

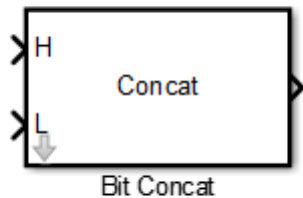
in the `hdldemo1ib` window. Alternatively, you can open the Bitwise Operators sublibrary directly by typing the following command at the MATLAB prompt:

hdl demolib_bitops

The following figure shows the Bitwise Operators sublibrary.



Bit Concat



Description

The Bit Concat block concatenates up to 128 input words into a single output. The input port labeled L designates the lowest-order input word; the port labeled H designates the highest-order input word. The right-left ordering of words in the output follows the low-high ordering of input signals.

The operation of the block depends on the number and dimensions of the inputs, as follows:

- Single input: The input can be a scalar or a vector. When the input is a vector, the coder concatenates the individual vector elements together.
- Two inputs: Inputs can be any combination of scalar and vector. When one input is scalar and the other is a vector, the coder performs scalar expansion. Each vector element is concatenated with the scalar, and the output has the same dimension as the vector. When both inputs are vectors, they must have the same size.
- Three or more inputs (up to a maximum of 128 inputs): Inputs must be uniformly scalar or vector. All vector inputs must have the same size.

Data Type Support

- Input: Fixed-point, integer (signed or unsigned), Boolean

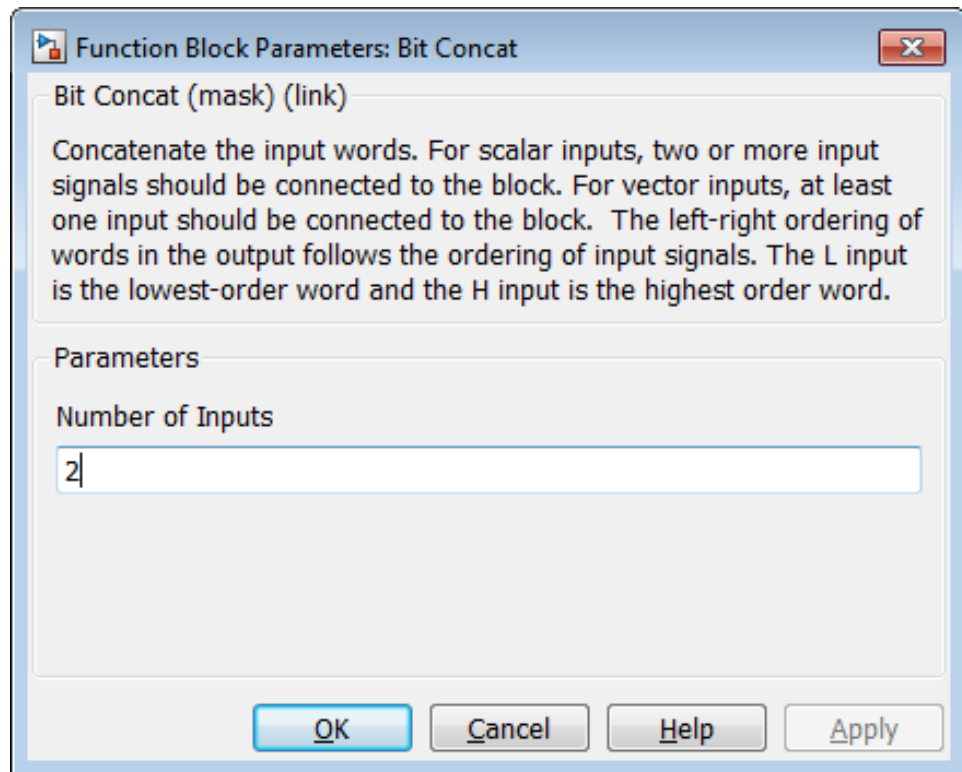
- Output: Unsigned fixed-point or integer (Maximum concatenated output word size: 128 bits)

HDL Implementation and Implementation Parameters

Implementation: default

Implementation Parameters: InputPipeline, OutputPipeline

Parameters and Dialog Box

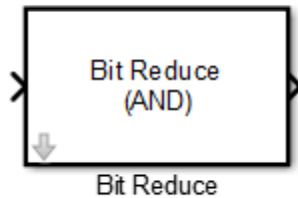


Number of Inputs: Enter an integer specifying the number of input signals. The number of input ports displayed on the block updates when **Number of Inputs** changes.

- Default: 2.
- Minimum: 1
- Maximum: 128

Caution Make sure that the **Number of Inputs** is equal to the number of signals you connect to the block. If unconnected inputs are present on the block, an error will occur at code generation time.

Bit Reduce



Description

The Bit Reduce block performs a selected bit reduction operation (AND, OR, or XOR) on all the bits of the input signal, reducing it to a single-bit result.

Data Type Support

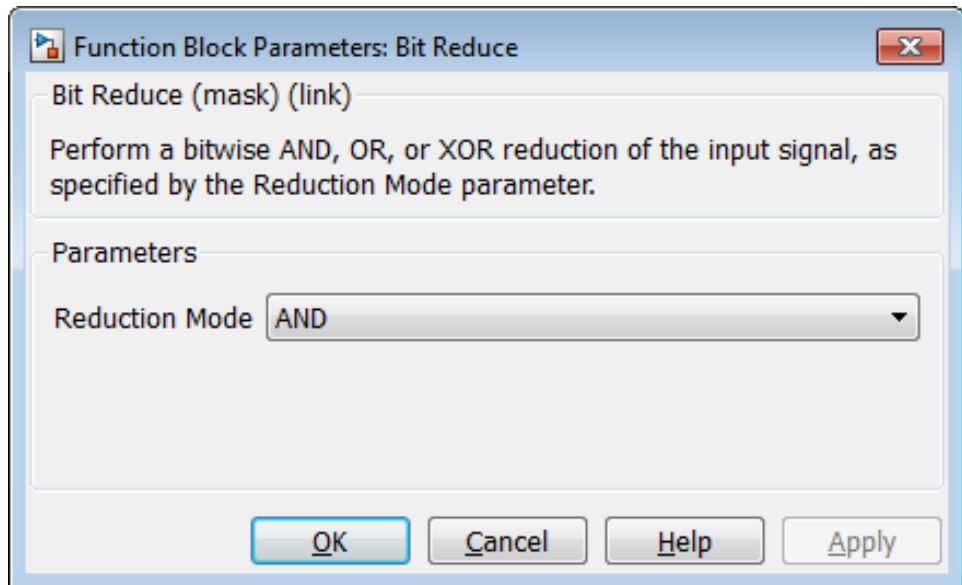
- Input: Fixed-point, integer (signed or unsigned), Boolean
- Output: `ufix1`

HDL Implementation and Implementation Parameters

Implementation: `default`

Implementation Parameters: `InputPipeline`, `OutputPipeline`

Parameters and Dialog Box



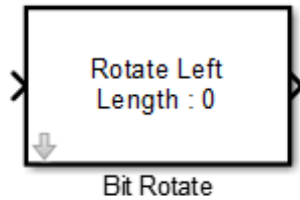
Reduction Mode

Default: AND

Specifies the reduction operation, as follows:

- AND: Perform a bitwise AND reduction of the input signal.
- OR: Perform a bitwise OR reduction of the input signal.
- XOR: Perform a bitwise XOR reduction of the input signal.

Bit Rotate



Description

The Bit Rotate block rotates the input signal left or right by a specified number of bit positions.

Data Type Support

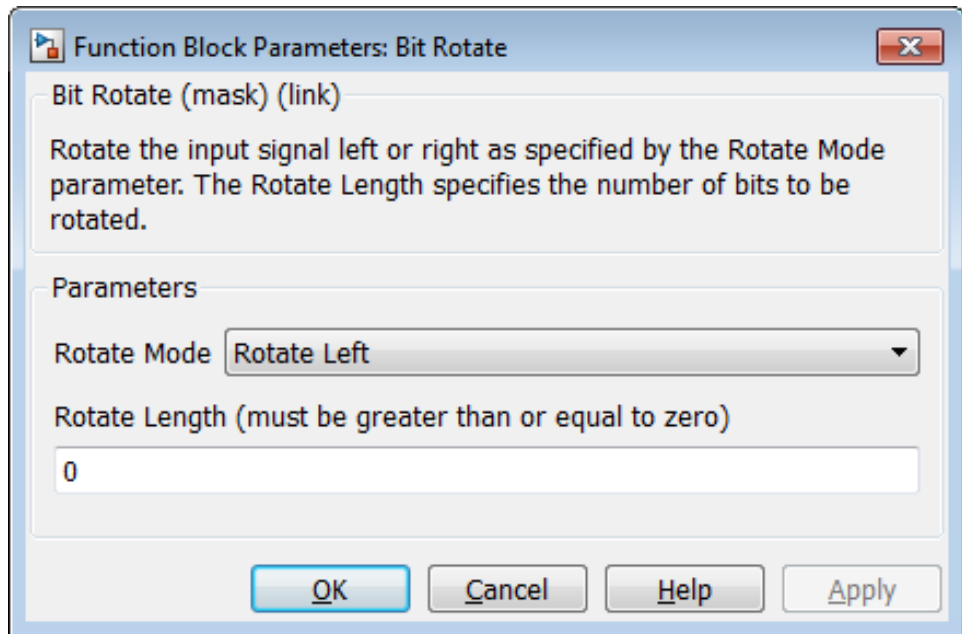
- Input: Fixed-point, integer (signed or unsigned), Boolean
 - Minimum word size: 2 bits
 - Maximum word size: 128 bits
- Output: Has the same data type as the input signal

HDL Implementation and Implementation Parameters

Implementation: default

Implementation Parameters: InputPipeline, OutputPipeline

Parameters and Dialog Box



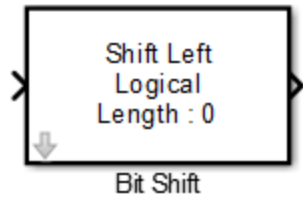
Rotate Mode: Specifies direction of rotation, either left or right.

Default: Rotate Left

Rotate Length: Specifies the number of bits to be rotated. **Rotate Length** must be greater than or equal to zero.

Default: 0

Bit Shift



Description

The Bit Shift block performs a logical or arithmetic shift on the input signal.

Data Type Support

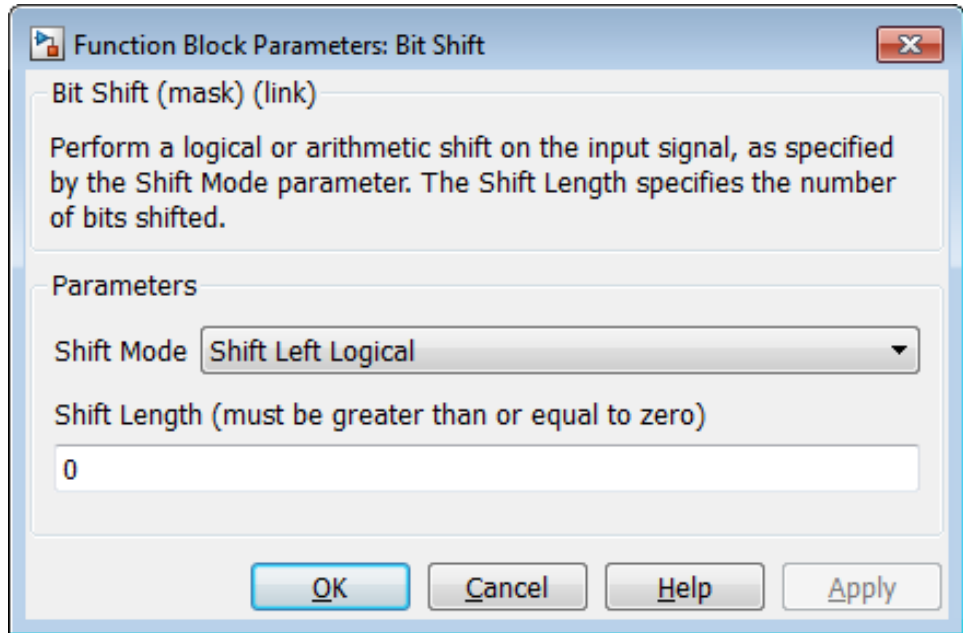
- Input: Fixed-point, integer (signed or unsigned), Boolean
 - Minimum word size: 2 bits
 - Maximum word size: 128 bits
- Output: Has the same data type as the input signal

HDL Implementation and Implementation Parameters

Implementation: default

Implementation Parameters: InputPipeline, OutputPipeline

Parameters and Dialog Box



Shift Mode

Default: Shift Left Logical

Specifies the type and direction of shift, as follows:

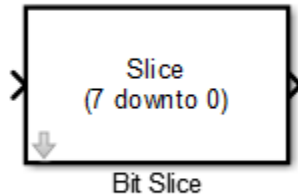
- Shift Left Logical
- Shift Right Logical
- Shift Right Arithmetic

Shift Length

Default: 0

Specifies the number of bits to be shifted. **Shift Length** must be greater than or equal to zero.

Bit Slice



Description

The Bit Slice block returns a field of consecutive bits from the input signal. The lower and upper boundaries of the bit field are specified by zero-based indices entered in the **LSB Position** and **MSB Position** parameters.

Data Type Support

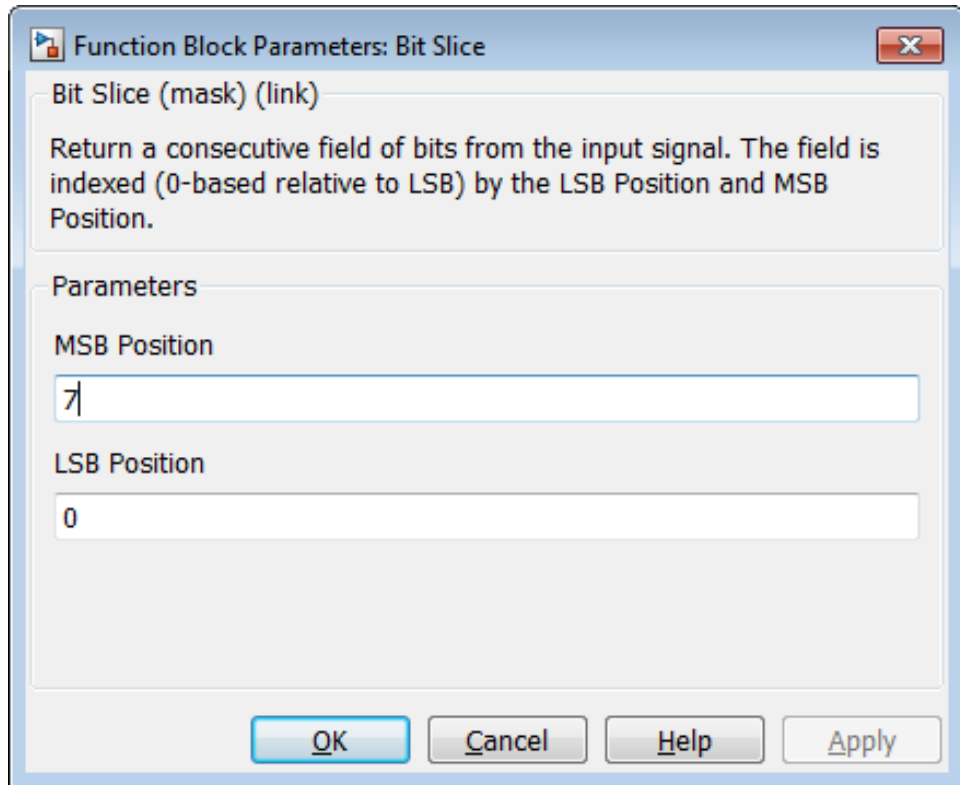
- Input: Fixed-point, integer (signed or unsigned), Boolean
- Output: unsigned fixed-point or unsigned integer

HDL Implementation and Implementation Parameters

Implementation: default

Implementation Parameters: InputPipeline, OutputPipeline

Parameters and Dialog Box



MSB Position

Default: 7

Specifies the bit position (zero-based) of the most significant bit (MSB) of the field to be extracted.

For an input word size *WS*, **LSB Position** and **MSB Position** should satisfy the following constraints:

$$WS > \text{MSB Position} \geq \text{LSB Position} \geq 0;$$

The word length of the output is computed as (MSB Position - LSB Position) + 1.

LSB Position

Default: 0

Specifies the bit position (zero-based) of the least significant bit (LSB) of the field to be extracted.

Generating Bit-True Cycle-Accurate Models

- “What Is the Generated Model?” on page 14-2
- “Locate Numeric Differences After Speed Optimization” on page 14-4
- “View Latency Differences After Area Optimization” on page 14-9
- “Defaults and Options for Generated Models” on page 14-12
- “Limitations for Generated Models” on page 14-17

What Is the Generated Model?

In some circumstances, significant differences in behavior can arise between a Simulink model and the HDL code generated from that model. Such differences fall into two categories:

- *Numerics*: differences in intermediate and/or final computations. For example, a selected block implementation may restructure arithmetic operations to optimize for speed (see “Locate Numeric Differences After Speed Optimization” on page 14-4). Where such numeric differences exist, the HDL code is no longer *bit-true* to the model.
- *Latency*: insertion of delays of one or more clock cycles at certain points in the HDL code. Some block implementations that optimize for area can introduce these delays. Where such latency exists, the timing of the HDL code is no longer *cycle-accurate* with respect to the model.

To help you evaluate such cases, the coder creates a *generated model* that is bit-true and cycle-accurate with respect to the generated HDL code. The generated model lets you:

- Run simulations that reflect the behavior of the generated HDL code.
- Create test benches based on the generated model, rather than the original model.
- Visually detect (by color highlighting of affected subsystems) differences between the original and generated models.

The coder creates a generated model as part of the code generation process, and generates test benches based on the generated model, rather than the original model. In cases where no latency or numeric differences occur, you can disregard the generated model except when generating test benches.

The coder also provides options so that you can:

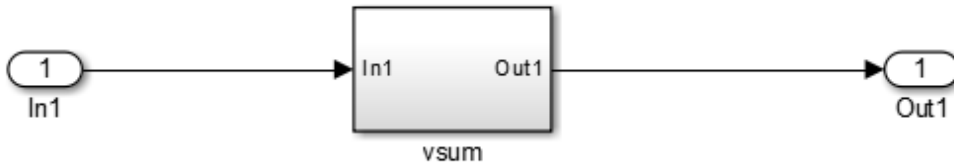
- Specify the color highlighting of differences between the original and generated models.
- Specify a name or prefix for the generated model.

“Defaults and Options for Generated Models” on page 14-12 describes these options.

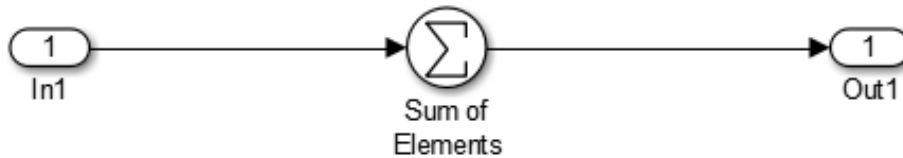
Locate Numeric Differences After Speed Optimization

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.

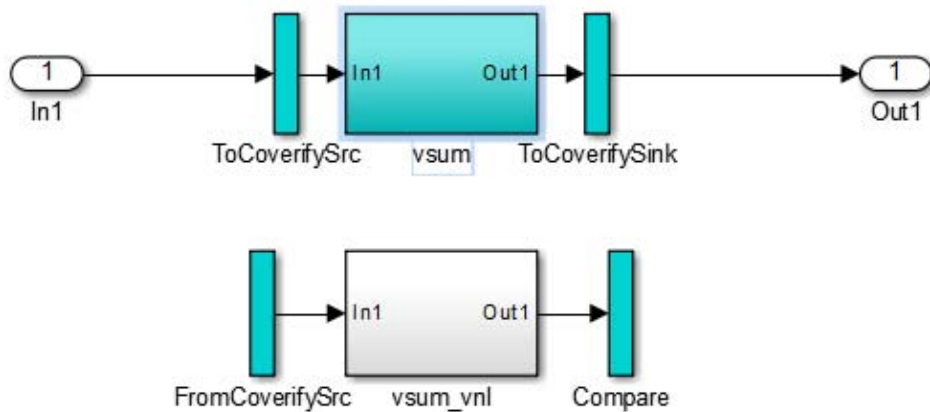


The model is configured to use the **Tree** implementation when generating HDL code for the Sum block within the `vsum` subsystem. This implementation, optimized for minimal latency, generates a tree-shaped structure of adders for the Sum block.

To select a nondefault implementation for an individual block:

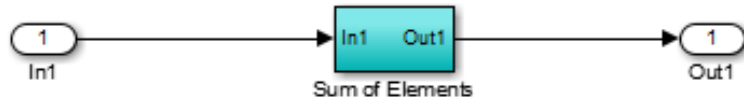
- 1** Right-click the block and select **HDL Code > HDL Block Properties** .
- 2** In the HDL Properties dialog box, select the desired implementation from the **Architecture** menu.
- 3** Click **Apply** and close the dialog box.

After code generation, you can view the validation model, `gm_simplevectorsum_tree_vnl`.

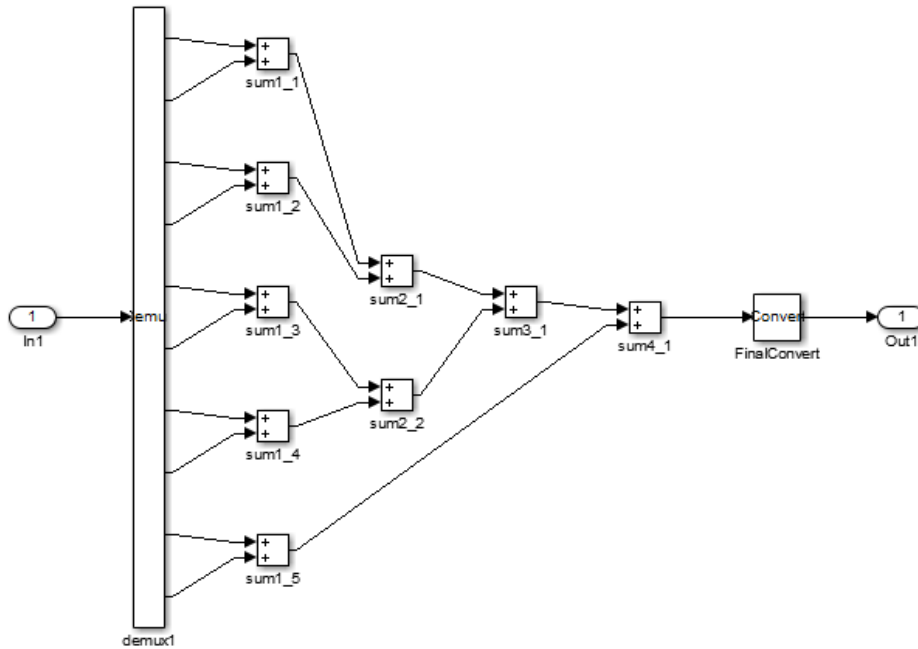


The `vsum` subsystem has been highlighted in cyan. This highlighting indicates that the subsystem differs in some respect from the `vsum` subsystem of the original model.

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.

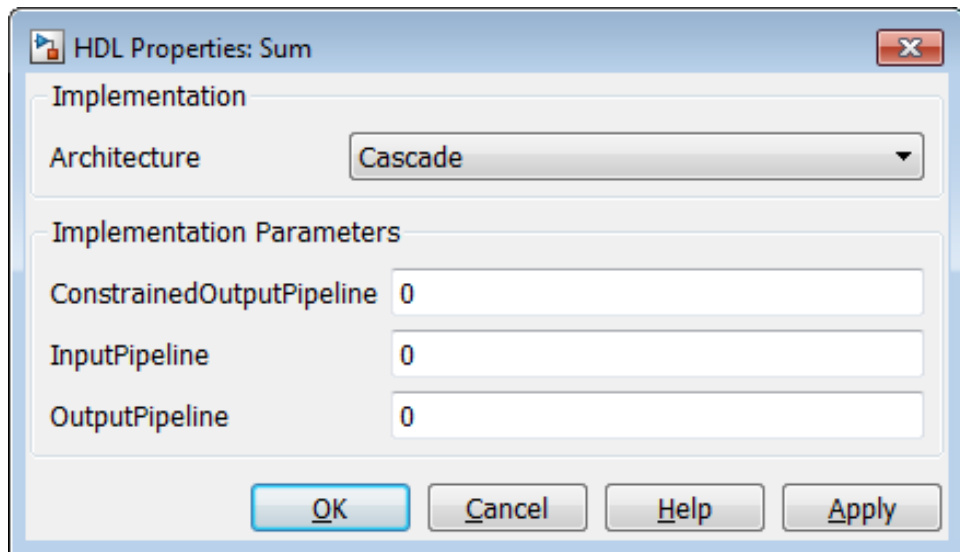


The generated model implements the vector sum as a tree of adders (Sum blocks). The vector input signal is demultiplexed and connected, as five pairs of operands, to the five leftmost adders. The widths of the adder outputs increase from left to right, as required to avoid overflow in computing intermediate results.

View Latency Differences After Area Optimization

This example uses the `simplevectorsum_cascade` model. This model is identical to the model in (“Locate Numeric Differences After Speed Optimization” on page 14-4), except that it uses a cascaded implementation for the Sum block. This implementation introduces latency differences.

The following figure shows the HDL Properties dialog box for a Sum block, with the Cascade implementation selected. This implementation generates a cascade of adders for the Sum block.



In the generated code, partial sums are computed by adders arranged in a cascade structure. Each adder computes a partial sum by demultiplexing and adding several inputs in succession. These computation take several clock cycles. On each cycle, an addition is performed; the result is then added to the next input.

To complete computations within one sample period, the system master clock runs faster than the nominal sample rate of the system. A latency of one clock cycle (in the case of this model) is required to transmit the final result to the

output. The inputs cannot change until the computations are complete and the final result is presented at the output.

The generated HDL code runs at two effective rates: a faster rate for internal computations, and a slower rate for input/output. A special timing controller entity (`vsum_tc`) generates these rates from a single master clock using counters and multiple clock enables. The `vsum_tc` entity definition is written to a separate code file.

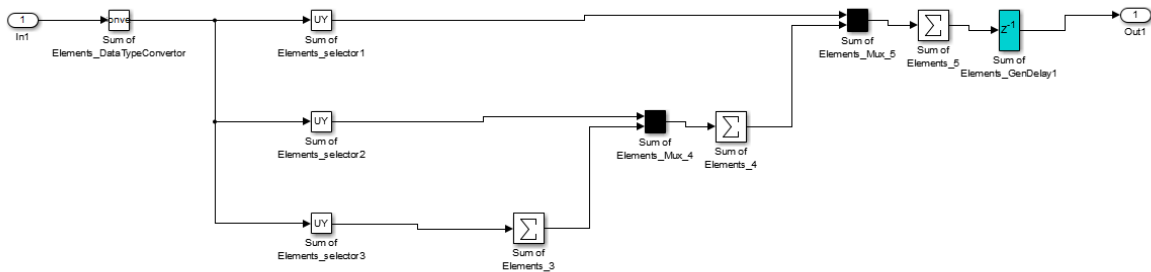
The generated model looks like this:



The subsystem is highlighted in cyan. This highlighting indicates that the subsystem differs in some respect from the `vsum` subsystem of the original model.

The following block diagram shows the `vsum` subsystem in the generated model. The subsystem has been restructured to reflect the structure of the generated HDL code; inputs are grouped and routed to three adders for partial sum computations.

A Unit Delay (highlighted in cyan) has been inserted before the final output. This block delays (in this case, for one sample period) the appearance of the final sum at the output. The delay reflects the latency of the generated HDL code.



Note The HDL code generated from the example model used in this section is bit-true to the original model.

However, in some cases, cascaded block implementations can produce numeric differences between the original model and the generated HDL code, in addition to the introduction of latency. Numeric differences can arise from saturation and rounding operations.

Defaults and Options for Generated Models

In this section...
“Defaults for Model Generation” on page 14-12
“GUI Options” on page 14-13
“Generated Model Properties for makehdl” on page 14-15

Defaults for Model Generation

This section summarizes the defaults that the coder uses when building generated models.

Model Generation

The coder creates a generated model as part of the code generation process. The coder builds the generated model in memory, before actual generation of HDL code. The HDL code and the generated model are bit-true and cycle-accurate with respect to one another.

Note The in-memory generated model is not written to a model file unless you explicitly save it.

Naming of Generated Models

The naming convention for generated models is:

```
prefix_modelName
```

where the default prefix is `gm_`, and the default `modelName` is the name of the original model.

If code is generated more than once from the same original model, and previously generated models exist in memory, an integer is suffixed to the name of each successively generated model. The suffix provides a unique name for each generated model. For example, if the original model is named `test`, generated models will be named `gm_test`, `gm_test0`, `gm_test1`, etc.

Note When regenerating code from your models, be sure to select the original model for code generation, not a previously generated model. Generating code from a generated model might introduce unintended delays or numeric differences that make the model operate incorrectly.

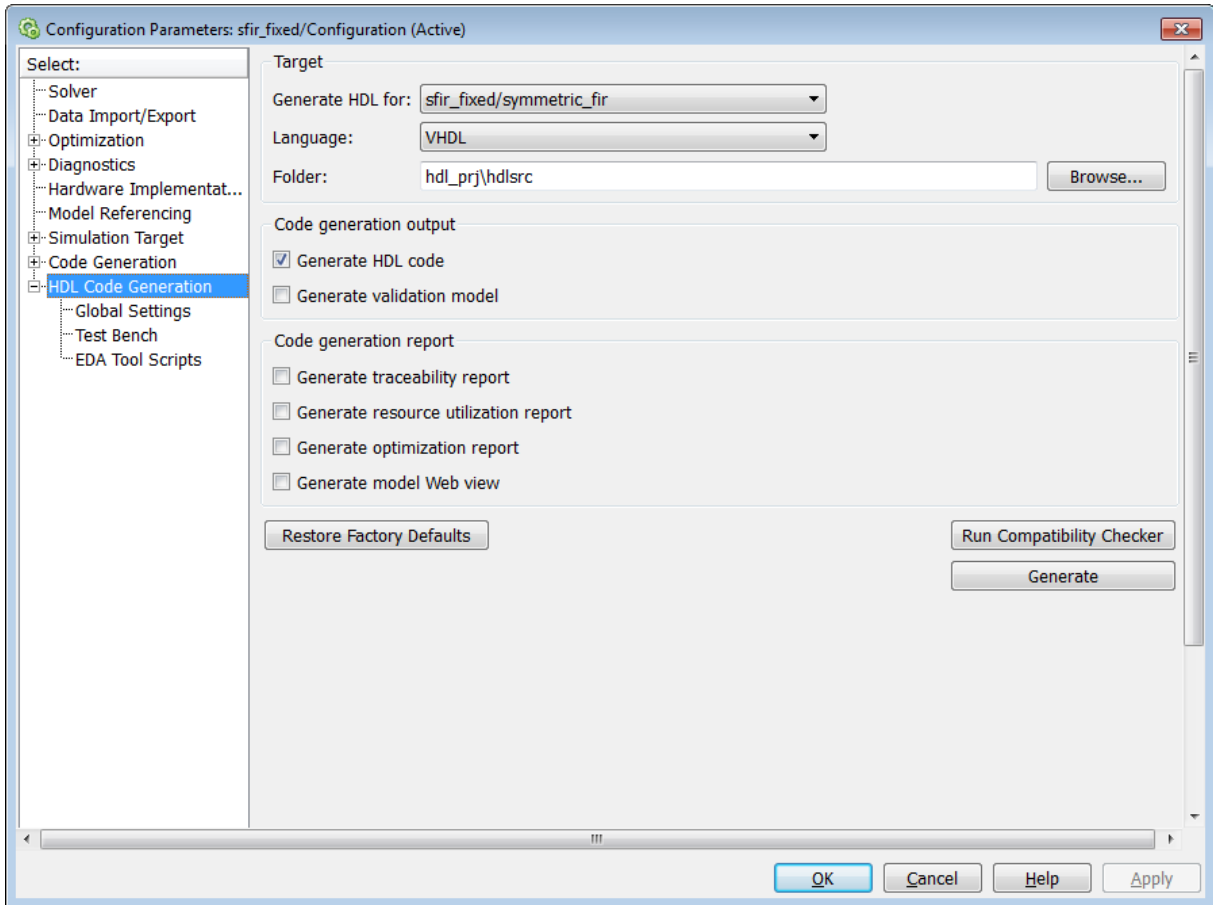
Block Highlighting

By default, blocks in a generated model that differ from the original model, and their ancestor (parent) blocks in the model hierarchy, are highlighted in the default color, cyan. You can quickly see whether differences have been introduced, by examining the root level of the generated model.

If the original and generated models match, no blocks appear highlighted.

GUI Options

The HDL Coder GUI provides high-level options controlling the generation and display of generated models. More detailed control is available through the `makehdl` command (see “Generated Model Properties for `makehdl`” on page 14-15). Generated model options are located in the top-level **HDL Code Generation** pane of the Configuration Parameters dialog box:



Options include:

- **Generate HDL code:** Generate HDL code for the device under test (DUT). By default, this check box is selected.
- **Generate validation model:** Generate a validation model to verify functional equivalence of the generated model with the original model. By default, this check box is cleared.

Generated Model Properties for makehdl

The following summary describes makehdl properties that provide detailed controls for the generated model.

Property and Value(s)	Description
'GeneratedmodelNameprefix', ['string']	The default name for the generated model is gm_modelname, where gm_ is the default prefix and modelname is the original model name. To override the default prefix, assign a string value to this property.
'Generatemodelname', ['string']	By default, the original model name is used as the modelname substring of the generated model name. To specify a different model name, assign a string value to this property.
'CodeGenerationOutput', 'string'	Controls the production of generated code and display of the generated model. Values are: <ul style="list-style-type: none"> • GenerateHDLCode: (Default) Generate code, but do not display the generated model. • GenerateHDLCodeAndDisplayGeneratedModel: Create and display generated model, but do not proceed to code generation. • DisplayGeneratedModelOnly: Generate both code and model, and display model when completed.
'GenerateHDLCode', ['on' 'off']	Controls whether or not to generate HDL code for the DUT. The default is 'on'.
'GenerateValidationModel', ['on' 'off']	Controls whether or not to generate a validation model. The default is 'off'.

Property and Value(s)	Description
'Highlightancestors', ['on' 'off']	By default, blocks in a generated model that differ from the original model, and their ancestor (parent) blocks in the model hierarchy, are highlighted in a color specified by the <code>Highlightcolor</code> property. If you do not want the ancestor blocks to be highlighted, set this property to 'off'.
'Highlightcolor', 'RGBName'	Specify the color used to highlight blocks in a generated model that differ from the original model (default: cyan). Specify the color (RGBName) as one of the following color string values: <ul data-bbox="690 668 897 1032" style="list-style-type: none">• cyan (default)• yellow• magenta• red• green• blue• white• black

Limitations for Generated Models

In this section...

“Fixed-Point Limitation” on page 14-17

“Double-Precision Limitation” on page 14-17

“Model Properties Not Supported for Generated Models” on page 14-18

Fixed-Point Limitation

The maximum Simulink fixed-point word size is 128 bits. HDL does not have such a limit. This can lead to cases in which the generated HDL code is not bit-true to the generated model.

When the result of a computation in the generated HDL code has a word size greater than 128 bits:

- The coder issues a warning.
- Computations in the generated model (and the generated HDL test bench) are limited to a result word size of 128 bits.
- This word size limitation does not apply to the generated HDL code, so results returned from the HDL code may not match the HDL test bench or the generated model.

Double-Precision Limitation

When the binary point in double-precision computations is very large or very small, the scaling can become `inf` or `0`. The limits of precision can be expressed as follows:

```
log2(realmin) ==> -1022
```

```
log2(realmax) ==> 1024
```

Where these limits are exceeded, the binary point is saturated and a warning is issued. If the generated HDL code has binary point scaling greater than 2^{1024} , the generated model has a maximum scaling of 2^{1024} .

Similarly if the generated HDL code has binary point scaling smaller than 2^{-1022} , then the generated model has scaling of 2^{-1022} .

Model Properties Not Supported for Generated Models

The coder disables the following model parameters during code generation, and restores them after code generation completes:

- **Block Reduction** (BlockReductionOpt)
- **Conditional input branch execution** (ConditionallyExecuteInputs)

These properties are disabled in the generated model, even if they are enabled in the source model.

Optimization

- “Optimization With Constrained Overclocking” on page 15-2
- “Maximum Oversampling Ratio” on page 15-5
- “Maximum Computation Latency” on page 15-7
- “Streaming” on page 15-9
- “Area Reduction with Streaming” on page 15-13
- “What Is the Validation Model?” on page 15-24
- “Resource Sharing” on page 15-25
- “Check Compatibility for Resource Sharing” on page 15-30
- “Delay Balancing” on page 15-31
- “Resolve Numerical Mismatch with Delay Balancing” on page 15-34
- “Hierarchy Flattening” on page 15-38
- “Loop Optimization” on page 15-41
- “Optimize Loops in the MATLAB Function Block” on page 15-42
- “RAM Mapping” on page 15-44
- “RAM Mapping with the MATLAB Function Block” on page 15-45
- “Insert Distributed Pipeline Registers in a Subsystem” on page 15-48
- “Distributed Pipelining and Hierarchical Distributed Pipelining” on page 15-53
- “Constrained Output Pipelining” on page 15-63
- “Pipeline Variables in the MATLAB Function Block” on page 15-65
- “Reduce Critical Path With Distributed Pipelining” on page 15-67

Optimization With Constrained Overclocking

In this section...

“Why Constrain Overclocking?” on page 15-2

“When to Use Constrained Overclocking” on page 15-2

“Set Overclocking Constraints” on page 15-3

“Constrained Overclocking Limitations” on page 15-4

Why Constrain Overclocking?

Overclocking can cause your design clock rate to exceed the maximum clock rate of your target hardware when your original design’s 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 you have a design that does not currently fit in the target hardware, but is already running at the target device’s maximum clock frequency, and you know 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 the coder a latency budget of N cycles to perform the computation. In this situation, the 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” on page 15-5.

To learn more about the **Max computation latency** parameter, see “Maximum Computation Latency” on page 15-7.

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 > 1
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” on page 15-5

- “Specify Maximum Computation Latency” on page 15-8

Constrained Overclocking Limitations

When you constrain overclocking, the following limitations apply:

- Your DUT must be single-rate if you set **Max oversampling** = 1.
- 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 15-5

“Specify Maximum Oversampling Ratio” on page 15-5

“Maximum Oversampling Ratio Limitations” on page 15-6

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 15-7

“Specify Maximum Computation Latency” on page 15-8

“Maximum Computation Latency Restrictions” on page 15-8

What Is Maximum Computation Latency?

The **Max computation latency** parameter enables you to specify a time budget for the coder when performing a single computation. Within this time budget, the 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.
$N > 1$	<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. Note that 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 the 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 the 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 15-9

“Specify Streaming” on page 15-10

“Requirements and Limitations for Streaming” on page 15-10

What is Streaming?

Streaming is an area optimization in which the 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 a large number of 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, the coder implements an entirely scalar data path. A streaming factor of 0 (the default) produces a fully parallel implementation (i.e., without sharing) for vector computations. Depending on the width of the data path, you can also specify streaming factors between these extrema.

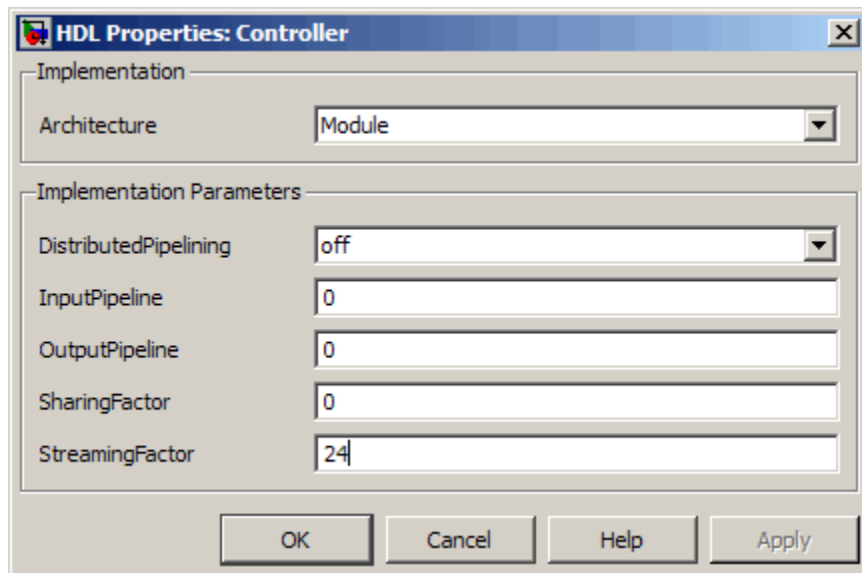
If you know the maximal vector dimensions and the sample rate for a subsystem, you can compute the possible streaming factors and resulting sample rates for the subsystem. However, even if the requested streaming factor is mathematically possible, the subsystem must meet other criteria for streaming. See “Requirements and Limitations for Streaming” on page 15-10 for details.

By default, when you apply the streaming optimization, the coder oversamples the shared hardware resource in order 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” on page 15-2.

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 “What Is the Validation Model?” on page 15-24.

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

The coder supports a large number of 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.

The 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, the 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. The coder supports a large number of 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 will fail.

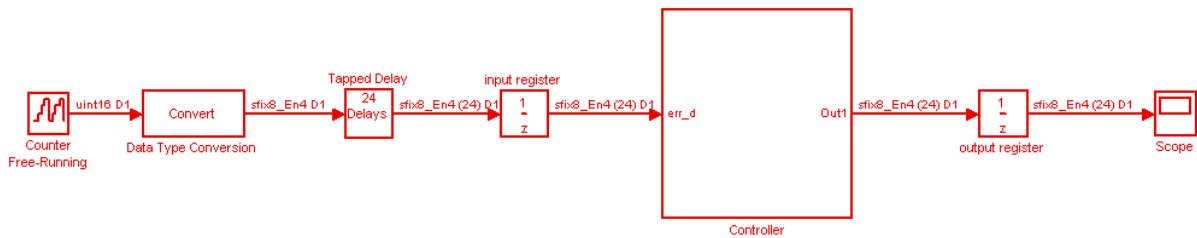
If the requested streaming factor cannot be implemented, the coder generates non-streaming code. It is good practice to generate an Optimization Report. The Streaming and Sharing page of the report provides information about conditions that prevent streaming.

Area Reduction with Streaming

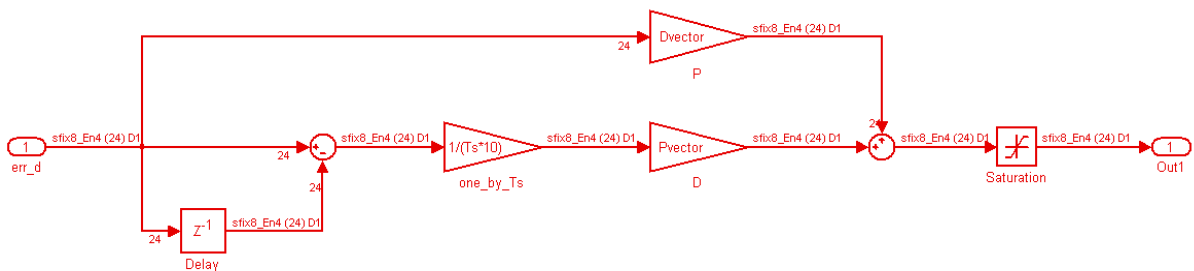
This example illustrates:

- 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 coder 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, the 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_vn1.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_serialcomp
    as hdlsrc\Saturation_out1_serialcomp.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 the 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

The coder also produces a Streaming and Sharing report that shows:

- The `StreamingFactor` 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

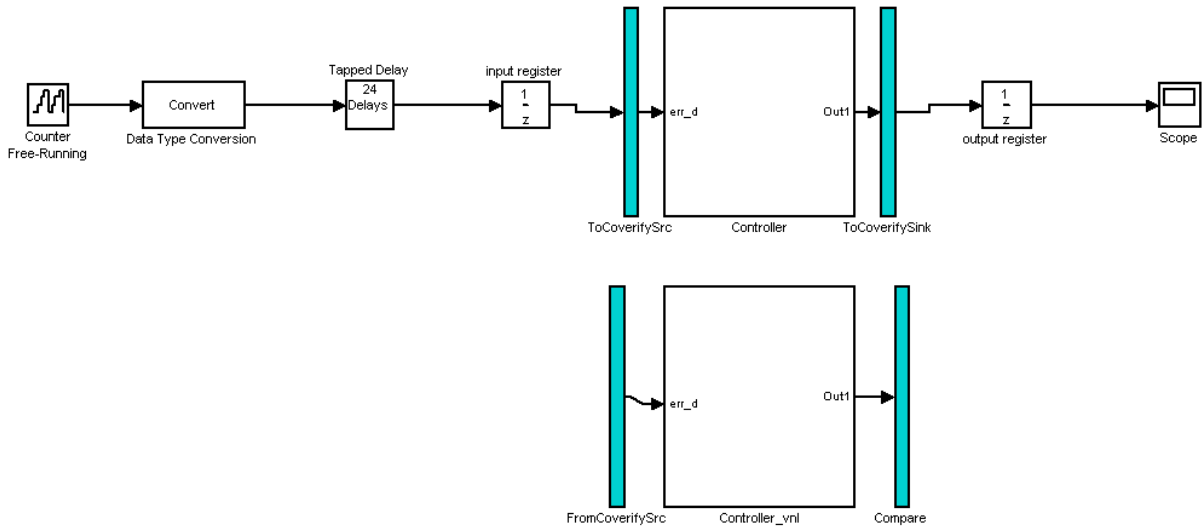
Path Delay Summary

Port	Path Delay
Controller/ce_out	1
Controller/Out1	1

Validation model: [gm_ex_pdcontroller_multi_instance4_vnl](#)

The Validation Model

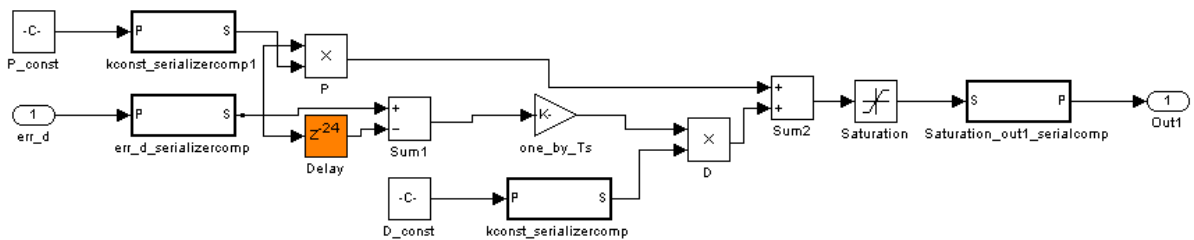
The following figure shows the validation model generated for the Controller subsystem.



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.

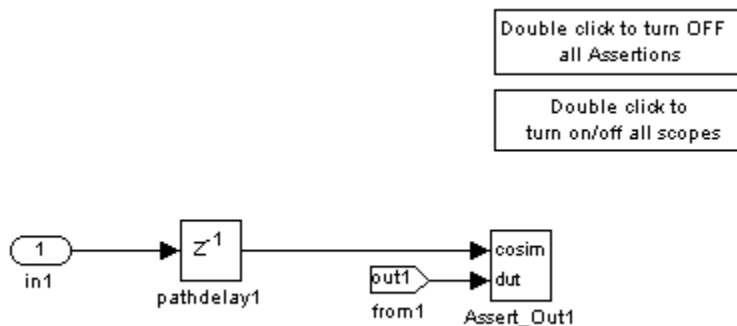
The following figure shows the interior of the Controller subsystem.



Inspection of the Controller subsystem shows that it is a multirate subsystem, having two rates that operate as follows:

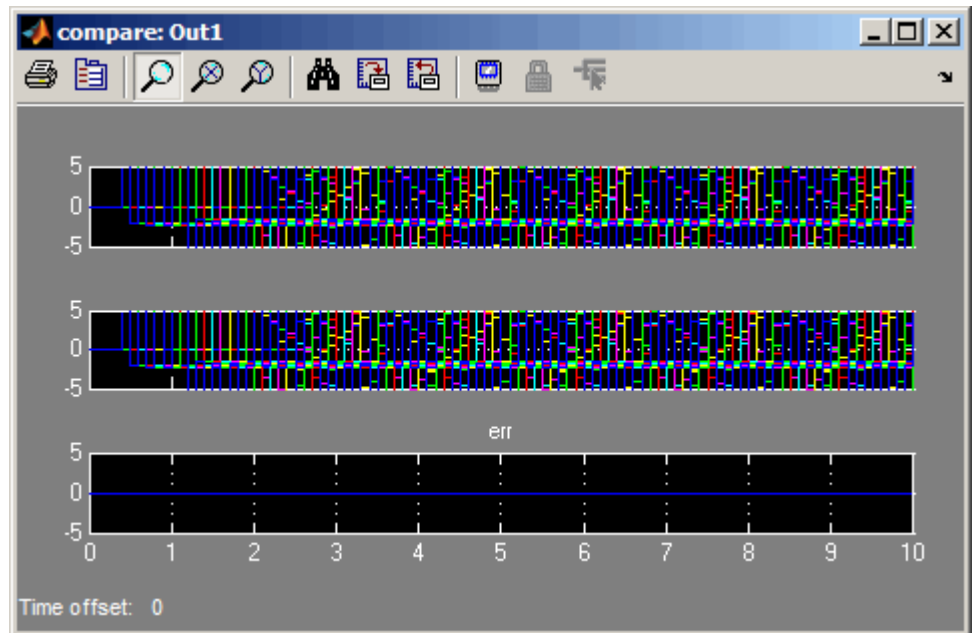
- Inputs and outputs run at the same rate as the exterior model.
- Dual-rate Serializer blocks receive vector data at the original rate and output a stream of scalar values at the higher (24x) rate.
- Interior blocks between Serializers and Deserializer run at the higher rate.
- The Deserializer block 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 should show a line through zero, indicating that the data comparisons were equal.



What Is the Validation Model?

Before generating code, the 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.

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.

Resource Sharing

In this section...
“What Is Resource Sharing?” on page 15-25
“Benefits and Costs of Resource Sharing” on page 15-26
“Specify Resource Sharing” on page 15-26
“Requirements for Resource Sharing” on page 15-27
“Resource Sharing Information in Reports” on page 15-29

What Is Resource Sharing?

Resource sharing is an area optimization in which the coder identifies multiple functionally equivalent resources within a subsystem or MATLAB Function block and replaces them with a single resource. The coder time-multiplexes the data over the shared resource to perform the same operations.

The *sharing factor* for a subsystem is the number of blocks that can share a single resource.

If you specify a nonzero sharing factor for a MATLAB Function block, the coder tries to identify and share that number of functionally equivalent multipliers.

If you specify a nonzero sharing factor for a subsystem, the coder tries to identify and share that number of functionally equivalent instances of the following types of blocks:

- Gain
- Product
- Atomic Subsystem
- MATLAB Function
- Model reference

By default, when you apply the sharing optimization, the coder oversamples the shared hardware resource in order 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” on page 15-2.

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 “What Is the Validation Model?” on page 15-24.

Benefits and Costs of Resource Sharing

Resource sharing can substantially reduce your chip area. For example, the generated code may 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 may use more registers.
- Reduces opportunities for distributed pipelining or retiming, because the coder does not pipeline across clock rate boundaries.

Specify Resource Sharing

To open an example showing how to use resource sharing, enter:

```
hdlcoder_sharing_optimization
```

Specify Resource Sharing from the UI

To specify resource sharing from the UI:

- 1** Right-click the subsystem 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 for Resource Sharing

On a Subsystem block, model reference, or MATLAB Function block, you can specify the resource sharing optimization.

Blocks to be shared within a subsystem 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 within a subsystem:

- The only state elements that these blocks can contain are Unit Delay and Delay blocks. Unit Delay and Delay blocks must have the **Initial condition** parameter set to 0.
- These blocks must not contain enabled or triggered subsystems.
- These blocks must not contain a subsystem that does not meet the requirements for resource sharing.

If you want to share MATLAB Function blocks within a subsystem, they must not use:

- Persistent variables
- Loop streaming
- Output pipelining

If you want to share model reference instances within a subsystem, all model references that point to the same submodel must have the same rate after optimizations and rate propagation. The model reference final rate may differ from the original rate, but all model references that point to the same submodel 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	The same checksum. Atomic subsystems must be identical to be shared.

Limitations for Atomic Subsystem Sharing

The coder cannot apply resource sharing to atomic subsystems that contain state elements other than the Delay and Unit Delay blocks. Therefore, you cannot share atomic subsystems that contain the following blocks or block implementations:

- Detect Change
- Discrete Transfer Fcn
- Enabled Subsystem
- HDL FFT
- HDL FIFO
- Math Function (conj, hermitian, transpose)
- MATLAB Function blocks that contain persistent variables
- Sqrt
- Triggered Subsystem
- Unit Delay Resettable
- Unit Delay Enabled Resettable
- 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” on page 15-33.

Resource Sharing Information in Reports

If you generate a code generation report, for each subsystem 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.
- Recommendations for other `SharingFactor` values that you could try.

For example, if you see `Other possible factors: [2 .. 16]` in the report, try setting `SharingFactor` to 2 or to 16. The coder determines its suggestions for other possible sharing factors based on the number of equivalent blocks in your design. However, the suggested sharing factors may not work because of the location of these equivalent blocks in your design.

Check Compatibility for Resource Sharing

To determine whether or not 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 for Resource Sharing” on page 15-27.

Delay Balancing

In this section...
“Why Use Delay Balancing” on page 15-31
“Specify Delay Balancing” on page 15-32
“Delay Balancing Limitations” on page 15-33

Why Use Delay Balancing

The coder supports several optimizations, block implementations, and options that introduce discrete delays into the model, with the goal of more efficient hardware usage or achieving higher clock rates. Examples include:

- *Optimizations*: Optimizations such as output pipelining, streaming, or resource sharing can introduce delays.
- *Cascading*: Some blocks support cascade implementations, which introduce a cycle of delay in the generated code.
- *Block implementations*: Some block implementations inherently introduce delays in the generated code. “Resolve Numerical Mismatch with Delay Balancing” on page 15-34 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 may 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 very large models with many signal paths or multiple sample rates.

To help you solve this problem, the coder supports *delay balancing*. When you enable delay balancing, if the coder detects introduction of new delays along one path, it 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 “What Is the Validation Model?” on page 15-24.

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, you must disable delay balancing at the model level. Note that 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 may therefore show mismatches between the original model and generated model.

To disable delay balancing for a subsystem within the top-level DUT subsystem:

- 1 Disable delay balancing for the model.
- 2 Enable delay balancing for the top-level DUT subsystem.
- 3 Disable delay balancing for a subsystem within the DUT subsystem.

To learn how to set delay balancing for a subsystem, see “Set Delay Balancing For a Subsystem” on page 11-51.

Delay Balancing Limitations

The following blocks do not support delay balancing:

- Cosimulation
- Data Type Duplicate
- Decrement To Zero
- Enable Port
- Frame Conversion
- Ground
- HDL FFT
- LMS Filter
- Model Reference
- To VCD File
- Trigger Port
- Magnitude-Angle to Complex

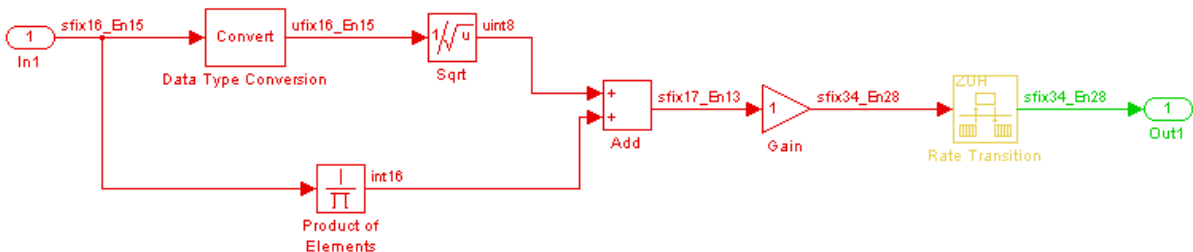
The following block implementations do not currently support delay balancing:

- `hdldefaults.ConstantSpecialHDL Emission`
- `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.

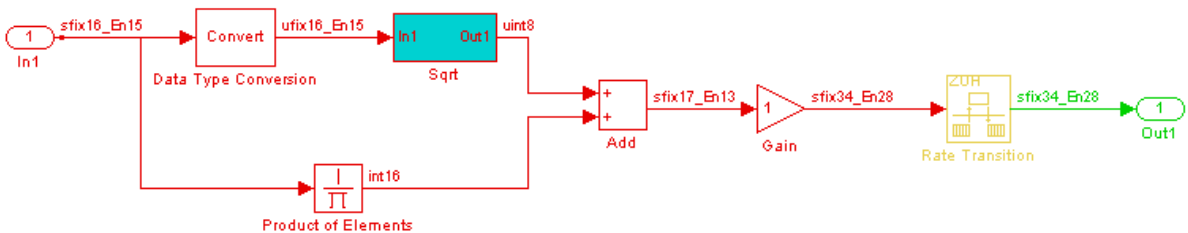
The following figure shows the DUT for the `ex_rsqrtdelaybalancing` model. The DUT is a simple multirate subsystem that includes a Reciprocal Square Root block (`Sqrt`). A Rate Transition block downsamples the output signal to a lower sample rate.



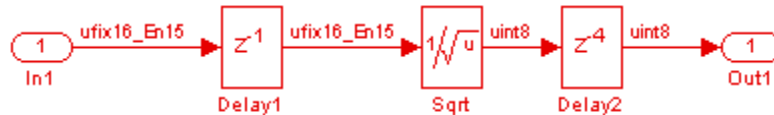
Generate HDL code without delay balancing and generate a validation model:

```
dut = 'ex_rsqrtdelaybalancing/Subsystem';
makehdl(dut, 'BalanceDelays', 'off', 'GenerateValidationModel', 'on');
```

Examination of the generated model shows that the coder has implemented the `Sqrt` block as a subsystem:



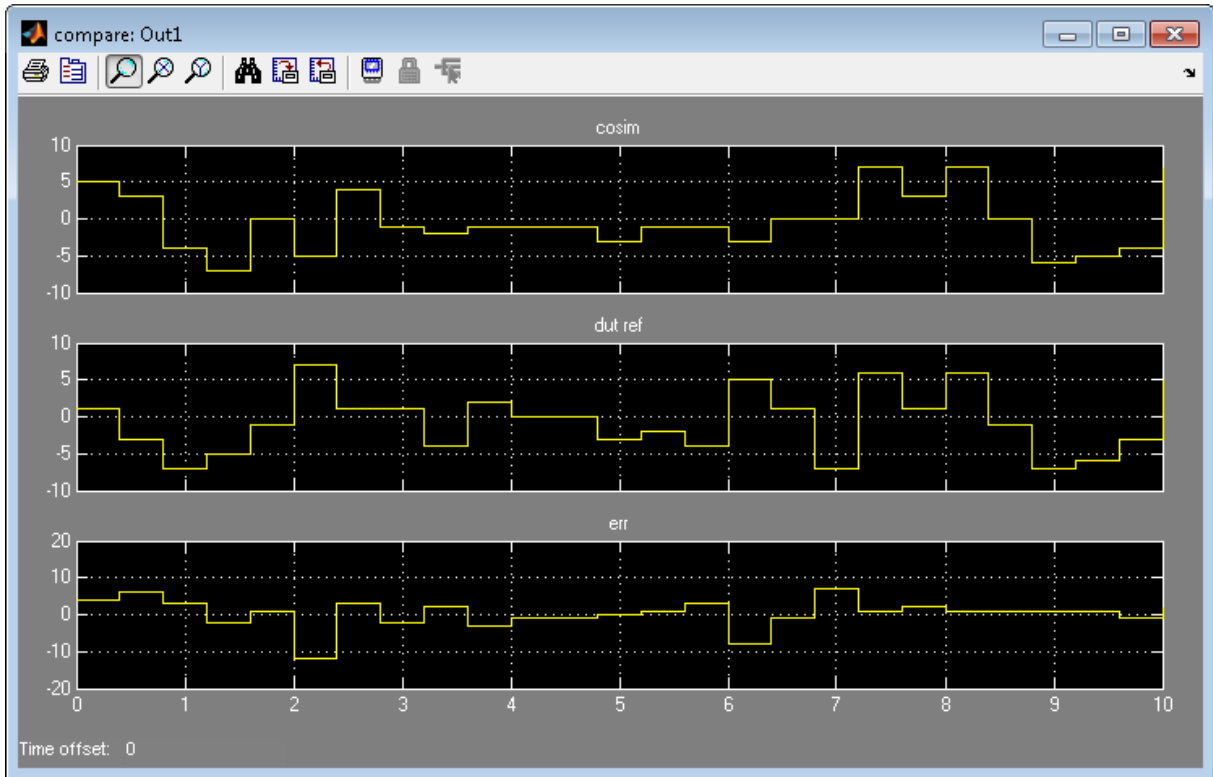
The following figure shows that the generated Sqrt subsystem introduces a total of 5 cycles of delay. (This behavior is inherent to the Reciprocal Square Root block implementation.) These delays map to registers in the generated HDL code when **UseRAM** is off.



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:

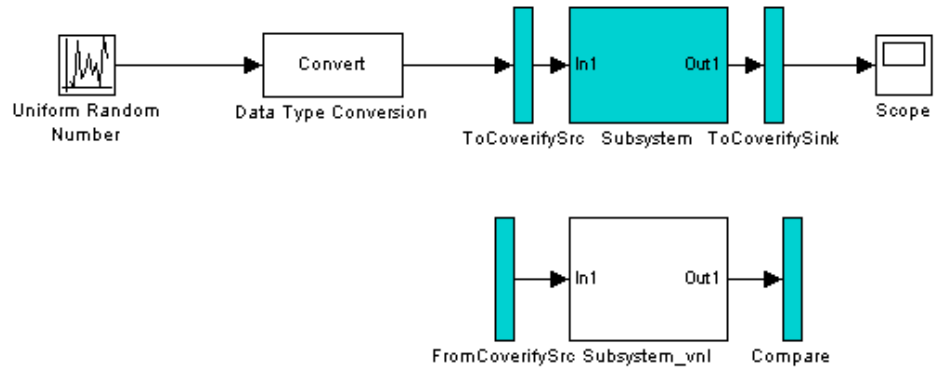
- The input signal branches into two parallel paths (to the `Sqrt` and product blocks) but only the branch to the `Sqrt` block introduces delays.
- The downsampling caused by the rate transition drops samples.

You can solve these problems by manually inserting delays in the generated model. However, using the coder's delay balancing capability produces more consistent results.

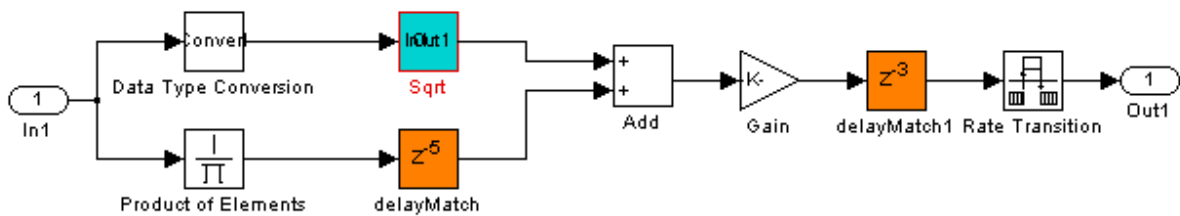
Generate HDL code with delay balancing and generate a validation model:

```
dut = 'ex_rsqrtdelaybalancing/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, the 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.



Hierarchy Flattening

In this section...

“What Is Hierarchy Flattening?” on page 15-38

“When To Flatten Hierarchy” on page 15-38

“Prerequisites For Hierarchy Flattening” on page 15-38

“Options For Hierarchy Flattening” on page 15-39

“How To Flatten Hierarchy” on page 15-39

“Limitations For Hierarchy Flattening” on page 15-40

What Is Hierarchy Flattening?

Hierarchy flattening enables you to remove subsystem hierarchy from the HDL code generated from your design.

The coder considers blocks within a flattened subsystem to be at the same level of hierarchy, and no longer grouped into separate subsystems. This consideration allows the coder to reorganize blocks for optimization across the original hierarchical boundaries, while preserving functionality.

When To Flatten Hierarchy

Flatten hierarchy to:

- Enable more extensive area and speed optimization.
- Reduce the number of HDL output files. For every subsystem flattened, the 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.
- An enabled or triggered subsystem.
- A masked subsystem.

Note This option removes subsystem boundaries before code generation. It does not necessarily generate HDL code with a completely flat hierarchy.

Loop Optimization

In this section...
“Loop Streaming” on page 15-41
“Loop Unrolling” on page 15-41

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

Loop Streaming

The coder streams a loop by instantiating the loop body once and using that instance for each loop iteration.

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

Loop Unrolling

The coder unrolls a loop by instantiating multiple instances of the loop body in the generated code.

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.

Optimize Loops in the MATLAB Function Block

In this section...

“MATLAB Function Block Loop Optimization Options” on page 15-42

“How to Optimize MATLAB Function Block Loops” on page 15-42

“Limitations for MATLAB Function Block Loop Optimization” on page 15-43

MATLAB Function Block Loop Optimization Options

The loop optimization options for a MATLAB Function block are listed in the following table.

Loop Optimization Setting	Description
none (default)	Do not optimize loops.
Unrolling	Unroll loops.
Streaming	Stream loops.

How to Optimize MATLAB Function Block 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

The 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.

The coder can stream the loop when the persistent variable is:

- Updated inside the loop and read outside the loop.
- Read within the loop and updated outside the loop.

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” on page 11-90.
- `MapPersistentVarsToRAM` to map persistent arrays in a MATLAB Function block to RAM. For details, see “MapPersistentVarsToRAM” on page 11-77.
- RAM blocks from the `hdldemo1ib` library. For details, see “RAM Blocks” on page 13-3.
- Blocks with a RAM implementation. For details, see “RAM” on page 11-85.

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

Note that 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

Note that 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” on page 20-23
- “MapPersistentVarsToRAM” on page 11-77

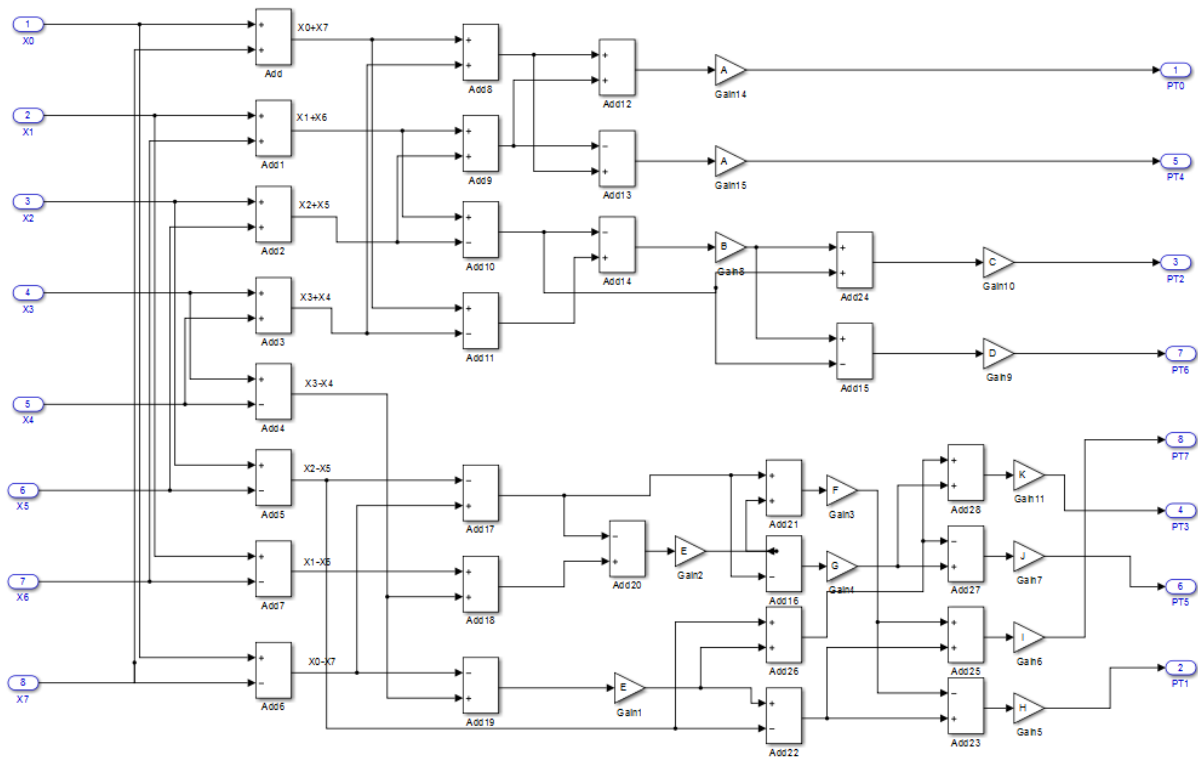
Insert Distributed Pipeline Registers in a Subsystem

This example shows how to use distributed pipelining with the `dct8_fixed` model.

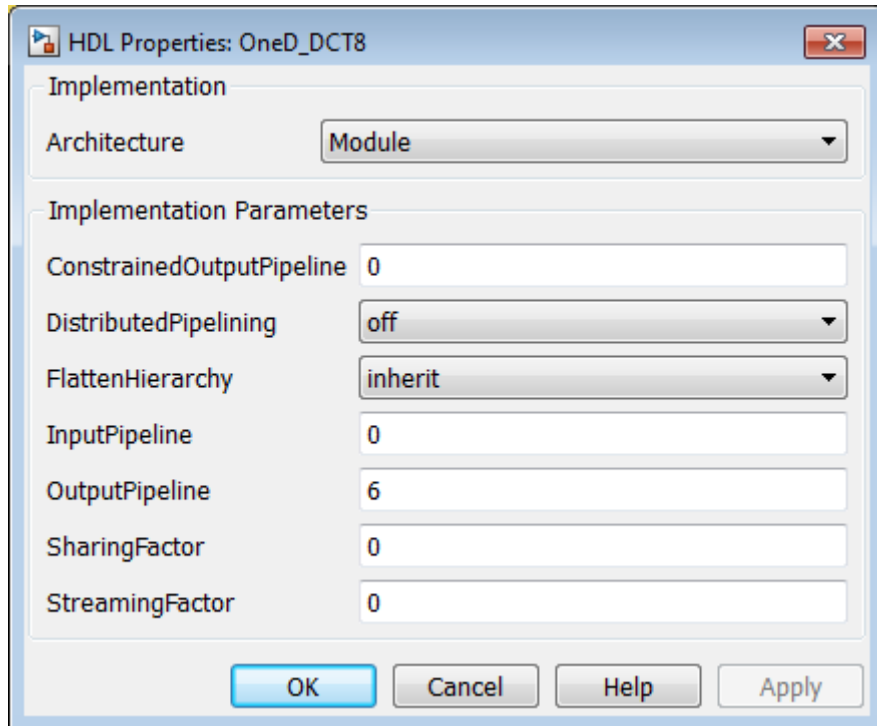
This example uses the following optimizations:

- Output pipelining
- Distributed pipelining

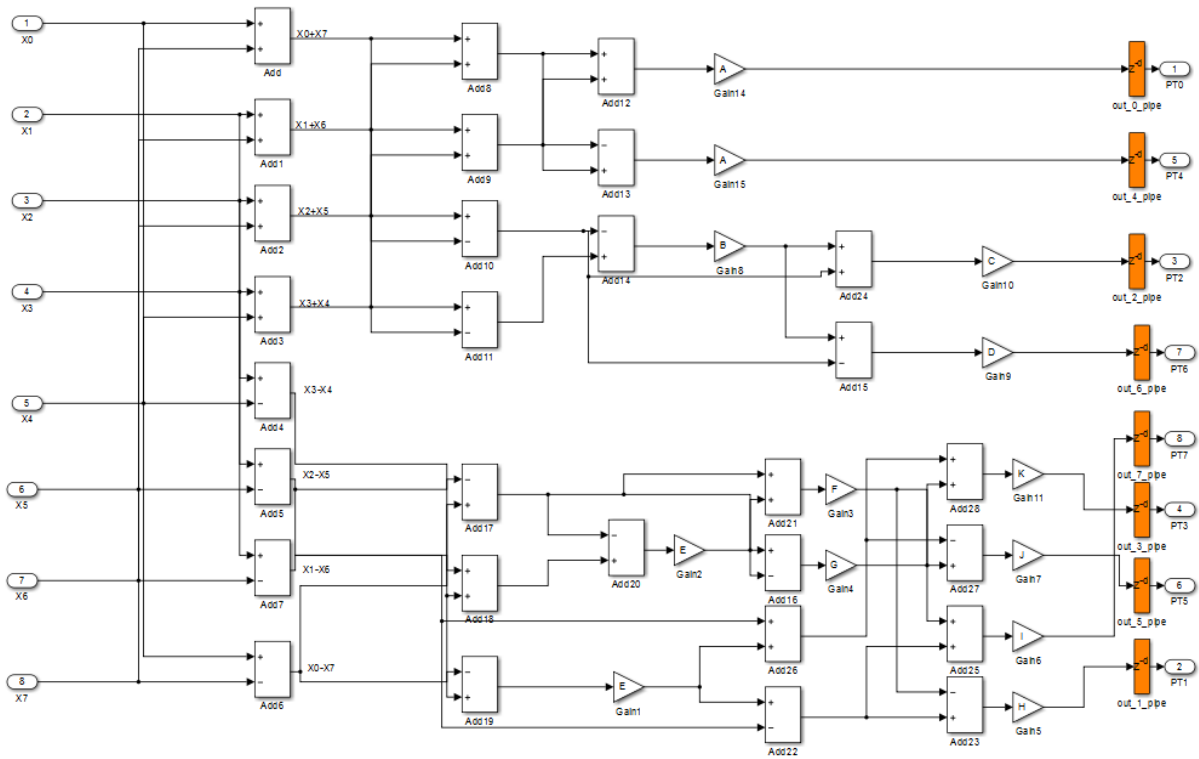
Open the model by typing `dct8_fixed` at the MATLAB prompt. The DUT is the `dct8_fixed/OneD_DCT8` subsystem.



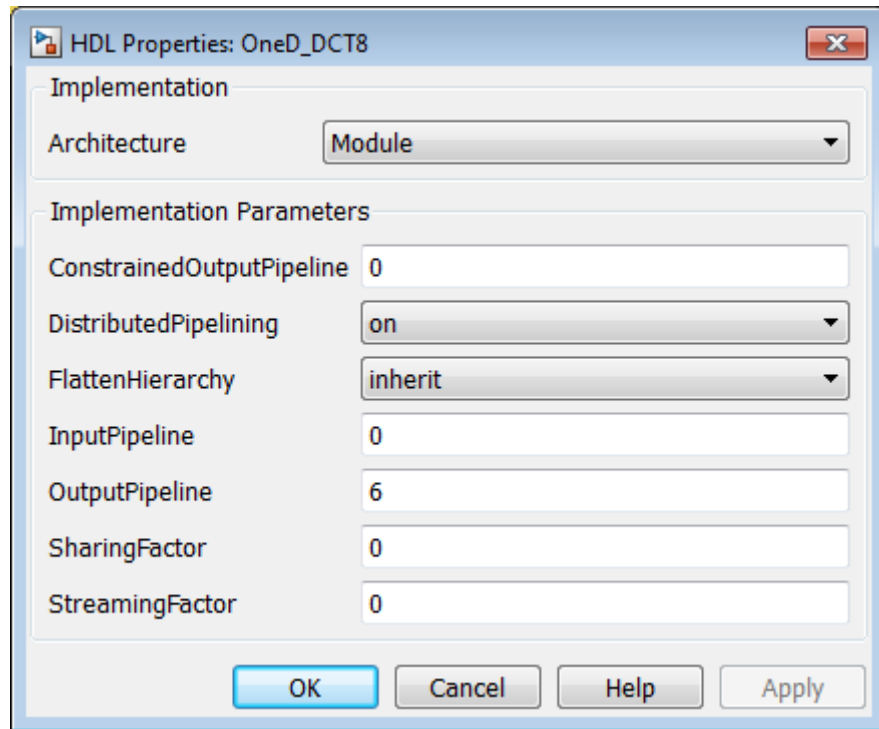
Set **DistributedPipelining** to off and **OutputPipeline** to 6 to insert 6 pipeline stages at the outputs of the DUT.



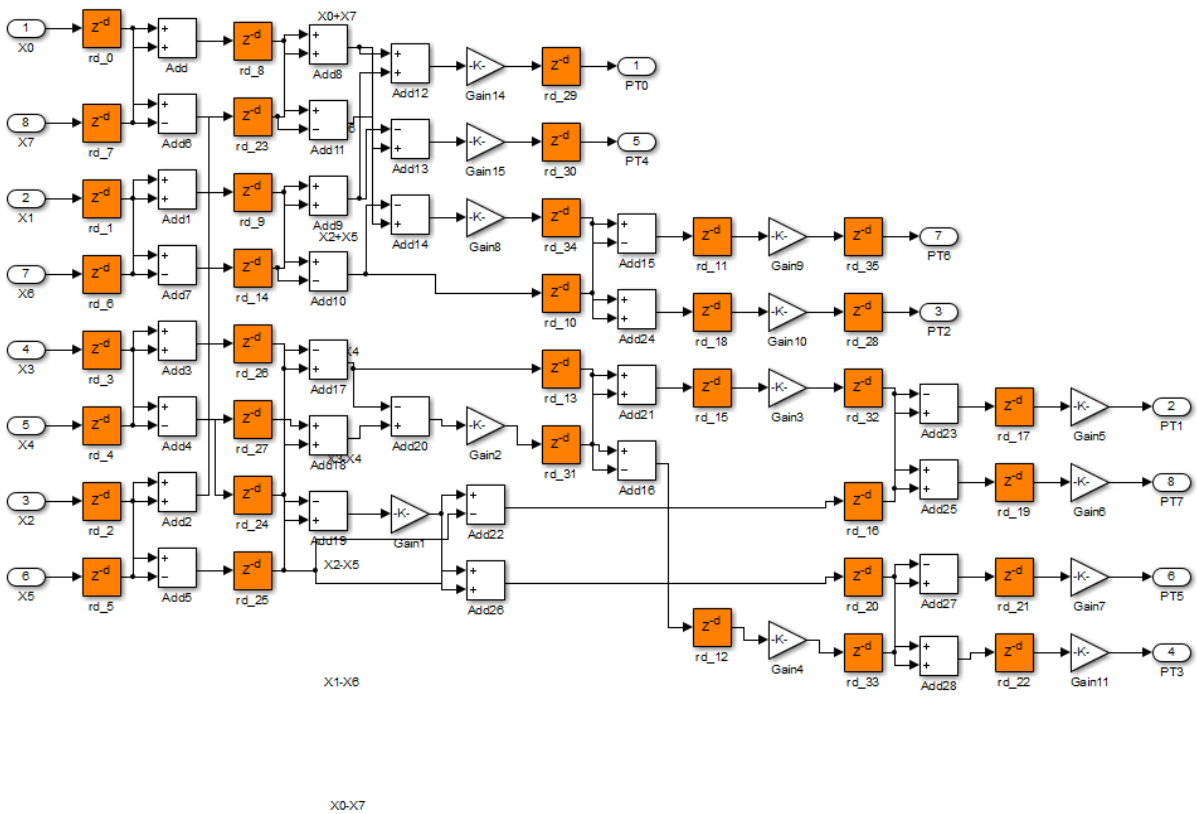
The generated model shows the placement of pipeline registers as highlighted delays at the outputs of the DUT. For more information about generated models, see “What Is the Generated Model?” on page 14-2.



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 6 pipeline registers for each path.



Distributed Pipelining and Hierarchical Distributed Pipelining

In this section...

“What is Distributed Pipelining?” on page 15-53

“Benefits and Costs of Distributed Pipelining” on page 15-55

“Requirements for Distributed Pipelining” on page 15-56

“Specify Distributed Pipelining” on page 15-56

“Limitations of Distributed Pipelining” on page 15-57

“What is Hierarchical Distributed Pipelining?” on page 15-59

“Benefits of Hierarchical Distributed Pipelining” on page 15-61

“Specify Hierarchical Distributed Pipelining” on page 15-61

“Limitations of Hierarchical Distributed Pipelining” on page 15-62

“Distributed Pipelining Workflow” on page 15-62

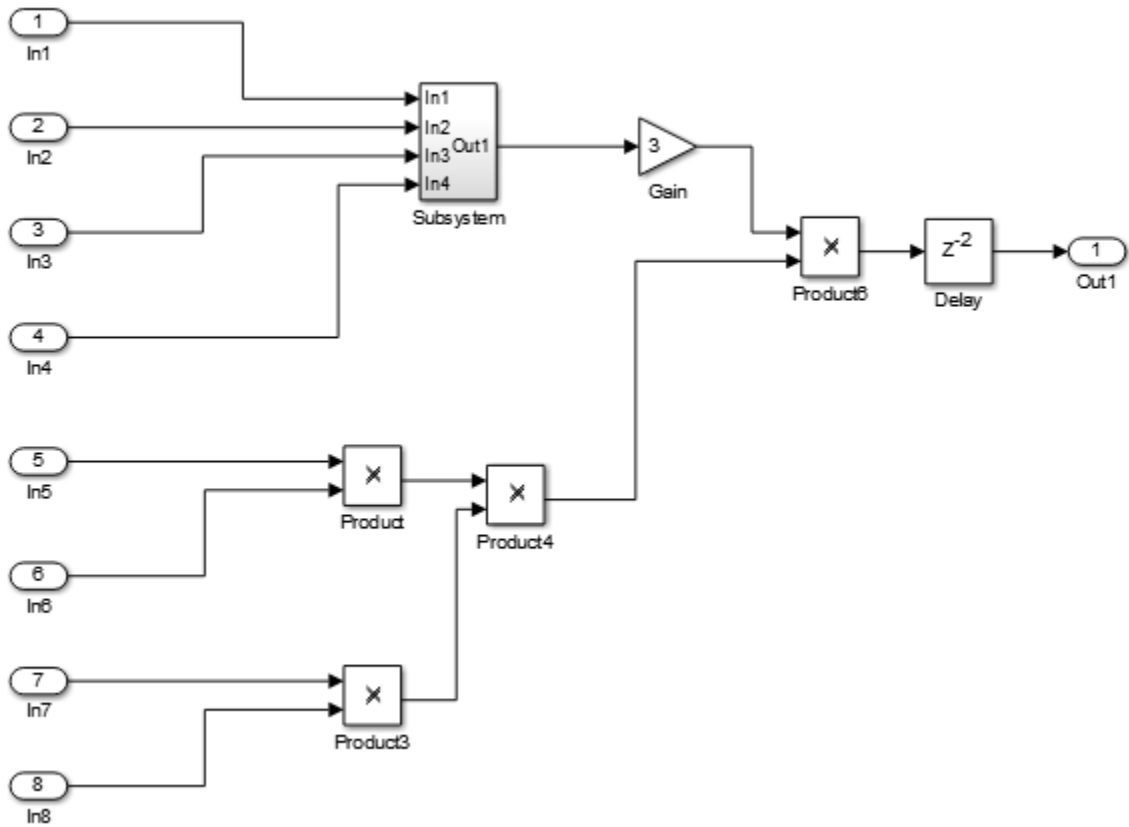
“Selected Bibliography” on page 15-62

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 coder 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.

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” on page 20-42.
- 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 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:

- “Use file I/O to read/write test bench data” on page 9-103
 - 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 coder generates pipeline registers at the outputs of 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.
- The 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)

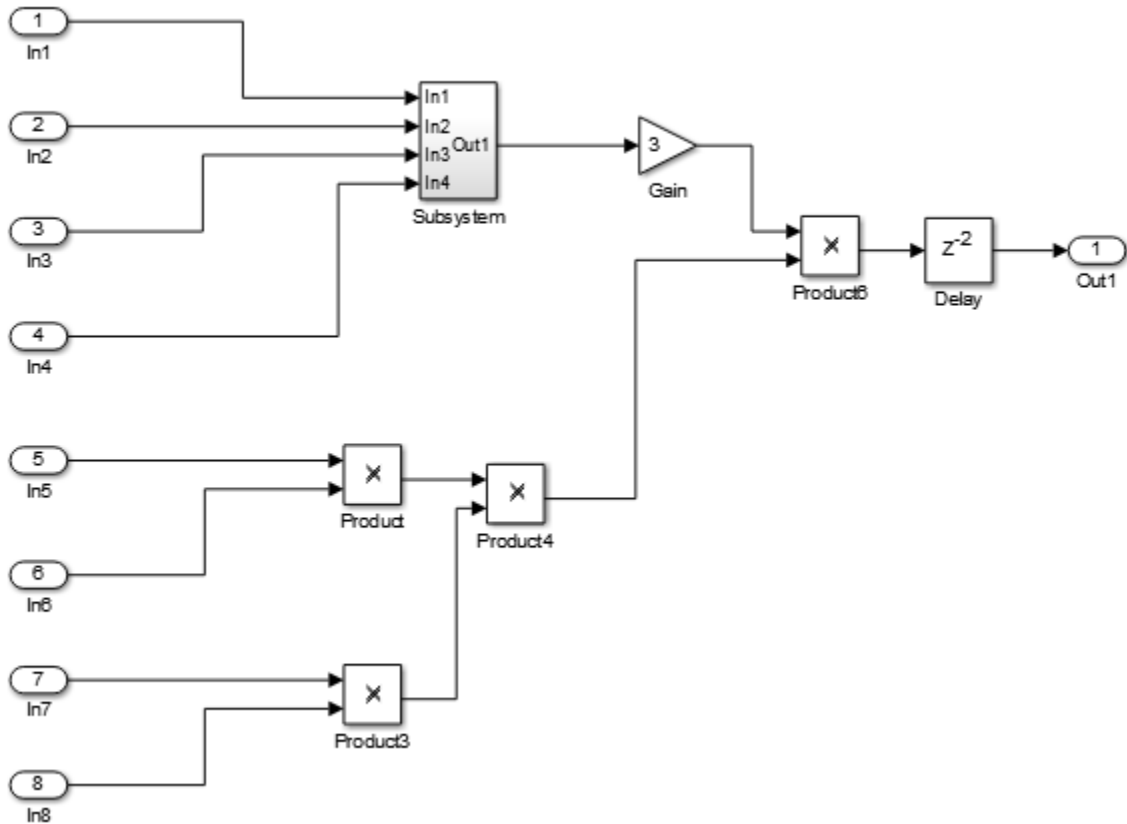
- Single Port RAM
- Dual Port RAM
- Simple Dual Port RAM
- If you enable distributed pipelining for a subsystem that contains blocks in the following list, the coder issues an error message and terminates code generation. To enable code generation to proceed, place these blocks inside one or more subsystems within the original subsystem and disable hierarchical distributed pipelining. The coder will distribute 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
 - dspsigops/Repeat
 - HDL Counter
 - dspadpt3/LMS Filter
 - dspsrcs4/Sine Wave
 - commcnvcod2/Viterbi Decoder
 - Triggered Subsystem
 - Counter Limited
 - Counter Free-Running
 - Frame Conversion

What is Hierarchical Distributed Pipelining?

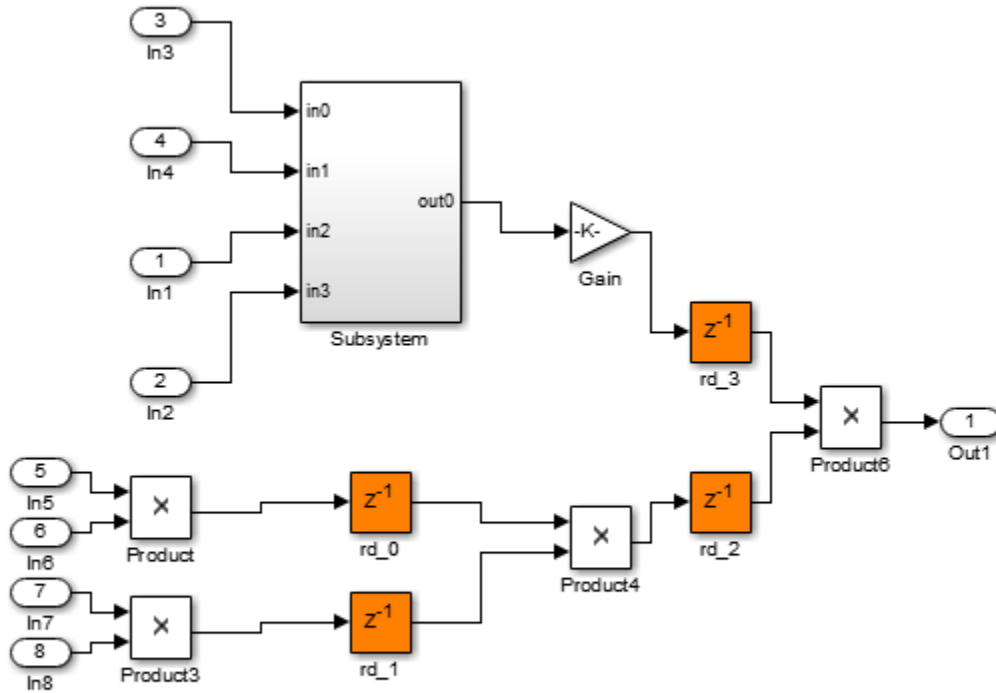
Hierarchical distributed pipelining extends the scope of distributed pipelining by moving delays across hierarchical boundaries within a subsystem while preserving subsystem hierarchy.

If a subsystem in the hierarchy does not have distributed pipelining enabled, the coder does not move delays across that subsystem.

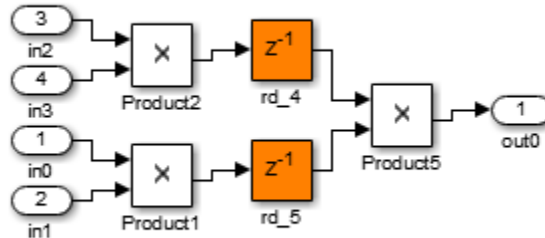
For example, the following model has one level of subsystem hierarchy:



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 design's 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” on page 15-67.

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 15-63

“When To Use Constrained Output Pipelining” on page 15-63

“Requirements for Constrained Output Pipelining” on page 15-63

“Specify Constrained Output Pipelining” on page 15-64

“Limitations of Constrained Output Pipelining” on page 15-64

What is Constrained Output Pipelining?

Constrained output pipelining enables you to specify a nonnegative number of registers at the outputs of a block. Distributed pipelining does not redistribute registers you specify with constrained output pipelining.

The coder redistributes existing delays within your design to try to meet the constraint. If the 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.

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 the 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_re
```

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

The 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.

Pipeline Variables in the MATLAB Function Block

In this section...

“Using the HDL Block Properties Dialog Box” on page 15-65

“Using the Command Line” on page 15-65

“Limitations of Variable Pipelining” on page 15-65

You can insert a pipeline register at the output of a specific MATLAB variable.

Using the HDL Block Properties Dialog Box

To pipeline variables in a MATLAB Function block using the HDL Block Properties dialog box:

- 1 Right-click the block and select **HDL Code > HDL Block Properties**.
- 2 For **VariablesToPipeline**, enter variable names for which you want the coder to insert an output register. Separate variable names with a space.

Using the Command Line

To pipeline variables in a MATLAB Function block, set the `VariablesToPipeline` block parameter using `hdlset_param`. Specify the list of variables as a string, with spaces separating the variables.

For example, if you have a MATLAB Function block, `myFn`, with three variables `v1`, `v2`, `v3`, you can insert pipeline registers at the outputs of the three variables by entering:

```
hdlset_param('full/path/to/myFn','VariablesToPipeline', 'v1 v2 v3')
```

Limitations of Variable Pipelining

The coder cannot insert a pipeline register for a MATLAB variable if it is:

- In a conditional statement or 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 use variable pipelining to add a pipeline register for y :

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

- In a data feedback loop. For example, in the following code, the t and $pvar$ variables cannot be pipelined:

```
persistent pvar;
t = u + pvar;
pvar = t + v;
```


Reduce Critical Path With Distributed Pipelining

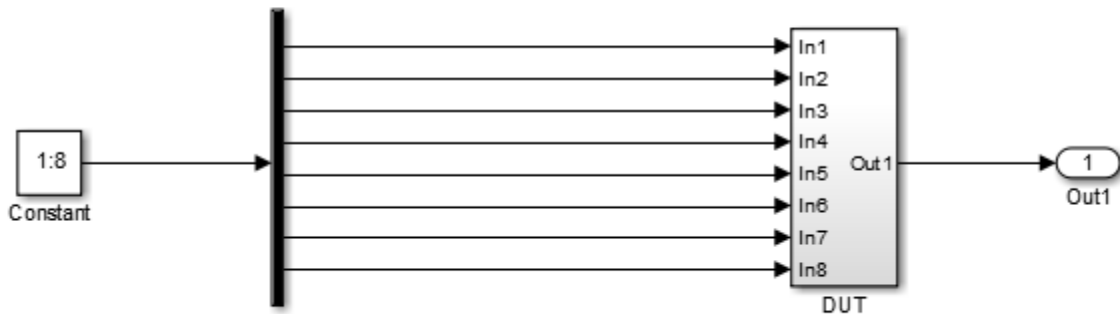
This example shows how to reduce your critical path using distributed pipelining, hierarchical distributed pipelining, output pipelining, and constrained output pipelining.

Before you begin, make sure you have a synthesis tool set up. If you do not have Xilinx ISE set up, you can follow this example, but you will not see maximum clock period results in the Result subpane.

Open the `simple_retiming` model.

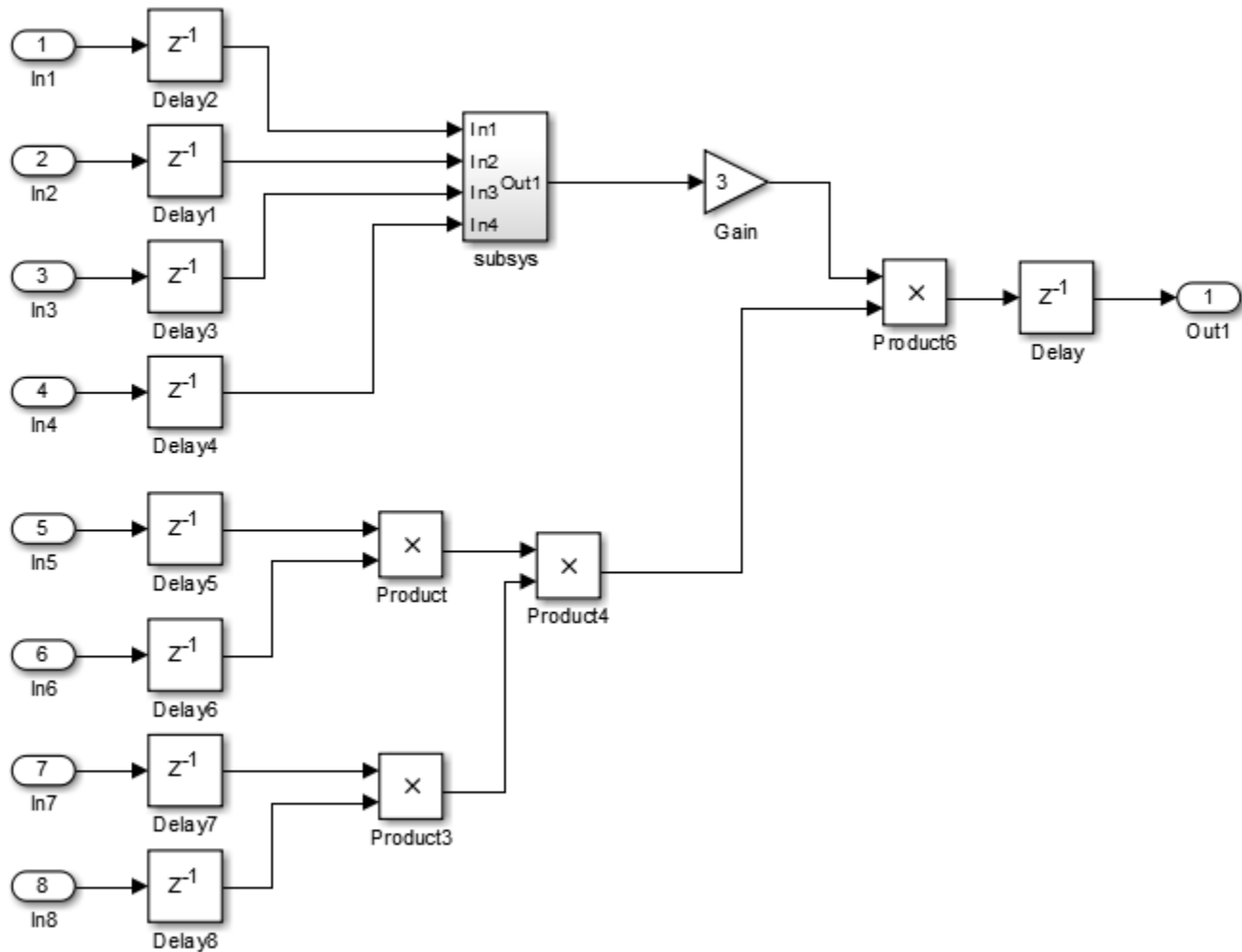
```
addpath(fullfile(docroot, 'toolbox', 'hdlcoder', 'examples'));
simple_retiming
```

 simple_retiming ▶

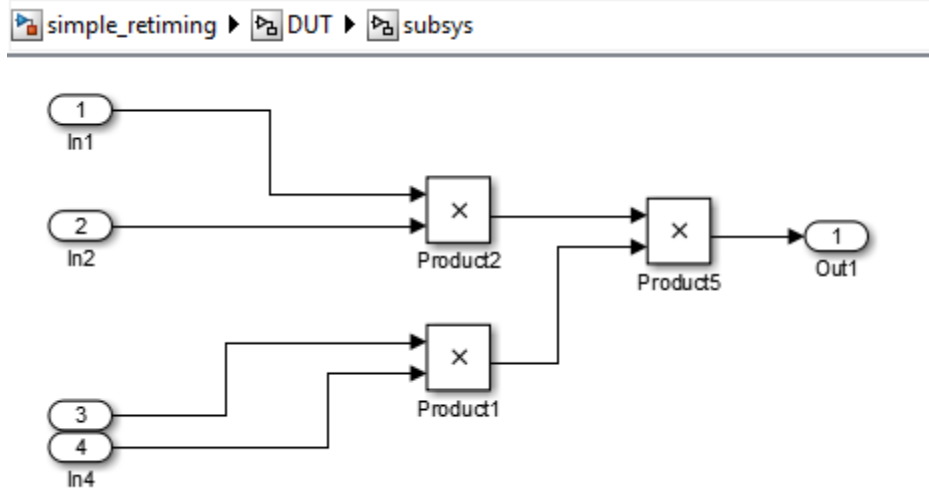


The top-level subsystem is the design under test (DUT). The DUT subsystem contains one subsystem, `subsys`, and other blocks.

simple_retiming ▸ DUT ▸



The subsystem 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 the 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 **DistributedPipelining**, 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**.

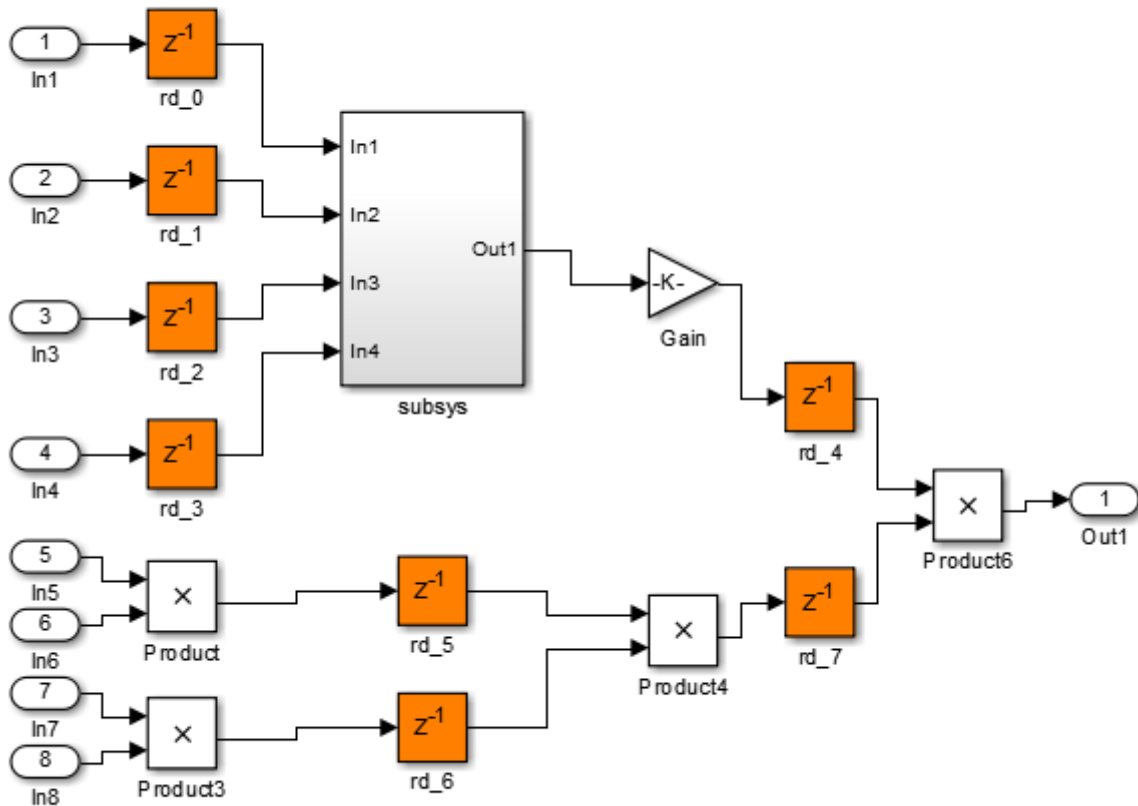
Minimum period: 22.025ns (Maximum Frequency: 45.403MHz)

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 the 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.

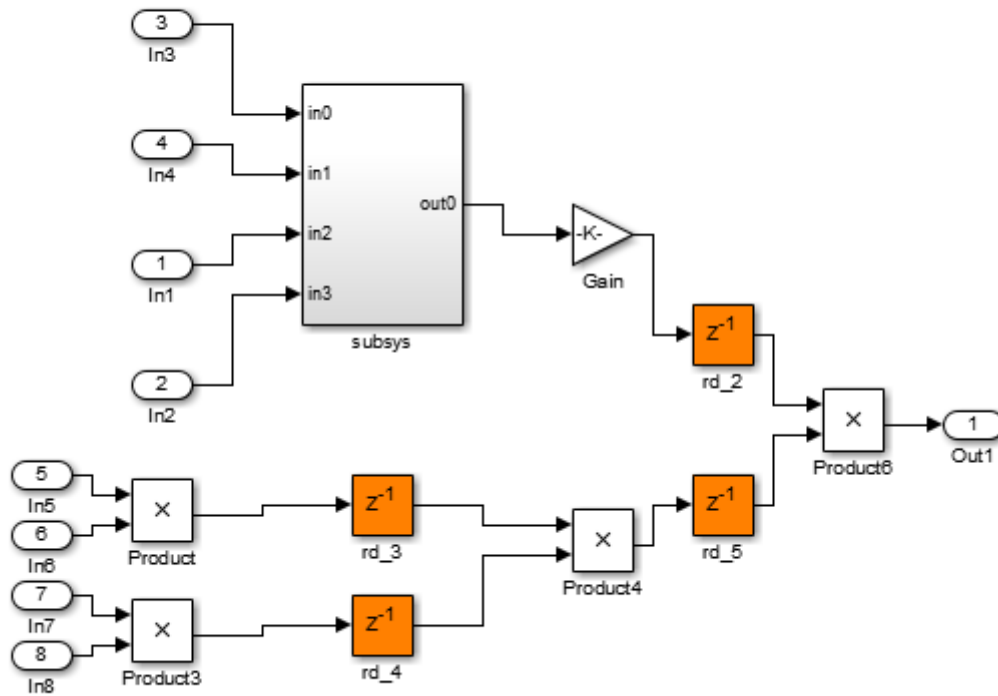
Minimum period: 17.263ns (Maximum Frequency: 57.928MHz)

The maximum clock frequency has increased.

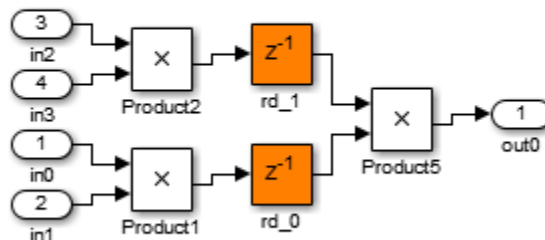
In the Code Generation Report, click **Distributed Pipelining** to open the distributed pipelining report. Click the `gm_simple_retiming` link to open the generated model.

The coder has distributed delays in the DUT and within the subsystem block.

gm_simple_retiming ▶ DUT ▶



gm_simple_retiming ▸ DUT ▸ subsystems



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.

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

- “Create and Use Code Generation Reports” on page 16-2
- “Resource Utilization Report” on page 16-5
- “Optimization Report” on page 16-7
- “Traceability Report” on page 16-10
- “Web View of Model in Code Generation Report” on page 16-28
- “Generate Code with Annotations or Comments” on page 16-34
- “Check Your Model for HDL Compatibility” on page 16-38
- “Create a Supported Blocks Library” on page 16-41
- “Trace Code Using the Mapping File” on page 16-43
- “Add or Remove the HDL Configuration Component” on page 16-46

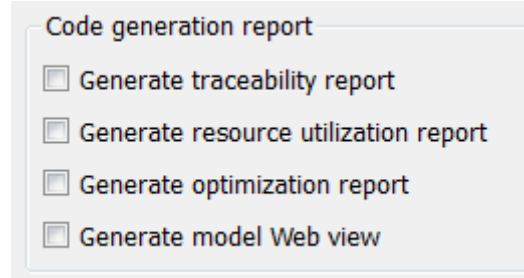
Create and Use Code Generation Reports

Information Included in Code Generation Reports

The coder 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, 'on'
Generate resource utilization report	ResourceReport, 'on'
Generate optimization report	OptimizationReport, 'on'
Generate model Web view	GenerateWebview, 'on'

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 16-7
- “Web View of Model in Code Generation Report” on page 16-28

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 Code 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**, the 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**, the 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: [qm_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 the 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

Traceability Report

In this section...
“Traceability Report Overview” on page 16-10
“Generating a Traceability Report from Configuration Parameters” on page 16-14
“Generating a Traceability Report from the Command Line” on page 16-17
“Keeping the Report Current” on page 16-20
“Tracing from Code to Model” on page 16-20
“Tracing from Model to Code” on page 16-22
“Mapping Model Elements to Code Using the Traceability Report” on page 16-25
“Traceability Report Limitations” on page 16-27

Traceability Report Overview

Even a relatively small model can generate hundreds of lines of HDL code. The coder provides the traceability report section to help you navigate more easily between the generated code and your source model. When you enable traceability, the 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 screenshot shows a window titled "HDL Code Generation Report" with a sidebar on the left and a main content area on the right. The sidebar contains a "Contents" section with links to "Summary", "Clock Summary", and "Traceability Report", and a "Generated Source Files" section with links to "combo_pkg.vhd", "Gain_Subsystem.vhd", "Chart.vhd", "MATLAB_Function.vhd", and "combo.vhd". The main content area displays the "Traceability Report for mcombo" with a "Table of Contents" listing sections like "Eliminated / Virtual Blocks" and "Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions". Below the table of contents is a section titled "Eliminated / Virtual Blocks" containing a table with two columns: "Block Name" and "Comment". The table lists various blocks and their comments, such as "<S3>/In1" with comment "Inport" and "<S8>:461" with comment "Not traceable". At the bottom of the window are "OK" and "Help" buttons.

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	Outport
<S3>/Out2	Outport
<S3>/Out3	Outport
<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	Outport

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

OK Help

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**.

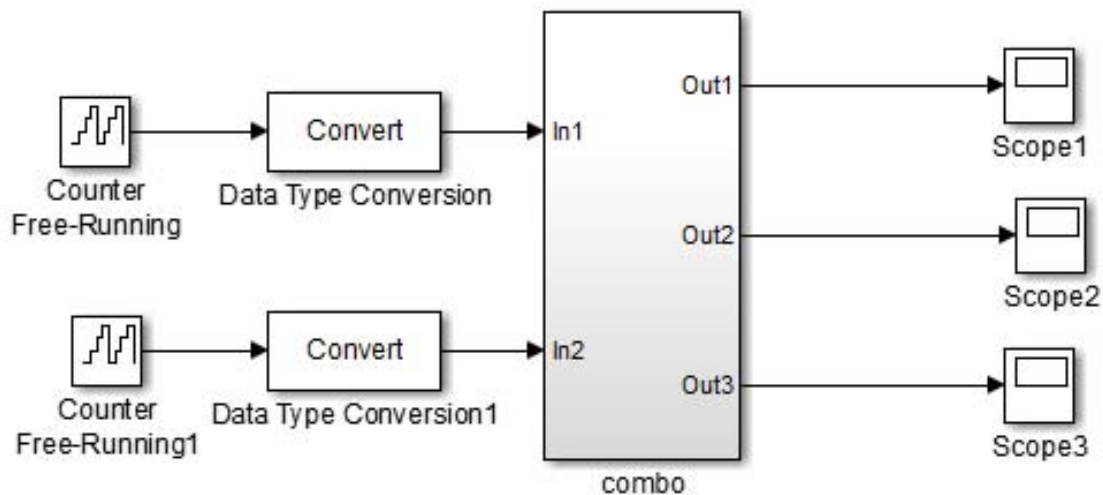
Note If your model includes blocks that have requirements comments, you can also render the comments as hyperlinked comments within the HTML code generation report. See “Requirements Comments and Hyperlinks” on page 16-35 for more information.

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:

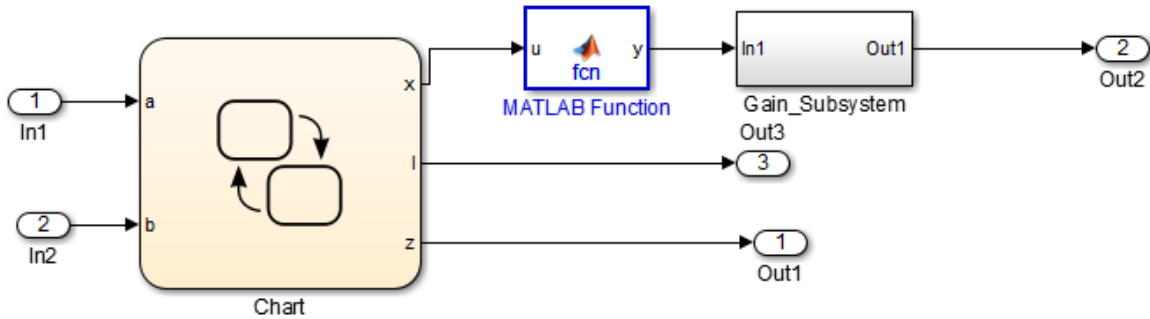
```
mcombo
```

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.



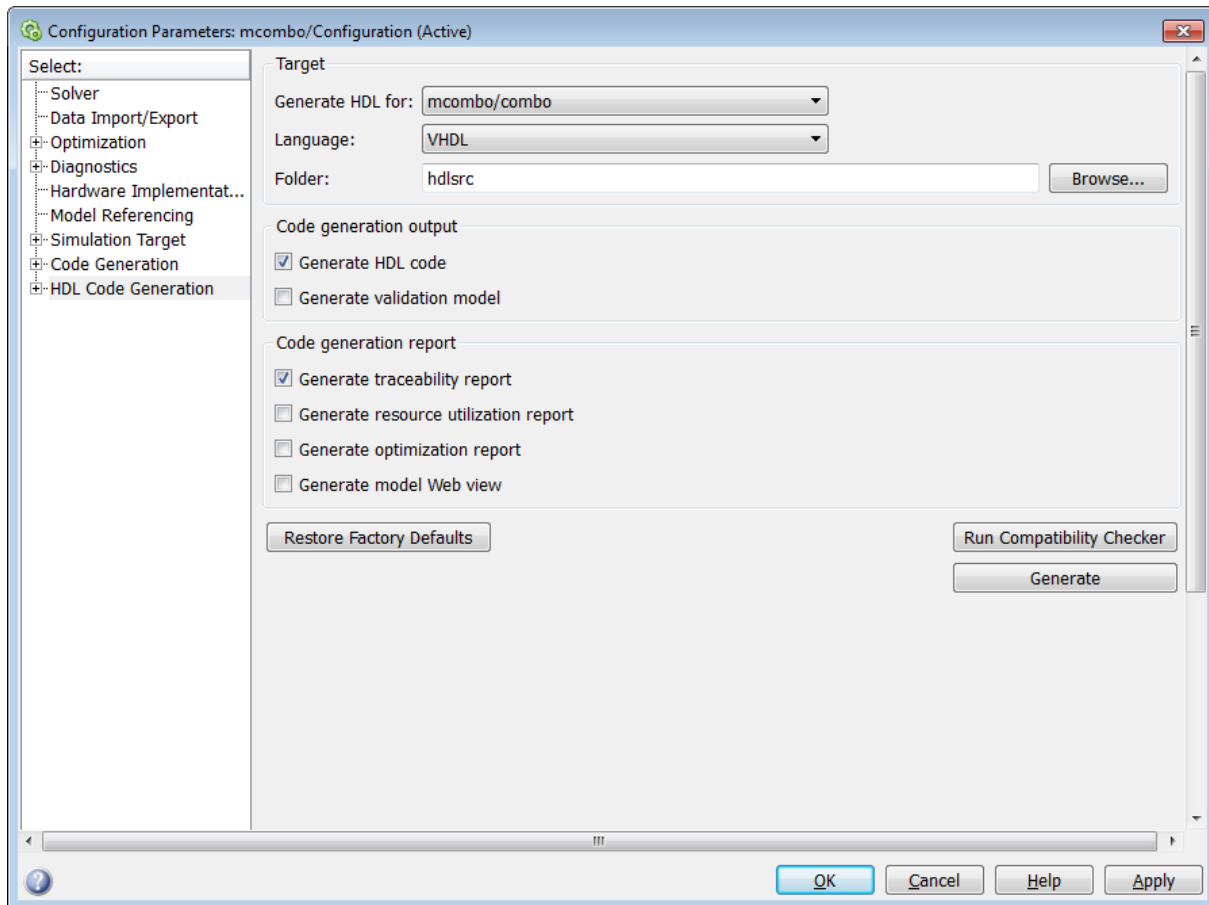
Generating a Traceability Report from Configuration Parameters

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 16-35 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**.



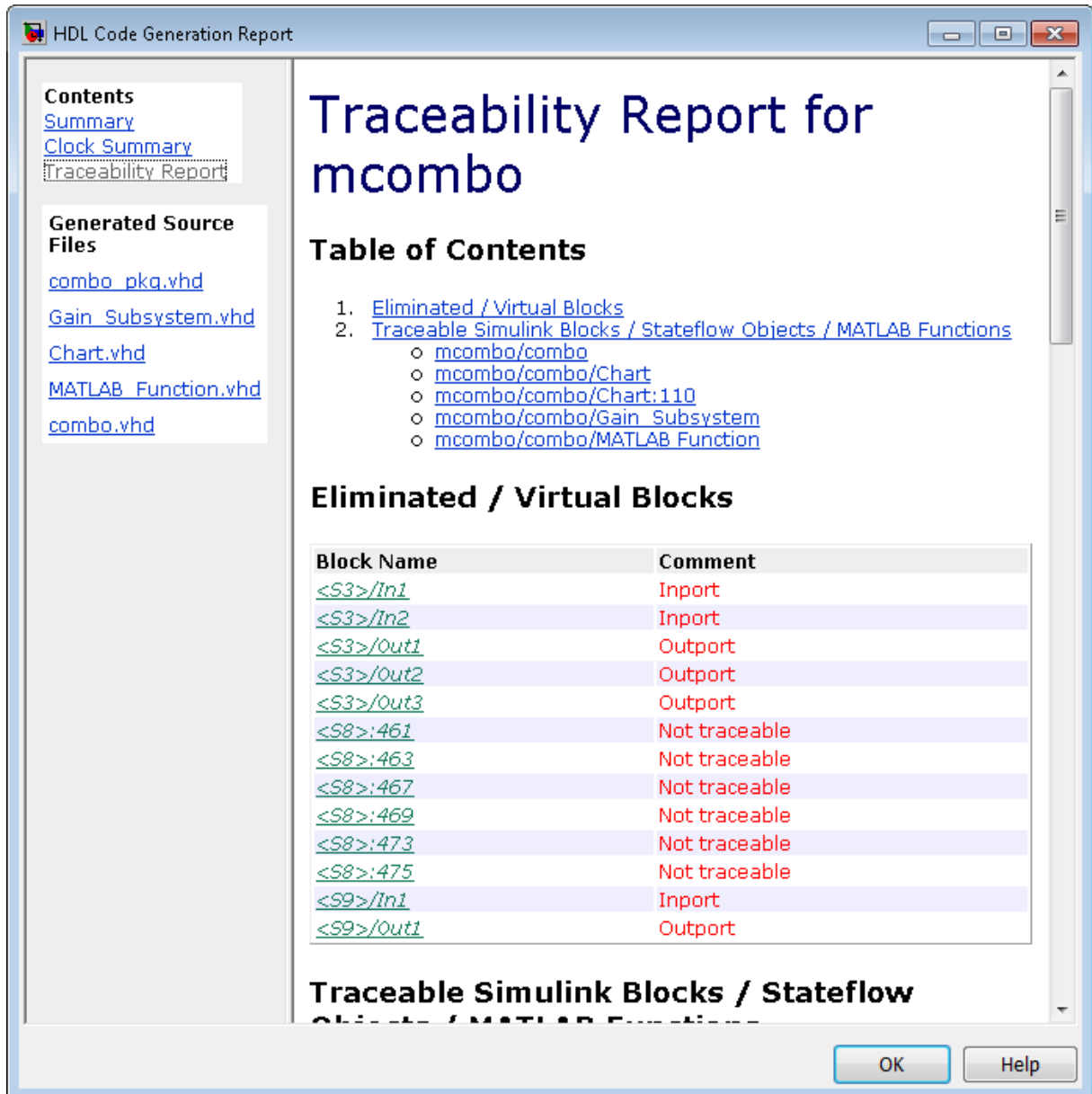
6 Click **Generate** to initiate code and report generation.

When you select **Generate traceability report**, the 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:



- 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 The 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 16-20
- “Tracing from Model to Code” on page 16-22
- “Mapping Model Elements to Code Using the Traceability Report” on page 16-25

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, the coder generates a report for the top-level model.

- 3 Enable the `makehdl` property `Traceability` by entering:

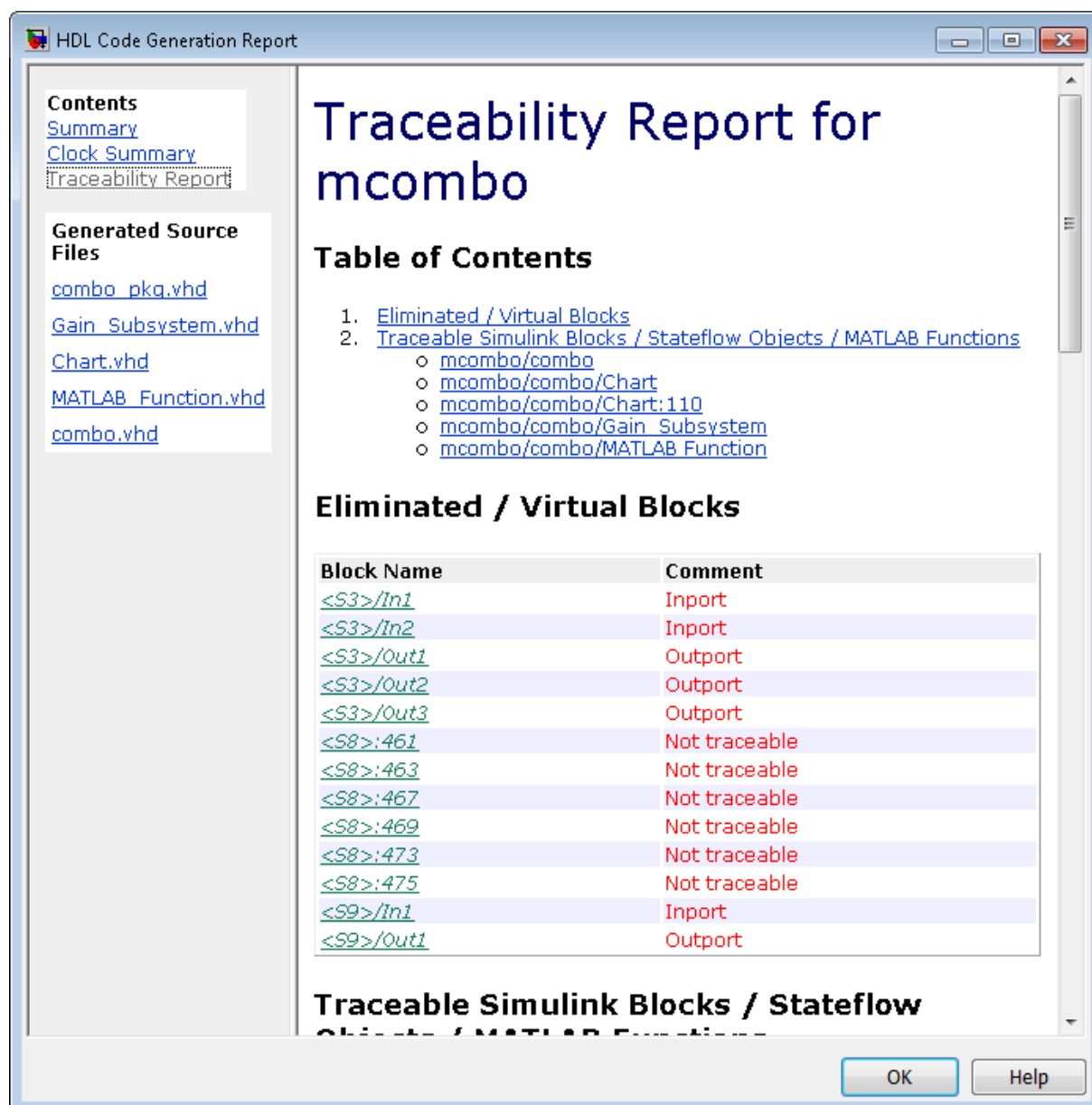
```
makehdl(gcb, 'Traceability', 'on');
```

When you enable traceability, the 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:



- 4 To view the different report sections or view the generated code files, click the hyperlinks in the **Contents** pane of the report window.

Tip The 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 16-20
- “Tracing from Model to Code” on page 16-22
- “Mapping Model Elements to Code Using the Traceability Report” on page 16-25

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.

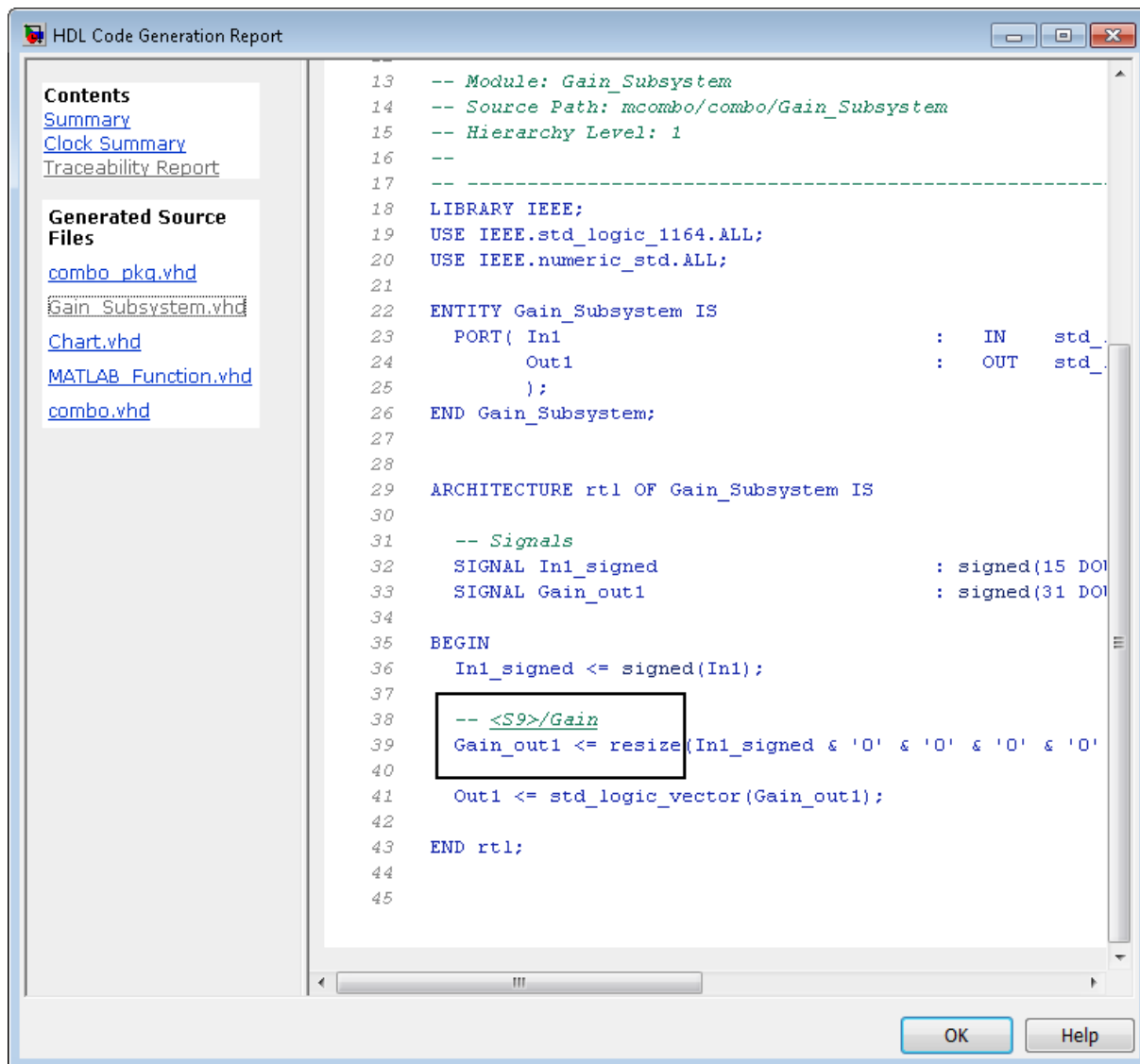
Tracing from Code to Model

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 16-14 or “Generating a Traceability Report from the Command Line” on page 16-17).

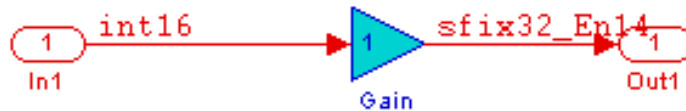
- 2 In the left pane of the HTML report window, click the desired file name in the **Generated Source Files** table to view a source code file.

The following figure shows a view of the source file Gain_Subsystem.vhd.



- 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

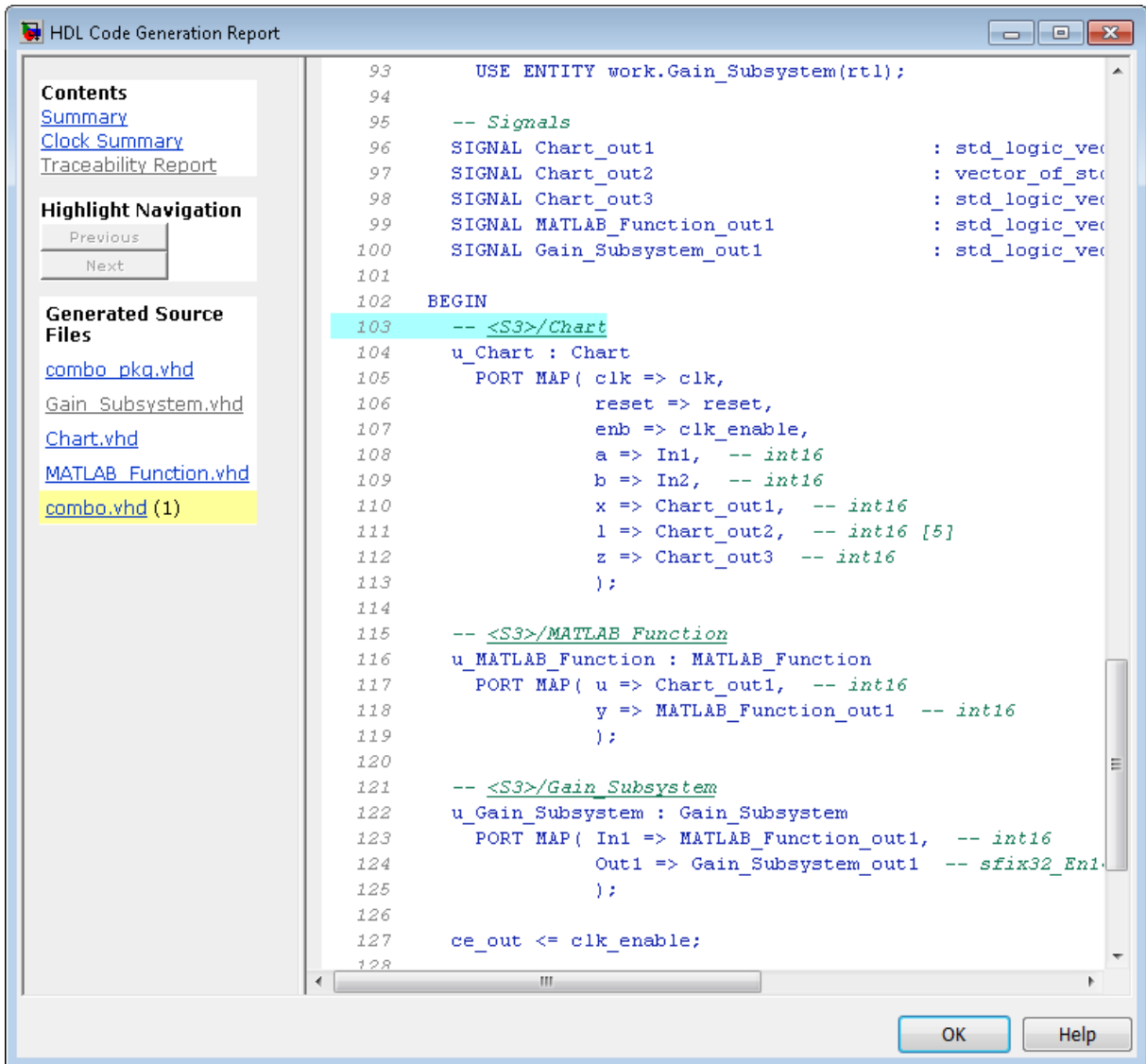
To trace a model component:

- 1 Generate code and open an HTML report for the desired DUT (see “Generating a Traceability Report from Configuration Parameters” on page 16-14 or “Generating a Traceability Report from the Command Line” on page 16-17).

Tip If you have not generated code for the model, the 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.

The screenshot shows a window titled "HDL Code Generation Report" with a sidebar on the left and a main content area on the right. The sidebar contains a "Contents" section with links for "Summary", "Clock Summary", and "Traceability Report", and a "Generated Source Files" section with links for "combo_pkg.vhd", "Gain_Subsystem.vhd", "Chart.vhd", "MATLAB_Function.vhd", and "combo.vhd". The main content area displays the "Traceability Report for mcombo" with a "Table of Contents" listing sections like "Eliminated / Virtual Blocks" and "Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions". Below the table of contents, the "Eliminated / Virtual Blocks" section contains a table with two columns: "Block Name" and "Comment".

Block Name	Comment
<S3>/In1	Inport
<S3>/In2	Inport
<S3>/Out1	Outport
<S3>/Out2	Outport
<S3>/Out3	Outport
<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	Outport

At the bottom of the window, there are "OK" and "Help" buttons.

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 16-28
“Generate HTML Code Generation Report with Model Web View” on page 16-29
“Model Web View Limitations” on page 16-33

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 `rtwdemo_md1reftop` model.
- 2 Open the Configuration Parameters dialog box or Model Explorer and navigate to the **Code Generation** pane.
- 3 Specify `ert.tlc` for the **System target file** parameter.
- 4 Open the **Code Generation > Report** pane.
- 5 Select the following parameters:
 - **Create code generation report**
 - **Open report automatically**
 - **Code-to-model**
 - **Model-to-code**
 - **Generate model Web view**

Note These settings specify only the top model, not referenced models.

- 6 Open the Configuration Parameters for the referenced model, `rtwdemo_md1refbot` and perform steps 3–5.
- 7 Save the models, `rtwdemo_md1reftop` and `rtwdemo_md1refbot`.
- 8 From the top model diagram, press **Ctrl+B**. After building the model and generating code, the code generation report for the top model opens in a MATLAB Web browser.
- 9 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' window. On the left, there is a navigation pane with sections for 'Contents', 'Generated Files', and 'Referenced Models'. The 'Generated Files' section is expanded to show 'Main file' (ert_main.c) and 'Model files' (rtwdemo_mdireftop.c, rtwdemo_mdireftop.h, rtwdemo_mdireftop_private.h, rtwdemo_mdireftop_types.h). The 'Referenced Models' section shows rtwdemo_mdireftop. The main area displays C code for the 'rtwdemo_mdireftop_step' function, with line numbers 49 to 68. The code includes local block I/O variables and a discrete pulse generator. Below the code is a Simulink model diagram titled 'rtwdemo_mdireftop'. The diagram shows three blocks: 'rtwdemo_mdireftop_upper' (pink), 'rtwdemo_mdireftop_lower' (green), and 'rtwdemo_mdireftop_upper' (blue). Each block has an 'input' and 'output' port. The pink block is connected to the green block, which is connected to the blue block. There are also three scope blocks: 'ScopeA', 'ScopeB', and 'ScopeC'. The diagram is annotated with 'CounterA', 'CounterB', and 'CounterC'. A text box in the diagram explains Model Reference and provides instructions for generating code for Simulink Coder and Embedded Coder. At the bottom right, there are 'OK' and 'Help' buttons.

- 10 Click a link in the code. The model Web view displays and highlights the corresponding block in the model.
- 11 To highlight the generated code for a referenced model block in your model, click CounterB. The corresponding code is highlighted in the source code pane.

Note You cannot open the referenced model diagram in the Web view by double-clicking the referenced model block in the top model.

- 12** To open the code generation report for a referenced model, in the left navigation pane, below **Referenced Models**, click the link, `rtwdemo_md1refbot`. The source files for the referenced model are displayed along with the Web view of the referenced model.
- 13** 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.

- 1** Open the `mcombo` model.

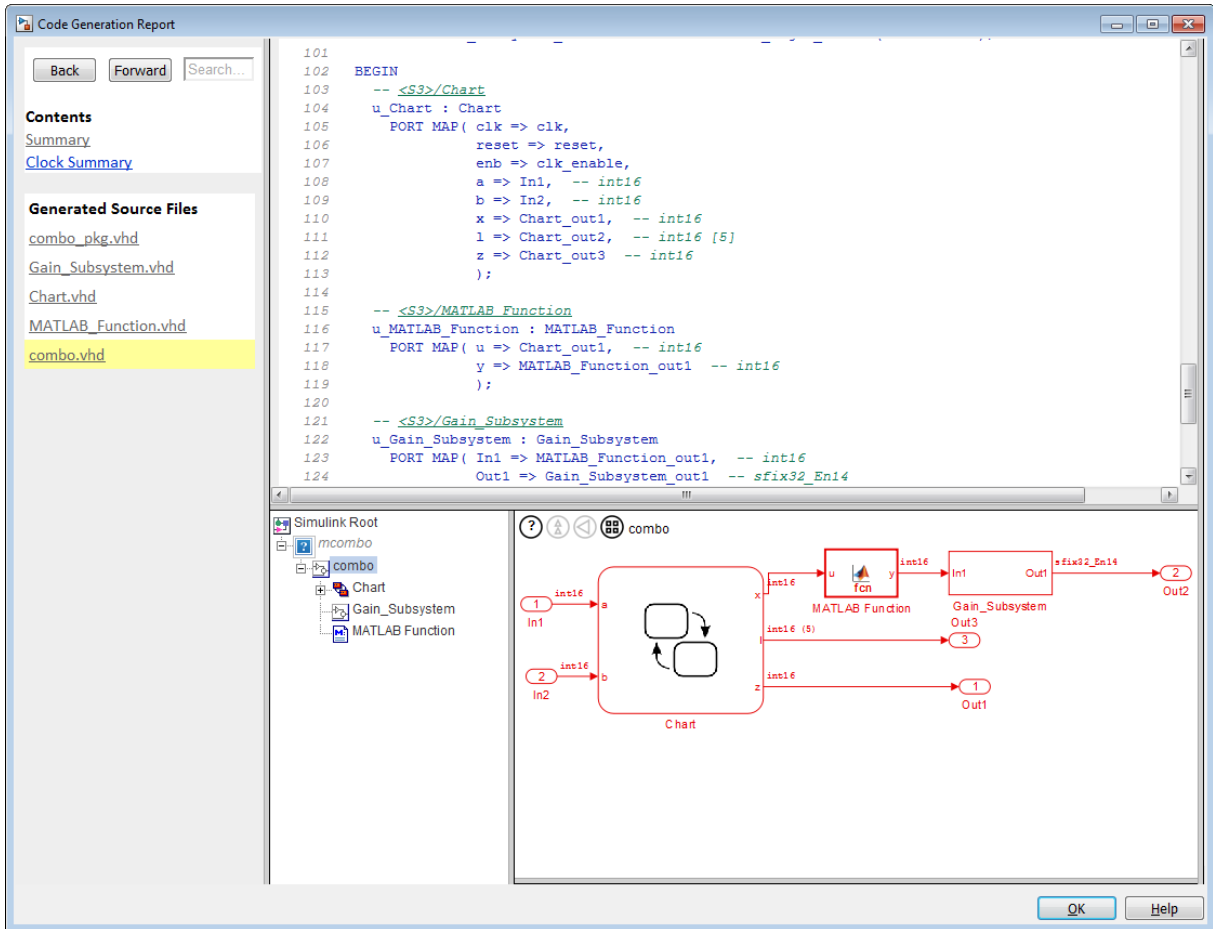
- 2** Open the **Configuration Parameters** dialog box or **Model Explorer** and navigate to the **HDL Code Generation** pane.

- 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.



- 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 Web Views” in the Simulink Report Generator documentation.

For more information about navigating between the generated code and the model diagram, see :

- “Trace Model Objects to Generated Code”
- “Trace Code to Model Objects Using Hyperlinks”

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 outport 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 16-34
“Text Comments” on page 16-34
“Requirements Comments and Hyperlinks” on page 16-35

The following sections describe how to use the coder 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. The 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 “Annotate Diagrams” 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. The 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. The 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.

```

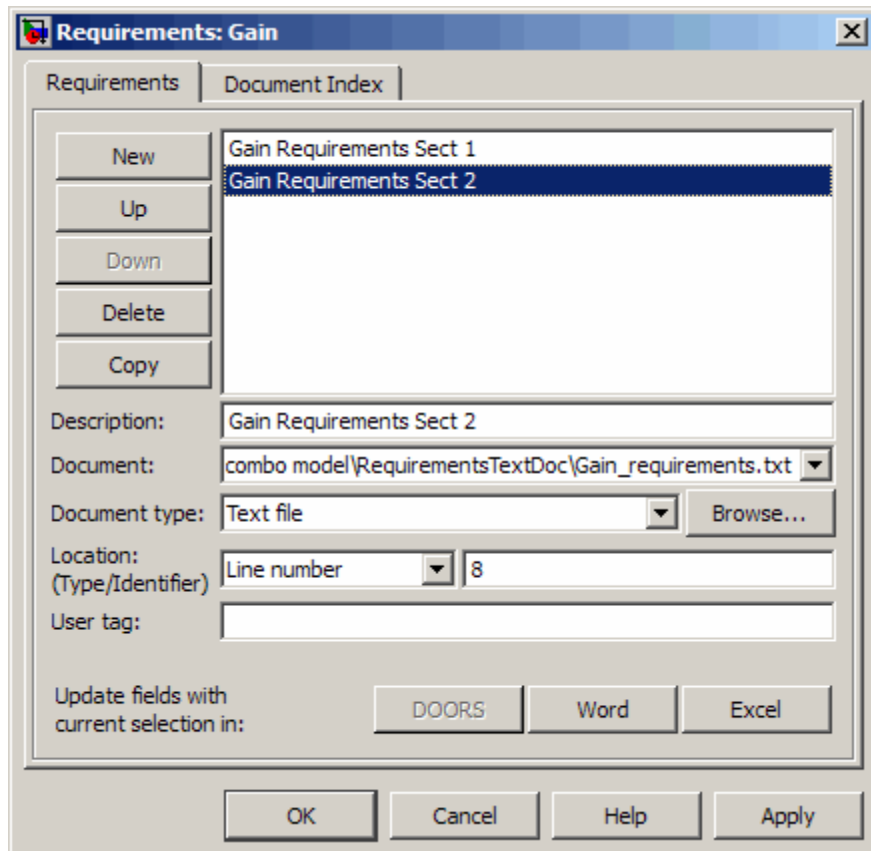
36 BEGIN
37     In1_signed <= signed(In1);
38
39     --
40     -- Block requirements for <S10>/Gain
41     -- 1. Gain Requirements Sect 1
42     -- 2. Gain Requirements Sect 2
43     Gain_gainparam <= to_signed(16384, 16);
44
45     Gain_out1 <= resize(In1_signed(15 DOWNT0 0) & '0'
46
47
48     Out1 <= std_logic_vector(Gain_out1);
49
50 END rtl;
```

- *Hyperlinked comments:* To include requirements comments as hyperlinked comments in an HTML code generation report, select both **Generate traceability report** and **Include requirements in block comments** in the Configuration Parameters dialog box.

If you generate code from the command line, set the Traceability and RequirementComments properties:

```
makehdl(gcb, 'Traceability', 'on', 'RequirementComments', 'on');
```

The comments include links back to a requirements document associated with the block and to the block within the original model. For example, the following figure shows two requirements links assigned to a Gain block. The links point to sections of a text requirements file.



The following figure shows hyperlinked requirements comments generated for the Gain block.

```
36 BEGIN
37   In1_signed <= signed(In1);
38
39   -- <S10>/Gain
40   --
41   --
42   -- Block requirements for <S10>/Gain
43   -- 1. Gain Requirements Sect 1
44   -- 2. Gain Requirements Sect 2
45   Gain_gainparam <= to_signed(16384, 16);
46
47   Gain_out1 <= resize(In1_signed(15 DOWNT0 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.

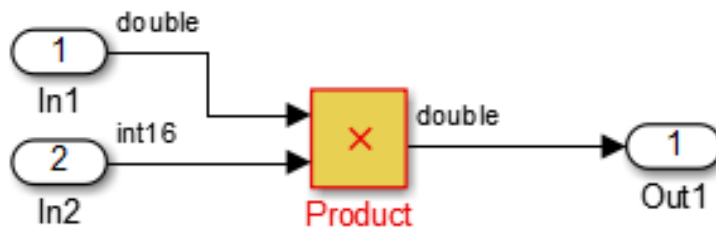
The screenshot shows a web browser window titled "HDL Check Report for ill_product/Subsystem". The address bar shows the location as "C:/Work/models/hdsrc/Subsystem_report.html". The main content area displays the following text:

HDL Code Generation Check Report for [ill_product/Subsystem](#)
Generated on 2008-04-30 14:29:10

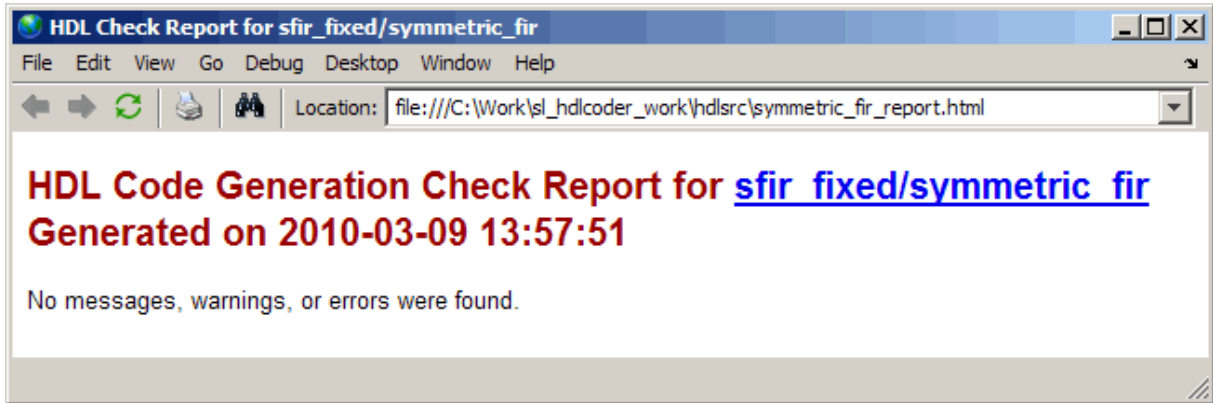
The following table describes blocks for which errors, warnings or messages were reported.

Simulink Block	Level	Description
ill_product/Subsystem/Product	Error	Unhandled mixed double and non-double datatypes at ports of block

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.m` utility creates a library of blocks that are currently supported for HDL code generation. The block library, `hdl1supported`, 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 the coder. To keep the `hdl1supported` library current, you should rebuild the library each time you install a new release. To create the library:

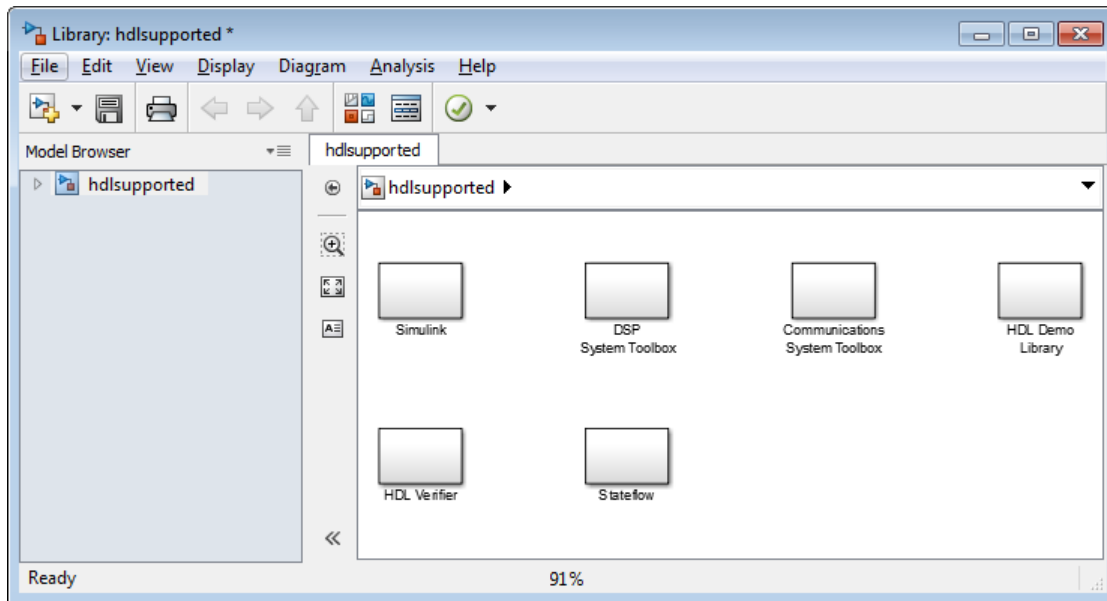
- 1 Type the following at the MATLAB prompt:

```
hdl1lib
```

`hdl1lib` starts generation of the `hdl1supported` library. Many libraries load during the creation of the `hdl1supported` 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 `hdl1supported`, because this document refers to the supported blocks library by that name.

The following figure shows the top-level view of the `hdl1supported` 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” on page 10-2.

Trace Code Using the Mapping File

Note This section refers to generated VHDL entities or Verilog modules generically as “entities.”

A *mapping file* is a text report file generated by `makehdl`. Mapping files are generated as an aid in tracing generated HDL entities back to the corresponding systems in the model.

A mapping file shows the relationship between systems in the model and the VHDL entities or Verilog modules that were generated from them. A mapping file entry has the form

```
path --> HDL_name
```

where *path* is the full path to a system in the model and *HDL_name* is the name of the VHDL entity or Verilog module that was generated from that system. The mapping file contains one entry per line.

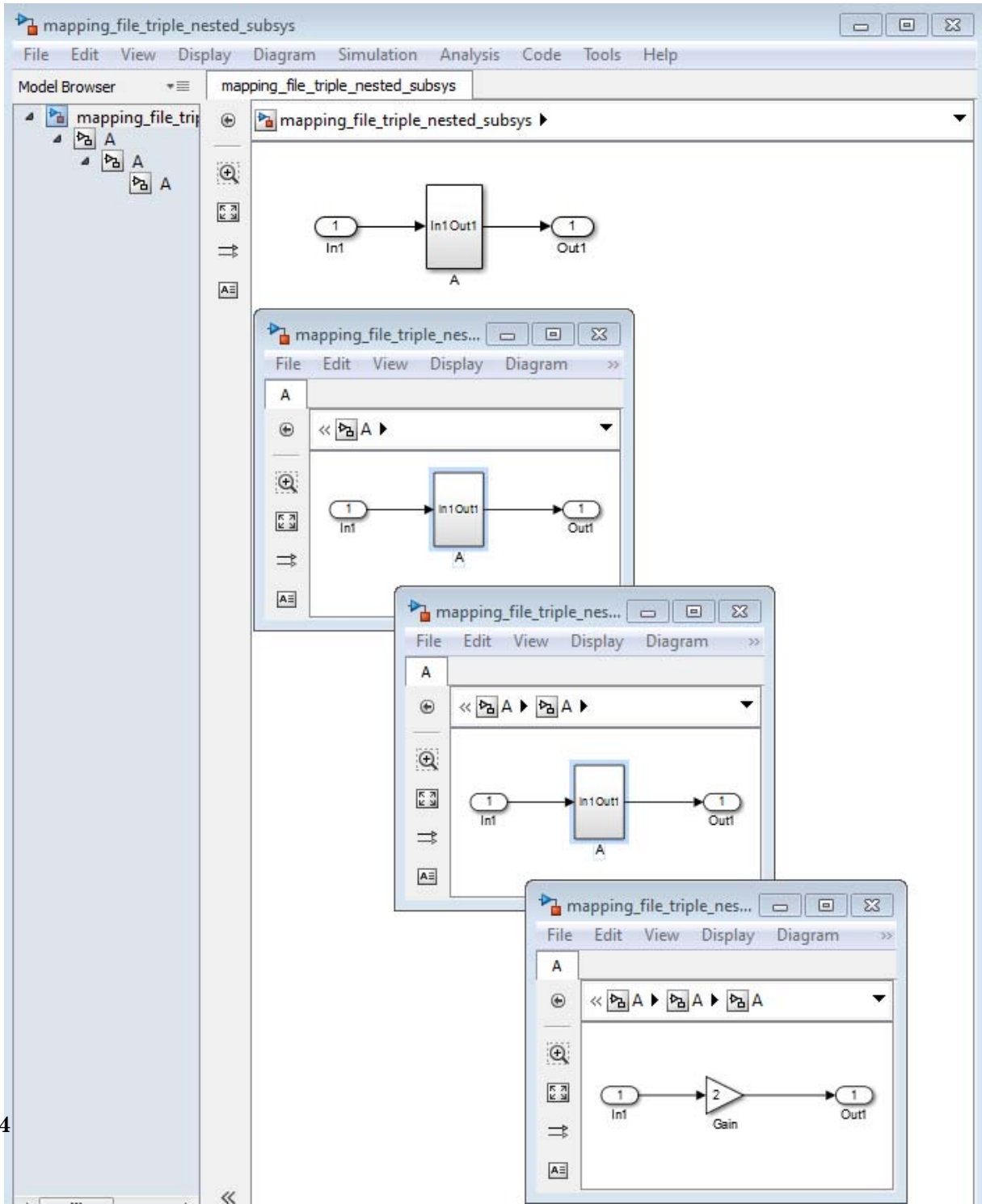
In simple cases, the mapping file may contain only one entry. For example, the `symmetric_fir` subsystem of the `sfir_fixed` model generates the following mapping file:

```
sfir_fixed/symmetric_fir --> symmetric_fir
```

Mapping files are more useful when HDL code is generated from complex models where multiple subsystems generate many entities, and in cases where conflicts between identically named subsystems are resolved by the coder.

If a subsystem name is unique within the model, the 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 16-46

“Adding the HDL Coder Configuration Component To a Model” on page 16-46

“Removing the HDL Coder Configuration Component From a Model” on page 16-47

What Is the HDL Configuration Component?

The *HDL configuration component* is an internal data structure that the 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**.

The 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 17-2
- “HDL Coding Standards” on page 17-4
- “Generate an HDL Coding Standard Report” on page 17-5
- “HDL Coding Standard Rules” on page 17-7
- “Generate an HDL Lint Tool Script” on page 17-11

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 the coder automatically corrected the code to conform to the coding standard.

The report also lists the rules in the coding standard with which the generated code complies. You can inspect the report to see which coding standard rules the coder checks.



To learn more about HDL coding standards, see “HDL Coding Standards” on page 17-4.

Rule Summary

The rule summary section shows the total numbers of errors, warnings, and messages, and lists the corresponding rules. Each rule shown in the summary links to the rule in the detailed rule hierarchy section.

Rule Hierarchy

The rule hierarchy section lists every rule the 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.

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” on page 17-7.

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

The 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. The coder can also generate third-party lint tool scripts to use to check your generated HDL code.

To learn more about the HDL coding standard report, see “HDL Coding Standard Report” on page 17-2.

Generate an HDL Coding Standard Report

In this section...

“Using the HDL Workflow Advisor” on page 17-5

“Using the Command Line” on page 17-5

To learn more about the HDL coding standard report, see “HDL Coding Standard Report” on page 17-2.

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 style** tab.
- 2** For **HDL coding standard**, select **Industry** 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 me
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.
### Industry Compliance report with 4 errors, 18 warnings, 5 messages.
### Generating Industry Compliance Report symmetric_fir_Industry_report.htm
```

```
### Generating SpyGlass script file sfir_fixed_symmetric_fir_spyglass.prj  
### HDL code generation complete.
```

To open the report, click the report link.

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.

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), and keywords in VHDL (IEEE1076.X) must not be used.	There are Verilog or VHDL keywords within the names in your design.	Update the names in your design so that they do not contain Verilog or VHDL keywords.
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 foo and Foo cannot be in the same scope.	Update the names in your design so that no two names within the same scope differ only in case.
1.A.A.9 <i>Warning</i>	Top-level module/entity and port names should be less than or equal to 16 characters in length and not be mixed-case.	A top-level module, entity, or port name in the generated code is longer than 16 characters, or uses letters with mixed case.	Update the indicated name in your design so that it is less than or equal to 16 characters long, and all letters are lowercase, or all letters are uppercase.

Rule / Severity	Message	Problem	How to fix
1.A.B.1 <i>Error</i>	<p>Module and Instance names should be between 2 and 32 characters in length. (Verilog)</p> <p>Entity names and instance names should be between 2 and 32 characters in length. (VHDL)</p>	<p>A module, instance, or entity name in the generated code is less than 2 characters long, or more than 32 characters long.</p>	<p>Update function names or subsystem names in your design to be between 2 and 32 characters long.</p>
1.A.C.3 <i>Error</i>	<p>Signal names, port names, parameter names, define names and function names should be between 2 and 40 characters in length. (Verilog)</p> <p>Signal names, variable names, type names, label names and function names should be between 2 and 40 characters in length. (VHDL)</p>	<p>A signal, port, parameter, define, or function name in the generated code is less than 2 characters long, or more than 40 characters long.</p>	<p>Update the indicated name in your design so that it is between 2 and 40 characters long.</p>

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>	The filename extension of an include file is not one of the standard extensions.	<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 VerilogFileExtension and VHDLFileExtension properties from the command line.</p>
2.C.D.1 <i>Error</i>	Do not specify flip-flop (or RAM) initial value using initial construct.	The generated HDL code for your design contains an unsynthesizable initial statement.	Disable the Initialize block RAM or Initialize all RAM blocks option in the HDL Workflow Advisor.
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>	The generated HDL code contains an if-elseif statement with more than 7 branches.	<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 ...</pre>

Rule / Severity	Message	Problem	How to fix
			else
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 to use fixed-point data types.

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. The coder can generate the following lint tool script file formats:

- Leda
- SpyGlass
- Custom

You can further customize the script with initialization, termination, and command strings.

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 **SpyGlass**, **Leda**, or **Custom**.
- 4** Enter text to customize the **Lint initialization**, **Lint command**, and **Lint termination** strings.

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 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 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, termination, and command 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, termination, and command strings:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLLintTool', 'Leda',  
        'HDLLintInit', 'myInitialization', 'HDLLintTerm',  
        'myTermination', 'HDLLintCmd', 'myCommand')
```

Interfacing Subsystems and Models to HDL Code

- “Generate Black Box Interface for Subsystem” on page 18-2
- “Generate Reusable Code for Atomic Subsystems” on page 18-8
- “Model Reference Code Generation Options” on page 18-17
- “Model Referencing for HDL Code Generation” on page 18-18
- “Generate Black Box Interface for Referenced Model” on page 18-21
- “Generate Code for Enabled and Triggered Subsystems” on page 18-23
- “Create a Xilinx System Generator Subsystem” on page 18-27
- “Create an Altera DSP Builder Subsystem” on page 18-30
- “Using Xilinx System Generator for DSP with HDL Coder” on page 18-33
- “Code Generation for HDL Cosimulation Blocks” on page 18-37
- “Generate a Cosimulation Model” on page 18-39
- “Customize the Generated Interface” on page 18-63
- “Pass-Through and No-Op Implementations” on page 18-66

Generate Black Box Interface for Subsystem

In this section...

“What Is a Black Box Interface?” on page 18-2

“Generate a Black Box Interface for a Subsystem” on page 18-2

“Generate Code for a Black Box Subsystem Implementation” on page 18-6

“Restriction for Multirate DUTs” on page 18-7

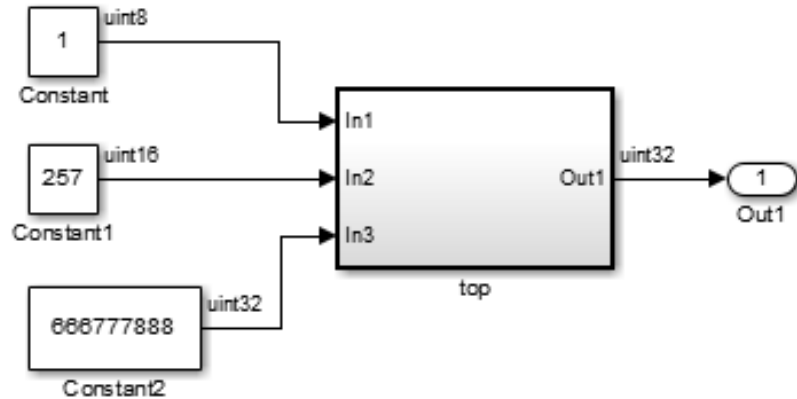
What Is a Black Box Interface?

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.

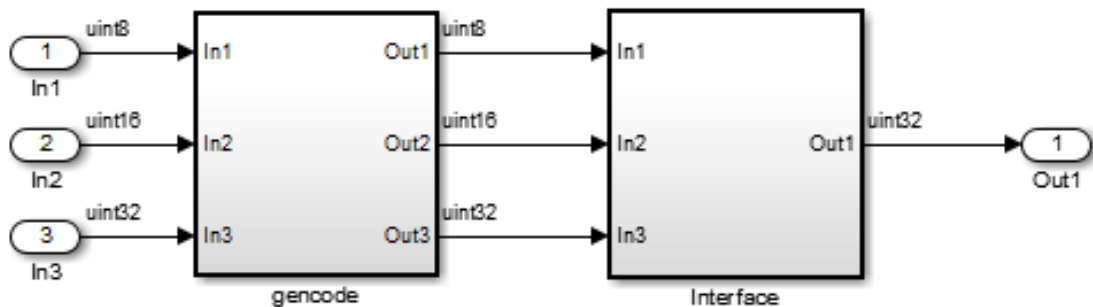
The black box implementation is available only for subsystem blocks below the level of the DUT. Virtual and atomic subsystem blocks of custom libraries that are below the level of the DUT also work with black box implementations.

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: BlackBox

Implementation Parameters

AddClockEnablePort	on
AddClockPort	on
AddResetPort	on
AllowDistributedPipelining	off
ClockEnableInputPort	clk_enable
ClockInputPort	clk
ConstrainedOutputPipeline	0
EntityName	
GenericList	
ImplementationLatency	-1
InlineConfigurations	on
InputPipeline	0
OutputPipeline	0
ResetInputPort	reset
VHDLArchitectureName	rtl
VHDLComponentLibrary	work

OK Cancel Help Apply

The HDL block parameters available for the black box implementation enable you to customize the generated interface. See “Customize the Generated Interface” on page 18-63 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 the Generated Interface” on page 18-63 describes how you can rename or suppress generation of these signals, and customize other aspects of the generated interface.

Restriction for Multirate DUTs

Note that you can generate at most one clock port and one clock enable port. You cannot generate code from a multirate DUT that contains a subsystem with a black box interface.

If you want to generate code for a multirate, multiclock DUT that includes a submodel, use model referencing. For details, see “Model Referencing for HDL Code Generation” on page 18-18.

Generate Reusable Code for Atomic Subsystems

In this section...
“Generate Reusable Code for Identical Atomic Subsystems” on page 18-8 “Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters” on page 18-12

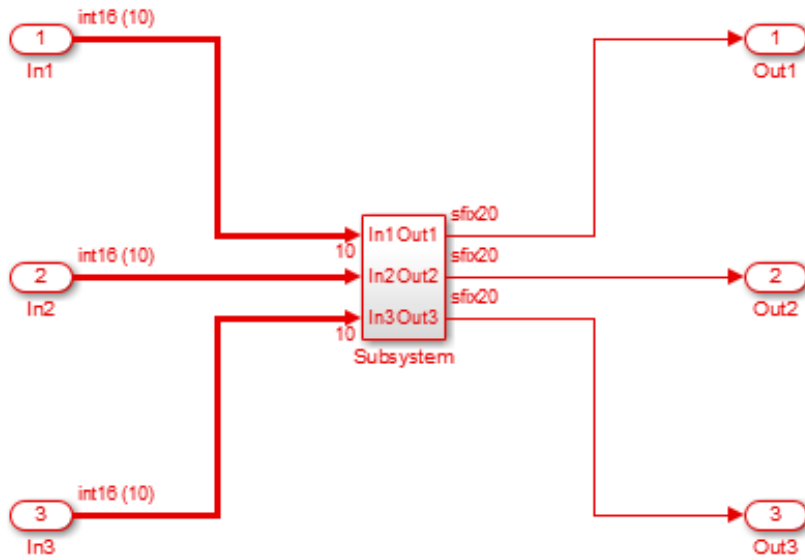
By default, the coder detects and generates reusable code for atomic subsystems that are identical, or identical except for their mask parameter values, at any level of the model hierarchy.

By generating reusable code, you can eliminate the creation of redundant source code files generated for identical subsystems. Each subsystem is configured as an atomic subsystem by selecting **Treat as atomic unit** in the Subsystem block dialog box.

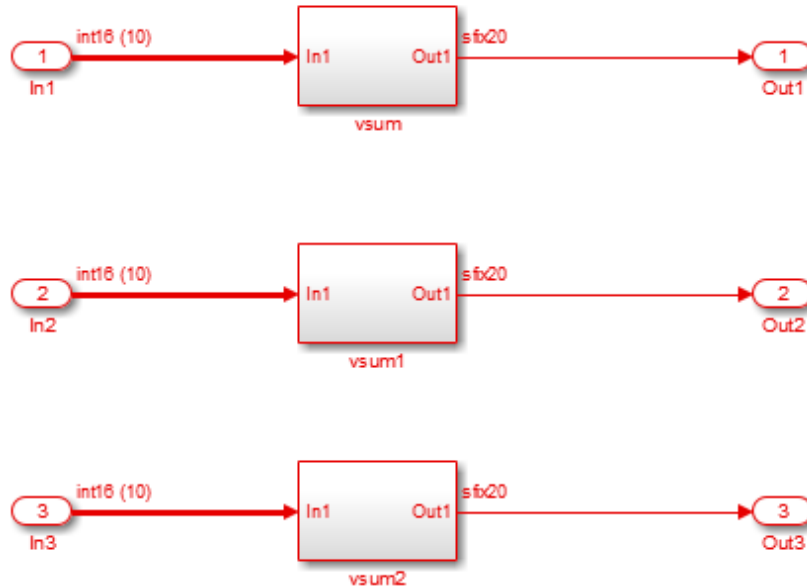
Generate Reusable Code for Identical Atomic Subsystems

An example of a subsystem containing identical subsystems is shown in the following figures.

simplevectorsum_3atomics ▶



simplevectorsum_3atomics ▶ Subsystem ▶



The DUT subsystem contains three subsystems that are identical except for their subsystem names.

By default, the coder generates a single source file, `vsum.vhd`, that provides the required entity and architecture definition for the `vsum` component. The listing below shows the `makehdl` command and its progress messages.

```
>> makehdl('simplevectorsum_3atomics/DUT')
### Generating HDL for 'simplevectorsum_3atomics/DUT'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.
```

```

### Begin VHDL Code Generation
### Working on simplevectorsum_3atomics/DUT/vsum ashdlsrc\vsum.vhd
### Working on simplevectorsum_3atomics/DUT ashdlsrc\DUT.vhd
### Generating package filehdlsrc\DUT_pkg.vhd
### HDL Code Generation Complete.

```

The file generated for the DUT subsystem (DUT.vhd) contains three instantiations of the vsum component, as shown in the following listing.

```

ARCHITECTURE rtl OF DUT IS

    -- Component Declarations
    COMPONENT vsum
        PORT( In1          : IN    vector_of_std_logic_vector16(0 TO 9); -- int16 [10]
              Out1        : OUT   std_logic_vector(19 DOWNT0 0) -- sfix20
              );
    END COMPONENT;

    -- Component Configuration Statements
    FOR ALL : vsum
        USE ENTITY work.vsum(rtl);

    -- Signals
    SIGNAL vsum_out1      : std_logic_vector(19 DOWNT0 0); -- ufix20
    SIGNAL vsum1_out1    : std_logic_vector(19 DOWNT0 0); -- ufix20
    SIGNAL vsum2_out1    : std_logic_vector(19 DOWNT0 0); -- ufix20

BEGIN

    u_vsum : vsum
        PORT MAP( In1 => In1, -- int16 [10]
                  Out1 => vsum_out1 -- sfix20
                );

    u_vsum1 : vsum
        PORT MAP( In1 => In2, -- int16 [10]
                  Out1 => vsum1_out1 -- sfix20
                );

    u_vsum2 : vsum

```

```

PORT MAP( In1 => In3, -- int16 [10]
          Out1 => vsum2_out1 -- sfix20
        );

Out1 <= vsum_out1;

Out2 <= vsum1_out1;

Out3 <= vsum2_out1;

END rtl;

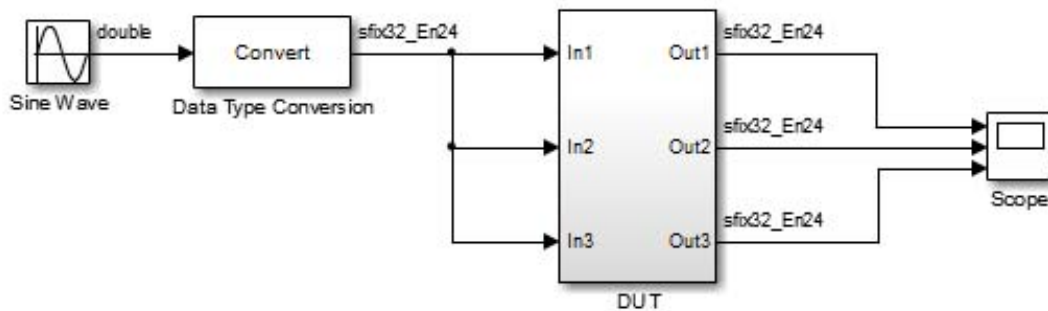
```

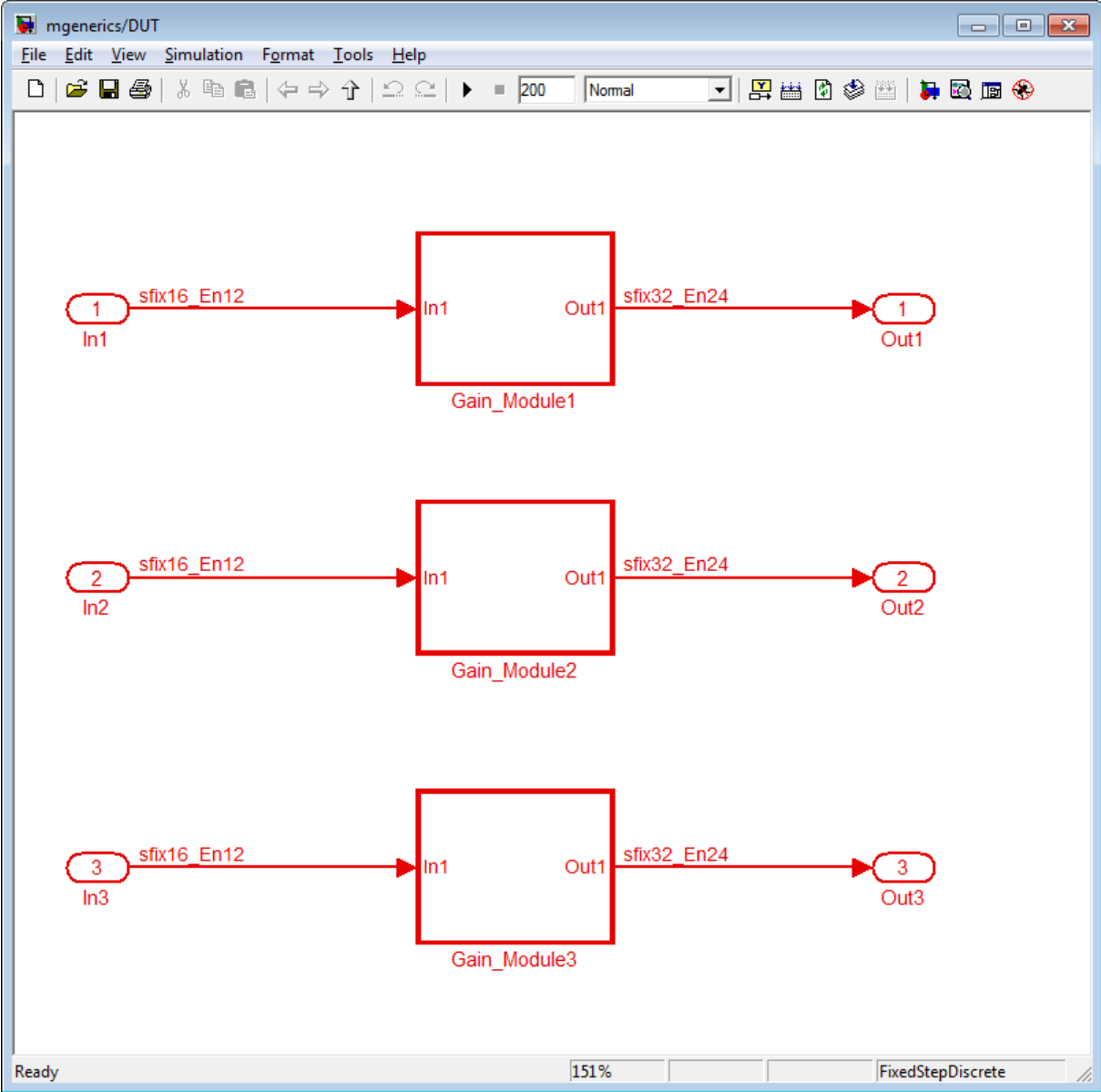
The `HandleAtomicSubsystem` property for `makehdl` lets you control generation of reusable code for atomic subsystems. `HandleAtomicSubsystem` is enabled by default. If you do not wish to generate reusable code for identical atomic subsystems, you can disable `HandleAtomicSubsystem` in your `makehdl` command, as shown in the following example.

```
makehdl(simplevectorsum_3atomics/DUT,'HandleAtomicSubsystem','off')
```

Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters

The following figures show an example of a DUT subsystem containing atomic subsystems that are identical except for their tunable mask parameter values.





In `mgenerics/DUT`, the gain modules are subsystems with gain values represented by tunable mask parameters. Gain values are 0.6 for `Gain_Module1`, 2.4 for `Gain_Module2`, and 6.8 for `Gain_Module3`.

When you enable **Generate parameterized HDL code from masked subsystem**, the coder generates a single source file, `Gain_Module1.v`. This file provides the module definition for the gain module component. The listing below shows the `makehdl` command and its progress messages.

```
>> makehdl('mgenerics/DUT','TargetLanguage','Verilog')
### Generating HDL for 'mgenerics/DUT'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### Begin Verilog Code Generation
### Working on mgenerics/DUT/Gain_Module1 as hdl_prj\hdlsrc\Gain_Module1.v
### Working on mgenerics/DUT as hdl_prj\hdlsrc\DUT.v
### Generating HTML files for code generation report
    in s:\mask2generics_example\hdl_prj\hdlsrc\html\mgenerics directory ...

### HDL Code Generation Complete.
```

The file generated for the DUT subsystem (`DUT.v`) contains three instantiations of the `Gain_Module1` component, as shown in the following listing.

```
module DUT
    (
        In1,
        In2,
        In3,
        Out1,
        Out2,
        Out3
    );

    input  signed [15:0] In1; // sfix16_En12
```



```
input  signed [15:0] In2; // sfix16_En12
input  signed [15:0] In3; // sfix16_En12
output signed [31:0] Out1; // sfix32_En24
output signed [31:0] Out2; // sfix32_En24
output signed [31:0] Out3; // sfix32_En24

wire signed [31:0] Gain_Module1_out1; // sfix32_En24
wire signed [31:0] Gain_Module2_out1; // sfix32_En24
wire signed [31:0] Gain_Module3_out1; // sfix32_En24

// <S1>/Gain_Module1
Gain_Module1 # (.param_gain(2458)
               )
             u_Gain_Module1 (.In1(In1), // sfix16_En12
                           .Out1(Gain_Module1_out1) // sfix32_En24
                           );

assign Out1 = Gain_Module1_out1;

// <S1>/Gain_Module2
Gain_Module1 # (.param_gain(9830)
               )
             u_Gain_Module2 (.In1(In2), // sfix16_En12
                           .Out1(Gain_Module2_out1) // sfix32_En24
                           );

assign Out2 = Gain_Module2_out1;

// <S1>/Gain_Module3
Gain_Module1 # (.param_gain(27853)
               )
             u_Gain_Module3 (.In1(In3), // sfix16_En12
                           .Out1(Gain_Module3_out1) // sfix32_En24
                           );

assign Out3 = Gain_Module3_out1;

endmodule // DUT
```

The file generated for the Gain_Module1 block contains a Verilog parameter, param_gain, as shown in the following listing.

```
module Gain_Module1
(
    In1,
    Out1
);

    input  signed [15:0] In1; // sfix16_En12
    output signed [31:0] Out1; // sfix32_En24

    parameter signed [15:0] param_gain = 2458; // sfix16_En12

    wire signed [15:0] kconst; // sfix16_En12
    wire signed [31:0] Gain_out1; // sfix32_En24

    assign kconst = param_gain;

    // <S2>/Gain
    assign Gain_out1 = In1 * kconst;

    assign Out1 = Gain_out1;

endmodule // Gain_Module1
```

The MaskParameterAsGeneric property for makehdl lets you control generation of reusable code for atomic subsystems. MaskParameterAsGeneric is disabled by default. If you wish to generate reusable code for identical atomic subsystems, you can enable MaskParameterAsGeneric in your makehdl command, as shown in the following example.

```
makehdl(mgenerics/DUT, 'MaskParameterAsGeneric', 'on')
```

See also “Generate parameterized HDL code from masked subsystem” on page 9-75.

Model Reference Code Generation Options

Simulink model referencing enables you to include other models in your DUT subsystem using the Model block.

When you generate HDL code, you can specify the Model block implementation to:

- Generate HDL code for the referenced model and nested submodels.
- Instantiate an HDL wrapper, or black box interface, for legacy or external HDL code.

If you specify a black box interface, the coder does not attempt to generate HDL code for the submodel.

To learn how to generate HDL for a referenced model, see “Model Referencing for HDL Code Generation” on page 18-18.

To learn how to generate a black box interface for a referenced model, see “Generate Black Box Interface for Referenced Model” on page 18-21.

Model Referencing for HDL Code Generation

In this section...

“Benefits of Model Referencing for Code Generation” on page 18-18

“How To Generate Code for a Referenced Model” on page 18-18

“Limitations for Model Reference Code Generation” on page 18-19

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.

The coder incrementally generates code for referenced models according to the **Configuration Parameters dialog box > Model Referencing pane > Rebuild** options.

However, the 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**, the 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, `submodel`, at the command line, enter:

```
hdlset_param ('mydut/submodel', '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

When you generate HDL code for referenced models, the following limitations apply:

- Block parameters for the Model block must be set to their default values.
- If multiple model references refer to the same model, their HDL block properties must be the same.
- Referenced models cannot be protected models.
- Hierarchical distributed pipelining must be disabled.

The coder cannot move registers across a model reference. Therefore, referenced models may inhibit the following optimizations:

- Distributed pipelining
- Constrained output pipelining

The coder cannot apply the streaming optimization to a model reference.

The coder can apply the resource sharing optimization to share submodel instances. However, this optimization can be applied only when all model references that point to the same submodel have the same rate after optimizations and rate propagation. The model reference final rate may differ from the original rate, but all model references that point to the same submodel must have the same final rate.

Generate Black Box Interface for Referenced Model

In this section...

“When to Generate a Black Box Interface” on page 18-21

“How to Generate a Black Box Interface” on page 18-21

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. The 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 submodel, use model referencing. For details, see “Model Referencing for HDL Code Generation” on page 18-18.

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 the Generated Interface” on page 18-63.

- 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.

Generate Code for Enabled and Triggered Subsystems

In this section...

“Code Generation for Enabled Subsystems” on page 18-23

“Code Generation for Triggered Subsystems” on page 18-24

“Best Practices for Using Enabled and Triggered Subsystems” on page 18-26

Code Generation for Enabled Subsystems

An enabled subsystem is a subsystem that receives a control signal via an Enable block. The enabled subsystem executes at each simulation step where the control signal has a positive value. For detailed information on how to construct and configure enabled subsystems, see “Create an Enabled Subsystem” in the Simulink documentation.

The coder supports HDL code generation for enabled subsystems that meet the following conditions:

- The DUT (i.e., the top-level subsystem for which code is generated) must not be an enabled subsystem.
- The coder does not support subsystems that are *both* triggered *and* enabled for HDL code generation.
- The enable signal must be a scalar.
- The data type of the enable signal must be either `boolean` or `ufix1`.
- Outputs of the enabled subsystem must have an initial value of 0.
- All inputs and outputs of the enabled subsystem (including the enable signal) must run at the same rate.
- The **Show output port** parameter of the Enable block must be set to `Off`.
- The **States when enabling** parameter of the Enable block must be set to `held` (i.e., the Enable block does not reset states when enabled).
- The **Output when disabled** parameter for the enabled subsystem output port(s) must be set to `held` (i.e., the enabled subsystem does not reset output values when disabled).

- The following blocks are not supported in enabled subsystems targeted for HDL code generation:
 - dspmlti4/CIC Decimation
 - dspmlti4/CIC Interpolation
 - dspmlti4/FIR Decimation
 - dspmlti4/FIR Interpolation
 - dspsigops/Downsample
 - dspsigops/Upsample
 - HDL Cosimulation blocks for HDL Verifier
 - simulink/Signal Attributes/Rate Transition
 - hlldemolib/FFT
 - hlldemolib/HDL Streaming FFT
 - hlldemolib/Dual Port RAM
 - hlldemolib/Simple Dual Port RAM
 - hlldemolib/Single Port RAM
 - Subsystem black box (SubsystemBlackBoxHDLInstantiation)

The Automatic Gain Controller example model to shows how you can use enabled subsystems in HDL code generation. To open the example model, enter:

```
hdlcoder_agc
```

Code Generation for Triggered Subsystems

A triggered subsystem is a subsystem that receives a control signal via a Trigger block. The enabled triggered executes for one clock cycle each time a trigger event occurs. For detailed information on how to define trigger events and configure triggered subsystems, see “Create a Triggered Subsystem” in the Simulink documentation.

The coder supports HDL code generation for triggered subsystems that meet the following conditions:

- The DUT (that is, the top-level subsystem for which code is generated) must not be a triggered subsystem.
- The coder does not support subsystems that are *both* triggered *and* enabled for HDL code generation.
- The trigger signal must be a scalar.
- The data type of the trigger signal must be either `boolean` or `ufix1`.
- Outputs of the triggered subsystem must have an initial value of 0.
- All inputs and outputs of the triggered subsystem (including the trigger signal) must run at the same rate. (See “Note on Use of the Signal Builder Block” on page 18-26 for information on a special case.)
- The **Show output port** parameter of the Trigger block must be set to `Off`.
- The following blocks are not supported in triggered subsystems targeted for HDL code generation:
 - Discrete-Time Integrator
 - `dspmlti4/CIC Decimation`
 - `dspmlti4/CIC Interpolation`
 - `dspmlti4/FIR Decimation`
 - `dspmlti4/FIR Interpolation`
 - `dspsigops/Downsample`
 - `dspsigops/Upsample`
 - HDL Cosimulation blocks for HDL Verifier
 - `simulink/Signal Attributes/Rate Transition`
 - `hlddemolib/FFT`
 - `hlddemolib/HDL Streaming FFT`
 - `hlddemolib/Dual Port RAM`
 - `hlddemolib/Simple Dual Port RAM`
 - `hlddemolib/Single Port RAM`
 - Subsystem black box (`SubsystemBlackBoxHDLInstantiation`)

Tip For best results the trigger signal should be a synchronous signal.

Best Practices for Using Enabled and Triggered Subsystems

It is good practice to consider the following when using enabled and triggered subsystems in models targeted for HDL code generation:

- For synthesis results to match Simulink results, Enable and Trigger ports should be driven by registered logic (with a synchronous clock) on the FPGA.
- The use of enabled or triggered subsystems can affect synthesis results in the following ways:
 - In some cases the system clock speed may drop by a small percentage.
 - Generated code will use more resources, scaling with the number of enabled or triggered subsystem instances and the number of output ports per subsystem.

Note on Use of the Signal Builder Block

When you connect outputs from a Signal Builder block to a triggered subsystem, you may need to use a Rate Transition block. To run all triggered subsystem ports at the same rate:

- If the trigger source is a Signal Builder block, but the other triggered subsystem inputs come from other sources, insert a Rate Transition block into the signal path before the trigger input.
- If all inputs (including the trigger) come from a Signal Builder block, they have the same rate, so special action is not required.

Create a Xilinx System Generator Subsystem

In this section...

“Why Use Xilinx System Generator Subsystems?” on page 18-27

“Requirements for Xilinx System Generator Subsystems” on page 18-27

“How to Create a Xilinx System Generator Subsystem” on page 18-28

“Limitations for Code Generation from Xilinx System Generator Subsystems” on page 18-28

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 18-28.

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 18-33.

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 `sfix8_en6` and `Fix_8_6` data types, but changing data sign, word length, or fraction length is not allowed.
- For Verilog code generation, Simulink block names in your design cannot be the same as Xilinx names. Similarly, Xilinx blocks in your design cannot have the same name as other Xilinx blocks. The 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 18-30
“Requirements for Altera DSP Builder Subsystems” on page 18-30
“How to Create an Altera DSP Builder Subsystem” on page 18-31
“Determine Clocking Requirements for Altera DSP Builder Subsystems” on page 18-31
“Limitations for Code Generation from Altera DSP Builder Subsystems” on page 18-32

Why Use Altera DSP Builder Subsystems?

You can generate HDL code from a model with both Simulink and Altera DSP Builder Advanced blocks using Altera DSP Builder (DSPB) subsystems.

Using both Simulink and Altera blocks in your model provides the following benefits:

- A single platform for combined Simulink and Altera DSP Builder simulation, code generation, and synthesis.
- Targeted code generation: Altera DSP Builder generates code from Altera blocks; HDL Coder generates code from Simulink blocks.
- HDL Coder area and speed optimizations for Simulink components.

Requirements for Altera DSP Builder Subsystems

You must group your Altera blocks into one or more Altera DSP Builder (DSPB) subsystems for code generation. A DSPB subsystem can contain a hierarchy of subsystems.

To generate code from a Altera DSP Builder subsystem, you must use Quartus II 13.0 or later.

A DSPB subsystem is a Subsystem block with:

- Architecture set to **Module**.
- A valid DSP Builder Advanced Blockset design, including a top-level Device block and DSP Builder Advanced blocks, as defined in the Altera DSP Builder documentation.

How to Create an Altera DSP Builder Subsystem

- 1** Create an Altera DSP Builder Advanced Blockset design as defined in the Altera DSP Builder documentation.
- 2** Create a subsystem containing the Altera DSP Builder Advanced Blockset design, and set its architecture to **Module**.

To see an example that shows HDL code generation for an Altera DSP Builder subsystem, see [Using Altera DSP Builder Advanced Blockset with HDL Coder](#).

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 the 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 the coder displays an error message that says that your DSPB subsystem rate is slower than the base rate, modify the DSPB subsystem inputs so that the DSPB subsystem runs at the base rate in the message.

For example, you can insert an Upsample block.

- 3** Repeat these steps until your DSPB subsystem rate matches the base rate.

To provide a custom clock for your DSPB subsystem:

- 1** In the HDL Workflow Advisor, for **HDL Code Generation > Set Code Generation Options > Set Advanced Options > Clock inputs**, select **Multiple**.
- 2** In the generated HDL code, connect your custom clocks to the DUT clock input ports that corresponds to your DSPB subsystems clock.

Limitations for Code Generation from Altera DSP Builder Subsystems

Code generation for Altera DSP Builder (DSPB) subsystems has the following limitations:

- The DUT subsystem cannot be a DSPB subsystem.
- DSPB subsystems must run at the Simulink model base rate. You may need to iteratively generate code to determine the base rate, because area optimizations can cause local multirate. See “Determine Clocking Requirements for Altera DSP Builder Subsystems” on page 18-31 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

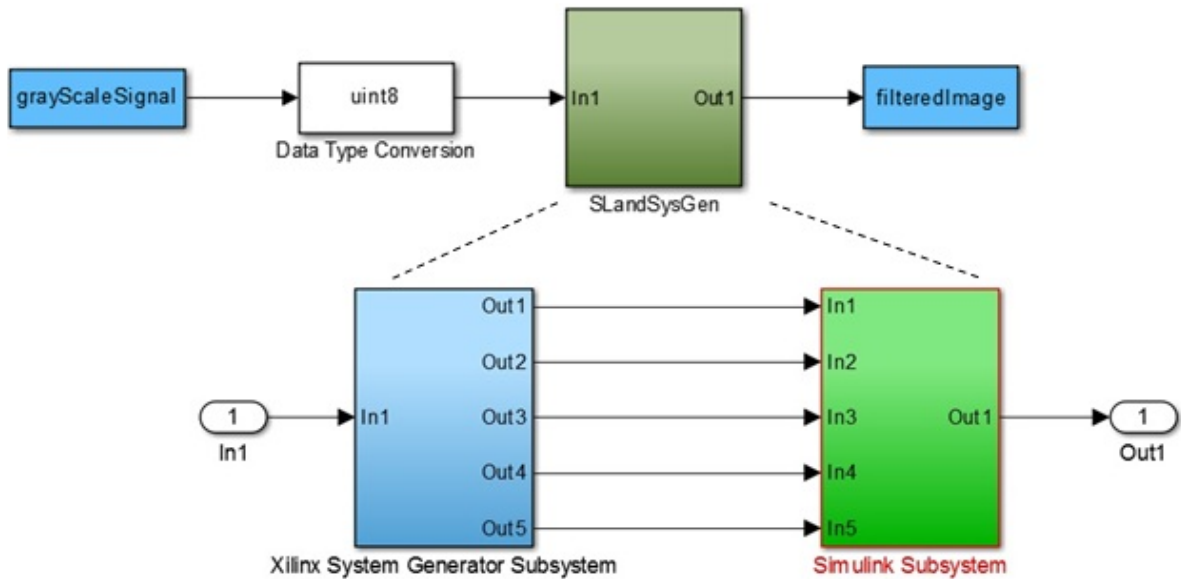
In order to use the Xilinx System Generator Subsystem block, you must have Xilinx ISE 13.4 set up with Simulink.

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_slsysgen/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_slsysgen');  
open_system('hdlcoder_slsysgen/SLandSysGen');
```



Create Xilinx System Generator Subsystem

To create a Xilinx System Generator subsystem:

- 1 Put the Xilinx blocks in one subsystem and set its architecture to "Module" (the default value).
- 2 Place a System Generator token at the top level of the subsystem. You can have subsystem hierarchy in a Xilinx System Generator Subsystem, but there must be a System Generator token at the top level of the hierarchy.

```
open_system('hdlcoder_slsgen/S_LandSysGen/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.

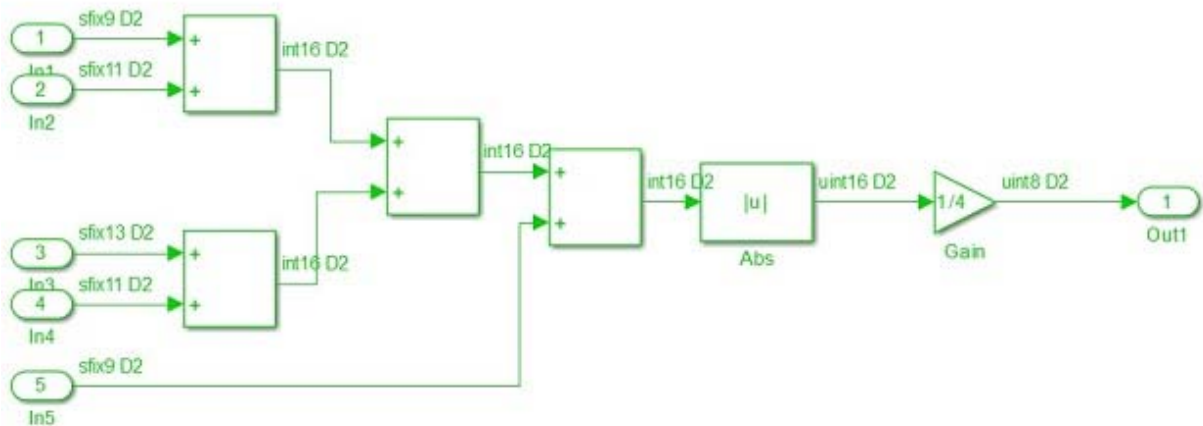
Gateway In blocks must not do non-trivial data type conversion. For example, a Gateway In block can convert between uint8 and UFix_8_0, but changing data sign, word length, or fraction length is not allowed.

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_s1sysgen/SLandSysGen');
```

You can also generate a testbench, simulate, and synthesize the design as you would for any other model.

Code Generation for HDL Cosimulation Blocks

The coder supports HDL code generation for the following HDL Cosimulation blocks:

- HDL Verifier for use with Mentor Graphics ModelSim
- HDL Verifier for use with Cadence Incisive

Each of the HDL Cosimulation blocks cosimulates a hardware component by applying input signals to, and reading output signals from, an HDL model that executes under an HDL simulator.

See the “Define HDL Cosimulation Block Interface” section of the HDL Verifier documentation for information on timing, latency, data typing, frame-based processing, and other issues that may be of concern to you when setting up an HDL cosimulation.

You can use an HDL Cosimulation block with the coder to generate an interface to your manually written or legacy HDL code. When an HDL Cosimulation block is included in a model, the coder generates a VHDL or Verilog interface, depending on the selected target language.

When the target language is VHDL, the generated interface includes:

- An entity definition. The entity defines ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. Clock enable and reset ports are also declared.
- An RTL architecture including a component declaration, a component configuration declaring signals corresponding to signals connected to the HDL Cosimulation ports, and a component instantiation.
- Port assignment statements as required by the model.

When the target language is Verilog, the generated interface includes:

- A module defining ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. The module also defines clock enable and reset ports, and wire declarations corresponding to signals connected to the HDL Cosimulation ports.

- A module instance.
- Port assignment statements as required by the model.

The requirements for using the HDL Cosimulation block for code generation are the same as those for cosimulation. If you want to check these conditions before initiating code generation, select **Simulation > Update Diagram**.

Generate a Cosimulation Model

In this section...

“What Is A Cosimulation Model?” on page 18-39

“Generating a Cosimulation Model from the GUI” on page 18-40

“Structure of the Generated Model” on page 18-46

“Launching a Cosimulation” on page 18-53

“The Cosimulation Script File” on page 18-55

“Complex and Vector Signals in the Generated Cosimulation Model” on page 18-58

“Generating a Cosimulation Model from the Command Line” on page 18-60

“Naming Conventions for Generated Cosimulation Models and Scripts” on page 18-61

“Limitations for Cosimulation Model Generation” on page 18-61

Note To use this feature, your installation must include an HDL Verifier license.

What Is A Cosimulation Model?

A cosimulation model is an automatically generated Simulink model configured for both Simulink simulation and cosimulation of your design with an HDL simulator. HDL Coder supports automatic generation of a cosimulation model as a part of the test bench generation process.

The cosimulation model includes:

- A behavioral model of your design, realized in a Simulink subsystem.
- A corresponding HDL Cosimulation block, configured to cosimulate the design using HDL Verifier. The 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 .

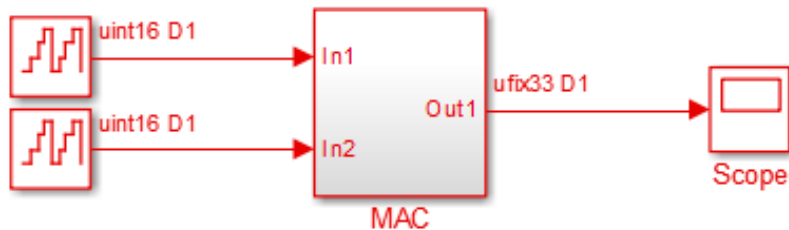
In addition to the generated model, the coder generates a TCL script that launches and configures your cosimulation tool. Comments in the script file document clock, reset, and other timing signal information defined by the coder for the cosimulation tool.

Generating a Cosimulation Model from the GUI

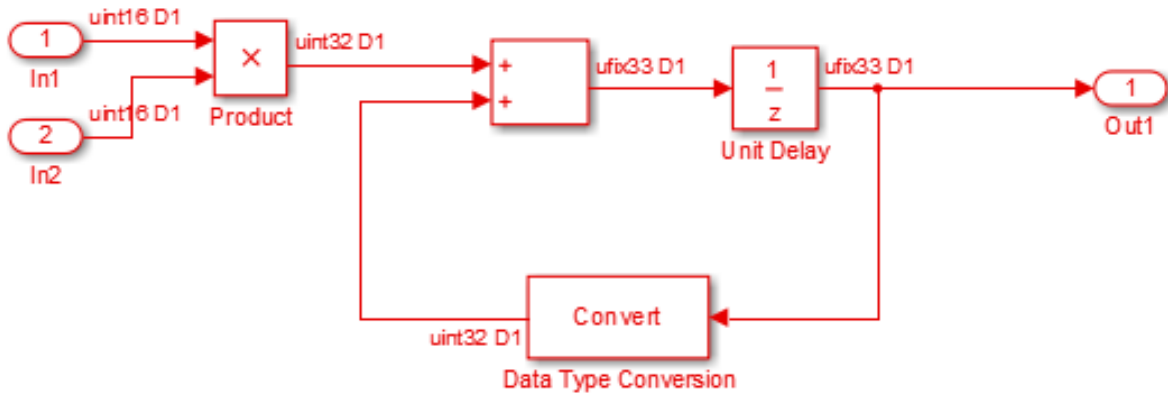
This example demonstrates the process for generating a cosimulation model. The example model, `hdl_cosim_demo1`, implements a simple multiply and accumulate (MAC) algorithm. Open the model by entering the name at the MATLAB command line:

```
hdl_cosim_demo1
```

The following figure shows the top-level model.

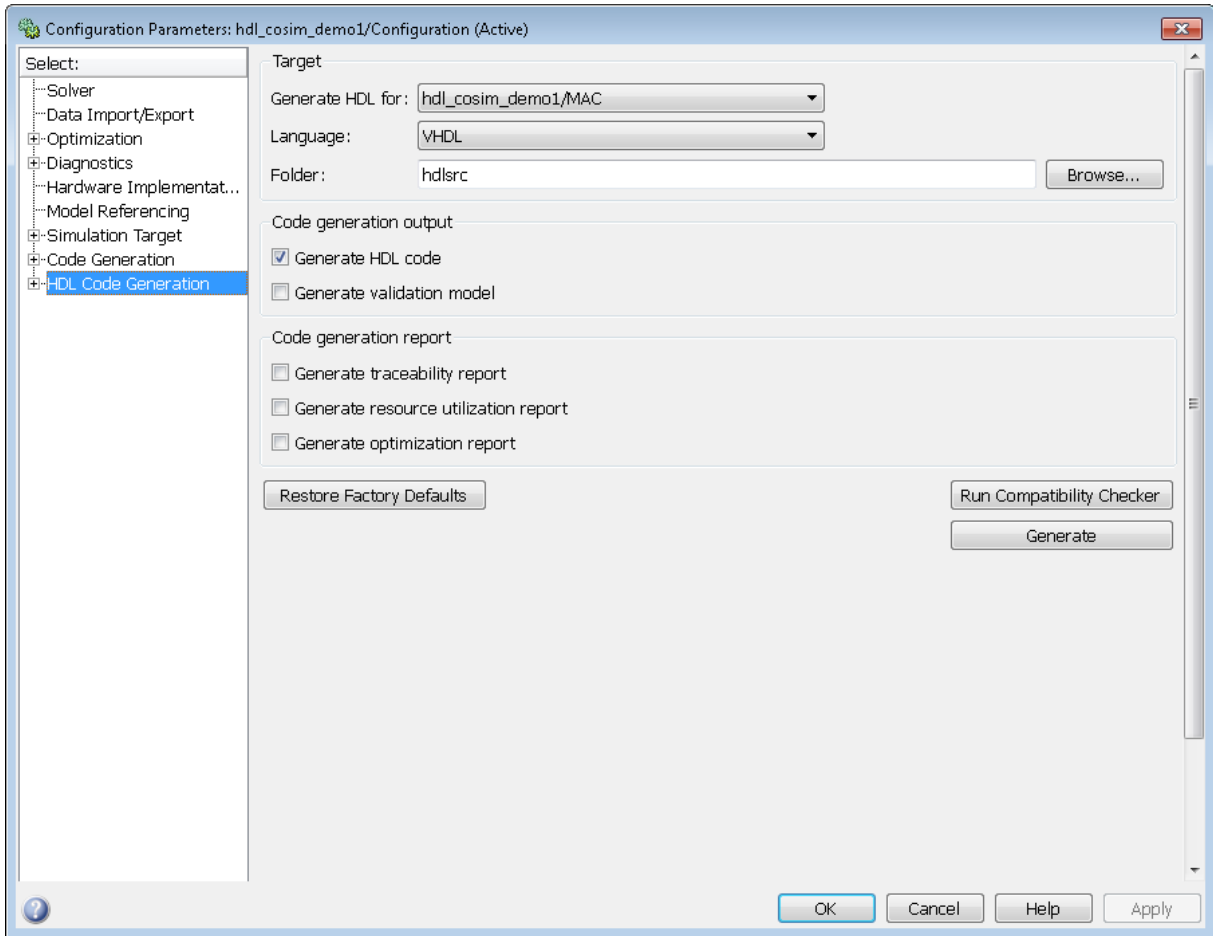


The DUT is the MAC subsystem.



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**. The coder displays progress messages, as shown in the following listing:

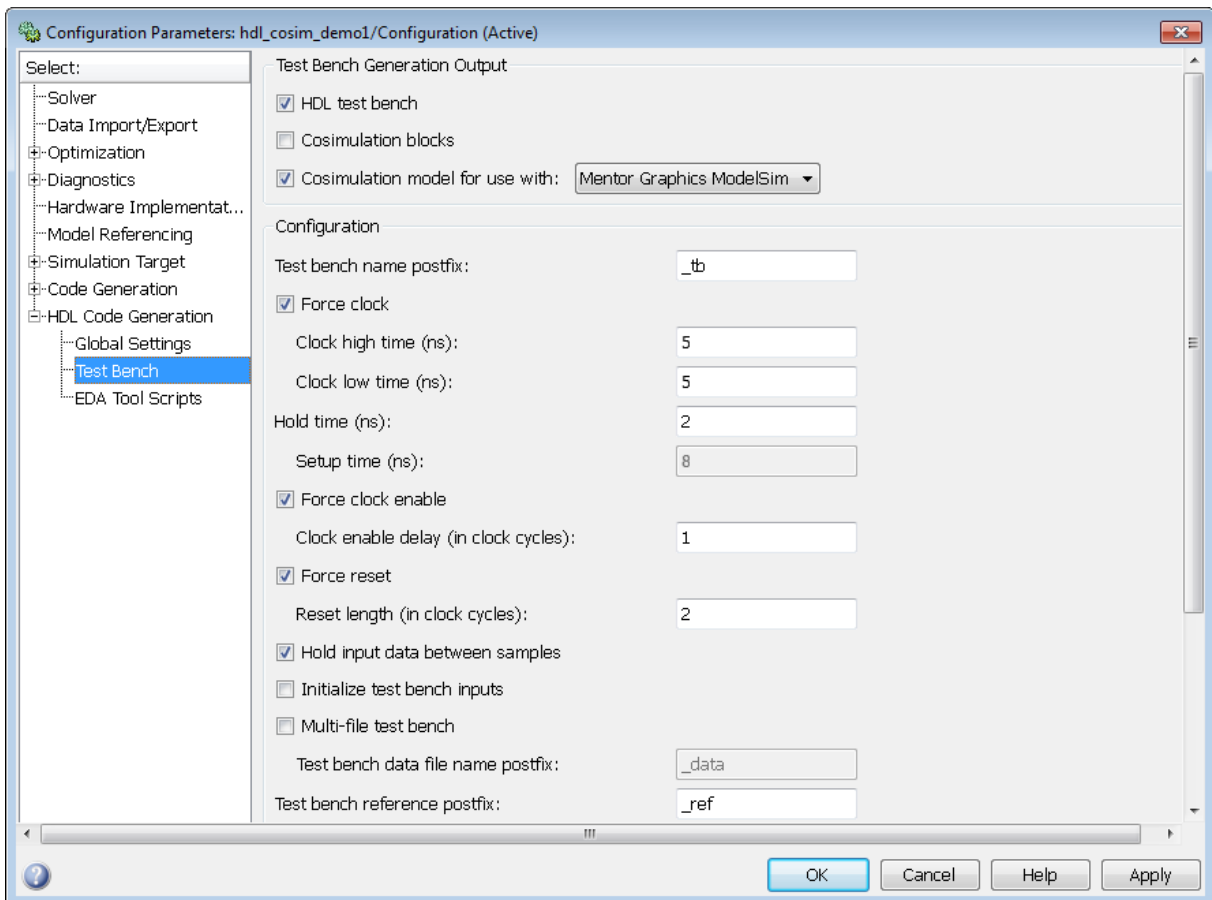
```
### Applying HDL Code Generation Control Statements
### Starting HDL Check.
### HDL Check Complete with 0 error, 0 warning and 0 message.

### Begin VHDL Code Generation
```

```
### Working on hd1_cosim_demo1/MAC as hd1src\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. The coder records option settings in a generated script file (see “The Cosimulation Script File” on page 18-55).
- 5 Click **Apply**.

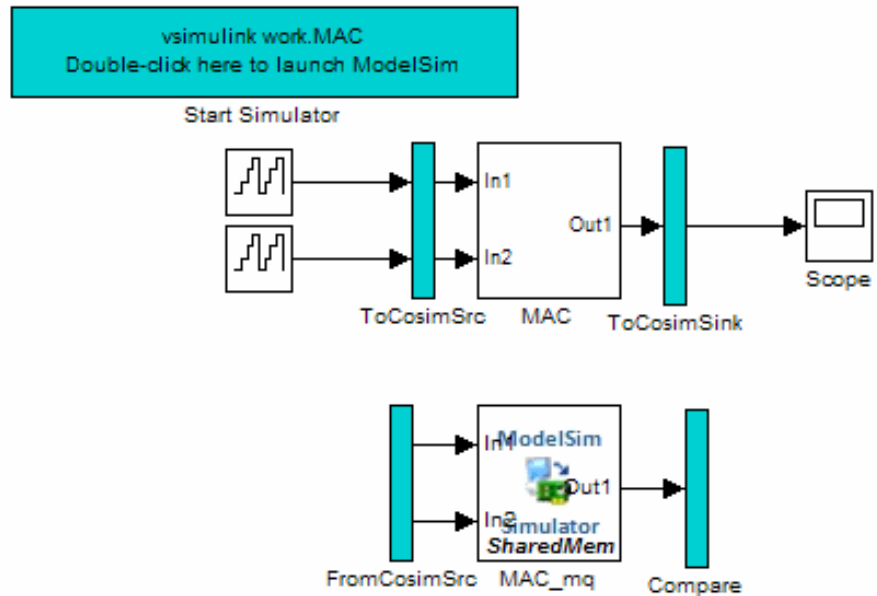
Next, generate test bench code and the cosimulation model:

- 1 Click **Generate Test Bench**. The 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, the 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, the 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 18-61.

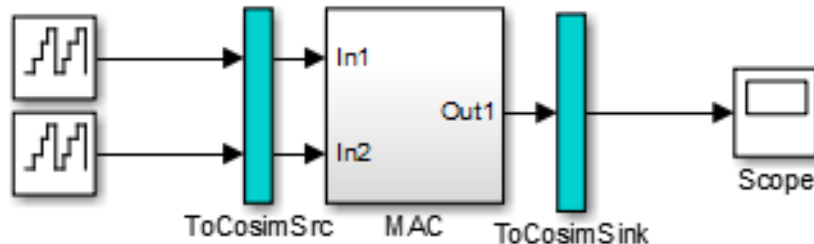
The next section, “Structure of the Generated Model” on page 18-46, describes the features of the model. Before running a cosimulation, become familiar with these features.

Structure of the Generated Model

You can set up and launch a cosimulation using controls located in the generated model. This section examines the model generated from the example MAC subsystem.

Simulation Path

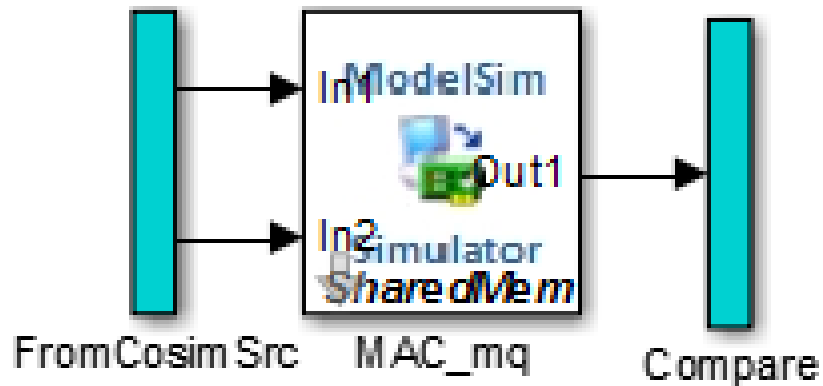
The model comprises two parallel signal paths. The *simulation path*, located in the upper half of the model window, is nearly identical to the original DUT. The purpose of the simulation path is to execute a normal Simulink simulation and provide a reference signal for comparison to the cosimulation results. The following figure shows the simulation path.



The two subsystems labelled `ToCosimSrc` and `ToCosimSink` do not change the performance of the simulation path. Their purpose is to capture stimulus and response signals of the DUT and route them to and from the HDL cosimulation block (see “Signal Routing Between Simulation and Cosimulation Paths” on page 18-49).

Cosimulation Path

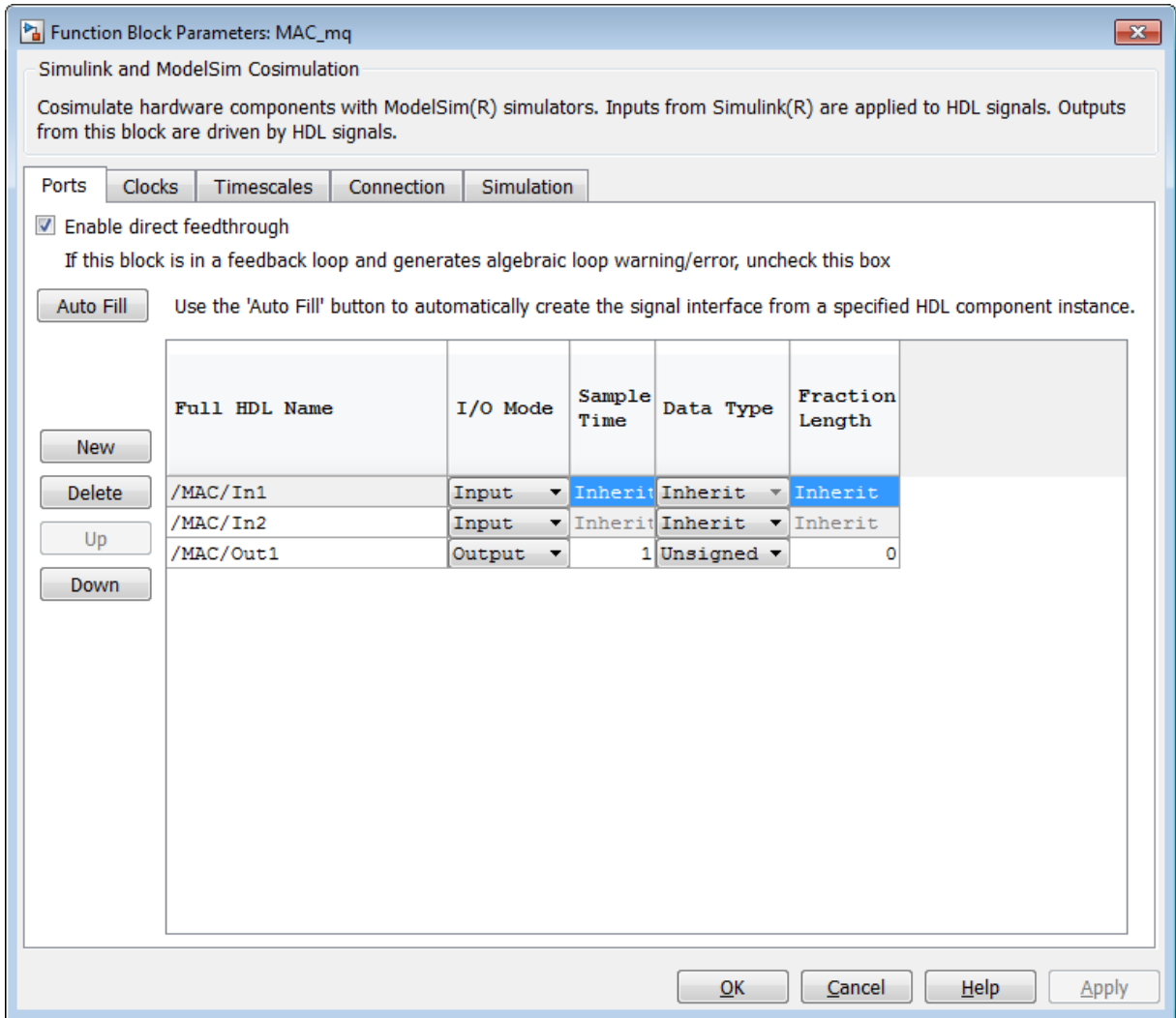
The *cosimulation path*, located in the lower half of the model window, contains the generated HDL Cosimulation block. The following figure shows the cosimulation path.



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 18-49).

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 18-51 for details.

The 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.

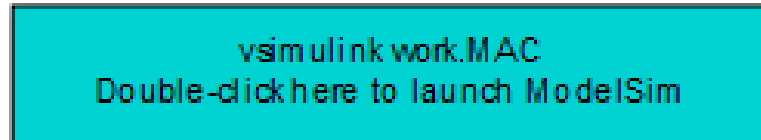


The coder sets the **Full HDL Name**, **Sample Time**, **Data Type**, and other fields as required by the model. The coder also configures other HDL Cosimulation block parameters under the **Timescales** and **Tcl** panes.

Tip The 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.



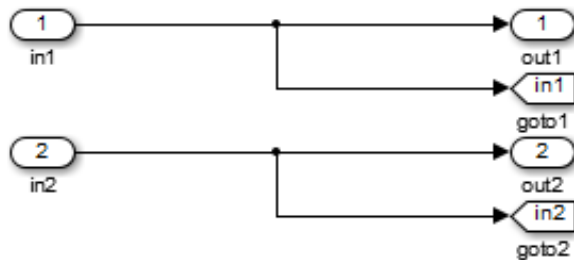
Start Simulator

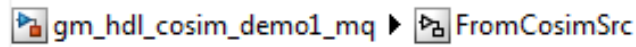
The commands executed when you double-click the Start Simulator icon launch and set up the cosimulation tool, but they do not start the actual cosimulation. “Launching a Cosimulation” on page 18-53 describes how to run a cosimulation with the generated model.

Signal Routing Between Simulation and Cosimulation Paths

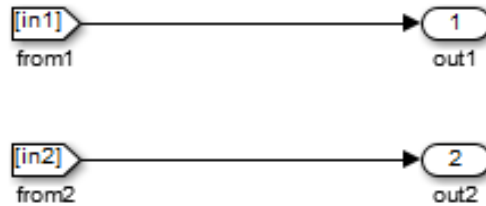
The generated model routes signals between the simulation and cosimulation paths using Goto and From blocks. For example, the Goto blocks in the ToCosimSrc subsystem route each DUT input signal to a corresponding From block in the FromCosimSrc subsystem. The following figures show the Goto and From blocks in each subsystem.

gm_hdl_cosim_demo1_mq ▶ ToCosimSrc





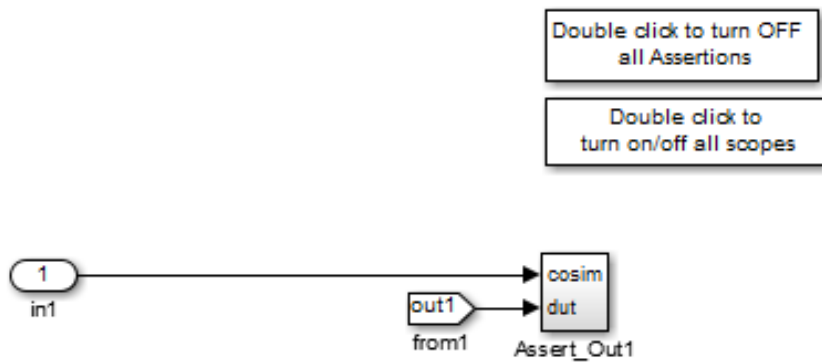
gm_hdl_cosim_demo1_mq ▶ FromCosimSrc



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 18-58 for further information.

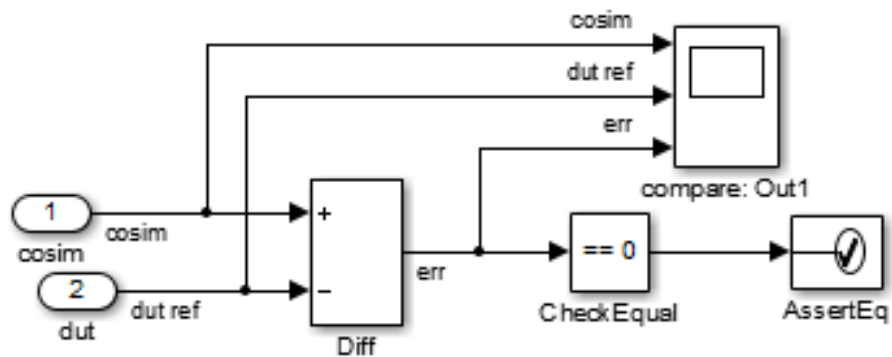
Controlling Assertions and Scope Displays

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, the 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.

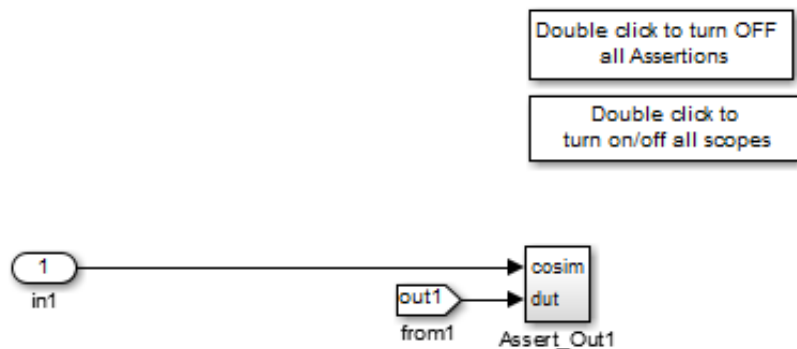
Tip Assertion messages are warnings and do not stop simulation.

Launching a Cosimulation

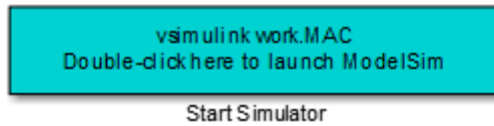
To run a cosimulation with the generated model:

- 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.

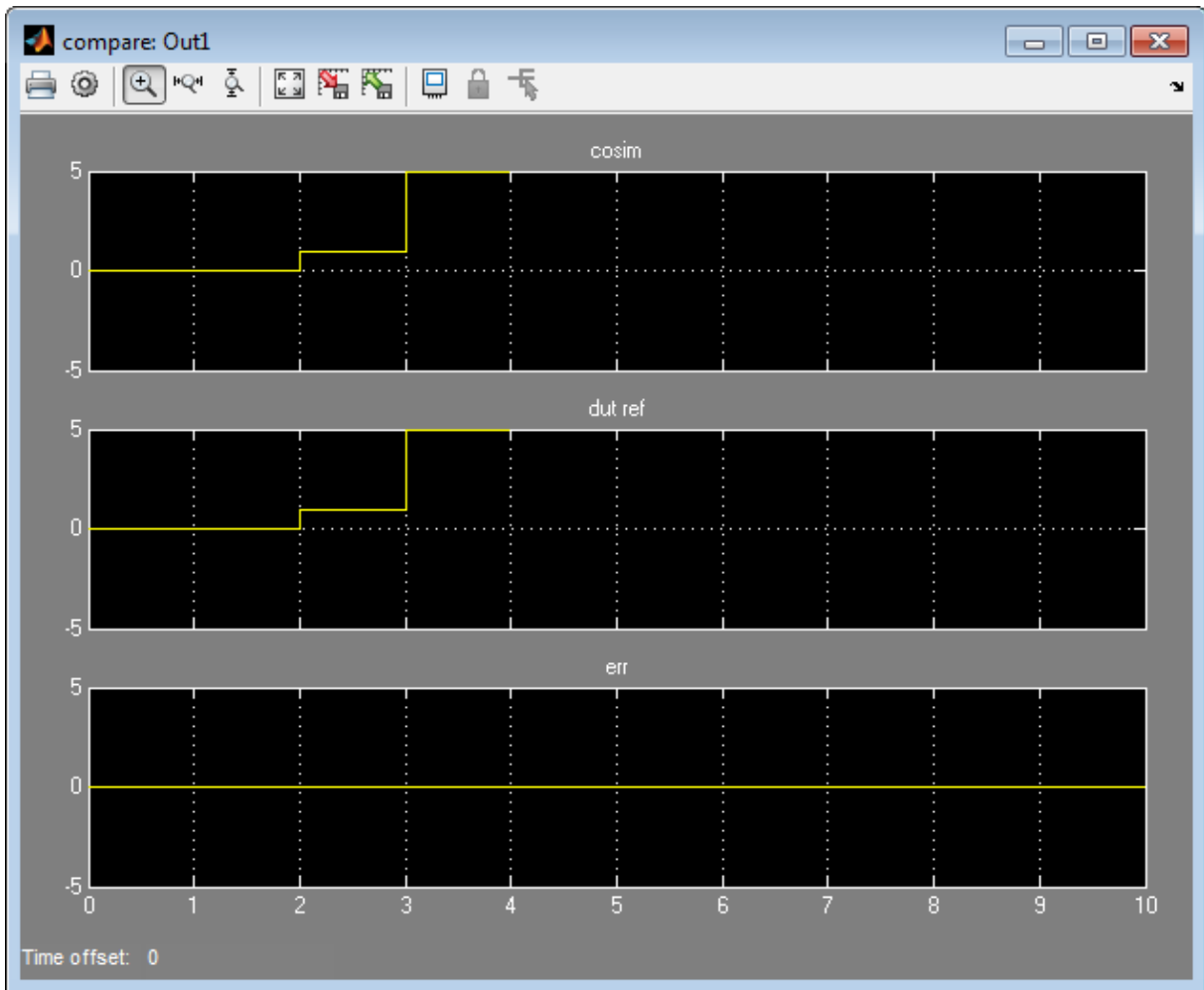
As the cosimulation runs, the HDL simulator displays messages like the following.

```
# 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. The 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:

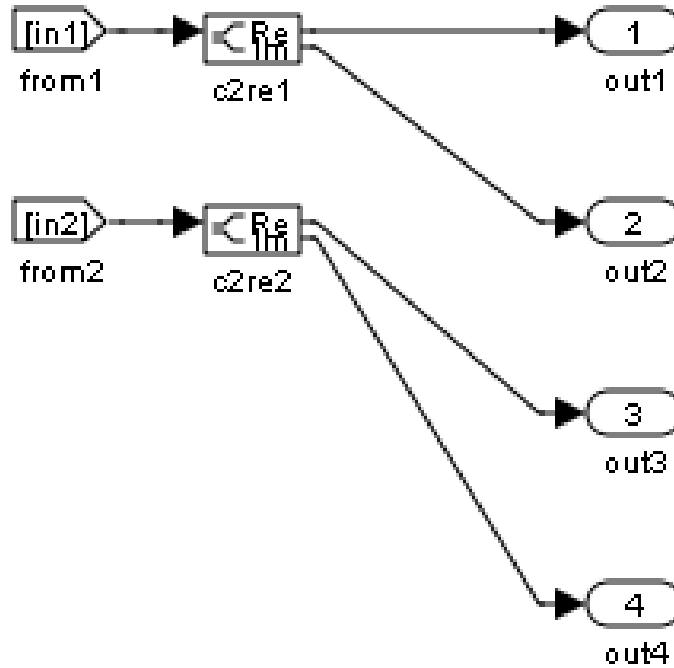
```
function tclCmds = gm_hdl_cosim_demo1_mq_tcl
tclCmds = {
    'do MAC_compile.do',...% Compile the generated code
    'vsimulink work.MAC',...% Initiate cosimulation
    'add wave /MAC/c1k',...% Add wave commands for chip input signals
    'add wave /MAC/reset',...
    'add wave /MAC/c1k_enable',...
    '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

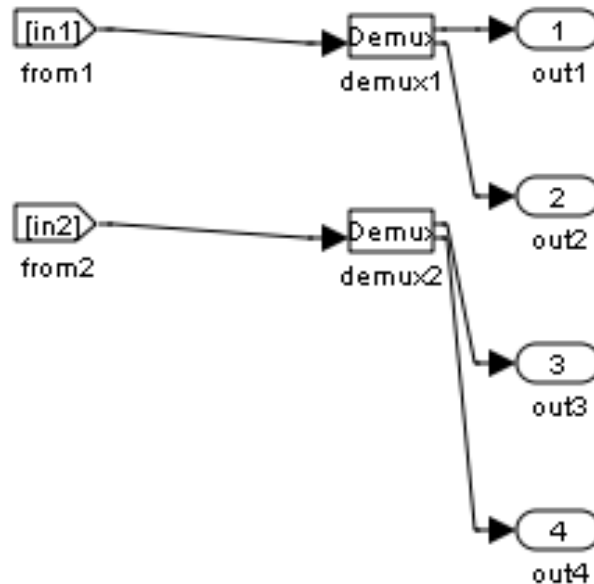
The generated cosimulation model automatically breaks complex inputs into real and imaginary parts. The following figure shows a `FromCosimSrc` subsystem that receives two complex input signals. The subsystem breaks the inputs into real and imaginary parts before passing them to the subsystem outputs.



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 coder 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. For more information, see “Direct Feedthrough Cosimulation”.

Alternatively, you can avoid the latency by specifying output pipelining (see “OutputPipeline” on page 11-80). 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.

Customize the Generated Interface

You can customize port names and set attributes of the external component when you generate an interface from the following block types:

- Model
- Subsystem
- HDL Cosimulation

Open the HDL 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 <code>ClockEnableInputPort</code> .
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 <code>ClockInputPort</code> .
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 <code>ResetInputPort</code> .
AllowDistributedPipelining	'on' 'off' Default: 'off'	If 'on', allow the coder to move registers across the block, from input to output or output to input.
ClockEnableInputPort	Default: 'clk_enable'	Specifies HDL name for block's clock enable input port.

Parameter Name	Values	Description
ClockInputPort	Default: 'clk'	Specifies HDL name for block's clock input signal.
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 configurations for the block, and require a user-supplied external configuration.

Parameter Name	Values	Description
InputPipeline	Default: '0'	Specifies the number of input pipeline stages (pipeline depth) in the generated code.
OutputPipeline	Default: '0'	Specifies the number of output pipeline stages (pipeline depth) in the generated code.
ResetInputPort	Default: 'reset'	Specifies HDL name for block's reset input.
VHDLArchitectureName (VHDL only)	Default: 'rtl'	Specifies RTL architecture name generated for the block. The architecture name is generated only if InlineConfigurations = 'on'.
VHDLComponentLibrary (VHDL only)	Default: 'work'	Specifies the library from which to load the VHDL component.

Pass-Through and No-Op Implementations

The 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. The 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 19-2
- “Requirements for Stateflow HDL Code Generation” on page 19-4
- “Map Chart Semantics to HDL” on page 19-9
- “Generate HDL for Mealy and Moore Finite State Machines” on page 19-13
- “Structure a Model for HDL Code Generation” on page 19-24
- “Design Patterns Using Advanced Chart Features” on page 19-25

Introduction to Stateflow HDL Code Generation

In this section...
“Overview” on page 19-2
“Examples” on page 19-2

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.

Examples

The following examples, illustrating HDL code generation from subsystems that include Stateflow charts, are available:

- Greatest Common Divisor
- Pipelined Configurable FIR
- 2D FDTD Behavioral Model

- CPU Behavioral Model

To open the example models, open the **Help**, navigate to the HDL Coder documentation, and click **Examples**.

Requirements for Stateflow HDL Code Generation

In this section...

“Overview” on page 19-4
“Location of Charts in the Model” on page 19-4
“Data Type Usage” on page 19-4
“Chart Initialization” on page 19-5
“Registered Output” on page 19-5
“Restrictions on Imported Code” on page 19-6
“Using Input and Output Events” on page 19-6
“Using For Loops” on page 19-6
“Other Restrictions” on page 19-7

Overview

This section summarizes the requirements and restrictions you should follow when configuring Stateflow charts that are intended to target HDL code generation. “Map Chart Semantics to HDL” on page 19-9 provides a more detailed rationale for most of these requirements.

Location of Charts in the Model

A chart intended for HDL code generation must be part of a Simulink subsystem. See “Structure a Model for HDL Code Generation” on page 19-24 for an example.

Data Type Usage

Supported Data Types

The current release supports a subset of MATLAB data types in charts intended for use in HDL code generation. Supported data types are

- Signed and unsigned integer

- Double and single

Note Results obtained from HDL code generated for models using double or single data types might not be bit-true to results obtained from simulation of the original model.

- Fixed point
- Boolean

Note Multidimensional arrays of these types are supported, with the exception of data types assigned to ports. Port data types must be either scalar or vector.

Chart Initialization

In charts intended for HDL code generation, enable the chart property **Execute (enter) Chart at Initialization**. When this property is enabled, default transitions are tested and actions reachable from the default transition taken are executed. These actions correspond to the reset process in HDL code. “Execution of a Chart at Initialization” describes existing restrictions under this property.

The reset action must not entail the delay of combinatorial logic. Therefore, do not perform arithmetic in initialization actions.

The chart property **Initialize Outputs Every Time Chart Wakes Up** controls whether or not output is persistent.

Selecting **Initialize Outputs Every Time Chart Wakes Up** generates HDL code that is more readable and has better synthesis results.

Registered Output

If you want to insert an output register that delays the chart output by a simulation cycle, use the OutputPipeline block property.

Restrictions on Imported Code

A chart intended for HDL code generation must be entirely self-contained. The following restrictions apply:

- Do not call MATLAB functions other than `min` or `max`.
- Do not use MATLAB workspace data.
- Do not call C math functions
- If the **Enable C-like bit operations** property is disabled, do not use the exponentiation operator (^). The exponentiation operator is implemented with the C Math Library function `pow`.
- Do not include custom code. Information entered in the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box is ignored.

Using Input and Output Events

The coder supports the use of input and output events with Stateflow charts, subject to the following constraints:

- You can define and use one and only one input event per Stateflow chart. (There is no restriction on the number of output events you can use.)
- The coder does not support HDL code generation for charts that have a single input event, and which also have nonzero initial values on the chart's output ports.
- All input and output events must be edge-triggered.

For detailed information on input and output events, see “Activate a Stateflow Chart Using Input Events” and “Activate a Simulink Block Using Output Events” in the Stateflow documentation.

Using For Loops

Do not explicitly use loops other than `for` loops in a chart intended for HDL code generation. Observe the following restrictions on `for` loops:

- The data type of the loop counter variable must be `int32`.
- The coder supports only constant-bounded loops.

The for loop example (sf_for) illustrates a design pattern for a for loop using a graphical function.

Other Restrictions

The coder imposes a number of additional restrictions on the use of classic chart features. These limitations exist because HDL does not support some features of general-purpose sequential programming languages.

- Do not define local events in a chart from which HDL code is to be generated.

Do not use the following implicit events:

- enter
- exit
- change

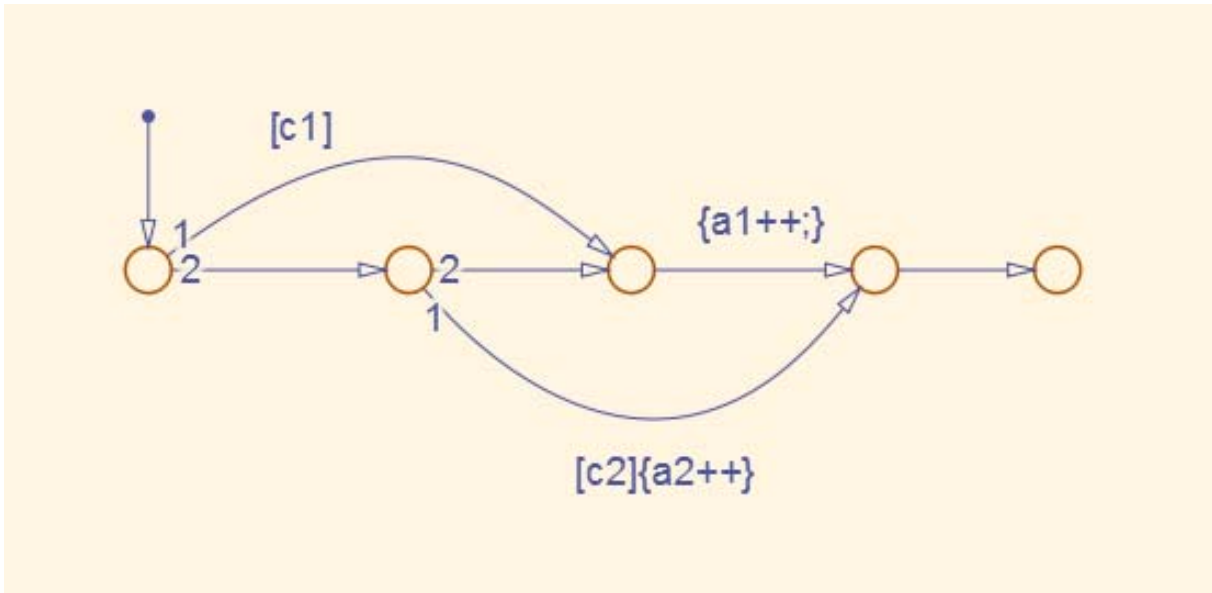
You can use the following implicit events:

- wakeup
- tick

Temporal logic can be used provided the base events are limited to these types of implicit events.

Note Absolute-time temporal logic is not supported for HDL code generation.

- Do not use recursion through graphical functions. The coder does not currently support recursion.
- HDL does not support a goto statement. Therefore, do not use unstructured flow diagrams, such as the flow diagram shown in the following figure.



- Do not read from output ports if you do not have the **Initialize Outputs Every Time Chart Wakes Up** chart option selected.
- Do not use Data Store Memory objects.
- Do not use pointer (&) or indirection (*) operators. See the discussion of “Pointer and Address Operations”.
- If a chart gets a runtime overflow error during simulation, it is possible to disable data range error checking and generate HDL code for the chart. However, in such cases results obtained from the generated HDL code might not be bit-true to results obtained from the simulation. Recommended practice is to enable overflow checking and eliminate overflow conditions from the model during simulation.

Map Chart Semantics to HDL

In this section...
“Hardware Realization of Stateflow Semantics” on page 19-9
“Restrictions for HDL Realization” on page 19-11

Hardware Realization of Stateflow Semantics

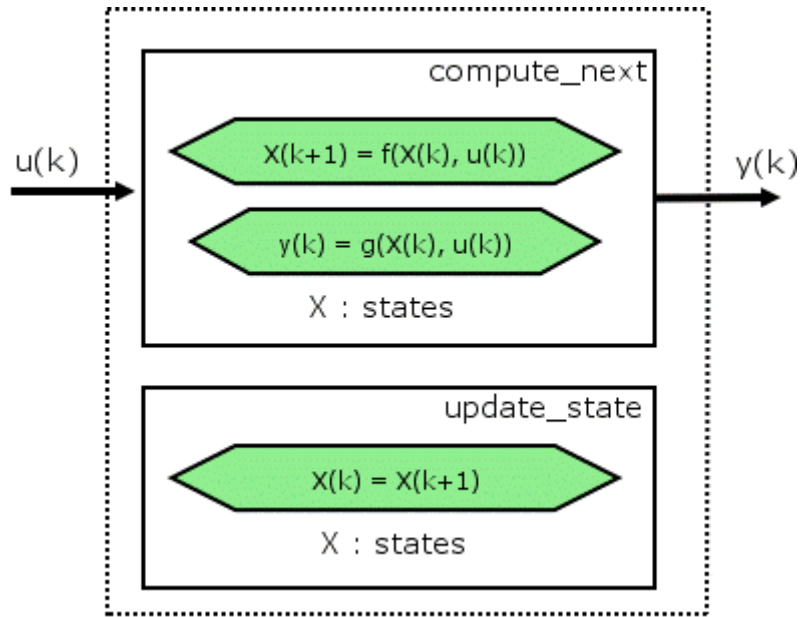
A mapping from Stateflow semantics to an HDL implementation has the following requirements:

- **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 speculatively 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.

In practice, output computations usually occur more often than state updates. The presence of inputs drives the computation of outputs. State transitions occur at regular intervals (whenever the chart is activated).

The following diagram shows a concurrent implementation of Stateflow semantics for output and update computations, intended for targeting HDL.



The HDL code generator reuses the original single-function implementation of Stateflow semantics almost without modification. There is one important difference: instead of computing with state variables directly, state computations are performed on local shadow variables. These variables are local to the HDL function `update_chart`. At the beginning of the `update_chart` functions, `current_state` is copied into the shadow variables. At the end of the `update_chart` function, the newly computed state is transferred to registers called collectively `next_state`. The values held in these registers are copied to `current_state` (also registered) when `update_state` is called.

By using local variables, this approach maps Stateflow sequential semantics to HDL sequential statements, avoiding the use of concurrent statements. For instance, 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.

Restrictions for HDL Realization

Some restrictions on chart usage are required to achieve a valid mapping from a chart to HDL code. These are summarized briefly in “Requirements for Stateflow HDL Code Generation” on page 19-4. The following sections give a more detailed rationale for most of these restrictions.

Self-Contained Charts

The Stateflow C target allows generated code to have some dependencies on code or data that is external to the chart. Stateflow charts intended for HDL code generation, however, must be self-contained. Observe the following rules for creating self-contained charts:

- Do not use C math functions such as `sin` and `pow`. There is no HDL counterpart to the C math library.
- Do not use calls to functions coded in a language other than HDL. For example, do not call MATLAB functions for a simulation target, as in the following statement:

```
ml disp( hello )
```

- Do not use custom code. There is no mechanism for embedding external HDL code into generated HDL code. Custom C code (user-written C code intended for linkage with C code generated from a Stateflow chart) is ignored during HDL code generation.

See also “External Component Interfaces”.

- Do not use pointer (`&`) or indirection (`*`) operators. Pointer and indirection operators have no function in a chart in the absence of custom code. Also, pointer and indirection operators do not map directly to synthesizable HDL.
- Do not share data (via Data Store Memory blocks) between charts. The coder does not map such global data to HDL, because HDL does not support global data.

Charts Must Not Use Features Unsupported by HDL

When creating charts intended for HDL code generation, follow these guidelines to avoid using Stateflow features that cannot be mapped to HDL:

- Avoid recursion. While charts permit recursion (through both event processing and user-written recursive graphical functions), HDL does not allow recursion.
- Do not use Stateflow and local events. These event types do not have equivalents in HDL. Therefore, these event types are not supported for HDL code generation.
- Avoid unstructured code. Although charts allow unstructured code to be written (through transition flow diagrams and graphical functions), this usage results in `goto` statements and multiple function return statements. HDL does not support either `goto` statements or multiple function return statements.
- Select the **Execute (enter) Chart At Initialization** chart property. This option executes the update chart function immediately following chart initialization. The option is required for HDL because outputs must be available at time 0 (hardware reset). You must select this option for bit-true HDL code generation.

Generate HDL for Mealy and Moore Finite State Machines

In this section...
“Overview” on page 19-13
“Generating HDL for a Mealy Finite State Machine” on page 19-14
“Generating HDL Code for a Moore Finite State Machine” on page 19-18

Overview

Stateflow charts support modeling of three types of state machines:

- Classic (default)
- Mealy
- Moore

This section discusses issues you should consider when generating HDL code for Mealy and Moore state machines.

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.

Moore and Mealy state charts can be functionally equivalent; an equivalent Mealy chart can derive from a Moore chart, and vice versa. A Mealy state machine has a richer description and usually requires a smaller number of states.

The principal advantages of using Mealy or Moore charts as an alternative to Classic charts are:

- At compile time, Mealy and Moore charts are validated for conformance to their formal definitions and semantic rules, and violations are reported.

- Moore charts generate more efficient code than Classic charts, for both C and HDL targets.

The execution of a Mealy or Moore chart at time t is the evaluation of the function represented by that chart at time t . The initialization property for output defines every output at every time step. Specifically, the output of a Mealy or Moore chart at one time step must not depend on the output of the chart at an earlier time step.

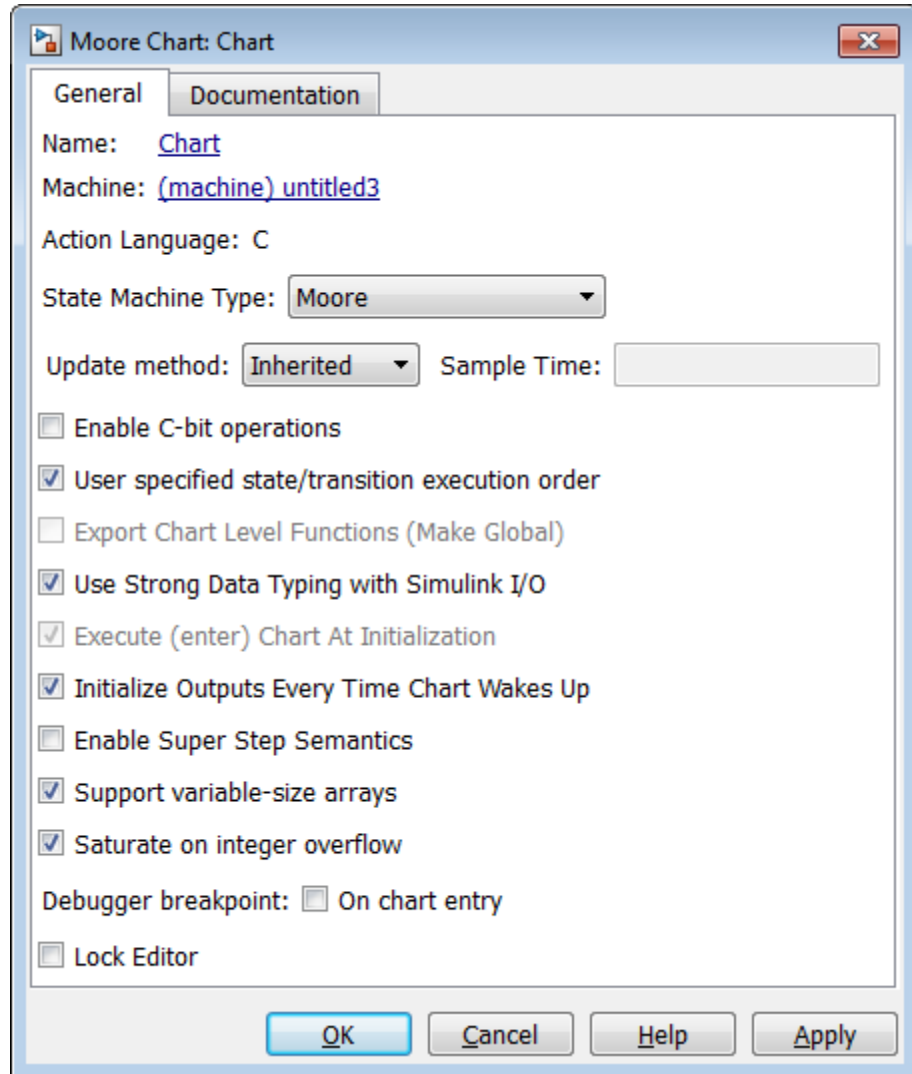
Consider the outputs of a chart. Stateflow charts permit output latching. That is, the value of an output computed at time t persists until time $t+d$, when it is overwritten. The output latching feature corresponds to registered outputs. Therefore, Mealy and Moore charts intended for HDL code generation should not use registered outputs.

Generating HDL for a Mealy Finite State Machine

When generating HDL code for a chart that models a Mealy state machine, make sure that:

- The chart meets the general code generation requirements, as described in “Requirements for Stateflow HDL Code Generation” on page 19-4.
- Actions are associated with inner and outer transitions only.

In addition, for better synthesis results and more readable HDL code, we recommend selecting the chart property **Initialize Outputs Every Time Chart Wakes Up**, as shown in the following figure.



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,


```
end

always @* begin
    is_MealyChart_next = is_MealyChart;
    seqFound_1 = 1'b0;
    state_1 = 0.0;

    case ( is_MealyChart)
        IN_s0 :
            begin
                if (u_double == 1.0) begin
                    state_1 = 1.0;
                    is_MealyChart_next = IN_s1;
                end
            end
        IN_s1 :
            begin
                if (u_double == 2.0) begin
                    state_1 = 2.0;
                    is_MealyChart_next = IN_s12;
                end
                else if (u_double != 1.0) begin
                    state_1 = 0.0;
                    is_MealyChart_next = IN_s0;
                end
            end
        IN_s12 :
            begin
                if (u_double == 1.0) begin
                    state_1 = 3.0;
                    is_MealyChart_next = IN_s121;
                end
                else begin
                    state_1 = 0.0;
                    is_MealyChart_next = IN_s0;
                end
            end
        IN_s121 :
            begin
                if (u_double == 1.0) begin
```

```
        state_1 = 1.0;
        is_MealyChart_next = IN_s1;
    end
    else if (u_double == 3.0) begin
        state_1 = 4.0;
        seqFound_1 = 1'b1;
        is_MealyChart_next = IN_s1213;
    end
    else if (u_double == 2.0) begin
        state_1 = 2.0;
        is_MealyChart_next = IN_s12;
    end
    else begin
        state_1 = 0.0;
        is_MealyChart_next = IN_s0;
    end
end
default :
begin
    if (u_double == 1.0) begin
        state_1 = 1.0;
        seqFound_1 = 1'b0;
        is_MealyChart_next = IN_s1;
    end
    else begin
        state_1 = 0.0;
        seqFound_1 = 1'b0;
        is_MealyChart_next = IN_s0;
    end
end
endcase

end
```

Generating HDL Code for a Moore Finite State Machine

When generating HDL code for a chart that models a Moore state machine, make sure that:

- The chart meets the general code generation requirements, as described in “Requirements for Stateflow HDL Code Generation” on page 19-4.
- Actions occur in states only. These actions are unlabeled, and execute when exiting the states or remaining in the states.

Moore actions must be associated with states, because output computation must be dependent only on states, not input. Therefore, the current configuration of active states at time step t determines output. Thus, the single action in a Moore state serves as both during and exit action. If state S is active when a chart wakes up at time t , it contributes to the output whether it remains active into time $t+1$ or not.

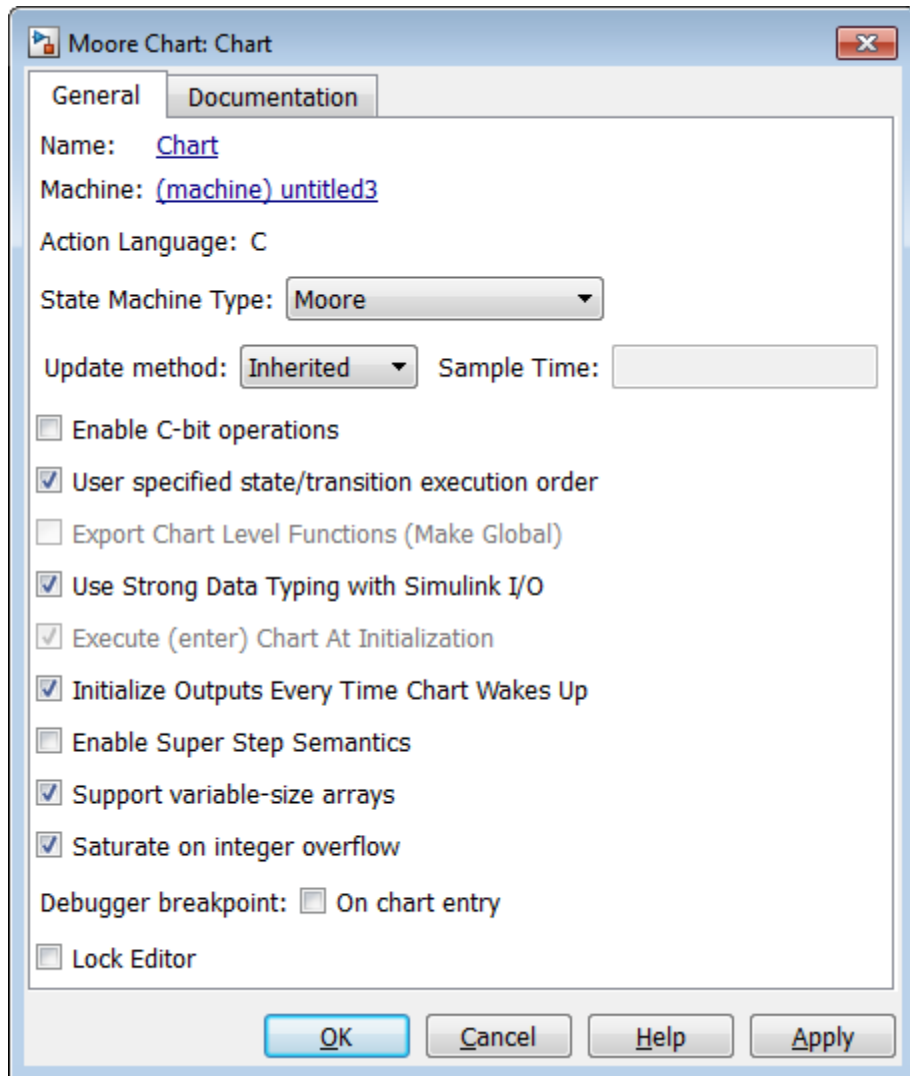
- Local data and graphical functions are not used.

Function calls and local data are not allowed in a Moore chart. This prevents output from depending on input in ways that would be difficult for the HDL code generator to verify. These restrictions strongly encourage coding practices that separate output and input.

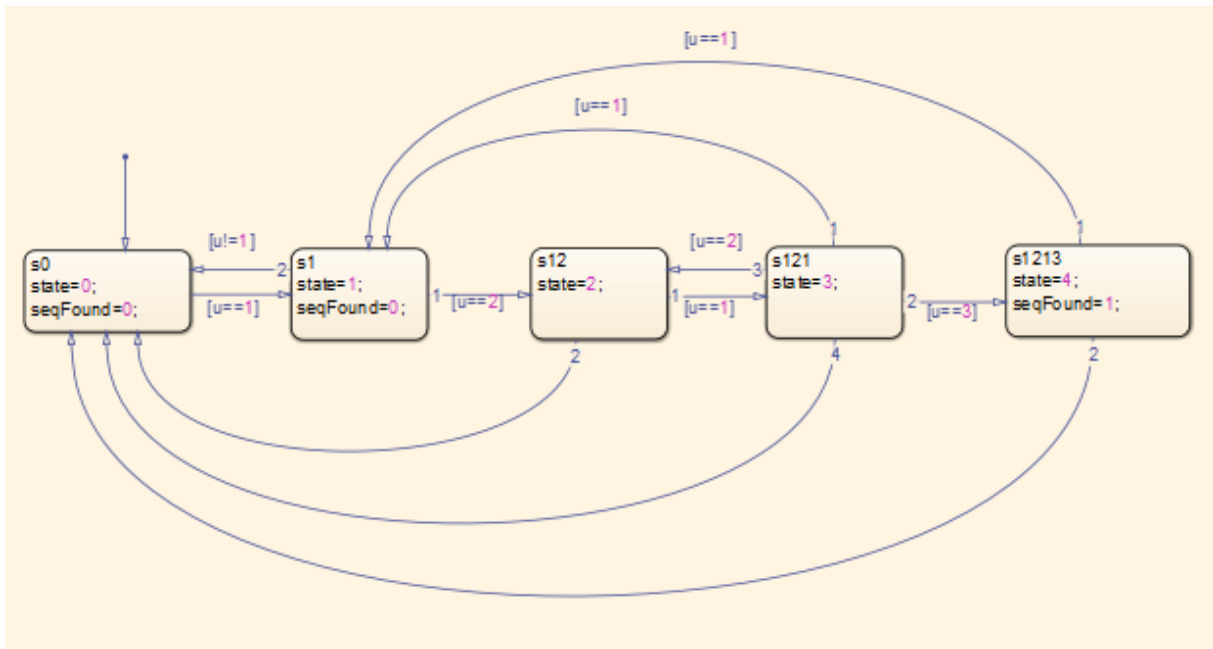
- No references to input occur outside of transition conditions.
- Output computation occurs only in leaf states.

The chart’s top-down semantics compute outputs as if actions were evaluated strictly before inner and outer flow diagrams.

In addition, for better synthesis results and more readable HDL code, we recommend selecting the chart property **Initialize Outputs Every Time Chart Wakes Up**, as shown in the following figure.



The following figure shows a Stateflow chart of a Moore state machine.



The following code example illustrates generated VHDL code for the Moore chart.

```

MooreChart_1_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    is_MooreChart <= IN_s0;
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      is_MooreChart <= is_MooreChart_next;
    END IF;
  END IF;
END PROCESS MooreChart_1_process;

MooreChart_1_output : PROCESS (is_MooreChart, u)
BEGIN
  is_MooreChart_next <= is_MooreChart;
  seqFound <= '0';

```

```
state <= 0.0;

CASE is_MooreChart IS
  WHEN IN_s0 =>
    state <= 0.0;
    seqFound <= '0';
  WHEN IN_s1 =>
    state <= 1.0;
    seqFound <= '0';
  WHEN IN_s12 =>
    state <= 2.0;
  WHEN IN_s121 =>
    state <= 3.0;
  WHEN OTHERS =>
    state <= 4.0;
    seqFound <= '1';
END CASE;

CASE is_MooreChart IS
  WHEN IN_s0 =>
    IF u = 1.0 THEN
      is_MooreChart_next <= IN_s1;
    END IF;
  WHEN IN_s1 =>
    IF u = 2.0 THEN
      is_MooreChart_next <= IN_s12;
    ELSIF u /= 1.0 THEN
      is_MooreChart_next <= IN_s0;
    END IF;
  WHEN IN_s12 =>
    IF u = 1.0 THEN
      is_MooreChart_next <= IN_s121;
    ELSE
      is_MooreChart_next <= IN_s0;
    END IF;
  WHEN IN_s121 =>
    IF u = 1.0 THEN
      is_MooreChart_next <= IN_s1;
    ELSIF u = 3.0 THEN
```

```
        is_MooreChart_next <= IN_s1213;
    ELSIF u = 2.0 THEN
        is_MooreChart_next <= IN_s12;
    ELSE
        is_MooreChart_next <= IN_s0;
    END IF;
WHEN OTHERS =>
    IF u = 1.0 THEN
        is_MooreChart_next <= IN_s1;
    ELSE
        is_MooreChart_next <= IN_s0;
    END IF;
END CASE;

END PROCESS MooreChart_1_output;
```

Structure a Model for HDL Code Generation

In general, generation of VHDL or Verilog code from a model containing a Stateflow chart does not differ greatly from HDL code generation from other models.

A chart intended for HDL code generation *must* be part of a subsystem that represents the Device Under Test (DUT). The DUT corresponds to the top level VHDL entity or Verilog module for which code is generated, tested and eventually synthesized. The top level Simulink components that drive the DUT correspond to the behavioral test bench.

You may need to restructure your models to meet this requirement. If the chart for which you want to generate code is at the root level of your model, embed the chart in a subsystem and connect the relevant signals to the subsystem inputs and outputs. In most cases, you can do this by simply clicking on the chart and then selecting **Diagram > Subsystem & Model Reference > Create Subsystem from Selection** in the model window.

Design Patterns Using Advanced Chart Features

In this section...

“Temporal Logic” on page 19-25

“Graphical Function” on page 19-27

“Hierarchy and Parallelism” on page 19-29

“Stateless Charts” on page 19-30

“Truth Tables” on page 19-32

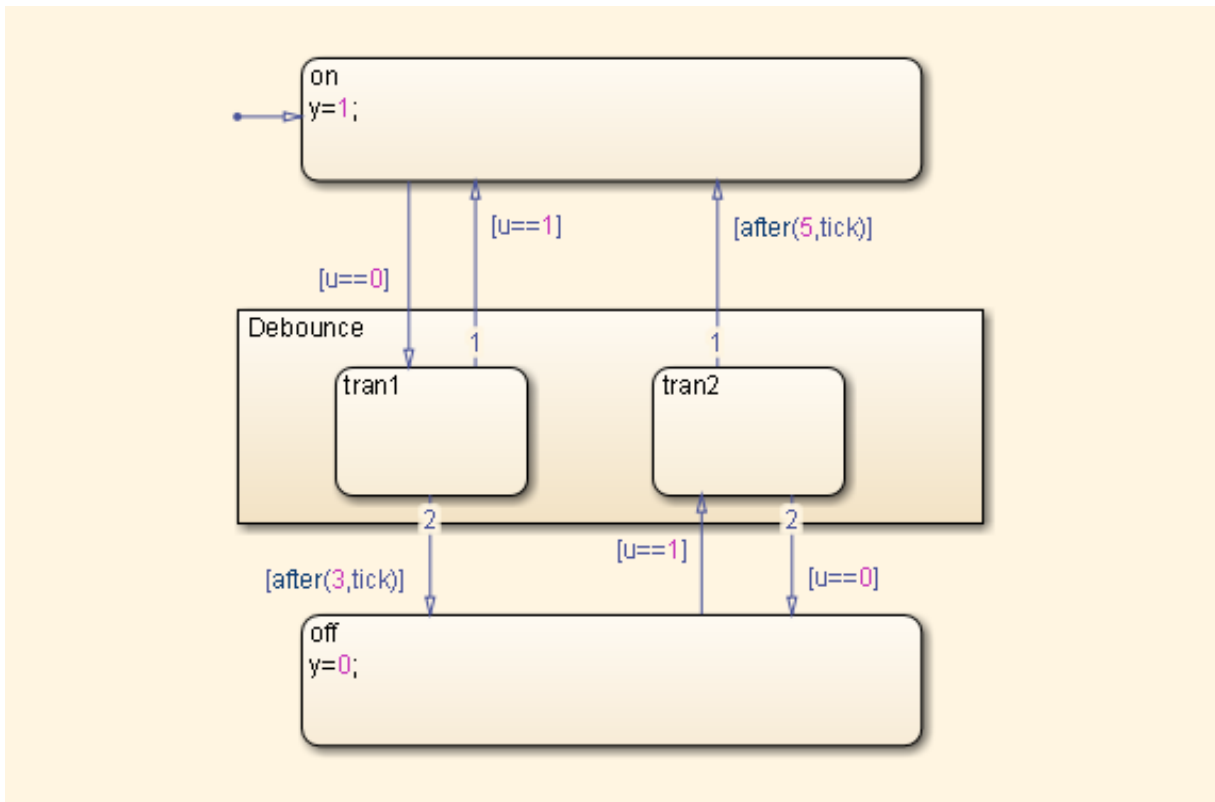
Temporal Logic

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 DOWNT0 0);
    BEGIN
        temporalCounter_i1_temp := temporalCounter_i1;
        is_Chart_next <= is_Chart;
        y_reg_next <= y_reg;
        IF temporalCounter_i1 < 7 THEN
            temporalCounter_i1_temp := temporalCounter_i1 + 1;
        END IF;

        CASE is_Chart IS
            WHEN IN_tran1 =>
    
```

```

IF u = 1.0 THEN
    is_Chart_next <= IN_on;
    y_reg_next <= 1.0;
ELSIF temporalCounter_i1_temp >= 3 THEN
    is_Chart_next <= IN_off;
    y_reg_next <= 0.0;
END IF;
WHEN IN_tran2 =>
    IF temporalCounter_i1_temp >= 5 THEN
        is_Chart_next <= IN_on;
        y_reg_next <= 1.0;
    ELSIF u = 0.0 THEN
        is_Chart_next <= IN_off;
        y_reg_next <= 0.0;
    END IF;
WHEN IN_off =>
    IF u = 1.0 THEN
        is_Chart_next <= IN_tran2;
        temporalCounter_i1_temp := to_unsigned(0, 8);
    END IF;
WHEN OTHERS =>
    IF u = 0.0 THEN
        is_Chart_next <= IN_tran1;
        temporalCounter_i1_temp := to_unsigned(0, 8);
    END IF;
END CASE;

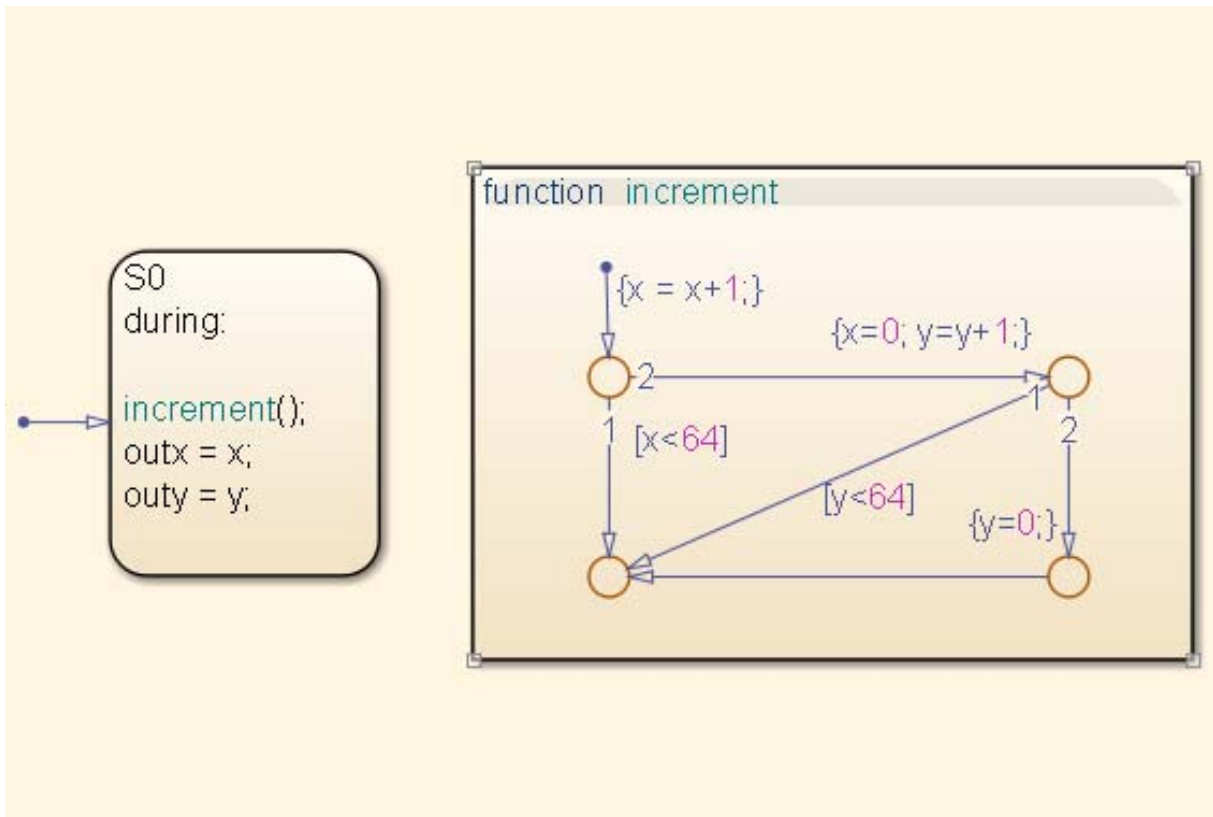
temporalCounter_i1_next <= temporalCounter_i1_temp;
END PROCESS Chart_1_output;

```

Graphical Function

A graphical function is a function defined graphically by a flow diagram. Graphical functions reside in a chart along with the diagrams that invoke them. Like MATLAB functions and C functions, graphical functions can accept arguments and return results. Graphical functions can be invoked in transition and state actions.

The following figure shows a graphical function that implements a 64-by-64 counter.



The following code excerpt shows VHDL code generated for this graphical function.

```
x64_counter_sf : PROCESS (x, y, outx_reg, outy_reg)
-- local variables
VARIABLE x_temp : unsigned(7 DOWNTO 0);
VARIABLE y_temp : unsigned(7 DOWNTO 0);
BEGIN
  outx_reg_next <= outx_reg;
  outy_reg_next <= outy_reg;
  x_temp := x;
  y_temp := y;
  x_temp := tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(x_temp, 9), 10)
```



```

+ tmw_to_unsigned(to_unsigned(1, 9), 10), 8);

    IF x_temp < to_unsigned(64, 8) THEN
        NULL;
    ELSE
        x_temp := to_unsigned(0, 8);
        y_temp := tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(y_temp, 9), 10)
+ tmw_to_unsigned(to_unsigned(1, 9), 10), 8);

        IF y_temp < to_unsigned(64, 8) THEN
            NULL;
        ELSE
            y_temp := to_unsigned(0, 8);
        END IF;

    END IF;

    outx_reg_next <= x_temp;
    outy_reg_next <= y_temp;
    x_next <= x_temp;
    y_next <= y_temp;
END PROCESS x64_counter_sf;

```

Hierarchy and Parallelism

Stateflow charts support both hierarchy (states containing other states) and parallelism (multiple states that can be active simultaneously).

In Stateflow semantics, parallelism is not synonymous with concurrency. Parallel states can be active simultaneously, but they are executed sequentially according to their execution order. (Execution order is displayed on the upper right corner of a parallel state).

For detailed information on hierarchy and parallelism, see “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.

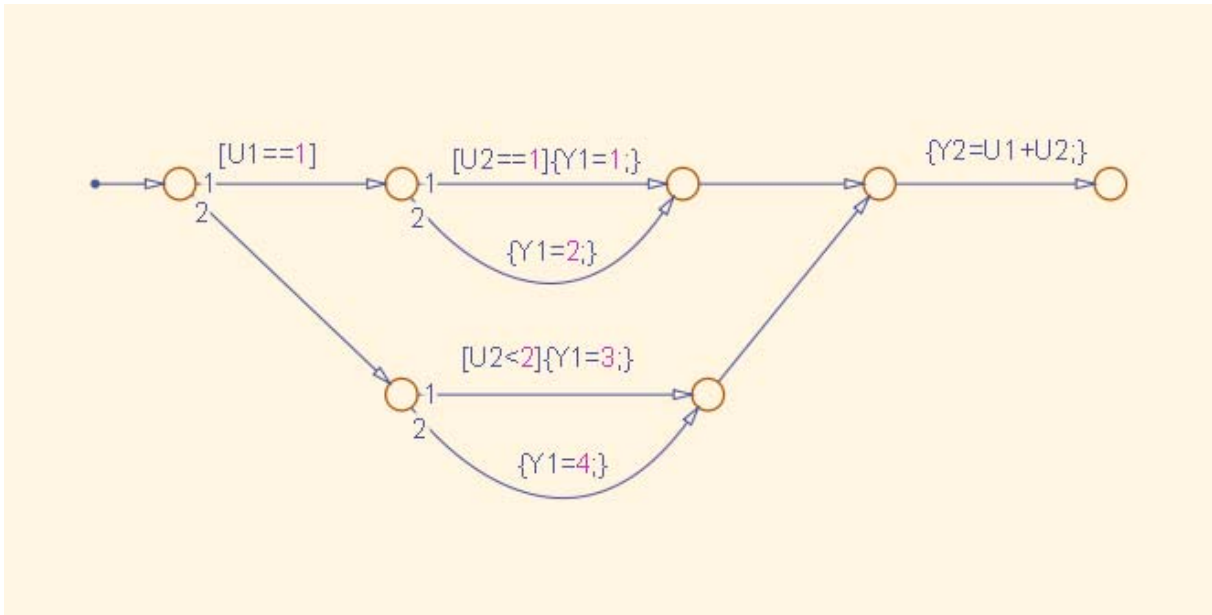
Stateless Charts

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;

  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

The coder supports HDL code generation for:

- Truth Table functions within a Stateflow chart
- Truth Table blocks in Simulink models

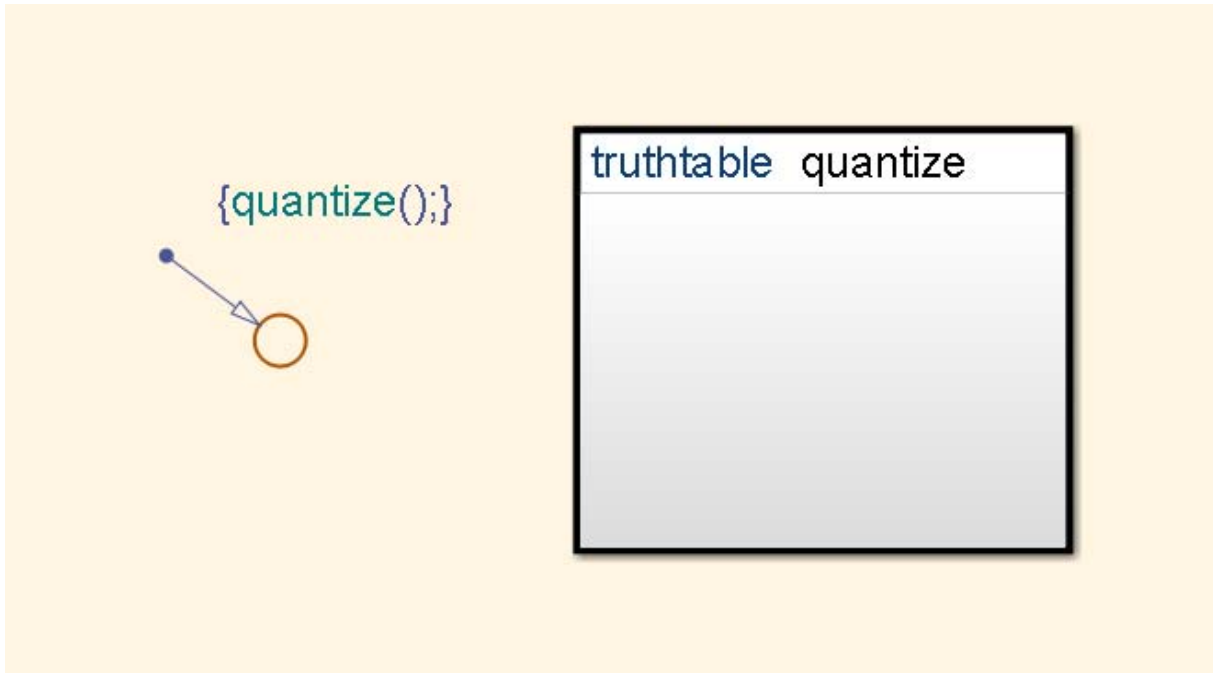
This section examines a Truth Table function in a chart, and the VHDL code generated for the chart.

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;
```

```
END IF;
```

```
END PROCESS quantizer;
```

Note When generating code for a Truth Table block in a Simulink model, the 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 20-2
- “Viterbi Decoder with the MATLAB Function Block” on page 20-3
- “Code Generation from a MATLAB Function Block” on page 20-4
- “MATLAB Function Block Design Patterns for HDL” on page 20-23
- “Design Guidelines for the MATLAB Function Block” on page 20-37
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 20-42
- “Limitations for MATLAB Function Block Code Generation” on page 20-49
- “MATLAB Language Support” on page 20-50

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 coder 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 20-4

“Setting Up” on page 20-7

“Creating the Model and Configuring General Model Settings” on page 20-8

“Adding a MATLAB Function Block to the Model” on page 20-8

“Set Fixed-Point Options for the MATLAB Function Block” on page 20-10

“Programming the MATLAB Function Block” on page 20-14

“Constructing and Connecting the DUT_eML_Block Subsystem” on page 20-15

“Compiling the Model and Displaying Port Data Types” on page 20-18

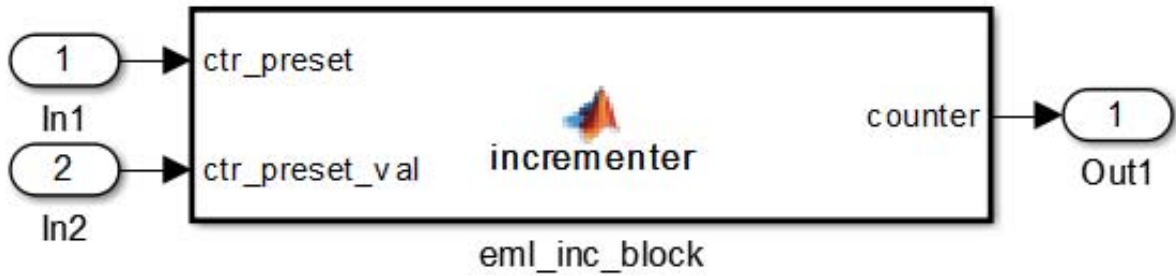
“Simulating the eml_hdl_incrementer_tut Model” on page 20-19

“Generating HDL Code” on page 20-20

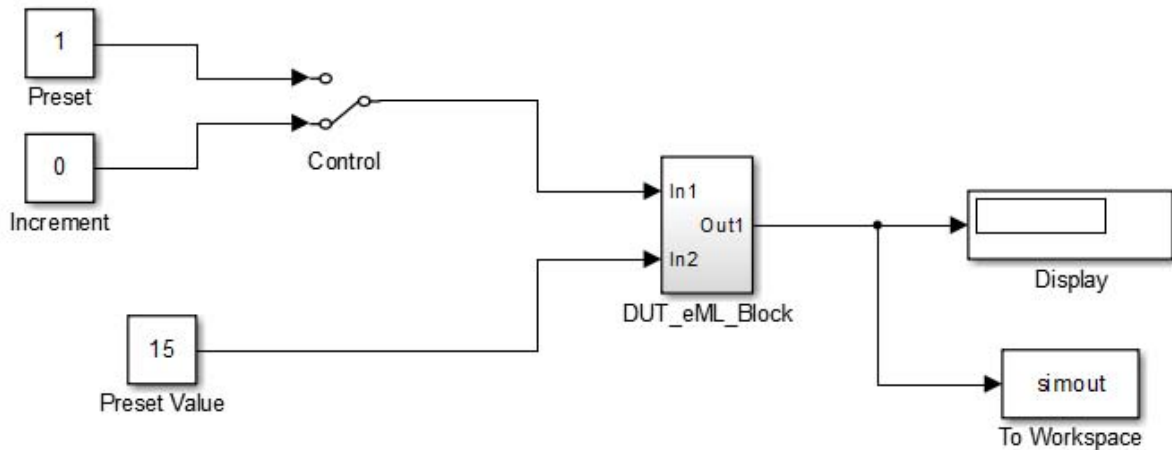
Counter Model Using the MATLAB Function block

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`.

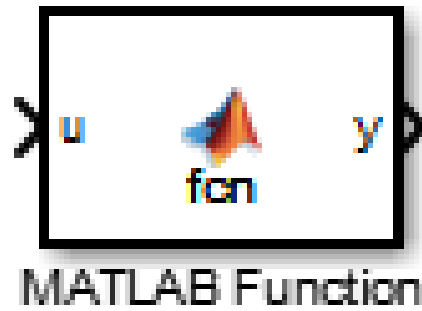
To set the model parameters:

- 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

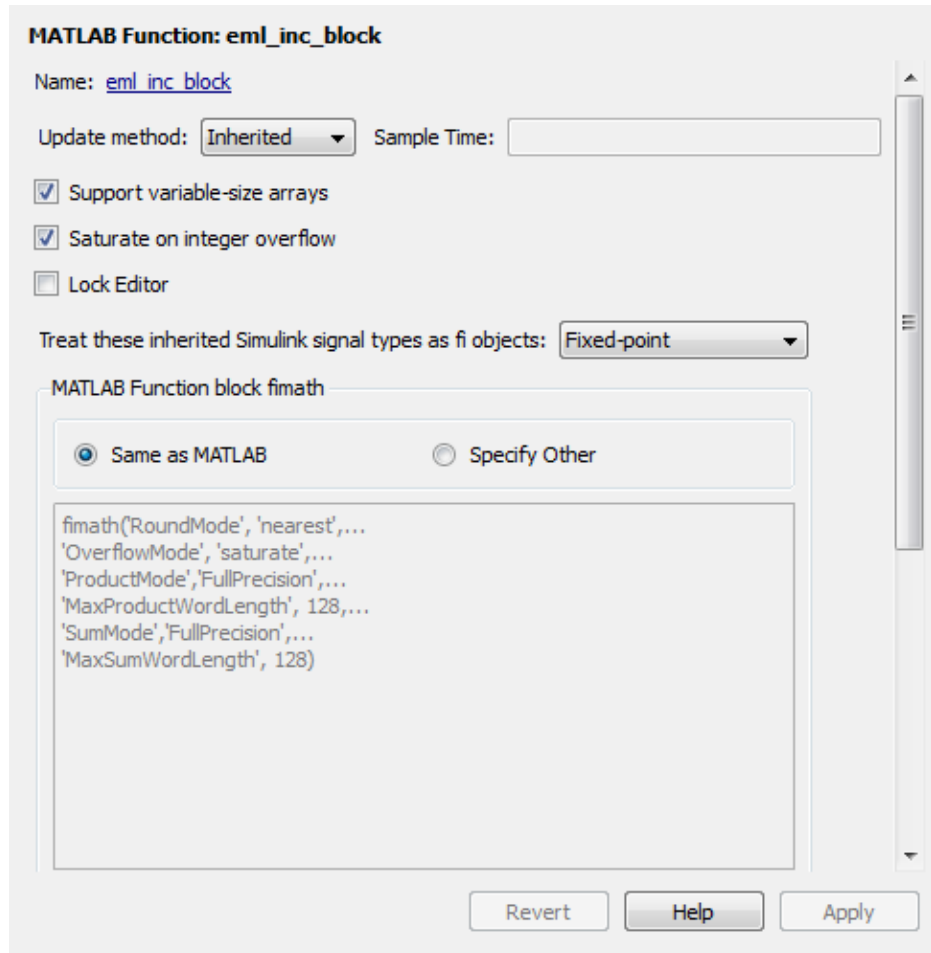
This section describes how to set up the FIMATH specification and other fixed-point options that are recommended for efficient HDL code generation from the MATLAB Function block. The recommended settings are:

- 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 20-8.
- 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.

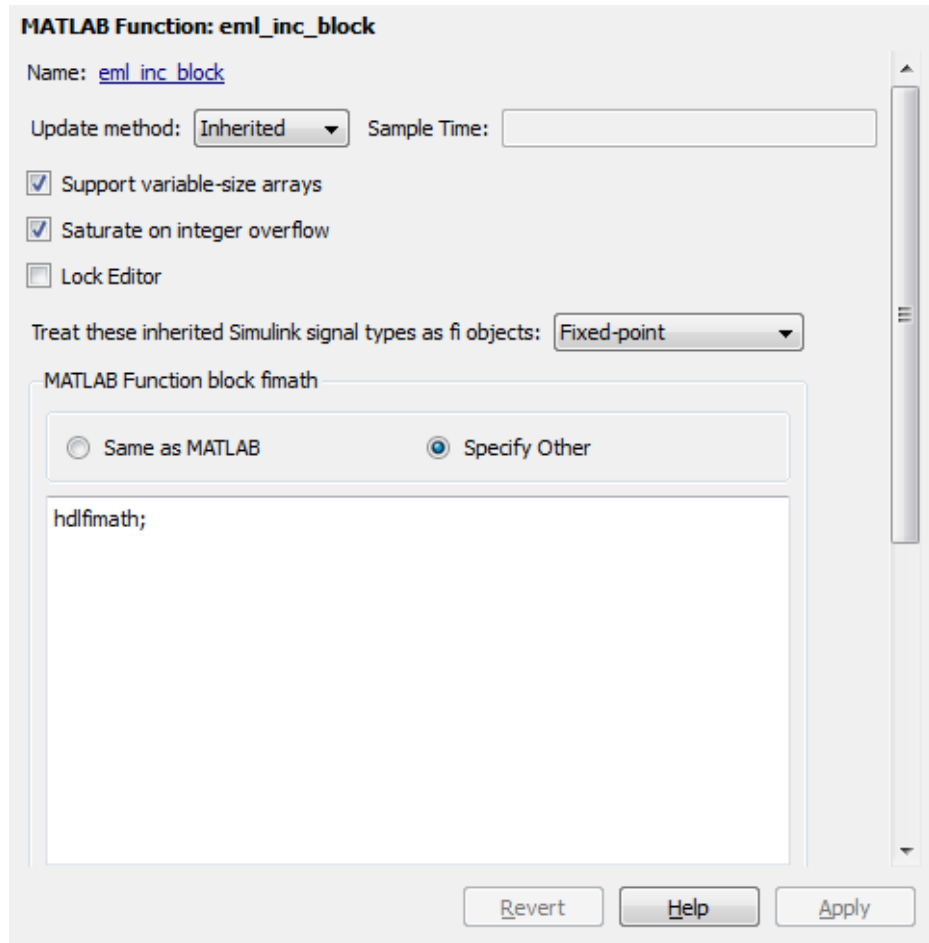


4 Select **Specify Other**. Selecting this option enables the **MATLAB Function block fimath** text entry field.

5 The `hdlfimath.m` 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.

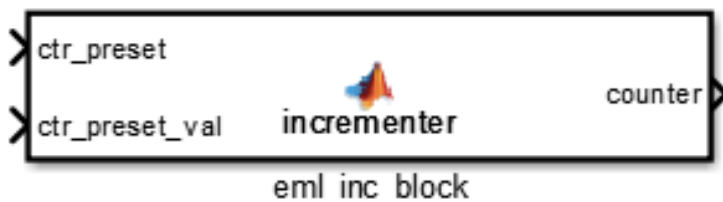
Programming the MATLAB Function Block

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 20-8.
- 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 20-6, 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 20-14 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:

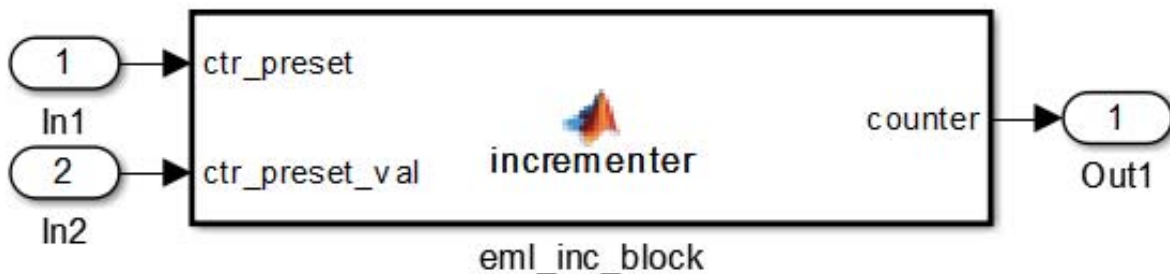
- 1 Click the `incrementer` function block.
- 2 Select **Diagram > Subsystem & Model Reference > Create Subsystem from Selection**.

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 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.

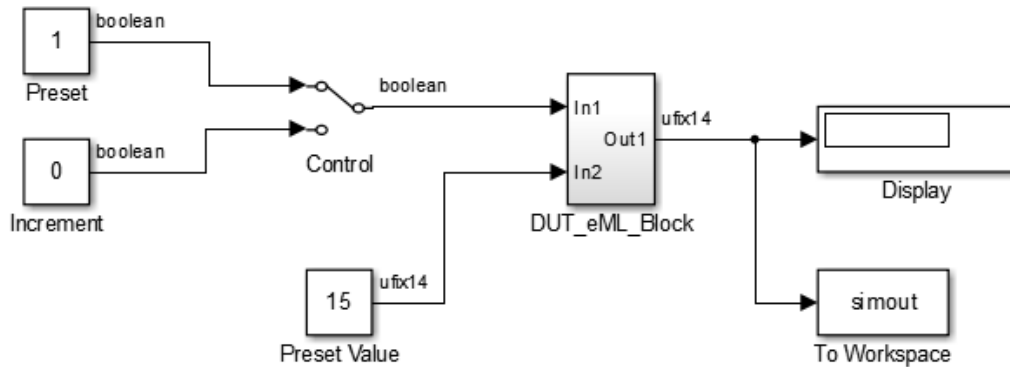
The build process builds an S-function for use in simulation. The build process includes generation of C code for the S-function. The code generation messages you see during the build process refer to generation of C code, not HDL code generation.

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.

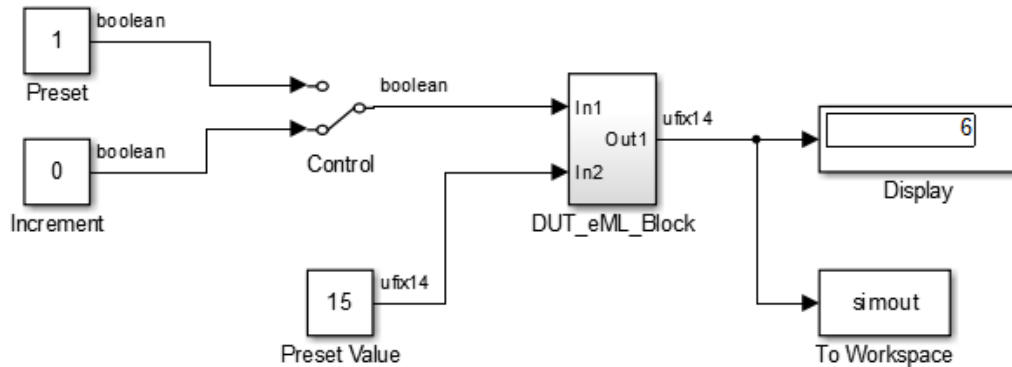


3 Save the model.

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.

The 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 16-43).
- 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.

MATLAB Function Block Design Patterns for HDL

In this section...

“The eml_hdl_design_patterns Library” on page 20-23

“Efficient Fixed-Point Algorithms” on page 20-25

“Model State Using Persistent Variables” on page 20-29

“Creating Intellectual Property with the MATLAB Function Block” on page 20-30

“Nontunable Parameter Arguments” on page 20-31

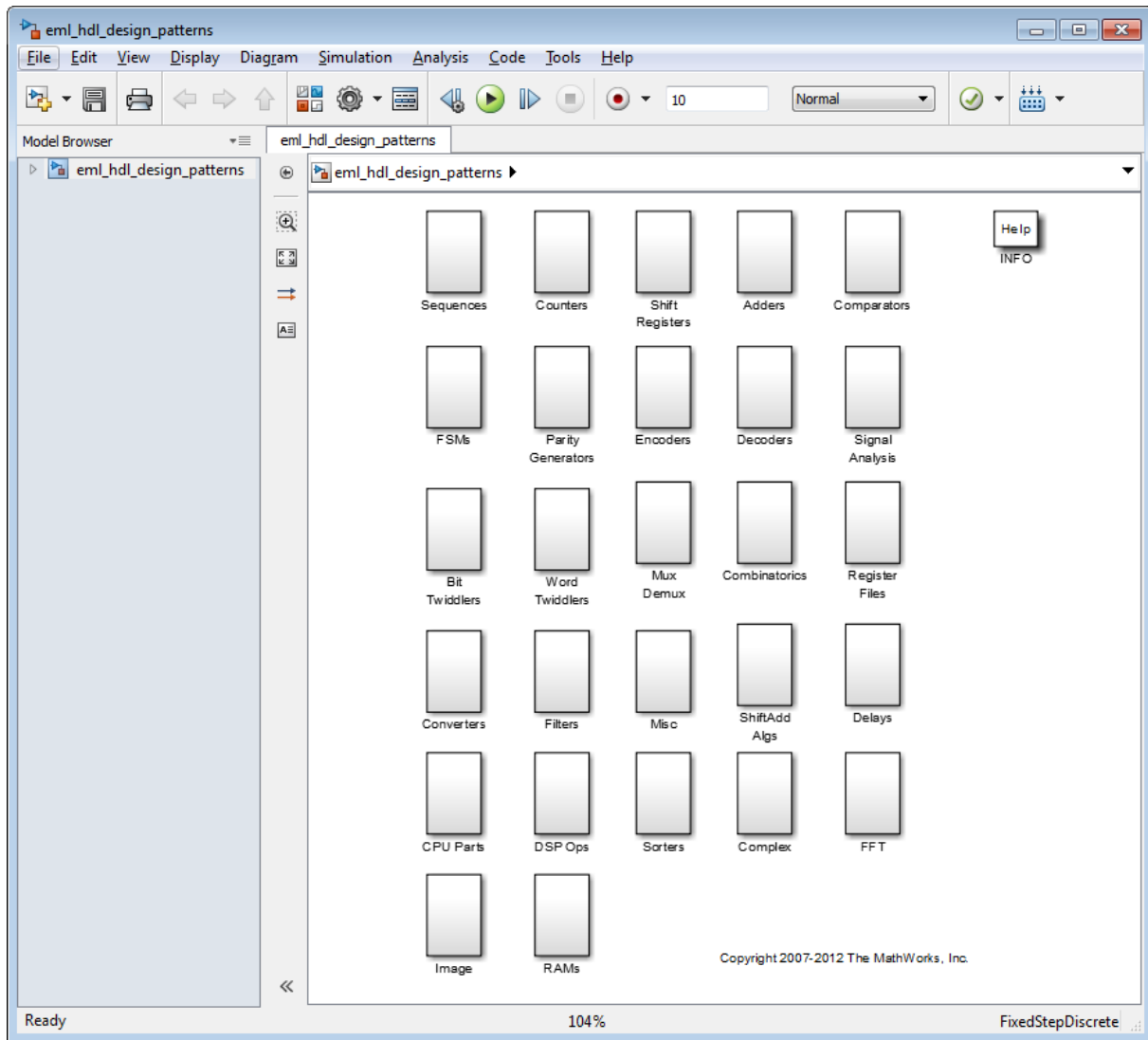
“Modeling Control Logic and Simple Finite State Machines” on page 20-31

“Modeling Counters” on page 20-33

“Modeling Hardware Elements” on page 20-35

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:

```
em1_hdl_design_patterns
```


You can use many blocks in the library as cookbook examples of various hardware elements, as follows:

- Copy a block from the library to your model and use it as a computational unit.
- Copy the code from the block and use it as a local function in an existing MATLAB Function block.

When you create custom blocks, you can control whether to inline or instantiate the HDL code generated from MATLAB Function blocks. Use the **Inline MATLAB Function block code** check box in the **HDL Code Generation > Global Settings > Coding style** section of the Configuration Parameters dialog box. For more information, see “Inline MATLAB Function block code” on page 9-74.

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. The coder supports all `fi` rounding and overflow modes.

HDL code generated from the MATLAB Function block is bit-true to MATLAB semantics. Generated code uses bit manipulation and bit access operators (for example, `Slice`, `Extend`, `Reduce`, `Concat`, etc.) that are native to VHDL and Verilog.

The following discussion shows how HDL code generated from the MATLAB Function block follows cast-before-sum semantics, in which addition and subtraction operands are cast to the result type before the addition or subtraction is performed.

Open the `eml_hdl_design_patterns` library and select the `Combinatorics/eml_expr` block. `eml_expr` implements a simple expression containing addition, subtraction, and multiplication operators with differing fixed point data types. The generated HDL code shows the conversion of this expression with fixed point operands. The MATLAB Function block uses the following code:

```
% fixpt arithmetic expression
expr = (a*b) - (a+b);

% cast the result to (sfix7_En4) output type
y = fi(expr, 1, 7, 4);
```

The default `fimath` specification for the block determines the behavior of arithmetic expressions using fixed point operands inside the MATLAB Function block:

```
fimath(...
    'RoundMode', 'ceil',...
    'OverflowMode', 'saturate',...
    'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
    'SumMode', 'FullPrecision', 'SumWordLength', 32,...
    'CastBeforeSum', true)
```

The data types of operands and output are as follows:

- a: (sfix5_En2)
- b: (sfix5_En3)
- y: (sfix7_En4).

Before HDL code generation, 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 20-37) 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 DOWNTO 7) /= "000"))
    OR ((expr(10) = '0') AND (expr(7 DOWNTO 1) = "0111111"))
    ELSE
    "1000000" WHEN (expr(10) = '1') AND (expr(9 DOWNTO 7) /= "111")
    ELSE
    std_logic_vector(expr(7 DOWNTO 1) + ("0" & expr(0)));

END fsm_SFHD;

```

Verilog code excerpt:

```

//MATLAB Function 'Subsystem/em1_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 the 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`.

The 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_SFHDH 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
end
```



```
        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` function. 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 DOWNT0 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 20-37

“Use Compiled External Functions With MATLAB Function Blocks” on page 20-37

“Build the MATLAB Function Block Code First” on page 20-38

“Use the hdlfimath Utility for Optimized FIMATH Settings” on page 20-38

“Use Optimal Fixed-Point Option Settings” on page 20-40

“Set the Output Data Type of MATLAB Function Blocks Explicitly” on page 20-41

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 20-10 for an example.

Use Compiled External Functions With MATLAB Function Blocks

The coder 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.

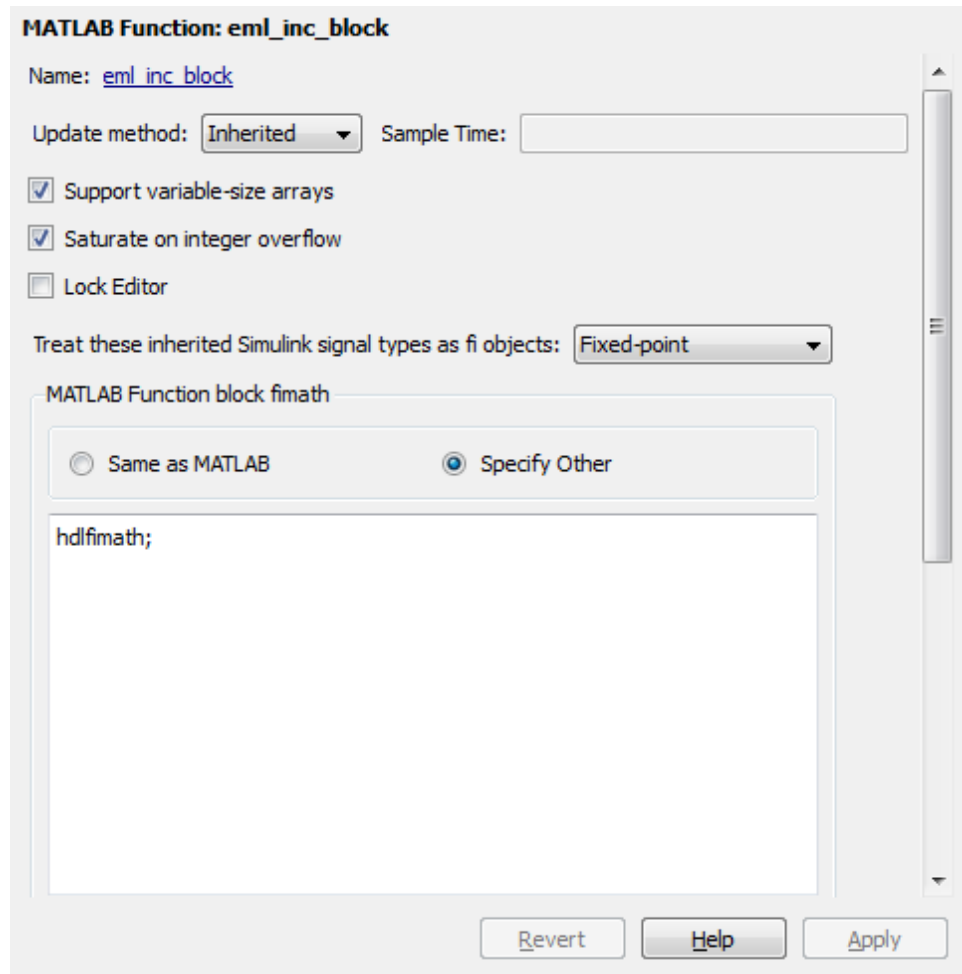
Such functions must be defined in files that are on the MATLAB Function block path. Use the `%#codegen` compilation directive to indicate that the MATLAB code is suitable for code generation. See “Function Definition” for information on how to create, compile, and invoke external functions.

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.m` 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);
```

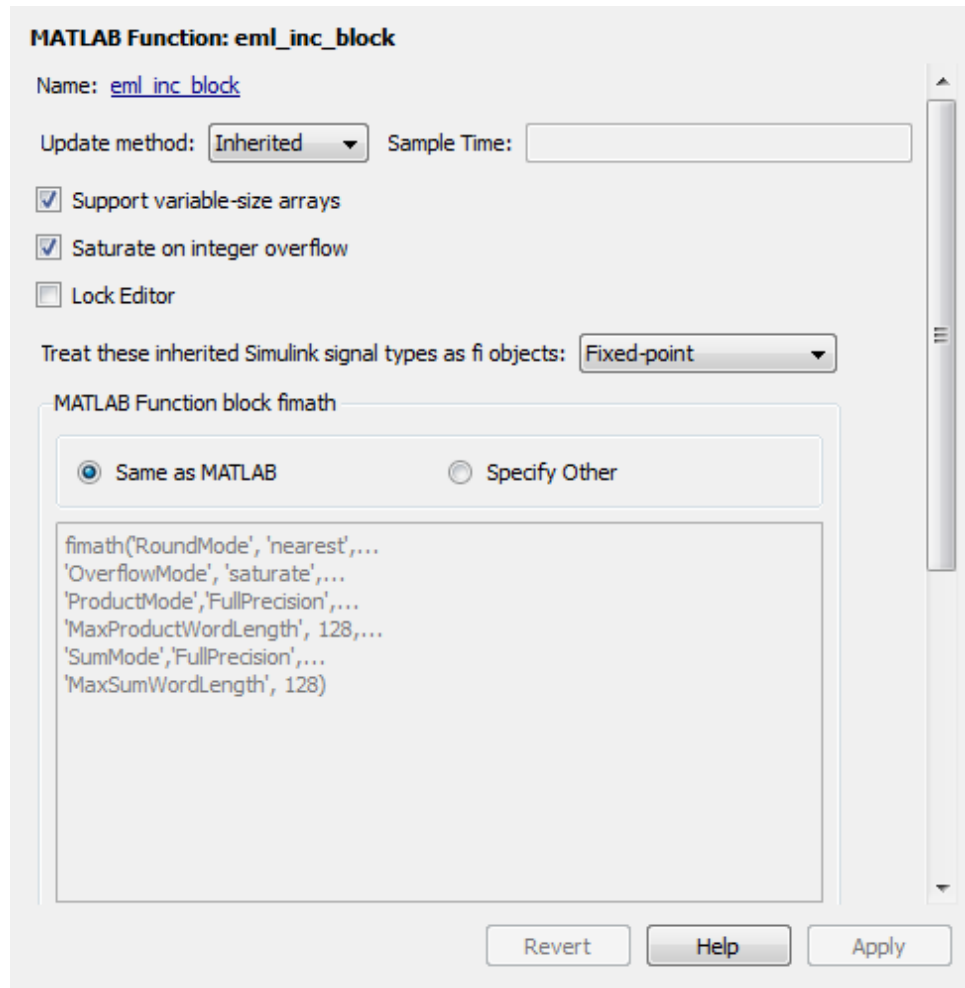
Note Use of 'floor' rounding mode for signed integer division will cause an error at code generation time. The HDL division operator does not support 'floor' rounding mode. Use 'round' mode, or else change the signed integer division operations to unsigned integer division.

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.



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 20-42

“Distributed Pipelining in a Multiplier Chain” on page 20-42

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” on page 11-70.

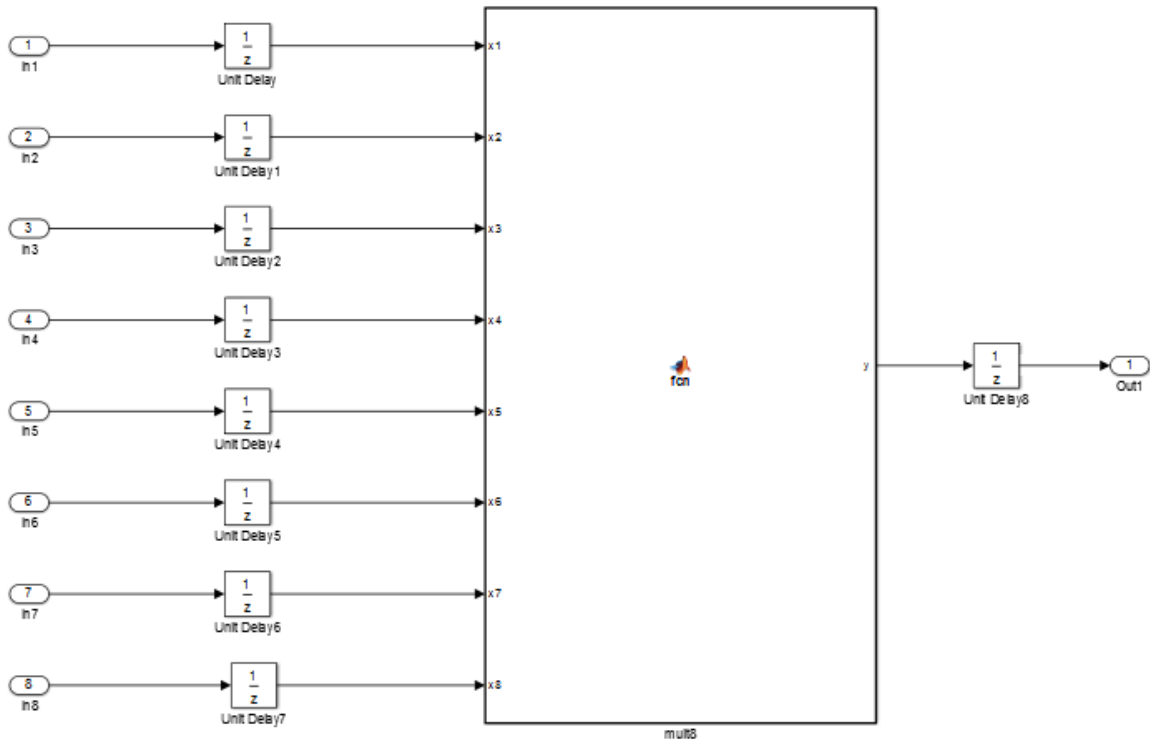
Distributed Pipelining in a Multiplier Chain

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:

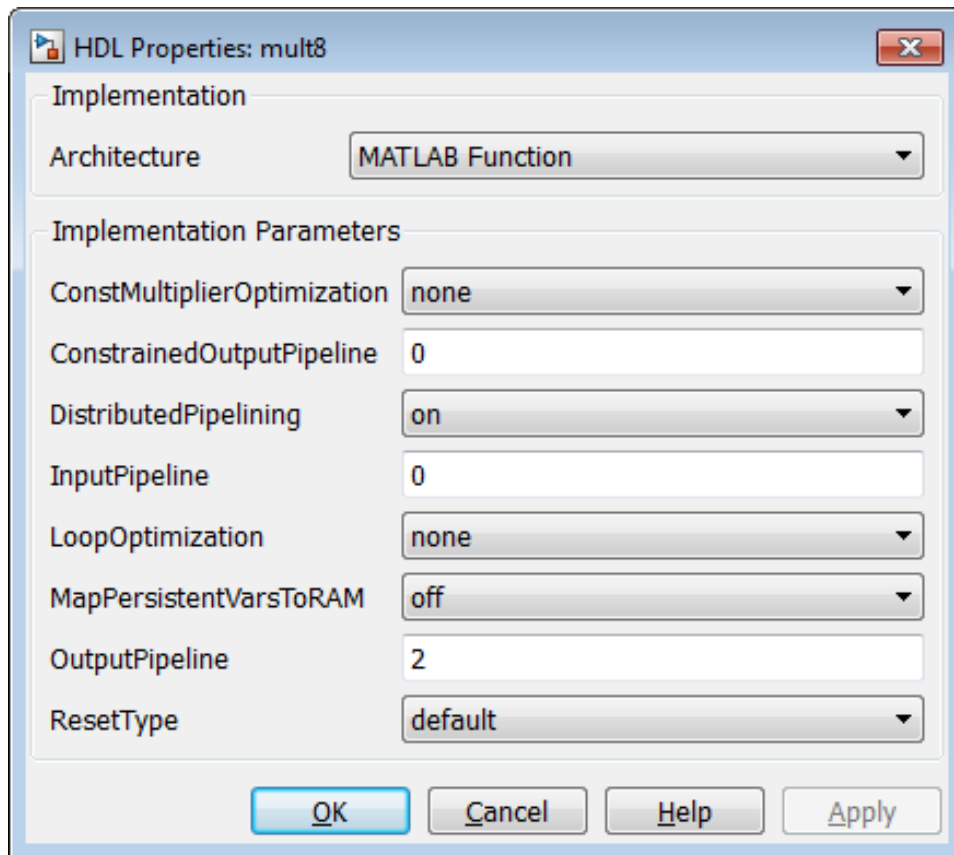
```
function y = fcn(x1,x2,x3,x4,x5,x6,x7,x8)
% A chained multiplication:
% y = (x1*x2)*(x3*x4)*(x5*x6)*(x7*x8)

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:

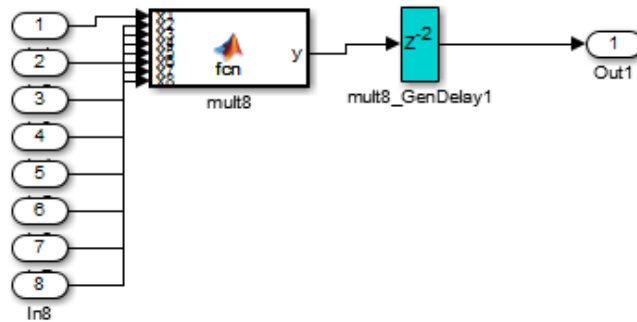


In the Configuration Parameters dialog box, the top-level **HDL Code Generation** options specify that:

- VHDL code is generated from the subsystem `mpipe_multchain/mult_chain`.
- The 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 the 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 DOWNT0 7) /= "00000000")
      ELSE "10000000" WHEN (mul_temp(15) = '1') AND (mul_temp(14 DOWNT0 7) /= "11111111")
      ELSE mul_temp(7 DOWNT0 0);

mul_temp_0 <= signed(x3) * signed(x4);

y2 <= "01111111" WHEN (mul_temp_0(15) = '0') AND (mul_temp_0(14 DOWNT0 7) /= "00000000")
      ELSE "10000000" WHEN (mul_temp_0(15) = '1') AND (mul_temp_0(14 DOWNT0 7) /= "11111111")
      ELSE mul_temp_0(7 DOWNT0 0);

mul_temp_1 <= signed(x5) * signed(x6);

y3 <= "01111111" WHEN (mul_temp_1(15) = '0') AND (mul_temp_1(14 DOWNT0 7) /= "00000000")
      ELSE "10000000" WHEN (mul_temp_1(15) = '1') AND (mul_temp_1(14 DOWNT0 7) /= "11111111")
      ELSE mul_temp_1(7 DOWNT0 0);

mul_temp_2 <= signed(x7) * signed(x8);

y4 <= "01111111" WHEN (mul_temp_2(15) = '0') AND (mul_temp_2(14 DOWNT0 7) /= "00000000")
      ELSE "10000000" WHEN (mul_temp_2(15) = '1') AND (mul_temp_2(14 DOWNT0 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;
```


Limitations for MATLAB Function Block Code Generation

The HDL compatibility checker (`checkhdl`) performs a basic compatibility check on the MATLAB Function block. HDL related warnings or errors may arise during code generation from a MATLAB Function block that is otherwise valid for simulation. Such errors are reported in a separate message window.

For more information about MATLAB language support and limitations, see “MATLAB Language Support” on page 20-50.

MATLAB Language Support

For the MATLAB language subset supported for HDL code generation from a MATLAB Function block, see:

- “Data Types and Scope” on page 1-2
- “Operators” on page 1-4
- “Control Flow Statements” on page 1-7
- “Persistent Variables” on page 1-9
- “Persistent Array Variables” on page 1-11
- “System Objects” on page 1-21
- “Complex Data Type Support” on page 1-12
- “Fixed-Point Bitwise Functions” on page 1-28
- “Fixed-Point Run-Time Library Functions” on page 1-35

Generating Scripts for HDL Simulators and Synthesis Tools

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 21-2
- “Structure of Generated Script Files” on page 21-3
- “Properties for Controlling Script Generation” on page 21-4
- “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9

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 21-4.
- Set script generation options in the **HDL Code Generation > EDA Tool Scripts** pane of the Configuration Parameters dialog box, as described in “Control Script Generation with the EDA Tool Scripts Pane” on page 21-9.

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 coder 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	'HDLCompileFilePostfix'	'_compile.do'
Simulation	'HDLSimFilePostfix'	'_sim.do'
Synthesis	'HDLSynthFilePostfix'	Depends on the selected synthesis tool. See HDLSynthTool.

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 <code>Init</code> section of the compilation script. The argument is the contents of the 'VHDLLibraryName' property, which defaults to 'work'. You can override the default <code>Init</code> string ('vlib work\n' by changing the value of 'VHDLLibraryName'.
Name: 'HDLCompileVHDLCmd' 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 'SimulatorFlags' property and the file name of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).
Name: 'HDLCompileVerilogCmd' Default: 'vlog %s %s\n'	Format 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 'SimulatorFlags' property and the file name of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).
Name: 'HDLCompileTerm' Default: ''	Format string passed to <code>fprintf</code> to write the termination portion of the compilation script.

Property Name and Default	Description
Name: 'HDLsimInit' Default: ['onbreak resume\n',... 'onerror resume\n']	Format string passed to fprintf to write the initialization section of the simulation script.
Name: 'HDLsimCmd' Default: 'vsim -novopt work.%s\n'	Format string passed to fprintf to write the simulation command. The implicit argument is the top-level module or entity name.
Name: 'HDLsimViewWaveCmd' Default: 'add wave sim:%s\n'	Format string passed to fprintf to write the simulation script waveform viewing command. The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.
Name: 'HDLsimTerm' Default: 'run -all\n'	Format string passed to fprintf to write the Term portion of the simulation script. The string is a synthesis project creation command. The implicit argument is the top-level module or entity name.
Name: 'HDLsynthInit'	Format string passed to fprintf to write the Init section of the synthesis script. The content of the string is specific to the selected synthesis tool. See HDLSynthTool.
Name: 'HDLsynthCmd'	Format string passed to fprintf to write the Cmd section of the synthesis script. The argument is the file name of the entity or module. The content of the string is specific to the selected synthesis tool.
Name: 'HDLsynthTerm'	Format string passed to fprintf to write the Term section of the synthesis script. The content of the

Property Name and Default	Description
	string is specific to the selected synthesis tool. See HDLSynthTool.

Examples

The following example specifies a Mentor Graphics ModelSim command for the Init phase of a compilation script for VHDL code generated from the subsystem system.

```
makehdl(system, 'HDLCompileInit', 'vlib mydesignlib\n')
```

The following example lists the resultant script, system_compile.do.

```
vlib mydesignlib
vcom system.vhd
```

The following example specifies that the 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
```

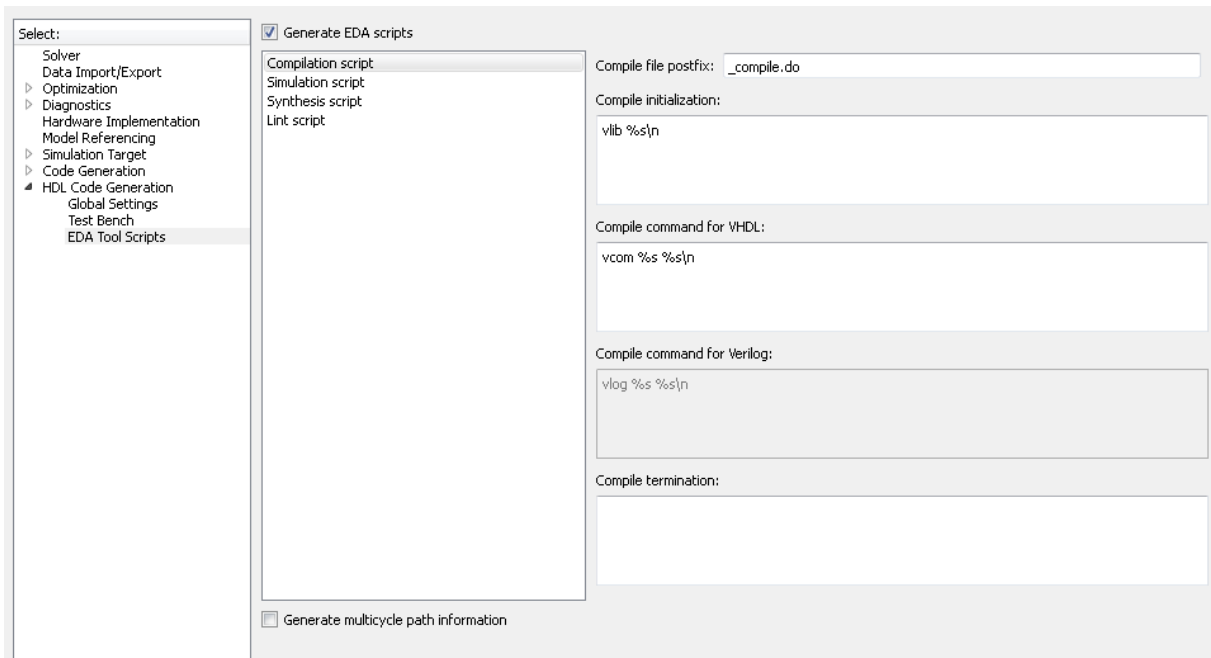
```
project set device xc4vsx35
project set package ff668
project set speed -10
process run "Synthesize - XST"
```

Control Script Generation with the EDA Tool Scripts Pane

You set options that control generation of script files on the **EDA Tool Scripts** pane. These options correspond to the properties described in “Properties for Controlling Script Generation” on page 21-4.

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.



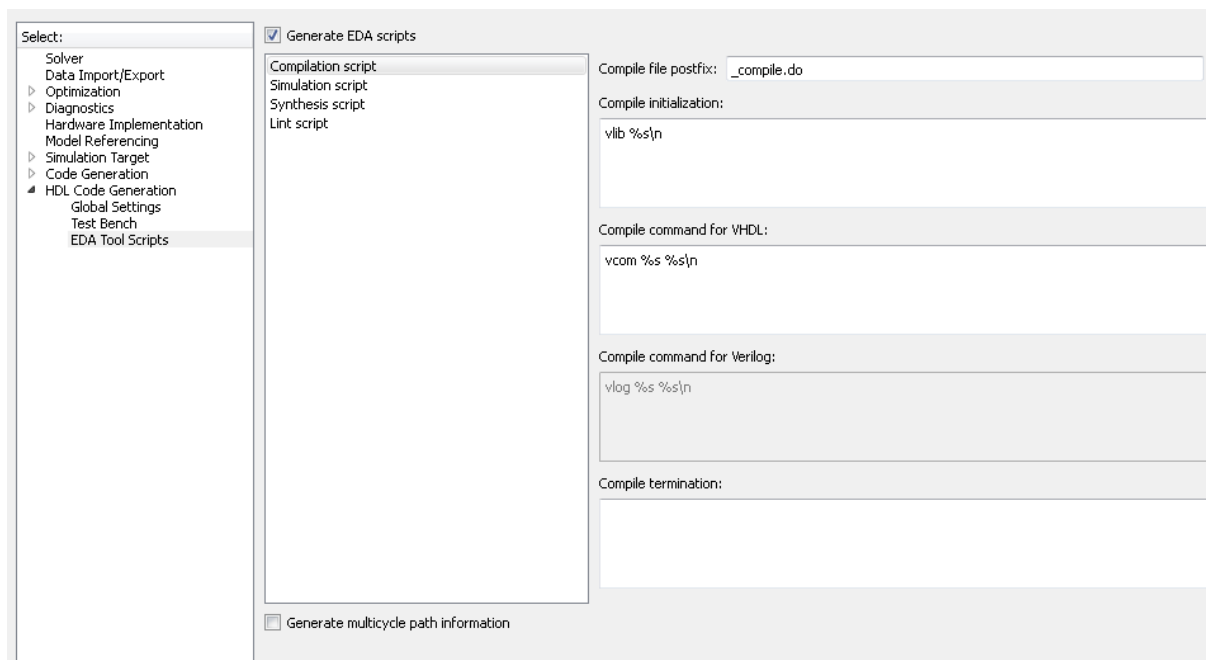
- 3 The **Generate EDA scripts** option controls the generation of script files. By default, this option is selected.

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 21-10 for further information.
 - **Simulation script:** Options related to customizing scripts for HDL simulators. See “Simulation Script Options” on page 21-11 for further information.
 - **Synthesis script:** Options related to customizing scripts for synthesis tools. See “Synthesis Script Options” on page 21-13 for further information.

Compilation Script Options

The following figure shows the **Compilation script** pane, with options set to their default values.

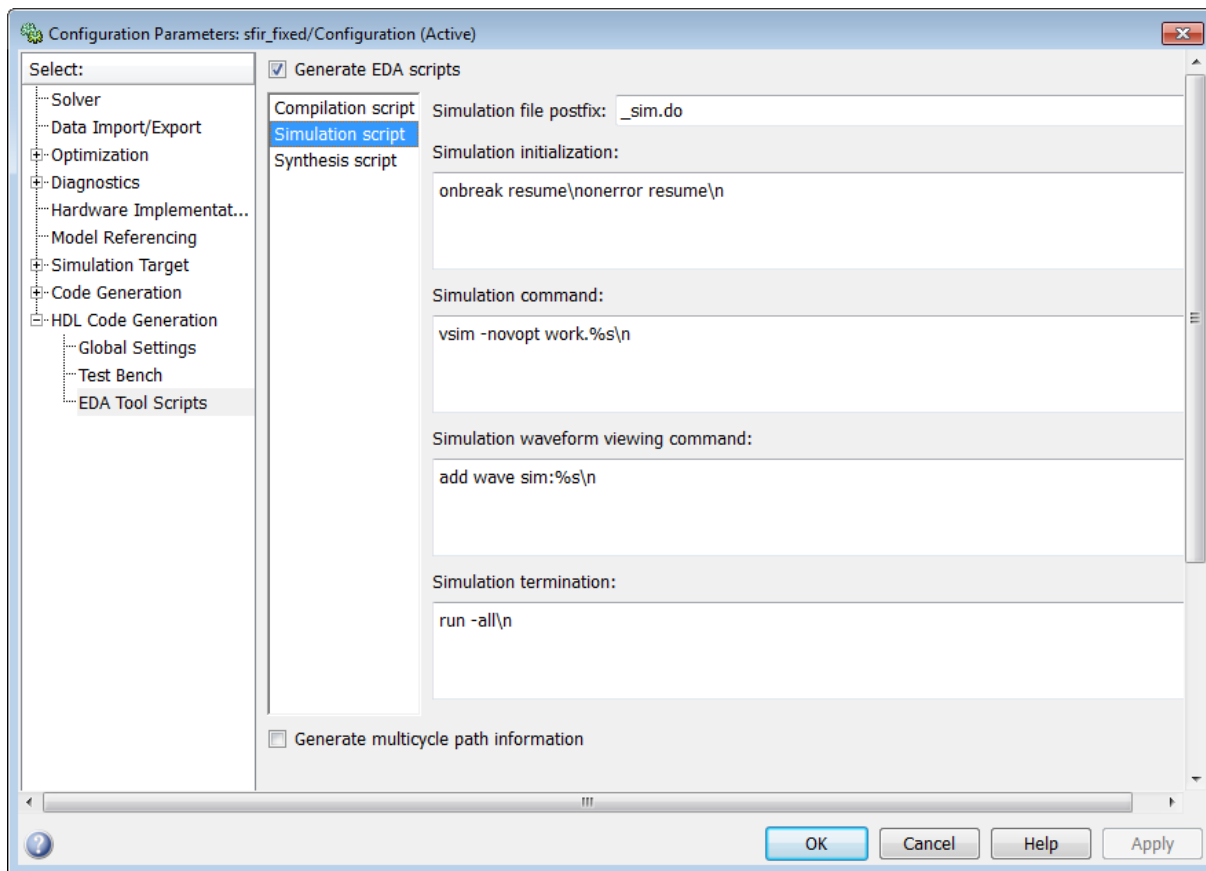


The following table summarizes the **Compilation script** options.

Option and Default	Description
Compile file postfix' '_compile.do'	Postfix string appended to the DUT name or test bench name to form the script file name.
Name: Compile initialization Default: 'vlib %s\n'	Format string passed to fprintf to write the Init section of the compilation script. The argument is the contents of the 'VHDLLibraryName' property, which defaults to 'work'. You can override the default <i>Init string ('vlib work\n'</i> by changing the value of
Name: Compile command for VHDL Default: 'vcom %s %s\n'	Format string passed to fprintf to write the Cmd section of the compilation script for VHDL files. The two arguments are the contents of the 'SimulatorFlags' property option and the filename of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).
Name: Compile command for Verilog Default: 'vlog %s %s\n'	Format string passed to fprintf to write the Cmd section of the compilation script for Verilog files. The two arguments are the contents of the 'SimulatorFlags' property and the filename of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).
Name: Compile termination Default: ' '	Format string passed to fprintf to write the termination portion of the compilation script.

Simulation Script Options

The following figure shows the **Simulation script** pane, with options set to their default values.



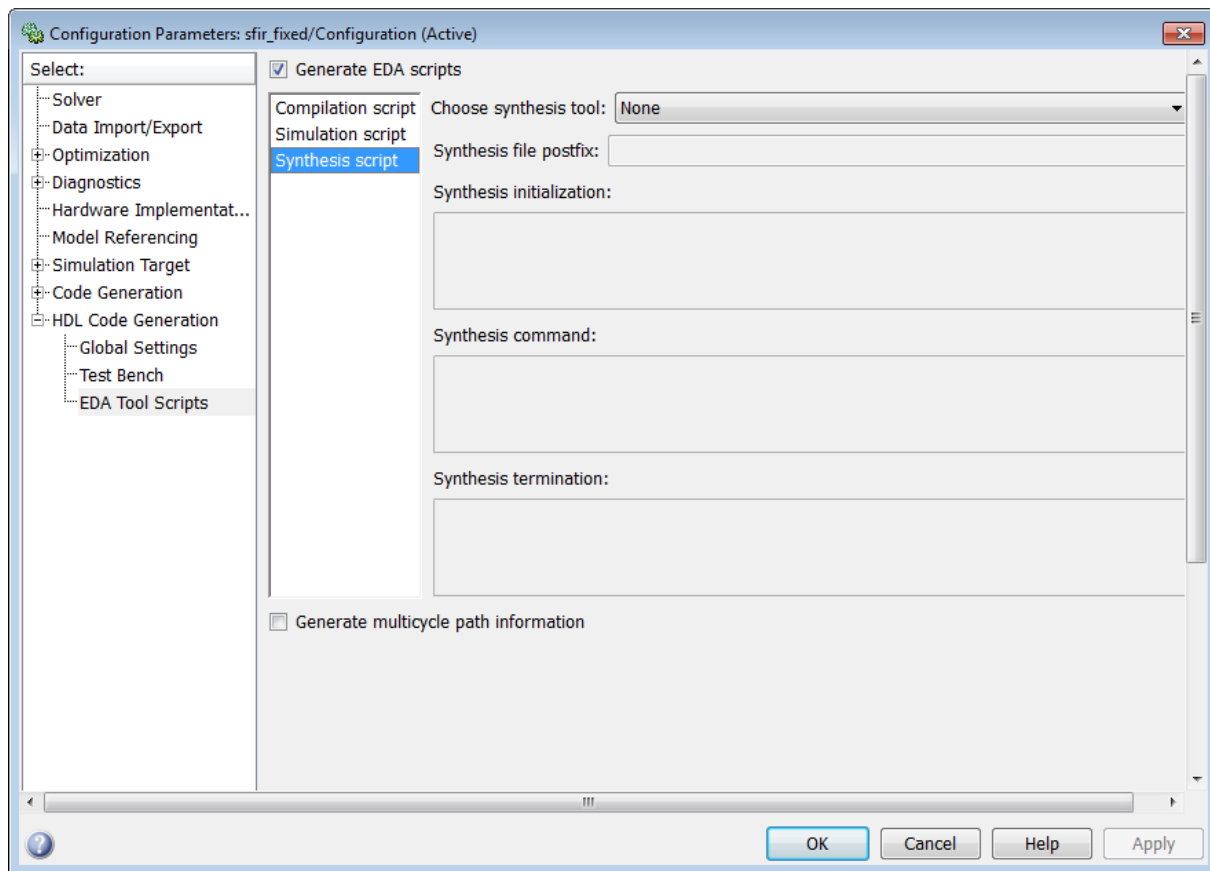
The following table summarizes the **Simulation script** options.

Option and Default	Description
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\n\nonerror resume\n\n']	Format string passed to fprintf to write the initialization section of the simulation script.

Option and Default	Description
Simulation command Default: 'vsim -novopt work.%s\n'	Format string passed to fprintf to write the simulation command. The implicit argument is the top-level module or entity name.
Simulation waveform viewing command Default: 'add wave sim:%s\n'	Format string passed to fprintf to write the simulation script waveform viewing command. The top-level module or entity signal names are implicit arguments.
Simulation termination Default: 'run -all\n'	Format string passed to fprintf 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.

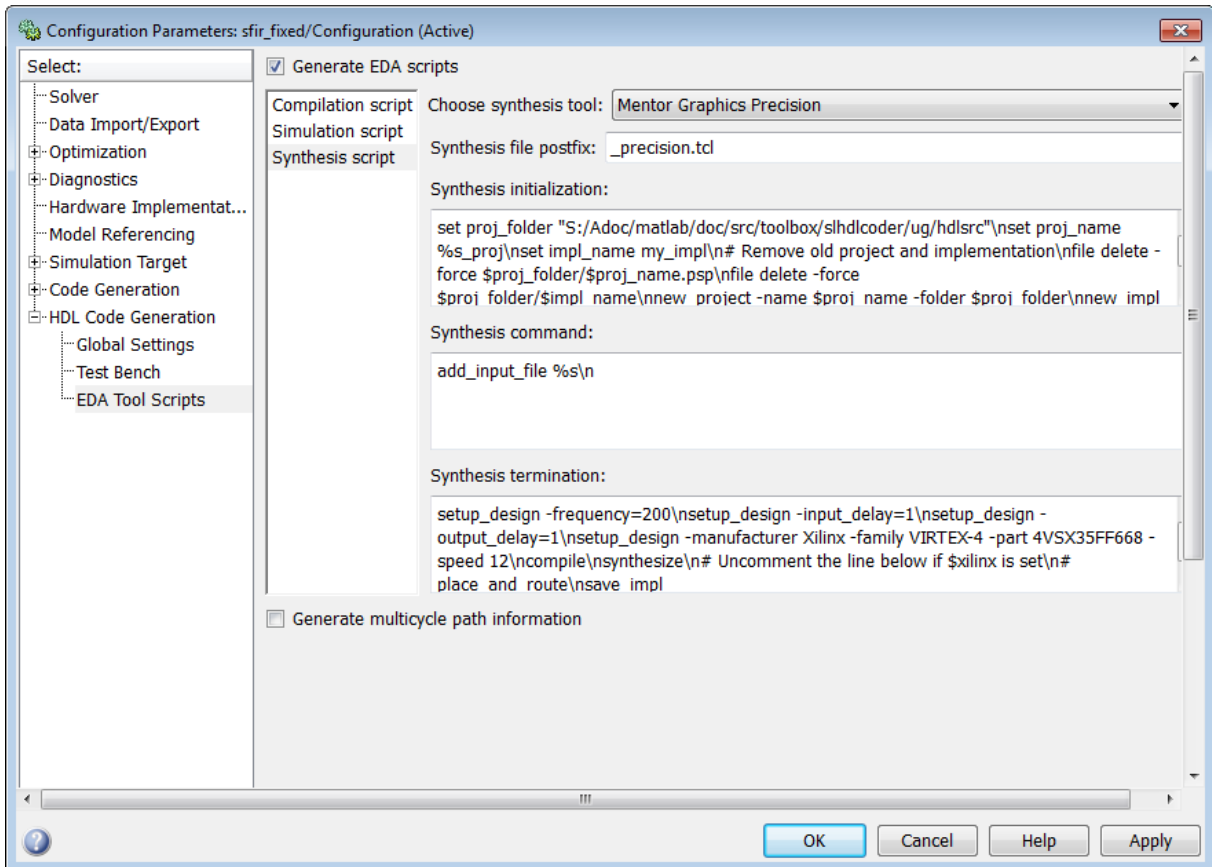


To enable synthesis script generation, select a synthesis tool from the **Choose synthesis tool** menu.

When you select a synthesis tool, the coder:

- Enables synthesis script generation.
- Enters a file name postfix (specific to the chosen synthesis tool) into the **Synthesis file postfix** field.
- Enters strings (specific to the chosen synthesis tool) into the initialization, command, and termination fields.

The following figure shows the default option values entered for the Mentor Graphics Precision tool.



The following table summarizes the **Synthesis script** options.

Option Name	Description
Choose synthesis tool	<p>None (default): do not generate a synthesis script</p> <p>ISE: generate a synthesis script for Xilinx ISE</p> <p>Libero: generate a synthesis script for Microsemi Libero</p> <p>Precision: generate a synthesis script for Mentor Graphics Precision</p> <p>Quartus: generate a synthesis script for Altera Quartus II</p> <p>Synplify: generate a synthesis script for Synopsys Synplify Pro</p> <p>Custom: generate a custom synthesis script</p>
Synthesis file postfix	<p>Your choice of synthesis tool sets the postfix for generated synthesis file names to one of the following:</p> <pre>_ise.tcl _libero.tcl _precision.tcl _quartus.tcl _synplify.tcl _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. 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 argument is the filename of the entity or module. 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. The content of the string is specific to the selected synthesis tool.</p>

Using the HDL Workflow Advisor

- “What Is the HDL Workflow Advisor?” on page 22-3
- “Open the HDL Workflow Advisor” on page 22-4
- “Using the HDL Workflow Advisor Window” on page 22-7
- “Save and Restore HDL Workflow Advisor State” on page 22-10
- “Fix a Workflow Advisor Warning or Failure” on page 22-14
- “View and Save HDL Workflow Advisor Reports” on page 22-17
- “Map to an FPGA Floating-Point Library” on page 22-22
- “FPGA Synthesis and Analysis” on page 22-28
- “Automated Workflows for Specific Targets and Tools” on page 22-40
- “Generate xPC Target Interface for Speedgoat Boards” on page 22-42
- “Target Altera FPGA Development Boards” on page 22-51
- “Target Xilinx FPGA Development Boards” on page 22-58
- “Custom IP Core Generation” on page 22-65
- “Custom IP Core Report” on page 22-68
- “Hardware and Software Codesign for Xilinx Zynq-7000 Platform” on page 22-75
- “Generate a Custom IP Core” on page 22-76
- “Processor and FPGA Synchronization” on page 22-80
- “Hardware and Software Codesign Workflow” on page 22-83

- “Install Support for Altera FPGA Boards” on page 22-92
- “Install Support for Xilinx FPGA Boards” on page 22-93
- “Install Support for Xilinx Zynq-7000 Platform” on page 22-94

What Is the HDL Workflow Advisor?

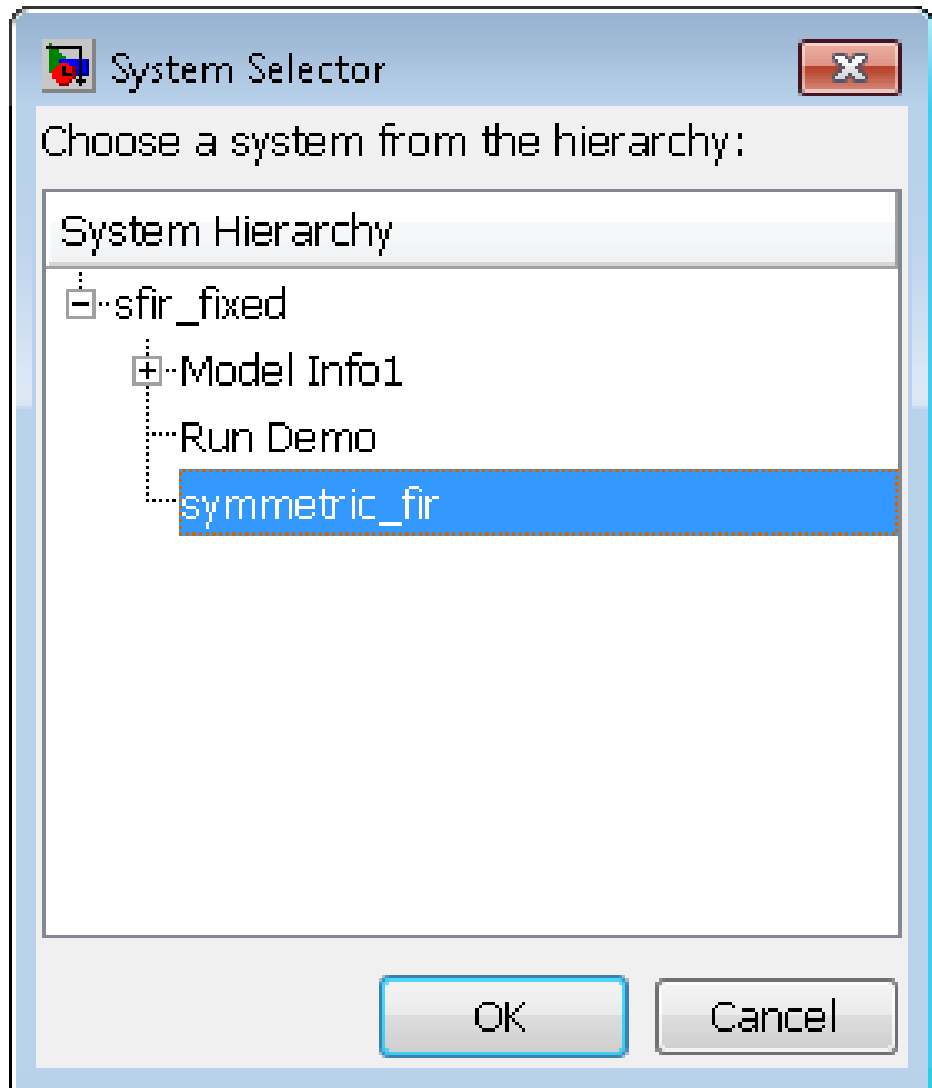
The HDL Workflow Advisor is a tool that supports and integrates the stages of the FPGA design process, such as:

- 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 xPC Target™, including FPGA-in-the-Loop simulation

Open the HDL Workflow Advisor

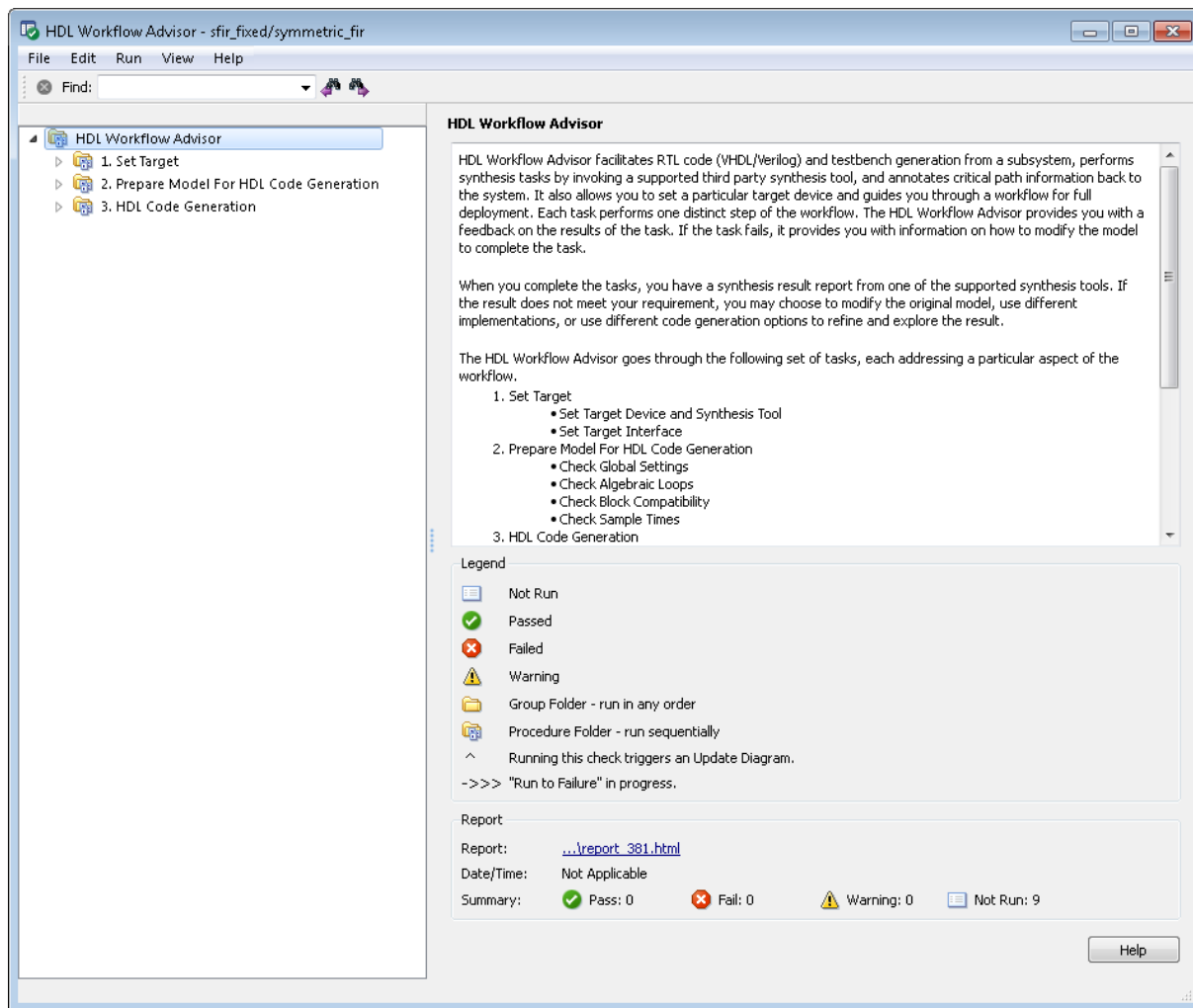
To start the HDL Workflow Advisor from a model:

- 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**.

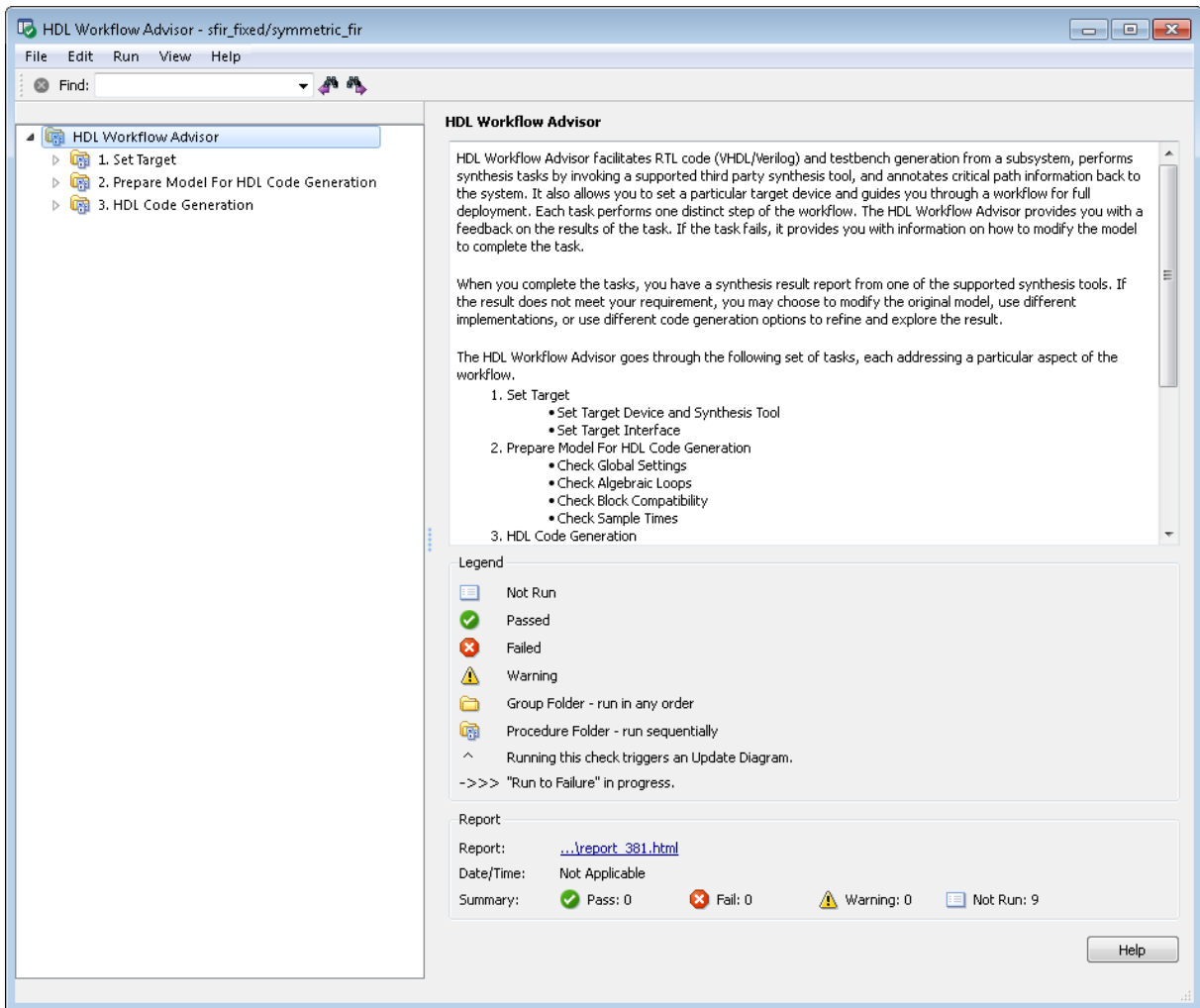
The HDL Workflow Advisor initializes and appears.



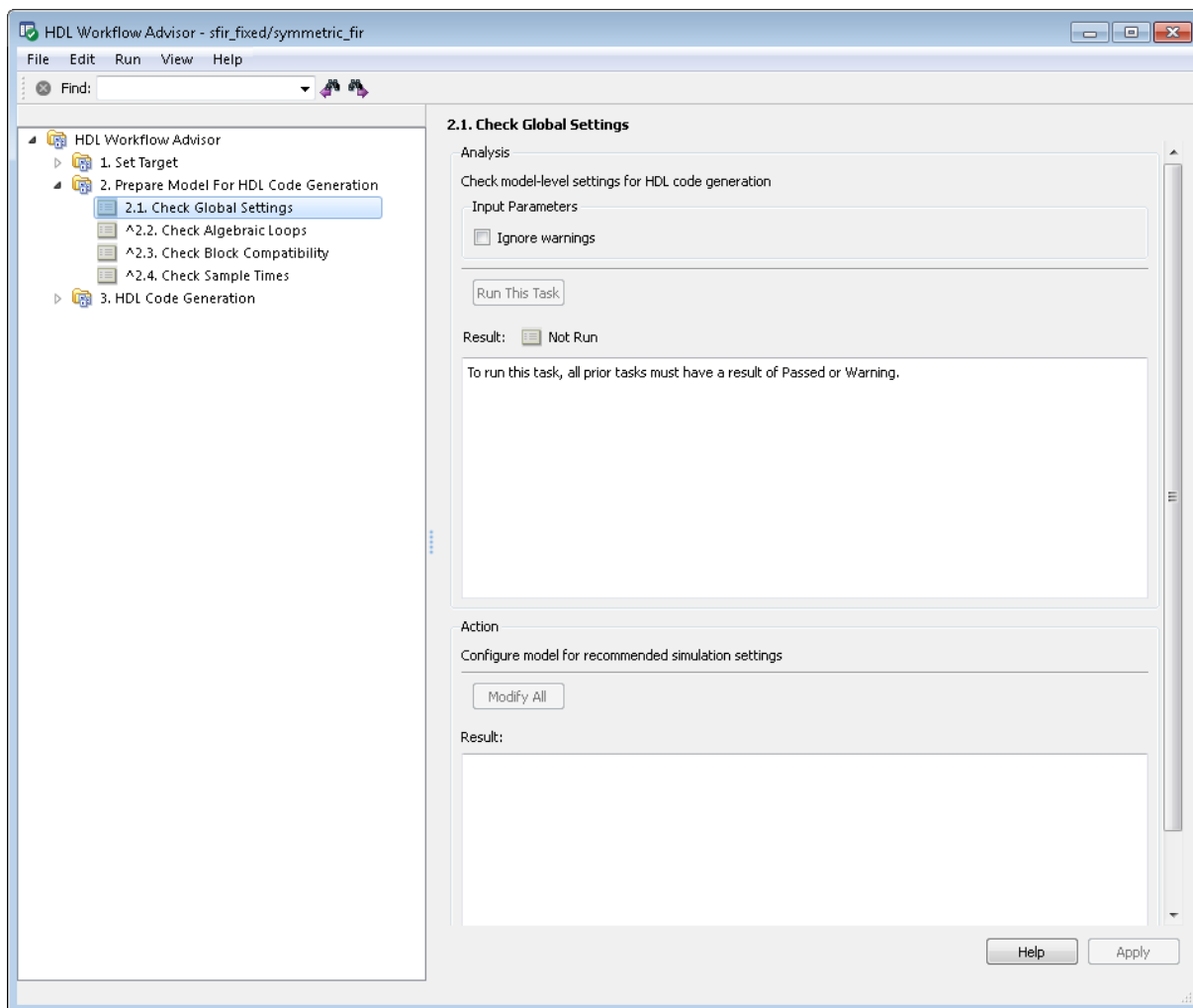
To start the HDL Workflow Advisor from the command line, enter `hdladvisor(system)`, where `system` is a handle or name of the model or subsystem that you want to check. For more information, see the `hdladvisor` function reference page.

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 22-10

“Limitations of the Save and Restore Process” on page 22-10

“Save the HDL Workflow Advisor State” on page 22-10

“Restore the HDL Workflow Advisor State” on page 22-12

How the Save and Restore Process Works

By default, the coder 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.

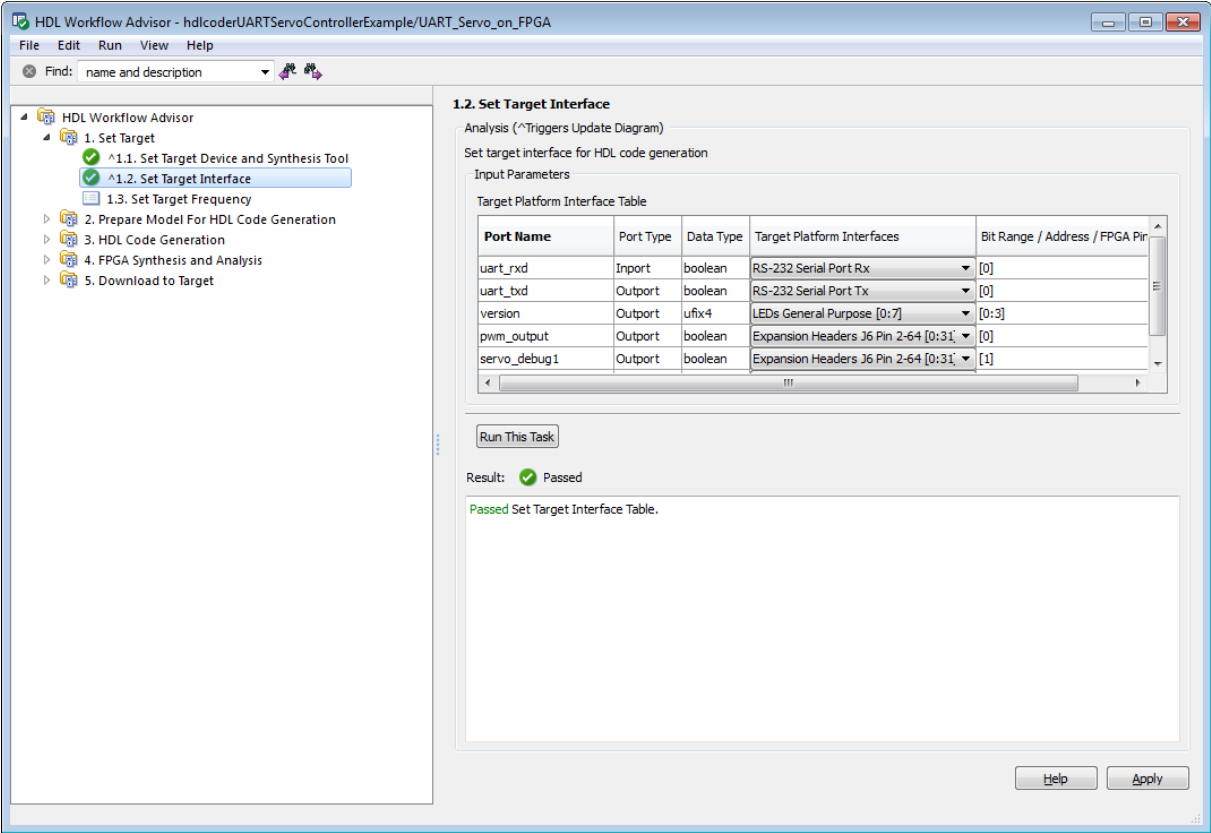
Limitations of the Save and Restore Process

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.

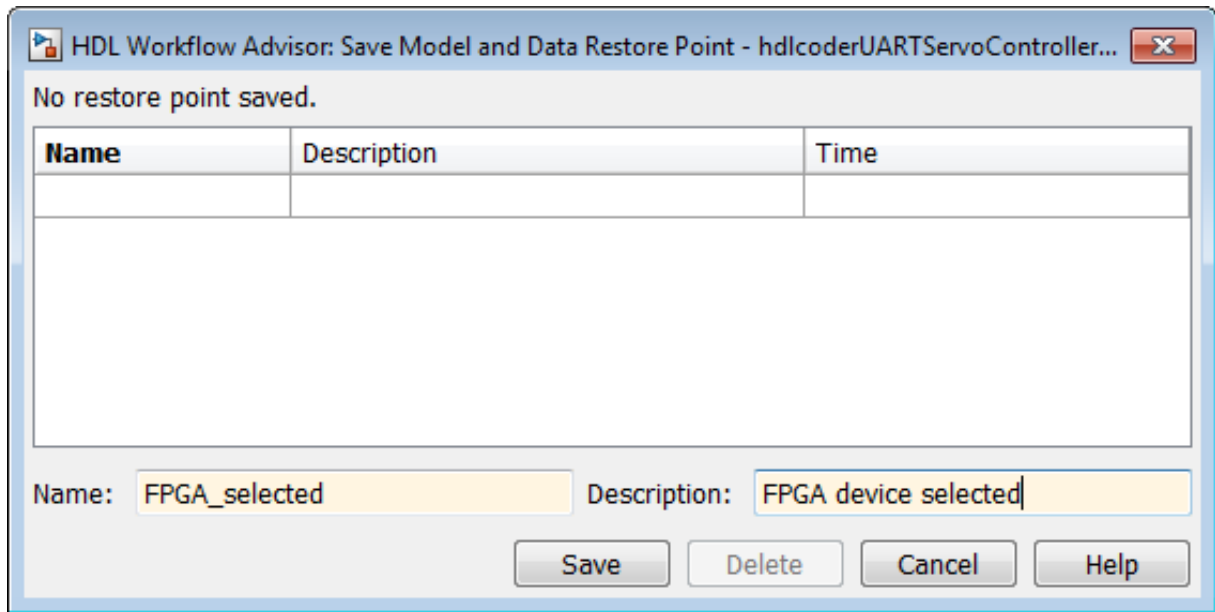
Save the HDL Workflow Advisor State

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.

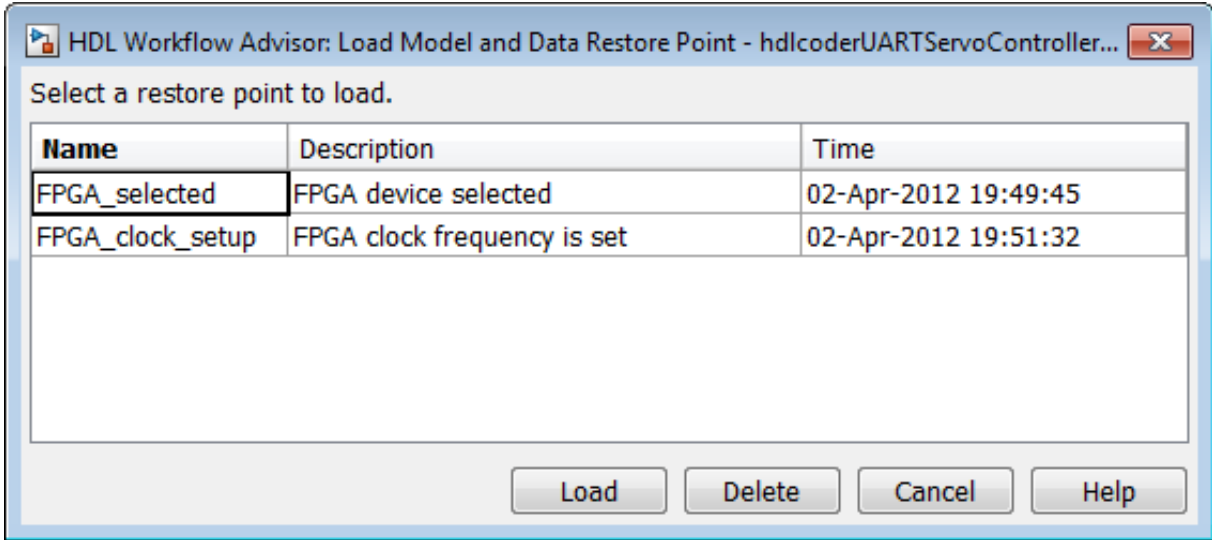


- 4 Click **Save**. The HDL Workflow Advisor saves a restore point of the current settings.

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**.

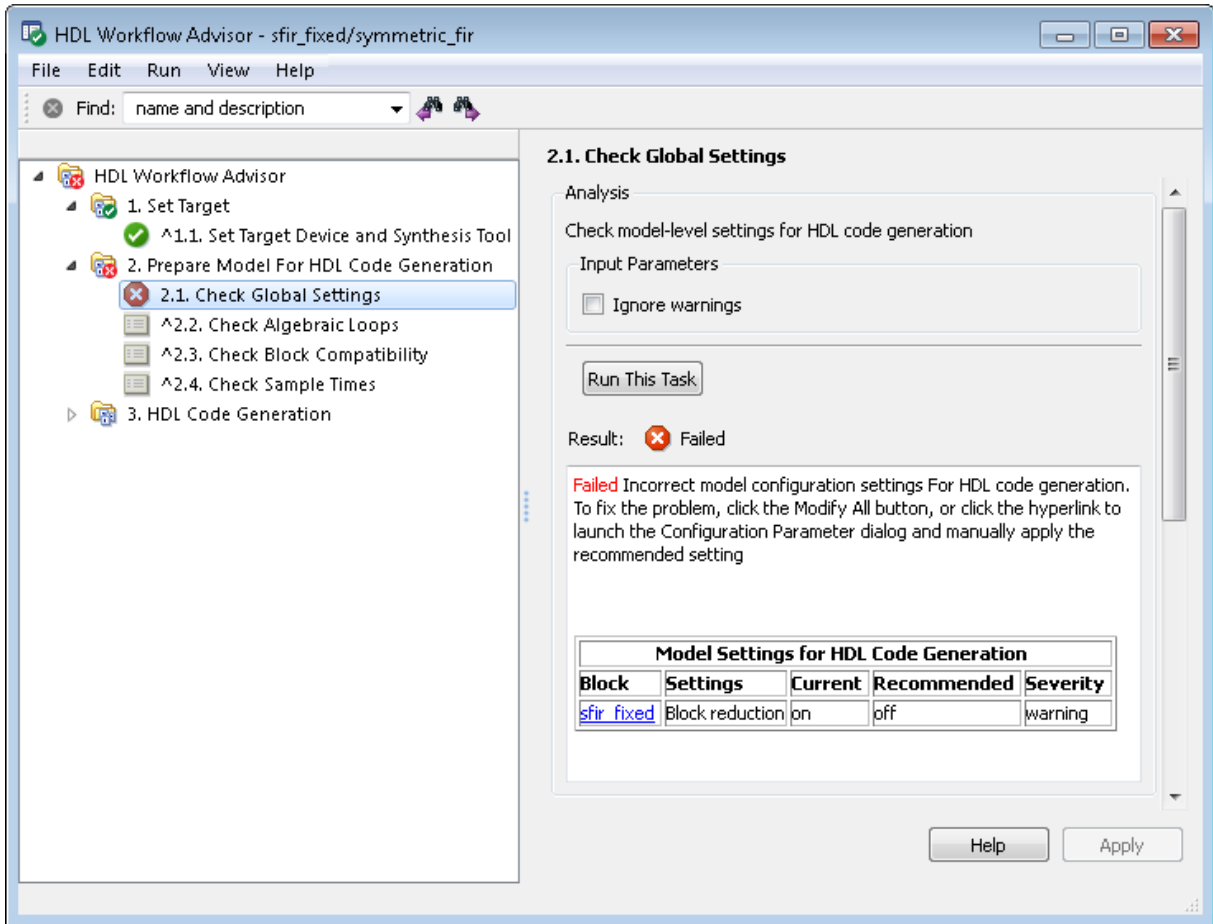
The HDL Workflow Advisor issues a warning that the restoration will overwrite current settings.

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 an **Analysis Result** subpane. The **Analysis 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 **Analysis Result** subpane automatically. In the following example, the **Check Global Settings** task has failed, displaying an incorrect model setting in the **Analysis Result** pane.



The Action subpane, below the Analysis Result subpane, contains a **Modify All** button. To fix the problems that appear in the Analysis Result subpane, click the **Modify All** button.

After you click **Modify All**, the Analysis Result subpane reports the changes that were applied. The task is set to a Not Run and enabled state, enabling you to rerun the task and proceed to the subsequent tasks.

Tip Review the **Analysis Result** box before automatically fixing failures. If you do not want to apply all of the recommended actions, do not click **Modify All** to fix warnings or failures.

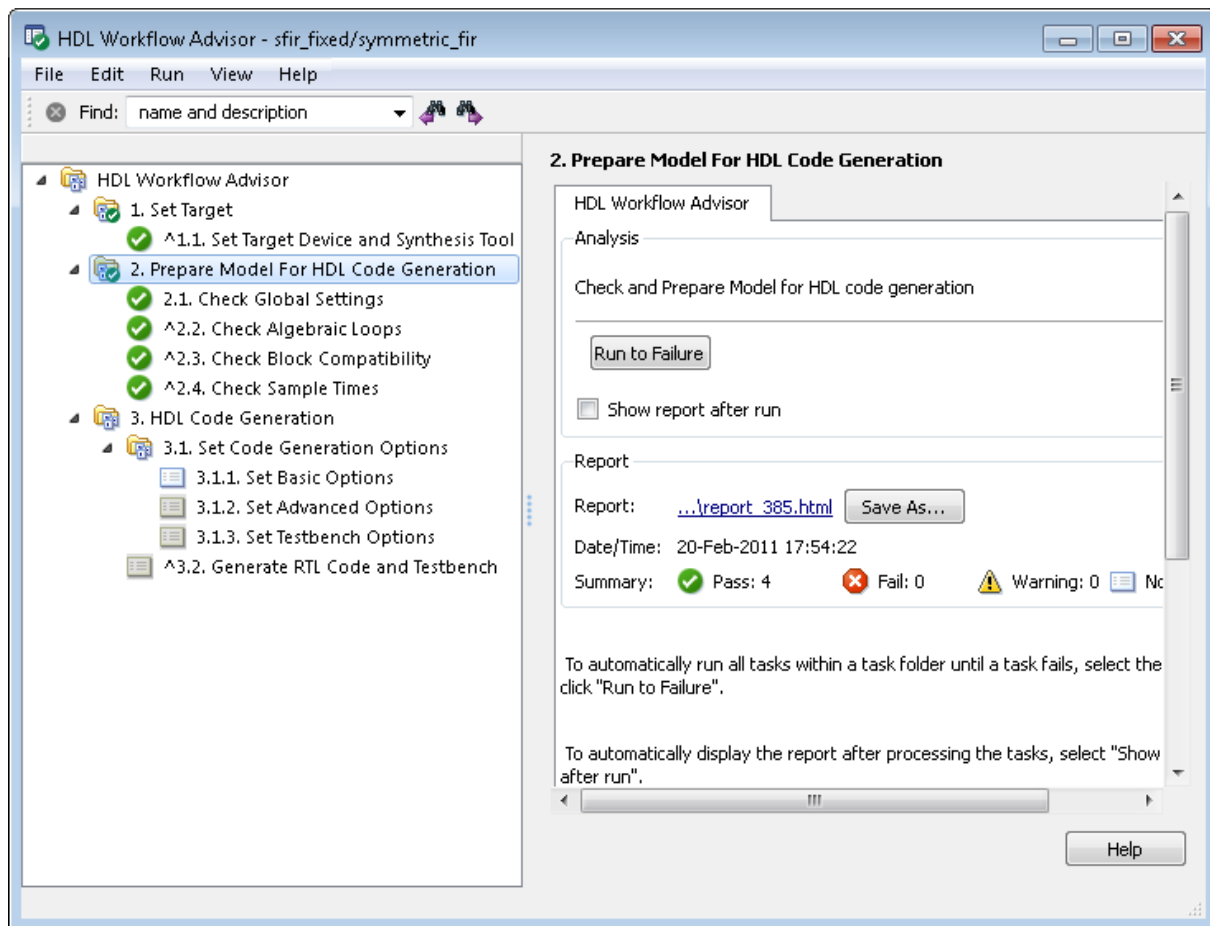
View and Save HDL Workflow Advisor Reports

In this section...
“Viewing HDL Workflow Advisor Reports” on page 22-17
“Saving HDL Workflow Advisor Reports” on page 22-21

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.

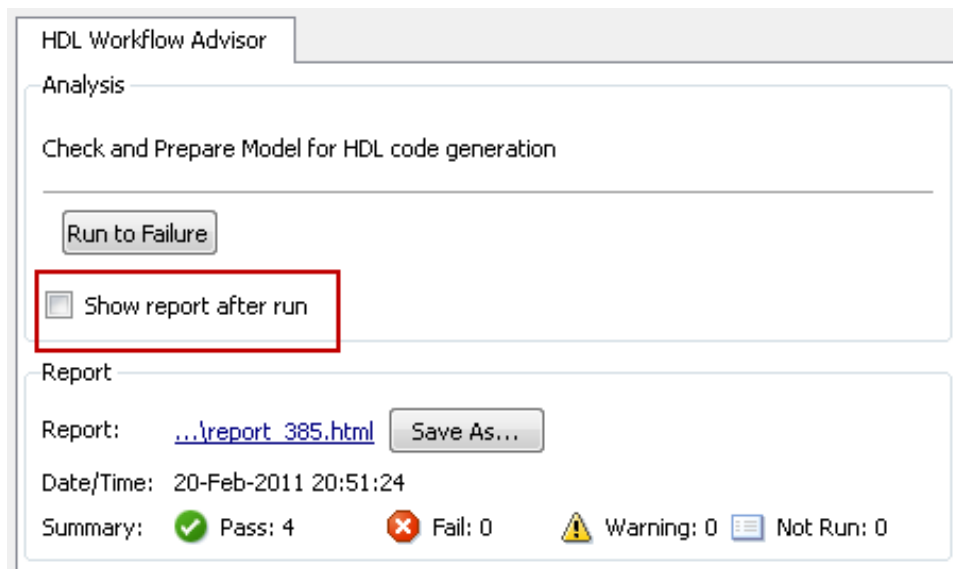
As you run checks, the HDL Workflow Advisor updates the reports with the latest information for each check in the folder. A message appears in the

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\`.

Map to an FPGA Floating-Point Library

In this section...

“What is an FPGA Floating-Point Library?” on page 22-22

“Why Map to an FPGA Floating Point Library?” on page 22-22

“Supported Floating-Point Operations” on page 22-22

“Setup for FPGA Floating-Point Library Mapping” on page 22-24

“How to Map to an FPGA Floating-Point Library” on page 22-24

“FPGA Floating-Point Library Mapping Results Analysis” on page 22-26

“Limitations for FPGA Floating-Point Library Mapping” on page 22-26

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

The 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

The 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

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”.

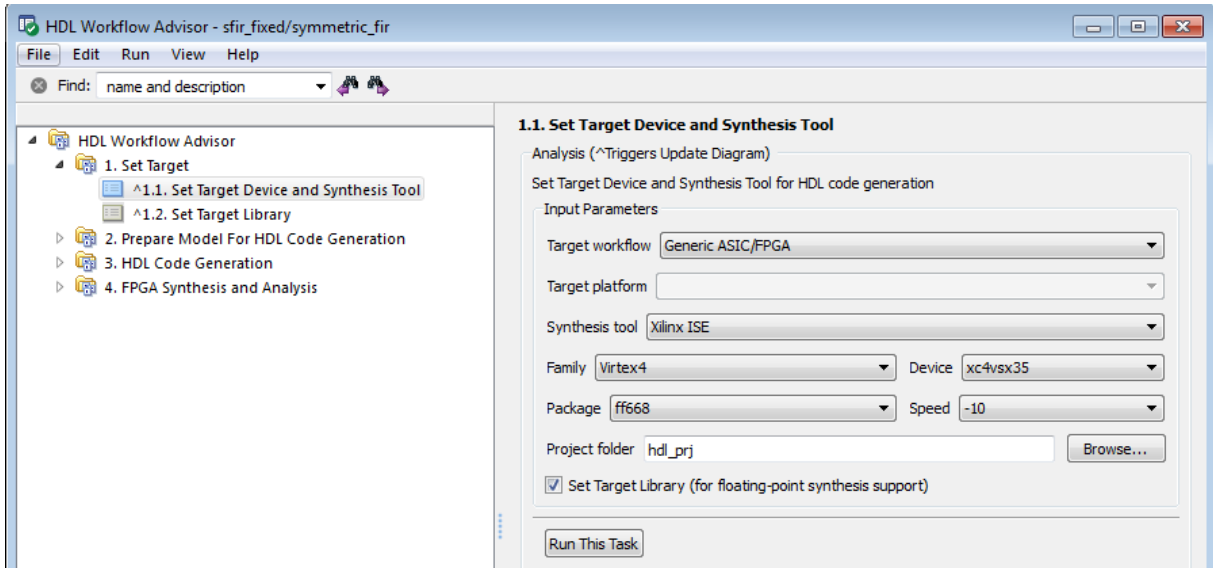
Note If you are using Altera Quartus 10.1 or 11.0, you must turn on the `AlteraBackwardIncompatibleSinCosPipeline` global property using `hdlset_param`. For example, to turn on `AlteraBackwardIncompatibleSinCosPipeline` for a model, `my_dut`, enter the following at the command line:

```
hdlset_param('my_dut', 'AlteraBackwardIncompatibleSinCosPipeline', 'on')
```

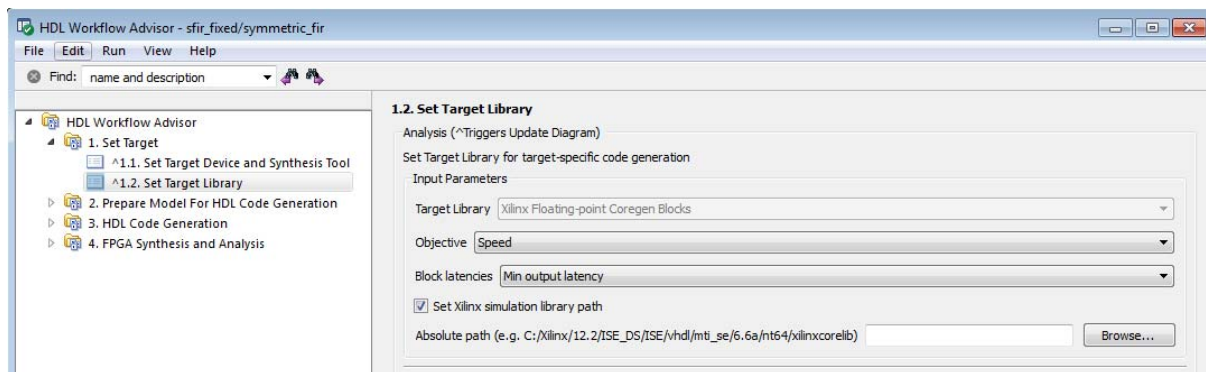
How to Map to an FPGA Floating-Point Library

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.
- 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, the coder automatically detects the simulation library path.

FPGA Floating-Point Library Mapping Results Analysis

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” on page 16-2.

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” on page 16-5.

The Optimization Report shows whether the 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” on page 16-7.

Limitations for 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
- Triggered subsystem

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 $u2 \neq 0$.
- Sum of Elements with an architecture other than Tree.
- Product of Elements with an architecture other than Tree.

Unsupported HDL Workflow Advisor modes:

- Cosimulation
- FPGA-in-the-Loop
- FPGA Turnkey
- xPC Target FPGA I/O

FPGA Synthesis and Analysis

In this section...

“FPGA Synthesis and Analysis Tasks Overview” on page 22-28

“Creating a Synthesis Project” on page 22-28

“Performing Synthesis, Mapping, and Place and Route” on page 22-30

“Annotating Your Model with Critical Path Information” on page 22-35

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

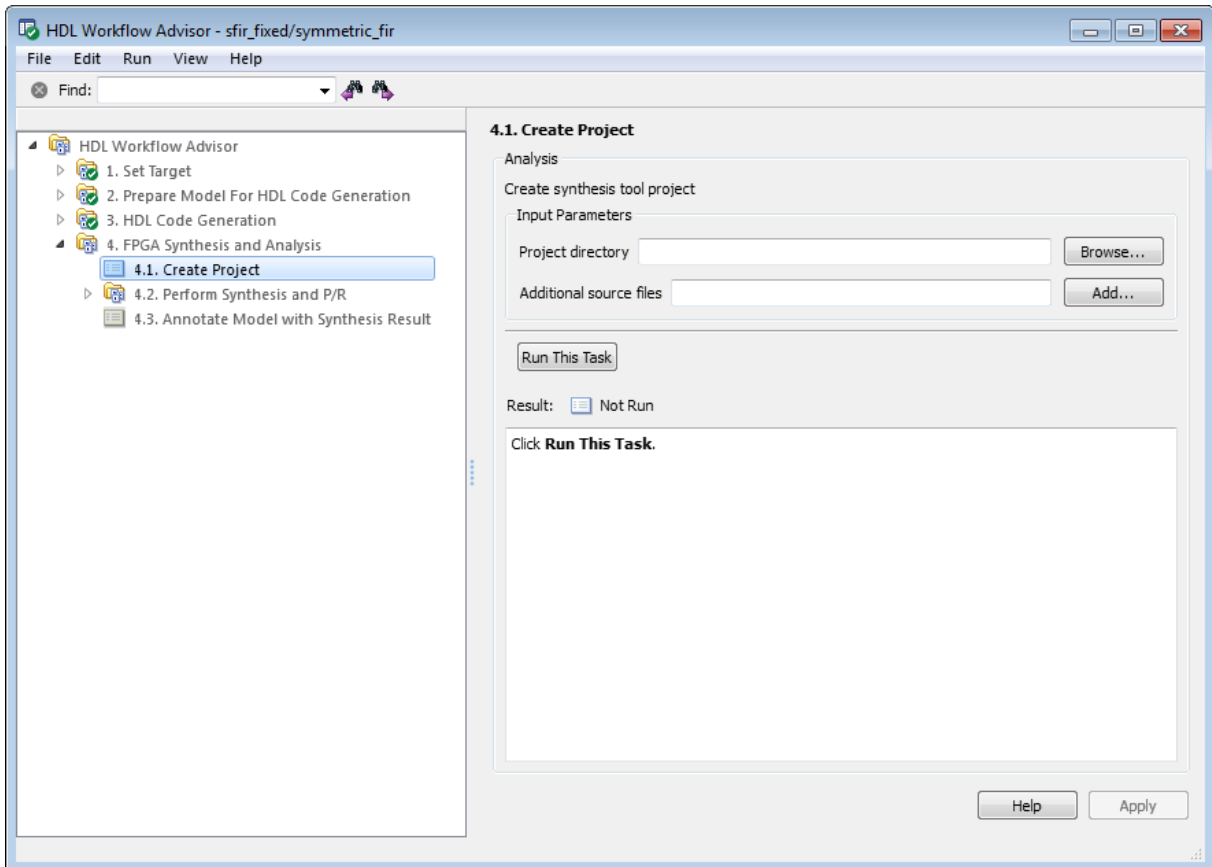
Note A supported synthesis tool must be installed, and the synthesis tool executable must be on the system path to perform the tasks in the **FPGA Synthesis and Analysis** folder. See “Third-Party Synthesis Tools” for more information.

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.

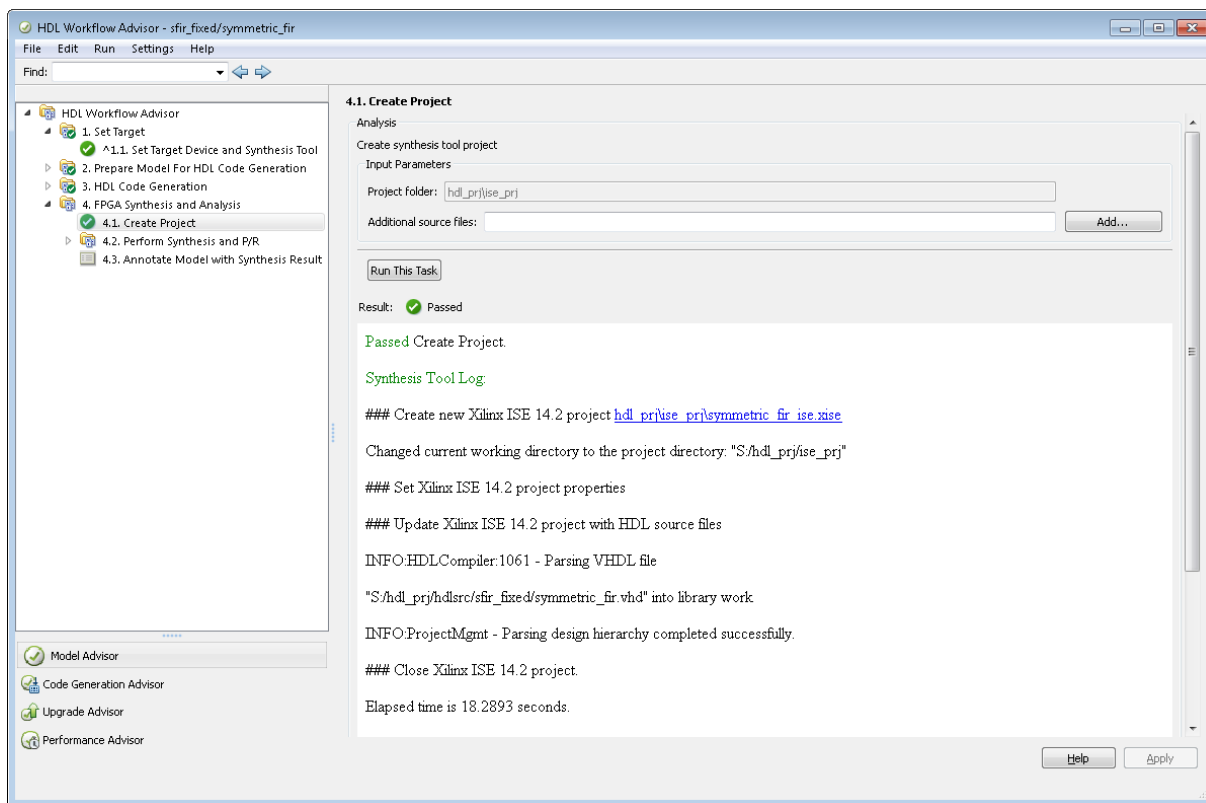


The **Create Project** task parameters are:

- **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.



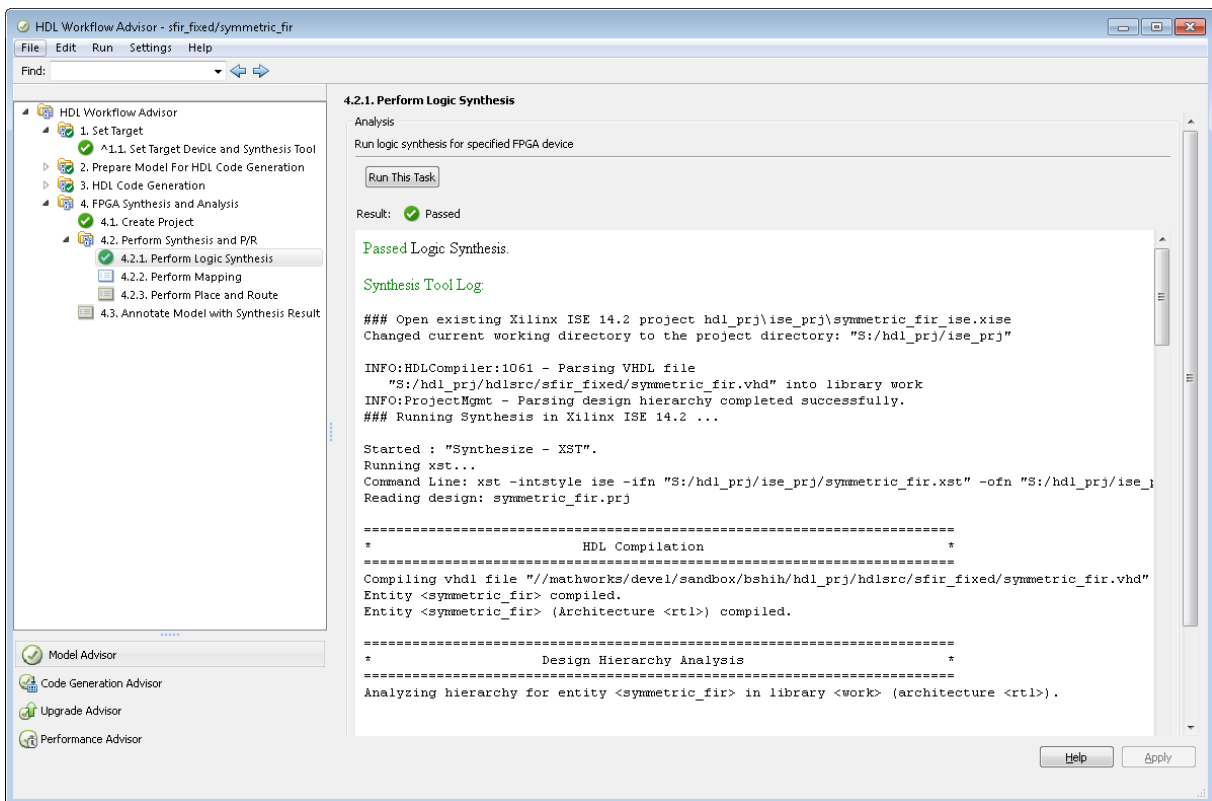
Performing Synthesis, Mapping, and Place and Route

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.



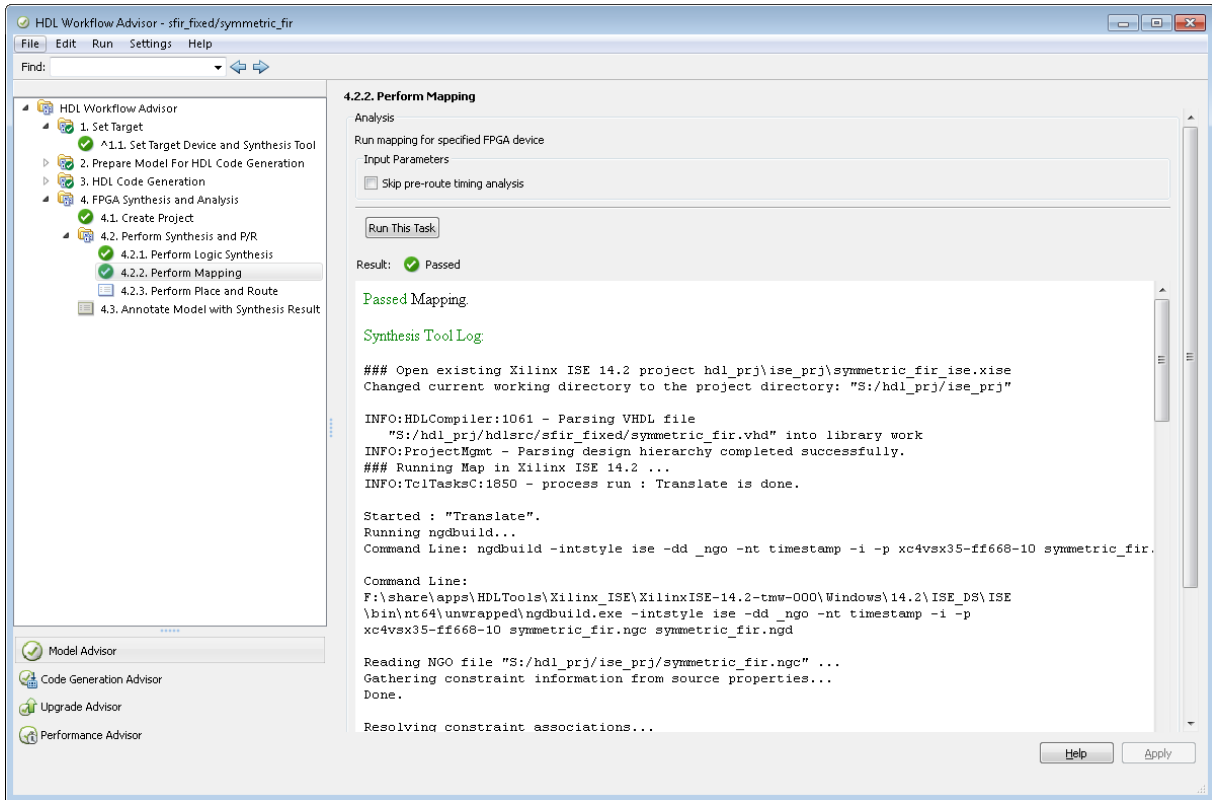
Performing Mapping

The **Perform Mapping** task does the following:

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

If your tool does not support early timing estimation, you can enable **Skip pre-route timing analysis**. When this option is enabled, the **Annotate Model with Synthesis Result** task sets **Critical path source** to **post-route**.

The following figure shows the HDL Workflow Advisor after passing the **Perform Mapping** task.



Performing Place and Route

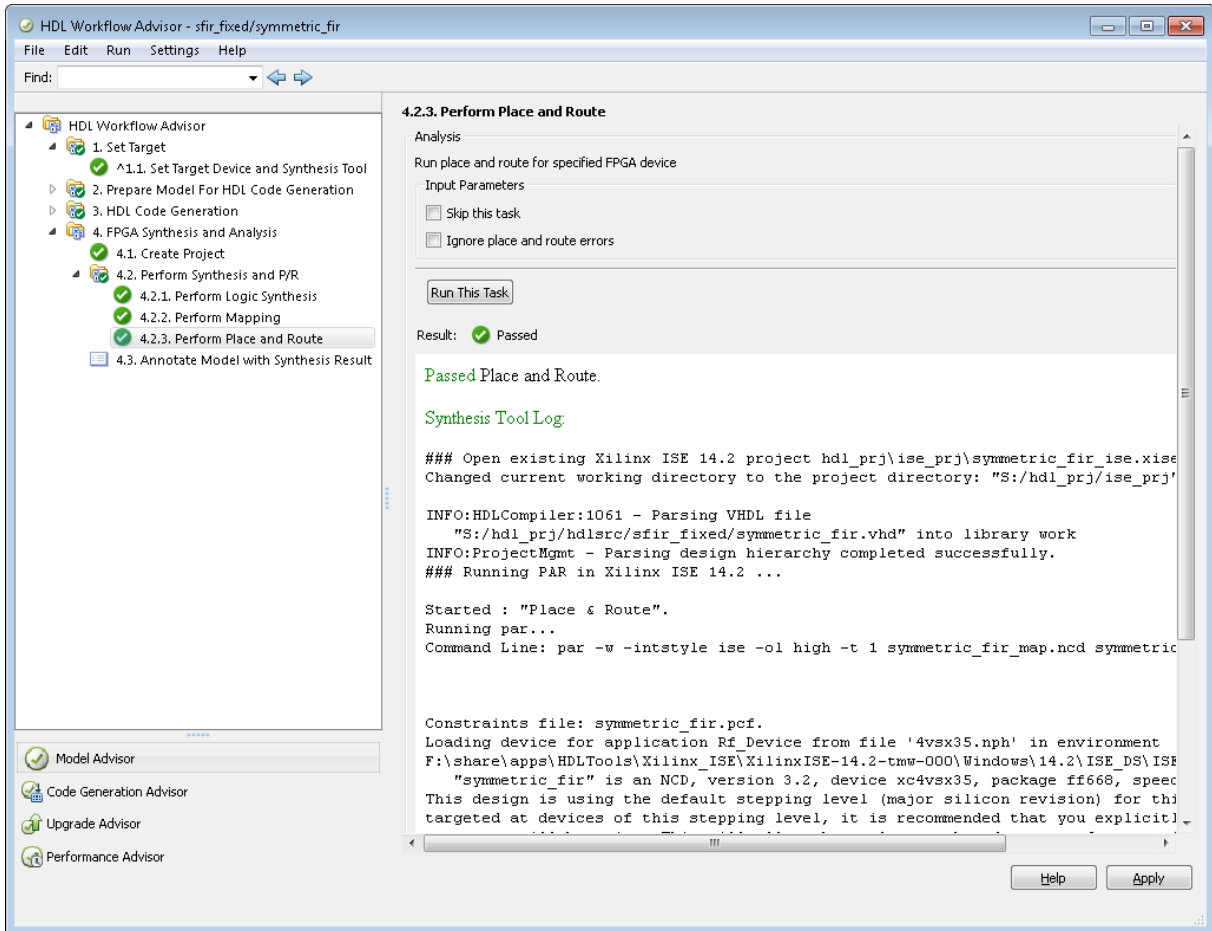
The **Perform Place and Route** task does the following:

- 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.

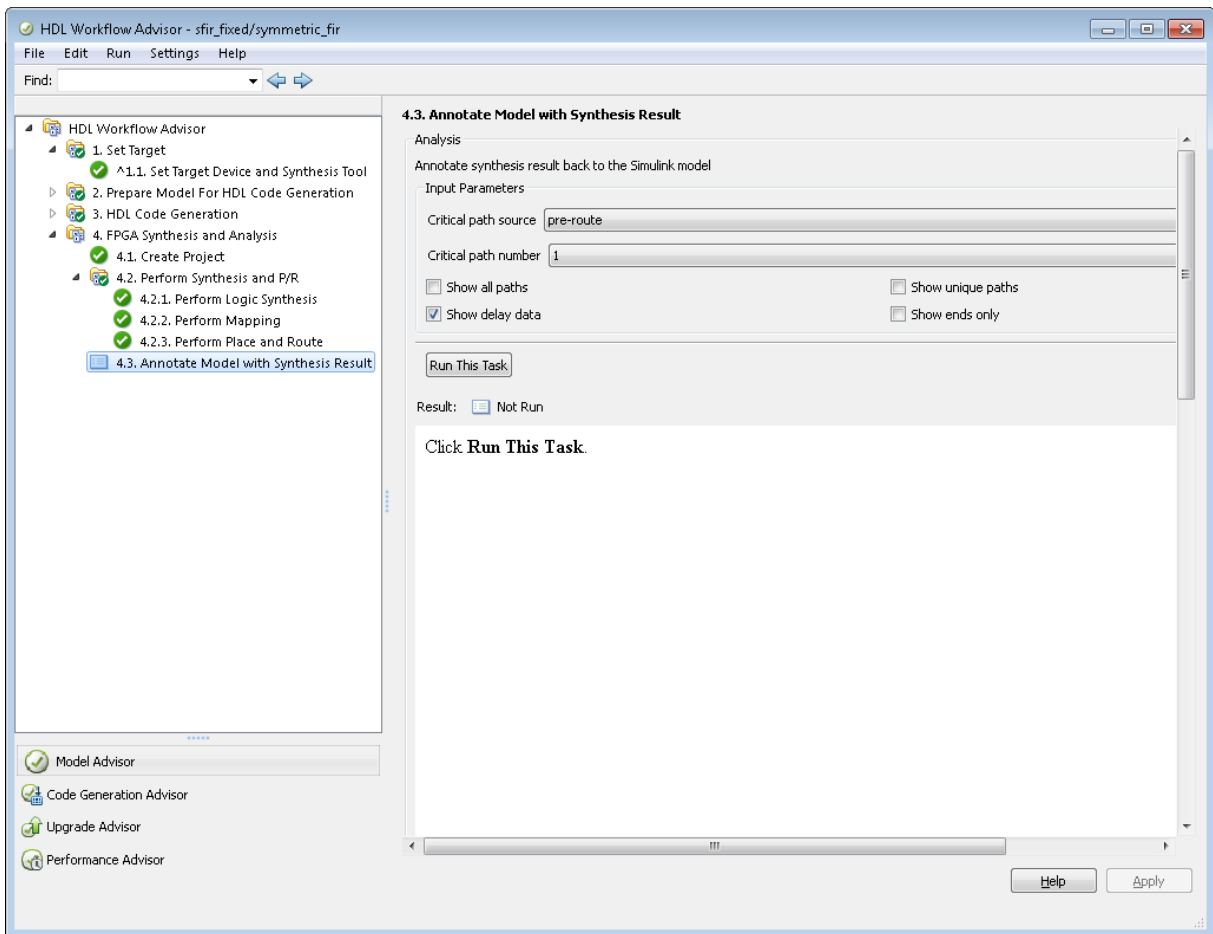


Annotating Your Model with Critical Path Information

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.



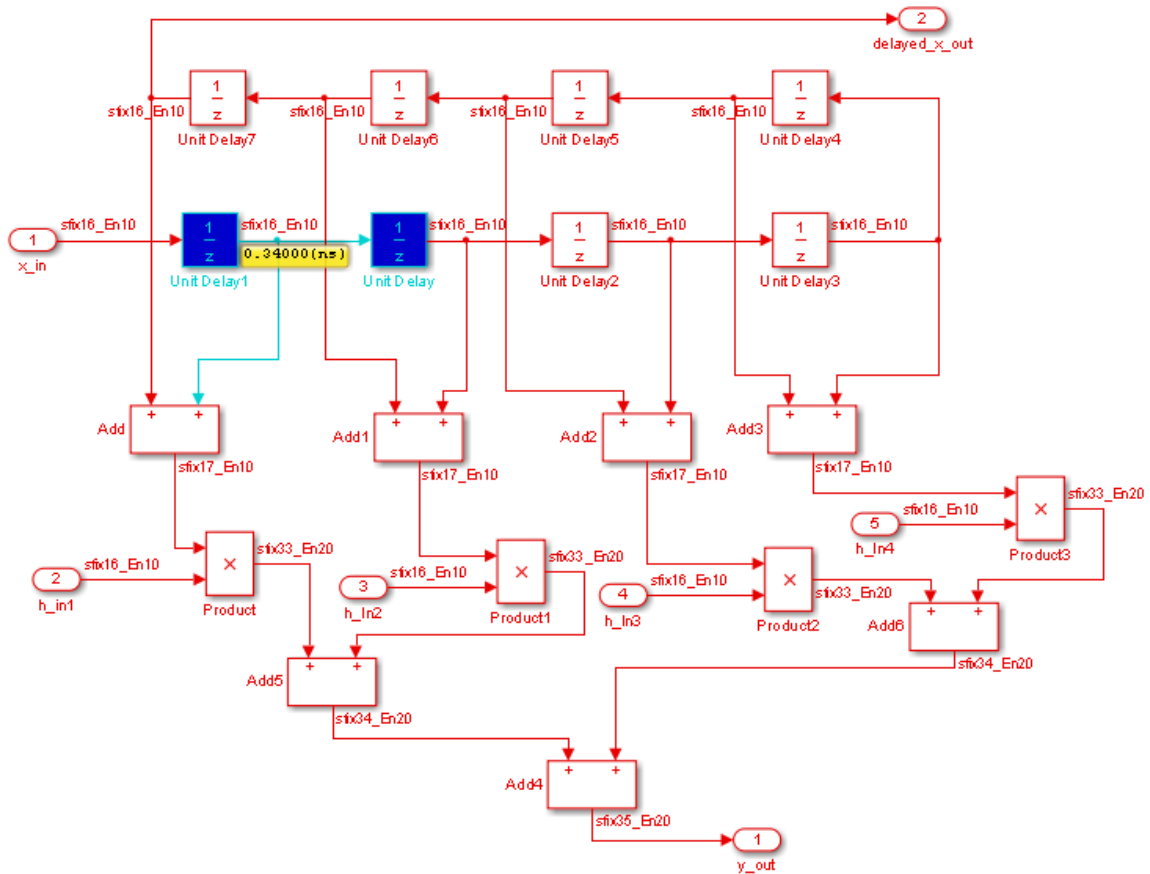
The task parameters are:

- **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.

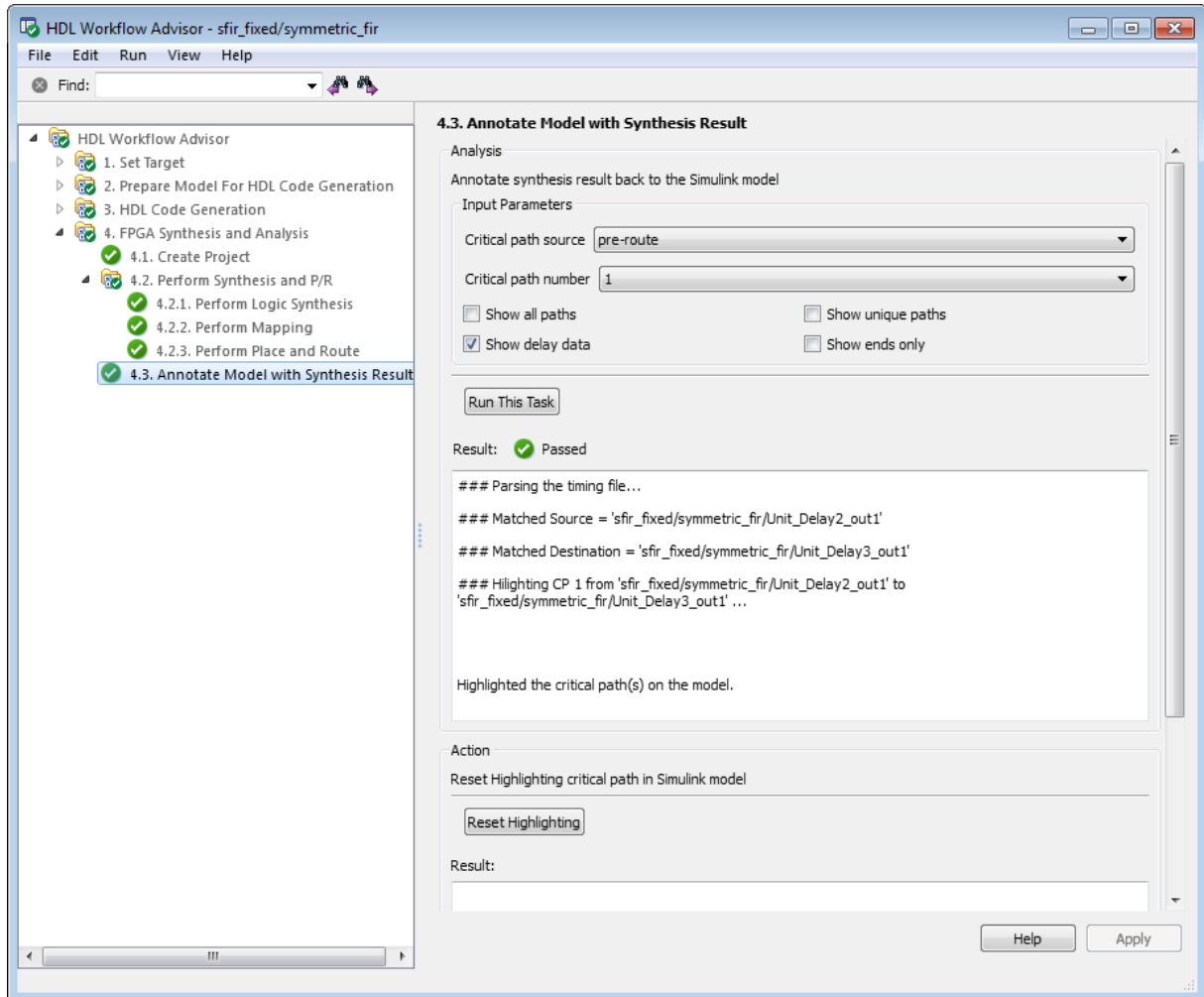
- **Critical path number:** You can annotate up to 3 critical paths. Select the number of paths you want to annotate. The default is 1.
- **Show all paths:** Show critical paths, including duplicate paths. The default is **off**.
- **Show unique paths:** Show only the first instance of a path that is 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, the 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.



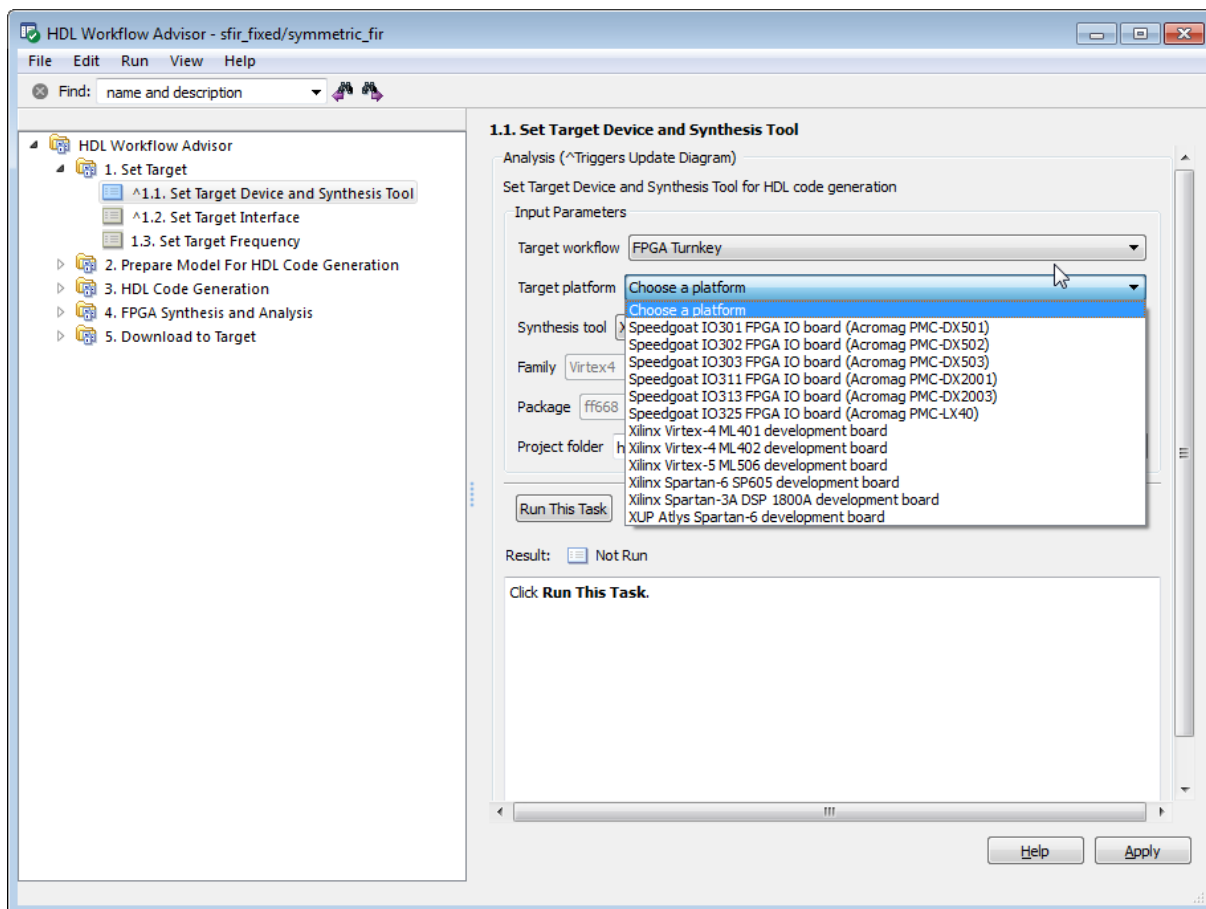
After the **Annotate Model with Synthesis Result** task runs to completion, the HDL Workflow Advisor enables the **Reset Highlighting** button in the **Action** subpane. When you click this button, the HDL Workflow Advisor:

- 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. The **Target platform** menu of the **Set Target Device and Synthesis Tool** task 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. See the following sections for information on automated workflows for specific types of targets:

- “Generate xPC Target Interface for Speedgoat Boards” on page 22-42
- “Target Xilinx FPGA Development Boards” on page 22-58

Generate xPC Target Interface for Speedgoat Boards

In this section...

“Select a Speedgoat Target Device” on page 22-42

“Set the Target Interface for Speedgoat Boards” on page 22-45

“Code Generation, Synthesis, and Generation of xPC Target Interface Subsystem” on page 22-48

This example shows how to generate a hardware-in-the-loop interface for Speedgoat board programming with xPC Target using the **xPC Target FPGA I/O** workflow.

To run this example, you must:

- Have a license for xPC Target software.
- Use Xilinx ISE 10.1.

Select a Speedgoat Target Device

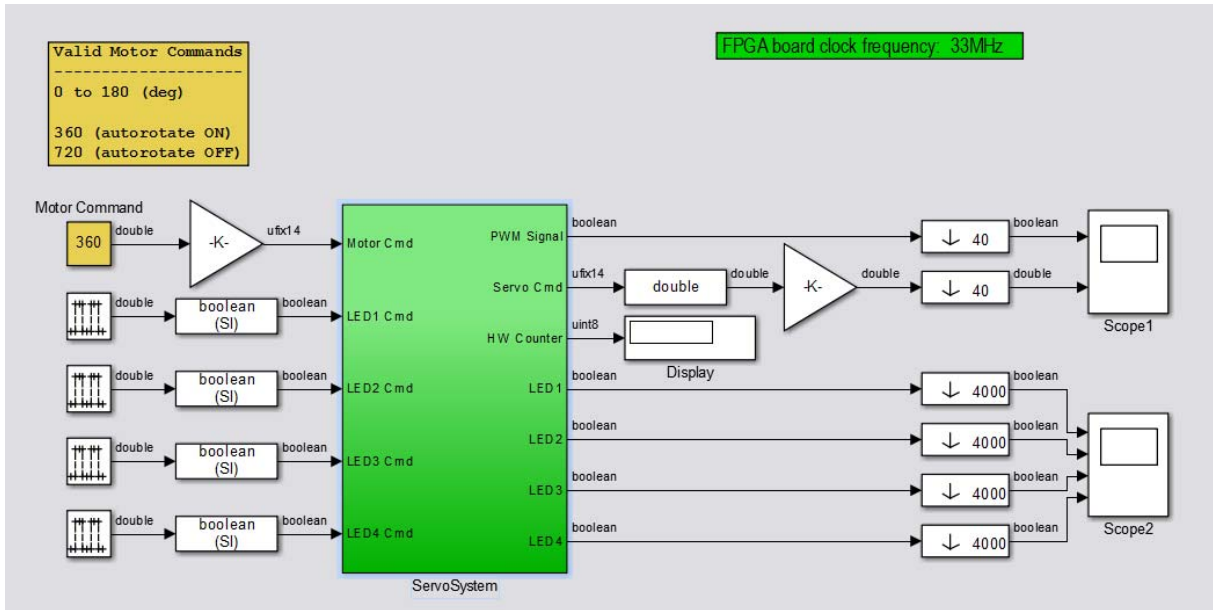
Note Before selecting a Speedgoat target device, see “Third-Party Synthesis Tools”.

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.

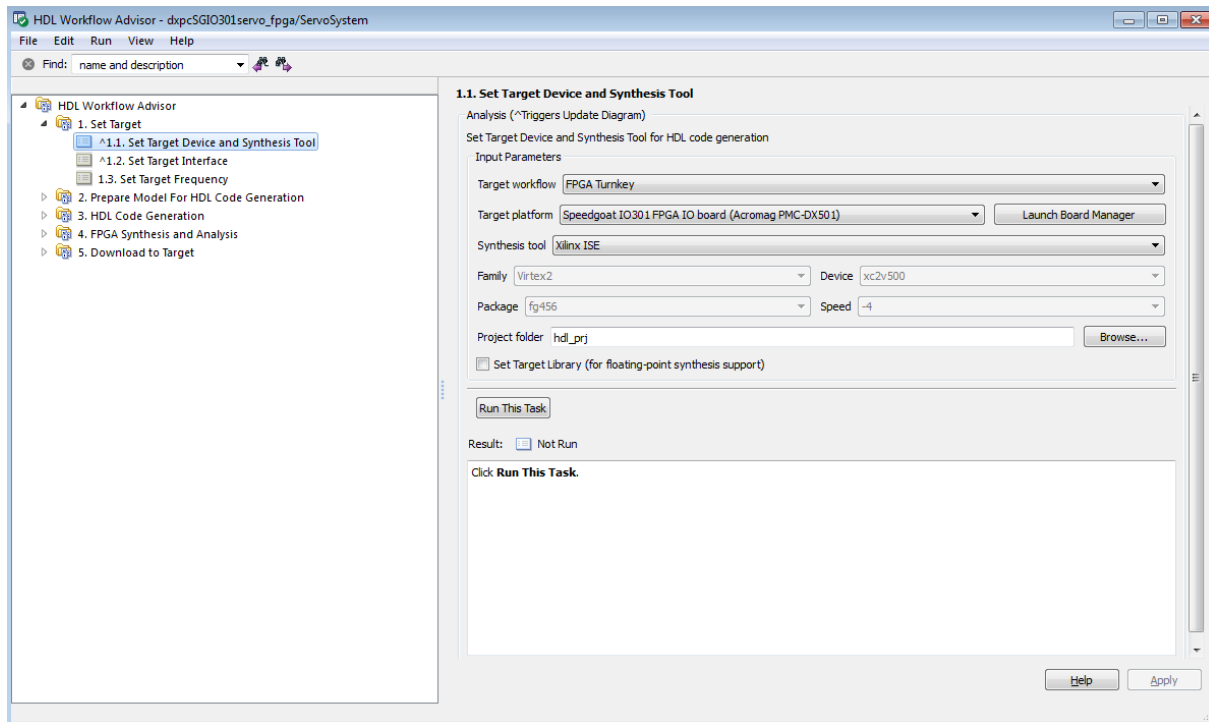


- 2 Right-click the ServoSystem block, and select **HDL Code > HDL Workflow Advisor**.
- 3 In the HDL Workflow Advisor, select **Set Target > Set Target Device and Synthesis Tool**.
- 4 For **Target workflow**, select **xPC Target FPGA I/O**.

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.

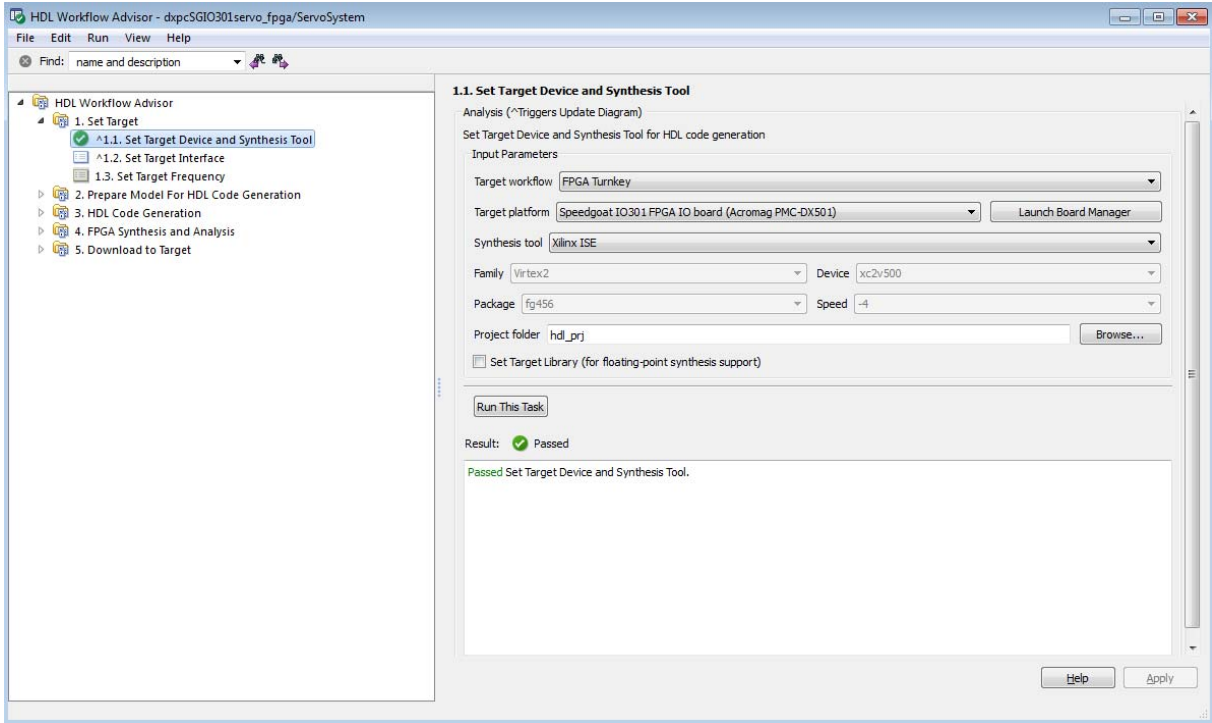
- 5 From the **Target platform** menu, select the Speedgoat IO301 board.

xPC Target and HDL Workflow Advisor support the same set of Speedgoat devices. For a list of supported boards, see “FPGA Support”.



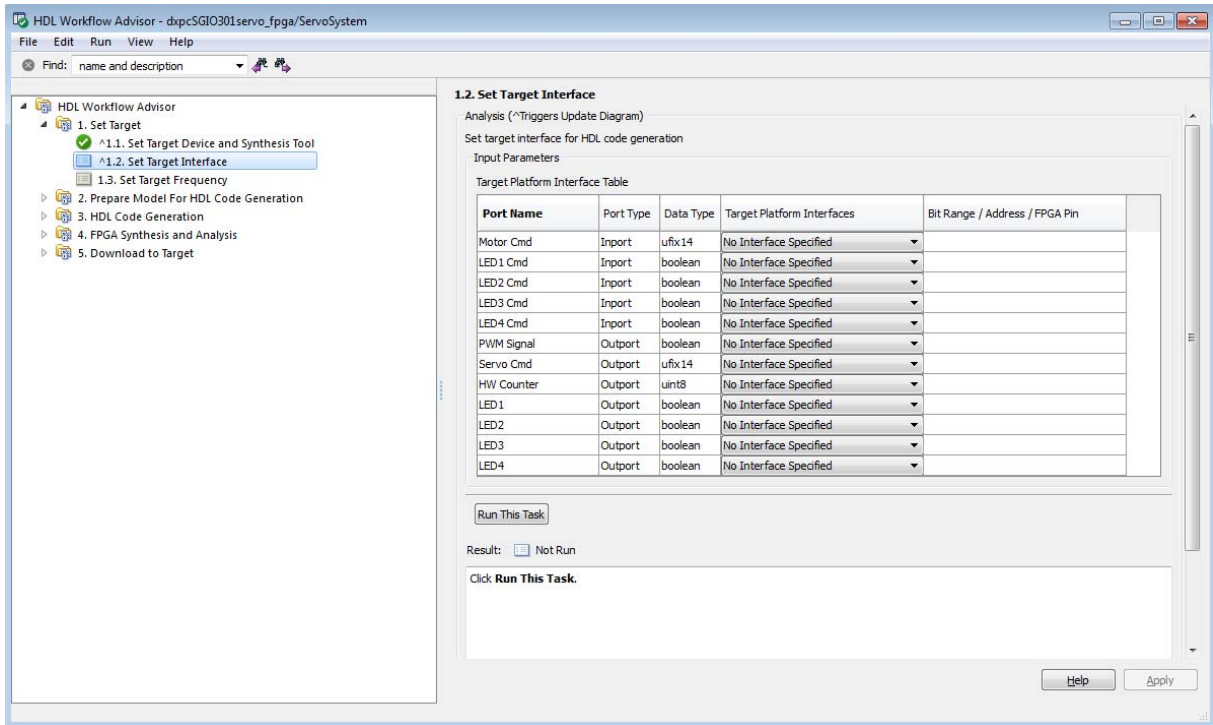
6 Click Run This Task.

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

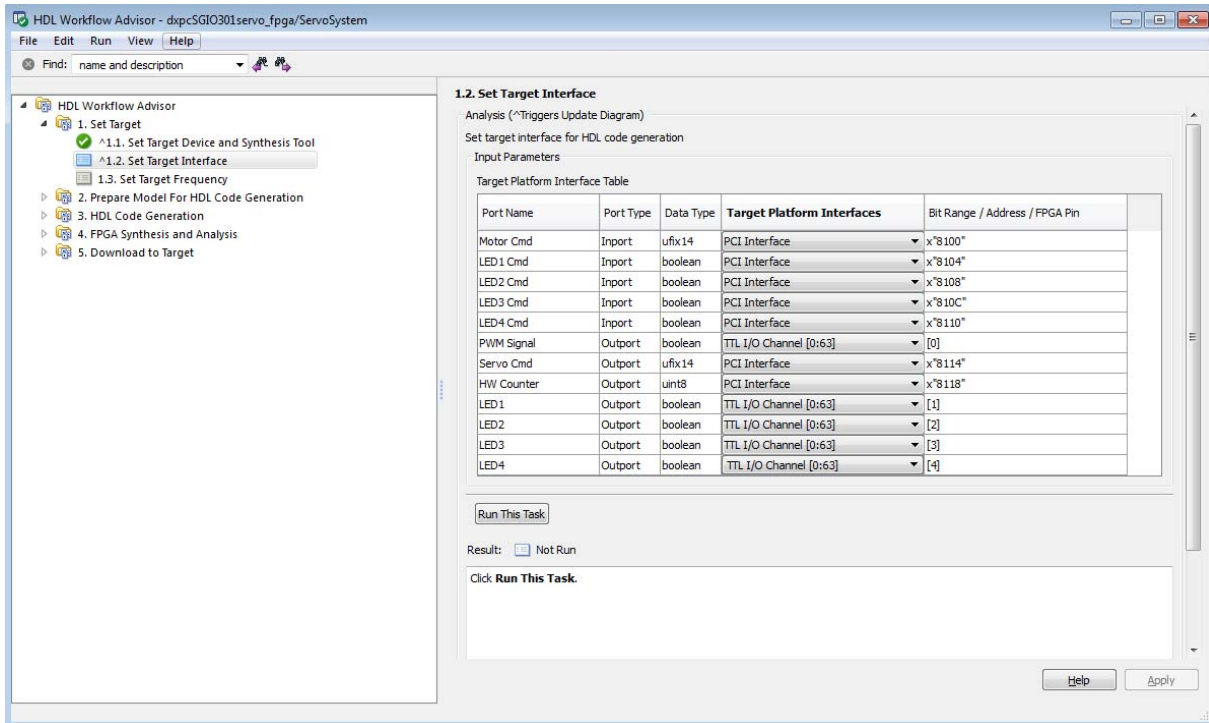
The **Set Target Interface** task in the HDL Workflow Advisor enables you to define how the inputs and outputs of the DUT map to the inputs and outputs of your Speedgoat target device.



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 xPC Target 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:

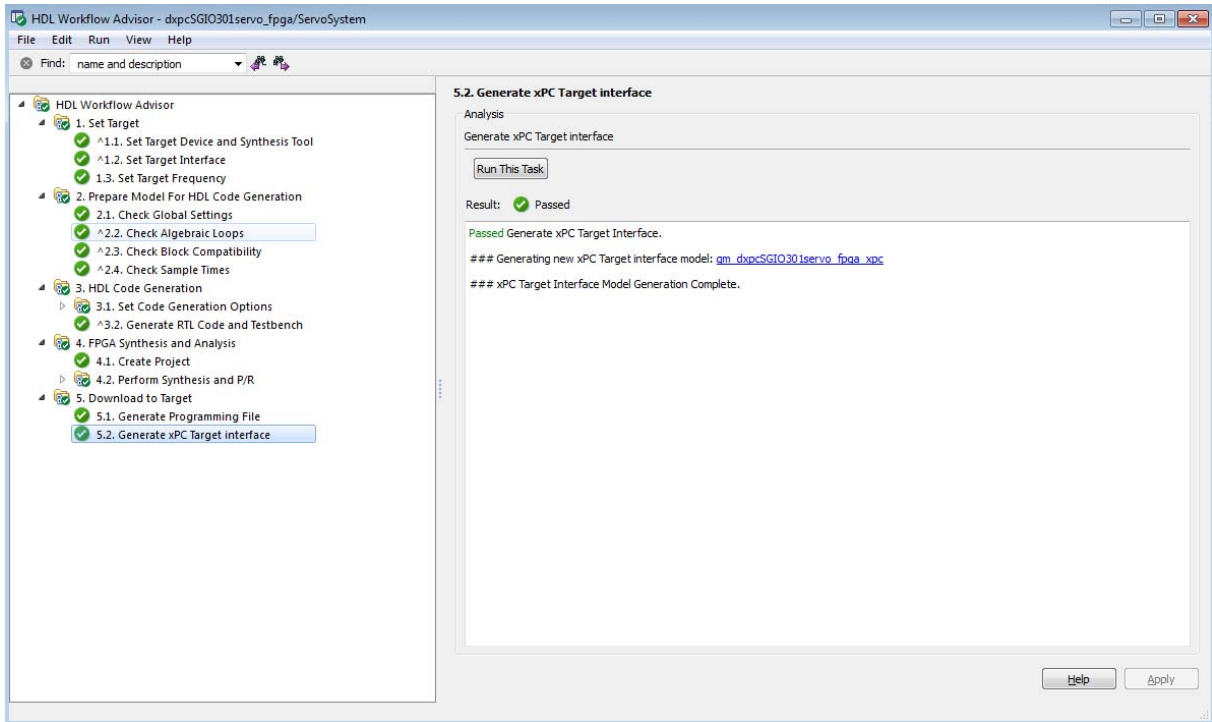
- **Prepare Model For HDL Code Generation:** Checking the model for HDL code generation compatibility.
- **HDL Code Generation:** Setting HDL-related options of the Model Configuration Parameters dialog box and generating HDL code.
- **FPGA Synthesis and Analysis:** Executing synthesis and timing analysis 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 an xPC Target 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 an xPC Target model.

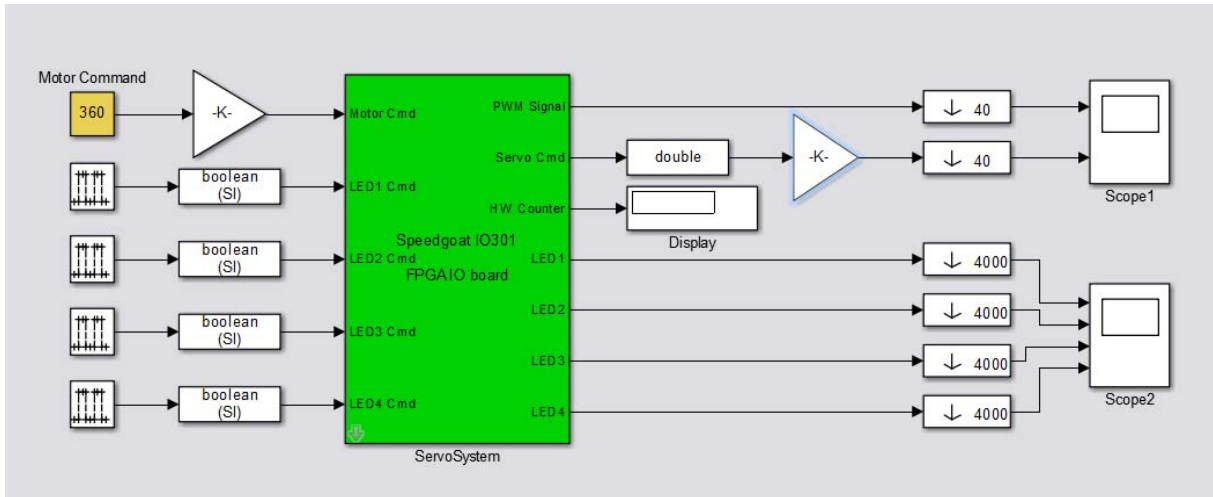
To run this sequence of tasks automatically:

- 1** Open the **Download to Target** task group.
- 2** Right-click **Generate xPC Target 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_dxcSGIO301servo_fpga_xpc`. Click the link to open the model.



The model contains the xPC Target interface subsystem. This new subsystem replaces the DUT (ServoSystem) in the original model. It replaces the internals of the original DUT with an xPC Target 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 xPC Target, see “FPGA Models” in the xPC Target documentation.

Target Altera FPGA Development Boards

In this section...

“Before You Begin” on page 22-51

“Open the Model” on page 22-51

“Select the Target Device” on page 22-52

“Set Target Interface and Target Frequency” on page 22-53

“Generate Code, Synthesize, and Program Target Device” on page 22-56

This example shows how to target an Altera FPGA development board for synthesis using the FPGA Turnkey workflow.

The `hdlcoderUARTServoControllerExample` model is designed to work with an Altera DE2-115 development and education board. The `UART_Servo_on_FPGA` subsystem receives commands through UART ports. The subsystem generates a pulse width modulation (PWM) waveform to control a servo motor.

Before You Begin

To run this example, you must have your synthesis tool set up. To learn how to set up your synthesis tool, see “Synthesis Tool Path Setup”.

This example uses the Altera DE2-115 development and education board. You can try this example with a different board, and specify the target interface according to that board’s interface definition. To see a list of boards supported for the FPGA Turnkey workflow, see “FPGA Turnkey Hardware”.

If you want to download the programming file, you must first connect the target device. However, if the target device is not connected, you can still generate the programming file.

Open the Model

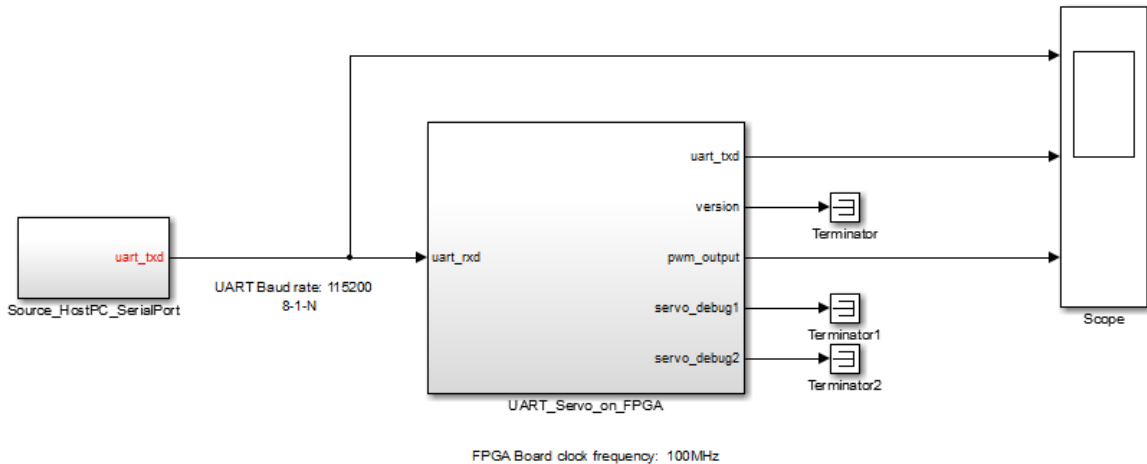
- 1 Add the example directory to your MATLAB path.

```
addpath(fullfile(docroot, 'toolbox', 'hdlcoder', 'examples'))
```

2 Open the model.

```
hdlcoderUARTServoControllerExample
```

Example of an FPGA Servo Motor Controller with UART interface

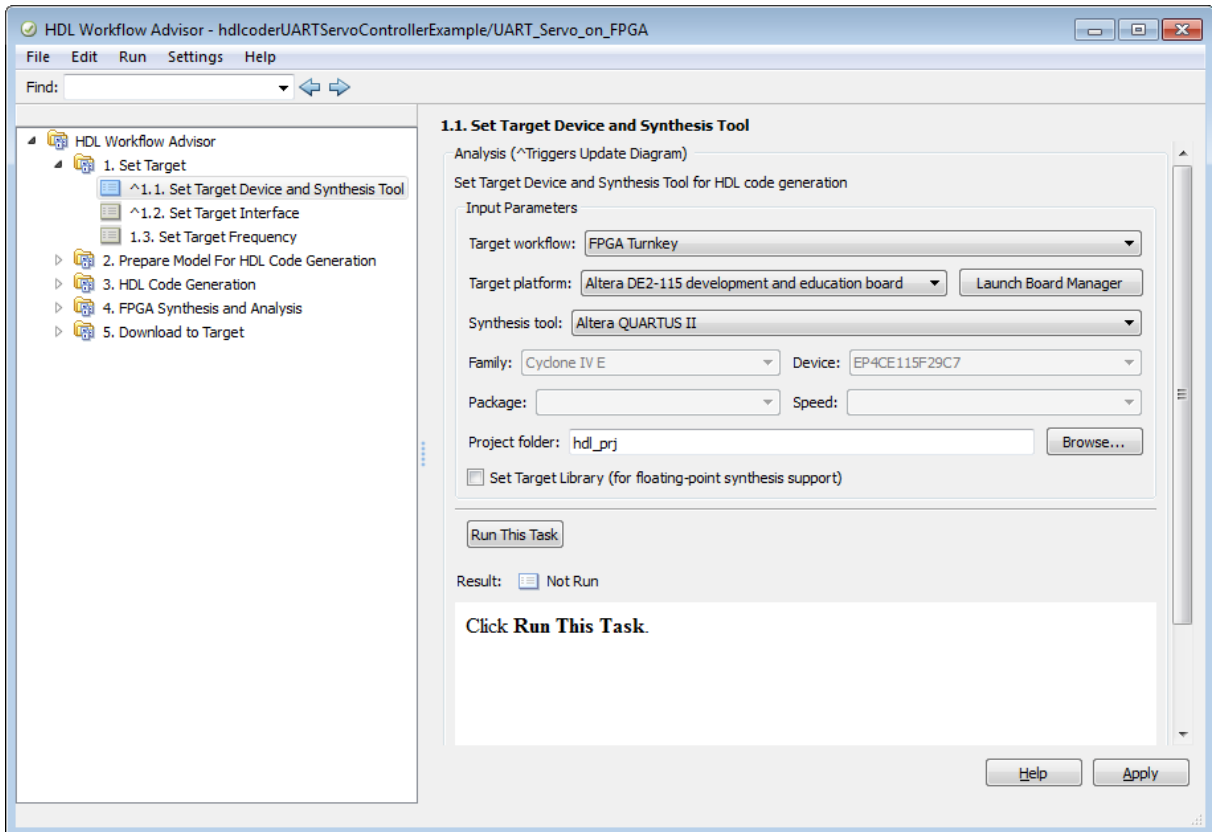


Select the Target Device

- 1** Right-click the `UART_Servo_on_FPGA` subsystem and select **HDL Code > HDL Workflow Advisor**.
- 2** In the HDL Workflow Advisor, select the **Set Target > Set Target Device and Synthesis Tool** task.
- 3** For **Target workflow**, select **FPGA Turnkey**.
- 4** For **Target platform**, select **Altera DE2-115 development and education board**.

If the board does not automatically appear in the list, select **Get more boards** to download the Altera FPGA Boards support package.

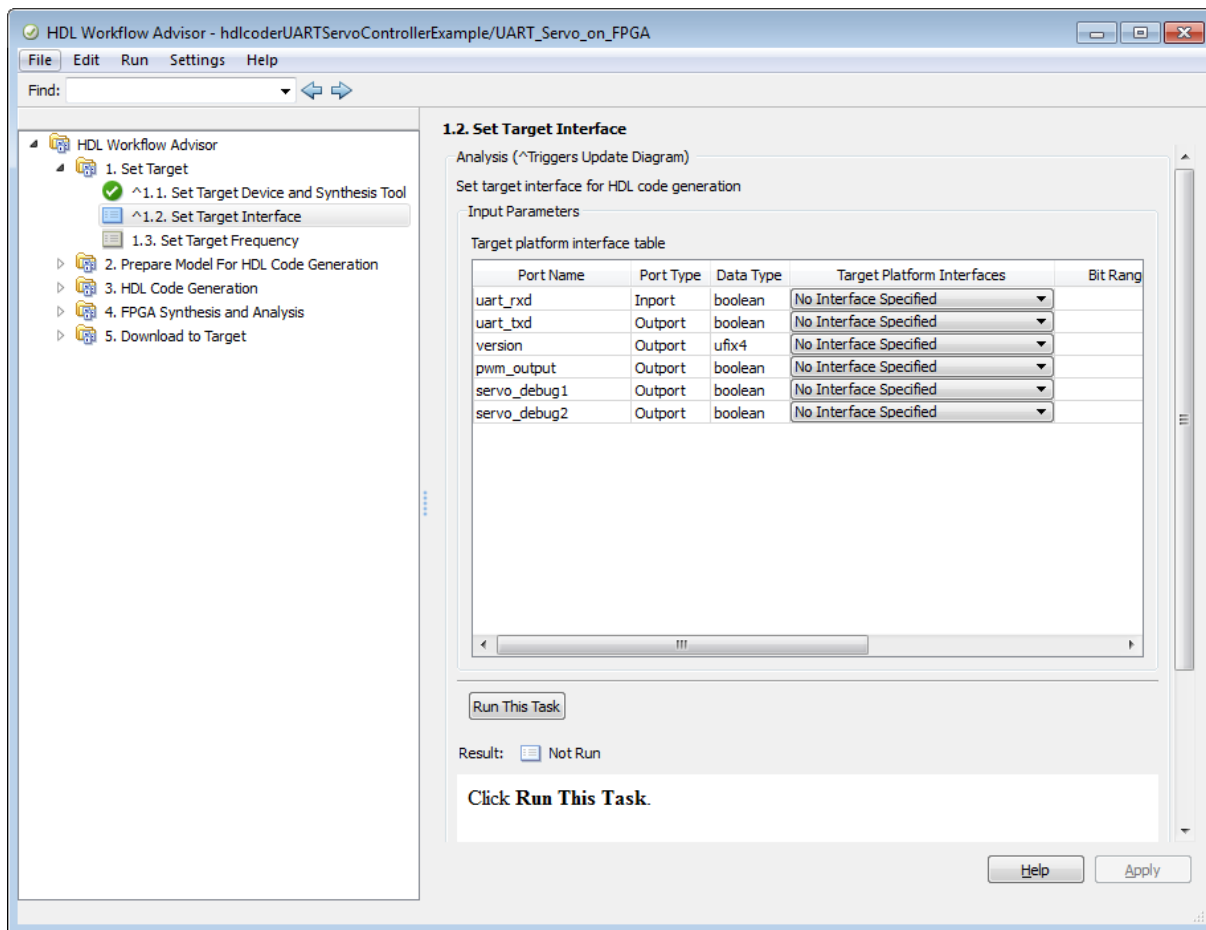
The HDL Workflow Advisor automatically sets the synthesis tool based on your board selection.



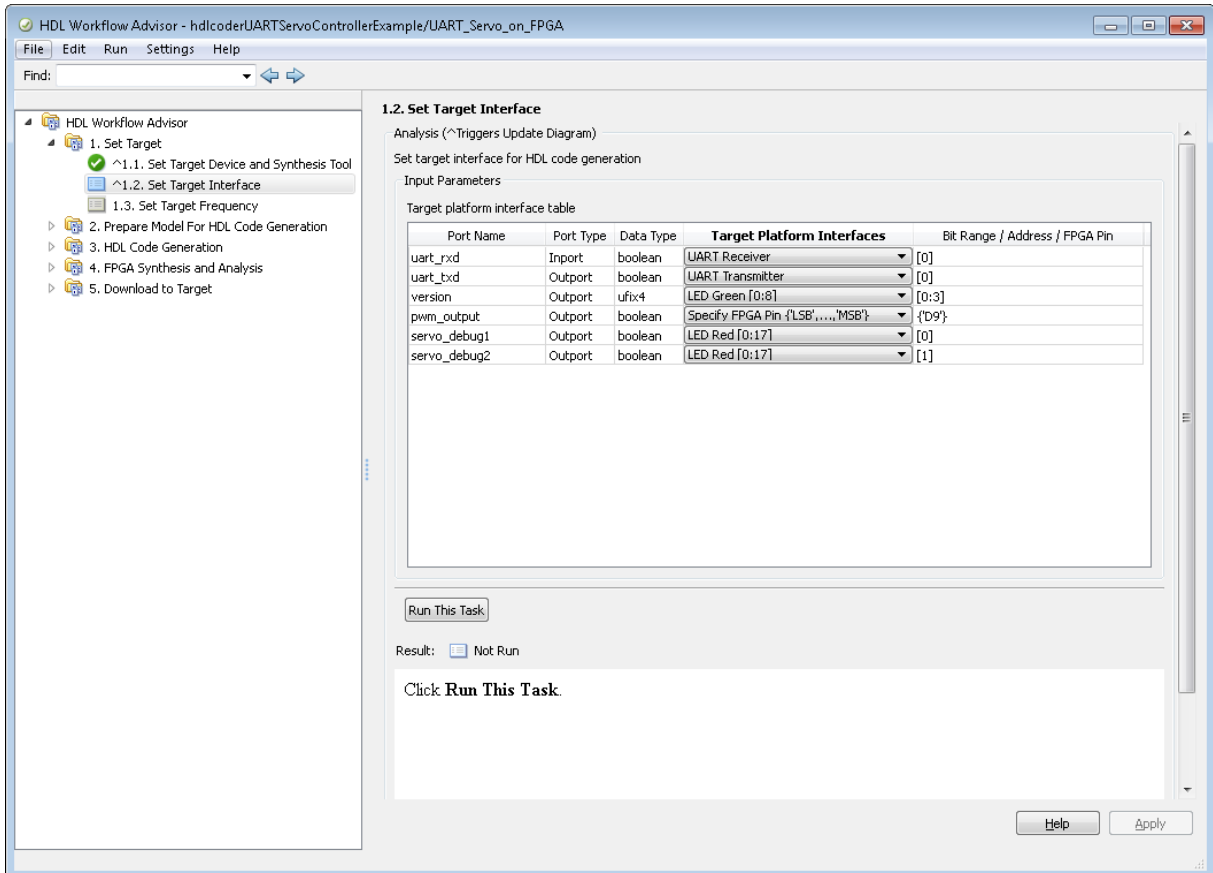
5 Click **Run This Task** .

Set Target Interface and Target Frequency

1 In the left pane of the HDL Workflow Advisor, select the **Set Target Interface** task.



2 For each port, select an option from the **Target Platform Interfaces** menu as shown in the following figure, and click **Apply**.



Each port is allocated to a specified bit position [b] or range of bit positions [1sb:msb]. The width of the specification, in bits, must equal the width of the port on the DUT. When you select options, the HDL Workflow Advisor automatically allocates a bit range. You can double-click in the **Bit Range / Address / FPGA Pin** column to edit the value.

For detailed information on each **Target Platform Interfaces** option, refer to your board documentation.

Note You must allocate at least one output port must to the target device. If you do not allocate any ports, the **Set Target Interface** task displays an error and terminates.

3 Click **Run This Task**.

4 In the **Set Target Frequency** task, set **FPGA system clock frequency** to 100 MHz, then click **Run This Task**.

In this example, the target frequency must be 100 MHz due to the fixed UART baud rate.

Generate Code, Synthesize, and Program Target Device

After selecting the target device and configuring its port interface, the HDL Workflow Advisor can 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** has two subtasks:
 - **Generate Programming File:** Generating an FPGA programming file.
 - **Program Target Device:** Downloading the programming file to the board.

To run this sequence of tasks automatically:

1 Open the **Download to Target** task group.

2 Right-click **Program Target device** and select **Run to Selected Task**.

The task sequence concludes by programming your target board with the generated programming file. You can then read the code generation and synthesis log files.

Target Xilinx FPGA Development Boards

In this section...

“Before You Begin” on page 22-58

“Open the Model” on page 22-58

“Select the Target Device” on page 22-59

“Set Target Interface and Target Frequency” on page 22-60

“Generate Code, Synthesize, and Program Target Device” on page 22-63

This example shows how to target a Xilinx FPGA development board for synthesis using the FPGA Turnkey workflow.

The `hdlcoderUARTServoControllerExample` model is designed to work with a Xilinx Virtex-5 ML506 development board. The `UART_Servo_on_FPGA` subsystem receives commands through UART ports. The subsystem generates a pulse width modulation (PWM) waveform to control a servo motor.

Before You Begin

To run this example, you must have your synthesis tool set up. To learn how to set up your synthesis tool, see “Synthesis Tool Path Setup”.

This example uses the Xilinx Virtex-5 ML506 development board. You can try this example with a different board, and specify the target interface according to that board’s interface definition. To see a list of boards supported for the FPGA Turnkey workflow, see “FPGA Turnkey Hardware”.

If you want to download the programming file, you must first connect the target device. However, if the target device is not connected, you can still generate the programming file.

Open the Model

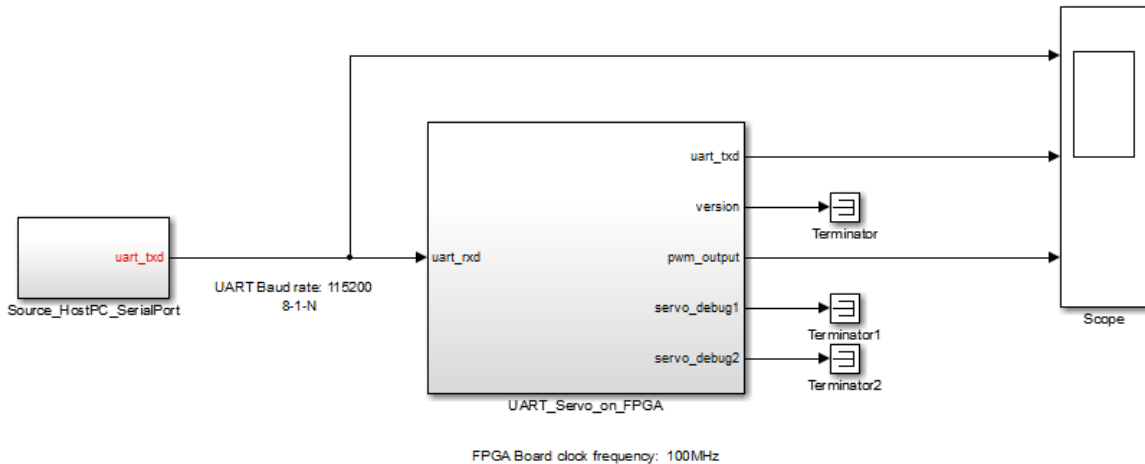
- 1 Add the example directory to your MATLAB path.

```
addpath(fullfile(docroot, 'toolbox', 'hdlcoder', 'examples'))
```

2 Open the model.

hdlcoderUARTServoControllerExample

Example of an FPGA Servo Motor Controller with UART interface

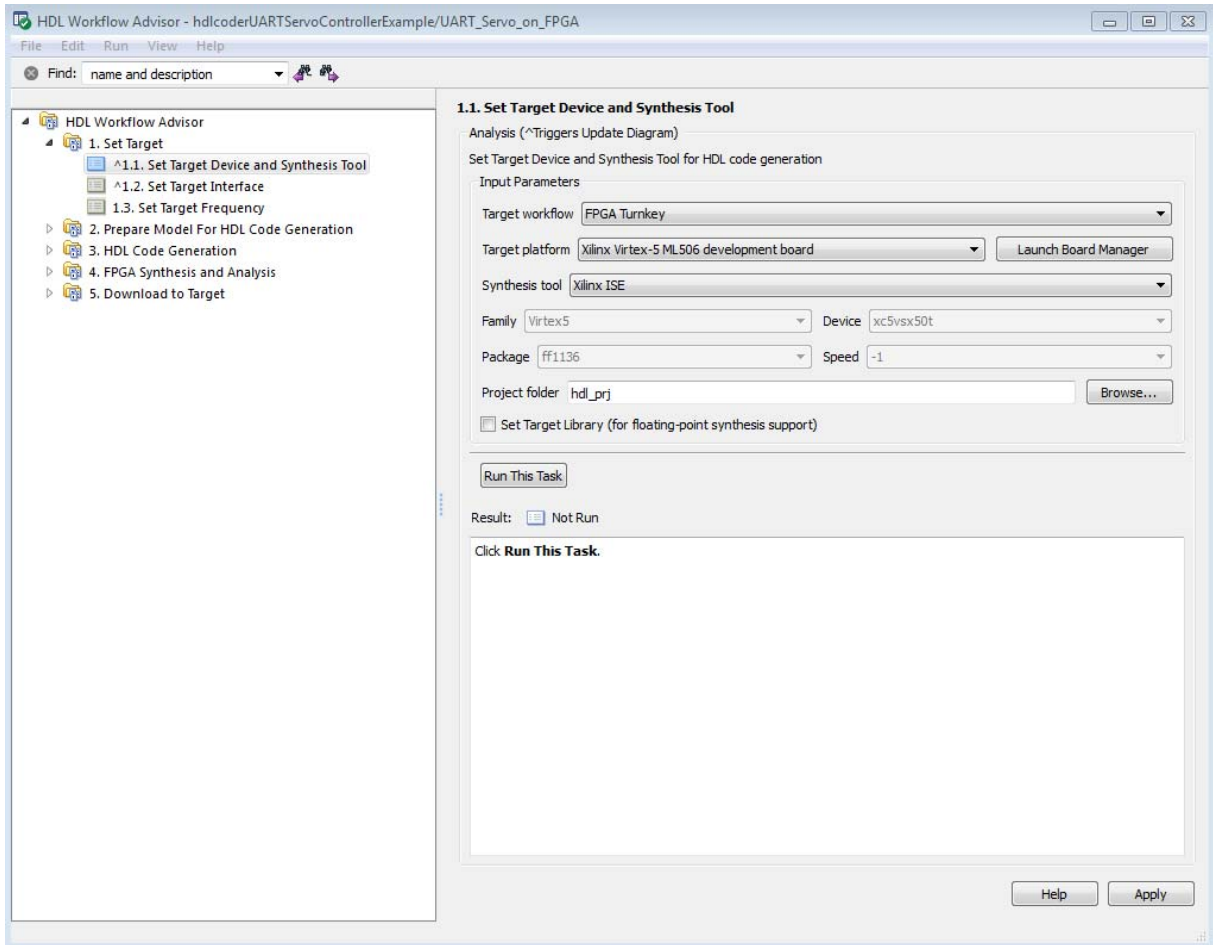


Select the Target Device

- 1** Right-click the `UART_Servo_on_FPGA` subsystem and select **HDL Code > HDL Workflow Advisor**.
- 2** In the HDL Workflow Advisor, select the **Set Target > Set Target Device and Synthesis Tool** task.
- 3** For **Target workflow**, select **FPGA Turnkey**.
- 4** For **Target platform**, select **Xilinx Virtex-5 ML506 development board**.

If the board does not automatically appear in the list, select **Get more boards** to download the Xilinx FPGA Boards support package.

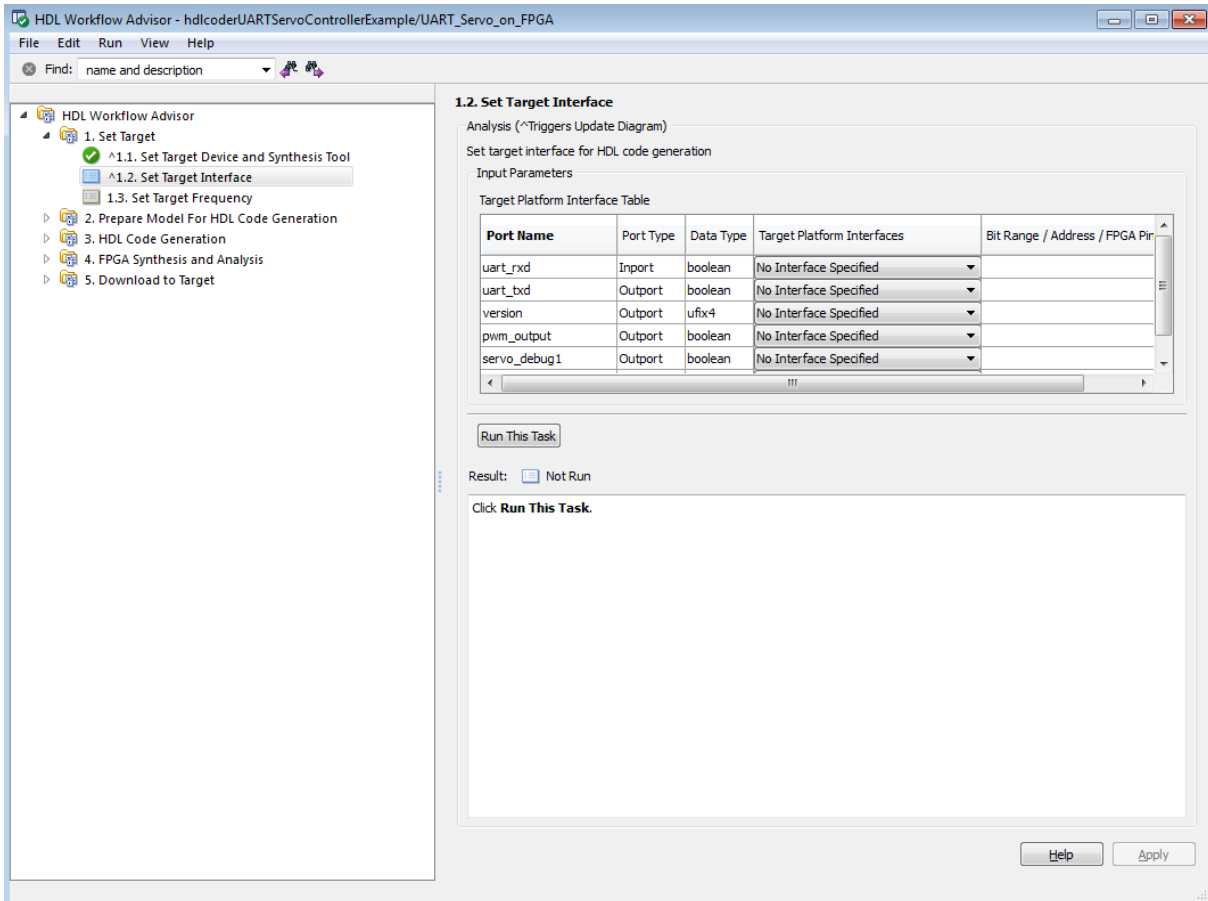
The HDL Workflow Advisor automatically sets the synthesis tool based on your board selection.



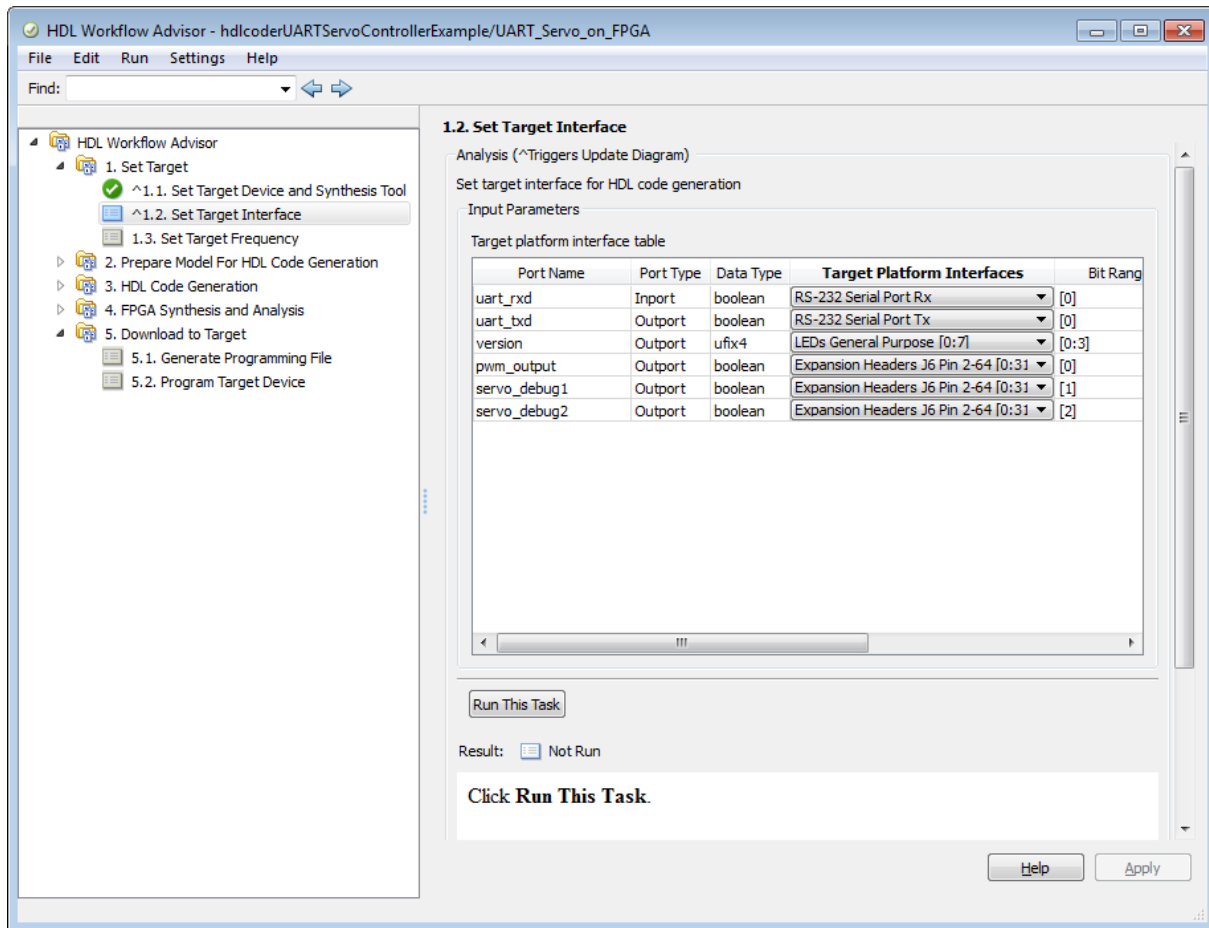
5 Click **Run This Task** .

Set Target Interface and Target Frequency

1 In the left pane of the HDL Workflow Advisor, select the **Set Target Interface** task.



2 For each port, select an option from the **Target Platform Interfaces** menu as shown in the following figure, and click **Apply**.



Each port is allocated to a specified bit position [b] or range of bit positions [lsb:msb]. The width of the specification, in bits, must equal the width of the port on the DUT. When you select options, the HDL Workflow Advisor automatically allocates a bit range. You can double-click in the **Bit Range / Address / FPGA Pin** column to edit the value.

For detailed information on each **Target Platform Interfaces** option, see your Xilinx Virtex-5 ML506 development board documentation.

Note You must allocate at least one output port must to the target device. If you do not allocate any ports, the **Set Target Interface** task displays an error and terminates.

3 Click **Run This Task**.

4 In the **Set Target Frequency** task, set **FPGA system clock frequency** to 100 MHz, then click **Run This Task**.

In this example, the target frequency must be set to 100MHz (the default) due to the fixed UART baud rate.

Generate Code, Synthesize, and Program Target Device

After selecting the target device and configuring its port interface, the HDL Workflow Advisor can 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** has two subtasks:
 - **Generate Programming File:** Generating an FPGA programming file.
 - **Program Target Device:** Downloading the programming file to the board.

Tip Before executing the **Program Target Device** task, make sure that your host PC is properly connected to the Xilinx Virtex-5 ML506 development board via a JTAG programming cable.

To run this sequence of tasks automatically:

- 1 Open the **Download to Target** task group.
- 2 Right-click **Program Target device** and select **Run to Selected Task**.

The task sequence concludes by programming your target board with the generated programming file. You can then read the code generation and synthesis log files.

Custom IP Core Generation

Using the HDL Workflow Advisor, you can generate a custom IP core from a model. The generated IP core is sharable and reusable. You can integrate it with a larger design by adding it in a Xilinx EDK environment.

To learn how to generate a custom IP core from a MATLAB design, see “Generate a Custom IP Core” on page 22-76.

To learn how to generate a custom IP core from Simulink, see “Generate a Custom IP Core” on page 22-76.

In this section...

“Custom IP Core Architectures” on page 22-65

“Target Platform Interfaces” on page 22-66

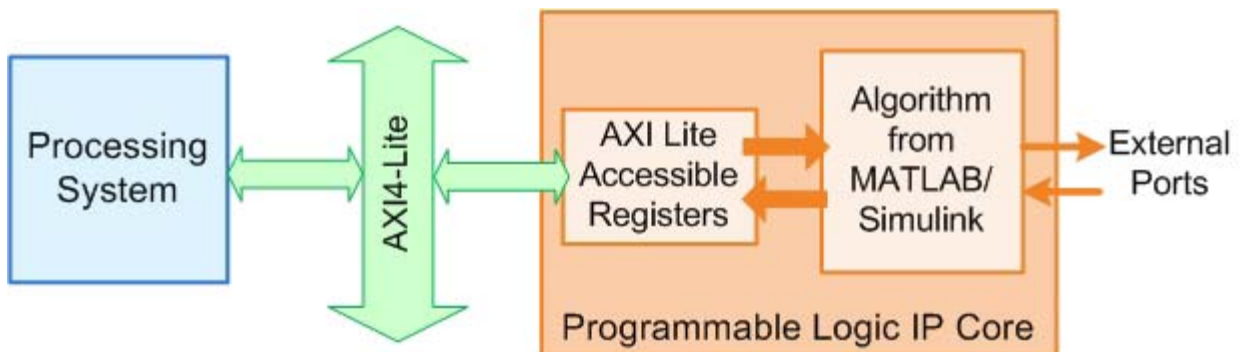
“Processor/FPGA Synchronization” on page 22-67

“Custom IP Core Generated Files” on page 22-67

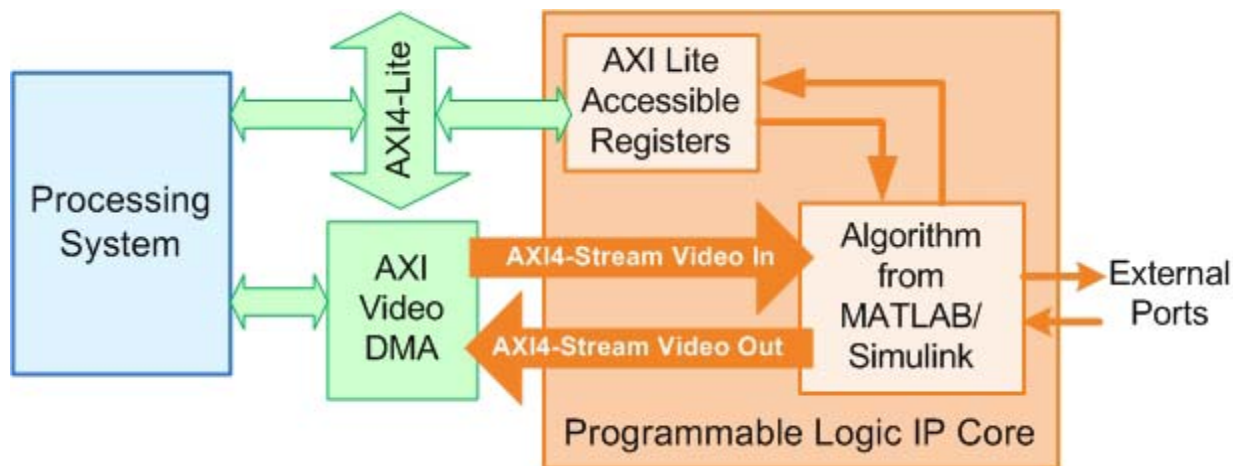
Custom IP Core Architectures

You can generate an IP core with an AXI4-Lite interface, or with both an AXI4-Lite and AXI4-Stream Video interfaces.

An IP core with an AXI4-Lite interface:



An IP core with both an AXI4-Lite interface and AXI4-Stream Video interfaces:



The Algorithm from MATLAB/Simulink block represents your DUT subsystem. The 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 interface for ports you want to access through the AXI4-Lite accessible registers. The coder generates 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-Lite and AXI4-Stream Video protocols, refer to the Xilinx AXI Reference Guide, UG761.

Processor/FPGA Synchronization

The coder generates synchronization logic in the IP core based on the processor/FPGA synchronization mode you choose.

When generating a custom IP core, the following processor/FPGA synchronization options are available:

- Free running (default)
- Coprocessing blocking

To learn more about the processor/FPGA synchronization modes, see “Processor and FPGA Synchronization” on page 22-80.

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. Files for each IP core you generate are in a subfolder, `IPCoreName_IPCoreVersion`, within the `ipcore` folder.

The following files are in the `IPCoreName_IPCoreVersion` folder and its subfolders:

- IP core definition files (.mpd, .pao).
- 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 EDK 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 22-68
“Target Interface Configuration” on page 22-69
“Register Address Mapping” on page 22-70
“IP Core User Guide” on page 22-71
“IP Core File List” on page 22-74

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_blink
IP core version	
IP core folder	hdl_prj\ipcore\hdlcoder_led_blinking_led_c
Target platform	
Target language	
Model	
Model version	
HDL Coder version	
IP core generated on	
IP core generated for	

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
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” on page 22-80.

To learn more about target platform interfaces, see “Custom IP Core Generation” on page 22-65.

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 [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

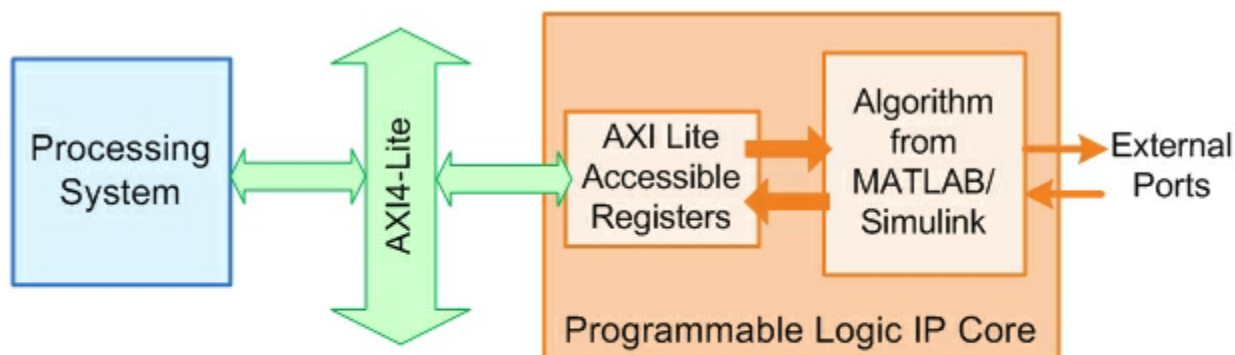
The IP Core User Guide section gives a high-level overview of the system architecture, describes the processor and FPGA synchronization mode, and gives instructions for integrating the IP core into your EDK environment.

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 and the IP core acts as slave. By accessing the generated registers via the AXI4-Lite bus, the processor can 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 0x0 to the IPCore_Enable register. To access the data ports of the MATLAB/Simulink algorithm, read or write data registers.

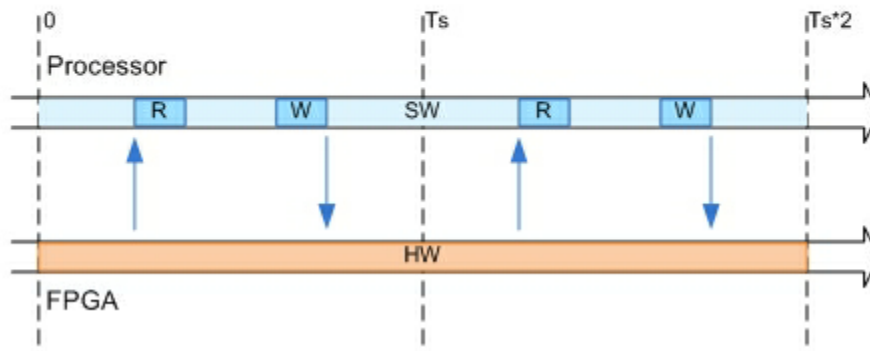


This IP core also support the **External Port** interface. To connect the external ports to the FPGA external assignment constraints in the Xilinx EDK environment.

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 and the IP core (HW). SW and HW runs independently. The data written from the processor to IP core takes effect immediately. 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 EDK environment.

EDK Environment Integration

This IP Core is generated for the Xilinx EDK environment. The following steps are an example showing how to integrate 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 to the XPS project user library.
2. In the XPS project, find the IP core in the user library and add the IP core to the design.
3. Connect the S_AXI port of the IP core to the embedded processor's AXI master port.
4. Connect the clock and reset ports of the IP core to the global clock and reset signals.
5. Assign a base address for the IP core.
6. Connect external ports and add FPGA pin assignment constraints.
7. Generate FPGA bitstream and download the bitstream to target device.

IP Core File List

The IP Core File List section lists the files and file folders that comprise your custom IP core.

The following figure is an example of an IP core file list.

IP Core File List

The IP core folder is located at:

[hdl_prj\ipcore\hdlcoder_led_blinking_led_counter_pcore_v1_00_a](#)

Following files are generated under this folder:

IP core definition files

[data\hdlcoder_led_blinking_led_counter_pcore_v2_1_0.mpd](#)

[data\hdlcoder_led_blinking_led_counter_pcore_v2_1_0.pao](#)

IP core report

[doc\hdlcoder_led_blinking_ip_core_report.html](#)

IP core HDL source files

[hdl\vhdl\led_counter_pkg.vhd](#)

[hdl\vhdl\led_counter.vhd](#)

[hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_dut.vhd](#)

[hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_axi_lite_module.vhd](#)

[hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_addr_decoder.vhd](#)

[hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_axi_lite.vhd](#)

[hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore.vhd](#)

IP core C header file

[include\hdlcoder_led_blinking_led_counter_pcore_addr.h](#)

Hardware and Software Codesign for Xilinx Zynq-7000 Platform

For an example that shows the hardware and software codesign workflow for the Xilinx Zynq-7000 Platform, see `hdlcoder_ip_core_tutorial_zynq`.

The example shows how to:

- Set up your Zynq hardware and tools.
- Generate an HDL IP core for your Simulink design.
- Integrate the IP core into an EDK project and program the Zynq hardware.
- Generate and build the embedded software, and run it on the ARM Cortex®-A9 processor.

Generate a Custom IP Core

In this section...

“Generate a Generic Custom IP Core” on page 22-76

“Generate a Custom IP Core for the Zynq-7000 Platform” on page 22-77

“Requirements and Limitations for Custom IP Core Generation” on page 22-78

Generate a Generic Custom IP Core

To generate a generic custom IP core to use in a Xilinx EDK environment:

- 1 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 Xilinx Platform** and click **Run This Task**.

If you do not see your target hardware in the dropdown menu, select **Get more** to download the target support package.

- 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 interface for ports you want to access through the AXI4-Lite accessible registers. The coder generates 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 In the **Generate RTL Code and IP Core** task, enable the **Generate IP core report** option to generate HTML documentation for the IP core.

- 6 If you want to set options in the other tasks, set them.
- 7 Right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.

The coder generates the IP core files in the specified output folder, including the HTML documentation.

To learn more about custom IP core generation, see “Custom IP Core Generation” on page 22-65.

Generate a Custom IP Core for the Zynq-7000 Platform

To generate a custom IP core to target the Xilinx ZC702 or ZedBoard™:

- 1 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 **Xilinx Zynq ZC702 evaluation kit**, or **Zedboard** and click **Run This Task**.

If you do not see your target hardware in the dropdown menu, select **Get more** to download the target support package.

- 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 interface for ports you want to access through the AXI4-Lite accessible registers. The coder generates 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.

- A board-specific interface, such as **DIP Switches [0:7]**, **Push Buttons L-R-U-D-S [0:4]**, **Pmod Connector JA1 [0:7]**, **Pmod Connector JB1 [0:7]**, **Pmod Connector JC1 [0:7]**, or **Pmod Connector JD1 [0:7]**. Use these external ports to connect to external IO pins on the FPGA board.

In the generated IP core, these ports are generic external ports. In a later step, if you use the HDL Workflow Advisor to integrate the generated IP core with embedded software in an EDK project, the coder connects these ports to the board-specific FPGA pins.

- 5** In the **Generate RTL Code and IP Core** task, enable the **Generate IP core report** option to generate HTML documentation for the IP core.
- 6** If you want to set options in the other tasks, set them.
- 7** Right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.

The coder generates the IP core files in the specified output folder, including the HTML documentation.

To learn more about custom IP core generation, see “Custom IP Core Generation” on page 22-65.

Requirements and Limitations for Custom IP Core Generation

To generate a custom IP core, the DUT subsystem must be an atomic system, and cannot contain Xilinx System Generator blocks.

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 use:

- Generate a custom IP core to use in a Xilinx EDK project.
- Use the **xPC Target FPGA I/O** workflow.

The following synchronization modes are available:

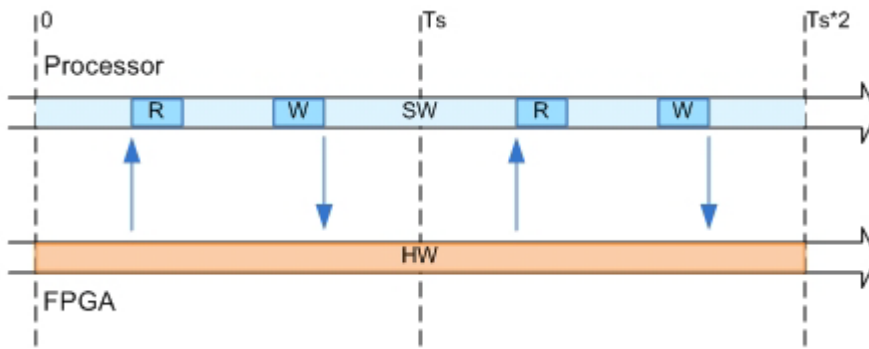
- Free running (default)
- Coprocessing blocking
- Coprocessing nonblocking with delay (available only for the **xPC Target FPGA I/O** workflow)

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.

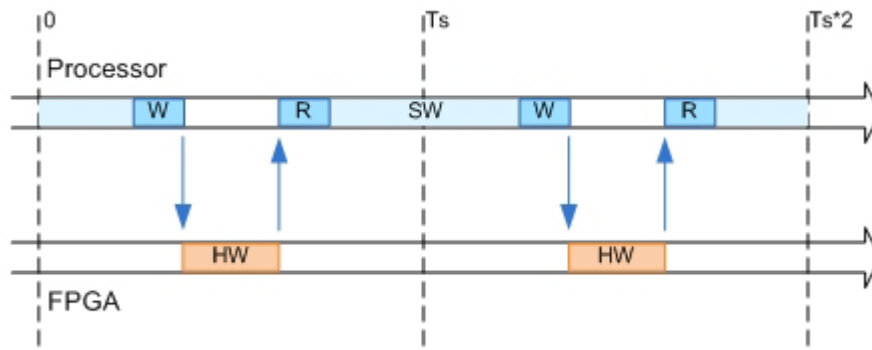


Coprocessing – Blocking Mode

In blocking coprocessor mode, the 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.



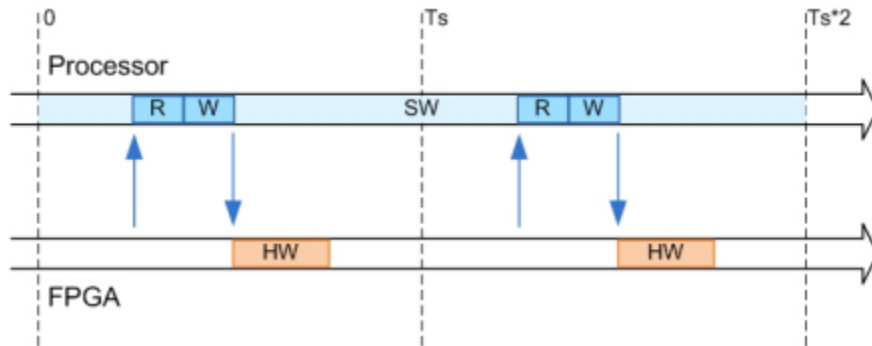
The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor writes to the FPGA, then stops and waits for an indication that the FPGA has finished processing before continuing to run. Each time the FPGA runs, it executes the logic generated for one DUT subsystem sample time.

Coprocessing – Nonblocking With Delay Mode

In delayed nonblocking coprocessor mode, the coder automatically generates synchronization logic for the FPGA so that the processor and FPGA run in tandem. This mode is available only for the **xPC Target 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.

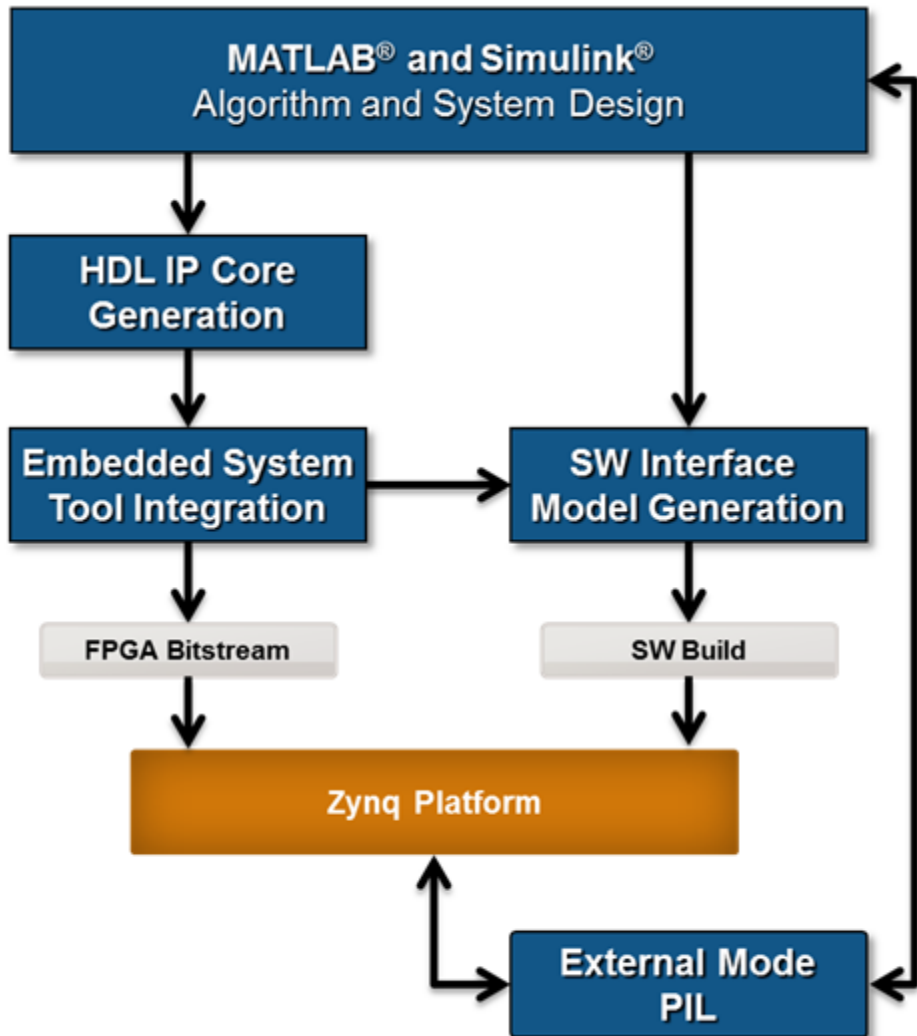
The following diagram shows how the processor and FPGA run in delayed nonblocking coprocessor mode.



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.

Hardware and Software Codesign Workflow

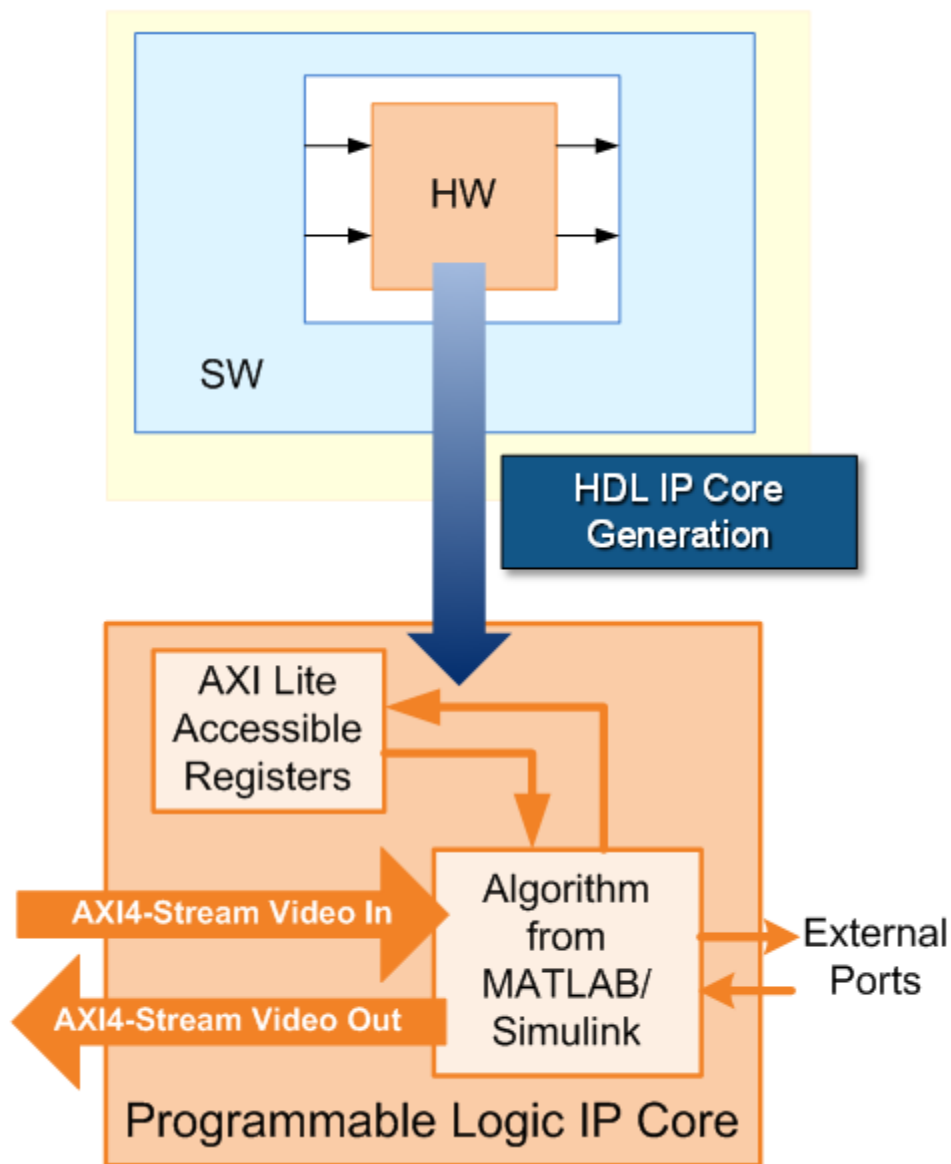
The hardware and software codesign workflow helps automate the deployment of your MATLAB and Simulink design to a Zynq-7000 All Programmable SoC. You can explore the best ways to partition and deploy your design by iterating through the following workflow.



- 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.

- 2** *HDL IP Core Generation*: Enclose the hardware part of your design in an atomic Subsystem block, and use the HDL Workflow Advisor to define and generate an HDL IP core. For more information, see “Custom IP Core Generation” on page 22-65.

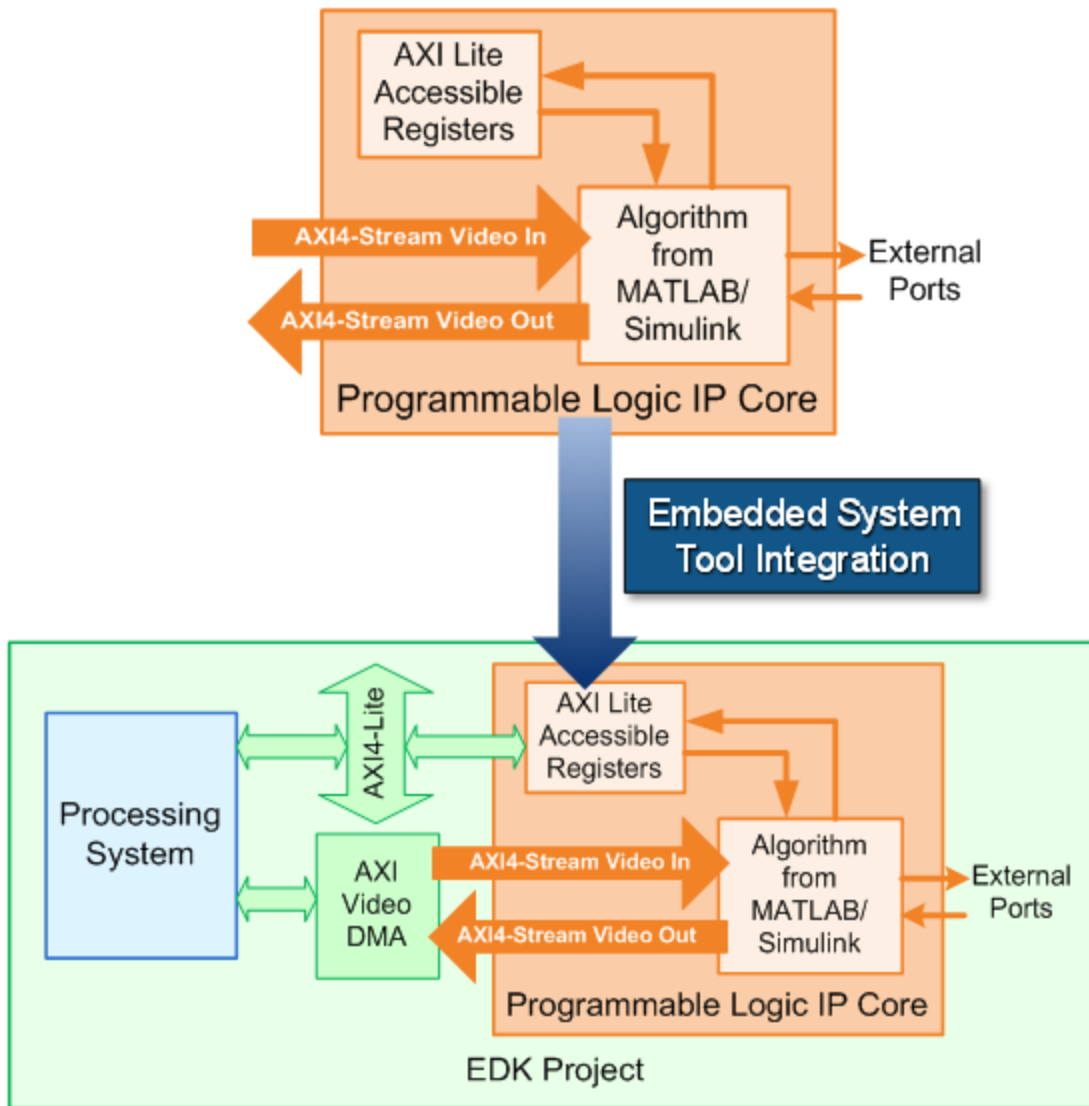
The following diagram shows a model that has been partitioned into a hardware part, in orange, and software part, in blue. HDL IP core generation creates an IP core from the hardware part of the model. The IP core hardware interface includes components such as AXI4-Lite interface-accessible registers, 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 Zynq hardware.

The *reference design* is a predefined EDK project. It contains all the elements the Xilinx software needs to deploy your design to the Zynq platform, except for the custom IP core and embedded software that you generate.

The following diagram shows the relationship between the reference design, in green, and the generated IP core, in orange.

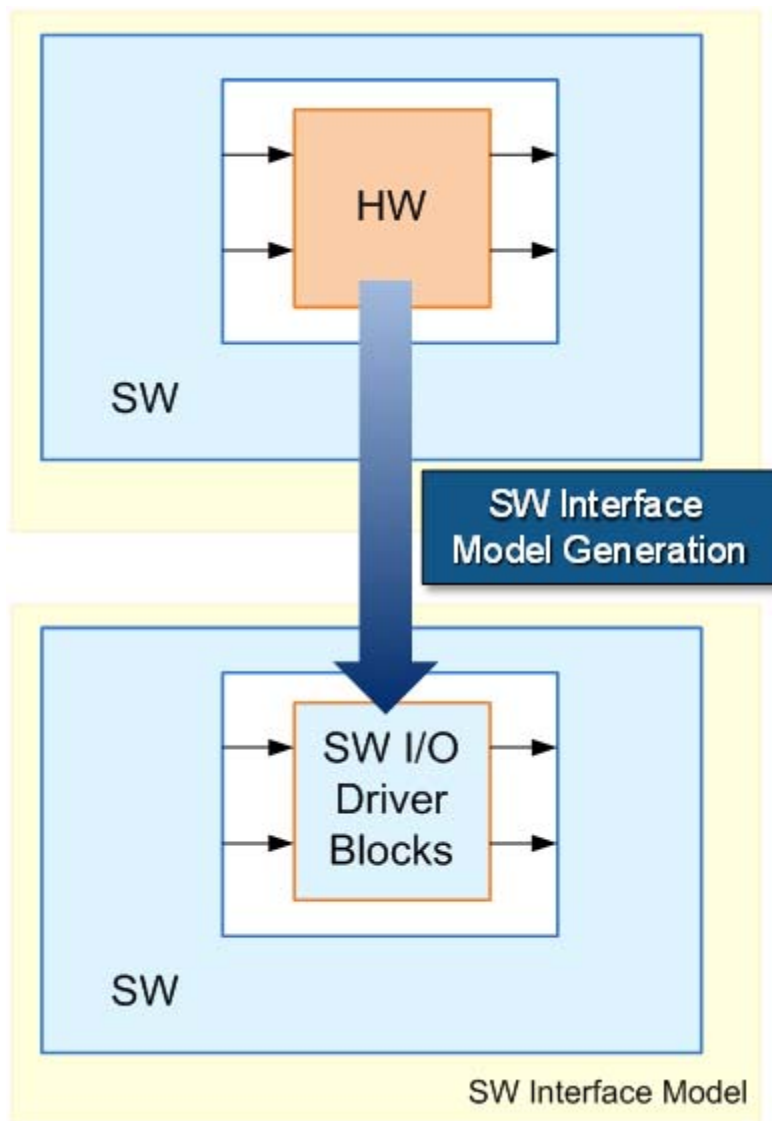


- 4 *SW Interface Model Generation* (requires a Simulink license): In the HDL Workflow Advisor, after you generate the IP core and insert it into the reference design, you can optionally generate a software interface model.

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 embedded code from the software interface model, build it, 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 *Zynq Platform and External Mode PIL*: Using the HDL Workflow Advisor, you program your FPGA bitstream to the Zynq 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.

Install Support for Altera FPGA Boards

You can use the HDL Coder FPGA Turnkey workflow with Altera FPGA Boards by installing the related support package.

To install the **HDL Coder Support Package for Altera FPGA Boards** from the HDL Workflow Advisor:

- 1** In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **FPGA Turnkey**.
- 2** For **Target platform**, select **Get more boards** to open the Support Package Installer.
- 3** In the Support Package Installer, select **Altera FPGA Boards** and follow the instructions provided by Support Package Installer to complete the installation.

Install Support for Xilinx FPGA Boards

You can use the HDL Coder FPGA Turnkey workflow with Xilinx FPGA Boards by installing the related support package.

To install the **HDL Coder Support Package for Xilinx FPGA Boards** from the HDL Workflow Advisor:

- 1** In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **FPGA Turnkey**.
- 2** For **Target platform**, select **Get more boards** to open the Support Package Installer.
- 3** In the Support Package Installer, select **Xilinx FPGA Boards** and follow the instructions provided by Support Package Installer to complete the installation.

Install Support for Xilinx Zynq-7000 Platform

You can use the HDL Coder IP core generation workflow with the Xilinx Zynq-7000 Platform by installing the related support package.

To install the **HDL Coder Support Package for Xilinx Zynq-7000 Platform** from the HDL Workflow Advisor:

- 1** In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.
- 2** For **Target platform**, select **Get more** to open the Support Package Installer.
- 3** In the Support Package Installer, select **Xilinx Zynq-7000** and follow the instructions provided by Support Package Installer to complete the installation.

HDL Test Bench

Generate Test Bench With File I/O

In this section...

“When to Use File I/O In Test Bench” on page 23-2

“How Test Bench Generation with File I/O Works” on page 23-2

“Test Bench Data Files” on page 23-3

“How to Generate Test Bench with File I/O” on page 23-3

“Limitations When Using File I/O In Test Bench” on page 23-4

When to Use File I/O In Test Bench

By default, the 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, the 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, the 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')
```

```
### 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 24-2
- “Create Custom FPGA Board Definition” on page 24-7
- “Create Xilinx KC705 Evaluation Board Definition File” on page 24-8
- “FPGA Board Manager” on page 24-22
- “New FPGA Board Wizard” on page 24-26
- “FPGA Board Editor” on page 24-37

FPGA Board Customization

In this section...
“Feature Description” on page 24-2
“Custom Board Management” on page 24-2
“FPGA Board Requirements” on page 24-3

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. All you need to add a board is 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 to import a custom board, remove a board, make a copy of a board for further modification, and verify a new board.

Custom Board Management

You manage FPGA custom boards through the following user interfaces:

- “FPGA Board Manager” on page 24-22: portal to adding, importing, deleting, and otherwise managing board definition files.
- “New FPGA Board Wizard” on page 24-26: This wizard guides you through creating a custom board definition file with information you obtain from the board specification documentation.
- “FPGA Board Editor” on page 24-37: user interface for viewing or editing board information.

To begin, review the “FPGA Board Requirements” on page 24-3 and then follow the steps described in “Create Custom FPGA Board Definition” on page 24-7.

FPGA Board Requirements

- “FPGA Device” on page 24-3
- “FPGA Design Software” on page 24-3
- “General Hardware Requirements” on page 24-3
- “Hardware Requirements for FPGA-in-the-Loop” on page 24-4

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 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 a 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 “Hardware Requirements for FPGA-in-the-Loop” on page 24-4).
- **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 Programmer.

Hardware Requirements for FPGA-in-the-Loop

An Ethernet connection between the FPGA board and its host PC is required for FIL. 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 and it can be automatically recognized by the FPGA programming software. If this is not the case, use FPGA programming software to program your FPGA with the correct options.

Supported Ethernet PHY Device. 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, 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.
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/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 Clock Frequency Requirement for GMII/RGMII Interface.

When GMII/RGMII interfaces are used, an exact 125MHz clock is required by FPGA 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.

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. This means you use the MDIO module to configure

Marvell 88E1111 to add internal delays. (See “FIL I/O” on page 24-31 for the usage of the MDIO module.)

Create Custom FPGA Board Definition

- 1 Be ready with the following:
 - a Board specification document. Any format you are comfortable with is fine, but 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 function `hdlsetuptoolpath` to configure the tool for use with MATLAB. For example:

```
>> hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', 'C:\Xilinx\13.4\ISE_DS\ISE\bin\nt64\ise.exe
```

- 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 24-22.
- 4 The wizard guides you through entering all board information. At each page, fill in the required fields. For assistance in entering board information, see “New FPGA Board Wizard” on page 24-26.
- 5 Save the board definition file. This is the last step and is automatically instigated when you click **Finish** in the New FPGA Board Wizard. See “Save Board Definition File” on page 24-17.

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 24-8 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 24-8

“What You Need to Know Before Starting” on page 24-8

“Start New FPGA Board Wizard” on page 24-9

“Provide Basic Board Information” on page 24-10

“Specify FPGA Interface Information” on page 24-12

“Enter FPGA Pin Numbers” on page 24-13

“Run Optional Validation Tests” on page 24-15

“Save Board Definition File” on page 24-17

“Use New FPGA Board” on page 24-18

Overview

For FPGA-in-the-Loop, you can use your own qualified FPGA board even if 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

- You need to know the following types of information about the board:
 - FPGA interface to the Ethernet PHY chip
 - Clock pins names and numbers
 - Reset pins names and numbers

In this example, the above information is supplied to you in this section. 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 function `hdlsetuptoolpath` to configure the tool for use with MATLAB. For example:

```
>> hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', 'C:\Xilinx\13.4\ISE_DS\ISE\bin\nt64\ise.exe');
```

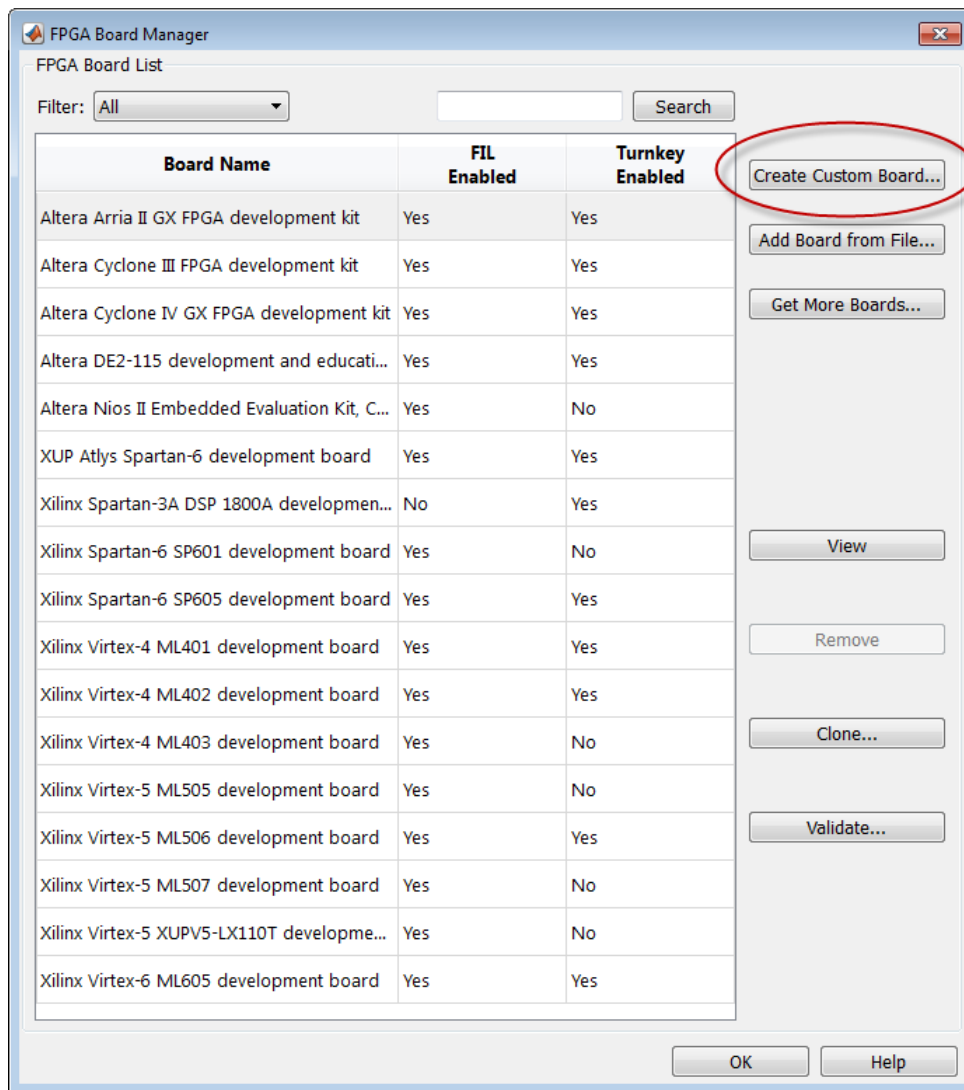
- If you want to verify programming the FPGA board after you add its definition file, you will need to have the custom board attached to your computer. However, having the board connected is not necessary for creating the board definition file.

Start New FPGA Board Wizard

- 1 Start the FPGA Board Manager by entering the following command at the MATLAB prompt:

```
>>fpgaBoardManager
```

- 2 Click **Create New Board** to open the New FPGA Board Wizard.

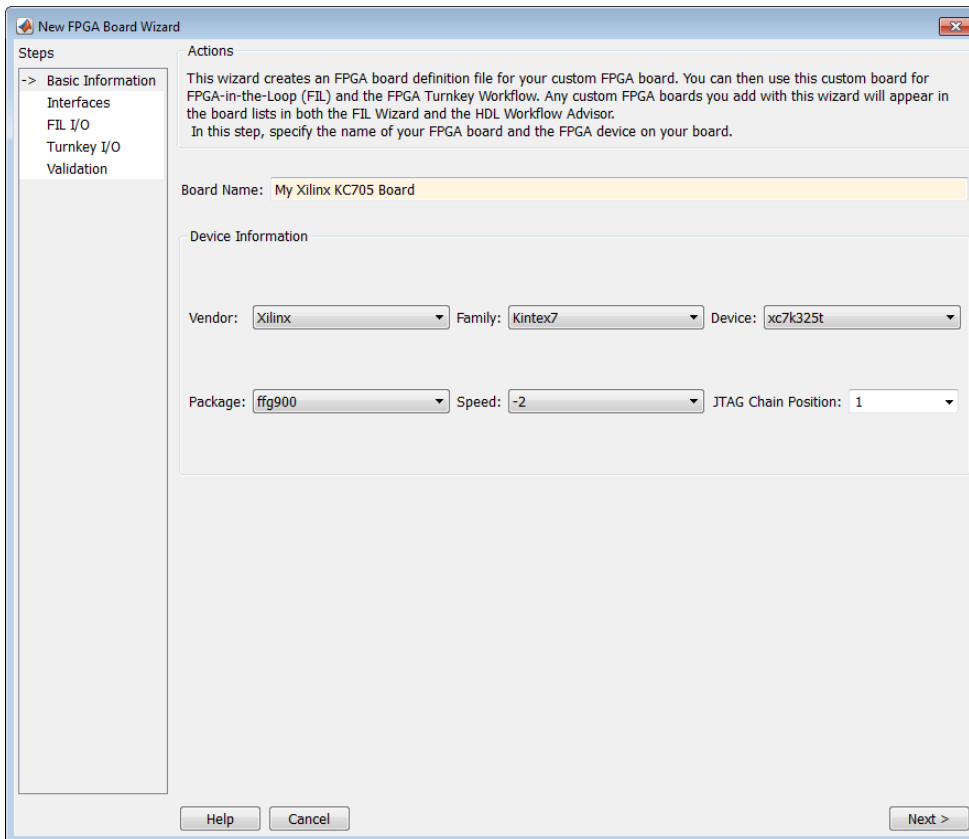


Provide Basic Board Information

- 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 wizard should now look like the following image.



The information you just entered can be found in the KC705 Evaluation Board for the Kintex-7 FPGA User Guide.

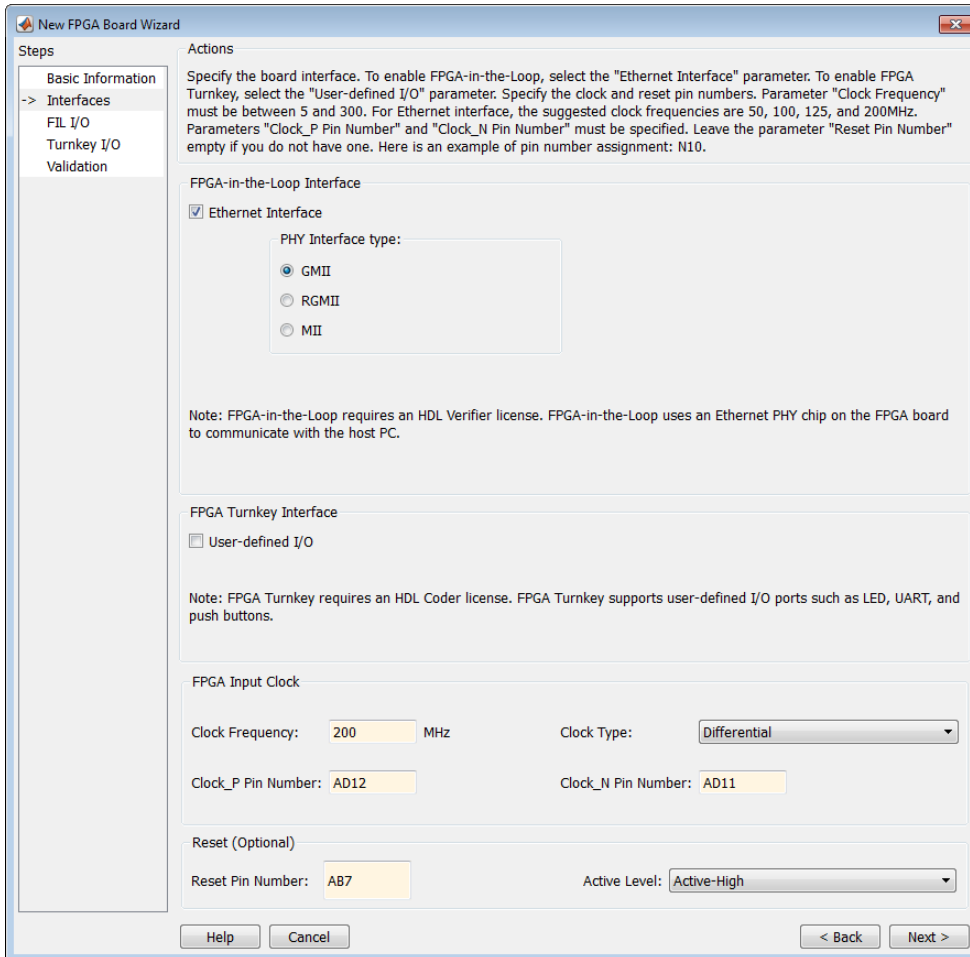
2 Click **Next**.

Specify FPGA Interface Information

- 1** In the Interfaces pane, perform the following tasks.
 - a** Check the **Ethernet Interface** option in the FPGA-in-the-Loop Interface section. 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 unchecked. FPGA Turnkey workflow is not the focus of this example.
 - d** **Clock Frequency:** Enter 200. Note that this Xilinx KC705 board has multiple clock sources. This 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** **Resent Pin Number:** Enter AB7. This will supply a global reset to the FPGA.
 - i** **Active Level:** Select **Active-High**.

You can obtain all necessary information from the board design specification.

The wizard should now look like the following image.



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.

Note that 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

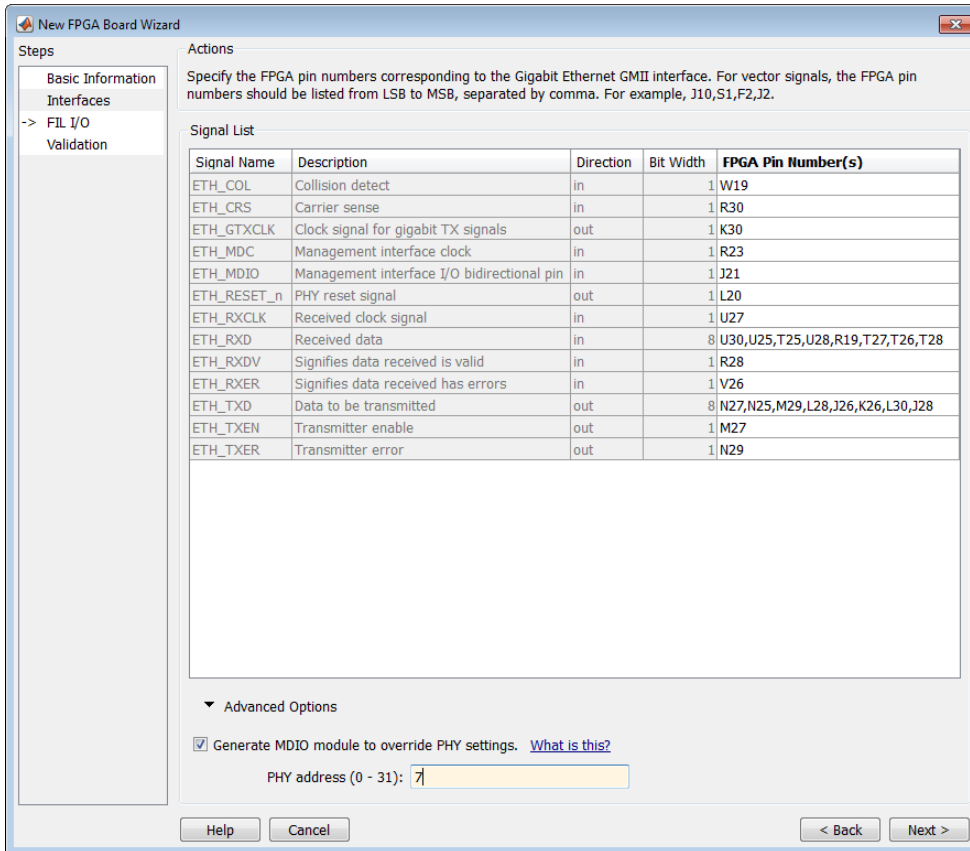
- 2 Click Advanced Options to expand the section.
- 3 Check the **Generate MDIO module to override PHY settings** option.

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 will not work if the PHY device 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.

The wizard should now look like the following image.



5 Click Next.

Run Optional Validation Tests

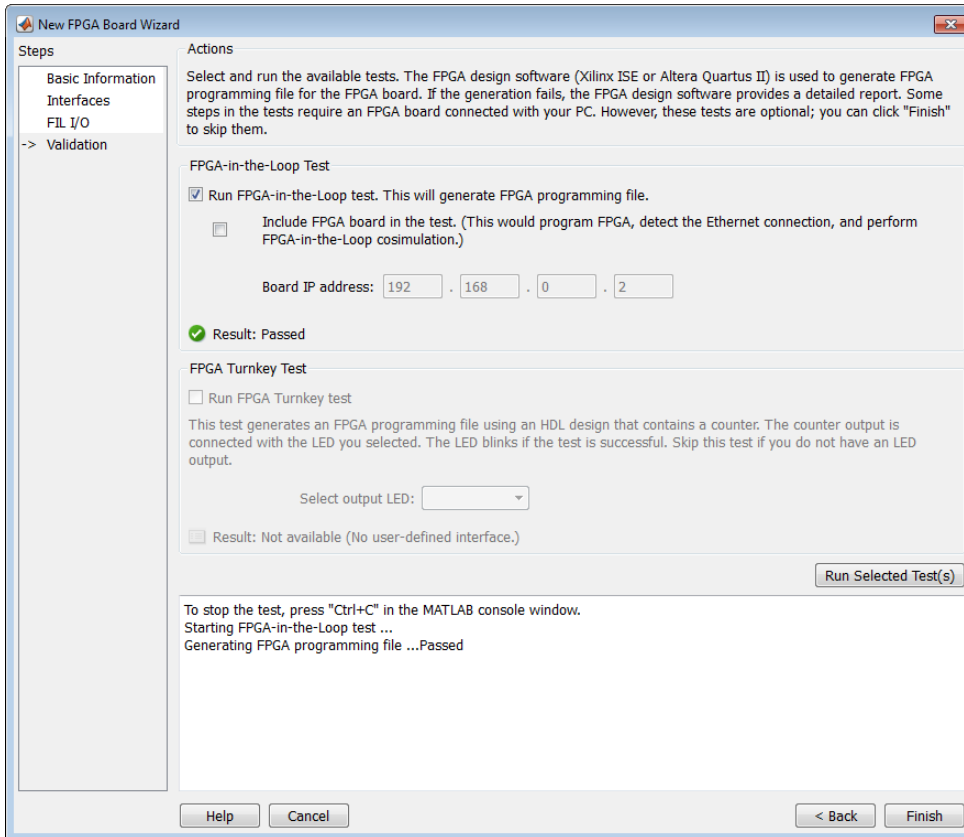
This step provides a validation test for you to verify if the entered information is correct by performing FPGA-in-the-Loop cosimulation. You will need Xilinx ISE 13.4 or higher versions installed on the same computer. This step is optional and you may skip it if you prefer.

Note For validation, you must have Xilinx or Altera on your path. Use the function `hdlsetuptoolpath` to configure the tool for use with MATLAB. For example:

```
>> hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', 'C:\Xilinx\13.4\ISE_DS\ISE\bin\nt64\ise.exe');
```

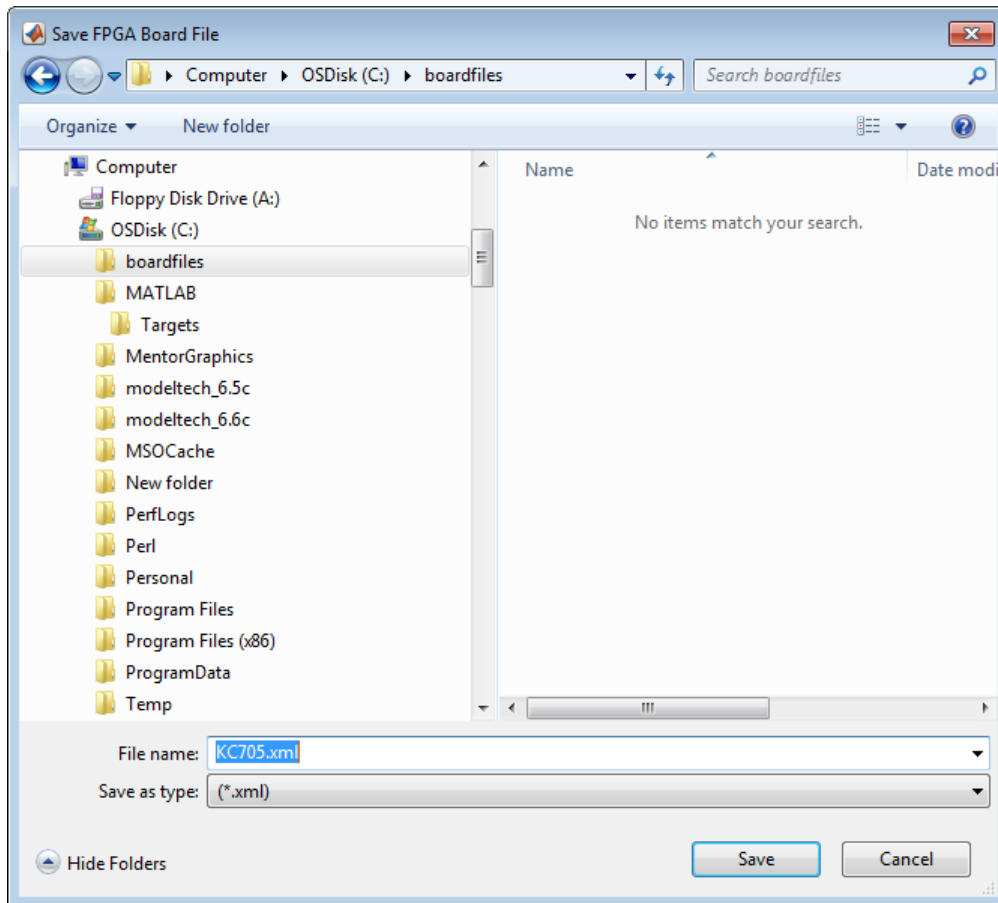
To run this test, perform the following actions.

- 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 will need to supply the IP address of the FPGA Board. This example assumes 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 will take about 10 minutes to complete.



Save Board Definition File

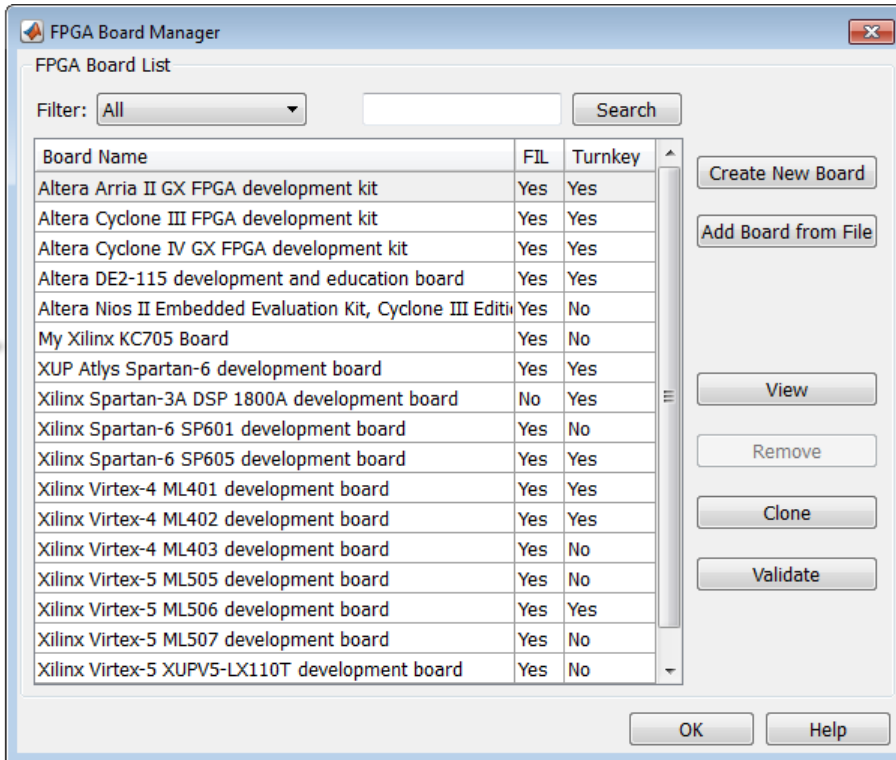
- 1 Click **Finish** to exit the New FPGA Board Wizard. A **Save As** dialog pops up and asks for the location of the FPGA board definition file. For this example, save as `C:\boardfiles\KC705.xml`.



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 just defined.

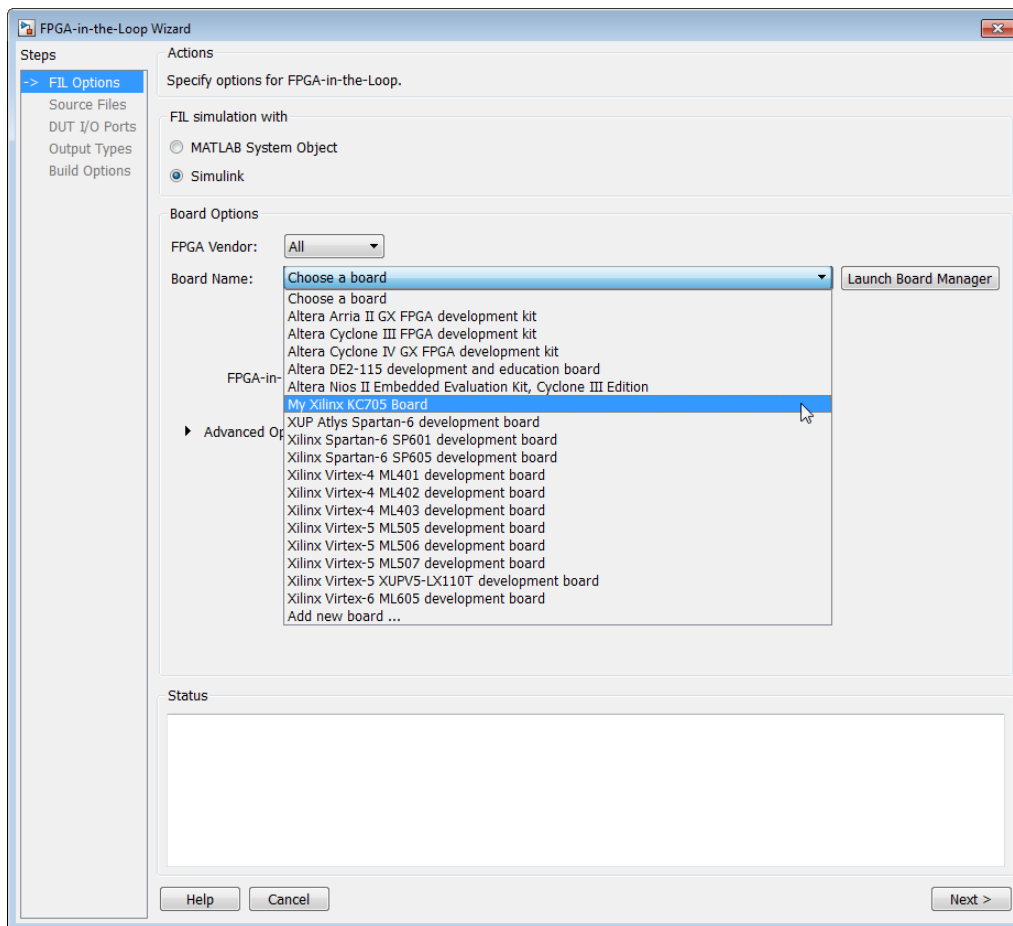


Click **OK** to close the FPGA Board Manager.

- 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
```

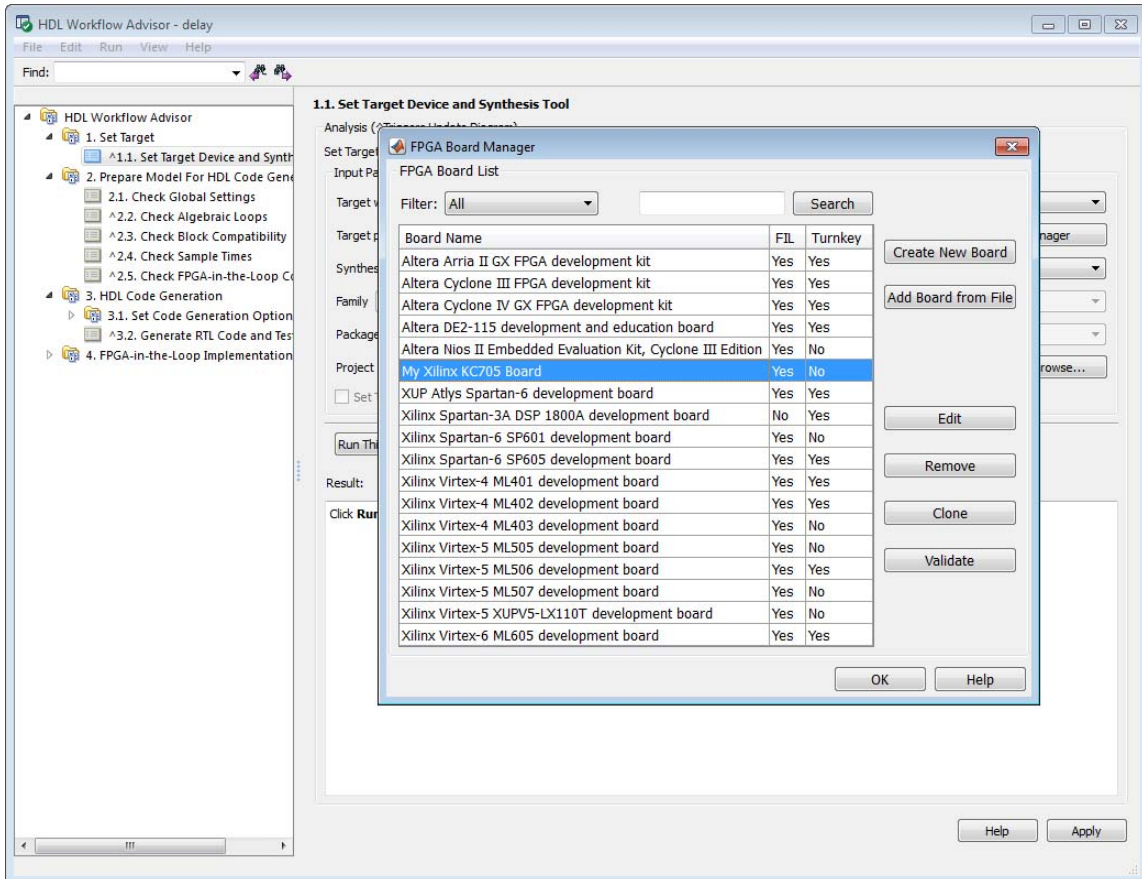
The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-Loop simulation.



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.



This concludes the example of adding a custom board definition file.

FPGA Board Manager

In this section...

“Introduction” on page 24-22

“Filter” on page 24-24

“Search” on page 24-24

“FIL Enabled/Turnkey Enabled” on page 24-24

“Create Custom Board” on page 24-24

“Add Board From File” on page 24-24

“Get More Boards” on page 24-24

“View/Edit” on page 24-25

“Remove” on page 24-25

“Clone” on page 24-25

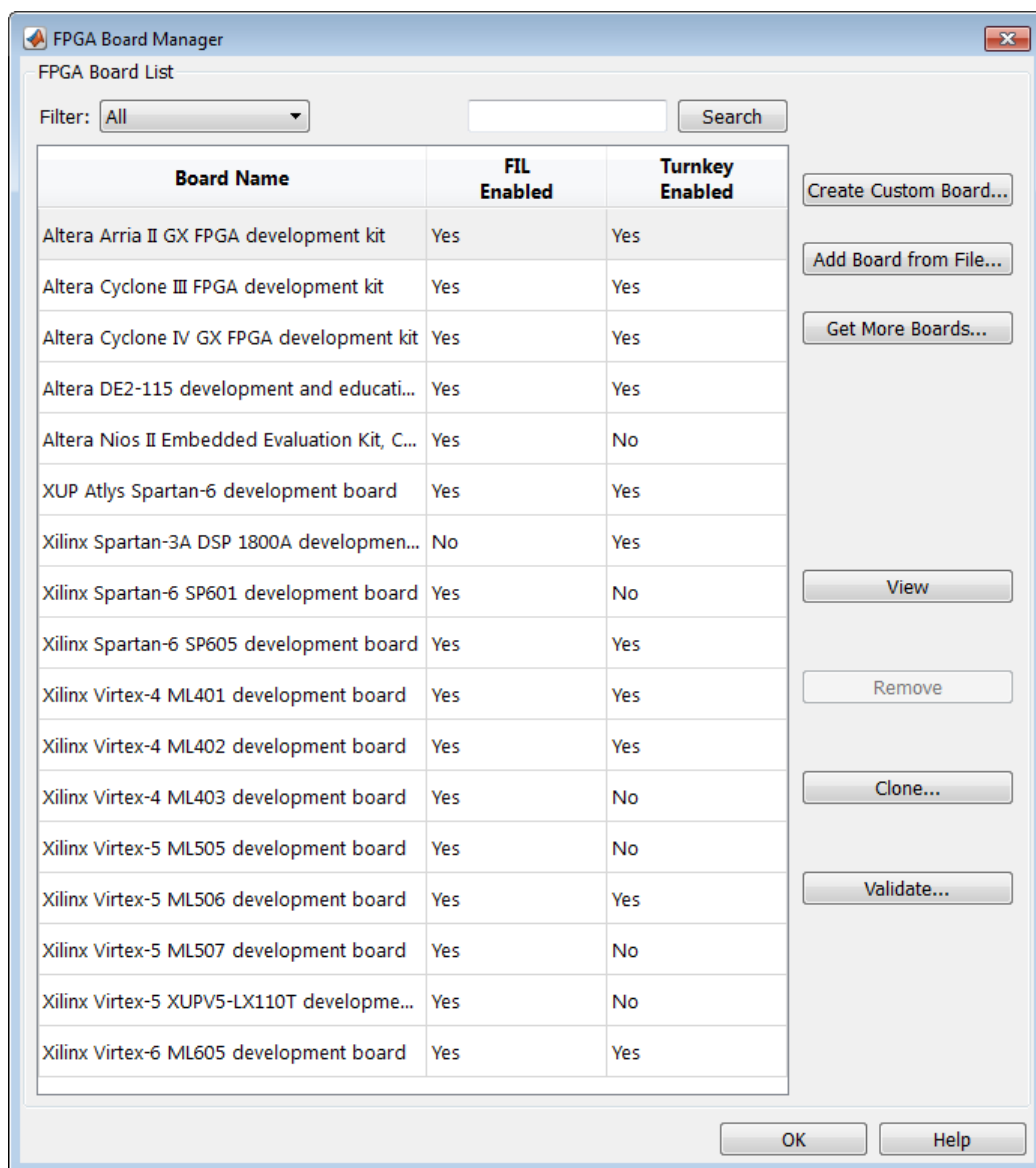
“Validate” on page 24-25

Introduction

The FPGA Board Manager is the portal to managing custom FPGA boards. You can create a new 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 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 24-26. You can find the process for creating a new board definition file in “Create Custom FPGA Board Definition” on page 24-7.

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 may view a read-only file but not edit it. See “FPGA Board Editor” on page 24-37.

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 24-15.

New FPGA Board Wizard

Using the New FPGA Board Wizard, you can enter all the required information needed to add a board to the FPGA board list. This list applies to both FIL and Turnkey workflows. Review “FPGA Board Requirements” on page 24-3 before adding a new FPGA board to make sure it is compatible with the workflow for which you want to use it.

Several buttons in the New FPGA Board Wizard assist in 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 have the option to exit with or without saving the information from your session.

Adding Boards Once for Multiple Users To add new boards globally, follow these instructions. Note that 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 directory will 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 a new FPGA board contains these steps:

In this section...

“Basic Information” on page 24-28

“Interfaces” on page 24-29

“FIL I/O” on page 24-31

“Turnkey I/O” on page 24-33

“Validation” on page 24-36

“Finish” on page 24-36

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 (selected), Interfaces, FIL I/O, Turnkey I/O, and Validation. The 'Actions' pane contains the following text: 'This wizard creates an FPGA board description file for your custom FPGA board. You can then use this custom board for FPGA-in-the-Loop (FIL) and the FPGA Turnkey Workflow. Any custom FPGA boards you add with this wizard will appear in the board lists in both the FIL Wizard and the HDL Workflow Advisor. In this step, specify the name of your FPGA board and the FPGA device on your board.'

The 'Board Name' field contains the text 'HDLV Custom Board 1'. The 'Device Information' section contains the following fields:

- Vendor: Xilinx
- Family: Spartan6
- Device: xc6sxc150t
- Package: fgg676
- Speed: -3
- JTAG Chain Position: 1

At the bottom right, there are four buttons: 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

Specify the board interface. To enable FPGA-in-the-Loop, select the "Ethernet 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

Ethernet Interface

PHY Interface type:

GMII

RGMII

MI

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: AD11

Reset (Optional)

Reset Pin Number: AB7 Active Level: Active-High

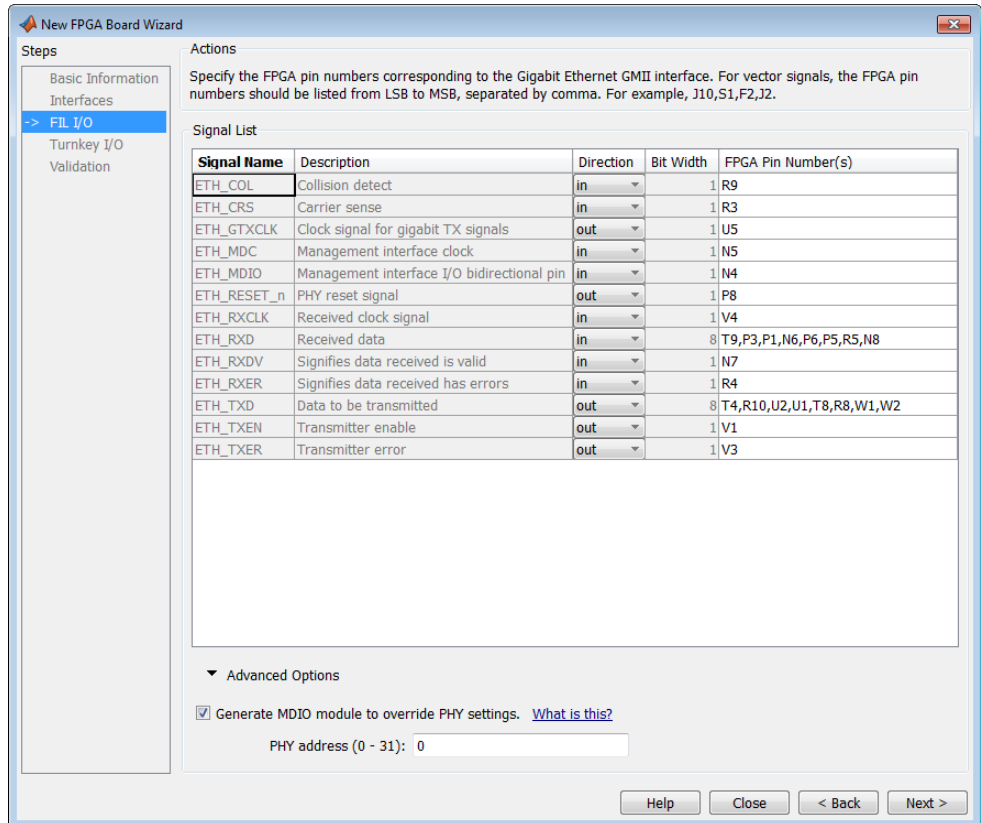
Help Cancel < Back Next >

- **FIL Interface:** If you want to use this board with FIL, check **Ethernet Interface**. Specify the **PHY Interface type** (found in the board specification file).

Note Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

- **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.** Clock details are required for both workflows. You can find all necessary information in the board specification file.
 - **Clock Frequency.** Must be between 5 and 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
 - **Clock Pin Number** . Must be specified. Example: N10.
 - **Clock Type** : Single_Ended or Differential.
- **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.

FIL I/O



Signal List: You must 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 (MDIO)?

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 checkbox if you are not using Marvell 88E1111.

The generated MDIO module is used to perform the following operations:

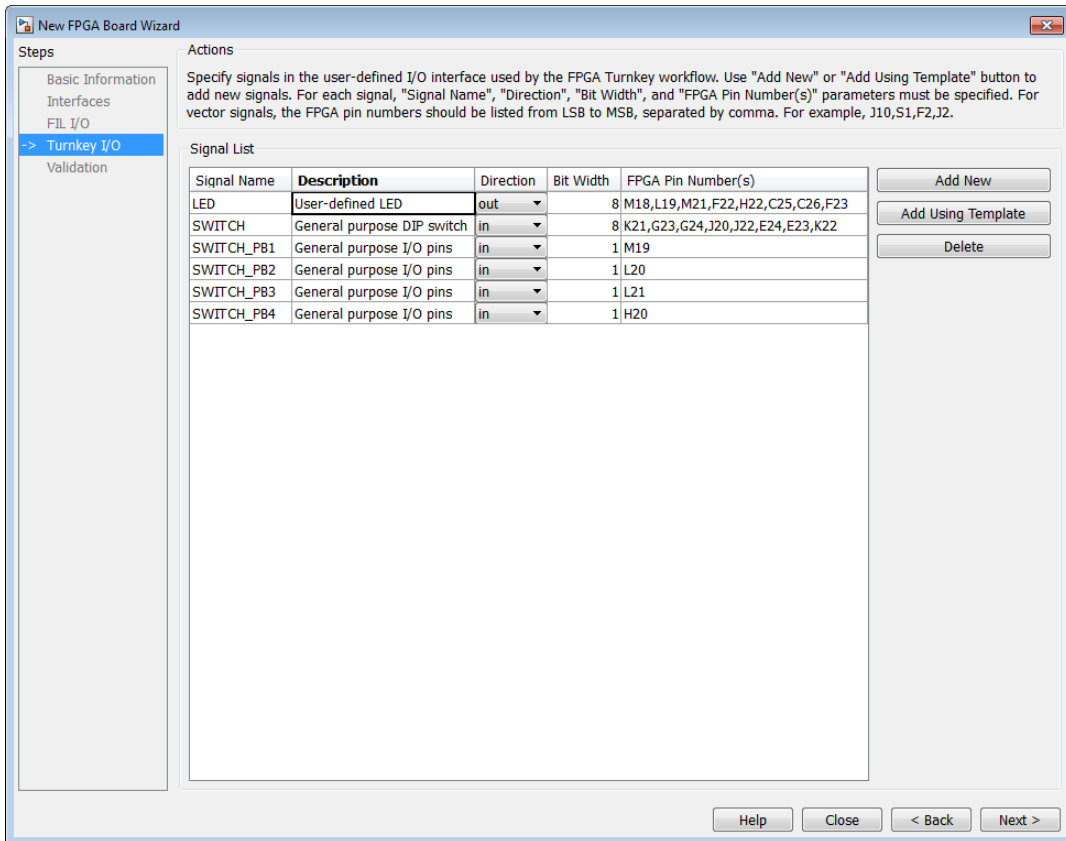
- **GMII mode:** The PHY device can start up using other modes, such as RGMII/SGMII. The generated MDIO module sets the PHY chip in GMII mode.
- **RGMII mode:** The PHY device can start up using other modes, such as GMII/SGMII. The generated MDIO module sets the PHY device in RGMII mode. In addition, the module sets the PHY chip to add internal delay for RX and TX clocks.
- **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



You must define at least one output port for the Turnkey I/O interface.

Signal List: You must 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:

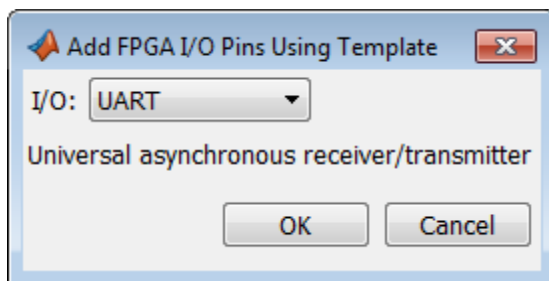
- A generic signal name
- Description
- Direction
- Bit width

You may change the values in any of these prepopulated fields.

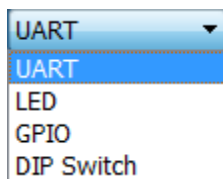
Delete: Delete the selected signal from list.

The following example demonstrates using the **Add Using Template** feature.

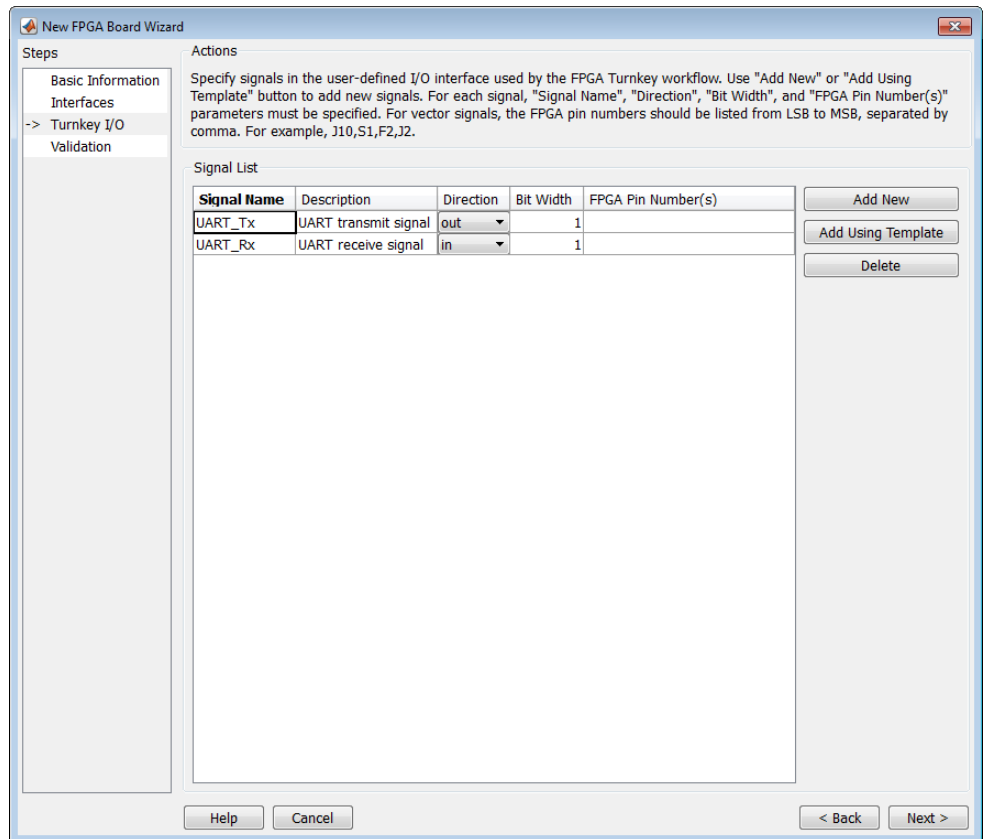
- 1 In the Turnkey I/O dialog, click **Add Using Template**.
- 2 You can now view the template dialog.



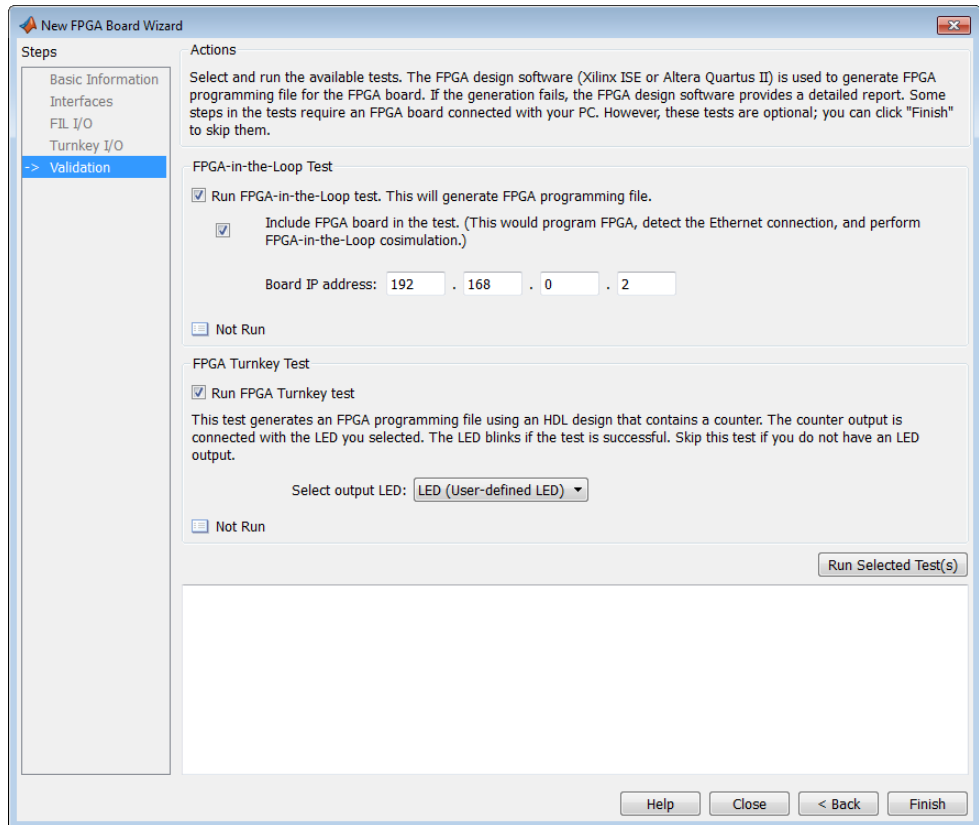
- 3 Pull down the I/O list and select from the following options:



- 4 Click **OK**.
- 5 The wizard adds the specified signal (or signals) to the I/O list.



Validation



Run the validation test. For FPGA Turnkey testing, you must have a board attached. For FIL testing, the board is optional.

Finish

When you have completed validation, click **Finish**. See “Save Board Definition File” on page 24-17.

FPGA Board Editor

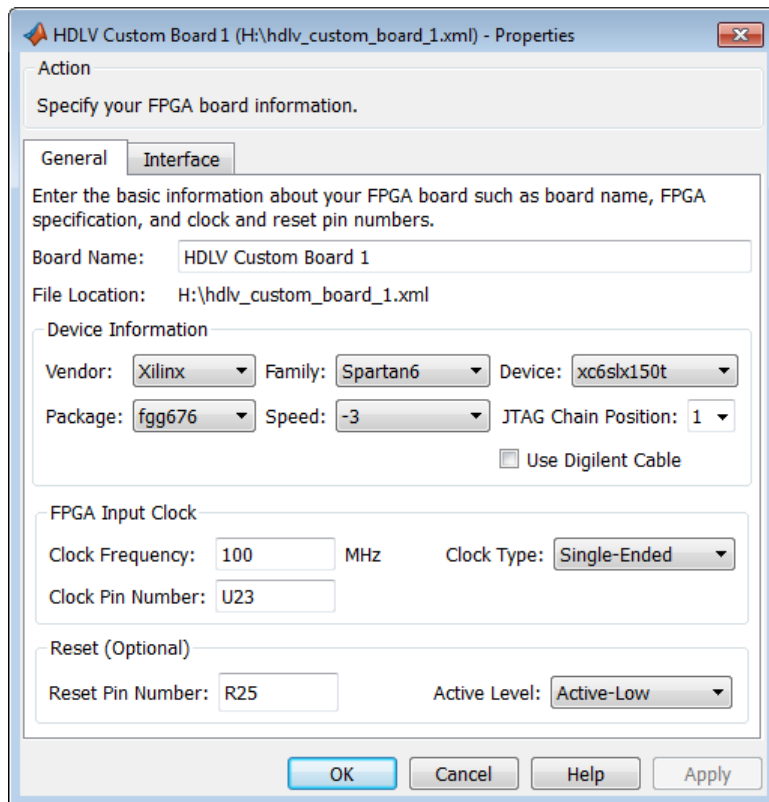
To edit a board definition XML file, you must first make it writeable. If the file is read-only, the FPGA Board Editor only lets you view the board configuration information. You cannot modify that information.

In this section...

“General” on page 24-37

“Interface” on page 24-39

General



The screenshot shows the 'HDLV Custom Board 1 (H:\hdlv_custom_board_1.xml) - Properties' dialog box. The 'General' tab is selected. The dialog contains the following fields and options:

- Action:** Specify your FPGA board information.
- General / Interface:** Two tabs, with 'General' selected.
- Enter the basic information about your FPGA board such as board name, FPGA specification, and clock and reset pin numbers.**
- Board Name:** HDLV Custom Board 1
- File Location:** H:\hdlv_custom_board_1.xml
- Device Information:**
 - Vendor:** Xilinx
 - Family:** Spartan6
 - Device:** xc6slx150t
 - Package:** fgg676
 - Speed:** -3
 - JTAG Chain Position:** 1
 - Use Digilent Cable
- FPGA Input Clock:**
 - Clock Frequency:** 100 MHz
 - Clock Type:** Single-Ended
 - Clock Pin Number:** U23
- Reset (Optional):**
 - Reset Pin Number:** R25
 - Active Level:** Active-Low

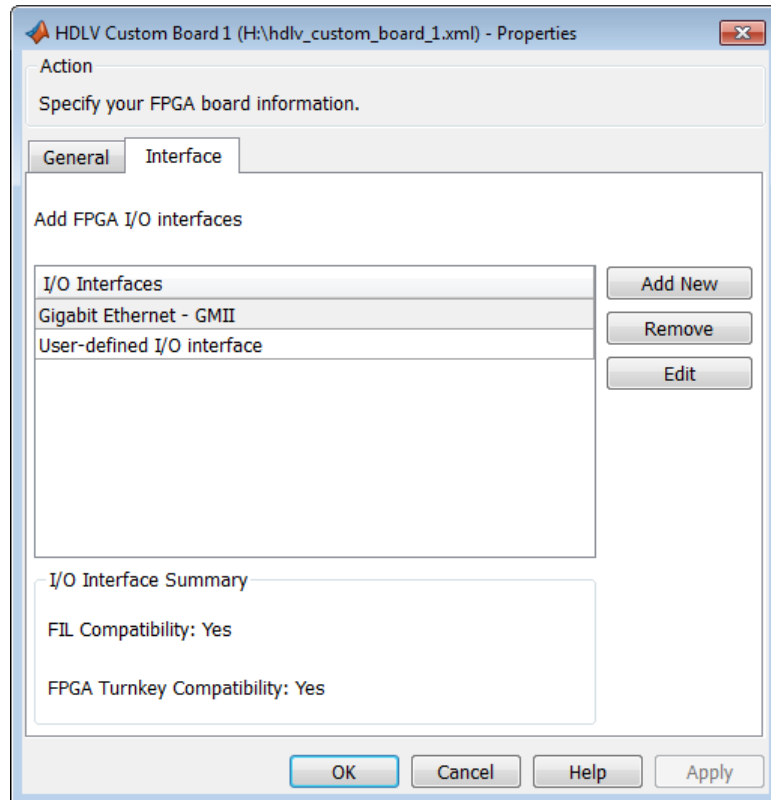
Buttons at the bottom: OK, Cancel, Help, Apply.

Board Name: Unique board name

Device Information:

- **Vendor:** Xilinx or Altera
- **Family:** Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device:** Device depends on the specified vendor and family. See the board specification file for applicable settings.
- For Xilinx boards only:
 - **Package:** Package depends on specified vendor, family, and device. See the board specification file for applicable settings.
 - **Speed:** Speed depends on package. See the board specification file for applicable settings.
 - **JTAG Chain Position:** Value indicates the starting position for JTAG chain. Consult the board specification file for this information.
- **FPGA Input Clock.** Clock details are required for both the FIL and Turnkey workflows. You can find all necessary information in the board specification file.
 - **Clock Frequency.** Must be between 5 and 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
 - **Clock Pin Number** . Must be specified. Example: N10.
 - **Clock Type** : Single_Ended or Differential.
- **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.

Interface



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 or **Remove** an interface.

HDL Workflow Advisor Tasks

HDL Workflow Advisor Tasks

In this section...

- “HDL Workflow Advisor Tasks Overview” on page 25-3
- “Set Target Overview” on page 25-6
- “Set Target Device and Synthesis Tool” on page 25-7
- “Set Target Library” on page 25-9
- “Set Target Interface” on page 25-10
- “Set Target Frequency” on page 25-11
- “Set Target Interface” on page 25-12
- “Prepare Model For HDL Code Generation Overview” on page 25-13
- “Check Global Settings” on page 25-14
- “Check Algebraic Loops” on page 25-15
- “Check Block Compatibility” on page 25-16
- “Check Sample Times” on page 25-17
- “Check FPGA-in-the-Loop Compatibility” on page 25-18
- “HDL Code Generation Overview” on page 25-19
- “Set Code Generation Options Overview” on page 25-20
- “Set Basic Options” on page 25-21
- “Set Advanced Options” on page 25-22
- “Set Testbench Options” on page 25-23
- “Generate RTL Code and Testbench” on page 25-24
- “Generate RTL Code and IP Core” on page 25-26
- “FPGA Synthesis and Analysis Overview” on page 25-27
- “Create Project” on page 25-28
- “Perform Synthesis and P/R Overview” on page 25-29
- “Perform Logic Synthesis” on page 25-30
- “Perform Mapping” on page 25-31

In this section...

“Perform Place and Route” on page 25-32

“Annotate Model with Synthesis Result” on page 25-33

“Download to Target Overview” on page 25-35

“Generate Programming File” on page 25-36

“Program Target Device” on page 25-37

“Generate xPC Target Interface” on page 25-38

“Save and Restore HDL Workflow Advisor State” on page 25-39

“FPGA-in-the-Loop Implementation” on page 25-39

“Set FIL Options” on page 25-39

“Build FPGA-in-the-Loop” on page 25-39

“Check USRP® Compatibility” on page 25-40

“Verify with HDL Cosimulation” on page 25-40

“Generate FPGA Implementation” on page 25-40

“Check SDR Compatibility” on page 25-40

“SDR FPGA Implementation” on page 25-41

“Set SDR Options” on page 25-41

“Build SDR” on page 25-43

“Embedded System Integration” on page 25-44

“Create Project” on page 25-44

“Generate Software Interface Model” on page 25-44

“Build FPGA Bitstream” on page 25-45

“Program Target Device” on page 25-45

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 an xPC Target 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
 - xPC Target FPGA I/O
 - IP Core Generation
 - Customization for an SDR Platform
- **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.
- **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.

Dependencies

Setting **Target workflow** to FPGA Turnkey or xPC Target FPGA I/O enables the following tasks:

- “Set Target Interface” on page 25-10

- “Set Target Frequency” on page 25-11
- Tasks in the **Download to Target** folder

Setting **Target workflow** to IP Core Generation enables the “Set Target Interface” on page 25-12 task.

Selecting **Set Target Library (for floating-point synthesis support)** causes the **Set Target Library** task to appear on the left.

See Also

For information on the Set Target Library task, see “Set Target Library” on page 25-9.

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 the 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, the coder issues a warning.

Absolute path: Enter the location of the simulation library. This option is available if **Set Xilinx simulation path** is selected.

Dependencies

This task appears when you select **Set Target Library (for floating-point synthesis support)** in the **Set Target Device and Synthesis Tool** task.

The **Set Xilinx simulation path** option is available when you select a Xilinx device in the **Set Target Device and Synthesis Tool** task.

See Also

For more information on targeting FPGA floating-point library blocks, see “Map to an FPGA Floating-Point Library” on page 22-22.

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.

Dependency

This task appears when you set **Target workflow** to FPGA Turnkey or xPC Target FPGA I/O.

Set Target Frequency

Automatically generate clock module for FPGA Turnkey or xPC Target FPGA I/O targets.

Leave entry unchanged if you wish to use the default value (same as input).

Dependency

This task appears when you set **Target workflow** to FPGA Turnkey or xPC Target FPGA I/O.

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 the 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 the 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.

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.

Dependency

This task appears when you set **Target workflow** to IP Core Generation, FPGA Turnkey, or xPC Target FPGA I/O.

See Also

- “Processor and FPGA Synchronization” on page 22-80

- “Custom IP Core Generation” on page 22-65
- “Generate xPC Target Interface for Speedgoat Boards” on page 22-42

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 coder 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 the coder does not support. The HDL Workflow Advisor reports incompatible blocks, including the full path to each block.

See Also

See “Blocks Supported for HDL Code Generation” on page 11-3 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 coder 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 to Failure**.

Tip After each task in this folder runs, the 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” on page 9-8 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” on page 9-22 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” on page 9-78 for information on each parameter.

To run the tasks in the **Set Code Generation Options** folder automatically, select the folder and click **Run to Failure**.

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” on page 9-8.

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 9-22.

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” on page 9-78.

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, the 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, the 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.
- **IP core version:** Enter the IP core version number.

The coder appends the version number to the IP core name to generate the output folder name.

- **IP core folder** (not editable): Shows the output folder name.
- **Generate IP core report:** Select this option to generate HTML documentation for the IP core.

Dependency

This task appears when you set **Target workflow** to IP Core Generation.

See Also

- “Custom IP Core Generation” on page 22-65
- “Generate a Custom IP Core” on page 22-76
- “Custom IP Core Report” on page 22-68

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” on page 22-28.

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” on page 22-28.

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:

- **Perform Logic Synthesis:** Launch supported FPGA synthesis tool and synthesize the generated HDL code.
- **Perform Mapping:** Launch supported FPGA synthesis tool and perform mapping and timing analysis.
- **Perform Place and Route:** Launch supported FPGA synthesis tool and 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” on page 22-28

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” on page 22-30.

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” on page 22-30.

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” on page 22-30.

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 Place and Route** task, 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 **Perform Mapping** task.

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, the coder displays the DUT with critical path information highlighted.

See Also

“Annotating Your Model with Critical Path Information” on page 22-35

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 xPC Target Interface** (for Speedgoat target devices only):
Generate a model that contains an xPC Target 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 xPC Target Interface

The **Generate xPC Target Interface** task generates a model containing an interface subsystem that you can plug in to an xPC Target 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” on page 22-10.

FPGA-in-the-Loop Implementation

Set FIL options and run FIL processing.

Set FIL Options

Set board IP and MAC addresses and select additional files, if required.

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 FCOMMS1 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

- **User logic data path**

Select either the Receiver data path or the Transmitter data path.

- If you select Receiver data path, you can optionally choose to **Include receiver decimation filter**. When you select this option, the SDR FPGA component includes the configurable decimation filters after the ADC. The default value of this checkbox is selected.
- If you select Transmitter data path, you can optionally choose to **Include transmitter interpolation filter**. The default value of this checkbox is selected.

- **User logic data path**

Select either the Receiver data path or the Transmitter data path.

- If you selected Receiver data path, you can optionally choose to **Include receiver decimation filter**. The default value of this checkbox is selected.
- If you selected Transmitter data path, you can optionally choose to **Include transmitter interpolation filter**. The default value of this checkbox is selected.

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 project. You should include only file types supported by ISE. If an included file does not exist, the HDL Workflow Advisor cannot create the ISE 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 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.

Embedded System Tool Input Parameter	Description
Embedded system tool	Embedded design tool.
Reference design	Predefined EDK project into which the coder inserts your generated IP core files.
Reference design path	If you are using the downloaded Xilinx Targeted Reference Design (TRD) instead of the default Reference Design , enter the path to the downloaded TRD.
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.

Add IP core device driver to Linux kernel: Select to insert the IP core node into the Linux device tree on the SD card on your Zynq board, reboot Linux, and add the IP core driver as a Linux 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.

Code Generation Control Files

- “READ THIS FIRST: Control File Compatibility and Conversion Issues” on page 26-2
- “Overview of Control Files” on page 26-4
- “Structure of a Control File” on page 26-7
- “Code Generation Control Objects and Methods” on page 26-8
- “Using Control Files in the Code Generation Process” on page 26-16
- “Specifying Block Implementations and Parameters in the Control File” on page 26-17
- “Generating Black Box Control Statements Using `hdlnewblackbox`” on page 26-23

READ THIS FIRST: Control File Compatibility and Conversion Issues

In this section...

“Conversion From Use of Control Files Recommended” on page 26-2

“Detaching Existing Models From Control Files” on page 26-2

“Applying Control File Settings” on page 26-3

“Backwards Compatibility” on page 26-3

Conversion From Use of Control Files Recommended

As of release R2010b, the coder does not support the attachment of a control file to a new model. Instead, the coder now saves nondefault HDL-related model settings, block implementation selections and implementation parameter settings to the model itself. This eliminates the need to maintain a separate control file. Because the coder saves only the nondefault parameter settings, the loading and saving of models is more efficient. The recommended practice is to discontinue use of control files and convert existing models. This simple process is described in the next section.

Detaching Existing Models From Control Files

If you have existing models with attached control files, you should convert them to the current format and remove control file linkage. To convert a model that has an attached control file:

- 1 Open the model. When the coder opens a model that has an attached control file, it loads and sets parameters as specified in the control file, and clears the control file linkage from the model. During this process, the coder displays the following messages:

```
Found HDL control file attached to the model 'test_model' ...
Loading control file 'test_model_control' ...
Successfully loaded control file 'test_model_control.m' ...
Please consider saving the model to make changes permanent ...
Detaching the HDL control file from the model...
```

- 2 Save the model. The model now preserves nondefault settings. The next time you open the model, the coder will not display control file status messages.

Note that although the model is now detached from the control file, the control file itself is preserved so that you can apply it to other models if you wish.

Applying Control File Settings

The coder provides the `hdlapplycontrolfile` utility as a quick way to transfer HDL settings from existing models that have attached control files to other models. See `hdlapplycontrolfile` for further information.

Backwards Compatibility

For backward compatibility, the coder continues to support models that have attached control files.

Overview of Control Files

In this section...

“What Is a Control File?” on page 26-4

“Selectable Block Implementations and Implementation Parameters” on page 26-5

“Implementation Mappings” on page 26-5

What Is a Control File?

Code generation control files (referred to in this document as *control files*) let you

- Save your model’s HDL code generation options in a persistent form.
- Extend the HDL code generation process and direct its details.

You attach a control file to your model using either the `makehdl` command or the Configuration Parameters dialog box. You do not need to know internal details of the code generation process to use a control file.

Control files support the following statement types:

- *Selection/action* statements provide a general framework for the application of different types of transformations to selected model components. Selection/action statements *select* a group of blocks within your model, and specify an *action* to be executed when code is generated for each block in the selected group.

Selection criteria include block type and location within the model. For example, you might select all built-in Gain blocks at or below the level of a certain subsystem within your model.

A typical action applied to such a group of blocks is to direct the code generator to execute a specific *block implementation method* when generating HDL code for the selected blocks. For example, for Gain blocks, you might choose a method that generates code that is optimized for speed or chip area.

- *Property setting* statements let you

- Select the model or subsystem from which code is to be generated.
- Set the values of code generation properties to be passed to the code generator. The properties and syntax are the same as those used for the `makehdl` command.
- Set up default or template HDL code generation settings for your organization.

Selectable Block Implementations and Implementation Parameters

Selection/action statements provide a general framework that lets you define how the coder acts upon selected model components. The current release supports one such action: execution of block implementation methods.

Block implementation methods are code generator components that emit HDL code for the blocks in a model. This document refers to block implementation methods as *block implementations* or simply *implementations*.

The coder provides at least one block implementation for every supported block . This is called the *default implementation*. In addition, the coder provides selectable alternate block implementations for certain block types. Each implementation is optimized for different characteristics, such as speed or chip area. For example, you can choose Gain block implementations that use canonic signed digit (CSD) techniques (reducing area), or use a default implementation that retains multipliers.

For many block implementations, you can set *implementation parameters* that provide a further level of control over how code is generated for a particular implementation. For example, most blocks support the 'OutputPipeline' implementation parameter. This parameter lets you specify the generation of output pipeline stages for selected blocks by passing in the required pipeline depth as the parameter value.

Implementation Mappings

Control files let you specify one or more *implementation mappings* that control how HDL code is to be generated for a specified group of blocks within the model. An implementation mapping is an association between a selected block or set of blocks within the model and a block implementation.

To select the set of blocks to be mapped to a block implementation, you specify

- A `modelscope`: a Simulink block path (which could incorporate an entire model or sublevel of the model, or a specific subsystem or block)
- A `blocktype`: a Simulink block type that corresponds to the selected block implementation

During code generation, each defined `modelscope` is searched for instances of the associated `blocktype`. For each such block instance encountered, the code generator uses the selected block implementation.

Structure of a Control File

The required elements for a code generation control file are as follows:

- A control file implements a single function, which is invoked during the code generation process.

The function must instantiate a *code generation control object*, set its properties, and return the object to the code generator.

Setting up a code generation control object requires the use of a small number of methods, as described in “Code Generation Control Objects and Methods” on page 26-8. You do not need to know internal details of the code generation control object or the class to which it belongs.

You construct the object using the `hdlnewcontrol` function. The argument to `hdlnewcontrol` is the name of the control file itself. Use the `mfilename` function to pass in the file name, as shown in the following example.

```
function c = dct8config
c = hdlnewcontrol(mfilename);

% Set target language for Verilog.
c.set('TargetLanguage','Verilog');

% Set top-level subsystem from which code is generated.
c.generateHDLFor('dct8_fixed/OneD_DCT8');
```

- Following the constructor call, your code will invoke methods of the code generation control object. The previous example calls the `set` and `generateHDLFor` methods. These and other public methods of the object are discussed in “Code Generation Control Objects and Methods” on page 26-8.
- Your control file must be attached to your model before code generation, as described in “Using Control Files in the Code Generation Process” on page 26-16. The interface between the code generator and your attached control file is automatic.
- A control file must be located in either the current working folder, or a folder that is in the MATLAB path.

However, your control files should not be located within the MATLAB tree because they could be overwritten by subsequent installations.

Code Generation Control Objects and Methods

In this section...

“Overview” on page 26-8

“hdlnewcontrol” on page 26-8

“forEach” on page 26-8

“forAll” on page 26-13

“set” on page 26-13

“generateHDLFor” on page 26-14

“hdlnewcontrolfile” on page 26-15

Overview

Code generation control objects are instances of the class `slhdlcoder.ConfigurationContainer`. This section describes the public methods of that class that you can use in your control files. Other methods of this class are for MathWorks internal development use only. The methods are described in the following sections:

hdlnewcontrol

The `hdlnewcontrol` function constructs a code generation control object. The syntax is

```
object = hdlnewcontrol(mfilename);
```

The argument to `hdlnewcontrol` is the name of the control file itself. Use the `mfilename` function to pass in the file name string.

forEach

This method establishes an implementation mapping between an HDL block implementation and a selected block or set of blocks within the model. The syntax is

```
object.forEach({'modelscope'}, ...
```

```
'blocktype', {'block_parms'}, ...
'implementation', {'implementation_parms'})
```

The `forEach` method selects a set of blocks (modelscope) that is searched, during code generation, for instances of a specified type of block (`blocktype`). Code generation for each block instance encountered uses the HDL block implementation specified by the `implementation` parameter.

Note You can use the `hdlnewforeach` function to generate `forEach` method calls for insertion into your control files. See “Generating Selection/Action Statements with the `hdlnewforeach` Function” on page 26-18 for more information.

The following table summarizes the arguments to the `forEach` method.

Argument	Type	Description
<code>block_parms</code>	Cell array of strings	Reserved for future use. Pass in an empty cell array ({}) as a placeholder.
<code>blocktype</code>	String	Block specification that identifies the type of block that is to be mapped to the HDL block implementation. Block specification syntax is the same as that used in the <code>add-block</code> command. For built-in blocks, the <code>blocktype</code> is of the form 'built-in/blockname' For other blocks, <code>blocktype</code> must include the full path to the library containing the block, for example: 'dsparch4/Digital Filter'

Argument	Type	Description
implementation	String	<p>The implementation string represents an HDL block implementation to be used in code generation for blocks that meet the <code>modelscope</code> and <code>blocktype</code> search criteria. Every block has at least one implementation. “Blocks with Multiple Implementations” on page 11-16 provides guidelines for specifying implementations, and lists supported blocks and their implementations.</p>
implementation_parms	Cell array of p/v pairs	<p>Cell array of property/value pairs that set code generation parameters for the block implementation specified by the implementation argument. Specify parameters as: 'Property', value where 'Property' is the name of the property and value is the value applied to the property. If the implementation does not have parameters, or you want to use default parameters, pass in an empty cell array ({}).</p> <p>“Block Implementation Parameters” on page 11-50 describes the syntax of each parameter, and describes how the parameter affects generated code.</p> <p>“Blocks with Multiple Implementations” on page 11-16 lists supported blocks and their implementations and parameters.</p> <p>You can use the <code>hdlnewforeach</code> function to obtain the parameter names for selected block(s) in a model. See “Specifying Block Implementations and Parameters in the Control File” on page 26-17.</p>

Argument	Type	Description
modelscope	String or cell array of strings	<p>Strings defining one or more Simulink paths: <code>{'path1' 'path2' ... 'pathN'}</code></p> <p>Each path defines a modelscope: a set of blocks that participate in an implementation mapping. The set of blocks in a modelscope could include the entire model, blocks at a specified level of the model, or a specific block or subsystem. A path terminating in a wildcard ('*') includes blocks at or below the model level specified by the path. You can use the period (.) to represent the root-level model at the top of a modelscope, instead of explicitly coding the model name. For example: <code>./subsyslevel/block</code>. See also “Representation of the Root Model in modelscopes” on page 26-11 and “Resolution of modelscopes” on page 26-12. Syntax for modelscope paths is</p> <ul style="list-style-type: none"> • <code>'model/*'</code>: all blocks in the model • <code>'model/subsyslevel/block'</code>: a specific block within a specific level of the model • <code>'model/subsyslevel/subsystem'</code>: a specific subsystem block within a specific level of the model • <code>'model/subsyslevel/*'</code>: any block within a specific model level

Representation of the Root Model in modelscopes

You can represent the root-level model at the top of a modelscope as:

- The full model name, as in the following listing:

```
cfg.forEach( 'aModel/Subsystem/MinMax', ...
    'built-in/MinMax', {}, ...
    'default');
```

If you explicitly code the model name in a modelscope, and then save the model under a different name, the control file becomes invalid because it

references the previous model name. It is then required that you edit the control file and change all such modelscopes to reference the new model.

- The period (.) character, representing the current model as an abstraction, as in the following listing:

```
cfg.forEach( './Subsystem/MinMax', ...
    'built-in/MinMax', {}, ...
    'Cascade');
```

If you represent the model in this way, and then save the model under a different name, the control file does not require a change. Using the period to represent the root-level model makes the modelscope independent of the model name, and therefore more portable.

When you save HDL code generation settings to a control file, the period is used to represent the root-level model.

Resolution of modelscopes

A possible conflict exists in the `forEach` specifications in the following example:

```
% 1. Use default (multipliers) Gain block implementation
% for one specific Gain block within OneD_DCT8 subsystem
c.forEach('./OneD_DCT8/Gain14',...
    'built-in/Gain', {},...
    'default', {});
% 2. Use factored CSD Gain optimization
% for all Gain blocks at or below level of OneD_DCT8 subsystem.
c.forEach('./OneD_DCT8/*',...
    'built-in/Gain', {},...
    'default', {'ConstMultiplierOptimization','FCSD'});
```

The first `forEach` call defines an implementation mapping for a specific block within the subsystem `OneD_DCT8`. The second `forEach` call specifies a non-default implementation parameter ('`ConstMultiplierOptimization`') for all blocks within or below the subsystem `OneD_DCT8`.

The coder resolves such ambiguities by giving higher priority to the more specific `modelscope`. In the example, the first `modelscope` is the more specific.

Five levels of `modelscope` priority from most specific (1) to least specific (5) are defined:

- 1 A/B/C/block
- 2 A/B/C/*
- 3 A/B/*
- 4 *
- 5 Unspecified. Use the default implementation.

forall

This method is a shorthand form of `forEach`. Only one `modelscope` path is specified. The `modelscope` argument is specified as a string (not a cell array) and it is implicitly terminated with `'/*'`. The syntax is

```
object.forAll('modelscope', ...  
             'blocktype', {'block_parms'}, ...  
             'implementation', {'implementation_parms'})
```

Other arguments are the same as those described for “`forEach`” on page 26-8.

set

The `set` method sets one or more code generation properties. The syntax is

```
object.set('PropertyName', PropertyValue,...)
```

The argument list specifies one or more code generation options as property/value pairs. You can set all the code generation properties *except* the `HDLControlFiles` property.

Note If you specify the same property in both your control file and your `makehdl` command, the property will be set to the value specified in the control file.

Likewise, when generating code via the GUI, if you specify the same property in both your control file and the **HDL Coder** options panes, the property will be set to the value specified in the control file.

generateHDLFor

This method selects the model or subsystem from which code is to be generated. The syntax is

```
object.generateHDLFor('simulinkpath')
```

The argument is a string specifying the full path to the model or subsystem from which code is to be generated.

To make your control files more portable, you can represent the root-level model in the path as an abstraction, as in the following example:

```
function c = newforeachexamp
c = hdlnewcontrol(mfilename);

% Set top-level subsystem from which code is generated.
c.generateHDLFor('./symmetric_fir');
...
```

The above `generateHDLFor` call is valid for any model containing a subsystem named `symmetric_fir` at the root level.

Use of this method is optional. You can specify the same parameter in the **Generate HDL for** menu in the **HDL Coder** pane of the Configuration Parameters dialog box, or in a `makehdl` command.

hdlnewcontrolfile

The coder provides the `hdlnewcontrolfile` utility to help you construct code generation control files. Given a selection of one or more blocks from your model, `hdlnewcontrolfile` generates a control file containing `forEach` statements and comments providing information about supported implementations and parameters, for the selected blocks. The generated control file is automatically opened in the MATLAB editor for further customization. See the `hdlnewcontrolfile` function reference page for details.

Using Control Files in the Code Generation Process

In this section...

“Where to Locate Your Control Files” on page 26-16

“Making Your Control Files More Portable” on page 26-16

Where to Locate Your Control Files

Before you create a control file or use a control file in code generation, be sure to observe the following requirements for the location of control files:

- A control file must be stored in a folder that is in the MATLAB path, or the current working folder.
- Do not locate a control file within the MATLAB tree, because it could be overwritten by subsequent MATLAB installations.

Making Your Control Files More Portable

It can be advantageous to code your control files so that they are independent of a particular model name. To do this, use the period (.) to represent the root-level model at the beginning of all modelscope paths. For example:

```
cfg.forEach( './Subsystem/MinMax', ...  
    'built-in/MinMax', {}, ...  
    'Cascade');
```

If you code modelscopes in this way, all modelscopes are interpreted as references to the current model, rather than as references to an explicitly named model. Therefore, you can save your model under a different name, and references to the root-level model will be valid.

Specifying Block Implementations and Parameters in the Control File

In this section...

“Overview” on page 26-17

“Generating Selection/Action Statements with the `hdlnewforeach` Function” on page 26-18

Overview

The coder provides a default HDL block implementation for supported blocks. In addition, the coder provides selectable alternate HDL block implementations for several block types. Using selection/action statements (`forEach` or `forall` method calls) in a control file, you can specify the block implementation to be applied to all blocks of a given type (within a specific `modelscope`) during code generation. For many implementations, you can also pass in implementation parameters that provide additional control over code generation details.

You select HDL block implementations by specifying the implementation name as a string. “Blocks Supported for HDL Code Generation” on page 11-3 summarizes the supported blocks, their implementation names, and implementation parameters. Pass in the implementation name and implementation parameters (if any) to the implementation argument of a `forEach` or `forall` call. The following example specifies the `Tree` implementation for all `Sum` blocks in a model, with 2 output pipeline stages.

```
config.forEach('*',...
    'built-in/Sum', {},...
    'Tree', {'OutputPipeline', 2});
```

Given the implementation name, the coder calls the corresponding code generation method. You do not need to know internal details of the implementation classes.

Generating Selection/Action Statements with the `hdlnewforeach` Function

Determining the block path, type, implementation specification, and implementation parameters for a large number of blocks in a model can be time-consuming. Use the `hdlnewforeach` function to create selection/action statements in your control files. Given a selection of one or more blocks from your model, `hdlnewforeach` returns the following for each selected block, as string data in the MATLAB workspace:

- A `forEach` call coded with the `modelscope`, `blocktype`, and default implementation arguments for the block
- (Optional) A cell array of strings enumerating the available implementations for the block.
- (Optional) A cell array of strings enumerating the names of implementation parameters corresponding to the block implementations. `hdlnewforeach` does not list data types and other details of block implementation parameters. These details are described in “Blocks with Multiple Implementations” on page 11-16.

Having generated this information, you can copy and paste the strings into your control file.

`hdlnewforeach` Example

This example uses `hdlnewforeach` to construct a `forEach` call that specifies generation of two output pipeline stages after the output of a selected Sum block within the `sfir_fixed` example model. To create the control file:

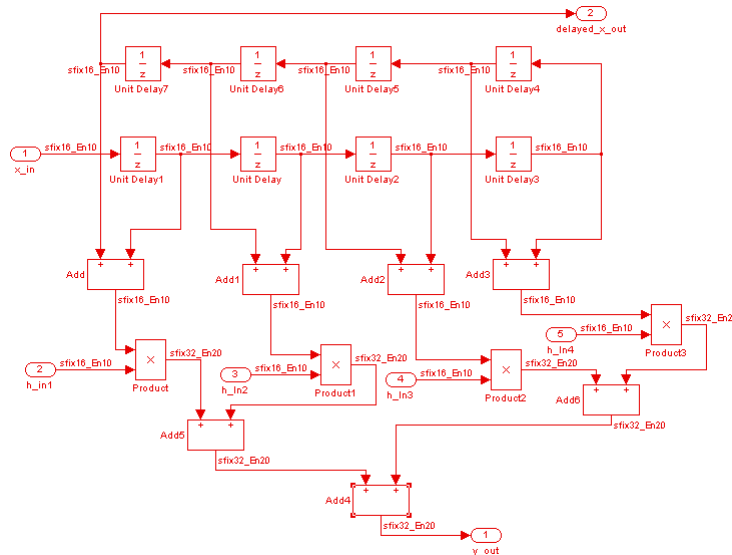
- 1** In the MATLAB Command Window, select **File > New > Blank M-File**. The MATLAB Editor opens an empty file.
- 2** Create a skeletal control file by entering the following code into the MATLAB Editor window:

```
function c = newforeachexamp
c = hdlnewcontrol(mfilename);

% Set top-level subsystem from which code is generated.
c.generateHDLFor('sfir_fixed/symmetric_fir');
```

```
% INSERT FOREACH CALL BELOW THIS LINE.
```

- 3** Save the file as `newforeachexamp.m`.
- 4** Open the `sfir_fixed` example model.
- 5** Close the `checkhdl` report window and activate the `sfir_fixed` model window.
- 6** In the `symmetric_fir` subsystem window, select the `Add4` block, as shown in the following figure.



Now you are ready to generate a `forEach` call for the selected block:

- 1** Type the following command at the MATLAB prompt.

```
[cmd,impl,parms] = hd1newforeach(gcb)
```

- 2** The command returns the following results:

```
c.forEach('./symmetric_fir/Add4',...
```

```
'built-in/Sum', {},...  
'default', {}); % Default architecture is 'Linear'
```

```
impl =  
  
    {3x1 cell}  
  
parms =  
  
    {1x2 cell}    {1x2 cell}    {1x2 cell}
```

The first return value, `cmd`, contains the generated `forEach` call. The `forEach` call specifies the default implementation for the Sum block, specified as `'default'`. Also by default, no parameters are passed in for this implementation.

- 3** The second return value, `impl`, is a cell array containing three strings representing the available implementations for the Sum block. The following example lists the contents of the `impl` array:

```
impl{1}  
  
ans =  
  
    'Linear'  
    'Cascade'  
    'Tree'
```

See the table for information about these implementations.

- 4** The third return value, `parms`, is a cell array containing three strings that represent the available implementations parameters corresponding to the previously listed Sum block implementations. The following example lists the contents of the `parms` array:

```
parms{1:3}
```

```

ans =

    'InputPipeline'    'OutputPipeline'

ans =

    'InputPipeline'    'OutputPipeline'

ans =

    'InputPipeline'    'OutputPipeline'

```

This listing shows that each of the Sum block implementations has two parameters, 'InputPipeline' and 'OutputPipeline'. This indicates that parameter/value pairs of the form {'OutputPipeline', val} can be passed in with any of the Sum block implementations.

hdlnewforeach does not provide information about the data type, valid range, or other constraints on val. Some implementation parameters take numeric values, while others take strings. See “Block Implementation Parameters” on page 11-50 for details on implementation parameters.

- 5** Copy the three lines of foreach code from the MATLAB Command Window and paste them into the end of your newforeachexamp.m file:

```

% INSERT FOREACH CALL BELOW THIS LINE.
c.foreach('./symmetric_fir/Add4',...
    'built-in/Sum', {},...
    'default', {}); % Default architecture is 'Linear'

```

- 6** In this example, you will specify the default Sum block implementation for the Add4 block, but with generation of two output pipeline stages before the final output. To do this, pass in the 'OutputPipeline' parameter with a value of 2. Modify the final line of the foreach call in your control file:

```

% INSERT FOREACH CALL BELOW THIS LINE.
c.foreach('./symmetric_fir/Add4',...
    'built-in/Sum', {},...
    'default', {'OutputPipeline', 2}); % Default architecture is 'Linear'

```

7 Save the control file.

8 The following code shows the complete control file:

```
function c = newforeachexamp
c = hdlnewcontrol(mfilename);

% Set top-level subsystem from which code is generated.
c.generateHDLFor('sfir_fixed/symmetric_fir');
% INSERT FOREACH CALLS HERE.
c.forEach('sfir_fixed/symmetric_fir/Add4',...
c.forEach('./symmetric_fir/Add4',...
'built-in/Sum', {},...
'default', {'OutputPipeline', 2}); % Default architecture is 'Linear'
```

Note For convenience, `hdlnewforeach` supports a more abbreviated syntax than that used in the previous example. See the `hdlnewforeach` reference page.

Generating Black Box Control Statements Using `hdlnewblackbox`

The `hdlnewblackbox` function provides a simple way to create the control file statements that are required to generate black box interfaces for one or more subsystems. `hdlnewblackbox` is similar to `hdlnewforeach`).

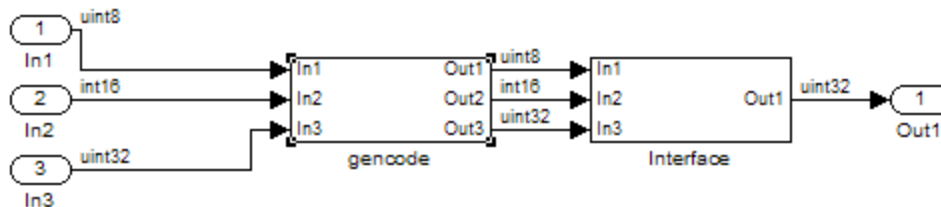
Given a selection of one or more subsystems from your model, `hdlnewblackbox` returns the following as string data in the MATLAB workspace for each selected subsystem:

- A `forEach` call coded with the `modelscope`, `blocktype`, and default implementation class (`SubsystemBlackBoxHDLInstantiation`) arguments for the block.
- (Optional) A cell array of strings enumerating the available implementations classes for the subsystem.
- (Optional) A cell array of cell arrays of strings enumerating the names of implementation parameters corresponding to the implementation classes. `hdlnewblackbox` does not list data types and other details of implementation parameters.

See `hdlnewblackbox` for the full syntax description of the function.

As an example, suppose that you want to generate black box control file statements for the subsystem `gencode` from the `subsystst` model. Using `hdlnewblackbox`, you can do this as follows:

- 1 Activate the `subsystst/top` subsystem window.
- 2 Select the subsystems for which you want to create control statements. In the following figure, `gencode` is selected.



3 Deselect the subsystemst/top subsystem.

4 Type the following command at the MATLAB prompt:

```
[cmd,impl,parms] = hdlnewblackbox
```

5 The command returns the following results:

```
cmd =

c.forEach('subsystemst/top/gencode',...
'built-in/SubSystem', {},...
'BlackBox', {});

impl =

{4x1 cell}

parms =

{} {1x11 cell} {1x12 cell} {1x11 cell}
```

The first return value, `cmd`, contains the generated `forEach` call. The `forEach` call specifies the default back box implementation for the subsystem blocks: `BlackBox`. Also by default, no parameters are passed in for this implementation.

- 6** The second return value, `impl`, is a cell array containing three strings listing available implementations for the Subsystem block. The following example lists the contents of the `impl` array:

```
>> impl{1}

ans =

    'hdldefaults.NoHDL Emission'
    'hdldefaults.SubsystemBlackBoxHDLInstantiation'
    'hdldefaults.XilinxBlackBoxHDLInstantiation'
    'hdldefaults.AlteraDSPBuilderBlackBox'
```

- 7** The third return value, `parms`, is a cell array containing strings that represent the available implementations parameters corresponding to the previously listed Subsystem block implementations. The parameters of interest in this case are those available for `BlackBox`. These are enumerated in `parms{2}`, as shown in the following listing:

```
parms{1}

ans =

Columns 1 through 4

    'ClockInputPort' [1x20 char]    'ResetInputPort'    'AddClockPort'

Columns 5 through 9

    'AddClockEnablePort' 'AddResetPort'    [1x20 char]    [1x20 char]    'EntityName'

Columns 10 through 11

    'InputPipeline' 'OutputPipeline'
```

Implementation parameters for subsystems and other black box interface classes are described in “Customize the Generated Interface” on page 18-63.

- 8** Having generated this information, you can now copy and paste the strings into a control file.

Support Packages

Support Packages

- “Support Packages and Support Package Installer” on page 27-2
- “Install This Support Package on Other Computers” on page 27-4
- “Open Examples for This Support Package” on page 27-6

Support Packages and Support Package Installer

What Is a Support Package?

A *support package* is an add-on that enables you to use a MathWorks product with specific third-party hardware and software.

Support packages can include:

- Simulink block libraries
- MATLAB functions, classes, and methods
- Firmware updates for the third-party hardware
- Automatic installation of third-party software
- Examples and tutorials

A *support package file* has a *.zip extension. This type of file contains MATLAB files, MEX files, and other supporting files required to install the support package. Use Support Package Installer to install these support package files.

A *support package installation file* has a *.mlpkginstall extension. You can double click this type of file to start Support Package Installer, which preselects a specific support package for installation. You can download these files from MATLAB Central File Exchange and use them to share support packages with others.

What Is Support Package Installer?

Support Package Installer is a wizard that guides you through the process of installing support packages.

You can use Support Package Installer to:

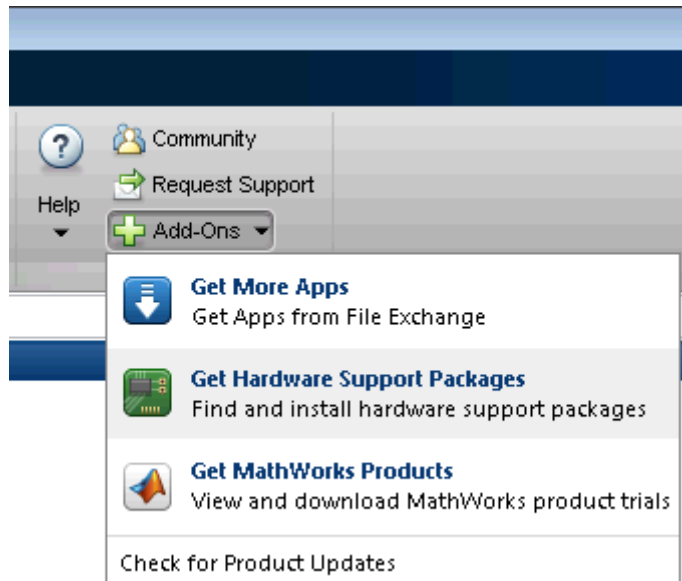
- Display a list of available, installable, installed, or updatable support packages
- Install, update, download, or uninstall a support package.
- Update the firmware on specific third-party hardware.

- Provide your MathWorks software with information about required third-party software.

If third-party software is included, Support Package Installer displays a list of the software and licenses for you to review before continuing.

You can start Support Package Installer in one of the following ways:

- On the MATLAB toolstrip, click **Add-Ons > Get Hardware Support Packages**.



- In the MATLAB Command Window, enter `supportPackageInstaller`.
- Double-click a support package installation file (*.mlpkginstall).

Install This Support Package on Other Computers

You can download a support package to one computer, and then install it on other computers. You can use this approach to:

- Save time when installing support packages on multiple computers.
- Install support packages on computers that are not connected to the Internet.

Before starting, select a computer to use for downloading. This computer must have the same base product license and platform as the computers upon which you are installing the support package. For example, suppose you want to install a Simulink support package on a group of computers that are running 64-bit Windows. To do so, you must first download the support package using a computer that has a Simulink license and is running 64-bit Windows.

Download the support package to one computer:

- 1** In the MATLAB Command Window, enter `supportPackageInstaller`.
- 2** In Support Package Installer, on the **Select an action** screen, choose **Install from Internet** or **Download from Internet**. Click **Next**.
- 3** On the following screen, select only one support package.

Notice the path of the **Download folder**. For example,
C:\MATLAB\SupportPackages\R2013b\downloads.

- 4** Using the file manager on your computer, open the downloads folder and observe its contents.
- 5** Using Support Package Installer, complete the installation or download process.

This process creates a folder within the **Download folder**. In some cases, if the support package requires another support package, this process creates an additional folder.

Prepare and share the support package files:

- 1 In the file manager, check how many folders were created during the installation or download process.
- 2 If more than one folder was created, combine the contents of the folders into the folder named after the support package.

For example,

C:\MATLAB\SupportPackages\R2013b\downloads*support_package_name*.

- 3 Make that folder available to other computers by sharing it on the network, or copying it to portable media, such as a USB flash drive.

Note Some support packages require that you install third-party software separately before completing the support package installation process. In that case, also make the third-party software available for installation on the other computers.

Install the support package on the other computers:

- 1 Run Support Package Installer on the other computer or computers.
- 2 On the **Install or update support package** screen, select the **Folder** option.
- 3 Use **Browse** to specify the location of the support package folder on the network or portable media.
- 4 Complete the instructions provided by Support Package Installer.

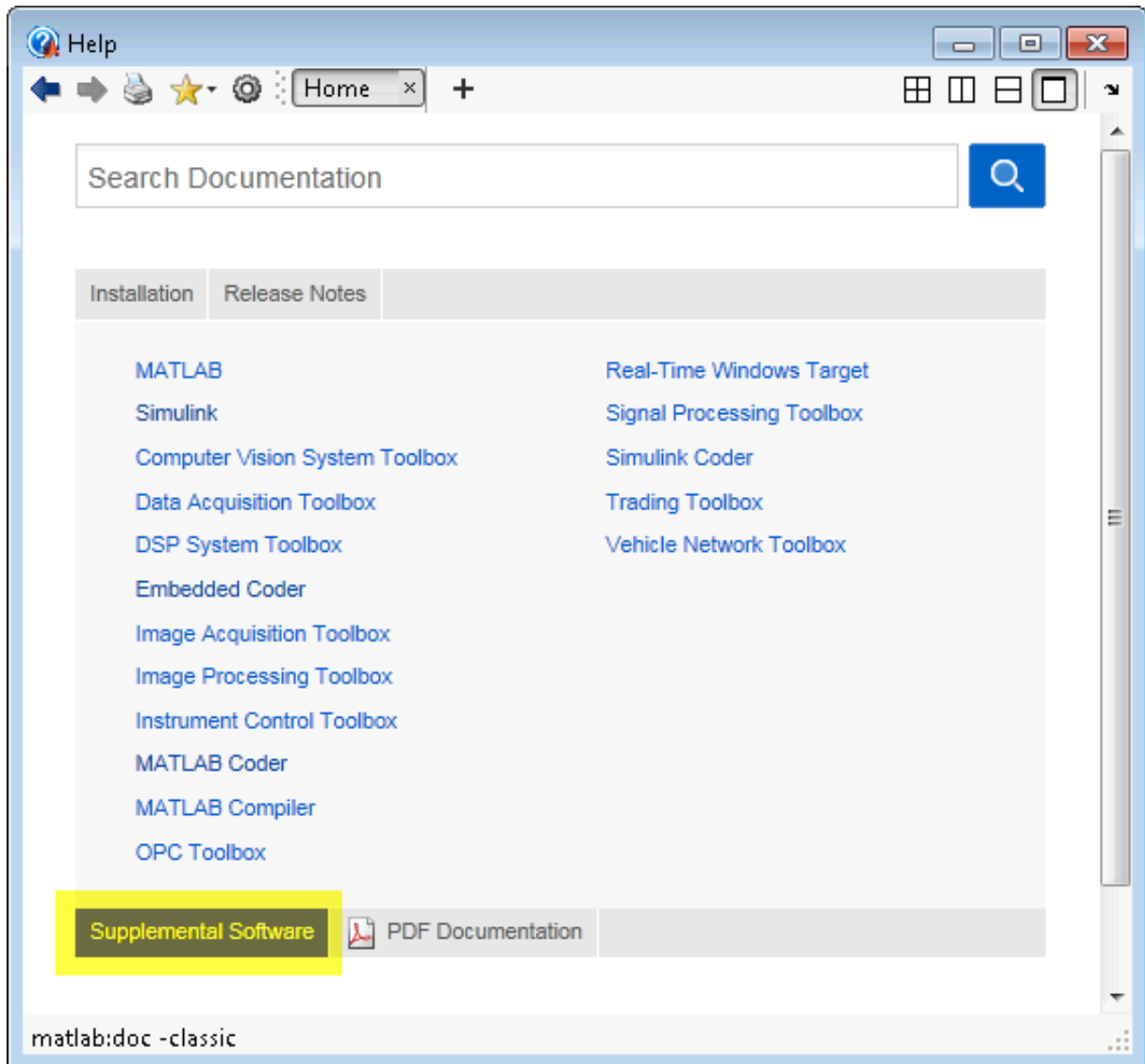
Open Examples for This Support Package

In this section...
“Using the Help Browser” on page 27-6
“Using the Block Library” on page 27-8
“Using Support Package Installer” on page 27-9

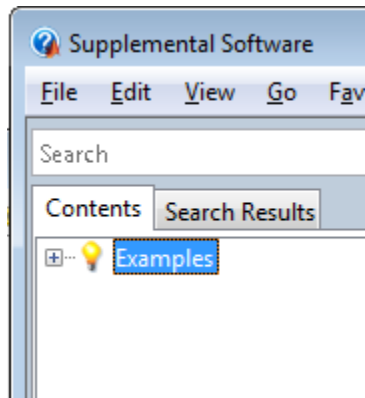
Using the Help Browser

You can open support package examples from the Help browser:

- 1 Enter `doc` in the MATLAB Command Window.
- 2 In the Help browser, click **Supplemental Software** in the lower left corner of the Home page.



3 In Supplemental Software, double-click **Examples**.



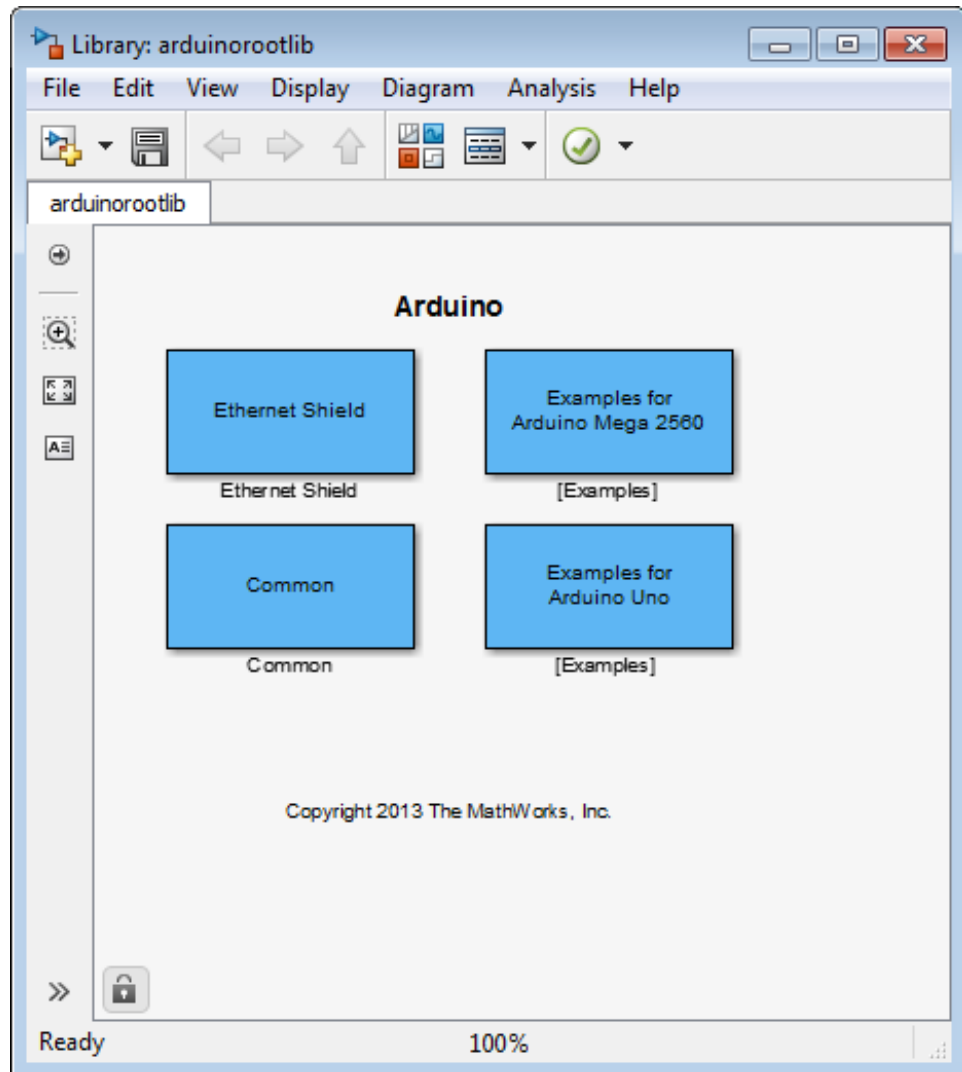
- 4 Select the examples for your support package.

Note For other types of examples, open the Help browser and search for your product name followed by “examples”.

Using the Block Library

To open support package from the support package block library:


- 1 Enter `simulink` in the MATLAB Command Window.
- 2 In Simulink Library Browser, open the support package block library.
- 3 In the block library, double-click the Examples block.



Using Support Package Installer

Support Package Installer (supportPackageInstaller) automatically displays the support package examples when you complete the process of installing and setting up a support package.

On the last screen in Support Package Installer, leave **Show support package examples** enabled and click **Finish**.

 Support Package Installer

Support package setup complete

You have completed the setup tasks.

Show support package examples

B

- Bit Concat block 13-50
- Bit Reduce block 13-50
- Bit Rotate block 13-50
- Bit Shift block 13-50
- Bit Slice block 13-50
- bit-true cycle-accurate models
 - bit-true to generated HDL code 14-2
- Bitwise Operator blocks 13-50
- block implementations
 - Constant 11-16
 - defined 26-5
 - Divide 11-16
 - Gain 11-16
 - Math Function 11-16
 - Maximum 11-16
 - Minimum 11-16
 - MinMax 11-16
 - multiple 11-16
 - parameters for 11-50
 - Product of Elements 11-16
 - restrictions on use of 11-29
 - special purpose 11-16
 - specifying in control file 26-17
 - Subsystem 11-16
 - Sum of Elements 11-16
 - summary of 11-3
- blocks
 - restrictions on use in test bench 11-49
 - supporting complex data type 11-103
- blockscope 26-8

C

- code generation control files. *See* control files
- code generation report 16-2
- Code, generation of
 - for referenced models 18-18
- complex data type
 - blocks supporting 11-103

- configuration parameters
 - EDA Tool Scripts pane 9-108
 - Choose synthesis tool 9-121
 - Compile command for Verilog 9-114
 - Simulation command 9-118
 - Simulation file postfix 9-116
 - Global Settings pane 9-25
 - Clocked process postfix 9-44
 - Complex imaginary part postfix 9-48
 - Complex real part postfix 9-47
 - Enable prefix 9-45
 - Entity conflict postfix 9-36
 - Inline VHDL configuration 9-67
 - Pipeline postfix 9-46
 - Split arch file postfix 9-43
 - HDL Code Generation pane 9-10
 - Test Bench pane 9-80
 - Clock enable delay 9-92
 - Clock high time (ns) 9-87
 - Clock low time (ns) 9-88
 - Reset length 9-95
 - Test bench name postfix 9-85
- Configuration Parameters dialog box
 - Code Generation (report)
 - Code-to-model 9-20
 - HDL Code Generation options in 9-2
- control files
 - control object method calls in 26-8
 - forAll 26-13
 - forEach 26-8
 - generateHDLFor 26-14
 - hdlnewcontrol 26-8
 - hdlnewcontrolfile 26-15
 - set 26-13
 - objects instantiated in 26-8
 - portability of 26-16
 - purpose of 26-4
 - required elements for 26-7
 - selecting block implementations in 26-5
 - specifying implementation mappings in 26-5

- statement types in
 - property setting 26-4
 - selection/action 26-4
- Cosimulation model 18-39

D

- Dual Port RAM block 13-3

E

- electronic design automation (EDA) tools
 - generation of scripts for
 - customized 21-2
- Enabled subsystems
 - code generation for 18-23

G

- generated model 15-24
- generated models
 - bit-true to generated HDL code 14-2
 - cycle-accuracy of 14-2
 - default options for 14-12
 - example of numeric differences 14-4
 - GUI options for 14-13
 - highlighted blocks in 14-12
 - latency example 14-9
 - makehdl properties for 14-15
 - naming conventions for 14-12
 - options for 14-12
- Generating cosimulation models 18-39

H

- HDL Code Generation options
 - in Configuration Parameters dialog box 9-2
 - in Model Explorer 9-3
- HDL Code menu 9-4
- HDL Code options
 - in Tools menu 9-4

- hdlnewforeach function
 - example 26-18
 - generating forEach calls with 26-18
- HTML code generation report 16-2

I

- implementation mapping
 - defined 26-5
- Interfaces, generation of
 - for Dual Port RAM block 13-3
 - for HDL Cosimulation blocks 18-37
 - for referenced models 18-21
 - for simple Dual Port RAM block 13-3
 - for Single Port RAM block 13-3

M

- MATLAB Function block
 - Distributed pipeline insertion 20-42
 - DistributedPipelining parameter
 - for 20-42
 - HDL code generation for 20-2
 - limitations 20-49
 - setting fixed point options 20-10
 - tutorial example 20-4
 - OutputPipeline parameter for 20-42
 - recommended settings for HDL code generation 20-37
 - speed optimization for 20-42
- MATLAB Function Block
 - design patterns in 20-23
- model configuration parameters
 - EDA Tool Scripts pane
 - Compile command for VHDL 9-113
 - Compile file postfix 9-111
 - Compile initialization 9-112
 - Compile termination 9-115
 - Generate EDA scripts 9-109
 - HDL lint tool 9-126

- lint command 9-128
- lint initialization 9-127
- lint termination 9-128
- Simulation initialization 9-117
- Simulation termination 9-120
- Simulation waveform viewing
 - command 9-119
- Synthesis command 9-125
- Synthesis file postfix 9-123
- Synthesis initialization 9-124
- Synthesis termination 9-126
- Global Settings pane
 - Balance Delays 9-53
 - Clock enable input port 9-29
 - Clock enable output port 9-52
 - Clock input port 9-28
 - Clock inputs 9-31
 - Comment in header 9-33
 - Concatenate type safe zeros 9-68
 - Emit time/date stamp in header 9-69
 - HDL coding standard 9-70
 - Hierarchical Distributed Pipelining 9-54
 - Input data type 9-49
 - Loop unrolling 9-65
 - Max computation latency 9-62
 - Max oversampling 9-61
 - Minimize clock enables 9-57
 - Minimize intermediate signals 9-72
 - Module name prefix 9-38
 - Optimize timing controller 9-55
 - Output data type 9-50
 - Oversampling factor 9-32
 - Package postfix 9-37
 - RAM mapping threshold (bits) 9-60
 - Represent constant values by
 - aggregates 9-63
 - Reserved word postfix 9-38
 - Reset asserted level 9-27
 - Reset input port 9-30
 - Reset type 9-26
 - Scalarize vector ports 9-71
 - Split entity and architecture 9-40
 - Split entity file postfix 9-42
 - Use "rising_edge" for registers 9-64
 - Use Verilog `timescale directives 9-66
 - Verilog file extension 9-34
 - VHDL file extension 9-35
- HDL Code Generation pane
 - Folder 9-14
 - Generate HDL code 9-15
 - Generate HDL for: 9-12
 - Generate model Web view 9-20
 - Generate parameterized HDL code from
 - masked subsystem 9-75
 - Generate resource optimization
 - report 9-19
 - Generate resource utilization report 9-18
 - Generate traceability report 9-17
 - Generate validation model 9-16
 - Include requirements in block
 - comments 9-73
 - Initialize all RAM blocks 9-76
 - Inline MATLAB Function block
 - code 9-74
 - Language 9-13
 - RAM Architecture 9-76
- pane
 - Generate cosimulation model 9-84
 - HDL test bench 9-81
- Test Bench pane
 - cosimulation blocks 9-82
 - Force clock 9-86
 - Force clock enable 9-91
 - Force reset 9-94
 - Hold input data between samples 9-97
 - Hold time (ns) 9-89
 - Ignore output data checking (number of
 - samples) 9-103
 - Initialize test bench inputs 9-98
 - Multi-file test bench 9-99

- Setup time (ns) 9-90
- Test bench data file name postfix 9-102
- Use file I/O to read/write test bench data 9-103
- Model Explorer
 - HDL Code Generation options in 9-3
- modelscope 26-8

N

- No-op block implementations 18-66

O

- options
 - Cosimulation model 18-39

P

- Pass-through block implementations 18-66

R

- RAM
 - blocks 13-3

- inferring 13-3

S

- Simple Dual Port RAM block 13-3
- Single Port RAM block 13-3
- Stateflow charts
 - code generation 19-2
 - requirements for 19-4
 - restrictions on 19-4

T

- Triggered subsystems
 - code generation for 18-23

V

- validation model 15-24

W

- Web view
 - browser requirements 16-28