

HDL Verifier™

User's Guide



MATLAB® & SIMULINK®

R2022b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

HDL Verifier™ User's Guide

© COPYRIGHT 2003–2022 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

Screenshots of ModelSim® in the HDL Verifier™ documentation are copyright protected by Mentor Graphics Corporation.

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

Patents

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

Revision History

August 2003	Online only	New for Version 1 (Release 13SP1)
February 2004	Online only	Revised for Version 1.1 (Release 13SP1)
June 2004	Online only	Revised for Version 1.1.1 (Release 14)
October 2004	Online only	Revised for Version 1.2 (Release 14SP1)
December 2004	Online only	Revised for Version 1.3 (Release 14SP1+)
March 2005	Online only	Revised for Version 1.3.1 (Release 14SP2)
September 2005	Online only	Revised for Version 1.4 (Release 14SP3)
March 2006	Online only	Revised for Version 2.0 (Release 2006a)
September 2006	Online only	Revised for Version 2.1 (Release 2006b)
March 2007	Online only	Revised for Version 2.2 (Release 2007a)
September 2007	Online only	Revised for Version 2.3 (Release 2007b)
March 2008	Online only	Revised for Version 2.4 (Release 2008a)
October 2008	Online only	Revised for Version 2.5 (Release 2008b)
March 2009	Online only	Revised for Version 2.6 (Release 2009a)
September 2009	Online only	Revised for Version 3.0 (Release 2009b)
March 2010	Online only	Revised for Version 3.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.2 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 3.4 (Release 2011b)
March 2012	Online only	Revised for Version 4.0 (Release 2012a)
September 2012	Online only	Revised for Version 4.1 (Release 2012b)
March 2013	Online only	Revised for Version 4.2 (Release 2013a)
September 2013	Online only	Revised for Version 4.3 (Release 2013b)
March 2014	Online only	Revised for Version 4.4 (Release 2014a)
October 2014	Online only	Revised for Version 4.5 (Release 2014b)
September 2015	Online only	Revised for Version 4.7 (Release 2015b)
March 2016	Online only	Revised for Version 5.0 (Release 2016a)
September 2016	Online only	Revised for Version 5.1 (Release 2016b)
March 2017	Online only	Revised for Version 5.2 (Release 2017a)
September 2017	Online only	Revised for Version 5.3 (Release 2017b)
March 2018	Online only	Revised for Version 5.4 (Release 2018a)
September 2018	Online only	Revised for Version 5.5 (Release 2018b)
March 2019	Online only	Revised for Version 5.6 (Release 2019a)
September 2019	Online only	Revised for Version 6.0 (Release 2019b)
March 2020	Online only	Revised for Version 6.1 (Release 2020a)
September 2020	Online only	Revised for Version 6.2 (Release 2020b)
March 2021	Online only	Revised for Version 6.3 (Release 2021a)
September 2021	Online only	Revised for Version 6.4 (Release 2021b)
March 2022	Online only	Revised for Version 6.5 (Release 2022a)
September 2022	Online only	Revised for Version 7.0 (Release 2022b)

HDL Verification with Cosimulation

Setup and Run Cosimulation for MATLAB

1

Set Up MATLAB-HDL Simulator Connection	1-2
Start MATLAB Server	1-2
Start HDL Simulator	1-2
Load an HDL Design for Verification	1-3
Run MATLAB-HDL Cosimulation	1-4
Process for Running MATLAB Cosimulation	1-4
Check MATLAB Cosimulation Server's Link Status	1-4
Run Cosimulation	1-4
Apply Stimuli to Cosimulation Session with force Command	1-7
Restart Simulation	1-8
Stop Simulation	1-8

HDL Cosimulation Using MATLAB Test Bench Function

2

Create a MATLAB Test Bench	2-2
Write HDL Modules for MATLAB Test Bench	2-3
Write a Test Bench Function	2-6
Set Up Cosimulation Test Bench	2-13
Place Test Bench onve MATLAB Search Path	2-13
Bind Test Bench Function Calls With matlabbt	2-13
Schedule Options for a Test Bench Session	2-15
Verify HDL Module with MATLAB Test Bench	2-18
Tutorial Overview	2-18
Set Up Tutorial Files	2-18
Start the MATLAB Server	2-19
Start ModelSim Simulator and Set Up for Cosimulation	2-20
Develop VHDL Code	2-21
Compile VHDL Code	2-23
Develop MATLAB Function	2-23
Load Simulation	2-24
Run Simulation	2-26
Shut Down Simulation	2-29

3 HDL Cosimulation Using MATLAB Component Function

Create a MATLAB Component Function	3-2
Write HDL Modules for MATLAB Visualization	3-3
Write a Component Function	3-6
Set Up Cosimulation Component	3-8
Place Component Function on MATLAB Search Path	3-8
Bind Component Function Calls With matlabcp	3-8
Schedule Options for a Component Session	3-11

4 HDL Cosimulation Using MATLAB System Object

Create a MATLAB System Object	4-2
--	------------

5 Set Up and Run Simulation for Simulink

Start HDL Simulator for Cosimulation in Simulink	5-2
Start HDL Simulator from MATLAB	5-2
Load Instance of HDL Module for Cosimulation	5-2
Run a Simulink Cosimulation Session	5-4
Set Simulink Model Configuration Parameters	5-4
Determine Available Socket Port Number	5-4
Check Connection Status	5-4
Run and Test Cosimulation Model	5-5
Set Parameters from a Tcl Script	5-7
Avoid Race Conditions in HDL Simulation with Test Bench	
Cosimulation and the HDL Verifier HDL Cosimulation Block	5-8

6 Simulink Test Bench for HDL Component

Simulink as a Test Bench	6-2
Communications During Test Bench Cosimulation	6-2
HDL Cosimulation Block Features for Test Bench Simulation	6-4

Create a Simulink Cosimulation Test Bench	6-6
Code an HDL Component	6-6
Configure HDL Cosimulation Block Interface	6-8
Verify HDL Module with Simulink Test Bench	6-27
Tutorial Overview	6-27
Develop VHDL Code	6-27
Compile VHDL Code	6-28
Create Simulink Model	6-29
Set Up ModelSim for Use with Simulink	6-37
Load Instances of VHDL Entity for Cosimulation with Simulink	6-37
Run Simulation	6-38
Shut Down Simulation	6-40
Automatic Verification of Generated HDL Code from Simulink ...	6-42

Replace HDL Component with Simulink Algorithm

7

Component Simulation with Simulink	7-2
How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Component Simulation	7-2
HDL Cosimulation Block Features for Component Simulation	7-3
Create Simulink Model for Component Cosimulation	7-5
Code an HDL Component	7-5
Define HDL Cosimulation Block Interface	7-7

Record Simulink Signal State Transitions for Post-Processing

8

Add a Value Change Dump (VCD) File	8-2
Introduction to the To VCD File Block	8-2
Using the To VCD File Block	8-2
Visually Compare Simulink Signals with HDL Signals	8-5
Tutorial: Overview	8-5
Tutorial: Instructions	8-5

HDL Code Import for Cosimulation

9

Prepare to Import HDL Code for Cosimulation	9-2
HDL Code Import Features	9-2
HDL Code Import Workflows	9-3

Cosimulation Wizard Navigation	9-3
Cosimulation Wizard Limitations	9-3
Import HDL Code for MATLAB Function	9-4
Cosimulation Type—MATLAB Function	9-4
HDL Files—MATLAB Function	9-5
HDL Compilation—MATLAB Function	9-6
HDL Modules—MATLAB Function	9-7
Callback Schedule—MATLAB Function	9-8
Script Generation—MATLAB Function	9-9
Complete the Component or Test Bench Function	9-10
Import HDL Code for MATLAB System Object	9-12
Cosimulation Type—MATLAB System Object	9-12
HDL Files—MATLAB System Object	9-13
HDL Compilation—MATLAB System Object	9-14
Simulation Options—MATLAB System Object	9-15
Input/Output Ports—MATLAB System Object	9-17
Output Port Details—MATLAB System Object	9-18
Clock/Reset Details—MATLAB System Object	9-19
Start Time Alignment—MATLAB System Object	9-20
System Object Generation	9-21
Write System Object Test Bench	9-21
Run Cosimulation and Verify HDL Design	9-22
Import HDL Code for HDL Cosimulation Block	9-24
Cosimulation Type—Simulink Block	9-24
HDL Files—Simulink Block	9-25
HDL Compilation—Simulink Block	9-26
Simulation Options—Simulink Block	9-26
Input/Output Ports—Simulink Block	9-29
Output Port Details—Simulink Block	9-30
Clock/Reset Details—Simulink Block	9-31
Start Time Alignment—Simulink Block	9-32
Generate Block	9-33
Complete Simulink Model	9-33
Use HDL Parameters in Cosimulation	9-35
Supported Data Types	9-35
Performing Cosimulation	9-37
Verify Raised Cosine Filter Design Using MATLAB	9-38
MATLAB and Cosimulation Wizard Tutorial Overview	9-38
Tutorial: Set Up Tutorial Files (MATLAB)	9-39
Tutorial: Launch Cosimulation Wizard (MATLAB)	9-39
Tutorial: Configure the Component Function with the Cosimulation Wizard	9-40
Tutorial: Customize Callback Function	9-46
Tutorial: Run Cosimulation and Verify HDL Design	9-49
Verify Raised Cosine Filter Design Using Simulink	9-51
Simulink and Cosimulation Wizard Tutorial Overview	9-51
Tutorial: Set Up Tutorial Files (Simulink)	9-51
Tutorial: Launch Cosimulation Wizard (Simulink)	9-52

Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard	9-52
Tutorial: Create Test Bench to Verify HDL Design	9-61
Tutorial: Run Cosimulation and Verify HDL Design	9-63
Help Button	9-66
Cosimulation Type	9-66
HDL Files	9-67
HDL Compilation	9-68
HDL Modules	9-69

HDL Cosimulation Guide

10

Set Up for HDL Cosimulation	10-2
Cosimulation Configurations	10-2
HDL Simulator Startup	10-5
Cosimulation Libraries	10-9
Cross-Network Cosimulation	10-14
Why Perform Cross-Network Cosimulation?	10-14
Preparing for Cross-Network Cosimulation	10-14
Performing Cross-Network Cosimulation Using MATLAB	10-15
Performing Cross-Network Cosimulation Using Simulink	10-16
Test Bench and Component Function Writing	10-19
Writing Functions Using the HDL Instance Object	10-19
Writing Functions Using Port Information	10-22
Simulation Speed Improvement Tips	10-26
Obtaining Baseline Performance Numbers	10-26
Analyzing Simulation Performance	10-26
Cosimulating Frame-Based Signals with Simulink	10-27
Race Conditions in HDL Simulators	10-33
Avoiding Race Conditions	10-33
Potential Race Conditions in Simulink Cosimulation Sessions	10-33
Potential Race Conditions in MATLAB Cosimulation Sessions	10-34
Further Reading	10-34
Supported Data Types	10-35
Converting HDL Data to Send to MATLAB or Simulink	10-35
Bit-Vector Indexing Differences Between MATLAB and HDL	10-37
Array Indexing Differences Between MATLAB and HDL	10-38
Converting Data for Manipulation	10-39
Converting Data for Return to the HDL Simulator	10-40
Simulation Timescales	10-44
Overview to the Representation of Simulation Time	10-44
Defining the Simulink and HDL Simulator Timing Relationship	10-45
Setting the Timing Mode with HDL Verifier	10-45
Specify Timing Relationship Automatically	10-46

Relative Timing Mode	10-48
Absolute Timing Mode	10-51
Timing Mode Usage Considerations	10-53
Setting HDL Cosimulation Block Port Sample Times	10-54
Clock, Reset, and Enable Signals	10-55
Driving Clocks, Resets, and Enables	10-55
Adding Signals Using Simulink Blocks	10-55
Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block	10-56
Driving Signals by Adding Force Commands	10-58
TCP/IP Socket Ports	10-61

FPGA-in-the-Loop

About FPGA-in-the-Loop Simulation

11

FPGA-in-the-Loop Simulation	11-2
What is FPGA-in-the-Loop Simulation?	11-2
What You Need To Know	11-4

Prepare FPGA Connection

12

FPGA-in-the-Loop Simulation Workflows	12-2
Download FPGA Board Support Package	12-3
HDL Verifier Support Packages for FPGA Boards	12-3
Install with Connection to Internet	12-3
Install Support Package Offline	12-3
Set Up FPGA Design Software Tools	12-4
Xilinx Software	12-4
Intel Software	12-4
Microchip Software	12-4
Guided Hardware Setup	12-5
Select Board and Interface for Use with FPGA Verification	12-5
Connection Requirements	12-5
Connection Setup	12-6
Configure Network Card on Host	12-9
Select a Drive and Load Firmware	12-9
PCI Express Driver Installation	12-10
Set Jumpers	12-10

Connect Hardware	12-11
Verify Setup	12-11
Open the Example	12-12
Manual Hardware Setup	12-13
Step 1. Set Up FPGA Development Board	12-13
Step 2. Set Up Board Connection	12-14
Prepare DUT For FIL Interface Generation	12-18
Files and Information Required for FIL Generation	12-18
Apply FIL System Object Requirements	12-18
Apply FIL Block Requirements	12-21

FIL Interface Generation and Simulation

13

Block Generation with the FIL Wizard	13-2
Step 1: Set Up FPGA Design Software Tools	13-2
Step 2: Start FIL Wizard	13-3
Step 3: Set FIL Options for FIL Block	13-3
Step 4: Add HDL Source Files for FIL Block	13-5
Step 5: Verify DUT I/O Ports for FIL Block	13-6
Step 6: Specify Output Types for FIL Block	13-7
Step 7: Specify Build Options for FIL Block	13-8
Step 8: Initiate Build	13-8
Step 9: Integrate and Simulate	13-9
System Object Generation with the FIL Wizard	13-12
Step 1: Set Up FPGA Design Software Tools	13-12
Step 2: Start FIL Wizard	13-13
Step 3: Set FIL Options for System Object	13-13
Step 4: Add HDL Source Files for System Object	13-15
Step 5: Verify DUT I/O Ports for System Object	13-16
Step 6: Specify Output Types for System Object	13-17
Step 7: Specify Build Options for System Object	13-18
Step 8: Initiate Build	13-18
Step 9: Integrate and Simulate	13-20

FIL Using HDL Coder HDL Workflow Advisor

14

FIL Simulation with HDL Workflow Advisor for Simulink	14-2
Step 1: Start HDL Workflow Advisor	14-2
Step 2: Set Target and Target Frequency	14-2
Step 3: Prepare Model for HDL Code Generation	14-3
Step 4: HDL Code Generation	14-3
Step 5: Set FPGA-in-the-Loop Options	14-3
Step 6: Generate FPGA Programming File and FPGA-in-the-Loop Model	14-5

Step 7: Load Programming File onto FPGA	14-6
Step 8: Run Simulation	14-7
FIL Simulation with HDL Workflow Advisor for MATLAB	14-8
Step 1: Start HDL Workflow Advisor	14-8
Step 2: Select Target	14-8
Step 3: Select Workflow	14-8
Step 4: Select FPGA-in-the-Loop Options	14-8
Step 5: Generate FPGA Programming File and Run Simulation ...	14-12

Troubleshooting FPGA-in-the-Loop

15

Troubleshooting FIL	15-2
----------------------------------	-------------

FPGA Board Customization

16

FPGA Board Customization	16-2
Feature Description	16-2
Custom Board Management	16-2
FPGA Board Requirements	16-2
Create Custom FPGA Board Definition	16-6
Create Xilinx KC705 Evaluation Board Definition File	16-7
Overview	16-7
What You Need to Know Before Starting	16-7
Start New FPGA Board Wizard	16-7
Provide Basic Board Information	16-8
Specify FPGA Interface Information	16-9
Enter FPGA Pin Numbers	16-10
Run Optional Validation Tests	16-12
Save Board Definition File	16-13
Use New FPGA Board	16-14
FPGA Board Manager	16-18
Introduction	16-18
Filter	16-19
Search	16-19
FIL Enabled/Turnkey Enabled	16-20
Create Custom Board	16-20
Add Board from File	16-20
Get More Boards	16-20
View/Edit	16-20
Remove	16-20
Clone	16-20
Validate	16-20

New FPGA Board Wizard	16-22
Basic Information	16-23
Interfaces	16-23
FIL I/O	16-26
Turnkey I/O	16-28
Validation	16-31
Finish	16-32
FPGA Board Editor	16-33
General Tab	16-33
Interface Tab	16-35

FPGA Data Capture

17

Data Capture Workflow	17-2
Generate and Integrate Data Capture IP Using HDL Workflow Advisor	17-3
Configure and Generate IP Core for an Existing HDL Design	17-3
Integrate IP into FPGA	17-4
Capture Data	17-5

AXI Manager

18

Set Up for AXI Manager	18-2
JTAG Considerations	18-3

SystemC TLM Generation

How TLM Component Generation Works

19

TLM Generation Algorithms	19-2
The TLM Generation Process	19-3
Generated TLM Files	19-5

20

TLM Component Architecture	20-2
Overview of Component Features	20-2
Memory Mapping	20-3
Command and Status Register	20-7
Interrupt	20-10
Test and Set Register	20-11
Registers and Signal Ports	20-11

Getting Started with TLM Component Generation

21

Get Started with TLM Generator	21-2
---	-------------

Generate TLM Component

22

TLM Component Generation Workflow	22-2
Subsystem Guidelines and Limitations	22-3
Select TLM Generator System Target	22-4
Select TLM Mapping Options	22-6
Socket Mapping	22-6
Memory Map Configuration	22-7
Select TLM Processing Options	22-8
Algorithm Processing	22-8
Interface Processing	22-8
Select TLM Timing Options	22-9
Select TLM Test Bench Options	22-10
Select TLM Compilation Options	22-12
Generate Component and Test Bench	22-15
Prepare IP-XACT File for Import	22-16
Required Information for Imported IP-XACT Files	22-16
Bus Interface Definition with No Memory Map	22-17
Bus Interface Definition with Memory Mapping	22-18
Mapping to a Signal Port	22-21

Contents of Generated IP-XACT File	22-24
Overview of Generated IP-XACT File	22-24
Generated Simulink Mapping	22-24
Generated Simulink Mapping in Memory Map	22-25
Generated Metadata	22-26
Implement Memory Map with SCML	22-28
What Is SCML?	22-28
Workflow	22-28
Generated Code	22-28

Run TLM Component Test Bench

23

Testing TLM Components	23-2
TLM Component Test Bench Overview	23-2
TLM Component Compilation	23-2
Automatic Verification of the Generated Component	23-2
Report Generation	23-3
Working with Configurations	23-3
Considerations When Creating a TLM Component Test Bench	23-3
Run TLM Component Test Bench	23-5

Export TLM Component to SystemC Environment

24

Export TLM Component	24-2
Identify Generated Files	24-2
Create Static Library with TLM Component	24-3
Create Standalone Executable with TLM Component	24-4
TLM Component Constructor	24-6

Configuration Parameters for TLM Generator Target

25

TLM Component Generation	25-2
TLM Mapping	25-2
TLM Processing	25-7
TLM Timing	25-8
TLM Testbench	25-9
TLM Compilation	25-13

UVM Component Generation Overview	26-2
UVM Component Generation Overview	26-2
Prepare Simulink Model for UVM Component Generation	26-2
Generated UVM Structure	26-4
Generated Files and Folder Structure	26-6
Supported Simulink Data Types	26-8
Limitations	26-10
Customize Generated UVM Code	26-12
Customize SystemVerilog File Banner	26-12
Customize HDL Simulation Timescale	26-13
Use Tunable Parameters to Generalize UVM Simulation	26-14
Tunable Parameters in Sequence Subsystem	26-15
Prepare Sequence for UVM Generation With Tunable Parameters	26-15
Generate UVM Sequence	26-15
Control Sequence Parameters in UVM Simulation	26-16
Tunable Parameters in Scoreboard Subsystem	26-17
Prepare Scoreboard for UVM Generation with Tunable Parameters	26-17
Generate UVM Scoreboard	26-17
Control Scoreboard Parameters in UVM Simulation	26-18

SystemVerilog DPI Component Generation

DPI Component Generation for MATLAB Function

Considerations for DPI Component Generation with MATLAB	27-2
Supported MATLAB Data Types	27-2
Generated Shared Library	27-4
Generated Test Bench	27-4
Generated Outputs	27-5
Generated SystemVerilog Wrapper	27-5
Simulation Considerations	27-5
Limitations	27-6
Use Variable-Sized Vector in SystemVerilog DPI Component	27-7

28

Generate DPI Component Using MATLAB	28-2
Create MATLAB Function and Test Bench	28-2
Generate SystemVerilog DPI Component	28-3
Run Generated Test Bench in HDL Simulator	28-7
Use Generated DPI Functions in SystemVerilog	28-9
Port Generated Component and Test Bench to Linux	28-10

DPI Component Generation for Simulink Subsystem

29

DPI Component Generation with Simulink	29-2
DPI Generation Overview	29-2
Supported Simulink Data Types	29-2
Generated SystemVerilog Wrapper	29-5
SystemVerilog Wrapper for Combinational Design	29-6
Generated Component Functions	29-7
Parameter Tuning	29-8
Test Point Access Functions	29-9
Extra Sample Delay	29-9
Multirate System Behavior	29-9
Customization	29-10
Limitations	29-11
SystemVerilog DPI Test Benches	29-12
Component Test Bench	29-12
HDL Code Test Bench	29-14

SystemVerilog DPI Component Generation for Simulink

30

Generate SystemVerilog DPI Component	30-2
Step 1. Select Target	30-2
Step 2. Select Toolchain	30-2
Step 3. Enable Test Point Access (Optional)	30-2
Step 4. Configure SystemVerilog Generation Options	30-3
Step 5. Generate SystemVerilog DPI Component	30-4
Customize Generated SystemVerilog Code	30-6
Set Up Model for Customized Code Generation	30-6
Generate Customized SystemVerilog DPI Component	30-7
Verify Generated Component Against Simulink Data	30-8
For Mentor Graphics ModelSim and Questa Simulators	30-8
For the Cadence Xcelium Simulator	30-8

Use Generated DPI Functions in SystemVerilog	30-9
Example	30-9
SystemVerilog DPI Component Test Point Access	30-10
Step 1. Choose Internal Signals	30-10
Step 2. Add Test Points	30-10
Step 3. Enable Component Interface	30-10
Step 4. Configure Access Function	30-11
Tune Gain Parameter During Simulation	30-12
Step 1. Create a Simple Gain Model	30-12
Step 2. Create Data Object for Gain Parameter	30-12
Step 3. Generate SystemVerilog DPI Component	30-14
Step 4. Add Parameter Tuning Code to SystemVerilog File	30-14
Step 5. Run Simulation with Parameter Change	30-15
Generate SystemVerilog Assertions from Simulink Test Bench ..	30-16
Generate Assertions Workflow	30-16
Trace Generated SystemVerilog Assertions	30-19
Disabling Assertions	30-20
Generate SystemVerilog Assertions and Functional Coverage ...	30-22
Create a Simulink Test Bench Model	30-22
Generate a UVM or SystemVerilog DPI Component	30-26
Run HDL Simulation with the Generated Component	30-26
Filter Assertions and Coverage Reports	30-27
Adjust Functional Coverage Goals	30-27
Verbose Mode	30-27
Trace Generated SystemVerilog Error Back to Simulink Source ..	30-28
Generate Cross-Platform DPI Components	30-30
Select Target Toolchain	30-30
Generate Component	30-30
Package Files for Transfer	30-31
Copy to Target Machine	30-31
Build Libraries	30-31
Limitations	30-32

Context-Sensitive Help for Generated SystemVerilog DPI Component

31

SystemVerilog DPI Pane	31-2
SystemVerilog DPI Overview	31-2
Customize SystemVerilog generated code	31-3
Source file template:	31-3
Report run-time error	31-3
Severity	31-3
Generate test bench	31-4
HDL simulator	31-4
Test point access	31-4
Ports data type	31-5

Connection	31-5
Composite Data Type	31-5
Scalarize matrix and vector ports	31-6
Component template type	31-6

HDL Verifier Examples

32

Comparing HDL and Simulink Code Coverage Using Cosimulation	32-2
Testing a Filter Component in MATLAB	32-8
Frame-based Scrambler Using Communications Toolbox	32-11
Manchester Receiver	32-14
Implement Spectrum Display Component in MATLAB	32-25
Manchester Receiver (Absolute Time Mode)	32-30
Manchester Receiver Using Communications Toolbox	32-33
Cosimulation Wizard for MATLAB System Object	32-37
Tutorial: Cosimulation Wizard for MATLAB Callback Function	32-51
Get Started with Simulink HDL Cosimulation	32-52
Get Started with SystemVerilog DPI Component Generation	32-68
Getting Started with Customizing Generated SystemVerilog Code	32-71
Get Started with MATLAB Based SystemVerilog DPI Generation	32-74
Building HDL Test Bench for QAM Transceiver Model	32-78
Generate FIFO Interface DPI Component for UART Receiver	32-80
Generate Bit Vector and Logic Vector Data Types	32-86
Generate Native SystemVerilog Assertions from Simulink	32-89
Generating Functional Coverage in SystemVerilog from Simulink Test verify Calls	32-96
Verify Viterbi Decoder Using HDL Cosimulation	32-103
Generate Parameterized UVM Test Bench from Simulink	32-106
Replace Behavioral DUT with AXI-Based RTL DUT in UVM Test Bench	32-113

Add Random Constraints to Sequences in UVM Test Bench	32-120
Change Parameters of Scoreboard in UVM Test Bench	32-130
Include Driver and Monitor in UVM Test Bench	32-137
Verify Viterbi Decoder Using MATLAB System Object and HDL Simulator	32-145
Batch Mode Cosimulation of Manchester Receiver	32-149
Sobel Edge Detection Algorithm with Computer Vision Toolbox	32-152
Cosimulation for Testing Filter Component Using MATLAB Test Bench	32-163
Manchester Receiver Using Multiple Cosimulation Blocks	32-169
Relating HDL Clocks and Resets with Simulink Sample Times	32-174
Timescales: Absolute, Relative and Automatic	32-181
Implement Filter Component of Oscillator in MATLAB	32-190
Manchester Receiver Using Mixed Design (Verilog and VHDL)	32-195
Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop	32-198
Cosimulation and FPGA-in-the-Loop in MATLAB-to-HDL Workflow ..	32-214
Verify the Combination of Hand-Written and Generated HDL Code ..	32-223
FPGA-in-the-Loop Simulation Using MATLAB System Object	32-231
Verify Viterbi Decoder Using MATLAB System Object and FPGA-in-the-Loop	32-234
Video Processing Acceleration Using FPGA-in-the-Loop	32-236
Accelerating Communications System Simulation Using FPGA-in-the-Loop	32-240
Algorithm Verification with a FIL Source Block	32-244
Verify Digital Up-Converter Using FPGA-in-the-Loop	32-247
Untimed SystemC/TLM Simulation	32-253
Loosely-Timed SystemC/TLM Simulation	32-259
No Memory Map Option	32-265
Auto-Generated Memory Map with Single Address Option	32-269

Auto-Generated Memory Map with Individual Address Option	32-274
Imported IP-XACT Without Memory Map	32-279
Imported IP-XACT with Memory Map	32-285
Verify 5G Wireless Applications Using SystemVerilog DPI	32-295

HDL Verification with Cosimulation

Setup and Run Cosimulation for MATLAB

Set Up MATLAB-HDL Simulator Connection

Start MATLAB Server

Start a MATLAB server for cosimulation with an HDL simulator. This is not required for Vivado® simulator users, since the simulation runs as a single process with a shared DLL.

Start the MATLAB server as follows:

- 1 Start MATLAB.
- 2 In the MATLAB Command Window, call the `hdldaemon` function with property name/property value pairs that specify whether the HDL Verifier software is to perform the following tasks:
 - Use shared memory or TCP/IP socket communication
 - Return time values in seconds or as 64-bit integers

See `hdldaemon` reference documentation for when and how to specify property name/property value pairs and for more examples of using `hdldaemon`.

The communication mode that you specify (shared memory or TCP/IP sockets) must match what you specify for the communication mode when you initialize the HDL simulator for use with a MATLAB cosimulation session using the `matlabtb` or `matlabcp` function. In addition, if you specify TCP/IP socket mode, the socket port that you specify with `hdldaemon` and `matlabtb` or `matlabcp` must match. See “TCP/IP Socket Ports” on page 10-61 for more information .

The MATLAB server can service multiple simultaneous HDL simulator modules and clients. However, your code must track the I/O associated with each entity or client.

Note You cannot begin an HDL Verifier transaction between MATLAB and the HDL simulator from MATLAB. The MATLAB server simply responds to function call requests that it receives from the HDL simulator.

This command sets up socket communication on port 4449, and specifies a 64-bit time resolution format for the MATLAB function's output ports.

```
hdldaemon('socket',4449,'time','int64')
```

Start HDL Simulator

Start the HDL simulator directly from MATLAB by calling the HDL Verifier function `vsim` or `nclaunch`.

```
>>vsim
```

You can call `vsim` or `nclaunch` with additional parameters; see the reference pages for details.

You must make sure the HDL simulator executables — also called `vsim` (ModelSim®) and `nclaunch` (Cadence® Xcelium™) — are on the system path. See your system documentation for instruction on setting environment variables.

Linux Users Make sure the HDL simulator executable is still on the system path after the shell is launched from MATLAB. If it is not, make sure the shell startup file does not remove it from the path environment variable.

Load an HDL Design for Verification

After you start the HDL simulator from MATLAB with a call to `vsim` or `nclaunch`, load an instance of an HDL module for verification or visualization with the function `vsimmatlab` or `hdlsimmatlab`. At this point, you should have coded and compiled your HDL model. Issue the function `vsimmatlab` or `hdlsimmatlab` for each instance of an entity or module in your model that you want to cosimulate. For example (for use with Xcelium):

```
hdlsimmatlab work.osc_top
```

This command loads the HDL Verifier library, opens a simulation workspace for `osc_top`, and display a series of messages in the HDL simulator command window as the simulator loads the entity (see example for remaining code).

Run MATLAB-HDL Cosimulation

In this section...

“Process for Running MATLAB Cosimulation” on page 1-4

“Check MATLAB Cosimulation Server's Link Status” on page 1-4

“Run Cosimulation” on page 1-4

“Apply Stimuli to Cosimulation Session with force Command” on page 1-7

“Restart Simulation” on page 1-8

“Stop Simulation” on page 1-8

Process for Running MATLAB Cosimulation

To start and control the execution of a simulation in the MATLAB environment, perform the following steps:

- 1 “Check MATLAB Cosimulation Server's Link Status” on page 1-4
- 2 “Run Cosimulation” on page 1-4
- 3 “Apply Stimuli to Cosimulation Session with force Command” on page 1-7
- 4 “Restart Simulation” on page 1-8 (if applicable).

Check MATLAB Cosimulation Server's Link Status

The first step to starting an HDL simulator and MATLAB test bench or component function session is to check the link status of the MATLAB server. Is the server running? If the server is running, what mode of communication and, if applicable, what TCP/IP socket port is the server using for its links? You can retrieve this information by using the MATLAB function `hdldaemon` with the `'status'` option. For example:

```
hdldaemon('status')
```

The function displays a message that indicates whether the server is running and, if it is running, the number of connections it is handling. For example:

```
HDLDaemon socket server is running on port 4449 with 0 connections
```

If the server is not running, the message reads

```
HDLDaemon is NOT running
```

See the Options: Inputs section in the `hdldaemon` reference documentation for information on determining the mode of communication and the TCP/IP socket in use.

Run Cosimulation

You can run a cosimulation session using both the MATLAB and HDL simulator GUIs (typical) or, to reduce memory demand, you can run the cosimulation using the command line interface (CLI) or in batch mode.

- “Cosimulation with MATLAB Using the HDL Simulator GUI” on page 1-5

- “Cosimulation with MATLAB Using the Command Line Interface (CLI)” on page 1-6
- “Cosimulation with MATLAB Using Batch Mode” on page 1-7

Cosimulation with MATLAB Using the HDL Simulator GUI

These steps describe a typical sequence for running a simulation interactively from the main HDL simulator window:

- 1 Set breakpoints in the HDL and MATLAB code to verify and analyze simulation progress.

How you set breakpoints in the HDL simulator will vary depending on what simulator application you are using.

In MATLAB, there are several ways you can set breakpoints; for example, by using the **Set/Clear Breakpoint** button on the toolbar.

- 2 Issue `matlabtb` command at the HDL simulator prompt.

When you begin a specific test bench or component session, you specify parameters that identify the following information:

- The mode and, if applicable, TCP/IP data for connecting to a MATLAB server (see `matlabtb` reference)
- The MATLAB function that is associated with and executes on behalf of the HDL instance. See “Bind HDL Module Component to MATLAB Test Bench Function” on page 2-15.
- Timing specifications and other control data that specifies when the module's MATLAB function is to be called. See “Schedule Options for a Test Bench Session” on page 2-15.

For example:

```
hdlsim> matlabtb osc_top -sensitivity /osc_top/sine_out
        -socket 4448 -mfunc hosctb
```

- 3 Start the simulation by entering the HDL simulator run command.

The run command offers a variety of options for applying control over how a simulation runs (refer to your HDL simulator documentation for details). For example, you can specify that a simulation run for several time steps.

The following command instructs the HDL simulator to run the loaded simulation for 50000 time steps:

```
run 50000
```

- 4 Step through the simulation and examine values.

How you step through the simulation in the HDL simulator will vary depending on what simulator application you are using.

In MATLAB, there are several ways you can step through code; for example, by clicking the **Step** toolbar button.

- 5 When you block execution of the MATLAB function, the HDL simulator also blocks and remains blocked until you clear all breakpoints in the function's code.
- 6 Resume the simulation, as desired.

How you resume the simulation in the HDL simulator will vary depending on what simulator application you are using.

In MATLAB, there are several ways you can resume the simulation; for example, by clicking the **Continue** toolbar button.

The following HDL simulator command resumes a simulation:

```
run -continue
```

For more information on HDL simulator and MATLAB debugging features, see the HDL simulator documentation and MATLAB online help or documentation.

Cosimulation with MATLAB Using the Command Line Interface (CLI)

Running your cosimulation session using the command-line interface allows you to interact with the HDL simulator during cosimulation, which can be helpful for debugging.

To use the CLI, specify "CLI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command.

The Tcl command you build to pass to the HDL simulator launch command must contain the run command or no cosimulation will take place.

Caution Close the terminal window by entering `quit -f` at the command prompt. Do not close the terminal window by clicking the "X" in the upper right-hand corner. This causes a memory-type error to be issued from the system. This is not a bug with HDL Verifier but just the way the HDL simulator behaves in this context.

You can type CTRL+C to interrupt and terminate the simulation in the HDL simulator but this action also causes the memory-type error to be displayed.

Specify CLI mode with `nclaunch` (Cadence Xcelium)

Issue the `nclaunch` command with "CLI" as the `runmode` property value, as follows (example entered into the MATLAB editor):

```
tclcmd = { ['cd ',projdir],...
  ['exec xmvlog -64bit', srcfile],...
  ['exec xmelab -64bit -access +wc lowpass_filter',...
  ['hdlsimmatlab -gui lowpass_filter ', ...
  ' -input "{@matlabtb lowpass_filter 10ns -repeat 10ns ...
    -mfunc filter_tb_incisive}"',...
  ' -input "{@force lowpass_filter.clk_enable 1 -after 0ns}"',...
  ' -input "{@force lowpass_filter.reset 1 -after 0ns 0 -after 22ns}"',...
  ' -input "{@force lowpass_filter.clk 1 -after 0ns 0 -after 5ns ...
    -repeat 10ns}"',...
  ' -input "{@deposit lowpass_filter.filter_in 0}"',...
  1};
```

```
nclaunch('tclstart',tclcmd,'runmode','CLI');
```

Specify CLI mode with `vsim` (Mentor Graphics ModelSim)

Issue the `vsim` command with "CLI" as the `runmode` property value, as follows (example entered into the MATLAB editor):

```
tclcmd = { ['cd ',unixprojdir],...
  'vlib work',... %create library (if applicable)
  'force /osc_top/clk_enable 1 0',...
```

```
'force /osc_top/reset 1 0, 0 120 ns',...
'force /osc_top/clk 1 0 ns, 0 40 ns -r 80ns',...
};

vsim('tclstart',tclcmd,'runmode','CLI');
```

Cosimulation with MATLAB Using Batch Mode

Running your cosimulation session in batch mode allows you to keep the process in the background, reducing demand on memory by disengaging the GUI.

To use the batch mode, specify "Batch" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command. After you issue the HDL Verifier HDL simulator launch command with batch mode specified, start the simulation in Simulink®. To stop the HDL simulator before the simulation is completed, issue the `breakHdlSim` command.

Specify Batch mode with `nclaunch` (Cadence Xcelium)

Issue the `nclaunch` command with "Batch" as the runmode parameter, as follows:

```
nclaunch('tclstart',manchestercmds,'runmode','Batch')
```

You can also set runmode to 'Batch with Xterm', which starts the HDL simulator in the background but shows the session in an Xterm.

Specify Batch mode with `vsim` (Mentor Graphics ModelSim)

On Windows®, specifying batch mode causes ModelSim to be run in a non-interactive command window. On Linux®, specifying batch mode causes ModelSim to be run in the background with no window.

Issue the `vsim` command with 'Batch' as the runmode parameter, as follows:

```
>> vsim('tclstart',manchestercmds,'runmode','Batch')
```

Apply Stimuli to Cosimulation Session with `force` Command

After you establish a connection between the HDL simulator and MATLAB, you can then apply stimuli to the test bench or component cosimulation environment. One way of applying stimuli is through the `iport` parameter of the linked MATLAB function. This parameter forces signal values by deposit.

Other ways to apply stimuli include issuing `force` commands in the HDL simulator main window (for ModelSim, you can also use the **Edit > Clock** option in the **ModelSim Signals** window).

For example, consider the following sequence of `force` commands:

- Xcelium

```
force osc_top.clk_enable 1 -after 0ns
force osc_top.reset 0 -after 0ns 1 -after 40ns 0 -after 120ns
force osc_top.clk 1 -after 0ns 0 -after 40ns -repeat 80ns
```

- ModelSim

```
VSIM n> force clk 0 0 ns, 1 5 ns -repeat 10 ns
VSIM n> force clk_en 1 0
VSIM n> force reset 0 0
```

These commands drive the following signals:

- The `clk` signal to 0 at 0 nanoseconds after the current simulation time and to 1 at 5 nanoseconds after the current HDL simulation time. This cycle repeats starting at 10 nanoseconds after the current simulation time, causing transitions from 1 to 0 and 0 to 1 every 5 nanoseconds, as the following diagram shows.



For example,

```
force /foobar/clk 0 0, 1 5 -repeat 10
```

- The `clk_en` signal to 1 at 0 nanoseconds after the current simulation time.
- The `reset` signal to 0 at 0 nanoseconds after the current simulation time.

Xcelium Users: Using HDL to Code Clock Signals Instead of the `force` Command

You should consider using HDL to code clock signals as `force` is a lower performance solution in the current version of Cadence Xcelium simulators.

The following are ways that a periodic force might be introduced:

- Via the Clock pane in the HDL Cosimulation block
- Via pre/post Tcl commands in the HDL Cosimulation block
- Via a user-input Tcl script to `xmsim`

All three approaches may lead to performance degradation.

Restart Simulation

Because the HDL simulator issues the service requests during a MATLAB cosimulation session, you must restart the session from the HDL simulator. To restart a session, perform the following steps:

- 1 Make the HDL simulator your active window, if your input focus was not already set to that application.
- 2 Reload HDL design elements and reset the simulation time to zero.
- 3 Reissue the `matlabtb` or `matlabcp` command.

Note To restart a simulation that is in progress, issue a break command and end the current simulation session before restarting a new session.

Stop Simulation

When you are ready to stop a test bench or component session, it is best to do so in an orderly way to avoid possible corruption of files and to see that all application tasks shut down cleanly. You should stop a session as follows:

- 1 Make the HDL simulator your active window, if your input focus was not already set to that application.

- 2** Halt the simulation. You must quit the simulation at the HDL simulator side or MATLAB may hang until the simulator is quit.
- 3** Close your project.
- 4** Exit the HDL simulator, if you are finished with the application.
- 5** Quit MATLAB, if you are finished with the application. If you want to shut down the server manually, stop the server by calling `hdldaemon` with the 'kill' option:

```
hdldaemon('kill')
```

For more information on closing HDL simulator sessions, see the HDL simulator documentation.

HDL Cosimulation Using MATLAB Test Bench Function

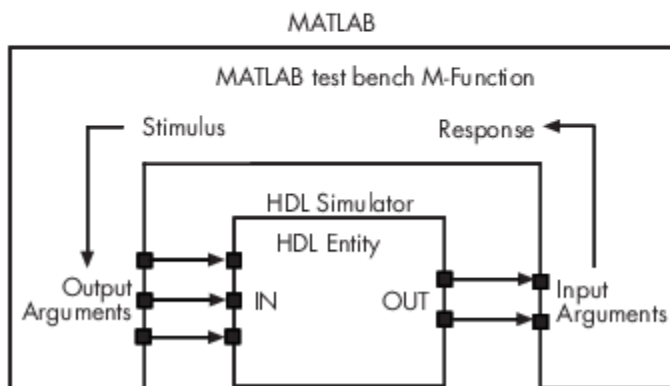
- “Create a MATLAB Test Bench” on page 2-2
- “Set Up Cosimulation Test Bench” on page 2-13
- “Verify HDL Module with MATLAB Test Bench” on page 2-18
- “Automatic Verification of Generated HDL Code from MATLAB” on page 2-30

Create a MATLAB Test Bench

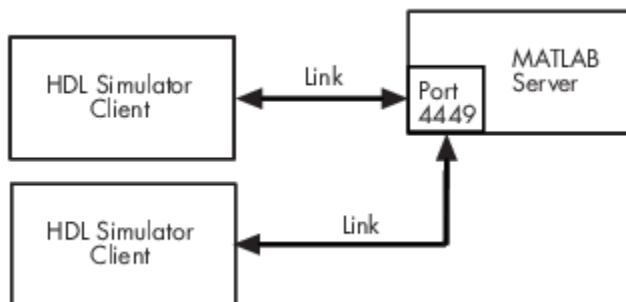
The HDL Verifier software provides a means for verifying HDL modules within the MATLAB environment. You do so by coding an HDL model and a MATLAB function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB test bench functions that communicate with the HDL simulator. Note that cosimulation with Vivado simulator does not support MATLAB function cosimulation.

MATLAB test bench functions let you verify the performance of the HDL model, or of components within the model. A test bench function drives values onto signals connected to input ports of an HDL design under test and receives signal values from the output ports of the module.

The following figure shows how a MATLAB function wraps around and communicates with the HDL simulator during a test bench simulation session.



When linked with MATLAB, the HDL simulator functions as the client, with MATLAB as the server. The following figure shows a multiple-client scenario connecting to the server at TCP/IP socket port 4449.



The MATLAB server can service multiple simultaneous HDL simulator sessions and HDL modules. However, you should follow recommended guidelines to help the server track the I/O associated with each module and session. The MATLAB server, which you start with the supplied function `hdl_daemon`, waits for connection requests from instances of the HDL simulator running on the same

or different computers. When the server receives a request, it executes the specified MATLAB function you have coded to perform tasks on behalf of a module in your HDL design. Parameters that you specify when you start the server indicate whether the server establishes shared memory or TCP/IP socket communication links.

Refer to “Cosimulation Configurations” on page 10-2 for valid machine configurations.

Note The programming, interfacing, and scheduling conventions for test bench functions and component functions are virtually identical. For the most part, the same procedures apply to both types of functions.

Follow these workflow steps to create a MATLAB test bench session for cosimulation with the HDL simulator.

- 1 “Write HDL Modules for MATLAB Test Bench” on page 2-3
- 2 “Write a Test Bench Function” on page 2-6
- 3 “Set Up MATLAB-HDL Simulator Connection” on page 1-2
- 4 “Place Test Bench on MATLAB Search Path” on page 2-13
- 5 “Bind Test Bench Function Calls With `matlabtb`” on page 2-13
- 6 “Schedule Options for a Test Bench Session” on page 2-15
- 7 Set breakpoints for interactive HDL debug (optional).
- 8 “Run MATLAB-HDL Cosimulation” on page 1-4

Write HDL Modules for MATLAB Test Bench

- “Coding HDL Modules for Verification with MATLAB” on page 2-3
- “Choose HDL Module Name for Use with MATLAB Test Bench” on page 2-3
- “Specify Port Direction Modes in HDL Module for Use with Test Bench” on page 2-4
- “Specify Port Data Types in HDL Modules for Use with Test Bench” on page 2-4
- “Compile and Elaborate HDL Design for Use with Test Bench” on page 2-5
- “Sample VHDL Entity Definition” on page 2-6

Coding HDL Modules for Verification with MATLAB

The most basic element of communication in the HDL Verifier interface is the HDL module. The interface passes all data between the HDL simulator and MATLAB as port data. The HDL Verifier software works with any existing HDL module. However, when you code an HDL module that is targeted for MATLAB verification, you should consider its name, the types of data to be shared between the two environments, and the direction modes.

Choose HDL Module Name for Use with MATLAB Test Bench

Although not required, when naming the HDL module, consider choosing a name that also can be used as a MATLAB function name. (Generally, naming rules for VHDL[®] or Verilog[®] and MATLAB are compatible.) By default, HDL Verifier software assumes that an HDL module and its simulation function share the same name. See “Bind Test Bench Function Calls With `matlabtb`” on page 2-13.

For details on MATLAB function-naming guidelines, see “MATLAB Programming Tips” on files and file names in the MATLAB documentation.

Specify Port Direction Modes in HDL Module for Use with Test Bench

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). The following table defines these three modes.

Use VHDL Mode...	Use Verilog Mode...	For Ports That...
IN	input	Represent signals that can be driven by a MATLAB function or Simulink test bench
OUT	output	Represent signal values that are passed to a MATLAB function or Simulink test bench
INOUT	inout	Represent bidirectional signals that can be driven by or pass values to a MATLAB function or Simulink test bench

Specify Port Data Types in HDL Modules for Use with Test Bench

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the HDL Verifier interface converts data types for the MATLAB environment, see “Supported Data Types” on page 10-35.

Note If you use unsupported types, the HDL Verifier software issues a warning and ignores the port at run time. For example, if you define your interface with five ports, one of which is a VHDL access port, at run time, then the interface displays a warning and your code sees only four ports.

Port Data Types for VHDL Entities

In your entity statement, you must define each port that you plan to test with MATLAB with a VHDL data type that is supported by the HDL Verifier software. The interface can convert scalar and array data of the following VHDL types to comparable MATLAB types:

- STD_LOGIC, STD_ULOGIC, BIT, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, and BIT_VECTOR
- INTEGER and NATURAL
- REAL
- TIME
- Enumerated types, including user-defined enumerated types and CHARACTER

The interface also supports all subtypes and arrays of the preceding types.

Note The HDL Verifier software does not support VHDL extended identifiers for the following components:

- Port and signal names used in cosimulation
- Enum literals when used as array indices of port and signal names used in cosimulation

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Modules

In your module definition, you must define each port that you plan to test with MATLAB with a Verilog port data type that is supported by the HDL Verifier software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note HDL Verifier software does not support Verilog escaped identifiers for port and signal names used in cosimulation. However, it does support simple identifiers for Verilog.

Compile and Elaborate HDL Design for Use with Test Bench

After you create or edit your HDL source files, use the HDL simulator compiler to compile and debug the code.

Compilation for ModelSim

You have the option of invoking the compiler from menus in the ModelSim graphic interface or from the command line with the `vcom` command. The following sequence of ModelSim commands creates and maps the design library `work` and compiles the VHDL file `modsimrand.vhd`:

```
ModelSim> vlib work
ModelSim> vmap work work
ModelSim> vcom modsimrand.vhd
```

The following sequence of ModelSim commands creates and maps the design library `work` and compiles the Verilog file `test.v`:

```
ModelSim> vlib work
ModelSim> vmap work work
ModelSim> vlog test.v
```

Note You should provide read/write access to the signals that are connecting to the MATLAB session for cosimulation. For higher performance, you want to provide access only to those signals used in cosimulation. You can check read/write access through the HDL simulator—see HDL simulator documentation for details.

Compilation for Xcelium

The Cadence Xcelium simulator allows for 1-step and 3-step processes for HDL compilation, elaboration, and simulation. The following Xcelium simulator command compiles the Verilog file `test.v`:

```
sh> xmvlog -64bit test.v
```

The following Xcelium simulator command compiles and elaborates the Verilog design `test.v`, and then loads it for simulation, in a single step:

```
sh> xmvlog -64bit +gui +access+rwc +linedebug test.v
```

The following sequence of Xcelium simulator commands performs all the same processes in multiple steps:

```
sh> xmvlog -64bit -linedebug test.v
sh> xmelab -64bit -access +rwc test
sh> xmsim test
```

Note You should provide read/write access to the signals that are connecting to the MATLAB session for cosimulation. The previous example shows how to provide read/write access to all signals in your design. For higher performance, you want to provide access only to those signals used in cosimulation. See the description of the `+access` flag to `xmvlog` and the `-access` argument to `xmelab` for details.

For more examples, see the HDL Verifier tutorials and demos. For details on using the HDL compiler, see the simulator documentation.

Sample VHDL Entity Definition

This sample VHDL code fragment defines the entity decoder. By default, the entity is associated with MATLAB test bench function `decoder`.

The keyword `PORT` marks the start of the entity's port clause, which defines two `IN` ports—`isum` and `qsum`—and three `OUT` ports—`adj`, `dvalid`, and `odata`. The output ports drive signals to MATLAB function input ports for processing. The input ports receive signals from the MATLAB function output ports.

Both input ports are defined as vectors consisting of five standard logic values. The output port `adj` is also defined as a standard logic vector, but consists of only two values. The output ports `dvalid` and `odata` are defined as scalar standard logic ports. For information on how the HDL Verifier interface converts data of standard logic scalar and array types for use in the MATLAB environment, see “Supported Data Types” on page 10-35.

```
ENTITY decoder IS
PORT (
  isum   : IN std_logic_vector(4 DOWNT0 0);
  qsum   : IN std_logic_vector(4 DOWNT0 0);
  adj    : OUT std_logic_vector(1 DOWNT0 0);
  dvalid : OUT std_logic;
  odata  : OUT std_logic);
END decoder ;
```

Write a Test Bench Function

Coding MATLAB Cosimulation Functions

Coding a MATLAB function to verify an HDL module or component requires that you follow specific coding conventions. You must also understand the data type conversions that occur, and program data type conversions for operating on data and returning data to the HDL simulator.

To code a MATLAB function to verify an HDL module or component, perform the following steps:

- 1 Learn the syntax for a MATLAB HDL Verifier test bench function. See “Syntax of a Test Bench Function” on page 2-7.

- 2 Understand how HDL Verifier software converts data from the HDL simulator for use in the MATLAB environment. See “Supported Data Types” on page 10-35.
- 3 Choose a name for the MATLAB function. See “Bind HDL Module Component to MATLAB Test Bench Function” on page 2-15.
- 4 Define expected parameters in the function definition line. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.
- 5 Determine the types of port data being passed into the function. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.
- 6 Extract and, if applicable to the simulation, apply information received in the `portinfo` structure. See “Gaining Access to and Applying Port Information” on page 10-24.
- 7 Convert data for manipulation in the MATLAB environment, as applicable. See “Converting HDL Data to Send to MATLAB or Simulink” on page 10-35.
- 8 Convert data that needs to be returned to the HDL simulator. See “Converting Data for Return to the HDL Simulator” on page 10-40.

For more tips, see “Test Bench and Component Function Writing” on page 10-19.

Syntax of a Test Bench Function

The syntax of a MATLAB test bench function is

```
function [iport, tnext] = MyFunctionName(oport, tnow, portinfo)
```

See the “MATLAB Function Syntax and Function Argument Definitions” on page 10-22 for an explanation of each of the function arguments.

Sample MATLAB Test Bench Function

This section uses a sample MATLAB function to identify sections of a MATLAB test bench function required by the HDL Verifier software. You can see the full text of the code used in this sample in the section “MATLAB Builder EX Function Example: manchester_decoder.m” on page 2-10.

For ModelSim Users This example uses a VHDL entity and MATLAB Builder™ EX function code drawn from the decoder portion of the Manchester Receiver example. For the complete VHDL and function code listings, see the following files:

```
matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\vhdl\manchester\decoder.vhd
```

```
matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\manchester_decoder.m
```

As the first step to coding a MATLAB Builder EX test bench function, you must understand how the data modeled in the VHDL entity maps to data in the MATLAB Builder EX environment. The VHDL entity decoder is defined as follows:

```
ENTITY decoder IS
PORT (
  isum   : IN std_logic_vector(4 DOWNT0 0);
  qsum   : IN std_logic_vector(4 DOWNT0 0);
  adj    : OUT std_logic_vector(1 DOWNT0 0);
  dvalid : OUT std_logic;
  odata  : OUT std_logic
);
END decoder ;
```

The following discussion highlights key lines of code in the definition of the `manchester_decoder` MATLAB Builder EX function:

1 Specify the MATLAB function name and required parameters.

The following code is the function declaration of the `manchester_decoder` MATLAB Builder EX function.

```
function [iport,tnext] = manchester_decoder(oport,tnow,portinfo)
```

See “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.

The function declaration performs the following actions:

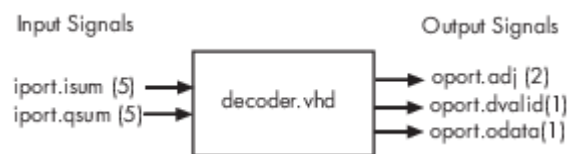
- Names the function. This declaration names the function `manchester_decoder`, which differs from the entity name `decoder`. Because the names differ, the function name must be specified explicitly later when the entity is initialized for verification with the `matlabtb` or `matlabtbeval` function. See “Bind HDL Module Component to MATLAB Test Bench Function” on page 2-15.
- Defines required argument and return parameters. A MATLAB Builder EX test bench function *must* return two parameters, `iport` and `tnext`, and pass three arguments, `oport`, `tnow`, and `portinfo`, and *must* appear in the order shown. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.

The function outputs must be initialized to empty values, as in the following code example:

```
tnext = [];  
iport = struct();
```

You should initialize the function outputs at the beginning of the function, to follow recommended best practice.

The following figure shows the relationship between the entity's ports and the MATLAB Builder EX function's `iport` and `oport` parameters.



For more information on the required MATLAB Builder EX test bench function parameters, see “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.

2 Make note of the data types of ports defined for the entity being simulated.

The HDL Verifier software converts HDL data types to comparable MATLAB Builder EX data types and vice versa. As you develop your MATLAB Builder EX function, you must know the types of the data that it receives from the HDL simulator and needs to return to the HDL simulator.

The VHDL entity defined for this example consists of the following ports

VHDL Example Port Definitions

Port	Direction	Type...	Converts to/ Requires Conversion to...
isum	IN	STD_LOGIC_VECTOR(4 DOWNTO 0)	A 5-bit column or row vector of characters where each bit maps to a standard logic character literal.
qsum	IN	STD_LOGIC_VECTOR(4 DOWNTO 0)	A 5-bit column or row vector of characters where each bit maps to a standard logic character literal.
adj	OUT	STD_LOGIC_VECTOR(1 DOWNTO 0)	A 2-element column vector of characters. Each character matches a corresponding character literal that represents a logic state and maps to a single bit.
dvalid	OUT	STD_LOGIC	A character that matches the character literal representing the logic state.
odata	OUT	STD_LOGIC	A character that matches the character literal representing the logic state.

For more information on interface data type conversions, see “Supported Data Types” on page 10-35.

3 Set up any required timing parameters.

The `tnext` assignment statement sets up timing parameter `tnext` such that the simulator calls back the MATLAB Builder EX function every nanosecond.

```
tnext = tnow+1e-9;
```

4 Convert output port data to MATLAB data types for processing.

The following code excerpt illustrates data type conversion of output port data.

```
%% Compute one row and plot
isum = isum + 1;
adj(isum) = mvl2dec(oport.adj');
data(isum) = mvl2dec([oport.dvalid oport.odata]);
```

.
.
.

The two calls to `mvl2dec` convert the binary data that the MATLAB Builder EX function receives from the entity's output ports, `adj`, `dvalid`, and `odata` to unsigned decimal values that MATLAB Builder EX can compute. The function converts the 2-bit transposed vector `oport.adj` to a decimal value in the range 0 to 4 and `oport.dvalid` and `oport.odata` to the decimal value 0 or 1.

"MATLAB Function Syntax and Function Argument Definitions" on page 10-22 provides a summary of the types of data conversions to consider when coding simulation MATLAB functions.

5 Convert data to be returned to the HDL simulator.

The following code excerpt illustrates data type conversion of data to be returned to the HDL simulator.

```
if isum == 17
    iport.isum = dec2mvl(isum,5);
    iport.qsum = dec2mvl(qsum,5);
else
    iport.isum = dec2mvl(isum,5);
end
```

The three calls to `dec2mvl` convert the decimal values computed by MATLAB Builder EX to binary data that the MATLAB Builder EX function can deposit to the entity's input ports, `isum` and `qsum`. In each case, the function converts a decimal value to 5-element bit vector with each bit representing a character that maps to a character literal representing a logic state.

"Converting Data for Return to the HDL Simulator" on page 10-40 provides a summary of the types of data conversions to consider when returning data to the HDL simulator.

MATLAB Builder EX Function Example: `manchester_decoder.m`

```
function [iport,tnext] = manchester_decoder(oport,tnow,portinfo)
% MANCHESTER_DECODER Test bench for VHDL 'decoder'
% [IPORT,TNEXT]=MANCHESTER_DECODER(OPORT,TNOW,PORTINFO) -
% Implements a test of the VHDL decoder entity which is part
% of the Manchester receiver demo. This test bench plots
% the IQ mapping produced by the decoder.
%
%      iport          oport
%      +-----+
% isum -(5)->|         |-(2)-> adj
% qsum -(5)->| decoder |-(1)-> dvalid
%           |         |-(1)-> odata
%           +-----+
%
% isum - Inphase Convolution value
% qsum - Quadrature Convolution value
% adj   - Clock adjustment ('01','00','10')
% dvalid - Data validity ('1' = data is valid)
% odata - Recovered data stream
%
% Adjust = 0 (00b), generate full 16 cycle waveform
% Copyright 2003-2009 The MathWorks, Inc.
persistent isum;
persistent qsum;
%persistent ga;
persistent x;
persistent y;
```



```

persistent adj;
persistent data;
global testisdone;
% This useful feature allows you to manually
% reset the plot by simply typing: >manchester_decoder
tnext = [];
iport = struct();

if nargin == 0,
    isum = [];
    return;
end

if exist('portinfo') == 1
    isum = [];
end

tnext = tnow+1e-9;
if isempty(isum), %% First call
    scale = 9;
    isum = 0;
    qsum = 0;
    for k=1:2,
        ga(k) = subplot(2,1,k);
        axis([-1 17 -1 17]);
        ylabel('Quadrature');
        line([0 16],[8 8],'Color','r','LineStyle',':','LineWidth',1)
        line([8 8],[0 16],'Color','r','LineStyle',':','LineWidth',1)
    end
    xlabel('Inphase');
    subplot(2,1,1);
    title('Clock Adjustment (adj)');
    subplot(2,1,2);
    title('Data with Validity');
    iport.isum = '00000';
    iport.qsum = '00000';
    return;
end

% compute one row, then plot
isum = isum + 1;
adj(isum) = bin2dec(oport.adj');
data(isum) = bin2dec([oport.dvalid oport.odata]);

if isum == 17,
    subplot(2,1,1);
    for k=0:16,
        if adj(k+1) == 0, % Bang on!
            line(k,qsum,'color','k','Marker','o');
        elseif adj(k+1) == 1, %
            line(k,qsum,'color','r','Marker','<');
        else
            line(k,qsum,'color','b','Marker','>');
        end
    end
    subplot(2,1,2);
    for k=0:16,
        if data(k+1) < 2, % Invalid
            line(k,qsum,'color','r','Marker','X');
        else
            if data(k+1) == 2, %Valid and 0!
                line(k,qsum,'color','g','Marker','o');
            else
                line(k,qsum,'color','k','Marker','.');
            end
        end
    end
end

isum = 0;
qsum = qsum + 1;
if qsum == 17,
    qsum = 0;
end

```

```
        disp('done');
        tnext = []; % suspend callbacks
        testisdone = 1;
        return;
    end
    iport.isum = dec2bin(isum,5);
    iport.qsum = dec2bin(qsum,5);
else
    iport.isum = dec2bin(isum,5);
end
```

See Also

More About

- “Test Bench and Component Function Writing” on page 10-19

Set Up Cosimulation Test Bench

This article explains how to set up a MATLAB test bench to cosimulate an HDL module represented as a MATLAB function. This workflow is not supported for the Vivado simulator.

Place Test Bench on MATLAB Search Path

Use MATLAB which Function to Find Test Bench

The MATLAB function that you are associating with an HDL component must be on the MATLAB search path or reside in the current working folder (see the MATLAB `cd` function). To verify whether the function is accessible, use the MATLAB `which` function. The following call to `which` checks whether the function `MyVhdlFunction` is on the MATLAB search path, for example:

```
which MyVhdlFunction  
/work/xcelium/MySym/MyVhdlFunction.m
```

If the specified function is on the search path, `which` displays the complete path to the function. If the function is not on the search path, `which` informs you that the file was not found.

Add Test Bench Function to MATLAB Search Path

To add a MATLAB function to the MATLAB search path:

- On the **Home** tab, in the **Environment** section, click **Set Path**.
- Use the `addpath` function.
- For temporary access, change the MATLAB working folder to a desired location with the `cd` command.

Bind Test Bench Function Calls With `matlabtb`

Invoke MATLAB Test Bench Command `matlabtb`

You invoke `matlabtb` by issuing the command in the HDL simulator. See the Examples section of the `matlabtb` reference page for several examples of invoking `matlabtb`.

Be sure to follow the path specifications for MATLAB test bench sessions when invoking `matlabtb`, as explained in “Specify HDL Signal/Port and Module Paths for MATLAB Test Bench Cosimulation” on page 2-13.

For instructions in issuing the `matlabtb` command, see “Run MATLAB-HDL Cosimulation” on page 1-4.

Specify HDL Signal/Port and Module Paths for MATLAB Test Bench Cosimulation

HDL Verifier software has specific requirements for specifying HDL design hierarchy, the syntax of which is described in the following sections: one for Verilog at the top level, and one for VHDL at the top level. Do not use a file name hierarchy in place of the design hierarchy name.

The rules stated in this section apply to signal/port and module path specifications for MATLAB cosimulation sessions. Other specifications may work but the HDL Verifier software does not officially recognize nor support them.

In the following example:

```
matlabtb u_osc_filter -mfunc oscfilter
```

`u_osc_filter` is the top-level component. If you specify a subcomponent, you must follow valid module path specifications for MATLAB cosimulation sessions.

Path Specifications for MATLAB Link Sessions with Verilog Top Level

- The path specification must start with a top-level module name.
- The path specification can include "." or "/" path delimiters, but it cannot include mixed delimiters.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig  
/top/sub/port_or_sig  
top  
top/sub  
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`
:
:sub

Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Path Specifications for MATLAB Link Sessions with VHDL Top Level

- The path specification can include the top-level module name, but you do not have to include it.
- The path specification can include "." or "/" path delimiters, but it cannot include mixed delimiters.
- The leaf module or signal must match the HDL language of the top-level module.

Examples for ModelSim and Xcelium Users

The following examples show valid signal and module path specifications:

```
top.port_or_sig  
/sub/port_or_sig  
top  
top/sub  
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`
- `:`
- `:sub`

Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Bind HDL Module Component to MATLAB Test Bench Function

By default, the HDL Verifier software assumes that the name for a MATLAB function matches the name of the HDL module that the function verifies. When you create a test bench or component function that has a different name than the design under test, you must associate the design with the MATLAB function using the `-mfunc` argument to `matlabtb`. This argument associates the HDL module instance to a MATLAB function that has a different name from the HDL instance.

For more information on the `-mfunc` argument and for a full list of `matlabtb` parameters, see the `matlabtb` function reference.

For details on MATLAB function naming guidelines, see "MATLAB Programming Tips" on files and file names in the MATLAB documentation.

Example of Binding Test Bench and Component Function Calls

In this first example, you form an association between the `inverter_vl` component and the MATLAB test bench function `inverter_tb` by invoking the function `matlabtb` with the `-mfunc` argument when you set up the simulation.

```
matlabtb inverter_vl -mfunc inverter_tb
```

The `matlabtb` command instructs the HDL simulator to call back the `inverter_tb` function when `inverter_vl` executes in the simulation.

In this second example, you bind the model `osc_top.u_osc_filter` to the component function `oscfilter`:

```
matlabcp osc_top.u_osc_filter -mfunc oscfilter
```

When the HDL simulator calls the `oscfilter` callback, the function knows to operate on the model `osc_top.u_osc_filter`.

Schedule Options for a Test Bench Session

About Scheduling Options for Test Bench Sessions

There are two ways to schedule the invocation of a MATLAB function:

- Using the arguments to the HDL Verifier function `matlabtb` or `matlabcp`
- Inside the MATLAB function using the `tnext` parameter

The two types of scheduling are not mutually exclusive. You can combine the `matlabtb` or `matlabcp` timing arguments and the `tnext` parameter of a MATLAB function to schedule test bench or component session callbacks.

Schedule Test Bench Session Using `matlabtb` Arguments

By default, the HDL Verifier software invokes a MATLAB test bench or component function once (at the time that you make the call to `matlabtb` or `matlabcp`). If you want to apply more control, and execute the MATLAB function more than once, use the command scheduling options. With these options, you can specify when and how often the HDL Verifier software invokes the relevant MATLAB function. If applicable, modify the function or specify timing arguments when you begin a MATLAB test bench or component function session with the `matlabtb` or `matlabcp` function.

You can schedule a MATLAB test bench or component function to execute using the command arguments under any of the following conditions:

- **Discrete time values**—Based on time specifications that can also include repeat intervals and a stop time
- **Rising edge**—When a specified signal experiences a rising edge
 - VHDL: Rising edge is {0 or L} to {1 or H}.
 - Verilog: Rising edge is the transition from 0 to x, z, or 1, and from x or z to 1.
- **Falling edge**—When a specified signal experiences a falling edge
 - VHDL: Falling edge is {1 or H} to {0 or L}.
 - Verilog: Falling edge is the transition from 1 to x, z, or 0, and from x or z to 0.
- **Signal state change**—When a specified signal changes state, based on a list using the `-sensitivity` argument to `matlabtb`.

Schedule Test Bench Functions With the `tnext` Parameter

You can control the callback timing of a MATLAB function by using that function's `tnext` parameter. This parameter passes a time value to the HDL simulator, and the value gets added to the simulation schedule for that function. If the function returns a null value (`[]`), the software does not add any new entries to the schedule.

You can set the value of `tnext` to a value of type `double` or `int64`. Specify `double` to express the callback time in seconds. For example, to schedule a callback in 1 ns, specify:

```
tnext = 1e-9
```

Specify `int64` to convert to an integer multiple of the current HDL simulator time resolution limit. For example: if the HDL simulator time precision is 1 ns, to schedule a callback at 100 ns, specify:

```
tnext=int64(100)
```

Note The `tnext` parameter represents time from the start of the simulation. Therefore, `tnext` must always be greater than `tnow`. If it is less, the software does not schedule a callback.

For more information on `tnext` and the function prototype, see “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.

Examples

In this first example, each time the HDL simulator calls the test bench function (via HDL Verifier), `tnext` schedules the next callback to the MATLAB function for 1 ns later, relative to the current simulation time:

```
tnext = [];  
.  
.  
.  
tnext = tnow+1e-9;
```

Using `tnext` you can dynamically decide the callback scheduling based on criteria specific to the operation of the test bench. For example, you can decide to stop scheduling callbacks when a data signal has a certain value:

```
if qsum == 17,  
    qsum = 0;  
    disp('done');  
    tnext = []; % suspend callbacks  
    testisdone = 1;  
    return;  
end
```

This next example demonstrates scheduling a component session using `tnext`. In the Oscillator example, the `oscfilter` function calculates a time interval at which the HDL simulator calls the callbacks. The component function calculates this interval on the first call to `oscfilter` and stores the result in the variable `fastestrate`. The variable `fastestrate` represents the sample period of the fastest oversampling rate supported by the filter. The function derives this rate from a base sampling period of 80 ns.

The following assignment statement sets the timing parameter `tnext`. This parameter schedules the next callback to the MATLAB component function, relative to the current simulation time (`tnow`).

```
tnext = tnow + fastestrate;
```

The function returns a new value for `tnext` each time the HDL simulator calls the function.

Verify HDL Module with MATLAB Test Bench

In this section...
“Tutorial Overview” on page 2-18
“Set Up Tutorial Files” on page 2-18
“Start the MATLAB Server” on page 2-19
“Start ModelSim Simulator and Set Up for Cosimulation” on page 2-20
“Develop VHDL Code” on page 2-21
“Compile VHDL Code” on page 2-23
“Develop MATLAB Function” on page 2-23
“Load Simulation” on page 2-24
“Run Simulation” on page 2-26
“Shut Down Simulation” on page 2-29

Tutorial Overview

This tutorial guides you through the basic steps for setting up an HDL Verifier application that uses MATLAB to verify a simple HDL design. In this tutorial, you develop, simulate, and verify a model of a pseudorandom number generator based on the Fibonacci sequence. The model is coded in VHDL.

Note This tutorial demonstrates creating and running a test bench using ModelSim SE 6.5. If you are not using this version, the messages and screen images from ModelSim may not appear to you exactly as they do in this tutorial.

This tutorial requires MATLAB, the HDL Verifier software, and the ModelSim HDL simulator.

In this tutorial, you will perform the following steps:

- 1 “Set Up Tutorial Files” on page 2-18
- 2 “Start the MATLAB Server” on page 2-19
- 3 “Start ModelSim Simulator and Set Up for Cosimulation” on page 2-20
- 4 “Develop VHDL Code” on page 2-21
- 5 “Compile VHDL Code” on page 2-23
- 6 “Develop MATLAB Function” on page 2-23
- 7 “Load Simulation” on page 2-24
- 8 “Run Simulation” on page 2-26
- 9 “Shut Down Simulation” on page 2-29

Set Up Tutorial Files

To help others have access to copies of the tutorial files, set up a folder for your own tutorial work:

- 1 Create a folder outside the scope of your MATLAB installation folder into which you can copy the tutorial files. The folder must be writable. This tutorial assumes that you create a folder named MyPlayArea.
- 2 Copy the following files to the folder you just created:

```
matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos
\modelsimrand_plot.m
```

```
matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\VHDL
\modelsimrand\modelsimrand.vhd
```

Start the MATLAB Server

This section describes starting MATLAB, setting up the current folder for completing the tutorial, starting the MATLAB server component, and checking for client connections, using shared memory or TCP/IP socket mode. These instructions assume you are familiar with the MATLAB user interface.

Perform the following steps:

- 1 Start MATLAB.
- 2 Set your MATLAB current folder to the folder you created in “Set Up Tutorial Files” on page 2-18.
- 3 Verify that the MATLAB server is running by calling function `hdldaemon` with the 'status' option in the MATLAB Command Window as shown here:

```
hdldaemon('status')
```

If the server is not running, the function displays

```
HDLDaemon is NOT running
```

If the server is running in TCP/IP socket mode, the message reads

```
HDLDaemon socket server is running on Port portnum with 0 connections
```

If the server is running in shared memory mode, the message reads

```
HDLDaemon shared memory server is running with 0 connections
```

If the server is not currently running, skip to step 5.

- 4 Shut down the server by typing

```
hdldaemon('kill')
```

You will see the following message that confirms that the server was shut down.

```
HDLDaemon server was shutdown
```

- 5 Start the server in TCP/IP socket mode by calling `hdldaemon` with the property name/property value pair 'socket' 0. The value 0 specifies that the operating system assign the server a TCP/IP socket port that is available on your system. For example

```
hdldaemon('socket', 0)
```

The server informs you that it has started by displaying the following message. The *portnum* will be specific to your system:

HDLDaemon socket server is running on Port *portnum* with 0 connections

Make note of *portnum* as you will need it when you issue the `matlabtb` command in “Load Simulation” on page 2-24.

You can alternatively specify that the MATLAB server use shared memory communication instead of TCP/IP socket communication; however, for this tutorial we will use socket communication as means of demonstrating this type of connection. For details on how to specify the various options, see the description of `hdldaemon`.

Start ModelSim Simulator and Set Up for Cosimulation

This section describes the basic procedure for starting the ModelSim software and setting up a ModelSim design library. These instructions assume you are familiar with the ModelSim user interface.

Perform the following steps:

- 1 Start ModelSim from the MATLAB environment by calling the function `vsim` in the MATLAB Command Window.

```
vsim
```

This function launches and configures ModelSim for use with the HDL Verifier software. The first folder of ModelSim matches your MATLAB current folder.

- 2 Verify the current ModelSim folder. You can verify that the current ModelSim folder matches the MATLAB current folder by entering the `ls` command in the ModelSim command window.



```
Transcript
ModelSim> ls
# compile_and_launch.tcl
# modsimrand.vhd
# modsimrand_plot.m
# transcript

ModelSim> ]
```

The command should list the files `modsimrand.vhd`, `modsimrand_plot.m`, `transcript`, and `compile_and_launch.tcl`.

If it does not, change your ModelSim folder to the current MATLAB folder. You can find the current MATLAB folder by looking in the Current Folder Browser or by viewing the Current folder navigation bar. In ModelSim, you can change the working folder by issuing the command

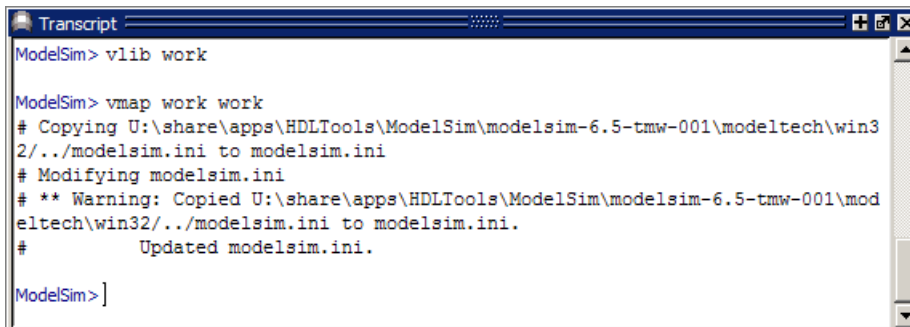
```
cd directory
```

Where *directory* is the folder you want to work from. Or you may also change directory by selecting **File > Change Directory...**

- 3 Create a design library to hold your compilation results. To create the library and required `_info` file, enter the `vlib` and `vmap` commands as follows:

```
ModelSim> vlib work
```

```
ModelSim> vmap work work
```



```
ModelSim> vlib work

ModelSim> vmap work work
# Copying U:\share\apps\HDLTools\ModelSim\modelsim-6.5-tmw-001\modeltech\win32\../modelsim.ini to modelsim.ini
# Modifying modelsim.ini
# ** Warning: Copied U:\share\apps\HDLTools\ModelSim\modelsim-6.5-tmw-001\modeltech\win32\../modelsim.ini to modelsim.ini.
# Updated modelsim.ini.

ModelSim> ]
```

Note You must use the ModelSim **File** menu or `vlib` command to create the library folder so that the required `_info` file is created. Do not create the library with operating system commands.

Develop VHDL Code

After setting up a design library, typically you would use the ModelSim Editor to create and modify your HDL code. For this tutorial, you do not need to create the VHDL code yourself. Instead, open and examine the existing file `modsimrand.vhd`. This section highlights areas of code in `modsimrand.vhd` that are of interest for a ModelSim and MATLAB test bench.

If you choose not to examine the HDL code at this time, skip to “Compile VHDL Code” on page 2-23.

You can open `modsimrand.vhd` in the edit window with the `edit` command, as follows:

```
ModelSim> edit modsimrand.vhd
```



```
ModelSim> edit modsimrand.vhd

ModelSim> ]
```

ModelSim opens its **edit** window and displays the VHDL code for `modsimrand.vhd`.

```

1 |-----
2 | -- Pseudo Random Word Generator
3 | -- Demonstration of 'Link for ModelSim'
4 | --
5 | --
6 | --
7 | -- Modelsim
8 | -->simmatlab work.modsimrand
9 | -->matlabtb modsimrand -mfunc modsimrand_plot -rising /modsimrand/clk
10 | -->force sim:/modsimrand/clk 0 0,1 5 ns -repeat 10 ns
11 | -->force sim:/modsimrand/clk_en 1
12 | -->force sim:/modsimrand/reset 1 0,0 50 ns
13 | -->run 80000
14 | --
15 | -- Copyright 2003 The MathWorks, Inc.
16 | -- $Revision: 1.1.6.1 $ $Date: 2009/03/02 22:08:59 $
17 | -----
18 |
19 | -----
20 | -- Entity: modsimrand
21 | -- Pseudo random algorithm
22 | -- Implements a uniform FN generator using
23 | -- a fibonacci sequence.
24 | -----
25 | LIBRARY IEEE;
26 | USE IEEE.std_logic_1164.all;
27 | USE IEEE.numeric_std.all;
28 |
29 | ENTITY modsimrand IS
30 | PORT (
31 |     clk      : IN std_logic ;

```

While you are viewing the file, note the following:

- The line ENTITY modsimrand contains the definition for the VHDL entity modsimrand:

```

ENTITY modsimrand IS
PORT (
    clk      : IN std_logic ;
    clk_en   : IN std_logic ;
    reset    : IN std_logic ;
    dout     : OUT std_logic_vector (31 DOWNT0 0);
END modsimrand;

```

This is the entity that will be verified in the MATLAB environment during the tutorial. Note the following:

- By default, the MATLAB server assumes that the name of the MATLAB function that verifies the entity in the MATLAB environment is the same as the entity name. You have the option of naming the MATLAB function explicitly. However, if you do not specify a name, the server expects the function name to match the entity name. In this example, the MATLAB function name is modsimrand_plot and does not match.
- The entity must be defined with a PORT clause that includes at least one port definition. Each port definition must specify a port mode (IN, OUT, or INOUT) and a VHDL data type that is supported by the HDL Verifier software.

The entity modsimrand in this example is defined with three input ports clk, clk_en, and reset of type STD_LOGIC and output port dout of type STD_LOGIC_VECTOR. The output port passes simulation output data out to the MATLAB function for verification. The optional input ports receive clock and reset signals from the function. Alternatively, the input ports can receive signals from ModelSim force commands.

For more information on coding port entities for use with MATLAB, see “Coding HDL Modules for Verification with MATLAB” on page 2-3.

- The remaining code for `modsimrand.vhd` defines a behavioral architecture for `modsimrand` that writes a randomly generated Fibonacci sequence to an output register when the clock experiences a rising edge.

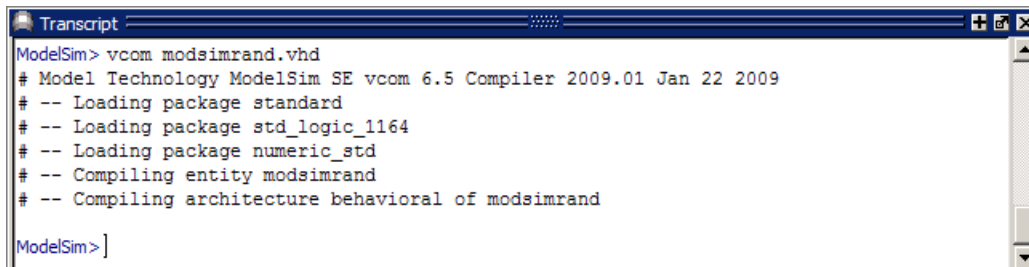
When you are finished examining the file, close the ModelSim **edit** window.

Compile VHDL Code

After you create or edit your VHDL source files, compile them. As part of this tutorial, compile `modsimrand.vhd`. One way of compiling the file is to click the file name in the project workspace and select **Compile > Compile All**. An alternative is to specify `modsimrand.vhd` with the `vcom` command, as follows:

```
ModelSim> vcom modsimrand.vhd
```

If the compilation succeeds, messages appear in the command window and the compiler populates the work library with the compilation results.



```

Transcript
ModelSim> vcom modsimrand.vhd
# Model Technology ModelSim SE vcom 6.5 Compiler 2009.01 Jan 22 2009
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling entity modsimrand
# -- Compiling architecture behavioral of modsimrand
ModelSim> ]

```

Develop MATLAB Function

The HDL Verifier software verifies HDL hardware in MATLAB as a function. Typically, at this point you would create or edit a MATLAB function that meets HDL Verifier requirements. For this tutorial, you do not need to develop the MATLAB test bench function yourself. Instead, open and examine the existing file `modsimrand_plot.m`.

If you choose not to examine the HDL code at this time, skip to “Load Simulation” on page 2-24.

Note `modsimrand_plot.m` is a lower-level component of the MATLAB Random Number Generator example. Plotting code within `modsimrand_plot.m` is not discussed in the next section. This tutorial focuses only on those parts of `modsimrand_plot.m` that are required for MATLAB to verify a VHDL model.

You can open `modsimrand_plot.m` in the MATLAB Editor. For example:

```
edit modsimrand_plot.m
```

While you are viewing the file, note the following:

- On line 1, you will find the MATLAB function name specified along with its required parameters:

```
function [iport,tnext] = modsimrand_plot(oport,tnow,portinfo)
```

This function definition is significant because it represents the communication channel between MATLAB and ModelSim. Note:

- When coding the function, you must define the function with two output parameters, `iport` and `tnext`, and three input parameters, `oport`, `tnow`, and `portinfo`. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.
- You can use the `iport` parameter to drive input signals instead of, or in addition to, using other signal sources, such as ModelSim `force` commands. Depending on your application, you might use any combination of input sources. However, if multiple sources drive signals to a single `iport`, you will need a resolution function to handle signal contention.
- On lines 22 and 23, you will find some parameter initialization:

```
tnext = [];  
iport = struct();
```

In this case, function outputs `iport` and `tnext` are initialized to empty values.

- When coding a MATLAB function for use with HDL Verifier, you need to know the types of the data that the test bench function receives from and needs to return to ModelSim and how HDL Verifier handles this data; see “Supported Data Types” on page 10-35. This function includes the following port data type definitions and conversions:
 - The entity defined for this tutorial consists of three input ports of type `STD_LOGIC` and an output port of type `STD_LOGIC_VECTOR`.
 - Data of type `STD_LOGIC_VECTOR` consists of a column vector of characters with one bit per character.
 - The interface converts scalar data of type `STD_LOGIC` to a character that matches the character literal for the corresponding enumerated type.

On line 62, the line of code containing `oport.dout` shows how the data that a MATLAB function receives from ModelSim might need to be converted for use in the MATLAB environment:

```
ud.buffer(cyc) = mvl2dec(oport.dout)
```

In this case, the function receives `STD_LOGIC_VECTOR` data on `oport`. The function `mvl2dec` converts the bit vector to a decimal value that can be used in arithmetic computations. “Supported Data Types” on page 10-35 provides a summary of the types of data conversions to consider when coding your own MATLAB functions.

- Feel free to browse through the rest of `modsimrand_plot.m`. When you are finished, go to “Load Simulation” on page 2-24.

Load Simulation

After you compile the VHDL source file, you are ready to load the model for simulation. This section explains how to load an instance of entity `modsimrand` for simulation:

- 1 Load the instance of `modsimrand` for verification. To load the instance, specify the `vsimmatlab` command as follows:

```
ModelSim> vsimmatlab modsimrand
```

The `vsimmatlab` command starts the ModelSim simulator, `vsim`, specifically for use with MATLAB. ModelSim displays a series of messages in the command window as it loads the entity's packages and architecture.


```
> force /modsimrand/clk 0 0 ns, 1 5 ns -repeat 10 ns
> force /modsimrand/clk_en 1
> force /modsimrand/reset 1 0, 0 50 ns
```

The first command forces the `clk` signal to value 0 at 0 nanoseconds and to 1 at 5 nanoseconds. After 10 nanoseconds, the cycle starts to repeat every 10 nanoseconds. The second and third force commands set `clk_en` to 1 and `reset` to 1 at 0 nanoseconds and to 0 at 50 nanoseconds.

The ModelSim environment is ready to run a simulation. Now, you need to set up the MATLAB function.

Run Simulation

This section explains how to start and monitor this simulation, and rerun it, if you desire. When you have completed as many simulation runs as desired, shut down the simulation as described in the next section.

Running the Simulation for the First Time

Before running the simulation for the first time, you must verify the client connection. You may also want to set breakpoints for debugging.

Perform the following steps:

- 1 Open ModelSim and MATLAB windows.
- 2 In MATLAB, verify the client connection by calling `hdldaemon` with the `'status'` option:

```
hdldaemon('status')
```

This function returns a message indicating a connection exists:

```
HDLDaemon socket server is running on port 4795 with 1 connection
```

Or

```
HDLDaemon shared memory server is running with 1 connection
```

Note If you attempt to run the simulation before starting the `hdldaemon` in MATLAB, you will receive the following warning:

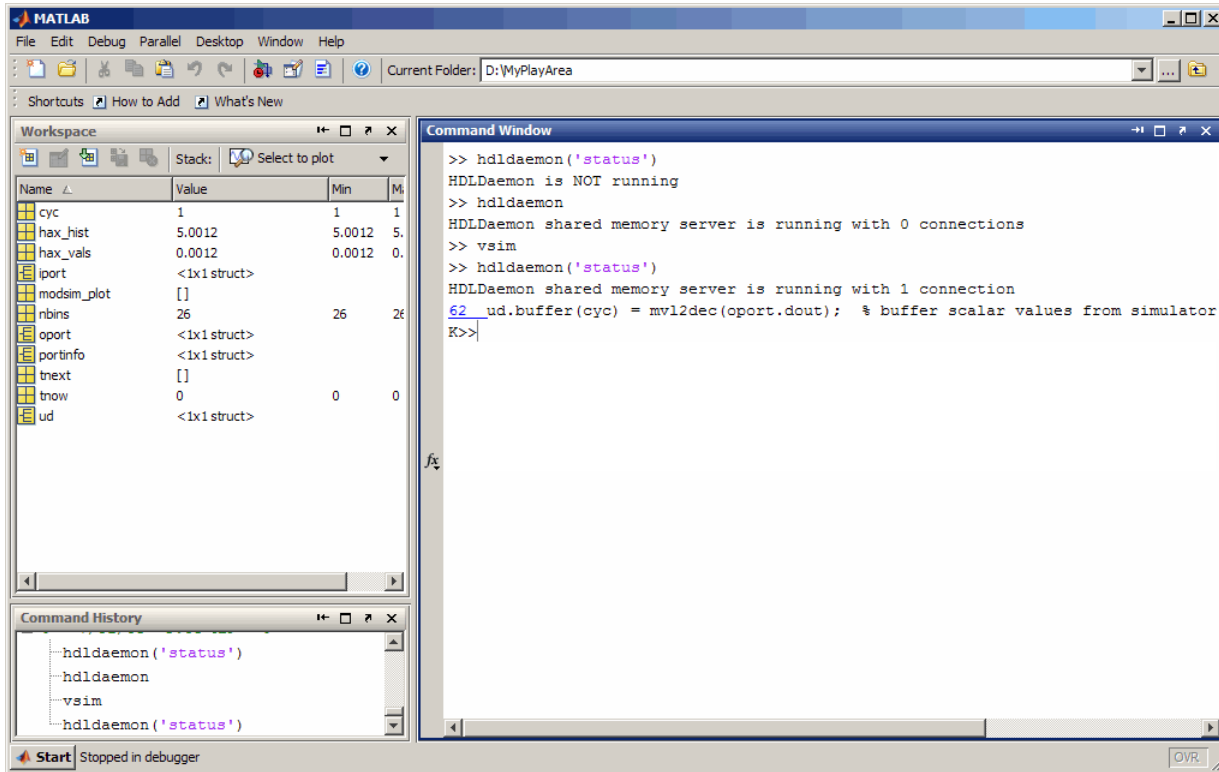
```
#ML Warn - MATLAB server not available (yet),
  The entity 'modsimrand' will not be active
```

- 3 Open `modsimrand_plot.m` in the MATLAB Editor.
- 4 Search for `oport.dout` and set a breakpoint at that line by clicking next to the line number. A red breakpoint marker will appear.
- 5 Return to ModelSim and enter the following command in the command window:

```
> run 80000
```

This command instructs ModelSim to advance the simulation 80,000 time steps (80,000 nanoseconds using the default time step period). Because you previously set a breakpoint in `modsimrand_plot.m`, however, the simulation runs in MATLAB until it reaches the breakpoint.

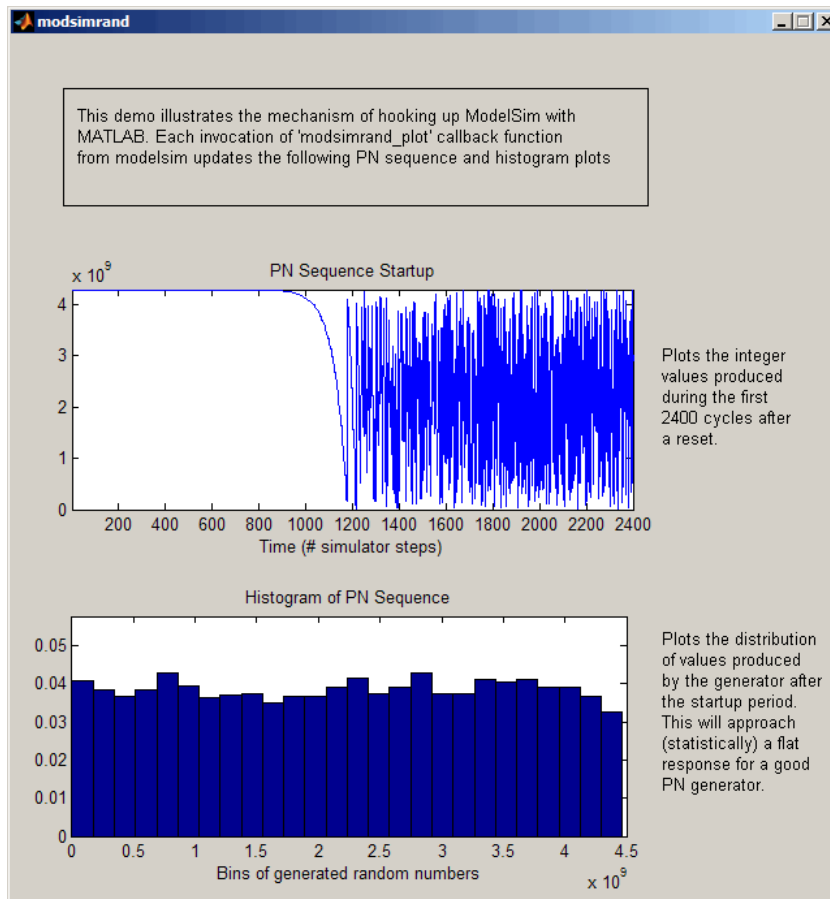
ModelSim is now blocked and remains blocked until you explicitly unblock it. While the simulation is blocked, note that MATLAB displays the data that ModelSim passed to the MATLAB function in the **Workspace** window.



In ModelSim, an empty figure window opens. You can use this window to plot data generated by the simulation.

- 6 Examine `oport`, `portinfo`, and `tnow` by hovering over these arguments inside the MATLAB Editor. Observe that `tnow`, the current simulation time, is set to 0. Also notice that, because the simulation has reached a breakpoint during the first call to `modsimrand_plot`, the `portinfo` argument is visible in the MATLAB workspace.
- 7 Click **Continue** in the MATLAB Editor. The next time the breakpoint is reached, notice that `portinfo` no longer appears in the MATLAB workspace. The `portinfo` function does not show because it is passed in only on the first function invocation. Also note that the value of `tnow` advances from 0 to 5e-009.
- 8 Clear the breakpoint by clicking the red breakpoint marker.
- 9 Unblock ModelSim and continue the simulation by clicking **Continue** in the MATLAB Editor.

The simulation runs to completion. As the simulation progresses, it plots generated data in a figure window. When the simulation completes, the figure window appears as shown here.



The simulation runs in MATLAB until it reaches the breakpoint that you just set. Continue the simulation/debugging session as desired.

Rerunning the Simulation

If you want to run the simulation again, you must restart the simulation in ModelSim, reinitialize the clock, and reset input signals. To do so:

- 1 Close the figure window.
- 2 Restart the simulation with the following command:

```
> restart
```

The **Restart** dialog box appears. Leave all the options enabled, and click **Restart**.

Note The **Restart** button clears the simulation context established by a `matlabtb` command. Thus, after restarting ModelSim, you must reissue the previous command or issue a new command.

- 3 Reissue the `matlabtb` command in the HDL simulator.

```
> matlabtb modsimrand -mfunc modsimrand_plot -rising /modsimrand/clock -socket portnum
```

- 4 Open `modsimrand_plot.m` in the MATLAB Editor.

- 5 Set a breakpoint at the same line as in the previous run.
- 6 Return to ModelSim and re-enter the following commands to reinitialize clock and input signals:

```
> force /modsimrand/clk 0 0,1 5 ns -repeat 10 ns
> force /modsimrand/clk_en 1
> force /modsimrand/reset 1 0, 0 50 ns
```
- 7 Enter a command to start the simulation, for example:

```
> run 80000
```

Shut Down Simulation

This section explains how to shut down a simulation in an orderly way.

In ModelSim, perform the following steps:

- 1 Stop the simulation on the client side by selecting **Simulate > End Simulation** or entering the `quit` command.
- 2 Quit ModelSim.

In MATLAB, you can just quit the application, which will shut down the simulation and also close MATLAB.

To shut down the server without closing MATLAB, you have the option of calling `hdldaemon` with the `'kill'` option:

```
hdldaemon('kill')
```

The following message appears, confirming that the server was shut down:

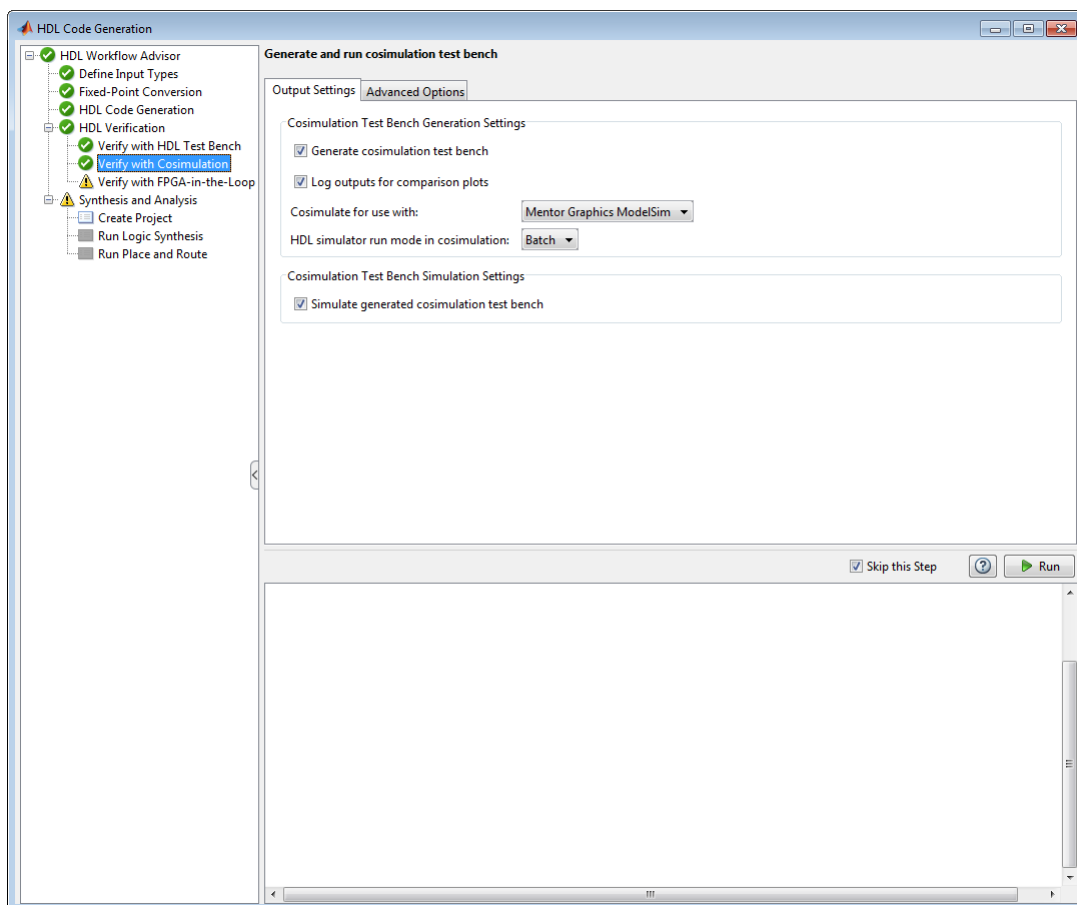
```
HDLDaemon server was shutdown
```

Automatic Verification of Generated HDL Code from MATLAB

The automatic verification feature integrates verification as part of the workflow for HDL cosimulation using the HDL Workflow Advisor. During this workflow, MATLAB generates a test bench for HDL cosimulation. This test bench compares the generated HDL DUT outputs (from the generated `hdlverifier.HDLCosimulation` System object™) with the original MATLAB function outputs. This step automatically runs this test bench and returns pass/fail information. If the outputs of the HDL DUT match the output of the original MATLAB function in the test bench, the test passes.

This feature requires an HDL Coder™ and an HDL Verifier license.

- 1 Start the MATLAB HDL Workflow Advisor.
- 2 Expand **HDL Verification** on the left pane and click **Verify with Cosimulation**.



- 3 Select **Generate HDL test bench** to instruct HDL Coder to generate HDL test bench code from your MATLAB test script (optional).
- 4 Select **Log outputs for comparison plots** if you would like to log and plot outputs of the reference design function and HDL simulator (optional).
- 5 For **Cosimulate for use with**, select Mentor Graphics ModelSim, Cadence Incisive, or Xilinx Vivado Simulator as the HDL simulator you want for cosimulation.
- 6 For HDL simulator run mode in cosimulation, select Batch mode for non-interactive simulation. Select GUI mode to view waveforms (not available for cosimulation with Vivado).

- 7 Select **Simulate generated cosimulation test bench** to automatically verify the generated HDL code in a cosimulation test bench.
- 8 For **Advanced Options**, select and set the optional parameters according to the descriptions in the following table.

Parameter	Description
Clock high time (ns)	Specify the number of nanoseconds the clock is high.
Clock low time (ns)	Specify the number of nanoseconds the clock is low.
Hold time (ns)	Specify the hold time for input signals and forced reset signals.
Clock enable delay (in clock cycles)	Specify time (in clock cycles) between deassertion of reset and assertion of clock enable.
Reset length (in clock cycles)	Specify time (in clock cycles) between assertion and deassertion of reset.

- 9 Optionally, select **Skip this step** if you don't want to verify with cosimulation.
- 10 Click **Run**.

If you selected Batch mode, a command window appears to launch the HDL simulator and run the cosimulation. This window is closed programmatically. If you selected GUI mode, the HDL simulator is opened and left open after simulation so that you may examine the waveforms and other signal data.

If there are errors, those messages appear in the message pane. Correct any errors and click **Run**.

HDL Cosimulation Using MATLAB Component Function

- “Create a MATLAB Component Function” on page 3-2
- “Set Up Cosimulation Component” on page 3-8

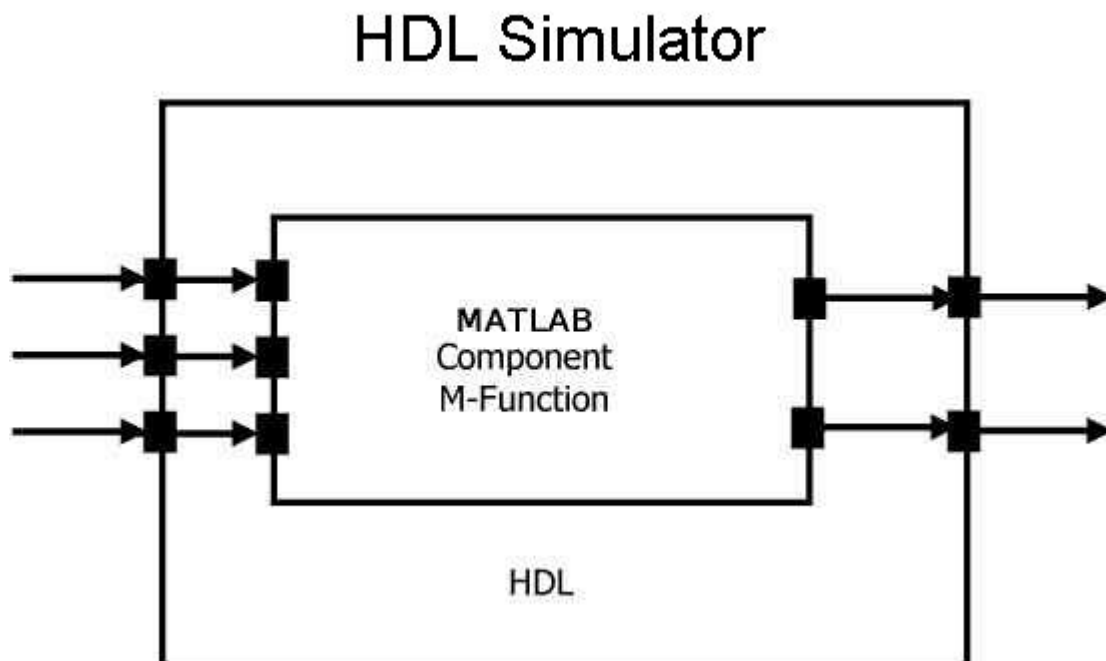
Create a MATLAB Component Function

The HDL Verifier software provides a means for visualizing HDL components within the MATLAB environment. You do so by coding an HDL model and a MATLAB function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB component functions that communicate with the HDL simulator.

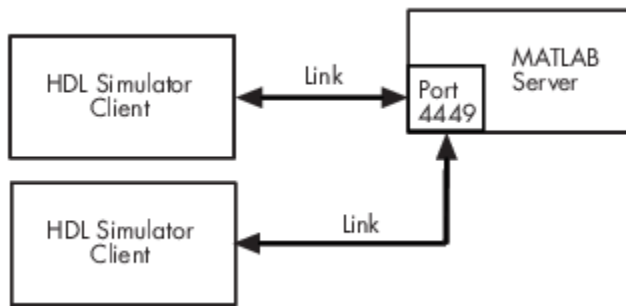
Note This workflow is not supported for Vivado cosimulation.

MATLAB component functions simulate the behavior of components in the HDL model. A stub module (providing port definitions only) in the HDL model passes its input signals to the MATLAB component function. The MATLAB component processes this data and returns the results to the outputs of the stub module. A MATLAB component typically provides some functionality (such as a filter) that is not yet implemented in the HDL code.

The following figure shows how an HDL simulator wraps around a MATLAB component function and how MATLAB communicates with the HDL simulator during a component simulation session.



When linked with MATLAB, the HDL simulator functions as the client, with MATLAB as the server. The following figure shows a multiple-client scenario connecting to the server at TCP/IP socket port 4449.



The MATLAB server can service multiple simultaneous HDL simulator sessions and HDL modules. However, you should follow recommended guidelines to help the server track the I/O associated with each module and session. The MATLAB server, which you start with the supplied function `hdl_daemon`, waits for connection requests from instances of the HDL simulator running on the same or different computers. When the server receives a request, it executes the specified MATLAB function you have coded to perform tasks on behalf of a module in your HDL design. Parameters that you specify when you start the server indicate whether the server establishes shared memory or TCP/IP socket communication links.

Refer to “Cosimulation Configurations” on page 10-2 for valid machine configurations.

Note The programming, interfacing, and scheduling conventions for test bench functions and component functions are virtually identical. For the most part, the same procedures apply to both types of functions.

Follow these workflow steps to create a MATLAB component function for cosimulation with the HDL simulator.

- 1 Create HDL module. Compile, elaborate, and simulate model in HDL simulator . See “Write HDL Modules for MATLAB Visualization” on page 3-3.
- 2 Create component MATLAB function. See “Write a Component Function” on page 3-6.
- 3 “Set Up MATLAB-HDL Simulator Connection” on page 1-2.
- 4 Place component function on MATLAB search path. See “Place Component Function on MATLAB Search Path” on page 3-8.
- 5 Bind HDL instance with component function using `matlabcp`. See “Bind Component Function Calls With `matlabcp`” on page 3-8.
- 6 Add scheduling options. See “Schedule Options for a Component Session” on page 3-11.
- 7 Set breakpoints for interactive HDL debug (optional).
- 8 Run cosimulation from HDL simulator. See “Run MATLAB-HDL Cosimulation” on page 1-4.

Write HDL Modules for MATLAB Visualization

- “Coding HDL Modules for Visualization with MATLAB” on page 3-4
- “Choose HDL Module Name for Use with MATLAB Component Function” on page 3-4
- “Specify Port Direction Modes in HDL Module for Use with Component Functions” on page 3-4
- “Specify Port Data Types in HDL Modules for Use with Component Functions” on page 3-4

- “Compile and Elaborate HDL Design for Use with Component Functions” on page 3-5

Coding HDL Modules for Visualization with MATLAB

The most basic element of communication in the HDL Verifier interface is the HDL module. The interface passes all data between the HDL simulator and MATLAB as port data. The HDL Verifier software works with any existing HDL module. However, when you code an HDL module that is targeted for MATLAB verification, you should consider its name, the types of data to be shared between the two environments, and the direction modes. The sections within this chapter cover these topics.

The process for coding HDL modules for MATLAB visualization is as follows:

- “Choose HDL Module Name for Use with MATLAB Component Function” on page 3-4
- “Specify Port Direction Modes in HDL Module for Use with Component Functions” on page 3-4
- “Specify Port Data Types in HDL Modules for Use with Component Functions” on page 3-4
- “Compile and Elaborate HDL Design for Use with Component Functions” on page 3-5

Choose HDL Module Name for Use with MATLAB Component Function

Although not required, when naming the HDL module, consider choosing a name that also can be used as a MATLAB function name. (Generally, naming rules for VHDL or Verilog and MATLAB are compatible.) By default, HDL Verifier software assumes that an HDL module and its simulation function share the same name. See “Bind Test Bench Function Calls With matlabb” on page 2-13.

For details on MATLAB function-naming guidelines, see “MATLAB Programming Tips” on files and file names in the MATLAB documentation.

Specify Port Direction Modes in HDL Module for Use with Component Functions

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). The following table defines these three modes.

Use VHDL Mode...	Use Verilog Mode...	For Ports That...
IN	input	Represent signals that can be driven by a MATLAB function or Simulink test bench
OUT	output	Represent signal values that are passed to a MATLAB function or Simulink test bench
INOUT	inout	Represent bidirectional signals that can be driven by or pass values to a MATLAB function or Simulink test bench

Specify Port Data Types in HDL Modules for Use with Component Functions

Port Data Types for VHDL Entities

In your entity statement, you must define each port that you plan to test with MATLAB with a VHDL data type that is supported by the HDL Verifier software. The interface can convert scalar and array data of the following VHDL types to comparable MATLAB types:

- STD_LOGIC, STD_ULOGIC, BIT, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, and BIT_VECTOR
- INTEGER and NATURAL

- REAL
- TIME
- Enumerated types, including user-defined enumerated types and CHARACTER

The interface also supports all subtypes and arrays of the preceding types.

Note The HDL Verifier software does not support VHDL extended identifiers for the following components:

- Port and signal names used in cosimulation
- Enum literals when used as array indices of port and signal names used in cosimulation

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Modules

In your module definition, you must define each port that you plan to test with MATLAB with a Verilog port data type that is supported by the HDL Verifier software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note HDL Verifier software does not support Verilog escaped identifiers for port and signal names used in cosimulation. However, it does support simple identifiers for Verilog.

Compile and Elaborate HDL Design for Use with Component Functions

After you create or edit your HDL source files, use the HDL simulator compiler to compile and debug the code.

Compilation for ModelSim

You have the option of invoking the compiler from menus in the ModelSim graphic interface or from the command line with the `vcom` command. The following sequence of ModelSim commands creates and maps the design library `work` and compiles the VHDL file `modsimrand.vhd`:

```
ModelSim> vlib work
ModelSim> vmap work work
ModelSim> vcom modsimrand.vhd
```

The following sequence of ModelSim commands creates and maps the design library `work` and compiles the Verilog file `test.v`:

```
ModelSim> vlib work
ModelSim> vmap work work
ModelSim> vlog test.v
```

Note You should provide read/write access to the signals that are connecting to the MATLAB session for cosimulation. For higher performance, you want to provide access only to those signals used in

cosimulation. You can check read/write access through the HDL simulator—see HDL simulator documentation for details.

Compilation for Xcelium

The Cadence Xcelium simulator allows for 1-step and 3-step processes for HDL compilation, elaboration, and simulation. The following Xcelium simulator command compiles the Verilog file `test.v`:

```
sh> xmvlog -64bit test.v
```

The following Xcelium simulator command compiles and elaborates the Verilog design `test.v`, and then loads it for simulation, in a single step:

```
sh> xmvlog -64bit +gui +access+rwc +linedebug test.v
```

The following sequence of Xcelium simulator commands performs all the same processes in multiple steps:

```
sh> xmvlog -64bit -linedebug test.v
sh> xmelab -64bit -access+rwc test
sh> xmsim test
```

Note You should provide read/write access to the signals that are connecting to the MATLAB session for cosimulation. The previous example shows how to provide read/write access to all signals in your design. For higher performance, you want to provide access only to those signals used in cosimulation. See the description of the `+access` flag to `xmvlog` and the `-access` argument to `xmelab` for details.

For more examples, see the HDL Verifier tutorials and demos. For details on using the HDL compiler, see the simulator documentation.

Write a Component Function

- “Overview to Coding an HDL Verifier Component Function” on page 3-6
- “Syntax of a Component Function” on page 3-7

Overview to Coding an HDL Verifier Component Function

Coding a MATLAB function that is to visualize an HDL module or component requires that you follow specific coding conventions. You must also understand the data type conversions that occur, and program data type conversions for operating on data and returning data to the HDL simulator.

To code a MATLAB function that is to verify an HDL module or component, perform the following steps:

- 1 Learn the syntax for a MATLAB HDL Verifier component function. See “Syntax of a Component Function” on page 3-7.
- 2 Understand how HDL Verifier software converts data from the HDL simulator for use in the MATLAB environment. See “Supported Data Types” on page 10-35.
- 3 Choose a name for the MATLAB component function. See “Invoke MATLAB Component Function Command `matlabcp`” on page 3-8.

- 4 Define expected parameters in the component function definition line. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.
- 5 Determine the types of port data being passed into the function. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.
- 6 Extract and, if applicable to the simulation, apply information received in the `portinfo` structure. See “Gaining Access to and Applying Port Information” on page 10-24.
- 7 Convert data for manipulation in the MATLAB environment, as applicable. See “Converting HDL Data to Send to MATLAB or Simulink” on page 10-35.
- 8 Convert data that needs to be returned to the HDL simulator. See “Converting Data for Return to the HDL Simulator” on page 10-40.

For more tips, see “Test Bench and Component Function Writing” on page 10-19.

Syntax of a Component Function

The syntax of a MATLAB component function is

```
function [iport, tnext] = MyFunctionName(oport, tnow, portinfo)
```

The input/output arguments, `iport` and `oport`, for a MATLAB component function are the reverse of the port arguments for a MATLAB test bench function. That is, the MATLAB component function returns signal data to the *outputs* and receives data from the *inputs* of the associated HDL module.

Initialize the function outputs to empty values at the beginning of the function as in the following example:

```
tnext = [];  
oport = struct();
```

See “MATLAB Function Syntax and Function Argument Definitions” on page 10-22 for an explanation of each of the function arguments. For more information on using `tnext` and `tnow` for simulation scheduling with `matlabcp`, see “Schedule Component Functions Using the `tnext` Parameter” on page 3-11.

See Also

More About

- “Test Bench and Component Function Writing” on page 10-19

Set Up Cosimulation Component

Set up a MATLAB component to cosimulate your HDL module. This workflow is not supported for Vivado cosimulation.

Place Component Function on MATLAB Search Path

- “Use MATLAB which Function to Find Component Function” on page 3-8
- “Add Component Function to MATLAB Search Path” on page 3-8

Use MATLAB which Function to Find Component Function

The MATLAB function that you are associating with an HDL component must be on the MATLAB search path or reside in the current working folder (see the MATLAB `cd` function). To verify whether the function is accessible, use the MATLAB `which` function. The following call to `which` checks whether the function `MyVhdlFunction` is on the MATLAB search path, for example:

```
which MyVhdlFunction  
/work/xcelium/MySym/MyVhdlFunction.m
```

If the specified function is on the search path, `which` displays the complete path to the function. If the function is not on the search path, `which` informs you that the file was not found.

Add Component Function to MATLAB Search Path

To add a MATLAB function to the MATLAB search path:

- On the **Home** tab, in the **Environment** section, click **Set Path**.
- Use the `addpath` function.
- For temporary access, change the MATLAB working folder to a desired location with the `cd` command.

Bind Component Function Calls With `matlabcp`

- “Invoke MATLAB Component Function Command `matlabcp`” on page 3-8
- “Specify HDL Signal/Port and Module Paths for MATLAB Component Function Cosimulation” on page 3-9
- “Bind HDL Module Component to MATLAB Component Function” on page 3-10

Invoke MATLAB Component Function Command `matlabcp`

You invoke `matlabcp` by issuing the command in the HDL simulator. See the Examples section of the `matlabcp` reference page for several examples of invoking `matlabcp`.

Be sure to follow the path specifications for MATLAB component function sessions when invoking `matlabcp`, as explained in “Specify HDL Signal/Port and Module Paths for MATLAB Component Function Cosimulation” on page 3-9.

For instructions in issuing the `matlabcp` command, see “Run MATLAB-HDL Cosimulation” on page 1-4.

Specify HDL Signal/Port and Module Paths for MATLAB Component Function Cosimulation

HDL Verifier software has specific requirements for specifying HDL design hierarchy, the syntax of which is described in the following sections: one for Verilog at the top level, and one for VHDL at the top level. Do not use a file name hierarchy in place of the design hierarchy name.

The rules stated in this section apply to signal/port and module path specifications for MATLAB cosimulation sessions. Other specifications may work but the HDL Verifier software does not officially recognize nor support them.

In the following example:

```
matlabtb u_osc_filter -mfunc oscfilter
```

`u_osc_filter` is the top-level component. If you specify a subcomponent, you must follow valid module path specifications for MATLAB cosimulation sessions.

Path Specifications for MATLAB Link Sessions with Verilog Top Level

- The path specification must start with a top-level module name.
- The path specification can include "." or "/" path delimiters, but it cannot include mixed delimiters.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/top/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

:

:sub

Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Path Specifications for MATLAB Link Sessions with VHDL Top Level

- The path specification can include the top-level module name, but you do not have to include it.
- The path specification can include "." or "/" path delimiters, but it cannot include mixed delimiters.
- The leaf module or signal must match the HDL language of the top-level module.

Examples for ModelSim and Xcelium Users

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

:

:sub

Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Bind HDL Module Component to MATLAB Component Function

By default, the HDL Verifier software assumes that the name for a MATLAB function matches the name of the HDL module that the function verifies. When you create a test bench or component function that has a different name than the design under test, you must associate the design with the MATLAB function using the `-mfunc` argument to `matlabtb`. This argument associates the HDL module instance to a MATLAB function that has a different name from the HDL instance.

For more information on the `-mfunc` argument and for a full list of `matlabtb` parameters, see the `matlabtb` function reference.

For details on MATLAB function naming guidelines, see "MATLAB Programming Tips" on files and file names in the MATLAB documentation.

Example of Binding Test Bench and Component Function Calls

In this first example, you form an association between the `inverter_vl` component and the MATLAB test bench function `inverter_tb` by invoking the function `matlabtb` with the `-mfunc` argument when you set up the simulation.

```
matlabtb inverter_vl -mfunc inverter_tb
```

The `matlabtb` command instructs the HDL simulator to call back the `inverter_tb` function when `inverter_vl` executes in the simulation.

In this second example, you bind the model `osc_top.u_osc_filter` to the component function `oscfilter`:

```
matlabcp osc_top.u_osc_filter -mfunc oscfilter
```

When the HDL simulator calls the `oscfilter` callback, the function knows to operate on the model `osc_top.u_osc_filter`.

Schedule Options for a Component Session

- “About Scheduling Options for Component Sessions” on page 3-11
- “Schedule Component Session Using `matlabcp` Arguments” on page 3-11
- “Schedule Component Functions Using the `tnext` Parameter” on page 3-11

About Scheduling Options for Component Sessions

There are two ways to schedule the invocation of a MATLAB function:

- Using the arguments to the HDL Verifier function `matlabtb` or `matlabcp`
- Inside the MATLAB function using the `tnext` parameter

The two types of scheduling are not mutually exclusive. You can combine the `matlabtb` or `matlabcp` timing arguments and the `tnext` parameter of a MATLAB function to schedule test bench or component session callbacks.

Schedule Component Session Using `matlabcp` Arguments

By default, the HDL Verifier software invokes a MATLAB test bench or component function once (at the time that you make the call to `matlabtb` or `matlabcp`). If you want to apply more control, and execute the MATLAB function more than once, use the command scheduling options. With these options, you can specify when and how often the HDL Verifier software invokes the relevant MATLAB function. If applicable, modify the function or specify timing arguments when you begin a MATLAB test bench or component function session with the `matlabtb` or `matlabcp` function.

You can schedule a MATLAB test bench or component function to execute using the command arguments under any of the following conditions:

- **Discrete time values**—Based on time specifications that can also include repeat intervals and a stop time
- **Rising edge**—When a specified signal experiences a rising edge
 - VHDL: Rising edge is {0 or L} to {1 or H}.
 - Verilog: Rising edge is the transition from 0 to x, z, or 1, and from x or z to 1.
- **Falling edge**—When a specified signal experiences a falling edge
 - VHDL: Falling edge is {1 or H} to {0 or L}.
 - Verilog: Falling edge is the transition from 1 to x, z, or 0, and from x or z to 0.
- **Signal state change**—When a specified signal changes state, based on a list using the `sensitivity` argument to `matlabtb`.

Schedule Component Functions Using the `tnext` Parameter

You can control the callback timing of a MATLAB function by using that function's `tnext` parameter. This parameter passes a time value to the HDL simulator, and the value gets added to the simulation schedule for that function. If the function returns a null value (`[]`), the software does not add any new entries to the schedule.

You can set the value of `tnext` to a value of type `double` or `int64`. Specify `double` to express the callback time in seconds. For example, to schedule a callback in 1 ns, specify:

```
tnext = 1e-9
```

Specify `int64` to convert to an integer multiple of the current HDL simulator time resolution limit. For example: if the HDL simulator time precision is 1 ns, to schedule a callback at 100 ns, specify:

```
tnext=int64(100)
```

Note The `tnext` parameter represents time from the start of the simulation. Therefore, `tnext` must always be greater than `tnow`. If it is less, the software does not schedule a callback.

For more information on `tnext` and the function prototype, see “MATLAB Function Syntax and Function Argument Definitions” on page 10-22.

Examples of Scheduling with `tnext`

In this first example, each time the HDL simulator calls the test bench function (via HDL Verifier), `tnext` schedules the next callback to the MATLAB function for 1 ns later, relative to the current simulation time:

```
tnext = [];  
.  
.  
.  
tnext = tnow+1e-9;
```

Using `tnext` you can dynamically decide the callback scheduling based on criteria specific to the operation of the test bench. For example, you can decide to stop scheduling callbacks when a data signal has a certain value:

```
if qsum == 17,  
    qsum = 0;  
    disp('done');  
    tnext = []; % suspend callbacks  
    testisdone = 1;  
    return;  
end
```

This next example demonstrates scheduling a component session using `tnext`. In the Oscillator example, the `oscfilter` function calculates a time interval at which the HDL simulator calls the callbacks. The component function calculates this interval on the first call to `oscfilter` and stores the result in the variable `fastestrate`. The variable `fastestrate` represents the sample period of the fastest oversampling rate supported by the filter. The function derives this rate from a base sampling period of 80 ns.

The following assignment statement sets the timing parameter `tnext`. This parameter schedules the next callback to the MATLAB component function, relative to the current simulation time (`tnow`).

```
tnext = tnow + fastestrate;
```

The function returns a new value for `tnext` each time the HDL simulator calls the function.

HDL Cosimulation Using MATLAB System Object

Create a MATLAB System Object

You can verify HDL modules using the HDL Cosimulation System object. You can use the System object as a test bench or you can use it to represent a component still under design.

The easiest way to create a test bench System object is to use the Cosimulation Wizard with existing HDL code. When cosimulating with Xcelium or ModelSim, you can also create an HDL Cosimulation System object manually.

You can find out more about the **Cosimulation Wizard** and creating System objects within these topics:

- For an example of how to use the HDL Cosimulation System object, see “Verify Viterbi Decoder Using MATLAB System Object and HDL Simulator” on page 32-145.
- For an example of converting existing HDL code to a System object test bench, see “Prepare to Import HDL Code for Cosimulation” on page 9-2.
- For general information about how to use System objects, see “System Design and Simulation in MATLAB”

See Also

`hdlverifier.HDLCosimulation` | `hdlverifier.VivadoHDLCosimulation`

Set Up and Run Simulation for Simulink

Start HDL Simulator for Cosimulation in Simulink

If you are using ModelSim or Xcelium simulators, use this workflow to start the HDL simulator from the MATLAB prompt.

If you are using the Vivado simulator, the HDL Cosimulation block loads and executes the compiled design for cosimulation, and there is no need to separately start the HDL simulator.

Start HDL Simulator from MATLAB

Start the HDL simulator directly from MATLAB by calling the HDL Verifier function `vsim` or `nclaunch`.

```
>>vsim
```

Note that if both tools (MATLAB and the HDL simulator) are not running on the same system, you must start the HDL simulator manually and load the HDL Verifier libraries yourself. See “Cosimulation Libraries” on page 10-9.

You can call `vsim` or `nclaunch` with additional parameters; see the reference pages for details.

You must make sure the HDL simulator executables — also called `vsim` (ModelSim) and `nclaunch` (Cadence Xcelium) — are on the system path. See your system documentation for instruction on setting environment variables.

Linux Users Make sure the HDL simulator executable is still on the system path after the shell is launched from MATLAB. If it is not, make sure the shell startup file does not remove it from the path environment variable.

When using Vivado simulator for cosimulation, there is no need to start the HDL simulator separately, since the cosimulation executes as a single process with a shared DLL file.

Load Instance of HDL Module for Cosimulation

Xcelium users load an instance of the HDL module for cosimulation using the `hdlsimulink` function. ModelSim users do the same using the `vsimulink` function.

Example of loading HDL Module instance — Xcelium users

After you start the HDL simulator from MATLAB, load an instance of an HDL module for cosimulation with the function `hdlsimulink`. Issue the command for each instance of an HDL module in your model that you want to cosimulate.

For example:

```
hdlsimulink work.manchester
```

Example of loading HDL Module instance — ModelSim users

After you start the HDL simulator from MATLAB, load an instance of an HDL module for cosimulation with the function `vsimulink`. Issue the command for each instance of an HDL module in your model that you want to cosimulate.

For example:

```
vsimulink work.manchester
```

This command opens a simulation workspace for `manchester` and displays a series of messages in the HDL simulator command window as the simulator loads the packages and architectures for the HDL module.

Run a Simulink Cosimulation Session

In this section...

“Set Simulink Model Configuration Parameters” on page 5-4

“Determine Available Socket Port Number” on page 5-4

“Check Connection Status” on page 5-4

“Run and Test Cosimulation Model” on page 5-5

“Set Parameters from a Tcl Script” on page 5-7

“Avoid Race Conditions in HDL Simulation with Test Bench Cosimulation and the HDL Verifier HDL Cosimulation Block” on page 5-8

Set Simulink Model Configuration Parameters

When you create a Simulink model that includes one or more HDL Verifier Cosimulation blocks, you might want to adjust certain Simulink parameter settings to best meet the needs of HDL modeling. For example, you might want to adjust the value of the **Stop time** parameter in the **Solver** pane of the Model Configuration Parameters dialog box.

You can adjust the parameters individually or you can use DSP System Toolbox™ Simulink model templates to automatically configure the Simulink environment with the recommended settings for digital signal processing modeling.

Parameter	Default Setting
'SingleTaskRateTransMsg'	'error'
'Solver'	'fixedstepdiscrete'
'EnableMultiTasking'	'off'
'StartTime'	'0.0'
'StopTime'	'inf'
'FixedStep'	'auto'
'SaveTime'	'off'
'SaveOutput'	'off'
'AlgebraicLoopMsg'	'error'

The default settings for SaveTime and SaveOutput improve simulation performance.

For more information on DSP System Toolbox Simulink model templates, see the DSP System Toolbox documentation.

Determine Available Socket Port Number

To determine an available socket number use: `ttcp -a` a shell prompt.

Check Connection Status

To check the connection status, on the **Modeling** tab, in the **Compile** section, click **Update Model**. If you have an error in the connection, Simulink will notify you.

The MATLAB command `pingHdlSim` can also be used to check the connection status. If a `-1` is returned, then there is no connection with the HDL simulator.

Run and Test Cosimulation Model

In general, the last stage of cosimulation is to run and test your model. There are some steps you must be aware of when changing your model during or between cosimulation sessions. You can run the cosimulation in one of three ways:

- “Cosimulation Using the Simulink and HDL Simulator GUIs” on page 5-5
- “Cosimulation with Simulink Using the Command Line Interface (CLI)” on page 5-5
- “Cosimulation with Simulink Using Batch Mode” on page 5-6

Cosimulation Using the Simulink and HDL Simulator GUIs

Start the HDL simulator and load your HDL design. For test bench cosimulation, begin simulation first in the HDL simulator. Then, in Simulink, in the **Simulation** tab, click **Run**. Simulink runs the model and displays any errors that it detects. You can alternate between the HDL simulator and Simulink GUIs to monitor the cosimulation results.

For component cosimulation, start the simulation in Simulink first, then begin simulation in the HDL simulator.

You can specify "GUI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command, but since using the GUI is the default mode for HDL Verifier, you do not have to.

Cosimulation with Simulink Using the Command Line Interface (CLI)

Running your cosimulation session using the command-line interface allows you to interact with the HDL simulator during cosimulation, which can be helpful for debugging.

To use the CLI, specify "CLI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command.

Caution Close the terminal window by entering `quit -f` at the command prompt. Do not close the terminal window by clicking the "X" in the upper right-hand corner. This causes a memory-type error to be issued from the system. This is not a bug with HDL Verifier but just the way the HDL simulator behaves in this context.

You can type CTRL+C to interrupt and terminate the simulation in the HDL simulator but this action also causes the memory-type error to be displayed.

Specify CLI mode with `nclaunch` (Cadence Xcelium)

Issue the `nclaunch` command with "CLI" as the `runmode` property value, as follows (example entered into the MATLAB editor):

```
tclcmd = { ['cd ',unixprojdir],...
           ['exec xmvlog -64bit -linedebug ',unixsrcfile1],...
           'exec xmelab -64bit -access +wc work.inverter_vl',...
           'hdlsimulink -gui work.inverter_vl'
```

```
};
```

```
nclaunch('tclstart',tclcmd,'runmode','CLI');
```

Specify CLI mode with vsim (Mentor Graphics ModelSim)

Issue the `vsim` command with "CLI" as the runmode property value, as follows (example entered into the MATLAB editor):

```
tclcmd = {'vlib work',...  
         'vlog addone_vlog.v add_vlog.v top_frame.v',...  
         'vsimulink top =socket 5002'};
```

```
vsim('tclstart',tclcmd,'runmode','CLI');
```

Cosimulation with Simulink Using Batch Mode

Running your cosimulation session in batch mode allows you to keep the process in the background, reducing demand on memory by disengaging the GUI.

To use the batch mode, specify "Batch" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command. After you issue the HDL Verifier HDL simulator launch command with batch mode specified, start the simulation in Simulink. To stop the HDL simulator before the simulation is completed, issue the `breakHdLSim` command.

Specify Batch mode with nclaunch (Cadence Xcelium)

Issue the `nclaunch` command with 'Batch' as the runmode parameter, as follows:

```
nclaunch('tclstart',manchestercmds,'runmode','Batch')
```

You can also set the runmode parameter to 'Batch with Xterm', which starts the HDL simulator in the background but shows the session in an Xterm.

Specify Batch mode with vsim (Mentor Graphics ModelSim)

On Windows, specifying batch mode causes ModelSim to be run in a non-interactive command window. On Linux, specifying batch mode causes ModelSim to be run in the background with no window.

Issue the `vsim` command with 'Batch' as the runmode parameter, as follows:

```
>> vsim('tclstart',manchestercmds,'runmode','Batch')
```

Test Cosimulation

If you wish to reset a clock during a cosimulation, you can do so in one of these ways:

- By entering HDL simulator force commands at the HDL simulator command prompt
- By specifying HDL simulator force commands in the **Post- simulation command** text field on the **Simulation** pane of the HDL Verifier Cosimulation block parameters dialog box.

See also "Clock, Reset, and Enable Signals" on page 10-55.

If you change any part of the Simulink model, including the HDL Cosimulation block parameters, update the model to reflect those changes. You can do this update in one of the following ways:

- Rerun the simulation
- On the **Modeling** tab, in the **Compile** section, click **Update Model**.

Set Parameters from a Tcl Script

You can create a Tcl script that lists the Tcl commands you want to execute on the HDL simulator, either pre- or post-simulation.

Tcl Scripts for ModelSim Users

You can create a ModelSim DO file that lists Tcl commands and then specify that file with the ModelSim `do` command as follows:

```
do mycosimstartup.do
```

or

```
do mycosimcleanup.do
```

You can include the `quit -f` command in an after-simulation Tcl command or DO file to force ModelSim to shut down at the end of a cosimulation session. Specify all after-simulation Tcl commands in a single cosimulation block, and place `quit` at the end of the command or DO file.

Except for `quit`, the command or DO file that you specify cannot include commands that load a ModelSim project or modify simulator state. For example, they cannot include commands such as `start`, `stop`, or `restart`.

Tcl Scripts for Xcelium Users

You can create an HDL simulator Tcl script that lists Tcl commands, and then specify that file with the HDL simulator `source` command as follows:

```
source mycosimstartup.script_extension
```

or

```
source mycosimcleanup.script_extension
```

You can include the `exit` command in an after-simulation Tcl script to force the HDL simulator to shut down at the end of a cosimulation session. Specify all after-simulation Tcl commands in a single cosimulation block and place `exit` at the end of the command or Tcl script.

Except for `exit`, the command or Tcl script that you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, they cannot include commands such as `run`, `stop`, or `reset`.

This example shows a Tcl script when the `-gui` argument was used with `hdlsimmatlab` or `hdlsimulink`:

```
after 1000 {xmsim -submit exit}
```

This example shows a Tcl exit script to use when the `-tcl` argument was used with `hdlsimmatlab` or `hdlsimulink`:

```
after 1000 {exit}
```

Avoid Race Conditions in HDL Simulation with Test Bench Cosimulation and the HDL Verifier HDL Cosimulation Block

In the HDL simulator, you cannot control the order in which clock signals (rising-edge or falling-edge) defined in the HDL Cosimulation block are applied, relative to the data inputs driven by these clocks. If you are careful to verify the relationship between the data and active edges of the clock, you can avoid race conditions that could create differing cosimulation results. See “Race Conditions in HDL Simulators” on page 10-33.

Simulink Test Bench for HDL Component

- “Simulink as a Test Bench” on page 6-2
- “Create a Simulink Cosimulation Test Bench” on page 6-6
- “Verify HDL Module with Simulink Test Bench” on page 6-27
- “Automatic Verification of Generated HDL Code from Simulink” on page 6-42

Simulink as a Test Bench

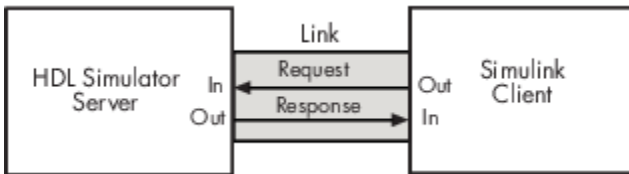
In this section...

“Communications During Test Bench Cosimulation” on page 6-2

“HDL Cosimulation Block Features for Test Bench Simulation” on page 6-4

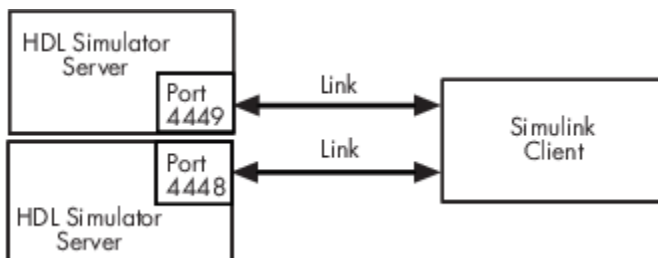
Communications During Test Bench Cosimulation

When you link the HDL simulator with a Simulink application, the simulator functions as the server, as shown in the following figure.



In this case, the HDL simulator responds to simulation requests it receives from cosimulation blocks in a Simulink model. You begin a cosimulation session from Simulink. After a session is started, you can use Simulink and the HDL simulator to monitor simulation progress and results. For example, you might add signals to a wave window to monitor simulation timing diagrams.

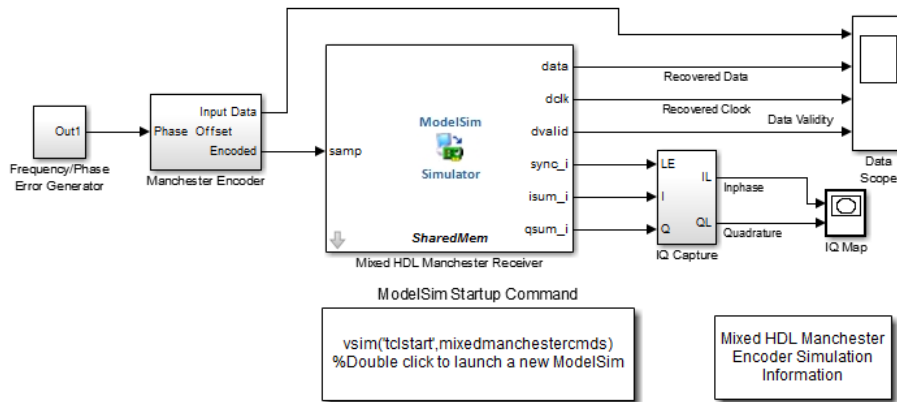
As the following figure shows, multiple cosimulation blocks in a Simulink model can request the service of multiple instances of the HDL simulator, using unique TCP/IP socket ports.



When you link the HDL simulator with a Simulink application, the simulator functions as the server. Using the HDL Verifier communications interface, an HDL Cosimulation block cosimulates a hardware component by applying input signals to and reading output signals from an HDL model under simulation in the HDL simulator.

Note Vivado users: Simulink executes the cosimulation as a single process, so there is no need to communicate with an HDL server.

This figure shows a sample Simulink model that includes an HDL Cosimulation block. The connection is using shared memory.



Copyright 2008-2009 The MathWorks, Inc.

The HDL Cosimulation block models a Manchester receiver that is coded in HDL. Other blocks and subsystems in the model include the following:

- Frequency Error Range block, Frequency Error Slider block, and Phase Event block
- Manchester encoder subsystem
- Data alignment subsystem
- Inphase/Quadrature (I/Q) capture subsystem
- Error Rate Calculation block from the Communications Toolbox™ software
- Bit Errors block
- Data Scope block
- Constellation Diagram block from the Communications Toolbox software

For information on getting started with Simulink software, see the Simulink online help or documentation.

How Simulink Drives Cosimulation Signals

Although you can bind the output ports of an HDL Cosimulation block to any signal in an HDL model hierarchy, you must use some caution when connecting signals to input ports. You want to verify that the signal you are binding to does not have other drivers. If it does, use resolved logic types; otherwise you may get unpredictable results.

If you need to use a signal that has multiple drivers and it is resolved (for example, it is of VHDL type `STD_LOGIC`), Simulink applies the resolution function at each time step defined by the signal's Simulink sample rate. Depending on the other drivers, the Simulink value may or may not get applied. Furthermore, Simulink has no control over signal changes that occur between its sample times.

Note Verify that signals used in cosimulation have read/write access. You can check read/write access through the HDL simulator—see HDL simulator documentation for details.

This rule applies to all signals on the **Ports**, **Clocks**, and **Simulation** panes and to signals added to the model in any other manner.

Multirate Signals During Test Bench Cosimulation

HDL Verifier software supports the use of multirate signals, signals that are sampled or updated at different rates, in a single HDL Cosimulation block. An HDL Cosimulation block exchanges data for each signal at the Simulink sample rate for that signal. For input signals, an HDL Cosimulation block accepts and honors all signal rates.

The HDL Cosimulation block also lets you specify an independent sample time for each output port. You must explicitly set the sample time for each output port, or accept the default. Using this setting lets you control the rate at which Simulink updates an output port by reading the corresponding signal from the HDL simulator.

Continuous Time Signals

Use the Simulink Zero-Order Hold block to apply a zero-order hold (ZOH) on continuous signals that are driven into an HDL Cosimulation block.

HDL Cosimulation Block Features for Test Bench Simulation

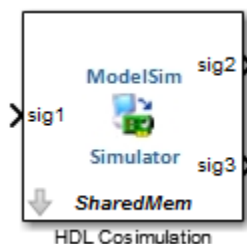
The HDL Verifier HDL Cosimulation block links hardware components that are concurrently simulating in the HDL simulator to the rest of a Simulink model.

You can link Simulink and the HDL simulator in two possible ways:

- As a single HDL Cosimulation block fitted into the framework of a larger system-oriented Simulink model.
- As a Simulink model made up of a collection of HDL Cosimulation blocks, each representing a specific hardware component.

The block mask contains panels for entering port and signal information, setting communication modes, adding clocks (Xcelium and ModelSim only), specifying pre- and post-simulation Tcl commands (Xcelium and ModelSim only), and defining the timing relationship.

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, you integrate the HDL representation into your Simulink model as an HDL Cosimulation block. There is one block for each supported HDL simulator. These blocks are located in the Simulink Library, within the HDL Verifier block library. As an example, the block for use with Mentor Graphics® ModelSim is shown in the next figure.



You configure an HDL Cosimulation block by specifying values for parameters in a block parameters dialog box. The HDL Cosimulation block parameters dialog box consists of tabbed panes that specify the following information:

- **Ports Pane:** Block input and output ports that correspond to signals, including internal signals, of your HDL design, and an output sample time.

- **Connection Pane:** Type of communication and related settings to be used for exchanging data between simulators.
- **Timescales Pane:** The timing relationship between Simulink software and the HDL simulator.
- **Clocks Pane** (Xcelium and ModelSim only): Optional rising-edge and falling-edge clocks to apply to your model.
- **Simulation Pane** (Xcelium and ModelSim only): Tcl commands to run before and after a simulation.

For more detail on each of these panes, see the HDL Cosimulation reference page.

Create a Simulink Cosimulation Test Bench

These steps describe how to cosimulate an HDL design using Simulink software as a test bench.

- 1 Create a Simulink test bench model by adding Simulink blocks from the Simulink block libraries. Run and test your model thoroughly before replacing or adding hardware model components as cosimulation blocks.
- 2 Code HDL module. Compile, elaborate, and simulate your module in your HDL simulator. See “Code an HDL Component” on page 6-6.
- 3 Start HDL simulator for use with MATLAB and Simulink and load HDL Verifier libraries. See “Start HDL Simulator for Cosimulation in Simulink” on page 5-2.
- 4 Add the HDL Cosimulation block to your Simulink test bench model. See “Insert HDL Cosimulation Block” on page 7-7.
- 5 Define HDL Cosimulation block interface. See “Configure HDL Cosimulation Block Interface” on page 6-8.
- 6 (Optional) Add the To VCD File block to log changes to variable values during a simulation session. See “Add a Value Change Dump (VCD) File” on page 8-2.
- 7 Start simulation in HDL simulator first, then run the Simulink model. See “Run a Simulink Cosimulation Session” on page 5-4.

Vivado users: replace steps 3-7 above with the following:

- 1 Use the **Cosimulation Wizard** to generate an HDL Cosimulation block for Vivado cosimulation. For details, see “Import HDL Code for HDL Cosimulation Block” on page 9-24.
- 2 Add the generated HDL Cosimulation block to your Simulink test bench model.
- 3 Start the simulation from Simulink. The HDL Cosimulation block loads the shared libraries and executes the cosimulation with Vivado.

Code an HDL Component

- “Specify Port Direction Modes in the HDL Component for Test Bench Use” on page 6-6
- “Specify Port Data Types in the HDL Component for Test Bench Use” on page 6-7
- “Compile and Elaborate HDL Design for Test Bench Use” on page 6-8

The HDL Verifier interface passes all data between the HDL simulator and Simulink as port data. The HDL Verifier software works with any existing HDL module. However, when you code an HDL module that is targeted for Simulink verification, you should consider the types of data to be shared between the two environments and the direction modes.

Specify Port Direction Modes in the HDL Component for Test Bench Use

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). The following table defines these three modes.

Use VHDL Mode...	Use Verilog Mode...	For Ports That...
IN	input	Represent signals that can be driven by a MATLAB function or Simulink test bench

Use VHDL Mode...	Use Verilog Mode...	For Ports That...
OUT	output	Represent signal values that are passed to a MATLAB function or Simulink test bench
INOUT	inout	Represent bidirectional signals that can be driven by or pass values to a MATLAB function or Simulink test bench

Specify Port Data Types in the HDL Component for Test Bench Use

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the HDL Verifier interface converts data types for the MATLAB environment, see “Supported Data Types” on page 10-35.

Note If you use unsupported types, the HDL Verifier software issues a warning and ignores the port at run time. For example, if you define your interface with five ports, one of which is a VHDL access port, at run time, then the interface displays a warning and your code sees only four ports.

Port Data Types for VHDL Entities

In your entity statement, you must define each port that you plan to test with MATLAB with a VHDL data type that is supported by the HDL Verifier software. The interface can convert scalar and array data of the following VHDL types to comparable MATLAB types:

- STD_LOGIC, STD_ULOGIC, BIT, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, and BIT_VECTOR
- INTEGER and NATURAL
- REAL
- TIME
- Enumerated types, including user-defined enumerated types and CHARACTER

The interface also supports all subtypes and arrays of the preceding types.

Note The HDL Verifier software does not support VHDL extended identifiers for the following components:

- Port and signal names used in cosimulation
- Enum literals when used as array indices of port and signal names used in cosimulation

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Entities

In your module definition, you must define each port that you plan to test with MATLAB with a Verilog port data type that is supported by the HDL Verifier software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer

- wire

Note HDL Verifier software does not support Verilog escaped identifiers for port and signal names used in cosimulation. However, it does support simple identifiers for Verilog.

Compile and Elaborate HDL Design for Test Bench Use

Refer to the HDL simulator documentation for instruction in compiling and elaborating the HDL design.

Configure HDL Cosimulation Block Interface

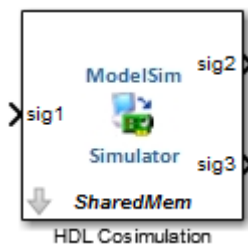
This workflow is an alternative to using the **Cosimulation Wizard** for configuring and generating an HDL Cosimulation block. For Vivado cosimulation, you must use the **Cosimulation Wizard**. See “Import HDL Code for HDL Cosimulation Block” on page 9-24.

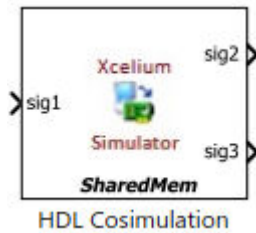
- “Insert HDL Cosimulation Block” on page 6-8
- “Connect Block Ports” on page 6-9
- “Open HDL Cosimulation Block Parameters” on page 6-9
- “Map HDL Signals to Block Ports” on page 6-9
- “Specify Signal Data Types” on page 6-19
- “Configure Simulink and HDL Simulator Timing Relationship” on page 6-19
- “Configure Communication Link in the HDL Cosimulation Block” on page 6-21
- “Specify Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box” on page 6-23
- “Programmatically Control Block Parameters” on page 6-25

Insert HDL Cosimulation Block

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, integrate the HDL representation into your Simulink model as an HDL Cosimulation block by performing the following steps:

- 1 Open your Simulink model, if it is not already open.
- 2 Delete the model component that the HDL Cosimulation block is to replace.
- 3 In the Simulink Library Browser, click the HDL Verifier block library. You can then select the block library for your supported HDL simulator. Select either the Mentor Graphics ModelSim HDL Cosimulation block, or the Cadence Xcelium HDL Cosimulation block, as shown below.





- 4 Copy the HDL Cosimulation block from the Library Browser to your model.

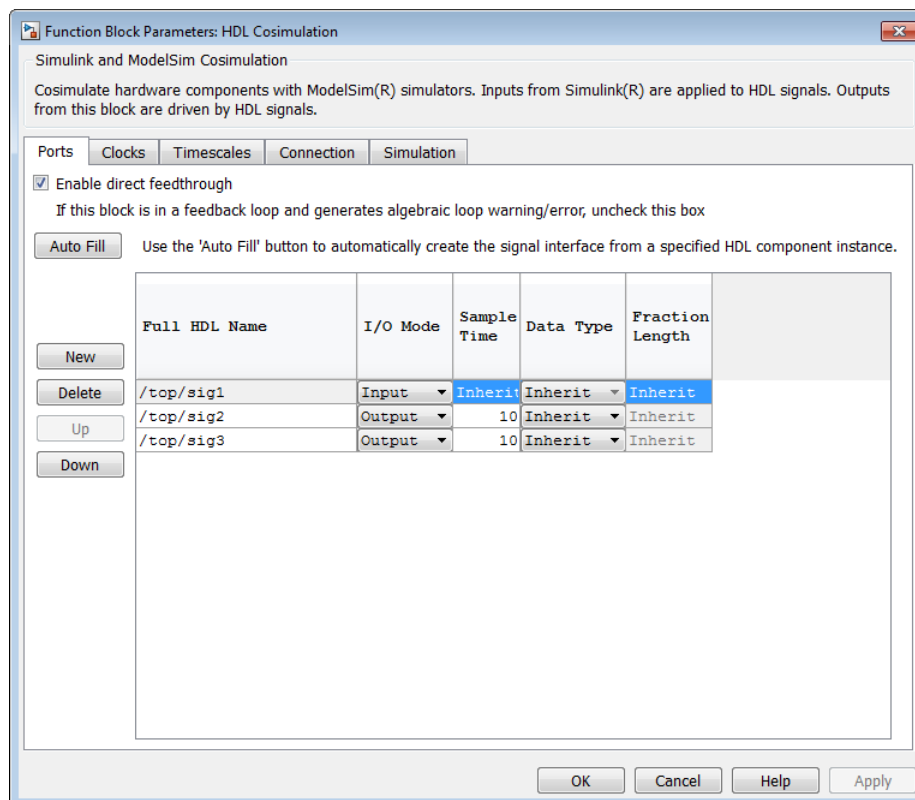
Connect Block Ports

Connect any HDL Cosimulation block ports to the applicable block ports in your Simulink model.

- To model a sink device, configure the block with inputs only.
- To model a source device, configure the block with outputs only.

Open HDL Cosimulation Block Parameters

To open the block parameters dialog box for the HDL Cosimulation block, double-click the block icon. Simulink displays the following Block Parameters dialog box (as an example, the dialog box for the HDL Cosimulation block for use with ModelSim is shown below).



Map HDL Signals to Block Ports

- “Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation” on page 6-10

- “Get Signal Information from HDL Simulator” on page 6-11
- “Enter Signal Information Manually” on page 6-15
- “Control Output Port Directly by Value of Input Port” on page 6-19

The first step to configuring your HDL Verifier HDL Cosimulation block is to map signals and signal instances of your HDL design to port definitions in your HDL Cosimulation block. In addition to identifying input and output ports, you can specify a sample time for each output port. You can also specify a fixed-point data type for each output port.

The signals that you map can be at any level of the HDL design hierarchy.

To map the signals, you can perform either of the following actions:

- Enter signal information manually into the **Ports** pane of the block parameters dialog box. This approach can be more efficient when you want to connect a small number of signals from your HDL model to Simulink.
- Use the **Auto Fill** button to have the HDL Cosimulation block obtain signal information for you by transmitting a query to the HDL simulator. This approach can save significant effort when you want to cosimulate an HDL model that has many signals that you want to connect to your Simulink model. However, in some cases, you will need to edit the signal data returned by the query.

Note Verify that signals used in cosimulation have read/write access. For higher performance, you want to provide access only to those signals used in cosimulation. This rule applies to all signals on the **Ports**, **Clocks**, and **Simulation** panes, and to all signals added in any other manner.

Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation

These rules are for signal/port and module path specifications in Simulink. Other specifications may work but are not explicitly or implicitly supported in this or future releases.

HDL designs generally do have hierarchy; that is the reason for this syntax. This specification does not represent a file name hierarchy.

Path Specifications for Verilog Top Level

- Path specification must start with a top-level module name.
- Path specification can include "." or "/" path delimiters, but cannot include a mixture.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/top/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- top.sub/port_or_sig

Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

:

:sub

Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Path Specifications for VHDL Top Level

- Path specification may include the top-level module name but it is not required.
- Path specification can include "." or "/" path delimiters, but cannot include a mixture.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

:

:sub

Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

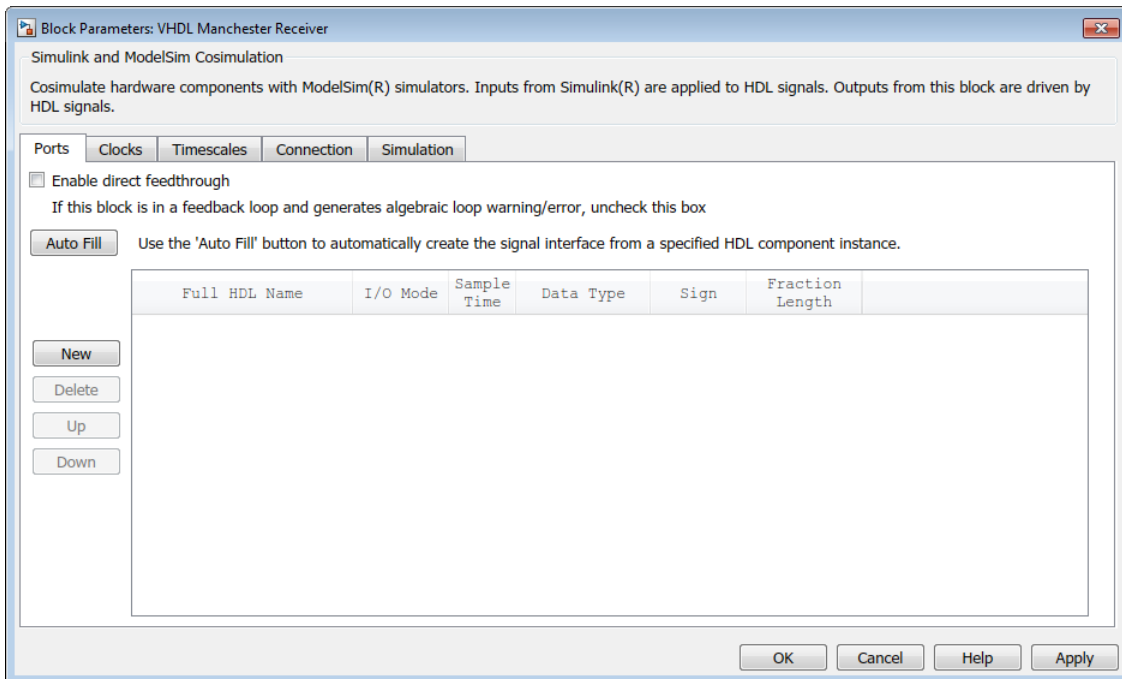
Get Signal Information from HDL Simulator

The **Auto Fill** button lets you begin an HDL simulator query and supply a path to a component or module in an HDL model under simulation in the HDL simulator. Usually, some change of the port information is required after the query completes. You must have the HDL simulator running with the HDL module loaded for **Auto Fill** to work.

The following example describes the required steps.

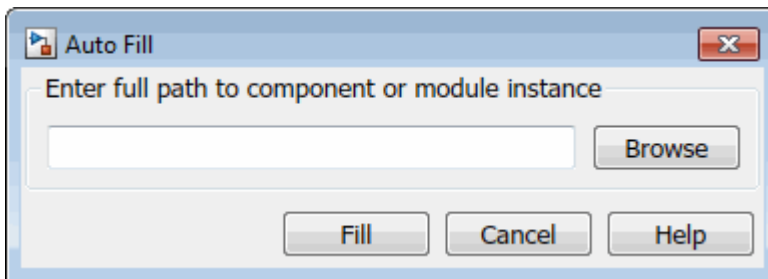
Note The example is based on a modified copy of the Manchester Receiver model, in which all signals were first deleted from the **Ports** and **Clocks** panes.

- 1 Open the block parameters dialog box for the HDL Cosimulation block. Click the **Ports** tab. The **Ports** pane opens (as an example, the **Ports** pane for the HDL Cosimulation block for use with ModelSim is shown in the illustrations below).



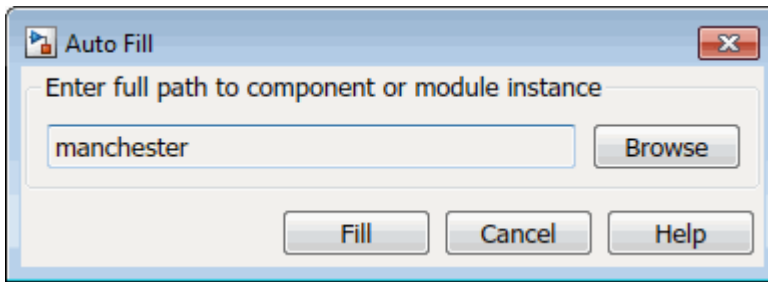
Tip Delete all ports before performing **Auto Fill** to make sure that no unused signal remains in the Ports list at any time.

- 2 Click the **Auto Fill** button. The **Auto Fill** dialog box opens.

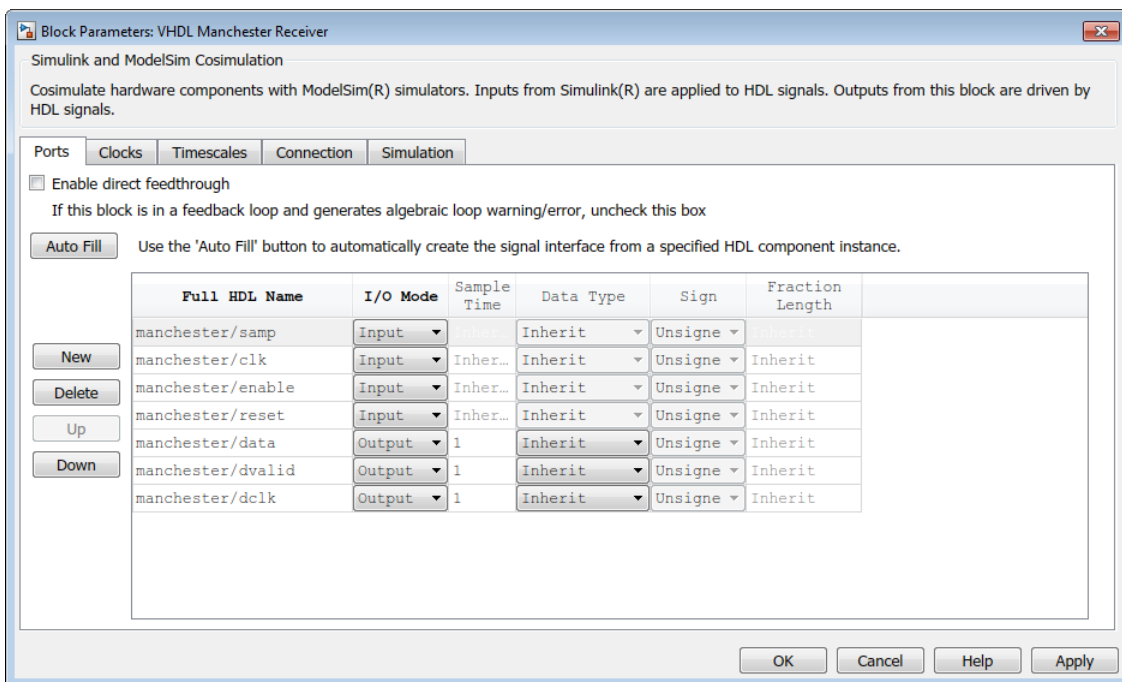


This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. The path you enter is not a file path and has nothing to do with the source files.

- 3 In this example, the Auto Fill feature obtains port data for a VHDL component called `manchester`. The HDL path is specified as `/top/manchester`. Path specifications will vary depending on your HDL simulator, see “Specify HDL Signal/Port and Module Paths for Simulink Component Cosimulation” on page 7-9.



- 4 Click **Fill** to dismiss the dialog box and the query is transmitted.
- 5 After the HDL simulator returns the port data, the Auto Fill feature enters it into the **Ports** pane, as shown in the following figure (examples shown for use with Cadence Xcelium).

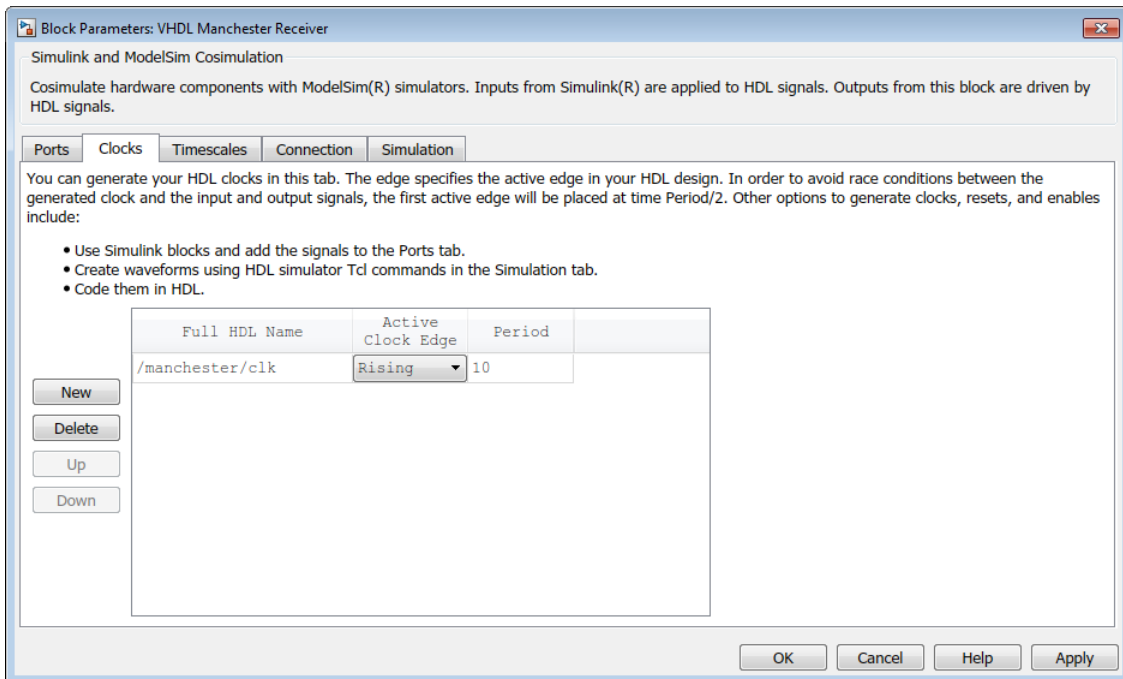
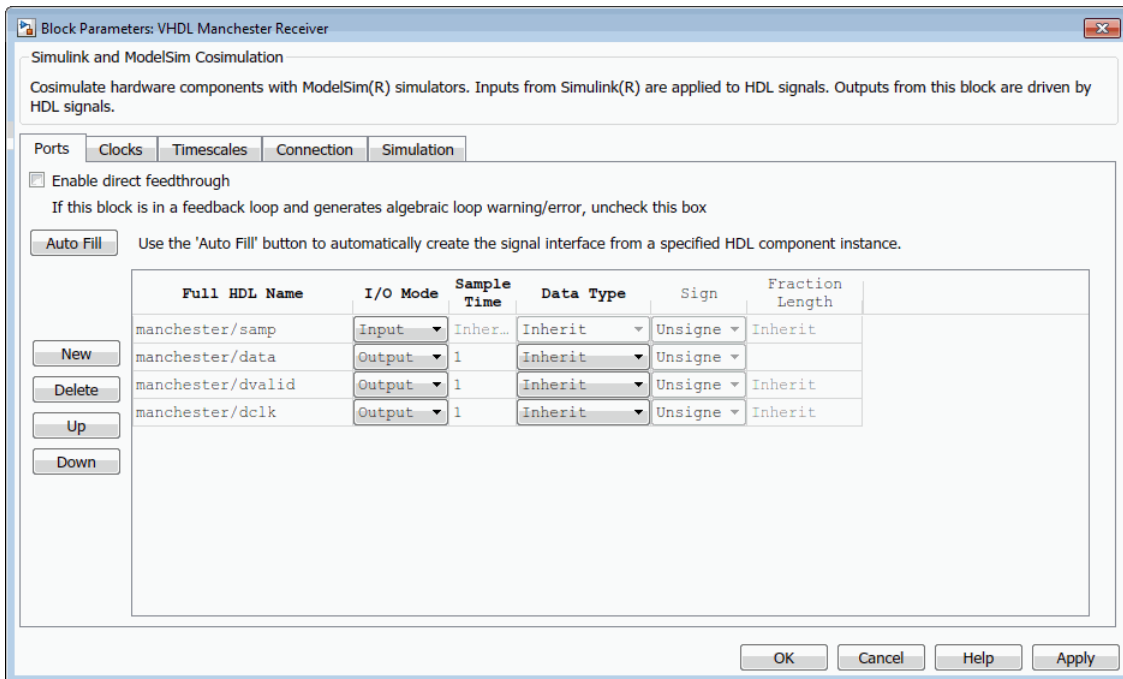


- 6 Click **Apply** to commit the port additions.
- 7 Delete unused signals from Ports pane and add Clock signal.

The preceding figure shows that the query entered clock, clock enable, and reset ports (labeled `clk`, `enable`, and `reset` respectively) into the ports list.

Delete the `clk`, `enable` and `reset` signals from the **Ports** pane, and add the `clk` signal in the **Clocks** pane.

These actions result in the signals shown in the next figures.



8 **Auto Fill** returns default values for output ports:

- **Sample time:** 1
- **Data type:** Inherit
- **Fraction length:** Inherit

You may need to change these values as required by your model. In this example, the **Sample time** should be set to 10 for all outputs. See “Specify Signal Data Types”.

- 9 Before closing the block parameters dialog box, click **Apply** to commit any edits you have made.

Observe that **Auto Fill** returned information about *all* inputs and outputs for the targeted component. In many cases, this will include signals that function in the HDL simulator but cannot be connected in the Simulink model. You may delete any such entries from the list in the **Ports** pane if they are unwanted. You *can* drive the signals from Simulink; you just have to define their values by laying down Simulink blocks.

Note that **Auto Fill** does not return information for internal signals. If your Simulink model needs to access such signals, you must enter them into the **Ports** pane manually. For example, in the case of the Manchester Receiver model, you would need to add output port entries for `top/manchester/sync_i`, `top/manchester/isum_i`, and `top/manchester/qsum_i`, as shown in step 8.

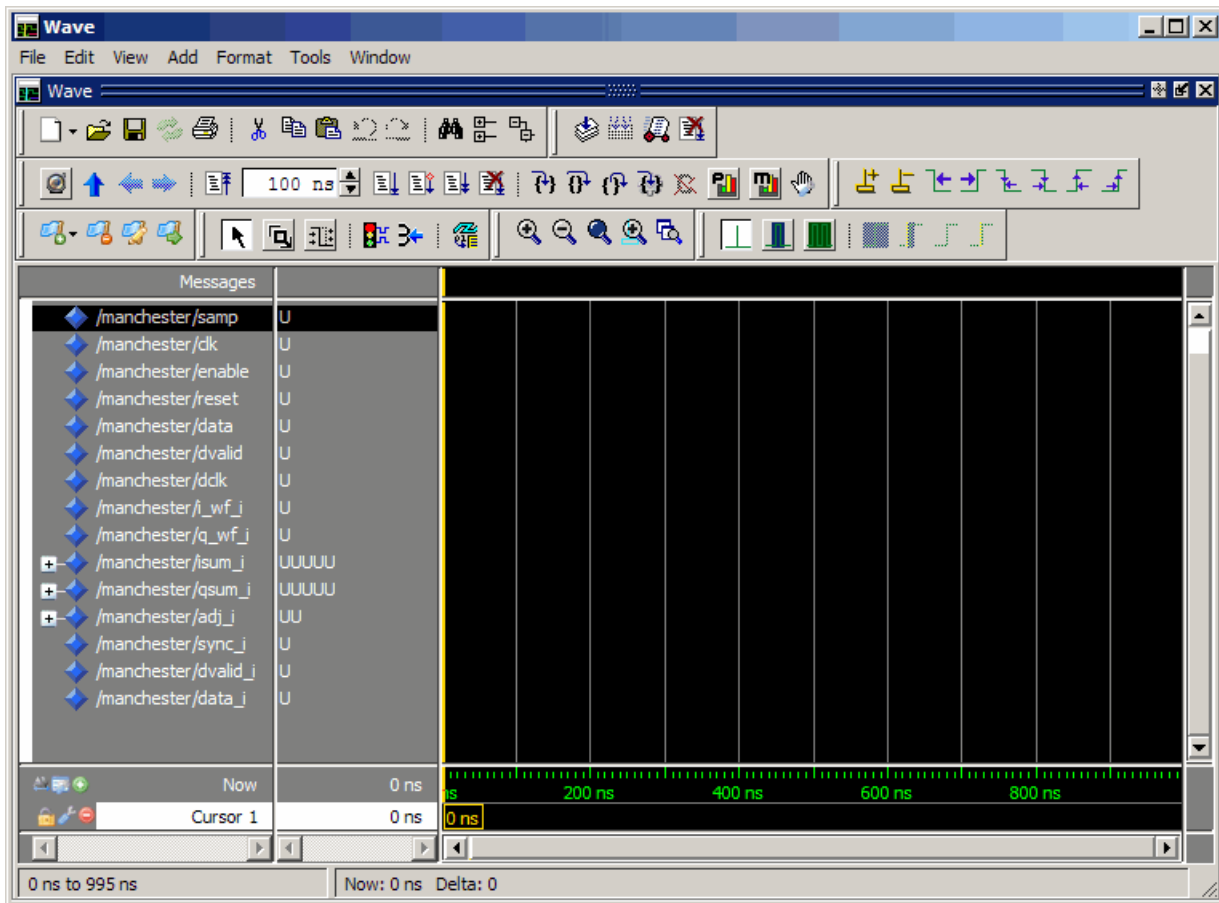
Xcelium and ModelSim users: Note that `clk`, `reset`, and `clk_enable` *may* be in the Clocks and Simulation panes but they don't *have* to be. These signals can be ports if you choose to drive them explicitly from Simulink.

Note When you import VHDL signals using **Auto Fill**, the HDL simulator returns the signal names in all capitals.

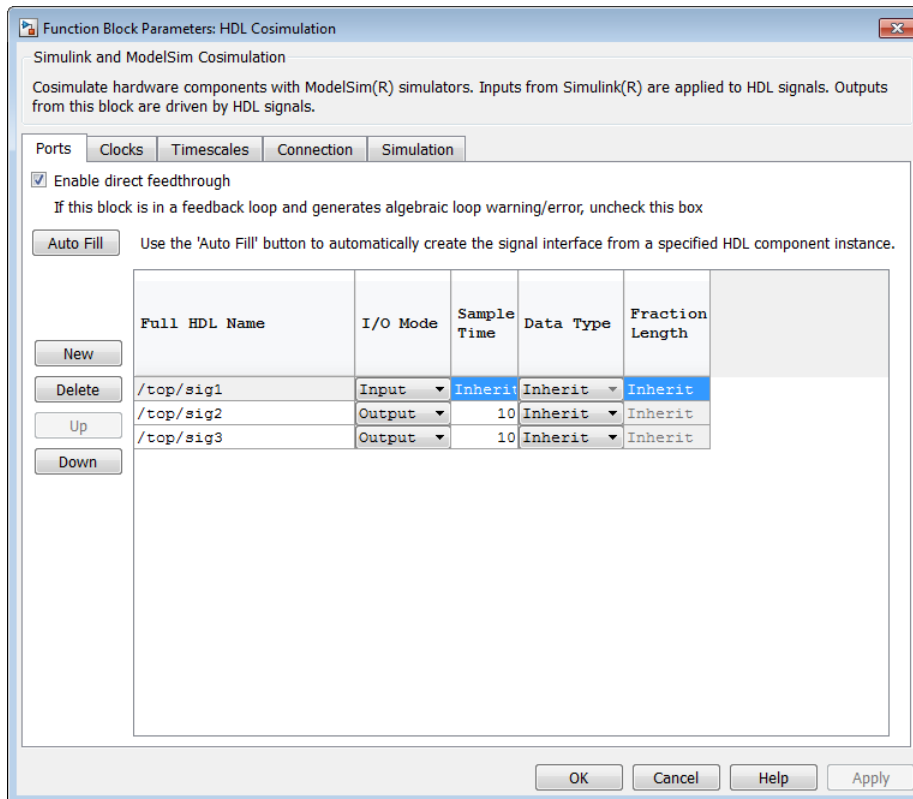
Enter Signal Information Manually

To enter signal information directly in the **Ports** pane, perform the following steps:

- 1 In the HDL simulator, determine the signal path names for the HDL signals you plan to define in your block. For example, in the ModelSim simulator, the following wave window shows all signals are subordinate to the top-level module `manchester`.



- 2 In Simulink, open the block parameters dialog box for your HDL Cosimulation block, if it is not already open.
- 3 Select the **Ports** pane tab. Simulink displays the following dialog box (example shown for use with Xcelium).



In this pane, you define the HDL signals of your design that you want to include in your Simulink block and set a sample time and data type for output ports. The parameters that you should specify on the **Ports** pane depend on the type of device the block is modeling as follows:

- For a device having both inputs and outputs: specify block input ports, block output ports, output sample times and output data types.

For output ports, accept the default or enter an explicit sample time. Data types can be specified explicitly, or set to `Inherit` (the default). In the default case, the output port data type is inherited either from the signal connected to the port, or derived from the HDL model.

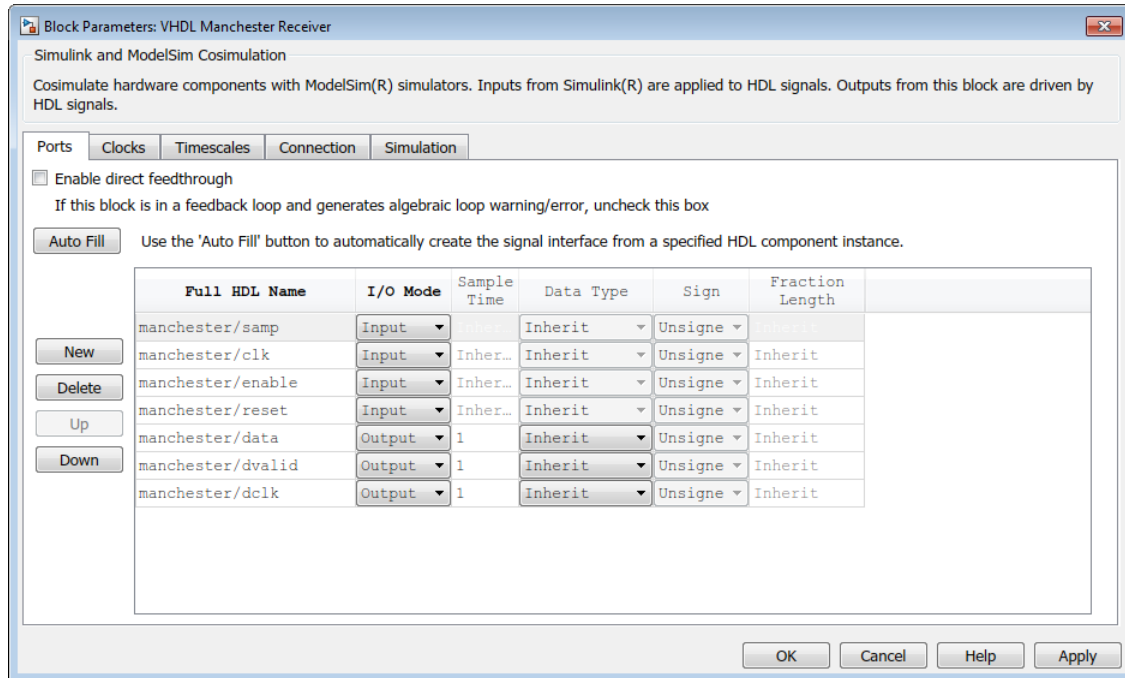
- For a sink device: specify block output ports.
- For a source device: specify block input ports.

4 Enter signal path names in the **Full HDL name** column by double-clicking on the existing default signal.

- Use HDL simulator path name syntax (as described in “Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation” on page 6-10).
- If you are adding signals, click **New** and then edit the default values. Select either **Input** or **Output** from the **I/O Mode** column.
- If you want to, set the **Sample Time**, **Data Type**, and **Fraction Length** parameters for signals explicitly, as discussed in the remaining steps.

When you have finished editing clock signals, click **Apply** to register your changes with Simulink.

The following dialog box shows port definitions for an HDL Cosimulation block. The signal path names match path names that appear in the HDL simulator **wave** window (Xcelium example shown).



Note When you define an input port, make sure that only one source is set up to force input to that port. If multiple sources drive a signal, your Simulink model may produce unpredictable results.

- 5 You must specify a sample time for the output ports. Simulink uses the value that you specify, and the current settings of the **Timescales** pane, to calculate an actual simulation sample time.

For more information on sample times in the HDL Verifier cosimulation environment, see “Simulation Timescales” on page 10-44.

- 6 You can configure the fixed-point data type of each output port explicitly if desired, or use a default (Inherited). In the default case, Simulink determines the data type for an output port as follows:

If Simulink can determine the data type of the signal connected to the output port, it applies that data type to the output port. For example, the data type of a connected Signal Specification block is known by back-propagation. Otherwise, Simulink queries the HDL simulator to determine the data type of the signal from the HDL module.

To assign an explicit fixed-point data type to a signal, perform the following steps:

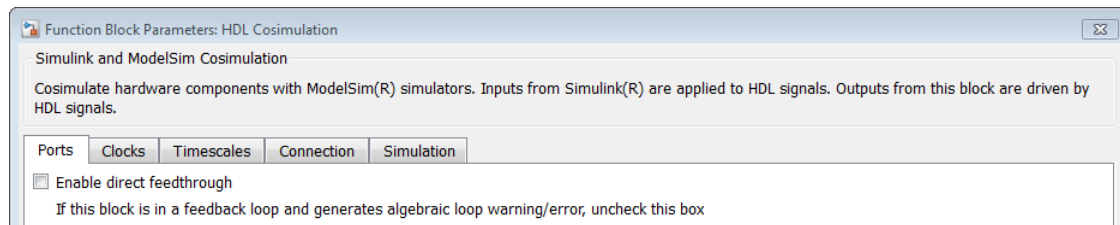
- a Select either Signed or Unsigned from the **Data Type** column.
- b If the signal has a fractional part, enter the **Fraction Length**.

For example, if the model has an 8-bit signal with Signed data type and a **Fraction Length** of 5, the HDL Cosimulation block assigns it the data type `sfix8_En5`. If the model has an Unsigned 16-bit signal with no fractional part (a **Fraction Length** of 0), the HDL Cosimulation block assigns it the data type `ufix16`.

7 Before closing the dialog box, click **Apply** to register your edits.

Control Output Port Directly by Value of Input Port

Enabling direct feedthrough allows input port value changes to propagate to the output ports in zero time, thus eliminating the possible delay at output sample in HDL designs with pure combinational logic. Specify the option to enable direct feedthrough on the **Ports** pane, as shown in the following figure.



Specify Signal Data Types

The **Data Type** and **Fraction Length** parameters apply only to output signals. See **Data Type** and **Fraction Length** on the Ports pane description of the HDL Cosimulation block.

Configure Simulink and HDL Simulator Timing Relationship

You configure the timing relationship between Simulink and the HDL simulator by using the **Timescales** pane of the block parameters dialog box. Before setting the **Timescales** parameters, read "Simulation Timescales" on page 10-44 to understand the supported timing modes and the issues that will determine your choice of timing mode.

You can specify either a relative or an absolute timing relationship between Simulink and the HDL simulator in the **Timescales** pane, as described in the HDL Cosimulation block reference.

Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the HDL Verifier interface:

- By defining the timing relationship manually (with **Timescales** pane)

When you define the relationship manually, you determine how many femtoseconds, picoseconds, nanoseconds, microseconds, milliseconds, seconds, or ticks in the HDL simulator represent 1 second in Simulink.

1 second in Simulink corresponds to in the HDL simulator

This quantity of HDL simulator time can be expressed in one of the following ways:

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation block defaults to relative timing mode for cosimulation. For more on relative timing mode, see "Relative Timing Mode" on page 10-48.
- In *absolute* units (such as milliseconds or nanoseconds). In this case, the cosimulation is said to operate in *absolute timing mode*. For more on absolute timing mode, see "Absolute Timing Mode" on page 10-51.

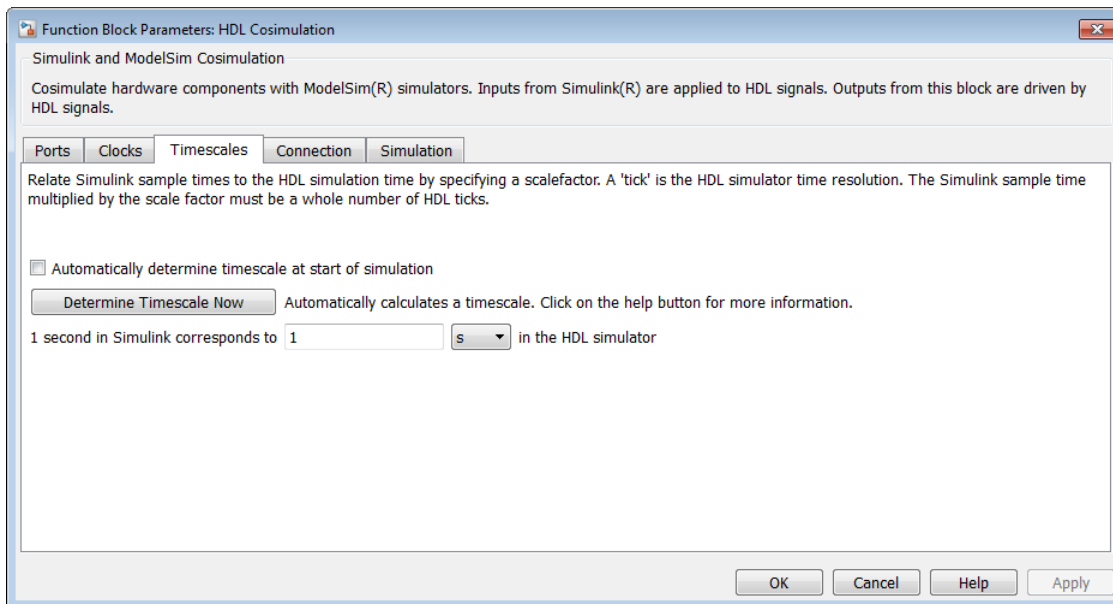
For more on relative and absolute time, see “Simulation Timescales” on page 10-44.

- By allowing HDL Verifier to define the timescale (with **Timescales** pane)

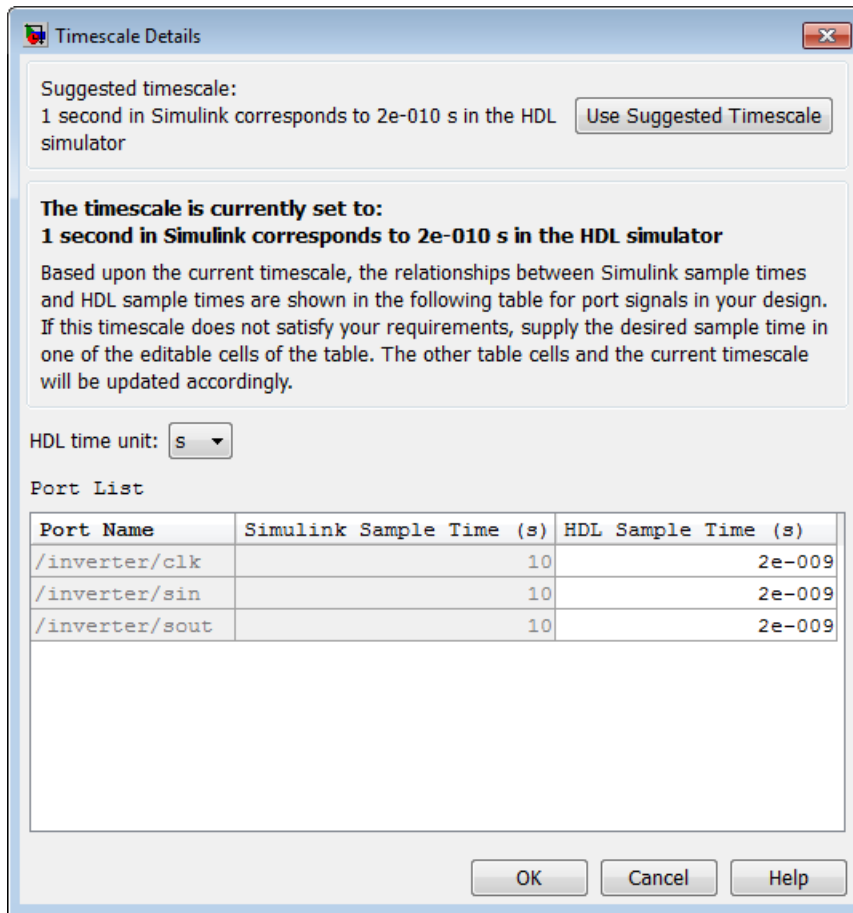
When you allow the link software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, the link product attempts to set the signal rate on the Simulink model port to the lowest possible number of HDL simulator ticks.

Before you begin, verify that the HDL simulator is running. HDL Verifier software can get the resolution limit of the HDL simulator only when that simulator is running.

You can choose to have HDL Verifier calculate a timescale while you are setting the parameters on the block dialog by clicking the **Timescale** option then clicking **Determine Timescale Now** or you can have HDL Verifier calculate the timescale when simulation begins by selecting **Automatically determine timescale at start of simulation**.



When you click **Determine Timescale Now**, HDL Verifier connects Simulink with the HDL simulator so that it can use the HDL simulator resolution to calculate the best timescale. You can accept the timescale HDL Verifier suggests or you can make changes in the port list directly. If you want to revert to the originally calculated settings, click **Use Suggested Timescale**. If you want to view sample times for all ports in the HDL design, select **Show all ports and clocks**.

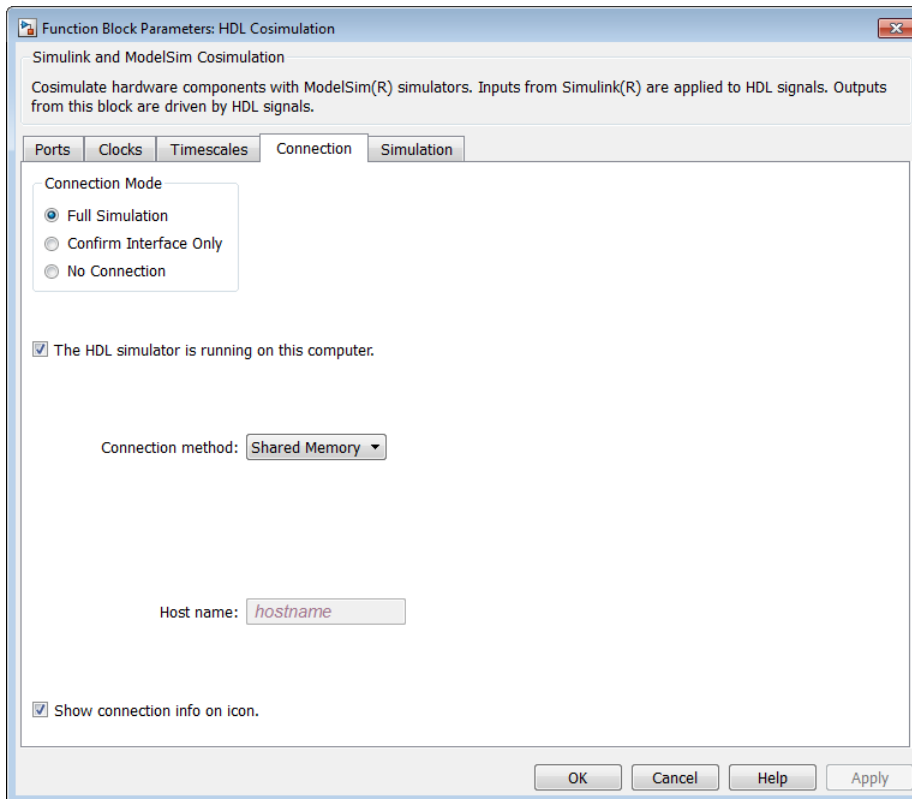


If you select **Automatically determine timescale at start of simulation**, you get the same dialog when the simulation starts in Simulink. Make the same adjustments at that time, if applicable, that you would if you clicked **Determine Timescale Now** when you were configuring the block.

Configure Communication Link in the HDL Cosimulation Block

You must select shared memory or socket communication. See “HDL Cosimulation with MATLAB or Simulink”.

After you decide which type of communication, configure a block's communication link with the **Connection** pane of the block parameters dialog box (example shown for use with ModelSim).



The following steps guide you through the communication configuration:

- 1 Determine whether Simulink and the HDL simulator are running on the same computer. If they are, skip to step 4.
- 2 Clear **The HDL simulator is running on this computer**. (This check box defaults to selected.) Because Simulink and the HDL simulator are running on different computers, HDL Verifier sets the **Connection method** to Socket.
- 3 Enter the host name of the computer that is running your HDL simulation (in the HDL simulator) in the **Host name** text field. In the **Port number or service** text field, specify a valid port number or service for your computer system. For information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports” on page 10-61. Skip to step 5.
- 4 If the HDL simulator and Simulink are running on the same computer, decide whether you are going to use shared memory or TCP/IP sockets for the communication channel. For information on the different modes of communication, see “HDL Cosimulation with MATLAB or Simulink”.

If you choose TCP/IP socket communication, specify a valid port number or service for your computer system in the **Port number or service** text field. For information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports” on page 10-61.

If you choose shared memory communication, select the **Shared memory** check box.

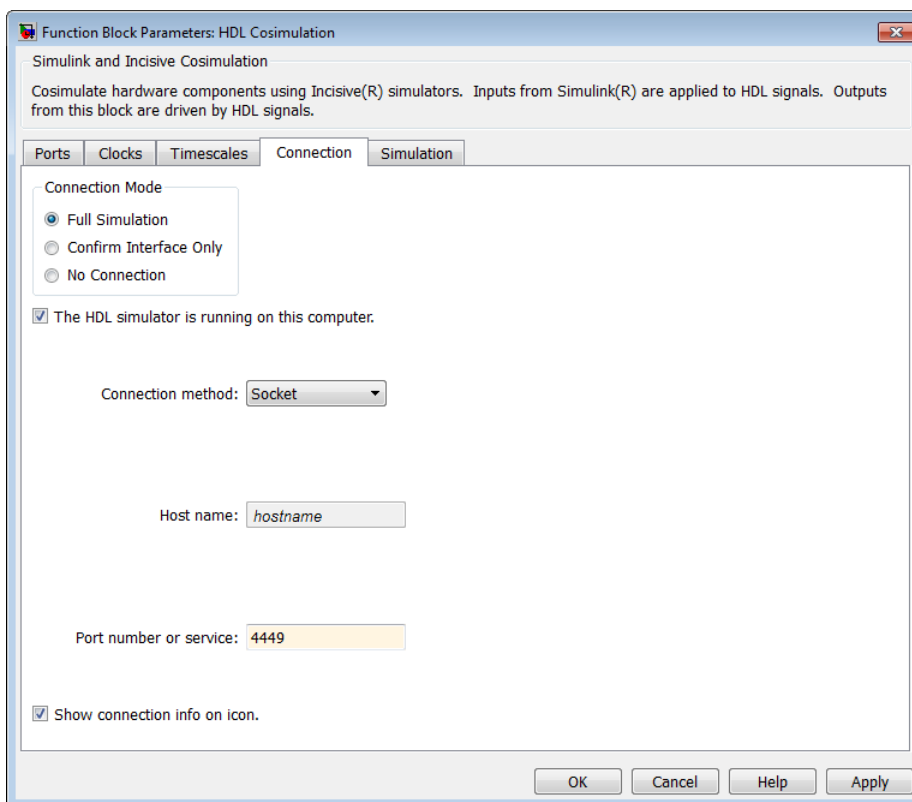
- 5 If you want to bypass the HDL simulator when you run a Simulink simulation, use the **Connection Mode** options to specify what type of simulation connection you want. Select one of the following options:
 - **Full Simulation**: Confirm interface and run HDL simulation (default).

- **Confirm Interface Only:** Check HDL simulator for expected signal names, dimensions, and data types, but do not run HDL simulation.
- **No Connection:** Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

With the second and third options, HDL Verifier software does not communicate with the HDL simulator during Simulink simulation.

6 Click **Apply**.

The following example dialog box shows communication definitions for an HDL Cosimulation block. The block is configured for Simulink and the HDL simulator running on the same computer, communicating in TCP/IP socket mode over TCP/IP port 4449.



Specify Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box

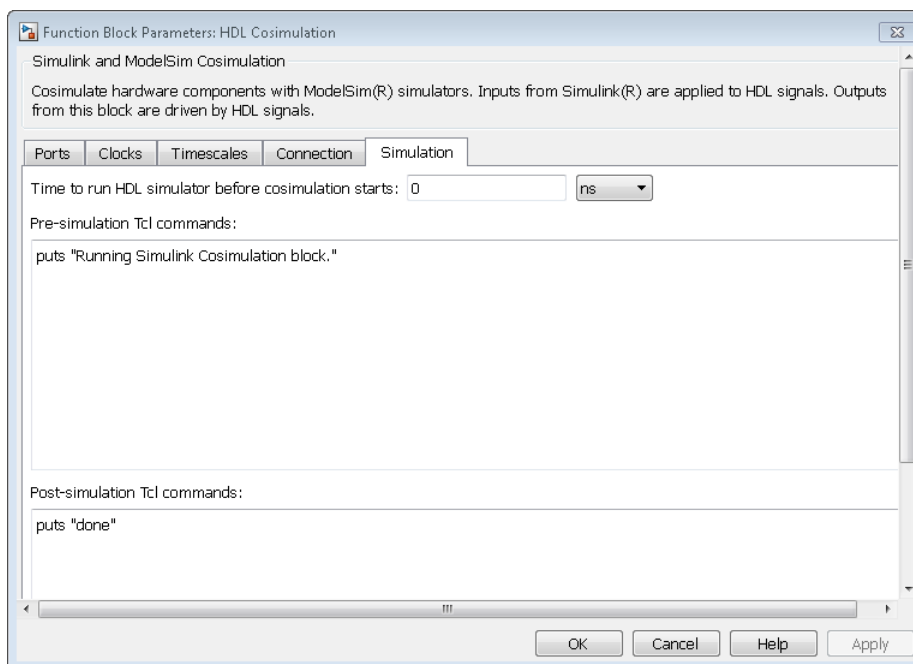
You have the option of specifying Tcl commands to execute before and after the HDL simulator simulates the HDL component of your Simulink model. Tcl is a programmable scripting language supported by most HDL simulation environments. Use of Tcl can range from something as simple as a one-line `puts` command to confirm that a simulation is running or as complete as a complex script that performs an extensive simulation initialization and startup sequence. For example, you can use the **Post-simulation command** field on the Simulation Pane to instruct the HDL simulator to restart at the end of a simulation run.

Note for ModelSim Users After each simulation, it takes ModelSim time to update the coverage result. To prevent the potential conflict between this process and the next cosimulation session, add a short pause between each successive simulation.

You can specify the pre-simulation and post-simulation Tcl commands by entering Tcl commands in the **Pre-simulation** commands or **Post-simulation** commands text fields in the **Simulation** pane of the HDL Cosimulation block parameters dialog box.

To specify Tcl commands, perform the following steps:

- 1 Select the **Simulation** tab of the block parameters dialog box. The dialog box appears as follows (example shown for use with ModelSim).

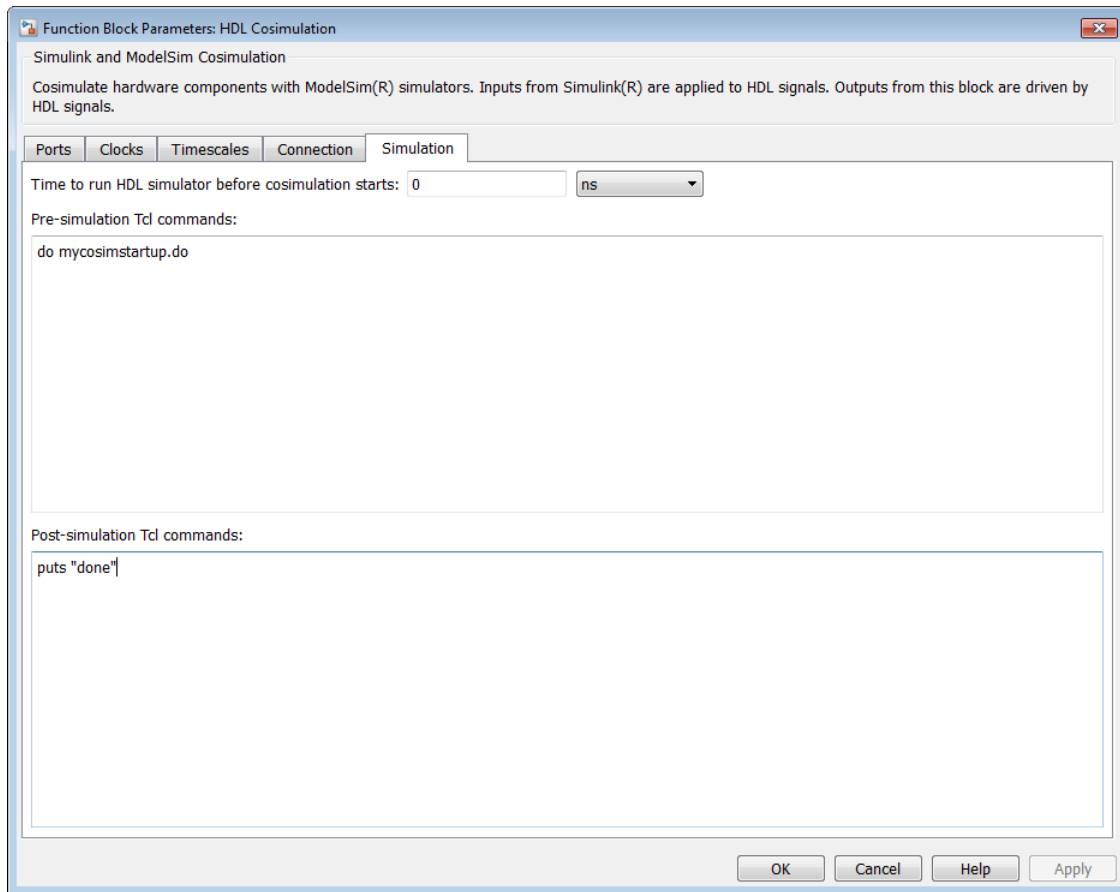


The **Pre-simulation commands** text box includes a `puts` command for reference purposes.

- 2 Enter one or more commands in the **Pre-simulation command** and **Post-simulation command** text boxes. You can specify one Tcl command per line in the text box or enter multiple commands per line by appending each command with a semicolon (;), which is the standard Tcl concatenation operator.

ModelSim DO Files

Alternatively, you can create a ModelSim DO file that lists Tcl commands and then specify that file with the ModelSim `do` command as shown in the following figure.



- 3 Click **Apply**.

Programmatically Control Block Parameters

One way to control block parameters is through the HDL Cosimulation block graphical dialog box. However, you can also control blocks by programmatically controlling the mask parameter values and the running of simulations. Parameter values can be read using the Simulink `get_param` function and written using the Simulink `set_param` function. All block parameters have attributes that indicate whether they are:

- Tunable — The attributes can change during the simulation run.
- Evaluated — The parameter value undergoes an evaluation to determine its actual value used by the S-Function.

The HDL Cosimulation block does not have any tunable parameters; thus, you get an error if you try to change a value while the simulation is running. However, it does have a few evaluated parameters.

You can see the list of parameters and their attributes by performing a right-mouse click on the block, selecting **View Mask**, and then the **Parameters** tab. The **Variable** column shows the programmatic parameter names. Alternatively, you can get the names programmatically by selecting the HDL Cosimulation block and then typing the following commands at the MATLAB prompt:

```
>> get_param(gcb, 'DialogParameters')
```

Some examples of using MATLAB to control simulations and mask parameter values follow. Usually, the commands are put into a script or function file and are called by several callback hooks available to the model developer. You can place the code in any of these suggested Simulink locations:

- In the model workspace. On the **Modeling** tab, in the **Design** section, click **Model Explorer**. In the Model Explorer dialog box, in the **Model Hierarchy** pane, select **Simulink Root > model_name > Model Workspace**. In the **Model Workspace** pane, from the **Data source** list, select **Model File**.
- In a model callback. On the **Modeling** tab, in the **Setup** section, click **Model Settings > Model Properties**. In the Model Properties dialog box, specify the callback function in the **Callbacks** tab.
- In a subsystem callback. Right-mouse click on an empty subsystem and then select **Properties > Callbacks**. Many of the HDL Verifier demos use this technique to start the HDL simulator by placing MATLAB code in the **OpenFcn** callback.
- In the HDL Cosimulation block callback. Right-mouse click on HDL Cosimulation block, and then select **Properties > Callbacks**.

Example: Scripting the Value of the Socket Number for HDL Simulator Communication

In a regression environment, you may need to determine the socket number for the Simulink/HDL simulator connection during the simulation to avoid collisions with other simulation runs. This example shows code that could handle that task. The script is for a 32-bit Linux platform.

```
ttcp_exec = [matlabroot '/toolbox/shared/hdlink/scripts/ttcp_glnx'];
[status, results] = system([ttcp_exec ' -a']);
if ~s
    parsed_result = textscan(results, '%s');
    avail_port = parsed_result{1}{2};
else
    error(results);
end

set_param('MyModel/HDL Cosimulation', 'CommPortNumber', avail_port);
```

See Also

HDL Cosimulation

Verify HDL Module with Simulink Test Bench

In this section...

“Tutorial Overview” on page 6-27
 “Develop VHDL Code” on page 6-27
 “Compile VHDL Code” on page 6-28
 “Create Simulink Model” on page 6-29
 “Set Up ModelSim for Use with Simulink” on page 6-37
 “Load Instances of VHDL Entity for Cosimulation with Simulink” on page 6-37
 “Run Simulation” on page 6-38
 “Shut Down Simulation” on page 6-40

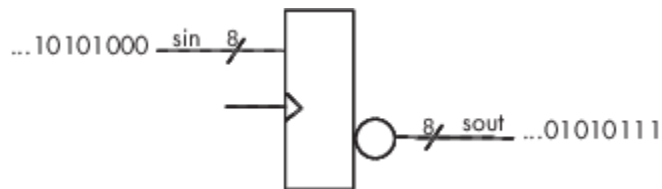
Tutorial Overview

This tutorial guides you through the basic steps for setting up an HDL Verifier session that uses Simulink and the HDL Cosimulation block to verify an HDL model. The HDL Cosimulation block cosimulates a hardware component by applying input signals to and reading output signals from an HDL model under simulation in ModelSim or Questa®. The HDL Cosimulation block supports simulation of either VHDL or Verilog models.

Note This tutorial is specific to Mentor Graphics simulator users; however, much of the process will be the same for Xcelium users. For cosimulating with the Vivado simulator, you must use the **Cosimulation Wizard** to generate the HDL Cosimulation block. For an example of this workflow, see “Import HDL Code for HDL Cosimulation Block” on page 9-24.

Develop VHDL Code

A typical Simulink and ModelSim scenario is to create a model for a specific hardware component in ModelSim that you later need to integrate into a larger Simulink model. The first step is to design and develop a VHDL model in ModelSim. In this tutorial, you use ModelSim and VHDL to develop a model that represents the following inverter:



The VHDL entity for this model will represent 8-bit streams of input and output signal values with an IN port and OUT port of type STD_LOGIC_VECTOR. An input clock signal of type STD_LOGIC will trigger the bit inversion process when set.

Perform the following steps:

- 1 Start ModelSim
- 2 Change to the writable folder MyPlayArea, which you may have created for another tutorial. If you have not created the folder, create it now. The folder must be writable.

```
ModelSim>cd C:/MyPlayArea
```

- 3 Open a new VHDL source edit window.
- 4 Add the following VHDL code:

```
-----  
-- Simulink and ModelSim Inverter Tutorial  
--  
-- Copyright 2003-2004 The MathWorks, Inc.  
--  
-----  
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
ENTITY inverter IS PORT (  
    sin : IN  std_logic_vector(7 DOWNTO 0);  
    sout: OUT std_logic_vector(7 DOWNTO 0);  
    clk : IN  std_logic  
);  
END inverter;  
  
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
ARCHITECTURE behavioral OF inverter IS  
BEGIN  
    PROCESS(clk)  
    BEGIN  
        IF (clk'EVENT AND clk = '1') THEN  
            sout <= NOT sin;  
        END IF;  
    END PROCESS;  
END behavioral;
```

- 5 Save the file to `inverter.vhd`.

Compile VHDL Code

This section explains how to set up a design library and compile `inverter.vhd`, as follows:

- 1 Verify that the file `inverter.vhd` is in the current folder by entering the `ls` command at the ModelSim command prompt.
- 2 Create a design library to hold your compilation results. To create the library and required `_info` file, enter the `vlib` and `vmap` commands as follows:

```
ModelSim> vlib work
```

```
ModelSim> vmap work work
```

If the design library `work` already exists, ModelSim *does not* overwrite the current library, but displays the following warning:

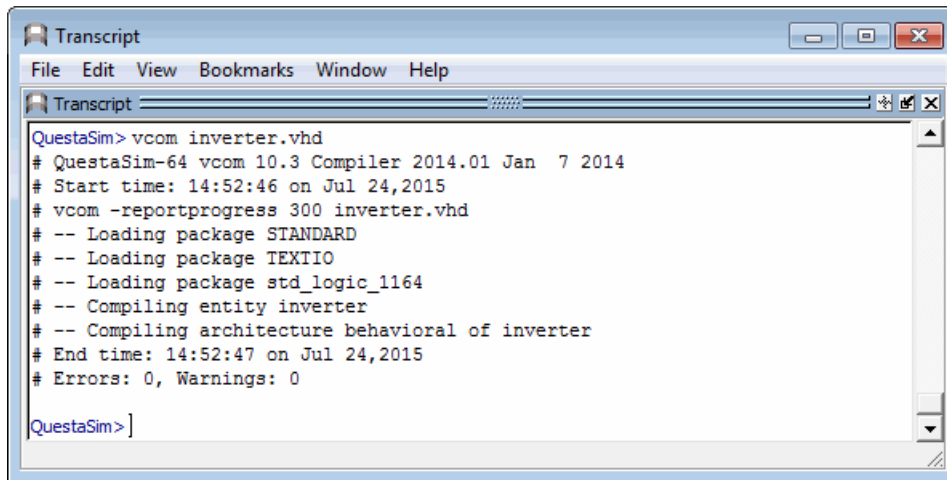
```
# ** Warning: (vlib-34) Library already exists at "work".
```

Note You must use the ModelSim **File** menu or `vlib` command to create the library folder so that the required `_info` file is created. Do not create the library with operating system commands.

- 3 Compile the VHDL file. One way of compiling the file is to click the file name in the project workspace and select **Compile > Compile All**. Another alternative is to specify the name of the VHDL file with the vcom command, as follows:

```
ModelSim> vcom inverter.vhd
```

If the compilations succeed, informational messages appear in the command window and the compiler populates the work library with the compilation results.



```

Transcript
File Edit View Bookmarks Window Help
Transcript
QuestaSim> vcom inverter.vhd
# QuestaSim-64 vcom 10.3 Compiler 2014.01 Jan 7 2014
# Start time: 14:52:46 on Jul 24,2015
# vcom -reportprogress 300 inverter.vhd
# -- Loading package STANDARD
# -- Loading package TEXTIO
# -- Loading package std_logic_1164
# -- Compiling entity inverter
# -- Compiling architecture behavioral of inverter
# End time: 14:52:47 on Jul 24,2015
# Errors: 0, Warnings: 0
QuestaSim>

```

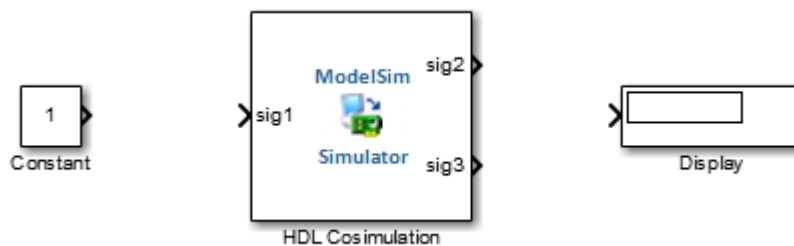
Create Simulink Model

Now create your Simulink model. For this tutorial, you create a simple Simulink model that drives input into a block representing the VHDL inverter you coded in “Develop VHDL Code” on page 6-27 and displays the inverted output.

Start by creating a model, as follows:

- 1 Start MATLAB, if it is not already running. Open a new model window. Then, open the Simulink **Library Browser**.
- 2 Drag the following blocks from the Simulink **Library Browser** to your model window:
 - Constant block from the Simulink **Sources** library
 - HDL Cosimulation block from the HDL Verifier block library
 - Display block from the Simulink **Sinks** library

Arrange the three blocks in the order shown in the following figure.

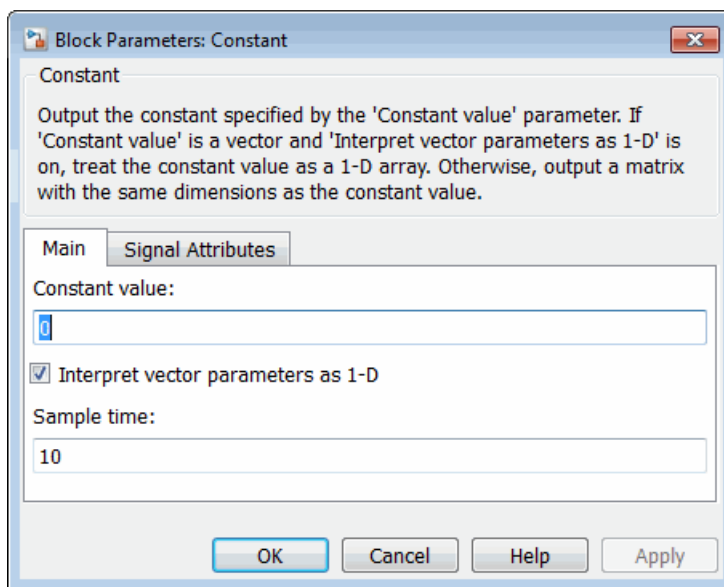


Next, configure the Constant block, which is the model's input source, by performing the following actions:

- 1 Double-click the Constant block icon to open the Constant block parameters dialog box. Enter the following parameter values in the **Main** pane:
 - **Constant value:** 0
 - **Sample time:** 10

Later you can change these initial values to see the effect various sample times have on different simulation runs.

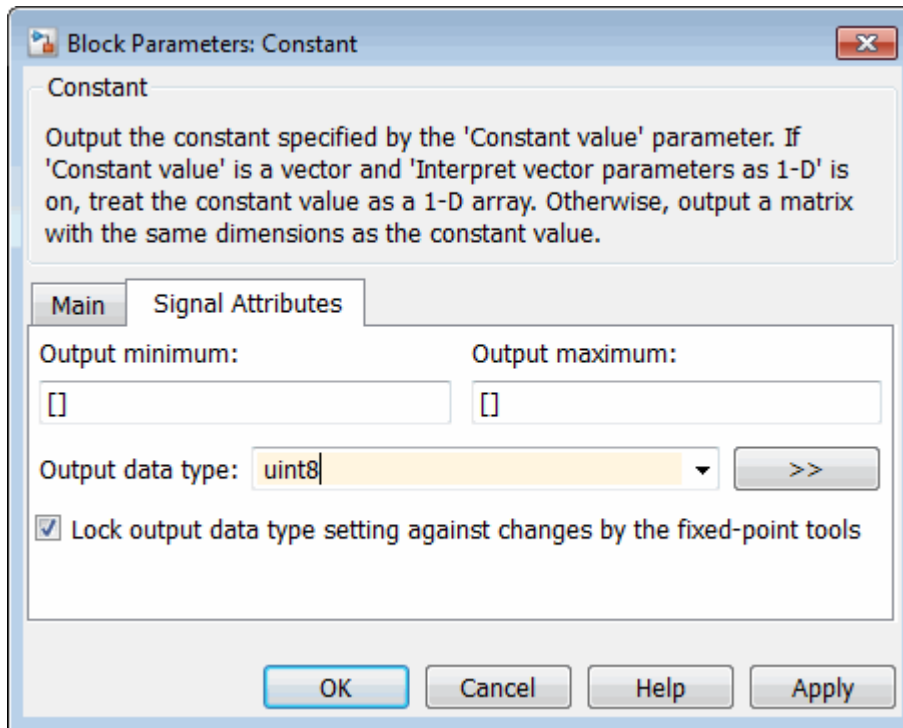
The dialog box should now appear as follows.



- 2 Click the **Signal Attributes** tab. The dialog box now displays the **Output data type** menu.

Select `uint8` from the **Output data type** menu. This data type specification is supported by HDL Verifier software without the need for a type conversion. It maps directly to the VHDL type for the VHDL port `sin`, `STD_LOGIC_VECTOR(7 DOWNTO 0)`.

The dialog box should now appear as follows.

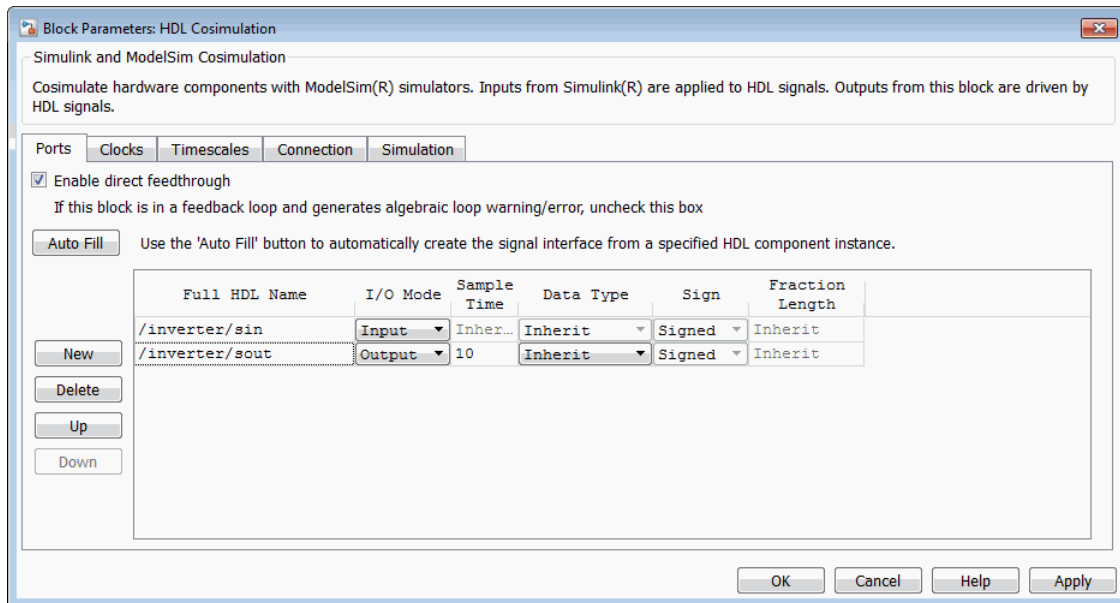


- 3 Click **OK**. The Constant block parameters dialog box closes and the value in the Constant block icon changes to 0.

Next, configure the HDL Cosimulation block, which represents the inverter model written in VHDL. Start with the **Ports** pane, by performing the following actions:

- 1 Double-click the HDL Cosimulation block icon. The **Block Parameters** dialog box for the HDL Cosimulation block appears. Click the **Ports** tab.
- 2 In the **Ports** pane, select the sample signal `/top/sig1` from the signal list in the center of the pane by double-clicking on it.
- 3 Replace the sample signal path name `/top/sig1` with `/inverter/sin`. Then click **Apply**. The signal name on the HDL Cosimulation block changes.
- 4 Similarly, select the sample signal `/top/sig2`. Change the **Full HDL Name** to `/inverter/sout`. Select **Output** from the **I/O Mode** list. Change the **Sample Time** parameter to 10. Then click **Apply** to update the list.
- 5 Select the sample signal `/top/sig3`. Click the **Delete** button. The signal is now removed from the list.

The **Ports** pane should appear as follows.



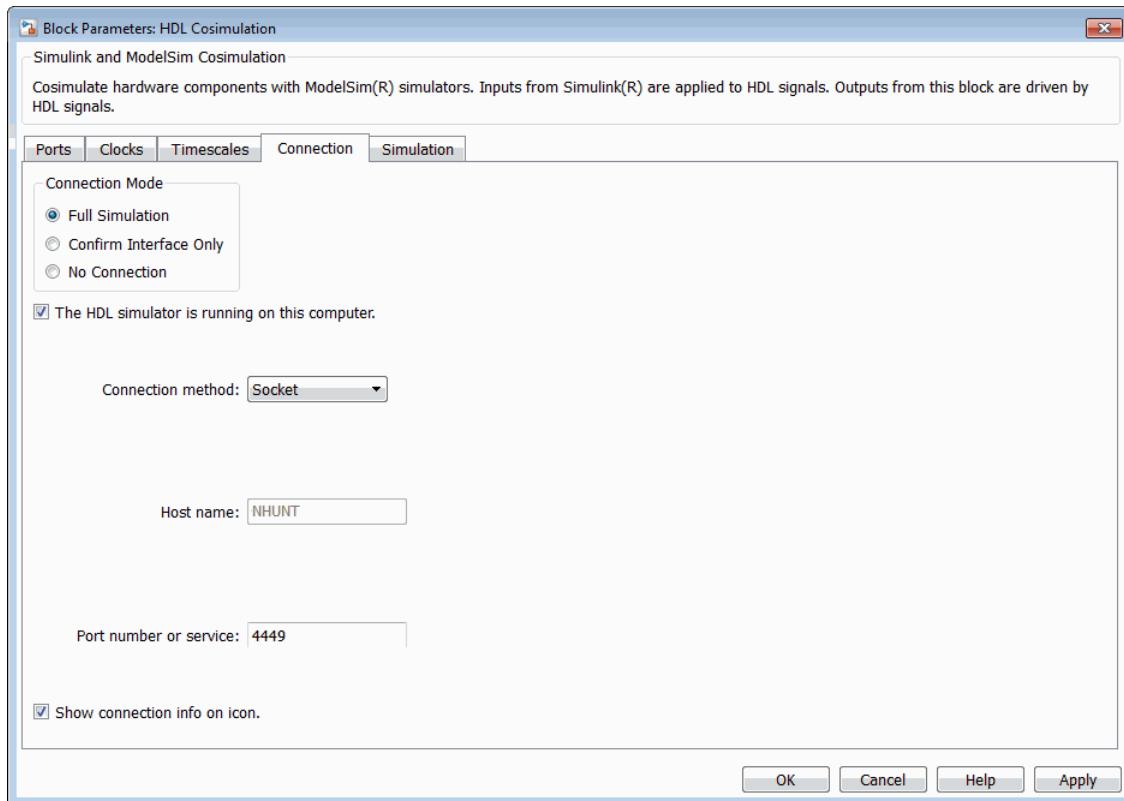
Now configure the parameters of the **Connection** pane by performing the following actions:

- 1 Click the **Connection** tab.
- 2 Leave **Connection Mode** as **Full Simulation**.
- 3 Select socket from the **Connection method** list. This option specifies that Simulink and ModelSim will communicate via a designated TCP/IP socket port. Observe that two additional fields, **Port number or service** and **Host name**, are now visible.

Note that, because **The HDL simulator is running on this computer** is selected by default, the **Host name** field is disabled. In this configuration, both Simulink and ModelSim execute on the same computer, so you do not need to enter a remote host system name.

- 4 In the **Port number or service** text box, enter socket port number 4449 or, if this port is not available on your system, another valid port number or service name. The model will use TCP/IP socket communication to link with ModelSim. Note what you enter for this parameter. You will specify the same socket port information when you set up ModelSim for linking with Simulink.

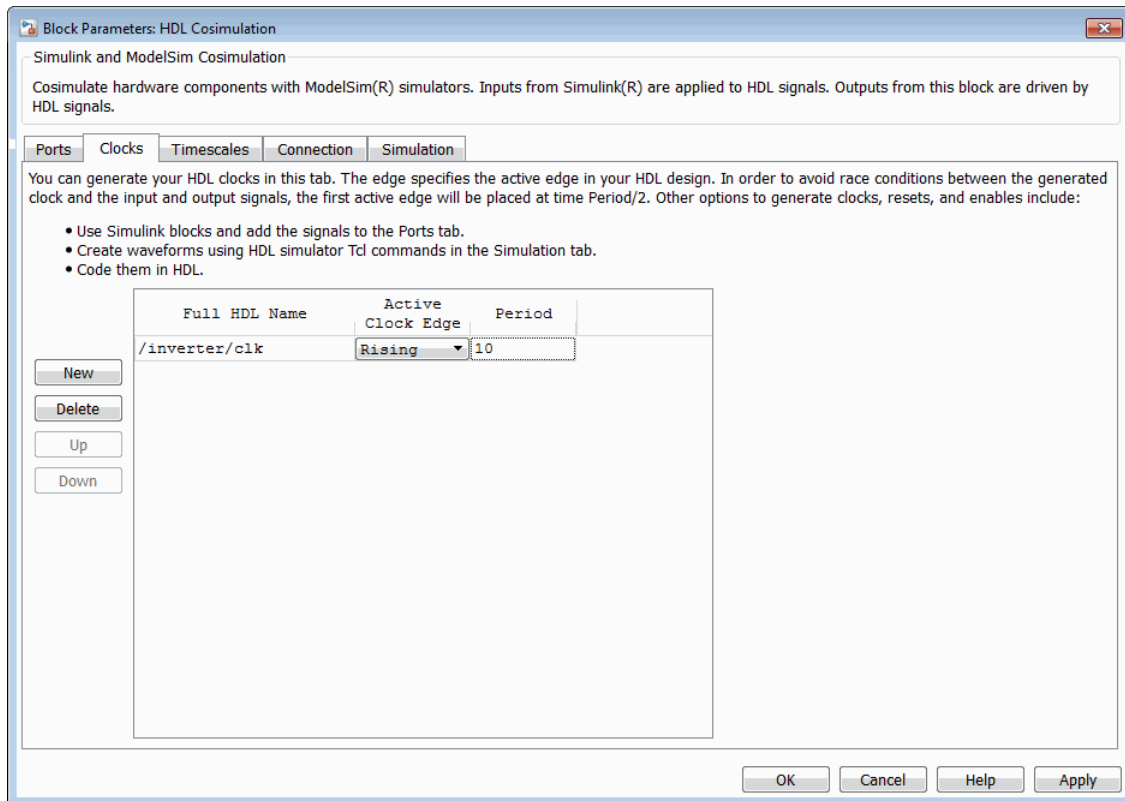
The **Connection** pane should appear as follows.



5 Click **Apply**.

Now configure the **Clocks** pane by performing the following actions:

- 1 Click the **Clocks** tab.
- 2 Click the **New** button. A new clock signal with an empty signal name is added to the signal list.
- 3 Double-click on the new signal name to edit. Enter the signal path `/inverter/clk`. Then select **Rising** from the **Edge** list. Set the **Period** parameter to **10**.
- 4 The **Clocks** pane should appear as follows.

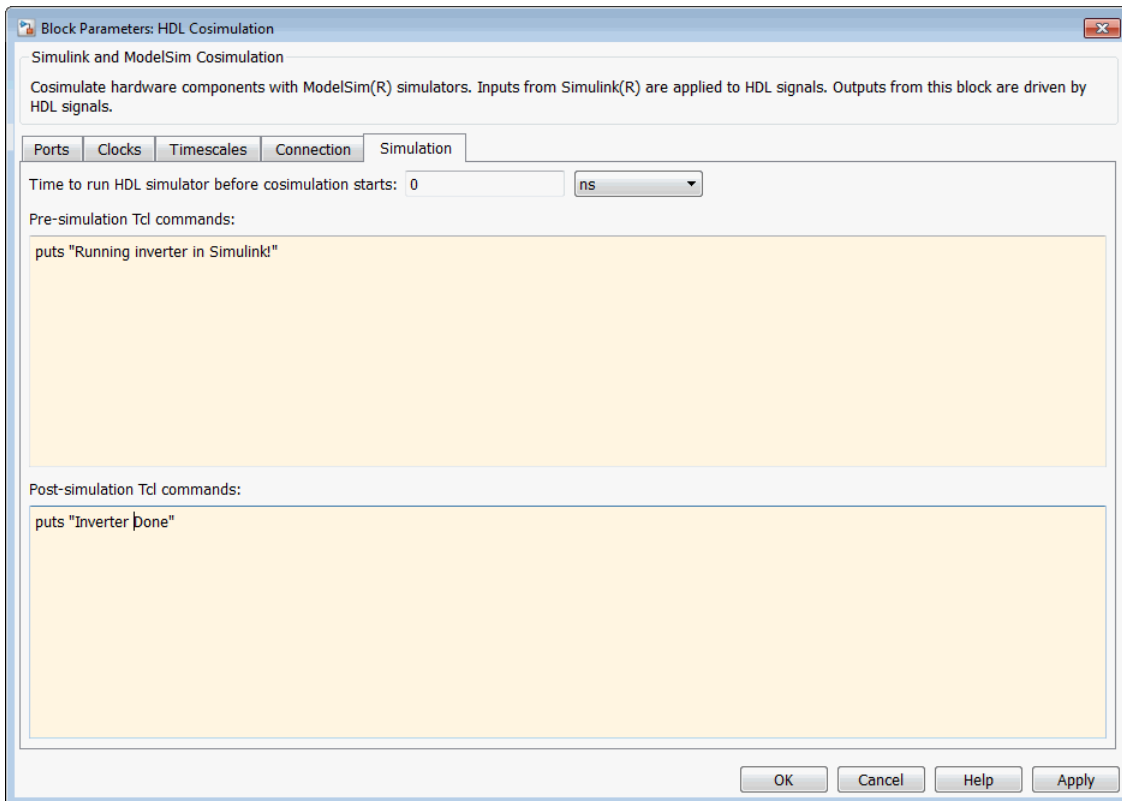


5 Click **Apply**.

Next, enter some simple Tcl commands to be executed before and after simulation, as follows:

- 1 Click the **Simulation** tab.
- 2 In the **Pre-simulation Tcl commands** text box, edit the default Tcl command:
puts "Running inverter in Simulink!"
- 3 In the **Post-simulation Tcl commands** text box, edit the default Tcl command:
puts "Inverter Done"

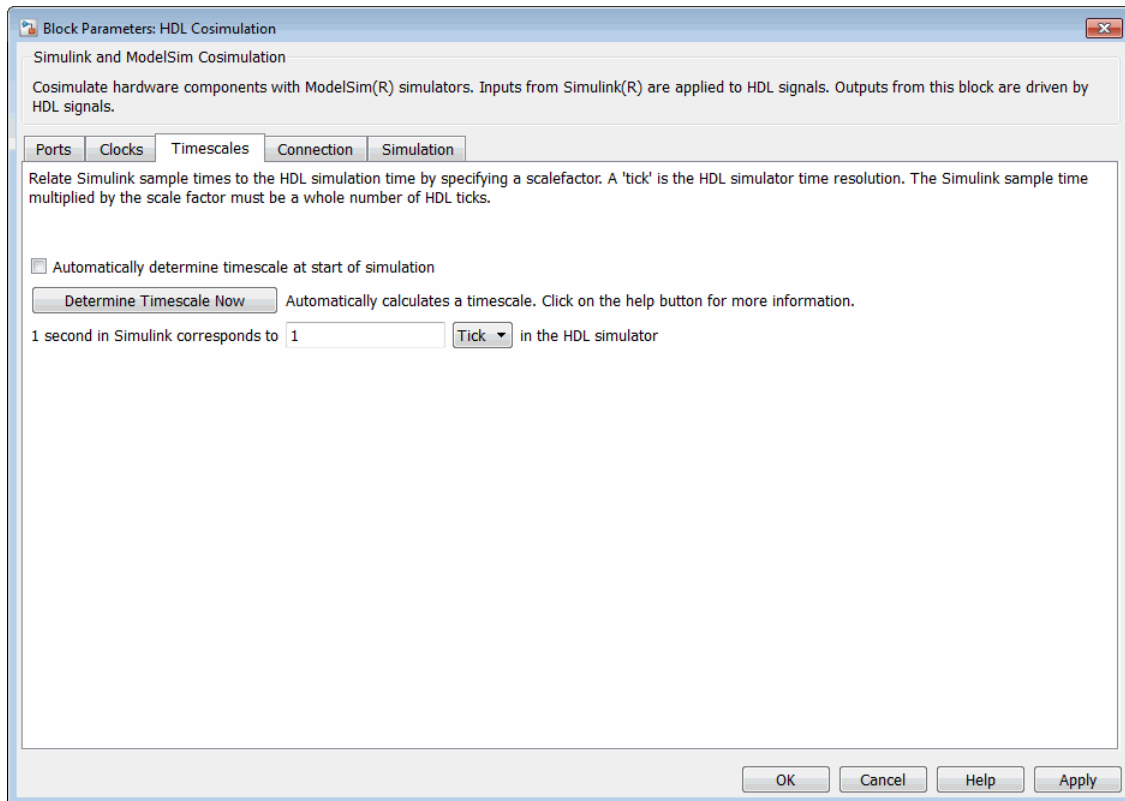
The **Simulation** pane should appear as follows.



- 4 Click **Apply**.

Next, view the **Timescales** pane to make sure it is set to its default parameters, as follows:

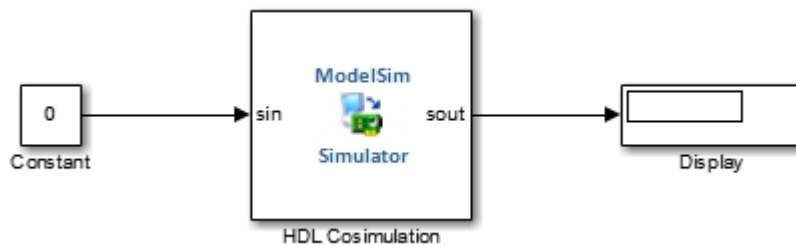
- 1 Click the **Timescales** tab.
- 2 The default settings of the **Timescales** pane are shown in the following figure. These settings are required for operation of this example. See "Simulation Timescales" on page 10-44 for further information.



- 3 Click **OK** to close the **Block Parameters** dialog box.

The final step is to connect the blocks, configure model-wide parameters, and save the model. Perform the following actions:

- 1 Connect the blocks as shown in the following figure.



At this point, you might also want to consider adjusting block annotations.

- 2 Configure the Simulink solver selection for a fixed-step, discrete simulation; this is required for cosimulation operation. Perform the following actions:
 - a In the **Modeling tab**, click **Model Settings**. The **Model Configuration Parameters** dialog box opens, displaying the **Solver selection** pane.
 - b Select **Fixed-step** from the **Type** menu.
 - c Select **discrete (no continuous states)** from the **Solver** menu.
 - d Click **Apply**.

- e Click **OK** to close the **Model Configuration Parameters** dialog box.

See “Set Simulink Model Configuration Parameters” on page 5-4 for further information on Simulink settings that are optimal for use with HDL Verifier software.

- 3 Save the model.

Set Up ModelSim for Use with Simulink

You now have a VHDL representation of an inverter and a Simulink model that applies the inverter. To start ModelSim such that it is ready for use with Simulink, enter the following command line in the MATLAB Command Window:

```
vsim('socketsimulink', 4449)
```

Note If you entered a different socket port specification when you configured the HDL Cosimulation block in Simulink, replace the port number 4449 in the preceding command line with the applicable socket port information for your model. The `vsim` function informs ModelSim of the TCP/IP socket to use for establishing a communication link with your Simulink model.

Load Instances of VHDL Entity for Cosimulation with Simulink

This section explains how to use the `vsimulink` command to load an instance of your VHDL entity for cosimulation with Simulink. The `vsimulink` command is an HDL Verifier variant of the ModelSim `vsim` command. It is made available as part of the ModelSim configuration.

To load an instance of the `inverter` entity, perform the following actions:

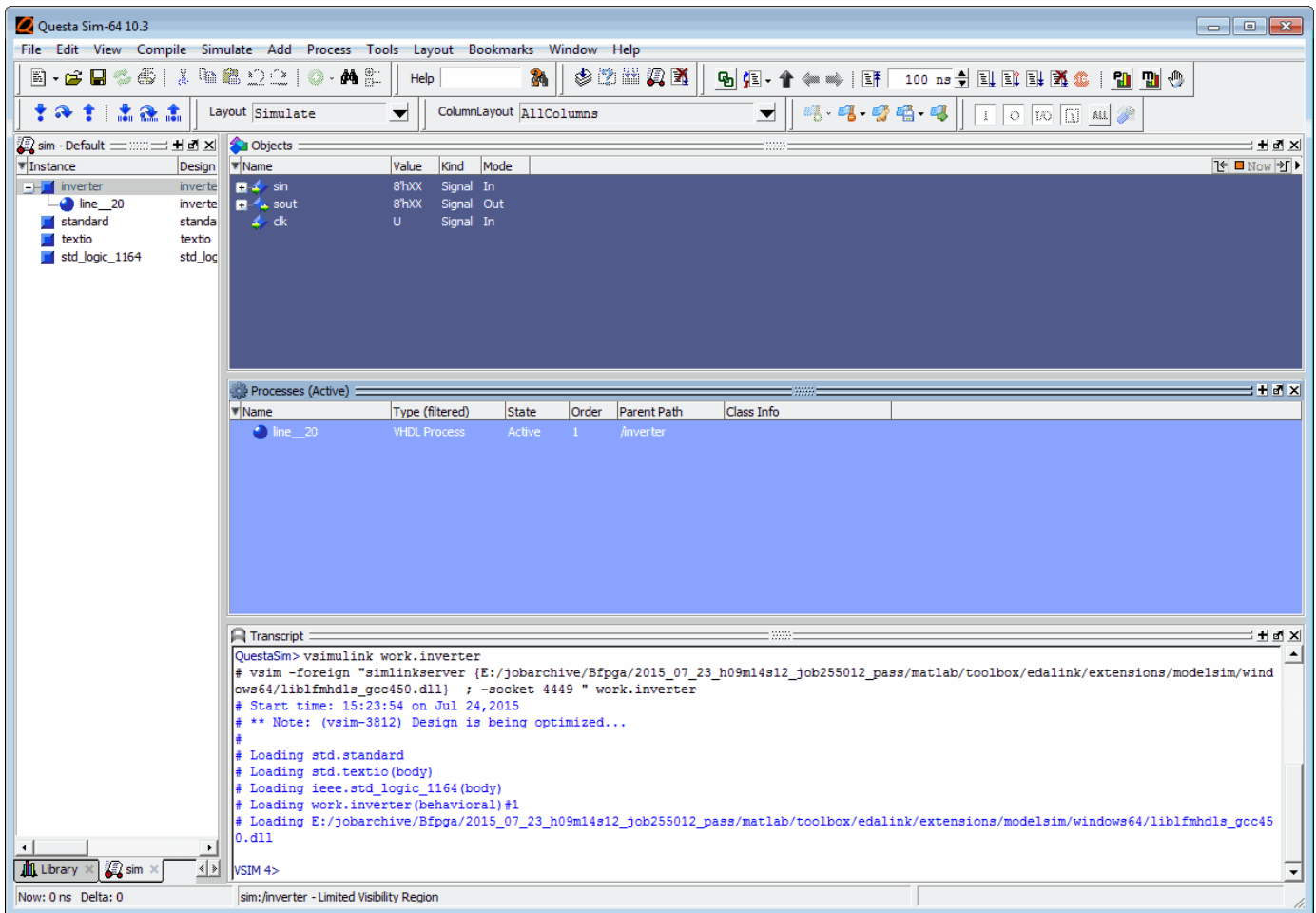
- 1 Change your input focus to the ModelSim window.
- 2 If your VHD file is not in the current folder, change your folder to the location of your `inverter.vhd` file. For example:

```
ModelSim> cd C:/MyPlayArea
```

- 3 Enter the following `vsimulink` command:

```
ModelSim> vsimulink work.inverter
```

ModelSim starts the `vsim` simulator such that it is ready to simulate entity `inverter` in the context of your Simulink model. The ModelSim command window display should be similar to the following.



Run Simulation

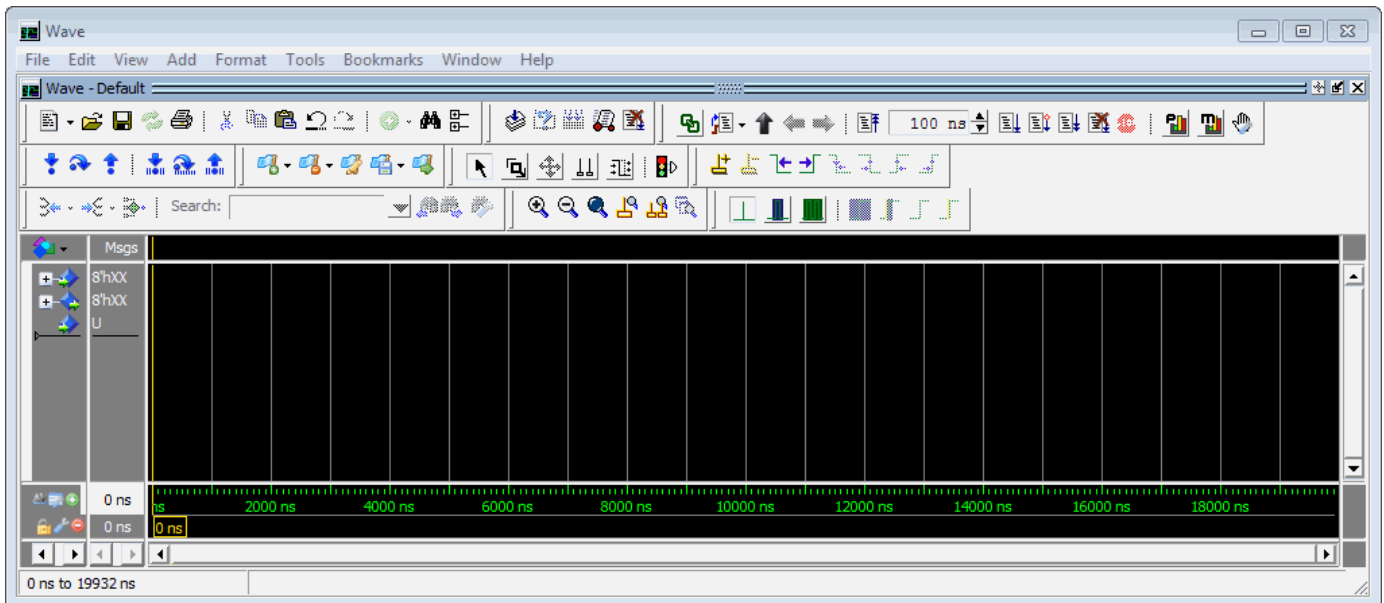
This section guides you through a scenario of running and monitoring a cosimulation session.

Perform the following actions:

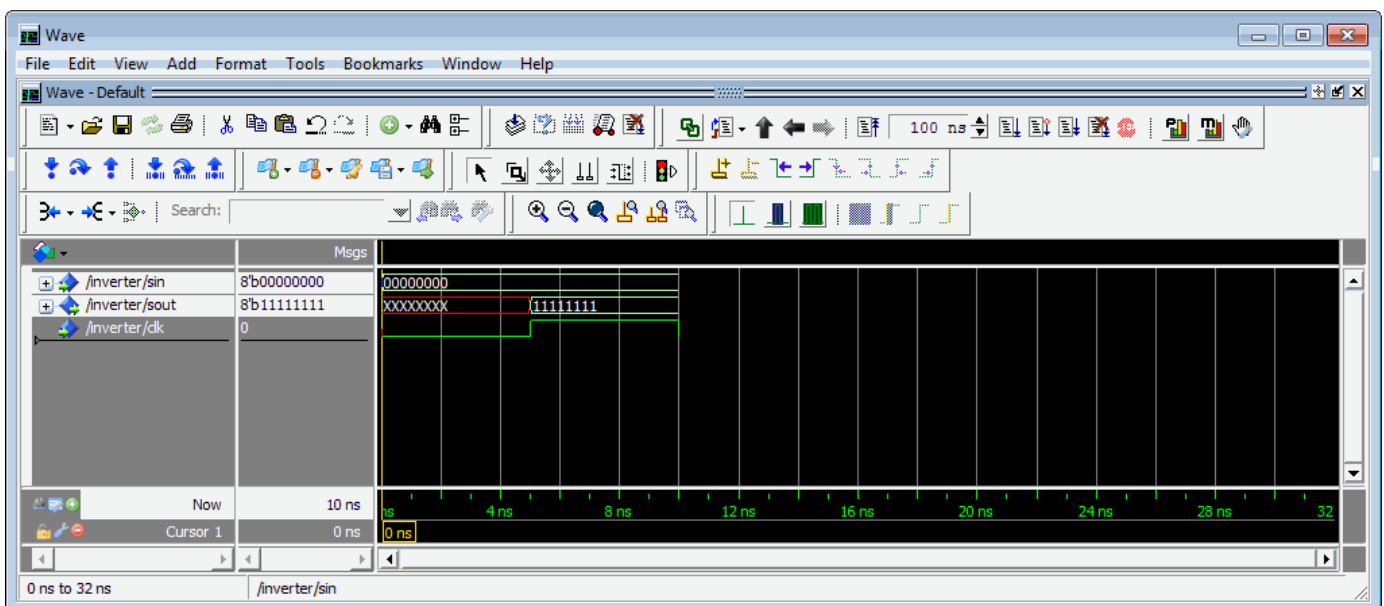
- 1 Open and add the inverter signals to a **wave** window by entering the following ModelSim command:

```
VSIM n> add wave /inverter/*
```

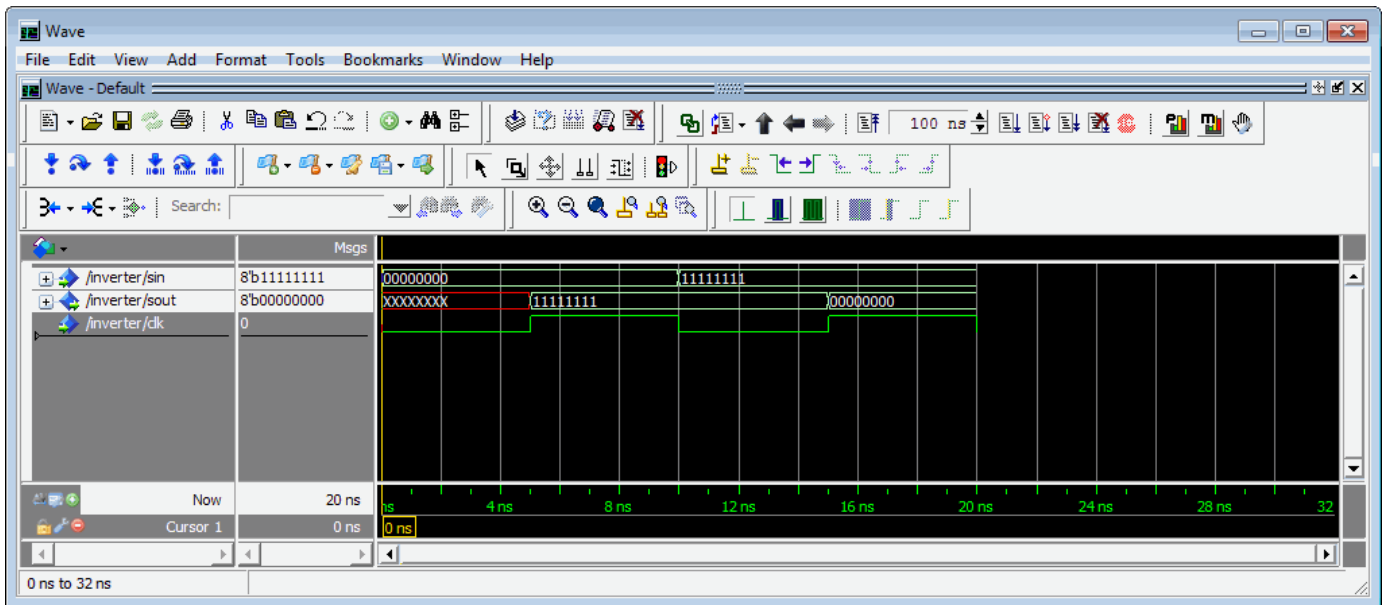
The following **wave** window appears.



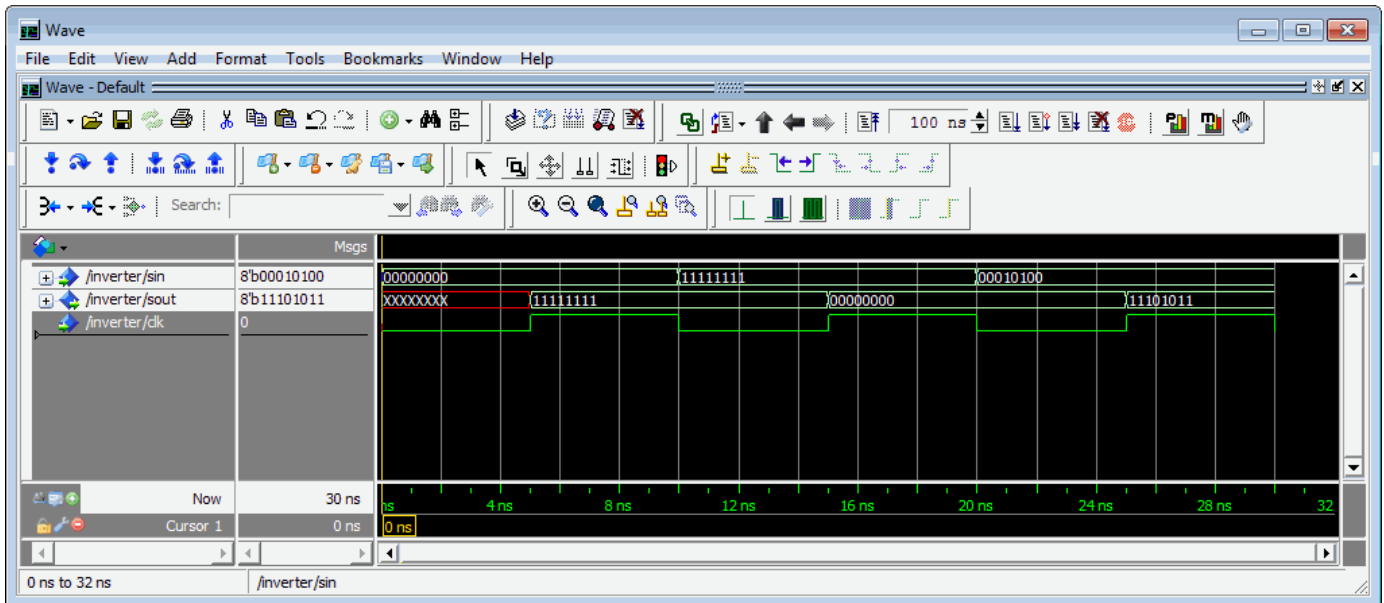
- 2 Change your input focus to your Simulink model window.
- 3 Start a Simulink simulation. The value in the Display block changes to 255. Also note the changes that occur in the ModelSim **wave** window. You might need to zoom in to get a better view of the signal data.



- 4 In the Simulink model, change **Constant value** to 255, save the model, and start another simulation. The value in the Display block changes to 0 and the ModelSim **wave** window is updated as follows.



- 5 In the Simulink model, change **Constant value** to 2 and **Sample time** to 20 and start another simulation. This time, the value in the Display block changes to 253 and the ModelSim **wave** window appears as shown in the following figure.



Note the change in the sample time in the **wave** window.

Shut Down Simulation

This section explains how to shut down a simulation in an orderly way, as follows:

- 1 In ModelSim, stop the simulation by selecting **Simulate > End Simulation**.
- 2 Quit ModelSim.

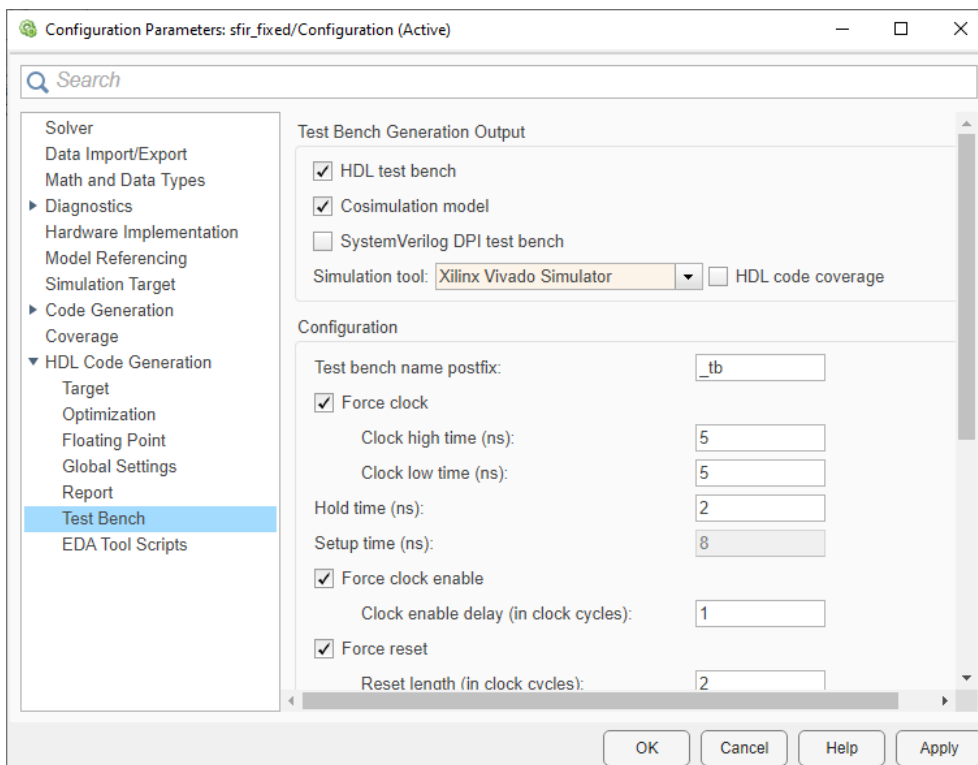
- 3 Close the Simulink model window.

Automatic Verification of Generated HDL Code from Simulink

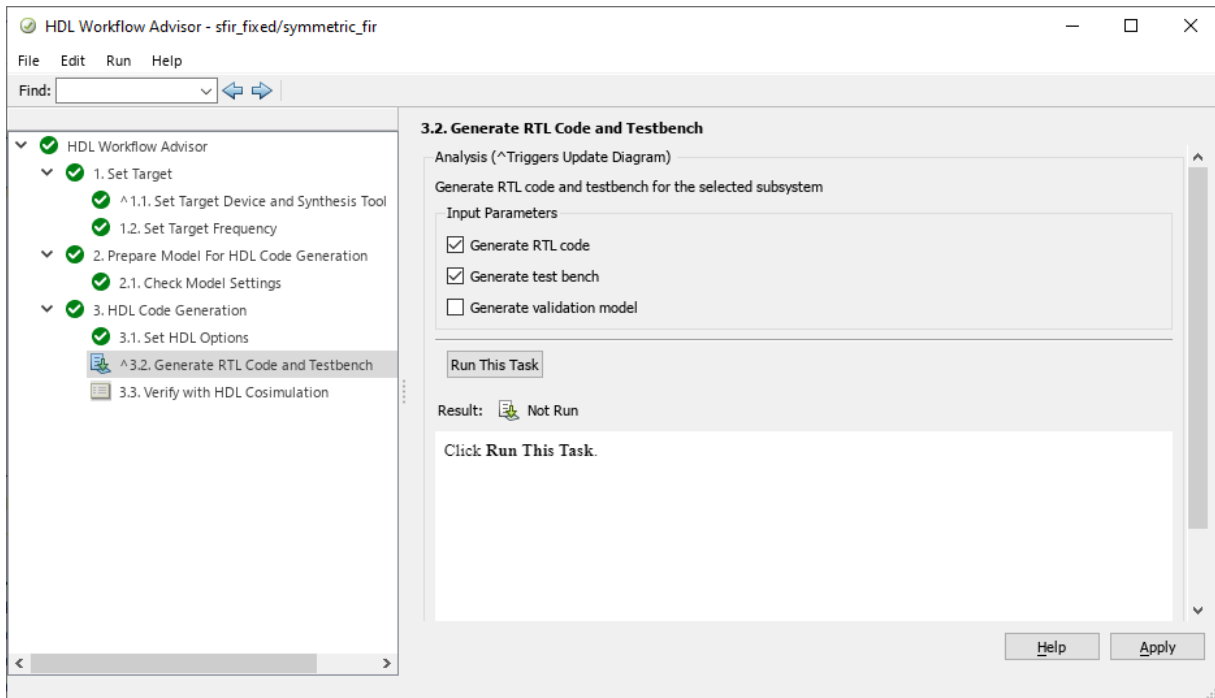
The automatic verification feature integrates verification as part of the workflow for HDL cosimulation using the HDL Workflow Advisor. During this workflow, Simulink generates a test bench model for HDL cosimulation. This test bench model compares the generated HDL DUT outputs (from the HDL Cosimulation block) with the original Simulink block outputs. This step automatically runs this test bench and returns pass/fail information. If the outputs of the HDL DUT match the output of original Simulink block in the test bench, the test passes.

This feature requires an HDL Coder and an HDL Verifier license.

- 1 To open HDL Workflow Advisor for your model, select the **APPS** tab on the Simulink toolstrip, and select **HDL Coder**. Then, click the **Workflow Advisor** button.
- 2 Step 1.1, select **Generic ASIC/FPGA**.
- 3 Run all steps under 2, **Prepare Model For HDL Code Generation**.
- 4 At step 3.1, **Set HDL Options**, click **HDL Code Generation Settings** to open the configuration parameters on the **HDL Code Generation** pane.
- 5 Set the HDL language, and under **Test Bench** select **Cosimulation model**. Then set **Simulation tool** to Mentor Graphics ModelSim, Cadence Incisive, or Xilinx Vivado Simulator for your HDL simulator. Click **OK** to return to the **HDL Workflow Advisor** window.



- 6 At Step 3.2, **Generate RTL Code and Testbench**, select **Generate test bench**. This selection causes Step 3.3 to appear. Click **Run This Task** to generate the RTL code and test bench.



- 7 At step 3.3, click **Run This Task**. The HDL Workflow Advisor and HDL Verifier verify the generated HDL using cosimulation between the HDL Simulator and the Simulink test bench. Any relevant status messages are displayed in the status window in the HDL Workflow Advisor.

Replace HDL Component with Simulink Algorithm

- “Component Simulation with Simulink” on page 7-2
- “Create Simulink Model for Component Cosimulation” on page 7-5

Component Simulation with Simulink

In this section...

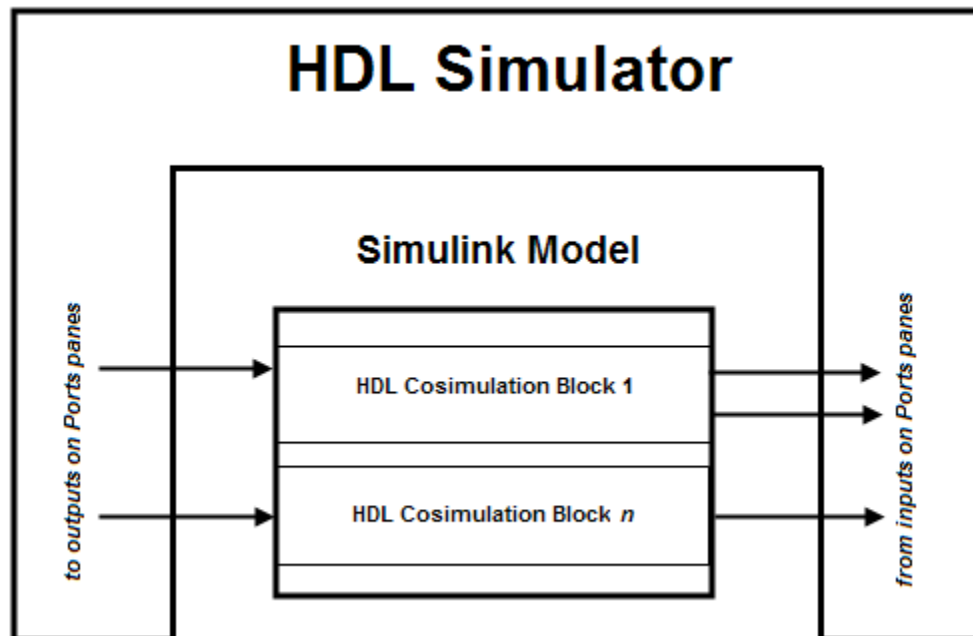
“How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Component Simulation” on page 7-2

“HDL Cosimulation Block Features for Component Simulation” on page 7-3

How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Component Simulation

When you link the HDL simulator with a Simulink application, the HDL simulator functions as the server. As the following diagram shows, the HDL Cosimulation blocks inside the Simulink model accept signals from the HDL module under simulation in the HDL simulator via the output ports on the Ports panes and return data via the input ports on the Ports panes.

Note This configuration is not supported for Vivado cosimulation.



How Simulink Drives Cosimulation Signals

Although you can bind the output ports of an HDL Cosimulation block to any signal in an HDL model hierarchy, you must use some caution when connecting signals to input ports. You want to verify that the signal you are binding to does not have other drivers. If it does, use resolved logic types; otherwise you may get unpredictable results.

If you need to use a signal that has multiple drivers and it is resolved (for example, it is of VHDL type `STD_LOGIC`), Simulink applies the resolution function at each time step defined by the signal's Simulink sample rate. Depending on the other drivers, the Simulink value may or may not get

applied. Furthermore, Simulink has no control over signal changes that occur between its sample times.

Note Verify that signals used in cosimulation have read/write access. You can check read/write access through the HDL simulator—see HDL simulator documentation for details.

This rule applies to all signals on the **Ports**, **Clocks**, and **Simulation** panes and to signals added to the model in any other manner.

Multirate Signals During Component Cosimulation

HDL Verifier software supports the use of multirate signals, signals that are sampled or updated at different rates, in a single HDL Cosimulation block. An HDL Cosimulation block exchanges data for each signal at the Simulink sample rate for that signal. For input signals, an HDL Cosimulation block accepts and honors all signal rates.

The HDL Cosimulation block also lets you specify an independent sample time for each output port. You must explicitly set the sample time for each output port, or accept the default. Using this setting lets you control the rate at which Simulink updates an output port by reading the corresponding signal from the HDL simulator.

Continuous Time Signals

Use the Simulink Zero-Order Hold block to apply a zero-order hold (ZOH) on continuous signals that are driven into an HDL Cosimulation block.

HDL Cosimulation Block Features for Component Simulation

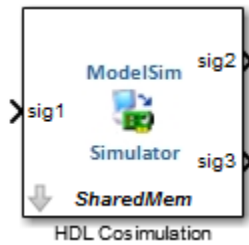
The HDL Verifier HDL Cosimulation block links hardware components that are concurrently simulating in the HDL simulator to the rest of a Simulink model.

You can link Simulink and the HDL simulator in two possible ways:

- As a single HDL Cosimulation block fitted into the framework of a larger system-oriented Simulink model.
- As a Simulink model made up of a collection of HDL Cosimulation blocks, each representing a specific hardware component.

The block mask contains panels for entering port and signal information, setting communication modes, adding clocks (Xcelium and ModelSim only), specifying pre- and post-simulation Tcl commands (Xcelium and ModelSim only), and defining the timing relationship.

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, you integrate the HDL representation into your Simulink model as an HDL Cosimulation block. There is one block for each supported HDL simulator. These blocks are located in the Simulink Library, within the HDL Verifier block library. As an example, the block for use with Mentor Graphics ModelSim is shown in the next figure.



You configure an HDL Cosimulation block by specifying values for parameters in a block parameters dialog box. The HDL Cosimulation block parameters dialog box consists of tabbed panes that specify the following information:

- **Ports Pane:** Block input and output ports that correspond to signals, including internal signals, of your HDL design, and an output sample time.
- **Connection Pane:** Type of communication and related settings to be used for exchanging data between simulators.
- **Timescales Pane:** The timing relationship between Simulink software and the HDL simulator.
- **Clocks Pane** (Xcelium and ModelSim only): Optional rising-edge and falling-edge clocks to apply to your model.
- **Simulation Pane** (Xcelium and ModelSim only): Tcl commands to run before and after a simulation.

See Also

HDL Cosimulation

Create Simulink Model for Component Cosimulation

For the most part, there is nothing different about creating a Simulink model to act as an HDL component than there is from creating a Simulink model to use as a test bench. When using Simulink as a component, you may have multiple HDL Cosimulation blocks rather than a single HDL Cosimulation block, though there's no limitation on how many HDL Cosimulation blocks you may use in either situation.

Note This configuration is not supported for Vivado cosimulation.

These steps describe how to cosimulate an HDL design that tests the algorithm being modeled with the Simulink software.

- 1 Create an HDL design. Compile, elaborate, and simulate your module in your HDL simulator. See “Code an HDL Component” on page 7-5.
- 2 Design algorithm and model algorithm in Simulink. Run and test your model thoroughly before replacing or adding hardware model components as Cosimulation blocks.
- 3 Start HDL simulator for use with MATLAB and Simulink and load HDL Verifier libraries. See “Start HDL Simulator for Cosimulation in Simulink” on page 5-2.
- 4 Add one or more HDL Cosimulation blocks to provide communication between simulators. See “Insert HDL Cosimulation Block” on page 7-7.
- 5 Set the parameters of the HDL Cosimulation block. See “Define HDL Cosimulation Block Interface” on page 7-7.
- 6 (Optional) Add the To VCD File block to log changes to variable values during a simulation session. See “Add a Value Change Dump (VCD) File” on page 8-2.
- 7 Start running the Simulink model first, then start cosimulation in the HDL simulator. See “Run a Simulink Cosimulation Session” on page 5-4.

Code an HDL Component

- “Specify Port Direction Modes in HDL Module for Component Simulation” on page 7-5
- “Specify Port Data Types in HDL Module for Component Simulation” on page 7-6
- “Compile and Elaborate HDL Design for Component Simulation” on page 7-7

The HDL Verifier interface passes all data between the HDL simulator and Simulink as port data. The HDL Verifier software works with any existing HDL module. However, when you code an HDL module that is targeted for Simulink verification, you should consider the types of data to be shared between the two environments and the direction modes.

Specify Port Direction Modes in HDL Module for Component Simulation

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). The following table defines these three modes.

Use VHDL Mode...	Use Verilog Mode...	For Ports That...
IN	input	Represent signals that can be driven by a MATLAB function or Simulink test bench
OUT	output	Represent signal values that are passed to a MATLAB function or Simulink test bench
INOUT	inout	Represent bidirectional signals that can be driven by or pass values to a MATLAB function or Simulink test bench

Specify Port Data Types in HDL Module for Component Simulation

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the HDL Verifier interface converts data types for the MATLAB environment, see “Supported Data Types” on page 10-35.

Note If you use unsupported types, the HDL Verifier software issues a warning and ignores the port at run time. For example, if you define your interface with five ports, one of which is a VHDL access port, at run time, then the interface displays a warning and your code sees only four ports.

Port Data Types for VHDL Entities

In your entity statement, you must define each port that you plan to test with MATLAB with a VHDL data type that is supported by the HDL Verifier software. The interface can convert scalar and array data of the following VHDL types to comparable MATLAB types:

- STD_LOGIC, STD_ULOGIC, BIT, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, and BIT_VECTOR
- INTEGER and NATURAL
- REAL
- TIME
- Enumerated types, including user-defined enumerated types and CHARACTER

The interface also supports all subtypes and arrays of the preceding types.

Note The HDL Verifier software does not support VHDL extended identifiers for the following components:

- Port and signal names used in cosimulation
- Enum literals when used as array indices of port and signal names used in cosimulation

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Entities

In your module definition, you must define each port that you plan to test with MATLAB with a Verilog port data type that is supported by the HDL Verifier software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg

- integer
- wire

Note HDL Verifier software does not support Verilog escaped identifiers for port and signal names used in cosimulation. However, it does support simple identifiers for Verilog.

Compile and Elaborate HDL Design for Component Simulation

Refer to the HDL simulator documentation for instruction in compiling and elaborating the HDL design.

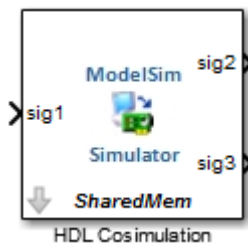
Define HDL Cosimulation Block Interface

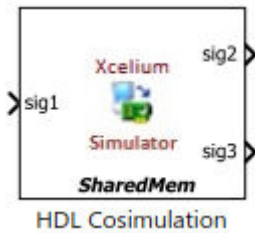
- “Insert HDL Cosimulation Block” on page 7-7
- “Connect Block Ports” on page 7-8
- “Open HDL Cosimulation Block Interface” on page 7-8
- “Map HDL Signals to Block Ports” on page 7-8
- “Specify Signal Data Types” on page 7-18
- “Configure Simulink and HDL Simulator Timing Relationship” on page 7-18
- “Configure Communication Link in the HDL Cosimulation Block” on page 7-20
- “Specify Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box” on page 7-22
- “Programmatically Control Block Parameters” on page 7-24

Insert HDL Cosimulation Block

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, integrate the HDL representation into your Simulink model as an HDL Cosimulation block by performing the following steps:

- 1 Open your Simulink model, if it is not already open.
- 2 Delete the model component that the HDL Cosimulation block is to replace.
- 3 In the Simulink Library Browser, click the HDL Verifier block library. You can then select the block library for your supported HDL simulator. Select either the Mentor Graphics ModelSim HDL Cosimulation block, or the Cadence Xcelium HDL Cosimulation block, as shown below.





- Copy the HDL Cosimulation block from the Library Browser to your model.

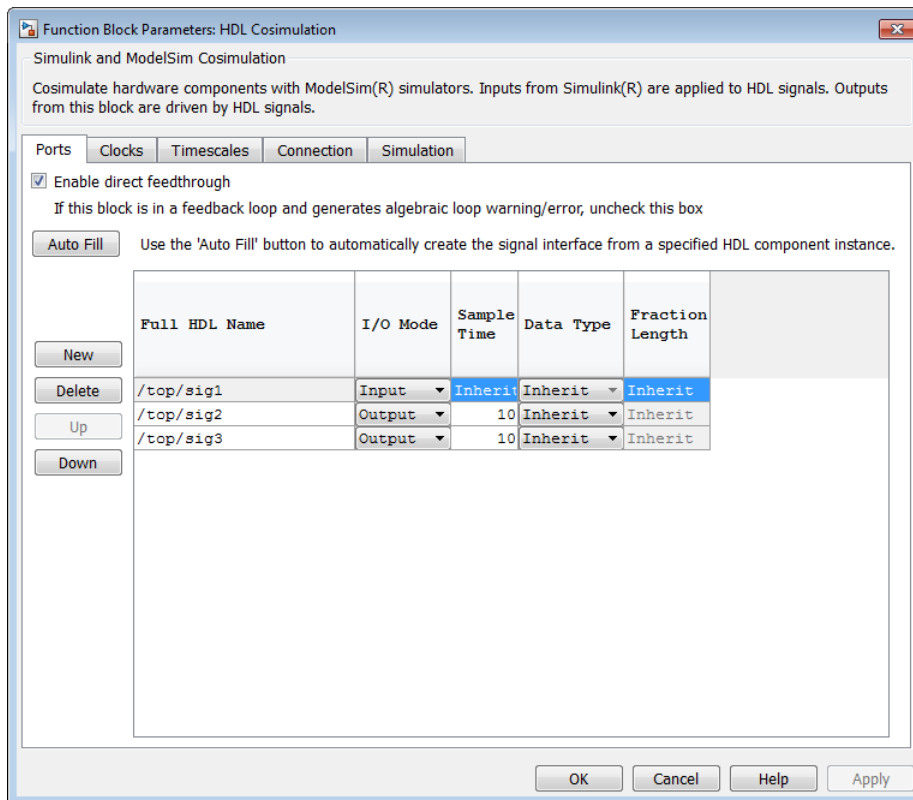
Connect Block Ports

Connect any HDL Cosimulation block ports to the applicable block ports in your Simulink model.

- To model a sink device, configure the block with inputs only.
- To model a source device, configure the block with outputs only.

Open HDL Cosimulation Block Interface

To open the block parameters dialog box for the HDL Cosimulation block, double-click the block icon. Simulink displays the following Block Parameters dialog box (as an example, the dialog box for the HDL Cosimulation block for use with ModelSim is shown below).



Map HDL Signals to Block Ports

- “Specify HDL Signal/Port and Module Paths for Simulink Component Cosimulation” on page 7-9

- “Get Signal Information from the HDL Simulator” on page 7-10
- “Enter Signal Information Manually” on page 7-14
- “Import Signal Information Directly by Value of Input Port” on page 7-18

The first step to configuring your HDL Verifier HDL Cosimulation block is to map signals and signal instances of your HDL design to port definitions in your HDL Cosimulation block. In addition to identifying input and output ports, you can specify a sample time for each output port. You can also specify a fixed-point data type for each output port.

The signals that you map can be at any level of the HDL design hierarchy.

To map the signals, you can perform either of the following actions:

- Enter signal information manually into the **Ports** pane of the block parameters dialog box. This approach can be more efficient when you want to connect a small number of signals from your HDL model to Simulink.
- Use the **Auto Fill** button to have the HDL Cosimulation block obtain signal information for you by transmitting a query to the HDL simulator. This approach can save significant effort when you want to cosimulate an HDL model that has many signals that you want to connect to your Simulink model. However, in some cases, you will need to edit the signal data returned by the query.

Note Verify that signals used in cosimulation have read/write access. For higher performance, you want to provide access only to those signals used in cosimulation. This rule applies to all signals on the **Ports**, **Clocks**, and **Simulation** panes, and to all signals added in any other manner.

Specify HDL Signal/Port and Module Paths for Simulink Component Cosimulation

These rules are for signal/port and module path specifications in Simulink. Other specifications may work but are not explicitly or implicitly supported in this or future releases.

HDL designs generally do have hierarchy; that is the reason for this syntax. This specification does not represent a file name hierarchy.

Path Specifications for Verilog Top Level

- Path specification must start with a top-level module name.
- Path specification can include "." or "/" path delimiters, but cannot include a mixture.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/top/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- top.sub/port_or_sig

Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`
:
:sub

Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Path Specifications for VHDL Top Level

- Path specification may include the top-level module name but it is not required.
- Path specification can include "." or "/" path delimiters, but cannot include a mixture.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig  
/sub/port_or_sig  
top  
top/sub  
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`
:
:sub

Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

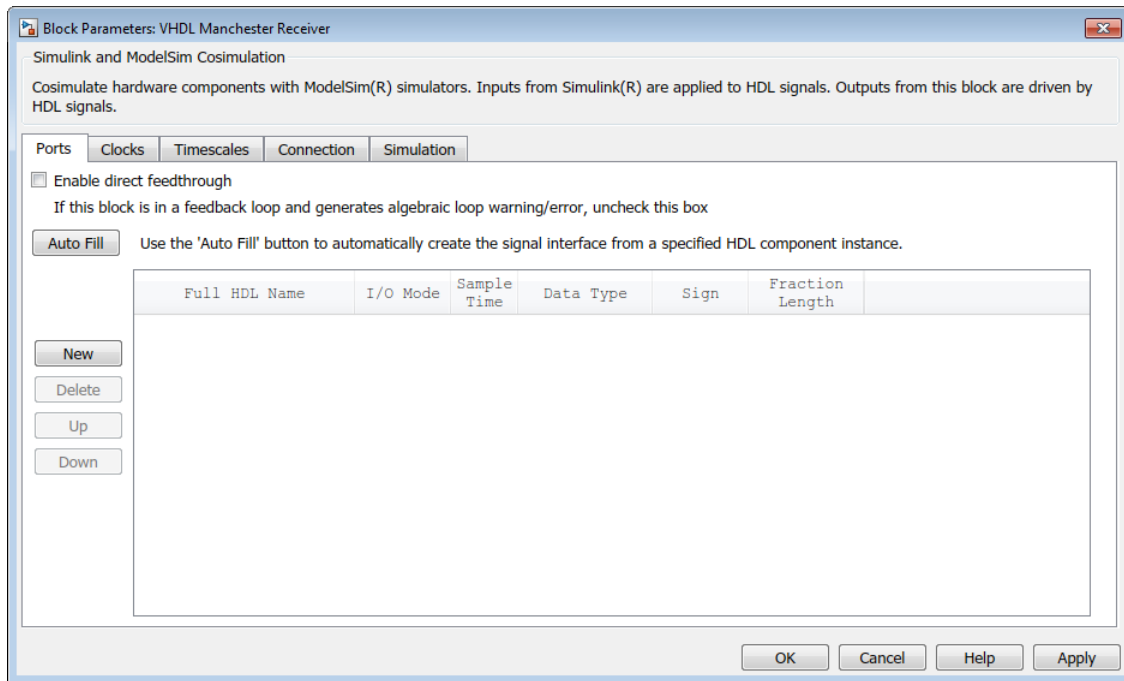
Get Signal Information from the HDL Simulator

The **Auto Fill** button lets you begin an HDL simulator query and supply a path to a component or module in an HDL model under simulation in the HDL simulator. Usually, some change of the port information is required after the query completes. You must have the HDL simulator running with the HDL module loaded for **Auto Fill** to work.

The following example describes the required steps.

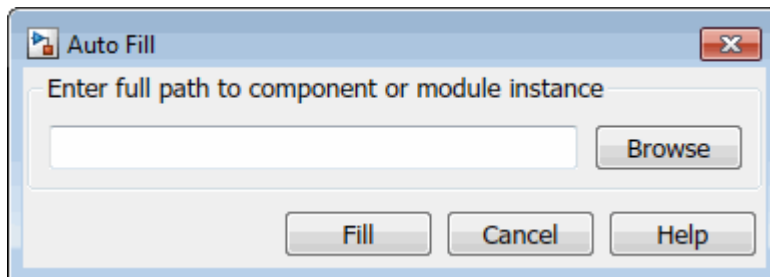
Note The example is based on a modified copy of the Manchester Receiver model, in which all signals were first deleted from the **Ports** and **Clocks** panes.

- 1 Open the block parameters dialog box for the HDL Cosimulation block. Click the **Ports** tab. The **Ports** pane opens (as an example, the **Ports** pane for the HDL Cosimulation block for use with ModelSim is shown in the illustrations below).



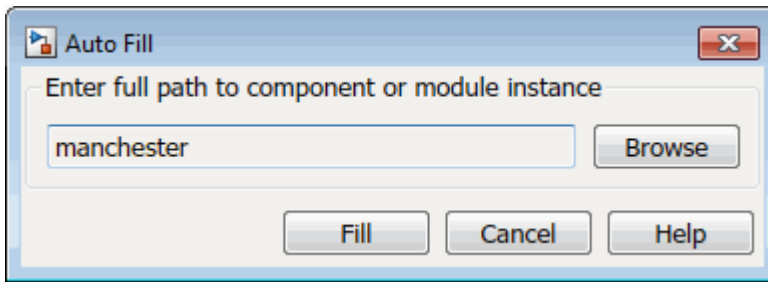
Tip Delete all ports before performing **Auto Fill** to make sure that no unused signal remains in the Ports list at any time.

- 2 Click the **Auto Fill** button. The **Auto Fill** dialog box opens.

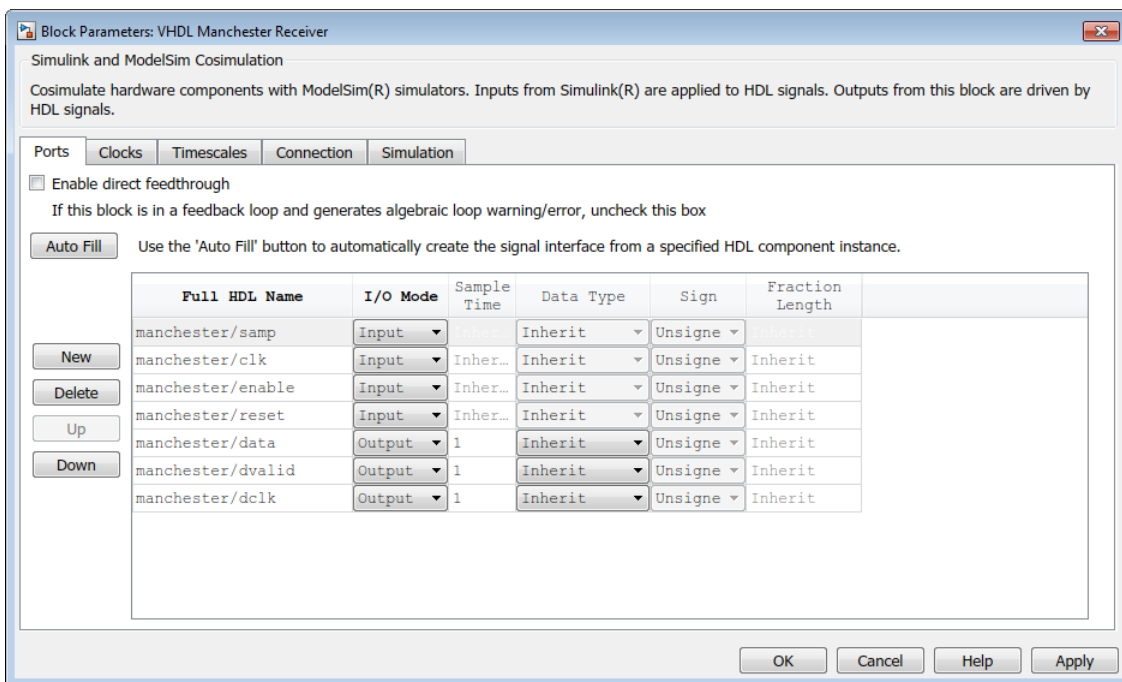


This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. The path you enter is not a file path and has nothing to do with the source files.

- 3 In this example, the Auto Fill feature obtains port data for a VHDL component called `manchester`. The HDL path is specified as `/top/manchester`. Path specifications will vary depending on your HDL simulator, see “Specify HDL Signal/Port and Module Paths for Simulink Component Cosimulation” on page 7-9.



- 4 Click **Fill** to dismiss the dialog box and the query is transmitted.
- 5 After the HDL simulator returns the port data, the Auto Fill feature enters it into the **Ports** pane, as shown in the following figure (examples shown for use with Cadence Xcelium).

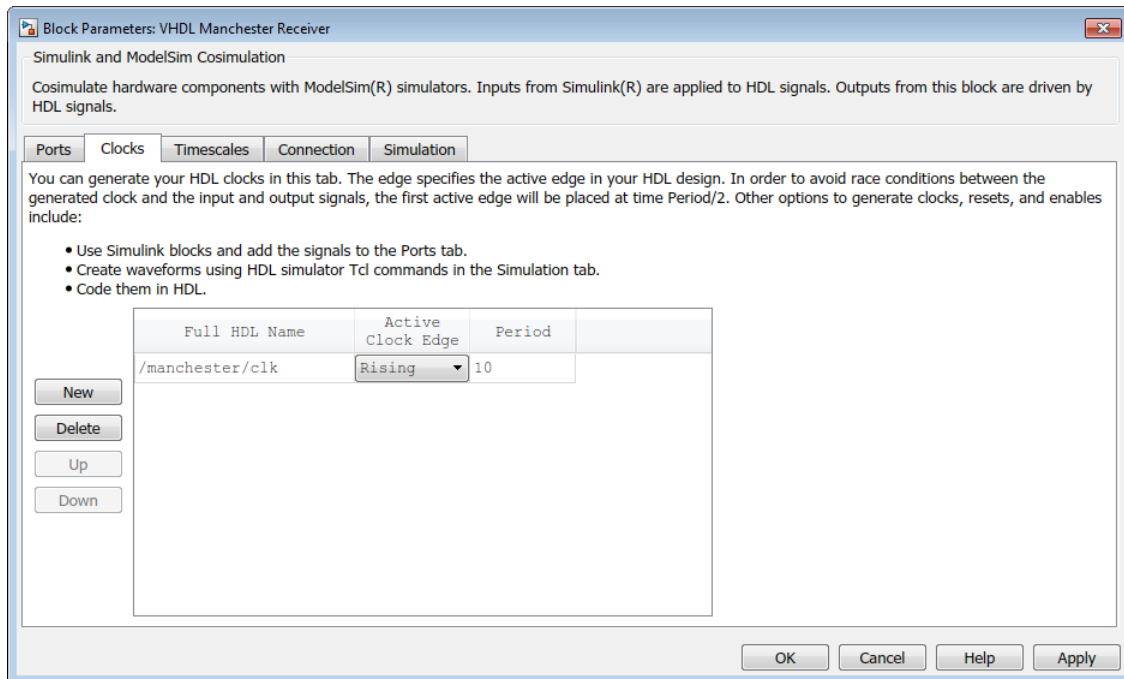
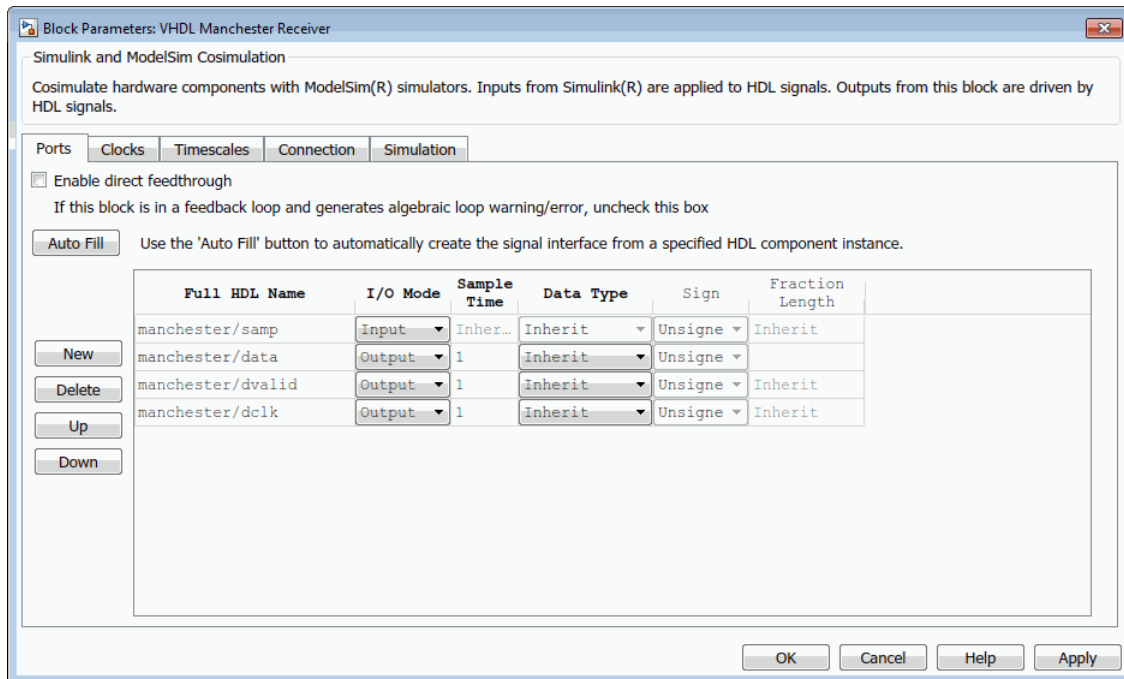


- 6 Click **Apply** to commit the port additions.
- 7 Delete unused signals from Ports pane and add Clock signal.

The preceding figure shows that the query entered clock, clock enable, and reset ports (labeled clk, enable, and reset respectively) into the ports list.

Delete the clk, enable and reset signals from the **Ports** pane, and add the clk signal in the **Clocks** pane.

These actions result in the signals shown in the next figures.



8 **Auto Fill** returns default values for output ports:

- **Sample time:** 1
- **Data type:** Inherit
- **Fraction length:** Inherit

You may need to change these values as required by your model. In this example, the **Sample time** should be set to 10 for all outputs. See “Specify Signal Data Types”.

- 9 Before closing the block parameters dialog box, click **Apply** to commit any edits you have made.

Observe that **Auto Fill** returned information about *all* inputs and outputs for the targeted component. In many cases, this will include signals that function in the HDL simulator but cannot be connected in the Simulink model. You may delete any such entries from the list in the **Ports** pane if they are unwanted. You *can* drive the signals from Simulink; you just have to define their values by laying down Simulink blocks.

Note that **Auto Fill** does not return information for internal signals. If your Simulink model needs to access such signals, you must enter them into the **Ports** pane manually. For example, in the case of the Manchester Receiver model, you would need to add output port entries for `top/manchester/sync_i`, `top/manchester/isum_i`, and `top/manchester/qsum_i`, as shown in step 8.

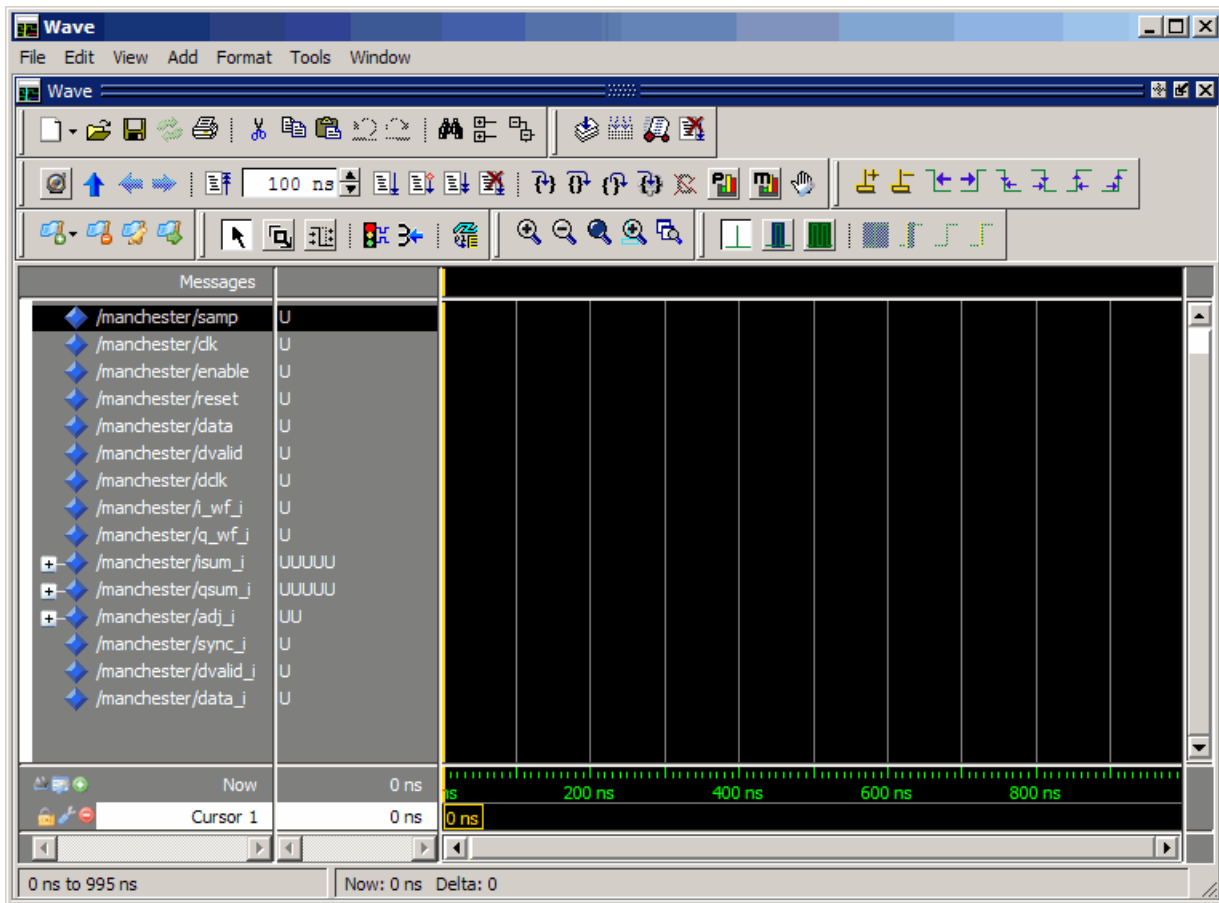
Xcelium and ModelSim users: Note that `clk`, `reset`, and `clk_enable` *may* be in the Clocks and Simulation panes but they don't *have* to be. These signals can be ports if you choose to drive them explicitly from Simulink.

Note When you import VHDL signals using **Auto Fill**, the HDL simulator returns the signal names in all capitals.

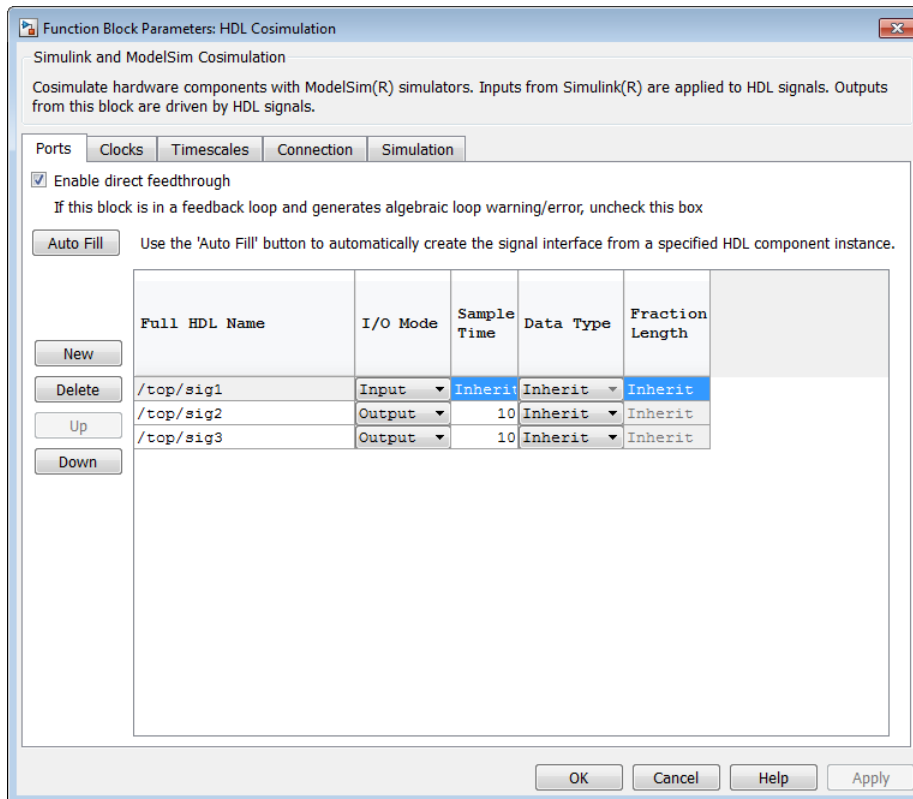
Enter Signal Information Manually

To enter signal information directly in the **Ports** pane, perform the following steps:

- 1 In the HDL simulator, determine the signal path names for the HDL signals you plan to define in your block. For example, in the ModelSim simulator, the following wave window shows all signals are subordinate to the top-level module `manchester`.



- 2 In Simulink, open the block parameters dialog box for your HDL Cosimulation block, if it is not already open.
- 3 Select the **Ports** pane tab. Simulink displays the following dialog box (example shown for use with Xcelium).



In this pane, you define the HDL signals of your design that you want to include in your Simulink block and set a sample time and data type for output ports. The parameters that you should specify on the **Ports** pane depend on the type of device the block is modeling as follows:

- For a device having both inputs and outputs: specify block input ports, block output ports, output sample times and output data types.

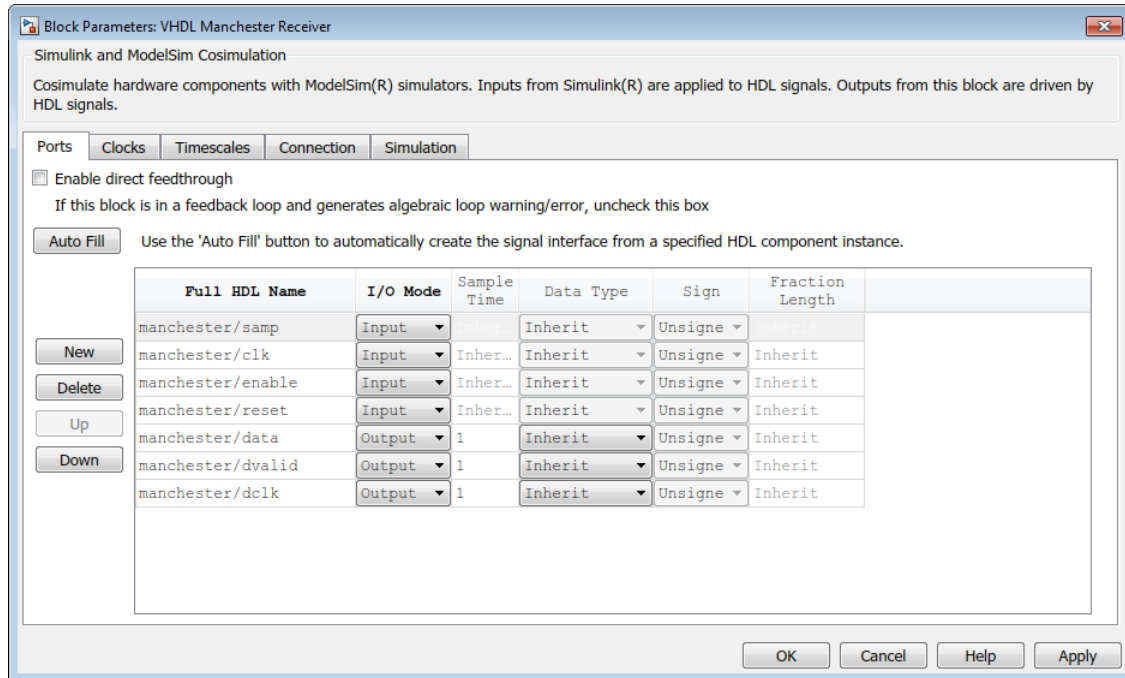
For output ports, accept the default or enter an explicit sample time. Data types can be specified explicitly, or set to `Inherit` (the default). In the default case, the output port data type is inherited either from the signal connected to the port, or derived from the HDL model.

- For a sink device: specify block output ports.
- For a source device: specify block input ports.

- 4 Enter signal path names in the **Full HDL name** column by double-clicking on the existing default signal.
 - Use HDL simulator path name syntax (as described in “Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation” on page 6-10).
 - If you are adding signals, click **New** and then edit the default values. Select either **Input** or **Output** from the **I/O Mode** column.
 - If you want to, set the **Sample Time**, **Data Type**, and **Fraction Length** parameters for signals explicitly, as discussed in the remaining steps.

When you have finished editing clock signals, click **Apply** to register your changes with Simulink.

The following dialog box shows port definitions for an HDL Cosimulation block. The signal path names match path names that appear in the HDL simulator **wave** window (Xcelium example shown).



Note When you define an input port, make sure that only one source is set up to force input to that port. If multiple sources drive a signal, your Simulink model may produce unpredictable results.

- 5 You must specify a sample time for the output ports. Simulink uses the value that you specify, and the current settings of the **Timescales** pane, to calculate an actual simulation sample time.

For more information on sample times in the HDL Verifier cosimulation environment, see “Simulation Timescales” on page 10-44.

- 6 You can configure the fixed-point data type of each output port explicitly if desired, or use a default (Inherited). In the default case, Simulink determines the data type for an output port as follows:

If Simulink can determine the data type of the signal connected to the output port, it applies that data type to the output port. For example, the data type of a connected Signal Specification block is known by back-propagation. Otherwise, Simulink queries the HDL simulator to determine the data type of the signal from the HDL module.

To assign an explicit fixed-point data type to a signal, perform the following steps:

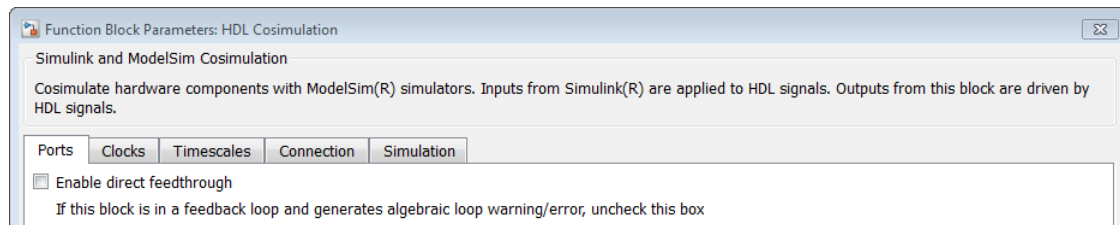
- a Select either Signed or Unsigned from the **Data Type** column.
- b If the signal has a fractional part, enter the **Fraction Length**.

For example, if the model has an 8-bit signal with Signed data type and a **Fraction Length** of 5, the HDL Cosimulation block assigns it the data type `sfixed8_En5`. If the model has an Unsigned 16-bit signal with no fractional part (a **Fraction Length** of 0), the HDL Cosimulation block assigns it the data type `ufixed16`.

7 Before closing the dialog box, click **Apply** to register your edits.

Import Signal Information Directly by Value of Input Port

Enabling direct feedthrough allows input port value changes to propagate to the output ports in zero time, thus eliminating the possible delay at output sample in HDL designs with pure combinational logic. Specify the option to enable direct feedthrough on the **Ports** pane, as shown in the following figure.



Specify Signal Data Types

The **Data Type** and **Fraction Length** parameters apply only to output signals. See **Data Type** and **Fraction Length** on the Ports pane description of the HDL Cosimulation block.

Configure Simulink and HDL Simulator Timing Relationship

You configure the timing relationship between Simulink and the HDL simulator by using the **Timescales** pane of the block parameters dialog box. Before setting the **Timescales** parameters, read “Simulation Timescales” on page 10-44 to understand the supported timing modes and the issues that will determine your choice of timing mode.

You can specify either a relative or an absolute timing relationship between Simulink and the HDL simulator in the **Timescales** pane, as described in the HDL Cosimulation block reference.

Define Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the HDL Verifier interface:

- By defining the timing relationship manually (with **Timescales** pane)

When you define the relationship manually, you determine how many femtoseconds, picoseconds, nanoseconds, microseconds, milliseconds, seconds, or ticks in the HDL simulator represent 1 second in Simulink.

1 second in Simulink corresponds to in the HDL simulator

This quantity of HDL simulator time can be expressed in one of the following ways:

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation block defaults to relative timing mode for cosimulation. For more on relative timing mode, see “Relative Timing Mode” on page 10-48.
- In *absolute* units (such as milliseconds or nanoseconds). In this case, the cosimulation is said to operate in *absolute timing mode*. For more on absolute timing mode, see “Absolute Timing Mode” on page 10-51.

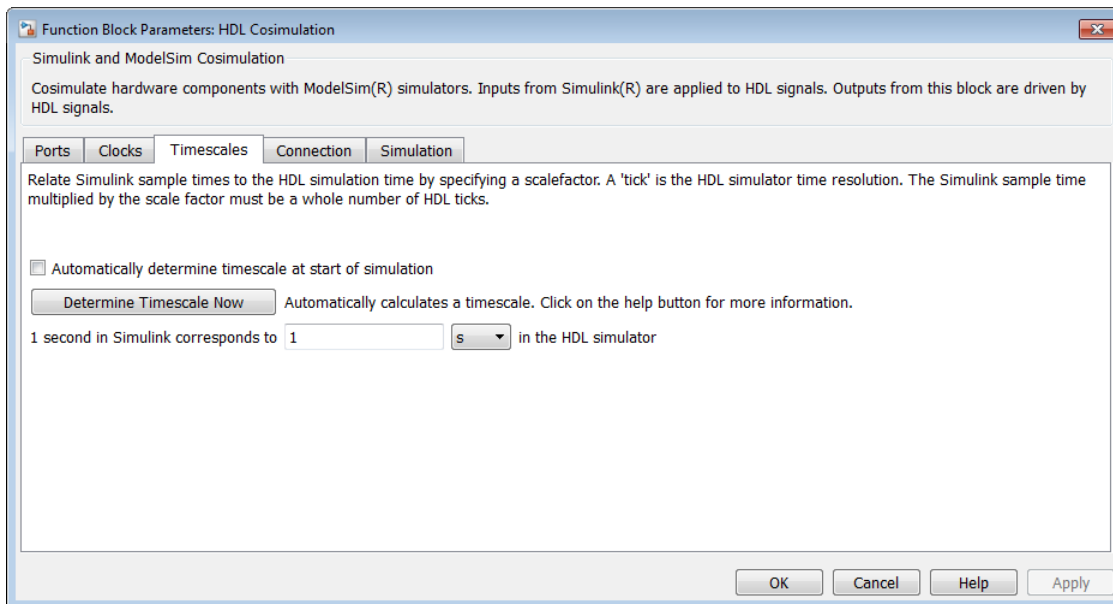
For more on relative and absolute time, see “Simulation Timescales” on page 10-44.

- By allowing HDL Verifier to define the timescale (with **Timescales** pane)

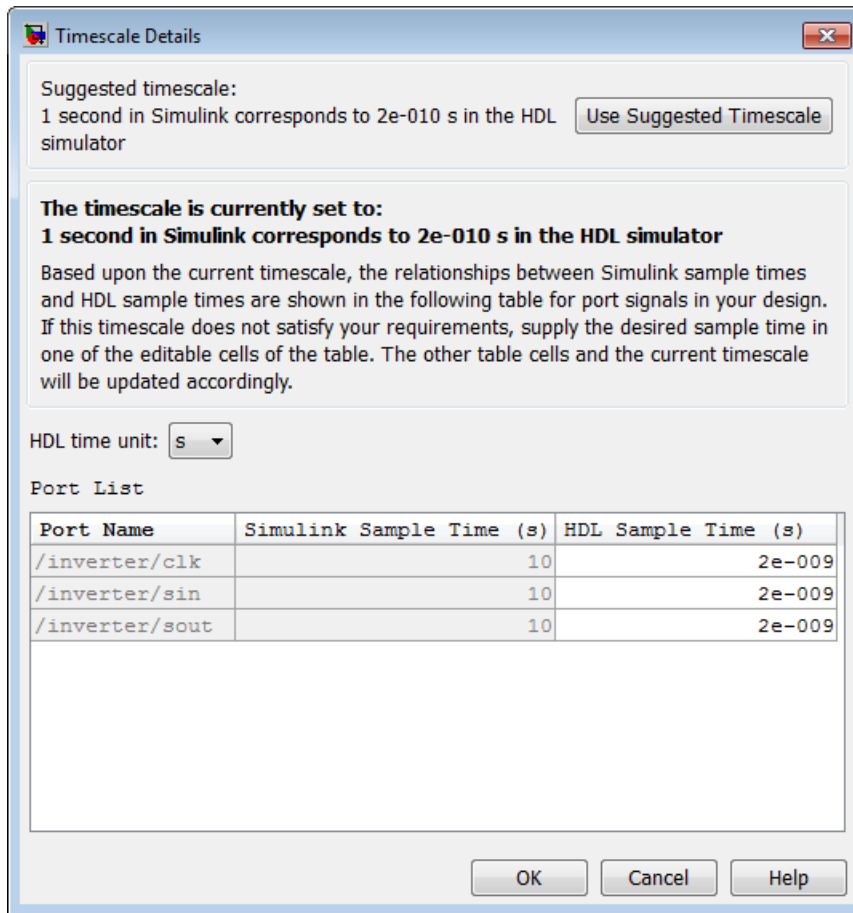
When you allow the link software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, the link product attempts to set the signal rate on the Simulink model port to the lowest possible number of HDL simulator ticks.

Before you begin, verify that the HDL simulator is running. HDL Verifier software can get the resolution limit of the HDL simulator only when that simulator is running.

You can choose to have HDL Verifier calculate a timescale while you are setting the parameters on the block dialog by clicking the **Timescale** option then clicking **Determine Timescale Now** or you can have HDL Verifier calculate the timescale when simulation begins by selecting **Automatically determine timescale at start of simulation**.



When you click **Determine Timescale Now**, HDL Verifier connects Simulink with the HDL simulator so that it can use the HDL simulator resolution to calculate the best timescale. You can accept the timescale HDL Verifier suggests or you can make changes in the port list directly. If you want to revert to the originally calculated settings, click **Use Suggested Timescale**. If you want to view sample times for all ports in the HDL design, select **Show all ports and clocks**.

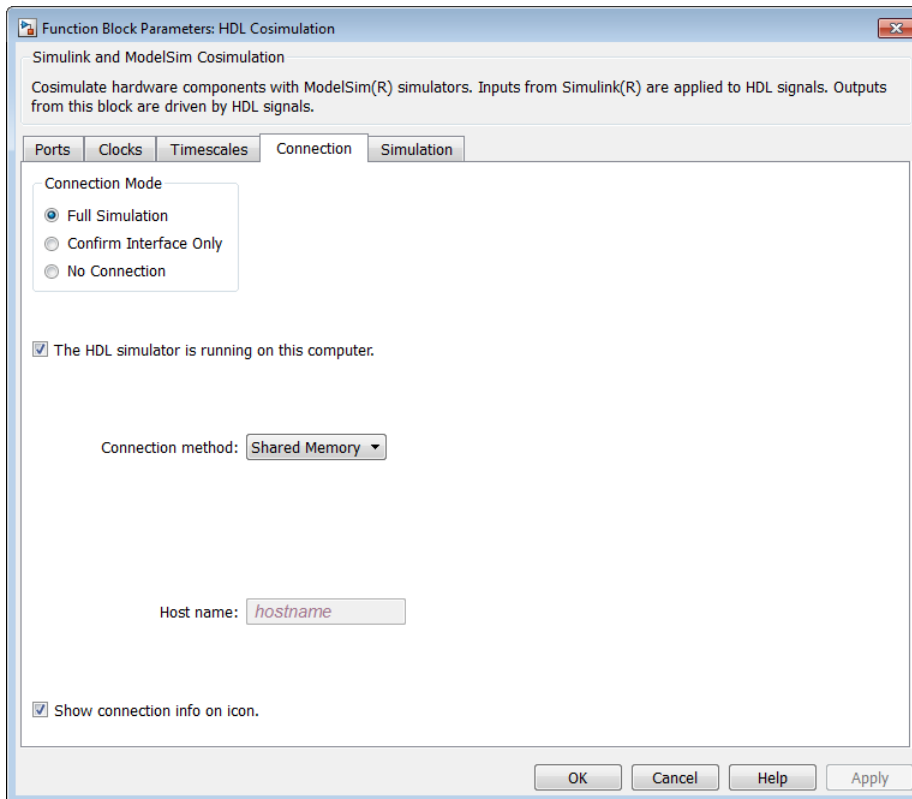


If you select **Automatically determine timescale at start of simulation**, you get the same dialog when the simulation starts in Simulink. Make the same adjustments at that time, if applicable, that you would if you clicked **Determine Timescale Now** when you were configuring the block.

Configure Communication Link in the HDL Cosimulation Block

You must select shared memory or socket communication. See “HDL Cosimulation with MATLAB or Simulink”.

After you decide which type of communication, configure a block's communication link with the **Connection** pane of the block parameters dialog box (example shown for use with ModelSim).



The following steps guide you through the communication configuration:

- 1 Determine whether Simulink and the HDL simulator are running on the same computer. If they are, skip to step 4.
- 2 Clear **The HDL simulator is running on this computer**. (This check box defaults to selected.) Because Simulink and the HDL simulator are running on different computers, HDL Verifier sets the **Connection method** to Socket.
- 3 Enter the host name of the computer that is running your HDL simulation (in the HDL simulator) in the **Host name** text field. In the **Port number or service** text field, specify a valid port number or service for your computer system. For information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports” on page 10-61. Skip to step 5.
- 4 If the HDL simulator and Simulink are running on the same computer, decide whether you are going to use shared memory or TCP/IP sockets for the communication channel. For information on the different modes of communication, see “HDL Cosimulation with MATLAB or Simulink”.

If you choose TCP/IP socket communication, specify a valid port number or service for your computer system in the **Port number or service** text field. For information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports” on page 10-61.

If you choose shared memory communication, select the **Shared memory** check box.

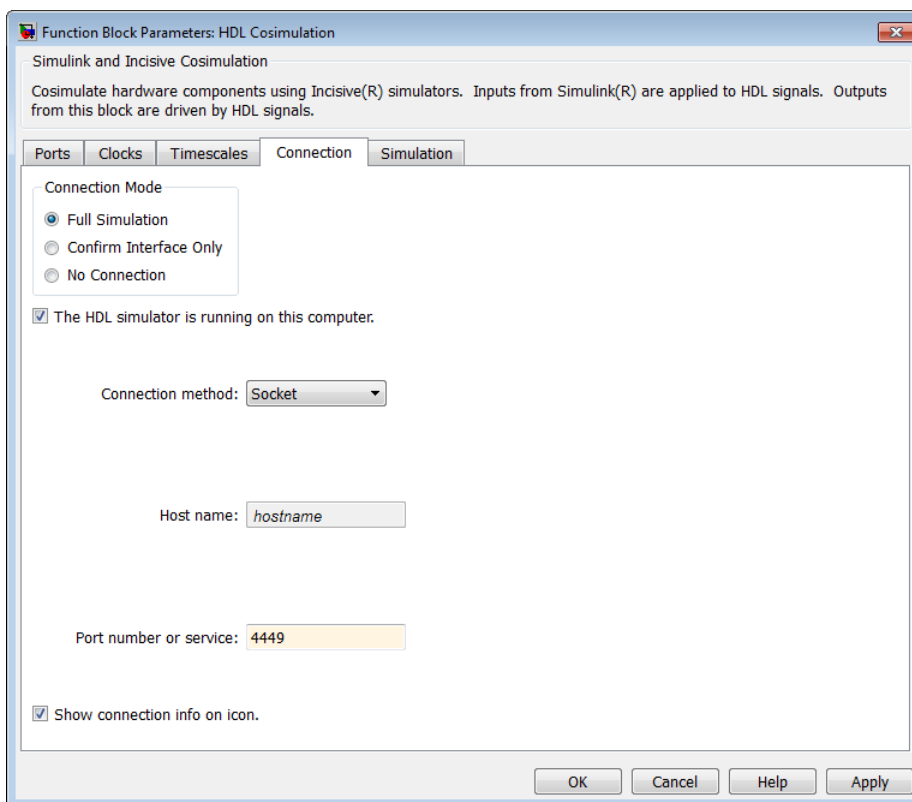
- 5 If you want to bypass the HDL simulator when you run a Simulink simulation, use the **Connection Mode** options to specify what type of simulation connection you want. Select one of the following options:
 - **Full Simulation**: Confirm interface and run HDL simulation (default).

- **Confirm Interface Only:** Check HDL simulator for expected signal names, dimensions, and data types, but do not run HDL simulation.
- **No Connection:** Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

With the second and third options, HDL Verifier software does not communicate with the HDL simulator during Simulink simulation.

6 Click **Apply**.

The following example dialog box shows communication definitions for an HDL Cosimulation block. The block is configured for Simulink and the HDL simulator running on the same computer, communicating in TCP/IP socket mode over TCP/IP port 4449.



Specify Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box

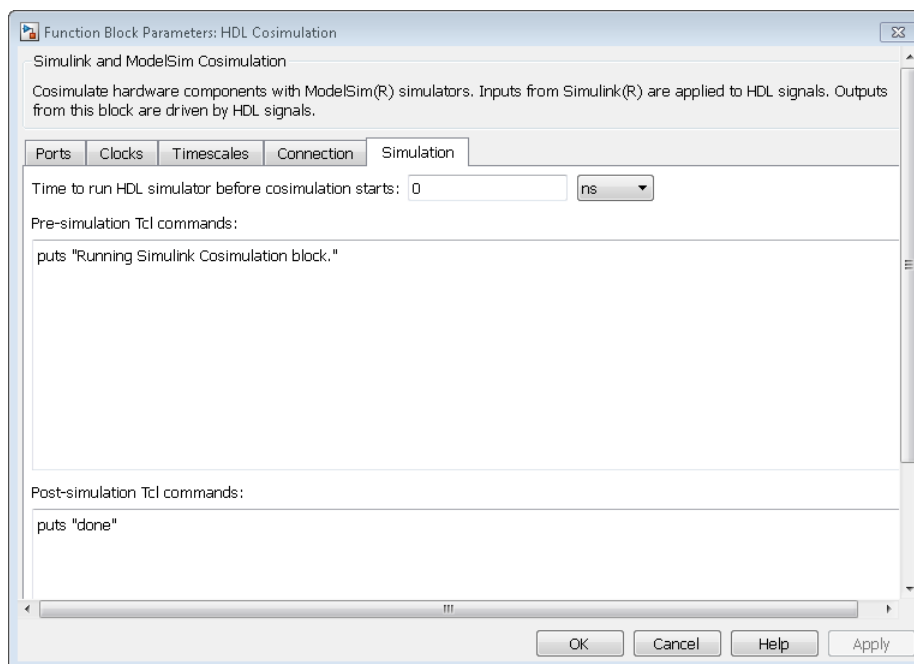
You have the option of specifying Tcl commands to execute before and after the HDL simulator simulates the HDL component of your Simulink model. Tcl is a programmable scripting language supported by most HDL simulation environments. Use of Tcl can range from something as simple as a one-line `puts` command to confirm that a simulation is running or as complete as a complex script that performs an extensive simulation initialization and startup sequence. For example, you can use the **Post-simulation command** field on the Simulation Pane to instruct the HDL simulator to restart at the end of a simulation run.

Note for ModelSim Users After each simulation, it takes ModelSim time to update the coverage result. To prevent the potential conflict between this process and the next cosimulation session, add a short pause between each successive simulation.

You can specify the pre-simulation and post-simulation Tcl commands by entering Tcl commands in the **Pre-simulation** commands or **Post-simulation** commands text fields in the **Simulation** pane of the HDL Cosimulation block parameters dialog box.

To specify Tcl commands, perform the following steps:

- 1 Select the **Simulation** tab of the block parameters dialog box. The dialog box appears as follows (example shown for use with ModelSim).

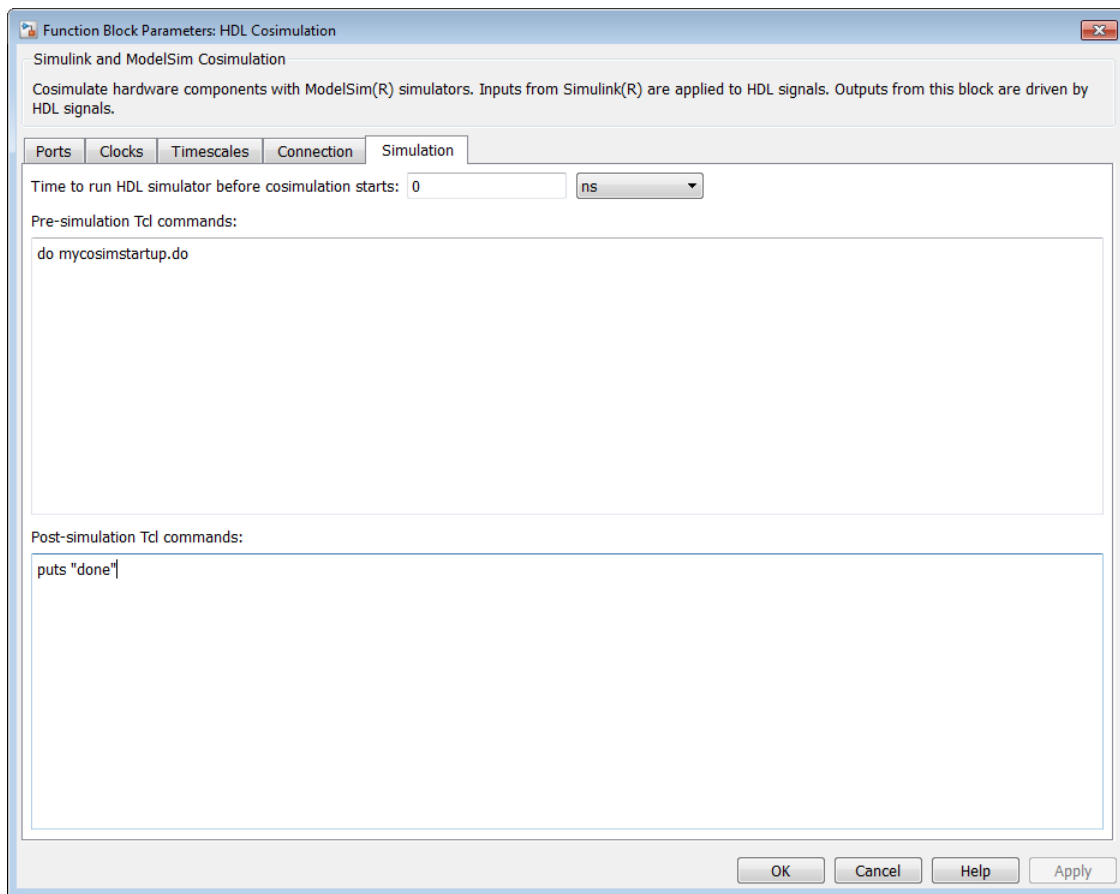


The **Pre-simulation commands** text box includes a `puts` command for reference purposes.

- 2 Enter one or more commands in the **Pre-simulation command** and **Post-simulation command** text boxes. You can specify one Tcl command per line in the text box or enter multiple commands per line by appending each command with a semicolon (;), which is the standard Tcl concatenation operator.

ModelSim DO Files

Alternatively, you can create a ModelSim DO file that lists Tcl commands and then specify that file with the ModelSim `do` command as shown in the following figure.



- 3 Click **Apply**.

Programmatically Control Block Parameters

One way to control block parameters is through the HDL Cosimulation block graphical dialog box. However, you can also control blocks by programmatically controlling the mask parameter values and the running of simulations. Parameter values can be read using the Simulink `get_param` function and written using the Simulink `set_param` function. All block parameters have attributes that indicate whether they are:

- Tunable — The attributes can change during the simulation run.
- Evaluated — The parameter value undergoes an evaluation to determine its actual value used by the S-Function.

The HDL Cosimulation block does not have any tunable parameters; thus, you get an error if you try to change a value while the simulation is running. However, it does have a few evaluated parameters.

You can see the list of parameters and their attributes by performing a right-mouse click on the block, selecting **View Mask**, and then the **Parameters** tab. The **Variable** column shows the programmatic parameter names. Alternatively, you can get the names programmatically by selecting the HDL Cosimulation block and then typing the following commands at the MATLAB prompt:

```
>> get_param(gcb, 'DialogParameters')
```


Some examples of using MATLAB to control simulations and mask parameter values follow. Usually, the commands are put into a script or function file and are called by several callback hooks available to the model developer. You can place the code in any of these suggested Simulink locations:

- In the model workspace. On the **Modeling** tab, in the **Design** section, click **Model Explorer**. In the Model Explorer dialog box, in the **Model Hierarchy** pane, select **Simulink Root > model_name > Model Workspace**. In the **Model Workspace** pane, from the **Data source** list, select **Model File**.
- In a model callback. On the **Modeling** tab, in the **Setup** section, click **Model Settings > Model Properties**. In the Model Properties dialog box, specify the callback function in the **Callbacks** tab.
- In a subsystem callback. Right-mouse click on an empty subsystem and then select **Properties > Callbacks**. Many of the HDL Verifier demos use this technique to start the HDL simulator by placing MATLAB code in the **OpenFcn** callback.
- In the HDL Cosimulation block callback. Right-mouse click on HDL Cosimulation block, and then select **Properties > Callbacks**.

Example: Scripting the Value of the Socket Number for HDL Simulator Communication

In a regression environment, you may need to determine the socket number for the Simulink/HDL simulator connection during the simulation to avoid collisions with other simulation runs. This example shows code that could handle that task. The script is for a 32-bit Linux platform.

```
ttcp_exec = [matlabroot '/toolbox/shared/hdlink/scripts/ttcp_glnx'];
[status, results] = system([ttcp_exec ' -a']);
if ~s
    parsed_result = textscan(results, '%s');
    avail_port = parsed_result{1}{2};
else
    error(results);
end

set_param('MyModel/HDL Cosimulation', 'CommPortNumber', avail_port);
```


Record Simulink Signal State Transitions for Post-Processing

- “Add a Value Change Dump (VCD) File” on page 8-2
- “Visually Compare Simulink Signals with HDL Signals” on page 8-5

Add a Value Change Dump (VCD) File

In this section...
“Introduction to the To VCD File Block” on page 8-2
“Using the To VCD File Block” on page 8-2

Introduction to the To VCD File Block

A value change dump (VCD) file logs changes to variable values, such as the values of signals, in a file during a simulation session. VCD files can be useful during design verification. Some examples of how you might apply VCD files include the following cases:

- For comparing results of multiple simulation runs, using the same or different simulator environments
- As input to post-simulation analysis tools
- For porting areas of an existing design to a new design

VCD files can provide data that you might not otherwise acquire unless you understood the details of a device's internal logic. In addition, they include data that can be graphically displayed or analyzed with post processing tools, including, for example, the extraction of data about a particular section of a design hierarchy or data generated during a specific time interval.

Another example, this specifically for ModelSim users, is the ModelSim `vcd2wlf` tool, which converts a VCD file to a Wave Log Format (WLF) file that you can view in a ModelSim **wave** window.

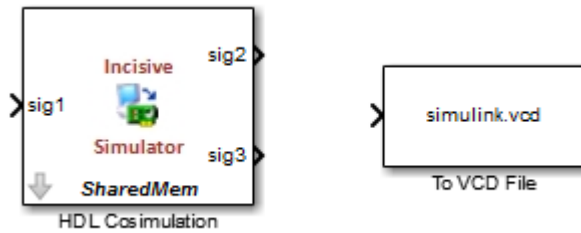
The To VCD File block provided in the HDL Verifier block library serves as a VCD file generator during Simulink sessions. The block generates a VCD file that contains information about changes to signals connected to the block's input ports and names the file with a specified file name.

Note The To VCD File block logs changes to states '1' and '0' only. The block does *not* log changes to states 'X' and 'Z'.

Using the To VCD File Block

To generate a VCD file, perform the following steps:

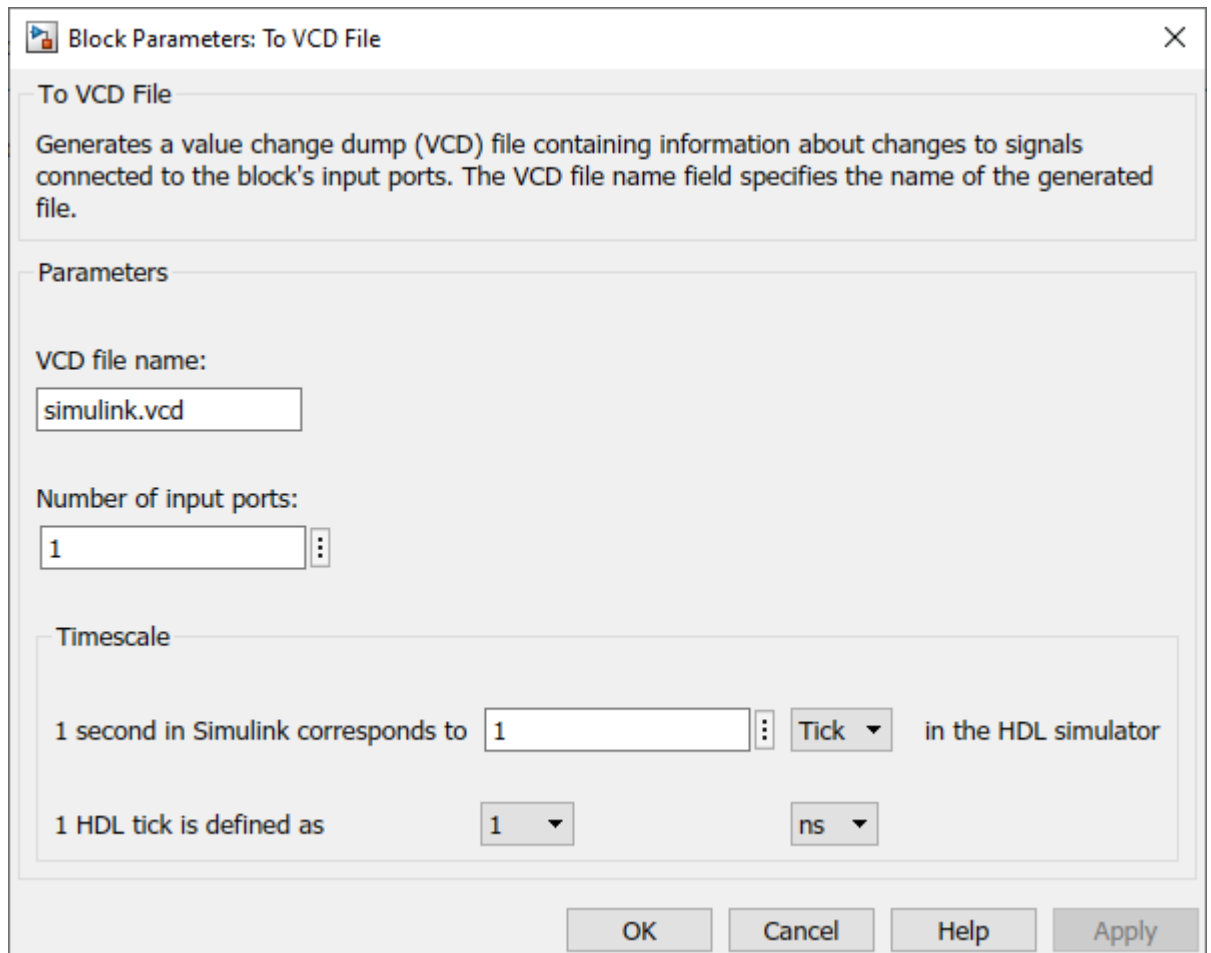
- 1 Open your Simulink model, if it is not already open.
- 2 Identify where you want to add the To VCD File block. For example, you might temporarily replace a scope with this block.
- 3 In the Simulink Library Browser, click HDL Verifier and then select the block library for your HDL simulator. You will see the HDL Cosimulation block icon and the To VCD File block icon.



- 4 Copy the To VCD File block from the Library Browser to your model by clicking the block and dragging it from the browser to your model window.
- 5 Connect the block ports to the applicable blocks in your Simulink model.

Note Because multidimensional signals are not part of the VCD specification, they are flattened to a 1-D vector in the file.

- 6 Configure the To VCD File block by specifying values for parameters in the Block Parameters dialog box, as follows:
 - a Double-click the block icon. Simulink displays the following dialog box.



- b** Specify a file name for the generated VCD file in the **VCD file name** text box.
 - If you specify a file name only, Simulink places the file in your current MATLAB folder.
 - Specify a complete path name to place the generated file in a different location.
 - If you want the generated file to have a .vcd file type extension, you must specify it explicitly.

Note Do not give the same file name to different VCD blocks. Doing so results in invalid VCD files.

- c** Specify an integer in the **Number of input ports** text box that indicates the number of block input ports on which signal data is to be collected. The block can handle up to 94^3 (830,584) signals, each of which maps to a unique symbol in the VCD file.
 - d** Click **OK**.
- 7** Choose a timing relationship between Simulink and the HDL simulator. The time scale options specify a correspondence between one second of Simulink time and some quantity of HDL simulator time. Choose relative time or absolute time. For more on the To VCD File time scale, see the reference documentation for the To VCD File block.
- 8** Run the simulation. Simulink captures the simulation data in the VCD file as the simulation runs.

For a description of the VCD file format see “VCD File Format”. For a sample application of a VCD file, see “Visually Compare Simulink Signals with HDL Signals” on page 8-5.

Visually Compare Simulink Signals with HDL Signals

In this section...
“Tutorial: Overview” on page 8-5
“Tutorial: Instructions” on page 8-5

Tutorial: Overview

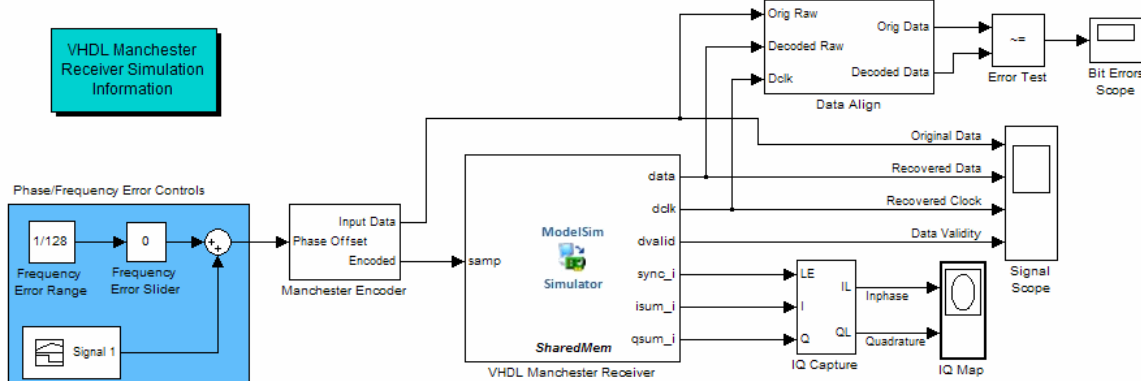
Note This tutorial and the tool used are specific to ModelSim users; however, much of the process will be the same for Xcelium users with a similar tool. See HDL simulator documentation for details.

VCD files include data that can be graphically displayed or analyzed with post-processing tools. An example of such a tool is the ModelSim `vcd2wlf` tool, which converts a VCD file to a WLF file that you can then view in a ModelSim **wave** window. This tutorial shows how you might apply the `vcd2wlf` tool.

Tutorial: Instructions

Perform the following steps to view VCD data:

- 1 Place a copy of the Manchester Receiver Simulink example model `manchestermodel` in a writable folder.
- 2 Open your writable copy of the Manchester Receiver model. On the **Simulation** tab, in the **File** section, click **Open**. Select the file `manchestermodel` and click **Open**. The Simulink model should appear as follows. The HDL Cosimulation block is marked “VHDL Manchester Receiver”.



Before running this model you must first launch ModelSim.
You can launch ModelSim on this computer using either a shared memory link or a TCP/IP socket link.

Shared memory link:

- 1) Be sure that the 'Connection' tab of the Cosimulation block dialog is set as follows:
'ModelSim running on this computer' is checked and 'Shared memory' is selected
- 2) Execute the following MATLAB command:
`vsim('tclstart','manchestercmds')`
- 3) Start the Simulink simulation.

```
vsim('tclstart','manchestercmds')
%Double-click here to launch a new ModelSim
```

ModelSim Startup Command(Shared Memory)

TCP/IP socket link:

- 1) Be sure that the 'Connection' tab of the Cosimulation block dialog is set as follows:
'ModelSim running on this computer' is checked and 'Socket' is selected
'Port number or service' matches the port number used in the command below.
- 2) Execute the following MATLAB command:
`vsim('tclstart','manchestercmds','socketsimulink',4442)`
- 3) Start the Simulink simulation.

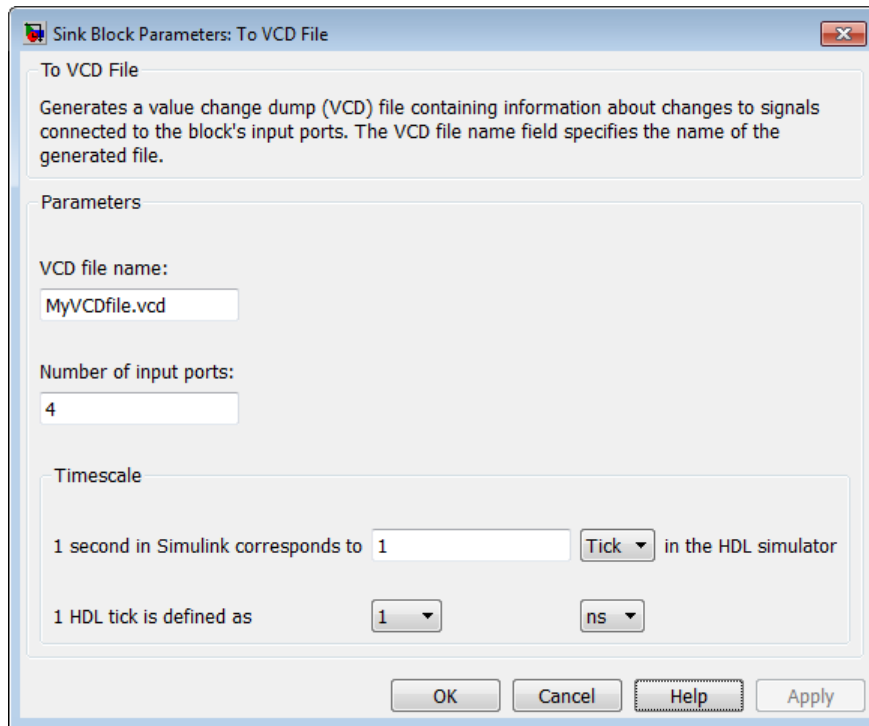
```
vsim('tclstart','manchestercmds','socketsimulink',4442)
%Double-click here to launch a new ModelSim
```

ModelSim Startup Command(TCP/IP Socket)

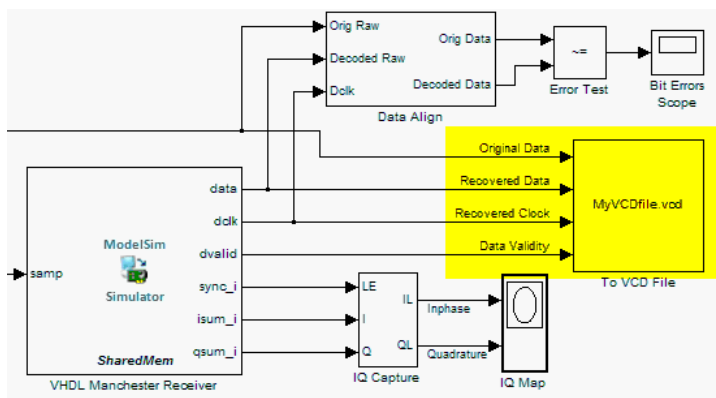
Copyright 2003-2009 The MathWorks, Inc.

Do not follow the numbered steps in the Manchester Receiver model. Follow only the steps provided in this tutorial.

- 3 Open the Library Browser.
- 4 Replace the Signal Scope block with a To VCD File block, as follows:
 - a Delete the Signal Scope block. The lines representing the signal connections to that block change to dashed lines, indicating the disconnection.
 - b Find and open the HDL Verifier block library.
 - c Click "For Use with Mentor Graphics ModelSim" to access the HDL Verifier Simulink blocks for use with ModelSim.
 - d Copy the To VCD File block from the Library Browser to the model by clicking the block and dragging it from the browser to the location in your model window previously occupied by the Signal Scope block.
 - e Double-click the To VCD File block icon. The Block Parameters dialog box appears.
 - f Type `MyVCDfile.vcd` in the **VCD file name** text box.
 - g Type 4 in the **Number of input ports** text box.



- h Click **OK**. Simulink applies the new parameters to the block.
- 5 Connect the signals **Original Data**, **Recovered Data**, **Recovered Clock**, and **Data Validity** to the block ports. The following display highlights the modified area of the model.



- 6 Save the model.
- 7 Select the following command line from the instructional text that appears in the demonstration model:

```
vsim('tclstart',manchestercmds,'socketsimulink',4442)
```

- 8 Paste the command in the MATLAB Command Window and execute the command line. This command starts ModelSim and configures it for a Simulink cosimulation session.
- 9 Open the HDL Cosimulation block parameters dialog box and select the **Connection** tab. Change the Connection method to Socket and "4442" for the TCP/IP socket port. The port you specify here must match the value specified in the call to the `vsim` command in the previous step.

- 10 Start the simulation from the Simulink model window.
- 11 When the simulation is complete, locate, open, and browse through the generated VCD file, MyVCDfile.vcd (any text editor will do).
- 12 Close the VCD file.
- 13 Change your input focus to ModelSim and end the simulation.
- 14 Change the current folder to the folder containing the VCD file and enter the following command at the ModelSim command prompt:

```
vcd2wlf MyVCDfile.vcd MyVCDfile.wlf
```

The vcd2wlf utility converts the VCD file to a WLF file that you display with the command `vsim -view`.

- 15 In ModelSim, open the wave file MyVCDfile.wlf as data set MyVCDwlf by entering the following command:


```
vsim -view MyVCDfile.wlf
```

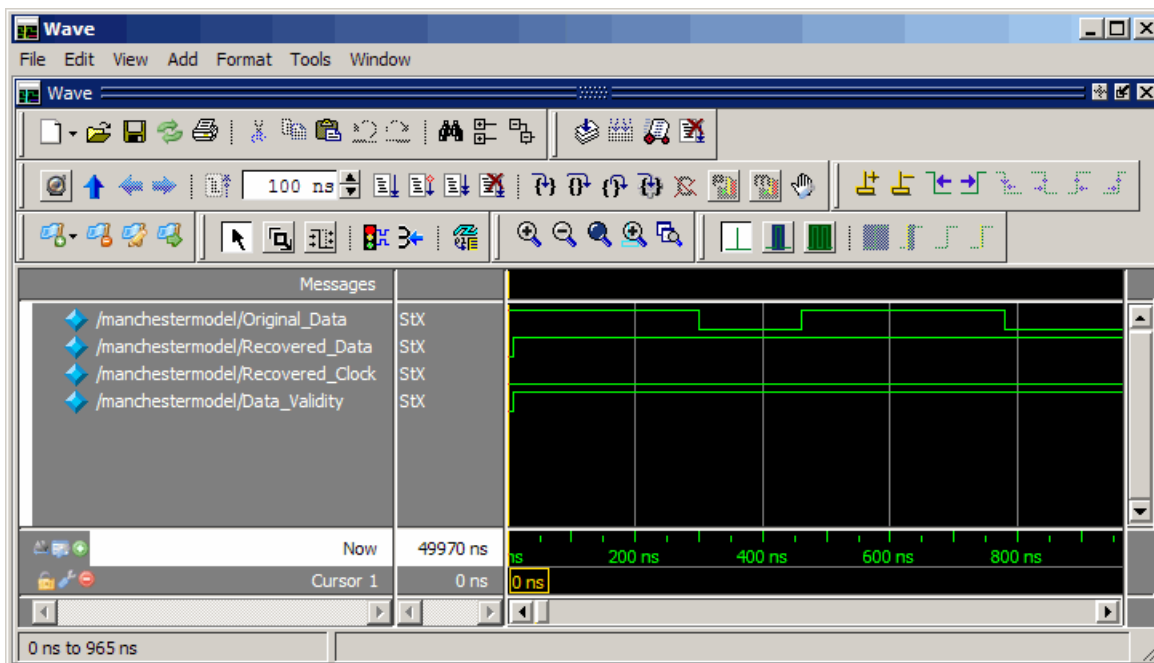
- 16 Open the MyVCDwlf data set with the following command:

```
add wave MyVCDfile:/*
```

A **wave** window appears showing the signals logged in the VCD file.

- 17

Click the Zoom Full button  to view the signal data. The **wave** window should appear as follows.



- 18 Exit the simulation. One way of exiting is to enter the following command:

```
dataset close MyVCDfile
```

ModelSim closes the data set, clears the **wave** window, and exits the simulation.

For more information on the `vcd2wlf` utility and working with data sets, see the ModelSim documentation.

HDL Code Import for Cosimulation

- “Prepare to Import HDL Code for Cosimulation” on page 9-2
- “Import HDL Code for MATLAB Function” on page 9-4
- “Import HDL Code for MATLAB System Object” on page 9-12
- “Import HDL Code for HDL Cosimulation Block” on page 9-24
- “Use HDL Parameters in Cosimulation” on page 9-35
- “Performing Cosimulation” on page 9-37
- “Verify Raised Cosine Filter Design Using MATLAB” on page 9-38
- “Verify Raised Cosine Filter Design Using Simulink” on page 9-51
- “Help Button” on page 9-66

Prepare to Import HDL Code for Cosimulation

HDL Code Import Features

The HDL Verifier **Cosimulation Wizard** lets you take existing HDL code, from any source, and use it to create a MATLAB component or test bench function, System object, or Simulink HDL Cosimulation block. You can then use one of these cosimulation interfaces for cosimulation with a supported HDL simulator. See “Supported EDA Tools and Hardware”.

After you finish running the wizard, you must complete some missing pieces in the generated cosimulation interface. For example, if you specified a MATLAB function, the generated script contains some simple port I/O instructions and empty routines, which you must fill in before you can run HDL cosimulation.

What You Need to Know

You are expected to understand the following about the HDL code you want to import:

- The name of the HDL files or compilation scripts to use in creating the block or function
- “Supported Data Types” on page 10-35 in HDL/MATLAB/Simulink
- For Simulink blocks and MATLAB System objects:
 - The name of the top module to be used for cosimulation
 - Output port types and sample times
 - Whether there are clocks and resets and which of them you want to use, and timing parameters
 - Timescale
- For MATLAB functions:
 - Whether you want to create a component function or test bench function, or both
 - How you want to trigger the callback (rising or falling edge, repeat, sensitivity)

For Simulink blocks, you must also have a destination model to receive the generated cosimulation interface block.

What the Cosimulation Wizard Needs to Know

The Cosimulation Wizard guides you through specifying this information (some information depends on which type of cosimulation interface you want it to create):

- Type of cosimulation (MATLAB, MATLAB System object, or Simulink)
- Which HDL simulator to use
- HDL files to be included and compilation instructions
- HDL module information
- Callback details
- Input and output port details
- Clock and reset information and HDL simulator start time alignment

HDL Code Import Workflows

When you are ready to begin:

- 1 Close your Vivado, ModelSim, or Xcelium simulator.
- 2 Open the **Cosimulation Wizard** from the MATLAB command prompt:

```
cosimWizard
```
- 3 Follow the workflow specific to the cosimulation interface you want to create:
 - “Import HDL Code for MATLAB Function” on page 9-4
 - “Import HDL Code for MATLAB System Object” on page 9-12
 - “Import HDL Code for HDL Cosimulation Block” on page 9-24

Cosimulation Wizard Navigation

On each selection pane there is a status window and navigational options.

- The status window displays the current options you have selected. Warnings are displayed here also.
- Click **Help** to display this HDL Code Import topic.
- Click **Cancel** to exit the Cosimulation Wizard without creating a cosimulation component.
- Click **Back** and **Next** to navigate forwards and backwards, respectively, through the application. Note that you can move forwards only after you have provided all information for the step you are on.

The last step of the Cosimulation Wizard generates the function scripts, System objects, or blocks and launches the specified HDL simulator.

- If you select a function or System object, the MATLAB Editor opens with the unfinished script or System object ready for editing.
- If you select a block, Simulink opens with the new block inside an untitled model.

Cosimulation Wizard Limitations

- When creating an HDL Cosimulation block or System object for use with Simulink, you may access only the I/O ports on the top level of the HDL design. If you want to cosimulate at multiple levels of your design, you cannot use this application to set up your HDL Cosimulation block or System object.
- You cannot create multiple HDL Cosimulation blocks, nor can you use multiple generated HDL Cosimulation blocks in the same model. This is primarily because you can only access the top level of the HDL design. There is no need for additional blocks.

Import HDL Code for MATLAB Function

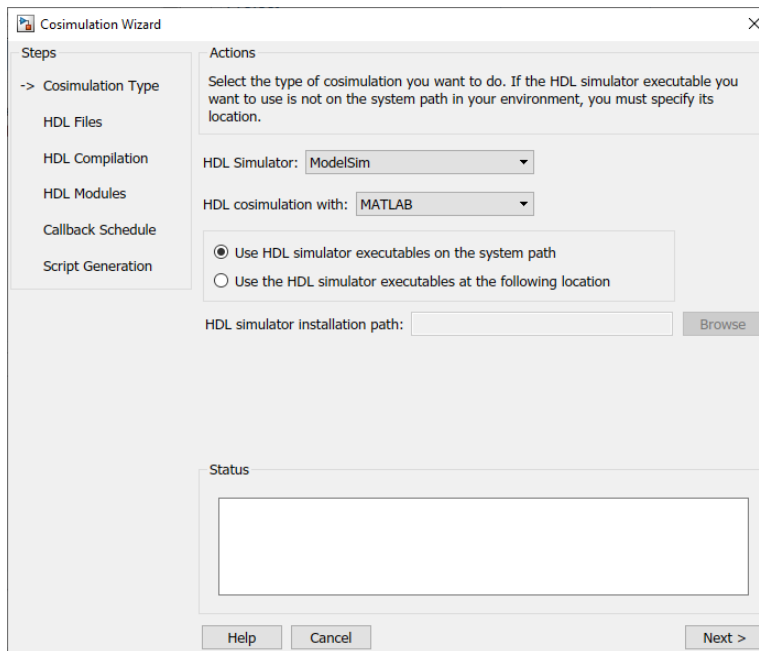
In this section...

“Cosimulation Type—MATLAB Function” on page 9-4
 “HDL Files—MATLAB Function” on page 9-5
 “HDL Compilation—MATLAB Function” on page 9-6
 “HDL Modules—MATLAB Function” on page 9-7
 “Callback Schedule—MATLAB Function” on page 9-8
 “Script Generation—MATLAB Function” on page 9-9
 “Complete the Component or Test Bench Function” on page 9-10

Cosimulation Type—MATLAB Function

If you have not yet done so, invoke the **Cosimulation Wizard**.

cosimWizard



- 1 In the **Cosimulation Type** pane, select MATLAB in the field **HDL cosimulation with** to create a MATLAB function template (test bench or component).
- 2 Select ModelSim or Xcelium for the **HDL Simulator**. This workflow is not supported for Vivado simulator.
- 3 Select **Use HDL simulator executables on the system path** if that is where the files are located. The Cosimulation Wizard assumes by default that they are on the system path.

If the HDL simulator executables are *not* on the system path, select **Use the following HDL simulator executables at the following location** and specify the folder location in the text box below.

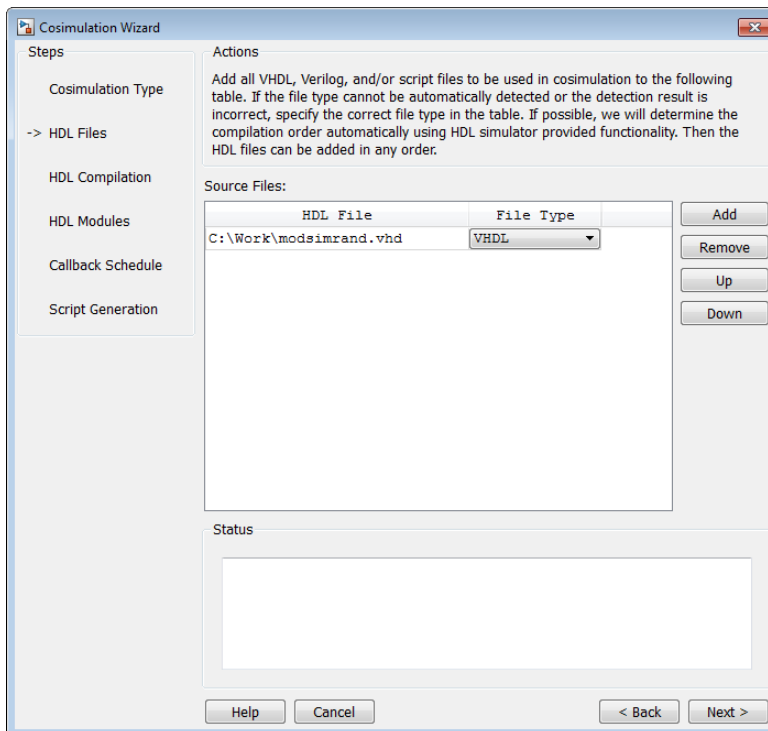
If you click **Next** and the **Cosimulation Wizard** does not find the executables, the following occurs:

- You are returned to this dialog and the **Cosimulation Wizard** displays an error in the status pane.
- The Cosimulation Wizard switches the option to **Use the following HDL simulator executables at the following location**.
- The Cosimulation Wizard makes the HDL simulation path field editable.

You must enter a valid path to the HDL simulator executables before you are allowed to continue.

4 Click **Next**.

HDL Files—MATLAB Function



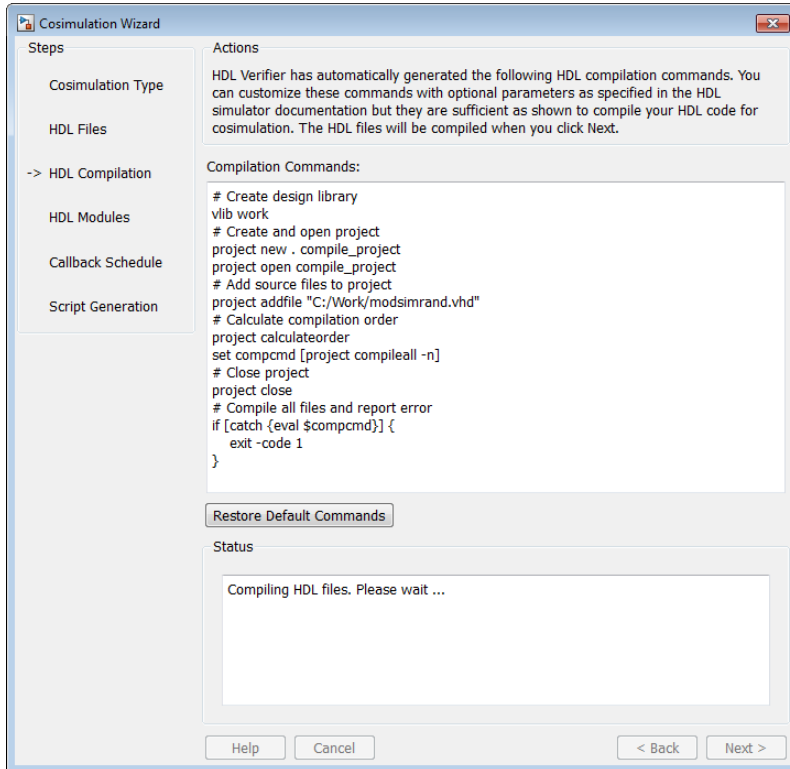
In the **HDL Files** pane, specify the files to be used in creating the function or block.

- The Cosimulation Wizard attempts to determine the file type of each file and display the type in the **File List** next to the file name. If the Cosimulation Wizard cannot determine the type or displays the wrong type, you can change the type directly in the **File Type** column.
- If possible, the Cosimulation Wizard will determine the compilation order automatically using HDL simulator provided functionality. This means you can add the files in any order.
- If you are using ModelSim, you will see compilation scripts listed as .do files (ModelSim macro file). If you are using Xcelium, you will see compilation scripts listed as system scripts.

- 1 Click **Add** to select one or more file names.
- 2 Remove files by first highlighting the file name in the **File List**, then clicking **Remove Selected File**.

- 3 Click **Next**.

HDL Compilation—MATLAB Function



In the **HDL Compilation** pane, you can review the generated HDL compilation commands. You may override and/or customize those commands, if you wish. If you included compilation scripts instead of HDL files, this pane will show you the command to run those scripts.

- 1 Enter any changes to the commands in the **Compilation Commands** box.

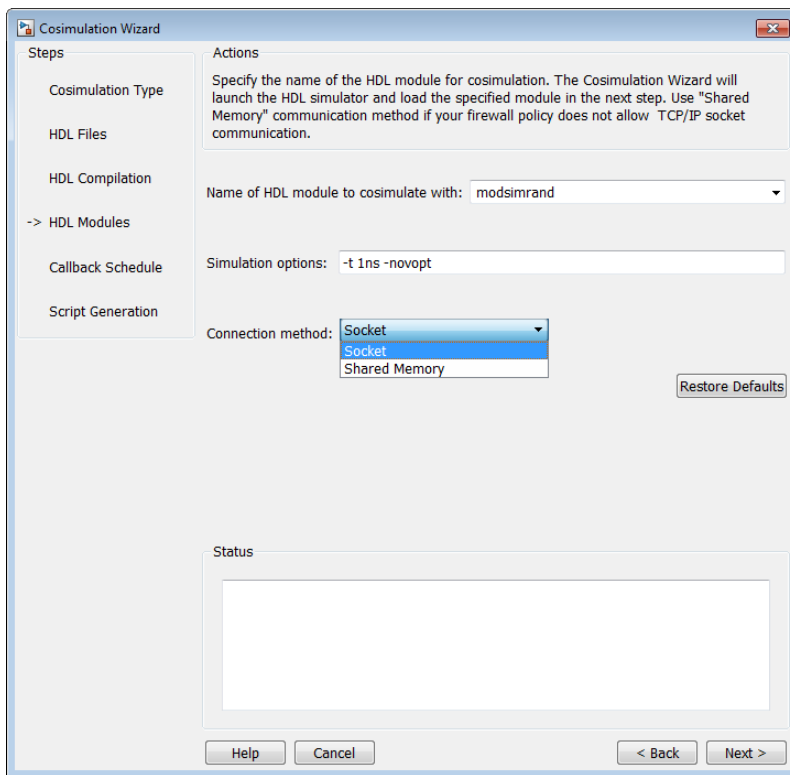
Note Do not include system shell commands; for example:

```
set file = a.vhd vcom $file
```

When control returns to the Cosimulation Wizard from executing the command, the variable no longer holds the value that was set. If you do try to include this type of command, you will see an error in the **Status** panel.

- 2 Click **Restore default commands** to go back to the generated HDL compilation commands. You are asked to confirm that you want to discard any changes.
- 3 Click **Next** to proceed.

HDL Modules—MATLAB Function



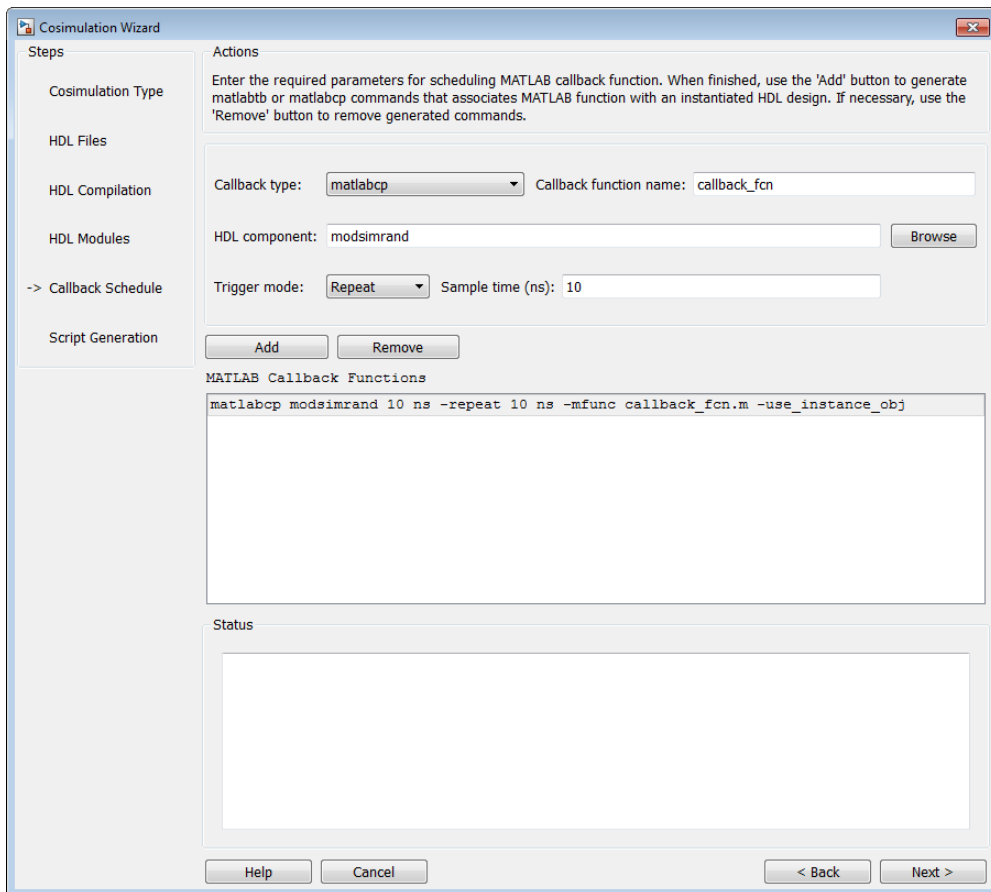
In the **HDL Module** pane, provide the name of the HDL module to be used in cosimulation.

- 1 Enter the name of the module at **Name of HDL module to cosimulate with**.
- 2 Specify additional simulation options at **Simulation options**. For example, in the previous image, the options shown are:
 - HDL simulator resolution
 - Turn off optimizations that remove signals from the simulation view

Click **Restore Defaults** to change the options back to the default.

- 3 For **Connection method**, select Shared Memory if your firewall policy does not allow TCP/IP socket communication.
- 4 Click **Next** to proceed to the next step. At this time in the process, the application performs the following actions in a command window:
 - Starts the HDL simulator.
 - Loads the HDL module in the HDL simulator.
 - Starts the HDL server, and waits to receive notice that the server has started.
 - Connects with the HDL server to get the port information.
 - Disconnects and shuts down the HDL server.

Callback Schedule—MATLAB Function



- 1 In the **Callback Schedule** pane, enter multiple component or test bench function callbacks from the HDL simulator. Enter the following information for each callback function:
 - **Callback type:** select `matlabcp` to create a component function or `matlabtb` to create a test bench function.
 - **Callback function name** (optional): Specify the name of component or test bench function, if it is not the same as the HDL component. The default assumption is that the function name is the same as the HDL component name.
 - **HDL component:** Enter component name manually or browse for it by clicking **Browse**.
 - **Trigger mode:** Specify one of the following to trigger the callback function:
 - Repeat
 - Rising Edge
 - Falling Edge
 - Sensitivity
 - **Sample time (ns) or Trigger Signal:**
 - If you selected trigger Repeat, enter the sample time in nanoseconds.
 - If you selected Rising Edge, Falling Edge, or Sensitivity, **Sample time (ns)** changes to **Trigger Signal**. Enter the signal name to be used to trigger the callback.

- You can browse the existing signals in the HDL component you specified by clicking **Browse**.
- Click **Add** to add the command to the MATLAB Callback Functions list.

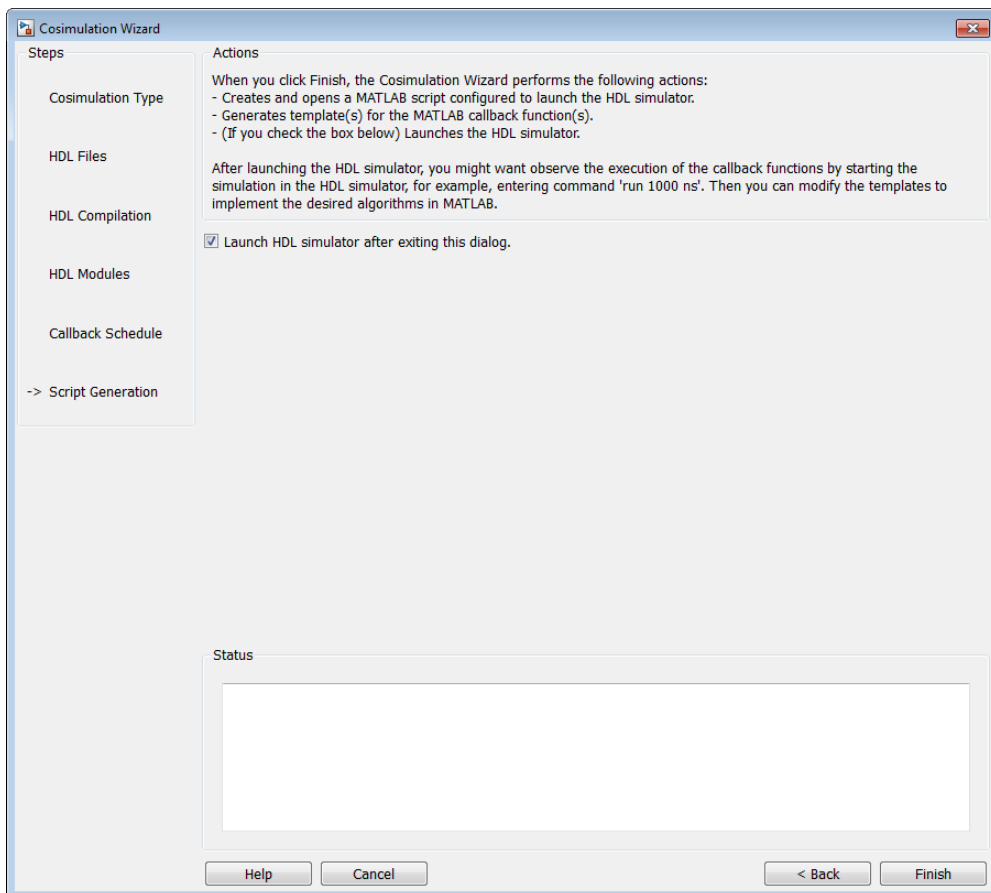
If you have more callback functions you want to schedule, repeat the above steps. If you want to remove any callback functions, highlight the line you want to remove and click **Remove**.

Note If you attempt to add a callback function for the same HDL module as an existing callback function in the MATLAB Callback Functions list, the new callback function will overwrite the existing one (this is true even if you change the callback type). You will see a warning in the **Status** window:

Warning: This HDL component already has a scheduled callback function, which is replaced by this new one.

- Click **Next**.

Script Generation—MATLAB Function



- Click **Back** to review or change your settings.
- Click **Finish** to generate scripts.

Generated Files—MATLAB Function

The Cosimulation Wizard creates the following files and opens each one in a separate MATLAB Editor windows.

- `launchHDLsimulator` — script for launching the HDL simulator for cosimulation with MATLAB.
- `compileHDLDesign` — compilation script you can reuse for subsequent compilation of this particular component.
- Function files (`*.m`) — component and test bench customized function templates, one for each component specified in the Cosimulation Wizard.

Complete the Component or Test Bench Function

The template that the wizard generates contains some simple port I/O instructions and empty routines where you add your own code, as shown in the example below. For a full example of creating and using a MATLAB function, see “Verify Raised Cosine Filter Design Using MATLAB” on page 9-38.

```
function osc_top_u_osc_filter1x(obj)
% Automatically generated MATLAB(R) callback function.

% Copyright 2010 The MathWorks, Inc.
% $Revision $

% Initialize state of callback function.
if (strcmp(obj.simstatus,'Init'))
    disp('Initializing states ...');

    % Store port information in userdata
    % The name strings of ports that sends data from HDL simulator to
    % MATLAB callback function
    obj.userdata.FromHdlPortNames = fields(obj.portinfo.out);
    obj.userdata.FromHdlPortNum   = length(fields(obj.portinfo.out));

    % The name strings of ports that sends data from MATLAB callback
    % function to HDL simulator
    obj.userdata.ToHdlPortNames   = fields(obj.portinfo.in);
    obj.userdata.ToHdlPortNum     = length(fields(obj.portinfo.in));

    % Initialize state
    obj.userdata.State = 0;
end

% Obj.tnow is the current HDL simulation time specified in seconds
disp(['Callback function is executed at time ' num2str(obj.tnow)]);

if(obj.userdata.FromHdlPortNum > 0)
    % The name of the first input port
    portName = obj.userdata.FromHdlPortNames{1};
    disp(['Reading input port ' portName]);
    % Convert the multi-valued logic value of the first port to decimal
    portValueDec = mvl2dec( ...
        obj.portvalues.(portName), ... % Multi-valued logic of the first port
        obj.portinfo.out.(portName).size); %#ok<NASGU> % Bit width
    % Then perform any necessary operations on this value passed by HDL simulator.
    % ...
    % Optionally, you can translate the port value into fixed point object,
    % e.g.
    % myfiobj = fi(portValueDec,1, 16, 4);
end

% Update your state(s). In the following example, we use this internal
% state to implement a one-bit counter
obj.userdata.State = ~obj.userdata.State;

if(obj.userdata.ToHdlPortNum > 0)
    % The name of the first output port in HDL
    portName = obj.userdata.ToHdlPortNames{1};
    disp(['Writing output port ' portName]);

    % Assign the first port value to internal state obj.userdata.State.
```

```
% Before assignment, convert decimal value to multi-valued logic.  
% You can change obj.userdata.State to another other valid decimal values.  
obj.portvalues.(portName) = dec2mvl(...  
    obj.userdata.State, ...  
    obj.portinfo.in.(portName).size);  
  
% Operate on other out ports, if there are any.  
% ...  
end
```

See Also

Cosimulation Wizard

Related Examples

- “Supported Data Types” on page 10-35

Import HDL Code for MATLAB System Object

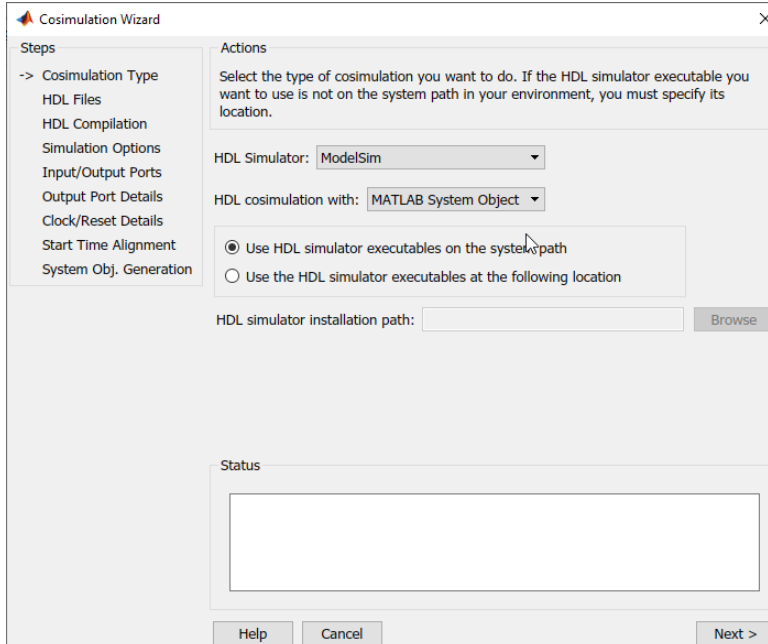
In this section...

“Cosimulation Type—MATLAB System Object” on page 9-12
 “HDL Files—MATLAB System Object” on page 9-13
 “HDL Compilation—MATLAB System Object” on page 9-14
 “Simulation Options—MATLAB System Object” on page 9-15
 “Input/Output Ports—MATLAB System Object” on page 9-17
 “Output Port Details—MATLAB System Object” on page 9-18
 “Clock/Reset Details—MATLAB System Object” on page 9-19
 “Start Time Alignment—MATLAB System Object” on page 9-20
 “System Object Generation” on page 9-21
 “Write System Object Test Bench” on page 9-21
 “Run Cosimulation and Verify HDL Design” on page 9-22

Cosimulation Type—MATLAB System Object

If you have not yet done so, invoke the **Cosimulation Wizard**.

cosimWizard



- 1 In the **Cosimulation Type** pane, select ModelSim, Xcelium, or Vivado Simulator for the **HDL Simulator**.
- 2 Select MATLAB System object in the field **HDL cosimulation with**.
- 3 Select **Use HDL simulator executables on the system path** if that is where the files are located. The Cosimulation Wizard assumes by default that they are on the system path.

If the HDL simulator executables are *not* on the system path, select **Use the following HDL simulator executables at the following location** and specify the folder location in the text box below.

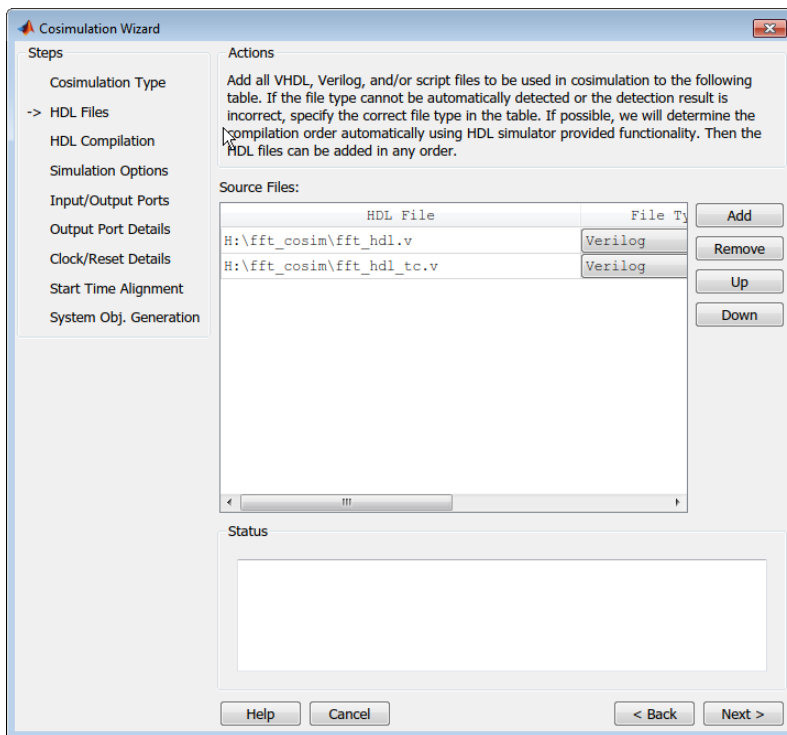
If you click **Next** and the Cosimulation Wizard does not find the executables, the following occurs:

- You are returned to this dialog and the Cosimulation Wizard displays an error in the status pane.
- The Cosimulation Wizard switches the option to **Use the following HDL simulator executables at the following location**.
- The Cosimulation Wizard makes the HDL simulation path field editable.

You must enter a valid path to the HDL simulator executables before you are allowed to continue.

4 Click **Next**.

HDL Files—MATLAB System Object

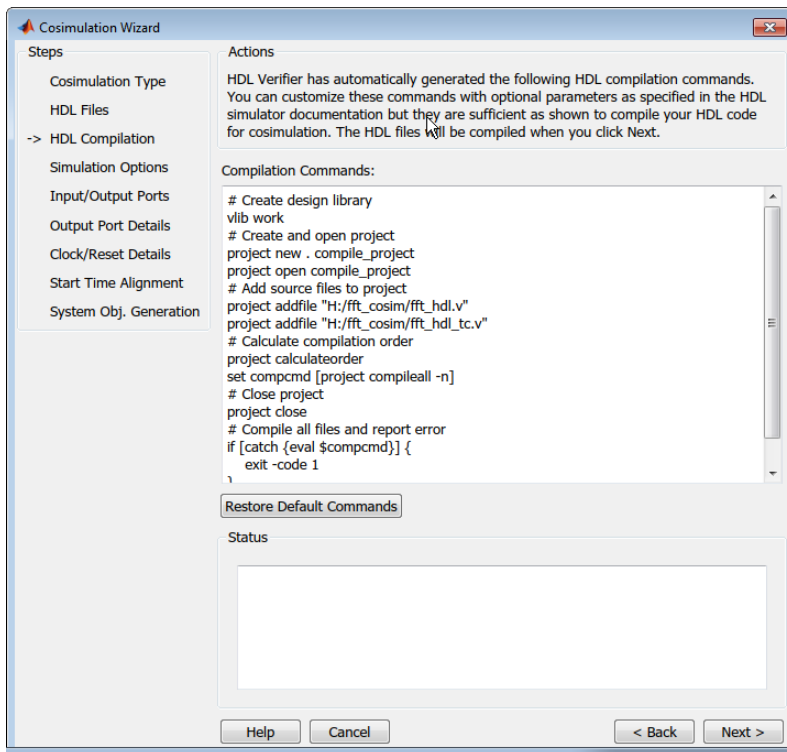


In the **HDL Files** pane, specify the files to be used in creating the function or block.

- The Cosimulation Wizard attempts to determine the file type of each file and display the type in the **File List** next to the file name. If the Cosimulation Wizard cannot determine the type or displays the wrong type, you can change the type directly in the **File Type** column.
- If possible, the Cosimulation Wizard will determine the compilation order automatically using HDL simulator provided functionality. If your simulator does not include this functionality, add the files in the order they should be compiled.

- If you are using ModelSim, you will see compilation scripts listed as .do files (ModelSim macro file). If you are using Xcelium or Vivado, you will see compilation scripts listed as system scripts.
- 1 Click **Add** to select one or more file names.
 - 2 Remove files by first highlighting the file name in the **File List**, then clicking **Remove Selected File**.
 - 3 Click **Next**.

HDL Compilation—MATLAB System Object



In the **HDL Compilation** pane, you can review the generated HDL compilation commands. You may override and/or customize those commands, if you wish. If you included compilation scripts instead of HDL files, this pane will show you the command to run those scripts.

- 1 Enter any changes to the commands in the **Compilation Commands** box.

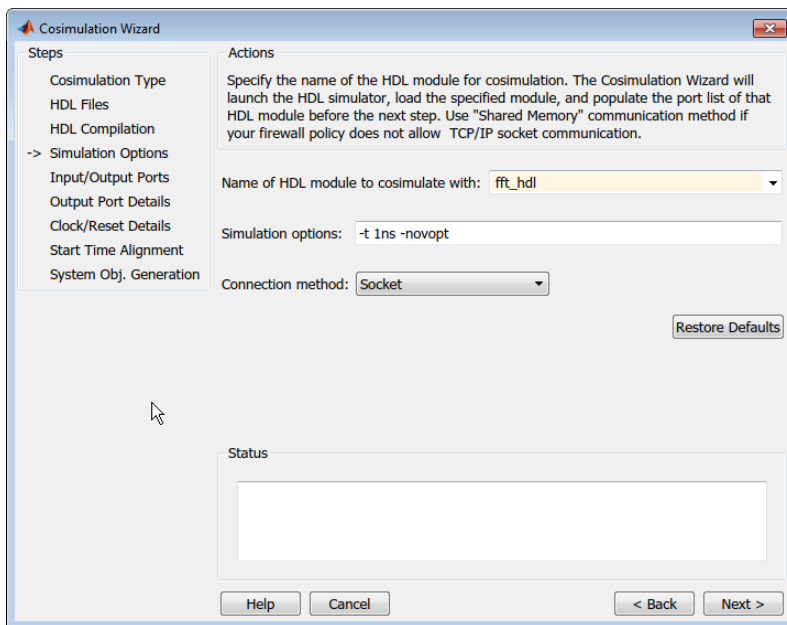
Note Do not include system shell commands; for example:

```
set file = a.vhd vcom $file
```

When control returns to the Cosimulation Wizard from executing the command, the variable no longer holds the value that was set. If you do try to include this type of command, you will see an error in the **Status** panel.

- 2 Click **Restore default commands** to go back to the generated HDL compilation commands. You are asked to confirm that you want to discard any changes.
- 3 Click **Next** to proceed.

Simulation Options—MATLAB System Object



Modelsim or Xcelium Users: In the **Simulation Options** pane, provide the name of the HDL module to be used in cosimulation.

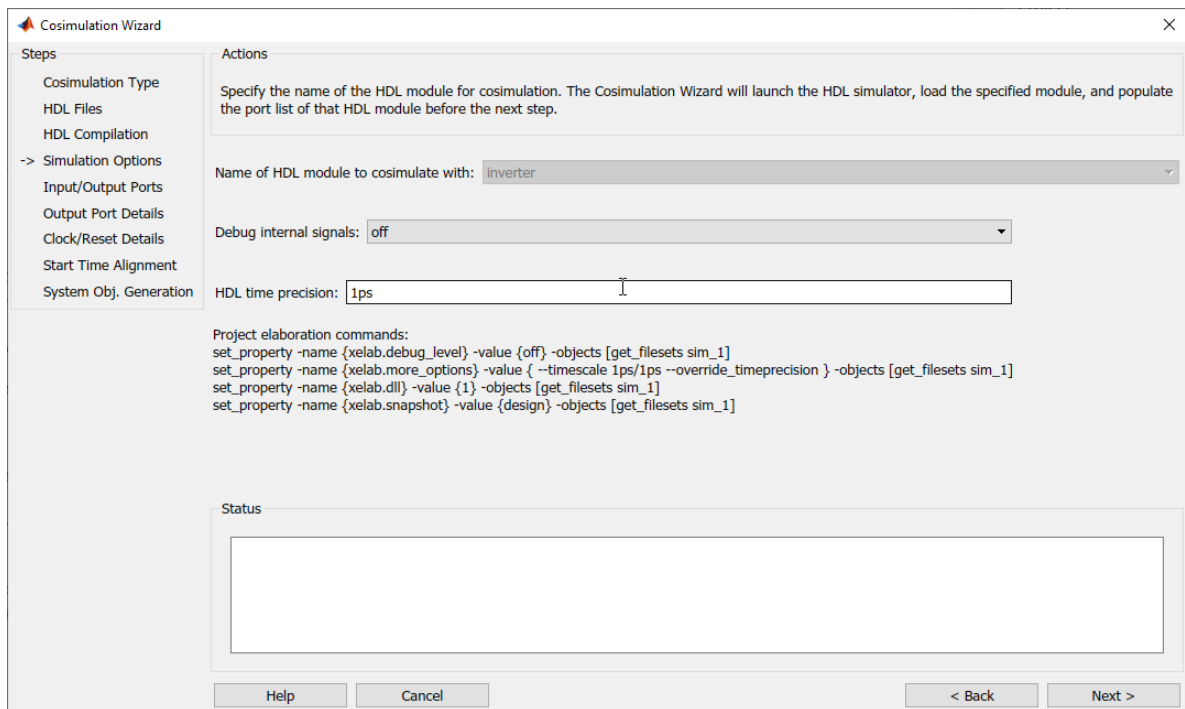
- 1 Enter the name of the module at **Name of HDL module to cosimulate with**.
- 2 Specify additional simulation options at **Simulation options**. For example, in the previous image, the options shown are:
 - HDL simulator resolution
 - Turn off optimizations that remove signals from the simulation view

Click **Restore Defaults** to change the options back to the default.

- 3 For **Connection method**, select Shared Memory if your firewall policy does not allow TCP/IP socket communication.
- 4 Click **Next** to proceed to the next step.

For Xcelium or ModelSim, the application performs the following actions in a command window:

- Starts the HDL simulator.
- Loads the HDL module in the HDL simulator.
- Starts the HDL server, and waits to receive notice that the server has started.
- Connects with the HDL server to get the port information.
- Disconnects and shuts down the HDL server.



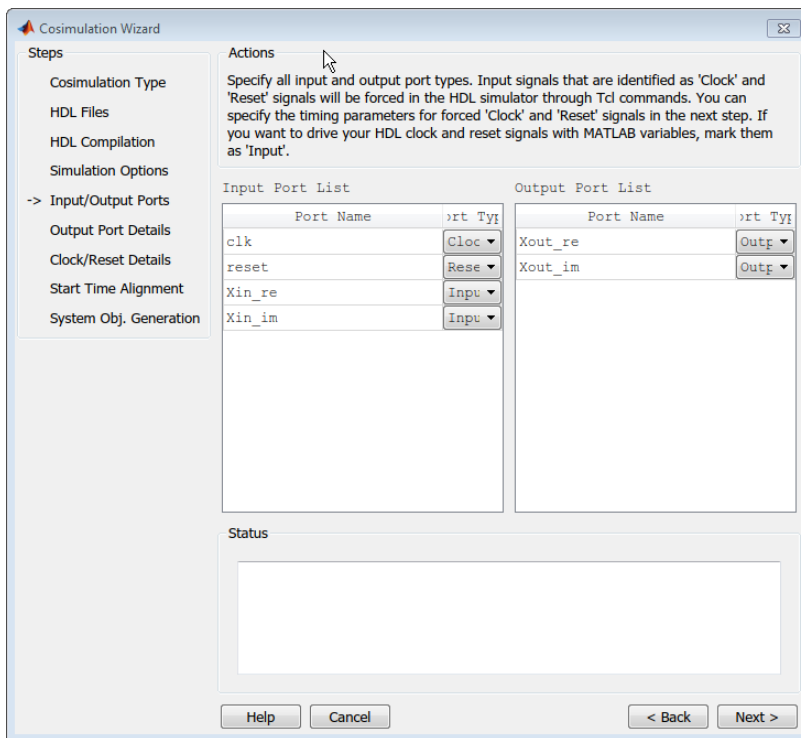
Vivado Simulator Users: When creating a system object for Vivado cosimulation, the wizard displays the name of the HDL top module.

To generate a waveform file, set **Debug internal signals** to wave.

In the **HDL time precision** parameter you can also change the simulation time precision.

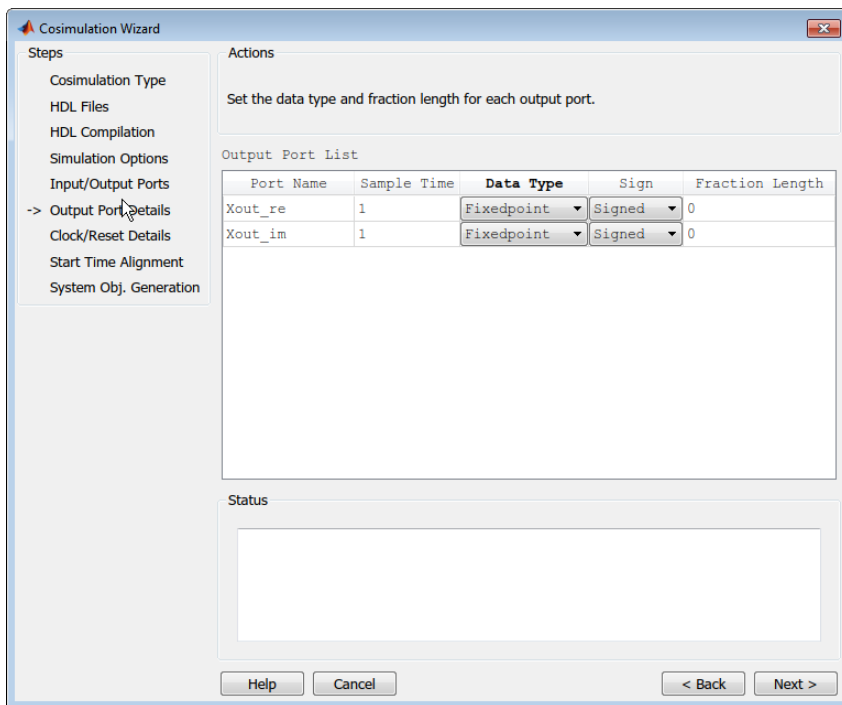
Click **Next** to create a shared library (dll file).

Input/Output Ports—MATLAB System Object



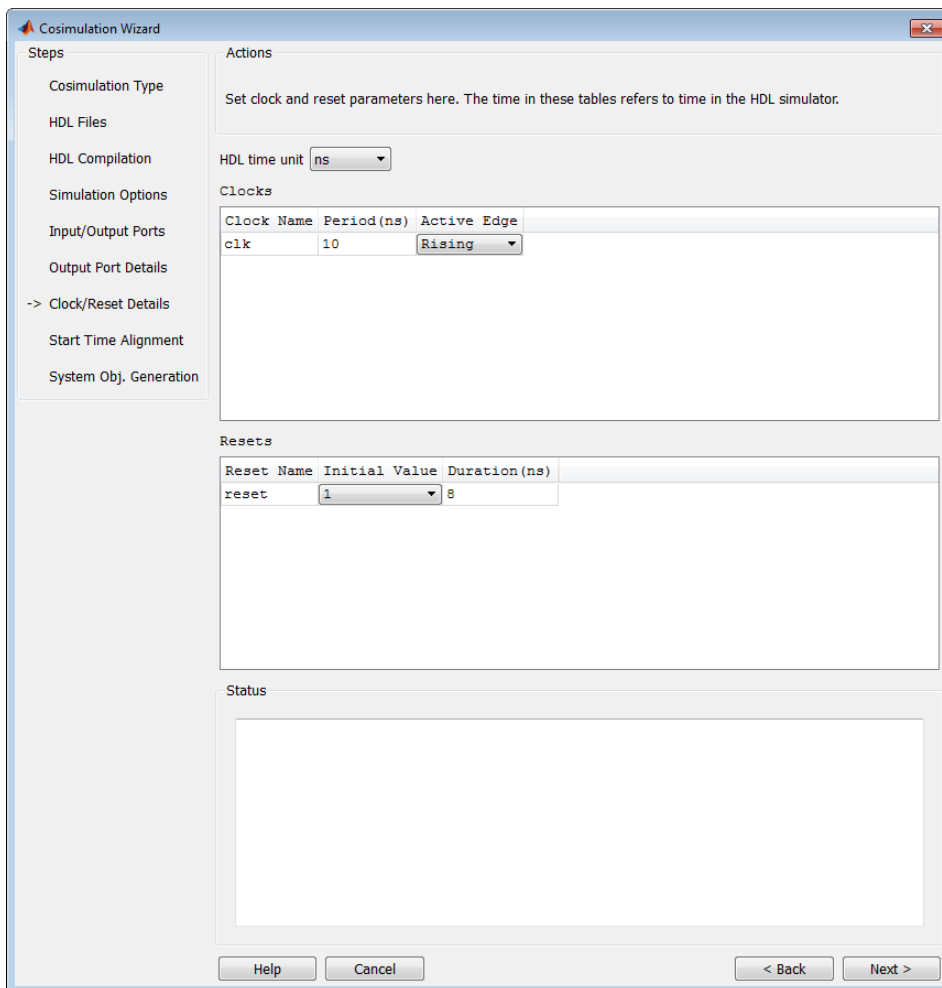
- 1 In the **Input/Output Ports** pane, specify the type of each input and output port (Input, Clock, Reset, or Unused).
 - The Cosimulation Wizard attempts to determine the port types for you, but you may override any setting. For supported cosimulation data types, see “Supported Data Types” on page 10-35.
 - MATLAB forces clock and reset signals in the HDL simulator through Tcl commands. You can specify clock and reset signal timing in a later step (see “Clock/Reset Details—MATLAB System Object” on page 9-19).
- 2 Click **Next**.

Output Port Details—MATLAB System Object



- 1 In the **Output Port Details** pane, set the sample time and data type for all output ports.
 - Sample time default is 1, the data type default is **Inherit** and **Signed**. These defaults are consistent with the way the HDL Cosimulation block mask (**Ports** tab) sets default settings for output ports (Simulink workflow).
 - If you select **Set all sample times and data types to 'Inherit'**, the ports inherit the times via back propagation (sample times are set to -1). However, back propagation may fail in some circumstances; see “Backpropagation in Sample Times” (Simulink).
- 2 Click **Next**.

Clock/Reset Details—MATLAB System Object

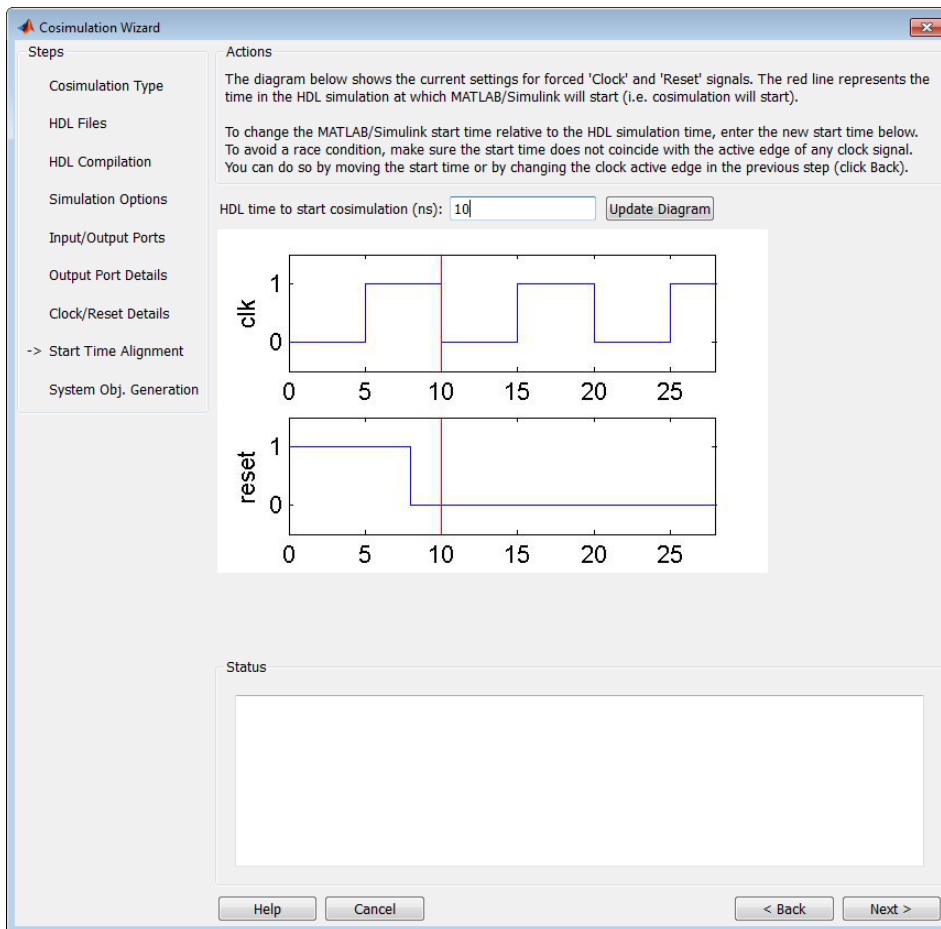


- 1 In the **Clock/Reset Details** pane, set the clock and reset parameters.
 - The time period specified here refers to time in the HDL simulator.
 - The clock default settings are a rising active edge and a period of 10 ns.
 - The reset default settings are an initial value of 0 and a duration of 15 ns.

The next screen provides a visual display of the simulation start time where you can review how the clocks and resets line up.

- 2 Click **Next**.

Start Time Alignment—MATLAB System Object



- 1 In the **Start Time Alignment** pane, review the current settings for clocks and resets. The purpose for this dialog is twofold:
 - To make sure the rising or falling edge is set as expected (from the previous step)
 - Examine the start time. If it coincides with the active edge of the clock, you need to adjust the HDL simulator start time.
 - Examine the reset signal. If it is synchronous with the clock active edge, you may have a possible race condition.

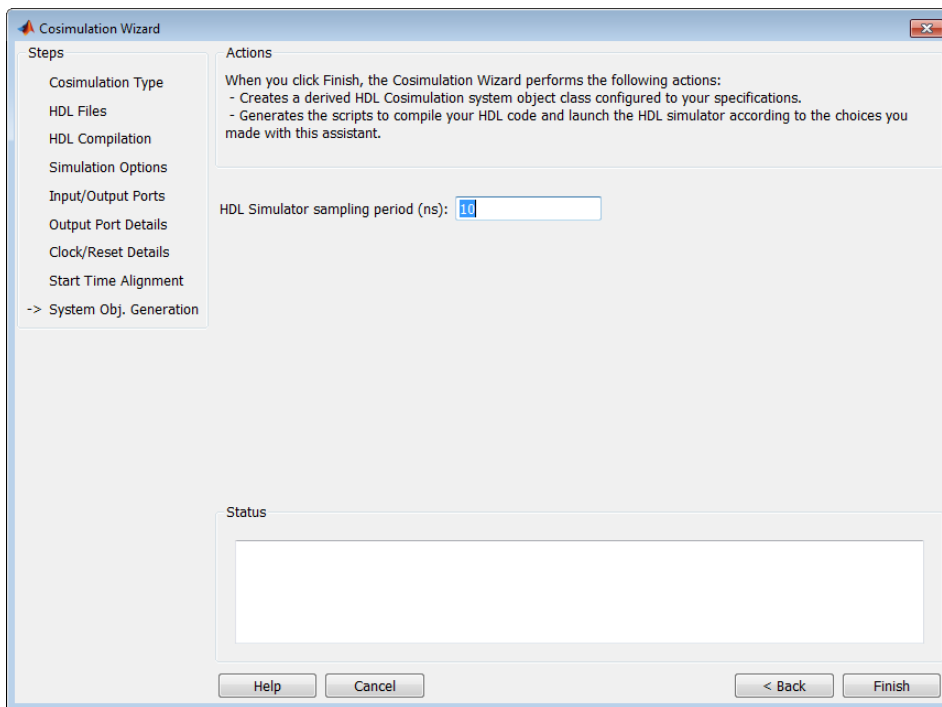
To avoid a race condition, make sure the start time does not coincide with the active edge of any clocks. You can do this by moving the start time or by changing clock active edges in the previous step.

- To make sure the start time is where you want it.

The HDL simulator start time is calculated from the clock and reset values on the previous pane. If you want, you can change the HDL simulator start time by entering a new value where you see **HDL time to start cosimulation (ns)**. Click **Update plot** to see your change applied.

- 2 Click **Next**.

System Object Generation



- 1 You can modify the HDL simulator sampling period before the wizard generates the System object. Enter the new value in the box labeled **HDL Simulator sampling period (ns)**.

The sampling period determines the elapsed time in the HDL Simulator separating each call to step in MATLAB. Most of the time the sampling period is equal to the clock period.

- 2 If your inputs and outputs are frame based (instead of sample based), select **Frame based processing**.
- 3 Click **Finish**.

After you click **Finish**, the wizard generates the following HDL files in the current directory:

- `compile_hdl_design_design_name.m` — Script for recompiling the HDL design
- `launch_hdl_simulator_design_name.m` — Script for relaunching the MATLAB System object server and starting the HDL simulator
- `hdlcosim_design_name.m` — Script for creating the HDLCosimulation System object

Write System Object Test Bench

Write the test bench for use with the newly generated HDL cosimulation System object. The test bench you write might look similar to the example shown next.

```

3 % Sinus generator creation (F=100Hz, Sampling=1000Hz, complex fix point output)
4 SinGenerator = dsp.SineWave('Frequency ', 100, ...
5                             'Amplitude', 1, ...
6                             'Method', 'Table lookup', ...
7                             'SampleRate', 1000, ...
8                             'OutputDataType', 'Custom', ...
9                             'CustomOutputDataType', numericitytype([], 10, 9), ..
10                            'ComplexOutput',true);
11
12 % HdlCosimulation System Object creation
13 fft_hdl = hdlcosim_fft_hdl;
14
15 % Simulate for 1000 samples
16 for ii=1:1000
17     % Read 1 sample from the sinus generator
18     ComplexSinus = step(SinGenerator);
19
20     % Send/receive 1 sample to/from the HDL FFT
21     [RealFft, ImagFft] = step(fft_hdl,real(ComplexSinus),imag(ComplexSinus));
22
23     % Store the FFT sample in a vector
24     ComplexFft(ii) = RealFft + ImagFft*1i;
25 end
26
27 % Discard the first 12 samples (initialization of the HDL FFT)
28 ComplexFft(1:12)=[];
29
30 % Display the FFT
31 plot(ComplexFft,'ro');
32 title('Fourier Coefficients in the Complex Plane');
33 xlabel('Real Axis');
34 ylabel('Imaginary Axis');
35
36 end

```

For the files used in this example, see “Cosimulation Wizard for MATLAB System Object” on page 32-37.

Run Cosimulation and Verify HDL Design

- 1 Launch the HDL simulator by executing the launch script created by the wizard (`launch_hdl_simulator_design_name.m`)
- 2 When the HDL simulator is ready, return to MATLAB and start the simulation by executing the test bench.
- 3 Verify the results.

See Also

Cosimulation Wizard

Related Examples

- “Supported Data Types” on page 10-35

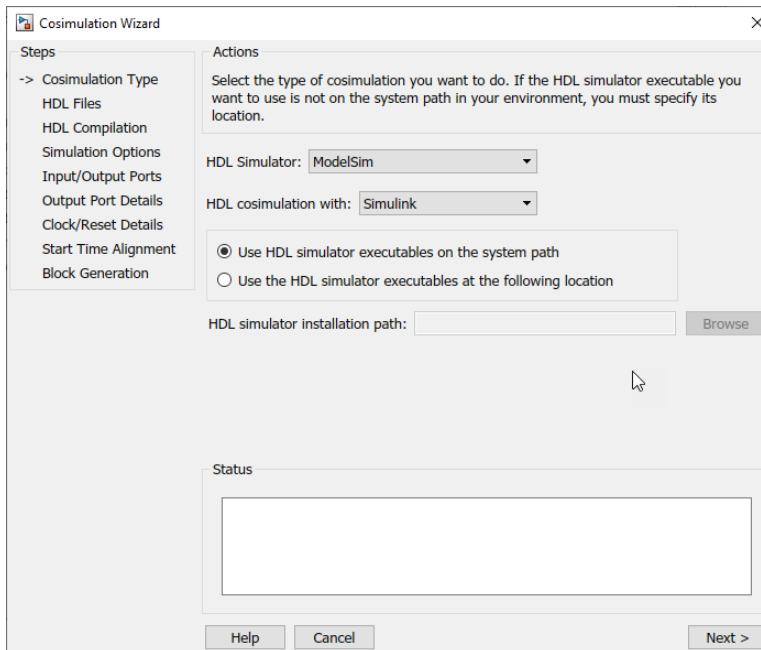
Import HDL Code for HDL Cosimulation Block

In this section...

“Cosimulation Type—Simulink Block” on page 9-24
 “HDL Files—Simulink Block” on page 9-25
 “HDL Compilation—Simulink Block” on page 9-26
 “Simulation Options—Simulink Block” on page 9-26
 “Input/Output Ports—Simulink Block” on page 9-29
 “Output Port Details—Simulink Block” on page 9-30
 “Clock/Reset Details—Simulink Block” on page 9-31
 “Start Time Alignment—Simulink Block” on page 9-32
 “Generate Block” on page 9-33
 “Complete Simulink Model” on page 9-33

Cosimulation Type—Simulink Block

Open your model, and on the **Apps** tab, click **HDL Verifier**. Then, in the **Mode** section select **HDL Cosimulation**, and click **Import HDL Files** to open the **Cosimulation Wizard**.



- 1 Select ModelSim, Xcelium, or Vivado Simulator for the **HDL Simulator**.
- 2 Select **Use HDL simulator executables on the system path** if that is where the files are located. The Cosimulation Wizard assumes by default that they are on the system path.

If the HDL simulator executables are *not* on the system path, select **Use the following HDL simulator executables at the following location** and specify the folder location in the text box below.

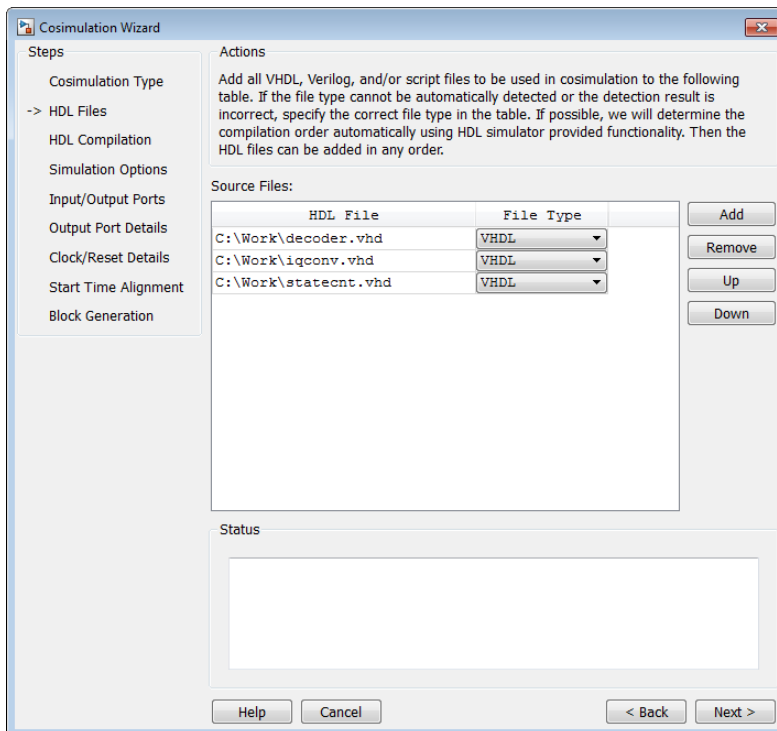
If you click **Next** and the Cosimulation Wizard does not find the executables, the following occurs:

- You are returned to this dialog and the Cosimulation Wizard displays an error in the status pane.
- The Cosimulation Wizard switches the option to **Use the following HDL simulator executables at the following location**.
- The Cosimulation Wizard makes the HDL simulation path field editable.

You must enter a valid path to the HDL simulator executables before you are allowed to continue.

3 Click **Next**.

HDL Files—Simulink Block



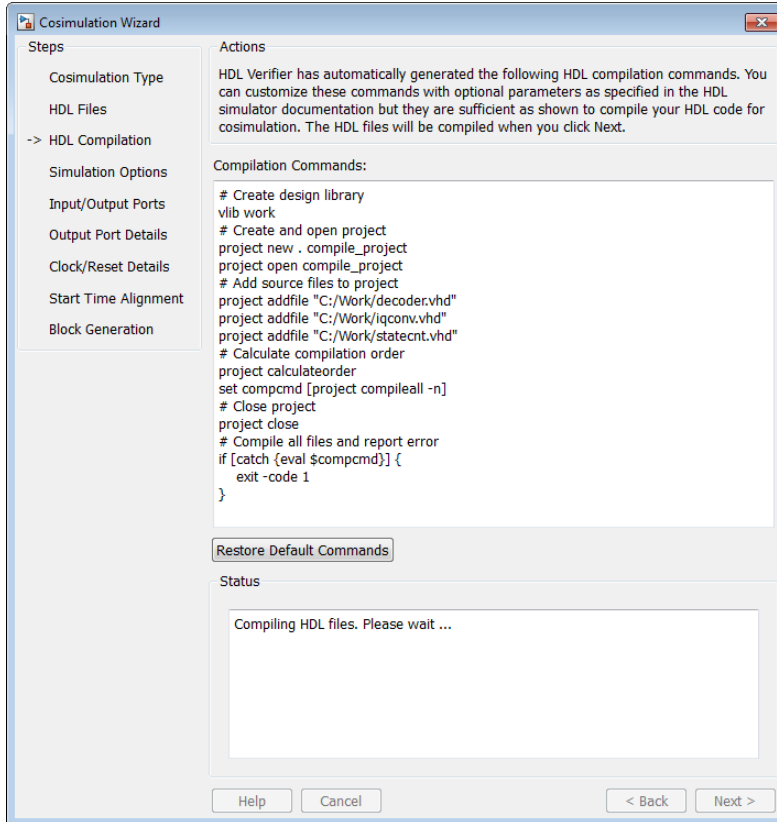
In the **HDL Files** pane, specify the files to be used in creating the function or block.

- The Cosimulation Wizard attempts to determine the file type of each file and display the type in the **File List** next to the file name. If the Cosimulation Wizard cannot determine the type or displays the wrong type, you can change the type directly in the **File Type** column.
- If possible, the Cosimulation Wizard will determine the compilation order automatically using HDL simulator provided functionality. This means you can add the files in any order.
- If you are using ModelSim, you will see compilation scripts listed as .do files (ModelSim macro file). If you are using Xcelium, you will see compilation scripts listed as system scripts.

- 1 Click **Add** to select one or more file names.
- 2 Remove files by first highlighting the file name in the **File List**, then clicking **Remove Selected File**.

- 3 Click **Next**.

HDL Compilation—Simulink Block



In the **HDL Compilation** pane, you can review the generated HDL compilation commands. You may override and/or customize those commands, if you wish. If you included compilation scripts instead of HDL files, this pane will show you the command to run those scripts.

- 1 Enter any changes to the commands in the **Compilation Commands** box.

Note Do not include system shell commands; for example:

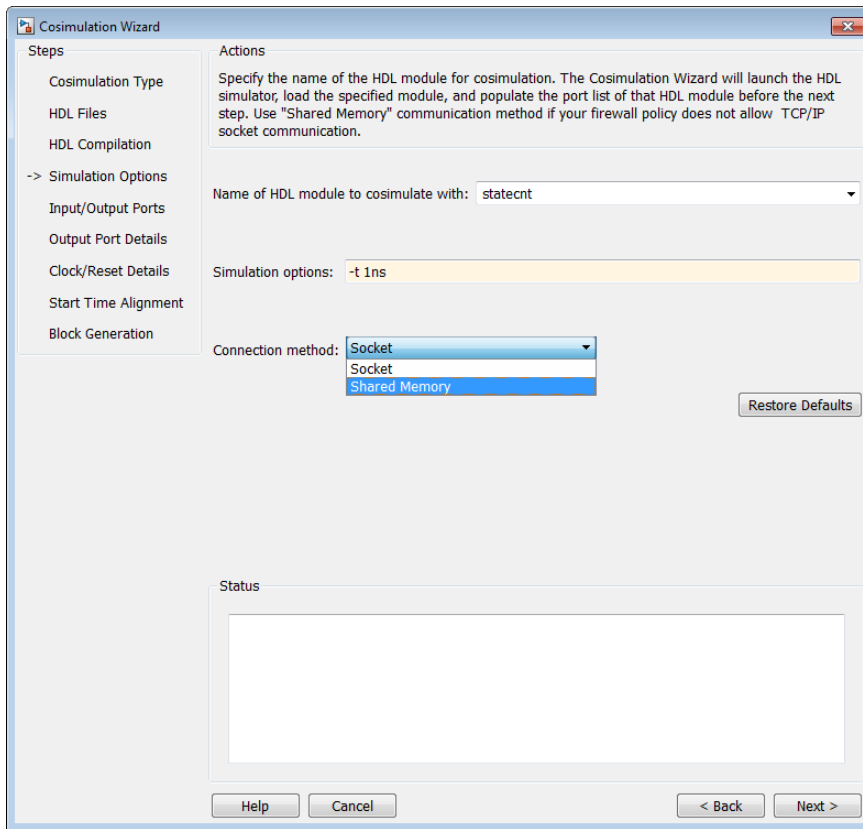
```
set file = a.vhd vcom $file
```

When control returns to the Cosimulation Wizard from executing the command, the variable no longer holds the value that was set. If you do try to include this type of command, you will see an error in the **Status** panel.

- 2 Click **Restore default commands** to go back to the generated HDL compilation commands. You are asked to confirm that you want to discard any changes.
- 3 Click **Next** to proceed.

Simulation Options—Simulink Block

Modelsim or Xcelium Users:



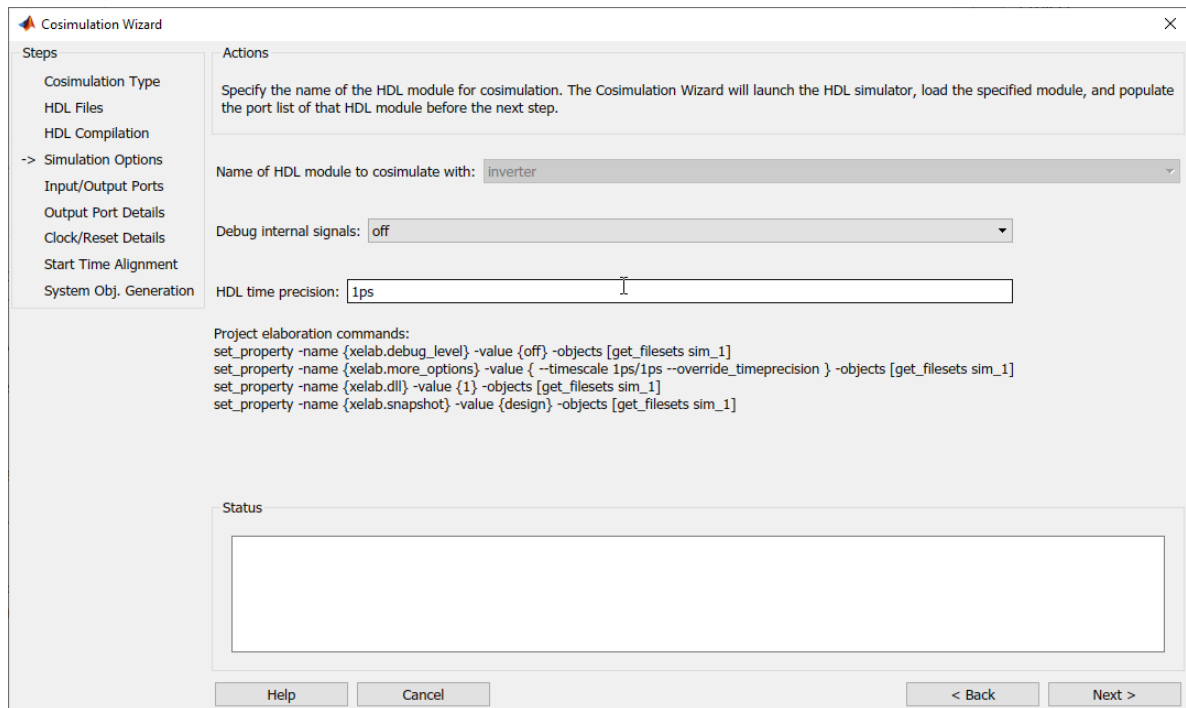
In the **Simulation Options** pane, provide the name of the HDL module to be used in cosimulation.

- 1 Enter the name of the module at **Name of HDL module to cosimulate with**.
- 2 Specify additional simulation options at **Simulation options**. For example, in the previous image, the options shown are:
 - HDL simulator resolution
 - Turn off optimizations that remove signals from the simulation view

Click **Restore Defaults** to change the options back to the default.

- 3 For **Connection method**, select Shared Memory if your firewall policy does not allow TCP/IP socket communication.
- 4 Click **Next** to proceed to the next step. At this time in the process, the application performs the following actions in a command window:
 - Starts the HDL simulator.
 - Loads the HDL module in the HDL simulator.
 - Starts the HDL server, and waits to receive notice that the server has started.
 - Connects with the HDL server to get the port information.
 - Disconnects and shuts down the HDL server.

Clicking **Next** also generates a parameter configuration file. For more information, see “Use HDL Parameters in Cosimulation” on page 9-35.

Vivado Simulator Users:

When creating a system object for Vivado cosimulation, the wizard displays the name of the HDL top module.

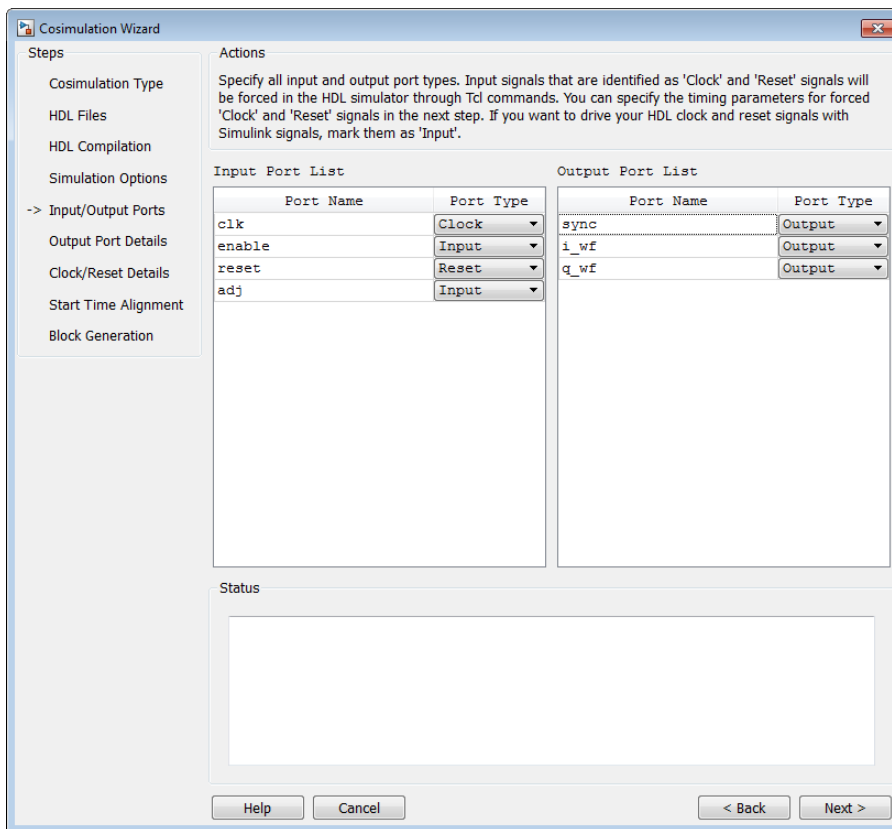
To generate a waveform file, set **Debug internal signals** to wave.

In the **HDL time precision** parameter you can also change the simulation time precision.

Click **Next** to create a shared library (dll file).

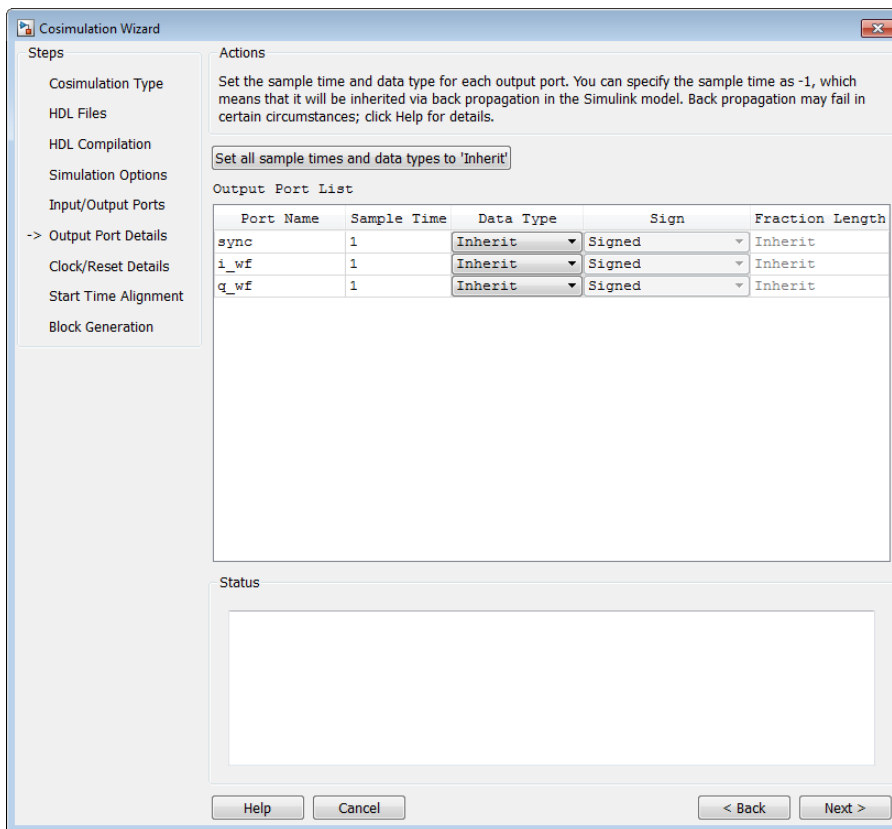
For Vivado cosimulation, this step creates a shared library.

Input/Output Ports—Simulink Block



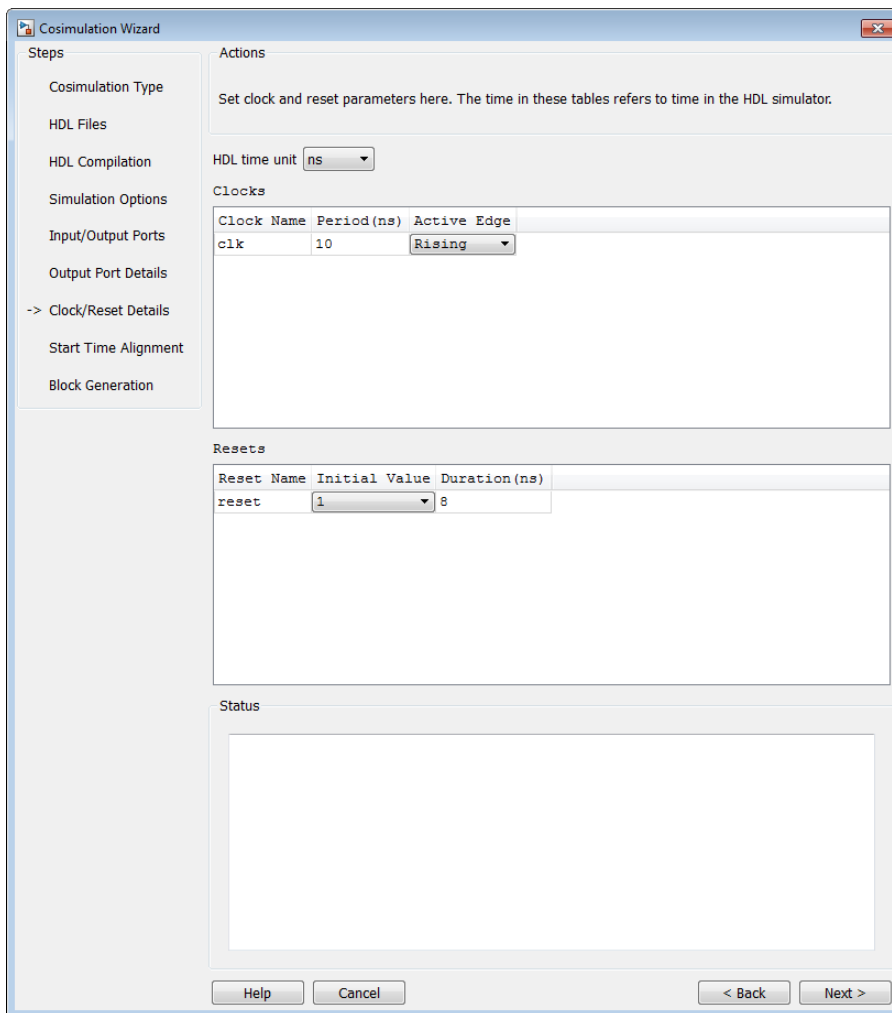
- 1 In the **Simulink Ports** pane, specify the type of each input and output port.
 - The Cosimulation Wizard attempts to determine the port types for you, but you may override any setting. For supported data types, see “Supported Data Types” on page 10-35.
 - For input ports, select Input, Clock, Reset, or Unused.
 - For output ports, select Output or Unused.
 - Simulink forces clock and reset signals in the HDL simulator through Tcl commands. You can specify clock and reset signal timing in a later step (see “Clock/Reset Details—Simulink Block” on page 9-31).
 - To drive your HDL clock and reset signals with Simulink signals, mark them as Input.
- 2 Click **Next** to proceed to “Output Port Details—Simulink Block” on page 9-30.

Output Port Details—Simulink Block



- 1 In the **Output Port Details** pane, set the sample time and data type for all output ports.
 - Sample time default is 1, the data type default is **Inherit** and **Signed**. These defaults are consistent with the way the HDL Cosimulation block mask (**Ports** tab) sets default settings for output ports.
 - If you select **Set all sample times and data types to 'Inherit'**, the ports inherit the times via back propagation (sample times are set to -1). However, back propagation may fail in some circumstances; see “Backpropagation in Sample Times” (Simulink).
- 2 Click **Next**.

Clock/Reset Details—Simulink Block

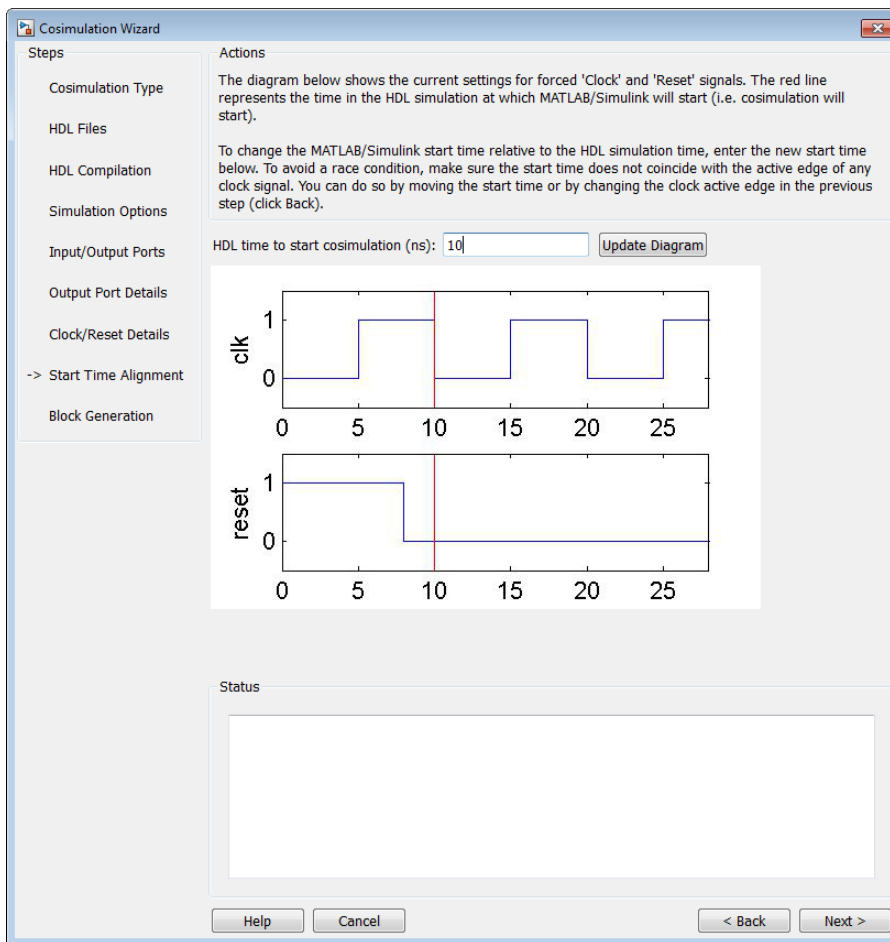


- 1 In the **Clock/Reset Details** pane, set the clock and reset parameters.
 - The time period specified here refers to time in the HDL simulator.
 - The clock default settings are a rising active edge and a period of 10 ns.
 - The reset default settings are an initial value of 0 and a duration of 15 ns.

The next screen provides a visual display of the simulation start time where you can review how the clocks and resets line up.

- 2 Click **Next**.

Start Time Alignment—Simulink Block



- 1 In the **Start Time Alignment** pane, review the current settings for clocks and resets. The purpose for this dialog is twofold:
 - To make sure the rising or falling edge is set as expected (from the previous step)
 - Examine the start time. If it coincides with the active edge of the clock, you need to adjust the HDL simulator start time.
 - Examine the reset signal. If it is synchronous with the clock active edge, you may have a possible race condition.

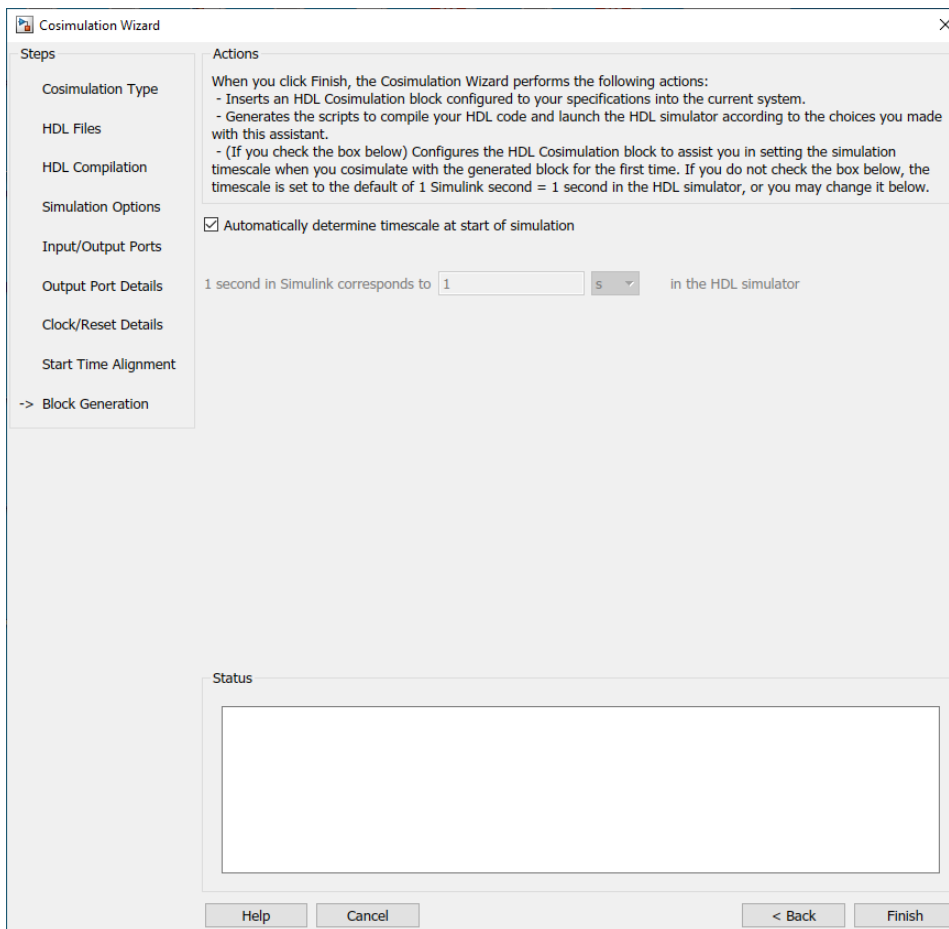
To avoid a race condition, make sure the start time does not coincide with the active edge of any clocks. You can do this by moving the start time or by changing clock active edges in the previous step.

- To make sure the start time is where you want it.

The HDL simulator start time is calculated from the clock and reset values on the previous pane. If you want, you can change the HDL simulator start time by entering a new value where you see **HDL time to start cosimulation (ns)**. Click **Update plot** to see your change applied.

- 2 Click **Next**.

Generate Block



- 1 Specify if you want HDL Verifier to determine the timescale when you start the simulation by selecting **Automatically determine timescale at start of simulation**. If you prefer to determine the timescale yourself, leave this box unchecked and enter the timescale value in the text boxes below. The default is to automatically determine timescale.

For more about timescales, see “Simulation Timescales” on page 10-44.

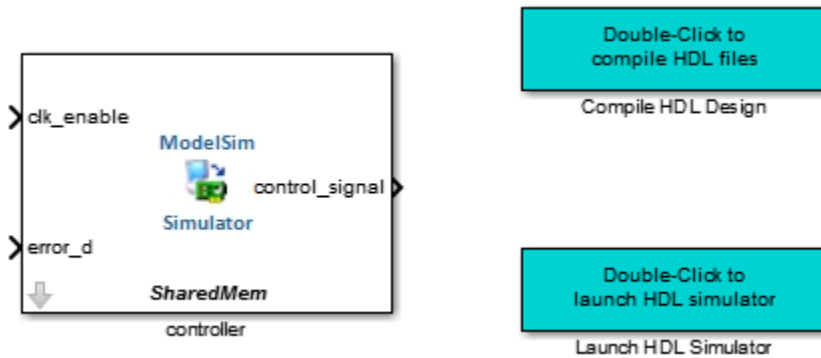
- 2 Click **Back** to review or change your settings.
- 3 Click **Finish** to generate the HDL cosimulation block.

Complete Simulink Model

The **Cosimulation Wizard** tool inserts the following items to your model:

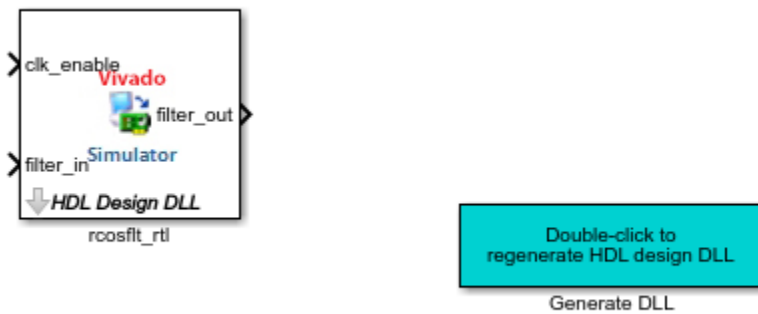
ModelSim or Xcelium users:

- An HDL Cosimulation block
- A utility function to compile the HDL design
- A utility function to launch the HDL simulator



Vivado users:

- An HDL Cosimulation block
- A utility block to generate a DLL file



- 1 Place the block so that the inputs and outputs to the HDL Cosimulation block line up.
- 2 Connect the blocks in the destination model to the HDL Cosimulation block.

Note If you opened the **Cosimulation Wizard** from the command line and not from the Simulink toolstrip, the HDL Cosimulation and the utility functions open in a new model. You first have to copy them to your model.

When you have completed the model, see “Performing Cosimulation” on page 9-37 for the next steps in HDL cosimulation.

See Also
Cosimulation Wizard

Related Examples

- “Supported Data Types” on page 10-35

Use HDL Parameters in Cosimulation

You can configure Verilog or VHDL parameters in a cosimulation. When you use the **Cosimulation Wizard** to generate an HDL Cosimulation block or an `hdlverifier.HDLCosimulation` System object, the **Simulation Options** step creates a configuration file named `parameter_DUT.cfg`, where *DUT* is the name of your HDL DUT. The configuration file includes a line for each HDL parameter, with a default value assigned. Uncomment the line for the parameter you want to configure and assign a value to override the default value.

For example, consider this generated configuration file, generated for ModelSim cosimulation.

```
# Uncomment lines below for any parameter whose default value you want to change.
# For parameters marked "N/A" (not available) the default value could not be
# determined, but you can override in the same way.

#-G/design_top/coeff1=0
#-G/design_top/coeff2=18
#-G/design_top/coeff3=74
```

To change the value of `coeff1` to 32, uncomment that line and assign a value of 32.

```
-G/design_top/coeff1=32
#-G/design_top/coeff2=18
#-G/design_top/coeff3=74
```

Similarly, when you cosimulate with Xcelium, the parameters in the configuration file are created with the `-gpg` directive to force value assignment for generics and parameters.

```
-gpg "design_top.coeff1=120
#-gpg "design_top.coeff2=18
#-gpg "design_top.coeff3=74
```

Supported Data Types

Supported Verilog data types

- Integer — Up to 32 bit
- Real
- String — Up to 256 byte

Supported VHDL data types

- Integer
- Real
- String — Up to 256 byte
- Time
- Bit
- Boolean
- Enum
- `std_logic`

This feature is not supported for Vivado cosimulation

See Also
Cosimulation Wizard

Related Examples

- “Import HDL Code for MATLAB System Object” on page 9-12
- “Import HDL Code for HDL Cosimulation Block” on page 9-24

Performing Cosimulation

When you are finished creating a function, System object, or block, select the topic below that describes how you to cosimulate your HDL code.

If you generated this cosimulation interface:	Select one of these topics:
MATLAB test bench function (<code>matlabtb</code>) Not supported for Vivado cosimulation.	<ul style="list-style-type: none"> • With a completed test bench, you can start at the next step: “Place Test Bench onve MATLAB Search Path” on page 2-13 • Review entire test bench function workflow: “Create a MATLAB Test Bench” on page 2-2
MATLAB component function (<code>matlabcp</code>) Not supported for Vivado cosimulation.	<ul style="list-style-type: none"> • With a completed component, you can start at the next step: “Place Component Function on MATLAB Search Path” on page 3-8 • Review entire test bench function workflow: “Create a MATLAB Component Function” on page 3-2
MATLAB System object	With a completed System object, you are ready to use it for HDL verification. See “Verify Viterbi Decoder Using MATLAB System Object and HDL Simulator” on page 32-145 for an example of using the MATLAB System object for HDL cosimulation.
Simulink Block	Place your HDL cosimulation block within a test bench or component model. See “Create a Simulink Cosimulation Test Bench” on page 6-6 or “Create Simulink Model for Component Cosimulation” on page 7-5. After you have an HDL cosimulation model, run the simulation. See “Run a Simulink Cosimulation Session” on page 5-4.

You can also view the following examples:

- “Verify Raised Cosine Filter Design Using MATLAB” on page 9-38
- “Verify Raised Cosine Filter Design Using Simulink” on page 9-51

Verify Raised Cosine Filter Design Using MATLAB

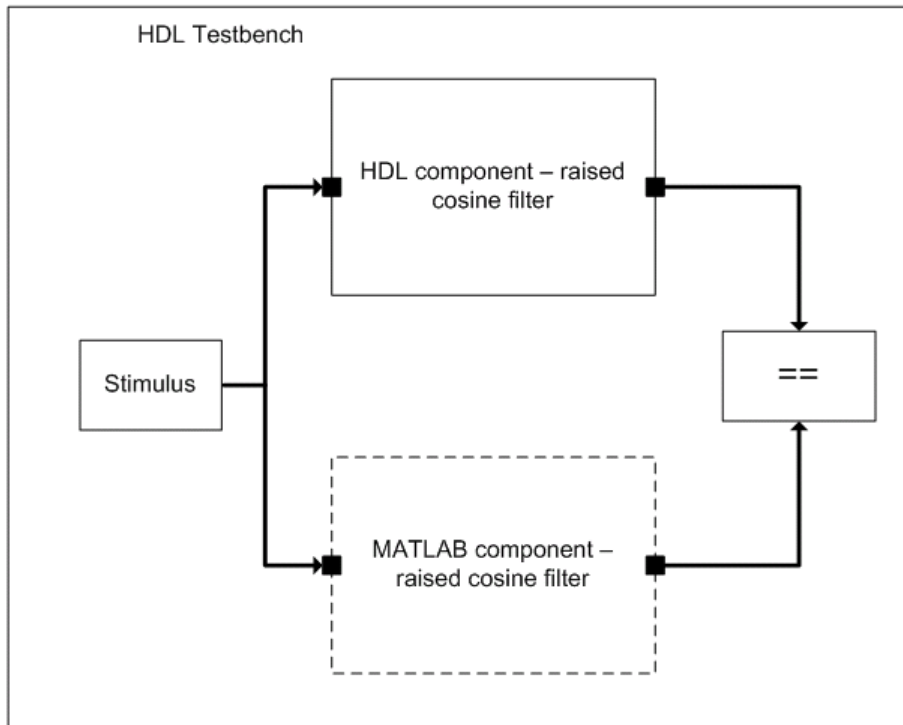
In this section...
“MATLAB and Cosimulation Wizard Tutorial Overview” on page 9-38
“Tutorial: Set Up Tutorial Files (MATLAB)” on page 9-39
“Tutorial: Launch Cosimulation Wizard (MATLAB)” on page 9-39
“Tutorial: Configure the Component Function with the Cosimulation Wizard” on page 9-40
“Tutorial: Customize Callback Function” on page 9-46
“Tutorial: Run Cosimulation and Verify HDL Design” on page 9-49

MATLAB and Cosimulation Wizard Tutorial Overview

This tutorial guides you through the basic steps for setting up an HDL Verifier cosimulation that uses MATLAB and the HDL Simulator. This cosimulation verifies an HDL design using a MATLAB component as the test bench. In this tutorial, you perform the steps to cosimulate MATLAB with the HDL simulator to verify the suitability of a raised cosine filter written in Verilog. Note that Vivado cosimulation is not supported with MATLAB function.

Note This tutorial requires MATLAB, HDL Verifier, Fixed-Point Designer™, and ModelSim or Xcelium HDL simulator. This tutorial also assumes that you have read “Import HDL Code for MATLAB Function” on page 9-4.

The HDL test bench instantiates two raised-cosine filter components: one is implemented in HDL, and the other is associated with a MATLAB callback function. The test bench also generates stimulus to both filters and compares their outputs.



Tutorial: Set Up Tutorial Files (MATLAB)

To help others access copies of the tutorial files, set up a folder for your own tutorial work by following these instructions:

- 1 Create a folder outside the scope of your MATLAB installation folder into which you can copy the tutorial files. The folder must be writable. This tutorial assumes that you create a folder named `MyTests`.
- 2 Copy all the files located in the following MATLAB folder to the folder you created:

```
matlabroot\toolbox\edalink\foundation\hdlldemo_src\tutorial
```

where *matlabroot* is the MATLAB root directory on your system.

- 3 You now have the following files in your working folder:

- `filter_tb.v`
- `mycallback_solution.m`
- `rcosflt_beh.v`
- `rcosflt_rtl.v`
- `rcosflt_tb.mdl` (not used in this tutorial)

Tutorial: Launch Cosimulation Wizard (MATLAB)

- 1 Start MATLAB.
- 2 Set the folder you created in "Tutorial: Set Up Tutorial Files (MATLAB)" on page 9-39 as your current folder in MATLAB.

- 3 At the MATLAB command prompt, enter:

```
>>cosimWizard
```

This command launches the Cosimulation Wizard.

Tutorial: Configure the Component Function with the Cosimulation Wizard

This tutorial leads you through the following wizard pages, designed to assist you in creating an HDL Verifier component function:

- “Tutorial: Specify Cosimulation Type (MATLAB)” on page 9-40
- “Tutorial: Select HDL Files (MATLAB)” on page 9-41
- “Tutorial: Specify HDL Compilation Commands (MATLAB)” on page 9-42
- “Tutorial: Select HDL Modules for Cosimulation (MATLAB)” on page 9-43
- “Tutorial: Specify Callback Schedule” on page 9-44
- “Tutorial: Generate Script” on page 9-45

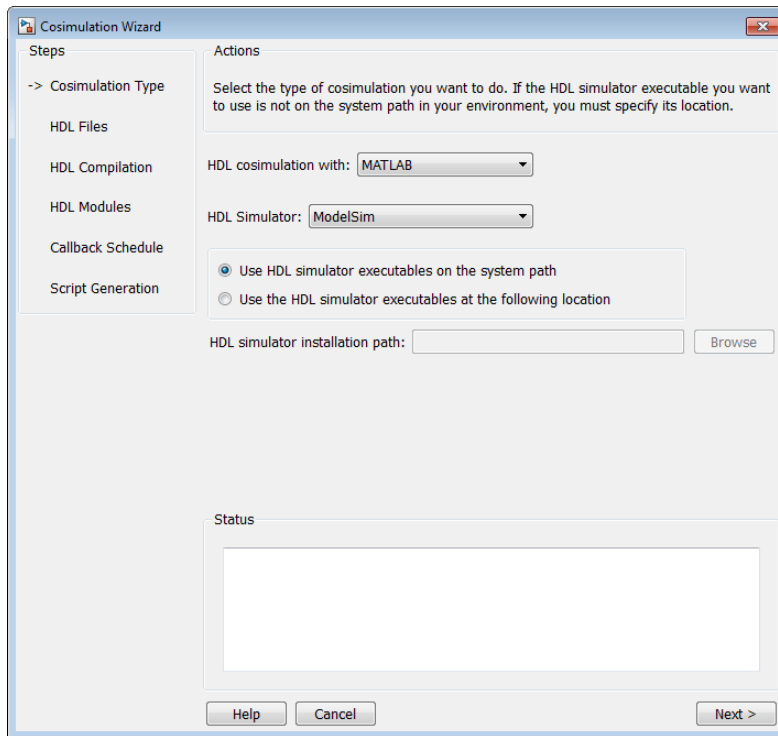
Tutorial: Specify Cosimulation Type (MATLAB)

In the Cosimulation Type page, perform the following steps:

- 1 Change **HDL cosimulation with** option set to MATLAB.
- 2 If you are using ModelSim, leave **HDL Simulator** option as ModelSim.
If you are using Xcelium, change **HDL Simulator** option to Xcelium.
- 3 Leave the default option **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path.

If the executables do not appear in the path, specify the HDL simulator path as described in “Cosimulation Type—MATLAB Function” on page 9-4.

4

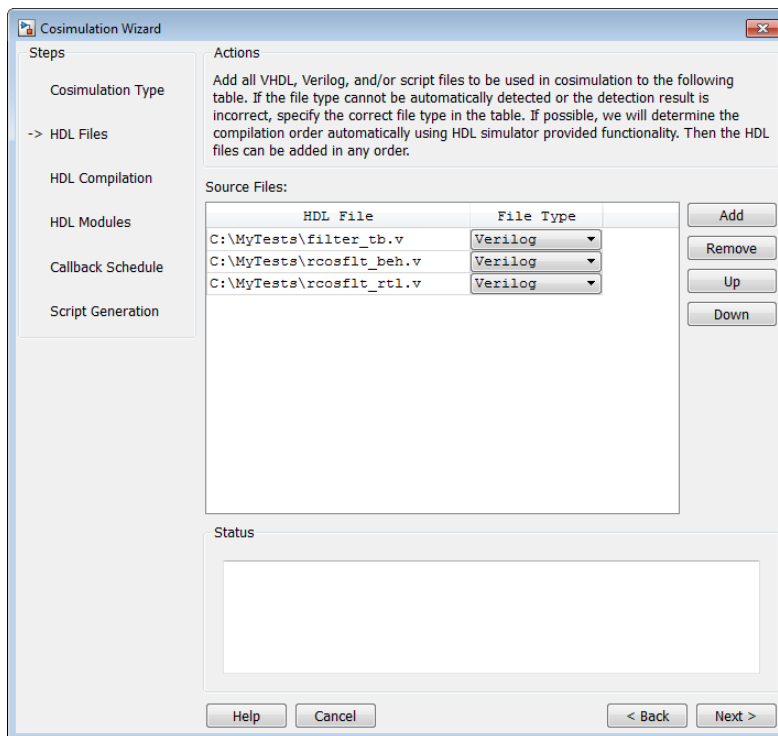


5 Click **Next** to proceed to the HDL Files page.

Tutorial: Select HDL Files (MATLAB)

In the HDL Files page, perform the following steps:

- 1 Add HDL files to file list.
 - a Click **Add** and browse to the directory you created in “Tutorial: Set Up Tutorial Files (MATLAB)” on page 9-39.
 - b Select the Verilog files `filter_tb.v`, `rcosflt_rtl.v`, and `rcosflt_beh.v`. You can select multiple files in the file browser by holding down the **CTRL** key while selecting the files with the mouse.
 - c Review the file in the file list with the file type identified as you expected.

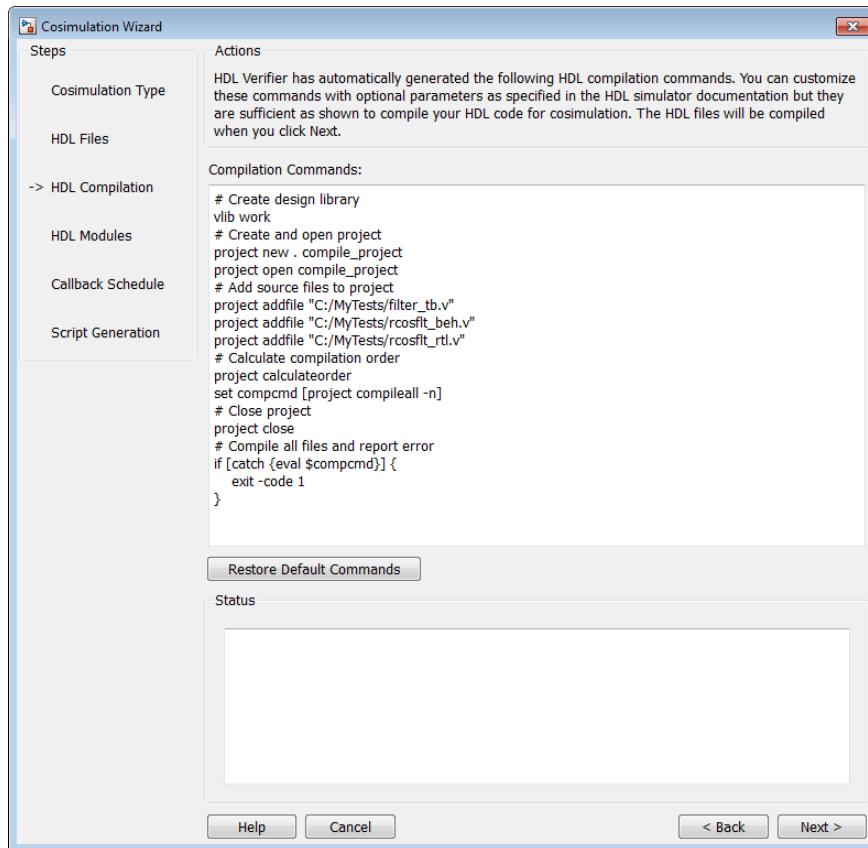


- 2 Click **Next** to proceed to the HDL Compilation page.

Tutorial: Specify HDL Compilation Commands (MATLAB)

Cosimulation Wizards lists the default commands in the Compilation Commands window. You do not need to change these defaults for this tutorial.

- 1 Examine compilation commands.
 - a ModelSim users: Your HDL Compilation pane looks similar to the following.



- b** Xcelium users: Your HDL compilation commands will look similar to the following:

```
xmvlog -64bit -update "/mathworks/home/user/MyTests/filter_tb.v"
xmvlog -64bit -update "/mathworks/home/user/MyTests/rcosflt_beh.v"
xmvlog -64bit -update "/mathworks/home/user/MyTests/rcosflt_rtl.v"
```

- 2** Click **Next** to proceed to the HDL Modules pane.

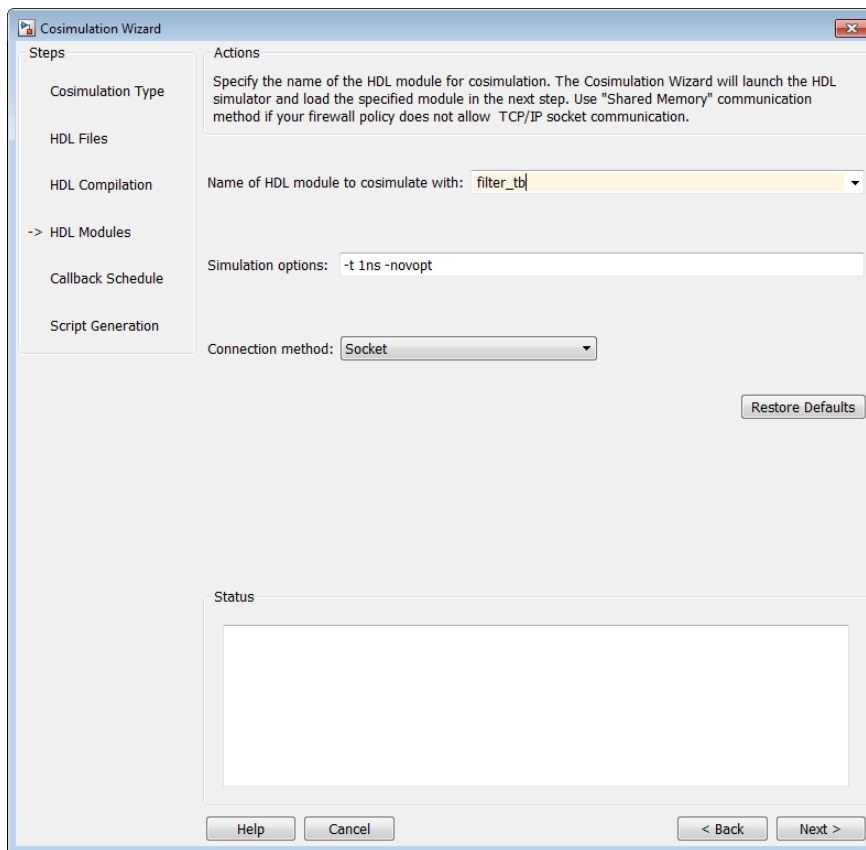
The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area. Change whatever settings you can to remove the error before proceeding to the next step.

Tutorial: Select HDL Modules for Cosimulation (MATLAB)

In the HDL Modules pane, perform the following steps:

- 1** Specify the name of the HDL module/entity for cosimulation.

At **Name of HDL module to cosimulate with**, select `filter_tb` from the drop-down list to specify the Verilog module you will use for cosimulation.



If you do not see `filter_tb` in the drop-down list, you can enter it manually.

- 2 For **Connection method**, select Shared Memory if your firewall policy does not allow TCP/IP socket communication.
- 3 Click **Next** to proceed to the Callback Schedule page.

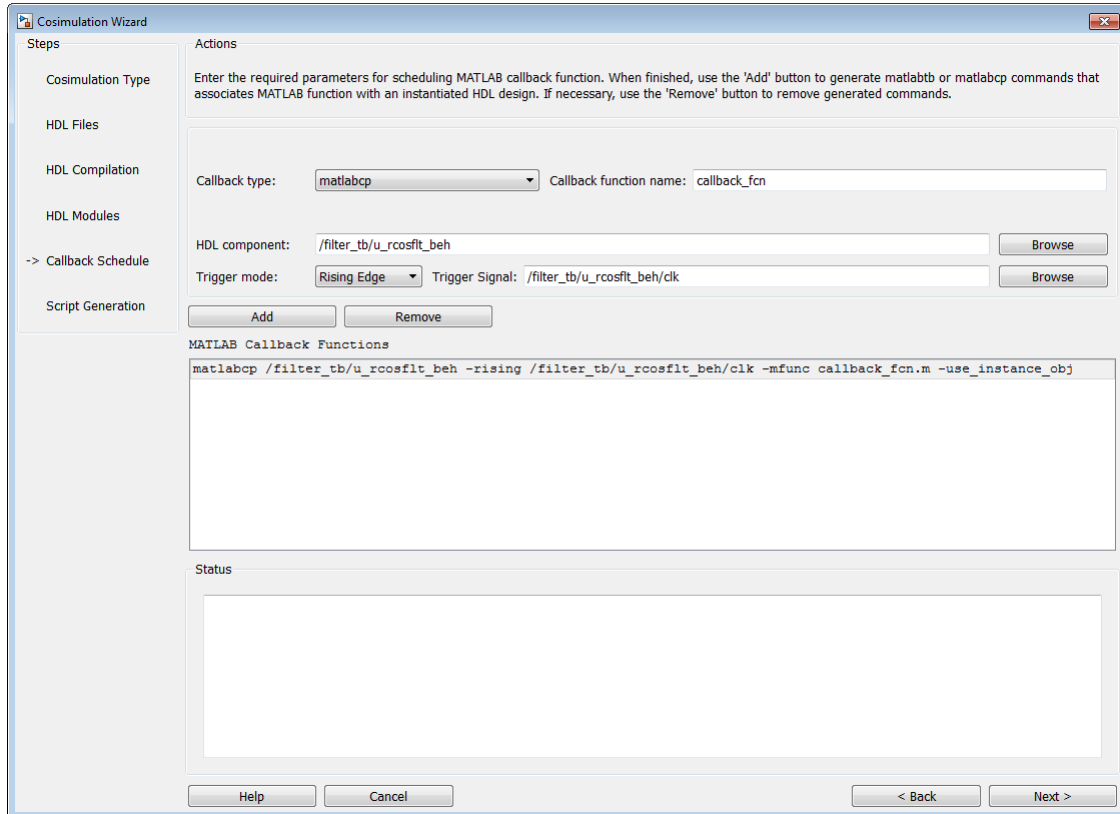
Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. After the wizard launches the HDL simulator, the Callback Schedule page appears. On Windows systems, the console remains open. Do not close the console; the application closes this window upon completion.

Tutorial: Specify Callback Schedule

In the Callback Schedule page, perform the following steps:

- 1 Leave **Callback type** as `matlabcp` (default). This type instructs the Cosimulation Wizard to create a MATLAB callback function as a component for cosimulation with the HDL simulator.
- 2 Leave **Callback function name** as `callback_fcn`. The wizard gives this name to the generated MATLAB callback function.
- 3 For **HDL component**, click **Browse**. Click the expander icon next to `filter_tb` to expand the selection. Select `u_rcosflt_beh`, and click **OK**. You have specified to the Cosimulation Wizard that the HDL simulator associate this component with the MATLAB callback function.
- 4 Set **Trigger mode** to Rising Edge.
- 5 For **Trigger Signal**, click **Browse**. Click the expander icon next to `filter_tb` to expand the selection. Select `u_rcosflt_beh`. In the ports list on the right, select `clk`. Click **OK**.

- 6 Click **Add**. The Cosimulation Wizard generates the corresponding `matlabcp` command that associates the HDL module `u_rcosflt_beh` with the MATLAB function `callback_fcn`, as shown in the following image:

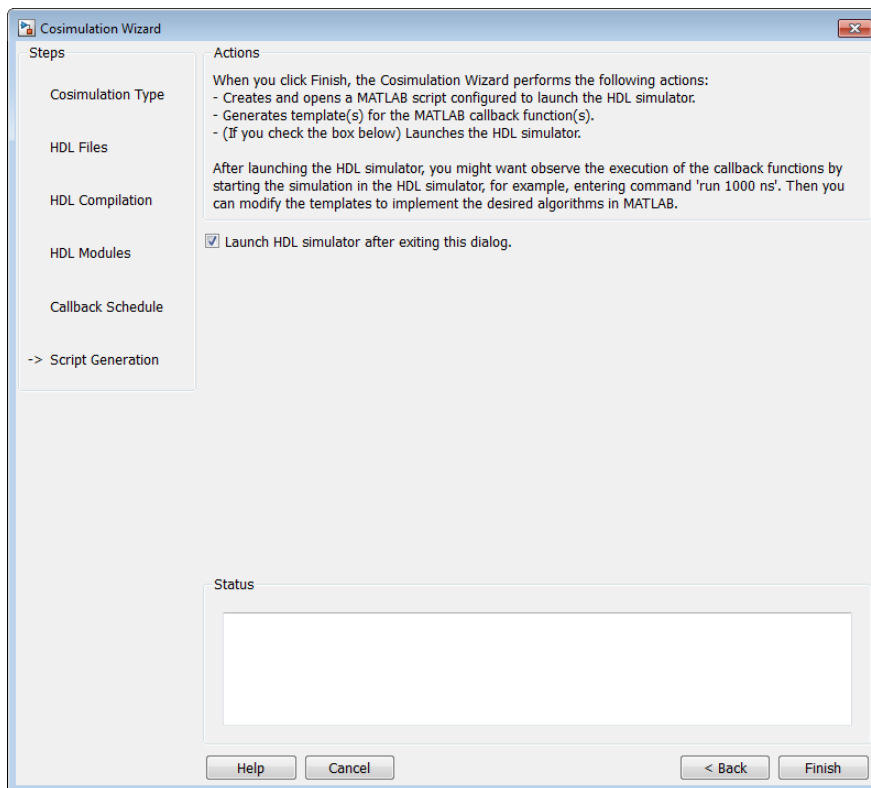


For more information on the callback parameters, see the reference page for `matlabcp`.

- 7 Click **Next** to proceed to the Generate Script page.

Tutorial: Generate Script

- 1 Leave **Launch HDL simulator after exiting this dialog** selected.



- 2 Click **Finish** to complete the Cosimulation Wizard session and generate scripts.

Tutorial: Customize Callback Function

After you click **Finish** in the Cosimulation Wizard, the application generates three HDL files in the current directory:

- `compile_hdl_design.m`: For recompiling the HDL design
- `launch_hdl_simulator.m`: To relaunch the MATLAB server and start the HDL simulator.
- `callback_fcfn.m`: The MATLAB callback function

In addition to launching the HDL simulator, HDL Verifier software opens the MATLAB Editor and loads `callback_fcfn.m` (partial image shown).

```

1 function callback_fcn(obj)
2 %
3 % MATLAB callback function template associated with HDL component(s):
4 % /rcosflt_rtl;
5 %
6 % File Name: callback_fcn.m
7 % Created: 02-Jun-2010 09:33:10
8 %
9 % Generated by EDA Cosimulation Assistant
10
11
12 % --- Initialize internal state(s) of callback function ---
13 if (strcmp(obj.simstatus,'Init'))
14     disp('Initializing states ...');
15     obj.userdata.State = 0;
16 end
17
18 % Display obj.tnow, which is the current HDL simulation time specified in
19 % seconds.
20 disp(['Callback function is executed at time ' num2str(obj.tnow)]);
21
22 % --- Read signal from HDL component ---
23 % Variable obj.portvalues.PortName contains the input value of port with
24 % name 'PortName' on the associated HDL component. The list of readable
25 % ports can be determined from the fields in struct obj.portinfo.out and
26 % obj.portinfo.inout.
27 %
28 % If obj.portvalues.PortName is multi-valued logic vector, you can convert
29 % it to decimal using function mvl2dec, e.g.,
30 %     decValue = mvl2dec(obj.portvalues.PortName, true);
31 %
32 % Optionally, you can also translate the port value into fixed-point
33 % object, e.g.
34 %     myfiobj = fi(decValue,1, 16, 4);
35

```

The generated template comprises four parts:

- Initialize internal state(s) of callback function
- Read signal from HDL component
- Write signal to HDL component
- Update internal state(s)

You modify this template to model a raised cosine filter in MATLAB following the instructions as shown in the following sections.

- “Tutorial: Define Internal States” on page 9-48
- “Tutorial: Read Signal from HDL Component” on page 9-48
- “Tutorial: Write Signal to HDL Component” on page 9-48
- “Tutorial: Update Internal States” on page 9-49

Note You can find a completed modified callback function in `mycallback_solution.m`. This function resides in the directory you copied the tutorial files into. You can use this file to overwrite the one in your current directory. Name the file "callback_fcn.m", and change the function name to `callback_fcn`.

Tutorial: Define Internal States

Define two internal states: a 49-element vector to hold filter inputs and a vector of filter coefficients.

Edit `callback_fcn.m` so that the internal state section contains the following code:

```

12 % --- Initialize internal state(s) of callback function ---
13 if (strcmp(obj.simstatus,'Init'))
14     disp('Initializing states ...');
15     obj.userdata.State = zeros(1, 49);
16     obj.userdata.Coeff = [ ...
17         0,    18,    74,    165,    269,    350,    360,    254,    0,
18     ...
19     -405,   -925,  -1476,  -1937,  -2158,  -1986,  -1292,    0,  1889,
20     ...
21     4285,   7010,   9817,  12420,  14530,  15906,  16384,  15906,  14530,
22     ...
23     12420,   9817,   7010,   4285,   1889,    0,  -1292,  -1986,  -2158,
24     ...
25     -1937,  -1476,  -925,  -405,    0,   254,   360,   350,   269,
26     ...
27     165,    74,    18,    0]; % Filter coefficients, sfix16_En14
28 end
29

```

Tutorial: Read Signal from HDL Component

Read the filter input and convert it to a decimal number in MATLAB.

Edit `callback_fcn.m` so that the read signal section contains the following code:

```

25 % --- Read signal from HDL component ---
26 portValueDec = mvl2dec( ...
27     obj.portvalues.filter_in, ...
28     true);
29

```

Tutorial: Write Signal to HDL Component

The input "reset" signal controls the filter output. If reset is low, then the output is the product of previous inputs and filter coefficients. MATLAB converts the decimal result to a multivalued logic output of the HDL component.

Edit `callback_fcn.m` so that the write signal section contains the following code:

```

46
47 % --- Write signal to HDL component ---
48 - if(obj.portvalues.reset == '1')
49 -     filter_out = 0;
50 - else
51 -     filter_out = (obj.userdata.State * obj.userdata.Coef');
52 - end
53 - obj.portvalues.filter_out = dec2mvl(...
54 -     filter_out, 34);
55

```

Tutorial: Update Internal States

Use the filter input to update the internal 49-element state.

Edit `callback_fcn.m` so that the update internal states section contains the following code:

```

66
67 % --- Update internal state(s) ---
68 - disp(['Updated internal state: ' num2str(obj.userdata.State)]);
69 - if(obj.portvalues.reset == '1')
70 -     obj.userdata.State = zeros(1, 49);
71 - else
72 -     obj.userdata.State = [portValueDec obj.userdata.State(1:48)];
73 - end
74
75

```

Tutorial: Run Cosimulation and Verify HDL Design

Switch to the HDL simulator and enter the following command in the HDL simulator console:

```
run 200 ns
```

You see the following output displayed in the HDL simulator:

```
VSIM 2> run 200 ns
# At time 31, output of RTL module matches output of MATLAB.
# At time 41, output of RTL module matches output of MATLAB.
# At time 51, output of RTL module matches output of MATLAB.
# At time 61, output of RTL module matches output of MATLAB.
# At time 71, output of RTL module matches output of MATLAB.
# At time 81, output of RTL module matches output of MATLAB.
# At time 91, output of RTL module matches output of MATLAB.
# At time 101, output of RTL module matches output of MATLAB.
# At time 111, output of RTL module matches output of MATLAB.
# At time 121, output of RTL module matches output of MATLAB.
# At time 131, output of RTL module matches output of MATLAB.
# At time 141, output of RTL module matches output of MATLAB.
# At time 151, output of RTL module matches output of MATLAB.
# At time 161, output of RTL module matches output of MATLAB.
# At time 171, output of RTL module matches output of MATLAB.
# At time 181, output of RTL module matches output of MATLAB.
# At time 191, output of RTL module matches output of MATLAB.
VSIM 3>
```

These messages indicate that the output of the HDL component matches the behavioral output of the MATLAB component.

Verify Raised Cosine Filter Design Using Simulink

In this section...

“Simulink and Cosimulation Wizard Tutorial Overview” on page 9-51

“Tutorial: Set Up Tutorial Files (Simulink)” on page 9-51

“Tutorial: Launch Cosimulation Wizard (Simulink)” on page 9-52

“Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard” on page 9-52

“Tutorial: Create Test Bench to Verify HDL Design” on page 9-61

“Tutorial: Run Cosimulation and Verify HDL Design” on page 9-63

Simulink and Cosimulation Wizard Tutorial Overview

This tutorial guides you through the basic steps for setting up an HDL Verifier application that uses Simulink and the HDL simulator to verify an HDL design, using a Simulink model as the test bench. In this tutorial, you perform the steps to cosimulate Simulink and the HDL simulator to verify a simple raised cosine filter written in Verilog.

Note This tutorial requires Simulink, HDL Verifier, Fixed-Point Designer, and ModelSim or Xcelium HDL simulator. This tutorial assumes that you have read “Import HDL Code for HDL Cosimulation Block” on page 9-24.

In this tutorial, you perform the following steps:

- 1 “Tutorial: Set Up Tutorial Files (Simulink)” on page 9-51
- 2 “Tutorial: Launch Cosimulation Wizard (Simulink)” on page 9-52
- 3 “Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard” on page 9-52
- 4 “Tutorial: Create Test Bench to Verify HDL Design” on page 9-61
- 5 “Tutorial: Run Cosimulation and Verify HDL Design” on page 9-63

Tutorial: Set Up Tutorial Files (Simulink)

To help others access copies of the tutorial files, set up a folder for your own tutorial work by following these instructions:

- 1 Create a folder outside the scope of your MATLAB installation folder into which you can copy the tutorial files. The folder must be writable. This tutorial assumes that you create a folder named `MyTests`.
- 2 Copy all the files located in the following directory to the folder you created:

```
matlabroot\toolbox\edalink\foundation\hdlmlink\demo_src\tutorial
```

where `matlabroot` is the MATLAB root directory on your system.

- 3 You now have all the following files in your working directory, although, for this tutorial, you will need only two of them:

- `filter_tb.v` (not used for this tutorial)

- mycallback_solution.m (not used for this tutorial)
- rcosflt_beh.v (not used for this tutorial)
- rcosflt_rtl.v
- rcosflt_rtl.vhd
- rcosflt_tb.mdl

Tutorial: Launch Cosimulation Wizard (Simulink)

- 1 Start MATLAB.
- 2 Set the directory you created in “Tutorial: Set Up Tutorial Files (Simulink)” on page 9-51 as your current directory in MATLAB.
- 3 At the MATLAB command prompt, enter the following:

```
>>cosimWizard
```

The command launches the Cosimulation Wizard.

Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard

This tutorial leads you through the following wizard pages, designed to assist you in creating an HDL Cosimulation block.

- “Tutorial: Specify Cosimulation Type (Simulink)” on page 9-52
- “Tutorial: Select HDL Files (Simulink)” on page 9-53
- “Tutorial: Specify HDL Compilation Commands (Simulink)” on page 9-54
- “Tutorial: Select Simulation Options for Cosimulation (Simulink)” on page 9-55
- “Tutorial: Specify Port Types” on page 9-57
- “Tutorial: Specify Output Port Details” on page 9-58
- “Tutorial: Set Clock and Reset Details” on page 9-58
- “Tutorial: Confirm Start Time Alignment” on page 9-59
- “Tutorial: Generate Block” on page 9-60

Tutorial: Specify Cosimulation Type (Simulink)

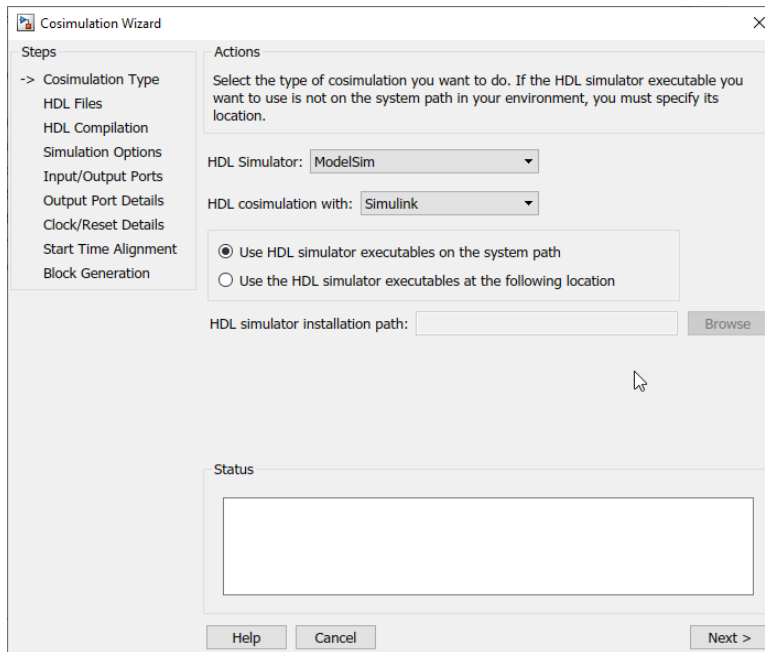
In the Cosimulation Type page, perform the following steps:

- 1 Leave **HDL cosimulation with** option set to Simulink.
- 2 If you are using ModelSim, leave **HDL Simulator** option as ModelSim.

If you are using Xcelium or Vivado, change **HDL Simulator** option to Xcelium or Vivado Simulator respectively.

- 3 Leave the default option **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path.

If these executable do not appear on the path, specify the HDL simulator path as described in “Cosimulation Type—Simulink Block” on page 9-24.

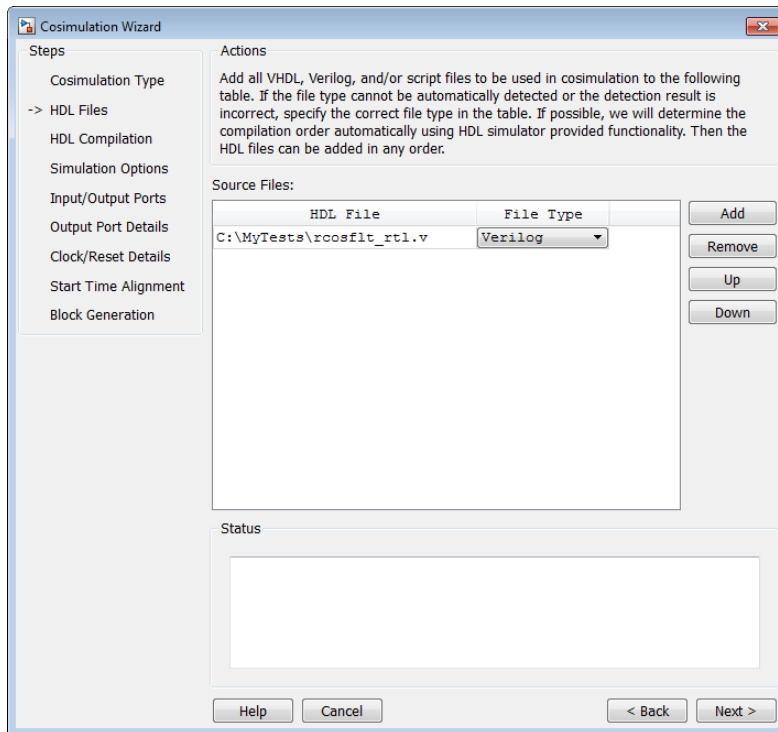


- 4 Click **Next** to proceed to the HDL Files page.

Tutorial: Select HDL Files (Simulink)

In the HDL Files page, perform the following steps:

- 1 Add HDL files to file list.
 - a Click **Add** and browse to the directory you created in "Tutorial: Set Up Tutorial Files (Simulink)" on page 9-51.
 - b For Verilog, select `rcosflt_rtl.v`. For VHDL, select `rcosflt_rtl.vhd`.
 - c Review the file in the file list with the file type identified as you expected.



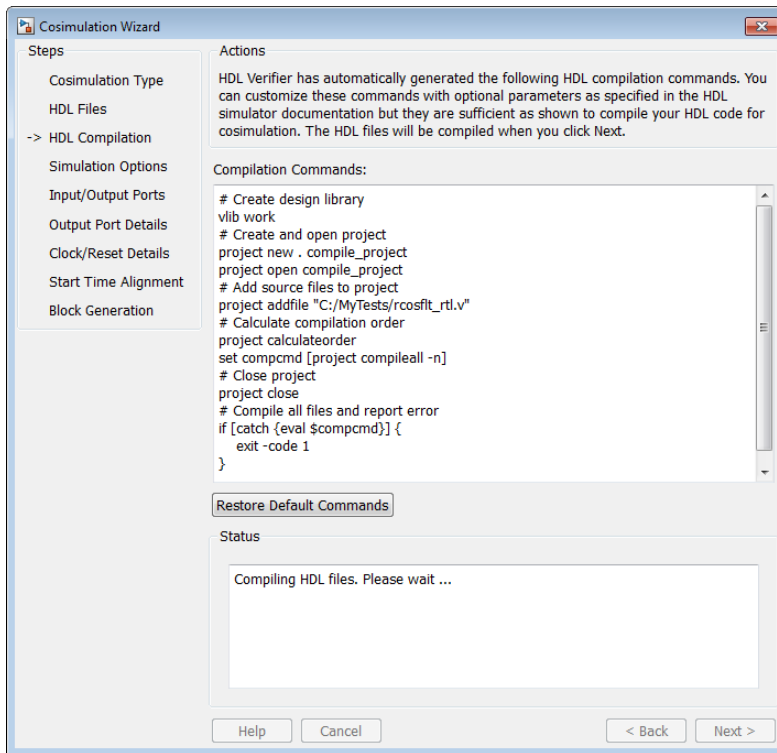
2 Click **Next** to proceed to the HDL Compilation page.

Tutorial: Specify HDL Compilation Commands (Simulink)

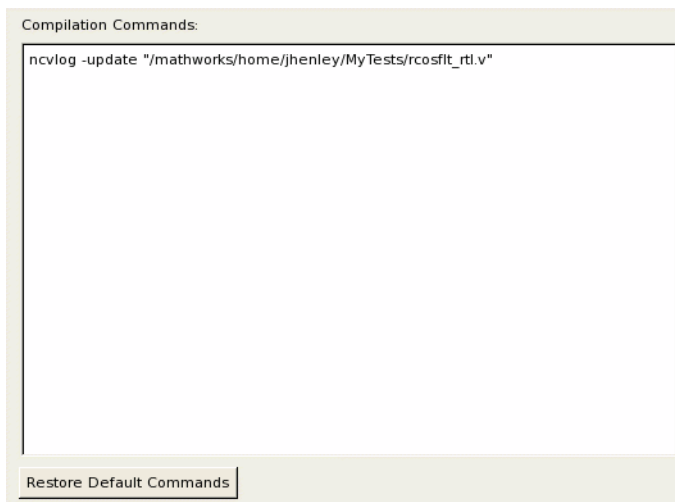
The Cosimulation Wizard lists the default commands in the Compilation Commands window. You do not need to change these commands for this tutorial.

When you run the Cosimulation Wizard with your own code, you may add or change the compilation commands in this window. For example, you can add the `-vlog01compat` switch.

ModelSim users: The HDL Compilation pane will look similar to the one in this figure:



Xcelium users: Your HDL Compilation pane will look similar to the one in the following figure.



Click **Next** to proceed to the HDL Modules pane.

The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area. Change whatever settings you can to remove the error before proceeding to the next step.

Tutorial: Select Simulation Options for Cosimulation (Simulink)

If you are using ModelSim or Xcelium, perform the following steps in the Simulation Options pane:

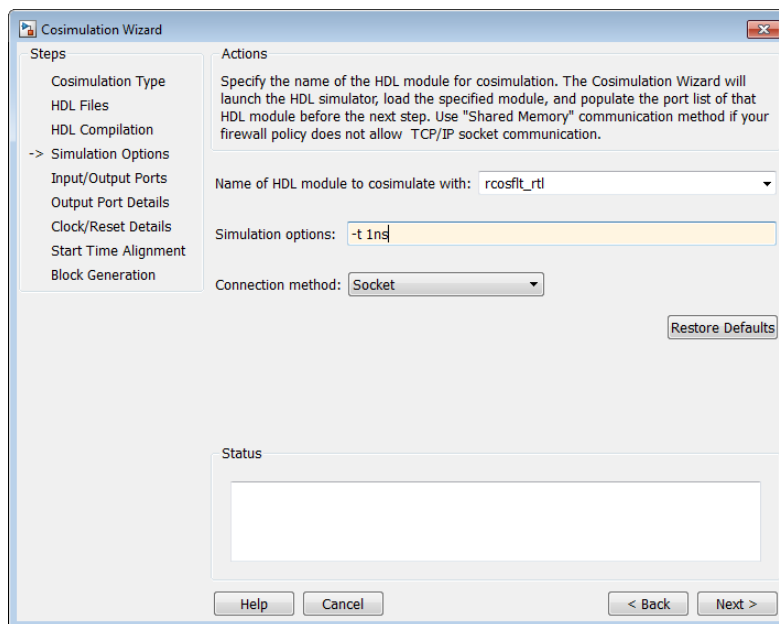
- 1 Specify the name of HDL module/entity for cosimulation.

From the drop-down list, select `rcosflt_rtl`. This module is the Verilog/VHDL module you use for cosimulation.

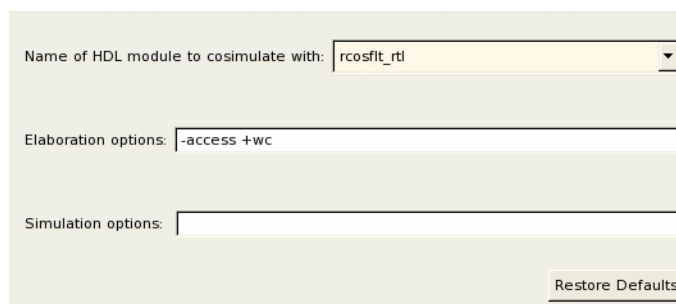
If you do not see `rcosflt_rtl` in the drop-down list, you can enter the file name manually.

- 2 For **Connection method**, select Shared Memory if your firewall policy does not allow TCP/IP socket communication.

The simulation options now look similar to those shown in the next figure.



Xcelium users: Your HDL Module options look similar to the following figure



- 3 Click **Next** to proceed to the Simulink Ports pane.

The Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. After the wizard launches the HDL simulator, the wizard populates the input and output ports on the Verilog/VHDL model `rcosflt_rtl` and displays them in the next step.

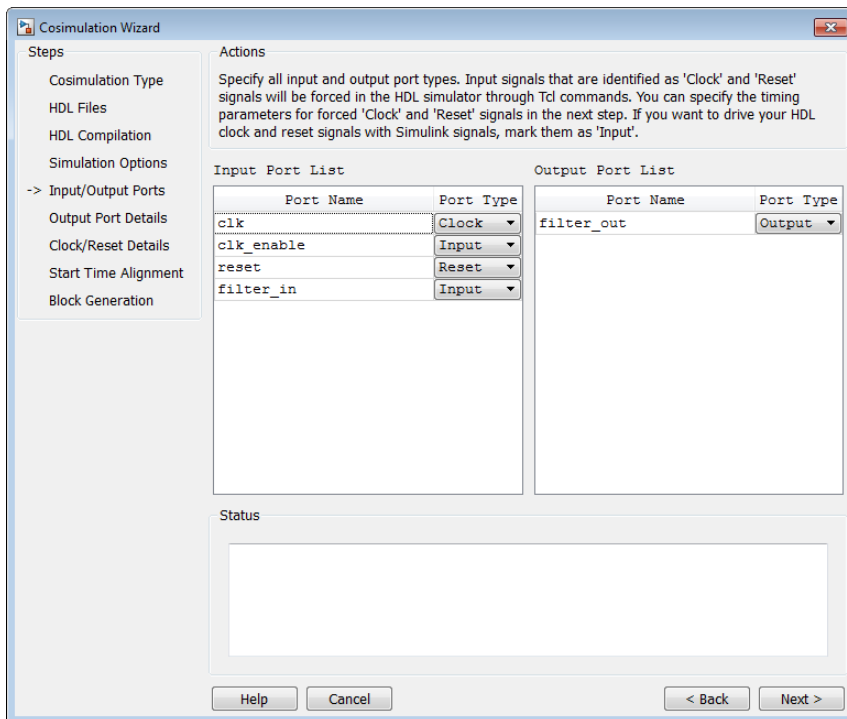
If you are using the Vivado simulator, Simulation Options pane looks a bit different.

- 1 **Name of HDL module to cosimulate with** is a read-only parameter, and it displays the top HDL module specified in the “Tutorial: Select HDL Files (Simulink)” on page 9-53 stage.
- 2 If you want to generate a waveform file for internal signals, set the **Debug internal signals** parameter to **wave**. This option might affect simulation performance. To disable waveform generation, set this parameter to **off**.
- 3 Set the **HDL time precision** parameter to the required time precision.
- 4 Click **Next** to proceed to the **Input/Output Ports** pane.

The Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. After the wizard launches the HDL simulator, the wizard populates the input and output ports on the Verilog/VHDL model `rcosflt_rtl` and displays them in the next step.

Tutorial: Specify Port Types

In this step, the **Cosimulation Wizard** displays two tables containing the input and output ports of `rcosflt_rtl`, respectively.



The **Cosimulation Wizard** attempts to identify the port type for each port. If the wizard incorrectly identifies a port, you can change the port type using these tables.

- For input ports, you can select from **Clock**, **Reset**, **Input**, or **Unused**. HDL Verifier connects only the input ports marked **Input** to Simulink during cosimulation.
- HDL Verifier connects output ports marked **Output** with Simulink during cosimulation. The wizard and Simulink ignore those output ports marked **Unused** during cosimulation.
- You can change the parameters for signals identified as **Clock** and **Reset** at a later step.

Accept the default port types and click **Next** to proceed to the **Output Port Details** page.

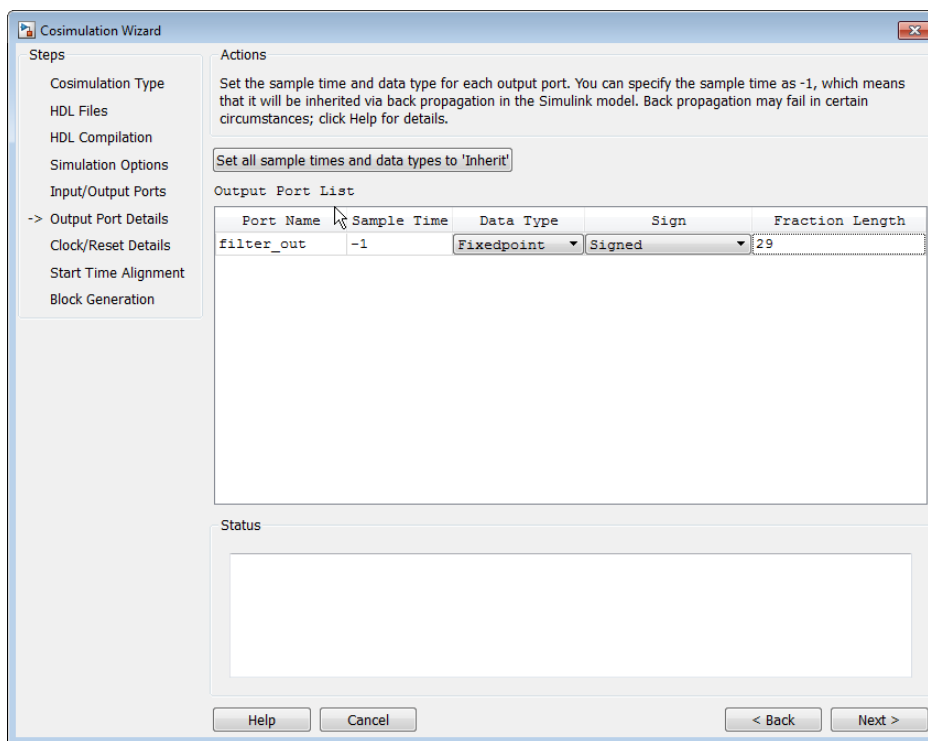
Tutorial: Specify Output Port Details

In the Output Port Details page, perform the following steps:

- 1 Set the sample time of `filter_out` to -1 to inherit via back propagation.
- 2 You can see from the Verilog code that the Cosimulation Wizard represents the output in a S34,29 format. Change the following fields:

- Data Type to Fixedpoint
- Sign to Signed
- Fraction Length to 29

. Your results now look similar to the following image.



- 3 Click **Next** to proceed to the Clock/Reset Details page.

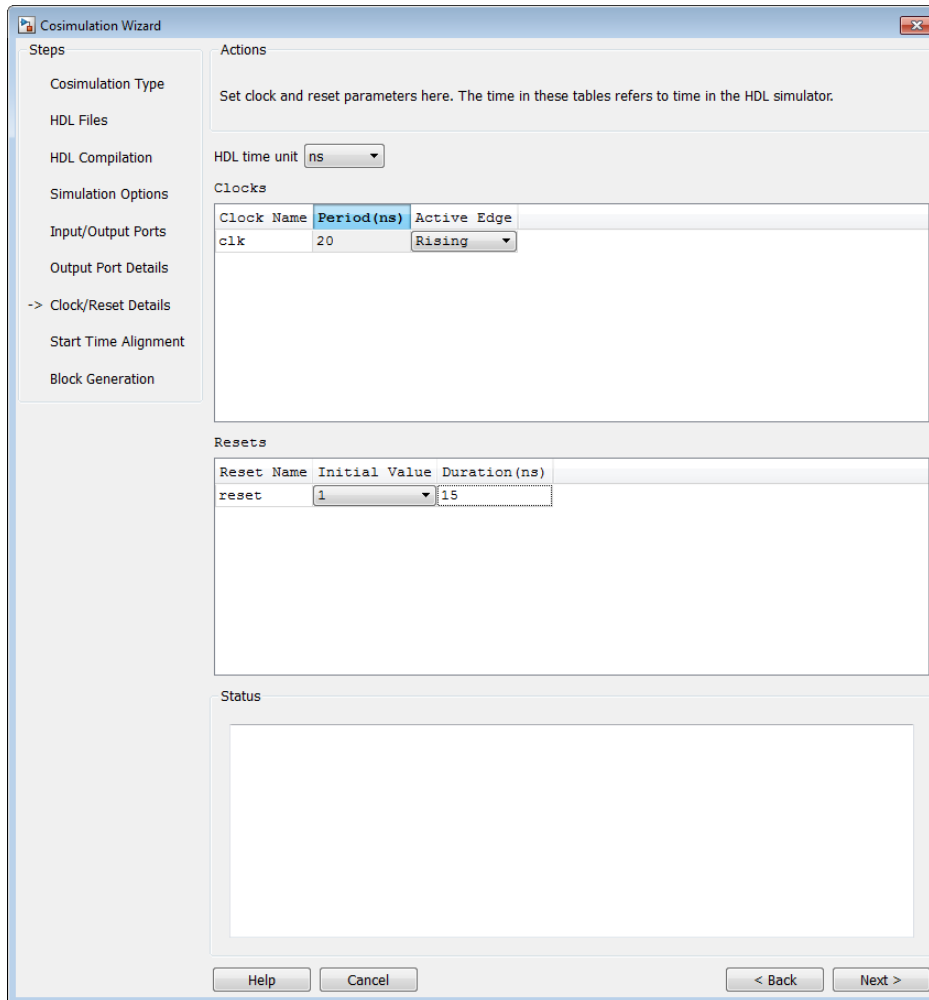
Tutorial: Set Clock and Reset Details

For this tutorial, set the clock **Period (ns)** to 20. From the Verilog code, you know that the reset is synchronous and the active value is 1. You can reset the entire HDL design at time 1 ns, triggered by the rising edge of the clock. Use a duration of 15 ns for the reset signal.

In the Clock/Reset Details page, perform the following steps:

- 1 Set clock period to 20.
- 2 Leave or set active edge to **Rising**.
- 3 Leave or set reset initial value to 1.
- 4 Set reset signal duration to 15.

Your clock and reset are now the same as those same signals shown in the following figure.



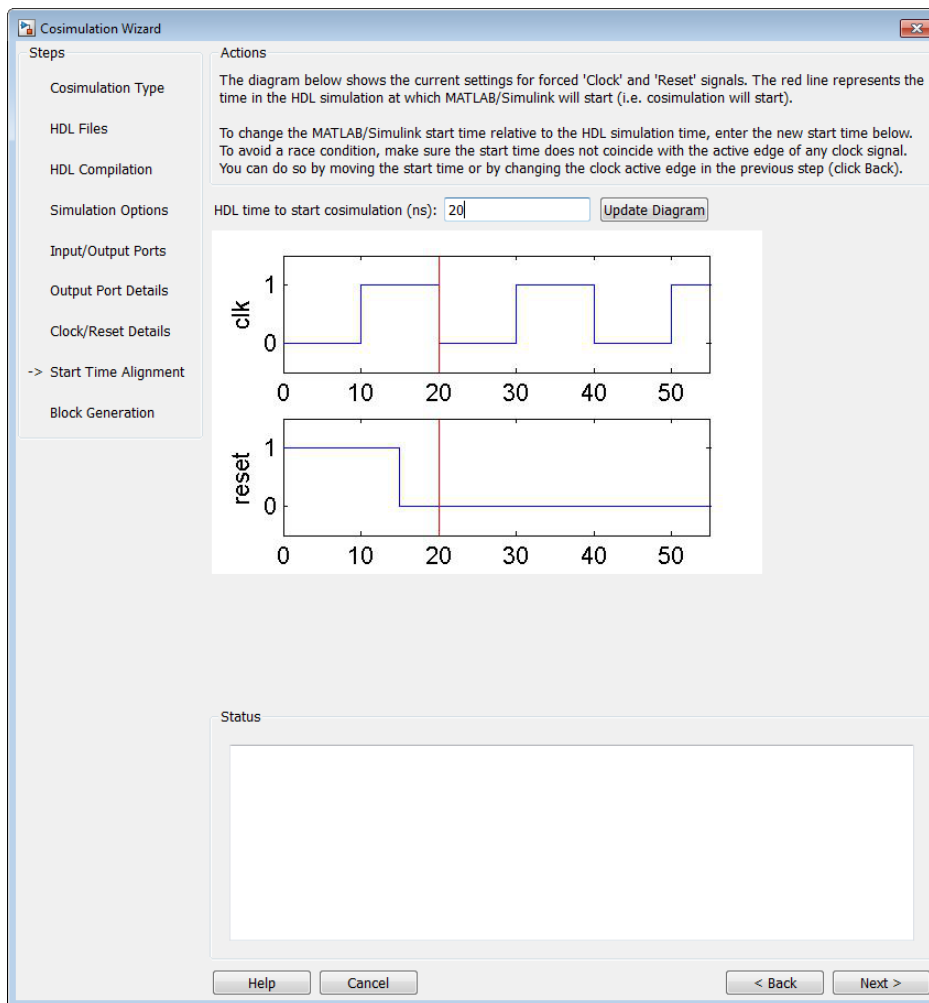
- 5 Click **Next** to proceed to the Start Time Alignment page.

Tutorial: Confirm Start Time Alignment

The Start Time Alignment page displays a plot for the waveforms of clock and reset signals. The Cosimulation Wizard shows the HDL time to start cosimulation with a red line. The start time is also the time at which the Simulink gets the first input sample from the HDL simulator.

- 1 Set or confirm Start Time Alignment

The active edge of our clock is a rising edge. Thus, at time 20 ns in the HDL simulator, the registered output of the raised cosine filter is stable. No race condition exists, and the default HDL time to start cosimulation (20 ns) is what we want for this simulation. You do not need to make any changes to the start time.

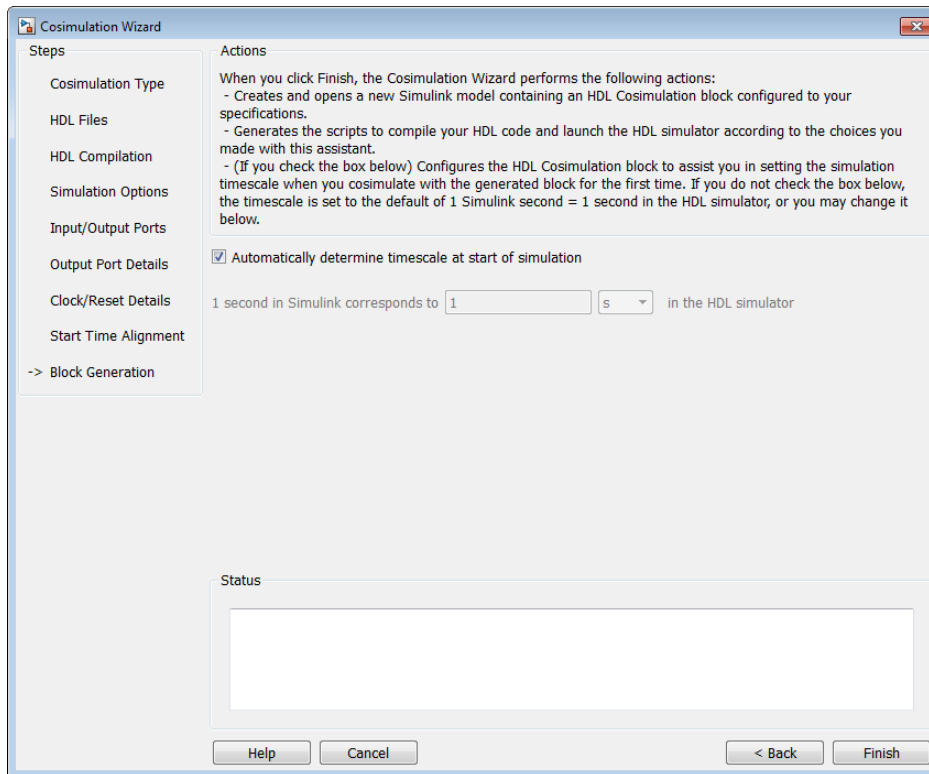


- 2 Click **Next** to proceed to Block Generation.

Tutorial: Generate Block

Before you generate the HDL Cosimulation block, you have the option to determine the timescale before you finish the Cosimulation Wizard. Alternately, you can instruct HDL Verifier to calculate a timescale later. Timescale calculation by the verification software occurs after you connect all the input/output ports of the generated HDL Cosimulation block and start simulation.

- 1 Leave **Automatically determine timescale at start of simulation** selected (default). Later, you will have the opportunity to view the calculated timescale and change that value before you begin simulation.

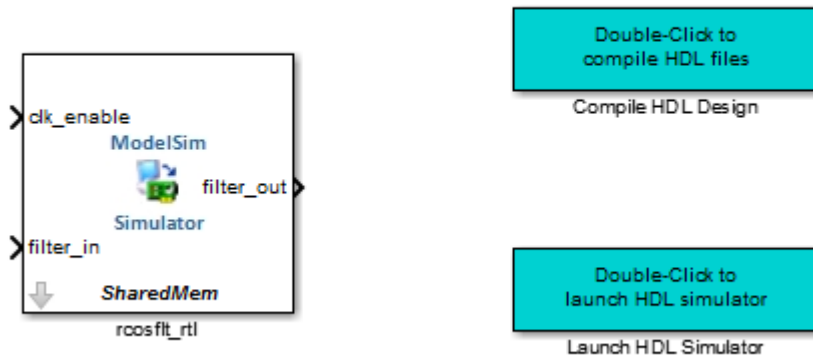


- 2 Click **Finish** to complete the Cosimulation Wizard session.

Tutorial: Create Test Bench to Verify HDL Design

For this tutorial, you do not actually create the test bench. Instead, you can find the finished model (`rcosflt_tb.mdl`) in the directory you created in “Tutorial: Set Up Tutorial Files (Simulink)” on page 9-51.

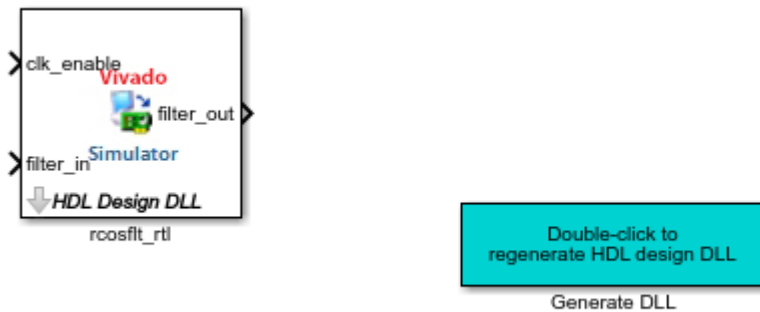
- 1 After you click **Finish**, Simulink creates a model and populates it with the following items:
 - An HDL Cosimulation block
 - A block to recompile the HDL design (contains a link to a script that is launched by double-clicking the block)
 - A block to launch the HDL simulator (contains a link to a script that is launched by double-clicking the block)



Leave the model for the moment and proceed to the next step.

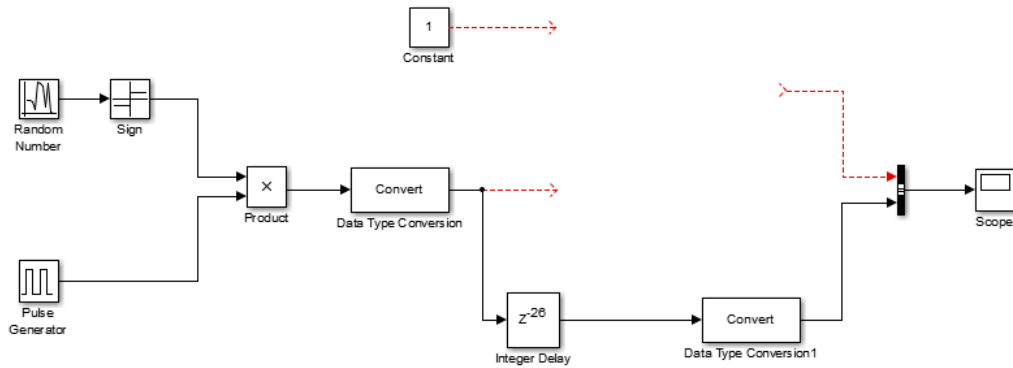
Vivado users: Simulink creates a model and populates it with the following items:

- An HDL Cosimulation block
- A block to regenerate the shared library (DLL file). The block contains a link to a script that is launched by double-clicking the block.

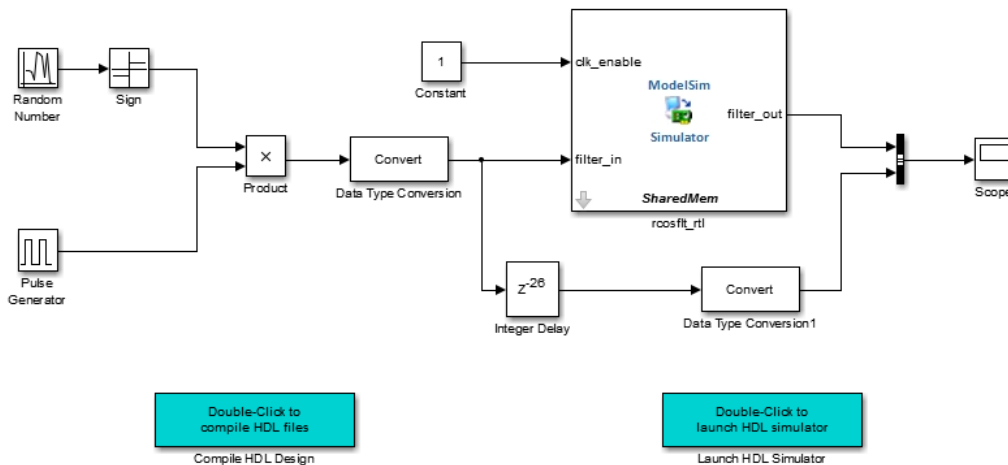


- 2 Open the file `rcosflt_tb`, located in the directory you created in "Tutorial: Set Up Tutorial Files (Simulink)" on page 9-51.

This file contains a model of a Simulink test bench. You will use this test bench to verify the HDL design for which you just generated a corresponding HDL Cosimulation block.



- 3 Add the HDL Cosimulation block to the test bench model as follows:
 - a Copy the HDL Cosimulation block from the newly generated model to this test bench model.
 - b Place the block so that the constant and convert blocks line up as inputs to the HDL Cosimulation block and the bus lines up as output.
 - c Connect the blocks in the test bench to the HDL Cosimulation block.
- 4 Copy the script blocks to the area below the test bench. Your model now looks similar to that in the following figure.



- 5 Save the model.

Tutorial: Run Cosimulation and Verify HDL Design

- 1 Launch the HDL simulator by double-clicking the block labeled **Launch HDL Simulator**.

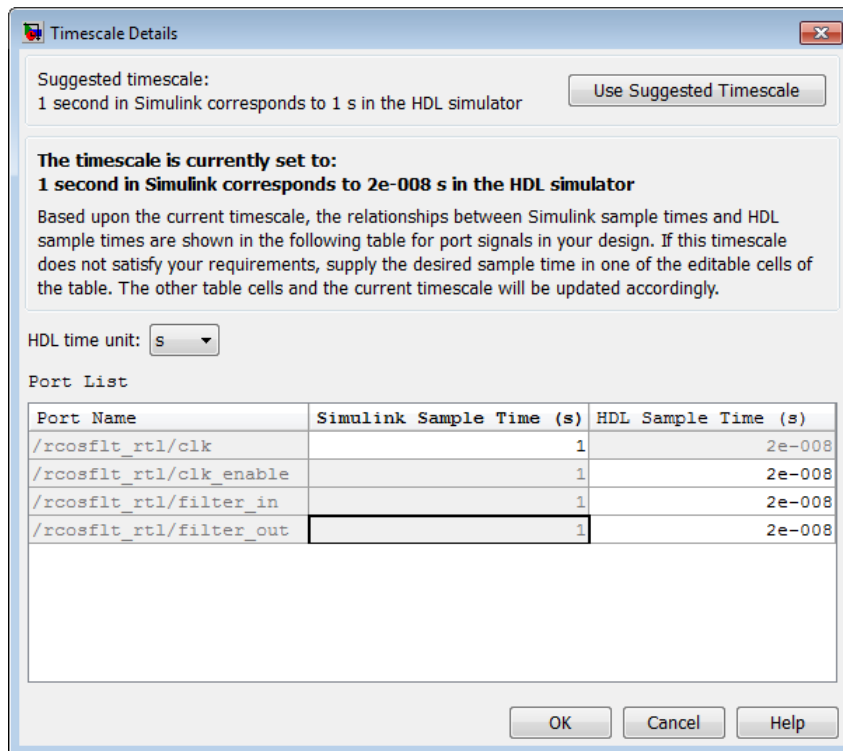
Vivado users: in the Simulink toolstrip, in the **Simulation** tab, click **Run** to start the cosimulation. The cosimulation will start, then stop, prompting you to determine timescale. Jump to step 3.

- 2 When the HDL simulator is ready, return to Simulink and start the simulation.
- 3 Determine timescale.

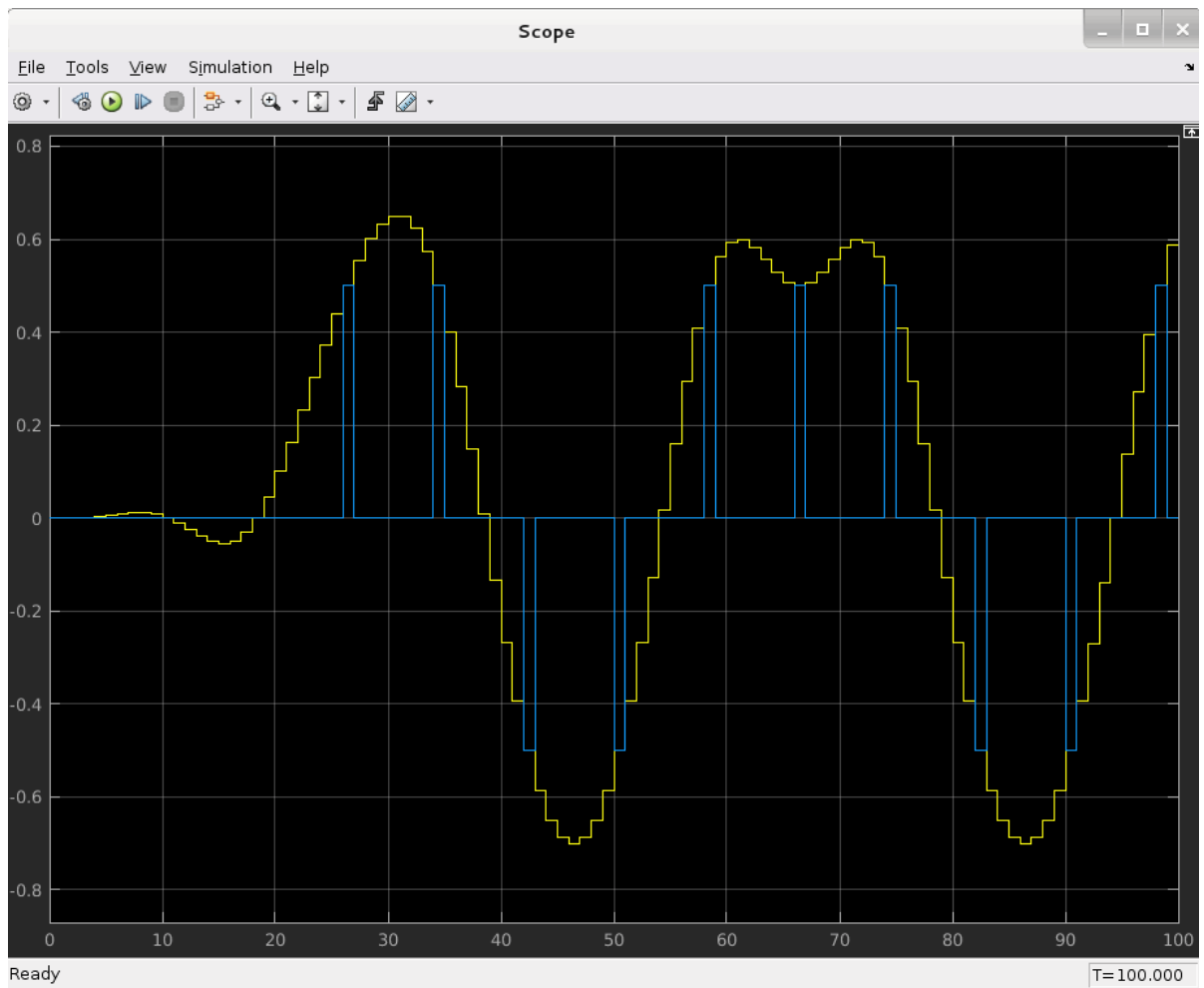
Recall that you selected **Automatically determine timescale at start of simulation** option on the last page of the Cosimulation Wizard. Because you did so, HDL Verifier launches the Timescale Details GUI instead of starting the simulation.

Both the HDL simulator and Simulink sample the `filter_in` and `filter_out` ports at 1 second. However, their sample time in the HDL simulator should be the same as the clock period (2 ns).

- a Change the Simulink sample time of `/rcosflt_rtl/clk` to 1 (seconds), and press **Enter**. The wizard then updates the table. The following figure shows the new timescale: 1 second in Simulink corresponds to 2e-008 s in the HDL simulator.



- b Click **OK** to exit Timescale Details.
- 4 Restart simulation.
 - 5 Verify the result from the scope in the test bench model. The scope displays both the delayed version of input to raised cosine filter and that filter's output. If you sample the output of this filter output directly, no inter-symbol-interference occurs



This step concludes the Cosimulation Wizard for use with Simulink tutorial.

Help Button

In this section...

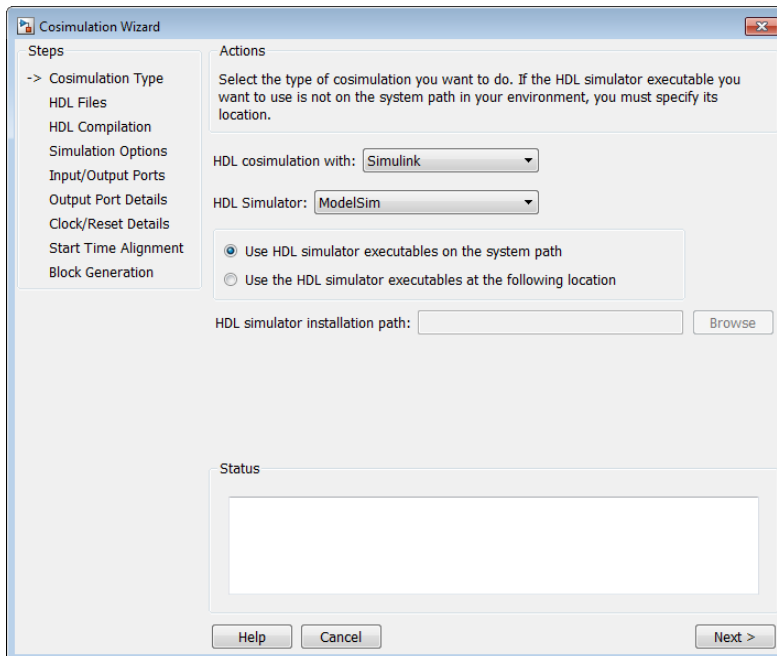
“Cosimulation Type” on page 9-66

“HDL Files” on page 9-67

“HDL Compilation” on page 9-68

“HDL Modules” on page 9-69

Cosimulation Type



- 1 Select your HDL Cosimulation workflow in the field **HDL cosimulation with:** Simulink, MATLAB, or MATLAB System Object. This setting instructs the wizard to create a block, function template, or System object, respectively.
- 2 Select the HDL simulator you want to use: ModelSim or Xcelium.
- 3 Select **Use HDL simulator executables on the system path** if that is where the files are located. The Cosimulation Wizard assumes by default that they are on the system path.

If the HDL simulator executables are *not* on the system path, select **Use the following HDL simulator executables at the following location** and specify the folder location in the text box below.

If you click **Next** and the Cosimulation Wizard does not find the executables, the following occurs:

- You are returned to this dialog and the Cosimulation Wizard displays an error in the status pane.
- The Cosimulation Wizard switches the option to **Use the following HDL simulator executables at the following location**.

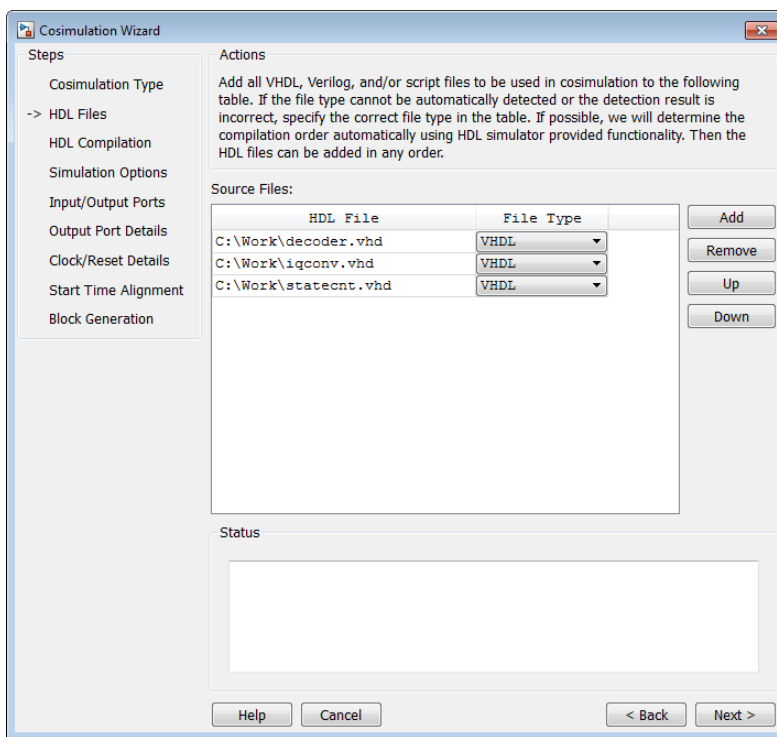
- The Cosimulation Wizard makes the HDL simulation path field editable.

You must enter a valid path to the HDL simulator executables before you are allowed to continue.

Next Steps

- For an HDL cosimulation block, start at “Cosimulation Type—Simulink Block” on page 9-24.
- For an HDL cosimulation function, start at “Cosimulation Type—MATLAB Function” on page 9-4.
- For an HDL cosimulation System object, start at “Cosimulation Type—MATLAB System Object” on page 9-12.

HDL Files



In the **HDL Files** pane, specify the files to be used in creating the function or block.

- 1 Click **Add** to select one or more file names.

The Cosimulation Wizard attempts to determine the file type of each file and display the type in the **File List** next to the file name. If the Cosimulation Wizard cannot determine the type or displays the wrong type, you can change the type directly in the **File Type** column.

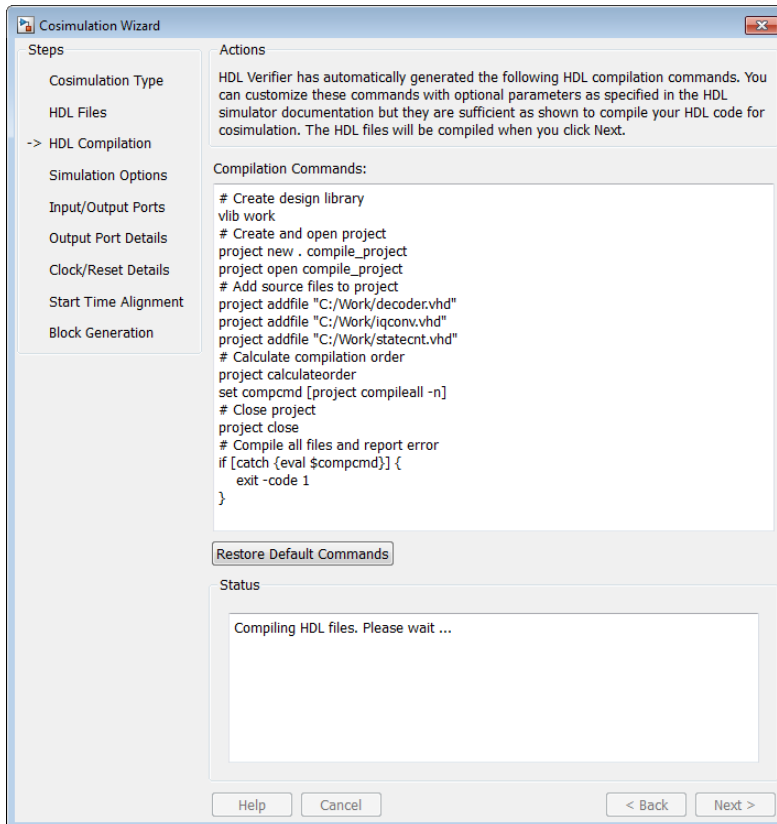
If you are using ModelSim, you will see compilation scripts listed as .do files (ModelSim macro file). If you are using Xcelium, you will see compilation scripts listed as system scripts.

- 2 Remove files by first highlighting the file name in the **File List**, then clicking **Remove Selected File**.

Next Steps

- For an HDL cosimulation block, start at “HDL Files—Simulink Block” on page 9-25.
- For an HDL cosimulation function, start at “HDL Files—MATLAB Function” on page 9-5.
- For an HDL cosimulation System object, start at “HDL Files—MATLAB System Object” on page 9-13.

HDL Compilation



In the **HDL Compilation** pane, you can review the generated HDL compilation commands. You may override and/or customize those commands, if you wish. If you included compilation scripts instead of HDL files, this pane will show you the command to run those scripts.

- 1 Enter any changes to the commands in the **Compilation Commands** box.

Note Do not include system shell commands; for example:

```
set file = a.vhd vcom $file
```

When control returns to the Cosimulation Wizard from executing the command, the variable no longer holds the value that was set. If you do try to include this type of command, you will see an error in the **Status** panel.

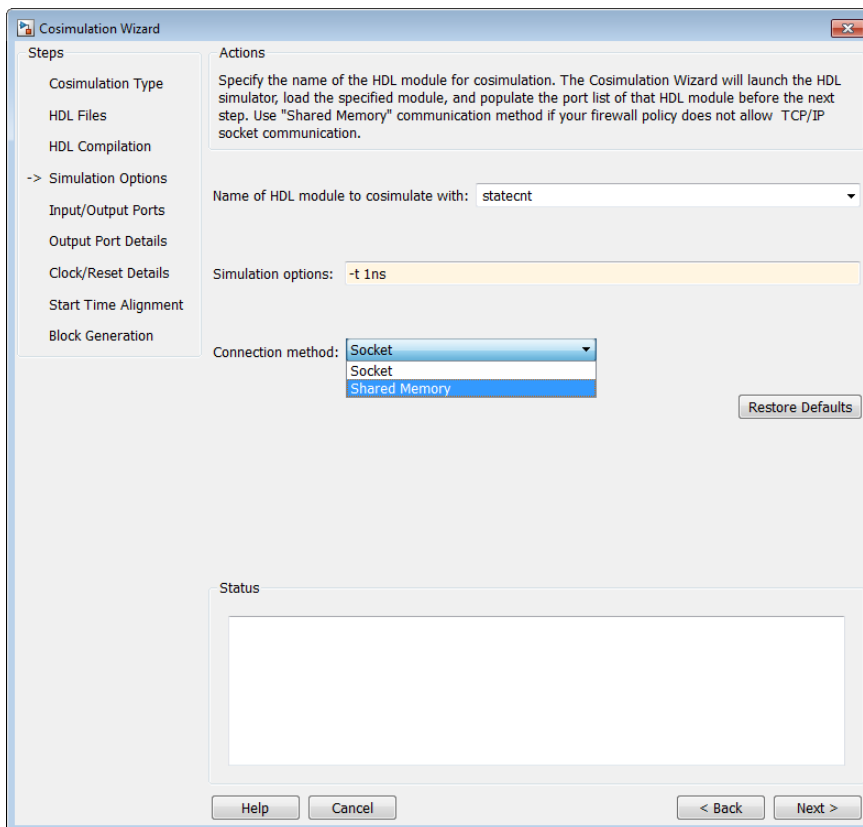
- 2 Click **Restore default commands** to go back to the generated HDL compilation commands. You are asked to confirm that you want to discard any changes.

Next Steps

- For an HDL cosimulation block, start at “HDL Compilation—Simulink Block” on page 9-26.
- For an HDL cosimulation function, start at “HDL Compilation—MATLAB Function” on page 9-6.
- For an HDL cosimulation System object, start at “HDL Compilation—MATLAB System Object” on page 9-14.

HDL Modules

HDL Modules—Simulink Block



In the **HDL Modules** pane, provide the name of the HDL module to be used in cosimulation.

- 1 Enter the name of the module at **Name of HDL module to cosimulate with**.
- 2 Specify additional simulation options at **Simulation options**. For example, in the previous image, the options shown are:
 - HDL simulator resolution
 - Turn off optimizations that remove signals from the simulation view

Click **Restore Defaults** to change the options back to the default.

- 3 When you proceed to the next step, the application performs the following actions in a command window:

- Starts the HDL simulator.
- Loads the HDL module in the HDL simulator.
- Starts the HDL server, and waits to receive notice that the server has started.
- Connects with the HDL server to get the port information.
- Disconnects and shuts down the HDL server.

Next Steps

- For an HDL cosimulation block, start at “Simulation Options—Simulink Block” on page 9-26.
- For an HDL cosimulation function, start at “HDL Modules—MATLAB Function” on page 9-7.
- For an HDL cosimulation System object, start at “Simulation Options—MATLAB System Object” on page 9-15.

HDL Cosimulation Guide

- “Set Up for HDL Cosimulation” on page 10-2
- “Cross-Network Cosimulation” on page 10-14
- “Test Bench and Component Function Writing” on page 10-19
- “Simulation Speed Improvement Tips” on page 10-26
- “Race Conditions in HDL Simulators” on page 10-33
- “Supported Data Types” on page 10-35
- “Simulation Timescales” on page 10-44
- “Clock, Reset, and Enable Signals” on page 10-55
- “TCP/IP Socket Ports” on page 10-61

Set Up for HDL Cosimulation

To cosimulate your HDL code with a MATLAB or Simulink design, you must first:

- Decide how to connect your HDL simulator with MATLAB or Simulink. You may have one or multiple HDL modules in a cosimulation setup. The modules are represented by `matlabcp` and `matlabtb` functions or `hdlcosimulation` system objects for MATLAB, or by HDL Cosimulation blocks for Simulink. See “Cosimulation Configurations” on page 10-2.
- Start the HDL simulator from MATLAB, or from a shell. You must use a shell for a cross-network simulation, such as if the HDL simulator runs on a different machine from your MATLAB host. Starting the simulator from MATLAB allows you to specify the library by name rather than exact path. See “HDL Simulator Startup” on page 10-5.
- If you require a non-default library or customized library location, specify the library when you start the HDL simulator. If you start the HDL simulator from MATLAB, use the name of the library. If you start the HDL simulator from a shell, use the library path. See “Cosimulation Libraries” on page 10-9.
- Optionally, use the configuration and diagnostic script to configure your library location and test the TCP/IP connection. This script is supported only for Linux machines. For Windows machines, you can create a configuration file. See “HDL Simulator Startup” on page 10-5.

Cosimulation Configurations

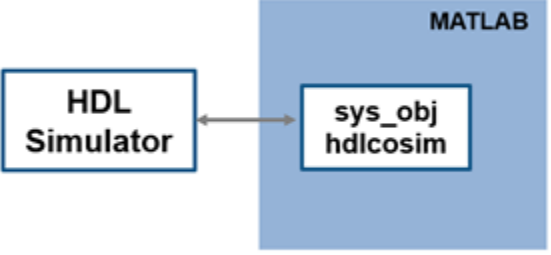
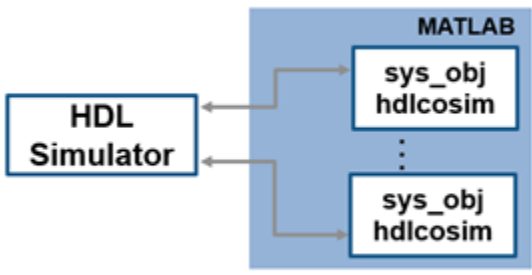
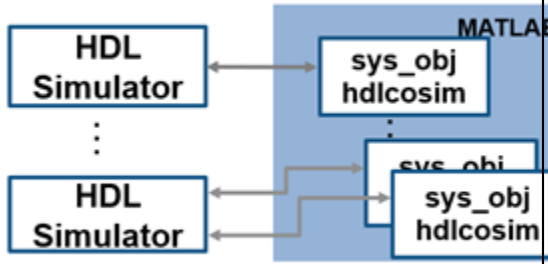
There are several ways to connect your HDL simulator to a design in MATLAB or Simulink. You can run your HDL simulator on the same or different host machine as MATLAB. Each HDL simulator can connect to one or more functions in MATLAB, or one or more HDL Cosimulation blocks in a Simulink model. In a network configuration, to identify your application servers, use an Internet address and a TCP/IP socket port.

Note

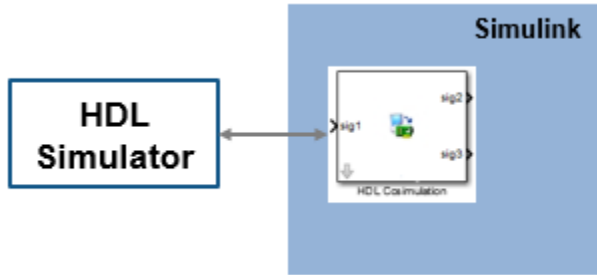
- Vivado cosimulation is supported via one HDL Cosimulation block in Simulink or one `VivadoHDLCosimulation` system object in MATLAB connected to the HDL simulator.
 - An instance of MATLAB can run only one instance of the MATLAB server (`hdldaemon`) at a time.
 - Each HDL simulator must communicate with a unique instance of the MATLAB server.
 - Shared memory communication is an option for configurations that require only one communication link on a single computing system.
 - TCP/IP socket communication is required for configurations that use multiple communication links on one or more computing systems. Unique TCP/IP socket ports distinguish the communication links.
-

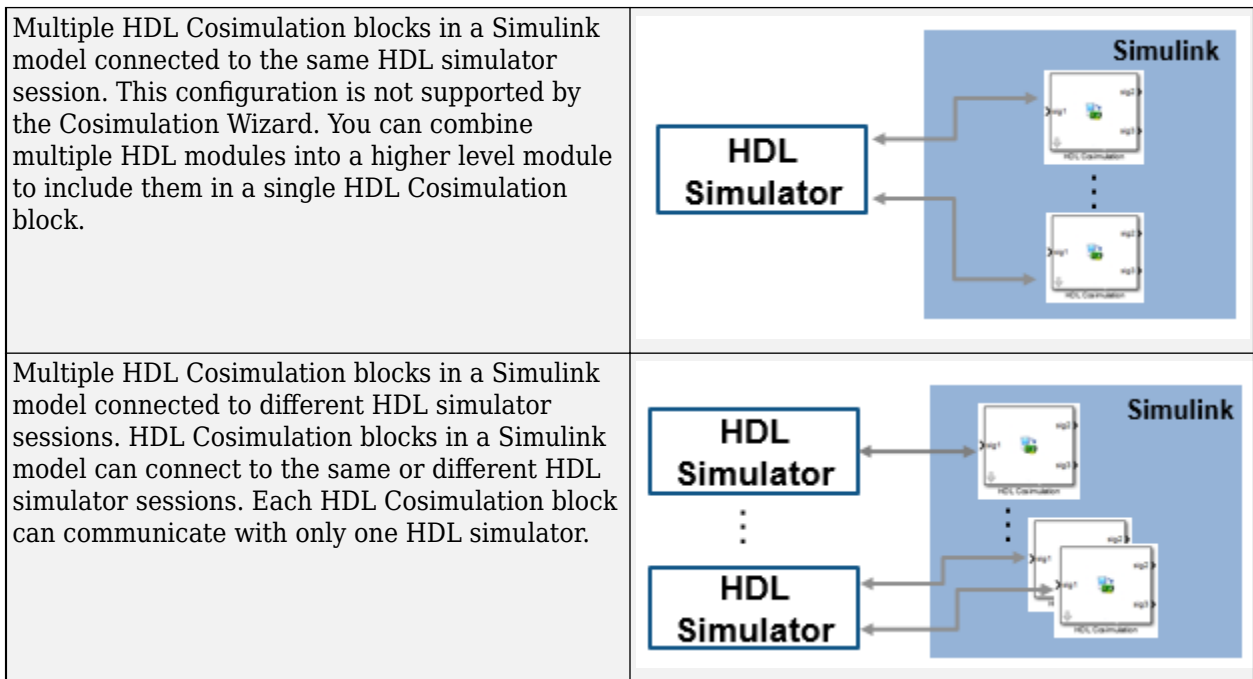
Valid Configurations for HDL Cosimulation with MATLAB

<p>An HDL simulator session connected to a MATLAB function through a single instance of the MATLAB server.</p>	<p>The diagram shows a box labeled 'HDL Simulator' on the left. A double-headed arrow connects it to a box labeled 'Server' on the right. The 'Server' and a box labeled 'Function foo()' are both enclosed within a larger blue-shaded box labeled 'MATLAB'. A double-headed arrow connects the 'Server' to the 'Function foo()'.</p>
<p>An HDL simulator session connected to multiple MATLAB functions through a single instance of the MATLAB server.</p>	<p>The diagram shows a box labeled 'HDL Simulator' on the left. A double-headed arrow connects it to a box labeled 'Server' on the right. The 'Server' and two boxes labeled 'foo()' and 'bar()' are all enclosed within a larger blue-shaded box labeled 'MATLAB'. Double-headed arrows connect the 'Server' to both 'foo()' and 'bar()'.</p>
<p>An HDL simulator session connected to a MATLAB function through multiple instances of the MATLAB server. Each instance runs within the scope of a unique MATLAB session. This configuration is not supported by the Cosimulation Wizard.</p>	<p>The diagram shows a box labeled 'HDL Simulator' on the left. Two arrows point from it to two separate blue-shaded boxes. The top box is labeled 'MATLAB 1' and contains a 'Server' box connected to a 'foo()' box. The bottom box is labeled 'MATLAB N' and contains a 'Server' box connected to a 'foo()' box. Vertical ellipses between the MATLAB boxes indicate multiple sessions.</p>
<p>Multiple HDL simulator sessions, each connected to a MATLAB function through multiple instances of the MATLAB server. Each instance runs within the scope of a unique MATLAB session.</p>	<p>The diagram shows two separate blue-shaded boxes. The top box is labeled 'MATLAB 1' and contains an 'HDL Simulator 1' box connected to a 'Server' box, which is connected to a 'foo()' box. The bottom box is labeled 'MATLAB N' and contains an 'HDL Simulator N' box connected to a 'Server' box, which is connected to a 'foo()' box. Vertical ellipses between the HDL simulator boxes and between the MATLAB boxes indicate multiple sessions.</p>

<p>An <code>hdlcosim</code> System object in a MATLAB session connected to a single HDL simulator session.</p> <p>Note This configuration is supported for Vivado cosimulation.</p>	 <p>The diagram shows a blue box labeled 'MATLAB' containing a white box labeled 'sys_obj_hdlcosim'. To the left of the MATLAB box is a white box labeled 'HDL Simulator'. A double-headed arrow connects the 'HDL Simulator' box to the 'sys_obj_hdlcosim' box.</p>
<p>Multiple <code>hdlcosim</code> system objects in a MATLAB session connected to the same HDL simulator session. This configuration is not supported by the Cosimulation Wizard. You can combine multiple HDL modules into a higher level module to include them in a single HDL Cosimulation System object.</p>	 <p>The diagram shows a blue box labeled 'MATLAB' containing two white boxes, both labeled 'sys_obj_hdlcosim', with vertical ellipsis between them. To the left of the MATLAB box is a white box labeled 'HDL Simulator'. Two arrows point from the 'HDL Simulator' box to each of the 'sys_obj_hdlcosim' boxes.</p>
<p>Multiple <code>hdlcosim</code> System objects in a MATLAB session connected to different HDL simulator sessions. <code>hdlcosim</code> System objects in a MATLAB session can connect to the same or different HDL simulator sessions. Each <code>hdlcosim</code> System object can communicate with only one HDL simulator.</p>	 <p>The diagram shows a blue box labeled 'MATLAB' containing two white boxes, both labeled 'sys_obj_hdlcosim', with vertical ellipsis between them. To the left of the MATLAB box are two white boxes, both labeled 'HDL Simulator', with vertical ellipsis between them. Arrows point from each 'HDL Simulator' box to one of the 'sys_obj_hdlcosim' boxes.</p>

Valid Configurations for Cosimulation with Simulink

<p>An HDL Cosimulation block in a Simulink model connected to a single HDL simulator session.</p> <p>Note This configuration is supported for Vivado cosimulation.</p>	 <p>The diagram shows a blue box labeled 'Simulink' containing a white box labeled 'HDL Cosimulation'. The 'HDL Cosimulation' block has three input ports labeled 'Hg1', 'Hg2', and 'Hg3'. To the left of the Simulink box is a white box labeled 'HDL Simulator'. A double-headed arrow connects the 'HDL Simulator' box to the 'HDL Cosimulation' block.</p>
---	--



HDL Simulator Startup

You can start the HDL simulator from MATLAB, or from a shell. You must use a shell for a cross-network simulation, such as if the HDL simulator runs on a different machine from your MATLAB host. Starting the simulator from MATLAB allows you to specify the library by name rather than exact path.

Vivado users are not required to start the HDL simulator separately from MATLAB or Simulink, since Vivado cosimulation is executed as one process with a shared DLL.

Set up for Cosimulation with Vivado

When using the Vivado simulator you must first use the **Cosimulation Wizard** or **HDL Workflow Advisor** to create an HDL Cosimulation block or an `hdlverifier.VivadoHDLCosimulation` System object.

Use the **Cosimulation Wizard** to generate cosimulation artifacts when you already have an HDL DUT. Use the **HDL Workflow Advisor** to generate an HDL DUT (requires HDL Coder), in addition to cosimulation artifacts.

For Vivado setup on Windows, you must set the following variables. If you are using the **Cosimulation Wizard**, it ensures you set these variables correctly:

- 1 `%PATH%` must point to Vivado executables (such as `vivado`, `xelab`, `xvlog`, and `xvhdl`).
- 2 `%PATH%` must point to Vivado libraries (`%XILINX_VIVADO%\lib\win64.o`).

For Vivado setup on Linux, you must set these variables before starting MATLAB:

- 1 `$PATH` must point to Vivado executables (the **Cosimulation Wizard** ensures that you set it properly).

- 2 `$LD_LIBRARY_PATH` must point to Vivado libraries (`$XILINX_VIVADO/lib/lnx64.o`). This must be done **before** starting MATLAB.

To set `$LD_LIBRARY_PATH` on Linux, perform the following in a `cs` shell prompt:

```
cs> setenv LD_LIBRARY_PATH /tools/Vivado/2020.2-mw-0/Lin/Vivado/2020.2/lib/lnx64.o:$LD_LIBRARY_PATH
cs> matlab
```

For other shell types, enter the following:

```
sh> export LD_LIBRARY_PATH=/tools/Vivado/2020.2-mw-0/Lin/Vivado/2020.2/lib/lnx64.o:$LD_LIBRARY_PATH
sh> matlab
```

Alternatively, create a `/etc/ld.so.conf.d/xilvivo.conf` file and then invoke it using `ldconfig` (requires root privileges). File contents:

```
# to automatically find Xilinx Vivado libraries from MATLAB applications such as HDL Cosimulation using HDL Verifier.
# (despite the name, this is a folder that holds linux *libraries*)
# reload updates to this file via: sudo ldconfig
/tools/Vivado/2020.2-mw-0/Lin/Vivado/2020.2/lib/lnx64.o
```

Then invoke it by executing:

```
sh> sudo ldconfig
```

Start the HDL Simulator from MATLAB

Each supported HDL simulator has a unique command that opens it from MATLAB.

Note If you use the Cosimulation Wizard, you do not need to start the HDL simulator separately.

HDL Simulator	Command to Open the Simulator	Example
Cadence Xcelium	<code>nclaunch</code>	“Start Cadence Xcelium from MATLAB” on page 10-7
Mentor Graphics ModelSim	<code>vsim</code>	“Start Mentor Graphics ModelSim from MATLAB” on page 10-7

In either function, you can specify the HDL Verifier library, the design to load, the type of communication connection information and other required parameters as name-value pair arguments. No special setup is required. See “Cosimulation Libraries” on page 10-9.

This function starts and configures the HDL simulator for use with the HDL Verifier software. By default, the function starts the first version of the simulator executable that it finds on the system path, as defined by the `path` variable. This function uses a temporary file that is overwritten each time the HDL simulator starts.

You can customize the startup file and communication mode to be used between MATLAB or Simulink and the HDL simulator by specifying name-value pairs when you call the function. For property details, see `nclaunch` or `vsim`.

To start a different version of the simulator executable than the first one found on the system path, use the `setenv` and `getenv` MATLAB functions to set and get the environment of any subshells spawned by `UNIX()`, `DOS()`, or `system()`.

If you specify a communication mode when you call one of the functions that open the HDL simulator, the specified mode applies to all HDL simulator sessions connected to either MATLAB or Simulink.

For more information on how HDL Verifier links the HDL simulator with MATLAB, see “Linking with MATLAB and the HDL Simulator”.

For a full cosimulation example that demonstrates starting the HDL simulator from MATLAB, see “Verify HDL Module with MATLAB Test Bench” on page 2-18.

Start Cadence Xcelium from MATLAB

To start the Cadence Xcelium simulator from MATLAB, at the MATLAB command prompt, enter:

```
nclaunch('PropertyName', 'PropertyValue', ...)
```

This example changes the folder location to VHDLproj and then opens Xcelium. Because the command line omits the 'hdlmdir' and 'startupfile' properties, nclaunch creates a temporary file. The 'tclstart' property specifies Tcl commands that load and initialize the HDL simulator for test bench instance modsimrand.

```
cd VHDLproj
nclaunch('tclstart',...
'hdlsimmatlab modsimrand; matlabtb modsimrand 10 ns -socket 4449')
```

This example changes the folder location to VHDLproj and then opens Xcelium. Because the function call omits the 'hdlmdir' and 'startupfile' properties, nclaunch creates a temporary file. The 'tclstart' property specifies a Tcl command that loads the VHDL entity parse in library work for cosimulation between nclaunch and Simulink. The 'socketsimulink' property specifies TCP/IP socket communication on the same computer, using port 4449.

```
cd VHDLproj
nclaunch('tclstart', 'hdlsimulink work.parse', 'socketsimulink', '4449')
```

Start Mentor Graphics ModelSim from MATLAB

To start the Mentor Graphics ModelSim HDL simulator from MATLAB, at the MATLAB command prompt, enter:

```
vsim('PropertyName', 'PropertyValue', ...)
```

This example changes the folder location to VHDLproj and then opens ModelSim. Because the vsim call omits the 'vsimdir' and 'startupfile' properties, the function creates a temporary DO file. The 'tclstart' property specifies Tcl commands that load and initialize the HDL simulator for test bench instance modsimrand.

```
cd VHDLproj
vsim('tclstart', 'vsimmatlab modsimrand; matlabtb modsimrand 10 ns -socket 4449')
```

This example changes the folder location to VHDLproj and then opens ModelSim. Because the vsim call omits the 'vsimdir' and 'startupfile' properties, vsim creates a temporary DO file. The 'tclstart' property specifies a Tcl command that loads the VHDL entity parse in library work for cosimulation between vsim and Simulink. The 'socketsimulink' property specifies TCP/IP socket communication on the same computer, using socket port 4449.

```
cd VHDLproj
vsim('tclstart', 'vsimulink work.parse', 'socketsimulink', '4449')
```

This example includes Tcl commands that run HDL compilation and simulation when the ModelSim software starts up.

```
vsim('tclstart', {'vlib work', 'vlog +acc clocked_inverter.v hdl_top.v', 'vsim +acc hdl_top'});
```

This example loads the hdl_top module for Simulink cosimulation. The vsimulink command also specifies socket number 5678 for communication with HDL Cosimulation blocks in Simulink models,

and specifies an HDL time precision of 10 ps. Specifying the socket this way is equivalent to using the `socketsimulink` property of the `vsim` function.

```
vsim('tclstart', ...
     {'vlib work', 'vlog -voptargs=+acc clocked_inverter.v hdl_top.v', ...
     'vsimulink hdl_top -socket 5678 -t 10ps'});
```

Start the HDL Simulator from a Shell

When you start the HDL simulator from a shell, you must first create a Tcl setup file for the HDL simulator. The setup file includes the location of the specified cosimulation libraries for MATLAB and Simulink. You can then include this file when you start your HDL simulator.

For more information on cosimulation libraries, see “Cosimulation Libraries” on page 10-9.

After you have created your Tcl setup file, you can edit it to add compilation and execution commands.

Start Cadence Xcelium from a Shell

Generate a setup file by using the `nclaunch` function at the MATLAB prompt.

```
nclaunch('starthdlsim','no','startupfile','cosim_procdefs.tcl','tclstart',{''})
```

This creates a Tcl setup file named `cosim_procdefs.tcl`, which contains definitions for functions that enable cosimulation with MATLAB or Simulink.

Now, load this generated setup file when you start Xcelium. In a shell prompt, enter the following.

```
xmsim design_name -64bit -input setupfile
```

Where *design_name* is the name of your top level design and *setupfile* is the name of the Tcl setup file (`cosim_procdefs.tcl` in this example, or `compile_and_launch.tcl` by default). For an example that uses a Tcl setup file, see “Cosimulation for Testing Filter Component Using MATLAB Test Bench” on page 32-163.

Once the simulator is launched, use the `hdlsimmatlab` function to load the HDL module, and the `matlabtb` or `matlabcp` function to connect to a MATLAB function. For example, to compile and launch Xcelium enter the following commands in a shell (assuming the Tcl script is named `cosim_procdefs.tcl`).

```
xmvlog -64bit cosim_lowpass_filter.v
xmclab -64bit -access +wc cosim_lowpass_filter
xmsim cosim_lowpass_filter -64bit -input cosim_procdefs.tcl
```

Then in Xcelium enter:

```
matlabtb cosim_lowpass_filter 10ns -repeat 10ns -mfunc cosim_lowpass_filtertb
force cosim_lowpass_filter.clk_enable 1 -after 0ns
force cosim_lowpass_filter.reset 1 -after 0ns 0 -after 22ns
force cosim_lowpass_filter.clk 1 -after 0ns 0 -after 5ns -repeat 10ns
deposit cosim_lowpass_filter.filter_in 0
run 10000ns
```

Start Mentor Graphics ModelSim from a Shell

Generate a setup file by using the `vsim` function at the MATLAB prompt.

```
vsim('startms','no','startupfile','cosim_procdefs.tcl')
```

This creates a Tcl setup file named `cosim_procdefs.tcl`, which contains definitions for functions that enable cosimulation with MATLAB or Simulink.

Now, use this generated configuration file when you “Start the HDL Simulator from a Shell” on page 10-8.

Now, load this generated setup file when you start ModelSim. In a shell prompt, enter the following.

```
vsim design_name -do setupfile
```

`setupfile` is the name of the Tcl setup file (`cosim_procdefs.tcl` in this example, or `compile_and_launch.tcl` by default). When you include the `design_name` argument, the `vsim` call also starts the simulation.

The configuration file defines the `-foreign` option to `vsim`. This option loads the HDL Verifier shared library and specifies its entry point. You can also specify any other existing configuration files you are using.

If you do not use the generated config file, you can load the client shared library and specify its entry point by executing `vsim` with a command like this:

```
vsim design_name -foreign matlabclient /path/library
```

where `path` is the path to the HDL Verifier cosimulation library. See “Cosimulation Libraries” on page 10-9 to find the applicable library name for your machine. For an example that uses a Tcl configuration file, see “Cosimulation for Testing Filter Component Using MATLAB Test Bench” on page 32-163.

Note You can also call this command from inside the HDL simulator.

Once the simulator is launched, use the `vsimmatlab` function to load the HDL module, and the `matlabtb` or `matlabcp` function to connect to a MATLAB function. For example, to compile and launch ModelSim enter the following commands in a shell (assuming the Tcl script is named `cosim_procdefs.tcl`):

```
vlib work
vlog cosim_lowpass_filter.v
vsim -c -do cosim_procdefs.tcl
```

Then in ModelSim enter:

```
vsimmatlab work.cosim_lowpass_filter
matlabtb cosim_lowpass_filter 10ns -repeat 10ns -mfunc cosim_lowpass_filtertb.m
force clk_enable 1 0
force reset 1 0, 0 25
force clk 1 0, 0 5 -repeat 10
run 10000ns
```

Cosimulation Libraries

It is recommended to use the same compiler for all libraries linked into the same executable. HDL Verifier versions of the library for the compilers that the HDL simulators support. Using the same libraries helps the cosimulation software stay compatible with other C++ libraries that you might link into the HDL simulator, including SystemC libraries.

If any of these conditions apply, choose the version of the HDL Verifier library that matches the compiler used for that code:

- You link other third-party applications into your HDL simulator.
- You compile and link SystemC code as part of your design or test bench.
- You link custom C/C++ applications into your HDL simulator.

If you do not link any other code into your HDL simulator, you can use any version of the supplied libraries. The function for opening the HDL simulator (`nclaunch` or `vsim`) chooses a default version of this library.

For examples on specifying HDL Verifier libraries when cosimulating across a network, see “Cross-Network Cosimulation” on page 10-14.

Library Naming Format

The HDL Verifier cosimulation libraries use the following naming format:

```
edalink/extensions/{version}/{arch}/lib{version_short_name}{client_server_tag}_{compiler_tag}.{libext}
```

Argument	Xcelium Values	ModelSim/Questa Values
version	incisive	modelsim
arch	linux64	linux64, windows32, or windows64
version_short_name	lfihdl	lfmhdl
client_server_tag	MATLAB: c Simulink: s	MATLAB: c Simulink: s
compiler_tag	gcc48, gcc63, tmwgcc	Linux: gcc474, gcc530, gcc740, tmwgcc Windows 32: gcc421vc12 Windows 64: gcc740vc15, tmwvs Note gcc740vc15 or gcc421vc12 requires Visual Studio® 2013 redistribute, available from Microsoft®.
libext	so	dll or so

For more on MATLAB build compilers, see MATLAB Build Compilers.

Default Libraries

The HDL Verifier scripts support the use of libraries compiled with versions of GCC supplied by the HDL tool vendors. The table lists the libraries shipped with HDL Verifier for each supported HDL simulator.

The default library is `tmwgcc` or `tmwvs` versions. Specify an alternative with the `libfile` argument to `vsim` or `nclaunch`.

Cadence Xcelium Libraries

Platform	MATLAB Library	Simulink Library
Linux 64	liblfihdlc_tmwgcc.so (default) liblfihdlc_gcc48.so liblfihdlc_gcc63.so	liblfihdlc_tmwgcc.so (default) liblfihdlc_gcc48.so liblfihdlc_gcc63.so

Mentor Graphics ModelSim and Questa Libraries

Platform	MATLAB Library	Simulink Library
Linux 64	liblfmhdlc_tmwgcc.so (default) liblfmhdlc_gcc474.so liblfmhdlc_gcc530.so liblfmhdlc_gcc740.so	liblfmhdlc_tmwgcc.so (default) liblfmhdlc_gcc474.so liblfmhdlc_gcc530.so liblfmhdlc_gcc740.so
Windows 32	liblfmhdlc_gcc421vc12.dll	liblfmhdlc_gcc421vc12.dll
Windows 64	liblfmhdlc_tmwvs.dll (default) liblfmhdlc_gcc740vc15.dll	liblfmhdlc_tmwvs.dll (default) liblfmhdlc_gcc740vc15.dll

Alternative HDL Simulator Libraries

You can use a different HDL-side library by specifying the `libfile` name-value pair when you call the `nclaunch` or `vsim` function. Choose the version of the library that matches the compiler and system libraries you are using for any other C/C++ libraries linked into the HDL simulator. Depending on the version of your HDL simulator, you might need to explicitly set additional paths in the `LD_LIBRARY_PATH` environment variable.

For example, to use a nondefault library:

- 1 Copy the system libraries from the MATLAB installation to the machine with the HDL simulator. The system libraries are installed in `matlabroot/sys/os`.
- 2 Modify the `LD_LIBRARY_PATH` environment variable to add the path to the copied system libraries.

Specify Alternate Library Using `nclaunch`

This example shows library settings for an HDL simulation that links in a custom C++ application, compiled with `gcc44`. Therefore, the simulator must use the cosimulation libraries compiled with `gcc44`, instead of using the default library. Both MATLAB and Xcelium are running on the same 64-bit Linux machine.

Modify the `PATH` variable so that the `nclaunch` function finds the desired version of the HDL simulator. Then, specify the library name with the `libfile` name-value pair. At the MATLAB command prompt, type:

```
currPath = getenv('PATH');
setenv('PATH',[ '/tools/IUS-1110/bin:' currPath]);
nclaunch('tclstart',{'exec xmvhdl -64bit inverter.vhd', ...
    'exec xmelab -64bit -access +rwc inverter', ...
    'hdlsimulink -gui inverter' }, ...
    'libfile','liblfihdlc_gcc48');
```

Verify the library resolution using `ldd` from within the `xmsim` console.

```
exec ldd /path/to/matlab/toolbox/edalink/extensions/incisive/linux64/liblfihdlc_gcc48.so
linux-vdso.so.1 => (0x00007fff2ffff000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f98361a0000)
libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f9835e99000)
```

```
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f9835c16000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f9835a00000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9835676000)
/lib64/ld-linux-x86-64.so.2 (0x00007f983661c000)
```

Specify Alternate Library for Xcelium Using System Shell

This example shows how to start Xcelium with an explicit option to specify the cosimulation library. You can start Xcelium from a system shell on the same machine as MATLAB, on a different machine, or on a machine with a different operating system.

This example code runs on a 64-bit Linux version of Xcelium. It does not matter what machine MATLAB is running on. Instead of using the default library in the Xcelium distribution, this example uses the library compiled with GCC 4.4.

Modify the PATH variable to point to the desired version of the HDL simulator. Although `xmsim` finds any GCC libraries in the installation, this example changes the `LD_LIBRARY_PATH` to show how to use a custom installation of GCC. In a csh-compatible system shell, enter:

```
setenv PATH /tools/ius/lrx/tools/bin/64bit:${PATH}
setenv LD_LIBRARY_PATH /tools/ius/lrx/tools/systemc/gcc/4.4-x86_64/install/lib64:${LD_LIBRARY_PATH}
xmvhdl -64bit inverter.vhd
xmehlab -64bit -access +rwc inverter
xmsim -tcl -loadvpi /tools/matlab/toolbox/edalink/extensions/incisive/linux64/liblfihdlc_gcc48:matlabclient inverter.vhd
```

You can check the library resolution using `ldd`, as in the previous example.

Specify Alternate Library Using vsim

This example shows library settings for an HDL simulation that uses some SystemC applications, compiled with `gcc450`. You can download this version of GCC with its associated system libraries from Mentor Graphics. Therefore, the simulator must use the cosimulation libraries compiled with `gcc450`, instead of using the default library. Both MATLAB and ModelSim are running on the same 64-bit Linux machine.

Modify the PATH variable so that the `vsim` function finds the desired version of the HDL simulator. Modify the `LD_LIBRARY_PATH` because the HDL simulator does not add the path to the system libraries. Then, specify the library name with the `libfile` name-value pair. At the MATLAB command prompt, type:

```
currPath = getenv('PATH');
currLdPath = getenv('LD_LIBRARY_PATH');
setenv('PATH',['/tools/modelsim-10.1c/bin:' currPath]);
setenv('LD_LIBRARY_PATH',['/tools/modelsim-10.1c/gcc-4.5.0-linux/lib:' currLdPath]);
vsim('tclstart',{'vlib work','vcom inverter.vhd','vsimulink inverter'}, ...
    'libfile','liblfmhds_gcc450');
```

Verify the library resolution using `ldd` from within the `vsim` GUI.

```
exec ldd /path/to/matlab/toolbox/edalink/extensions/modelsim/linux64/liblfmhds_gcc450.so
linux-vdso.so.1 => (0x00007fff06652000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f505083d000)
libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f5050536000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f50502b3000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f505009d000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f504fd13000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5050cb8000)
```

Specify Alternate ModelSim Library Using System Shell

This example shows how to start ModelSim with an explicit option to specify the cosimulation library. You can start ModelSim from a system shell on the same machine as MATLAB, on a different machine, or on a machine with a different operating system.

This example code runs on a 64-bit Linux version of ModelSim. It does not matter what machine MATLAB is running on. Instead of using the default library in the ModelSim distribution, this example uses the library compiled with GCC 4.5.0. You can download this version of GCC with its associated system libraries from Mentor Graphics.

Modify the PATH variable to point to the desired version of the HDL simulator. Modify the LD_LIBRARY_PATH because the HDL simulator does not add the path to the system libraries, unless you saved the GCC at the root of the ModelSim installation. In a csh-compatible system shell, enter:

```
setenv PATH /tools/questasim/bin:${PATH}
setenv LD_LIBRARY_PATH /tools/mtigcc/gcc-4.5.0-linux_x86_64/lib64:${LD_LIBRARY_PATH}
setenv MTI_VCO_MODE 64
vlib work
vcom +acc+inverter inverter.vhd
vsim +acc+inverter -foreign \
    "matlabclient /tools/matlab/toolbox/edalink/extensions/modelsim/linux64/liblhmhdc_gcc450.so" \
    work.inverter
```

You can check the library resolution using `ldd`, as in the previous example.

See Also

Functions

`hdl_daemon` | `nc_launch` | `vsim`

Blocks

HDL Cosimulation

More About

- “Cross-Network Cosimulation” on page 10-14
- “Verify HDL Module with MATLAB Test Bench” on page 2-18
- “Verify HDL Module with Simulink Test Bench” on page 6-27

Cross-Network Cosimulation

In this section...

“Why Perform Cross-Network Cosimulation?” on page 10-14

“Preparing for Cross-Network Cosimulation” on page 10-14

“Performing Cross-Network Cosimulation Using MATLAB” on page 10-15

“Performing Cross-Network Cosimulation Using Simulink” on page 10-16

Why Perform Cross-Network Cosimulation?

You can perform cross-network cosimulation when your setup comprises one machine running MATLAB and Simulink software and another machine running the HDL simulator. Typically, a Windows-platform machine runs the MATLAB and Simulink software, while a Linux machine runs the HDL simulator. However, these procedures apply to any combination of platforms that HDL Verifier and the HDL simulator support.

Vivado cosimulation runs as a single process with a shared DLL, and therefore does not support cross-network cosimulation.

Preparing for Cross-Network Cosimulation

Before you cosimulate between the HDL simulator and MATLAB or Simulink across a network, perform the following steps:

- 1 Create your design and testing files.

ModelSim Users

- Create and compile your HDL design, and create your MATLAB function (for MATLAB cosimulation) or Simulink model (for Simulink cosimulation).
- If you are going to cosimulate with Simulink, use the `-voptargs=+acc` flag when you compile so that the design is not optimized, and include the same flag when you issue the `vsim` command (see “Performing Cross-Network Cosimulation Using Simulink” on page 10-16). Using this flag retains some unused signals from the design which are required by the Simulink model to run and display the results.

Xcelium Users

Create, compile, and elaborate your HDL design, and create your MATLAB function (for MATLAB cosimulation), or Simulink model (for Simulink cosimulation).

- 2 Copy HDL Verifier libraries to the machine with the HDL simulator
 - a Go to the system where you installed MATLAB. Then, find the folder in the MATLAB distribution where the HDL Verifier libraries reside.

You can usually find the libraries in the default installed folder:

```
matlabroot/toolbox/edalink/extensions/adaptor/platform/productlibraryname_
compiler_tag.ext
```

where the variable shown in the following table have the values indicated.

Variable	Value
<i>matlabroot</i>	The location where you installed the MATLAB software; default value is MATLAB/ <i>version</i> where <i>version</i> is the installed release (for example, R2009a).
<i>adaptor</i>	incisive or modelsim
<i>platform</i>	The operating system of the machine with the HDL simulator, for example, linux32. (For more information, see “Cosimulation Libraries” on page 10-9.)
<i>productlibraryname</i>	The name of the library files for MATLAB and for Simulink (for example, liblfmhd1c, liblfmhd1s for ModelSim users; liblfihd1c, liblfihd1s for Xcelium users). See “Cosimulation Libraries” on page 10-9.
<i>compiler_tag</i>	The compiler used to create the library (for example, gcc32 or spro). For more information, see “Cosimulation Libraries” on page 10-9.
<i>ext</i>	dll (dynamic link library—Windows only) or so (shared library extension)

For a list of all the HDL Verifier HDL shared libraries shipped, see “Default Libraries” on page 10-10.

- b** From the MATLAB machine, copy the HDL Verifier libraries you plan to use (which you determined in step 2) to the machine where you installed the HDL simulator. Make note of the location to which you copied the libraries; you'll need this information when you are actually establishing the connection to the HDL simulator. For purposes of this example, the sample code refers to the destination folder as HDLSERVER_LIB_LOCATION.

If you now want to cosimulate with MATLAB, see “Performing Cross-Network Cosimulation Using MATLAB” on page 10-15. If you want to cosimulate with Simulink, see “Performing Cross-Network Cosimulation Using Simulink” on page 10-16.

Performing Cross-Network Cosimulation Using MATLAB

To perform an HDL-simulator-to-MATLAB cosimulation session across a network, follow these steps:

ModelSim Users

- 1** In MATLAB, get an available socket using `hdldaemon`:

```
hdldaemon('socket',0)
```

Or assign one (that you know is available):

```
hdldaemon('socket',4449)
```

- 2** On the machine with the HDL simulator, launch the HDL simulator from a shell with the following command:

```
vsim -foreign "matlabclient /HDLSERVER_LIB_LOCATION/library_name;" design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
<i>library_name</i>	The name of the library you copied to the machine with the HDL simulator (in “Preparing for Cross-Network Cosimulation” on page 10-14).
<i>design_name</i>	The VHDL or Verilog design you want to load

- 3 In the HDL simulator, schedule the test bench or component (`matlabcp` or `matlabtb`). Specify the socket port number from step 1 and the name of the host machine where `hdldaemon` is running.

Xcelium Users

- 1 In MATLAB, get an available socket using `hdldaemon`:

```
hdldaemon('socket',0)
```

Or assign one:

```
hdldaemon('socket',4449)
```

- 2 Create a MATLAB configuration file (for loading the functions used in the HDL simulator) with the following contents:

```
//Command file for MATLAB HDL Verifier.
//Loading of foreign Library and HDL simulator functions.

-loadcfc /HDLSERVER_LIB_LOCATION/library_name:matlabclient
//TCL wrappers for MATLAB commands
-input @proc "nomatlabtb" "{args}" "{call" "nomatlabtb" "\$args}"
-input @proc "matlabtb" "{args}" "{call" "matlabtb" "\$args}"
-input @proc "matlabcp" "{args}" "{call" "matlabcp" "\$args}"
-input @proc "matlabtbeval" "{args}" "{call" "matlabtbeval" "\$args}"
```

Where *library_name* is the name of the library you copied in “Preparing for Cross-Network Cosimulation” on page 10-14. You may name this configuration file anything you like.

- 3 On the machine with the HDL simulator, launch the HDL simulator from a shell with the following command:

```
xmsim -gui -f matlab_config_file design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
<i>matlab_config_file</i>	The name of the MATLAB configuration file (from step 3)
<i>design_name</i>	The VHDL or Verilog design you want to load

- 4 In the HDL simulator, schedule the test bench or component (`matlabcp` or `matlabtb`). Specify the socket port number from step 1 and the name of the host where `hdldaemon` is running.

Performing Cross-Network Cosimulation Using Simulink

When you want to perform an HDL-simulator-to-Simulink cosimulation session across a network, follow these steps:

ModelSim Users

- 1 Launch the HDL simulator from a shell with the following command:

```
vsim -foreign "simlinkserver /HDLSERVER_LIB_LOCATION/library_name;
             -socket socket_num" -voptargs=+acc design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
<i>library_name</i>	The name of the library you copied to the machine with the HDL simulator (in "Preparing for Cross-Network Cosimulation" on page 10-14).
<i>socket_num</i>	The socket number you have chosen for this connection
<i>design_name</i>	The VHDL or Verilog design you want to load

- 2 On the machine with MATLAB and Simulink, start Simulink and open your model.
- 3 Double-click on the HDL Cosimulation block to open the Function Block Parameters dialog box.
- 4 Click on the **Connections** tab.
 - a Clear "The HDL simulator is running on this computer." HDL Verifier changes the Connection method to Socket.
 - b In the text box labeled **Host name**, enter the host name of the machine where the HDL simulator is located.
 - c In the text box labeled **Port number or service**, enter the socket number from step 1.
 - d Click **OK** to exit block dialog box, and save your changes.

Xcelium Users

- 1 Launch the HDL simulator from a shell with the following command:

```
xmsim -gui -loadvpi "/HDLSERVER_LIB_LOCATION/library_name:simlinkserver"
      +socket=socket_num design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
<i>library_name</i>	The name of the library you copied to the machine with the HDL simulator (in "Preparing for Cross-Network Cosimulation" on page 10-14).
<i>socket_num</i>	The socket number you have chosen for this connection
<i>design_name</i>	The VHDL or Verilog design you want to load

- 2 On the machine with MATLAB and Simulink, start Simulink and open your model.
- 3 Double-click on the HDL Cosimulation block to open the Function Block Parameters dialog box.
- 4 Click on the **Connections** tab.
 - a Clear the check box labeled **The HDL simulator is running on this computer**. HDL Verifier changes the Connection method to Socket.

- b** In the **Host name** box, enter the host name of the machine where the HDL simulator is located.
- c** In the **Port number or service** box, enter the socket number from step 1.
- d** Click **OK** to exit block dialog box, and save your changes.

Next, run your simulation, add more blocks, or make other desired changes. For instructions on using Simulink and the HDL simulator for cosimulation, see “Simulink as a Test Bench” on page 6-2 or “Component Simulation with Simulink” on page 7-2.

Test Bench and Component Function Writing

In this section...

“Writing Functions Using the HDL Instance Object” on page 10-19

“Writing Functions Using Port Information” on page 10-22

Writing Functions Using the HDL Instance Object

This section explains how you use the `use_instance_obj` argument for MATLAB functions `matlabcp` and `matlabtb`. This feature replaces the `iport`, `oport`, `tnext`, `tnow`, and `portinfo` arguments of the MATLAB function definition. Instead, an HDL instance object is passed to the function as an argument. With this feature, `matlabcp` and `matlabtb` function callbacks get the HDL instance object passed in: to hold state, provide read/write access protection for signals, and allow you to add state as desired.

With this feature, you gain the following advantages:

- Use of the same MATLAB function to represent behavior for different instances of the same module in HDL without need to create one-off wrapper functions.
- No need for special `portinfo` argument on first invocation.
- No need to use persistent or global variables.
- Better feedback and protections on reading/writing of signals.
- Use of object fields to identify the instance path and whether the call comes from a component or test bench function.
- Use of the field argument to pass user-defined arguments from the `matlabcp` or `matlabtb` instantiation on the HDL side to the function callbacks.

The `use_instance_obj` argument is identical for both `matlabcp` and `matlabtb`. You include the `-use_instance_obj` argument with `matlabcp` or `matlabtb` in the following format:

```
matlabcp modelname -mfunc funcname -use_instance_obj
```

When you use `use_instance_obj`, HDL Verifier passes an HDL instance object to the function specified with the `-mfunc` argument. The function called has the following signature:

```
function MyFunctionName(hdl_instance_obj)
```

The HDL instance object, `hdl_instance_obj`, has the fields shown in the following table.

Field	Read/Write Access	Description
<code>tnext</code>	Write only	Used to schedule a callback during the set time value. This field is the same as <code>tnext</code> in the old <code>portinfo</code> structure. For example: <pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9</pre> This line of code schedules a callback at time = 5 nanoseconds from <code>tnow</code> .

Field	Read/Write Access	Description
userdata	Read/Write	Stores state variables of the current <code>matlabcp</code> instance. You can retrieve the variables the next time the callback of this instance is scheduled.
simstatus	Read only	Stores the status of the HDL simulator. The HDL Verifier software sets this field to 'Init' during the first callback for this particular instance and to 'Running' thereafter. This field value is a read-only property. <pre>>> hdl_instance_obj.simstatus ans= Init</pre>
instance	Read only	Stores the full path of the Verilog/VHDL instance associated with the callback. <code>instance</code> is a read-only property. The value of this field equals that of the module instance specified with the function call. For example: In the HDL simulator: <pre>hdlsim> matlabcp osc_top -mfunc oscfilter use_instance_obj</pre> In MATLAB: <pre>>> hdl_instance_obj.instance ans= osc_top</pre>
argument	Read only	Stores the argument set by the <code>-argument</code> option of <code>matlabcp</code> . For example: <pre>matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo</pre> The verification software supports the <code>-argument</code> option only when you use it with <code>-use_instance_obj</code> , otherwise the argument is ignored. <code>argument</code> is a read-only property. <pre>>> hdl_instance_obj.argument ans= foo</pre>
portinfo	Read only	Stores information about the VHDL and Verilog ports associated with this instance. This field value is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the <code>portinfo</code> structure passes information such as the port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information" on page 10-24. <pre>hdl_instance_obj.portinfo.field1.field2.field3</pre> <p>Note When you use <code>use_instance_obj</code>, you access <code>tscale</code> through the HDL instance object. If you do not use <code>use_instance_obj</code>, you can still access <code>tscale</code> through <code>portinfo</code>.</p>

Field	Read/Write Access	Description
tscale	Read only	Stores the resolution limit (tick) in seconds of the HDL simulator. This field value is a read-only property. <pre>>> hdl_instance_obj.tscale</pre> <pre>ans= 1.0000e-009</pre> <p>Note When you use <code>use_instance_obj</code>, you access <code>tscale</code> through the HDL instance object. If you do not use <code>use_instance_obj</code>, you can still access <code>tscale</code> through <code>portinfo</code>.</p>
tnow	Read only	Stores the current time. This field value is a read-only property. <pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + fastestrate;</pre>
portvalues	Read/Write	Stores the current values of and sets new values for the output and input ports for a <code>matlabcp</code> instance. For example: <pre>>> hdl_instance_obj.portvalues</pre> <pre>ans = Read Only Input ports: clk_enable: [] clk: [] reset: [] Read/Write Output ports: sine_out: [22x1 char]</pre>
linkmode	Read only	Stores the status of the callback. The HDL Verifier software sets this field to <code>'testbench'</code> if the callback is associated with <code>matlabtb</code> and <code>'component'</code> if the callback is associated with <code>matlabcp</code> . This field value is a read-only property. <pre>>> hdl_instance_obj.linkmode</pre> <pre>ans= component</pre>

Example: Using matlabcp and the HDL Instance Object

In this example, the HDL simulator makes repeated calls to `matlabcp` to bind multiple HDL instances to the same MATLAB function. Each call contains `-argument` as a constructor parameter to differentiate behavior.

```
> matlabcp u1_filter1x -mfunc osc_filter -use_instance_obj -argument "oversample=1"
> matlabcp u1_filter8x -mfunc osc_filter -use_instance_obj -argument "oversample=8"
> matlabcp u2_filter8x -mfunc osc_filter -use_instance_obj -argument "oversample=8"
```

The MATLAB function callback, `osc_filter.m`, sets up user instance-based state using `obj userdata`, queries port and simulation context using other `obj` fields, and uses the passed in `obj.argument` to differentiate behavior.

```
function osc_filter(obj)
    if (strcmp(obj.simstatus,'Init'))
        ud = struct('Nbits', 22, 'Norder', 31, 'clockperiod', 80e-9, 'phase', 1);
        eval(obj.argument);
        if (~exist('oversample','var'))
```

```

        error('HdlLinkDemo:UseInstanceObj:BadCtorArg', ...
            'Bad constructor arg to osc_filter callback. Expecting
            ''oversample=value''.');
    end
    ud.oversample      = oversample;
    ud.oversampleperiod = ud.clockperiod/ud.oversample;
    ud.InDelayLine     = zeros(1,ud.Norder+1);

    centerfreq = 70/256;
    passband   = [centerfreq-0.01, centerfreq+0.01];
    b          = fir1((ud.Norder+1)*ud.oversample-1, passband./ud.oversample);
    ud.Hresp    = ud.oversample .* b;

    obj.userdata = ud;
end
...

```

Writing Functions Using Port Information

- “MATLAB Function Syntax and Function Argument Definitions” on page 10-22
- “Oscfilter Function Example” on page 10-23
- “Gaining Access to and Applying Port Information” on page 10-24

MATLAB Function Syntax and Function Argument Definitions

The syntax of a MATLAB component function is

```
function [oport, tnext] = MyFunctionName(iport, tnow, portinfo)
```

The syntax of a MATLAB test bench function is

```
function [iport, tnext] = MyFunctionName(oport, tnow, portinfo)
```

The input/output arguments (*iport* and *oport*) for a MATLAB component function are the reverse of the port arguments for a MATLAB test bench function. That is, the MATLAB component function returns signal data to the *outputs* and receives data from the *inputs* of the associated HDL module.

For more information on using *tnext* and *tnow* for simulation scheduling, see “Schedule Component Functions Using the *tnext* Parameter” on page 3-11.

The following table describes each of the test bench and component function parameters and the roles they play in each of the functions.

Parameter	Test Bench	Component
<i>iport</i>	<i>Output</i> Structure that forces (by deposit) values onto signals connected to input ports of the associated HDL module.	<i>Input</i> Structure that receives signal values from the input ports defined for the associated HDL module at the time specified by <i>tnow</i> .
<i>tnext</i>	<i>Output, optional</i> Specifies the time at which the HDL simulator schedules the next callback to MATLAB. <i>tnext</i> should be initialized to an empty value (<code>[]</code>). If <i>tnext</i> is not later updated, no new entries are added to the simulation schedule.	<i>Output, optional</i> Same as test bench.

Parameter	Test Bench	Component
oport	<i>Input</i> Structure that receives signal values from the output ports defined for the associated HDL module at the time specified by tnow.	<i>Output</i> Structure that forces (by deposit) values onto signals connected to output ports of the associated HDL module.
tnow	<i>Input</i> Receives the simulation time at which the MATLAB function is called. By default, time is represented in seconds. For more information see “Schedule Component Functions Using the tnext Parameter” on page 3-11.	Same as test bench.
portinfo	<i>Input</i> For the first call to the function only (at the start of the simulation), portinfo receives a structure whose fields describe the ports defined for the associated HDL module. For each port, the portinfo structure passes information such as the port's type, direction, and size.	Same as test bench.

If you are using `matlabcp`, initialize the function outputs to empty values at the beginning of the function as in the following example:

```
tnext = [];
oport = struct();
```

Note When you import VHDL signals, signal names in `iport`, `oport`, and `portinfo` are returned in all capitals.

You can use the port information to create a generic MATLAB function that operates differently depending on the port information supplied at startup. For more information on port data, see “Gaining Access to and Applying Port Information” on page 10-24.

Oscfilter Function Example

The following code gives the definition of the `oscfilter` MATLAB component function.

```
function [oport,tnext] = oscfilter(iport, tnow, portinfo)
```

The function name `oscfilter`, differs from the entity name `u_osc_filter`. Therefore, the component function name must be passed in explicitly to the `matlabcp` command that connects the function to the associated HDL instance using the `-mfunc` parameter.

The function definition specifies all required input and output parameters, as listed here:

<code>oport</code>	Forces (by deposit) values onto the signals connected to the entity's output ports, <code>filter1x_out</code> , <code>filter4x_out</code> and <code>filter8x_out</code> .
<code>tnext</code>	Specifies a time value that indicates when the HDL simulator will execute the next callback to the MATLAB function.
<code>oport</code>	Receives HDL signal values from the entity's input port, <code>osc_in</code> .
<code>tnow</code>	Receives the current simulation time.
<code>portinfo</code>	For the first call to the function, receives a structure that describes the ports defined for the entity.

The following figure shows the relationship between the HDL entity's ports and the MATLAB function's `oport` and `oport` parameters (example shown is for use with ModelSim).



Gaining Access to and Applying Port Information

HDL Verifier software passes information about the entity or module under test in the `portinfo` structure. The `portinfo` structure is passed as the third argument to the function. It is passed only in the first call to your ModelSim function. You can use the information passed in the `portinfo` structure to validate the entity or module under simulation. Three fields supply the information, as indicated in the next sample. . The content of these fields depends on the type of ports defined for the VHDL entity or Verilog module.

```
portinfo.field1.field2.field3
```

The following table lists possible values for each field and identifies the port types for which the values apply.

HDL Port Information

Field...	Can Contain...	Which...	And Applies to...
<i>field1</i>	in	Indicates the port is an input port	All port types
	out	Indicates the port is an output port	All port types
	inout	Indicates the port is a bidirectional port	All port types
	tscale	Indicates the simulator resolution limit in seconds as specified in the HDL simulator	All types
<i>field2</i>	<i>portname</i>	Is the name of the port	All port types
<i>field3</i>	type	Identifies the port type For VHDL: integer, real, time, or enum For Verilog: 'verilog_logic' identifies port types reg, wire, integer	All port types
	<i>right (VHDL only)</i>	The VHDL RIGHT attribute	VHDL integer, natural, or positive port types
	<i>left (VHDL only)</i>	The VHDL LEFT attribute	VHDL integer, natural, or positive port types
	size	VHDL: The size of the matrix containing the data Verilog: The size of the bit vector containing the data	All port types
	label	VHDL: A character literal or label Verilog: the character vector '01ZX'	VHDL: Enumerated types, including predefined types BIT, STD_LOGIC, STD_ULONGIC, BIT_VECTOR, and STD_LOGIC_VECTOR Verilog: All port types

The first call to the ModelSim function has three arguments including the `portinfo` structure. Checking the number of arguments is one way you can verify that `portinfo` was passed. For example:

```
if(nargin ==3)
  tscale = portinfo.tscale;
end
```

Simulation Speed Improvement Tips

In this section...

“Obtaining Baseline Performance Numbers” on page 10-26

“Analyzing Simulation Performance” on page 10-26

“Cosimulating Frame-Based Signals with Simulink” on page 10-27

Obtaining Baseline Performance Numbers

You can baseline the performance numbers by timing the execution of the HDL and the Simulink model separately and adding them together; you may not expect better performance than that. Make sure that the separate simulations are representative: running an HDL-only simulator with unrealistic input stimulus could be much faster than when realistic input stimulus is provided.

Analyzing Simulation Performance

While cosimulation entails a certain amount of overhead, sometimes the HDL simulation itself also slows performance. Ask yourself these questions when trying to analyze and improve performance:

Consideration	Suggestions for Improving Speed
Are you are using NFS or other remote file systems?	How fast is the file system? Consider using a different type or expect that the file system you're using will impact performance.
Are you using separate machines for Simulink and the HDL simulator?	How fast is the network? Wait until the network is quieter or contact your system administrator for advice on improving the connection.
Are you using the same machine for Simulink and the HDL simulator?	<ul style="list-style-type: none"> • Are you using shared pipes instead of sockets? Shared memory is faster. • Are the Simulink and HDL processes large enough to cause swaps to disk? Consider adding more memory; otherwise be aware that you're running a huge process and expect it to impact performance.
Are you using optimal (that is, as large as possible) Simulink sample rates on the HDL Cosimulation block?	<p>For example, if you set the output sample rate to 1 but only use every 10th sample, you could make the rate 10 and reduce the traffic between Simulink and the HDL simulator.</p> <p>Another example is if you place a very fast clock as an input to the HDL Cosimulation block, but have none of the other inputs need such a fast rate. In that case, you should generate the clock in HDL or (Xcelium and ModelSim users only) via the Clocks or Simulation pane on the HDL Cosimulation block.</p>

Consideration	Suggestions for Improving Speed
ModelSim users: Are you compiling/elaborating the HDL using the vopt flow?	Use <code>-voptargs==+acc</code> to optimize your design for maximum (HDL) simulator speed (ModelSim users only).
Are you using Simulink Accelerator mode?	Acceleration mode can speed up the execution of your model. See "Accelerating Models" in the <i>Simulink User's Guide</i> .
If you have the Communications Toolbox software, have you considered using Framed signals?	Framed signals reduce the number of Simulink/HDL interactions.

Cosimulating Frame-Based Signals with Simulink

Overview to Cosimulation with Frame-Based Signals

Frame-based processing can improve the computational time of your Simulink models, because multiple samples can be processed at once. Use of frame-based signals also lets you simulate the behavior of frame-based systems more realistically. The HDL Simulator block supports processing of single-channel frame-based signals.

A frame of data is a collection of sequential samples from a single channel or multiple channels. One frame of a single-channel signal is represented by a M-by-1 column vector. A signal is frame based if it is propagated through a model one frame at a time.

Frame-based processing requires the DSP System Toolbox software. Source blocks from the Sources library let you specify a frame-based signal by setting the **Samples per frame** block parameter. Most other signal processing blocks preserve the frame status of an input signal. You can use the Buffer block to buffer a sequence of samples into frames.

See "Working with Signals" in the DSP System Toolbox documentation for detailed information about frame-based processing.

Using Frame-Based Processing

You do not need to configure the HDL Simulator block in any special way for frame-based processing. To use frame-based processing in a cosimulation, connect one or more single-channel frame-based signals to one or more input ports of the HDL Simulator block. All such signals must meet the requirements described in "Frame-Based Processing Requirements and Restrictions" on page 10-27. The HDL Simulator block configures any outputs for frame-based operation at the suitable frame size.

Use of frame-based signals affects only the Simulink side of the cosimulation. The behavior of the HDL code under simulation in the HDL simulator does not change in any way. Simulink assumes that HDL simulator processing is sample based. Simulink assembles samples acquired from the HDL simulator into frames as required. Conversely, Simulink transmits output data to the HDL simulator in frames, which are unpacked and processed by the HDL simulator one sample at a time.

Frame-Based Processing Requirements and Restrictions

Observe the following restrictions and requirements when connecting frame-based signals in to an HDL Simulator block:

- Connection of mixed frame-based and sample-based signals to the same HDL Simulator block is not supported.

- Only single-channel frame-based signals can be connected to the HDL Simulator block. Use of multichannel (matrix) frame-based signals is not supported in this release.
- All frame-based signals connected to the HDL Simulator block must have the same frame size.

Frame-based processing in the Simulink model is transparent to the operation of the HDL model under simulation in the HDL simulator. The HDL model is presumed to be sample-based. The following constraint also applies to the HDL model under simulation in the HDL simulator: Specify VHDL signals as scalar values, not vectors or arrays (with the exception of bit vectors. VHDL and Verilog bit vectors are converted to the suitably-sized fixed-point scalar data type by the HDL Cosimulation block).

Frame-Based Cosimulation Example

This example shows the use of the HDL Simulator block to cosimulate a VHDL implementation of a simple lowpass filter. In the example, you will compare the performance of the simulation using frame-based and sample-based signals.

Note This tutorial is specific to ModelSim users; however, much of the process will be the same for Xcelium users.

The example files are (in *matlabroot*):

- The example model:

```
\toolbox\edalink\extensions\modelsim\modelsimdemos\frame_filter_cosim
```

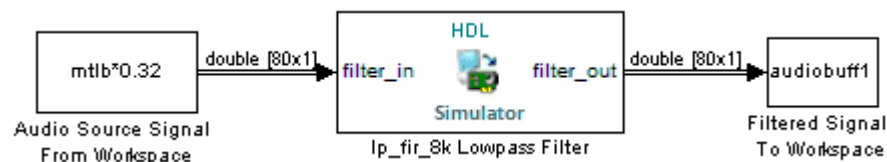
- VHDL code for the filter to be cosimulated:

```
\toolbox\edalink\extensions\modelsim\modelsimdemos\VHDL\frame_demos\lp_fir_8k.vhd
```

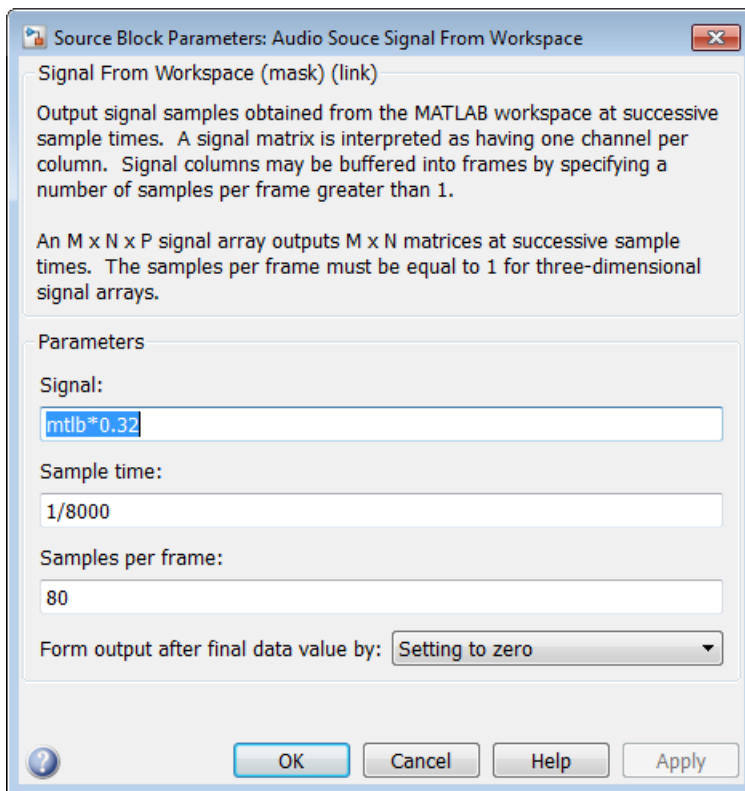
The filter was designed with Filter Designer and the code was generated by the Filter Design HDL Coder™.

The example uses the data file *matlabroot\toolbox\signal\signal\mtlb.mat* as an input signal. This file contains a speech signal. The sample data is of data type `double`, sampled at a rate of 8 kHz.

The next figure shows the `frame_filter_cosim` model.



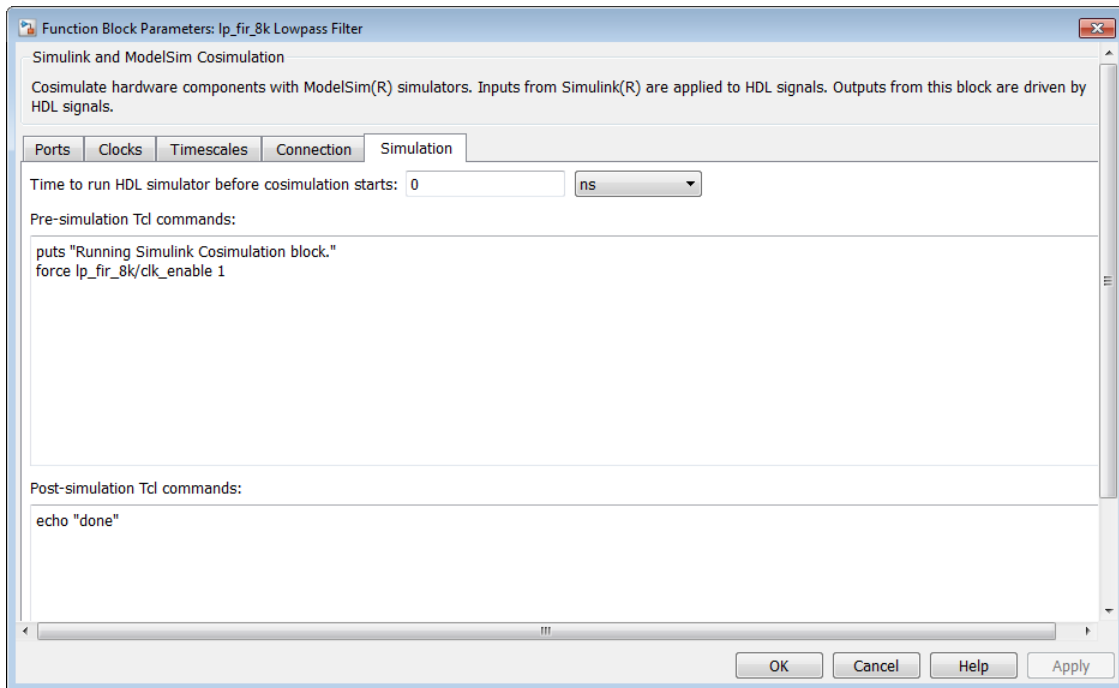
The Audio Source Signal From Workspace block provides an input signal from the workspace variable `mtlb`. The block is configured for an 8 kHz sample rate, with a frame size of 80, as shown in this figure.



The sample rate and frame size of the input signal propagate throughout the model.

The VHDL code file `lp_fir_8k.vhd` implements a simple lowpass FIR filter with a cutoff frequency of 1500 Hz. The HDL Simulator block simulates this HDL module. The HDL Simulator block ports and clock signal are configured to match the corresponding signals on the VHDL entity.

For the ModelSim simulation to execute as we want it to, the `clk_enable` signal of the `lp_fir_8k` entity must be forced high. The signal is forced by a pre-simulation command transmitted by the HDL Simulator block. The command has been entered into the **Simulation** pane of the HDL Simulator block, as shown in the following figure (example shown for use with ModelSim).



The HDL Simulator block returns output in the workspace variable `audiobuff1` via the Filtered Signal To Workspace block.

To run the cosimulation, perform the following steps:

- 1 Start MATLAB and make it your active window.
- 2 Set up and change to a writable working folder that is outside the context of your MATLAB installation folder.
- 3 Add the example folder to the MATLAB path:

```
matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos
```

- 4 Copy the VHDL file `lp_fir_8k.vhd` to your working folder.
- 5 Open the example model.

```
open frame_filter_cosim.slx
```

- 6 Load the source speech signal, which will be filtered, into the MATLAB workspace.

```
load mtlb
```

If you have a compatible sound card, you can play back the source signal by typing the following commands at the MATLAB command prompt:

```
a = audioplayer(mtlb,8000);
play(a);
```

- 7 Start ModelSim by typing the following command at the MATLAB command prompt:

```
vsim
```

The ModelSim window should now be active. If not, start it.

- 8 At the ModelSim prompt, create a design library, and compile the VHDL filter code from the source file `lp_fir_8k.vhd`, by typing the following commands:


```
vlib work
vmap work work
vcom lp_fir_8k.vhd
```

- 9** The lowpass filter to be simulated is defined as the entity `lp_fir_8k`. At the ModelSim prompt, load the instantiated entity `lp_fir_8k` for cosimulation:

```
vsimulink lp_fir_8k
```

ModelSim is now set up for cosimulation.

- 10** Start MATLAB. Run a simulation and measure elapsed time as follows:

```
tic; sim(gcs); toc
Elapsed time is 2.7190 seconds.
```

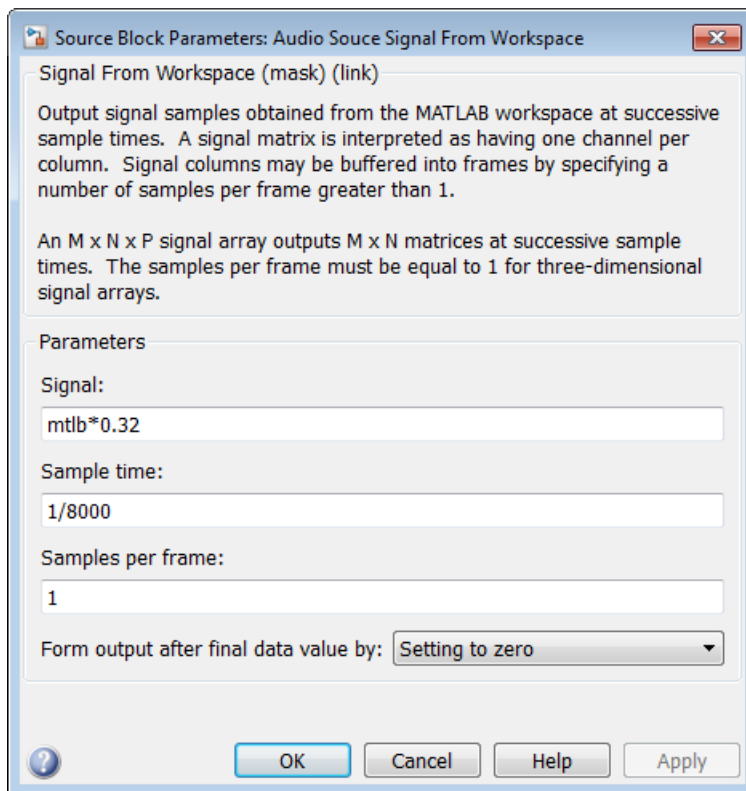
The timing in this code excerpt is typical for a run of this model given a simulation **Stop time** of 1 second and a frame size of 80 samples. Timings are system-dependent and will vary slightly from one simulation run to the next.

Take note of the timing you obtained. For the next simulation run, you will change the model to sample-based operation and obtain a comparative timing.

- 11** MATLAB stores the filtered audio signal returned from ModelSim in the workspace variable `audiobuff1`. If you have a compatible sound card, you can play back the filtered signal to hear the effect of the lowpass filter. Play the signal by typing the following commands at the MATLAB command prompt:

```
b = audioplayer(audiobuff1,8000);
play(b);
```

- 12** Open the block parameters dialog box of the Audio Source Signal From Workspace block and set the **Samples per frame** property to 1, as shown in this figure.



- 13 Close the dialog box, and select the Simulink window. On the **Modeling** tab, in the **Compile** section, click **Update Model**.

Now the source signal (and all signals inheriting from it) is a scalar.

- 14 Start ModelSim. At the ModelSim prompt, type

```
restart
```

- 15 Start MATLAB. Run a simulation and measure elapsed time as follows:

```
tic; sim(gcs); toc
Elapsed time is 3.8440 seconds.
```

Observe that the elapsed time has increased significantly with a sample-based input signal. The timing in this code excerpt is typical for a sample-based run of this model given a simulation **Stop time** of 1 second. Timings are system-dependent and will vary slightly from one simulation run to the next.

- 16 Close down the simulation in an orderly way. In ModelSim, stop the simulation by selecting **Simulate > End Simulation**, and quit ModelSim. Then, close the Simulink model window.

Race Conditions in HDL Simulators

In this section...

“Avoiding Race Conditions” on page 10-33

“Potential Race Conditions in Simulink Cosimulation Sessions” on page 10-33

“Potential Race Conditions in MATLAB Cosimulation Sessions” on page 10-34

“Further Reading” on page 10-34

Avoiding Race Conditions

A well-known issue in hardware simulation is the potential for different results on different runs when race conditions are present. Because the HDL simulator is a highly parallel execution environment, you must write the HDL such that the results do not depend on the ordering of process execution.

Although there are well-known coding idioms for achieving a realistic simulation of a design under test, you must always take special care at the test bench/DUT interfaces for applying stimulus and reading results, even in pure HDL environments. For an HDL/foreign language interface, such as with a Simulink or MATLAB cosimulation session, the problem is compounded if you do not have a common synchronization signal, such as a clock, coordinating the flow of data.

Potential Race Conditions in Simulink Cosimulation Sessions

All the signals on the interface of an HDL Cosimulation block in the Simulink library have an intrinsic sample rate associated with them. This sample rate can be thought of as an implicit clock that controls the simulation time at which a value change can occur. Because this implicit clock is completely unknown to the HDL engine (that is, it is not an HDL signal), the times at which input values are driven into the HDL or output values are sampled from the HDL are asynchronous to any clocks coded in HDL directly, even if they are nominally at the same frequency.

For Simulink value changes scheduled to occur at a specific simulation time, the HDL simulator does not make any guarantees as to the order that value change occurs versus some other blocking signal assignment. Thus, if the Simulink values are driven/sampled at the same time as an active clock edge in the HDL, there is a race condition.

For cases where your active HDL clock edge and your intrinsic Simulink active clock edges are at the same frequency, you can promote desired data propagation by offsetting one of those edges. Because the Simulink sample rates are always aligned with time 0, you can accomplish this offset by shifting the active clock edge in the HDL off of time 0. If you are coding the clock stimulus in HDL, use a delay operator ("after" or "#") to accomplish this offset.

When using a Tcl "force" command to describe the clock waveform, you can simply put the first active edge at some nonzero time. Using a nonzero value allows a Simulink sample rate that is the same as the fundamental clock rate in your HDL. This example shows a 20 ns clock (so the Simulink sample rates will also be every 20 ns) with an active positive edge that is offset from time 0 by 2 ns (example shown for use with Xcelium):

```
> force top.clk = 1'b0 -after 0 ns 1'b1 -after 2 ns 1'b0
    -after 12 ns -repeat 20 ns
```

For HDL Cosimulation blocks with Clock panes, you can define the clock period and active edge in that pane. The waveform definition places the **non-active** edge at time 0 and the **active** edge at time $T/2$. This placement sets the maximum setup and hold times for a clock with a 50% duty cycle.

If the Simulink sample rates are at a different frequency than the HDL clocks, then you must synchronize the signals between the HDL and Simulink as you would do with any multiple time-domain design, even one in pure HDL. For example, you can place two synchronizing flip-flops at the interface.

If your cosimulation does not include clocks, then you must also treat the interfacing of Simulink and the HDL code as being between asynchronous time domains. You may need to over-sample outputs to see that all data transitions are captured.

Potential Race Conditions in MATLAB Cosimulation Sessions

When you use the `-sensitivity`, `-rising_edge`, or `-falling_edge` scheduling options to `matlabtb` or `matlabcp` to trigger MATLAB function calls, the propagation of values follow the same semantics as a pure HDL design; the triggers must occur before the results can be calculated. You still can have race conditions, but they can be analyzed within the HDL alone.

However, when you use the `-time` scheduling option to `matlabtb` or `matlabcp`, or use `tnext` within the MATLAB function itself, the driving of signal values or sampling of signal values cannot be guaranteed in relation to any HDL signal changes. It is as if the potential race conditions in that time-based scheduling are like an implicit clock that is unknown to the HDL engine and not visible by just looking at the HDL code.

The remedies are the same as for the Simulink signal interfacing: make sure that the sampling and driving of signals does not occur at the same simulation times as the MATLAB function calls.

Further Reading

Problems interfacing designs from test benches and foreign languages, including race conditions in pure HDL environments, are well-known and extensively documented. Some texts that describe these issues include:

- The documentation for each vendor's HDL simulator product
- The HDL standards specifications
- Writing Testbenches: Functional Verification of HDL Models, Janick Bergeron, 2nd edition, © 2003
- Verilog and SystemVerilog Gotchas, Stuart Sutherland and Don Mills, © 2007
- SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, Chris Spear, © 2007
- Principles of Verifiable RTL Design, Lionel Bening and Harry D. Foster, © 2001

Supported Data Types

In this section...
“Converting HDL Data to Send to MATLAB or Simulink” on page 10-35
“Bit-Vector Indexing Differences Between MATLAB and HDL” on page 10-37
“Array Indexing Differences Between MATLAB and HDL” on page 10-38
“Converting Data for Manipulation” on page 10-39
“Converting Data for Return to the HDL Simulator” on page 10-40

Converting HDL Data to Send to MATLAB or Simulink

If your HDL application needs to send HDL data to a MATLAB function or a Simulink block, you may first need to convert the data to a type supported by MATLAB and the HDL Verifier software.

To program a MATLAB function or a Simulink block for an HDL model, you must understand the type conversions required by your application. You may also need to handle differences between the array indexing conventions used by the HDL you are using and MATLAB (see following section).

The data types of arguments passed in to the function determine the following:

- The types of conversions required before data is manipulated
- The types of conversions required to return data to the HDL simulator

The following table summarizes how the HDL Verifier software converts supported VHDL data types to MATLAB types based on whether the type is scalar or array.

VHDL-to-MATLAB Data Type Conversions

VHDL Types...	As Scalar Converts to...	As Array Converts to...
STD_LOGIC, STD_ULOGIC, and BIT	A character that matches the character literal for the desired logic state.	
STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, and UNSIGNED		A column vector of characters (as defined in VHDL Conversions for the HDL Simulator) with one bit per character.
Arrays of STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, and UNSIGNED		An array of characters (as defined above) with a size that is equivalent to the VHDL port size.
INTEGER and NATURAL	Type int32.	Arrays of type int32 with a size that is equivalent to the VHDL port size.
REAL	Type double.	Arrays of type double with a size that is equivalent to the VHDL port size.
TIME	Type double for time values in seconds and type int64 for values representing simulator time increments (see the description of the 'time' option in hdldaemon).	Arrays of type double or int64 with a size that is equivalent to the VHDL port size.
Enumerated types	Character vector or string scalar that contains the MATLAB representation of a VHDL label or character literal. For example, the label high converts to 'high' and the character literal 'c' converts to ''c''.	Cell array of character vectors or string array with each element equal to a label for the defined enumerated type. Each element is the MATLAB representation of a VHDL label or character literal. For example, the vector (one, '2', three) converts to the column vector ['one'; ''2''; 'three']. A user-defined enumerated type that contains only character literals, and then converts to a vector or array of characters as indicated for the types STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, and UNSIGNED.

The following table summarizes how the HDL Verifier software converts supported Verilog data types to MATLAB types. The software supports packed arrays up to 128bits for Verilog.

Verilog-to-MATLAB Data Type Conversions

Verilog Types...	Converts to...
wire, reg	A character or a column vector of characters that matches the character literal for the desired logic states (bits).

The following table summarizes how the HDL Verifier software converts supported SystemVerilog data types to MATLAB types. The software supports packed arrays up to 128bits for SystemVerilog.

SystemVerilog-to-MATLAB Data Type Conversions

SystemVerilog Types...	Converts to...
wire, reg, logic	A character or a column vector of characters that matches the character literal for the desired logic states (bits).
integer	A 32-element column vector of characters that matches the character literal for the desired logic states (bits). Supported for outputs only.
bit	boolean/ufix1
byte	int8/uint8
shortint	int16/uint16
int	int32
longint	int64/uint64
real	double
packed array	<ul style="list-style-type: none"> Input to HDL Cosimulation block: ufix/fix, or matrix of ufix/fix Output from HDL Cosimulation block: ufix^a

^a For currently supported packed array input or output to the HDL Cosimulation block, the total number of bits in the packed array has a 128 upper limit.

Note SystemVerilog support includes signals of the above types. The following SystemVerilog types are not supported:

- Unpacked arrays and multi-dimensional arrays
- shortreal SystemVerilog type
- SystemVerilog aggregate types such as struct and union
- SystemVerilog interfaces

Bit-Vector Indexing Differences Between MATLAB and HDL

In HDL, you have the flexibility to define a bit-vector with either MSB-0 or LSB-0 numbering. In MATLAB, bit-vectors are always considered LSB-0 numbering. In order to prevent data corruption, it is recommended that you use LSB-0 indexing for your HDL interfaces.

If you define a logic vector in VHDL as:

```
signal s1 : std_logic_vector(7 downto 0);
```

Or in Verilog as:

```
reg[7:0] s1;
```

It is mapped to int8 in MATLAB, with s1[7] as the MSB. Alternatively, if you define your VHDL logic vector as:

```
signal s1 : std_logic_vector(0 to 7);
```

Or in Verilog as:

```
reg[0:7] s1;
```

It is mapped to int8 in MATLAB, with s1[0] as the MSB.

Array Indexing Differences Between MATLAB and HDL

In multidimensional arrays, the same underlying OS memory buffer maps to different elements in MATLAB and the HDL simulator (this mapping only reflects different ways the different languages offer for naming the elements of the same array). When you use both the `matlabtb` and `matlabcp` functions, be careful to assign and interpret values consistently in both applications.

In HDL, a multidimensional array declared as:

```
type matrix_2x3x4 is array (0 to 1, 4 downto 2) of std_logic_vector(8 downto 5);
```

has a memory layout as follows:

bit	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
dim1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
dim2	4	4	4	4	3	3	3	3	2	2	2	2	4	4	4	4	3	3	3	3	2	2	2	2
dim3	8	7	6	5	8	7	6	5	8	7	6	5	8	7	6	5	8	7	6	5	8	7	6	5

This same layout corresponds to the following MATLAB 4x3x2 matrix:

bit	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
dim1	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
dim2	1	1	1	1	2	2	2	2	3	3	3	3	1	1	1	1	2	2	2	2	3	3	3	3
dim3	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2

Therefore, if H is the HDL array and M is the MATLAB matrix, the following indexed values are the same:

```

b1  H(0,4,8) = M(1,1,1)
b2  H(0,4,7) = M(2,1,1)
b3  H(0,4,6) = M(3,1,1)
b4  H(0,4,5) = M(4,1,1)
b5  H(0,3,8) = M(1,2,1)
b6  H(0,3,7) = M(2,2,1)
...
b19 H(1,3,6) = M(3,2,2)
b20 H(1,3,5) = M(4,2,2)
b21 H(1,2,8) = M(1,3,2)
b22 H(1,2,7) = M(2,3,2)
b23 H(1,2,6) = M(3,3,2)
b24 H(1,2,5) = M(4,3,2)

```


You can extend this indexing to N-dimensions. In general, the dimensions—if numbered from left to right—are reversed. The right-most dimension in HDL corresponds to the left-most dimension in MATLAB.

Converting Data for Manipulation

Depending on how your simulation MATLAB function uses the data it receives from the HDL simulator, you may need to code the function to convert data to a different type before manipulating it. The following table lists circumstances under which you would require such conversions.

Required Data Conversions

If You Need the Function to...	Then...
Compute numeric data that is received as a type other than <code>double</code>	Use the <code>double</code> function to convert the data to type <code>double</code> before performing the computation. For example: <pre>datas(inc+1) = double(idata);</pre>
Convert a standard logic or bit vector to an unsigned integer or positive decimal	Use the <code>mv12dec</code> function to convert the data to an unsigned decimal value. For example: <pre>uval = mv12dec(oport.val)</pre> This example assumes the standard logic or bit vector is composed of the character literals '1' and '0' only. These are the only two values that can be converted to an integer equivalent. The <code>mv12dec</code> function converts the binary data that the MATLAB function receives from the entity's <code>osc_in</code> port to unsigned decimal values that MATLAB can compute. See <code>mv12dec</code> for more information on this function.
Convert a standard logic or bit vector to a negative decimal	Use the following application of the <code>mv12dec</code> function to convert the data to a signed decimal value. For example: <pre>suval = mv12dec(oport.val, true);</pre> This example assumes the standard logic or bit vector is composed of the character literals '1' and '0' only. These are the only two values that can be converted to an integer equivalent.

Examples

The following code excerpt illustrates data type conversion of data passed in to a callback:

```
InDelayLine(1) = InputScale * mv12dec(iport.osc_in',true);
```

This example tests port values of VHDL type `STD_LOGIC` and `STD_LOGIC_VECTOR` by using the `all` function as follows:

```
all(oport.val == '1' | oport.val
== '0')
```

This example returns `True` if all elements are '1' or '0'.

Converting Data for Return to the HDL Simulator

If your simulation MATLAB function needs to return data to the HDL simulator, you may first need to convert the data to a type supported by the HDL Verifier software. The following tables list circumstances under which such conversions are required for VHDL and Verilog.

Note When data values are returned to the HDL simulator, the char array size must match the HDL type, including leading zeroes, if applicable. For example:

```
oport.signal = dec2mvl(2)
```

will only work if `signal` is a 2-bit type in HDL. If the HDL type is anything else, you *must* specify the second argument:

```
oport.signal = dec2mvl(2, N)
```

where `N` is the number of bits in the HDL data type.

VHDL Conversions for the HDL Simulator

To Return Data to an IN Port of Type...	Then...
STD_LOGIC, STD_ULOGIC, or BIT	<p>Declare the data as a character that matches the character literal for the desired logic state. For STD_LOGIC and STD_ULOGIC, the character can be 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', or '-'. For BIT, the character can be '0' or '1'. For example:</p> <pre>iport.s1 = 'X'; %STD_LOGIC iport.bit = '1'; %BIT</pre>
STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	<p>Declare the data as a column vector or row vector of characters (as defined above) with one bit per character. For example:</p> <pre>iport.s1v = 'X10ZZ'; %STD_LOGIC_VECTOR iport.bitv = '10100'; %BIT_VECTOR iport.uns = dec2mvl(10,8); %UNSIGNED, 8 bits</pre>
Array of STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	<p>Declare the data as an array of type character with a size that is equivalent to the VHDL port size. See "Array Indexing Differences Between MATLAB and HDL" on page 10-38.</p>
INTEGER or NATURAL	<p>Declare the data as an array of type int32 with a size that is equivalent to the VHDL array size. Alternatively, convert the data to an array of type int32 with the MATLAB int32 function before returning it. Be sure to limit the data to values with the range of the VHDL type. If you want to, check the right and left fields of the portinfo structure. For example:</p> <pre>iport.int = int32(1:10)';</pre>
REAL	<p>Declare the data as an array of type double with a size that is equivalent to the VHDL port size. For example:</p> <pre>iport.dbl = ones(2,2);</pre>
TIME	<p>Declare a VHDL TIME value as time in seconds, using type double, or as an integer of simulator time increments, using type int64. You can use the two formats interchangeably and what you specify does not depend on the hdl_daemon 'time' option (see hdl_daemon), which applies to IN ports only. Declare an array of TIME values by using a MATLAB array of identical size and shape. All elements of a given port are restricted to time in seconds (type double) or simulator increments (type int64), but otherwise you can mix the formats. For example:</p> <pre>iport.t1 = int64(1:10)'; %Simulator time %increments iport.t2 = 1e-9; %1 nsec</pre>

To Return Data to an IN Port of Type...	Then...
Enumerated types	<p>Declare the data as a character vector or string scalar for scalar ports or a cell array of character vectors or string array for array ports with each element equal to a label for the defined enumerated type. The 'label' field of the portinfo structure lists all valid labels (see "Gaining Access to and Applying Port Information" on page 10-24). Except for character literals, labels are not case sensitive. In general, you should specify character literals completely, including the single quotes, as in the first example shown here. .</p> <pre> iport.char = {'A', 'B'}; %Character %literal iport.undef = 'mylabel'; %User-defined label </pre>
Character array for standard logic or bit representation	<p>Use the dec2mvl function to convert the integer. For example:</p> <pre> oport.slva =dec2mvl([23 99],8)'; </pre> <p>This example converts two integers to a 2-element array of standard logic vectors consisting of 8 bits.</p>

Verilog Conversions for the HDL Simulator

To Return Data to an input Port of Type...	Then...
reg, wire	<p>Declare the data as a character or a column vector of characters that matches the character literal for the desired logic state. For example:</p> <pre> iport.bit = '1'; </pre>

SystemVerilog Conversions for the HDL Simulator

To Return Data to an input Port of Type...	Then...
reg, wire, logic	<p>Declare the data as a character or a column vector of characters that matches the character literal for the desired logic state. For example:</p> <pre> iport.bit = '1'; </pre>
integer	<p>Declare the data as a 32-element column vector of characters (as defined above) with one bit per character.</p>

Packed arrays are supported up to 128bits.

Note SystemVerilog support includes only scalar signals of the above types. The following SystemVerilog types are not supported:

- Arrays and multi-dimensional arrays
- shortreal SystemVerilog type
- SystemVerilog aggregate types such as struct and union
- SystemVerilog interfaces

See Also

Cosimulation Wizard | HDL Cosimulation | `hdlverifier.HDLCosimulation`

Related Examples

- “Import HDL Code for MATLAB Function” on page 9-4
- “Import HDL Code for MATLAB System Object” on page 9-12
- “Import HDL Code for HDL Cosimulation Block” on page 9-24

Simulation Timescales

In this section...

- “Overview to the Representation of Simulation Time” on page 10-44
- “Defining the Simulink and HDL Simulator Timing Relationship” on page 10-45
- “Setting the Timing Mode with HDL Verifier” on page 10-45
- “Specify Timing Relationship Automatically” on page 10-46
- “Relative Timing Mode” on page 10-48
- “Absolute Timing Mode” on page 10-51
- “Timing Mode Usage Considerations” on page 10-53
- “Setting HDL Cosimulation Block Port Sample Times” on page 10-54

Overview to the Representation of Simulation Time

The representation of simulation time differs significantly between the HDL simulator and Simulink. Each application has its own timing engine and the verification software must synchronize the simulation times between the two.

In the HDL simulator, the unit of simulation time is referred to as a *tick*. The duration of a tick is defined by the HDL simulator *resolution limit*. The default resolution limit is 1 ns, but may vary depending on the simulator.

- **ModelSim Users:**

To determine the current ModelSim resolution limit, enter `echo $resolution` or `report simulator state` at the ModelSim prompt. You can override the default resolution limit by specifying the `-t` option on the ModelSim command line, or by selecting a different Simulator Resolution in the ModelSim Simulate dialog box. Available resolutions in ModelSim are 1x, 10x, or 100x in units of fs, ps, ns, us, ms, or sec. See the ModelSim documentation for further information.

- **Xcelium Users:**

To determine the current HDL simulator resolution limit, enter `echo $timescale` at the HDL simulator prompt. See the HDL simulator documentation for further information.

- **Vivado Users:** Open the HDL Cosimulation block mask. You can see the value for the **HDL Time Precision** parameter in the **Block Info** tab. To change this value, you must open the **Cosimulation Wizard**, and regenerate the block specifying a different **HDL time precision** parameter in the **Simulation Options** pane.

Simulink maintains simulation time as a double-precision value scaled to seconds. This representation accommodates modeling of both continuous and discrete systems.

The relationship between Simulink and the HDL simulator timing affects the following aspects of simulation:

- Total simulation time
- Input port sample times
- Output port sample times

- Clock periods

During a simulation run, Simulink communicates the current simulation time to the HDL simulator at each intermediate step. (An intermediate step corresponds to a Simulink sample time hit. Upon each intermediate step, new values are applied at input ports, or output ports are sampled.)

To bring the HDL simulator up-to-date with Simulink during cosimulation, you must convert sampled Simulink time to HDL simulator time (ticks) and allow the HDL simulator to run for the computed number of ticks.

Defining the Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the HDL Verifier interface:

- By defining the timing relationship manually (with **Timescales** pane)

When you define the relationship manually, you determine how many femtoseconds, picoseconds, nanoseconds, microseconds, milliseconds, seconds, or ticks in the HDL simulator represent 1 second in Simulink.

- By allowing HDL Verifier to define the timescale (with **Timescales** pane)

When you allow the software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, HDL Verifier attempts to set the signal rate on the Simulink model port to the lowest possible number of HDL simulator ticks.

Setting the Timing Mode with HDL Verifier

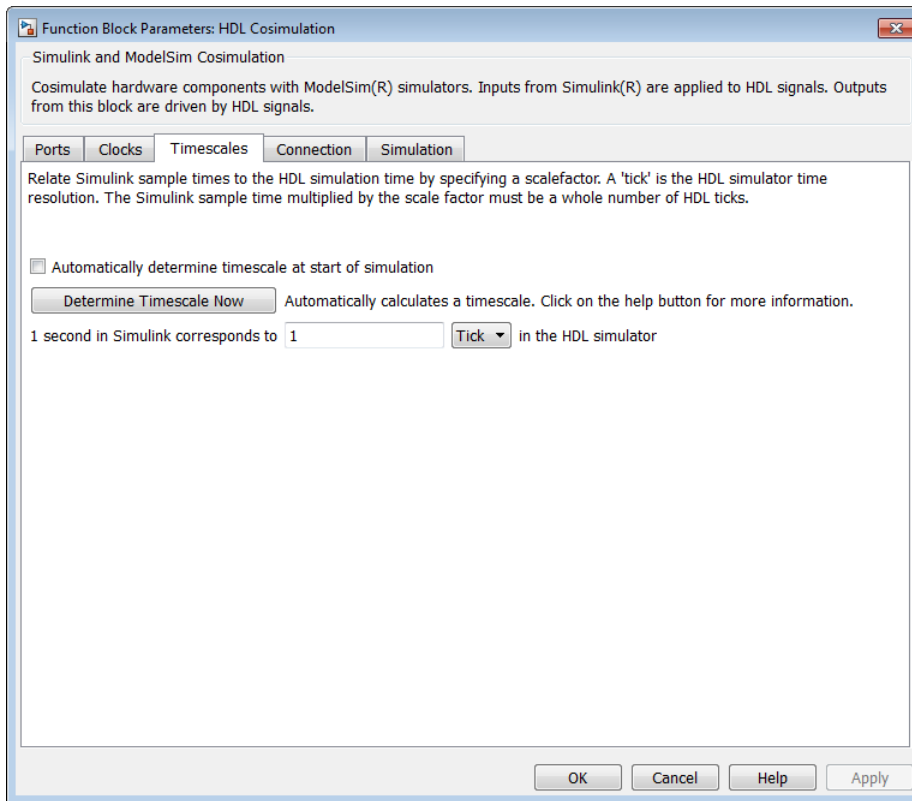
The **Timescales** pane of the HDL Cosimulation block parameters dialog box defines a correspondence between one second of Simulink time and some quantity of HDL simulator time. This quantity of HDL simulator time can be expressed in one of the following ways:

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation block defaults to relative timing mode for cosimulation. For more on relative timing mode, see “Relative Timing Mode” on page 10-48.
- In *absolute* units (such as milliseconds or nanoseconds). In this case, the cosimulation is said to operate in *absolute timing mode*. For more on absolute timing mode, see “Absolute Timing Mode” on page 10-51.

The **Timescales** pane lets you choose an optimal timing relationship between Simulink and the HDL simulator, either by entering the HDL simulator equivalent or by letting HDL Verifier calculate a timescale for you.

You can choose to have HDL Verifier calculate a timescale while you are setting the parameters on the block dialog by clicking the **Timescale** option then clicking **Determine Timescale Now** (this parameter shows as **Show Times and Suggest Timescale** for Vivado) or you can have HDL Verifier calculate the timescale when simulation begins by selecting **Automatically determine timescale at start of simulation**.

The next figure shows the default settings of the **Timescales** pane (example shown is for use with ModelSim).

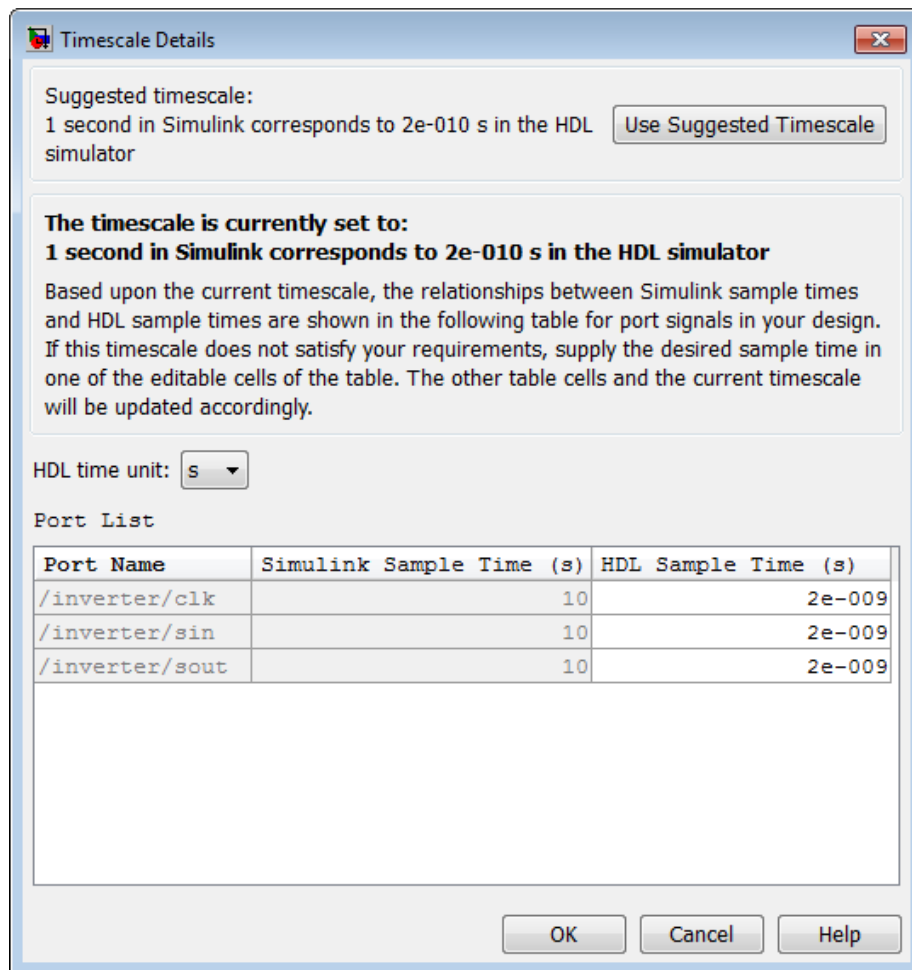


For instructions on setting the timing mode either manually or with the **Timescales** dialog box, see the **Timescales** pane in the HDL Cosimulation block reference.

Specify Timing Relationship Automatically

To have the HDL Verifier software calculate the timing relationship for you:

- 1 Start the HDL simulator. HDL Verifier software can obtain the resolution limit of the HDL simulator only when that simulator is running.
- 2 Choose to have HDL Verifier software suggest a timescale, immediately, or calculate the timescale when you start the Simulink simulation.
 - To have the calculation performed while you are configuring the block, on the **Timescale** tab, click **Determine Timescale Now**. The software connects Simulink with the HDL simulator so that Simulink can use the HDL simulator resolution to calculate the best timescale. The link then displays those results to you in the **Timescale Details** dialog box.



You can accept the suggested timescale, or change the port list directly:

- To revert to the originally calculated settings, click **Use Suggested Timescale**.
- To view sample times for all ports in the HDL design, select **Show all ports and clocks**.
- To have the calculation performed when the simulation begins, select **Automatically determine timescale at start of simulation**, and click **Apply**. You obtain the same **Timescale Details** dialog box when the simulation starts in Simulink.

For the results to display, make sure that the HDL simulator is running and the design is loaded for cosimulation. The simulation does not have to be running.

HDL Verifier software analyzes all the clock and port signal rates from the HDL Cosimulation block when the software calculates the scale factor.

The link software returns the sample rate in either seconds or ticks:

- If the results are in seconds, then the link software was able to resolve the timing differences in favor of fidelity (absolute time).
- If the results are in ticks, then the link software was best able to resolve the timing differences in favor of efficiency (relative time).

Each time you select **Determine Timescale Now** or **Automatically determine timescale at start of simulation**, an interactive display opens. This display explains the results of the timescale calculations. If the link software cannot calculate a timescale for the given sample times, adjust your sample times in the **Port List**.

- 3 Click **Apply** to commit your changes.

Restrictions

- HDL Verifier does not support timescales calculated automatically from frame-based signals.
- HDL Verifier software cannot automatically calculate a sample timescale based on any signals driven via Tcl commands or in the HDL simulator. The link software cannot perform such calculations because it cannot know the rates of these signals.
- For the results to display, make sure that the HDL simulator is running and the design is loaded for cosimulation. The simulation does not have to be running.

Relative Timing Mode

Relative timing mode defines the following one-to-one correspondence between simulation time in Simulink and the HDL simulator:

One second in Simulink corresponds to *N ticks* in the HDL simulator, where N is a scale factor.

This correspondence holds regardless of the HDL simulator timing resolution.

The following pseudocode shows how Simulink time units are converted to HDL simulator ticks:

```
InTicks = N * tInSecs
```

where *InTicks* is the HDL simulator time in ticks, *tInSecs* is the Simulink time in seconds, and N is a scale factor.

Operation of Relative Timing Mode

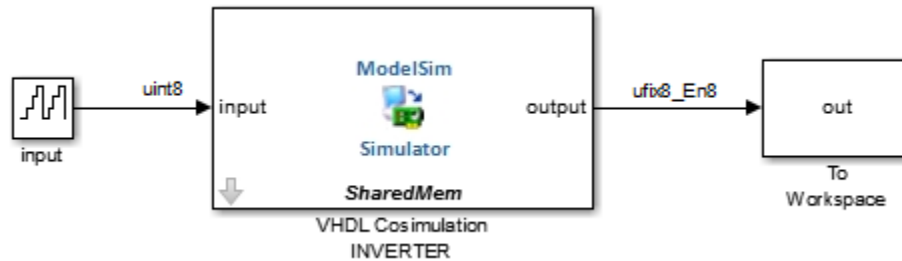
The HDL Cosimulation block defaults to relative timing mode, with a scale factor of 1. Thus, 1 Simulink second corresponds to 1 tick in the HDL simulator. In the default case:

- If the total simulation time in Simulink is specified as N seconds, then the HDL simulation will run for exactly N ticks (i.e., N ns at the default resolution limit).
- Similarly, if Simulink computes the sample time of an HDL Cosimulation block input port as *Tsi* seconds, new values will be deposited on the HDL input port at exact multiples of *Tsi* ticks. If an output port has an explicitly specified sample time of *Tso* seconds, values will be read from the HDL simulator at multiples of *Tso* ticks.

Relative Timing Mode Example

To understand how relative timing mode operates, review cosimulation results from the following example model.

For Use with ModelSim



The model contains an HDL Cosimulation block (labeled “VHDL Cosimulation INVERTER”) simulating an 8-bit inverter that is enabled by an explicit clock. The inverter has a single input and a single output. The following sample shows VHDL code for the inverter:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY inverter IS PORT (
    inport : IN  std_logic_vector := "11111111";
    outport: OUT std_logic_vector := "00000000";
    clk:IN  std_logic
);
END inverter;

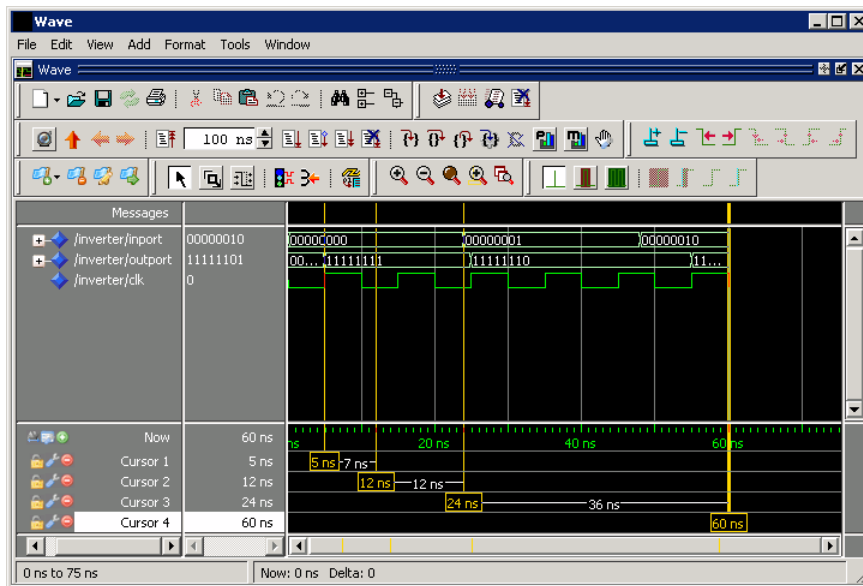
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ARCHITECTURE behavioral OF inverter IS
BEGIN
    PROCESS(clk)
    BEGIN
        IF (clk'EVENT AND clk = '1') THEN
            outport <= NOT inport;
        END IF;
    END PROCESS;
END behavioral;
```

A cosimulation of this model might have the following settings:

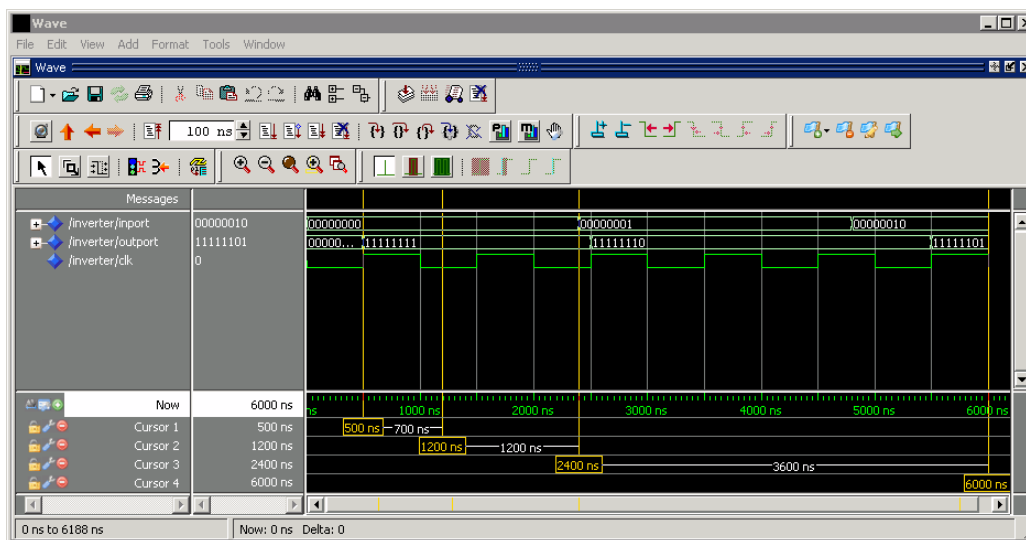
- Simulation parameters in Simulink:
 - **Timescales** parameters: default (relative timing with a scale factor of 1)
 - Total simulation time: 60 s
 - Input port (/inverter/inport) sample time: 24 s
 - Output port (/inverter/outport) sample time: 12 s
 - Clock (inverter/clk) period: 10 s
- ModelSim resolution limit: 1 ns

The next figure shows the ModelSim **wave** window after a cosimulation run of the example Simulink model for 60 ns. The **wave** window shows that ModelSim simulated for 60 ticks (60 ns). The inputs change at multiples of 24 ns and the outputs are read from ModelSim at multiples of 12 ns. The clock is driven low and high at intervals of 5 ns.

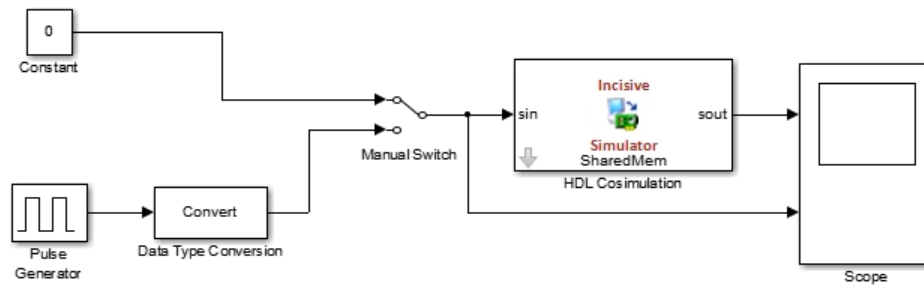


Now consider a cosimulation of the same model, this time configured with a scale factor of 100 in the **Timescales** pane.

The ModelSim **wave** window in the next figure shows that Simulink port and clock times were scaled by a factor of 100 during simulation. ModelSim simulated for 6 microseconds (60 * 100 ns). The inputs change at multiples of 24 * 100 ns and outputs are read from ModelSim at multiples of 12 * 100 ns. The clock is driven low and high at intervals of 500 ns.



For Use with Xcelium



The model contains an HDL Cosimulation block (labeled `HDL_Cosimulation1`) simulating an 8-bit inverter that is enabled by an explicit clock. The inverter has a single input and a single output. The following code excerpt lists the Verilog code for the inverter:

```
module inverter_clock_vl(sin, sout,clk);

input [7:0] sin;
output [7:0] sout;
input clk;
reg [7:0] sout;

always @(posedge clk)
  sout <= ! (sin);
endmodule
```

A cosimulation of this model might have the following settings:

- Simulation parameters in Simulink:
 - **Timescales** parameters: 1 Simulink second = 10 HDL simulator ticks
 - Total simulation time: 30 s
 - Input port (`inverter_clock_vl.sin`) sample time: N/A
 - Output port (`inverter_clock_vl.sout`) sample time: 1 s
 - Clock (`inverter_clock_vl.clk`) period: 5 s
- HDL simulator resolution limit: 1 ns

The previous example was excerpted from the HDL Verifier Inverter tutorial. For more information, see HDL Verifier demos.

Absolute Timing Mode

Absolute timing mode lets you define the timing relationship between Simulink and the HDL simulator in terms of absolute time units and a scale factor:

One second in Simulink corresponds to $(N * Tu)$ seconds in the HDL simulator, where Tu is an absolute time unit (for example, ms, ns, etc.) and N is a scale factor.

In absolute timing mode, all sample times and clock periods in Simulink are quantized to HDL simulator ticks. The following pseudocode illustrates the conversion:

```
tInTicks = tInSecs * (tScale / tRL)
```

where:

- `tInTicks` is the HDL simulator time in ticks.
- `tInSecs` is the Simulink time in seconds.
- `tScale` is the timescale setting (unit and scale factor) chosen in the **Timescales** pane of the HDL Cosimulation block.
- `tRL` is the HDL simulator resolution limit.

For example, given a **Timescales** pane setting of 1 s and an HDL simulator resolution limit of 1 ns, an output port sample time of 12 ns would be converted to ticks as follows:

$$tInTicks = 12ns * (1s / 1ns) = 12$$

Operation of Absolute Timing Mode

To configure the Timescales parameters for absolute timing mode, you select a unit of absolute time that corresponds to a Simulink second, rather than selecting `Tick`.

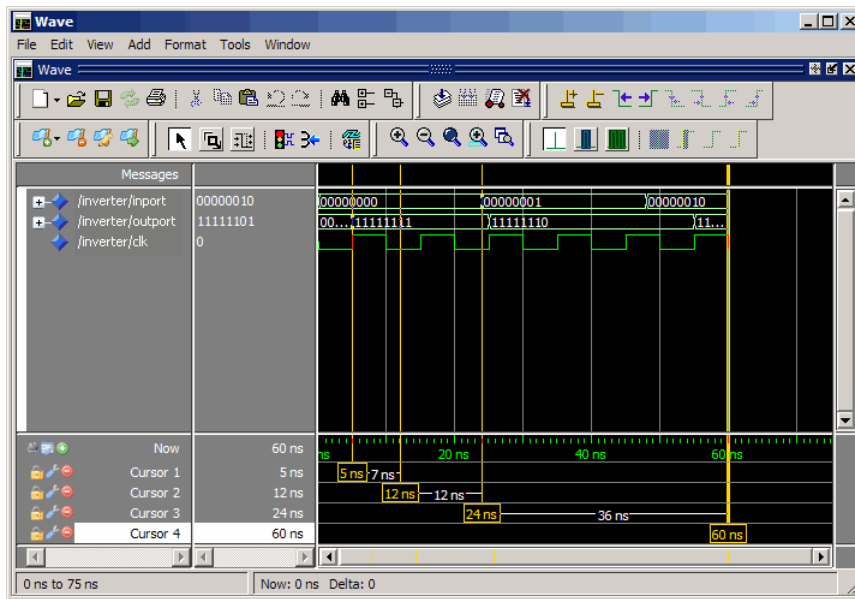
Absolute Timing Mode Example

To understand the operation of absolute timing mode, you will again consider the example model discussed in “Operation of Relative Timing Mode” on page 10-48. Suppose that the model is reconfigured as follows:

- Simulation parameters in Simulink:
 - **Timescale** parameters: 1 s of Simulink time corresponds to 1 s of HDL simulator time.
 - Total simulation time: $60e-9$ s (60ns)
 - Input port (`/inverter/inport`) sample time: $24e-9$ s (24 ns)
 - Output port (`/inverter/outport`) sample time: $12e-9$ s (12 ns)
 - Clock (`inverter/clk`) period: $10e-9$ s (10 ns)
- HDL simulator resolution limit: 1 ns

Given these simulation parameters, the Simulink software will cosimulate with the HDL simulator for 60 ns, during which Simulink will sample inputs at intervals of 24 ns, update outputs at intervals of 12 ns, and drive clocks at intervals of 10 ns.

The following figure shows a ModelSim **wave** window after a cosimulation run.



Timing Mode Usage Considerations

When setting a timescale mode, you may need to choose your setting based on the following considerations.

- “Timing Mode Usage Restrictions” on page 10-53
- “Noninteger Time Periods” on page 10-53

Timing Mode Usage Restrictions

The following restrictions apply to the use of absolute and relative timing modes:

- When multiple HDL Cosimulation blocks in a model are communicating with a single instance of the HDL simulator, all HDL Cosimulation blocks must have the same **Timescales** pane settings.
- If you change the **Timescales** pane settings in an HDL Cosimulation block between consecutive cosimulation runs, you must restart the simulation in the HDL simulator.
- If you specify a Simulink sample time that cannot be expressed as a whole number of HDL ticks, you will get an error.

Noninteger Time Periods

When using noninteger time periods, the HDL simulator cannot represent such an infinitely repeating value. So the simulator truncates the time period, but it does so differently than how Simulink truncates the value, and the two time periods no longer match up.

The following example demonstrates how to set the timing relationship in the following scenario: you want to use a sample period of $\frac{1}{3Hz}$ in Simulink, which corresponds to a noninteger time period.

The key idea here is that you must always be able to relate a Simulink time with an HDL tick. The HDL tick is the finest time slice the HDL simulator recognizes; for ModelSim, the default tick is 1 ns, but it can be made as precise as 1 fs.

However, a 3 Hz signal actually has a period of 333.3333333333... ms, which is not a valid tick period for the HDL simulator. The HDL simulator will truncate such numbers. But Simulink does not make the same decision; thus, for cosimulation where you are trying to keep two independent simulators in synchronization, you should not assume anything. Instead you have to decide whether it is convenient to truncate or round the number.

Therefore, the solution is to "snap" either the Simulink sample time or the HDL sample time (via the timescale) to valid numbers. There are infinite possibilities, but here are some possible ways to perform a snap:

- Change Simulink sample times from 1/3 sec to 0.33333 sec and set the cosimulation block timescale to '1 second in Simulink = 1 second in the HDL simulator'. If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 0.33333 sec.
- Keep Simulink sample times at 1/3 sec. and 1 second in Simulink = 6 ticks in the HDL simulator.
 - If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 1/3. Briefly, this specification tells Simulink to make each Simulink sample time correspond to every $(1/3 \cdot 6) = 2$ ticks, regardless of the HDL time resolution.
 - If your default HDL simulator resolution is 1 ns, that means your HDL sample times are every 2 ns. This sample time will work in a way so that for every Simulink sample time there is a corresponding HDL sample time.
 - However, Simulink thinks in terms of 1/3 sec periods and the HDL in terms of 2 ns periods. Thus, you could get confused during debug. If you want this to match the real period (such as to 5 places, i.e. 333.33 ms), you can follow the next option listed.
- Keep Simulink sample times at 1/3 sec and 1 second in Simulink = 0.99999e9 ticks in the HDL simulator. If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 1/3.

Setting HDL Cosimulation Block Port Sample Times

In general, Simulink handles the sample time for the ports of an HDL Cosimulation block as follows:

- If an input port is connected to a signal that has an explicit sample time, based on forward propagation, Simulink applies that rate to that input port.
- If an input port is connected to a signal that *does not have* an explicit sample time, Simulink assigns a sample time that is equal to the least common multiple (LCM) of all identified input port sample times for the model.
- After Simulink sets the input port sample periods, it applies user-specified output sample times to all output ports. Sample times must be explicitly defined for all output ports.

If you are developing a model for cosimulation in *relative* timing mode, consider the following sample time guideline:

Specify the output sample time for an HDL Cosimulation block as an integer multiple of the resolution limit defined in the HDL simulator. Use the HDL simulator command `report simulator state` to check the resolution limit of the loaded model. If the HDL simulator resolution limit is 1 ns and you specify a block's output sample time as 20, Simulink interacts with the HDL simulator every 20 ns.

Clock, Reset, and Enable Signals

In this section...

“Driving Clocks, Resets, and Enables” on page 10-55

“Adding Signals Using Simulink Blocks” on page 10-55

“Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block” on page 10-56

“Driving Signals by Adding Force Commands” on page 10-58

Driving Clocks, Resets, and Enables

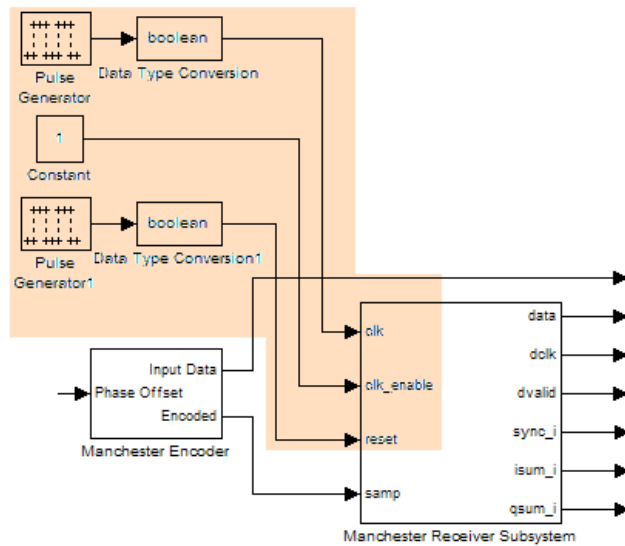
You can create rising-edge or falling-edge clocks, resets, or clock enable signals that apply internal stimuli to your model under cosimulation. You can add these signals by:

- “Adding Signals Using Simulink Blocks” on page 10-55
- “Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block” on page 10-56
- “Driving Signals by Adding Force Commands” on page 10-58
- Implementing these signals directly in HDL code. If your model is part of a much larger HDL design, you (or the larger model designer) may choose to implement these signals in the Verilog or VHDL files. However, that implementation exceeds the scope of this documentation; see an HDL reference for more information.

Adding Signals Using Simulink Blocks

Add rising-edge or falling-edge clocks, resets, or clock enable signals to your Simulink model using Simulink blocks. See the Simulink User Guide and Reference for instructions on adding blocks to a model.

In the following example excerpt, the shaded area shows a clock, a reset, and a clock enable signal as input to a multiple HDL Cosimulation block model. These signals are created using two Simulink data type conversion blocks and a constant source block, which connect to the HDL Cosimulation block labeled “Manchester Receiver Subsystem”.



Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block

Note For ModelSim and Xcelium users only.

When you specify a clock in your block definition, Simulink creates a rising-edge or falling-edge clock that drives the specified HDL signal.

Simulink attempts to create a clock that has a 50% duty cycle and a predefined phase that is inverted for the falling edge case. If applicable, Simulink degrades the duty cycle to accommodate odd Simulink sample times, with a worst case duty cycle of 66% for a sample time of $T=3$.

Whether you have configured the **Timescales** pane for relative timing mode or absolute timing mode, the following restrictions apply to clock periods:

- If you specify an explicit clock period, you must enter a sample time equal to or greater than 2 resolution units (ticks).
- If the clock period (whether explicitly specified or defaulted) is not an even integer, Simulink cannot create a 50% duty cycle, and therefore the HDL Verifier software creates the falling edge at

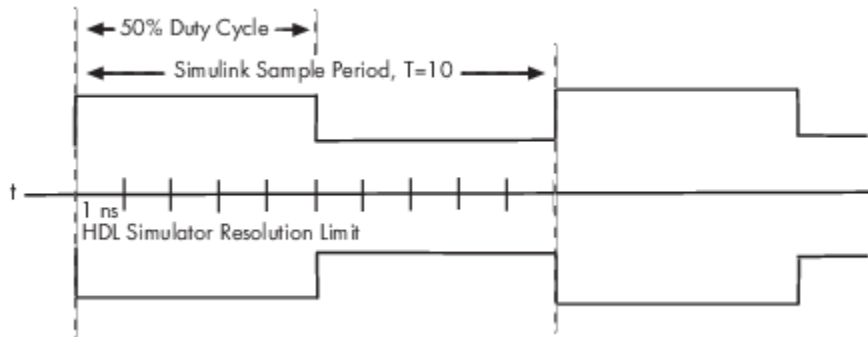
$$\text{clockperiod}/2$$

(rounded down to the nearest integer).

For more information on calculating relative and absolute timing modes, see “Defining the Simulink and HDL Simulator Timing Relationship” on page 10-45.

The following figure shows a timing diagram that includes rising and falling edge clocks with a Simulink sample time of $T=10$ and an HDL simulator resolution limit of 1 ns. The figure also shows that given those timing parameters, the clock duty cycle is 50%.

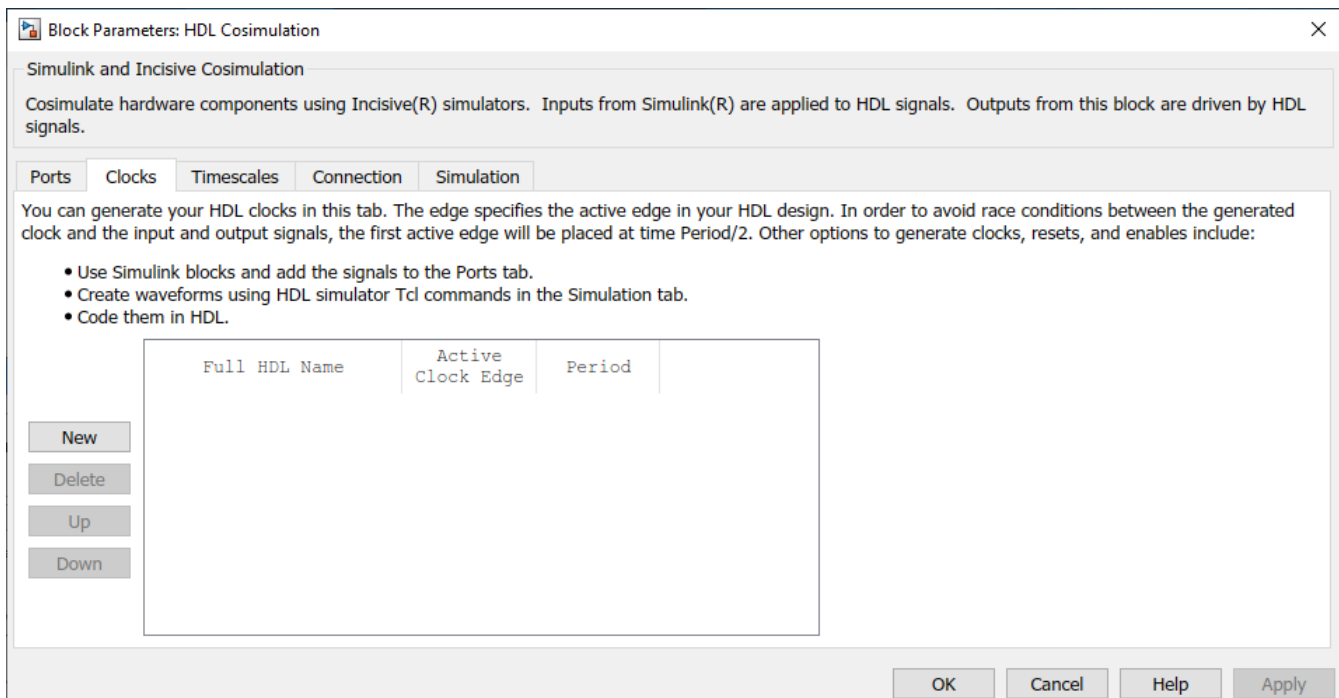
Rising Edge Clock



Falling Edge Clock

To create clocks, perform the following steps:

- 1 In the HDL simulator, determine the clock signal path names you plan to define in your block. To do so, you can use the same method explained for determining the signal path names for ports in step 1 of "Map HDL Signals to Block Ports".
- 2 Select the **Clocks** tab of the Block Parameters dialog box. Simulink displays the dialog box as shown in the next figure (example shown for use with Xcelium).



- 3 Click **New** to add a new clock signal.
- 4 Edit the clock signal path name directly in the table under the **Full HDL Name** column by double-clicking the default clock signal name (`/top/clock`). Then, specify your new clock using HDL simulator path name syntax. See "Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation" on page 6-10.

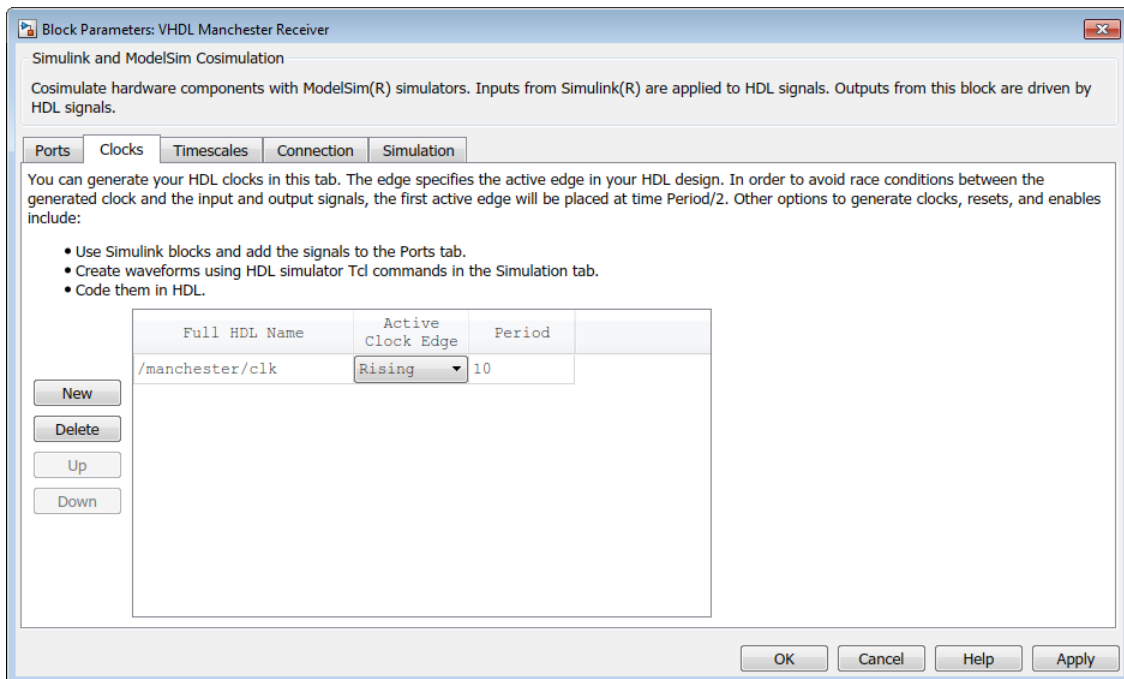
The HDL simulator does not support vectored signals in the **Clocks** pane. Signals must be logic types with 1 and 0 values.

- 5 To specify whether the clock generates a rising-edge or falling edge signal, select **Rising** or **Falling** from the **Active Clock Edge** list.
- 6 The **Period** field specifies the clock period. Accept the default (2), or override it by entering the desired clock period explicitly by double-clicking in the **Period** field.

Specify the **Period** field as an even integer, with a minimum value of 2.

- 7 When you have finished editing clock signals, click **Apply** to register your changes with Simulink.

The following dialog box defines the rising-edge clock `clk` for the HDL Cosimulation block, with a default period of 2 (example shown for use with Xcelium).



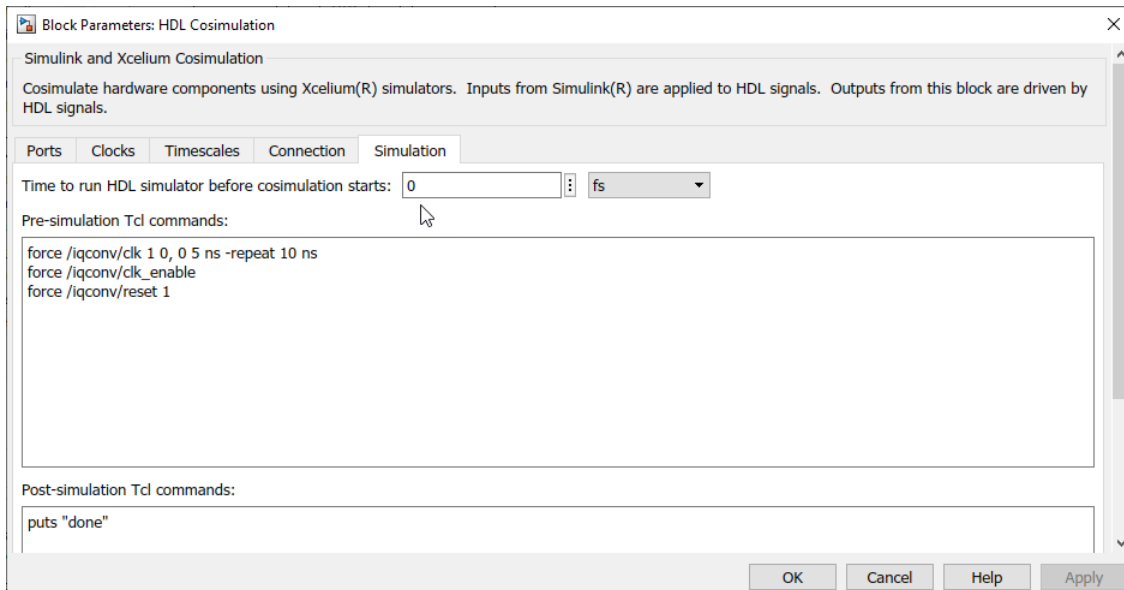
Driving Signals by Adding Force Commands

You can drive clocks, resets, and enable signals in either of two ways:

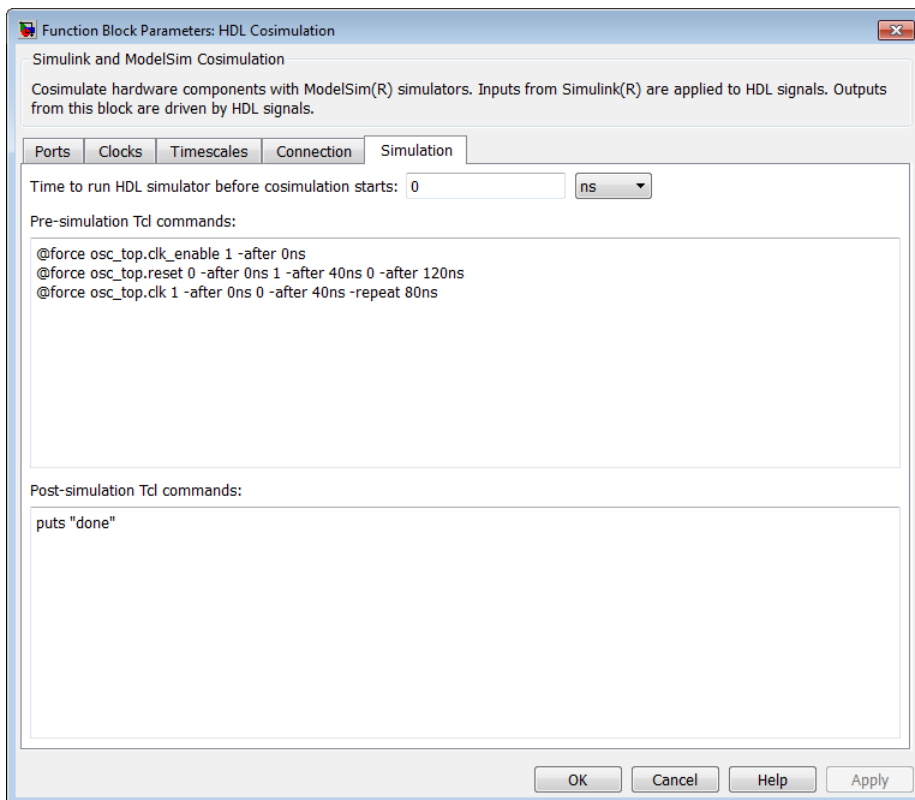
- By adding force commands to the **Simulation** pane (ModelSim and Xcelium users only)
- By driving signals with one of the HDL Verifier HDL simulator launch commands (`vsim` or `ncLaunch`) and the force command

Examples: force Command entered in HDL Cosimulation block Simulation Pane

The following is an example of entering force commands in the **Simulation** pane of the HDL Cosimulation block for use with Xcelium:



The following is an example of entering force commands in the **Simulation** pane of the HDL Cosimulation block for use with ModelSim:



Examples: force Command used with HDL Verifier HDL Simulator Launch Command

vsim function and force command (ModelSim users):

```
vsim('tclstart', {'force /iqconv/clk 1 0, 0 5 ns -repeat 10 ns ',  
  'force /iqconv/clk_enable 1', 'force /iqconv/reset 1'});
```

nclaunch function and force command (Xcelium users):

```
nclaunch('tclstart', ['-input "@force osc_top.clk_enable 1 -after 0ns"',  
  '-input "@force osc_top.reset 0 -after 0ns 1 -after 40ns 0 -after 120ns"',  
  '-input "@force osc_top.clk 1 -after 0ns 0 -after 40ns -repeat 80ns"']);
```

TCP/IP Socket Ports

When you specify a TCP/IP socket port, choose an available port or service name (alias). If you are uncertain what port is available, use `hdldaemon('socket', 0)` to get an available port. If you choose a port that is already in use, you will get an error message.

When you set up communication between computers, you must specify the host name as well as the port name on the client side.

Examples:

```
<port-num>                4449
<port-alias>              matlabservice
<host>:<port-num>         compa:4449
<port-alias>@<host-ia>   matlabservice@123.34.55.23
```

Note that TCP/IP port filtering on either the client or server side can cause the HDL Verifier interface to fail to make a connection. If you get an error, remove filtering (see OS user guide), or try a different port.

See Also

Functions

`nclaunch` | `vsim` | `hdldaemon`

Blocks

HDL Cosimulation

Related Examples

- “Set Up MATLAB-HDL Simulator Connection” on page 1-2

FPGA-in-the-Loop

About FPGA-in-the-Loop Simulation

FPGA-in-the-Loop Simulation

In this section...
“What is FPGA-in-the-Loop Simulation?” on page 11-2
“What You Need To Know” on page 11-4

What is FPGA-in-the-Loop Simulation?

- “Overview” on page 11-2
- “Communication Channel” on page 11-3
- “Downstream Workflow Automation” on page 11-3

Overview

FPGA-in-the-loop (FIL) simulation provides the capability to use Simulink or MATLAB software for testing designs in real hardware for any existing HDL code. The HDL code can be either manually written or software generated from a model subsystem.

You must have HDL code to perform FIL simulation. There are two FIL workflows:

- You have existing HDL code (FIL wizard).

Note The FIL wizard uses any synthesizable HDL code including code automatically generated from Simulink models by HDL Coder software

- You have MATLAB code or a Simulink model *and* an HDL Coder license (HDL workflow advisor).

Note When you use FIL in the Workflow Advisor, HDL Coder uses the loaded design to create the HDL code.

No matter which workflow you choose, FIL performs the following processes when it creates the block or System object:

- Generates a FIL block or FIL System object that represents the HDL code
- Provides synthesis, logical mapping, place-and-route (PAR), programming file generation, and communications channel.
- Loads the design onto an FPGA

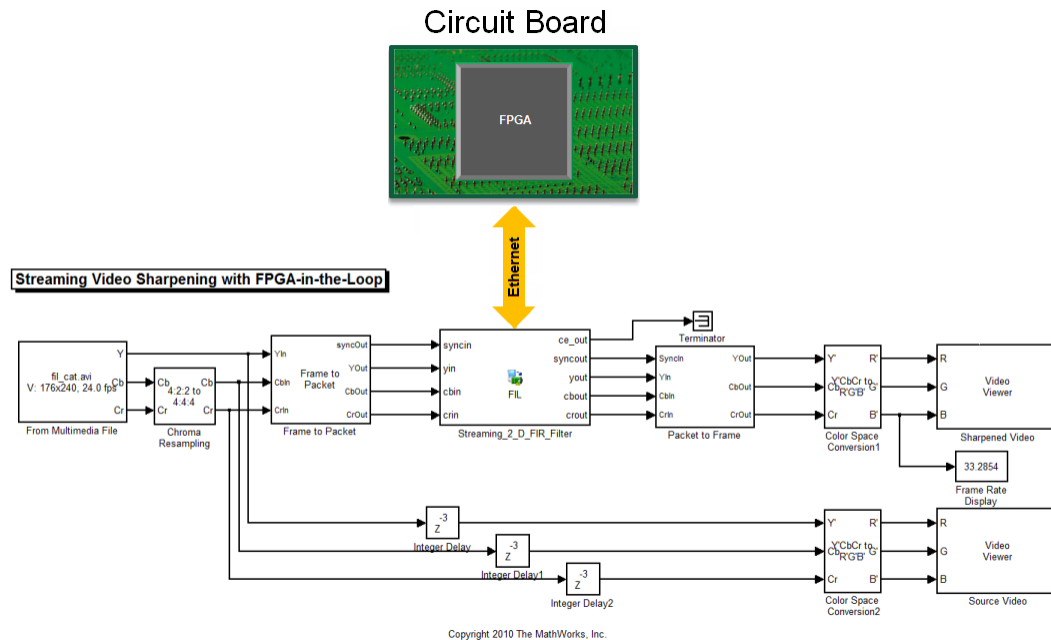
All these capabilities are designed for a particular board and tailored to your RTL code.

As part of FIL simulation, the block or System object and your model or application:

- Transmits data from Simulink or MATLAB to the FPGA
- Receives data from the FPGA
- Exercises the design in a real environment

FIL Communications

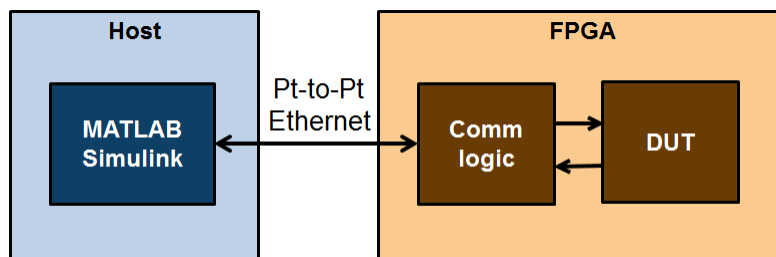
The following figure demonstrates how HDL Verifier communicates between Simulink and the FPGA board using FIL simulation.



Note HDL Verifier assumes that there is only one download cable connected to the host computer, and that the FPGA programming software can automatically detect this connection. If not, use FPGA programming software to program your FPGA with the correct options.

System-Level View

All DUT I/Os are routed to Simulink through the FIL communication logic.



Communication Channel

FIL provides the communication channel for sending and receiving data between Simulink and the FPGA. This channel can be a JTAG, Ethernet, or PCI Express® connection. Communication between Simulink and the FPGA is strictly synchronized to provide a reliable verification environment.

Downstream Workflow Automation

To create the FIL programming file, the software performs the following tasks:

- Generates HDL code for the specified DUT and creates an ISE project.
- Along with your FPGA design software, synthesizes, maps, places and routes, and creates a programming file for the FPGA.
- Downloads the programming file to the FPGA on the development board through the normal configuration connection. Typically, that connection is a serial line over a USB cable (see the board user guide for how to make this connection).
 - For FIL simulation blocks, clicking **Load** on the FIL block mask initiates the programming file download.
 - For FIL simulation System objects, issuing the `programFPGA` method initiates the programming file download.

What You Need To Know

For FIL simulation, you must have the following items or information ready:

- For FIL wizard:
 - Provide HDL code (either manually written or software generated) for the design you intend to test.
 - Select HDL files and specify the top-level module name.
 - Review port settings and make sure the FIL wizard identified input and output signals and signal sizes as expected.
 - If you are using Simulink, provide a Simulink model ready to receive the generated FIL block.
- For HDL Workflow Advisor:

You can generate code and run FIL from any suitable Simulink model. Follow the workflow for `FPGA-in-the-Loop`. See “FIL Simulation with HDL Workflow Advisor for Simulink” on page 14-2. For MATLAB code, see the workflow described in “FIL Simulation with HDL Workflow Advisor for MATLAB” on page 14-8.

Prepare FPGA Connection

- “FPGA-in-the-Loop Simulation Workflows” on page 12-2
- “Download FPGA Board Support Package” on page 12-3
- “Set Up FPGA Design Software Tools” on page 12-4
- “Guided Hardware Setup” on page 12-5
- “Manual Hardware Setup” on page 12-13
- “Prepare DUT For FIL Interface Generation” on page 12-18

FPGA-in-the-Loop Simulation Workflows

You must have HDL code to perform FIL simulation. There are two FIL workflows:

- You have existing HDL code (FIL wizard).

Note The FIL wizard uses any synthesizable HDL code including code automatically generated from Simulink models by HDL Coder software

- You have MATLAB code or a Simulink model *and* an HDL Coder license (HDL workflow advisor).

Note When you use FIL in the Workflow Advisor, HDL Coder uses the loaded design to create the HDL code.

For either workflow, the first three steps are the same:

- 1 “Download FPGA Board Support Package” on page 12-3 or create custom board definition files for use with FIL (see “FPGA Board Customization” on page 16-2)
- 2 “Prepare DUT For FIL Interface Generation” on page 12-18
- 3 “Set Up FPGA Design Software Tools” on page 12-4

For the next step, click the link for the workflow you are going to follow:

- If you have existing HDL code, select block or System object generation using the FIL wizard:
 - “Block Generation with the FIL Wizard” on page 13-2
 - “System Object Generation with the FIL Wizard” on page 13-12
- If you need the HDL workflow advisor to generate HDL code, select block or System object generation using HDL workflow advisor:
 - “FIL Simulation with HDL Workflow Advisor for Simulink” on page 14-2
 - “FIL Simulation with HDL Workflow Advisor for MATLAB” on page 14-8

Note To use the HDL Coder HDL workflow advisor for Simulink to generate a FIL interface, you must have an HDL Coder license.

Download FPGA Board Support Package

In this section...

“HDL Verifier Support Packages for FPGA Boards” on page 12-3

“Install with Connection to Internet” on page 12-3

“Install Support Package Offline” on page 12-3

HDL Verifier Support Packages for FPGA Boards

The FPGA board support packages contain the definition files for all the supported boards for FPGA-in-the-loop (FIL) simulation, FPGA data capture, or AXI manager. Before you can use one of the features that connects to the FPGA board, download at least one of the FPGA board support packages or customize your own board definition file.

To download FPGA board support packages, select the MATLAB **HOME** tab and click **Add-Ons**. To customize a board definition file, see FPGA Board Manager on page 16-2. You can also install it offline.

Install with Connection to Internet

After you have downloaded an FPGA board support package, you can use the FPGA verification features from HDL Verifier, such as “FPGA-in-the-Loop Simulation” on page 11-2, “AXI Manager”, or “FPGA Data Capture”.

Install Support Package Offline

To install the board support packages without an Internet connection, first download the packages on a computer that *does* have an Internet connection.

- 1 On the computer with the Internet connection, start MATLAB.
- 2 Select **Add-Ons > Get Hardware Support Packages**.
- 3 Select **Download from Internet**.
- 4 Select the board support package you want to download. Click **Next** and follow the installer prompts. The last screen displays where you can find the downloaded file.
- 5 Copy the downloaded file to a shared network drive or removable media, such as a USB drive.

Then, on the computer where you want to install the board support packages:

- 1 Copy the downloaded file to the host computer.
- 2 Start MATLAB.
- 3 Select **Add-Ons > Get Hardware Support Packages**.
- 4 Select **Install from folder**.
- 5 Specify the path to the folder where the downloaded files are located.
- 6 Click **Next**, and follow the installer prompts to install the board support package. If you do actually have an Internet connection, you are prompted to log in to your MathWorks® account.

Set Up FPGA Design Software Tools

In this section...

“Xilinx Software” on page 12-4
 “Intel Software” on page 12-4
 “Microchip Software” on page 12-4

Xilinx Software

Set up your system environment for accessing Xilinx tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path.

- Windows with ISE —

```
hdlsetuptoolpath...
('ToolName','Xilinx ISE','ToolPath','C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64')
```

This example assumes that the Xilinx ISE design suite is installed at `C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64`.

- Linux with Vivado —

```
hdlsetuptoolpath...
('ToolName','Xilinx Vivado','ToolPath','local/Xilinx/Vivado/bin')
```

This example assumes that the Xilinx Vivado software is installed at `local/Xilinx/Vivado/bin`.

Intel Software

Set up your system environment for accessing from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath...
('ToolName','Altera Quartus II','ToolPath','C:\Altera\12.0\quartus\bin64')
```

This example assumes that the Intel FPGA design software is installed at `C:\Altera\12.0\quartus\bin64`.

Microchip Software

To set up your Microchip Software environment, first add the FIL IP to Libero® SoC (or Libero SoC Polarfire®) Mega Vault. See “Setup and Configuration” (HDL Verifier Support Package for Microchip FPGA Boards) for additional instructions.

Next, set up your system environment for accessing from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath('ToolName','Microchip Libero SoC',...
'ToolPath','C:\Microsemi\Libero_SoC_v12.0\Designer\bin\libero.exe')
```

This example assumes that the Microchip Libero SoC (or Libero SoC Polarfire) software is installed at `C:\Microsemi\Libero_SoC_v12.0\Designer\bin`.

Guided Hardware Setup

In this section...

“Select Board and Interface for Use with FPGA Verification” on page 12-5
“Connection Requirements” on page 12-5
“Connection Setup” on page 12-6
“Configure Network Card on Host” on page 12-9
“Select a Drive and Load Firmware” on page 12-9
“PCI Express Driver Installation” on page 12-10
“Set Jumpers” on page 12-10
“Connect Hardware” on page 12-11
“Verify Setup” on page 12-11
“Open the Example” on page 12-12

Select Board and Interface for Use with FPGA Verification

The guided hardware setup for FPGA and SoC boards helps you get started with FPGA-in-the-loop (FIL), FPGA data capture, or AXI manager more quickly. Before you run the guided setup, make sure you have all the required hardware ready and any required third-party tools already installed.

The guided setup starts automatically during support package download and installation. After the support package you selected is installed, you are prompted to select your board name and the interface you want to use with this board. You can select only interfaces supported by the hardware board. A PCI Express connection is not supported on Linux.

Note To rerun the support package setup at any time:

On the MATLAB **Home** tab, in the **Environment** section, select **Help > Check for Updates**.

Connection Requirements

The guided setup wizard displays a checklist of the hardware requirements. Confirm that you have the hardware required to complete the setup process.

Note Do not connect to the board or turn it on until you are prompted at a later step.

Ethernet

- FPGA or SoC development board
- USB-JTAG cable with installed vendor software (Vivado or Quartus®)
- Ethernet cable
- Dedicated Gigabit network interface card (NIC) or a USB 3.0 Gigabit Ethernet adapter dongle
- Power supply adapter (if the board requires one)

Ethernet on Zynq SoC Board

- Zynq® SoC board
- Ethernet cable
- Dedicated Gigabit NIC or a USB 3.0 Gigabit Ethernet adapter dongle
- Memory secure digital (SD) card and SD card reader
- Power supply adapter (if the board requires one)

JTAG

- FPGA or SoC development board
- USB-JTAG cable with installed vendor software (Vivado or Quartus)
- Power supply adapter (if the board requires one)

PCI Express

HDL Verifier supports a PCI Express connection for Windows operating systems only.

- FPGA development board
- USB-JTAG cable with installed vendor software (Vivado or Quartus)
- A PCI Express slot and available space on the motherboard
- Power supply adapter (if the board requires one)

Connection Setup

The guided setup wizard displays the setup steps for the selected interface. Follow these steps to set up your hardware board with the selected interface.

Ethernet

- 1** Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2** Connect the AC power cord to the power plug, and plug the power supply adapter cable into the hardware board.
- 3** Use the crossover Ethernet cable to connect the Ethernet connector on the hardware board directly to the Ethernet adapter on your computer.
- 4** Use the JTAG download cable to connect the hardware board to the computer.
- 5** Make sure that all the jumpers on the hardware board are in the factory default position.
- 6** Turn the power switch of the hardware board on.

Ethernet on Zynq SoC Board

- 1** Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2** Follow the guided setup to:
 - a** Configure the network interface card in the host computer. See “Configure Network Card on Host” on page 12-9.

- b** Copy the compatible SD card image files for the hardware board to an SD card drive path. See “Select a Drive and Load Firmware” on page 12-9.
- c** Configure the jumpers on the hardware board. See “Set Jumpers” on page 12-10.
- d** Connect the hardware board. See “Connect Hardware” on page 12-11.

JTAG

- 1** Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2** Connect the AC power cord to the power plug, and plug the power supply adapter cable into the hardware board.
- 3** Use the JTAG download cable to connect the hardware board to the computer.
- 4** Make sure that all the jumpers on the hardware board are in the factory default position.
- 5** Turn the power switch of the hardware board on.

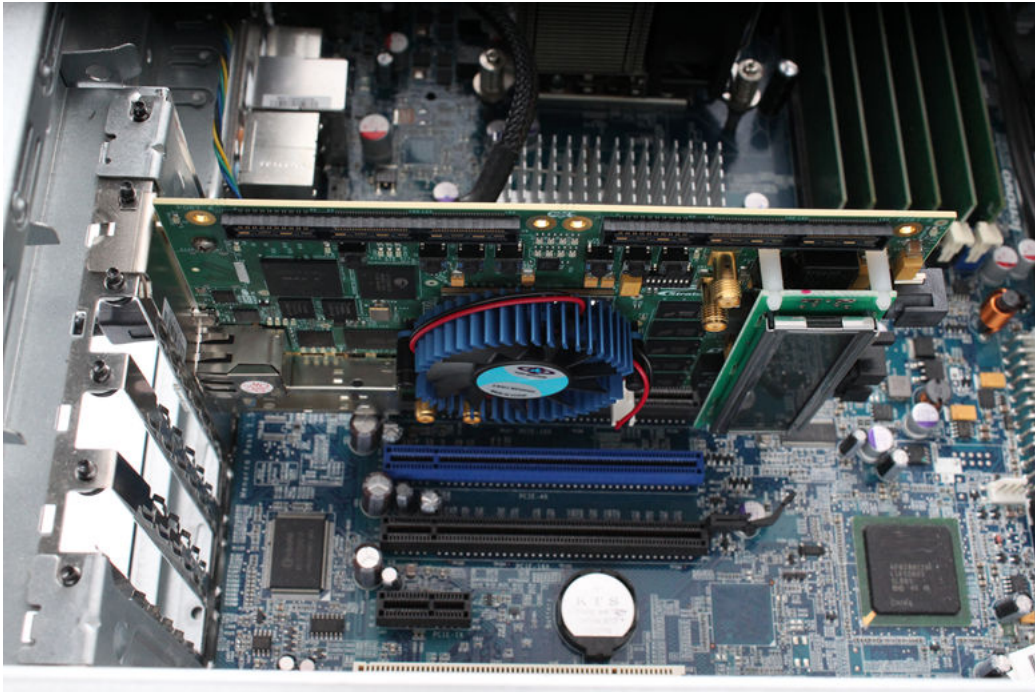
PCI Express

- 1** Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2** Select the maximum number of PCI Express lanes that the board supports. For details, refer to the user manual for the board.

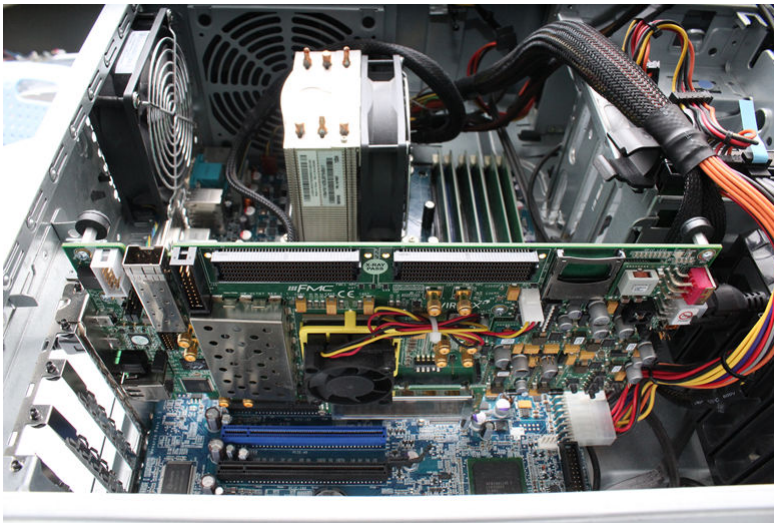
Supported Board	PCI Express Setup	Documentation
DSP Development Kit, Stratix® V Edition	Set the three switches (PCIE_PRSNT2nx1, x4, x8) in dip switch SW6 to ON. This setting selects 8-lane PCIe (default board setting).	https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/stratix/v-gs.html
Cyclone® V GT FPGA Development Kit	Set the two switches(PCIE_x1, x4) in dip switch SW3 to ON. This setting selects 4-lane PCIe (default board setting).	https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/cyclone/v-gt.html
Kintex®-7 KC705	Set jumper J32 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (default board setting).	https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html
Virtex®-7 VC707	Set jumper J49 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (<i>not</i> the default board setting).	https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html

- 3** Turn the host computer off.
- 4** Install the hardware board in a PCI Express slot inside the host computer.

This figure shows the Stratix V board installed in a host computer. This installation applies to all supported Intel VC boards.



This figure shows the VC707 board installed in a host computer. The power cable is on the right. This installation applies to all supported Xilinx boards.



- 5 For Xilinx boards, plug the external power supply into the wall outlet. Then, plug the power supply adapter cable into the hardware board.
Intel boards do not use an external power supply.
- 6 Connect the JTAG cable to the hardware board and the computer. When you use a PCI Express connection, the JTAG cable is still required to program the FPGA.



- 7 Turn the power switch of the hardware board on.
- 8 Start up the host computer.

Configure Network Card on Host

This step is required only when you select the Ethernet interface.

If you have already configured the network card, you can skip this step.

Specify the NIC on the host computer that you want to use with the hardware board. If you have only one NIC, you must disconnect from the Internet while using the NIC. In this case, consider using a USB 3.0 Gigabit Ethernet adapter dongle. If you add a NIC or a USB 3.0 Gigabit Ethernet adapter dongle during this setup step, click **Refresh** to see the new hardware in the list.

Leave the IP address for the NIC as the default. Alternatively, specify the IP address in dotted quad format, for example, 192.168.0.1.

Select a Drive and Load Firmware

This step is required only when you select the Ethernet interface on a Zynq SoC board.

Next, the installer must write an FPGA image to an SD card. This FPGA image is included with the support package. The image includes the embedded software and the FPGA programming file necessary for using the hardware board as an I/O peripheral.

- 1 Insert a 4 GB or larger SD card into the card reader on the host computer. The card must be in FAT32 format. Select the appropriate drive from the list. If you have already downloaded the FPGA image, skip this step.



Note Unlock the SD card before downloading the firmware image to the card. Keep the card unlocked while the card is in the Zynq board card reader.

- 2 Write the FPGA image to the SD card. In the guided setup, select the location of the SD drive containing the card, then click **Next**. On the next screen, to copy the programming file from the host computer to the SD card, click **Write**. This process erases any existing data on the card.

PCI Express Driver Installation

This step is required only when you select the PCI Express interface.

If you have already installed the PCI Express drivers, you can skip this step.

Install the PCI Express drivers before you use FIL, FPGA data capture, or AXI manager with a PCI Express connection. This step performs the driver installation for you. The process can take 10 or more minutes to install, and might require system administrator privileges.

You can let the support package setup install the drivers now, or you can choose to perform the setup again later. To run the support package setup, on the MATLAB **Home** tab, in the **Environment** section, select **Help > Check for Updates**.

Set Jumpers

This step is required only when you select the Ethernet interface on a Zynq SoC board or the PCI Express interface.

Configure the jumpers on the Zynq SoC board so that you can use it as a peripheral device. These jumper settings make it so that the board starts up from the SD card. Make sure that the board is turned off.

The jumper settings are different for each board. To learn more about the jumper settings on the supported Zynq SoC boards, see “Set Jumper Switches” (HDL Verifier Support Package for Xilinx FPGA Boards).

Connect Hardware

This step is required only when you select the Ethernet interface on a Zynq SoC board.

Follow these instructions for connecting the hardware. The guided setup wizard provides labeled pictures of the steps for each board. See “Connect Hardware” (HDL Verifier Support Package for Xilinx FPGA Boards).

- 1 Remove the SD card from the host computer and insert it into the hardware board.
- 2 Connect an Ethernet cable to the board. Connect the other end of the Ethernet cable to the selected NIC.
- 3 Connect the power cable.
- 4 Turn the power on.

Verify Setup

You can verify the hardware setup for Ethernet and JTAG connections. This step runs the tests to verify the connection between the host computer and the hardware board. Before you run the test, make sure that:

- 1 You have installed the appropriate vendor tool and that the tool is on the MATLAB path. See “Set Up FPGA Design Software Tools” on page 12-4.
- 2 The board is turned on.

This step runs the following tests to verify the connection for the selected interface.

Ethernet

- 1 Generate an FPGA programming file for your hardware board.
- 2 Program the FPGA.
- 3 Detect Ethernet connection.

Ethernet on Zynq SoC Board

- 1 Verify the IP address configuration on the host computer.
- 2 Verify the Ethernet connection between the host computer and the hardware board.
- 3 Read and write the memory locations on the hardware board using AXI manager.

JTAG

- 1 Generate an FPGA programming file for your board.
- 2 Program the FPGA.
- 3 Perform a FIL cosimulation with your board.

If the connection is not successful, the most common reasons are that the board is not connected properly or it is not turned on. Check the cable connections and power switch and try again.

Open the Example

When the installer completes your hardware setup, you can exit the installer or open the examples to get started.

See Also

More About

- “FPGA-in-the-Loop Simulation” on page 11-2
- “Data Capture Workflow” on page 17-2
- “Set Up for AXI Manager” on page 18-2

Manual Hardware Setup

You can use FPGA-in-the-loop (FIL), FPGA data capture, or AXI manager via Ethernet, JTAG, and PCI Express connections. Some FPGA boards support multiple connection methods, and some boards support only one method. Choose setup instructions based on the connection method you plan to use for FIL simulation.

When possible, use the guided setup. To run the support package setup, or modify your installation:

On the MATLAB **Home** tab, in the **Environment** section, select **Help > Check for Updates**.

For more about the guided setup, see “Guided Hardware Setup” on page 12-5.

Step 1. Set Up FPGA Development Board

JTAG or Ethernet Connection

- 1 Make sure that the board power switch is OFF during these setup steps.
- 2 Make sure that all jumpers on the FPGA development board are in the factory default position.
- 3 Connect the AC power cord to the power plug.
- 4 Plug the power supply adapter cable into the FPGA development board.
- 5 Connect the JTAG cable to the FPGA development board and the computer. When you use Ethernet for FIL simulation, the JTAG cable is still required to program the FPGA.
- 6 If you plan to use an Ethernet connection for FIL simulation, connect the crossover Ethernet cable between the FPGA development board and the Ethernet adapter on your computer.
- 7 Turn on the power switch on the FPGA board.

PCI Express Connection

- 1 Make sure that the board power switch is OFF during these setup steps.
- 2 Select the maximum number of PCI Express (PCIe) lanes supported by the board. to Refer to the user manual for the board for details.

Supported Board	PCI Express Setup	Documentation
DSP Development Kit, Stratix V Edition	Set the three switches (PCIE_PRSNT2nx1, x4, x8) in dip switch SW6 to ON. This setting selects 8-lane PCIe (default board setting).	https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/stratix/v-gs.html
Cyclone V GT FPGA Development Kit	Set the two switches(PCIE_x1, x4) in dip switch SW3 to ON. This setting selects 4-lane PCIe (default board setting).	https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/cyclone/v-gt.html
Kintex-7 KC705	Set jumper J32 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (default board setting).	https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html

Supported Board	PCI Express Setup	Documentation
Virtex-7 VC707	Set jumper J49 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (<i>not</i> the default board setting).	https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html

- 3 Turn off the host computer.
- 4 Install the FPGA development board in a PCI Express slot inside host computer.
- 5 For Xilinx boards, plug the external power supply into the wall outlet. Then plug the power supply adapter cable into the FPGA development board.

Intel boards do not use an external power supply.

- 6 Connect the JTAG cable to the FPGA development board and the computer. When you use PCI Express for FIL simulation, the JTAG cable is still required to program the FPGA.
- 7 Turn on the power switch on the FPGA board.
- 8 Start up the host computer.

Step 2. Set Up Board Connection

HDL Verifier assumes that there is only one download cable connected to the host computer, and that the FPGA programming software can automatically detect this connection. If not, use FPGA programming software to program your FPGA with the correct options.

- “JTAG Connection” on page 12-14
- “Ethernet Connection” on page 12-14
- “PCI Express Connection” on page 12-17

JTAG Connection

Intel

- 1 Install USB Blaster I or II cable driver.
- 2 When using Linux operating systems: The Quartus II library must be present in LD_LIBRARY_PATH *before* you start MATLAB. Prepend the Linux distribution library path before the Quartus II library on LD_LIBRARY_PATH. For example, /lib/x86_64-linux-gnu:\$QUARTUS_LATEST/./linux64.

Xilinx

- 1 For Linux operating systems: Install Digilent® Adept2.

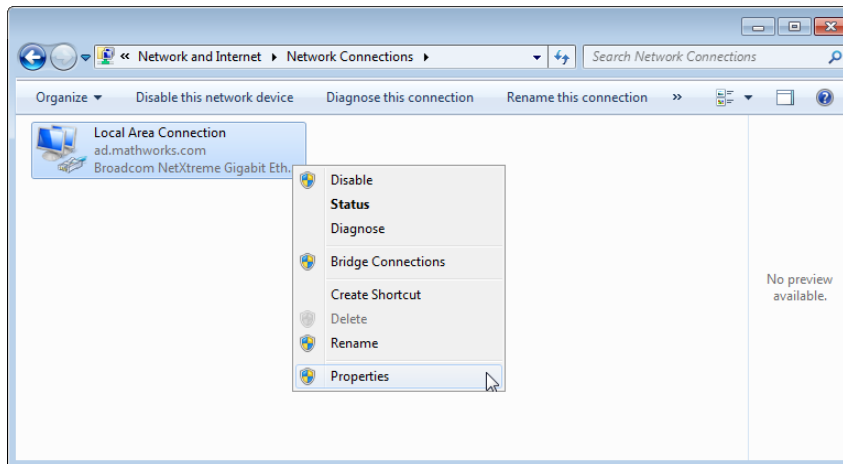
Ethernet Connection

Follow these instructions to set up a Gigabit Ethernet network adapter on your computer for FIL simulation.

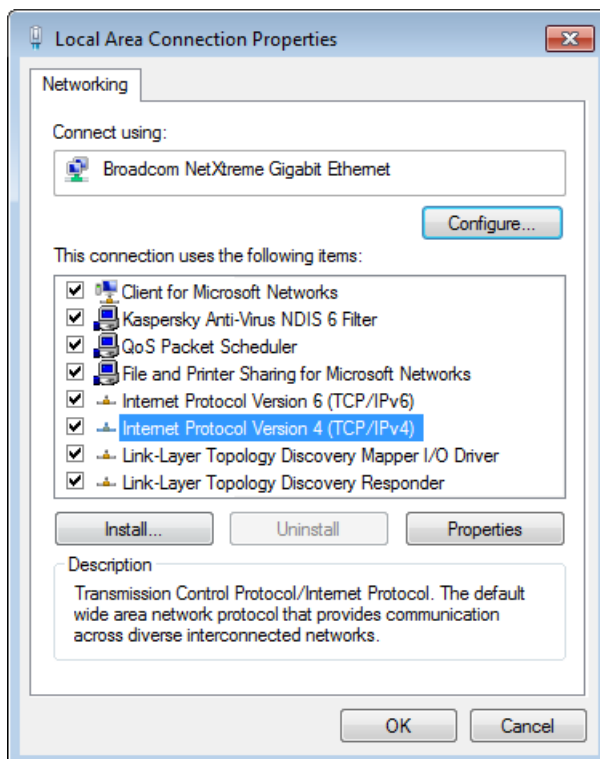
Windows 7 Setup

- 1 Open the Control Panel and type "view network connections" in the search bar. Select **View network connections** in the search results.

- Right-click the connection icon to your FPGA development board, and select Properties from the pop-up menu.

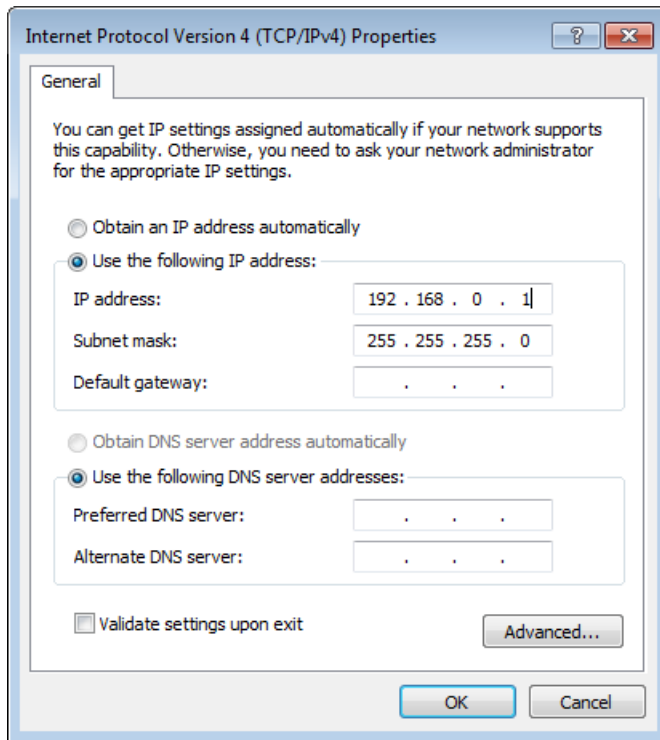


- Under **This connection uses the following items**, select **Internet Protocol Version 4 (TCP/IPv4)**, and click **Properties**.



- Select **Use the following IP address** and set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This value indicates your host computer address. Set the **Subnet mask** to 255.255.255.0.

The figure shows an example TCP/IP configuration.



- 5 Click **OK** to exit TCP/IP Properties.
- 6 Click **Close** to exit Local Area Connection Properties.

Windows Vista Setup

- 1 Open the Control Panel.
- 2 Click Network and Sharing Center, and then click Manage network connections.
- 3 Right-click the connection icon to your FPGA development board, and select Properties from the pop-up menu.
- 4 Under **This connection uses the following items**, select **Internet Protocol Version 4 (TCP/IPv4)**, and click **Properties**.
- 5 Select **Use the following IP address** and set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This value indicates your host computer address. Set the **Subnet mask** to 255.255.255.0.
- 6 Click **OK** to exit TCP/IP Properties.
- 7 Click **Close** to exit Local Area Connection Properties.

Windows XP Setup

- 1 Open the Control Panel.
- 2 Open Network connections.
- 3 Right-click the connection icon to your FPGA development board, and select Properties from the pop-up menu.
- 4 Under **This connection uses the following items**, select **Internet Protocol (TCP/IP)**, and click **Properties**.

- 5 Select **Use the following IP address** and set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This value indicates your host computer address. Set the **Subnet mask** to 255.255.255.0.
- 6 Click **OK** to exit TCP/IP Properties.
- 7 Click **Close** to exit Local Area Connection Properties.

Linux Setup

Use the `ifconfig` command to set up your local address. For example:

```
% ifconfig eth1 192.168.0.1
```

In this example, `eth1` is the second Ethernet adapter on the Linux computer. Check your system to determine which Ethernet adapter is connected to the FPGA development board. This command sets the local IP address to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100.

PCI Express Connection

FIL over PCI Express connection is supported only for 64-bit Windows operating systems.

- 1 Install the PCI Express drivers for your board using the FPGA board support package installer.
- 2 After you program your FPGA development board, restart your computer. The operating system automatically detects the new PCI Express connection. See "Step 9: Integrate and Simulate" > "Load Programming File onto FPGA" > "PCI Express Connection" under "Block Generation with the FIL Wizard" on page 13-2 or "System Object Generation with the FIL Wizard" on page 13-12.

Prepare DUT For FIL Interface Generation

In this section...
“Files and Information Required for FIL Generation” on page 12-18
“Apply FIL System Object Requirements” on page 12-18
“Apply FIL Block Requirements” on page 12-21

Files and Information Required for FIL Generation

- “For FIL Wizard” on page 12-18
- “For HDL Workflow Advisor” on page 12-18

For FIL Wizard

Have the following items or information ready:

- Provide HDL code (either manually written or software generated) for the design you intend to test.
- Select HDL files and specify the top-level module name.
- Review port settings and make sure the FIL wizard identified input and output signals and signal sizes as expected.
- If you are using Simulink, provide a Simulink model ready to receive the generated FIL block.

Next Steps

- If you are creating a FIL System object, next go to “Apply FIL System Object Requirements” on page 12-18.
- If you are creating a FIL block, next go to “Apply FIL Block Requirements” on page 12-21.

For HDL Workflow Advisor

You can generate code and run FIL from any suitable Simulink model.

Next Steps

- If you are creating a FIL System object, next go to “Apply FIL System Object Requirements” on page 12-18.
- If you are creating a FIL block, next go to “Apply FIL Block Requirements” on page 12-21.

Apply FIL System Object Requirements

- “The FIL Process for System Objects” on page 12-18
- “HDL Code Considerations for FIL System Objects” on page 12-19
- “FIL-Specific Rules for System Objects” on page 12-21
- “MATLAB Code Considerations for FIL System Objects” on page 12-21

The FIL Process for System Objects

The FIL wizard and HDL Coder HDL Workflow Advisor each perform the following actions:

- Convert HDL code into System object inputs and outputs.
- Walk you through identifying: FPGA device, source files, I/O ports, and port info.
- Add logic to the device under test (DUT) to communicate with MATLAB.

Generally, this logic is small and has minimal impact on the fit of your design onto the FPGA.

- Create the programming file and a FIL System object.

Note If a design does not fit in the device or does not meet timing goals, the software may not create a programming file. In this situation, you may see a warning that the design does not meet the timing goals, but it still generates a programming file, or you may get an error and no programming file. Either change your design, or use a different development board.

When FIL interface generation is complete, you can use the method `programFPGA` to load the programming file to the FPGA board. You can also use this method to adjust runtime options and signal attributes.

When you are ready to begin, read through the following topics and make sure that your DUT adheres to the rules and guidelines described in each section:

- “HDL Code Considerations for FIL System Objects” on page 12-19
- “FIL-Specific Rules for System Objects” on page 12-21
- “MATLAB Code Considerations for FIL System Objects” on page 12-21

When you are finished with these sections, next go to either “System Object Generation with the FIL Wizard” on page 13-12 or “FIL Simulation with HDL Workflow Advisor for MATLAB” on page 14-8.

HDL Code Considerations for FIL System Objects

Follow these rules when using legacy or auto-generated HDL code for generating a FIL System object.

Category	Considerations
HDL files	All HDL names must be legal as defined in the VHDL 1993 standard.
Top-level design	<ul style="list-style-type: none"> • The top-level design must be VHDL or Verilog. • The top-level HDL file must contain an entity/module with the same name as the file name. • FIL block generation supports both combinatorial and sequential logic. For combinatorial logic, CLK, CLK_ENABLE, and RESET are not required.

Category	Considerations
Inputs and outputs	<ul style="list-style-type: none"> • Input and output ports must be of the following types: <ul style="list-style-type: none"> • <code>std_logic</code> (VHDL) • <code>std_logic_vector</code> (VHDL) • <code>Reg</code>, <code>wire</code> (Verilog) • Vector ports range must be: <ul style="list-style-type: none"> • Descending (e.g. <code>9 DOWNTO 0, 9:0</code>) • Literal. Use of generics (VHDL) or parameters (Verilog) is not supported. (e.g. <code>a DOWNTO b</code> or <code>a:b</code> is not supported) <p style="margin-left: 40px;">Descending <code>TO</code> syntax is not supported</p> • For Verilog, ports names must be lowercase. Module name must be lowercase, also. • All input and output ports must be included. • There must be at least one output port.
Clock	<ul style="list-style-type: none"> • Sequential HDL design must have only one clock at the top entity. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. • Name your clock signal <code>clock</code> or <code>clk</code>. If the clock is not named <code>clock</code> or <code>clk</code>, designate which signal is the clock signal in the FIL wizard. • Clock port must be 1-bit. For VHDL, it must be of type <code>std_logic</code>.
Reset	<ul style="list-style-type: none"> • The HDL design must have a reset to be able to reset the FPGA before simulation. • For sequential design, there must be only one reset. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. • Name your reset signal <code>reset</code> or <code>rst</code>. If the reset is not named <code>reset</code> or <code>rst</code>, designate which signal is the reset signal in the FIL wizard. • Reset port must be 1-bit. For VHDL, these ports must be of type <code>std_logic</code>.
Clock enable	<ul style="list-style-type: none"> • For sequential design, if you choose a clock enable, there must be only one. • Clock enable port must be 1-bit. For VHDL, these ports must be of type <code>std_logic</code>. • If you have a clock enable, name it one of the following: <code>clock_enable</code>, <code>clock_enb</code>, <code>clock_en</code>, <code>clk_enable</code>, <code>clk_enb</code>, <code>clk_en</code>, <code>ce</code>. If the clock enable is not named one of these names, designate which signal is the clock enable signal in the FIL wizard.
DUT entity	<p>All the ports at DUT level must specify a bit width. Using a variable as the bit width is not allowed.</p>
Clock edge	<p>Clock the DUT input and output ports by positive edge. Negative edge is not allowed.</p>

Category	Considerations
Non-supported data types	<ul style="list-style-type: none"> • Bidirectional ports • Arrays, record types
Non-supported constructs	<ul style="list-style-type: none"> • VHDL configuration statement • Verilog include files • Macros • Escaped names • Duplicated port names (Verilog)

FIL-Specific Rules for System Objects

FIL input and output data set limits	<ul style="list-style-type: none"> • Total input data size must be less than 1467 bytes. The input data size is the sum of the input bits rounded up to the nearest byte. • Output data size must also be less than 1467 bytes. The output data size is the sum of the output bits rounded up to the nearest byte.
Output frame size	Output frame size = Input frame size × OverclockingFactor / OutputDownsample

MATLAB Code Considerations for FIL System Objects

MATLAB compatibilities	<p>HDL Verifier FIL simulation supports only the following data types:</p> <ul style="list-style-type: none"> • Integer • Logical • Fixed point
------------------------	--

Apply FIL Block Requirements

- “The FIL Process for Blocks” on page 12-21
- “HDL Code Considerations for FIL Blocks” on page 12-22
- “Simulink Model Considerations for FIL Blocks” on page 12-23
- “FIL-Specific Rules for Blocks” on page 12-24

The FIL Process for Blocks

The FIL wizard and HDL Coder HDL Workflow Advisor each perform the following actions:

- Convert HDL code into block signals with timing applied.
- Walk you through identifying: FPGA device, source files, I/O ports, and port info.
- Add logic to the device under test (DUT) to communicate with Simulink.

Generally, this logic is small and has minimal impact on the fit of your design onto the FPGA.

- Create the programming file and a FIL simulation block.

Note If a design does not fit in the device or does not meet timing goals, the software may not create a programming file. In this situation, you may see a warning that the design does not meet the timing

goals, but it still generates a programming file, or you may get an error and no programming file. Either change your design, or use a different development board.

After FIL interface generation is complete, use the FIL block mask to load the programming file to the FPGA board. You can also adjust runtime options and signal attributes.

When you are ready to begin, read through the following topics and make sure that your DUT adheres to the rules and guidelines described in each section:

- “HDL Code Considerations for FIL Blocks” on page 12-22
- “Simulink Model Considerations for FIL Blocks” on page 12-23
- “FIL-Specific Rules for Blocks” on page 12-24

When you are finished with these sections, next go to “Block Generation with the FIL Wizard” on page 13-2 or “FIL Simulation with HDL Workflow Advisor for Simulink” on page 14-2.

HDL Code Considerations for FIL Blocks

Follow these rules when using legacy or auto-generated HDL code for generating a FIL block.

Category	Considerations
HDL files	All HDL names must be legal as defined in the VHDL 1993 standard.
Top-level design	<ul style="list-style-type: none"> • The top-level design must be VHDL or Verilog. • The top-level HDL file must contain an entity/module with the same name as the file name. • FIL block generation supports both combinatorial and sequential logic. For combinatorial logic, CLK, CLK_ENABLE, and RESET are not required.
Inputs and outputs	<ul style="list-style-type: none"> • Input and output ports must be of the following types: <ul style="list-style-type: none"> • std_logic (VHDL) • std_logic_vector (VHDL) • Reg, wire (Verilog) • Vector ports range must be: <ul style="list-style-type: none"> • Descending (e.g. 9 DOWNTO 0, 9:0) • Literal. Use of generics (VHDL) or parameters (Verilog) is not supported. (e.g. a DOWNTO b or a:b is not supported) <p style="margin-left: 40px;">Descending TO syntax is not supported</p> • For Verilog, ports names must be lowercase. Module name must be lowercase, also. • All input and output ports must be included. • There must be at least one output port.

Category	Considerations
Clock	<ul style="list-style-type: none"> Sequential HDL design must have only one clock at the top entity. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. Name your clock signal <code>clock</code> or <code>clk</code>. If the clock is not named <code>clock</code> or <code>clk</code>, designate which signal is the clock signal in the FIL wizard. Clock port must be 1-bit. For VHDL, it must be of type <code>std_logic</code>.
Reset	<ul style="list-style-type: none"> The HDL design must have a reset to be able to reset the FPGA before simulation. For sequential design, there must be only one reset. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. Name your reset signal <code>reset</code> or <code>rst</code>. If the reset is not named <code>reset</code> or <code>rst</code>, designate which signal is the reset signal in the FIL wizard. Reset port must be 1-bit. For VHDL, these ports must be of type <code>std_logic</code>.
Clock enable	<ul style="list-style-type: none"> For sequential design, if you choose a clock enable, there must be only one. Clock enable port must be 1-bit. For VHDL, these ports must be of type <code>std_logic</code>. If you have a clock enable, name it one of the following: <code>clock_enable</code>, <code>clock_enb</code>, <code>clock_en</code>, <code>clk_enable</code>, <code>clk_enb</code>, <code>clk_en</code>, <code>ce</code>. If the clock enable is not named one of these names, designate which signal is the clock enable signal in the FIL wizard.
DUT entity	All the ports at DUT level must specify a bit width. Using a variable as the bit width is not allowed.
Clock edge	Clock the DUT input and output ports by positive edge. Negative edge is not allowed.
Non-supported data types	<ul style="list-style-type: none"> Bidirectional ports Arrays, record types
Non-supported constructs	<ul style="list-style-type: none"> VHDL configuration statement Verilog include files Macros Escaped names Duplicated port names (Verilog)

Simulink Model Considerations for FIL Blocks

Follow these rules for integrating the FIL block into your Simulink model.

Category	Considerations
General model rules	<ul style="list-style-type: none"> • Use Single tasking solver mode (set with Configuration Parameters). HDL Verifier FIL does not support multitasking solver mode. • Choose discrete, fixed-step solvers or variable-step solvers. HDL Verifier FIL supports both types of solvers.
Incompatibilities with Simulink	<p>HDL Verifier FIL simulation currently does not support the following:</p> <ul style="list-style-type: none"> • Instantiation of the FIL block in a triggered subsystem • Instantiation of the FIL block in an asynchronous function-call subsystem • A continuous sample time • A nonzero sample time offset
Initialization	<p>RAM Initialization: Simulink starts from time 0 every time, which means the RAM in a Simulink model is initialized to zero for each run. However, this assumption is not true in hardware. RAM in the FPGA holds its value from the end of one simulation to the start of the next. If you have RAM in your design, the first simulation matches Simulink, but subsequent runs may not match. The workaround is to reload the FPGA bitstream before rerunning the simulation. To reload the bitstream, click the Load on the FIL block mask.</p>

FIL-Specific Rules for Blocks

FIL block settings rules	<ul style="list-style-type: none"> • The input frame size must be an integer multiple of the output frame size. • All signals must be of the same bit-width as their corresponding port in the hardware. • In frame mode, all inputs must have the same frame size and all outputs must have the same frame size (but possibly different from the inputs). • When processing as frames, all input signals must have the same sample times and all output signals must have the same sample times. The output sample time can be different from the input sample time. • When processing as samples, only scalars are supported. When processing as frames, only column vectors (N-by-1) are supported. • Supported data types are built-in data types and fixed-point data types. • The output frame size must be less than the input frame size. This requirement ensures that the output frame has enough data to drive a value at time 0. You can avoid this error by either decreasing the output frame size or sample time, or increasing the input frame size or sample time.
FIL byte size limit	<ul style="list-style-type: none"> • Total input data size must be less than 1467 bytes. The input data size is the sum of the input bits rounded up to the nearest byte. • Output data size must also be less than 1467 bytes. The output data size is the sum of the output bits rounded up to the nearest byte.

FIL Interface Generation and Simulation

- “Block Generation with the FIL Wizard” on page 13-2
- “System Object Generation with the FIL Wizard” on page 13-12

Block Generation with the FIL Wizard

In this section...

“Step 1: Set Up FPGA Design Software Tools” on page 13-2

“Step 2: Start FIL Wizard” on page 13-3

“Step 3: Set FIL Options for FIL Block” on page 13-3

“Step 4: Add HDL Source Files for FIL Block” on page 13-5

“Step 5: Verify DUT I/O Ports for FIL Block” on page 13-6

“Step 6: Specify Output Types for FIL Block” on page 13-7

“Step 7: Specify Build Options for FIL Block” on page 13-8

“Step 8: Initiate Build” on page 13-8

“Step 9: Integrate and Simulate” on page 13-9

Step 1: Set Up FPGA Design Software Tools

Xilinx Software

Set up your system environment for accessing Xilinx tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path.

- Xilinx ISE —

```
hdlsetuptoolpath('ToolName','Xilinx ISE','ToolPath','C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64')
```

This example assumes that the Xilinx ISE design suite is installed at `C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64`.

- Xilinx Vivado —

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\apps\Vivado\2013.4-mw-0\Win\bin\vivado')
```

This example assumes that Xilinx Vivado is installed at `C:\apps\Vivado\2013.4-mw-0\Win\bin\vivado`.

Intel Software

Set up your system environment for accessing Intel tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\Altera\12.0\quartus\bin64')
```

This example assumes that the Intel FPGA design software is installed at `C:\Altera\12.0\quartus\bin64`.

Microchip Software

Set up your system environment for accessing Microchip tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath('ToolName','Microchip Libero SoC','ToolPath','C:\Microsemi\Libero_SoC_v12.0\Designer\bin\libero.exe')
```

This example assumes that the Microchip FPGA design software is installed at `C:\Microsemi\Libero_SoC_v12.0\Designer\bin`.

Step 2: Start FIL Wizard

Open the FPGA-in-the-loop wizard by selecting one of the following invocation methods:

- In the MATLAB command window, type the following:

```
>> filWizard
```
- In the Simulink toolstrip, on the **Apps** tab, under **Code Verification, Validation and Test** click **FIL Wizard**.

To restore a previous session, use this command:

```
filWizard('./Subsystem_fil/Subsystem_fil.mat')
```

Step 3: Set FIL Options for FIL Block

In the **FIL Options** page:

- 1 FIL Simulation:** Select Simulink.
- 2 Board Name:** Select an FPGA development board. If you have not yet downloaded an HDL Verifier FPGA board support package, see “Download FPGA Board Support Package” on page 12-3. (If you do not see any boards listed, then you have not yet downloaded a support package). If you plan to define a custom board yourself, see “FPGA Board Customization” on page 16-2.
- 3 FPGA-in-the-Loop Connection:** FIL simulation connection method. The options in the drop-down menu update depending on the connection methods supported for the target board you

selected. If the target board and HDL Verifier support the connection, you can choose Ethernet, JTAG, or PCI Express.

4 Advanced Options:

When you select an Ethernet connection, you can adjust the board IP and MAC addresses, if necessary.

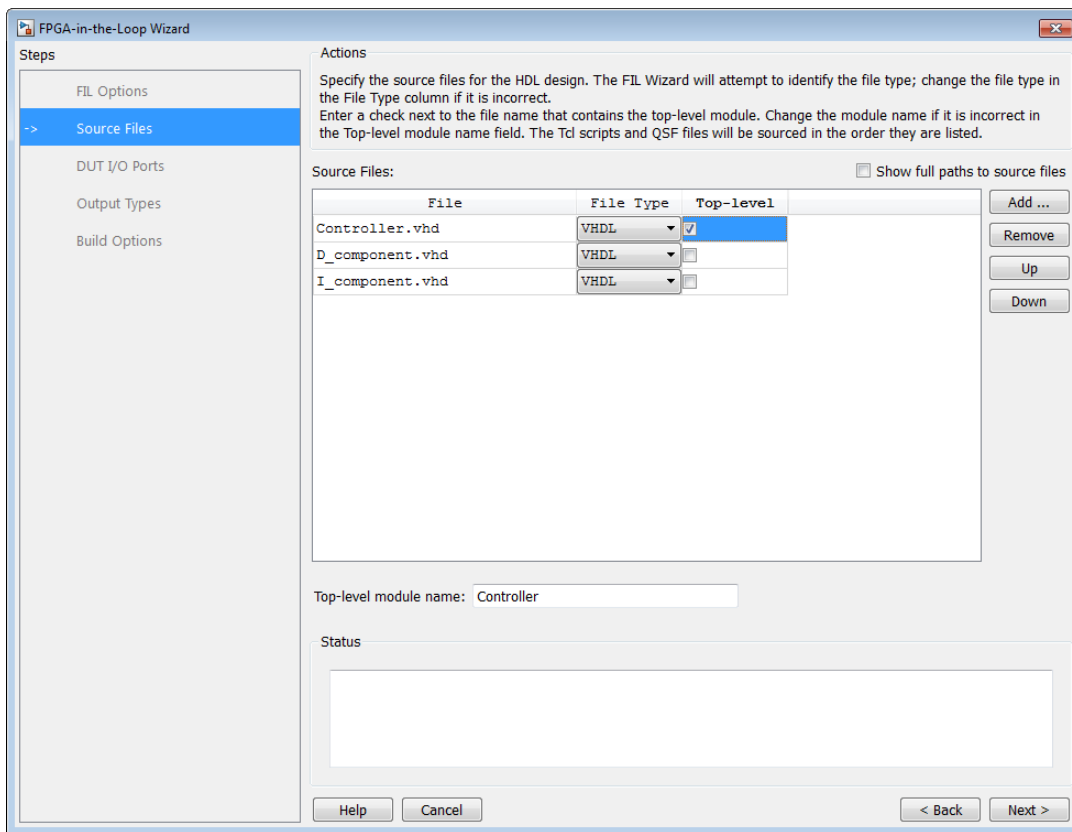
Option	Instructions
Board IP address	<p>Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2).</p> <p>If the default board IP address (192.168.0.2) is in use by another device, or you need a different subnet, change the Board IP address according to the following guidelines:</p> <ul style="list-style-type: none"> • The subnet address, typically the first three bytes of board IP address, must be the same as the subnet of the host IP address. • The last byte of the board IP address must be different from the last byte of the host IP address. • The board IP address must not conflict with the IP addresses of other computers. <p>For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.</p>
Board MAC address	<p>Under most circumstances, you do not need to change the board MAC address. If you connect more than one FPGA development board to a single host computer, change the board MAC address for any additional boards so that each address is unique. You must have a separate NIC for each board.</p> <p>To change the Board MAC address, click in the Board MAC address field. Specify an address that is different from that belonging to any other device attached to your computer. To obtain the Board MAC address for a specific FPGA development board, refer to the label affixed to the board or consult the product documentation.</p>

FPGA system clock frequency (MHz): Enter a target clock frequency. For Intel boards and Xilinx ISE-supported boards, *filWizard* checks the requested frequency against those possible for the requested board. If the requested frequency is not possible for this board, *filWizard* returns an error and suggests an alternate frequency. For Xilinx Vivado-supported boards, or PCI Express boards, *filWizard* cannot check the frequency. The synthesis tools make a best effort attempt at the requested frequency but may choose an alternate frequency if the specified frequency was not achievable. The default is 25 MHz.

Enable data buffering on FPGA: Select this option to enhance simulation performance. When selected, FIL utilizes BRAMs on the FPGA to buffer Ethernet packets in frame-based processing mode. Clear this parameter when BRAM resources are scarce in your design. Available for Ethernet connection only.

5 Click **Next**.

Step 4: Add HDL Source Files for FIL Block



In the **Source Files** page:

- 1 Specify the HDL design to be cosimulated in the FPGA. These files are the HDL design files to be verified on the FPGA board.

Indicate source files by clicking **Add**. Select files using the file selection dialog box.

The FIL wizard attempts to identify the source file types. If any of the file types is not what you expect, you can change it by selecting from the **File Type** drop-down list. Acceptable file types are:

- VHDL
- Verilog
- Netlist
- Tcl script
- Constraints
- Others

"Others" refers to the following:

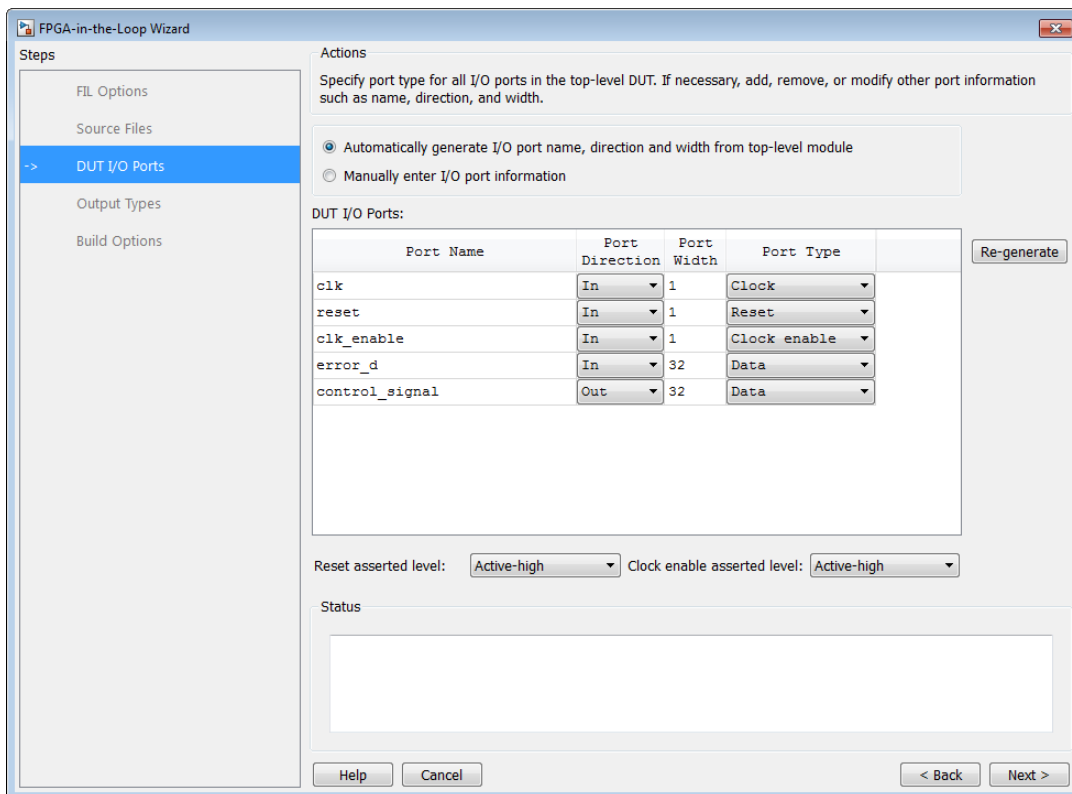
- For Intel, files specified as **Other** are added to the FPGA project, but they have no impact on the generated block. For example, you can put some comments in a readme file and include it in this file list.

- For Xilinx, files specified as **Other** can be any file accepted by Xilinx ISE. ISE looks at the file extension to determine how to use this file. For example, if you add `foo.vhd` to the list and specify it as **Other**, ISE treats the file as a VHDL file.
- 2 Specify which file contains the top-level HDL file.

Check the box on the row of the HDL file that contains the top-level HDL module in the column titled **Top-level**. The FIL wizard automatically fills the **Top-level module name** field with the name of the selected HDL file. If the top-level module name and file name do not match, you can manually change the top-level module name in this dialog box. Indicate the top-level module name before you continue.

- 3 (Optional) To display the full paths to the source files, check the box titled **Show full paths to source files**.
- 4 Click **Next**.

Step 5: Verify DUT I/O Ports for FIL Block



In the **DUT I/O Ports** page:

- 1 Review the port listing. The FIL wizard parses the top-level HDL module to obtain all the I/O ports and display them in the DUT I/O Ports table. The parser attempts to determine the port types from the port names. The wizard then displays these signals under Port Type.

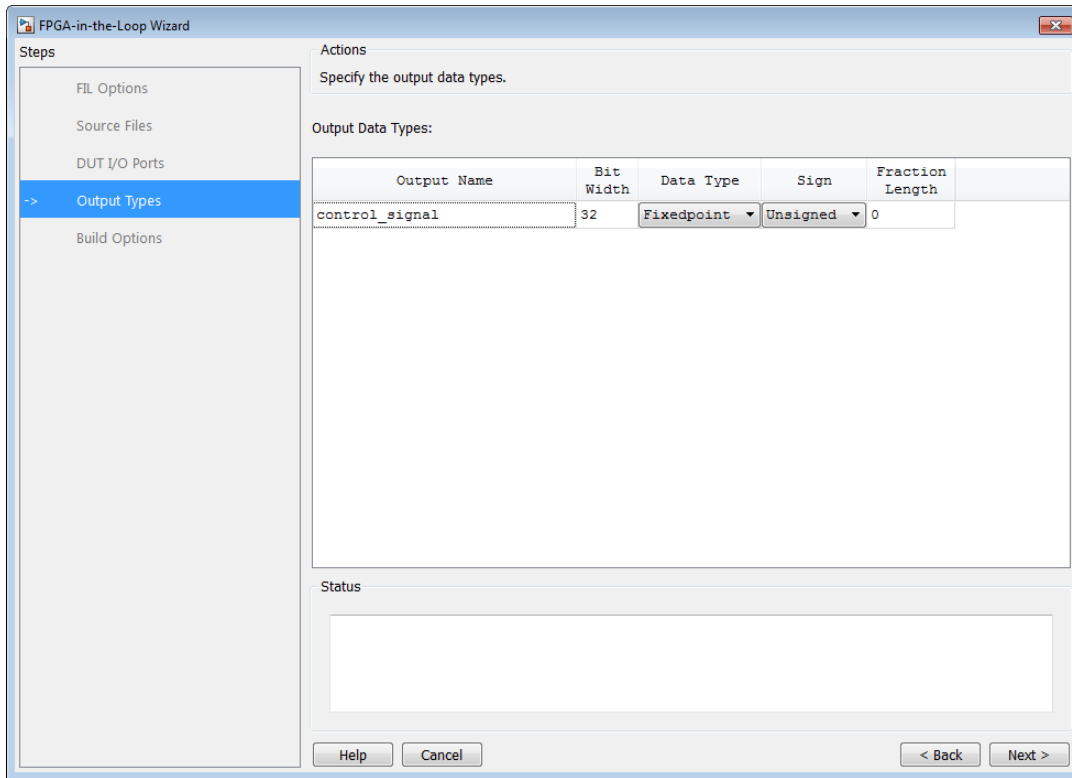
Make sure all input/output/reset ports/clocks are mapped as you expect. If the parser assigned an incorrect port type for any port, you can manually change the signal. For synchronous design, specify a Clock, Reset, or, if desired, a Clock enable signal. The port types specified in this table must be the same as in the HDL code. There must be at least one output port.

Select **Manually enter port information** to add or remove signals.

Click **Regenerate** to reload the table with the original port definitions (from the HDL code).

- 2 Click **Next**.

Step 6: Specify Output Types for FIL Block



In the **Output Types** page:

- 1 Specify output data types. The wizard assigns data types. If any output data type is not what you expect, manually change the type.

Select from:

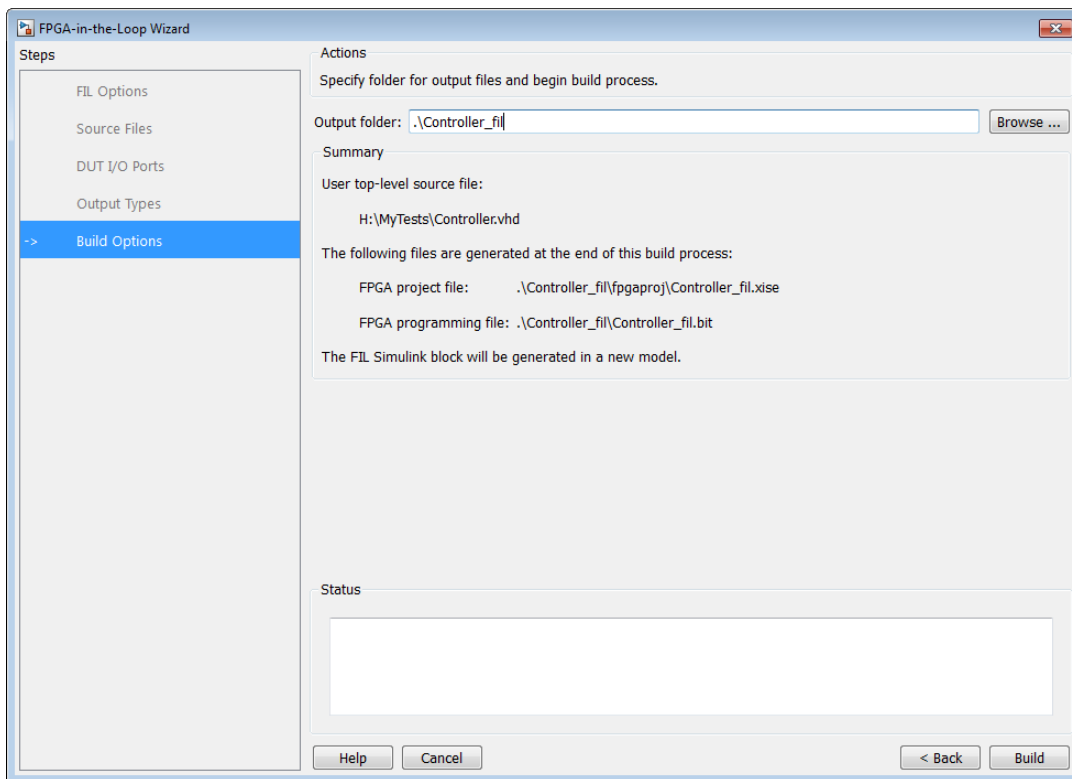
- Fixedpoint
- Integer
- Logical

The data type can depend on the specified bit width.

You can specify the output type to be Signed, Unsigned, or Fraction Length.

- 2 Click **Next**.

Step 7: Specify Build Options for FIL Block



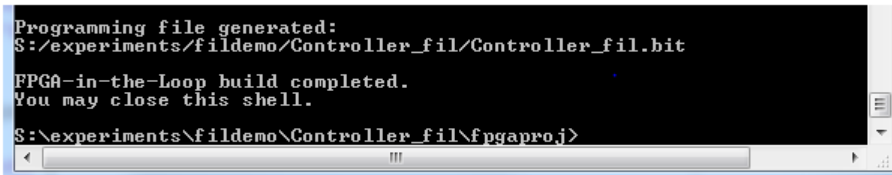
In the **Build Options** page:

- Specify the folder for the output files. You can use the default option. Usually the default is a subfolder named after the top-level module, located under the current folder.
- The **Summary** displays the locations of the ISE project file and the FPGA programming file. You may need those two files for advanced operations on the FIL block mask.

Step 8: Initiate Build

Click **Build** to initiate FIL block generation.

- 1 The FIL wizard generates a FIL block named after the top-level module and places it in a new model.
- 2 The FIL wizard opens a command window.
 - In this window, the FPGA design software performs synthesis, fit, PAR, and FPGA programming file generation.
 - When the process completes, a message in the command window prompts you to close the window.



```

Programming file generated:
S:/experiments/fildemo/Controller_fil/Controller_fil.bit
FPGA-in-the-Loop build completed.
You may close this shell.
S:\experiments\fildemo\Controller_fil\fgaproj>

```

Step 9: Integrate and Simulate

Insert FIL Block Into Model

In your model, replace the DUT subsystem with the FIL block generated in the new model. Save the model under a different name. You can then use the original model as a reference model.

If you generated your FIL block from the HDL workflow advisor, it is unlikely that you need to adjust any settings on the FIL block. If you generated your FIL block using the FIL wizard, you may want to adjust some settings. For instructions on adjusting the FIL block settings, see [FIL Simulation](#).

Load Programming File onto FPGA

Intel Board Instructions for Linux

On Linux hosts, you often must be a superuser to program the bit file onto Intel boards. This requirement can restrict testing on these boards. However, there is a way to work around the superuser requirement.

Fix the device permission issue for Intel bitstream programming under Debian® 5 by creating or modifying a rules file. This solution is a one time file modification that requires SUPERUSER privileges.

- Option 1: Create a rules file (e.g., `92-altera.rules`) under `/etc/udev/rules.d/` with the following contents:

```
# Altera USB-Blaster

ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001", GROUP="users"
```

- Option 2: Add the following lines to an existing rule file under `/etc/udev/rules.d/`

```
# Altera USB-Blaster

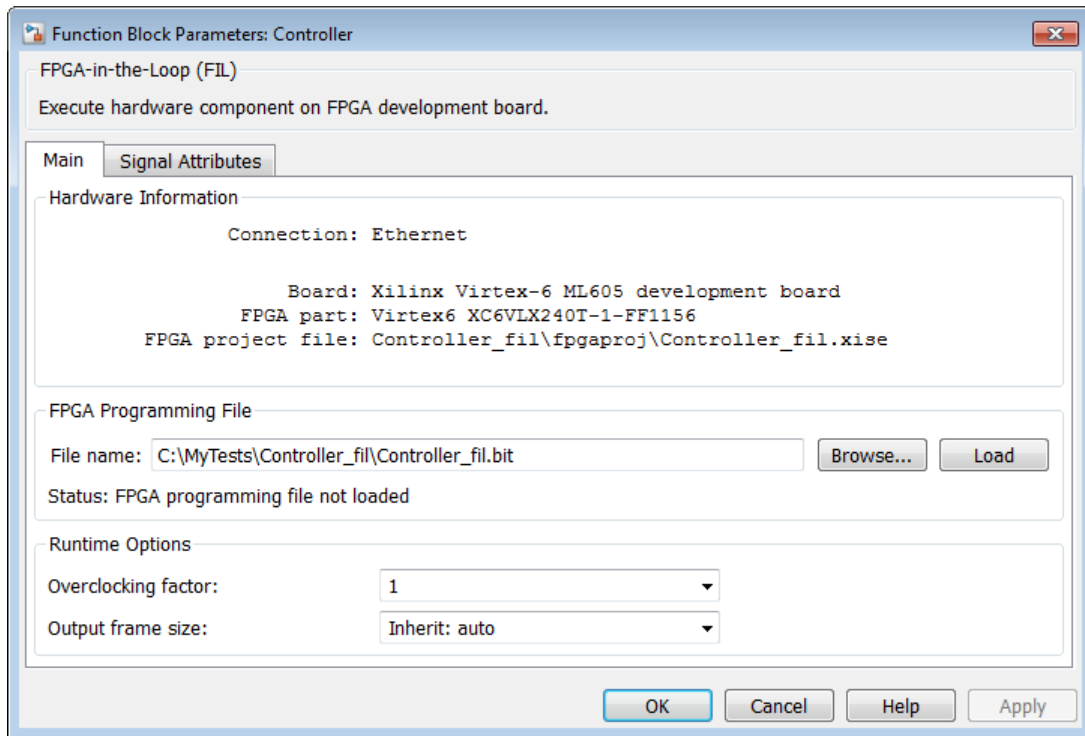
ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001", GROUP="users"
```

Ensure that your FPGA development board is set up, turned on, and connected to your machine using a JTAG cable. Programming uses the JTAG interface, even if you select a different connection for simulation.

Perform the following steps to program the FPGA:

- 1 Double-click the FIL block in your Simulink model to open the block mask.
- 2 On the **Main** tab, click **Load** to download the programming file to the FPGA via the JTAG cable.

The load process can take from a few minutes to several minutes or longer, depending on how large the subsystem is. Sometimes, the process can take an hour and a half or longer for large subsystems.



- 3 A message window indicates when the FPGA programming file has loaded as expected. Click **OK**.

Ethernet Connection

If you are using an Ethernet connection, you can test if the FPGA board is connected to your host computer through the ping test. Open a command-line window, and enter the following command:

```
> ping 192.168.0.2
```

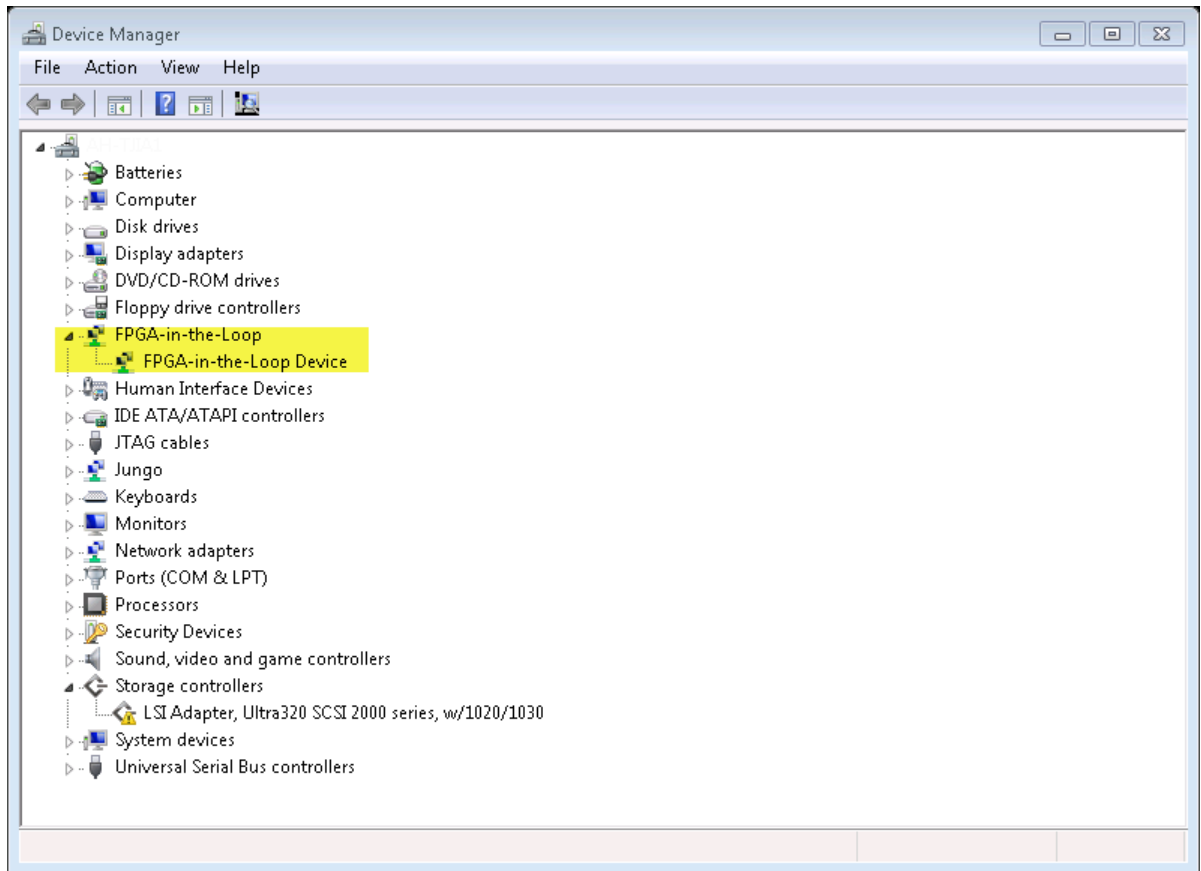
If you changed the board IP address when you set up the network adapter, replace 192.168.0.2 with your board IP address. If the Gigabit Ethernet connection has been set up, you receive the ping reply from the FPGA development board.

PCI Express Connection

If you are using a PCI Express connection, after you program the FPGA, restart the host computer. The host computer cannot detect the new PCI Express device without a restart.

If you are using a PCI Express connection with an Intel board, the board receives power through the host computer. If the host computer turns off, the FPGA loses your programmed design. To restart without losing power to the board, from the **Start** menu, select **Restart**. To avoid disrupting a long simulation, disable the **Sleep** settings for the host computer.

After the restart, check the **Device Manager** in Windows. A new FPGA-in-the-loop device appears in the list.



Run Simulation

In Simulink, run the model that includes the FIL Simulation block. The results of the FIL simulation should match the results of the Simulink reference model or of the original HDL code.

Note RAM Initialization: Simulink starts from time 0 every time, which means the RAM in a Simulink model is initialized to zero for each run. However, this assumption is not true in hardware. RAM in the FPGA holds its value from the end of one simulation to the start of the next. If you have RAM in your design, the first simulation matches Simulink, but subsequent runs may not match. The workaround is to reload the FPGA bitstream before rerunning the simulation. To reload the bitstream, click the **Load** on the FIL block mask.

See Also

More About

- “System Object Generation with the FIL Wizard” on page 13-12
- “FPGA-in-the-Loop Simulation Workflows” on page 12-2
- “Guided Hardware Setup” on page 12-5
- “Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop” on page 32-198
- “Verify Digital Up-Converter Using FPGA-in-the-Loop” on page 32-247

System Object Generation with the FIL Wizard

In this section...

“Step 1: Set Up FPGA Design Software Tools” on page 13-12
 “Step 2: Start FIL Wizard” on page 13-13
 “Step 3: Set FIL Options for System Object” on page 13-13
 “Step 4: Add HDL Source Files for System Object” on page 13-15
 “Step 5: Verify DUT I/O Ports for System Object” on page 13-16
 “Step 6: Specify Output Types for System Object” on page 13-17
 “Step 7: Specify Build Options for System Object” on page 13-18
 “Step 8: Initiate Build” on page 13-18
 “Step 9: Integrate and Simulate” on page 13-20

Step 1: Set Up FPGA Design Software Tools

Xilinx Software

Set up your system environment for accessing Xilinx tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path.

- Xilinx ISE —

```
hdlsetuptoolpath('ToolName','Xilinx ISE','ToolPath','C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64')
```

This example assumes that the Xilinx ISE design suite is installed at `C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64`.

- Xilinx Vivado —

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\apps\Vivado\2013.4-mw-0\Win\bin\vivado')
```

This example assumes that Xilinx Vivado is installed at `C:\apps\Vivado\2013.4-mw-0\Win\bin\vivado`.

Intel Software

Set up your system environment for accessing Intel tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\Altera\12.0\quartus\bin64')
```

This example assumes that the Intel FPGA design software is installed at `C:\Altera\12.0\quartus\bin64`.

Microchip Software

Set up your system environment for accessing Microchip tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath('ToolName','Microchip Libero SoC','ToolPath','C:\Microsemi\Libero_SoC_v12.0\Designer\bin\libero.exe')
```

This example assumes that the Microchip FPGA design software is installed at `C:\Microsemi\Libero_SoC_v12.0\Designer\bin`.

Step 2: Start FIL Wizard

Open the **FPGA-in-the-Loop Wizard**.

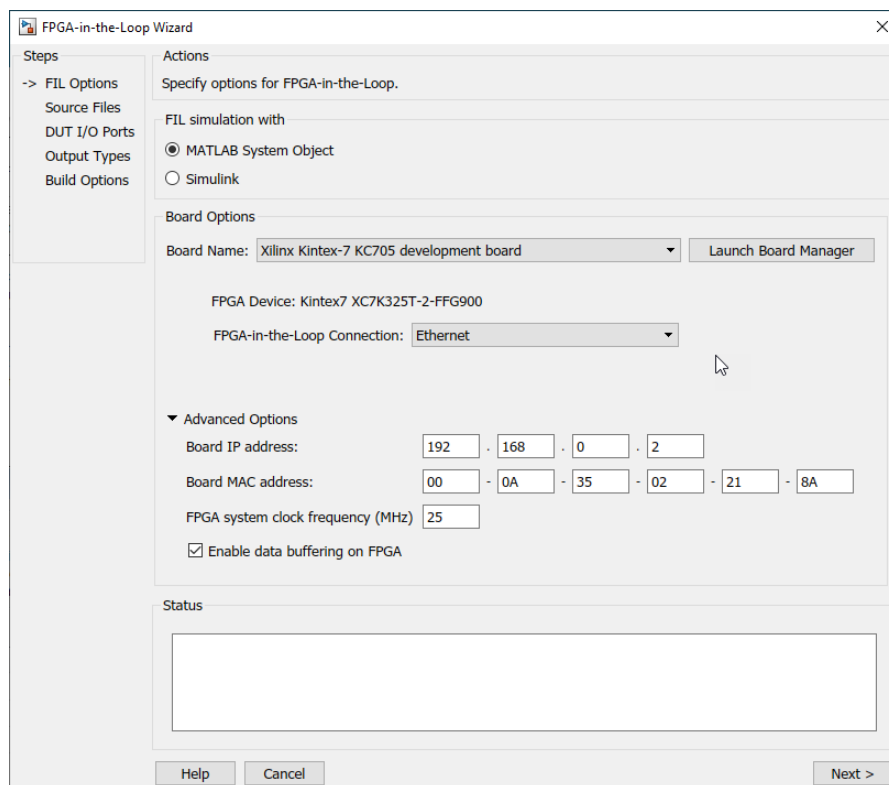
In the MATLAB command window, type the following:

```
>> filWizard
```

To restore a previous session, use this command:

```
filWizard('./Subsystem_fil/Subsystem_fil.mat')
```

Step 3: Set FIL Options for System Object



(This page is for FIL System object. For Simulink block FIL options, see “Step 3: Set FIL Options for FIL Block” on page 13-3.)

In the **FIL Options** page:

- 1 **FIL Simulation with:** Select MATLAB System Object.
- 2 **Board Name:** Select an FPGA development board. If you have not yet downloaded an HDL Verifier FPGA board support package, see “Download FPGA Board Support Package” on page 12-3. (If you do not see any boards listed, then you have not yet downloaded a support package). If you plan to define a custom board yourself, see “FPGA Board Customization” on page 16-2.
- 3 **FPGA-in-the-Loop Connection:** FIL simulation connection method. The options in the drop-down menu update depending on the connection methods supported for the target board you

selected. If the target board and HDL Verifier support the connection, you can choose Ethernet, JTAG, or PCI Express.

4 Advanced Options:

When you select an Ethernet connection, you can adjust the board IP and MAC addresses, if necessary.

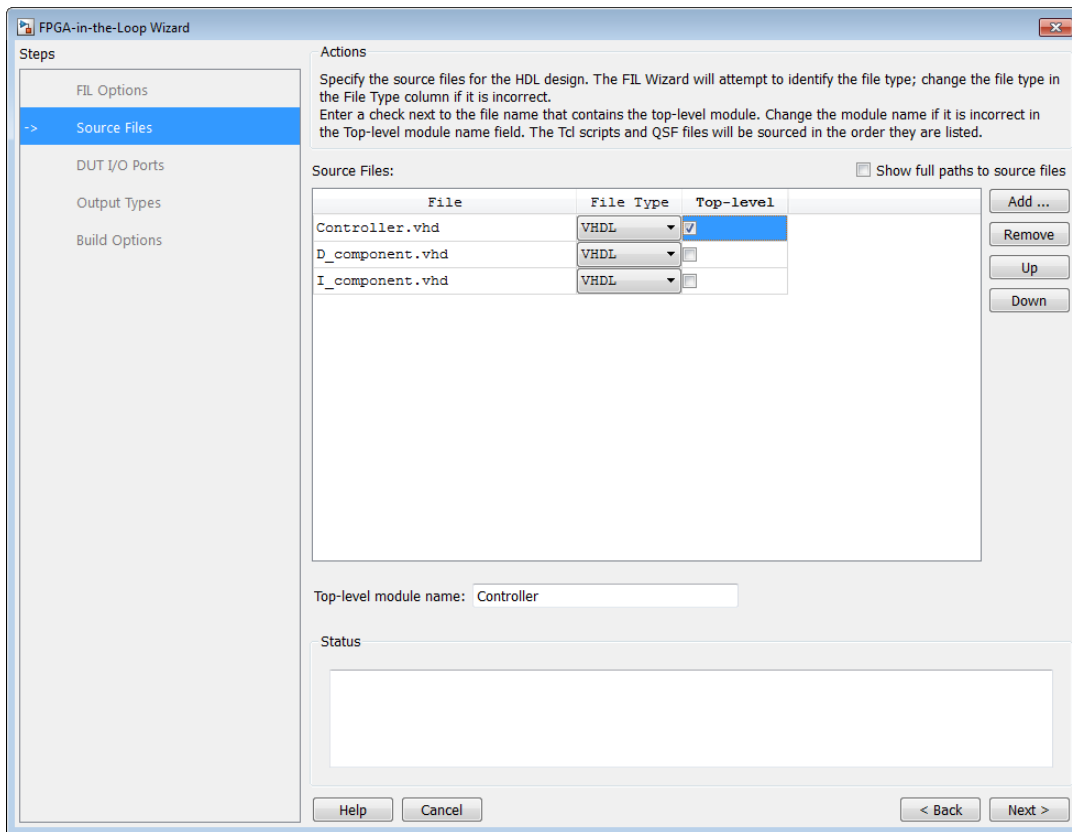
Option	Instructions
Board IP address	<p>Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2).</p> <p>If the default board IP address (192.168.0.2) is in use by another device, or you need a different subnet, change the Board IP address according to the following guidelines:</p> <ul style="list-style-type: none"> • The subnet address, typically the first three bytes of board IP address, must be the same as the subnet of the host IP address. • The last byte of the board IP address must be different from the last byte of the host IP address. • The board IP address must not conflict with the IP addresses of other computers. <p>For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.</p>
Board MAC address	<p>Under most circumstances, you do not need to change the board MAC address. If you connect more than one FPGA development board to a single host computer, change the board MAC address for any additional boards so that each address is unique. You must have a separate NIC for each board.</p> <p>To change the Board MAC address, click in the Board MAC address field. Specify an address that is different from that belonging to any other device attached to your computer. To obtain the Board MAC address for a specific FPGA development board, refer to the label affixed to the board or consult the product documentation.</p>

FPGA system clock frequency (MHz): Enter a target clock frequency. For Intel boards and Xilinx ISE-supported boards, *filWizard* checks the requested frequency against those possible for the requested board. If the requested frequency is not possible for this board, *filWizard* returns an error and suggests an alternate frequency. For Xilinx Vivado-supported boards, or PCI Express boards, *filWizard* cannot check the frequency. The synthesis tools make a best effort attempt at the requested frequency but may choose an alternate frequency if the specified frequency was not achievable. The default is 25 MHz.

Enable data buffering on FPGA: Select this option to enhance simulation performance. When selected, FIL utilizes BRAMs on the FPGA to buffer Ethernet packets in frame-based processing mode. Clear this parameter when BRAM resources are scarce in your design. Available for Ethernet connection only.

5 Click **Next**.

Step 4: Add HDL Source Files for System Object



(This page is for FIL System object. For Simulink block HDL source files, see “Step 4: Add HDL Source Files for FIL Block” on page 13-5.)

In the **Source Files** page:

- 1 Specify the HDL design to be cosimulated in the FPGA. These files are the HDL design files to be verified on the FPGA board.

Indicate source files by clicking **Add**. Select files using the file selection dialog box.

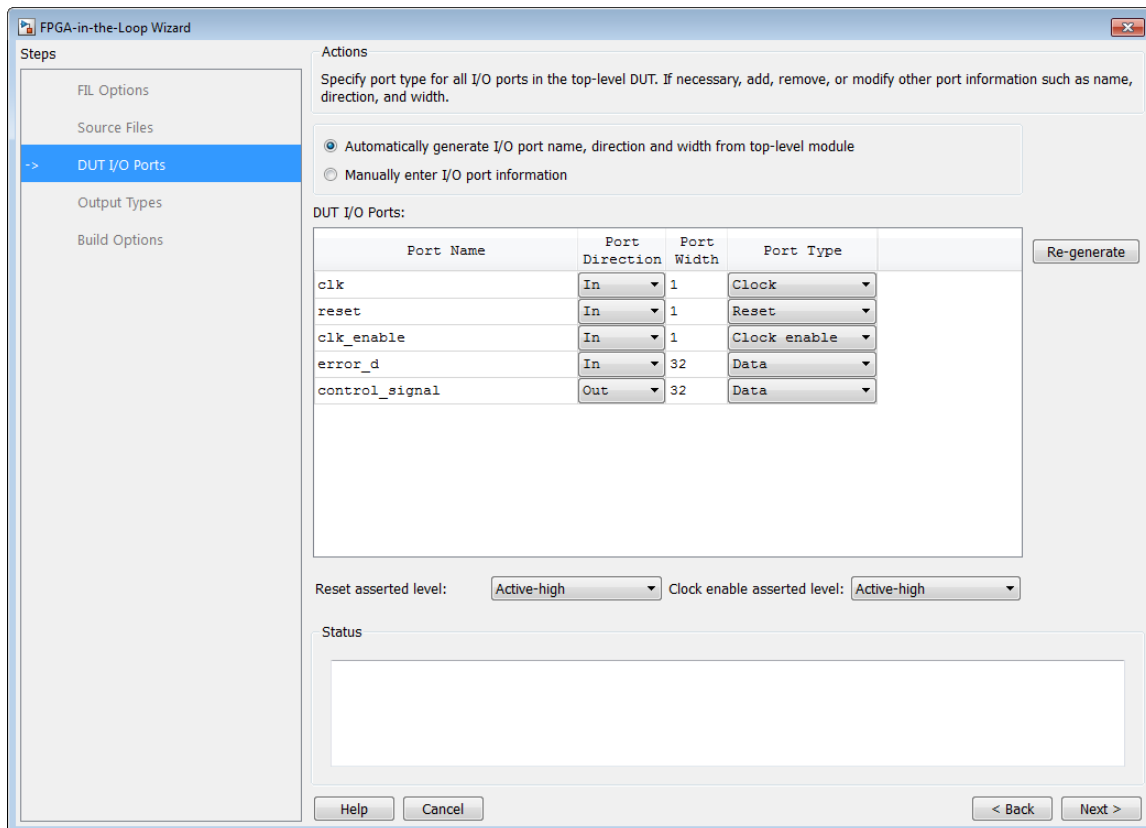
The FIL wizard attempts to identify the source file types. If any of the file types is not what you expect, you can change it by selecting from the **File Type** drop-down list. Acceptable file types are:

- VHDL
- Verilog
- Netlist
- Tcl script
- Constraints
- Others

"Others" refers to the following:

- For Intel, files specified as **Other** are added to the FPGA project, but they have no impact on the generated block. For example, you can put some comments in a `readme` file and include it in this file list.
 - For Xilinx, files specified as **Other** can be any file accepted by Xilinx ISE. ISE looks at the file extension to determine how to use this file. For example, if you add `foo.vhd` to the list and specify it as **Other**, ISE treats the file as a VHDL file.
- 2 Specify which file contains the top-level HDL file.
- Check the box on the row of the HDL file that contains the top-level HDL module in the column titled **Top-level**. The FIL wizard automatically fills the **Top-level module name** field with the name of the selected HDL file. If the top-level module name and file name do not match, you can manually change the top-level module name in this dialog box. Indicate the top-level module name before you continue.
- 3 (Optional) To display the full paths to the source files, check the box titled **Show full paths to source files**.
 - 4 Click **Next**.

Step 5: Verify DUT I/O Ports for System Object



(This page is for FIL with a System object. For Simulink, see “Step 5: Verify DUT I/O Ports for FIL Block” on page 13-6.)

In the **DUT I/O Ports** page:

- 1 Review the port listing. The FIL wizard parses the top-level HDL module to obtain all the I/O ports and display them in the DUT I/O Ports table. The parser attempts to determine the port types from the port names. The wizard then displays these signals under Port Type.

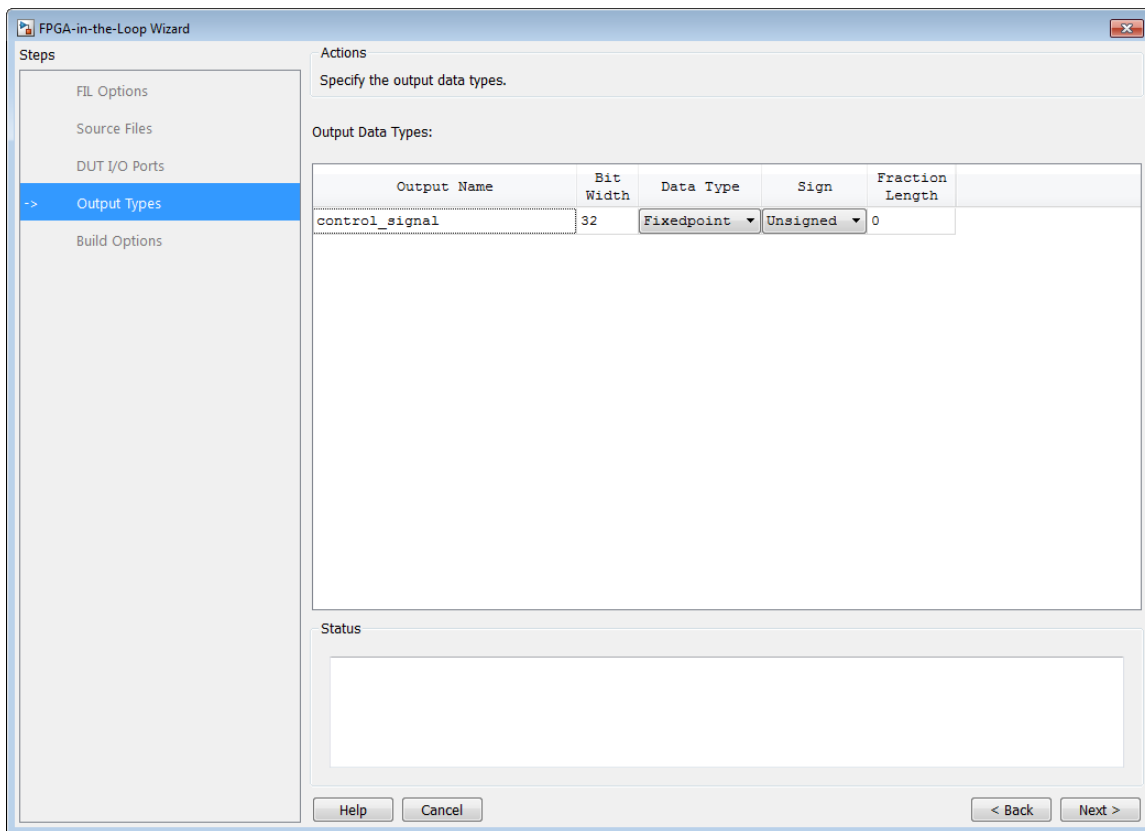
Make sure all input/output/reset ports/clocks are mapped as you expect. If the parser assigned an incorrect port type for any port, you can manually change the signal. For synchronous design, specify a Clock, Reset, or, if desired, a Clock enable signal. The port types specified in this table must be the same as in the HDL code. There must be at least one output port.

Select **Manually enter port information** to add or remove signals.

Click **Regenerate** to reload the table with the original port definitions (from the HDL code).

- 2 Click **Next**.

Step 6: Specify Output Types for System Object



(This page is for FIL System object. For Simulink block output types, see “Step 6: Specify Output Types for FIL Block” on page 13-7.)

In the **Output Types** page:

- 1 Specify output data types. The wizard assigns data types. If any output data type is not what you expect, manually change the type.

Select from:

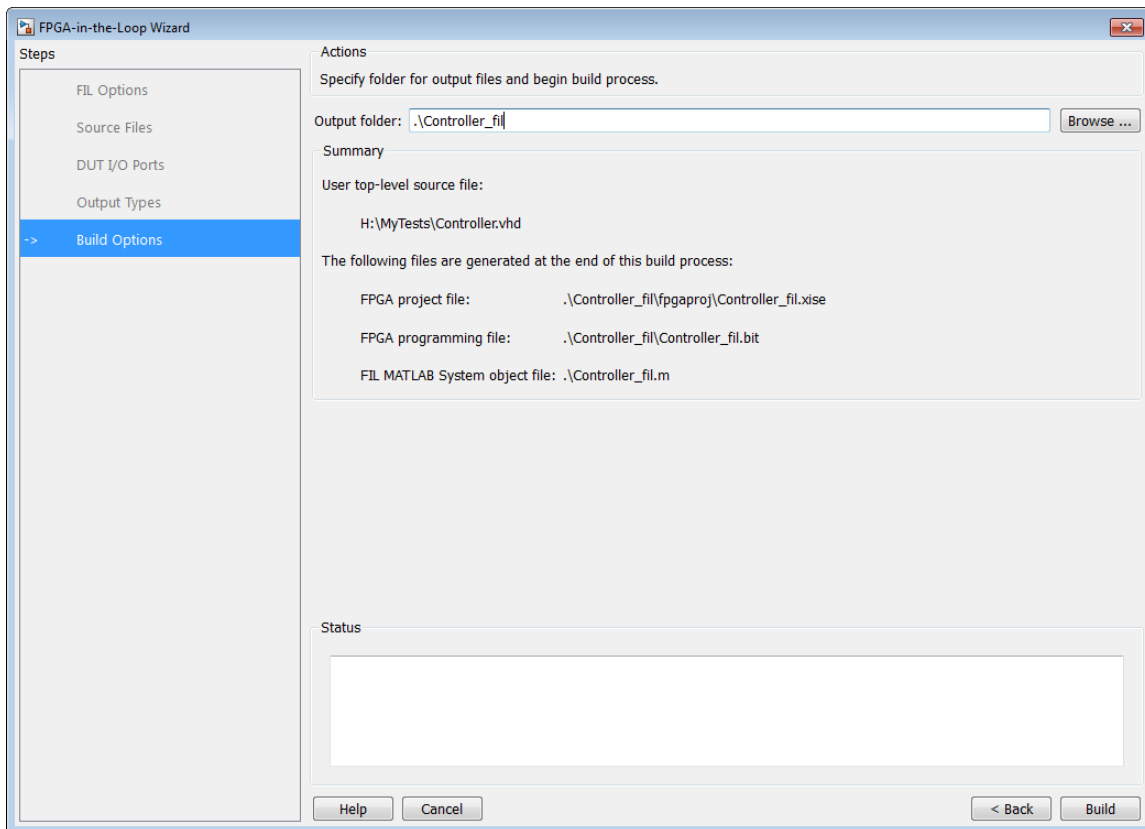
- Fixedpoint
- Integer
- Logical

The data type can depend on the specified bit width.

You can specify the output type to be Signed, Unsigned, or Fraction Length.

- 2 Click **Next**.

Step 7: Specify Build Options for System Object



(This page is for FIL System object. For Simulink, see “Step 7: Specify Build Options for FIL Block” on page 13-8.)

In the **Build Options** page:

- Specify the folder for the output files. You can use the default option. Usually the default is a subfolder named after the top-level module, located under the current folder.
- The **Summary** displays the locations of the ISE project file and the FPGA programming file. You may need those two files for advanced operations on the FIL block mask.

Step 8: Initiate Build

Click **Build** to initiate FIL System object generation.

1 The FIL wizard generates the following files:

- In the `./toplevel_fil/` folder, a MATLAB function named `toplevel_programFPGA.m`, where `toplevel` is the name of the HDL top level. This file contains the code to download the FPGA programming file to the FPGA.

```
function topLevel_programFPGA

    %Load the bitstream in the FPGA
    filProgramFPGA('Xilinx', '/dir/mybitstream.bit', 1);
end
```

- A MATLAB file named `toplevel_fil.m`, where `toplevel` is the name of the HDL top level. This file contains a class definition derived from `hdlverifier.FILSimulation` and initializes the private properties. This file is located in the current folder.

The following is a sample of a class definition file generated using the FIL wizard from a DUT named `fft8`.

```
classdef fft8_fil < hdlverifier.FILSimulation
% fft8_fil is a filWizard generated class used for FPGA-In-the-Loop
% simulation with the 'fft8' DUT.
% fft8_fil connects MATLAB with a FPGA and cosimulate with it by
% writing inputs in the FPGA and reading outputs from the FPGA.
%
% MYFIL = fft8_fil
%
% FIL syntax:
%
% [out1, out2, ...] = MYFIL(in1, in2, ...) connect to the FPGA,
% write in1, in2, ... to the FPGA and read out1, out2, ... from
% the FPGA
%
% fft8_fil methods:
%
% release - Allow property value and input characteristics changes, and
%          release connection to FPGA board
% clone - Create fft8_fil object with same property values
% islocked - Locked status (logical)
% programFPGA - Load the programming file in the FPGA
%
% fft8_fil properties:
%
% DUTName - DUT top level name
% InputSignals - Input paths in the HDL code
% InputBitWidths - Width in bit of the inputs
% OutputSignals - Output paths in the HDL code
% OutputBitWidths - Width in bit of the outputs
% OutputDataTypes - Data type of the outputs
% OutputSigned - Sign of the outputs
% OutputFractionLengths - Fraction lengths of the outputs
% OutputDownsampling - Downsampling factor and phase of the outputs
% OverclockingFactor - Overclocking factor of the hardware
% SourceFrameSize - Frame size of the source (only for HDL source block)
% Connection - Parameters for the connection with the board
% FPGAVendor - Name of the FPGA chip vendor
% FPGABoard - Name of the FPGA board
% FPGAProgrammingFile - Path of the Programming file for the FPGA
% ScanChainPosition - Position of the FPGA in the JTAG scan chain
%
% File Name: fft8_fil.m
% Created: 26-Apr-2012 18:18:06
%
% Generated by FIL Wizard

properties (Nontunable)
    DUTName = 'fft8';
end

methods
    function obj = fft8_fil

        %THE FOLLOWING PROTECTED PROPERTIES ARE SPECIFIC TO THE HW DUT
        %AND MUST NOT BE EDITED (RERUN THE FIL WIZARD TO CHANGE THEM)
```

```

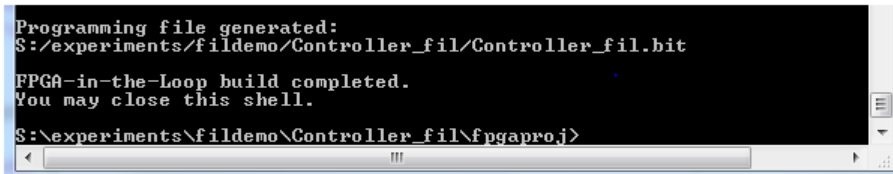
obj.InputSignals = char('Xin_re','Xin_im');
obj.InputBitWidths = [10,10];
obj.OutputSignals = char('Xout_re','Xout_im');
obj.OutputBitWidths = [13,13];
obj.Connection = char('UDP','192.168.0.2','00-0A-35-02-21-8A');
obj.FPGAVendor = 'Xilinx';
obj.FPGABoard = 'XUP Atlys Spartan-6 development board';
obj.ScanChainPosition = 1 ;

%THE FOLLOWING PUBLIC PROPERTIES ARE RELATED TO THE SIMULATION
%AND CAN BE EDITED WITHOUT RERUNING THE FIL WIZARD
obj.OutputSigned = [true,true];
obj.OutputDataTypes = char('fixedpoint','fixedpoint');
obj.OutputFractionLengths = [9,9];
obj.OutputDownsampling = [1,0];
obj.OverclockingFactor = 1;
obj.SourceFrameSize = 1;
obj.FPGAProgrammingFile = 'S:\MATLAB\demo\fft8_fil\fft8_fil.bit';
    end
end
end

```

2 The FIL wizard opens a command window.

- In this window, the FPGA design software performs synthesis, fit, PAR, and FPGA programming file generation.
- When the process completes, a message in the command window prompts you to close the window.



```

Programming file generated:
S:/experiments/fildemo/Controller_fil/Controller_fil.bit

FPGA-in-the-Loop build completed.
You may close this shell.

S:\experiments\fildemo\Controller_fil\fpgaproj>

```

Step 9: Integrate and Simulate

Create System Object

Create a custom `FILSimulation` System object from the class definition file derived using the FIL wizard. This code snippet creates an instance of the class and initializes all properties.

```
MYFIL = toplevel_fil
```

If you generated your FIL System object from the HDL Workflow Advisor, it is unlikely that you need to adjust any settings. If you generated your FIL System object using the FIL Wizard, you may want to adjust some settings. You can adjust any writable property using one of these methods:

- Change the property with the set method:

```
MYFIL.set('FPGAProgrammingFile','c:\work\filfiles')
```

- Set the property directly:

```
MYFIL.FPGAProgrammingFile='c:\work\filfiles'
```

- Edit `toplevel_fil.m` directly. If you edit the `.m` file, instantiate the object in the workspace again, if you had done so previously.

For details about the object properties, see `hdlverifier.FILSimulation`.

Load Programming Files onto FPGA

You can program the FPGA using either the `programFPGA` function, or the `programFPGA` method of the FIL System object. If you have not yet performed the “Guided Hardware Setup” on page 12-5, do so now before loading the programming files.

- `programFPGA` function:

```
./toplevel_fil/toplevel_programFPGA
```

- `programFPGA` method:

```
MYFIL.programFPGA
```

MYFIL is an instance of a `FILSimulation` object.

Run Simulation

- 1 Call the System object in your MATLAB code.
- 2 Run your MATLAB code as you normally would. Make sure that you have performed the “Guided Hardware Setup” on page 12-5 before beginning.

The first call to the object establishes communication with the FPGA board.

See Also

More About

- “Block Generation with the FIL Wizard” on page 13-2
- “FPGA-in-the-Loop Simulation Workflows” on page 12-2
- “Guided Hardware Setup” on page 12-5
- “FPGA-in-the-Loop Simulation Using MATLAB System Object” on page 32-231
- “Verify Viterbi Decoder Using MATLAB System Object and FPGA-in-the-Loop” on page 32-234

FIL Using HDL Coder HDL Workflow Advisor

- “FIL Simulation with HDL Workflow Advisor for Simulink” on page 14-2
- “FIL Simulation with HDL Workflow Advisor for MATLAB” on page 14-8

FIL Simulation with HDL Workflow Advisor for Simulink

In this section...

“Step 1: Start HDL Workflow Advisor” on page 14-2

“Step 2: Set Target and Target Frequency” on page 14-2

“Step 3: Prepare Model for HDL Code Generation” on page 14-3

“Step 4: HDL Code Generation” on page 14-3

“Step 5: Set FPGA-in-the-Loop Options” on page 14-3

“Step 6: Generate FPGA Programming File and FPGA-in-the-Loop Model” on page 14-5

“Step 7: Load Programming File onto FPGA” on page 14-6

“Step 8: Run Simulation” on page 14-7

Step 1: Start HDL Workflow Advisor

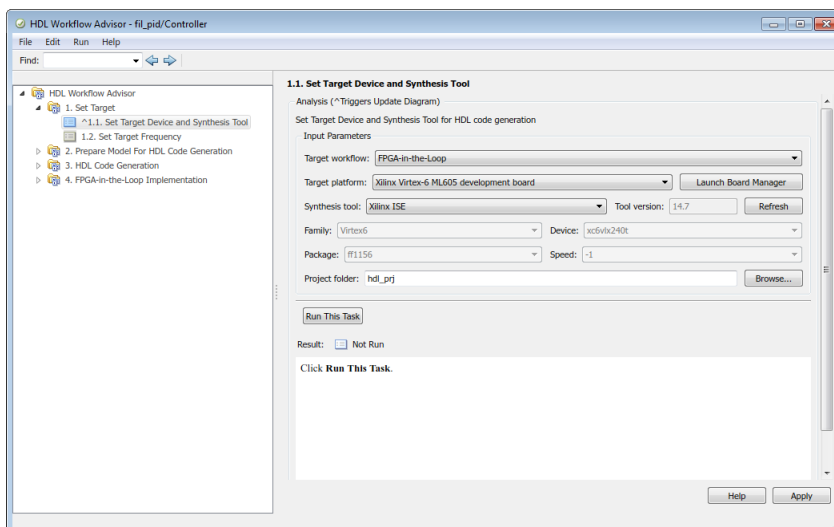
Follow instructions for invoking the HDL Workflow Advisor. See “Getting Started with the HDL Workflow Advisor” (HDL Coder).

Note You must have an HDL Coder license to generate HDL code using the HDL Workflow Advisor.

Step 2: Set Target and Target Frequency

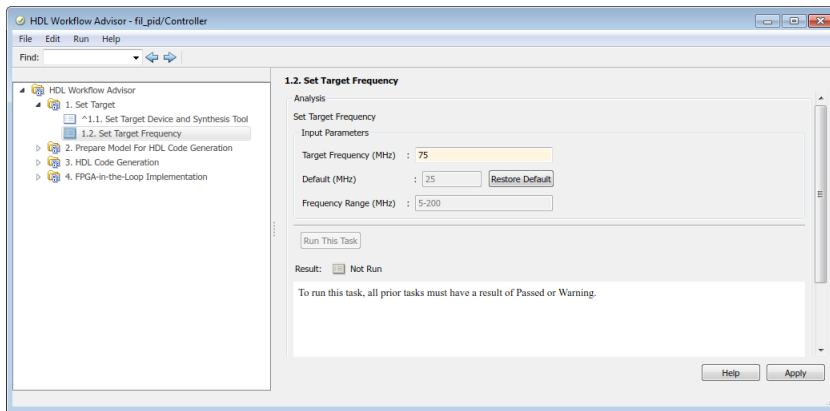
At step 1, **Set Target**, click **1.1 Set Target Device and Synthesis Workflow** and do the following:

- 1 Select **FPGA-in-the-Loop** from the pull-down list at **Target Workflow**.
- 2 Under **Target Platform**, select a development board from the pull-down list. **Family**, **Device**, **Package**, and **Speed** are filled in by the HDL Workflow Advisor. If you have not yet downloaded an HDL Verifier FPGA board support package, select **Get more boards**. Then return to this step after you have downloaded an FPGA board support package.
- 3 For **Folder**, enter the folder name to save the project files into. The default is `hdl_prj` under the current working folder.



After you select a FIL target in Step 1.1, click **1.2 Set Target Frequency**.

- 1 Set the **Target Frequency (MHz)** for the clock speed of your design implemented on the FPGA. The available range of frequencies is shown in the **Frequency Range (MHz)** parameter. For Intel boards and Xilinx boards, Workflow Advisor checks the requested frequency against those possible for the requested board. If the requested frequency is not possible for this board, Workflow Advisor returns an error and suggests an alternate frequency. For Xilinx Vivado-supported boards, or PCI Express boards, Workflow Advisor cannot check the frequency. The synthesis tools make a best effort attempt at the requested frequency but may choose an alternate frequency if the specified frequency was not achievable. The default is 25 MHz.



Step 3: Prepare Model for HDL Code Generation

At step 2, **Prepare Model for HDL Code Generation**, perform steps 2.1–2.4 as described in “Prepare Model for HDL Code Generation Overview” (HDL Coder).

In addition, perform step 2.5 **Check FPGA-in-the-Loop Compatibility** to verify that the model is compatible with FIL.

Step 4: HDL Code Generation

At step 3, **HDL Code Generation**, perform steps 3.1 and 3.2 as described in “HDL Code Generation Overview” (HDL Coder).

Step 5: Set FPGA-in-the-Loop Options

At step 4.1, **Set FPGA-in-the-Loop Options**, change these options if necessary:

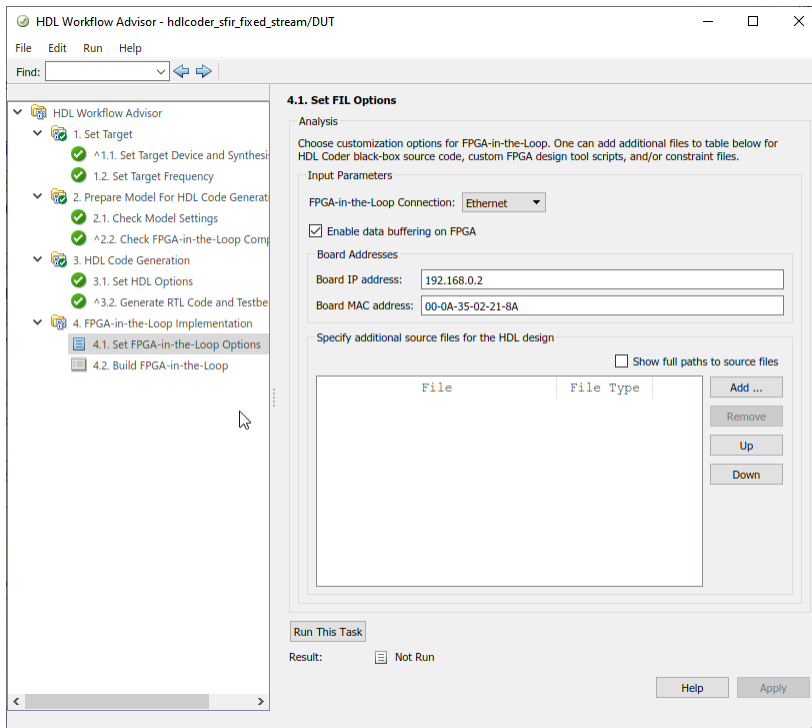
- **FPGA-in-the-Loop Connection:** FIL simulation connection method. The options in the drop-down menu update depending on the connection methods supported for the target board you selected. If the target board and HDL Verifier support the connection, you can choose Ethernet, JTAG, or PCI Express.
- **Enable data buffering on FPGA:** Select this option to enhance simulation performance. When selected, FIL utilizes BRAMs on the FPGA to buffer Ethernet packets in frame-based processing mode. Clear this parameter when BRAM resources are scarce in your design. Available for Ethernet connection only.
- **Board Address:**

When you select an Ethernet connection, you can adjust the board IP and MAC addresses, if necessary.

Option	Instructions
Board IP address	<p>Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2).</p> <p>If the default board IP address (192.168.0.2) is in use by another device, or you need a different subnet, change the Board IP address according to the following guidelines:</p> <ul style="list-style-type: none"> • The subnet address, typically the first three bytes of board IP address, must be the same as the subnet of the host IP address. • The last byte of the board IP address must be different from the last byte of the host IP address. • The board IP address must not conflict with the IP addresses of other computers. <p>For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.</p>
Board MAC address	<p>Under most circumstances, you do not need to change the board MAC address. If you connect more than one FPGA development board to a single host computer, change the board MAC address for any additional boards so that each address is unique. You must have a separate NIC for each board.</p> <p>To change the Board MAC address, click in the Board MAC address field. Specify an address that is different from that belonging to any other device attached to your computer. To obtain the Board MAC address for a specific FPGA development board, refer to the label affixed to the board or consult the product documentation.</p>

- **Specify additional source files for the HDL design:**

Indicate additional source files for the DUT using **Add**. To (optionally) display the full paths to the source files, check the box titled **Show full paths to source files**. The HDL Workflow Advisor attempts to identify the source file type. If the file type is incorrect, you can change it by selecting from the **File Type** drop-down list.



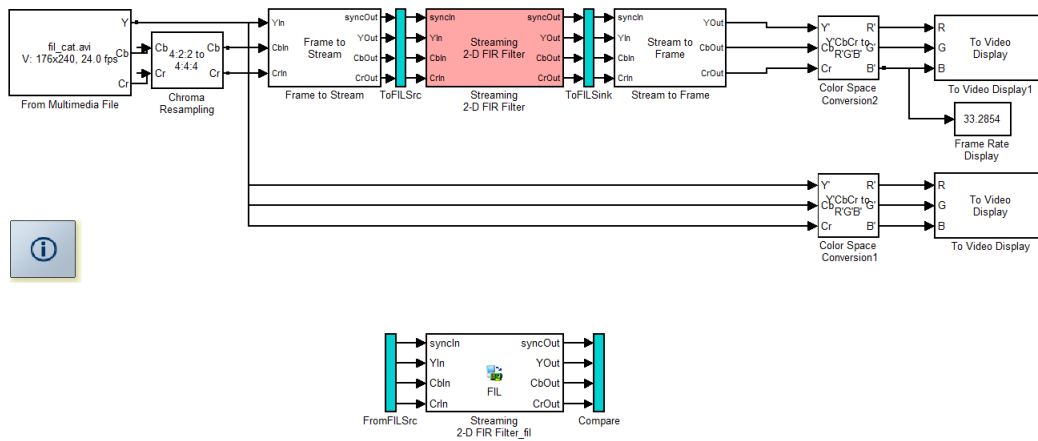
FIL Options

Step 6: Generate FPGA Programming File and FPGA-in-the-Loop Model

At step 4.2, **Build FPGA-in-the-Loop**, click **Run this task**.

During the build process, the following actions occur:

- The HDL Workflow Advisor generates a FIL block named after the top-level module and places it in a new model. The next figure shows an example of the new model containing the FIL block.



- After new model generation, the HDL Workflow Advisor opens a command window:
 - In this window, the FPGA design software performs synthesis, fit, PAR, and FPGA programming file generation.
 - When the process completes, a message in the command window prompts you to close the window.

```

Programming file generated:
S:\experiments\fildemo\Controller_fil\Controller_fil.bit

FPGA-in-the-Loop build completed.
You may close this shell.

S:\experiments\fildemo\Controller_fil\fpgaproj>
    
```

- The HDL Workflow Advisor builds a test bench model around the generated FIL block.

Step 7: Load Programming File onto FPGA

Ensure your FPGA development board is set up, powered on, and connected to your machine as directed by the board manufacturer documentation. Then, perform the following steps to program the FPGA:

- 1 Double-click the FIL block in your Simulink model to open the block mask.
- 2 On the **Main** tab, click **Load** to download the programming file to the FPGA.

The load process may take several minutes, depending on how large the subsystem is. For very large subsystems, the process can take an hour or longer.

For further troubleshooting tips, see “Load Programming File onto FPGA” on page 13-9.

Step 8: Run Simulation

In Simulink, on the **Simulation** tab, click **Run**. The results of the FIL simulation should match those of the Simulink reference model or of the original HDL code.

Note Regarding initialization: Simulink starts from time 0 every time, which means the RAM in Simulink is initialized to zero. However, this is not true in hardware. If you have RAM in your design, the first simulation will match Simulink, but any subsequent runs may not match.

The workaround is to reload the FPGA bitstream before re-running the simulation. To do this, click **Load** on the FIL block mask.

FIL Simulation with HDL Workflow Advisor for MATLAB

In this section...

“Step 1: Start HDL Workflow Advisor” on page 14-8

“Step 2: Select Target” on page 14-8

“Step 3: Select Workflow” on page 14-8

“Step 4: Select FPGA-in-the-Loop Options” on page 14-8

“Step 5: Generate FPGA Programming File and Run Simulation” on page 14-12

Step 1: Start HDL Workflow Advisor

Follow instructions for invoking the HDL Workflow Advisor in MATLAB. See “Getting Started with the HDL Workflow Advisor” (HDL Coder).

Note You must have an HDL Coder license to generate HDL code using the HDL Workflow Advisor.

Step 2: Select Target

Under **Select Code Generation Target**, make sure **Workflow** is set to **Generic ASIC/FPGA**.

Step 3: Select Workflow

Under **HDL Verification**, select **Verify with FPGA-in-the-Loop**.

Step 4: Select FPGA-in-the-Loop Options

- 1 Generate FPGA-in-the-Loop test bench:** Select this option to generate a test bench for simulation with FPGA-in-the-loop.
- 2 Log outputs for comparison plots:** This optional selection lets you log and plot the outputs of the reference design function and the FPGA.
- 3 Board Name:** Select one of the FPGA development boards. If you have not yet downloaded an HDL Verifier FPGA board support package, select **Get more boards**. Then return to this step after you have downloaded an FPGA board support package.
- 4 FPGA-in-the-Loop Connection:** FIL simulation connection method. The options in the drop-down menu update depending on the connection methods supported for the target board you selected. If the target board and HDL Verifier support the connection, you can choose **Ethernet**, **JTAG**, or **PCI Express**.
- 5 Board IP Address and Board MAC Address:**

When you select an Ethernet connection, you can adjust the board IP and MAC addresses, if necessary.

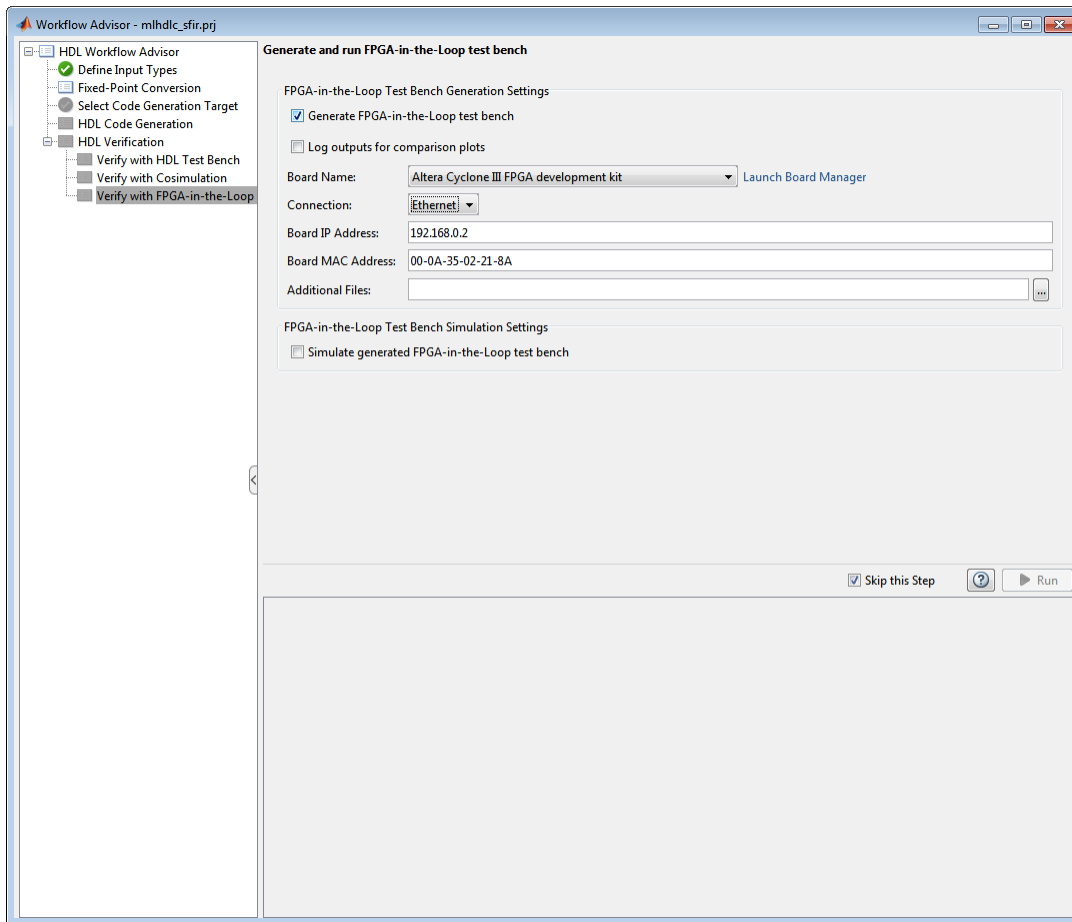
Option	Instructions
Board IP address	<p>Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2).</p> <p>If the default board IP address (192.168.0.2) is in use by another device, or you need a different subnet, change the Board IP address according to the following guidelines:</p> <ul style="list-style-type: none"> • The subnet address, typically the first three bytes of board IP address, must be the same as the subnet of the host IP address. • The last byte of the board IP address must be different from the last byte of the host IP address. • The board IP address must not conflict with the IP addresses of other computers. <p>For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.</p>
Board MAC address	<p>Under most circumstances, you do not need to change the board MAC address. If you connect more than one FPGA development board to a single host computer, change the board MAC address for any additional boards so that each address is unique. You must have a separate NIC for each board.</p> <p>To change the Board MAC address, click in the Board MAC address field. Specify an address that is different from that belonging to any other device attached to your computer. To obtain the Board MAC address for a specific FPGA development board, refer to the label affixed to the board or consult the product documentation.</p>

6 Additional files

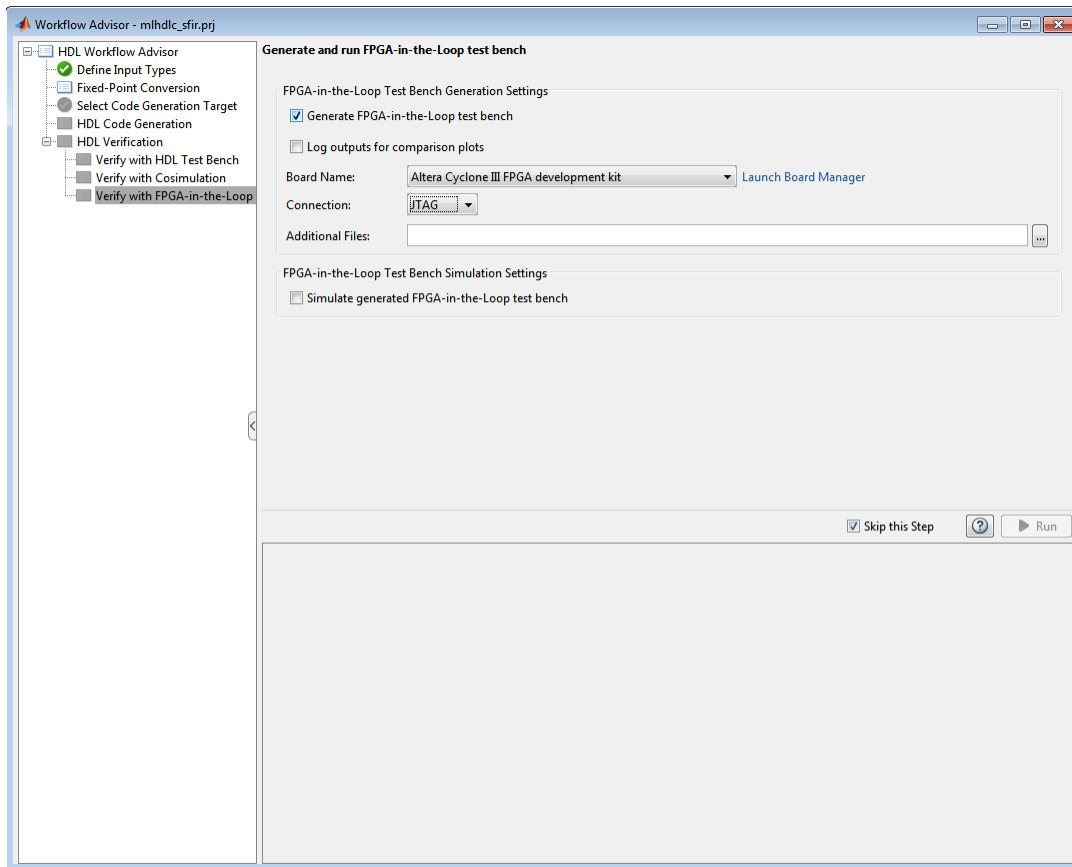
Enter the names of any additional source files for the DUT. If you have more than one additional source file, use the ... button to add more.

7 FPGA-in-the-Loop Test Bench Simulation Settings:

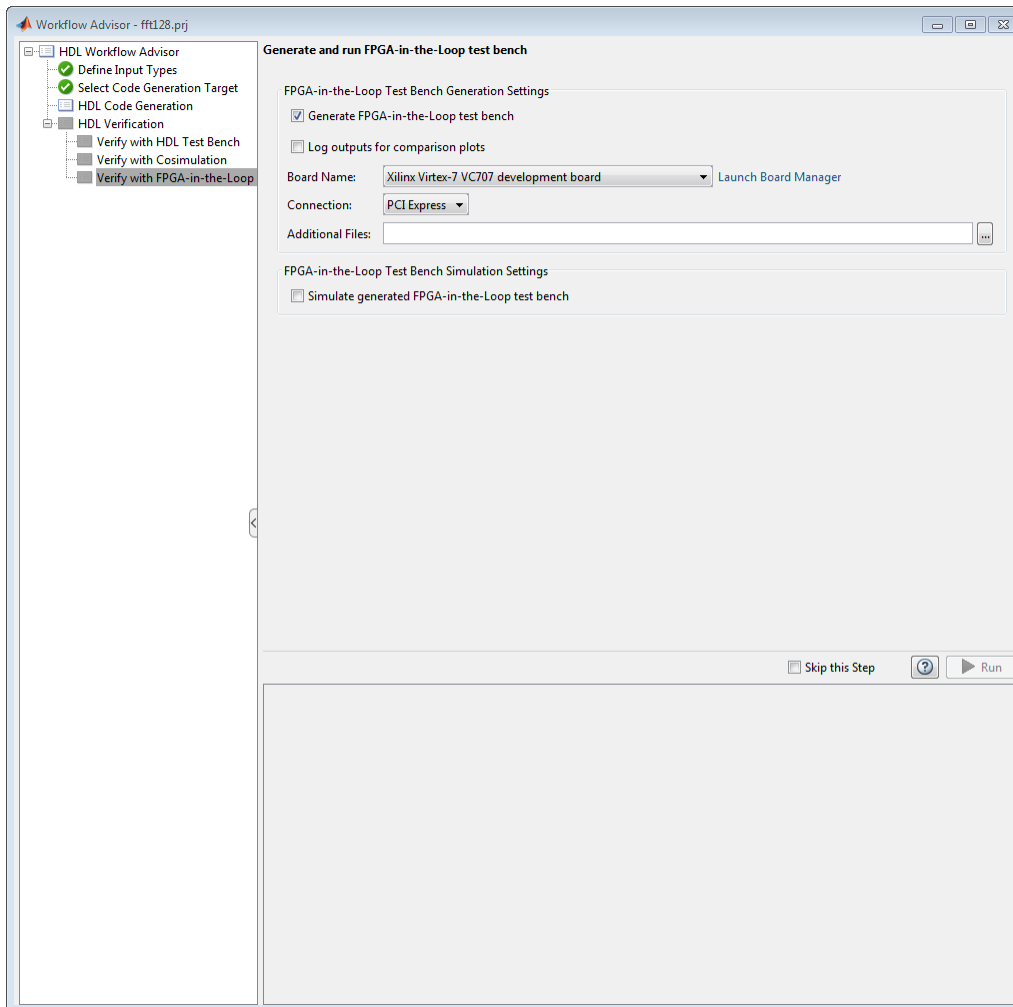
If you want the HDL Workflow Advisor to open the FIL simulation, check the box for **Simulate generated FPGA-in-the-Loop test bench**.



FIL Over Ethernet



FIL Over JTAG



FIL Over PCI Express

Step 5: Generate FPGA Programming File and Run Simulation

If you have not yet run the previous steps, right-click **Verify with FPGA-in-the-Loop** and select **Run to Selected Task**. Otherwise, click **Run**.

This step generates a custom `hdlverifier.FILSimulation` System object that provides an interface to your design running on the FPGA board, and generates a test bench that uses this object to connect to the FPGA board.

If you selected **Simulate generated FPGA-in-the-Loop test bench**, this step loads the FPGA programming file onto the FPGA, and runs the automatically generated test bench with FPGA-in-the-loop.

If you did not select **Simulate generated FPGA-in-the-Loop test bench**, you must load the FPGA programming file manually, using either the customized `toplevel_programFPGA` function, or the `programFPGA` method of the generated object. Reminder: if you have not yet performed the “Guided Hardware Setup” on page 12-5 or “Set Up FPGA Design Software Tools” on page 12-4, do so now before loading the programming files.

- Generated *toplevel_programFPGA* function:

```
./toplevel_fil/toplevel_programFPGA
```

- *programFPGA* object function:

```
MYFIL.programFPGA
```

To run your design on the FPGA board, run the generated test bench, or use the generated object in your own MATLAB code. The first call to the object establishes communication with the FPGA board.

Troubleshooting FPGA-in-the-Loop

Troubleshooting FIL

If you get a message or error at any time during the FIL process (from generating the FIL block to running the simulation), consult one of the following tables for a possible reason and solution.

Message or Error	Reason	Fix
Design does not meet timing goals (this message is generated from the FPGA design software)	The design does not meet timing goals and the software was unable to create the programming file.	Change some part of your design or use a different development board.
Failed to load bitstream	The default libusb shipped with the Debian client is not compatible with iMPACT.	Consult Xilinx user documentation for Linux distribution compatibility of ISE tools.
RAM in design does not match up to Simulink RAM after first simulation run	Simulink starts from time 0 every time, which means the RAM in Simulink is initialized to zero. However, this is not true in hardware. If you have RAM in your design, the first simulation will match Simulink, but any subsequent runs may not match.	The workaround is to reload the FPGA before re-running the simulation.

Message or Error	Reason	Fix
<p>Did not receive data from attached hardware (Ethernet connection)</p>	<p>The connectivity between the host and FPGA development board was lost during the simulation. This error could be caused by a bad network interface card (NIC), bad cable, or loss of power. It also could be caused by an issue with the operating system IP stack where the IP address / MAC address binding is being refreshed, interfering with the transmission of data from the development board to the host.</p>	<p>Check the cables and power so that connectivity can be re-established.</p> <p>You can avoid the IP address / MAC address refresh issue by setting a static entry in the ARP cache (the table that holds the address bindings). You will need to gather the IP address and MAC address by examining the Hardware Information section of the FIL block mask. The following examples will assume the default values of 192.168.0.2 for the IP address and 00-0A-35-02-21-8A for the MAC address.</p> <p>For Windows: With system administrator privileges, execute the following in a command shell:</p> <pre>cmd> arp -s 192.168.0.2 00-0A-35-02-21-8A</pre> <p>To confirm that the operation outcome was as you expected, examine the table and verify the output shows a static entry type:</p> <pre>cmd> arp -a 192.168.0.2</pre> <pre>Interface: 192.168.0.8 --- 0x16 Internet Address Physical Address Type 192.168.0.2 00-0a-35-02-21-8a static</pre> <p>For Linux: As root or via "sudo" privileges, execute the following in a command shell (note that the MAC address delimiter is ":", instead of "-"): </p> <pre>sh> sudo /usr/sbin/arp -s 192.168.0.2 00:0A:35:02:21:8A</pre> <p>To confirm the operation outcome was as you expected, examine the table and verify the output shows a static entry type (so noted by the PERM string):</p> <pre>sh> sudo /usr/sbin/arp -a 192.168.0.2</pre> <pre>? (192.168.0.2) at 00:0a:35:02:21:8a [ether] PERM on eth3</pre>

Message or Error	Reason	Fix
Did not receive data from attached hardware (design frequency)	The configured frequency is too high or too low for the FIL hardware design.	Configure the frequency of your design to the default 25MHz, and rebuild the design, using one of the following workflows: 1 If using filWizard: In the FIL Options section, set Advanced Options > FPGA system clock frequency (MHz) , to 25. Click Next , and continue with the remaining steps to build your design. See FPGA-in-the-Loop Wizard for more details. 2 If using HDL Workflow Advisor: In step 1.2, set Target Frequency (MHz) to 25. Click Run This Task , and continue with the remaining steps to build your design. See “FIL Simulation with HDL Workflow Advisor for Simulink” on page 14-2 for more details.
Failed to load shared library sld_hapi.dll (JTAG connection)	The Altera® Quartus II executables are not on the system path.	Put the Altera Quartus II executables on the system path. If using Linux, make sure that the Quartus II library is on LD_LIBRARY_PATH before you start MATLAB
Failed to load shared library libslid_hapi_dll_loader.so (JTAG connection)	Two possible reasons: <ul style="list-style-type: none"> The version of Altera Quartus II on the host computer is not supported. The Altera Quartus II executables are not on the system path. 	<ul style="list-style-type: none"> Make sure you are using Altera Quartus II version 13.1 or higher on the host computer. Make sure that the Quartus II library is on LD_LIBRARY_PATH before you start MATLAB
Cannot load any more object with static TLS	There is a finite number of libraries with TLS initialization that can be loaded for a given process. Ensure the Altera Quartus II library has priority.	Add your location of Altera/15.0-mw-0/Linux/quartus/Linux64/libjtag_client.so to LD_PRELOAD. Then restart MATLAB.
Undefined reference to lzma_code@XZ_5.0 (JTAG connection)	The Quartus II library liblzma.so.5 has over-shadowed the Linux distribution version of liblzma.so.5.	Prepend the Linux distribution library path before the Quartus II library on LD_LIBRARY_PATH. For example, /lib/x86_64-linux-gnu:\$QUARTUS_PATH.

Message or Error	Reason	Fix
Unable to find the JTAG communication cable attached to the host computer (JTAG connection)	JTAG cable is not connected. It is also possible that the JTAG cable is defective.	Use the JTAG download cable to connect the FPGA development board with the computer.
Failed to open SLD hub (JTAG connection)	The SLD hub is missing. It is required for FPGA-in-the-loop simulation with the Altera JTAG cable.	Make sure that the FPGA is programmed with the correct programming file, which contains the SLD hub.
Reset pin not connected to RESET push button (Alternate message: "Did not get version" displayed in the cosim block)	Most likely scenario is that you changed the Ethernet card but did not re-program the FPGA, although other reasons may be also possible.	Use the FPGA Board Manager to see if there is a reset pin specified for the custom or built-in board. If there is a reset pin specified, look at the board specification manual to see which push button it is connected to.
Sporadic data mismatch when running FIL if your DUT doesn't have a clock enable signal.	The gated clock to the DUT does not meet timing.	Add a clock enable signal to your DUT.
For Xilinx devices, the workflow generates a Vivado project but fails to generate a bitstream. ERROR: Synthesis failed	When executing the FIL workflow on a Linux machine, an xterm opens in the background to execute bitstream generation.	Install xterm on your Linux machine.

FPGA Board Customization

- “FPGA Board Customization” on page 16-2
- “Create Custom FPGA Board Definition” on page 16-6
- “Create Xilinx KC705 Evaluation Board Definition File” on page 16-7
- “FPGA Board Manager” on page 16-18
- “New FPGA Board Wizard” on page 16-22
- “FPGA Board Editor” on page 16-33

FPGA Board Customization

In this section...

“Feature Description” on page 16-2

“Custom Board Management” on page 16-2

“FPGA Board Requirements” on page 16-2

Feature Description

Both HDL Coder and HDL Verifier software include a set of predefined FPGA boards you can use with the Turnkey or FPGA-in-the-loop (FIL) workflows. You can view the lists of these supported boards in the HDL Workflow Advisor or in the FIL wizard. With the FPGA Board Manager, you can add additional boards to use either of these workflows. To add a board, you need the relevant information from the board specification documentation.

The FPGA Board Manager is the hub for accessing wizards and dialog boxes that take you through the steps necessary to create a custom board configuration. You can also access options for:

- Importing a custom board
- Copying a board definition file for further modification
- Verifying a new board

Custom Board Management

You manage FPGA custom boards through the following user interfaces:

- “FPGA Board Manager” on page 16-18: portal to adding, importing, deleting, and otherwise managing board definition files.
- “New FPGA Board Wizard” on page 16-22: This wizard guides you through creating a custom board definition file with information you obtain from the board specification documentation.
- “FPGA Board Editor” on page 16-33: user interface for viewing or editing board information.

To begin, review the “FPGA Board Requirements” on page 16-2 and then follow the steps described in “Create Custom FPGA Board Definition” on page 16-6.

FPGA Board Requirements

- “FPGA Device” on page 16-2
- “FPGA Design Software” on page 16-3
- “General Hardware Requirements” on page 16-3
- “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 16-3
- “JTAG Connection Requirements for FPGA-in-the-Loop” on page 16-5

FPGA Device

Select one of the following links to view a current list of supported FPGA device families:

- For use with FPGA-in-the-loop (FIL), see “Supported FPGA Device Families for Board Customization”.
- For use with FPGA Turnkey, see “Supported FPGA Device Families for Board Customization” (HDL Coder).

FPGA Design Software

Altera Quartus II or Xilinx ISE is required. See product documentation for HDL Coder or HDL Verifier for the specific software versions required.

The following MathWorks tools are required to use FIL or FPGA Turnkey.

Workflow	Required Tools
FPGA-in-the-loop	<ul style="list-style-type: none"> • HDL Verifier • Fixed-Point Designer
FPGA Turnkey	<ul style="list-style-type: none"> • HDL Coder • Simulink • Fixed-Point Designer

General Hardware Requirements

To use an FPGA development board, make sure that you have the following FPGA resources:

- **Clock:** An external clock connected to the FPGA is required. The clock can be differential or single-ended. The accepted clock frequency is from 5 MHz to 300 MHz. When used with FIL, there are additional requirements to the clock frequency (see “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 16-3).
- **Reset:** An external reset signal connected to the FPGA is optional. When supplied, this signal functions as the global reset to the FPGA design.
- **JTAG download cable:** A JTAG download cable that connects host computer and FPGA board is required for the FPGA programming. The FPGA must be programmable using Xilinx iMPACT or Altera Quartus II.

Ethernet Connection Requirements for FPGA-in-the-Loop

- “Supported Ethernet PHY Device” on page 16-3
- “Ethernet PHY Interface” on page 16-4
- “Special Timing Considerations for RGMII” on page 16-4
- “Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface” on page 16-4

Supported Ethernet PHY Device

On the FPGA board, the Ethernet MAC is implemented in FPGA. An Ethernet PHY chip is required to be on the FPGA board to connect the physical medium to the Media Access (MAC) layer in the FPGA.

Note When programming the FPGA, HDL Verifier assumes that there is only one download cable connected to the Host computer. It also assumes that the FPGA programming software automatically recognizes the cable. If not, use FPGA programming software to program your FPGA with the correct options.

The FIL feature is tested with the following Ethernet PHY chips and may not work with other Ethernet PHY devices.

Ethernet PHY Chip	Test
Marvell® Alaska 88E1111	For GMII, RGMII, SGMII, and 100 Base-T MII interfaces
National Semiconductor DP83848C	For 100 Base-T MII interface only

Ethernet PHY Interface

The Ethernet PHY chip must be connected to the FPGA using one of the following interfaces:

Interface	Note
Gigabit Media Independent Interface (GMII)	Only 1000 Mb/s speed is supported using this interface.
Reduced Gigabit Media Independent Interface (RGMII)	Only 1000 Mb/s speed is supported using this interface.
Serial Gigabit Media Independent Interface (SGMII)	Only 1000 Mb/s speed is supported using this interface.
Media Independent Interface (MII)	Only 100 Mb/s speed is supported using this interface.

Note For GMII, the TXCLK (clock signal for 10/100 Mb/s signal) signal is not required because only 1000 Mb/s speed is supported.

In addition to the standard GMII/RGMII/SGMII/MII interface signals, FPGA-in-the-loop also requires an Ethernet PHY chip reset signal (ETH_RESET_n). This active-low reset signal performs the PHY hardware reset by FPGA. It is active-low.

Special Timing Considerations for RGMII

When the RGMII interface is used, the MAC on the FPGA assumes that the data are aligned with the edges of reference clock as specified in the original RGMII v1.3 standard. In this case, PC board designs provide additional trace delay for clock signals.

The RGMII v2.0 standard allows the transmitter to integrate this delay so that PC board delay is not required. Marvell Alaska 88E1111 has internal registers to add internal delays to RX and TX clocks. The internal delays are not added by default, which means that you must use the MDIO module to configure Marvell 88E1111 to add internal delays. For more information on the MDIO module, see “FIL I/O” on page 16-26.

Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface

When GMII/RGMII/SGMII interfaces are used, the FPGA requires an exact 125 MHz clock to drive the 1000 Mb/s communication. This clock is derived from the user supplied external clock using the clock module or PLL.

Not all external clock frequencies can derive an exact 125 MHz clock frequency. The acceptable clock frequencies vary depending on the FPGA device family. The recommended clock frequencies are 50, 100, 125, and 200 MHz.

JTAG Connection Requirements for FPGA-in-the-Loop

Vendor	Required Hardware	Required Software
Intel	USB Blaster I or USB Blaster II download cable	<ul style="list-style-type: none"> • USB Blaster I or II driver • For Windows operating systems: Quartus Prime executable directory must be on system path. • For Linux operating systems: versions below Quartus II 13.1 are not supported. Quartus II 14.1 is not supported. Only 64-bit Quartus is supported. Quartus library directory must be on LD_LIBRARY_PATH before starting MATLAB. Prepend the Linux distribution library path before the Quartus library on LD_LIBRARY_PATH. For example, /lib/x86_64-linux-gnu:\$QUARTUS_PATH.
Xilinx	Digilent download cable <ul style="list-style-type: none"> • If your board has an onboard Digilent USB-JTAG module, use a USB cable • If your board has a standard Xilinx 14 pin JTAG connector, use with HS2 or HS3 cable from Digilent 	<ul style="list-style-type: none"> • For Windows operating systems: Xilinx Vivado executable directory must be on system path. • For Linux operating systems: Digilent Adept 2. For the installation steps, see “Install Digilent Adept 2 Runtime” (HDL Verifier Support Package for Xilinx FPGA Boards).
	FTDI USB-JTAG cable <ul style="list-style-type: none"> • Supported for boards with onboard FT4232H, FT232H, or FT2232H devices implementing USB-to JTAG 	Install these D2XX drivers. <ul style="list-style-type: none"> • For Windows operating systems: 2.12.28 (64 bit) • For Linux operating systems: 1.4.22 (64 bit) For the installation guide, see D2XX Drivers from the FTDI Chip website.
Microchip	JTAG connection not supported	

Create Custom FPGA Board Definition

- 1 Be ready with the following:
 - a Board specification document. Any format you are comfortable with is fine. However, if you have it in an electronic version, you can search for the information as it is required.
 - b If you plan to validate (test) your board definition file, set up FPGA design software tools:

For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.
- 2 Open the FPGA Board Manager by typing `fpgaBoardManager` in the MATLAB command window. Alternatively, if you are using the HDL Workflow Advisor, you can click **Launch Board Manager** at Step 1.1.
- 3 Open the New FPGA Board wizard by clicking **Create Custom Board**. For a description of all the tasks you can perform with the FPGA Board Manager, see “FPGA Board Manager” on page 16-18.
- 4 The wizard guides you through entering all board information. At each page, fill in the required fields. For assistance in entering board information, see “New FPGA Board Wizard” on page 16-22.
- 5 Save the board definition file. This step is the last and is automatically instigated when you click **Finish** in the New FPGA Board wizard. See “Save Board Definition File” on page 16-13.

Your custom board definition now appears in the list of available FPGA Boards in the FPGA Board Manager. If you are using HDL Workflow Advisor, it also shows in the **Target platform** list.

Follow the example “Create Xilinx KC705 Evaluation Board Definition File” on page 16-7 for a demonstration of adding a custom FPGA board with the New FPGA Board Manager.

Create Xilinx KC705 Evaluation Board Definition File

In this section...

“Overview” on page 16-7
 “What You Need to Know Before Starting” on page 16-7
 “Start New FPGA Board Wizard” on page 16-7
 “Provide Basic Board Information” on page 16-8
 “Specify FPGA Interface Information” on page 16-9
 “Enter FPGA Pin Numbers” on page 16-10
 “Run Optional Validation Tests” on page 16-12
 “Save Board Definition File” on page 16-13
 “Use New FPGA Board” on page 16-14

Overview

For FPGA-in-the-loop, you can use your own qualified FPGA board, even if it is not in the pre-registered FPGA board list supplied by MathWorks. Using the New FPGA Board wizard, you can create a board definition file that describes your custom FPGA board.

In this example, you can follow the workflow of creating a board definition file for the Xilinx KC705 evaluation board to use with FIL simulation.

What You Need to Know Before Starting

- Check the board specification so that you have the following information ready:
 - FPGA interface to the Ethernet PHY chip
 - Clock pins names and numbers
 - Reset pins names and numbers

In this example, the required information is supplied to you. In general, you can find this type of information in the board specification file. This example uses the KC705 Evaluation Board for the Kintex-7 FPGA User Guide, published by Xilinx.

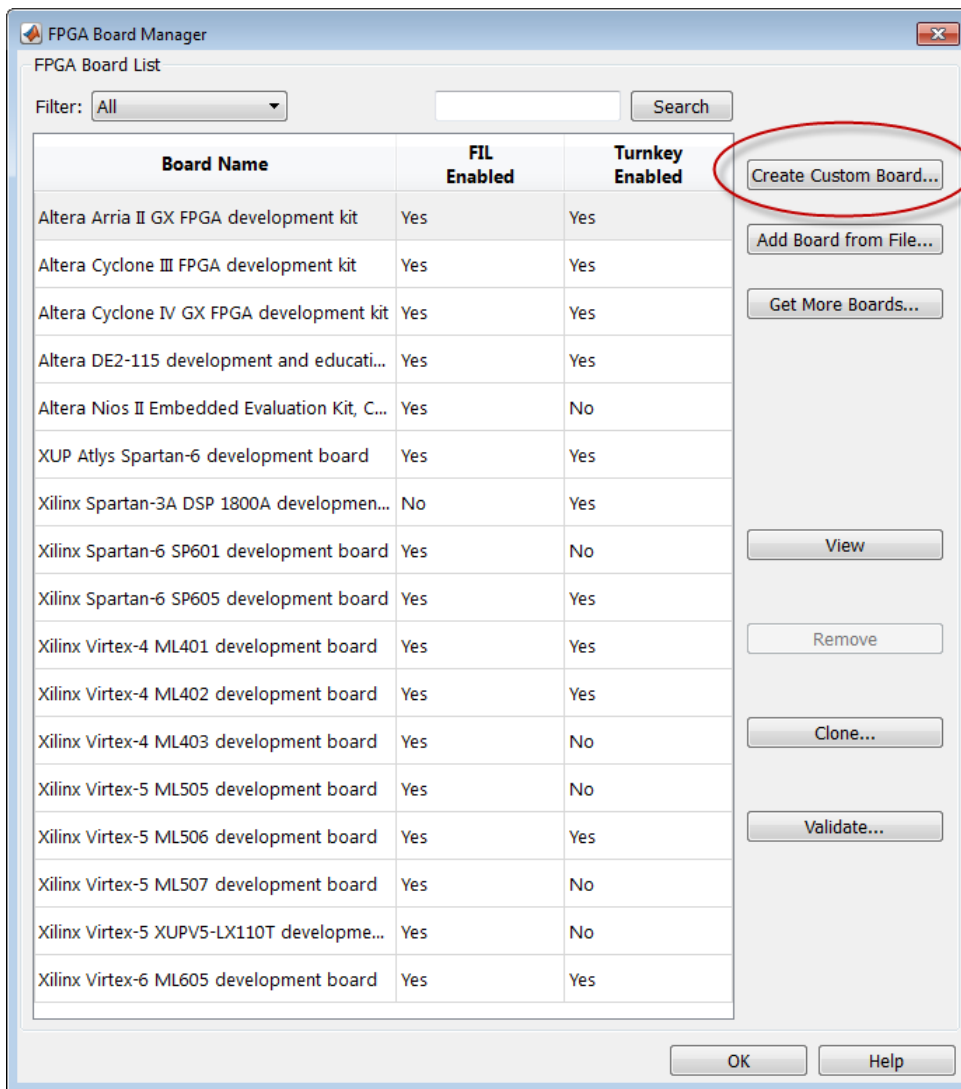
- For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.
- To verify programming the FPGA board after you add its definition file, attach the custom board to your computer. However, having the board connected is not necessary for creating the board definition file.

Start New FPGA Board Wizard

- 1 Start the FPGA Board Manager by entering the following command at the MATLAB prompt:

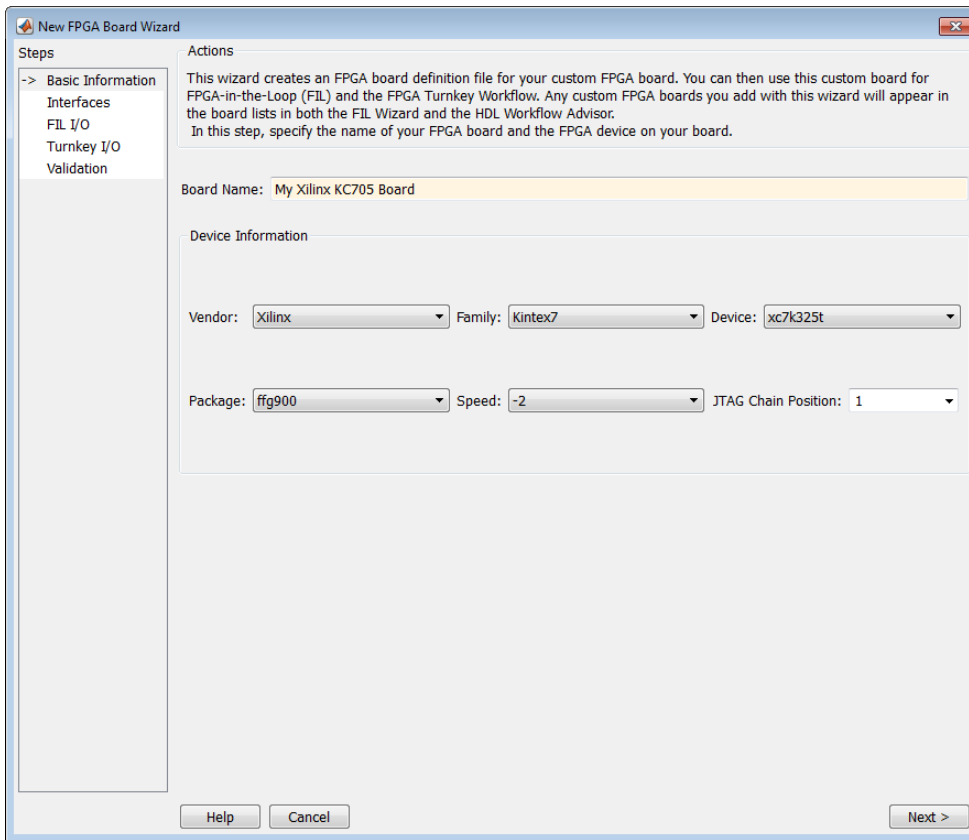
```
>>fpgaBoardManager
```

- 2 Click **Create Custom Board** to open the New FPGA Board wizard.



Provide Basic Board Information

- In the Basic Information pane, enter the following information:
 - **Board Name:** Enter "My Xilinx KC705 Board"
 - **Vendor:** Select Xilinx
 - **Family:** Select Kintex7
 - **Device:** Select xc7k325t
 - **Package:** Select ffg900
 - **Speed:** Select -2
 - **JTAG Chain Position:** Select 1



The information you just entered can be found in the KC705 Evaluation Board for the Kintex-7 FPGA User Guide.

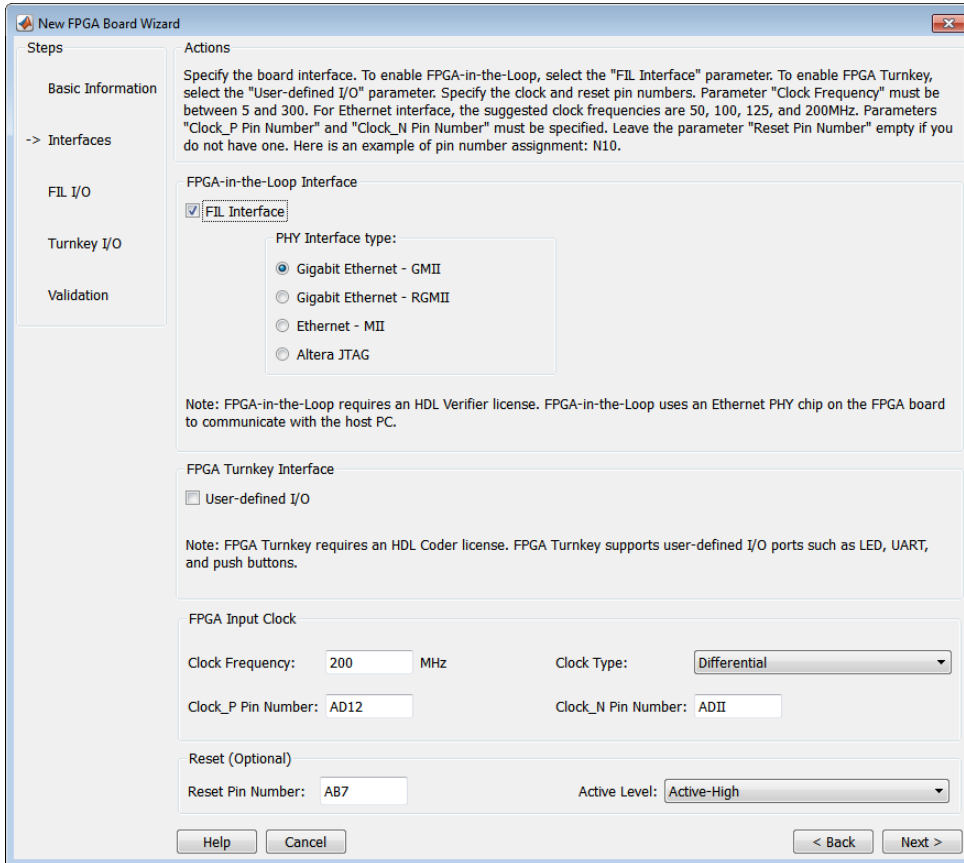
- 2 Click **Next**.

Specify FPGA Interface Information

- 1 In the Interfaces pane, perform the following tasks.
 - a Select **FIL Interface**. This option is required for using your board with FPGA-in-the-loop.
 - b Select **GMII** in the PHY Interface Type. This option indicates that the onboard FPGA is connected to the Ethernet PHY chip via a GMII interface.
 - c Leave the **User-defined I/O** option in the FPGA Turnkey Interface section cleared. FPGA Turnkey workflow is not the focus of this example.
 - d **Clock Frequency:** Enter 200. This Xilinx KC705 board has multiple clock sources. The 200 MHz clock is one of the recommended clock frequencies for use with Ethernet interface (50, 100, 125, and 200 MHz).
 - e **Clock Type:** Select **Differential**.
 - f **Clock_P Pin Number:** Enter AD12.
 - g **Clock_N Pin Number:** Enter AD11.
 - h **Clock IO Standard** — Leave blank.
 - i **Reset Pin Number:** Enter AB7. This value supplies a global reset to the FPGA.

- j **Active Level:** Select Active-High.
- k **Reset IO Standard** — Leave blank.

You can obtain all necessary information from the board design specification.



- 2 Click **Next**.

Enter FPGA Pin Numbers

- 1 In the FIL/I/O pane, enter the numbers for each FPGA pin. This information is required.

Pin numbers for RXD and TXD signals are entered from the least significant digit (LSD) to the most significant digit (MSB), separated by a comma.

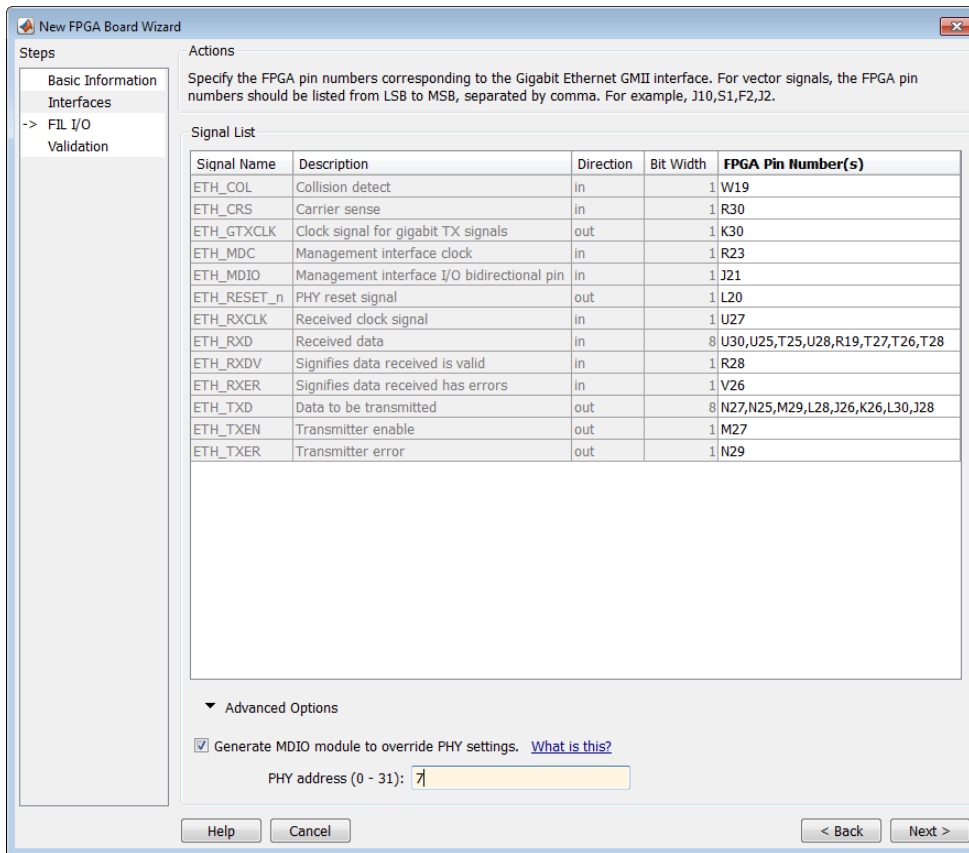
For signal name...	Enter FPGA pin number...
ETH_COL	W19
ETH_CRS	R30
ETH_GTXCLK	K30
ETH_MDC	R23
ETH_MDIO	J21
ETH_RESET_n	L20

For signal name...	Enter FPGA pin number...
ETH_RXCLK	U27
ETH_RXD	U30,U25,T25,U28,R19,T27,T26,T28
ETH_RXDV	R28
ETH_RXER	V26
ETH_TXD	N27,N25,M29,L28,J26,K26,L30,J28
ETH_TXEN	M27
ETH_TXER	N29

- 2 Click Advanced Options to expand the section.
- 3 Check the **Generate MDIO module to override PHY settings** option.

This option is selected for the following reasons:

- There are jumpers on the Xilinx KC705 board that configure the Ethernet PHY device to MII, GMII, RGMII, or SGMII mode. Since this example uses the GMII interfaces, the FPGA board does not work if the PHY devices are set to the wrong mode. When the **Generate MDIO module to override PHY settings** option is selected, the FPGA uses the Management Data Input/Output (MDIO) bus to override the jumper settings and configure the PHY chip to the correct GMII mode.
 - This option currently only applies to Marvell Alaska PHY device 88E1111 and this KC705 board is using the Marvel device.
- 4 **PHY address (0 - 31):** Enter 7.



5 Click **Next**.

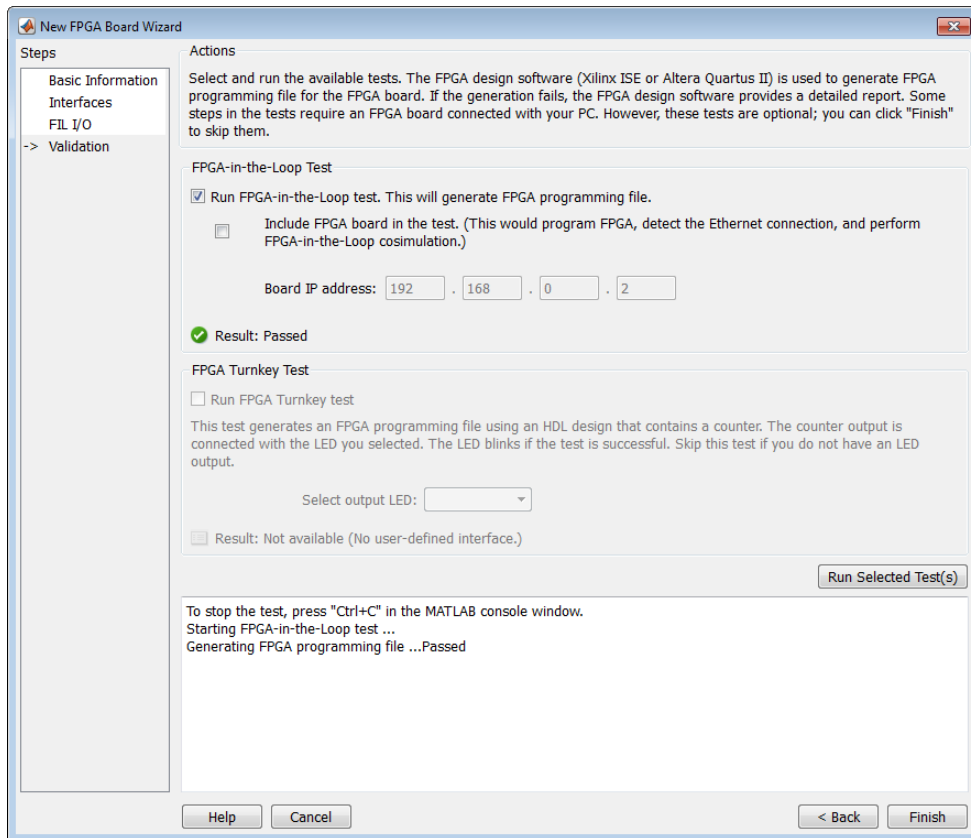
Run Optional Validation Tests

This step provides a validation test for you to verify if the entered information is correct by performing FPGA-in-the-loop cosimulation. You need Xilinx ISE 13.4 or higher versions installed on the same computer. This step is optional and you can skip it, if you prefer.

Note For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

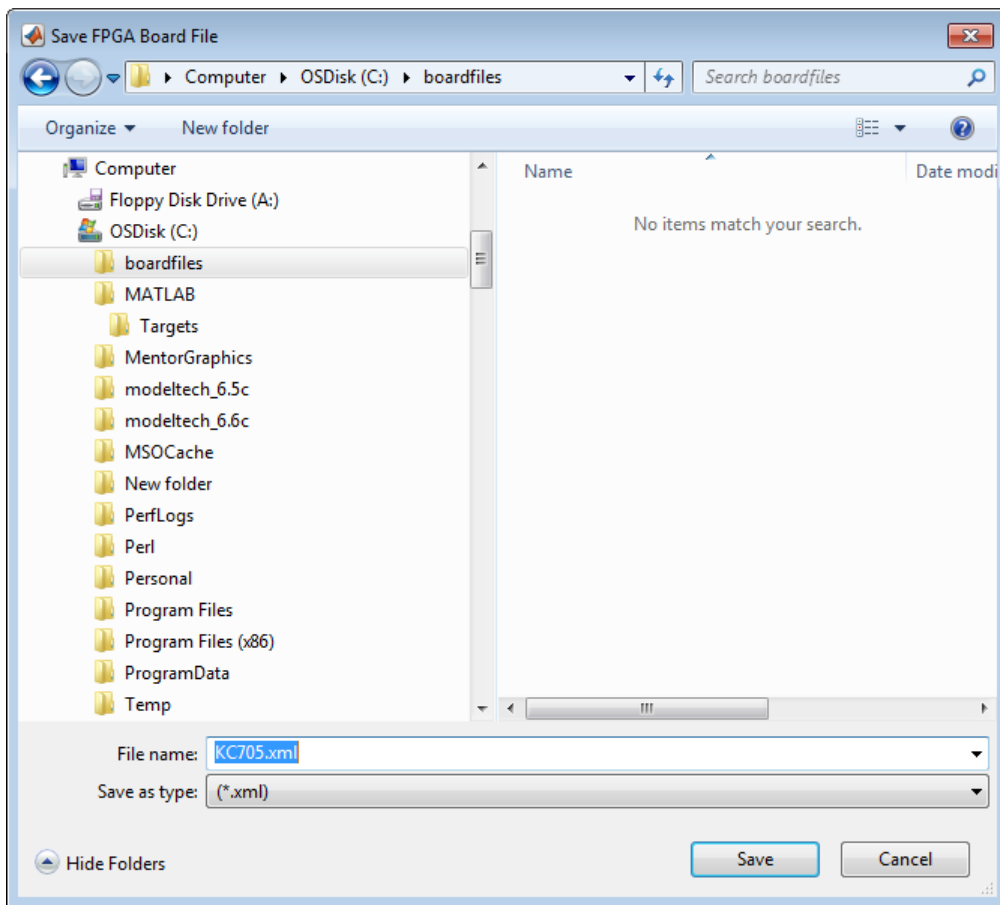
To run this test, perform the following actions.

- 1 Check the **Run FPGA-in-the-Loop test** option.
- 2 If you have the board attached, check the **Include FPGA board in the test** option. You need to supply the IP address of the FPGA Board. This example assumes that the Xilinx KC705 board is attached to your host computer and it has an IP address of 192.168.0.2.
- 3 Click **Run Selected Test(s)**. The tests take about 10 minutes to complete.



Save Board Definition File

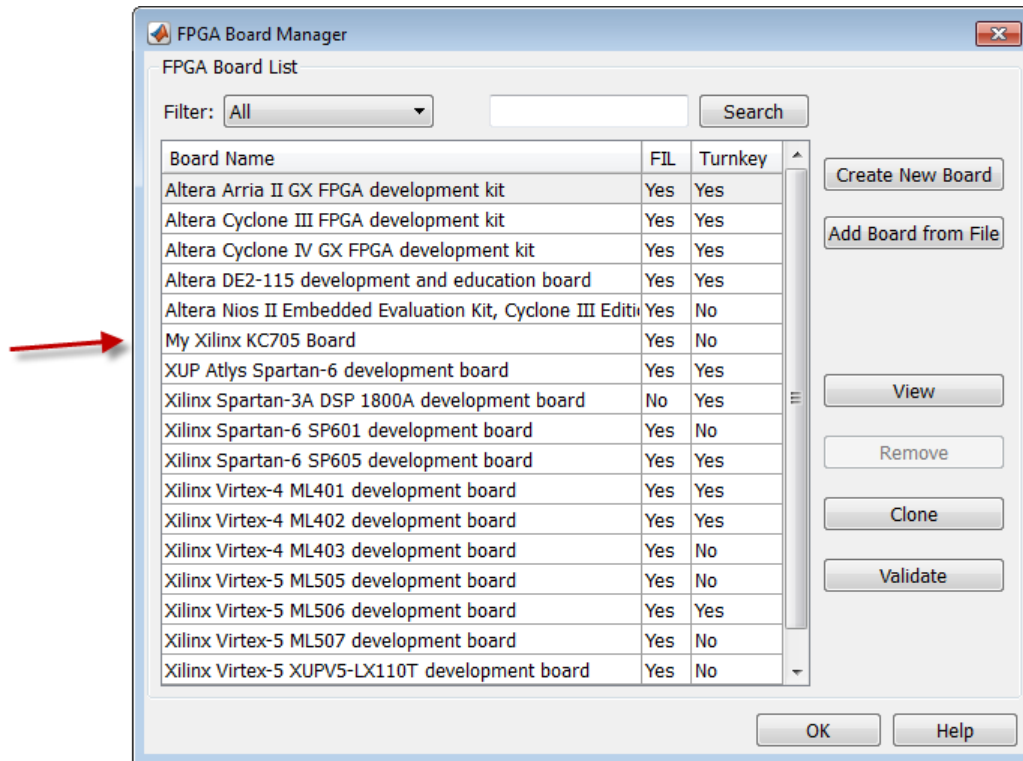
- 1 Click **Finish** to exit the New FPGA Board wizard. A **Save As** dialog box pops up and asks for the location of the FPGA board definition file. For this example, save as C:\boardfiles \KC705.xml.



- 2 Click **Save** to save the file and exit.

Use New FPGA Board

- 1 After you save the board definition file, you are returned to the FPGA Board Manager. In the FPGA Board List, you can now see the new board you defined.



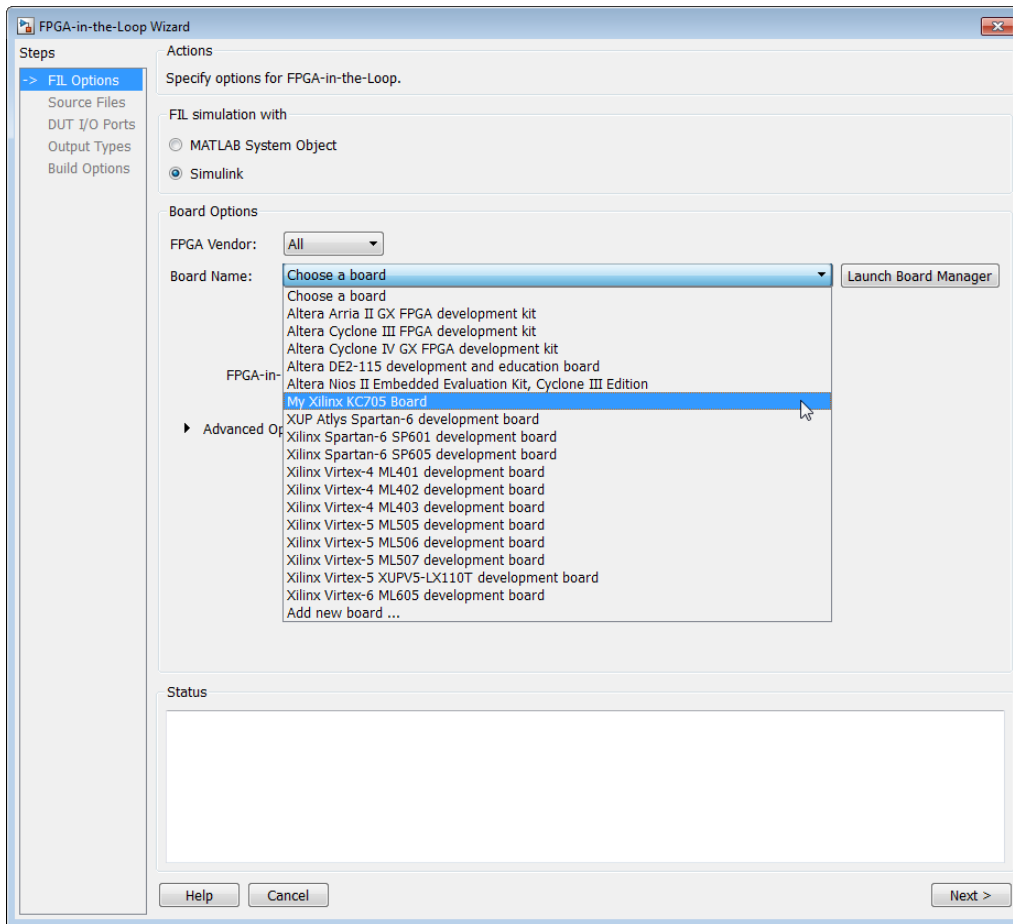
Click **OK** to close the FPGA Board Manager.

- 2 You can view the new board in the board list from either the FIL wizard or the HDL Workflow Advisor.

- a Start the FIL wizard from the MATLAB prompt.

```
>>filWizard
```

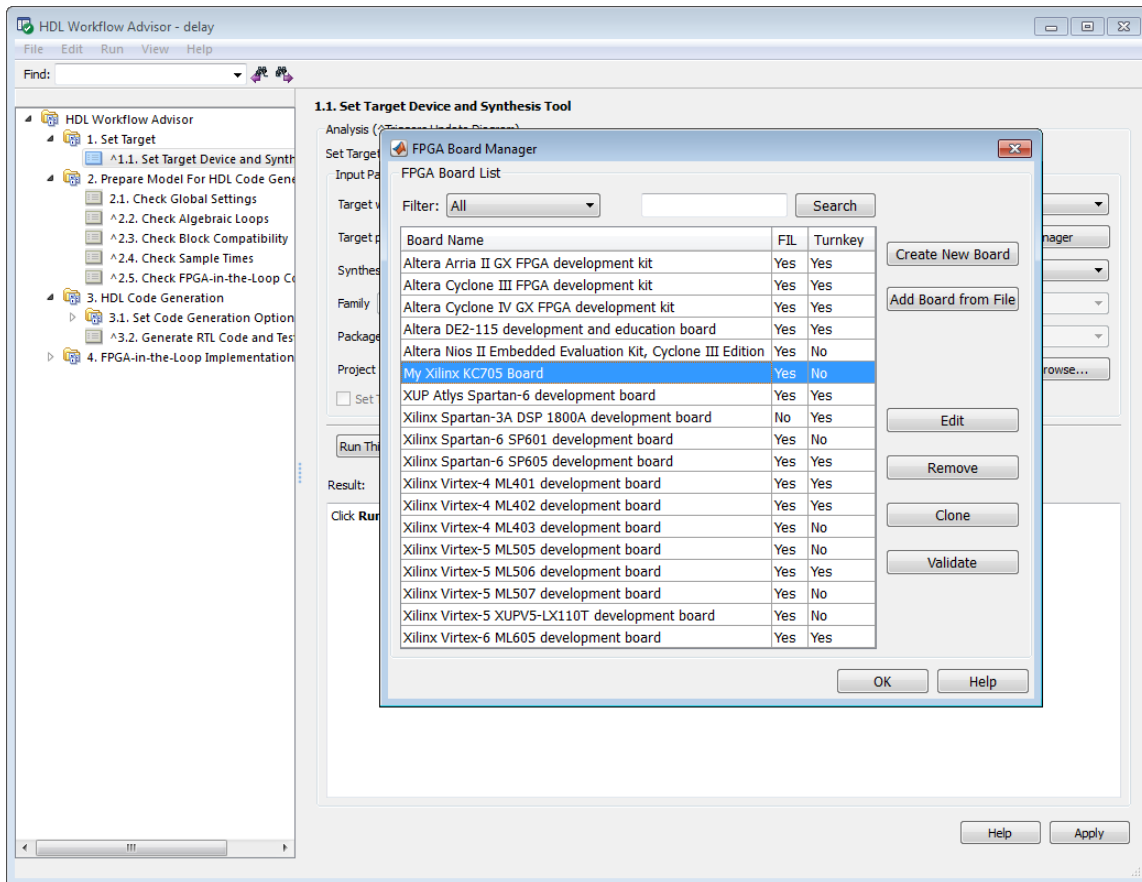
The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.



b Start HDL Workflow Advisor.

In step 1.1, select **FPGA-in-the-Loop** and click **Launch Board Manager**.

The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.



FPGA Board Manager

In this section...

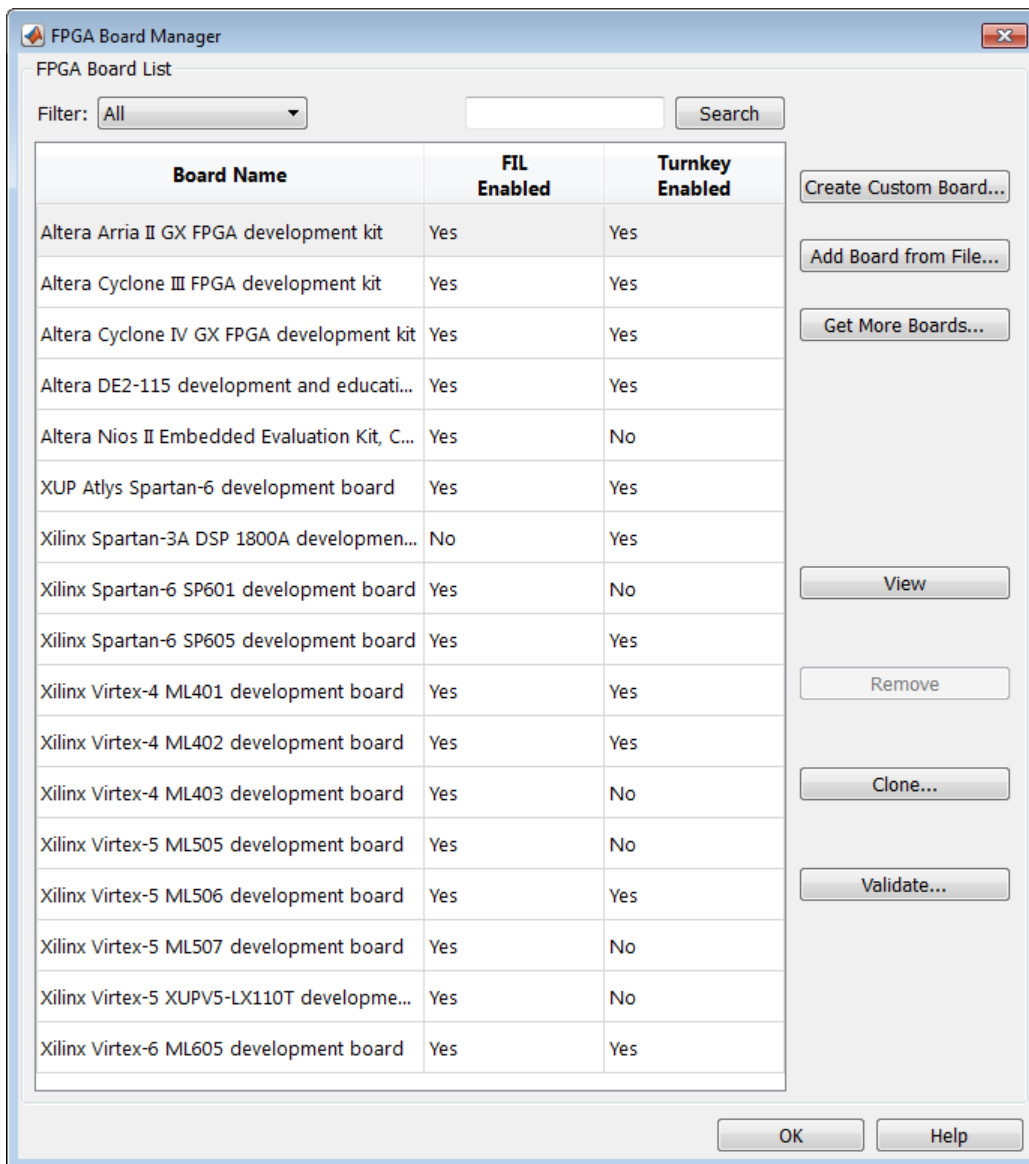
“Introduction” on page 16-18
“Filter” on page 16-19
“Search” on page 16-19
“FIL Enabled/Turnkey Enabled” on page 16-20
“Create Custom Board” on page 16-20
“Add Board from File” on page 16-20
“Get More Boards” on page 16-20
“View/Edit” on page 16-20
“Remove” on page 16-20
“Clone” on page 16-20
“Validate” on page 16-20

Introduction

The FPGA Board Manager is the portal to managing custom FPGA boards. You can create a board definition file or edit an existing one. You can even import a custom board from an existing board definition file.

You start the FPGA Board Manager by one of the following methods:

- By typing `fpgaBoardManager` in the MATLAB command window
- From the FIL wizard by clicking **Launch Board Manager** on the first page
- From the HDL Workflow Advisor (when using HDL Coder) at Step 1.1



Filter

Choose one of the following views:

- All boards
- Only those boards that were preinstalled with HDL Verifier or HDL Coder
- Only custom boards

Search

Find a specific board in the list or those boards that fully or partially match your search string.

FIL Enabled/Turnkey Enabled

These columns indicate whether the specified board is supported for FIL or Turnkey operations.

Create Custom Board

Start New FPGA Board wizard. See “New FPGA Board Wizard” on page 16-22. You can find the process for creating a board definition file in “Create Custom FPGA Board Definition” on page 16-6.

Add Board from File

Import a board definition file (.xml).

Get More Boards

Download FPGA board support packages for use with FIL

- 1 Click **Get more boards**.
- 2 Follow the prompts in the Support Package Installer to download an FPGA board support package.
- 3 When the download is complete, you can see the new boards in the board list in the FPGA Board Manager.

Offline Support Package Installation You can install an FPGA board support package without an internet connection. See “Install Support Package Offline” on page 12-3.

View/Edit

View board configurations and modify the information. You can view a read-only file but not edit it. See “FPGA Board Editor” on page 16-33.

Remove

Remove custom board from the list. This action does not delete the board definition XML file.

Clone

Makes a copy of an existing custom board for further modification.

Validate

Runs the validation tests for FIL See “Run Optional Validation Tests” on page 16-12.

See Also

Related Examples

- “Create Custom FPGA Board Definition” on page 16-6
- “Create Xilinx KC705 Evaluation Board Definition File” on page 16-7

New FPGA Board Wizard

Using the New FPGA Board wizard, you can enter all the required information to add a board to the FPGA board list. This list applies to both FIL and Turnkey workflows. Review “FPGA Board Requirements” on page 16-2 before adding an FPGA board to make sure that it is compatible with the workflow for which you want to use it.

Several buttons in the New FPGA Board wizard help with navigation:

- **Back:** Go to a previous page to review or edit data already entered.
- **Next:** Go to next page when all requirements of current page have been satisfied.
- **Help:** Open Doc Center, and display this topic.
- **Cancel:** Exit New FPGA Board wizard. You can exit with or without saving the information from your session.

Adding Boards Once for Multiple Users To add new boards globally, follow these instructions. To access a board added globally, all users must be using the same MATLAB installation.

- 1 Create the following folder:

matlabroot/toolbox/shared/eda/board/boardfiles

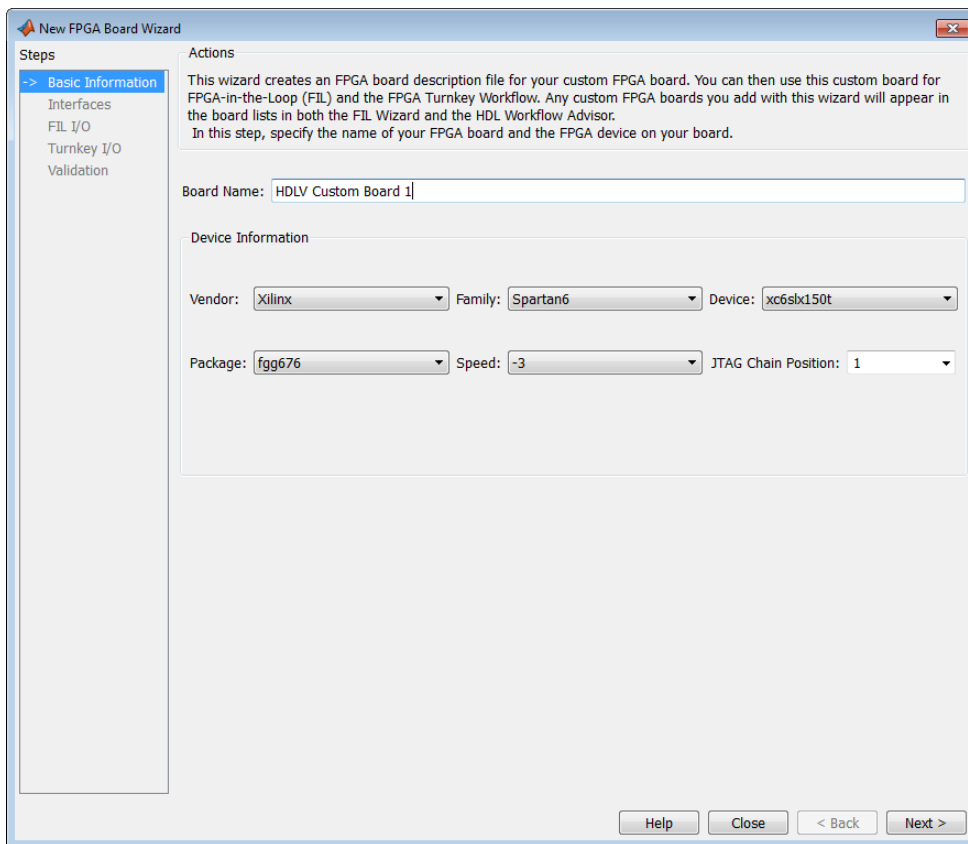
- 2 Copy the board description XML file to the *boardfiles* folder.
- 3 After copying the XML file, restart MATLAB. The new board appears in the FPGA board list for either or both the FIL and Turnkey workflows.

All boards under this folder show-up in the FPGA board list automatically for users with the same MATLAB installation. You do not need to use FPGA Board Manager to add these boards again.

The workflow for adding an FPGA board contains these steps:

In this section...
“Basic Information” on page 16-23
“Interfaces” on page 16-23
“FIL I/O” on page 16-26
“Turnkey I/O” on page 16-28
“Validation” on page 16-31
“Finish” on page 16-32

Basic Information



Board Name: Enter a unique board name.

Device Information:

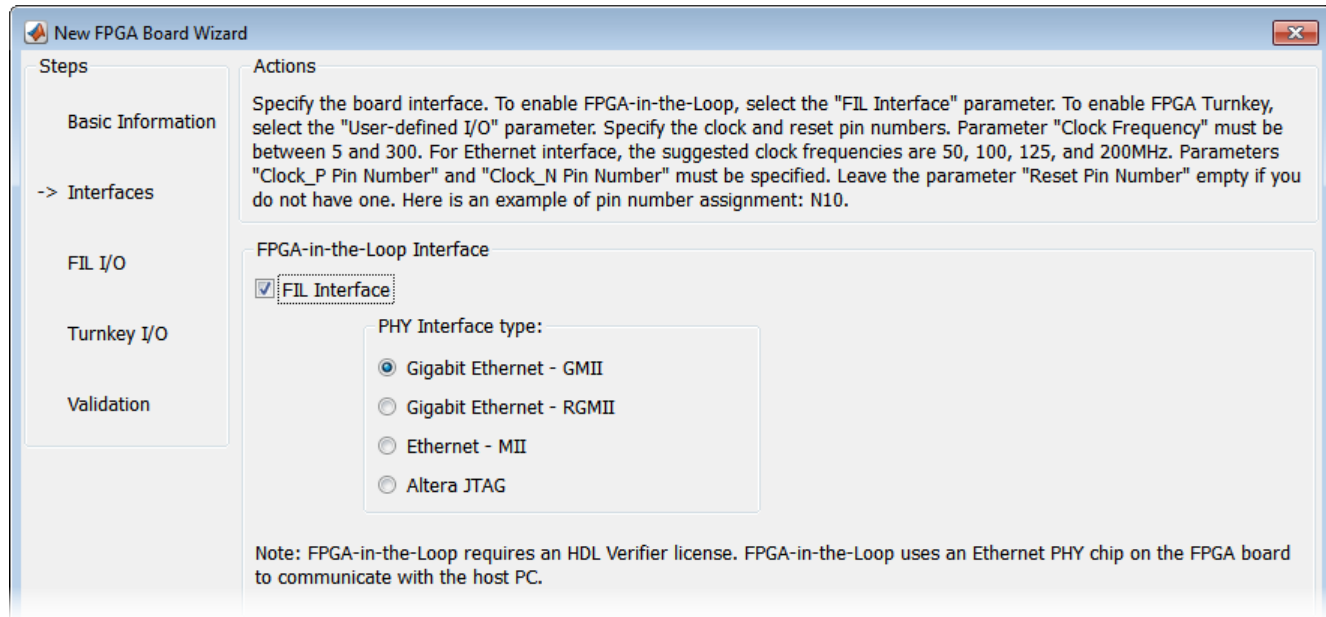
- **Vendor:** Xilinx or Altera
- **Family:** Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device:** Use the board specification file to select the correct device.
- For Xilinx boards only:
 - **Package:** Use the board specification file to select the correct package.
 - **Speed:** Use the board specification file to select the correct speed.
 - **JTAG Chain Position:** Value indicates the starting position for JTAG chain. Consult the board specification file for this information.

Interfaces

- “FIL Interface for Altera Boards” on page 16-24
- “FIL Interface for Xilinx Boards” on page 16-25
- “FPGA Turnkey Interface” on page 16-25

- “FPGA Input Clock and Reset” on page 16-26

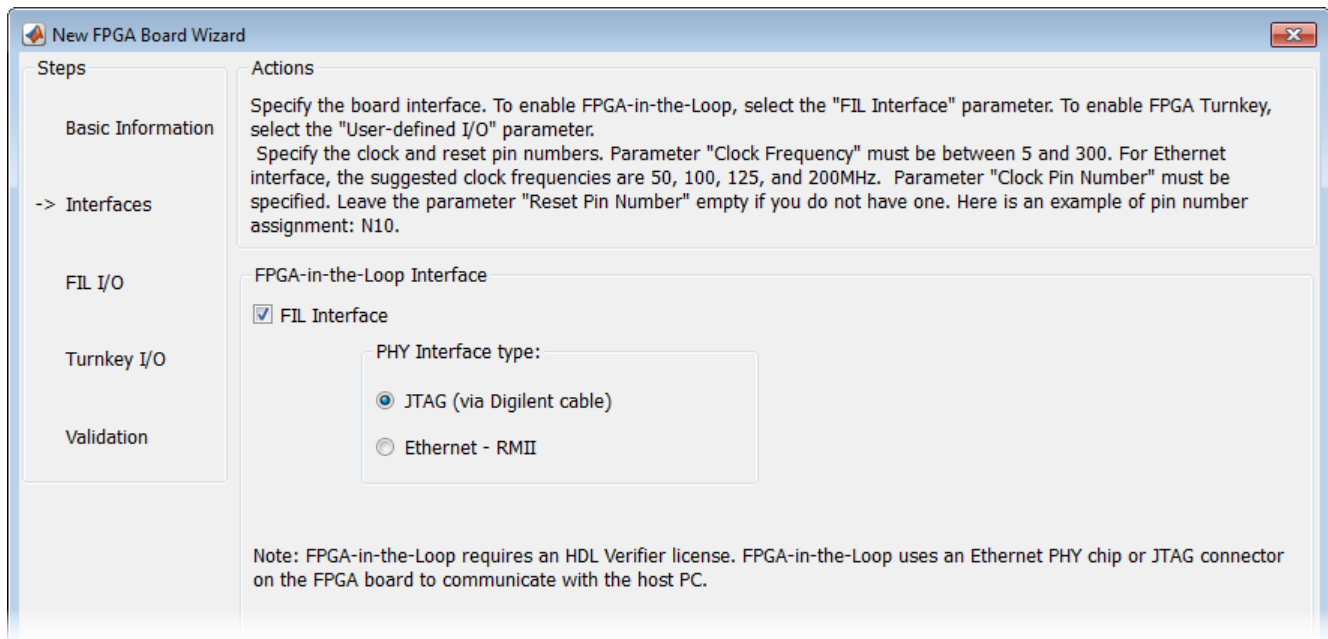
FIL Interface for Altera Boards



- 1 FPGA-in-the-Loop:** To use this board with FIL, select **FIL Interface**.
- Select one of the following **PHY Interface types**:
 - **Gigabit Ethernet – GMII**
 - **Gigabit Ethernet – RGMII**
 - **Gigabit Ethernet – SGMII** (the SGMII option appears if you select a board from the Stratix V or Stratix IV device families)
 - **Ethernet – MII**
 - **Altera JTAG** (Altera boards only)

Note Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

FIL Interface for Xilinx Boards



- 1 **FPGA-in-the-Loop Interface:** To use this board with FIL, select **FIL Interface**.
- 2 Select one of the following **PHY Interface types**:
 - **JTAG (via Digilent cable)** (Xilinx boards only)
 - **Ethernet – RMII**

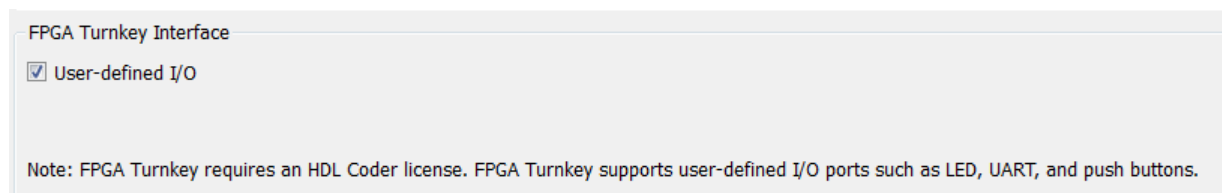
Note Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

For more information on how to set up the JTAG connection for Xilinx boards, see “JTAG with Digilent Cable Setup” on page 16-35.

Limitations

When you simulate your FPGA design through a Digilent JTAG cable, you cannot use any other debugging feature that requires access to the JTAG; for example, the Vivado Logic Analyzer.

FPGA Turnkey Interface



FPGA Turnkey Interface: If you want to use with board with the HDL Coder FPGA Turnkey workflow, select **User-defined I/O**.

FPGA Input Clock and Reset

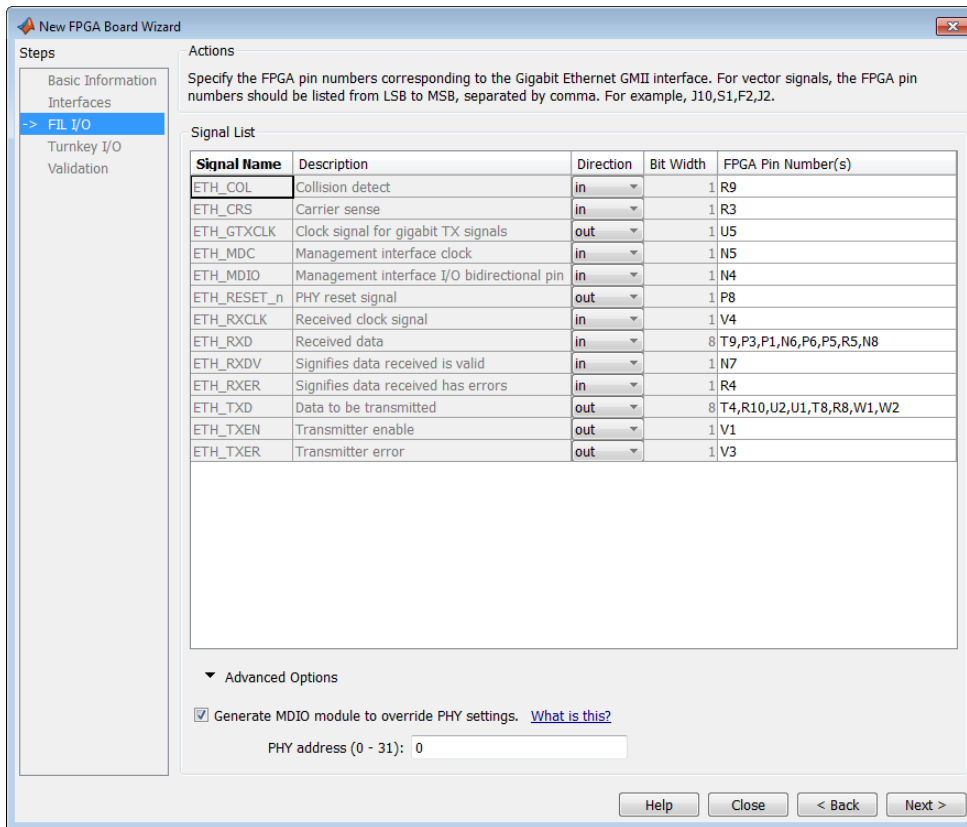
The screenshot shows a configuration form for FPGA Input Clock and Reset. It is divided into two main sections:

- FPGA Input Clock:**
 - Clock Frequency: 200 MHz
 - Clock Type: Differential
 - Clock_P Pin Number: E19
 - Clock_N Pin Number: E18
 - Clock IO Standard: LVDS
- Reset (Optional):**
 - Reset Pin Number: AV40
 - Active Level: Active-High
 - Reset IO Standard: LVCMOS18

- FPGA Input Clock** — Clock details are required for both workflows. You can find all necessary information in the board specification file.
 - Clock Frequency** — Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
 - Clock Type** — `Single_Ended` or `Differential`.
 - Clock Pin Number** (`Single_Ended`) — Must be specified. Example: N10.
 - Clock_P Pin Number** (`Differential`) — Must be specified. Example: E19.
 - Clock_N Pin Number** (`Differential`) — Must be specified. Example: E18.
 - Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, LVDS.
- Reset (Optional)** — If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.
 - Reset Pin Number** — Leave empty if you do not have one.
 - Active Level** — `Active-Low` or `Active-High`.
 - Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, LVCMOS33.

FIL I/O

When you select an Ethernet connection to your board, you must specify pins for the Ethernet signals on the FPGA.



Signal List: Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas.

Note If your PHY chip does not have the optional TX_ER pin, tie ETH_TXER to one of the unused pins on the FPGA.

Generate MDIO module to override PHY settings: See the next section on FPGA Board Management Data Input/Output Bus (MDIO) to determine when to use this feature. If you do select this option, enter the PHY address.

What Is the Management Data Input/Output Bus?

Management Data Input/Output (MDIO) is a serial bus, defined in the IEEE® 802.3 standard, that connects MAC devices and Ethernet PHY devices. The FPGA MAC uses the MDIO bus to set control registers in the Ethernet PHY device on the board.

Currently only the Marvell 88E1111 PHY chip is supported by this MDIO module implementation. Do not select this check box if you are not using Marvell 88E1111.

The generated MDIO module is used to perform the following operations:

- **GMII mode:** The PHY device can start up using other modes, such as RGMII/SGMII. The generated MDIO module sets the PHY chip in GMII mode.

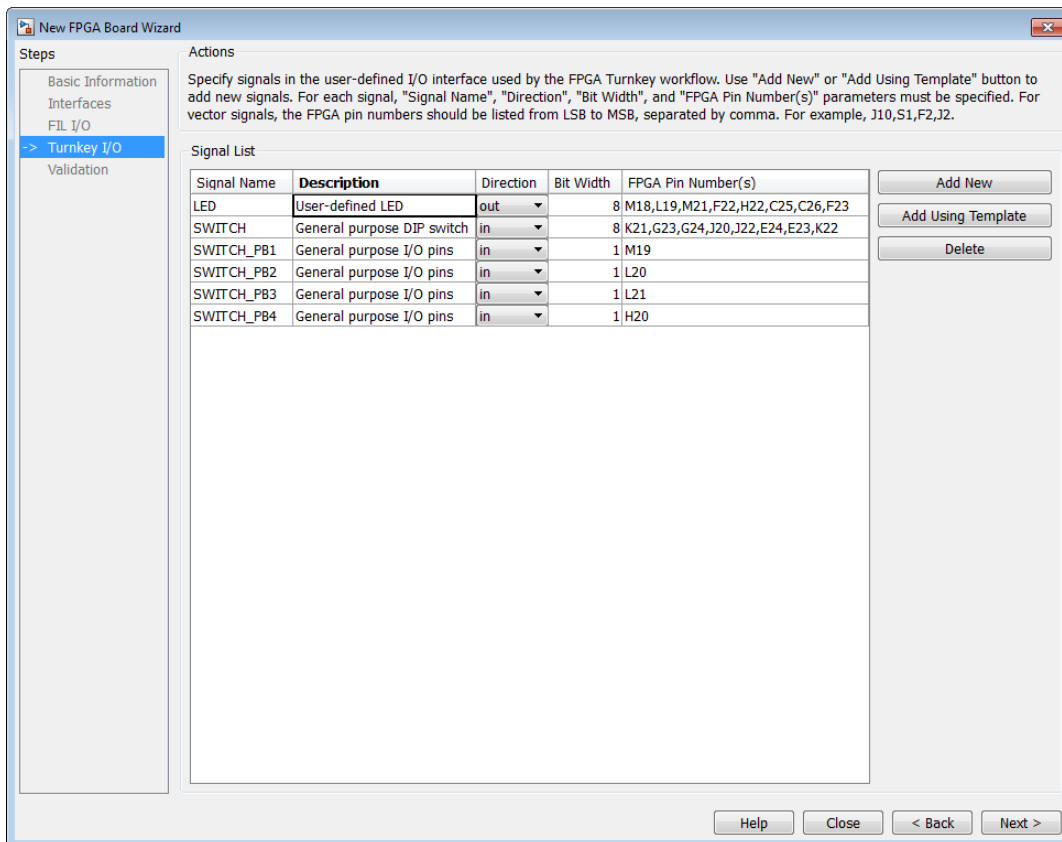
- **RGMI mode:** The PHY device can start up using other modes, such as GMII/SGMII. The generated MDIO module sets the PHY device in RGMI mode. In addition, the module sets the PHY chip to add internal delay for RX and TX clocks.
- **SGMI mode:** The PHY device can start up using other modes, such as RGMI/GMII. The generated MDIO module sets the PHY chip in SGMI mode.
- **MII mode:** The generated MDIO module sets the PHY device in GMII compatible mode. The module also sets the autonegotiation register to remove the 1000 Base-T capability advertisement. This reset ensures that the autonegotiation process does not select 1000 Mbits/s speed, which is not supported in MII mode.

When To Select MDIO: Select the **Generate MDIO module to override PHY settings** option when both the following conditions are met:

- The onboard Ethernet PHY device is Marvell 88E1111.
- The PHY device startup settings are not compatible with the FPGA MAC. The MDIO modules for different PHY modes must override these settings, as previously described.

Specifying the PHY Address: The PHY address is a 5-bit integer. The value is determined by the CONFIG[0] and CONFIG[1] pin on Marvell 88E1111 PHY device. See the board manual for this value.

Turnkey I/O



Note Provide FIL I/O for an Ethernet connection only. Define at least one output port for the Turnkey I/O interface.

Signal List: Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas. The number of pin numbers must match the bit width of the corresponding signal.

Add New: You are prompted to enter all entries in the signal list manually.

Add Using Template: The wizard prepopulates a new signal entry for UART, LED, GPIO, or DIP Switch signals with the following:

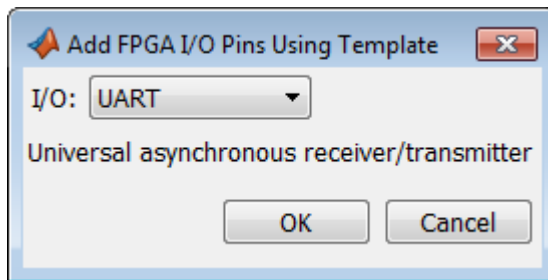
- A generic signal name
- Description
- Direction
- Bit width

You can change the values in any of these prepopulated fields.

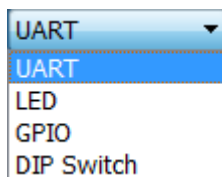
Delete: Delete the selected signal from list.

The following example demonstrates using the **Add Using Template** feature.

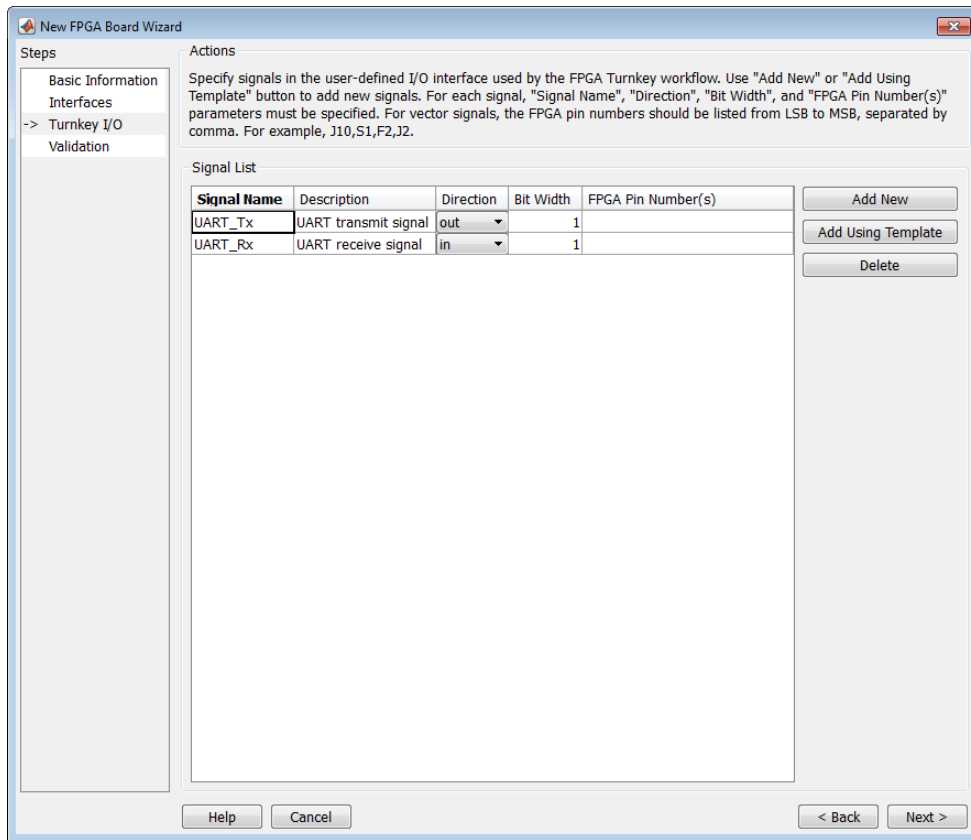
- 1 In the Turnkey I/O dialog box, click **Add Using Template**.
- 2 You can now view the template dialog box.



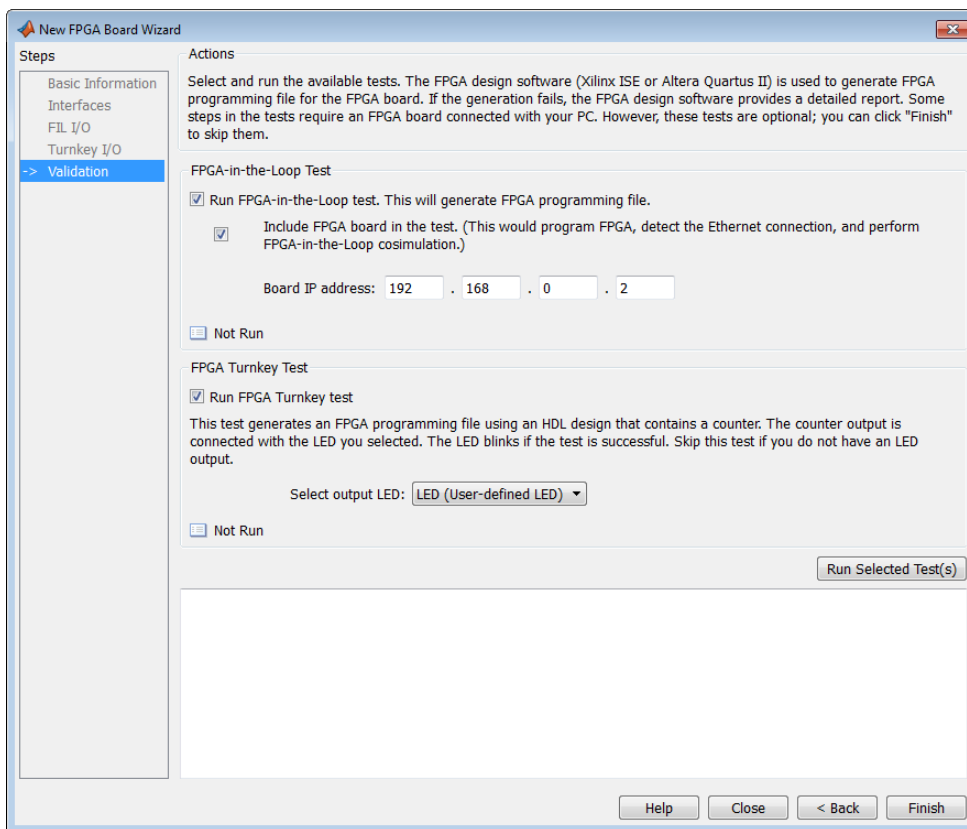
- 3 Pull down the I/O list and select from the following options:



- 4 Click **OK**.
- 5 The wizard adds the specified signal (or signals) to the I/O list.



Validation



FPGA-in-the-Loop Test

- **Run FPGA-in-the-Loop test:** Select to generate an FPGA programming file.
 - **Include FPGA board in the test:** (Optional) This selection program the FPGA with the generated programming file, detects the Ethernet connection (if selected), and performs FPGA-in-the-loop simulation.
 - **Board IP address:** (Ethernet connection only) Use this option for setting the board IP address if it is not the default IP address (192.168.0.2).

If necessary, change the computer IP address to a different subnet from 192.168.0.x when you set up the network adapter. If the default board IP address 192.168.0.2 is in use by another device, change the Board IP address according to the following guidelines:

- The subnet address, typically the first 3 bytes of board IP address, must be the same as the host IP address.
- The last byte of the board IP address must be different from the host IP address.
- The board IP address must not conflict with the IP addresses of other computers.

For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.

FPGA Turnkey Test

- **Run FPGA Turnkey test:** Select to generate an FPGA programming file using an HDL design that contains a counter. You must have a board attached.
- **Select output LED:** The counter's output is connected with the LED you select. Skip this test if you do not have an LED output.

Finish

When you have completed validation, click **Finish**. See "Save Board Definition File" on page 16-13.

FPGA Board Editor

In this section...

“General Tab” on page 16-33

“Interface Tab” on page 16-35

To edit a board definition XML file, first make it writeable. If the file is read-only, the FPGA Board Editor only lets you view the board configuration information. You cannot modify that information.

General Tab

Xilinx Virtex-7 VC707 development board - Copy (S:\Xilinx_pcie\zedboard\camera\matlab\new...)

Action

Specify your FPGA board information.

General Interface

Enter the basic information about your FPGA board such as board name, FPGA specification, and clock and reset pin numbers.

Board Name: My Xilinx Virtex-7 VC707 development board

File Location: S:\Xilinx_pcie\zedboard\camera\matlab\newboard - Copy.xml

Device Information

Vendor: Xilinx Family: Virtex7 Device: xc7vx485t

Package: ffg1761 Speed: -2 JTAG Chain Position: 1

FPGA Input Clock

Clock Frequency: 200 MHz Clock Type: Differential

Clock_P Pin Number: E19 Clock_N Pin Number: E18

Clock IO Standard: LVDS

Reset (Optional)

Reset Pin Number: AV40 Active Level: Active-High

Reset IO Standard: LVCMOS18

OK Cancel Help Apply

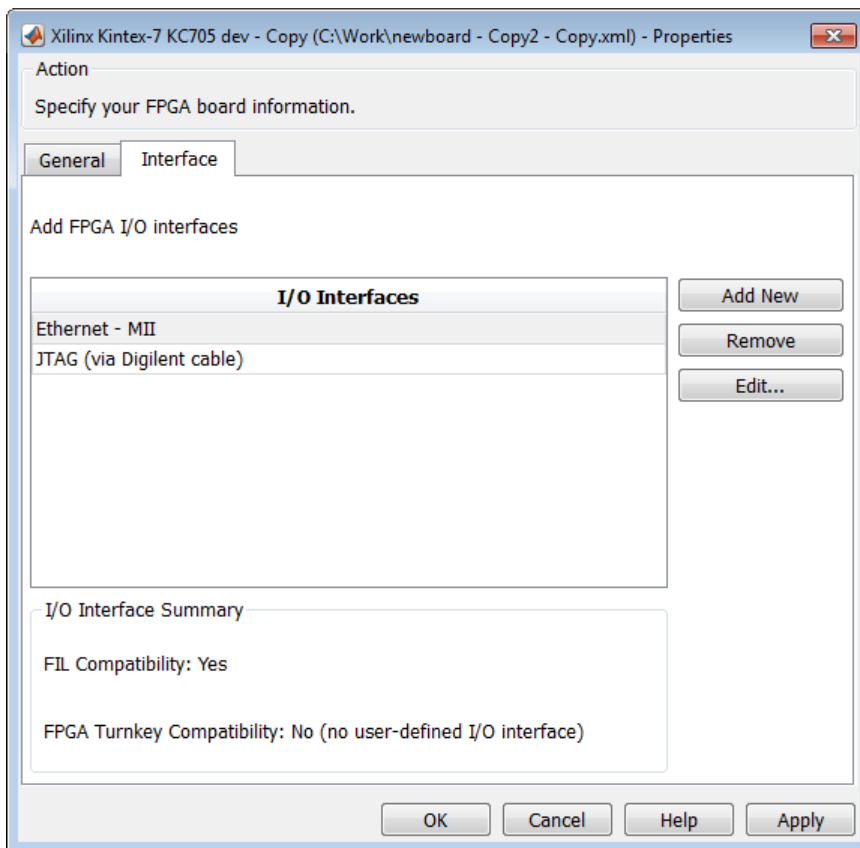
Board Name: Unique board name

Device Information:

- **Vendor:** Xilinx or Altera
- **Family:** Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device:** Device depends on the specified vendor and family. See the board specification file for applicable settings.
- For Xilinx boards only:

- **Package:** Package depends on specified vendor, family, and device. See the board specification file for applicable settings.
- **Speed:** Speed depends on package. See the board specification file for applicable settings.
- **JTAG Chain Position:** Value indicates the starting position for JTAG chain. Consult the board specification file for this information.
- **FPGA Input Clock.** Clock details are required for both the FIL and Turnkey workflows. You can find all necessary information in the board specification file.
 - **Clock Frequency.** Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
 - **Clock Type:** Single_Ended or Differential.
 - **Clock Pin Number** (Single_Ended) — Must be specified. Example: N10.
 - **Clock_P Pin Number** (Differential) — Must be specified. Example: E19.
 - **Clock_N Pin Number** (Differential) — Must be specified. Example: E18.
 - **Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, LVDS.
- **Reset (Optional).** If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.
 - **Reset Pin Number.** Leave empty if you do not have one.
 - **Active Level :** Active-Low or Active-High.
 - **Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, LVCMOS33.

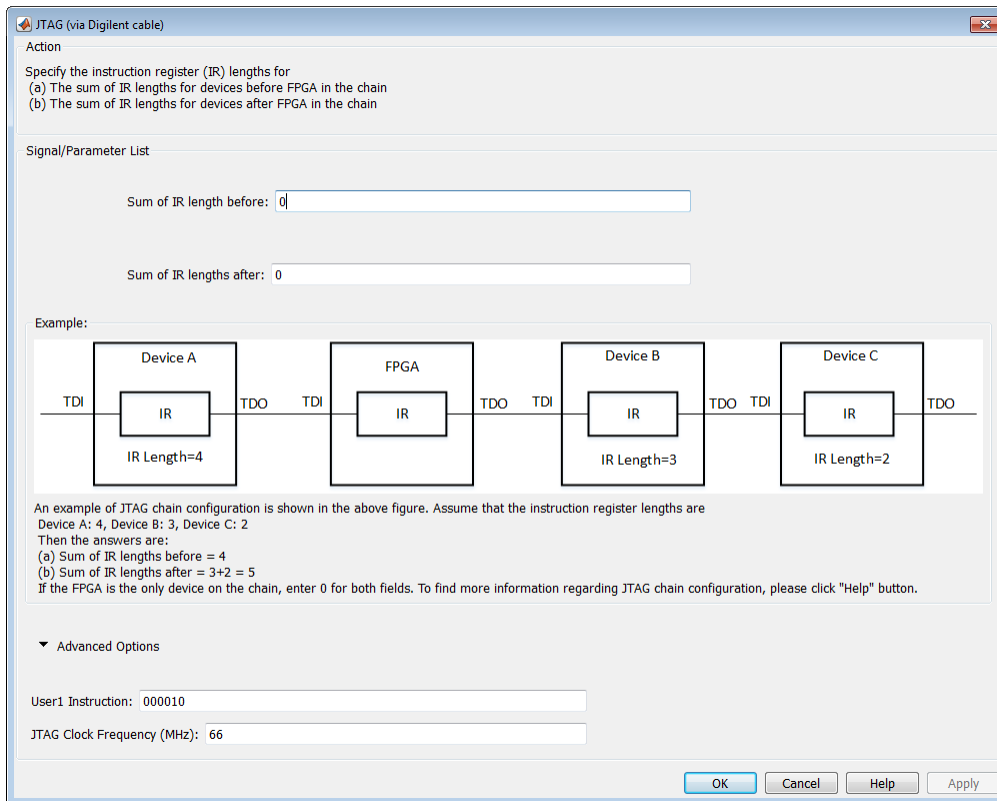
Interface Tab



The Interface page describes the supported FPGA I/O Interfaces. Select any listed interface and click **View** to see the **Signal List**. If the board definition file has write permission, you can also **Add New** interface, **Edit** the interface, or **Remove** an interface.

JTAG with Digilent Cable Setup

Note Enter information for the JTAG cable setup carefully. If the settings are incorrect, the simulation errors out and does not work. If you are still unsure about how to setup your JTAG cable after reading these instructions, contact MathWorks technical support with detailed information about your board.



- 1 **Signal/Parameter List** — Provide the sum of the lengths of the instruction registers (IR) for all devices before and after the FPGA in the chain.
 - If the FPGA is the only item in the device chain, use zeros in both **Sum of IR length before** and **Sum of IR length after**.
 - If you are using a Zynq device, and it is the only item in the device chain, enter 4 in **Sum of IR length before** and 0 in **Sum of IR length after**.

If your board does not meet either of those conditions, follow these instructions to obtain the IR lengths:

- a Connect the FPGA board to your computer using the JTAG cable. Turn on the board.
- b Make sure that you installed the cable drivers during Vivado installation.
- c Open Vivado Hardware Manager and select **Open a new hardware target**. In the dialog box is a summary of the IR lengths for all devices for that target.
- d Sum the IR lengths before the FPGA and enter the total in **Sum of IR length before**. Sum the IR lengths after the FPGA and enter the total in **Sum of IR length after**.

Vivado Hardware Manager cannot recognize the IR length of less common devices. For these devices, consult the device manual for instruction register length.

- 2 **Advanced Options** — If the default values are not the same as the most common settings for many devices, set the **User1 Instruction** and **JTAG Clock Frequency (MHz)** parameters. The most common settings are 000010 and 66, respectively.
 - **User1 Instruction** — The JTAG USER1 Instruction defined in the Xilinx Bscane2 primitive. This binary instruction number, defined by Xilinx, varies from device to device. For most of the

7-series devices, this instruction is 000010. If your device has a different value, enter it in this parameter.

To find this value, look at the `bsd` file for your specific device, found in your Vivado installation. For example, for the XA7A32T-CPG236 device, the `bsd` file is located in Vivado \2020.2\data\parts\xilinx\artix7\public\bsdl\xc7a35t_cpg236.bsd.

Open this file. The USER1 value is 000010. Enter this value at **User1 Instruction**.

```
"USER1      (000010),"
```

- **JTAG Clock Frequency (MHz)** — Clock frequency used by the JTAG circuit. This value varies by device. You can find this value in the same `bsd` file described under **User1 Instruction**. For example, the JTAG clock frequency is 66 MHz for device XA7A32T-CPG236:

```
attribute TAP_SCAN_CLOCK of TCK : signal is (66.0e6, BOTH);
```

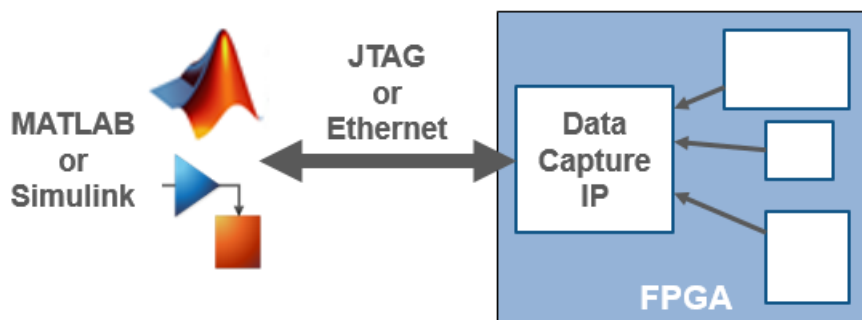

FPGA Data Capture

Data Capture Workflow

Use the FPGA data capture feature to observe signals from your design while the design is running on the FPGA. This feature captures a window of signal data from the FPGA and returns the data to MATLAB or Simulink over a JTAG or Ethernet interface.

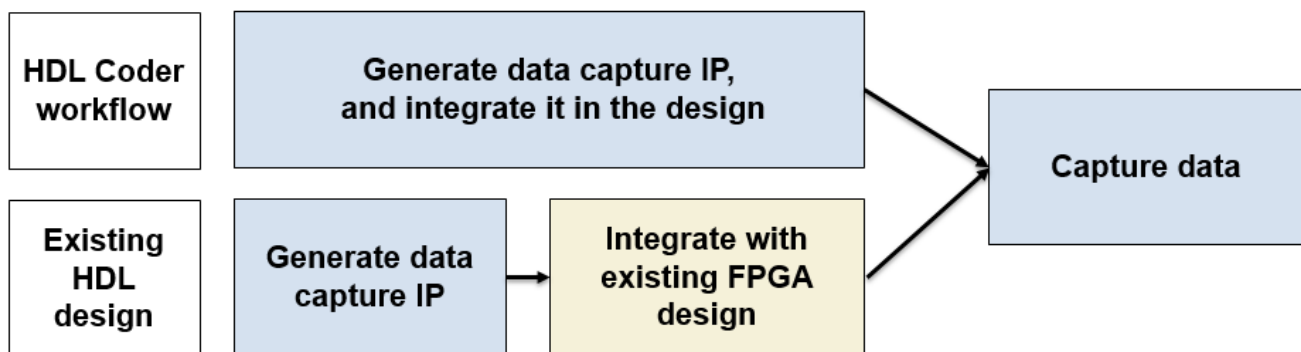
Note FPGA data capture support for JTAG connections is available for Intel and Xilinx FPGA boards. Support for Ethernet connections is available for Xilinx FPGA boards only.

To use this feature, you must download a hardware support package for your FPGA board. See “Download FPGA Board Support Package” on page 12-3.



You can choose between two workflows to capture data from your FPGA board and return it to MATLAB or Simulink.

- First workflow — If you generate the HDL IP with HDL Coder, use the **HDL Workflow Advisor** tool to generate the data capture IP and integrate it into your FPGA design.
- Second workflow — If you have an existing HDL design, use HDL Verifier tools to generate the data capture IP. Then, manually integrate the generated IP into your FPGA design.



To capture signals from your design, HDL Verifier generates an IP core that communicates with MATLAB. Use the HDL Coder workflow to automatically integrate the data capture IP core in your design. Otherwise, manually integrate this IP core into your HDL project and deploy it to the FPGA along with the rest of your design. Then, use one of these methods to capture data.

- For capturing data to MATLAB - HDL Verifier generates a customized app that returns the captured signal data. Alternatively, you can use the generated System object to capture data programmatically.
- For capturing data to Simulink - HDL Verifier generates a block that has output ports corresponding to the signals you captured.

In both cases, you can specify data types for the captured data, number of windows to capture, trigger condition that controls when to capture the data, and capture condition that controls which data to capture.

When the design is running on the FPGA, first the generated IP core waits for the trigger condition that you specify. Define a trigger condition by specifying values matched on one or more signals. When the trigger is detected, the logic captures the designated signals to a buffer and returns the data over the JTAG or Ethernet interface to the host machine. You can then analyze and display these signals in your MATLAB workspace or Simulink model.

To make the best use of the buffer size and capture only the valid data, you can also define a capture condition. Define a capture condition in the same way as you define the trigger condition. When both the trigger is detected and the capture condition is satisfied, the logic captures only the valid values of the designated signals.

Generate and Integrate Data Capture IP Using HDL Workflow Advisor

When you use **HDL Workflow Advisor** tool to generate your HDL design, first mark desired signals as “Configure Signals as Test Points” (Simulink) in Simulink. Configure your design using **HDL Workflow Advisor** tool to:

- Select the type of connection channel by setting the **FPGA Data Capture (HDL Verifier required)** parameter in the **Set Target Reference Design** task. For more information, see “Set Target Reference Design” (HDL Coder).
- Enable test point generation by selecting the **Enable HDL DUT port generation for test points** option in the **Set Target Interface** task. For more information, see “Set Target Interface” (HDL Coder).
- Connect test point signals to the FPGA Data Capture interface in the **Set Target Interface** task.
- Set up the buffer size and maximum sequence depth for data collection in the **Generate RTL Code and IP Core** task. To include capture condition logic in the IP core, select **Include capture condition logic in FPGA Data Capture**. For more information, see “Generate RTL Code and IP Core” (HDL Coder).

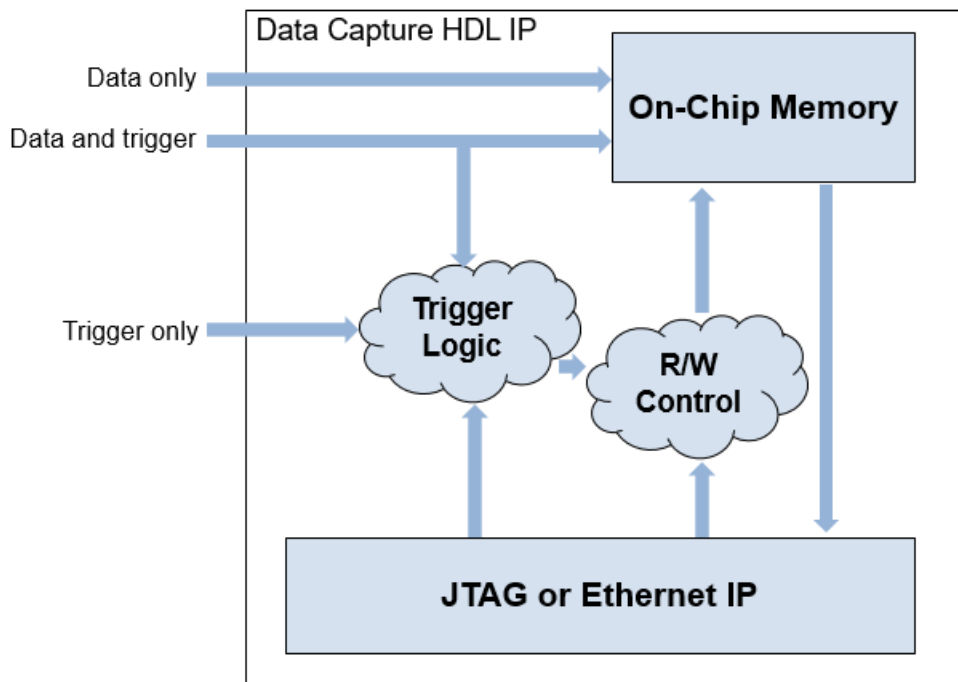
Then, execute the remaining steps to generate HDL for your design and program the FPGA. The data capture IP core is integrated into the generated FPGA design. You can now “Capture Data” on page 17-5 from the FPGA.

For an example of using data capture with **HDL Workflow Advisor**, see “Debug IP Core Using FPGA Data Capture” (HDL Coder).

Configure and Generate IP Core for an Existing HDL Design

Before capturing FPGA data, you must first specify which signals to capture and how many data samples to return. When using an existing HDL design, use the **FPGA Data Capture Component Generator** tool to configure settings and generate the data capture IP core. The IP core contains:

- A port for each signal you want to capture or use as part of a trigger condition
- Memory to capture the number of samples you requested for each signal
- JTAG or Ethernet interface logic to communicate with MATLAB
- Trigger and capture condition logic that can be configured at run time
- A ready-to-capture signal to control data flow from the FPGA



The **FPGA Data Capture Component Generator** tool also generates a customized **FPGA Data Capture** tool, System object, and model that communicate with the FPGA.

Integrate IP into FPGA

For MATLAB to communicate with the FPGA, you must integrate the generated HDL IP core into your FPGA design. If you used the **HDL Workflow Advisor** tool to generate your data capture IP, this step is automated. In this case, data capture IP operates on a single-clock rate, which is the primary clock of your design under test (DUT). If you did not use the **HDL Workflow Advisor** tool, follow these instructions in the generation report.

- 1 Create an FPGA project.
- 2 Navigate to the `hdlsrc` folder.
- 3 Follow one of these steps based on your connection type.
 - JTAG — Add the generated HDL files in the `hdlsrc` folder into your FPGA project. Then, instantiate the HDL IP core, `datacapture`, in your HDL code. Connect `datacapture` to the signals you requested for capture and triggers.
 - Ethernet — Run the `insertEthernet.tcl` script in the Vivado Tcl console by entering the source `./insertEthernet.tcl` command.

Compile the project and program the FPGA with the new image via a JTAG cable. You can now “Capture Data” on page 17-5 from the FPGA.

Capture Data

The FPGA data capture IP core communicates over the JTAG or Ethernet cable between your FPGA board and the host computer. Make sure that the required cable is connected. Before capturing data, you can set data types for the captured data, and set trigger conditions that specify when to capture the data. To configure these options and capture data, you can:

- Open the **FPGA Data Capture** tool. Set trigger, capture condition, and data type parameters, and then capture data into the MATLAB workspace.
- Use the generated System object derived from `hdlverifier.FPGADataReader`. Set the data types, trigger condition, and capture condition using the methods and properties of the System object, and then call the object to capture data.
- In Simulink, open the generated model, and configure the parameters of the FPGA Data Reader block. Then, run the model to capture data.

After you capture the data into the MATLAB workspace or Simulink model, you can analyze, verify, and display the data.

See Also

More About

- “Getting Started with the HDL Workflow Advisor” (HDL Coder)
- “Debug IP Core Using FPGA Data Capture” (HDL Coder)
- “Download FPGA Board Support Package” on page 12-3
- “HDL Verifier Support Package for Intel FPGA Boards”
- “HDL Verifier Support Package for Xilinx FPGA Boards”

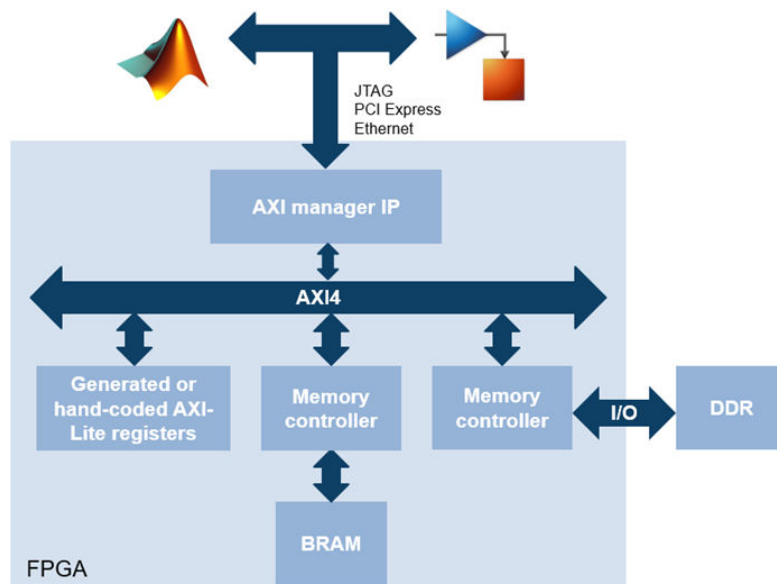
AXI Manager

Set Up for AXI Manager

Note MATLAB AXI master has been renamed to AXI manager. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

To access on-board memory locations from MATLAB or Simulink, you must include the AXI manager IP in your FPGA design. This IP connects to subordinate memory locations on the board. The IP also responds to read and write commands from MATLAB or Simulink, over JTAG, PCI Express (PCIe), or Ethernet cable.

To use this feature, you must download a hardware support package for your FPGA board. See "Download FPGA Board Support Package" on page 12-3.



To set up the AXI manager IP for access from MATLAB or Simulink, follow these setup steps:

- 1 Include the AXI manager IP in your FPGA design. To add the path for the IP files to your project, call the `setupAXIManagerForVivado` or `setupAXIManagerForQuartus` functions.
- 2 Open Vivado or Quartus, and from the IP Catalog select the AXI manager IP in your FPGA design.
 - When using JTAG as a physical connection, select AXI Manager.
 - When using Ethernet as a physical connection, select UDP AXI Manager and Ethernet MAC Hub and add them to your project.
 - When using PCIe as a physical connection, select PCIe AXI Manager and add it to your project.
- 3 In your FPGA project, specify which addresses the AXI manager IP is allowed to access.

Note The AXI manager IP supports AXI4 Lite, AXI4, and Altera Avalon slave memory locations. The FPGA interconnect automatically converts AXI4 transactions to the protocol of each address.

- 4 Compile your FPGA project, including the AXI manager IP.

- 5 Connect your FPGA board to your host computer using a physical cable (JTAG, PCIe, or Ethernet cable).
- 6 Program the FPGA with your compiled design.

Note Alternatively, you can perform these steps in the HDL Coder guided workflow by using a sample reference design, such as the one included in these examples: “Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow” (HDL Coder) or “Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow” (HDL Coder).

After loading the design on your FPGA, you can access memory-mapped locations on the board.

To access the board from MATLAB, create an `axi_manager` object and use the `readmemory` and `writememory` methods to read and write memory-mapped locations on the board.

To access the board from Simulink, create a Simulink model and include AXI Manager Read and AXI Manager Write in it. Configure the blocks to read and write memory-mapped locations on the board.

JTAG Considerations

When using JTAG as a physical connection to your board, you might have additional IPs that use the same JTAG connection. Such IPs include FPGA data capture, Intel SignalTap II, or Xilinx Vivado Logic Analyzer cores. The AXI manager IP can coexist in your design with other IPs that use the JTAG connection, however, only one of these applications can use the JTAG cable at a time. Release the `axi_manager` object to return the JTAG resource for use by other applications.

The most common conflicting use of the JTAG cable is to reprogram the FPGA. Stop any FPGA data capture or AXI manager JTAG connection before you can use the cable to program the FPGA.

The maximum data rate between host computer and FPGA is limited by the JTAG clock frequency. For Intel boards, the JTAG clock frequency is 12 MHz or 24 MHz. For Xilinx boards, the JTAG clock frequency is 33 MHz or 66 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

See Also

More About

- “Download FPGA Board Support Package” on page 12-3
- “HDL Verifier Support Package for Intel FPGA Boards”
- “HDL Verifier Support Package for Xilinx FPGA Boards”

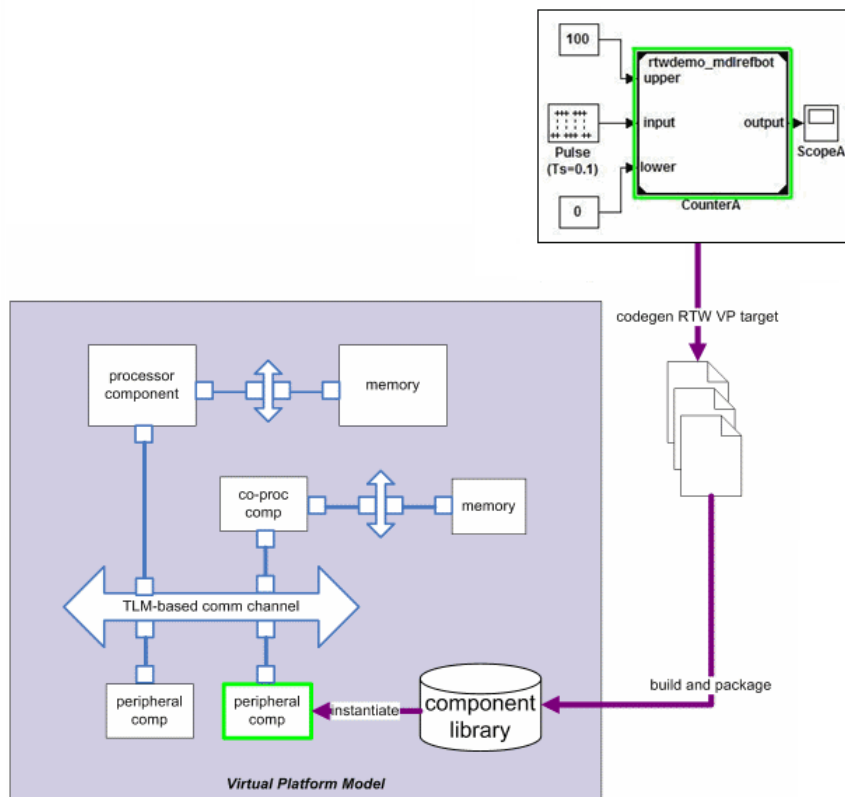
SystemC TLM Generation

How TLM Component Generation Works

- “TLM Generation Algorithms” on page 19-2
- “The TLM Generation Process” on page 19-3
- “Generated TLM Files” on page 19-5

TLM Generation Algorithms

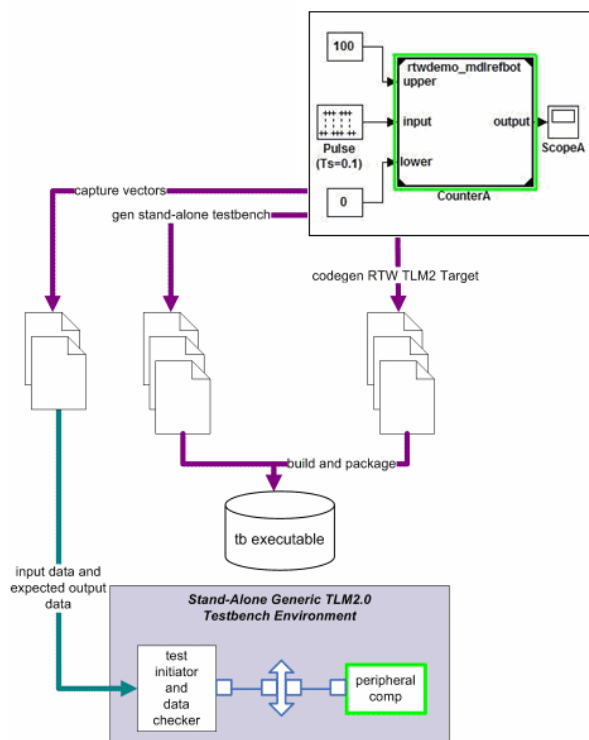
The algorithm you use to generate the TLM component can be made of any combination of Simulink blocks that can generate C code. These blocks generally belong to a subsystem. Simulink Coder™ software generates ANSI® C code from those blocks that HDL Verifier software then customizes with the settings specified using the TLM component generator to create the files that make up the virtual platform model. For an example of how this process works, see the following illustration.



The TLM Generation Process

After you obtain the TLM component files generated by HDL Verifier software, you can compile the TLM component and the optional test bench with OSCI SystemC libraries and the OSCI TLM libraries. To do so, use the makefile supplied by HDL Verifier to create your virtual platform executable (e.g., mysimulation.exe).

The following diagram illustrates the complete set of artifacts you can generate including the TLM component, the TLM component test bench, and the set of test vectors to be executed by the test bench. Simulink generates these vectors while performing model execution when you verify the TLM component from within Simulink (see “Run TLM Component Test Bench” on page 23-5).



The following general workflow describes the process for creating an OSCI-compatible TLM component representing the Simulink algorithm:

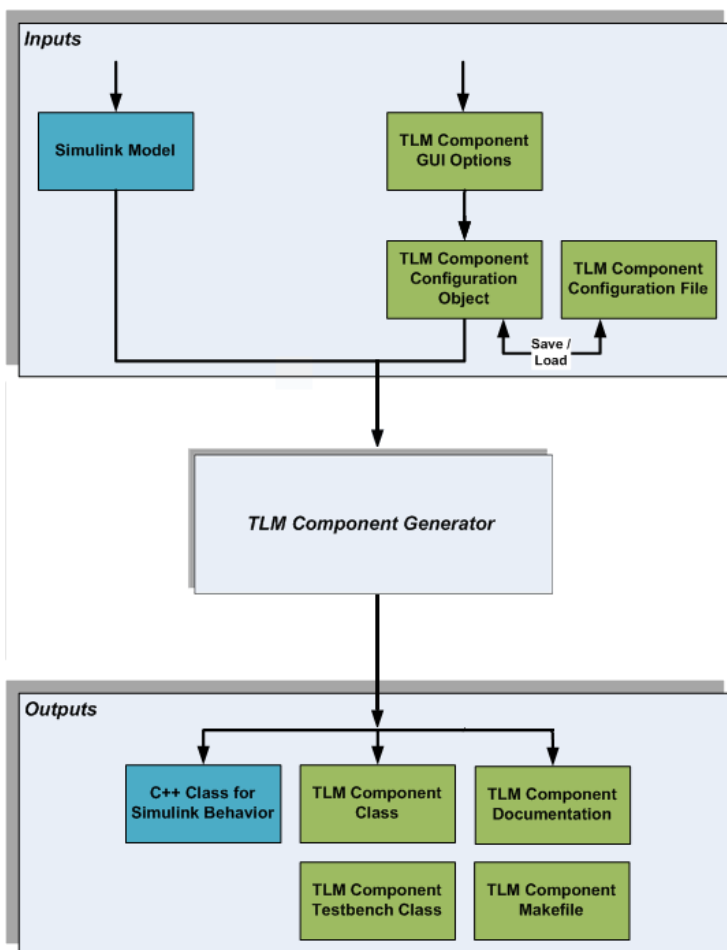
- 1 Create Simulink model representing algorithm.
- 2 Select required architectural model (i.e., virtual platform model) parameters via the Simulink Configuration Parameters dialog box. See “Subsystem Guidelines and Limitations” on page 22-3.
- 3 (Optional) If you want, restore any desired configuration sets at this time. Because the topic of configuration sets is outside the scope of this workflow description, refer to the section “Model Reference Basics” (Simulink) in the Simulink documentation.
- 4 Initiate code generation.

- 5 Save configuration options with the model for future use.

Generated TLM Files

HDL Verifier software generates the following files:

- C/C++ code containing the Simulink model behavior (.cpp and .h files)
- Virtual platform TLM component class (.cpp and .h files)
- TLM component documentation (HTML)
- TLM component test bench (if specified) (.cpp and .h files)
- Test bench stimulus and expected response vectors (MATLAB formatted data)
- Makefiles for building the TLM component and standalone test bench (makefile format)
- IP-XACT XML file. For details, see “Contents of Generated IP-XACT File” on page 22-24.



After code generation is complete, you can then use these generated files (outputs) to create the standalone TLM executable. See “Export TLM Component” on page 24-2.

TLM Component Architecture

TLM Component Architecture

In this section...

“Overview of Component Features” on page 20-2

“Memory Mapping” on page 20-3

“Command and Status Register” on page 20-7

“Interrupt” on page 20-10

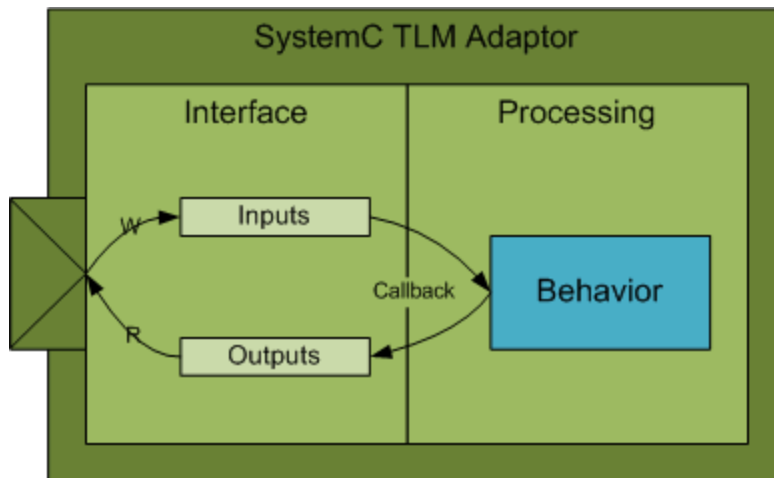
“Test and Set Register” on page 20-11

“Registers and Signal Ports” on page 20-11

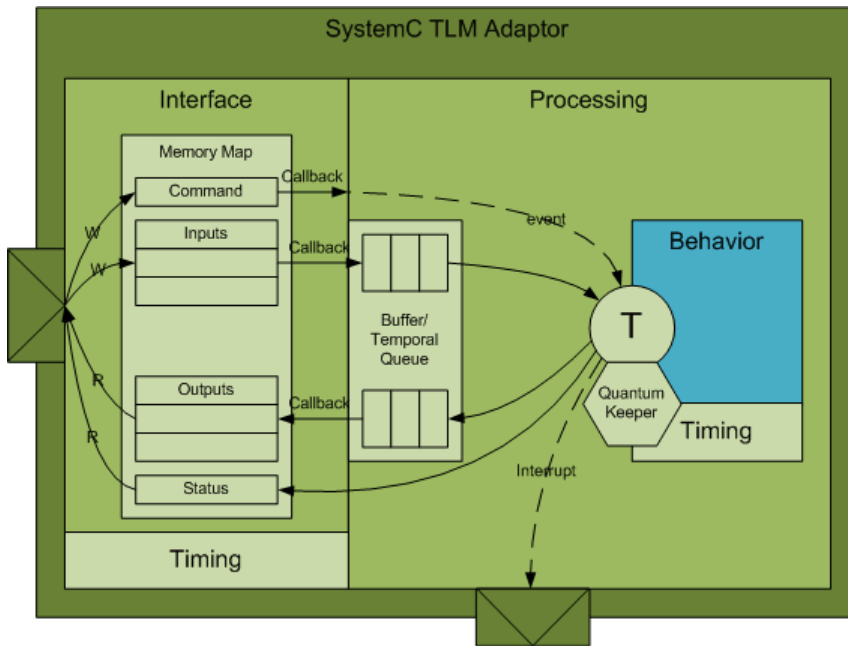
Overview of Component Features

The TLM generator exports a target TLM component from a Simulink model subsystem. The target TLM component has a single TLM socket that supports read and write transactions using the TLM generic protocol and generic payload.

The following diagram illustrates the simplest behavior you can specify for the generated TLM component. It contains no memory map or command and status register, and executes transactions immediately.



To control the architecture of the generated TLM component, you can choose among several options. Incorporating a memory map is one of the most effective options. The following figure demonstrates the behavior of a generated TLM component with a full complement of features enabled.



You can set options for the following TLM component features:

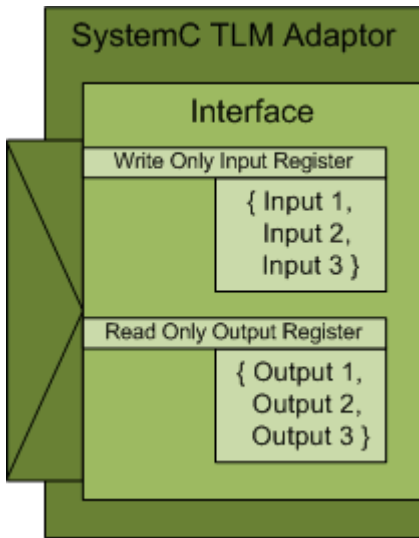
- “Memory Mapping” on page 20-3
- “Command and Status Register” on page 20-7
- “Interrupt” on page 20-10
- “Test and Set Register” on page 20-11
- “Registers and Signal Ports” on page 20-11
- Interface Timing — Model time used by transactions in a real system.
- Algorithm Execution — Implement the component as a SystemC thread or a callback function.

Memory Mapping

- “No Memory Map” on page 20-3
- “Automatically Generated Memory Map with Single Address” on page 20-4
- “Automatically Generated Memory Map with Individual Addresses” on page 20-6

No Memory Map

The no memory map option generates a TLM component with only one read and one write register without any address. The Simulink model inputs are represented by the write register and the outputs are represented by the read register.



TLM Generic Payload			
Command	Address	Length	Data
Write	N/A	Input Reg. Size	{ Input 1, Input 2, Input 3 }
Read		Output Reg. Size	{ Output 1, Output 2, Output 3 }

Without a memory map, the generated TLM component has the following characteristics:

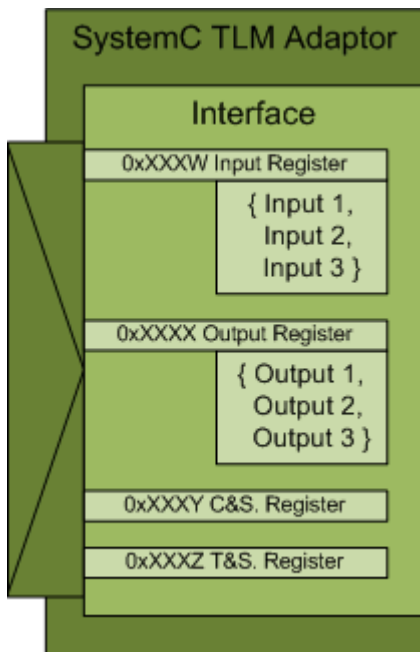
- Has a single input register and a single output register.
- Does not need—and ignores—an address in the read and write requests during SystemC simulation to select specific registers on the device.
 - Receives all input data in a single write request, and a read request receives all output data in the return value
- Has input and output registers either sized to hold an entire data set required or created by the TLM component when it executes the behavior (algorithm step function) in your virtual platform environment
- When input registers are full, this condition triggers (schedules) execution of the behavior in the SystemC simulator. Output registers are handled the same way.
- All defaults for commands and status are applied.

When you generate the TLM component with this option, you can use it in a virtual platform (VP) as:

- A standalone component in a verification test bench
- A direct bound co-processing unit
- A device attached to a communication channel using a protocol adapter

Automatically Generated Memory Map with Single Address

The automatically generated memory map with single address option generates a TLM component with only one read data register and one write data register with one address each.



TLM Generic Payload			
Command	Address	Length	Data
Write	0xXXXW	Input Reg. Size	{ Input 1, Input 2, Input 3 }
Read	0xXXXX	Output Reg. Size	{ Output 1, Output 2, Output 3 }
Read/Write	0xXXXY	CSR Reg. Size	Command & Status Reg. Data
Read/Write	0xXXXZ	TSR Reg. Size	Test & Set Reg. Data

The Simulink model inputs are represented by the write register, and the outputs are represented by the read register. HDL Verifier software automatically assigns the addresses required to access those specific registers during code generation. Those addresses give the specific offsets required to address each individual register via read and write operations. Definition of the base address for the entire generated TLM component should be defined by the virtual platform that the TLM component resides in. The offset address definitions appear in a definition file that is generated along with the TLM component.

With a single address memory map, the generated TLM component has the following characteristics:

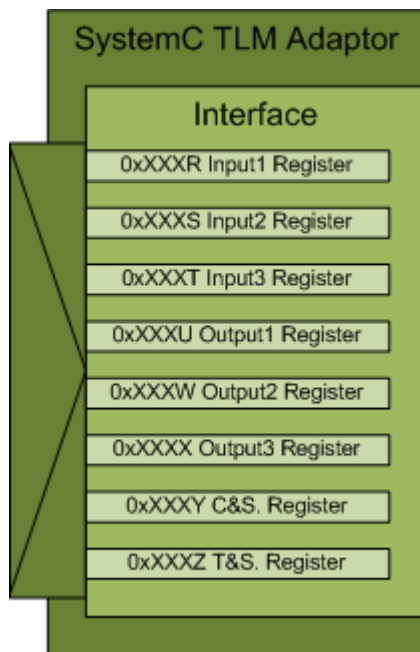
- Has a single input register and a single output register, and optional command and status register and test and set register.
- Must have an address in the read and write requests during SystemC simulation to select specific registers on the device.
 - Receives all input data in a single write request, and a read request receives all output data in the return value

- Has input and output registers either sized to hold an entire data set required or created by the TLM component when it executes the behavior (algorithm step function) in your virtual platform environment
- If a command and status register is not used or if the command and status register is used and the default values apply, when input register is full, content is pushed into buffer, which then triggers (schedules) execution of the behavior in the SystemC simulator. If the command and status register is used and the Push Input Command is set to 1, the initiator module moves the input data set from the input register to the input buffer. Output registers are handled the same way.
- If a command and status register is not used, all defaults for commands and status are applied.

When you generate the TLM component with this option, you can use it in a virtual platform (VP) as a standalone component in a test bench, or you can attach it to a communication channel.

Automatically Generated Memory Map with Individual Addresses

The automatically generated memory map with individual address option generates a TLM component with one read data register per model output and write data register per model input with individual addresses.



TLM Generic Payload			
Command	Address	Length	Data
Write	0xFFFFR	Input1 Reg. Size	Input 1 Data
Write	0xFFFFS	Input2 Reg. Size	Input 2 Data
Write	0xFFFFT	Input3 Reg. Size	Input 3 Data
Read	0xFFFFU	Output1 Reg. Size	Output 1 Data
Read	0xFFFFW	Output2 Reg. Size	Output 2 Data
Read	0xFFFFX	Output3 Reg. Size	Output 3 Data
Read/Write	0xFFFFY	CSR Reg. Size	Command & Status Reg. Data
Read/Write	0xFFFFZ	TSR Reg. Size	Test & Set Reg. Data

Each Simulink model input is represented by its corresponding write register, and each output is represented by its corresponding read register. HDL Verifier software automatically assigns the addresses required to access those specific registers during code generation. Those addresses give the specific offsets required to address each individual register via read and write operations. Definition of the base address for the entire generated TLM component should be defined by the virtual platform that the TLM component resides in. The offset address definitions appear in a definition file that is generated along with the TLM component.

With an individual address memory map, the generated TLM component has the following characteristics:

- Each input register and each output register has its own address as well as an optional command and status register and test and set register.
- Must have an address in the read and write requests during SystemC simulation to select specific registers on the device.
 - Each input and output register must be accessed individually.
- Initiator module can write or read each input and output register in multiple and/or partial transactions.
- The size of each input and output register is the size of the data.
- Execution is triggered when all input has been written or when command and set register bits are set to Automatic. If set to manual, the initiator module moves the input data set from the input register to the input buffer.
- Output registers are refreshed when all output registers have been read or when command and set registers bits are set to Automatic. If set to manual, the initiator module moves the output data set from the output buffer to the output register.

When you generate the TLM component with this option, you can use it in a virtual platform (VP) as a standalone component in a test bench, or you can attach it to a communication channel.

Command and Status Register

You can choose to generate a TLM component with an automatically generated memory map with addresses. When you do so, the TLM generator offers you the option to incorporate a Command and Status register (CSR) in the generated TLM component. The definition for this register appears in the table.

Write-Only Bits

Write-only (WO) bits assert mutually exclusive commands. You can assert only one command bit in any single write operation to the CSR. If more than one command bit is set in the write to the CSR, the command is undefined. You activate each command by writing a 1 to a command bit in the register. Then, each command bit is automatically cleared after the command has been executed. You do not have to write a 0 to the register to clear a command bit. Write-Only bits are always returned as 0 in any read of the CSR. Writing a command does not overwrite the Read/Write or Write-Only bits.

Read-and-Write Bits

Use Read and Write (R/W) bits to obtain the current status and setting. R/W bit are sticky, meaning that after you set them by writing a 1 to the bit in the register, an R/W bit remains set until a 0 is written to the same bit or the Reset command is invoked. Read-and-Write bits return their actual values to any read of the CSR.

A single write operation to the CSR sets all Read-and-Write bits in the register. You can choose to set only some of the bits and maintain the previous values of others. Before you do so, you must first read the CSR and then modify the values according to your requirements. After you complete modifications, you can write the entire 32 bits back to the CSR.

Read-Only Bits

Read-Only (RO) bits provide status information. The generated TLM component automatically sets and clears their values, and an initiator module can read them to learn status. Read-Only bits do not change their actual values during any read or write of the CSR.

Register Definition

The following table contains the entire register definition.

<7>	<6>	<5>	<4>	<3>	<2>	<1>	<0>
Reserved				Interrupt Disable	Interrupt Status	Start	Reset
				R/W	RO	WO	WO
<15>	<14>	<13>	<12>	<11>	<10>	<9>	<8>
Reserved		Output Auto Mode	Pull Output	Reserved		Input Auto Mode	Push Input
		R/W	WO			R/W	WO
<23>	<22>	<21>	<20>	<19>	<18>	<17>	<16>
Reserved							
<31>	<30>	<29>	<28>	<27>	<26>	<25>	<24>
Reserved							

The following table explains how the bits are defined.

Bit	Name	Read/Write Status	Description
CSR<0>	Reset Command	Write-only	<p>When set to 1, the following are true:</p> <ul style="list-style-type: none"> • Input register contents are made invalid. • Output register contents are made invalid. • All CSR bits are set to 0 except the following: <ul style="list-style-type: none"> • Input Buffer Empty bit is set to 1. • Output Buffer Empty bit is set to 1. • Input Auto Mode is set to default. • Output Auto Mode is set to default. <p>Automatically returns to 0 after command execution.</p>
CSR<1>	Start Command	Write-only	<p>Manually triggers execution of the TLM component behavior using the input data set that is currently in the input register when there is no input buffering.</p> <p>When input buffering is used, this command is undefined.</p>
CSR<2>	Interrupt Status	Read-only	<p>Reflects the current state of the Interrupt signal. Provides status only. Sets and clears itself automatically.</p>
CSR<3>	Interrupt Disable	Read-and-write	<p>When set to 0, allows interrupts to be generated on the Interrupt signal and reflected in the Interrupt Status bit of the CSR.</p> <p>When set to 1, disables generation of interrupts.</p>
CSR<8>	Push Input Command	Write-only	<p>When buffering is used and the Input Mode is equal to 0 (manual mode), this command allows an initiator module to move the input data set from the input register to the input buffer. It then triggers execution of the TLM component behavior.</p> <p>When buffering is not used, this command is undefined.</p> <p>When Input Mode is 1 (automatic), this command is undefined.</p>

Bit	Name	Read/Write Status	Description
CSR<9>	Input Mode	Read-and-write	<p>When set to 1 (automatic), movement of the input data set from the input register to the input buffer and execution of the TLM component behavior is triggered automatically, if a complete data set has been written to the input register.</p> <p>When set to 0 (manual), movement of the input data set from the input register to the input buffer and execution of the behavior must be manually initiated. Do so by writing the Start Command bit to 1, if no buffering is used, or writing the Push Input Command to 1, if buffering is present.</p> <p>By default the Input Mode is set to 1 (automatic). To change it to 0 (manual), specify it in the TLM component constructor parameters.</p>
CSR<12>	Pull Output Command	Write-only	<p>When buffering is used and the Output Mode is set to 0 (manual mode), this command allows an initiator module to move the output data set from the head of the output buffer to the output register.</p> <p>When buffering is not used, this command has no effect.</p> <p>When Output Mode is 1 (automatic), this command is undefined.</p>
CSR<13>	Output Mode	Read-and-write	<p>When set to 1 (automatic), movement of data from the head of the output buffer to the output register is triggered automatically by the execution of the TLM component behavior.</p> <p>When set to 0 (manual), movement of data from the head of the output buffer to the output register must be manually initiated. Do so by writing the Pull Output Command to 1, if buffering is present.</p> <p>By default the Output Mode is set to 1 (automatic). To change it to 0 (manual), specify it in the TLM component constructor parameters.</p>

Interrupt

You can add an interrupt signal added to the generated TLM component. The TLM component asserts this signal whenever new outputs are available in any output register. The signal is automatically cleared whenever a value is read from any output register.

The Interrupt signal is a SystemC Boolean signal active high. The Interrupt Active bit in the Status Register reflects the state of the interrupt signal.

Test and Set Register

You can use the optional test and set register to control access to a shared TLM component in your SystemC environment. Any read of this register returns the current value and sets the register to a new, asserted value in an atomic operation. In systems with multiple initiator modules, executing this task usually requires access to the same target. If so, then an initiator module has exclusive access to the generated TLM component, as long as all other initiator modules follow. The initiator modules must read the test and set register and use the target device only when that read operation returns a value of 0. An initiator module can verify that any subsequent read of the test and set register returns a value of 1, which indicates to other initiator modules that the device is busy. After gaining exclusive access to the TLM component, an initiator module releases the component when the target operations complete by writing a 0 to the test and set register.

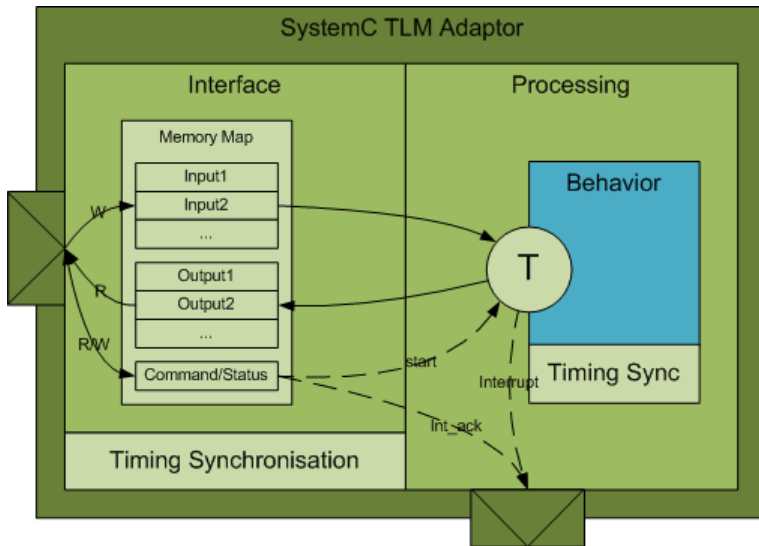
Registers and Signal Ports

Registers

The TLM component reads and writes inputs and outputs directly from the interface register during algorithm processing. After the initiator writes all input registers (if in AUTO mode) or when the initiator writes the START command in the CSR, the algorithm begins processing. A SystemC wait function generates all timings.

Caution To prevent corruption of the algorithm processing results, do not allow an initiator to perform a read or write of the registers during processing.

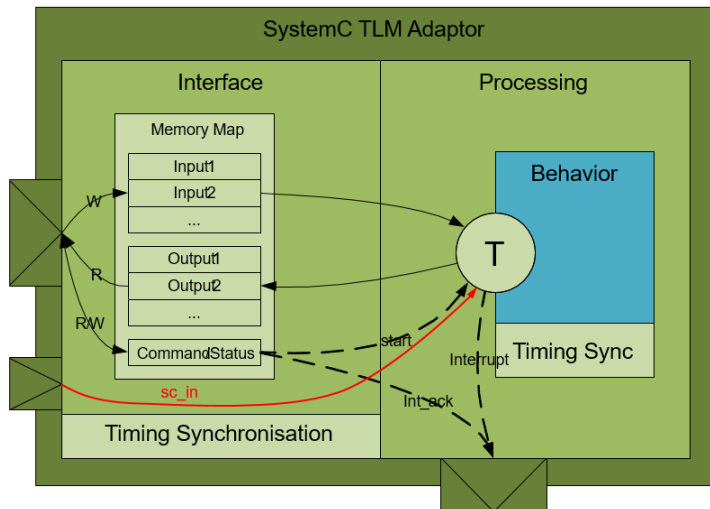
This diagram shows a TLM Adaptor with registered interfaces.



Signal Port

The TLM Component reads and writes inputs and outputs through a `sc_signal` port (`sc_in` or `sc_out`). These inputs/outputs are not registered. When the step function is executed, it reads the current value of the `sc_in` ports, executes and writes the result in the `sc_out` ports.

This diagram shows a TLM Adaptor with register interfaces, and an `sc_in` port (in red).



Getting Started with TLM Component Generation

Get Started with TLM Generator

This example shows how to configure a Simulink® model to generate a SystemC™/TLM component using the tlmgenerator target for either Simulink Coder™ or Embedded Coder®.

For this example, we use a Simulink model of a FIR filter as the basis of the SystemC/TLM generation.

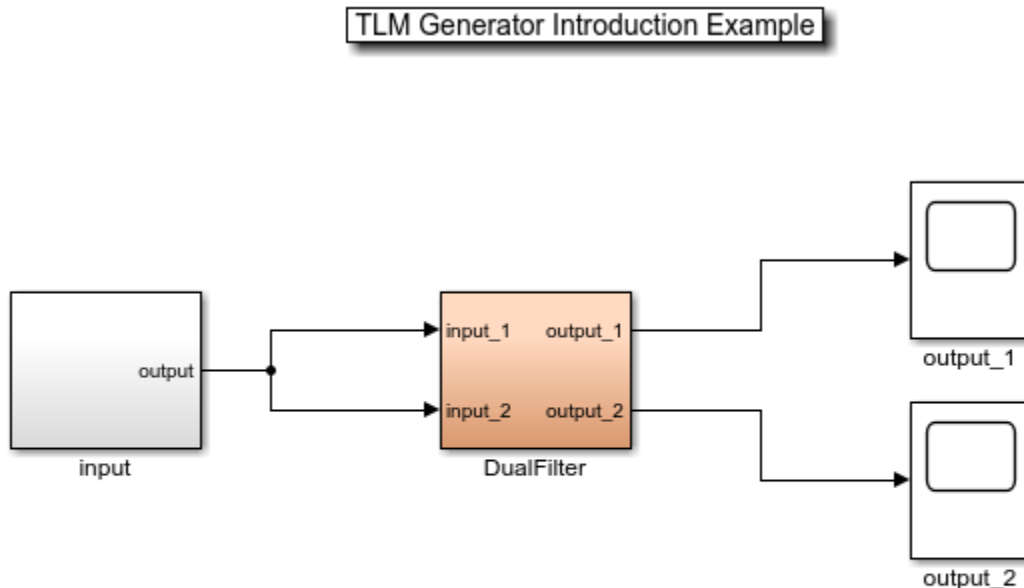
Requirements to run this example:

- SystemC 2.3.1 (includes the TLM library)
- For code verification, make and a compatible GNU-compiler, gcc, in your path on Linux®, or Visual Studio® compiler in your path on Windows®

Note: The example includes a code generation build procedure. Simulink does not permit you to build programs in the MATLAB® installation area. If necessary, change to a working directory that is not in the MATLAB installation area prior to starting any build.

1. Open Preconfigured Model

To open the **FIR Filter model**, click the **Open Model** button.



This model shows the TLM component generation with a dual 4 taps FIR filter.

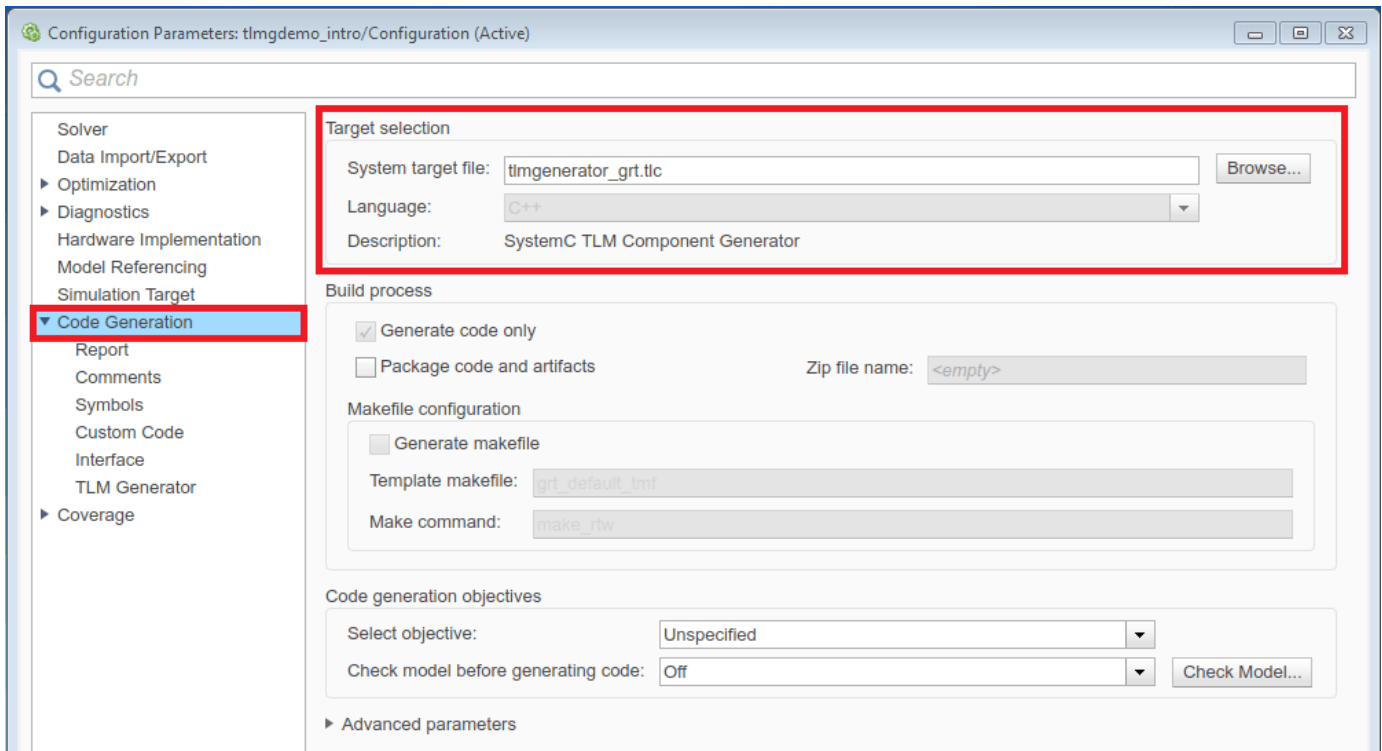
Copyright 1994-2020 The MathWorks, Inc.

2. Set Simulink Coder Target to TLM Generator

a. Open the **Configuration Parameters** dialog box by selecting Simulation > Model Configuration Parameters in the model window.

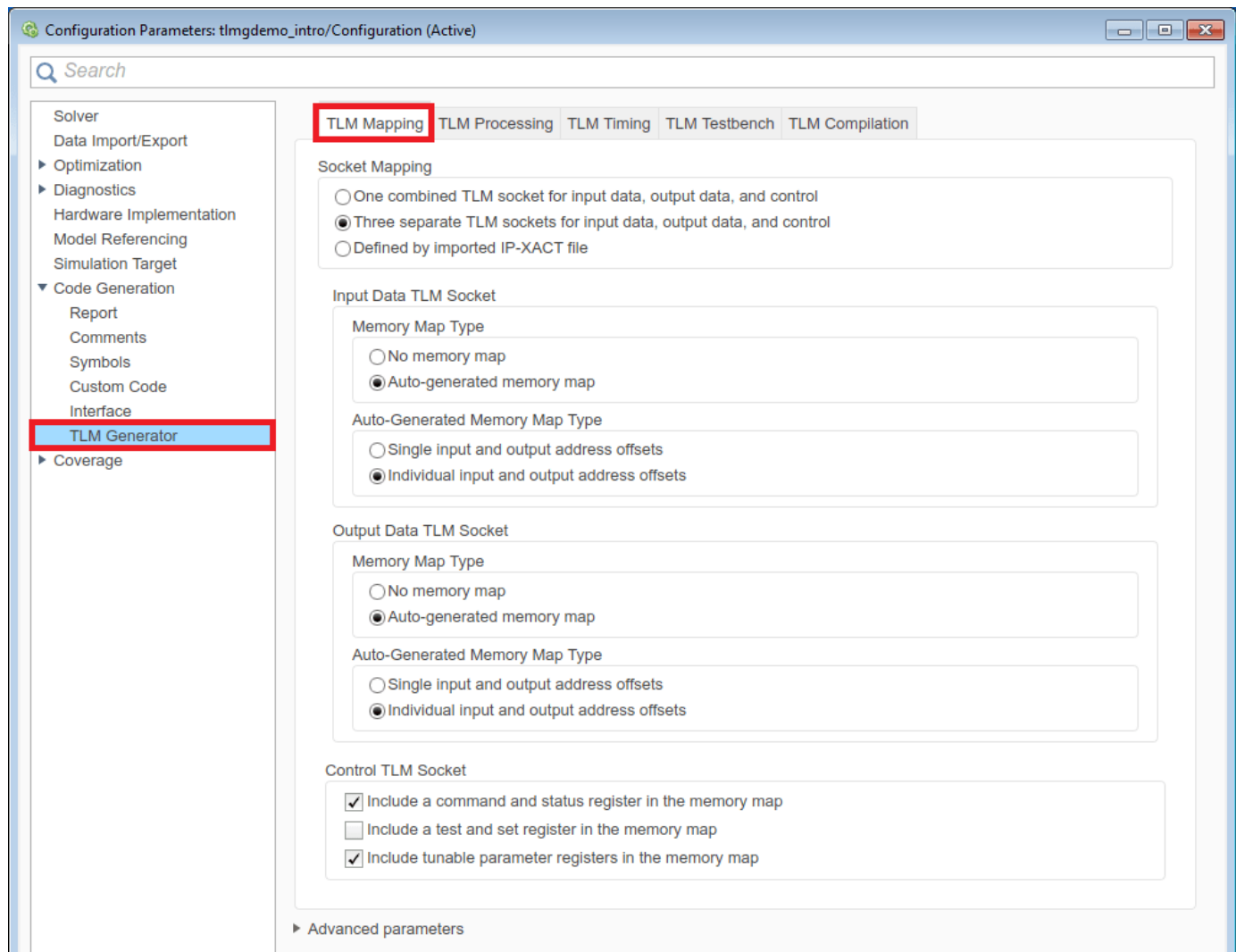
b. In the **Configuration Parameters** dialog box, select the **Code Generation** view in the left-hand pane.

c. Under **System target file**, click Browse to select the TLM generator target. You can choose `tlmgenerator_grt.tlc` to use Simulink Coder or `tlmgenerator_ert.tlc` to use Embedded Coder for HDL Code generation. For this example, select **`tlmgenerator_grt.tlc`**.



3. Open TLM Generator View

In the **Configuration Parameters** dialog box, select the **TLM Generator** view in the left-hand pane.



The **TLM Generator** view has five tabs:

- TLM Mapping
- TLM Processing
- TLM Timing
- TLM Testbench
- TLM Compilation

You will need to set different generator options in each pane.

4. Select TLM Mapping Options

In the **TLM Mapping** tab, **Socket Mapping** allows you to select the number of sockets for input data, output data, and control. Select the option, **Three separate TLM sockets for input data, output data, and control**.

Socket Mapping

- One combined TLM socket for input data, output data, and control
 Three separate TLM sockets for input data, output data, and control
 Defined by imported IP-XACT file

The TLM Socket options allow you to define three different memory maps for your generated TLM component.

Input Data TLM Socket

Memory Map Type

- No memory map
 Auto-generated memory map

Auto-Generated Memory Map Type

- Single input and output address offsets
 Individual input and output address offsets

Output Data TLM Socket

Memory Map Type

- No memory map
 Auto-generated memory map

Auto-Generated Memory Map Type

- Single input and output address offsets
 Individual input and output address offsets

Control TLM Socket

- Include a command and status register in the memory map
 Include a test and set register in the memory map
 Include tunable parameter registers in the memory map

For this example, select the following for the input and output data sockets:

- **Auto-generated memory map** for the input and output data sockets
- **Individual input and output address offsets** for each data socket. This option generates a TLM component with one read register per model output and one write register per model input with individual addresses. Each Simulink model input is bound to its corresponding write register and each output is bound to its corresponding read register.

The other memory map options you may consider are:

- **No memory map:** This option generates a TLM component with only one read and one write register without any address. The Simulink model inputs are bound to the write register and the outputs are bound to the read register.
- [Auto-generated memory map with] **Single input and output address offsets:** This option generates a TLM component with only one read and one write register with one address each. The Simulink model inputs are bound to the write register and the outputs are bound to the read register.

When you generate the component with a memory map, you can add a command and status register, a test and set register, and tunable parameter registers.

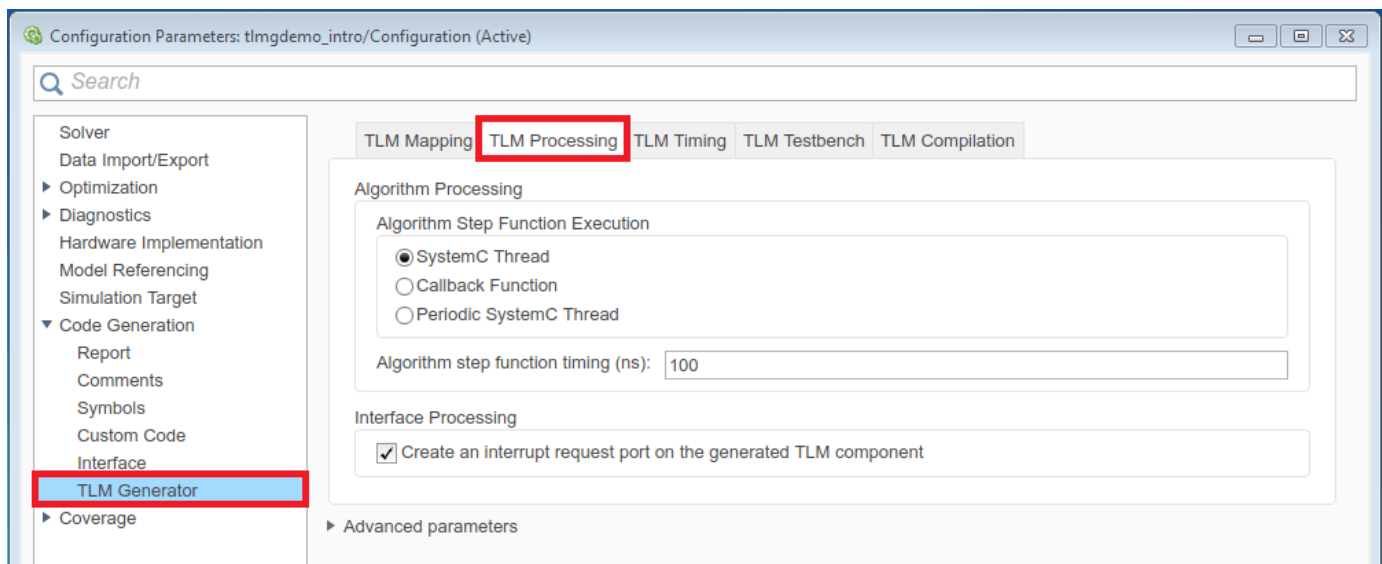
For this example, select the following for the Control TLM Socket:

- **Include a command and status register in the memory map.** The command and status register allows you to write command and read status during SystemC simulation; for example, manual buffering or buffer status.
- **Include tunable parameter registers in the memory map.** Tunable parameter registers allow you to read or write the tunable parameter values.

Although not used in this example, a test and set register can be used as a mutex when multiple initiators access the component during SystemC simulation.

5. Select TLM Processing Options

In the **TLM Generation** pane, select the **TLM Processing** tab. The **Algorithm Processing** and the **Interface processing** options allow you to define different buffering and processing behaviors for your generated TLM component.



The algorithm execution options are:

- **SystemC Thread:** The step function algorithm is executed in its own independent SystemC thread.
- **Callback Function:** The step function algorithm is executed in a callback function called from the interface.

- **Periodic SystemC Thread:** The step function algorithm is executed in its own time periodic independent SystemC thread.

The step function timing is determined by the value in the **Algorithm step function timing (ns)** field. The algorithm timing is counted with a wait() in the thread.

For this example, select **SystemC Thread** and enter 100 in the field for **Algorithm step function timing**.

Interface processing options:

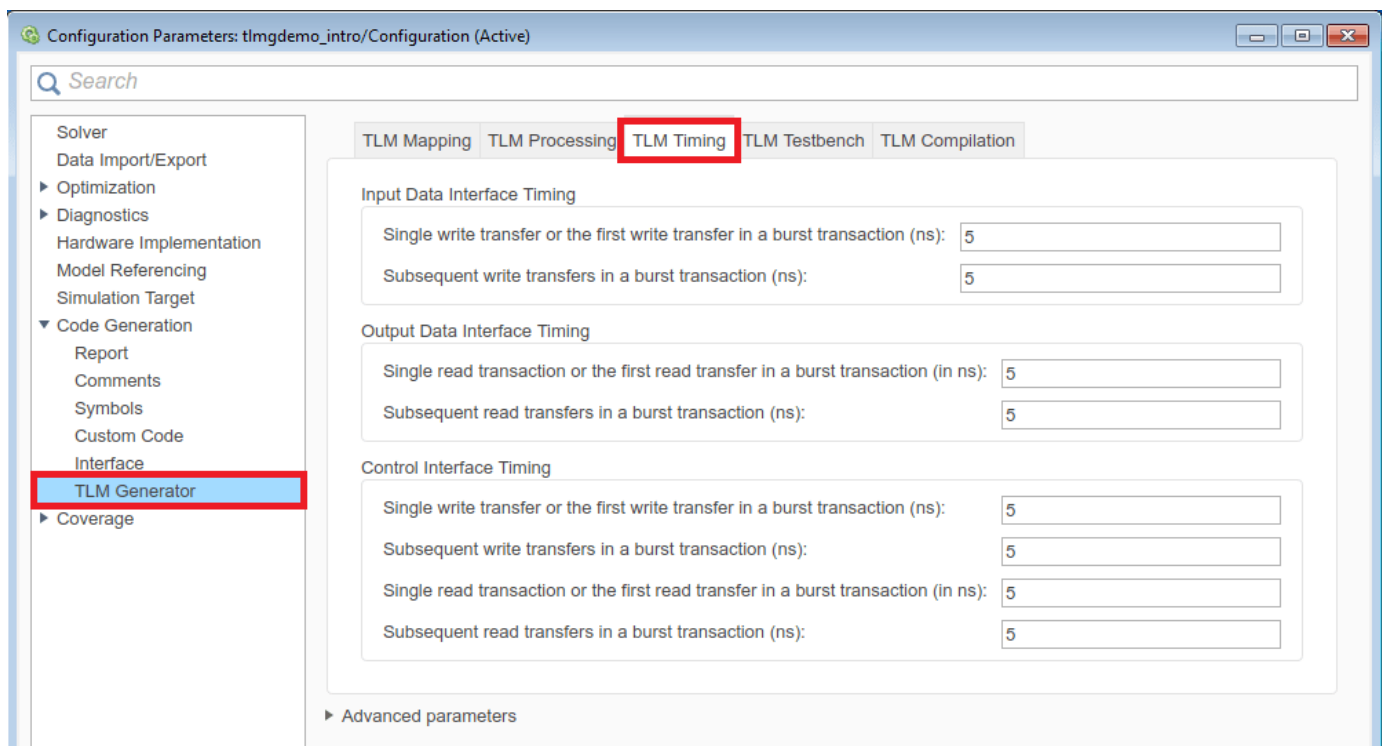
- **Create an interrupt request port on the generated TLM component:** This option creates an interrupt port (type signal, bool>) that is triggered every time a set of input has been processed.

For this example, select or enter the following choices:

- Select **Create an interrupt request port on the generated TLM component**.

6. Select the TLM Timing Options

Select the **TLM Timing** tab. The **Interface Timing** section allows you to define the timing of the component input/output interface and processing thread.

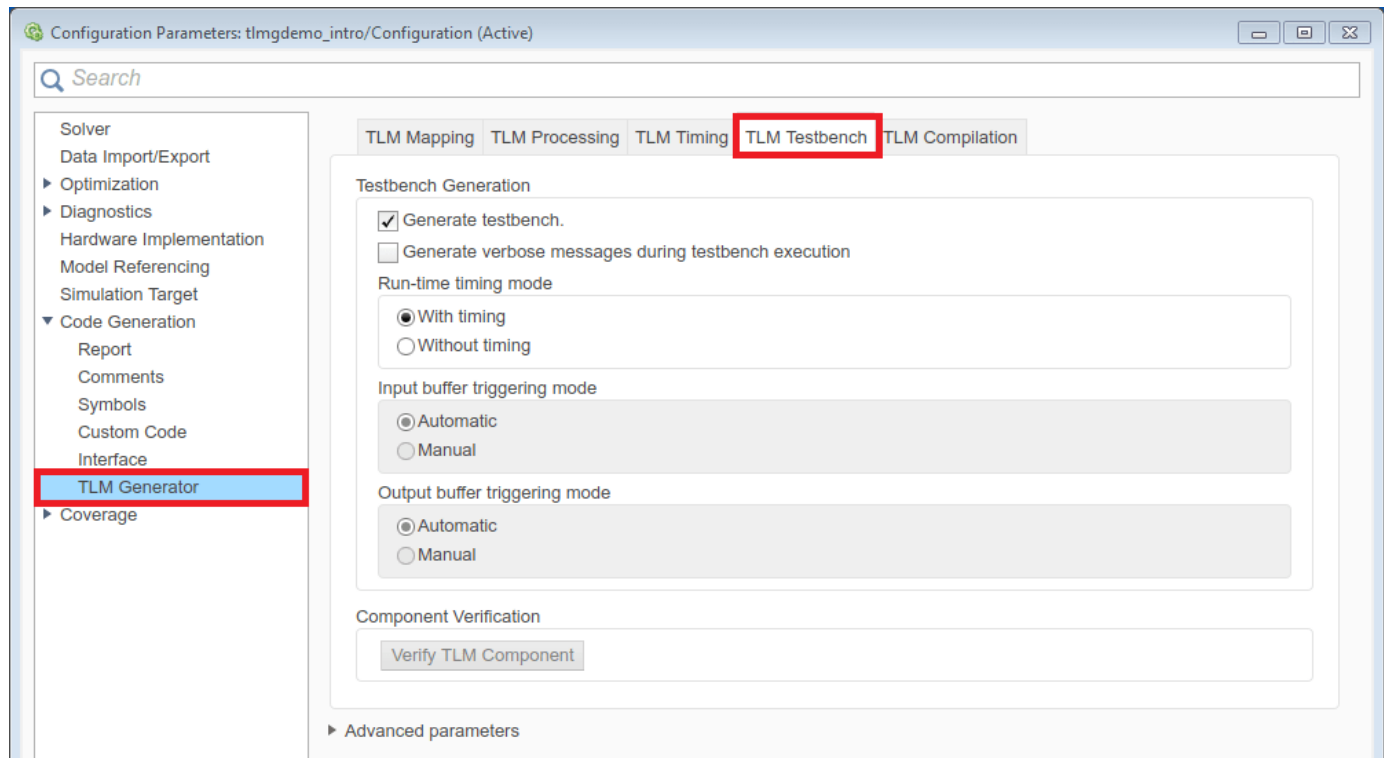


For this example, the input and output delays are counted with wait() in the interface. Set a time value of 5ns for each of the transactions of **Input Data Interface Timing**, **Output Data Interfacing Timing**, and **Control Interface Timing**.

7. Select TLM Testbench View

Select the **TLM Testbench** tab. The TLM Generator target can generate a stand-alone SystemC/TLM test bench alongside the TLM component to verify the generated algorithm in the context of a TLM

initiator/target pair. The TLM Testbench view provides run-time options for when the test bench code is generated and executed.



With the TLM Testbench options, you can:

- Choose to see verbose messages echoed to the command window during the SystemC/TLM execution including TLM transaction and synchronization messages.
- Indicate that the test bench should execute with or without timing annotations.
- Indicate whether the initiator controls moving input and output datasets between the registers and the buffers or whether the component performs the moves automatically.

For this example, select **Generate testbench**, **With timing** for **Run-time timing mode**, and **Automatic** for both **Input buffer triggering mode** and **Output buffer triggering mode**.

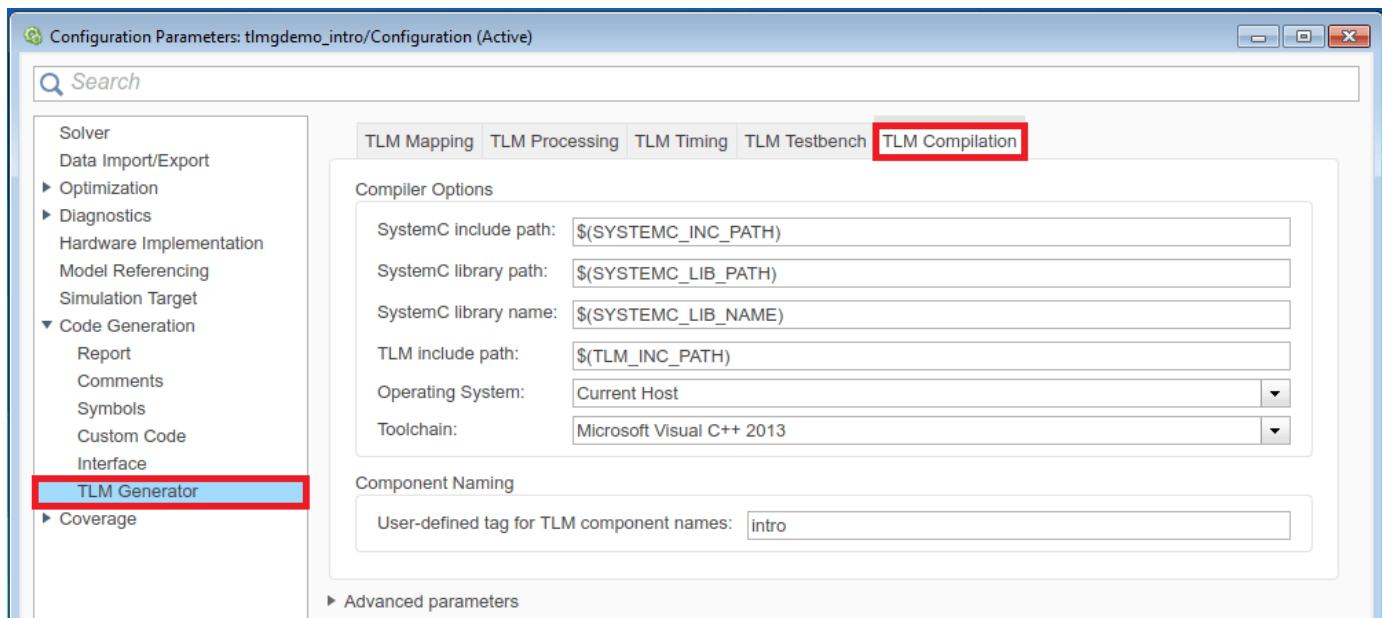
After code generation has successfully occurred for a component and test bench, the **Verify TLM Component** button becomes enabled. **Verify TLM Component** performs the following:

- Builds the generated code using make and generated makefiles.
- Runs Simulink to capture input stimulus and expected results.
- Converts the Simulink data to TLM vectors.
- Runs the stand-alone SystemC/TLM testbench executable.
- Converts the TLM results back to Simulink data.
- Performs a data comparison.
- Generates a Figure window for any signals that had data mis-compare.

The compilation of the generated files assumes the presence of make and a compatible GNU-compiler, gcc, in your path on Linux®, or Visual Studio® compiler in your path on Windows®.

8. Select TLM Compilation View

Select the **TLM Compilation** tab. This pane provides options to control the generation of makefiles used to compile the generated code.



Compiler Options:

The SystemC and TLM include library path options allow you to specify where the makefiles can find the SystemC and TLM installations. The default values allow you to use environment variables so that updates to your SystemC or TLM installations do not require updating your Simulink models. You can set up the environment prior to invoking MATLAB or use the MATLAB setenv command.

For this example, these are the environment variable values that were tested with standard OSCI installations located in /tools:

- SYSTEMC_INC_PATH=/tools/systemc-2.3.0/include
- SYSTEMC_LIB_PATH=/tools/systemc-2.3.0/lib-linux64
- SYSTEMC_LIB_NAME=libsystemc.a (Linux) or systemc.lib (Windows)
- TLM_INC_PATH=/tools/systemc-2.3.0/include

Toolchain:

On Windows this option allows you to select a compiler toolchain when multiple version of Microsoft Visual Studio are installed on the same machine. On Linux this option is fixed on gcc.

Component Naming:

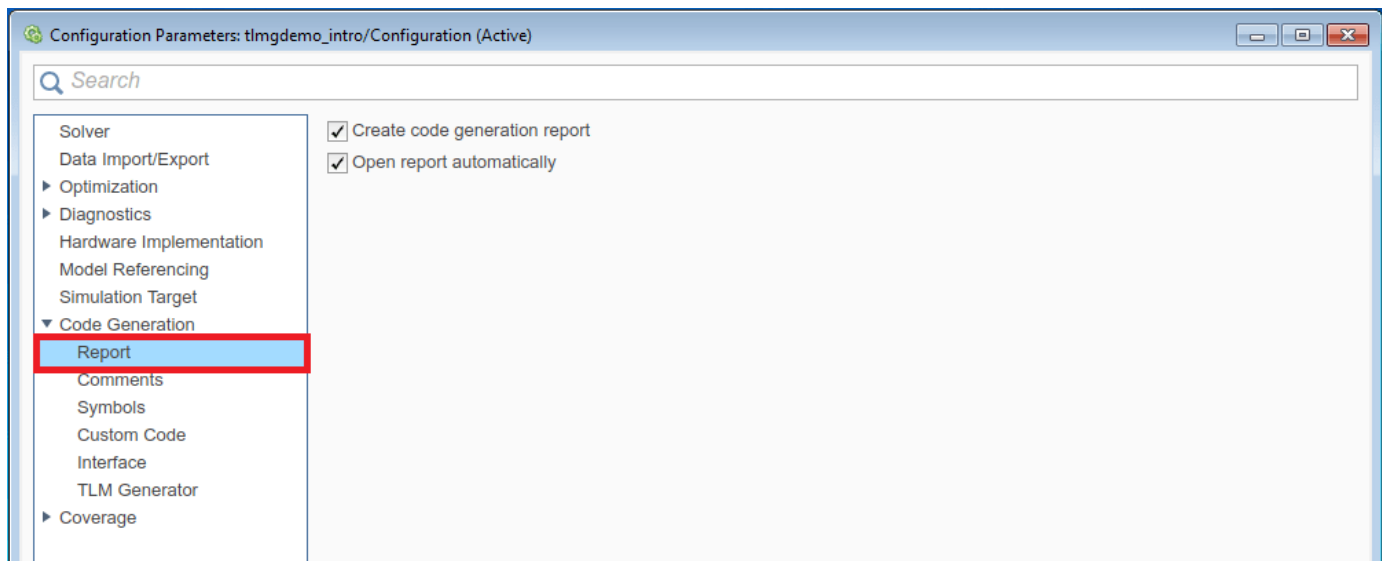
This option allows you to add your own tag to the name of the generated component. The generated component name is built according to the following cases:

- If a user tag is specified: `modelname_usertag_tlm`
- If the user tag field is empty: `modelname_tlm`

For this example, enter **intro** for user tag.

9. Select Report

Select **Report** in the left-hand pane. For this example, select **Create code generation report** and **Open report automatically**. These options generate a html report during component generation. The Code generation report details the contents of each generated file.



10. Save TLM Generator Options

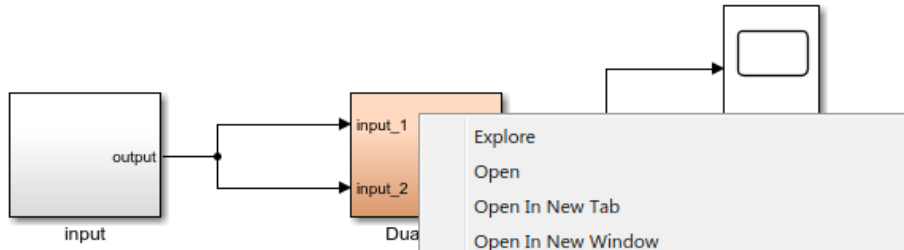
Click **OK** to apply these settings and exit the **Configuration Parameters** dialog box.

11. Build Model

In the model window, right-click on the DualFilter block and select **C/C++ Code > Generate Code for this Subsystem** in the context menu to start TLM component generation.



TLM Generator Introduction Example



This model shows the TLM component g

Copyright 1994-2

- Explore
- Open
- Open In New Tab
- Open In New Window
- Cut Ctrl+X
- Copy Ctrl+C
- Paste Ctrl+V
- Comment Through Ctrl+Shift+Y
- Comment Out Ctrl+Shift+X
- Delete Del
- Find Referenced Variables
- Subsystem & Model Reference ▶
- Test Harness ▶
- Format ▶
- Rotate & Flip ▶
- Arrange ▶
- Mask ▶
- Library Link ▶
- Signals & Ports ▶
- Requirements Traceability ▶
- Linear Analysis ▶
- Design Verifier ▶
- Coverage ▶
- Model Advisor ▶
- Fixed-Point Tool... ▶
- Model Transformer ▶
- C/C++ Code ▶
- HDL Code ▶
- PLC Code ▶
- Polyspace ▶
- Block Parameters (Subsystem)
- Properties...
- Help

- Embedded Coder Quick Start
- Code Generation Advisor
- Build This Subsystem**
- Export Functions
- Generate S-Function
- Navigate To C/C++ Code
- Open Subsystem Report

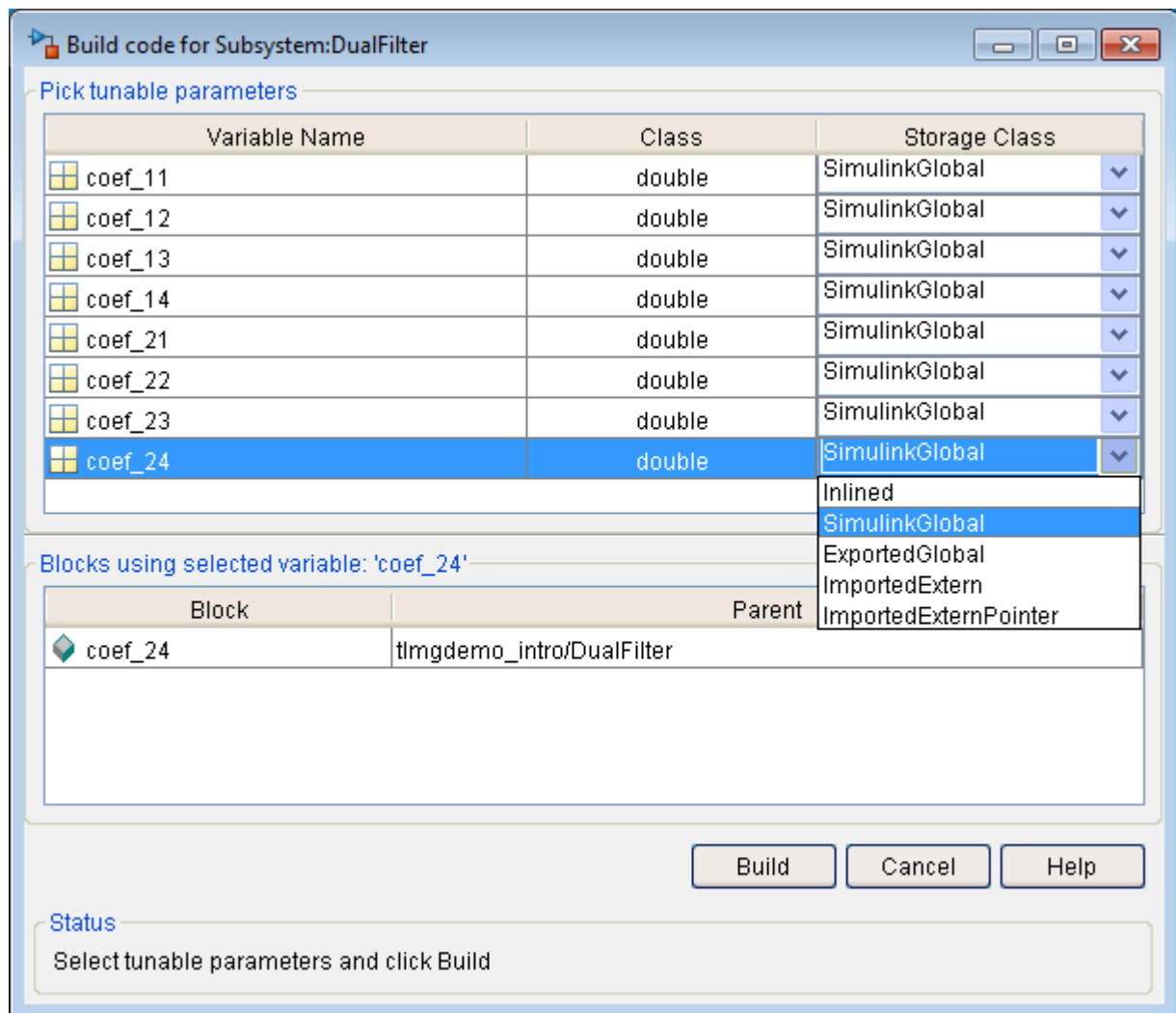
Alternatively, you can execute the following command in the MATLAB command window:

```
>> slbuild('tlmgdemo_intro/DualFilter');
```

During execution, you will be prompted to select the tunable parameters. The dropdown list of each coefficient allows you to select the storage class of the variable. The Storage Class options are:

- **Inlined** - The inlined parameters are not tunable.
- **SimulinkGlobal** - The SimulinkGlobal variables are tunable.

ExportedGlobal, **ImportedExtern** and **ImportedExternPointer** are not supported by the TLM Generation model.



The option **Simulink Global** has been selected for this example. Click **Build**. The TLM generation is completed when you see the following message appear in the MATLAB command window:

```
### Starting Simulink Coder build procedure for model: DualFilter
### Successful completion of Simulink Coder build procedure for model: DualFilter
```


Build Summary

Top model targets built:

Model	Action	Rebuild Reason
DualFilter	Code generated.	Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 20.733s

12. Open Generated Files

Open the generated files in the MATLAB web browser by clicking on the links in the generated report or in the MATLAB Editor (the generated files and report are located in your current working directory):

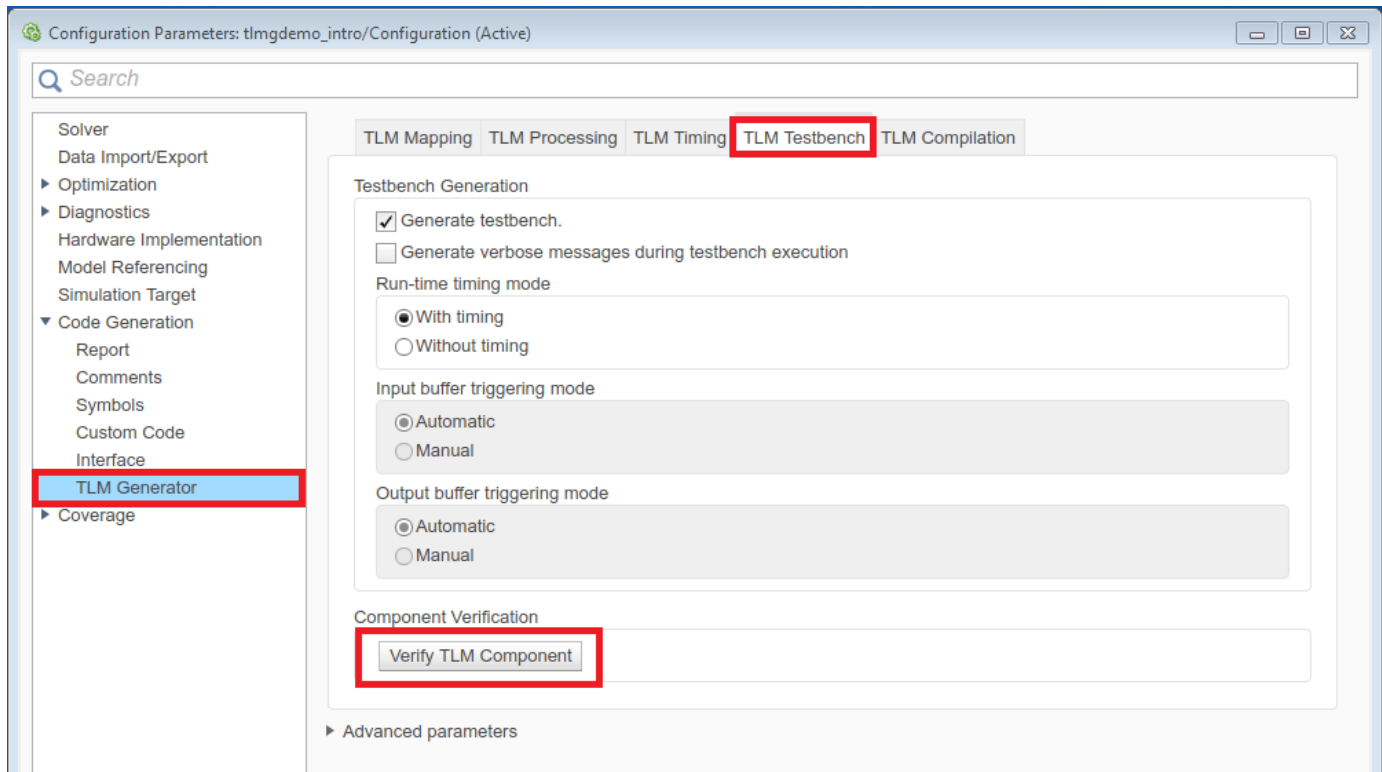
- DualFilter_VP/DualFilter_intro_tlm_doc/html/DualFilter_codegen_rpt.html
- DualFilter_VP/DualFilter_intro_tlm/DualFilter_intro_tlm.xml
- DualFilter_VP/DualFilter_intro_tlm/include/DualFilter_intro_tlm_def.h
- DualFilter_VP/DualFilter_intro_tlm/include/DualFilter_intro_tlm.h
- DualFilter_VP/DualFilter_intro_tlm/src/DualFilter_intro_tlm.cpp
- DualFilter_VP/DualFilter_intro_tlm_tb/src/DualFilter_intro_tlm_tb.h
- DualFilter_VP/DualFilter_intro_tlm_tb/src/DualFilter_intro_tlm_tb.cpp
- DualFilter_VP/DualFilter_intro_tlm_tb/src/DualFilter_intro_tlm_tb_main.cpp

13. Verify Generated Code

a. Open the **Model Configuration Parameters** dialog box by selecting Simulation > Configuration Parameters in the model window.

b. In the **Configuration Parameters** dialog box, select the **TLM Generator** view, and then select the **TLM Testbench** tab.

c. In the **TLM Testbench** pane, click **Verify TLM Component**, to run the generated testbench.



Alternatively, you can execute the following command in the MATLAB command window:

```
verifyTlmgDemoModel('intro')
```

This verify step performs the following actions:

- Builds the generated code.
- Runs Simulink to capture input stimulus and expected results.
- Converts the Simulink data to TLM vectors.
- Runs the stand-alone SystemC/TLM test bench executable.
- Converts the TLM results back to Simulink data.
- Performs a data comparison.
- Generates a Figure window for any signals that had data mis-compares.

14. Review Execution Log

The option to generate test bench allows you to see how the test bench initiator threads interact and synchronize with the target. Look for the comparison result at the end of the log and verify that the data comparison is successful.

```
### Comparing expected vs. actual results.
Data successfully compared for signal tlmg_out1.
Data successfully compared for signal tlmg_out2.
### Component verification completed
```

This concludes the Getting Started with TLM Generator example.

Generate TLM Component

- “TLM Component Generation Workflow” on page 22-2
- “Subsystem Guidelines and Limitations” on page 22-3
- “Select TLM Generator System Target” on page 22-4
- “Select TLM Mapping Options” on page 22-6
- “Select TLM Processing Options” on page 22-8
- “Select TLM Timing Options” on page 22-9
- “Select TLM Test Bench Options” on page 22-10
- “Select TLM Compilation Options” on page 22-12
- “Generate Component and Test Bench” on page 22-15
- “Prepare IP-XACT File for Import” on page 22-16
- “Contents of Generated IP-XACT File” on page 22-24
- “Implement Memory Map with SCML” on page 22-28

TLM Component Generation Workflow

The following workflow lists the steps required to generate a TLM component using HDL Verifier software:

- 1** Develop algorithm in Simulink. See “Subsystem Guidelines and Limitations” on page 22-3.
- 2** “Select TLM Generator System Target” on page 22-4
- 3** “Select TLM Mapping Options” on page 22-6
- 4** “Select TLM Processing Options” on page 22-8
- 5** “Select TLM Timing Options” on page 22-9
- 6** “Select TLM Test Bench Options” on page 22-10
- 7** “Select TLM Compilation Options” on page 22-12
- 8** “Generate Component and Test Bench” on page 22-15
- 9** (Optional) “Run TLM Component Test Bench” on page 23-5 (verify TLM component)
- 10** “Export TLM Component” on page 24-2

Subsystem Guidelines and Limitations

Most subsystems that can be converted to C code are suitable for generating a TLM component. When you are considering a subsystem for TLM generation, keep in mind the following limitations:

- Simulink subsystem limitations for TLM generation:
 - Same limitations as the Embedded Coder target if you are using Embedded Coder. If you are using Simulink Coder license, then Simulink Coder limitations are the ones that apply.
 - Bus data type not supported
 - Variable-size signals not supported
- Simulink subsystem limitations for TLM test bench generation:
 - Composite Simulink signal types not supported (e.g., buses, no-contiguous memory mux block outputs)
 - Multirate subsystems are not supported (however, constants are supported)
 - Complex signals are not supported
 - Subsystems with “action” ports are not supported (e.g., triggered, enabled, if Action, switch case Action)
- SystemC/TLM generated component limitations:
 - TLM simple target socket (with blocking and debug interfaces) using Generic Payload
 - TLM target only (no TLM initiator generation)
 - 32-bit bus width only (address align on 4 bytes)
 - No byte enable
 - No endianness option
 - No streaming
 - No DMI
 - Generic Payload extensions ignored

See Also

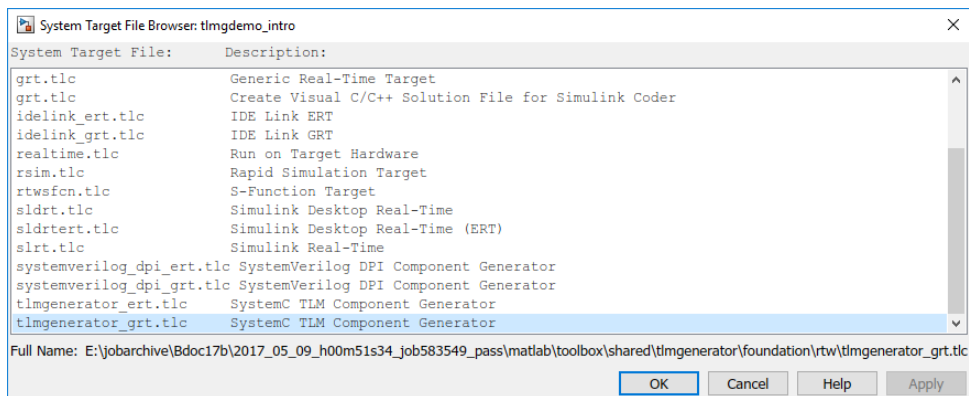
More About

- “TLM Component Generation Workflow” on page 22-2

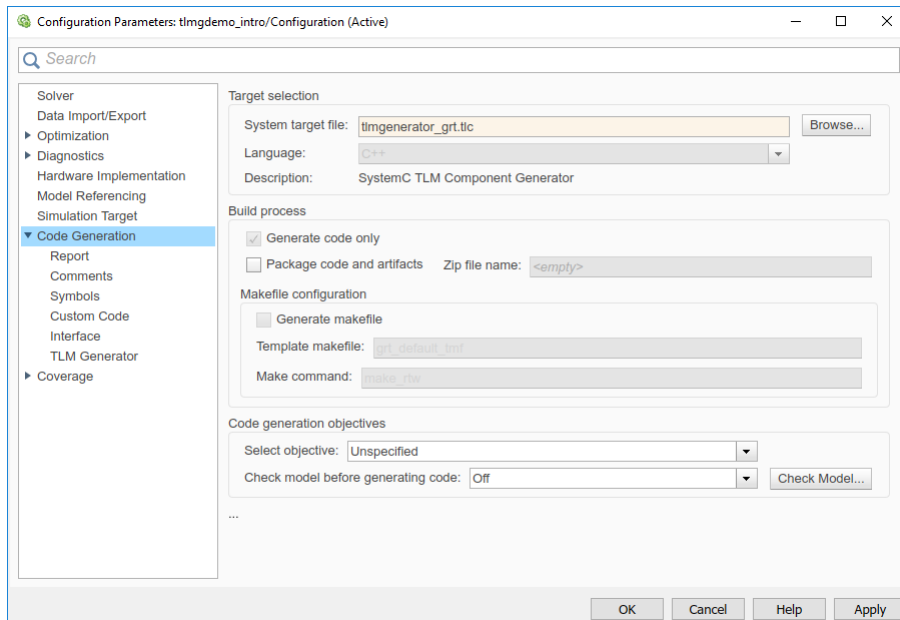
Select TLM Generator System Target

To activate the TLM component generation options, select the system target file.

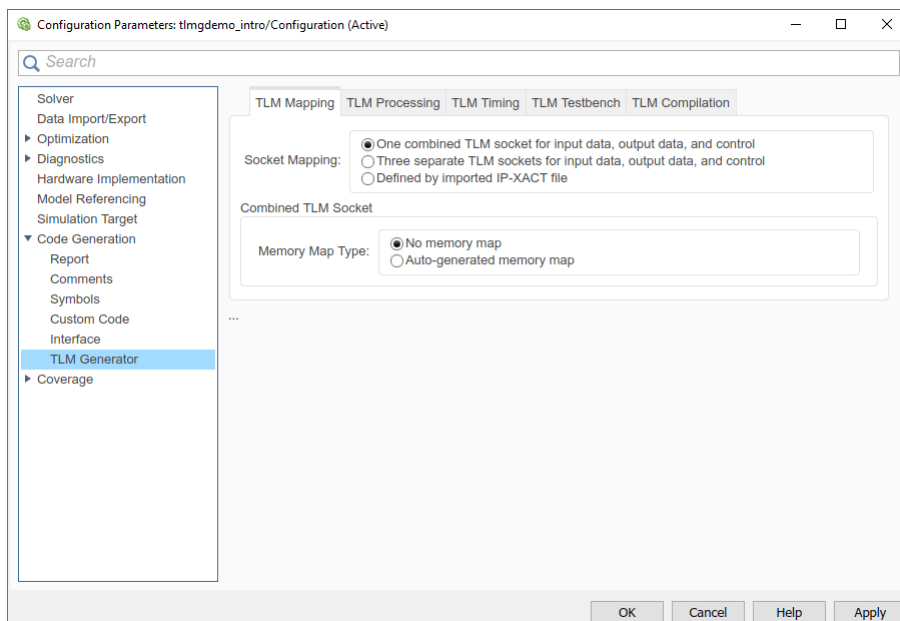
- 1 Select subsystem. See “Subsystem Guidelines and Limitations” on page 22-3 for help on selecting a suitable subsystem.
- 2 On the **Modeling** tab, in the **Setup** section, click **Model Settings**. The **Configuration Parameters** dialog opens.
- 3 Select **Code Generation** on the left pane.
- 4 Click **Browse** on **System Target File**. Then, follow these guidelines for selecting the correct system target file:
 - With Simulink Coder license, select: `tlmgenerator_grt.tlc`
 - With Embedded Coder license (Simulink Coder license is also required), select: `tlmgenerator_ert.tlc` or `tlmgenerator_grt.tlc`. Target `tlmgenerator_ert.tlc` allows you to access its additional code generation options using the Model Configuration Parameters dialog box.



- 5 Click **OK** to see the new **TLM Generator** option under Code Generation.



6 Click **TLM Generator** to display the TLM Generation options panes.



See Also

More About

- “TLM Component Generation Workflow” on page 22-2

Select TLM Mapping Options

In this section...

“Socket Mapping” on page 22-6

“Memory Map Configuration” on page 22-7

Select socket and memory map configuration for your TLM component on the **TLM Mapping** tab. You can select a single socket, or three sockets, generated from your Simulink model, or choose to import a custom socket map using an IP-XACT file. If you select one of the fixed socket configurations, you can specify additional memory map options.

Socket Mapping

You can choose to have a single, combined TLM socket for input data, output data, and control or you can choose three separate TLM socket for input data, output data, and control so that you can connect the sockets to different buses. Alternatively, you can customize the socket mapping using an IP-XACT file.

- **One combined TLM socket for input data, output data, and control** — Selecting this option displays the **Combined TLM Socket** parameters to configure the generated memory map. See “Memory Map Configuration” on page 22-7.
- **Three separate TLM sockets for input data, output data, and control** — Selecting this option displays separate memory map parameters for the three sockets. See “Memory Map Configuration” on page 22-7.
- **Defined by imported IP-XACT file** — When prompted, provide the path and file name of the IP-XACT file.

Import IP-Xact File

IP-XACT file:

Generate code for unmapped IP-XACT registers/bitfields

Generate code for unmapped IP-XACT signal ports

Implement memory map with SCML

See “Prepare IP-XACT File for Import” on page 22-16 for IP-XACT file requirements.

By default, only those registers and signal ports mapped to Simulink signals are implemented in the generated TLM component. Select **Generate code for unmapped IP_XACT registers/bitfields** to include all registers from the IP-XACT file in the generated TLM component. Select **Generate code for unmapped IP_XACT signal ports** to include all signal ports from the IP-XACT file in the generated TLM component.

When you import an IP-XACT file, you can optionally generate an interface compatible with the System C Modeling Library (SCML). Select **Implement memory map with SCML**. See “Implement Memory Map with SCML” on page 22-28. To use this feature, you must install SCML from Synopsys®.

Memory Map Configuration

For each socket, choose the socket mapping type. For a description of the options available under **Memory Map Type**, see “Memory Mapping” on page 20-3.

If you choose **Auto-generated memory map**, the options expand to include the **Auto-Generated Memory Map Type** section, as shown in the following figures:

Single TLM Socket	Separate TLM Sockets
<p>Socket Mapping</p> <p><input checked="" type="radio"/> One combined TLM socket for input data, output data, and control</p> <p><input type="radio"/> Three separate TLM sockets for input data, output data, and control</p> <p><input type="radio"/> Defined by imported IP-Xact file</p>	<p>Socket Mapping</p> <p><input type="radio"/> One combined TLM socket for input data, output data, and control</p> <p><input checked="" type="radio"/> Three separate TLM sockets for input data, output data, and control</p> <p><input type="radio"/> Defined by imported IP-Xact file</p>
<p>Combined TLM Socket</p> <p>Memory Map Type</p> <p><input type="radio"/> No memory map</p> <p><input checked="" type="radio"/> Auto-generated memory map</p>	<p>Input Data TLM Socket</p> <p>Memory Map Type</p> <p><input type="radio"/> No memory map</p> <p><input checked="" type="radio"/> Auto-generated memory map</p>
<p>Auto-Generated Memory Map Type</p> <p><input checked="" type="radio"/> Single input and output address offsets</p> <p><input type="radio"/> Individual input and output address offsets</p>	<p>Auto-Generated Memory Map Type</p> <p><input checked="" type="radio"/> Single input and output address offsets</p> <p><input type="radio"/> Individual input and output address offsets</p>
<p><input type="checkbox"/> Include a command and status register in the memory map</p> <p><input type="checkbox"/> Include a test and set register in the memory map</p> <p><input type="checkbox"/> Include tunable parameter registers in the memory map</p>	<p>Control TLM Socket</p> <p><input type="checkbox"/> Include a command and status register in the memory map</p> <p><input type="checkbox"/> Include a test and set register in the memory map</p> <p><input type="checkbox"/> Include tunable parameter registers in the memory map</p>
	<p>Output Data TLM Socket</p> <p>Memory Map Type</p> <p><input type="radio"/> No memory map</p> <p><input checked="" type="radio"/> Auto-generated memory map</p>
	<p>Auto-Generated Memory Map Type</p> <p><input checked="" type="radio"/> Single input and output address offsets</p> <p><input type="radio"/> Individual input and output address offsets</p>

- Select the autogenerated memory map type for each TLM socket:
 - **Single input and output address offsets** — See “Automatically Generated Memory Map with Single Address” on page 20-4.
 - **Individual input and output address offsets** — See “Automatically Generated Memory Map with Individual Addresses” on page 20-6.
- For the control socket (either separate or combined), select any of the following options:
 - **Include a command and status register in the memory map** — See “Registers and Signal Ports” on page 20-11.
 - **Include a test and set register in the memory map** — See “Test and Set Register” on page 20-11.
 - **Include tunable parameter registers in the memory map** —The tunable parameter registers allow you to adjust the TLM component before or during simulation.

See Also

More About

- “TLM Component Generation Workflow” on page 22-2

Select TLM Processing Options

To execute your TLM model, choose between a SystemC thread, a callback function or a time-periodic thread in your generated TLM component.

Algorithm Processing

Algorithm Step Function Execution:

SystemC Thread
 Callback Function
 Periodic SystemC Thread

Algorithm step function timing (ns):

Interface Processing

Create an interrupt request port on the generated TLM component

Algorithm Processing

- **SystemC Thread** — The algorithm executes in its own independent SystemC thread. When the input buffers are full or when you write a command in the command and status register, an event is triggered. The system scheduler then picks up and executes that function. This option typically results in more realistic simulations but slower execution times.
- **Callback Function** — The algorithm executes in a callback function called from the interface. When the input buffers are full or when you write a specific command in the command and status register, the function is called directly. This option results in faster execution but could be less realistic because the callback method does not process events in the same order as they would occur in a real world scenario.
- **Periodic SystemC Thread** — A time-periodic thread executes the behavior of the algorithm. The period of the thread is derived from the Simulink block base sample rate.

In the **Algorithm step function timing (ns)** parameter, enter the time in nanoseconds. a wait() task counts the algorithm timing.

Interface Processing

Select **Create an interrupt request port on the generated TLM component** to create an interrupt request port. This interrupt triggers every time a set of inputs is processed.

See Also

More About

- “TLM Component Generation Workflow” on page 22-2

Select TLM Timing Options

Specify timing parameters to approximate the actual time consumed by operations in a real system. These timing values are stored in the TLM component and supplied to the SystemC environment when the TLM component is used. Your system simulation environment must perform accounting of execution times in the system, as described in the *OSCI TLM-2.0 Language Reference Manual*. These values add temporal realism to your system simulations.

For all timing options, specify the desired time in nanoseconds. Each socket has independent timing parameters. The specified timing values are implemented as constants in the generated code. This delay is implemented as a `wait()` function.

At runtime, you can dynamically control the TLM component via a `backdoor` interface to enable and disable the return of timing information. See the generated test bench code for details (locate `mw_backdoorcfg_IF`).

Combined Interface Timing	
Single write transfer or the first write transfer in a burst transaction (ns):	<input type="text" value="10"/>
Subsequent write transfers in a burst transaction (ns):	<input type="text" value="10"/>
Single read transaction or the first read transfer in a burst transaction (in ns):	<input type="text" value="10"/>
Subsequent read transfers in a burst transaction (ns):	<input type="text" value="10"/>

See Also

More About

- “TLM Component Generation Workflow” on page 22-2

Select TLM Test Bench Options

These options control generation of an automatic test bench that compares your generated TLM component with your Simulink model. This test bench is not supported if you generate a TLM component for an operating system different than your MATLAB host machine.

The screenshot shows a dialog box for configuring testbench generation. It includes the following options:

- Generate testbench.
- Generate verbose messages during testbench execution
- Run-time timing mode:
 - With timing
 - Without timing
- Input buffer triggering mode:
 - Automatic
 - Manual
- Output buffer triggering mode:
 - Automatic
 - Manual
- Component Verification:
 - Verify TLM Component

Use the test bench options to specify these options:

- **Generate testbench** — Select to generate a test bench for the generated TLM component.
- **Generate verbose messages during testbench execution** — The default is not to generate these messages.
- **Run-time timing mode** — Specify whether the test bench executes with or without timing annotations. When you select **With timing**, the target annotates TLM component transactions with delays, and the initiator module honors them. The initiator module synchronizes immediately following the transaction execution.

When you select **Without timing**, the target does not annotate TLM component transaction with delays. The initiator module and target only perform synchronization using zero-time wait calls.

- **Buffer triggering modes** — Specify whether the initiator controls moving datasets between the registers and the buffers or if the component moves the datasets automatically. In your TLM environment, these specifications are performed via a runtime configuration command. You can change them dynamically throughout simulation.

The default is **Automatic** mode. If you instead choose **Manual** mode, the initiator module must explicitly write a command to the command and status register to move the input data set from the register to the input buffer, or move the output data set from the output buffer to the output register.

Manual mode enables an initiator module to reuse a complete or partial input data set for a subsequent execution of the algorithm, thereby saving simulation time by avoiding data TLM component transactions don't need. For example, if the target uses a full memory map and the initiator module detects that only one of the values is changing, the initiator module may execute TLM component transactions only for the changing value. The initiator module then writes a push command to execute the algorithm.

Note For this field to be enabled, select **Include a command and status register in the memory map** in the **TLM Generation** tab.

- **Component Verification**

After code generation is successfully completed, you can use **Verify TLM Component** to perform the following actions:

- Build the generated code using make and generated makefiles.
- Run Simulink to capture input stimulus and expected results.
- Convert the Simulink data to TLM vectors.
- Run the standalone SystemC/TLM test bench executable.
- Convert the TLM results back to Simulink data.
- Perform a data comparison.
- Generate a Figure window for any signals that had data mismatches.

See Also

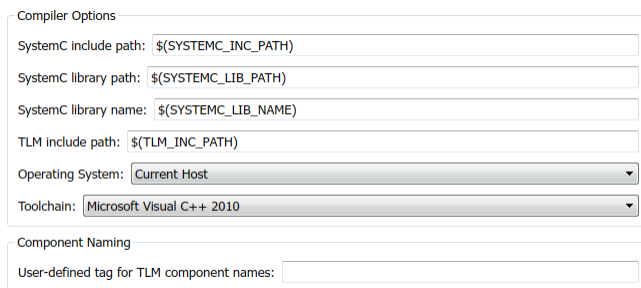
More About

- “TLM Component Generation Workflow” on page 22-2

Select TLM Compilation Options

Along with the generated component, the TLM generator also generates a makefile for building the shared libraries. Use the options on the **TLM Compilation** tab to specify makefile attributes before you generate code. You can generate a TLM component to run on a different operating system than that of your MATLAB machine. Specify the compiler parameters for the target machine where you will run the makefile.

The default values are environment variables (for example, `$SYSTEMC_INC_PATH`). If you use the default variable name and define these environment variables in your system, you can usually update your installation without having to update your Simulink models.



Compiler Options

SystemC include path:

SystemC library path:

SystemC library name:

TLM include path:

Operating System:

Toolchain:

Component Naming

User-defined tag for TLM component names:

- **SystemC include path** — Specify the location of the include folder in your SystemC installation. For example:

```
/systemc-2.2.0/include
```

Alternately, use the default environment variable and define `$SYSTEMC_INC_PATH` in your system.

- **SystemC library path** — Specify the location of the library folder in your SystemC installation. For example:

```
/systemc-2.2.0/lib
```

Alternately, use the default environment variable and define `$SYSTEMC_LIB_PATH` in your system.

- **SystemC library name** — Specify the name of the SystemC library in your SystemC installation. For example:

- Windows: `systemc.lib`
- Linux: `libsystemc.a`

Alternately, use the default environment variable and define `$SYSTEMC_LIB_NAME` in your system.

- **TLM Include Path** — Specify the location of the include folder in your TLM installation. For example:

```
/tlm-2.0.1/include
```

Alternately, use the default environment variable and define `$TLM_INC_PATH` in your system. Since SystemC 2.2, the TLM library is included with SystemC. Therefore, this path might be the same as `$SYSTEMC_INC_PATH`.

- **Operating System** — You can generate a TLM component for an operating system different from that of your MATLAB host machine. Select **Windows 64** or **Linux 64**. The **Toolchain** options change depending on your target operating system.
- **Toolchain** — Specify a compiler from the **Toolchain** drop-down list. The available options are the compiler versions installed on your computer. The default option is the version most recently installed. See “TLM Generation Requirements” for a list of supported compilers.

If you choose **Implement memory map with SCML** on the **TLM Mapping** tab, specify the location of your SCML installation using these additional options.

Compiler Options

SystemC include path:

SystemC library path:

SystemC library name:

TLM include path:

SCML include path:

SCML library path:

SCML library name:

SCML logging library name:

Operating System:

Toolchain:

Component Naming

User-defined tag for TLM component names:

- **SCML include path** — Specify the location of the include folder in your SCML installation. For example:

```
/scml-2.2/include
```

Alternately, use the default environment variable and define `$SCML_INC_PATH` in your system.

- **SCML library path** — Specify the location of the library folder in your SCML installation. For example:
 - Windows: `/scml-2.2/lib/win64`
 - Linux: `/scml-2.2/lib/glnxa64`

Alternately, use the default environment variable and define `$SCML_LIB_PATH` in your system.

- **SCML library name** — Specify the name of the SCML library in your SCML installation. For example:

```
scml2-vs-11.0.lib
```

Alternately, use the default environment variable and define `$SCML_LIB_NAME` in your system.

- **SCML logging library name** — Specify the name of the SCML logging library in your SCML installation. For example:

```
scml2_logging-vs-11.0.lib
```

Alternately, use the default environment variable and define `$SCML_LOGGING_LIB_NAME` in your system.

Component Naming

- **User-defined tag for TLM component names** — Add additional text to your TLM component class name identifier. To see how the user tag is applied, see “Identify Generated Files” on page 24-2.

See Also

More About

- “TLM Component Generation Workflow” on page 22-2

Generate Component and Test Bench

- 1 If you have not yet done so, finish selecting the TLM generation options and click **OK** to save your changes and exit the TLM Generation options panes.
- 2 Generate code. Select one of the following ways:
 - Press **Ctrl-B** (full model).
 - Right-click the subsystem and select **C/C++ Code > Build This Subsystem**.
 - In the Simulink toolstrip, select the **C Code** tab, and click **Generate Code** (this option builds the full model).

Note Generate the component and test bench on the architecture you plan to use for running the SystemC simulation.

- 3 Go to “Run TLM Component Test Bench” on page 23-5 (optional).

For more about using the generated TLM component, see “Export TLM Component” on page 24-2.

Prepare IP-XACT File for Import

In this section...

“Required Information for Imported IP-XACT Files” on page 22-16

“Bus Interface Definition with No Memory Map” on page 22-17

“Bus Interface Definition with Memory Mapping” on page 22-18

“Mapping to a Signal Port” on page 22-21

To customize the TLM interface of the component you want to generate, you can import your own IP-XACT XML file into the TLM generator.

For more information about importing the IP-XACT file, see “Select TLM Mapping Options” on page 22-6.

Required Information for Imported IP-XACT Files

All IP-XACT XML files must contain information specific to MathWorks, defined in elements within the component. If this information is not present, the TLM generator cannot parse the IP-XACT file.

The following parameter name-value pairs are required for `<spirit:component>`:

- `<spirit:parameter>`

```

      <spirit:name>MWVendor</spirit:name>
      <spirit:value>MathWorks</spirit:value>
    </spirit:parameter>

```
- `<spirit:parameter>`

```

      <spirit:name>MWVersion</spirit:name>
      <spirit:value>1.0</spirit:value>
    </spirit:parameter>

```
- `<spirit:parameter>`

```

      <spirit:name>MWModel</spirit:name>
      <spirit:value>name_of_model</spirit:value>
    </spirit:parameter>

```
- `<spirit:parameter>`

```

      <spirit:name>MWBlock</spirit:name>
      <spirit:value>name_of_block</spirit:value>
    </spirit:parameter>

```

This image shows these required elements within an IP-XACT XML file.

```

<spirit:parameters>
  <spirit:parameter>
    <spirit:name>MWVendor</spirit:name>
    <spirit:value>MathWorks</spirit:value>
  </spirit:parameter>
  <spirit:parameter>
    <spirit:name>MWVersion</spirit:name>
    <spirit:value>1.0</spirit:value>
  </spirit:parameter>
  <spirit:parameter>
    <spirit:name>MWModel</spirit:name>
    <spirit:value>tlmgdemo_ipxactnomem</spirit:value>
  </spirit:parameter>
  <spirit:parameter>
    <spirit:name>MWBlock</spirit:name>
    <spirit:value>DualFilter</spirit:value>
  </spirit:parameter>
</spirit:parameters>

```

Bus Interface Definition with No Memory Map

- “General Guidelines” on page 22-17
- “Simulink Mapping with No Memory Map” on page 22-17

General Guidelines

Write the bus definitions for your model according to the IEEE Standard for IP-XACT 1685-2009.

If you want to use the Simulink mapping, all bus interfaces that contain Simulink mapping must be slave interfaces.

Each bus interface with no memory map must have one of the following element arrangements for Simulink mapping:

- No mapping to Simulink
- Mapping to Simulink inputs, Simulink outputs, or a mix of inputs and outputs
- Mapping to Simulink tunable parameters

Although each bus interface can have only one arrangement, the IP-XACT file can contain multiple bus interface definitions, each having a different arrangement.

Simulink Mapping with No Memory Map

Each `<spirit:busInterface>` definition containing a Simulink mapping is mapped to the TLM target socket. Within the `<spirit:parameters>` tag, add `<spirit:parameter>` name-value pairs that define the Simulink mapping. For example:

```

<spirit:parameter>
  <spirit:name>MWMapInput</spirit:name>
  <spirit:value>input_1</spirit:value>
</spirit:parameter>

```

This image shows some bus interfaces that are mapped to Simulink inputs.

```

<spirit:busInterfaces>
  <spirit:busInterface>
    <spirit:name>InOut</spirit:name>
    <spirit:slave> </spirit:slave>
    <spirit:parameters>
      <spirit:parameter>
        <spirit:name>MwMapInput</spirit:name>
        <spirit:value>input_1</spirit:value>
      </spirit:parameter>
      <spirit:parameter>
        <spirit:name>MwMapInput</spirit:name>
        <spirit:value>input_2</spirit:value>
      </spirit:parameter>
      <spirit:parameter>
        <spirit:name>MwMapOutput</spirit:name>
        <spirit:value>output_1</spirit:value>
      </spirit:parameter>
      <spirit:parameter>
        <spirit:name>MwMapOutput</spirit:name>
        <spirit:value>output_2</spirit:value>
      </spirit:parameter>
    </spirit:parameters>
  </spirit:busInterface>
  <spirit:busInterface>
    <spirit:name>Config</spirit:name>
    <spirit:slave> </spirit:slave>
    <spirit:parameters>
      <spirit:parameter>
        <spirit:name>MwMapParam</spirit:name>
        <spirit:value>coef_11</spirit:value>
      </spirit:parameter>
      <spirit:parameter>
        <spirit:name>MwMapParam</spirit:name>
        <spirit:value>coef_12</spirit:value>
      </spirit:parameter>
    </spirit:parameters>
  </spirit:busInterface>
</spirit:busInterfaces>

```

The inputs are mapped together in one bus interface definition. The outputs are in a separate bus interface. The filter coefficients are in another, separate bus interface.

Alternatively, you can define the inputs and outputs together in a single bus interface definition. However, the filter coefficients must remain in their own separate bus interface definition.

Bus Interface Definition with Memory Mapping

- “General Guidelines” on page 22-18
- “Simulink Mapping Within a Memory Map” on page 22-19

General Guidelines

Write the bus definitions for your model according to the IEEE Standard for IP-XACT 1685-2009. The following permissions apply:

- Input registers — Write-only or read-write
- Output registers — Read-only or read-write
- Parameters register — Read-only, write-only, or read-write, depending on your requirements

Make the spirit size of each register, in bits, greater than or equal to the size of that Simulink input, output, or parameter.

If you want to use the Simulink mapping, all bus interfaces that contain the Simulink mapping must be slave interfaces.

Simulink Mapping Within a Memory Map

If you have a memory map reference in the bus interface, then you must express the Simulink mapping in the memory map, not in the bus interface.

The Simulink mapping for each register can consist of these element arrangements:

- No mapping to Simulink (that is, no mapping information is needed in the register)
- Mapping to Simulink inputs, Simulink outputs, or a mix of inputs and outputs
- Mapping to Simulink tunable parameters

Registers cannot have multiple input-outputs. However, the bus interface can be mapped to multiple registers, each having a different arrangement.

To add inputs, outputs, or parameters to the IP-XACT file, follow these steps.

- 1** Each `<spirit:busInterface>` definition containing a Simulink mapping is mapped to the TLM target socket. Add a `<spirit:parameter>` name-value pair that indicates to the TLM generator that there is Simulink mapping in the memory map.

```
<spirit:parameter>
  <spirit:name>MwMap</spirit:name>
  <spirit:value>>true</spirit:value>
</spirit:parameter>
```

- 2** In each `<spirit:memoryMap>` section, in each `<spirit:register>` definition, within the `<spirit:parameters>` tag, add a `<spirit:parameter>` name-value pair with the Simulink mapping.

```
<spirit:parameter>
  <spirit:name>MwMapInput</spirit:name>
  <spirit:value>input1</spirit:value>
</spirit:parameter>
```

This image demonstrates this arrangement for a Simulink input.

```

<spirit:busInterfaces>
  <spirit:busInterface>
    <spirit:name>Input</spirit:name>
    <spirit:slave>
      <spirit:memoryMapRef spirit:memoryMapRef="memorymap_input"/>
    </spirit:slave>
    <spirit:parameters>
      <spirit:parameter>
        <spirit:name>MwMap</spirit:name>
        <spirit:value>true</spirit:value>
      </spirit:parameter>
    </spirit:parameters>
  </spirit:busInterface>
</spirit:busInterfaces>
<spirit:memoryMaps>
  <spirit:memoryMap>
    <spirit:name>memorymap_input</spirit:name>
    <spirit:addressBlock>
      <spirit:name>INPUT_REG</spirit:name>
      <spirit:baseAddress spirit:id="base_address_input" spirit:resolve="user">0x00000000</spirit:baseAddress>
      <spirit:range>16</spirit:range>
      <spirit:width>64</spirit:width>
      <spirit:usage>register</spirit:usage>
      <spirit:register>
        <spirit:name>INPUT_REG_1</spirit:name>
        <spirit:addressOffset>0x00</spirit:addressOffset>
        <spirit:size>64</spirit:size>
        <spirit:access>write-only</spirit:access>
        <spirit:reset>
          <spirit:value>0x00</spirit:value>
        </spirit:reset>
        <spirit:parameters>
          <spirit:parameter>
            <spirit:name>MwMapInput</spirit:name>
            <spirit:value>input_1</spirit:value>
          </spirit:parameter>
        </spirit:parameters>
      </spirit:register>
      <spirit:register>
        <spirit:name>INPUT_REG_2</spirit:name>
        <spirit:addressOffset>0x08</spirit:addressOffset>
        <spirit:size>64</spirit:size>
        <spirit:access>write-only</spirit:access>
        <spirit:reset>
          <spirit:value>0x00</spirit:value>
        </spirit:reset>
        <spirit:parameters>
          <spirit:parameter>
            <spirit:name>MwMapInput</spirit:name>
            <spirit:value>input_2</spirit:value>
          </spirit:parameter>
        </spirit:parameters>
      </spirit:register>
    </spirit:addressBlock>
  </spirit:memoryMap>
</spirit:memoryMaps>

```

- 3 To optionally specify field locations within a register, specify a `<spirit:field>` definition in the `<spirit:register>`. Use the `<spirit:bitWidth>` and `<spirit:bitOffset>` tags to define each `<spirit:field>`. Include the `<spirit:parameter>` name-value pair with the Simulink mapping in the `<spirit:field>` definition.

```

<spirit:field>
  <spirit:name>OUTPUT_1</spirit:name>
  <spirit:bitOffset>32</spirit:bitOffset>
  <spirit:bitWidth>32</spirit:bitWidth>
  <spirit:access>read-only</spirit:access>
  <spirit:parameters>
    <spirit:parameter>
      <spirit:name>MwMapOutput</spirit:name>

```

```

        <spirit:value>output_1</spirit:value>
      </spirit:parameter>
    </spirit:parameters>
  </spirit:field>

```

- 4 To optionally exclude a register from the Simulink mapping, add a `<spirit:parameter>` name-value pair in the `<spirit:register>` definition. Specify the name as `MWMap` and the value as `false` to exclude the register from the memory map.

```

<spirit:register>
  <spirit:name>EXCLUDED_REG_1</spirit:name>
  <spirit:addressOffset>0x38</spirit:addressOffset>
  <spirit:size>64</spirit:size>
  <spirit:access>read-only</spirit:access>
  <spirit:parameters>
    <spirit:parameter>
      <spirit:name>MWMap</spirit:name>
      <spirit:value>>false</spirit:value>
    </spirit:parameter>
  </spirit:parameters>
</spirit:register>

```

To exclude an address block from the Simulink memory mapping, add a `<spirit:parameter>` name-value pair in the `<spirit:addressblock>` definition. Specify the name as `MWMap` and the value as `false` to exclude the address block from the memory map.

```

<spirit:addressBlock>
  <spirit:name>EXCLUDED_BANK</spirit:name>
  <spirit:baseAddress spirit:id="EXCLUDED_BANK_ADDR" spirit:resolve="user">0x00030000</spirit:baseAddress>
  <spirit:range>128</spirit:range>
  <spirit:width>64</spirit:width>
  <spirit:usage>register</spirit:usage>
  <spirit:register>
    <spirit:name>EXCLUDED_REG_2</spirit:name>
    <spirit:addressOffset>0x00</spirit:addressOffset>
    <spirit:size>64</spirit:size>
    <spirit:access>read-only</spirit:access>
  </spirit:register>
  <spirit:register>
    <spirit:name>EXCLUDED_REG_3</spirit:name>
    <spirit:addressOffset>0x08</spirit:addressOffset>
    <spirit:size>64</spirit:size>
    <spirit:access>read-only</spirit:access>
  </spirit:register>
  <spirit:parameters>
    <spirit:parameter>
      <spirit:name>MWMap</spirit:name>
      <spirit:value>>false</spirit:value>
    </spirit:parameter>
  </spirit:parameters>
</spirit:addressBlock>

```

For a complete example of Simulink memory mapping to a TLM component, see “Imported IP-XACT with Memory Map” on page 32-285.

Mapping to a Signal Port

You can generate an unregistered `sc_signal` port. When the step function is executed, it reads the current value of the `sc_in` ports, passes them all to the step function, executes the step function and writes the step function result in the `sc_out` ports.

To add input and output ports, specify the following in your IP-XACT file:

- 1 Specify the port as `<spirit:port>` of type `<spirit:wire>`.

- 2 Specify the port direction as `<spirit:direction>`. Set the direction to `in`, to generate an `sc_in` port. Set direction to `out` to generate an `sc_out` port.
- 3 By default, the data type of the port is the same as the subsystem input or output. You can optionally define a data type for the port by describing it in `<spirit:wireTypeDef>`.
- 4 To define the mapping of the TLM port to a Simulink input or output, specify the name-value pair `MWMapInput` or `MWMapOutput` within a `<spirit:vendorExtension>` `<spirit:parameters>` `<spirit:parameter>` tag.

This image shows an example of mapping to ports.

```

<spirit:model>
  <spirit:ports>
    <spirit:port>
      <spirit:name>LPF_input</spirit:name>
      <spirit:wire>
        <spirit:direction>in</spirit:direction>
        <spirit:wireTypeDefs>
          <spirit:wireTypeDef>
            <spirit:typeName>double</spirit:typeName>
            <spirit:typeDefinition>systemc.h</spirit:typeDefinition>
          </spirit:wireTypeDef>
        </spirit:wireTypeDefs>
      </spirit:wire>
      <spirit:vendorExtensions>
        <spirit:parameters>
          <spirit:parameter>
            <spirit:name>MWMapInput</spirit:name>
            <spirit:value>LPF_input</spirit:value>
          </spirit:parameter>
        </spirit:parameters>
      </spirit:vendorExtensions>
    </spirit:port>
    <spirit:port>
      <spirit:name>LPF_output</spirit:name>
      <spirit:wire>
        <spirit:direction>out</spirit:direction>
        <spirit:wireTypeDefs>
          <spirit:wireTypeDef>
            <spirit:typeName>double</spirit:typeName>
            <spirit:typeDefinition>systemc.h</spirit:typeDefinition>
          </spirit:wireTypeDef>
        </spirit:wireTypeDefs>
      </spirit:wire>
      <spirit:vendorExtensions>
        <spirit:parameters>
          <spirit:parameter>
            <spirit:name>MWMapOutput</spirit:name>
            <spirit:value>LPF_output</spirit:value>
          </spirit:parameter>
        </spirit:parameters>
      </spirit:vendorExtensions>
    </spirit:port>
  </spirit:ports>
</spirit:model>

```

See Also

More About

- “Contents of Generated IP-XACT File” on page 22-24

External Websites

- IEEE Standard for IP-XACT 1685-2009

Contents of Generated IP-XACT File

In this section...
“Overview of Generated IP-XACT File” on page 22-24
“Generated Simulink Mapping” on page 22-24
“Generated Simulink Mapping in Memory Map” on page 22-25
“Generated Metadata” on page 22-26

Overview of Generated IP-XACT File

The TLM generator automatically generates an IP-XACT file that complies with IEEE Standard for IP-XACT 1685-2009. You can find this file in the same folder as the generated makefile.

The generated IP-XACT file contains the following:

- Mapping information between Simulink and the generated TLM component.
- Metadata specific to MathWorks and the model. This data is intended primarily for reference, but it is required when importing the file for TLM generation.

Generated Simulink Mapping

Each bus interface that uses Simulink mapping without a memory map is defined in the generated file as:

- Inputs
- Outputs
- A combination of inputs and outputs
- Parameters

You can combine inputs and outputs in a single bus interface definition, but you cannot mix parameters and I/O. These elements are defined in a `<spirit:parameter>` name-value pair.

This example from a generated IP-XACT file shows Simulink mapping without a memory map.

```

- <spirit:busInterfaces>
  - <spirit:busInterface>
    <spirit:name>InOut</spirit:name>
    <spirit:slave> </spirit:slave>
    - <spirit:parameters>
      - <spirit:parameter>
        <spirit:name>MWMMapInput</spirit:name>
        <spirit:value>input_1</spirit:value>
      </spirit:parameter>
      - <spirit:parameter>
        <spirit:name>MWMMapInput</spirit:name>
        <spirit:value>input_2</spirit:value>
      </spirit:parameter>
      - <spirit:parameter>
        <spirit:name>MWMMapOutput</spirit:name>
        <spirit:value>output_1</spirit:value>
      </spirit:parameter>
      - <spirit:parameter>
        <spirit:name>MWMMapOutput</spirit:name>
        <spirit:value>output_2</spirit:value>
      </spirit:parameter>
    </spirit:parameters>
  </spirit:busInterface>
</spirit:busInterfaces>

```

Generated Simulink Mapping in Memory Map

In each bus interface with a memory map, the Simulink mapping is expressed in the memory map, not in the bus interface.

The bus interface definition, <spirit:busInterface>, contains a <spirit:parameter> name-value pair indicating that there is a memory map in use for the interface.

```

- <spirit:parameters>
  - <spirit:parameter>
    <spirit:name>MWMMap</spirit:name>
    <spirit:value>true</spirit:value>
  </spirit:parameter>
</spirit:parameters>

```

The memory map interface, <spirit:memoryMap>, contains a <spirit:parameter> name-value pair with the Simulink mapping within each register:

```
- <spirit:register>
  <spirit:name>INPUT_REG_1</spirit:name>
  <spirit:addressOffset>0x0</spirit:addressOffset>
  <spirit:size>64</spirit:size>
  <spirit:access>write-only</spirit:access>
  - <spirit:reset>
    <spirit:value>0x00</spirit:value>
  </spirit:reset>
  - <spirit:parameters>
    - <spirit:parameter>
      <spirit:name>MWMapInput</spirit:name>
      <spirit:value>input_1</spirit:value>
    </spirit:parameter>
  </spirit:parameters>
</spirit:register>
```

The Simulink mapping for each register is defined in the generated file as:

- Input
- Outputs
- A combination of inputs and outputs
- Parameters

You can combine inputs and outputs in a single bus interface definition, but you cannot mix parameters and I/O.

Generated Metadata

Each component definition, `<spirit:component>`, contains information specific to MathWorks and the model. This information is located within a `<spirit:parameter>` element, specified with the `<spirit:name>` and `<spirit:value>` tags. If you plan to import the generated IP-XACT file for use with the TLM generator, these fields are required.

This example shows the metadata in a generated IP-XACT file.

```
- <spirit:component xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/index.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5">
.
.
.
- <spirit:parameters>
- <spirit:parameter>
  <spirit:name>MWVendor</spirit:name>
  <spirit:value>MathWorks</spirit:value>
</spirit:parameter>
- <spirit:parameter>
  <spirit:name>MWVersion</spirit:name>
  <spirit:value>1.0</spirit:value>
</spirit:parameter>
- <spirit:parameter>
  <spirit:name>MWModel</spirit:name>
  <spirit:value>timgdemo_intro</spirit:value>
</spirit:parameter>
- <spirit:parameter>
  <spirit:name>MWBlock</spirit:name>
  <spirit:value>DualFilter</spirit:value>
</spirit:parameter>
</spirit:parameters>
</spirit:component>
```

See Also

Related Examples

- “Prepare IP-XACT File for Import” on page 22-16

More About

- IEEE Standard for IP-XACT 1685-2009

Implement Memory Map with SCML

In this section...

“What Is SCML?” on page 22-28

“Workflow” on page 22-28

“Generated Code” on page 22-28

What Is SCML?

The SystemC Modeling Library (SCML) is a TLM 2.0 compatible API library for creating TLM model interfaces for use with Synopsys prototyping tools. These tools enable early software integration and testing. The SCML interface provides backdoor register access for the Synopsys tools during simulation. Use HDL Verifier software to export a TLM component with an SCML interface for seamless use with the Synopsys prototyping tools.

Workflow

To generate a TLM component with SCML memory map:

- 1 Install SCML. You can download SCML from Synopsys, see <https://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi>.
- 2 Open **Configuration Parameters>Code Generation>TLM Generator**. See “Select TLM Generator System Target” on page 22-4.
- 3 On the **TLM Mapping** tab, provide an IP-XACT file describing the memory map of your component. Then select the SCML option. See “Select TLM Mapping Options” on page 22-6.
- 4 Specify the location of your SCML installation on the **TLM Compilation** tab. See “Select TLM Compilation Options” on page 22-12.
- 5 Generate code for your model as you would for any other model. See “Generate Component and Test Bench” on page 22-15.

Generated Code

When you generate code for your model, the TLM generator creates the same set of files to implement the TLM component as it would without SCML. The files are named *SystemName_scml* rather than *SystemName_tlm*.

SCML supports bit widths of 8, 16, 32, 64, 128, and 256. When generating the SCML interface for Simulink signals, the generator rounds up to the next supported size.

IP-XACT classes are translated to SCML classes according to this mapping.

IP-XACT Class	SCML Class
spirit::businterface	scml2::tlm2_gp_target_adapter
spirit:addressBlock	scml2::memory
spirit:register	scml2::reg

IP-XACT Class	SCML Class
spirit:field	scml2::bitfield

The SCML interface has no effect on test bench generation for the TLM component. The test bench does not use the SCML access functions.

See Also

External Websites

- <https://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi>

Run TLM Component Test Bench

- “Testing TLM Components” on page 23-2
- “Run TLM Component Test Bench” on page 23-5

Testing TLM Components

In this section...

“TLM Component Test Bench Overview” on page 23-2

“TLM Component Compilation” on page 23-2

“Automatic Verification of the Generated Component” on page 23-2

“Report Generation” on page 23-3

“Working with Configurations” on page 23-3

“Considerations When Creating a TLM Component Test Bench” on page 23-3

TLM Component Test Bench Overview

The test bench generation option is controlled by the **TLM Testbench** tab of the Configuration Parameters dialog box. This option creates a standalone SystemC test bench for the generated component. The test bench works by applying test vectors against the generated TLM component and checking the results of each transaction. When you click the **Verify TLM Component** button on the **TLM Testbench** tab, the test vectors are automatically captured from a Simulink simulation of your model .

You can configure the generated test bench to specify the timing mode and the triggering modes for input and output buffering. The latter choice allows you to indicate whether the initiator module controls moving input and output data sets between the registers and the buffers or whether the component performs the moves automatically. Optionally, the test bench can also produce verbose messages at runtime to help you see the status of the SystemC simulation.

Note A TLM test bench is not supported when you generate a component for a host with a different operating system from your MATLAB machine.

TLM Component Compilation

The **TLM Compilation** tab in the Configuration Parameters dialog box provides SystemC and TLM library location information. You can use environment variables to specify these locations.

The information you provide is used to construct makefile. You can use these makefiles to build the component and test bench. You can also use this makefile to build an executable of the TLM component and test bench outside of the MATLAB environment.

Automatic Verification of the Generated Component

The **TLM Testbench** tab of the configuration parameters provides a **Verify TLM Component** button that:

- Automatically generates input stimulus and expected output data
- Builds and executes the component and the test bench together
- Automatically checks the outputs of the component

It performs the checking by capturing the outputs from the SystemC simulation, converting them to Simulink data, and comparing them in Simulink to the results of the Simulink simulation.

Report Generation

The `tlmgenerator` target supplies an HTML document containing details about the generated component. The document contains links to the generated source code files. Report generation can be configured via the Simulink Coder **Report** pane in the configuration parameters. Report generation is not strictly a test bench feature, but the process does include use of test bench files.

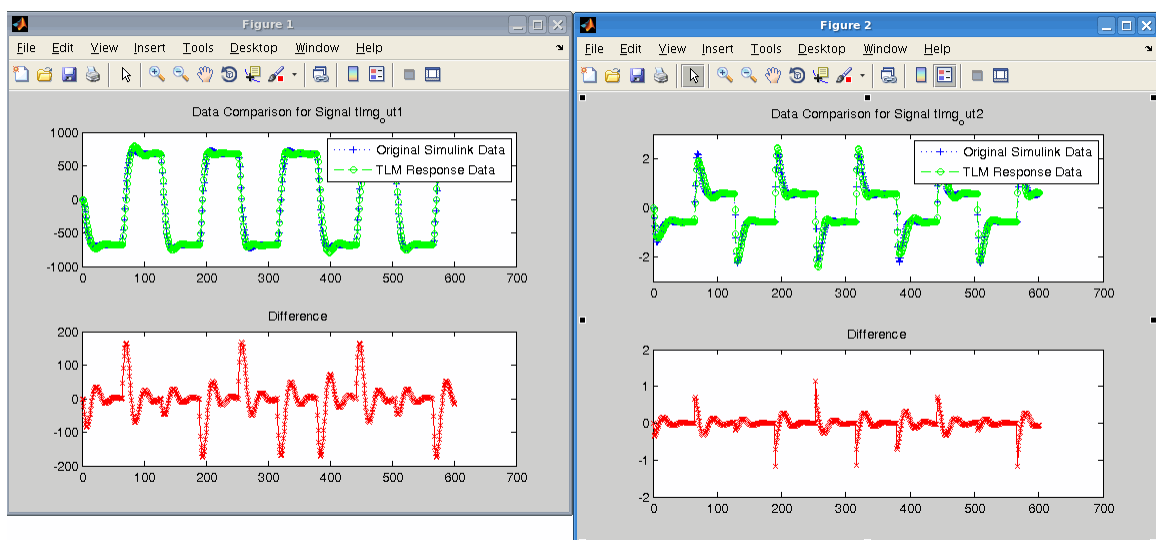
Working with Configurations

After you select configuration options, you can save them with your Simulink model. You can also restore saved configurations made in a previous session. In addition, you can save and choose from multiple configurations for a given model. See the section "Overview of Model Referencing" in the Simulink documentation, for information on working with configurations.

Considerations When Creating a TLM Component Test Bench

For optimizing your generated TLM code and achieving the desired test bench, you should keep the following considerations in mind when developing your Simulink model:

- Your model can use only a single rate.
- The composite signals on your model must be contiguous in memory. You can make mux and bus output signals contiguous with the Signal Conversion block.
- If your model contains complex signals, you must split them first. Split complex signals with the Simulink Complex to Real-Imag block. You can then combine the signals again with the Real-Imag to Complex block on the other side of your design.
- Your design can contain a Triggered or Enabled subsystem, but the design you generate cannot itself be a Triggered or Enabled subsystem.
- HDL Verifier can generate a Simulink design that involves continuous time signals. When the Simulink simulation and the captured vector replay in SystemC, they may not yield exactly the same results. The plot of the difference reveals essentially the same curve with numerical differences that are more pronounced at signal transitions, as shown in the following MATLAB Figure windows.



This difference occurs because the Simulink signal capture necessarily makes the signals discrete and thus the same exact data is not used in both the Simulink and stand-alone SystemC simulations. You can improve the fidelity of the discrete signal simulation in SystemC by choosing a smaller fundamental step size in Simulink before clicking **Verify TLM Component**.

Run TLM Component Test Bench

After the TLM component and test bench have been generated, you can verify the generated TLM component using the test bench that was just created:

- 1** Open Model Configuration Parameters. Click on TLM Generation.
- 2** Select the **TLM Testbench** pane.
- 3** Click **Verify TLM Component**. The software performs the following actions:
 - Builds the generated code using make and generated makefiles.
 - Runs Simulink to capture input stimulus and expected results.
 - Converts the Simulink data to TLM vectors.
 - Runs the standalone SystemC/TLM test bench executable.
 - Converts the TLM results back to Simulink data.
 - Performs a data comparison.
 - Generates a Figure window for any signals that had data miscompares.

Note You must generate the component and test bench before you can select **Verify TLM Component**. See “Generate Component and Test Bench” on page 22-15.

Export TLM Component to SystemC Environment

- “Export TLM Component” on page 24-2
- “TLM Component Constructor” on page 24-6

Export TLM Component

In this section...

“Identify Generated Files” on page 24-2

“Create Static Library with TLM Component” on page 24-3

“Create Standalone Executable with TLM Component” on page 24-4

Identify Generated Files

After code generation completes, go to your working folder. There you can find the following folder: *model_name_VP/*. This folder contains the files generated for the TLM component. The files appear under the subfolders described in the following table.

Directory Name	Files	Description
<i>model_name</i>	<code>include/model_name*.h</code> <code>src/model_name.cpp</code>	Files relative to the behavior of the model. These files are independent of the TLM options. HDL Verifier provides a makefile for you to build a static library from these source files. If another TLM component is generated from the same model, these files are regenerated (if the model has not changed, the files will be identical). If you generate a second TLM version of the same model with a different tag the TLM files are added to the <i>_VP</i> folder with the new tag. It is possible for the <i>_VP</i> folder to contain multiple TLM variations of the same model all using the same behavior files.
<i>model_name_usertag_tlm</i>	<code>include/model_name_usertag_tlm.h</code> <code>src/model_name_usertag_tlm.cpp</code> <code>include/model_name_usertag_tlm_def.h</code>	These files contain the TLM interface to wrap the core behavior. This file contains addresses and definitions to communicate with the component through the TLM target port using a TLM generic payload. The files are sorted in subdirectories by source and header. HDL Verifier provides a makefile for you to build a static library from these source files.

Directory Name	Files	Description
<i>model_name_usertag_tlm_tb</i>	<pre>include/model_name_usertag_tlm_tb.h src/model_name_usertag_tlm_tb.cpp src/model_name_usertag_tlm_tb_main.cpp</pre>	<p>These files contain the core behavior of the test bench.</p> <p>This file instantiates and binds the component and the test bench together.</p> <p>The files are sorted in subdirectories by source and header.</p> <p>HDL Verifier software provides a makefile for you to build an executable from these source file and the component static library. This executable requires the following:</p> <ul style="list-style-type: none"> • Certain MATLAB libraries the executable needs to be built and run. These MATLAB libraries are the static libraries <code>libmat.a</code> and <code>libmx.a</code> and their dynamic counterparts. • The vector <code>.mat</code> files generated when you click the Verify TLM Component button. Before building the component and test bench on the virtual platform, verify that the TLM component includes these files.
<i>model_name_usertag_tlm_doc/</i>	<code>html/model_name_codegen_rpt.html</code>	This file is the entry point of the HTML documentation.

Create Static Library with TLM Component

Create a static library that contains the generated TLM component by following the steps described for Linux or Windows. Execute these steps for the operating system where you will run the TLM component.

Linux Users

- 1 Open a Linux console window.
- 2 Navigate to the `model_name_VP/model_name_usertag_tlm/` folder.
- 3 Execute the following command to start the library compilation:

```
make -f makefile.gnu all
```

If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.

- 4 When the system finishes compiling, locate a library file named `libmodel_name_usertag_tlm.a` in the `model_name_VP/model_name_usertag_tlm/lib/` folder.

Windows Users

If you have not already, make sure that `MATLAB\version\bin\win32` or `MATLAB\version\bin\win64` has been added to your user path.

You can choose one of the following ways to compile your project:

- Compile in Visual Studio (open the *model_name_usertag_tlm.vcproj* project in Visual Studio and follow the application instructions for compiling your project).
- Compile in a console window.

- 1 Open a system console window.
- 2 Load the compilation tool chain by entering the following at the system prompt:

Win32 users:

```
X:\>"%VS80COMNT00LS%\..\..\VC\vcvarsall" x86
```

Win64 users:

```
X:\>"%VS80COMNT00LS%\..\..\VC\vcvarsall" x64
```

If you have a later version of Visual Studio, you may need to enter "%VS100COMNT00LS%...", "%VS90COMNT00LS%..." or "%VS80COMNT00LS%..." instead. Type set at the system prompt for a list of environment variables; in that list you can find the environment variable pointing to where the tool chain is installed.

- 3 In the *same* system console, navigate to the *model_name_VP/model_name_usertag_tlm/* folder.
- 4 Execute the following command to start the library compilation:

```
X:\>nmake /f makefile.mk all
```

If you want to obtain symbols for source code debugging, use the *all-debug* target instead of *all*.

- 5 When the system finishes compiling, locate a library file named *model_name_usertag_tlm.lib* in the *model_name_VP/model_name_usertag_tlm/lib/* folder.

Note The temporary object files reside in the *model_name_VP/model_name_usertag_tlm/obj/* folder.

Create Standalone Executable with TLM Component

You can create a standalone TLM executable in the command shell by following the steps for Linux or Windows. Execute these steps for the operating system where you will run the TLM component.

Linux Users

- 1 Open a Linux console window.
- 2 Navigate to the *model_name_VP/model_name_usertag_tlm_tb/* folder.
- 3 Execute the following command to start the library compilation:

```
make -f makefile_tb.gnu all
```

If you want to obtain symbols for source code debugging, use the *all-debug* target instead of *all*.

Note Executing this command also automatically builds a static library with the TLM component source files.

- 4 When the system finishes compiling, locate an executable file named *model_name_usertag_tlm_tb.exe* in the *model_name_VP/model_name_usertag_tlm_tb/* folder.

Windows Users

If you have not already, make sure that `MATLAB\version\bin\win32` or `MATLAB\version\bin\win64` has been added to your user path.

You can choose one of the following ways to compile your project:

- Compile in Visual Studio (open the *model_name_usertag_tlm.vcproj* project in Visual Studio and follow the application instructions for compiling your project).
- Compile in a console window.

- 1 Open a system console window.

- 2 Load the compilation tool chain by entering the following at the system prompt:

Win32 users:

```
X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x86
```

Win64 users:

```
X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x64
```

If you have a later version of Visual Studio, you may need to enter "`%VS100COMNTOOLS%. . .`", "`%VS90COMNTOOLS%. . .`" or "`%VS80COMNTOOL%. . .`" instead. Type `set` at the system prompt for a list of environment variables; in that list you can find the environment variable pointing to where the tool chain is installed.

- 3 In the *same* system console, navigate to the *model_name_VP/model_name_usertag_tlm_tb/* folder.

- 4 Execute the following command to start the library compilation:

```
X:\>nmake /f makefile.mk all
```

If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.

Note Executing this command also automatically builds a static library with the TLM component source files.

- 5 When the system finishes compiling, locate an executable file named *model_name_usertag_tlm_tb.exe* in the *model_name_VP/model_name_usertag_tlm_tb/* folder.

TLM Component Constructor

The generated TLM component has the following constructor function prototype:

```
model_name_usertag_tlm(sc_core::sc_module_name module_name, ...
    eTimingType DefaultTiming = TIMED,
    eModeType InputDefaultMode = AUTO, eModeType OutputDefaultMode = AUTO);
```

Where:

- `module_name` is a `sc_core::sc_module_name` type. It is a character vector that contains the instance name.
- `DefaultTiming` is an `eTimingType` {TIMED, UNTIMED}. It determines whether the TLM component is timed or untimed at the beginning of the SystemC simulation. By default, the component initializes `DefaultTiming` to TIMED, but you can change it to UNTIMED. Also during the simulation, you can change the TLM component timing by calling the function `SetTimingParam` (`eTimingType` Type).
- `InputDefaultMode` is an `eModeType` { MANUAL,AUTO}. It determines whether the TLM component input mode is manual or auto at the beginning of the SystemC simulation (and also after SystemC resets the component). By default, the TLM component initializes `InputDefaultMode` to AUTO, but you can change it to MANUAL.
- `OutputDefaultMode` is an `eModeType` { MANUAL,AUTO}. It determines whether the TLM component output mode is manual or auto at the beginning of the SystemC simulation (and also after SystemC resets the component). By default, the TLM component initializes `OutputDefaultMode` to AUTO, but you can change it to MANUAL.

Configuration Parameters for TLM Generator Target

TLM Component Generation

In this section...

“TLM Mapping” on page 25-2
 “TLM Processing” on page 25-7
 “TLM Timing” on page 25-8
 “TLM Testbench” on page 25-9
 “TLM Compilation” on page 25-13

TLM Mapping

TLM Generator Overview

The following user interface tabs contain the parameters for setting options on the generated TLM component:

- **TLM Mapping:** Specify options for socket and memory mapping. See “Select TLM Mapping Options” on page 22-6.
- **TLM Processing:** Specify options for algorithm and interface processing. See “Select TLM Processing Options” on page 22-8.
- **TLM Timing:** Specify options for combined interface timing, or for individual timing for input data, output data, and control sockets. See “Select TLM Timing Options” on page 22-9.
- **TLM Testbench:** Specify options for the generation and runtime behavior of a standalone SystemC/TLM component test bench. See “Select TLM Test Bench Options” on page 22-10.
- **TLM Compilation:** Specify generated TLM component compilation options. See “Select TLM Compilation Options” on page 22-12.

Socket Mapping

Choose the type of TLM socket for input data, output data, and control.

Settings

Default: One combined TLM socket for input data, output data, and control

- **One combined TLM socket for input data, output data, and control:** Create one combined TLM socket in the generated TLM component.
- **Three separate TLM socket for input data, output data, and control:** Create three separate TLM sockets. Generate each data socket with the following options:
 - Auto-generated memory map (or without memory map)
 - Command and status registers
 - Test and set registers
 - Tunable parameter registers
- **Defined by imported IP-XACT file:** Define the memory map for the TLM component according to instructions in an IP-XACT file. When you select this option, you must specify the IP-XACT file to use. See “Prepare IP-XACT File for Import” on page 22-16.

Dependencies

This parameter enables **Combined TLM Socket** or **TLM Socket for Input Data, TLM Socket for Output Data**, and **TLM Socket for Control with Memory Map**.

Setting this parameter to `One combined TLM socket for input data, output data, and control` opens the **Combined TLM Socket** options selection.

Setting this parameter to `Three separate TLM socket for input data, output data, and control` opens the **TLM Socket for Input Data, TLM Socket for Output Data, and TLM Socket for Control with Memory Map** options selection.

Setting this parameter to `Defined by imported IP-XACT file` opens the **Import IP-XACT File** options selection.

Command-Line Information

Parameter: `tlmgComponentSocketMapping`

Type: `string`

Value: |

Default:

Memory Map Type

Choose the type of addressing scheme for the combined TLM socket or the separate TLM input data and output data sockets.

Settings

Default: No memory map

- **No memory map:** Create a single input register and a single output register in the generated TLM component
- **Auto-generate memory map:** Create a single input address and a single output address for all inputs and outputs or create a separate input register for every input signal and a separate output register for every output signal

Dependencies

This parameter enables **Auto-Generated memory map Type**.

Setting this parameter to `Auto-generate memory map` opens the **Auto-Generated Memory Map Type** options selection.

Command-Line Information

Parameter: `tlmgComponentAddressing (for combined TLM socket)|
tlmgComponentAddressingInput | tlmgComponentAddressingOutput`

Type: `string`

Value: `'No memory map' | 'Auto-generated memory map'`

Default: `'No memory map'`

See Also

“Memory Mapping” on page 20-3

Auto-Generated Memory Map Type

Choose the type of addressing scheme to be automatically generated.

Settings

Default: Single input and output address offsets

- **Single input and output address offsets:** Create a single address offset for the inputs and a single address offset for the outputs
- **Individual input and output address offsets:** Generate an address for each input and each output

Dependencies

Auto-Generated memory map enables this parameter.

Command-Line Information

Parameter: `tlmgAutoAddressSpecType` (for combined TLM socket) | `tlmgAutoAddressSpecTypeInput` | `tlmgAutoAddressSpecTypeOutput`

Type: string

Value: 'Single input and output address offsets' | 'Individual input and output address offsets'

Default: 'Single input and output address offsets'

See Also

“Memory Mapping” on page 20-3

Import IP-XACT File

Define the memory map of the TLM component from an imported file.

Settings

Default: No file specified, no SCML implementation

- **IP-XACT file:** Specify a file that defines the memory map for the TLM component, in IP-XACT format.
- **Generate code for unmapped IP-XACT registers/bitfields:** Include registers without Simulink mapping in the generated TLM component.
- **Generate code for unmapped IP-XACT signal ports:** Include signal ports without Simulink mapping in the generated TLM component.
- **Implement memory map with SCML:** Generate an interface compatible with the SystemC Modeling Library (SCML). See “Implement Memory Map with SCML” on page 22-28.

Dependencies

When you select **Implement memory map with SCML**, also set path variables for the SCML libraries on the **TLM Compilation** tab. When you select **Implement memory map with SCML**, the **Algorithm Step Function Execution** option on the **TLM Processing** tab is set to **SystemC Thread**.

Command-Line Information

Parameter: `tlmgIPXACTPath`

Type: string

Value:
Default:
Parameter: tlmgIPXactUnmapped
Type: string
Value: 'on' | 'off'
Default: 'off'
Parameter: tlmgIPXactUnmappedSig
Type: string
Value: 'on' | 'off'
Default: 'off'
Parameter: tlmgSCMLOnOff
Type: string
Value: 'on' | 'off'
Default: 'off'

See Also

“Memory Mapping” on page 20-3

Include a command and status register in the memory map

Allows an initiator to send the TLM component commands such as "reset" and "start", as well as read status bits such as "interrupt active", "output buffer overflowed", and "input buffer empty".

Settings

Default: On

On

Include a command and status register in the memory map

Off

Do not include a command and status register in the memory map

Dependencies

If you selected a combined TLM socket, **Auto-Generated Memory Map** enables this parameter.

If you selected Separate TLM sockets, this parameter is automatically enabled for the control TLM socket.

Command-Line Information

Parameter: tlmgCommandStatusRegOnOff
Type: string
Value: 'on' | 'off'
Default: 'on'

See Also

“Command and Status Register” on page 20-7

Include a test and set register in the memory map

Provides a means of controlling access to a shared TLM target device in your SystemC environment.

Settings**Default:** Off On

Include a test and set register in the memory map. Any read of this register will return the current value and set the register to a new, asserted value in an atomic operation.

 Off

Do not include a test and set register in the memory map

Dependencies

If you selected a combined TLM socket, **Auto-Generated Memory Map** enables this parameter.

If you selected separate TLM sockets, this parameter is automatically enabled for the control TLM socket.

Command-Line Information**Parameter:** tlmgTestAndSetRegOnOff**Type:** string**Value:** 'on' | 'off'**Default:** 'off'**See Also**

“Test and Set Register” on page 20-11

Include tunable parameter registers in the memory map

The read/write tunable parameter registers are used by the initiator to change the values of the algorithm tunable parameters.

Settings**Default:** On On

Include tunable parameter registers in the memory map

 Off

Do not include tunable parameter registers in the memory map

Dependencies

If you selected a combined TLM socket, **Auto-Generated Memory Map** enables this parameter.

If you selected separate TLM sockets, this parameter is automatically enabled for the control TLM socket.

Command-Line Information**Parameter:** tlmgTunableParamRegOnOff**Type:** string**Value:** 'on' | 'off'

Default: 'on '

See Also

“Memory Map Configuration” on page 22-7

TLM Processing

Algorithm Step Function Execution

Choose the type of function execution trigger you want to use in the generated TLM component.

Settings

Default: SystemC Thread

- **SystemC Thread:** Event triggers system scheduler to execute function
- **Callback:** Function is executed as soon as input buffer is full or command is written to command register
- **Periodic SystemC Thread:** Function is executed by a periodic thread. The period is derived from the Simulink sample rate.

Command-Line Information

Parameter: tlmAlgorithmProcessingType

Type: string

Value: 'SystemC Thread' | 'Callback' | 'Periodic SystemC Thread'

Default: 'SystemC Thread'

See Also

“Select TLM Processing Options” on page 22-8

Algorithm step function timing (ns)

Specify the time in nanoseconds for modeling the algorithm execution time in the TLM environment.

Settings

Default: 100

Command-Line Information

Parameter: tlmAlgorithmProcessingTime

Type: int

Value:

Default: 100

See Also

“Select TLM Processing Options” on page 22-8

Create an interrupt request port on the generated TLM component

Specify that an interrupt signal be added to the generated TLM component.

Settings**Default:** Off On

Create an interrupt request port on the generated TLM component. This signal will be asserted whenever new outputs are available in the output register(s) and will be automatically cleared whenever any value is read from the output register(s).

 Off

Do not create an interrupt request port on the generated TLM component

Command-Line Information**Parameter:** tlmgIrqPortOnOff**Type:** string**Value:** 'on' | 'off'**Default:** 'on'**See Also**

“Interrupt” on page 20-10

TLM Timing**Single write transfer or the first write transfer in a burst transaction (ns)**

Specify the time in nanoseconds for the TLM component to execute a single write transfer or the first write transfer in a burst transaction.

Settings**Default:** 10**Command-Line Information****Parameter:** tlmgFirstWriteTime (for combined TLM socket) | tlmgFirstWriteTimeInput | tlmgFirstWriteTimeCtrl**Type:** int**Value:****Default:** 10**See Also**

“Select TLM Timing Options” on page 22-9

Subsequent write transfers in a burst transaction (ns)

Specify the time in nanoseconds for the TLM component to execute a subsequent write transfer in a burst transaction.

Settings**Default:** 10

Command-Line Information

Parameter: tlmgSubsequentWritesInBurstTime (for combined TLM socket) | tlmgSubsequentWritesInBurstTimeInput | tlmgSubsequentWritesInBurstTimeCtrl

Type: int

Value:

Default: 10

See Also

“Select TLM Timing Options” on page 22-9

Single read transaction or the first read transfer in a burst transaction (ns)

Specify the time in nanoseconds for the TLM component to execute a single read transaction or the first read transaction in a burst transaction.

Settings

Default: 10

Command-Line Information

Parameter: tlmgFirstReadTime (for combined TLM socket) | tlmgFirstReadTimeOutput | tlmgFirstReadTimeCtrl

Type: int

Value:

Default: 10

See Also

“Select TLM Timing Options” on page 22-9

Subsequent read transfers in a burst transaction (in ns)

Specify the time in nanoseconds for the TLM component to execute a subsequent read transfer in a burst transaction.

Settings

Default: 10

Command-Line Information

Parameter: tlmgSubsequentReadsInBurstTime (for combined TLM socket) | tlmgSubsequentReadsInBurstTimeOutput | tlmgSubsequentReadsInBurstTimeCtrl

Type: int

Value:

Default: 10

See Also

“Select TLM Timing Options” on page 22-9

TLM Testbench**Generate testbench**

Generate a standalone SystemC test bench in order to verify the generated TLM component using the same input stimulus as used in Simulink.

Settings

Default: On

- On
Generate test bench for TLM component
- Off
Do not generate test bench

Dependencies

This parameter enables all other parameters on this tab.

Command-Line Information

Parameter: tlmGenerateTestbench

Type: string

Value: 'on' | 'off'

Default: 'on'

See Also

“Testing TLM Components” on page 23-2

Generate verbose messages during testbench execution

Generate verbose messages during test bench execution.

Settings

Default: Off

- On
Test bench generates verbose runtime messages
- Off
Test bench does not generate verbose messages

Dependencies

Generate testbench enables this parameter.

Command-Line Information

Parameter: tlmVerboseTbMessagesOnOff

Type: string

Value: 'on' | 'off'

Default: 'off'

See Also

“Select TLM Test Bench Options” on page 22-10

Run-time timing mode

Specify the timing mode to be used by the generated test bench and TLM component.

Settings

Default: With timing

- **With timing:** The target annotates TLM component transactions with delays and the initiator will honor them. The initiator synchronizes immediately following the transaction execution.
- **Without timing:** The target does not annotate TLM component transaction with any delays. The initiator and target only perform synchronization using zero-time wait calls.

Dependencies

Generate testbench enables this parameter.

Command-Line Information

Parameter: tlmgRuntimeTimingMode

Type: string

Value: 'With timing' | 'Without timing'

Default: 'With timing'

See Also

“Select TLM Test Bench Options” on page 22-10

Input buffer triggering mode

Specify when data is moved from the input register to the execution buffer. In your TLM environment, this specification is done via a runtime configuration command and can be changed dynamically throughout simulation.

Settings

Default: Automatic

- **Automatic:** The TLM component automatically moves input data sets from the input registers to the input buffer.
- **Manual:** The initiator must explicitly write a command to the command and status register in order to move the input data set from the register to the input buffer.

Dependencies

The following parameters must each be selected to enable the **Input buffer triggering mode** parameter:

- **Include a command and status register in the memory map:** Must be selected.
- **Generate testbench:** Must be selected.

Command-Line Information

Parameter: tlmgInputBufferTriggerMode

Type: string

Value: 'Automatic' | 'Manual'

Default: 'Automatic'

See Also

“Select TLM Test Bench Options” on page 22-10

Output buffer triggering mode

Specify when data is moved from the results buffer to the output register. In your TLM environment, this specification is done via a runtime configuration command and can be changed dynamically throughout simulation.

Settings

Default: Automatic

- **Automatic:** The TLM component automatically moves output data sets from the output buffer to the output registers.
- **Manual:** The initiator must explicitly write a command to the command and status register in order to move the output data set from the output buffer to the output registers.

Dependencies

The following parameters must each be selected to enable the **Input buffer triggering mode** parameter:

- **Include a command and status register in the memory map:** Must be selected.
- **Generate testbench:** Must be selected.

Command-Line Information

Parameter: tlmOutputBufferTriggerMode

Type: string

Value: 'Automatic' | 'Manual'

Default: 'Automatic'

See Also

“Select TLM Test Bench Options” on page 22-10

Component Verification

Click **Verify TLM Component** to verify the generated TLM component. The software then performs the following actions:

- Builds the generated code using make and generated makefiles.
- Runs Simulink® to capture input stimulus and expected results.
- Converts the Simulink data to TLM vectors.
- Runs the standalone SystemC/TLM test bench executable.
- Converts the TLM results back to Simulink data.
- Performs a data comparison.
- Generates a Figure window for any signals that had data mismatches.

Dependencies

To enable this button, you must have previously generated code for your model, as well as a testbench. **Generate testbench** must be selected.

See Also

“Select TLM Test Bench Options” on page 22-10

TLM Compilation**SystemC include path**

Specify the SystemC include path. This string is written directly into the generated makefiles. The default is chosen such that if you define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: \$(SYSTEMC_INC_PATH)

Command-Line Information

Parameter: tlmgSystemCIncludePath

Type: string

Value:

Default: '\$(SYSTEMC_INC_PATH)'

See Also

“Select TLM Compilation Options” on page 22-12

SystemC library path

Specify the location of the library directory in your SystemC installation. This string is written directly into the generated makefiles. The default is chosen such that if you define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: \$(SYSTEMC_LIB_PATH)

Command-Line Information

Parameter: tlmgSystemCLibPath

Type: string

Value:

Default: '\$(SYSTEMC_LIB_PATH)'

See Also

“Select TLM Compilation Options” on page 22-12

SystemC library name

Specify the name of the SystemC library in your SystemC installation. This string is written directly into the generated makefiles. The default is chosen such that if you define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: `$(SYSTEMC_LIB_NAME)`

Command-Line Information

Parameter: `tlmgSystemCLibName`

Type: string

Value:

Default: `'$(SYSTEMC_LIB_NAME)'`

See Also

“Select TLM Compilation Options” on page 22-12

TLM include path

Specify the location of the TLM include directory in your TLM installation. This string is written directly into the generated makefiles. The default is chosen such that if you define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: `$(TLM_INC_PATH)`

Command-Line Information

Parameter: `tlmgTLMIncludePath`

Type: string

Value:

Default: `'$(TLM_INC_PATH)'`

See Also

“Select TLM Compilation Options” on page 22-12

SCML include path

Specify the SCML include path. This string is written directly into the generated makefiles. The default is chosen such that if you define the environment variable you should be able to update your SCML installation without having to update your Simulink models.

Settings

Default: `$(SCML_INC_PATH)`

Command-Line Information

Parameter: `tlmgSCMLIncludePath`

Type: string

Value:

Default: `'$(SCML_INC_PATH)'`

See Also

“Select TLM Compilation Options” on page 22-12

SCML library path

Specify the SCML library path. This string is written directly into the generated makefiles. The default is chosen such that if you define the environment variable you should be able to update your SCML installation without having to update your Simulink models.

Settings

Default: \$(SCML_LIB_PATH)

Command-Line Information

Parameter: tlmgSCMLLibPath

Type: string

Value:

Default: '\$(SCML_LIB_PATH)'

See Also

“Select TLM Compilation Options” on page 22-12

SCML library name

Specify the SCML library name. This string is written directly into the generated makefiles. The default is chosen such that if you define the environment variable you should be able to update your SCML installation without having to update your Simulink models.

Settings

Default: \$(SCML_LIB_NAME)

Command-Line Information

Parameter: tlmgSCMLLibName

Type: string

Value:

Default: '\$(SCML_LIB_NAME)'

See Also

“Select TLM Compilation Options” on page 22-12

SCML logging library name

Specify the name of the SCML logging library. This string is written directly into the generated makefiles. The default is chosen such that if you define the environment variable you should be able to update your SCML installation without having to update your Simulink models.

Settings

Default: \$(SCML_LOGGING_LIB_NAME)

Command-Line Information**Parameter:** `tlmgSCMLLoggingLibName`**Type:** string**Value:****Default:** `'$(SCML_LOGGING_LIB_NAME)'`**See Also**

“Select TLM Compilation Options” on page 22-12

Operating System

Specify the target operating system for the generated TLM code.

Settings**Default:** `'Current Host'`**Command-Line Information****Parameter:** `tlmgTargetOSSelect`**Type:** string**Value:** `'Current Host' | 'Linux 64' | 'Windows 64'`**Default:** `'Current Host'`**See Also**

“Select TLM Compilation Options” on page 22-12

Toolchain

Specify a compiler from the drop-down list. The available options list the compiler versions installed on your computer; the default option is the version most recently installed.

Settings

Available options are compiler versions installed on your computer; default option is version most recently installed.

Command-Line Information**Parameter:** `tlmgCompilerSelect`**Type:** string**Value:****Default:** `Linux — GCC, Windows — Visual Studio 20XX`, where `XX` is the version most recently installed.**See Also**

“Select TLM Compilation Options” on page 22-12

User-defined tag for TLM component names

Add additional text to your TLM component class name identifier, the input and output data structures, and the directory to place the generated code.

Settings**No Default****Command-Line Information****Parameter:** tlmgUserTagForNaming**Type:** string**Value:****Default:****See Also**

“Select TLM Compilation Options” on page 22-12

UVM Component Generation

- “UVM Component Generation Overview” on page 26-2
- “Customize Generated UVM Code” on page 26-12
- “Use Tunable Parameters to Generalize UVM Simulation” on page 26-14
- “Tunable Parameters in Sequence Subsystem” on page 26-15
- “Tunable Parameters in Scoreboard Subsystem” on page 26-17

UVM Component Generation Overview

UVM Component Generation Overview

If you have a Simulink Coder license, you can generate a Universal Verification Methodology (UVM) test bench and additional components from a Simulink model. Generating UVM components enables a direct transition from your Simulink environment to a UVM framework.

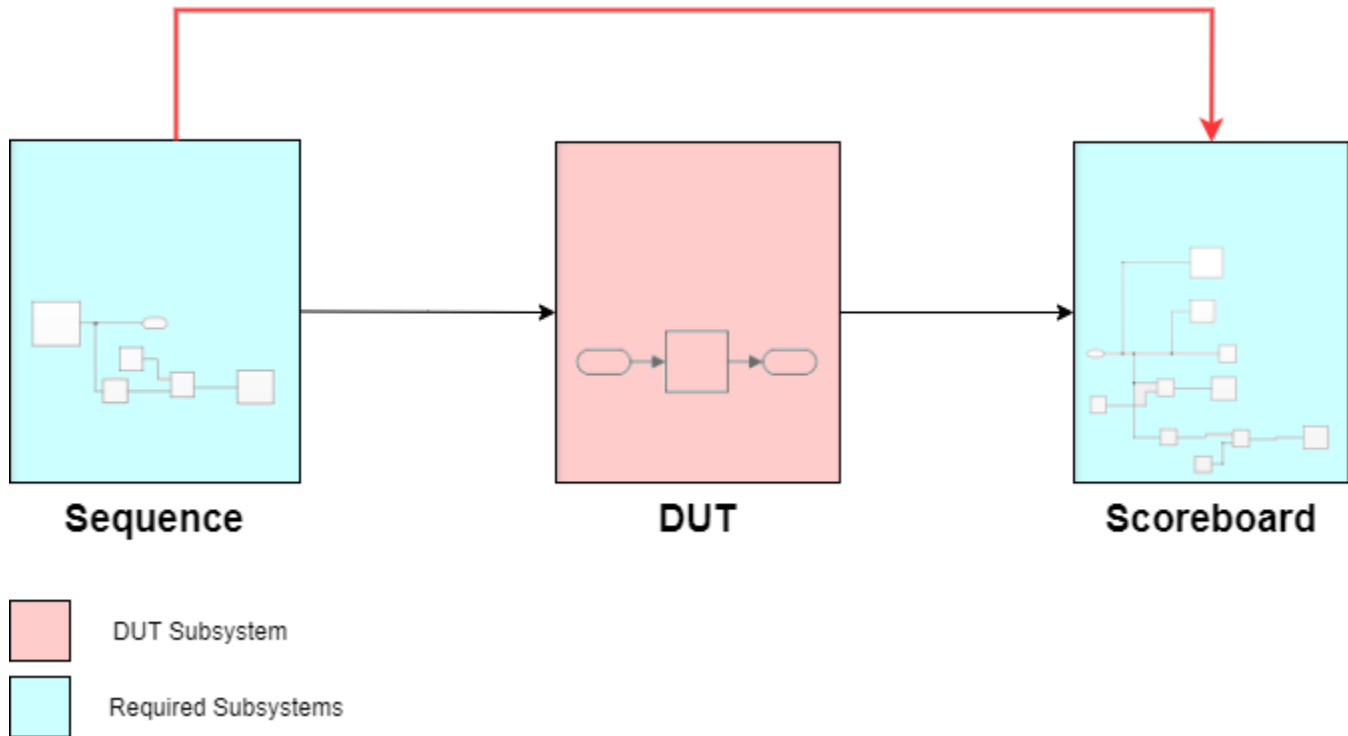
HDL Verifier exports Simulink subsystems as generated C code inside UVM components with a direct programming interface (DPI). You can integrate these generated components into your existing UVM environment. You can also use the generated UVM test bench to test an HDL DUT by replacing the generated behavioral DUT with your detailed HDL design.

Prepare Simulink Model for UVM Component Generation

Your Simulink model must include these subsystems.

- A DUT subsystem. This subsystem generates a SystemVerilog DPI (SVDPI) behavioral model of your DUT. For more information about SystemVerilog DPI Generation, see “DPI Component Generation with Simulink” on page 29-2.
- A sequence subsystem. This subsystem creates stimulus and drives it to the DUT.
- A scoreboard subsystem. This subsystem collects and checks the output of the DUT.
- The sequence can also drive signals directly to the scoreboard, as illustrated in red in the Simulink Model Structure figure.

For details on how to create a subsystem, see “Create Subsystems” (Simulink).



Simulink Model Structure

In addition to the previous structure, you can optionally include these subsystems.

- A driver subsystem
- A monitor subsystem
- A predictor subsystem

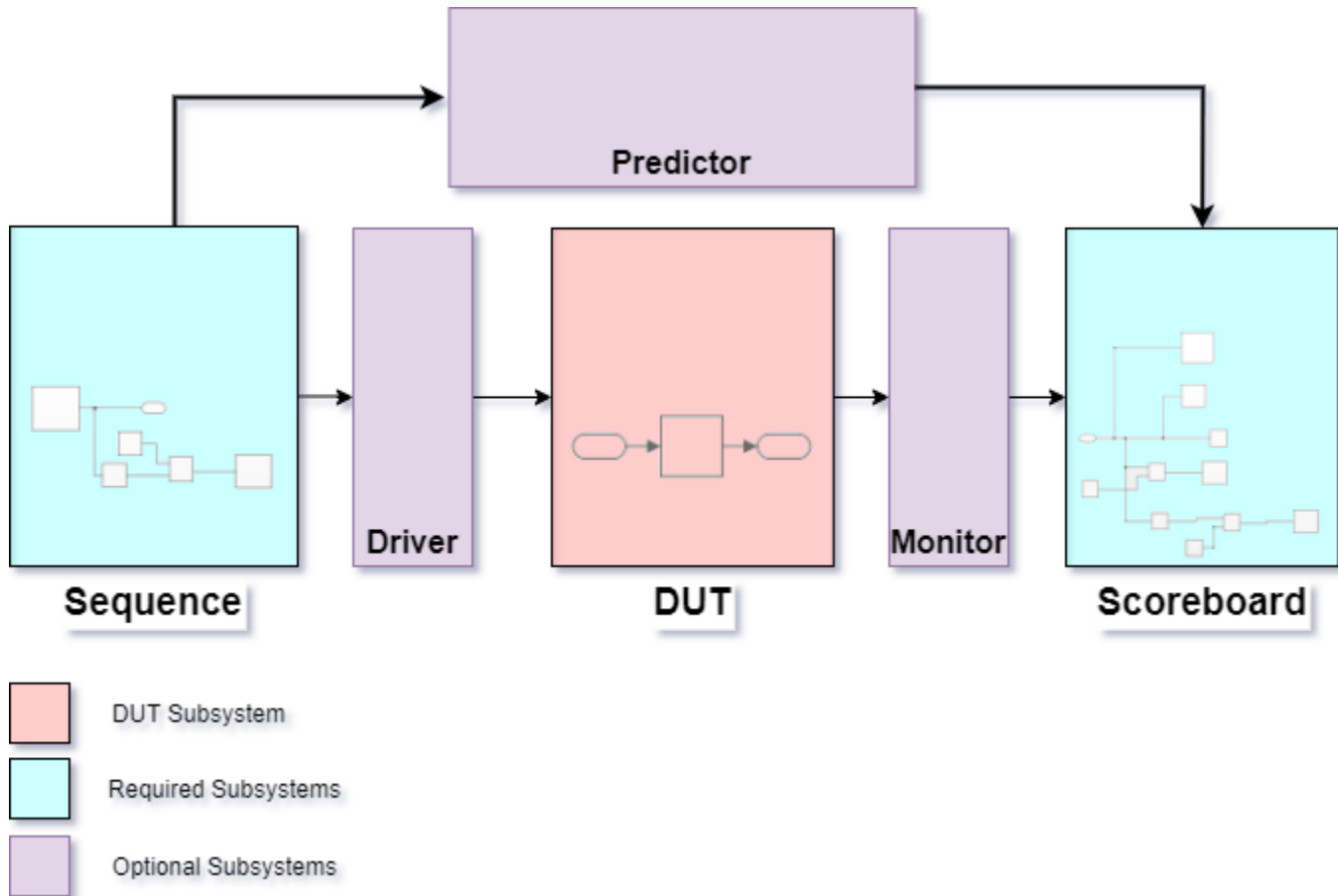
Note All the subsystems in your Simulink model must have names that start with a letter and use a combination of alphanumeric characters and underscores.

When adding a monitor, driver, or predictor subsystem, this feature supports these configurations.

- All signals coming out of the sequence must be connected to the driver, predictor, or the scoreboard. Other output signals are ignored for UVM generation.
- If your model includes a driver subsystem, then all signals coming out of the driver must be connected to the DUT.
- If your model includes a driver subsystem, then all input signals to the driver must originate in the sequence.
- If your model includes a monitor subsystem, then all signals coming out of the DUT must be connected to the monitor.
- If your model includes a monitor subsystem, then all signals coming out of the monitor must be connected to the scoreboard.
- If your model includes a predictor subsystem, then all input signals to the predictor must originate in the sequence. Additional inputs are ignored for UVM generation.

- If your model includes a predictor subsystem, then all output signals from the predictor must connect to the scoreboard. Additional outputs are ignored for UVM generation.

This image shows a Simulink model that includes a driver, a monitor, and a predictor subsystem.



Select System Target

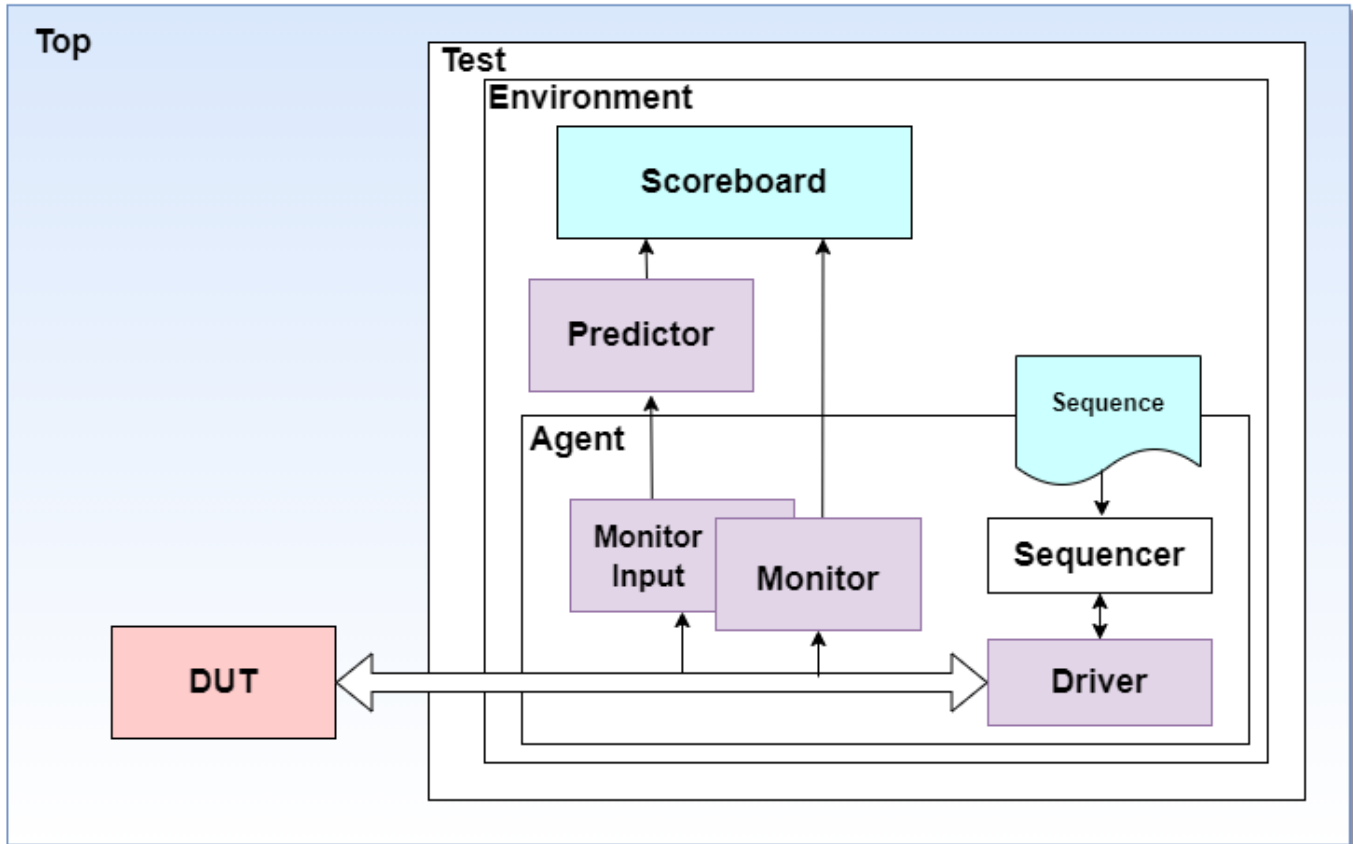
Because UVM generation utilizes the technology for generating SystemVerilog DPI, you must first select a supporting system target file. Open the configuration parameters dialog box, and select **Code Generation** from the left pane. For **System target file**, click **Browse**, and then select `systemverilog_dpi_grt.tlc` from the list.

Alternatively, if you have the Embedded Coder product, you can select target `systemverilog_dpi_ert.tlc`. This target enables you to access additional code generation options when you select **Code Generation** from the left pane of the Configuration Parameters dialog box.

For an example of UVM generation, see “Generate Parameterized UVM Test Bench from Simulink” on page 32-106.

Generated UVM Structure

Use the `uvmbuild` function to generate this structure of UVM components.



- **Top** - This module instantiates a generated behavioral DUT and the test environment. The top module has clock and reset signals that propagate into the design.
- **DUT** - a behavioral design-under-test module is generated from your Simulink DUT subsystem.
- **Test** - This module includes the UVM environment and sequence class.
- **Sequence** - This UVM object defines a set of transactions. The sequence object is generated from your Simulink sequence subsystem.
- **Environment** - This module includes the agent and generated scoreboard, and optionally a predictor.
- **Scoreboard** - The UVM scoreboard is generated from your Simulink scoreboard subsystem. The scoreboard compares expected results with output from the DUT.
- **Agent** - The UVM agent includes a sequencer, driver, and monitor. If a direct path exists from the Simulink sequence subsystem to the Simulink scoreboard subsystem, an additional monitor is included to monitor that signal.

- Sequencer - This module controls the flow of sequence transactions to the DUT.
- Driver - This module is generated from your Simulink driver subsystem and transforms each transaction from the Sequence to signals that the behavioral DUT understands.
- Monitor - This module is generated from your Simulink monitor subsystem and it samples the signals from the behavioral DUT and generates transactions that are sent to the UVM scoreboard.
- Monitor Input - This module is generated if you have a predictor or a direct connection from the sequence to the scoreboard subsystem. The monitor input samples the signals from the sequence and generates transactions that are sent to the UVM predictor or the scoreboard.
- Predictor - This module is generated from your Simulink predictor subsystem. The predictor represents a golden model of the DUT. It receives input from the sequence, calculates the results, and drives the results to the scoreboard to compare against the DUT results.

For more information about the UVM components and structure, see UVM reference guide.

Generated Files and Folder Structure

When generating UVM components, HDL Verifier generates SystemVerilog DPI (SV-DPI) components from your DUT, sequence, and scoreboard subsystems, as well as optional SV-DPI components for driver, monitor or predictor subsystems if your model includes them. The artifacts of DPI generation are placed in a directory named `uvm_build` in your working directory, that includes these two directories:

- `top_model_dpi_components` - This directory includes all generated DPI components.
- `top_model_uvm_testbench` - This directory includes the UVM testbench, the generated DUT, and shared libraries.

Where `top_model` is the name of your top-level Simulink model. You can change the default directory by setting the `buildDirectory` property in the `uvmcodegen.uvmconfig` configuration object.

The `top_model_dpi_components` directory contains directories for each one of the subsystems in the top model (DUT, sequence, scoreboard, driver, monitor, or predictor) named `subsystem_build`. Each directory includes:

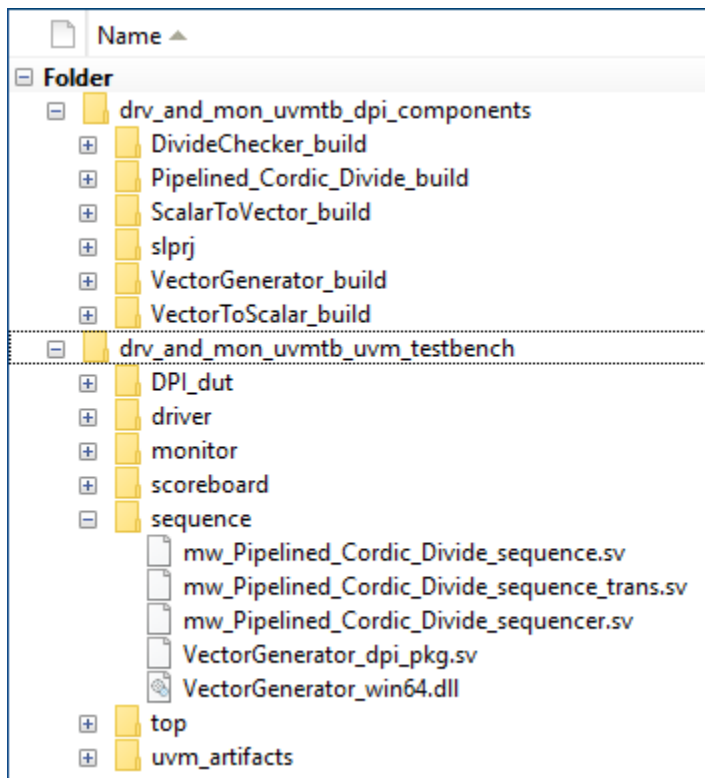
- `subsystem_dpi_pkg.sv` - SystemVerilog package file with function declarations for the component
- `subsystem_dpi.sv` - The generated SystemVerilog component
- DPI component and header files with extensions `.c` and `.h`
- Metadata and information files with extensions `.mat`, `.txt`, `.dmr`, `.tmw`, and `.def`
- A makefile for compiling the components into `.o` and `.so` files

The `top_model_uvm_testbench` directory includes several subfolders for the various generated UVM components:

- `DPI_dut` - This folder contains a copy of the SystemVerilog package, module files, and a `.dll` file from the `dut_build` folder.
- `driver` (optional) - This folder is generated if you specified a driver subsystem to the `uvmbuild` function. This folder contains a UVM driver, a copy of the SystemVerilog package, and a shared library file (dll-file or so-file) from the `driver_build` folder.

- `monitor` (optional) - This folder is generated if you specified a monitor subsystem to the `uvmbuild` function. This folder contains a UVM monitor, a copy of the SystemVerilog package, and a shared library file (dll-file or so-file) from the `monitor_build` folder.
- `predictor` (optional) - This folder is generated if you specified a predictor subsystem to the `uvmbuild` function. This folder contains a UVM predictor, a copy of the SystemVerilog package, a predictor transaction, and a shared library file (dll-file or so-file) from the `predictor_build` folder.
- `scoreboard` - This folder contains a copy of the SystemVerilog package and a .dll file from the `scoreboard_build` folder. This folder also includes the UVM scoreboard class, a scoreboard configuration object, and a scoreboard transaction that defines the input transaction type for the scoreboard.
- `sequence` - This folder contains a copy of the SystemVerilog package and a .dll file from the `sequence_build` folder. This folder also includes the UVM sequencer, the sequence class, and a sequence transaction that defines the transaction type from the sequencer to the driver.
- `top` - This folder contains the SystemVerilog package and module files for the top Simulink model. This folder also contains scripts for HDL-simulator execution.
- `uvm_artifacts` - This folder contains these SystemVerilog files.
 - `mw_DUT_agent.sv` - This file includes a UVM agent that instantiates sequence, driver, and monitor.
 - `mw_DUT_environment.sv` - This file includes a UVM environment, that instantiates an agent and a scoreboard.
 - `mw_DUT_if.sv` - This file defines the DUT SystemVerilog interface type. It contains DUT inputs and outputs, as well as ports for clock, reset, and clock-enable signals.
 - `mw_DUT_monitor_input.sv` - This file includes a pass-through UVM monitor. The monitor samples signals from the driver to the scoreboard or predictor.
 - `mw_DUT_test.sv` - This file includes a UVM test, that instantiates an environment and sequence. The test module starts the transactions by calling `seq.start`.
 - `mw_dpi_types_pkg.sv` - This file contains definitions of generated SystemVerilog types, such as `enum` and `struct`, exposed by UVM component interfaces. Only UVM components which use these types import this package.
 - `mw_DUT_driver.sv` - This file includes a pass-through UVM driver by default. When specifying a driver subsystem to the `uvmbuild` function, this module includes a scheduler and the API calls to the DPI component `Driver_dpi_pkg.sv`.
 - `mw_DUT_monitor.sv` - This file includes a pass-through UVM monitor. The monitor samples signals from the DUT to the scoreboard. When specifying a monitor subsystem to the `uvmbuild` function, this module includes a scheduler and the API calls to the DPI-component `Monitor_dpi_pkg.sv`.

This image shows the generated directory structure for a top level model named `drv_and_mon_uvmtb`.



Supported Simulink Data Types

Supported Simulink data types are converted to SystemVerilog data types, as shown in this table.

Generated SystemVerilog Types

MATLAB	SystemVerilog		
	Compatible C Type	Logic Vector	Bit Vector
uint8	byte unsigned	logic [7:0]	bit [7:0]
uint16	shortint unsigned	logic [15:0]	bit [15:0]
uint32	int unsigned	logic [31:0]	bit [31:0]
uint64	longint unsigned	logic [63:0]	bit [63:0]
int8	byte	logic signed [7:0]	bit signed [7:0]
int16	shortint	logic signed [15:0]	bit signed [15:0]
int32	int	logic signed [31:0]	bit signed [31:0]
int64	longint	logic signed [63:0]	bit signed [63:0]
boolean	byte unsigned	logic [0:0]	bit [0:0]
fixed-point	The port is sign extended to a built-in C type, such as int, int unsigned, byte, byte unsigned, etc.	logic [n-1:0] logic signed [n-1:0] The logic vector length (n) is equal to the wordlength. The sign is inherited from the fixed point type.	bit [n-1:0] bit signed [n-1:0] The bit vector length (n) is equal to the wordlength. The sign is inherited from the fixed point type.
single	shortreal		
double	real		
complex	You can choose between a SystemVerilog struct data type or flattened ports for real and imaginary parts in the SystemVerilog interface. To choose between these options, in the left pane of the Configuration Parameters dialog box, select Code Generation > SystemVerilog DPI , and then set the Composite data type parameter to structure or flattened.		

MATLAB	SystemVerilog		
	Compatible C Type	Logic Vector	Bit Vector
vectors, matrices	<p>You can choose between SystemVerilog arrays or scalar ports. To choose between these options, in the left pane of the Configuration Parameters dialog box, select Code Generation > SystemVerilog DPI, and then select the Scalarize matrix and vector ports parameter.</p> <p>For example, a two-element vector of type <code>uint32</code> in Simulink generates this SystemVerilog vector port:</p> <pre>input logic [31:0] vecInput [0:1]</pre> <p>When you select Scalarize matrix and vector ports, the generated SystemVerilog includes these two ports, each of type <code>logic [31:0]</code>:</p> <pre>input logic [31:0] vecInput_0, input logic [31:0] vecInput_1</pre> <p>When generating vector and array ports, the coder flattens matrices in column-major order.</p>		
nonvirtual bus	<p>You can choose between a SystemVerilog <code>struct</code> type or flattened ports for separate component signals in the SystemVerilog interface. To choose between these options, in the left pane of the Configuration Parameters dialog box, select Code Generation > SystemVerilog DPI section, and set Composite data type to <code>structure</code> or <code>flattened</code>.</p>		
enumerated data types	enum		

Limitations

- By default, HDL Verifier converts matrices and vectors to one-dimensional arrays in SystemVerilog. For example, a 4-by-2 matrix in Simulink is converted to a one-dimensional array of eight elements in SystemVerilog. To generate multiple scalar ports in the SystemVerilog interface, select **Scalarize matrix and vector ports** in the configuration parameters.
- The `uvmbuild` function ignores Simulink components that are not specified as a DUT, sequence, scoreboard, driver, monitor, or predictor subsystems.
- You can use feedback loops inside any of the subsystems, but not between them.
- The sequence, scoreboard, and predictor subsystems must operate at a single rate, and the fundamental sample times of their subsystems must be equal. For more information about sample times, see “Sample Times in Systems” (Simulink).
- The fundamental sample times of the driver, DUT, and monitor subsystems must be equal. Their ports can be multirate, but the greatest common divisor (GCD) or fundamental sample time must be the same.
- The sample time of the sequence, scoreboard, and predictor subsystems must be greater than or equal to the fundamental sample time of the driver, DUT, and monitor.

See Also

`uvmbuild`

More About

- “DPI Component Generation with Simulink” on page 29-2
- “Generate Parameterized UVM Test Bench from Simulink” on page 32-106

External Websites

- UVM Standard

Customize Generated UVM Code

Customize SystemVerilog File Banner

When using the `uvmbuild` function to generate a UVM test bench, the generated SystemVerilog files have a default file banner. The default banner includes the file location, the date and time that the file was created, and the MATLAB and HDL Verifier versions that the file was created with.

You can customize the generated SystemVerilog files with text or these optional tokens by inserting them inside comment statements in the banner.

- `%<Date>` - Date the file was generated (taken from computer clock)
- `%<FileName>` - Name of the generated file
- `%<FilePath>` - Full path to the location of the generated file
- `%<HDLV_Ver>` - HDL Verifier version that created the file
- `%<MATLAB_Ver>` - MATLAB version that created the file
- `%<ModelName>` - Name of the model
- `%<ModelVersion>` - A serial number, incremented by 1 each time you save the model
- `%<LastModifiedDate>` - Date when the model was last saved (from **Last saved on** field on the Model Properties dialog box)

Customize Banner in Subsystem Description

To customize a banner in SystemVerilog components that are mapped from a Simulink subsystem (for example, sequence or scoreboard subsystems), right-click the Simulink subsystem and select **Properties**. In the Properties dialog box, click the **General** tab and, in the **Description** pane, enter your custom text. For example:

```
This is my custom banner
%<Date>
%<HDLV_Ver>
```

For this example, after executing the `uvmbuild` function, the generated SystemVerilog presents this banner:

```
//This is my custom banner
//2020-05-08 15:06:16
//HDL Verifier 6.1
```

Alternatively, you can set the **Description** parameter by using the `set_param` function. For example, to set a custom banner for a scoreboard subsystem named `scr`, execute the following commands:

```
custom_banner = sprintf('This is my scoreboard\nFile path: %<FilePath>');
set_param(scr,'Description',custom_banner);
uvmbuild(dut,seq,scr,'Driver',drv,'Monitor',mon);
```

Customize Banner in Top Model

To customize a banner in the top-level model or in SystemVerilog components that are not mapped to a Simulink subsystem (for example, agent or environment), set the **Description** parameter in the top Simulink model. On the Simulink toolstrip, click the **Modeling** tab, and then click **Model Explorer**. In the **Model Properties** pane on the right, click the **Description** tab and enter your custom banner text. In the generated SystemVerilog files, this text appears in the top-model files and the

SystemVerilog files in the `uvm_testbench/uvm_artifacts` folder, which are not mapped to a specific subsystem in Simulink.

Alternatively, you can set the top-model **Description** parameter by using the `set_param` function. For example, to set a custom banner for a top model named `top`, execute the following:

```
set_param('top','Description','This is a top level comment')
uvmbuild(dut,seq,scr,'Driver',drv,'Monitor',mon);
```

Customize HDL Simulation Timescale

By default, when the `uvmbuild` function generates a UVM test bench, the HDL simulation timescale is configured to ``timescale 1ns/1ns`. You can customize the timescale to a different value by creating a `uvmcodegen.uvmconfig` configuration object, and then using that UVM configuration object in the `uvmbuild` function, as in this example:

```
cfg = uvmcodegen.uvmconfig('timescale','1ps/1ps');
uvmbuild(dut,seq,scr,'Config',cfg);
```

See Also

`uvmbuild` | `uvmcodegen.uvmconfig`

More About

- “UVM Component Generation Overview” on page 26-2

Use Tunable Parameters to Generalize UVM Simulation

Universal Verification Methodology (UVM) supports tunable parameters in generated SystemVerilog components in several ways.

- Create a SystemVerilog parameter using the DPI component tunable parameter methodology. For more information about using tunable parameters in DPI components, see “Tune Gain Parameter During Simulation” on page 30-12.
- Parameterize a sequence subsystem to create constrained random stimulus. For more information about tunable parameters in a sequence subsystem, see “Tunable Parameters in Sequence Subsystem” on page 26-15.
- Generalize the scoreboard for different scenario checks without the need to regenerate the UVM files. For more information about tunable parameters in a scoreboard subsystem, see “Tunable Parameters in Scoreboard Subsystem” on page 26-17.

To learn more about UVM component generation, see “UVM Component Generation Overview” on page 26-2.

See Also

`uvmbuild`

More About

- “UVM Component Generation Overview” on page 26-2
- “Tune Gain Parameter During Simulation” on page 30-12
- “Tunable Parameters in Sequence Subsystem” on page 26-15
- “Tunable Parameters in Scoreboard Subsystem” on page 26-17

Tunable Parameters in Sequence Subsystem

When a Simulink sequence subsystem includes tunable parameters, the `uvmbuild` function generates a sequence object that contains a SystemVerilog parameter for each tunable parameter. These parameters can be randomized, and their minimum, maximum, and default values are derived from the Simulink parameter.

Prepare Sequence for UVM Generation With Tunable Parameters

To prepare the sequence subsystem for Universal Verification Methodology (UVM) generation with tunable parameters, take these steps.

- 1 Set up your Simulink model for DPI and UVM generation. On the **Modeling** tab in Simulink, click **Model Settings**. In the Configuration Parameters dialog box, select **Code Generation** on the left pane. Then, set **System target file** to `systemverilog_dpi_grt.tlc`. If you have the Embedded Coder product you can alternatively set this value to `systemverilog_dpi_ert.tlc`.
- 2 Create a data object for your parameter by using the `Simulink.Parameter` object. For example, to create a parameter named `dataValue`, enter this code at the MATLAB command prompt.

```
dataValue = Simulink.Parameter
open dataValue
```

Define properties for the `dataValue` parameter. In the `Simulink.Parameter` window, set these values.

- **Value** - Set the default value for the generated UVM parameter.
- **Minimum** - Set the minimum value for generated UVM parameter.
- **Maximum** - Set the maximum value for generated UVM parameter.
- **Data type** - Set the data type for the generated UVM parameter. For the sequence subsystem, note the difference between using integer and floating point data types for constrained random parameters.
- **Storage class** - Select `Model default`, `SimulinkGlobal`, or `ExportedGlobal`.

Use `Model default` when your parameter is instance-specific. Use either `SimulinkGlobal` or `ExportedGlobal` to generate a global variable. Setting **Storage class** to `Auto` optimizes the parameter during code generation, and is not recommended.

- 3 Use the parameter you created in the sequence subsystem of your Simulink model. For example, you can parameterize a Constant block to generate random values within a given range by setting its `Constant value` parameter to the Simulink parameter you previously defined.

Generate UVM Sequence

Use the `uvmbuild` function to generate a UVM test bench. In addition to the files regularly generated by `uvmbuild`, the function adds components to the file `top_model_name_uvmbuild/uvmbuild/sequence/mw_DUT_sequence.sv`.

- When you use integer data types, the sequence contains two added SystemVerilog random constraint code-blocks.

- Default value
- range

The values for these constraints are derived from the value you set in the Simulink parameter.

- When you use floating point data types, the sequence contains these two additional functions.
 - `pre_randomize` - This function sets the default value for the tunable parameter.
 - `post_randomize` - This function checks that the value set for the parameter is within the range defined in the Simulink parameter.

When using integer data types, these functions are empty.

For more information about the files generated by the `uvmbuild` function, see “Generated Files and Folder Structure” on page 26-6.

Control Sequence Parameters in UVM Simulation

After generating UVM components for your system, you can control stimulus generation in one of these two ways.

If a tunable parameter is of integer data type, you can turn on or off the default value and range constraints. You can also add and set additional SystemVerilog constraints.

If a tunable parameter is of floating point data type, you can:

- Write your own randomize function by using the `pre_randomize` function.
- Change the default value of the parameter by extending the `pre_randomize` function.

The `post_randomize` function checks that the floating point tunable parameter is within the range defined in the Simulink model. If the **Minimum** and **Maximum** properties of the Simulink parameter are empty, this function is empty.

See Also

`uvmbuild`

More About

- “Tunable Parameters in Scoreboard Subsystem” on page 26-17
- “UVM Component Generation Overview” on page 26-2

Tunable Parameters in Scoreboard Subsystem

When a Simulink scoreboard subsystem includes tunable parameters, the `uvmbuild` function generates a scoreboard configuration object that contains a SystemVerilog parameter for each tunable parameter. This feature enables you to generalize the scoreboard components, promoting reuse of the same SystemVerilog scoreboard across different scenarios by changing the parameter value as a command line argument.

Prepare Scoreboard for UVM Generation with Tunable Parameters

To prepare the scoreboard subsystem for Universal Verification Methodology (UVM) generation with tunable parameters, take these steps.

- 1 Set up your Simulink model for DPI and UVM generation. On the **Modeling** tab in Simulink, click **Model Settings**. In the Configuration Parameters dialog box, select **Code Generation** on the left pane. Then, set **System target file** to `systemverilog_dpi_grt.tlc`. If you have the Embedded Coder product you can alternatively set this value to `systemverilog_dpi_ert.tlc`.
- 2 Create a data object for your parameter by using the `Simulink.Parameter` object. For example, to create a parameter named `dataValue`, enter this code at the MATLAB command prompt.

```
dataValue = Simulink.Parameter
open dataValue
```

Define properties for the `dataValue` parameter. In the `Simulink.Parameter` window, set these values.

- **Value** - Set the default value for the generated UVM parameter.
- **Data type** - Set the data type for the generated UVM parameter. For the sequence subsystem, note the difference between using integer and floating point data types for constrained random parameters.
- **Storage class** - Select `Model default`, `SimulinkGlobal`, or `ExportedGlobal`.

Use `Model default` when your parameter is instance-specific. Use either `SimulinkGlobal` or `ExportedGlobal` to generate a global variable. Setting **Storage class** to `Auto` optimizes the parameter during code generation, and is not recommended.

- 3 Use the parameter you created in the scoreboard subsystem of your Simulink model. Control the parameter value from the command line to test different scenarios.

Generate UVM Scoreboard

Use the `uvmbuild` function to generate a UVM test bench. In addition to the files regularly generated by `uvmbuild`, the function adds these generated files.

- `top_model_name_uvmbuild/uvmbuild/uvmbuild/uvmbuild/uvmbuild/uvmbuild/mw_DUT_scoreboard_cfg_obj.sv` - This file contains a configuration object for the scoreboard. It defines the default value for the tunable parameter as the value you set in the Simulink parameter.
- `top_model_name_uvmbuild/uvmbuild/uvmbuild/uvmbuild/uvmbuild/uvmbuild/mw_DUT_scoreboard.sv` - The function `start_of_simulation_phase` sets the configuration object.

SystemVerilog DPI Component Generation

DPI Component Generation for MATLAB Function

- “Considerations for DPI Component Generation with MATLAB” on page 27-2
- “Use Variable-Sized Vector in SystemVerilog DPI Component” on page 27-7

Considerations for DPI Component Generation with MATLAB

You can export a MATLAB function as a component with a direct programming interface (DPI) for use in a SystemVerilog simulation. HDL Verifier wraps generated C code with a DPI wrapper that communicates with a SystemVerilog thin interface function in a SystemVerilog simulation.

For MATLAB, you generate the component using the `dpigen` function.

Note You must have a MATLAB Coder license to use this feature.

Supported MATLAB Data Types

Supported MATLAB data types are converted to SystemVerilog data types, as shown in this table. When using the `dpigen` function, use the `PortsDataType` property to select `Compatible C type`, `Logic vector`, or `Bit vector` data types.

Generated SystemVerilog Types

MATLAB	SystemVerilog		
	Compatible C Type	Logic Vector	Bit Vector
uint8	byte unsigned	logic [7:0]	bit [7:0]
uint16	shortint unsigned	logic [15:0]	bit [15:0]
uint32	int unsigned	logic [31:0]	bit [31:0]
uint64	longint unsigned	logic [63:0]	bit [63:0]
int8	byte	logic signed [7:0]	bit signed [7:0]
int16	shortint	logic signed [15:0]	bit signed [15:0]
int32	int	logic signed [31:0]	bit signed [31:0]
int64	longint	logic signed [63:0]	bit signed [63:0]
logical	byte unsigned	logic [0:0]	bit [0:0]
fi (fixed-point data type)	Depends on the fixed-point word length. If the fixed-point word length is greater than the host word size (for example, 64-bit vs. 32-bit), then this data type cannot be converted to a SystemVerilog data type by MATLAB Coder and you will get an error. If the fixed-point word length is less than or equal to the host word size, MATLAB Coder converts the fixed-point data type to a built-in C type.	logic [n-1:0] logic signed [n-1:0] The logic vector length (n) is equal to the wordlength. The sign is inherited from the fixed point type.	bit [n-1:0] bit signed [n-1:0] The bit vector length (n) is equal to the wordlength. The sign is inherited from the fixed point type.
single	shortreal		
double	real		
complex	The coder flattens complex signals into real and imaginary parts in the SystemVerilog component.		
vectors, matrices	arrays For example, a 4-by-2 matrix in MATLAB is converted into a one-dimensional array of eight elements in SystemVerilog. By default, the coder flattens matrices in column-major order. To change to row-major order, use the <code>-rowmajor</code> option with the <code>dpigen</code> function. For additional information, see “Generate Code That Uses Row-Major Array Layout” (MATLAB Coder).		

MATLAB	SystemVerilog		
	Compatible C Type	Logic Vector	Bit Vector
structure	The coder flattens structure elements into separate ports in the SystemVerilog component.		
enumerated data types	enum		

Generated Shared Library

Function `dpigen` automatically compiles the shared library needed to run the exported DPI component in the SystemVerilog environment. The makefile that builds the shared library has the extension `_rtw.mk`. For example, for `fun.m`, the make file name is `fun_rtw.mk`.

During compilation, the function `dpigen` generates a library file.

- Windows 64: `function_win64.dll`
- Linux: `function.so`

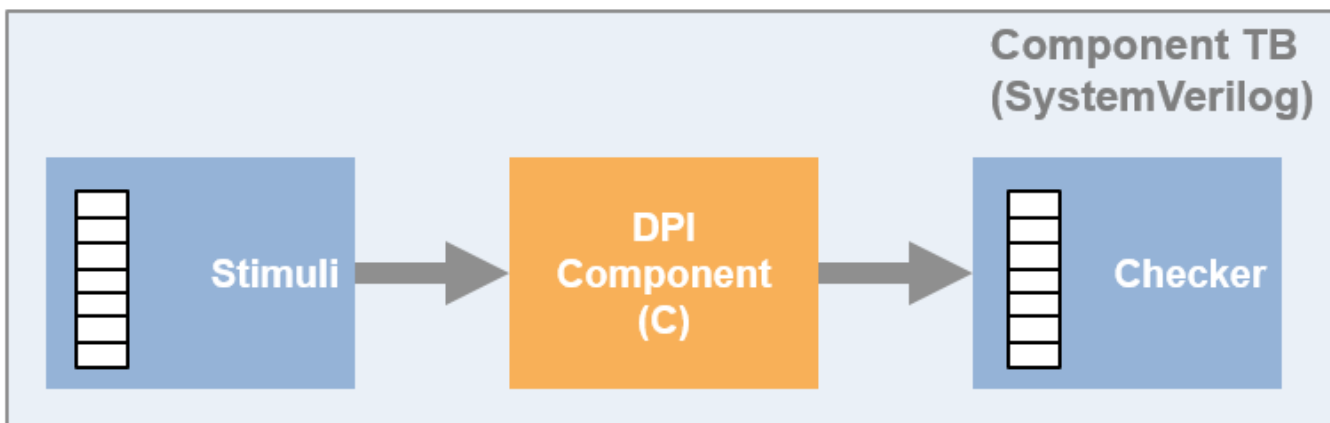
`function` is the name of the MATLAB function you generated the DPI component from.

Note If you use 64-bit MATLAB on Windows, you get a 64-bit DLL, which can be used only with a 64-bit HDL simulator.

Make sure that your MATLAB version matches your HDL simulator version.

Generated Test Bench

Function `dpigen` also creates a test bench. You can use this test bench to verify that the generated SystemVerilog component is functionally equivalent to the original MATLAB function. The generator runs your MATLAB code to save input and output data vectors for use in the test bench. This test bench is not intended as a replacement for a system test bench for your own application. However, you can use the generated test bench as a starting example when creating your own system test bench.



Generated Outputs

- C and header files from your algorithm, generated by MATLAB Coder
- C and header files for the DPI wrapper, generated by HDL Verifier
- SystemVerilog file that exposes the component and adds control signals
- SystemVerilog package file that contains all the function declarations of the DPI component
- SystemVerilog test bench (with the `-testbench` option)
- Data files used with the HDL simulator (with the `-testbench` option)
- HDL simulator scripts, such as `*.do` or `*.sh` (with the `-testbench` option)
- Makefile `*.mk`

Generated SystemVerilog Wrapper

Generated Control Signals

SystemVerilog code generated for a sequential function by function `dpigen` contains a set of control signals.

- `clk`: synchronization clock
- `clk_enable`: clock enable
- `reset`: asynchronous reset

When generating SystemVerilog for a combinational design, the generated code does not have the above control signals. Specify sequential or combinational by using the `ComponentTemplateType` argument of the `dpigen` function.

Generated Initialize Function

All SystemVerilog code generated by the `dpigen` function includes an `Initialize` function. The `Initialize` function is called at the beginning of the simulation.

For example:

```
import "DPI" function void DPI_Subsystem_initialize();
```

If the asynchronous reset signal is high (goes from 0 to 1), `Initialize` is called again.

Simulation Considerations

When simulating the DPI component in your HDL environment, `reset`, `clock`, and `clk_enable` behave as follows:

- When `clk_enable` is 0, the DPI output function is not executed, and the output values are not updated.
- When `clk_enable` is 1 and `reset` is 0, the DPI output function is executed on the positive edge of the clock signal.
- When `reset` is 1, the internal state of the DPI component is set to its initial value. This action is equivalent to using the `clear` function in MATLAB to update persistent variables. Then output values then reflect the DPI component initial state and current input values. For more details on persistent variables, see “Persistent Variables”.

Limitations

- Large fixed-point numbers that exceed the system word length are not supported.
- Some optimizations, such as constant folding, are not supported because they change the interface of the generated C function. For more information, see “MATLAB Coder Optimizations in Generated Code” (MATLAB Coder).
- HDL Verifier limits matrices and vectors to one-dimensional arrays in SystemVerilog. For example, a 4-by-2 matrix in MATLAB is converted to a one-dimensional array of eight elements in SystemVerilog.
- The PostCodegen callback in configuration objects is not supported.

See Also

Related Examples

- “Create MATLAB Function and Test Bench” on page 28-2
- “Generate SystemVerilog DPI Component” on page 28-3
- “Run Generated Test Bench in HDL Simulator” on page 28-7
- “Use Generated DPI Functions in SystemVerilog” on page 28-9

Use Variable-Sized Vector in SystemVerilog DPI Component

This example shows how to configure, generate, and use a SystemVerilog DPI (SVDPI) component with variable-length inputs or outputs.

Design Task

When you generate a SVDPI from a MATLAB® function that includes a variable-sized vector input or output, the result is a SystemVerilog module with a variable-sized input or output. Follow this example to configure, generate, and use the component in a SystemVerilog environment.

MATLAB Function

This example uses the function `varSizeVectorSupport`, a sinus function where the size of the input `vec` is variable. The output of the function is derived from the input, and therefore the output size is also variable.

Testbench

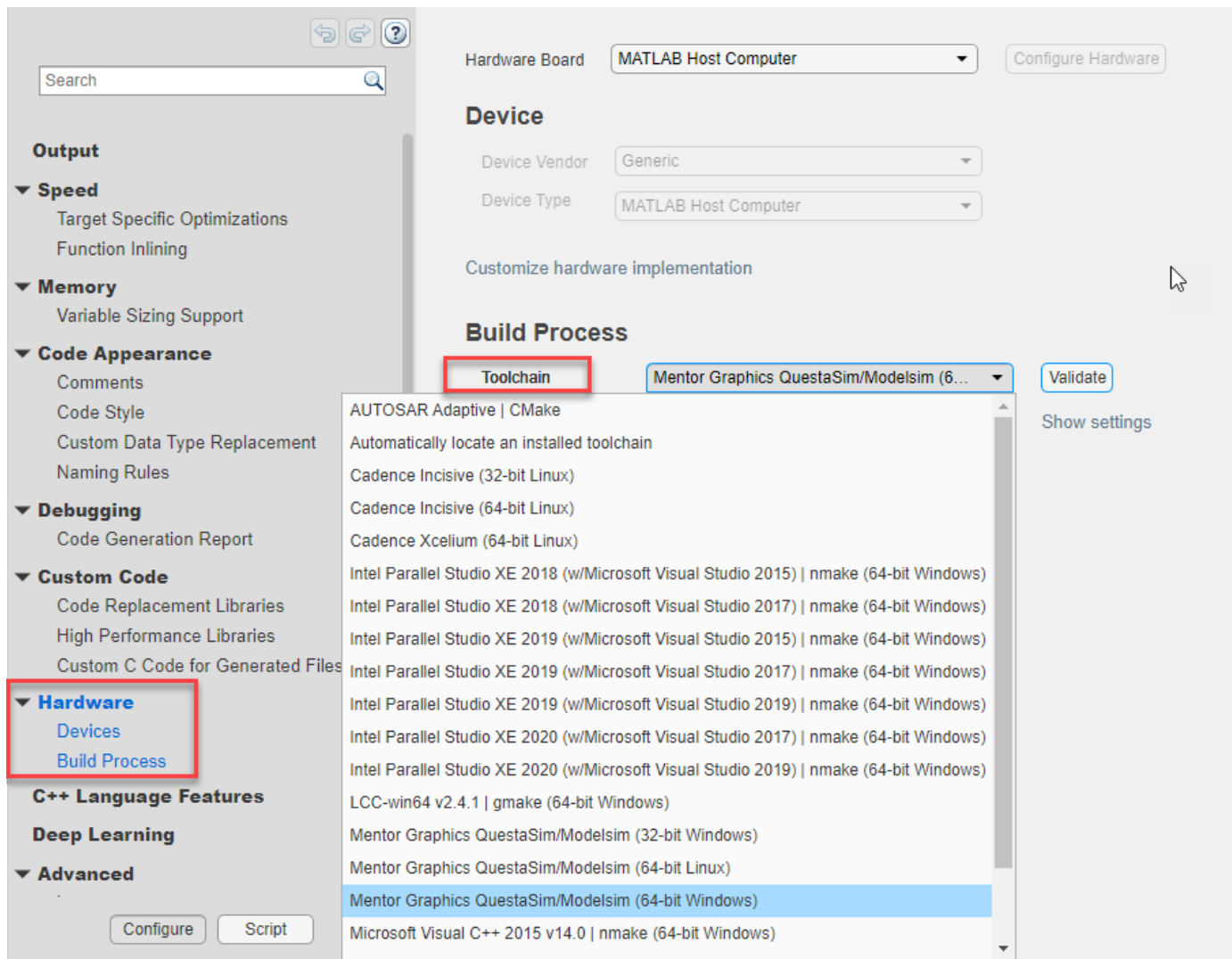
Use the provided `varSizeVectorSupport_tb` testbench to stimulate the function with fixed-sized signals. Then, generate a SystemVerilog testbench to test the generated SystemVerilog module. You can later change the port types of the generated SystemVerilog testbench (or write a new testbench) without the need to regenerate a DPI module from the `varSizeVectorSupport` function.

Configure Generation Options

To specify a supported toolchain, use a `coder.config` (MATLAB Coder) configuration object. Set the `build_type` to `'dll'`.

```
configObj = coder.config('dll');
```

Then, set the toolchain to a supported simulator. Double-click the generated object in the MATLAB workspace to open the configuration object dialog box. Select **Hardware** on the left pane, and then under **Build Process** set **Toolchain** to a supported simulator (Cadence® Xcelium™, Mentor Graphics® Questa® or ModelSim®).



Alternatively, you can set the toolchain at the MATLAB command prompt. For example, this code sets the toolchain to Mentor Graphics QuestaSim/Modelsim (64-bit Windows).

```
configObj.Toolchain = 'Mentor Graphics QuestaSim/Modelsim (64-bit Windows)';
```

Generate SystemVerilog DPI Component

To generate the SystemVerilog module, use the `dpigen` function.

- You must pass the configuration object to the `dpigen` function using the `-config` argument.
- Use the `-args` argument to specify the size of the generated ports. To set an upper bound value to the variable-length vector, specify that value in the `coder.Type` object. For example, to set an upper bound of 20 for `vec`, enter this code at the MATLAB command prompt.

```
dpigen varSizeVectorSupport -args {coder.typeof(1,[1 20],[0 1]),1,1} -config configObj
```

- Use `inf` to specify that the port size is of variable-length and unbounded in the generated SystemVerilog.

- To generate a SystemVerilog testbench from a MATLAB testbench function, use the `-testbench` argument.

```
dpigen varSizeVectorSupport -testbench varSizeVectorSupport_tb ...
    -args {coder.typeof(1,[1 inf],[0 1]),1,1} -config config0bj
```

Generated Interface

Because `vec` is specified as a variable-sized vector during code generation in this example, the generated SystemVerilog includes the variable-sized input `vec` and output `y`. The data type for output `y` is derived from the input data type. These variable-sized ports are declared as a SystemVerilog open array (`[]`). This code shows the generated interface for the `varSizeVectorSupport` function.

```
module varSizeVectorSupport_dpi(
    input bit clk,
    input bit clk_enable,
    input bit reset,
    input real vec [],
    input real amp,
    input real freq,
    output shortint y []
);
```

Simulate in ModelSim

Verify that ModelSim is on the system path. Navigate to the directory named `codegen\dll\varSizeVectorSupport\dpi_tb`. To run the testbench and verify the generated component in ModelSim, enter this command at the MATLAB command prompt.

```
!vsim < run_tb_mq.do
```

Limitations

This feature does not support:

- Variable-sized matrices (convert matrices to a variable-sized vector if needed)
- Structure data-types with fields of variable size
- Variable-sized arrays of structures
- Cross-platform DPI component generation
- Not supported on Synopsys® VCS® simulator

See Also

`coder.config` | `coder.typeof` | `dpigen`

Related Examples

- “Considerations for DPI Component Generation with MATLAB” on page 27-2
- “Use Generated DPI Functions in SystemVerilog” on page 30-9

DPI Component Generation (MATLAB)

Generate DPI Component Using MATLAB

In this section...

“Create MATLAB Function and Test Bench” on page 28-2
 “Generate SystemVerilog DPI Component” on page 28-3
 “Run Generated Test Bench in HDL Simulator” on page 28-7
 “Use Generated DPI Functions in SystemVerilog” on page 28-9
 “Port Generated Component and Test Bench to Linux” on page 28-10

Create MATLAB Function and Test Bench

- “Create MATLAB Function” on page 28-2
- “Create Test Bench” on page 28-2
- “Run Test Bench” on page 28-3

Create MATLAB Function

Code the MATLAB function you want to export to a SystemVerilog environment. For information about coding MATLAB functions, see “Function Basics” in the MATLAB documentation.

Consider adding the compilation directive `%#codegen` to your function. This directive can help you diagnose and fix violations that would result in errors during code generation. See “Compilation Directive `%#codegen`” (MATLAB Coder).

While you code your function, keep in mind the “Limitations” on page 29-11, which describe the various aspects of DPI component generation that you must know. These aspects include which data types are valid, what files are generated, and how the shared libraries are compiled.

In this example, the MATLAB function `fun.m` takes a single input and multiplies it by 2. The function includes the compilation directive `%#codegen`.

```
function y = fun(x)
%#codegen
y = x * 2;
```

The process of creating the MATLAB includes writing the code, creating the test bench, and running the test bench in an iterative process. When you are satisfied that your function does what you intend it to do, continue on to “Generate SystemVerilog DPI Component” on page 28-3.

Create Test Bench

Create a test bench to exercise the function. In this example, the test bench applies a test vector against `fun.m` and plots the output.

```
function sample=fun_tb
% Testbench should not require input, however you can give an output.

% Define a test vector
tVecIn = [1,2,3,4,5];

% Exercise fun.m and plot results to make sure function is working correctly
tVecOut = arrayfun(@(in) fun(in),tVecIn);
plot(tVecIn,tVecOut);
grid on;
```

```
% Get my sample input to use it with function dpigen.
sample = tVecIn(1);
```

Note that a test bench should not have inputs. The test bench can load test vectors using MAT files or any other data file, so it does not require inputs.

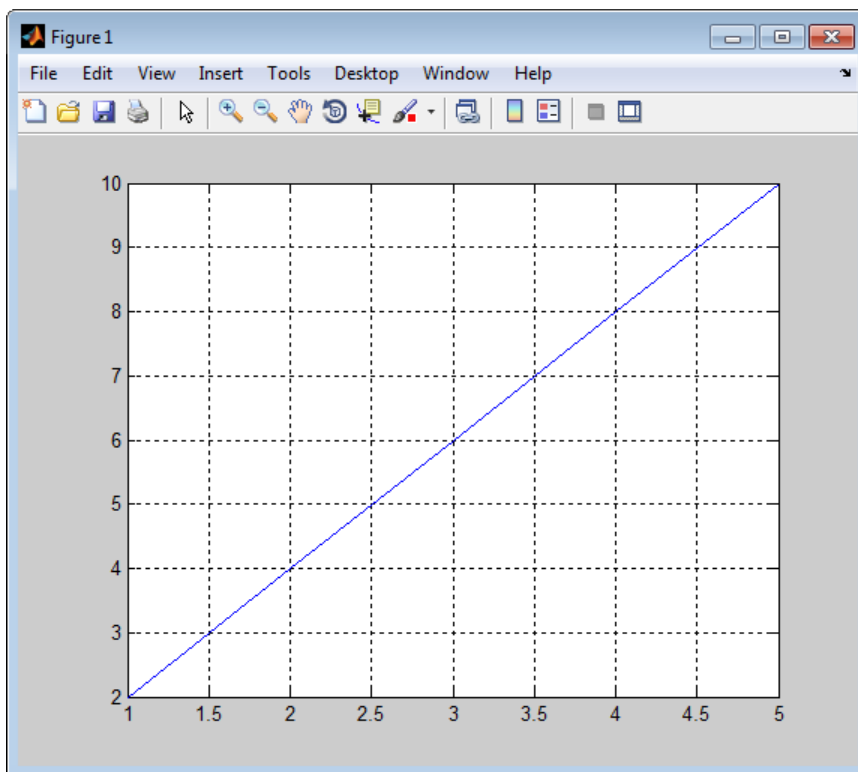
The output of `fun_tb`, `sample`, is going to be used as the function inputs argument for `fun.m` during the call to `dpigen`, which is why it is a single element. See “Generate SystemVerilog DPI Component” on page 28-3.

Run Test Bench

```
fun_tb
```

```
ans =
```

```
1
```



Next, generate the SystemVerilog DPI component. See “Generate SystemVerilog DPI Component” on page 28-3.

Generate SystemVerilog DPI Component

- “Generate DPI Component with `dpigen` Function” on page 28-4
- “Examine Generated Package File” on page 28-5
- “Examine Generated Component” on page 28-5
- “Examine Generated Test Bench” on page 28-6

Generate DPI Component with dpigen Function

Use the function `dpigen` to generate the DPI component. This function has several optional input arguments. At a minimum, specify the MATLAB function you want to generate a component for and the function inputs. If you also want to generate a test bench to exercise the generated component, use the `-testbench` option.

```
dpigen func -args input_arg -testbench test_bench_name
```

- 1 Define the inputs as required by the function. In this example, `sample` is a scalar value of type `double`.

```
sample = 1;
```

- 2 Call the DPI component generator function:

```
dpigen fun -args sample -testbench fun_tb
```

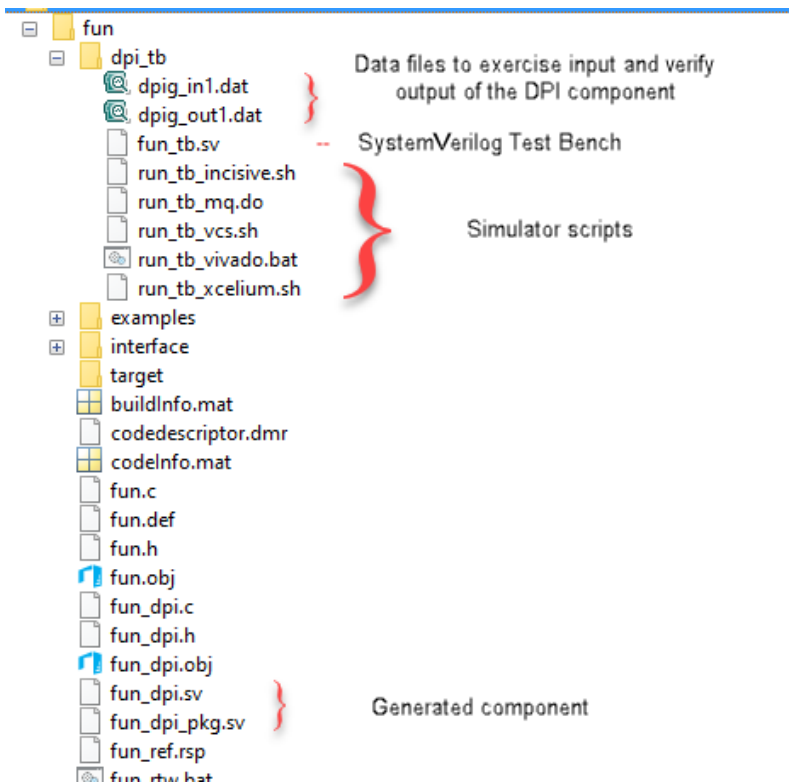
The command, issued as shown, performs the following tasks:

- Generates `fun_dpi.sv` - a SystemVerilog component for the function `fun.m`. The function inputs for `fun.m` are specified in `sample`.
- Generates `fun_dpi_pkg.sv` - a SystemVerilog package file. This file contains all the imported function declarations.
- Creates a test bench for the generated component.

For this call to `dpigen`, MATLAB outputs the following messages:

```
### Generating DPI Wrapper fun_dpi.c
### Generating DPI Wrapper header file fun_dpi.h
### Generating SystemVerilog module package fun_dpi_pkg.sv
### Generating SystemVerilog module fun_dpi.sv
### Generating makefiles for: fun_dpi
### Compiling the DPI Component
### Generating SystemVerilog test bench fun_tb.sv
### Generating test bench simulation script for Mentor Graphics QuestaSim/Modelsim run_tb_mq.do
### Generating test bench simulation script for Cadence Incisive run_tb_incisive.sh
### Generating test bench simulation script for Cadence Xcelium run_tb_xcelium.sh
### Generating test bench simulation script for Synopsys VCS run_tb_vcs.sh
### Generating test bench simulation script for Vivado Simulator run_tb_vivado.bat
```

The function shown in the previous example generates the following folders and files:



Examine Generated Package File

Examine the generated package file. Note the declarations of the `initialize`, `reset`, `terminate`, and `fun` functions.

This example shows the code generated for `fun_dpi_pkg.sv`.

```
// File: C:\fun_example\codegen\dll\fun\fun_dpi_pkg.sv
// Created: 2017-12-19 09:18:00
// Generated by MATLAB 9.5 and HDL Verifier 5.4

`timescale 1ns / 1ns
package fun_dpi_pkg;

// Declare imported C functions
import "DPI" function chandle DPI_fun_initialize(input chandle existhandle);
import "DPI" function chandle DPI_fun_reset(input chandle objhandle,input real x,output real y);
import "DPI" function void DPI_fun(input chandle objhandle,input real x,output real y);

import "DPI" function void DPI_fun_terminate(input chandle existhandle);

endpackage : fun_dpi_pkg
```

Examine Generated Component

Examine the generated component so that you can understand how the `dpigen` function converted MATLAB code to SystemVerilog code. For more information on what the function includes, see “Generated SystemVerilog Wrapper” on page 29-5.

This example shows the code generated for `fun_dpi.sv`.

```
// File: C:\fun_example\codegen\dll\fun\fun_dpi.sv
// Created: 2017-12-19 09:18:00
// Generated by MATLAB 9.5 and HDL Verifier 5.4
```

```

`timescale 1ns / 1ns
import fun_dpi_pkg::*;

module fun_dpi(
    input bit clk,
    input bit clk_enable,
    input bit reset,
    input real x,
    output real y
);

   chandle objhandle=null;
    real y_temp;

    initial begin
        objhandle=DPI_fun_initialize(objhandle);
    end

    final begin
        DPI_fun_terminate(objhandle);
    end

    always @(posedge clk or posedge reset) begin
        if(reset== 1'b1) begin
            objhandle=DPI_fun_reset(objhandle,x,y_temp);
            y<=y_temp;
        end
        else if(clk_enable) begin
            DPI_fun(objhandle,x,y_temp);
            y<=y_temp;
        end
    end
end
endmodule

```

Examine Generated Test Bench

Examine the generated test bench so you can see how function `dpigen` created this test bench from the MATLAB code. For more information on the generated test bench, see “Generated Test Bench” on page 27-4.

This example shows the code generated for `fun_tb.sv`.

```

// File: C:\fun_example\codegen\dll\fun\dpi_tb\fun_tb.sv
// Created: 2017-12-19 09:18:13
// Generated by MATLAB 9.5 and HDL Verifier 5.4

`timescale 1ns / 1ns
module fun_tb;
    real x;
    real y_ref;
    real y_read;
    real y;
    // File Handles
    integer fid_x;
    integer fid_y;
    // Other test bench variables
    bit clk;
    bit clk_enable;
    bit reset;
    integer fscanf_status;
    reg testFailure;
    reg tbDone;
    bit[63:0] real_bit64;
    bit[31:0] shortreal_bit64;
    parameter CLOCK_PERIOD= 10;
    parameter CLOCK_HOLD= 2;
    parameter RESET_LEN= 2*CLOCK_PERIOD+CLOCK_HOLD;
    // Initialize variables
    initial begin
        clk = 1;
        clk_enable = 0;
        testFailure = 0;
        tbDone = 0;
        reset = 1;
        fid_x = $fopen("dpig_in1.dat","r");
        fid_y = $fopen("dpig_out1.dat","r");
        // Initialize multirate counters

```

```

        #RESET_LEN reset = 0;
    end
    // Clock
    always #(CLOCK_PERIOD/2) clk = ~ clk;
    always@(posedge clk) begin
        if (reset == 0) begin
            #CLOCK_HOLD
            clk_enable <= 1;
            fscanf_status = $fscanf(fid_x, "%h", real_bit64);
            x = $bitstoreal(real_bit64);
            if ($feof(fid_x))
                tbDone = 1;
            fscanf_status = $fscanf(fid_y, "%h", real_bit64);
            y_read = $bitstoreal(real_bit64);
            if ($feof(fid_y))
                tbDone = 1;
            y_ref <= y_read;
            if (clk_enable == 1) begin
                assert ( ((y_ref - y) < 2.22045e-16) && ((y_ref - y) > -2.22045e-16) ) else begin
                    testFailure = 1;
                    $display("ERROR in output y_ref at time %0t :", $time);
                    $display("Expected %e; Actual %e; Difference %e", y_ref, y, y_ref-y);
                end
                if (tbDone == 1) begin
                    if (testFailure == 0)
                        $display("*****TEST COMPLETED (PASSED)*****");
                    else
                        $display("*****TEST COMPLETED (FAILED)*****");
                    $finish;
                end
            end
        end
    end
end

// Instantiate DUT
fun_dpi u_fun_dpi(
    .clk(clk),
    .clk_enable(clk_enable),
    .reset(reset),
    .x(x),
    .y(y)
);
endmodule

```

Next, run the generated test bench in the HDL simulator. See “Run Generated Test Bench in HDL Simulator” on page 28-7. If you plan to port the component and optional test bench from Windows to Linux, see “Port Generated Component and Test Bench to Linux” on page 28-10.

Run Generated Test Bench in HDL Simulator

- “Run Test Bench in ModelSim and QuestaSim Simulators” on page 28-7
- “Run Test Bench in Xcelium Simulator” on page 28-8
- “Run Test Bench in VCS Simulator” on page 28-9

This section includes instructions for running the generated test bench in one of the supported HDL simulators: Mentor Graphics ModelSim and QuestaSim, Cadence Xcelium, and Synopsys VCS®. It is possible that this code will work in other (unsupported) HDL simulators but it is not guaranteed.

Choose the workflow for your HDL simulator.

Run Test Bench in ModelSim and QuestaSim Simulators

- 1 Start ModelSim or QuestaSim in GUI mode.
- 2 Change your current directory to the `dpi_tb` folder under the code generation directory in MATLAB.
- 3 Enter the following command in the shell to start your simulation:

```
do run_tb_mq.do
```

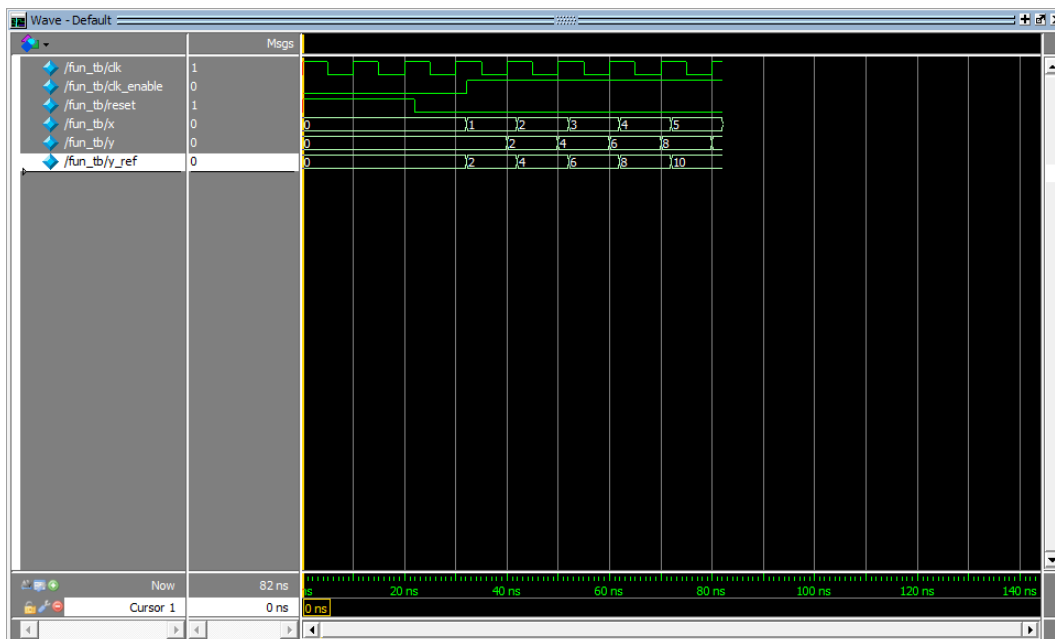
This generated script contains the name of the component and test bench, and instructions to the HDL simulator for running the test bench.

When the simulation finishes, you should see the following text displayed in your console:

```
*****TEST COMPLETED (PASSED)*****
```

This message tells you that the test bench was run against the generated component successfully.

The following wave form image from this example demonstrates that the generated test bench was successfully exercised in the HDL simulator.



Next, import your component. See “Use Generated DPI Functions in SystemVerilog” on page 28-9.

Run Test Bench in Xcelium Simulator

- 1 Launch Xcelium.
- 2 Start your terminal shell.
- 3 Change the current directory to `dpi_tb` under the code generation directory in MATLAB.
- 4 Enter the following command in the shell to start the simulation:

```
sh run_tb_ncsim.sh
```

This generated script contains the name of the component and test bench, and instructions to the HDL simulator for running the test bench.

When the simulation finishes, you should see the following text displayed in your console:

```
*****TEST COMPLETED (PASSED)*****
```

This message tells you that the test bench was run against the generated component successfully.

Run Test Bench in VCS Simulator

- 1 Launch VCS.
- 2 Start your terminal shell.
- 3 Change the current directory to `dpi_tb` under the code generation directory in MATLAB.
- 4 Enter the following command in your shell to start the simulation:

```
sh run_tb_vcs.sh
```

This generated script contains the name of the component and test bench, and instructions to the HDL simulator for running the test bench.

When the simulation finishes, you should see the following text displayed in your console:

```
*****TEST COMPLETED (PASSED)*****
```

This message tells you that the test bench was run against the generated component successfully.

Use Generated DPI Functions in SystemVerilog

To use the generated DPI component in a SystemVerilog test bench, first you must include the package file in your SystemVerilog environment. This will have the DPI functions available within the scope of your SystemVerilog module. Then, you must call the generated functions. When you compile the SystemVerilog code that contains the imported generated functions, use a DPI-aware SystemVerilog compiler and specify the component and package file names along with the SystemVerilog code.

The following example demonstrates adding the generated DPI component for `fun.m` to a SystemVerilog module.

- 1 Call the `Initialize` function.


```
DPI_fun_initialize();
```
- 2 Call the function generated from `fun.m`.


```
DPI_fun(x,y);
```

You can now modify the generated code as needed.

Example

```
module test_twofun_tb;

    initial begin
        DPI_fun_initialize();
    end

    always@(posedge clk) begin
        #1
        DPI_fun(x,y);
    end
end
```

Port Generated Component and Test Bench to Linux

To port the component and optional test bench from a Windows operating system to a Linux operating system, follow one of these workflows based on your HDL simulator.

Note You must have an Embedded Coder license for porting your component from Windows to Linux.

- “Generate Generic DPI Component” on page 28-10
- “Generate Simulator-Specific DPI Component” on page 28-11

Generate Generic DPI Component

Follow this workflow for all of the supported HDL simulators. The software supports the Mentor Graphics ModelSim, Mentor Graphics QuestaSim, Cadence Xcelium, and Synopsys VCS simulators. You generate the DPI component generic to all the supported HDL simulators.

Tasks on Windows Host Machine

- 1 Create a `coder.config` object. Change the target HW device type to LP64 for the Linux operating system.

```
cfg=coder.config('dll');
cfg.HardwareImplementation.TargetHWDeviceType='Generic->64-bit Embedded Processor (LP64)';
```

- 2 Run the `dpigen` function using the option `-config` to use the `config` object that you created in step 1. Use the option `-c` so that the `dpigen` function generates only code.

```
dpigen -config cfg DataTypes.m -args InputSample -c
```

- 3 Generate a zip archive to port to Linux, navigate to the source folder that contains the `buildInfo` file, and execute these commands at the MATLAB command prompt:

```
load buildInfo
packNGo(buildInfo)
```

- 4 Navigate to the top-level folder. Find the ZIP archive that you generate in step 3, which has the same name as the MATLAB function. Copy the ZIP archive to the Linux machine.

Tasks on Linux Target Machine

- 1 Unzip the file using the `-j` option to extract all the files with a flattened folder structure. You can unzip the contents into any folder.

```
unzip -j DataTypes.zip
```

- 2 **a** Copy this generic makefile script into an empty file:

```
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)

SHARE_LIB_NAME=DPI_Component.so

all: $(SRC) $(SHARE_LIB_NAME)
    @echo "### Successfully generated all binary outputs."

$(SHARE_LIB_NAME): $(OBJ)
    gcc -shared -lm $(OBJ) -o $@

.c.o:
    gcc -c -fPIC -Wall -pedantic -Wno-long-long -fwrapv -O0 $< -o $@
```

- b** Replace `DPI_Component.so` with the name of the shared library you want to create.
- c** Save the script as `Porting_DPIC.mk` in the folder that contains the zip files.

- 3 Build the shared library.

```
make -f Porting_DPIC.mk all
```

To use the generated component with SystemVerilog, see “Use Generated DPI Functions in SystemVerilog” on page 28-9.

- 4 (Optional) Run the test bench that the software generated in Windows.

- a Copy the contents of the `dpi_tb` folder from the Windows host machine to the Linux target machine.
- b Run the test bench.

To run the test bench in an HDL simulator, see “Run Generated Test Bench in HDL Simulator” on page 28-7.

Generate Simulator-Specific DPI Component

Follow this workflow for the Mentor Graphics ModelSim, Mentor Graphics QuestaSim, and Cadence Xcelium HDL simulators. You generate the DPI component for your chosen simulator.

Use this workflow for easier porting steps. If you use this workflow, you do not need to create a generic makefile script and build the shared library on the Linux target machine to use the generated component. If you generate a test bench to exercise the generated component, you do not need to copy the test bench folder from the Windows host machine to the Linux target machine separately.

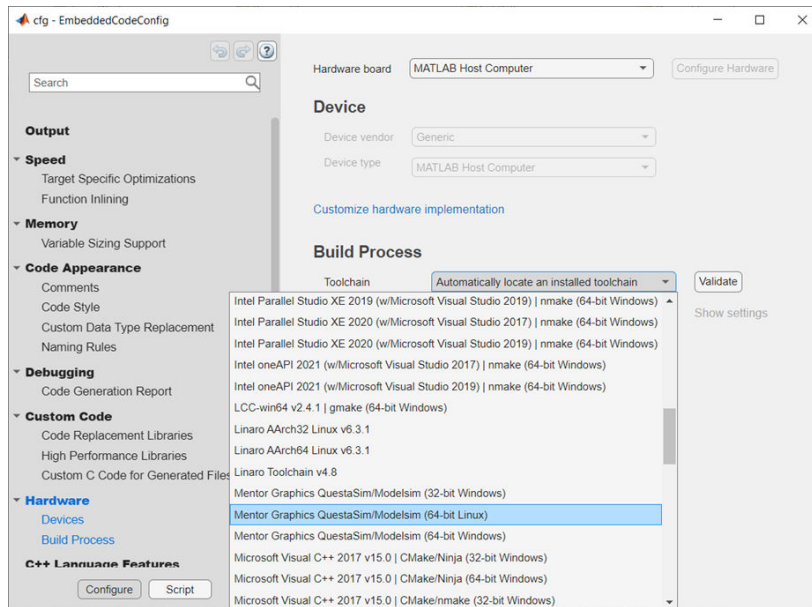
Tasks on Windows Host Machine

- 1 Create a `coder.config` object.

```
cfg=coder.config('dll');
```

- 2 Select the target simulator and operating system by configuring the `coder.config` object that you created in step 1. In the MATLAB workspace, double-click `cfg` to open the **EmbeddedCodeConfig** dialog for `cfg`. On the **Hardware** pane, under **Build Process**, select a target **Toolchain**. This option specifies the target simulator and operating system where you run simulations. The supported cross-product toolchains are:

- Cadence Xcelium (64-bit Linux)
- Mentor Graphics ModelSim/QuestaSim (64-bit Linux)



- 3 Run the `dpigen` function using the `-config` option to use the `config` object. Use the `-testbench` option if you also want to generate a test bench to exercise the generated component. Use the `-c` option so that the `dpigen` function generates only code.

```
dpigen -config cfg DataTypes.m -args InputSample -testbench DataTypes_tb.m -c
```

- 4
 - a Generate a ZIP archive to port to Linux, navigate to the source folder, which contains the `buildInfo` file.

```
load buildInfo
```

- b Use the `packNGo` function to package the generated files and any required dependencies before copying them to the target machine. Specify the argument `'minimalHeaders'`, `false` to include all the header files on the include path in the ZIP archive.

```
packNGo(buildInfo, 'minimalHeaders', false)
```

- 5 Navigate to the top-level folder. Find the ZIP archive that you generate in step 3, which has the same name as the MATLAB function. Copy the ZIP archive to the Linux machine. The ZIP archive contains the ModelSim file with the `.do` extension or the Xcelium file with the `.sh` extension.

Tasks on Linux Target Machine

- 1 Unzip the file into a folder of your choice.
- 2
 - a Start the HDL simulator.
 - b Navigate to the folder that contains the generated files that you unzipped in step 1.
 - For the ModelSim and QuestaSim simulators, use the transcript window to navigate.
 - For the Xcelium simulator, use the terminal shell to navigate.
 - c Build the project and run the test bench in the HDL simulator.
 - For the ModelSim and QuestaSim simulators, run these commands.

```
do DataTypes.do
do run_tb_mq.do
```


- For the Xcelium simulator, run these commands.

```
sh DataTypes.sh  
sh run_tb_xcelium.sh
```


DPI Component Generation for Simulink Subsystem

- “DPI Component Generation with Simulink” on page 29-2
- “SystemVerilog DPI Test Benches” on page 29-12

DPI Component Generation with Simulink

In this section...

“DPI Generation Overview” on page 29-2
 “Supported Simulink Data Types” on page 29-2
 “Generated SystemVerilog Wrapper” on page 29-5
 “SystemVerilog Wrapper for Combinational Design” on page 29-6
 “Generated Component Functions” on page 29-7
 “Parameter Tuning” on page 29-8
 “Test Point Access Functions” on page 29-9
 “Extra Sample Delay” on page 29-9
 “Multirate System Behavior” on page 29-9
 “Customization” on page 29-10
 “Limitations” on page 29-11

DPI Generation Overview

If you have a Simulink Coder license, you can generate SystemVerilog DPI components using one of two methods.

Export SystemVerilog DPI Component for Subsystem

HDL Verifier integrates with Simulink Coder to export a subsystem as generated C code inside a SystemVerilog component with a direct programming interface (DPI). You can integrate this component into your HDL simulation as a behavioral model. The coder provides options to customize the generated SystemVerilog structure. The component generator supports test point access and tunable parameters. The coder optionally generates a SystemVerilog test bench that verifies the generated DPI component against data vectors from your Simulink subsystem. This feature is available in the Model Configuration Parameters dialog box, under **Code Generation**. See “Generate SystemVerilog DPI Component” on page 30-2.

Generate SystemVerilog Test Bench in HDL Coder

From HDL Coder, you can generate a SystemVerilog DPI test bench. Use the test bench to verify your generated HDL code using C code generated from your entire Simulink model, including the DUT and data sources. To use this feature, your entire model must support C code generation with Simulink Coder. You can access this feature in HDL Workflow Advisor under **HDL Code Generation > Set Testbench Options**, or in the Model Configuration Parameters dialog box, under **HDL Code Generation > Test Bench**. Alternatively, for command-line access, set the `GenerateSVDPIITestBench` property of `makehdl.tb`. See “Verify HDL Design Using SystemVerilog DPI Test Bench” (HDL Coder).

Supported Simulink Data Types

Supported Simulink data types are converted to SystemVerilog data types, as shown in this table.

You can choose a bit vector, logic vector, or a compatible C type. Choose in Configuration Parameters dialog box, in the **Code Generation > SystemVerilog DPI** section, under **SystemVerilog Ports > Ports data type**.

Generated SystemVerilog Types

MATLAB	SystemVerilog		
	Compatible C Type	Logic Vector	Bit Vector
uint8	byte unsigned	logic [7:0]	bit [7:0]
uint16	shortint unsigned	logic [15:0]	bit [15:0]
uint32	int unsigned	logic [31:0]	bit [31:0]
uint64	longint unsigned	logic [63:0]	bit [63:0]
int8	byte	logic signed [7:0]	bit signed [7:0]
int16	shortint	logic signed [15:0]	bit signed [15:0]
int32	int	logic signed [31:0]	bit signed [31:0]
int64	longint	logic signed [63:0]	bit signed [63:0]
boolean	byte unsigned	logic [0:0]	bit [0:0]
fixed-point	The port is sign extended to a built-in C type, such as int, int unsigned, byte, byte unsigned, etc.	logic [n-1:0] logic signed [n-1:0] The logic vector length (n) is equal to the wordlength. The sign is inherited from the fixed point type.	bit [n-1:0] bit signed [n-1:0] The bit vector length (n) is equal to the wordlength. The sign is inherited from the fixed point type.
single	shortreal		
double	real		
complex	You can choose between a SystemVerilog struct data type or flattened ports for real and imaginary parts in the SystemVerilog interface. To choose between these options, in the left pane of the Configuration Parameters dialog box, select Code Generation > SystemVerilog DPI , and then set the Composite data type parameter to structure or flattened.		

MATLAB	SystemVerilog		
	Compatible C Type	Logic Vector	Bit Vector
vectors, matrices	<p>You can choose between SystemVerilog arrays or scalar ports. To choose between these options, in the left pane of the Configuration Parameters dialog box, select Code Generation > SystemVerilog DPI, and then select the Scalarize matrix and vector ports parameter.</p> <p>For example, a two-element vector of type <code>uint32</code> in Simulink generates this SystemVerilog vector port:</p> <pre>input logic [31:0] vecInput [0:1]</pre> <p>When you select Scalarize matrix and vector ports, the generated SystemVerilog includes these two ports, each of type <code>logic [31:0]</code>:</p> <pre>input logic [31:0] vecInput_0, input logic [31:0] vecInput_1</pre> <p>When generating vector and array ports, the coder flattens matrices in column-major order.</p>		
nonvirtual bus	<p>You can choose between a SystemVerilog <code>struct</code> type or flattened ports for separate component signals in the SystemVerilog interface. To choose between these options, in the left pane of the Configuration Parameters dialog box, select Code Generation > SystemVerilog DPI section, and set Composite data type to <code>structure</code> or <code>flattened</code>.</p>		
enumerated data types	enum		

Generated SystemVerilog Wrapper

- “Generated Control Signals” on page 29-5
- “Generated SystemVerilog Module Interface” on page 29-5

Generated Control Signals

All sequential SystemVerilog code generated by the SystemVerilog DPI generator contains these control signals:

- `clk` - synchronization clock
- `clk_enable` - clock enable
- `reset` - asynchronous reset

When you generate a SystemVerilog wrapper for a combinational model, the wrapper does not include these ports.

Generated SystemVerilog Module Interface

Choose between a port-list, or an interface declaration. Set this option in the Configuration Parameters, under **Code Generation > SystemVerilog DPI > SystemVerilog ports > Connection**.

- `Port list` - generates a SystemVerilog module with a port list in the header, representing its interface.

For example:

```
module MyMod_dpi(
    input bit clk,
    input bit clk_enable,
    input bit reset,
    /* Simulink signal name: 'in1' */
    input real in1 ,
    /* Simulink signal name: 'out1' */
    output real out1
);

...
endmodule
```

- **Interface** - generates a SystemVerilog module with an interface name in the header, and a separate declaration of the interface.

For example:

```
interface simple_if;
    bit clk;
    bit clk_enable;
    bit reset;
    /* Simulink signal name: 'in1' */
    real in1 ;
    /* Simulink signal name: 'out1' */
    real out1 ;
endinterface

module MyMod_dpi(
    simple_if vif
);
...
endmodule
```

Command-Line Alternative: Use the `set_param` function and set the `DPIPortConnection` parameter to either 'Interface' or 'Port List'.

For example:

```
set_param(bdroot, 'DPIPortConnection','Interface')
```

SystemVerilog Wrapper for Combinational Design

You can generate a SystemVerilog wrapper for a combinational model by selecting **Code Generation > SystemVerilog DPI** on the left pane, and then under **Component Template** set **Component template type** to **Combinational**.

When generating a combinational SystemVerilog wrapper, the interface does not include `clk`, `clk_enable`, or `reset` ports.

For example:

```
module MyMod_dpi(
    /* Simulink signal name: 'in1' */
    input real in1 ,
```



```

    /* Simulink signal name: 'out1' */
    output real out1
);

...
endmodule

```

Command-Line Alternative: Use the `set_param` function and set the `DPIComponentTemplateType` parameter to either `'Sequential'` or `Combinational'`.

Combinational SystemVerilog Limitations

- Select a combinational template only if your Simulink model is purely combinational.
- If your Simulink model is sequential, you must select the sequential component template type. A sequential model will not work correctly with a combinational template.
- If your model includes a delay block, it is considered a sequential design.
- A model which is partially sequential and partially combinational is not supported for SystemVerilog DPI generation.

Generated Component Functions

SystemVerilog code generated by the SystemVerilog DPI generator contains these functions:

```

// Declare imported C functions
import "DPI" functionchandle DPI_subsystemname_initialize(chandle existhandle);
import "DPI" function void DPI_subsystemname_output(input chandle objhandle,
    input real In1, inout real Out1);
import "DPI" function void DPI_subsystemname_terminate(input chandle objhandle);

```

And for sequential designs, the code also includes the following functions:

```

import "DPI" functionchandle DPI_subsystemname_reset(input chandle objhandle,
    input real In1, inout real Out1);
import "DPI" function void DPI_subsystemname_update(input chandle objhandle,
    input real In1);

```

Here, *subsystemname* is the name of the subsystem you generated code for.

If your model also contains tunable parameters, see “Parameter Tuning” on page 29-8.

- Initialize function — The `Initialize` function is called at the beginning of the simulation.

For example, for a subsystem titled `dut`:

```

initial begin
    objhandle = DPI_dut_initialize(objhandle);
end

```

- Reset function — Call the `reset` function when you would like to reset the simulation to a known reset state.

For example, for a subsystem titled `dut`:

```

initial begin
    objhandle = DPI_dut_reset(objhandle, 0, 0);
end

```

- Output function — At the positive edge of clock, if `clk_enable` is high, the output function is called first, followed by the update function.

For example, for a subsystem titled `dut`:

```
if(clk_enable) begin
    DPI_dut_output(objhandle, dut_In1, dut_Out1);
    DPI_dut_update(objhandle, dut_In1);
end
```

- Update function

At the positive edge of clock, if `clk_enable` is high, the update function is called after the output function.

For example, for a subsystem titled `dut`:

```
if(clk_enable) begin
    DPI_dut_output(objhandle, dut_In1, dut_Out1);
    DPI_dut_update(objhandle, dut_In1);
end
```

- Terminate function

Set specific conditions for early termination of simulation.

For example, for a subsystem titled `dut`:

```
if (condition for termination) begin
    DPI_dut_terminate(objhandle);
end
```

The function details in the SystemVerilog code generated from your system vary. You can examine the generated code for specifics. For an example of the generated functions in context, see “Get Started with SystemVerilog DPI Component Generation” on page 32-68.

Parameter Tuning

You can run different simulations with various values for the parameters in your Simulink model. If your system has tunable parameters, the generated SystemVerilog code also contains a Set Parameter function for each tunable parameter.

The DPI component generator generates a Set Parameter function for each tunable parameter in the format `DPI_subsystemname_setparam_tunableparametername`.

In this example, the tunable gain parameter has its own `setparam_gain` function.

```
import "DPI" function void DPI_dut_setparam_gain(input chandle objhandle, input real dut_P_gain);
```

The generated SystemVerilog code does not call this function. Instead, the default parameters are used. To change those parameters during simulation, explicitly call the specific `setparam` function. For example, in the subsystem titled `dut`, you can change the gain during simulation to a value of 6 by inserting the following call:

```
DPI_dut_setparam_gain(objhandle, 6);
```

To make a parameter tunable, create a data object from your subsystem before generating the SystemVerilog code. See “Tune Gain Parameter During Simulation” on page 30-12.

Test Point Access Functions

This feature enables you to access internal signals of the SystemVerilog DPI component in your HDL simulator. You can designate internal signals in your model as test points and configure the SystemVerilog DPI generator to create individual or grouped access functions.

You can also enable logging on test points. With logging enabled, you can use the generated test bench to compare logged data from Simulink with values observed while running the SystemVerilog component.

See “SystemVerilog DPI Component Test Point Access” on page 30-10 and “Get Started with SystemVerilog DPI Component Generation” on page 32-68.

Extra Sample Delay

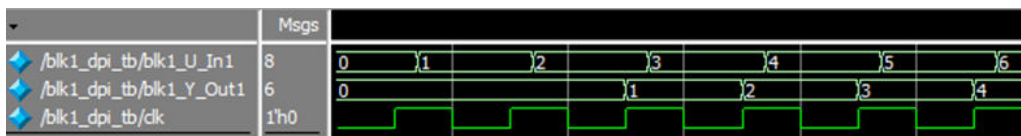
Compared with the original Simulink model, the generated SystemVerilog module introduces one extra sample delay at the output. For example, in the following Simulink model, the output is one-sample delayed version of the input signal.



The generated C code preserves this behavior, and the output comprises a one-sample delayed version of the input signal. However, in the SystemVerilog wrapper file, the clock signal is used to synchronize the input and output signals:

```
always @(posedge clk) begin
    DPI_blk2_output(blk2_In1, blk2_Out1);
    DPI_blk2_update();
end
```

The output of the SystemVerilog module can only be updated on the rising edge of the clock. This requirement introduces an extra sample delay.



Multirate System Behavior

By default, Simulink subsystems have a fundamental sample time variable (*FundST*) that indicates when, during simulation, the subsystem produces outputs and updates its internal state. With multirate systems, you can specify different sample times for different ports. For additional information, see “What Is Sample Time?” (Simulink).

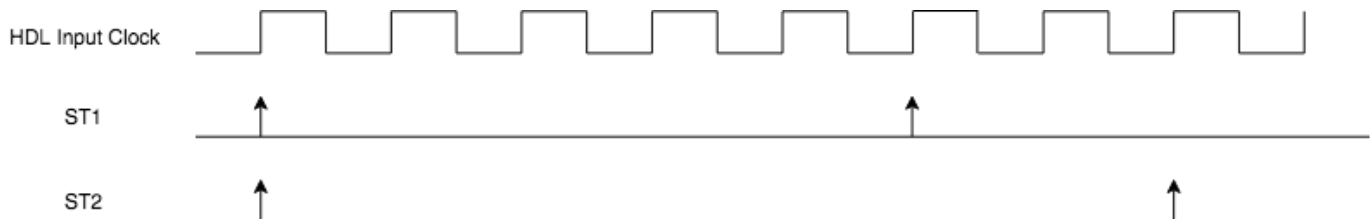
When a multirate subsystem generates a DPI component, the DPI component runs at a sample time that is equal to the greatest common divisor of all sample times in the subsystem.

For example, assume a subsystem has a fundamental sample time of 0.01 (that is, *FundST* is 0.01) and sample times *ST1* and *ST2* of 0.5 and 0.7, respectively.

$ST1=0.5$, $ST2=0.7$. The DPI component runs at a sample time of 0.1 (because 0.1 is the greatest common divisor of 0.5 and 0.7). To successfully acquire a signal with a sample time of 0.5 ($ST1$) or 0.7 ($ST2$), the HDL clock signal must toggle five or seven times, respectively.

When a DPI component is generated from the top level, the component executes at the fundamental sample time.

This diagram shows the relationship between the HDL input clock and sample times $ST1$, and $ST2$.



Customization

You can customize the generated SystemVerilog wrapper by modifying the template included with HDL Verifier (`svdpi_grt_template.vgt`). Alternatively, you can create your own custom template. Provide anchors for the generated code in your template to verify that the template generates valid SystemVerilog code.

The default SystemVerilog template, provided by HDL Verifier, is `svdpi_grt_template.vgt`. In this template, special `clken_in` and `clken_out` control signals are added to the SystemVerilog module interface.

You can generate SystemVerilog DPI components from multiple subsystems and connect them together in an HDL simulator. When you do so, these control signals determine the execution order of those components. They also minimize the delay between the Simulink signal and the SystemVerilog signal.

You can also specify your own template file with the following conditions:

- The file must be on the MATLAB path and searchable.
- The file must have a `.vgt` extension.

You can use these optional tokens to customize the generated code by inserting them inside comment statements throughout the template:

- `%<FileName>`
- `%<PortList>`
- `%<EnumDataTypeDefinitions>`
- `%<ImportInitFunction>`
- `%<ImportOutputFunction>`
- `%<ImportUpdateFunction>`
- `%<ImportSetParamFunction>`
- `%<CallInitFunction>`
- `%<CallUpdateFunction>`

- %<CallOutputFunction>
- %<IsLibContinuous>
- %<ObjHandle>

See “Customize Generated SystemVerilog Code” on page 30-6 for instructions on customizing your code.

Note The SystemVerilog DPI component generator does not generate test benches for customized components.

Limitations

- By default, HDL Verifier converts matrices and vectors to one-dimensional arrays in SystemVerilog. For example, a 4-by-2 matrix in Simulink converts to a one-dimensional array of eight elements in SystemVerilog. To generate multiple scalar ports in the SystemVerilog interface, select **Scalarize matrix and vector ports** in the configuration parameters.
- SystemVerilog DPI component generation supports the following subsystems for code generation only. There is no test bench support for these subsystems.
 - Triggered subsystem
 - Enabled subsystem
 - Subsystem with action port

For best results, avoid exporting multiple subsystems separately because it can be difficult to achieve the correct execution order. Instead, combine multiple subsystems into one and generate code from the newly created, single subsystem.

See Also

Related Examples

- “Generate SystemVerilog DPI Component” on page 30-2
- “Generate Cross-Platform DPI Components” on page 30-30
- “Customize Generated SystemVerilog Code” on page 30-6
- “Verify Generated Component Against Simulink Data” on page 30-8
- “Use Generated DPI Functions in SystemVerilog” on page 30-9

SystemVerilog DPI Test Benches

HDL Verifier provides two types of test benches that generate a C-language component and integrate it into a SystemVerilog test bench with a direct programming interface (DPI). One test bench verifies a generated C component against saved data vectors from your Simulink subsystem. The other test bench verifies generated HDL code against a C component generated from the entire Simulink model.

- **Component Test Bench** — When you generate a C component from a Simulink subsystem for use as a DPI component, you can optionally generate a SystemVerilog test bench. The test bench verifies the generated DPI component against data vectors from your Simulink model. This feature is available in the Model Configuration Parameters dialog box, under **Code Generation**. See “Generate SystemVerilog DPI Component” on page 30-2.
- **HDL Code Test Bench** — When you generate HDL code from a subsystem, using HDL Coder, you can optionally generate a SystemVerilog test bench. This test bench compares the output of the HDL implementation against the results of the Simulink model. You can access this feature in HDL Workflow Advisor under **HDL Code Generation > Set Testbench Options**, or in the Model Configuration Parameters dialog box, under **HDL Code Generation > Test Bench**. Alternatively, for command-line access, set the `GenerateSVDPIITestBench` property of `makehdl.tb`. See “Verify HDL Design Using SystemVerilog DPI Test Bench” (HDL Coder).

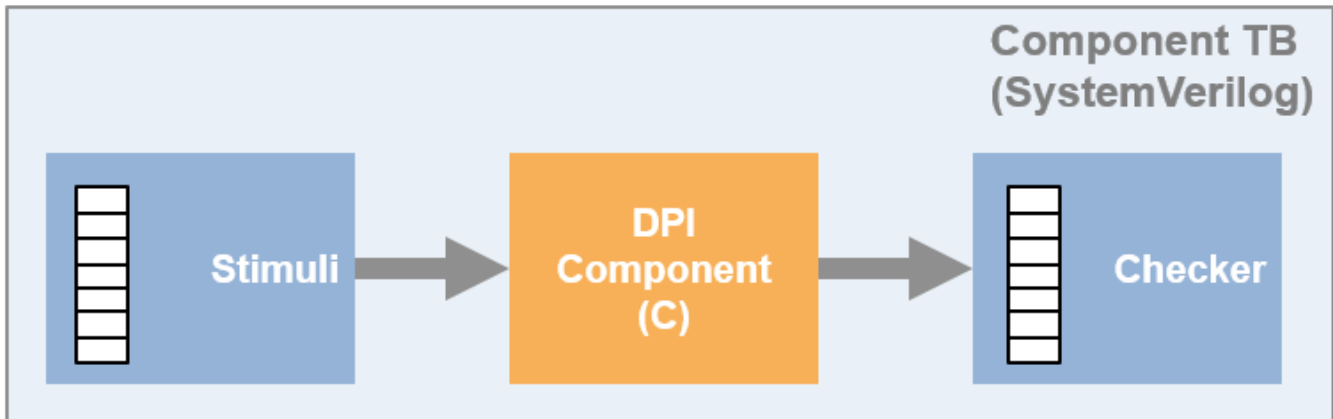
Both types of test benches require a Simulink Coder license.

Limitations

- HDL Verifier converts matrices and vectors to one-dimensional arrays in SystemVerilog. For example, a 4-by-2 matrix in Simulink is converted to a one-dimensional array of eight elements in SystemVerilog.
 - These subsystems do not support DPI test bench generation:
 - Triggered subsystem
 - Enabled subsystem
 - Subsystem with action port
-

Component Test Bench

The SystemVerilog DPI component generator also creates a test bench. You can use this test bench to verify that the generated SystemVerilog component is functionally equivalent to the original Simulink subsystem. The test bench saves data vectors from your Simulink simulation to apply as stimuli and to check against the output of the component. This test bench is not intended as a replacement for a system test bench for your own application. However, you can use the generated test bench as a starting example for your own system test bench.

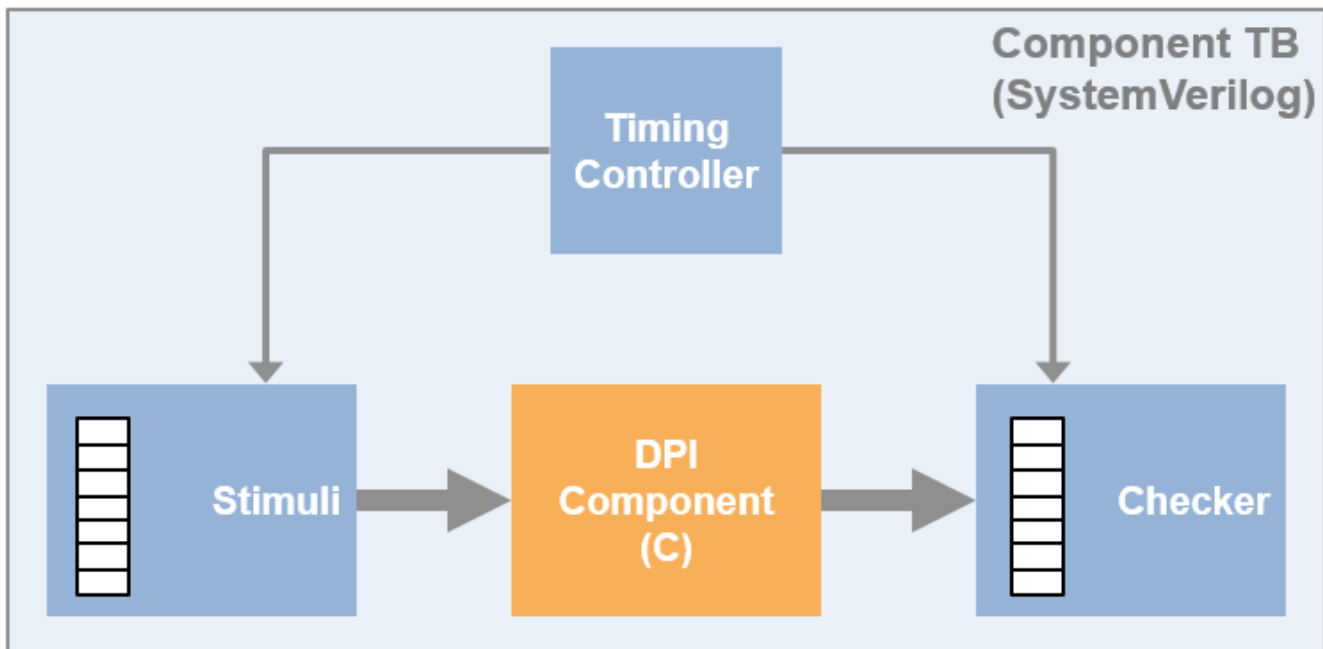


If you enable logging on test points in your model, the generated test bench also compares their signal values in the SystemVerilog component with logged values from Simulink.

Note HDL Verifier does not support test bench generation for custom generated SystemVerilog code. See "Customization" on page 29-10.

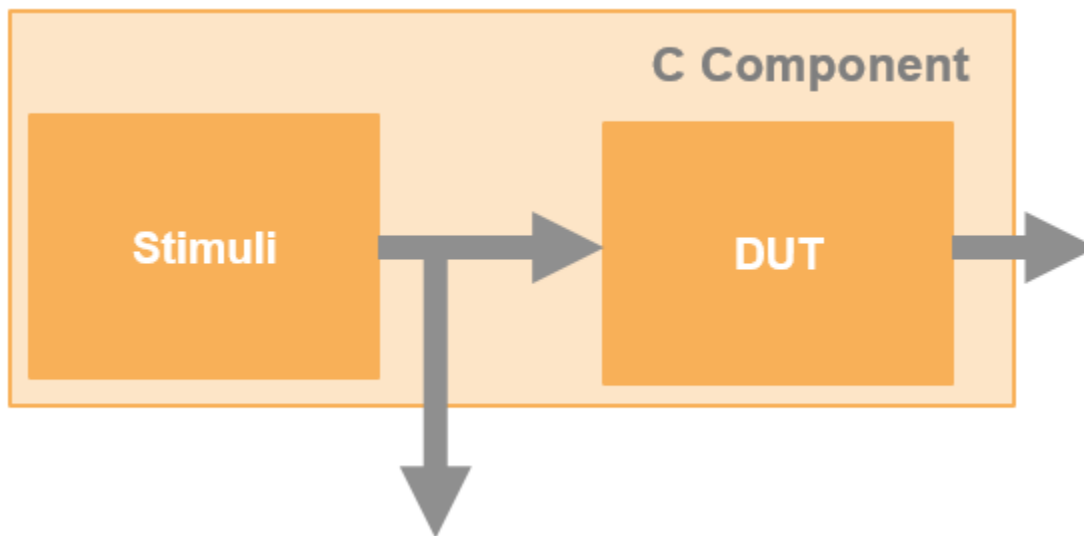
Multirate Component Test Bench

When your subsystem contains signals with more than one sample rate, the generated test bench includes a timing controller module. The timing controller generates input clock signals at the appropriate rates. Input stimuli and expected data outputs are applied and checked according to their sample rates.



HDL Code Test Bench

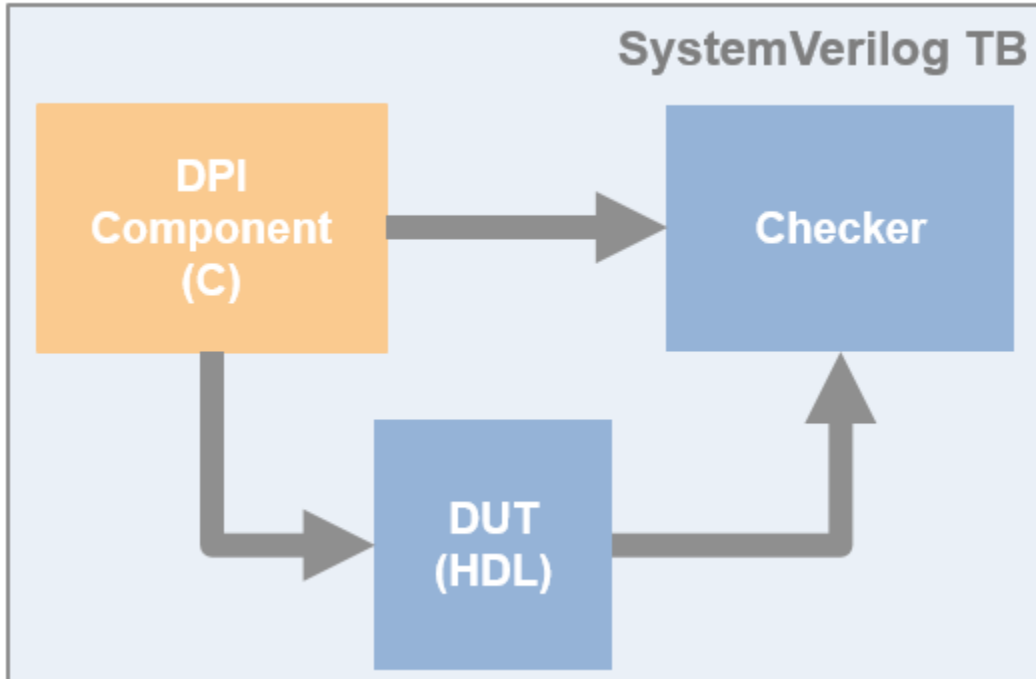
When you generate HDL code from a subsystem, using HDL Coder, you can also generate a SystemVerilog DPI test bench. This test bench compares the output of the HDL implementation against the results of the Simulink model. In addition to C code for your DUT subsystem, the coder also generates C code for the portion of your model that generates the input stimuli. Generation of this test bench is faster than the default HDL test bench for large data sets. This advantage is because the coder does not run the Simulink model to obtain the input and output data vectors. The generated C component calculates input stimuli and the output results for comparison with the HDL implementation.



The generated SystemVerilog test bench includes:

- Generated Verilog or VHDL code for your subsystem
- Generated C component
- Code to compare the output of the HDL code with the output of the C component.

Run this test bench to verify the generated HDL code implements the same algorithm as your Simulink model.



See Also

Related Examples

- "Generate SystemVerilog DPI Component" on page 30-2
- "Verify HDL Design Using SystemVerilog DPI Test Bench" (HDL Coder)

SystemVerilog DPI Component Generation for Simulink

- “Generate SystemVerilog DPI Component” on page 30-2
- “Customize Generated SystemVerilog Code” on page 30-6
- “Verify Generated Component Against Simulink Data” on page 30-8
- “Use Generated DPI Functions in SystemVerilog” on page 30-9
- “SystemVerilog DPI Component Test Point Access” on page 30-10
- “Tune Gain Parameter During Simulation” on page 30-12
- “Generate SystemVerilog Assertions from Simulink Test Bench” on page 30-16
- “Generate SystemVerilog Assertions and Functional Coverage” on page 30-22
- “Generate Cross-Platform DPI Components” on page 30-30

Generate SystemVerilog DPI Component

Step 1. Select Target

- 1 Open your model, and on the **Apps** tab, click **HDL Verifier**. Then, on the **HDL Verifier** tab, click **C Code Settings**. The **Configuration Parameters** dialog opens on the **Code Generation** pane.
- 2 At **System target file**, under **Target Selection**, click **Browse**. Select `systemverilog_dpi_grt.tlc` from the list.
 - Alternatively, if you have an Embedded Coder license, you can select target `systemverilog_dpi_ert.tlc`. This target enables you to access its additional code generation options on the **Code Generation** pane of the Model Configuration Parameters dialog box.

If you are generating a cross-platform component, you must select `systemverilog_dpi_ert.tlc` for the **System target file** parameter.

Step 2. Select Toolchain

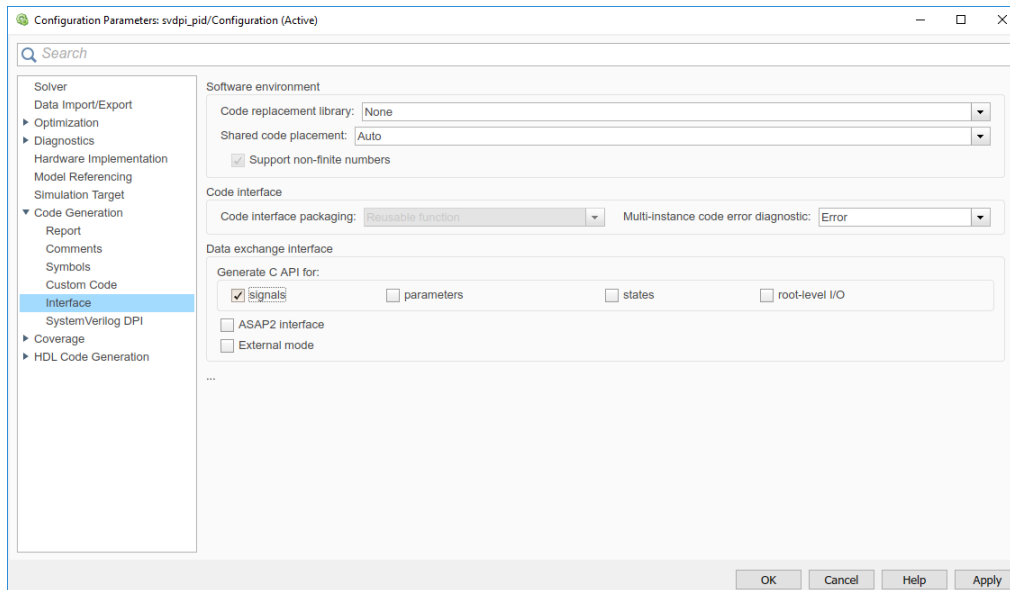
Still on the **Code Generation** pane, select a **Toolchain**. To generate a shared library for the same operating system as the host machine, select a compiler from the list of installed compilers or select `Automatically locate an installed toolchain`. To use the compiler included with the HDL simulator, or to generate a component for a different operating system, or to generate an HDL simulator project rather than a shared library, select an HDL simulator and your target operating system.

You can optionally add additional compilation flags. Under **Build Configuration**, select `Specify`. To display the current flags, click **Show Settings**.

Step 3. Enable Test Point Access (Optional)

Complete this step if you designated internal signals in your model as test points and want to access them in the generated DPI component.

- 1 In the left pane, select **Code Generation > Interface**.
- 2 In the **Generate C API for** section, verify that the **signals** check box is selected.



- 3 Select **Code Generation > SystemVerilog DPI**.
- 4 For **Generate access function to test point**, select One function per Test Point or One function for all Test Points.

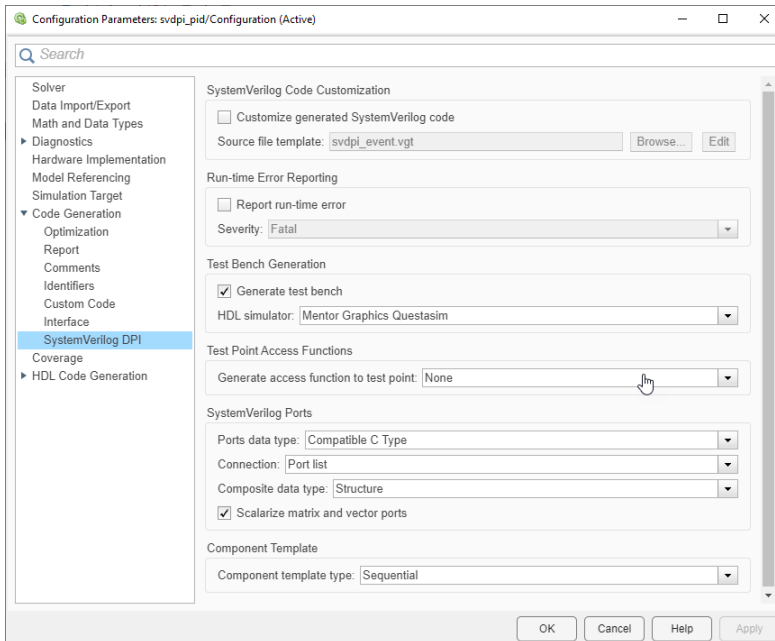
See “SystemVerilog DPI Component Test Point Access” on page 30-10.

Step 4. Configure SystemVerilog Generation Options

- 1 In the left pane, select **Code Generation > SystemVerilog DPI**.
- 2 Select **Report run-time error** to export run-time errors from Simulink to your HDL simulation. Not all Simulink blocks provide run-time error checks. You can add run-time checks by adding an Assertion block to your Simulink model.
- 3 Select **Generate test bench** to generate a test bench. The test bench checks the generated C component against data vectors from your Simulink subsystem.
- 4 In the **SystemVerilog Ports** section, set these parameters.
 - Select the SystemVerilog data types. (optional)
 - Set **Connection** to Port list or Interface.
 - Set **Composite data type** to Structure. This option creates SystemVerilog struct data types for any nonvirtual buses or for complex data types. Alternatively, select Flattened to create flattened ports.
 - Select **Scalarize matrix and vector ports** to create multiple SystemVerilog scalar ports from a Simulink vector or array. Clear this option to preserve arrays on the interface.
 - If your design is sequential and registered, set **Component template type** to Sequential. If your model is purely combinational (with no clock delays), set **Component template type** to Combinational.

Tip When using HDL Coder for code generation, match the generated interface by selecting the following options:

- Set **Ports data type** to Logic Vector.
- Set **Composite data type** to Flattened.
- Select **Scalarize matrix and vector ports**.



- 5 Click **OK** to accept these settings and to close the Configuration Parameters dialog box.

Step 5. Generate SystemVerilog DPI Component

- 1 In your model, right-click the block containing the subsystem you want to generate the component from. Select **Code > C/C++ Code > Build this Subsystem**.
- 2 Click **Build** in the dialog box.

The SystemVerilog component is generated as *subsystem_build/subsystem_dpi.sv*, where *subsystem* is the name of the subsystem from which you generated the DPI component. This build also results in a generated package file named *subsystem_build/subsystem_dpi_pkg.sv*, which includes all the function declarations for the component.

If you built the component for the host machine, you can now use the component. To copy the built component to another machine with the same operating system, copy these files:

- Shared library, *subsystem.so*, or *subsystem_win64.dll*
- Generated SystemVerilog wrapper, *subsystem_dpi.sv*
- Generated SystemVerilog package file, *subsystem_dpi_pkg.sv*
- Generated test bench folder, *dpi_tb* (optional)

To port the component to another machine with a different operating system, follow the instructions in “Generate Cross-Platform DPI Components” on page 30-30.

See Also

Related Examples

- “Use Generated DPI Functions in SystemVerilog” on page 30-9
- “Verify Generated Component Against Simulink Data” on page 30-8

Customize Generated SystemVerilog Code

In this section...

- “Set Up Model for Customized Code Generation” on page 30-6
- “Generate Customized SystemVerilog DPI Component” on page 30-7

Set Up Model for Customized Code Generation

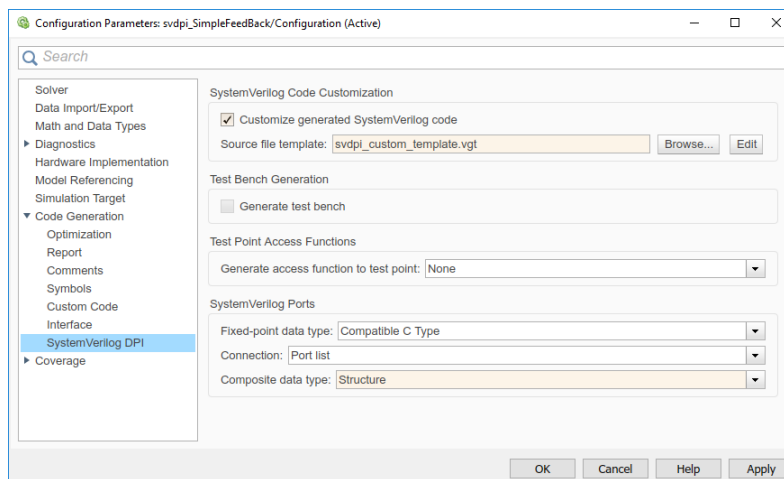
- 1 Open your model, and on the **Apps** tab, click **HDL Verifier**. Then, on the **HDL Verifier** tab, click **C Code Settings**. The **Configuration Parameters** dialog opens on the **Code Generation** pane.
- 2 For **System target file**, click **Browse** and select `systemverilog_dpi_grt.tlc`.

If you have a license for Embedded Coder, you can select target `systemverilog_dpi_ert.tlc`. This target enables you to access its additional code generation options (on the Code Generation pane in Model Configuration Parameters).

- 3 For **Toolchain**, in the **Build process** section, select the toolchain you want to use from the list. See “Generate Cross-Platform DPI Components” on page 30-30 for guidance on selecting a toolchain.

You can optionally select flags for compilation. For **Build configuration**, select **Specify**. Click **Show Settings** to display the current flags.

- 4 In the left pane, expand **Code Generation** and select **SystemVerilog DPI**.
- 5 Select **Customize generated SystemVerilog code**.
- 6 Specify the SystemVerilog template you want to use by setting **Source file template**.



Select **Edit** to see the contents of the specified **Source file template**. This example shows the content of the template file provided with HDL Verifier, `svdpi_grt_template.vgt`:



```

1  =====
2  **
3  ** Abstract:
4  ** This is the example SystemVerilog source code template.
5  ** With this template you are able to highly customize and
6  ** manipulate the appearance of the generated code. The template is
7  ** ideally suited for adding company information, including version
8  ** control tokens, adjusting the code's appearance, and so on.
9  **
10 ** There is a list of optional tokens that can be used to customize the
11 ** generated code.
12 **
13 ** %<FileName>, %<PortList>, %<EnumDataTypeDefinitions>, %<ImportInitFunction>, %<ImportOutputFunction>,
14 ** %<ImportUpdateFunction>, %<ImportSetParamFunction>, %<CallInitFunction>,
15 ** %<CallUpdateFunction>, %<CallOutputFunction>, %<IsLibContinuous>
16 ** %<ObjHandle>
17 **
18 ** For more customization options, see the HDL Verifier User's Guide.
19 **
20 ** Copyright 2013 The MathWorks, Inc.
21 |
22 `timescale 1ns / 1ns
23
24 module %<FileName>
25 (
26     input clk,
27     input reset,
28     input clken_in,
29     output clken_out,
30     %<PortList>
31 );
32
33     parameter isLibContinuous = %<IsLibContinuous>;
34     %<ObjHandle>
35     %<ImportInitFunction>
36     %<ImportOutputFunction>
37     %<ImportUpdateFunction>
38     %<ImportSetParamFunction>
39
40     always @(reset) begin
41         %<CallInitFunction>
42     end
43
44     always @(posedge clk) begin
45         if(isLibContinuous == 1) begin
46             %<CallOutputFunction>
47         end
48         %<CallUpdateFunction>
49     end
50
51     always @(clken_in) begin
52         if(isLibContinuous == 0) begin
53             %<CallOutputFunction>
54         end
55     end
56
57     assign #1 clken_out = clken_in;
58
59 endmodule
60

```

For more about the customized template, see “Customization” on page 29-10.

- 7 Click **OK** to accept these options and close the Configuration Parameters dialog box. Next, go to “Generate Customized SystemVerilog DPI Component” on page 30-7.

Generate Customized SystemVerilog DPI Component

- 1 In the **HDL Verifier** tab on the Simulink toolstrip, select **Generate DPI Component**.

You can alternatively use the `sbuild` function from the MATLAB command line.

- 2 If you built the component for the host machine, you can now use the component. If you intend to port the component to another machine with a different operating system, see “Generate Cross-Platform DPI Components” on page 30-30.

Verify Generated Component Against Simulink Data

For Mentor Graphics ModelSim and Questa Simulators

- 1 Start ModelSim or Questa in GUI mode.
- 2 Change your current folder to the dpi_tb folder under the code generation folder in your HDL simulator installation.
- 3 Enter the following command to start your simulation:

```
do run_tb_mq.do
```

- 4 When the simulation finishes, it displays the following message in your console:

```
*****TEST COMPLETED (PASSED)*****
```

For the Cadence Xcelium Simulator

- 1 Start your terminal shell.
- 2 Change the current folder to "dpi_tb" under the code generation folder.
- 3 Enter the following command in your shell.

```
sh run_tb_ncsim.sh
```

- 4 When the simulation finishes, it displays the following message in your console:

```
*****TEST COMPLETED (PASSED)*****
```

Use Generated DPI Functions in SystemVerilog

To use the generated DPI component in a SystemVerilog environment, first import the generated functions with “DPI” declarations within your SystemVerilog module. Then, call and schedule the generated functions. When you compile the SystemVerilog code that contains the imported generated functions, use a DPI-aware SystemVerilog compiler and specify the component file names.

The following example demonstrates adding the generated DPI component for the DPI_blk block to a SystemVerilog module.

- 1 Import the generated functions:

```
import "DPI" function void DPI_blk1_initialize();
import "DPI" function void DPI_blk1_ouptut(output real blk1_Y_Out1);
import "DPI" function void DPI_blk1_update(input real blk1_U_0n1);
```

- 2 Call the Initialize function.

```
DPI_blk1_initialize();
```

- 3 Schedule the output and update function calls:

```
DPI_blk1_output( blk1_Y_Out1);
DPI_blk1_update( blk1_U_In1);
```

Example

```
import "DPI" function void DPI_blk1_initialize();
import "DPI" function void DPI_blk1_ouptut(output real blk1_Y_Out1);
import "DPI" function void DPI_blk1_update(input real blk1_U_0n1);

module test_twoblock_tb;

    initial begin
        DPI_blk1_initialization();
    end

    always@(posedge clk) begin
        #1
        DPI_blk1_output(blk1_Y_Out1);
        DPI_blk1_update(blk1_U_In1);
    end

    always@(posedge clk)
    begin
        blk1_U_In1 = blk1_U_In1 + 1.0;
    end
end
```

SystemVerilog DPI Component Test Point Access

You can designate internal signals in your model as test points and configure the SystemVerilog DPI generator to create one or more access functions. You can also enable logging on test points. Then you can use the generated test bench to compare the Simulink data with values observed while running the SystemVerilog component.

Step 1. Choose Internal Signals

Choose an internal signal in your model, following these guidelines:

- Enable the test point at the source of the signal. If the test point is on a connecting signal, such as between subsystems, the signal might be optimized out of the generated code.
- Select a signal that is not an input or output of your component. If you select an I/O signal, the generator does not provide an access function. Such an access function is redundant because you already have visibility of the I/O signals.
- Signals of type `enum` are not supported.
- Virtual signals and buses are not supported.
- Continuous, asynchronous, and triggered sample times are not supported.
- Multirate designs are not supported for signal logging. You can add a test point, and generate an access function. However, the test bench is single rate and cannot perform a comparison against logged data at different rates.
- Model references are not supported. If you want to add a test point in a model reference, you first have to wire the signal out of the model reference. Once the signal is accessible in your model, you can select it as a test point.

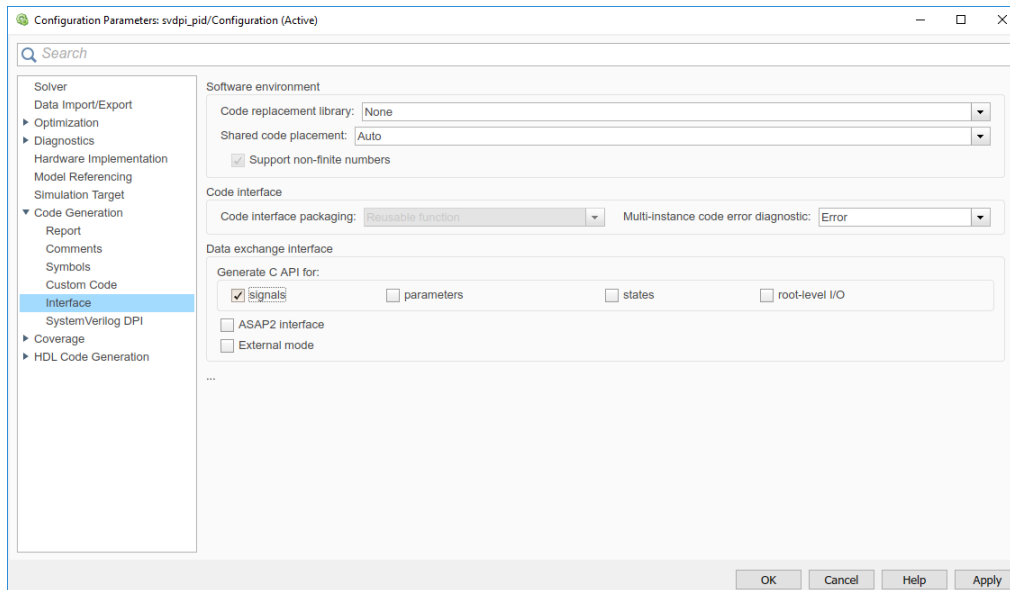
Step 2. Add Test Points

- 1 In your model, right-click the signal and select **Properties**.
- 2 Select the **Test point** check box.
- 3 Give the test point a unique name in the **Signal name** box.
- 4 Optionally, select **Log signal data**. This check box enables the generated test bench to compare logged data from the model against values observed while running the generated component. The test bench uses the generated access functions to fetch the signal values during the simulation.

For more details on test points and logging in Simulink, see “Configure Signals as Test Points” (Simulink).

Step 3. Enable Component Interface

- 1 Open your model, and on the **Apps** tab, click **HDL Verifier**. Then, on the **HDL Verifier** tab, click **C Code Settings**. The **Configuration Parameters** dialog opens on **Code Generation**.
- 2 On the left pane, under **Code Generation**, select **Interface**.
- 3 In **Generate C API for**, ensure the **signals** check box is selected. The other check boxes do not affect the DPI component or test bench.



Step 4. Configure Access Function

- 1 In the **HDL Verifier** tab, click **SystemVerilog Settings**.
- 2 For **Generate access function to test point**, select **One function per Test Point** or **One function for all Test Points**.

If you select **One function for all Test Points**, a single function returns values for all test points.

```
DPI_TestPointAccessFcn(inputchandle objhandle, input real Name1, inout real Name2);
```

If you select **One function per Test Point**, each signal has a separate access function.

```
DPI_Name_TestPoint(inputchandle objhandle, inout real Name);
```

If you select **None**, the tool does not generate access functions.

See Also

Related Examples

- “Get Started with SystemVerilog DPI Component Generation” on page 32-68

Tune Gain Parameter During Simulation

In this section...

“Step 1. Create a Simple Gain Model” on page 30-12

“Step 2. Create Data Object for Gain Parameter” on page 30-12

“Step 3. Generate SystemVerilog DPI Component” on page 30-14

“Step 4. Add Parameter Tuning Code to SystemVerilog File” on page 30-14

“Step 5. Run Simulation with Parameter Change” on page 30-15

Step 1. Create a Simple Gain Model

To perform the example steps yourself, first create an example model.

The example model has a single gain block with a gain parameter that is tuned during simulation.



- 1 Open the Simulink Block Library and click Commonly Used Blocks.
- 2 Add an Inport block.
- 3 Add a Gain block. Double-click to open the block mask and change the value in the **Gain** parameter to gain.
- 4 Double click the Gain block to open the block mask. In the **Signal Attributes** tab, and set **Output data type** to uint8.

Note If you leave **Output data type** as **Inherit: Inherit via internal rule**, Simulink Coder selects a data type based on the default value of the Gain parameter, which might not accommodate the value when tuning the parameter after DPI generation. Specify a specific type for the block's output signal, to avoid incorrect type setting by Simulink Coder.

- 5 Add an Output block.
- 6 Connect all blocks as shown in the preceding diagram.

Step 2. Create Data Object for Gain Parameter

- 1 Create a data object for the gain parameter:

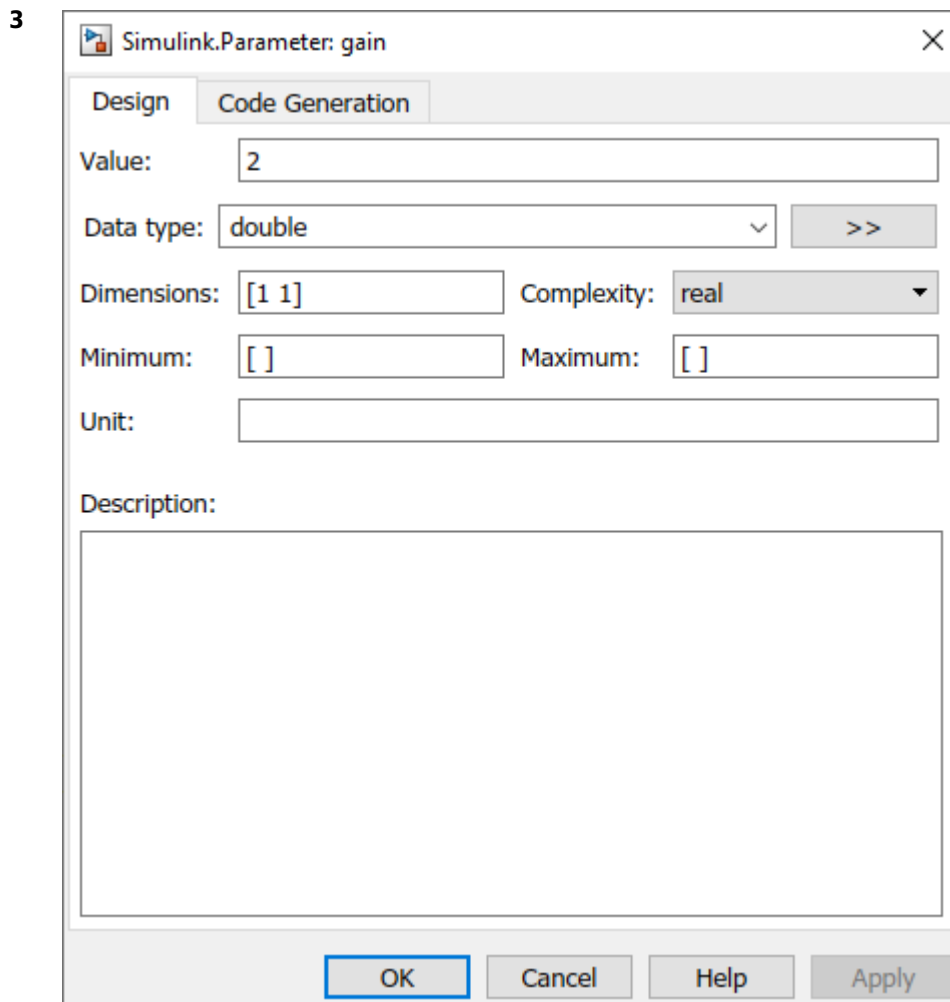
At the MATLAB command prompt, type:

```
gain = Simulink.Parameter
```

- 2 Next, type:

```
open('gain')
```

This command opens the property dialog box for the parameter object.



On the **Design** tab, select the following values:

- **Value:** 2
- **Data type:** double

On the **Code Generation** tab, select the following values:

- **Storage class:** Model default

4 Click **OK**.

For more information about using parameter objects for code generation, see “C Code Generation Configuration for Model Interface Elements” (Simulink Coder).

Note Setting **Storage class** to Auto optimizes the parameter during code generation. Recommended values when generating DPI tunable parameters are:

- Model default
- SimulinkGlobal

- ExportedGlobal

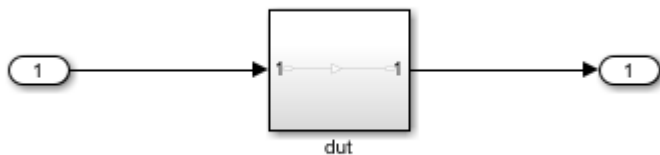
Use `Model default` when your parameter is instance-specific, and choose between `SimulinkGlobal` and `ExportedGlobal` to generate a global variable.

Step 3. Generate SystemVerilog DPI Component

- 1 On the Simulink **Apps** tab click **HDL Verifier**. In the right pane, set the **HDL Verifier Mode** to **DPI Component Generation**.
- 2 On the **HDL Verifier** tab, click **C Code Settings**. The **Configuration Parameters** dialog opens on **Code Generation**.
- 3 At **System target file**, click **Browse** and select `systemverilog_dpi_grt.tlc`.
 - If you have a license for Embedded Coder, you can select target `systemverilog_dpi_ert.tlc`. This target allows you to access its additional code generation options (on the Code Generation pane in Model Configuration Parameters).
- 4 At **Toolchain**, under **Build process**, select the toolchain you want to use from the list. See “Generate Cross-Platform DPI Components” on page 30-30 for guidance on selecting a toolchain.

You can optionally select flags for compilation. Under **Build Configuration**, select `Specify` from the drop-down list. Click **Show Settings** to display the current flags.

- 5 In the Code Generation group, click **SystemVerilog DPI**.
- 6 Leave both **Generate test bench** and **Customize generated SystemVerilog code** cleared (because this example modifies the generated SystemVerilog code, any test bench generated at the same time does not have the correct results).
- 7 Click **OK** to accept these settings and to close the Configuration Parameters dialog box.
- 8 In the example model, right-click the gain block and select **Create Subsystem from Selection**. For this example, rename the subsystem `dut`.



- 9 In the Simulink Toolstrip, on the **HDL Verifier** tab, click **Generate DPI Component**.

The SystemVerilog component is generated as `dut_build/dut_dpi.sv` in your current working folder.

Step 4. Add Parameter Tuning Code to SystemVerilog File

- 1 Open the file `dut_build/dut_dpi.sv` and examine the generated code.
- 2 In this example, after you call the `reset` function, call the `DPI_dut_setparam_gain` function with the new parameter value. For example, here the gain is changed to 6:


```
DPI_dut_setparam_gain(objhandle, 6);
```

- 3 If the asynchronous reset signal is high (goes from 0 to 1), call the reset function again.

```
if(reset == 1'b1) begin
    DPI_dut_reset(objhandle, dut_U_In1, dut_Y_Out1);
    DPI_dut_setparam_gain(objhandle, 6);
end
```

- 4 The SystemVerilog code now looks like this:

```
module dut_dpi(
    input clk,
    input clk_enable,
    input reset,
    input real dut_U_In1,
    output real dut_Y_Out1
);

   chandle objhandle=null;
    // Declare imported C functions
    import "DPI" function chandle DPI_dut_initialize(chandle existhandle);
    import "DPI" function void DPI_dut_reset
        (input chandle objhandle, input real dut_U_In1, inout real dut_Y_Out1);
    import "DPI" function void DPI_dut_output
        (input chandle objhandle, input real dut_U_In1, inout real dut_Y_Out1);
    import "DPI" function void DPI_dut_update
        (input chandle objhandle, input real dut_U_In1);
    import "DPI" function void DPI_dut_terminate(input chandle objhandle);
    import "DPI" function void DPI_dut_setparam_gain
        (input chandle objhandle, input real dut_P_gain);

    initial begin
        objhandle = DPI_dut_initialize(objhandle);
        DPI_dut_setparam_gain(objhandle, 6);
    end

    final begin
        DPI_dut_terminate(objhandle);
    end

    always @(posedge clk or posedge reset) begin
        if(reset == 1'b1) begin
            DPI_dut_reset(objhandle, dut_U_In1, dut_Y_Out1);
            DPI_dut_setparam_gain(objhandle, 6);
        end
        else if(clk_enable) begin
            DPI_dut_output(objhandle, dut_U_In1, dut_Y_Out1);
            DPI_dut_update(objhandle, dut_U_In1);
        end
    end
end
endmodule
```

Step 5. Run Simulation with Parameter Change

To run your simulation, build the shared library and export the component, as explained in the following topics:

- Rebuild shared library as described in “Build Libraries” on page 30-31.
- “Use Generated DPI Functions in SystemVerilog” on page 30-9

Generate SystemVerilog Assertions from Simulink Test Bench

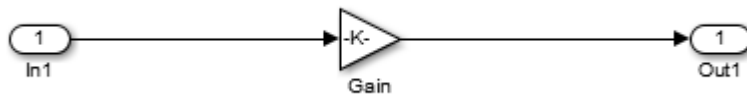
The DPI assertion block checks whether its input signal is zero. Use this block to check that your Simulink test bench behaves as expected by creating a Boolean expression and connecting it to the block. Generating SystemVerilog creates an immediate assertion in your generated module. Use this block to check that your stimulus behaves as expected in both your Simulink and SystemVerilog environments.

Generate Assertions Workflow

This example shows how to create a model with an assertion block that emits a warning when the output of a gain block is zero. Then use a counter to display the model output.

1 Create a Simulink Model

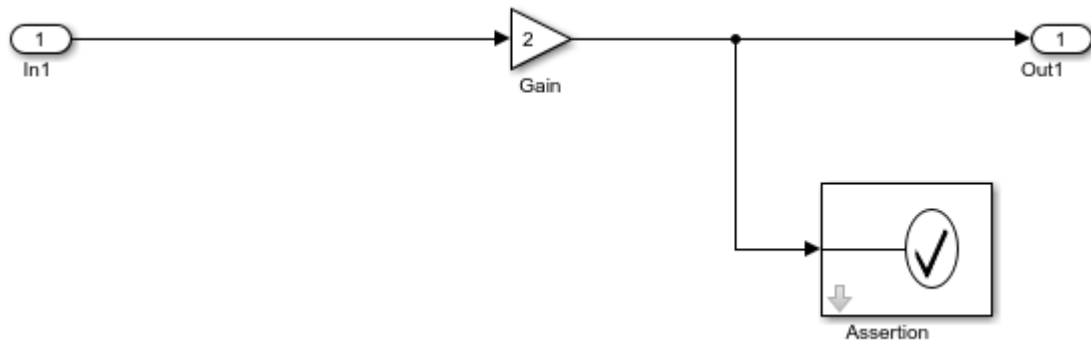
The example model has a single gain block. This example creates a warning every time the gain output is zero.



- 1 Open the **Simulink Block Library > Commonly Used Blocks**.
- 2 Add an Inport block.
- 3 Add a Gain block. Double-click this block to open its parameters. Set the value of **Gain** to 2.
- 4 Add an Outport block.
- 5 Connect all blocks as shown in the preceding diagram.

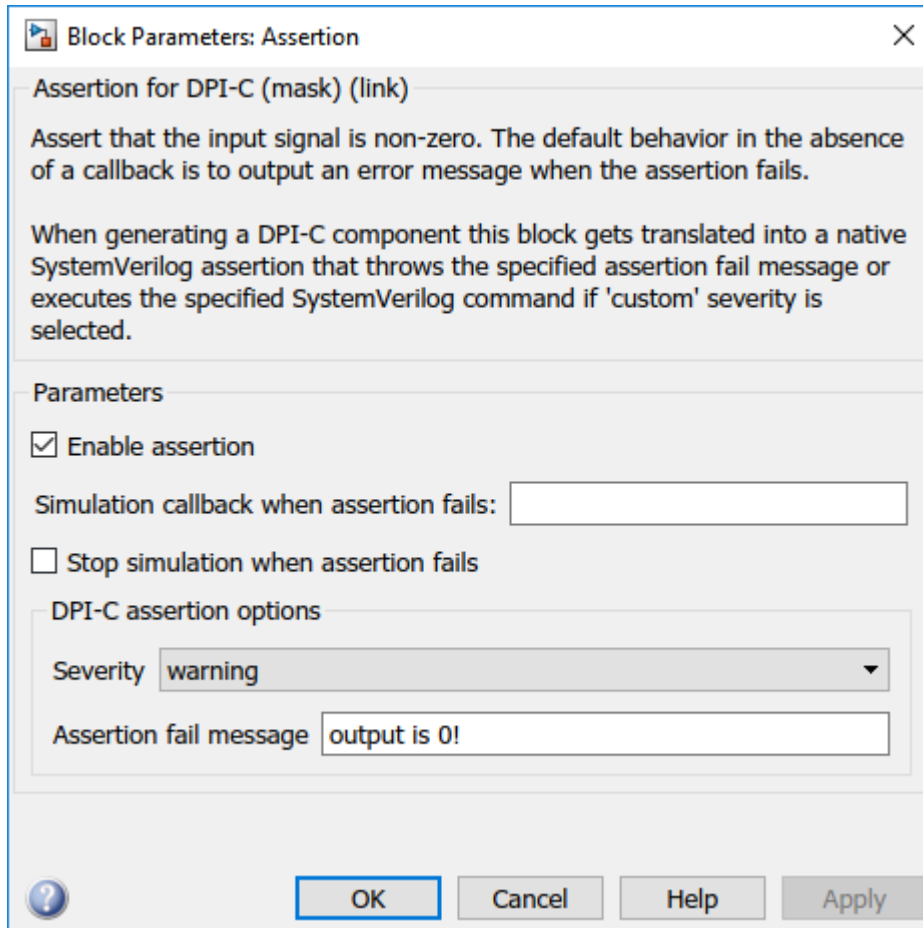
2 Add and Configure Assertion Block

Find the Assertion block in the **Libraries** tree view by selecting **HDL Verifier > For Use with DPI-C SystemVerilog**. Add this block to your model, then connect the output of the Gain block to the input of the assertion block.



This example uses the Assertion block to monitor the Gain output and return a warning when the signal is zero. Double-click the Assertion block to configure its parameters. Set **Severity** to

warning and the **Assertion fail message** to "output is 0!". Make sure that **Enable assertion** is selected.



Note The **Parameters** section control the Simulink execution, and they are identical to the parameters in the Simulink Assertion block. The **DPI-C assertion options** control the assertion behavior only in the generated SystemVerilog.

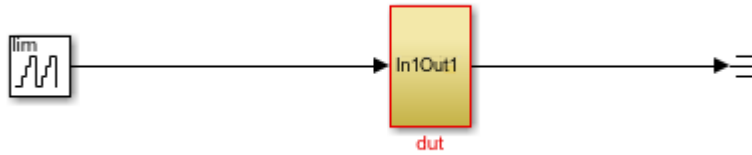
3 Customize the Assertion

Customize the assertion by setting **Severity** to custom and typing a custom SystemVerilog command in the **Assertion custom command** box. This command can include system tasks such as \$display or \$time.

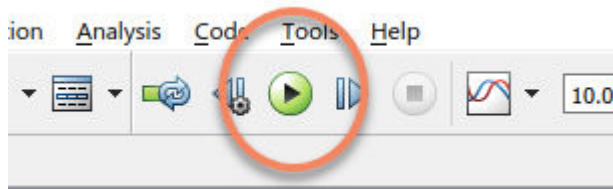
You can further customize assertion behavior using the generated `DPI_getAssertionInfo(obj)` SystemVerilog function. This function checks for executed assertions and returns all information recorded for that assertion in a SystemVerilog struct array. For each assertion that was executed in that clock cycle, the function returns the **Status**, **Message**, and **Severity** of the assertion.

4 Run Simulation in Simulink

Before simulating, connect a stimulus source and a sink to your subsystem. This example uses a counter that generates 0-1-2-3 sequences.



To build and run the simulation, click the **Run** button on toolbar.



Note the warnings from the assertion block in the output.

```

Simulation 13
12:18 PM Elapsed: 0.602 sec
Assertion detected in 'assertion/dut/Assertion_/Assertion' at time 0
Component: Simulink | Category: Block warning
Assertion detected in 'assertion/dut/Assertion_/Assertion' at time 0.8
Component: Simulink | Category: Block warning
Assertion detected in 'assertion/dut/Assertion_/Assertion' at time 1.6
Component: Simulink | Category: Block warning
Assertion detected in 'assertion/dut/Assertion_/Assertion' at time 2.4
Component: Simulink | Category: Block warning
Assertion detected in 'assertion/dut/Assertion_/Assertion' at time 3.2
Component: Simulink | Category: Block warning
Assertion detected in 'assertion/dut/Assertion_/Assertion' at time 4
Component: Simulink | Category: Block warning
Assertion detected in 'assertion/dut/Assertion_/Assertion' at time 4.8
Component: Simulink | Category: Block warning

```

5 Generate SystemVerilog DPI Component

- 1 On the Simulink **Apps** tab click **HDL Verifier**. Then, on the **HDL Verifier** tab, click **C Code Settings**. The **Configuration Parameters** dialog opens on **Code Generation**.
- 2 At **System target file**, click **Browse** and select `systemverilog_dpi_grt.tlc`.
 - If you have a license for Embedded Coder, you can select target `systemverilog_dpi_ert.tlc`. This target allows you to access its additional code generation options (on the Code Generation pane in Model Configuration Parameters).

- 3 In the Code Generation group, click **SystemVerilog DPI**.
- 4 To enable automatic test bench generation, select the **Generate test bench** check box.
- 5 Click **OK** to accept these settings and close the Configuration Parameters dialog box.
- 6 On the **HDL Verifier** tab, click **Generate DPI Component**.

The SystemVerilog component is generated as `dut_build/dut_dpi.sv` in your current working folder. In addition, a package file including function declarations is generated in as `dut_build/dut_dpi_pkg.sv` in your current working folder.

6 Run the SystemVerilog Simulation

In the **HDL Verifier** tab click **Select Simulator** to open the Configuration Parameters on the **SystemVerilog DPI** pane. Then, select a simulator from the **HDL simulator** list. Click **OK**.

- 1 To start the simulator in GUI mode, expand the **Run Testbench** button and select **Launch Simulator in GUI Mode**.
- 2 For ModelSim or Questa, enter the following command to start your simulation.

```
do run_tb_mq.do
```

Notice the simulation warnings displayed by the assertion:

```
run -all
# ** Warning: assertion:14:output is 0!
#   Time: 40 ns   Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 80 ns   Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 120 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 160 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 200 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 240 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 280 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 320 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 360 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 400 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 440 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 480 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#   Time: 520 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# *****TEST COMPLETED (PASSED)*****
# ** Note: $finish      : ./dut_dpi_tb.sv(62)
#   Time: 542 ns Iteration: 0 Instance: /dut_dpi_tb
# End time: 14:16:43 on Dec 29,2017, Elapsed time: 0:00:04
# Errors: 0, Warnings: 13
```

Trace Generated SystemVerilog Assertions

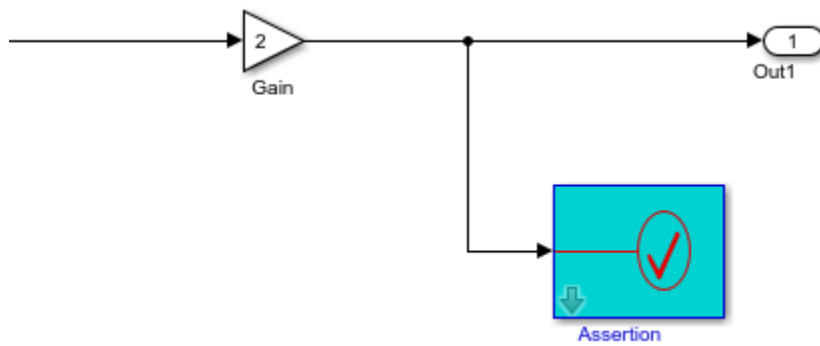
After running a SystemVerilog simulation with a generated assertion, your log file displays warnings and errors. To identify which assertion block originated a specific warning or error output, use the `hilite_system` function.

Each warning displays a number identifying the specific Assertion block that generated that warning. That number is the Simulink identifier (SID) of that block. For example, the following shows a warning generated by assertion block with SID number 14.

```
# ** Warning: assertion:14:output is 0!
```

To highlight the block that generated this warning, execute the following code in your MATLAB command window.

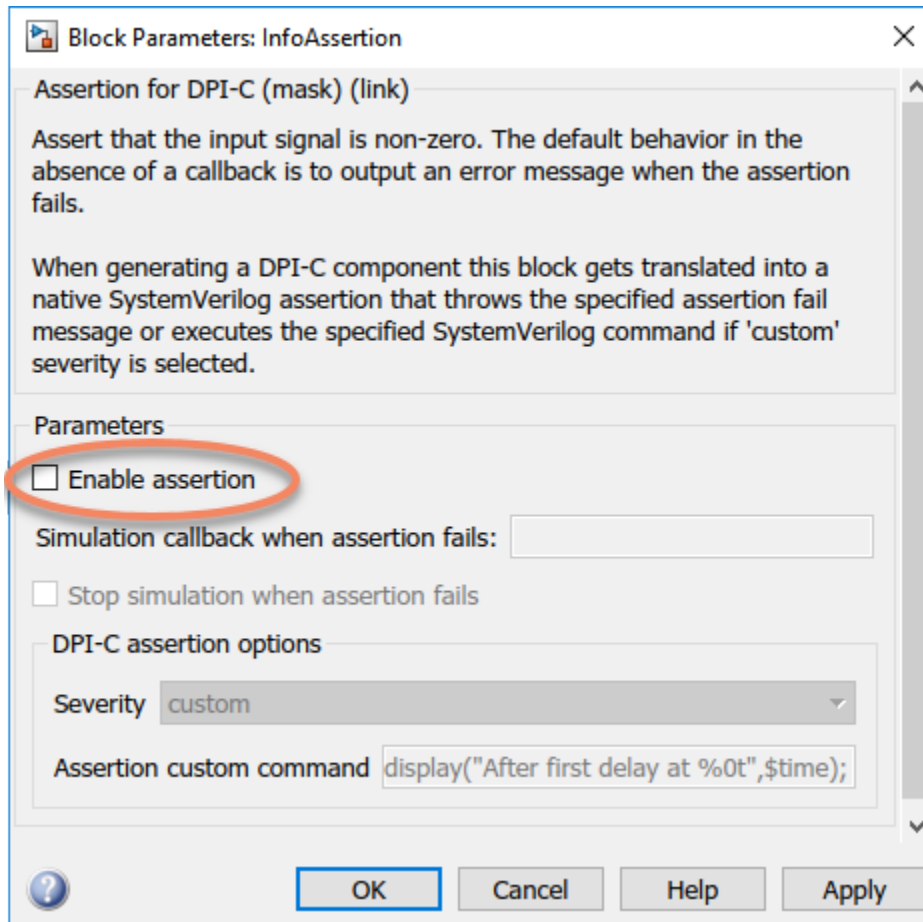
```
hilite_system('assertion:14')
```



Disabling Assertions

You can disable any assertion block from executing either in your Simulink environment or in your SystemVerilog environment. Disable the assertion in Simulink if you want an assertion ignored both in Simulink and in SystemVerilog. Disable the assertion in SystemVerilog if you do not want to regenerate code from Simulink, but want the ability to disable assertions during a SystemVerilog simulation.

Disabling Assertions in Simulink. You can disable the DPI-C Assertion block from checking its input signal or emitting warnings or errors by clearing the **Enable assertion** check box in the Assertion block parameters.



Clearing this check box disables the assertion from emitting any warnings or errors and generating a SystemVerilog assertion.

Disabling Assertions in SystemVerilog. In the SystemVerilog environment, you can disable the assertion by providing a Simulink Identifier as a command-line argument to the HDL simulator. For example, when using ModelSim and assuming the SID is 14, you can disable the output of any warnings, error messages, or custom commands produced by the Assertion block with the following plus-argument:

```
vsim -c -voptargs=+acc -sv_lib ../dut_win64 work.dut_dpi_tb +assertion:14
```

See Also

Assertion

More About

- “Generate SystemVerilog DPI Component” on page 30-2

Generate SystemVerilog Assertions and Functional Coverage

SystemVerilog DPI component generation and Universal Verification Methodology (UVM) test bench generation workflows enable you to reuse Simulink verification models in the resulting SystemVerilog. Simulink model verification blocks such as Assertion or Check Dynamic Lower Bound, and calls to `verify` statements create error checks and functional coverage points in the generated SystemVerilog.

When a Simulink assertion or `verify` call fails, it generates a SystemVerilog error by default. When either succeeds, it generates a SystemVerilog cover point which logs a PASS result. Assertions and `verify` statement behaviors can be customized using SystemVerilog command line arguments and the HDL Verifier Assertion block. For more information about customization, see “Customize Assertion” on page 30-24.

Create a Simulink Test Bench Model

In Simulink, create a model for the device under test (DUT), and then create a test bench for the model. You can use a combination of assertion blocks from the Simulink / Model Verification library and blocks that contain `verify` statements from the Simulink Test library, such as:

- A `verify` statement
- A block from the “Model Verification” (Simulink) library
- An HDL Verifier Assertion block

Create Simulink Test Sequence

In your test bench model, include a `verify` statement by adding one or more of these blocks:

- Test Assessment
- Test Sequence
- Chart

To create and edit test steps, use the “Test Sequence Editor” (Simulink Test). In the test sequence, use `verify` statements to assess the simulation, as described in “Test Sequence and Assessment Syntax” (Simulink Test).

The `verify` statement and the Test Sequence block represent a temporal check in Simulink. When you generate a SystemVerilog DPI component, the temporal logic is located in the generated C code. The SystemVerilog wrapper contains an immediate assertion that triggers when the `verify` condition is violated.

Include Simulink Model Verification Blocks

You can also include these assertion blocks from the Simulink / “Model Verification” (Simulink) library.

- Assertion
- Check Dynamic Gap
- Check Dynamic Range
- Check Static Gap

- Check Static Range
- Check Dynamic Lower Bound
- Check Dynamic Upper Bound
- Check Input Resolution
- Check Static Lower Bound
- Check Static Upper Bound
- Check Discrete Gradient

In addition, you can include the HDL Verifier Assertion block to create customizable assertions. For an example that uses the HDL Verifier Assertion block, see “Generate Native SystemVerilog Assertions from Simulink” on page 32-89.

In SystemVerilog, every model verification block and `verify` statement is mapped to an assertion and a coverage point. You can adjust coverage goals, filter specific assertions, and see verbose information for each of the `verify` statements.

You can use multiple `verify` statements and assertion blocks in your model.

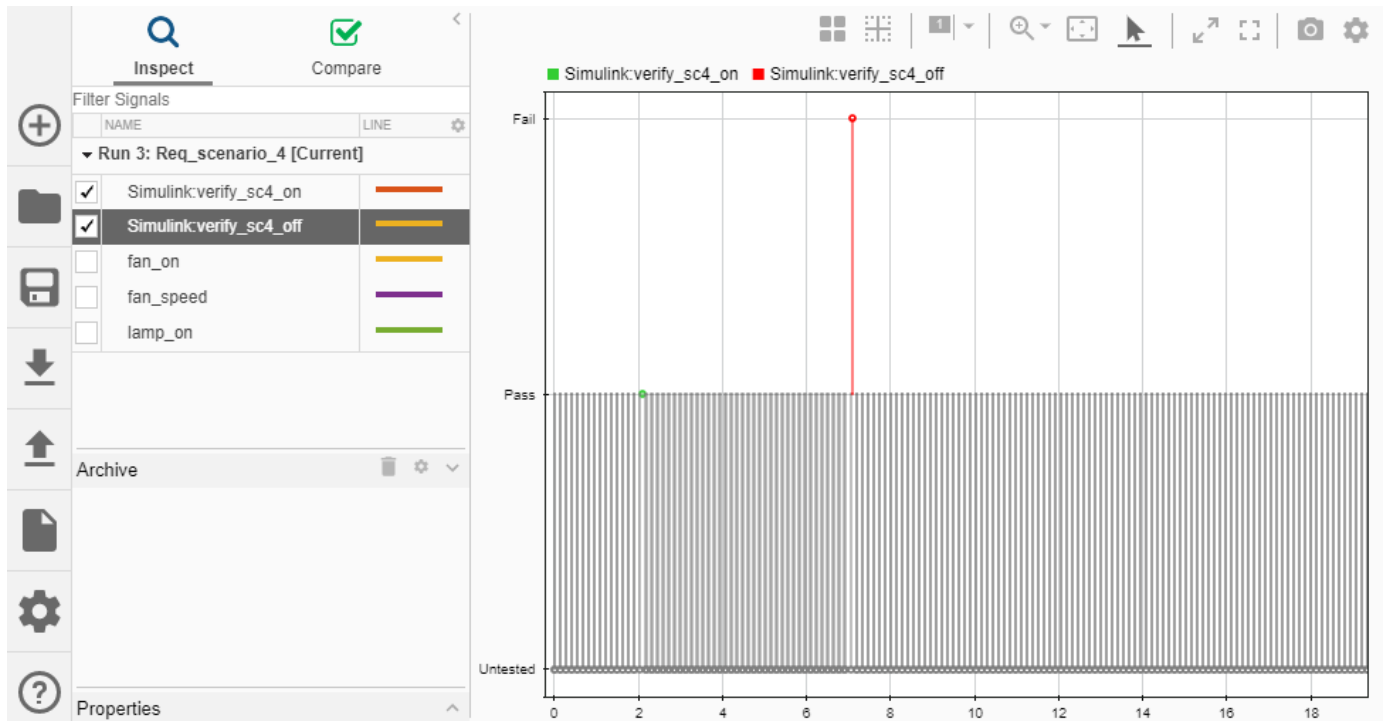
When simulating your design in Simulink, the simulation warns if the assertion or the `verify` assessment fails.



You can view and inspect the simulation results by using the **Simulation Data Inspector**. Open the **Simulation Data Inspector** by entering this code at the MATLAB command prompt.

```
Simulink.sdi.view
```

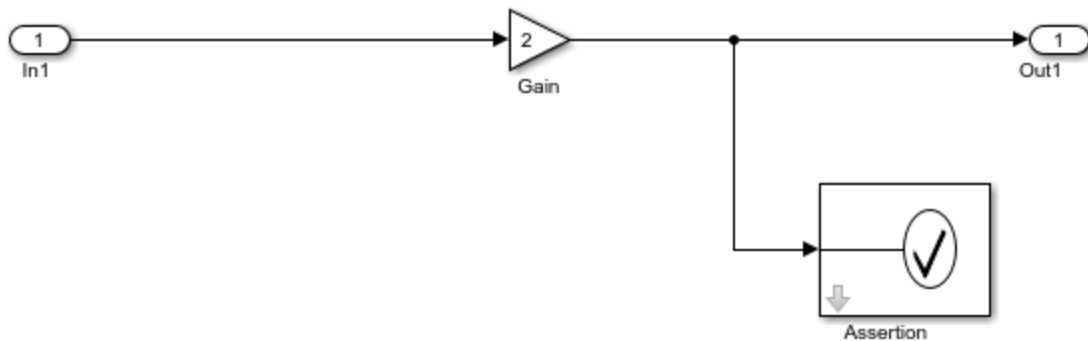
To view signals over time, select them in the left pane of the **Simulation Data Inspector**.



Customize Assertion

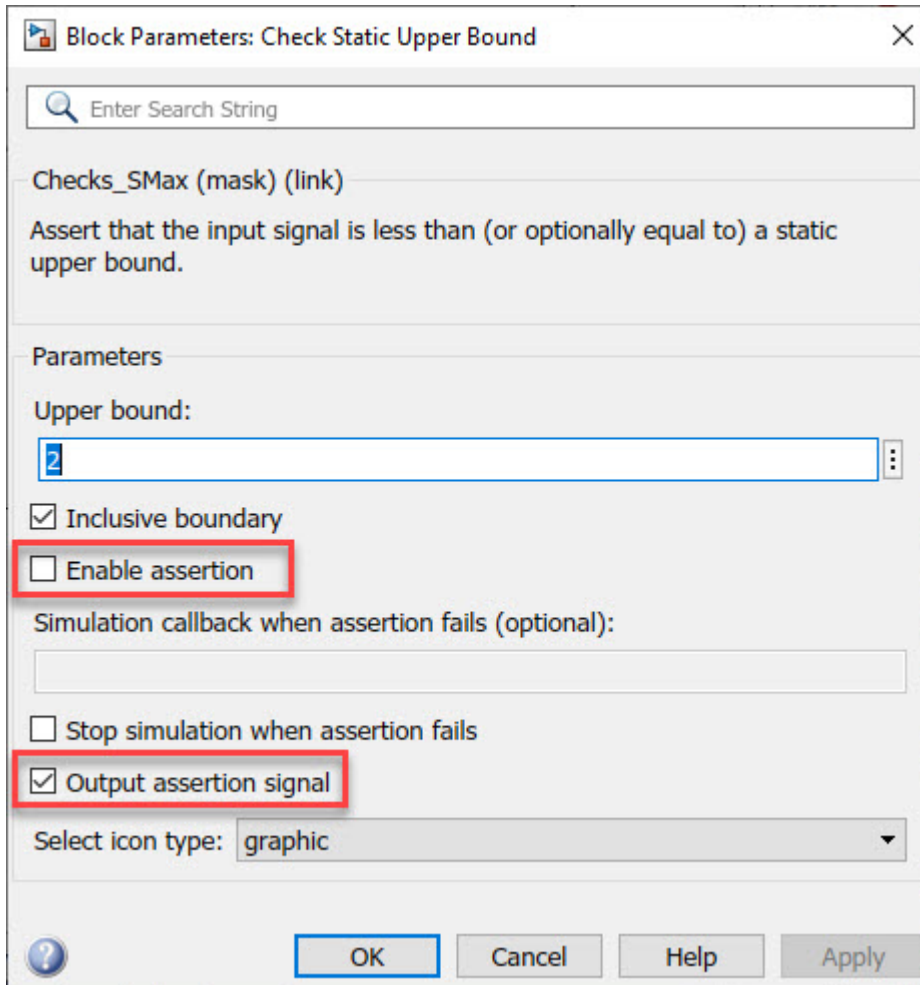
You can customize the SystemVerilog immediate assertion in two ways:

- Include an HDL Verifier Assertion block, and customize the generated SystemVerilog immediate assertion. You can set a custom message when the assertion fails and can choose between warning, error, or a custom command when the assertion fails. For an example that uses the HDL Verifier Assertion block, see “Generate Native SystemVerilog Assertions from Simulink” on page 32-89. The result is similar to this figure.



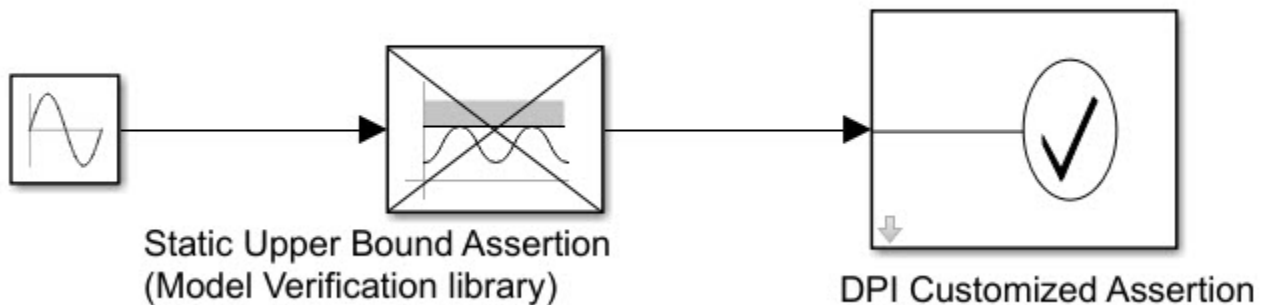
- To customize an assertion from the Model Verification library, connect the block output to a customizable HDL Verifier Assertion block, by following these steps.
 - 1 Add a block from the Simulink / Model Verification library to your model.
 - 2 Open the block mask, and then set these parameters (as shown in this figure):

- Clear the **Enable assertion** parameter to prevent redundant assertion outputs.
- Select the **Output assertion signal** parameter to create a boolean output signal that captures the assertion.



- 3 Add an Assertion block from the HDL Verifier / For Use with DPI-C SystemVerilog library, and connect the output signal of the Model Verification block to the input port of the Assertion block.
- 4 Customize the Assertion block by following the steps in the “Generate Assertions Workflow” on page 30-16 example.

The result is similar to this figure.



Generate a UVM or SystemVerilog DPI Component

Configure the Model for Code Generation

In the Configuration Parameters dialog box, select **Code Generation** in the left pane. Under **Target Selection**, set **System Target File** to `systemverilog_dpi_grt.tlc` or to `systemverilog_dpi_ert.tlc` when using Embedded Coder.

Select **SystemVerilog DPI** in the left pane. Under **SystemVerilog Ports**, set the data type and connection settings. Click **OK**.

Generate a UVM or SystemVerilog DPI Component

Note To generate a UVM or DPI Component, the assertion or test block must be inside a Simulink subsystem.

In Simulink, right-click the subsystem block, which contains the test sequence, and select **C/C++ Code > Build This Subsystem**. Click **Build** in the dialog box that opens.

Alternatively, you can use the MATLAB command line to generate the DPI component. Use the `slbuild` function to build the system. For example, to build a subsystem named "My_verify_tst", enter this code at the MATLAB command line.

```
slbuild('My_verify_tst');
```

You can also use the `uvmbuild` function to generate a UVM test bench. If your test model contains `verify` statements, they are mapped to assertions in your UVM environment, and coverage data is collected.

Run HDL Simulation with the Generated Component

Change your current folder to the `dpi_tb` folder, which is under the code generation folder in your HDL simulator installation. Start your HDL simulator, and run the generated script to start the simulation. The simulation output is consistent with the Simulink output.

After the simulation completes, coverage information is displayed for each assertion. By default, an assertion is considered covered if it was evaluated at least once.

```
# ** Info: Gathering coverage for          2 Simulink verify() calls.
#   Time: 0 ns   Scope: Req_4_dpi_pkg.VerifyInterfaceT.initVerifyInfo File: ../Req_4_dpi_pkg.sv Line: 159
# ** Error: Req_scenario_4:32:60: At step 'Check2' verify id 'Simulink:verify_sc4_off' Failed
#   Time: 750 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 237
# *****TEST COMPLETED (PASSED)*****
# ** Note: $finish      : ./Req_4_dpi_tb.sv(98)
#   Time: 2042 ns Iteration: 0 Instance: /Req_4_dpi_tb
# ** Info: Instance coverage for verify 'Req_scenario_4:32:39', coverpoint 'pass_cp': metric=100.00, at_least= 1 ( COVERED)
#   Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 259
# ** Info: Instance coverage for verify 'Req_scenario_4:32:60', coverpoint 'pass_cp': metric= 0.00, at_least= 1 ( UNCOVERED)
#   Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 259
# ** Info: Overall coverage for Req_4_dpi_verify_calls: metric= 50.00 ( UNCOVERED)
#   Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 266
# End time: 12:19:50 on Jun 08,2020, Elapsed time: 0:00:00
# Errors: 1, Warnings: 6
```

For additional information on running the HDL simulation, see “Verify Generated Component Against Simulink Data” on page 30-8.

Filter Assertions and Coverage Reports

Each generated error or warning displays a unique name identifying its origin. That number is the unique Simulink identifier (SID) of that block. For example, this log shows an error that was generated by a block with SID `Req_scenario_4:32:60`.

```
# ** Error: Req_scenario_4:32:60: At step 'Check2' verify id 'Simulink:verify_sc4_off' Failed
```

You might have several steps in a test sequence that utilize a `verify` assessment or several DPI components logging warnings from a simulation. In your test model, you can filter the generated output for specific `verify` checks by specifying the associated SID as a plus argument on the command line and equating the SID to `-1`. For example, to turn off all of the output and functional coverage for SID `Req_scenario_4:32:60`, enter this code at the HDL command line.

```
vsim -c -sv_lib ../Req_4 work.Req_4_dpi_tb +Req_scenario_4:32:60=-1
```

Adjust Functional Coverage Goals

You can use assertion blocks and `verify` statements to gather functional coverage during a SystemVerilog simulation. After generating SystemVerilog using the `uvmbuild` or `slbuild` functions, define coverage goals for each assertion. After a SystemVerilog simulation completes, view the results in the generated log file, or use a third party tool to extract the results. The default coverage goal is at least one passing execution of the assertion or `verify` call.

To increase the functional coverage goal for a specific assertion, specify the associated SID as a plus argument in the command line, and equate the SID to your coverage goal. For example, to increase the coverage goal of a `verify` statement with SID `Req_scenario_4:32:60` from the default of one to two passing checks, enter this code at the HDL command line.

```
vsim -c -sv_lib ../Req_4 work.Req_4_dpi_tb +Req_scenario_4:32:60=2
```

Verbose Mode

By default, the generated DPI component outputs an error when a functional coverage point is evaluated and fails. To see additional output generated by the functional coverage point, enter the argument `+VERBOSE_VERIFY` at the HDL simulation command line. This argument adds this additional information:

- UNTESTED - When the functional coverage point was not evaluated
- PASSED - When the functional coverage point was evaluated and the test passed

For example, when using ModelSim, enter this code at the command line.

```
vsim -c -sv_lib ../Req_4 work.Req_4_dpi_tb +VERBOSE_VERIFY
```

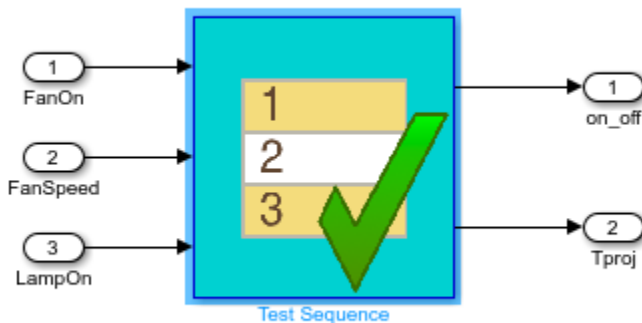
Trace Generated SystemVerilog Error Back to Simulink Source

After running a SystemVerilog simulation with a generated test sequence, your log file displays warnings and errors. To identify which block originated a specific warning or error output, use the `hilite_system` function.

For example, to highlight the block that generated a warning for SID Req_scenario_4:32:60, enter this code at the MATLAB command line.

```
hilite_system('Req_scenario_4:32:60');
```

This figure highlights the verify statement and the test sequence block that created the warning.



```
Check2
on_off = false;
verify(FanOn == false && LampOn == false,...
'Simulink:verify_sc4_off',...
'System should turn off above max on temp')
```

```
End
```

See Also

Blocks

Test Sequence | Test Assessment | Chart | Assertion | Check Dynamic Gap | Check Dynamic Range | Check Static Gap | Check Static Range | Check Dynamic Lower Bound | Check Dynamic Upper Bound | Check Input Resolution | Check Static Lower Bound | Check Static Upper Bound | Check Discrete Gradient

MATLAB Language Syntax

verify

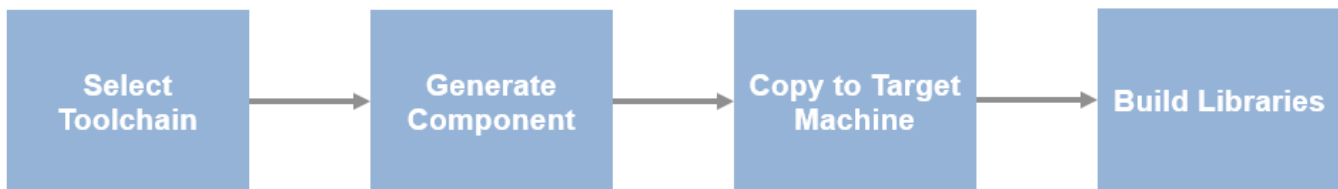
More About

- “Generating Functional Coverage in SystemVerilog from Simulink Test verify Calls” on page 32-96
- “Generate SystemVerilog DPI Component” on page 30-2
- “UVM Component Generation Overview” on page 26-2
- “verify Statements” (Simulink Test)

Generate Cross-Platform DPI Components

Use DPI component generation to export a Simulink subsystem to a C language component with a digital programming interface (DPI) for use in a Verilog or SystemVerilog simulation. You can customize DPI generation for ModelSim or Xcelium (Linux only), or you can generate a generic DLL.

When you generate the component from a Windows 64 host machine, you can also build the component libraries and run the simulation on a different operating system. If your target and host are not the same, you must port and build the shared libraries or HDL simulator projects manually. You cannot port a DPI component generated on a Linux machine to any other operating system.



Select Target Toolchain

When your target machine uses the same operating system as your host, you can select an installed compiler or request that the tools find a compiler automatically. If you want to generate a simulator project, or if you have no other compilers installed, select an HDL simulator for the same operating system as the host. However, if your target operating system is different from the host, you must select a target simulator and operating system.

Open your model, and on the **Apps** tab, click **HDL Verifier**. Select **DPI Component Generation** on the left pane, and on the **HDL Verifier** tab, click **C Code Settings**. The **Configuration Parameters** dialog opens on the **Code Generation** pane. The, under **Toolchain settings**, select a target **Toolchain**. This option specifies the target simulator and operating system where you run simulations. The supported cross-product toolchains are:

- Mentor Graphics ModelSim/Questasim (64-bit Windows) (available from Windows host only)
- Mentor Graphics ModelSim/Questasim (32-bit Windows) (available from Windows host only)
- Cadence Xcelium (64-bit Linux)
- Mentor Graphics ModelSim/Questasim (64-bit Linux)

To build a shared library for a different operating system, you must select one of the simulator options. You can then build the library on your target machine.

Generate Component

On the configuration parameters, under **Target Selection**, for **System target file** click **Browse**. Select `systemverilog_dpi_ert.tlc` from the list.

To generate your component and an optional test bench, follow “Generate SystemVerilog DPI Component” on page 30-2 from step 3 and on.

Package Files for Transfer

After generating the component, use the `packNGo` function to package the generated files and any required dependencies before copying them to the target machine.

```
load('subsystem_build/buildInfo.mat')
packNGo(buildInfo, 'minimalHeaders', false)
```

Where *subsystem* is the name of the Simulink subsystem used for DPI generation. The result is a zip file named *subsystem.zip*.

Copy to Target Machine

To use your generated component on a different operating system, you must copy the generated package file to the target machine, unzip it, and build the libraries there.

- 1 Copy the generated *subsystem.zip* file from the host machine to the target machine. The *.zip* file is located in the same folder as your model. The ModelSim *.do* file or Xcelium *.sh* file is included in the *.zip* file.
- 2 Unzip the file into a folder of your choice.

Build Libraries

When you generate the component on the host machine, the libraries are built for that operating system. To port the component to a different operating system, you must build the components manually on the target machine. To build your simulator project or generic shared library, find your target operating system and HDL simulator in the table and follow the instructions.

Target Operating System	HDL Simulator	Build Instructions
Windows 32	ModelSim	<ul style="list-style-type: none"> • Check that the <code>gcc_ver_mingw32</code> library is installed in your ModelSim installation folder. This compiler is available when you install ModelSim. Install the compiler before building the component. • Start the ModelSim HDL simulator. • In the command window, change to the folder where you unzipped the generated files. • Build the project with this command: <pre>do subsystem.do</pre> <p>Where <i>subsystem</i> is the name of the Simulink subsystem used for DPI generation.</p>

Target Operating System	HDL Simulator	Build Instructions
Linux	ModelSim	<ul style="list-style-type: none"> Start the ModelSim HDL simulator. In the command window, change to the folder where you unzipped the generated files. Build the project with this command: <code>do subsystem.do</code> <p>Where <i>subsystem</i> is the name of the Simulink subsystem used for DPI generation.</p>
	Xcelium	<ul style="list-style-type: none"> In a terminal shell with Xcelium on the path, build the project with this command: <code>sh subsystem.sh</code> <p>Where <i>subsystem</i> is the name of the Simulink subsystem used for DPI generation.</p>

Limitations

Cross-platform DPI generation does not support model referencing.

See Also

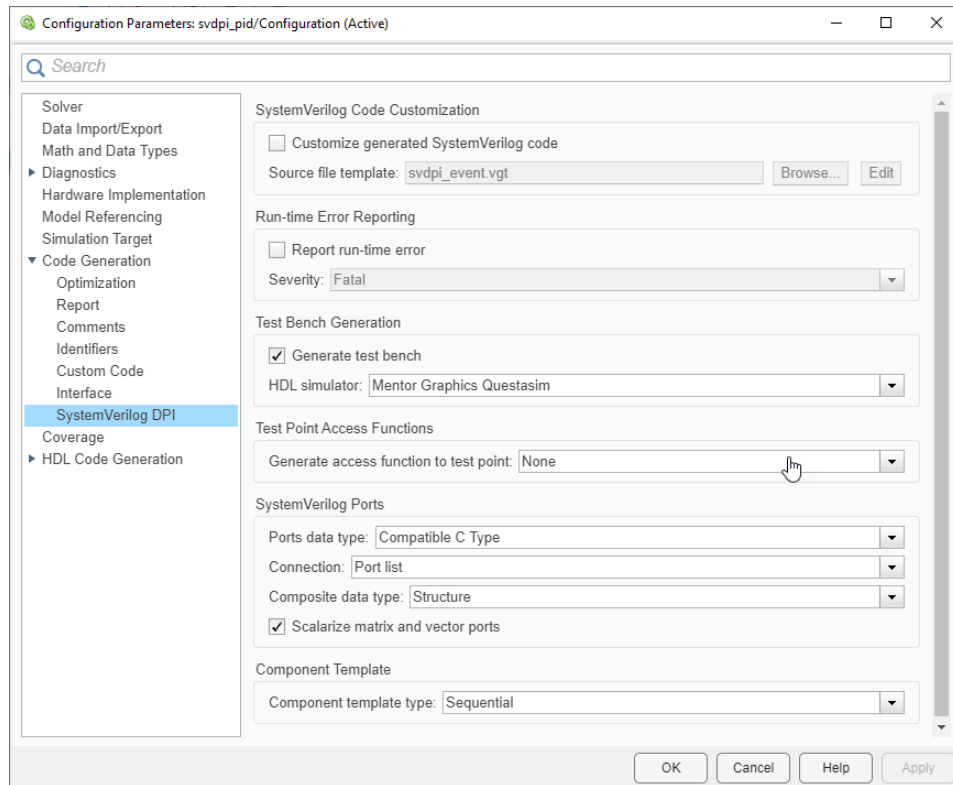
packNGo

Related Examples

- “Use Generated DPI Functions in SystemVerilog” on page 30-9
- “Verify Generated Component Against Simulink Data” on page 30-8

Context-Sensitive Help for Generated SystemVerilog DPI Component

SystemVerilog DPI Pane



In this section...

- “SystemVerilog DPI Overview” on page 31-2
- “Customize SystemVerilog generated code” on page 31-3
- “Source file template:” on page 31-3
- “Report run-time error” on page 31-3
- “Severity” on page 31-3
- “Generate test bench” on page 31-4
- “HDL simulator” on page 31-4
- “Test point access” on page 31-4
- “Ports data type” on page 31-5
- “Connection” on page 31-5
- “Composite Data Type” on page 31-5
- “Scalarize matrix and vector ports” on page 31-6
- “Component template type” on page 31-6

SystemVerilog DPI Overview

Specify options for exporting a Simulink algorithm (model or subsystem) with a DPI interface for Verilog or SystemVerilog Simulation. You can wrap generated C code with a DPI wrapper that communicates with a SystemVerilog thin interface function in a SystemVerilog simulation.

This feature is available in the Model Configuration Parameters dialog. You must have an Embedded Coder license to use this feature.

Customize SystemVerilog generated code

Indicate that you want to customize the generated SystemVerilog code.

Settings

Default: Off

On
Customize generated SystemVerilog code

Off
Do not customize generated SystemVerilog code

Dependencies

You must enter a template file in **Source file template:** if you want the generator to include customized code.

Source file template:

Specify the file name and location of the template you want to use for customizing the generated SystemVerilog code. You may use one of the templates supplied by HDL Verifier, or you may specify your own template file with the following conditions:

- The file must be on MATLAB path and be searchable.
- The file must have a `.vgt` extension.

Report run-time error

Select this parameter to export run-time errors from a Simulink execution to the DPI and UVM simulation environment.

Severity

Select the severity for run-time errors.

Default: Fatal

Info

Set the run-time error to a ``uvm_info` macro for UVM components or a `$display` statement for DPI components.

Warning

Set the run-time error to a ``uvm_warning` macro for UVM components or a `$warning` statement for DPI components.

Error

Set the run-time error to a ``uvm_error` macro for UVM components or a `$error` statement for DPI components.

Fatal

Set the run-time error to a ``uvm_fatal` macro for UVM components or a `$fatal` statement for DPI components.

Dependencies

To enable this parameter, select **Report run-time error**.

Generate test bench

Indicate that you want to generate a test bench for the DPI component.

Settings

Default: Off



On

Create a test bench for the generated DPI component



Off

Do not create a test bench for the generated DPI component

HDL simulator

Select the HDL simulator to use when simulating the testbench in MATLAB

Default: Mentor Graphics Questasim

Mentor Graphics Questasim

Set the HDL simulator to Mentor Graphics Questa.

Cadence Xcelium

Set the HDL simulator to Cadence Xcelium.

Synopsys VCS

Set the HDL simulator to Synopsys VCS.

Vivado Simulator

Set the HDL simulator to Xilinx Vivado.

Test point access

Select the type of test point access functions to generate in the SystemVerilog DPI component.

Settings

Default: None

None

The tool does not generate test point access functions.

One function per Test Point

The component includes a separate access function for each signal.

```
DPI_Name_TestPoint(inputchandle objhandle,inout real Name);
```

One function for all Test Points

The component includes a single access function that returns values for all test points.

```
DPI_TestPointAccessFcn(inputchandle objhandle,input real Name1,inout real Name2);
```

Ports data type

Select the SystemVerilog data type that will be used for ports that have fixed-point data.

Settings

Default: Compatible C Type

Compatible C Type

Generate a compatible C Type interface for the port.

Bit Vector

Generate a Bit Vector Type interface for the port.

Logic Vector

Generate a Logic Vector Type interface for the port.

Connection

Select how signals are connected when the module is instantiated.

Settings

Default: Port list

Port list

Generate a SystemVerilog module with a port list in the header, representing its interface.

Interface

Generate a SystemVerilog interface, and a module using that interface.

Composite Data Type

Choose how the SystemVerilog ports are generated when your Simulink model includes a port which is a `Nonvirtual` bus or a `complex` data type. Choose between interfaces with `struct` data types or flattened SystemVerilog ports.

Settings

Default: Flattened

Flattened

Generate a SystemVerilog module with flattened ports.

Structure

Generate a SystemVerilog module with `struct` data type ports.

Scalarize matrix and vector ports

Choose how the SystemVerilog ports are generated when your Simulink model includes a port which is an `array` or `matrix` data type.

When selecting this box, each element in the array or matrix creates a scalar port in the generated SystemVerilog.

When clearing this box, the generated SystemVerilog ports preserve the `array` or `matrix` as defined in Simulink.

Component template type

Select a template for SystemVerilog-DPI generation.

- `Sequential` - Creates a registered design, with a clock and reset port. This is the default behavior.
- `Combinational` - Creates a combinational design, with no clock and reset ports. When selecting this option, the outputs immediately reflect changes in the inputs.

HDL Verifier Examples

Comparing HDL and Simulink Code Coverage Using Cosimulation

This example shows how to achieve complete code coverage of an HDL cruise controller design using Simulink® and an HDL Simulator.

Introduction

The HDL code associated with this model is generated via HDL Coder™ from a Simulink behavioral model of the cruise controller. A test bench model is provided to verify the correctness of the HDL code by comparing the output of the HDL cosimulation block with that of the original behavioral block. The testcases in the test bench model are generated through Simulink Design Verifier™ from the original behavioral model for achieving complete model coverage. This example shows that those automatically generated testcases also achieve complete HDL code coverage. You do not need Simulink Design Verifier installed to run this example.

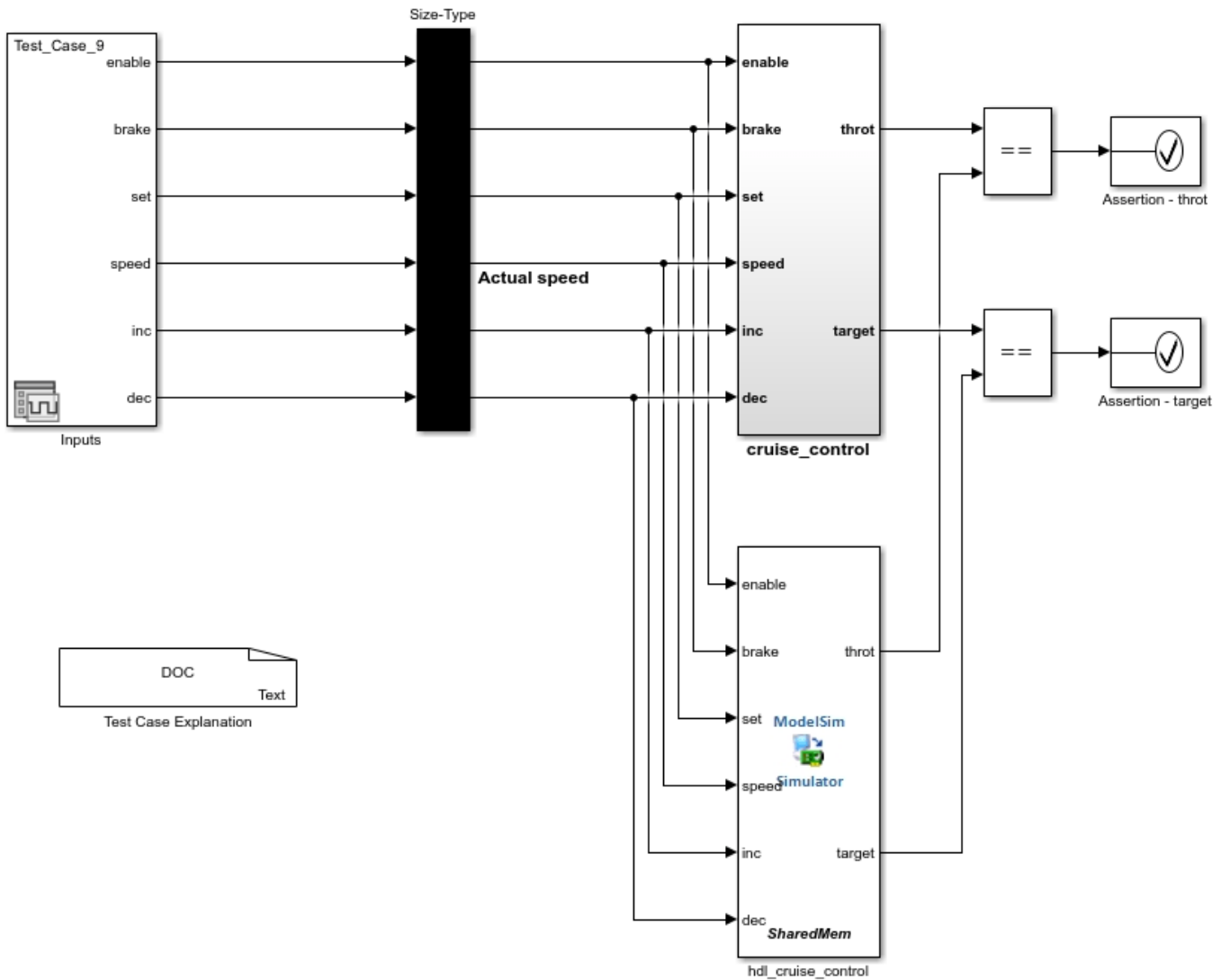
Open the Simulink Model

If you are using ModelSim or QuestaSim, the model `cruise_control_modelsim.slx` should be open. If you are using Xcelium, close the ModelSim model and open the model `cruise_control_incisive.slx`.

Note that the code coverage function is an optional feature in ModelSim PE. Make sure that your version of ModelSim has the proper code coverage license to run this example.

```
% For ModelSim:
modelName = 'cruise_control_modelsim';
open_system(modelName);

% For Xcelium:
modelName = 'cruise_control_incisive';
open_system(modelName);
```

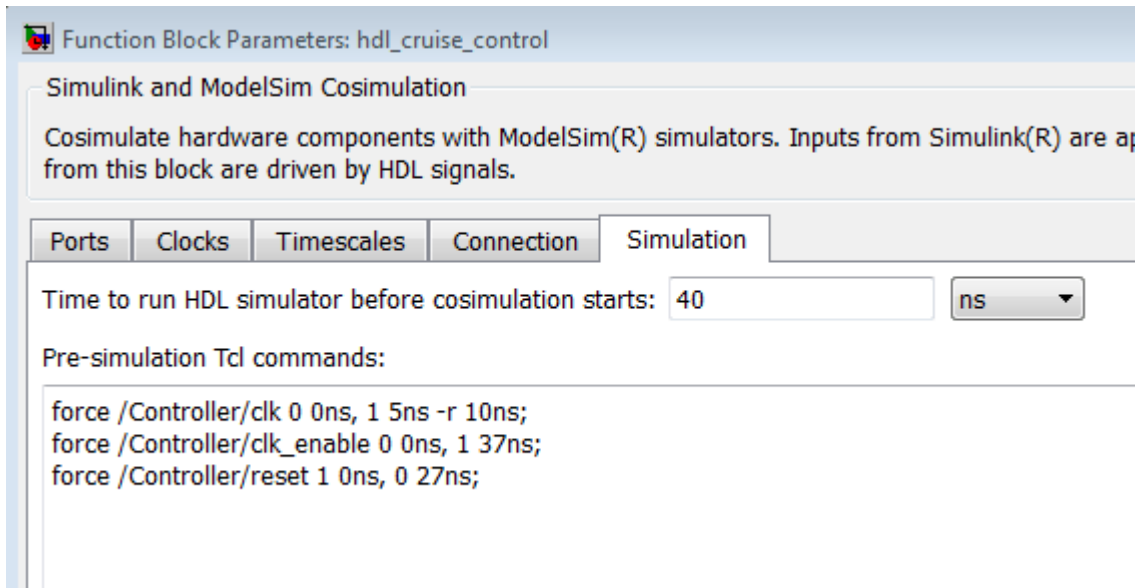
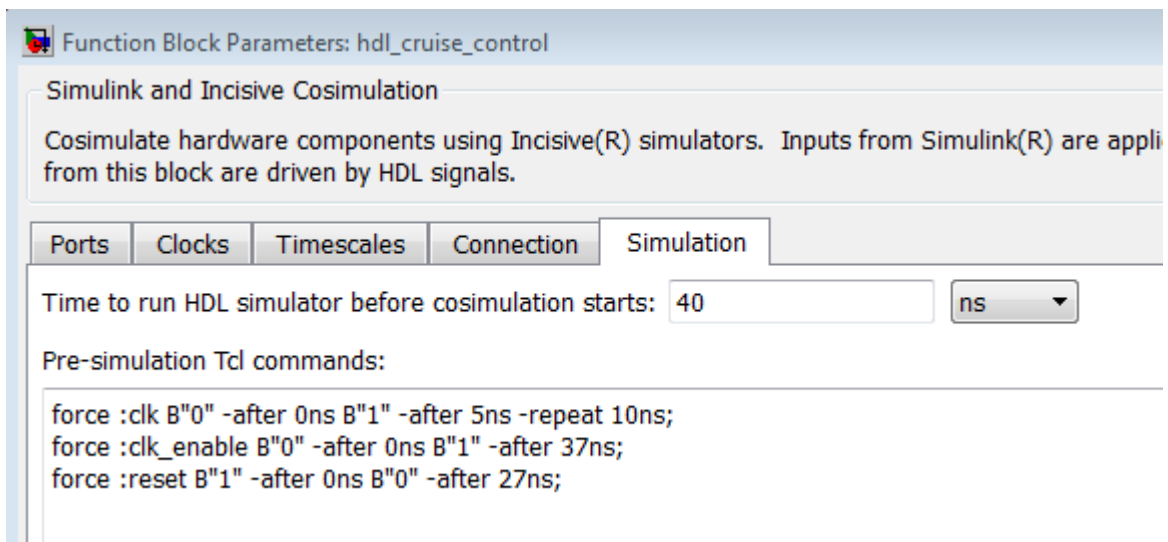


Copyright 2011 The MathWorks, Inc.

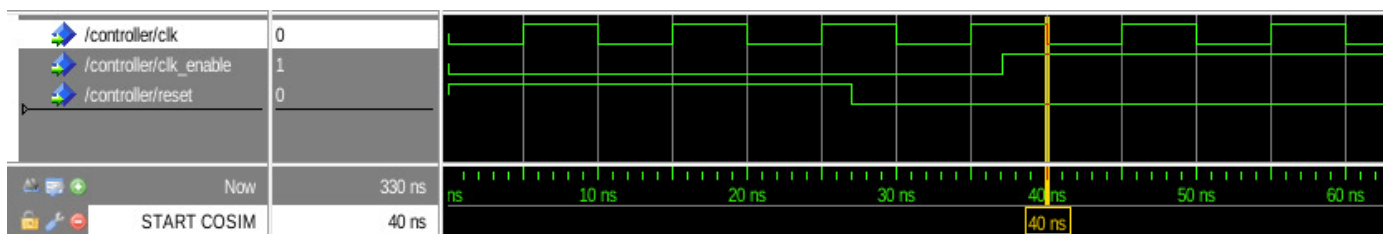
Setting up Clocks, Resets, and Clock Enables in the HDL Simulation

The clock, reset, and clock enable signals are not part of the Simulink simulation. We define drivers for them using the "Pre-simulation Tcl commands" in the "Simulation" tab of the cosimulation block mask. We also set the "Time to run HDL simulator before cosimulation starts" such that the HDL has successfully gotten out of reset before any input and output data values are exchanged.

For ModelSim:

**For Xcelium:**

The resulting waveforms in HDL:



The Tcl force commands are used to generate waveforms for the control signals:

- 1 The clock period is 10 ns

- 2 The clock enable signal is active at 37 ns.
- 3 The reset signal is asserted from 0 to 27 ns.

The cosimulation starting time should be aligned with the falling edge of the HDL clock to avoid a race condition since the HDL signals change their values at the rising edge of the HDL clock. Therefore, the value of this parameter should be an integer multiple of the 10 ns clock period.

We also want to run the HDL design past reset, but stop before the first active clock edge enabled by `clk_enable`. This is to match the behavioral block, which updates its internal states right after the simulation starts.

Based on the above considerations, the option "Time to run HDL simulator before cosimulation starts" is set to 40 ns. When running each testcase, the "Pre-simulation Tcl commands" are applied in the HDL simulator first. Then the HDL simulator advances its time by 40 ns to apply the reset, clock, and `clk_enable` signals before the cosimulation starts.

Launch HDL Simulator for Cosimulation

For Modelsim:

```
% Commands to compile and invoke Modelsim with code coverage enabled.
tclCmds = {
    'vlib work',...                               % Create ModelSim library
    'vcom +cover cruise_hdlsrc/PI_Controller.vhd',... % Compile VHDL code with code coverage
    'vcom +cover cruise_hdlsrc/Controller.vhd',... % Compile VHDL code with code coverage
    'vsimulink -coverage work.Controller',... % Load simulation
    'puts "Ready for cosimulation..."',...
};
% Now we launch the HDL simulator and wait for it to be ready.
vsim('tclstart',tclCmds);
disp('Waiting for HDL simulator to start ...');
processid = pingHdlSim(240);
disp('HDL simulator is ready to cosimulate.');
```

For Xcelium:

```
% Commands to compile and invoke Xcelium with code coverage enabled.
tclCmds = {
    'exec xmvhdl -64bit -v93 cruise_hdlsrc/PI_Controller.vhd',... % Compile VHDL code
    'exec xmvhdl -64bit -v93 cruise_hdlsrc/Controller.vhd',... % Compile VHDL code
    'exec xmelab -64bit -coverage all -vhdl_time_precision 1ns -access +wc Controller',... % Elab
    'hdlsimulink -covoverwrite Controller',... % Load simulation
    'puts "Ready for cosimulation..."',...
};
% Now we launch the HDL simulator and wait for it to be ready.
nclaunch('tclstart',tclCmds,'runmode','CLI');
disp('Waiting for HDL simulator to start ...');
processid = pingHdlSim(240);
disp('HDL simulator is ready to cosimulate.');
```

Run Simulation

There are nine testcases in the test bench model. This example runs this model with all testcases to produce the code coverage result. After finishing each cosimulation session, there is no need to restart the HDL simulator since the HDL signal is reset properly at the beginning of each simulation. After each simulation, a short pause is added to ensure time for the HDL simulator to update the coverage result before the next iteration.

```

% Run the cosimulation
for k = 1:9
    get_param([modelName '/Inputs'],'ActiveScenario'); % get ActiveScenario parameter for changing
    set_param([modelName '/Inputs'],'ActiveScenario',['Test_Case_' num2str(k)]) % set a Test case
    sim(modelName); % Run simulation
    pause(5); % pause for writing coverage to database
end

```

Observe Code Coverage Result

The HDL simulator accumulates coverage as we iterate through all the testcases. When the simulation is finished, 100% code coverage is achieved.

For ModelSim: You can examine the coverage result in the "Coverage" tab of the UI.

The screenshot shows the 'Questa Instance Coverage' window for instance '/controller'. It displays a table with the following data:






Instance	Branches	Expressions	Statements	Toggle	Total
Total	100%	100%	100%	100%	100%
controller	100%	100%	100%	100%	100%
u_Pi_Controller	100%	100%	100%	100%	100%

For Xcelium: We dump the coverage results and use the "imc" tool to visualize the results.

```

% Dump and visualize the coverage results
tclHdlSim('coverage -dump test');
system('imc -gui -load test &');

```

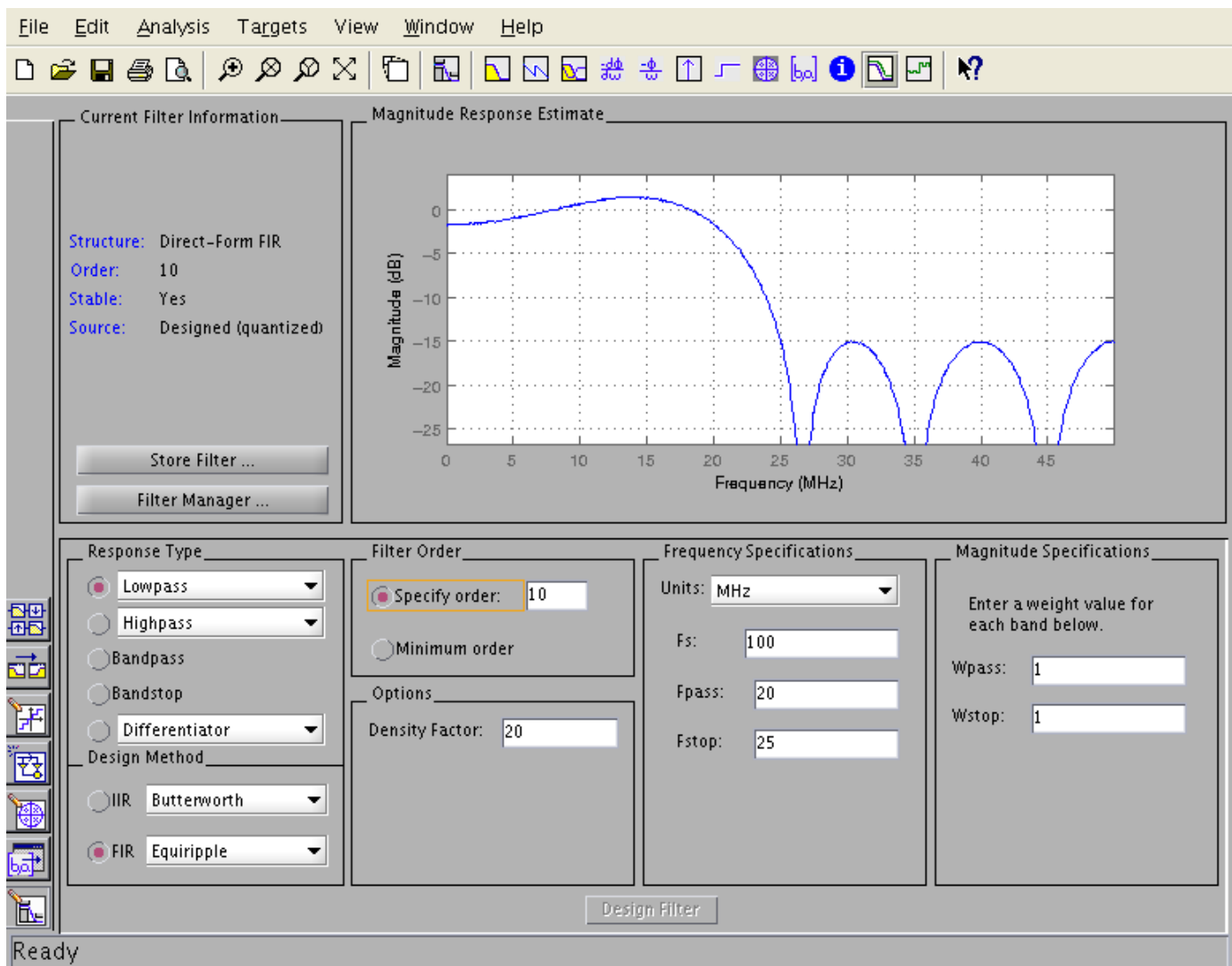
		Details	Verification Metrics	
		Metrics	Source	Attributes
Ex	UNR	Name	Overall Average Grade	Overall Covered
		Overall	 100%	122 / 122 (100%)
		Code	 100%	122 / 122 (100%)
		Block	 100%	56 / 56 (100%)
		Statement	n/a	0 / 0 (n/a)
		Expression	 100%	24 / 24 (100%)
		Toggle	 100%	42 / 42 (100%)
		FSM	n/a	0 / 0 (n/a)

Testing a Filter Component in MATLAB

This example shows how MATLAB® can be used as a test bench for an HDL component. We compile an HDL low pass filter and then test it using `matlabtb`.

A temporary directory is used for the compiled and elaborated HDL files. Once a snapshot is ready, we start a Cadence® Xcelium® simulator session. (You must have the Xcelium simulator executables on your PATH.) We use a shared memory connection between MATLAB the HDL simulator, so both must be on the same computer.

In this example, a Verilog low pass filter has been designed and generated with the MathWorks™ Filter Design HDL Coder™ product and our job is to test it by comparing it to the MATLAB filter. We will not be running the filter designer in this example. The filter has a sample time of 10 ns (a sample rate of 100 MHz) with a passband $F_{pass} = 20$ MHz and a stopband $F_{stop} = 25$ MHz. The Verilog filter has a delay of two samples.



The file **lowpass_filter.v** contains the low pass filter generated by Filter Design HDL Coder.

The file `filter_tb_incisive.m` contains the MATLAB test bench.

```
srcfile = fullfile(matlabroot, 'toolbox', 'edalink', 'extensions', 'incisive', 'incisivedemos', 'Filter
```

Create Project Directory

We create a temporary working directory in which the project will be compiled.

```
projdir = tempname;
warnstatus = warning('off', 'MATLAB:MKDIR:DirectoryExists');
mkdir(projdir);
warning(warnstatus);
```

Start the MATLAB Server

We start the MATLAB server, `hdldaemon`, such that it uses shared memory communication.

```
hdldaemon;
```

Specify Tcl Commands

Next we specify the Tcl commands to execute in the HDL simulator before the simulation is run. The following lists of commands will execute in a Tcl shell. The commands will compile and elaborate the project and then launch `xmsim` through the `hdlsimmatlab` Tcl command. All commands preceded with `-input` are passed to `xmsim` and are executed in the `xmsim` Tcl shell. These commands will :

- change to the working directory
- compile the verilog filter
- elaborate the filter and turn on read write access to the ports
- start `xmsim` with MATLAB test bench support by calling `hdlsimmatlab`, remaining commands are executed in `xmsim`
- schedule the MATLAB function `filter_tb_incisive` to be called every 10 ns
- enable the clock
- reset the module after 22 ns
- drive a 10 ns clock
- initialize the filter input to 0

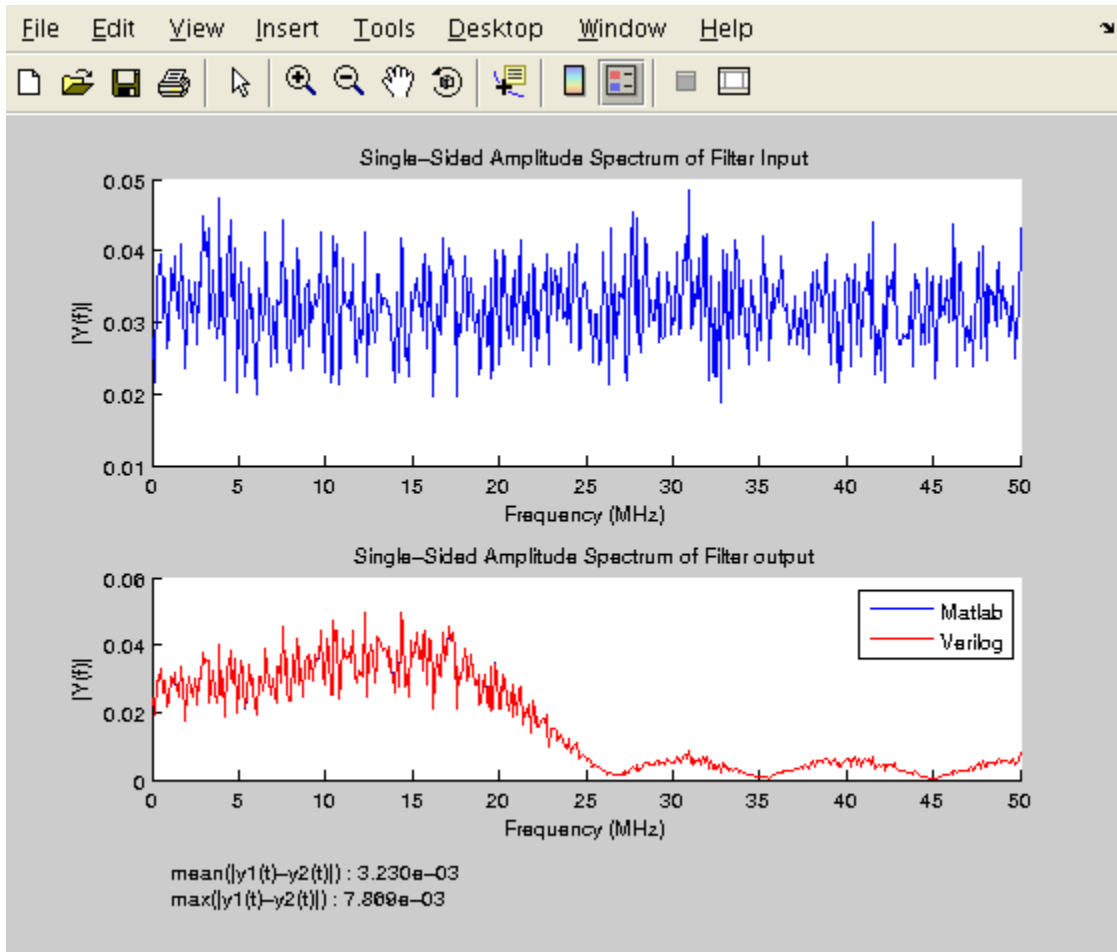
```
tclcmd = { ['cd ',projdir],...
           ['exec xmvlog -64bit ' srcfile],...
           ['exec xmelab -64bit -access +wc lowpass_filter',...
           ['hdlsimmatlab -gui lowpass_filter ', ...
           ' -input "{@matlabtb lowpass_filter 10ns -repeat 10ns -mfunc filter_tb_incisive}"',
           ' -input "{@force lowpass_filter.clk_enable 1 -after 0ns}"',...
           ' -input "{@force lowpass_filter.reset 1 -after 0ns 0 -after 22ns}"',...
           ' -input "{@force lowpass_filter.clk 1 -after 0ns 0 -after 5ns -repeat 10ns}"',...
           ' -input "{@deposit lowpass_filter.filter_in 0}"',...
           ]};
```

Start the Xcelium® Simulator

Now we start the HDL simulator via the `nclaunch` command. The `'tclstart'` property causes the specified Tcl commands to be run at startup. Once the HDL simulator is launched, begin the simulation using the `run` command in the `xmsim` console, specifying the appropriate simulation time. For example type `run 100000`.

```
nclaunch('tclstart',tclcmd);
```

At this point a MATLAB figure with two subplots will open. The upper plot shows the spectrum of the input signal. The lower plot shows the spectrum of the HDL filter output overlaid with the spectrum of the MATLAB filter output. You will have to zoom in to see the difference between the output spectrums. The running mean and maximum of the absolute error in the time domain is also shown below the plots. The figure will be updated every 1024 samples.



This concludes this example.

Be sure to quit the HDL simulator once you are done with this example as each time the example is run, a new simulator session is started. Also keep in mind that this example created some temporary files in a temporary directory.

Frame-based Scrambler Using Communications Toolbox

This example illustrates the validation of an HDL implementation of a 6-order scrambler. A scrambler is used in communication systems to randomize transitions in the transmitted signal by shuffling the bits. One purpose of scrambling is to reduce the length of strings of 0s or 1s in a transmitted signal, since a long string of 0s or 1s may cause transmission synchronization problems. Scrambling may also be used as a cheap encryption technique. This example consists of two models. The first model (`scrambler_frame`) validates the HDL implementation and the second model (`scrambler_fsk`) uses the HDL scrambler as part of a communication channel.

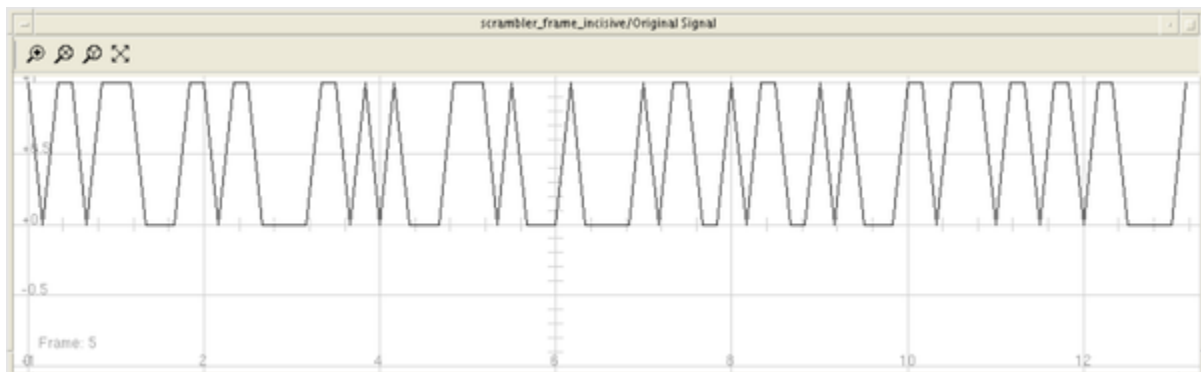
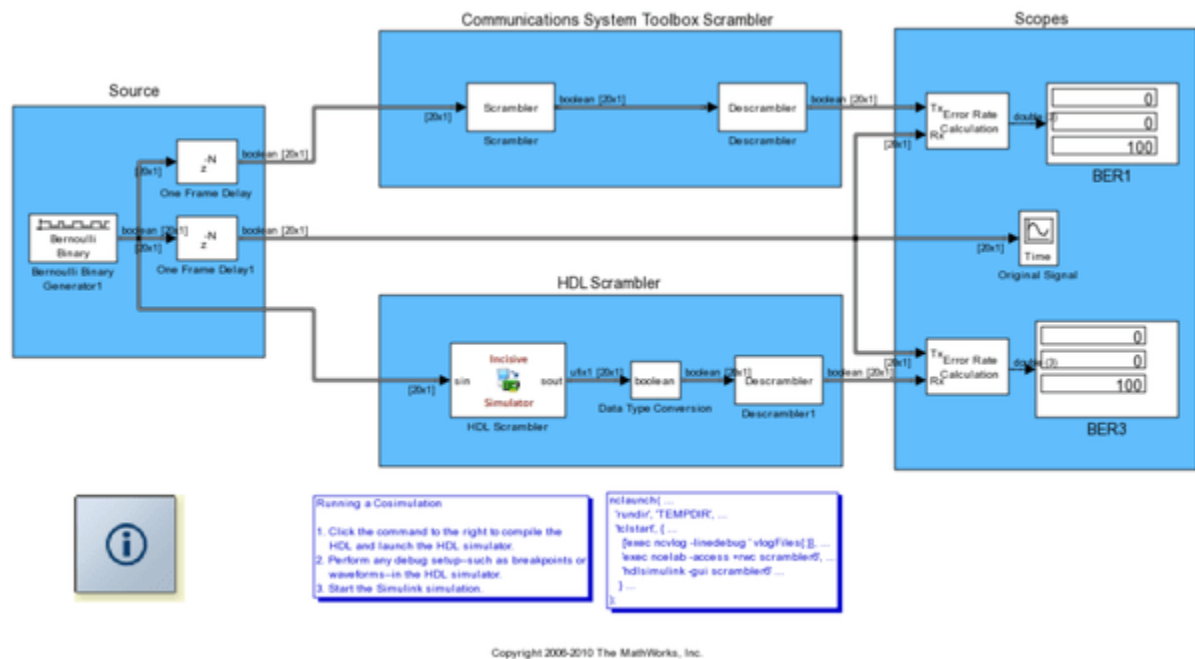
You need the following products to run this example:

- MATLAB
- Simulink
- Communications Toolbox
- HDL Verifier
- Cadence Xcelium software

HDL Implementation and Validation

The `scrambler_frame` model illustrates the validation of an HDL implementation of a 6-order scrambler. The handwritten HDL code is intended to replicate the behavior of the Scrambler block in Communications Toolbox™. The EDA Simulator Link™ software is used here as a validation tool to test the functional equivalence of the HDL scrambler to the Communications Toolbox scrambler.

The model generates a binary sequence and simultaneously drives it to the HDL implementation and the original Simulink® block. Two copies of the Descrambler block from Communications Toolbox are used to reconstitute the original data stream from the outputs of the two scramblers. The outputs are compared to the input data sequence to detect any discrepancies that may be introduced by differences between the scramblers using the Error Calculation block. Because the HDL Cosimulation block always delays the data by one frame, we delay the data in all other branches so that the comparison will be valid.

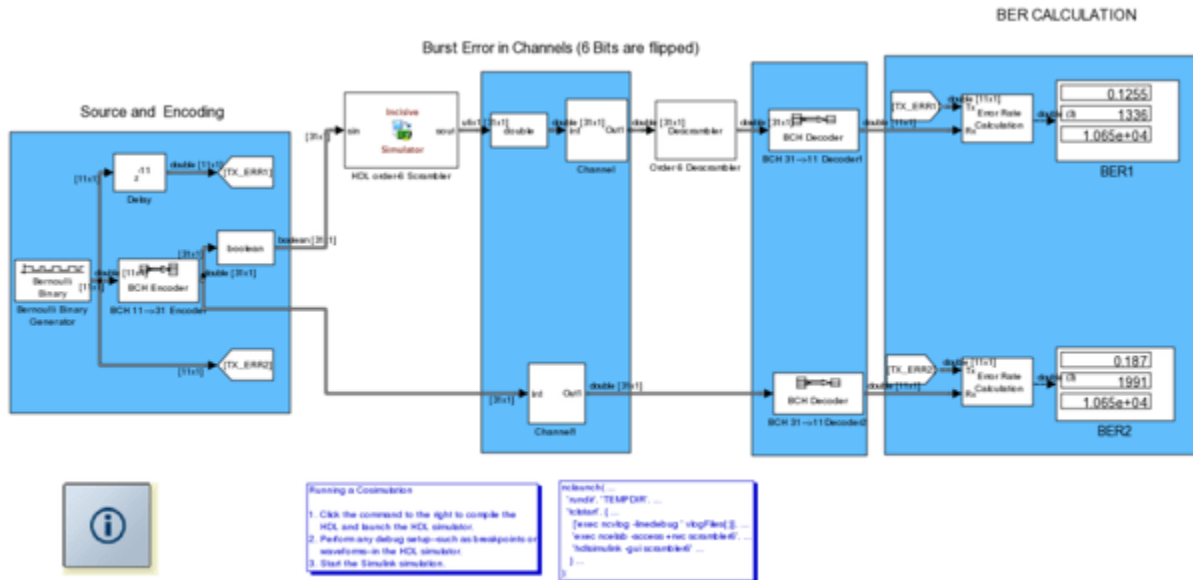


Channel Encoding with an HDL Scrambler Block

The `scrambler_fsk` model shows the use of a Scrambler as part of a communication channel. The block of interest is a 6-order scrambler implemented in HDL and cosimulated with Simulink by the HDL Verifier software. It has previously been verified to be functionally equivalent to the Scrambler block in the Communications Toolbox. The Descrambler block is from the Communications Toolbox and is also of order-6. The channel uses 11 bits per frame and BCH 11-->31 bits encoding (and the opposite BCH 31-->11 bits decoding). This encoding method is theoretically able to correct burst errors with up to 5 flipped bits. The data is passed through a channel with burst errors of 6 consecutive bits (i.e., 6 out of 11 bits in each frame will be flipped).

The data is then transmitted using 2-ary FSK modulator and demodulator blocks and a Gaussian channel with a very high signal-to-noise ratio, so we assume that almost all errors are burst errors and not Gaussian white noise errors. The locations of the flipped bits are randomly chosen in each frame. Since the current encoding configuration cannot fix all errors when 6 bits are flipped, there will be some errors after decoding the data and the BER (Bit Error Rate) will not be zero. The model contains 2 channels: one includes Scrambler / Descrambler blocks and the other one does not. In the

case of our model, it is clear that the BER is improved (becomes lower) when we use the Scrambler / Descrambler blocks as part of our communication channel.



Copyright 2006-2009 The MathWorks, Inc.

Manchester Receiver

The Manchester Receiver example shows how to use the HDL Verifier™ to design, test, and verify a VHDL Manchester Receiver model with clock recovery capabilities.

Background on Manchester Encoding

Transmission of digital data frequently requires some form of modulation to overcome limits in a physical signal channel. One technique used for modulating digital data is Manchester Encoding. This technique has the following useful characteristics:

- The transmit clock signal can be easily extracted from the received data.
- The encoded signal never produces frequency components near DC, regardless of the data, which is useful for transmission over channels that require AC coupling.
- The encoding circuit is very simple and stateless.

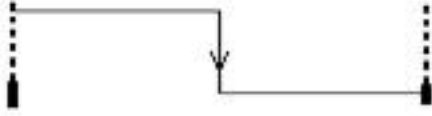
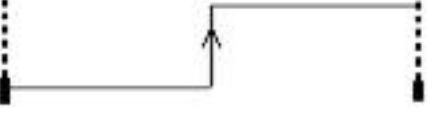
On the negative side, Manchester coding requires substantial bandwidth (above the Shannon limit), which tends to limit its usefulness in wireless applications. However, for connected applications such as short haul Optical fiber and Ethernet, it is frequently a good solution.

The following sections discuss:

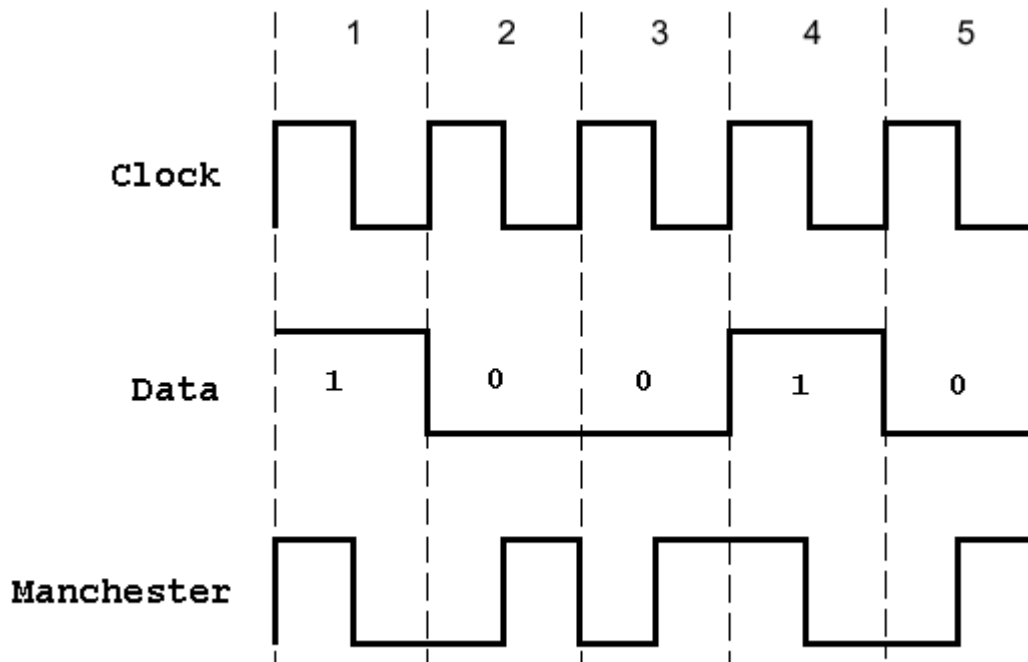
- The Encoding
- The Receiver
- Decoding with Inphase and Quadrature Convolution

The Encoding

Manchester Encoding involves a transmitter that encodes clock and data signals in a synchronous bit stream, such that each bit represents a signal transition. The following table shows how each bit setting is defined for an encoding:

Bit Setting	Transition	Encoded Waveform
1	1 to 0	
0	0 to 1	

Transitions in the Manchester Encoding occur at the center and beginning of each bit. The transition at the center is defined by the bit value, while the transition at the beginning is dependent on the value of the previous bit. Consider the following diagram:



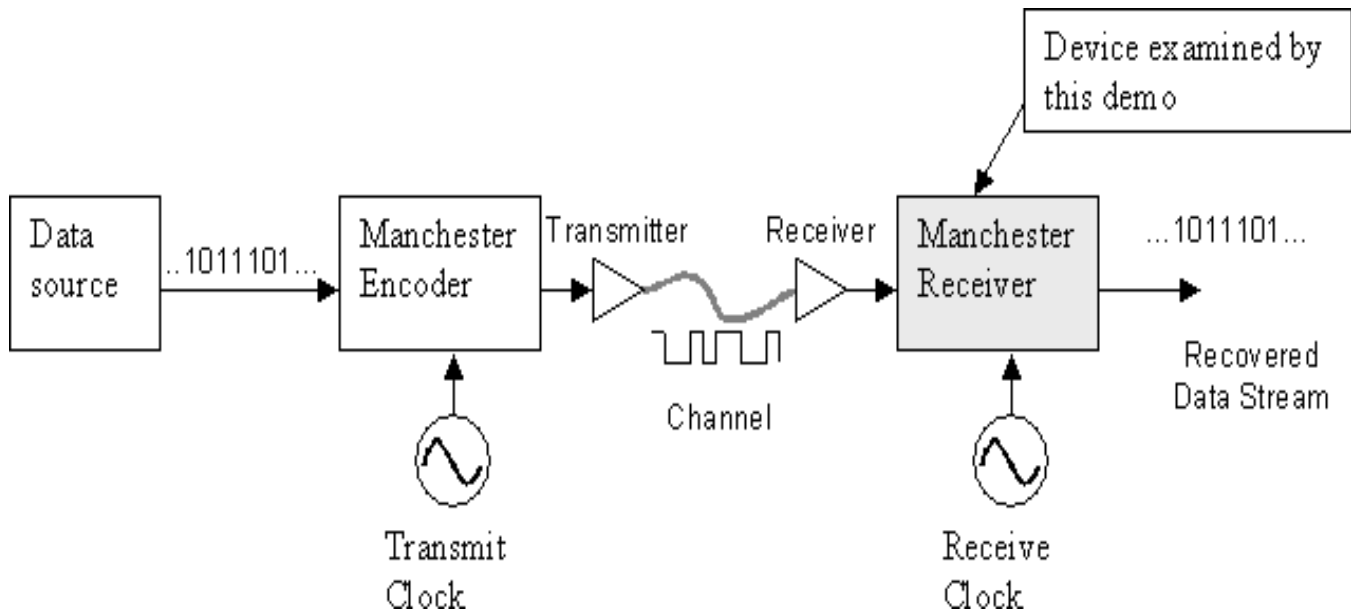
As the preceding Manchester encoded signals shows:

- The value of 1 for the first bit forces a high-to-low transition at the center of that bit.
- The value of 0 for the second bit forces a low-to-high transition at the center of that bit and, because the first bit transitioned from high-to-low, no transition occurs at the start of that bit.
- The value of 0 for the third bit forces a low-to-high transition at the center of that bit and because the second bit transitioned from low-to-high, a high-to-low transition occurs at the start of that bit.
- The value of 1 for the fourth bit forces a high-to-low transition at the center of that bit and, because the third bit transitioned from low-to-high, no transition occurs at the start of that bit.
- The value of 0 for the fifth bit forces a low-to-high transition at the center of that bit and, because the fourth bit transitioned from high-to-low, no transition occurs at the start of that bit.

The Receiver

A device that receives the encoded bit stream is responsible for decoding the bit stream by separating the clock and data information. In most cases, the receiver must retrieve the original data stream by using only the encoded signal. This simplifies the communications channel, but means the receiver must overcome the following:

- Differences between the clock used to encode the signal and the clock in the receiver (see the figure below).
- The two clocks can be close in frequency, but small frequency errors occur.
- The phase between the clocks will be arbitrary.



The Manchester Receiver example validates the computations performed by a Manchester Receiver device that is modeled in VHDL and simulated in ModelSim®. Numerous approaches are available for implementing a Manchester receiver. This example uses a Delay Lock Loop (DLL) that requires the receiver to use a clock that is very close in frequency to the transmit clock. This results in a simple clock recovery circuit that has a limited frequency lock range.

The receiver over-samples the received data stream at 16 times the data rate. Thus, the receive clock must have a nominal period of 1/16th the data period. To compensate for minor differences between the transmitting and receiving clocks or drifts in the channel delay, the receiver adjusts its data period by up to one clock cycle (+/-) per data period. Thus, the receiver can use 15, 16 or 17 clock cycles to recover the data encoded from the incoming sampled signal. For example, if the receiver clock is slightly faster than the transmitter clock (frequency error), the receive cycle occasionally needs to add an extra clock cycle to compensate.

Large sudden phase errors, such as those that occur at startup time, require multiple data periods to acquire a good lock on the signal. By limiting the maximum phase correction to 1/16th of the total data period, the receiver can be slow to correct large phase errors.

Decoding with Inphase and Quadrature Convolution

Decoding a received Manchester signal can occur in several ways, but the approach taken in this example is to consider Manchester Encoding as a digital phase modulation with two symbols: +180 and -180 degrees. By convolving the incoming signal with a reference inphase (I) and quadrature (Q) waveform at the modulation frequency, it is possible to extract the data and retrieve information about any phase errors in the received waveform. After one data cycle, the receiver computes two values (referred to as *Isum* and *Qsum* in the VHDL code), which are measurements of the I/Q convolution value. The receiver then decodes the values to predict:

- The original transmitted data value for the cycle
- An estimate of the phase error between the incoming signal and the receiver's data period

A critical aspect of this design is the interpretation of the I/Q convolution values. At the end of a data receive cycle, the receiver translates the I/Q values into an estimate of the transmitted data and

phase error. One way to present this information is to show these interpretations as plots versus measured I/Q values.

Data is considered invalid if the measured I and Q are completely ambiguous about the encoded data value.

In a similar way, you can generate an I/Q mapping of the phase correction value in plot format. Such a plot gives a visual representation of the decoding block. In practice, the details of this mapping have strong impact on the stability and performance of the Manchester receiver.

In the ideal case where the receiver is perfectly locked to the incoming waveform, the receive cycle is 16 cycles long and the measured I/Q convolution values are fairly easy to interpret. However, data cycles that are 15 or 17 cycles long create some bias in the measurement of the IQ convolution. It is possible to customize the I/Q measurement during these cycles, but that would increase the size and complexity of the receiver. Instead, the data acquisition cycle is extended or shortened with no change in decoding the resulting values. However, this decoder bias can create problems with dithering or reduced noise immunity.

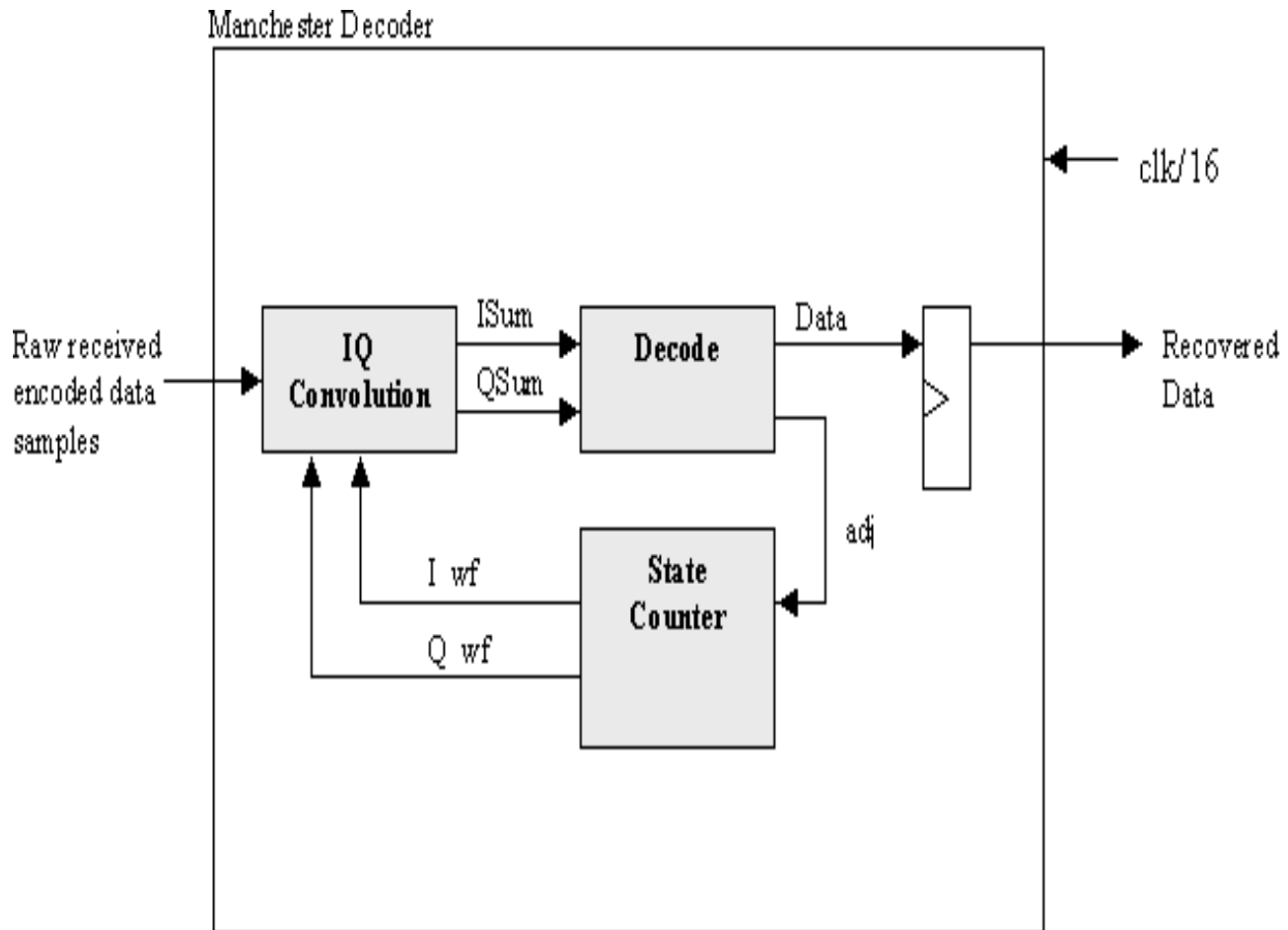
VHDL Implementation of a Manchester Receiver

The focus of this example is a VHDL implementation of a Manchester Receiver. Decoding a Manchester encoded signal presents several challenges, the most prominent of which is clock recovery. The clock is embedded in the received signal and must be extracted to reproduce the original data stream. The figure below shows the example design, which is divided into three main sections of VHDL code:

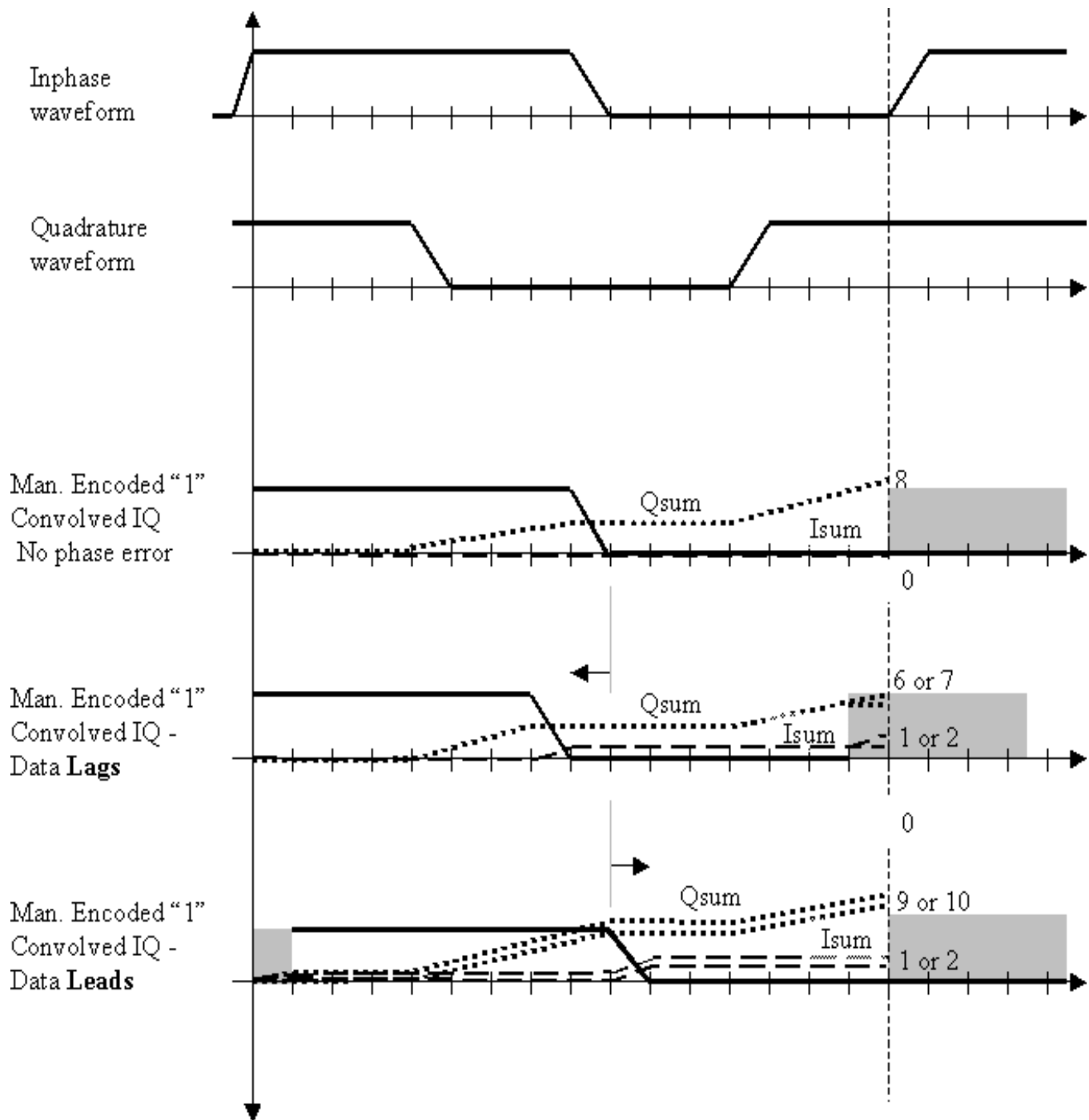
IQ convolver: Samples the received signal and computes the convolution for the inphase (I) and quadrature (Q) waveforms. For each waveform, the computation is implemented as the sum of XOR operations on the sample and decoded waveform received from the state counter.

Decoder: Models a combinatorial circuit that interprets the results of the I/Q convolver.

State counter: Generates the I/Q waveforms that are convolved with received signals, taking into account phase errors (lags and leads), as necessary. The phase of the I/Q generator is adjusted to match the incoming Manchester encoded waveform. To accomplish the necessary adjustment, at the beginning of a new cycle, the state counter checks an adjustment value, *adj*, and then changes the period of the next I/Q cycle. This adjustment value is limited to adding or removing a single clock period from the 16 periods that are nominally used for an I/Q waveform.



The following timing diagram shows an inphase waveform, quadrature waveform, and the convolved results with no phase error, data lags, and data leads.



MATLAB® Models of VHDL Components

The example includes three MATLAB® functions that test the VHDL model. A MATLAB function maps to each of the three VHDL components:

I/Q convolver: Verifies that the VHDL I/Q convolver code computes expected output for a randomly generated stream of samples. The MATLAB function verifies this by computing the convolution for the

inphase and quadrature waveforms (I_wf and Q_wf). The computation is implemented as an XOR and accumulation of the binary signals.

Decoder: Displays a plot of the I/Q mapping generated by the decoder for visual verification.

State counter: Generates the inphase and quadrature waveforms. During test benching, this MATLAB function has complete control of signals applied during the simulation, including clock generation, resets, and so on.

Running the Example

Starting the MATLAB Server

The example starts the MATLAB server, `hdldaemon`, such that it uses TCP socket communication with a socket port number identified as available by the operating system.

```
hdldaemon('socket',0)      % Activate MATLAB server to accept foreign VHDL calls
```

The example then calls `hdldaemon` with the 'status' option to get the assigned port number and store it in `portnum` for future reference.

```
dstat = hdldaemon('status');
portnum = dstat.ipc_id;
```

Both the server and client parts of an application link must use the same port number. Thus, at some point, the example program needs to forward the `portnum` over to ModelSim.

Testing the Decoder

The first component of the Manchester Receiver model to be tested is the decoder. The script creates 2 plots of the transfer function of this entity. This test is simply a visualization of the decoder behavior. The example script:

1. Sets a `testisdone` flag to 0 and displays informational messages.

```
global testisdone;
testisdone = 0;
```

2. Sets the project directory to a directory that has write access and is suitable for holding a ModelSim project.

```
projectdir = pwd;
```

3. Changes the format of the project directory and decoder VHDL file specifications to the UNIX® format, which ModelSim and Tcl use, by replacing backslashes (\) with forward slashes (/).

```
unixprojectdir = ["" strep(projectdir, '\', '/') ""];
unixsrcfile = ["" ...
    strep(fullfile(matlabroot, 'examples', 'hdlverifier', 'data', 'decoder.vhd'), '\', '/') ...
    ""];
```

4. Defines a sequence of Tcl commands to be executed in the context of ModelSim.

```
tclcmd = { ['cd ' unixprojectdir ],...
    'vlib work',...
    ['vcom -performdefaultbinding ' unixsrcfile],...
    'vsimmatlab work.decoder',...
    ['matlabtb decoder -mfunc manchester_decoder -socket ' num2str(portnum)],...
}
```

```
'run 3000',...
'quit -f'};
```

The previous list defines each command:

- Changes to the writable project directory.
- Adjusts the placement of the ModelSim window so it does not obscure the MATLAB window.
- Creates the project library work if it does not already exist.
- Compiles the VHDL file. The example script specifies the 'performdefaultbinding' option to enable default bindings in the event that they have been disabled in the modelsim.ini file.
- Loads an instance of the VHDL entity decoder for MATLAB test benching with the vsimmatlab command. This command is an HDL Verifier extension to the ModelSim command set.
- Initiates a MATLAB test benching session for the loaded instance of entity decoder with the matlabt command. This command is an extension to the ModelSim command set. The command in the example specifies the entity instance, the MATLAB function that is to test the entity (manchester_decoder.m), and TCP socket communication with socket port portnum. For a link to be established between ModelSim and MATLAB, the value of portnum must match the socket port that was specified when the MATLAB server (hdldaemon) was started.
- Runs the ModelSim simulation for 3000 iterations of the current resolution limit. By default, the simulation runs for 3000 nanoseconds. Quits ModelSim without asking for confirmation.

5. Starts ModelSim for use with the HDL Verifier with a call to vsim.

```
vsim('startupfile','decoder.do','tclstart',tclcmd);
```

This command starts ModelSim with a Tcl command script that executes some general-purpose startup commands and then the user-defined commands specified with the 'tclstart' property. The 'startfile' property causes vsim to write the entire startup Tcl command script to decoder.do for future reference or use.

6. Displays informational messages and waits for (manchester_decoder.m) to run to completion.

```
disp('Waiting for testing of ''decoder.vhd'' to complete');
disp('(flag from manchester_decoder.m indicates completion)');
while testisdone == 0,
    pause(0.501);
end
disp('MATLAB test of decoder.vhd is complete!');
disp('Check the generated plot for results.');
```

Testing the I/Q Convolver

Similarly for the I/Q convolver. The example script:

1. Sets a testisdone flag to 0 and displays informational messages.

```
testisdone = 0;
```

2. Sets the project directory to a directory that has write access and is suitable for holding a ModelSim project.

```
projectdir = pwd;
```

3. Changes the format of the project directory and decoder VHDL file specifications to the UNIX format, which ModelSim and Tcl use, by replacing backslashes (\) with forward slashes (/).

```

unixprojectdir = ["" stprep(projectdir, '\', '/') ""];
unixsrcfile = ["" ...
    stprep(fullfile(matlabroot, 'examples', 'hdlverifier', 'data', 'iqconv.vhd'), '\', '/') ...
    ""];

```

4. Defines a sequence of Tcl commands to be executed in the context of ModelSim.

```

tclcmd = { ['cd ' unixprojectdir ],...
    'vlib work',...
    ['vcom -performdefaultbinding ' unixsrcfile],...
    'vsimmatlab work.iqconv',...
    'force /iqconv/clk 1 0, 0 5 ns -repeat 10 ns ',...
    'force /iqconv/enable 1',...
    'force /iqconv/reset 1',...
    'run 100',...
    ['matlabtb iqconv -rising /iqconv/clk -mfunc manchester_iqconv -socket ', num2str(portnum)],...
    'run 1000',...
    'quit -f'};

```

The following list defines each command:

- Changes to the writable project directory.
- Adjusts the placement of the ModelSim window so it does not obscure the MATLAB window.
- Creates the project library work if it does not already exist.
- Compiles the VHDL file. The example script specifies the `-performdefaultbinding` option to enable default bindings in the event that they have been disabled in the `modelsim.ini` file.
- Loads an instance of the VHDL entity `iqconv` for MATLAB test benching with the `vsimmatlab` command. This command is an HDL Verifier extension to the ModelSim command set.
- Applies the ModelSim `force` command to drive the entity's `clk`, `enable`, and `reset` signals, which get passed on to the test bench as `oport` data. The first `force` command specifies that `clk` is to be set to 1 at time equals 0, to 0 after 5 nanoseconds, and repeat the high-to-low cycle every 10 nanoseconds. The second and third `force` commands set the `enable` and `reset` signals to 1.
- Runs the ModelSim simulation for 100 iterations of the current limit. By default, the simulation runs for 100 nanoseconds. This accounts for the startup phase.
- Initiates a MATLAB test benching session for the loaded instance of entity `iqconv` with the `matlabtb` command. This command is an extension to the ModelSim command set. The command in the example specifies the entity instance `iqconv`, the event that triggers an invocation of the MATLAB function, the MATLAB function that is to test the entity (`manchester_iqconv.m`), and TCP socket communication with socket port `portnum`. The `-rising` option specifies that the MATLAB function be called when `clk` experiences a rising edge. For a link to be established between ModelSim and MATLAB, the value specified with `-socket` must match the socket port that was specified when the MATLAB server (`hdldaemon`) was started.
- Runs the ModelSim simulation for 1000 iterations of the current resolution limit. By default, the simulation runs for 1000 nanoseconds.
- Quits ModelSim without asking for confirmation.

5. Starts ModelSim for use with the HDL Verifier with a call to `vsim`.

```
vsim('startupfile', 'iqconv.do', 'tclstart', tclcmd);
```

This command starts ModelSim with a Tcl command script that executes some general-purpose startup commands and then the user-defined commands specified with the `'tclstart'` property. The

'startfile' property causes vsim to write the entire startup Tcl command script to iqconv.do for future reference or use.

6. Displays informational messages and waits for (manchester_iqconv.m) to run to completion.

```
disp('Waiting for testing of 'iqconv.vhd' to complete');
disp('(flag from manchester_iqconv.m indicates completion)');
while testisdone == 0,
    pause(0.501);
end
disp('Test of iqconv.vhd complete (If it failed, there would be an error message printed above)');
```

Testing the State Counter

Now we test the State Counter (statecnt.vhd). The script will create checks isum and qsum outputs for a randomly generated stream of data samples and sets the testisdone flag to 0 and displays informational messages.

```
testisdone = 0;
projectdir = pwd;
unixprojectdir = ['" strrep(projectdir, '\', '/') '"'];
unixsrcfile = ['" ...
    strrep(fullfile(matlabroot, 'examples', 'hdlverifier', 'data', 'statecnt.vhd'), '\', '/') ...
    '"'];

tclcmd = { ['cd ' unixprojectdir ],...
    'vlib work',...
    ['vcom -performdefaultbinding ' unixsrcfile],...
    'vsimmatlab -t lns work.statecnt ',...
    'force /statecnt/clk 1 0, 0 5 ns -repeat 10 ns ',...
    ['matlabtb statecnt -mfunc manchester_statecnt -socket ', num2str(portnum)],...
    'run 30000',...
    'quit -f'};
```

The following list defines each TCL command in the tclcmd:

- Changes to the writable project directory.
- Adjusts the placement of the ModelSim window so it does not obscure the MATLAB window.
- Creates the project library work if it does not already exist.
- Compiles the VHDL file. The example script specifies the `-performdefaultbinding` option to enable default bindings in the event that they have been disabled in the `modelsim.ini` file. Loads an instance of the VHDL entity `statecnt` for MATLAB test benching with the `vsimmatlab` command. This command is an HDL Verifier extension to the ModelSim command set. The `-t` option specifies a ModelSim simulator time resolution of 1 nanosecond (the default). Applies the ModelSim force command to drive the entity's `clk` signal, which gets passed on to the test bench as `oport data`. The force command specifies that `clk` is to be set to 1 at time equals 0, to 0 after 5 nanoseconds, and repeat the high-to-low cycle every 10 nanoseconds.
- Initiates a MATLAB test benching session for the loaded instance of entity `statecnt` with the `matlabtb` command. This command is an extension to the ModelSim command set. The command in the example specifies the entity instance `statecnt`, the MATLAB function that is to test the entity (`manchester_statecnt.m`), and TCP socket communication with socket port `portnum`. For a link to be established between ModelSim and MATLAB, the value specified with `-socket` must match the socket port that was specified when the MATLAB server (`hdldaemon`) was started.
- Runs the ModelSim simulation for 30000 iterations of the current resolution limit. By default, the simulation runs for 30000 nanoseconds.

- Quits ModelSim without asking for confirmation.

```
vsim('startupfile','statecnt.do','tclstart',tclcmd);  
  
disp('Waiting for testing of ''statecnt.vhd'' to complete');  
disp('(flag from manchester_statecnt.m indicates completion)');  
while testisdone == 0,  
    pause(0.501);  
end  
disp('MATLAB test of statecnt.vhd is complete!');  
disp('Check the generated plot for results.');
```

Kill hlldaemon

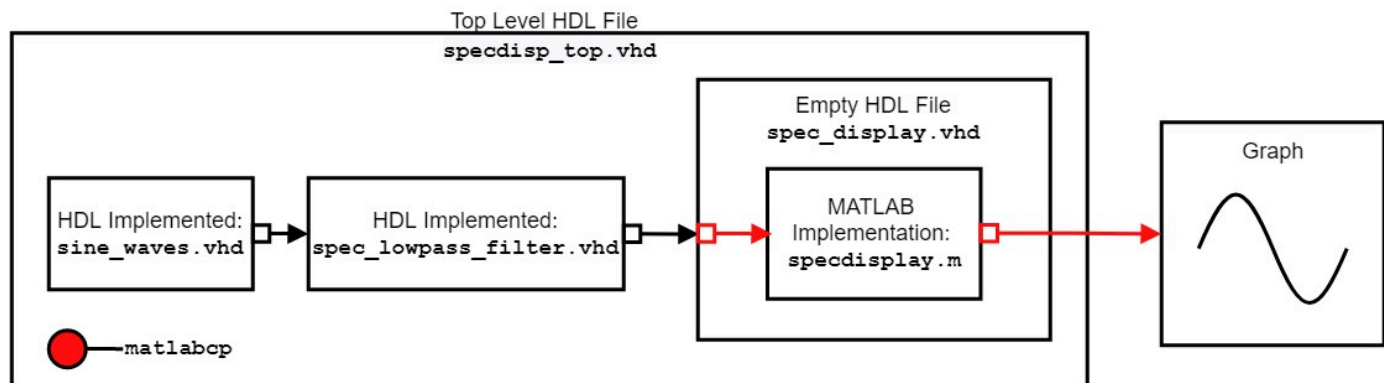
```
hdlldaemon('KILL');
```

Test of statecnt.vhd complete (Examine plot produced). This concludes the manchester tutorial example.

Implement Spectrum Display Component in MATLAB

This example shows how to run MATLAB® with Mentor Graphics® ModelSim®/Questasim® to visualize the output of a larger HDL project, by using the HDL Verifier™ function `matlabcp`. `matlabcp` needs a MATLAB function modeling the component behavior, along with an empty HDL component with input and output ports declared. These files are necessary as `matlabcp` uses the empty HDL component as a shell for communication between MATLAB and the HDL simulator, while the MATLAB function provides the functionality of the component. Using `matlabcp` allows MATLAB to process the simulator data as the simulation progress, saving time needed to transform the original outputs into a visual format. For information on creating the necessary files for `matlabcp`, see “Create a MATLAB Component Function” on page 3-2.

In this example, you use `matlabcp` to visualize the spectrum of a signal from a VHDL® project, which consists of a sine wave generator, a lowpass filter, and an empty HDL file representing the spectrum display component. `matlabcp` uses the empty HDL file to send the project output into a MATLAB function, which buffers the data into frames of 128 samples. Each time a frame is completed, the MATLAB function plots the spectrum of the unfiltered and filtered data in the frame. This use of `matlabcp` for reading the HDL data streamlines the transition between raw simulator data and visual plot, as MATLAB performs the data processing and graphic creation in parallel with the HDL simulator. This diagram shows an overview of the cosimulation system.



This example considers the point of view of an HDL developer in mind, so the main emphasis of this example is on using the HDL simulator and command-line terminal to perform testing and using MATLAB to manipulate and visualize the data from the HDL simulation.

Overview of Design and Script Files

This example uses two types of files: design files and script files.

Design Files

- The file `sine_waves.vhd` contains a frequency hopping sine wave implemented in VHDL.
- The file `spec_lowpass_filter.vhd` contains a Filter Design HDL Coder™ product generated lowpass filter.
- The file `spec_display.vhd` contains the empty component that `matlabcp` uses to feed the VHDL data into the MATLAB code.
- The file `specdisp_top.vhd` contains the top-level wiring between the signal source, lowpass filter, and empty component for `matlabcp`.

- The file `specdisplay.m` contains the MATLAB code that processes the inputs from the HDL simulator and plots their spectrum.

Script Files

- The file `qcommands_spec_w.tcl` contains the commands that are sent into ModelSim/Questasim on Windows® for cosimulation to occur.
- The file `qcommands_spec_l.tcl` contains the commands that are sent into ModelSim/Questasim on Linux® for cosimulation to occur.

Additionally, this example requires ModelSim/Questasim and MATLAB to be on the system path, as you must be able to run these products from the terminal.

Adjust the Script Files

This example uses the script files provided to link MATLAB and the HDL simulator through the use of two commands. These commands are a command that invokes the HDL simulator, and a command that uses a MATLAB shared library to create the connection for cosimulation. For this example to run, you need to change the command that invokes the HDL simulator, because the default command relies on an absolute path to the MATLAB shared library. Adjust the script files based on the operating system.

- Windows: This figure shows the two required commands for cosimulation with Modelsim/Questasim on Windows. The file `qcommands_spec_w.tcl` contains these two commands. Line 3 is the call to open ModelSim/Questasim, where the `-foreign` supplies the path to the MATLAB shared library, such that ModelSim/Questasim loads in the shared library. Line 4 is the call to a function in the shared library that connects the two programs together.

```
3 vsim -c work.specdisp_top -foreign {matlabclient
  {matlabroot\toolbox\edalink\extensions\modelsim\windows64\liblfmhdlc_gcc450vc12.dll} }
4 matlabcp u_spec_display -mfunc specdisplay
```

For this example to work, you must change line 3 in `qcommands_spec_w.tcl`. In this command, replace `matlabroot` with the installation location of MATLAB so that the shared library can be located and loaded.

- Linux: This figure shows the two required commands for cosimulation with Modelsim/Questasim on Linux. The file `qcommands_spec_l.tcl` contains these two commands. Line 3 is the call to open ModelSim/Questasim, where the `-foreign` supplies the path to the MATLAB shared library, such that ModelSim/Questasim loads in the shared library. Line 4 is the call to a function in the shared library that connects the two programs together.

```
3 vsim -c work.specdisp_top -foreign {matlabclient {matlabroot/toolbox/edalink/
  extensions/modelsim/linux64/liblfmhdlc_gcc450.so} }
4 matlabcp u_spec_display -mfunc specdisplay
```

For this example to work, you must change line 3 in `qcommands_spec_l.tcl`. In this command, replace `matlabroot` with the installation location of MATLAB so that the shared library can be located and loaded.

Run the Cosimulation

To start cosimulation, open MATLAB and run the HDL Link MATLAB server for communication between the HDL simulator and MATLAB. To complete these actions, use these commands based on the operating system.

- Windows: `matlab -nodesktop -r "hdldaemon"`
- Linux: `xterm -e "matlab -nodesktop -r "hdldaemon"" &`

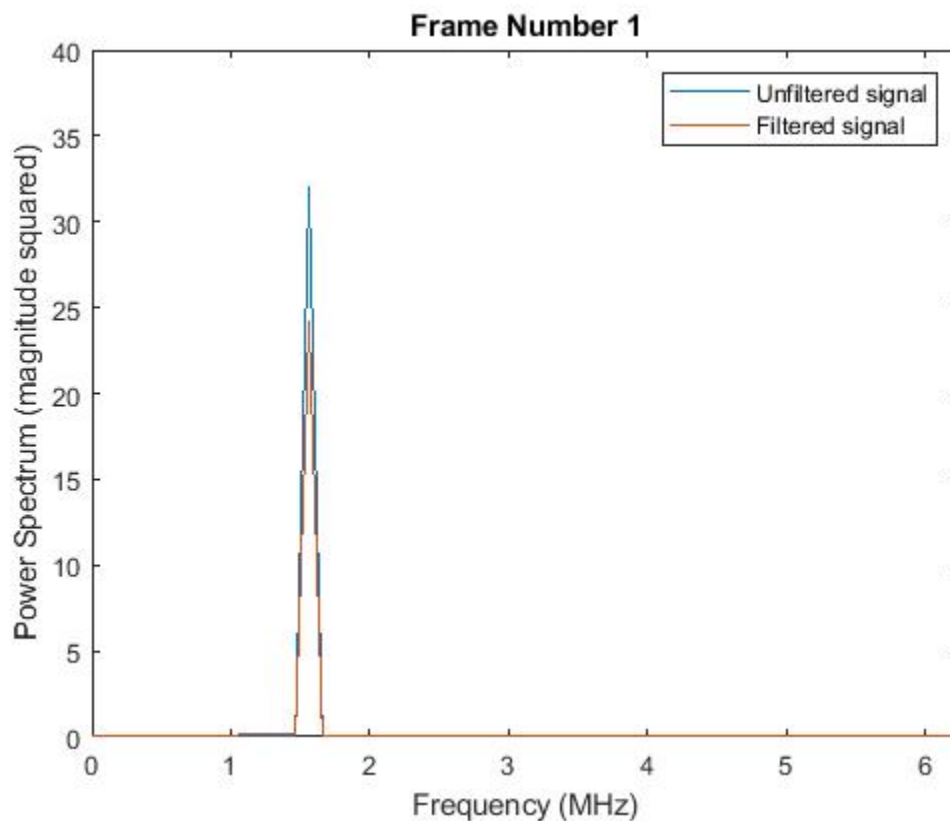
This command opens MATLAB in a separate terminal, in headless mode, while the HDL Link MATLAB server starts. The terminal that is running MATLAB displays this message:

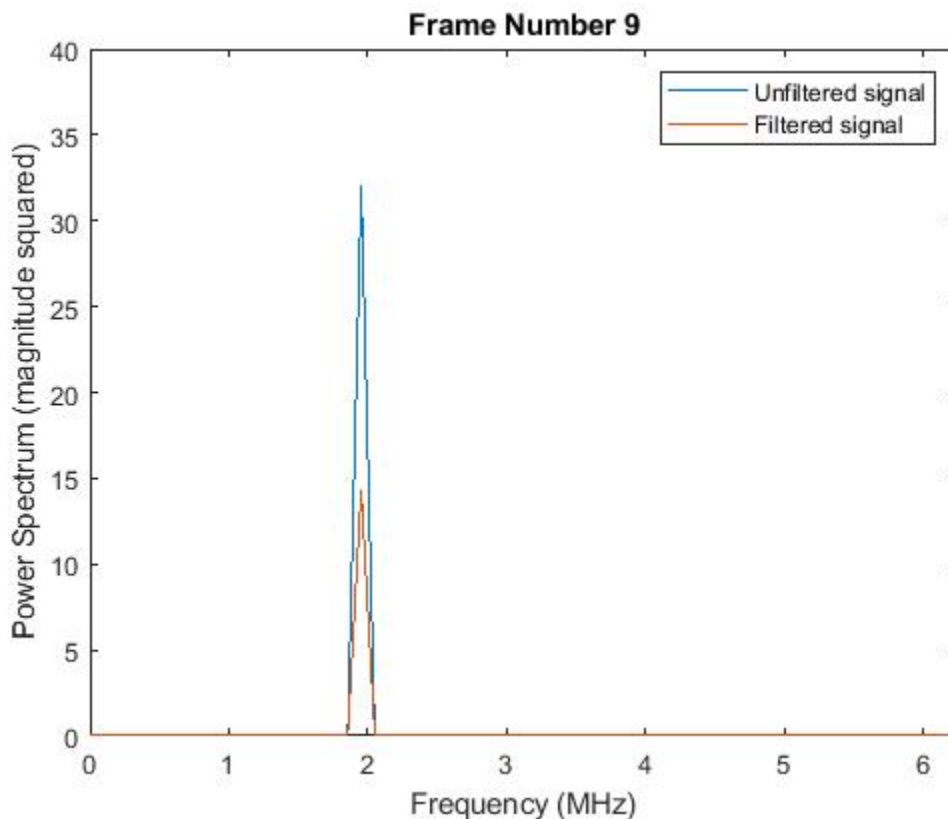
```
"HDLDaemon shared memory server is running with 0 connections"
```

After this message appears, enter the applicable command based on operating system into a system terminal.

- Windows: `vsim -c -do qcommands_spec_w.tcl`
- Linux: `vsim -c -do qcommands_spec_l.tcl`

This command starts the cosimulation. Over the course of the simulation, MATLAB captures, processes, and displays multiple frames. These plots show the first and last frames of the simulation.





Summary

This example shows how to use MATLAB to interpret and display results after the HDL simulation has finished processing. Even though the simulator can display the unmodified results of the project, applying transformations to the results for visualization can be difficult. In this example, the spectrum of the frequency hopping sine wave was desired, but to obtain this spectrum, a Fourier transform is required. This requirement significantly complicates the process of coding, simulating, and verifying the results.

Using MATLAB and `matlabcp` can significantly speed up this visualization, as MATLAB can read data from the simulation and utilize built-in functionality to apply the Fourier transform and display the results. By using MATLAB in the verification process, the complexity of visualizing the data is reduced, streamlining the process of converting simulation results into visualized data.

See Also

`matlabcp` | `hdldaemon` | `vsim` | `nclaunch`

Related Topics

- “Set Up for HDL Cosimulation” on page 10-2
- “Create a MATLAB Component Function” on page 3-2
- “Set Up Cosimulation Component” on page 3-8
- “Run MATLAB-HDL Cosimulation” on page 1-4

Copyright 2005-2021 The MathWorks, Inc.

Manchester Receiver (Absolute Time Mode)

This example shows verification of a Manchester encoder using HDL Verifier with Simulink. Manchester encoding is a simple modulation scheme that converts baseband digital data to an encoded waveform with no DC component. The most widely known application of this technique is Ethernet.

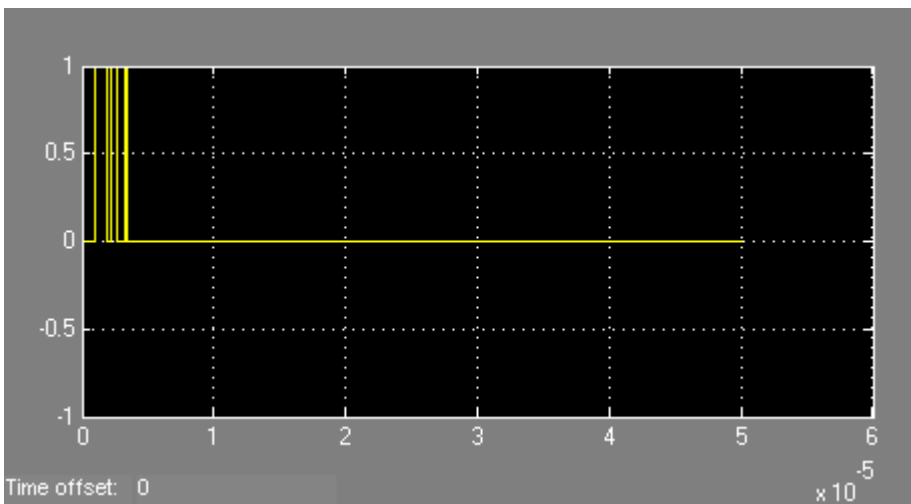
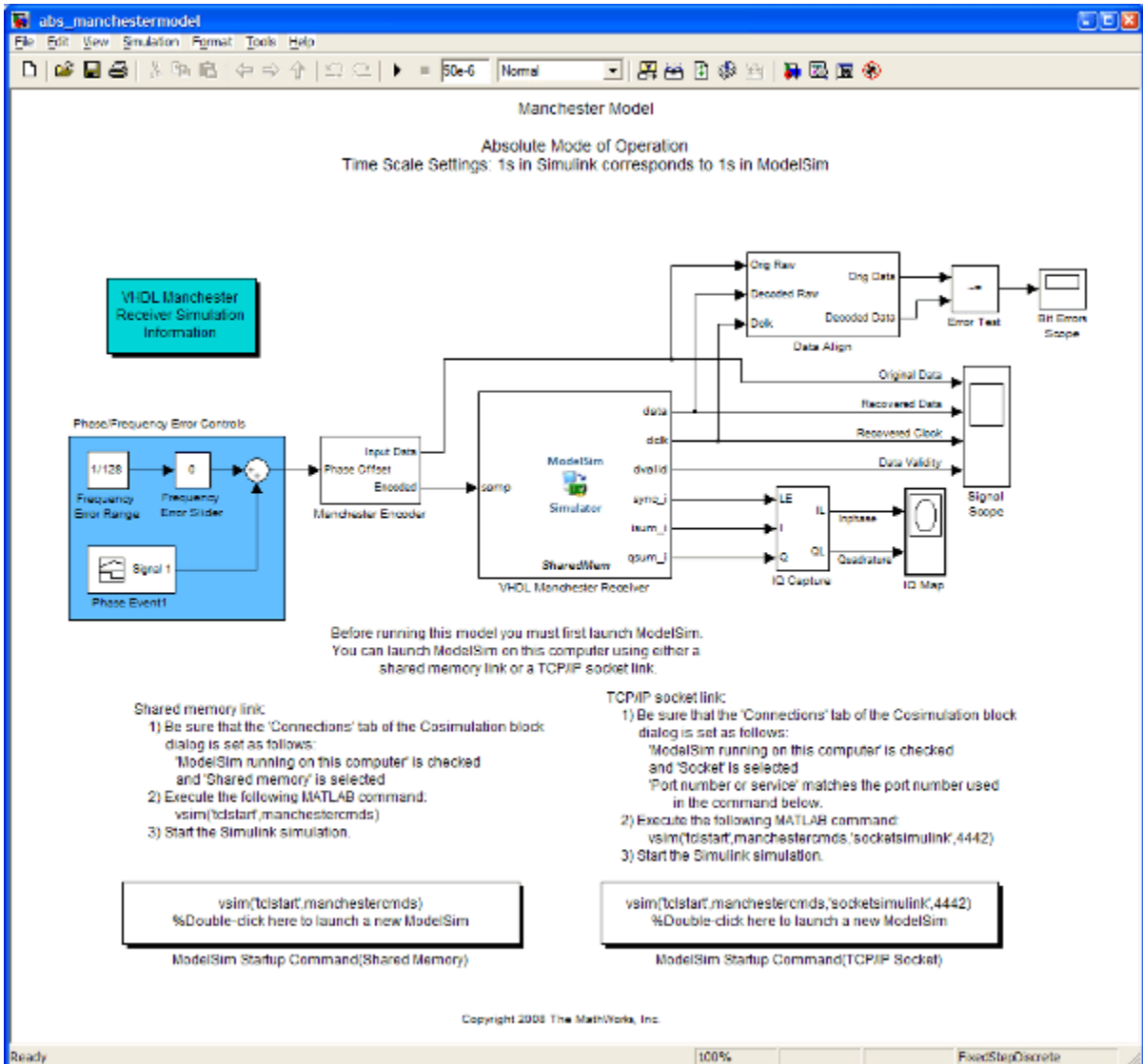
This model simulates a digital receiver of Manchester encoded data. The receiver is implemented in VHDL. The receiver uses a simple DLL (delay lock loop) clock recovery mechanism, which requires multiple cycles to lock with the incoming data stream. The performance of the receiver is explored by applying phase and frequency errors to a randomly generated stream of bits that is encoded using a simple MATLAB® function: `manchesterencoder()`.

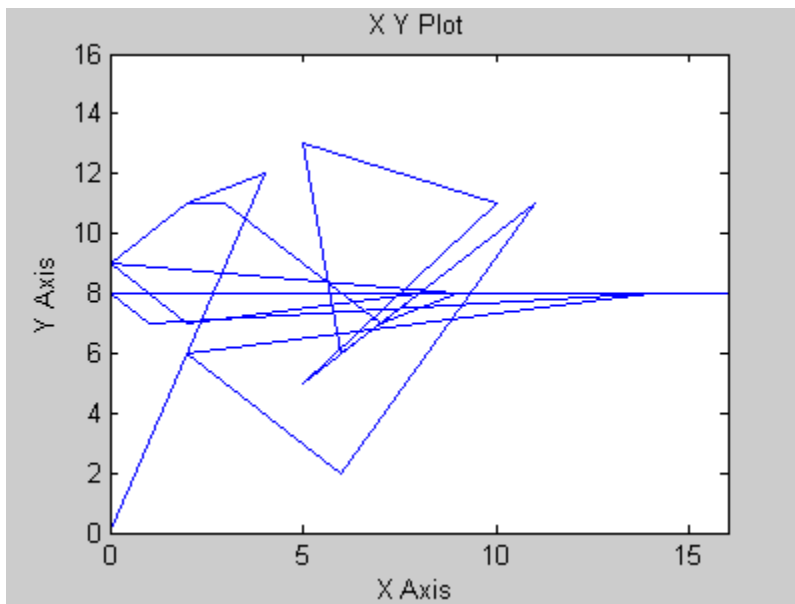
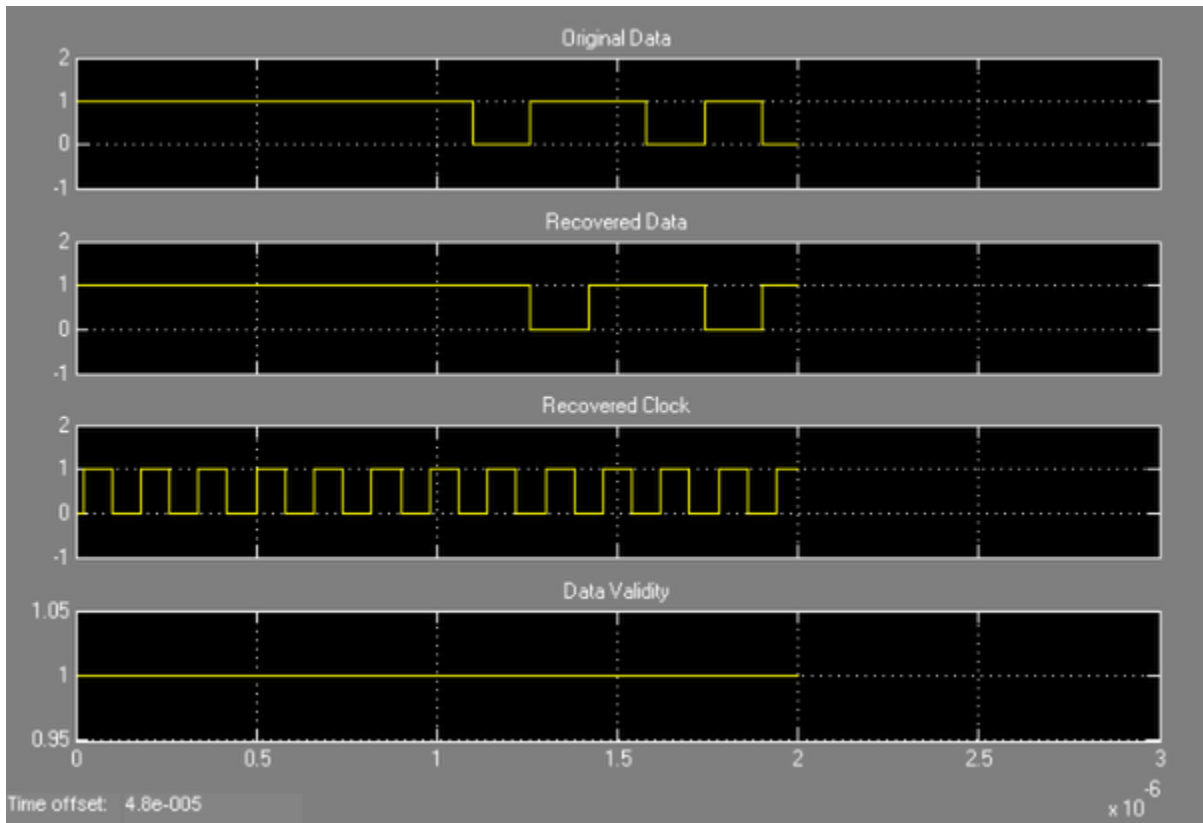
The VHDL code runs in ModelSim® as a ModelSim VHDL Cosimulation block labeled VHDL Manchester Receiver.

```
vsim('tclstart',manchestercmds);
pause(5);

open_system('abs_manchestermodel')

cmd = get_param('abs_manchestermodel/VHDL Manchester Receiver','TclPostSimCommand');
set_param('abs_manchestermodel/VHDL Manchester Receiver','TclPostSimCommand',[cmd '; quit -f;'])
sim('abs_manchestermodel')
```





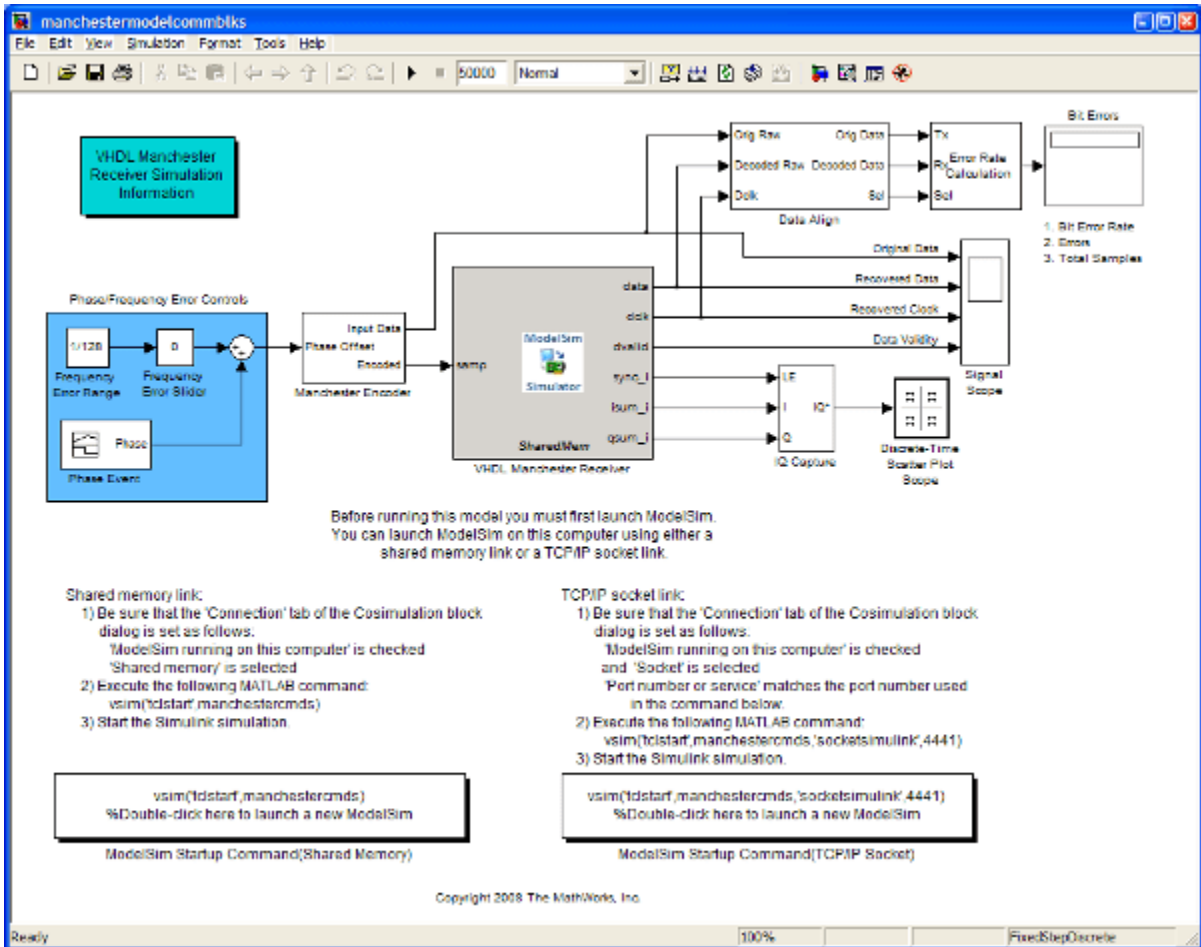
Manchester Receiver Using Communications Toolbox

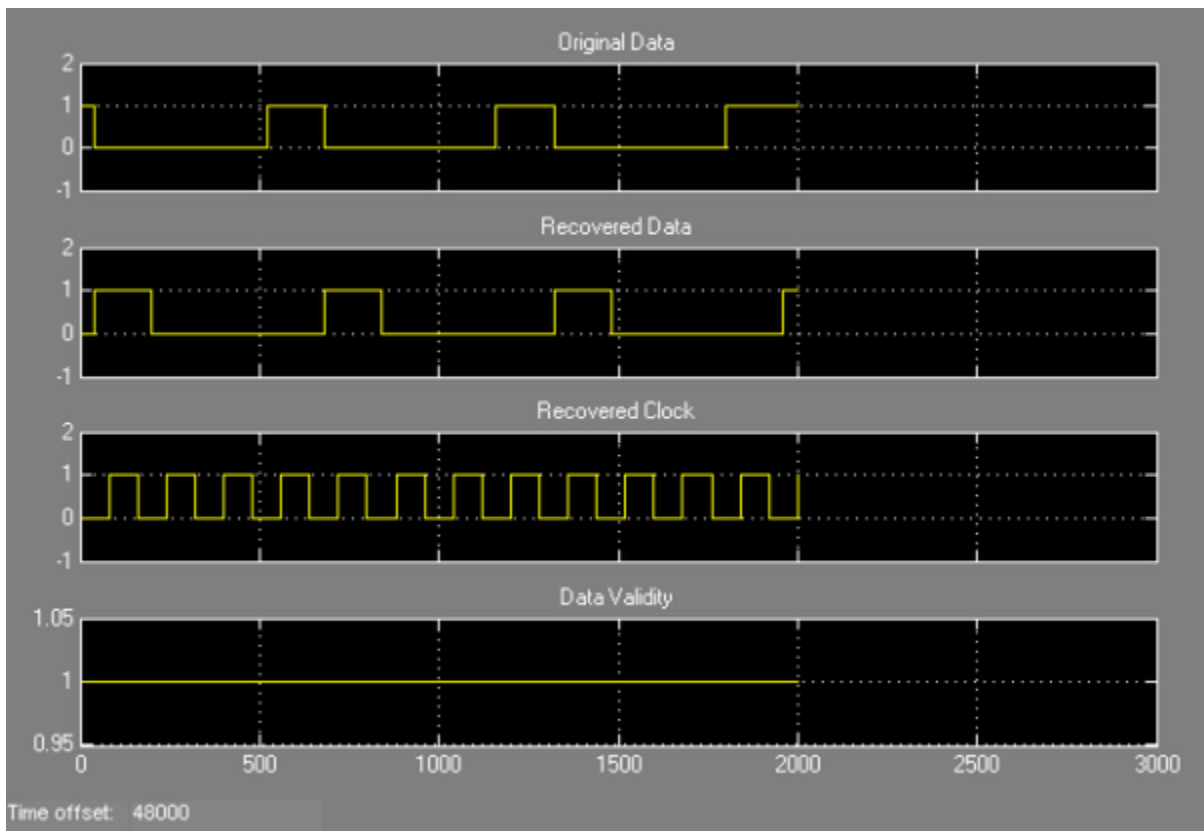
This example shows verification of a Manchester encoder. Manchester encoding is a simple modulation scheme that converts baseband digital data to an encoded waveform with no DC component. The most widely known application of this technique is Ethernet.

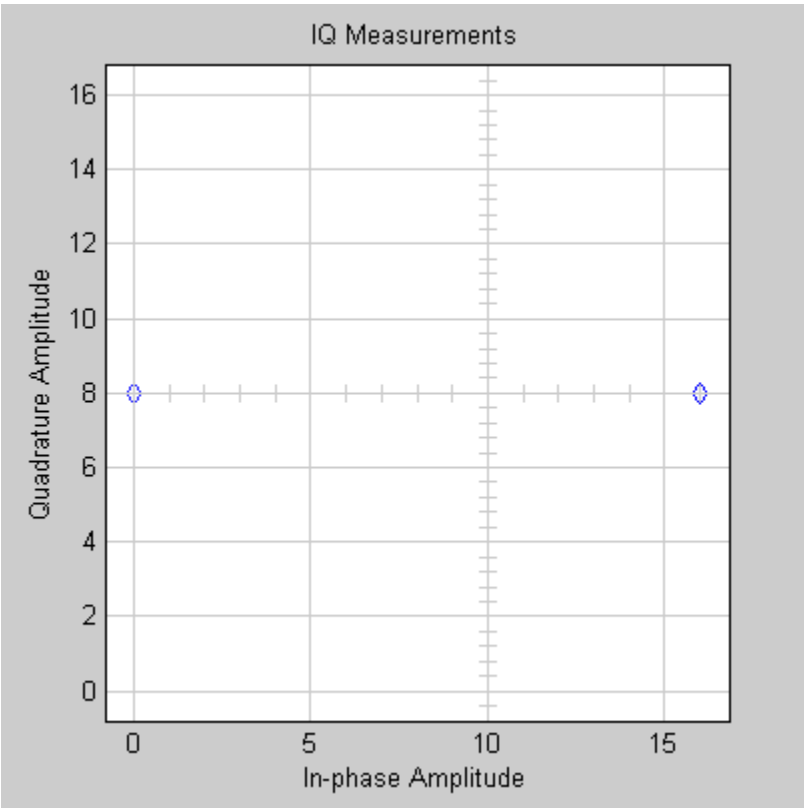
This model simulates a digital receiver of Manchester encoded data. The receiver is implemented in VHDL. The receiver uses a simple DLL (delay lock loop) clock recovery mechanism, which requires multiple cycles to lock with the incoming data stream. The performance of the receiver is explored by applying phase and frequency errors to a randomly generated stream of bits that is encoded using a simple MATLAB® function: `manchesterencoder()`.

The VHDL code runs in ModelSim® as a ModelSim® VHDL Cosimulation block labeled VHDL Manchester Receiver.

```
vsim('tclstart',manchestercmds);  
pause(5);  
open_system('manchestermodelcommblocks')  
cmd = get_param('manchestermodelcommblocks/VHDL Manchester Receiver','TclPostSimCommand');  
set_param('manchestermodelcommblocks/VHDL Manchester Receiver','TclPostSimCommand',[cmd ' ; quit -f  
sim('manchestermodelcommblocks')
```







Cosimulation Wizard for MATLAB System Object

Set up an HDL Verifier™ application using the Cosimulation Wizard.

This example uses a MATLAB® System object and following HDL simulators to verify a register transfer level (RTL) design.

- Vivado® Simulator from Xilinx®
- ModelSim® or Questa® from Mentor Graphics®
- Xcelium® from Cadence®

The example design is a Fast Fourier Transform (FFT) of size 8 written in Verilog. The FFT is commonly used in digital signal processing applications to produce frequency distribution of a signal.

To verify the correctness of this FFT, a MATLAB System object testbench is provided. This testbench generates a periodic sinusoidal input to the HDL design under test (DUT) and plots the Fourier Coefficients in the Complex Plane.

The Cosimulation Wizard takes the provided Verilog file of this FFT as its input. It also collects user input required for setting up cosimulation in each step. At the end of the example, the Cosimulation Wizard generates a MATLAB script that instantiates a configured HdlCosimulation System object, a MATLAB script that compiles HDL design and a MATLAB script that launches the HDL simulator for cosimulation.

1. Launch Cosimulation Wizard

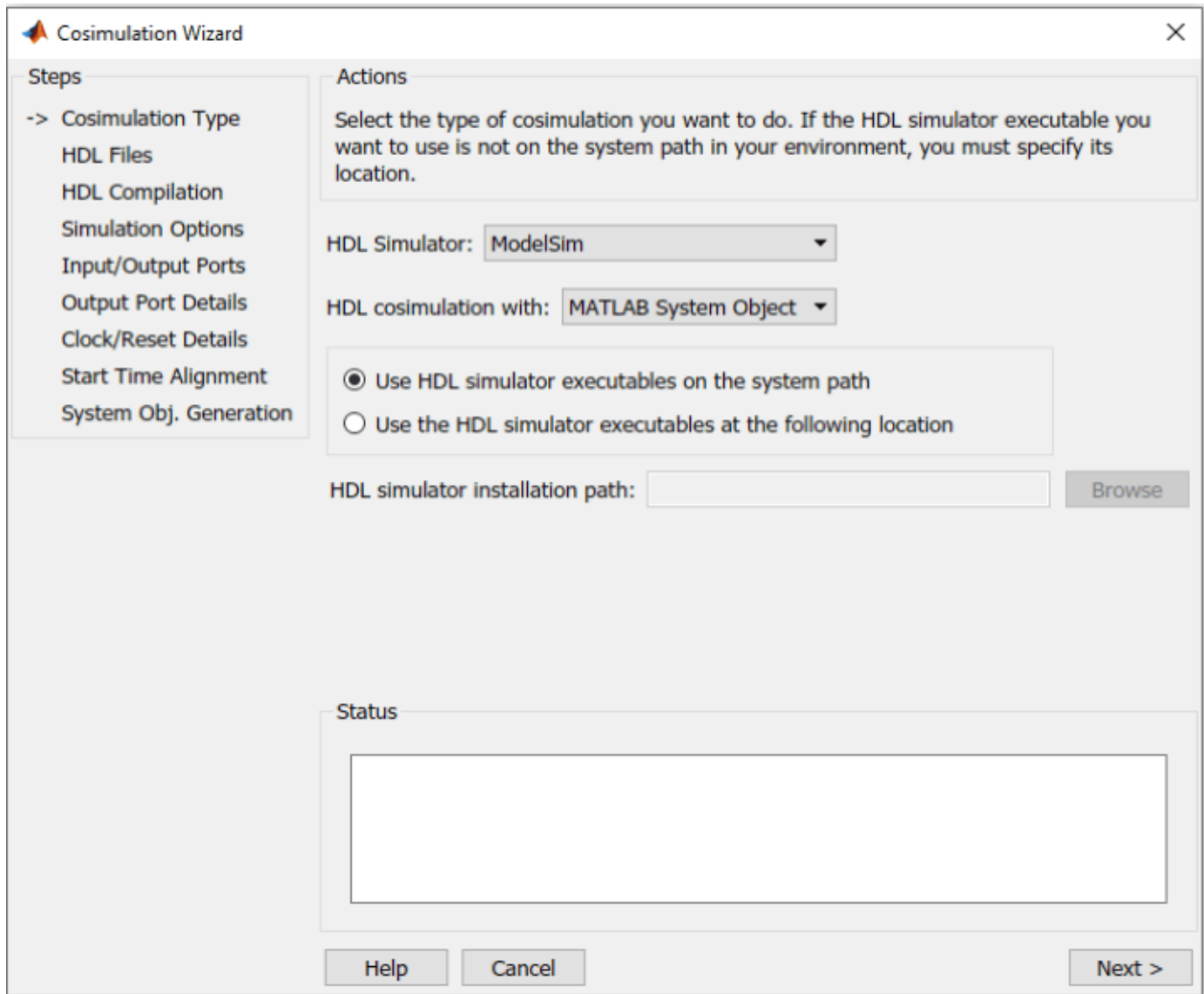
Launch the Cosimulation Wizard tool by executing this command in MATLAB.

```
cosimWizard
```

2. Specify Cosimulation Type

On the Cosimulation Type page, perform the following steps:

- a. If you are using ModelSim, set **HDL Simulator** to ModelSim.



If you are using Xcelium, set **HDL Simulator** to Xcelium.

If you are using Vivado Simulator, set **HDL Simulator** to Vivado Simulator.

b. Set **HDL cosimulation** to MATLAB System Object.

c. Do not change the default **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path. If these executable do not appear on the path, specify the HDL simulator path.

d. Click **Next**.

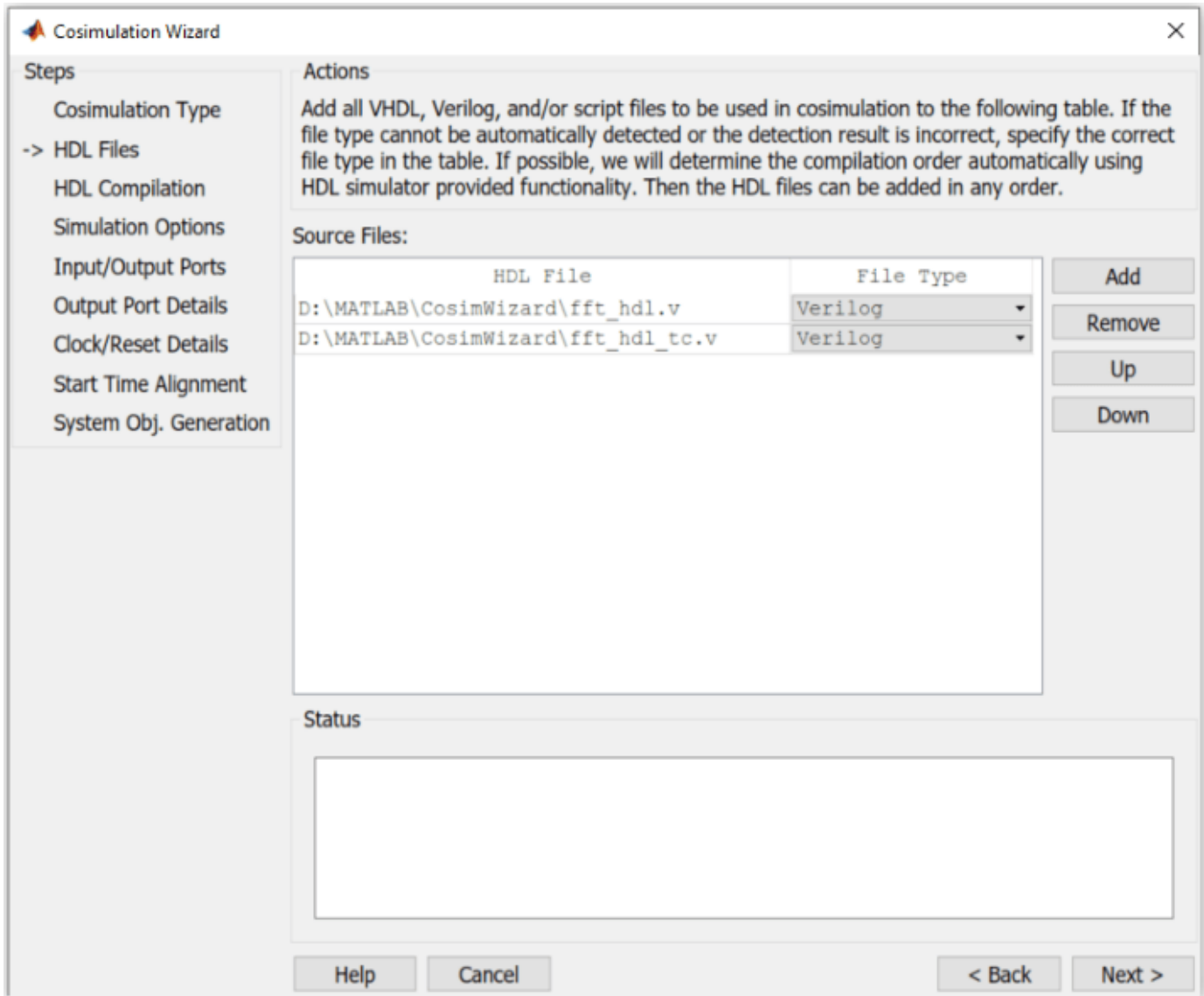
3. Select HDL Files

On the HDL Files page, perform the following steps:

a. Add HDL files to the file list:

- Click **Add** and select the Verilog files **fft_hdl.v** and **fft_hdl_tc.v** in your example folder.
- Review the files in the file list to make sure the file type is correctly identified.

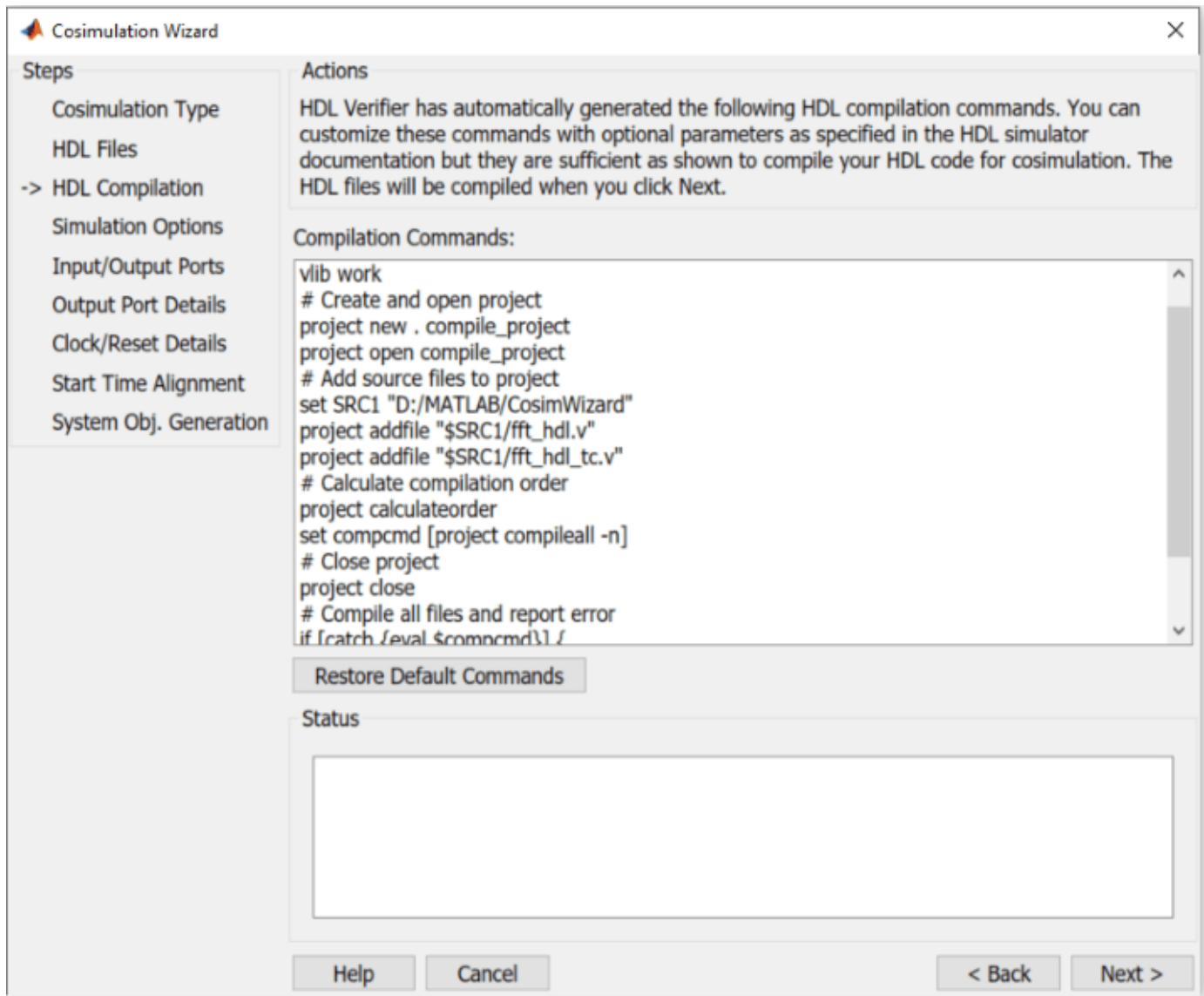
b. Click **Next**.



4. Specify HDL Compilation Commands

The Cosimulation Wizard lists the default commands in the Compilation Commands window. For this example, you do not need to change these commands.

Compilation commands for the ModelSim follow.



Click **Next**. The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area. Correct the error before proceeding to the next step.

5. Select HDL Modules for Cosimulation

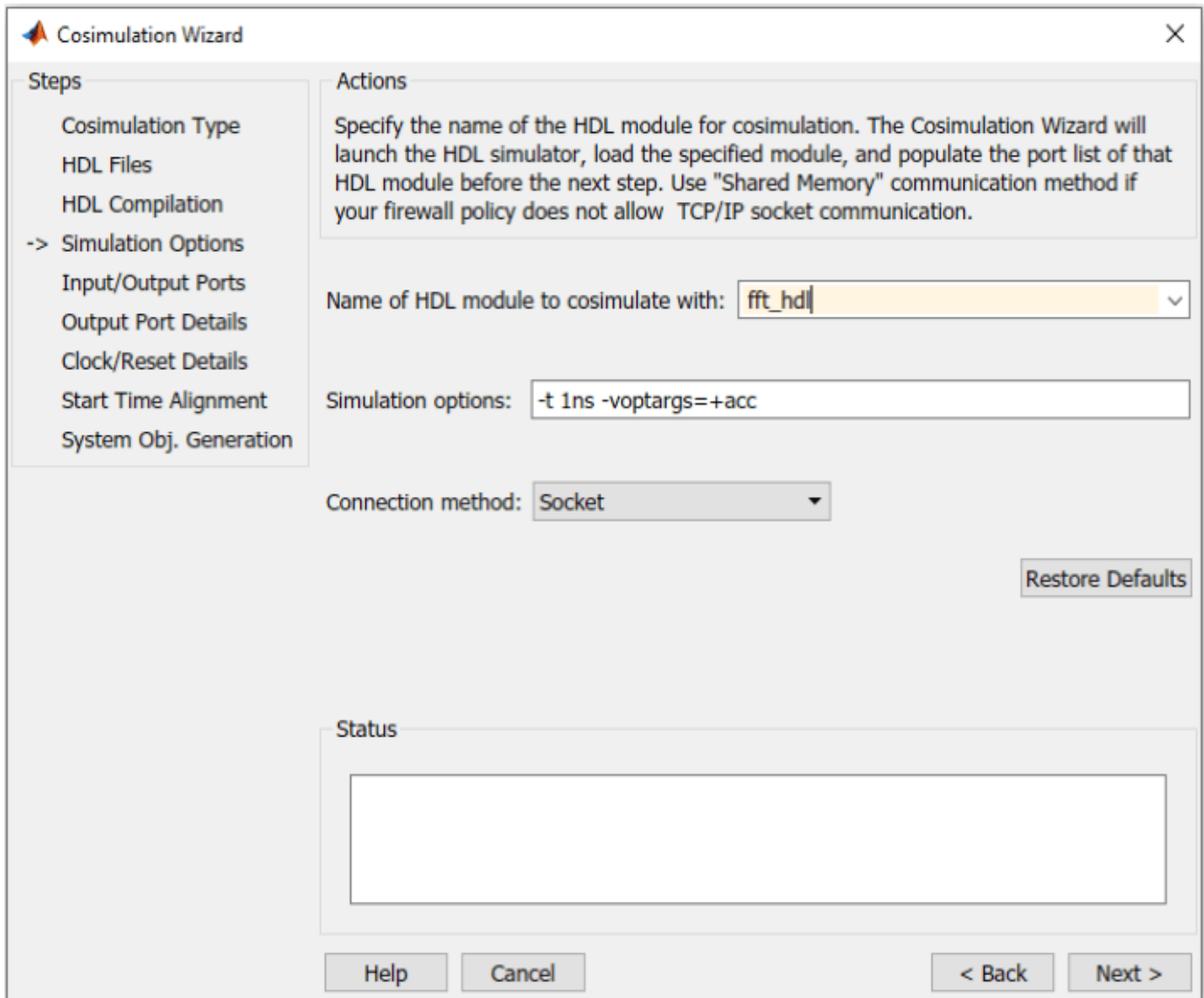
On the Simulation Options page, perform the following steps:

- a. Specify the name of the HDL module or entity for cosimulation.

For ModelSim or Xcelium

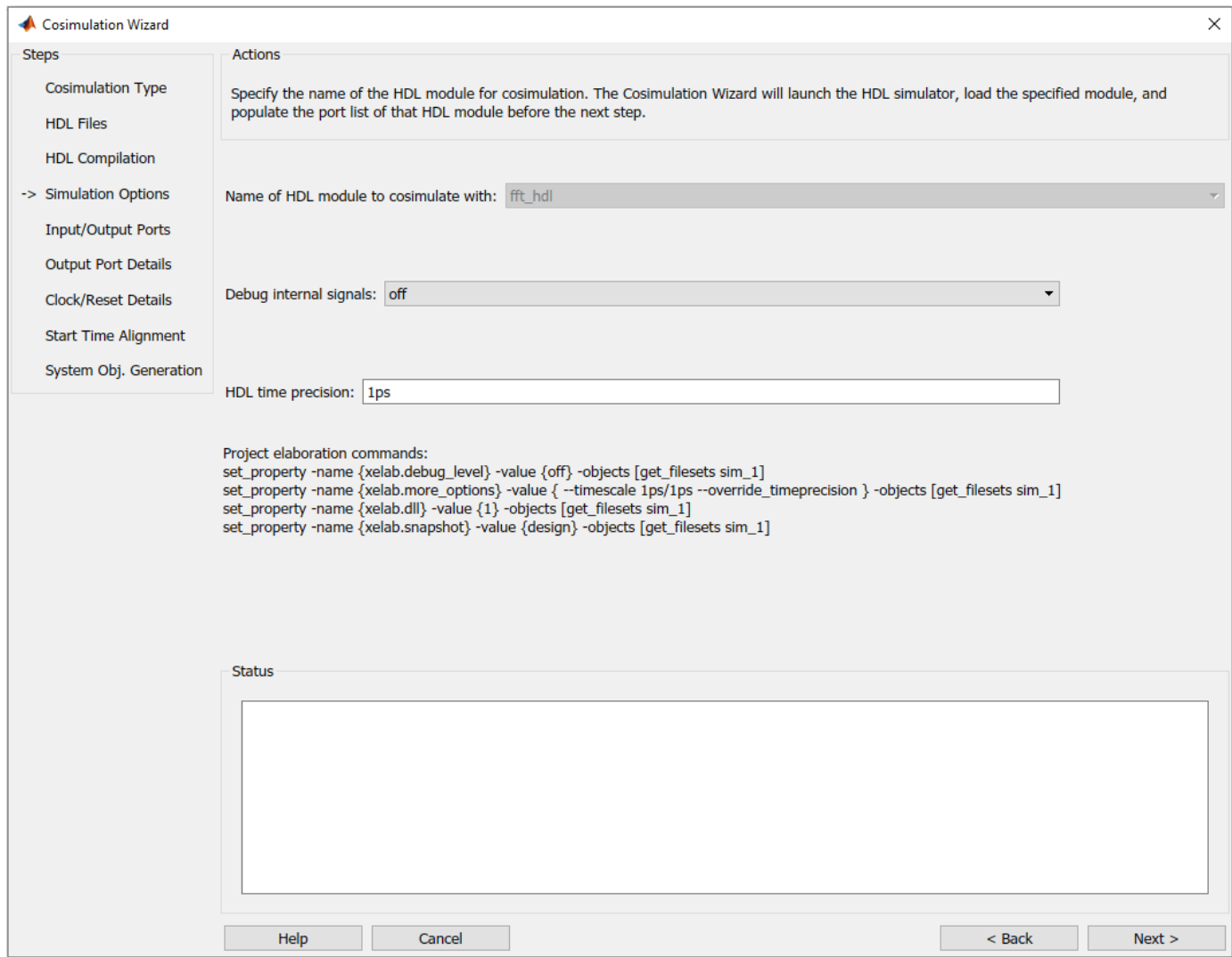
From the list, select `fft_hdl`. This module is the Verilog module you use for cosimulation. If you do not see `fft_hdl` in the list, enter the file name manually.

The Simulation options for the ModelSim follow.



For Vivado Simulator

For the Vivado simulator, name of Verilog module is selected by default. The Simulation options for Vivado simulator follow. .



b. Click **Next**. The Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. When the wizard launches the HDL simulator successfully, the wizard populates the input and output ports on the Verilog model **fft_hdl** and displays them in the next step.

6. Specify Input/Output Port Types

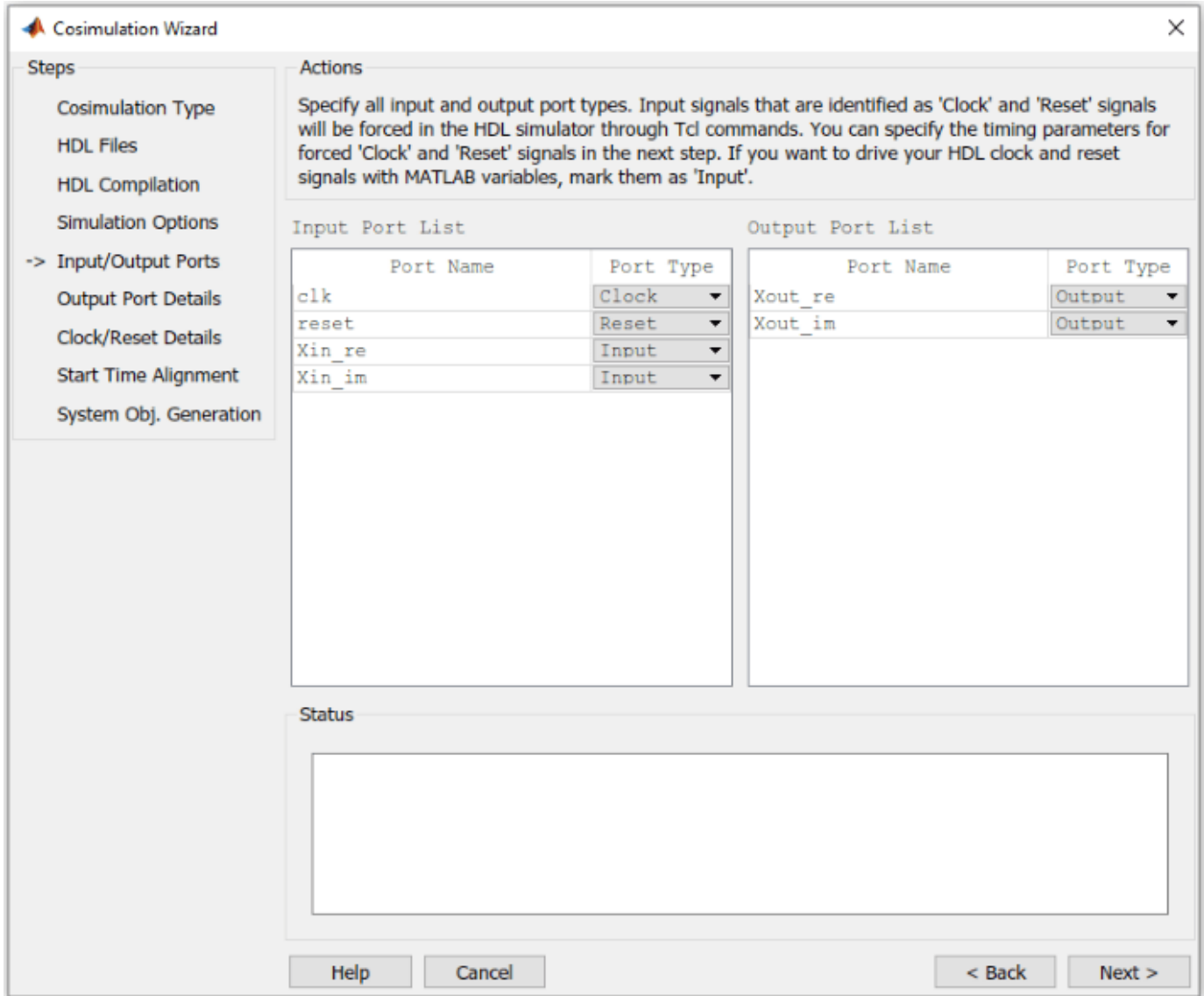
In this step, the Cosimulation Wizard displays two tables containing the input and output ports of **fft_hdl**, respectively.

The Cosimulation Wizard attempts to correctly identify the port type for each port. If the wizard incorrectly identifies a port, you can change the port type using these tables.

- For input ports, you can select **Clock**, **Reset**, **Input**, or **Unused**. HDL Verifier connects only the input ports marked **Input** to MATLAB during cosimulation.
- HDL Verifier connects output ports marked **Output** with MATLAB during cosimulation. The link software and MATLAB ignore those output ports marked **Unused** during cosimulation.

- You can change the parameters for signals identified as Clock and Reset in a later step.

For this example, accept the default port types and click **Next**.

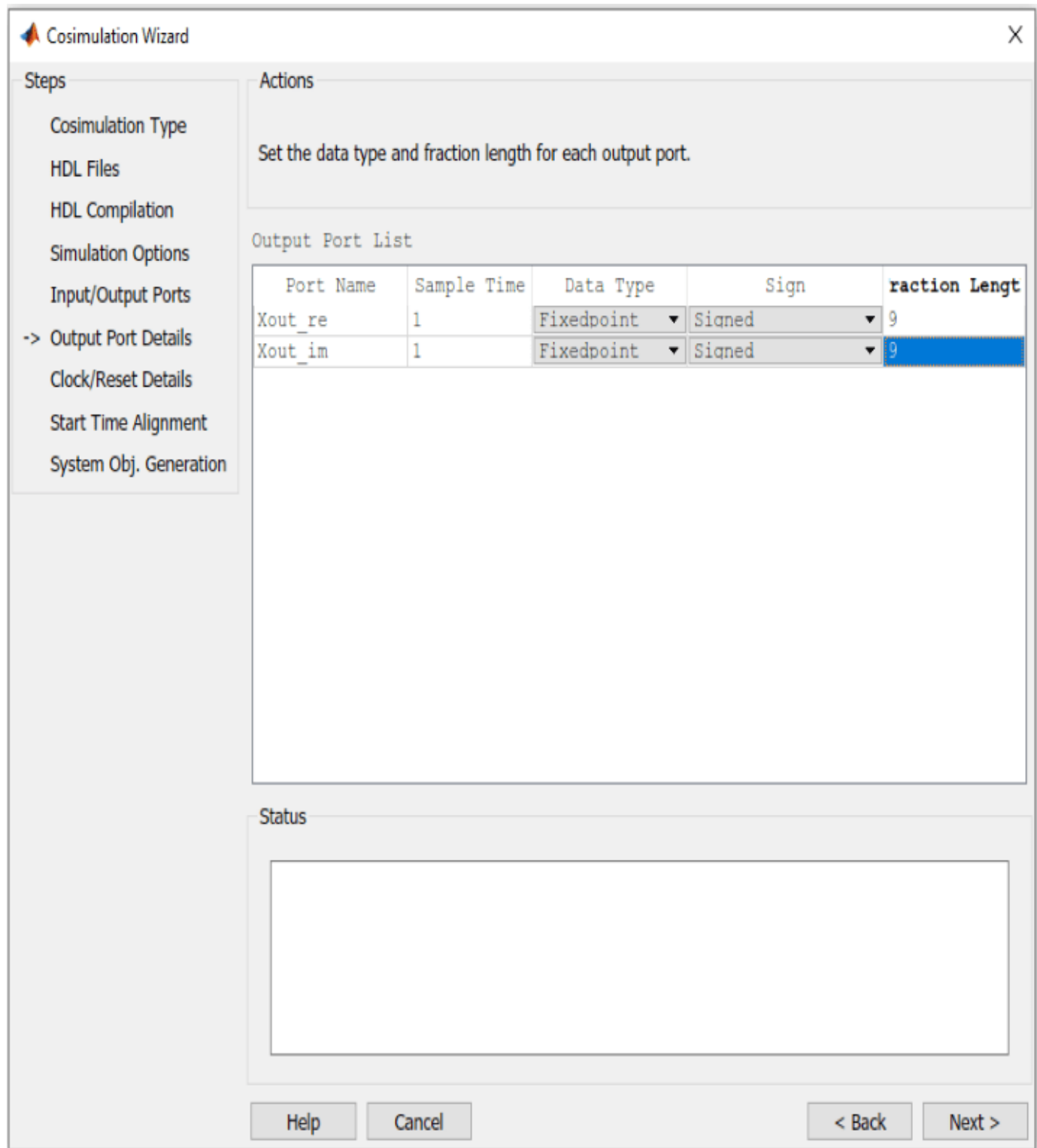


7. Specify Output Port Details

For this example, the HDL FFT outputs are signed, 13 bits long with 9 bits of fraction length. On the Output Port Details page, perform the following steps:

- Note that the **Sample Time** cannot be changed and is always fixed to 1 when you use the HdlCosimulation System object.
- Set the **Data Type** to Fixedpoint for both outputs.
- Set the **Sign** to Signed for both inputs.
- Set the **Fraction Length** to 9 for both outputs.

e. Click **Next**.



Cosimulation Wizard

Steps

- Cosimulation Type
- HDL Files
- HDL Compilation
- Simulation Options
- Input/Output Ports
- > Output Port Details
- Clock/Reset Details
- Start Time Alignment
- System Obj. Generation

Actions

Set the data type and fraction length for each output port.

Output Port List

Port Name	Sample Time	Data Type	Sign	Fraction Length
Xout_re	1	Fixedpoint	Signed	9
Xout_im	1	Fixedpoint	Signed	9

Status

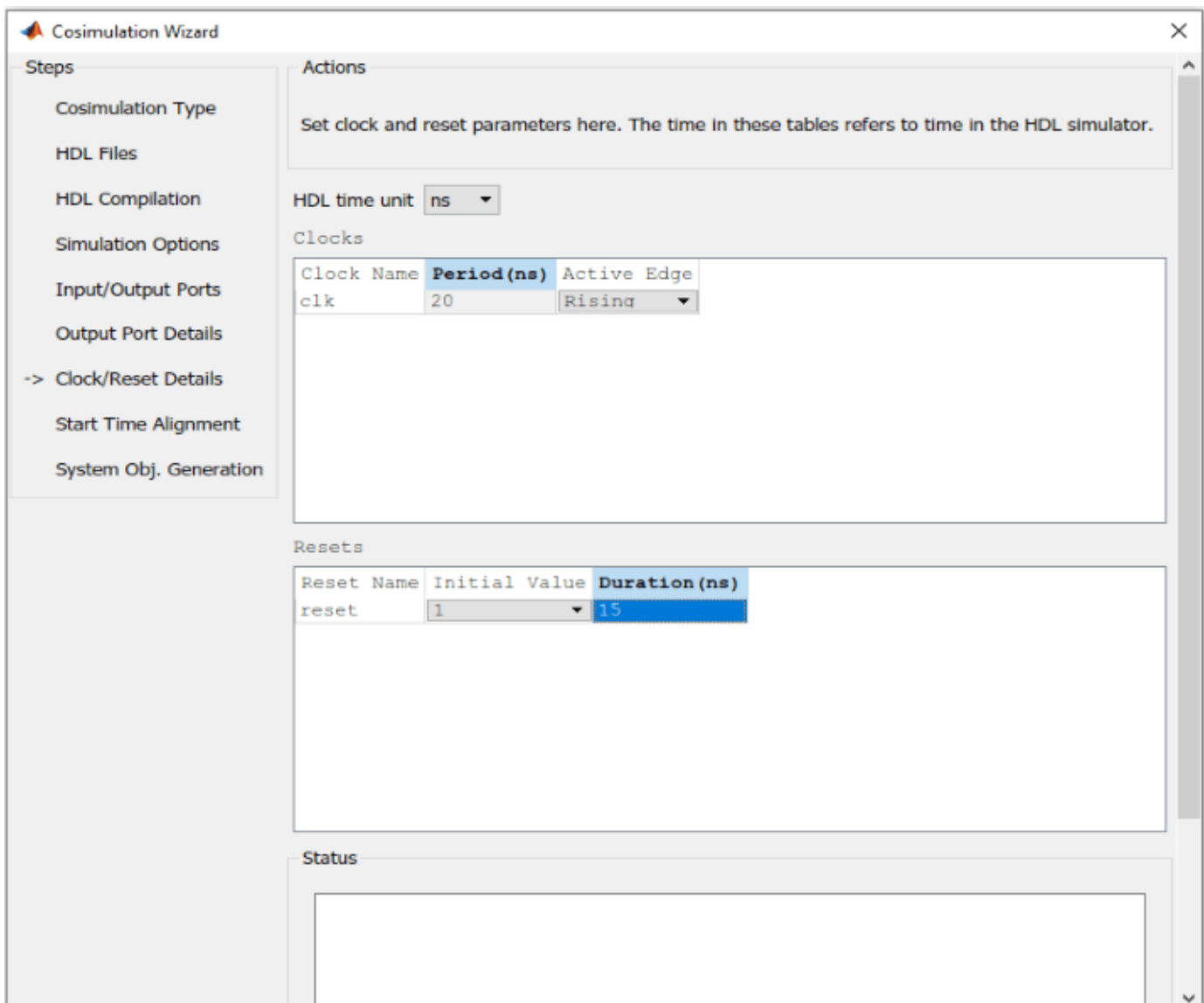
Help Cancel < Back Next >

8. Set Clock and Reset Details

Set the clock period (ns) to 20. The Verilog code indicates that the reset is synchronous and the active value is 1. You can reset the entire HDL design at time 1 ns, triggered by the rising edge of the clock. Use a duration of 15 ns for the reset signal. On the Clock/Reset Details page, perform the following steps:

- a. Set the clock period to 20 .
- b. Set the active edge to Rising .
- c. Set the reset initial value to 1 .
- d. Set the reset signal duration to 15 .

Click **Next**.



9. Confirm Start Time Alignment

The Start Time Alignment page displays a plot for the waveforms of clock and reset signals. The Cosimulation Wizard indicates the HDL time to start cosimulation with a red line. The start time is also the time at which the System object gets the first input sample from the HDL simulator. The active edge of the clock is a rising edge. Thus, at time 20 ns in the HDL simulator, the registered output of the FFT is stable. No race condition exists and the default HDL time to start cosimulation (20 ns) is correct.

Click **Next**.

Cosimulation Wizard

Steps

- Cosimulation Type
- HDL Files
- HDL Compilation
- Simulation Options
- Input/Output Ports
- Output Port Details
- Clock/Reset Details
- > Start Time Alignment
- System Obj. Generation

Actions

The diagram below shows the current settings for forced 'Clock' and 'Reset' signals. The red line represents the time in the HDL simulation at which MATLAB/Simulink will start (i.e. cosimulation will start).

To change the MATLAB/Simulink start time relative to the HDL simulation time, enter the new start time below. To avoid a race condition, make sure the start time does not coincide with the active edge of any clock signal. You can do so by moving the start time or by changing the clock active edge in the previous step (click Back).

HDL time to start cosimulation (ns):

clk

reset

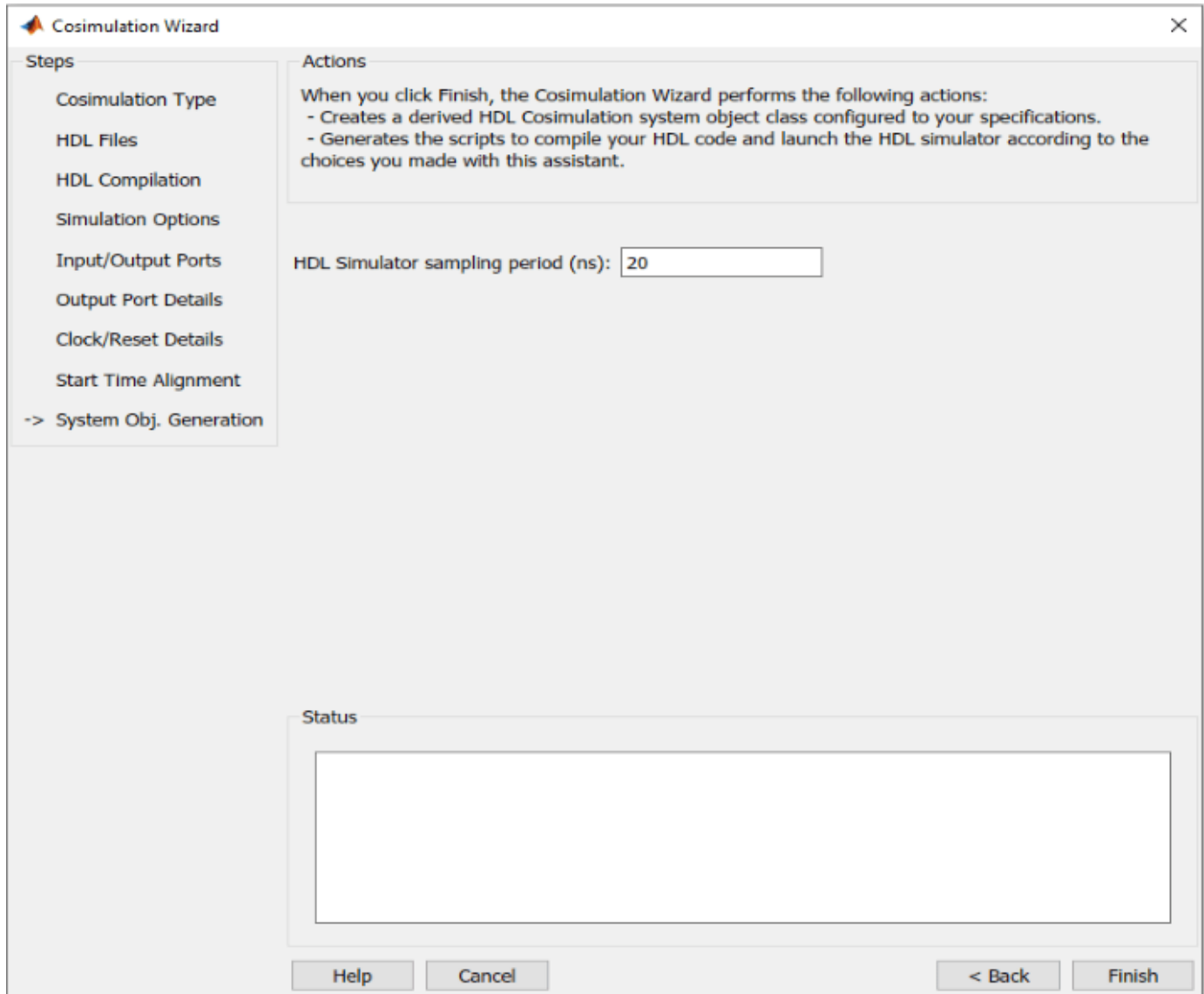
Status

10. Generate System Object

Before the Cosimulation Wizard generates the scripts, you have the option to modify the HDL Simulator sampling period. The sampling period determines the time in the HDL Simulator that

elapses between each call to step in MATLAB. The sampling period is typically equal to the clock period. You can also specify if your inputs and outputs are frame based (instead of sample based).

Click **Finish** to complete the Cosimulation Wizard session.



11. Create Test Bench to Verify HDL Design

For this example, you do not actually create the test bench. Instead, you can find the finished script **fft_tb.m** in the directory where your verilog files reside.

After you click **Finish** in the Cosimulation Wizard, the application generates three MATLAB scripts in the current directory.

For ModelSim and Xcelium

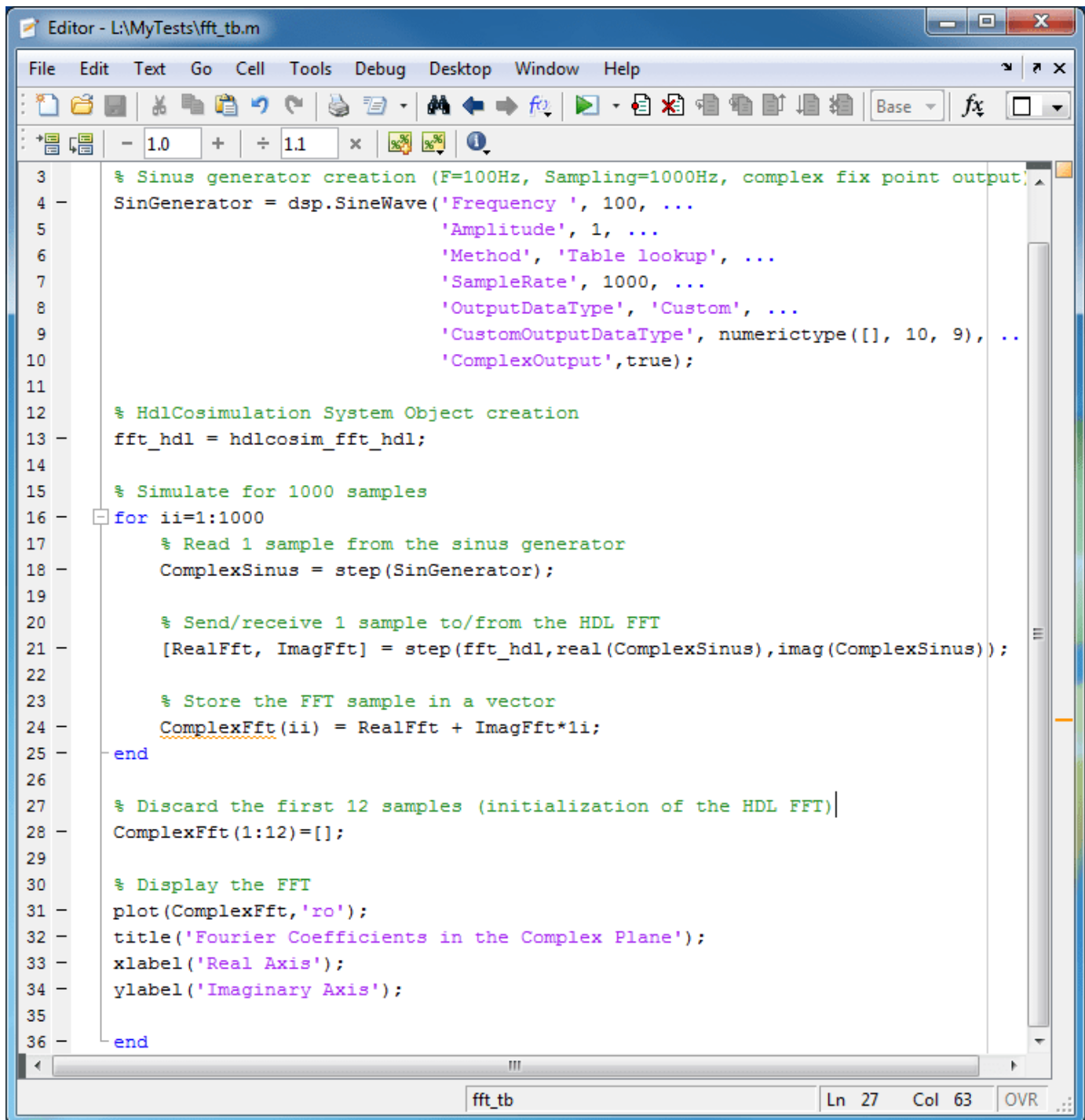
- **compile_hdl_design_fft_hdl.m**: To recompile the HDL design.

- **launch_hdl_simulator_fft_hdl.m**: Relaunches the MATLAB System object server and starts the HDL simulator.
- **hdlcosim_fft_hdl.m**: Creates the HdlCosimulation System object.

For Vivado Simulator

- **hdlverifier_compile.m**: Recompiles the HDL design.
- **hdlverifier_gendll_fft_hdl.m**: Creates a compiled shared library containing the HDL design and simulation kernel integrated into the behavior of the System object.
- **hdlcosim_fft_hdl.m**: Creates the HdlCosimulation System object.

Open the files **fft_tb.m** and **hdlcosim_fft_hdl.m**, located in the same directory as the Verilog files, and observe the HdlCosimulation System object calls. **hdlcosim_fft_hdl.m** contains the HdlCosimulation instantiation and **fft_tb.m** contains a MATLAB System object test bench. Use this test bench to verify the HDL design for the corresponding HdlCosimulation System object.



```

Editor - L:\MyTests\fft_tb.m
File Edit Text Go Cell Tools Debug Desktop Window Help
Base fx
- 1.0 + ÷ 1.1 × %%% %%%
3 % Sinus generator creation (F=100Hz, Sampling=1000Hz, complex fix point output)
4 SinGenerator = dsp.SineWave('Frequency ', 100, ...
5                               'Amplitude', 1, ...
6                               'Method', 'Table lookup', ...
7                               'SampleRate', 1000, ...
8                               'OutputDataType', 'Custom', ...
9                               'CustomOutputDataType', numerictype([], 10, 9), ..
10                              'ComplexOutput', true);
11
12 % HdlCosimulation System Object creation
13 fft_hdl = hdlcosim_fft_hdl;
14
15 % Simulate for 1000 samples
16 for ii=1:1000
17     % Read 1 sample from the sinus generator
18     ComplexSinus = step(SinGenerator);
19
20     % Send/receive 1 sample to/from the HDL FFT
21     [RealFft, ImagFft] = step(fft_hdl,real(ComplexSinus),imag(ComplexSinus));
22
23     % Store the FFT sample in a vector
24     ComplexFft(ii) = RealFft + ImagFft*1i;
25 end
26
27 % Discard the first 12 samples (initialization of the HDL FFT)
28 ComplexFft(1:12)=[];
29
30 % Display the FFT
31 plot(ComplexFft,'ro');
32 title('Fourier Coefficients in the Complex Plane');
33 xlabel('Real Axis');
34 ylabel('Imaginary Axis');
35
36 end
fft_tb Ln 27 Col 63 OVR

```

12. Run Cosimulation and Verify HDL Design

For ModelSim and Xcelium

Launch the HDL simulator by executing the script `launch_hdl_simulator_fft_hdl.m`.

```
launch_hdl_simulator_fft_hdl
```

When the HDL simulator is ready, return to MATLAB and start the simulation by executing the script **fft_tb.m**.

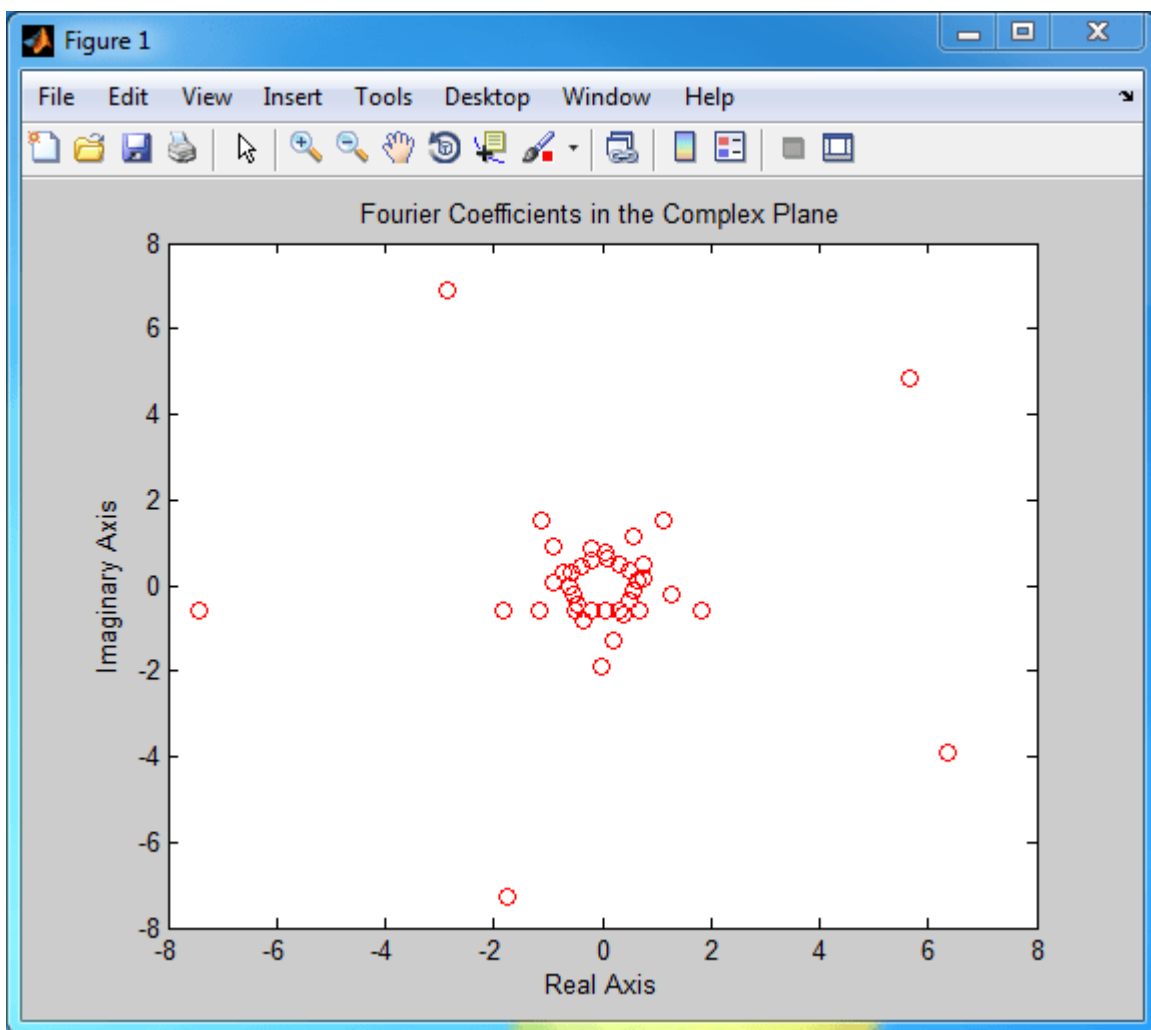
```
fft_tb
```

For Vivado Simulator

Start the simulation by executing the script **fft_tb.m**.

```
fft_tb
```

Verify the result from the plot in the test bench. The plot displays the Fourier coefficients in the complex plane.



See Also

- `hdlverifier.HDLCosimulation`
- `hdlverifier.VivadoHDLCosimulation`
- Cosimulation Wizard

Tutorial: Cosimulation Wizard for MATLAB Callback Function

This tutorial guides you through the basic steps for setting up an HDL Verifier™ application using Cosimulation Wizard.

Cosimulation Wizard is a Graphical User Interface (GUI) that guides you through the process of setting up cosimulation between MATLAB® or Simulink® and a Hardware Description Language (HDL) simulator. The supported HDL simulators include ModelSim® and Questa® from Mentor Graphics and Cadence Incisive®.

In this tutorial, we use MATLAB and ModelSim to verify a register transfer level (RTL) design of a raised cosine filter written in Verilog. The raised cosine filter is commonly used as a pulse shaping filter in digital communication systems. It produces no inter-symbol interference (ISI) for the input of modulated pulses.

A Verilog testbench is provided to generate the stimulus to the raised cosine filter. To verify the correctness of this HDL implementation, the testbench calls a MATLAB callback function that instantiates a reference model of the raised cosine filter. The testbench compares the output of the reference model to that of the RTL implementation.

The Cosimulation Wizard takes the provided Verilog files as its input. It also collects user input required for setting up cosimulation in each step. At the end of the tutorial, the Cosimulation Wizard generates a MATLAB script that compiles the HDL design, a MATLAB script that launches the HDL simulator for cosimulation, and a template for the MATLAB callback function. After modifying the generated template to implement the behavior of the raised cosine filter, you can verify the correctness of the RTL design.

For full content of this tutorial, please follow this link [Tutorial: Cosimulation Wizard for MATLAB Callback Function](#).

Get Started with Simulink HDL Cosimulation

Set up an HDL Verifier™ application using the Cosimulation Wizard in the Simulink® environment.

The Cosimulation Wizard is a graphical user interface (GUI) that guides you through the process of setting up cosimulation between MATLAB® or Simulink® and a Hardware Description Language (HDL) simulator.

In this example, you use Simulink and an HDL simulator to verify a design of a raised cosine filter written in Verilog. The raised cosine filter is commonly used as a pulse shaping filter in digital communication systems. It produces no inter-symbol interference (ISI) for the input of modulated pulses.

To verify the functionality of this raised cosine filter, a Simulink testbench is provided. This testbench generates input to the HDL design under test (DUT) and plots the waveforms of both input and output.

The Cosimulation Wizard takes the provided Verilog file of this raised cosine filter as its input. It also collects user input required for setting up cosimulation in each step. At the end of the example, the Cosimulation Wizard generates a Simulink block that represents the HDL design in the Simulink model, a MATLAB script that compiles HDL design, and a MATLAB script that launches the HDL simulator for cosimulation. During simulation, you can watch the input and output waveforms of this HDL filter in Simulink.

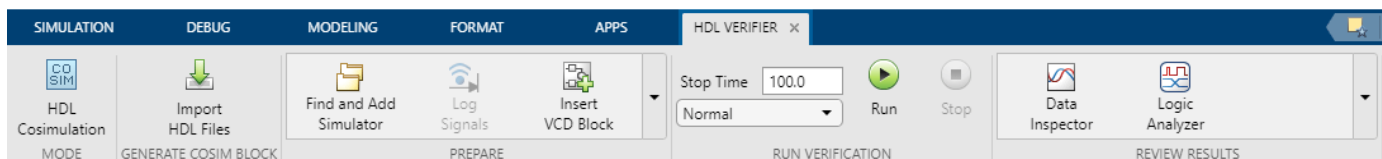
Requirements and Prerequisites

This example requires Simulink and one of these HDL simulators to verify a register transfer level (RTL) design.

- Vivado® simulator from Xilinx®
- ModelSim® or Questa® from Mentor Graphics®
- Xcelium® from Cadence®

Launch Cosimulation Wizard

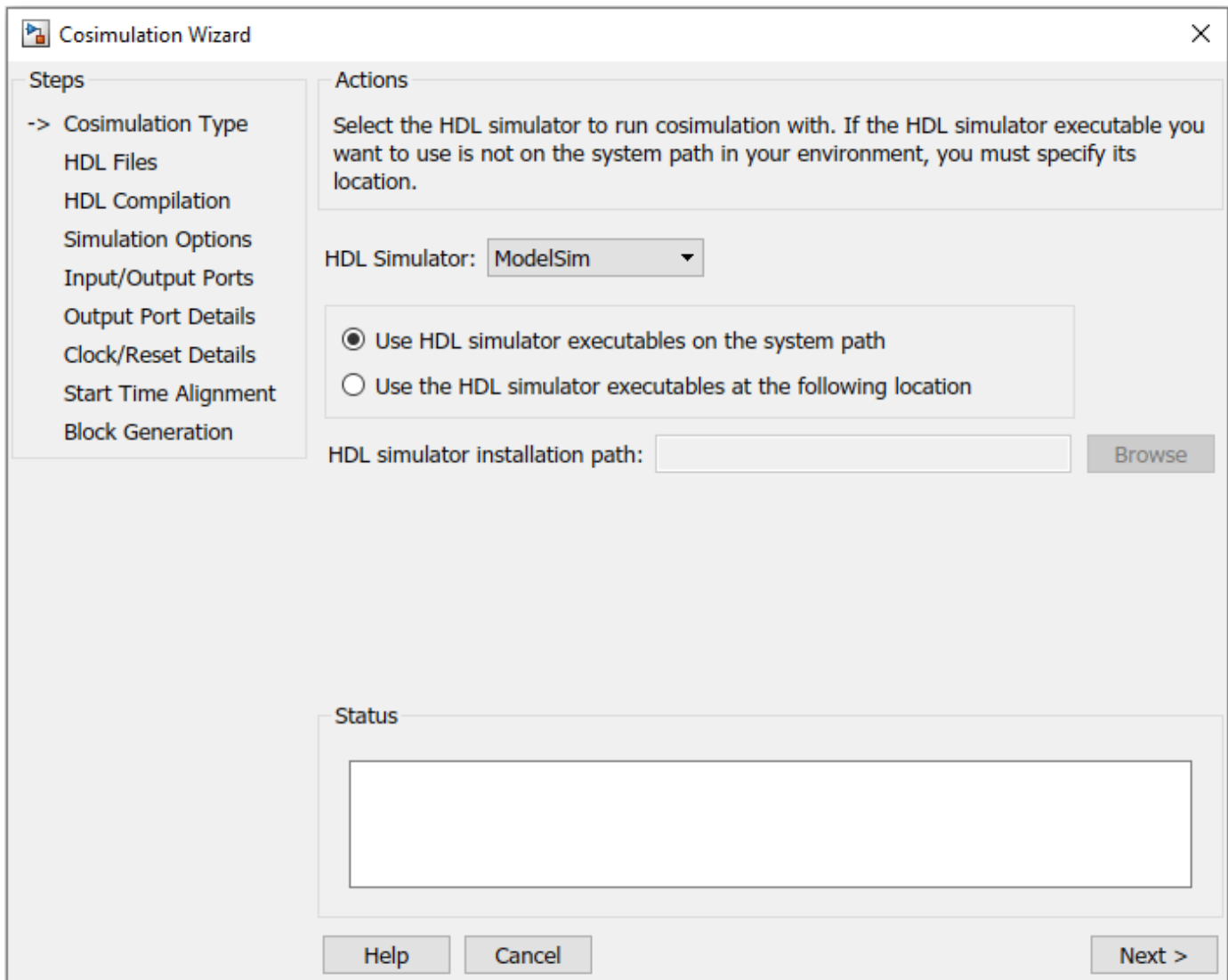
To launch the Cosimulation Wizard from the model, select the **Apps** tab in the Simulink toolstrip and click **HDL Verifier**. This action adds the **HDL Verifier** tab to the Simulink toolstrip. Then, in the **Mode** section, select **HDL Cosimulation**. Click **Import HDL Files** in the **Generate Cosim Block** section.



Configure HDL Cosimulation Block with Cosimulation Wizard

In the **Cosimulation Type** page, perform the following steps:

- 1.a). If you are using ModelSim or Questa, leave the **HDL Simulator** option as ModelSim.



b). If you are using Xcelium, change the **HDL Simulator** option to Xcelium.

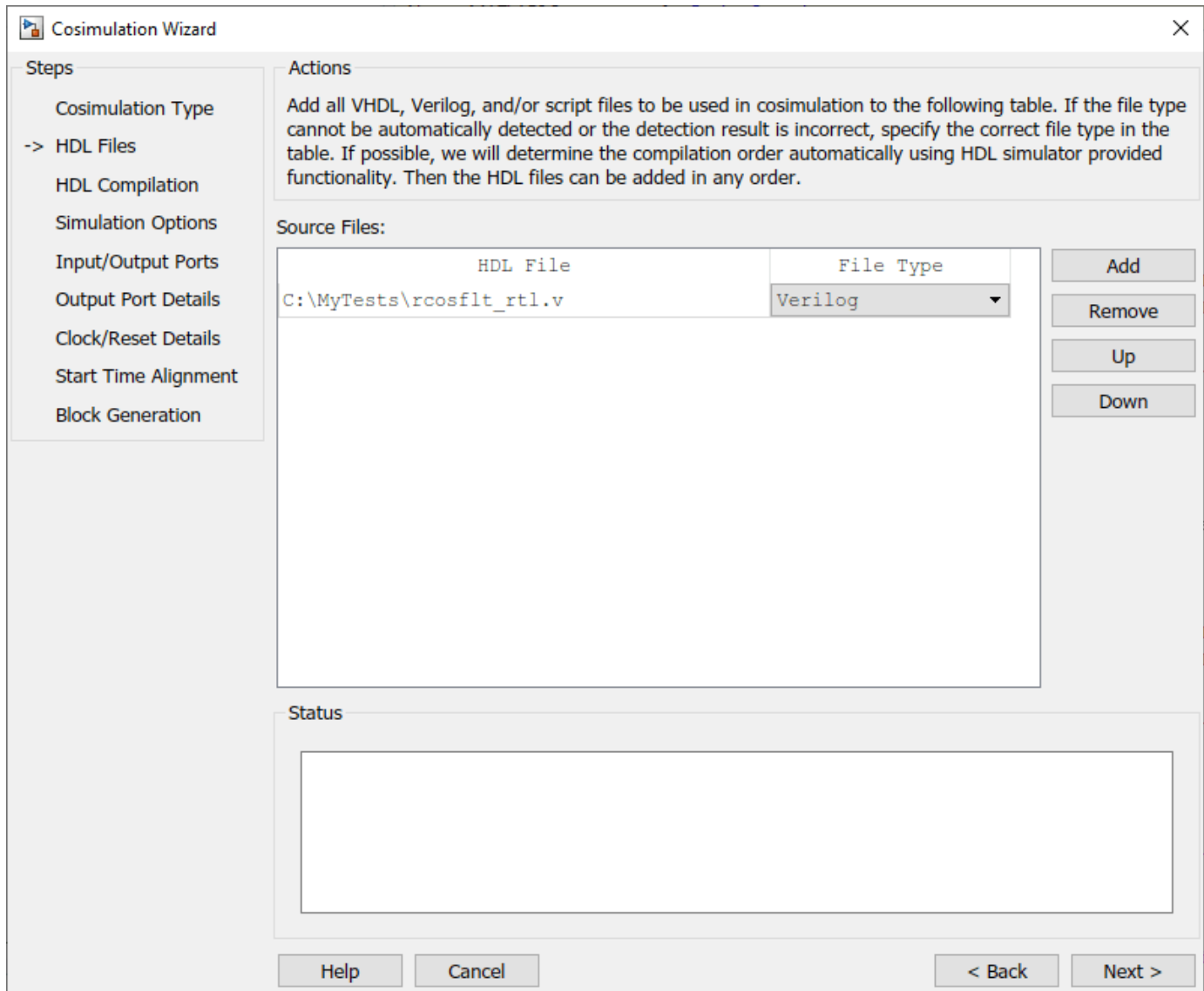
c). If you are using Vivado simulator, change the **HDL Simulator** option to Vivado Simulator.

2. Leave the default option **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path. If these executables do not appear on the path, click on the **Browse** button to specify the location of these executables.

Click **Next** to proceed to the HDL Files page.

In the **HDL Files** page, perform the following steps:

- 1 Click **Add** and select either `rcostflt_rtl.v` for Verilog or `rcosflt_rtl.vhd` for VHDL.
- 2 Review the file in the file list with the file type identified as you expected.

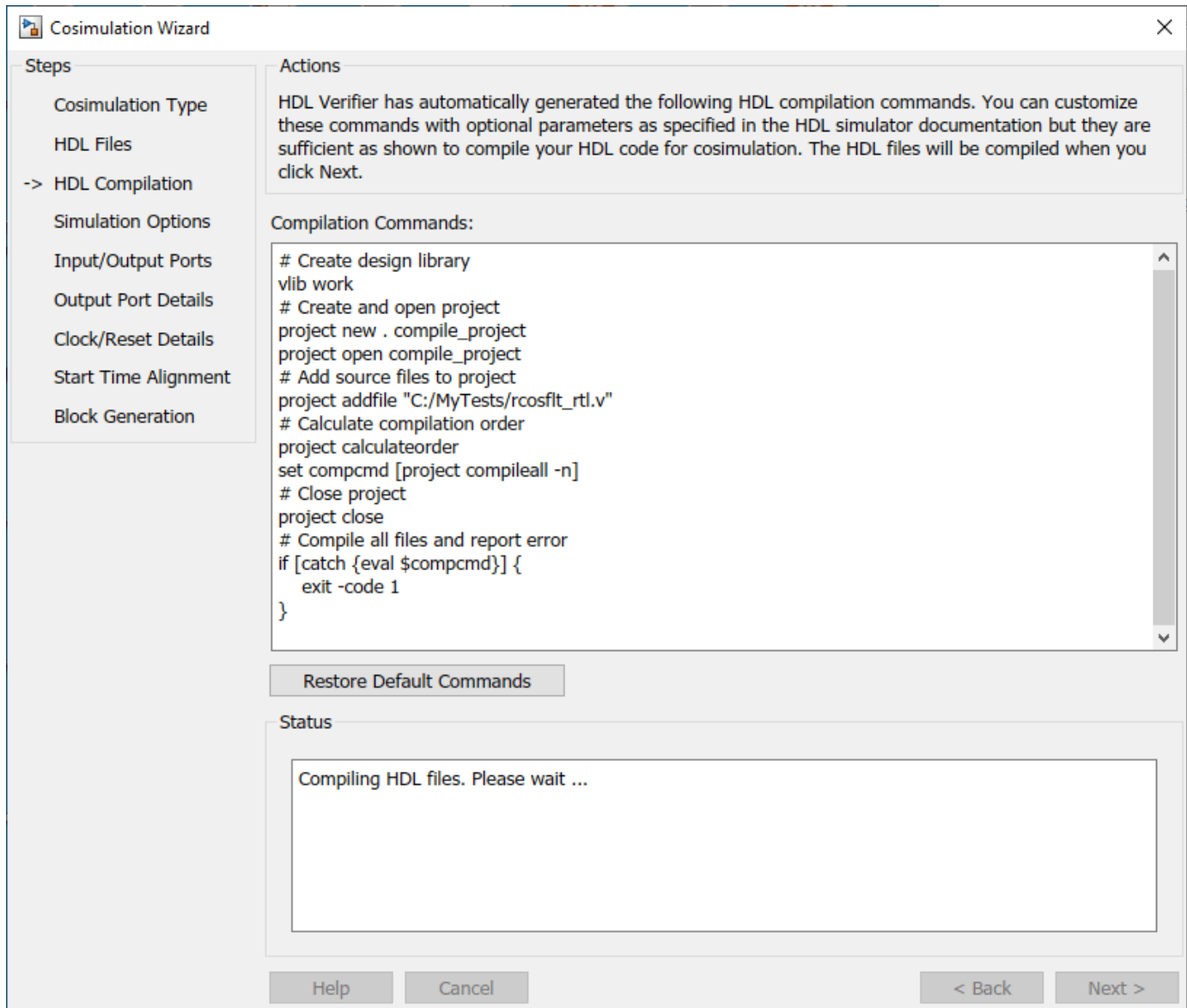


Click **Next** to proceed to the HDL Compilation page.

In the **HDL Compilation** page, the Cosimulation Wizard lists the default commands in the Compilation Commands window. You do not need to change these commands for this tutorial.

When you run the Cosimulation Wizard with your own code, you may add or change the compilation commands in this window.

The following figure shows the compilation commands for ModelSim.

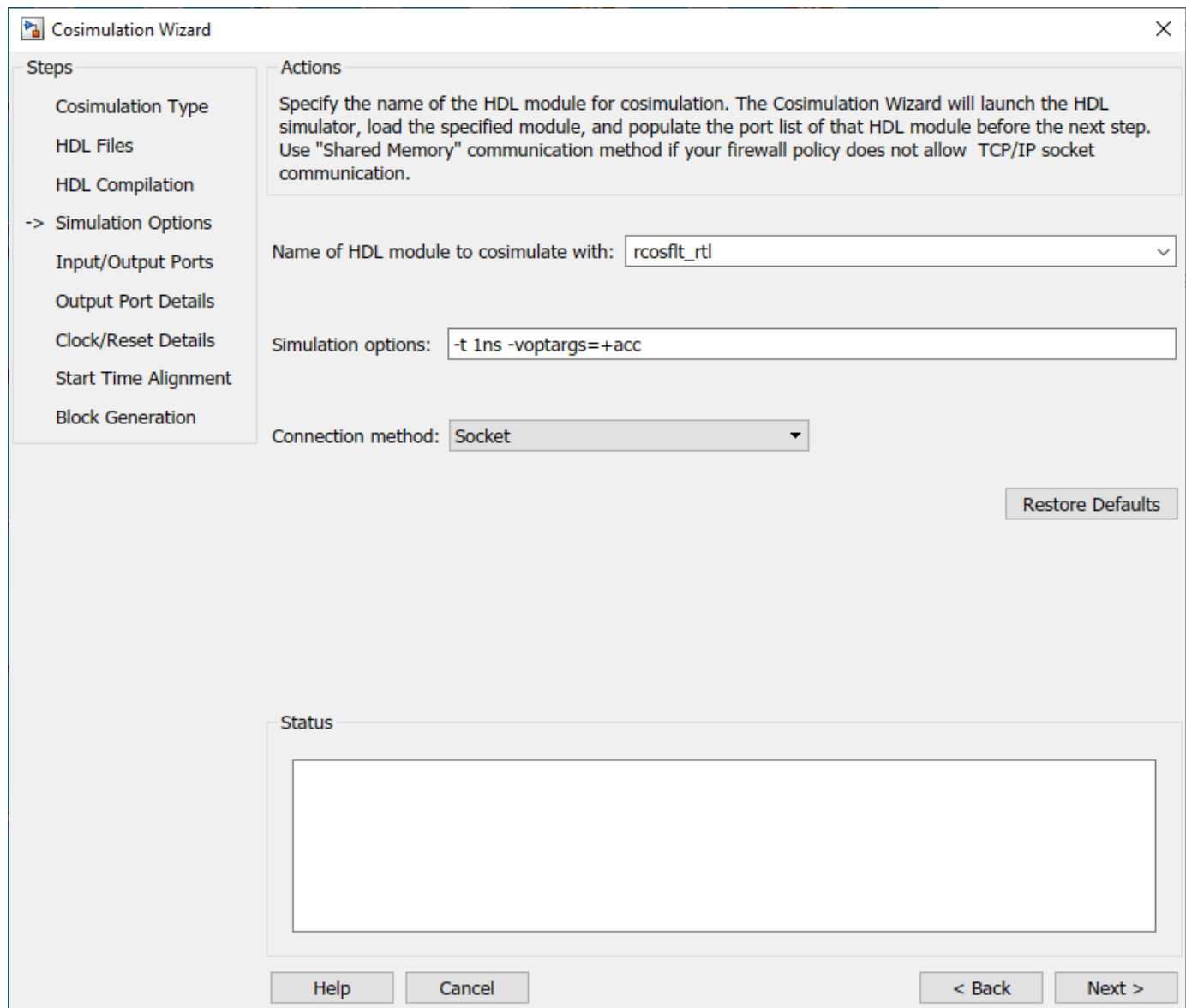


Click **Next** to proceed to the HDL Modules pane. This will in turn trigger the compilation. The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area.

In the **Simulation Options** pane, perform the following steps.

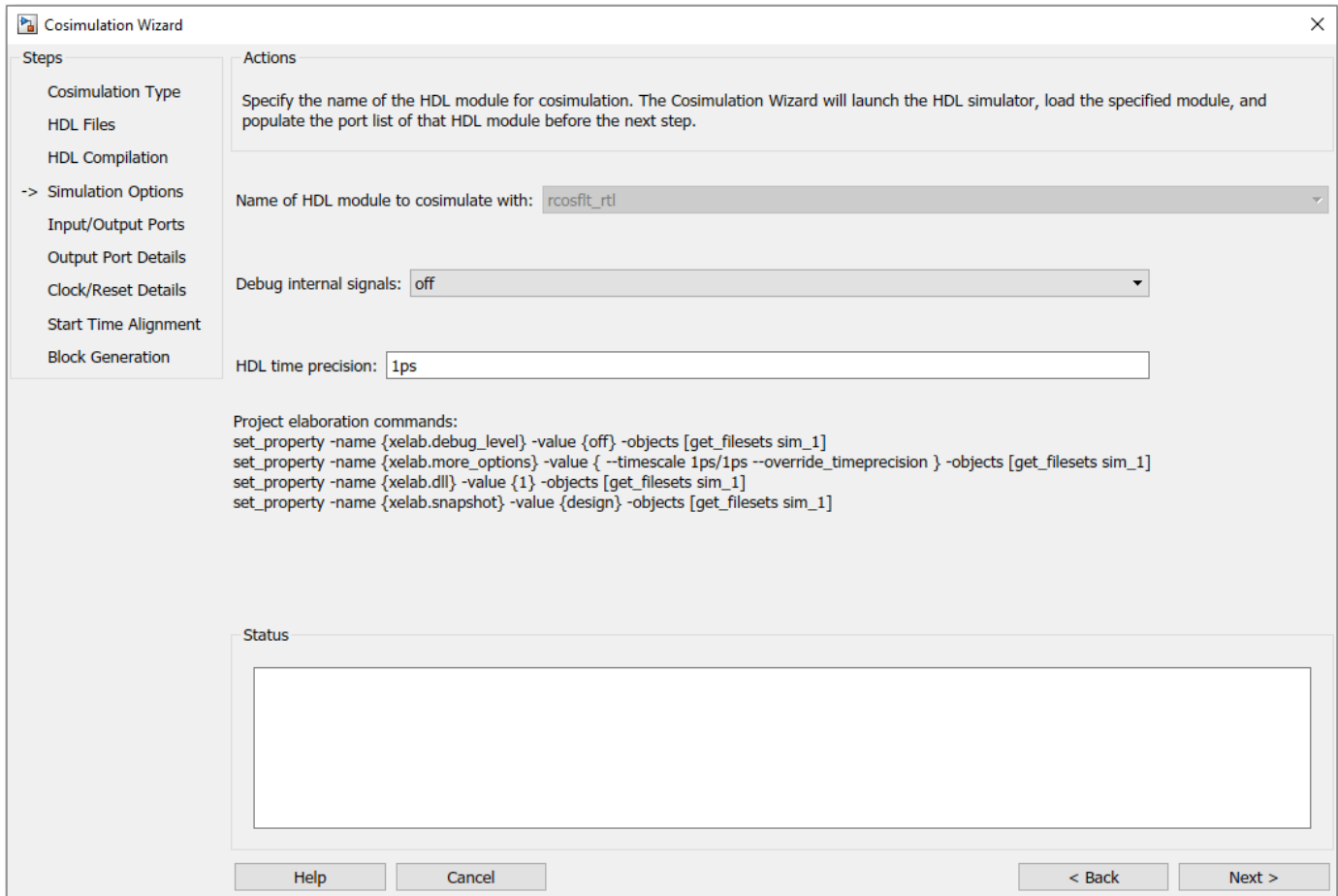
ModelSim or Xcelium

- 1 a) Specify the name of HDL module/entity for cosimulation. From the drop-down list, select `rcosflt_rtl`. This module is the Verilog/VHDL module you use for cosimulation. If you do not see `rcosflt_rtl` in the drop-down list, you can enter the file name manually.
- 2 b) For **Connection method**, select Shared Memory if your firewall policy does not allow TCP/IP socket communication.



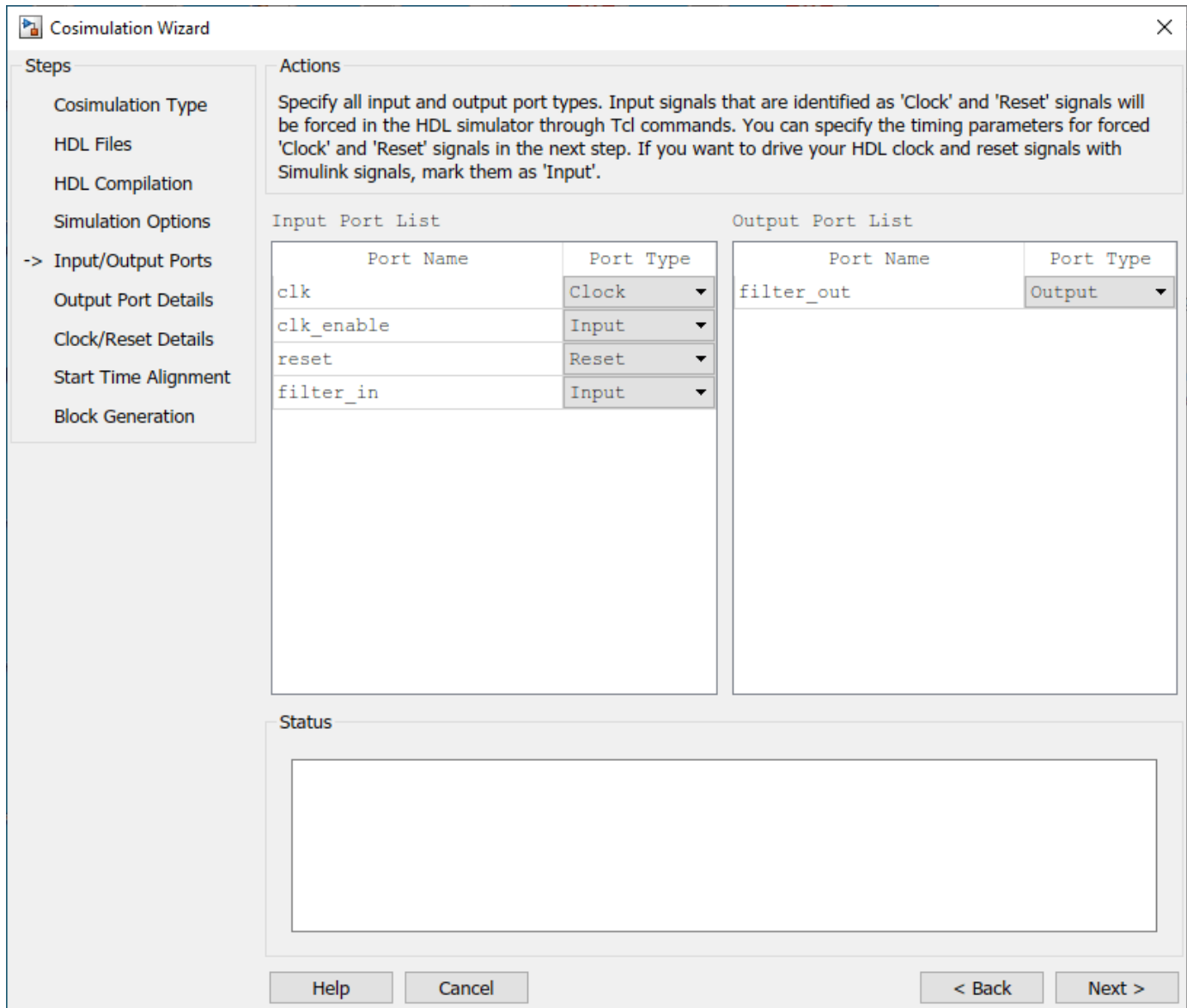
Vivado Simulator

- 1 By default, name of the module is set to `rcosflt_rtl`. This module is the Verilog/VHDL module you use for cosimulation.
- 2 Set **Debug internal signals** to off and **HDL time precision** to 1ps for this example.



Click **Next** to proceed to the Simulink Ports pane. The Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. After the wizard launches the HDL simulator, the wizard populates the input and output ports on the Verilog/VHDL module `rcosflt_rtl` and displays them in the next step.

In the **Specify Port Types** step, the Cosimulation Wizard displays two tables containing the input and output ports of `rcosflt_rtl`, respectively.



The Cosimulation Wizard attempts to identify the port type of each port. If the wizard incorrectly identifies a port, you can change the port type using these tables.

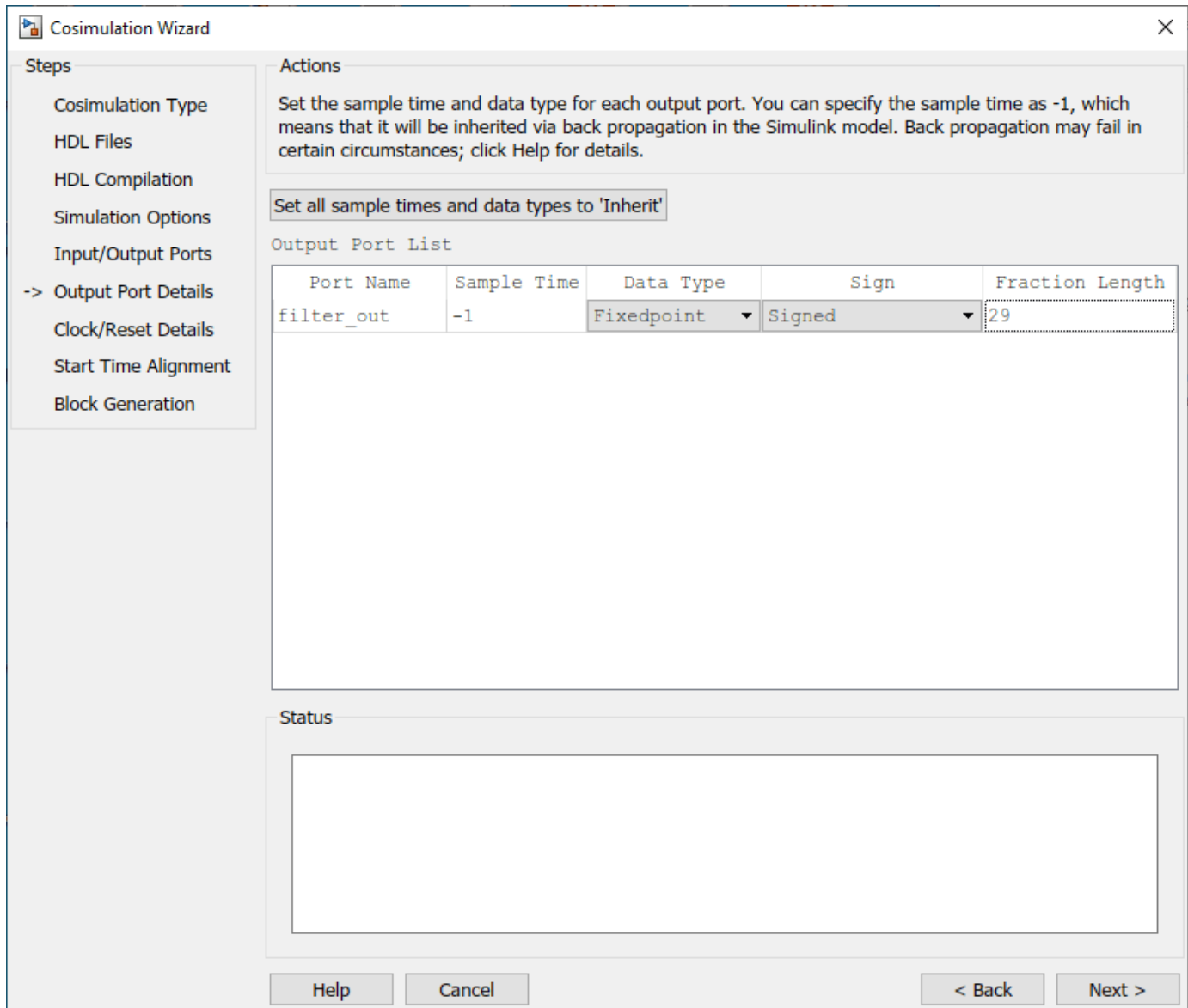
- 1 For input ports, you can select from **Clock**, **Reset**, **Input**, or **Unused**. HDL Verifier connects only the input ports marked **Inputs** to Simulink during cosimulation.
- 2 HDL Verifier connects output ports marked **Output** with Simulink during cosimulation. The wizard and Simulink ignore those output ports marked **Unused** during cosimulation.
- 3 You can change the parameters for signals identified as **Clock** and **Reset** at a later step.

Accept the default port types and click **Next** to proceed to the Output Port Details page.

In the **Output Port Details** page, perform the following steps:

- 1 Set the sample time of `filter_out` to -1 to inherit via back propagation.

- 2 You can see from the Verilog code that the Cosimulation Wizard represents the output in a `sfixed34_En29` format. Change the following fields:
- Data Type to Fixedpoint
 - Sign to Signed
 - Fraction Length to 29



Click **Next** to proceed to the Clock/Reset Details page.

In the **Clock/Reset** page, perform the following steps.

ModelSim or Xcelium

- 1 Set HDL time unit to ns.

- 2 Set clock period to 20.
- 3 Leave or set active edge to Rising.
- 4 Leave or set reset initial value to 1.
- 5 Set reset signal duration to 15.

Cosimulation Wizard

Steps

- Cosimulation Type
- HDL Files
- HDL Compilation
- Simulation Options
- Input/Output Ports
- Output Port Details
- > Clock/Reset Details
- Start Time Alignment
- Block Generation

Actions

Set clock and reset parameters here. The time in these tables refers to time in the HDL simulator.

HDL time unit ns

Clocks

Clock Name	Period(ns)	Active Edge
clk	20	Rising

Resets

Reset Name	Initial Value	Duration(ns)
reset	1	15

Status

Help Cancel < Back Next >

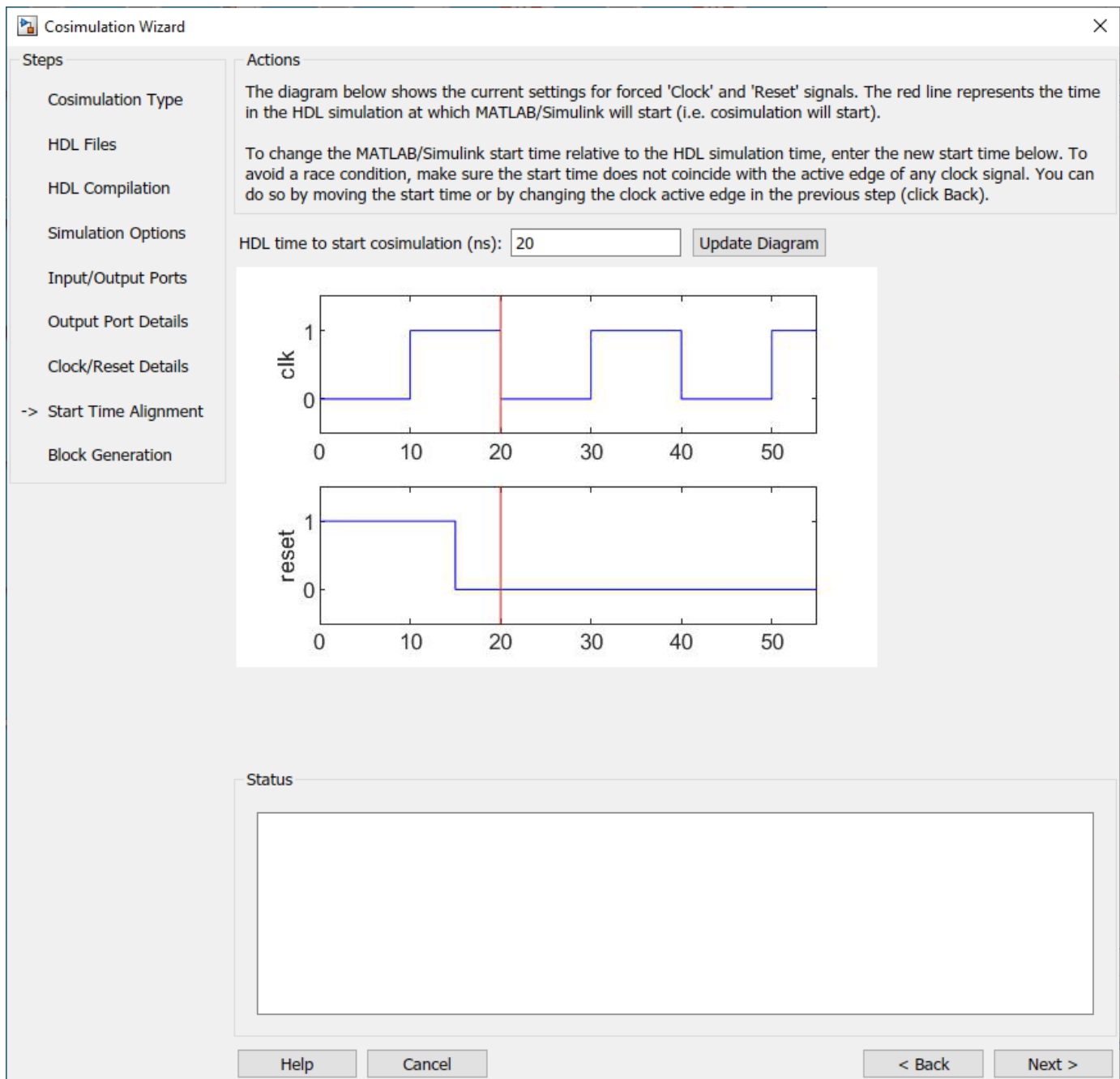
Vivado Simulator

- 1** Set the HDL time unit to **ps**.
- 2** Set the clock period to **20**.
- 3** Leave or set the active edge to **Rising**.
- 4** Leave or set the reset initial value to **1**.
- 5** Set the reset signal duration to **15**.

Click **Next** to proceed to the Start Time Alignment page.

The Start Time Alignment page displays a plot for the waveforms of clock and reset signals. The Cosimulation Wizard shows the HDL time to start cosimulation with a red line. The start time is also the time at which the Simulink gets the first input sample from the HDL simulator.

In the **Start Time Alignment** page, set the alignment. The active edge of our clock is a rising edge. Thus, at time 20 ns in the ModelSim or Xcelium (20 ps in the Vivado simulator), the registered output of the raised cosine filter is stable. No race condition exists, and the default HDL time to start cosimulation is what we want for this simulation. You do not need to make any changes to the start time.



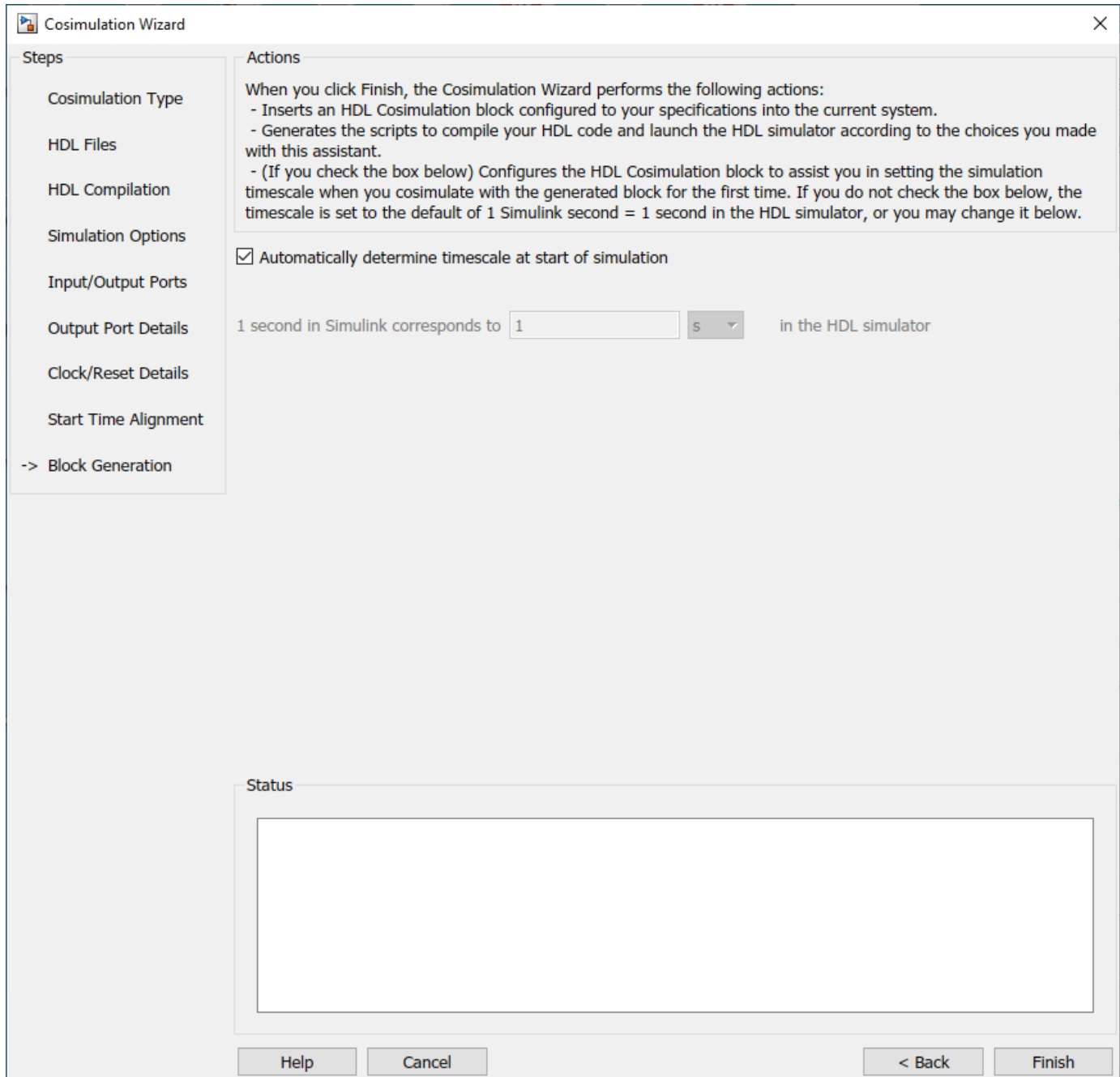
Click **Next** to proceed to Block Generation.

Before you generate the HDL Cosimulation block, you have the option to determine the timescale before you finish the Cosimulation Wizard. Alternately, you can instruct HDL Verifier to calculate a timescale later. Timescale calculation by the verification software occurs after you connect all the input/output ports of the generated HDL Cosimulation block and start simulation.

In the **Block Generation** page, perform the following steps.

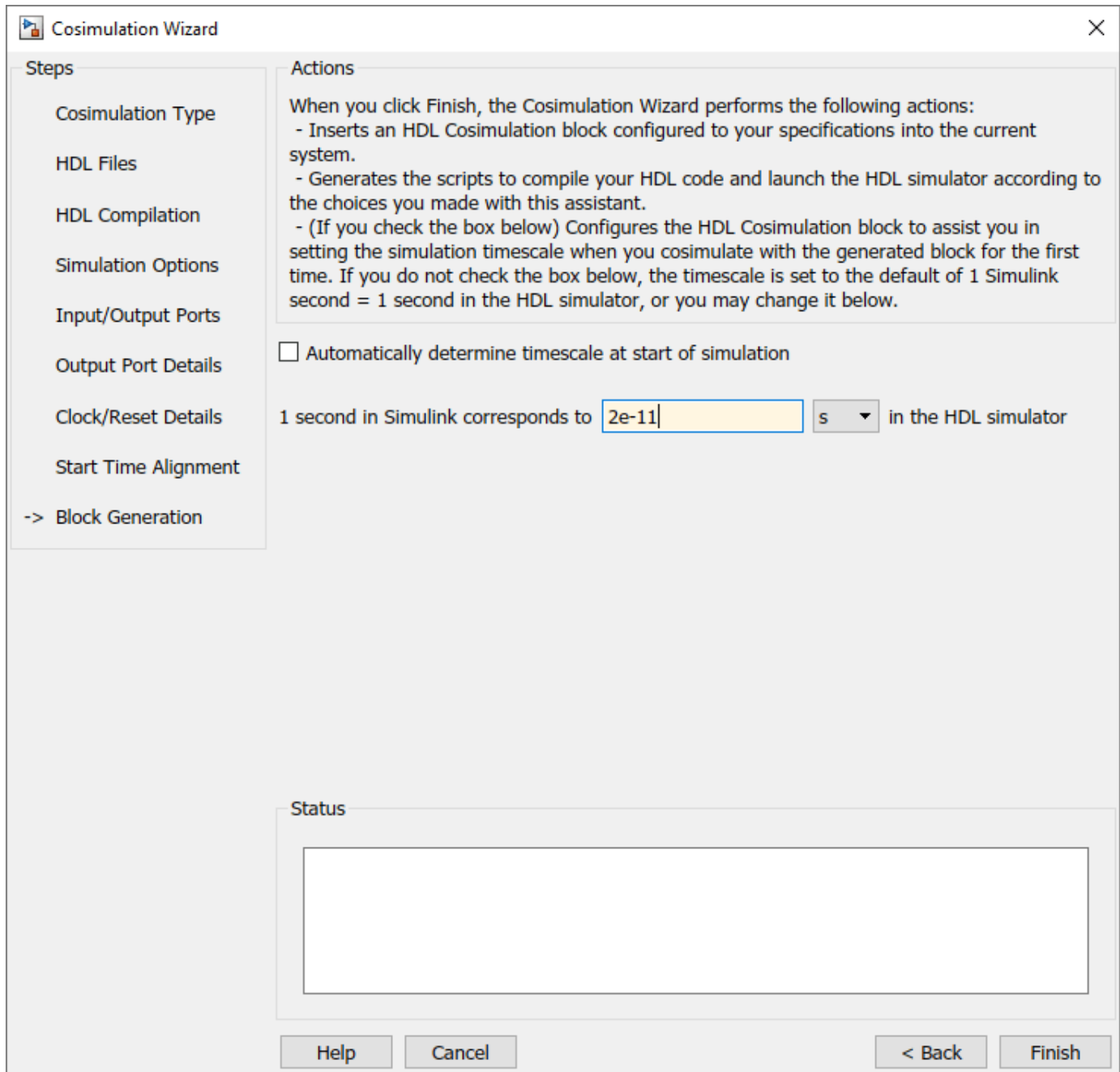
ModelSim or Xcelium

Leave **Automatically determine timescale at start of simulation** selected (default). Later, you will have the opportunity to view the calculated timescale and change that value before you begin simulation.



Vivado Simulator

Uncheck **Automatically determine the timescale at start of simulation** and set the timescale by setting **1 second in Simulink corresponds to** to $2e-11$ s in the HDL simulator.



Click **Finish** to complete the Cosimulation Wizard session.

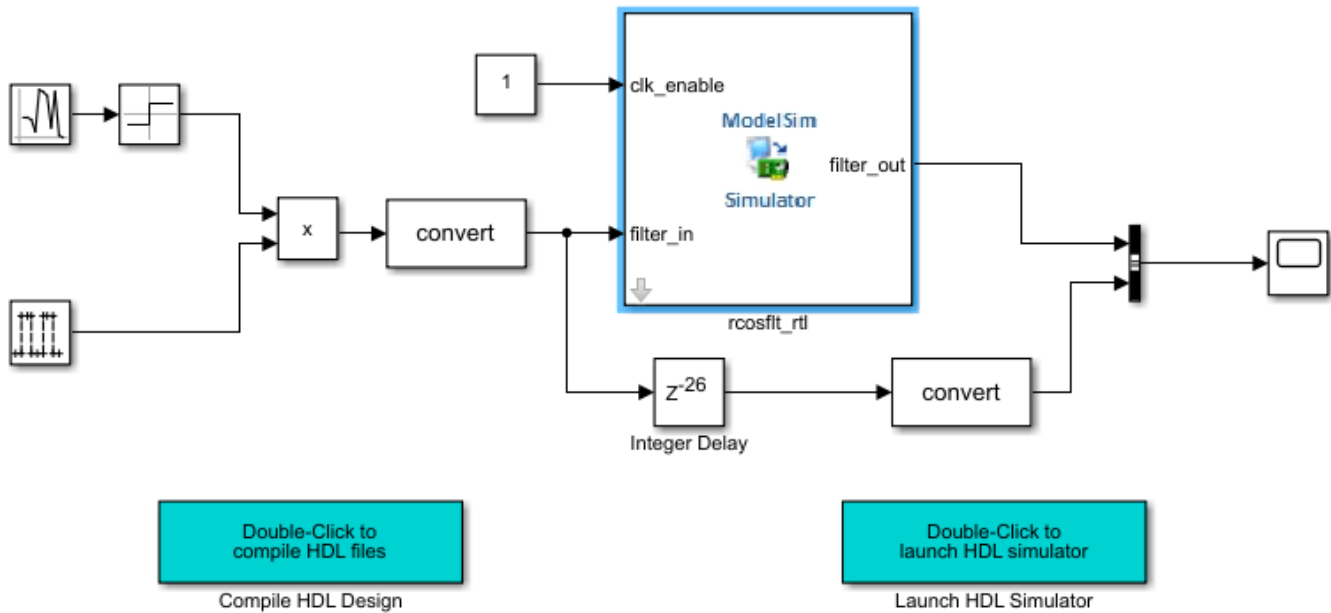
Create Test Bench to Verify HDL Design

In this example, the Simulink test bench model `rcosflt_tb` has been provided. After you click **Finish** in the Cosimulation Wizard, Simulink inserts the following items at the center of the model canvas:

- An HDL Cosimulation block

- A block to recompile the HDL design (contains a link to a script that is launched by double-clicking the block)
- A block to launch the HDL simulator (contains a link to a script that is launched by double-clicking the block) (For Modelsim and Xcelium only)

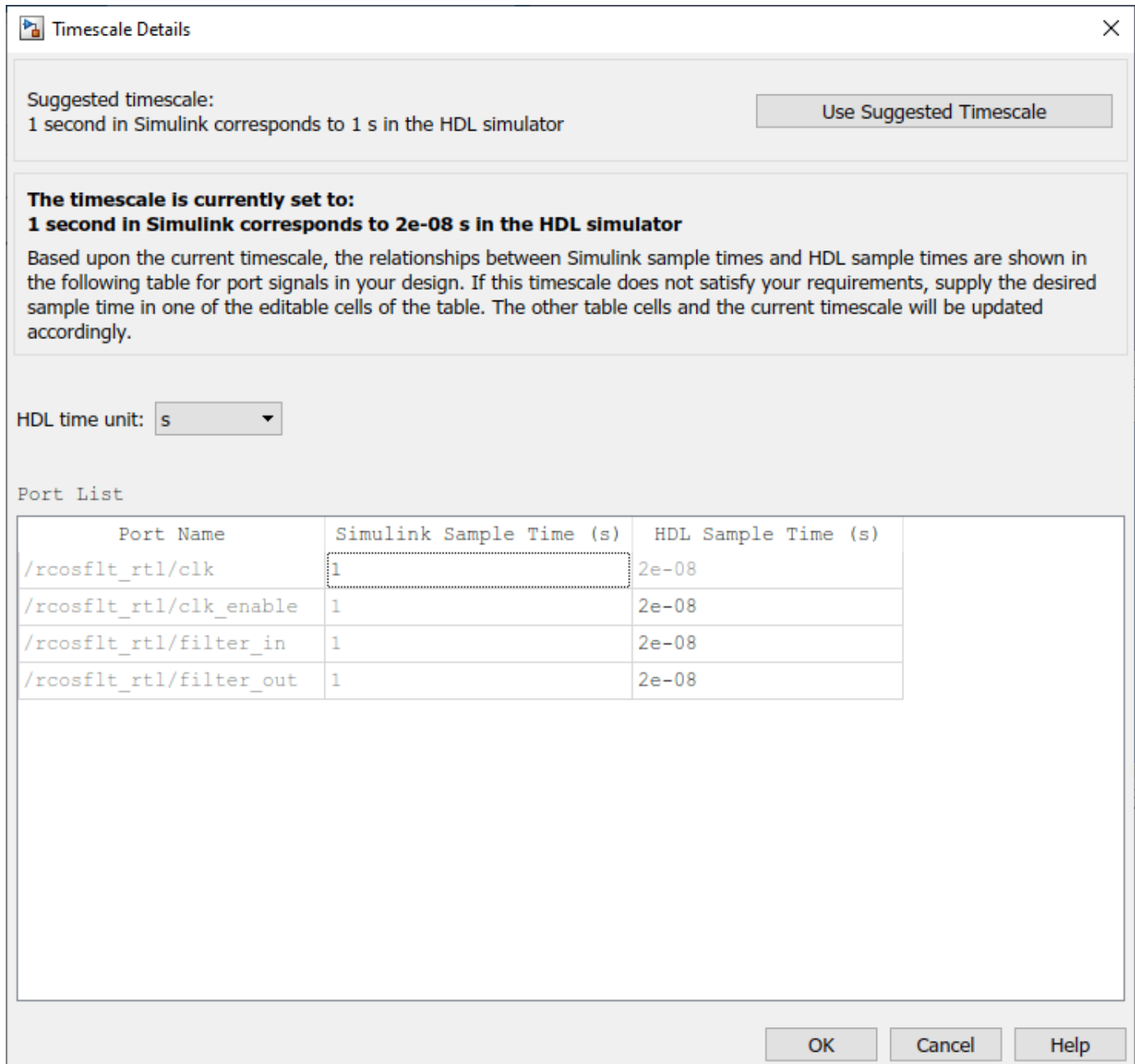
Position the HDL Cosimulation block so that the constant and convert blocks line up as inputs to the HDL Cosimulation block and the bus lines up as output. Connect the blocks. Your model now looks similar to that in the following figure.



Run Cosimulation and Verify HDL Design

ModelSim or Xcelium

- 1 Launch the HDL simulator by double-clicking the block labeled **Launch HDL Simulator**.
- 2 When the HDL simulator is ready, return to Simulink and start simulation.
- 3 Determine timescale. Recall that you selected **Automatically determine timescale at start of simulation** option on the last page of the Cosimulation Wizard. When doing so, HDL Verifier launches the Timescale Details graphical interface instead of starting the simulation. Both the HDL simulator and Simulink sample the `filter_in` and `filter_out` ports at 1 second. However, their sample time in the HDL simulator should be the same as the clock period (20 ns). Change the Simulink sample time of `/rcosflt_rtl/clk` to 1 (seconds), and press **Enter**. The wizard then updates the table. The following figure shows the new timescale: 1 second in Simulink corresponds to 2e-008 s in the HDL simulator.

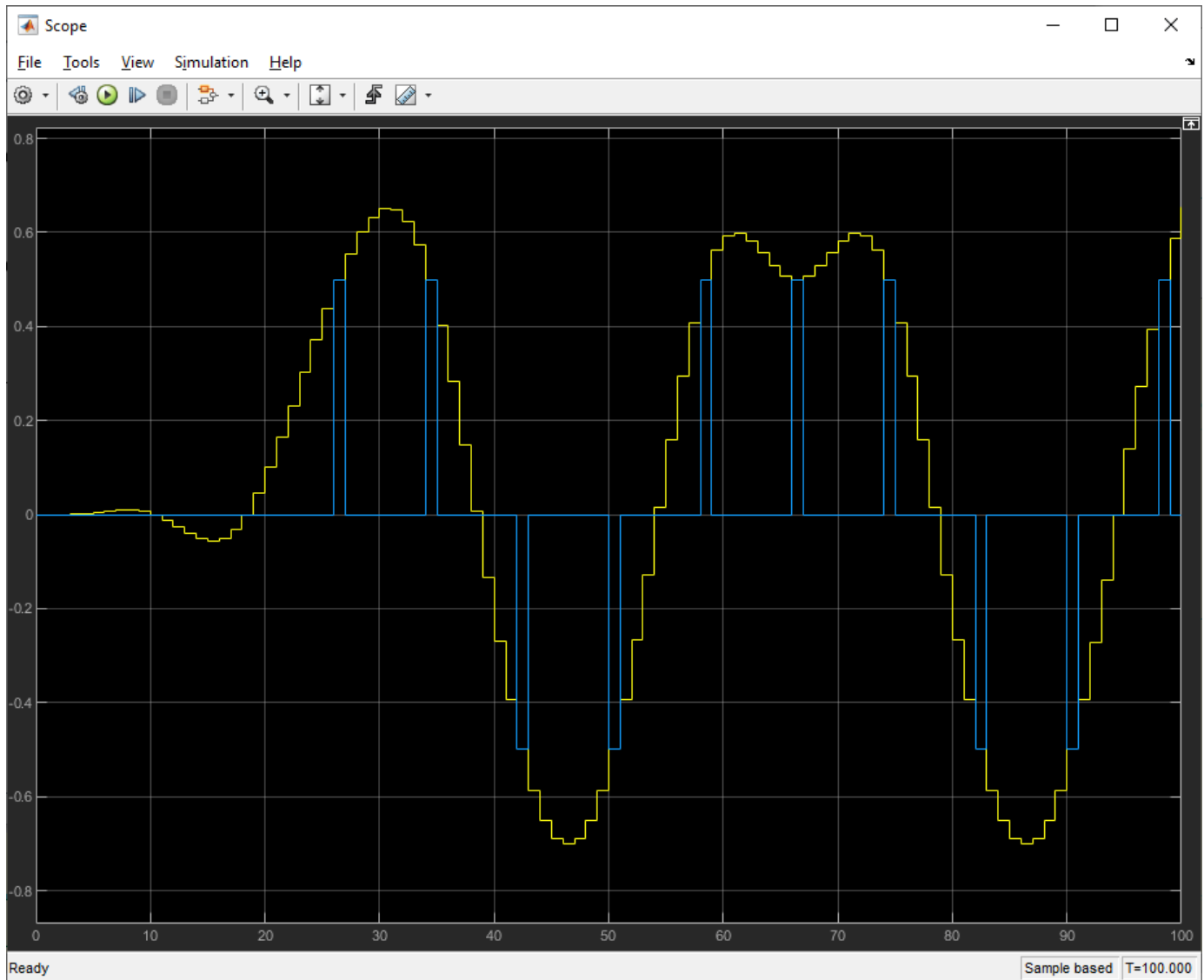


Click **OK** to close the Timescale Details dialog. Restart Simulink simulation and verify the results from the scope in the test bench model.

Vivado Simulator

In the Simulink toolstrip, on the **Simulation** tab, click **run** to start the simulation. There is no separate process for the HDL simulation, since the Simulink block executes a single process with a shared DLL.

The scope displays both the delayed version of input to raised cosine filter and that filter's output. If you sample the output of this filter output directly, no inter-symbol-interference occurs.



See Also

- `hdlverifier.HDLCosimulation`
- `hdlverifier.VivadoHDLCosimulation`
- Cosimulation Wizard
- HDL Cosimulation

Get Started with SystemVerilog DPI Component Generation

This example shows how to generate a SystemVerilog DPI component from a proportional-integral-derivative (PID) controller in a Simulink® model and how to export the component to an HDL simulator.

Requirements and Prerequisites

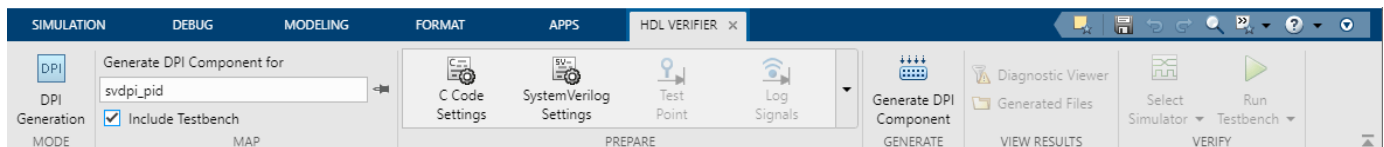
These products are required for this example.

- One of these supported HDL simulators: Mentor Graphics® ModelSim®/QuestaSim® or Cadence® Xcelium™
- One of these supported C compilers: Microsoft® Visual C++ or GNU GCC

Set Up Model for Code Generation

To set up the model with the correct target file, open the **HDL Verifier** app by clicking its app icon from the **Apps** tab. This action adds the **HDL Verifier** tab to the Simulink Toolstrip. Then, in the **Mode** section select **DPI Component Generation** to set the system target file of the model to "systemverilog_dpi_grt.tlc". If Embedded Coder® is installed, the target file is set to "systemverilog_dpi_ert.tlc" instead.

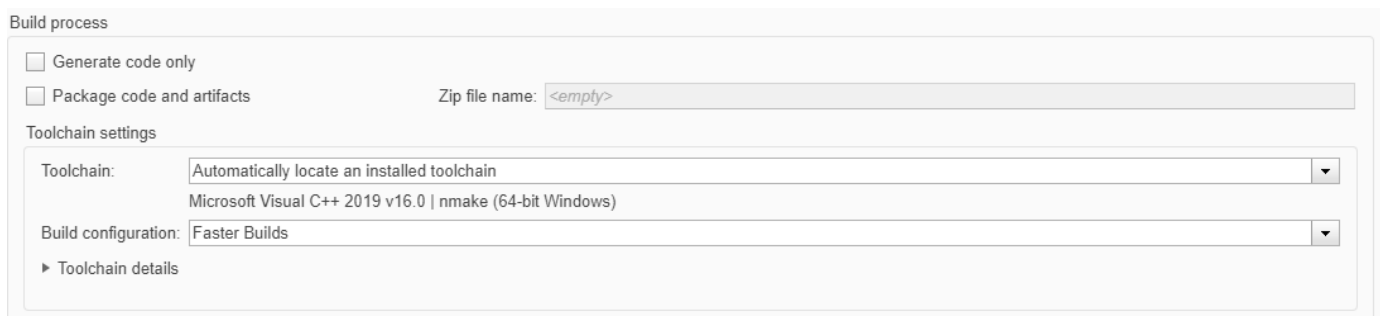
To generate a SystemVerilog test bench for the DPI component, in the **Map** section on the **HDL Verifier** tab, select **Include Testbench**.



Open the configuration parameters for the model, by clicking **C Code Settings** in the **Prepare** section.

In the **Toolchain settings** section, select one of the Visual Studio versions if you are using Windows or one of the GCC toolchains if you are using Linux.

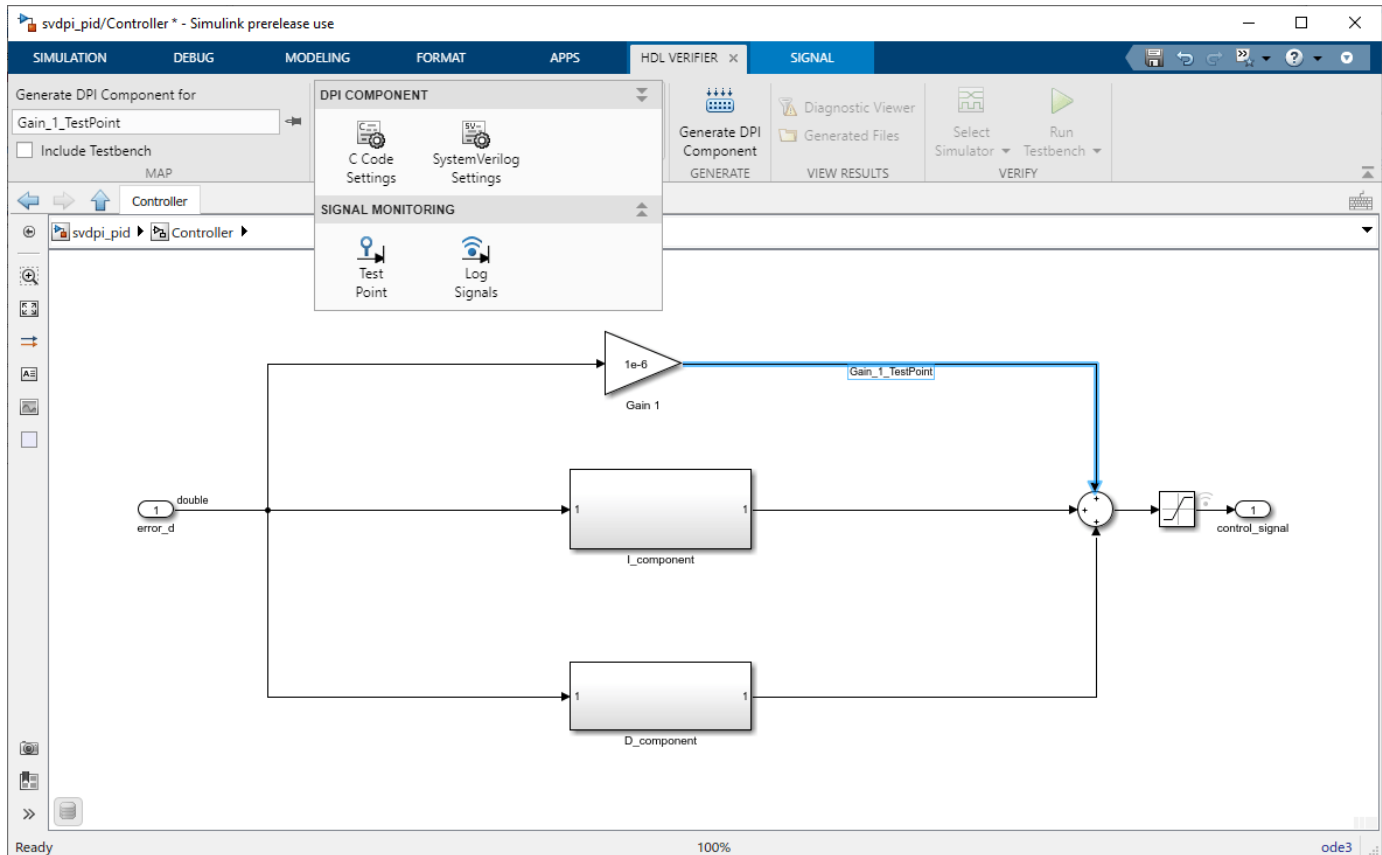
Clear **Generate code only**.



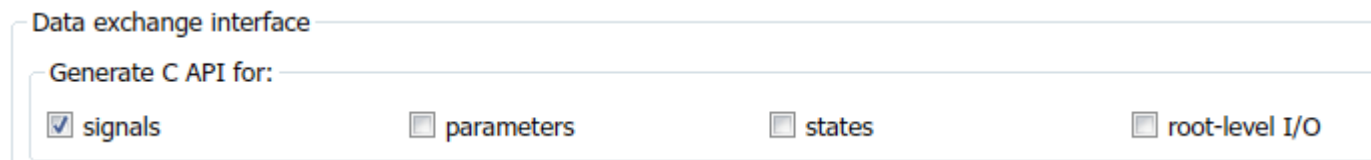
Select Internal Signals for Test Point Logging (Optional)

To access internal signals of the DPI component in the SystemVerilog environment, use the DPI-C test point logging capability.

Double-click a signal you want to access to highlight the signal and to enter a signal name for it. To mark the signal as a test point, from the **Prepare** section of the Simulink Toolstrip, click **Test Point**. To also capture the test vector of the internal signal and playing the vector back in the generated testbench, click **Log Signals** in the gallery while the signal is highlighted.



Enable C API options. In the Configuration Parameters dialog box, on the **Code Generation > Interface** tab, select **signals**.



Choose the interface of the SystemVerilog functions that you want to use to access the test points. Customize the generated DPI component by using option on the **Code Generation > SystemVerilog DPI** tab.

Under **Test Point Access Functions**, when you set **Generate access function to test point** to **None**, test points that are marked are ignored, and no access functions are generated. Changing the

value to One function per test point enables you to access each test point independently. This figure shows the interface generated for this example.

```
DPI_Gain_1_TestPoint(inputchandle objhandle,inout real Gain_1_TestPoint);
DPI_Gain_4_TestPoint_And_Logging(inputchandle objhandle,inout real Gain_4_TestPoint_And_Logging);
```

Using the value One function for all test points enables you to access all of the test points with one function call.

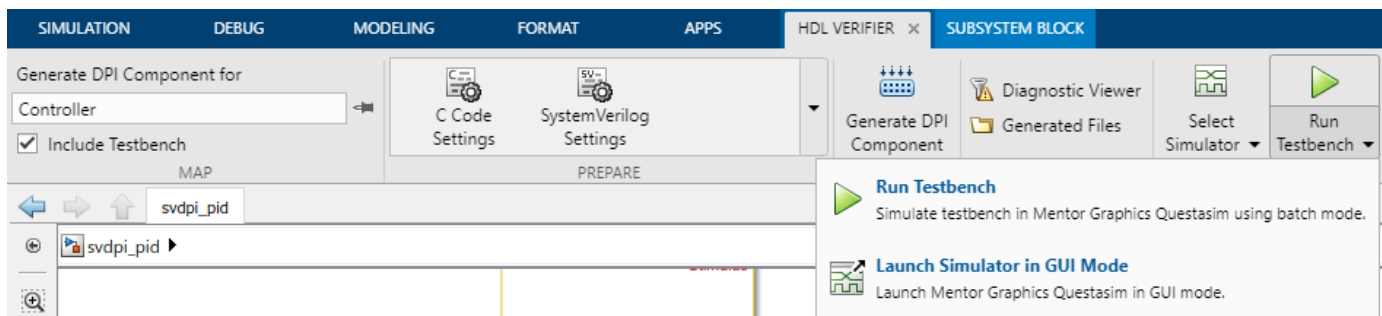
Generate SystemVerilog DPI Component

- 1 In the "svdpi_pid" Simulink model, select the Controller Subsystem block. In the **Generate** section on the toolstrip, click **Generate DPI Component**.
- 2 Click **Build** in the dialog box that appears.
- 3 The SystemVerilog component is generated as "Controller_build/Controller_dpi.sv". When code generation is complete, examine the new component.

Run Generated Test Bench

To select an HDL simulator, click **Select Simulator** in the **Verify** section of the toolstrip, and select an HDL simulator or add a simulator to the path.

To simulate the SystemVerilog testbench in batch mode, click **Run Testbench**. Alternatively, you can execute the simulation in GUI mode by clicking **Run Testbench > Launch Simulator in GUI mode**.



When the simulation finishes, this text prints in the console.

```
*****TEST COMPLETED (PASSED)*****
```

Getting Started with Customizing Generated SystemVerilog Code

This example shows you how to customize the generated SystemVerilog code in the SystemVerilog DPI Component Generation process.

Requirements and Prerequisites

Products required for this example:

- MATLAB®
- Simulink®
- Simulink Coder™
- Mentor Graphics® ModelSim®/QuestaSim®
- One of the supported C compilers: Microsoft® Visual C++, or GNU GCC

Background

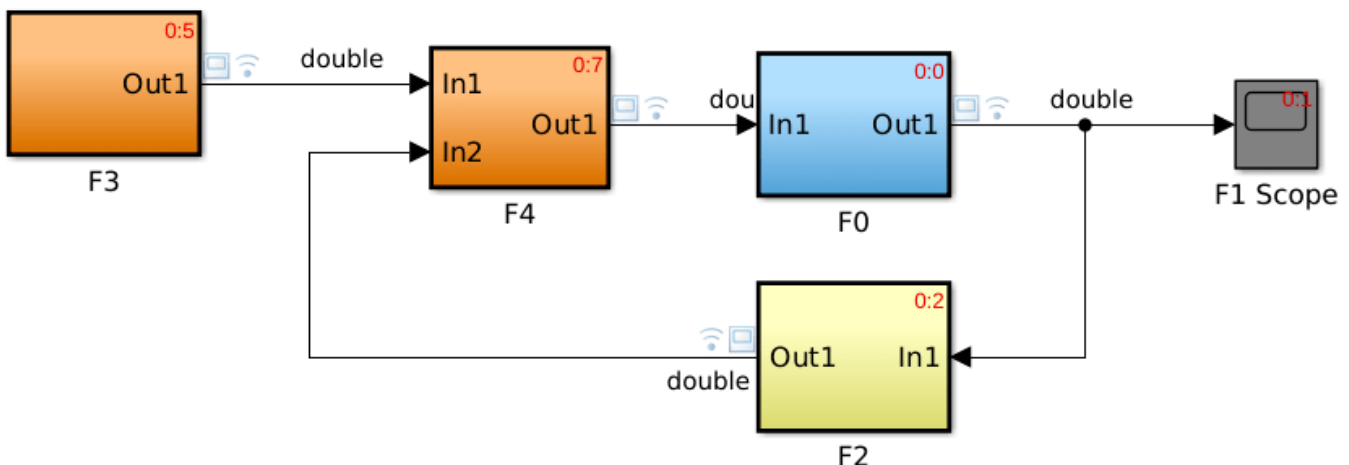
If the default generated SystemVerilog code does not meet your requirements, you have the option to customize the generated code. This example shows how to customize the generated code.

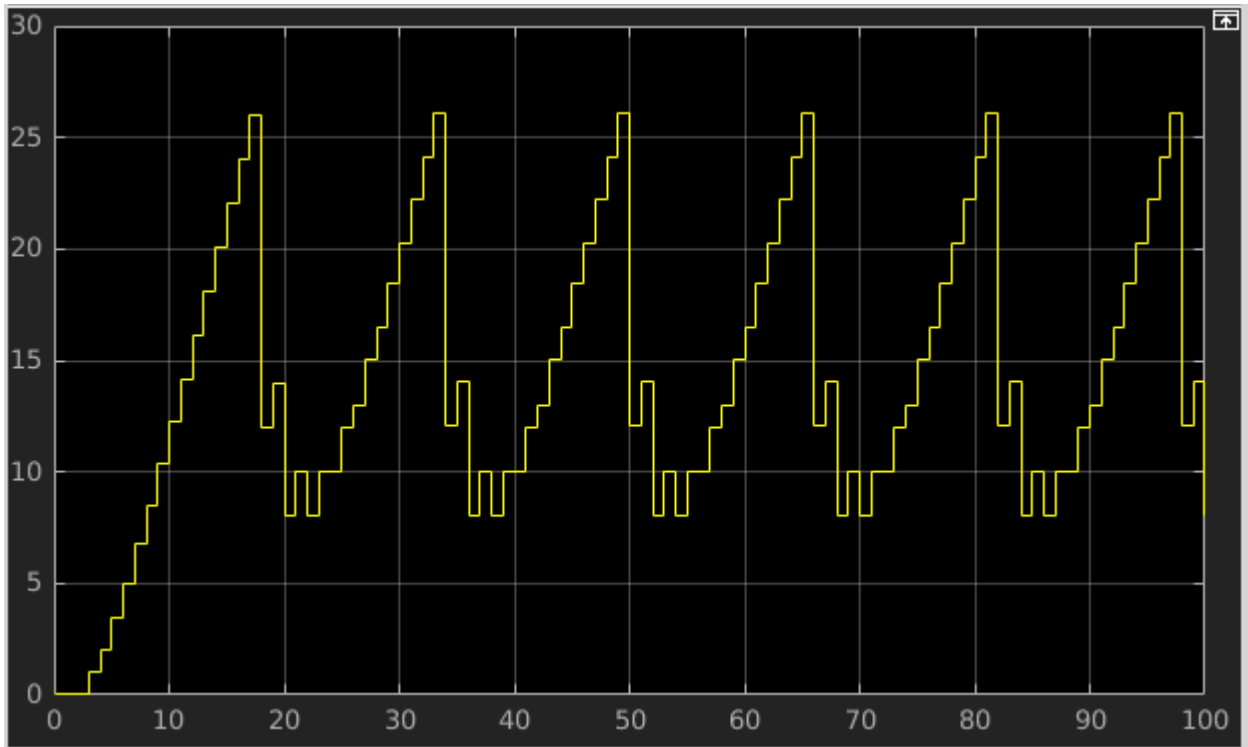
In the following Simulink model the generated code corresponding to subsystems, F0, F2, F3, and F4, are exported separately as SystemVerilog modules. By default, each module contains three control signals: clock, clock enable, and reset. In HDL simulation, the subsystem update and output functions will be called on the rising edge of the clock signal. An event scheduler would trigger those update and output function in HDL simulator in the same sequence as in Simulink model.

Open Example

Run the following code to open the design

```
open_system('svdpi_SimpleFeedBack');
```





Set Up Model for Code Generation

Open the Simulink Model Configuration Parameter panel from the `svpid_SimpleFeedBack` model. Set the following parameters:

Select Code Generation -> System Target File. Click Browse button and select "systemverilog_dpi_grt.tlc".

In Toolchain setting, select one of the Visual Studio versions if you are using Windows, or one of the GCC toolchains if you are using Linux. Make sure that option "Generate code only" is unchecked.

Select Code Generation -> SystemVerilog DPI, and check the option "Customize generated SystemVerilog code". Make sure that "Source file template" is set to default template "svdpi_event.vgt".

You can also click the "Edit" button to view the default SystemVerilog template.

Now we are done with the Configuration Parameter panel. Click "OK" button to close it.

Generate SystemVerilog DPI Component

In the "svdpi_SimpleFeedBack" model, generate C code for subsystems F0, F2, F3, and F4. You can generate C code from the command-line by run the following commands in MATLAB:

```
slbuild('svdpi_SimpleFeedBack/F0')
slbuild('svdpi_SimpleFeedBack/F2')
slbuild('svdpi_SimpleFeedBack/F3')
slbuild('svdpi_SimpleFeedBack/F4')
```


Now the C code for those subsystems are generated in subdirectories "F0_build", "F2_build", "F3_build" and "F4_build", respectively.

Test Bench Files

In the testbench "SimpleFeedback_tb.sv" the control signal is connected so that it flows from modules with higher execution order in Simulink to modules with lower execution order in Simulink.

Run the Generated Test Bench

- Start ModelSim/QuartaSim in GUI mode and change the directory to the current directory in MATLAB. In ModelSim/QuartaSim, enter the following command to compile your design

```
do build.do
```

- In ModelSim/QuartaSim, enter the following command to simulate your design

```
do sim.do
```

When simulation finishes, you can exam the difference between the output of each subsystem and the captured signal in ModelSim's waveform window.

Get Started with MATLAB Based SystemVerilog DPI Generation

This example shows you how to generate a SystemVerilog DPI component for a programmable square-wave generator written in MATLAB®, and export it to an HDL simulator.

For demonstrative purposes, this example uses Modelsim® 10.3c in 64-bit Windows® 7. However, this same procedure can be easily replicated for other systems and simulators.

Requirements and Prerequisites

Products required for this example:

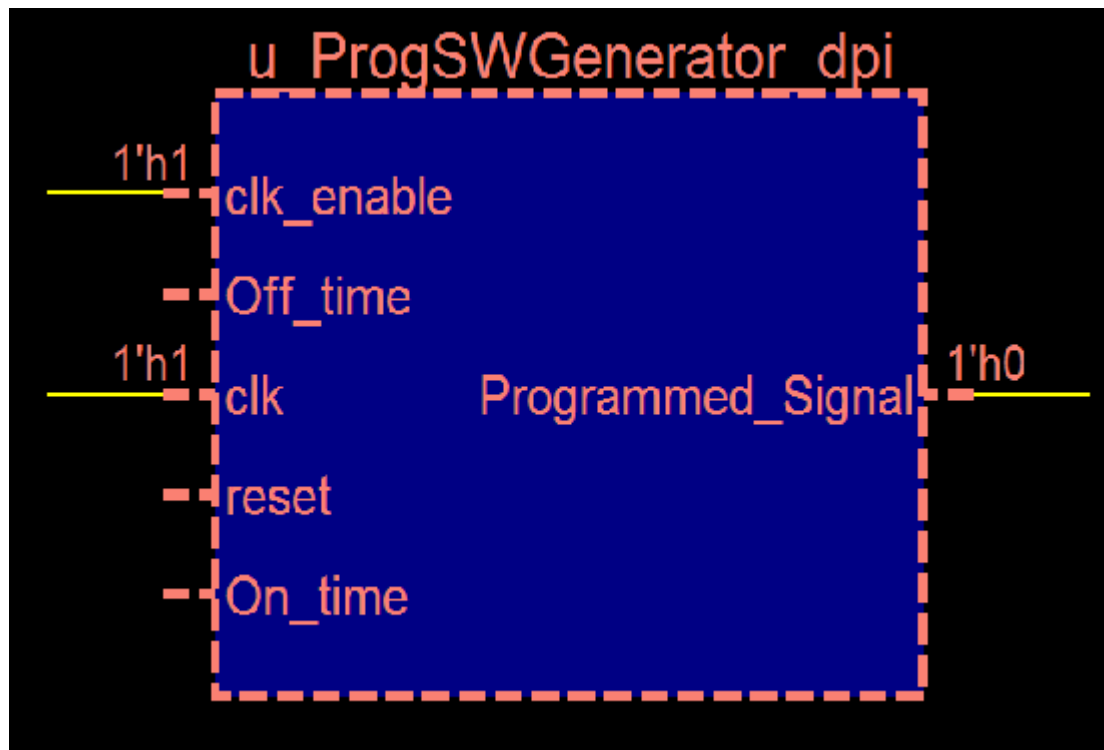
- MATLAB Coder™
- Simulators: Mentor Graphics® ModelSim®/QuestaSim® or Cadence® Xcelium™
- One of the supported C compilers: Microsoft® Visual C++, or GNU GCC

MATLAB Design

The MATLAB code used in this example demonstrates a simple programmable square wave generator. This example also provides a MATLAB test bench that exercises the design.

- Design: ProgSWGGenerator
- Test Bench: ProgSWGGenerator_tb

The following image illustrates the DPI component that is generated in this example:



- The 'On_time' and 'Off_time' control the output signal duty cycle.

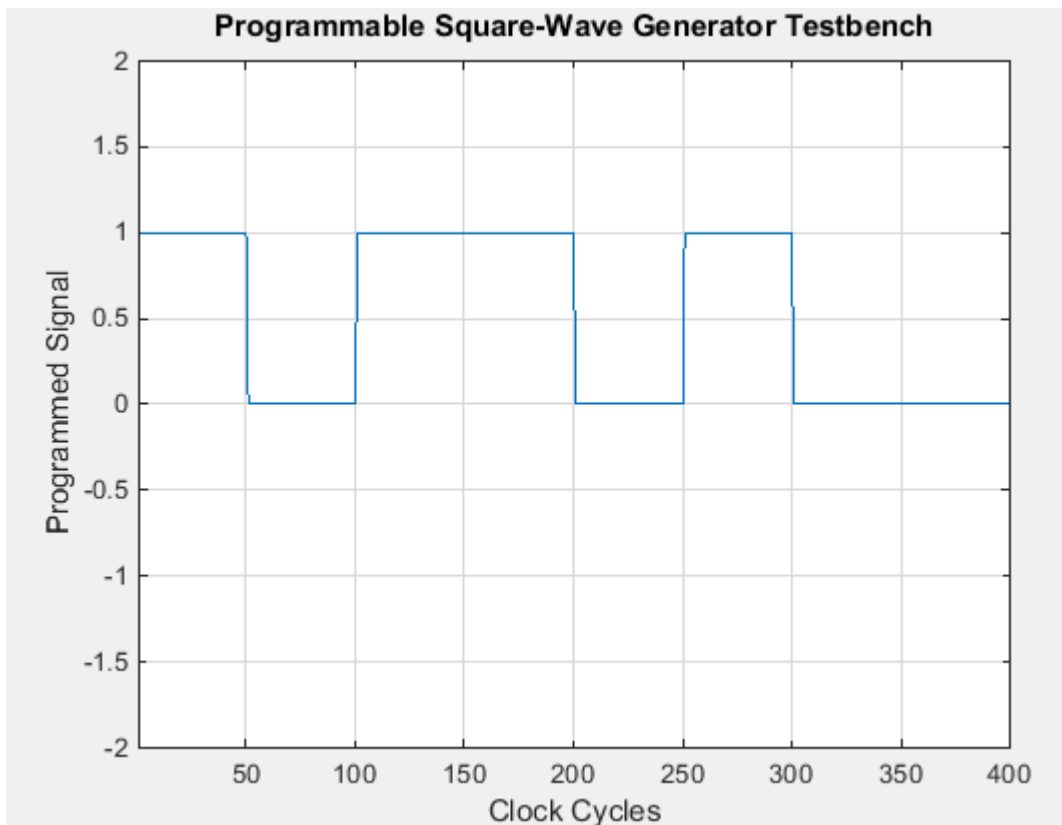
For example, if On_time=1 and Off_time=1, the square wave generated will have 50% duty cycle.

Simulate Design with Supplied MATLAB Testbench

To make sure there are no run-time errors, and that the design meets the requirements, simulate the design with the supplied testbench prior to code generation. Enter the following command in MATLAB:

```
ProgSWGGenerator_tb
```

It should plot the figure below:



Note that the plot shows the output signal that was programmed to behave as follows:

- Clock cycles<100: on=1,off=1
- 100<Clock cycles<250: on=2,off=1
- 250<Clock cycles: on=1,off=2

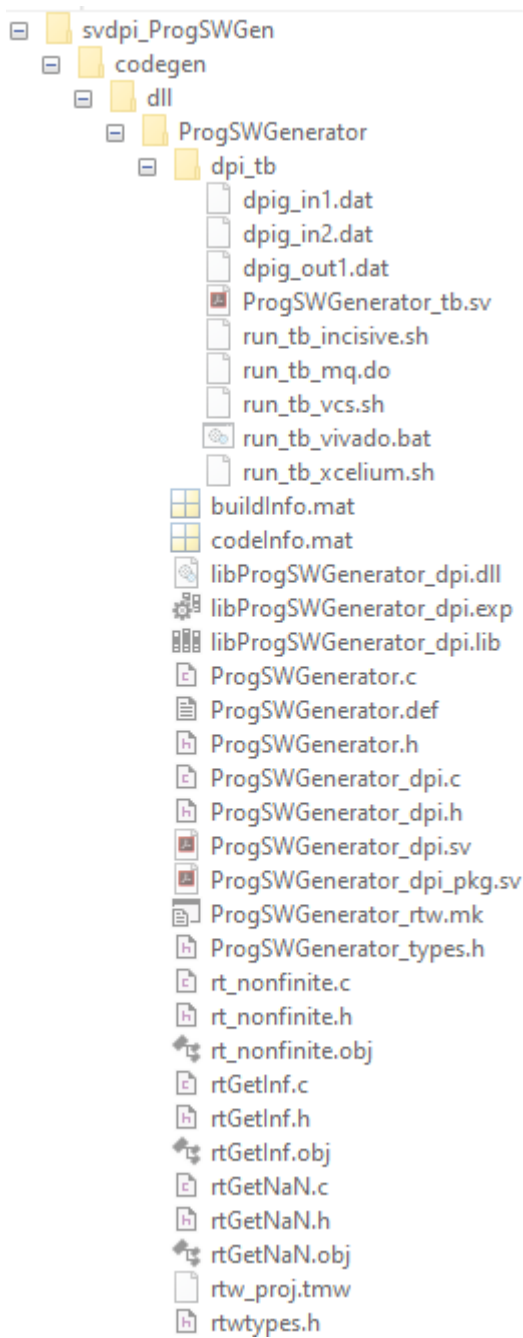
Generate the DPI component using DPIGEN command

Using DPIGEN, generate the DPI component.

Execute the DPIGEN command in MATLAB as follows:

```
dpigen -testbench ProgSWGGenerator_tb ProgSWGGenerator -args {0,0}
```

The following directory structure is generated:



dpi_tb: Folder where all the testbench related files are.

ProgSWGenerator_dpi.sv: Generated DPI component.

ProgSWGenerator_dpi_pkg.sv: Generated SystemVerilog package.

libProgSWGenerator_dpi.dll: Library containing the definitions of all imported functions.

Note: DPIGEN will automatically try to compile the library. In order to just generate the files without compiling you should use the -c option. For example:

```
dpigen -c -testbench ProgSWGenerator_tb ProgSWGenerator -args {0,0}
```

Run Generated Testbench in HDL Simulator

For ModelSim/Questasim, perform the following steps:

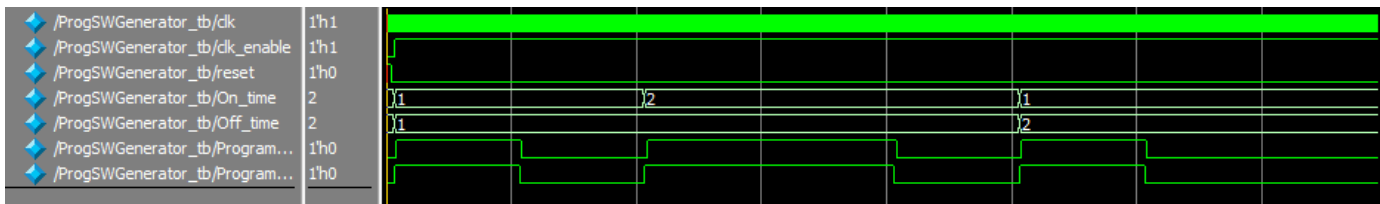
- Start ModelSim/Questasim in GUI mode.
- In the HDL Simulator, Change your current directory to "dpi_tb" under the code generation directory.
- Enter the following command to start your simulation

```
do run_tb_mq.do
```

For example:

```
Questasim> do run_tb_mq.do
```

The following wave form are generated:



Note that the output wave (the bottom one) behaves in exactly the same way as the signal the was plotted in the MATLAB testbench.

For Xcelium and VCS simulator:

- Start your terminal shell
- Change the current directory to "dpi_tb" under the code generation directory
- For Xcelium enter the following command in your shell.

```
sh run_tb_xcelium.sh
```

- For VCS enter the following command in your shell.

```
sh run_tb_vcs.sh
```

- When the simulation finishes, you should see the following text printed in your console:

```
*****TEST COMPLETED (PASSED)*****
```

This ends the Getting Started with MATLAB based SystemVerilog DPI Component Generation example.

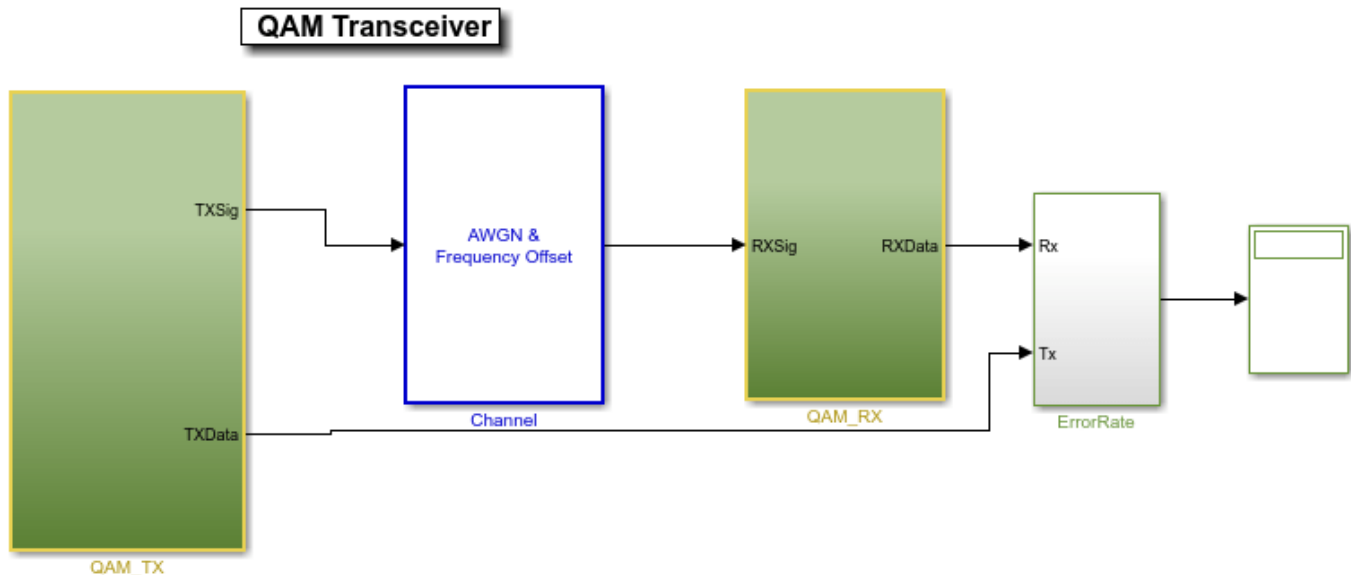
Building HDL Test Bench for QAM Transceiver Model

This example shows how to build a behavioral test bench using SystemVerilog DPI-C component generation. This test bench is used for the verification of synthesizable HDL code of a 64-QAM transmitter and receiver.

This example requires these additional tools. * One of the supported HDL simulators: Cadence® Xcelium™, or Mentor Graphics® ModelSim®/QuestaSim® * One of the supported C compiler: Microsoft® Visual Studio™ for Windows, or GNU GCC for Linux®

Overview

The top-level structure of the QAM receiver model is shown in the following figure. The QAM Tx HDL and QAM Rx HDL subsystems have been optimized for HDL code generation. Using HDL Coder, we can generate HDL code from those two subsystems. This example is shipped with generated HDL code, so you do not have to generate the code for this exercise.



Copyright 2014 The MathWorks, Inc.

Set Up Model for Code Generation

To build a complete behavioral testbench in HDL, we will need the behavioral models for the Channel subsystem, and for the ErrorRate subsystem. With these models, we can generate SystemVerilog DPI-C components for those two subsystems. Before generating DPI-C components, we need to set code generation options first.

Open Simulink Model Configuration Parameter panel from the `svdpi_qam` model. Set the following parameters:

Select Code Generation -> System Target File. Click Browse button and select "systemverilog_dpi_grt.tlc".

If you have Embedded Coder you can alternatively choose the 'systemverilog_dpi_ert' target file.

In Toolchain setting, select one of the Visual Studio versions if you are using Windows, or one of the GCC toolchains if you are using Linux.

Next, export the DPI-C components by executing the following two commands in MATLAB:

```
>> slbuild('svdpi_qam/Channel')
>> slbuild('svdpi_qam/ErrorRate')
```

Run Generated Test Bench

For Mentor Graphics ModelSim/QuartaSim,

- 1 Start ModelSim/QuartaSim in GUI mode.
- 2 Change your current directory to the current MATLAB directory
- 3 Enter the following command to start your simulation:

```
do QAM_DPIC_tb_mq.do
```

For Cadence Xcelium:

- Start your terminal shell.
- Change your current directory to the current MATLAB directory.
- Enter the following command in your shell.

```
sh QAM_DPIC_tb_xcelium.sh
```

At the end of the simulation, the error rate is printed as follows:

```
***** Simulation Summary *****
```

```
Bit error rate : 0.001356
```

```
Number of errors : 4.000000
```

```
Number of received bits : 2950.000000
```

```
*****
```

Generate FIFO Interface DPI Component for UART Receiver

This example shows the full workflow of how to generate a SystemVerilog DPI component for a FIFO buffer interface meant to be integrated with a UART receiver. The interface is written in MATLAB, and exported to an HDL simulator. The SystemVerilog files for the UART receiver and its test bench are also provided.

For demonstrative purposes, this example uses Modelsim 10.3c in 64-bit Windows 7. However, this same procedure can be easily replicated for other systems and simulators.

Requirements and Prerequisites

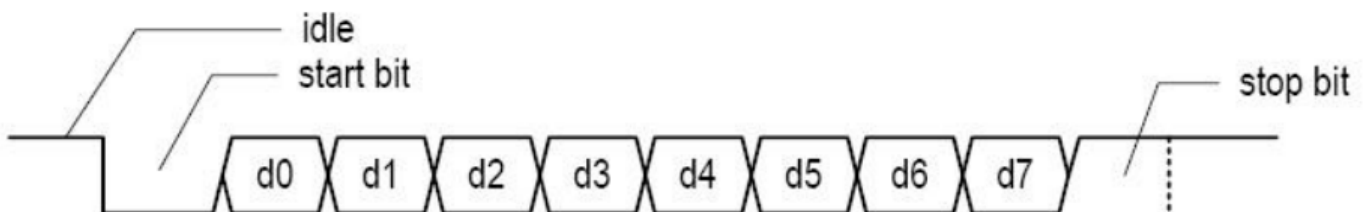
Products required for this example:

- MATLAB Coder®
- Simulators: Mentor Graphics® ModelSim®/QuestaSim® or Cadence® Xcelium™
- One of the supported C compiler: Microsoft® Visual C++, or GNU GCC

Background

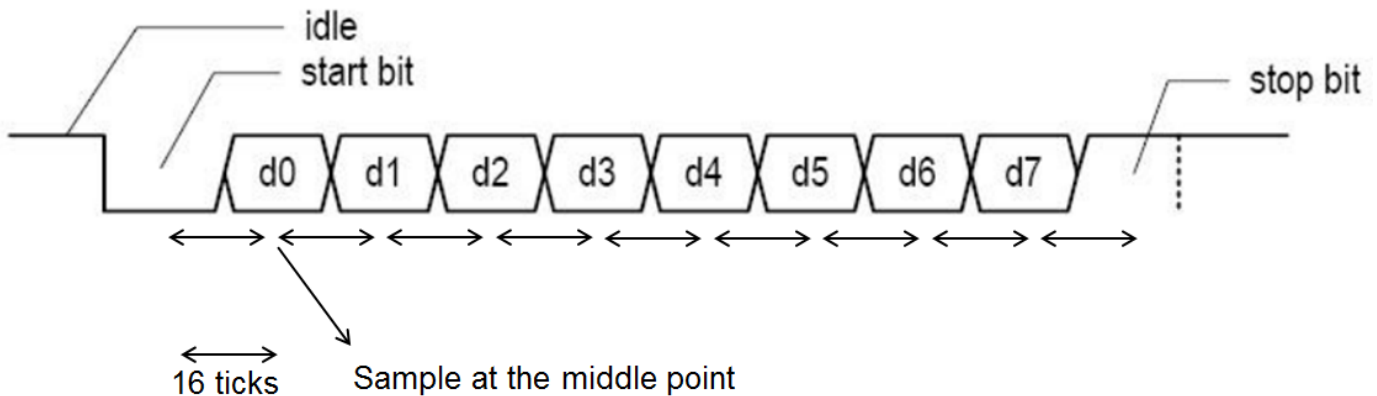
A universal asynchronous receiver and transmitter (UART) is a circuit that sends and receives data through a serial line. UART's are usually used with the RS-232 standard and contain a receiver and transmitter. However in this example only a receiver is used.

The serial transmission starts with a 'start bit' which is 0, followed by data bits, and ends with a 'parity bit' and a 'stop bit'. Transmission of a single byte is show in the following figure:

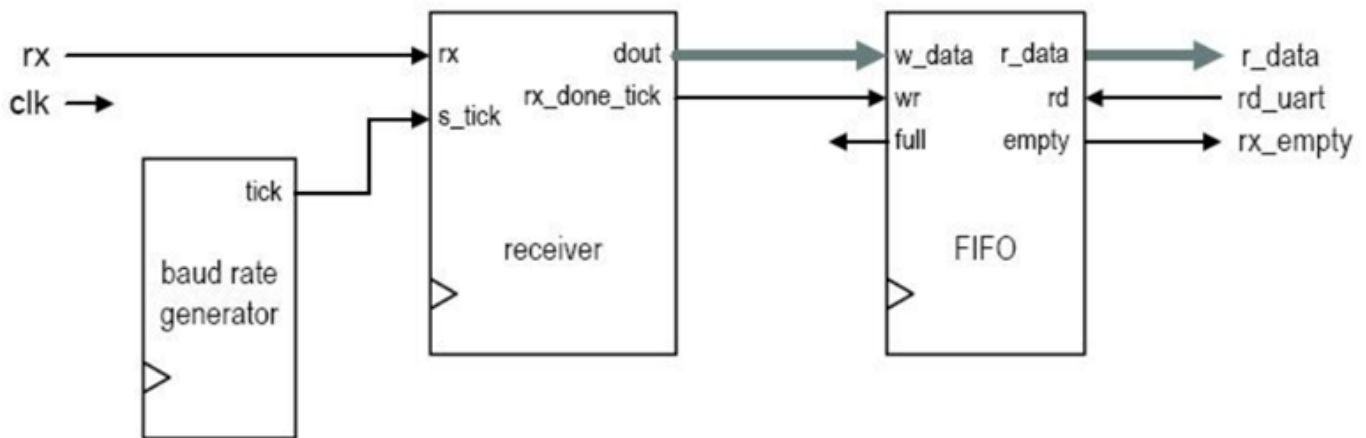


The transfer is asynchronous, which mean that there is no clock information, therefore the receiver and transmitter must agree on what baud rate, stop bits and parity bit are used. In this example one byte will be transferred with a baud rate of 19,200, 1 stop bit and no parity bits.

An oversampling scheme will be used to estimate the middle point of the data bit at a rate 16 times the baud rate, as shown in the figure below.



The following schematic illustrates the UART receiver design.



Step 1: MATLAB Design

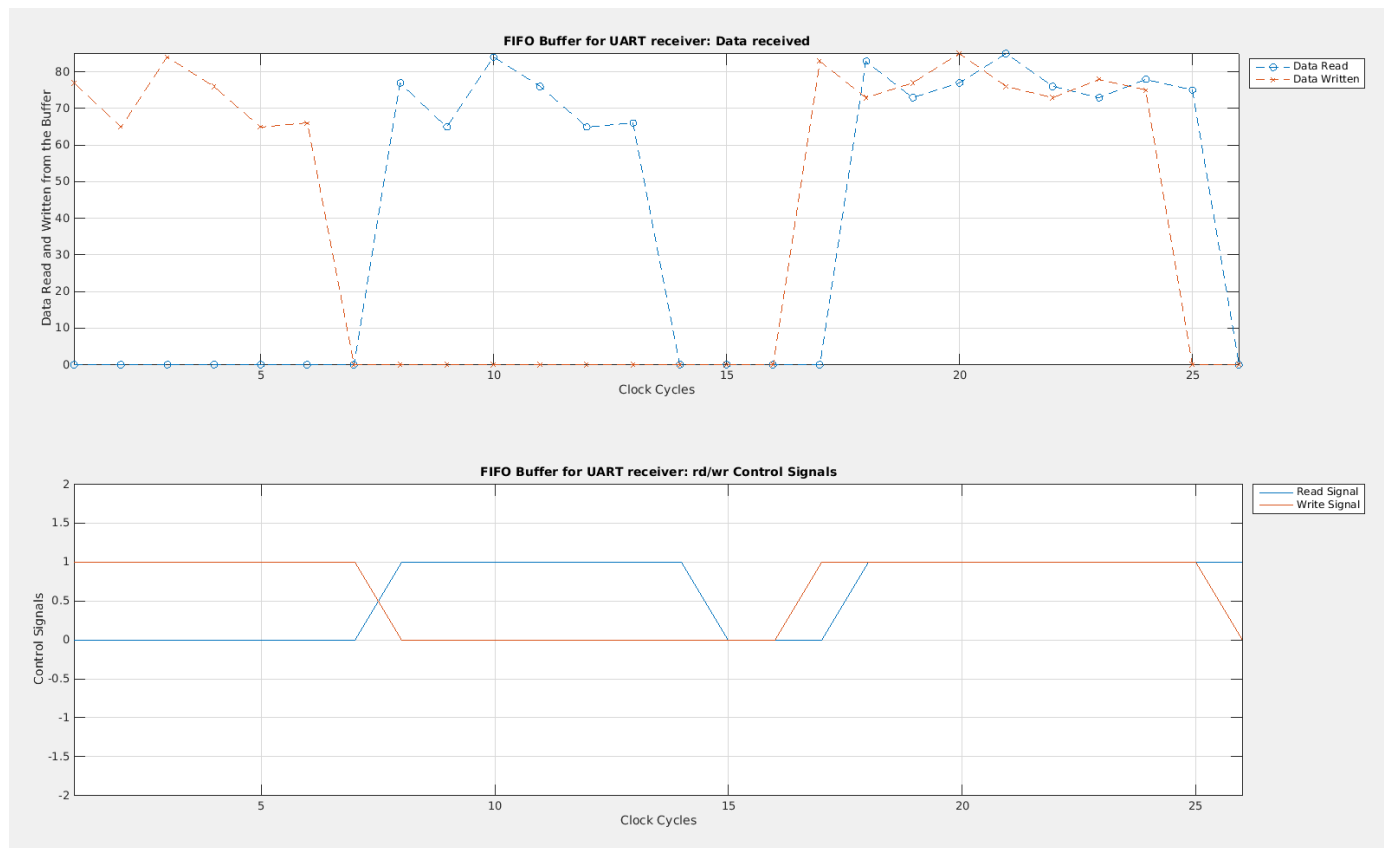
The first step is to write the MATLAB code that will satisfy the requirements of your design, you should try to capture the requirements in a testbench. In this example our design consists of a first in first out (FIFO) buffer of 8 words.

- 1 Design: FIFO_Buffer
- 2 Test Bench: FIFO_Buffer_tb

Step 2: Make sure the MATLAB testbench captures the requirements

Run the testbench to make sure there are no runtime errors, the figure below should be plotted.

FIFO_Buffer_tb



Note that the testbench is exercising the design in the following way:

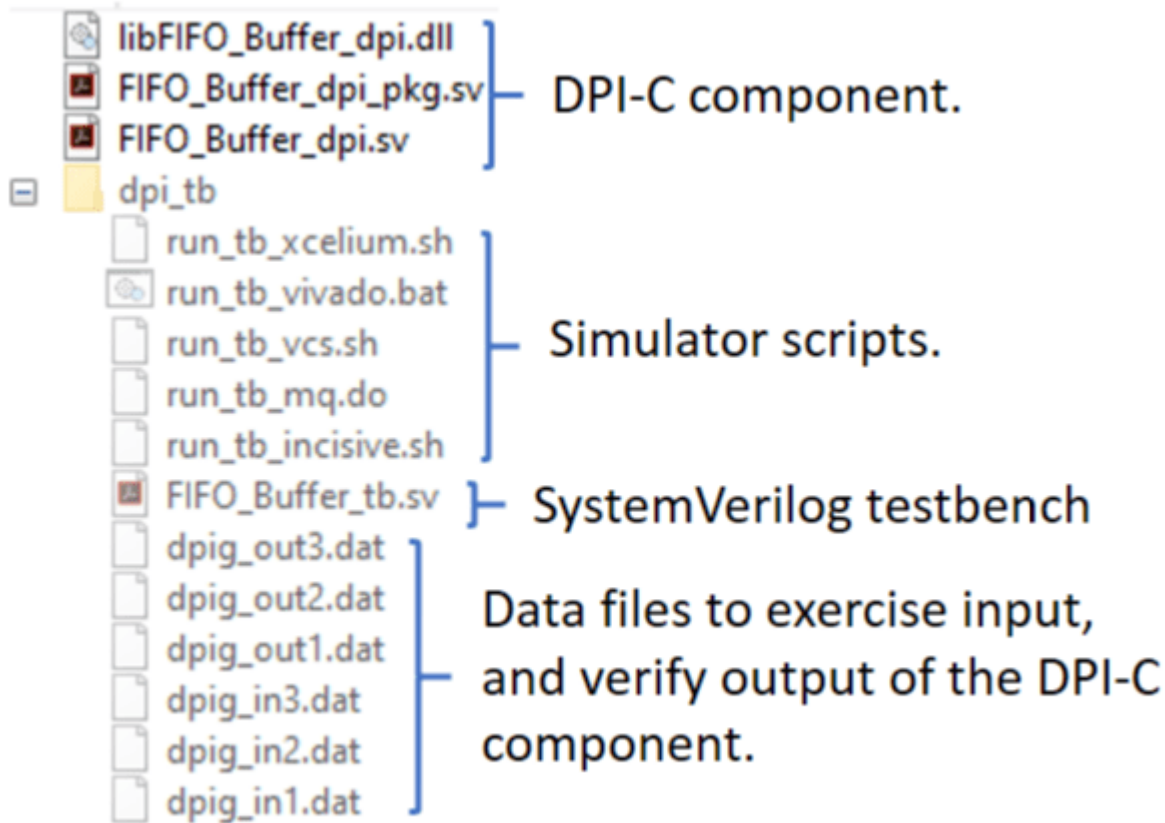
- Clock cycles < 15: Write 6 bytes (write signal enabled, read signal disabled), then Read the 6 bytes (read signal enabled, write signal disabled).
- Clock cycles > 15: Read and write simultaneously. (For example: The byte that is written is read in the next clock cycle).

Step 3: Generate the DPI component and verify the behavior in the HDL Simulator

To generate the component execute the following command:

```
dpigen -testbench FIFO_Buffer_tb FIFO_Buffer -args {0,int8(0),0}
```

The figure below shows the relevant files for this example.

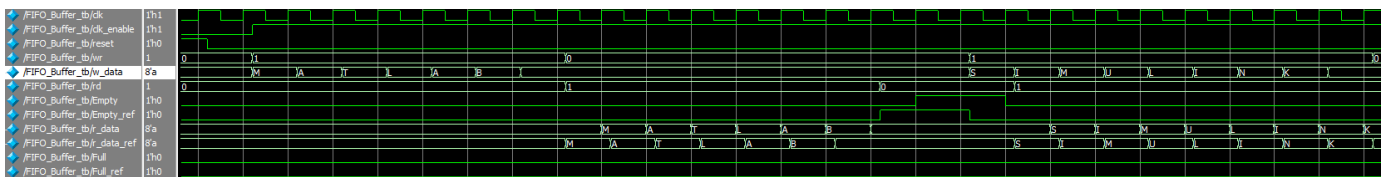


Once DPIGEN generates the DPI component and its testbench you can run the SystemVerilog testbench by following the steps below:

- Start ModelSim/Questasim in GUI mode.
- Change your current directory to codegen/dll/FIFO_Buffer/dpi_tb under the code generation directory in your HDL simulator.
- Enter the following command to start your simulation

```
do run_tb_mq.do
```

The following wave forms will be generated:



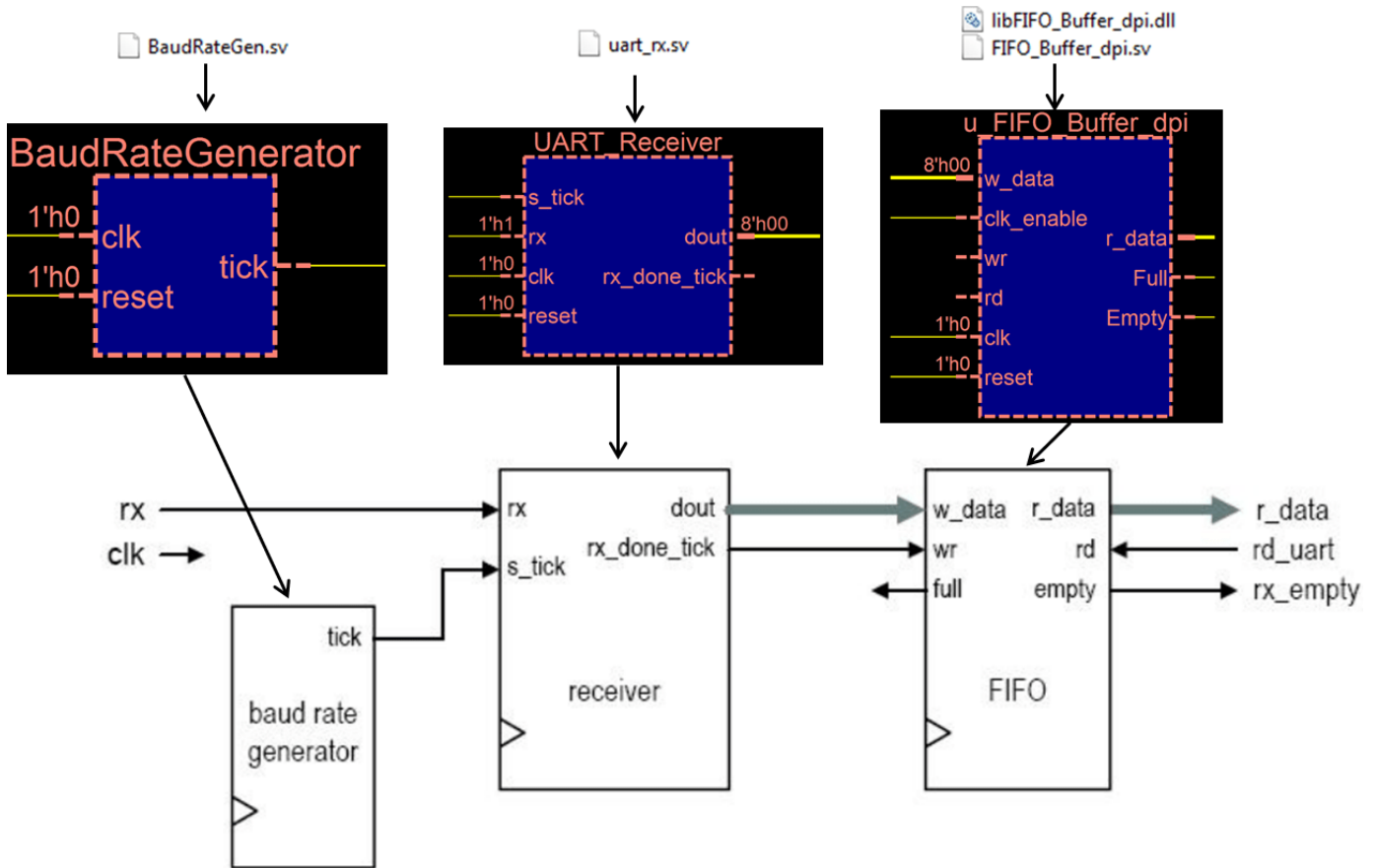
Note that this matches our MATLAB testbench where we write 6 bytes and then read them, followed by simultaneously writing and reading 8 bytes.

- When the simulation finishes, you should see the following text printed in your console:

*****TEST COMPLETED (PASSED)*****

Step 4: Integrate the Generated DPI Component into the UART Receiver design

After the DPI component behavior has been verified, it is time to integrate it into the UART receiver. The following figure shows the files required for the different components.

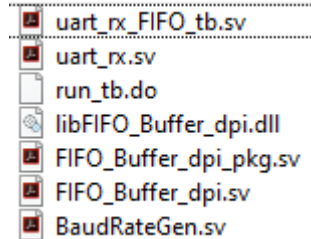


To exercise the UART receiver with the DPI component integrated, the testbench named 'uart_rx_FIFO_tb.sv' has been provided. The UART component is instantiated in the testbench as shown below:

```
// Instantiate DUT generated using MATLAB Based SystemVerilog DPI
FIFO_Buffer_dpi u_FIFO_Buffer_dpi(
    .clk(clk),
    .clk_enable(clk_enable),
    .reset(reset),
    .wr(rx_done_tick),
    .w_data(dout),
    .rd(rd),
    .Empty(Empty),
    .r_data(r_data),
    .Full(Full)
);
```

Step 5: Simulate UART Receiver

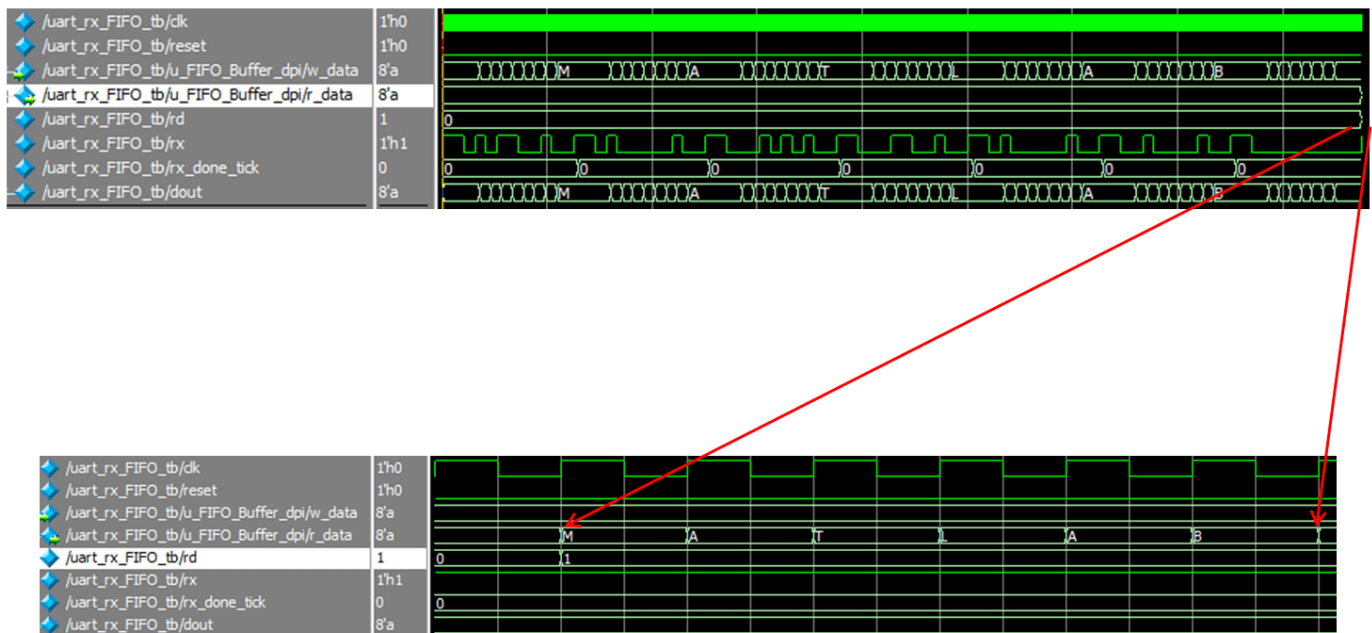
To simulate the design successfully, make sure the following files are in the same folder:



Similar to step 3, open Modelsim and run the .do file using:

do `run_tb.do`

The following wave forms are generated:



In the testbench 6 ASCII letters ('MATLAB') are transferred via the serial 'rx' signal, and written into the FIFO buffer. After the 6th letter transfer is complete the 'rd' signal is asserted to get the letters ('MATLAB') back in 'r_data'.

References

Pong P. Chu (2008), FPGA Prototyping by Verilog Examples. Hoboken, New Jersey: John Wiley & Sons, Inc.

Generate Bit Vector and Logic Vector Data Types

This example shows you how to generate bit or logic vector data types in the SystemVerilog interface of the DPI component. This capability is useful whenever having an exact width of the port is important to integrate the DPI component in your testbench.

Requirements and Prerequisites

Products required for this example:

- MATLAB®
- Simulink®
- Simulink Coder™
- Fixed-Point Designer™
- Mentor Graphics® ModelSim®/QuestaSim®
- One of the supported C compilers: Microsoft® Visual C++, or GNU GCC

Overview

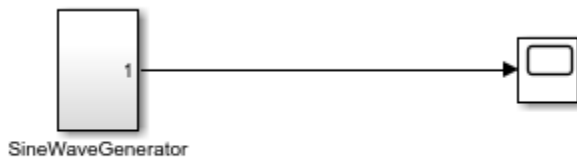
If the Simulink model has a port with fixed-point data type, by default, the generated SystemVerilog code maps the port to a compatible C type that can hold the necessary bits. For more information about these mappings, see "Supported Simulink Data Types" in the documentation.

This Simulink model generates a discrete sine wave using a signed fixed-point data type containing 20 bits. By default the generated SystemVerilog code uses 'int' data type to represent this port. This example shows how to generate SystemVerilog code that uses bit or logic vectors to represent the exact length of the fixed-point number.

Open Example

Run the following code to open the design.

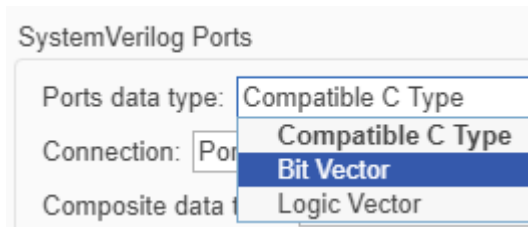
```
open_system('svdpi_BitVector');
```



Copyright 2017 The MathWorks, Inc.

Set Up Model for Code Generation

The model is preconfigured with one of the DPI-C system target files (`systemverilog_dpi_grt.tlc`). To generate SystemVerilog code with bit or logic vectors, open the configuration parameters. Then, in **Code Generation > System Verilog DPI**, select Bit Vector from the **Ports data type** menu.



Alternatively, you can set this parameter by executing:

```
set_param('svdpi_BitVector', 'DPIFixedPointDataType', 'BitVector');
```

Generate SystemVerilog DPI Component

- 1 In the "svdpi_BitVector" model, right click the SineWaveGenerator block, and select C/C++ Code -> Build This Subsystem.
- 2 Click Build in the dialog box that appears.
- 3 The build generates C code for the SineWaveGenerator subsystem, a System Verilog DPI wrapper "SineWaveGenerator_build/SineWaveGenerator_dpi.sv", and a package file "SineWaveGenerator_build/SineWaveGenerator_dpi_pkg.sv".

Alternatively you can generate the component by executing:

```
slbuild('svdpi_BitVector/SineWaveGenerator');
```

Inspect the SystemVerilog Code

Open the DPI component "SineWaveGenerator_dpi.sv" and note the data type at the interface:

```
module SineWaveGenerator_dpi(
    input clk,
    input clk_enable,
    input reset,

    /* Simulink signal name: 'Out1' */
    output bit signed [19:0] SineWaveGenerator_Y_Out1
);
```

If you select "Logic Vector" then the SystemVerilog DPI-C component uses the SystemVerilog "logic" type for the interface.

If you choose the default "Compatible C Types", then the generated SystemVerilog data type is "int" as shown:

```
module SineWaveGenerator_dpi(
    input clk,
    input clk_enable,
    input reset,

    /* Simulink signal name: 'Out1' */
    output int SineWaveGenerator_Y_Out1
);
```

Considerations for Port Types

- "Compatible C Types" supports ports up to 64 bits wide. If your design contains ports with word length greater than 64, then you must select "Bit Vector" or "Logic Vector".
- If your design contains a vector of fixed-point numbers and you select "Bit Vector" or "Logic Vector" then the resulting data type is a mixed (unpacked and packed) SystemVerilog array. For example a vector of size 3 containing fixed-point numbers of word length 2 would map to bit [1:0] PortName [3].

Generate Native SystemVerilog Assertions from Simulink

This example shows you how to generate native SystemVerilog assertions from assertions in a Simulink® model. This capability is useful whenever you need the same assertion behavior in Simulink and in your HDL testing environment.

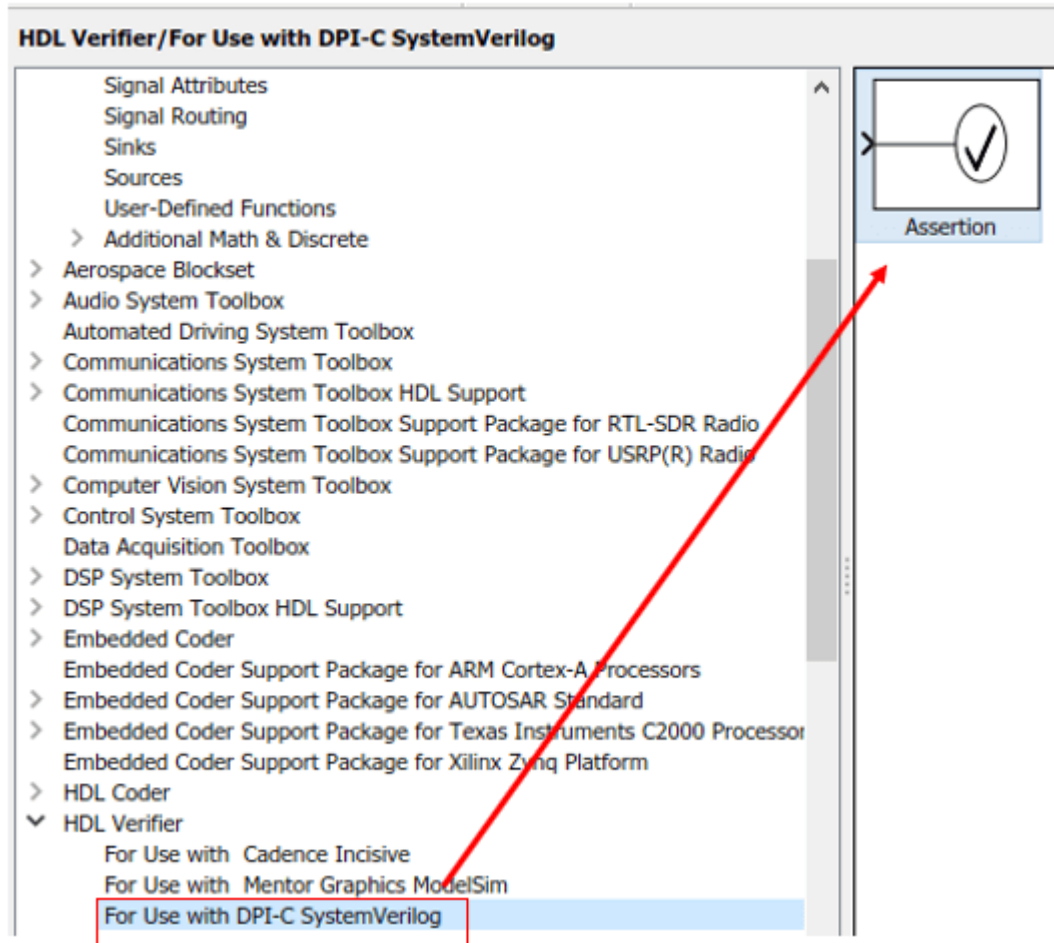
Requirements and Prerequisites

Products required for this example:

- MATLAB®
- Simulink®
- Simulink Coder™
- Mentor Graphics® ModelSim®/QuestaSim® or supported SystemVerilog simulator
- One of the supported C compilers: Microsoft® Visual C++, or GNU GCC

Overview

To generate a DPI-C component that contains SystemVerilog assertions, the Simulink model must use the **Assertion for DPI-C** block. Find this block in the **HDL Verifier-> For Use with DPI-C SystemVerilog** library as shown below:



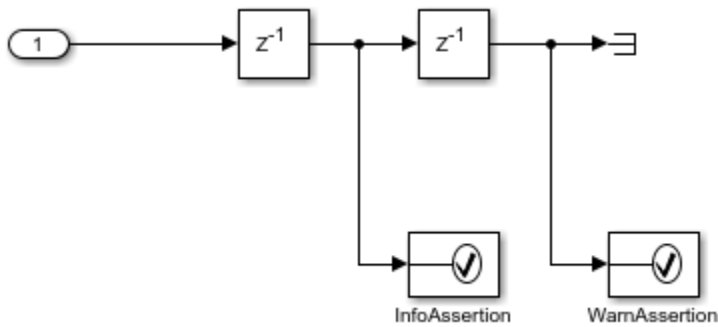
This block can be used just like the generic **Assertion** block in the **Model Verification** library. The simulation behavior is identical to the Simulink Assertion block, however during DPI-C component generation the block generates a native SystemVerilog assertion for each **Assertion for DPI-C** block present in the model.

Set Up Example

The model in this example contains two **Assertion for DPI-C** block. One of them is used to provide information for a delay using the customization option, the second block is set to provide warnings when a second delay is over.

Run the following code to open the design.

```
open_system('svdpi_assertion');
```

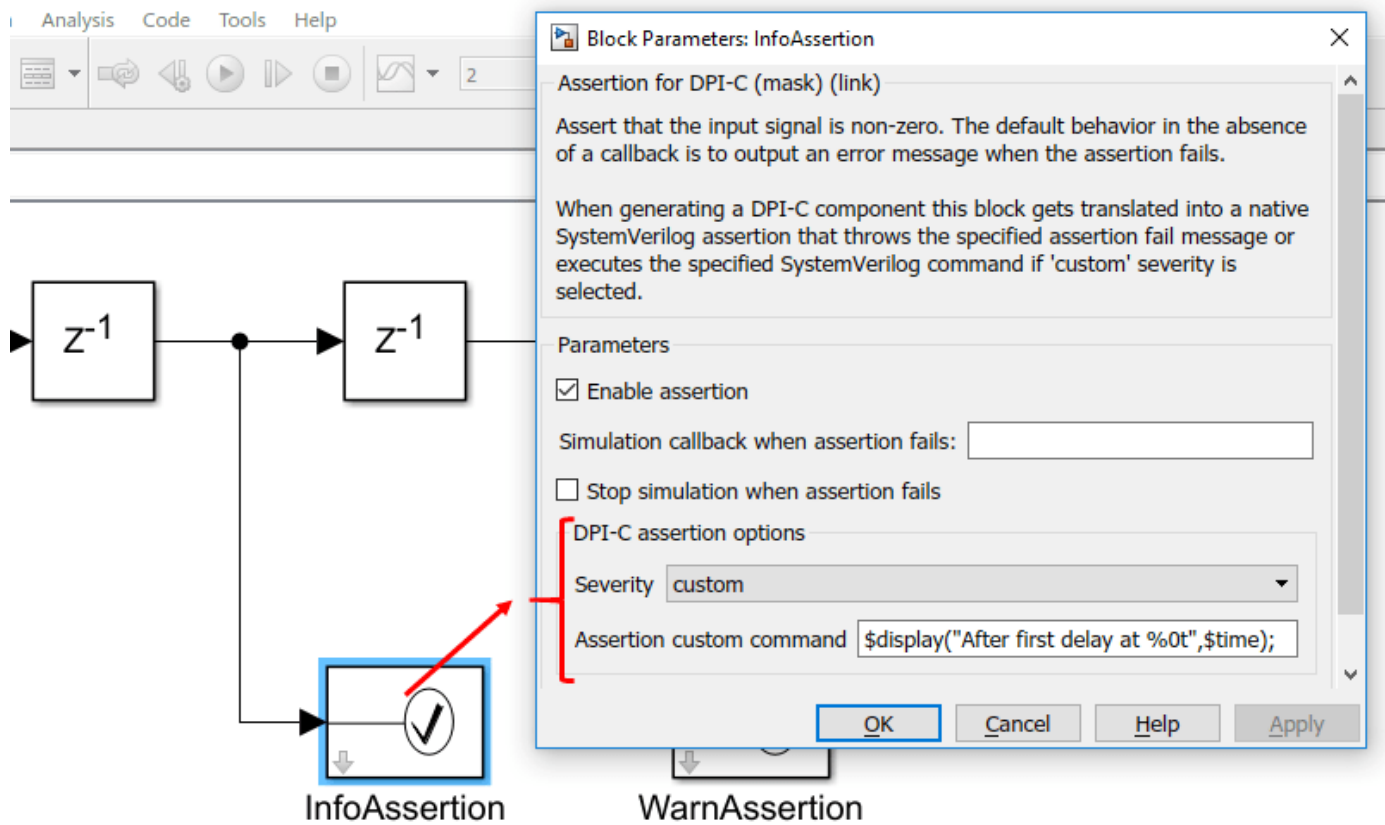


Set Up Model for Code Generation

The model is preconfigured with one of the DPI-C system target files (`systemverilog_dpi_grt.tlc`). Before generating the DPI-C component make sure to configure the desired assertion behavior in SystemVerilog via the block mask.

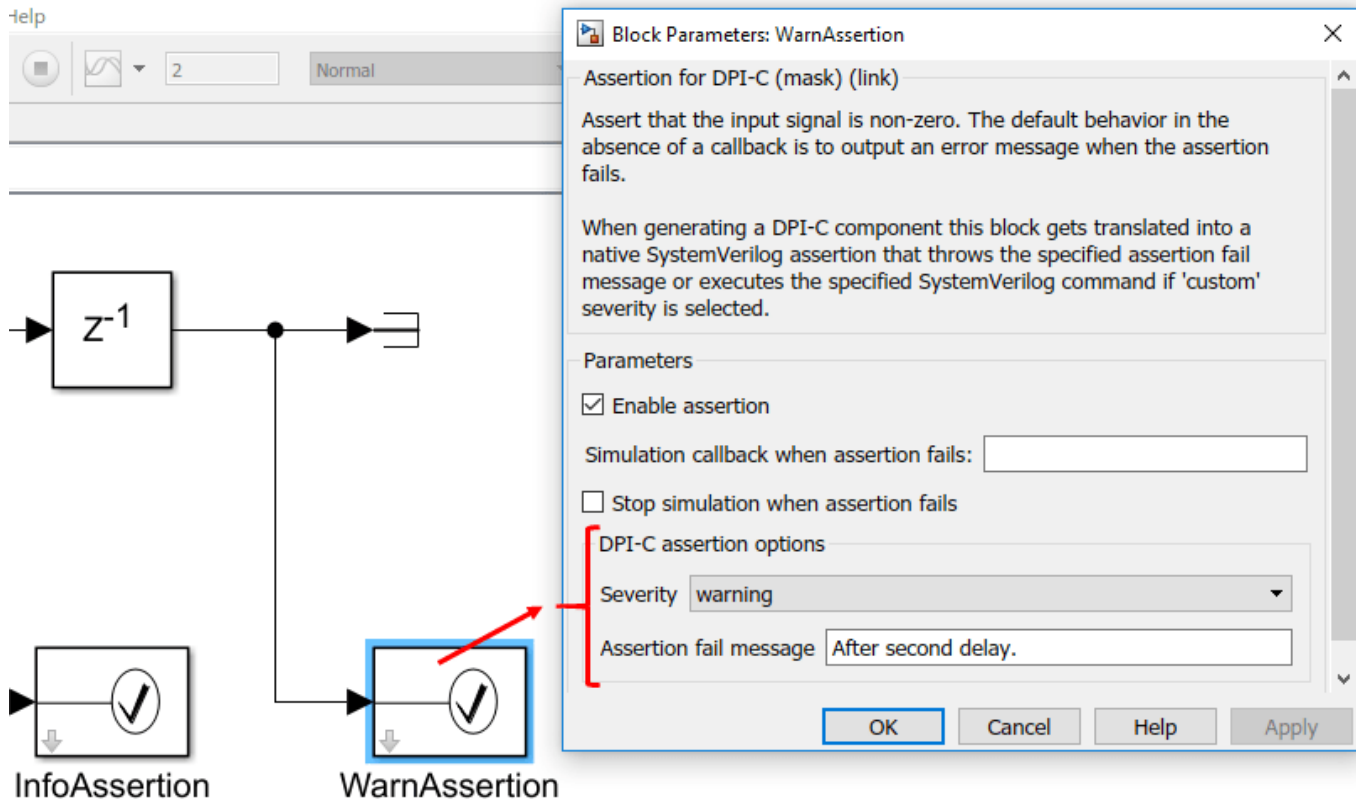
By default the assertion block will generate a SystemVerilog error (`$error("")`) with an empty message when triggered. This can be changed either by providing an error message, changing the error to a warning (`$warning("")`) with a warning message, or providing your own custom SystemVerilog command to be executed.

The picture below shows the SystemVerilog behavior configuration for the assertion block after the first delay. The parameters outside of the **DPI-C assertion options** group only affect Simulink simulation behavior.



Notice that the **Assertion custom command** is a valid SystemVerilog statement.

The second assertion block is configured to output a warning with a specified message as shown below.



If we run the model we can see that two assertion warnings are triggered in Simulink:

Simulation 3

03:45 PM Elapsed: 5 sec

```

Assertion detected in 'svdpi\_assertion/DPI C Assertion/InfoAssertion/Assertion' at time 1
Component: Simulink | Category: Block warning
Assertion detected in 'svdpi\_assertion/DPI C Assertion/InfoAssertion/Assertion' at time 2
Component: Simulink | Category: Block warning
Assertion detected in 'svdpi\_assertion/DPI C Assertion/WarnAssertion/Assertion' at time 2
Component: Simulink | Category: Block warning

```

Generate SystemVerilog DPI-C Component

- 1 In the **svdpi_assertion** model, right click the **DPI_C_Assertion** block, and select C/C++ Code -> Build This Subsystem.
- 2 Click Build in the dialog box that appears.
- 3 The build generates C code for the **DPI_C_Assertion** subsystem, and a System Verilog DPI-C wrapper and package file named "**DPI_C_Assertion_build/DPI_C_Assertion_dpi.sv**" and "**DPI_C_Assertion_build/DPI_C_Assertion_dpi_pkg.sv**".

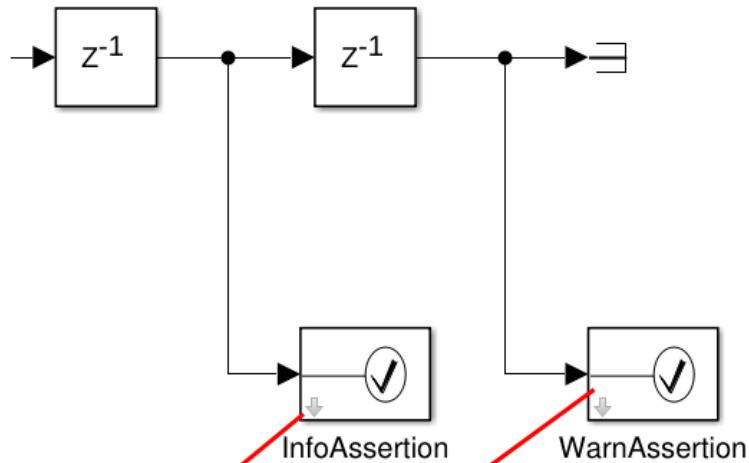
Alternatively you can generate the component by executing:

```
slbuild('svdpi_assertion/DPI_C_Assertion');
```

Run Generated Testbench in HDL Simulator

For this example ModelsSim/Questasim simulator will be used. To get more detailed instructions on how to run the testbench please refer to "Getting Started with SystemVerilog DPI Component Generation".

After running the testbench notice the messages and the warnings that are being thrown by the DPI-C component.



```
# After first delay at 50
# After first delay at 60
# ** Warning: svdpi_assertion:7:After second delay.
#   Time: 60 ns  Scope: DPI_C_Assertion_dpi_tb.u_DPI_C_Assertion_dpi File: ../DPI_C_Assertion_dpi.sv Line: 52
# *****TEST COMPLETED (PASSED)*****
```

Tracing a SystemVerilog Assertion Back to Simulink

If you want to trace the assertion that generated the warning back to Simulink you need to find the Simulink Identifier (SID) from the warning message as shown below:

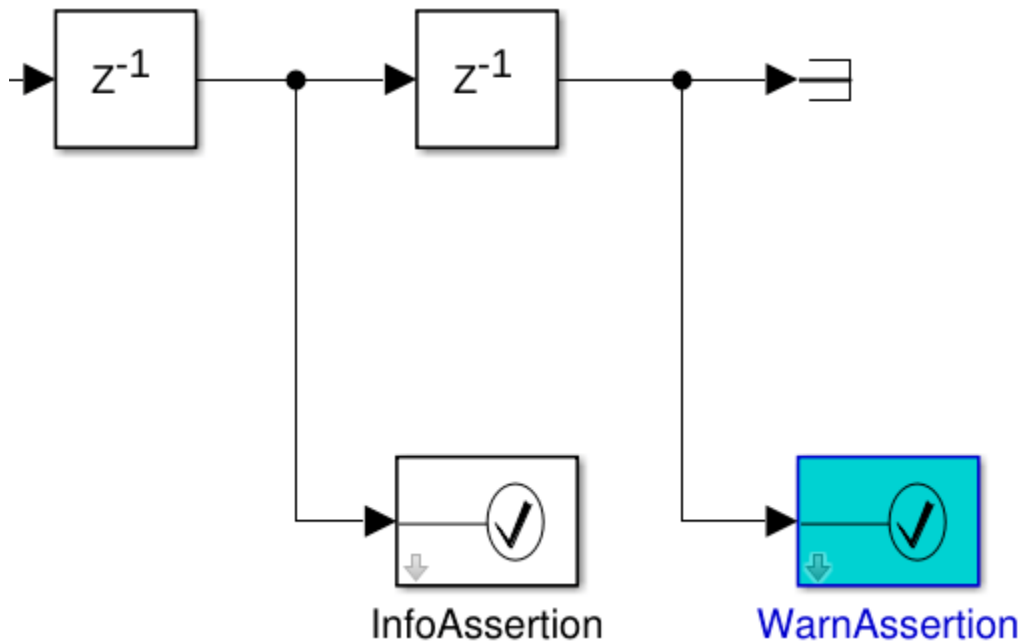
Simulink assertion block SID

```
# After first delay at 50
# After first delay at 60
# ** Warning: svdpi_assertion:7:After second delay.
#   Time: 60 ns  Scope: DPI_C_Assertion_dpi_tb.u_DPI_
# *****TEST COMPLETED (PASSED)*****
```

Once you find the SID for the assertion block you can use Simulink programmatic API's to highlight the corresponding block. Execute:

```
Simulink.ID.hilite('svdpi_assertion:7');
```

This will highlight the relevant block.



Filtering an Assertion in the HDL Simulator

If you want to filter an assertion in the HDL Simulator, you need to supply the SID of the block you want to filter as a plusargs argument to the HDL Simulator.

For instance the SID of the "InfoAssertion" assertion block is "svdpi_assertion:6". So in order to filter the informative messages given by this block we need to supply the argument "+svdpi_assertion:6" to the HDL Simulator. For this example in ModelSim/Questasim the simulation command would be:

```
vsim -c -voptargs=+acc -sv_lib ../DPI_C_Assertion work.DPI_C_Assertion_dpi_tb +svdpi_assertion:6
```

Generating Functional Coverage in SystemVerilog from Simulink Test verify Calls

This example demonstrates how to test a projector control system using model simulation, and how to generate a SystemVerilog DPI component for some of the controller's high level requirements that are specified in a Test Sequence block. This will allow the requirement verification used for model simulation to be reused in the HDL simulator with minimal effort.

The model was taken from the example "Projector Controller Testing Using verify and Real-Time Tests" (Simulink Test) shipped with Simulink Test™, and simplified to show only requirement scenario 4.

To learn more about verify statements, see "Assess Model Simulation Using verify Statements" (Simulink Test).

Other Prerequisites

In addition to the stated product requirements, this example requires:

- A supported HDL simulator. See "UVM and DPI Component Generation Requirements".
- A supported C compiler. See "Select and Configure C or C++ Compiler" (Simulink Coder).

Overview

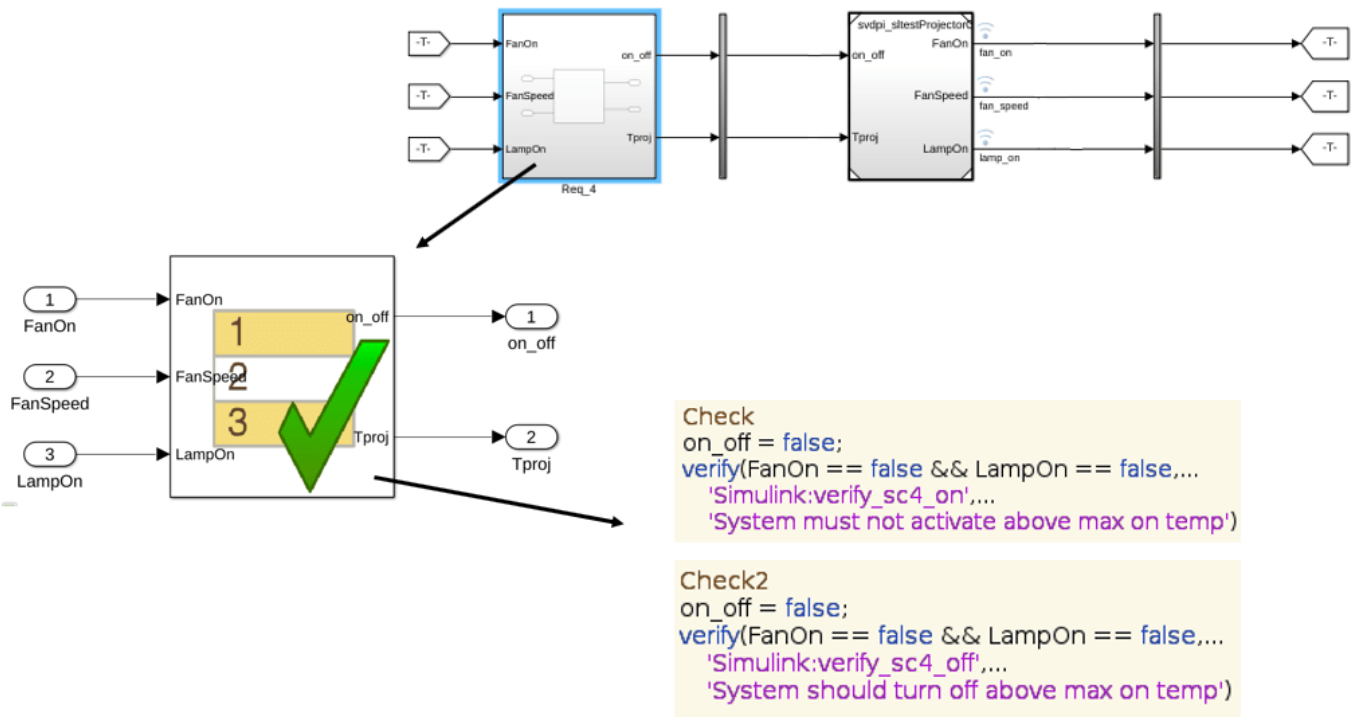
The test verifies the controller against its requirements using test sequences that exercise the top-level controller model. The controller uses a push button input and a temperature sensor input, and outputs signals controlling the fan, fan speed, and projector lamp.

The objective is to generate a SystemVerilog DPI component that captures high level requirement number 4 of the controller. For more information about the requirements refer to the word document `sltestProjectorCtrlReqs.docx` in the example referred above.

Requirement 4 tries to turn on and off the projector when the projector temperature (Tproj) is high. The scenario has the following steps in the Test Sequence block:

- 1 Set projector temperature to 50 degree Celsius.
- 2 Try to turn on.
- 3 The system should not turn on.
- 4 Set the temperature to 50 degree Celsius.
- 5 Try to turn off.
- 6 The system should turn off.

The picture below shows the test bench for the above requirement and how `verify` is used to check that the projector turns on or off depending on the scenario.



Set Up Model for Code Generation

The model and test bench are preconfigured with one of the SystemVerilog DPI system target files (systemverilog_dpi_grt.tlc). Open the test harness Req_scenario_4 by executing:

```
testFile = 'svdpi_sltestProjectorCtrlTests.mldatx';
testHarness = 'Req_scenario_4';
model = 'svdpi_sltestProjectorController';
open_system(model)

sltest.harness.open(model, testHarness)
```

Generate SystemVerilog DPI Component

- 1 In the **Req_scenario_4** test bench, right click the **Req_4** subsystem block which contains the test sequence block and select. **C/C++ Code -> Build This Subsystem**.
- 2 Click **Build** in the dialog box that appears.
- 3 The build generates C code for the **Req_4** subsystem, and a SystemVerilog DPI wrapper and package file named "Req_4_build/Req_4_dpi.sv" and "Req_4_build/Req_4_dpi_pkg.sv".

Note that some verification warnings will be triggered, this will be explained later.

Alternatively you can generate the component by executing:

```
slbuild('Req_scenario_4/Req_4');
```

Run Generated Testbench in HDL Simulator

For this example ModelsSim/Questasim simulator will be used. For detailed instructions on how to run the testbench refer to “Get Started with SystemVerilog DPI Component Generation” on page 32-68.

```
cd Req_4_build/dpi_tb
! vsim -c -do run_tb_mq.do # Run ModelSim/Questasim in console mode
cd ../..
```

Examine the HDL simulation output and notice the following:

- An info message shows that functional coverage will be gathered for 2 verify calls in the component
- There is an error flagged regarding a failure of the controller to shut off when the temperature is above a limit.
- The test is marked as PASSED because the SystemVerilog simulation results match the Simulink simulation results.
- The functional coverage shows that coverage is achieved for the first verify call but is not achieved for the second.
- The overall functional coverage goal is not met.

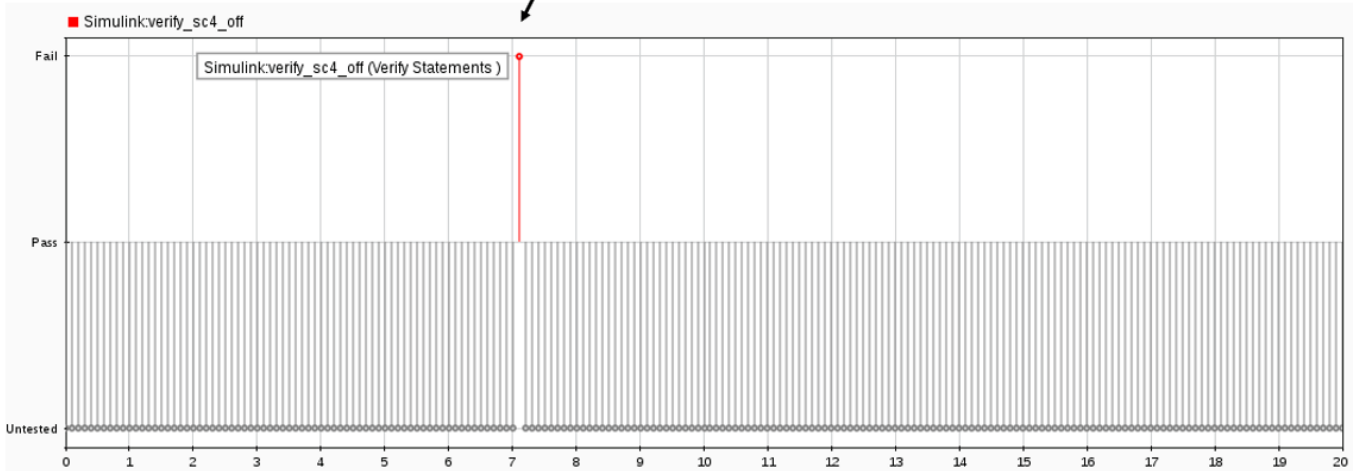
```
# ** Info: Gathering coverage for          2 Simulink verify() calls.
# Time: 0 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.initVerifyInfo File: ../Req_4_dpi_pkg.sv Line: 159
# ** Error: Req_scenario_4:32:60: At step 'Check2' verify id 'Simulink:verify_sc4_off' Failed
# Time: 750 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 237
# *****TEST COMPLETED (PASSED)*****
# ** Note: $finish      : ./Req_4_dpi_tb.sv(98)
# Time: 2042 ns Iteration: 0 Instance: /Req_4_dpi_tb
# ** Info: Instance coverage for verify 'Req_scenario_4:32:39', coverpoint 'pass_cp': metric=100.00, at least= 1 ( COVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 259
# ** Info: Instance coverage for verify 'Req_scenario_4:32:60', coverpoint 'pass_cp': metric= 0.00, at least= 1 ( UNCOVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 259
# ** Info: Overall coverage for Req_4_dpi_verify_calls: metric= 50.00 ( UNCOVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 266
# End time: 12:19:50 on Jun 08,2020, Elapsed time: 0:00:00
# Errors: 1, Warnings: 6
```

The error is consistent with the simulation results in Simulink (below). Opening the Test Manager reveals that the controller fails to shut off when the on_off button is pressed when the temperature is above a limit. To open test manager you can execute:

```
sltest.testmanager.load(testFile);
sltest.testmanager.view;
```

Resolving the failure would require to modify the OnOff check subsystem in the main model. The other requirement, `verify_sc4_on`, is satisfied, as shown in both the Simulink Test and SystemVerilog coverage results.

Test verification failed at t = 7.1 : System should turn off above max on temp.
 Step [Check2](#) in Test Sequence Block '[Req_scenario_4/Req_4/Test Sequence](#)': `verify(Fan0n == false && Lamp0n == false,...`
 Component: Simulink Test | Category: Runtime warning



Tracing a SystemVerilog Error Back to Simulink

If you want to trace the verify statement that generated the error back to Simulink you need to find the Simulink Identifier (SID) from the error message as shown below:

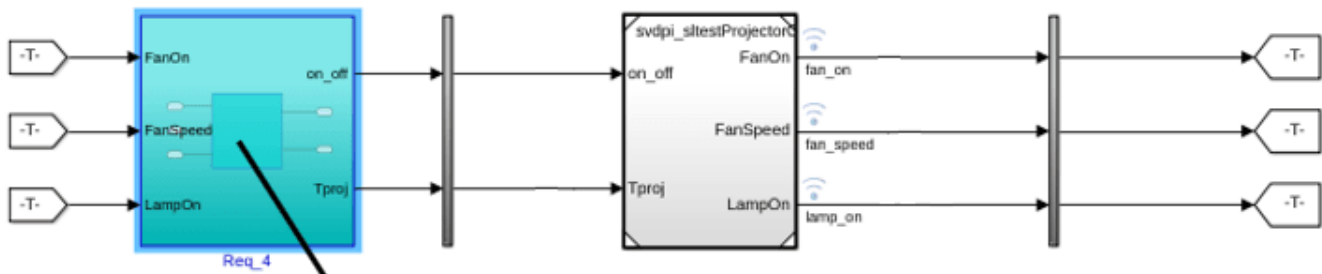
Simulink Identifier (SID)

```
# ** Error: Req_scenario_4:32:60: At step 'Check2' verify id 'Simulink:verify_sc4_off' Failed
#   Time: 750 ns   Scope: Req_4_dpi_tb.u_Req_4_dpi File: ../Req_4_dpi.sv Line: 77
# *****TEST COMPLETED (PASSED)*****
# ** Note: $finish      : ./Req_4_dpi_tb.sv(89)
#   Time: 2042 ns  Iteration: 0  Instance: /Req_4_dpi_tb
```

Once you find the SID for the step id in the test sequence block you can use a function to highlight the corresponding block. Execute the following command:

```
Simulink.ID.hilite('Req_scenario_4:32:60');
```

This will highlight the relevant block as shown below.



```

Check2
on_off = false;
verify(FanOn == false && LampOn == false,...
'Simulink:verify_sc4_off',...
'System should turn off above max on temp')

```

Filter a Specific Verify Assessment

To filter any verify assessment status checks in the HDL simulator, supply the SID of the assessment you want to filter as a plusargs argument to the HDL simulator. Such a filter will mean no errors and no coverage will be checked for that assessment. For instance you can filter the error that `verify_sc4_off` gives by supplying the argument `"+Req_scenario_4:32:60"` to the HDL simulator. You can do this through an environment variable so you don't have to modify the generated script.

```

% Clear environment variables that influence the SV simulation
setenv EXTRA_SVDPI_COMP_ARGS
setenv EXTRA_SVDPI_SIM_ARGS

% Filter the failing verify()
setenv EXTRA_SVDPI_SIM_ARGS "+Req_scenario_4:32:60=-1"
cd Req_4_build/dpi_tb
! vsim -c -do run_tb_mq.do # Run ModelSim/Questasim in console mode
cd ../../

```

Notice that the HDL simulation now shows:

- A warning that one of the verify assessments is being filtered
- An info message that coverage will be gathered for the other assessment
- There are no longer any errors.

- The test is marked as PASSED because the SystemVerilog simulation results match the Simulink simulation results.
- The functional coverage shows that coverage is achieved for the enabled assessment
- The overall functional coverage goal is met.

```
# ** Warning: Filtering functional coverage for Req_scenario_4:32:60 due to plusarg value <= 0.
# Time: 0 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.initVerifyInfo File: ../Req_4_dpi_pkg.sv Line: 139
# ** Info: Gathering coverage for 1 Simulink verify() calls.
# Time: 0 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.initVerifyInfo File: ../Req_4_dpi_pkg.sv Line: 159
# *****TEST COMPLETED (PASSED)*****
# ** Note: $finish : ./Req_4_dpi_tb.sv(98)
# Time: 2042 ns Iteration: 0 Instance: /Req_4_dpi_tb
# ** Info: Instance coverage for verify 'Req_scenario_4:32:39', coverpoint 'pass_cp': metric=100.00, at_least= 1 ( COVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 259
# ** Info: Overall coverage for Req_4_dpi_verify_calls: metric=100.00 ( COVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 266
# End time: 12:57:07 on Jun 08,2020, Elapsed time: 0:00:00
# Errors: 0, Warnings: 7
```

Increase the Coverage Goal of a Specific Verify Assessment

You can also change the desired functional coverage goal for any assessment by supplying a positive value to the plus arg. The default goal is to see at least 1 PASS status for the verify call. If you wanted to ensure that there were at least 2 verify PASS status checks you would supply a "2" as the plus arg value.

```
% Clear environment variables that influence the SV simulation
setenv EXTRA_SVDPI_COMP_ARGS
setenv EXTRA_SVDPI_SIM_ARGS

% Filter the failing |verify| and set a coverage goal of 2 for the other |verify|
setenv EXTRA_SVDPI_SIM_ARGS '+Req_scenario_4:32:60=-1 +Req_scenario_4:32:39=2'
cd Req_4_build/dpi_tb
! vsim -c -do run_tb_mq.do # Run ModelSim/QuestaSim in console mode
cd ../../
```

Notice that the HDL simulation now shows that the non-filtered verify is not meeting the coverage goal of at least 2 PASSES and therefore the test as a whole is not either.

```
# ** Warning: Filtering functional coverage for Req_scenario_4:32:60 due to plusarg value <= 0.
# Time: 0 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.initVerifyInfo File: ../Req_4_dpi_pkg.sv Line: 139
# ** Info: Gathering coverage for 1 Simulink verify() calls.
# Time: 0 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.initVerifyInfo File: ../Req_4_dpi_pkg.sv Line: 159
# *****TEST COMPLETED (PASSED)*****
# ** Note: $finish : ./Req_4_dpi_tb.sv(98)
# Time: 2042 ns Iteration: 0 Instance: /Req_4_dpi_tb
# ** Info: Instance coverage for verify 'Req_scenario_4:32:39', coverpoint 'pass_cp': metric= 0.00, at_least= 2 ( UNCOVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 259
# ** Info: Overall coverage for Req_4_dpi_verify_calls: metric= 0.00 ( UNCOVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 266
# End time: 13:29:56 on Jun 08,2020, Elapsed time: 0:00:00
# Errors: 0, Warnings: 7
```

Log Pass, Fail, and Untested Status for All Verify Assessments

You can add log output for all status checks for all unfiltered verify calls by adding the +VERBOSE_VERIFY plus arg. This might be useful if you need to check on the timing and distribution of the UNTESTED, PASS, and FAIL verify status values.

```
% Clear environment variables that influence the SV simulation
setenv EXTRA_SVDPI_COMP_ARGS
setenv EXTRA_SVDPI_SIM_ARGS

% Log every status check.
setenv EXTRA_SVDPI_SIM_ARGS +VERBOSE_VERIFY
```



```
cd Req_4_build/dpi_tb
! vsim -c -do run_tb_mq.do # Run ModelSim/QuartaSim in console mode
cd ../../
```

Notice that the HDL simulation now shows each UNTESTED and PASS status check value as a SystemVerilog info message and each FAIL status a SystemVerilog error.

```
# ** Info: Gathering coverage for          2 Simulink verify() calls.
# Time: 0 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.initVerifyInfo File: ../Req_4_dpi_pkg.sv Line: 159
# ** Info: Req_scenario_4:32:39: At step 'Check' verify id 'Simulink:verify_sc4_on' is Untested
# Time: 40 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 231
# ** Info: Req_scenario_4:32:60: At step 'Check2' verify id 'Simulink:verify_sc4_off' is Untested
# Time: 40 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 231
# ** Info: Req_scenario_4:32:39: At step 'Check' verify id 'Simulink:verify_sc4_on' is Untested
#
# <<< SNIP >>>
#
# ** Info: Req_scenario_4:32:39: At step 'Check' verify id 'Simulink:verify_sc4_on' Passed
# Time: 250 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 227
# ** Info: Req_scenario_4:32:60: At step 'Check2' verify id 'Simulink:verify_sc4_off' is Untested
# Time: 250 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 231
#
# <<< SNIP >>>
#
# ** Info: Req_scenario_4:32:39: At step 'Check' verify id 'Simulink:verify_sc4_on' is Untested
# Time: 740 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 231
# ** Info: Req_scenario_4:32:60: At step 'Check2' verify id 'Simulink:verify_sc4_off' is Untested
# Time: 740 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 231
# ** Info: Req_scenario_4:32:39: At step 'Check' verify id 'Simulink:verify_sc4_on' is Untested
# Time: 750 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 231
# ** Error: Req_scenario_4:32:60: At step 'Check2' verify id 'Simulink:verify_sc4_off' Failed
# Time: 750 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 237
#
# <<< SNIP >>>
#
# ** Info: Req_scenario_4:32:39: At step 'Check' verify id 'Simulink:verify_sc4_on' is Untested
# Time: 2040 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 231
# ** Info: Req_scenario_4:32:60: At step 'Check2' verify id 'Simulink:verify_sc4_off' is Untested
# Time: 2040 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.checkVerifyStatus File: ../Req_4_dpi_pkg.sv Line: 231
# *****TEST COMPLETED (PASSED)*****
# ** Note: $finish      : ../Req_4_dpi_tb.sv(98)
# Time: 2042 ns Iteration: 0 Instance: /Req_4_dpi_tb
# ** Info: Instance coverage for verify 'Req_scenario_4:32:39', coverpoint 'pass_cp': metric=100.00, at least= 1 ( COVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 259
# ** Info: Instance coverage for verify 'Req_scenario_4:32:60', coverpoint 'pass_cp': metric= 0.00, at least= 1 ( UNCOVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 259
# ** Info: Overall coverage for Req_4_dpi_verify_calls: metric= 50.00 ( UNCOVERED)
# Time: 2042 ns Scope: Req_4_dpi_pkg.VerifyInterfaceT.reportVerifyCoverage File: ../Req_4_dpi_pkg.sv Line: 266
# End time: 13:40:40 on Jun 08,2020, Elapsed time: 0:00:00
# Errors: 1, Warnings: 6
```

Conclusion

SystemVerilog DPI component generation and the Test Sequence block from Simulink Test™ can be used to migrate verification logic from Simulink to a HDL simulator with minimal effort.

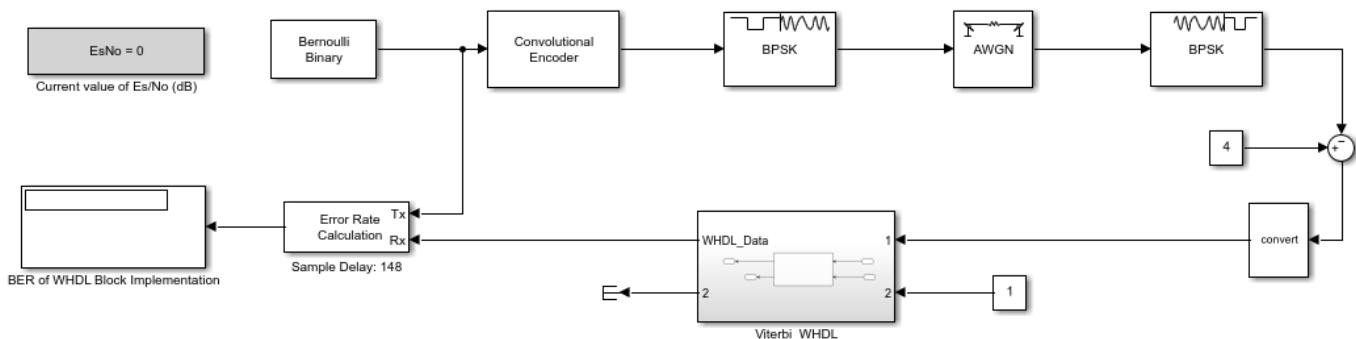
Verify Viterbi Decoder Using HDL Cosimulation

This example shows how to generate and verify HDL code to implement a fixed-point Viterbi decoder.

To run this example, in addition to the required MATLAB® products, you must install and include on the MATLAB system path either Mentor Graphics® ModelSim®/Questasim® or Cadence® Xcelium®.

Overview of Simulink Model

Open the Simulink® model `viterbi_codegen.slx`. This model generates HDL code for a fixed-point Viterbi decoder.



Copyright 2011-2021 The MathWorks, Inc.

The model uses binary phase-shift keying (BPSK) and additive white Gaussian noise (AWGN) blocks to simulate the wireless transmission of data. In the top model, the parameter **EsNo**, which represents the average signal energy to noise ratio, affects the transmission of data. By default, the **EsNo** parameter is set to 0.

After you initiate the data transmission, the test bench feeds the data into the Viterbi Decoder (Wireless HDL Toolbox) block, which is implemented using the Wireless HDL Toolbox™ product. The Viterbi Decoder block attempts to recover the original data but might have errors in the recovery. To measure how accurate this decoder is, the test bench sends the decoded data to an Error Rate Calculation (Communications Toolbox) block along with the original data. Then the Display (Simulink) block displays the results from this calculation.

Generate HDL Code

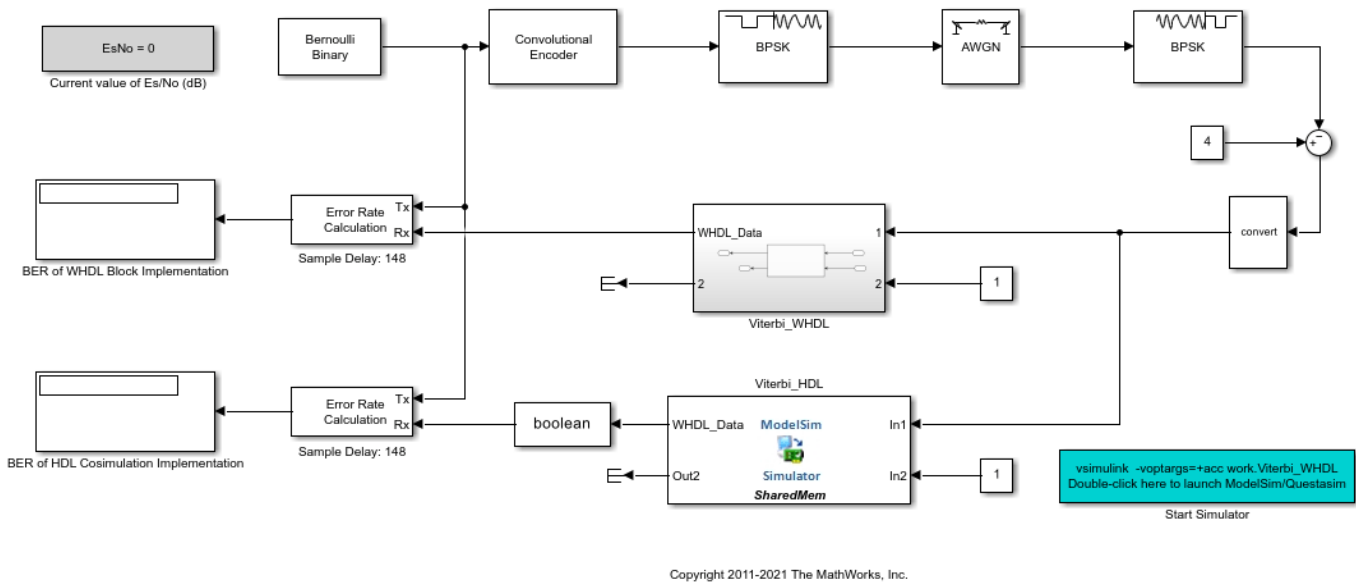
To open the **HDL Coder**(TM) app, on the **Apps** tab in the Simulink Toolstrip, click the **HDL Coder** app icon. To select the toolchain you want to use for your cosimulation, first click **Settings** to open the Configurations Parameters dialog box. In the left pane, click **HDL Code Generation**, then **Test Bench**. For the **Simulation tool** parameter, select the toolchain. Apply the changes by clicking **OK**.

To generate HDL code for the Viterbi decoder and open a new Simulink model, click **Generate Testbench**, then **HDL Cosimulation**.

Launch HDL Simulator

You can connect and format the new Simulink model to accommodate your test bench. This example includes two prepared models: `viterbi_modelsim.slx` and `viterbi_xcelium.slx`. Choose the

model that fits your toolchain. This example uses the ModelSim/Questasim Simulink model, which is shown in the figure.



To launch the HDL simulator, double-click the Start Simulator block in the model. In addition to launching the HDL simulator, this action inputs the commands to compile the HDL code and prepares for cosimulation with MATLAB and Simulink.

Run Simulation

When the HDL simulator finishes compiling the HDL files and preparing for simulation, the text `Ready for cosimulation ...` appears in the HDL simulator command window. After this text appears, return to the open model in Simulink and run the simulation from there.

When the simulation finishes, the Simulink model displays the results. In this example, the results are displayed as the bit error rate (BER) shown in the two Display blocks. The two displays show the BER results from the Viterbi Decoder block from the Wireless HDL Toolbox Product and the HDL coded block implemented using HDL Coder. Based on the results, the HDL Coder implementation yields the same results as the original block.

Rerun Simulation with New Parameters

The parameter **EsNo** controls the behavior of the transmission. Change this parameter to change the simulation behavior. For example, enter this command at the MATLAB command prompt.

```
EsNo = 5;
```

Changing this parameter does not require new HDL code to be generated, as this change does not affect the Viterbi block. To repeat this example with the new parameter value, run the simulation again from the open Simulink model.

Finish Simulation

After you are finished with simulation, close the HDL simulator session. Then, return to Simulink and close the model.

See Also**Functions**

makehdl (HDL Coder) | makehdltb (HDL Coder)

Blocks

Viterbi Decoder (Wireless HDL Toolbox)

Related Topics

- “Set Up for HDL Cosimulation” on page 10-2
- “Run MATLAB-HDL Cosimulation” on page 1-4
- “Generate HDL Code” (Wireless HDL Toolbox)
- “Choose a Test Bench for Generated HDL Code” (HDL Coder)

Generate Parameterized UVM Test Bench from Simulink

This example shows how you can develop a design and test bench in Simulink® and generate an equivalent simulation for a Universal Verification Methodology (UVM) environment using `uvmbuild`. Related examples show how you can extend this test bench to refine your verification using protocol-specific drivers, constrained random sequences, and parameterized scoreboards.

Introduction

This example walks you through a top-down design development process of an HDL implementation. In such a workflow, you design a behavioral algorithm in Simulink and test it using surrounding blocks for stimulus generation and results checking. Once the simulation confirms that the design meets its requirements, you deliver any collateral needed to the downstream HDL implementation team. You need to re-verify the HDL implementation meets the requirements as simulated in Simulink as well as any other unique aspects of the design such as protocol interfaces that were not modeled in Simulink.

Ordinarily the hand-off process can be tedious and the source of many errors. The HDL implementation and HDL Design Verification (DV) engineers must:

- Translate written specifications to HDL and testing environments.
- Understand the run-time behavior of the Simulink simulation environment such as how the stimulus is created, processed, and checked.
- Translate the run-time behaviors to SystemVerilog implementations.
- Integrate the stimulus, design, and response checking into a runnable SystemVerilog to confirm that the translated behaviors behave the same as the original Simulink simulation.
- Integrate these main SystemVerilog components into a UVM context in order to allow extending the Simulink testing with DV-authored verification. This extended testing might include randomized testing, SystemVerilog assertions, functional coverage, and code coverage.

Using the HDL Verifier™ UVM generation capabilities, this hand-off process is automated. The DV engineer gets a verified UVM test environment that matches the testing performed in Simulink and can easily update that environment to meet their downstream verification needs.

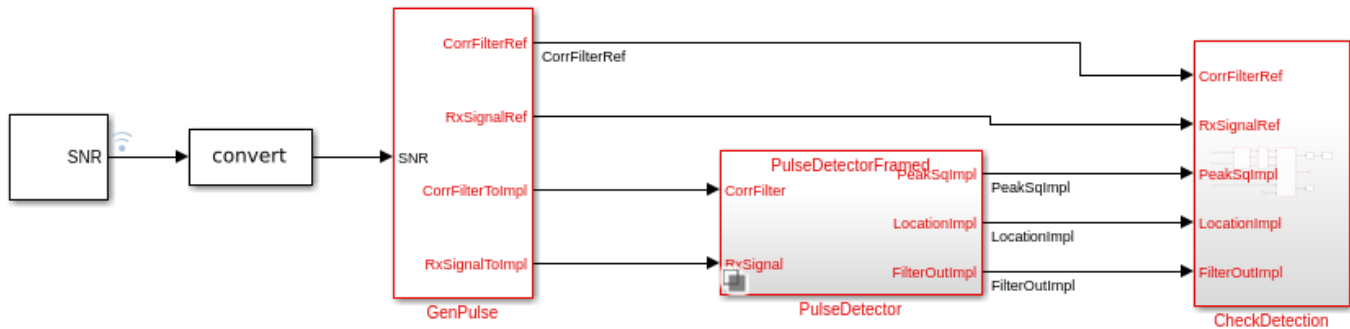
Design and Test in Simulink

Write your algorithm and add a test bench to it. The model consists of typical subsystems for a test bench such as stimulus generation, the design under test (DUT), and response checking.

In this design, the source subsystem creates a random pulse of 64 samples of information embedded at a random location in a 5000 sample frame of noise. It also generates a set of 64 optimal matched filter coefficients for detection of the pulse. The inputs are fed to both the design and the response checker. The response checker verifies that the pulse is detected at the right location in the noisy waveform. Proper operation is confirmed through console output. If the expected power of the detected signal is not within certain limits, an assertion is fired.

?

Generate Parameterized UVM Testbench from Simulink for a Pulse Detector Design



Copyright 2019 The MathWorks, Inc.

Simulating the model provides confirmation that in five generated pulses, five are detected. A three paneled figure shows a Tx Signal (the original pulse), an Rx Signal (the pulse embedded in noise), and the filtered output of a reference implementation that shows where a peak is detected. The output signal is delayed by one frame.

```
[FrameNum= 0] Peak location=2163.000000, mag-squared=0.280 using global max
[FrameNum= 0] Peak detected from impl=2163 error(abs)=0
[FrameNum= 0] Peak mag-squared from impl=0.280, error(abs)=0.000 error(pct)=0.017

[FrameNum= 1] Peak location=2163.000000, mag-squared=0.200 using global max
[FrameNum= 1] Peak detected from impl=2163 error(abs)=0
[FrameNum= 1] Peak mag-squared from impl=0.199, error(abs)=0.000 error(pct)=0.190

[FrameNum= 2] Peak location=2163.000000, mag-squared=0.224 using global max
[FrameNum= 2] Peak detected from impl=2163 error(abs)=0
[FrameNum= 2] Peak mag-squared from impl=0.223, error(abs)=0.000 error(pct)=0.183

[FrameNum= 3] Peak location=2163.000000, mag-squared=0.200 using global max
[FrameNum= 3] Peak detected from impl=2163 error(abs)=0
[FrameNum= 3] Peak mag-squared from impl=0.200, error(abs)=0.000 error(pct)=0.043

[FrameNum= 4] Peak location=2163.000000, mag-squared=0.255 using global max
[FrameNum= 4] Peak detected from impl=2163 error(abs)=0
[FrameNum= 4] Peak mag-squared from impl=0.255, error(abs)=0.000 error(pct)=0.031

[FrameNum= 5] Peak location=2163.000000, mag-squared=0.241 using global max
[FrameNum= 5] Peak detected from impl=2163 error(abs)=0
[FrameNum= 5] Peak mag-squared from impl=0.241, error(abs)=0.000 error(pct)=0.187

[FrameNum= 6] Peak location=2163.000000, mag-squared=0.241 using global max
[FrameNum= 6] Peak detected from impl=2163 error(abs)=0
[FrameNum= 6] Peak mag-squared from impl=0.241, error(abs)=0.000 error(pct)=0.019
```

```

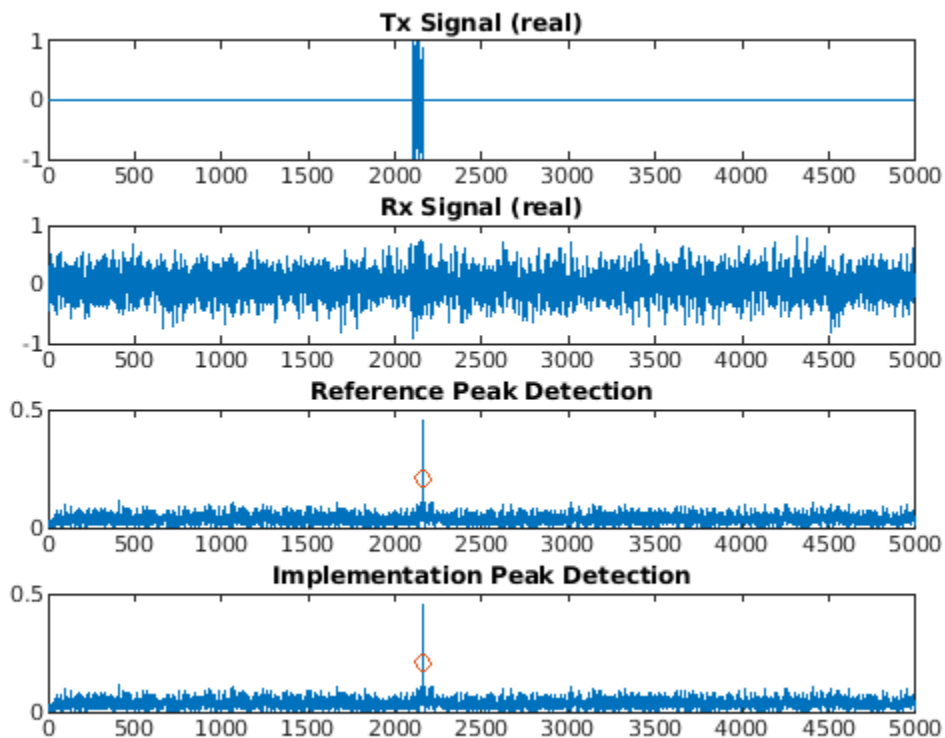
[FrameNum= 7] Peak location=2163.000000, mag-squared=0.225 using global max
[FrameNum= 7] Peak detected from impl=2163 error(abs)=0
[FrameNum= 7] Peak mag-squared from impl=0.225, error(abs)=0.000 error(pct)=0.032

[FrameNum= 8] Peak location=2163.000000, mag-squared=0.239 using global max
[FrameNum= 8] Peak detected from impl=2163 error(abs)=0
[FrameNum= 8] Peak mag-squared from impl=0.239, error(abs)=0.000 error(pct)=0.037

[FrameNum= 9] Peak location=2163.000000, mag-squared=0.225 using global max
[FrameNum= 9] Peak detected from impl=2163 error(abs)=0
[FrameNum= 9] Peak mag-squared from impl=0.225, error(abs)=0.000 error(pct)=0.146

[FrameNum= 10] Peak location=2163.000000, mag-squared=0.207 using global max
[FrameNum= 10] Peak detected from impl=2163 error(abs)=0
[FrameNum= 10] Peak mag-squared from impl=0.207, error(abs)=0.000 error(pct)=0.134

```



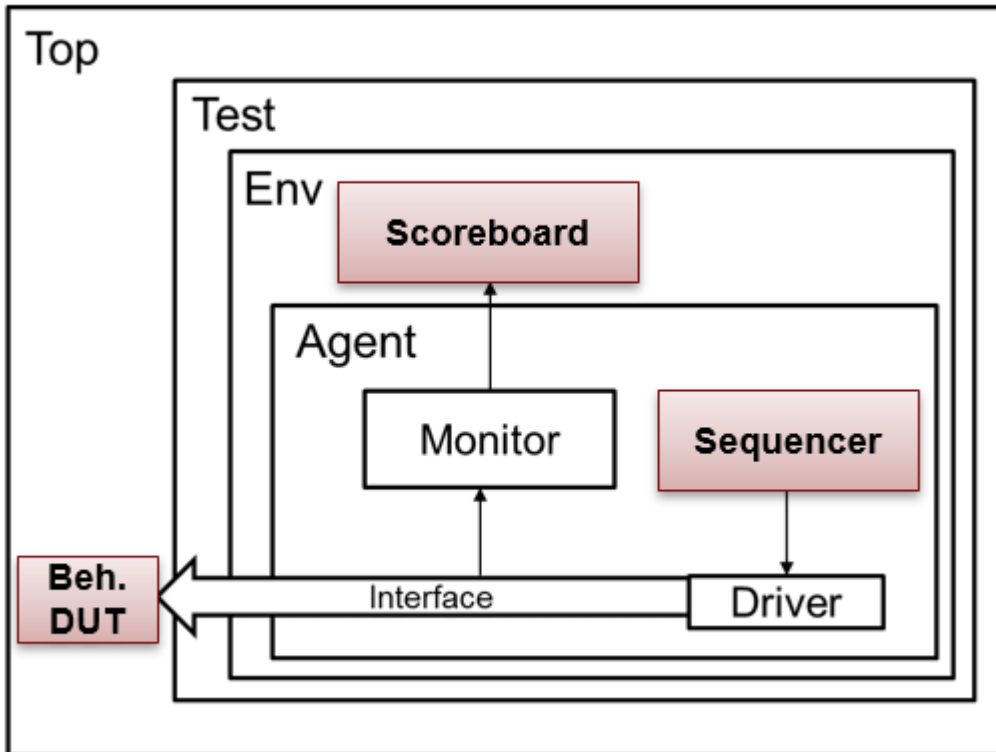
Generate an Executable UVM Test Bench

Use the `uvmbuild` function to export your design to a UVM environment. The UVM test bench provides structure to the HDL verification process and allows for all of the Simulink test bench components and test cases to be reused by the implementation verification team. Standard component definitions separate the pieces of the environment by their role in the simulation. For this example:

- `PulseDetector` is mapped to the DUT SystemVerilog module

- GenPulse subsystem is mapped to the sequence_item creation for the Sequencer UVM component
- CheckDetection subsystem is mapped to the Scoreboard UVM component.

The generated UVM test bench is shown below:



```
% Generate a UVM test bench
design      = 'prm_uvmtb/PulseDetector'
sequence   = 'prm_uvmtb/GenPulse'
scoreboard = 'prm_uvmtb/CheckDetection'
uvmbuild(design, sequence, scoreboard)
```

Each of the highlighted pieces of the UVM test bench are implemented by wrapping generated C-code from the Simulink subsystem and calling its entry points using DPI. The following image shows a couple of the function declarations for the PulseDetector subsystem.

```
5  `timescale 1ns / 1ns
6  package PulseDetector_dpi_pkg;
7
8  |
9  // Declare imported C functions
10 import "DPI-C" function chandle DPI_PulseDetector_initialize(chandle existhandle);
11 import "DPI-C" function chandle DPI_PulseDetector_reset(input chandle objhandle,
12 /*Simulink signal name: 'y'*/
13 input int unsigned y_Handle [64],
14 /*Simulink signal name: 'y_h'*/
15 input int unsigned y_h,
16 /*Simulink signal name: 'valid in'*/
```

The SystemVerilog/UVM code determines the timing of the DPI calls. For example, in the PulseDetector SystemVerilog module:

- The "initialize" DPI call is triggered by an "initial" code block.
- The "terminate" DPI call is triggered by a "final" code block.
- The "reset" DPI call is triggered by an active reset signal.
- The "output" and "update" DPI calls are triggered by a rising clock edge where reset is not active and the clock enable is active.

```

34     initial begin
35         objhandle = DPI_PulseDetector_initialize(objhandle);
36     end
37
38     final begin
39         DPI_PulseDetector_terminate(objhandle);
40     end
41
42     always @(posedge clk or posedge reset) begin
43         if(reset== 1'b1) begin
44             objhandle=DPI_PulseDetector_reset(objhandle,
45             y_Handle, y_h, valid_in, data_out_temp, valid_out_temp);
46             data_out<=data_out_temp;
47             valid_out<=valid_out_temp;
48         end
49         else if(clk_enable) begin
50             DPI_PulseDetector_output(objhandle,
51             y_Handle, y_h, valid_in, data_out_temp, valid_out_temp);
52             DPI_PulseDetector_update(objhandle,
53             y_Handle, y_h, valid_in);
54             data_out<=data_out_temp;
55             valid_out<=valid_out_temp;
56         end
57     end

```

Run the UVM Test Bench

The `uvmbuild` process also generates a script to run a simulation of the UVM test. Scripts are generated for the following simulators:

- Mentor Graphics® Modelsim® and Questa®: `run_tb_mq.do`
- Cadence® Xcelium™: `run_tb_xcelium.sh`
- Synopsys® VCS®: `run_tb_vcs.sh`

The generated script for ModelSim is shown.

```

vlib work

eval vlog \
  ../DPI_dut/PulseDetector_dpi_pkg.sv \
  ../sequence/GenPulse_dpi_pkg.sv \
  ../scoreboard/CheckDetection_dpi_pkg.sv \
  prm_uvmtb_pkg.sv \
  $EXTRA_UVM_COMP_ARGS \
  mw_PulseDetector_top.sv \

eval vsim \
  $EXTRA_UVM_SIM_ARGS \
  -L work \
  -voptargs=+acc \
  -sv_lib ../DPI_dut/PulseDetector \
  -sv_lib ../sequence/GenPulse \
  -sv_lib ../scoreboard/CheckDetection \
  +UVM_TESTNAME=mw_PulseDetector_test \
  mw_PulseDetector_top \

add wave -position end sim:/mw_PulseDetector_top/dutif/*

run -all

```

Execute the generated script to verify the UVM execution matches the Simulink execution. Because the sequence is parameterized with the SNR input port, its default value will be 0.0 in UVM. In order to properly compare the simulation runs, we need to change its default value to 2.0 (which has a bit value of 0b10_000000), to match Simulink; this can be done via a plusarg which we pass to the script via an environment variable.

```

% Clear environment variables that influence the UVM simulation'
setenv EXTRA_UVM_SIM_ARGS
setenv EXTRA_UVM_COMP_ARGS
setenv UVM_TOP_MODULE

% Simulate the UVM test bench using an SNR of 2.0
cd uvm_build/prm_uvmtb_uvm_testbench/top
setenv EXTRA_UVM_SIM_ARGS +SNR_default_inp_val=10000000
! vsim -do run_tb_mq.do      % ModelSim/Questasim (gui)
! vsim -c -do run_tb_mq.do  % ModelSim/Questasim (console)
! ./run_tb_xcelium.sh       % Xcelium (console)
! ./run_tb_vcs.sh          % VCS (console)
cd ../../../../

```

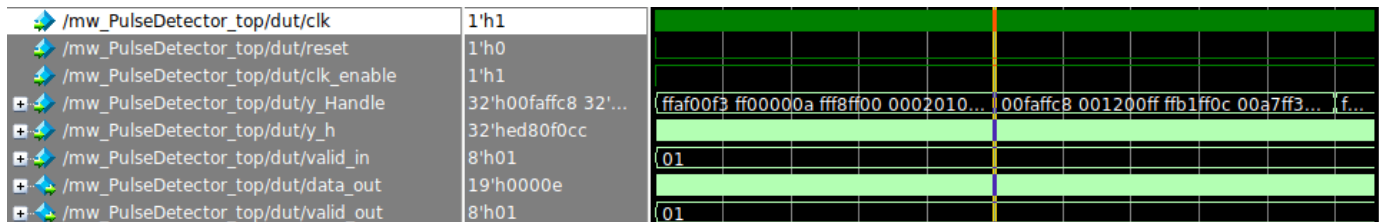
The simulation log shows the same diagnostic messages:

```

# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.3
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217) @ 0: reporter [Questa UVM] questa_uvm::init(+struct)
# UVM_INFO @ 0: reporter [RNTST] Running test mw_PulseDetector_test...
#
#
# [FrameNum= 0] Peak location=2163.000000, mag-squared=0.280 using global max
# [FrameNum= 0] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 0] Peak mag-squared from impl=0.280, error(abs)=0.000 error(pct)=0.017
#
# [FrameNum= 1] Peak location=2163.000000, mag-squared=0.200 using global max
# [FrameNum= 1] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 1] Peak mag-squared from impl=0.199, error(abs)=0.000 error(pct)=0.190
#
# [FrameNum= 2] Peak location=2163.000000, mag-squared=0.224 using global max
# [FrameNum= 2] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 2] Peak mag-squared from impl=0.223, error(abs)=0.000 error(pct)=0.183
#
# [FrameNum= 3] Peak location=2163.000000, mag-squared=0.200 using global max
# [FrameNum= 3] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 3] Peak mag-squared from impl=0.200, error(abs)=0.000 error(pct)=0.043
#
# [FrameNum= 4] Peak location=2163.000000, mag-squared=0.255 using global max
# [FrameNum= 4] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 4] Peak mag-squared from impl=0.255, error(abs)=0.000 error(pct)=0.031
#
# [FrameNum= 5] Peak location=2163.000000, mag-squared=0.241 using global max
# [FrameNum= 5] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 5] Peak mag-squared from impl=0.241, error(abs)=0.000 error(pct)=0.187
#
# [FrameNum= 6] Peak location=2163.000000, mag-squared=0.241 using global max
# [FrameNum= 6] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 6] Peak mag-squared from impl=0.241, error(abs)=0.000 error(pct)=0.019
#
# [FrameNum= 7] Peak location=2163.000000, mag-squared=0.225 using global max
# [FrameNum= 7] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 7] Peak mag-squared from impl=0.225, error(abs)=0.000 error(pct)=0.032
#
# [FrameNum= 8] Peak location=2163.000000, mag-squared=0.239 using global max
# [FrameNum= 8] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 8] Peak mag-squared from impl=0.239, error(abs)=0.000 error(pct)=0.037
#
# [FrameNum= 9] Peak location=2163.000000, mag-squared=0.225 using global max
# [FrameNum= 9] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 9] Peak mag-squared from impl=0.225, error(abs)=0.000 error(pct)=0.146
#
# [FrameNum= 10] Peak location=2163.000000, mag-squared=0.207 using global max
# [FrameNum= 10] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 10] Peak mag-squared from impl=0.207, error(abs)=0.000 error(pct)=0.134
#
# [FrameNum= 11] Peak location=2163.000000, mag-squared=0.265 using global max
# [FrameNum= 11] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 11] Peak mag-squared from impl=0.265, error(abs)=0.000 error(pct)=0.012
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @ 145: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 4
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
# ** Report counts by id
# [Questa UVM] 2
# [RNTST] 1
# [TEST_DONE] 1
# ** Note: $finish : //mathworks/hub/3rdparty/R2019b/4037151/share/Questasim/questasim/win64/./verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
# Time: 145 ns Iteration: 65 Instance: /mw_PulseDetector_top
.

```

And the waveform shows the timing of the DUT interface signals. The cursor is placed at a frame boundary and shows the instantaneous update of the matched filter coefficients.



Replace Behavioral DUT with AXI-Based RTL DUT in UVM Test Bench

This example shows how to move from a simple behavioral DUT interface to an RTL DUT interface which uses AXI Lite and AXI Stream bus protocols. Several of the UVM components must change such as the driver and monitor, but the original test bench structure and the sequence and scoreboard components can be re-used without modification.

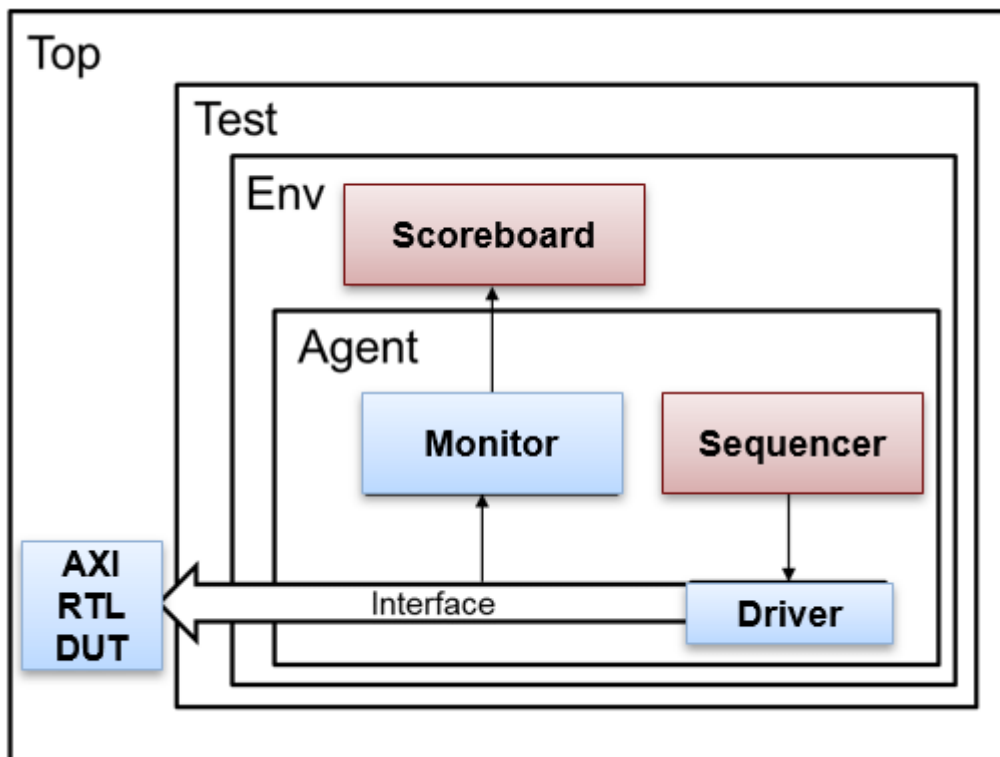
Introduction

See the example `Generate Parameterized UVM Test Bench from Simulink` for a description of the design and the background on generating a UVM test bench. To generate the default test bench for this example execute:

```
% Generate a UVM test bench
design      = 'prm_uvmtb/PulseDetector'
sequence   = 'prm_uvmtb/GenPulse'
scoreboard = 'prm_uvmtb/CheckDetection'
uvmbuild(design, sequence, scoreboard)
```

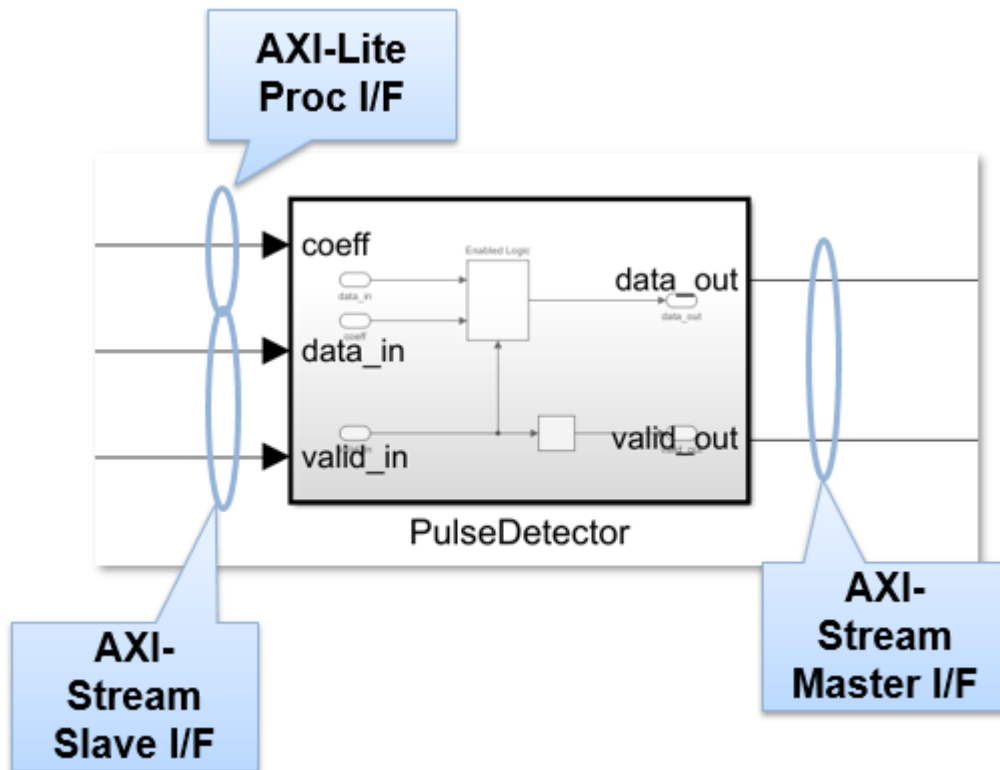
The DUT in Simulink represents the functional behavior of the pulse detector and does not use any hardware protocol interfaces that are typical of HDL IP cores. The UVM test bench uses the Simulink design as a stand-in for the actual HDL implementation. The next step in the verification workflow integrates an actual HDL implementation that uses AXI-based protocols into the same generated UVM test bench.

To use an RTL DUT you must substitute pieces of the UVM test bench, as shown in blue:



Mapping Ports to AXI Interfaces

For this RTL DUT, the `coeff` port is mapped to a processor interface, AXI4-Lite, the `data_in` port is mapped to an AXI4-Stream slave interface, and the `data_out` port is mapped to an AXI4-Stream master interface as shown below.



You can manually write this RTL or use HDL Coder's IP core generation workflow to create it. This model has a variant algorithm which can generate an HDL IP core using the HDL Coder product.

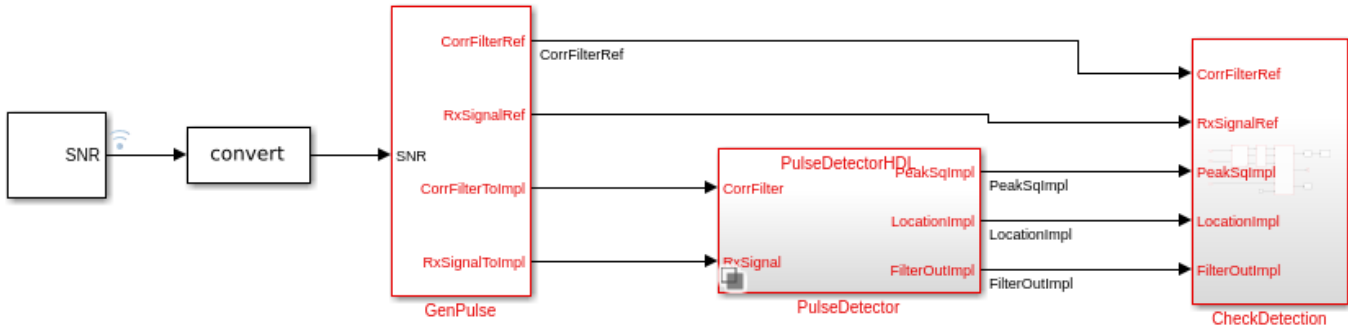
Simulating the HDL Variant in Simulink

To simulate with this variant execute:

```
% Simulate the RTL generatable algorithm variant
set_param('prm_uvmtb/PulseDetector','LabelModeActiveChoice','HDL')
set_param('prm_uvmtb/CheckDetection/ExpectsDelay','LabelModeActiveChoice','Frame delay')
sim('prm_uvmtb')
```



Generate Parameterized UVM Testbench from Simulink for a Pulse Detector Design



Copyright 2019 The MathWorks, Inc.

```
[FrameNum= 0] No peak found in Ref or Impl.

[FrameNum= 1] Peak location=2163.000000, mag-squared=0.280 using global max
[FrameNum= 1] Peak detected from impl=2170 error(abs)=7
[FrameNum= 1] Peak mag-squared from impl=0.280, error(abs)=0.000 error(pct)=0.017

[FrameNum= 2] Peak location=2163.000000, mag-squared=0.200 using global max
[FrameNum= 2] Peak detected from impl=2170 error(abs)=7
[FrameNum= 2] Peak mag-squared from impl=0.199, error(abs)=0.000 error(pct)=0.190

[FrameNum= 3] Peak location=2163.000000, mag-squared=0.224 using global max
[FrameNum= 3] Peak detected from impl=2170 error(abs)=7
[FrameNum= 3] Peak mag-squared from impl=0.223, error(abs)=0.000 error(pct)=0.183

[FrameNum= 4] Peak location=2163.000000, mag-squared=0.200 using global max
[FrameNum= 4] Peak detected from impl=2170 error(abs)=7
[FrameNum= 4] Peak mag-squared from impl=0.200, error(abs)=0.000 error(pct)=0.043

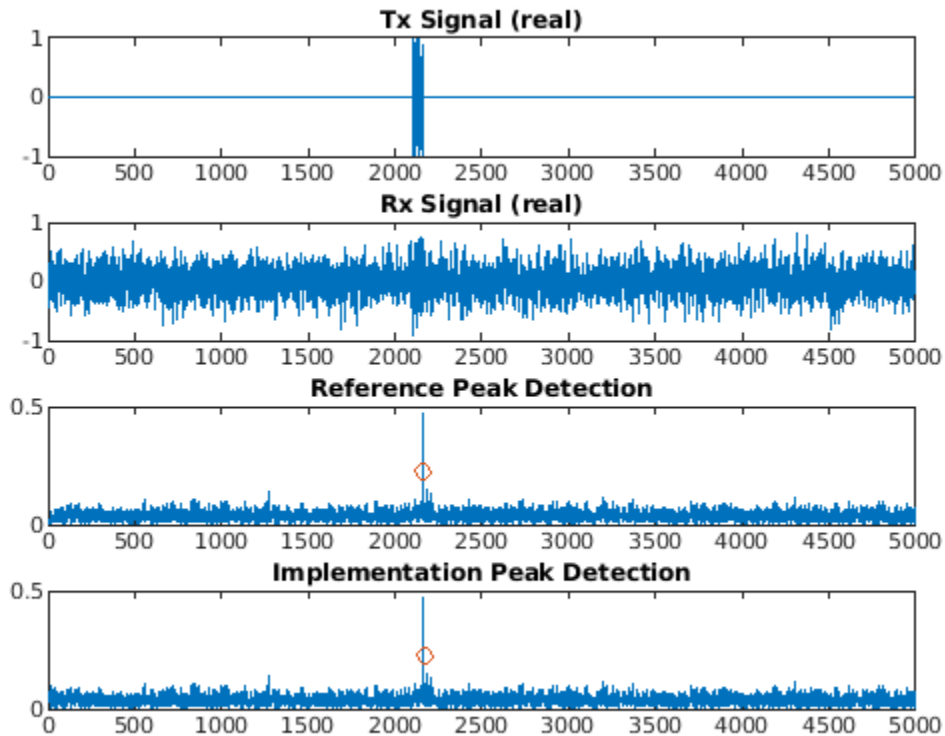
[FrameNum= 5] Peak location=2163.000000, mag-squared=0.255 using global max
[FrameNum= 5] Peak detected from impl=2170 error(abs)=7
[FrameNum= 5] Peak mag-squared from impl=0.255, error(abs)=0.000 error(pct)=0.031

[FrameNum= 6] Peak location=2163.000000, mag-squared=0.241 using global max
[FrameNum= 6] Peak detected from impl=2170 error(abs)=7
[FrameNum= 6] Peak mag-squared from impl=0.241, error(abs)=0.000 error(pct)=0.187

[FrameNum= 7] Peak location=2163.000000, mag-squared=0.241 using global max
[FrameNum= 7] Peak detected from impl=2170 error(abs)=7
[FrameNum= 7] Peak mag-squared from impl=0.241, error(abs)=0.000 error(pct)=0.019

[FrameNum= 8] Peak location=2163.000000, mag-squared=0.225 using global max
[FrameNum= 8] Peak detected from impl=2170 error(abs)=7
```

```
[FrameNum= 8] Peak mag-squared from impl=0.225, error(abs)=0.000 error(pct)=0.032
[FrameNum= 9] Peak location=2163.000000, mag-squared=0.239 using global max
[FrameNum= 9] Peak detected from impl=2170 error(abs)=7
[FrameNum= 9] Peak mag-squared from impl=0.239, error(abs)=0.001 error(pct)=0.241
[FrameNum= 10] Peak location=2163.000000, mag-squared=0.225 using global max
[FrameNum= 10] Peak detected from impl=2170 error(abs)=7
[FrameNum= 10] Peak mag-squared from impl=0.225, error(abs)=0.000 error(pct)=0.146
```



Generating the HDL

If you have HDL Coder, you can set up a tool path and generate RTL for this variant. For example, to create Xilinx Vivado compatible IP, you plug in your tool path and execute the following:

```
% Generate the AXI-based RTL HDL IP (HDL Coder required)
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', '/tools/Vivado/2018.3/bin/vivado')
prm_uvmtb_hdlworkflow
```

This will place the resulting HDL IP files into `hdl_prj/hdlsrc/prm_uvmtb`. However, the RTL implementation has been included with this demo located in `overrides_AXIDUT/AXIIPSource` and so no HDL Coder license is needed.

Overriding the Necessary UVM Components

The UVM files to replace the original DUT are located in `overrides_AXIDUT`. There is no need to modify any of the generated files from the original `uvmbuild` invocation.

- `mw_PulseDetector_AXIRTL_if.sv`: Redefines the main DUT interface class. Compare against the original interface definition.
- `mw_PulseDetector_AXIRTL_driver.sv`: Drives input transactions according to the AXI protocols of the DUT interface. Compare against the original driver.
- `mw_PulseDetector_AXIRTL_monitor/monitor_input.sv`: Receives streamed AXI data and converts to same transaction types as before. Compare against the original monitor definition.
- `mw_PulseDetector_AXIRTL_top.sv`: Instantiate the AXI RTL DUT and include the main UVM packages. Compare against the original top definition.

Simulating the UVM Test Bench

Execute the test bench with the new RTL DUT to verify the UVM execution matches the Simulink execution. Because the sequence is parameterized with the SNR input port, its default value will be 0.0 in UVM. In order to properly compare the simulation runs, we need to change its default value to 2.0 (which has a bit value of 0b10_000000), to match Simulink; this can be done via a plusarg which we pass to the script via an environment variable.

We have many new RTL files to compile and we must override the top level design unit. We pass these updates to the script through environment variables.

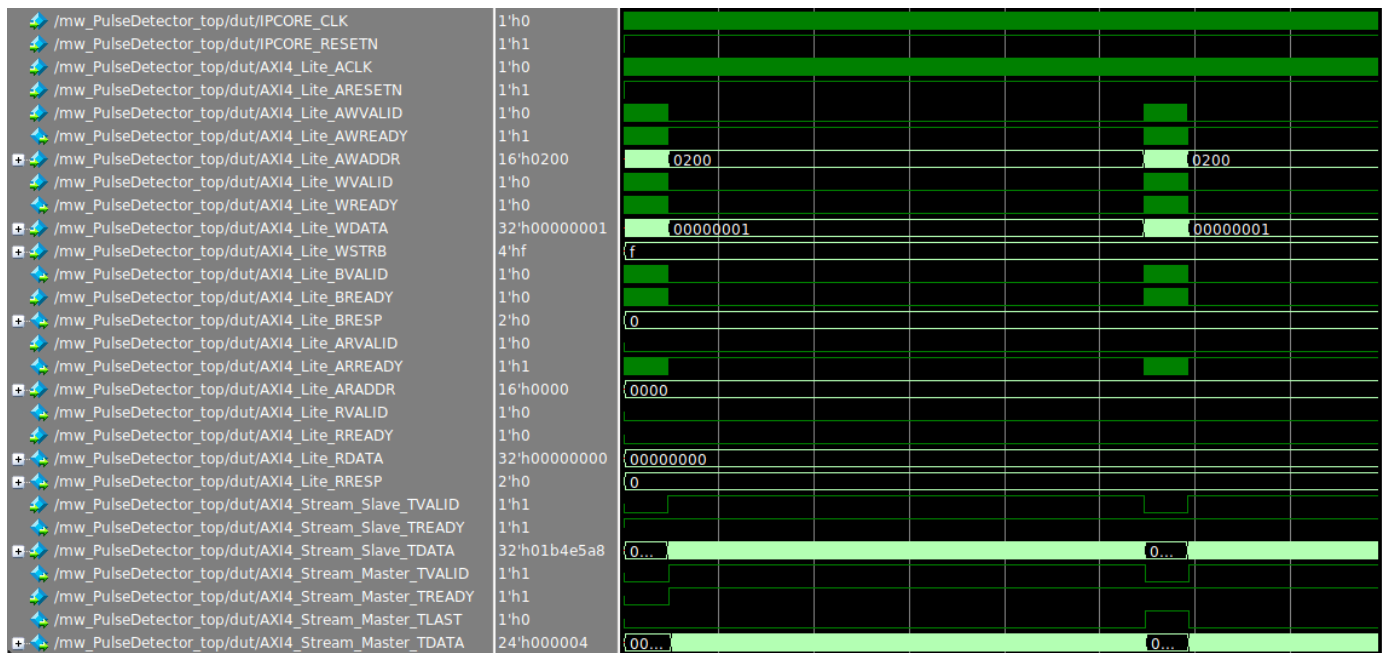
```
% Clear environment variables that influence the UVM simulation
setenv EXTRA_UVM_SIM_ARGS;
setenv EXTRA_UVM_COMP_ARGS;
setenv UVM_TOP_MODULE;
% Simulate the UVM test bench using the AXI RTL DUT
cd uvm_build/prm_uvmtb_uvm_testbench/top
setenv EXTRA_UVM_COMP_ARGS '-f ../../../../overrides_AXIDUT/extra_comp_args.f'
setenv EXTRA_UVM_SIM_ARGS '+SNR_default_inp_val=10000000 +UVM_TESTNAME=mw_PulseDetector_AXIRTL_t
setenv UVM_TOP_MODULE mw_PulseDetector_AXIRTL_top
! vsim -do run_tb_mq.do      % ModelSim/Questasim (gui)
! vsim -c -do run_tb_mq.do  % ModelSim/Questasim (console)
! ./run_tb_xcelium.sh      % Xcelium (console)
! ./run_tb_vcs.sh         % VCS (console)
cd ../../../../
```

The simulation log and a waveform snippet are shown below. Notice the log shows the same results as before, 5 detected pulses. However, also notice the waveforms show different timing for exercising the design. Instead of a steady stream of always valid data, for each frame of 5000 signal samples, the 64 coefficients are first programmed via the processor interface, then the 5000 samples are streamed.

```

# [FrameNum= 0] Peak location=2163.000000, mag-squared=0.280 using global max
# [FrameNum= 0] Peak detected from impl=2170 error(abs)=7
# [FrameNum= 0] Peak mag-squared from impl=0.280, error(abs)=0.000 error(pct)=0.017
#
# [FrameNum= 1] Peak location=2163.000000, mag-squared=0.200 using global max
# [FrameNum= 1] Peak detected from impl=2170 error(abs)=7
# [FrameNum= 1] Peak mag-squared from impl=0.199, error(abs)=0.000 error(pct)=0.190
#
# [FrameNum= 2] Peak location=2163.000000, mag-squared=0.224 using global max
# [FrameNum= 2] Peak detected from impl=2170 error(abs)=7
# [FrameNum= 2] Peak mag-squared from impl=0.223, error(abs)=0.000 error(pct)=0.183
#
# [FrameNum= 3] Peak location=2163.000000, mag-squared=0.200 using global max
# [FrameNum= 3] Peak detected from impl=2170 error(abs)=7
# [FrameNum= 3] Peak mag-squared from impl=0.200, error(abs)=0.000 error(pct)=0.043
#
# [FrameNum= 4] Peak location=2163.000000, mag-squared=0.255 using global max
# [FrameNum= 4] Peak detected from impl=2170 error(abs)=7
# [FrameNum= 4] Peak mag-squared from impl=0.255, error(abs)=0.000 error(pct)=0.031
#
# [FrameNum= 5] Peak location=2163.000000, mag-squared=0.241 using global max
# [FrameNum= 5] Peak detected from impl=2170 error(abs)=7
# [FrameNum= 5] Peak mag-squared from impl=0.241, error(abs)=0.000 error(pct)=0.187

```



Conclusion and Next Steps

This example has shown how a design and test bench developed in Simulink can be used to generate a fully executable UVM test bench. The `uvmbuild` command automates the generation, compilation, and integration of key components into the UVM framework.

HDL verification engineers can confirm overall coverage from Simulink and augment the coverage with their own library of native UVM sequences.

They can also substitute a behavioral design from Simulink with an RTL design that is wrapped in hardware protocols such as AXI4 with no changes to the original sequence generator and response checkers.

Add Random Constraints to Sequences in UVM Test Bench

This example shows how to add constrained random verification to a Universal Verification Methodology (UVM) test bench generated from Simulink®. Both Simulink parameters and input ports to the stimulus generation results in randomizable sequence class data members in the UVM test bench. Through the use of standard UVM class inheritance and factory overrides, the design verification engineer can add new and valuable constrained random testcases to their UVM test suite.

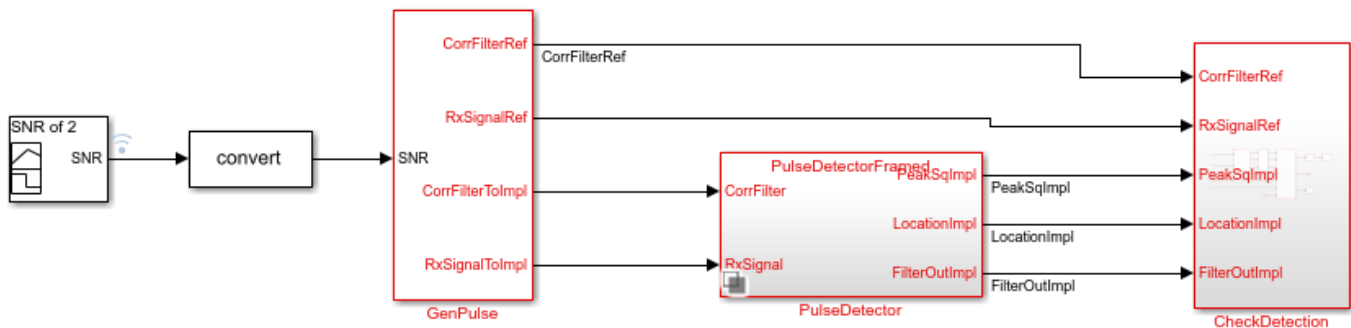
Introduction

This example extends a pulse generation UVM test bench to add constrained random testing. See the example [Generate Parameterized UVM Test Bench from Simulink](#) for a description of the design and the background on generating a UVM test bench. To generate the default test bench for this example, execute:

```
% Generate a UVM test bench
design      = 'prm_uvmtb/PulseDetector'
sequence   = 'prm_uvmtb/GenPulse'
scoreboard = 'prm_uvmtb/CheckDetection'
uvmbuild(design, sequence, scoreboard)
```



Generate Parameterized UVM Testbench from Simulink for a Pulse Detector Design



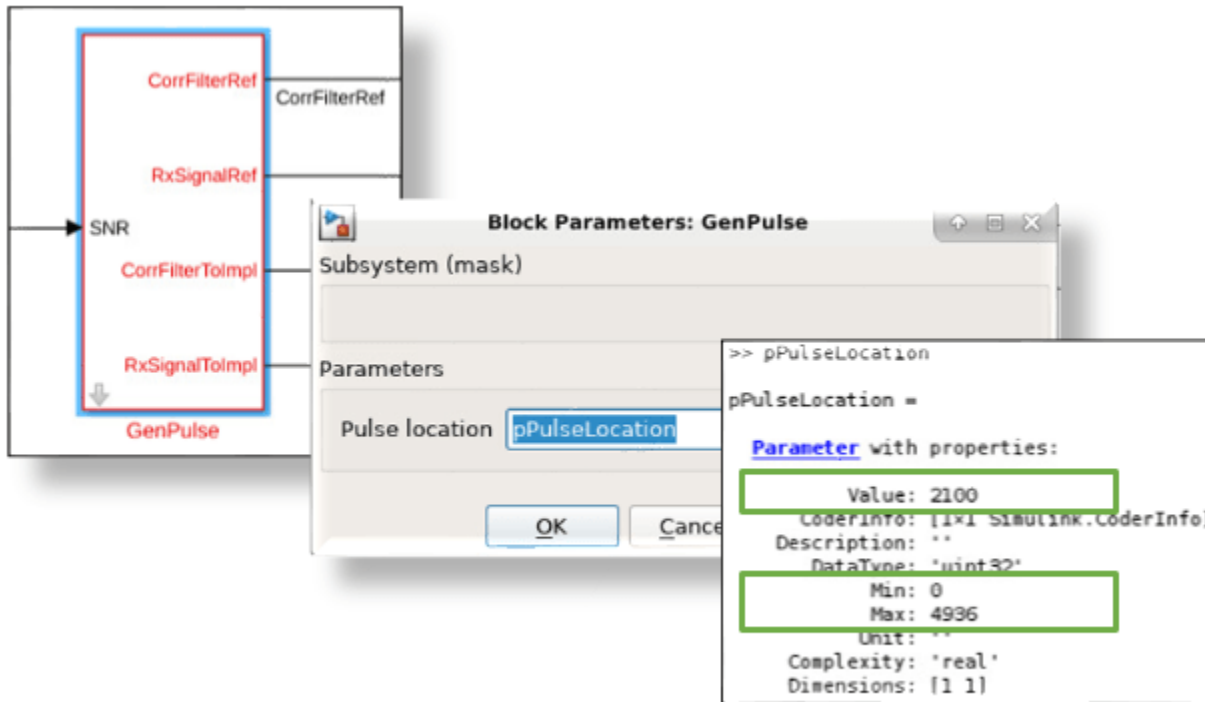
Copyright 2019 The MathWorks, Inc.

The Parameterization of the Pulse Generator

In the model the stimulus generation is parameterized using a dialog parameter for the pulse location and an input port for the signal-to-noise ratio (SNR). In the generated UVM, these parameters are data members of the `mw_PulseDetector_sequence` class with constraints that reflect information from the model.

See `mw_PulseDetector_sequence.sv`.

- `pPulseLocation` : This dialog parameter of the `GenPulse` subsystem indicates the start location of the 64 sample pulse in the larger 5000 sample frame. To retain in the SystemVerilog use a Simulink.Parameter. In that parameter, the default value is 2100 and the valid range is [0, 4936]. (The pulse must be entirely within the frame of 5000 samples.)



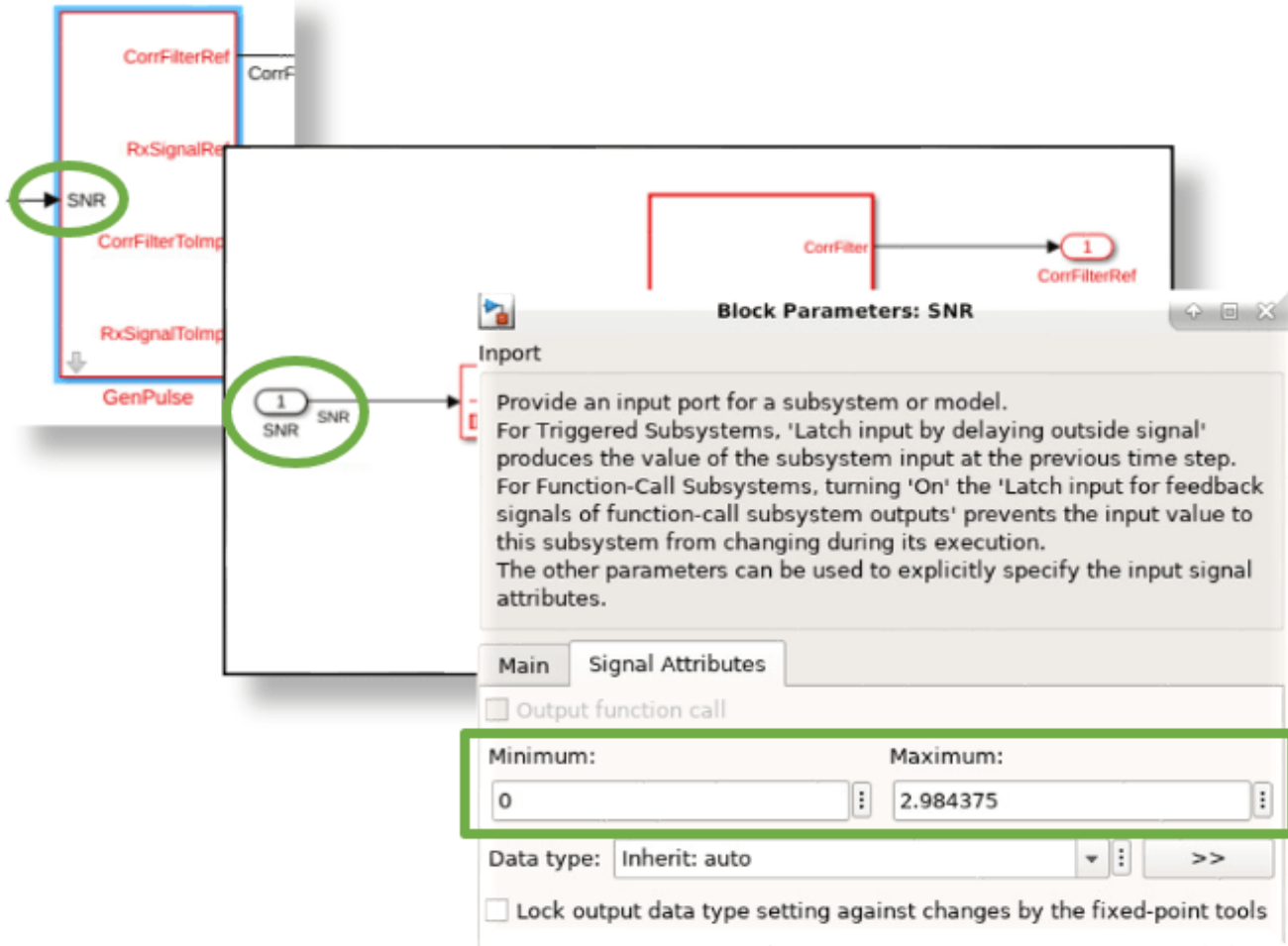
In the generated UVM code, two constraints are placed on the pulse location sequence member for a default value and a min, max range as shown in this code snippet:

```

18 // Simulink tunable parameters
19 rand int unsigned RTWStructParam_pPulseLocation;
20
21 // Simulink tunable parameter default values
22 constraint mw_PulseDetector_sequence_default_prm_val{
23     RTWStructParam_pPulseLocation==32'd2100;
24 }
25
26 // Simulink tunable parameter ranges
27 constraint mw_PulseDetector_sequence_range_vals{
28     RTWStructParam_pPulseLocation>=32'd0;
29     RTWStructParam_pPulseLocation<=32'd4936;
30 }

```

- `SNR` port : This input port to the generation subsystem indicates the overall magnitude relationship of the generated pulse to the generated noise. Its datatype is a `ufix8_En6` fixed point number with an intrinsic range of [0, 3.984375]. The port itself has a been given a more constrained range of [0, 2.984375] which corresponds to fixed point bit values of `0b00_000000`, `0b10_111111`.



Because it is an input port there is no notion of a default value in Simulink. In the generated UVM code, two constraints are placed on the sequence member for a min, max range as well as a default value that matches the minimum. Code is also added to support a command-line default value override via a plus arg. Example code:

```

32 // Simulink input port parameters
33 rand logic [7:0] SNR;
34 logic [7:0] SNR_default;
35 // Simulink input port parameter default values
36 constraint mw_PulseDetector_sequence_default_inp_prm_vals {
37     SNR==SNR_default;
38 }
39 // Simulink input port parameter ranges
40 constraint mw_PulseDetector_sequence_inp_range_vals{
41     SNR>=8'd0;
42     SNR<=8'b10111111;
43 }
44
45 function new (string name = "mw_PulseDetector_sequence");
46     super.new (name);
47     // Allow plusarg default for input port parameters
48     if (!$value$plusargs("RTWStructParam_SNR=%b",SNR_default))
49         SNR_default=8'b00000000;
50 endfunction // new

```

Default UVM Simulation Behavior

The default sequence generation body () follows the usual pattern of getting a grant, randomizing a sequence item, sending it along to sequencer, then waiting for its completion.

```

78 virtual task body ();
79     req = mw_PulseDetector_sequence_trans::type_id::create ("req");
80     //Number of transactions based on Simulink stop time
81     repeat(12)
82     begin
83         wait_for_grant ();
84         randomize_params ();
85         call_dpi_fcns ();
86         send_request (req);
87         wait_for_item_done ();
88     end
89     repeat(2)
90     begin
91         start_item(req);
92         finish_item(req);
93     end
94 endtask // body

```

Because of the default value constraints, if you run the default test it will use fixed values for the parameters. For this model, that means a pulse start location of 2100 and an SNR of 0.0. An SNR of 0.0 will result in no detectable pulses, so a plusarg specifies a more interesting default value. A fixed point bit value of 8'b10000000 is a float value of 2.0.

```

% Clear environment variables that influence the UVM simulation
setenv EXTRA_UVM_SIM_ARGS
setenv EXTRA_UVM_COMP_ARGS
setenv UVM_TOP_MODULE

% Simulate the UVM test bench using plusarg override
cd uvm_build/prm_uvmtb_uvm_testbench/top
setenv EXTRA_UVM_SIM_ARGS +SNR_default_inp_val=10000000

```

```
! vsim -do run_tb_mq.do      % ModelSim/QuestaSim (gui)
! vsim -c -do run_tb_mq.do  % ModelSim/QuestaSim (console)
! ./run_tb_xcelium.sh      % Xcelium (console)
! ./run_tb_vcs.sh          % VCS (console)
cd ../../..
```

```
# [FrameNum= 0] Peak location=2163.000000, mag-squared=0.280 using global max
# [FrameNum= 0] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 0] Peak mag-squared from impl=0.280, error(abs)=0.000 error(pct)=0.017
#
# [FrameNum= 1] Peak location=2163.000000, mag-squared=0.200 using global max
# [FrameNum= 1] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 1] Peak mag-squared from impl=0.199, error(abs)=0.000 error(pct)=0.190
#
# [FrameNum= 2] Peak location=2163.000000, mag-squared=0.224 using global max
# [FrameNum= 2] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 2] Peak mag-squared from impl=0.223, error(abs)=0.000 error(pct)=0.183
#
# [FrameNum= 3] Peak location=2163.000000, mag-squared=0.200 using global max
# [FrameNum= 3] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 3] Peak mag-squared from impl=0.200, error(abs)=0.000 error(pct)=0.043
#
# [FrameNum= 4] Peak location=2163.000000, mag-squared=0.255 using global max
# [FrameNum= 4] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 4] Peak mag-squared from impl=0.255, error(abs)=0.000 error(pct)=0.031
#
# [FrameNum= 5] Peak location=2163.000000, mag-squared=0.241 using global max
# [FrameNum= 5] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 5] Peak mag-squared from impl=0.241, error(abs)=0.000 error(pct)=0.187
```

Constrained Random Verification Using In-Lined Constraints

You can use standard UVM techniques to add randomized behavior to the stimulus generation. In this example, the testplan calls for using SNR values between 0.75 and 1.00 to determine how robust the algorithm is in detecting pulses in this range. For this case, the location of the pulse does not matter and is fixed at the end of the 5000 sample frames.

See `mw_PulseDetector_SEQCRT_param_overrides.sv`.

To achieve this testplan goal, a derived sequence class is created. It has the following extra behaviors:

- in the `pre_body()` override, the use of the default value constraints is turned off
- in the `post_randomize()` override, the randomized value is printed out so that SNR values can be related to whether pulses are detected or not. It uses a class utility function `fixed2real` to give a friendly value.
- in the `randomize_params()` override, an in-line constraint of "randomize with" restricts SNR values to be between 0.75 and 1.0. (You can get these bit values from MATLAB by using a `fi` variable and its `bin` method.)

```

14 // Define inline constraints in a new sequence object
15 class mw_PulseDetector_sequence_inlineCRT extends mw_PulseDetector_sequence;
16     `uvm_object_utils (mw_PulseDetector_sequence_inlineCRT)
17
18     virtual task pre_body();
19         super.pre_body();
20         // disable the default value constraints as we do not want those for CRT
21         mw_PulseDetector_sequence_default_prm_val.constraint_mode(0);
22         mw_PulseDetector_sequence_default_inp_prm_vals.constraint_mode(0);
23     endtask
24
25     function void post_randomize();
26         super.post_randomize();
27         `uvm_info(get_type_name(), $sformatf("\n\tSeq param vals: SNR=%2.4f, Loc=%4d\n",
28             fixed2real(SNR,6.0), RTWStructParam_pPulseLocation), UVM_LOW);
29     endfunction
30
31     virtual task randomize_params();
32     if (!randomize() with {
33         RTWStructParam_pPulseLocation == 4936;
34         SNR >= 8'b00110000; // >= 0.75
35         SNR <= 8'b01000000; // <= 1.0
36     })
37         `uvm_error("RNDFAIL",{"Unable to randomize sequence configuration in ", get_full_name() });
38     endtask
39
40     function new (string name = "mw_PulseDetector_sequence_inlineCRT");
41         super.new(name);
42     endfunction // new
43 endclass : mw_PulseDetector_sequence_inlineCRT

```

A new test is created which tells the factory to use the new sequence class when constructing the test bench.

```

45 class mw_PulseDetector_test_inlineCRT extends mw_PulseDetector_test;
46     `uvm_component_utils (mw_PulseDetector_test_inlineCRT)
47
48     function new (string name = "mw_PulseDetector_test_inlineCRT", uvm_component parent);
49         super.new (name, parent);
50         // Use the new sequence obj
51         mw_PulseDetector_sequence::type_id::set_type_override(mw_PulseDetector_sequence_inlineCRT::get_type());
52     endfunction // new
53 endclass : mw_PulseDetector_test_inlineCRT

```

To run the UVM simulations using these new classes, you can use the original scripts with extra arguments given through environment variables.

```

% Simulate the UVM test bench using inlined constraint overrides
cd uvm_build/prm_uvmtb_uvm_testbench/top
setenv EXTRA_UVM_COMP_ARGS '-f ../../../../overrides_SEQCRT/extra_comp_args.f'
setenv EXTRA_UVM_SIM_ARGS +UVM_TESTNAME=mw_PulseDetector_test_inlineCRT
! vsim -do run_tb_mq.do % ModelSim/QuestaSim (gui)
! vsim -c -do run_tb_mq.do % ModelSim/QuestaSim (console)
! ./run_tb_xcelium.sh % Xcelium (console)
! ./run_tb_vcs.sh % VCS (console)
cd ../../../../

```

```

# UVM_INFO @ 0: reporter [RNTST] Running test mw_PulseDetector_test_inlineCRT...
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(28) @ 15: uvm_test_top.env.PulseDetector_agent.sqr@@seq [mw_PulseDetector_sequence_inlineCRT]
#   Seq param vals: SNR=0.7500, Loc=4936
#
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(28) @ 25: uvm_test_top.env.PulseDetector_agent.sqr@@seq [mw_PulseDetector_sequence_inlineCRT]
#   Seq param vals: SNR=0.9844, Loc=4936
#
# [FrameNum= 0] Peak location=4999.000000, mag-squared=0.054 using global max
# [FrameNum= 0] Peak detected from impl=4999 error(abs)=0
# [FrameNum= 0] Peak mag-squared from impl=0.053, error(abs)=0.000 error(pct)=0.792
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(28) @ 35: uvm_test_top.env.PulseDetector_agent.sqr@@seq [mw_PulseDetector_sequence_inlineCRT]
#   Seq param vals: SNR=0.7969, Loc=4936
#
# [FrameNum= 1] Peak location=4999.000000, mag-squared=0.046 using global max
# [FrameNum= 1] Peak detected from impl=4999 error(abs)=0
# [FrameNum= 1] Peak mag-squared from impl=0.045, error(abs)=0.000 error(pct)=0.823
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(28) @ 45: uvm_test_top.env.PulseDetector_agent.sqr@@seq [mw_PulseDetector_sequence_inlineCRT]
#   Seq param vals: SNR=0.8438, Loc=4936
#
# [FrameNum= 2] No peak found in Ref or Impl.
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(28) @ 55: uvm_test_top.env.PulseDetector_agent.sqr@@seq [mw_PulseDetector_sequence_inlineCRT]
#   Seq param vals: SNR=0.7500, Loc=4936
#
# [FrameNum= 3] Peak location=4999.000000, mag-squared=0.036 using global max
# [FrameNum= 3] Peak detected from impl=4999 error(abs)=0
# [FrameNum= 3] Peak mag-squared from impl=0.035, error(abs)=0.000 error(pct)=1.273
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(28) @ 65: uvm_test_top.env.PulseDetector_agent.sqr@@seq [mw_PulseDetector_sequence_inlineCRT]
#   Seq param vals: SNR=0.7969, Loc=4936
#
# [FrameNum= 4] Peak location=4999.000000, mag-squared=0.043 using global max
# [FrameNum= 4] Peak detected from impl=4999 error(abs)=0
# [FrameNum= 4] Peak mag-squared from impl=0.043, error(abs)=0.000 error(pct)=0.252
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(28) @ 75: uvm_test_top.env.PulseDetector_agent.sqr@@seq [mw_PulseDetector_sequence_inlineCRT]
#   Seq param vals: SNR=0.7500, Loc=4936
#
# [FrameNum= 5] No peak found in Ref or Impl.
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(28) @ 85: uvm_test_top.env.PulseDetector_agent.sqr@@seq [mw_PulseDetector_sequence_inlineCRT]
#   Seq param vals: SNR=0.9531, Loc=4936

```

Examine the output and see that the generated SNR is always between [0.75, 1.00] and the pulse starting location is always 4936. With such an SNR range, observe that sometimes a pulse is detected and sometimes it is not.

Constrained Random Verification Using Class-Based Constraints

Inline constraints are good for honing in on specific test conditions. For fancier constrained random testing, you can create new constraint statement blocks in a derived sequence class. Here, the testplan requires that the location of the pulse varies across the frame using specific "buckets" of interesting locations in the beginning, middle, and end of the frame. A special location of 0 ensures that when no pulse is created, no pulse is detected. Likewise, for the SNR, several categories of interesting SNR ranges are defined such as SNR_never_detected and SNR_always_detected. The testwriter now can easily use these constraints to cover these scenarios.

See `mw_PulseDetector_SEQCRT_param_overrides.sv`.

To achieve these goals, the derived sequence class has the following extra behaviors:

- add a "loc_bucket" member declared as "randc".
- use that loc_bucket member to imply location ranges.
- add a "snr_bucket" member. It is not randomized, but rather is set by the testwriter.
- use the selected snr_bucket to imply randomization within specific SNR ranges.
- turn off the conflicting, default value constraints
- print out the randomized values for inspection during simulation


```

58 //
59 typedef enum { SNR_min, SNR_max, SNR_never_detected, SNR_always_detected, SNR_above1, SNR_none } snr_bucket_t;
60
61 class mw_PulseDetector_sequence_classCRT extends mw_PulseDetector_sequence;
62     `uvm_object_utils (mw_PulseDetector_sequence_classCRT)
63
64     // This constraint ensures we vary the location of the pulse to several locations
65     // across the 5000 sample frame.
66     randc bit [1:0] loc_bucket;
67     constraint c_pulse_location {
68         loc_bucket==0 -> RTWStructParam_pPulseLocation == 0; // 0 means no pulse is present--the signal is just noise
69         loc_bucket==1 -> RTWStructParam_pPulseLocation inside {[1:1000]};
70         loc_bucket==2 -> RTWStructParam_pPulseLocation inside {[1001:3000]};
71         loc_bucket==3 -> RTWStructParam_pPulseLocation inside {[3001:4936]};
72     }
73
74     snr_bucket_t snr_bucket;
75     constraint c_snr {
76         snr_bucket==SNR_min -> SNR == 8'b00000000; // at min
77         snr_bucket==SNR_max -> SNR == 8'b10111111; // at max
78         snr_bucket==SNR_never_detected -> SNR <= 8'b00001111; // below 0.2344
79         snr_bucket==SNR_always_detected -> SNR >= 8'b01011111; // above 1.4844
80         snr_bucket==SNR_above1 -> SNR >= 8'b01000000; // above 1.0000
81         // otherwise: random within existing min/max constraints
82     }
83
84     function new (string name = "mw_PulseDetector_sequence_classCRT");
85         super.new (name);
86         // disable the default value constraints as we do not want those for CRT
87         mw_PulseDetector_sequence_default_prm_val.constraint_mode(0);
88         mw_PulseDetector_sequence_default_inp_prm_vals.constraint_mode(0);
89         if (!uvm_resource_db#(snr_bucket_t)::read_by_name("SNR_sequence_trans", "SNR_control", snr_bucket))
90             snr_bucket = SNR_none;
91     endfunction
92
93     function void post_randomize();
94         super.post_randomize();
95         `uvm_info(get_type_name(),
96             $sformatf("\n\tSeq param vals: loc_bucket=%2d -> Loc=%4d, snr_bucket=%2d -> SNR=%2.4f\n",
97                 loc_bucket, RTWStructParam_pPulseLocation, snr_bucket, fixed2real(SNR,6.0)),
98                 UVM_LOW);
99     endfunction
100 endclass : mw_PulseDetector_sequence_classCRT

```

A new test is created which tells the UVM factory to use the new sequence class when constructing the test bench and also sets the specific SNR bucket through the `uvm_resource_db`.

```

104 class mw_PulseDetector_test_classCRT extends mw_PulseDetector_test;
105     `uvm_component_utils (mw_PulseDetector_test_classCRT)
106
107     function new (string name = "mw_PulseDetector_test_classCRT", uvm_component parent);
108         super.new (name, parent);
109         // Use the new sequence configuration type
110         mw_PulseDetector_sequence::type_id::set_type_override(mw_PulseDetector_sequence_classCRT::get_type());
111         uvm_resource_db#(snr_bucket_t)::set("SNR_sequence_trans", "SNR_control", SNR_always_detected);
112     endfunction // new
113 endclass : mw_PulseDetector_test_classCRT

```

To run the UVM simulations using these new classes, you can use the original scripts with extra arguments given through environment variables.

```

% Simulate the UVM test bench using derived sequence class overrides
cd uvm_build/prm_uvmtb_uvm_testbench/top
setenv EXTRA_UVM_COMP_ARGS '-f ../.././overrides_SEQCRT/extra_comp_args.f'
setenv EXTRA_UVM_SIM_ARGS +UVM_TESTNAME=mw_PulseDetector_test_classCRT
! vsim -do run_tb_mq.do      % ModelSim/Questasim (gui)
! vsim -c -do run_tb_mq.do  % ModelSim/Questasim (console)

```

```

! ./run_tb_xcelium.sh          % Xcelium (console)
! ./run_tb_vcs.sh             % VCS (console)
cd ../../..

# UVM_INFO @ 0: reporter [RNTST] Running test mw_PulseDetector_test_classCRT...
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(98) @ 15: uvm_test_top.env.PulseDetector_agent.sqr@seq [mw_PulseDetector_sequence_classCRT]
#   Seq param vals: loc_bucket= 1 -> Loc= 439, snr_bucket= 3 -> SNR=2.3125
#
#
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(98) @ 25: uvm_test_top.env.PulseDetector_agent.sqr@seq [mw_PulseDetector_sequence_classCRT]
#   Seq param vals: loc_bucket= 0 -> Loc= 0, snr_bucket= 3 -> SNR=2.9688
#
#
# [FrameNum= 0] Peak location=502.000000, mag-squared=0.200 using global max
# [FrameNum= 0] Peak detected from impl=502 error(abs)=0
# [FrameNum= 0] Peak mag-squared from impl=0.200, error(abs)=0.000 error(pct)=0.155
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(98) @ 35: uvm_test_top.env.PulseDetector_agent.sqr@seq [mw_PulseDetector_sequence_classCRT]
#   Seq param vals: loc_bucket= 3 -> Loc=3935, snr_bucket= 3 -> SNR=1.4844
#
#
# [FrameNum= 1] No peak found in Ref or Impl.
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(98) @ 45: uvm_test_top.env.PulseDetector_agent.sqr@seq [mw_PulseDetector_sequence_classCRT]
#   Seq param vals: loc_bucket= 2 -> Loc=1930, snr_bucket= 3 -> SNR=2.5625
#
#
# [FrameNum= 2] Peak location=3998.000000, mag-squared=0.129 using global max
# [FrameNum= 2] Peak detected from impl=3998 error(abs)=0
# [FrameNum= 2] Peak mag-squared from impl=0.128, error(abs)=0.000 error(pct)=0.212
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(98) @ 55: uvm_test_top.env.PulseDetector_agent.sqr@seq [mw_PulseDetector_sequence_classCRT]
#   Seq param vals: loc_bucket= 3 -> Loc=4807, snr_bucket= 3 -> SNR=1.5156
#
#
# [FrameNum= 3] Peak location=1993.000000, mag-squared=0.267 using global max
# [FrameNum= 3] Peak detected from impl=1993 error(abs)=0
# [FrameNum= 3] Peak mag-squared from impl=0.267, error(abs)=0.000 error(pct)=0.020
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(98) @ 65: uvm_test_top.env.PulseDetector_agent.sqr@seq [mw_PulseDetector_sequence_classCRT]
#   Seq param vals: loc_bucket= 1 -> Loc= 339, snr_bucket= 3 -> SNR=1.5781
#
#
# [FrameNum= 4] Peak location=4870.000000, mag-squared=0.126 using global max
# [FrameNum= 4] Peak detected from impl=4870 error(abs)=0
# [FrameNum= 4] Peak mag-squared from impl=0.126, error(abs)=0.000 error(pct)=0.152
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(98) @ 75: uvm_test_top.env.PulseDetector_agent.sqr@seq [mw_PulseDetector_sequence_classCRT]
#   Seq param vals: loc_bucket= 2 -> Loc=2428, snr_bucket= 3 -> SNR=2.8125
#
#
# [FrameNum= 5] Peak location=402.000000, mag-squared=0.208 using global max
# [FrameNum= 5] Peak detected from impl=402 error(abs)=0
# [FrameNum= 5] Peak mag-squared from impl=0.208, error(abs)=0.000 error(pct)=0.050
# UVM_INFO ./mw_PulseDetector_SEQCRT_param_overrides.sv(98) @ 85: uvm_test_top.env.PulseDetector_agent.sqr@seq [mw_PulseDetector_sequence_classCRT]
#   Seq param vals: loc_bucket= 0 -> Loc= 0, snr_bucket= 3 -> SNR=1.6562
#

```

Examine the output to see that the generated SNR is always above 1.4844 and the pulse starting location is visiting each of the ranges defined for `loc_bucket`. You can also confirm that when a location is 0, since no pulse is created, no pulse is detected.

Bypass Simulink Stimulus Generation

The generated sequences above are using randomization to affect the input parameters and arguments to the underlying GenPulse DPI-C functions. The actual transaction values are outputs of those functions. If you want to take full control of the actual transactions delivered to the sequencer, you can bypass the usual DPI calls and randomize the `req` data member.

See `mw_PulseDetector_SEQCRT_param_overrides.sv`.

```

121 class mw_PulseDetector_sequence_nodpi extends mw_PulseDetector_sequence;
122     `uvm_object_utils (mw_PulseDetector_sequence_nodpi)
123
124     virtual task randomize_params();
125         if(!req.randomize();
126             `uvm_error("RNDFAIL",{"Unable to randomize sequence_item in ", get_full_name()});
127         endtask // randomize_params
128
129     virtual task call_dpi_fcns();
130         // do nothing -- bypass use of DPI functions altogether
131         endtask // call_dpi_fcns
132 endclass : mw_PulseDetector_test_nodpi

```


In this case because the pulse, coefficients, and noise are all interrelated, completely randomizing their values will not yield valid test cases. But this technique is useful for other transaction types.

Conclusions and Next Steps

In a full UVM environment, whenever randomization is introduced it is usually essential to include coverage and to ensure different parts of the environment are aware of which random values were generated. In these examples randomized values are simply printed out. Adding coverage for generated SNR and location values--and relating those to detection and error thresholds--is an exercise left to the reader.

Change Parameters of Scoreboard in UVM Test Bench

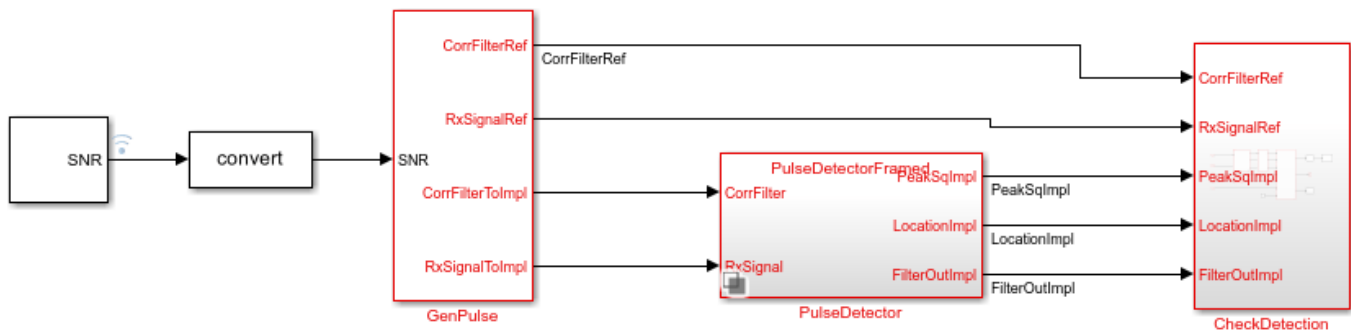
This example shows how to control the parameters of the response checker in a UVM test bench generated from Simulink®. Such parameterization helps the design verification engineer reuse the scoreboard under different testing scenarios. Simulink parameters to the checker will result in a configuration object whose values can be set in a derived test class (randomized or not) or directly via a command line plus arg.

Introduction

See the example Generate Parameterized UVM Test Bench from Simulink for a description of the design and the background on generating a UVM test bench. To generate the default test bench for this example, execute:

```
% Generate a UVM test bench
design      = 'prm_uvmtb/PulseDetector'
sequence   = 'prm_uvmtb/GenPulse'
scoreboard = 'prm_uvmtb/CheckDetection'
uvmbuild(design, sequence, scoreboard)
```

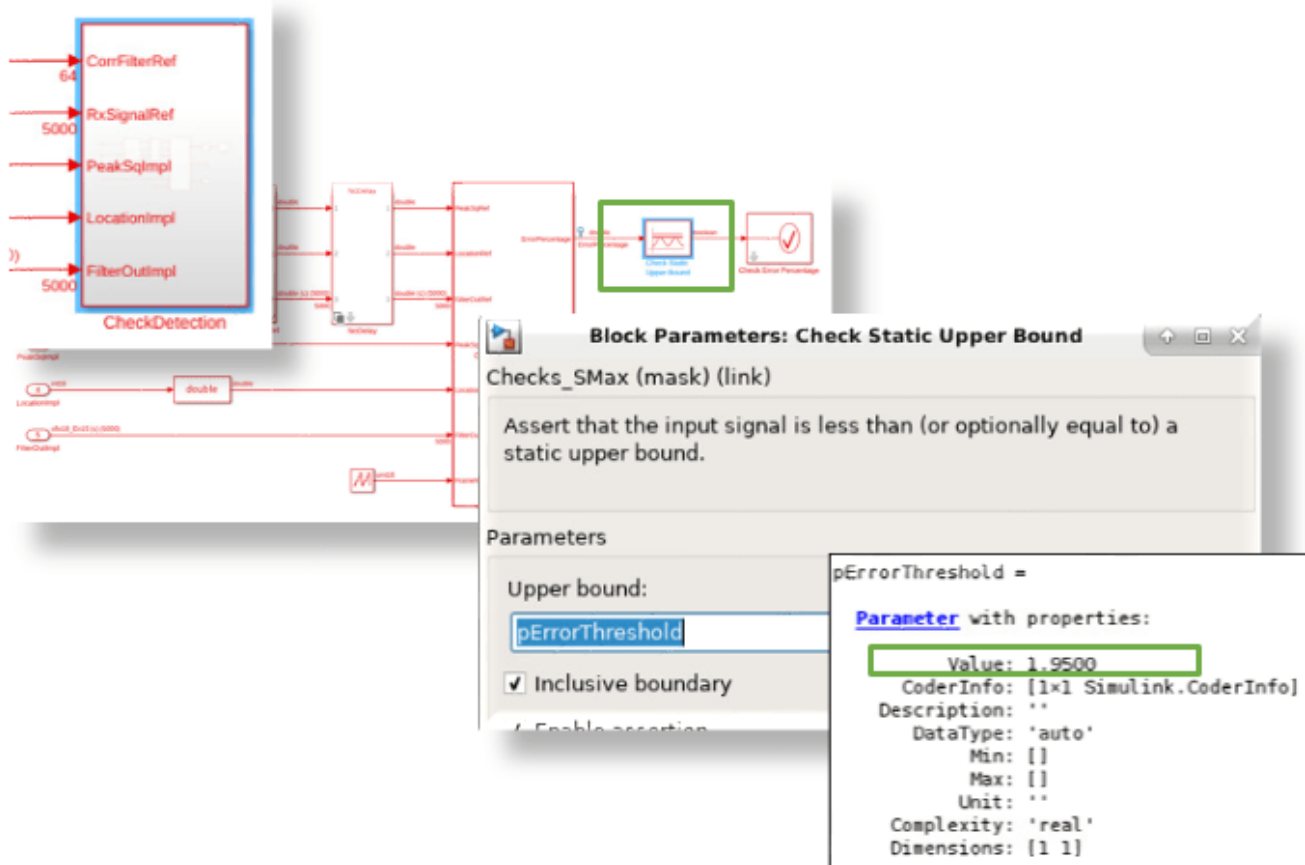
Generate Parameterized UVM Testbench from Simulink for a Pulse Detector Design



Copyright 2019 The MathWorks, Inc.

The Parameterization of the Scoreboard

In the model the response checking is parameterized by an error threshold variable used in an assertion checking block. It is designated as a parameter to retain in the SystemVerilog by using a Simulink.Parameter, pErrorThreshold. In that specification, it is given a default value of 1.95 percent, which reflects the tolerance of error difference between the golden reference and the actual DUT. (In this testbench, the discrepancy is due to double precision versus a fixed point implementations of the detection logic.)



In the generated UVM code, the parameter is placed in a configuration object, `mw_PulseDetector_scoreboard_cfg_obj`,

See `mw_PulseDetector_scoreboard_cfg_obj.sv`.

```

5 class mw_PulseDetector_scoreboard_cfg_obj extends uvm_object;
6   `uvm_object_utils (mw_PulseDetector_scoreboard_cfg_obj)
7
8   //Simulink tunable parameters
9   real RTWStructParam_pErrorThreshold;
10
11 function new (string name = "mw_PulseDetector_scoreboard_cfg_obj");
12   super.new (name);
13   //Get tunable parameter scalar values from plusargs
14   if (!$value$plusargs("RTWStructParam_pErrorThreshold=%f",RTWStructParam_pErrorThreshold)) begin
15     //If no plusarg is found then revert to the Simulink default value
16     RTWStructParam_pErrorThreshold=1.95;
17   end
18 endfunction // new
19
20 endclass:mw_PulseDetector_scoreboard_cfg_obj

```

which is instantiated in the build_phase of the test, `mw_PulseDetector_test`:

See `mw_PulseDetector_test.sv`.

```

21 virtual function void build_phase (uvm_phase phase);
22 super.build_phase (phase);
23 env = mw_PulseDetector_environment::type_id::create ("env", this);
24 seq = mw_PulseDetector_sequence::type_id::create ("seq", this);
25
26 //Create scoreboard configuration object
27 mw_PulseDetector_scoreboard_cfg_obj_id=mw_PulseDetector_scoreboard_cfg_obj::type_id::create("mw_PulseDetector_scoreboard_cfg_obj_id",this);
28 //Set tunable parameter values in the configuration database
29 uvm_config_db#(mw_PulseDetector_scoreboard_cfg_obj)::set(this,"env.PulseDetector_scoreboard","mw_PulseDetector_scoreboard_cfg_obj_id",mw_PulseDetector_scoreboard_cfg_obj_id);
30 endfunction // build_phase

```

and set via the `uvm_config_db` in the scoreboard, `mw_PulseDetector_scoreboard`, in its `start_of_simulation_phase`:

See `mw_PulseDetector_scoreboard.sv`.

```

48 virtual function void start_of_simulation_phase (uvm_phase phase);
49 super.start_of_simulation_phase (phase);
50 objhandle = DPI_CheckDetection_initialize(null);
51
52 //Get scoreboard configuration object from the database
53 if(!uvm_config_db#(mw_PulseDetector_scoreboard_cfg_obj)::get(null,get_full_name(),"mw_PulseDetector_scoreboard_cfg_obj_id",mw_PulseDetector_scoreboard_cfg_obj_id)) begin
54 //if configuration object is not in the database then use Simulink default configuration
55 mw_PulseDetector_scoreboard_cfg_obj_id=mw_PulseDetector_scoreboard_cfg_obj::type_id::create("mw_PulseDetector_scoreboard_cfg_obj_id",this);
56 end
57
58 //DPI function call to set tunable parameters
59 DPI_CheckDetection_setparam_pErrorThreshold(objhandle,mw_PulseDetector_scoreboard_cfg_obj_id.RTWStructParam_pErrorThreshold);
60 endfunction:start_of_simulation_phase

```

Updating the Error Threshold Using a Plus Argument

The test plan originally called for a tolerance of 1.95 percent. If later, the design specification was tightened to reflect a more safety critical requirement to 0.50 percent, this can be done using the command line. In our support scripts, we use an environment variable to affect the SystemVerilog command line plus argument.

```

% Clear environment variables that influence the UVM simulation
setenv EXTRA_UVM_SIM_ARGS
setenv EXTRA_UVM_COMP_ARGS
setenv UVM_TOP_MODULE

% Simulate the UVM test bench using plusarg overrides
cd uvm_build/prm_uvmtb_uvm_testbench/top
setenv EXTRA_UVM_SIM_ARGS '+SNR_default_inp_val=01000000 +RTWStructParam_pErrorThreshold=0.50'
! vsim -do run_tb_mq.do      % ModelSim/QuestaSim (gui)
! vsim -c -do run_tb_mq.do  % ModelSim/QuestaSim (console)
! ./run_tb_xcelium.sh      % Xcelium (console)
! ./run_tb_vcs.sh         % VCS (console)
cd ../../../../

```

```

# UVM_INFO @ 0: reporter [RNTST] Running test mw_PulseDetector_test...
#
# [FrameNum= 0] Peak location=2163.000000, mag-squared=0.088 using global max
# [FrameNum= 0] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 0] Peak mag-squared from impl=0.088, error(abs)=0.000 error(pct)=0.121
#
# [FrameNum= 1] Peak location=2163.000000, mag-squared=0.050 using global max
# [FrameNum= 1] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 1] Peak mag-squared from impl=0.050, error(abs)=0.000 error(pct)=0.536
# ** Error: prm_uvmtb:786:Error percentage above allowed threshold!
#   Time: 45 ns   Scope: prm_uvmtb_pkg.mw_PulseDetector_scoreboard.run_phase File: ../prm_uvmtb_uvmbuild/uvmtb_testbench/top/./scoreboard/mw_PulseDetector_scoreboard.sv Line: 79
#
# [FrameNum= 2] Peak location=2163.000000, mag-squared=0.079 using global max
# [FrameNum= 2] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 2] Peak mag-squared from impl=0.079, error(abs)=0.000 error(pct)=0.412
#
# [FrameNum= 3] Peak location=2163.000000, mag-squared=0.046 using global max
# [FrameNum= 3] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 3] Peak mag-squared from impl=0.046, error(abs)=0.000 error(pct)=0.692
# ** Error: prm_uvmtb:786:Error percentage above allowed threshold!
#   Time: 65 ns   Scope: prm_uvmtb_pkg.mw_PulseDetector_scoreboard.run_phase File: ../prm_uvmtb_uvmbuild/uvmtb_testbench/top/./scoreboard/mw_PulseDetector_scoreboard.sv Line: 79
#
# [FrameNum= 4] Peak location=2163.000000, mag-squared=0.074 using global max
# [FrameNum= 4] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 4] Peak mag-squared from impl=0.074, error(abs)=0.000 error(pct)=0.369
#
# [FrameNum= 5] Peak location=2163.000000, mag-squared=0.061 using global max
# [FrameNum= 5] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 5] Peak mag-squared from impl=0.061, error(abs)=0.000 error(pct)=0.624
# ** Error: prm_uvmtb:786:Error percentage above allowed threshold!
#   Time: 85 ns   Scope: prm_uvmtb_pkg.mw_PulseDetector_scoreboard.run_phase File: ../prm_uvmtb_uvmbuild/uvmtb_testbench/top/./scoreboard/mw_PulseDetector_scoreboard.sv Line: 79
#

```

Observe that with an SNR of 1.0 the design sometimes violates the 0.50 percent error threshold.

Randomize Error Threshold Using Default Configuration Object in Test

You can generate and set values directly from a test since the scoreboard configuration object is a member. Here we randomize a threshold in the range [0.050, 0.500] percent. Notice that the threshold is of type `real` and therefore cannot be randomized directly.

See `mw_PulseDetector_SCRPRM_param_overrides.sv`.

```

21 class mw_PulseDetector_test_SCRPRM extends mw_PulseDetector_test;
22   `uvm_component_utils (mw_PulseDetector_test_SCRPRM)
23   virtual function void build_phase (uvm_phase phase);
24       real rthresh;
25       super.build_phase (phase);
26       rthresh = $itor($urandom_range(50,500)) / 1000.0;
27       mw_PulseDetector_scoreboard_cfg_obj_id.RTWStructParam_pErrorThreshold = rthresh;
28       `uvm_info(get_type_name(), $sformatf("Setting error threshold to %4.3f percent\n", rthresh), UVM_LOW);
29   endfunction // build_phase
30
31   function new (string name = "mw_PulseDetector_test_SCRPRM", uvm_component parent);
32       super.new (name, parent);
33   endfunction // new
34
35 endclass : mw_PulseDetector_test_SCRPRM

```

To run the UVM simulations using these new classes, we can use the original scripts with extra arguments given through environment variables.

```

% Simulate the UVM test bench using randomized configuration object setting
cd uvm_build/prm_uvmtb_uvmtb_testbench/top
setenv EXTRA_UVM_COMP_ARGS '-f ../.././overrides_SCRPRM/extra_comp_args.f'
setenv EXTRA_UVM_SIM_ARGS '+SNR_default_inp_val=01000000 +UVM_TESTNAME=mw_PulseDetector_test_SCRPRM'
! vsim -do run_tb_mq.do      % ModelSim/QuestaSim (gui)
! vsim -c -do run_tb_mq.do  % ModelSim/QuestaSim (console)
! ./run_tb_xcelium.sh      % Xcelium (console)
! ./run_tb_vcs.sh         % VCS (console)
cd ../../../../

```

```

# UVM_INFO @ 0: reporter [RNTST] Running test mw_PulseDetector_test_SCRPRM...
# UVM_INFO ./mw_PulseDetector_SCRPRM_param_overrides.sv(28) @ 0: uvm_test_top [mw_PulseDetector_test_SCRPRM] Setting error threshold to 0.150 percent
#
#
# [FrameNum= 0] Peak location=2163.000000, mag-squared=0.088 using global max
# [FrameNum= 0] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 0] Peak mag-squared from impl=0.088, error(abs)=0.000 error(pct)=0.121
#
# [FrameNum= 1] Peak location=2163.000000, mag-squared=0.050 using global max
# [FrameNum= 1] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 1] Peak mag-squared from impl=0.050, error(abs)=0.000 error(pct)=0.536
# ** Error: prm_uvmtb:786:Error percentage above allowed threshold!
#   Time: 45 ns Scope: prm_uvmtb_pkg.mw_PulseDetector_scoreboard.run_phase File: ../prm_uvmtb_uvmbuild/uvm_testbench/top/./scoreboard/mw_PulseDetector_scoreboard.sv Line: 79
#
# [FrameNum= 2] Peak location=2163.000000, mag-squared=0.079 using global max
# [FrameNum= 2] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 2] Peak mag-squared from impl=0.079, error(abs)=0.000 error(pct)=0.412
# ** Error: prm_uvmtb:786:Error percentage above allowed threshold!
#   Time: 55 ns Scope: prm_uvmtb_pkg.mw_PulseDetector_scoreboard.run_phase File: ../prm_uvmtb_uvmbuild/uvm_testbench/top/./scoreboard/mw_PulseDetector_scoreboard.sv Line: 79
#
# [FrameNum= 3] Peak location=2163.000000, mag-squared=0.046 using global max
# [FrameNum= 3] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 3] Peak mag-squared from impl=0.046, error(abs)=0.000 error(pct)=0.692
# ** Error: prm_uvmtb:786:Error percentage above allowed threshold!
#   Time: 65 ns Scope: prm_uvmtb_pkg.mw_PulseDetector_scoreboard.run_phase File: ../prm_uvmtb_uvmbuild/uvm_testbench/top/./scoreboard/mw_PulseDetector_scoreboard.sv Line: 79
#
# [FrameNum= 4] Peak location=2163.000000, mag-squared=0.074 using global max
# [FrameNum= 4] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 4] Peak mag-squared from impl=0.074, error(abs)=0.000 error(pct)=0.369
# ** Error: prm_uvmtb:786:Error percentage above allowed threshold!
#   Time: 75 ns Scope: prm_uvmtb_pkg.mw_PulseDetector_scoreboard.run_phase File: ../prm_uvmtb_uvmbuild/uvm_testbench/top/./scoreboard/mw_PulseDetector_scoreboard.sv Line: 79
#
# [FrameNum= 5] Peak location=2163.000000, mag-squared=0.061 using global max
# [FrameNum= 5] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 5] Peak mag-squared from impl=0.061, error(abs)=0.000 error(pct)=0.624
# ** Error: prm_uvmtb:786:Error percentage above allowed threshold!
#   Time: 85 ns Scope: prm_uvmtb_pkg.mw_PulseDetector_scoreboard.run_phase File: ../prm_uvmtb_uvmbuild/uvm_testbench/top/./scoreboard/mw_PulseDetector_scoreboard.sv Line: 79
#

```

Observe that in this run, a threshold of 0.150 percent is chosen and that several frames end up violating this threshold.

Randomize Error Threshold Using Derived Configuration Object

You can also create more sophisticated randomization of the configuration settings by creating a derived scoreboard configuration object. In this example, the test plan requires that the threshold fall in different interesting ranges that reflect different operating environments of the detector.

See `mw_PulseDetector_SCRPRM_param_overrides.sv`.

To achieve this goal, the derived configuration object has the follow extra behaviors:

- add a "threshold_bucket" member declared as `randc`
- use that `threshold_bucket` member to imply threshold ranges
- print out the randomized values for inspection during simulation


```

41 class mw_PulseDetector_scoreboard_cfg_CRT extends mw_PulseDetector_scoreboard_cfg_obj;
42     `uvm_object_utils(mw_PulseDetector_scoreboard_cfg_CRT)
43
44     randc bit[1:0] threshold_bucket;
45
46     function void post_randomize();
47         int ithresh;
48         real rthresh;
49         case (threshold_bucket)
50             2'd0: ithresh = $urandom_range( 0, 10);
51             2'd1: ithresh = $urandom_range( 10, 100);
52             2'd2: ithresh = $urandom_range(100, 150);
53             2'd3: ithresh = $urandom_range(150, 200);
54         endcase
55         rthresh = $itor(ithresh)/100.0;
56         RTWStructParam_pErrorThreshold = rthresh;
57         `uvm_info(get_type_name(), $sformatf("Setting error threshold to %4.2f percent\n", rthresh), UVM_LOW);
58     endfunction // post_randomize
59
60     function new (string name = "mw_PulseDetector_scoreboard_cfg_CRT");
61         super.new (name);
62     endfunction
63 endclass : mw_PulseDetector_scoreboard_cfg_CRT

```

A new test is created which tells the UVM factory to use the new configuration object and to randomize its settings during the build phase.

```

66 class mw_PulseDetector_test_SCRPRM2 extends mw_PulseDetector_test;
67     `uvm_component_utils (mw_PulseDetector_test_SCRPRM2)
68
69     function new (string name = "mw_PulseDetector_test_SCRPRM2", uvm_component parent);
70         super.new (name, parent);
71         mw_PulseDetector_scoreboard_cfg_obj::type_id::set_type_override(mw_PulseDetector_scoreboard_cfg_CRT::get_type());
72     endfunction // new
73
74     virtual function void build_phase (uvm_phase phase);
75         super.build_phase (phase);
76         mw_PulseDetector_scoreboard_cfg_obj_id.randomize();
77     endfunction // build_phase
78 endclass : mw_PulseDetector_test_SCRPRM2

```

To run the UVM simulations using these new classes, we can use the original scripts with extra arguments given through environment variables.

```

% Simulate the UVM test bench using randomized derived configuration object
cd uvm_build/prm_uvmtb_uvm_testbench/top
setenv EXTRA_UVM_COMP_ARGS '-f ../../../../overrides_SCRPRM/extra_comp_args.f'
setenv EXTRA_UVM_SIM_ARGS '+SNR_default_inp_val=01000000 +UVM_TESTNAME=mw_PulseDetector_test_SCRPRM2'
! vsim -do run_tb_mq.do          % ModelSim/QuestaSim (gui)
! vsim -c -do run_tb_mq.do      % ModelSim/QuestaSim (console)
! ./run_tb_xcelium.sh          % Xcelium (console)
! ./run_tb_vcs.sh              % VCS (console)
cd ../../../../

```

```
# UVM_INFO @ 0: reporter [RNTST] Running test mw_PulseDetector_test_SCRPRM2...
# UVM_INFO ./mw_PulseDetector_SCRPRM_param_overrides.sv(57) @ 0: reporter [mw_PulseDetector_scoreboard_cfg_CRT] Setting error threshold to 1.01 percent
#
# [FrameNum= 0] Peak location=2163.000000, mag-squared=0.088 using global max
# [FrameNum= 0] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 0] Peak mag-squared from impl=0.088, error(abs)=0.000 error(pct)=0.121
#
# [FrameNum= 1] Peak location=2163.000000, mag-squared=0.050 using global max
# [FrameNum= 1] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 1] Peak mag-squared from impl=0.050, error(abs)=0.000 error(pct)=0.536
#
# [FrameNum= 2] Peak location=2163.000000, mag-squared=0.079 using global max
# [FrameNum= 2] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 2] Peak mag-squared from impl=0.079, error(abs)=0.000 error(pct)=0.412
#
# [FrameNum= 3] Peak location=2163.000000, mag-squared=0.046 using global max
# [FrameNum= 3] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 3] Peak mag-squared from impl=0.046, error(abs)=0.000 error(pct)=0.692
#
# [FrameNum= 4] Peak location=2163.000000, mag-squared=0.074 using global max
# [FrameNum= 4] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 4] Peak mag-squared from impl=0.074, error(abs)=0.000 error(pct)=0.369
#
# [FrameNum= 5] Peak location=2163.000000, mag-squared=0.061 using global max
# [FrameNum= 5] Peak detected from impl=2163 error(abs)=0
# [FrameNum= 5] Peak mag-squared from impl=0.061, error(abs)=0.000 error(pct)=0.624
#
```

Observe that for this run a threshold of 1.01 percent is chosen and that all frames are under this threshold.

Conclusions and Next Steps

In a full UVM environment, whenever randomization is introduced it is usually essential to include coverage and to ensure different parts of the environment are aware of which random values were generated. In these examples randomized values are simply printed out. Adding coverage for generated error thresholds and analyzing the actual signal errors over the course of many simulation runs is an exercise left to the reader.

See Also

uvmbuild

Related Topics

“Generate Parameterized UVM Test Bench from Simulink” on page 32-106

“Replace Behavioral DUT with AXI-Based RTL DUT in UVM Test Bench” on page 32-113

Include Driver and Monitor in UVM Test Bench

This example uses the `uvmbuild` function to generate a Universal Verification Methodology (UVM) test bench, which includes a UVM driver and monitor from a Simulink® design and test bench.

Introduction

Transitioning from frame-based behavioral modeling to scalar modeling in Simulink requires conversion blocks to transform signal sample times and sizes. The conversion blocks divide the Simulink test bench in these two domains.

- Frame-based domain: The part of the test bench that manipulates and operates on data as frames.
- Scalar-based domain: The part of the test bench that manipulates and operates on data as scalars.

The UVM framework uses transaction level modeling to process transactions from the sequence (stimulus generator) and to verify the resulting transactions in the scoreboard (checker).

When translating a Simulink test bench to a UVM test bench, keeping these domains separate is important. The domain separation enables better componentization and reusability of the test bench components.

Design and Test in Simulink

Write the scalar-based algorithm and add a test bench around it that separates the frame-based domain and the scalar domain using conversion subsystems (for example, buffer and unbuffer, rate transition, or a combination of these blocks). The model consists of a stimulus generation subsystem, a scalar-based design under test (DUT), and a response checking subsystem. Additional conversion subsystems are introduced between the stimulus generation and the DUT to convert the data from frames to scalars and between the DUT and the response checker to convert the data from scalars to frames.

The example uses Simulink to design and test a pipelined divide cordic algorithm and determine a maximum error threshold for a given number of pipeline stages. In this design, seven pipeline stages are tested.

The design contains there two sample times in Simulink.

- The slow sample time (green) operates in terms of high level transactions agnostic of the lower level algorithm being tested in the DUT. In this example, the transactions consists of three vectors: numerator, denominator, and the results of division from the cordic algorithm.
- The fast sample time (red) operates in terms of scalars. The pipelined DUT samples two scalars at the red sample time, and the resulting cordic division result comes out after seven red sample time hit when the valid out signal is asserted.

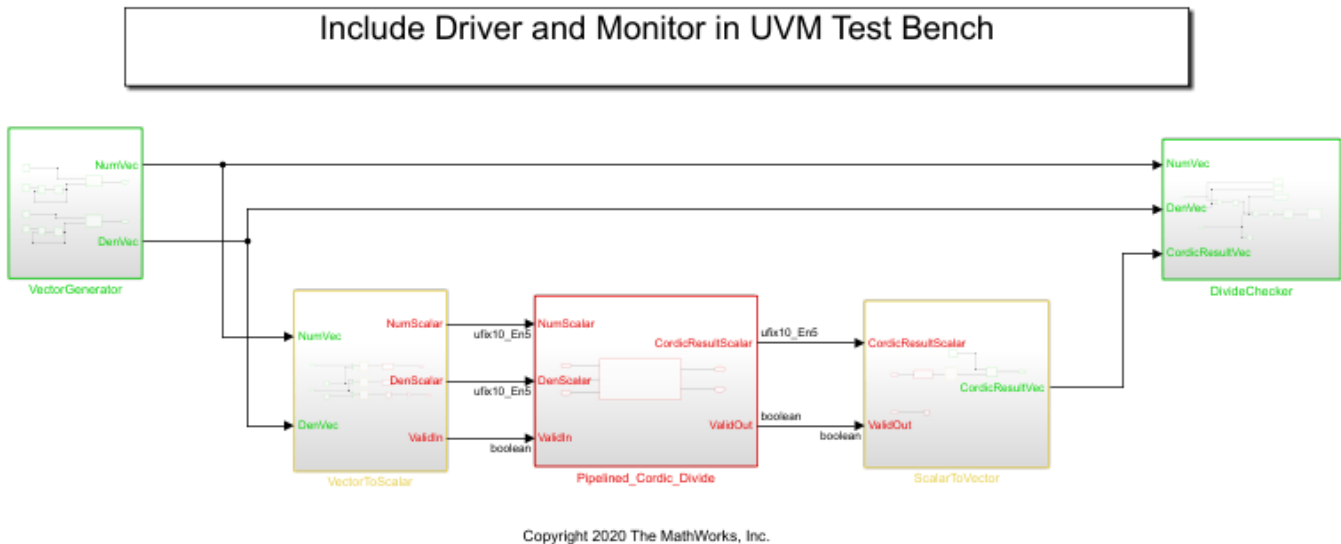
The `VectorGenerator` subsystem generates different numerator and denominator vector sequences. The size and range of the vectors are defined in the `InitFcn` callback under Simulink model explorer.

The `VectorToScalar` subsystem translates the high level transactions (vectors) to a scalar sequence that the DUT is able to understand.

The `ScalarToVector` subsystem assembles the resulting vector from the scalar sequence it receives from the DUT.

The DivideChecker subsystem verifies that the resulting vector contains the correct result of the division within a certain error threshold. If the error is greater than the threshold, then the simulation outputs a warning. In addition the DivideChecker subsystem outputs three MAT files that you can plot to visualize the results.

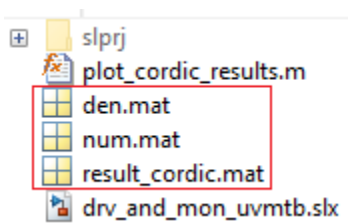
The model is shown below:



Open and simulate the model. Because no assertions are fired, no error threshold violations exist.

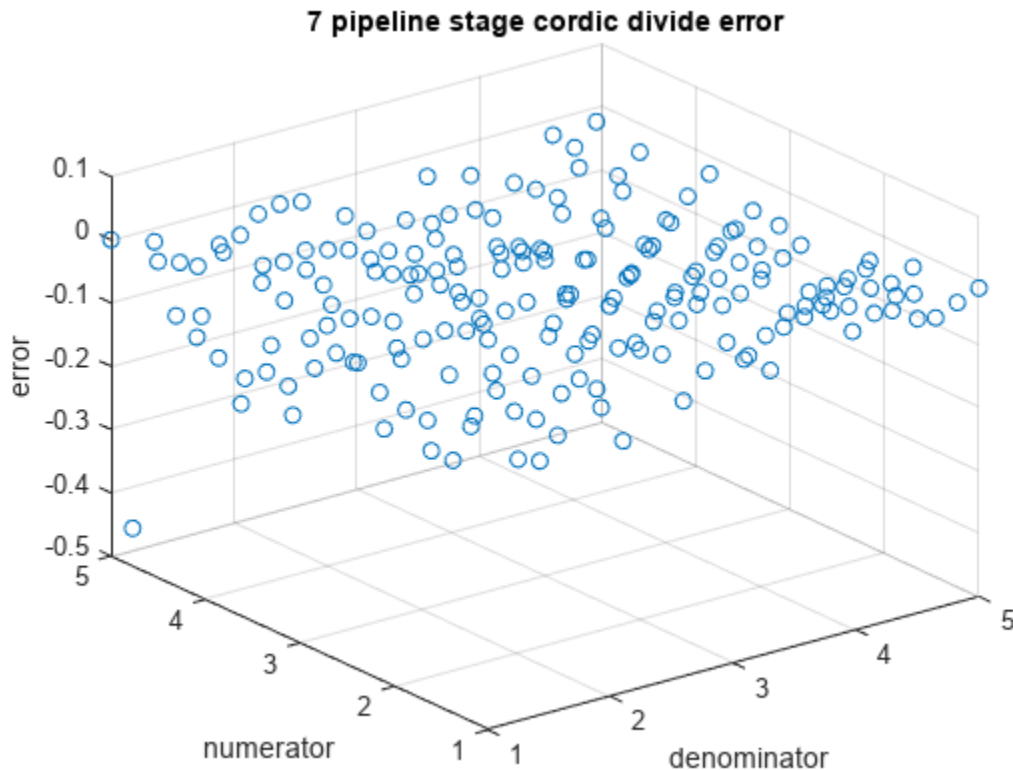
```
open_system('drv_and_mon_uvmtb');
r=sim('drv_and_mon_uvmtb');
```

The simulation writes three MAT files to the disk: num.mat, den.mat, and result_cordic.mat. The files contain the denominators, numerators, and results of the cordic division respectively.



A MATLAB function is provided to generate a 3-D scatter plot from the MAT files. The resulting graph helps visualize the error and further confirms that the error is less than the threshold.

```
plot_cordic_results('num.mat', 'den.mat', 'result_cordic.mat');
```



Generate UVM Test Bench with Driver and Monitor

Use the `uvmbuild` function to export your design to a UVM environment and to specify the Simulink subsystems that you want to map to the UVM driver, monitor, or both.

The UVM driver and monitor enable you to separate between frame-based and scalar-based domains. When translated to UVM framework, the frame-based domain is mapped to UVM transactions, and the scalar-based domain is mapped to the time-aware UVM components (driver, dut, and monitor). Standard component definitions separate the pieces of the environment by their role in the simulation. For this example:

- VectorGenerator subsystem is mapped to the UVM sequence.
- VectorToScalar subsystem is mapped to the UVM driver.
- Pipelined_Cordic_Divide subsystem is mapped to the DPI DUT SystemVerilog module.
- ScalarToVector subsystem is mapped to the UVM monitor.
- DivideChecker subsystem is mapped to the UVM scoreboard.

To generate a UVM test bench out of the Simulink design, execute this code.

```
sequence = 'drv_and_mon_uvmtb/VectorGenerator';
driver = 'drv_and_mon_uvmtb/VectorToScalar';
dpi_dut = 'drv_and_mon_uvmtb/Pipelined_Cordic_Divide';
monitor = 'drv_and_mon_uvmtb/ScalarToVector';
scoreboard = 'drv_and_mon_uvmtb/DivideChecker';
uvmbuild(dpi_dut,sequence,scoreboard,'Driver',driver,'Monitor',monitor);
```

```

### Starting DPI subsystem generation for UVM test bench
### Starting build procedure for model: Pipelined_Cordic_Divide
### Starting SystemVerilog DPI Component Generation
### Generating DPI H Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating DPI C Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating UVM module package C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier
### Generating SystemVerilog module C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverif
### Generating makefiles for: Pipelined_Cordic_Divide_dpi
### Invoking make to build the DPI Shared Library
### Successful completion of build procedure for model: Pipelined_Cordic_Divide

```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
Pipelined_Cordic_Divide	Code generated and compiled.	Code generation information file does not

1 of 1 models built (0 models already up to date)

Build duration: 0h 0m 47.574s

```

### Starting build procedure for model: VectorGenerator
### Starting SystemVerilog DPI Component Generation
### Generating DPI H Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating DPI C Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating UVM module package C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier
### Generating SystemVerilog module C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverif
### Generating makefiles for: VectorGenerator_dpi
### Invoking make to build the DPI Shared Library
### Successful completion of build procedure for model: VectorGenerator

```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
VectorGenerator	Code generated and compiled.	Code generation information file does not exist.

1 of 1 models built (0 models already up to date)

Build duration: 0h 0m 20.695s

```

### Starting build procedure for model: VectorToScalar
### Starting SystemVerilog DPI Component Generation
### Generating DPI H Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating DPI C Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating UVM module package C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier
### Generating SystemVerilog module C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverif
### Generating makefiles for: VectorToScalar_dpi
### Invoking make to build the DPI Shared Library
### Successful completion of build procedure for model: VectorToScalar

```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
VectorToScalar	Code generated and compiled.	Code generation information file does not exist.

```

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.911s
### Starting build procedure for model: ScalarToVector
### Starting SystemVerilog DPI Component Generation
### Generating DPI H Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating DPI C Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating UVM module package C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier
### Generating SystemVerilog module C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverif
### Generating makefiles for: ScalarToVector_dpi
### Invoking make to build the DPI Shared Library
### Successful completion of build procedure for model: ScalarToVector

```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
ScalarToVector	Code generated and compiled.	Code generation information file does not exist.

```

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.089s
### Starting build procedure for model: DivideChecker
### Starting SystemVerilog DPI Component Generation
### Generating DPI H Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating DPI C Wrapper C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier-ex4
### Generating UVM module package C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverifier
### Generating SystemVerilog module C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\21\tpedb50693\hdlverif
### Generating makefiles for: DivideChecker_dpi
### Invoking make to build the DPI Shared Library
### Successful completion of build procedure for model: DivideChecker

```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
DivideChecker	Code generated and compiled.	Code generation information file does not exist.

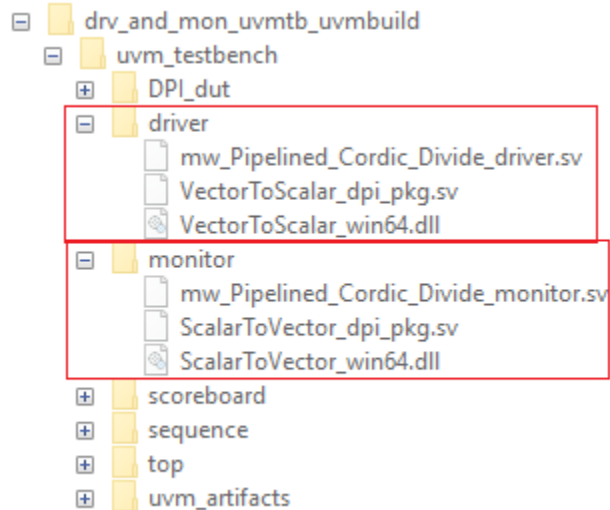
```

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 20.436s
### Starting UVM test bench generation for model: drv_and_mon_uvmtb
### Generating UVM transaction object ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/scoreboard/mw_Pipelined_Cordic_Divide
### Generating UVM interface ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/uvm_artifacts/mw_Pipelined_Cordic_Divide
### Generating UVM sequence ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/sequence/mw_Pipelined_Cordic_Divide
### Generating UVM sequencer ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/sequence/mw_Pipelined_Cordic_Divide
### Generating UVM sequence transaction ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/sequence/mw_Pipelined_Cordic_Divide
### Generating UVM driver ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/driver/mw_Pipelined_Cordic_Divide
### Generating UVM monitor ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/monitor/mw_Pipelined_Cordic_Divide
### Generating UVM input monitor ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/uvm_artifacts/mw_Pipelined_Cordic_Divide
### Generating UVM agent ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/uvm_artifacts/mw_Pipelined_Cordic_Divide
### Generating UVM scoreboard ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/scoreboard/mw_Pipelined_Cordic_Divide
### Generating UVM scoreboard configuration object ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/scoreboard/mw_Pipelined_Cordic_Divide
### Generating UVM environment ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/uvm_artifacts/mw_Pipelined_Cordic_Divide
### Generating UVM test ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/uvm_artifacts/mw_Pipelined_Cordic_Divide
### Generating UVM top ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/top/mw_Pipelined_Cordic_Divide

```

```
### Generating UVM test package ./uvm_build/drv_and_mon_uvmtb_uvm_testbench/top/drv_and_mon_uvmtb_uvm_testbench
### Generating UVM test bench simulation script for Mentor Graphics QuestaSim/Modelsim ./uvm_build
```

The UVM driver and monitor are generated in separate directories as shown in this figure.



Run UVM Test Bench

The generated UVM test bench can be executed by going to UVM test bench top module directory *drv_and_mon_uvmtb_uvmbuild\uvm_testbench\top* and executing one of the generated HDL simulator scripts. For this example, ModelSim/QuestaSim® is used. The commands to execute are shown in this figure.

```
>> %Simulate the UVM test bench using ModelSim/Questasim (console)
>> cd drv_and_mon_uvmtb_uvmbuild\uvm_testbench\top\
>> !vsim -c -do run_tb_mq.do
```

To confirm that the UVM simulation matches the Simulink test bench, check this requirements.

- No error threshold violations are thrown. This is verified by looking at the UVM simulation log and checking that no UVM errors exist. The UVM Simulation log is shown in this figure.

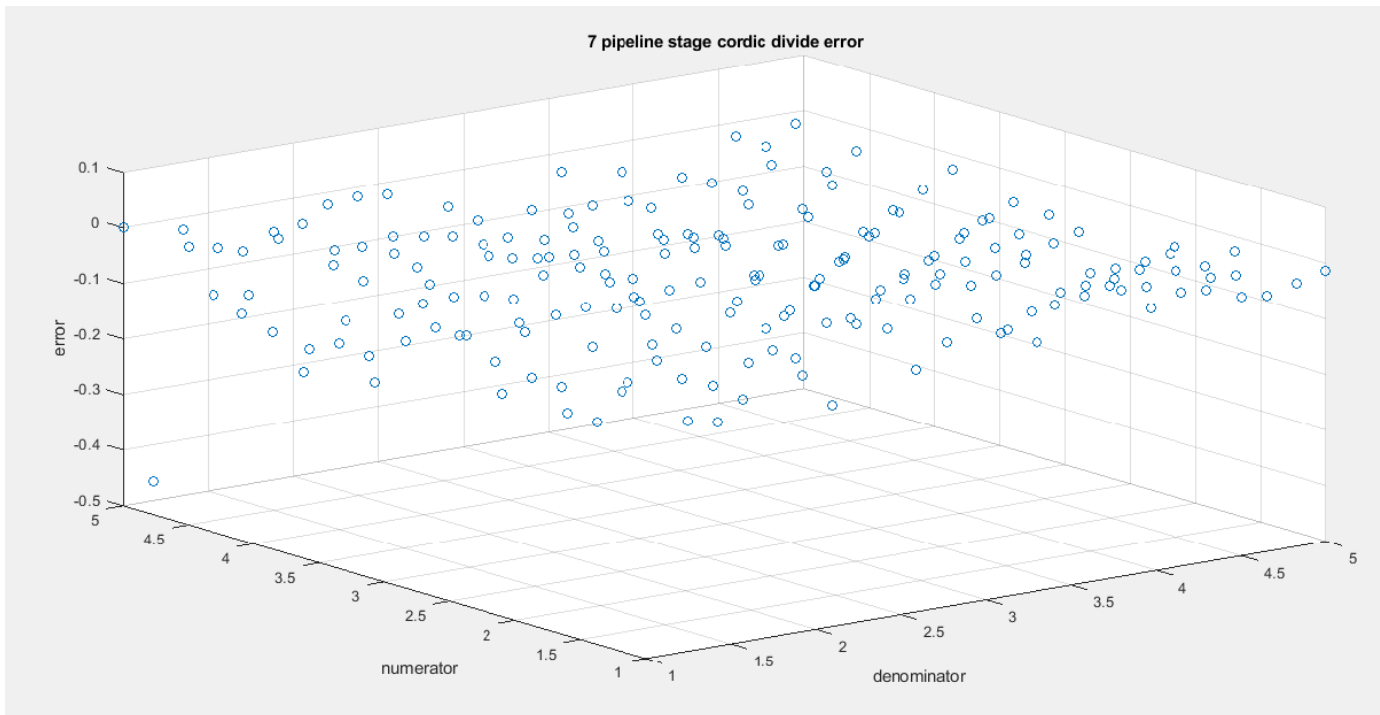
```
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(277) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.3
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(278) @ 0: reporter [Questa UVM]  questa_uvm::init(+struct)
# UVM_INFO @ 0: reporter [RNTST] Running test mw_Pipelined_Cordic_Divide_test...
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 3845: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 4
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
# ** Report counts by id
# [Questa UVM] 2
# [RNTST] 1
# [TEST_DONE] 1
# ** Note: $finish : //mathworks/hub/3rdparty/R2020b/5487679/share/Questasim/Win/win64/./verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
# Time: 3845 ns Iteration: 61 Instance: /mw_Pipelined_Cordic_Divide_top
```

- The UVM scoreboard outputs three MAT files (*num.mat*, *den.mat*, and *result_cordic.mat*) at the end of the UVM simulation. Visually, the 3-D scattered plot generated from the Simulink design

must match the plot from the UVM simulation. The plot from the UVM simulation MAT files can be generated using this command:

```
>> plot_cordic_results('drv_and_mon_uvmtb_uvmbuild\uvmtb\top\num.mat',...
    'drv_and_mon_uvmtb_uvmbuild\uvmtb\top\den.mat',...
    'drv_and_mon_uvmtb_uvmbuild\uvmtb\top\result_cordic.mat');
```

The resulting UVM simulation 3-D scatter plot is shown in this figure. The plot matches the Simulink design plot shown in previous sections.



Conclusion

Keeping the frame-based and scalar-based behavioral modeling separate in Simulink and UVM framework results in these benefits.

- The domain separation enables the modeling of frame-based transaction processing at a higher abstraction level without having to introduce lower level conversions that are dependent on the algorithm being developed.
- The VectorGenerator and DivideChecker subsystems can be reused for different Simulink or UVM projects that have a different algorithm but require the same high level transaction generation and checking.
- The domain separation enables the DUT algorithm to be multirate.
- The domain separation enables the high level transaction to operate in a different data type than the DUT. In this example, the DUT has port data types of `ufx10_en5`, but the VectorGenerator and DivideChecker subsystems are of `double` data type.
- If you have Simulink Test® you can optionally map the conversion blocks in the Simulink Test Harness to the UVM driver and monitor. The domain separation enables a more streamlined UVM test bench generation workflow.

The `uvmbuild` function enables you to separate the frame-based and scalar domains by providing additional optional name-value pairs to specify Simulink subsystems that are mapped to a UVM driver, monitor, or both.

Verify Viterbi Decoder Using MATLAB System Object and HDL Simulator

This example shows you how to use MATLAB® System objects and an HDL simulator to cosimulate a Viterbi decoder implemented in VHDL.

The HDL Verifier™ product lets you verify the design implemented in Verilog or VHDL using MATLAB System objects. The product allows you to cosimulate the HDL code with MATLAB and verify the model against the HDL implementation. This example uses MATLAB System objects and following HDL simulators to cosimulate a Viterbi decoder.

- Vivado® Simulator from Xilinx®
- ModelSim® or Questa® from Mentor Graphics®
- Xcelium® from Cadence®

Set Simulation Parameters and Instantiate Communication System Objects

If you are using Xcelium, set simulator variable to Xcelium.

```
Simulator = 'Xcelium';
```

If you are using ModelSim/QuartaSim, set simulator variable to ModelSim.

```
Simulator = 'ModelSim';
```

If you are using Vivado simulator The HDL cosimulation System object for Vivado simulator can be created by using the **Cosimulation Wizard** tool only. For more information on the **Cosimulation Wizard** tool, see Cosimulation Wizard.

The following code sets up the simulation parameters and instantiates the System objects that represent the channel encoder, BPSK modulator, AWGN channel, BPSK demodulator, and error rate calculator. Those objects comprise the system around the Viterbi decoder and can be thought of as the test bed for the Viterbi HDL implementation.

```
EsNo = 0; % Energy per symbol to noise power spectrum density ratio in dB
```

```
FrameSize = 1024; % Number of bits in each frame
```

Convolution Encoder

```
hConEnc = comm.ConvolutionalEncoder;
```

BPSK Modulator

```
hMod = comm.BPSKModulator;
```

AWGN channel

```
hChan = comm.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (Es/No)', ...
    'SamplesPerSymbol',1, ...
    'EsNo',EsNo);
```

BPSK demodulator

```
hDemod = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio', ...
                             'Variance',0.5*10^(-EsNo/10));
```

Error Rate Calculator

```
hError = comm.ErrorRate('ComputationDelay',100,'ReceiveDelay',58);
```

Instantiate Cosimulation System Object and Launch HDL Simulator

For ModelSim or Xcelium

1. The HDL cosimulation System object for ModelSim or Xcelium can be created by using either the **Cosimulation Wizard** tool or `hdlcosim` function. This example uses the `hdlcosim` function to generate the HDL cosimulation System Object. The System object represents the HDL implementation of the Viterbi decoder in this simulation system. The object's interface is common for all simulators. As a convenience to avoid writing some HDL testbench code, we generate waveforms for the clocks and resets using simulator-specific Tcl code.

```
hDec = hdlcosim('InputSignals', {'/viterbi_block/In1','/viterbi_block/In2'}, ...
              'OutputSignals', {'/viterbi_block/Out1'}, ...
              'OutputSigned', false, ...
              'OutputFractionLengths', 0, ...
              'TCLPostSimulationCommand', 'echo "done";', ...
              'PreRunTime', {10,'ns'}, ...
              'Connection', {'Shared'}, ...
              'SampleTime', {10,'ns'});

switch Simulator
case 'ModelSim'
    hDec.TCLPreSimulationCommand = ...
        'force /viterbi_block/clk_enable 1 0; force /viterbi_block/clk 0 0 ns, 1 5 ns -repeat 1000';
case 'Xcelium'
    hDec.TCLPreSimulationCommand = ...
        'force :clk B"0" -after 0ns B"1" -after 5ns -repeat 10ns; force reset B"1" -after 0ns B"0" -after 5ns -repeat 10ns';
end
```

2. The `vsim` and `nclaunch` command launches HDL simulator. The launched HDL simulator session compiles the HDL design and loads the HDL simulation. You are ready to perform cosimulation when the HDL simulation is fully loaded in simulator.

```
disp('Launching HDL simulator...');
switch Simulator
case 'ModelSim'
    vsim('tclstart',viterbi_cosimulation_tclcmds('vsimmatlabsysobj'));
case 'Xcelium'
    nclaunch('tclstart',viterbi_cosimulation_tclcmds('hdlsimmatlabsysobj'));
end
Timeout=30;
processid = pingHdlSim(Timeout);
Check if HDL simulator is ready for Cosimulation.
assert(ischar(processid),['Timeout: HDL simulator took more than ', num2str(Timeout), ' seconds to launch.']);
disp('...Simulator is ready for cosimulation.');
```

For Vivado Simulator

1. To generate the HDL cosimulation System object by using the **Cosimulation Wizard**, follow upto step 6 mentioned in the “Cosimulation Wizard for MATLAB System Object” on page 32-37 .

On the **Input/Output Ports** page, perform the following steps.

- a. Set the **clk** Port Name to Clock.
- b. Set the **reset** and **clk_enable** Port Name to Reset.
- c. Set the **In1** and **In2** Port Name to Input.
- d. Set the **ce_out** Port Name to Unused.
- e. Set the Out1 Port Name to Output.
- f. Click **Next**.

On the **Output Port Details**, perform the following steps.

- a. Set **Sample Time** to 10.
- b. Set **Sign** to Unsigned.
- c. Set fraction length to 0.
- d. Click **Next**.

On the **Clock/Reset Details** page perform the following steps.

- a. Set the clock period to 10.
- b. Set the **reset Initial Value** to 1 and **Duration** to 8.
- c. Set the **clk_enable Initial Value** to 0 and **Duration** to 1.
- d. Click **Next**.

On the **Start Time Alignment** page, perform the following steps.

- a. Set the Pre Run Time by setting **HDL time to start cosimulation** to 0.
- b. Click **Update Diagram**.
- c. Click **Next**.

On the **System Obj. Generation** page set **HDL simulator sampling period** to 10 and click **Finish**.

2. Generated System object script should look as follows.

```
xsiData = createXsiData( ...
    'design', 'xsim.dir/design/xsimk', ...
    'lang', 'vhdl', ...
    'prec', 'lps', ...
    'types', {'Logic' 'Logic' 'Logic' }, ...
    'dims', {3 3 1} ...
);

obj = hdlcosim( ...
    'HDL Simulator', 'Vivado Simulator', ...
    'InputSignals', {'/viterbi_block/In1', '/viterbi_block/In2'}, ...
```

```

'OutputSignals', {'/viterbi_block/Out1'}, ...
'OutputSigned', [false], ...
'OutputDataTypes', {'fixedpoint'}, ...
'OutputFractionLengths', [0], ...
'ClockResetSignals', {'/viterbi_block/clk' '/viterbi_block/reset' '/viterbi_block/clk_enable'
'ClockResetTypes', {'Active Rising Edge Clock' 'Step 1 to 0' 'Step 0 to 1' }, ...
'ClockResetTimes', {{10,'ps'} {8,'ps'} {1,'ps'} }, ...
'PreRunTime', {0,'ps'}, ...
'SampleTime', {10,'ps'}, ...
'XSIData', xsiData ...
);

```

3. Assign the System object to a new variable `hDec` by using this command in MATLAB.

```
hDec = hdlcosim_viterbi_block;
```

Run Cosimulation

This example simulates the BPSK communication system in MATLAB incorporating the Viterbi decoder HDL implementation via the cosimulation System object. This section of the code calls the processing loop to process the data frame-by-frame with 1024 bits in each data frame.

```

for counter = 1:20480/FrameSize
    data          = randi([0 1],FrameSize,1);
    encodedData   = hConEnc(data);
    modSignal     = hMod(encodedData);
    receivedSignal = hChan(modSignal);
    demodSignalSD = hDemod(receivedSignal);
    quantizedValue = fi(4-demodSignalSD,0,3,0);
    input1        = quantizedValue(1:2:2*FrameSize);
    input2        = quantizedValue(2:2:2*FrameSize);

    receivedBits  = hDec(input1, input2);
    errors        = hError(data, double(receivedBits));
end

```

Display Bit Error Rate

The bit error rate is displayed for the Viterbi decoder.

```
sprintf('Bit Error Rate is %d\n',errors(1))
```

Destroy Cosimulation System Object to Release HDL Simulator

The HDL simulator is unblocked when the HDL cosimulation System object is destroyed in MATLAB. Close the HDL simulator session manually.

```
clear hDec;
```

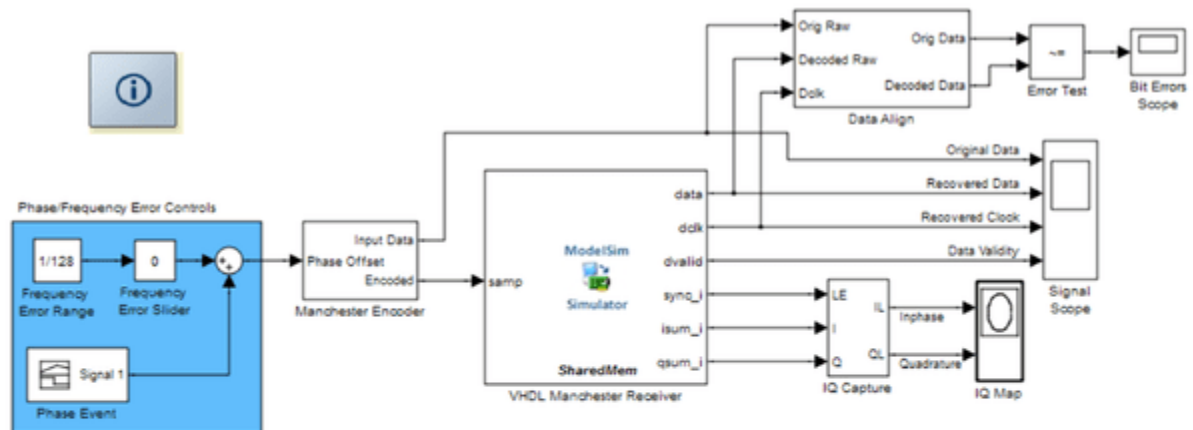
See Also

- `hdlverifier.HDLCosimulation`
- `hdlverifier.VivadoHDLCosimulation`
- Cosimulation Wizard

Batch Mode Cosimulation of Manchester Receiver

This example illustrates using MATLAB® to start HDL simulator in batch mode and performing cosimulation with Simulink® using the HDL Verifier™ HDL Cosimulation block.

1. ModelSim/Questasim



Double-click the commands to the right to start cosimulation.
We execute the following steps in those commands

1. Start ModelSim(R) HDL simulator in batch mode by specifying 'runmode' property of vsim command
2. Wait until HDL simulator is ready for cosimulation by using pingHdlSim command
3. Start cosimulation

```
vsim('tclstart',manchestercmds,...
     'runmode','Batch');
pingHdlSim(100);
sim('manchester_batch',[0 50000]);
% Double-click here to start simulation
```

Cosimulation Startup Command

Copyright 2003-2009 The MathWorks, Inc.

The commands displayed in the illustration above show the following steps:

- The call to the HDL Verifier vsim command starts HDL simulator in batch mode by setting the 'runmode' property to 'Batch'. Issuing vsim also launches the HDL simulator, and additional commands (specified in **manchestercmds.m**) compile the HDL design and load the HDL Verifier HDL cosimulation library.

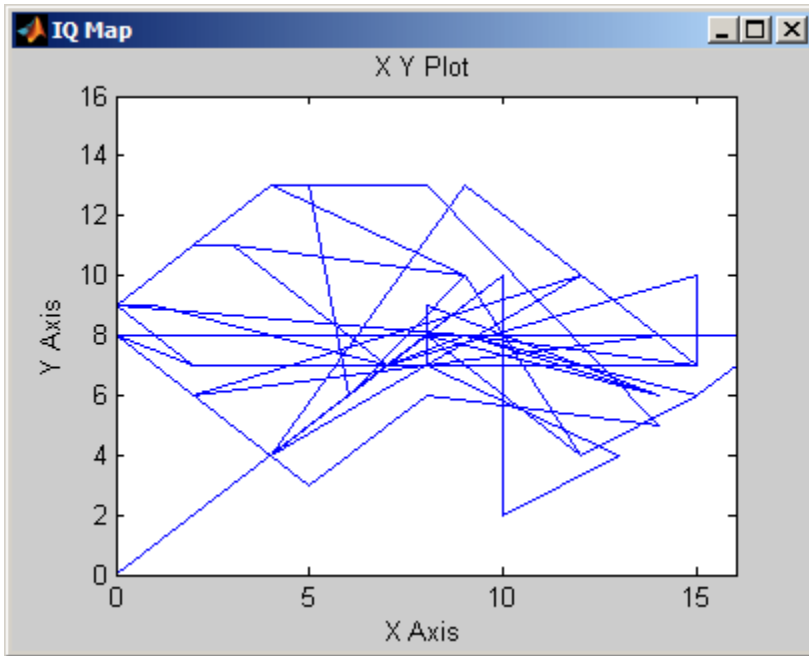
```
vsim('tclstart', manchestercmds, 'runmode', 'Batch');
```

- If you are running on a Linux® machine, the HDL simulator process starts in the background. On Windows®, a new command window opens for HDL simulator. MATLAB and Simulink now wait to begin cosimulation.
- The HDL Verifier **pingHdlSim** command detects if the HDL simulator server is ready for cosimulation. The timeout argument to pingHdlSim specifies that it will wait for 100 seconds for HDL simulator to start. If HDL simulator fails to start within that time, an error is reported.

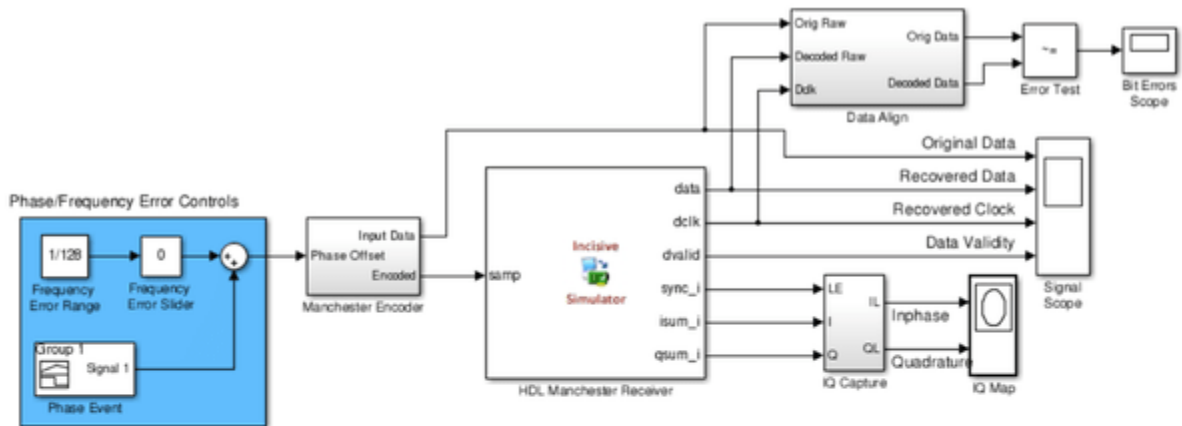
```
pingHdlSim(100);
```

- Simulink begins cosimulation when MATLAB detects (through pingHdlSim) that the HDL simulator server is ready.

```
sim('manchester_batch',[0 50000]);
```



2. Xcelium



Click the commands to the right to start cosimulation.
We execute the following steps in those commands

1. Start Incisive(R) HDL simulator in batch mode by specifying 'runmode' property of nclaunch command
2. Wait until HDL simulator is ready for cosimulation by using pingHdlSim command
3. Start cosimulation

```
nclaunch( ...
'runDir', 'TEMPDIR', ...
'clstart', { ...
[ 'exec ncvtlog -linedebug 'vlogFiles{...} ...
'exec ncelab -access +rwc manchester', ...
'hdlSimulink manchester' }, ...
'runmode', 'Batch');
pingHdlSim(100);
sim('manchester_batch_incisive',[0 50000]);
% Single-click here to start cosimulation
```

Cosimulator Startup Command

Copyright 2006-2010 The MathWorks, Inc.

The commands displayed in the illustration above show the following steps:

- The call to the HDL Verifier `nclaunch` command starts HDL simulator in batch mode by setting the 'runmode' property to 'Batch'. Issuing `nclaunch` also compiles and elaborates the HDL design, and loads the HDL Verifier HDL cosimulation library.

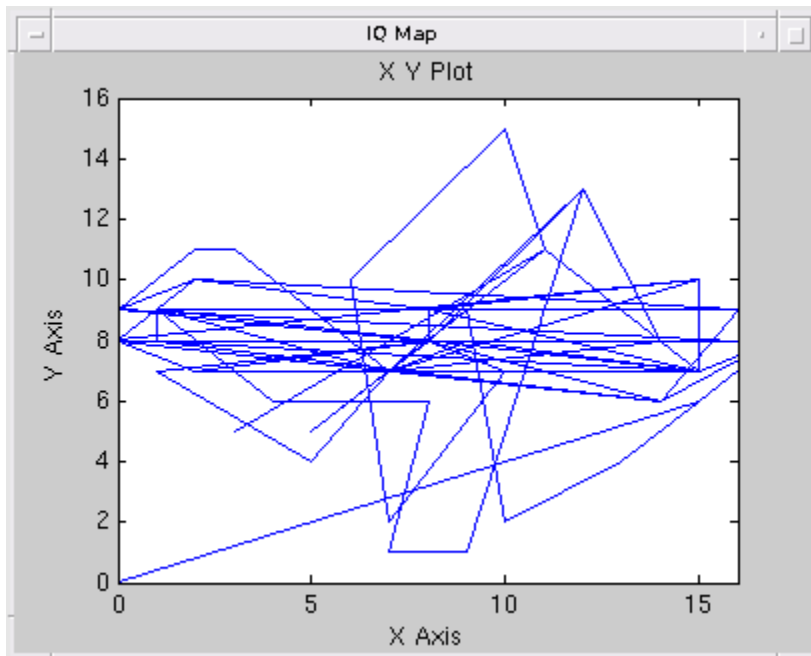
```
nclaunch( ...
    'rundir', 'TEMPDIR', ...
    'tclstart', { ...
        ['exec xmvlog -64bit -linedebug ' vlogFiles{:}], ...
        'exec xmelab -64bit -access +rwc manchester', ...
        'hdlsimulink manchester' }, ...
    'runmode', 'Batch');
```

- The HDL simulator process starts in the background. MATLAB and Simulink now wait to begin cosimulation.
- The HDL Verifier `pingHdlSim` command detects if the HDL simulator server is ready for cosimulation. The timeout argument to `pingHdlSim` specifies that it will wait for 100 seconds for HDL simulator to start. If HDL simulator fails to start within that time, an error is reported.

```
pingHdlSim(100);
```

- Simulink begins cosimulation when MATLAB detects (through `pingHdlSim`) that the HDL simulator server is ready.

```
sim('manchester_batch_incisive',[0 50000]);
```



- When the cosimulation completes, the HDL simulator exits automatically.

Sobel Edge Detection Algorithm with Computer Vision Toolbox

This example shows and explains the "**Top-Down**" design methodology that is applied to Sobel Edge Detection algorithm. The Sobel Edge Detection algorithm is a popular yet simple edge detection algorithm and is the focus of this example. With this example you will learn:

- How Simulink® allows you to design a digital signal processing (DSP) algorithm at a system level.
- How to elaborate on the design to make it realizable in hardware.
- How to co-simulate hand-written HDL code (corresponding to your Simulink model) with ModelSim® or Cadence® Xcelium® in the Simulink environment.

This approach is an example of **Model Based Design**. To learn more about Model Based Design refer to <https://www.mathworks.com/solutions/model-based-design.html>

You need the following products to run all the models in the example:

- MATLAB®
- Simulink
- Fixed-Point Designer™
- HDL Verifier™
- Computer Vision Toolbox™
- DSP System Toolbox™
- ModelSim SE or PE. or Cadence Xcelium

The Executable Specification of Sobel Edge Detection Algorithm

As designs become larger and more complicated, it has become necessary to describe a design at a high level. This high level description not only enables the designer to run simulations faster, it can also be used throughout the development process for verification. This resulting process allows developers to identify bugs early on and avoid costly bug discovery towards the end of development. This high-level design is usually done by system engineers.

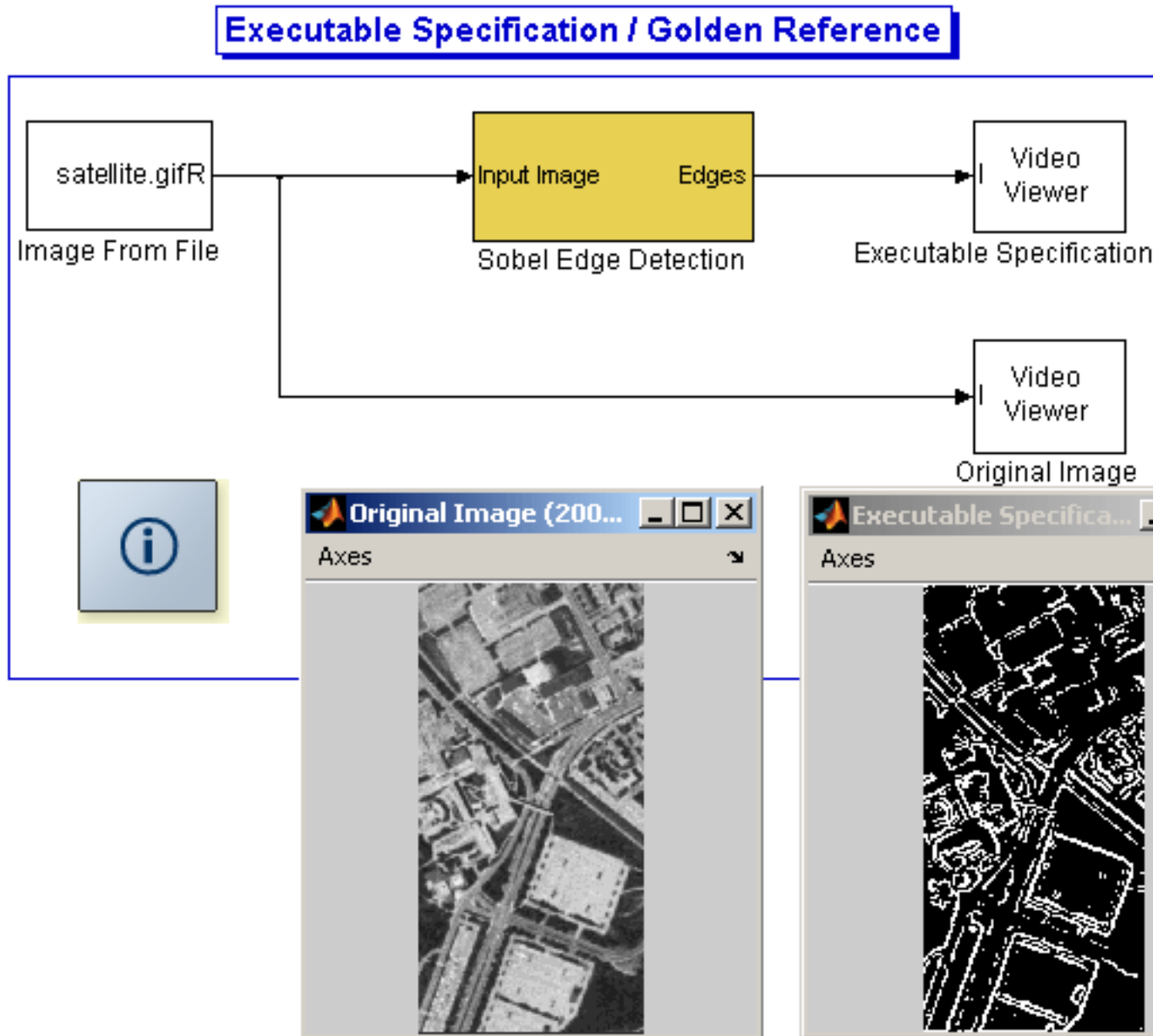
To implement a DSP algorithm on hardware such as ASIC or an FPGA, a system level engineer first designs the algorithm and verifies that the algorithm satisfies the project requirements. This design later becomes the golden reference for the engineers responsible for taking the algorithm to the hardware.

In this example, the Sobel Edge Detection algorithm has been implemented in Simulink. Open the executable model and double click on the "Sobel Edge Detection" block to learn how the algorithm is implemented in Simulink. When you double-click on the Sobel Edge Detection block, you can see that the algorithm is comprised of two 2D filters, one to calculate the gradient in the column direction (top filter) and one to calculate the gradient in the row direction (bottom filter). Both filters use a 3x3 kernel.

This Simulink model serves as the specification for the rest of the development path. It is an **executable specification**, meaning you can readily execute this model in the Simulink environment.

This example uses a satellite image as the input to the edge detection algorithm. This image serves as an input test vector and is used throughout the example. If engineers who are responsible for the hardware implementation of the algorithm work in the Simulink environment as well, there is no need

for extra overhead in porting the test vectors into different applications or creating test harnesses that are prone to human errors. The test harness that is used in the executable specification is used throughout the example.

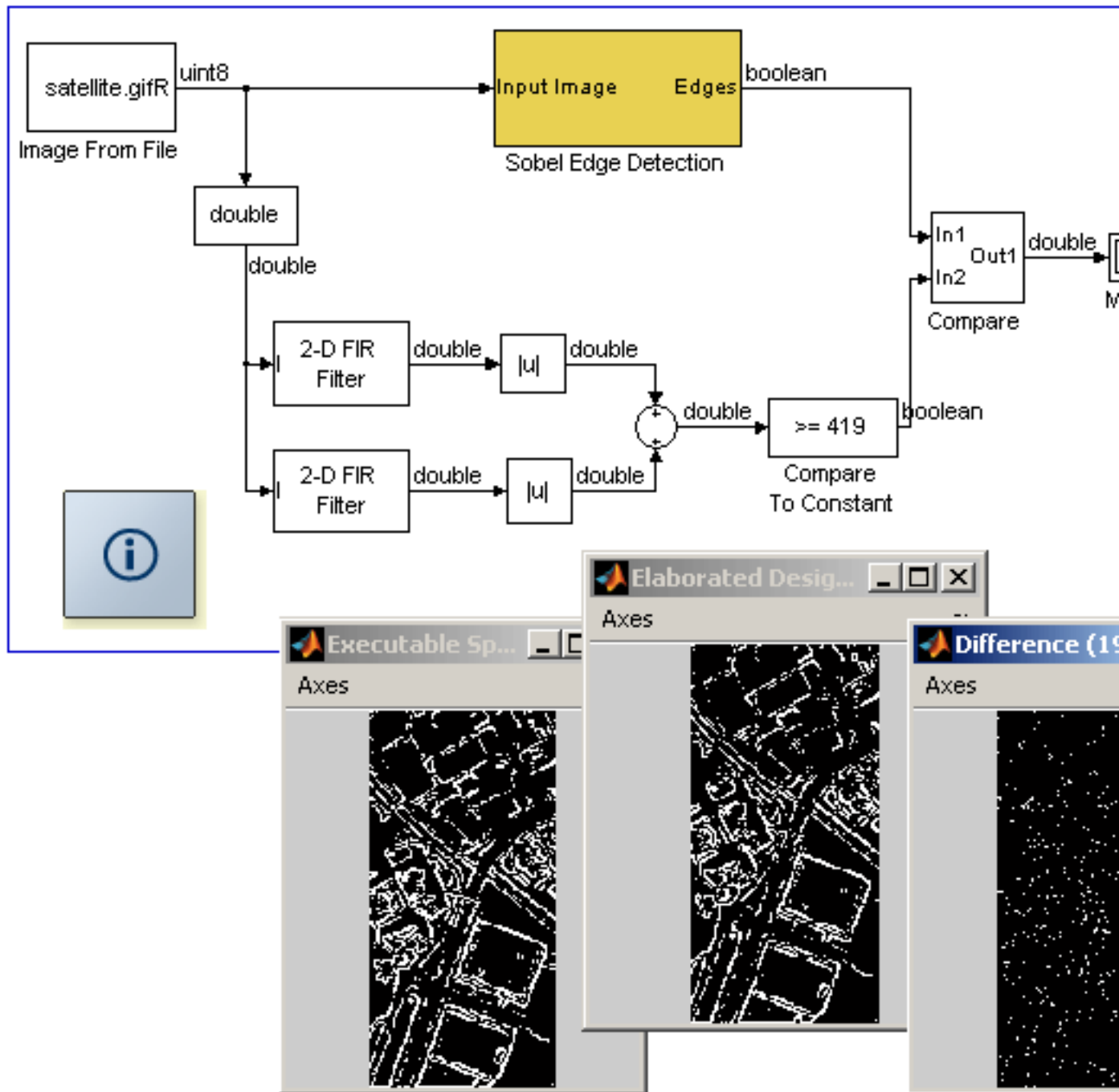


Design and Verification of the Algorithm for Implementation

When designing the executable model, the system engineer generally does not keep the implementation details in mind but rather implements the algorithm to match the behavioral requirements. Once the system engineer submits the executable specification to the development team, the development team may need to make modifications to the executable specification to fit the design into a real-time system that may have limited resources such as memory or processing power.

In this example, the developers may decide to eliminate the square operations and replace them with the absolute value operation for more efficient hardware implementation. This will cause a difference between the cosimulation result and the golden reference but for the sake of this example, we assume that the difference is acceptable. Open the edge detection design model. You can see the same test vectors as the previous model are still being used. The result can be easily verified against the golden reference. A numerical display shows the mean difference between the golden reference and the new design.

Design and Verification



Fixed-Point Design

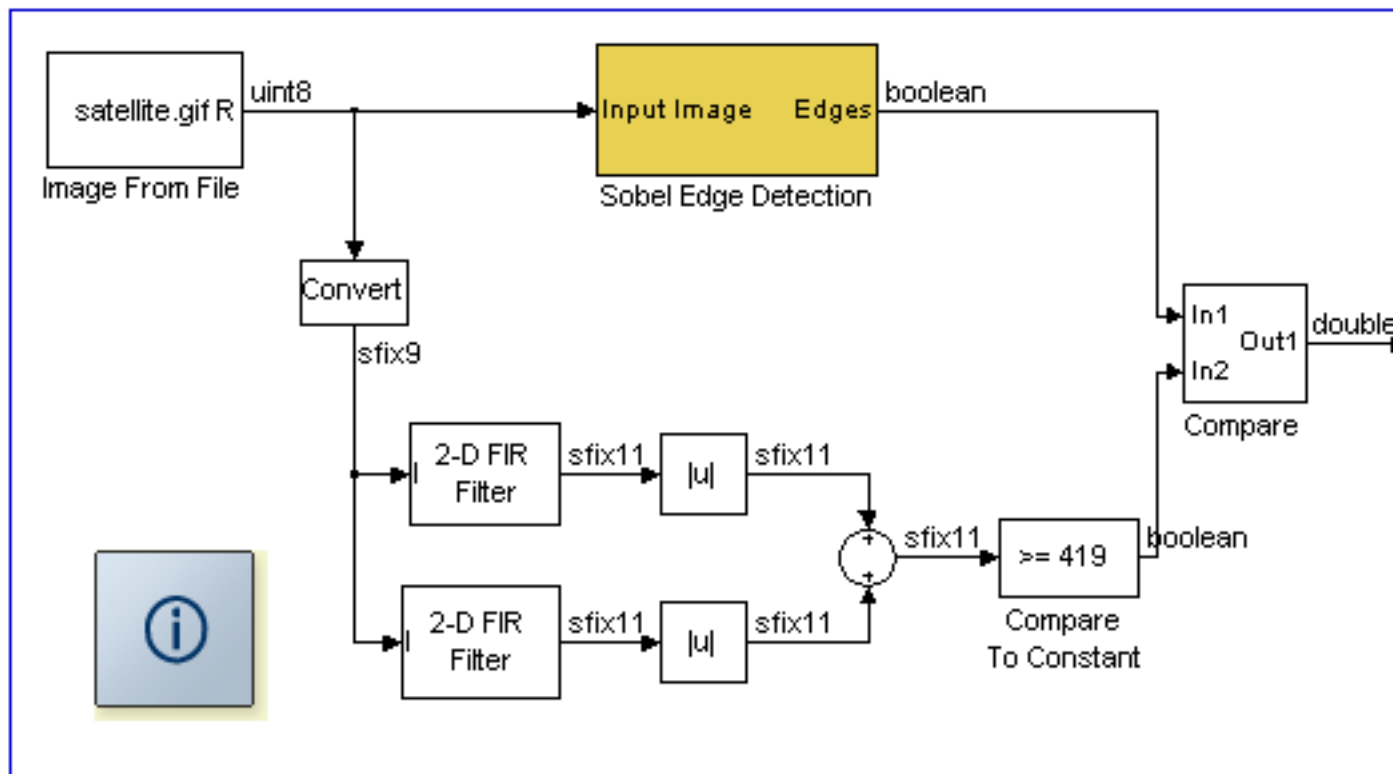
Since our ultimate goal is to implement the algorithm in an FPGA or an ASIC, we must convert our double precision design to a fixed-point design. This can be done easily using Simulink. We use the double precision model we developed in the previous section to directly develop a fixed-point model

without introducing any new blocks. Simulink allows us to determine the number of bits and scaling for data as well as mathematical operations, and provides a great environment for analyzing the fixed-point operation of a system.

In this fixed-point design, the input to the filters is a signed 9-bit integer and the outputs of the filters are signed 11-bit integers. If you double-click on each computational block, such as the filters or the sum blocks, you can see that the developer can easily tune the bit width and scaling related to the internal computations of that block. This gives huge leverage to the designer to compromise between matching the output of the golden reference while using the least number of bits necessary to save area on the device.

Open the fixed-point model and examine how the fixed-point is implemented by double clicking on the computational blocks, such as the 2D filters or the addition block, and looking at the corresponding fixed-point panel.

Fixed-Point Design and Verification



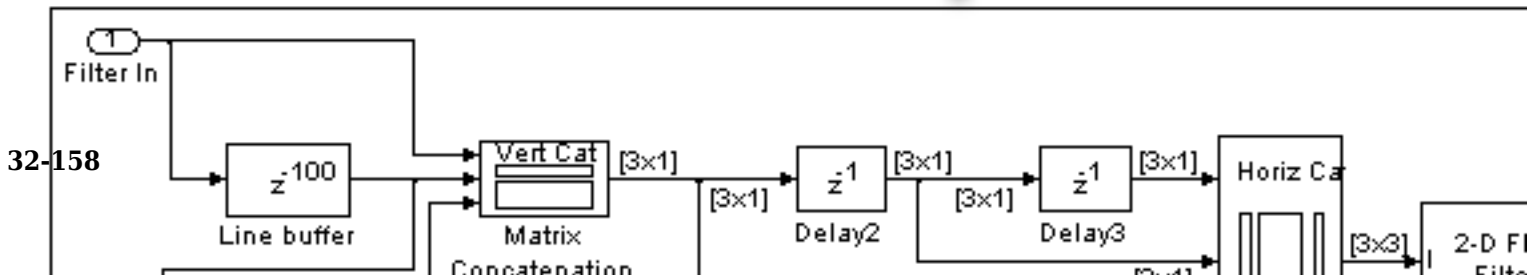
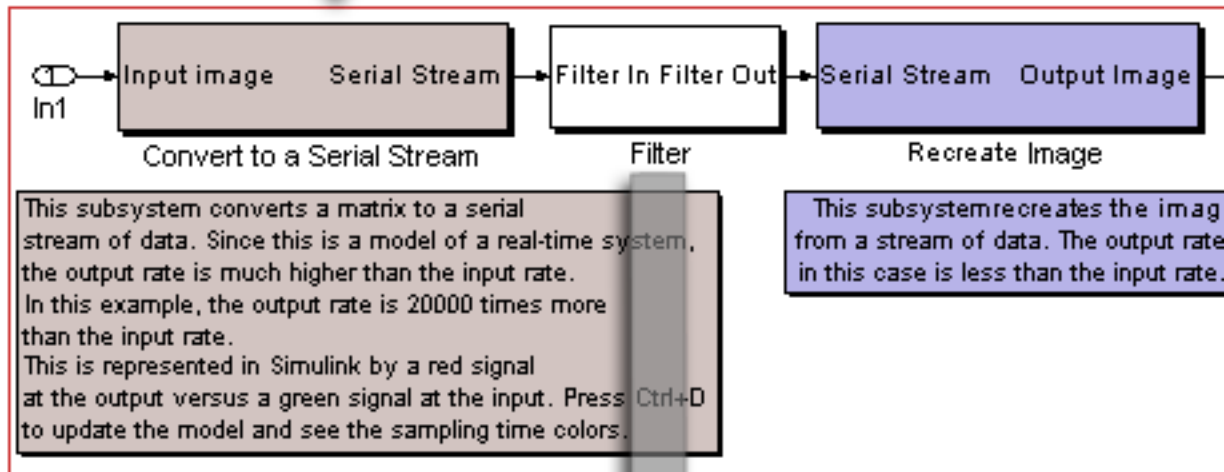
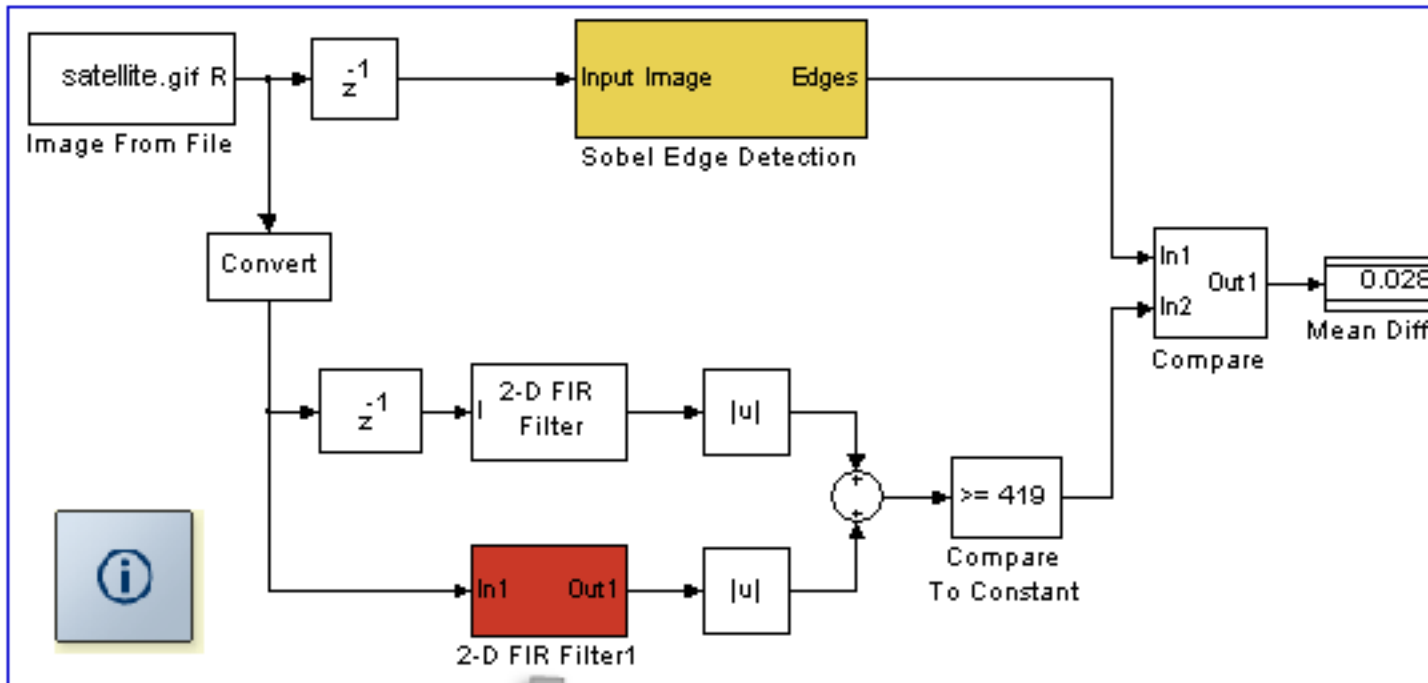
Elaboration of the Design

In our example, the input to the edge detection algorithm has been a two-dimensional image of size 200x100. In a real-time system, the input is most likely not a matrix but a serial stream of data; for example, this serial stream of data can be generated by a Charge-Coupled Device (CCD). Therefore, we need to modify the structure of the design such that the edge detection algorithm accepts and performs 2D filtering on a serial stream of data.

To this extent, we first serialize the input image. Then we perform the 2D filtering on this serial data. We later de-serialize the stream of data to be able to compare the output to the golden reference. To see how this is done please refer to edge detection elaboration model.

This operation is done only for the bottom filter. As expected, the new design is still producing the same exact results as before. Two delay elements have been added to compensate for the buffering in the serializer block. This design also showcases the multi-rate capability of Simulink. The output of the serializer block is 20000 times higher in sampling rate relative to the input to that block.

Elaboration and Verification



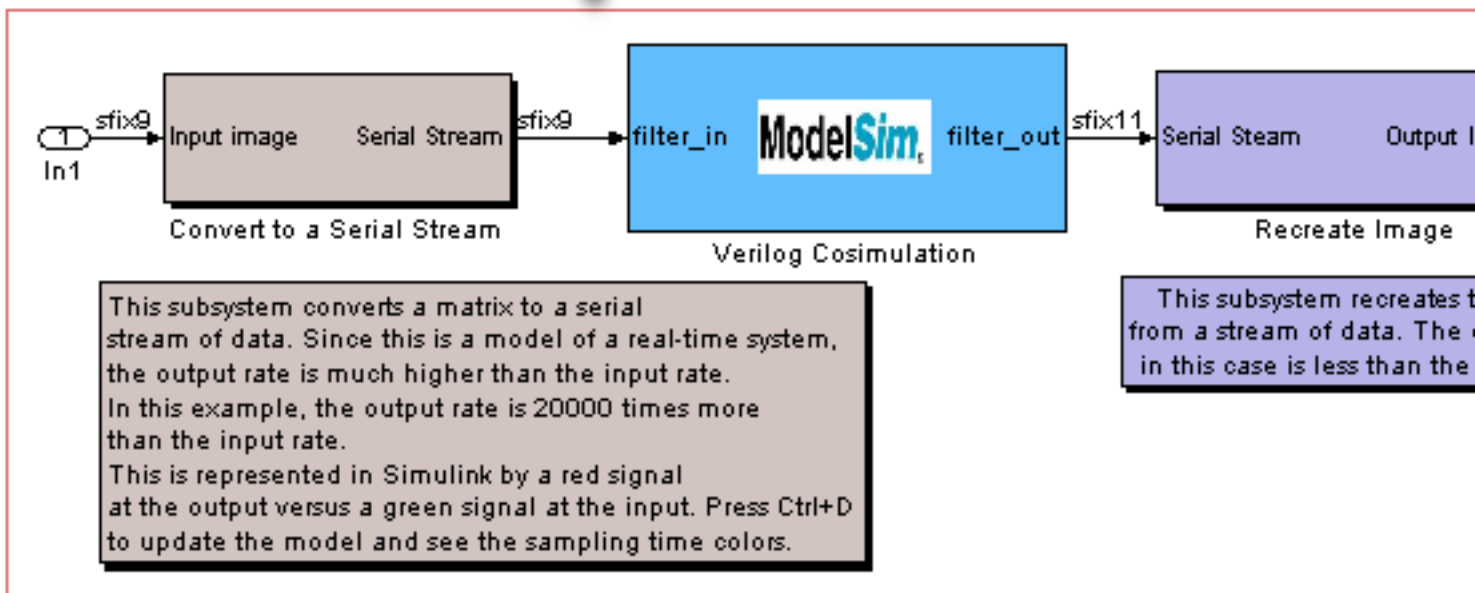
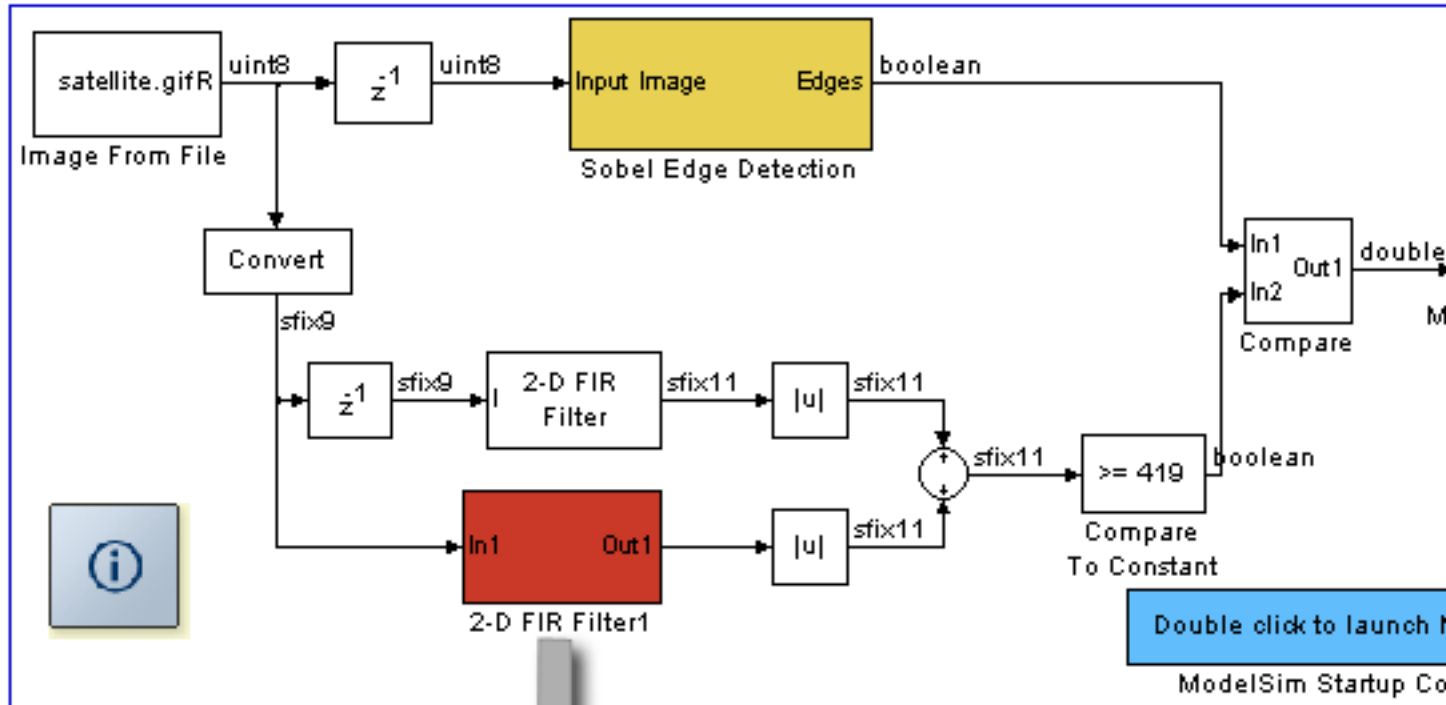
Simulink® Cosimulation

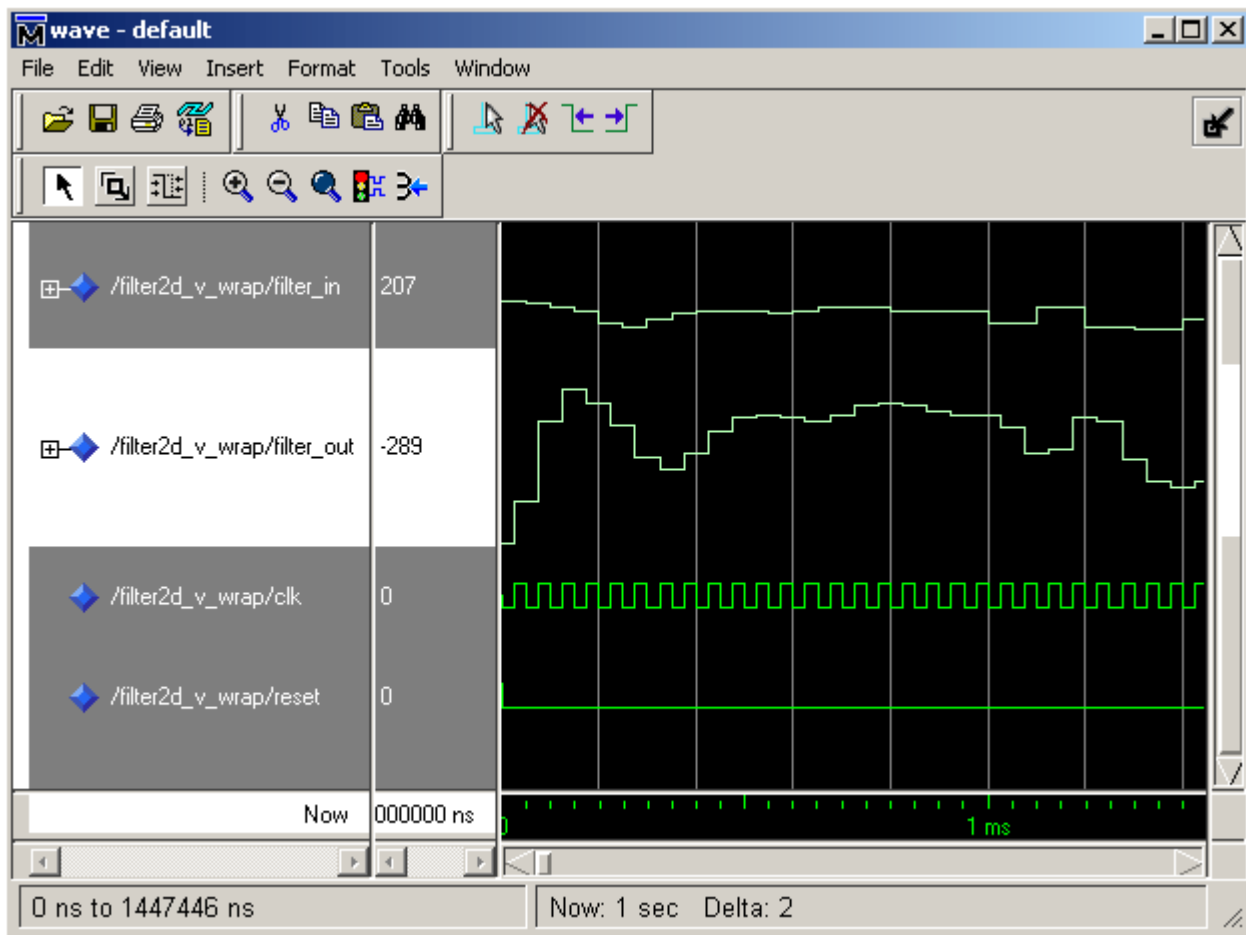
The model from the previous section can be passed to the HDL designer who can use the 2D filter designed in the last section to write the corresponding VHDL or Verilog code. Once the code is written, the HDL designer can use HDL Verifier™ to simulate the HDL design in the Simulink environment using ModelSim or Xcelium, and compare the output of the HDL design to the output of the executable specification. Note that in this process, there is no need for generating an HDL test bench. The Simulink model feeds the input test vector to ModelSim or Xcelium through HDL Verifier and extracts the data from ModelSim or Xcelium back to the Simulink environment. The HDL designer can readily verify whether the HDL code runs in accordance with the specifications.

- ModelSim/Questasim

Refer to VHDL model or Verilog model to see how HDL Verifier is used to cosimulate Simulink and ModelSim. The last figure shows a snap shot of the signals displayed in ModelSim.

Verilog Co-simulation, Simulink and ModelSim



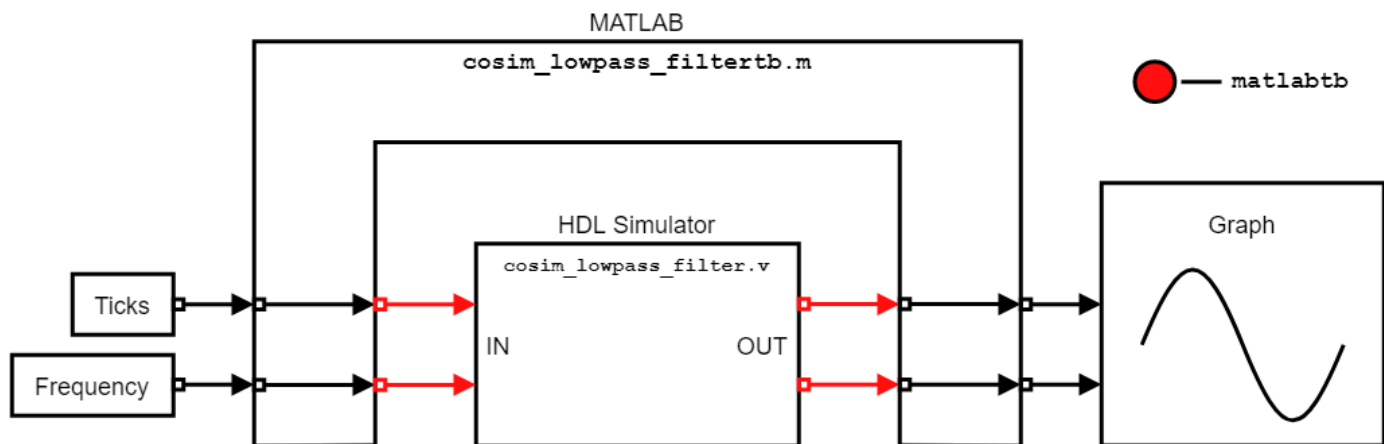


- Xcelium Refer to Verilog model to see how HDL Verifier software is used to cosimulate Simulink and the Xcelium platform.

Cosimulation for Testing Filter Component Using MATLAB Test Bench

This example shows how to run either Mentor Graphics® ModelSim®/Questasim® or Cadence® Xcelium® with MATLAB® in batch mode to test an HDL component using a MATLAB test bench, by using the HDL Verifier™ function `matlabtb`. The function `matlabtb` facilitates this testing by feeding MATLAB-generated input data into the HDL component and feeding the output from the component back into MATLAB. When you generate complex waveforms in HDL, using `matlabtb` enables you to spend more time running tests than writing test waveforms. `matlabtb` also reads the results, so the testing process can leverage MATLAB to apply data transformations to the output before plotting. For more information on writing test benches for cosimulation, see “Create a MATLAB Test Bench” on page 2-2.

In this example, MATLAB generates a sine wave with an adjustable frequency, which is fed into a low-pass filter implemented in Verilog®. `matlabtb` passes the filter outputs to MATLAB for plotting. This process of testing is possible with MATLAB and with the HDL simulator in command-line mode. This diagram shows an overview of the simulation.



This example considers the point of view of an HDL developer, so the main emphasis of the example is on using the HDL simulator and command-line terminal to perform testing and using MATLAB as an aid for supplying inputs and interpreting outputs.

Overview of Design and Script Files

This example uses two types of files: design files and script files.

Design Files

- The file `cosim_lowpass_filter.v` contains a Filter Design HDL Coder™ Toolbox generated filter with a sample time of 10 ns (a sample rate of 100 MHz), a passband $F_{pass} = 20$ MHz and a stopband $F_{stop} = 25$ MHz.
- The file `cosim_lowpass_filtertb.m` contains the MATLAB test bench for this lowpass filter. The test bench uses variables `ticks` and `frequency` to create a sinusoid to pass into the filter before interpreting and plotting the outputs.

Script Files

- The file `qcommands_cosim_w.tcl` contains the commands that are sent into ModelSim/Questasim on Windows® for cosimulation to occur.
- The file `qcommands_cosim_l.tcl` contains the commands that are sent into ModelSim/Questasim on Linux® for cosimulation to occur.
- The file `xcelium_cosim.tcl` contains the commands for opening the Xcelium simulator so that MATLAB can communicate with the simulator.
- The file `xmcommands_cosim.tcl` contains the commands that are sent into Xcelium for cosimulation to occur.

In this example, because you run the HDL simulator executables and MATLAB from a terminal, you must include them on the system path for this example to work. This example does not use the Filter Design HDL Coder Toolbox product.

Adjust Script Files

This example uses the script files provided to link MATLAB and the HDL simulator. The script uses two commands: a command that invokes the HDL simulator and a command that uses a MATLAB shared library to create the connection for cosimulation. For this example to run, you need to change the command that invokes the HDL simulator, because the default command relies on an absolute path to the MATLAB shared library. Adjust the script files based on the simulator and operating system.

ModelSim/Questasim

- Windows: This figure shows the two required commands for cosimulation with Modelsim/Questasim on Windows. The file `qcommands_cosim_w.tcl` contains these two commands. Line 3 is the call to open ModelSim/Questasim, where the `-foreign` argument supplies the path to the MATLAB shared library, such that Modelsim/Questasim loads in the MATLAB shared library. Line 4 is the call to a function specified in the shared library that connects the two programs together.

```
3 vsim -c cosim_lowpass_filter -foreign {matlabclient
  {matlabroot\toolbox\edalink\extensions\modelsim\windows64\liblfmhdlc_gcc450vc12.dll} }
4 matlabtb cosim_lowpass_filter 10ns -repeat 10ns -mfunc cosim_lowpass_filtertb.m
```

For this example to work, you must change line 3 in `qcommands_cosim_w.tcl`. In this command, swap `matlabroot` with the installation location of MATLAB so that the shared library can be located and loaded.

- Linux: This figure shows the two required commands for cosimulation with Modelsim/Questasim on Linux. The file `qcommands_cosim_l.tcl` contains these two commands. Line 3 is the call to open ModelSim/Questasim, where the `-foreign` argument supplies the path to the MATLAB shared library, such that Modelsim/Questasim loads in the MATLAB shared library. Line 4 is the call to a function specified in the shared library that connects the two programs together.

```
3 vsim -c cosim_lowpass_filter -foreign {matlabclient {matlabroot/toolbox/edalink/
  extensions/modelsim/linux64/liblfmhdlc_gcc450.so} }
4 matlabtb cosim_lowpass_filter 10ns -repeat 10ns -mfunc cosim_lowpass_filtertb.m
```

For this example to work, you must change line 3 in `qcommands_cosim_l.tcl`. In this command, swap `matlabroot` with the installation location of MATLAB so that the shared library can be located and loaded.

Xcelium

This figure shows lines 3-6 of `xcelium_cosim.tcl`, which contain the call to open the Xcelium simulator. Line 4 tells Xcelium to run a command to link MATLAB and the simulator from the MATLAB shared library. Line 6 tells Xcelium to load the MATLAB shared library.

```
3 exec <@stdin >@stdout xmsim -batch cosim_lowpass_filter -64bit \  
4 -input {@call matlabtb cosim_lowpass_filter 10ns -repeat 10ns -mfunc cosim_lowpass_filtertb} \  
5 -input {xmcommands_cosim.tcl}\  
6 -loadcfc {matlabroot/toolbox/edalink/extensions/incisive/linux64/liblfihdlc_gcc41.so:matlabclient}
```

For this example to work, you must change line 6 in `xcelium_cosim.tcl`. In this command, swap `matlabroot` with the installation location of MATLAB so that the shared library can be located and loaded.

Run Cosimulation

To start the cosimulation, open MATLAB and run the HDL Link MATLAB server for communication between the HDL simulator and MATLAB. To complete these actions, use these commands based on the operating system.

- Windows: `matlab -nodesktop -r "hdldaemon"`
- Linux: `xterm -e "matlab -nodesktop -r "hdldaemon"" &`

This command opens MATLAB in a separate terminal, in headless mode, while the HDL Link MATLAB server starts. The terminal that is running MATLAB displays this message:

```
"HDLDaemon shared memory server is running with 0 connections"
```

After this message appears, enter these commands at the MATLAB command prompt.

```
global frequency;  
global ticks;  
frequency = 1e6;  
ticks = 0;
```

These commands define and initialize the two variables that control the testbench file `cosim_lowpass_filtertb.m`. The variable `frequency` represents the frequency of the input wave sent by the test bench, and the variable `ticks` indicates how much simulator time has passed since the simulation started and is used to plot the output.

Next, start the cosimulation. Enter the applicable command (based on the simulator and operating system) into a system terminal.

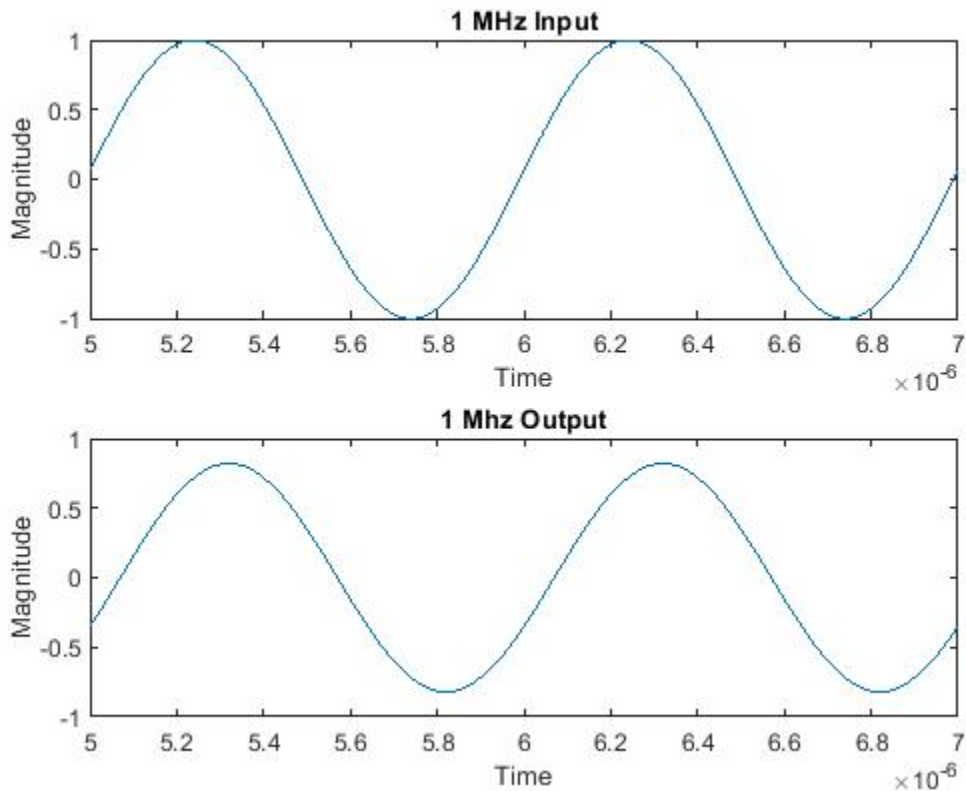
ModelSim/Questasim

- Windows: `vsim -c -do qcommands_cosim_w.tcl`
- Linux: `vsim -c -do qcommands_cosim_l.tcl`

Xcelium

- Linux: `tclsh xcelium_cosim.tcl`

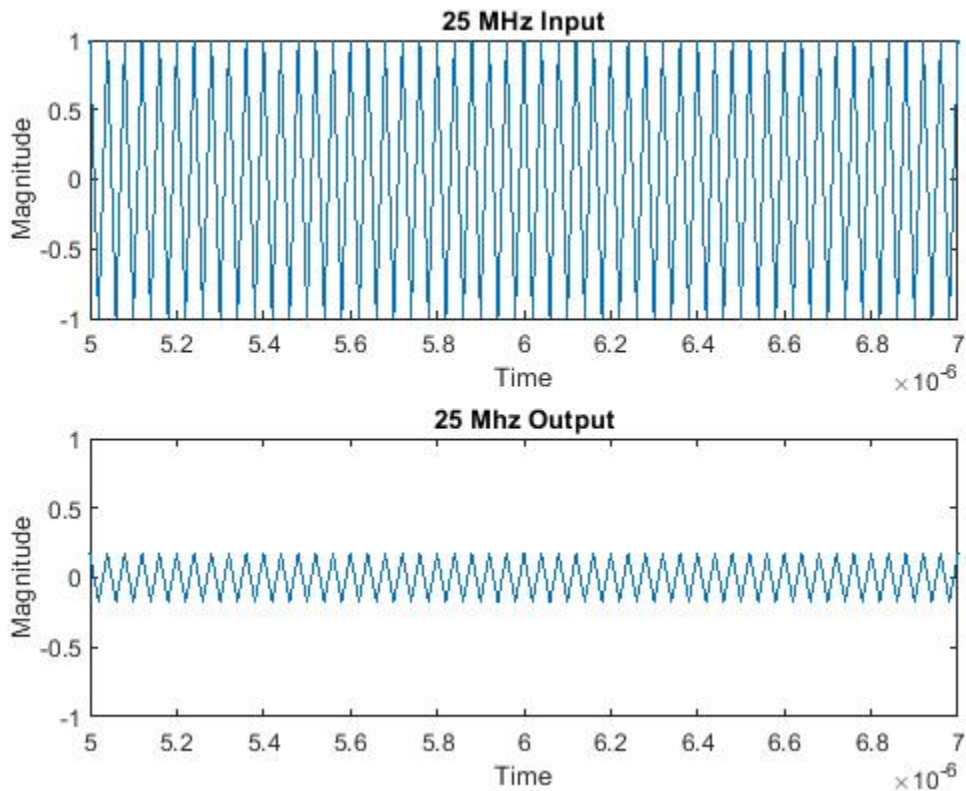
After the cosimulation is complete, MATLAB should plot the filtered and unfiltered results, similar to these figures.



Adjust Parameters and Rerun Cosimulation

To run the simulation again with a different input from the MATLAB test bench, you do not need to restart the HDL simulator or MATLAB itself. Instead, adjust the two variables defined earlier: `ticks` and `frequency`. After you change `frequency` to your desired value and reset `ticks` to 0, run the simulation from within the HDL terminal window to produce the new results.

For example, change `frequency` to `25e6` and reset `ticks` to 0 using the MATLAB terminal. Then, in the terminal running the HDL simulator, enter this command: `run 10000ns`. These figures show the results of these settings.



Summary

This example shows how to use ModelSim/Questasim or Xcelium with MATLAB to update and run simulations by adjusting the characteristics of the input wave. MATLAB defines and controls the characteristics of the input, and displays the output, even when you run MATLAB in a terminal. Using MATLAB in conjunction with an HDL simulator enables you to quickly run through many tests.

In this example, only the variable frequency controls the input. However, you can use MATLAB to generate a large number of complex test waveforms before using these waveforms in cosimulation tests and to display the results. Using MATLAB to create these test waveforms streamlines the design process, by cutting down the time needed to construct and add in various test waveforms, and removing the time needed to process the simulation results into visual data.

See Also

`matlabtb` | `hdldaemon` | `vsim` | `nclaunch`

Related Topics

- "Set Up for HDL Cosimulation" on page 10-2
- "Create a MATLAB Test Bench" on page 2-2
- "Set Up Cosimulation Test Bench" on page 2-13
- "Run MATLAB-HDL Cosimulation" on page 1-4

Copyright 2009-2021 The MathWorks, Inc.

Manchester Receiver Using Multiple Cosimulation Blocks

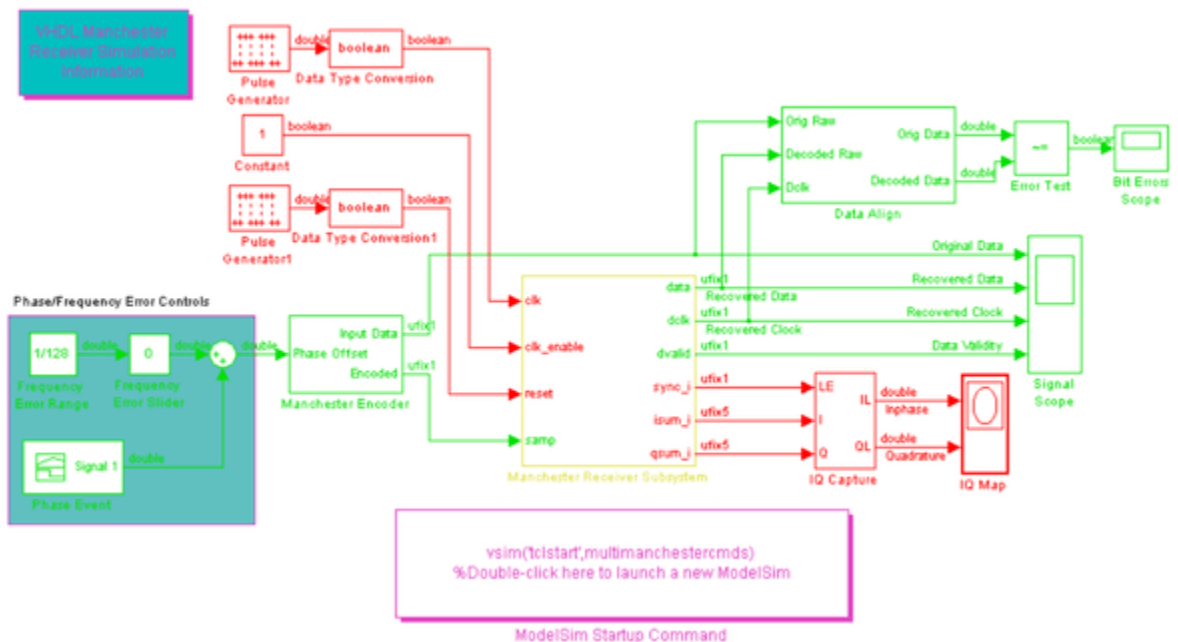
This example simulates a digital receiver of Manchester encoded data. Manchester encoding is a simple modulation scheme that converts baseband digital data to an encoded waveform with no DC component. The most widely known application of this technique is Ethernet.

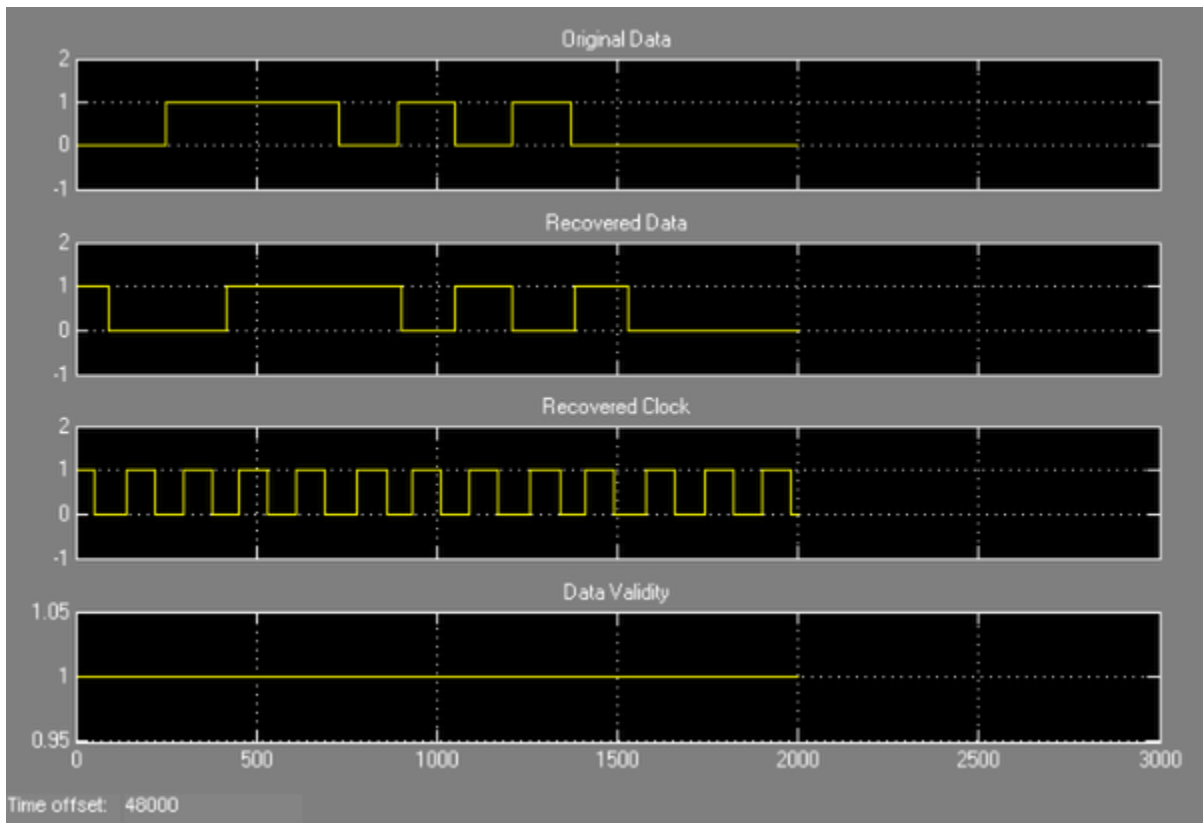
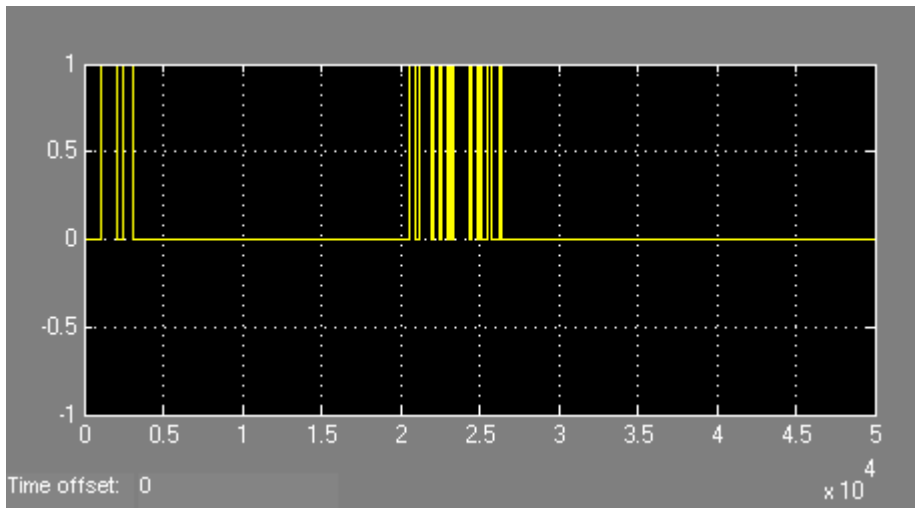
The receiver is implemented in HDL. The receiver uses a simple DLL (delay lock loop) clock recovery mechanism, which requires multiple cycles to lock with the incoming data stream. The performance of the receiver is explored by applying phase and frequency errors to a randomly generated stream of bits that is encoded using a simple MATLAB® function: manchesterencoder().

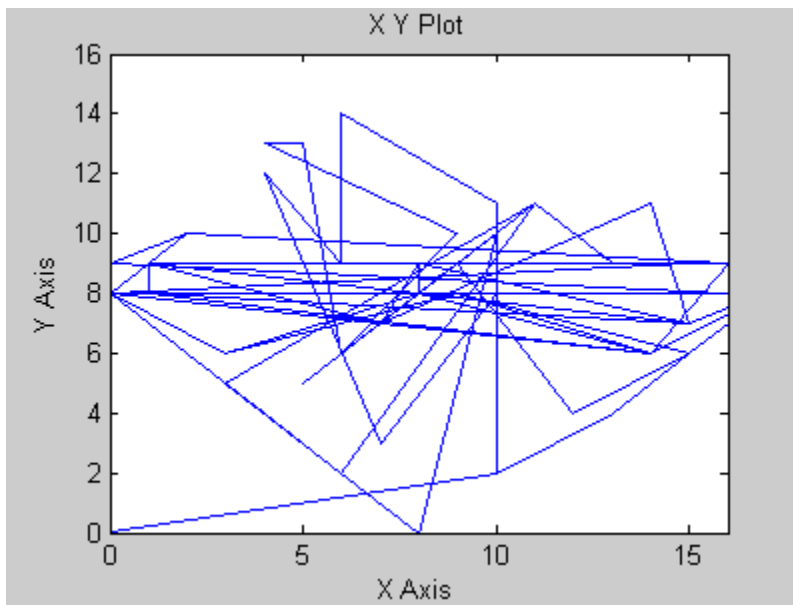
1. ModelSim/Questasim

The VHDL code runs in ModelSim® as three ModelSim VHDL Cosimulation blocks under Manchester Receiver Subsystem labeled State Counter, IQ Converter and Decoder.

Open the multiple-block model and click on the ModelSim Startup Command box to launch ModelSim. Start cosimulation in Simulink by clicking "Run".



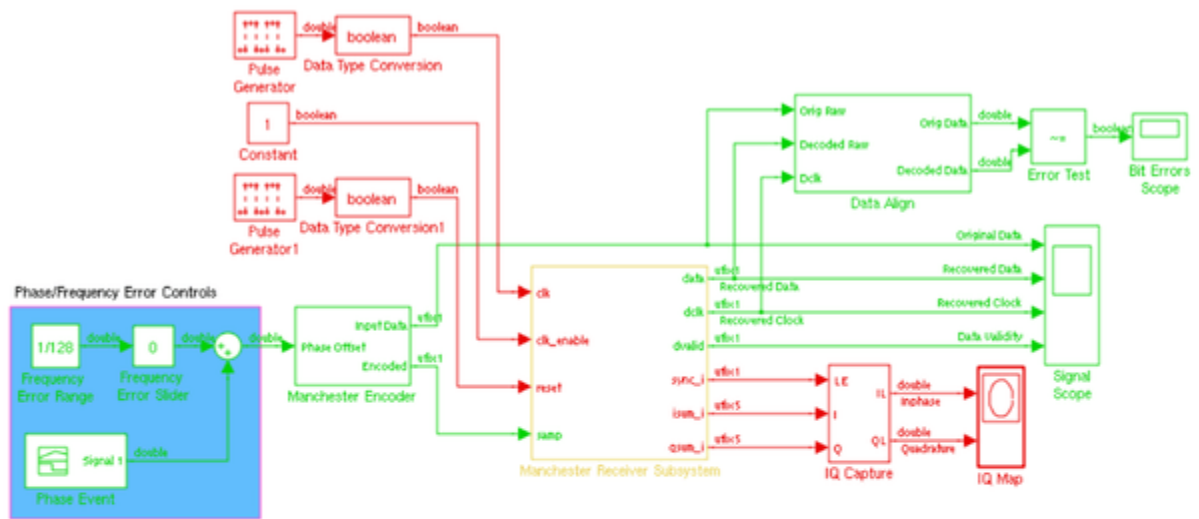




2. Xcelium

The HDL code runs in the Xcelium simulator and its execution is reflected in Simulink as the behaviors of three HDL Cosimulation blocks under the Manchester Receiver Subsystem. They are labeled State Counter, IQ Converter and Decoder.

Open the multiple-block model and follow the steps outlined in the 'Running a Cosimulation' annotation.

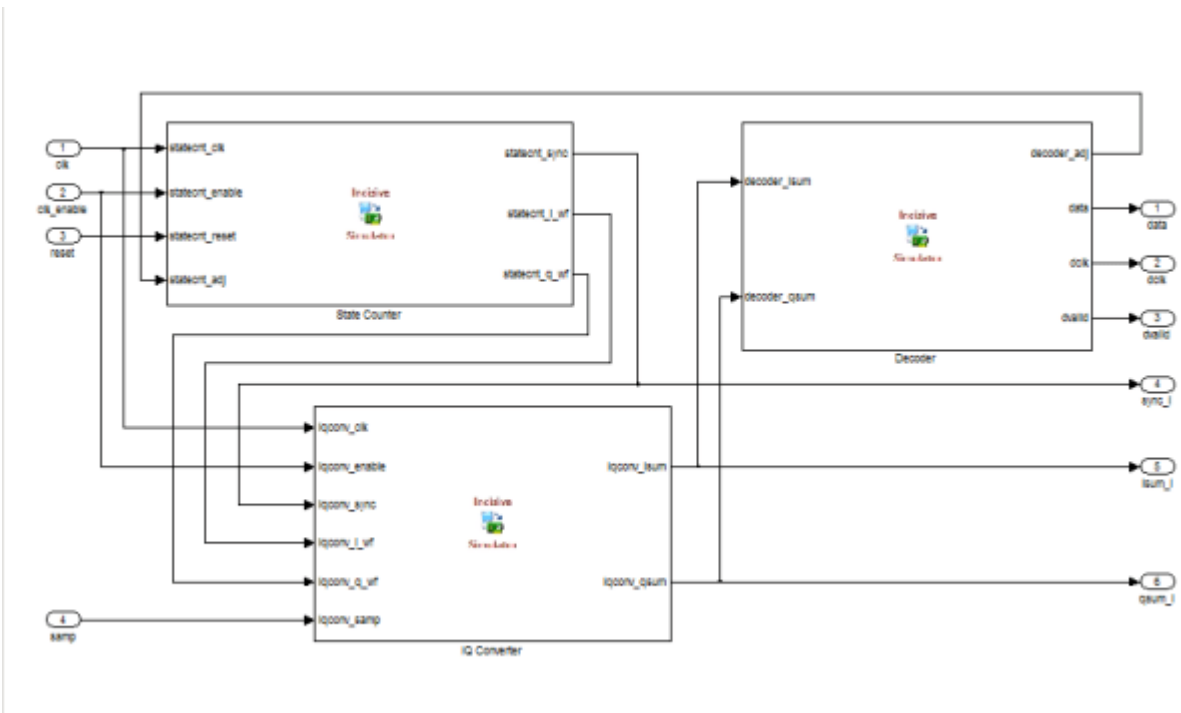


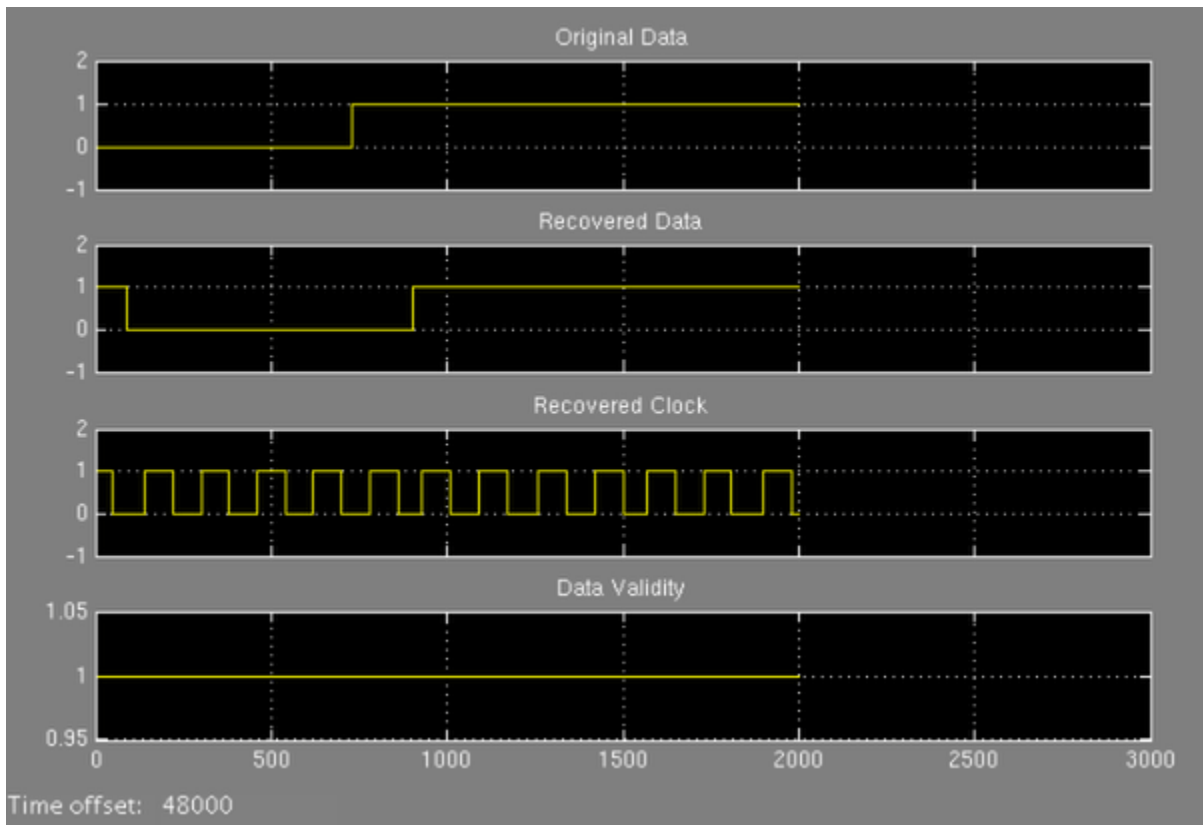
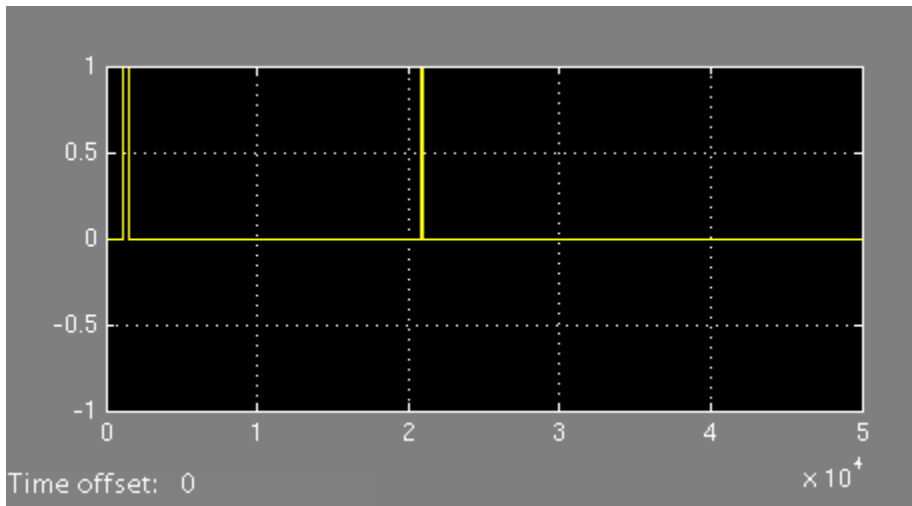
- Running a Cosimulation**
1. Click the command to the right to compile the HDL and launch the HDL simulator.
 2. Perform any debug setup in the HDL simulator such as setting breakpoints or creating signal probes.
 3. Start the cosimulation in Simulink by choosing either 'Simulation/Start' or 'Tools/Simulink Debugger'.

```

nclaunch( ...
'xrun', 'TEMPDIR', ...
'xclstart', { ...
[exec ncxvlog -linedebug 'vlogFiles()]; ...
'exec ncelab -access +rnc multimanchester', ...
'hdlsimulink -gui multimanchester' ...
} ...
);
    
```

Copyright 2006-2008 The MathWorks, Inc.





Relating HDL Clocks and Resets with Simulink Sample Times

This example illustrates the relationship of Simulink® sample times to HDL clocks and resets by using the HDL Verifier™ to cosimulate a simple synchronous Verilog parity check module. The example also contains the following:

- Explains how delta-time iterations in the HDL simulator (ModelSim® or Xcelium™) may affect cosimulation results
- Shows the use of the Clocks pane in the HDL Cosimulation block to drive clock signals in HDL
- Shows how you can accurately compare cosimulation results by taking the HDL reset logic delays into consideration

Verilog Code Used for Cosimulation

Parity checking is a method of adding a parity bit to a data stream in order to check that data for any errors. In this example we will use an "even parity bit" scheme. The HDL module is designed to be synchronous and updates its state on every rising edge of clock.

The synchronous Parity Checker module (paritychecker_clk_dut.v) accepts an 8-bit input, outputs 1 even-parity bit, and is driven on every rising clock edge.

Effects of Changing Simulink Sample Times in a Cosimulation

If you are using ModelSim or QuestaSim, the model parity_check_clk.slx should be open. If you are using Xcelium, close the ModelSim model and open the model parity_check_clk_in.slx.

1) Before running this model, launch ModelSim by clicking on the annotation below:

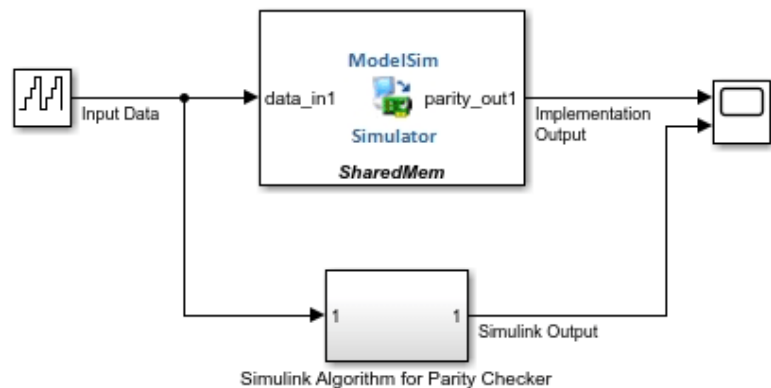
```
vsim {clstart} paritycmds_clk
```

ModelSim Startup Command (Shared Memory)

The function 'paritycmds_clk' creates Tcl commands which execute when ModelSim launches
Click here to see what's in it.

2) Observe the following:

- The settings within the HDL Cosimulation block's Ports pane
- The HDL module is driven by a clock with $T=8\text{ns}$ which is generated in HDL



3) Set sample time of output (Tout):

a) Oversampling

Tout=4e-9

b) Undersampling

Tout=32e-9

c) Sampling at clk frequency

Tout=8e-9

Tout = 8e-09

Current value of Tout

Ts = 8e-09

Sample time of input (Ts)



In the model we use an 8-bit counter to provide input data to the HDL code through the HDL Cosimulation block, and its equivalent Simulink algorithm. A scope is used to view their outputs and compare results. The model showcases how the Simulink sampling rate affects cosimulation with an HDL module. The clickable annotations can be used to change the sample time of the HDL Cosimulation block's output port (Tout).

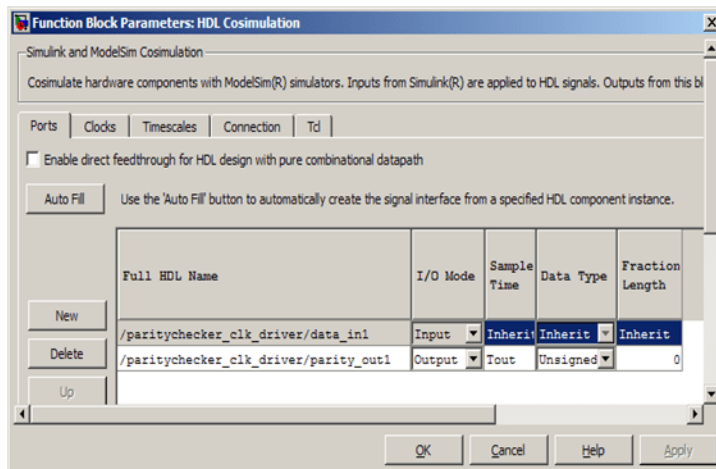
Note that the clock and reset inputs for the design under test are generated within the HDL driver module, (paritychecker_clk_driver.v). Reset is held high for the first 16ns and is low thereafter. The clock has a period of 8ns, and is set up such that its first rising edge occurs at 4ns. Hence the module is capable of updating its output at a maximum rate of 8ns, that is, at every rising edge of clock.

1. Launch ModelSim or Xcelium

Before running the model, you must first launch the HDL simulator. Use the startup command provided within the model for this.

2. Observe the settings within the HDL Cosimulation block's Ports pane

Double-click on the HDL Cosimulation block to edit the cosimulation parameters. The Block Parameters dialog appears. Select the Ports tab.



- The sample time of the output port (parity_out1) is set to Tout. There are numerous ways to specify the value of Tout. We have set the initial value of Ts from within the model's PreLoadFcn callback (see "Create Model Callbacks" (Simulink)). We set new values for Tout using either of the clickable annotations provided in the model. You can set the value of Tout at the MATLAB® Command prompt as well.
- Note that the option for allowing direct feedthrough has not been checked - this is because our Verilog code is not purely combinational.

The model provides three clickable annotations for setting the sampling time of the output ports of the HDL Cosimulation block) Tout = 32ns, 8ns, and 4ns.

3. Run the model with all three versions of Tout

Tout = 32ns

- Output of HDL module is sampled by Simulink at every 32ns

- The sampling rate of the output port is lower than the clock rate, $T_{out} = 4x(\text{HDL clock period})$
- The output of the HDL Cosimulation block is Undersampled, as a result of which and the `parity_out1` signal within the HDL simulator does not match up with the Simulink scope result

Tout = 8ns

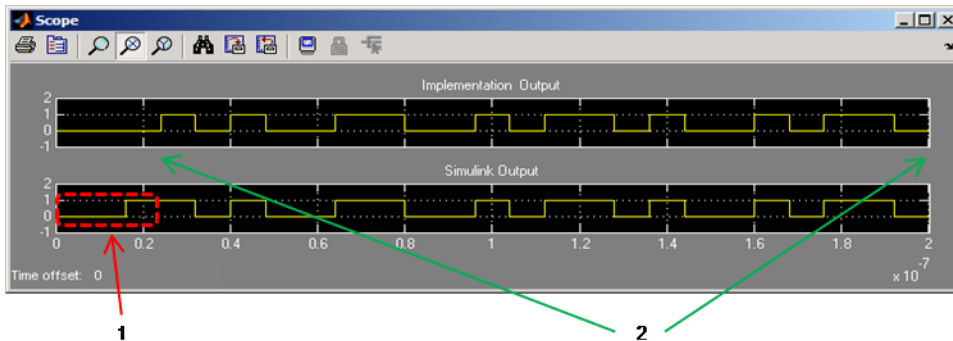
- Output of HDL module is sampled by Simulink at every 8ns
- The sampling rate of the output port perfectly matches the clock rate, $T_{out} = (\text{HDL clock period})$
- None of the outputs are missed, and the comparison waveforms match up

Tout = 4ns

- Output of HDL module is sampled by Simulink at every 2ns
- The sampling rate of the is higher than the clock rate, $T_s = 0.5x(\text{HDL clock period})$
- The output is Oversampled and the higher output sampling rate does not pay any dividend here

Hence understanding the clocking rate of a synchronous HDL module can be advantageous for cosimulation if you do not want to oversample or undersample the output from HDL.

4. Observe results in the Simulink Scope when Tout = 8ns



You will notice that the outputs from the Simulink algorithm match the outputs obtained from the HDL Cosimulation block (labeled as 2 in the image), except for the first 24ns (labeled as 1 in the image). The initial values of the two outputs do not match up due to the reset logic used within HDL (which Simulink does not know about and does not incorporate in its algorithm). We will discuss this in detail later in the example.

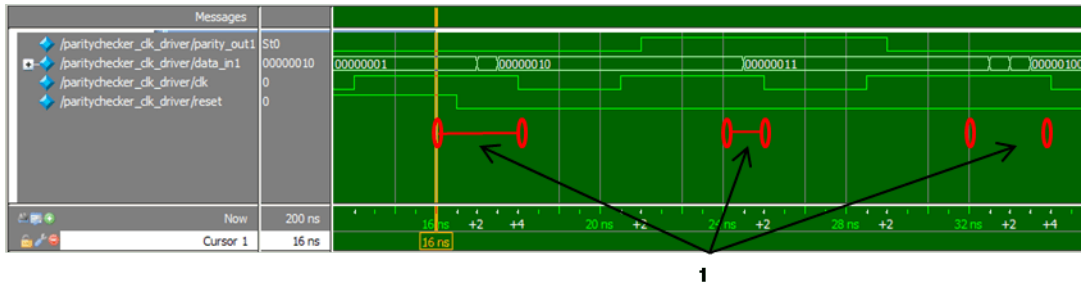
Effect of Clocks Driving an HDL Module and Race Conditions

It is important to understand that the Simulink engine does not work in delta-time cycles and hence Simulink queries the output port of the HDL Cosimulation block at definite discrete time intervals. On the other hand, the HDL simulator does not make any guarantees as to the order of a value change versus some other blocking signal assignment. Thus, if the Simulink values are driven/ sampled at the same time as an active clock edge in the HDL, there is a *race condition*. In order to avoid such race conditions it is essential that Simulink values are not driven/ sampled at the same time as an active clock edge in HDL.

In the Verilog code `paritychecker_clk_driver.v` note how the positive edge of the clock (which is the active edge) has purposely been offset by half its period so as avoid a potential race condition.

Case for 24ns delay

The Verilog code is driven such that the module is reset for the first 16ns. However, the output mismatch seen on the Simulink scope is for 24ns. In order to better understand why this delay occurs, we have captured a snapshot of the HDL simulator waveform (with delta-time delays and events expanded), when the simulation was run with $T_{out}=8ns$:



At 16ns, the output `parity_out1` within the HDL simulator still holds its previous state, since the output is only slated to change at rising edge of clock. Hence Simulink samples the previous state of output at 16ns. Also note, that even though we have set $T_s=T_{out}=8ns$ in Simulink, the Simulink engine does not know how the HDL simulator will perform its delta-time iterations. Hence the outputs could be sampled by Simulink within the delta-time ranges (labeled *1*) shown in the image above.

Convenience Clock

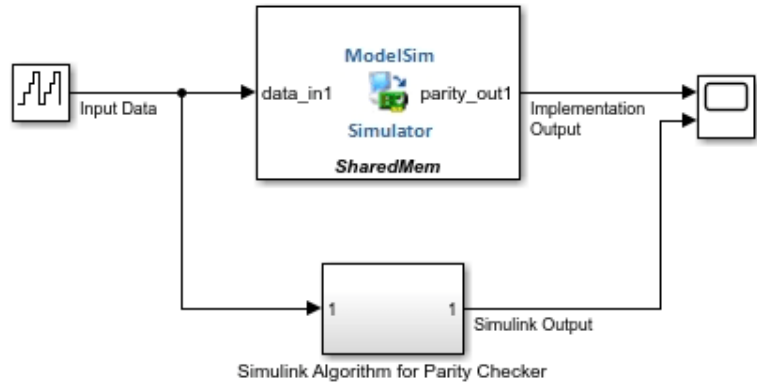
Instead of creating your own driver code (testbench) for the HDL module, you can use the HDL Cosimulation block's convenience clock to generate the clocking input. You can specify the clock period (T) and its active edge in the *Clocks* pane of the HDL Cosimulation block. This clock generated by the HDL Cosimulation block has a deliberate $T/2$ delay applied to the first active clock edge (which you specify) - in order to avoid race conditions. In order to show how to use this clock, we have provided models, `parity_check_convclk.slx` (ModelSim) and `parity_check_convclk_in.slx` (Xcelium) which use a clock created by the HDL Cosimulation block as the driving signal for the modified version of the even parity checker modules (`paritychecker_convclk.v`).

1) Before running this model, launch ModelSim:

```
vsim(tclstart_paritycmds_convclk)
```

ModelSim Startup Command (Shared Memory)

The function 'paritycmds_convclk' creates Tcl commands which execute when ModelSim launches. Click here to see what's in it.



2) The clock is generated by the Cosimulation block with a period $T = 8\text{ns}$

3) Set sample time of output (Tout):

Tout=8e-9

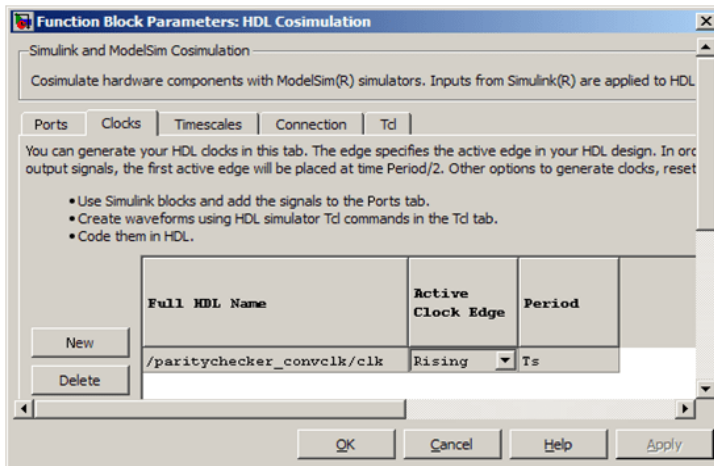
Tout = 8e-09
Current value of Tout

Ts = 8e-09
Sample time of input (Ts)



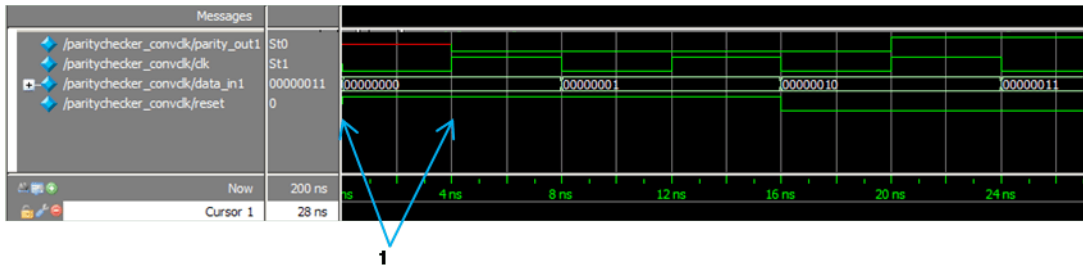
Copyright 2003-2020 The MathWorks, Inc.

The Clocks pane of the HDL Cosimulation blocks is set as shown in the image below. Notice how the active edge of clock is set to be rising:



Run the model and observe outputs

The results are the same as those obtained in the example “Timescales: Absolute, Relative and Automatic” on page 32-181. The output of the HDL simulator waveform is captured in the image below:



Notice how an initial phase shift of $T/2$ is applied to the first active edge of `clk` - shown by the label *1* in the image.

Accounting for Reset Delay in Order to Compare Cosimulation Results

Simulink does not know about any resetting logic that the HDL module may have in place and does not incorporate such reset logic in its algorithm. Hence the HDL cosimulation results will be out of sync with respect to the Simulink algorithm (subsystem).

Now, if you need to compare the output of the HDL Cosimulation block with the results obtained from Simulink's algorithm, you have to ensure that both these simulations are synchronized. There are a number of ways to achieve this, one of which are shown in `parity_check_reset.slx` (ModelSim) and `parity_check_reset_in.slx` (Xcelium).

1) Before running this model, launch ModelSim:

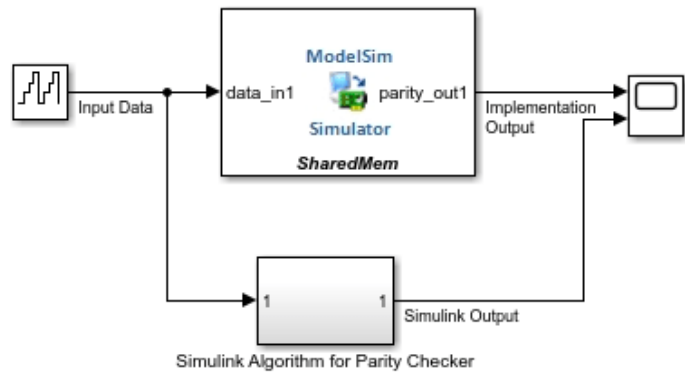
```
vsim('tclstart' paritycmds_reset)
```

ModelSim Startup Command (Shared Memory)

The function 'paritycmds_reset' creates Tcl commands which execute when ModelSim launches. A snippet of these Tcl commands is:

```
'add wave /paritychecker_convclk*', ...
'echo Setting the HDL Simulator resolution to: $resolution', ...
'run 16' ...
```

[Click here to see what's in it.](#)



2) The 'tclstart' is used to provide Tcl commands to the HDL simulator and run the HDL module for the duration of Reset (which is 16ns)

Clock is generated by the Cosimulation block with a period $T = 8ns$

3) Set sample time of output (Tout):

```
Tout=8e-9
```

Tout = 8e-09

Current value of Tout

Ts = 8e-09

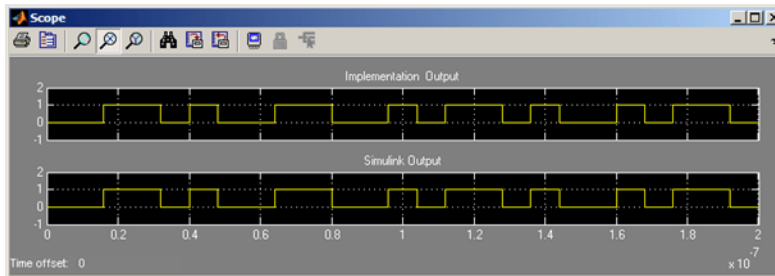
Sample time of input (Ts)



Copyright 2009-2020 The MathWorks, Inc.

Here we use the `tclstart` arguments to run the HDL simulator for 16ns (reset period) immediately after HDL Verifier sets up the cosimulation link. Hence the HDL module runs for 16ns before you start the simulation in Simulink.

Run the model and observe Scope output



Key Points to Note:

- All port sample times and clock specifications are in Simulink time. For example, if timescale is set to '1s in Simulink corresponds to 1s in HDL simulator', the clock period should be $T=8\text{ns}$. However, if the timescale is set to '1s in Simulink corresponds to 1ns in HDL simulator', the clock period should be $T=8\text{s}$. This is explained in the example "Timescales: Absolute, Relative and Automatic" on page 32-181
- The clock generated in the clocks pane is meant to drive HDL code only
- All signals driven from the Tcl pane or in the `tclstart` of the HDL simulator's launch command are in HDL time
- All signals driven from within HDL code are in HDL time

Timescales: Absolute, Relative and Automatic

This example illustrates the various Timescale settings within the HDL Cosimulation block and explains how these affect the timing relationship of Simulink® and the HDL simulator. We use a simple Verilog parity check model to show the timing relationship of Simulink and the HDL simulator (ModelSim® or Xcelium™) used for cosimulation.

Introduction

The model illustrates cosimulation of an HDL implementation of a simple parity checker and showcases the various options available in the *Timescales* pane of the HDL Cosimulation block. In that pane, Simulink time is related to HDL simulator time either in an absolute or relative sense. You can set the relationship explicitly or have the software automatically pick a relationship in either mode based on knowledge of the Simulink and HDL designs.

Open the Simulink Model

If you are using ModelSim or QuestaSim, the model `parity_check_mq.slx` should be open. If you are using Xcelium, close the ModelSim model and open the model `parity_check_in.slx`.

```
% For ModelSim:
modelName = 'parity_check_mq';
open_system(modelName);

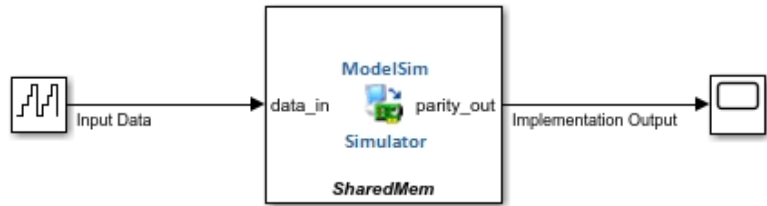
% For Xcelium:
modelName = 'parity_check_in';
open_system(modelName);
```

1) Before running this model, launch ModelSim by clicking on the annotation below:

```
vsim('to_start', paritycmds('1ns'))
```

ModelSim Startup Command (Shared Memory)

The function 'paritycmds' creates Tcl commands which execute when ModelSim launches
Click here to see what's in it.



2) Observe the settings within the HDL Cosimulation block's Ports pane

3) Choose mode of operation:

a) `paritytimescale(absolute)`

b) `paritytimescale(relative)`

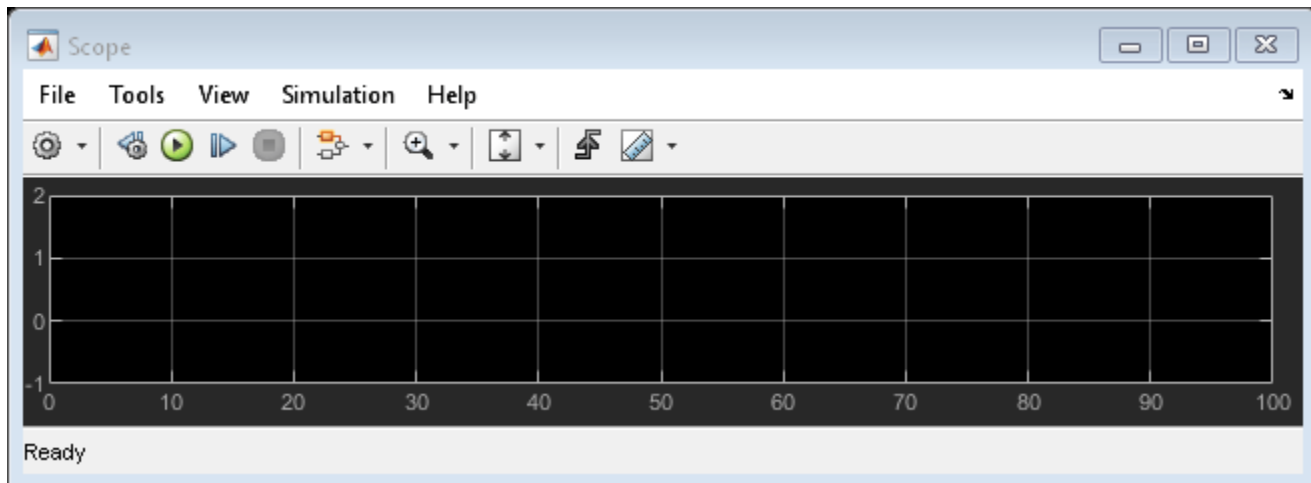
c) Auto Timescale

- Set Simulink sample time, $T_s = 10$;
Within the 'Timescales' pane of HDL Cosimulation block, click on the Auto Timescale button, and apply the changes.
- Now try the same procedure using an irrational value for T_s , for example, $T_s = \pi * 1e-9$

$T_s = 10$
Value of T_s



Copyright 2009-2020 The MathWorks, Inc.



Verilog Code Used for Cosimulation

Parity checking is a method of adding a parity bit to a data stream in order to check that data for any errors. In this example we will use "even parity bit" scheme in which the parity bit is set to 1 if the number of ones in a given set of bits is odd (making the total number of ones, including the parity bit, even).

We select the input to the parity checker to be 8-bits and output as 1 bit. The Verilog code used for this example is paritychecker.v.

Preparing the Model for Cosimulation

In order to guide you through the various steps of this example, we have provided numbered steps which you can follow. Each numbered step is followed by a brief explanation of the same.

1. Launch ModelSim/Xcelium

Before running the model, you must first launch the HDL simulator. Use the startup command provided within the model for this or run the simulator-specific command below. This also sets the HDL simulator resolution to 1ns.

For Modelsim/Questasim:

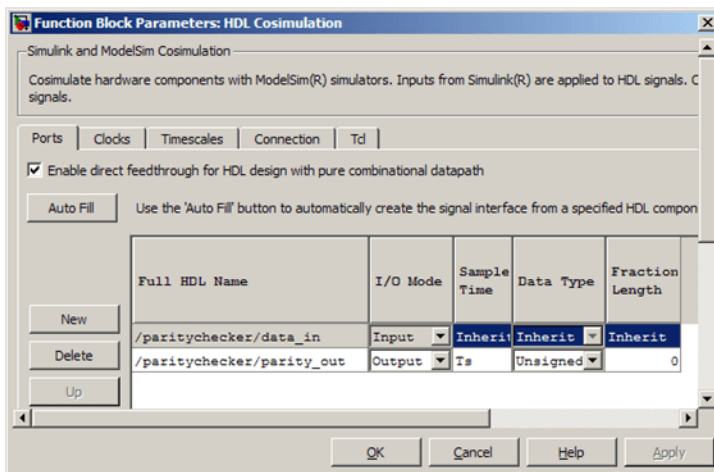
```
vsim('tclstart', paritycmds('1ns'));
```

For Xcelium:

```
nclaunch('tclstart', paritycmds_in('1ns'));
```

2. Observe the settings within the HDL Cosimulation block's Ports pane

Double-click on the HDL Cosimulation block to edit the cosimulation parameters. Select the Ports pane.



- The sample time of the output signal is set to Ts - which is the sampling rate of the model. There are numerous ways to specify the value of Ts. We have set Ts from within the model's PreLoadFcn callback (see "Create Model Callbacks" (Simulink)). You can set the value of Ts in the base workspace at the MATLAB® Command prompt as well.
- Note that the option for allowing direct feedthrough has been checked - this is because our Verilog code is purely combinational.

This model is set up to work in three timing modes: Absolute, Relative, and Auto Timescale. You can read more about these modes in user guide section, "Simulation Timescales" on page 10-44.

Absolute Timescale

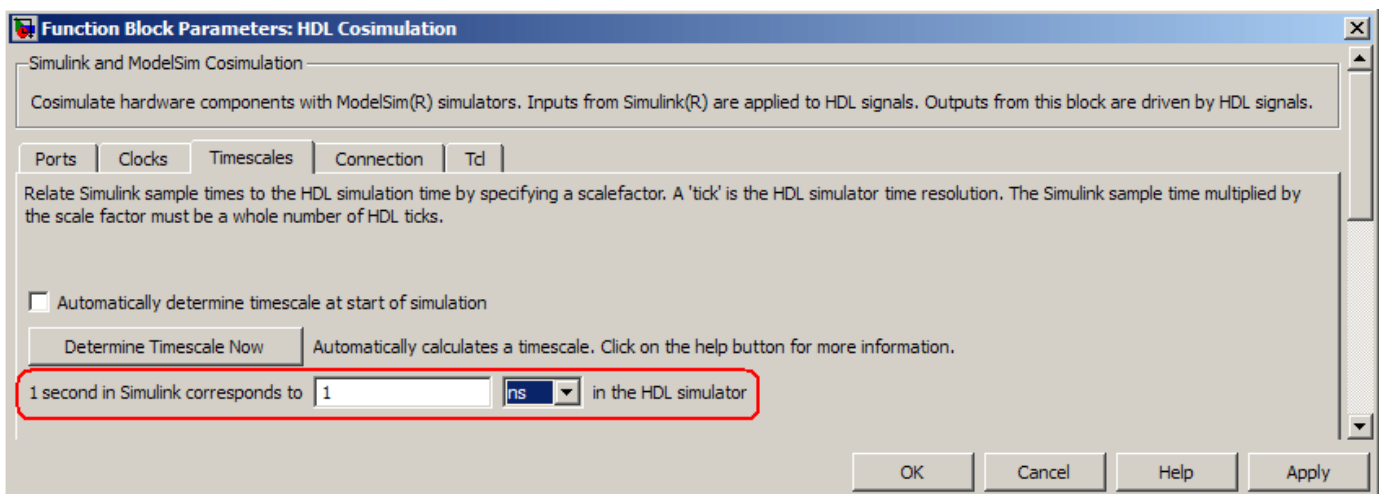
Absolute timing mode lets you define the timing relationship between Simulink and the HDL simulator in terms of absolute time units and a scale factor.

3. Click on the Absolute Mode annotation in the model to set the following:

- Sample time $T_s = 10$
- Set Timescales pane of HDL Cosimulation block to "1s in Simulink corresponds to 1ns in the HDL simulator"

Or invoke the following function:

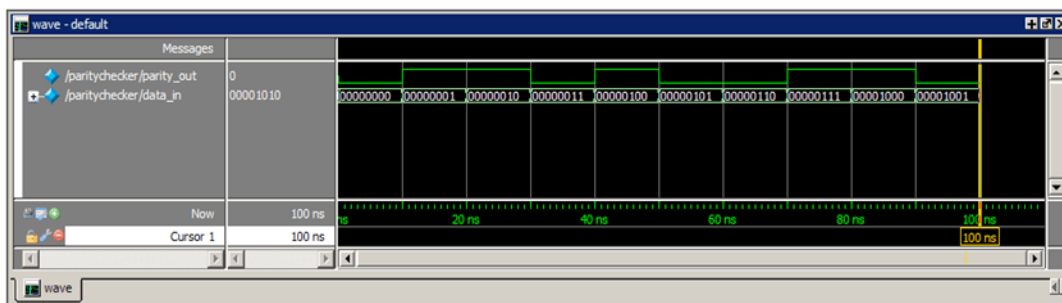
```
paritytimescale('absolute');
```

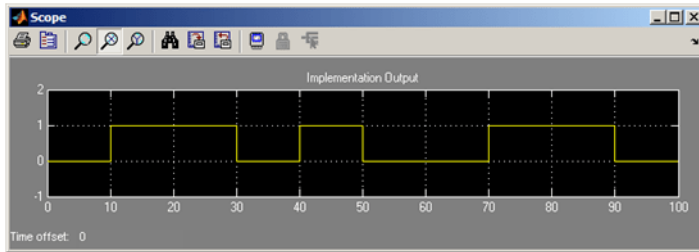


The absolute time units in this case are 'ns' and the scale factor is '1'.

4. Run the model

The HDL simulator waveform shows the simulation run for 100ns, whereas the Simulink scope depicts how the model has run for 100s. This is expected for a Timescale setting of 1s in Simulink = 1ns in the HDL simulator.





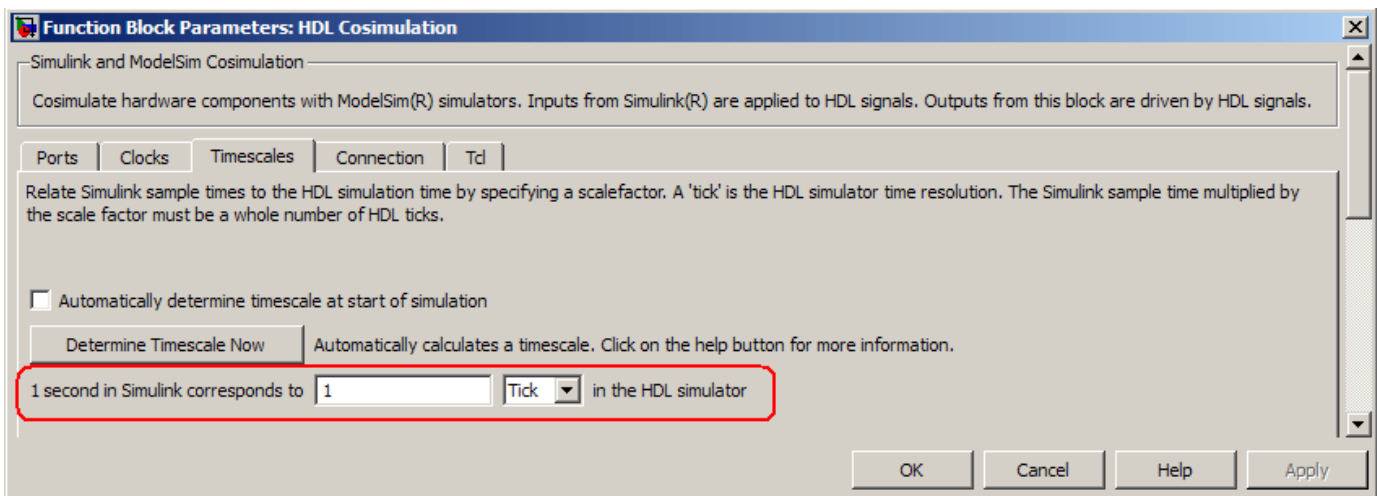
Relative Timescale

Relative timing mode lets you define the timing relationship between Simulink and the HDL simulator in relative terms - that is, as some number of HDL simulator ticks.

5. Click on the Relative Mode annotation in the model to set the following:

- Sample time $T_s = 10$
- Set Timescales pane of HDL Cosimulation block to "1s in Simulink corresponds to 1 Tick in the HDL simulator":

```
paritytimescale('relative');
```



The time units in this case are HDL simulator ticks and the scale factor is '1'.

6. Restart the HDL simulation.

For ModelSim: You can invoke a restart from within the HDL simulator.

```
vsim> restart
```

For Xcelium: Close all HDL simulator windows first and relaunch the simulator.

```
nclaunch('tclstart',paritycmds_in('lns'));
```

7. Run the model

The HDL simulator waveform shows the simulation run for 100ns, whereas the Simulink scope depicts how the model has run for 100s. This is expected for a Timescale setting of 1s in Simulink = 1

Tick in the HDL simulator (and the resolution of the simulator is set to 1ns). The waveforms obtained are similar to those we got in the absolute timescaling mode.

8. Change HDL Simulator resolution

Now we will see the effect of changing the HDL simulator resolution to 1ps. Close all of the HDL simulator windows and restart using a 1ps timescale resolution.

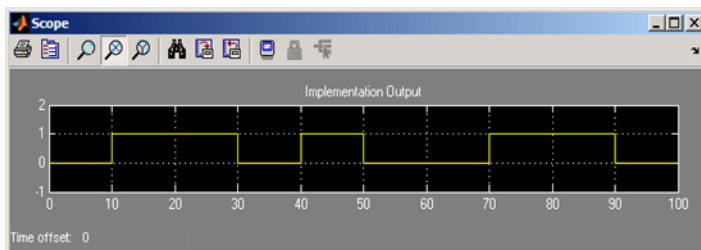
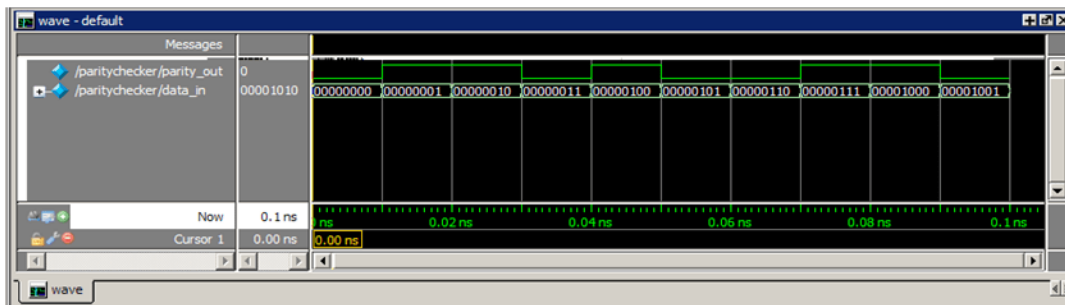
For Modelsim/Questasim:

```
vsim('tclstart',paritycmds('1ps'));
```

For Xcelium:

```
nclaunch('tclstart',paritycmds_in('1ps'));
```

Run the Simulink model for the same 10 samples. The HDL simulator runs for 100ps since the Timescale is set to 1s in Simulink = 1 Tick in the HDL simulator.



Auto Timescale

Within the Timescales pane of the HDL Cosimulation block the *Determine Timescale Now* pushbutton calculates a timing relationship between Simulink and the HDL simulator. Please note that the link needs to find the resolution of the HDL simulator, and hence you will need to have the HDL simulator up and running.

Restart the HDL simulator using 1ns Resolution

Close all open HDL simulator windows first. Then:

For ModelSim:

```
vsim('tclstart',paritycmds('1ns'));
```

For Xcelium:

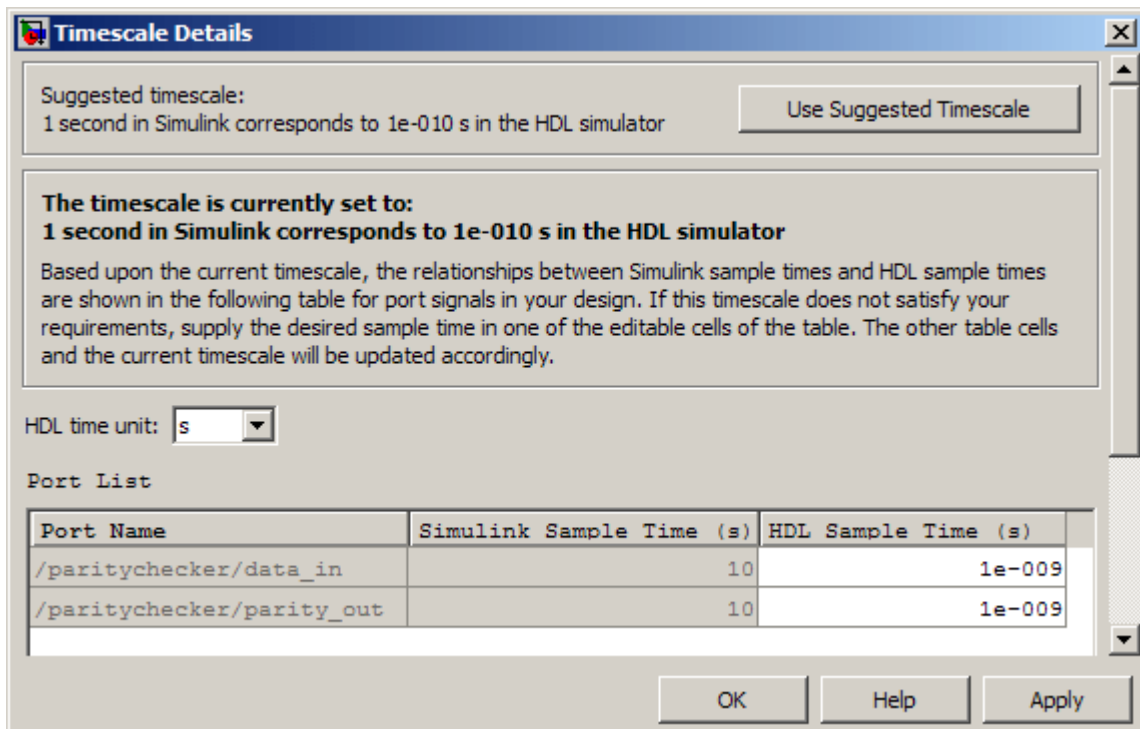
```
nclaunch('tclstart',paritycmds_in('1ns'));
```

When you allow the link software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink (absolute timescale mode). If this setting is not possible, the HDL Verifier attempts to set the signal rate on the Simulink model port to the lowest possible number of HDL simulator ticks (relative timescale mode).

Now we will use the Auto Timescale within the model as shown in the next step:

9. Click on Determine Timescale Now button in the Timescales pane

This will suggest to set the Timescale to "1s in Simulink = 1e-10s in the HDL simulator". An info pane will be displayed:



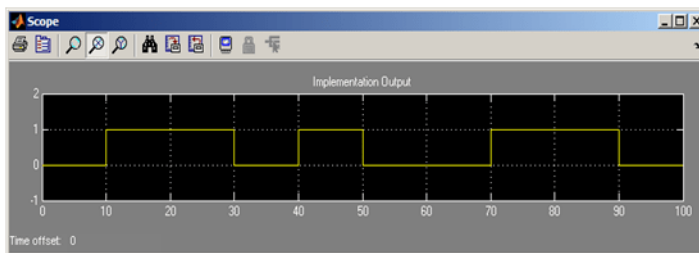
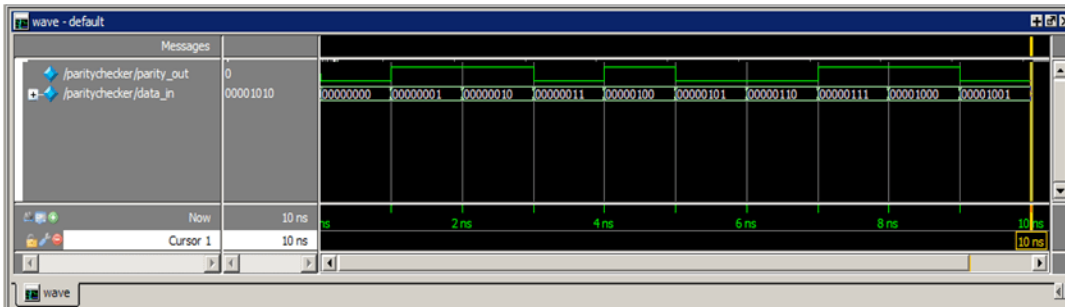
The HDL Verifier software attempted to first achieve a 1:1 timescale setting, but failed to do so because the resolution of the HDL simulator is set to 1ns, and representing $T_s=10$ seconds would amount to $10e9$ number of ticks which is greater than the $2^{31} - 1 (=2.1475e9)$ allowable limit. The number $2^{31} - 1$ is the maximum value for an int32.

Once the HDL Verifier cannot establish an absolute 1:1 timescale it switches to relative timescale mode in which it will equate the fundamental sample time ($T_s=10s$) with 1 HDL tick. Hence 10s in Simulink will correspond to 1 HDL tick, that is, 1s in Simulink corresponds to 0.1 Tick in the HDL simulator.

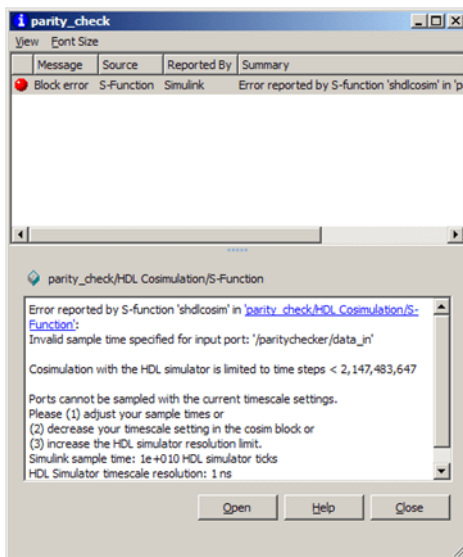
Apply the changes suggested by the HDL Cosimulation block.

11. Run the model

The waveforms within the HDL simulator and from the Simulink scope are shown below. Note how the HDL simulator progresses through 10ns whereas the Simulink model runs for 100s:

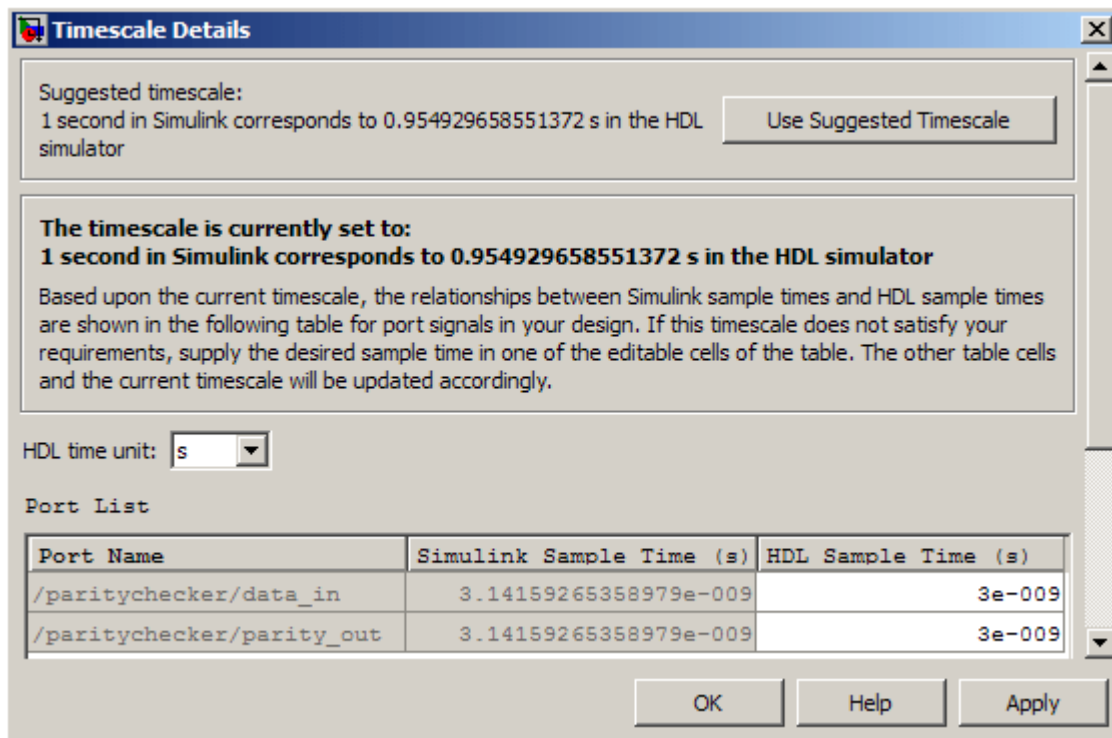


It is important to understand why the HDL Verifier did not choose a 1:1 mapping. If you were to set the Timescales setting to "1s in Simulink corresponds to 1s in the HDL simulator", apply this change, and run the model, Simulink will give you the following error:



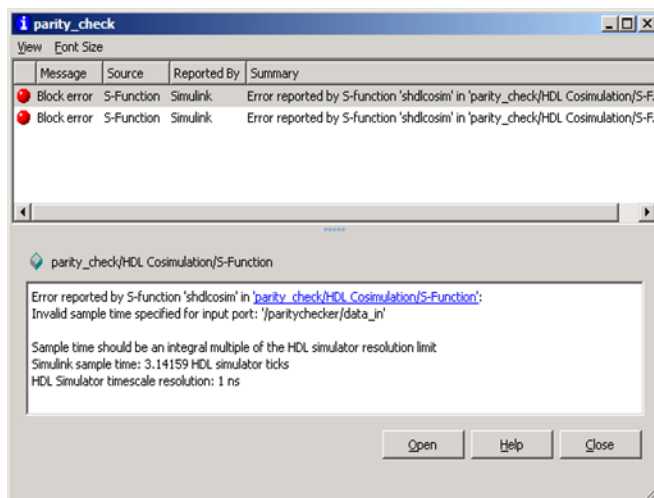
12. Set an irrational sampling time ($T_s = \pi * 1e-9$)

If you were to use irrational sample times within Simulink, for example, $T_s = \pi * 1e-9$, the Auto Timescale will not be able to maintain an exact relationship between Simulink simulation and HDL simulation time. In this case, although the results of cosimulation are correct, the time axes of the Simulink scope and HDL simulator will not have a 1:1 relation.



13. Set Timescale to '1s in Simulink = 1s in HDL simulator'

In order to understand why 1:1 relationship is not possible, set the Timescale to '1s in Simulink = 1s in HDL simulator', and run the model (after restarting the simulation in the HDL simulator). A 1:1 timing relationship is not possible in this case as pi is an irrational number and is not an integral multiple of the 1ns (the HDL simulator resolution). Simulink will give you an error message in this case, as shown in the following image:

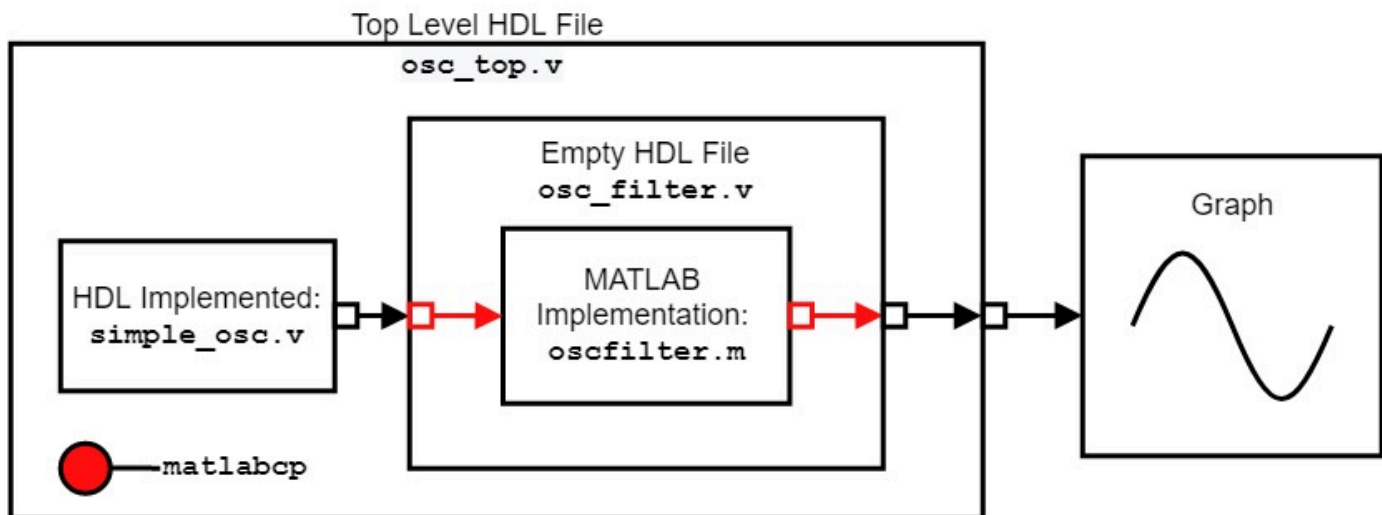


Note: Each time you change the Timescale setting within the HDL Cosimulation block, you will need to restart the simulation run within the HDL simulator.

Implement Filter Component of Oscillator in MATLAB

This example shows how to implement a filter component for an HDL project in Mentor Graphics® ModelSim®/Questasim® or Cadence® Xcelium® by using MATLAB® the HDL Verifier™ function `matlabcp`. Instead of using HDL to model a complex component for use in an HDL project, you can use `matlabcp` to implement the component instead. `matlabcp` needs a MATLAB function modeling the component behavior, along with an empty HDL component for input and output ports declared. These files are necessary as `matlabcp` uses the empty component as a shell for communication between MATLAB and the HDL simulator, while the MATLAB function provides the functionality. Using `matlabcp` when verifying a single component allows MATLAB to simulate the various parts of the system, saving time and effort needed to create HDL code for testing. This approach shifts the focus onto generating tests and testing the component in question, creating a more efficient verification workflow. For information on creating the necessary files for `matlabcp`, see “Create a MATLAB Component Function” on page 3-2.

In this example, the project compiles an oscillator in VHDL® and defines the filter component using MATLAB, before running an HDL simulation. The oscillator is a simple fixed-point sine wave generator written in HDL, and feeds a lowpass filter. This lowpass filter is a 255th order (256-tap) filter with 8x oversampling that is implemented using the polyphase technique in MATLAB. The filter inputs and outputs are carried by `matlabcp` between the HDL simulator and MATLAB, with the results displayed using the simulator waveform viewer. By using simulator waveform viewer to display the results, this example demonstrates that `matlabcp` can seamlessly integrate a component designed in MATLAB into a larger HDL workflow without requiring MATLAB in the later steps. This diagram shows the overall flow of this cosimulation process.



This example considers the point of view of an HDL developer, so the main emphasis of this example will be on using the HDL simulator and command-line terminal to perform testing, and using MATLAB to implement some of the project components.

Overview of Design Files and Script Files

This example uses two types of files: design files and script files.

Design Files

- The file `simple_osc.vhd` contains the simple sinusoidal oscillator designed using VHDL.
- The file `osc_filter.vhd` is the empty component that is used for `matlabcp` to receive the component inputs and transmit the component outputs.
- The file `osc_top.vhd` contains the top-level model of the oscillator and lowpass filter.
- The file `oscfilter.m` contains the behavioral implementation of the filter, implemented in MATLAB.

Script Files

- The file `qcommands_osc_w.tcl` contains the commands that are sent into ModelSim/Questasim on Windows® for cosimulation to occur.
- The file `qcommands_osc_l.tcl` contains the commands that are sent into ModelSim/Questasim on Linux® for cosimulation to occur.
- The file `xcelium_osc.tcl` contains the commands for opening the Xcelium simulator so that MATLAB can communicate with the simulator.
- The file `xmcommands_osc.tcl` contains the commands that are sent into Xcelium for cosimulation to occur.

Additionally, this example requires access to ModelSim/Questasim or Xcelium and MATLAB from the terminal. You must include these products on the system path.

Adjust the Script Files

This example uses the provided script files to link MATLAB and the HDL simulator. The script uses two commands: a command that invokes the HDL simulator and a command from the MATLAB shared library which creates the connection for cosimulation. To run this example, you need to change the command that invokes the HDL simulator, as the default command relies on the absolute path to the MATLAB shared library. Adjust the script files based on the simulator and operating system.

ModelSim/Questasim:

- Windows: The figure shows lines 3 and 4 of `qcommands_osc_w.tcl`. These lines contain the commands necessary for cosimulation with Modelsim/Questasim on Windows. Line 3 is the call to open ModelSim/Questasim, where the `-foreign` argument supplies the path to the MATLAB shared library, such that Modelsim/Questasim loads in the shared library. Line 4 is the call to a function specified in the shared library that connects the two programs together.

```
3 vsim work.osc_top -foreign {matlabclient
  {matlabroot\toolbox\edalink\extensions\modelsim\windows64\liblfmhdhc_gcc450vc12.dll} }
4 matlabcp u_osc_filter -mfunc oscfilter
```

For this example to work, you must change line 3 in `qcommands_osc_w.tcl`. Swap `matlabroot` with the install location of MATLAB, so that the shared library can be located and loaded in.

- Linux: The figure below shows lines 3 and 4 of `qcommands_osc_l.tcl`. These lines contain the commands necessary for cosimulation with Modelsim/Questasim on Linux. Line 3 is the call to open ModelSim/Questasim, where the `-foreign` argument supplies the path to the MATLAB shared library, such that Modelsim/Questasim loads in the shared library. Line 4 is the call to a function specified in the shared library that connects the two programs together.

```
3 vsim work.osc_top -foreign {matlabclient {matlabroot/toolbox/edalink/
  extensions\modelsim/linux64/liblfmhdhc_gcc450.so} }
4 matlabcp u_osc_filter -mfunc oscfilter
```

For this example to work, you must change line 3 in `qcommands_osc_l.tcl`. Swap `matlabroot` with the install location of MATLAB, so that the shared library can be located and loaded in.

Xcelium

The figure shows lines 3--6 of `xcelium_osc.tcl`, which contain the command that opens the Xcelium simulator. Line 4 tells Xcelium to run a command from the shared library to link the two programs. Line 6 tells Xcelium to load the MATLAB shared library.

```
3 exec <@stdin >@stdout xmsim -gui osc_top -64bit \  
4 -input {@call matlabcp :u_osc_filter -mfunc oscfilter} \  
5 -input {xmcommands_osc.tcl} \  
6 -loadcfc {matlabroot/toolbox/edalink/extensions/incisive/linux64/liblfihdlc_gcc41.so:matlabclient}
```

For this example to run you must change `xcelium_osc.tcl` on line 6. Here you need to replace `matlabroot` with the installation location of MATLAB so that the shared library can be located and loaded in.

Run Cosimulation

To start cosimulation, open MATLAB and run the HDL Link MATLAB server for communication between the HDL simulator and MATLAB. To complete these actions, use these commands based on operating system.

- Windows: `matlab -nodesktop -r "hdldaemon"`
- Linux: `xterm -e "matlab -nodesktop -r "hdldaemon"" &`

This opens MATLAB in a separate terminal window, in headless mode, and starts the HDL Link MATLAB server. The terminal that is running MATLAB displays this message.

```
"HDLDaemon shared memory server is running with 0 connections"
```

After this message appears, start the cosimulation by entering the applicable command (based on the simulator and operating system) into a system terminal.

ModelSim/Questasim

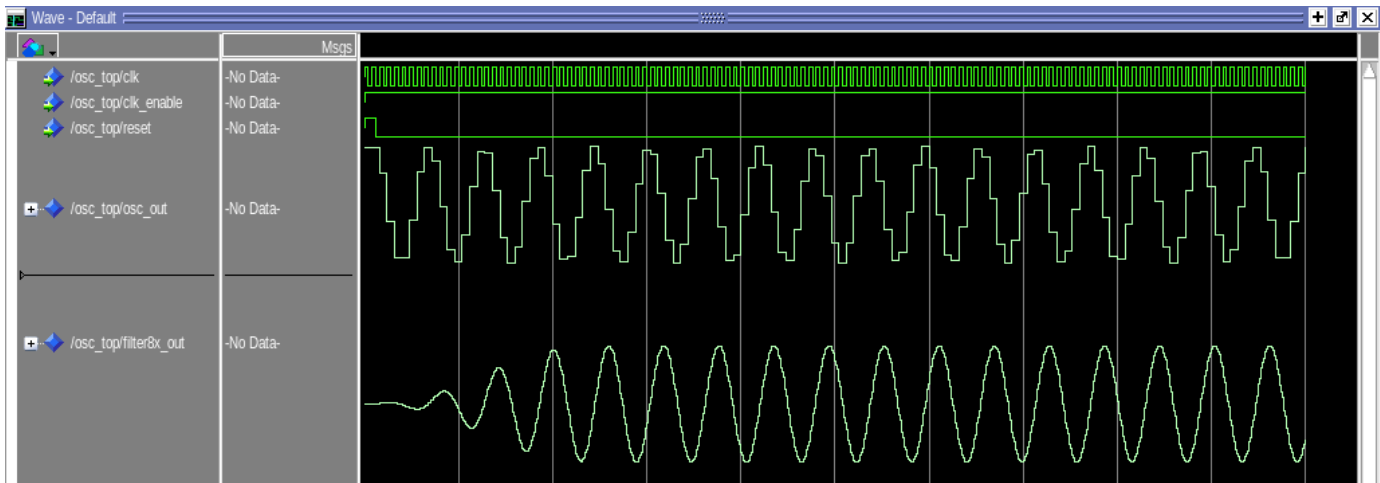
- Windows: `vsim -do qcommands_osc_w.tcl`
- Linux: `vsim -do qcommands_osc_l.tcl`

Xcelium

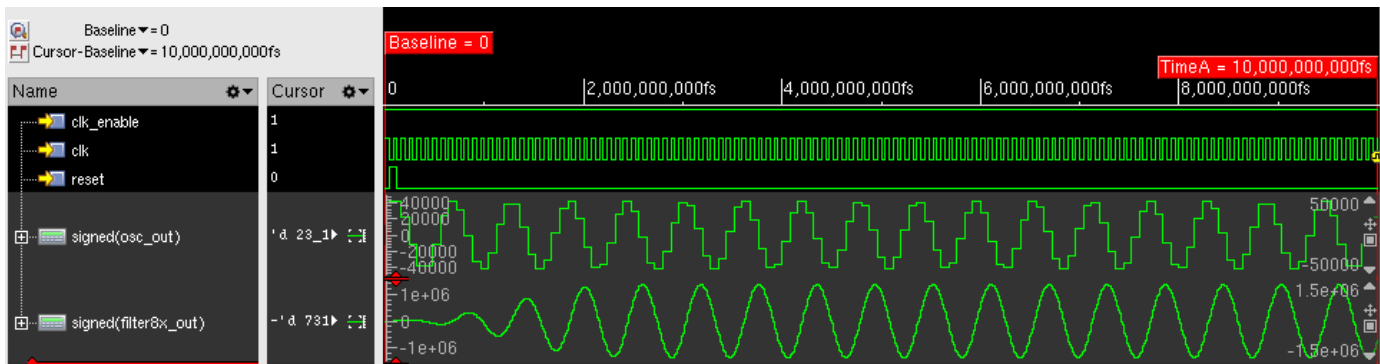
- Linux: `tclsh xcelium_osc.tcl`

After you enter the respective commands, cosimulation starts and produces these plots in the respective HDL simulator. Your plots might appear different right after cosimulation, but zooming to encompass the entire time axis can yield similar looking plots.

ModelSim/Questasim



Xcelium



Summary

This example shows how to use MATLAB as a substitute for a major component in the HDL project, replacing the HDL implementation of the lowpass filter. The file `osc_filter.vhd` is a placeholder in the system to feed the inputs into the MATLAB function and send the outputs from the MATLAB function into the rest of the project. The implementation of the filter and process of filtering the input to produce the output is completely driven by the MATLAB function.

You can use MATLAB to replace various components in the HDL simulation project with behavioral implementations in MATLAB, less time is needed to prepare the HDL project. This approach shifts the focus onto testing and verification of the component in question, instead of spending time verifying and creating the other components of the project.

See Also

`matlabcp` | `hdldaemon` | `vsim` | `nclaunch`

Related Topics

- “Set Up for HDL Cosimulation” on page 10-2
- “Create a MATLAB Component Function” on page 3-2
- “Set Up Cosimulation Component” on page 3-8

- “Run MATLAB-HDL Cosimulation” on page 1-4

Copyright 2003-2021 The MathWorks, Inc.

Manchester Receiver Using Mixed Design (Verilog and VHDL)

This example shows verification of a Manchester encoder using mixed HDL languages, VHDL and Verilog. Manchester encoding is a simple modulation scheme which converts baseband digital data into an encoded waveform with no DC component. The most widely known application of this technique is Ethernet.

This model simulates a pure-digital receiver of Manchester encoded data. The receiver is implemented in VHDL/Verilog. The receiver uses a simple DLL clock recovery mechanism, which requires multiple cycles to lock with the incoming data stream. The performance of the receiver is explored by applying phase and frequency errors to a randomly generated stream of bits that is encoded using a simple MATLAB® function: manchesterencoder().

The actual VHDL/Verilog code will run in ModelSim®/Xcelium® using the cosimulation block called "Mixed HDL Manchester Receiver"

In this example VHDL implementations are used for the lower level blocks and the top block implementation is in Verilog. Connections are made to some signals in Verilog and others in VHDL via the ports pane of the HDL Cosimulation block. In spite of the differences in HDL languages, the syntax for the connections is consistent. It is also important to notice that the ports pane is used here to connect to HDL signals that are not actually ports at the top-level module. In fact, connections can be made to signals at any level of the HDL hierarchy by the HDL Verifier cosimulation block.

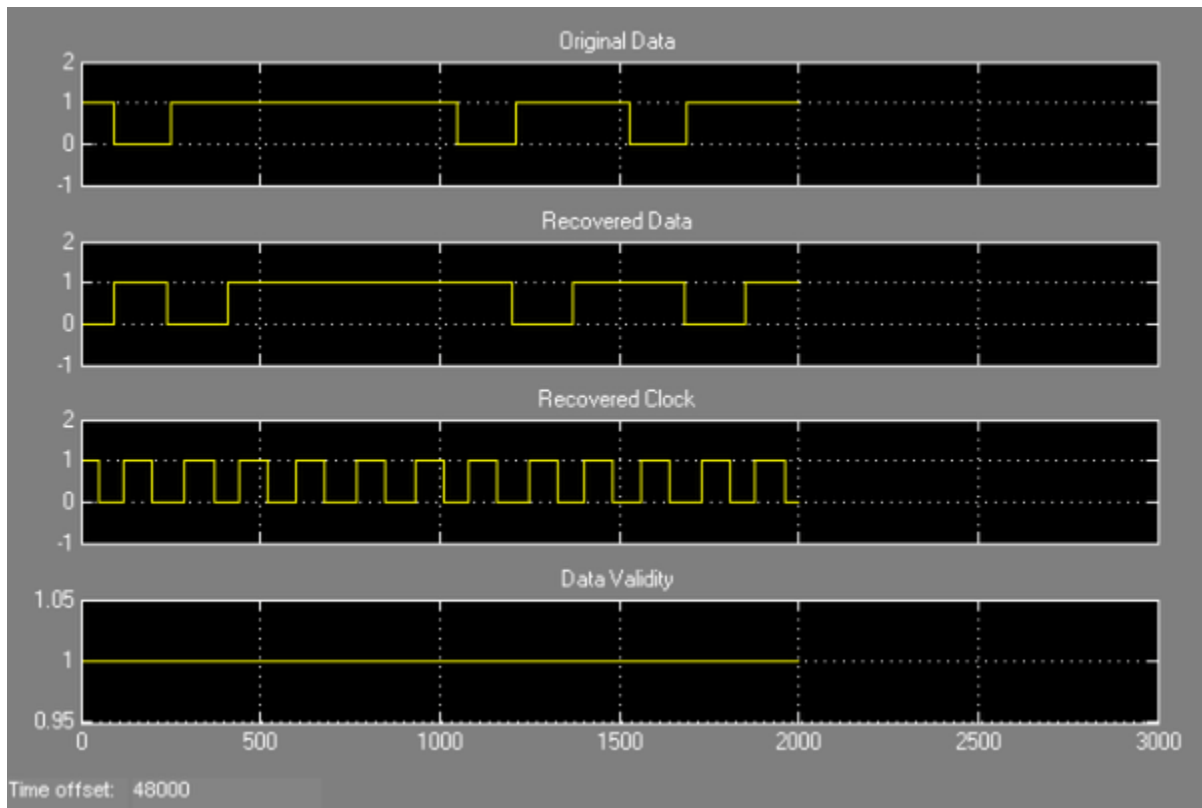
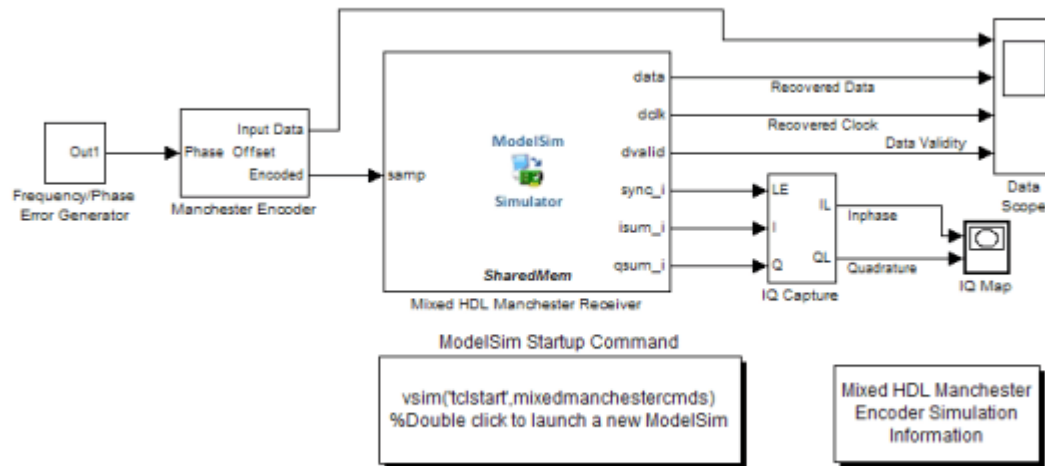
Languages used for the implementations of the HDL blocks:

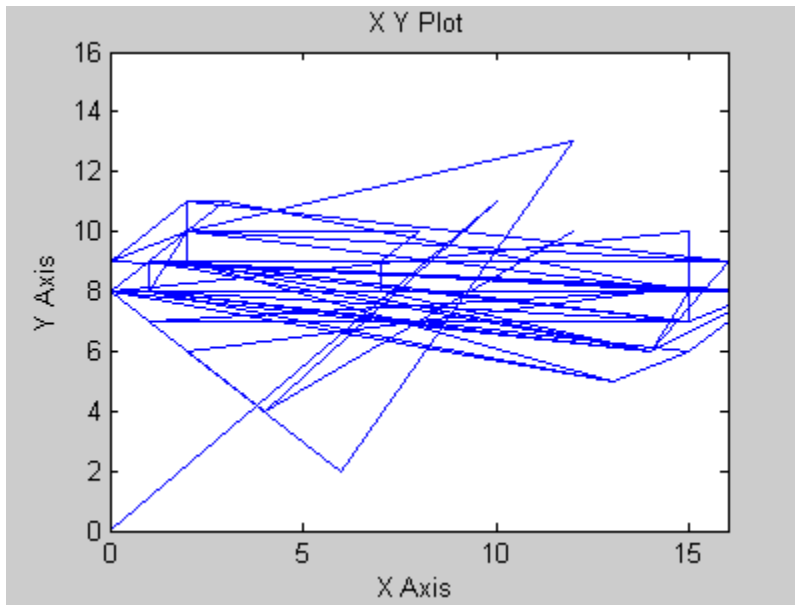
- top-level (manchester): Verilog
- decoder: VHDL
- iq converter: VHDL
- state counter: VHDL

The actual VHDL and Verilog code will run in the HDL simulator and its execution will be seen in Simulink as the behavior of the HDL Cosimulation cosimulation block called "Mixed HDL Manchester Receiver."

Open the ModelSim® mixed-language model.

Open the Xcelium® mixed-language model.





Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop

This example shows you how to set up an FPGA-in-the-Loop (FIL) application using HDL Verifier™.

The application uses Simulink® and an FPGA development board to verify the HDL implementation of a proportional-integral-derivative (PID) controller. In this example, Simulink generates the desired position of a motor and simulates the motor controlled by this PID controller.

Requirements and Prerequisites

Aside from the listed MathWorks products, other requirements include:

- FPGA design software (Xilinx® ISE® design suite, Xilinx Vivado® design suite, Intel® Quartus® II design software, or Microchip Libero® SoC design software), with a supported version listed in “FPGA Verification Requirements”.
- One of the supported FPGA development boards. For supported hardware, see “Supported FPGA Devices for FPGA Verification”.
- For connection using Ethernet: Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable.
- For connection using JTAG: USB Blaster I or II cable and driver for Intel FPGA boards. Digilent® JTAG cable and driver for Xilinx FPGA boards.
- For connection using PCI Express®: FPGA board installed into PCI Express slot of host computer.

Prerequisites:

MATLAB® and FPGA design software can either be locally installed on your computer or on a network accessible device. If you use software from the network you will need a second network adapter installed in your computer to provide a private network to the FPGA development board. Consult the hardware and networking guides for your computer to learn how to install the network adapter.

Step 1: Set Up FPGA Development Board

Skip this step and step 2 if you are using PCI Express connection for simulation. If you have not set up your PCI Express connection, use the support package installation software to guide you through PCI Express setup.

Use the following steps to set up your FPGA development board.

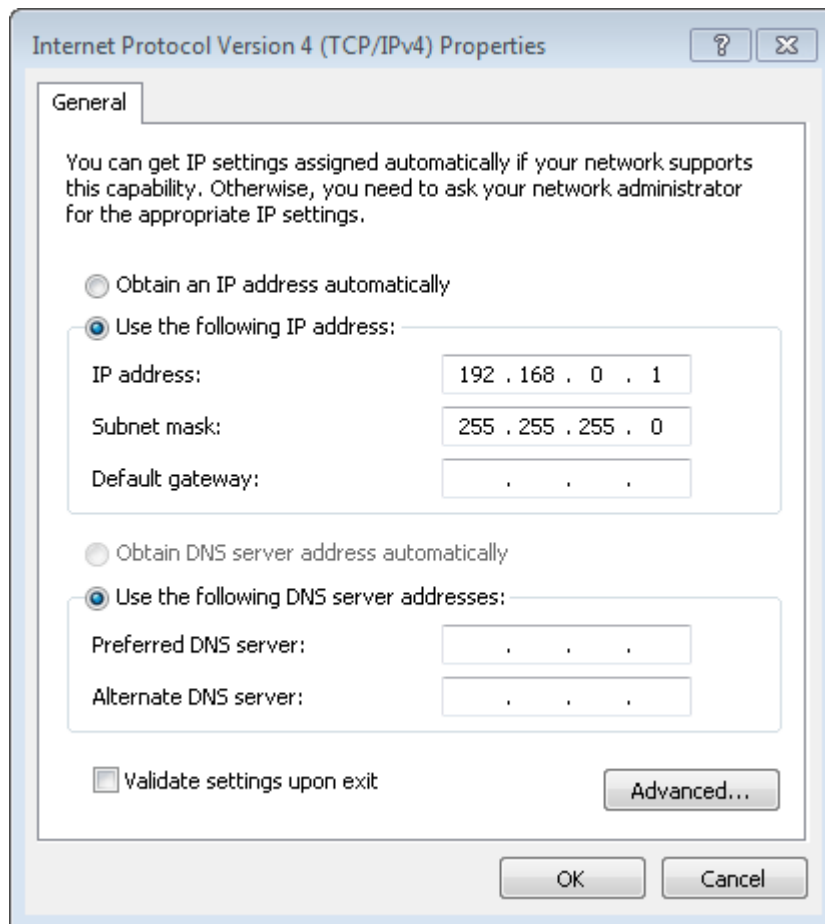
- 1 Make sure that the power switch remains **OFF**.
- 2 Connect the AC power cord to the power plug. Plug the power supply adapter cable into the FPGA development board.
- 3 Connect the Ethernet connector on the FPGA development board directly to the Ethernet adapter on your computer using the crossover Ethernet cable.
- 4 Use the JTAG download cable to connect the FPGA development board with the computer.
- 5 Make sure that all jumpers on the FPGA development board are in the factory default position, except for Microchip PolarFire, which requires special settings. See “Installing Microchip Polarfire Evaluation Kit” (HDL Verifier Support Package for Microchip FPGA Boards).

Step 2: Set Up Host Computer-Board Connection

Skip this step if you are using JTAG connection for simulation. For connection with Ethernet, you must have a Gigabit Ethernet network adapter on your computer to run this example.

On Windows®, do the following steps:

- 1 Open the **Control Panel**.
- 2 Type **View network connections** in the search bar. Select **View network connections** in the search results.
- 3 Right click the connection icon to your FPGA development board and select **Properties** from the pop-up menu.
- 4 Under **This connection uses the following items**, select **Internet Protocol Version 4 (TCP/IPv4)** and click **Properties**.
- 5 Select **Use the following IP address:**. Set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This is your host computer address. Set the **Subnet mask** to 255.255.255.0. Your TCP/IP properties should now look the same as in the following figure:



On Linux®:

Use the **ifconfig** command to set up your local address. For example:

```
% ifconfig eth1 192.168.0.1
```

In this example, eth1 is the second Ethernet adapter on the Linux computer. Check your system to determine which Ethernet adapter is connected to the FPGA development board. The above command sets the local IP address to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100.

Step 3: Prepare Example Resources

1. Set Up FPGA design software

Before using FPGA-in-the-Loop, set up your system environment for accessing FPGA design software. You can use the function **hdlsetuptoolpath** to add ISE, Vivado, Quartus, or Libero SoC design software to the system path for the current MATLAB session. For information about supported versions of FPGA design software, see “FPGA Verification Requirements”. Example command lines for each tool are given below. Substitute with your actual executable if it is different.

For Xilinx FPGA boards using ISE design software, run:

```
hdlsetuptoolpath('ToolName','Xilinx ISE','ToolPath','C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\ise.exe')
```

For Xilinx FPGA boards using Vivado design software, run:

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

For Intel boards, run:

```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\altera\20.1.1\quartus\bin\quartus
```

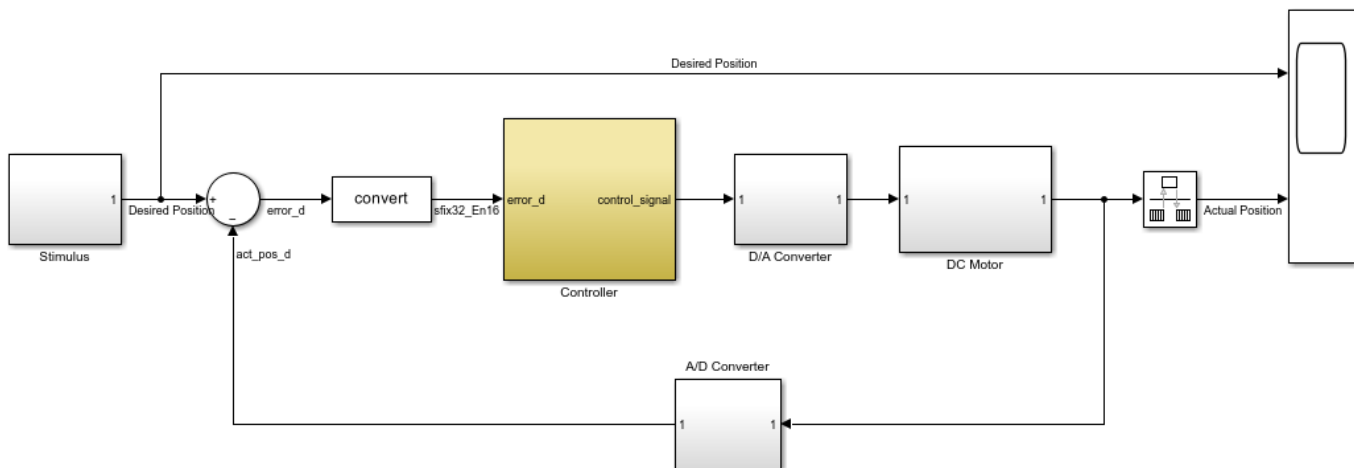
For Microchip boards, run:

```
hdlsetuptoolpath('ToolName','Microchip Libero SoC','ToolPath','C:\Microsemi\Libero_SoC_v12.0\Des
```

2. Open the fil_pid model.

This model contains a fixed-point PID controller implemented with basic Simulink blocks. This model also contains a DC motor model controlled by this PID controller as well as the desired DC motor position as the input stimulus.

Run this model now and observe the desired and actual motor positions in the scope.

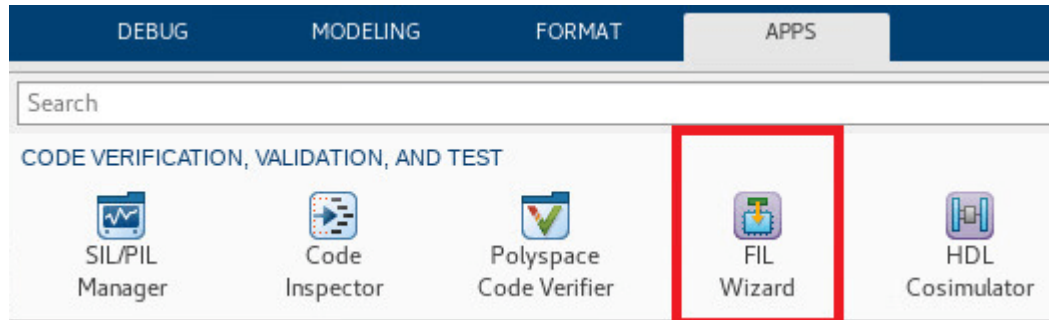


Copyright 2010-2011 The MathWorks, Inc.

Step 4: Launch FPGA-in-the-Loop (FIL) Wizard

Launch the FPGA-in-the-Loop Wizard by doing the following:

Open the Apps gallery and select FIL Wizard from the Code Verification, Validation, and Test section.



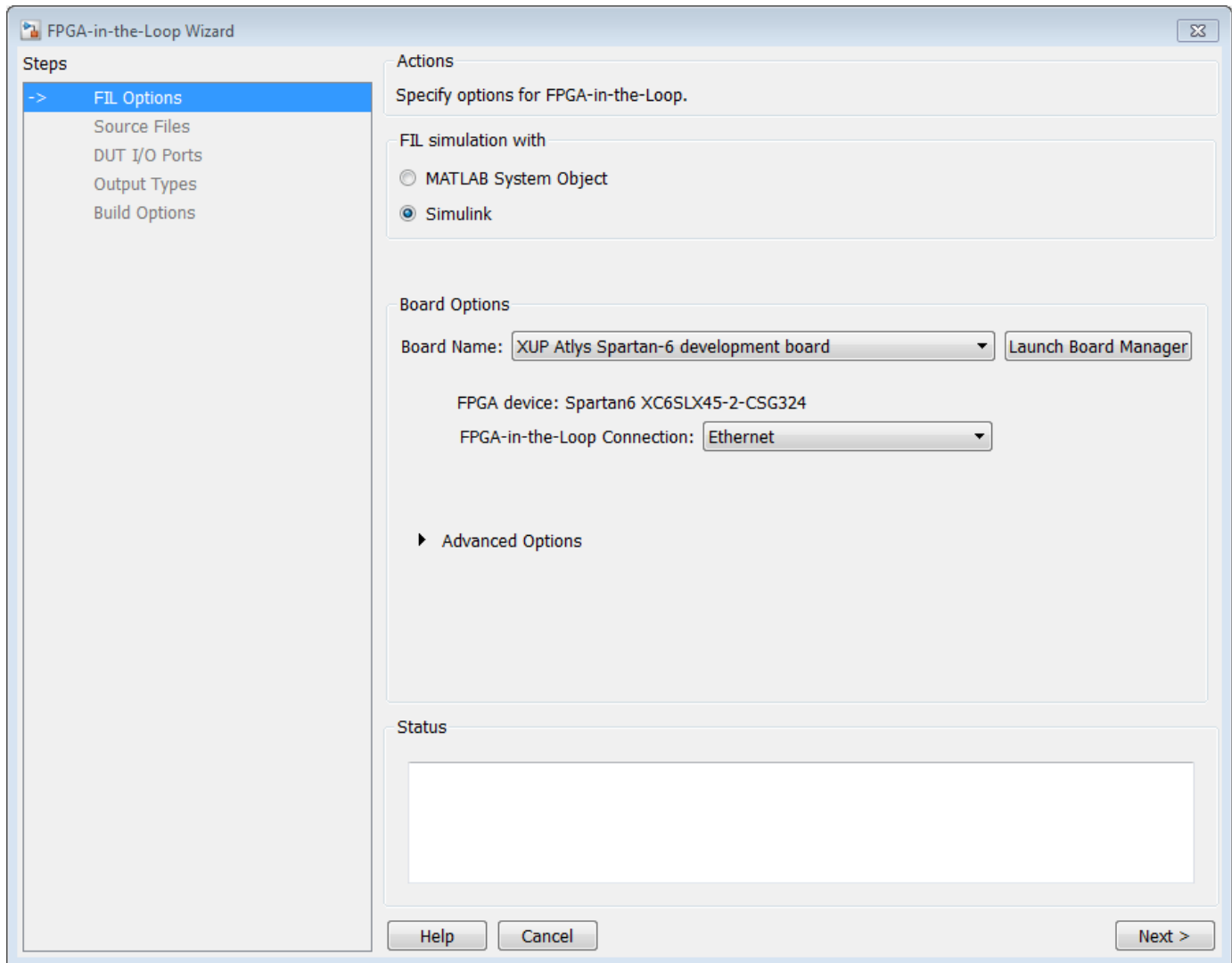
Alternatively, you can enter the `filWizard` command at the MATLAB command prompt.

```
filWizard
```

Step 5: Specify Hardware Options in FIL Wizard

Set the FIL options for the FPGA development board.

1. Specify if the wizard will generate a FIL Simulink block or a FILSimulation MATLAB System Object. For this example, select **Simulink** for **FIL simulation with** Simulink.
2. For **Board Name**, select the FPGA development board connected to your host computer. If your board is not on the list, select one of the following options:
 - "Get more boards..." to download the FPGA board support package(s) (this option starts the Support Package Installer).
 - "Create custom board..." to create the FPGA board definition file for your particular FPGA board (this option starts the New FPGA Board Manager).



3. Select the connection for simulation. The available connection methods are Ethernet and JTAG. Not all boards support both connection methods.

4. Ethernet connection only: If you changed your computer's IP address to a different subnet from 192.168.0.x when you set up the network adapter, or if the default board IP address 192.168.0.2 is in use by another device, expand **Advanced Options** and change the **Board IP address** according to the following guidelines:

- The subnet address, typically the first three bytes of board IP address, must be the same as those of the host IP address.
- The last byte of the board IP address must be different from that of the host IP address.
- The board IP address must not conflict with the IP addresses of other computers.

For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3 if it is available. Do not change **Board MAC address**.

▼ **Advanced Options**

Board IP address: 192 . 168 . 8 . 3

Board MAC address: 00 - 0A - 35 - 02 - 21 - 8A

FPGA system clock frequency (MHz) 25

5. Optional: If you would like to change the DUT clock frequency from the default (25MHz), you can expand **Advanced Options** and change the **FPGA system clock frequency (MHz)**.

6. Click **Next** to continue.

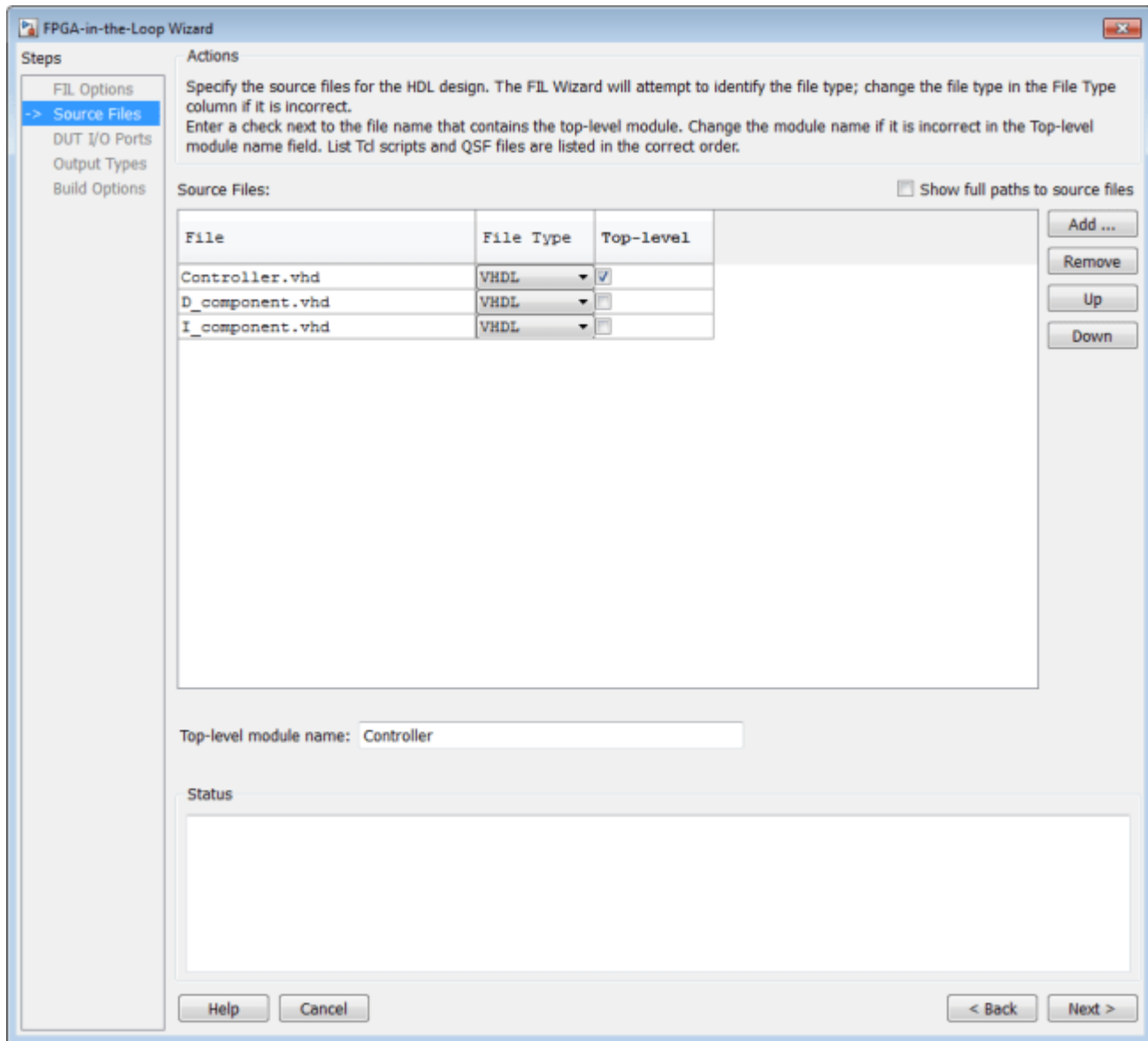
Step 6: Specify HDL Files in the FIL Wizard

Specify the HDL design to be implemented in the FPGA.

1. Click Add and browse to the directory you created in Prepare Example Resources.
2. Select these HDL files in the pid_hdlsrc directory:
 - Controller.vhd
 - D_component.vhd
 - I_component.vhd

These are the HDL design files to be verified on the FPGA board.

3. In the **Source Files** table, check the checkbox on the row of file **Controller.vhd** to specify that this HDL file contains the top-level HDL module.



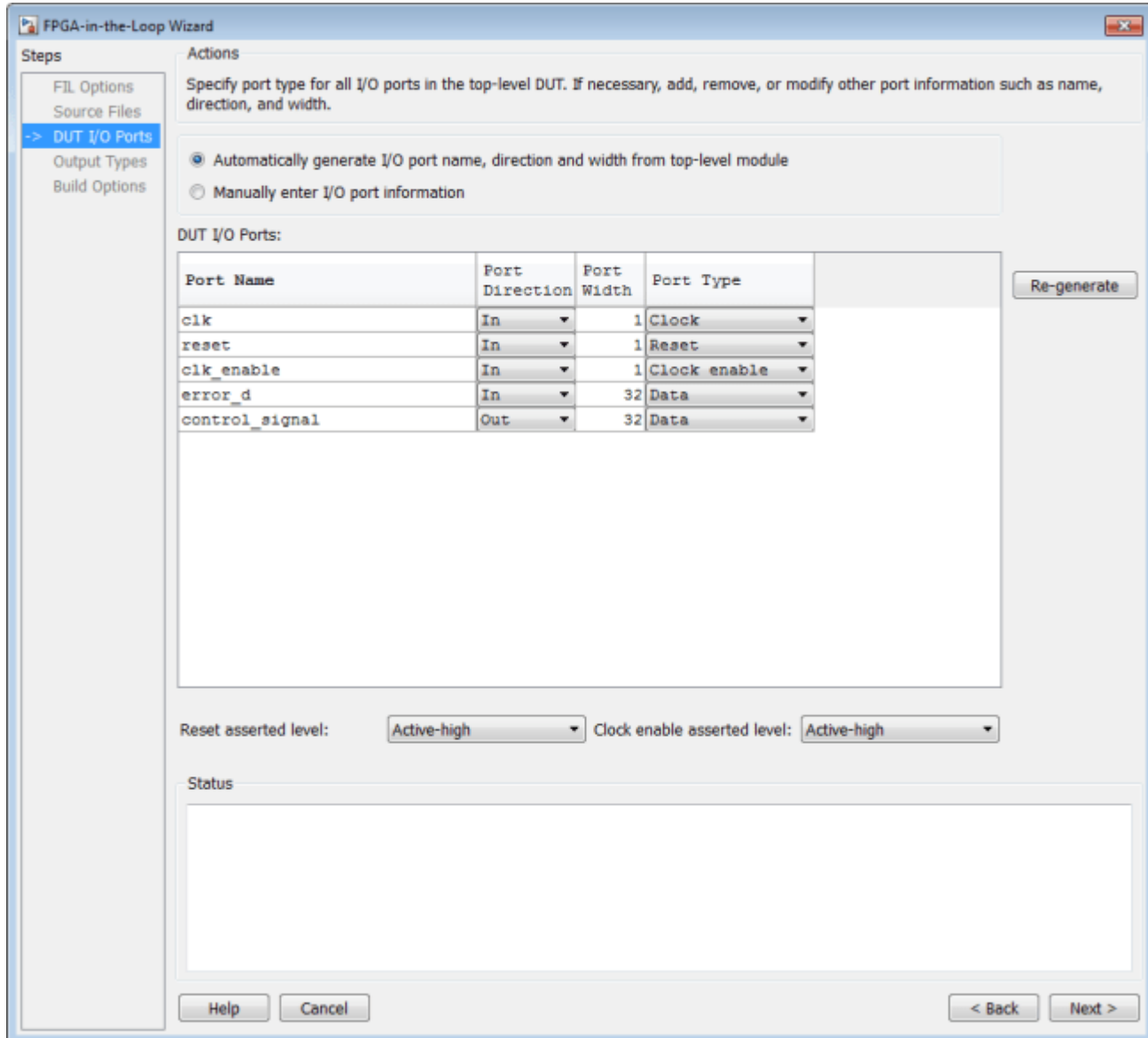
The FIL Wizard automatically fills the **Top-level module name** field with the name of the selected HDL file; in this case, **Controller**. In this example, the top-level module name matches the file name so that you do not need to change it. If the top-level module name and file name did not match, you would manually correct the top-level module name in this dialog.

Click **Next** to continue.

Step 7: Review I/O Ports in FIL Wizard

The FIL Wizard parses the top-level HDL module Controller in Controller.vhd to obtain all the I/O ports and display them in the DUT **I/O Ports** table. The parser attempts to automatically determine the possible port types by looking at the port names and displays these signals under Port Type.

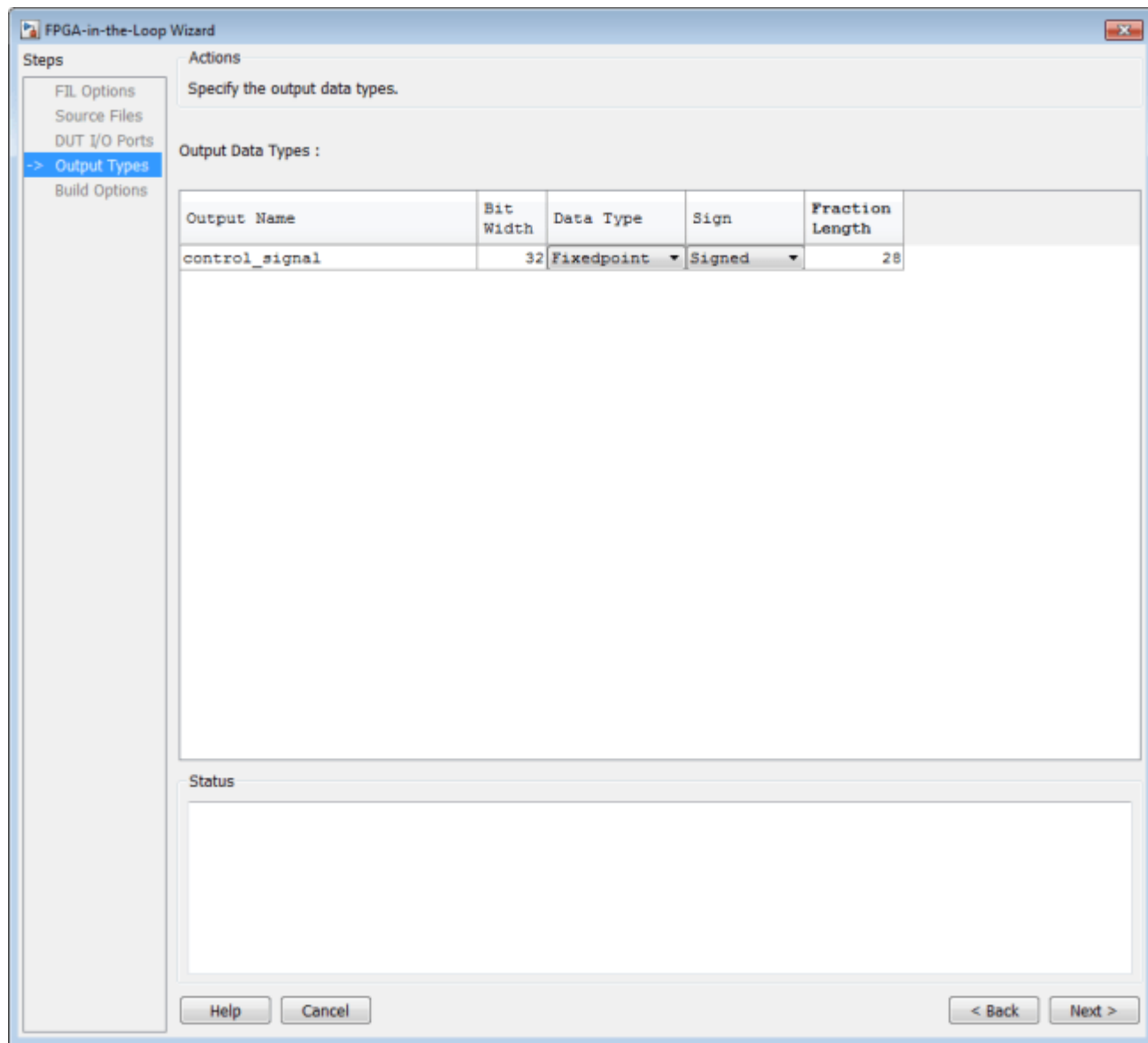
1. Review the port listing. If the parser assigned an incorrect port type for any given port, you can manually change the signal. For synchronous design, specify a Clock, Reset, or Clock enable signal. In this example, the FIL Wizard automatically fills the table correctly.



2. Click **Next** to continue.

Step 8: Set Output Data Types in FIL Wizard

1. For the HDL output **control_signal** change **Data Type** to **Fixedpoint**, **Sign** to **Signed** and **Fraction Length** to **28**. This will make the generated FIL block set the output signal of the FPGA design-under-test (DUT) to the correct data type.

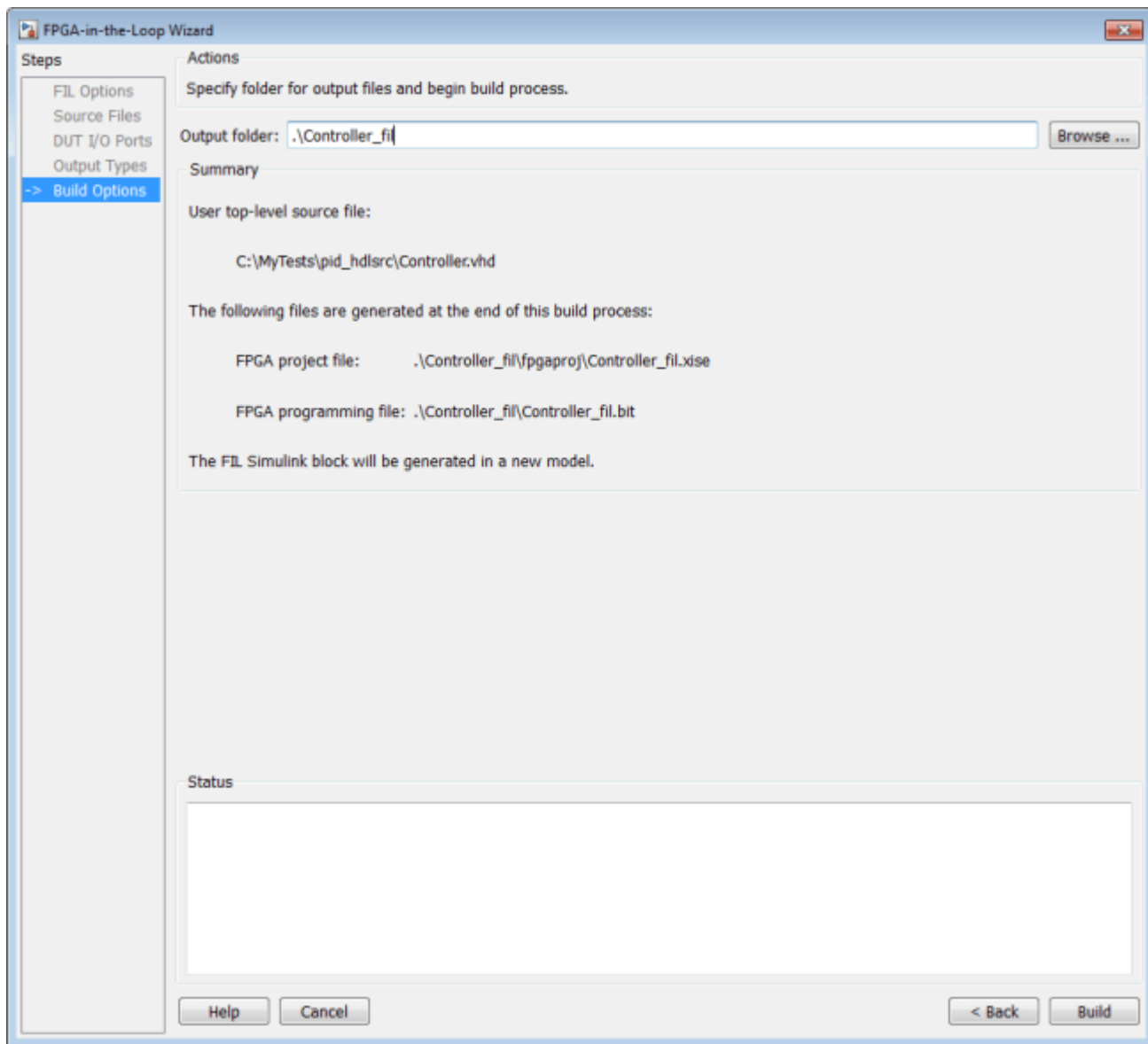


2. Click **Next** to continue.

Step 9: Review Build Options in FIL Wizard

1. Specify the folder for the output files. For this example, use the default option, which is a subfolder named **Controller_fil** under the current directory.

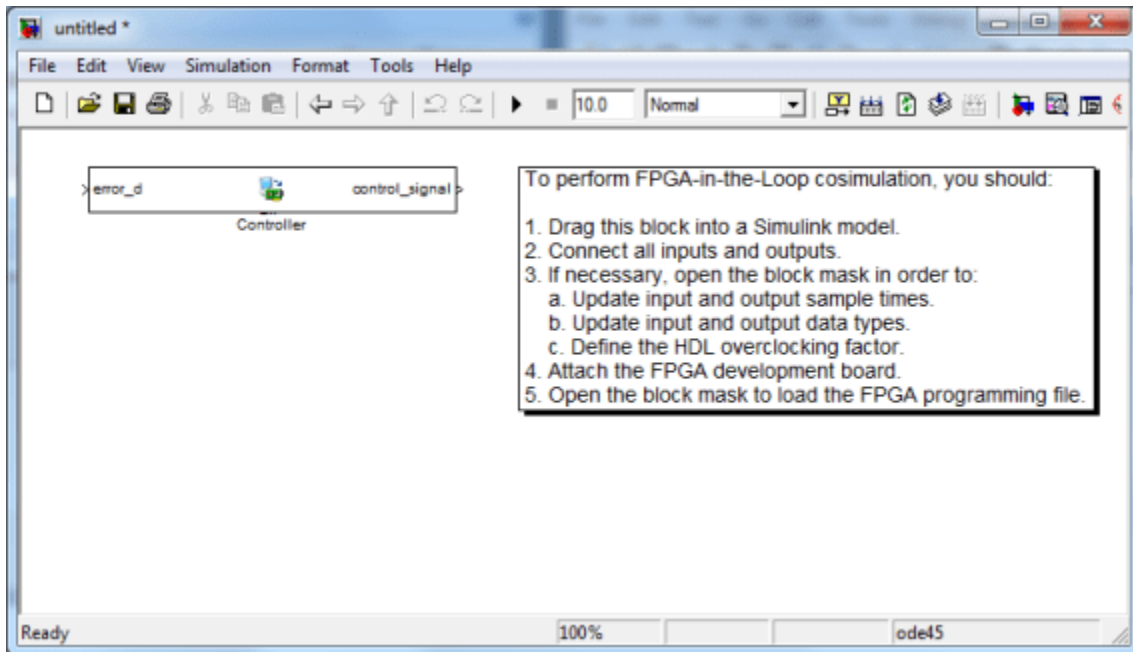
The **Summary** displays the locations of the FPGA project file and the FPGA programming file. You may need those two files for advanced operations.



2. Click **Build** to start the build process.

During the build process, the following actions occur:

- A FIL block named Controller is generated in a new model as shown in the following figure. Do not close this model.



- After new model generation, the FIL Wizard opens a command window where the FPGA design software performs synthesis, fit, place-and-route, timing analysis, and FPGA programming file generation.
- When the FPGA design software process is finished, a message in the command-line window lets you know you can close the window. Close the window and proceed to the next step.

```

Process "Generate Programming File" completed successfully
INFO:TclTasksC:1850 - process run : Generate Programming File is done.

Programming file generated:
C:\MyTests\Controller_fil\Controller_fil.bit

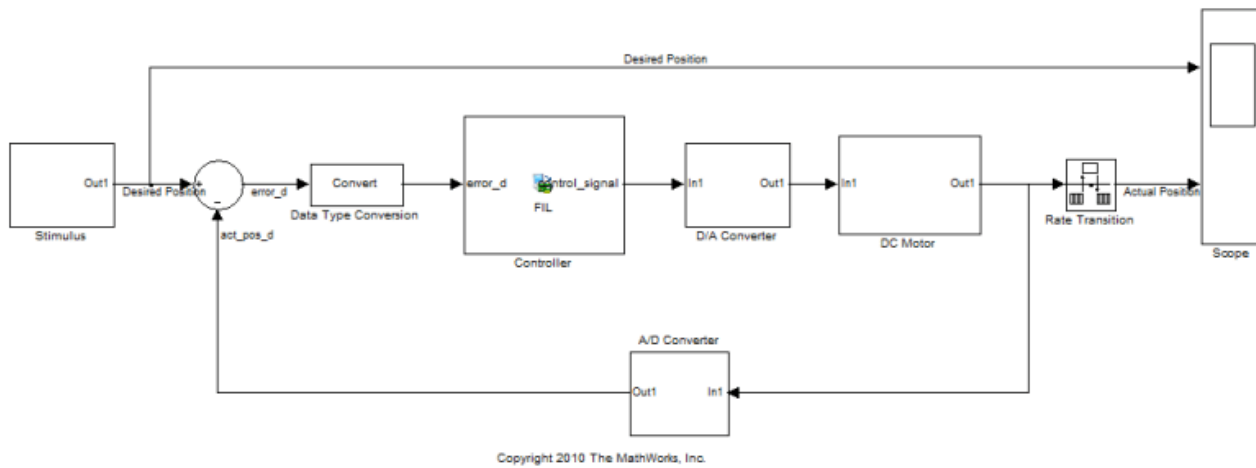
FPGA-in-the-Loop build completed.
You may close this shell.

C:\MyTests\Controller_fil\fpgaproj>

```

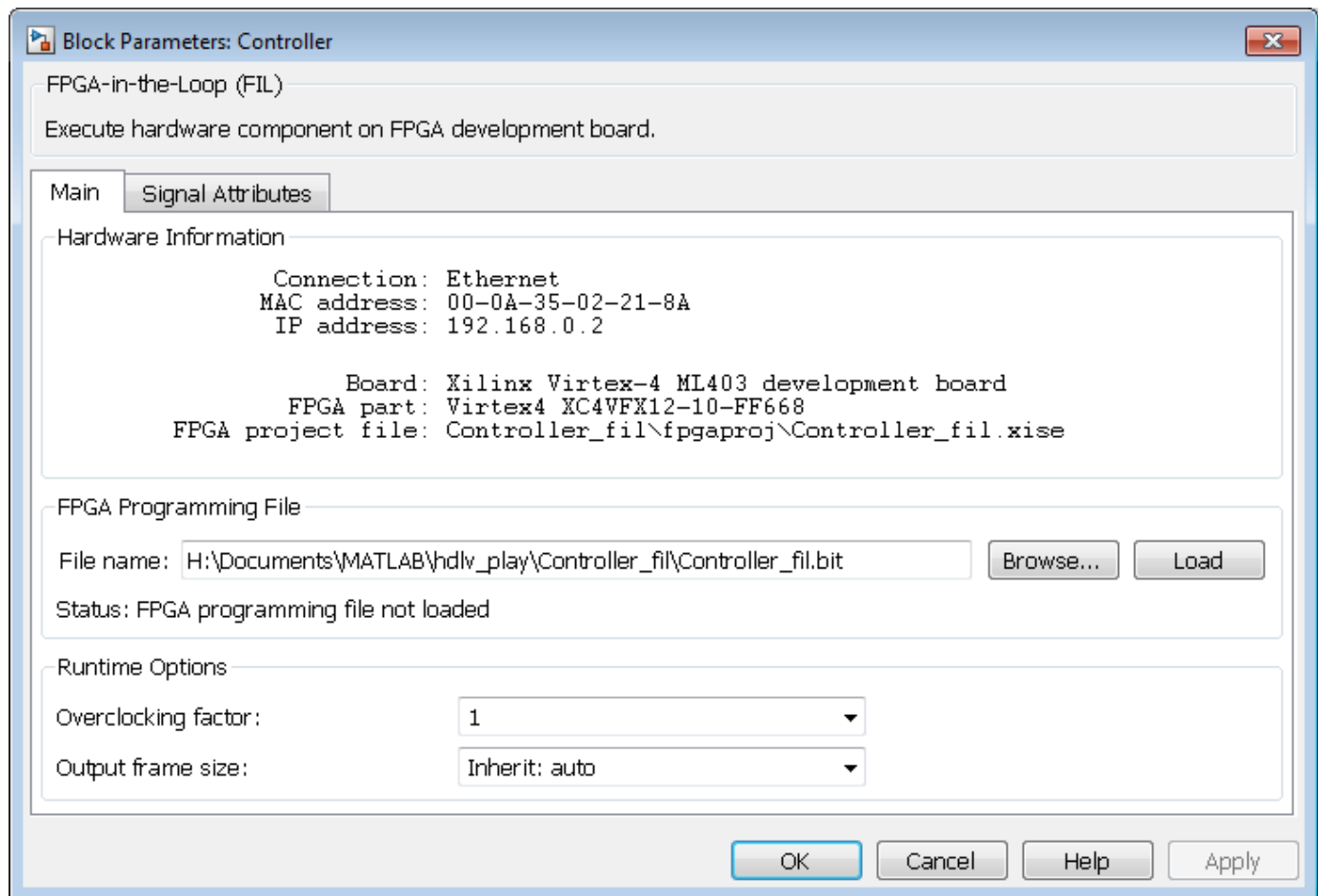
Step 10: Set Up Model

In the `fil_pid` model, replace the **Controller** subsystem with the FIL block generated in the new model. The modified `fil_pid` model now appears as shown in the following illustration:

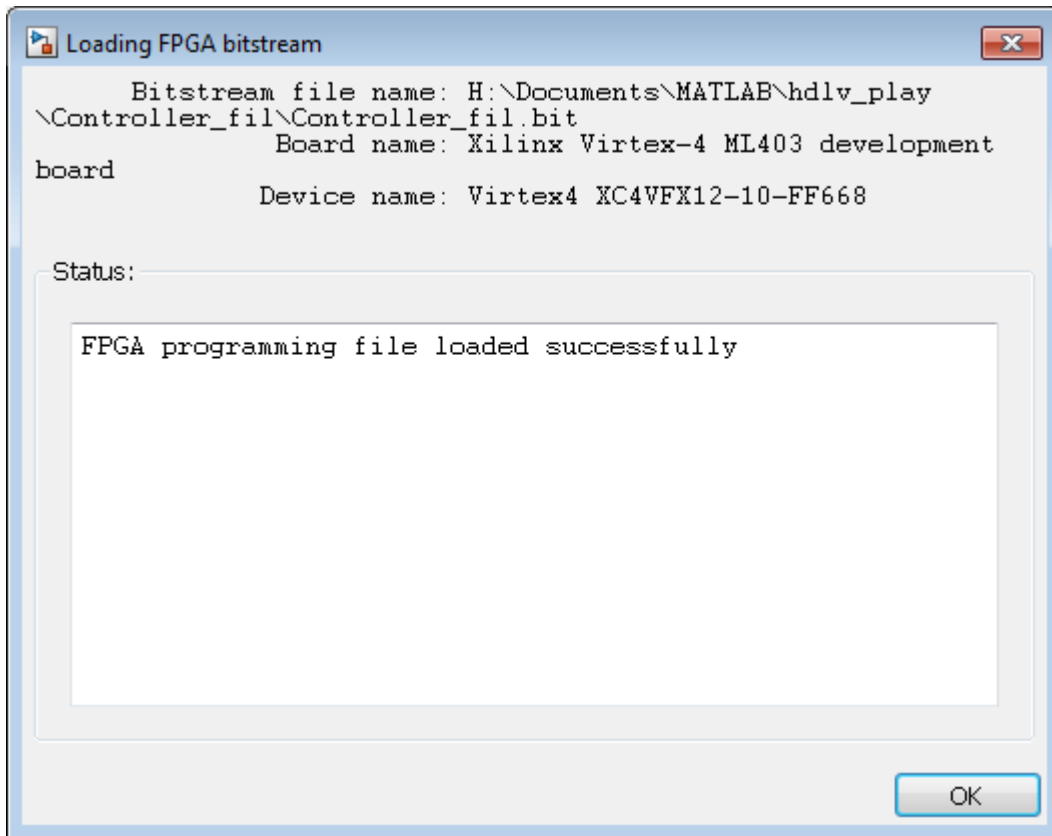


Step 11: Program FPGA

1. Switch FPGA development board power **ON**.
2. Double-click the FIL block in the fil_pid model to open the block mask.
3. In the opened block mask, click **Load**.



If your board is connected to the host computer through the JTAG cable properly, a message window displays to indicate that the FPGA programming file is loaded successfully. Click **OK** to dismiss this dialog.



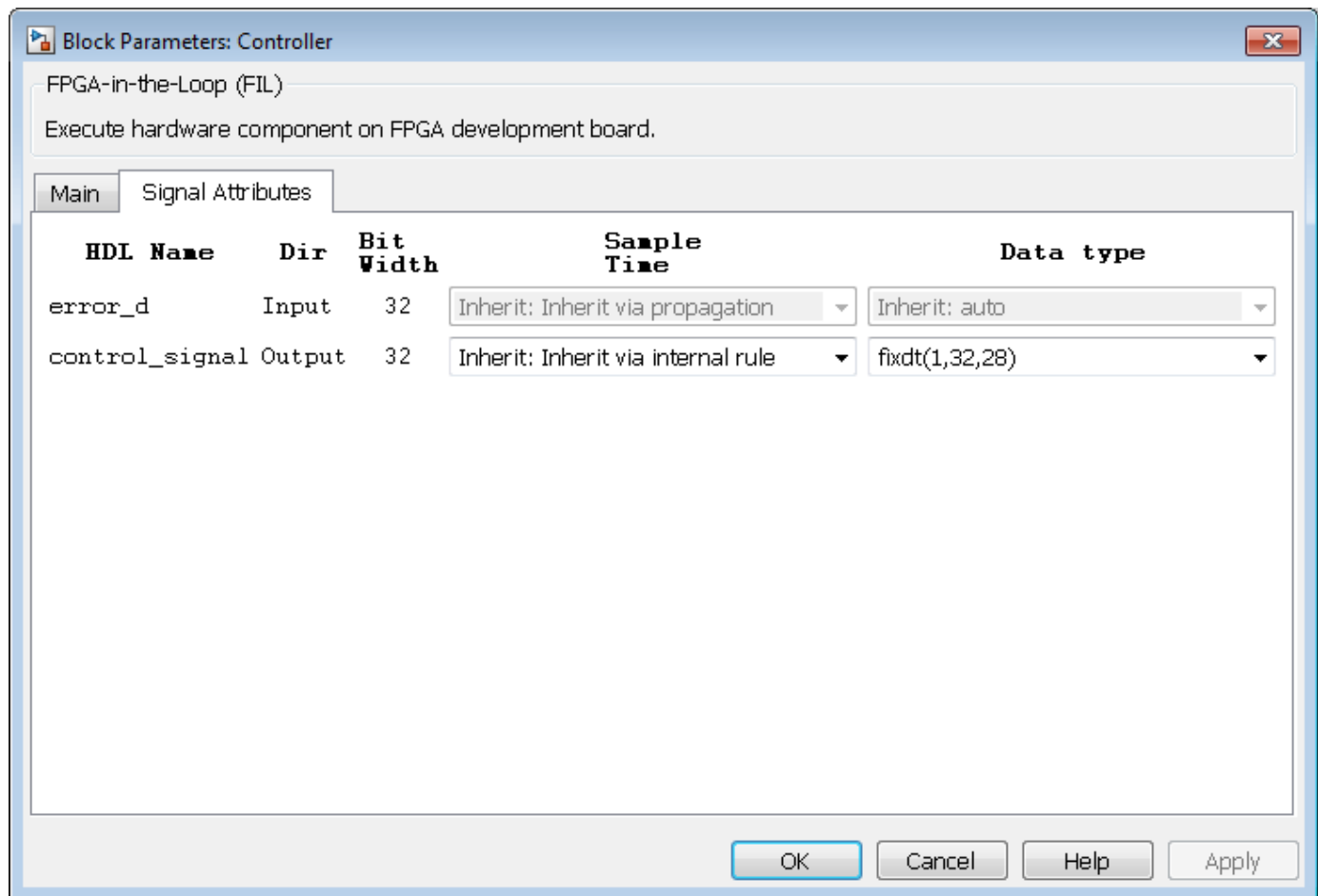
4. Ethernet connection only: You can test if the FPGA board is connected to your host computer properly through the ping test. Launch a command-line window and enter the following command:

```
C:\MyTests> ping 192.168.0.2
```

If you changed the board IP address when you set up the network adapter, replace 192.168.0.2 with your board IP address. If the Gigabit Ethernet connection has been set up properly, you should see the ping reply from the FPGA development board.

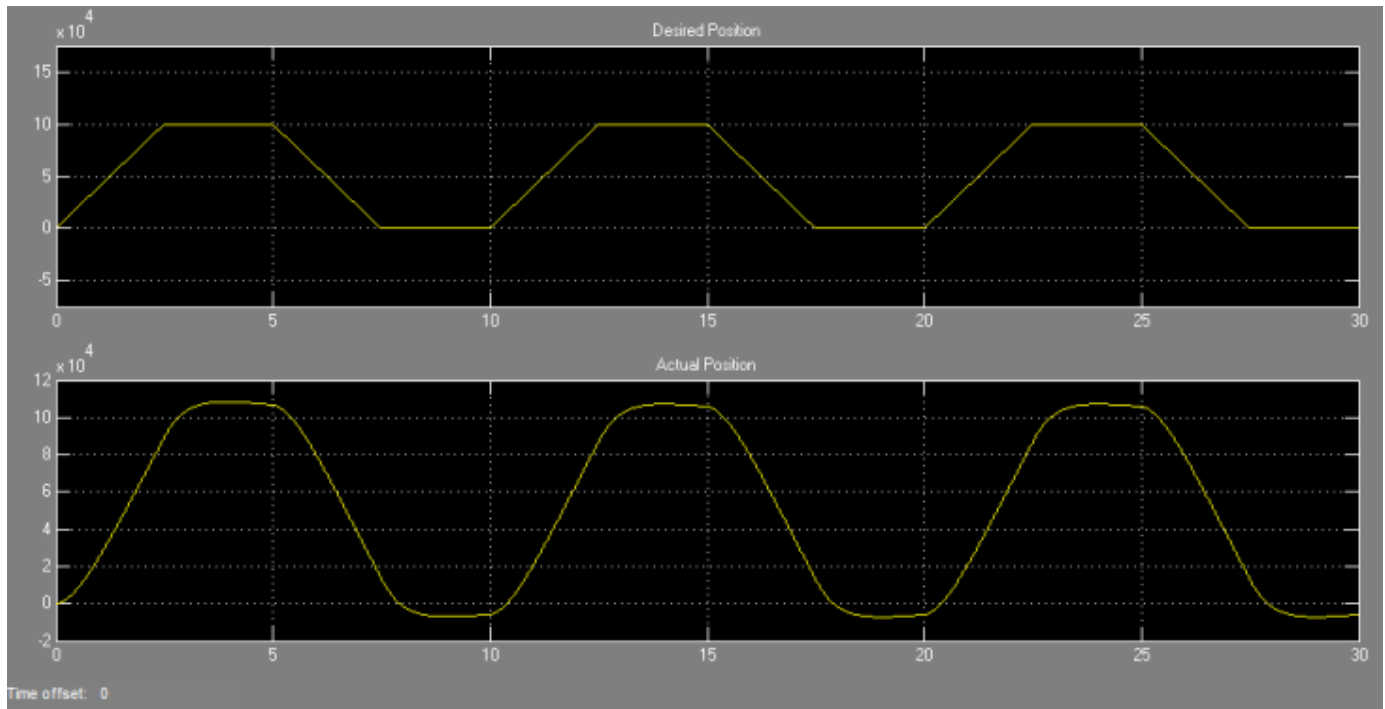
Step 12: Review Parameters of FIL Block

1. In the FIL block mask, click the **Signal Attributes** tab.
2. Verify that the **Data Type** of the HDL signal **control_signal** is **fixdt(1,32,28)**. If it is not, change it.
3. Click **OK** to close the block mask.



Step 13: Run FIL

1. Start simulation of the `fil_pid` model.
2. When the simulation is done, view the waveform of the desired and actual positions of the motor in the scope. Note that the results of FIL simulation should match those of the Simulink reference model that you simulated in **Prepare Example Resources**.



Cosimulation and FPGA-in-the-Loop in MATLAB-to-HDL Workflow

This example shows how to verify generated HDL code using HDL Cosimulation and FPGA-in-the-Loop as steps in the HDL code generation workflow for MATLAB® to HDL.

Requirements and Prerequisites

Products required for this example:

- MATLAB
- Fixed-Point Designer™
- HDL Verifier™
- FPGA design software (Xilinx® ISE® or Vivado® design suite or Intel® Quartus® II design software)
- One of the “Supported FPGA Devices for FPGA Verification” and accessories
- For connection using Ethernet: Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable
- For connection using JTAG: USB Blaster I or II cable and driver for Intel FPGA boards. Digilent® JTAG cable and driver for Xilinx FPGA boards.

Prerequisites:

MATLAB and FPGA design software can either be locally installed on your computer or on a network accessible device. If you use software from the network you will need a second network adapter installed in your computer to provide a private network to the FPGA development board. Consult the hardware and networking guides for your computer to learn how to install the network adapter.

MATLAB Design

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

Let us take a look at the MATLAB design.

```
type('mlhdlc_sysobj_ex.m');
```

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

% Copyright 2011-2015 The MathWorks, Inc.

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

```

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

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

a1 = h1p + h8p;
a2 = h2p + h7p;
a3 = h3p + h6p;
a4 = h4p + h5p;

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

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

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

end

type('mlhdlc_sysobj_ex_tb.m');

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

% Copyright 2011-2015 The MathWorks, Inc.

clear mlhdlc_sysobj_ex;

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

h1 = -0.1339;
h2 = -0.0838;

```

```
h3 = 0.2026;
h4 = 0.4064;

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

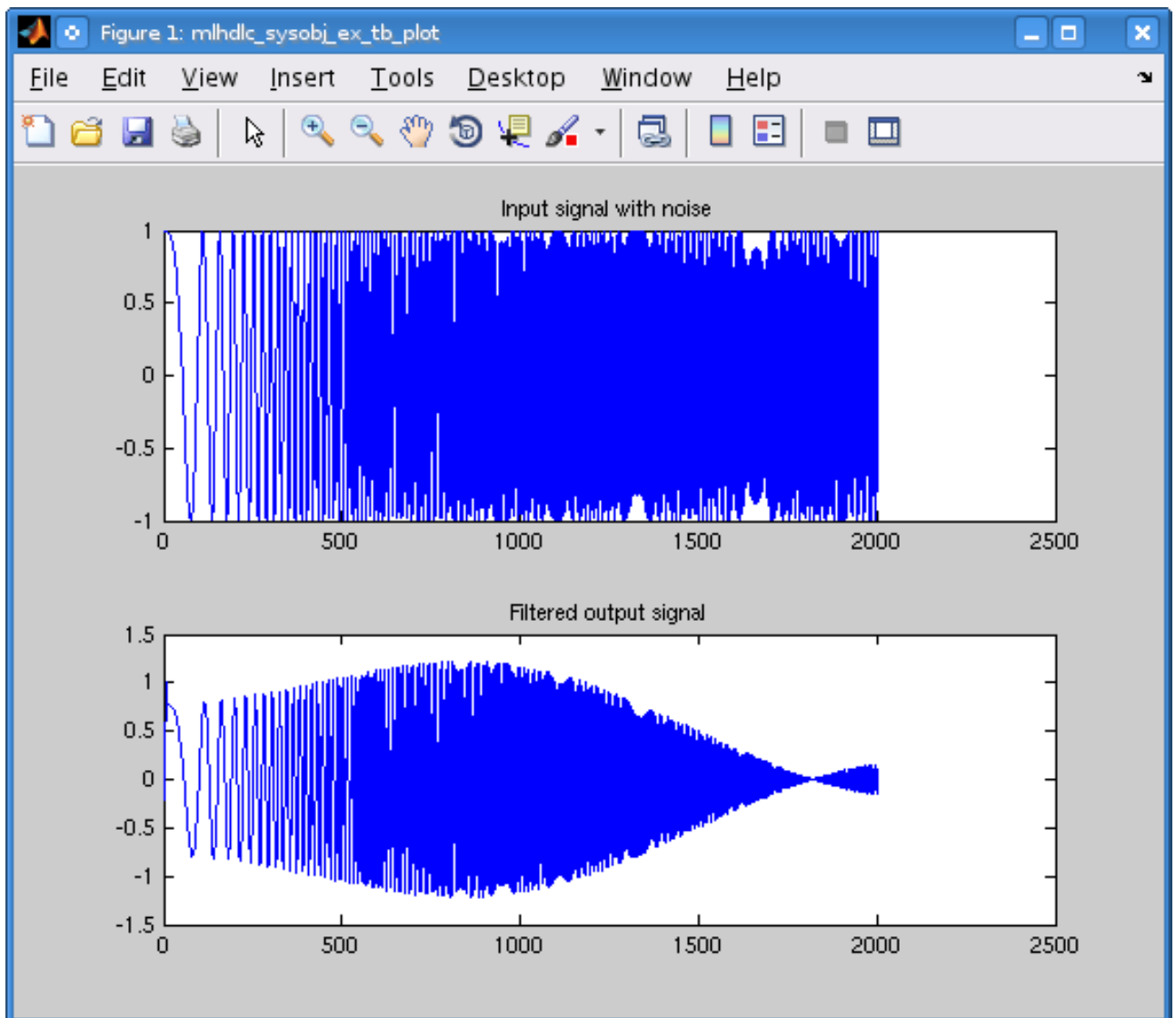
for ii=1:len
    data = x_in(ii);
    % call to the design 'sfir' that is targeted for hardware
    [y_out_sysobj(ii), x_out_sysobj(ii)] = mlhdlc_sysobj_ex(data, h1, h2, h3, h4);
end

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

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors. Run this command in MATLAB.

```
mlhdlc_sysobj_ex_tb
```

Create a New HDL Coder™ Project

To create a new project, enter the following command:

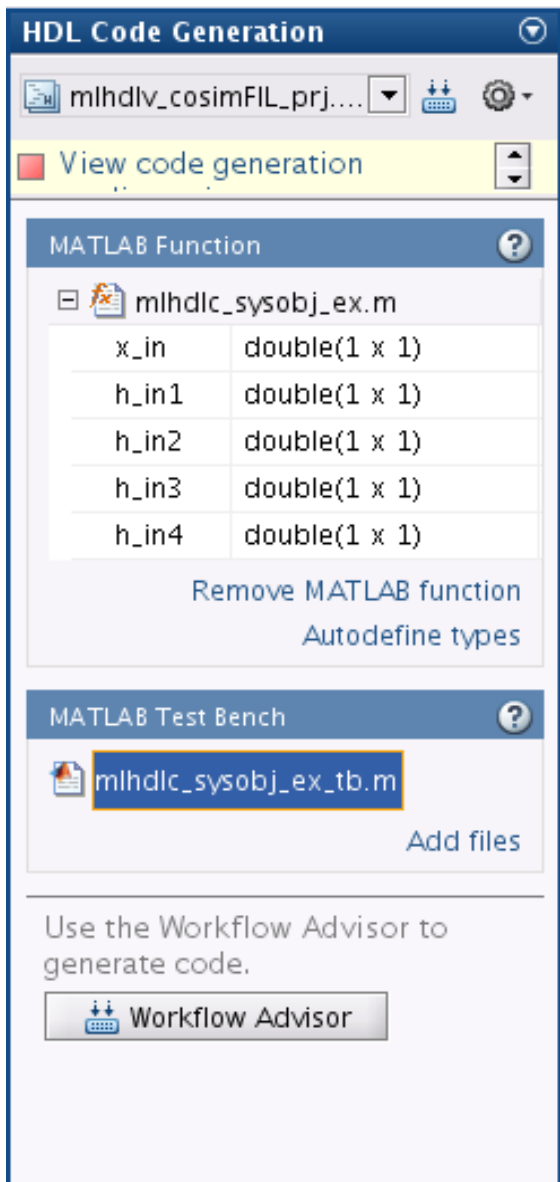
```
coder -hdlcoder -new mlhdlv_cosimFIL_prj
```

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

You can refer to the “Get Started with MATLAB to HDL Workflow” (HDL Coder) tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Define Input Data Types

For this floating point MATLAB System object all inputs are Double Scalar. Specify them as such using the drop-down menus provided for the MATLAB Function.



Run Fixed-Point Conversion and HDL Code Generation

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

Examine the generated HDL code by clicking the links in the log panel of the Workflow Advisor.

Generate Cosimulation Test Bench and Run HDL Cosimulation

Select the "Verify with Cosimulation" step.

Select the checkbox labelled "Generate cosimulation test bench". That action will enable other choices in the dialog, allowing you to elect to log outputs for comparison, choose your preferred HDL simulator and simulate the generated cosimulation test bench. Check both of the remaining checkboxes and select your preferred HDL simulator.

Select "GUI" as the HDL simulator run mode.

Generate and run cosimulation test bench

Output Settings \ Advanced Options \

Cosimulation Test Bench Generation Settings

- Generate cosimulation test bench
- Log outputs for comparison plots

Cosimulate for use with: Mentor Graphics ModelSim ▼

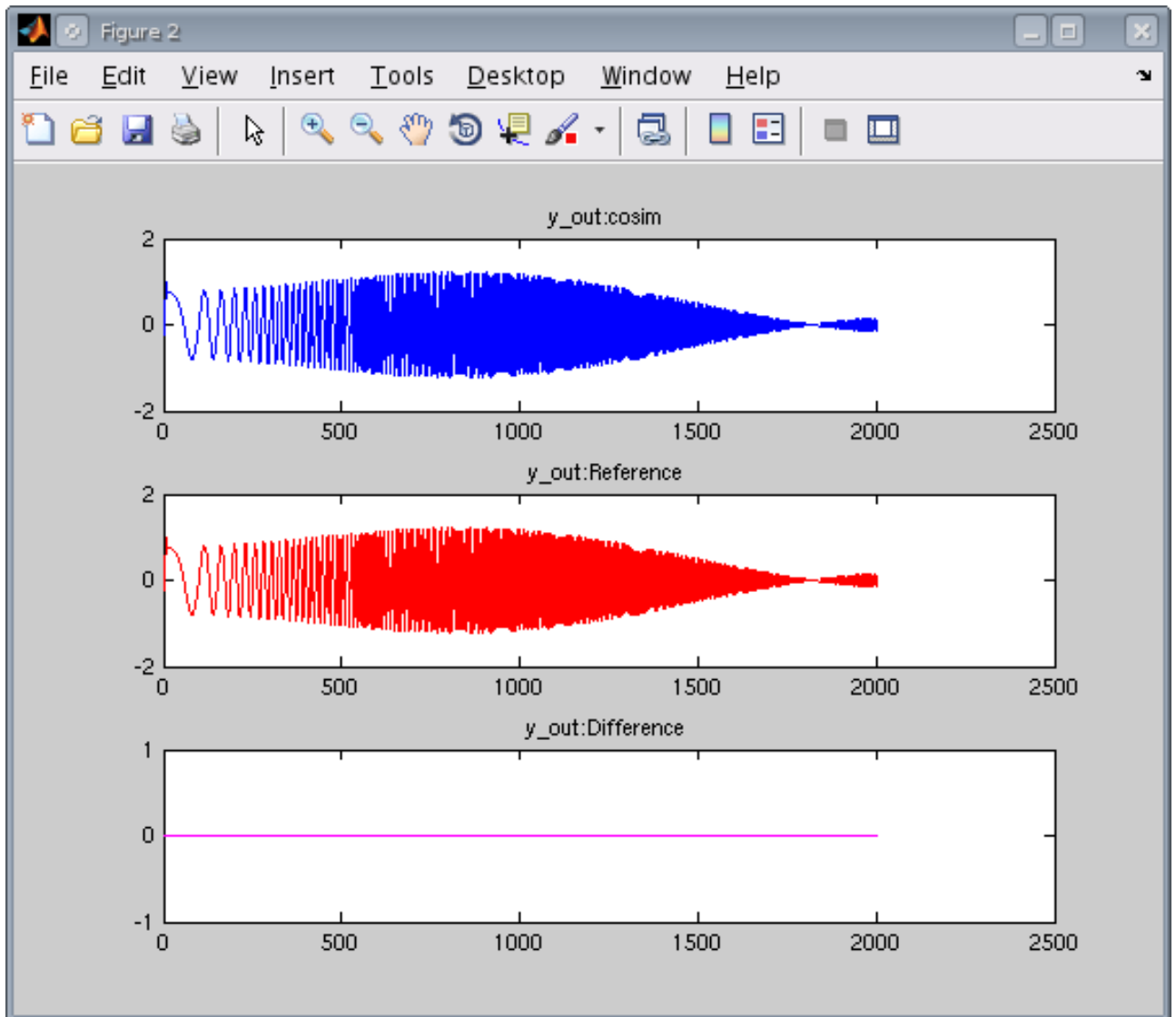
HDL simulator run mode in cosimulation: GUI ▼

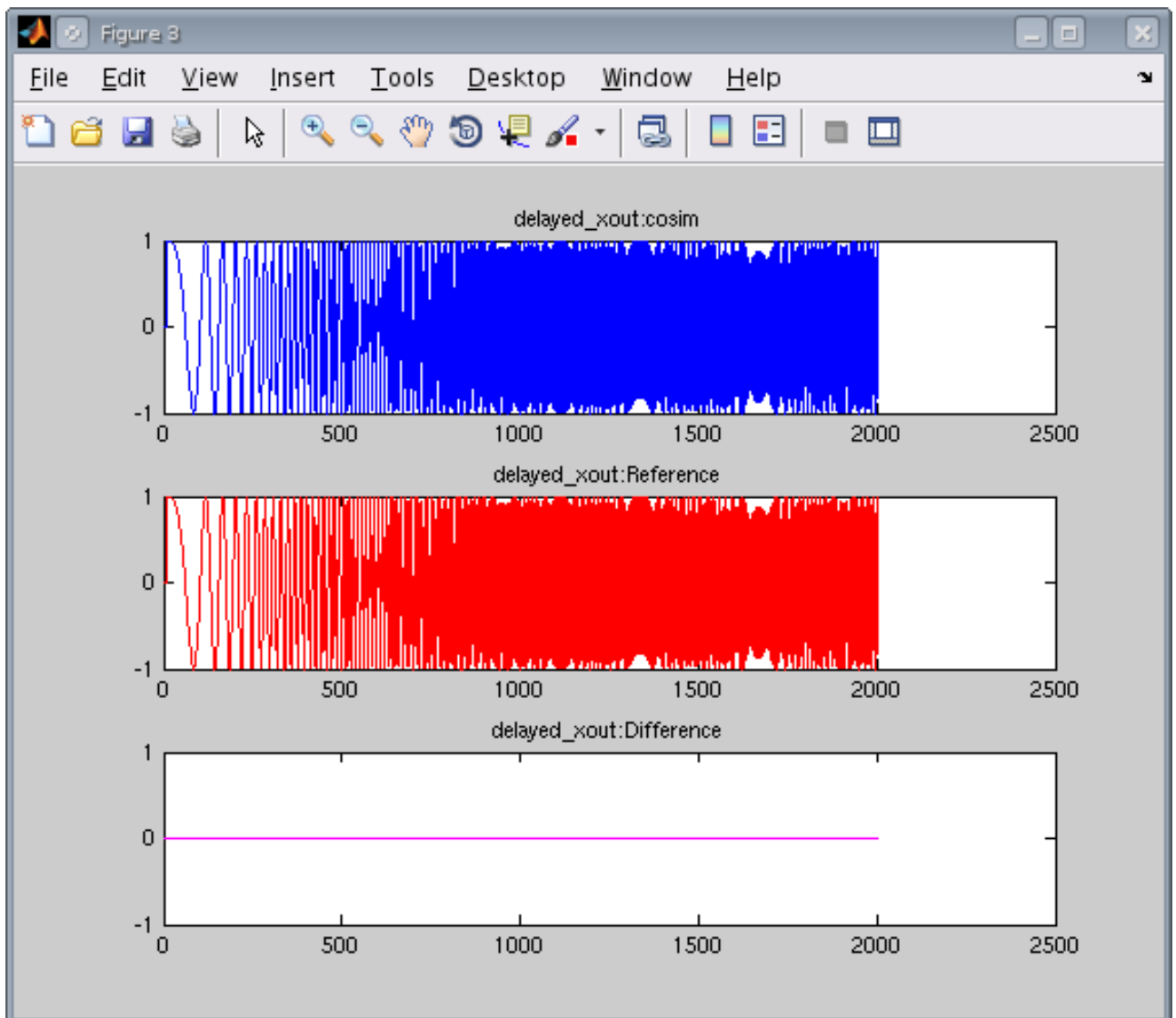
Cosimulation Test Bench Simulation Settings

- Simulate generated cosimulation test bench

Skip this Step ? Run

Click "Run" and observe the plots comparing the outputs of the cosimulation to the outputs of the fixed point System object. Also note the signal transitions in the HDL simulator waveform viewer.





Set Up Your FPGA Development Board

Refer to Set Up FPGA Development Board for information on setting up your FPGA board and computer to communicate for FPGA-in-the-Loop simulation.

Generate FPGA-in-the-Loop (FIL) Test Bench and Run FIL Simulation

Select the "Verify with FPGA-in-the-Loop" step in the left side panel of the Workflow Advisor.

Select all 3 checkboxes (for FPGA-in-the-Loop test bench generation, output logging, simulation of the generated FIL test bench).

Select the FPGA development board of your choice.

Generate and run FPGA-in-the-Loop test bench

FPGA-in-the-Loop Test Bench Generation Settings

Generate FPGA-in-the-Loop test bench

Log outputs for comparison plots

Board Name: [Launch Board Manager](#)

Board IP Address:

FPGA Board MAC Address:

Additional Files: ...

FPGA-in-the-Loop Test Bench Simulation Settings

Simulate generated FPGA-in-the-Loop test bench

Skip this Step

Click "Run" and observe the plots comparing the outputs of the FPGA to the outputs of the fixed point System object.

Clean up the Generated Files

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

```
clear mex;  
cd (hdlverif_demo_dir);  
rmdir(mlhdlv_temp_dir, 's');
```

Verify the Combination of Hand-Written and Generated HDL Code

This example uses HDL cosimulation and FPGA-in-the-loop (FIL) simulation to verify an HDL design comprising generated and legacy HDL code. The term "legacy" is used here to indicate code that may have been hand-written, purchased from a third party or generated for another project and saved for reuse in this design.

The legacy code in this example implements a finite state machine (FSM) that is a sub-module of a Multiple Input - Multiple Output (MIMO) decoder intended for use in a wireless communications system. Most of the MIMO decoder has been developed in Simulink and the HDL code for it will be generated by HDL Coder. The FSM belongs inside that Simulink design. The legacy code for the FSM will be integrated with the Simulink model and incorporated into the FPGA implementation via the code generation process.

The example will show how the designer or verification engineer can use the HDL Verifier cosimulation wizard to integrate the legacy FSM with the Simulink model and verify it. HDL Cosimulation provides full visibility and control, enabling debugging and verification of the code.

After successful integration of the legacy FSM, the cosimulation block automatically incorporates the legacy code when HDL code is generated from the Simulink model, resulting in a complete FPGA implementation of the MIMO decoder. Finally, the entire design is verified on the actual FPGA using FPGA-in-the-loop.

Outline of the Example

- 1 Use the cosimulation wizard to import legacy HDL code into a Simulink model
- 2 Verify the legacy HDL code by cosimulating it and comparing results with a behavioral model
- 3 Generate HDL code for the entire MIMO decoder using the cosimulation block in a Blackbox
- 4 Validate the MIMO decoder with FPGA-in-the-loop

Requirements and Prerequisites

For cosimulation and FPGA-in-the-loop you'll need the following software and hardware:

- One of the supported HDL simulators. For supported simulators see "Cosimulation Requirements".
- FPGA design software
- One of the supported FPGA development boards. For supported hardware, see "Supported FPGA Devices for FPGA Verification".
- For connection using Ethernet: Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable
- For connection using JTAG: USB Blaster I or II cable and driver for Intel FPGA boards. Digilent® JTAG cable and driver for Xilinx FPGA boards.

MATLAB® and FPGA design software can either be locally installed on your computer or on a network accessible device. If you use software from the network you will need a second network adapter installed in your computer to provide a private network to the FPGA development board. Consult the hardware and networking guides for your computer to learn how to install the network adapter.

Note: The example includes code generation. If you do not have access to HDL Coder software you can skip the code generation step in this example and use the HDL files provided for you together with the FIL wizard to simulate them with FPGA-in-the-loop.

Create a Reference Model for the Finite State Machine

A reference model is a simulation model of the behavior expected of an implementation. It is typically used in HDL verification by instantiating it alongside the RTL implementation, giving the same inputs to both and comparing their outputs. The advantages of reference models in verification are that they can be developed independently of the implementation (often by a different person) providing an independent validation of the expected behavior; they are easier to create than the actual implementation (not requiring synthesis or actual device timing) and they usually run fast in simulation.

The first step in verifying the legacy HDL code in this example is to create a reference model for that part of the design. That has already been done for the FSM. Open the behavioral_mimo.slx model. Double-click into the MIMO Decoder subsystem and you'll see that the FSM subsystem contains a MATLAB function block that implements the behavior of the FSM. This reference model will be used to verify the legacy HDL code for the FSM.

1. Use Cosimulation Wizard to Import Legacy HDL Code

Invoke the cosimulation wizard by typing the following at the MATLAB command prompt:

```
cosimWizard
```

Select HDL cosimulation with Simulink and your preferred HDL simulator from the drop-down list. If the HDL simulator is not on your system path, provide the path and click Next.

Add FSMSubsystem.vhd, FSMSubsystem_pkg.vhd, and Embedded_Controller.vhd files (located in the "verify_legacy_hdlsrc" folder) using the cosimWizard's Add button and reorder the list to place FSMSubsystem.vhd at the bottom and FSMSubsystem_pkg.vhd at the top of the list, for correct compilation ordering. Then click Next.

Click Next on the following 2 panels to accept the default values and arrive at the Input/Output Ports panel. In the list of Input Ports, select the following Port Type values from the drop-down lists for the first 3 ports:

```
clk          : Port Type = clock
reset       : Port Type = reset
clk_enable  : Port Type = reset
```

This identification of Port Types causes the cosimulation block to force those signals in the HDL simulator rather than require that they be driven in the Simulink diagram. In this example we treat the clk_enable port as another reset for cosimulation. Before you proceed to the next step similarly select "unused" for the ce_out, causing it to be omitted from the cosimulation block since it is not needed in Simulink.

The cosimulation wizard automatically identifies inputs and outputs in the HDL code and creates the cosimulation block for Simulink based on the ports it finds there. There are some details about the output ports that it cannot learn from the HDL code. In the HDL code the outputs are simply collections of bits with no indication of how you would like to interpret those bits in Simulink. You have to tell the cosimulation wizard whether you want those bits to be seen as signed or unsigned values and, if they are to be interpreted as fixed-point numbers, where to put the radix point.

In the Output Port Details panel refine the data type for each output. In the case of this design the output ports are to be interpreted as follows. Note that there are multiple scalar ports in the HDL code for the vector ports (out_1, out_6, out_9, out_10, out_11, out_12):

```

out_1  : Signed,   Fraction Length = 0 (4 scalar ports)
out_2  : Unsigned, Fraction Length = 0
out_3  : Unsigned, Fraction Length = 0
out_4  : Unsigned, Fraction Length = 0
out_5  : Signed,   Fraction Length = 10
out_6  : Signed,   Fraction Length = 10 (3 scalar ports)
out_7  : Signed,   Fraction Length = 2
out_8  : Unsigned, Fraction Length = 0
out_9  : Signed,   Fraction Length = 0 (4 scalar ports)
out_10 : Signed,   Fraction Length = 0 (4 scalar ports)
out_11 : Signed,   Fraction Length = 10 (4 scalar ports)
out_12 : Signed,   Fraction Length = 10 (4 scalar ports)
out_13 : Unsigned, Fraction Length = 0
out_14 : Signed,   Fraction Length = 0

```

On the Clock/Reset Details panel set the following values:

```

clk Period = 10 ns, Active Edge = Rising
reset Initial Value = 1, Duration = 27 ns
clk_enable Initial Value = 0, Duration = 37 ns

```

Click Next to proceed to the Start Time Alignment panel and set the "HDL time to start cosimulation (ns)" to 40.

Proceed to the final step and de-select the checkbox for "Automatically determine timescale at start of simulation". For this example we know that the timescale for cosimulation should be 1 second in Simulink corresponds to 10 ns in the HDL simulator. See HDL Verifier documentation for information on using the automatic timescale setting feature for other designs. Set the aforementioned timescale and click Finish.

The cosimulation block will be generated for importing the legacy HDL code into the Simulink model. You can drag and drop the newly generated cosimulation block and the 2 convenience command blocks into the Simulink model, inside the FSMSubsystem block and connect it to the output ports of the FSMSubsystem. A cosimulation model, with comparators and assertion blocks inside the MIMO Decoder subsystem, has been provided for this example. The comparators and assertion blocks have been added to alert you to any mismatches between the outputs of the reference model for the Embedded Controller and the legacy HDL implementation.

Use the following command to resize the generated cosimulation block to make it easier to insert it into the cosimulation model:

```
set_param('untitled/fsm subsystem', 'Position', [0 0 165 852]);
```

Open the cosim_mimo.slx model. Drag the new block and convenience command blocks created by cosimWizard into the cosimulation model, replacing the placeholder subsystem inside the MIMODecoder subsystem.

2. Cosimulate to Verify the Legacy HDL Code

In your cosimulation model double-click the "Launch HDL Simulator" block to launch your chosen HDL simulator. Click the Play button in Simulink to start the cosimulation and observe that warning messages are displayed in the MATLAB window. These are indicating mismatches on the output signals because of a discrepancy between the reference FSM model and the HDL implementation.

Now you can use Simulink and HDL simulator debugging features to isolate the problem and fix the bug. In this case the errors arise because a state transition arc was missed in the HDL implementation. Notice in the HDL simulator's waveform display that the FSM state gets stuck very early in the simulation.

Fix the Hand-Written HDL Code and Rerun the Cosimulation

The corrected HDL code has been supplied for this example. Use the following command to copy the new code to your working directory, overwriting the bad version of `Embedded_Controller.vhd`:

```
copyfile(fullfile('verify_legacy_hdlsrc', 'fixed_hdl', 'Embedded_Controller.vhd'), 'verify_legacy')
```

Recompile the Legacy HDL code by double-clicking the "Compile HDL Design" block. Exit the HDL simulator if it is still open following the previous execution of the cosimulation and relaunch the HDL simulator, then replay the cosimulation. You should observe no mismatches this time.

Now that you've debugged and verified the legacy HDL code for the Embedded Controller you can go on to verify the entire MIMODecoder with FPGA-in-the-loop.

Set FPGA Design Software Environment

Before using FPGA-in-the-loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function `hdlsetuptoolpath` to add FPGA design software to the system path for the current MATLAB session.

Prepare the Model for HDL Code Generation

To prepare the model for FPGA-in-the-loop incorporating the legacy HDL code and generating new HDL code for the remainder of the MIMO Decoder you need to do 2 things to complete the FPGA implementation:

- 1 edit the cosimulation model to remove the FSM reference design
- 2 use the HDL Coder Blackbox to incorporate the legacy HDL into the model for code generation

If you want to follow all steps to prepare the model for HDL code generation using the HDL Blackbox, save the cosimulation model with a different name and proceed with the rest of model preparation as follows:

1. edit the cosimulation model to remove the FSM reference design
 - inside the MIMO Decoder subsystem delete the `Embedded_Controller` function block
 - delete the "from" blocks that drive `Embedded_Controller` inputs with the exception of the `enablecoder` input
 - delete the comparators and assertion blocks on the outputs
 - reconnect the cosimulation block outputs to the inputs of `DelaySubsystem1`
2. use the HDL Coder Blackbox to incorporate the legacy HDL into the model for code generation
 - select the cosimulation block and type control-G to create a subsystem
 - right-click on the new cosimulation subsystem and select HDL Code and HDL Block Properties
 - select Architecture = `BlackBox`
 - enter `FSMSubsystem` in the `EntityName` parameter

- enter 0 in the ImplementationLatency parameter
- OK the HDL Block properties dialog

3. rerun the simulation to update the diagram.

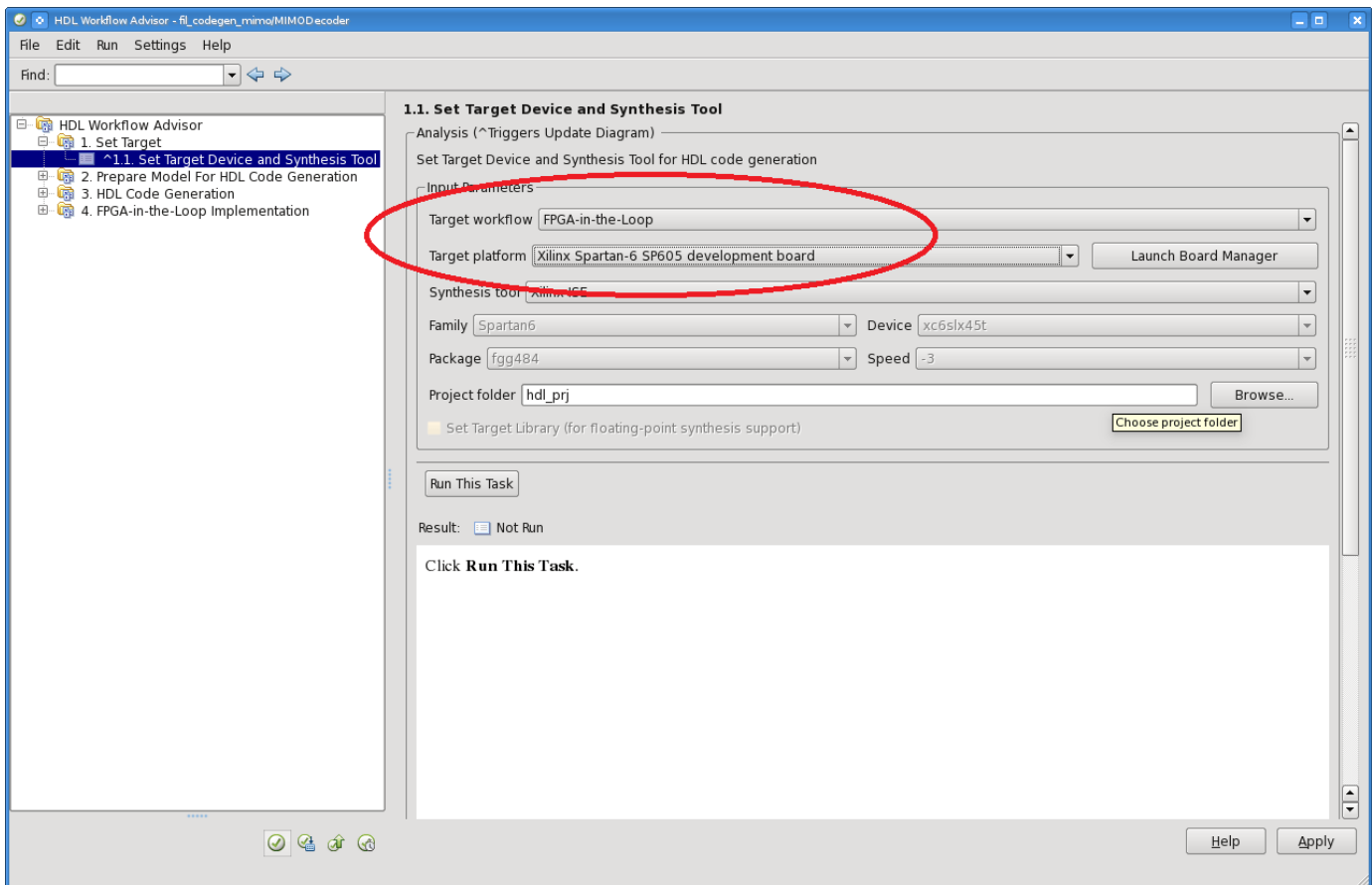
- double-click the "Launch HDL Simulator" block to launch the HDL simulator
- click the Play button in Simulink to start the cosimulation
- save the model

3. Generate HDL Code and FPGA-in-the-Loop

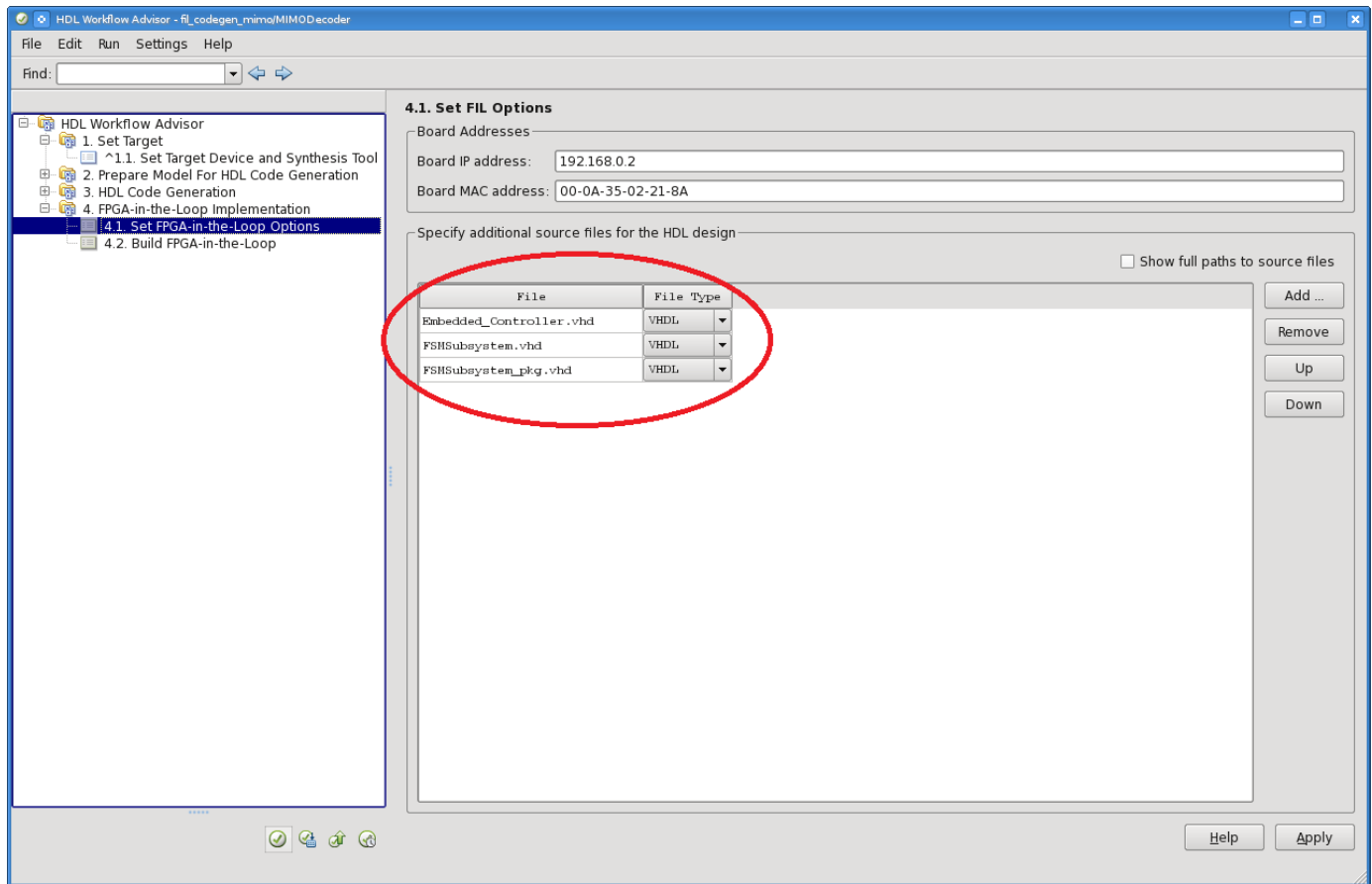
This step requires HDL Coder. If you do not have this software, you can use pre-generated HDL files for FIL simulation. Jump directly to step 5. FIL Simulation Using filWizard.

If you want to follow the process to generate the HDL files yourself return to the top level of the model, right click on the MIMODecoder subsystem and under "HDL Code" launch the HDL Coder Workflow Advisor.

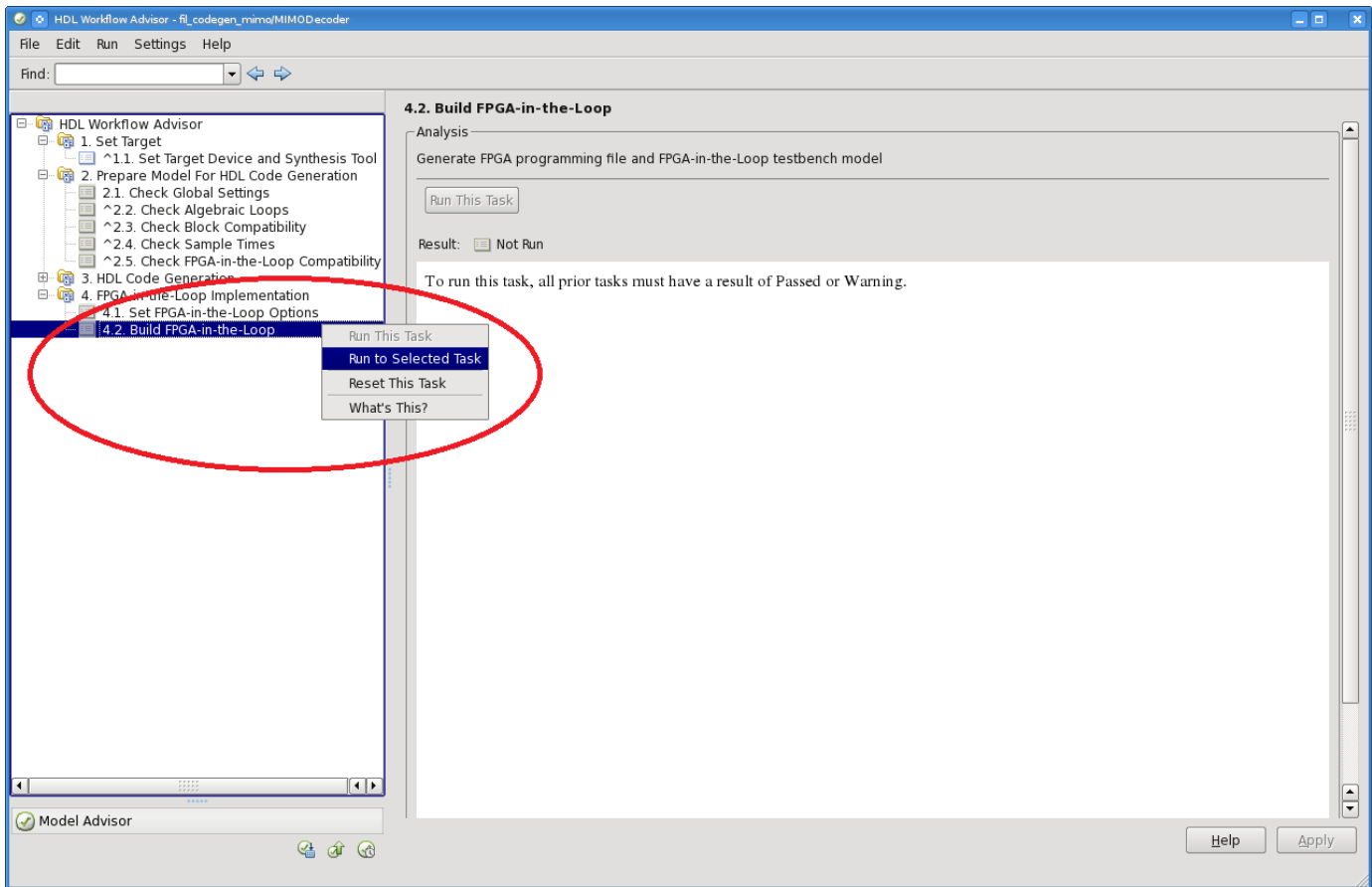
- Step 1.1: select FPGA-in-the-loop Target Workflow, select your preferred FPGA development board from the drop-down list, and identify a writeable directory to hold the generated HDL code.



- Step 4.1: in Set FPGA Options select "Add" and use the browser to navigate to the EmbeddedController HDL files you copied to your working folder in Step 1 and modified with the fixed HDL code in Step 3.



- Step 4.2: Right-click on step 4.2 of the workflow in the left-hand navigation tree and select "Run to this task". This step may take several minutes because it includes the steps to synthesize, map, and route the design for the FPGA device.



The result will be an FPGA programming file for FPGA-in-the-loop simulation of the MIMO Decoder subsystem and a new model containing the original model (including the legacy HDL for the FSM) of the decoder alongside the FPGA-in-the-loop block. It will also have comparators with assertion blocks to identify mismatching signals similar to those we saw in the cosimulation model.

4. Verify the Design with FPGA-in-the-Loop Simulation

Since the generated verification model includes the cosimulation for the FSMSubsystem you will need to use the HDL simulator to run the entire FIL model. Make sure that the HDL simulator from your previous cosimulation is shut down and relaunch the HDL simulator.

In the FPGA-in-the-loop model generated in Step 3, open the FIL block.

Select "Load" to download the FPGA programming file to the device on your board.

Click Play in the Simulink model to run FPGA-in-the-loop simulation.

Observe the results in the comparison scopes and the ErrorRate Calculation in the model. Your FIL simulation results should exactly match the reference model.

5. FIL Simulation Using FIL Wizard

This step is the alternative to Step 4 for those who do not have HDL Coder software. If you've completed Step 4 you need not continue with this step.

The pre-generated HDL files are located in the "verify_legacy_gen_hdlsrc" folder. You can create the FPGA programming file for FPGA-in-the-loop using the FIL wizard. The FIL wizard will also create a FIL block which you can discard because the FIL model provided for this example already contains the FIL block.

Open the FIL wizard by entering the following command:

```
filWizard
```

- In **FIL Options** select your FPGA development board from the list.
- In **Source Files** select **Add** and choose all of the files in the folder `verify_legacy_gen_hdlsrc` and identify **MIMODecoder.vhd** as the top level file.
- Accept the default values for the remainder of the filWizard options
- Wait for the FIL block and FPGA programming file to be created. This may take several minutes due to the time required to synthesize and route the FPGA implementation.
- Open the `gm_fil_codegen_mimo_fil.slx` model and drag the newly generated FIL block into the model at the location indicated.
- Open the FIL block mask, click on the Signal Attributes tab. Change the data type for each `rx_decoded` output to `fixdt(1,6,0)` to match the data type of the behavioral block.
- Open the FIL block mask, click on the Main tab, select Load and wait for the FPGA programming file to be downloaded to the device.
- Press Play in the Simulink model to run FPGA-in-the-loop.

Observe the results in the comparison scopes and the ErrorRate Calculation in the model. Your FIL simulation results should exactly match the reference model.

This concludes the example of Using HDL Cosimulation and FPGA-in-the-loop to Verify HDL Designs.

FPGA-in-the-Loop Simulation Using MATLAB System Object

This example uses a MATLAB® System object and an FPGA to verify a register transfer level (RTL) design of a Fast Fourier Transform (FFT) of size 8 written in Verilog. The FFT is commonly used in digital signal processing to produce frequency distribution of a signal.

To verify the correctness of this FFT, a MATLAB System object testbench is provided. This testbench generates a periodic sinusoidal input to the HDL design under test (DUT) and plots the Fourier Coefficients in the Complex Plane.

Set FPGA Design Software Environment

Before using FPGA-in-the-Loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function `hdlsetuptoolpath` to add FPGA design software to the system path for the current MATLAB session.

Launch FilWizard

Launch the FIL Wizard prepopulated with the FFT example information. Enter your FPGA board information in the first step, follow every step of the Wizard and generate the FPGA programming file and FIL System object.

```
filWizard('fft_hdlsrc/fft8_sysobj_fil.mat');
```

Program FPGA

Program the FPGA with the generated programming file. Before continuing, make sure the FIL Wizard has finished the FPGA programming file generation. Also make sure your FPGA board is turned on and connected properly.

```
run('fft8_fil/fft8_programFPGA');
```

Instantiate SineWave System Objects

The following code instantiates the system objects that represent the sine wave generator (F=100Hz, Sampling=1000Hz, complex fix point output).

```
SinGenerator = dsp.SineWave('Frequency ', 100, ...
    'Amplitude', 1, ...
    'Method', 'Table lookup', ...
    'SampleRate', 1000, ...
    'OutputDataType', 'Custom', ...
    'CustomOutputDataType', numerictype([], 10, 9), ...
    'ComplexOutput', true);
```

Instantiate the FPGA-in-the-Loop System Object

`fft8_fil` is a customized FILSimulation System object, which represents the HDL implementation of the FFT running on the FPGA in this simulation system.

```
Fft = fft8_fil;
```

Run the Simulation

This example simulates the sine wave generator and the FFT HDL implementation via the FPGA-in-the-Loop System object. This section of the code calls the processing loop to process the data sample-by-sample.

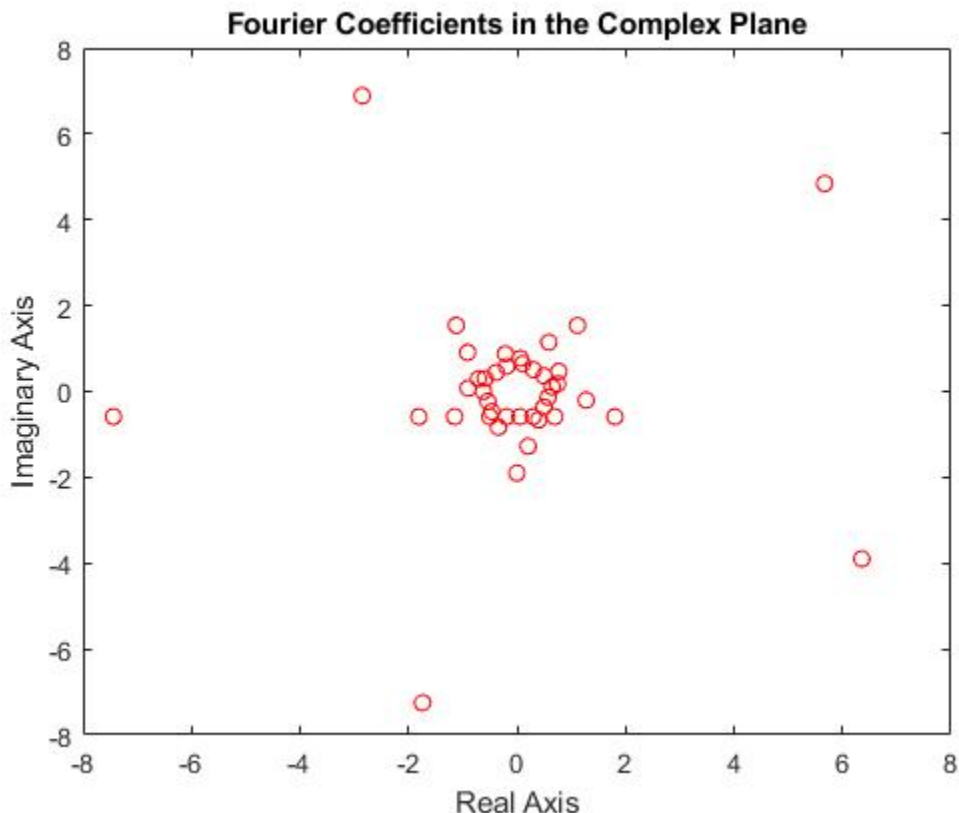
```
for ii=1:1000
    % Read 1 sample from the sine wave generator
    ComplexSinus = step(SinGenerator);
    % Send/receive 1 sample to/from the HDL FFT on the FPGA
    [RealFft, ImagFft] = step(Fft,real(ComplexSinus),imag(ComplexSinus));
    % Store the FFT sample in a vector
    ComplexFft(ii) = RealFft + ImagFft*1i;
end
```

Display the Fourier Coefficients

Plot the Fourier Coefficients in the Complex Plane.

```
% Discard the first 12 samples (initialization of the HDL FFT)
ComplexFft(1:12)=[];

% Display the FFT
plot(ComplexFft,'ro');
title('Fourier Coefficients in the Complex Plane');
xlabel('Real Axis');
ylabel('Imaginary Axis');
```



This concludes the "FPGA-in-the-Loop simulation using MATLAB System Object" example.

Verify Viterbi Decoder Using MATLAB System Object and FPGA-in-the-Loop

This example shows you how to use MATLAB® System objects and FPGA-in-the-Loop to simulate a Viterbi decoder implemented in VHDL® on an FPGA.

Set FPGA Design Software Environment

Before using FPGA-in-the-Loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function `hdlsetuptoolpath` to add FPGA design software to the system path for the current MATLAB session.

Launch FilWizard

Click the **Open Script** button. Then, launch the FIL Wizard prepopulated with the Viterbi example information. Enter your FPGA board information in the first step, follow every step of the Wizard and generate the FPGA programming file and FIL System object.

```
filWizard('viterbi_hdlsrc/viterbi_sysobj_fil.mat');
```

Program FPGA

Program the FPGA with the generated programming file. Before continuing, make sure the FIL Wizard has finished the FPGA programming file generation. Also make sure your FPGA board is turned on and connected properly.

```
run('viterbi_block_fil/viterbi_block_programFPGA');
```

Set Simulation Parameters and Instantiate Communication System Objects

The following code sets up the simulation parameters and instantiates the system objects that represent the channel encoder, BPSK modulator, AWGN channel, BPSK demodulator, and error rate calculator. Those objects comprise the system around the Viterbi decoder and can be thought of as the test bed for the Viterbi HDL implementation.

```
EsNo = 0;    % Energy per symbol to noise power spectrum density ratio in dB
FrameSize = 1024; % Number of bits in each frame

% Convolution Encoder
hConEnc = comm.ConvolutionalEncoder;
% BPSK Modulator
hMod = comm.BPSKModulator;
% AWGN channel
hChan = comm.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (Es/No)',...
    'SamplesPerSymbol',1,...
    'EsNo',EsNo);
% BPSK demodulator
hDemod = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio',...
    'Variance',0.5*10^(-EsNo/10));
% Error Rate Calculator
hError = comm.ErrorRate('ComputationDelay',100,'ReceiveDelay', 58);
```

Instantiate the FPGA-in-the-Loop System Object

`viterbi_block_fil` is a customized `FILSimulation` System object, which represents the HDL implementation of the Viterbi decoder running on the FPGA in this simulation system.

```
hDec = viterbi_block_fil;
```

Run the Simulation

This example simulates the BPSK communication system in MATLAB incorporating the Viterbi decoder HDL implementation via the FPGA-in-the-Loop System object. This section of the code calls the processing loop to process the data frame-by-frame with 1024 bits in each data frame.

```
for counter = 1:20480/FrameSize
    data = randi([0 1],FrameSize,1);
    encodedData = step(hConEnc, data);
    modSignal = step(hMod, encodedData);
    receivedSignal = step(hChan, modSignal);
    demodSignalSD = step(hDemod, receivedSignal);
    quantizedValue = fi(4-demodSignalSD,0,3,0);
    input1 = quantizedValue(1:2:2*FrameSize);
    input2 = quantizedValue(2:2:2*FrameSize);
    % Send/receive 1 frame to/from the HDL viterbi decoder on the FPGA
    [ce_out, receivedBits] = step(hDec,input1, input2);
    errors = step(hError, data, double(receivedBits));
end
```

Display the Bit-Error Rate

The Bit-Error Rate is displayed for the Viterbi decoder.

```
sprintf('Bit Error Rate is %d\n',errors(1))
```

This concludes the "Verifying Viterbi Decoder Using MATLAB System Object and FPGA-in-the-Loop" example.

Video Processing Acceleration Using FPGA-in-the-Loop

This example uses FPGA-in-the-Loop (FIL) simulation to accelerate a video processing simulation with Simulink® by adding an FPGA. The process shown analyzes a simple system that sharpens an RGB video input at 24 frames per second.

This example uses the Computer Vision Toolbox™ in conjunction with HDL Coder™ and HDL Verifier™ to show a design workflow for implementing FIL simulation.

Tools required to run this example:

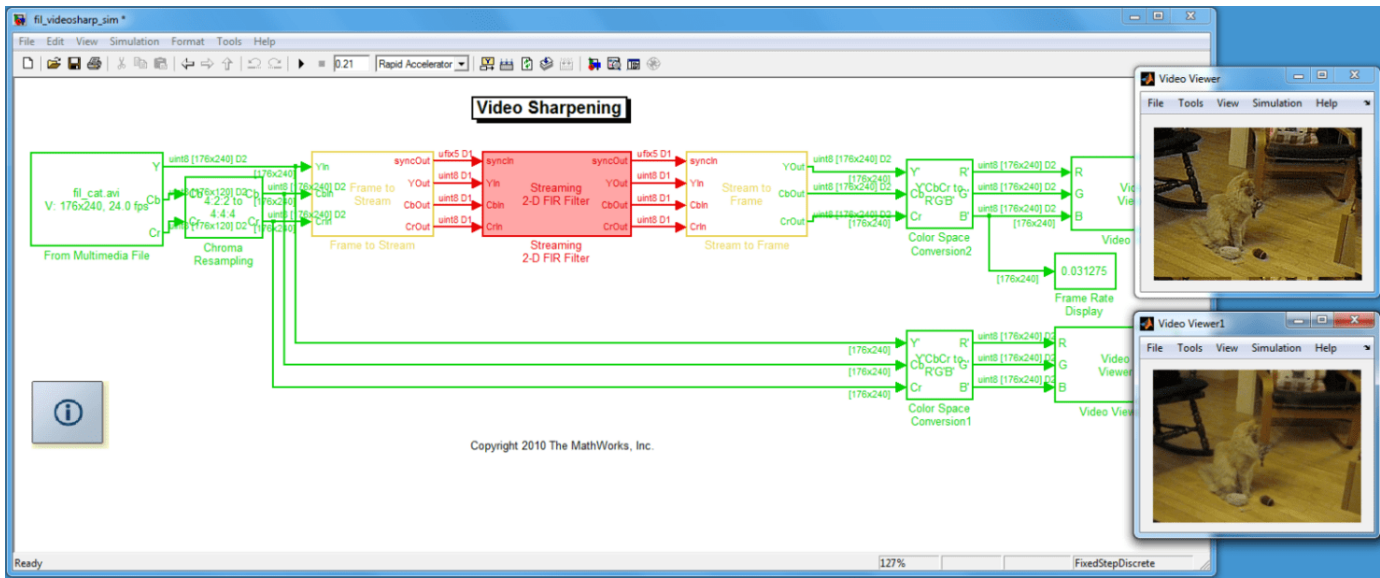
- FPGA design software (Xilinx® ISE® or Vivado® design suite or Intel® Quartus® Prime design software)
- One of the supported FPGA development boards and accessories (the ML403, SP601, BeMicro SDK, and Cyclone III Starter Kit boards are not supported for this example). For more information about supported hardware, see “Supported FPGA Devices for FPGA Verification”.
- For connection using Ethernet: Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable
- For connection using JTAG: USB Blaster I or II cable and driver for Altera FPGA boards. Digilent® JTAG cable and driver for Xilinx FPGA boards.
- For connection using PCI Express®: FPGA board installed into PCI Express slot of host computer.

MATLAB® and FPGA design software can either be locally installed on your computer or on a network accessible device. If you use software from the network you will need a second network adapter installed in your computer to provide a private network to the FPGA development board. Consult the hardware and networking guides for your computer to learn how to install the network adapter.

1. Open and Execute the Simulink Model

Open the model and run the simulation for 0.21s.

Due to the large quantity of data to process , the simulation is not fluent. We will improve the simulation speed in the following steps by using a FPGA-in-the-Loop.



2. Generate HDL Code

Generate HDL code for the Streaming Video Sharpening subsystem by performing these steps:

- a. Right-click on the block labeled Streaming 2-D FIR Filter.
- b. Select **HDL Code > Generate HDL for Subsystem** in the context menu.

Alternatively, you can generate HDL code by entering the following command at the MATLAB prompt:

```
makehdl('fil_videosharp_sim/Streaming 2-D FIR Filter')
```

If you do not want to generate HDL code, you can use the pre-generated HDL files located in the `videosharp_hdlsrc` folder.

3. Set Up FPGA Design Software

Before using FPGA-in-the-Loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function `hdlsetuptoolpath` to add Xilinx Vivado or Intel Quartus Prime to the system path for the current MATLAB session.

For Xilinx FPGA boards, run

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin');
```

This example assumes that the Xilinx Vivado executable is located in `C:\Xilinx\Vivado\2019.2\bin`. Substitute with your actual executable location if it is different.

For Intel boards, run

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', 'ToolPath', 'C:\Intel\quartus\18.1\bin64');
```

This example assumes that the Intel Quartus Prime executable is located in `C:\Intel\quartus\18.1\bin64`. Substitute with your actual executable location if it is different.

4. Run FPGA-in-the-Loop Wizard

Enter the following command at the MATLAB prompt to launch the FIL Wizard:

```
filWizard;
```

4.1 Hardware Options

Select a board in the board list.

4.2 Source Files

a. Add the previously generated HDL source files for the Streaming Video Sharpening subsystem.

b. Select `Streaming_2_D_FIR_Filter.vhd` as the Top-level file.

4.3 DUT I/O Ports

Do not change anything in this view.

4.4 Build Options

a. Select an output folder.

b. Click Build to build the FIL block and the FPGA programming file.

During the build process, the following actions occur:

- A FIL block named `Streaming_2_D_FIR_Filter` is generated in a new model. Do not close this model.
- After new model generation, the FIL Wizard opens a command window where the FPGA design software performs synthesis, fit, place-and-route, timing analysis, and FPGA programming file generation. When the FPGA design software process is finished, a message in the command window lets you know you can close the window. Close the window.

c. Close the `fil_videosharp_sim.slx` model.

5. Open and Complete the Simulink Model for FIL

a. Open the `fil_videosharp_fpga.slx` model.

b. Copy in it the previously generated FIL block to `fil_videosharp_fpga.slx` where it says "Replace this with FIL block"

6. Configure FIL Block

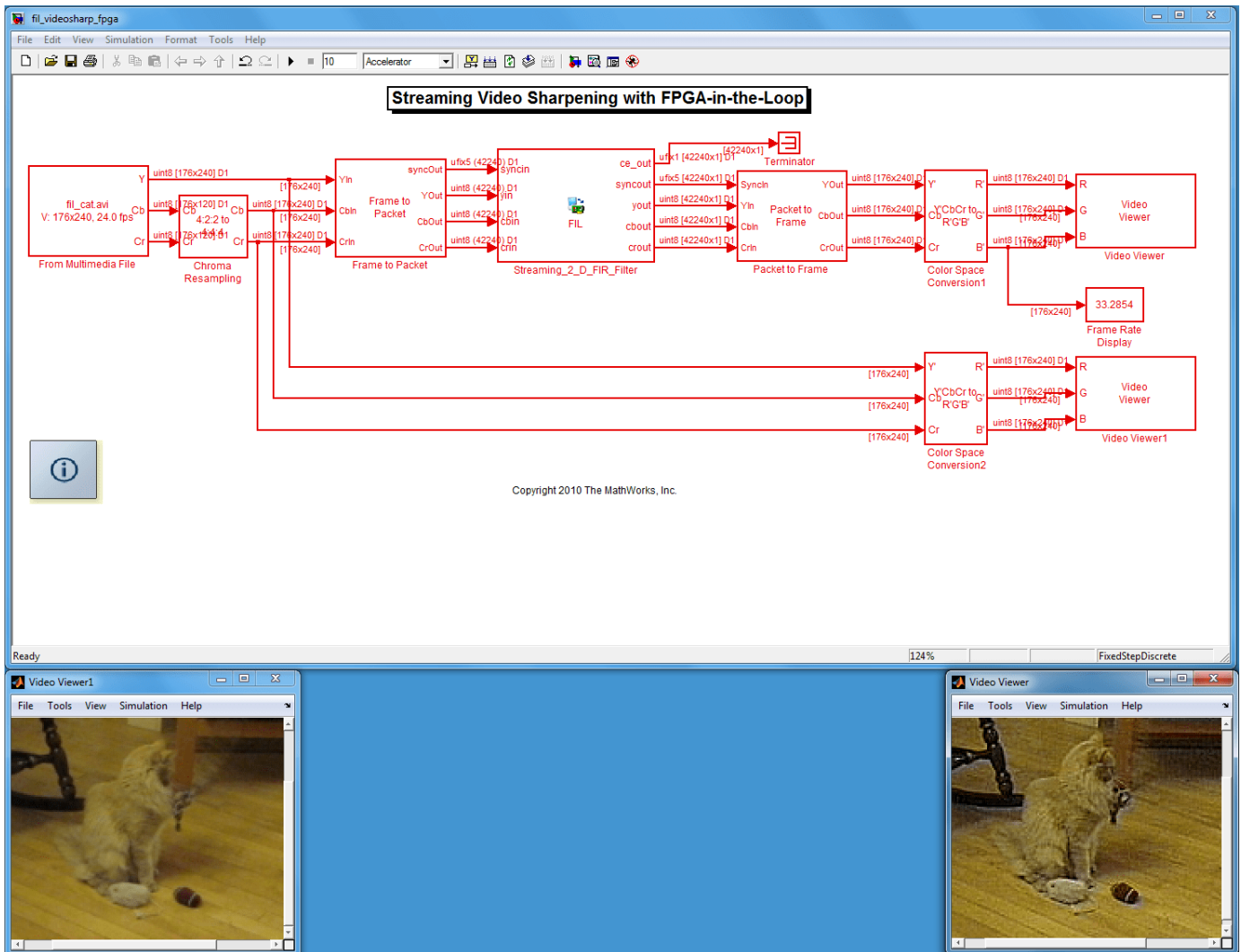
a. Double-click the FIL block in the Streaming Video Sharpening with FPGA-in-the-Loop model to open the block mask.

b. Click **Load**.

c. Click **OK** to close the block mask.

7. Run FIL Simulation

Run the simulation for 10s and observe the performance improvement.



This concludes the Video Processing Acceleration using FPGA-In-the-Loop example.

Accelerating Communications System Simulation Using FPGA-in-the-Loop

This example uses FPGA-in-the-Loop (FIL) simulation to accelerate part of a communications system. The application uses the Viterbi Algorithm to decode a convolutional encoded random stream that is BPSK modulated, sent through an AWGN channel, and then demodulated. Using a sample-by-sample approach leads to a modest speedup over normal Simulink® simulation, while using the "Process as Frames" option leads to further speedup.

This example uses the Communications Toolbox™ in conjunction with HDL Coder™ and HDL Verifier™ to show a design workflow for accelerating simulation using FPGA-in-the-loop.

Requirements and Prerequisites

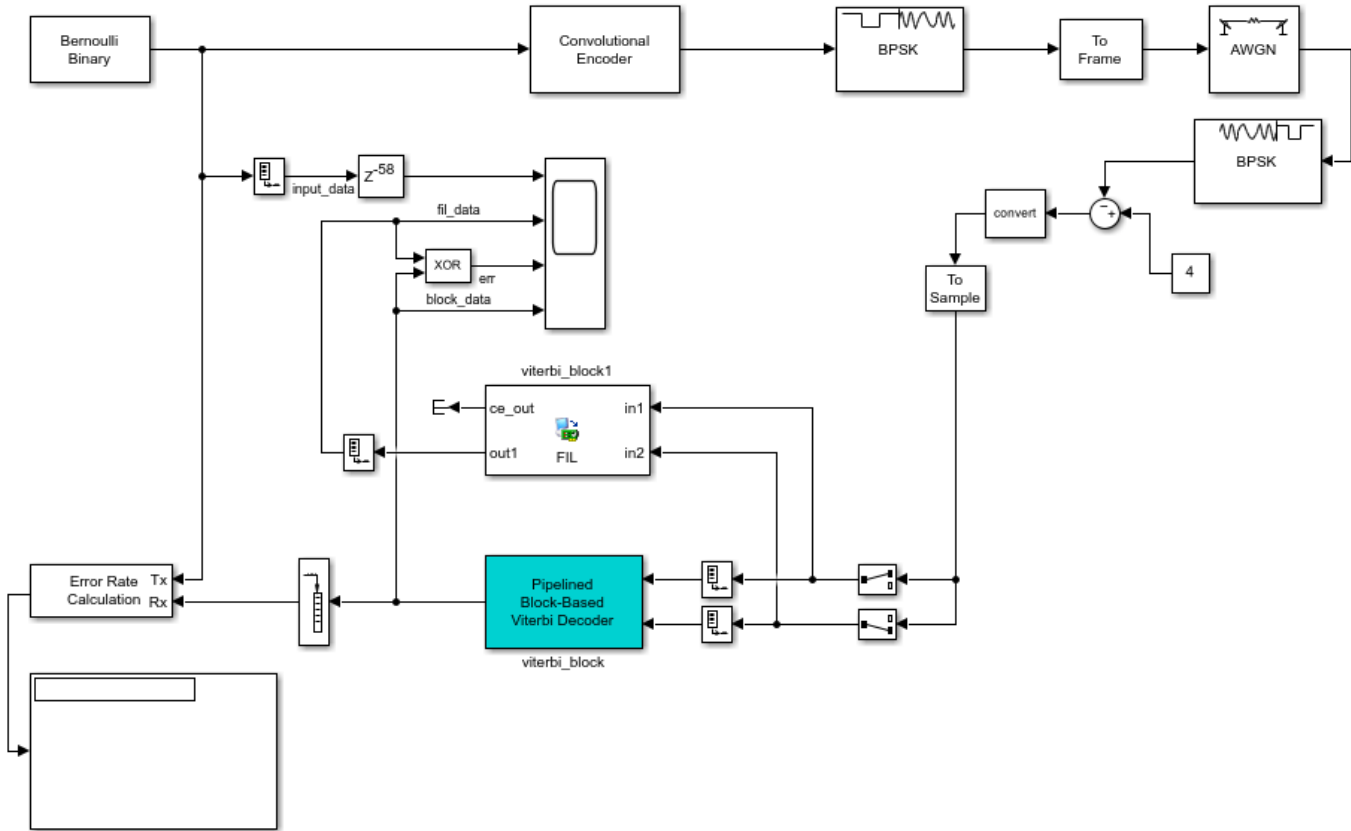
Tools required to run this example:

- FPGA design software
- One of the supported FPGA development boards. For supported hardware, see "Supported FPGA Devices for FPGA Verification".
- For connection using Ethernet: Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable
- For connection using JTAG: USB Blaster I or II cable and driver for Intel FPGA boards. Digilent® JTAG cable and driver for Xilinx FPGA boards.
- For connection using PCI Express®: FPGA board installed into PCI Express slot of host computer.

Requirements: MATLAB® and FPGA design software can either be locally installed on your computer or on a network accessible device. If you use software from the network you will need a second network adapter installed in your computer to provide a private network to the FPGA development board. Consult the hardware and networking guides for your computer to learn how to install the network adapter.

1. Open and Execute the Simulink Model

Open the model. Due to the large quantity of data to process, the simulation takes approximately 9 seconds without FIL. We will improve the simulation speed in the following steps by using a FPGA-in-the-Loop.



Copyright 2010-2011 The MathWorks, Inc.

2. Generate HDL Code

This step requires HDL Coder. If you do not have HDL Coder, you can use pre-generated HDL files in the current directory.

If you are going to use these copied files, go directly to step 3.

Generate HDL code for the Viterbi block subsystem by performing these steps:

a. Right click on the existing FIL block labeled `viterbi_block1`. Click on Delete to remove this block for code generation.

b. In the **Modeling** tab, click **Model Settings**.

d. Click on the **HDL Code Generation** pane and make sure the `hdlcoderviterbi_for_fil/viterbi_block` is selected.

e. Click **Generate**.

Alternatively, you can generate HDL code by entering the following command at the MATLAB prompt:

```
makehdl('hdlcoderviterbi_for_fil/viterbi_block')
```

3. Set FPGA Design Software Environment

Before using FPGA-in-the-Loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function `hdlsetuptoolpath` to add FPGA design software to the system path for the current MATLAB session.

4. Run FPGA-in-the-Loop Wizard

To launch the FIL Wizard, enter the following command at the MATLAB prompt:

```
filWizard;
```

4.1 Hardware Options

Select a board from the board list. Click Next to continue.

4.2 Source Files

- a. Add all the previously generated HDL source files for the `Viterbi` Block subsystem.
- b. Select the file `viterbi_block.vhd` as the Top-level file. You may need to make the FPGA-in-the-Loop Wizard window wider in order to see these options.
- c. Notice that the `viterbi_block` has been entered for you as the default top-level module name. Click Next to continue.

4.3 DUT I/O Ports

Observe that the ports of the top-level module have been correctly identified. Click Next to continue.

4.4 Build Options

- a. Select an output folder.
- b. Click Build to build the FIL block and programming file.

During the build process, the following actions occur:

- A FIL block named `viterbi_block` is generated in a new model.
- After new model generation, the FIL Wizard opens a command window where the FPGA design software performs synthesis, mapping, place-and-route, timing analysis, and FPGA programming file generation. For this block, these steps take about 20 minutes.

When the FPGA design software process is finished, a message in the command window lets you know you can close the window.

5. Open and Complete the Simulink Model for FIL

- a. Open the `hdlcoderviterbi_for_fil.mdl`
- b. Copy into it the previously generated FIL block and connect it either in parallel to or in place of the `viterbi_block`. Note that the original block has the inputs on the right. To make the FIL block have its inputs on the right, right-click the block, and then select **Format > Flip Block**.

6. Configure FIL Block

- a. Double-click the FIL block in the model to open the block mask.

b. Click **Load**.

c. Click **OK** to close the block mask.

7. Run FIL Simulation

Run the simulation for 20480 seconds and observe the performance.

```
FrameSize = 1;  
tic;  
sim('hdlcoderviterbi_for_fil');  
fs1 = toc
```

You can try setting the frame size to a larger number. For this example, the frame size is set to 1024 bytes.

```
FrameSize = 1024;  
tic;  
sim('hdlcoderviterbi_for_fil');  
fs2 = toc
```

In our tests, the time to simulate was around 16 seconds when `FrameSize = 1`, which is about the same as simulating in Simulink without FIL, but the simulation time was decreased to around 12 seconds when the frame size was increased to 1024 using a Xilinx Spartan-6 SP605 board. This particular board and system gives an overall speedup of around 1.7 times faster, but other boards and communications system may be even faster.

```
speedup = fs1 / fs2
```

By removing the Simulink block version and only simulating the FIL version, and by adding more blocks to the FIL part of the design and removing the Simulink scopes and displays as much as possible, even greater speedups are possible.

This concludes the Accelerating Communications System Simulation Using FPGA-In-the-Loop example.

Algorithm Verification with a FIL Source Block

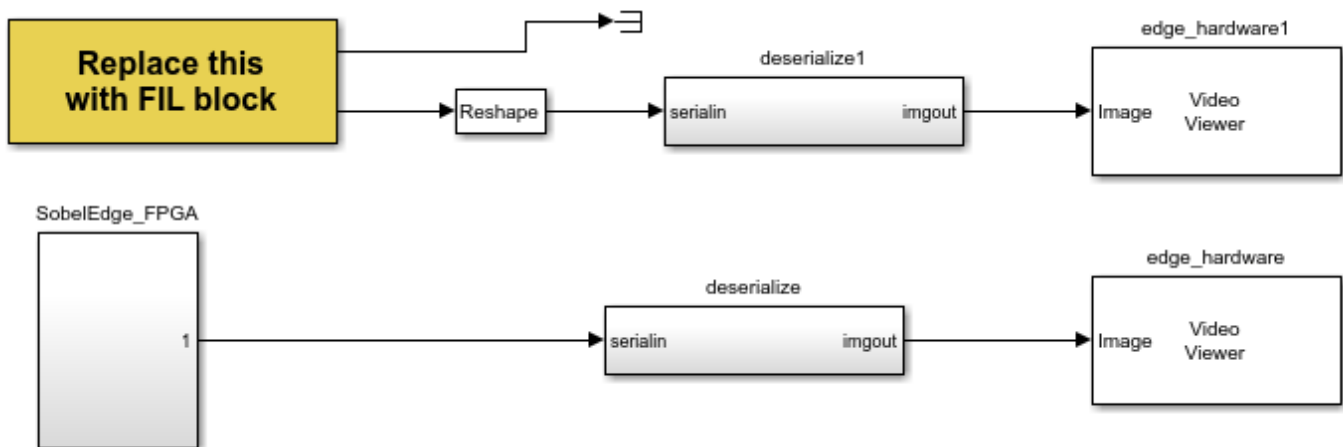
This example shows how to verify an HDL design quickly and efficiently using FPGA-in-the-Loop (FIL). The test bench is synthesized together with the Design Under Test (DUT), the result is downloaded to an FPGA board, and the test is run with FIL. Using FIL for this process enables high speed generation and processing of test stimulus, with results returned to Simulink for analysis. This example executes these tasks for an image processing use case.

Requirements

Tools required for this example:

- Signal Processing Toolbox
- FPGA design software
- One of the supported FPGA development boards and accessories (the ML403 board is not supported for this example). For more information about supported hardware, see “Supported FPGA Devices for FPGA Verification”.
- For connection using Ethernet: Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable
- For connection using JTAG: USB Blaster I or II cable and driver for Intel FPGA boards. Digilent® JTAG cable and driver for Xilinx FPGA boards.
- For connection using PCI Express®: FPGA board installed into PCI Express slot of host computer.

Sobel Edge Detection



The `sobel_edge_hardware` subsystem is the algorithm to be synthesized into an FPGA. The stimulus is a gif file containing the image (a stop sign). In this example the image is moved inside the FPGA along with the algorithm, a FIL block is created for that FPGA design.

The steps taken to implement this technique are illustrated in the model `fil_sobel_model`. In this model, the gif file is contained by a lookup table and a counter is used to scan through the lines in the image and send them to the algorithm. The FIL block is added to the model in parallel with the behavioral blocks to enable directly checking the outputs of the FPGA against the behavioral model as a simple example of results analysis. click the button to open the model.

Generate HDL Code

If you have HDL Coder, you can generate code for the "SobelEdge_FPGA" subsystem using the HDL Workflow Adviser or Configuration Parameter UI (see HDL Coder documentation for more information).

If you do not have HDL Coder, you can use the pre-generated HDL files in the current directory.

Set Up FPGA Design Software

Before using FPGA-in-the-Loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function `hdlsetuptoolpath` to add FPGA design software to the system path for the current MATLAB session.

Configure and Build FPGA-in-the-Loop

The FIL Wizard guides you in configuring settings necessary for building FPGA-in-the-Loop. Launch the wizard with the following command:

```
filWizard;
```

1. In Hardware Options, select the FPGA development board connected to your host computer. If necessary, you can also customize the Board IP and MAC Address under Advanced Options. Click **Next** to continue.
2. In Source Files, add the following generated HDL files for the design to the source file table using **Browse**.

```
SobelEdge_FPGA.vhd,  
DualPortRAM_128x9b.vhd  
DualPortRAM_128x9b_block.vhd  
sobel_edge_eml.vhd  
sobel_edge_hardware.vhd  
SobelEdge_FPGA_pkg.vhd  
u_d_ram.vhd  
u_d_ram_block.vhd
```

Check the top-level checkbox next to `SobelEdge_FPGA`. Click **Next** to continue.

3. In DUT I/O Ports, the input and output port information, such as port name, direction, width and port type are automatically generated from the HDL file. Port types, such as Clock and Data, are generated based on port names; you may change the selection as necessary. For this example, the generated port types are correct, and you can click **Next**.

4. In Build Options, specify the folder for FIL output files. You can use the default value for this example. Click **Build**. Clicking **Build** causes the FIL Wizard to generate all the necessary files for FPGA-in-the-Loop simulation and perform the following actions:

- Generates a FIL block in a new Simulink® model
- Opens a command-line window to compile the FPGA project and generate the FPGA programming file

The FPGA project compilation process takes several minutes. When the process is finished, you are prompted to close the command-line window. Close this window now.

Configure FIL Block

To prepare for FPGA-in-the-Loop simulation, follow the steps below to configure the FIL block.

1. Open the test bench model `fil_sobel_model` and copy the generated FIL block into the model.
2. Double-click the FIL block to open the block mask. Click **Load** to program the FPGA with the generated programming file.
3. Under Runtime Options, change **Overclocking factor** to 1. This specifies that each input value is sampled once by the FPGA clock before the input changes value.
4. Set the output frame size to 20000.
5. On the FIL block mask, click on the Signal Attributes tab. Change the data type for `out1` to `boolean` to match the data type of the behavioral Sobel block.
5. Click **OK** to close the block mask.

Results

The model provided in this example will compare the results obtained from the behavioral block with those from the FPGA. You can run that model and observe the equivalence. You can also observe the relative simulation performance of the behavioral simulation and the FIL simulation by creating a new model containing only the behavioral path and another containing only the path with the FIL block. Run each of those models separately using:

```
tic;  
sim(your_model_name);  
toc
```

You can see that FIL is faster. This increase in speed is because there is only one FIL function call for 20000 samples while the behavioral model needs 20000 function calls to process all of the data. Therefore you will get better performance if the size of the image is bigger. This step concludes this example.

Verify Digital Up-Converter Using FPGA-in-the-Loop

This example shows you how to verify a digital up-converter design generated with Filter Design HDL Coder™ using FPGA-in-the-Loop simulation.

Requirements

Tools required for this example:

- FPGA design software
- One of the supported FPGA development boards and accessories (the ML403 board is not supported for this example). For more information about supported hardware, see “Supported FPGA Devices for FPGA Verification”.
- For connection using Ethernet: Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable
- For connection using JTAG: JTAG cable with USB Blaster I or II, USB Blaster driver
- For connection using PCI Express®: FPGA board installed into PCI Express slot of host computer.

Create Cascade Filter for DUC

A digital up-converter (DUC) is a digital circuit that converts a digital baseband signal to a passband signal. A DUC is composed of three filtering stages; each stage filters the input signal with a lowpass interpolating filter, followed by a sample rate change. In this example, the DUC is a cascade of two FIR interpolation filters and a CIC interpolation filter, as described in the example HDL Digital Up-Converter (DUC).

1. Create the two FIR and CIC filters.

```

pfir = [0.0007    0.0021   -0.0002   -0.0025   -0.0027    0.0013    0.0049    0.0032 ...
        -0.0034   -0.0074   -0.0031    0.0060    0.0099    0.0029   -0.0089   -0.0129 ...
        -0.0032    0.0124    0.0177    0.0040   -0.0182   -0.0255   -0.0047    0.0287 ...
         0.0390    0.0049   -0.0509   -0.0699   -0.0046    0.1349    0.2776    0.3378 ...
         0.2776    0.1349   -0.0046   -0.0699   -0.0509    0.0049    0.0390    0.0287 ...
        -0.0047   -0.0255   -0.0182    0.0040    0.0177    0.0124   -0.0032   -0.0129 ...
        -0.0089    0.0029    0.0099    0.0060   -0.0031   -0.0074   -0.0034    0.0032 ...
         0.0049    0.0013   -0.0027   -0.0025   -0.0002    0.0021    0.0007 ];

hpfir = dsp.FIRInterpolator(2, pfir);
hpfir.FullPrecisionOverride = false;
hpfir.CoefficientsDataType = 'Custom';
hpfir.CustomCoefficientsDataType = numeric([],16);
hpfir.ProductDataType = 'Custom';
hpfir.CustomProductDataType = numeric([],31,31);
hpfir.AccumulatorDataType = 'Custom';
hpfir.CustomAccumulatorDataType = numeric([],16,15);
hpfir.OutputDataType = 'Custom';
hpfir.CustomOutputDataType = numeric([],16,15);
hpfir.RoundingMethod = 'Nearest';

cfir = [-0.0007   -0.0009    0.0039    0.0120    0.0063   -0.0267   -0.0592   -0.0237 ...
        0.1147    0.2895    0.3701    0.2895    0.1147   -0.0237   -0.0592   -0.0267 ...
        0.0063    0.0120    0.0039   -0.0009   -0.0007];

hcfir = dsp.FIRInterpolator(2, cfir);
hcfir.FullPrecisionOverride = false;

```

```

hcfir.CoefficientsDataType = 'Custom';
hcfir.CustomCoefficientsDataType = numerictype([],16);
hcfir.ProductDataType = 'Custom';
hcfir.CustomProductDataType = numerictype([],31,31);
hcfir.AccumulatorDataType = 'Custom';
hcfir.CustomAccumulatorDataType = numerictype([],16,15);
hcfir.OutputDataType = 'Custom';
hcfir.CustomOutputDataType = numerictype([],16,15);
hcfir.RoundingMethod = 'Nearest';

hcic = dsp.CICInterpolator(32, 1, 5);
hcic.FixedPointDataType = 'Minimum section word lengths';
hcic.OutputWordLength = 20;

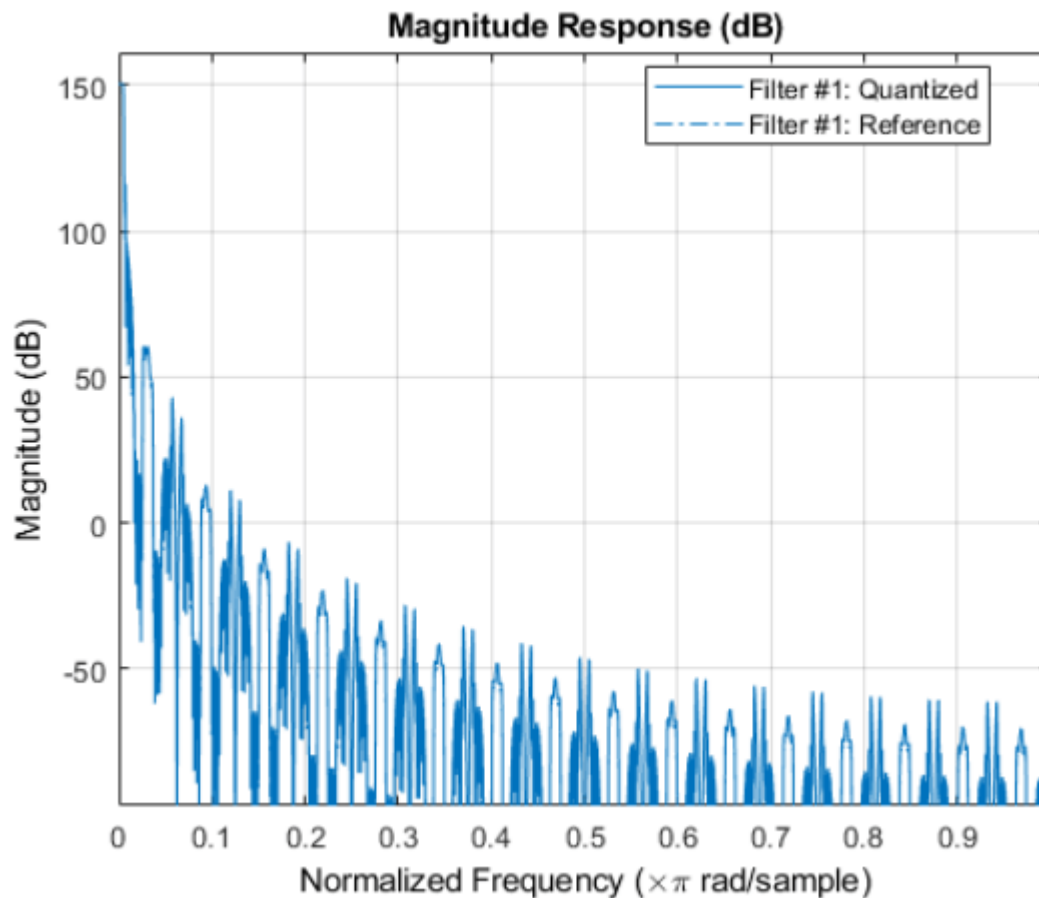
```

2. Create a cascade filter using these filters.

```
hduc = dsp.FilterCascade(hpfir, hcfir, hcic);
```

The frequency response of the cascade filter is shown in the following figure.

```
fvtool(hduc, 'Arithmetic', 'fixed');
```



Generate HDL Code

When the cascade filter is ready, generate HDL code for the DUC using the Filter Design HDL Coder function `generatehdl`, with the property 'AddPipelineRegisters' set to 'on'.

```
generatehdl(hduc, 'Name', 'hdlduc', 'AddPipelineRegisters', 'on', 'InputDataType', numericity(1
```

This option inserts pipeline registers between filter stages, and allows the generated filter to be synthesized at a higher clock frequency.

If you do not have Filter Design HDL Coder, you can use the pre-generated HDL files in the current directory.

Set Up FPGA Design Software

Before using FPGA-in-the-Loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function `hdlsetuptoolpath` to add FPGA design software to the system path for the current MATLAB session.

Configure and Build FPGA-in-the-Loop

The FIL Wizard guides you in configuring settings necessary for building FPGA-in-the-Loop. Launch the wizard with the following command:

```
filWizard
```

1. In Hardware Options, select the FPGA development board connected to your host computer. If necessary, you can also customize the Board IP and MAC Address under Advanced Options. Click *Next" to continue.
2. In Source Files, add the following generated HDL files for the DUC to the source file table using **Browse**.

```
hdlduc.vhd
hdlduc_stage1.vhd
hdlduc_stage2.vhd
hdlduc_stage3.vhd
```

Select the top-level checkbox next to `hdlduc.vhd`. Click *Next" to continue.

3. In DUT I/O Ports, the input and output port information, such as port name, direction, width and port type are automatically generated from the HDL file. Port types, such as Clock and Data, are generated based on port names; you may change the selection as necessary. For this example, the generated port types are correct, and you can click **Next**.

4. In Build Options, specify the folder for FIL output files. You can use the default value for this example. Click **Build**. Clicking **Build** causes the FIL Wizard to generate all the necessary files for FPGA-in-the-Loop simulation and perform the following actions:

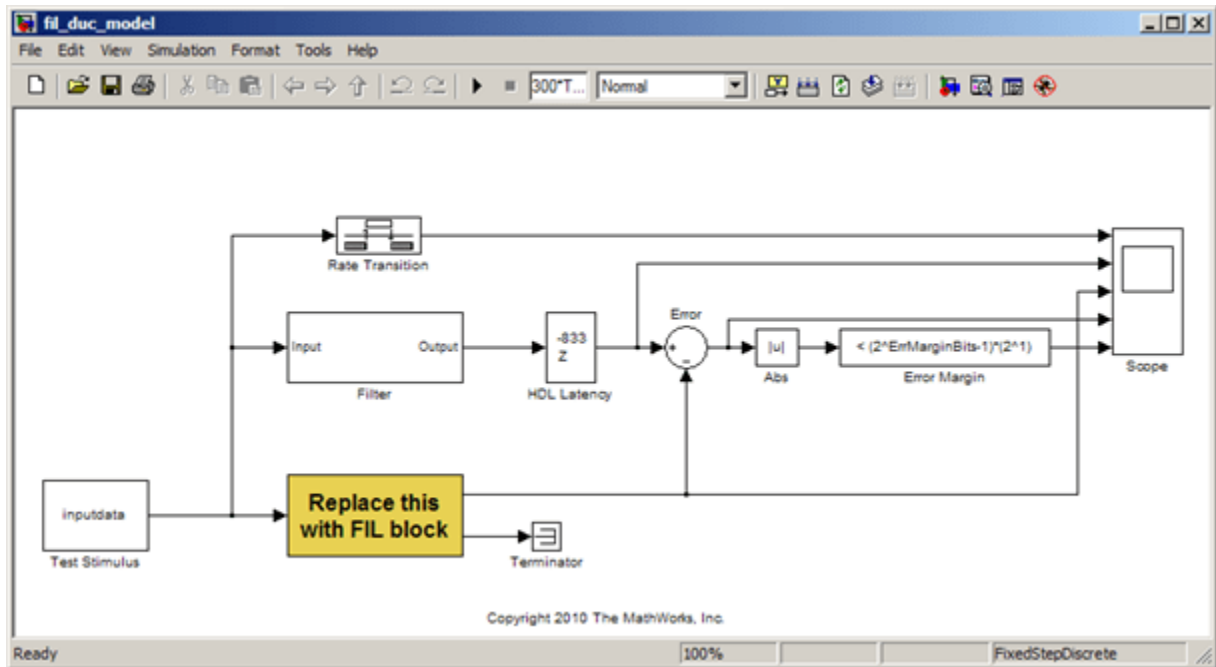
- Generates a FIL block in a new Simulink® model
- Opens a command-line window to compile the FPGA project and generate the FPGA programming file

The FPGA project compilation process takes several minutes. When the process is finished, you are prompted to close the command-line window. Close this window now.

Configure FIL Block

To prepare for FPGA-in-the-Loop simulation, follow the steps below to configure the FIL block.

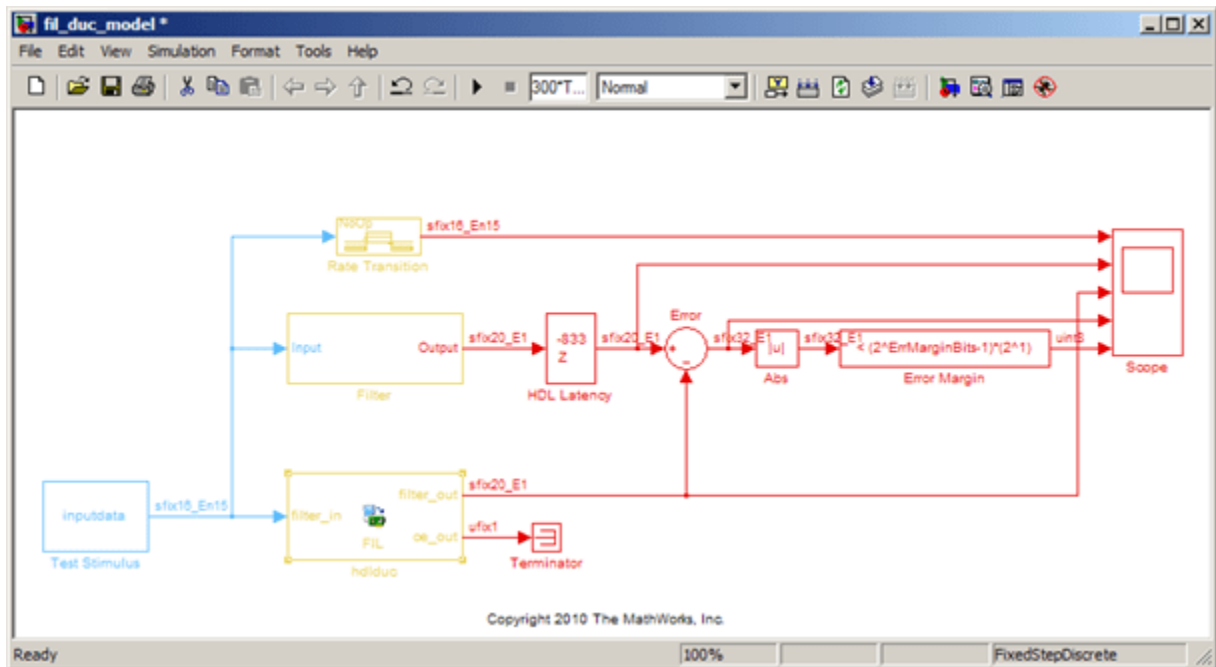
1. Open the **Open Model** button to open the test bench model and copy the generated FIL block into the model.



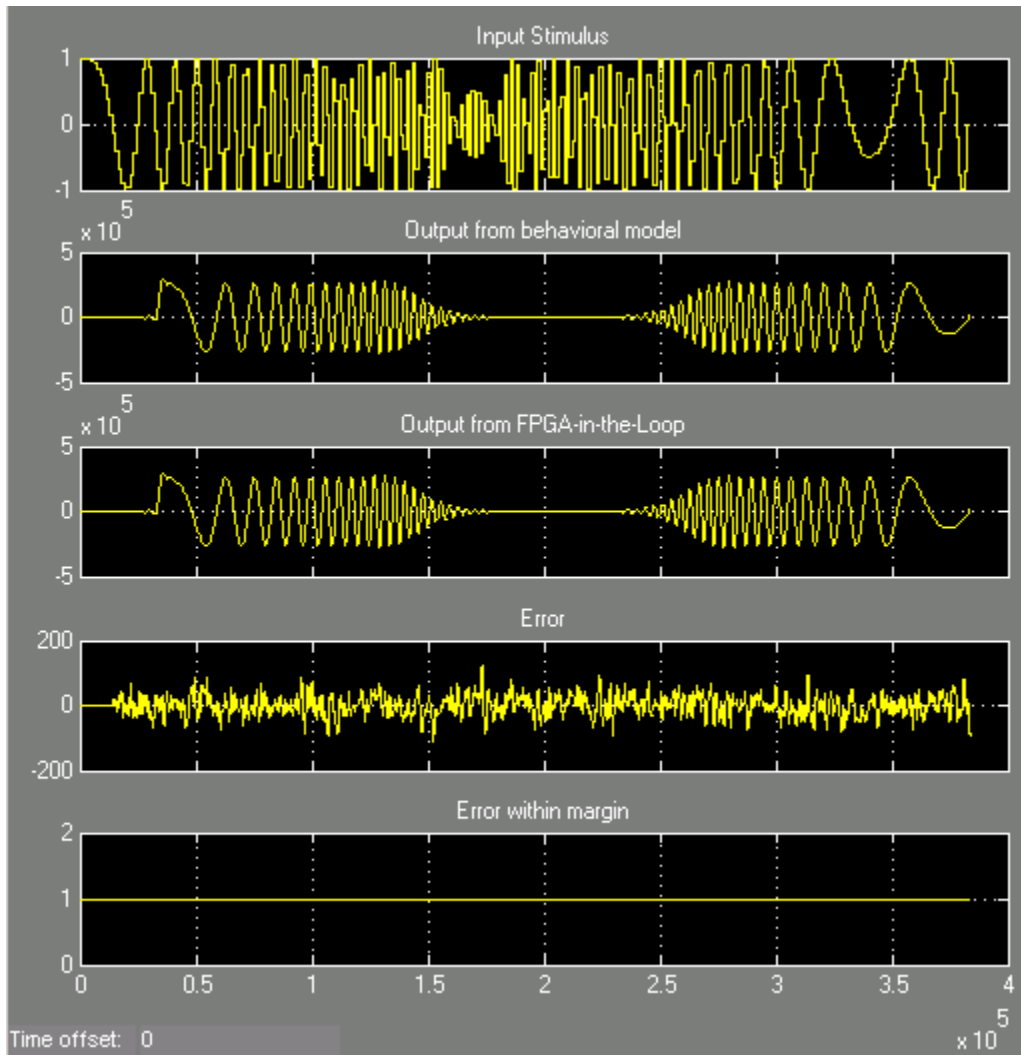
2. Double-click the FIL block to open the block mask. Click **Load** to program the FPGA with the generated programming file.
3. Under Runtime Options, change **Overclocking factor** to 128. This specifies that an input value is sampled 128 times by the FPGA clock before the input changes value.
4. On the FIL block mask, click on the Signal Attributes tab. Change the data type for filter_out to `fixdt(1,20,-1)` to match the data type of the behavioral filter block.
5. Click **OK** to close the block mask.

Verify Generated Filter

In this example, the generated filter running on FPGA is compared to a behavioral filter block. Delays are added to the output of the behavioral filter to match the HDL latency of the generated filter.



Run simulation. Observe the output waveforms from the behavioral filter block, the FIL block, and the error margin. Because the behavioral filter block does not have pipeline registers, there are small differences between the behavioral filter block output and the FIL block output. These errors are within the error margin.



This concludes the example.

Untimed SystemC/TLM Simulation

This example highlights the use of the 'Untimed' timing mode when you generate a SystemC™/TLM component from a Simulink® model using the tlmgenerator target for either Simulink® Coder™ or Embedded Coder®.

In Simulink® models, the movement of data between sources and sinks is controlled by signal sample rates and a centralized timing solver. In SystemC/TLM models, interactions between data sinks and sources are controlled by the SystemC simulation kernel and time advances through SC_THREADS cooperatively yielding control to another thread through wait calls.

For untimed SystemC/TLM simulations, the model ignores annotated delays for communication interfaces and processing. In such models, the goal is only to have a simulation that yields correct results by ensuring that initiators and targets can successfully synchronize the movement of data. There is no attempt to evaluate the performance of a deployed system. No simulation time should elapse while in an untimed simulation because the synchronization is completely event based.

For this example we use a Simulink model of a FIR filter as the basis of the SystemC/TLM generation.

Products required to run this demo:

- SystemC 2.3.1 (includes the TLM library)
- For code verification, make and a compatible GNU-compiler, gcc, in your path on Linux®, or Visual Studio® compiler in your path on Windows®

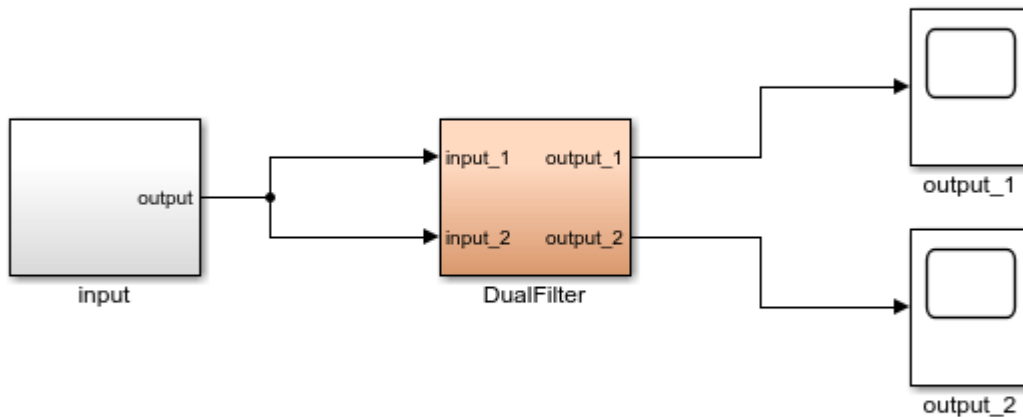
Note: The example includes a code generation build procedure. Simulink does not permit you to build programs in the MATLAB installation area. If necessary, change to a working directory that is not in the MATLAB installation area prior to starting any build.

1. Open the Preconfigured Model

To open the **TLM untimed testbench model**, click the **Open Model** button.

The following model opens in Simulink.

TLM Generator Untimed Testbench Example

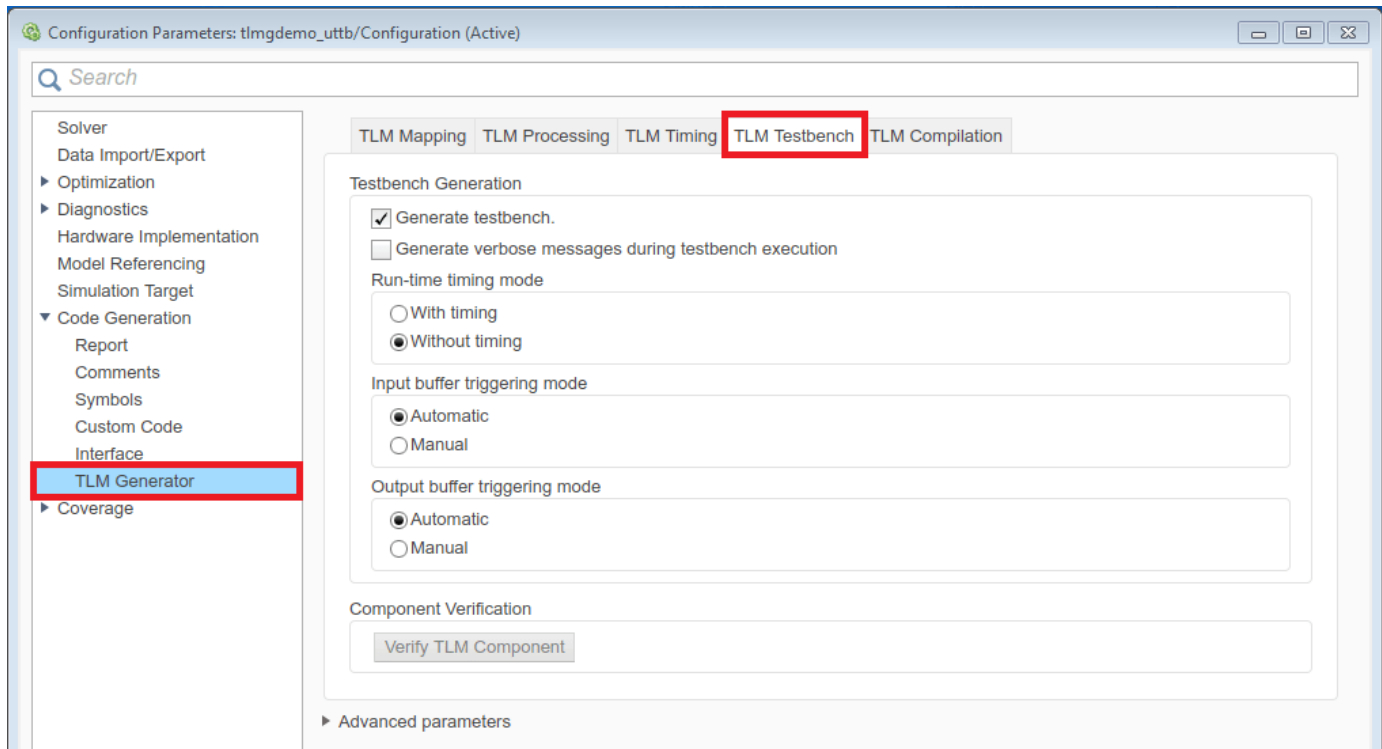


This model shows the generation of a testbench for a stand-alone SystemC verification of a generated TLM component for a dual 4 taps FIR filter.

Copyright 1994-2013 The MathWorks, Inc.

2. Review the TLM Generator Target Configuration Options

In the Simulink Toolstrip, select **Model Settings** from the **Modeling** tab. In the **Configuration Parameters** dialog box, select the **TLM Generator** view and the tab **TLM Testbench** and review the testbench settings as shown in the following image. Select the verbose testbench message checkbox to see the full log of initiator/target interaction in the SystemC/TLM simulation. Since this will be thousands of lines, if desired, deselect the option to get a terse log of the SystemC/TLM simulation.



3. Build the Model

In the model window, right-click on the DualFilter block and select **C/C++ Code > Generate Code for this Subsystem** in the context menu to start the TLM component and testbench generation. Or you can execute the following command in the MATLAB command window:

```
>> slbuild('tlmgdemo_uttb/DualFilter')
```

The generation is completed when the following message appears in the MATLAB command window:

```
### Starting Simulink Coder build procedure for model: DualFilter
### Successful completion of Simulink Coder build procedure for model: DualFilter
```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
DualFilter	Code generated.	Code generation information file does not exist.

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 17.255s
```

4. Open the Generated Files

Open the generated testbench source code in the MATLAB web browser by clicking on 'DualFilter_uttb_tlm_tb.cpp' in the generated report or in the MATLAB Editor:

- DualFilter_VP/DualFilter_uttb_tlm_tb/src/DualFilter_uttb_tlm_tb.cpp

- DualFilter_VP/DualFilter_uttb_tlm_tb/src/mw_support_tb.cpp

5. Review the Generated Code

The specification of 'Without timing' results in the testbench instructing the TLM component to run in untimed mode by means of a special configuration interface, `mw_backdoorcfg_IF`. As the code indicates, the choice of timing mode is dynamic and can be changed during the course of simulation. Since this choice is purely a simulation construct, it is not programmed by means of a "front-door" TLM transaction.

Additionally, as the code indicates, the testbench must setup a local helper object with the correct timing mode in order to ensure that its initiator threads use the proper synchronization in the TLM transaction calls.

```

69 // -----
70 // Set core TLM component timing mode: Untimed or Timed.
71 // -----
72 // Set whether the target should return delays based on the specified timing
73 // parameters in the 'TLM Generation' options or to just return '0' for all
74 // transaction and algorithm delays.
75 //
76 // In your TLM environment, this can be dynamically programmed during
77 // simulation execution, but for this testbench we set it just once at the
78 // beginning of the simulation based on the 'TLM Testbench' configset
79 // options.
80 //
81 mw_backdoorcfg->SetTimingParam(mw_backdoorcfg_IF::UNTIMED);
82
83 // mw_backdoorcfg->SetTimingParam(mw_backdoorcfg_IF::TIMED);
84
85 // The timing mode also needs to be set up in a testbench support object.
86 //
87 // With timing      Testbench timing mode
88 // -----          -----
89 //      No          Untimed
90 //      Yes         LooselyTimed
91 //
92 // Untimed          : Sync to '0' delays after every transaction and for
93 //                   every timed wait.
94 // Loosely Timed    : Sync to the delays returned from the target after
95 //                   every transaction and use non-zero times for timed
96 //                   waits.
97 //
98 mw_sync_tb sync(m_utils, mw_sync_tb::Untimed);
99

```

The implementation of the untimed synchronization class, `mw_syncuntimed_tb`, in the `mw_support_tb.cpp` file shows the details of the various synchronization calls used by the initiator threads and the TLM transactions. As previously stated, the calls utilize waits of `SC_ZERO_TIME` and no delays are allowed to accumulate.


```

614 void mw_syncuntimed_tb::incLocalTime(sc_time t)
615 {
616     syncToLocalTime();
617 }
618
619 void mw_syncuntimed_tb::setLocalTime(sc_time t)
620 {
621     // do nothing
622 }
623
624 sc_time mw_syncuntimed_tb::getLocalTime()
625 {
626     return SC_ZERO_TIME;
627 }
628
629 void mw_syncuntimed_tb::syncToExplicitTime(sc_time t)
630 {
631     m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS,
632                     "##\t waiting for explicit amount of time (time = 0 ns + 0 ns).\n");
633     wait(SC_ZERO_TIME);
634 }
635
636 void mw_syncuntimed_tb::syncToLocalTime()
637 {
638     m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS,
639                     "##\t syncing to local time offset (delay = 0 ns).\n");
640     wait(SC_ZERO_TIME);
641 }
642
643 void mw_syncuntimed_tb::syncToDataReady(sc_core::sc_in<bool> & irq)
644 {
645     m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS,
646                     "##\t syncing to interrupt signal for data ready in output buffer...\n");
647     if (irq.read() == true) {
648         m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS,
649                         "##\t...saw interrupt already asserted\n");
650     } else {
651         wait(irq.posedge_event());
652         m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS, "##\t...saw interrupt\n");
653     }
654 }
655

```

6. Verify the Generated Code

In the Configuration Parameters dialog box, go in TLM Testbench configuration parameter dialog and push the **Verify TLM Component** button to run the generated testbench in the untimed timing mode or in the MATLAB command window execute:

```
>> verifyTlmgDemoModel('uttb')
```

This action will:

- build the generated code
- run Simulink to capture input stimulus and expected results
- convert the Simulink data to TLM vectors
- run the stand-alone SystemC/TLM testbench executable
- convert the TLM results back to Simulink data
- perform a data comparison

- generate a Figure window for any signals that had data mis-compares

Loosely-Timed SystemC/TLM Simulation

This example highlights the use of the 'Timed' timing mode when you generate a SystemC™/TLM component from a Simulink® model using the tlmgenerator target for either Simulink® Coder™ or Embedded Coder®.

In Simulink® models, the movement of data between sources and sinks is controlled by signal sample rates and a centralized timing solver. In SystemC/TLM models, interactions between data sinks and sources are controlled by the SystemC simulation kernel and time advances through SC_THREADS cooperatively yielding control to another thread through `wait` calls.

For timed SystemC/TLM simulations the model adheres to the annotated delays for communication interfaces and processing. For loosely-timed simulations, the model does not allow threads to "run ahead" in time and so there is no idea of a "local time" that is different than some other thread's sense of time. Instead, initiators immediately synchronize to the returned delays from TLM transactions and generally perform non-zero timed waits in order to execute polling loops and synchronization-on-demand. Simulation time will advance in a loosely-timed model to reflect communication, processing, and polling delays in the system. The goal of the model, in addition to proper synchronization of data movement as with the untimed system, is to have a functionally correct simulation with a fast wall-clock execution times. This speed allows for near real-time software development on the SystemC/TLM architectural model.

For this example we use a Simulink model of a FIR filter as the basis of the SystemC/TLM generation.

Products required to run this demo:

- SystemC 2.3.1 (includes the TLM library)
- For code verification, make and a compatible GNU-compiler, gcc, in your path on Linux®, or Visual Studio® compiler in your path on Windows®

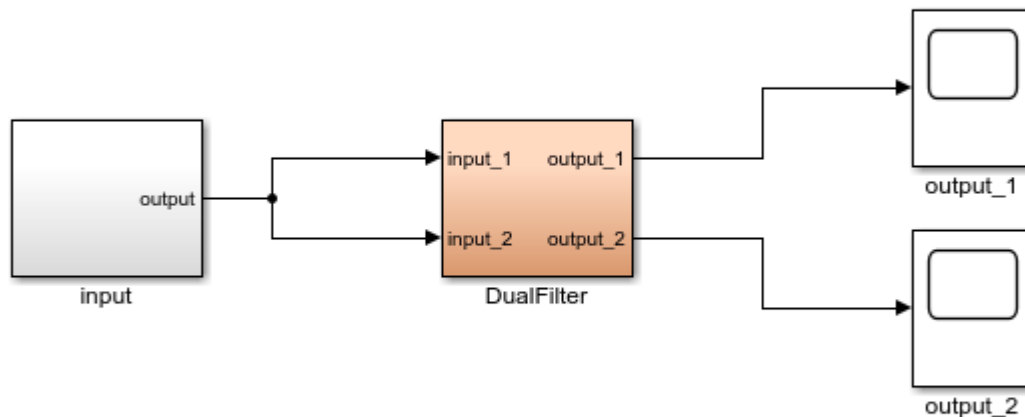
Note: The example includes a code generation build procedure. Simulink does not permit you to build programs in the MATLAB installation area. If necessary, change to a working directory that is not in the MATLAB installation area prior to starting any build.

1. Open the Preconfigured Model

To open the **TLM loosely-timed testbench model**, click the **Open Model** button.

The following model opens in Simulink.

TLM Generator Loosely-Timed Testbench Example

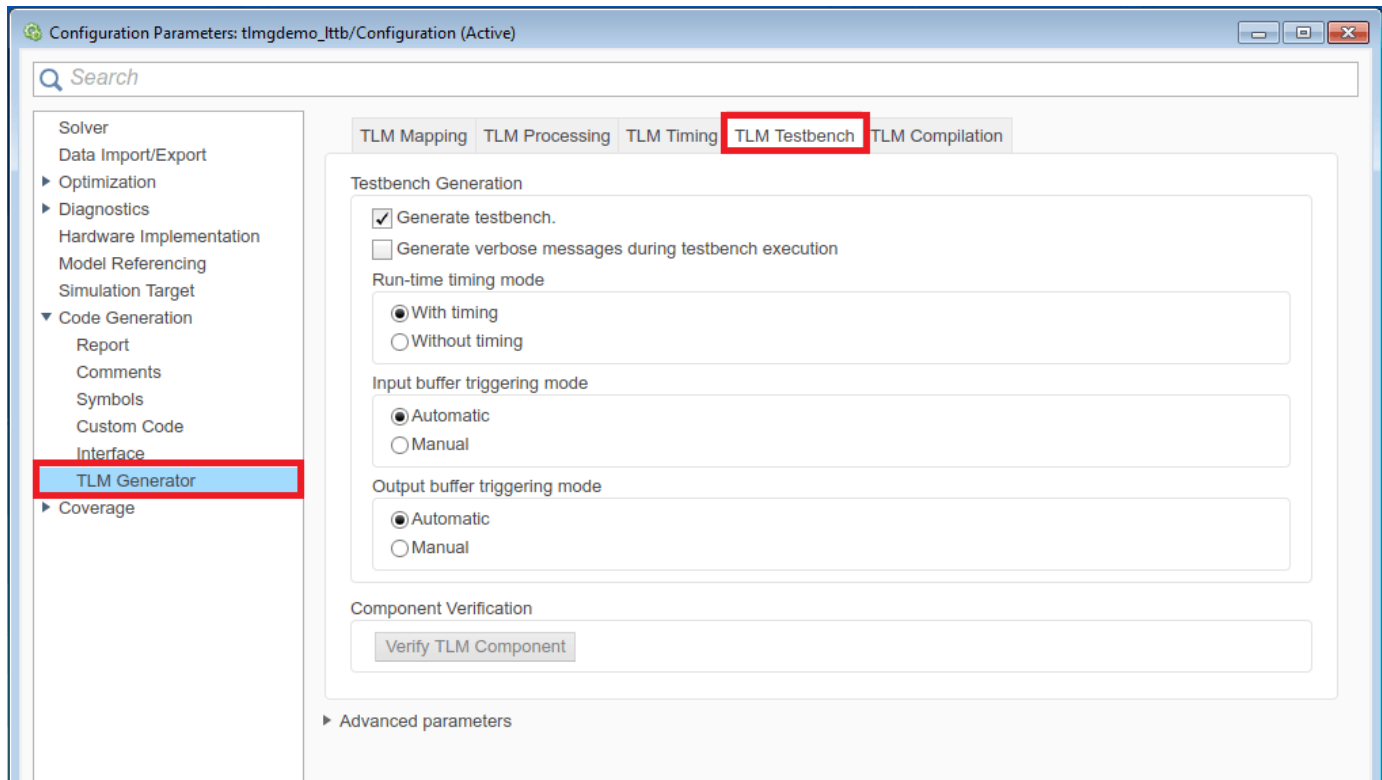


This model shows the generation of a testbench for a stand-alone SystemC verification of a generated TLM component for a dual 4 taps FIR filter.

Copyright 1994-2013 The MathWorks, Inc.

2. Review the TLM Generator Target Configuration Options

In the Simulink Toolstrip, select **Model Settings** from the **Modeling** tab. In the **Configuration Parameters** dialog box, select the **TLM Generator** view and then tab **TLM Testbench** view and review the testbench settings as shown in the following image. Select the verbose testbench message checkbox to see the full log of initiator/target interaction in the SystemC/TLM simulation. Since this will be thousands of lines, if desired, deselect the option to get a terse log of the SystemC/TLM simulation.



3. Build the Model

In the model window, right-click on the DualFilter block and select **C/C++ Code > Generate Code for this Subsystem** in the context menu to start the TLM component and testbench generation. Or you can execute the following command in the MATLAB command window:

```
>> slbuild('tlmgdemo_lttb/DualFilter')
```

The generation is completed when the following message appears in the MATLAB command window:

```
### Starting Simulink Coder build procedure for model: DualFilter
### Successful completion of Simulink Coder build procedure for model: DualFilter
```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
DualFilter	Code generated.	Code generation information file does not exist.

1 of 1 models built (0 models already up to date)

Build duration: 0h 0m 14.374s

4. Open the Generated Files

Open the generated testbench source code in the MATLAB web browser by clicking on 'DualFilter_lttb_tlm_tb.cpp' in the generated report or in the MATLAB Editor:

- DualFilter_VP/DualFilter_lttb_tlm_tb/src/DualFilter_lttb_tlm_tb.cpp
- DualFilter_VP/DualFilter_lttb_tlm_tb/src/mw_support_tb.cpp

5. Review the Generated Code

The specification of 'With timing' results in the testbench instructing the TLM component to run in timed mode by means of a special configuration interface, `mw_backdoorcfg_IF`. As the code indicates, the choice of timing mode is dynamic and can be changed during the course of simulation. Since this choice is purely a simulation construct, it is not programmed by means of a "front-door" TLM transaction.

Additionally, as the code indicates, the testbench must setup a local helper object with the correct timing mode in order to ensure that its initiator threads use the proper synchronization in the TLM transaction calls.

```

69 // -----
70 // Set core TLM component timing mode: Untimed or Timed.
71 // -----
72 // Set whether the target should return delays based on the specified timing
73 // parameters in the 'TLM Generation' options or to just return '0' for all
74 // transaction and algorithm delays.
75 //
76 // In your TLM environment, this can be dynamically programmed during
77 // simulation execution, but for this testbench we set it just once at the
78 // beginning of the simulation based on the 'TLM Testbench' configset
79 // options.
80 //
81 // m_backdoorcfg->SetTimingParam(mw_backdoorcfg_IF::UNTIMED);
82 m_backdoorcfg->SetTimingParam(mw_backdoorcfg_IF::TIMED);
83
84 // The timing mode also needs to be set up in a testbench support object.
85 //
86 // With timing      Testbench timing mode
87 // -----
88 //      No          Untimed
89 //      Yes          LooselyTimed
90 //
91 // Untimed          : Sync to '0' delays after every transaction and for
92 //                   every timed wait.
93 // Loosely Timed    : Sync to the delays returned from the target after
94 //                   every transaction and use non-zero times for timed
95 //                   waits.
96 //
97 // mw_sync_tb       sync(m_utils, mw_sync_tb::Untimed);
98 mw_sync_tb sync(m_utils, mw_sync_tb::LooselyTimed);
99

```

The implementation of the loosely-timed synchronization class, `mw_syncfunctimed_tb`, in the `mw_support_tb.cpp` file shows the details of the various synchronization calls used by the initiator

threads and the TLM transactions. Unlike untimed simulations, the calls utilize waits of non-zero times, but, as with untimed simulations, no "local time" delays are allowed to accumulate.

```

666 void mw_syncfunctimed_tb::incLocalTime(sc_time t)
667 {
668     m_delay += t;
669     syncToLocalTime();
670 }
671
672 void mw_syncfunctimed_tb::setLocalTime(sc_time t)
673 {
674     m_delay = t;
675 }
676
677 sc_time mw_syncfunctimed_tb::getLocalTime()
678 {
679     return m_delay;
680 }
681
682 void mw_syncfunctimed_tb::syncToExplicitTime(sc_time t)
683 {
684     m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS,
685                     "##\t waiting for explicit amount of time (time = %s + %s).\n",
686                     m_delay.to_string().c_str(), t.to_string().
687                     c_str() );
688     wait(m_delay);
689     wait(t);
690     m_delay = SC_ZERO_TIME;
691 }
692
693 void mw_syncfunctimed_tb::syncToLocalTime()
694 {
695     m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS,
696                     "##\t syncing to local time offset (delay = %s).\n",
697                     (m_delay.to_string()).c_str() );
698     wait(m_delay);
699     m_delay = SC_ZERO_TIME;
700 }
701
702 void mw_syncfunctimed_tb::syncToDataReady(sc_core::sc_in<bool> & irq)
703 {
704     m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS,
705                     "##\t syncing to interrupt signal for data ready in output buffer...\n");
706     if (irq.read() == true) {
707         m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS,
708                         "##\t...saw interrupt already asserted\n");
709     } else {
710         wait(irq.posedge_event());
711         m_utils.printMsg(TLMG_PRINT_SYNC_TO_FUNCS, "##\t...saw interrupt\n");
712     }
713 }

```

6. Verify the Generated Code

In the Configuration Parameters dialog box, go in TLM Testbench configuration parameter dialog and push the **Verify TLM Component** button to run the generated testbench in the loosely-timed timing mode or in the MATLAB command window execute:

```
>> verifyTlmgDemoModel('lttb')
```

This action will:

- build the generated code

- run Simulink to capture input stimulus and expected results
- convert the Simulink data to TLM vectors
- run the stand-alone SystemC/TLM testbench executable
- convert the TLM results back to Simulink data
- perform a data comparison
- generate a Figure window for any signals that had data mis-compares

No Memory Map Option

This example highlights the use of the no memory map option when you generate a SystemC™/TLM component from a Simulink® model using the tlmgenerator target for either Simulink Coder or Embedded Coder™.

In Simulink, each block input or output is bound point-to-point to another block. In SystemC/TLM, each component communicates through a TLM socket. This socket handles all the incoming and outgoing communication formatted inside TLM transactions. Because each system handles communication differently, we must define a communication interface for the SystemC/TLM component when it is generated from a Simulink model. Depending on the intended use of this SystemC/TLM component, this communication interface could require building a memory map (or address) for each input/output in the component. This memory map may be simple or detailed.

The no memory map option generates a TLM component with only one read and one write register without any address. The Simulink model inputs are bound to the write register and the outputs are bound to the read register. When created with this option, the generated TLM component could be used in a virtual platform (VP) as a standalone component in a test bench, as a direct bound co-processing unit, or it could be attached to a communication channel using an adapter.

For this example we use a Simulink model of a FIR filter as the basis of the SystemC/TLM generation.

Requirements to run this example:

- SystemC 2.3.1 (includes the TLM library)

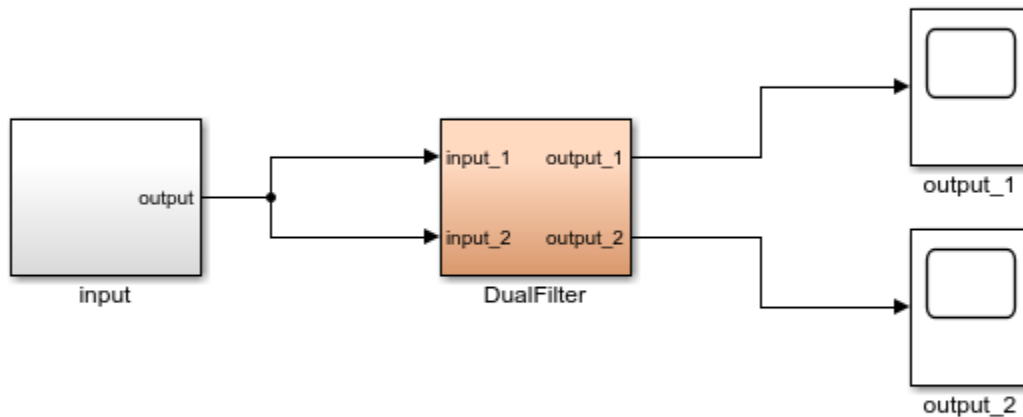
Note: The example includes a code generation build procedure. Simulink does not permit you to build programs in the MATLAB installation area. If necessary, change to a working directory that is not in the MATLAB installation area prior to starting any build.

1. Open the Preconfigured Model

To open the **FIR Filter model with no memory map**, click the **Open Model** button.

The following model opens in Simulink.

TLM Generator No Memory Map Example

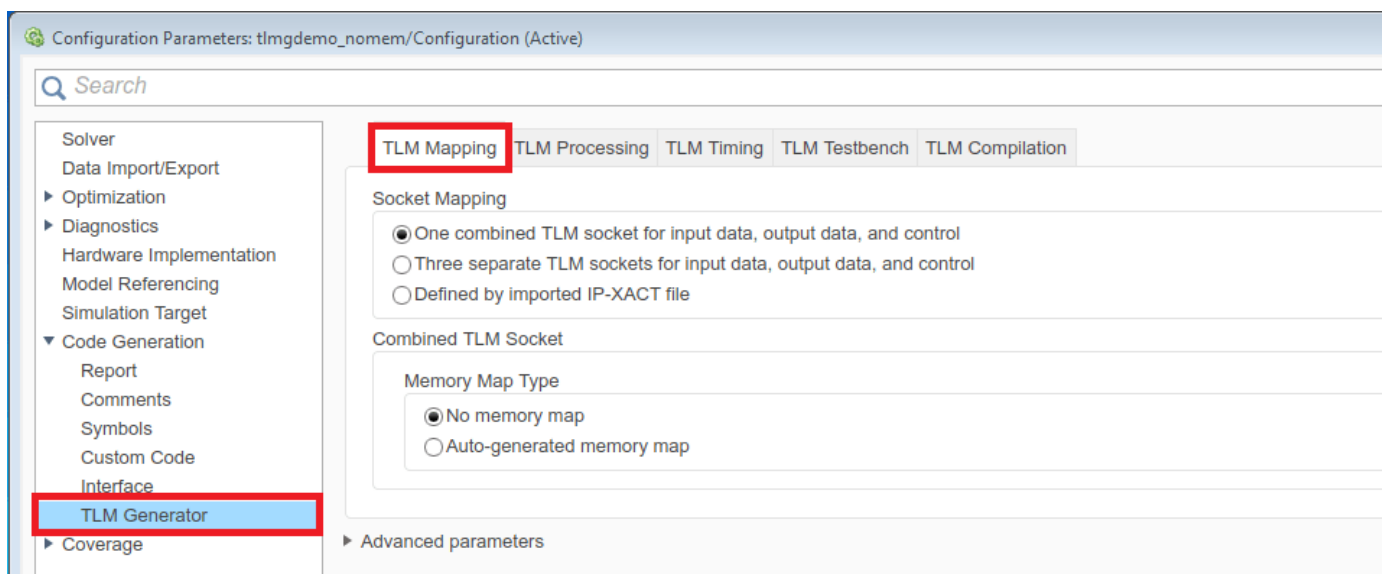


This model shows the TLM component with no-memory map generated for a dual 4 taps FIR filter.

Copyright 1994-2013 The MathWorks, Inc.

2. Set the Option to No Memory Map

In the **Configuration Parameters** dialog box, select the **TLM Generator** view in the left-hand pane. In the **TLM Mapping** tab, under **Socket Mapping**, select **One combined TLM socket for input data, output data and control**. Under **Combined TLM Socket**, select **No memory map**, as shown in the following image.



3. Set the Other Options

Set the remaining TLM Generation options according to your preferences (or leave the default values as they are) and click **OK** to apply these settings and exit the **Configuration Parameters** dialog box.

4. Build the Model

In the model window, right-click on the DualFilter block and select **C/C++ Code > Generate Code for this Subsystem** in the context menu to start the TLM component generation. Or you can execute the following command in the MATLAB command window:

```
>> slbuild('tlmgdemo_nomem/DualFilter');
```

The generation is completed when the following message appears in the MATLAB command window:

```
### Starting Simulink Coder build procedure for model: DualFilter
### Successful completion of Simulink Coder build procedure for model: DualFilter
```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
DualFilter	Code generated.	Code generation information file does not exist.

1 of 1 models built (0 models already up to date)

Build duration: 0h 0m 13.925s

5. Open the Generated Files

Open the following file in the MATLAB Editor:

- DualFilter_VP/DualFilter_nomem_tlm/include/DualFilter_nomem_tlm_def.h

6. Observe the Generated Code

Find the following lines of code. These lines represent the input and output definitions of the FIR Filter when generated as a TLM component with no memory map.

```
8 //===== DualFilter_combined_IN_BANK BANK =====
9 //===== DATA =====
10 #define input_1_REG_DIM          1
11 #define input_2_REG_DIM          1
12
13 typedef real_T input_1_REG_T;
14 typedef real_T input_2_REG_T;
15
16 //Type of the input data
17 typedef struct {
18     input_1_REG_T input_1_REG;
19     input_2_REG_T input_2_REG;
20 } DualFilter_combined_IN_BANK_T;
21
```

To provide a set of input to the generated TLM component, an TLM initiator should send a write TLM transaction with no address and with a payload containing a datum of type **DualFilter_combined_IN_BANK_T**.

```
22 //===== DualFilter_combined_OUT_BANK BANK =====
23 //===== DATA =====
24 #define output_1_REG_DIM          1
25 #define output_2_REG_DIM          1
26
27 typedef real_T output_1_REG_T;
28 typedef real_T output_2_REG_T;
29
30 //Type of the output data
31 typedef struct {
32     output_1_REG_T output_1_REG;
33     output_2_REG_T output_2_REG;
34 } DualFilter_combined_OUT_BANK_T;
```

To obtain a set of output from the generated TLM component, an TLM initiator should send a read TLM transaction with no address and with a payload containing a datum of type **DualFilter_combined_OUT_BANK_T**.

Auto-Generated Memory Map with Single Address Option

This example highlights the use of the auto-generated memory map with single address option when you generate a SystemC™/TLM component from a Simulink® model using the tlmgenerator target for either Simulink Coder or Embedded Coder™.

In Simulink, each component input or output is bound point-to-point to another component. In SystemC/TLM, each component communicates through a TLM socket. This socket handles all the incoming and outgoing communication formatted inside TLM transactions. Because each system handles communication differently, we must define a communication interface for the SystemC/TLM component when it is generated from a Simulink model. Depending on the intended use of this SystemC/TLM component, this communication interface could require building a memory map (or address) for each input/output in the component. This memory map may be simple or detailed.

The auto-generated memory map with single address option generates a TLM component with only one read and one write register with one address each. The Simulink model inputs are bound to the write register and the outputs are bound to the read register. When created with this option, the generated TLM component could be used in a virtual platform (VP) as a standalone component in a test bench or it could be attached to a communication channel.

For this example we use a Simulink model of a FIR filter as the basis of the SystemC/TLM generation.

Requirements to run this example:

- SystemC 2.3.1 (includes the TLM library)

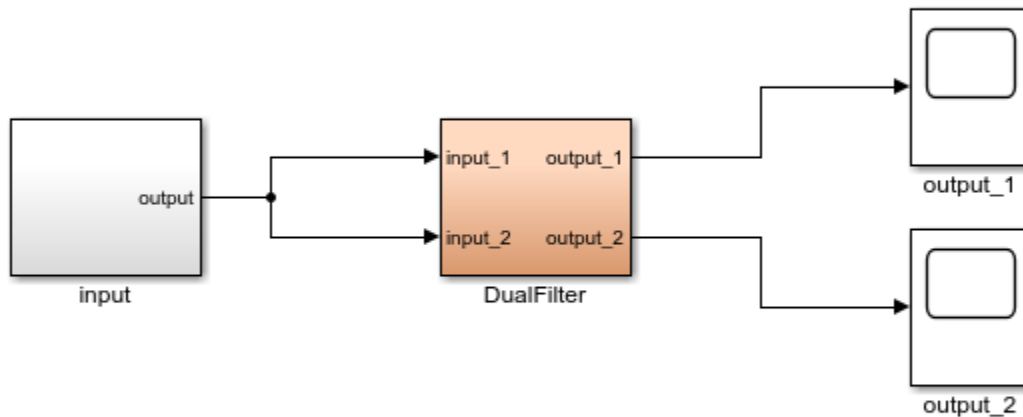
Note: The example includes a code generation build procedure. Simulink does not permit you to build programs in the MATLAB installation area. If necessary, change to a working directory that is not in the MATLAB installation area prior to starting any build.

1. Open the Preconfigured Model

To open the **FIR Filter model with auto-generated memory map with single address**, click the **Open Model** button.

The following model opens in Simulink.

TLM Generator Auto Generated Memory Map with Single Address Example

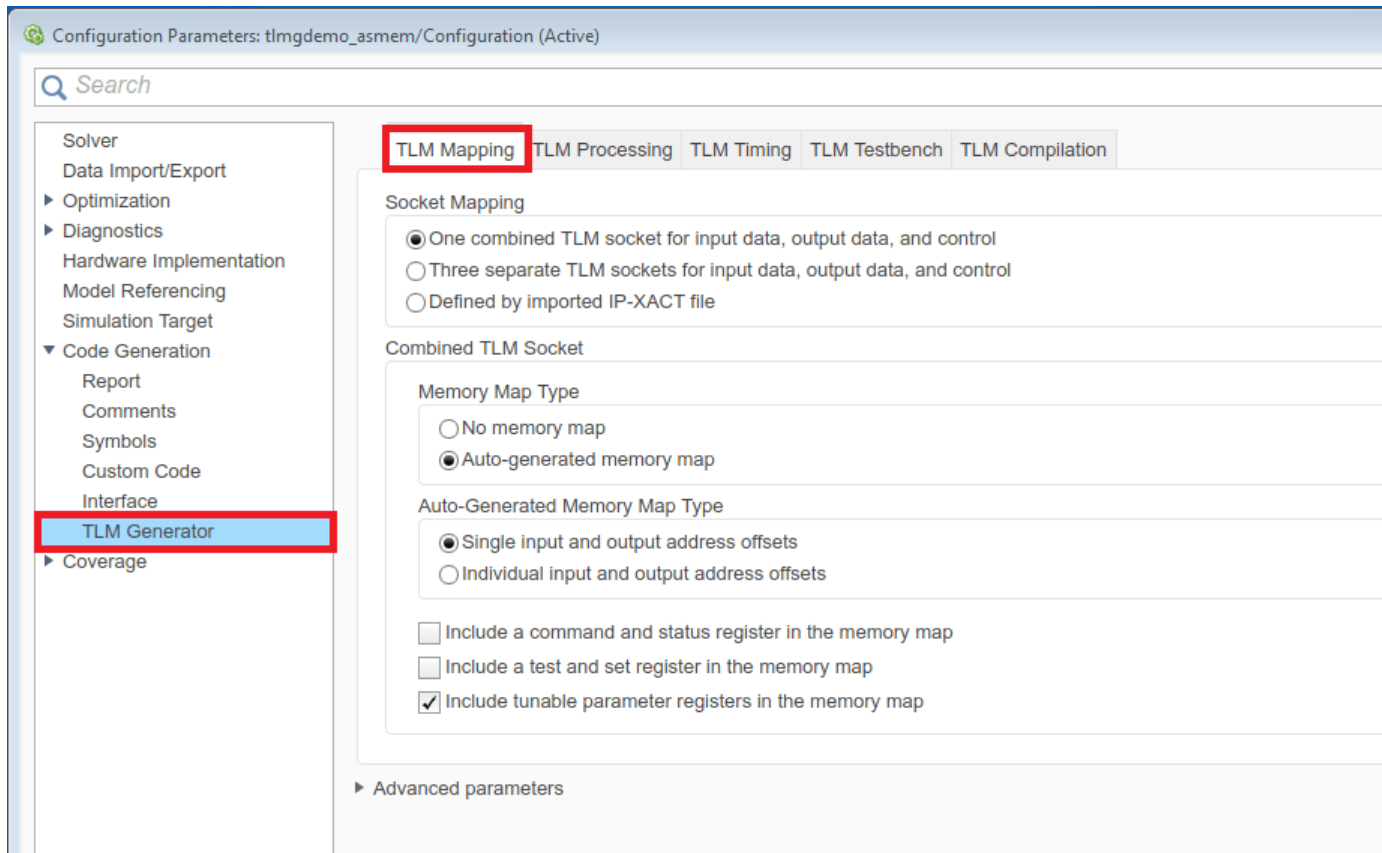


This model shows the TLM component with single input/output memory map generated for a dual 4 taps FIR filter.

Copyright 1994-2013 The MathWorks, Inc.

2. Set the Option to Auto-Generated Memory Map with Single Address

In the **Configuration Parameters** dialog box, select the **TLM Generator** view in the left-hand pane. In the **TLM Mapping** tab, under **Socket Mapping** option select **One combined TLM socket for input data, output data and control**. Under **Combined TLM Socket**, select **Auto-generated memory map** and **Single input and output address offsets**, as shown in the following image.



3. Set the Other Options

Set the remaining **TLM Generation** options according to your preferences (or leave the default values as they are) and click **OK** to apply these settings and exit the **Configuration Parameters** dialog box.

4. Build the Model

In the model window, right-click on the DualFilter block and select **C/C++ Code > Generate Code for this Subsystem** in the context menu to start the TLM component generation. Or you can execute the following command in the MATLAB command window:

```
>> slbuild('tlmgdemo_asmem/DualFilter');
```

The generation is completed when the following message appears in the MATLAB command window:

```
### Starting Simulink Coder build procedure for model: DualFilter
### Successful completion of Simulink Coder build procedure for model: DualFilter
```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
DualFilter	Code generated.	Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 18.128s

5. Open the Generated Files

Open the following files in the MATLAB Editor:

- DualFilter_VP/DualFilter_asmem_tlm/include/DualFilter_asmem_tlm_def.h

6. Observe the Generated Code

Find the following lines of code. These lines represent the input and output definitions of the FIR Filter when generated as a TLM component with single address memory map.

```

 8 //===== DualFilter_combined_IN_BANK BANK =====
 9 //Bank base address
10 #define DualFilter_combined_IN_BANK_ADDR 0x00000000
11
12 //===== DATA =====
13 #define input_1_REG_DIM                1
14 #define input_2_REG_DIM                1
15
16 typedef real_T input_1_REG_T;
17 typedef real_T input_2_REG_T;
18
19 //Type of the input data
20 typedef struct {
21     input_1_REG_T input_1_REG;
22     input_2_REG_T input_2_REG;
23 } DualFilter_combined_IN_BANK_T;

```

To provide a set of input to the generated TLM component, an TLM initiator should send a write TLM transaction with a payload containing a data of type **DualFilter_combined_IN_BANK_T** at the address **DualFilter_combined_IN_BANK_ADDR**.


```
25 //===== DualFilter_combined_OUT_BANK BANK =====
26 //Bank base address
27 #define DualFilter_combined_OUT_BANK_ADDR 0x00000010
28
29 //===== DATA =====
30 #define output_1_REG_DIM                1
31 #define output_2_REG_DIM                1
32
33 typedef real_T output_1_REG_T;
34 typedef real_T output_2_REG_T;
35
36 //Type of the output data
37 typedef struct {
38     output_1_REG_T output_1_REG;
39     output_2_REG_T output_2_REG;
40 } DualFilter_combined_OUT_BANK_T;
```

To obtain a set of output from the generated TLM component, an TLM initiator should send a read TLM transaction with a payload containing a data of type **DualFilter_combined_OUT_BANK_T** at the address **DualFilter_combined_OUT_BANK_ADDR**.

Auto-Generated Memory Map with Individual Address Option

This example highlights the use of the auto-generated memory map with individual address option when you generate a SystemC™/TLM component from a Simulink® model using the tlmgenerator target for either Simulink Coder or Embedded Coder™.

In Simulink, each component input or output is bound point-to-point to another component. In SystemC/TLM, each component communicates through a TLM socket. This socket handles all the incoming and outgoing communication formatted inside TLM transactions. Because each system handles communication differently, we must define a communication interface for the SystemC/TLM component when it is generated from a Simulink model. Depending on the intended use of this SystemC/TLM component, this communication interface could require building a memory map (or address) for each input/output in the component. This memory map may be simple or detailed.

The auto-generated memory map with individual address option generates a TLM component with one read register per model output and write register per model input with individual addresses. Each Simulink model input is bound to its corresponding write register and each output is bound to its corresponding read register. When created with this option, the generated TLM component could be used in a virtual platform (VP) as a standalone component in a test bench or it could be attached to a communication channel.

For this example we use a Simulink model of a FIR filter as the basis of the SystemC/TLM generation.

Requirements to run this example:

- SystemC 2.3.1 (includes the TLM library)

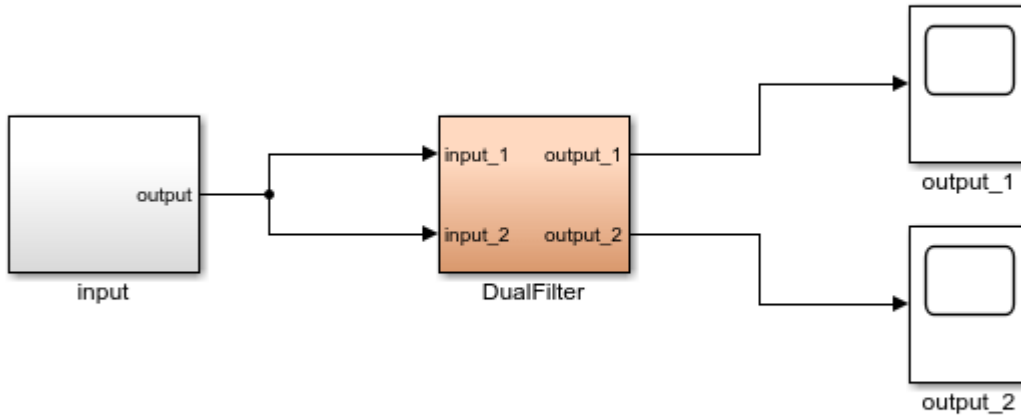
Note: The example includes a code generation build procedure. Simulink does not permit you to build programs in the MATLAB installation area. If necessary, change to a working directory that is not in the MATLAB installation area prior to starting any build.

1. Open the Preconfigured Model

To open the **FIR Filter model with auto-generated memory map with individual address**, click the **Open Model** button.

The following model opens in Simulink.

TLM Generator Auto Generated Memory Map with Individual Address Example

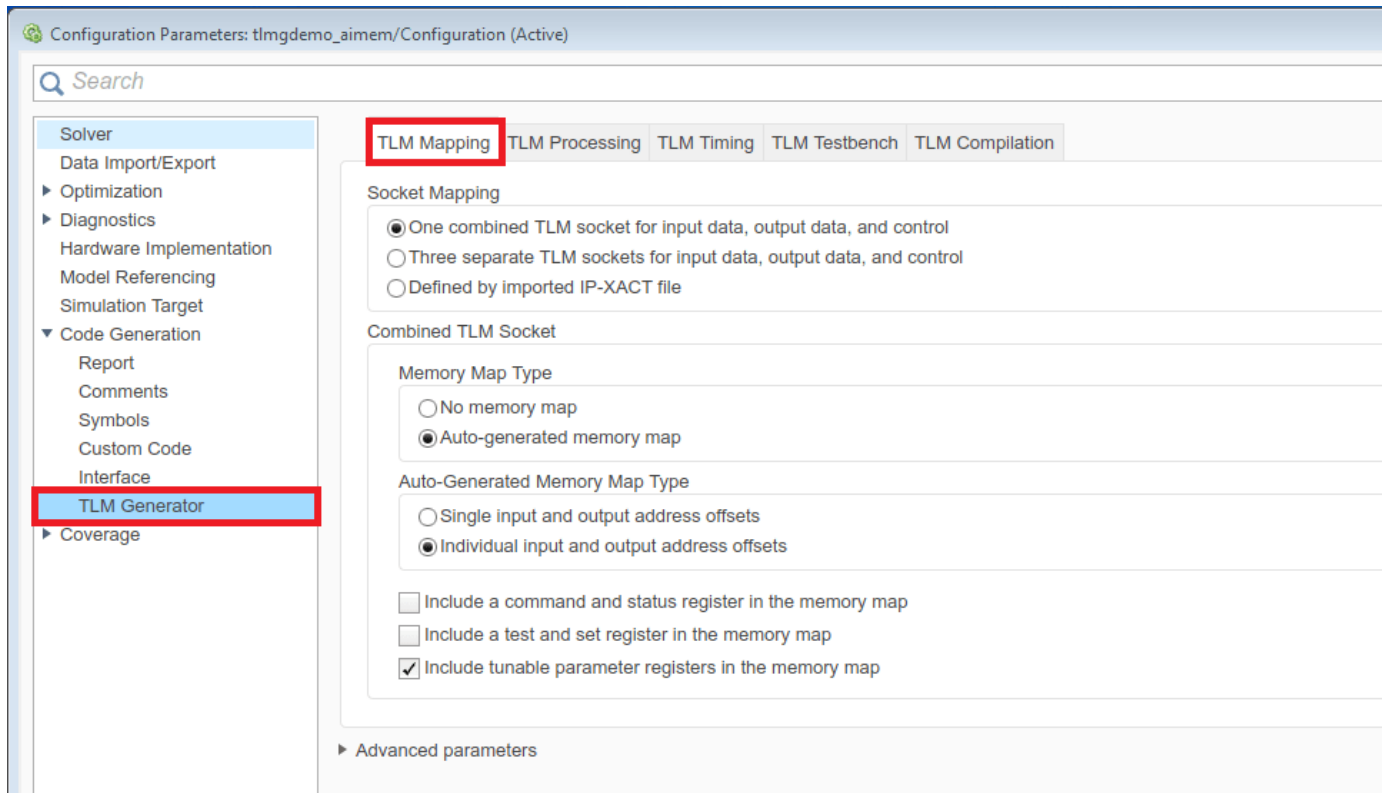


This model shows the TLM component with individual input/output memory map generated for a dual 4 taps FIR filter.

Copyright 1994-2013 The MathWorks, Inc.

2. Set the Option to Auto-Generated Memory Map with Individual Address

In the **Configuration Parameters** dialog box, select the **TLM Generation** view in the left-hand pane. In the **TLM Mapping** tab, under **Socket Mapping**, select **One combined TLM socket for input data, output data and control**. Under **Combined TLM Socket**, select **Auto-generated memory map** and **Individual input and output address offsets**, as shown in the following image.



3. Set the Other Options

Set the remaining **TLM Generation** options according to your preferences (or leave the default values as they are) and click **OK** to apply these settings and exit the **Configuration Parameters** dialog box.

4. Build the Model

In the model window, right-click on the DualFilter block and select **C/C++ Code > Generate Code for this Subsystem** in the context menu to start the TLM component generation. Or you can execute the following command in the MATLAB command window:

```
>> slbuild('tlmgdemo_aimem/DualFilter');
```

The generation is completed when the following message appears in the MATLAB command window:

```
### Starting Simulink Coder build procedure for model: DualFilter
### Successful completion of Simulink Coder build procedure for model: DualFilter
```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
DualFilter	Code generated.	Code generation information file does not exist.

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 32.266s
```

5. Open the Generated Files

Open the following files in the MATLAB Editor:

- DualFilter_VP/DualFilter_aimem_tlm/include/DualFilter_aimem_tlm_def.h

6. Observe the Generated Code

Find the following lines of code. These lines represent the input and output definitions of the FIR Filter when generated as a TLM component with individual address memory map.

```

8 //===== DualFilter_combined_IN_BANK BANK =====
9 //Bank base address
10 #define DualFilter_combined_IN_BANK_ADDR 0x00000000
11
12 //===== REGISTERS =====
13 //Address of the registers
14 #define input_1_REG_ADDR          0x00000000
15 #define input_2_REG_ADDR          0x00000008
16
17 //===== DATA =====
18 #define input_1_REG_DIM           1
19 #define input_2_REG_DIM           1
20
21 typedef real_T input_1_REG_T;
22 typedef real_T input_2_REG_T;
23

```

To provide a set of input to the generated TLM component, an TLM initiator should send two write TLM transactions:

- one with a payload containing a data of type **input_1_REG_T** at the address **DualFilter_combined_IN_BANK_ADDR + input_1_REG_ADDR**.
- one with a payload containing a data of type **input_2_REG_T** at the address **DualFilter_combined_IN_BANK_ADDR + input_2_REG_ADDR**.

```
24 //===== DualFilter_combined_OUT_BANK BANK =====
25 //Bank base address
26 #define DualFilter_combined_OUT_BANK_ADDR 0x00000010
27
28 //===== REGISTERS =====
29 //Address of the registers
30 #define output_1_REG_ADDR          0x00000000
31 #define output_2_REG_ADDR          0x00000008
32
33 //===== DATA =====
34 #define output_1_REG_DIM           1
35 #define output_2_REG_DIM           1
36
37 typedef real_T output_1_REG_T;
38 typedef real_T output_2_REG_T;
```

To obtain a set of output from the generated TLM component, an TLM initiator should send three read TLM transactions:

- one with a payload containing a data of type **output_1_REG_T** at the address **DualFilter_combined_OUT_BANK_ADDR + output_1_REG_ADDR**.
- one with a payload containing a data of type **output_2_REG_T** at the address **DualFilter_combined_OUT_BANK_ADDR + output_2_REG_ADDR**.

Imported IP-XACT Without Memory Map

This example shows how to use an imported IP-XACT specification without memory map when you generate a SystemC™/TLM component from a Simulink® model with the tlmgenerator target. You can use the tlmgenerator target for either Simulink Coder or Embedded Coder™.

In Simulink, each block input or output is bound point-to-point to another block. In SystemC/TLM, each component communicates through a TLM socket. This socket handles all the incoming and outgoing communication formatted inside TLM transactions. Because each system handles communication differently, you must define a communication interface for the SystemC/TLM component when it is generated from a Simulink model. Depending on the intended use of this SystemC/TLM component, this communication interface could require building a memory map (or address) for each input/output in the component. This memory map may be simple, detailed, or imported from an IP-XACT specification file.

The import IP-XACT option generates a TLM component according to an IP-XACT specification file. In this particular example, the IP-XACT file specified one tlm socket without memory map for the Simulink model inputs/output and one socket without memory map for the tunable parameters. The Simulink model inputs are bound to a unique write register, and the outputs are bound to a unique read register in the first tlm socket. The tunable parameters are bound to a unique read/write register in the second tlm socket. When created with this option, the generated TLM component can be used in a virtual platform (VP) as a standalone component in a test bench, or as a direct bound co-processing unit, or it could be attached to a communication channel using an adapter.

For this example, we use a Simulink model of a FIR filter as the basis for SystemC/TLM generation.

Requirements to run this example:

- SystemC 2.3.1 (includes the TLM library)

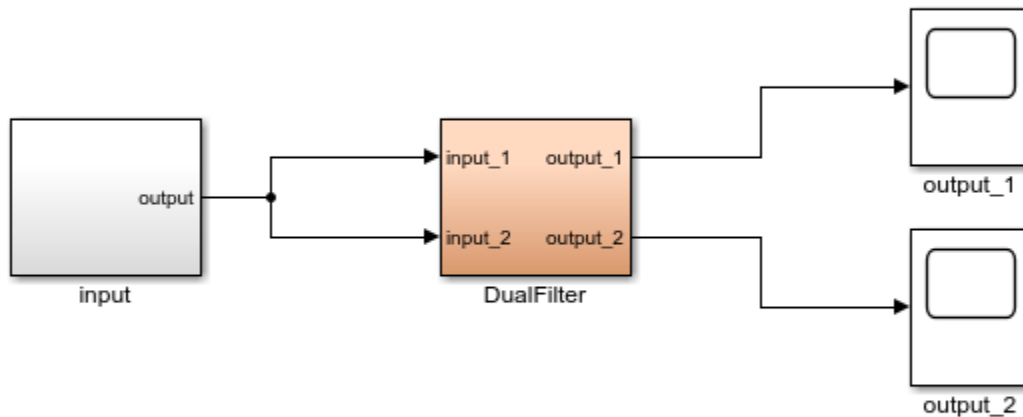
Note: The example includes a code generation build procedure. Simulink does not permit you to build programs in the MATLAB® installation area. If necessary, change to a working directory that is not in the MATLAB installation area prior to starting any build.

1. Open Preconfigured Model

To open the **FIR Filter model with imported IPXact (without memory map)**, click the **Open Model** button.

The following model opens in Simulink.

TLM Generator Imported IPXact Without Memory Map Example



This model shows the generation of a TLM component with an IPXact imported specification without memory map for a dual 4 taps FIR filter.

Copyright 1994-2014 The MathWorks, Inc.

2. Open IP-XACT File

Open the IP-XACT file used by this example in the MATLAB Editor. Alternatively, execute the following command in the MATLAB command window:

```
>> open('tlmgdemo_ipxactnomem.xml');
```

The IP-XACT file opens in the MATLAB editor.

3. Examine the IP-XACT File

With the MATLAB Editor, find the following lines of code (MWMapInput, MWMapOutput). These lines represent the FIR Filter input and output mapping on the Inout TLM socket (represented by a bus interface in IPXact).


```

11 <spirit:busInterface>
12   <spirit:name>InOut</spirit:name>
13   <spirit:busType spirit:version="2.3" spirit:vendor="systemc.org" spirit:name="generic" spirit:library="tlm"/>
14   <spirit:abstractionType spirit:version="2.3" spirit:vendor="systemc.org" spirit:name="generic" spirit:library="tlm"/>
15   <spirit:slave>
16   </spirit:slave>
17   <spirit:endianness>little</spirit:endianness>
18   <spirit:parameters>
19     <spirit:parameter>
20       <spirit:name>MWMMapInput</spirit:name>
21       <spirit:value>input_1</spirit:value>
22     </spirit:parameter>
23     <spirit:parameter>
24       <spirit:name>MWMMapInput</spirit:name>
25       <spirit:value>input_2</spirit:value>
26     </spirit:parameter>
27     <spirit:parameter>
28       <spirit:name>MWMMapOutput</spirit:name>
29       <spirit:value>output_1</spirit:value>
30     </spirit:parameter>
31     <spirit:parameter>
32       <spirit:name>MWMMapOutput</spirit:name>
33       <spirit:value>output_2</spirit:value>
34     </spirit:parameter>
35   </spirit:parameters>
36 </spirit:busInterface>

```

Similarly, the following lines of code (MWMMapParam) represent the FIR Filter parameter mapping on the Config TLM socket.

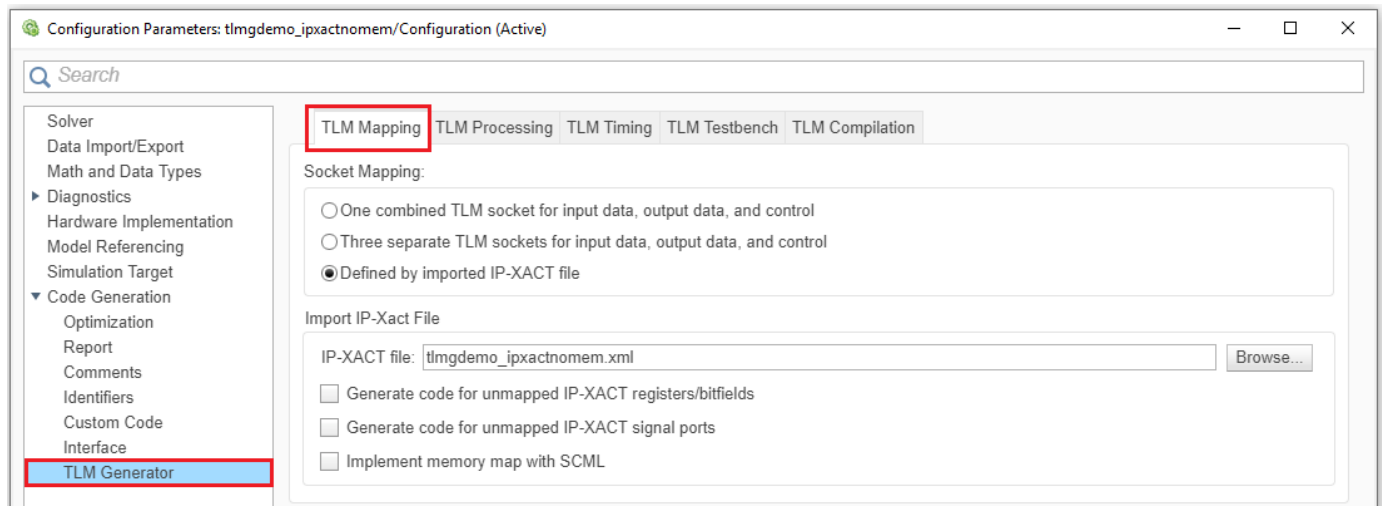
```

37 <spirit:busInterface>
38   <spirit:name>Config</spirit:name>
39   <spirit:busType spirit:version="2.3" spirit:vendor="systemc.org" spirit:name="generic" spirit:library="tlm"/>
40   <spirit:abstractionType spirit:version="2.3" spirit:vendor="systemc.org" spirit:name="generic" spirit:library="tlm"/>
41   <spirit:slave>
42   </spirit:slave>
43   <spirit:endianness>little</spirit:endianness>
44   <spirit:parameters>
45     <spirit:parameter>
46       <spirit:name>MWMMapParam</spirit:name>
47       <spirit:value>coef_11</spirit:value>
48     </spirit:parameter>
49     <spirit:parameter>
50       <spirit:name>MWMMapParam</spirit:name>
51       <spirit:value>coef_12</spirit:value>
52     </spirit:parameter>
53     <spirit:parameter>
54       <spirit:name>MWMMapParam</spirit:name>
55       <spirit:value>coef_13</spirit:value>
56     </spirit:parameter>
57     <spirit:parameter>
58       <spirit:name>MWMMapParam</spirit:name>
59       <spirit:value>coef_14</spirit:value>
60     </spirit:parameter>
61   </spirit:parameters>
62 </spirit:busInterface>

```

4. Set Option to Defined by Imported IP-XACT File

- a. In the **Configuration Parameters** dialog box, select the **TLM Generator** view in the left-hand pane.
- b. In the **TLM Mapping** tab, under **Socket Mapping**, select **Defined by imported IP-XACT file**.
- c. Under **Import IP-XACT File**, browse to the location of the IP-XACT file, as shown in the following image.



5. Set Other Options

Set the remaining TLM Generation options according to your preferences (or leave the default values as they are) and click **OK** to apply these settings and exit the **Configuration Parameters** dialog box.

6. Build the Model

In the model window, right-click on the DualFilter block and select **C/C++ Code > Generate Code for this Subsystem**. This option starts the TLM component generation. Alternatively you can execute the following command in the MATLAB command window:

```
>> slbuild('tlm demo_ipxactnomem/DualFilter');
```

TLM generation is completed when the following message appears in the MATLAB command window:

```
### Starting Simulink Coder build procedure for model: DualFilter
### Successful completion of Simulink Coder build procedure for model: DualFilter
```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
DualFilter	Code generated.	Code generation information file does not exist.

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 20.156s
```

7. Open Generated Files

Open the following file in the MATLAB Editor:

DualFilter_VP/DualFilter_ipxactnomem_tlm/include/DualFilter_ipxactnomem_tlm_def.h

8. Examine Generated Code

Using MATLAB Editor, find the following lines of code. These lines represent the input and output definitions of the FIR Filter when generated as a TLM component with no memory map.

```

8 //===== InOut_IN_BANK BANK =====
9 //===== DATA =====
10 #define input_1_REG_DIM          1
11 #define input_2_REG_DIM          1
12
13 typedef real_T input_1_REG_T;
14 typedef real_T input_2_REG_T;
15
16 //Type of the input data
17 typedef struct {
18     input_1_REG_T input_1_REG;
19     input_2_REG_T input_2_REG;
20 } InOut_IN_BANK_T;

```

To provide a set of inputs to the generated TLM component, a TLM initiator should send a write TLM transaction with no address at the InOut socket and with a payload containing a datum of type **InOut_IN_BANK_T**.

```

22 //===== InOut_OUT_BANK BANK =====
23 //===== DATA =====
24 #define output_1_REG_DIM         1
25 #define output_2_REG_DIM         1
26
27 typedef real_T output_1_REG_T;
28 typedef real_T output_2_REG_T;
29
30 //Type of the output data
31 typedef struct {
32     output_1_REG_T output_1_REG;
33     output_2_REG_T output_2_REG;
34 } InOut_OUT_BANK_T;

```

To obtain a set of outputs from the generated TLM component, a TLM initiator should send a read TLM transaction with no address at the InOut socket and with a payload containing a datum of type **InOut_OUT_BANK_T**.

```
39 //===== DualFilter_Config_PARAM_BANK BANK =====
40 //===== DATA =====
41 #define coef_11_REG_DIM          1
42 #define coef_12_REG_DIM          1
43 #define coef_13_REG_DIM          1
44 #define coef_14_REG_DIM          1
45
46 typedef real_T coef_11_REG_T;
47 typedef real_T coef_12_REG_T;
48 typedef real_T coef_13_REG_T;
49 typedef real_T coef_14_REG_T;
50
51 //Type of the param data
52 typedef struct {
53     coef_11_REG_T coef_11_REG;
54     coef_12_REG_T coef_12_REG;
55     coef_13_REG_T coef_13_REG;
56     coef_14_REG_T coef_14_REG;
57 } DualFilter_Config_PARAM_BANK_T;
```

To modify the generated TLM component tunable parameter, a TLM initiator should send a write TLM transaction with no address at the Config socket and with a payload containing a datum of type **DualFilter_Config_PARAM_BANK_T**.

Imported IP-XACT with Memory Map

This example shows how to use an imported IP-XACT specification with memory map when you generate a SystemC™ or a Transaction Level Modeling (TLM) component from a Simulink® model with the **tlmgenerator** target. You can use the **tlmgenerator** target for either Simulink Coder™ or Embedded Coder™.

In Simulink, each block input or output is bound point-to-point to another block. In SystemC or TLM, each component communicates through a TLM socket. This socket handles all the incoming and outgoing communication formatted as TLM transactions. Because each system handles communication differently, you must define a communication interface for the SystemC or TLM component when it is generated from a Simulink model. Depending on the intended use of this SystemC or TLM component, this communication interface could require building a memory map for each input or output in the component. This memory map may be simple, detailed, or imported from an IP-XACT specification file.

The import IP-XACT option generates a TLM component from a Simulink model according to an IP-XACT specification file. In this example, the IP-XACT file specifies one tlm socket with memory map for tunable parameters. Each parameter is bound to a corresponding read or write register in this tlm socket. Use the generated TLM component in a virtual platform (VP) as a standalone component in a test bench or attach it to a communication channel.

In this example, we use a Simulink model of a FIR filter as the basis for SystemC or TLM generation.

Requirements to run this example:

- SystemC 2.3.1 (includes the TLM library)

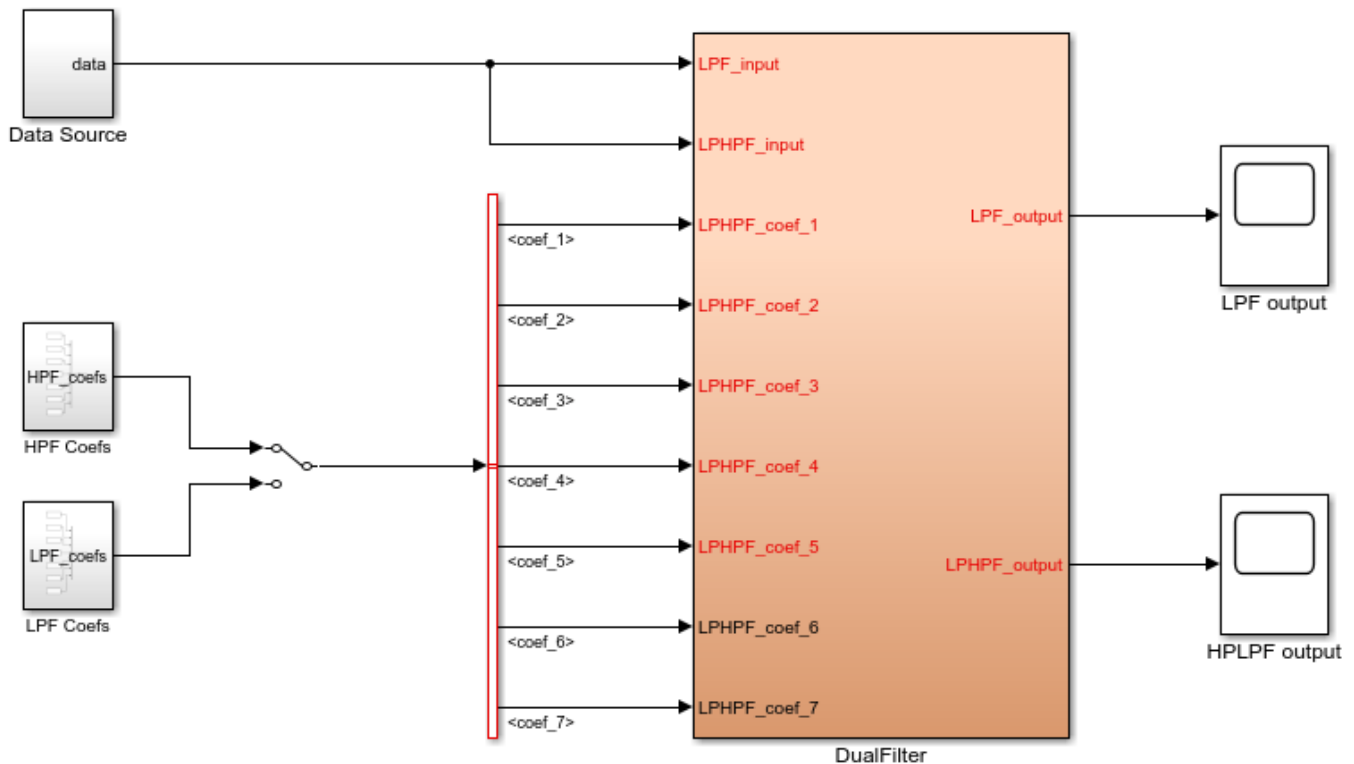
Note: This example includes a code generation build procedure. Simulink does not permit you to build programs in the MATLAB® installation area. If necessary, change to a working directory that is not in the MATLAB installation area prior to starting any build.

1. Open Preconfigured Model

To open the **FIR Filter model with imported IPXact (with memory map)**, click the **Open Model** button.

The following model opens in Simulink.

TLM Generator Imported IPXact With Memory Map Example



Copyright 2005-2019 The MathWorks, Inc.

2. Open IP-XACT File

Open the IP-XACT file used by this example in the MATLAB Editor. Alternatively, execute the following command in the MATLAB command window:

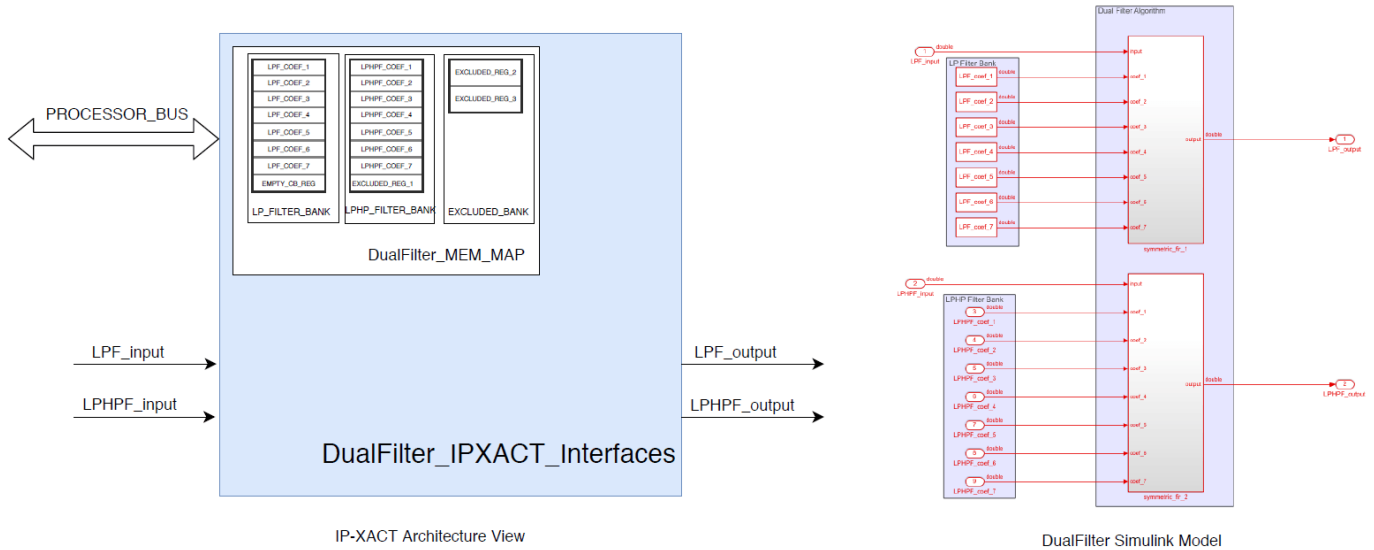
```
>> open('tlmgdemo_ipxactmem.xml');
```

The IP-XACT file opens in the MATLAB editor.

3. Examine the IP-XACT File

The architecture view of this IP-XACT file and the corresponding Simulink model are shown below. This IP-XACT contains a DualFilter component which has two in ports (**LPF_input** and **LPHPF_input**), two out ports (**LPF_output** and **LPHPF_output**), and one bus interface

(**PROCESSOR_BUS**). These in and out ports are mapped to the inputs and outputs of DualFilter model, respectively. The bus interface (**PROCESSOR_BUS**) contains a memory map (**DualFilter_MEM_MAP**). Inside the memory map, there are three address blocks (**LP_FILTER_BANK**, **LPHP_FILTER_BANK**, and **EXCLUDED_BANK**). **LP_FILTER_BANK** contains registers which are mapped to LP filter coefficient parameters in the Simulink model. **LPHP_FILTER_BANK** contains registers which are mapped to LPHP filter coefficient inputs in the Simulink model. **EXCLUDED_BANK** is an additional register bank which does not have a corresponding representation in the Simulink model.



This section (MWMMapInput) indicates that the Simulink model input **LPF_input** is mapped to TLM component in port **LPF_input**.

```

308     <spirit:port>
309         <spirit:name>LPF_input</spirit:name>
310         <spirit:wire>
311             <spirit:direction>in</spirit:direction>
312             <spirit:wireTypeDefs>
313                 <spirit:wireTypeDef>
314                     <spirit:typeName>double</spirit:typeName>
315                     <spirit:typeDefinition>systemc.h</spirit:typeDefinition>
316                 </spirit:wireTypeDef>
317             </spirit:wireTypeDefs>
318         </spirit:wire>
319         <spirit:vendorExtensions>
320             <spirit:parameters>
321                 <spirit:parameter>
322                     <spirit:name>MWMMapInput</spirit:name>
323                     <spirit:value>LPF_input</spirit:value>
324                 </spirit:parameter>
325             </spirit:parameters>
326         </spirit:vendorExtensions>
327     </spirit:port>

```

Similarly, this section (MWMMapOutput) indicates that the Simulink model output **LPF_output** is mapped to TLM component out port **LPF_output**.

```

...
348     <spirit:port>
349         <spirit:name>LPF_output</spirit:name>
350         <spirit:wire>
351             <spirit:direction>out</spirit:direction>
352             <spirit:wireTypeDefs>
353                 <spirit:wireTypeDef>
354                     <spirit:typeName>double</spirit:typeName>
355                     <spirit:typeDefinition>systemc.h</spirit:typeDefinition>
356                 </spirit:wireTypeDef>
357             </spirit:wireTypeDefs>
358         </spirit:wire>
359         <spirit:vendorExtensions>
360             <spirit:parameters>
361                 <spirit:parameter>
362                     <spirit:name>MWMMapOutput</spirit:name>
363                     <spirit:value>LPF_output</spirit:value>
364                 </spirit:parameter>
365             </spirit:parameters>
366         </spirit:vendorExtensions>
367     </spirit:port>

```


This section (MWMMap) indicates that the bus interface **PROCESSOR_BUS** is mapped to a TLM socket. This bus interface contains a memory map named **DualFilter_MEM_MAP**.

```

10 <spirit:busInterfaces>
11   <spirit:busInterface>
12     <spirit:name>PROCESSOR_BUS</spirit:name>
13     <spirit:busType spirit:version="2.3" spirit:vendor="systemc.org" spirit:name="generic" spirit:library="tlm"/>
14     <spirit:abstractionType spirit:version="2.3" spirit:vendor="systemc.org" spirit:name="generic" spirit:library="tlm"/>
15     <spirit:slave>
16       <spirit:memoryMapRef spirit:memoryMapRef="DualFilter_MEM_MAP"/>
17     </spirit:slave>
18     <spirit:endianness>little</spirit:endianness>
19     <spirit:bitsInLau>8</spirit:bitsInLau>
20     <spirit:bitSteering>off</spirit:bitSteering>
21     <spirit:parameters>
22       <spirit:parameter>
23         <spirit:name>MWMMap</spirit:name>
24         <spirit:value>true</spirit:value>
25       </spirit:parameter>
26     </spirit:parameters>
27   </spirit:busInterface>
28 </spirit:busInterfaces>

```

This section (MWMMapParam) represents the mapping of the LP filter parameter **LPF_coef_1** to register **LPF_COEF_1**.

```

38   <spirit:register>
39     <spirit:name>LPF_COEF_1</spirit:name>
40     <spirit:addressOffset>0x00</spirit:addressOffset>
41     <spirit:size>64</spirit:size>
42     <spirit:reset>
43       <spirit:value>0x00</spirit:value>
44     </spirit:reset>
45     <spirit:parameters>
46       <spirit:parameter>
47         <spirit:name>MWMMapParam</spirit:name>
48         <spirit:value>LPF_coef_1</spirit:value>
49       </spirit:parameter>
50     </spirit:parameters>
51   </spirit:register>

```

This section (MWMMapInput) represents the LPHP filter input **coef_1** mapping to register **LPHPF_COEF_1**.

```

151     <spirit:usage>register</spirit:usage>
152     <spirit:register>
153         <spirit:name>LPHPF_COEF_1</spirit:name>
154         <spirit:addressOffset>0x00</spirit:addressOffset>
155         <spirit:size>64</spirit:size>
156         <spirit:reset>
157             <spirit:value>0x00</spirit:value>
158         </spirit:reset>
159         <spirit:access>write-only</spirit:access>
160         <spirit:parameters>
161             <spirit:parameter>
162                 <spirit:name>MWMMapInput</spirit:name>
163                 <spirit:value>coef_1</spirit:value>
164             </spirit:parameter>
165         </spirit:parameters>
166     </spirit:register>

```

If you do not want to generate code for a specific address block or register from the memory map, set MWMMap to false for the address block or register. This section (MWMMap) shows how to exclude an address block named **EXCLUDED_BANK** from TLM code generation

```

273 <spirit:addressBlock>
274     <spirit:name>EXCLUDED_BANK</spirit:name>
275     <spirit:baseAddress spirit:id="EXCLUDED_BANK_ADDR" spirit:resolve="user">0x00030000</spirit:baseAddress>
276     <spirit:range>16</spirit:range>
277     <spirit:width>64</spirit:width>
278     <spirit:usage>register</spirit:usage>
279     <spirit:register>
280         <spirit:name>EXCLUDED_REG_2</spirit:name>
281         <spirit:addressOffset>0x00</spirit:addressOffset>
282         <spirit:size>64</spirit:size>
283         <spirit:access>read-only</spirit:access>
284         <spirit:reset>
285             <spirit:value>0x00</spirit:value>
286         </spirit:reset>
287     </spirit:register>
288     <spirit:register>
289         <spirit:name>EXCLUDED_REG_3</spirit:name>
290         <spirit:addressOffset>0x08</spirit:addressOffset>
291         <spirit:size>64</spirit:size>
292         <spirit:access>read-only</spirit:access>
293         <spirit:reset>
294             <spirit:value>0x00</spirit:value>
295         </spirit:reset>
296     </spirit:register>
297     <spirit:parameters>
298         <spirit:parameter>
299             <spirit:name>MWMMap</spirit:name>
300             <spirit:value>>false</spirit:value>
301         </spirit:parameter>
302     </spirit:parameters>
303 </spirit:addressBlock>

```

Similarly, this section (MWMMap) shows how to exclude a register named **EXCLUDED_REG_1** from TLM code generation

```

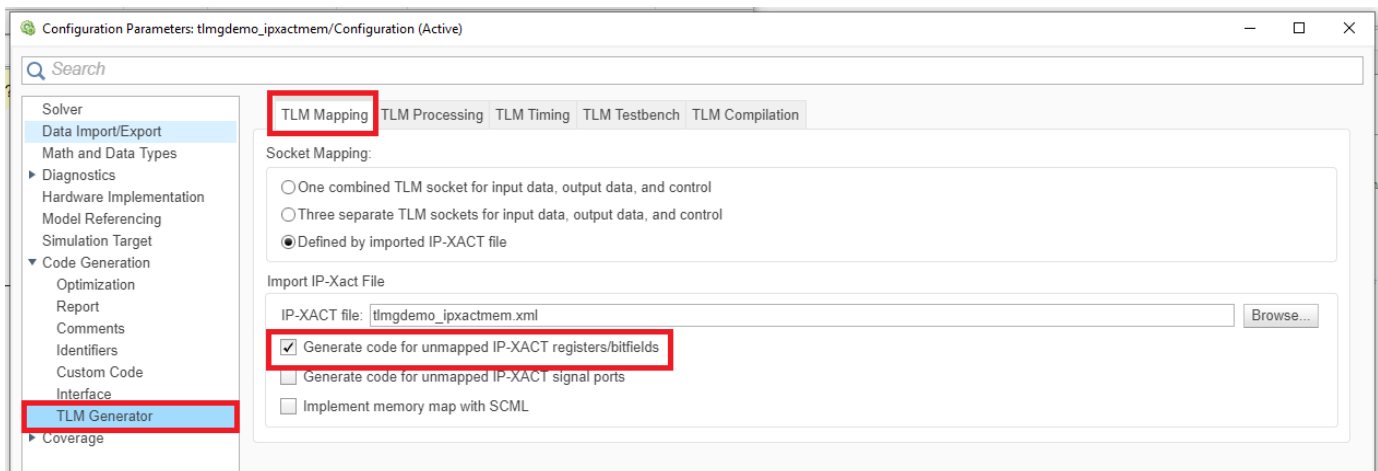
257     <spirit:register>
258         <spirit:name>EXCLUDED_REG_1</spirit:name>
259         <spirit:addressOffset>0x38</spirit:addressOffset>
260         <spirit:size>64</spirit:size>
261         <spirit:access>read-only</spirit:access>
262         <spirit:reset>
263             <spirit:value>0x00</spirit:value>
264         </spirit:reset>
265         <spirit:parameters>
266             <spirit:parameter>
267                 <spirit:name>MWMMap</spirit:name>
268                 <spirit:value>>false</spirit:value>
269             </spirit:parameter>
270         </spirit:parameters>
271     </spirit:register>

```

4. Set Configuration Parameter Options

To configure this model for TLM-generation, set the following options in the configuration parameters.

- a. In the **Configuration Parameters** dialog box, select the **TLM Generator** view in the left-hand pane.
- b. In the **TLM Mapping** tab, under **Socket Mapping**, select **Defined by imported IP-XACT file**.
- c. Under **Import IP-XACT File**, browse to the location of the IP-XACT file, as shown in the following image.
- d. (Optional) If you want to generate code for registers/bitfields which do not have corresponding representation in the Simulink model, under **Import IP-XACT File**, select **Generate code for unmapped IP-XACT registers/bitfields**.



5. Build the Model

In the model window, right-click on the DualFilter block and select **C/C++ Code > Build This Subsystem**. In the popup window click **Build**. This option starts the TLM component generation. Alternatively you can execute the following command in the MATLAB command window:

```
>> slbuild('tlmgdemo_ipxactmem/DualFilter');
```

TLM generation is completed when the following message appears in the MATLAB command window:

```
### Starting Simulink Coder build procedure for model: DualFilter
### Successful completion of Simulink Coder build procedure for model: DualFilter
```

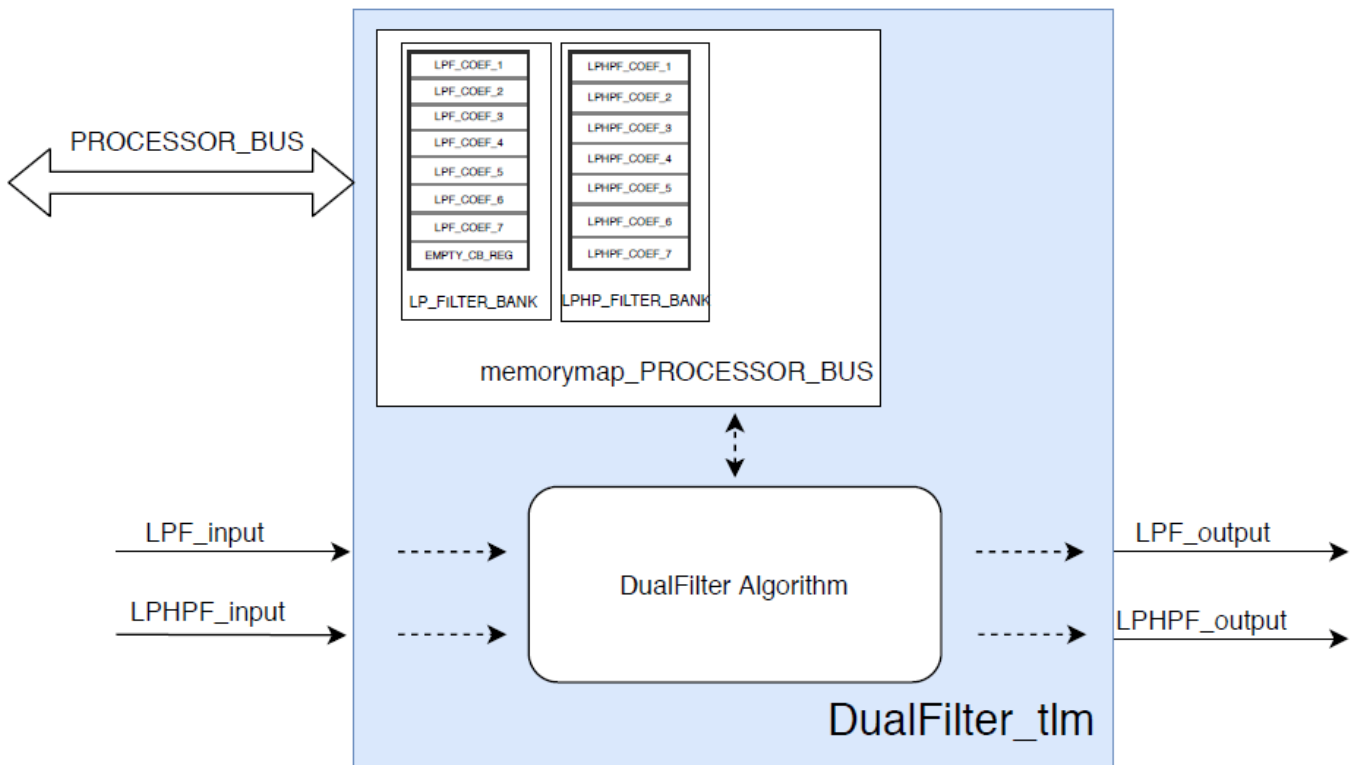
Build Summary

Top model targets built:

Model	Action	Rebuild Reason
DualFilter	Code generated.	Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 21.69s

The architecture view of generated TLM component is shown below:



6. Examine Generated Code

Open the following file in the MATLAB Editor:

DualFilter_VP/DualFilter_ipxactmem_tlm/include/DualFilter_ipxactmem_tlm.h

This section represents the mapping of the inputs and outputs of the FIR filter to the generated TLM component in and out ports, respectively.

```
24 DualFilter_ipxactmem_tlm( sc_core::sc_module_name module_name, eTimingType
25     DefaultTiming = TIMED, eModeType InputDefaultMode = AUTO, eModeType
26     OutputDefaultMode = AUTO);
27 ~DualFilter_ipxactmem_tlm(void);
28 tlm_utils::simple_target_socket<DualFilter_ipxactmem_tlm,
29     PROCESSOR_BUS BUSWIDTH> PROCESSOR_BUS;
30 sc_in<double> LPF_input;
31 sc_in<double> LPHPF_input;
32 sc_out<double> LPF_output;
33 sc_out<double> LPHPF_output;
```

Open the following file in the MATLAB Editor:

DualFilter_VP/DualFilter_ipxactmem_tlm/include/DualFilter_ipxactmem_tlm_def.h

This section shows the address map of the LP filter and LPHP filter coefficient registers, as specified in the IP-XACT file.

```

8 //===== LP_FILTER_BANK BANK =====
9 //Bank base address
10 #define LP_FILTER_BANK_ADDR          0x00010000
11
12 //===== REGISTERS =====
13 //Address of the registers
14 #define LPF_COEF_1_ADDR              0x00000000
15 #define LPF_COEF_2_ADDR              0x00000008
16 #define LPF_COEF_3_ADDR              0x00000010
17 #define LPF_COEF_4_ADDR              0x00000018
18 #define LPF_COEF_5_ADDR              0x00000020
19 #define LPF_COEF_6_ADDR              0x00000028
20 #define LPF_COEF_7_ADDR              0x00000030
21 #define EMPTY_CB_REG_ADDR           0x00000038
    .
    .
    .
42 //===== LPHP_FILTER_BANK BANK =====
43 //Bank base address
44 #define LPHP_FILTER_BANK_ADDR        0x00020000
45
46 //===== REGISTERS =====
47 //Address of the registers
48 #define LPHPF_COEF_1_ADDR            0x00000000
49 #define LPHPF_COEF_2_ADDR            0x00000008
50 #define LPHPF_COEF_3_ADDR            0x00000010
51 #define LPHPF_COEF_4_ADDR            0x00000018
52 #define LPHPF_COEF_5_ADDR            0x00000020
53 #define LPHPF_COEF_6_ADDR            0x00000028
54 #define LPHPF_COEF_7_ADDR            0x00000030

```

To modify the generated TLM register **LPF_COEF_1**, a TLM initiator should send a TLM transaction at the address **LP_FILTER_BANK_ADDR + LPF_COEF_1_ADDR**.

7. Conclusion

This example shows how to generate the TLM component from the Simulink model based on an imported IP-XACT specification. This IP-XACT includes mapping a Simulink port to a TLM port, mapping a Simulink parameter to a TLM register, and mapping a Simulink input to a TLM register. This example also demonstrates how to exclude an address block (or register) from the memory map.

Verify 5G Wireless Applications Using SystemVerilog DPI

This example shows how to use SystemVerilog DPI components to verify 5G wireless applications in an HDL environment.

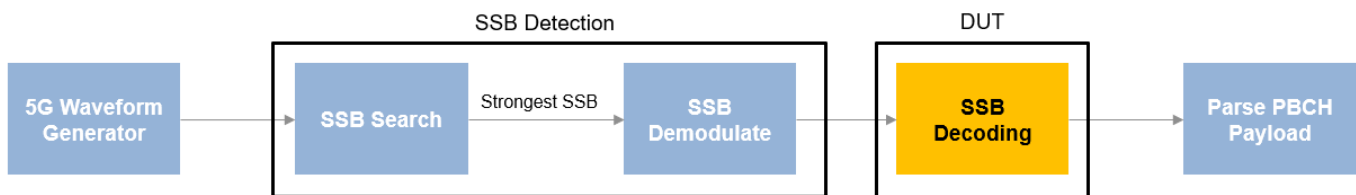
The system in this example uses various 5G components and a parameterizable 5G waveform generator to validate the behavior of the Synchronization Signal Block (SSB) decoding section of the Master Information Block (MIB) recovery process.

The verification workflow includes these key benefits:

- 3GPP 5G New Radio (NR) standard requires deep domain expertise. Creating a standard-compliant waveform verification model can be challenging. Generating a DPI component from a Wireless HDL Toolbox™ waveform generator simplifies the test bench design process by automatically creating a standard compliant verification IP.
- The parameterizable 5G waveform generator tests the DUT in different scenarios. You can reconfigure the parameters to create a series of test cases to meet coverage.
- The standalone 5G DPI components generated from Simulink® and MATLAB® can be reused and integrated in customized test benches.
- The component-based workflow makes designing a standalone test bench faster. You can splice different modules in the top-level test bench to test different 5G function components.
- Full functional control at the top level test bench enables component manipulation according to the process status changes. This control results in performance gains compared to a test-vector-based HDL test bench.

MIB Recovery Process

MIB recovery requires SSB detection, demodulation, and decoding. This example shows how to validate the HDL code generated by HDL Coder™ for the SSB decoding module.



SSB detection performs a primary synchronization sequence (PSS) search, orthogonal frequency division multiplexing (OFDM) demodulation, and a secondary synchronization sequence (SSS) search. SSB detection has two modes of operation: search and demodulation. In search mode, the detection searches for SSBs and returns their parameters. In demodulation mode, the detection recovers a specified SSB, OFDM-demodulates its resource grid, and searches for the SSS within the appropriate resource elements. The details of SSB detection and demodulation are described in the “NR HDL Cell Search” (Wireless HDL Toolbox) example.

SSB decoding performs a demodulation reference signal (DMRS) search, channel estimation and phase equalization, and broadcast channel (BCH) decoding steps. The details of SSB decoding are described in the “NR HDL MIB Recovery” (Wireless HDL Toolbox) example.

5G waveform generator uses 5G Toolbox™ functions to generate a test waveform, which is then applied to the SSB detection in search mode. After the strongest SSB is determined, the test

waveform is applied to the SSB detection in demodulation mode to recover a specified SSB resource grid and search for the SSS within the appropriate resource element.

After an SSB is detected and demodulated, it needs to be decoded to extract the MIB content. When SSB decoding has the demodulated grid, the SSB decoding module decodes the SSB and output the PBCH payload, which is then parsed to extract the MIB data.

File Structure

This example uses these files.

Simulink models

- `nrhdLSSBDetection.slx`: This Simulink model uses the `nrhdLSSBDetectionFR1Core` model reference to simulate the behavior of the SSB detection part of the MIB recovery process.
- `nrhdLSSBDetectionFR1Core.slx`: This model reference implements the SSB detection algorithm.
- `nrhdLSSBDecoding.slx`: This Simulink model uses the `nrhdLSSBDecodingCore` model reference to simulate the behavior of the SSB decoding part of the MIB recovery process.
- `nrhdLSSBDecodingCore.slx`: This model reference implements the SSB decoding algorithm.

Simulink data dictionary

- `nrhdLReceiverData.sldd`: This Simulink data dictionary contains bus objects that define the buses contained in the example models.

MATLAB code

- `generate5GWaveform.m`: This function is a modified version of the Wireless HDL Toolbox™ 5G waveform generator, which is C code generation compatible.
- `runSSBDetectionModelSearch.m`: This script executes and verifies the `nrhdLSSBDetection` model in search mode.
- `runSSBDecodingModel.m`: This script uses the MATLAB reference to implement the cell search algorithm and then runs the `nrhdLSSBDecoding` Simulink model. The script verifies the operation of the model using 5G Toolbox and the MATLAB reference code.
- `nrsvdpiexamples`: This package contains the MATLAB reference code and utility functions for verifying the implementation models.

Pregenerated HDL test bench components (available for Windows® only)

- `5GNRCellDecodeDPITB`: This folder contains generated DPI components, HDL code of the decoding module, and the top-level test bench with associated build and simulation scripts.

Set Up for HDL Simulation

This section describes the workflow for generating the DPI component for each 5G functional component and HDL code for the SSB decoding component. The provided top-level test bench instantiates all of the generated components to validate the behavior of the HDL code that is generated from the SSB decoding block. To enable reuse of individual SystemVerilog DPI components and use a subset of the components in a test bench, generate SystemVerilog DPI for each 5G component individually.

The `5GNRCellDecodeDPITB` folder contains all of the necessary generated components. If you do not want to regenerate these components, skip this section.

5G Waveform Generator

This function uses 5G Toolbox functions to generate a test waveform. The 5G waveform generator has three input arguments: `ncellid`, `SNR`, and `frequencyOffset`. When you use this waveform generator in a SystemVerilog test bench, you can test different scenarios by providing different values for `SNR`, `frequencyOffset`, and `ncellid` without changing the component code. For this example, use this command to generate the DPI component from the MATLAB function `generate5GWaveform`.

```
dpigen generate5GWaveform -args {0,0,0} -PortsDataType LogicVector
```

The `-args {0,0,0}` parameter indicates that three scalar inputs of type `double`. The `-PortsDataType LogicVector` parameter indicates generating a logic vector type interface for the port.

SSB Detection

This component is introduced in the `nrhdlSSBDetection` Simulink model and the top-level test bench uses this component for the SSB search and SSB demodulation. Use these commands to run a search mode simulation and verify the results in MATLAB.

```
clear all;
runSSBDetectionModelSearch;
```

Then use this command to generate a DPI component for the Simulink subsystem `nrhdlSSBDetection/SSB Detection`.

```
rtwbuild('nrhdlSSBDetection/SSB Detection');
```

Choose Strongest PSS

This component is introduced in the `nrhdlSSBDetection` Simulink model and the top-level test bench uses this component to determine the strongest PSS from the PSSs detected by the SSB search. Use this command to generate a DPI component for the Simulink subsystem `nrhdlSSBDetection/chooseStrongestPSS`.

```
rtwbuild('nrhdlSSBDetection/chooseStrongestPSS');
```

SSB Decoding

This component is introduced in the `nrhdlSSBDecoding` Simulink model and is the DUT in this example. Use these commands to run an SSB decoding simulation in MATLAB.

```
clear all;
runSSBDecodingModel;
```

Then, use this command to generate HDL code from this component.

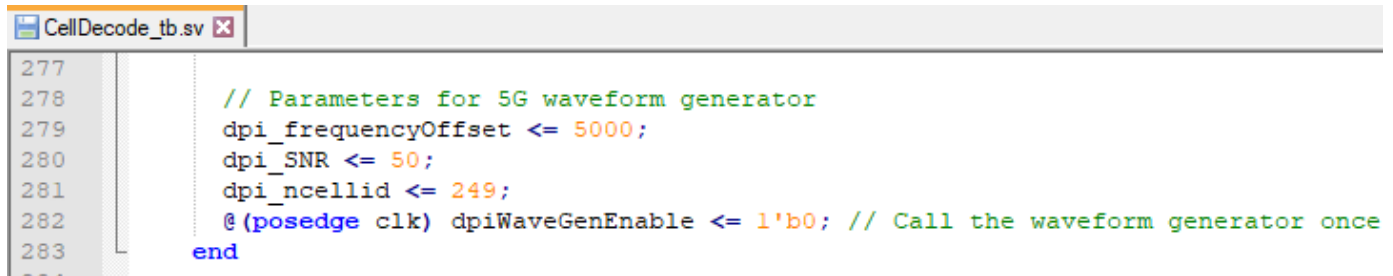
```
makehdl('nrhdlSSBDecoding/SSB Decoding', 'TargetLanguage', 'Verilog');
```

Parse PBCH Payload

This component is introduced in the `nrhdlSSBDecoding` Simulink model and the top-level test bench uses this component to parse the PBCH payload to obtain the MIB information. Use this command to generate the DPI component for the Simulink subsystem `nrhdlSSBDecoding/parsePBCHPayload`.

```
rtwbuild('nrhd\SSBDecoding\parsePBCHPayload');
```

The 5GNRCellDecodeDPITB folder contains a top-level test bench, `CellDecode_tb.sv`, to simulate the entire process described in the MIB Recovery Process section. In this example, the `ncellid` parameter is set to 249, the `SNR` parameter is set to 50, and the `frequencyOffset` parameter is set to 5000. You can modify the values of these parameters to test the design in different scenarios.



```

277
278 // Parameters for 5G waveform generator
279 dpi_frequencyOffset <= 5000;
280 dpi_SNR <= 50;
281 dpi_ncellid <= 249;
282 @(posedge clk) dpiWaveGenEnable <= 1'b0; // Call the waveform generator once
283 end

```

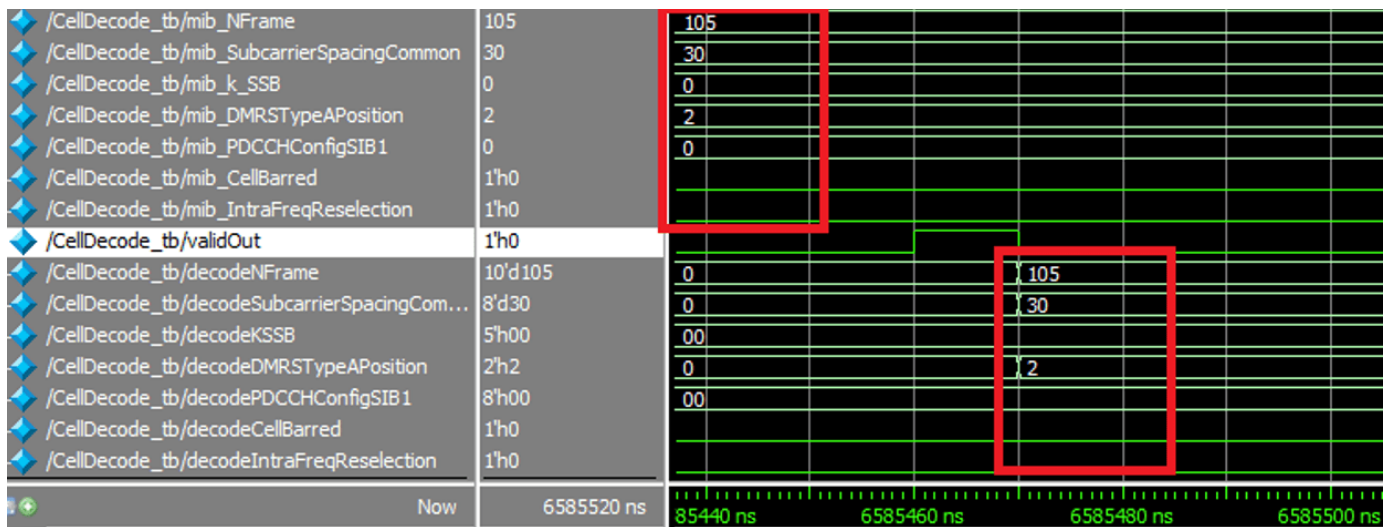
Run Test Bench

Add the QuestaSim simulator to the MATLAB system path, and then navigate to the 5GNRCellDecodeDPITB folder. To compile and simulate the DUT in QuestaSim, enter these commands at the MATLAB prompt.

```
!vsim < compile_dut.do
!vsim < sim_5G_waveform.do
```

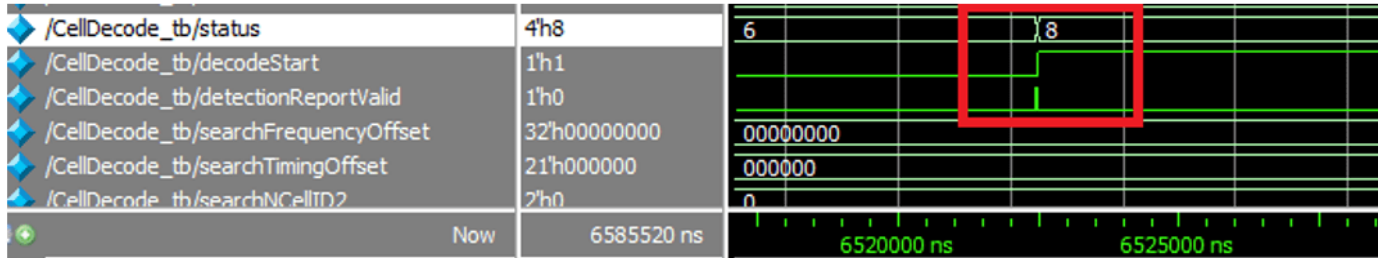
Observe the following simulation results:

- In this figure of a waveform, signal names that start with "mib" carry MIB information from the waveform generator, and signal names that start with "decode" carry MIB information from the decode process. The waveform shows that the decoded MIB information matches the MIB information from the waveform generator.

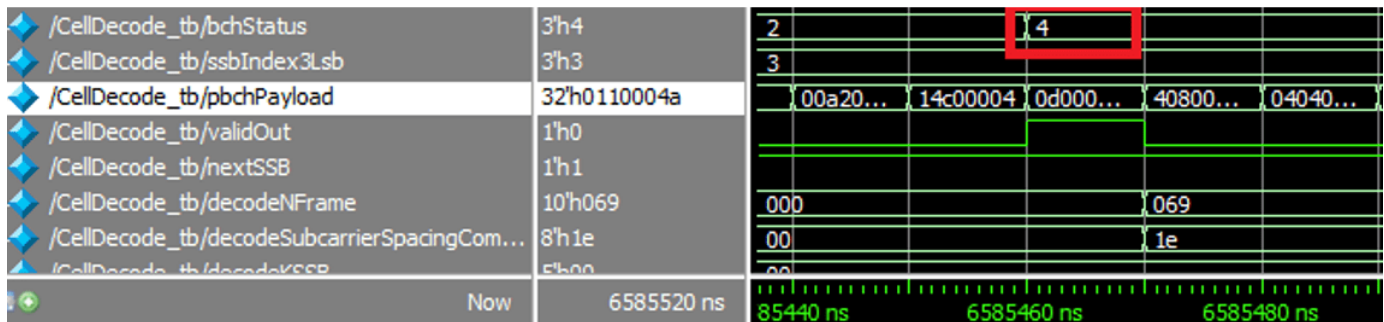


Using this approach, you can dynamically react to the status of each 5G functional component to save simulation time compared to a vector-based test bench.

- In this figure of a waveform, the value of the detection status changes from 6 to 8, indicating that the demodulation operation is complete. The SSS is found, and a demodulated resource grid is returned. In this case, you can start the SSB decoding process instead of waiting for SSB demodulation to finish processing the input vectors if you are using a vector-based HDL test bench.



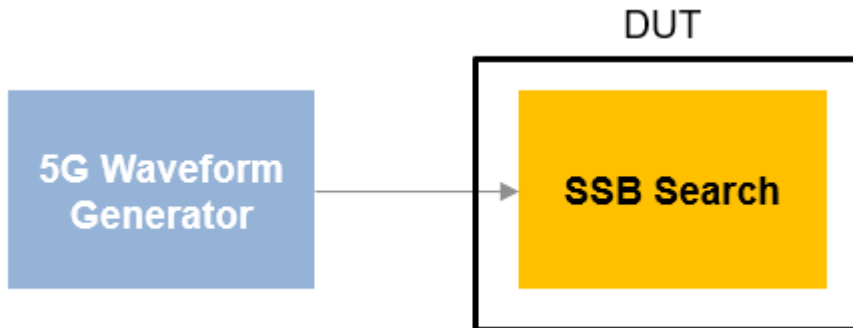
- In this figure of a waveform, the value of the decoding status changes from 2 to 4, indicating that the MIB is detected. In this case, you can stop the simulation rather than finish processing the grid resource data. In contrast, the vector-based test bench approach requires simulation for a fixed amount of time before analyzing the results.



Reuse DPI Component

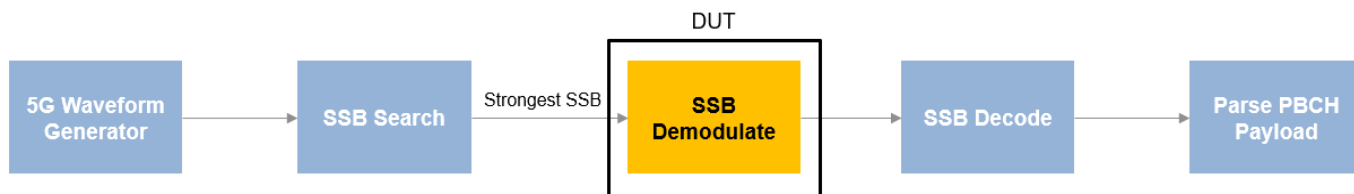
You can reuse the generated 5G function DPI components in a customized test bench.

- Use these components to test a subset of the MIB recover process. For example, reuse the 5G waveform generator to validate the behavior of an SSB search module.



- Use these components to test different 5G function components in the MIB recovery process. For example, when you are validating the behavior of SSB demodulation in the MIB recovery process, you can reuse the 5G waveform generator, SSB search and parse PBCH payload DPI components,

and generate a DPI component from SSB decoding. Instantiate these components in your top-level test bench to validate the behavior of the SSB demodulate module.



Conclusion

This example shows how to use a standalone test bench with DPI components to validate the SSB decoding module of an MIB recovery process. The HDL Verifier™ generated DPI components support tunable parameters, which enable customization of the 5G test waveform from the top-level test bench. You can reuse each generated DPI component in other custom HDL test benches. You can use this workflow for verifying the HDL IP in your wireless application.

Related Topics

- “NR HDL Cell Search” (Wireless HDL Toolbox)
- “NR HDL MIB Recovery” (Wireless HDL Toolbox)