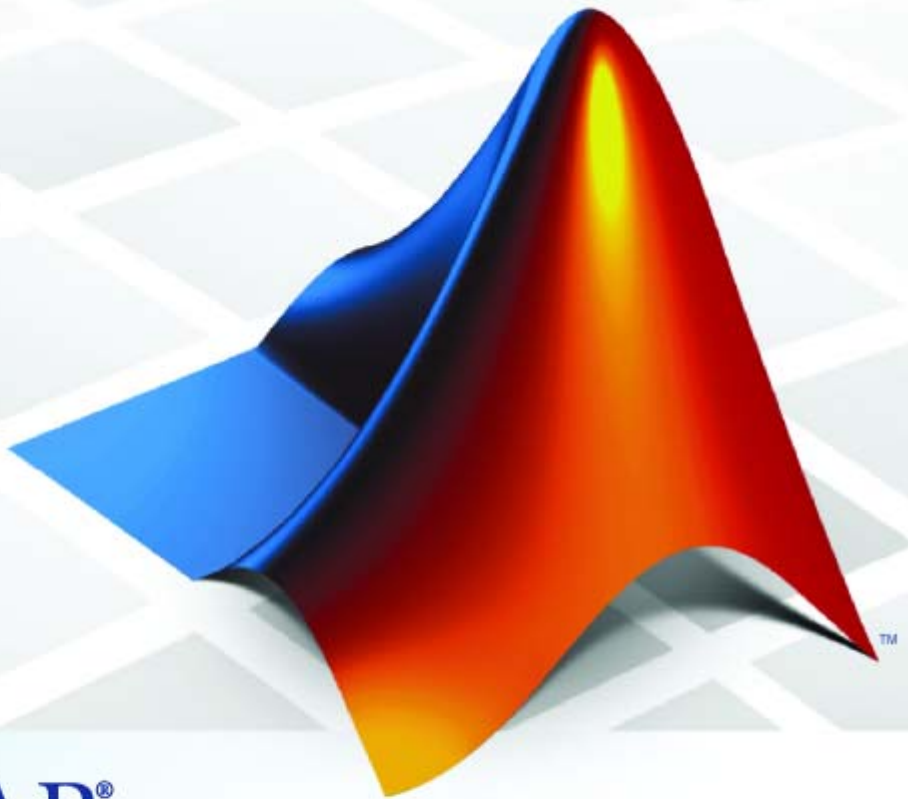


Simulink® HDL Coder™ 1

User's Guide



MATLAB®
& **SIMULINK®**

How to Contact The MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
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.

Simulink® HDL Coder™ User's Guide

© COPYRIGHT 2006–2009 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

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

Revision History

September 2006 Online only
March 2007 Online only
September 2007 Online only
March 2008 Online only
October 2008 Online only
March 2009 Online only

New for Version 1.0 (Release 2006b)
Updated for Version 1.1 (Release 2007a)
Revised for Version 1.2 (Release 2007b)
Revised for Version 1.3 (Release 2008a)
Revised for Version 1.4 (Release 2008b)
Revised for Version 1.5 (Release 2009a)

Getting Started

1

Product Overview	1-2
Automated HDL Code Generation in the Hardware	
Development Process	1-2
Summary of Key Features	1-3
Expected Users and Prerequisites	1-7
Software Requirements and Installation	1-8
Software Requirements	1-8
Installing the Software	1-9
Available Help and Demos	1-10
Online Help	1-10
Demos	1-10

Introduction to HDL Code Generation

2

Before You Generate Code	2-2
Overview of Exercises	2-3
The sfir_fixed Demo Model	2-4
Generating HDL Code Using the Command Line	
Interface	2-7
Overview	2-7
Creating a Directory and Local Model File	2-7
Initializing Model Parameters with hdlsetup	2-8
Generating a VHDL Entity from a Subsystem	2-10

Generating VHDL Test Bench Code	2-12
Verifying Generated Code	2-13
Generating a Verilog Module and Test Bench	2-14
Generating HDL Code Using the GUI	2-16
Simulink® HDL Coder GUI Overview	2-16
Creating a Directory and Local Model File	2-19
Viewing Coder Options in the Configuration Parameters	
Dialog Box	2-20
Creating a Control File	2-22
Initializing Model Parameters with hdlsetup	2-24
Selecting and Checking a Subsystem for HDL	
Compatibility	2-26
Generating VHDL Code	2-27
Generating VHDL Test Bench Code	2-30
Verifying Generated Code	2-31
Generating Verilog Model and Test Bench Code	2-31
Simulating and Verifying Generated HDL Code	2-32

Code Generation Options in the Simulink® HDL Coder GUI

3

Viewing and Setting HDL Coder Options	3-2
HDL Coder Options in the Configuration Parameters Dialog	
Box	3-2
HDL Coder Options in the Model Explorer	3-3
HDL Coder Menu	3-5
HDL Coder Pane: General	3-6
HDL Coder Top-Level Pane Overview	3-7
File name	3-8
Generate HDL for	3-9
Language	3-10
Directory	3-11
Code Generation Output	3-12
Generate traceability report	3-13

HDL Coder Pane: Global Settings	3-14
Global Settings Overview	3-16
Reset type	3-17
Reset asserted level	3-18
Clock input port	3-19
Clock enable input port	3-20
Reset input port	3-21
Comment in header	3-22
Verilog file extension	3-23
VHDL file extension	3-24
Entity conflict postfix	3-25
Package postfix	3-26
Reserved word postfix	3-27
Split entity and architecture	3-28
Split entity file postfix	3-30
Split arch file postfix	3-31
Clocked process postfix	3-32
Enable prefix	3-33
Pipeline postfix	3-34
Complex real part postfix	3-35
Complex imaginary part postfix	3-36
Input data type	3-37
Output data type	3-38
Clock enable output port	3-40
Represent constant values by aggregates	3-41
Use "rising_edge" for registers	3-42
Loop unrolling	3-43
Cast before sum	3-44
Use Verilog `timescale directives	3-45
Inline VHDL configuration	3-46
Concatenate type safe zeros	3-47
Optimize timing controller	3-48
HDL Coder Pane: Test Bench	3-50
Test Bench Overview	3-52
Test bench name postfix	3-53
Force clock	3-54
Clock high time (ns)	3-55
Clock low time (ns)	3-56
Hold time (ns)	3-57
Setup time (ns)	3-58
Force clock enable	3-59
Clock enable delay (in clock cycles)	3-60
Force reset	3-62

Reset length (in clock cycles)	3-63
Hold input data between samples	3-65
Initialize test bench inputs	3-66
Multi-file test bench	3-67
Test bench reference postfix	3-69
Test bench data file name postfix	3-70
Ignore output data checking (number of samples)	3-71
Generate cosimulation blocks	3-73
HDL Coder Pane: EDA Tool Scripts	3-74
EDA Tool Scripts Overview	3-76
Generate EDA scripts	3-77
Compile file postfix	3-78
Compile Initialization	3-79
Compile command for VHDL	3-80
Compile command for Verilog	3-81
Compile termination	3-82
Simulation file postfix	3-83
Simulation initialization	3-84
Simulation command	3-85
Simulation waveform viewing command	3-86
Simulation termination	3-87
Synthesis file postfix	3-88
Synthesis initialization	3-89
Synthesis command	3-90
Synthesis termination	3-91

Generating HDL Code for Multirate Models

4

Overview	4-2
Configuring Multirate Models for HDL Code	
Generation	4-3
Overview	4-3
Configuring Model Parameters	4-3
Configuring Sample Rates in the Model	4-4
Constraints for Rate Transition Blocks and Other Blocks in Multirate Models	4-4

Example: Model with a Multirate DUT	4-6
Properties Supporting Multirate Code Generation ...	4-9
Overview	4-9
HoldInputDataBetweenSamples	4-9
OptimizeTimingController	4-9

Code Generation Control Files

5

Overview of Control Files	5-2
What Is a Control File?	5-2
Selectable Block Implementations and Implementation Parameters	5-3
Implementation Mappings	5-4
Control File Demo	5-4
 Structure of a Control File	 5-5
 Code Generation Control Objects and Methods	 5-7
Overview	5-7
hdlnewcontrol	5-7
forEach	5-7
forall	5-12
set	5-12
generateHDLFor	5-13
hdlnewcontrolfile	5-14
 Using Control Files in the Code Generation Process ..	 5-15
Where to Locate Your Control Files	5-15
Creating a Control File and Saving Your HDL Code Generation Settings	5-15
Making Your Control Files More Portable	5-19
Associating an Existing Control File with Your Model	5-19
Detaching a Control File from Your Model	5-22
Setting Up HDL Code Generation Defaults with a Control File	5-22

Specifying Block Implementations and Parameters in the Control File	5-24
Overview	5-24
Generating Selection/Action Statements with the hdlnewforeach Function	5-24

Specifying Block Implementations and Parameters for HDL Code Generation

6

Summary of Block Implementations	6-2
Blocks with Multiple Implementations	6-22
Overview	6-22
Implementations for Commonly Used Blocks	6-23
Math Function Block Implementations	6-27
Divide Block Implementations	6-31
Subsystem Interfaces and Special-Purpose Implementations	6-33
A Note on Cascade Implementations	6-34
Block-Specific Usage, Requirements, and Restrictions for HDL Code Generation	6-35
Block Usage, Requirements, and Restrictions	6-35
Restrictions on Use of Blocks in the Test Bench	6-40
Block Implementation Parameters	6-41
Overview	6-41
CoeffMultipliers	6-41
Distributed Arithmetic Implementation Parameters for Digital Filter Block	6-42
InputPipeline	6-52
OutputPipeline	6-53
ResetType	6-53
Interface Generation Parameters	6-54
Blocks That Support Complex Data	6-56
Complex Coefficients and Data Support for the Digital Filter Block	6-60

Accessing the hdl demolib Library Blocks	7-2
RAM Blocks	7-4
Overview of RAM Blocks	7-4
Dual Port RAM Block	7-6
Simple Dual Port RAM Block	7-7
Single Port RAM Block	7-9
Code Generation with RAM Blocks	7-12
Generic RAM and ROM Demos	7-13
Limitations for RAM Blocks	7-13
HDL Counter	7-15
Overview	7-15
Counter Modes	7-15
Control Ports	7-17
Defining the Counter Data Type and Size	7-20
HDL Implementation and Implementation Parameters ..	7-21
Parameters and Dialog Box	7-22
HDL FFT	7-27
Overview	7-27
Block Inputs and Outputs	7-28
HDL Implementation and Implementation Parameters ..	7-30
Parameters and Dialog Box	7-30
Bitwise Operators	7-35
Overview of Bitwise Operator Blocks	7-35
Bit Concat	7-37
Bit Reduce	7-39
Bit Rotate	7-41
Bit Shift	7-43
Bit Slice	7-45

Generating Bit-True Cycle-Accurate Models

8

Overview of Generated Models	8-2
Example: Numeric Differences	8-4
Example: Latency	8-8
Defaults and Options for Generated Models	8-12
Defaults for Model Generation	8-12
GUI Options	8-13
Generated Model Properties for makehdl	8-14
Fixed-Point and Double-Precision Limitations for Generated Models	8-17
Fixed-Point Limitation	8-17
Double-Precision Limitation	8-17

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

9

Creating and Using a Code Generation Report	9-2
Traceability and the Code Generation Report	9-2
Generating an HTML Code Generation Report from the GUI	9-4
Generating an HTML Code Generation Report from the Command Line	9-7
Keeping the Report Current	9-9
Tracing from Code to Model	9-9
Tracing from Model to Code	9-11
Mapping Model Elements to Code Using the Traceability Report	9-15
HTML Code Generation Report Limitations	9-17
HDL Compatibility Checker	9-18

Supported Blocks Library	9-22
Annotating Generated Code with Comments and Requirements	9-24
Code Tracing Using the Mapping File	9-25

Interfacing Subsystems and Models to HDL Code

10

Overview of HDL Interfaces	10-2
Generating a Black Box Interface for a Subsystem	10-3
Generating Black Box Control Statements Using hdlnewblackbox	10-5
Generating Interfaces for Referenced Models	10-10
Code Generation for Enabled Subsystems	10-11
Code Generation for HDL Cosimulation Blocks	10-13
Customizing the Generated Interface	10-15
Pass-Through and No-Op Implementations	10-17
Limitation on Generated Verilog Interfaces	10-18

Stateflow HDL Code Generation Support

11

Introduction to Stateflow HDL Code Generation	11-2
-----------------------------------------------------	------

Overview	11-2
Demos and Related Documentation	11-2
Quick Guide to Requirements for Stateflow HDL Code	
Generation	11-4
Overview	11-4
Location of Charts in the Model	11-4
Data Type Usage	11-4
Chart Initialization	11-5
Registered Output	11-5
Restrictions on Imported Code	11-6
Other Restrictions	11-6
Mapping Chart Semantics to HDL	
Software Realization of Chart Semantics	11-8
Hardware Realization of Stateflow Semantics	11-10
Restrictions for HDL Realization	11-13
Using Mealy and Moore Machine Types in HDL Code	
Generation	11-15
Overview	11-15
Generating HDL for a Mealy Finite State Machine	11-16
Generating HDL Code for a Moore Finite State Machine ..	11-19
Structuring a Model for HDL Code Generation	
11-24	
Design Patterns Using Advanced Chart Features	
Temporal Logic	11-30
Graphical Function	11-33
Hierarchy and Parallelism	11-35
Stateless Charts	11-39
Truth Tables	11-42

Generating HDL Code with the Embedded MATLAB Function Block

12

Introduction	12-2
--------------------	------

HDL Applications for the Embedded MATLAB Function Block	12-2
Related Documentation and Demos	12-3
Tutorial Example: Incrementer	12-4
Example Model Overview	12-4
Setting Up	12-7
Creating the Model and Configuring General Model Settings	12-8
Adding an Embedded MATLAB Function Block to the Model	12-8
Setting Optimal Fixed-Point Options for the Embedded MATLAB Function Block	12-10
Programming the Embedded MATLAB Function Block ...	12-12
Constructing and Connecting the DUT_eML_Block Subsystem	12-15
Compiling the Model and Displaying Port Data Types ...	12-20
Simulating the eml_hdl_incrementer_tut Model	12-20
Generating HDL Code	12-21
Useful Embedded MATLAB Function Block Design	
Patterns for HDL	12-25
The eml_hdl_design_patterns Library	12-25
Efficient Fixed-Point Algorithms	12-27
Using Persistent Variables to Model State	12-31
Creating Intellectual Property with the Embedded MATLAB Function Block	12-32
Modeling Control Logic and Simple Finite State Machines	12-33
Modeling Counters	12-35
Modeling Hardware Elements	12-36
Using Fixed-Point Bitwise Functions	12-39
Overview	12-39
Bitwise Functions Supported for HDL Code Generation ..	12-39
Bit Slice and Bit Concatenation Functions	12-44
Shift and Rotate Functions	12-45
Using Complex Signals	12-49
Introduction	12-49
Declaring Complex Signals	12-49
Conversion Between Complex and Real Signals	12-51

Arithmetic Operations on Complex Numbers	12-51
Support for Vectors of Complex Numbers	12-55
Other Operations on Complex Numbers	12-56
Distributed Pipeline Insertion	12-58
Overview	12-58
Example: Multiplier Chain	12-59
Limitations	12-67
Recommended Practices	12-68
Introduction	12-68
Use Compiled External M-Functions on the Embedded	
MATLAB Path	12-68
Build the Embedded MATLAB Code First	12-68
Use the hdlfmath Utility for Optimized FIMATH	
Settings	12-69
Use Optimal Fixed-Point Option Settings	12-70
Language Support	12-72
Fixed-Point Runtime Library Support	12-72
Variables and Constants	12-73
Use of Nontunable Parameter Arguments	12-77
Arithmetic Operators	12-77
Relational Operators	12-78
Logical Operators	12-79
Control Flow Statements	12-79
Other Limitations	12-81

Generating Scripts for HDL Simulators and Synthesis Tools

13

Overview of Script Generation for EDA Tools	13-2
Defaults for Script Generation	13-3
Custom Script Generation	13-4

Overview	13-4
Structure of Generated Script Files	13-4
Properties for Controlling Script Generation	13-5
Controlling Script Generation with the EDA Tool Scripts GUI Pane	13-8

Property Reference

14

Language Selection Properties	14-2
File Naming and Location Properties	14-2
Reset Properties	14-2
Header Comment and General Naming Properties	14-3
Script Generation Properties	14-4
Port Properties	14-5
Advanced Coding Properties	14-6
Test Bench Properties	14-7
Generated Model Properties	14-9

Properties — Alphabetical List

15

Function Reference

16

Code Generation Functions	16-2
Utility Functions	16-3
Control File Utilities	16-4

Functions — Alphabetical List

17

Examples

A

Generating HDL Code Using the Command Line Interface	A-2
Generating HDL Code Using the GUI	A-2
Verifying Generated HDL Code in an HDL Simulator ..	A-2

Index

Getting Started

- “Product Overview” on page 1-2
- “Expected Users and Prerequisites” on page 1-7
- “Software Requirements and Installation” on page 1-8
- “Available Help and Demos” on page 1-10

Product Overview

In this section...
“Automated HDL Code Generation in the Hardware Development Process” on page 1-2
“Summary of Key Features” on page 1-3

Automated HDL Code Generation in the Hardware Development Process

Simulink® HDL Coder™ software lets you generate hardware description language (HDL) code based on Simulink® models and Stateflow® finite-state machines. The coder brings the Model-Based Design approach into the domain of application-specific integrated circuit (ASIC) and field programmable gate array (FPGA) development. Using the coder, system architects and designers can spend more time on fine-tuning algorithms and models through rapid prototyping and experimentation and less time on HDL coding.

Typically, you use a Simulink model to simulate a design intended for realization as an ASIC or FPGA. Once satisfied that the model meets design requirements, you run the Simulink HDL Coder compatibility checker utility to examine model semantics and blocks for HDL code generation compatibility. You then invoke the coder, using either the command line or the graphical user interface. The coder generates VHDL or Verilog code that implements the design embodied in the model.

Usually, you also generate a corresponding test bench. You can use the test bench with HDL simulation tools to drive the generated HDL code and evaluate its behavior. The coder generates scripts that automate the process of compiling and simulating your code in these tools. You can also use EDA Simulator Link™ MQ, EDA Simulator Link IN or EDA Simulator Link DS software from The MathWorks™ to cosimulate generated HDL entities within a Simulink model.

The test bench feature increases confidence in the correctness of the generated code and saves time spent on test bench implementation. The design and test process is fully iterative. At any point, you can return to the original model, make modifications, and regenerate code.

When the design and test phase of the project has been completed, you can easily export the generated HDL code to synthesis and layout tools for hardware realization. The coder generates synthesis scripts for the Synplify® family of synthesis tools.

Extending the Code Generation Process

There are a number of ways to extend the code generation process.

By attaching a *code generation control file* to your model, you can direct many details of the code generation process. At the simplest level, you can use a control file to set code generation options; such a control file could be used as a template for code generation in your organization.

Control files also let you specify how code is generated for selected sets of blocks within the model. The coder provides alternate HDL *block implementations* for a variety of blocks. You can use statements in a control file to select from among implementations optimized for characteristics such as speed, chip area, or low latency.

In some cases, block-specific optimizations may introduce latencies (delays) or numeric computations (for example, saturation or rounding operations) in the generated code that are not in the original model. To help you evaluate such cases, the coder creates a *generated model* — a Simulink model that corresponds exactly to the generated HDL code. This generated model lets you run simulations that produce results that are bit-true to the HDL code, and whose timing is cycle-accurate with respect to the HDL code.

You can interface generated HDL code to existing or legacy HDL code. One way to do this is to use a subsystem in your model as a placeholder for an HDL entity, and generate a *black box* interface (comprising I/O port definitions only) to that entity. Another way is to generate a cosimulation interface by placing an HDL Cosimulation block in your model.

Summary of Key Features

Key features and components of the coder include

- Generation of synthesizable VHDL or Verilog code from Simulink models and Stateflow charts

- Code generation configured and initiated via graphical user interface, command-line interface, or M-file programs
- Test bench generation (VHDL or Verilog) for validating generated code
- Generation of models that are bit-true and cycle-accurate with respect to generated HDL code
- Numerous options for controlling the contents and style of the generated HDL code and test bench
- Block support:
 - Simulink built-in blocks
 - Signal Processing Blockset™ blocks
 - EDA Simulator Link MQ HDL Cosimulation block
 - EDA Simulator Link IN HDL Cosimulation block
 - EDA Simulator Link DS HDL Cosimulation block
 - Stateflow chart
 - Embedded MATLAB™ Function block
 - Library of HDL-specific block implementations for FFT, hardware counter, bitwise operators, and RAMs
 - User-selectable optimized block implementations provided for commonly used blocks
- Code generation control files support:
 - Selection of alternate block implementations for specific blocks or sets of blocks in the model
 - Specification of code generation options (such as input or output pipelining) for most block implementations
 - Setting of general code generation options
 - Selection of the model or subsystem from which code is to be generated.
 - Definition of default or template HDL code generation settings for your organization

- Generation of subsystem-based identification comments and mapping files for easy tracing of HDL entities back to corresponding elements of the original model
- Text from DocBlock and Simulink annotations rendered as comments in generated code
- Generation of interfaces to existing HDL code via:
 - Black box subsystem implementation
 - Cosimulation with Mentor Graphics® ModelSim® HDL simulator (requires EDA Simulator Link MQ)
 - Cosimulation with Cadence Incisive® HDL simulator (requires EDA Simulator Link IN software)
 - Cosimulation with Synopsis Discovery VCS HDL simulator (requires EDA Simulator Link DS software)
- Compatibility checker utility that examines your model for HDL code generation compatibility, and generates HTML report with hyperlinks to problematic blocks
- Generation of scripts for EDA tools:
 - Mentor Graphics ModelSim
 - Synplify
- Model features supported for code generation:
 - Real data types (fixed-point and double)

Note Results obtained from HDL code generated for models using double data types cannot be guaranteed to be bit-true to results obtained from simulation of the original model.

- Complex signals can be used in the test bench without restriction.
- Complex signals can be used in the DUT with a restricted set of blocks (see “Blocks That Support Complex Data” on page 6-56)
- Fixed-step, discrete, single-rate and multirate models

- Scalar and vector ports (row or column vectors only)

Expected Users and Prerequisites

Users of this product are system and hardware architects and designers who develop, optimize, and verify ASICs or FPGAs. These designers are experienced with VHDL or Verilog but can benefit from automated HDL code generation.

Users are expected to have prerequisite knowledge in the following areas:

- Hardware design and system integration
- VHDL or Verilog
- MATLAB®
- Simulink
- Simulink® Fixed Point™
- Signal Processing Blockset
- HDL simulators, such as the Mentor Graphics ModelSim simulator or Cadence Incisive simulator
- Synthesis tools, such as Synplify

Software Requirements and Installation

In this section...
“Software Requirements” on page 1-8
“Installing the Software” on page 1-9

Software Requirements

The coder requires the following software from The MathWorks:

- MATLAB
- Simulink
- Simulink Fixed Point
- Fixed-Point Toolbox™

The following related products are recommended for use with the coder:

- Stateflow
- Filter Design Toolbox™ (This software is required for generating HDL code for the Digital Filter block in certain cases. See “Summary of Block Implementations” on page 6-2.)
- EDA Simulator Link IN
- EDA Simulator Link MQ
- EDA Simulator Link DS
- Signal Processing Toolbox™
- Signal Processing Blockset

Software Requirements for Demos

To operate some demos shipped with this release, the following related products are required:

- Filter Design Toolbox

- Filter Design HDL Coder™
- EDA Simulator Link MQ
- Communications Toolbox™ (required to use Viterbi Decoder demo)
- Communications Blockset™ (required to use Viterbi Decoder demo)
- Image Processing Toolbox™ (required to use Image Reconstruction demos)

VHDL and Verilog Language Support

Before installing the coder , make sure that you have compatible compilers and other tools. Generated code is compatible with HDL compilers, simulators and other tools that support:

- VHDL versions 93 and 02
- Verilog-2001 (IEEE 1364-2001) or later

Installing the Software

For information on installing the required software listed previously, and optional software, see the MATLAB installation documentation for your platform.

After completing your installation:

- Read “Before You Generate Code” on page 2-2 to learn about recommended practices for ensuring that your models are compatible with HDL code generation.
- Work through the examples in Chapter 2, “Introduction to HDL Code Generation” to acquaint yourself with the operation of the product.

Available Help and Demos

In this section...
“Online Help” on page 1-10
“Demos” on page 1-10

Online Help

The following online help is available:

- Online help is available in the MATLAB Help browser. Click the **Simulink HDL Coder** product link in the browser’s Contents pane.
- To view documentation in PDF format, click the **Simulink HDL Coder > Printable Documentation (PDF)** link in the browser’s Contents pane.
- M-help for the command-line interface functions `makehdl`, `makehdltb`, `checkhdl`, `hdl1lib`, and `hdlsetup` is available through the `doc` and `help` commands. For example:

```
help makehdl
```

Demos

To access models demonstrating aspects of HDL code generation:

- 1 In the command-line window, type the following command:

```
demods
```

The **Help** window opens.

- 2 In the **Demos** pane on the left, select **Simulink > Simulink HDL Coder**.
- 3 The right pane displays hyperlinks to the available demos. Click the link to the desired demo and follow the demo instructions.

Introduction to HDL Code Generation

- “Before You Generate Code” on page 2-2
- “Overview of Exercises” on page 2-3
- “The `sfir_fixed` Demo Model” on page 2-4
- “Generating HDL Code Using the Command Line Interface” on page 2-7
- “Generating HDL Code Using the GUI” on page 2-16
- “Simulating and Verifying Generated HDL Code” on page 2-32

Before You Generate Code

The exercises in this introduction use a preconfigured demo model. All blocks in this demo model support HDL code generation, and the parameters of the model itself have been configured properly for HDL code generation.

After you complete the exercises, you will probably proceed to generating HDL code from your existing models, or newly constructed models. Before you generate HDL code from your own models, you should do the following to ensure that your models are HDL code generation compatible:

- Use the `hdl1lib.m` utility to create a library of all blocks that are currently supported for HDL code generation, as described in “Supported Blocks Library” on page 9-22. By constructing models with blocks from this library, you can ensure HDL compatibility for all your models.

The set of supported blocks will change in future releases, so you should rebuild your supported blocks library each time you install a new version of this product.

- Use the **Run Compatibility Checker** option (described in “Selecting and Checking a Subsystem for HDL Compatibility” on page 2-26) to check HDL compatibility of your model or DUT and generate an HDL Code Generation Check Report.

Alternatively, you can invoke the `checkhdl` function (see `checkhdl`) to run the compatibility checker.

- Before generating code, use the M-file utility `hdlsetup.m`, as described in “Initializing Model Parameters with `hdlsetup`” on page 2-8, to set up your model for HDL code generation quickly and consistently.

Overview of Exercises

The coder supports HDL code generation in your choice of environments:

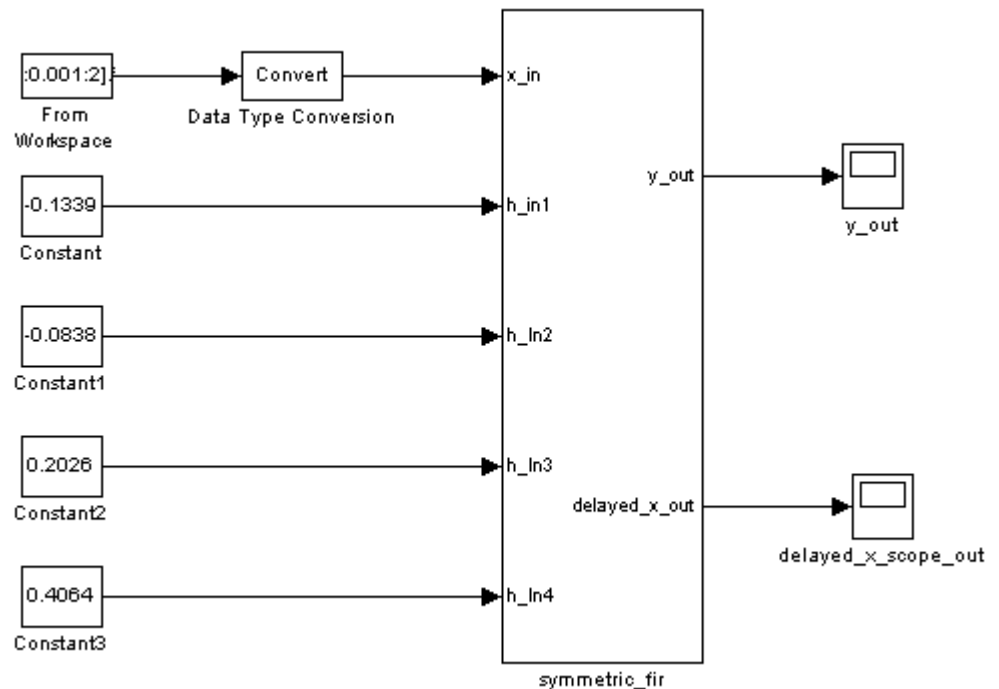
- The MATLAB Command Window supports code generation using the `makehdl`, `makehdltb`, and other functions.
- The Simulink GUI (the Configuration Parameters dialog box and/or Model Explorer) provides an integrated view of the model simulation parameters and HDL code generation parameters and functions.

The hands-on exercises in this chapter introduce you to the mechanics of generating and simulating HDL code, using the same model to generate code in both environments. In a series of steps, you will

- Configure a simple model for code generation.
- Generate VHDL code from a subsystem of the model.
- Generate a VHDL test bench and scripts for the Mentor Graphics ModelSim simulator to drive a simulation of the model.
- Compile and execute the model and test bench code in the simulator.
- Generate and simulate Verilog code from the same model.
- Check a model for compatibility with the coder.

The `sfir_fixed` Demo Model

These exercises use the `sfir_fixed` demo model as a source model for HDL code generation. The model simulates a symmetric finite impulse response (FIR) filter algorithm, implemented with fixed-point arithmetic. The following figure shows the top level of the model.



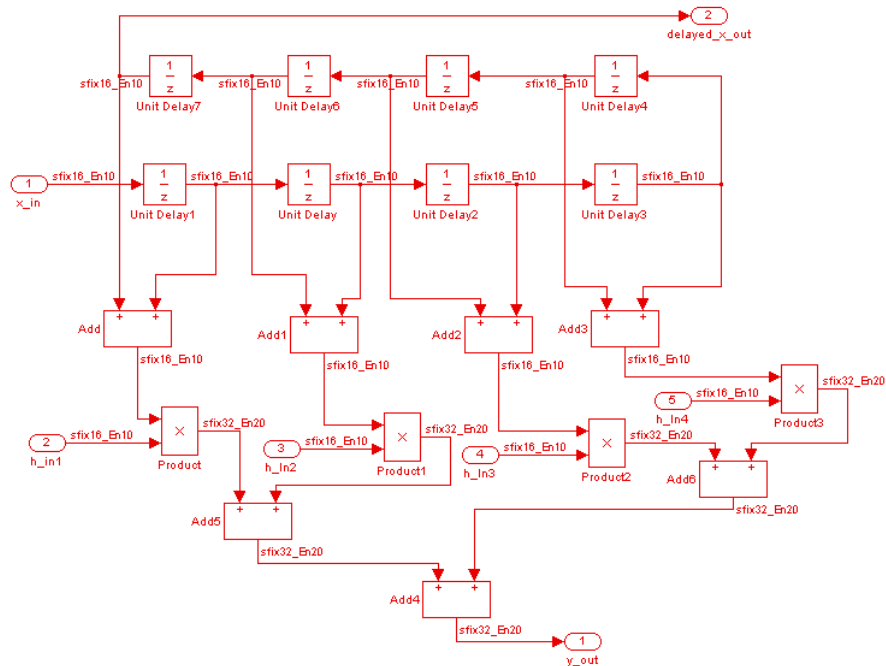
This model employs a division of labor that is useful in HDL design:

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity will be generated, tested, and eventually synthesized from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients.

The Scope blocks are used in simulation only. They are virtual blocks, and do not generate any HDL code.

The following figure shows the `symmetric_fir` subsystem.



Appropriate fixed-point data types propagate throughout the subsystem. Inputs inherit the data types of the signals presented to them. Where required, internal rules of the blocks determine the correct output data type, given the input data types and the operation performed (for example, the Product blocks output 32-bit signals).

The filter outputs a 32-bit fixed-point result at the `y_out` port, and also replicates its input (after passing it through several delay stages) at the `delayed_x_out` port.

In the exercises that follow, you generate VHDL code that implements the `symmetric_fir` subsystem as an entity. You then generate a test bench from the top-level model. The test bench drives the generated entity, for the required number of clock steps, with stimulus data generated from the Signal From Workspace block.

Generating HDL Code Using the Command Line Interface

In this section...

“Overview” on page 2-7

“Creating a Directory and Local Model File” on page 2-7

“Initializing Model Parameters with hdlsetup” on page 2-8

“Generating a VHDL Entity from a Subsystem” on page 2-10

“Generating VHDL Test Bench Code” on page 2-12

“Verifying Generated Code” on page 2-13

“Generating a Verilog Module and Test Bench” on page 2-14

Overview

This exercise provides a step-by-step introduction to code and test bench generation commands, their arguments, and the files created by the code generator. The exercise assumes that you have familiarized yourself with the demo model (see “The `s1r_fixed` Demo Model” on page 2-4).

Creating a Directory and Local Model File

Make a local copy of the demo model and store it in a working directory, as follows.

- 1 Start the MATLAB software.
- 2 Create a directory named `s1_hdlcoder_work`, for example:

```
mkdir C:\work\s1_hdlcoder_work
```

The `s1_hdlcoder_work` directory will store a local copy of the demo model and to store directories and code generated by the coder. The location of the directory does not matter, except that it should not be within the MATLAB directory tree.

- 3 Make the `s1_hdlcoder_work` directory your working directory, for example:

```
cd C:\work\s1_hdlcoder_work
```

- 4 To open the demo model, type the following command at the MATLAB prompt:

```
demods
```

- 5 The **Help** window opens. In the **Demos** pane on the left, click the + for **Simulink**. Then click the + for **Simulink HDL Coder**. Then double-click the list entry for the Symmetric FIR Filter Demo.

The `sfir_fixed` model opens.

- 6 Select **Save As** from the Simulink **File** menu and save a local copy of `sfir_fixed.mdl` to your working directory.
- 7 Leave the `sfir_fixed` model open and proceed to the next section.

Initializing Model Parameters with `hdlsetup`

Before generating code, you must set some parameters of the model. Rather than doing this manually, use the M-file utility, `hdlsetup.m`. The `hdlsetup` command uses the `set_param` function to set up models for HDL code generation quickly and consistently.

To set the model parameters:

- 1 At the MATLAB command prompt, type

```
hdlsetup('sfir_fixed')
```

- 2 Select **Save** from the **File** menu, to save the model with its new settings.

Before continuing with code generation, consider the settings that `hdlsetup` applies to the model.

`hdlsetup` configures the **Solver** options that are recommended or required by the coder. These are

- **Type:** Fixed-step. (The coder currently supports variable-step solvers under limited conditions. See `hdlsetup`.)

- **Solver:** Discrete (no continuous states). Other fixed-step solvers could be selected, but this option is usually the correct 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.

`hdlsetup` also configures the model start and stop times and fixed-step size as follows:

- **Start Time:** 0.0 s
- **Stop Time:** 10 s
- **Fixed step size (fundamental periodic sample time):** auto

If **Fixed step size** is set to auto the step size is chosen automatically, based on the sample times specified in the model. In the demo model, only the Signal From Workspace block specifies an explicit sample time (1 s); all other blocks inherit this sample time.

The model start and stop times determine the total simulation time. This in turn determines the size of data arrays that are generated to provide stimulus and output data for generated test benches. For the demo model, computation of 10 seconds of test data does not take a significant amount of time. Computation of sample values for more complex models can be time consuming. In such cases, you may want to decrease the total simulation time.

The remaining parameters set by `hdlsetup` affect error severity levels, data logging, and model display options. If you want to view the complete set of model parameters affected by `hdlsetup`, open `hdlsetup.m` in the MATLAB Editor.

The model parameter settings provided by `hdlsetup` are intended as useful defaults, but they may not be appropriate for all your applications. For example, `hdlsetup` sets a default **Simulation stop time** of 10 s. A total simulation time of 1000 s would be more realistic for a test of the `sfir_fixed` demo model. If you would like to change the simulation time, enter the desired value into the **Simulation stop time** field of the Simulink window.

See the “Model Parameters” table in the “Model and Block Parameters” section of the Simulink documentation for a summary of user-settable model parameters.

Generating a VHDL Entity from a Subsystem

In this section, you will use the `makehdl` function to generate code for a VHDL entity from the `symmetric_fir` subsystem of the demo model. `makehdl` also generates script files for third-party HDL simulation and synthesis tools.

`makehdl` lets you specify numerous properties that control various features of the generated code. In this example, you will use defaults for all `makehdl` properties.

Before generating code, make sure that you have completed the steps described in “Creating a Directory and Local Model File” on page 2-7 and “Initializing Model Parameters with `hdlsetup`” on page 2-8.

To generate code:

- 1** Select **Current Directory** from the **Desktop** menu in the MATLAB window. This displays the MATLAB Current Directory browser, which lets you easily access your working directory and the files that will be generated within it.
- 2** At the MATLAB prompt, type the command

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

This command directs the coder to generate code from the `symmetric_fir` subsystem within the `sfir_fixed` model, using default values for all properties.

- 3** As code generation proceeds, the coder displays progress messages. The process should complete successfully with the message

```
### HDL Code Generation Complete.
```

Observe that the names of generated files in the progress messages are hyperlinked. After code generation completes, you can click these hyperlinks to view the files in the MATLAB Editor.

`makehdl` compiles the model before generating code. Depending on model display options (such as port data types, etc.), the appearance of the model may change after code generation.

- 4 By default, `makehdl` generates VHDL code. Code files and scripts are written to a *target directory*. The default target directory is a subdirectory of your working directory, named `hdlsrc`.

A folder icon for the `hdlsrc` directory is now visible in the Current Directory browser. To view generated code and script files, double-click the `hdlsrc` folder icon.

- 5 The files that `makehdl` has generated in the `hdlsrc` directory are
 - `symmetric_fir.vhd`: VHDL code. This file contains an entity definition and RTL architecture implementing the `symmetric_fir` filter.
 - `symmetric_fir_compile.do`: Mentor Graphics ModelSim compilation script (`vcom` command) to compile the generated VHDL code.
 - `symmetric_fir_synplify.tcl`: Synplify synthesis script
 - `symmetric_fir_map.txt`: Mapping file. This report file maps generated entities (or modules) to the subsystems that generated them (see “Code Tracing Using the Mapping File” on page 9-25).
- 6 To view the generated VHDL code in the MATLAB Editor, double-click the `symmetric_fir.vhd` file icon in the Current Directory browser.

At this point it is suggested that you study the ENTITY and ARCHITECTURE definitions while referring to “HDL Code Generation Defaults” on page 17-23 in the `makehdl` reference documentation. The reference documentation describes the default naming conventions and correspondences between the elements of a model (subsystems, ports, signals, etc.) and elements of generated HDL code.

- 7 Before proceeding to the next section, close any files you have opened in the editor. Then, click the Go Up One Level button in the Current Directory browser, to set the current directory back to your `s1_hdlcoder_work` directory.
- 8 Leave the `sfir_fixed` model open and proceed to the next section.

Generating VHDL Test Bench Code

In this section, you use the test bench generation function, `makehdltb`, to generate a VHDL test bench. The test bench is designed to drive and verify the operation of the `symmetric_fir` entity that was generated in the previous section. A generated test bench includes

- Stimulus data generated by signal sources connected to the entity under test.
- Output data generated by the entity under test. During a test bench run, this data is compared to the outputs of the VHDL model, for verification purposes.
- Clock, reset, and clock enable inputs to drive the entity under test.
- A component instantiation of the entity under test.
- Code to drive the entity under test and compare its outputs to the expected data.

In addition, `makehdltb` generates Mentor Graphics ModelSim scripts to compile and execute the test bench.

This exercise assumes that your working directory is the same as that used in the previous section. This directory now contains an `hdlsrc` folder containing the previously generated code.

To generate a test bench:

- 1** At the MATLAB prompt, type the command

```
makehdltb('sfir_fixed/symmetric_fir')
```

This command generates a test bench that is designed to interface to and validate code generated from `symmetric_fir` (or from a subsystem with a functionally identical interface). By default, VHDL test bench code, as well as scripts, are generated in the `hdlsrc` target directory.

- 2** As test bench generation proceeds, the coder displays progress messages. The process should complete successfully with the message

```
### HDL TestBench Generation Complete.
```


- 3** To view generated test bench and script files, double-click the `hdlsrc` folder icon in the Current Directory browser. Alternatively, you can click the hyperlinked names of generated files in the code test bench generation progress messages.

The files generated by `makehdltb` are:

- `symmetric_fir_tb.vhd`: VHDL test bench code and generated test and output data.
 - `symmetric_fir_tb_compile.do`: Mentor Graphics ModelSim compilation script (vcom commands). This script compiles and loads both the entity to be tested (`symmetric_fir.vhd`) and the test bench code (`symmetric_fir_tb.vhd`).
 - `symmetric_fir_tb_sim.do`: Mentor Graphics ModelSim script to initialize the simulator, set up **wave** window signal displays, and run a simulation.
- 4** If you want to view the generated test bench code in the MATLAB Editor, double-click the `symmetric_fir.vhd` file icon in the Current Directory browser. You may want to study the code while referring to the `makehdltb` reference documentation, which describes the default actions of the test bench generator.
 - 5** Before proceeding to the next section, close any files you have opened in the editor. Then, click the Go Up One Level button in the Current Directory browser, to set the current directory back to your `s1_hdlcoder_work` directory.

Verifying Generated Code

You can now take the previously generated code and test bench to an HDL simulator for simulated execution and verification of results. See “Simulating and Verifying Generated HDL Code” on page 2-32 for an example of how to use generated test bench and script files with the Mentor Graphics ModelSim simulator.

Generating a Verilog Module and Test Bench

The procedures for generating Verilog code differ only slightly from those for generating VHDL code. This section provides an overview of the command syntax and the generated files.

Generating a Verilog Module

By default, `makehdl` generates VHDL code. To override the default and generate Verilog code, you must pass in a property/value pair to `makehdl`, setting the `TargetLanguage` property to `'verilog'`, as in this example.

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

The previous command generates Verilog source code, as well as scripts for the simulation and the synthesis tools, in the default target directory, `hdlsrc`.

The files generated by this example command are:

- `symmetric_fir.v`: Verilog code. This file contains a Verilog module implementing the `symmetric_fir` subsystem.
- `symmetric_fir_compile.do`: Mentor Graphics ModelSim compilation script (`vlog` command) to compile the generated Verilog code.
- `symmetric_fir_synplify.tcl`: Synplify synthesis script.
- `symmetric_fir_map.txt`: Mapping file. This report file maps generated entities (or modules) to the subsystems that generated them (see “Code Tracing Using the Mapping File” on page 9-25).

Generating and Executing a Verilog Test Bench

The `makehdltb` syntax for overriding the target language is exactly the same as that for `makehdl`. The following example generates Verilog test bench code to drive the Verilog module, `symmetric_fir`, in the default target directory.

```
makehdltb('sfir_fixed/symmetric_fir','TargetLanguage','verilog')
```

The files generated by this example command are

- `symmetric_fir_tb.v`: Verilog test bench code and generated test and output data.
- `symmetric_fir_tb_compile.do`: Mentor Graphics ModelSim compilation script (vlog commands). This script compiles and loads both the entity to be tested (`symmetric_fir.v`) and the test bench code (`symmetric_fir_tb.v`).
- `symmetric_fir_tb_sim.do`: Mentor Graphics ModelSim script to initialize the simulator, set up **wave** window signal displays, and run a simulation.

The following listing shows the commands and responses from a test bench session using the generated scripts:

```
ModelSim>vlib work
ModelSim> do symmetric_fir_tb_compile.do
# Model Technology ModelSim SE vlog 6.0 Compiler 2004.08 Aug 19 2004
# -- Compiling module symmetric_fir
#
# Top level modules:
# symmetric_fir
# Model Technology ModelSim SE vlog 6.0 Compiler 2004.08 Aug 19 2004
# -- Compiling module symmetric_fir_tb
#
# Top level modules:
# symmetric_fir_tb
ModelSim>do symmetric_fir_tb_sim.do
# vsim work.symmetric_fir_tb
# Loading work.symmetric_fir_tb
# Loading work.symmetric_fir
# **** Test Complete. ****
# Break at
C:/work/sl_hdlcoder_work/vlog_code/symmetric_fir_tb.v line 142
# Simulation Breakpoint:Break at
C:/work/sl_hdlcoder_work/vlog_code/symmetric_fir_tb.v line 142
# MACRO ./symmetric_fir_tb_sim.do PAUSED at line 14
```

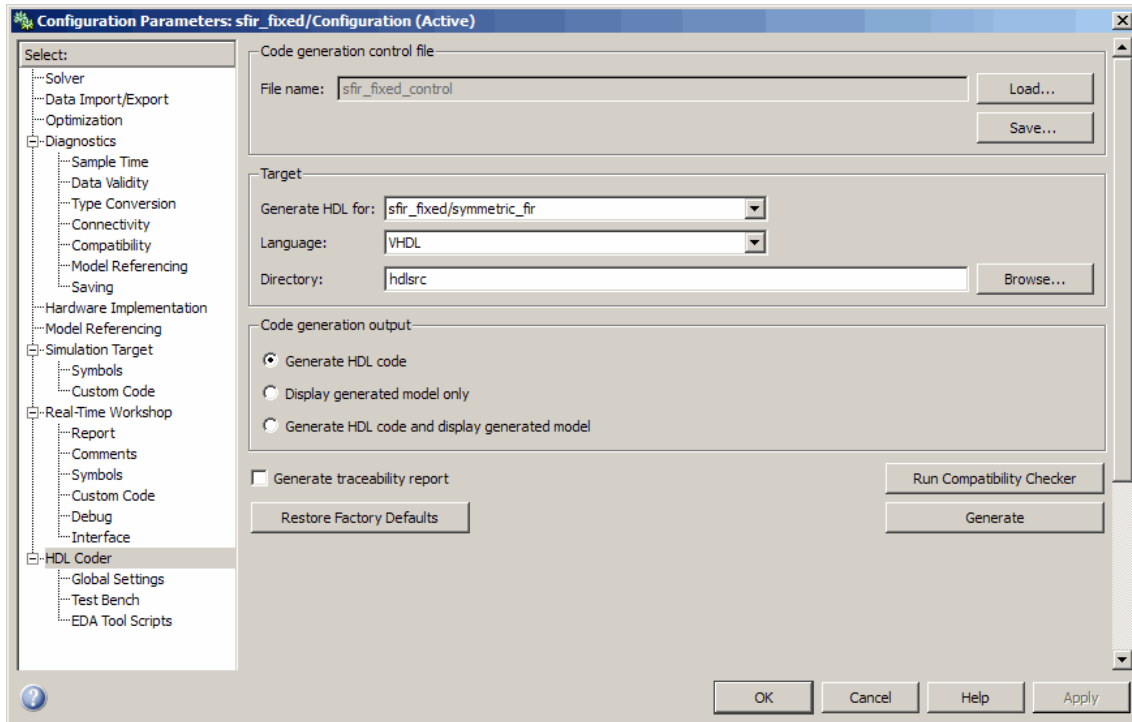
Generating HDL Code Using the GUI

In this section...
“Simulink® HDL Coder GUI Overview” on page 2-16
“Creating a Directory and Local Model File” on page 2-19
“Viewing Coder Options in the Configuration Parameters Dialog Box” on page 2-20
“Creating a Control File” on page 2-22
“Initializing Model Parameters with hdlsetup” on page 2-24
“Selecting and Checking a Subsystem for HDL Compatibility” on page 2-26
“Generating VHDL Code” on page 2-27
“Generating VHDL Test Bench Code” on page 2-30
“Verifying Generated Code” on page 2-31
“Generating Verilog Model and Test Bench Code” on page 2-31

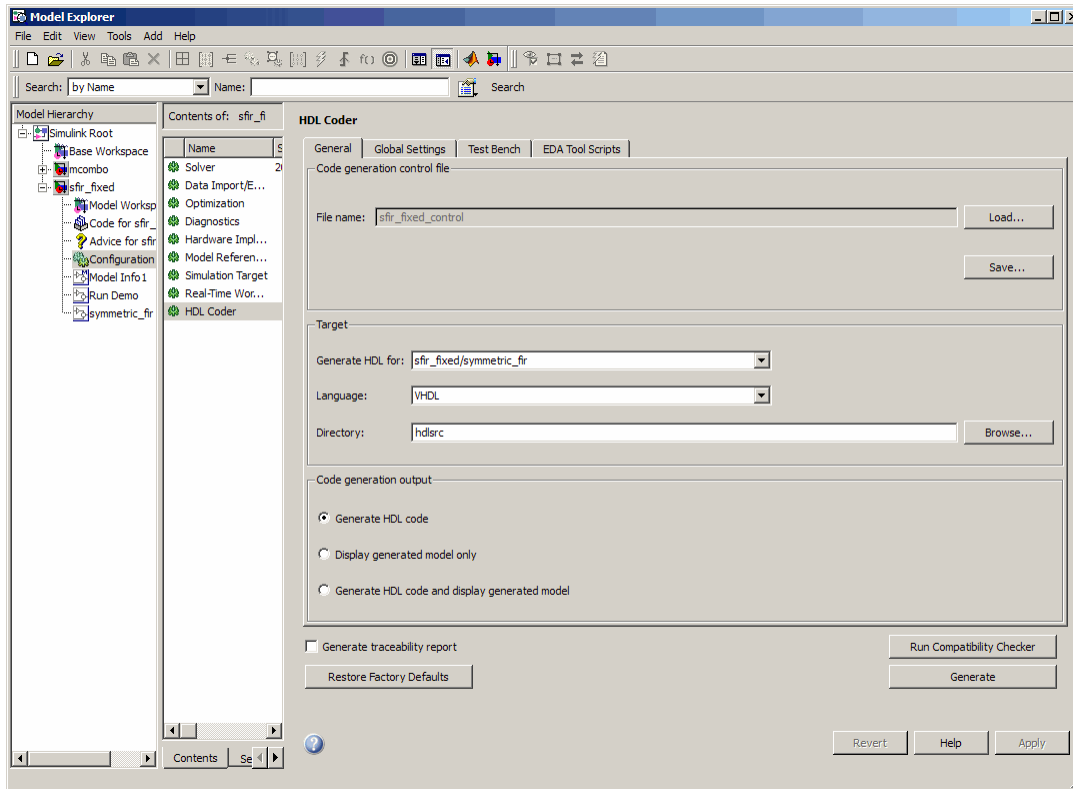
Simulink HDL Coder GUI Overview

You can view and edit options and parameters that affect HDL code generation in the Configuration Parameters dialog box, or in the Model Explorer.

The following figure shows the top-level **HDL Coder** options pane as displayed in the Configuration Parameters dialog box.



The following figure shows the top-level **HDL Coder** options pane as displayed in the Model Explorer.



If you are not familiar with Simulink configuration sets and how to view and edit them in the Configuration Parameters dialog box, see the following documentation:

- “Configuration Sets”
- “Configuration Parameters Dialog Box”

If you are not familiar with the Model Explorer, see “Exploring, Searching, and Browsing Models”.

In the hands-on code generation exercises that follow, you will use the Configuration Parameters dialog box to view and set the coder options and controls. The exercises use the `sfir_fixed` demo model (see “The `sfir_fixed` Demo Model” on page 2-4) in basic code generation and verification steps.

Creating a Directory and Local Model File

In this section you will setup the directory and a local copy of the demo model.

Creating a Directory

Start by setting up a working directory:

- 1 Start the MATLAB software.
- 2 Create a directory named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

You will use `sl_hdlcoder_work` to store a local copy of the demo model and to store directories and code generated by the coder. The location of the directory does not matter, except that it should not be within the MATLAB directory tree.

- 3 Make the `sl_hdlcoder_work` directory your working directory, for example:

```
cd C:\work\sl_hdlcoder_work
```

Making a Local Copy of the Model File

Next, make a copy of the `sfir_fixed` demo model:

- 1 To open the demo model, type the following command at the MATLAB prompt:

```
demოს
```

The **Help** window opens.

- 2 In the **Demos** pane on the left, click the + for **Simulink**. Then click the + for **Simulink HDL Coder**. Then double-click the list entry for the Symmetric FIR Filter demo.

The `sfir_fixed` model opens.

- 3 Select **Save As** from the **File** menu and save a local copy of `sfir_fixed.mdl` to your working directory.

4 Leave the `sfir_fixed` model open and proceed to the next section.

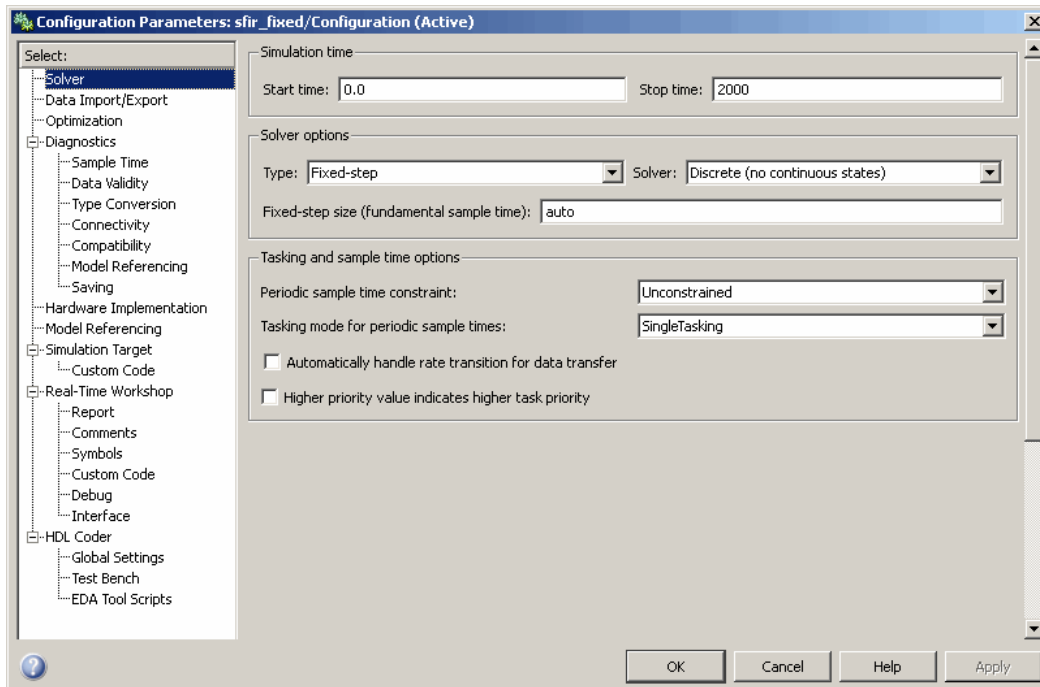
Viewing Coder Options in the Configuration Parameters Dialog Box

The coder option settings are displayed as a category of the model's active configuration set. You can view and edit these options in the Configuration Parameters dialog box, or in the Model Explorer. This discussion uses the Configuration Parameters dialog box.

To access the coder settings:

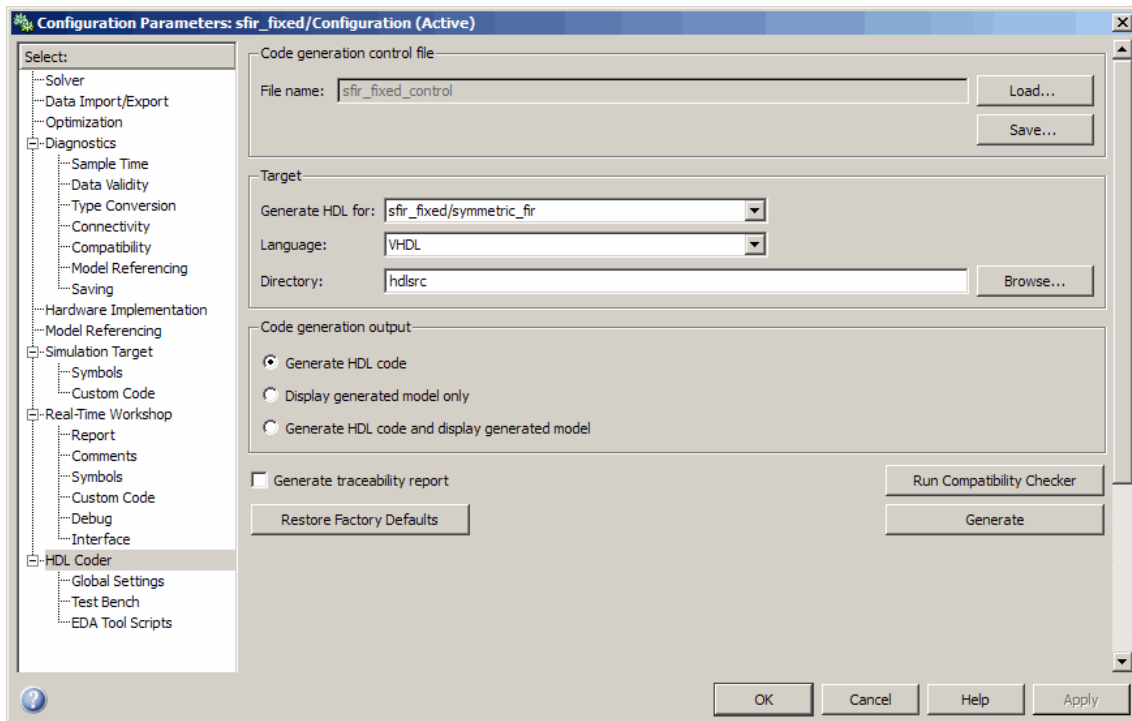
- 1 Select **Configuration Parameters** from the **Simulation** menu in the `sfir_fixed` model window.

The Configuration Parameters dialog box opens with the **Solver** options pane displayed, as shown in the following figure.



2 Observe that the **Select** tree in the left pane of the dialog box includes an **HDL Coder** category, as shown.

3 Click the **HDL Coder** category in the **Select** tree. The **HDL Coder** pane is displayed, as shown in the following figure.



The **HDL Coder** pane contains top-level options and buttons that control the HDL code generation process. Several other categories of options are available under the **HDL Coder** entry in the **Select** tree. This exercise uses a small subset of these options, leaving the others at their default settings.

Chapter 3, “Code Generation Options in the Simulink® HDL Coder GUT” summarizes all the options available in the **HDL Coder** category.

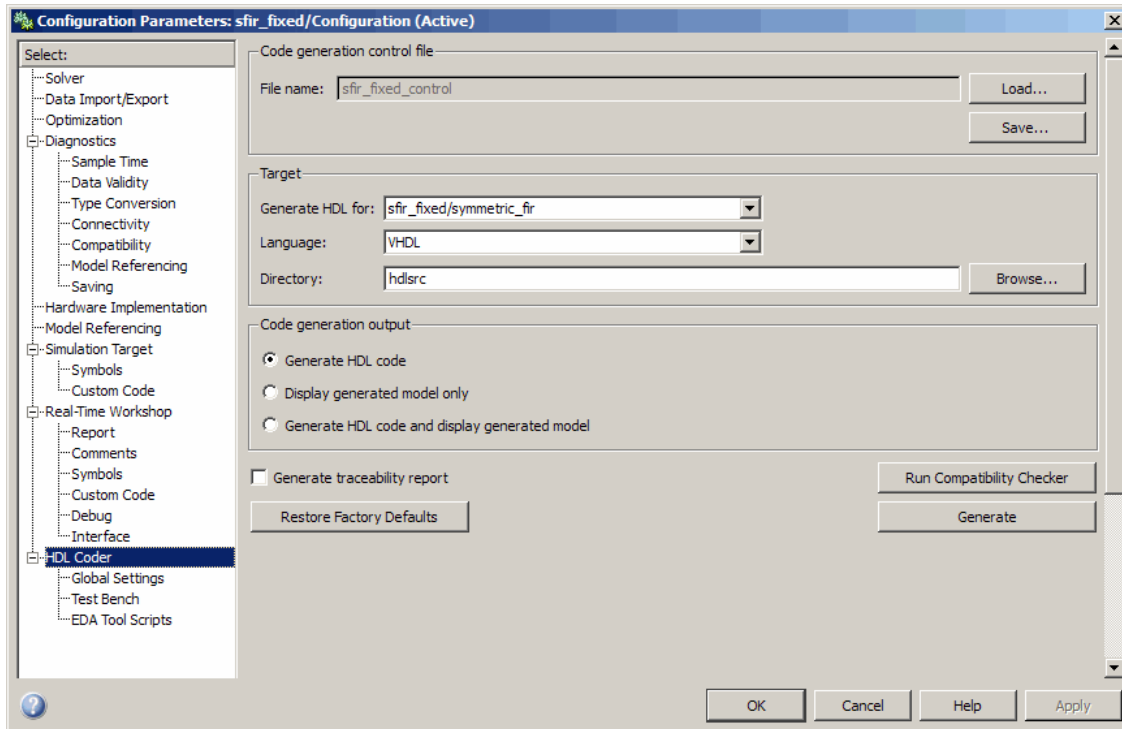
Creating a Control File

Code generation control files (referred to in this document as *control files*) let you

- Save your model's HDL code generation options.
- Extend the HDL code generation process and direct its details.

A control file is an M-file that you attach to your model, using either the `makehdl` command or the Configuration Parameters dialog box. In this tutorial, you will use a control file to save HDL code generation options. This is a required step with most models, because HDL code generation settings are not saved in the `.mdl` file like other components of a model's configuration set. If you want your HDL code generation settings to persist across sessions with a model, you must save your current settings to a control file. The control file is then linked to the model, and the linkage is preserved when you save the model.

When a control file is linked to a model, the control file name is displayed in the **File name** field of the top-level **HDL Coder** options pane. The `sfir_fixed` demo model is attached to the control file `sfir_fixed_control.m`. This control file is stored within the MATLAB demo directories and should not be overwritten. For use in this tutorial, you will save the current HDL code generation options to a new control file in the working directory. Later in the tutorial, you will change some options and save them to the control file.



To save the current HDL code generation options to a new control file:

- 1 Open the Configuration Parameters dialog box and select the **HDL Coder** options pane.
- 2 Under **Code generation control file**, click the **Save** button. A standard file dialog box opens.
- 3 Navigate to your current working directory and save the file as `sfir_fixed_control.m`.
- 4 Select **Save** from the **File** menu. When you save the model, the control file linkage information is written to the `.mdl` file, and the control file linkage persists in future sessions with your model.

This tutorial uses a control file only as a mechanism for saving HDL code generation settings. This simple application of a control file does not require knowledge of any internal details about the file. You can also use a control file to direct or customize many details of the code generation process. It is strongly recommended that you read Chapter 5, “Code Generation Control Files”, after completing this tutorial.

Initializing Model Parameters with `hdlsetup`

Before generating code, you must set some parameters of the model. Rather than doing this manually, use the M-file utility, `hdlsetup.m`. The `hdlsetup` command uses the `set_param` function to set up models for HDL code generation quickly and consistently.

To set the model parameters:

- 1 At the MATLAB command prompt, type

```
hdlsetup('sfir_fixed')
```

- 2 Select **Save** from the **File** menu, to save the model with its new settings.

You do not need to update the control file at this point, because `hdlsetup` modifies only the model parameters, not the HDL code generation options.

Before continuing with code generation, consider the settings that `hdlsetup` applies to the model.

`hdlsetup` configures **Solver** options that are recommended or required by the coder. These are

- **Type:** Fixed-step. (The coder currently supports variable-step solvers under limited conditions. See `hdlsetup`.)
- **Solver:** Discrete (no continuous states). Other fixed-step solvers could be selected, but this option is usually the correct 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.

`hdlsetup` also configures the model start and stop times and fixed-step size as follows:

- **Start Time:** 0.0 s
- **Stop Time:** 10 s
- **Fixed step size (fundamental periodic sample time):** auto

If **Fixed step size** is set to auto the step size is chosen automatically, based on the sample times specified in the model. In the demo model, only the Signal From Workspace block specifies an explicit sample time (1 s); all other blocks inherit this sample time.

The model start and stop times determine the total simulation time. This in turn determines the size of data arrays that are generated to provide stimulus and output data for generated test benches. For the demo model, computation of 10 seconds of test data does not take a significant amount of time. Computation of sample values for more complex models can be time consuming. In such cases, you may want to decrease the total simulation time.

The remaining parameters set by `hdlsetup` affect error severity levels, data logging, and model display options. If you want to view the complete set of model parameters affected by `hdlsetup`, open `hdlsetup.m` in the MATLAB Editor.

The model parameter settings provided by `hdlsetup` are intended as useful defaults, but they may not be appropriate for all your applications. For example, `hdlsetup` sets a default **Simulation stop time** of 10 s. A total simulation time of 1000 s would be more realistic for a test of the `sfir_fixed` demo model. If you would like to change the simulation time, enter the desired value into the **Simulation stop time** field of the Simulink window.

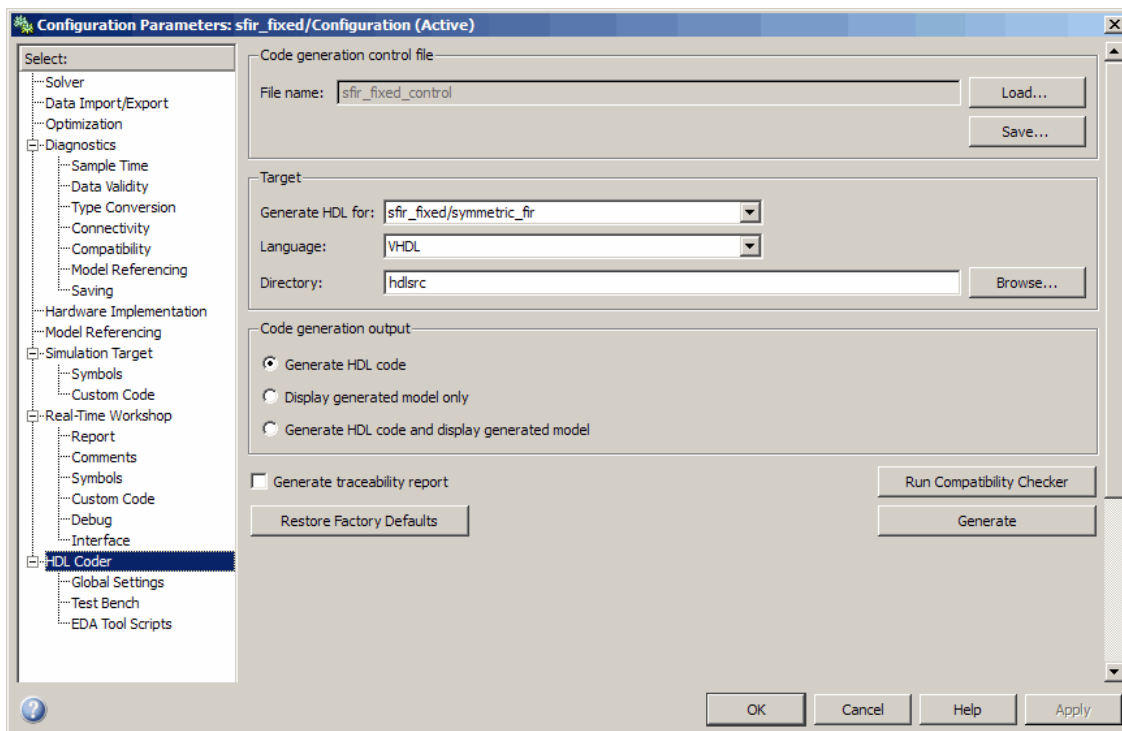
See the “Model Parameters” table in the “Model and Block Parameters” section of the Simulink documentation for a summary of user-settable model parameters.

Selecting and Checking a Subsystem for HDL Compatibility

The coder generates code from either the current model or from a subsystem at the root level of the current model. You use the **Generate HDL for** menu to select the model or subsystem from which code is to be generated. Each entry in the menu shows the full path to the model or one of its subcomponents.

The `sfir_fixed` demo model is configured with the `sfixed_fir/symmetric_fir` subsystem selected for code generation. If this is not the case, make sure that the `symmetric_fir` subsystem is selected for code generation, as follows:

- 1 Select `sfixed_fir/symmetric_fir` from the **Generate HDL for** menu.
- 2 Click **Apply**. The dialog box should now appear as shown in the following figure.

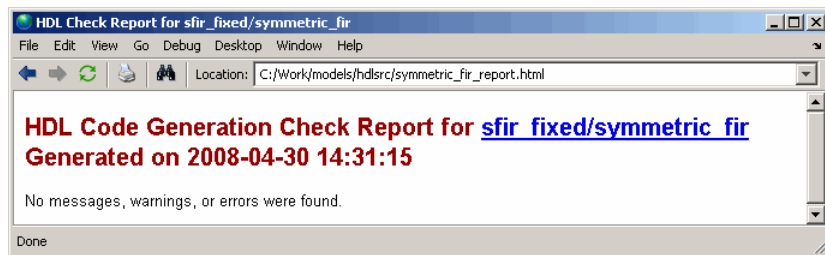


To check HDL compatibility for the subsystem:

- 1 Click the **Run Compatibility Checker** button.
- 2 The HDL compatibility checker examines the system selected in the **Generate HDL for** menu for any compatibility problems. In this case, the selected subsystem is fully HDL-compatible, and the compatibility checker displays the following message:

```
### Starting HDL Check.
### HDL Check Complete with 0 errors, warnings and messages.
```

- 3 The compatibility checker also displays an HTML report in a Web browser, as shown in the following figure.

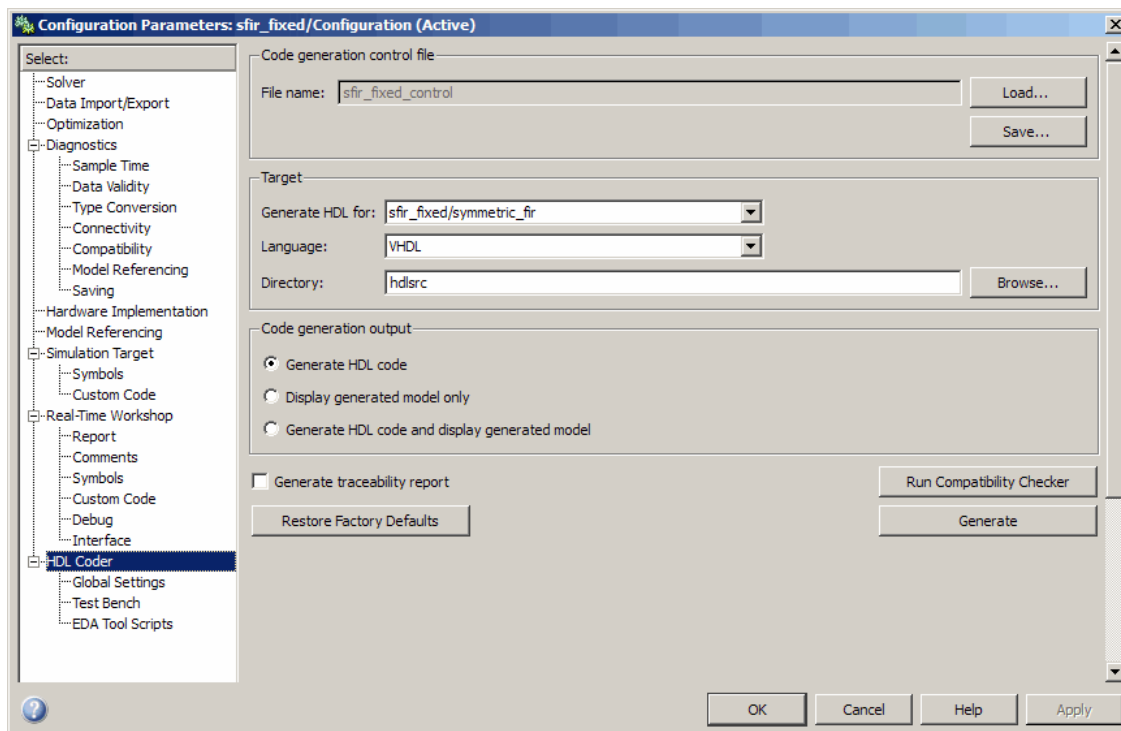


Generating VHDL Code

The top-level **HDL Coder** options are now set as follows:

- The **Generate HDL for** field specifies the `sfixed_fir/symmetric_fir` subsystem for code generation.
- The **Language** field specifies (by default) generation of VHDL code.
- The **Directory** field specifies a *target directory* that stores generated code files and scripts. The default target directory is a subdirectory of your working directory, named `hd1src`.

The following figure shows these settings.



Before generating code, select **Current Directory** from the **Desktop** menu in the MATLAB window. This displays the Current Directory browser, which lets you easily access your working directory and the files that will be generated within it.

To generate code:

- 1 Click the **Generate** button.
- 2 As code generation proceeds, the coder displays progress messages. The process should complete successfully with the message

```
### HDL Code Generation Complete.
```


Observe that the names of generated files in the progress messages are hyperlinked. After code generation completes, you can click these hyperlinks to view the files in the MATLAB Editor.

The coder compiles the model before generating code. Depending on model display options (such as port data types, etc.), the appearance of the model may change after code generation.

- 3** A folder icon for the `hdlsrc` directory is now visible in the Current Directory browser. To view generated code and script files, double-click the `hdlsrc` folder icon.
- 4** The files that were generated in the `hdlsrc` directory are
 - `symmetric_fir.vhd`: VHDL code. This file contains an entity definition and RTL architecture implementing the `symmetric_fir` filter.
 - `symmetric_fir_compile.do`: Mentor Graphics ModelSim compilation script (`vcom` command) to compile the generated VHDL code.
 - `symmetric_fir_synplify.tcl`: Synplify synthesis script.
 - `symmetric_fir_map.txt`: Mapping file. This report file maps generated entities (or modules) to the subsystems that generated them (see “Code Tracing Using the Mapping File” on page 9-25).
- 5** To view the generated VHDL code in the MATLAB Editor, double-click the `symmetric_fir.vhd` file icon in the Current Directory browser.

At this point it is suggested that you study the ENTITY and ARCHITECTURE definitions while referring to “HDL Code Generation Defaults” on page 17-23 in the `makehdl` reference documentation. The reference documentation describes the default naming conventions and correspondences between the elements of a model (subsystems, ports, signals, etc.) and elements of generated HDL code.

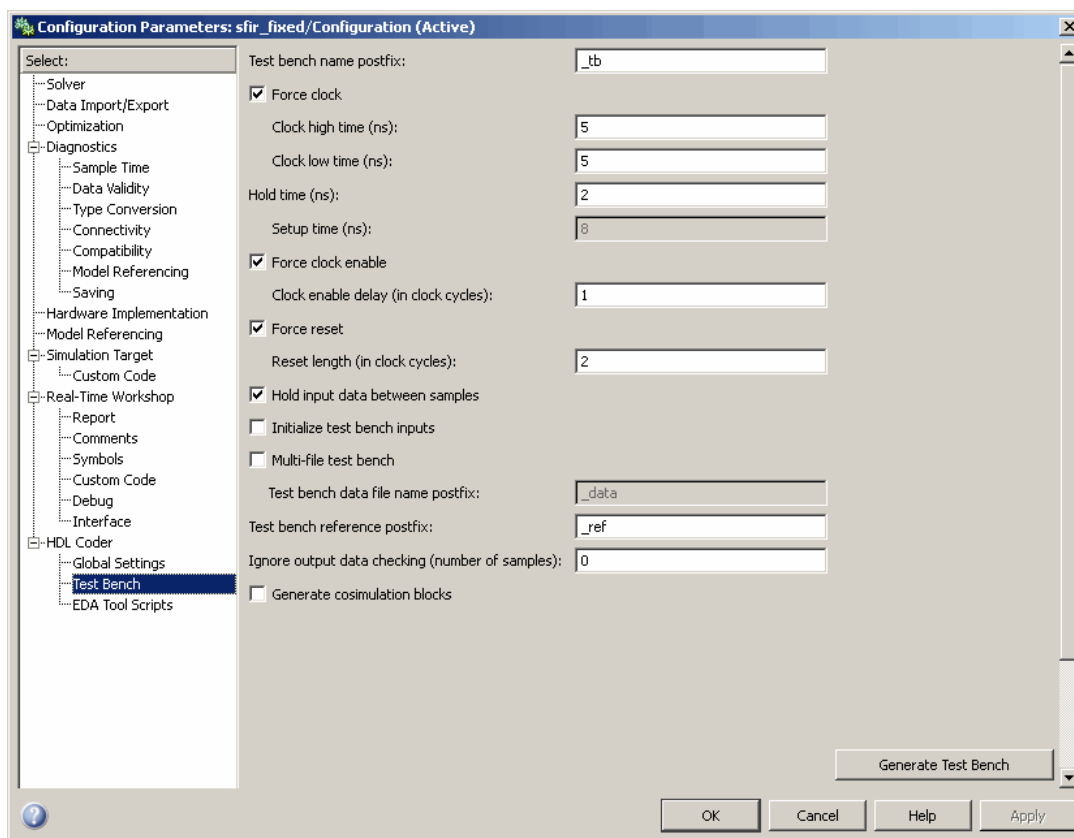
- 6** Before proceeding to the next section, close any files you have opened in the editor. Then, click the Go Up One Level button in the Current Directory browser, to set the current directory back to your `s1_hdlcoder_work` directory.

Generating VHDL Test Bench Code

At this point, the **Generate HDL for**, **Language**, and **Directory** fields are set as they were in the previous section. Accordingly, you can now generate VHDL test bench code to drive the VHDL code generated previously for the `sfixed_fir/symmetric_fir` subsystem. The code will be written to the same target directory as before.

To generate a test bench:

- 1 Click the **Test Bench** entry in the **HDL Coder** list in the **Select** tree. The **Test Bench** pane is displayed, as shown in the following figure.



2 Click the **Generate Test bench** button.

3 As test bench generation proceeds, the coder displays progress messages. The process should complete successfully with the message

```
### HDL TestBench Generation Complete.
```

4 The files that were generated in the `hdlsrc` directory are

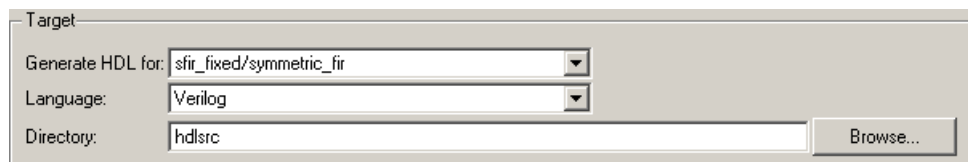
- `symmetric_fir_tb.vhd`: VHDL test bench code and generated test and output data.
- `symmetric_fir_tb_compile.do`: Mentor Graphics ModelSim compilation script (vcom commands). This script compiles and loads both the entity to be tested (`symmetric_fir.vhd`) and the test bench code (`symmetric_fir_tb.vhd`).
- `symmetric_fir_tb_sim.do`: Mentor Graphics ModelSim script to initialize the simulator, set up **wave** window signal displays, and run a simulation.

Verifying Generated Code

You can now take the generated code and test bench to an HDL simulator for simulated execution and verification of results. See “Simulating and Verifying Generated HDL Code” on page 2-32 for an example of how to use generated test bench and script files with the Mentor Graphics ModelSim simulator.

Generating Verilog Model and Test Bench Code

The procedure for generating Verilog code is the same as for generating VHDL code (see “Generating a VHDL Entity from a Subsystem” on page 2-10 and “Generating VHDL Test Bench Code” on page 2-12), except that you should select **Verilog** from the **Language** field of the **HDL Coder** options, as shown in the following figure.



Simulating and Verifying Generated HDL Code

Note This section requires the use of the Mentor Graphics ModelSim simulator.

This section assumes that you have generated code from the `sfir_fixed` demo model as described in either of the following exercises:

- “Generating HDL Code Using the Command Line Interface” on page 2-7
- “Generating HDL Code Using the GUI” on page 2-16

In this section you compile and run a simulation of the previous generated model and test bench code. The scripts generated by the coder let you do this with just a few simple commands. The procedure is the same, whether you generated code in the command line environment or in the GUI.

To run the simulation:

- 1** Start the Mentor Graphics ModelSim software.
- 2** Set the working directory to the directory in which you previously generated code.

```
ModelSim>cd C:/work/sl_hdlcoder_work/hdlsrc
```

- 3** Use the generated compilation script to compile and load the generated model and text bench code. The following listing shows the command and responses.

```
ModelSim>do symmetric_fir_tb_compile.do
# Model Technology ModelSim SE vcom 6.0 Compiler 2004.08 Aug 19 2004
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling entity symmetric_fir
# -- Compiling architecture rtl of symmetric_fir
# Model Technology ModelSim SE vcom 6.0 Compiler 2004.08 Aug 19 2004
# -- Loading package standard
```

```

# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling package symmetric_fir_tb_pkg
# -- Compiling package body symmetric_fir_tb_pkg
# -- Loading package symmetric_fir_tb_pkg
# -- Loading package symmetric_fir_tb_pkg
# -- Compiling entity symmetric_fir_tb
# -- Compiling architecture rtl of symmetric_fir_tb
# -- Loading entity symmetric_fir
    
```

- 4** Use the generated simulation script to execute the simulation. The following listing shows the command and responses. The warning messages are benign.

```

ModelSim>do symmetric_fir_tb_sim.do
# vsim work.symmetric_fir_tb
# Loading C:\Applications\ModelTech_6_0\win32\..\std.standard
# Loading C:\Applications\ModelTech_6_0\win32\..\ieee.std_logic_1164(body)
# Loading C:\Applications\ModelTech_6_0\win32\..\ieee.numeric_std(body)
# Loading work.symmetric_fir_tb_pkg(body)
# Loading work.symmetric_fir_tb(rtl)
# Loading work.symmetric_fir(rtl)
# ** Warning: NUMERIC_STD."<": metavalue detected, returning FALSE
#   Time: 0 ns   Iteration: 0   Instance: /symmetric_fir_tb
.
.
.
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#   Time: 0 ns   Iteration: 1   Instance: /symmetric_fir_tb
# ** Note: *****TEST COMPLETED *****
#   Time: 140 ns   Iteration: 1   Instance: /symmetric_fir_tb
    
```

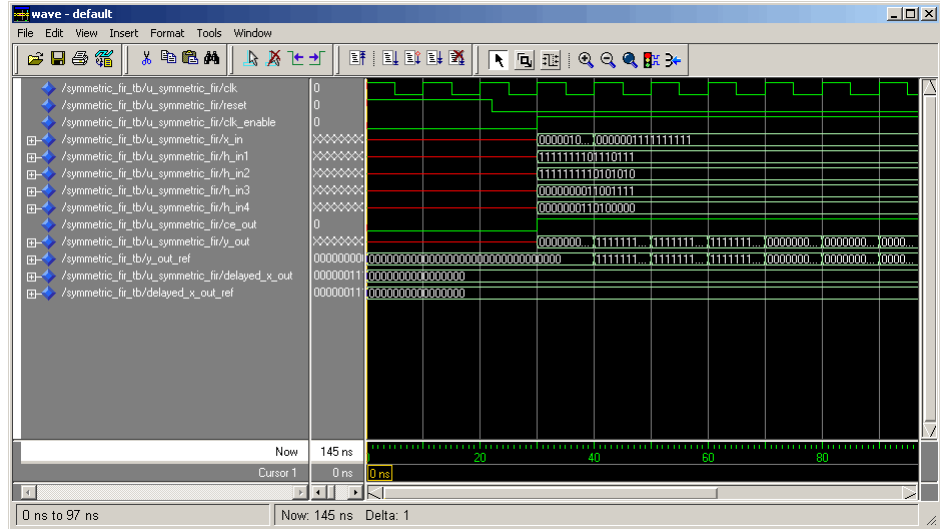
The test bench termination message indicates that the simulation has run to completion successfully, without any comparison errors.

```

# ** Note: *****TEST COMPLETED *****
    
```

- 5** The simulation script displays all inputs and outputs in the model (including the reference signals `y_out_ref` and `delayed_x_out_ref`) in the

Mentor Graphics ModelSim **wave** window. The following figure shows the signals displayed in the **wave** window.



- 6 Exit the Mentor Graphics ModelSim simulator when you finish viewing signals.
- 7 Close any files you have opened in the MATLAB Editor. Then, click the **Go Up One Level** button in the Current Directory browser, to set the current directory back to your `sl_hdlcoder_work` directory.

Code Generation Options in the Simulink HDL Coder GUI

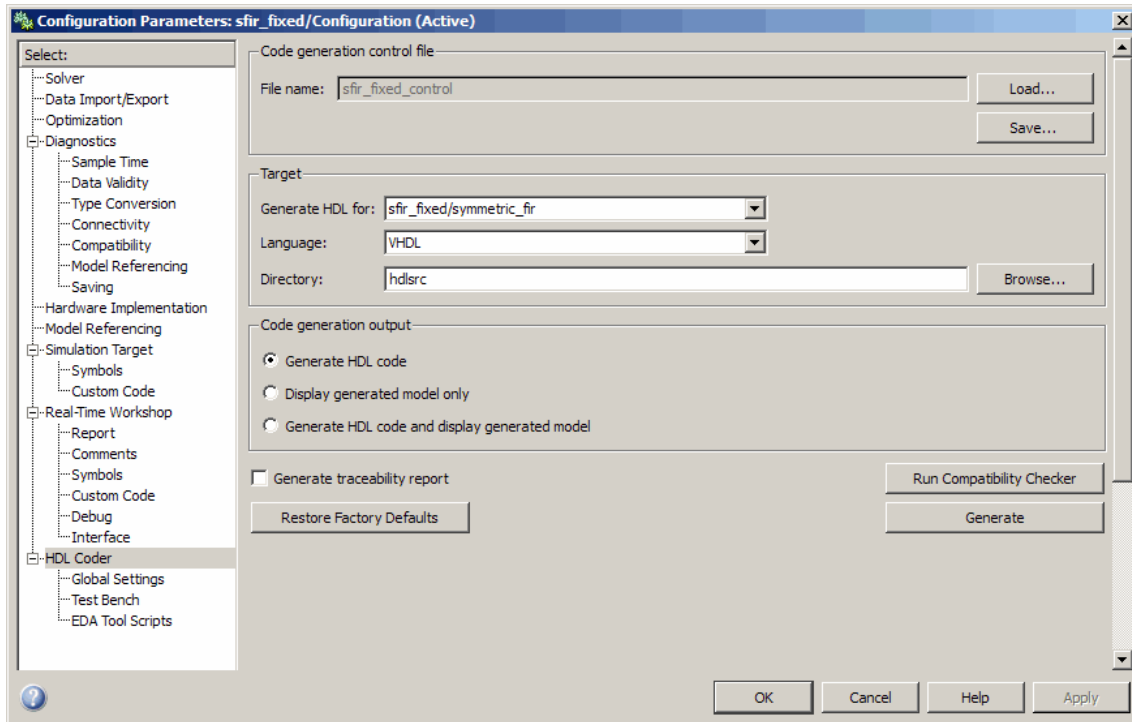
- “Viewing and Setting HDL Coder Options” on page 3-2
- “HDL Coder Pane: General” on page 3-6
- “HDL Coder Pane: Global Settings” on page 3-14
- “HDL Coder Pane: Test Bench” on page 3-50
- “HDL Coder Pane: EDA Tool Scripts” on page 3-74

Viewing and Setting HDL Coder Options

In this section...
“HDL Coder Options in the Configuration Parameters Dialog Box” on page 3-2
“HDL Coder Options in the Model Explorer” on page 3-3
“HDL Coder Menu” on page 3-5

HDL Coder Options in the Configuration Parameters Dialog Box

The following figure shows the top-level **HDL Coder** pane as displayed in the Configuration Parameters dialog box. To open this dialog box, select **Simulation > Configuration Parameters** in the Simulink window. Then select **HDL Coder** from the list on the left.



If you are not familiar with Simulink configuration sets and how to view and edit them in the Configuration Parameters dialog box, see the “Configuration Sets” and “Configuration Parameters Dialog Box” sections of the Simulink documentation.

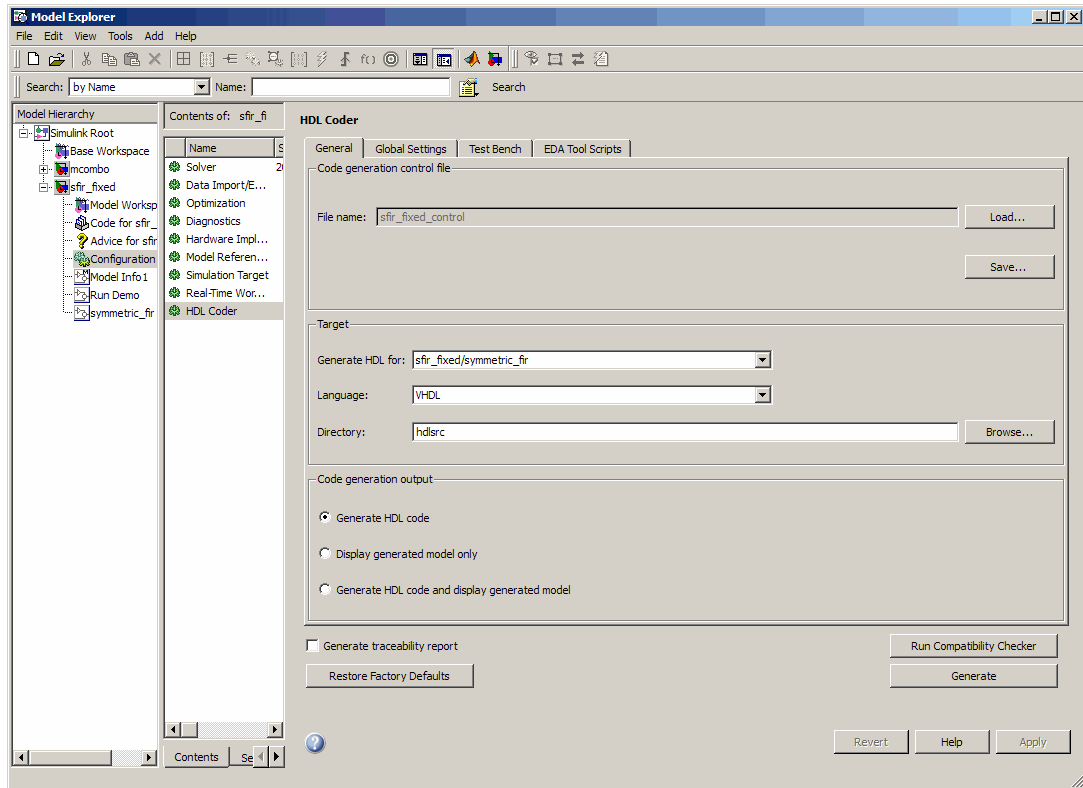
Note When the **HDL Coder** pane of the Configuration Parameters dialog box is displayed, clicking the **Help** button displays general help for the Configuration Parameters dialog box.

HDL Coder Options in the Model Explorer

The following figure shows the top-level **HDL Coder** pane as displayed in the **Dialog** pane of the Model Explorer.

To view this dialog box:

- 1 Select **View > Model Explorer** in the Simulink window.
- 2 Select your model's active configuration set in the **Model Hierarchy** tree on the left.
- 3 Select **HDL Coder** from the list in the **Contents** pane.

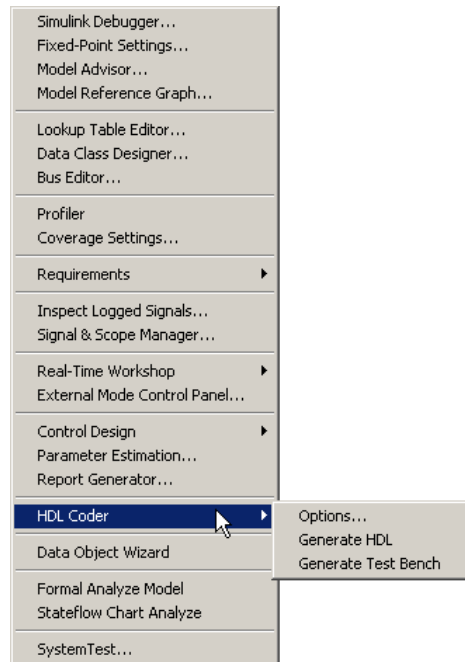


When the **HDL Coder** pane is selected in the Model Explorer, clicking the **Help** button displays the documentation specific to the current tab.

If you are not familiar with the Model Explorer, see “Exploring, Searching, and Browsing Models”.

HDL Coder Menu

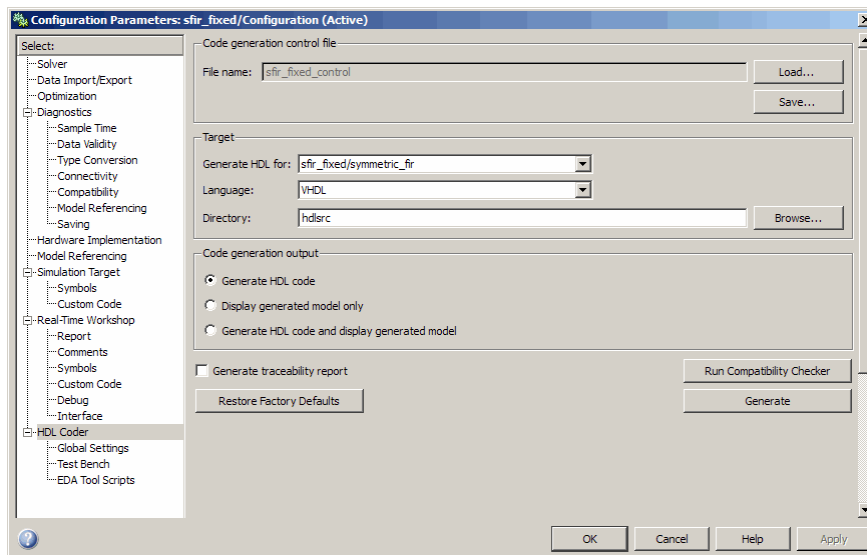
The **HDL Coder** submenu of the **Tools** menu (see the following figure) provides shortcuts to the HDL code generation options. You can also use this menu to initiate code generation.



The **HDL Coder** submenu options are:

- **Options:** Open the **HDL Coder** 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 from the top (root) level of the current model in the **Generate HDL for** menu, the **Generate Test Bench** menu option is disabled.

HDL Coder Pane: General



In this section...

“HDL Coder Top-Level Pane Overview” on page 3-7

“File name” on page 3-8

“Generate HDL for” on page 3-9

“Language” on page 3-10

“Directory” on page 3-11

“Code Generation Output” on page 3-12

“Generate traceability report” on page 3-13

HDL Coder Top-Level Pane Overview

The top-level **HDL Coder** pane contains buttons that initiate code generation and compatibility checking, and sets parameters that affect overall operation of code generation.

Buttons in the HDL Coder Top-Level Pane

The buttons in the **HDL Coder** pane perform important functions related to code generation and control file linkage and maintenance. 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 any compatibility problems. See also `checkhdl`.

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

Load: Opens a standard file selection dialog box so that you can navigate to and select a control file and load it into memory. See also *Using Control Files in the Code Generation Process*.

Save: Opens a standard file save dialog box so that you can save current HDL code generation settings to a specified control file. See also *Using Control Files in the Code Generation Process*.

Restore Factory Defaults: Clears the **File Name** field and unlinks the current control file from the model. See also *Using Control Files in the Code Generation Process*.

File name

Displays the file name of the currently selected control file (if any). This is a display-only field.

Settings

Default: No control file name displayed.

- To select a control file, click **Load**, navigate to the desired control file, and select it. The **File Name** field displays the name of the selected file.
- To clear the **File Name** field and unlink the current control file, click the **Restore Factory Defaults** button.

Command-Line Information

Property: HDLControlFiles

Type: string

Value: Pass in a cell array containing a string that specifies a control file to be attached to the current model.

Default: No control file is specified.

See Also

Using Control Files in the Code Generation Process

Generate HDL for

Select the subsystem or model from which code is generated. The list includes the path to the root model and to all root-level 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

Directory

Enter a path to the directory into which code is generated. Alternatively, click **Browse** to navigate to and select a directory. The selected directory is referred to as the target directory.

Settings

Default: The default target directory is a subdirectory of your working directory, named `hdlsrc`.

Command-Line Information

Property: `TargetDirectory`

Type: `string`

Value: A valid path to your target directory

Default: `'hdlsrc'`

See Also

- `TargetDirectory`
- `makehdl`

Code Generation Output

This option button group contains options related to the creation and display of generated models. Click the desired button to select an option.

Settings

Default: Generate HDL code

- **Generate HDL code:** Generate HDL code without displaying the generated model.
- **Display generated model only:** Display the generated model without generating HDL code.
- **Generate HDL Code and display generated model:** Display the generated model after HDL code generation completes.

Command-Line Information

Property: CodeGenerationOutput

Type: string

Value: 'GenerateHDLCode' |
'GenerateHDLCodeAndDisplayGeneratedModel' |
'DisplayGeneratedModelOnly'

Default: 'GenerateHDLCode'

See Also

Defaults and Options for Generated Models

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

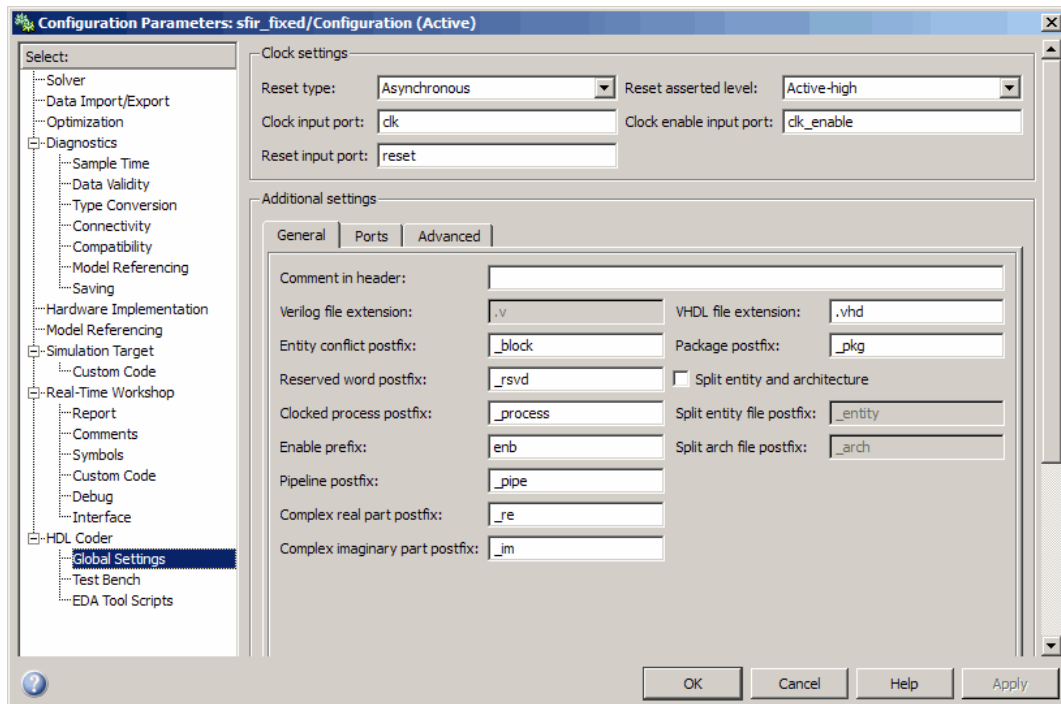
Value: 'on' | 'off'

Default: 'off'

See Also

Traceability

HDL Coder Pane: Global Settings



In this section...

- “Global Settings Overview” on page 3-16
- “Reset type” on page 3-17
- “Reset asserted level” on page 3-18
- “Clock input port” on page 3-19
- “Clock enable input port” on page 3-20
- “Reset input port” on page 3-21
- “Comment in header” on page 3-22
- “Verilog file extension” on page 3-23
- “VHDL file extension” on page 3-24

In this section...

“Entity conflict postfix” on page 3-25

“Package postfix” on page 3-26

“Reserved word postfix” on page 3-27

“Split entity and architecture” on page 3-28

“Split entity file postfix” on page 3-30

“Split arch file postfix” on page 3-31

“Clocked process postfix” on page 3-32

“Enable prefix” on page 3-33

“Pipeline postfix” on page 3-34

“Complex real part postfix” on page 3-35

“Complex imaginary part postfix” on page 3-36

“Input data type” on page 3-37

“Output data type” on page 3-38

“Clock enable output port” on page 3-40

“Represent constant values by aggregates” on page 3-41

“Use "rising_edge" for registers” on page 3-42

“Loop unrolling” on page 3-43

“Cast before sum” on page 3-44

“Use Verilog ``timescale` directives” on page 3-45

“Inline VHDL configuration” on page 3-46

“Concatenate type safe zeros” on page 3-47

“Optimize timing controller” on page 3-48

Global Settings Overview

The **Global Settings** pane lets you set options 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: Any identifier that is legal 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: Any identifier that is legal 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: Any identifier that is legal in the target language

Default: 'reset'

See Also

ResetInputPort

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 as appropriate 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.

Dependencies

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: Any string that is legal 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: Any string value 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_rsrd`.

Command-Line Information

Property: `ReservedWordPostfix`

Type: `string`

Default: `'_rsrd'`

See Also

`ReservedWordPostfix`

Split entity and architecture

Specify whether generated VHDL entity and architecture code is written to a single VHDL file or to separate files.

Settings

Default: Off

- On
VHDL entity and architecture definitions are written to separate files.
- Off
VHDL entity and architecture code is written to a single VHDL file.

Tips

The names of the entity and architecture files derive from the base file name (as specified by the generating model or subsystem name). By default, postfix strings identifying the file as an entity (`_entity`) or architecture (`_arch`) are appended to the base file name. You can override the default and specify your own postfix string.

For example, instead of all generated code residing in `MyFIR.vhd`, you can specify that the code reside in `MyFIR_entity.vhd` and `MyFIR_arch.vhd`.

Dependencies

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

Selecting this option enables the following parameters:

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

Command-Line Information

Property: `SplitEntityArch`
Type: `string`
Value: `'on' | 'off'`

Default: 'off'

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`

Dependencies

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 Timing_Controller
  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'`

Using a control file, you can specify a generation of input and/or output pipeline registers for selected blocks. The coder appends the string specified by the **Pipeline postfix** option when generating code for such 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.

Dependencies

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.

Dependencies

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`

Represent constant values by aggregates

Specify whether all constants in VHDL code are represented by aggregates, including constants that are less than 32 bits.

Settings

Default: Off

On

The coder represents all 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.

Tips

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.

Dependencies

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

Cast before sum

Specify whether operands in addition and subtraction operations are type cast to the result type before executing the operation.

Settings

Default: On

- On
Typecast input values in addition and subtraction operations to the result type before operating on the values.
- Off
Preserve the types of input values during addition and subtraction operations and then convert the result to the result type.

Command-Line Information

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

See Also

CastBeforeSum

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.

Dependencies

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 any file that instantiates 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

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 (`Timing_Controller.vhd` or `Timing_Controller.v`) 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.

Command-Line Information

Property: `OptimizeTimingController`

Type: `string`

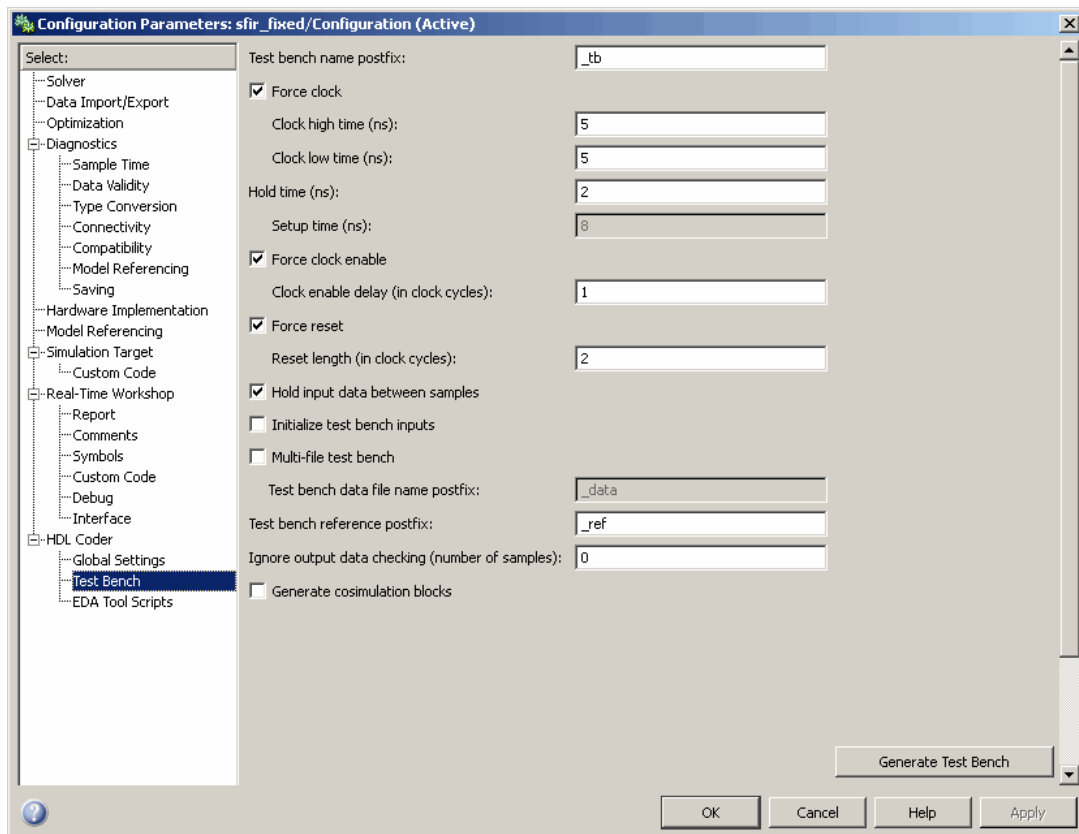
Value: `'on' | 'off'`

Default: `'on'`

See Also

`OptimizeTimingController`

HDL Coder Pane: Test Bench



In this section...

- “Test Bench Overview” on page 3-52
- “Test bench name postfix” on page 3-53
- “Force clock” on page 3-54
- “Clock high time (ns)” on page 3-55
- “Clock low time (ns)” on page 3-56
- “Hold time (ns)” on page 3-57

In this section...

“Setup time (ns)” on page 3-58

“Force clock enable” on page 3-59

“Clock enable delay (in clock cycles)” on page 3-60

“Force reset” on page 3-62

“Reset length (in clock cycles)” on page 3-63

“Hold input data between samples” on page 3-65

“Initialize test bench inputs” on page 3-66

“Multi-file test bench” on page 3-67

“Test bench reference postfix” on page 3-69

“Test bench data file name postfix” on page 3-70

“Ignore output data checking (number of samples)” on page 3-71

“Generate cosimulation blocks” on page 3-73

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`.

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.

Dependencies

This parameter is enabled when **Force clock** is selected.

Command-Line Information

Property: ClockHighTime

Type: integer or double (with a maximum of 6 significant digits after the decimal point)

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.

Dependencies

This parameter is enabled when **Force clock** is selected.

Command-Line Information

Property: ClockLowTime

Type: integer or double (with a maximum of 6 significant digits after the decimal point)

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 or double (with a maximum of 6 significant digits after the decimal point)

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, there is no corresponding command-line property.

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

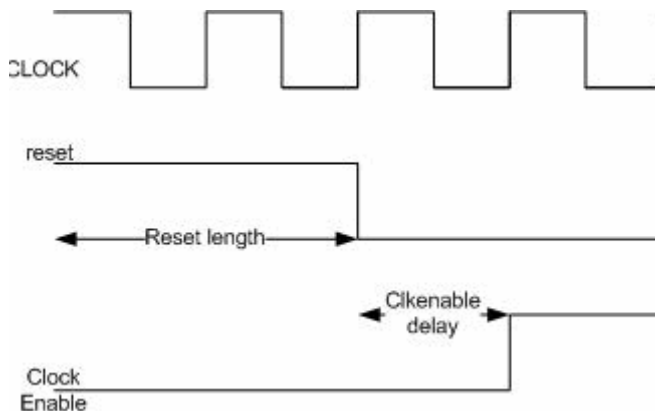
Clock enable delay (in clock cycles)

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

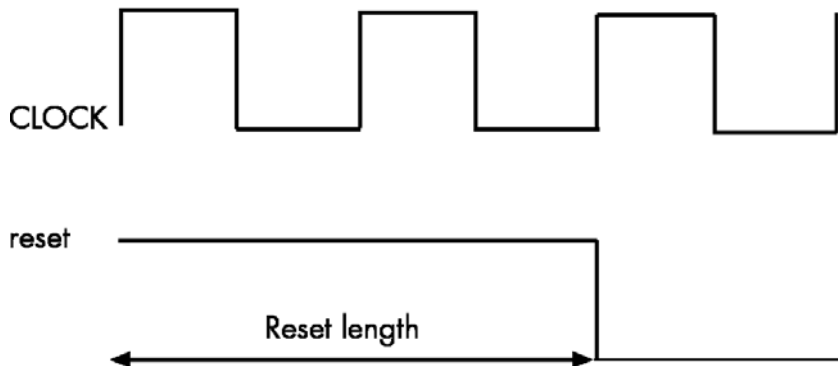
Reset length (in clock cycles)

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 correct setting for **Hold input data between samples**. This setting matches the behavior of a Simulink simulation, in which subrate signals are always 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 all HDL test bench code and 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

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 correctly 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 any 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 Embedded MATLAB Function block (see “Distributed Pipeline Insertion” on page 12-58)
- When you specify the `{'ResetType', 'None'}` parameter for any of the following block types:
 - Integer Delay
 - Tapped Delay
 - Unit Delay
 - Unit Delay Enabled
- When generating a black box interface to existing manually written HDL code

Command-Line Information

Property: IgnoreDataChecking

Type: integer

Default: 0

See Also

IgnoreDataChecking

Generate cosimulation blocks

Generate a model containing HDL Cosimulation block(s) for use in testing the DUT.

Settings

Default: Off



On

When this option is selected, if your installation is licensed for one or more of the following HDL simulation products, the coder generates and opens a model that contains an HDL Cosimulation block for each licensed product:

- EDA Simulator Link MQ
- EDA Simulator Link IN
- EDA Simulator Link DS

The generated HDL Cosimulation blocks are configured 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.

Command-Line Information

Property: GenerateCoSimBlock

Type: string

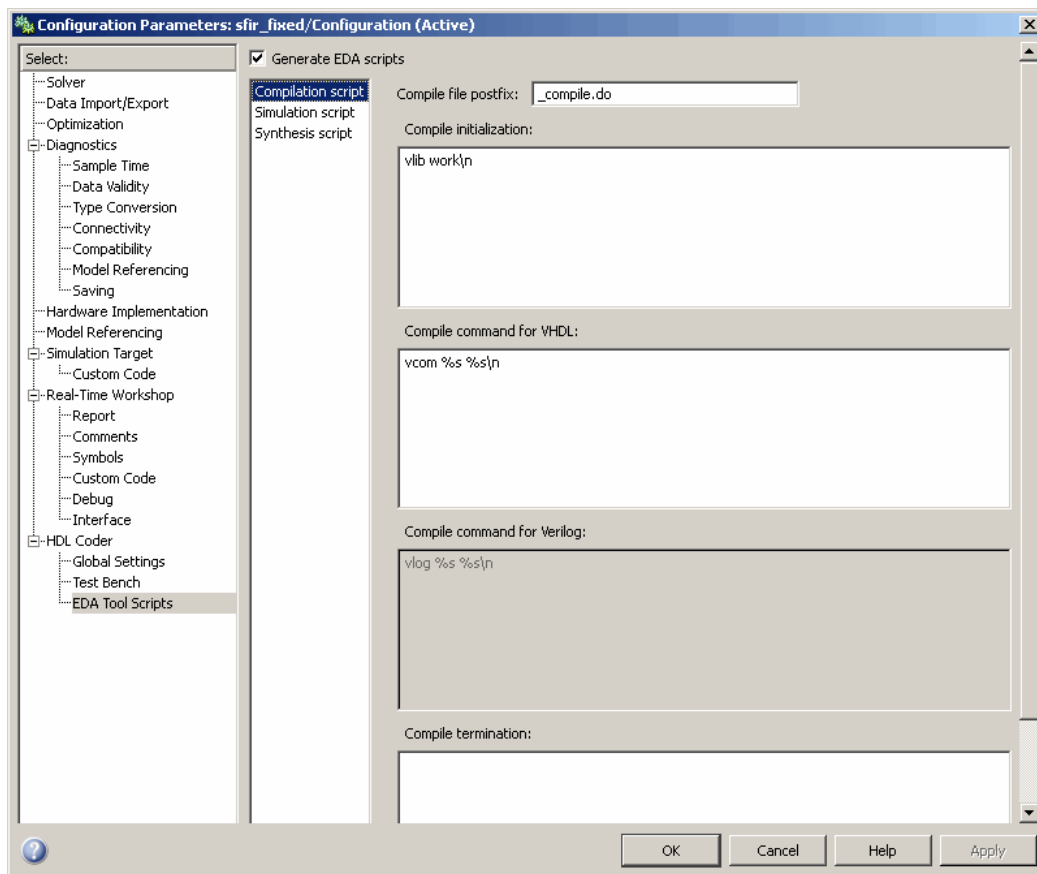
Value: 'on' | 'off'

Default: 'off'

See Also

GenerateCoSimBlock

HDL Coder Pane: EDA Tool Scripts



In this section...

“EDA Tool Scripts Overview” on page 3-76

“Generate EDA scripts” on page 3-77

“Compile file postfix” on page 3-78

“Compile Initialization” on page 3-79

“Compile command for VHDL” on page 3-80

In this section...

“Compile command for Verilog” on page 3-81

“Compile termination” on page 3-82

“Simulation file postfix” on page 3-83

“Simulation initialization” on page 3-84

“Simulation command” on page 3-85

“Simulation waveform viewing command” on page 3-86

“Simulation termination” on page 3-87

“Synthesis file postfix” on page 3-88

“Synthesis initialization” on page 3-89

“Synthesis command” on page 3-90

“Synthesis termination” on page 3-91

EDA Tool Scripts Overview

The **EDA Tool Scripts** pane lets you set all 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

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- EDAScriptGeneration

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

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- `HDLCompileFilePostfix`

Compile Initialization

Specify a format string passed to `fprintf` to write the Init section of the compilation script.

Settings

Default: `vlib work\n`

The Init phase of the script performs any required setup actions, such as creating a design library or a project file.

Command-Line Information

Property: `HDLCompileInit`

Type: `string`

Default: `'vlib work\n'`

See Also

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- `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

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- `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

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- `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 takes no arguments.

Command-Line Information

Property: `HDLCompileTerm`

Type: `string`

Default: `''`

See Also

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- `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

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- `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 any 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

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- 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

- Controlling Script Generation with the EDA Tool Scripts GUI Pane.
- HDLSimCmd

Simulation waveform viewing command

Specify the waveform viewing command written to simulation script.

Settings

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

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

Command-Line Information

Property: HDLSimViewWaveCmd

Type: string

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

See Also

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- 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 takes no arguments.

Command-Line Information

Property: `HDLSimTerm`

Type: `string`

Default: `'run -all\n'`

See Also

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- `HDLSimTerm`

Synthesis file postfix

Specify a postfix string appended to file name for generated Synplify synthesis scripts.

Settings

Default: `_synplify.tcl`

For example, if the name of the device under test is `my_design`, the coder adds the postfix `_synplify.tcl` to form the name `my_design_synplify.tcl`.

Command-Line Information

Property: `HDLSynthFilePostfix`

Type: `string`

Default: `'_synplify.tcl'`

See Also

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- `HDLSynthFilePostfix`

Synthesis initialization

Specify a format string passed to `fprintf` to write the initialization section of the synthesis script.

Settings

Default: `project -new %s.prj\n`

The default string is a synthesis project creation command. The implicit argument is the top-level module or entity name.

Command-Line Information

Property: HDLSynthInit

Type: string

Default: `'project -new %s.prj\n'`

See Also

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- HDLSynthInit

Synthesis command

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

Settings

Default: `add_file %s\n`

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

Command-Line Information

Property: HDLSynthCmd

Type: string

Default: `'add_file %s\n'`

See Also

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- HDLSynthCmd

Synthesis termination

Specify a format string passed to `fprintf` to write the termination portion of the synthesis script.

Settings

Default:

```
['set_option -technology VIRTEX4\n',...  
'set_option -part XC4VSX35\n',...  
'set_option -synthesis_onoff_pragma 0\n',...  
'set_option -frequency auto\n',...  
'project -run synthesis\n']
```

The termination phase (Term) is the final execution phase of the script. The Term phase takes no arguments.

Command-Line Information

Property: HDLSynthTerm

Type: string

Default: ['set_option -technology VIRTEX4\n', 'set_option -part XC4VSX35\n', 'set_option -synthesis_onoff_pragma 0\n', 'set_option -frequency auto\n', 'project -run synthesis\n']

See Also

- Controlling Script Generation with the EDA Tool Scripts GUI Pane
- HDLSynthTerm

Generating HDL Code for Multirate Models

- “Overview” on page 4-2
- “Configuring Multirate Models for HDL Code Generation” on page 4-3
- “Example: Model with a Multirate DUT” on page 4-6
- “Properties Supporting Multirate Code Generation” on page 4-9

Overview

The coder supports HDL code generation for single-clock, single-tasking 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.

HDL code generated from multirate models employs a single clock. A *timing controller* (`Timing_Controller`) entity generates the required rates from a single master clock using one or more counters and multiple clock enables. The master clock rate (always the fastest rate in the model) is referred to as the *base rate*. The rates generated from the master clock are referred to as *subrates*. The `Timing_Controller` entity definition is written to a separate code file (`Timing_Controller.vhd` or `Timing_Controller.v`).

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. These are discussed in the following sections.

Configuring Multirate Models for HDL Code Generation

In this section...

“Overview” on page 4-3

“Configuring Model Parameters” on page 4-3

“Configuring Sample Rates in the Model” on page 4-4

“Constraints for Rate Transition Blocks and Other Blocks in Multirate Models” on page 4-4

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 ensures that your multirate model is set up correctly 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 correct 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** ensures that any illegal rate transitions are detected 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.

Constraints for Rate Transition Blocks and Other Blocks in Multirate Models

This section describes constraints you should observe when configuring Rate Transition, Upsample, Downsample, Zero-Order Hold, and various types of delay blocks in multirate models intended for HDL code generation.

Rate Transition Blocks

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 Blocks

Use Rate Transition blocks, rather than any of the following block types, to create rate transitions in models intended for HDL code generation:

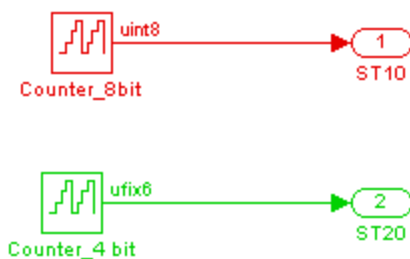
- Unit Delay
- Unit Delay Enabled
- Integer Delay
- Tapped Delay
- Zero-Order Hold

All types of Delay blocks listed should be configured to have the same input and output sample rates.

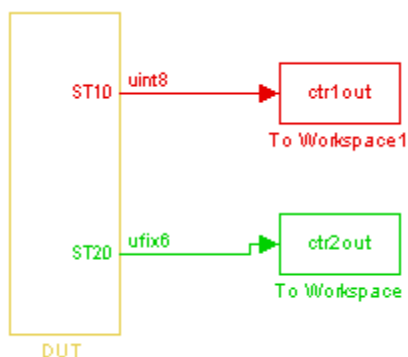
Zero-Order Hold blocks must be configured with inherited (–1) sample times.

Example: Model with 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.

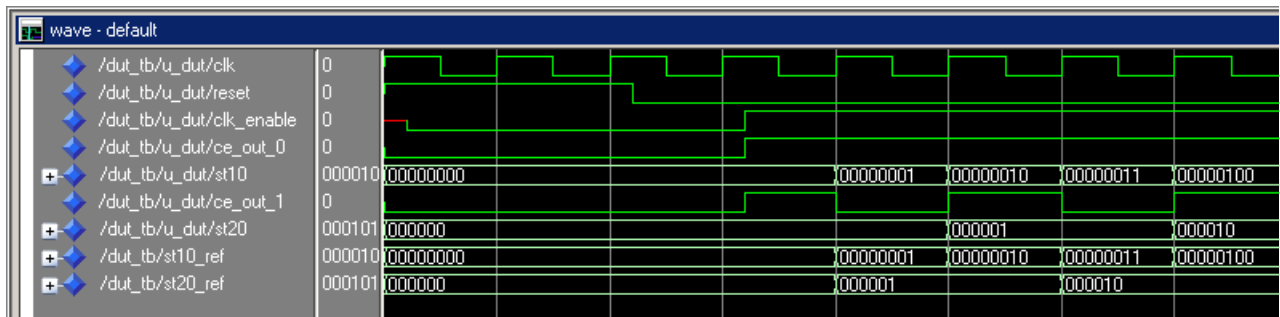
```

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 TimingController 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.

Properties Supporting Multirate Code Generation

In this section...
“Overview” on page 4-9
“HoldInputDataBetweenSamples” on page 4-9
“OptimizeTimingController” on page 4-9

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.

Code Generation Control Files

- “Overview of Control Files” on page 5-2
- “Structure of a Control File” on page 5-5
- “Code Generation Control Objects and Methods” on page 5-7
- “Using Control Files in the Code Generation Process” on page 5-15
- “Specifying Block Implementations and Parameters in the Control File” on page 5-24

Overview of Control Files

In this section...
“What Is a Control File?” on page 5-2
“Selectable Block Implementations and Implementation Parameters” on page 5-3
“Implementation Mappings” on page 5-4
“Control File Demo” on page 5-4

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.

A control file is an M-file that you attach to your model, using either the `makehdl` command or the Configuration Parameters dialog box. You do not need to know any internal details of the code generation process to use a control file.

In the current release, 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, many 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.

See Chapter 6, “Specifying Block Implementations and Parameters for HDL Code Generation” for a complete summary of all supported blocks and their implementations and implementation parameters.

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.

Control File Demo

The “Getting Started with Control Files” demo illustrates the use of simple control files to define implementation mappings and generate Verilog code. The demo is located in the **Demos** pane on the left of the MATLAB Help browser. To run the demo, select **Simulink > Simulink HDL Coder > Getting Started with Control Files** in the **Demos** pane. Then follow the demo instructions.

Structure of a Control File

The required elements for a code generation control file are as follows:

- A control file is an M-file that 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 5-7. You do not need to know internal details of the code generation control object or the class to which it belongs.

The object is constructed 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 all other public methods of the object are discussed in “Code Generation Control Objects and Methods” on page 5-7.
- 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 5-15. The interface between the code generator and your attached control file is automatic.
- A control file must be located in either the current working directory, or a directory that is in the MATLAB path.

However, your control files should not be located within the MATLAB directory tree because they could be overwritten by subsequent installations.

Code Generation Control Objects and Methods

In this section...

“Overview” on page 5-7

“hdlnewcontrol” on page 5-7

“forEach” on page 5-7

“forAll” on page 5-12

“set” on page 5-12

“generateHDLFor” on page 5-13

“hdlnewcontrolfile” on page 5-14

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. All 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 5-24 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 <code>'built-in/blockname'</code> For other blocks, <code>blocktype</code> must include the full path to the library containing the block, for example: <code>'dsparch4/Digital Filter'</code>

Argument	Type	Description
implementation	String	<p>An HDL block implementation to be used in code generation for all blocks that meet the <code>modelscope</code> and <code>blocktype</code> search criteria. Specify <code>implementation</code> as <code>package.class</code>, for example:</p> <pre>hdldefaults.GainMultHDLEmission</pre> <p>“Summary of Block Implementations” on page 6-2 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 <code>implementation</code> argument. Specify parameters as: <code>'Property', value</code> where <code>'Property'</code> is the name of the property and <code>value</code> is the value applied to the property. If the implementation has no parameters, or you want to use default parameters, pass in an empty cell array (<code>{}</code>).</p> <p>“Block Implementation Parameters” on page 6-41 describes the syntax of each parameter, and describes how the parameter affects generated code.</p> <p>“Summary of Block Implementations” on page 6-2 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 5-24.</p>

Argument	Type	Description
modelscope	String or cell array of strings	<p>Strings defining one or more Simulink paths: { 'path1' 'path2' ... 'pathN' }</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, all blocks at a specified level of the model, or a specific block or subsystem. A path terminating in a wildcard ('*') includes all blocks at or below the model level specified by the path. Syntax for modelscope paths is</p> <ul style="list-style-type: none"> • 'model/*': all blocks in the model • 'model/subsyslevel/block': a specific block within a specific level of the model • 'model/subsyslevel/subsystem': a specific subsystem block within a specific level of the model • 'model/subsyslevel/*': any block within a specific model level <p>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: './subsyslevel/block'. See also “Representation of the Root Model in modelscope” on page 5-10 and “Resolution of modelscope” on page 5-11.</p>

Representation of the Root Model in modelscope

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', {}, ...
    'hdldefaults.MinMaxCascadeHDL Emission' );
```

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 necessary to 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', {}, ...
    'hdldefaults.MinMaxCascadeHDL Emission');
```

If you represent the model in this way, and then save the model under a different name, the control file does not require any 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('dct8_fixed/OneD_DCT8/Gain14',...
    'built-in/Gain', {},...
    'hdldefaults.GainMultHDL Emission');
% 2. Use factored CSD Gain block implementation
% for all Gain blocks at or below level of OneD_DCT8 subsystem.
c.forEach('dct8_fixed/OneD_DCT8/*',...
    'built-in/Gain', {},...
    'hdldefaults.GainFCSDHDL Emission');
```

The first `forEach` call defines an implementation mapping for a specific block within the subsystem `OneD_DCT8`. The second `forEach` call defines a different implementation mapping for all blocks within or below the subsystem `OneD_DCT8`.

The coder resolves such ambiguities by always giving higher priority to the more specific `modelscope`. In the example, the `Gain14` block uses the `hdldefaults.GainMultHDLEmission` implementation, while all other blocks within or below the subsystem `OneD_DCT8` use the `hdldefaults.GainFCSDHDLEmission` implementation.

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'})
```

All other arguments are the same as those described for “`forEach`” on page 5-7.

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 any of the code generation properties documented in Chapter 15, “Properties — Alphabetical List”, *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 all supported implementations and parameters, for all 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 5-15

“Creating a Control File and Saving Your HDL Code Generation Settings” on page 5-15

“Making Your Control Files More Portable” on page 5-19

“Associating an Existing Control File with Your Model” on page 5-19

“Detaching a Control File from Your Model” on page 5-22

“Setting Up HDL Code Generation Defaults with a Control File” on page 5-22

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 directory that is in the MATLAB path, or the current working directory.
- Do not locate a control file within the MATLAB directory tree, because it could be overwritten by subsequent MATLAB installations.

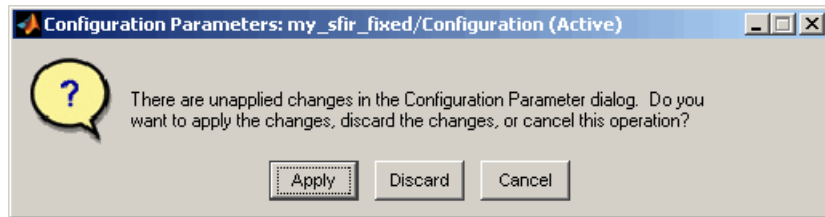
Creating a Control File and Saving Your HDL Code Generation Settings

Note When you save a Simulink model, your HDL code generation settings are not saved with the model like other components of the model's configuration set. If you want your HDL code generation settings to persist across sessions with a model, you must save your current settings to a control file. The control file is then linked to the model, and the linkage is preserved when you save the model.

Saving Your HDL Code Generation Settings to a Control File

To save your current HDL code generation settings to a control file:

- 1 Open the Configuration Parameters dialog box and select the **HDL Coder** pane.
- 2 In the **Code generation control file** subpane, click **Save**.
- 3 If you have changed HDL code generation settings but have not yet applied them, the following prompt is displayed.



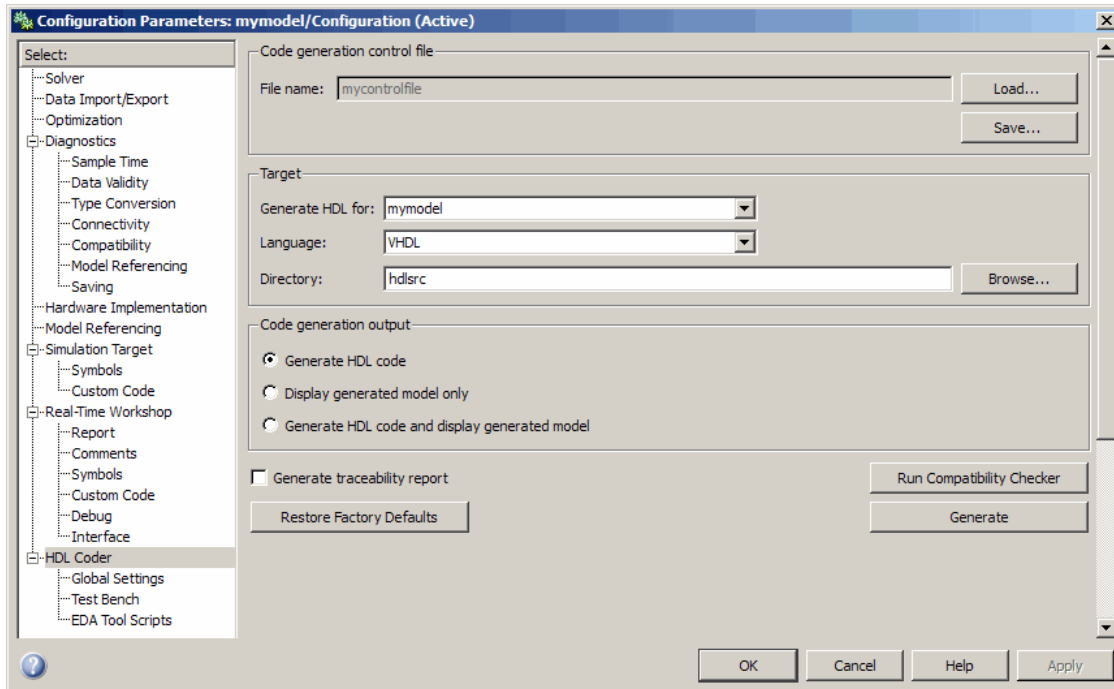
Click **Apply** to apply any HDL code generation option settings you may have changed.

- 4 A standard file dialog box opens. Navigate to the directory where you want to save the control file. This directory must be either the current working directory, or a directory that is in the MATLAB path.

Specifying a control file that is not on the MATLAB path or in the current working directory is not recommended. Instead, you should modify the MATLAB path such that the directory where you want to store the control file is on the path.

Do not locate the control file within the MATLAB directory tree, because it could be overwritten by subsequent MATLAB installations.

- 5 Enter the desired file name and save the file.
- 6 Linkage between the model and the control file is now established. The control file name is displayed in the **File name** field, as shown in the following figure.



- 7 Save the model if you want the control file linkage to persist in future sessions with your model.

The control file you saved contains a `generateHDLFor` statement (see “generateHDLFor” on page 5-13) that specifies the path to the DUT specified in the **Generate HDL for** field. In this path, the root-level model is represented by the period (see “Representation of the Root Model in modelscopes” on page 5-10, rather than by an explicit model name reference. This makes the control file more portable.

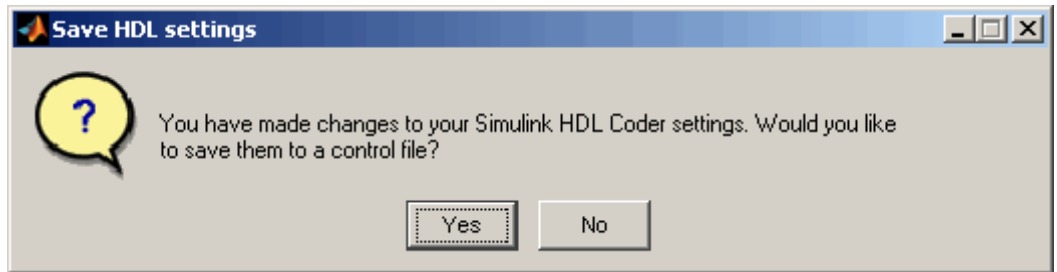
If you later select a different DUT for code generation, or make structural changes to your model (such as renaming the DUT), be sure to update this path information by resaving the control file.

The control file also preserves the values of all HDL code generation properties in the form of a call to the `set` method (see “set” on page 5-12). Properties are passed in to the call in alphabetical order.

- 8 If desired, you can now customize the control file using the MATLAB Editor or some other text editor. For example, you may want to add `ForEach` statements to define block implementation bindings. After you edit and save your changes to the control file, be sure to reload it by clicking **Load**.

Saving Your HDL Code Generation Settings when Closing Your Model

When you close your model, the coder displays the following message if you have made changes to the HDL code generation settings but have not yet saved them to a control file.



If you click **Yes**, a standard file dialog box opens. You can then navigate to the desired directory and save the control file.

Save the control file to a directory that is on the MATLAB path, or the current working directory. If necessary, modify the MATLAB path such that the directory where you want to store the control file is on the path.

Do not locate the control file within the MATLAB directory tree, because it could be overwritten by subsequent MATLAB installations.

Creating a Control File Manually

You can create a control file manually using the MATLAB Editor or some other text editor. See “Structure of a Control File” on page 5-5 to make sure your files are set up correctly.

One reason for creating a control file manually is to create a control file that sets defaults for a subset of HDL code generation properties. See “Setting

Up HDL Code Generation Defaults with a Control File” on page 5-22 for an example.

If you create a control file manually, you must link it to your model, as described in “Associating an Existing Control File with Your Model” on page 5-19.

If the control file you have created is not on the MATLAB path, you should modify the path to include the directory where the control file is stored.

Making Your Control Files More Portable

It can be advantageous to code your control files so that they are independent of any 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', {}, ...  
            'hdldefaults.MinMaxCascadeHDLemission');
```

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 all references to the root-level model will be valid.

Associating an Existing Control File with Your Model

A control file must be associated with your model before you can use the control file in code generation.

If you are generating code using the `makehdl` or `makehdltb` commands, use the `HDLControlFiles` property to specify the location of the control file. A control file must be located in the current working directory or on the MATLAB path.

In the following example, the control file is assumed to be located on the MATLAB path or in the current working directory, and to have the default file-name extension `.m`.

```
makehdl('HDLControlFiles', {'dct8config'});
```

If you are using the GUI to generate code, specify the location of the control file as follows:

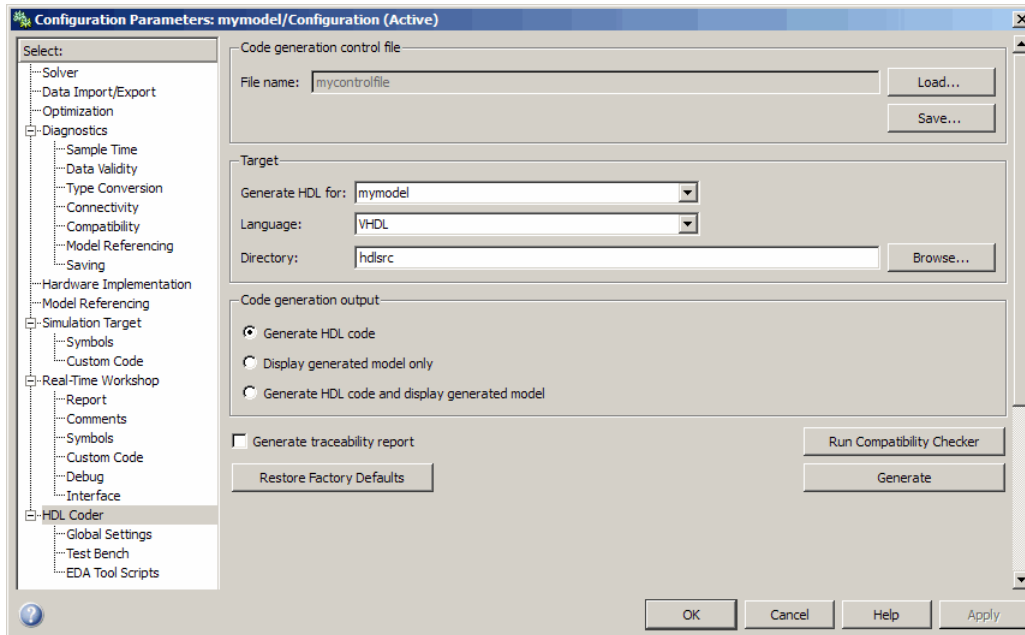
- 1** Open the Configuration Parameters dialog box and select the **HDL Coder** pane.
- 2** Check the **File name** field to see if a control file is already linked to the model. If the **File name** field is blank, the model has no linked control file; proceed to step 3.

If the **File name** field is populated, the model is linked to a control file. If you want to replace that linkage and load in a different control file, proceed to step 3. Otherwise, no action is required.

- 3** In the **Code generation control file** subpane, click **Load**.
- 4** A standard file dialog box opens. Navigate to the desired control file and select it.

A control file must be in the current working directory or on the MATLAB path. If you attempt to load a control file from a directory that does not meet this requirement, the coder will display an error message.

- 5** The control file name appears in the **File name** field, as shown in the following figure.



6 Click **Apply**.

7 The control file is now linked to your model and is used when code is generated. Save the model if you want the control file linkage to persist in future sessions with your model.

Detaching a Control File from Your Model

The quickest (and recommended) way to detach a control file from your model is to click **Restore Factory Defaults**. This button removes the control file linkage, clears the **File name** field, and resets all HDL code generation properties to their default settings.

Note **Restore Factory Defaults** resets all HDL code generation settings. This action cannot be cancelled or undone. To recover previous settings, you must close the model without saving it, and then reopen it.

Any of the following actions also detach a control file from a model:

- Attach another control file, using either the **Load** button or a call to `makehdl`
- Close the model after attaching a control file, without saving the model
- Clear the `HDLControlFiles` property by passing a null file name argument to `makehdl`, as in this example:

```
makehdl(gcb, 'HDLControlFiles', {' '});
```

Setting Up HDL Code Generation Defaults with a Control File

The Model Configuration Preferences dialog box of the Model Explorer does not currently include HDL code generation settings. However, you can use a control file to define HDL code generation settings that you can subsequently load into any model. You can use such a control file to set up default or template HDL code generation settings for your projects or organization.

For example, suppose that you want the following settings to be applied to all models for a certain HDL project:

- Code is generated in Verilog.
- Generated code is written to a subdirectory of the user's working directory, named `vlog_gen_code`.
- Use of Verilog ``timescale` directives is disabled.

The following code example lists a control file that enforces these requirements:

```
function c = my_sfir_fixed_control

c = hdlnewcontrol(mfilename);
c.set( ...
    'TargetDirectory',      'vlog_gen_code',...
    'TargetLanguage',      'verilog',...
    'UseVerilogTimescale', 'off'...
);
```

An important feature of this control file is that it does not contain any code referencing elements that are specific to any particular model (such as paths in `generateHDLFor` or `forEach` calls). Therefore, the control file is portable and can be loaded into any model.

Loading a control file for the purpose of setting up defaults into a model is no different than loading any other control file (see “Associating an Existing Control File with Your Model” on page 5-19). However, if you load the same control file into multiple models, take care not to overwrite the original control file.

Specifying Block Implementations and Parameters in the Control File

In this section...

“Overview” on page 5-24

“Generating Selection/Action Statements with the `hdlnewforeach` Function” on page 5-24

Overview

The coder provides a default HDL block implementation for all 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 an implementation package and class, in the form `package.class`. Pass in the `package.class` specification and implementation parameters (if any) to the implementation argument of a `forEach` or `forall` call, as in the following example.

```
config.forEach('*',...
    'built-in/Sum', {},...
    'hdldefaults.SumRTW', {'OutputPipeline', 2});
```

Given the `package.class` specification, the coder will call the appropriate code generation method. You do not need to know any internal details of the implementation classes.

Generating Selection/Action Statements with the `hdlnewforeach` Function

Determining the block path, type, implementation `package.class` 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 correct `modelscope`, `blocktype`, and default implementation arguments for the block
- (Optional) A cell array of strings enumerating the available implementations for the block, in `package.class` form
- (Optional) A cell array of strings enumerating the names of implementation parameters (if any) corresponding to the block implementations. `hdlnewforeach` does not list data types and other details of block implementation parameters. These details are described in “Block Implementation Parameters” on page 6-41.

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` demo model. To create the control file:

- 1** In the MATLAB Command Window, select **File > New > M-File**. The MATLAB Editor opens an empty M-file.
- 2** Create a skeletal control file by entering the following code into the M-file window:

```
function c = newforeachexamp
c = hdlnewcontrol(mfilename);

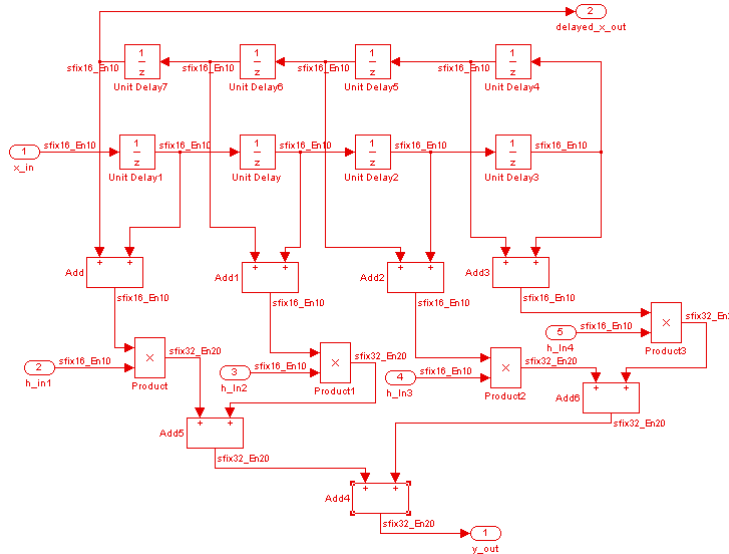
% Set top-level subsystem from which code is generated.
c.generateHDLFor('sfir_fixed/symmetric_fir');
% INSERT FOREACH CALL BELOW THIS LINE.
```

- 3** Save the file as `newforeachexamp.m`.
- 4** Open the `sfir_fixed` demo model.

- 5** Before invoking `hdlnewforeach`, you must run `checkhdl` or `makehdl` to build in-memory information about the model. At the MATLAB command prompt, run `checkhdl` on the `symmetric_fir` subsystem, as shown in the following code example:

```
checkhdl('sfir_fixed/symmetric_fir')
### Starting HDL Check.
### HDL Check Complete with 0 errors, warnings and messages.
```

- 6** Close the `checkhdl` report window and activate the `sfir_fixed` model window.
- 7** 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] = hdlnewforeach(gcb)
```

- 2** The command returns the following results:

```

c.forEach('sfir_fixed/symmetric_fir/Add4',...
'built-in/Sum', {},...
'hdldefaults.SumRTW', {});

impl =

    {4x1 cell}

parms =

    {1x2 cell}    {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: `hdldefaults.SumRTW`. 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 =

    'hdldefaults.SumCascadeHDLEmission'
    'hdldefaults.SumLinearHDLEmission'
    'hdldefaults.SumRTW'
    'hdldefaults.SumTreeHDLEmission'

```

See the table Built-In/Sum on page 6-26 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{1:4}
```

```
ans =  
  
    'InputPipeline'    'OutputPipeline'  
  
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 6-41 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('sfir_fixed/symmetric_fir/Add4',...  
    'built-in/Sum', {},...  
    'hdldefaults.SumRTW', {});
```

- 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('sfir_fixed/symmetric_fir/Add4',...
'built-in/Sum', {},...
'hdldefaults.SumRTW', {'OutputPipeline', 2});
```

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',...
'built-in/Sum', {},...
'hdldefaults.SumRTW', {'OutputPipeline', 2});
```

The demo “Getting Started with Output Pipeline Commands in Control Files” gives a more detailed example of pipelining, including analysis of resulting clock rate improvements in a synthesized HDL model.

Note For convenience, `hdlnewforeach` supports a more abbreviated syntax than that used in the previous example. See the `hdlnewforeach` reference page.

Specifying Block Implementations and Parameters for HDL Code Generation

- “Summary of Block Implementations” on page 6-2
- “Blocks with Multiple Implementations” on page 6-22
- “Block-Specific Usage, Requirements, and Restrictions for HDL Code Generation” on page 6-35
- “Block Implementation Parameters” on page 6-41
- “Blocks That Support Complex Data” on page 6-56

Summary of Block Implementations

The following table summarizes all blocks that are supported for HDL code generation and their available implementations in the current release. The columns signify

- *Simulink Block*: Library path and block name.
- *Blockscope*: Block path and name to be passed as a blockscope string argument to `forEach` or `forall`.
- *Implementations and Parameters*: Names of available implementations, and parameters supported for the implementation (if any). For blocks that have more than one implementation listed, see “Blocks with Multiple Implementations” on page 6-22 for information on the trade-offs involved in choosing different implementations.

When specifying an implementation argument to `forEach` or `forall`, use the format `package.class`, for example, `hdldefaults.AssignmentHDL Emission` or `hdlstateflow.StateflowHDL Instantiation`. Almost all implementation classes currently belong to the package `hdldefaults`. In the following table, the package name is given explicitly only for classes that belong to some other package.

See “Block Implementation Parameters” on page 6-41 for information on implementation parameters and how to specify them.

Some blocks have specific requirements and restrictions on how they are configured for HDL code generation. The table provides links to relevant documentation for blocks that have such requirements.

Note Support for complex signals is limited to a subset of the blocks listed in this section. See “Blocks That Support Complex Data” on page 6-56.

Simulink Block	Blockscope	Implementations and Parameters
<p>commseqgen2/PN Sequence Generator</p> <p>(See “PN Sequence Generator Block Requirements and Restrictions” on page 6-39.)</p>	commseqgen2/PN Sequence Generator	<p>PNGenHDL Emission</p> <p><i>Parameters:</i> OutputPipeline, InputPipeline</p>
<p>dspadpt3/LMS Filter</p> <p>(See “LMS Filter Usage and Restrictions” on page 6-36.)</p>	dspadpt3/LMS Filter	<p>LMSFilterHDL Emission</p> <p><i>Parameters:</i> OutputPipeline, InputPipeline</p>
<p>dsparch4/Biquad Filter</p> <p>(See “Biquad Filter Block Requirements and Restrictions” on page 6-35, “CoeffMultipliers” on page 6-41.)</p>	dsparch4/Biquad Filter	<p>BiquadFilterHDL Instantiation</p> <p><i>Parameters:</i> OutputPipeline, InputPipeline, CoeffMultipliers</p>
<p>dsparch4/Digital Filter</p> <p>(See “Digital Filter Block Requirements and Restrictions” on page 6-35, “CoeffMultipliers” on page 6-41, “Distributed Arithmetic Implementation Parameters for Digital Filter Block” on page 6-42.)</p>	dsparch4/Digital Filter	<p>DigitalFilterHDL Instantiation</p> <p><i>Parameters:</i> OutputPipeline, InputPipeline, CoeffMultipliers, DALUTPartition, DARadix</p>
<p>dspindex/Multiport Selector</p>	dspindex/Multiport Selector	<p>MultiportSelectorHDL Emission</p> <p><i>Parameters:</i> OutputPipeline, InputPipeline</p>

Simulink Block	Blockscope	Implementations and Parameters
dspindex/Variable Selector	dspindex/Variable Selector	VariableSelectorHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
dspmlti4/CIC Decimation (See “Multirate CIC Decimation and Multirate FIR Decimation Blocks Requirements and Restrictions” on page 6-37.)	dspmlti4/CIC Decimation	CICDecimationHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline
dspmlti4/CIC Interpolation (See “Multirate CIC Interpolation and Multirate FIR Interpolation Blocks Requirements and Restrictions” on page 6-38.)	dspmlti4/CIC Interpolation	CICInterpolationHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline
dspmlti4/FIR Decimation (See “Multirate CIC Decimation and Multirate FIR Decimation Blocks Requirements and Restrictions” on page 6-37, “CoeffMultipliers” on page 6-41.)	dspmlti4/FIR Decimation	FIRDecimationHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline, CoeffMultipliers

Simulink Block	Blockscope	Implementations and Parameters
dspmlti4/FIR Interpolation (See “Multirate CIC Interpolation and Multirate FIR Interpolation Blocks Requirements and Restrictions” on page 6-38, “CoeffMultipliers” on page 6-41.)	dspmlti4/FIR Interpolation	FIRInterpolationHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline, CoeffMultipliers
dspsigattribs/Convert 1-D to 2-D	dspsigattribs/Convert 1-D to 2-D	PassThroughHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
dspsigattribs/Frame Conversion	built-in/FrameConversion	FrameConversionHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
dspsigops/Delay	dspsigops/Delay	DSPDelayHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline, ResetType
dspsigops/Downsample	dspsigops/Downsample	DownsampleHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
dspsigops/Upsample	dspsigops/Upsample	UpsampleHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
dspsigops/NCO (See “NCO Block Requirements and Restrictions” on page 6-39.)	dspsigops/NCO	NCOHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
dspsnks4/Matrix Viewer	dspsnks4/Matrix Viewer	NoHDL Emission

Simulink Block	Blockscope	Implementations and Parameters
dspsnks4/Signal To Workspace	dspsnks4/Signal To Workspace	NoHDL Emission
dspsnks4/Spectrum Scope	dspsnks4/Spectrum Scope	NoHDL Emission
dspsnks4/Time Scope	built-in/Scope	NoHDL Emission
dspsnks4/Vector Scope	dspsnks4/Vector Scope	NoHDL Emission
dspsnks4/Waterfall	dspsnks4/Waterfall	NoHDL Emission
dspsrcs4/DSP Constant	dspsrcs4/DSP Constant	Constant (<i>default</i>) ConstantHDL Emission <i>Parameters:</i> OutputPipeline
dspsrcs4/Sine Wave (See “Sine Wave Block Requirements and Restrictions” on page 6-40.)	dspsrcs4/Sine Wave	SineWaveHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
dspstat3/Maximum	dspstat3/Maximum	MinMaxTree MinMaxCascade MinMaxTreeHDL Emission MinMaxCascadeHDL Emission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.

Simulink Block	Blockscope	Implementations and Parameters
dspstat3/Minimum	dspstat3/Minimum	MinMaxTree MinMaxCascade MinMaxTreeHDL Emission MinMaxCascadeHDL Emission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.
hlddemolib/Bit Concat (See “Bitwise Operators” on page 7-35.)	hlddemolib/Bit Concat	BitConcat <i>Parameters:</i> OutputPipeline, InputPipeline.
hlddemolib/Bit Reduce (See “Bitwise Operators” on page 7-35.)	hlddemolib/Bit Reduce	BitReduce <i>Parameters:</i> OutputPipeline, InputPipeline
hlddemolib/Bit Rotate (See “Bitwise Operators” on page 7-35.)	hlddemolib/Bit Rotate	BitRotate <i>Parameters:</i> OutputPipeline, InputPipeline
hlddemolib/Bit Shift (See “Bitwise Operators” on page 7-35.)	hlddemolib/Bit Shift	BitShift <i>Parameters:</i> OutputPipeline, InputPipeline
hlddemolib/Bit Slice (See “Bitwise Operators” on page 7-35.)	hlddemolib/Bit Slice	BitSlice <i>Parameters:</i> OutputPipeline, InputPipeline
hlddemolib/Dual Port RAM (See “Dual Port RAM Block” on page 7-6.)	hlddemolib/Dual Port RAM	RamBlockDualHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline, AddClockEnablePort

Simulink Block	Blockscope	Implementations and Parameters
hdlldemolib/HDL Counter (See “HDL Counter” on page 7-15.)	hdlldemolib/HDL Counter	HDLCounterHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
hdlldemolib/HDL FFT (See “HDL FFT” on page 7-27.)	hdlldemolib/HDL FFT	FFTDITMRHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
hdlldemolib/Simple Dual Port RAM (See “Simple Dual Port RAM Block” on page 7-7.)	hdlldemolib/Simple Dual Port RAM	RamBlockSimpDualHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline
hdlldemolib/Single Port RAM (See “Single Port RAM Block” on page 7-9.)	hdlldemolib/Single Port RAM	RamBlockSingleHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline, AddClockEnablePort
lflinklib/HDL Cosimulation	lflinklib/HDL Cosimulation	hdlincisive.IncisiveHDLInstantiation <i>Parameters:</i> See “Interface Generation Parameters” on page 6-54.
modelsimlib/HDL Cosimulation	modelsimlib/HDL Cosimulation	ModelSimHDLInstantiation <i>Parameters:</i> See “Interface Generation Parameters” on page 6-54.
modelsimlib/To VCD File	modelsimlib/To VCD File	NoHDL Emission
sfib/Chart (See Chapter 11, “Stateflow HDL Code Generation Support”.)	sfib/Chart	hdlstateflow .StateflowHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline, DistributedPipelining

Simulink Block	Blockscope	Implementations and Parameters
sfib/Truth Table	sfib/Truth Table	hdlstateflow .StateflowHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline, DistributedPipelining
Signal Routing/From	built-in/From	FromBlock <i>Parameters:</i> OutputPipeline, InputPipeline
Signal Routing/Go To	built-in/Goto	GotoBlock <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Enabled	simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Enabled	UnitDelayEnabledHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline, ResetType
simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Decrement Real World	simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Decrement Real World	IncrementOrDecrementRWV <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Increment Real World	simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Decrement Real World	IncrementOrDecrementRWV <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Decrement Store Integer	simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Decrement Real World	IncrementOrDecrementSI <i>Parameters:</i> OutputPipeline, InputPipeline

Simulink Block	Blockscope	Implementations and Parameters
simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Increment Store Integer	simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Decrement Real World	IncrementOrDecrementSI <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Commonly Used Blocks/Constant	built-in/Constant	Constant (<i>default</i>) ConstantHDL Emission ConstantSpecialHDL Emission <i>Parameters:</i> Both implementations support OutputPipeline. ConstantSpecialHDL Emission also supports Value parameter (see Built-In/Constant on page 6-23).
simulink/Commonly Used Blocks/Data Type Conversion (See “Data Type Conversion Block Requirements and Restrictions” on page 6-35.)	built-in/ DataTypeConversion	DataTypeConversionRTW DataTypeConversionHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Commonly Used Blocks/Demux	built-in/Demux	Demux (<i>default</i>) DemuxHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Commonly Used Blocks/Gain	built-in/Gain	GainMultHDL Emission GainFCSDHDL Emission GainCSDHDL Emission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.

Simulink Block	Blockscope	Implementations and Parameters
simulink/Commonly Used Blocks/Ground	built-in/Ground	Constant (<i>default</i>) ConstantHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Commonly Used Blocks/In1	built-in/Inport	NoHDL Emission (Input ports are generated automatically.)
simulink/Commonly Used Blocks/Logical Operator	built-in/Logic	LogicHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Commonly Used Blocks/Mux	built-in/Mux	Mux (<i>default</i>) MuxHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Commonly Used Blocks/Out1	built-in/Outport	NoHDL Emission (Output ports are generated automatically.)
simulink/Commonly Used Blocks/Product	built-in/Product	ProductRTW ProductLinearHDL Emission ProductTree ProductTreeHDL Emission ProductCascade ProductCascadeHDL Emission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline. Note: ProductTree and ProductCascade are supported for Product blocks having a single

Simulink Block	Blockscope	Implementations and Parameters
		vector input that has two or more elements.
simulink/Commonly Used Blocks/Relational Operator	built-in/RelationalOperator	RelationalOperatorHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Commonly Used Blocks/Scope	built-in/Scope	NoHDL Emission
simulink/Commonly Used Blocks/Sum	built-in/Sum	SumRTW SumLinearHDL Emission SumTreeHDL Emission SumCascade SumCascadeHDL Emission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline. Note: SumTreeHDL Emission and SumCascade are supported for Sum blocks having a single vector input that has two or more elements.
simulink/Commonly Used Blocks/Switch	built-in/Switch	SwitchRTW (<i>default</i>) SwitchHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Commonly Used Blocks/Terminator	built-in/Terminator	NoHDL Emission
simulink/Commonly Used Blocks/Unit Delay	built-in/UnitDelay	UnitDelayRTW UnitDelayHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline, ResetType

Simulink Block	Blockscope	Implementations and Parameters
simulink/Discontinuities /Saturation Dynamic	simulink/Discontinuities /Saturation Dynamic	SaturationDynamic <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Discrete /Discrete-Time Integrator (See “Discrete-Time Integrator Requirements and Restrictions” on page 6-36.)	built-in/ DiscreteIntegrator	DiscreteTimeIntegratorRTW <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Discontinuities /Saturation	built-in//Saturation	SaturationHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Discrete/Integer Delay	simulink/ Discrete/Integer Delay	IntegerDelayHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline, ResetType
simulink/Discrete/Memory	built-in/Memory	MemoryHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Discrete/Tapped Delay	simulink/Discrete/ Tapped Delay	TappedDelayHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline, ResetType
simulink/Discrete/ Zero-Order Hold	built-in/ZeroOrderHold	ZeroOrderHoldHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Logic and Bit Operations/Bit Clear	simulink/Logic and Bit Operations/Bit Clear	BitOpsHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline

Simulink Block	Blockscope	Implementations and Parameters
simulink/Logic and Bit Operations/Bit Set	simulink/Logic and Bit Operations/Bit Set	BitOpsHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Logic and Bit Operations/Bitwise Operator	simulink/Logic and Bit Operations/Bitwise Operator	BitOpsHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Logic and Bit Operations/Compare To Constant	simulink/Logic and Bit Operations/Compare To Constant	CompareToConstHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Logic and Bit Operations/Extract Bits	simulink/Logic and Bit Operations/Extract Bits	ExtractBits <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Logic and Bit Operations/Compare To Zero	simulink/Logic and Bit Operations/Compare To Zero	CompareToZeroHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Logic and Bit Operations/Shift Arithmetic	simulink/Logic and Bit Operations/Shift Arithmetic	BitOpsHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Lookup Tables/Lookup Table (See “Lookup Table Requirements and Restrictions” on page 6-37.)	built-in/Lookup	LookupHDLInstantiation LookupHDLEmission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.
simulink/Math Operations/Abs	built-in/Abs	AbsHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline

Simulink Block	Blockscope	Implementations and Parameters
simulink/Math Operations/Add	built-in/Sum	<p>SumRTW</p> <p>SumTreeHDLEmission</p> <p>SumLinearHDLEmission</p> <p>SumCascade</p> <p>SumCascadeHDLEmission</p> <p><i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.</p> <p>Note: SumTreeHDLEmission and SumCascade are supported for Add blocks having a single vector input with multiple elements.</p>
simulink/Math Operations/Assignment	built-in/Assignment	<p>AssignmentHDLEmission</p> <p><i>Parameters:</i> OutputPipeline, InputPipeline</p>
simulink/Math Operations/Complex to Real-Imag	built-in /ComplexToRealImag	<p>ComplexToRealImag (<i>default</i>)</p> <p>ComplexToRealImagHDLEmission</p> <p><i>Parameters:</i> OutputPipeline, InputPipeline</p>
simulink/Math Operations/Divide (See “Divide Block Implementations” on page 6-31 for information on computation of reciprocal.)	built-in/Product	<p>ProductRTW</p> <p>ProductLinearHDLEmission</p> <p><i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.</p>

Simulink Block	Blockscope	Implementations and Parameters
simulink/Math Operations/Divide/reciprocal (The reciprocal operation is a special case, supporting two implementations, as described in “Divide Block Implementations” on page 6-31.)	built-in/Product	ProductLinearHDL Emission RecipNewtonHDL Emission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.
simulink/Math Operations/Math Function (sqrt, reciprocal, conj, hermitian, transpose)	built-in/Math	See “Math Function Block Implementations” on page 6-27.
simulink/Math Operations/Matrix Concatenate	built-in/Concatenate	MuxHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Math Operations/MinMax	built-in/MinMax	MinMaxTree MinMaxCascade MinMaxCascadeHDL Emission MinMaxTreeHDL Emission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.

Simulink Block	Blockscope	Implementations and Parameters
simulink/Math Operations/Product of Elements	built-in/Product	ProductRTW ProductTree ProductTreeHDLEmission ProductLinearHDLEmission ProductCascade ProductCascadeHDLEmission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.
simulink/Math Operations/Real-Imag to Complex	built-in /RealImagtoComplex	RealImagtoComplex (<i>default</i>) RealImagtoComplexHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Math Operations/Reshape	simulink/Math Operations/Reshape	PassThroughHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Math Operations/Sign	built-in/Signum	SignumHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Math Operations/Subtract	built-in/Sum	SumRTW SumTree SumTreeHDLEmission SumLinearHDLEmission SumCascade SumCascadeHDLEmission <i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.

Simulink Block	Blockscope	Implementations and Parameters
		<p>Note: SumTree and SumCascade are supported for Subtract blocks having a single vector input with multiple elements.</p>
simulink/Math Operations/Sum of Elements	built-in/Sum	<p>SumRTW SumTree SumTreeHDL Emission SumLinearHDL Emission SumCascade SumCascadeHDL Emission</p> <p><i>Parameters:</i> All implementations support OutputPipeline, InputPipeline.</p> <p>Note: SumTree and SumCascade are supported for Sum of Elements blocks having a single vector input with multiple elements.</p>
simulink/Math Operations/Unary Minus	simulink/Math Operations/Unary Minus	<p>UnaryMinusHDL Emission</p> <p><i>Parameters:</i> OutputPipeline, InputPipeline</p>
simulink/Math Operations/Vector Concatenate	built-in/Concatenate	<p>MuxHDL Emission</p> <p><i>Parameters:</i> OutputPipeline, InputPipeline</p>
simulink/Model Verification/Assertion	built-in/Assertion	NoHDL Emission
simulink/Model Verification/Check Discrete Gradient	simulink/Model Verification/Check Discrete Gradient	NoHDL Emission

Simulink Block	Blockscope	Implementations and Parameters
simulink/Model Verification/Check Dynamic Gap	simulink/Model Verification/Check Dynamic Gap	NoHDLEmission
simulink/Model Verification/Check Dynamic Lower Bound	simulink/Model Verification/Check Dynamic Lower Bound	NoHDLEmission
simulink/Model Verification/Check Dynamic Range	simulink/Model Verification/Check Dynamic Range	NoHDLEmission
simulink/Model Verification/Check Dynamic Upper Bound	simulink/Model Verification/Check Dynamic Upper Bound	NoHDLEmission
simulink/Model Verification/Check Input Resolution	simulink/Model Verification/Check Input Resolution	NoHDLEmission
simulink/Model Verification/Check Static Gap	simulink/Model Verification/Check Static Gap	NoHDLEmission
simulink/Model Verification/Check Static Lower Bound	simulink/Model Verification/Check Static Lower Bound	NoHDLEmission
simulink/Model Verification/Check Static Range	simulink/Model Verification/Check Static Range	NoHDLEmission
simulink/Model Verification/Check Static Upper Bound	simulink/Model Verification/Check Static Upper Bound	NoHDLEmission
simulink/Model-Wide Utilities/DocBlock	simulink/Model-Wide Utilities/DocBlock	DocBlockHDLEmission NoHDLEmission

Simulink Block	Blockscope	Implementations and Parameters
simulink/Ports & Subsystems/Enable (See “Code Generation for Enabled Subsystems” on page 10-11.)	built-in/Enable	EnablePort
simulink/Ports & Subsystems/Model	built-in/ModelReference	ModelReferenceHDLInstantiation <i>Parameters:</i> See “Interface Generation Parameters” on page 6-54.
simulink/Signal Attributes/Data Type Duplicate	simulink/Signal Attributes/Data Type Duplicate	NoHDL Emission
simulink/Signal Attributes/Data Type Propagation	simulink/Signal Attributes/Data Type Propagation	NoHDL Emission
simulink/Signal Attributes/Rate Transition	built-in/RateTransition	RateTransitionHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Signal Attributes/Signal Conversion	built-in/SignalConversion	PassThroughHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Signal Attributes/Signal Specification	built-in/ SignalSpecification	SignalSpecificationHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Signal Routing/Index Vector	built-in/MultiPortSwitch	MultiPortSwitchHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Signal Routing/Multiport Switch	built-in/MultiPortSwitch	MultiPortSwitchHDL Emission <i>Parameters:</i> OutputPipeline, InputPipeline

Simulink Block	Blockscope	Implementations and Parameters
simulink/Signal Routing/Selector	built-in/Selector	SelectorHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Sinks/Display	built-in/Display	NoHDLEmission
simulink/Sinks/Floating Scope	built-in/Scope	NoHDLEmission
simulink/Sinks/Stop Simulation	built-in/Stop	NoHDLEmission
simulink/Sinks/To File	built-in/ToFile	NoHDLEmission
simulink/Sinks/To Workspace	built-in/ToWorkspace	NoHDLEmission
simulink/Sinks/XY Graph	simulink/Sinks/XY Graph	NoHDLEmission
simulink/Sources/Counter Free-Running	simulink/Sources/Counter Free-Running	CounterFreeRunningHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/Sources/Counter Limited	simulink/Sources/Counter Limited	CounterLimitedHDLEmission <i>Parameters:</i> OutputPipeline, InputPipeline
simulink/User-Defined Functions/Embedded MATLAB Function (See Chapter 12, “Generating HDL Code with the Embedded MATLAB Function Block”.)	simulink/User-Defined Functions/Embedded MATLAB Function	hdlstateflow .StateflowHDLInstantiation <i>Parameters:</i> OutputPipeline, InputPipeline, DistributedPipelining

Blocks with Multiple Implementations

In this section...

“Overview” on page 6-22

“Implementations for Commonly Used Blocks” on page 6-23

“Math Function Block Implementations” on page 6-27

“Divide Block Implementations” on page 6-31

“Subsystem Interfaces and Special-Purpose Implementations” on page 6-33

“A Note on Cascade Implementations” on page 6-34

Overview

The tables in this section summarize the block types that have multiple implementations. The Implementations column gives the `package.class` specification you should use in your control files. The Description column summarizes the trade-offs involved in choosing different implementations.

The coder provides a default HDL block implementation for all supported blocks. If you want to use the default implementation, you do not usually need to specify it explicitly in a control file. However, the following example illustrates a situation in which the default implementation is specified as an exception for one particular block:

```
% 1. Use default (multipliers) Gain block implementation
% for one specific Gain block within OneD_DCT8 subsystem.
c.forEach('dct8_fixed/OneD_DCT8/Gain14',...
    'built-in/Gain', {},...
    'hdldefaults.GainMultHDL Emission');
% 2. Use factored CSD Gain block implementation
% or all other Gain blocks at or below
% level of OneD_DCT8 subsystem.
c.forEach('dct8_fixed/OneD_DCT8/*',...
    'built-in/Gain', {},...
    'hdldefaults.GainFCSDHDL Emission');
```


Implementations for Commonly Used Blocks

Built-In/Constant

Implementations	Parameters	Description
hdldefaults. Constant	Unspecified (<i>Default</i>)	This implementation emits the value of the Constant block.
	{'OutputPipeline', NStages}	See “OutputPipeline” on page 6-53.
hdldefaults. ConstantSpecialHDL Emission	Unspecified (<i>Default</i>)	By default, this implementation emits the character 'Z' for each bit in the signal. For example, for a 4-bit signal, 'ZZZZ' would be emitted.
	{'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, 'ZZZZ' would be emitted.
	{'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, 'XXXX' would be emitted.
	{'OutputPipeline', NStages}	See “OutputPipeline” on page 6-53.

Note

`hdldefaults.ConstantSpecialHDL Emission` does not support the double data type. If this implementation is specified for a Constant of type double, an error will result at code generation time.

Built-In/Gain

Implementations	Description
<code>hdldefaults.GainMultHDL Emission</code>	<i>Default.</i> This implementation retains multiplier operations in HDL code generated by the Gain block.
<code>hdldefaults.GainCSDHDL Emission</code>	This implementation decreases the area used by the model while maintaining or increasing clock speed, using canonic signed digit (CSD) techniques. CSD replaces multiplier operations with shift and add operations. CSD minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.
<code>hdldefaults.GainFCSDHDL Emission</code>	This implementation lets you achieve a greater area reduction than CSD, at the cost of decreasing clock speed. This implementation uses factored CSD techniques, which replace multiplier operations with shift and add operations on prime factors of the operands.

Built-In/Lookup Table

Implementations	Description
<code>hdldefaults.LookupHDL Emission</code>	<i>Default.</i> Nonhierarchical lookup table.
<code>hdldefaults.LookupHDL Instantiation</code>	This implementation generates an additional level of HDL hierarchy (which does not exist in the Simulink model) for the lookup table.

See also “Lookup Table Requirements and Restrictions” on page 6-37.

Signal Processing Blockset/Minimum

Implementation	Description
hdldefaults.MinMaxTree	<i>Default.</i> This implementation is large and slow but has minimal latency.
hdldefaults.MinMaxTreeHDLEmission	This implementation is large and slow but has minimal latency.
hdldefaults.MinMaxCascade	This implementation is optimized for latency * area, with medium speed. See “A Note on Cascade Implementations” on page 6-34.
hdldefaults.MinMaxCascadeHDLEmission	This implementation is optimized for latency * area, with medium speed. See “A Note on Cascade Implementations” on page 6-34.

Signal Processing Blockset/Maximum

Implementation	Description
hdldefaults.MinMaxTree	<i>Default.</i> This implementation is large and slow but has minimal latency.
hdldefaults.MinMaxTreeHDLEmission	This implementation is large and slow but has minimal latency.
hdldefaults.MinMaxCascade	This implementation is optimized for latency * area, with medium speed. See “A Note on Cascade Implementations” on page 6-34.
hdldefaults.MinMaxCascadeHDLEmission	This implementation is optimized for latency * area, with medium speed. See “A Note on Cascade Implementations” on page 6-34.

Built-In/MinMax

Implementation	Description
<code>hdldefaults.MinMaxTree</code>	<i>Default.</i> This implementation is large and slow but has minimal latency.
<code>hdldefaults.MinMaxTreeHDLEmission</code>	This implementation is large and slow but has minimal latency.
<code>hdldefaults.MinMaxCascade</code>	This implementation is optimized for latency * area, with medium speed. See “A Note on Cascade Implementations” on page 6-34.
<code>hdldefaults.MinMaxCascadeHDLEmission</code>	This implementation is optimized for latency * area, with medium speed. See “A Note on Cascade Implementations” on page 6-34.

Built-In/Product

Implementations	Description
<code>hdldefaults.ProductRTW</code>	<i>Default.</i> Generates a chain of N operations (multipliers) for N inputs.
<code>hdldefaults.ProductLinearHDLEmission</code>	Generates a chain of N operations (multipliers) for N inputs.
<code>hdldefaults.ProductTree</code> <code>hdldefaults.ProductTreeHDLEmission</code>	This implementation has minimal latency but is large and slow. It generates a tree-shaped structure of multipliers.
<code>hdldefaults.ProductCascade</code> <code>hdldefaults.ProductCascadeHDLEmission</code>	This implementation optimizes latency * area and is faster than the tree implementation. It computes partial products and cascades multipliers. See “A Note on Cascade Implementations” on page 6-34.

Built-In/Sum

Implementation	Description
<code>hdldefaults.SumRTW</code>	<i>Default.</i> Generates a chain of N operations (adders) for N inputs.

Built-In/Sum (Continued)

Implementation	Description
hdldefaults.SumLinearHDL Emission	Generates a chain of N operations (adders) for N inputs.
hdldefaults.SumTree hdldefaults.SumTreeHDL Emission	This implementation has minimal latency but is large and slow. Generates a tree-shaped structure of adders.
hdldefaults.SumCascade hdldefaults.SumCascadeHDL Emission	This implementation optimizes latency * area and is faster than the tree implementation. It computes partial sums and cascades adders. See “A Note on Cascade Implementations” on page 6-34.

Math Function Block Implementations

The Math Function block `sqrt`, `reciprocal`, `conj`, `hermitian`, and `transpose` functions are supported for HDL code generation.

By specifying an implementation and parameter(s) in your control file, you can choose from among several algorithms for computing these functions. The following tables summarize the available Math Function block implementations and parameters.

simulink/Math Operations/Math Function (sqrt)

Implementations	Parameters	Description
hdldefaults .SqrtBitsetHDL Emission (Default implementation)	{'UseMultiplier', 'on'}	(Default parameter): Compute <code>sqrt</code> using multiply/add algorithm (Simulink default algorithm).
	{'UseMultiplier', 'off'}	Compute <code>sqrt</code> using bitset shift/addition algorithm.
	{'InputPipeline', NStages}	See “InputPipeline” on page 6-52 .
	{'OutputPipeline', NStages}	See “OutputPipeline” on page 6-53.

simulink/Math Operations/Math Function (sqrt) (Continued)

Implementations	Parameters	Description
hdldefaults .SqrtNewtonHDL Emission	{ 'Iterations', N }	Compute sqrt using iterative Newton method. The argument N specifies the number of iterations. The default value for N is 5. The recommended value for N is between 3 and 10. The coder will generate a message if N is outside the recommended range.
	{ 'InputPipeline', NStages }	See “InputPipeline” on page 6-52.
	{ 'OutputPipeline', NStages }	See “OutputPipeline” on page 6-53.

Notes on the sqrt implementations:

- Input must be an unsigned scalar.
- The output is a fixed-point scalar.
- The Math Function block from the hdl1lib library has sqrt selected in its **Function** menu.

simulink/Math Operations/Math Function (reciprocal)

Implementations	Parameters	Description
hdldefaults .RecipDivHDL Emission	Unspecified (<i>Default</i>)	Compute reciprocal as 1/N, using the HDL divide (/) operator to implement the division.
	{ 'InputPipeline', NStages }	See “InputPipeline” on page 6-52.
	{ 'OutputPipeline', NStages }	See “OutputPipeline” on page 6-53.

simulink/Math Operations/Math Function (reciprocal) (Continued)

Implementations	Parameters	Description
hdldefaults .RecipNewtonHDLEmission	{ 'Iterations', N }	<p>Compute reciprocal using iterative Newton method. The argument N specifies the number of iterations.</p> <p>The default value for N is 4.</p> <p>The recommended value for N is between 2 and 10. The coder will generate a message if N is outside the recommended range.</p>
	{ 'InputPipeline', NStages }	See “InputPipeline” on page 6-52.
	{ 'OutputPipeline', NStages }	See “OutputPipeline” on page 6-53.

Notes on the reciprocal implementations:

- 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.

simulink/Math Operations/Math Function (conj)

Implementations	Parameters	Description
hdldefaults .ComplexConjugateHDL Emission	Unspecified (<i>Default</i>)	Compute complex conjugate. See Math Function in the Simulink documentation.
	{ 'InputPipeline', NStages }	See “InputPipeline” on page 6-52.
	{ 'OutputPipeline', NStages }	See “OutputPipeline” on page 6-53.

simulink/Math Operations/Math Function (hermitian)

Implementations	Parameters	Description
hdldefaults .HermitianHDL Emission	Unspecified (<i>Default</i>)	Compute hermitian. See Math Function in the Simulink documentation.
	{ 'InputPipeline', NStages }	See “InputPipeline” on page 6-52 .
	{ 'OutputPipeline', NStages }	See “OutputPipeline” on page 6-53.

simulink/Math Operations/Math Function (transpose)

Implementations	Parameters	Description
hdldefaults .TransposeHDL Emission	Unspecified (<i>Default</i>)	Compute array transpose. See Math Function in the Simulink documentation.
	{ 'InputPipeline', NStages }	See “InputPipeline” on page 6-52 .
	{ 'OutputPipeline', NStages }	See “OutputPipeline” on page 6-53.

simulink/Math Operations/Math Function (parent class)

Implementations	Parameters	Description
hdldefaults .MathFunctionHDLEmission	Unspecified (<i>Default</i>)	Use the default implementation for the function (sqrt,reciprocal, or conj) selected on the block.
	{ 'UseMultiplier', 'on' } (use with sqrt only)	If the function selected on the block is sqrt, compute sqrt using multiply/add algorithm (Simulink default algorithm). If the function selected on the block is not sqrt, an error results.
	{ 'UseMultiplier', 'off' } (use with sqrt only)	If the function selected on the block is sqrt, compute sqrt using bitset shift/addition algorithm. If the function selected on the block is not sqrt, an error results.
	{ 'InputPipeline', NStages }	See “InputPipeline” on page 6-52.
	{ 'OutputPipeline', NStages }	See “OutputPipeline” on page 6-53.

Divide Block Implementations

The Divide block normally supports the hdldefaults.ProductLinearHDLEmission implementations.

However, the reciprocal operation of the Divide block is a special case. When the reciprocal operation is selected, the Divide block supports the implementations described in the following table.

simulink/Math Operations/Divide (reciprocal computation only)

Implementations	Parameters	Description
hdldefaults .ProductRTW (Default implementation)	Unspecified (Default)	When computing a reciprocal, compute 1/N using the HDL divide (/) operator to implement the division.
	{'InputPipeline', NStages}	See “InputPipeline” on page 6-52.
	{'OutputPipeline', NStages}	See “OutputPipeline” on page 6-53.
hdldefaults .ProductLinearHDL Emission'	Unspecified (Default)	When computing a reciprocal, compute 1/N using the HDL divide (/) operator to implement the division.
	{'InputPipeline', NStages}	See “InputPipeline” on page 6-52.
	{'OutputPipeline', NStages}	See “OutputPipeline” on page 6-53.
hdldefaults .RecipNewtonHDL Emission	{'Iterations', N}	When computing a reciprocal, use iterative Newton method. The argument N specifies the number of iterations. The default value for N is 4. The recommended value for N is between 2 and 10. The coder will generate a message if N is outside the recommended range.
	{'InputPipeline', NStages}	See “InputPipeline” on page 6-52.
	{'OutputPipeline', NStages}	See “OutputPipeline” on page 6-53.

Notes on the reciprocal implementations:

- 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.

Subsystem Interfaces and Special-Purpose Implementations

Built-In/SubSystem

Implementation	Description
hdldefaults.SubsystemBlackBoxHDLInstantiation	<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>
hdldefaults.NoHDL Emission	<p>This implementation completely removes the subsystem from the generated code. This lets you use a subsystem in simulation but treat it as a “no-op” in the HDL code.</p>

For more information on subsystem implementations, see Chapter 10, “Interfacing Subsystems and Models to HDL Code”.

Special-Purpose Implementations

Implementation	Description
<code>hdldefaults.PassThroughHDL Emission</code>	Provides a pass-through implementation in which the block's inputs are passed directly to its outputs. (In effect, the block becomes a wire in the HDL code.) Several blocks are supported with a pass-through implementation.
<code>hdldefaults.NoHDL Emission</code>	This implementation completely removes the block from the generated code. This lets you use the block in simulation but treat it as a “no-op” in the HDL code. This implementation is used for many blocks (such as Scopes and Assertions) that are significant in simulation but would be meaningless in HDL code. You can also use this implementation as an alternative implementation for subsystems.

For more information related to special-purpose implementations, see Chapter 10, “Interfacing Subsystems and Models to HDL Code”.

A Note on Cascade Implementations

Cascade implementations are available 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 for HDL Code Generation

In this section...

“Block Usage, Requirements, and Restrictions” on page 6-35

“Restrictions on Use of Blocks in the Test Bench” on page 6-40

Block Usage, Requirements, and Restrictions

This section discusses requirements and restrictions that apply to the use of specific block types in HDL code generation.

Biquad Filter Block Requirements and Restrictions

- 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.

Data Type Conversion Block Requirements and Restrictions

If a Data Type Conversion block is configured for double to fixed-point or fixed-point to double conversion, a warning is displayed during code generation.

Digital Filter Block Requirements and Restrictions

- When the Digital Filter block **Discrete-time filter object** option is selected, Filter Design Toolbox software is required to generate code for the block.
- The Digital Filter block **Input port(s)** option is not supported for HDL code generation.
- The Digital Filter block supports complex data for fully parallel FIR and CIC structures only. See “Complex Coefficients and Data Support for the Digital Filter Block” on page 6-60.

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 resets is not supported for HDL code generation. Set **External reset** to none.
- 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.

LMS Filter Usage and Restrictions

Restrictions.

- 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.

Usage Note. By default, the LMS Filter implementation (LMSFilterHDL Emission) 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

Lookup Table Requirements and Restrictions

The coder does not support the **Lookup method** options (such as Interpolation-Extrapolation) displayed on the Lookup Table block GUI. Generated HDL code assumes the existence of a full table.

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 **Multirate filter object (MFILT)** is selected:
 - 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 **Multirate filter object (MFILT)** is selected, 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 **Dialog parameters** is selected, the following fixed-point options are not supported for HDL coder generation:
 - Slope and Bias scaling
 - Inherit via internal rule

For the Multirate CIC Decimation block:

- When **Multirate filter object (MFILT)** is selected, 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 **Dialog parameters** is selected, 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 **Multirate filter object (MFILT)** is selected:
 - 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 **Multirate filter object (MFILT)** is selected, 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 **Dialog parameters** is selected, the following fixed-point options are not supported for HDL coder generation:
 - **Coefficients:** Slope and Bias scaling
 - **Product Output:** Inherit via internal rule

For the Multirate CIC Interpolation block:

- When **Multirate filter object (MFILT)** is selected, 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 **Dialog parameters** is selected, the **Filter Structure** option Zero-latency interpolator is not supported for HDL code generation. Select Interpolator in the **Filter Structure** drop-down menu.

NCO Block Requirements and Restrictions

Inputs:

- The phase increment and phase offset support only integer or fixed-point data types.
- The phase increment and phase offset can be either scalars or vectors.

Outputs:

- Only fixed point data types are supported for the quantization error (Qerr) port and output signals.

Parameters:

- **Add internal dither** is not supported 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 if 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 if there are fewer than 4 accumulator bits.

PN Sequence Generator Block Requirements and Restrictions

This block requires Communications Blockset.

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.

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 type `single` or `double`.

Restrictions on Use of Blocks in the Test Bench

In a model intended for use in HDL code generation, the DUT is typically modeled as a subsystem at the top level of the model, driven by other blocks or subsystems at the top level. These components make up the test bench.

Blocks that belong to the blocksets and toolboxes in the following list should not be directly connected to the DUT at the top level of the model. Instead, they should be placed in a subsystem, which is then connected to the DUT. All blocks in the following blocksets are subject to this restriction:

- RF Blockset™
- SimDriveline™
- SimEvents®
- SimMechanics™
- SimPowerSystems™
- Simscape™

Block Implementation Parameters

In this section...

“Overview” on page 6-41

“CoeffMultipliers” on page 6-41

“Distributed Arithmetic Implementation Parameters for Digital Filter Block” on page 6-42

“InputPipeline” on page 6-52

“OutputPipeline” on page 6-53

“ResetType” on page 6-53

“Interface Generation Parameters” on page 6-54

Overview

Block implementation parameters let you control details of the code generated for specific block implementations. Block implementation parameters are passed to `forEach` or `forAll` calls (see “`forEach`” on page 5-7) as cell arrays of property/value pairs of the form

```
{'PropertyName', value}
```

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.

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 in a control file using the following syntax:

- `{'CoeffMultipliers', '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 decreases the area used by the filter while maintaining or increasing clock speed.

- {'CoeffMultipliers', '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.
- {'CoeffMultipliers', 'multipliers'} (default): Retain multiplier operations.

The coder supports `CoeffMultipliers` for the filter block implementations shown in the following table:

Block	Implementation
dsparch4/Digital Filter	hdldefaults.DigitalFilterHDLInstantiation
dspmlti4/FIR Decimation	hdldefaults.FIRDecimationHDLInstantiation
dspmlti4/FIR Interpolation	hdldefaults.FIRInterpolationHDLInstantiation
dsparch4/Biquad Filter	hdldefaults.BiquadFilterHDLInstantiation

The following `forEach` call specifies that code generated for all FIR Decimation blocks in the model will use the CSD optimization:

```
config.forEach('*',...
    'dspmlti4/FIR Decimation', {},...
    'hdldefaults.FIRDecimationHDLInstantiation,...
    {'CoeffMultipliers', 'csd'});
```

Distributed Arithmetic Implementation Parameters for Digital Filter Block

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 single-rate FIR structures of the Digital Filter block, as given in the following table.

Block	Implementation	FIR Structures That Support DA
dsparch4/Digital Filter	hdldefaults. DigitalFilterHDLInstantiation	<ul style="list-style-type: none"> • dfilt.dffir • dfilt.dfsymfir • dfilt.dfasymdir

This section briefly summarizes the operation of DA. 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

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 pre-adders.

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 performance at the expense of area. You can control the degree of parallelism by specifying the *DARadix* implementation parameter in a control file. *DARadix* lets you specify the number of bits processed simultaneously in DA (see “*DARadix* Implementation Parameter” on page 6-51).

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. The reduction is significant.

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 in a control file (see “`DALUTPartition Implementation Parameter`” on page 6-46).

Requirements and Considerations for Generating Distributed Arithmetic Code

You can control how DA code is generated by using the `DALUTPartition` and `DARadix` implementation parameters in a control file. 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 6-46
- “`DARadix Implementation Parameter`” on page 6-51

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.

In distributed arithmetic the product and accumulator operations are merged, and computations are done at full precision. The **Product output** and **Accumulator** properties of the Digital Filter block are ignored and set to full precision.

DALUTPartition Implementation Parameter

Syntax: 'DALUTPartition', [p1 p2... pN]

DALUTPartition enables DA code generation and specifies the number and size of LUT partitions used for DA.

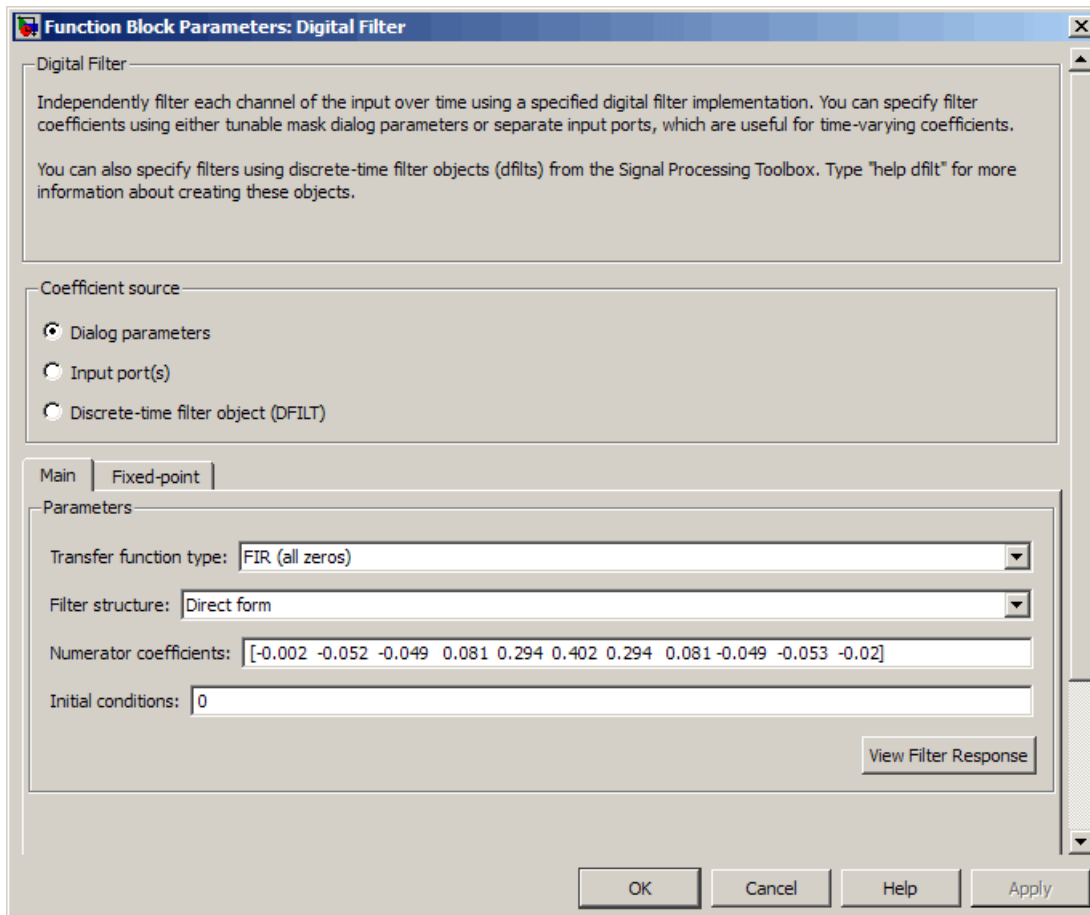
Specify LUT partitions as a vector of integers [p1 p2...pN] where:

- 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 6-46.)

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.

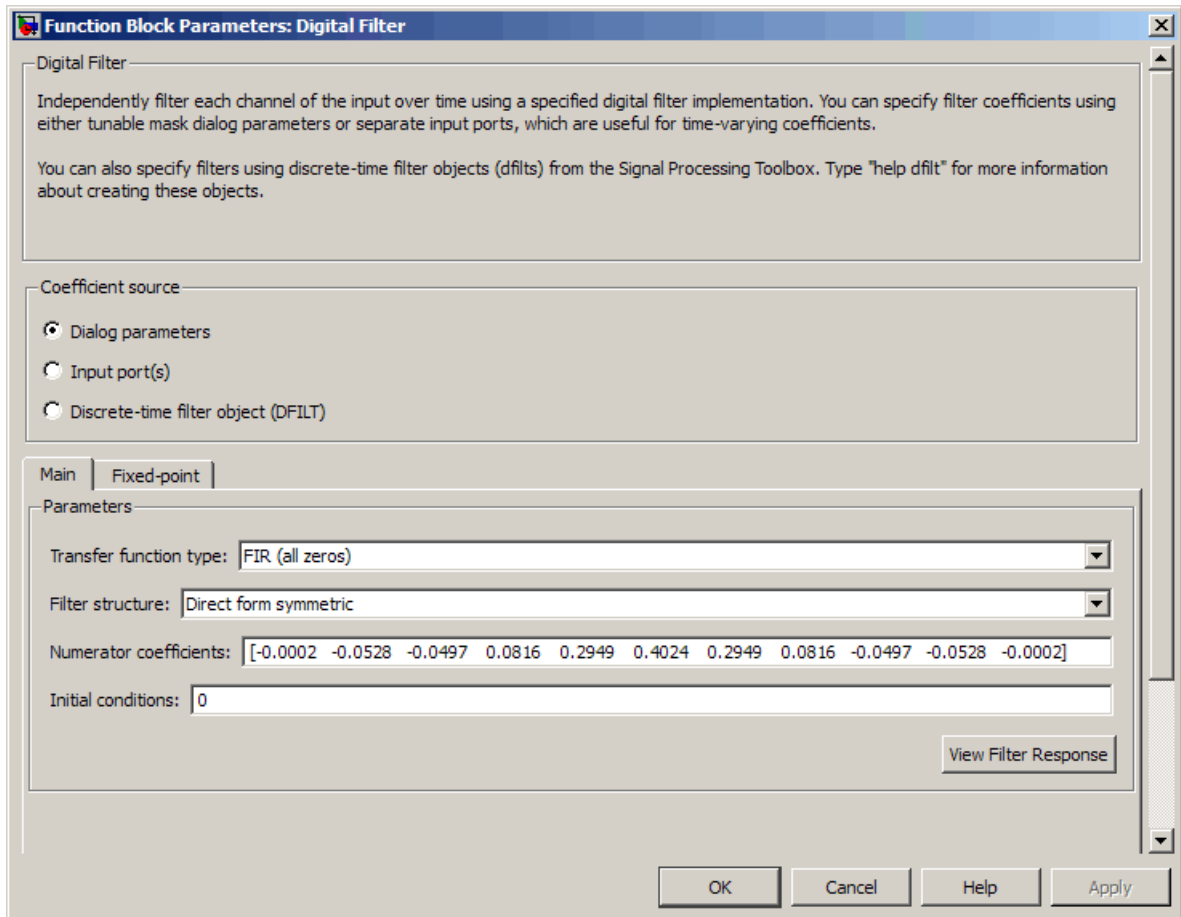


The following control file defines one possible LUT partitioning for this filter:

```
function c = filter_da_config1
c = hdlnewcontrol(mfilename);

c.forEach('*',...
'dsparch4/Digital Filter', {},...
'hdldefaults.DigitalFilterHDLInstantiation', {'DALUTpartition',[4 4 3]});
```

The following figure shows a Digital Filter configured for a direct-form symmetric FIR filter of length 6:



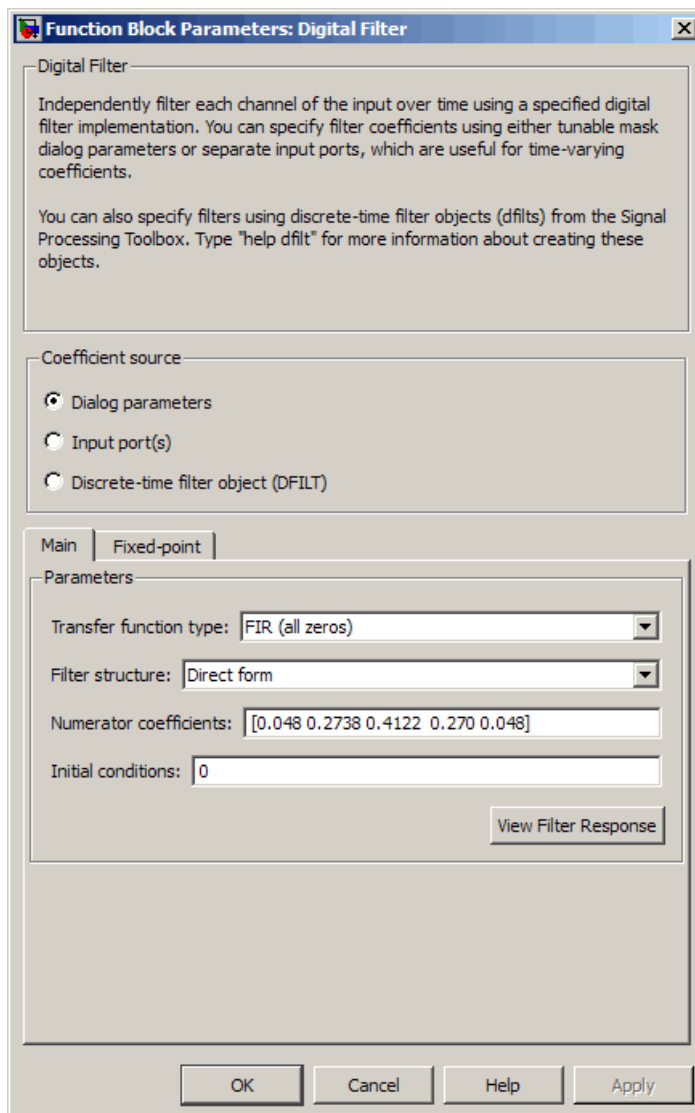
The following control file defines a possible LUT partitioning for this filter.

```
function c = filter_da_config1
c = hdlnewcontrol(mfilename);

c.forEach('*',...
'dsparch4/Digital Filter', {},...
```

```
'hdldefaults.DigitalFilterHDLInstantiation', {'DALutpartition',[3 3]});
```

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.



The following control file specifies a partition that is equal to the filter length:

```
function c = filter_da_config1
c = hdlnewcontrol(mfilename);
```

```
c.forEach('*',...
'dsparch4/Digital Filter', {},...
'hdldefaults.DigitalFilterHDLInstantiation', {'DALutpartition',5});
```

DARadix Implementation Parameter

Syntax: 'DARadix', N

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.

Note When setting a DARadix value for symmetrical (dfilt.dfsymfir) and asymmetrical (dfilt.dfasymfir) filters, see “Considerations for Symmetrical and Asymmetrical Filters” on page 6-51.

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` 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.

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.

Syntax:

```
{'InputPipeline', nStages}
```

where `nStages` ≥ 0 .

The following `forEach` call specifies an input pipeline depth of two stages for all `Sum` blocks in the model:

```
config.forEach('*', ...  
    'built-in/Sum', {}, ...  
    'hdldefaults.SumRTW', {'InputPipeline', 2});
```

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 Coder** 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.

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.

Syntax:

```
{'OutputPipeline', nStages}
```

where `nStages` \geq 0.

The following `forEach` call specifies an output pipeline depth of two stages for all Sum blocks in the model:

```
config.forEach('*',...
    'built-in/Sum', {},...
    'hdldefaults.SumRTW', {'OutputPipeline', 2});
```

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 Coder** 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.

See also “Distributed Pipeline Insertion” on page 12-58.

ResetType

The `ResetType` implementation parameter lets you suppress generation of reset logic for the following block types:

- `dpsigops/Delay`
- `simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Enabled`

- simulink/Commonly Used Blocks/Unit Delay
- simulink/Discrete/Integer Delay
- simulink/Discrete/Tapped Delay

Syntax:

```
{ 'ResetType', 'default' }  
{ 'ResetType', 'none' }
```

When you specify { 'ResetType', '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.

The default specification is { 'ResetType', 'default' }. In this case, the coder follows the Global Settings/Advanced **Reset type** option for the specified blocks.

The following control file specifies suppression of reset logic for a specific unit delay block within a subsystem.

```
function c = resetnone_examp  
  
% Control file for resetnone_examp  
c = hdlnewcontrol(mfilename);  
c.generateHDLFor('resetnone_examp/HDLSubsystem');  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% Suppress reset logic for Unit Delay block  
  
c.forEach('resetnone_examp/HDLSubsystem/Unit Delay',...  
    'built-in/UnitDelay', {},...  
    'hdldefaults.UnitDelayRTW', {'ResetType','none'});
```

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
- lflinklib/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 “Customizing the Generated Interface” on page 10-15.

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 supported by the coder. These blocks are listed 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/Digital Filter	Fully parallel FIR and CIC structures support complex data. See “Complex Coefficients and Data Support for the Digital Filter Block” on page 6-60.
dspindex/Multiport Selector	
dspsigattrs/Convert 1-D to 2-D	

Simulink Block	Restrictions
dspsigattrs/Frame Conversion	
dspsigops/Delay	Only DSPDelayHDL Emission implementation supports complex data.
dspsigops/Downsample	
dspsigops/NCO	
dspsigops/Upsample	
dspsrcs4/DSP Constant	
dspsrcs4/Sine Wave	
hdl demolib/Dual Port RAM	
hdl demolib/Simple Dual Port RAM	
hdl demolib/Single Port RAM	
hdl demolib/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/Commonly Used Blocks/Gain	
simulink/Commonly Used Blocks/Ground	
simulink/Commonly Used Blocks/Product	

Simulink Block	Restrictions
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/Integer 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/Lookup Table	
simulink/Math Operations/Add	
simulink/Math Operations/Assignment	
simulink/Math Operations/Complex to Real-Imag	
simulink/Math Operations/Unary Minus	

Simulink Block	Restrictions
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 ProductLinearHDLemission 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 SumLinearHDLemission implementation supports complex data.
simulink/Math Operations/Sum of Elements	Only SumLinearHDLemission implementation supports complex data.
simulink/Math Operations/Vector Concatenate	
simulink/Signal Attributes/Rate Transition	
simulink/Signal Attributes/Signal Conversion	
simulink/Signal Attributes/Signal Specification	
simulink/Signal Routing/Index Vector	
simulink/Signal Routing/Multiport Switch	

Simulink Block	Restrictions
simulink/Signal Routing/Selector	
simulink/User-Defined Functions/Embedded MATLAB Function	See also “Using Complex Signals” on page 12-49.

Complex Coefficients and Data Support for the Digital Filter Block

The coder supports use of complex coefficients and complex input signals for fully parallel FIR and CIC filter structures of the Digital Filter block. 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
mfilt.cicdecim	Y	N/A	N/A
mfilt.cicinterp	Y	N/A	N/A
mfilt.firdecim	Y	Y	N
mfilt.firinterp	Y	Y	N

The hdl demolib Block Library

- “Accessing the hdl demolib Library Blocks” on page 7-2
- “RAM Blocks” on page 7-4
- “HDL Counter” on page 7-15
- “HDL FFT” on page 7-27
- “Bitwise Operators” on page 7-35

Accessing the hdl demolib Library Blocks

The `hdl demolib` library provides HDL-specific block implementations supporting simulation and code generation for:

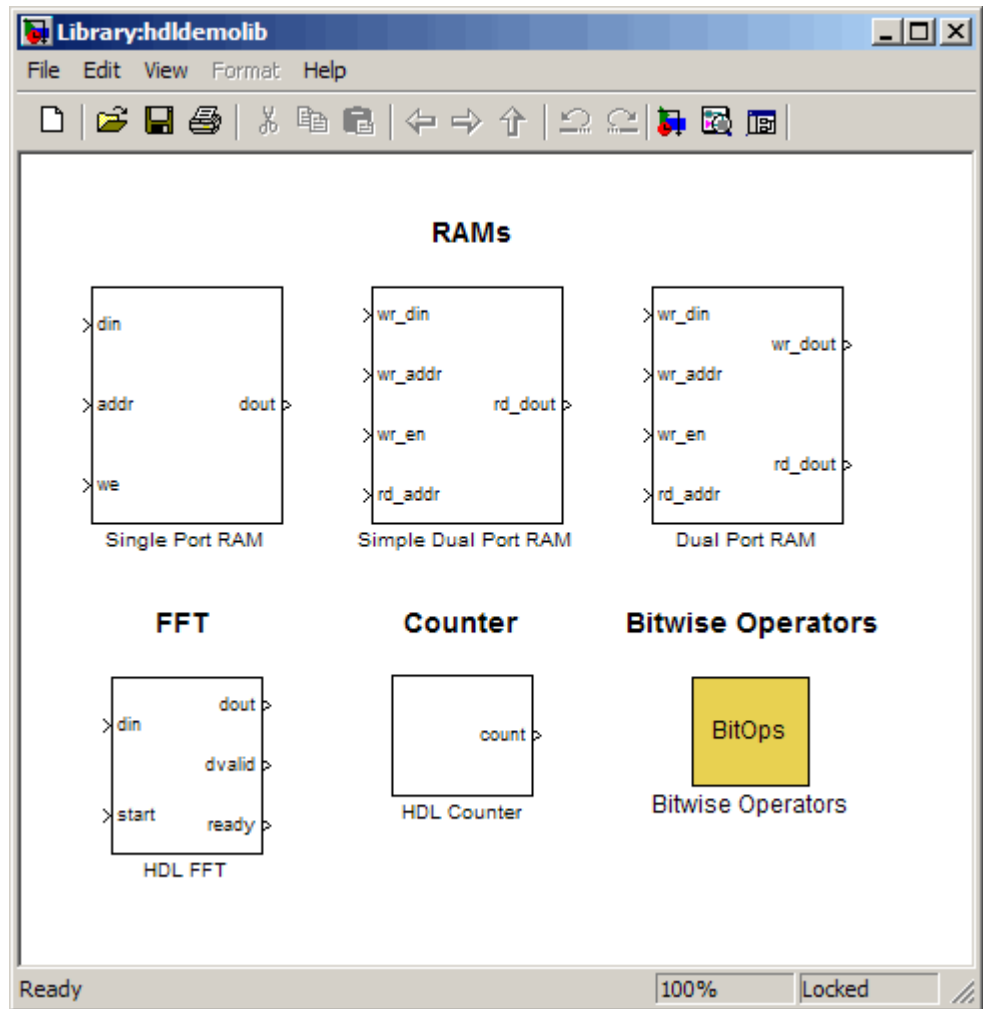
- Single and dual-port RAMs
- Counter with single-shot and free-running modes
- Minimum resource FFT
- Operations on bits and bit fields

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
```

The following figure shows the top-level `hdl demolib` library window.



RAM Blocks

In this section...
“Overview of RAM Blocks” on page 7-4
“Dual Port RAM Block” on page 7-6
“Simple Dual Port RAM Block” on page 7-7
“Single Port RAM Block” on page 7-9
“Code Generation with RAM Blocks” on page 7-12
“Generic RAM and ROM Demos” on page 7-13
“Limitations for RAM Blocks” on page 7-13

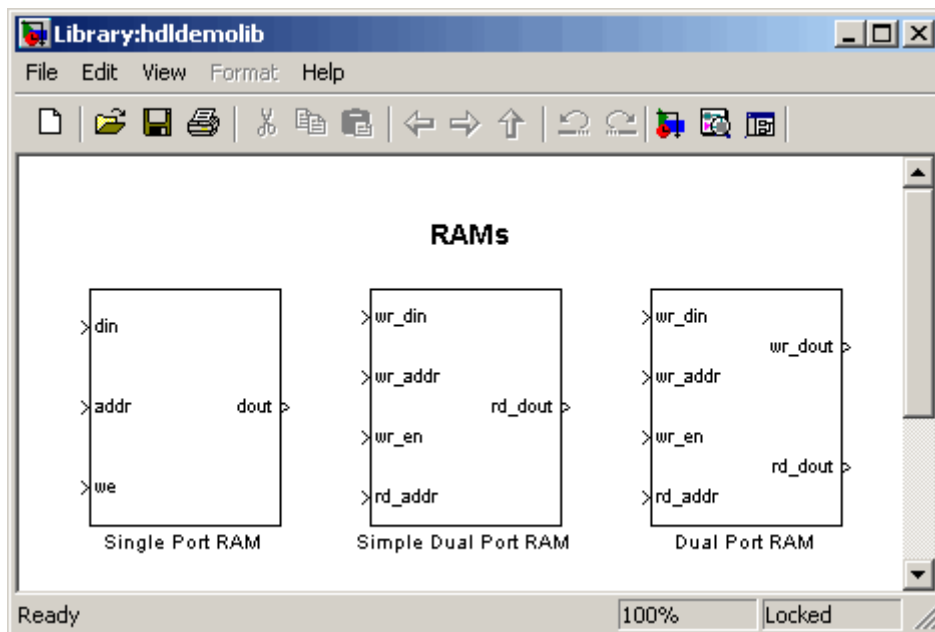
Overview of RAM Blocks

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:

```
hdl demolib
```

Then, drag the desired RAM block from the `hdl demolib` library to your model, and set the block parameters and connect signals following the guidelines in the following sections.

RAM Block Demo

The RAM-Based FIR Filter demo (`hdl coderfirram.mdl`) provides an example of VHDL code generation for a Dual Port RAM block. Run this demo to acquaint yourself with the generated code.

The HDL device under test (DUT) in the model is the `FIR_RAM` subsystem. The `FIR_RAM` subsystem contains a Dual Port RAM block. The entity and architecture definitions generated for this block are written to `DualPortRAM_Inst0.vhd`.

The code generated for the top-level DUT, `FIR_RAM.vhd`, contains the component instantiation for the Dual Port RAM block.

Dual Port RAM Block

Dual Port RAM Block Ports and Parameters

The following figure shows the Dual Port RAM block.



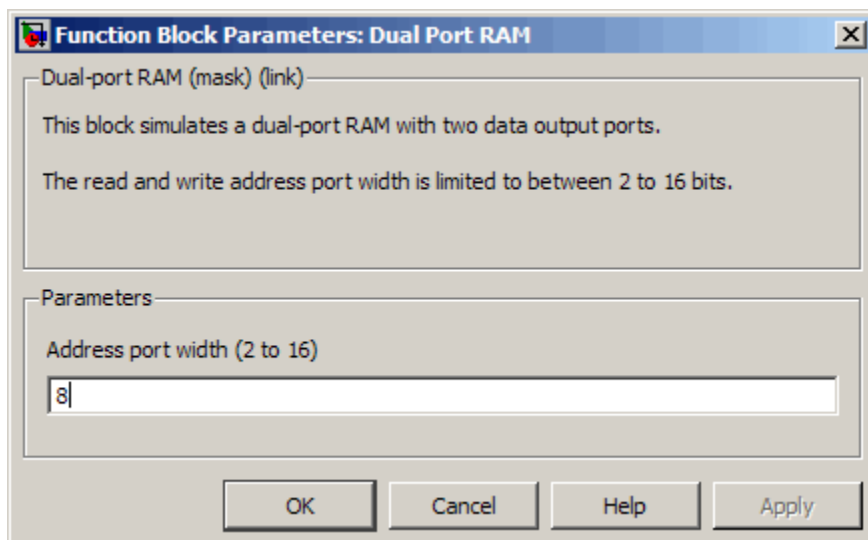
The block has the following input and output ports:

- `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 desired 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 16 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.

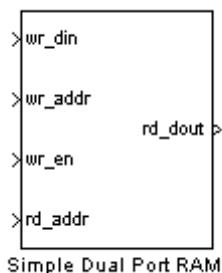
Read-During-Write Behavior

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`).

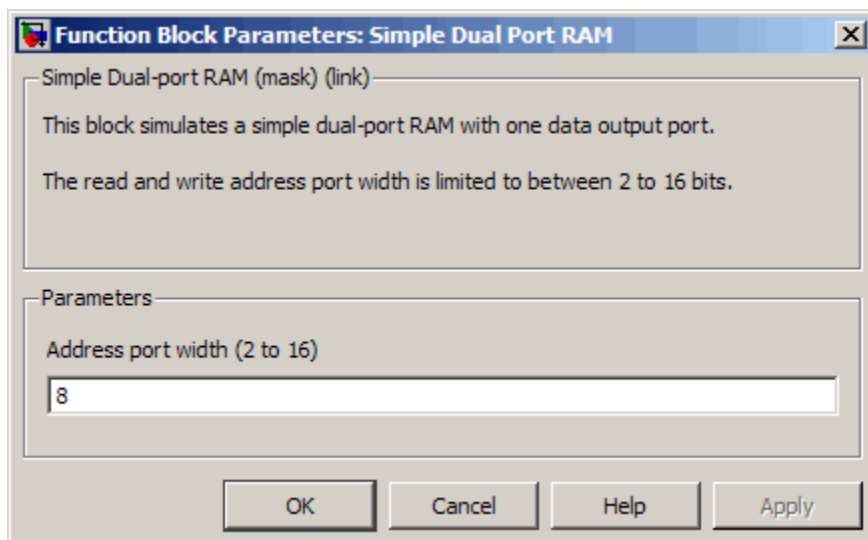
The block has the following input and output ports:

- `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 desired 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 16 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.



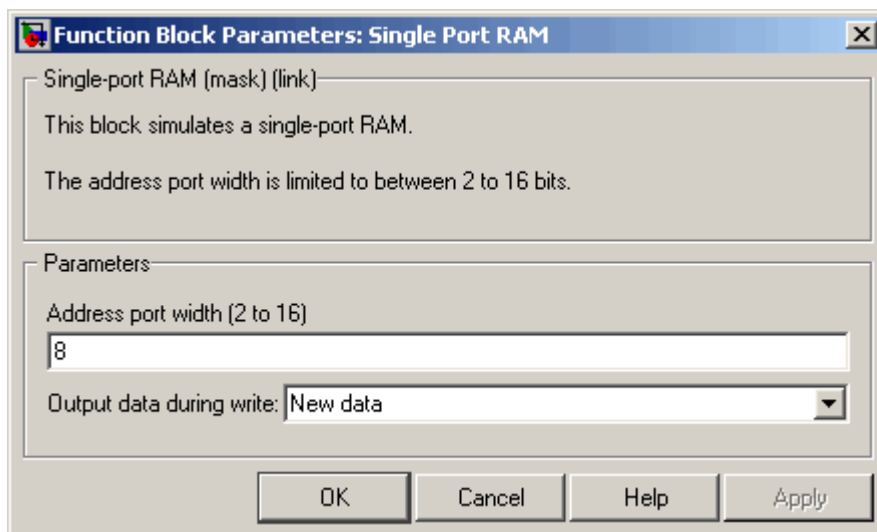
The block has the following input and output ports:

- **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 desired 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 16 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** drop-down 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 7-13 for further information on read-during-write behavior in hardware.

Code Generation with RAM Blocks

The following general considerations apply to code generation for any of the RAM blocks:

- Code generated for a RAM block is generated to a separate file in the target directory. The naming convention for this file is *blockname.ext*, where *blockname* is derived from the name assigned to the RAM block, and *ext* is the target language filename extension.
- RAM blocks are implemented as subsystems, primarily for use in simulation. The coder generates a top-level interface (entity and RTL architecture) for the block; code is not generated for the underlying blocks. The generated interface is similar to the subsystem interface described in “Generating a Black Box Interface for a Subsystem” on page 10-3.
- For all RAM blocks, data reads out from the output ports with a latency of 1 clock cycle.
- The generated code for the RAM blocks does not include a reset signal. Generation of a reset is omitted because in the presence of a reset signal, synthesis tools would not infer a RAM from the HDL code.
- Most synthesis tools will infer RAM from the generated HDL code. However, your synthesis tool may not map the generated code to RAM for the following reasons:
 - A small RAM size: your synthesis tool may implement a small RAM with registers for better performance.
 - The presence of a clock enable signal. It is possible to suppress generation of a clock enable signal Dual Port RAM and Single Port RAM blocks, as described in “Limitations for RAM Blocks” on page 7-13.

Take care to verify that your synthesis tool produces the expected result when synthesizing code generated for the Dual Port RAM block.

If data output at the write port is not required, you can achieve better RAM inferring with synthesis tools by using the Simple Dual Port RAM block rather than the Dual Port RAM block.

Generic RAM and ROM Demos

Generic RAM Template Supports RAM Without a Clock Enable Signal

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 new Getting Started with RAM and ROM in Simulink demo. To open the demo, type the following command at the MATLAB prompt:

```
hdlcoderramrom
```

Generating ROM with Lookup Table and Unit Delay Blocks

Simulink HDL Coder does not provide a ROM block, but you can easily build one using basic Simulink blocks. The new Getting Started with RAM and ROM in Simulink demo includes an example in which a ROM is built using a Lookup Table block and a Unit Delay block. To open the demo, 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.

- Some synthesis tools do not support RAM inference with a clock enable. For the Dual Port RAM and Single Port RAM blocks, you can suppress generation of the clock enable signal. These blocks support the `AddClockEnablePort` implementation parameter. The default setting for `AddClockEnablePort` is 'on'. To suppress to generation of the clock enable signal, set `AddClockEnablePort` to off for the desired RAM block(s) in a control file, as in the following example.

```
function c = controlfilename

% Control file for hdlcoderfirram

c = hdlnewcontrol(mfilename);

c.generateHDLFor('hdlcoderfirram/FIR_RAM');
c.forEach('hdlcoderfirram/FIR_RAM/Dual Port RAM',...
    'hlddemolib/Dual Port RAM', {},...
    'hdldefaults.RamBlockDualHDLInstantiation',...
    {'AddClockEnablePort','off'});
```

- If you suppress the generation of the clock enable signal in a multirate model that has RAM blocks running at a slower rate than the model's base rate (fastest rate), the behavior of the generated code will no longer match that of the Simulink model. If you want to ensure that the Simulink model and the generated code behave identically, make sure that the RAM blocks run at the base rate of the model.

HDL Counter

In this section...

“Overview” on page 7-15

“Counter Modes” on page 7-15

“Control Ports” on page 7-17

“Defining the Counter Data Type and Size” on page 7-20

“HDL Implementation and Implementation Parameters” on page 7-21

“Parameters and Dialog Box” on page 7-22

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 has no 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** drop-down 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 never equal the **Count to value**, or 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 never reaches the **Count to value**:

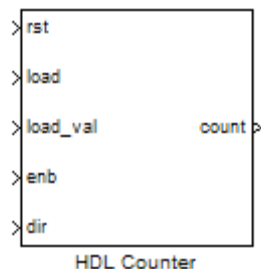
0 2 4 6 0 2 4 6 ...

Control Ports

By default, the HDL Counter has no 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 all control ports:

- All control ports are synchronous.
- All control ports except the load value input have Boolean data type.
- All control ports must have the same sample time.
- If any 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.
- All 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.

In effect, when the signal at the `dir` port is 0, the counter reverses direction. 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

Count Direction Signal Value	Step Value Sign	Actual Count Direction
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

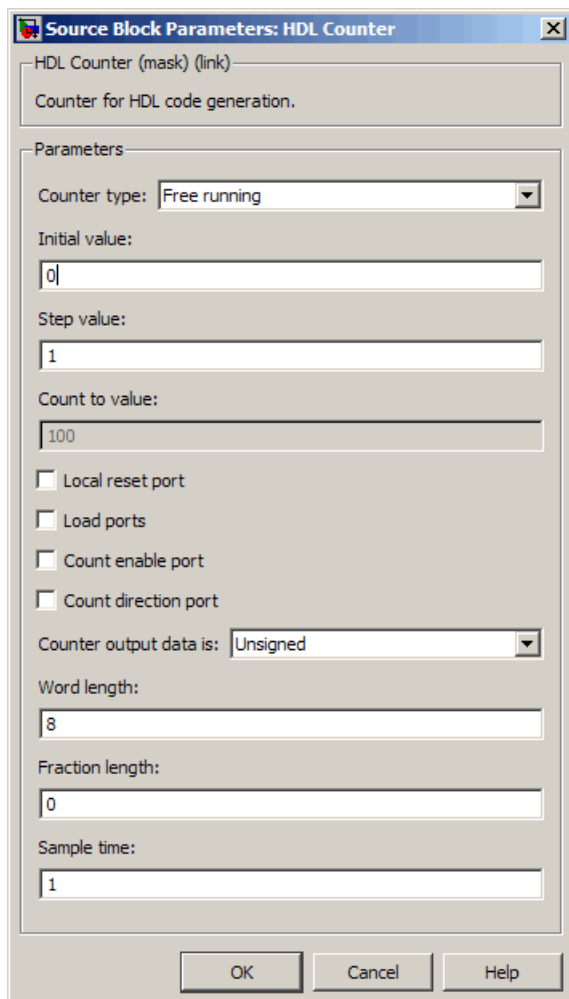
Fraction length: To define an integer counter, accept the default **Fraction length** of 0. To define a fixed-point counter, enter the number of bits to the right of the binary point.

HDL Implementation and Implementation Parameters

Implementation: `hdldefaults.HDLCounterHDL Emission`

Implementation Parameters: `InputPipeline, OutputPipeline`

Parameters and Dialog Box



Counter type

Default: Free running

This drop-down menu selects the operation mode of the counter (see “Counter Modes” on page 7-15). 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 7-16.

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 7-19).

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 never equal the **Count to value**, or may overflow and iterate through the counter range some number of times before reaching the **Count to value** (see “Count Limited Mode” on page 7-16).

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 7-18.

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 7-18.

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 7-19.

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 7-19.

Counter output data is:

Default: Unsigned

This drop-down 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 has no 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 any 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 7-17.)

HDL FFT

In this section...

“Overview” on page 7-27

“Block Inputs and Outputs” on page 7-28

“HDL Implementation and Implementation Parameters” on page 7-30

“Parameters and Dialog Box” on page 7-30

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 Signal Processing Blockset 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 Signal Processing Blockset FFT block.

The operation of the HDL FFT block differs from the Signal Processing Blockset 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 7-28).

HDL FFT Block Demo

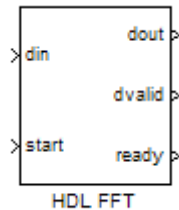
To get started with the HDL FFT block, run the “Using the Minimum Resource HDL FFT” demo. The demo is located in the Simulink/Simulink HDL Coder/Signal Processing demo library.

The demo 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 demo, 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 Signal Processing Blockset FFT block. The output frames from the Signal Processing Blockset FFT block are buffered into a FIFO and compared to the output of the HDL FFT block. Examination of the demo results shows the outputs to be identical.

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.

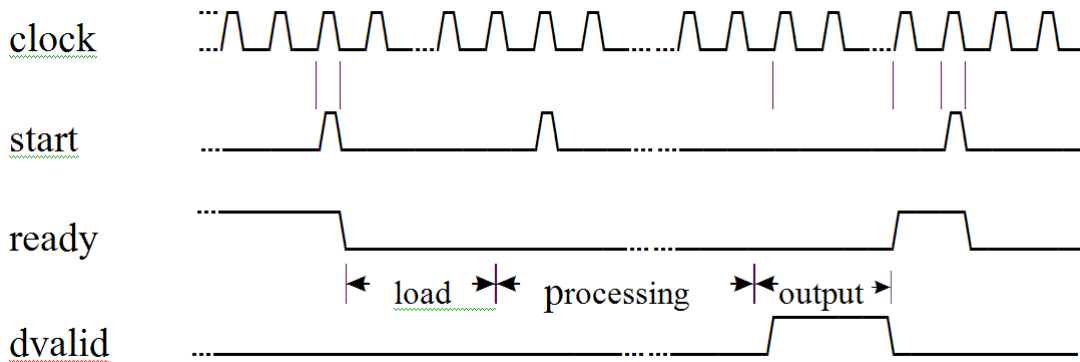
The output ports are:

- **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 correct and 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 all N -point FFT outputs are sent out.

The expression T_{cycle} denotes the total number of clock cycles required by the HDL FFT block to complete an FFT of length N . T_{cycle} is defined as follows:

- Where $N > 8$

$$T_{\text{cycle}} = 3N/2 - 2 + \log_2(N) * (N/2 + 3);$$

- Where $N = 8$

$$T_{\text{cycle}} = 3N/2 - 1 + \log_2(N) * (N/2 + 3);$$

Given `Tcycle`, you can then define the period between assertions of the HDL FFT start signal in any way that is suitable to your application. For example, in the “Using the Minimum Resource HDL FFT” demo, 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 demo model, `startLen` determines the period of a Pulse Generator that drives the HDL FFT block’s start input.

In the demo, 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: `hdldefaults.FFTDITMRHDLemission`

Implementation Parameters: `InputPipeline`, `OutputPipeline`

Parameters and Dialog Box

The following figure shows the HDL FFT block dialog box, with all parameters at their default settings.

Function Block Parameters: HDL FFT [X]

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 Signal Processing Blockset FFT block. See also the FFT block reference section in the Signal Processing Blockset documentation.

Overflow mode

Default: Saturate

The HDL FFT block supports all overflow modes of the Signal Processing Blockset FFT block. See also the FFT block reference section in the Signal Processing Blockset 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 always 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 are always saturated and rounded 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 always skips the divide-by-two operation on butterfly outputs for fixed-point signals.

Bitwise Operators

In this section...

“Overview of Bitwise Operator Blocks” on page 7-35

“Bit Concat” on page 7-37

“Bit Reduce” on page 7-39

“Bit Rotate” on page 7-41

“Bit Shift” on page 7-43

“Bit Slice” on page 7-45

Overview of Bitwise Operator Blocks

The Bitwise Operator sublibrary provides commonly used operations on bits and bit fields.

All 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

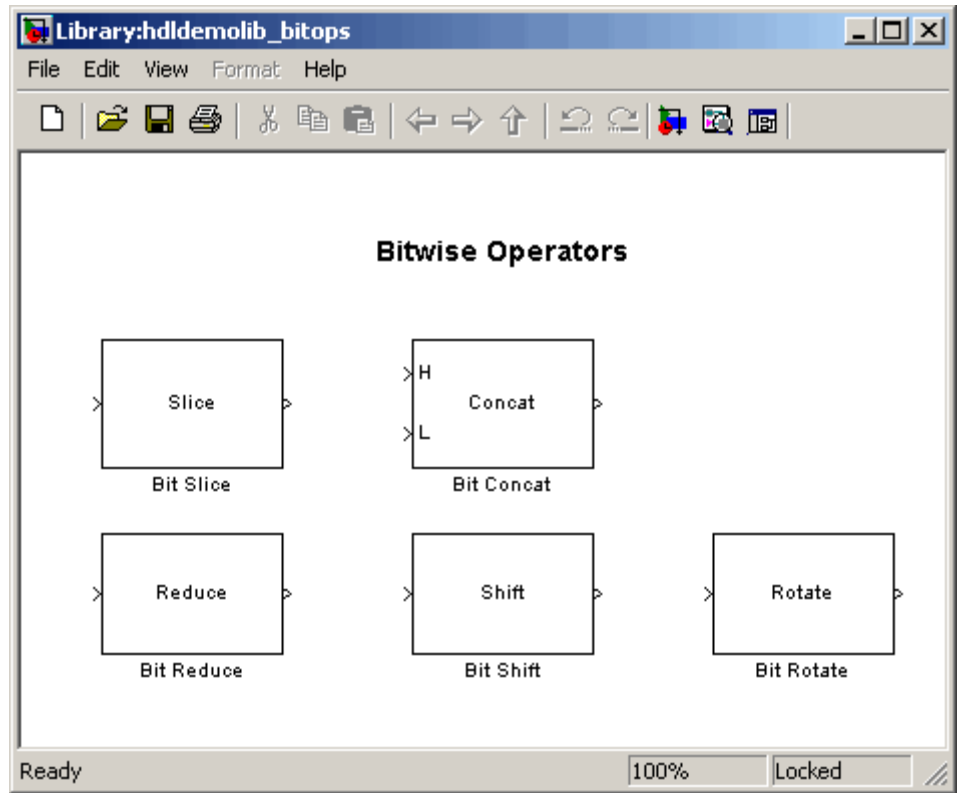


Bitwise Operators

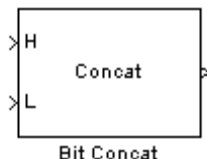
in the `hdl demolib` window. Alternatively, you can open the Bitwise Operators sublibrary directly by typing the following command at the MATLAB prompt:

hddemolib_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 all 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

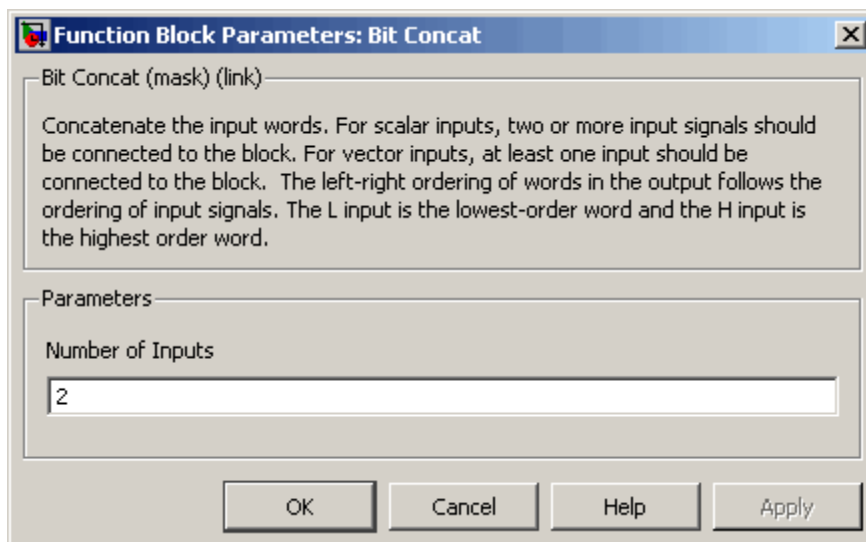
- Input: Fixed-point, integer (signed or unsigned), Boolean
- Output: Unsigned fixed-point or integer (Maximum concatenated output word size: 128 bits)

HDL Implementation and Implementation Parameters

Implementation: `hdldefaults.BitConcat`

Implementation Parameters: InputPipeline, OutputPipeline

Parameters and Dialog Box

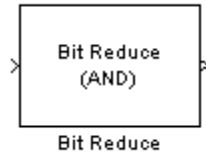


Number of Inputs: Enter an integer specifying the number of input signals. The number of input ports displayed on the block updates when **Number of Inputs** changes.

- 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

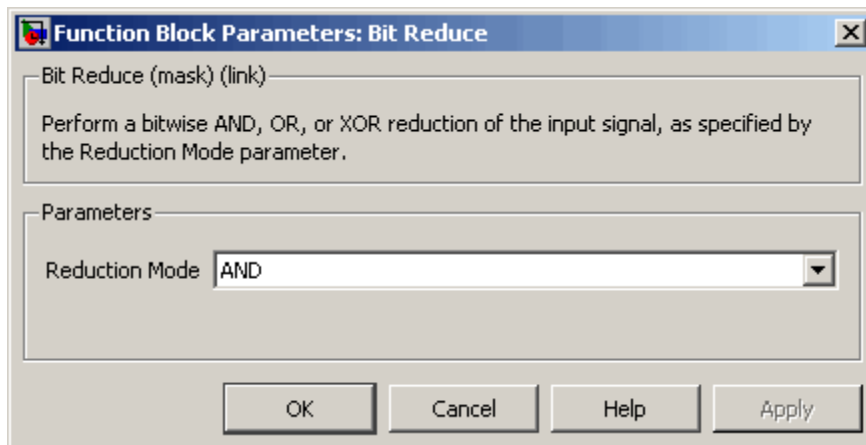
- Input: Fixed-point, integer (signed or unsigned), Boolean
- Output: always `ufix1`

HDL Implementation and Implementation Parameters

Implementation: `hdldefaults.BitReduce`

Implementation Parameters: `InputPipeline`, `OutputPipeline`

Parameters and Dialog Box



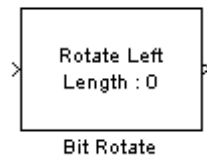
Reduction Mode

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

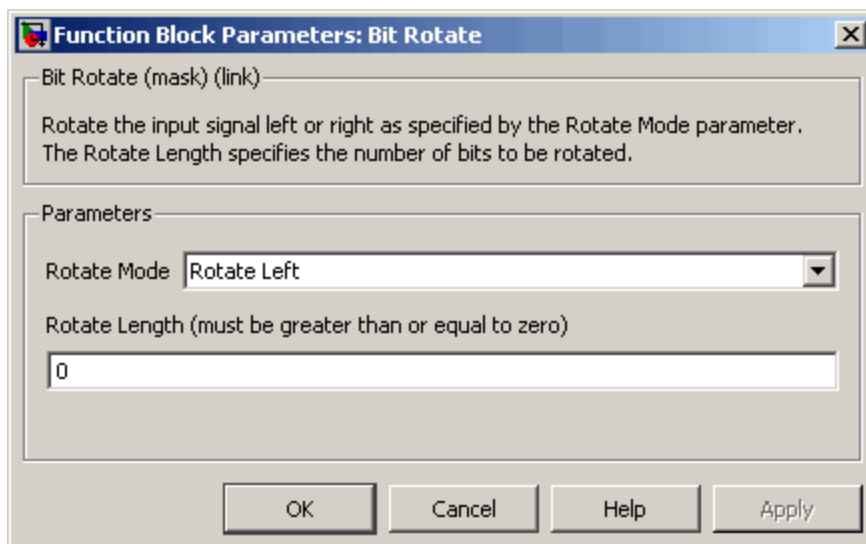
- Input: Fixed-point, integer (signed or unsigned), Boolean
 - Minimum word size: 2 bits
 - Maximum word size: 128 bits
- Output: Has the same data type as the input signal

HDL Implementation and Implementation Parameters

Implementation: `hdldefaults.BitRotate`

Implementation Parameters: `InputPipeline`, `OutputPipeline`

Parameters and Dialog Box



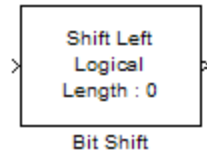
Rotate Mode: Specifies direction of rotation, either left or right.

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

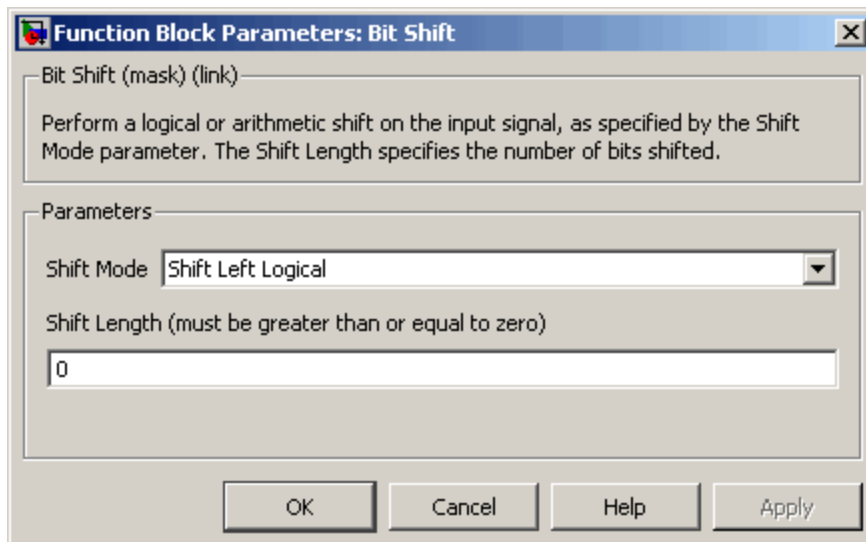
- Input: Fixed-point, integer (signed or unsigned), Boolean
 - Minimum word size: 2 bits
 - Maximum word size: 128 bits
- Output: Has the same data type as the input signal

HDL Implementation and Implementation Parameters

Implementation: `hdldefaults.BitShift`

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

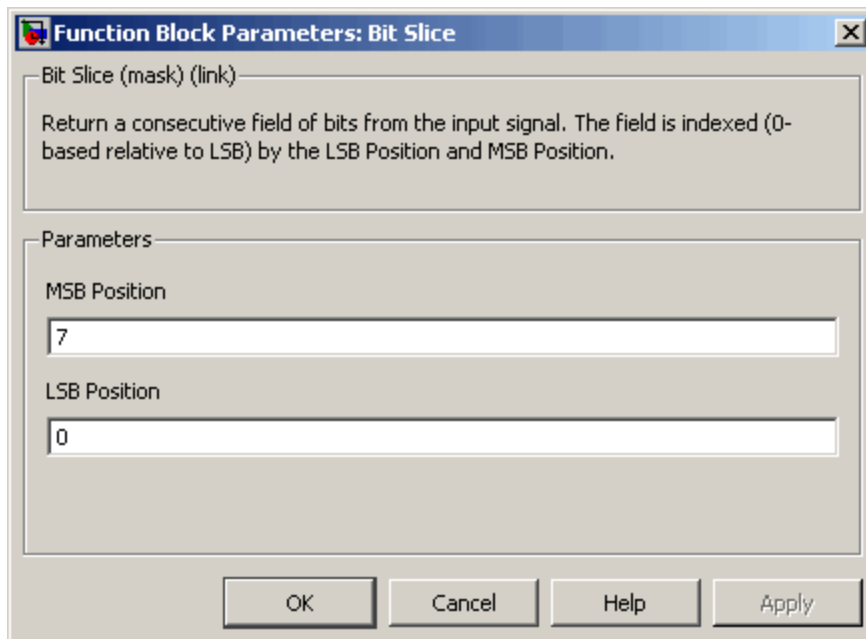
- Input: Fixed-point, integer (signed or unsigned), Boolean
- Output: unsigned fixed-point or unsigned integer

HDL Implementation and Implementation Parameters

Implementation: `hdldefaults.BitSlice`

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 $(\text{MSB Position} - \text{LSB Position}) + 1$.

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

- “Overview of Generated Models” on page 8-2
- “Example: Numeric Differences” on page 8-4
- “Example: Latency” on page 8-8
- “Defaults and Options for Generated Models” on page 8-12
- “Fixed-Point and Double-Precision Limitations for Generated Models” on page 8-17

Overview of Generated Models

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 “Example: Numeric Differences” on page 8-4). 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 accurately 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) all differences between the original and generated models.

The coder always creates a generated model as part of the code generation process, and always 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 that let you

- Suppress display of the generated model.
- Create and display the only generated model, with code generation suppressed.

- Specify the color highlighting of differences between the original and generated models.
- Specify a name or prefix for the generated model.

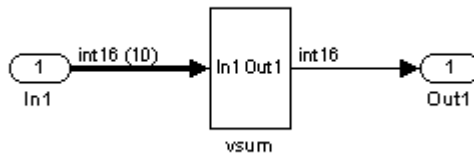
These options are described in “Defaults and Options for Generated Models” on page 8-12.

Example: Numeric Differences

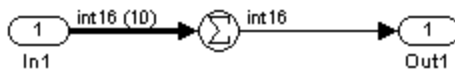
This example first examines a simple model that uses a code generation control file to select a speed-optimized Sum block implementation. It then examines a generated model and locates the numeric changes introduced by the optimization.

If you are not familiar with code generation control files and selection of block implementations, see Chapter 5, “Code Generation Control Files”.

The model, `simplevectorsum`, consists of a subsystem, `vsum`, driven by a vector input of width 10, with a scalar output. The following figure shows the root level of the model.



The device under test is the `vsum` subsystem, shown in the following figure. The subsystem contains a Sum block, configured for vector summation.



The model is configured to use a code generation control file, `svsumctrl.m`. The control file (shown in the following listing) maps the `SumTreeHDL Emission` implementation to the Sum block within the `vsum` subsystem. This implementation, optimized for minimal latency, generates a tree-shaped structure of adders for the Sum block.

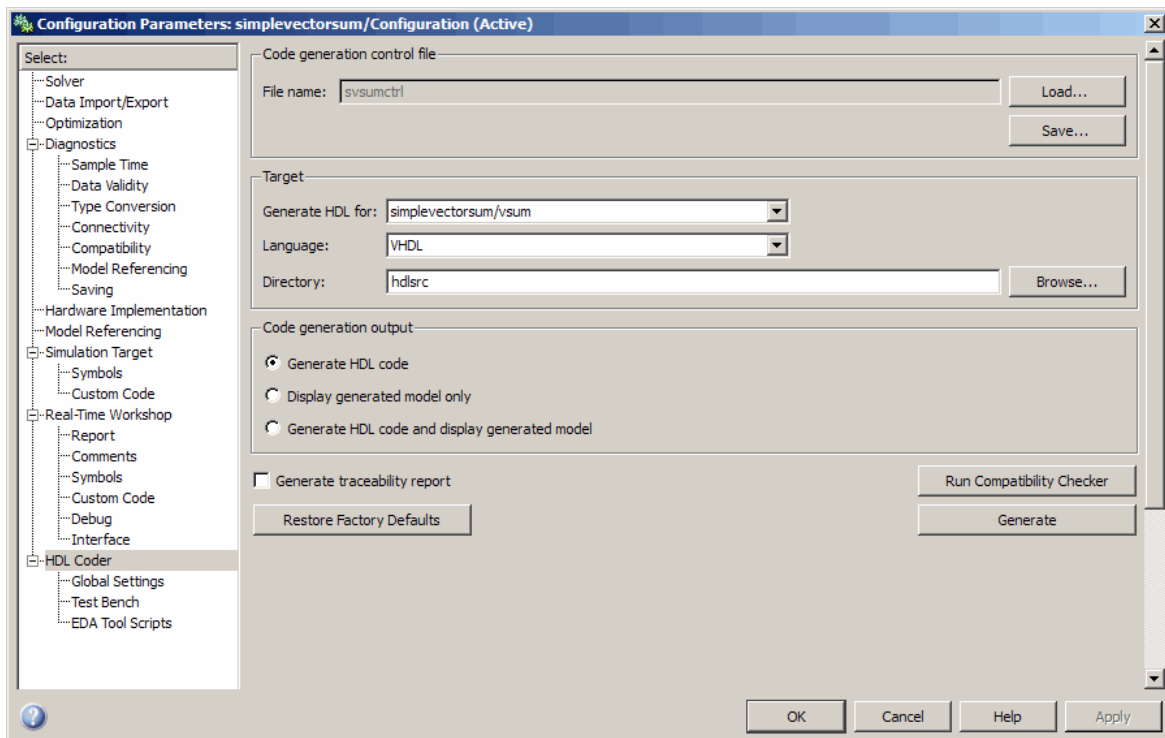
```
function config = svsumctrl
% Code generation control file for simplevectorsum model.
```

```

config = hdlnewcontrol(mfilename);
% Specify tree-structured adders implementaton for Sum block.
config.forEach('simplevectorsum/vsum/Sum',...
    'built-in/Sum',{},...
    'hdldefaults.SumTreeHDLemission',{});

```

The **File name** field of the Configuration Parameters dialog box (shown in the following figure) specifies that this control file is to be used during code generation.



When code generation is initiated, the coder displays messages similar to those shown in the following example. The messages indicate that the control file is applied; control file processing is followed by creation of the generated model and generation of HDL code.

```

### Applying HDL Code Generation Control Statements

```

```

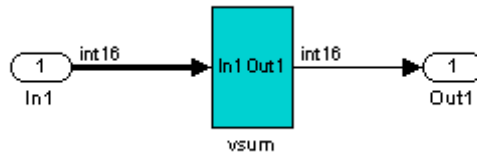
### 1 Control Statements to be applied

### Begin Model Generation
### Generating new model: gm_simplevectorsum.mdl
### Model Generation Complete.

### Begin VHDL Code Generation
### Generating package file hdlsrc\vsum_pkg.vhd
### Working on simplevectorsum/vsum as hdlsrc\vsum.vhd
### HDL Code Generation Complete.

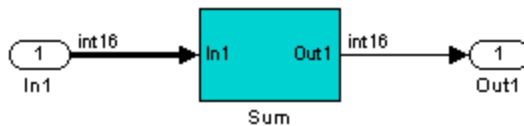
```

The generated model, gm_ simplevectorsum, is displayed after code generation. This model is shown in the following figure.

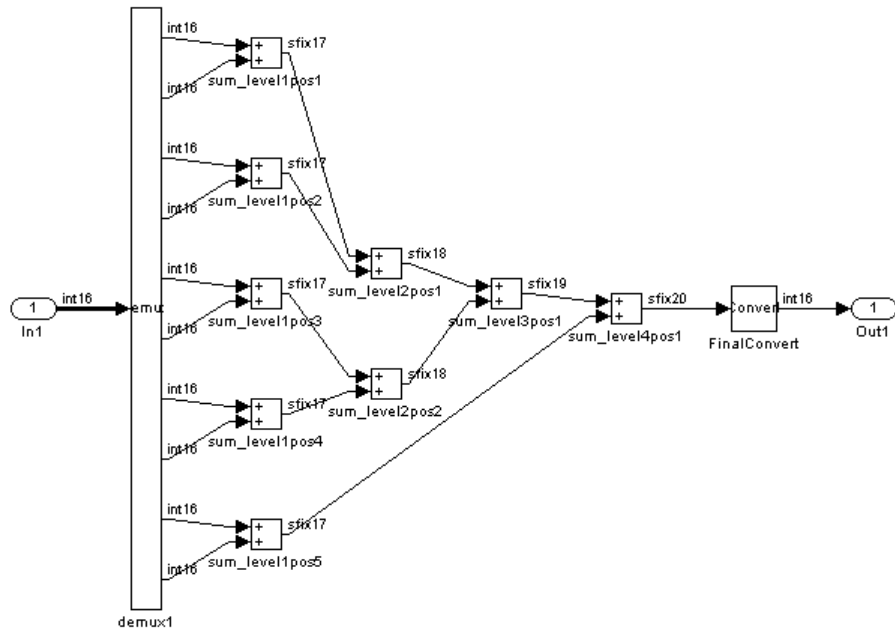


At the root level, this model appears identical to the original model, except that the vsum subsystem has been highlighted in cyan. This highlighting indicates that the subsystem differs in some respect from the vsum subsystem of the original model.

The following figure shows the vsum subsystem in the generated model. Observe that the Sum block is now implemented as a subsystem, which is also highlighted.



The following figure shows the internal structure of the Sum subsystem.



The vector sum is implemented 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. A Data Conversion block, inserted before the final output, converts the 20-bit fixed-point result to the int16 data type required by the model.

Example: Latency

This example uses the `simplevectorsum_cascade` model. This model is identical to the model in the previous example (“Example: Numeric Differences” on page 8-4), except that it uses a control file that selects a cascaded implementation for the Sum block. This implementation introduces both latency and numeric differences.

The model is configured to use the control file `svsum_cascade_ctrl.m`. The control file (shown in the following listing) maps the `SumCascadeHDL Emission` implementation to the Sum block within the `vsum` subsystem. This implementation generates a cascade of adders for the Sum block.

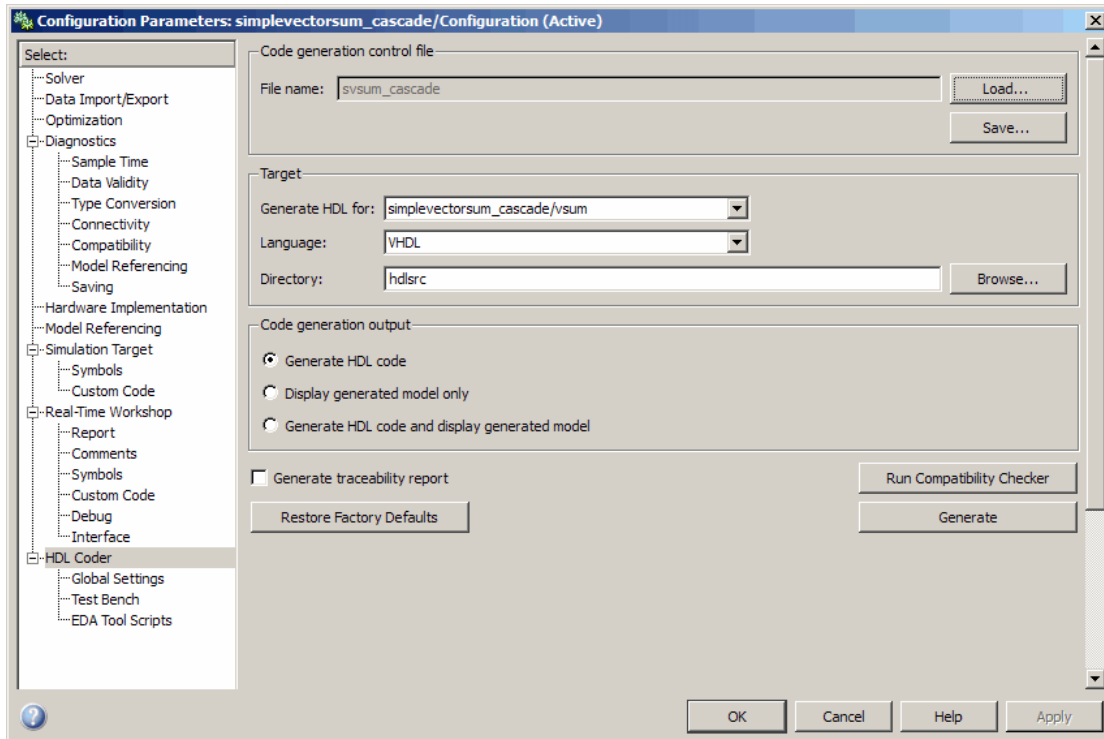
```
function config = svsum_cascade_ctrl
% Code generation control file for simplevectorsum model.

config = hdlnewconfig(mfilename);

% specify cascaded adders implementation for Sum block

config.forEach('simplevectorsum_cascade/vsum/Sum',...
    'built-in/Sum',{},...
    'hdldefaults.SumCascadeHDL Emission',{});
```

The **File name** field of the Configuration Parameters dialog box (shown in the following figure) specifies that this control file is used during code generation.



When code generation is initiated, the coder displays messages similar to those shown in the following example. The messages indicate that the control file is applied; control file processing is followed by creation of the generated model and generation of HDL code.

```

### Applying HDL Code Generation Control Statements
###   1 Control Statements to be applied

### Begin Model Generation
### Generating new model: gm_simplevectorsum_cascade.mdl
### Model Generation Complete.

### Begin VHDL Code Generation
### Generating package file hdlsrc\simplevectorsum_cascade_pkg.vhd
### Working on simplevectorsum_cascade/vsum as hdlsrc\vsum.vhd
### Working on Timing Controller as hdlsrc\Timing_Controller.vhd

```

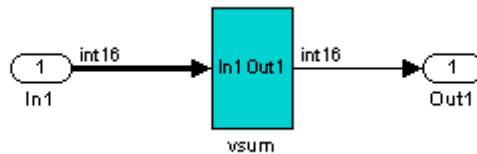
```
### Working on simplevectorsum_cascade as hdlsrc\simplevectorsum_cascade.vhd
### HDL Code Generation Complete.
```

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 all 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 all computations have been performed 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 generates these rates from a single master clock using counters and multiple clock enables. The `Timing_Controller` entity definition is written to a separate code file.

The generated model, `gm_simplevectorsum_cascade`, is displayed after code generation. This model is shown in the following figure.

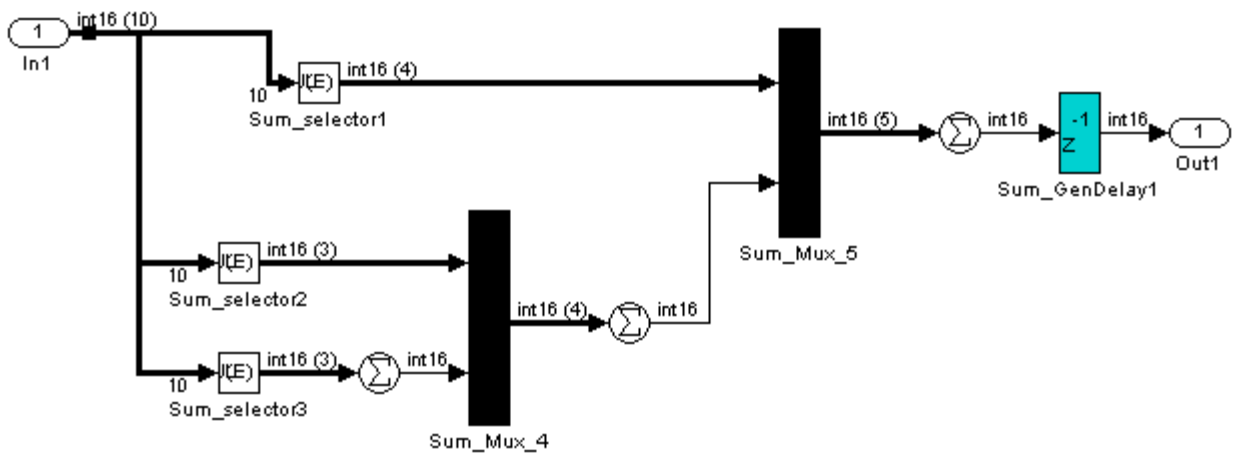


As in the previous (`gm_simplevectorsum`) example, the `vsum` subsystem is highlighted in cyan. This highlighting indicates that the subsystem differs in some respect from the `vsum` subsystem of the original model.

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 8-12
“GUI Options” on page 8-13
“Generated Model Properties for makehdl” on page 8-14

Defaults for Model Generation

This section summarizes the defaults used by the coder when generated models are built.

Model Generation

The coder always creates a generated model as part of the code generation process. The generated model is built 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 model(s) exist in memory, an integer is suffixed to the name of each successively generated model. The suffix ensures that each generated model has a unique name. For example, if the original model is

named `test`, generated models will be named `gm_test`, `gm_test0`, `gm_test1`, etc.

Note Take care, when regenerating code from your models, to select the original model for code generation, not a previously generated model. Generating code from a generated model may introduce unintended delays or numeric differences that could make the model operate incorrectly.

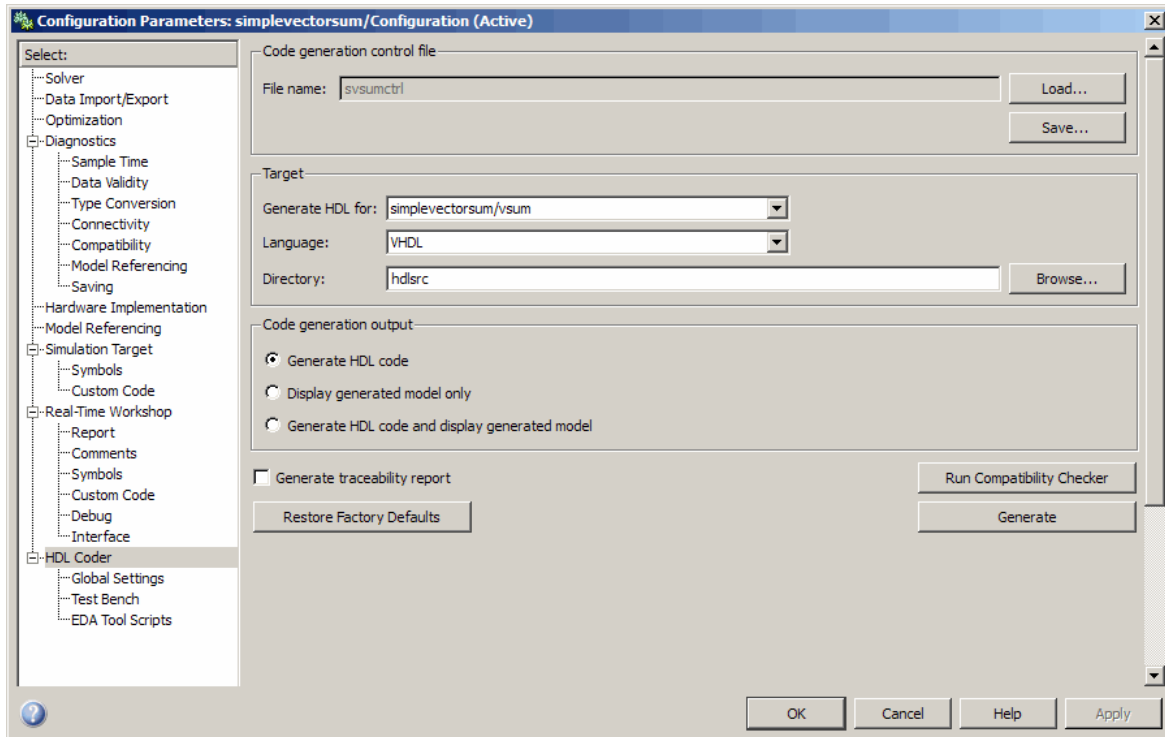
Block Highlighting

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 any differences have been introduced, by examining the root level of the generated model.

If there are no differences between the original and generated models, no blocks will be highlighted.

GUI Options

The Simulink 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 8-14). Generated model options are located in the top-level **HDL Options** pane of the Configuration Parameters dialog box, as shown in the following figure.



The options are

- **Generate HDL code:** (Default) Generate code, but do not display the generated model.
- **Display generated model only:** Create and display the generated model, but do not proceed to code generation.
- **Generate HDL code and display generated model:** Generate both code and model, and display the model when completed.

Generated Model Properties for makehdl

The following table summarizes 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 <code>gm_modelname</code> , where <code>gm_</code> is the default prefix and <code>modelName</code> 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 <code>modelName</code> 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"> • <code>GenerateHDLCode</code>: (Default) Generate code, but do not display the generated model. • <code>GenerateHDLCodeAndDisplayGeneratedModel</code>: Create and display generated model, but do not proceed to code generation. • <code>DisplayGeneratedModelOnly</code>: Generate both code and model, and display model when completed.
'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 (<code>RGBName</code>) as one of the following color string values: <ul style="list-style-type: none"> • cyan (default) • yellow • magenta • red

Property and Value(s)	Description
	<ul style="list-style-type: none">• green• blue• white• black

Fixed-Point and Double-Precision Limitations for Generated Models

In this section...

“Fixed-Point Limitation” on page 8-17

“Double-Precision Limitation” on page 8-17

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} .

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

- “Creating and Using a Code Generation Report” on page 9-2
- “HDL Compatibility Checker” on page 9-18
- “Supported Blocks Library” on page 9-22
- “Annotating Generated Code with Comments and Requirements” on page 9-24
- “Code Tracing Using the Mapping File” on page 9-25

Creating and Using a Code Generation Report

In this section...

“Traceability and the Code Generation Report” on page 9-2

“Generating an HTML Code Generation Report from the GUI” on page 9-4

“Generating an HTML Code Generation Report from the Command Line” on page 9-7

“Keeping the Report Current” on page 9-9

“Tracing from Code to Model” on page 9-9

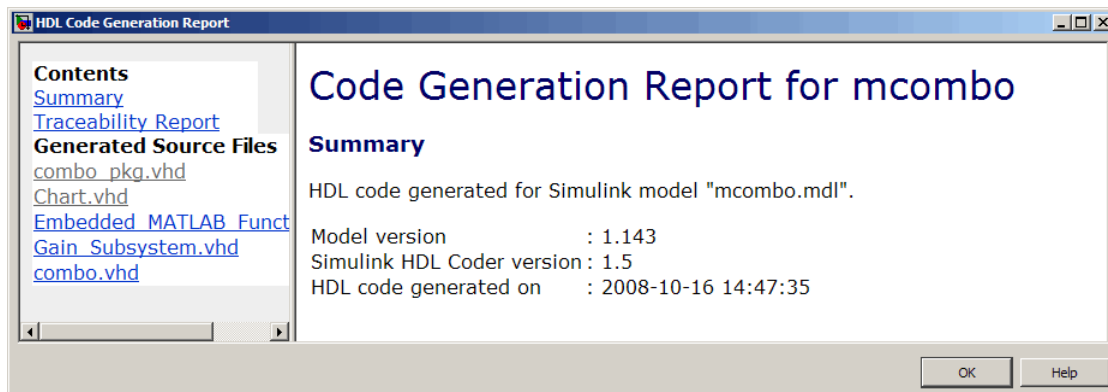
“Tracing from Model to Code” on page 9-11

“Mapping Model Elements to Code Using the Traceability Report” on page 9-15

“HTML Code Generation Report Limitations” on page 9-17

Traceability and the Code Generation Report

Even a relatively small model can generate hundreds of lines of HDL code. To help you navigate more easily between the generated code and your source model, the coder provides a *traceability* option that lets you generate reports from either the GUI or the command line. When you enable traceability, the coder creates and displays an HTML code generation report during the code generation process. The following figure shows a typical report.



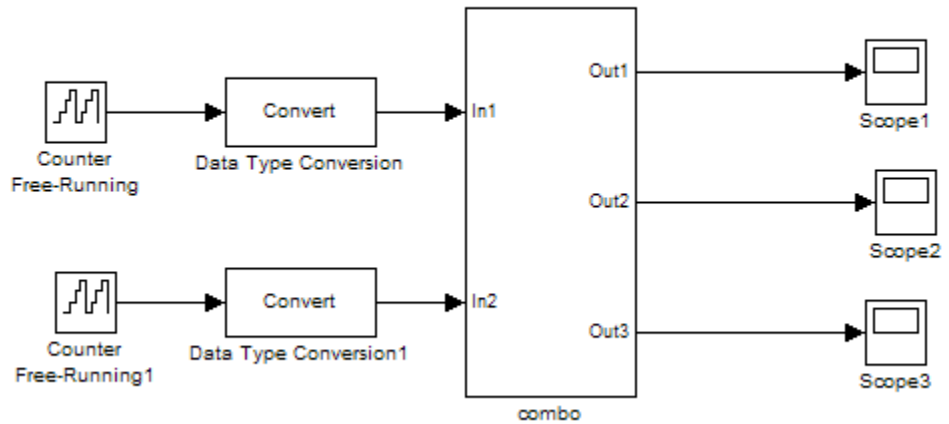
The report has several sections:

- The **Summary** section lists version and date information.
- The **Traceability Report** lets you account for **Eliminated / Virtual Blocks** that are untraceable, versus the listed **Traceable Simulink Blocks / Stateflow Objects / Embedded MATLAB Scripts**, providing a complete mapping between model elements and code.
- 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 Coder > Navigate to Code** from the context menu.

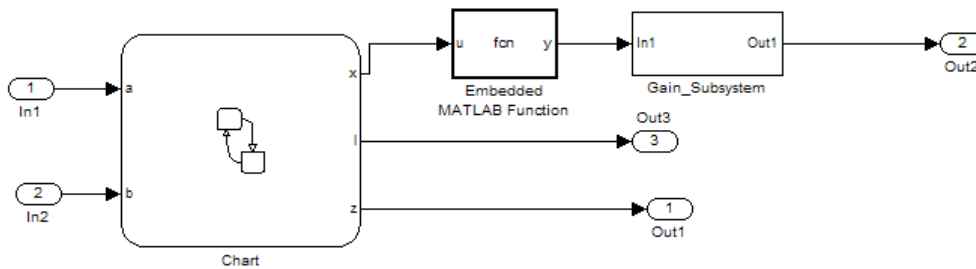
In the following sections, the `mcombo` demonstration model is used to illustrate stages in the workflow for generating code generation reports from the GUI and from the command line. The model is available in the `demos` directory as the following file:

```
MATLABROOT\toolbox\hdlcoder\hdlcoderdemos\mcombo.mdl
```

This figure shows the root-level `mcombo` model.



Simulink HDL Coder supports report generation for subsystems, blocks, Stateflow charts, and Embedded MATLAB blocks. The `combo` subsystem, shown in the following figure, includes each of these components.

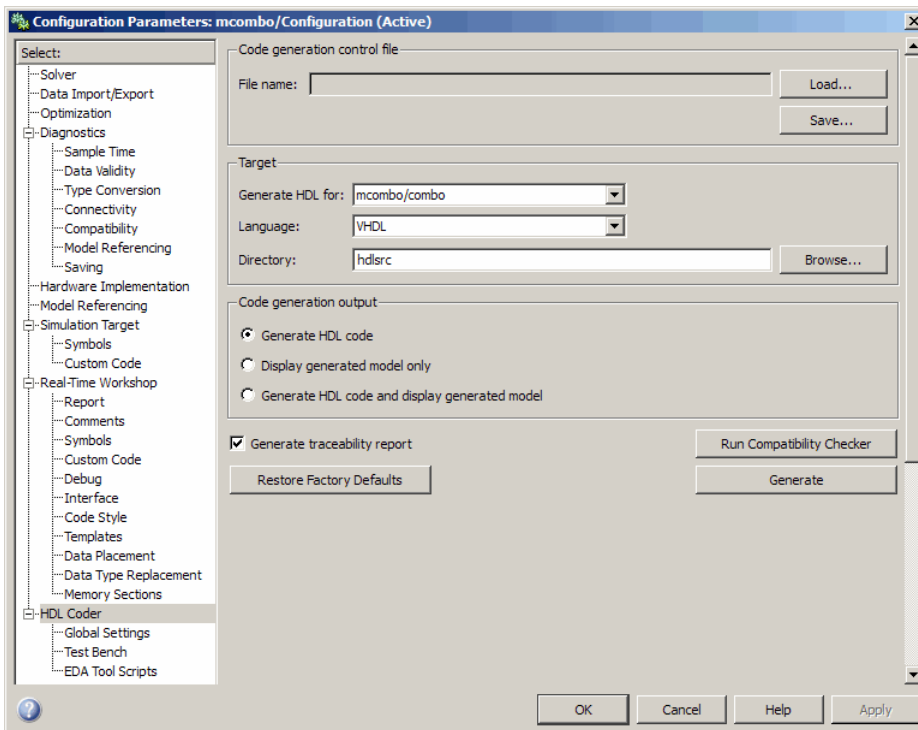


Generating an HTML Code Generation Report from the GUI

To generate a Simulink HDL Coder code generation report from the GUI:

- 1 With your model open, open the Configuration Parameters dialog box or Model Explorer and navigate to the **HDL Coder** pane.

- 2 To enable report generation, select **Generate traceability report**.
- 3 Make sure that the correct DUT is selected for code generation. When generating reports, the coder requires that a subsystem be selected for code generation. An error results if the root-level model is selected. In the following figure, the subsystem `mcombo/combo` is selected in the **Generate HDL for list**.
- 4 Click **Apply**. The dialog box should now appear as shown in the following figure.



- 5 Click the **Generate** button to initiate code and report generation.

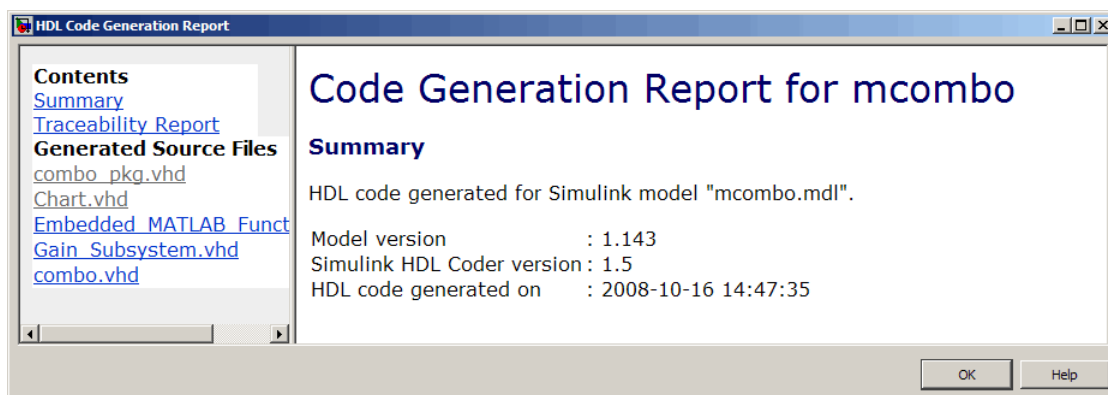
When you select **Generate traceability report**, the coder generates HTML report files as part of the code generation process. Report file generation is the final phase of that process. As code generation proceeds,

the coder displays progress messages. The process should complete successfully with messages similar to the following:

```
### Generating HTML files for traceability in slprj\hdl\mcombo\html directory ...

### HDL Code Generation Complete.
```

When code generation completes, the coder displays the HTML code generation report in a new window. The following figure shows the report generated for the `combo` subsystem of the `mcombo` model.



- 6 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 in the `slprj\hdl\subsystem_name\html` directory of the build directory. The top-level HTML report file is named `subsystem_codegen_rpt.html`. However, since 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 further information on using the report you have generated for tracing, see:

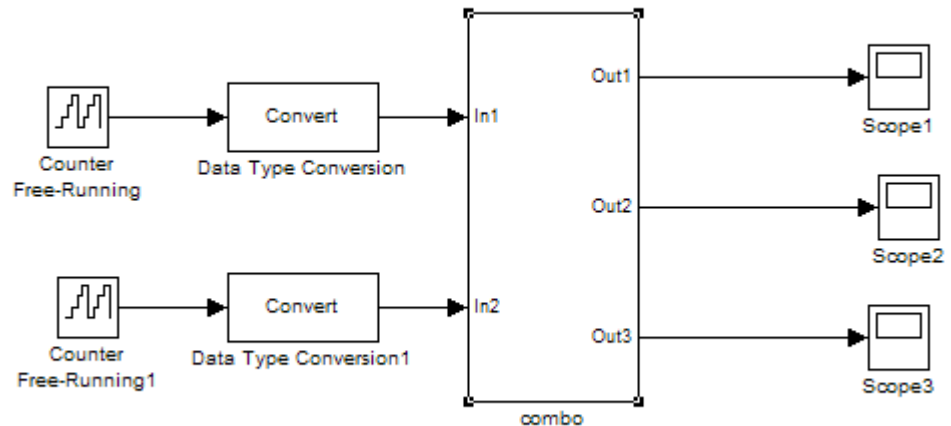
- “Tracing from Code to Model” on page 9-9

- “Tracing from Model to Code” on page 9-11
- “Mapping Model Elements to Code Using the Traceability Report” on page 9-15

Generating an HTML Code Generation Report from the Command Line

To generate a Simulink HDL Coder code generation report from the command line, enable the `makehdl` property `Traceability` as follows:

- 1 Open your model and select the desired subsystem as the device under test (DUT) for code generation. When generating reports, the coder requires that a subsystem be selected for code generation. An error results if the root-level model is selected. In the following figure, the subsystem `mcombo/combo` is selected.



- 2 At the MATLAB prompt, type the command:

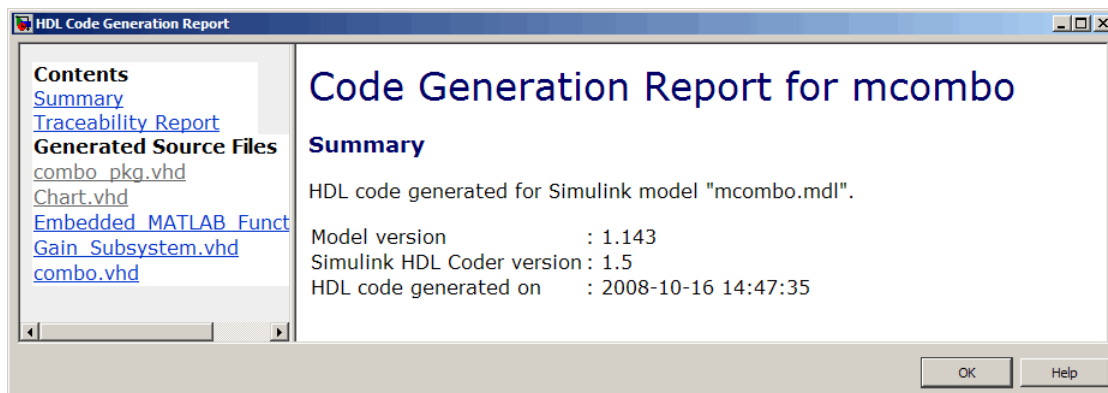
```
makehdl(gcb, 'Traceability', 'on');
```

When Traceability is enabled, 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 should complete successfully with messages similar to the following:

```
### Generating HTML files for traceability in slprj\hdl\mcombo\html directory ...
```

```
### HDL Code Generation Complete.
```

When code generation completes, the coder displays the HTML code generation report in a new window. The following figure shows the report generated for the `combo` subsystem of the `mcombo` model.



- 3 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 in the `slprj\hdl\subsystem_name\html` directory of the build directory. The top-level HTML report file is named `subsystem_codegen_rpt.html`. However, since 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 further information on using the report you have generated for tracing, see:

- “Tracing from Code to Model” on page 9-9
- “Tracing from Model to Code” on page 9-11
- “Mapping Model Elements to Code Using the Traceability Report” on page 9-15

Keeping the Report Current

If you generate a code generation report for a model, and subsequently make changes to the model, the report may be invalidated.

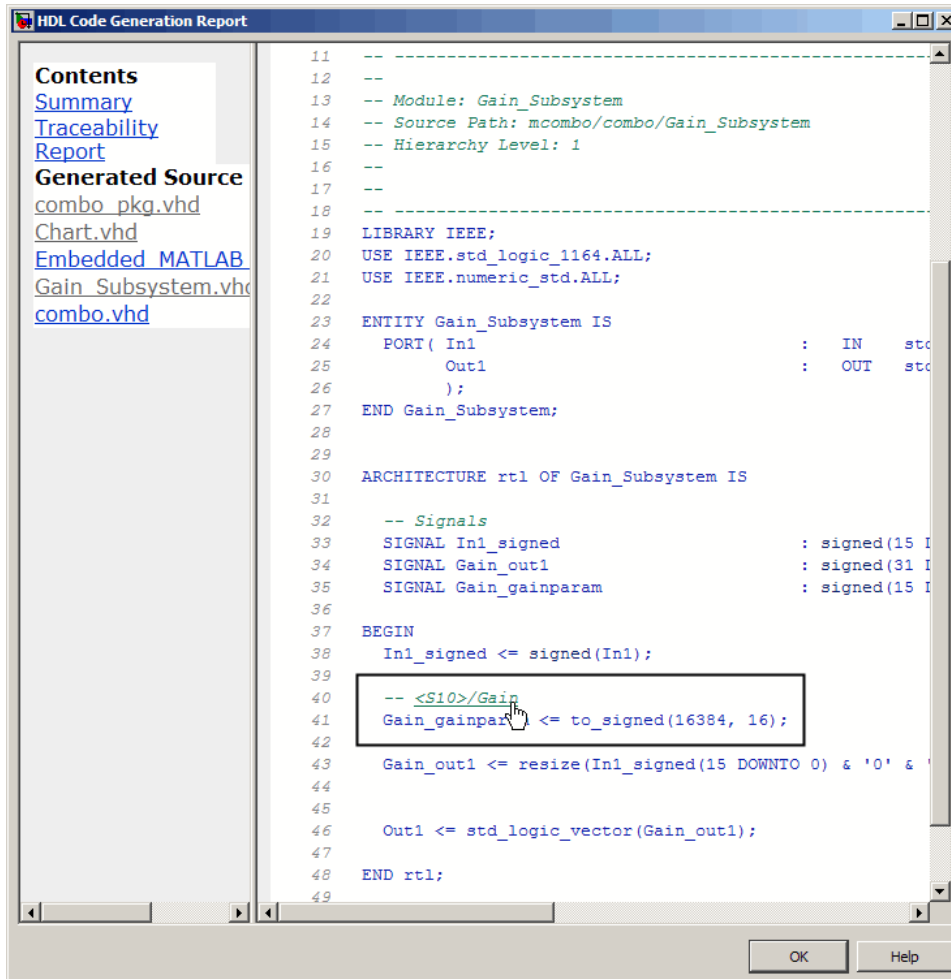
To keep your code generation report current, you should regenerate HDL code and the report after modifying the source model.

If you close and then reopen a model without making any changes, the report remains valid.

Tracing from Code to Model

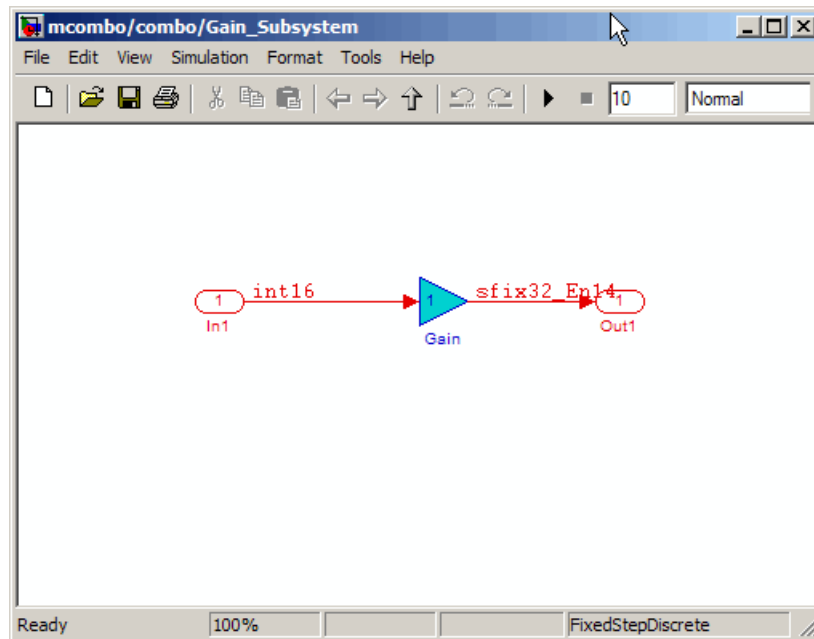
To trace from generated code to your model:

- 1** Generate code and open an HTML report for the desired DUT (see “Generating an HTML Code Generation Report from the GUI” on page 9-4 or “Generating an HTML Code Generation Report from the Command Line” on page 9-7).
- 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 any of the hyperlinks present to highlight a 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 all code references to that component in the HTML code generation report window. You select any of the following for tracing:

- Subsystem
- Simulink block
- Embedded MATLAB block
- Stateflow chart, or any of the following elements of a Stateflow chart:
 - State
 - Transition
 - Truth Table
 - Embedded MATLAB block within a chart

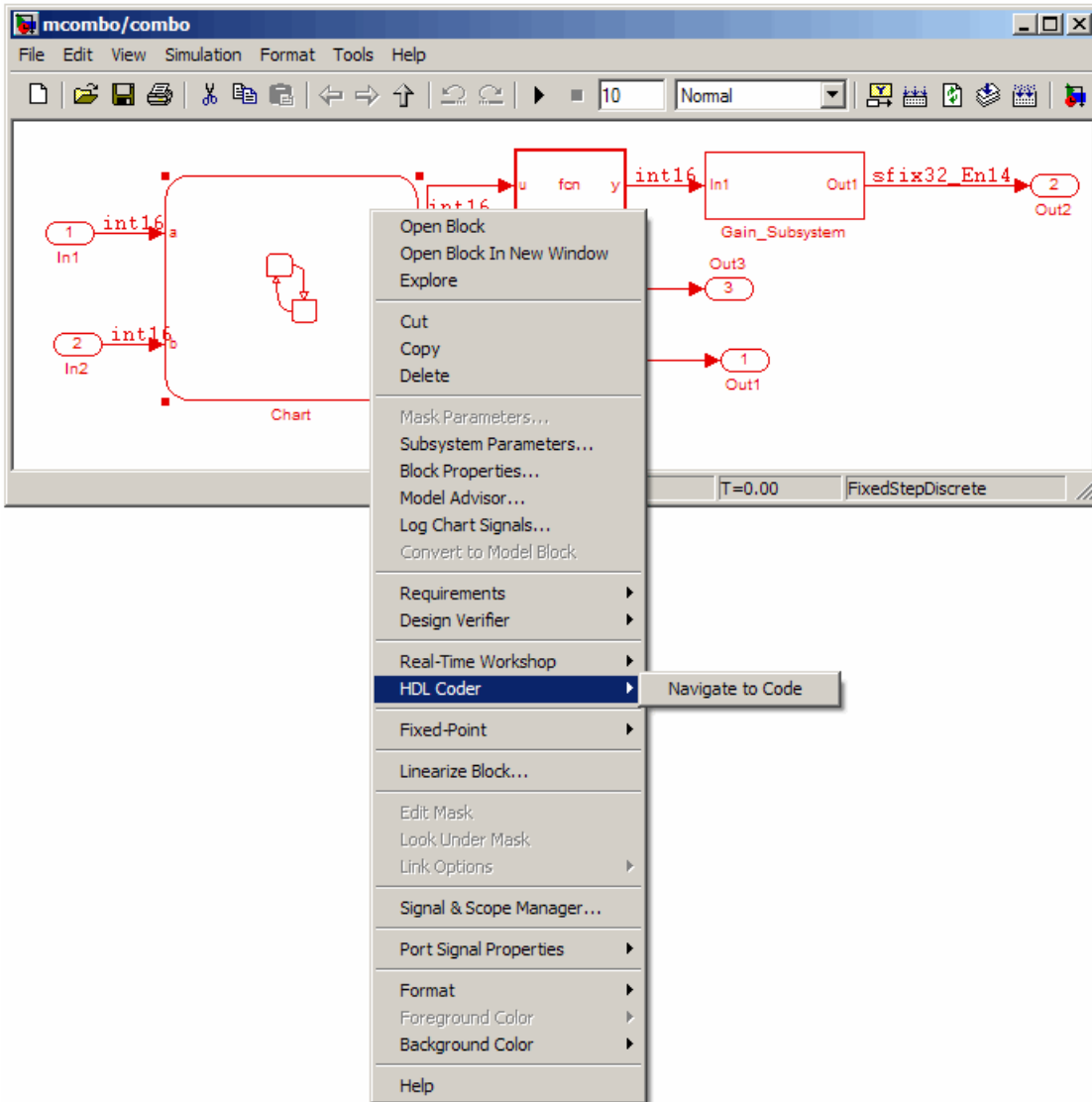
To trace a model component:

- 1 Generate code and open an HTML report for the desired DUT (see “Generating an HTML Code Generation Report from the GUI” on page 9-4 or “Generating an HTML Code Generation Report from the Command Line” on page 9-7).

Tip If code has not been generated for the model, the **HDL Coder > Navigate to Code** menu item is disabled.

- 2 In the model window, right-click the component.
- 3 In the context menu, select **HDL Coder > Navigate to Code**.

In the following figure, the context menu is displayed over the Stateflow chart within the combo subsystem.

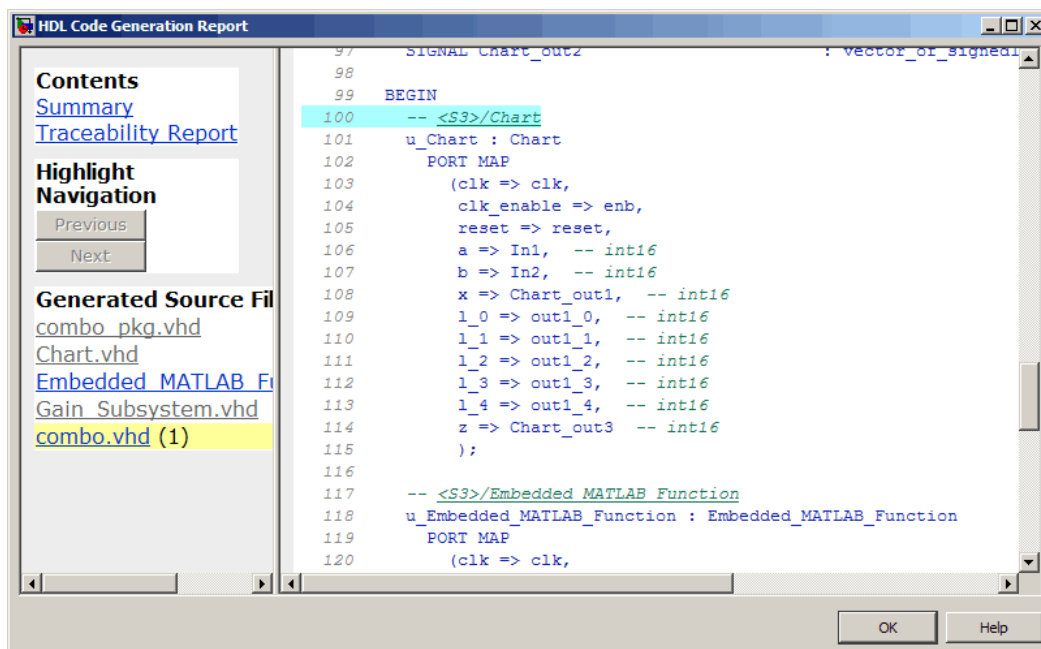


- 4** Selecting **Navigate to Code** activates the HTML code generation report window. The following figure shows the result of tracing the Stateflow chart within the combo subsystem.

In the right pane of the report window, the highlighted tag (<S3>/Chart) at line 100 indicates the beginning of the code generated code for the chart.

In the left pane of the report window, the total number of highlighted lines of code (in this case 1) is displayed next to the source file name (combo.vhd).

The left pane of the report window also contains **Previous** and **Next** buttons. These buttons let you navigate through multiple instances of code generated for a selected component. In this example, there is only one such instance, so the buttons are disabled.



Note The HDL Coder context menu option **Navigate to Code** is disabled when code has not been generated.

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 Embedded MATLAB functions**

The following figure shows the beginning of the traceability report generated for the combo subsystem of the mcombo model.

HDL Code Generation Report

Contents
[Summary](#)
[Traceability Report](#)

Highlight Navigation
[Previous](#)
[Next](#)

Generated Source File
[combo_pkg.vhd](#)
[Chart.vhd](#)
[Embedded MATLAB Function.vhd](#)
[Gain_Subsystem.vhd](#)
[combo.vhd \(1\)](#)

Traceability Report for mcombo

Table of Contents

- [Eliminated / Virtual Blocks](#)
- [Traceable Simulink Blocks / Stateflow Objects / Embedded MATLAB Scripts](#)
 - [mcombo/combo](#)
 - [mcombo/combo/Chart](#)
 - [mcombo/combo/Chart:110](#)
 - [mcombo/combo/Embedded MATLAB Function](#)
 - [mcombo/combo/Gain_Subsystem](#)

Eliminated / Virtual Blocks

Block Name	Comment
<S3>/In1	Inport
<S3>/In2	Inport
<S3>/Out1	Output
<S3>/Out2	Output
<S3>/Out3	Output
<S8>:397	Not traceable
<S8>:399	Not traceable
<S8>:403	Not traceable
<S8>:405	Not traceable
<S8>:409	Not traceable
<S8>:411	Not traceable
<S10>/In1	Inport
<S10>/Out1	Output

Traceable Simulink Blocks / Stateflow Objects / Embedded MATLAB Scripts

Subsystem: [mcombo/combo](#)

Object Name	Code Location
<S3>/Chart	combo.vhd:100
<S3>/Embedded MATLAB Function	combo.vhd:117
<S3>/Gain_Subsystem	combo.vhd:127

Stateflow chart: [mcombo/combo/Chart](#)

OK Help

HTML Code Generation Report Limitations

The following limitations apply to Simulink HDL Coder HTML code generation reports:

- If a block name in your model contains a single quote ('), code-to-model and model-to-code 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 may be able to trace individual blocks within the subsystem.

HDL Compatibility Checker

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 an M-file script, or from the GUI.

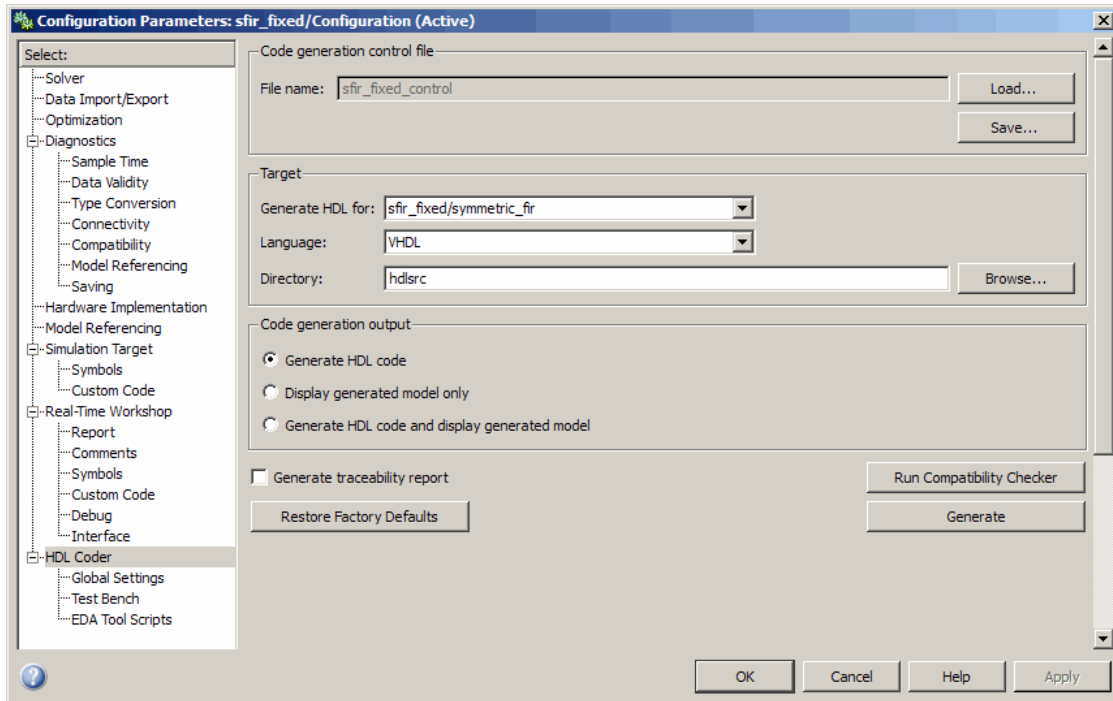
To run the compatibility checker from the command line or an M-file script, use the `checkhdl` function. The syntax of the function is

```
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 Coder** options category. The following figure shows the **HDL Coder** pane of the Configuration Parameters dialog box.



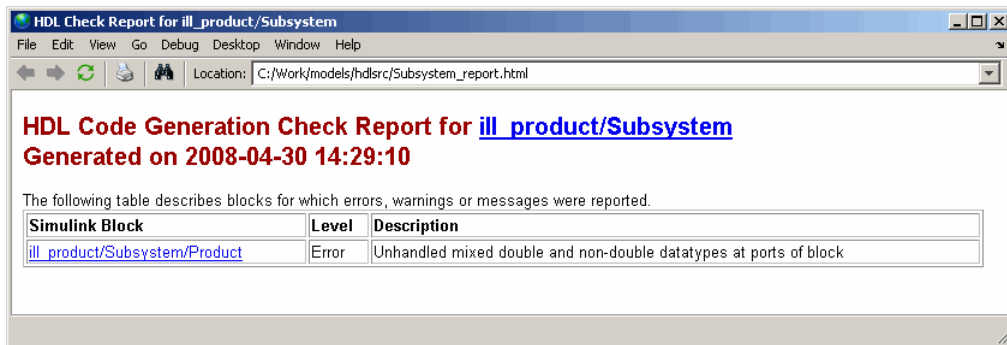
2 Select the subsystem you want to check from the **Generate HDL for** pop-up menu.

3 Click the **Run Compatibility Checker** button.

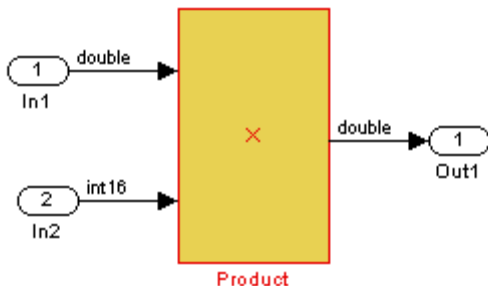
The HDL compatibility checker examines the specified system for any compatibility problems, such as use of unsupported blocks, illegal data type usage, etc. The HDL compatibility checker generates an HDL Code Generation Check Report, which is stored in the target directory. The report file naming convention is `system_report.html`, where `system` is the name of the subsystem or model that was passed in to the HDL compatibility checker.

The HDL Code Generation Check Report is displayed in a MATLAB Web Browser window. Each entry in the HDL Code Generation Check Report is hyperlinked to the block or subsystem that caused the problem. When you click the hyperlink, the block of interest highlights and displays (provided that the model referenced by the report is open).

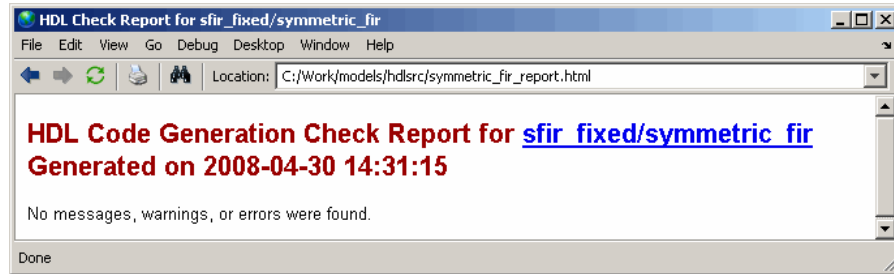
The following figure shows an HDL Code Generation Check Report that was generated for a subsystem with a Product block that was configured with a mixture of double and integer port data types. This configuration is legal in a model, but incompatible with HDL code generation.



When you click the hyperlink in the left column, the subsystem containing the offending block opens. The block of interest is highlighted, as shown in the following figure.



The following figure shows an HDL Code Generation Check Report that was generated for a subsystem that passed all compatibility checks. In this case, the report contains only a hyperlink to the subsystem that was checked.



Supported Blocks Library

The M-file utility `hdl1ib.m` creates a library of all blocks that are currently supported for HDL code generation. The block library, `hdlsupported.mdl`, affords quick access to all supported blocks. By constructing models using blocks from this library, you can ensure that your models are compatible with HDL code generation.

The set of supported blocks will change in future releases of the coder. To keep the `hdlsupported.mdl` current, you should rebuild the library each time you install a new release. To create the library:

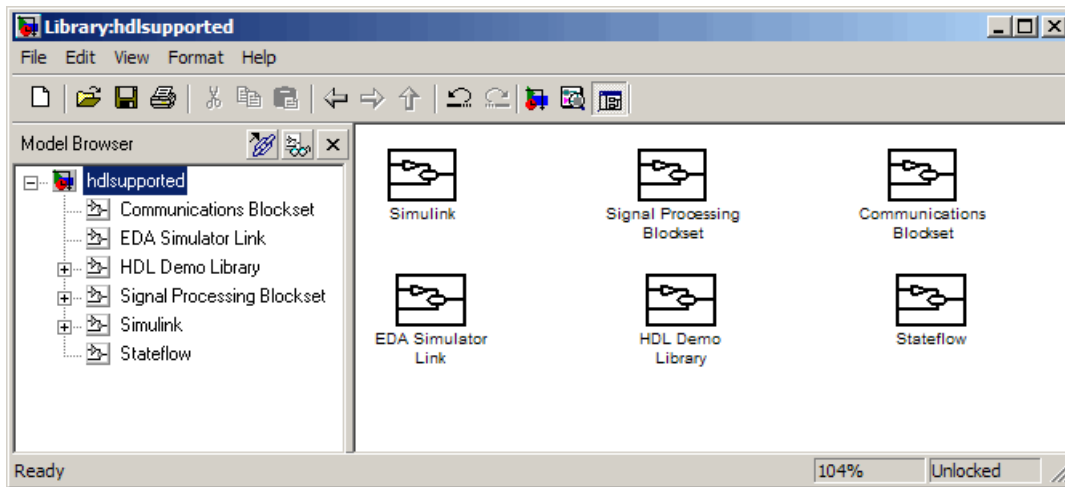
- 1 Type the following at the MATLAB prompt:

```
hdl1ib
```

`hdl1ib` starts generation of the `hdlsupported` library. Many libraries load during the creation of the `hdlsupported` library. When `hdl1ib` completes generation of the library, it does not unload these libraries.

- 2 After the library is generated, you must save it to a directory of your choice. You should retain the file name `hdlsupported.mdl`, because this document refers to the supported blocks library by that name.

The following figure shows the top-level view of the `hdlsupported.mdl` library.



Parameter settings for blocks in the hdl_supported library may differ from corresponding blocks in other libraries.

For detailed information about supported blocks and HDL block implementations, see Chapter 6, “Specifying Block Implementations and Parameters for HDL Code Generation”.

Annotating Generated Code with Comments and Requirements

The coder lets you add text annotations to generated code, in the form of comments or requirements comments. You can add annotations to your code in any of the following ways:

- Enter text directly on the block diagram as Simulink annotations. Text from Simulink annotations is rendered in generated code as plain text comments. 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 “Annotating Diagrams” in the Simulink documentation for general information on annotations.

- Place a DocBlock at the desired level of your model and enter text comments. Text from the DocBlock is rendered 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.

- Assign requirements to blocks, as described in “Adding and Viewing Requirement Links” in the *Simulink® Verification and Validation™ User’s Guide*. Requirements that you assign to Simulink blocks are automatically included as comments in generated code.

Code Tracing 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.

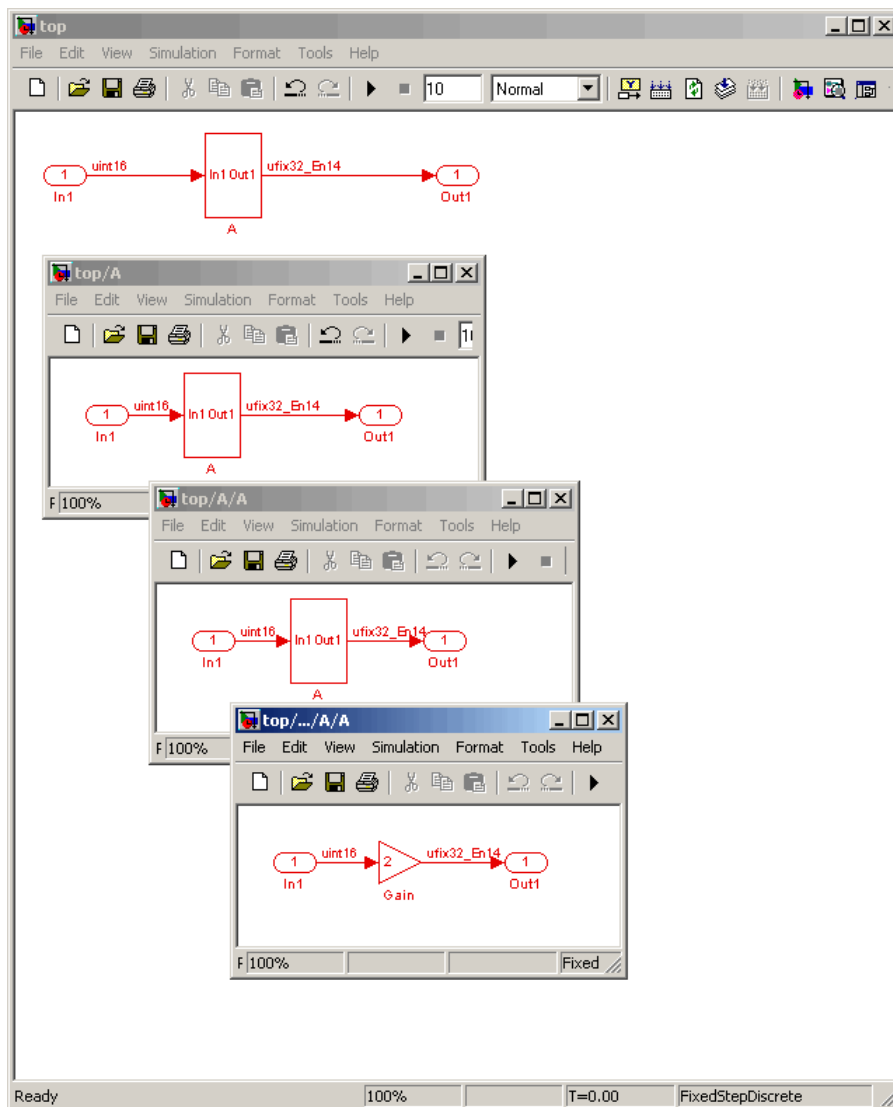
In simple cases, the mapping file may contain only one entry. For example, the `symmetric_fir` subsystem of the `sfir_fixed` demo model generates the following mapping file:

```
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('top/A')
### Working on top/A/A/A as A_entity1.vhd
### Working on top/A/A as A_entity2.vhd
### Working on top/A as A.vhd

### HDL Code Generation Complete.
```

The following example lists the contents of the resultant mapping file.

```
top/A/A/A --> A_entity1
top/A/A --> A_entity2
top/A --> A
```

Given this information, you could trace any generated entity back to its corresponding subsystem by using the `open_system` command, for example:

```
open_system('top/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: top/A
-- Created: 2005-04-20 10:23:46
-- Hierarchy Level: 0
```


Interfacing Subsystems and Models to HDL Code

- “Overview of HDL Interfaces” on page 10-2
- “Generating a Black Box Interface for a Subsystem” on page 10-3
- “Generating Interfaces for Referenced Models” on page 10-10
- “Code Generation for Enabled Subsystems” on page 10-11
- “Code Generation for HDL Cosimulation Blocks” on page 10-13
- “Customizing the Generated Interface” on page 10-15
- “Pass-Through and No-Op Implementations” on page 10-17
- “Limitation on Generated Verilog Interfaces” on page 10-18

Overview of HDL Interfaces

The coder provides a number of different ways to generate interfaces to your manually-written or legacy HDL code. Depending on your application, you may want to generate such an interface from different levels of your model:

- Subsystem
- Model referenced by a higher-level model
- HDL Cosimulation block
- RAM blocks

For most such interfaces, you can use interface generation parameters in your control file to control generation and naming of ports and other attributes of the generated interface.

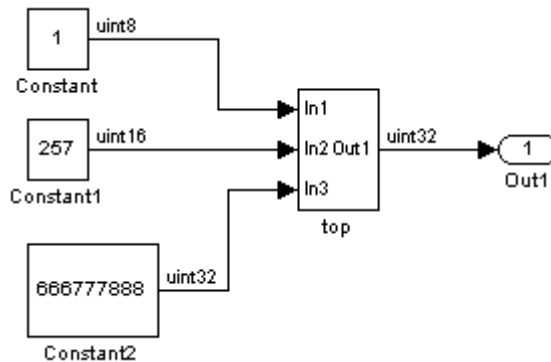
You can also generate a pass-through (wire) HDL implementation for a subsystem, or omit code generation entirely for a subsystem. Both of these techniques can be useful in cases where you need a subsystem in your simulation, but do not need the subsystem in your generated HDL code.

Generating a Black Box Interface for a Subsystem

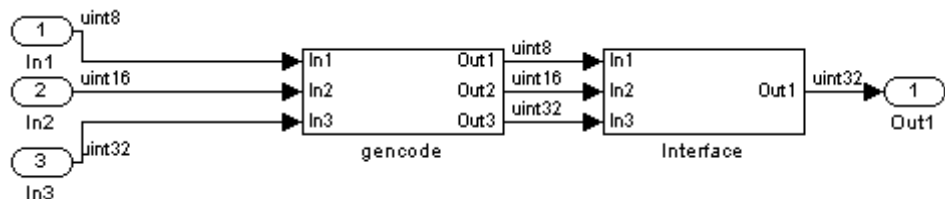
A *black box* interface for a subsystem is a generated VHDL component or Verilog module that includes only the HDL input/output port definitions for the subsystem. By generating such a component, you can use a subsystem in your model to generate an interface to existing manually written HDL code, third-party IP, or other code generated by Simulink HDL Coder.

To generate the interface, you use a control file to map one or more Subsystem blocks to the `hdldefaults.SubsystemBlackBoxHDLInstantiation` implementation. (See Chapter 5, “Code Generation Control Files” for a detailed description of the structure and use of control files.)

As an example, consider the model and subsystem shown in the following figures. The model, `subsystemst`, contains a subsystem, `top`, which is the device under test.



The subsystem `top` contains two lower-level subsystems, `gencode` and `Interface`.



Suppose that you want to generate HDL code from `top`, with a black box interface from the `Interface` subsystem. The first step would be to create a control file that defines the path and block type for the `Interface` subsystem, and maps this subsystem to the `hdldefaults.SubsystemBlackBoxHDLInstantiation` implementation. The following listing shows an example control file.

```
function control = blackbox_ctrl
control = hdlnewcontrol(mfilename);
% Generate a black box interface for the subsystem labeled
% Interface within the top-level device

control.forEach( ...
    'subsystst/top/Interface', ...
    'built-in/SubSystem', {}, ...
    'hdldefaults.SubsystemBlackBoxHDLInstantiation');
```

The control file is attached to the model when code generation is invoked. In the following `makehdl` command line, VHDL code is generated by default.

```
makehdl('subsystst/top','HDLControlFiles',{'blackbox_ctrl.m'})
### Applying User Configuration File: blackbox_ctrl.m

### Begin Vhdl Code Generation
### Working on subsystst/top/gencode as hdlsrc/gencode.vhd
### Working on subsystst/top as hdlsrc/top.vhd
### HDL Code Generation Complete.
```

In the `makehdl` progress messages, observe that the `gencode` subsystem generates a separate code 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 `subsystst/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 DOWNT0 0); -- uint8
```



```

        In2      : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In3      : IN  std_logic_vector(31 DOWNTO 0); -- uint32
        Out1     : OU  std_logic_vector(31 DOWNTO 0) -- uint32
    );
END COMPONENT;
...
u_Interface : Interface
    PORT MAP
        (clk => clk,
         clk_enable => enb_const_rate,
         reset => reset,
         In1 => gencode_out1, -- uint8
         In2 => gencode_out2, -- int16
         In3 => gencode_out3, -- uint32
         Out1 => Interface_out1 -- uint32
        );
    enb_const_rate <= clk_enable;

    ce_out <= enb_const_rate;

```

By default, the black box interface generated for subsystems includes clock, clock enable, and reset ports, as shown in the preceding example. “Customizing the Generated Interface” on page 10-15 describes how you can rename or suppress generation of these signals, and customize other aspects of the generated interface.

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` (see “Generating Selection/Action Statements with the `hdlnewforeach` Function” on page 5-24).

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 correct `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, in `package.class` form.
- (Optional) A cell array of cell arrays of strings enumerating the names of implementation parameters (if any) 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.

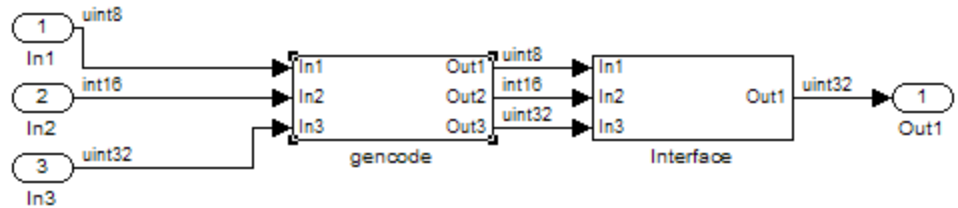
As an example, suppose that you want to generate black box control file statements for the subsystem `gencode` from the `subsystemst` model. Using `hdlnewblackbox`, you can do this as follows:

- 1 Before invoking `hdlnewblackbox`, you must build in-memory information about the model once. To do this, run `checkhdl` on the top subsystem, as shown in the following code example:

```
checkhdl('subsystemst/top')
### Starting HDL Check.
### HDL Check Complete with 0 error, 0 warning and 0 message.
```

Alternatively you can click the **Run Compatibility Checker** button in the **HDL Coder** pane of the Configuration Parameters dialog box.

- 2 Close the `checkhdl` report window and activate the `subsystemst/top` subsystem window.
- 3 Select the subsystems for which you want to create control statements. In the following figure, `gencode` is selected.



4 Deselect the subsystem/top subsystem.

5 Type the following command at the MATLAB prompt:

```
[cmd,impl,parms] = hdlnewblackbox
```

6 The command returns the following results:

```
cmd =

c.forEach('subsystem/top/gencode',...
'built-in/SubSystem', {},...
'hdldefaults.SubsystemBlackBoxHDLInstantiation', {});

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: `hdldefaults.SubsystemBlackBoxHDLInstantiation`. Also by default, no parameters are passed in for this implementation.

- 7** 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.NoHDEmission'
    'hdldefaults.SubsystemBlackBoxHDLInstantiation'
    'hdldefaults.XilinxBlackBoxHDLInstantiation'
    'hdldefaults.AlteraDSPBuilderBlackBox'
```

- 8** 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 `hdldefaults.SubsystemBlackBoxHDLInstantiation`. These are enumerated in `parms{2}`, as shown in the following listing:

```
parms{2}

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 “Customizing the Generated Interface” on page 10-15.

- 9 Having generated this information, you can now copy and paste the strings into a control file.

Generating Interfaces for Referenced Models

The Simulink model referencing feature allows you to include models in other models as blocks. Included models are referenced through Model blocks (see the “Referencing a Model” documentation for detailed information).

For Model blocks, the coder generates a VHDL component or a Verilog module instantiation. However, `makehdl` does not attempt to generate HDL code for the models referenced from Model blocks. You must generate HDL code for each referenced model individually. To generate code for a referenced model:

- 1 Select the referencing Model block.
- 2 Double-click the Model block to open the referenced model.
- 3 Invoke the `checkhdl` and `makehdl` functions to check and generate code from that model.

Note The `checkhdl` function does not check port data types within the referenced model.

The Model block is useful for multiply instantiated blocks, or for blocks for which you already have manually written HDL code. The generated HDL will contain all the 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`.

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 “Enabled Subsystems” in the Simulink documentation.

The coder supports HDL code generation for enabled subsystems that meet the following conditions:

- The enable signal must be a scalar.
- The data type of the enable signal must be either `boolean` or `ufix1`.
- All inputs and outputs of the enabled subsystem (including the enable signal) must run at the same rate.
- The **States when enabling** parameter of the Enable block is 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) is 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 EDA Simulator Link MQ, EDA Simulator Link IN, and EDA Simulator Link DS
 - `Model`
 - `simulink/Signal Attributes/Rate Transition`

- hdl demolib/FFT
- Subsystem black box (SubsystemBlackBoxHDLInstantiation)

Consider the following when using enabled subsystems in models targeted for HDL code generation:

- For synthesis results to match Simulink results, Enable ports should be driven by registered logic (with a synchronous clock) on the FPGA.
- The use of enabled subsystems can affect synthesis results in the following ways:
 - In some cases the system clock speed may drop by a small percentage.
 - In all cases more resources will be used, scaling with the number of enabled subsystem instances and the number of output ports per subsystem.

See the Automatic Gain Controller demo model for an example of the use of enabled subsystems in HDL code generation. The location of the demo is:

```
MATLABROOT\toolbox\hdlcoder\hdlcoderdemos\hdlcoder_agc.mdl
```


Code Generation for HDL Cosimulation Blocks

The coder supports HDL code generation for the HDL Cosimulation blocks provided by the following products:

- EDA Simulator Link MQ
- EDA Simulator Link IN
- EDA Simulator Link DS

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.

The documentation for each of these products contains a “Preparing for Cosimulation” section, which discusses 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 access this information using one of the following links:

- EDA Simulator Link MQ: “Define the HDL Cosimulation Block Interface (Simulink as Test Bench)”
- EDA Simulator Link IN: “Define the HDL Cosimulation Block Interface (Simulink as Test Bench)”
- EDA Simulator Link DS (UNIX® platform only): “Define the HDL Cosimulation Block Interface (Simulink as Test Bench)”

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 **Update Diagram** from the **Edit** menu.

Customizing the Generated Interface

Interface generation parameters let you customize port names and other attributes of interfaces generated for the following block types:

- simulink/Ports & Subsystems/Model
- built-in/Subsystem
- lflinklib/HDL Cosimulation
- modelsimlib/HDL Cosimulation

The following table summarizes the names, value settings, and purpose of the interface generation parameters. All parameters have string data type.

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> .
ClockEnableInputPort	Default: 'clk_enable'	Specifies HDL name for block's clock enable input port.
ClockInputPort	Default: 'clk'	Specifies HDL name for block's clock input signal.

Parameter Name	Values	Description
EntityName	Default: Entity name is derived from the block name, modified if necessary to generate a legal VHDL entity name.	Specifies VHDL entity or Verilog module name generated for the block.
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.
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'.

Pass-Through and No-Op Implementations

The coder provides special-purpose implementations that let you use a block as a wire, or simply omit a block entirely, in the generated HDL code. These implementations are summarized in the following table.

Implementation	Description
<code>hdldefaults.PassThroughHDL Emission</code>	Provides a pass-through implementation in which the block's inputs are passed directly to its outputs. (In effect, the block becomes a wire in the HDL code.)
<code>hdldefaults.NoHDL Emission</code>	Completely removes the block from the generated code. Lets you use the block in simulation but treat it as a no-op in the HDL code. You can also use this implementation as an alternative implementation for subsystems.

The coder uses these implementations for many built-in blocks (such as Scopes and Assertions) that are significant in simulation but would be meaningless in HDL code.

Limitation on Generated Verilog Interfaces

This section describes a limitation in the current release that applies to generation of Verilog interfaces for the following blocks:

- EDA Simulator Link MQ HDL Cosimulation block
- EDA Simulator Link IN HDL Cosimulation block
- EDA Simulator Link DS HDL Cosimulation block
- Model block
- Subsystem black box implementation
(`hdldefaults.SubsystemBlackBoxHDLInstantiation`)

When the target language is Verilog, only scalar ports are supported for code generation for these block types. Use of vector ports that are on these blocks will be reported as errors on the compatibility checker (`checkhdl`) report, and will raise a code generator (`makehdl`) run-time error.

Stateflow HDL Code Generation Support

- “Introduction to Stateflow HDL Code Generation” on page 11-2
- “Quick Guide to Requirements for Stateflow HDL Code Generation” on page 11-4
- “Mapping Chart Semantics to HDL” on page 11-8
- “Using Mealy and Moore Machine Types in HDL Code Generation” on page 11-15
- “Structuring a Model for HDL Code Generation” on page 11-24
- “Design Patterns Using Advanced Chart Features” on page 11-30

Introduction to Stateflow HDL Code Generation

In this section...
“Overview” on page 11-2
“Demos and Related Documentation” on page 11-2

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 any other model. 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.

Demos and Related Documentation

Demos

The following demos, 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 demo models, type the following command:

```
demos
```

This command opens the **Help** window. In the **Demos** pane on the left, select **Simulink > Simulink HDL Coder**. Then, double-click the icon for any of the following demos, and follow the instructions in the demo window.

Related Documentation

If you are familiar with Stateflow charts and Simulink models but have not yet tried HDL code generation, see the hands-on exercises in Chapter 2, “Introduction to HDL Code Generation”.

If you are not familiar with Stateflow charts, see *Stateflow Getting Started Guide*. See also the *Stateflow and Stateflow® Coder™ User’s Guide*.

Quick Guide to Requirements for Stateflow HDL Code Generation

In this section...
“Overview” on page 11-4
“Location of Charts in the Model” on page 11-4
“Data Type Usage” on page 11-4
“Chart Initialization” on page 11-5
“Registered Output” on page 11-5
“Restrictions on Imported Code” on page 11-6
“Other Restrictions” on page 11-6

Overview

This section summarizes the requirements and restrictions you should follow when configuring Stateflow charts that are intended to target HDL code generation. “Mapping Chart Semantics to HDL” on page 11-8 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 “Structuring a Model for HDL Code Generation” on page 11-24 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 cannot be guaranteed to 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 all 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.

Registered Output

The chart property **Initialize Outputs Every Time Chart Wakes Up** exists specifically for HDL code generation. This property lets you control whether output is persistent (stored in registers) from one sample time to the next. Such use of registers is termed *registered output*.

When the **Initialize Outputs Every Time Chart Wakes Up** option is deselected (the default), registered output is used.

When the **Initialize Outputs Every Time Chart Wakes Up** option is selected, registered output is not used. A default initial value (defined in the **Initial value** field of the **Value Attributes** pane of the Data Properties

dialog box) is given to each output when the chart wakes up. This assignment guarantees that there is no reference to outputs computed in previous time steps.

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. Any information entered in the Target Options dialog box is ignored.

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 machine-parented data, machine-parented events, or 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:

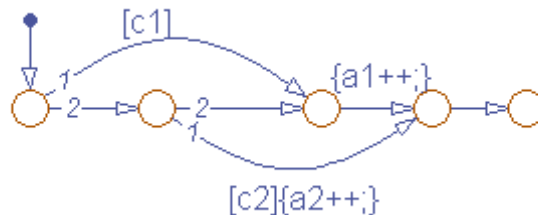
- `wakeup`
- `tick`

Temporal logic can be used provided the base events are limited to these types of implicit events.

- Do not use recursion through graphical functions. The coder does not currently support recursion.
- Do not explicitly use loops other than for loops, such as in flow diagrams.

Only constant-bounded loops are supported for HDL code generation. See the FOR Loop demo (sf_for.mdl) to learn how to create a for loop using a graphical function.

- 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 outputs are not registered. (Outputs are not registered if the **Initialize Outputs Every Time Chart Wakes Up** option is selected. See also “Registered Output” on page 11-5.)
- 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 the coder cannot guarantee that results obtained from the generated HDL code are bit-true to results obtained from the simulation. Recommended practice is to enable overflow checking and eliminate overflow conditions from the model during simulation.

Mapping Chart Semantics to HDL

In this section...

“Software Realization of Chart Semantics” on page 11-8

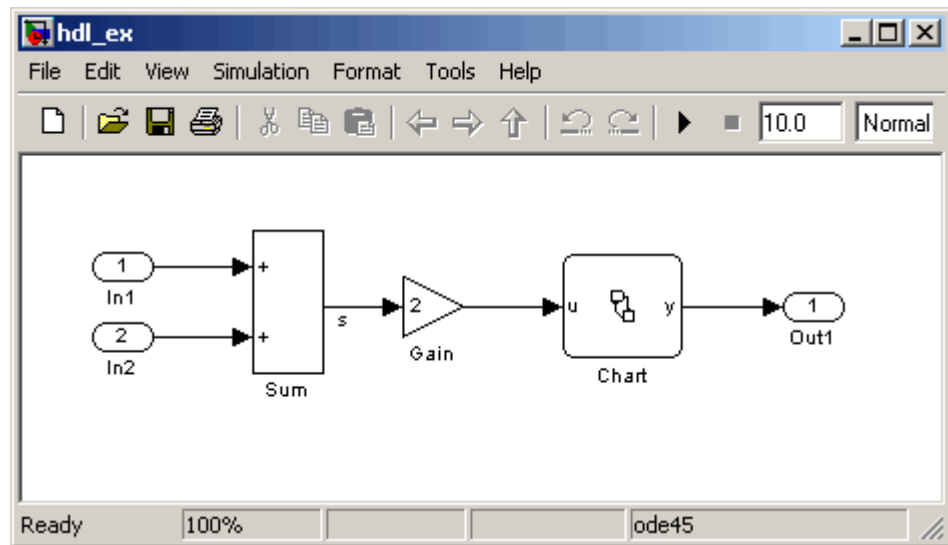
“Hardware Realization of Stateflow Semantics” on page 11-10

“Restrictions for HDL Realization” on page 11-13

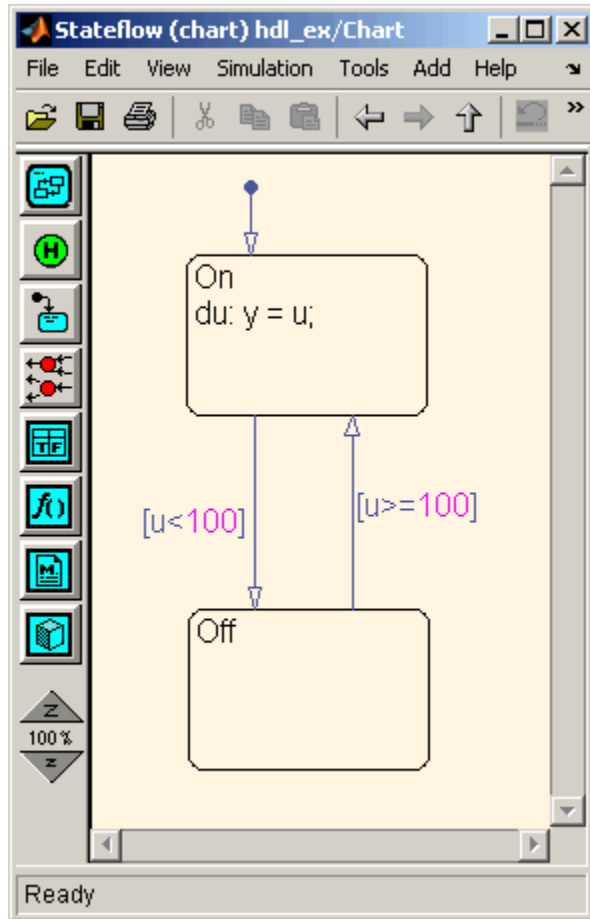
Software Realization of Chart Semantics

The top-down semantics of a chart describe how the chart executes. chart semantics describe an explicit sequential execution order for elements of the chart, such as states and transitions. These deterministic, sequential semantics map naturally to sequential programming languages, such as C. To support the rich semantics of a chart in the Simulink environment, it is necessary to combine the state variable updates and output computation in a single function.

Consider the example model shown in the following figure. The root level of the model contains three blocks (Sum, Gain and a Stateflow chart) connected in series.



The chart from the model is shown in the following figure.



The following Real-Time Workshop® C code excerpt was generated from this example model. The code illustrates how the chart combines the output computation and state-variable update.

```

/* Output and update for atomic system: '<Root>/Chart' */
void hdl_ex_Chart(void)
{
    /* Stateflow: '<Root>/Chart' */

```

```
switch (hdl_ex_DWork.Chart.is_c1_hdl_ex) {
case hdl_ex_IN_Off:
    if (hdl_ex_B.Gain >= 100.0) {
        hdl_ex_DWork.Chart.is_c1_hdl_ex = (uint8_T)hdl_ex_IN_On;
    }

    break;

case hdl_ex_IN_On:
    if (hdl_ex_B.Gain < 100.0) {
        hdl_ex_DWork.Chart.is_c1_hdl_ex = (uint8_T)hdl_ex_IN_Off;
    } else {
        hdl_ex_B.y = hdl_ex_B.Gain;
    }

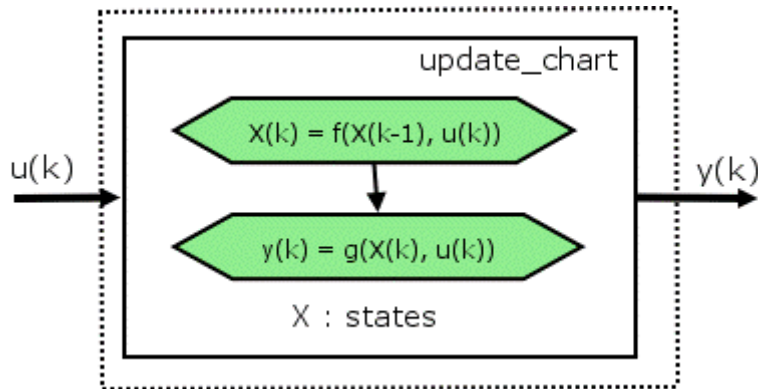
    break;

default:
    hdl_ex_DWork.Chart.is_c1_hdl_ex = (uint8_T)hdl_ex_IN_On;
    break;
}
}
```

The preceding code assigns either the state or the output, but not both. Values of output variables, as well as state, persist from one time step to another. If an output value is not assigned during a chart execution, the output simply retains its value (as defined in a previous execution).

Hardware Realization of Stateflow Semantics

The following diagram shows a sequential implementation of Stateflow semantics for output/update computations, appropriate for targeting the C language.



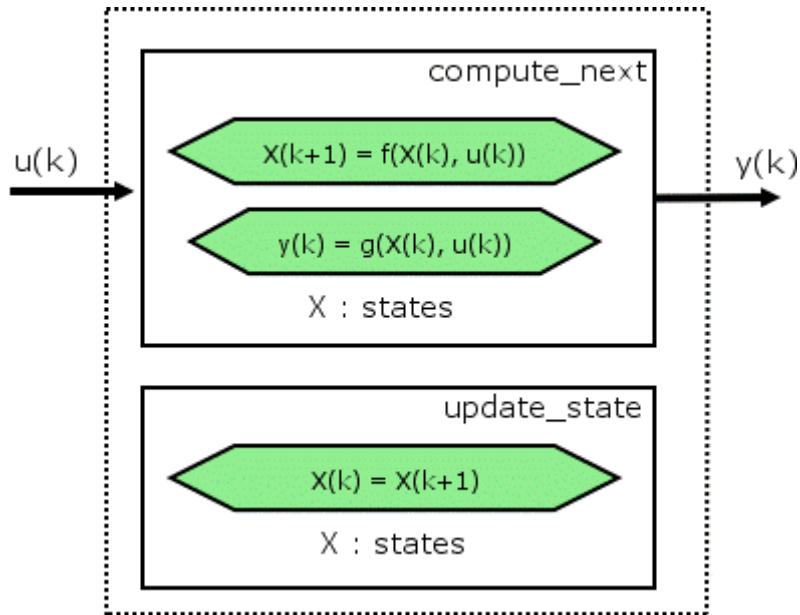
A mapping from Stateflow semantics to an HDL implementation demands a different approach. The following requirements must be met:

- **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 ordering must be correct.

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, appropriate 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, all 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 safely 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. The execution sequence is automatically correct.

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 “Quick Guide to Requirements for Stateflow HDL Code Generation” on page 11-4. 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 M or any language other than HDL. For example, do not call M 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 Chapter 10, “Interfacing Subsystems and Models to HDL Code”.

- 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 machine-parented data or 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 machine-parented 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 needed for HDL because outputs must be available at time 0 (hardware reset). You must select this option to ensure bit-true HDL code generation.

Using Mealy and Moore Machine Types in HDL Code Generation

In this section...

“Overview” on page 11-15

“Generating HDL for a Mealy Finite State Machine” on page 11-16

“Generating HDL Code for a Moore Finite State Machine” on page 11-19

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. See “Building Mealy and Moore Charts” for detailed information on 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 to ensure that they conform 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.

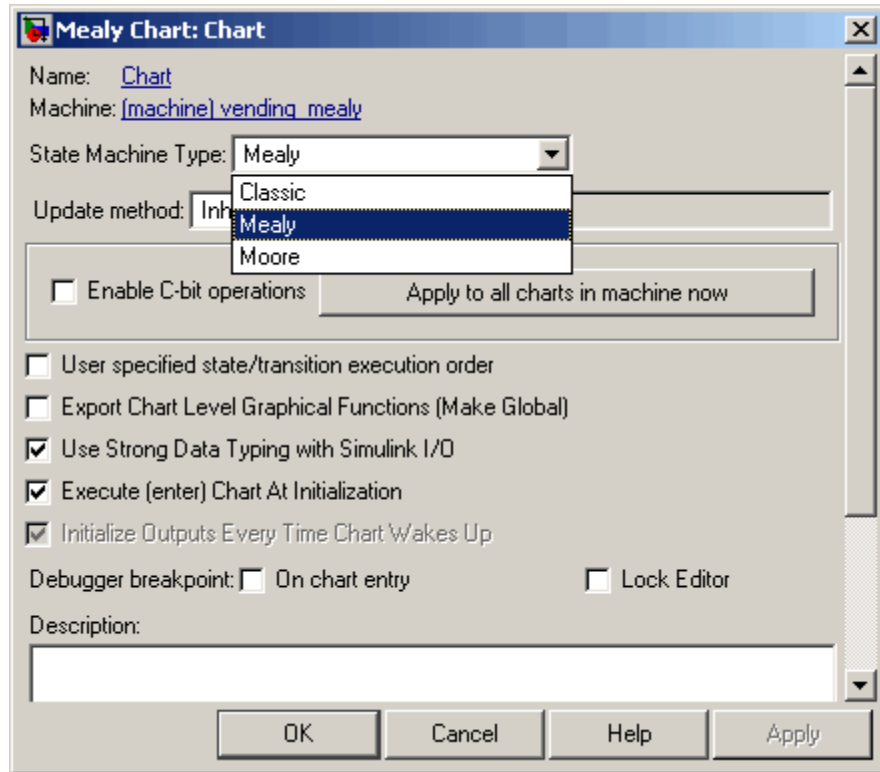
The execution of a Mealy or Moore chart at time t is the evaluation of the function represented by that chart at time t . The initialization property for output ensures that every output is defined at every time step. Specifically, the output of a Mealy or Moore chart at one time step must not depend 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

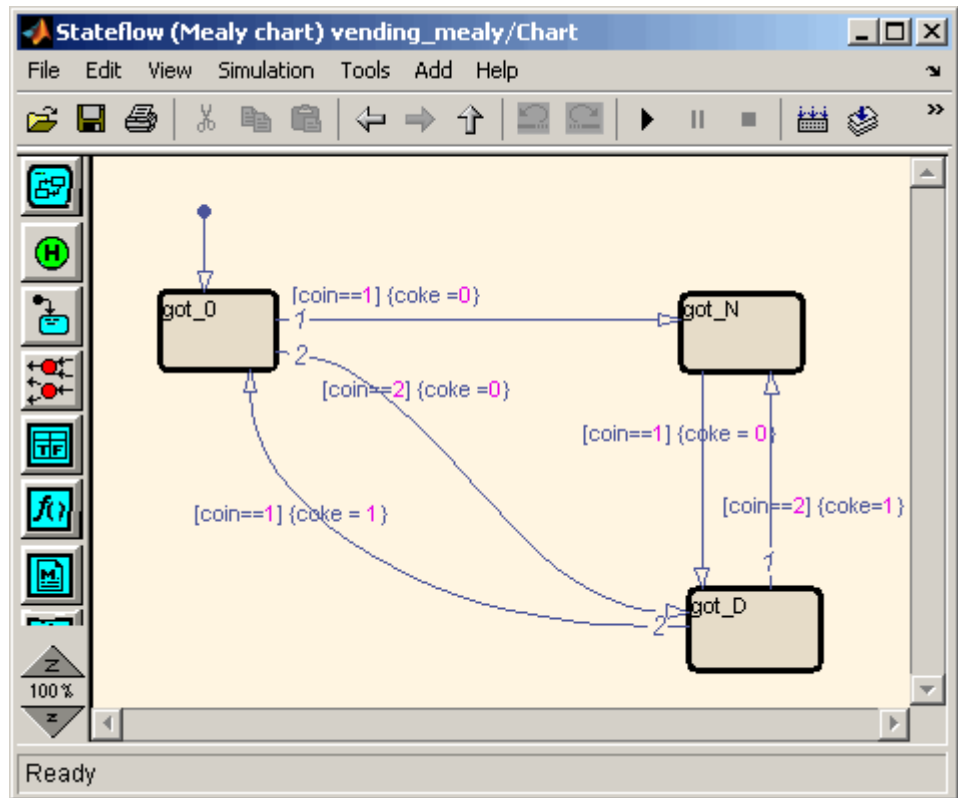
- The chart meets all general code generation requirements, as described in “Quick Guide to Requirements for Stateflow HDL Code Generation” on page 11-4.
- The **Initialize Outputs Every Time Chart Wakes Up** option is selected. This option is selected automatically when the Mealy option is selected from the **State Machine Type** pop-up menu, as shown in the following figure.



- Actions are associated with transitions inner and outer transitions only.

Mealy actions are associated with transitions. In Mealy machines, output computation is expected to be driven by the change on inputs. In fact, the dependence of output on input is the fundamental distinguishing factor between the formal definitions of Mealy and Moore machines. The requirement that actions be given on transitions is to some degree stylistic, rather than necessary to enforce Mealy semantics. However, it is natural that output computation follows input conditions on input, because transition conditions are primarily input conditions in any machine type.

The following figure shows an example of a chart that models a Mealy state machine.



The following code example lists the VHDL process code generated for the Mealy chart.

```

Chart : PROCESS (is_Chart, coin)
  -- local variables
  BEGIN
    is_Chart_next <= is_Chart;
    coke <= '0';

    CASE is_Chart IS
      WHEN IN_got_0 =>

        IF coin = 1.0 THEN
          coke <= '0';

```



```
        is_Chart_next <= IN_got_N;
    ELSIF coin = 2.0 THEN
        coke <= '0';
        is_Chart_next <= IN_got_D;
    END IF;

    WHEN IN_got_D =>

        IF coin = 2.0 THEN
            coke <= '1';
            is_Chart_next <= IN_got_N;
        ELSIF coin = 1.0 THEN
            coke <= '1';
            is_Chart_next <= IN_got_0;
        END IF;

    WHEN IN_got_N =>

        IF coin = 1.0 THEN
            coke <= '0';
            is_Chart_next <= IN_got_D;
        END IF;

    WHEN OTHERS =>
        is_Chart_next <= IN_got_0;
    END CASE;

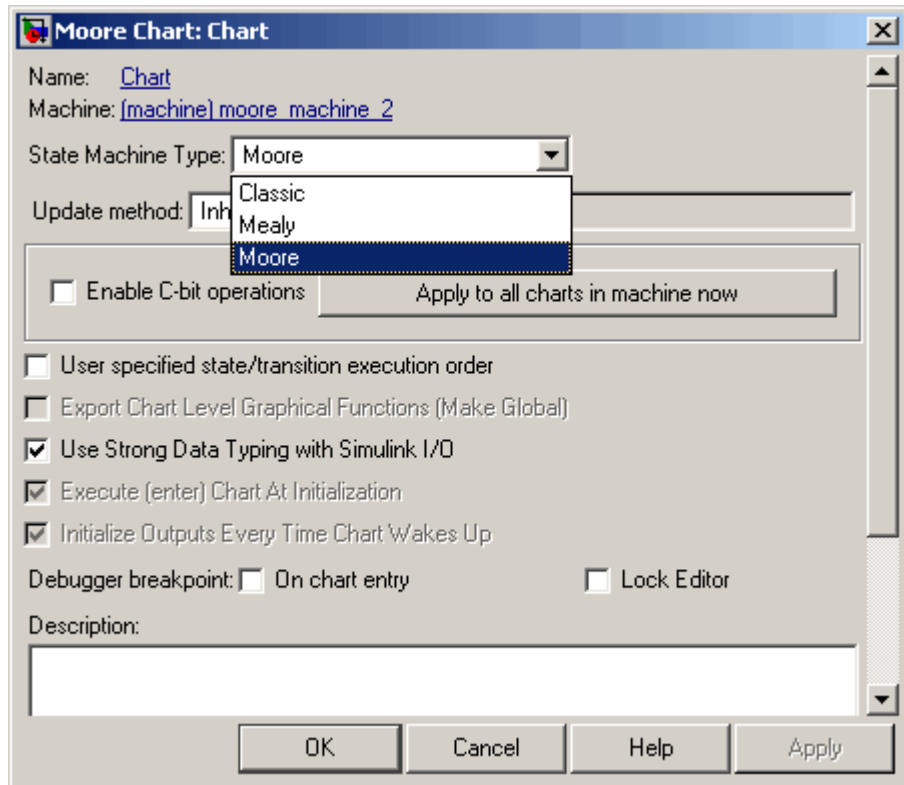
END PROCESS Chart;
```

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 all general code generation requirements, as described in “Quick Guide to Requirements for Stateflow HDL Code Generation” on page 11-4.

- The **Initialize Outputs Every Time Chart Wakes Up** option is selected. This option is selected automatically when the Moore option is selected from the **State Machine Type** pop-up menu, as shown in the following figure.



- 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.

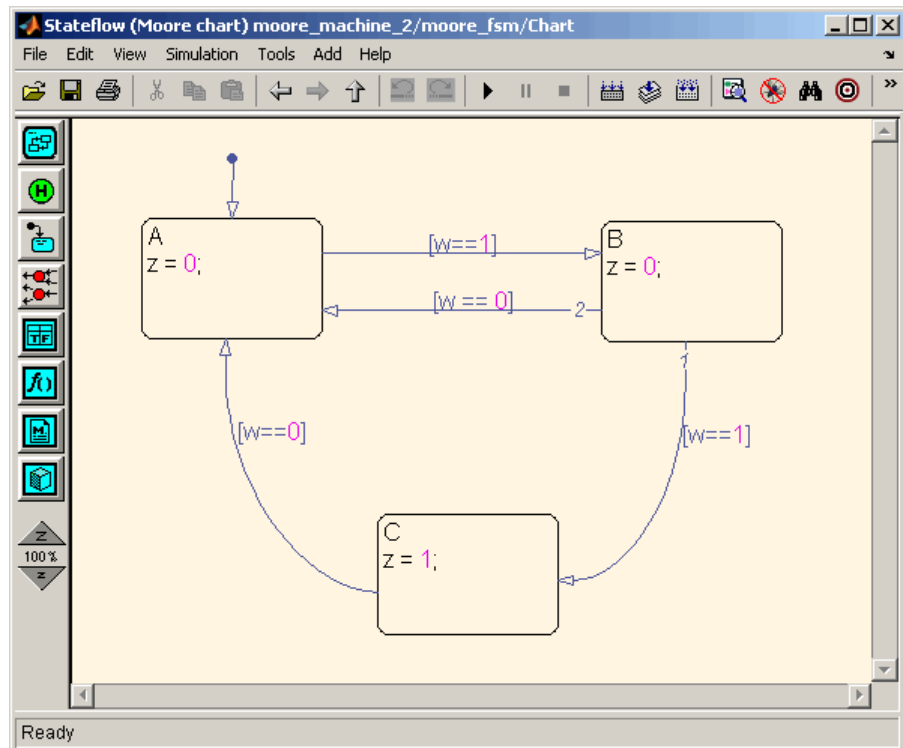
- No local data or graphical functions are used.

Function calls and local data are not allowed in a Moore chart. This ensures that output does not depend 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.

This restriction guarantees that the chart's top-down semantics compute outputs as if actions were evaluated strictly before inner and outer flow diagrams.

The following figure shows a Stateflow chart of a Moore state machine.



The following code example illustrates generated Verilog code for the Moore chart.

```
Chart : PROCESS (is_Chart, w)
  -- local variables
  VARIABLE is_Chart_temp : T_state_type_is_Chart;
BEGIN
  is_Chart_temp := is_Chart;
  z <= '0';

  CASE is_Chart_temp IS
    WHEN IN_A =>
      z <= '0';
    WHEN IN_B =>
      z <= '0';
    WHEN IN_C =>
      z <= '1';
    WHEN OTHERS =>
      is_Chart_temp := IN_NO_ACTIVE_CHILD;
  END CASE;

  CASE is_Chart_temp IS
    WHEN IN_A =>

      IF w = '1' THEN
        is_Chart_temp := IN_B;
      END IF;

    WHEN IN_B =>

      IF w = '1' THEN
        is_Chart_temp := IN_C;
      ELSIF w = '0' THEN
        is_Chart_temp := IN_A;
      END IF;

    WHEN IN_C =>

      IF w = '0' THEN
        is_Chart_temp := IN_A;
      END IF;
```

```
        WHEN OTHERS =>
            is_Chart_temp := IN_A;
        END CASE;

        is_Chart_next <= is_Chart_temp;
    END PROCESS Chart;
```

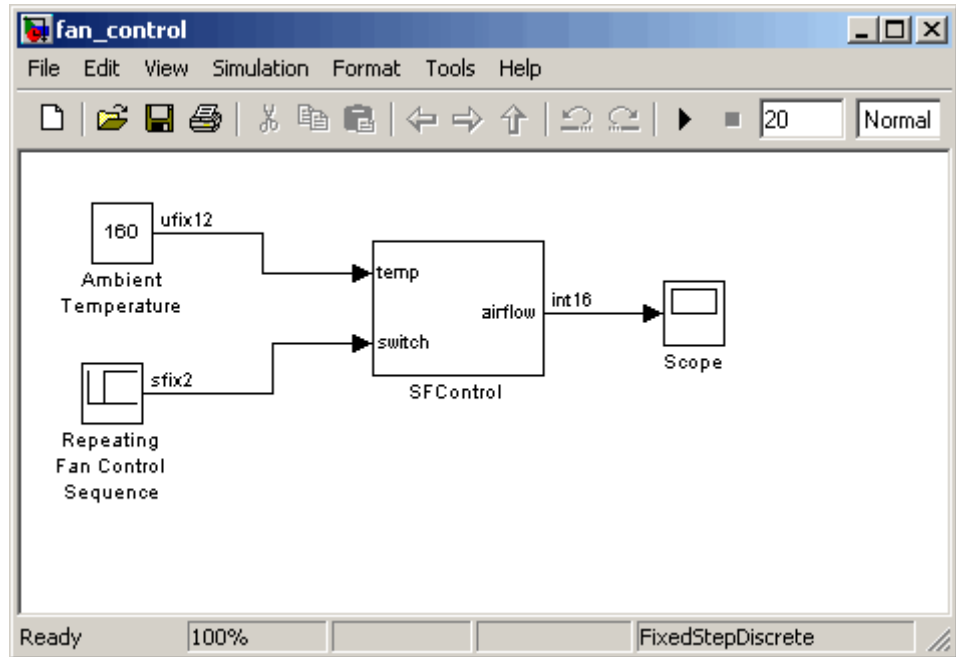
Structuring a Model for HDL Code Generation

In general, generation of VHDL or Verilog code from a model containing a Stateflow chart does not differ greatly from HDL code generation from any other model.

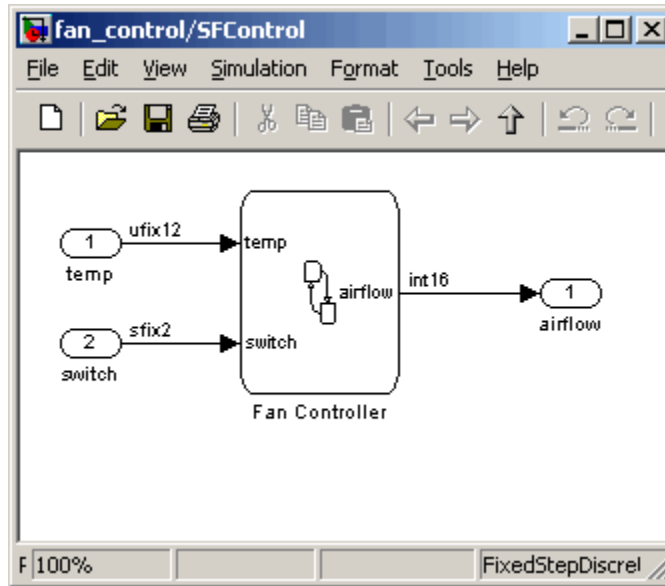
A chart intended for HDL code generation *must* be part of a subsystem that represents the Device Under Test (DUT). The DUT corresponds to the top level VHDL entity or Verilog module for which code is generated, tested and eventually synthesized. The top level Simulink components that drive the DUT correspond 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 appropriate signals to the subsystem inputs and outputs. In most cases, you can do this by simply clicking on the chart and then selecting **Edit > Create Subsystem** in the model window.

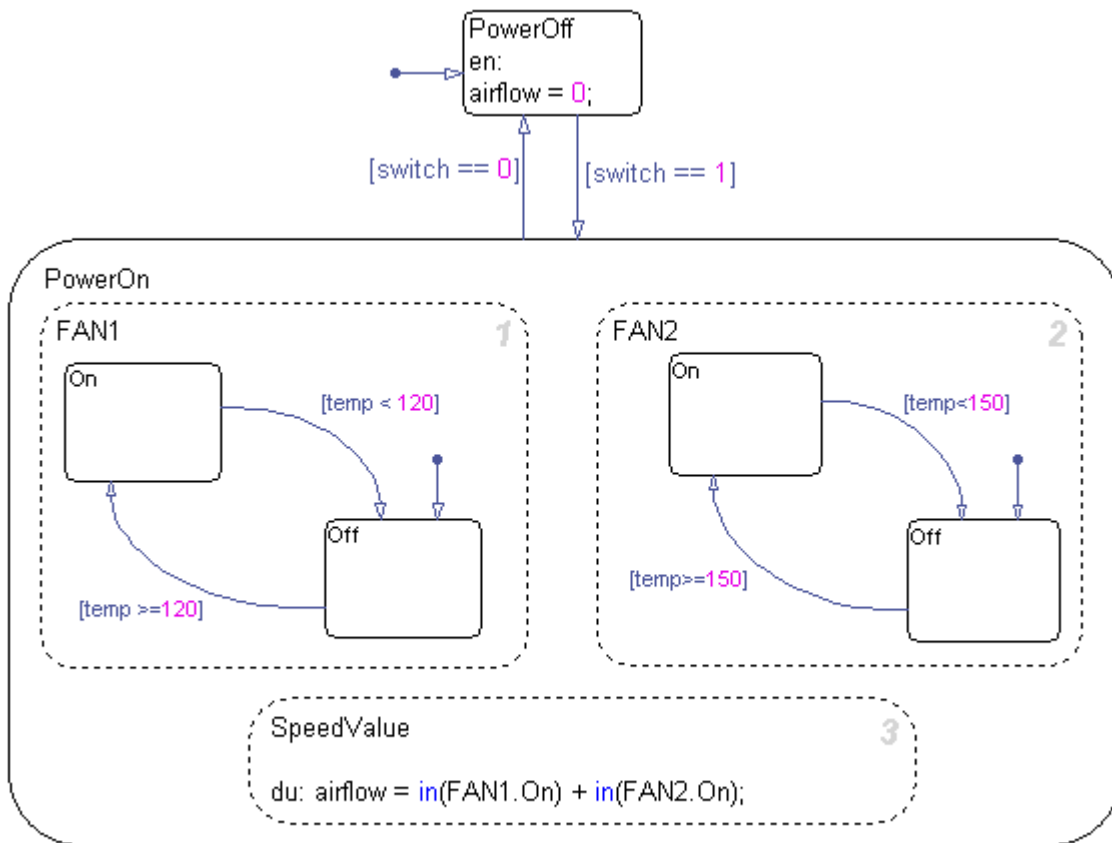
As an example of a properly structured model, consider the `fan_control` model shown in the following figure. In this model, the subsystem `SFControl` is the DUT. Two input signals drive the DUT.



The SFControl subsystem, shown in the following figure, contains a Stateflow chart, Fan Controller. The chart that has two inputs and an output.



The Fan Controller chart, shown in the following figure, models a simple system that monitors input temperature data (temp) and turns on the two fans (FAN1 and FAN2) based on the range of the temperature. A manual override input (switch) is provided to turn the fans off forcibly. At each time step the Fan Controller outputs a value (airflow) representing the number of fans that are turned on.



The following makehdl command generates VHDL code (by default) for the subsystem containing the chart.

```
makehdl(`fan_control/SF_Control')
```

As code generation for this subsystem proceeds, the coder displays progress messages as shown in the following listing:

```
### Begin VHDL Code Generation
### Working on fan_control/SFControl as hd1src/SFControl.vhd
```

```
### Working on fan_control/SFControl/Fan Controller as hdlsrc\Fan_Controller.vhd
Stateflow parsing for model "fan_control"...Done
Stateflow code generation for model "fan_control"...Done
### HDL Code Generation Complete.
```

As the progress messages indicate, the coder generates a separate code file for each level of hierarchy in the model. The following VHDL files are written to the target directory, `hdlsrc`:

- `Fan_Controller.vhd` contains the entity and architecture code (`Fan_Controller`) for the chart.
- `SFControl.vhd` contains the code for the top level subsystem. This file also instantiates a `Fan_Controller` component.

The coder also generates a number of other files (such as scripts for HDL simulation and synthesis tools) in the target directory. See the “HDL Code Generation Defaults” on page 17-23 for full details on generated files.

The following code excerpt shows the entity declaration generated for the `Fan_Controller` chart in `Fan_Controller.vhd`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY Fan_Controller IS
    PORT (
        clk : IN std_logic;
        clk_enable : IN std_logic;
        reset : IN std_logic;
        temp : IN std_logic_vector(11 DOWNTO 0);
        b_switch : IN std_logic_vector(1 DOWNTO 0);
        airflow : OUT std_logic_vector(15 DOWNTO 0));
END Fan_Controller;
```

This model shows the use of fixed point data types without scaling (e.g. `ufix12`, `sfix2`), as supported for HDL code generation. At the entity/instantiation boundary, all signals in the generated code are typed as `std_logic` or `std_logic_vector`, following general VHDL coding standard

conventions. In the architecture body, these signals are assigned to the corresponding typed signals for further manipulation and access.

Design Patterns Using Advanced Chart Features

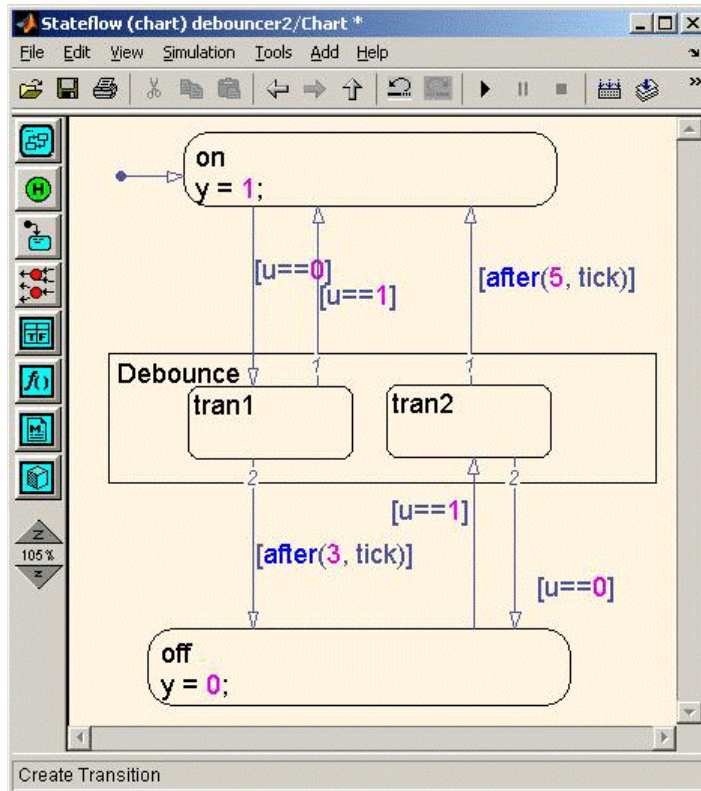
In this section...
“Temporal Logic” on page 11-30
“Graphical Function” on page 11-33
“Hierarchy and Parallelism” on page 11-35
“Stateless Charts” on page 11-39
“Truth Tables” on page 11-42

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 from states, and in state actions. Although temporal logic does not introduce any new events into a Stateflow model, it is useful to think of the change of value of a temporal logic condition as an event. You can use temporal logic operators in many cases where a counter is required. A common use case would be to use temporal logic to implement a time-out counter.

For detailed information about temporal logic, see “Using Temporal Logic in State Actions and Transitions”.

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 : PROCESS (is_Chart, temporalCounter_i1, y_reg, u)
  -- local variables
  VARIABLE temporalCounter_i1_temp : unsigned(7 DOWNTO 0);
BEGIN
  is_Chart_next <= is_Chart;
  y_reg_next <= y_reg;
  temporalCounter_i1_temp := temporalCounter_i1;

  IF temporalCounter_i1_temp < to_unsigned(7, 8) THEN
    temporalCounter_i1_temp :=
tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(temporalCounter_i1_temp, 9), 10)
+ tmw_to_unsigned(to_unsigned(1, 9), 10), 8);

```

```

END IF;

CASE is_Chart IS
  WHEN IN_tran1 =>

    IF u = '1' THEN
      is_Chart_next <= IN_on;
      y_reg_next <= '1';
    ELSIF temporalCounter_i1_temp >= to_unsigned(3, 8) THEN
      is_Chart_next <= IN_off;
      y_reg_next <= '0';
    END IF;

  WHEN IN_tran2 =>

    IF temporalCounter_i1_temp >= to_unsigned(5, 8) THEN
      is_Chart_next <= IN_on;
      y_reg_next <= '1';
    ELSIF u = '0' THEN
      is_Chart_next <= IN_off;
      y_reg_next <= '0';
    END IF;

  WHEN IN_off =>

    IF u = '1' THEN
      is_Chart_next <= IN_tran2;
      temporalCounter_i1_temp := to_unsigned(0, 8);
    END IF;

  WHEN IN_on =>

    IF u = '0' THEN
      is_Chart_next <= IN_tran1;
      temporalCounter_i1_temp := to_unsigned(0, 8);
    END IF;

  WHEN OTHERS =>
    is_Chart_next <= IN_on;

```

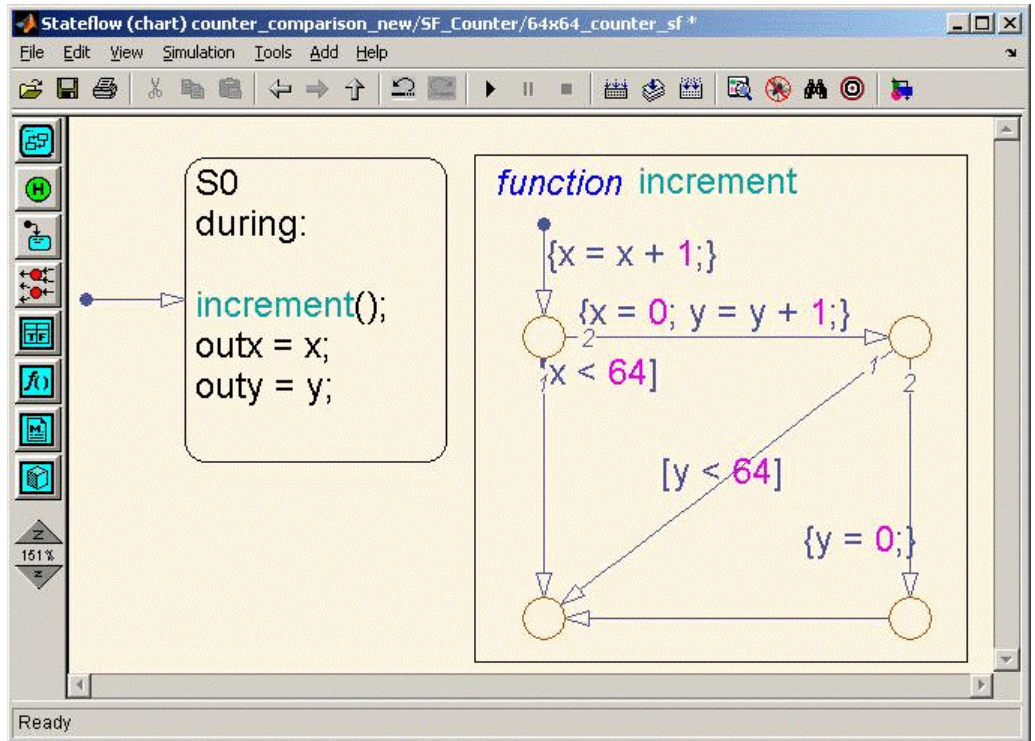
```
        y_reg_next <= '1';  
    END CASE;  
  
    temporalCounter_i1_next <= temporalCounter_i1_temp;  
END PROCESS Chart;
```

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 “Stateflow Chart Notation” chapter of the Stateflow documentation includes a detailed description of graphical functions.

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).

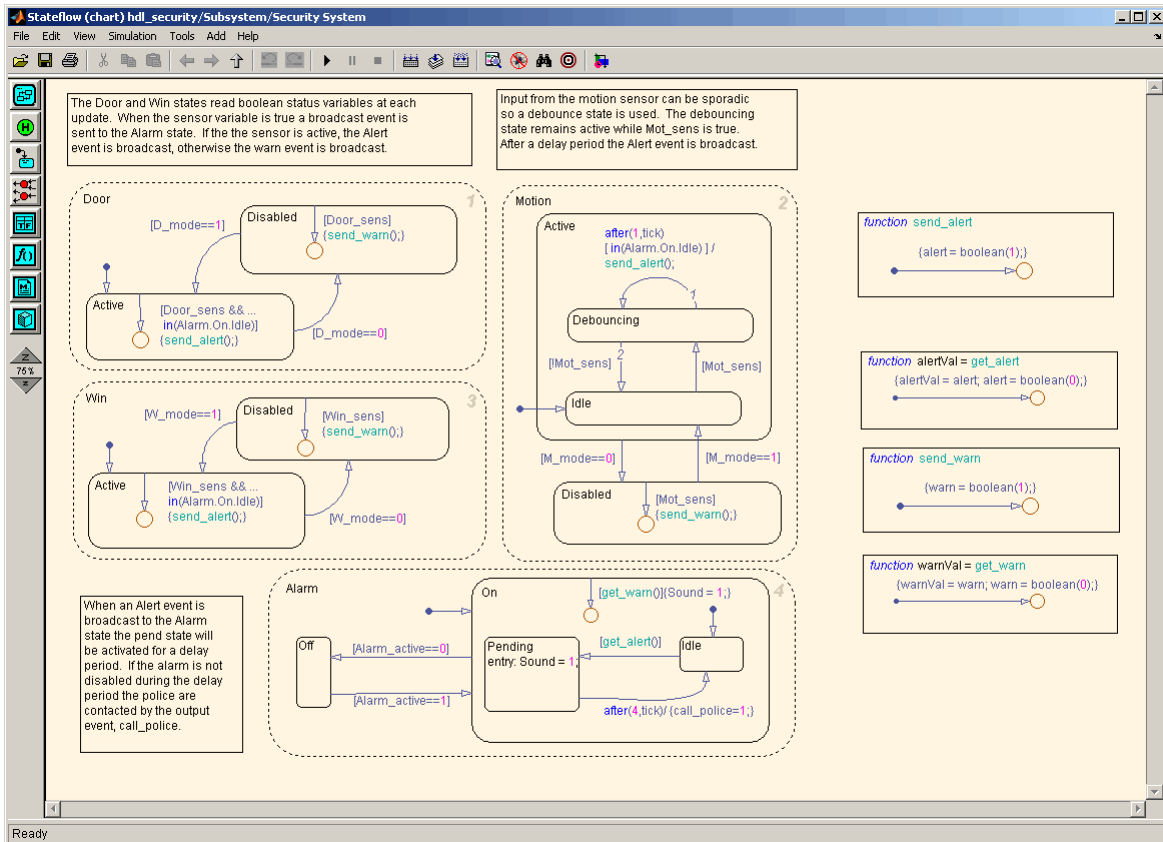
For detailed information on hierarchy and parallelism, see “Stateflow Hierarchy of Objects” and “Execution Order for Parallel States”.

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.

The following figure shows a chart that models a security system. The chart contains

- Simultaneously active parallel states (in order of execution: Door, Motion, Win, Alarm).
- Hierarchy, where the parallel states contain child states. For example, the Motion state contains Active and Inactive states, and the Active state contains further nested states (Debouncing and Idle).
- Graphical functions (such as `send_alert` and `send_warn`) that set and reset flags, simulating broadcast and reception of events. These functions are used, rather than local events, because local events are not supported for HDL code generation.



The following VHDL code excerpt was generated for the parallel Door and Motion states from this chart. The higher-level CASE statements corresponding to Door and Motion are generated sequentially to match Stateflow simulation semantics. The hierarchy of nested states maps to nested CASE statements in VHDL.

```

CASE is_Door IS
  WHEN IN_Active =>

    IF D_mode = '0' THEN
      is_Door_next <= IN_Disabled;
    ELSIF tmw_to_boolean(Door_sens AND tmw_to_stdlogic(is_On = IN_Idle)) THEN

```

```

        alert_temp := '1';
    END IF;

    WHEN IN_Disabled =>

        IF D_mode = '1' THEN
            is_Door_next <= IN_Active;
        ELSIF tmw_to_boolean(Door_sens) THEN
            warn_temp := '1';
        END IF;

    WHEN OTHERS =>
        --On the first sample call the door mode is set to active.
        is_Door_next <= IN_Active;
    END CASE;

    --This state models the modes of a motion detector sensor and implements logic
    -- to respond when that sensor is producing a signal.

    CASE is_Motion IS
        WHEN IN_Active =>

            IF M_mode = '0' THEN
                is_Active_next <= IN_NO_ACTIVE_CHILD;
                is_Motion_next <= IN_Disabled;
            ELSE

                CASE is_Active IS
                    WHEN IN_Debouncing =>

                        IF tmw_to_boolean('1'
                            AND tmw_to_stdlogic(temporalCounter_i2_temp >=
                                to_unsigned(1, 8)))
                            AND tmw_to_stdlogic(is_On = IN_Idle))
                        THEN

                            alert_temp := '1';
                            is_Active_next <= IN_Debouncing;
                            temporalCounter_i2_temp := to_unsigned(0, 8);
                        ELSIF tmw_to_boolean( NOT Mot_sens) THEN
                            is_Active_next <= b_IN_Idle;

```

```

        END IF;

    WHEN b_IN_Idle =>

        IF tmw_to_boolean(Mot_sens) THEN
            is_Active_next <= IN_Debouncing;
            temporalCounter_i2_temp := to_unsigned(0, 8);
        END IF;

    WHEN OTHERS =>
        NULL;
    END CASE;

```

Stateless Charts

Charts consisting of pure flow diagrams (i.e., charts having no states) are useful in capturing *if-then-else* constructs used in procedural languages like C. The “Stateflow Chart Notation” chapter in the Stateflow documentation discusses flow diagrams in detail.

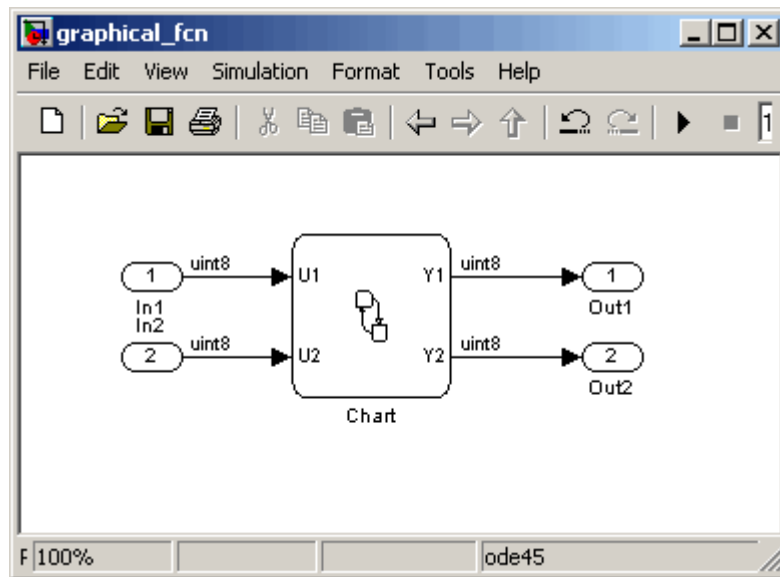
As an example, consider the following logic, expressed in C-like pseudocode.

```

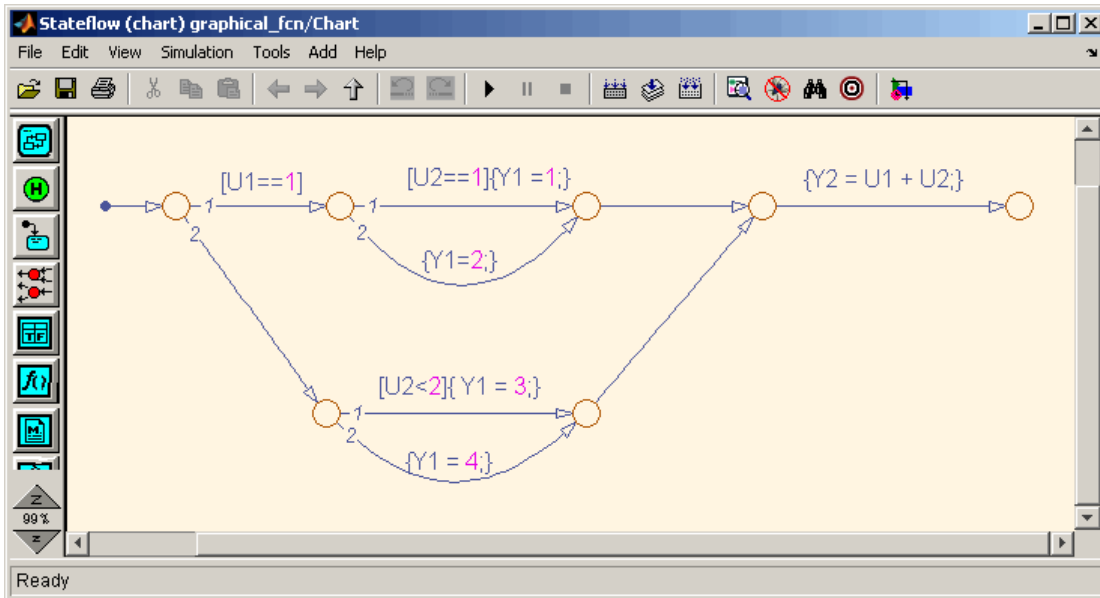
if(U1==1) {
    if(U2==1) {
        Y = 1;
    }else{
        Y = 2;
    }
}else{
    if(U2<2) {
        Y = 3;
    }else{
        Y = 4;
    }
}

```

The following figures illustrate how to model this control flow using a stateless chart. The root model contains a subsystem and inputs and outputs to the chart.



The following figure shows the flow diagram that implements the if-then-else logic.



The following generated VHDL code excerpt shows the nested IF-ELSE statements obtained from the flow diagram.

```

Chart : PROCESS (Y1_reg, Y2_reg, U1, U2)
  -- local variables
BEGIN
  Y1_reg_next <= Y1_reg;
  Y2_reg_next <= Y2_reg;

  IF unsigned(U1) = to_unsigned(1, 8) THEN

    IF unsigned(U2) = to_unsigned(1, 8) THEN
      Y1_reg_next <= to_unsigned(1, 8);
    ELSE
      Y1_reg_next <= to_unsigned(2, 8);
    END IF;
  END IF;

```

```
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 chart (see “Truth Table Functions” in the Stateflow documentation)
- Truth Table blocks in Simulink models (see Truth Table in the Stateflow documentation)

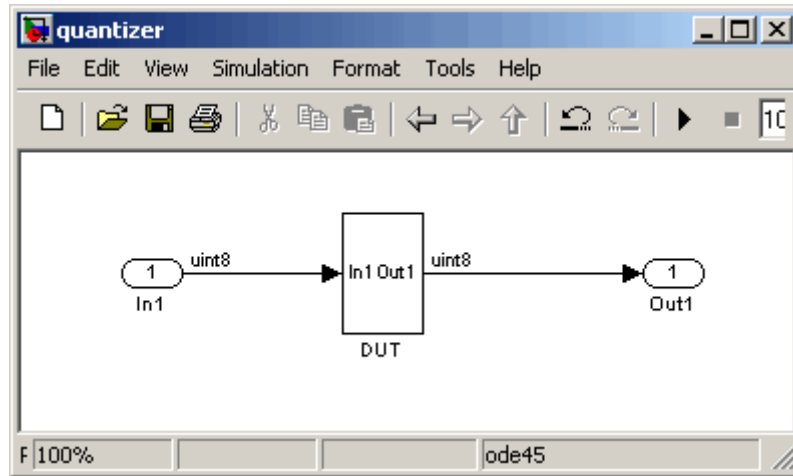
This section examines a Truth Table function in a chart, and the VHDL code generated for the chart.

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:

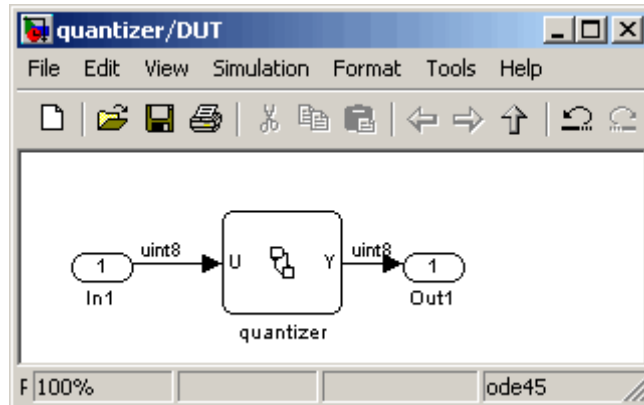
```
Y = 1 when 0 <= U <= 10
Y = 2 when 10 < U <= 17
Y = 3 when 17 < U <= 45
Y = 4 when 45 < U <= 52
Y = 5 when 52 < U
```

A stateless chart with a single call to a Truth Table function can represent this logic succinctly.

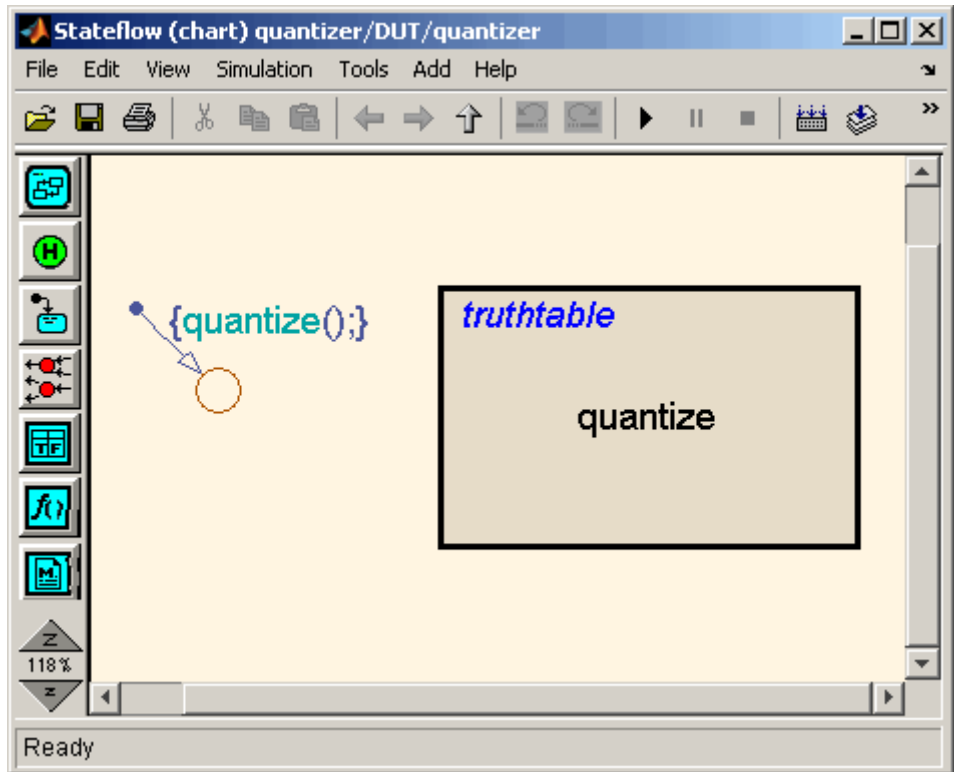
The following figure shows a model containing a subsystem, DUT.



The subsystem contains a chart, quantizer, as shown in the following figure.



The next figure shows the quantizer chart, containing the Truth Table.



The following figure shows the threshold logic, as displayed in the Truth Table Editor.

The screenshot shows the Stateflow (truth table) quantizer/DUT/quantizer.quantize Truth Table Editor. The interface includes a menu bar (File, Edit, Settings, Add, Help) and a toolbar with various icons. The main area is divided into two sections: the Condition Table and the Action Table.

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 Embedded MATLAB Function Block

- “Introduction” on page 12-2
- “Tutorial Example: Incrementer” on page 12-4
- “Useful Embedded MATLAB Function Block Design Patterns for HDL” on page 12-25
- “Using Fixed-Point Bitwise Functions” on page 12-39
- “Using Complex Signals” on page 12-49
- “Distributed Pipeline Insertion” on page 12-58
- “Recommended Practices” on page 12-68
- “Language Support” on page 12-72
- “Other Limitations” on page 12-81

Introduction

In this section...

“HDL Applications for the Embedded MATLAB Function Block” on page 12-2

“Related Documentation and Demos” on page 12-3

HDL Applications for the Embedded MATLAB Function Block

The Embedded 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 an Embedded 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 Embedded 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 Embedded 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 Embedded MATLAB Function block.

The Embedded MATLAB Function block supports a powerful 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 deinterleaver
- Modulators and demodulators

- Multipath channel models; impairment models
- Timing recovery algorithms
- Viterbi algorithm; Maximum Likelihood Sequence Estimation (MLSE)
- Adaptive equalizer algorithms

Related Documentation and Demos

The following documentation and demos provide further information on the Embedded MATLAB Function block.

Related Documentation

For general documentation on the Embedded MATLAB Function block, see:

- “Using the Embedded MATLAB Function Block”
- Embedded MATLAB Function block reference

The coder supports most of the fixed-point runtime library functions supported by the Embedded MATLAB Function block. See “Working with the Fixed-Point Embedded MATLAB Subset” in the Fixed-Point Toolbox documentation for a complete list of these functions, and general information on limitations that apply to the use of Fixed-Point Toolbox with the Embedded MATLAB function block.

Demos

The `hdlcoderviterbi2.mdl` demo models a Viterbi decoder, incorporating an Embedded MATLAB Function block for use in simulation and HDL code generation. To open the model, type the following command at the MATLAB command prompt:

```
hdlcoderviterbi2
```

The `hdlcodercpu_em1.mdl` demo models a CPU with a Harvard RISC architecture, incorporating many Embedded MATLAB Function blocks to simulate and generate code for CPU and memory elements. To open the model, type the following command at the MATLAB command prompt:

```
hdlcodercpu_em1
```

Tutorial Example: Incrementer

In this section...

“Example Model Overview” on page 12-4

“Setting Up” on page 12-7

“Creating the Model and Configuring General Model Settings” on page 12-8

“Adding an Embedded MATLAB Function Block to the Model” on page 12-8

“Setting Optimal Fixed-Point Options for the Embedded MATLAB Function Block” on page 12-10

“Programming the Embedded MATLAB Function Block” on page 12-12

“Constructing and Connecting the DUT_eML_Block Subsystem” on page 12-15

“Compiling the Model and Displaying Port Data Types” on page 12-20

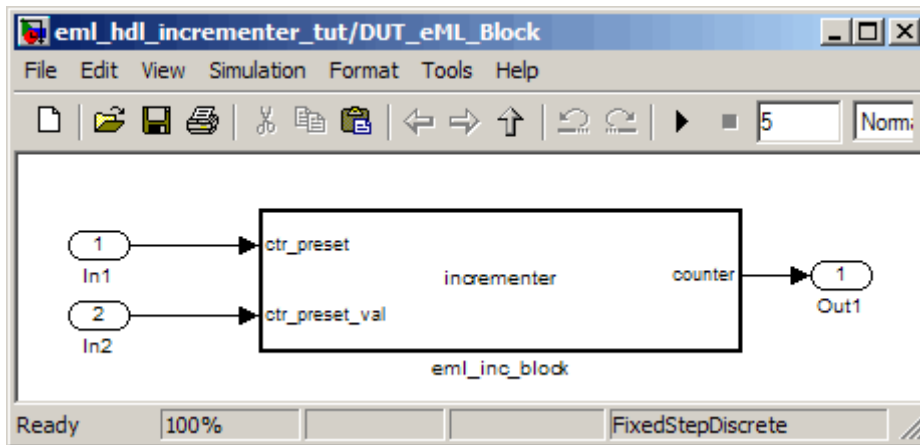
“Simulating the eml_hdl_incrementer_tut Model” on page 12-20

“Generating HDL Code” on page 12-21

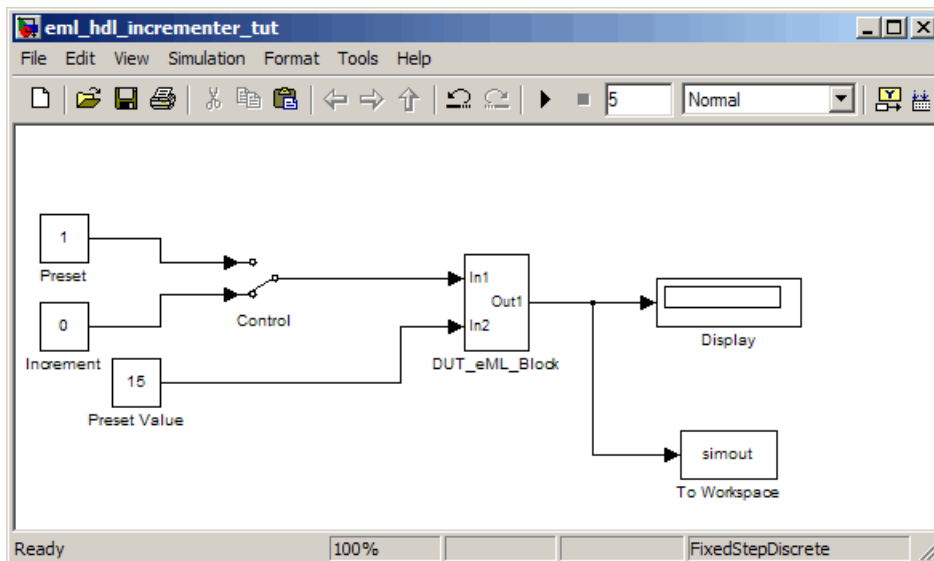
Example Model Overview

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 an Embedded 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 Embedded MATLAB Function block. The function returns the counter value (`counter`) at the output of the Embedded MATLAB Function block.

The Embedded MATLAB Function block is contained in a subsystem, `DUT_eML_Block`. The subsystem functions as the device under test (DUT) from which HDL code is generated. The following figure shows the subsystem.



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 any HDL code.) The following figure shows the model.



Tip If you do not want to construct the model step by step, or do not have time, the example model is available in the demos directory as the following file:

```
MATLABROOT\toolbox\hdlcoder\hdlcoderdemos\eml_hdl_incrementer.mdl
```

After you open the model, save a copy of it to your local directory as `eml_hdl_incrementer_tut.mdl`.

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 supported
% for HDL code generation from Embedded MATLAB Function blocks.
%
% 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 = numericitytype(0,14,0);

fm = hdlfimath;
hdl_fi = fi(val, nt, fm);
```

Setting Up

Before you begin building the example model, set up a working directory for your model and generated code.

Setting Up a Directory

- 1 Start the MATLAB software.
- 2 Create a directory named `eml_tut`, for example:

```
mkdir D:\work\eml_tut
```

The `eml_tut` directory stores the model you create, and also contains directories and generated code. The location of the directory does not matter, except that it should not be within the MATLAB directory tree.

- 3 Make the `eml_tut` directory your working directory, 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, using the M-file utility, `hdlsetup.m`. 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`” on page 2-8 for further information about `hdlsetup`.

To set the model parameters:

- 1 Create a new model.
- 2 Save the model as `eml_hdl_incrementer_tut.mdl`.
- 3 At the MATLAB command prompt, type:

```
hdlsetup('eml_hdl_incrementer_tut')
```

- 4 Select **Configuration Parameters** from the **Simulation** menu in the `eml_hdl_incrementer_tut` model window.

The Configuration Parameters dialog box opens with the **Solver** options pane displayed.

- 5 Set the following **Solver** options, which are useful in simulating this model:

Fixed step size: 1

Stop time: 5

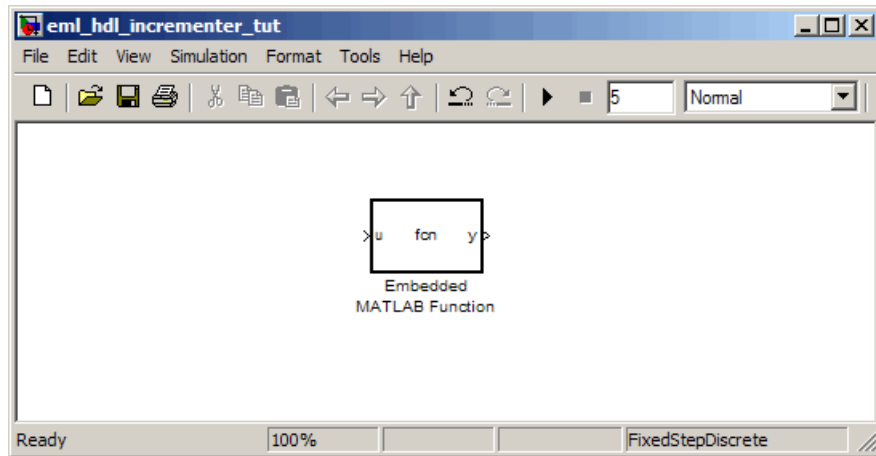
- 6 Click **Apply**. Then close the Configuration Parameters dialog box.
- 7 Select **Save** from the Simulink **File** menu, to save the model with its new settings.

Adding an Embedded MATLAB Function Block to the Model

- 1 Open the Simulink Library Browser. Then, select the Simulink/User-Defined Functions sublibrary.

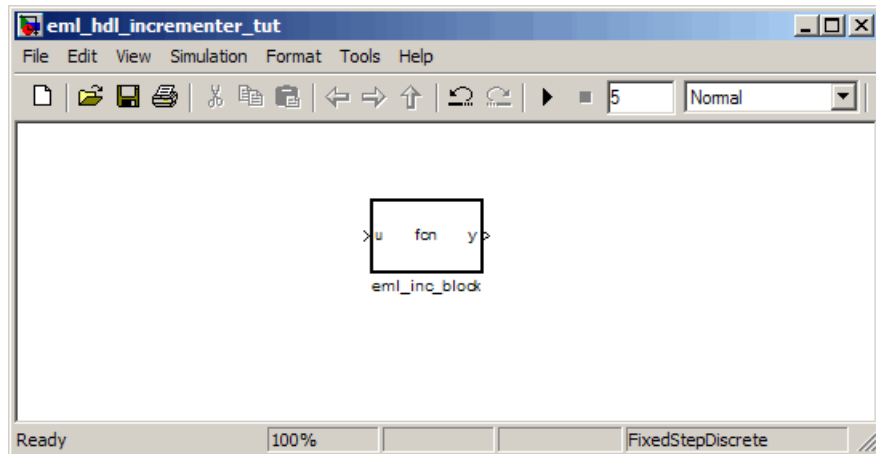
- 2 Select the Embedded MATLAB Function block from the library window and add it to the model.

The model should now appear as shown on the following figure.



- 3 Change the block label from Embedded MATLAB Function to eml_inc_block.

The model should now appear as shown on the following figure.



- 4 Save the model.
- 5 Close the Simulink Library Browser window.

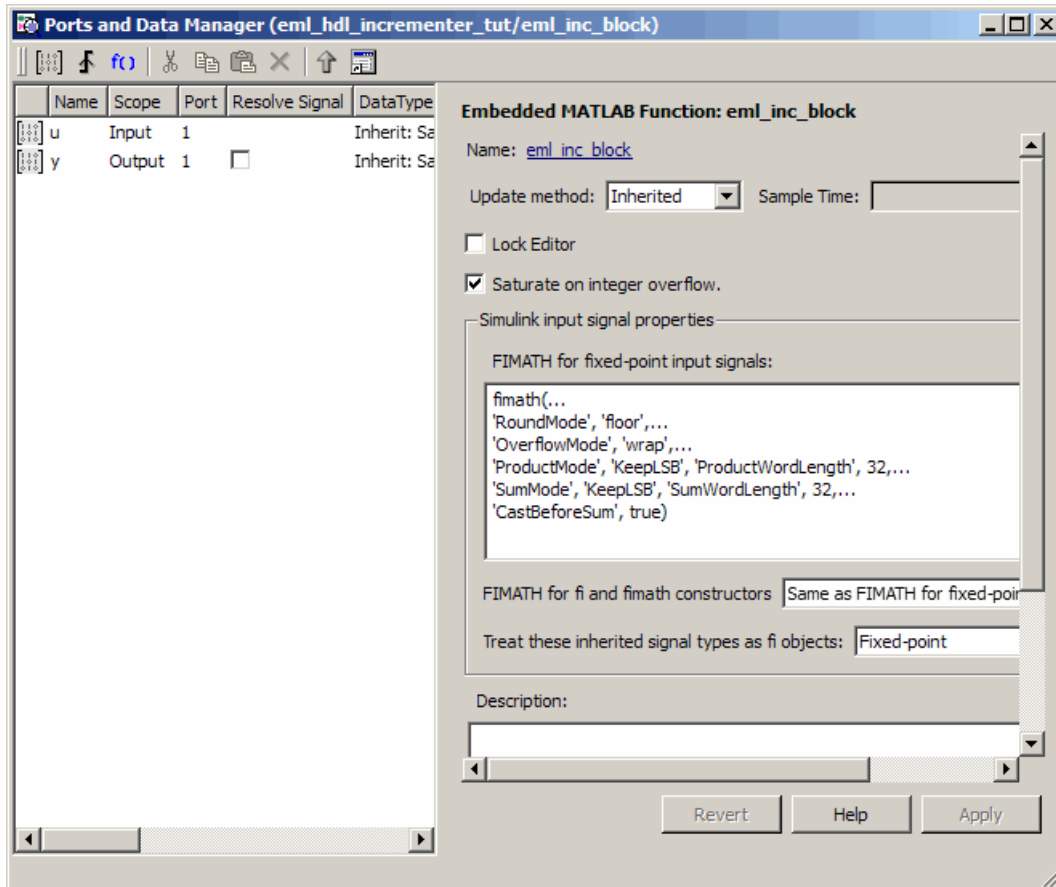
Setting Optimal Fixed-Point Options for the Embedded MATLAB Function Block

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 Embedded 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:

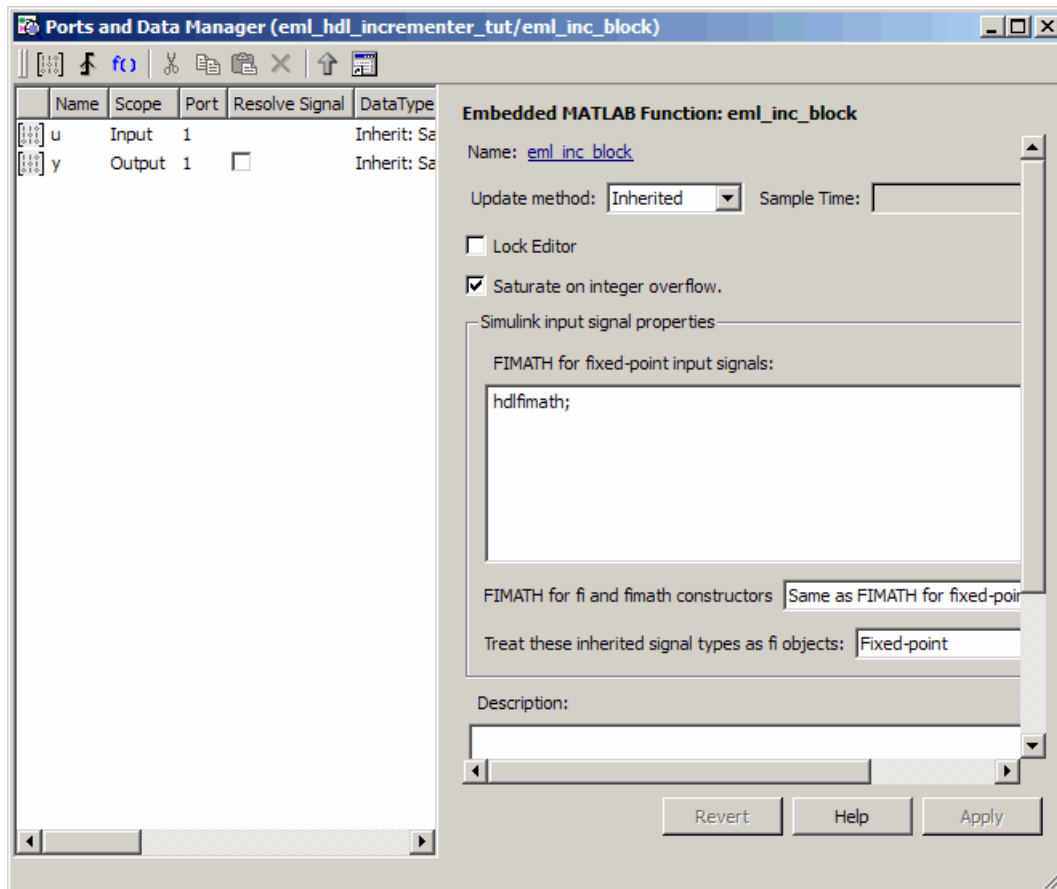
- 1 If it is not already open, open the `eml_hdl_incrementer_tut` model that you created in “Adding an Embedded MATLAB Function Block to the Model” on page 12-8.
- 2 Double-click the Embedded MATLAB Function block to open it for editing. The Embedded MATLAB Function block editor appears.
- 3 Select **Edit Data/Ports** from the **Tools** menu. The Ports and Data Manager dialog box opens, displaying the default FIMATH specification and other properties for the Embedded MATLAB Function block.



- 4** The M-function `hdlfimath.m` is a utility that defines a FIMATH specification that is optimized for HDL code generation. Replace the default **FIMATH for fixed-point signals** specification with a call to `hdlfimath` as follows:

```
hdlfimath;
```

- 5** Click **Apply**. The Embedded MATLAB Function block properties should now appear as shown in the following figure.



6 Close the Ports and Data Manager dialog box.

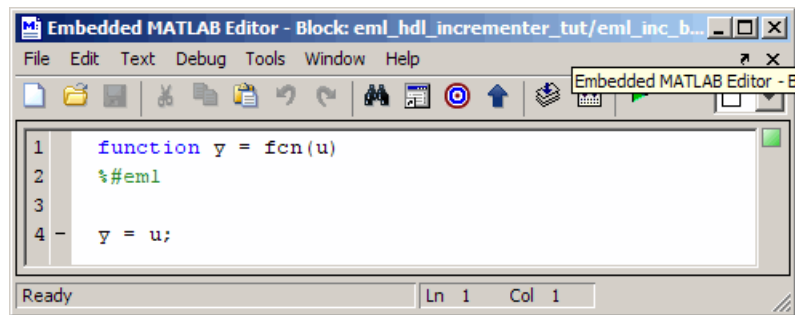
7 Save the model.

Programming the Embedded MATLAB Function Block

The next step is add code to the Embedded MATLAB Function block to define the incrementer function, and then use diagnostics to check for errors.

To program the function:

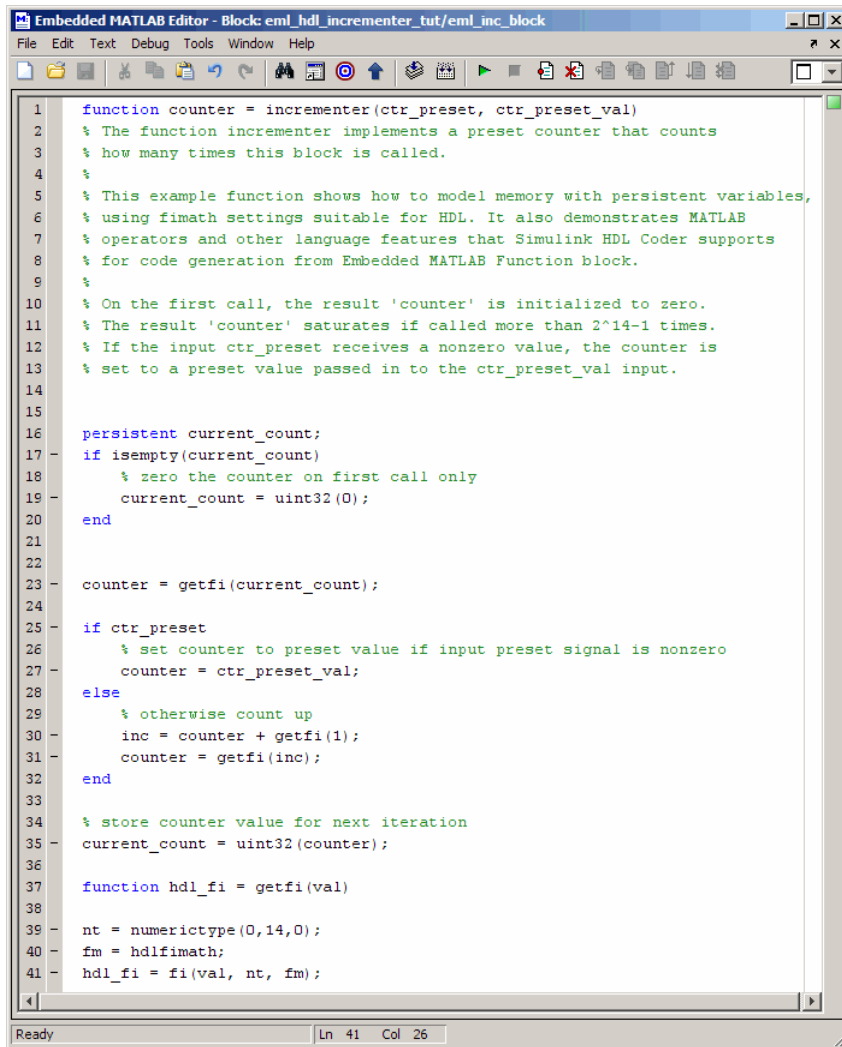
- 1 If not already open, open the `eml_hdl_incrementer_tut` model that you created in “Adding an Embedded MATLAB Function Block to the Model” on page 12-8.
- 2 Double-click the Embedded MATLAB Function block to open it for editing. The Embedded MATLAB Editor appears. The editor displays a default function definition, as shown in the following figure.



The next step is to replace the default function with the incrementer function.

- 3 select **Select All** fromn the **Edit** menu of the Embedded MATLAB Editor. Then, delete all the default code.
- 4 Copy the complete `incrementer` function definition from the listing given in “The Incrementer Function Code” on page 12-6, and paste it into the editor.

The Embedded MATLAB Editor should appear as shown in the following figure:



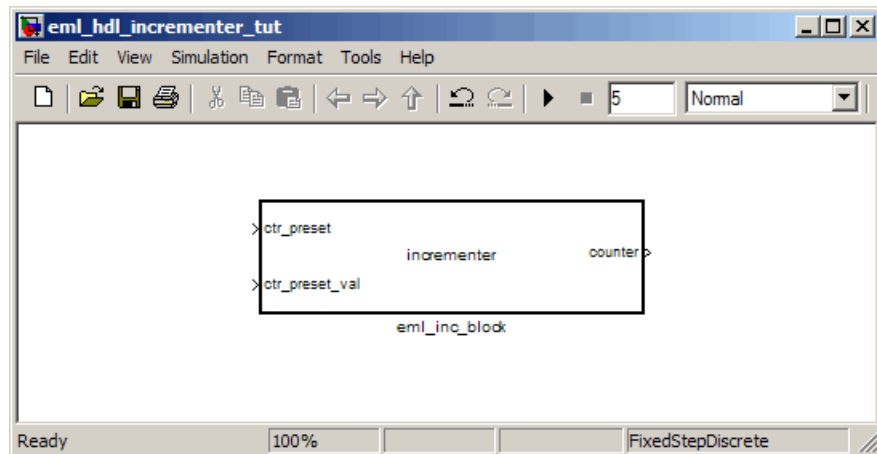
```
1 function counter = incrementer(ctr_preset, ctr_preset_val)
2 % The function incrementer implements a preset counter that counts
3 % how many times this block is called.
4 %
5 % This example function shows how to model memory with persistent variables,
6 % using fimath settings suitable for HDL. It also demonstrates MATLAB
7 % operators and other language features that Simulink HDL Coder supports
8 % for code generation from Embedded MATLAB Function block.
9 %
10 % On the first call, the result 'counter' is initialized to zero.
11 % The result 'counter' saturates if called more than 2^14-1 times.
12 % If the input ctr_preset receives a nonzero value, the counter is
13 % set to a preset value passed in to the ctr_preset_val input.
14
15
16 persistent current_count;
17 if isempty(current_count)
18     % zero the counter on first call only
19     current_count = uint32(0);
20 end
21
22
23 counter = getfi(current_count);
24
25 if ctr_preset
26     % set counter to preset value if input preset signal is nonzero
27     counter = ctr_preset_val;
28 else
29     % otherwise count up
30     inc = counter + getfi(1);
31     counter = getfi(inc);
32 end
33
34 % store counter value for next iteration
35 current_count = uint32(counter);
36
37 function hdl_fi = getfi(val)
38
39 nt = numerictype(0,14,0);
40 fm = hdlfimath;
41 hdl_fi = fi(val, nt, fm);
```

5 Select **Save Model** from the **File** menu in the Embedded MATLAB Editor.

Saving the model updates the model window, redrawing the Embedded MATLAB Function block.

Changing the function header of the Embedded MATLAB Function block makes the following changes to the Embedded MATLAB Function block in the model:

- 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. The model should now resemble the following figure.



- 7 Save the model again.

Constructing and Connecting the DUT_eML_Block Subsystem

This section assumes that you have completed “Programming the Embedded MATLAB Function Block” on page 12-12 with a successful build. In this section, you construct a subsystem containing the `incrementer` function block, to be used as the device under test (DUT) from which HDL code is generated. 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 incremter function block as follows:

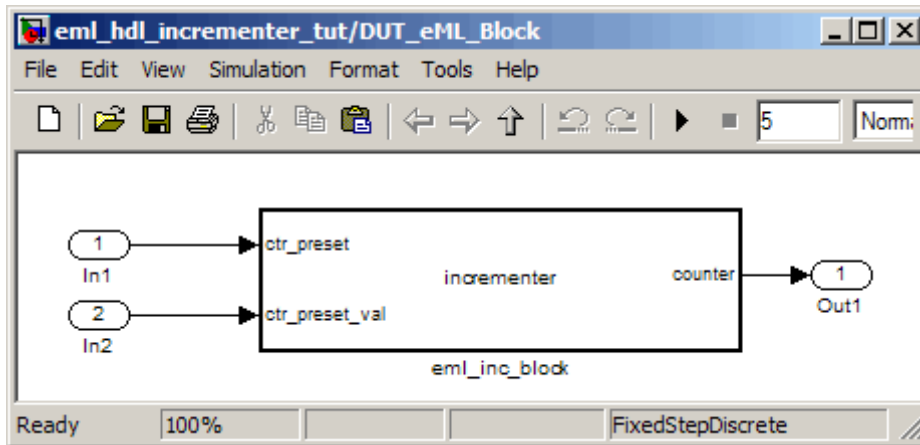
- 1 Click the incremter function block.
- 2 From the **Edit** menu, select **Create Subsystem**.

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 Embedded MATLAB Function Block

- 1 Double-click the subsystem to view its interior. As shown in the following figure, the subsystem contains the incremter function block, with input and output ports connected.

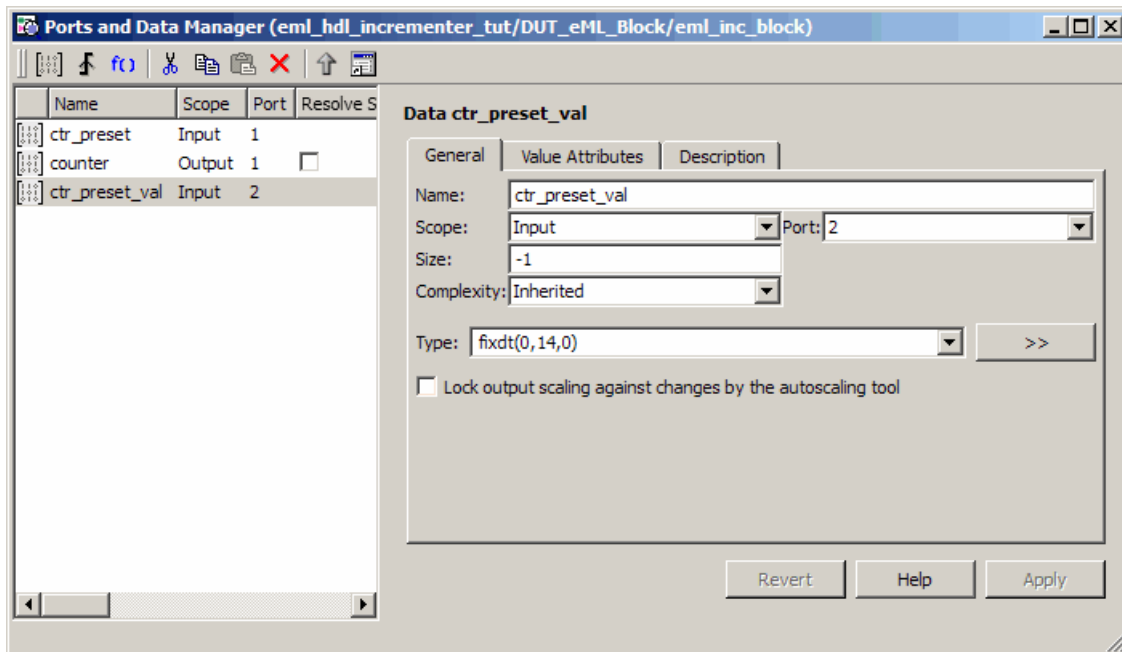


- 2 Double-click the incremter function block, to open the Embedded MATLAB Editor. In the editor window, select **Edit Data/Ports** from the **Tools** menu. The Ports and Data Manager dialog box opens.
- 3 Select the `ctr_preset` entry in the port list on the left. Click the button labeled `>>` to display the Data Type Assistant. Set the **Mode** property for

this port to **Built in**. Set the **Data type** property to **boolean**. Click the button labeled **<<** to close the Data Type Assistant. Click **Apply**.

- 4 Select the `ctr_preset_val` entry in the port list on the left. Click the button labeled **>>** to display the Data Type Assistant. Set the **Mode** property for this port to **Fixed point**. Set the **Signedness** property for this port to **Unsigned**. Set the **Word length** property to 14. Click the button labeled **<<** to close the Data Type Assistant. Click **Apply**.
- 5 Select the `counter` entry in the port list on the left. Click the button labeled **>>** to display the Data Type Assistant. Verify that the **Mode** property for this port is set to **Inherit: Same as Simulink**. Click the button labeled **<<** to close the Data Type Assistant. Click **Apply**.

The Ports and Data Manager dialog box should now appear as shown in the following figure.



- 6 Close the Ports and Data Manager dialog box and the editor.

- 7 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 mode** 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 mode** 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

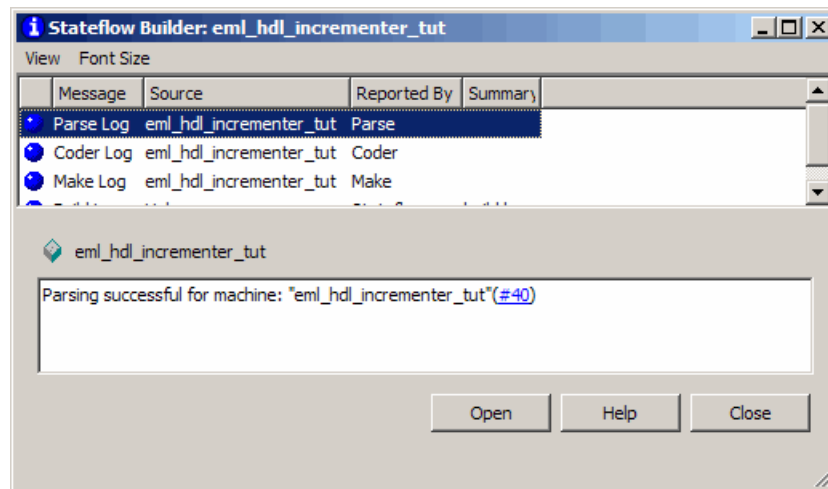
Use the built-in diagnostics of Embedded MATLAB Function blocks to test for syntax errors as follows:

- 1 If it is not already open, open the `eml_hdl_incrementer_tut` model.
- 2 Double-click the Embedded MATLAB Function block `incrementer` to open it for editing.
- 3 In the Embedded MATLAB Editor, select **Build** from the **Tools** menu (or press **Ctrl+B**) to compile and build the Embedded MATLAB Function block code.

The build process displays some progress messages. These messages will include some warnings, because the ports of the Embedded MATLAB Function block are not yet connected to any signals. You can ignore these warnings.

The build process builds a C-MEX 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 to HDL code generation.

When the build concludes successfully, a message window appears.

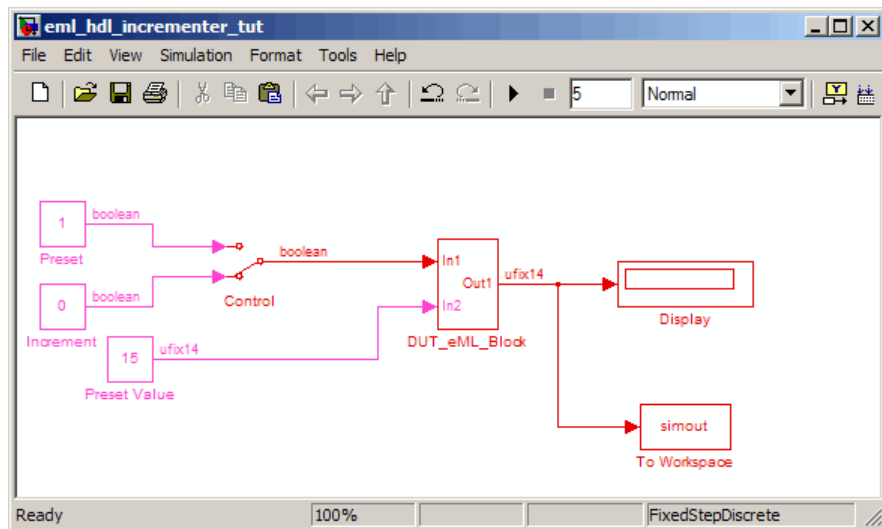


If errors are found, the Diagnostics Manager window lists them. See “Using the Embedded MATLAB Function Block” for information on debugging Embedded MATLAB Function block build errors.

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 that the model structure and settings are correct, and update the model display.

- 1 From the Simulink **Format** menu, select **Port/Signal Displays > Port Data Types**.
- 2 From the Simulink **Edit** menu, select **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. The model should now appear as shown in the following figure.



- 3 Save the model.

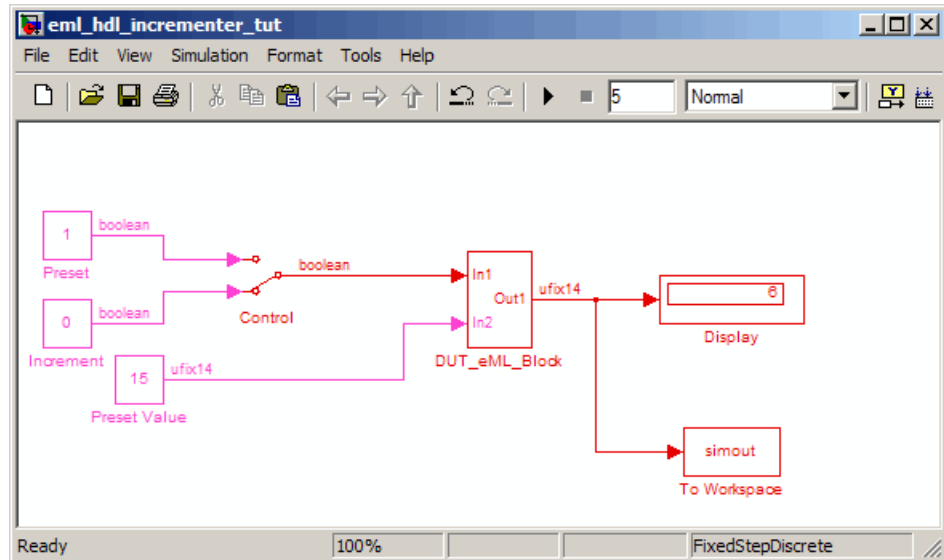
Simulating the eml_hdl_incrementer_tut Model

Click the **Start Simulation** icon to run a simulation.

If necessary, 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 presented at the ctr_preset port, the simulation returns a value of 6, as shown in the following figure.



You may want to experiment with the results of toggling the **Control** switch, changing the **Preset Value** constant, and changing the total simulation time. You may 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.

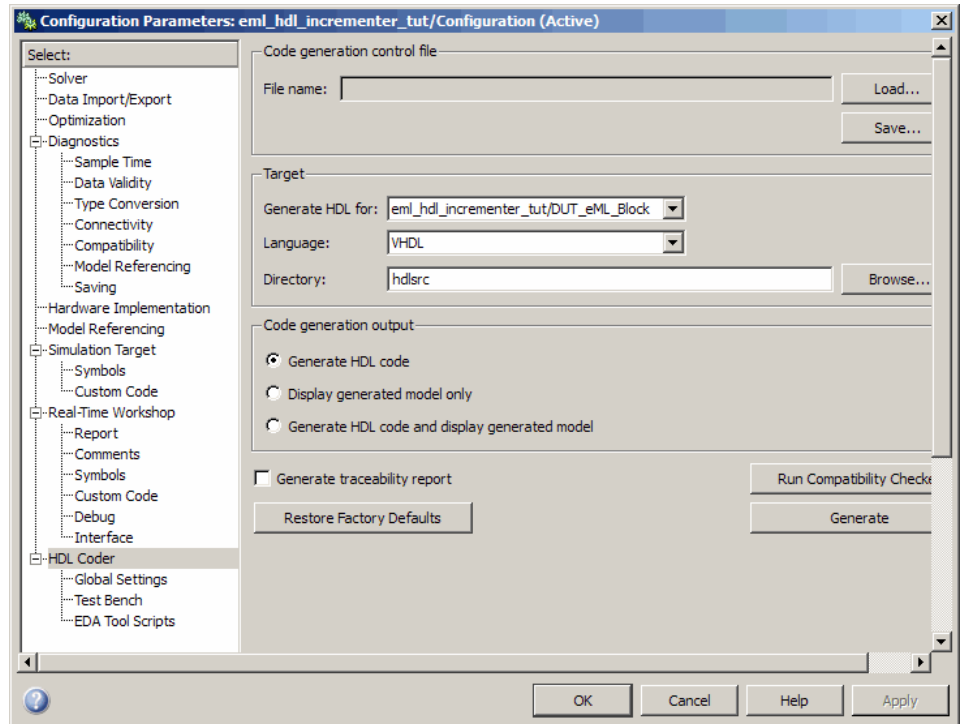
Selecting the Subsystem for Code Generation

Select the `DUT_eML_Block` subsystem for code generation, as follows:

- 1 Open the Configuration Parameters dialog box. Click the **HDL Coder** category in the **Select** tree in the left pane of the dialog box.

2 Select `eml_hdl_incrementer_tut/DUT_eML_Block` from the **Generate HDL for** list.

3 Click **Apply**. The dialog box should now appear as shown in the following figure.



Generating VHDL Code

The top-level **HDL Coder** 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 **Directory** field specifies (by default) that the code generation target directory is a subdirectory of your working directory, named `hdlsrc`.

Before generating code, select **Current Directory** from the **Desktop** menu in the MATLAB window. This displays the Current Directory browser, which lets you easily access your working directory 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, etc.), the appearance of the model may change after code generation.

- 2 As code generation proceeds, the coder displays progress messages. The process should complete successfully with the message

```
### 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 em1_hdl_incrementer_tut/DUT_eML_Block as hdlsrc\DUT_eML_Block.vhd
### Working on em1_hdl_incrementer_tut/DUT_eML_Block/em1_inc_block as hdlsrc\em1_inc_block.vhd
Embedded MATLAB parsing for model "em1_hdl_incrementer_tut"...Done
Embedded MATLAB code generation for model "em1_hdl_incrementer_tut"...Done
### HDL Code Generation Complete.
```

Observe that 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` directory is now visible in the Current Directory 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 Embedded 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` directory are:

- `eml_inc_blk.vhd`: VHDL code. This file contains entity and architecture code implementing the actual computations generated for the Embedded 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 `Embedded_MATLAB_Function.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 Embedded MATLAB Function block.

The other files that were generated in the `hdlsrc` directory 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 “Code Tracing Using the Mapping File” on page 9-25).
- 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 Directory browser.

At this point you should study the ENTITY and ARCHITECTURE definitions while referring to “HDL Code Generation Defaults” on page 17-23 in the `makehdl` reference documentation. The reference documentation describes the default naming conventions and correspondences between the elements of a model (subsystems, ports, signals, etc.) and elements of generated HDL code.

Useful Embedded MATLAB Function Block Design Patterns for HDL

In this section...

“The eml_hdl_design_patterns Library” on page 12-25

“Efficient Fixed-Point Algorithms” on page 12-27

“Using Persistent Variables to Model State” on page 12-31

“Creating Intellectual Property with the Embedded MATLAB Function Block” on page 12-32

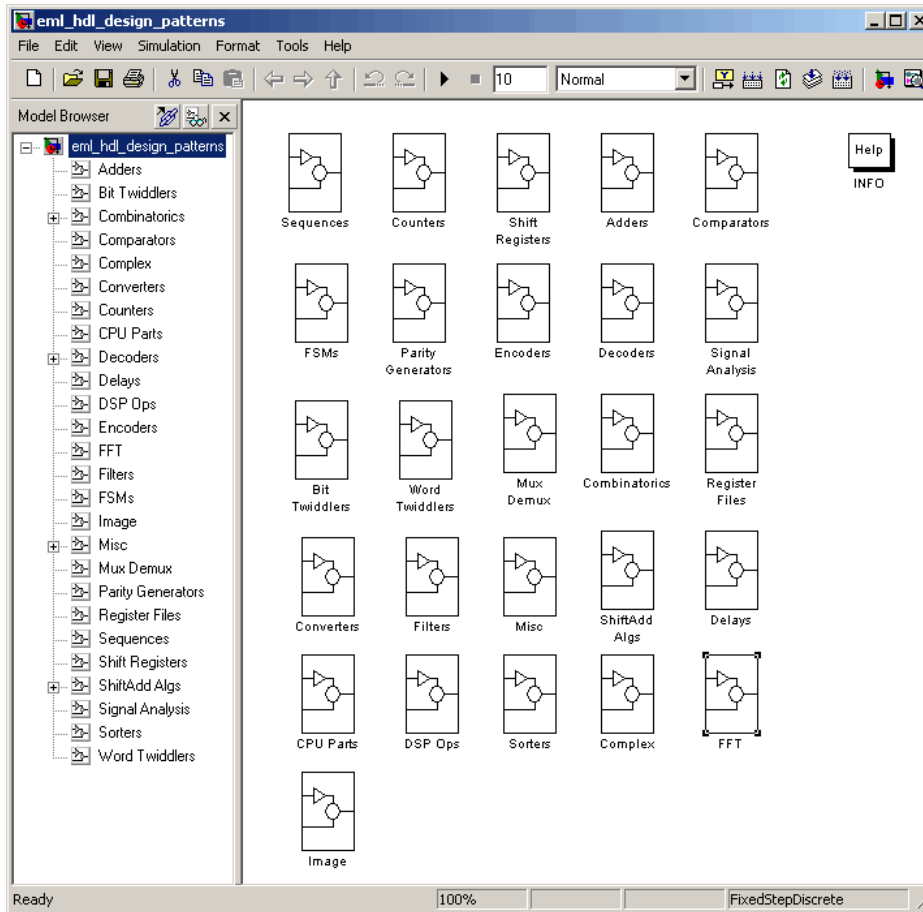
“Modeling Control Logic and Simple Finite State Machines” on page 12-33

“Modeling Counters” on page 12-35

“Modeling Hardware Elements” on page 12-36

The eml_hdl_design_patterns Library

The eml_hdl_design_patterns library is an extensive collection of examples demonstrating useful applications of the Embedded MATLAB Function block in HDL code generation. The following figure shows the library.



The location of the library in the MATLAB directory structure is

```
MATLABROOT\toolbox\hdlcoder\hdlcoderdemos\em1_hdl_design_patterns.mdl
```

Refer to example models in the `em1_hdl_design_patterns` library while reading the following sections. To open the library, type the following command at the MATLAB command 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, (generating code in a separate HDL file).
- Copy the code from the block and use it as a subfunction in an existing Embedded MATLAB Function block (generating inline HDL code).

Efficient Fixed-Point Algorithms

The Embedded MATLAB Function block supports fixed point arithmetic using the Fixed-Point Toolbox `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 Embedded MATLAB Function block is bit-true to MATLAB semantics. Generated code uses bit manipulation and bit access operators (e.g., Slice, Extend, Reduce, Concat, etc.) that are native to VHDL and Verilog.

The following discussion shows how HDL code generated from the Embedded 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 `Combinatrics/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 following listing shows the code embedded in the Embedded MATLAB Function block.

```
% 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 Embedded 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 “Recommended Practices” on page 12-68) 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
  --Embedded MATLAB Function 'Subsystem/eml_expr': '<S2>:1'
  -- fixpt arithmetic expression
  --'<S2>:1:4'
  mul_temp <= signed(a) * signed(b);
  sub_cast <= resize(mul_temp, 11);
  add_cast <= resize(signed(a & '0'), 7);
  add_cast_0 <= resize(signed(b), 7);
  add_temp <= add_cast + add_cast_0;
  sub_cast_0 <= resize(add_temp & '0' & '0', 11);

```

```

expr <= sub_cast - sub_cast_0;
-- cast the result to correct output type
-- '<S2>:1:7'

y <= "0111111" WHEN ((expr(10) = '0') AND (expr(9 DOWNT0 7) /= "000"))
    OR ((expr(10) = '0') AND (expr(7 DOWNT0 1) = "0111111"))
    ELSE
"1000000" WHEN (expr(10) = '1') AND (expr(9 DOWNT0 7) /= "111")
    ELSE
    std_logic_vector(expr(7 DOWNT0 1) + ("0" & expr(0)));

END fsm_SFHDL;

```

Verilog code excerpt:

```

//Embedded MATLAB Function 'Subsystem/eml_expr': '<S2>:1'
// fixpt arithmetic expression
// '<S2>:1:4'
assign mul_temp = a * b;
assign sub_cast = mul_temp;
assign add_cast = {a[4], {a, 1'b0}};
assign add_cast_0 = b;
assign add_temp = add_cast + add_cast_0;
assign sub_cast_0 = {{2{add_temp[6]}}, {add_temp, 2'b00}};
assign expr = sub_cast - sub_cast_0;
// cast the result to correct output type
// '<S2>:1:7'
assign y = (((expr[10] == 0) && (expr[9:7] != 0))
    || ((expr[10] == 0) && (expr[7:1] == 63)) ? 7'sb0111111 :
    ((expr[10] == 1) && (expr[9:7] != 7) ? 7'sb1000000 :
    expr[7:1] + $signed({1'b0, expr[0]}));

```

These code excerpts show that the generated HDL code from the Embedded 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.

Using Persistent Variables to Model State

To model sophisticated control logic, the ability to model registers is a basic requirement. In the Embedded 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. State-holding elements in hardware also require this behavior. Similarly, state-holding elements should retain their values across clock sample times. The values of persistent variables can also be changed using global and local reset conditions.

The subsystem `Delays` in the `eml_hdl_design_patterns` library illustrates how persistent variables can be used to simulate various kinds of delay blocks.

The `unit_delay` block delays the input sample by one simulation time step. A persistent variable is used to hold the value, as shown in the following code listing:

```
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;
```

In this example, `u` is a fixed-point operand of type `sfix6`. In the generated HDL code, initialization of persistent variables is moved into the master reset region in the initialization process as follows.

```
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 DOWNT0 0));
END Unit_Delay;

ARCHITECTURE fsm_SFHD OF Unit_Delay IS

BEGIN

    initialize_Unit_Delay : PROCESS (clk, reset)
    BEGIN
        IF reset = '1' THEN
            y <= std_logic_vector(to_signed(0, 16));
        ELSIF clk'EVENT AND clk = '1' THEN
            IF clk_enable = '1' THEN
                y <= u;
            END IF;
        END IF;
    END PROCESS initialize_Unit_Delay;
```

Refer to the `Delays` subsystem 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 invocations of the Embedded MATLAB Function block in a model.

Creating Intellectual Property with the Embedded MATLAB Function Block

The Embedded MATLAB Function block lets you quickly author intellectual property (IP). It also lets you rapidly create alternate implementations of a part of an algorithm.

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 (e.g., 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. This can be a powerful technique for tuning the generated hardware or customizing your algorithm.

By using Embedded MATLAB Function blocks in this way, you can guide the detailed operation of the HDL code generator even while writing high-level algorithms.

Modeling Control Logic and Simple Finite State Machines

Embedded 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 Embedded MATLAB Function block.

The following listing implements a Moore state machine.

```
function Z = moore_fsm(A)

persistent moore_state_reg;
if isempty(moore_state_reg)
    moore_state_reg = fi(0, 0, 2, 0);
end

S1 = 0;
S2 = 1;
S3 = 2;
S4 = 3;

switch uint8(moore_state_reg)

    case S1,
```

```
        Z = true;
        if (~A)
            moore_state_reg(1) = S1;
        else
            moore_state_reg(1) = S2;
        end
    case S2,
        Z = false;
        if (~A)
            moore_state_reg(1) = S1;
        else
            moore_state_reg(1) = S2;
        end
    case S3,
        Z = false;
        if (~A)
            moore_state_reg(1) = S2;
        else
            moore_state_reg(1) = S3;
        end
    case S4,
        Z = true;
        if (~A)
            moore_state_reg(1) = S1;
        else
            moore_state_reg(1) = S3;
        end
    otherwise,
        Z = false;
end
```

In this example, a persistent variable (`moore_state_reg`) models state variables. The output depends only on the state variables, thus modeling a Moore machine.

The FSMs/`mealy_fsm_blk` block in the `eml_hdl_design_patterns` library implements a Mealy state machine. A Mealy state machine differs from a Moore state machine in that the outputs depend on inputs as well as state variables.

The Embedded 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 Embedded MATLAB Function blocks intended for HDL code generation, there are some simple HDL related coding requirements:

- The top level Embedded MATLAB Function block must be called once per time step.
- It must be possible to fully unroll program loops.
- Persistent variables with proper 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 Embedded 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 Embedded 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;
        --Embedded MATLAB Function 'Subsystem/shift_reg_1by32': '<S2>:1'
        --shift register 1 by 32
        --'<S2>:1:1
        -- return sr[31]
        --'<S2>:1:10'
        sr_out <= sr(31);

        IF shift /= '0' THEN
            --'<S2>:1:12'
            -- sr_new[32:1] = sr[31:1] & sr_in
            --'<S2>:1:14'
            sr_next <= sr(30 DOWNTO 0) & sr_in;
        END IF;

    END PROCESS shift_reg_1by32;

```

The Shift Registers/`shift_reg_1by64` block shows a 64 bit shifter. In this case, the shifter uses two fixed point words, to represent the operand, overcoming the 32-bit word length limitation for fixed-point integers.

Browse the `eml_hdl_design_patterns` model for other useful hardware elements that can be easily implemented using the Embedded MATLAB Function Block.

Using Fixed-Point Bitwise Functions

In this section...

“Overview” on page 12-39

“Bitwise Functions Supported for HDL Code Generation” on page 12-39

“Bit Slice and Bit Concatenation Functions” on page 12-44

“Shift and Rotate Functions” on page 12-45

Overview

The Embedded MATLAB Function block supports many bitwise functions that operate on fixed-point integers of arbitrary length. For general information on Embedded MATLAB bitwise functions, see “Bitwise Operations” in the Fixed-Point Toolbox documentation.

This section describes HDL code generation support for these functions. “Bitwise Functions Supported for HDL Code Generation” on page 12-39 summarizes the supported functions, with notes that describe considerations specific to HDL code generation. “Bit Slice and Bit Concatenation Functions” on page 12-44 and “Shift and Rotate Functions” on page 12-45 provide usage examples, with corresponding Embedded MATLAB Function block code and generated HDL code.

The Bit Twiddlers/hdl_bit_ops block in the eml_hdl_design_patterns library provides further examples of how to use these functions for various bit manipulation operations.

Bitwise Functions Supported for HDL Code Generation

The following table summarizes Embedded MATLAB Function block 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.

Embedded MATLAB Function blocks follow the MATLAB (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. See “Bitwise Operations” in the Fixed-Point Toolbox documentation for information on the individual functions.

Embedded MATLAB Function Block 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 style="text-align: center;"><code>&a[lidx:ridx]</code></p>	<code>bitandreduce</code>
<code>bitcmp(a)</code>	Bitwise complement	<code>bitcmp</code>

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

Embedded MATLAB Function Block Syntax	Description	See Also
bitror(a, idx)	<p>Rotate right.</p> <p>idx must be a positive integer. The value of idx can be greater than the word length of a. idx is always normalized to $\text{mod}(\text{idx}, \text{wlen})$, where wlen is the word length of a.</p> <p>For VHDL, generates the ror operator.</p> <p>For Verilog, generates the following expression (where wl is the word length of a:</p> <pre style="text-align: center;">a >> idx a << wl - idx</pre>	bitror
bitset(a, idx, val)	<p>Set or clear bit(s) at position idx.</p> <p>If val = 0, clears the indicated bit(s). Otherwise, sets the indicated bits.</p>	bitset
bitshift(a, idx)	<p>Note: for efficient HDL code generation use, use bitsll, bitsrl, or bitsra <i>instead of</i> bitshift.</p> <p>Shift left or right, based on the positive or negative integer value of idx.</p> <p>idx must be an integer.</p> <p>For positive values of idx, shift left idx bits.</p> <p>For negative values of idx, shift right idx bits.</p> <p>If idx is a variable, generated code contains logic for both left shift and right shift.</p> <p>Result values saturate if the overflowMode of a is set to saturate.</p>	bitshift
bitsliceget(a, lidx, ridx)	<p>Access consecutive set of bits from lidx to ridx.</p> <p>Output data type: ufixN, where N = lidx-ridx+1.</p>	bitsliceget

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

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

Bit Slice and Bit Concatenation Functions

This section shows you how to use the Embedded MATLAB 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 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

```

Shift and Rotate Functions

The Embedded MATLAB Function block supports shift and rotate functions that mimic HDL-specific operators without saturation and rounding logic.

The following Embedded MATLAB 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 = bitlsl(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
```

Using Complex Signals

In this section...

“Introduction” on page 12-49

“Declaring Complex Signals” on page 12-49

“Conversion Between Complex and Real Signals” on page 12-51

“Arithmetic Operations on Complex Numbers” on page 12-51

“Support for Vectors of Complex Numbers” on page 12-55

“Other Operations on Complex Numbers” on page 12-56

Introduction

This section describes Embedded MATLAB Function block support for complex data types for HDL code generation. See also the `eml_hdl_design_patterns` library for numerous examples showing HDL related applications of complex arithmetic in Embedded MATLAB Function blocks.

Declaring Complex Signals

The following Embedded MATLAB Function block 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 Embedded MATLAB Function block 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 DOWNT0 0);
        x_im : OUT std_logic_vector(7 DOWNT0 0);
        y_re : OUT std_logic_vector(7 DOWNT0 0);
        y_im : OUT std_logic_vector(7 DOWNT0 0);
        z_re : OUT std_logic_vector(7 DOWNT0 0);
        z_im : OUT std_logic_vector(7 DOWNT0 0));
END complex_dec1;

ARCHITECTURE fsm_SFHDL OF complex_dec1 IS

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

As shown in the example, all complex inputs, outputs and local variables declared in Embedded 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 an Embedded MATLAB Function block remains complex during the entire length of the program, following Embedded MATLAB Function block language rules.

Conversion Between Complex and Real Signals

The Embedded MATLAB Function block 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 Embedded MATLAB Function block 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 for the Embedded MATLAB Function Block, the coder supports the following arithmetic operators for complex numbers composed of all base types (integer, fixed-point, double):

- Addition (+)
- Subtraction (-)

- Multiplication (*)

The coder supports division only for the Fixed-Point Toolbox `divide` function (see `divide` in the Fixed-Point Toolbox documentation). 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 = hdlfimath

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 necessary 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, numerictype(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 necessary 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, numerictype(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, numerictype(x), fimath(x));
END fsm_SFHDL;

```

Support for Vectors of Complex Numbers

Embedded MATLAB Function Block supports HDL code generation 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);
```

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

In both VHDL and Verilog code generated for the Embedded MATLAB Function Block, 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 Embedded MATLAB Function Block 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 Embedded_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 Embedded_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 always returns 0 for complex numbers, is particularly useful for writing Embedded MATLAB algorithms 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 “Supported Functions and Limitations of the Fixed-Point Embedded MATLAB Subset” in the Fixed-Point Toolbox documentation.

Distributed Pipeline Insertion

In this section...

“Overview” on page 12-58

“Example: Multiplier Chain” on page 12-59

“Limitations” on page 12-67

Overview

Distributed pipeline insertion is a special optimization for HDL code generated from Embedded 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.

The coder performs distributed pipeline insertion when you specify both of the following implementation parameters for Embedded MATLAB Function blocks or Stateflow charts in a control file:

- `{'OutputPipeline', nStages}`: the number of pipeline stages (`nStages`) must be greater than zero.
- `{'DistributedPipelining', 'on'}`: enables distributed pipeline insertion.

Under these conditions, the coder inserts pipeline stages in the generated code (whenever possible), rather than generating pipeline stages at the output of the HDL code. The `nStages` argument defines the number of pipeline stages to be inserted.

Retiming is recommended during RTL synthesis to effect further optimization, if possible.

In a small number of cases, the coder generates conventional output pipeline registers, even if `{'DistributedPipelining', 'on'}` is specified. See “Limitations” on page 12-67 for a description of these cases.

The default value for `DistributedPipelining` is `'off'`.

The `DistributedPipelining` property applies only to Embedded MATLAB Function blocks or Stateflow charts within a subsystem.

The following table summarizes the combined effect of the `DistributedPipelining` and `OutputPipeline` parameters.

DistributedPipelining	OutputPipeline, nStages	Result
'off' (default)	Unspecified (nStages defaults to 0)	No pipeline registers are inserted.
	nStages > 0	nStages output registers are introduced at the output of the block.
'on'	Unspecified (nStages defaults to 0)	No pipeline registers are inserted. <code>DistributedPipelining</code> has no effect.
	nStages > 0	nStages registers are introduced inside the block, based on critical path analysis.

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. Then set the **Ignore output data checking (number of samples)** option (or the `IgnoreDataChecking` property, if you are using the command-line interface) accordingly. For further information see:

- “Ignore output data checking (number of samples)” on page 3-71
- `IgnoreDataChecking`

Example: Multiplier Chain

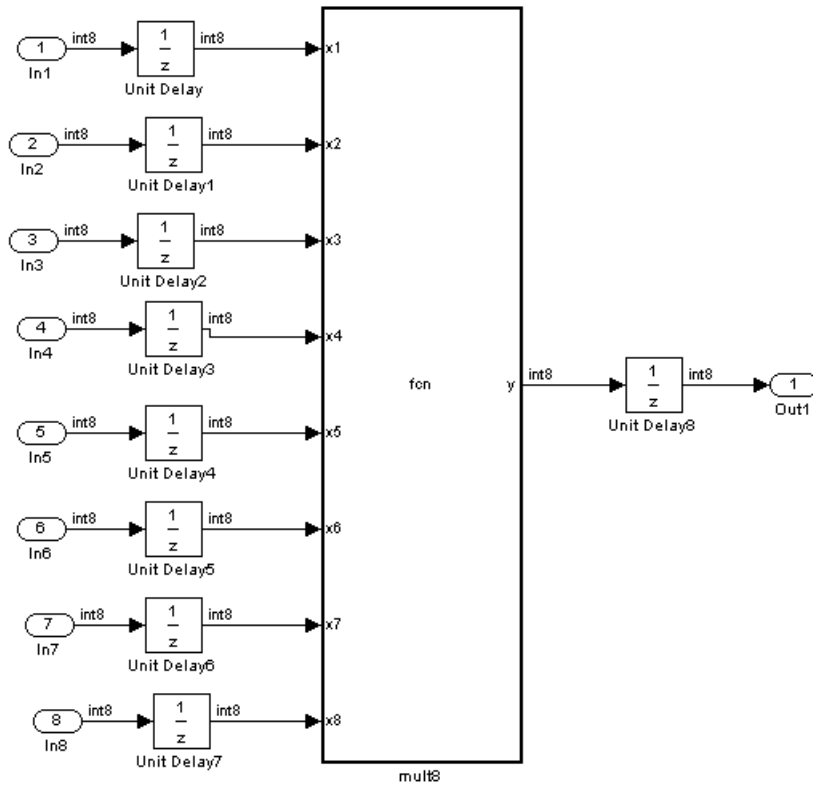
This section examines distributed pipeline insertion as applied to a simple model that implements a chain of 5 multiplications. If you are unfamiliar with control files and implementation parameters, see “Specifying Block

Implementations and Parameters in the Control File” on page 5-24 before studying this example.

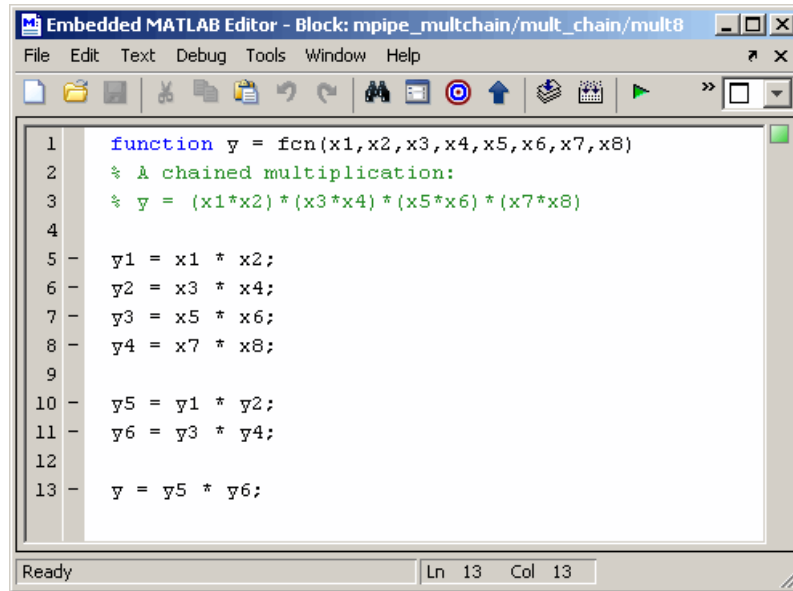
The example model and the associated control file are available in the demos directory as the following files:

```
MATLABROOT\toolbox\hdlcoder\hdlcoderdemos\mpipe_multchain.mdl  
MATLABROOT\toolbox\hdlcoder\hdlcoderdemos\pipeline_control.m
```

The root level model contains a subsystem `multi_chain`. The `multi_chain` subsystem functions as the device under test (DUT) from which HDL code is generated. The subsystem drives an Embedded MATLAB Function block, `mult8`. The following figure shows the subsystem.



The following figure shows a chain of multiplications as coded in the `mult8` Embedded MATLAB Function block.



The screenshot shows the Embedded MATLAB Editor window titled "Block: mpipe_multchain/mult_chain/mult8". The editor contains the following MATLAB code:

```

1  function y = fcn(x1,x2,x3,x4,x5,x6,x7,x8)
2  % & chained multiplication:
3  % y = (x1*x2)*(x3*x4)*(x5*x6)*(x7*x8)
4
5  - y1 = x1 * x2;
6  - y2 = x3 * x4;
7  - y3 = x5 * x6;
8  - y4 = x7 * x8;
9
10 - y5 = y1 * y2;
11 - y6 = y3 * y4;
12
13 - y = y5 * y6;

```

The status bar at the bottom indicates "Ready" and "Ln 13 Col 13".

To apply distributed pipeline insertion to this block, the control file `pipeline_control.m` must be invoked when HDL code is generated for the DUT. The control file specifies generation of two pipeline stages for the Embedded MATLAB Function block, and enables the distributed pipeline optimization, as shown in the following code listing:

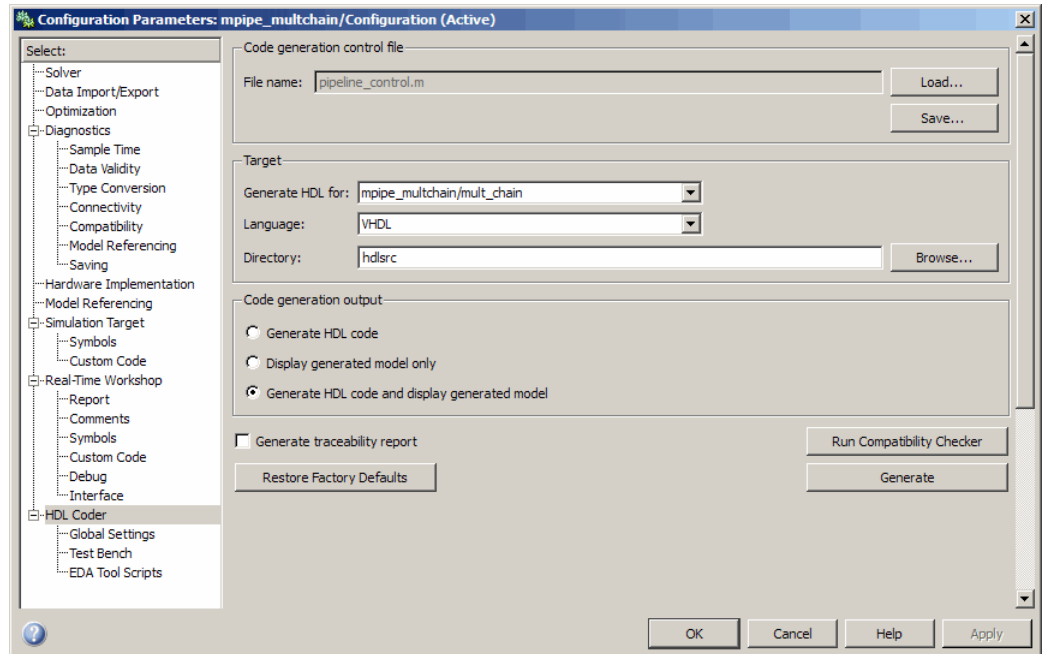
```

function c = pipeline_control
c = hdlnewcontrol(mfilename);
c.forEach('*',...
' eml_lib/Embedded MATLAB Function', {}, ...
'hdlstateflow.StateflowHDLInstantiation', ...
{'OutputPipeline', 2, 'DistributedPipelining', 'on'});

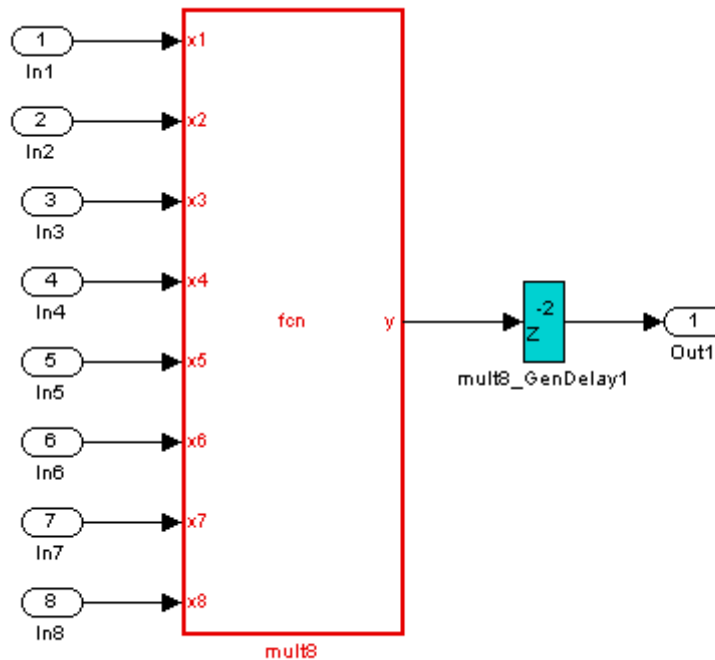
```

The following figure shows the top-level **HDL Coder** options for the model in the Configuration Parameters dialog box. The options are configured so that:

- The control file `pipeline_control.m` is attached to the model.
- VHDL code is generated from the subsystem `mpipe_multchain/mult`.
- 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` subsystem. This ensures that simulations of the model accurately reflect the behavior of the generated HDL code. The following figure shows the inserted Integer Delay block.



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

The following limitations apply to distributed pipeline insertion:

- If the Embedded MATLAB Function block code or Stateflow chart contains any matrix with a statically unresolvable index, the coder generates pipeline registers at the output(s).
- In the current release, if the Embedded MATLAB Function block code defines any persistent variables the coder generates pipeline registers at the output(s).
- In the current release, if a Stateflow chart contains any state or local variable, the coder generates pipeline registers at the output(s).
- The latencies of the operations currently chosen are approximate. Therefore, pipelining results may not be optimal in cases where the relative operation latencies in the target platform do not match the trend of the chosen latencies.

Recommended Practices

In this section...
“Introduction” on page 12-68
“Use Compiled External M-Functions on the Embedded MATLAB Path” on page 12-68
“Build the Embedded MATLAB Code First” on page 12-68
“Use the hdlfimath Utility for Optimized FIMATH Settings” on page 12-69
“Use Optimal Fixed-Point Option Settings” on page 12-70

Introduction

This section describes recommended practices when using the Embedded MATLAB Function block for HDL code generation.

By setting Embedded MATLAB Function block options as described in this section, you can significantly increase the efficiency of generated HDL code. See “Setting Optimal Fixed-Point Options for the Embedded MATLAB Function Block” on page 12-10 for an example.

Use Compiled External M-Functions on the Embedded MATLAB Path

The coder supports HDL code generation from Embedded MATLAB Function blocks that include compiled external M-functions. This feature lets you write reusable M-code and call it from multiple Embedded MATLAB Function blocks.

Such functions must be defined in M-files that are on the Embedded MATLAB path, and must include the `##eml` compilation directive. See “Adding the Compilation Directive `##eml`” in the Embedded MATLAB documentation for information on how to create, compile, and invoke external M-functions.

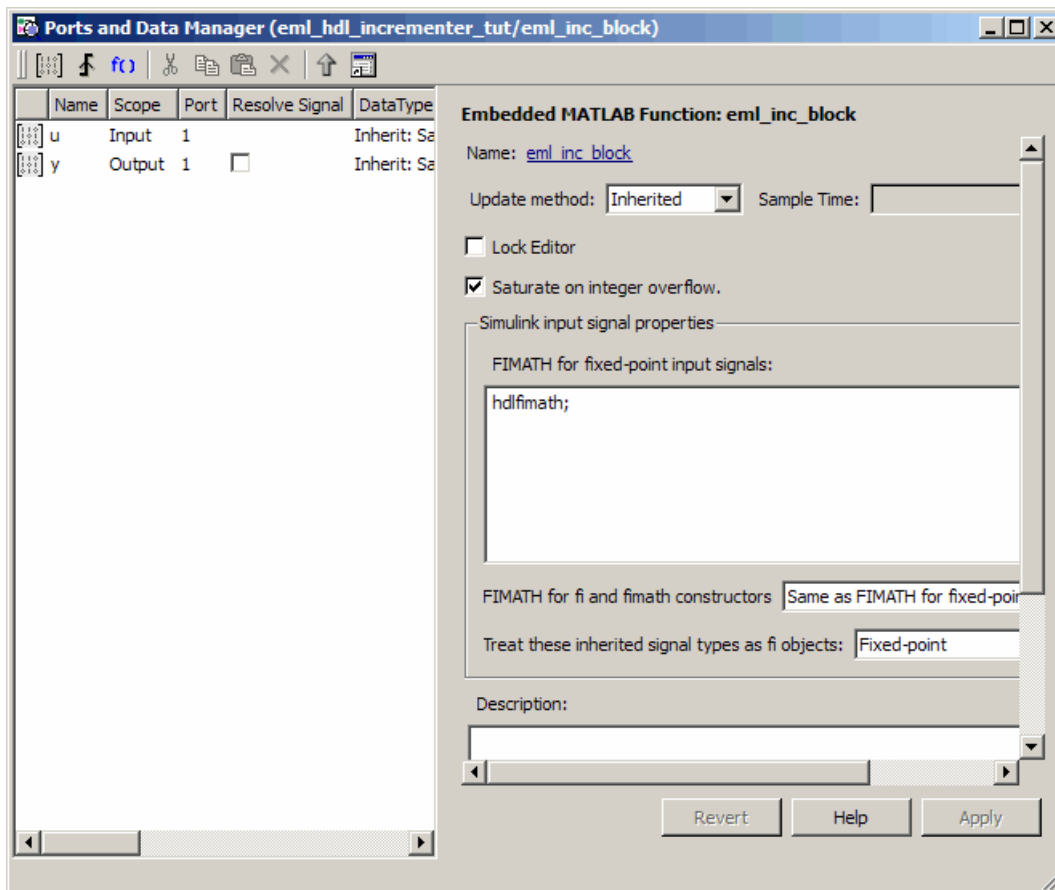
Build the Embedded MATLAB Code First

Before generating HDL code for a subsystem containing an Embedded MATLAB Function block, it is strongly recommended that you build the

Embedded MATLAB code to check for errors. To build the code, select **Build** from the **Tools** menu in the Embedded MATLAB Function block editor (or press **CTRL+B**).

Use the `hdlfimath` Utility for Optimized FIMATH Settings

The M-function `hdlfimath.m` is a utility that defines a FIMATH specification that is optimized for HDL code generation. It is strongly recommended that you replace the default **FIMATH for fixed-point signals** specification with a call to the `hdlfimath` function, as shown in the following figure.



The following listing shows the FIMATH setting defined by `hdlfimath`.

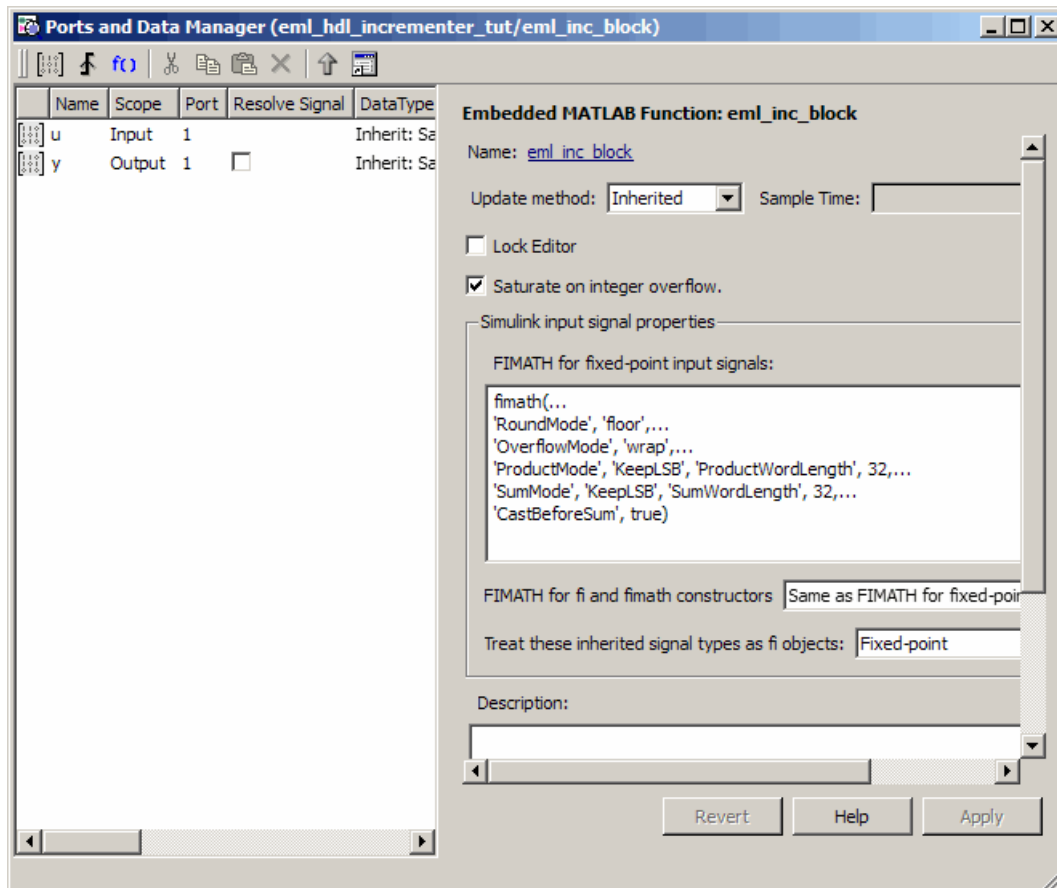
```
hdlfm = fimath(...  
    'RoundMode', 'floor',...  
    'OverflowMode', 'wrap',...  
    'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...  
    'SumMode', 'FullPrecision', 'SumWordLength', 32,...  
    'CastBeforeSum', true);
```

Note 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.



Language Support

In this section...
“Fixed-Point Runtime Library Support” on page 12-72
“Variables and Constants” on page 12-73
“Use of Nontunable Parameter Arguments” on page 12-77
“Arithmetic Operators” on page 12-77
“Relational Operators” on page 12-78
“Logical Operators” on page 12-79
“Control Flow Statements” on page 12-79

Fixed-Point Runtime Library Support

The coder supports most of the fixed-point runtime library functions supported by the Embedded MATLAB Function block. For a complete list of these functions, see “Supported Functions and Limitations of the Fixed-Point Embedded MATLAB Subset” in the Fixed-Point Toolbox documentation.

Some functions are not supported, or are subject to some restrictions. These functions are summarized in the following table.

Function	Restriction	Notes
disp	Not supported	
get	Not supported	This function returns a struct. Struct data types are not supported in this release.
pow2	Not supported	
real	Not supported	
divide	Supported, with restrictions	The divisor must be a constant and a power of two.

Function	Restriction	Notes
subsasgn	Supported, with restrictions	Subscripted assignment supported; see “Data Type Usage” on page 12-73
subsref	Supported, with restrictions	Subscripted reference supported; see “Data Type Usage” on page 12-73

Variables and Constants

This section summarizes supported data types and typing rules for variable and constants, and the use of persistent variables in modeling registers.

Data Type Usage

When generating code for the Embedded MATLAB Function block, the coder supports a subset of MATLAB data types. The following table summarizes supported and unsupported data types.

Type(s)	Support	Notes
Integer	Supported: <ul style="list-style-type: none"> • uint8, uint16, uint32, • int8, int16, int32 	
Real	Supported: <ul style="list-style-type: none"> • double • single 	HDL code generated with <code>double</code> or <code>single</code> data types is not synthesizable.
Character	Supported: <ul style="list-style-type: none"> • char 	
Logical	Supported: <ul style="list-style-type: none"> • Boolean 	

Type(s)	Support	Notes
Fixed point	Supported: <ul style="list-style-type: none"> • Scaled (binary point only) fixed point numbers • Custom integers (zero binary point) 	Fixed point numbers with slope (not equal to 1.0) and bias (not equal to 0.0) are not supported. Maximum word size for fixed-point numbers is 32 bits.
Vectors	Supported: <ul style="list-style-type: none"> • unordered {N} • row {1, N} • column {N, 1} 	The maximum number of vector elements allowed is 2^{32} . A variable must be fully defined before it is subscripted.
Matrix	N/A	Matrix data types are not supported in the current release.
Struct	N/A	Struct data types are not supported in the current release.
Cell arrays	N/A	Cell arrays are not supported in the current release.

Typing Ports, Variables and Constants

Strong typing rules are applied to Embedded MATLAB Function blocks, as follows:

- All input and output port data types must be resolved at model compilation time.
 - If the data type of an input port is unspecified when the model is compiled, the port is assigned the data type of the signal driving the port.
 - If the data type of an output port is unspecified when the model is compiled, the output port type is determined by the first assignment to the output variable.
- Similarly, all constant literals are strongly typed. If you do not specify the data type of a constant explicitly, its type is determined by internal

rules. To specify the data type of a constant, use cast functions (e.g., `uint8`, `uint16`, etc.) or `fi` functions using `fimath` specifications.

- After you have defined a variable, do not change its data type. Variable types cannot be changed dynamically by assigning a different value. Dynamic typing will lead to a compile time error.
- After you have defined a variable, do not change its size. Variables cannot be grown or resized dynamically.
- Do not use output variables to model registered output; Embedded MATLAB Function block outputs are never persistent. Use persistent variables for this purpose, as described in “Persistent Variables” on page 12-75.

Persistent Variables

Persistent variables let you model registers. If you need to preserve state between invocations of an Embedded MATLAB Function block, use persistent variables.

Each persistent variable must be initialized with a statement specifying its size and type before it is referenced. You can initialize a persistent variable with either a constant value or a variable, as in the following code listings:

```
% 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
```

When testing whether a persistent variable has been initialized, it is good practice to use simple logical expressions, as in the preceding examples. Using

simple expressions ensures that the HDL code for the test is generated in the reset process, and therefore is executed only once.

You can initialize multiple variables based on a single simple 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
```

See also “The Incrementer Function Code” on page 12-6 for an example of the initialization and use of a persistent variable.

Note If persistent variables are not initialized properly, unnecessary sentinel variables can appear in the generated code.

Limitation on Use of Persistent Variables. As described in “Using Persistent Variables to Model State” on page 12-31, you can use persistent variables in Embedded MATLAB code to simulate various kinds of delay blocks.

However, note that the ports on the Embedded MATLAB Function block act as direct feedthrough ports during simulation. The delay constructs internal to the Embedded MATLAB Function block are not recognized during simulation. Therefore a feedback loop in the model causes an algebraic loop condition.

To work around this limitation:

- Keep the combinatorial logic inside the Embedded MATLAB Function block for one of the blocks in the loop which has a persistent variable for the output or input. Remove the persistent variable.

- Place a Unit Delay block external to the Embedded MATLAB Function block.

Use of Nontunable Parameter Arguments

An Embedded MATLAB function argument can be declared to be a *parameter argument* by setting its **Scope** to **Parameter** in the Ports and Data Manager GUI. Such a parameter argument does not appear as a signal port on the block. Parameter arguments for Embedded MATLAB Function blocks do not take their values from signals in the Simulink model. Instead, their values come from parameters defined in a parent Simulink masked subsystem or variables defined in the MATLAB base workspace.

Only *nontunable* parameters are supported for HDL code generation. If you declare parameter arguments in Embedded MATLAB function code that is intended for HDL code generation, be sure to clear the **Tunable** option for each such parameter argument.

See also “Parameter Arguments in Embedded MATLAB Functions” in the Simulink documentation.

Arithmetic Operators

When generating code for the Embedded MATLAB Function block, the coder supports the arithmetic operators (and their M-function equivalents) listed in the following table.

Operation	Operator Syntax	M-Function Equivalent	Fixed Point Support?
Binary addition	A+B	plus(A,B)	Y
Matrix multiplication	A*B	mtimes(A,B)	Y
Arraywise multiplication	A.*B	times(A,B)	Y
Matrix right division	A/B	mrdivide(A,B)	Y
Arraywise right division	A./B	rdivide(A,B)	Y
Matrix left division	A\B	mldivide(A,B)	Y
Arraywise left division	A.\B	ldivide(A,B)	Y

Operation	Operator Syntax	M-Function Equivalent	Fixed Point Support?
Matrix power	A^B	mpower(A,B)	Y
Arraywise power	A.^B	power(A,B)	Y
Complex transpose	A'	ctranspose(A)	Y
Matrix transpose	A.'	transpose(A)	Y
Matrix concat	[A B]	None	Y
Matrix index Note: A variable must be fully defined before it is subscripted.	A(r c)	None	Y

Relational Operators

When generating code for the Embedded MATLAB Function block, the coder supports the relational operators (and their M-function equivalents) listed in the following table.

Relation	Operator Syntax	M Function Equivalent	Fixed-Point Support?
Less than	A<B	lt(A,B)	Y
Less than or equal to	A<=B	le(A,B)	Y
Greater than or equal to	A>=B	ge(A,B)	Y
Greater than	A>B	gt(A,B)	Y
Equal	A==B	eq(A,B)	Y
Not equal	A~=B	ne(A,B)	Y

Logical Operators

When generating code for the Embedded MATLAB Function block, the coder supports the logical operators (and their M function equivalents) listed in the following table.

Relation	Operator Syntax	M Function Equivalent	Fixed-Point Support?	Notes
Logical And	A&B	and(A,B)	Y	
Logical Or	A B	or(A,B)	Y	
Logical Xor	A xor B	xor(A,B)	Y	
Logical And (short circuiting)	A&&B	N/A	Y	Use short circuiting logical operators within conditionals. See also “Control Flow Statements” on page 12-79.
Logical Or (short circuiting)	A B	N/A	Y	Use short circuiting logical operators within conditionals. See also “Control Flow Statements” on page 12-79.
Element complement	~A	not(A)	Y	

Control Flow Statements

When generating code for the Embedded MATLAB Function block, the coder imposes some restrictions on the use of control flow statements and constructs. The following table summarizes supported and unsupported control flow statements.

Control Flow Statement	Notes
break continue return	Do not use these statements within loops. Use of these statements in a loop causes the coder to report the following error: Unstructured flow graph or loop containing [statement type] not supported for HDL

Control Flow Statement	Notes
while	<p>while loops are not supported. Use of while loops causes the coder to report the following error:</p> <p style="padding-left: 40px;">Unstructured flow graph or loop containing [statement type] not supported for HDL</p>
for	<p>for loops without static bounds are not supported. Use of for loops without static bounds causes the coder to report the following error:</p> <p style="padding-left: 40px;">Unstructured flow graph or loop containing [statement type] not supported for HDL</p> <p>Do not use the & and operators within conditions of a for statement. Instead, use the && and operators.</p> <p>The Embedded MATLAB Function block does not support nonscalar expressions in the conditions of for statements. 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>The Embedded MATLAB Function block does not support nonscalar expressions in the conditions of if statements. Use the all or any functions to collapse logical vectors into scalars.</p>
switch	<p>The HDL code matches the behavior of the switch statement; the first matching case statement is executed.</p> <p>Use only scalars in conditional expressions in a switch statement.</p> <p>Use of fi variables in switch or case conditionals is not supported. For HDL code generation, the usage is restricted to uint8, uint16, uint32, sint8, sint16, and sint32.</p> <p>If multiple case statements make assignments to the same variable, then their numeric type and fimath specification should match that variable.</p>

Other Limitations

This section lists other limitations that apply when generating HDL code with the Embedded MATLAB Function block. These limitations are:

- The HDL compatibility checker (checkhdl) performs only a basic compatibility check on the Embedded MATLAB Function block. HDL related warnings or errors may arise during code generation from an Embedded MATLAB Function block that is otherwise valid for simulation. Such errors are reported in a separate message window.
- The Embedded MATLAB subset does not support nested functions. Subfunctions are supported, however. For an example, see “Tutorial Example: Incrementer” on page 12-4.
- Use of multiple values on the left side of an expression is not supported. For example, an error results from the following assignment statement:

```
[t1, t2, t3] = [1, 2, 3];
```


Generating Scripts for HDL Simulators and Synthesis Tools

- “Overview of Script Generation for EDA Tools” on page 13-2
- “Defaults for Script Generation” on page 13-3
- “Custom Script Generation” on page 13-4

Overview of Script Generation for EDA Tools

The coder supports generation of script files for third-party electronic design automation (EDA) tools. These scripts let you compile and simulate generated HDL code or synthesize generated HDL code.

Using the defaults, you can automatically generate scripts for the following tools:

- Mentor Graphics ModelSim simulator
- Synplify family of synthesis tools

Defaults for Script Generation

By default, script generation takes place automatically, as part of the code and test bench generation process.

All script files are generated in the target directory.

When you generate HDL code for a model or subsystem *system*, the coder writes the following script files:

- *system_compile.do*: Mentor Graphics ModelSim compilation script. This script contains commands to compile the generated code, but not to simulate it.
- *system_synplify.tcl*: Synplify synthesis script

When you generate test bench code for a model or subsystem *system*, the coder writes the following script files:

- *system_tb_compile.do*: Mentor Graphics ModelSim compilation script. This script contains commands to compile the generated code and test bench.
- *system_tb_sim.do*: Mentor Graphics ModelSim simulation script. This script contains commands to run a simulation of the generated code and test bench.

Custom Script Generation

In this section...

“Overview” on page 13-4

“Structure of Generated Script Files” on page 13-4

“Properties for Controlling Script Generation” on page 13-5

“Controlling Script Generation with the EDA Tool Scripts GUI Pane” on page 13-8

Overview

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 the appropriate property name/property value arguments, as described in “Properties for Controlling Script Generation” on page 13-5.
- Set script generation options in the **EDA Tool Scripts** pane of the Simulink GUI, as described in “Controlling Script Generation with the EDA Tool Scripts GUI Pane” on page 13-8.

Structure of Generated Script Files

A generated EDA script consists of three sections, generated and executed in the following order:

- 1** An initialization (`Init`) phase. The `Init` phase performs any 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 takes no 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 any legal `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'

Script Type	Property	Default Value
Simulation	'HDLSimFilePostfix'	'_sim.do'
Synthesis	'HDLSynthFilePostfix'	'_synplify.tcl'

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 work\n'	Format string passed to <code>fprintf</code> to write the <code>Init</code> section of the compilation script.
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).

Property Name and Default	Description
Name: 'HDLCompileVerilogCmd' Default: 'vlog %s %s\n'	Format string passed to <code>fprintf</code> to write the Cmd section of the compilation script for Verilog files. The two 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.
Name: 'HDLsimInit' Default: ['onbreak resume\n',... 'onerror resume\n']	Format string passed to <code>fprintf</code> to write the initialization section of the simulation script.
Name: 'HDLsimCmd' Default: 'vsim -novopt work.%s\n'	Format string passed to <code>fprintf</code> 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 <code>fprintf</code> to write the simulation script waveform viewing command. The implicit argument is the top-level module or entity name.
Name: 'HDLsimTerm' Default: 'run -all\n'	Format string passed to <code>fprintf</code> to write the Term portion of the simulation script
Name: 'HDLSynthInit' Default: 'project -new %s.prj\n'	Format string passed to <code>fprintf</code> 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.

Property Name and Default	Description
Name: 'HDLSynthCmd' Default: 'add_file %s\n'	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.
Name: 'HDLSynthTerm' Default: <pre data-bbox="185 539 635 690"> ['set_option -technology VIRTEX4\n',... 'set_option -part XC4VSX35\n',... 'set_option -synthesis_onoff_pragma 0\n',... 'set_option -frequency auto\n',... 'project -run synthesis\n'] </pre>	Format string passed to fprintf to write the Term section of the synthesis script.

Example

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
```

Controlling Script Generation with the EDA Tool Scripts GUI Pane

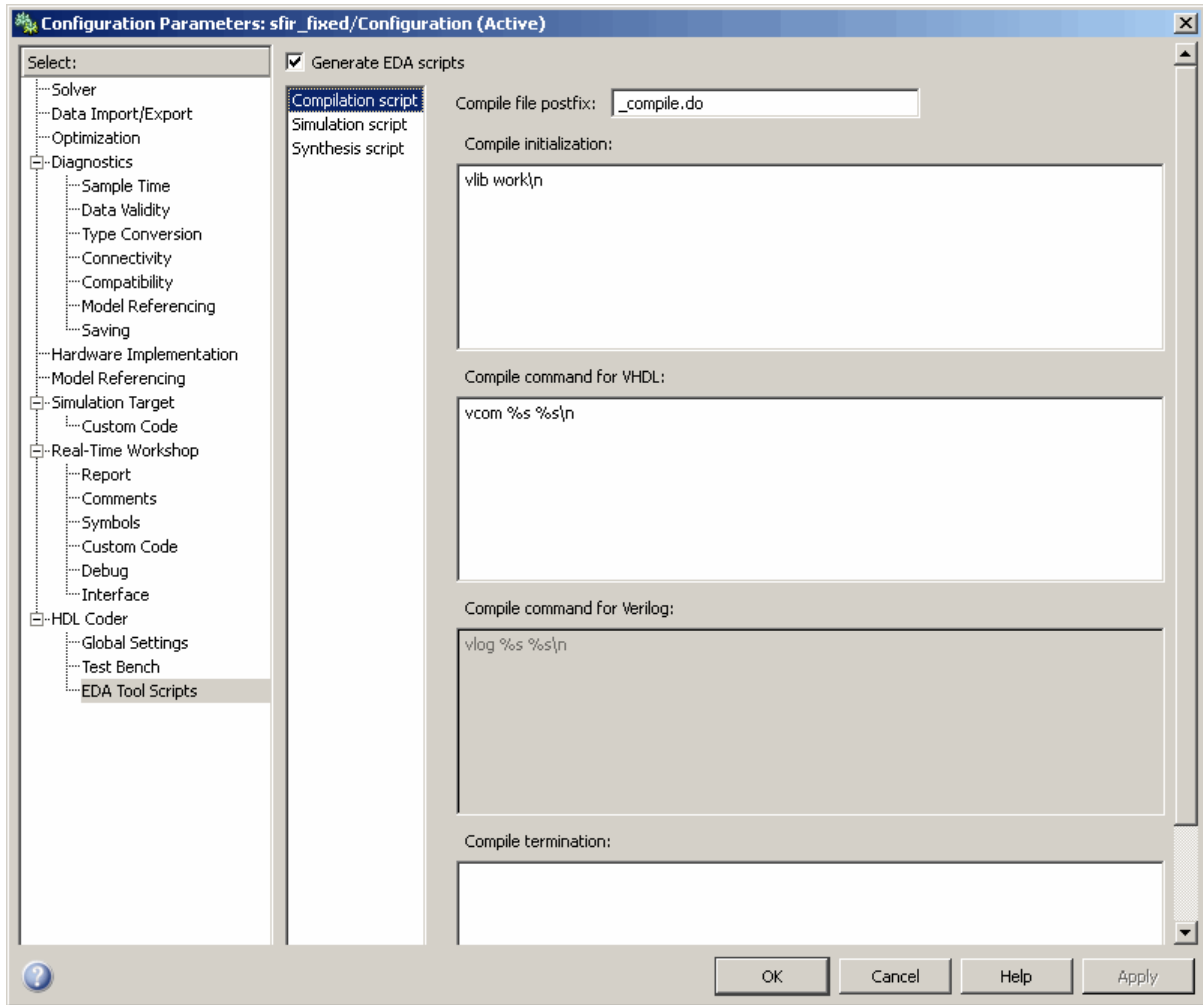
The **EDA Tool Scripts** pane of the GUI lets you set all options that control generation of script files. These options correspond to the properties described in “Properties for Controlling Script Generation” on page 13-5

To view and set options in the **EDA Tool Scripts** GUI pane:

- 1** Select **Configuration Parameters** from the **Simulation** menu in the model window.

The Configuration Parameters dialog box opens with the **Solver** options pane displayed.

- 2** Click the **EDA Tool Scripts** entry in the **Select** tree in the left pane of the Configuration Parameters dialog box. By default, the **EDA Tool Scripts** pane is displayed, with the **Compilation script** options group selected, as shown in the following figure.



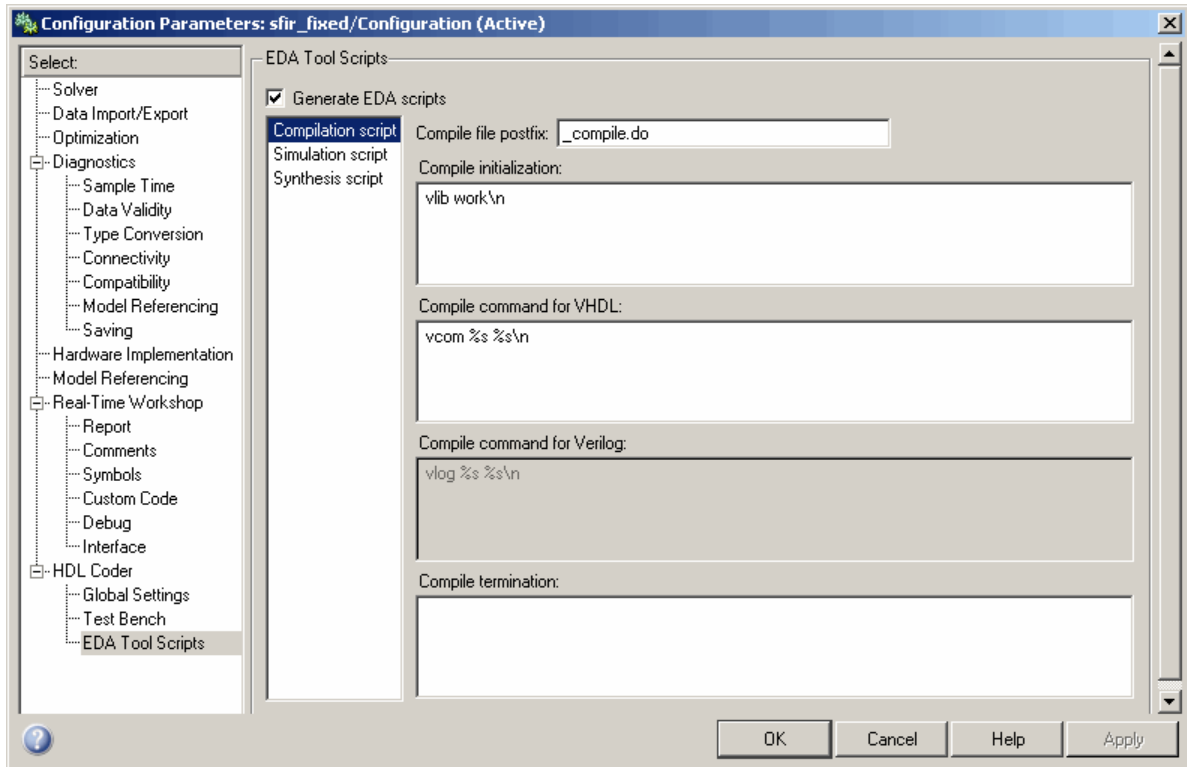
3 The **Generate EDA scripts** option controls the generation of script files. By default, this option is selected.

If you want to disable script generation, deselect this option 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 13-11 for further information.
 - **Simulation script:** Options related to customizing scripts for HDL simulators. See “Simulation Script Options” on page 13-13 for further information.
 - **Synthesis script:** Options related to customizing scripts for synthesis tools. See “Synthesis Script Options” on page 13-15 for further information.

Compilation Script Options

The following figure shows the **Compilation script** pane, with all options set to their default values.



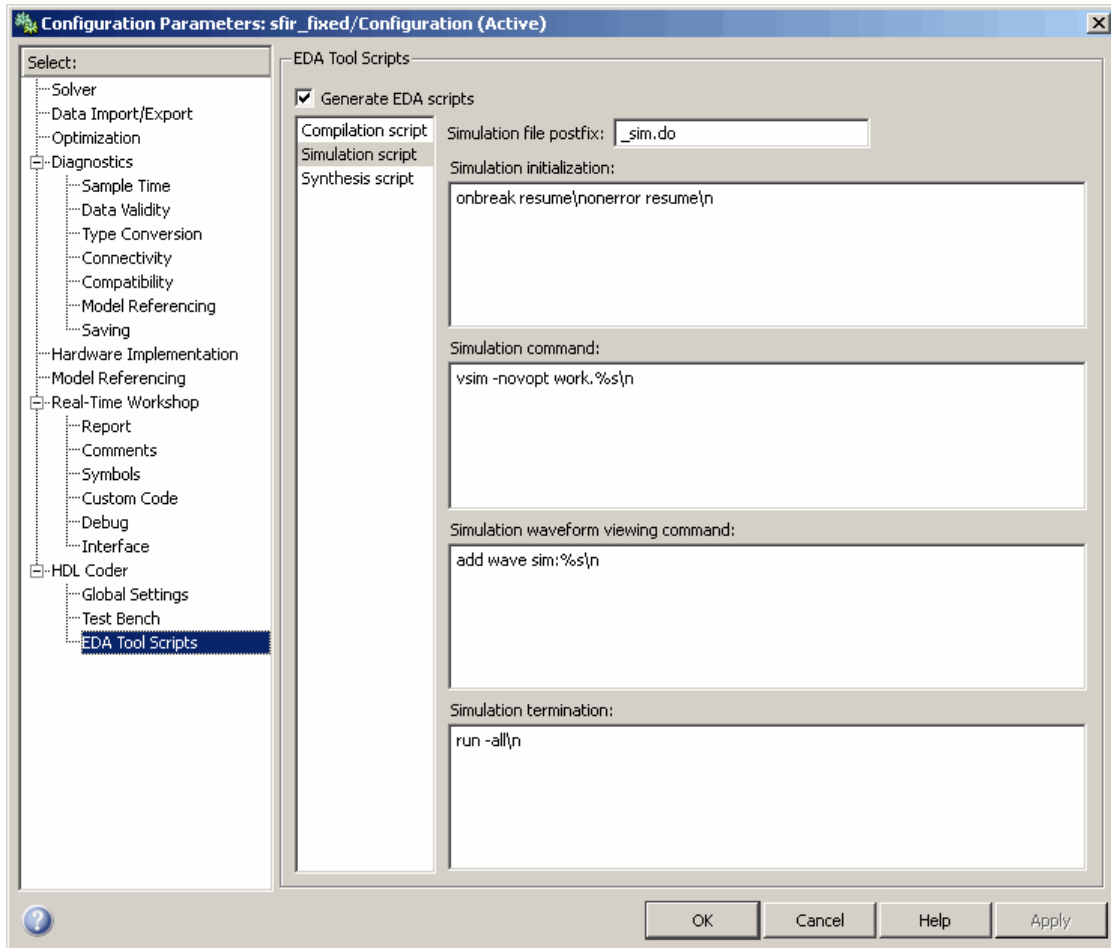
The following table summarizes the **Compilation script** options.

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 work\n'	Format string passed to fprintf to write the Init section of the compilation script.

Option and Default	Description
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 all options set to their default values.



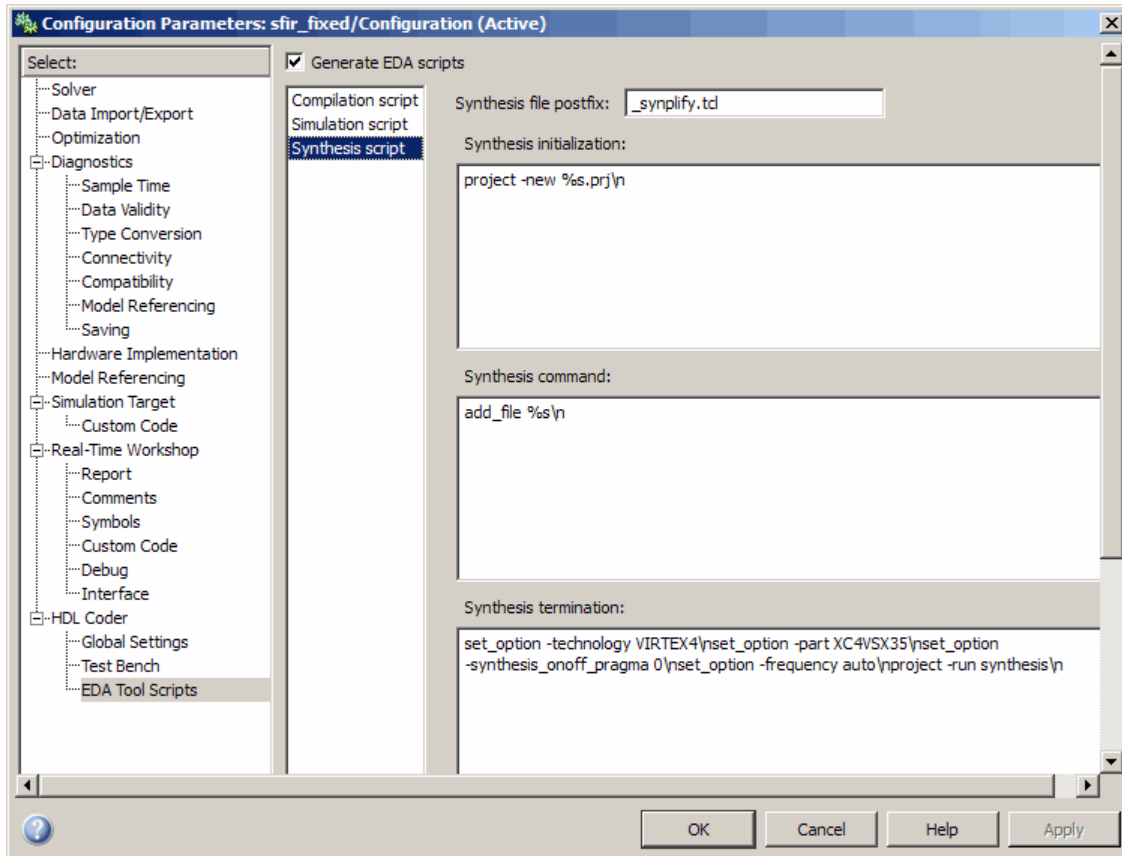
The following table summarizes the **Simulation script** options.

Option and Default	Description
Simulation file postfix '_sim.do'	Postfix string appended to the model name or test bench name to form the simulation script file name.

Option and Default	Description
Simulation initialization Default: <code>['onbreak resume\nonerror resume\n']</code>	Format string passed to <code>fprintf</code> to write the initialization section of the simulation script.
Simulation command Default: <code>'vsim -novopt work.%s\n'</code>	Format string passed to <code>fprintf</code> to write the simulation command. The implicit argument is the top-level module or entity name.
Simulation waveform viewing command Default: <code>'add wave sim:%s\n'</code>	Format string passed to <code>fprintf</code> to write the simulation script waveform viewing command. The top-level module or entity signal names are implicit arguments.
Simulation termination Default: <code>'run -all\n'</code>	Format string passed to <code>fprintf</code> to write the Term portion of the simulation script.

Synthesis Script Options

The following figure shows the **Synthesis script** pane, with all options set to their default values.



The following table summarizes the **Synthesis script** options.

Option Name and Default	Description
Name: Synthesis initialization Default: 'project -new %s.prj\n'	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.

Option Name and Default	Description
Name: Synthesis command Default: 'add_file %s\n'	Format string passed to fprintf to write the Cmd section of the synthesis script. The argument is the filename of the entity or module.
Name: Synthesis termination Default: <pre data-bbox="185 527 630 677">'set_option -technology VIRTEX4\n',... 'set_option -part XC4VSX35\n',... 'set_option -synthesis_onoff_pragma 0\n',... 'set_option -frequency auto\n',... 'project -run synthesis\n']</pre>	Format string passed to fprintf to write the Term section of the synthesis script.

Property Reference

Language Selection Properties (p. 14-2)	Properties for selecting language of generated HDL code
File Naming and Location Properties (p. 14-2)	Properties that name and specify location of generated files
Reset Properties (p. 14-2)	Properties that specify reset signals in generated code
Header Comment and General Naming Properties (p. 14-3)	Properties affecting generation of header comments and process, module, component instance, and other name strings
Script Generation Properties (p. 14-4)	Properties affecting generation of script files for simulation and synthesis tools
Port Properties (p. 14-5)	Properties that specify port characteristics in generated code
Advanced Coding Properties (p. 14-6)	Advanced HDL coding properties
Test Bench Properties (p. 14-7)	Properties that specify generated test bench code
Generated Model Properties (p. 14-9)	Properties for controlling naming and appearance of generated models

Language Selection Properties

TargetLanguage	Specify HDL language to use for generated code
----------------	------------------------------------------------

File Naming and Location Properties

HDLMapPostfix	Specify postfix string appended to file name for generated mapping file
TargetDirectory	Identify directory into which generated output files are written
VerilogFileExtension	Specify file type extension for generated Verilog files
VHDLFileExtension	Specify file type extension for generated VHDL files

Reset Properties

ResetAssertedLevel	Specify asserted (active) level of reset input signal
ResetLength	Define length of time (in clock cycles) during which reset is asserted
ResetType	Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers
ResetValue	Specify constant value to which test bench forces reset input signals

Header Comment and General Naming Properties

<code>ClockProcessPostfix</code>	Specify string to append to HDL clock process names
<code>ComplexImagPostfix</code>	Specify string to append to imaginary part of complex signal names
<code>ComplexRealPostfix</code>	Specify string to append to real part of complex signal names
<code>EntityConflictPostfix</code>	Specify string to append to duplicate VHDL entity or Verilog module names
<code>InstancePrefix</code>	Specify string prefixed to generated component instance names
<code>PackagePostfix</code>	Specify string to append to specified model or subsystem name to form name of package file
<code>ReservedWordPostfix</code>	Specify string appended to identifiers for entities, signals, constants, or other model elements that conflict with VHDL or Verilog reserved words
<code>SplitArchFilePostfix</code>	Specify string to append to specified name to form name of file containing model's VHDL architecture
<code>SplitEntityArch</code>	Specify whether generated VHDL entity and architecture code is written to single VHDL file or to separate files
<code>SplitEntityFilePostfix</code>	Specify string to append to specified model name to form name of generated VHDL entity file
<code>VectorPrefix</code>	Specify string prefixed to vector names in generated code

Script Generation Properties

EDAScriptGeneration	Enable or disable generation of script files for third-party tools
HDLCompileFilePostfix	Specify postfix string appended to file name for generated Mentor Graphics ModelSim compilation scripts
HDLCompileInit	Specify string written to initialization section of compilation script
HDLCompileTerm	Specify string written to termination section of compilation script
HDLCompileVerilogCmd	Specify command string written to compilation script for Verilog files
HDLCompileVHDLCmd	Specify command string written to compilation script for VHDL files
HDLSimCmd	Specify simulation command written to simulation script
HDLSimFilePostfix	Specify postfix string appended to file name for generated Mentor Graphics ModelSim simulation scripts
HDLSimInit	Specify string written to initialization section of simulation script
HDLSimTerm	Specify string written to termination section of simulation script
HDLSimViewWaveCmd	Specify waveform viewing command written to simulation script
HDLSynthCmd	Specify command written to synthesis script

HDLSynthFilePostfix	Specify postfix string appended to file name for generated Synplify synthesis scripts
HDLSynthInit	Specify string written to initialization section of synthesis script
HDLSynthTerm	Specify string written to termination section of synthesis script

Port Properties

ClockEnableInputPort	Name HDL port for model's clock enable input signals
ClockEnableOutputPort	Specify name of clock enable output port
ClockInputPort	Name HDL port for model's clock input signals
EnablePrefix	Specify base name string for internal clock enables in generated code
InputType	Specify HDL data type for model's input ports
OutputType	Specify HDL data type for model's output ports
ResetInputPort	Name HDL port for model's reset input

Advanced Coding Properties

<code>BlockGenerateLabel</code>	Specify string to append to block labels used for HDL <code>GENERATE</code> statements
<code>CastBeforeSum</code>	Enable or disable type casting of input values for addition and subtraction operations before execution of operation
<code>CheckHDL</code>	Check model or subsystem for HDL code generation compatibility
<code>HDLControlFiles</code>	Attach code generation control file to model
<code>HoldInputDataBetweenSamples</code>	Specify how long substrate signal values are held in valid state
<code>InlineConfigurations</code>	Specify whether generated VHDL code includes inline configurations
<code>InstanceGenerateLabel</code>	Specify string to append to instance section labels in VHDL <code>GENERATE</code> statements
<code>LoopUnrolling</code>	Specify whether VHDL <code>FOR</code> and <code>GENERATE</code> loops are unrolled and omitted from generated VHDL code
<code>OptimizeTimingController</code>	Optimize timing controller entity for speed and code size by implementing separate counters per rate
<code>OutputGenerateLabel</code>	Specify string that labels output assignment block for VHDL <code>GENERATE</code> statements
<code>PipelinePostfix</code>	Specify string to append to names of input or output pipeline registers generated for pipelined block implementations

SafeZeroConcat	Specify syntax for concatenated zeros in generated VHDL code
Traceability	Enable or disable creation of HTML code generation report with code-to-model and model-to-code hyperlinks
UseAggregatesForConst	Specify whether all constants are represented by aggregates, including constants that are less than 32 bits
UserComment	Specify comment line in header of generated HDL and test bench files
UseRisingEdge	Specify VHDL coding style used to check for rising edges when operating on registers
UseVerilogTimescale	Use compiler <code>`timescale</code> directives in generated Verilog code
Verbosity	Specify level of detail for messages displayed during code generation

Test Bench Properties

ClockHighTime	Specify period, in nanoseconds, during which test bench drives clock input signals high (1)
ClockLowTime	Specify period, in nanoseconds, during which test bench drives clock input signals low (0)
ForceClock	Specify whether test bench forces clock input signals
ForceClockEnable	Specify whether test bench forces clock enable input signals

<code>ForceReset</code>	Specify whether test bench forces reset input signals
<code>GenerateCoSimBlock</code>	Generate model containing HDL Cosimulation block(s) for use in testing DUT
<code>HoldTime</code>	Specify hold time for input signals and forced reset input signals
<code>IgnoreDataChecking</code>	Specify number of samples during which output data checking is suppressed
<code>InitializeTestBenchInputs</code>	Specify initial value driven on test bench inputs before data is asserted to DUT
<code>MultifileTestBench</code>	Divide generated test bench into helper functions, data, and HDL test bench code files
<code>SimulatorFlags</code>	Specify simulator flags to apply to generated compilation scripts
<code>TestBenchClockEnableDelay</code>	Define elapsed time (in clock cycles) between deassertion of reset and assertion of clock enable
<code>TestBenchDataPostFix</code>	Specify suffix added to test bench data file name when generating multi-file test bench
<code>TestBenchPostFix</code>	Specify suffix to test bench name
<code>TestBenchReferencePostFix</code>	Specify string appended to names of reference signals generated in test bench code

Generated Model Properties

<code>CodeGenerationOutput</code>	Control production of generated code and display of generated model
<code>GeneratedmodelName</code>	Specify name of generated model
<code>Generatedmodelnameprefix</code>	Specify prefix to name of generated model
<code>Highlightancestors</code>	Highlight ancestors of blocks in generated model that differ from original model
<code>Highlightcolor</code>	Specify color for highlighted blocks in generated model

Properties — Alphabetical List

BlockGenerateLabel

Purpose	Specify string to append to block labels used for HDL GENERATE statements
Settings	'string' Default: '_gen' Specify a postfix string to append to block labels used for HDL GENERATE statements.
See Also	InstanceGenerateLabel, OutputGenerateLabel

Purpose	Enable or disable type casting of input values for addition and subtraction operations before execution of operation
Settings	'on' (default) Typecast input values in addition and subtraction operations to the result type before operating on the values. 'off' Preserve the types of input values during addition and subtraction operations and then convert the result to the result type.
See Also	InlineConfigurations, LoopUnrolling, SafeZeroConcat, UseAggregatesForConst, UseRisingEdge, UseVerilogTimescale

CheckHDL

Purpose	Check model or subsystem for HDL code generation compatibility
Settings	<p>'on'</p> <p>Check the model or subsystem for HDL compatibility before generating code, and report any problems encountered. This is equivalent to executing the <code>checkhdl</code> function before calling <code>makehdl</code>.</p> <p>'off' (default)</p> <p>Do not check the model or subsystem for HDL compatibility before generating code.</p>
See Also	<code>checkhdl</code> , <code>makehdl</code>

Purpose

Name HDL port for model's clock enable input signals

Settings

'string'

Default: 'clk_enable'

The string specifies the name for the model's clock enable input port.

If you override the default with (for example) the string 'filter_clock_enable' for the generating subsystem `filter_subsys`, the generated entity declaration might look as follows:

```
ENTITY filter_subsys IS
    PORT( clk          : IN  std_logic;
          filter_clock_enable : IN  std_logic;
          reset        : IN  std_logic;
          filter_subsys_in  : IN  std_logic_vector (15 DOWNTO 0);
          filter_subsys_out : OUT std_logic_vector (15 DOWNTO 0);
    );
END filter_subsys;
```

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`. See `ReservedWordPostfix` for more information.

Usage Notes

The clock enable signal is asserted active high (1). Thus, the input value must be high for the generated entity's registers to be updated.

See Also

`ClockInputPort`, `InputType`, `OutputType`, `ResetInputPort`

ClockEnableOutputPort

Purpose Specify name of clock enable output port

Settings 'string'

Default: 'ce_out'

The string specifies the name for the generated clock enable output port.

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

Purpose	Specify period, in nanoseconds, during which test bench drives clock input signals high (1)
Settings	<p>ns</p> <p>Default: 5</p> <p>The clock high time is expressed as a positive integer or double (with a maximum of 6 significant digits after the decimal point).</p> <p>The <code>ClockHighTime</code> and <code>ClockLowTime</code> 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.</p>
Usage Notes	The coder ignores this property if <code>ForceClock</code> is set to 'off'.
See Also	<code>ClockLowTime</code> , <code>ForceClock</code> , <code>ForceClockEnable</code> , <code>ForceReset</code> , <code>HoldTime</code>

ClockInputPort

Purpose Name HDL port for model's clock input signals

Settings 'string'

Default: 'clk'.

The string specifies the clock input port name.

If you override the default with (for example) the string 'filter_clock' for the generated entity `my_filter`, the generated entity declaration might look as follows:

```
ENTITY my_filter IS
    PORT( filter_clock : IN std_logic;
          clk_enable   : IN std_logic;
          reset        : IN std_logic;
          my_filter_in : IN std_logic_vector (15 DOWNT0 0); -- sfix16_En15
          my_filter_out: OUT std_logic_vector (15 DOWNT0 0); -- sfix16_En15
    );
END my_filter;
```

If you specify a 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`. See `ReservedWordPostfix` for more information.

See Also `ClockEnableInputPort`, `InputType`, `OutputType`

Purpose

Specify period, in nanoseconds, during which test bench drives clock input signals low (0)

Settings

Default: 5

The clock low time is expressed as a positive integer or double (with a maximum of 6 significant digits after the decimal point).

The `ClockHighTime` and `ClockLowTime` 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.

**Usage
Notes**

The coder ignores this property if `ForceClock` is set to 'off'.

See Also

`ClockHighTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`

ClockProcessPostfix

Purpose Specify string to append to HDL clock process names

Settings 'string'

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` in the following block declaration from the register name `delay_pipeline` and the default postfix string `_process`:

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
    .
    .
    .
```

See Also `PackagePostfix`, `ReservedWordPostfix`

Purpose	Control production of generated code and display of generated model
Settings	<code>'string'</code> Default: <code>'GenerateHDLCode'</code> Generate code but do not display the generated model. <code>'GenerateHDLCodeAndDisplayGeneratedModel'</code> Generate both code and model, and display model when completed. <code>'DisplayGeneratedModelOnly'</code> Create and display generated model, but do not proceed to code generation.
See Also	“Defaults and Options for Generated Models” on page 8-12

ComplexImagPostfix

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

Settings 'string'
Default: '_im'.

See Also ComplexRealPostfix

Purpose	Specify string to append to real part of complex signal names
Settings	'string' Default: 're'.
See Also	ComplexImagPostfix

EDAScriptGeneration

Purpose	Enable or disable generation of script files for third-party tools
Settings	'on' (default) Enable generation of script files. 'off' Disable generation of script files.
See Also	Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose	Specify base name string for internal clock enables in generated code
Settings	'string' Default: 'enb' Specify the string used as the base name for internal clock enables and other flow control signals in generated code.
Usage Notes	<p>Where only a single clock enable is generated, <code>EnablePrefix</code> specifies the signal name for the internal clock enable signal.</p> <p>In some cases multiple clock enables are generated (for example, when a cascade block implementation for certain blocks is specified). In such cases, <code>EnablePrefix</code> specifies a base signal name for the first clock enable that is generated. For other clock enable signals, numeric tags are appended to <code>EnablePrefix</code> to form unique signal names. For example, the following code fragment illustrates two clock enables that were generated when <code>EnablePrefix</code> was set to 'test_clk_enable' :</p>

```
COMPONENT Timing_Controller
  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;
```

EntityConflictPostfix

Purpose Specify string to append to duplicate VHDL entity or Verilog module names

Settings 'string'
Default: 'block'
The specified postfix resolves duplicate VHDL entity or Verilog module names.
For example, if the coder detects two entities with the name MyFilt, the coder names the first entity MyFilt and the second instance MyFilt_block.

See Also PackagePostfix, ReservedWordPostfix

Purpose

Specify whether test bench forces clock input signals

Settings

'on' (default)

Specify that the test bench forces the clock input signals. When this option is set, the clock high and low time settings control the clock waveform.

'off'

Specify that a user-defined external source forces the clock input signals.

See Also

ClockLowTime, ClockHighTime, ForceClockEnable, ForceReset, HoldTime

ForceClockEnable

Purpose	Specify whether test bench forces clock enable input signals
Settings	'on' (default) Specify that 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' Specify that a user-defined external source forces the clock enable input signals.
See Also	ClockHighTime, ClockLowTime, ForceClock, HoldTime

Purpose

Specify whether test bench forces reset input signals

Settings

'on' (default)

Specify that the test bench forces the reset input signals. If you enable this option, you can also specify a hold time to control the timing of a reset.

'off'

Specify that a user-defined external source forces the reset input signals.

See Also

ClockHighTime, ClockLowTime, ForceClock, HoldTime

GenerateCoSimBlock

Purpose

Generate model containing HDL Cosimulation block(s) for use in testing DUT

Settings

'on'

If your installation is licensed for one or more of the following HDL simulation products, the coder generates and opens a model that contains an HDL Cosimulation block for each licensed product:

- EDA Simulator Link MQ
- EDA Simulator Link IN
- EDA Simulator Link DS

The generated HDL Cosimulation blocks are configured 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.

The coder appends the string (if any) specified by the `CosimLibPostfix` property to the names of the generated HDL Cosimulation blocks.

'off' (default)

Do not generate HDL Cosimulation blocks.

Purpose Specify name of generated model

Settings 'string'

By default, the name of a generated model is the same as that of the original model. Assign a string value to `GeneratedmodelName` to override the default.

See Also “Defaults and Options for Generated Models” on page 8-12

Generatedmodelnameprefix

Purpose Specify prefix to name of generated model

Settings 'string'

Default: 'gm_'

The specified string is prepended to the sanme of the generated model.

See Also “Defaults and Options for Generated Models” on page 8-12

Purpose	Specify string written to initialization section of compilation script
Settings	'string' Default: 'vlib work\n'.
See Also	Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

HDLCompileTerm

Purpose	Specify string written to termination section of compilation script
Settings	'string' The default is the null string ('').
See Also	Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose

Specify postfix string appended to file name for generated Mentor Graphics ModelSim compilation scripts

Settings

'string'

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.

HDLCompileVerilogCmd

Purpose Specify command string written to compilation script for Verilog files

Settings 'string'

Default: 'vlog %s %s\n'.

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).

See Also Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose Specify command string written to compilation script for VHDL files

Settings 'string'

Default: 'vcom %s %s\n'.

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).

See Also Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

HDLControlFiles

Purpose

Attach code generation control file to model

Settings

{'string'}

Pass in a cell array containing a string that specifies a control file to be attached to the current model. Defaults are

- File name extension: `.m`
- Location of file: the control file must be on the MATLAB path or in the current working directory. Therefore you need only specify the file name; do not specify path information.

The following example specifies a control file, using the default for the file name extension.

```
makehdl('HDLControlFiles', {'dct8config'});
```

Specify a control file that is on the MATLAB path, or in the current working directory. If necessary, you should modify the MATLAB path such that the desired control file is on the path before generating code. Then attach the control file to the model.

Note The current release supports specification of a single control file.

Usage Notes

To clear the property (so that no control file is invoked during code generation), pass in a cell array containing the null string, as in the following example:

```
makehdl(gcb, 'HDLControlFiles', {''});
```

See Also

For a detailed description of the structure and use of control files, see Chapter 5, “Code Generation Control Files”.

Purpose Specify postfix string appended to file name for generated mapping file

Settings 'string'

Default: '_map.txt'.

For example, if the name of the device under test is `my_design`, the coder adds the postfix `_map.txt` to form the name `my_design_map.txt`.

HDLsimCmd

Purpose Specify simulation command written to simulation script

Settings 'string'
Default: 'vsim -novopt work.%s\n'.
The implicit argument is the top-level module or entity name.

See Also Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose	Specify string written to initialization section of simulation script
Settings	'string' The default string is <pre>['onbreak resume\n',... 'onerror resume\n']</pre>
See Also	Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

HDLsimFilePostfix

Purpose Specify postfix string appended to file name for generated Mentor Graphics ModelSim simulation scripts

Settings 'string'
Default: `_sim.do`.
For example, if the name of your test bench file is `my_design`, the coder adds the postfix `_sim.do` to form the name `my_design_tb_sim.do`.

Purpose	Specify string written to termination section of simulation script
Settings	'string' Default: 'run -all\n'.
See Also	Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

HDLSimViewWaveCmd

Purpose Specify waveform viewing command written to simulation script

Settings 'string'

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

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

See Also Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose	Specify command written to synthesis script
Settings	'string' Default: 'add_file %s\n'. The implicit argument is the file name of the entity or module.
See Also	Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

HDLSynthInit

Purpose	Specify string written to initialization section of synthesis script
Settings	'string' Default: 'project -new %s.prj\n', which is a synthesis project creation command. The implicit argument is the top-level module or entity name.
See Also	Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose Specify postfix string appended to file name for generated Synplify synthesis scripts

Settings 'string'

Default: `_synplify.tcl`.

For example, if the name of the device under test is `my_design`, the coder adds the postfix `_synplify.tcl` to form the name `my_design_synplify.tcl`.

HDLSynthTerm

Purpose Specify string written to termination section of synthesis script

Settings 'string'

The default string is

```
['set_option -technology VIRTEX4\n',...  
'set_option -part XC4VSX35\n',...  
'set_option -synthesis_onoff_pragma 0\n',...  
'set_option -frequency auto\n',...  
'project -run synthesis\n']
```

See Also Chapter 13, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose	Highlight ancestors of blocks in generated model that differ from original model
Settings	'on' (default) Highlight blocks in a generated model that differ from the original model, and their ancestor (parent) blocks in the model hierarchy. 'off' Highlight only the blocks in a generated model that differ from the original model without highlighting their ancestor (parent) blocks in the model hierarchy.
See Also	“Defaults and Options for Generated Models” on page 8-12

Highlightcolor

Purpose Specify color for highlighted blocks in generated model

Settings 'string'
Default: 'cyan'.
Specify the color as one of the following color string values:

- 'cyan'
- 'yellow'
- 'magenta'
- 'red'
- 'green'
- 'blue'
- 'white'
- 'black'

See Also “Defaults and Options for Generated Models” on page 8-12

HoldInputDataBetweenSamples

Purpose

Specify how long subrate signal values are held in valid state

Settings

'on' (default)

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.

Usage Notes

In most cases, the default ('on') is the correct setting for this property. This setting matches the behavior of a Simulink simulation, in which subrate signals are always held valid through each base-rate clock period.

In some cases (for example modeling memory or memory interfaces), it is desirable to set `HoldInputDataBetweenSamples` to 'off'. In this way you can obtain diagnostic information about when data is in an invalid ('X') state.

See Also

`HoldTime`, Chapter 4, “Generating HDL Code for Multirate Models”

HoldTime

Purpose

Specify hold time for input signals and forced reset input signals

Settings

ns

Default: 2

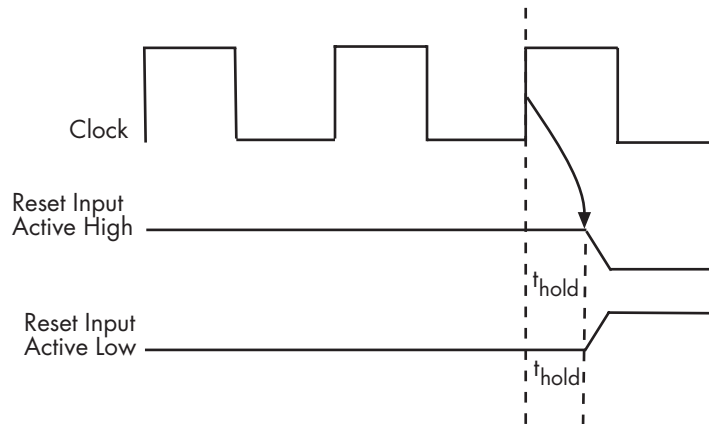
Specify the number of nanoseconds during which the model's data input signals and forced reset input signals are held past the clock rising edge.

The hold time is expressed as a positive integer or double (with a maximum of 6 significant digits after the decimal point).

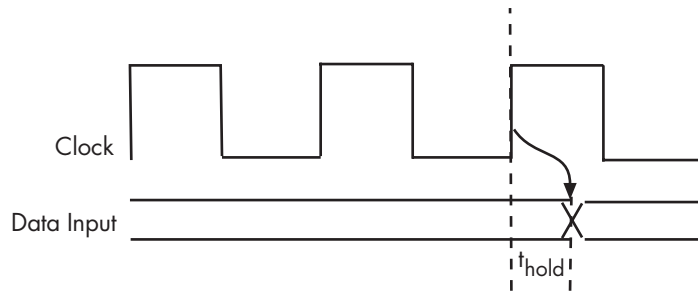
This option applies to reset input signals only if forced resets are enabled.

Usage Notes

The hold time is the amount of time that reset input signals and input data are held past the clock rising edge. The following figures show the application of a hold time (t_{hold}) for reset and data input signals when the signals are forced to active high and active low.



Hold Time for Reset Input Signals



Hold Time for Data Input Signals

Note A reset signal is always asserted for two cycles plus t_{hold} .

See Also

ClockHighTime, ClockLowTime, ForceClock

IgnoreDataChecking

Purpose	Specify number of samples during which output data checking is suppressed
Settings	<p>N</p> <p>Default: 0.</p> <p>N must be a positive integer.</p> <p>When $N > 0$, the test bench suppresses output data checking for the first N output samples after the clock enable output (<code>ce_out</code>) is asserted.</p>
Usage Notes	<p>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 <code>IgnoreDataChecking</code> accordingly.</p> <p>Be careful to specify N correctly 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.</p> <p>You should use <code>IgnoreDataChecking</code> in cases where there is any state (register) initial condition in the HDL code that does not match the Simulink state, including the following specific cases:</p> <ul style="list-style-type: none">• When you specify the <code>'DistributedPipelining', 'on'</code> parameter for the Embedded MATLAB Function block (see “Distributed Pipeline Insertion” on page 12-58).• When you specify the <code>'ResetType', 'None'</code> parameter for any of the following block types:<ul style="list-style-type: none">▪ Integer Delay▪ Tapped Delay▪ Unit Delay▪ Unit Delay Enabled• When generating a black box interface to existing manually-written HDL code.

Purpose Specify initial value driven on test bench inputs before data is asserted to DUT

Settings

- 'on'
- Initial value driven on test bench inputs is '0'.
- 'off' (default)
- Initial value driven on test bench inputs is 'X' (unknown).

InlineConfigurations

Purpose	Specify whether generated VHDL code includes inline configurations
Settings	'on' (default) Include VHDL configurations in any file that instantiates 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.
Usage Notes	VHDL 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, you should suppress the generation of inline configurations.
See Also	LoopUnrolling, SafeZeroConcat, UseAggregatesForConst, UseRisingEdge

Purpose

Specify HDL data type for model's input ports

Settings

Default (for VHDL): 'std_logic_vector'

Specifies VHDL type STD_LOGIC_VECTOR for the model's input ports.

'signed/unsigned'

Specifies VHDL type SIGNED or UNSIGNED for the model's input ports.

'wire' (Verilog)

If the target language is Verilog, the data type for all ports is wire. This property is not modifiable in this case.

See Also

ClockEnableInputPort, OutputType

InstanceGenerateLabel

Purpose Specify string to append to instance section labels in VHDL GENERATE statements

Settings 'string'
Default: '_gen'
Specify a postfix string to append to instance section labels in VHDL GENERATE statements.

See Also BlockGenerateLabel, OutputGenerateLabel

Purpose Specify string prefixed to generated component instance names

Settings 'string'
Default: 'u_'
Specify a string to be prefixed to component instance names in generated code.

LoopUnrolling

Purpose	Specify whether VHDL FOR and GENERATE loops are unrolled and omitted from generated VHDL code
Settings	<p>'on'</p> <p>Unroll and omit FOR and GENERATE loops from the generated VHDL code.</p> <p>In Verilog code, loops are always unrolled.</p> <p>If you are using an electronic design automation (EDA) tool that does not support GENERATE loops, you can enable this option to omit loops from your generated VHDL code.</p> <p>'off' (default)</p> <p>Include FOR and GENERATE loops in the generated VHDL code.</p>
Usage Notes	The setting of this option does not affect results obtained from simulation or synthesis of generated VHDL code.
See Also	InlineConfigurations, SafeZeroConcat, UseAggregatesForConst, UseRisingEdge

Purpose	Divide generated test bench into helper functions, data, and HDL test bench code files
Settings	<p>'on'</p> <p>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 <code>TestBenchPostfix</code> property, and the <code>TestBenchDataPostfix</code> property as follows:</p> <p><i>DUTname_TestBenchPostfix_TestBenchDataPostfix</i></p> <p>For example, if the DUT name is <code>symmetric_fir</code>, and the target language is VHDL, the default test bench file names are:</p> <ul style="list-style-type: none">• <code>symmetric_fir_tb.vhd</code>: test bench code• <code>symmetric_fir_tb_pkg.vhd</code>: helper functions package• <code>symmetric_fir_tb_data.vhd</code>: data package <p>If the DUT name is <code>symmetric_fir</code> and the target language is Verilog, the default test bench file names are:</p> <ul style="list-style-type: none">• <code>symmetric_fir_tb.v</code>: test bench code• <code>symmetric_fir_tb_pkg.v</code>: helper functions package• <code>symmetric_fir_tb_data.v</code>: test bench data <p>'off' (default)</p> <p>Write a single test bench file containing all HDL test bench code and helper functions and test bench data.</p>
See Also	<code>TestBenchPostFix</code> , <code>TestBenchDataPostFix</code>

OptimizeTimingController

Purpose

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

Settings

'on' (default)

A timing controller code file (`Timing_Controller.vhd` or `Timing_Controller.v`) 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.

When `OptimizeTimingController` is set 'on' (the default), the coder generates multiple counters (one counter for each rate in the model). The benefit of this optimization is that it generates faster logic, and the size of the generated code is usually much smaller.

'off'

When `OptimizeTimingController` is set 'off', the timing controller uses one counter to generate all rates in the model.

See Also

Chapter 4, “Generating HDL Code for Multirate Models”, `EnablePrefix`

Purpose	Specify string that labels output assignment block for VHDL GENERATE statements
Settings	'string' Default: 'outputgen' Specify a postfix string to append to output assignment block labels in VHDL GENERATE statements.
See Also	BlockGenerateLabel, OutputGenerateLabel

OutputType

Purpose

Specify HDL data type for model's output ports

Settings

'Same as input data type' (VHDL default)

'std_logic_vector'

Output ports have VHDL type STD_LOGIC_VECTOR.

'signed/unsigned'

Output ports have type SIGNED or UNSIGNED.

'wire' (Verilog)

If the target language is Verilog, the data type for all ports is wire. This property is not modifiable in this case.

See Also

ClockEnableInputPort, InputType

Purpose Specify string to append to specified model or subsystem name to form name of package file

Settings 'string'
Default: '_pkg'
The coder applies this option only if a package file is required for the design.

See Also ClockProcessPostfix, EntityConflictPostfix,
ReservedWordPostfix

PipelinePostfix

Purpose

Specify string to append to names of input or output pipeline registers generated for pipelined block implementations

Settings

'string'

Default: '_pipe'

Using a control file, you can specify a generation of input and/or output pipeline registers for selected blocks. The coder appends the string specified by the `PipelinePostfix` property when generating code for such pipeline registers.

For example, suppose you specify a pipelined output implementation for Product blocks in a model, as in the following excerpt from a control file:

```
c.forEach('*',...
  'built-in/Product', {},...
  'hdldefaults.ProductRTW',...
  {'OutputPipeline', 2});
```

The following `makehdl` command invokes the control file, specifying that the string `'testpipe'` is to be appended to generated pipeline registers.

```
makehdl([modelName, '/', topname], 'HDLControlFile',...
  {'sfir_fixed_pipe1_test'}, 'PipelinePostfix', 'testpipe');
```

The following excerpts from generated VHDL code show an output port definition, the associated pipeline register definition and the related process code, implementing two pipeline stages:

```
SIGNAL Product_out1          : signed(32 DOWNTO 0); -- sfix33_En20
SIGNAL Product_out1testpipe  : signed(32 DOWNTO 0); -- sfix33_En20
.
.
.
Product_out1testpipe <= Add_out1 * s_1;
Product1testpipe_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
```

```
int_delay_pipe_1(0 TO 1) <= (OTHERS => (OTHERS => '0'));
ELSIF clk'event AND clk = '1' THEN
  IF enb = '1' THEN
    int_delay_pipe_1(0) <= Product1_out1testpipe;
    int_delay_pipe_1(1) <= int_delay_pipe_1(0);
  END IF;
END IF;
END PROCESS Product1testpipe_process;
Product_out1 <= int_delay_pipe(1);
```

See Also

“Block Implementation Parameters” on page 6-41, “InputPipeline” on page 6-52, “OutputPipeline” on page 6-53

ReservedWordPostfix

Purpose Specify string appended to identifiers for entities, signals, constants, or other model elements that conflict with VHDL or Verilog reserved words

Settings 'string'
Default: '_rsvd'.
The reserved word postfix is applied 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`.

See Also `ClockProcessPostfix`, `EntityConflictPostfix`,
`ReservedWordPostfix`

Purpose

Specify asserted (active) level of reset input signal

Settings

'active-high' (default)

Specify that the reset input signal must be driven high (1) to reset registers in the model. For example, the following code fragment checks whether reset is active high before populating the delay_pipeline register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
    .
    .
    .
```

'active-low'

Specify that the reset input signal must be driven low (0) to reset registers in the model. For example, the following code fragment checks whether reset is active low before populating the delay_pipeline register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
    IF reset = '0' THEN
        delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
    .
    .
    .
```

See Also

ResetType

ResetInputPort

Purpose Name HDL port for model's reset input

Settings 'string'

Default: 'reset'.

The string specifies the name for the model's reset input port. If you override the default with (for example) the string 'chip_reset' for the generating system `myfilter`, the generated entity declaration might look as follows:

```
ENTITY myfilter IS
  PORT( clk           : IN  std_logic;
        clk_enable   : IN  std_logic;
        chip_reset    : IN  std_logic;
        myfilter_in   : IN  std_logic_vector (15 DOWNTO 0);
        myfilter_out  : OUT std_logic_vector (15 DOWNTO 0);
        );
END myfilter;
```

If you specify a 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`. See `ReservedWordPostfix` for more information.

Usage Notes

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.

See Also `ClockEnableInputPort`, `InputType`, `OutputType`

Purpose

Define length of time (in clock cycles) during which reset is asserted

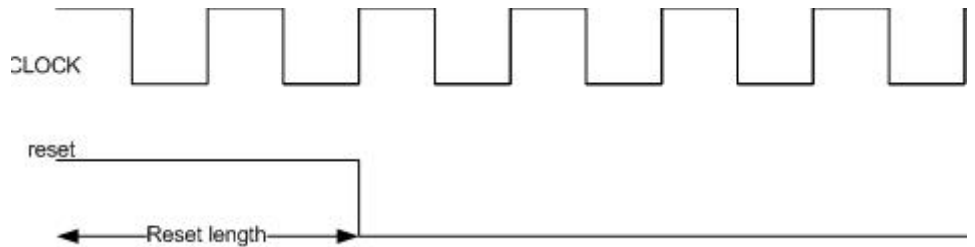
Settings

N

Default: 2.

N must be an integer greater than or equal to 0.

Resetlength defines N, the number of clock cycles during which reset is asserted. The following figure illustrates the default case, in which the reset signal (active-high) is asserted for 2 clock cycles.



ResetType

Purpose

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

Settings

'async' (default)

Use asynchronous reset logic. The following process block, generated by a Unit Delay block, illustrates the use of asynchronous resets. When the reset signal is asserted, the process block performs a reset, without checking for a clock event.

```
Unit_Delay1_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay1_out1 <= (OTHERS => '0');
  ELSIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      Unit_Delay1_out1 <= signed(x_in);
    END IF;
  END IF;
END PROCESS Unit_Delay1_process;
```

'sync'

Use synchronous reset logic. Code for a synchronous reset follows. The following process block, generated by a Unit Delay block, checks for a clock event, the rising edge, before performing a reset:

```
Unit_Delay1_process : PROCESS (clk)
BEGIN
  IF rising_edge(clk) THEN
    IF reset = '1' THEN
      Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF clk_enable = '1' THEN
      Unit_Delay1_out1 <= signed(x_in);
    END IF;
  END IF;
END PROCESS;
```

```
END PROCESS Unit_Delay1_process;
```

See Also [ResetAssertedLevel](#)

ResetValue

Purpose	Specify constant value to which test bench forces reset input signals
Settings	'active high' (default) Specify that the test bench set the reset input signal to active high (1). 'active low' Specify that the test bench set the reset input signal to active low (0).
Usage Notes	The setting for this option must match the setting of the reset asserted level specified for the test bench. The coder ignores the setting of this option if forced resets are disabled.
See Also	ForceReset, ResetType, ResetAssertedLevel

Purpose	Specify syntax for concatenated zeros in generated VHDL code
Settings	<p>'on' (default)</p> <p>Use the type-safe syntax, '0' & '0', for concatenated zeros. Typically, this syntax is preferred.</p> <p>'off'</p> <p>Use the syntax "000000..." for concatenated zeros. This syntax can be easier to read and is more compact, but it can lead to ambiguous types.</p>
See Also	LoopUnrolling, UseAggregatesForConst, UseRisingEdge

SimulatorFlags

Purpose	Specify simulator flags to apply to generated compilation scripts
Settings	'string' Default: '' Specify options that are specific to your application and the simulator you are using. For example, if you must use the 1076–1993 VHDL compiler, specify the flag <code>-93</code> .
Usage Notes	The flags you specify with this option are added to the compilation command in generated compilation scripts. The simulation command string is specified by the <code>HDLCompileVHDL</code> or <code>HDLCompileVerilog</code> properties.

Purpose	Specify string to append to specified name to form name of file containing model's VHDL architecture
Settings	'string' Default: '_arch'. This option applies only if you direct the coder to place the generated VHDL entity and architecture code in separate files.
Usage Notes	The option applies only if you direct the coder to place the filter's entity and architecture in separate files.
See Also	SplitEntityArch, SplitEntityFilePostfix

SplitEntityArch

Purpose

Specify whether generated VHDL entity and architecture code is written to single VHDL file or to separate files

Settings

'on'

Write the generated VHDL code to a single file.

'off' (default)

Write the code for the generated VHDL entity and architecture to separate files.

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`.

Note This property is specific to VHDL code generation. It does not apply to Verilog code generation and should not be enabled when generating Verilog code.

See Also

`SplitArchFilePostfix`, `SplitEntityFilePostfix`

Purpose Specify string to append to specified model name to form name of generated VHDL entity file

Settings 'string'
Default: '_entity'
This option applies only if you direct the coder to place the generated VHDL entity and architecture code in separate files.

See Also SplitArchFilePostfix, SplitEntityArch

TargetDirectory

Purpose Identify directory into which generated output files are written

Settings 'string'

Default: 'hdlsrc'

Specify a subdirectory under the current working directory into which generated files are written. The string can specify a complete path name.

If the target directory does not exist, the coder creates it.

See Also VerilogFileExtension, VHDLFileExtension

Purpose Specify HDL language to use for generated code

Settings

- 'VHDL' (default)
Generate VHDL filter code.
- 'verilog'
Generate Verilog filter code.

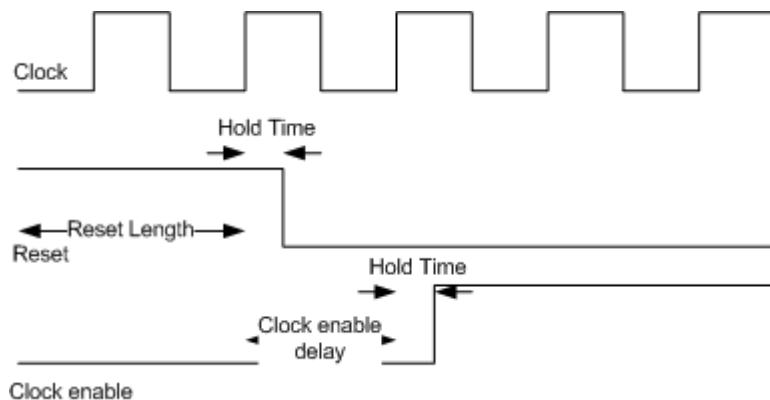
TestBenchClockEnableDelay

Purpose Define elapsed time (in clock cycles) between deassertion of reset and assertion of clock enable

Settings N (integer number of clock cycles) Default: 1

The TestBenchClockEnableDelay property specifies a delay time N, expressed in base-rate clock cycles (the default value is 1) elapsed between the time the reset signal is deasserted and the time the clock enable signal is first asserted. TestBenchClockEnableDelay works in conjunction with the HoldTime property; After deassertion of reset, the clock enable goes high after a delay of N base-rate clock cycles plus the delay specified by HoldTime.

In the figure below, the reset signal (active-high) deasserts after the interval labelled Hold Time. The clock enable asserts after a further interval labelled Clock enable delay.



See Also HoldTime, ResetLength

Purpose

Specify suffix added to test bench data file name when generating multi-file test bench

Settings

'string'

Default: '_data'.

The coder applies `TestBenchDataPostFix` only when generating a multi-file test bench (i.e. when `MultifileTestBench` is set 'on').

For example, if the name of your DUT is `my_test`, and `TestBenchPostFix` has the default value `_tb`, the coder adds the postfix `_data` to form the test bench data file name `my_test_tb_data`.

See Also

`MultifileTestBench`, `TestBenchPostFix`

TestBenchPostFix

Purpose Specify suffix to test bench name

Settings 'string'
Default: '_tb'.

For example, if the name of your DUT is `my_test`, the coder adds the postfix `_tb` to form the name `my_test_tb`.

See Also `MultifileTestBench`, `TestBenchDataPostFix`

Purpose Specify string appended to names of reference signals generated in test bench code

Settings 'string'
Default: '_ref'.
Reference signal data is represented as arrays in the generated test bench code. The string specified by TestBenchReferencePostFix is appended to the generated signal names.

Traceability

Purpose

Enable or disable creation of HTML code generation report with code-to-model and model-to-code hyperlinks

Settings

'on'

Create and display an HTML code generation report. See “Creating and Using a Code Generation Report” on page 9-2 for detailed information.

'off' (default)

Do not create an HTML code generation report.

Purpose	Specify whether all constants are represented by aggregates, including constants that are less than 32 bits
Settings	<p>'on'</p> <p>Specify that all constants, including constants that are less than 32 bits, be represented by aggregates. The following VHDL code show a scalar less than 32 bits represented as an aggregate:</p> <pre>GainFactor_gainparam <= (14 => '1', OTHERS => '0');</pre> <p>'off' (default)</p> <p>Specify that the coder represent 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:</p> <pre>GainFactor_gainparam <= to_signed(16384, 16);</pre>
See Also	LoopUnrolling, SafeZeroConcat, UseRisingEdge

UserComment

Purpose Specify comment line in header of generated HDL and test bench files

Settings 'string'

The comment is generated in each of the generated code and test bench files. The code generator adds leading comment characters as appropriate for the target language. When newlines or line feeds are included in the string, the code generator emits single-line comments for each newline.

For example, the following `makehdl` command adds two comment lines to the header in a generated VHDL file.

```
makehdl(gcb, 'UserComment', 'This is a comment line.\nThis is a second line.')
```

The resulting header comment block for subsystem `symmetric_fir` would appear as follows:

```
-- -----  
--  
-- Module: symmetric_fir  
-- Simulink Path: sfir_fixed/symmetric_fir  
-- Created: 2006-11-20 15:55:25  
-- Hierarchy Level: 0  
--  
-- This is a comment line.  
-- This is a second line.  
--  
-- Simulink model description for sfir_fixed:  
-- This model shows how to use Simulink HDL Coder to check, generate,  
-- and verify HDL for a fixed-point symmetric FIR filter.  
--  
-- -----
```

Purpose

Specify VHDL coding style used to check for rising edges when operating on registers

Settings

'on'

Use the VHDL `rising_edge` function to check for rising edges when operating on registers. The following code, generated from a Unit Delay block, tests `rising_edge` as shown in the following PROCESS block:

```
Unit_Delay1_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay1_out1 <= (OTHERS => '0');
  ELSIF rising_edge(clk) THEN
    IF clk_enable = '1' THEN
      Unit_Delay1_out1 <= signed(x_in);
    END IF;
  END IF;
END PROCESS Unit_Delay1_process;
```

'off' (default)

Check for clock events when operating on registers. The following code, generated from a Unit Delay block, checks for a clock event as shown in the ELSIF statement of the following PROCESS block:

```
Unit_Delay1_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay1_out1 <= (OTHERS => '0');
  ELSIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      Unit_Delay1_out1 <= signed(x_in);
    END IF;
  END IF;
```

UseRisingEdge

```
END PROCESS Unit_Delay1_process;
```

Usage Notes

The two coding styles have different simulation behavior when the clock transitions from 'X' to '1'.

See Also

LoopUnrolling, SafeZeroConcat, UseAggregatesForConst

Purpose	Use compiler <code>`timescale</code> directives in generated Verilog code
Settings	<code>'on'</code> (default) Use compiler <code>`timescale</code> directives in generated Verilog code. <code>'off'</code> Suppress the use of compiler <code>`timescale</code> directives in generated Verilog code.
Usage Notes	The <code>`timescale</code> 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.
See Also	LoopUnrolling, SafeZeroConcat, UseAggregatesForConst, UseRisingEdge

VectorPrefix

Purpose Specify string prefixed to vector names in generated code

Settings 'string'
Default: 'vector_of_'
Specify a string to be prefixed to vector names in generated code.

Purpose

Specify level of detail for messages displayed during code generation

Settings

n

Default: 0 (minimal messages displayed).

When *Verbosity* is set to 0, minimal code generation progress messages are displayed as code generation proceeds. When *Verbosity* is set to 1, more detailed progress messages are displayed.

VerilogFileExtension

Purpose Specify file type extension for generated Verilog files

Settings 'string'
The default file type extension for generated Verilog files is .v.

See Also TargetLanguage

Purpose	Specify file type extension for generated VHDL files
Settings	'string' The default file type extension for generated VHDL files is .vhd.
See Also	TargetLanguage

VHDLFileExtension

Function Reference

Code Generation Functions

makehdl

Generate HDL RTL code from model or subsystem

makehdltb

Generate HDL test bench from model

Utility Functions

<code>checkhdl</code>	Check subsystem or model for compatibility with HDL code generation
<code>hdl1lib</code>	Create library of blocks that support HDL code generation
<code>hdlnewblackbox</code>	Generate customizable control file from selected subsystem or blocks
<code>hdlsetup</code>	Set model parameters for HDL code generation

Control File Utilities

<code>hdlnewcontrol</code>	Construct a code generation control object for use in a control file
<code>hdlnewcontrolfile</code>	Generate customizable control file from selected subsystem or blocks
<code>hdlnewforeach</code>	Generate <code>forEach</code> calls for insertion into code generation control files

Functions — Alphabetical List

checkhdl

Purpose Check subsystem or model for compatibility with HDL code generation

Syntax

```
checkhdl
checkhdl(bdroot)
checkhdl('modelName')
checkhdl('modelName/subsys')
checkhdl(gcb)
output = checkhdl(arg)
```

Description checkhdl is a utility that checks a subsystem or model for compatibility with HDL code generation. If any incompatibilities are detected (for example, use of unsupported blocks or illegal data type usage), checkhdl displays information on the blocks and potential problems in an HTML report.

checkhdl examines (by default) the current model for compatibility with HDL code generation.

checkhdl(bdroot) examines the current model for compatibility with HDL code generation.

checkhdl('modelName') examines the model explicitly specified by 'modelName' for compatibility with HDL code generation.

checkhdl('modelName/subsys') examines a specified subsystem within the model specified by 'modelName' for compatibility with HDL code generation.

'subsys' specifies the name of the subsystem to be checked. In the current release, 'subsys' must be at the top (root) level of the current model; it cannot be a subsystem nested at a lower level of the model hierarchy.

checkhdl(gcb) examines the currently selected subsystem within the current model for compatibility with HDL code generation.

checkhdl generates an HTML HDL Code Generation Check Report. The report file-naming convention is *system_report.html*, where *system* is the name of the subsystem or model that was passed in to

`checkhdl`. The report is written to the target directory. `checkhdl` also displays the report in a browser window.

The report is in table format. Each entry in the table is hyperlinked to a block or subsystem that caused a problem. When you click the hyperlink, the block of interest highlights and displays (provided that the model referenced by the report is open).

If no errors are encountered, the report contains only a hyperlink to the subsystem or model that was checked.

Alternatively, you can also specify an output argument, using the following syntax:

```
output = checkhdl(arg)
```

where *arg* specifies a model or subsystem in any of the forms described previously.

When an output argument is specified, `checkhdl` returns a 1xN struct array with one entry for each error, warning or message. In this case, no report is generated (see “Examples” on page 17-4).

Use `checkhdl` to check your subsystems or models before generating HDL code.

`checkhdl` reports three levels of compatibility problems:

- *Errors*: Errors will cause `makehdl` to error out. These issues must be fixed before HDL code can be generated. A typical error would be the use of an unsupported data type.
- *Warnings*: Warnings may cause problems in the generated code, but generally allow HDL code generation to continue. For example, the presence of an unsupported block in the model would raise a warning. In this case, the code generator attempts to proceed as if the block were not present in the design. This could lead to errors later in the code generation process, which would then terminate.
- *Messages*: Messages are indications that the HDL code generator may treat data types in a way that differs from what might be expected. For example, single-precision floating-point data types are

checkhdl

automatically converted to double-precision because neither VHDL nor Verilog support single-precision data types.

Note If a model or subsystem passes `checkhdl` without errors, that does *not* imply that `makehdl` will complete successfully, since not all block parameters are verified in this release. However, if `checkhdl` reports an error, `makehdl` will not successfully complete HDL code generation.

For convenience, `checkhdl` also takes the same property-value pairs as `makehdl` and `makehdltb`.

Examples

The following example checks the subsystem `symmetric_fir` within the model `sfir_fixed` for HDL code generation compatibility. If problems are encountered, an HTML report is generated.

```
checkhdl('sfir_fixed/symmetric_fir')
```

The following example checks the subsystem `symmetric_fir_err` within the model `sfir_fixed_err` for HDL code generation compatibility. Information on problems encountered is returned in the struct `output`. The first element of `output` is then displayed.

```
output = checkhdl('sfir_fixed_err/symmetric_fir_err')
### Starting HDL Check.
...
### HDL Check Complete with 4 errors, warnings and messages.

output =

1x4 struct array with fields:
    path
    type
    message
    level
```

```
output(1)

ans =

    path: 'sfir_fixed_err/symmetric_fir_err/Product'
    type: 'block'
message: 'Unhandled mixed double and non-double datatypes at ports of block'
level: 'Error'
```

See Also [makehdl](#)

hdlLib

Purpose	Create library of blocks that support HDL code generation
Syntax	<code>hdlLib</code>
Description	<p><code>hdlLib</code> creates a library of blocks that are supported for HDL code generation. The library is named <code>hdlSupported.mdl</code>. After the library is generated, you must save it to a directory of your choice.</p> <p><code>hdlLib</code> loads many block libraries during the creation of the <code>hdlSupported</code> library. (This will cause a license checkout.) When <code>hdlLib</code> completes generation of the library, it does not unload block libraries.</p> <p>The <code>hdlSupported</code> library affords quick access to all supported blocks. By constructing models using blocks from this library, you can ensure block-level compatibility of your model with the coder.</p> <p>Parameter settings for blocks in the <code>hdlSupported</code> library may differ from corresponding blocks in other libraries.</p> <p>The set of supported blocks will change in future releases of the coder. To keep the <code>hdlSupported.mdl</code> current, you should rebuild the library and table each time you install a new release.</p>

Purpose Generate customizable control file from selected subsystem or blocks

Syntax

```
hdlnewblackbox
hdlnewblackbox('blockpath')
hdlnewblackbox({'blockpath1','blockpath2',... 'blockpathN'})
[cmd, impl] = hdlnewblackbox
[cmd, impl] = hdlnewblackbox('blockpath')
[cmd, impl] = hdlnewblackbox({'blockpath1','blockpath2',
    ... 'blockpathN'})
[cmd, impl, params] = hdlnewblackbox
[cmd, impl, params] = hdlnewblackbox('blockpath')
[cmd, impl, params] = hdlnewblackbox({'blockpath1',
    'blockpath2',... 'blockpathN'})
```

Description The `hdlnewblackbox` utility helps you construct `forEach` calls for use in code generation control files when generating black box interfaces. Given a selection of one or more blocks from your model, `hdlnewblackbox` returns the following as string data in the MATLAB workspace for each selected block:

- A `forEach` call coded with the correct `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, in `package.class` form.
- (Optional) A cell array of cell arrays of strings enumerating the names of implementation parameters (if any) corresponding to the implementation classes. `hdlnewblackbox` does not list data types and other details of implementation parameters.

`hdlnewblackbox` returns a `forEach` call for each selected block in the model.

`hdlnewblackbox('blockpath')` returns a `forEach` call for the block specified by the `'blockpath'` argument. The `'blockpath'` argument is a string specifying the full Simulink path to the desired block.

`hdlnewblackbox({'blockpath1','blockpath2',... 'blockpathN'})` returns a `forEach` call for the blocks specified by the `{'blockpath1','blockpath2',... 'blockpathN'}` arguments. The `{'blockpath1','blockpath2',... 'blockpathN'}` arguments are passed as a cell array of strings, each string specifying the full Simulink path to a desired block.

`[cmd, impl] = hdlnewblackbox` returns a `forEach` call for each selected block in the model to the string variable `cmd`. The call also returns `impl`, a cell array of cell arrays of strings enumerating the available implementations for the block.

`[cmd, impl] = hdlnewblackbox('blockpath')` returns a `forEach` call for the block specified by the `'blockpath'` argument to the string variable `cmd`. The call also returns `impl`, a cell array of cell arrays of strings enumerating the available implementations for the block. The `'blockpath'` argument is a string specifying the full Simulink path to the desired block.

`[cmd, impl] = hdlnewblackbox({'blockpath1','blockpath2',... 'blockpathN'})` returns a `forEach` call for the blocks specified by the `{'blockpath1','blockpath2',... 'blockpathN'}` arguments to the string variable `cmd`. The call also returns `impl`, a cell array of cell arrays of strings enumerating the available implementations for the block. The `{'blockpath1','blockpath2',... 'blockpathN'}` arguments are passed as a cell array of strings, each string specifying the full Simulink path to a desired block.

`[cmd, impl, params] = hdlnewblackbox` returns a `forEach` call for each selected block in the model to the string variable `cmd`. The call also returns:

- `impl`, a cell array of cell arrays of strings enumerating the available implementations for the block.
- `params`, a cell array of cell arrays of strings enumerating the available implementation parameters corresponding to each implementation.

`[cmd, impl, params] = hdlnewblackbox('blockpath')` returns a `forEach` call for the block specified by the `'blockpath'` argument to the string variable `cmd`. The call also returns:

- `impl`, a cell array of cell arrays of strings enumerating the available implementations for the block.
- `params`, a cell array of cell arrays of strings enumerating the available implementation parameters corresponding to each implementation.

The `'blockpath'` argument is a string specifying the full Simulink path to the desired block.

`[cmd, impl, params] = hdlnewblackbox({'blockpath1','blockpath2',... 'blockpathN'})` returns a `forEach` call for the blocks specified by the `{'blockpath1','blockpath2',... 'blockpathN'}` arguments to the string variable `cmd`. The call also returns:

- `impl`, a cell array of cell arrays of strings enumerating the available implementations for the block.
- `params`, a cell array of cell arrays of strings enumerating the available implementation parameters corresponding to each implementation.

The `{'blockpath1','blockpath2',... 'blockpathN'}` arguments are passed as a cell array of strings, each string specifying the full Simulink path to a desired block.

Usage Notes

Before invoking `hdlnewblackbox`, you must run `checkhdl` or `makehdl` to build in-memory information about the model. If you do not run `checkhdl` or `makehdl`, `hdlnewblackbox` will display an error message indicating that you should run `checkhdl` or `makehdl`.

After invoking `hdlnewblackbox`, you will generally want to insert the `forEach` calls returned by the function into a control file, and use the implementation information returned to specify a nondefault block implementation.

hdlnewblackbox

Examples

```
% Return a forEach call for a specific subsystem to the MATLAB workspace
hdlnewblackbox('sfir_fixed/symmetric_fir');
%
% Return forEach calls for all currently selected blocks to the MATLAB workspace
hdlnewblackbox;
%
% Return forEach calls, implementation names, and implementation parameter names
% for all currently selected blocks to string variables
[cmd,impl,parms] = hdlnewblackbox;
```


Purpose Construct a code generation control object for use in a control file

Syntax `object = hdlnewcontrol(mfilename)`

Description `object = hdlnewcontrol(mfilename)` constructs and returns a control generation control object (`object`) that is linked to a code generation control file.

The argument to `hdlnewcontrol` is the name of the control file itself. Use the `mfilename` function to pass in the file name string.

Tip The `hdlnewcontrol` function constructs an instance of the class `hdlnewcontrol` is a wrapper function provided to let you instantiate such objects. You should not directly call the constructor of the class.

In your control files, use only the public methods of the class `slhdlcoder.ConfigurationContainer`. All public methods are described in this document. In your control files. All other methods of this class are for MathWorks internal development use only.

See also

- Chapter 5, “Code Generation Control Files”

hdlnewcontrolfile

Purpose Generate customizable control file from selected subsystem or blocks

Syntax

```
hdlnewcontrolfile
hdlnewcontrolfile('blockpath')
hdlnewcontrolfile({'blockpath1','blockpath2',
    ...'blockpathN'})
t = hdlnewcontrolfile(...)
```

Description 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:

- A `c.generateHDLFor` call specifying the full path to the currently selected block or subsystem from which code is to be generated.
- `c.forEach` calls for all selected blocks that have HDL implementations.
- Comments providing information about all supported implementations and parameters for all selected blocks that have HDL implementations.
- `c.set` calls for any global HDL Coder options that are set to nondefault values.

Generated control files are automatically opened as untitled files in the MATLAB editor for further customization. The file naming sequence for successively generated control files is `Untitled1`, `Untitled2`, ... `UntitledN`.

To use a generated control file in code generation, you must save it and attach it to a model. (See also “Associating an Existing Control File with Your Model” on page 5-19.)

`hdlnewcontrolfile` returns a control file containing a `forEach` statement and comments for each selected block in the model.

`hdlnewcontrolfile('blockpath')` returns a control file containing a `forEach` statement and comments for the block specified by the

'blockpath' argument. The 'blockpath' argument is a string specifying the full Simulink path to the desired block.

`hdlnewcontrolfile({'blockpath1','blockpath2',... 'blockpathN'})` returns a control file containing a `forEach` statement and comments for the blocks specified by the `{'blockpath1','blockpath2',... 'blockpathN'}` arguments. The `{'blockpath1','blockpath2',... 'blockpathN'}` arguments are passed as a cell array of strings, each string specifying the full Simulink path to a desired block.

`t = hdlnewcontrolfile(...)` returns control statements as text in the string variable `t`, instead of returning a control file.

Usage Notes

You can use the generated control file as:

- A starting point for development of a customized control file.
- A source of information or documentation of the HDL code generation parameter settings in the model.

Examples

```
% Generate control file for a specific block
hdlnewcontrolfile('sfir_fixed/symmetric_fir/Product1');
%
% Generate a control file for all currently selected blocks
hdlnewcontrolfile;
%
% Generate a control file for two specific blocks
hdlnewcontrolfile({'sfir_fixed/symmetric_fir/Add1',...
'sfir_fixed/symmetric_fir/Product2'});
```

hdlnewforeach

Purpose Generate forEach calls for insertion into code generation control files

Syntax

```
hdlnewforeach
hdlnewforeach('blockpath')
hdlnewforeach({'blockpath1','blockpath2',...})
[cmd, impl] = hdlnewforeach
[cmd, impl] = hdlnewforeach('blockpath')
[cmd, impl] = hdlnewforeach({'blockpath1','blockpath2',...})
[cmd, impl, parms] = hdlnewforeach
[cmd, impl, parms] = hdlnewforeach('blockpath')
[cmd, impl, parms] = hdlnewforeach({'blockpath1','blockpath2',
    ...})
```

Description The coder provides the `hdlnewforeach` utility to help you construct `forEach` calls for use in code generation 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 correct `modelscope`, `blocktype`, and default implementation arguments for the block.
- (Optional) A cell array of cell arrays of strings enumerating the available implementations for the block, in `package.class` form.
- (Optional) A cell array of cell arrays of strings enumerating the names of implementation parameters (if any) corresponding to the block implementations. See “Block Implementation Parameters” on page 6-41 for that data types and other details of block implementation parameters.

`hdlnewforeach` returns a `forEach` call for each selected block in the model. Each call is returned as a string.

`hdlnewforeach('blockpath')` returns a `forEach` call for a specified block in the model. The call is returned as a string.

The `'blockpath'` argument is a string specifying the full path to the desired block.

`hdlnewforeach({'blockpath1', 'blockpath2', ...})` returns a `forEach` call for each specified block in the model. Each call is returned as a string.

The `{'blockpath1', 'blockpath2', ...}` argument is a cell array of strings, each of which specifies the full path to a desired block.

`[cmd, impl] = hdlnewforeach` returns a `forEach` call for each selected block in the model to the string variable `cmd`. In addition, the call returns a cell array of cell arrays of strings (`impl`) enumerating the available implementations for the block.

`[cmd, impl] = hdlnewforeach('blockpath')` returns a `forEach` call for a specified block in the model to the string variable `cmd`. In addition, the call returns a cell array of cell arrays of strings (`impl`) enumerating the available implementations for the block.

The `'blockpath'` argument is a string specifying the full path to the desired block.

`[cmd, impl] = hdlnewforeach({'blockpath1', 'blockpath2', ...})` returns a `forEach` call for each specified block in the model to the string variable `cmd`. In addition, the call returns a cell array of cell arrays of strings (`impl`) enumerating the available implementations for the block.

The `{'blockpath1', 'blockpath2', ...}` argument is a cell array of strings, each of which specifies the full path to a desired block.

`[cmd, impl, parms] = hdlnewforeach` returns a `forEach` call for each selected block in the model to the string variable `cmd`. In addition, the call returns:

- A cell array of cell arrays of strings (`impl`) enumerating the available implementations for the block.
- A cell array of cell arrays of strings (`parms`) enumerating the available implementation parameters corresponding to each implementation.

hdlnewforeach

`[cmd, impl, parms] = hdlnewforeach('blockpath')` returns a `forEach` call for a specified block in the model to the string variable `cmd`. In addition, the call returns:

- A cell array of cell arrays of strings (`impl`) enumerating the available implementations for the block.
- A cell array of cell arrays of strings (`parms`) enumerating the available implementation parameters corresponding to each implementation.

The `'blockpath'` argument is a string specifying the full path to the desired block.

`[cmd, impl, parms] = hdlnewforeach({'blockpath1', 'blockpath2', ...})` returns a `forEach` call for each specified block in the model to the string variable `cmd`. In addition, the call returns:

- A cell array of cell arrays of strings (`impl`) enumerating the available implementations for the block.
- A cell array of cell arrays of strings (`parms`) enumerating the available implementation parameters corresponding to each implementation.

The `{'blockpath1', 'blockpath2', ...}` argument is a cell array of strings, each of which specifies the full path to a desired block.

Usage Notes

Before invoking `hdlnewforeach`, you must build in-memory information about the model once. To do this, run `checkhdl`. If you do not run `checkhdl`, `hdlnewforeach` will display an error message indicating that you should run `checkhdl` or `makehdl`.

`hdlnewforeach` returns an empty string for blocks that do not have an HDL implementation. `hdlnewforeach` also returns an empty string for subsystems, which are a special case. Subsystems do not have a default implementation class, but special-purpose subsystems implementations are provided (see Chapter 10, “Interfacing Subsystems and Models to HDL Code”).

After invoking `hdlnewforeach`, you will generally want to insert the `forEach` calls returned by the function into a control file, and use the implementation and parameter information returned to specify a nondefault block implementation. See “Generating Selection/Action Statements with the `hdlnewforeach` Function” on page 5-24 for a worked example.

Examples

The following example generates `forEach` commands for two explicitly specified blocks.

```
hdlnewforeach({'sfir_fixed/symmetric_fir/Add4',...
'sfir_fixed/symmetric_fir/Product2'})

ans =

c.forEach('sfir_fixed/symmetric_fir/Add4',...
'built-in/Sum', {},...
'hdldefaults.SumRTW', {});

c.forEach('sfir_fixed/symmetric_fir/Product2',...
'built-in/Product', {},...
'hdldefaults.ProductRTW', {});
```

The following example generates a `forEach` command for an explicitly specified `Sum` block. The implementation and parameters information returned is listed after the `forEach` command.

```
[cmd,impl, parms] = hdlnewforeach('sfir_fixed/symmetric_fir/Add4')

cmd =

c.forEach('sfir_fixed/symmetric_fir/Add4',...
'built-in/Sum', {},...
'hdldefaults.SumRTW', {});

impl =
```

hdlnewforeach

```
        {4x1 cell}

parms =

        {1x2 cell}    {1x2 cell}    {1x2 cell} {1x2 cell}
>> impl{1}

ans =

        'hdldefaults.SumCascadeHDL Emission'
        'hdldefaults.SumLinearHDL Emission'
        'hdldefaults.SumRTW'
        'hdldefaults.SumTreeHDL Emission'

>> parms{1:4}

ans =

        'InputPipeline'    'OutputPipeline'

ans =

        'InputPipeline'    'OutputPipeline'

ans =

        'InputPipeline'    'OutputPipeline'

ans =

        'InputPipeline'    'OutputPipeline'
```


Purpose Set model parameters for HDL code generation

Syntax `hdlsetup`
`hdlsetup('model')`

Description `hdlsetup` changes the parameters of the current model (bdroot) to values that are commonly used for HDL code generation.

`hdlsetup('model')` changes the parameters of the model specified by the 'model' argument to values that are commonly used for HDL code generation.

A model should be open before you invoke the `hdlsetup` command.

The `hdlsetup` command uses the `set_param` function to set up models for HDL code generation quickly and consistently. The model parameters settings provided by `hdlsetup` are intended as useful defaults, but they may not be appropriate for all your applications.

To view the complete set of model parameters affected by `hdlsetup`, view `hdlsetup.m` in the MATLAB editor.

See the “Model Parameters” table in the “Model and Block Parameters” section of the Simulink documentation for a summary of user-settable model parameters.

How `hdlsetup` Configures Solver Options

`hdlsetup` configures **Solver** options that are recommended or required by the coder. These are

- **Type: Fixed-step.** This is the recommended solver type for most HDL applications.

The coder currently supports variable-step solvers under the following limited conditions:

- The device under test (DUT) is single-rate.
- The sample times of all signals driving the DUT are greater than 0.

- **Solver:** Discrete (no continuous states). Other fixed-step solvers could be selected, but this option is usually the correct 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.

Purpose

Generate HDL RTL code from model or subsystem

Syntax

```
makehdl()  
makehdl(bdroot)  
makehdl('modelName')  
makehdl('modelName/subsys')  
makehdl(gcb)  
makehdl('PropertyName', PropertyValue,...)  
makehdl(bdroot, 'PropertyName', PropertyValue,...)  
makehdl('modelName', 'PropertyName', PropertyValue,...)  
makehdl('modelName/subsys', 'PropertyName', PropertyValue,...)  
makehdl(gcb, 'PropertyName', PropertyValue,...)
```

Description

makehdl generates HDL RTL code (VHDL or Verilog) from a model or subsystem. We will refer to a model or subsystem from which code is generated as the *device under test (DUT)*.

makehdl() generates HDL code from the current model (by default), using default values for all properties.

makehdl(bdroot) generates HDL code from the current model, using default values for all properties.

makehdl('modelName') generates HDL code from the model explicitly specified by 'modelName', using default values for all properties.

makehdl('modelName/subsys') generates HDL code from a subsystem within the model specified by 'modelName', using default values for all properties.

'subsys' specifies the name of the subsystem. In the current release, this must be a subsystem at the top (root) level of the current model; it cannot be a subsystem nested at a lower level of the model hierarchy.

makehdl(gcb) generates HDL code from the currently selected subsystem within the current model, using default values for all properties.

`makehdl('PropertyName', PropertyValue, ...)` generates HDL code from the current model (by default), explicitly specifying one or more code generation options as property/value pairs.

`makehdl(bdroot, 'PropertyName', PropertyValue, ...)` generates HDL code from the current model, explicitly specifying one or more code generation options as property/value pairs.

`makehdl('modelname', 'PropertyName', PropertyValue, ...)` generates HDL code from the model explicitly specified by 'modelname', explicitly specifying one or more code generation options as property/value pairs.

`makehdl('modelname/subsys', 'PropertyName', PropertyValue, ...)` generates HDL code from a subsystem within the model specified by 'modelname', explicitly specifying one or more code generation options as property/value pairs.

'subsys' specifies the name of the subsystem. In the current release, this must be a subsystem at the top (root) level of the current model; it cannot be a subsystem nested at a lower level of the model hierarchy.

`makehdl(gcb, 'PropertyName', PropertyValue, ...)` generates HDL code from the currently selected subsystem within the current model, explicitly specifying one or more code generation options as property/value pairs.

Property/value pairs are passed in the form

```
'PropertyName', PropertyValue
```

These property settings determine characteristics of the generated code, such as HDL element naming and whether certain optimizations are applied. The next section, “HDL Code Generation Defaults” on page 17-23, summarizes the default actions of the code generator.

For detailed descriptions of each property and its effect on generated code, see Chapter 15, “Properties — Alphabetical List”, and Chapter 14, “Property Reference”.

HDL Code Generation Defaults

This section summarizes the default actions of the code generator. Most defaults can be overridden by passing in appropriate property/value settings to `makehdl`. Chapter 15, “Properties — Alphabetical List” describes all `makehdl` properties in detail.

Target Language, File Packaging and Naming

- The `TargetLanguage` property determines whether VHDL or Verilog code is generated. The default is VHDL.
- `makehdl` writes generated files to `hdlsrc`, a subdirectory of the current working directory. This directory is called the *target directory*. `makehdl` creates a target directory if it does not already exist.
- `makehdl` generates separate HDL source files for the DUT and each subsystem within it. In addition, `makehdl` generates script files for HDL simulation and synthesis tools. File names derive from the name of the DUT. File names are assigned by the coder and are not user-assignable. The following table summarizes file-naming conventions.

File	Name
Verilog source code	<code>system.v</code> , where <i>system</i> is the name of the DUT.
VHDL source code	<code>system.vhd</code> , where <i>system</i> is the name of the DUT.

File	Name
Timing controller code	Timing_Controller.vhd (VHDL) or Timing_Controller.v (Verilog). This file contains a module defining timing signals (clock, reset, external clock enable inputs and clock enable output) in a separate entity or module. Timing controller code is generated if required by the design; a purely combinatorial model does not generate timing controller code.
Mentor Graphics ModelSim compilation script	<i>system_compile.do</i> , where <i>system</i> is the name of the DUT.
Synplify synthesis script	<i>system_synplify.tcl</i> , where <i>system</i> is the name of the DUT.
VHDL package file	<i>system_pkg.vhd</i> , where <i>system</i> is the name of the DUT. A package file is generated only if the design requires a VHDL package.
Mapping file	<i>system_map.txt</i> , where <i>system</i> is the name of the DUT. This report file maps generated entities (or modules) to the subsystems that generated them. See “Code Tracing Using the Mapping File” on page 9-25.

Entities, Ports, and Signals

- Unique names are assigned to generated VHDL entities or Verilog modules. Entity or module names are derived from the names of the DUT. Name conflicts are resolved by the use of a postfix string.
- HDL port names are assigned according to the following conventions:

HDL Port	Name
Input	Same as corresponding port name on the DUT (name conflicts resolved according to rules of the target language)
Output	Same as corresponding port name on the DUT (name conflicts resolved according to rules of the target language)
Clock input	clk
Clock enable input	clk_enable
Clock enable output	ce_out
Reset input	reset

- HDL port directions and data types
 - Port direction: IN or OUT, corresponding to the port on the DUT.
 - Clock, clock enable, and reset port data types: VHDL type `STD_LOGIC_VECTOR` or Verilog type `wire`.
 - Input and output port data types: VHDL type `STD_LOGIC_VECTOR` or Verilog type `wire`. Port widths are determined by the model.
- HDL signal names and data types:

- HDL signals generated from named signals in the model retain their signal names.
- For unnamed signals in the model, HDL signal names are derived from the concatenated names of the block and port connected to the signal in the DUT: *blockname_portname*. Conflicting names are made unique according to VHDL or Verilog rules.
- Signal data types are determined by the data type of the corresponding signal in the model. Each signal declaration is annotated with a comment indicating the data type.

General HDL Code Settings

- VHDL-specific defaults:
 - Generated VHDL files include both entity and architecture code.
 - VHDL configurations are placed in any file that instantiates a component.
 - VHDL code checks for rising edges via the logic `IF clock'event AND clock='1' THEN...`, when operating on registers.
 - When creating labels for VHDL GENERATE statements, `makehdl` appends `_gen` to section and block names. `makehdl` names output assignment block labels with the string `outputgen`.
- A type-safe representation is used for concatenated zeros: `'0' & '0'...`
- Generated code for registers uses asynchronous reset logic with an active-high (1) reset level.
- The postfix string `_process` is appended to process names.
- On Microsoft® Windows® platforms, carriage return/linefeed (CRLF) character sequences are never emitted in generated code.

Code Optimizations

- In general, generated HDL code produces results that are bit-true and cycle-accurate with respect to the original model (that is, the HDL code exactly reproduces simulation results from the model).

However, some block implementations generate code that includes certain block-specific performance and area optimizations. These optimizations can produce numeric results or timing differences that differ from those produced by the original model (see Chapter 8, “Generating Bit-True Cycle-Accurate Models”).

Examples

- The following call to `makehdl` generates Verilog code for the subsystem `symmetric_fir` within the model `sfir_fixed`.

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

- The following call to `makehdl` generates VHDL code for the current model. Code is generated into the target directory `hdlsrc`, with all code generation options set to default values.

```
makehdl(bdroot)
```

- The following call to `makehdl` directs the HDL compatibility checker (see `checkhdl`) to check the subsystem `symmetric_fir` within the model `sfir_fixed` before code generation starts. If no compatibility errors are encountered, `makehdl` generates VHDL code for the subsystem `symmetric_fir`. Code is generated into the target directory `hdlsrc`, with all code generation options set to default values.

```
makehdl('sfir_fixed/symmetric_fir','CheckHDL','on')
```

See Also

`makehdltb`, `checkhdl`

makehdltb

Purpose Generate HDL test bench from model

Syntax

```
makehdltb('modelName/subsys')  
makehdltb('modelName/subsys', 'PropertyName', PropertyValue,  
...)
```

Description `makehdltb('modelName/subsys')` generates an HDL test bench from the specified subsystem within a model, using default values for all properties. The *modelName/subsys* argument gives the path to the subsystem under test. This subsystem must be at the top (root) level of the current model. The generated test bench is designed to interface to and validate HDL code generated from *subsys* (or from a subsystem with a functionally identical public interface).

A typical practice is to generate HDL code for a subsystem, followed immediately by generation of a test bench to validate the same subsystem (see “Examples” on page 17-31).

Note If `makehdl` has not previously executed successfully within the current session, `makehdltb` generates model code before generating the test bench code.

Test bench code and model code must both be generated in the same target language. If the target language specified for `makehdltb` differs from the target language specified for the previous `makehdl` execution, `makehdltb` will regenerate model code in the same language specified for the test bench.

Properties passed in to `makehdl` persist after `makehdl` executes, and (unless explicitly overridden) will be passed in to subsequent `makehdltb` calls during the same session.

```
makehdltb('modelName/subsys', 'PropertyName',  
PropertyValue,...) generates an HDL test bench from the specified
```

subsystem within a model, explicitly specifying one or more code generation options as property/value pairs.

Property/value pairs are passed in the form

```
'PropertyName', PropertyValue
```

These property settings determine characteristics of the test bench code. Many of these properties are identical to those for `makehdl`, while others are specific to test bench generation (for example, options for generation of test bench stimuli). The next section, “Defaults for Test Bench Code Generation” on page 17-29, summarizes the defaults that are specific to generated test bench code.

For detailed descriptions of each property and its effect on generated code, see Chapter 15, “Properties — Alphabetical List”, and Chapter 14, “Property Reference”.

Generating Stimulus and Output Data

`makehdltb` generates test data from signals connected to inputs of the subsystem under test. Sample values for each stimulus signal are computed and stored for each time step of the simulation. The signal data is represented as arrays in the generated test bench code.

To help you validate generated HDL code, `makehdltb` also generates output data from signals connected to outputs of the subsystem under test. Like input data, sample values for each output signal are computed and stored for each time step of the simulation. The signal data is represented as arrays in the generated test bench code.

The total simulation time (set by the model’s **Stop Time** parameter) determines the size of the stimulus and output data arrays. Computation of sample values can be time-consuming. Consider speeding up generation of signal data by entering a shorter **Stop Time**.

Defaults for Test Bench Code Generation

This section describes defaults that apply specifically to generation of test bench code. `makehdltb` has many properties and defaults in

common with makehdl. See “HDL Code Generation Defaults” on page 17-23 for a summary of these common properties and defaults.

File Packaging and Naming

By default, makehdltb generates an HDL source file containing test bench code and arrays of stimulus and output data. In addition, makehdltb generates script files that let you execute a simulation of the test bench and the HDL entity under test. Generated test bench file names (like makehdl generated file names) are based on the name of the DUT. The following table summarizes the default test bench file-naming conventions.

File	Name
Verilog test bench	<i>system_tb.v</i> , where <i>system</i> is the name of the system under test
VHDL test bench	<i>system_tb.vhd</i> , where <i>system</i> is the name of the system under test
Mentor Graphics ModelSim compilation script	<i>system_tb_compile.do</i> , where <i>system</i> is the name of the DUT
Mentor Graphics ModelSim simulation script	<i>system_tb_sim.do</i> , where <i>system</i> is the name of the DUT

Other Test Bench Settings

- The test bench forces clock, clock enable, and reset input signals.
- The test bench forces clock enable and reset input to active high (1).

- The clock input signal is driven high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- The test bench forces reset signals.
- The test bench applies a hold time of 2 nanoseconds to reset and data input signals.

Examples

In the following example, `makehdl` generates VHDL code for the subsystem `symmetric_fir`. After the coder indicates successful completion of code generation, `makehdltb` generates a VHDL test bench for the same subsystem.

```
makehdl('sfir_fixed/symmetric_fir')
### Applying HDL Code Generation Control Statements

### Begin VHDL Code Generation
### Working on sfir_fixed/symmetric_fir ashdlsrc\symmetric_fir.vhd
### HDL Code Generation Complete.
makehdltb('sfir_fixed/symmetric_fir')
### Begin TestBench Generation
### Generating Test bench:hdlsrc\symmetric_fir_tb.vhd
### Please wait ...

### HDL TestBench Generation Complete.
```

See Also

`makehdl`

Examples

Use this list to find examples in the documentation.

Generating HDL Code Using the Command Line Interface

- “Creating a Directory and Local Model File” on page 2-7
- “Initializing Model Parameters with hdlsetup” on page 2-8
- “Generating a VHDL Entity from a Subsystem” on page 2-10
- “Generating VHDL Test Bench Code” on page 2-12
- “Verifying Generated Code” on page 2-13

Generating HDL Code Using the GUI

- “Creating a Directory and Local Model File” on page 2-19
- “Viewing Coder Options in the Configuration Parameters Dialog Box” on page 2-20
- “Creating a Control File” on page 2-22
- “Initializing Model Parameters with hdlsetup” on page 2-24
- “Selecting and Checking a Subsystem for HDL Compatibility” on page 2-26
- “Generating VHDL Code” on page 2-27
- “Generating VHDL Test Bench Code” on page 2-30
- “Verifying Generated Code” on page 2-31

Verifying Generated HDL Code in an HDL Simulator

- “Simulating and Verifying Generated HDL Code” on page 2-32

A

- addition operations
 - typecasting 15-3
- advanced coding properties 14-6
- application-specific integrated circuits (ASICs) 1-2
- architectures
 - setting postfix from command line 15-67
- asserted level, reset
 - setting 15-59
- asynchronous resets
 - setting from command line 15-62

B

- Bit Concat block 7-35
- Bit Reduce block 7-35
- Bit Rotate block 7-35
- Bit Shift block 7-35
- Bit Slice block 7-35
- bit-true cycle-accurate models
 - bit-true to generated HDL code 8-2
- Bitwise Operator blocks 7-35
- block implementations
 - Constant 6-22
 - defined 5-3
 - Divide 6-22
 - Gain 6-22
 - Lookup Table 6-22
 - Math Function 6-22
 - Maximum 6-22
 - Minimum 6-22
 - MinMax 6-22
 - multiple 6-22
 - parameters for 6-41
 - Product of Elements 6-22
 - restrictions on use of 6-35
 - special purpose 6-22
 - specifying in control file 5-24
 - Subsystem 6-22
 - Sum of Elements 6-22
 - summary of 6-2

- block labels
 - for GENERATE statements 15-2
 - for output assignment blocks 15-53
 - specifying postfix for 15-2
- BlockGenerateLabel property 15-2
- blocks
 - restrictions on use in test bench 6-40
 - supporting complex data type 6-56
- blockscope 5-7

C

- CastBeforeSum property 15-3
- checkhdl function 17-2
- CheckHDL property 15-4
- clock
 - specifying high time for 15-7
 - specifying low time for 15-9
- clock enable input port
 - specifying forced signals for 15-18
- clock input port 15-8
 - specifying forced 15-17
- clock process names
 - specifying postfix for 15-10
- clock time
 - high 15-7
 - low 15-9
- ClockEnableInputPort property 15-5
- ClockEnableOutputPort property 15-6
- ClockHighTime property 15-7
- ClockInputPort property 15-8
- ClockLowTime property 15-9
- ClockProcessPostfix property 15-10
- code generation control files. *See* control files
- code generation report 9-2
- code, generated
 - advanced properties for customizing 14-6
- CodeGenerationOutput property 15-11

- comments, header
 - as property value 15-78
- complex data type
 - blocks supporting 6-56
- complex signals
 - in Embedded MATLAB Function block 12-49
- ComplexImagPostfix property 15-12
- ComplexRealPostfix property 15-13
- configuration parameters
 - EDA Tool Scripts pane 3-76
 - Compile command for Verilog 3-81
 - Compile command for VHDL 3-80
 - Compile file postfix 3-78
 - Compile Initialization 3-79
 - Compile termination 3-82
 - Generate EDA scripts 3-77
 - Simulation command 3-85
 - Simulation file postfix 3-83
 - Simulation initialization 3-84
 - Simulation termination 3-87
 - Simulation waveform viewing
 - command 3-86
 - Synthesis command 3-90
 - Synthesis file postfix 3-88
 - Synthesis initialization 3-89
 - Synthesis termination 3-91
- Global Settings pane 3-16
 - Cast before sum 3-44
 - Clock enable input port 3-20
 - Clock enable output port 3-40
 - Clock input port 3-19
 - Clocked process postfix 3-32
 - Comment in header 3-22
 - Complex imaginary part postfix 3-36
 - Complex real part postfix 3-35
 - Concatenate type safe zeros 3-47
 - Enable prefix 3-33
 - Entity conflict postfix 3-25
 - Inline VHDL configuration 3-46
 - Input data type 3-37
 - Loop unrolling 3-43
 - Optimize timing controller 3-48
 - Output data type 3-38
 - Package postfix 3-26
 - Pipeline postfix 3-34
 - Represent constant values by
 - aggregates 3-41
 - Reserved word postfix 3-27
 - Reset asserted level 3-18
 - Reset input port 3-21
 - Reset type 3-17
 - Split arch file postfix 3-31
 - Split entity and architecture 3-28
 - Split entity file postfix 3-30
 - Use "rising_edge" for registers 3-42
 - Use Verilog `timescale directives 3-45
 - Verilog file extension 3-23
 - VHDL file extension 3-24
- HDL Coder pane 3-7
 - Directory 3-11
 - Generate HDL for: 3-9
 - Generate traceability report 3-13
 - Language 3-10
- pane
- Test Bench pane 3-52
 - Clock enable delay 3-60
 - Clock high time (ns) 3-55
 - Clock low time (ns) 3-56
 - Force clock 3-54
 - Force clock enable 3-59
 - Force reset 3-62
 - Generate cosimulation blocks 3-73
 - Hold input data between samples 3-65
 - Hold time (ns) 3-57
 - Ignore output data checking (number of samples) 3-71
 - Initialize test bench inputs 3-66
 - Multi-file test bench 3-67
 - Reset length 3-63

- Setup time (ns) 3-58
- Test bench data file name postfix 3-70
- Test bench name postfix 3-53
- Configuration Parameters dialog box
 - HDL Coder options in 3-2
- configurations, inline
 - suppressing from command line 15-46
- constants
 - setting representation from command line 15-77
- control files
 - attaching to model 5-19
 - control object method calls in 5-7
 - forAll 5-12
 - forEach 5-7
 - generateHDLFor 5-13
 - hdlnewcontrol 5-7 17-11 17-14
 - hdlnewcontrolfile 5-14
 - set 5-12
 - creation of 5-15
 - demo for 5-4
 - detaching to model 5-22
 - loading 5-19
 - objects instantiated in 5-7
 - portability of 5-19
 - purpose of 5-2
 - required elements for 5-5
 - saving 5-15
 - selecting block implementations in 5-3
 - specifying implementation mappings in 5-4
 - statement types in
 - property setting 5-2
 - selection/action 5-2

D

- data input port
 - specifying hold time for 15-42
- demos 1-10
- directory, target 15-70

- Dual Port RAM block 7-4

E

- EDAScriptGeneration property 15-14
- electronic design automation (EDA) tools
 - generation of scripts for
 - customized 13-4
 - defaults for 13-3
 - overview of 13-2
- Embedded MATLAB Function block
 - Distributed pipeline insertion 12-58
 - DistributedPipelining parameter
 - for 12-58
 - HDL code generation for 12-2
 - language support 12-72
 - limitations 12-81
 - setting fixed point options 12-10
 - tutorial example 12-4
 - OutputPipeline parameter for 12-58
 - recommended settings for HDL code generation 12-68
 - speed optimization for 12-58
 - use of complex signals with 12-49
- Embedded MATLAB Function Block
 - design patterns in 12-25
- EnablePrefix property 15-15
- entities
 - setting postfix from command line 15-69
- entity name conflicts 15-16
- EntityConflictPostfix property 15-16

F

- field programmable gate arrays (FPGAs) 1-2
- file extensions
 - Verilog 15-84
 - VHDL 15-85
- file location properties 14-2
- file names

- for architectures 15-67
- for entities 15-69
- file naming properties 14-2
- files, generated
 - splitting 15-68
- force reset hold time 15-42
- ForceClock property 15-17
- ForceClockEnable property 15-18
- ForceReset property 15-19
- FPGAs (field programmable gate arrays) 1-2
- functions
 - checkhdl 17-2
 - hdlldlib 17-6
 - hdlnewblackbox 17-7
 - hdlnewcontrolfile 17-12
 - hdlnewforeach 17-14
 - hdlsetup 17-19
 - makehdl 17-21
 - makehdltb 17-28

G

- GenerateCoSimBlock property 15-20
- generated models
 - bit-true to generated HDL code 8-2
 - cycle-accuracy of 8-2
 - default options for 8-12
 - example of numeric differences 8-4
 - GUI options for 8-13
 - highlighted blocks in 8-12
 - latency example 8-8
 - makehdl properties for 8-14
 - naming conventions for 8-12
 - options for 8-12
- GeneratedmodelName property 15-21
- Generatedmodelnameprefix property 15-22

H

- hardware description languages (HDLs) 1-2

See also Verilog; VHDL

- HDL Coder menu 3-5
- HDL Coder options
 - in Configuration Parameters dialog box 3-2
 - in Model Explorer 3-3
 - in Tools menu 3-5
- HDLCompileFilePostfix property 15-25
- HDLCompileInit property 15-23
- HDLCompileTerm property 15-24
- HDLCompileVerilogCmd property 15-26
- HDLCompileVHDLCmd property 15-27
- HDLControlFiles property 15-28
- hdlldlib function 17-6
- HDLMapPostfix property 15-29
- hdlnewblackbox function 17-7
- hdlnewcontrolfile function 17-12
- hdlnewforeach function 17-14
 - example 5-24
 - generating forEach calls with 5-24
- HDLs (hardware description languages) 1-2
 - See also* Verilog; VHDL
- hdlsetup function 17-19
- HDLSimCmd property 15-30
- HDLSimFilePostfix property 15-32
- HDLSimInit property 15-31
- HDLSimTerm property 15-33
- HDLSimViewWaveCmd property 15-34
- HDLSynthCmd property 15-35
- HDLSynthFilePostfix property 15-37
- HDLSynthInit property 15-36
- HDLSynthTerm property 15-38
- header comment properties 14-3
- Highlightancestors property 15-39
- Highlightcolor property 15-40
- hold time 15-42
- HoldInputDataBetweenSamples time 15-41
- HoldTime property 15-42
- HTML code generation report 9-2

I

- IgnoreDataChecking property 15-44
- implementation mapping
 - defined 5-4
- InitializeTestBenchInputs property 15-45
- inline configurations
 - specifying 15-46
- InlineConfigurations property 15-46
- input ports
 - specifying data type for 15-47
- InputType property 15-47
- instance sections 15-48
- InstanceGenerateLabel property 15-48
- InstancePrefix property 15-49
- Interfaces, generation of
 - black box 10-3
 - for Dual Port RAM block 7-4
 - for HDL Cosimulation blocks 10-13
 - for referenced models 10-10
 - for simple Dual Port RAM block 7-4
 - for Single Port RAM block 7-4

L

- labels
 - block 15-53
- language
 - target 15-71
- language selection properties 14-2 14-9
- loops
 - unrolling 15-50
- LoopUnrolling property 15-50

M

- makehdl function 17-21
- makehdltb function 17-28
- Model Explorer
 - HDL Coder options in 3-3
- modelscope 5-7

- MultifileTestBench property 15-51

N

- name conflicts 15-16
- names
 - clock process 15-10
 - package file 15-55
- naming properties 14-3
- No-op block implementations 10-17

O

- online help 1-10
- OptimizeTimingController property 15-52
- output ports
 - specifying data type for 15-54
- OutputGenerateLabel property 15-53
- OutputType property 15-54

P

- package files
 - specifying postfix for 15-55
- PackagePostfix property 15-55
- Pass-through block implementations 10-17
- PipelinePostfix property 15-56
- port properties 14-5
- ports
 - clock enable input 15-5
 - clock input 15-8
 - input 15-47
 - output 15-54
 - reset input 15-60
- properties
 - advanced coding 14-6
 - BlockGenerateLabel 15-2
 - CastBeforeSum 15-3
 - CheckHDL 15-4
 - ClockEnableInputPort 15-5
 - ClockEnableOutputPort 15-6

ClockHighTime 15-7
ClockInputPort 15-8
ClockLowTime 15-9
ClockProcessPostfix 15-10
CodeGenerationOutput 15-11
coding 14-6
ComplexImagPostfix 15-12
ComplexRealPostfix 15-13
EDAScriptGeneration 15-14
EnablePrefix 15-15
EntityConflictPostfix 15-16
file location 14-2
file naming 14-2
ForceClock 15-17
ForceClockEnable 15-18
ForceReset 15-19
GenerateCoSimBlock 15-20
generated models 14-9
GeneratedmodelName 15-21
Generatedmodelnameprefix 15-22
HDLCompileFilePostfix 15-25
HDLCompileInit 15-23
HDLCompileTerm 15-24
HDLCompileVerilogCmd 15-26
HDLCompileVHDLCmd 15-27
HDLControlFiles 15-28
HDLMapPostfix 15-29
HDLSimCmd 15-30
HDLSimFilePostfix 15-32
HDLSimInit 15-31
HDLSimTerm 15-33
HDLSimViewWaveCmd 15-34
HDLSynthCmd 15-35
HDLSynthFilePostfix 15-37
HDLSynthInit 15-36
HDLSynthTerm 15-38
header comment 14-3
Highlightancestors 15-39
Highlightcolor 15-40
HoldInputDataBetweenSamples 15-41
HoldTime 15-42
IgnoreDataChecking 15-44
InitializeTestBenchInputs 15-45
InlineConfigurations 15-46
InputType 15-47
InstanceGenerateLabel 15-48
InstancePrefix 15-49
language selection 14-2
LoopUnrolling 15-50
MultifileTestBench 15-51
naming 14-3
OptimizeTimingController 15-52
OutputGenerateLabel 15-53
OutputType 15-54
PackagePostfix 15-55
PipelinePostfix 15-56
port 14-5
ReservedWordPostfix 15-58
reset 14-2
ResetAssertedLevel 15-59
ResetInputPort 15-60
ResetLength 15-61
ResetType 15-62
ResetValue 15-64
SafeZeroConcat 15-65
script generation 14-4
SimulatorFlags 15-66
SplitArchFilePostfix 15-67
SplitEntityArch 15-68
SplitEntityFilePostfix 15-69
TargetDirectory 15-70
TargetLanguage 15-71
test bench 14-7
TestBenchClockEnableDelay 15-72
TestBenchDataPostFix 15-73
TestBenchPostfix 15-74
TestBenchReferencePostFix 15-75
Traceability 15-76
UseAggregatesForConst 15-77
UserComment 15-78

- UseRisingEdge 15-79
- UseVerilogTimescale 15-81
- VectorPrefix 15-82
- Verbosity 15-83
- VerilogFileExtension 15-84
- VHDLFileExtension 15-85

R

RAM

- blocks 7-4
- inferring 7-4

requirements

- product 1-8

reserved words

- specifying postfix for 15-58

ReservedWordPostfix property 15-58

reset input port 15-60

reset properties 14-2

ResetAssertedLevel property 15-59

ResetInputPort property 15-60

ResetLength property 15-61

resets

- setting asserted level for 15-59
- specifying forced 15-19
- types of 15-62

ResetType property 15-62

ResetValue property 15-64

restoring factory default options 5-22

S

SafeZeroConcat property 15-65

script generation properties 14-4

sections

- instance 15-48

Simple Dual Port RAM block 7-4

SimulatorFlags property 15-66

Single Port RAM block 7-4

SplitArchFilePostfix property 15-67

SplitEntityArch property 15-68

SplitEntityFilePostfix property 15-69

Stateflow charts

- code generation 11-2

- requirements for 11-4

- restrictions on 11-4

subtraction operations

- typecasting 15-3

synchronous resets

- setting from command line 15-62

T

TargetDirectory property 15-70

TargetLanguage property 15-71

test bench properties 14-7

test benches

- specifying clock enable input for 15-18

- specifying forced clock input for 15-17

- specifying forced resets for 15-19

TestBenchClockEnabledDelay property 15-72

TestBenchDataPostFix property 15-73

TestBenchPostfix property 15-74

TestBenchReferencePostFix property 15-75

time

- clock high 15-7

- clock low 15-9

- hold 15-42

timescale directives

- specifying use of 15-81

Traceability property 15-76

typecasting 15-3

U

UseAggregatesForConst property 15-77

UserComment property 15-78

UseRisingEdge property 15-79

UseVerilogTimescale property 15-81

V

VectorPrefix property 15-82
Verbosity property 15-83
Verilog 1-2
 file extension 15-84
VerilogFileExtension property 15-84
VHDL 1-2

 file extension 15-85

VHDLFileExtension property 15-85

Z

zeros, concatenated 15-65